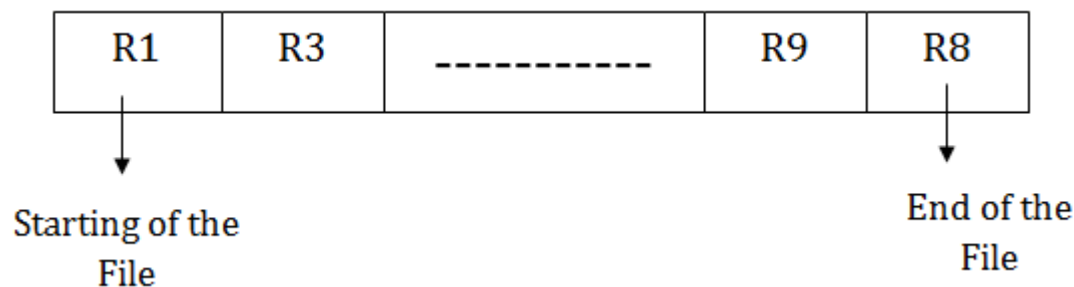


Sequential File Organization

This method is the easiest method for file organization. In this method, files are stored sequentially. This method can be implemented in two ways:

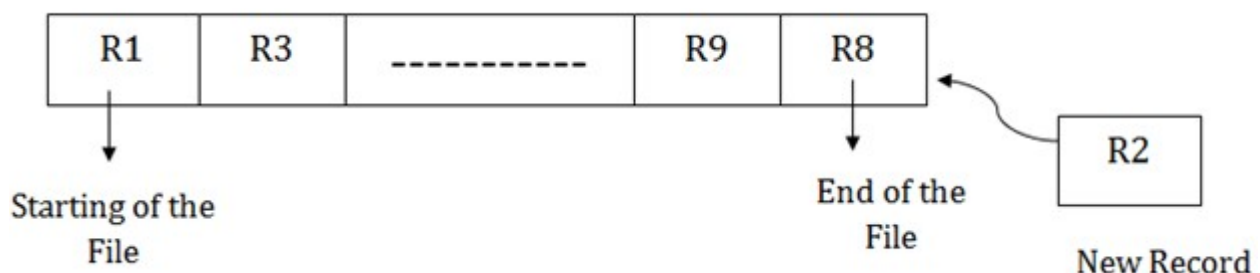
1. Pile File Method:

- It is a quite simple method. In this method, we store the record in a sequence, i.e., one after another. Here, the record will be inserted in the order in which they are inserted into tables.
- In case of updating or deleting of any record, the record will be searched in the memory blocks. When it is found, then it will be marked for deleting, and the new record is inserted.



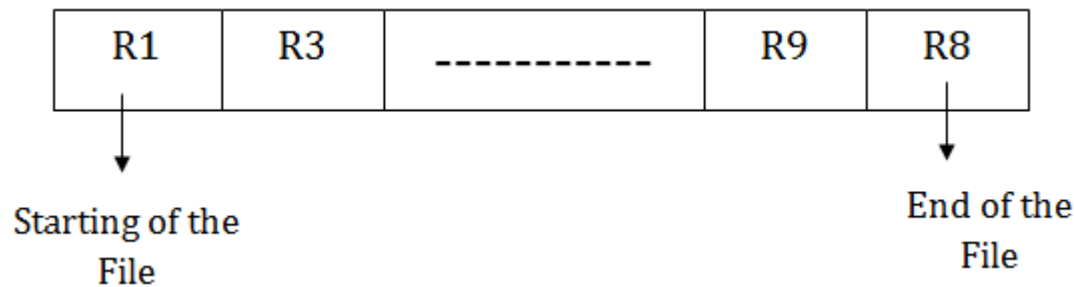
Insertion of the new record:

Suppose we have four records R1, R3 and so on upto R9 and R8 in a sequence. Hence, records are nothing but a row in the table. Suppose we want to insert a new record R2 in the sequence, then it will be placed at the end of the file. Here, records are nothing but a row in any table.



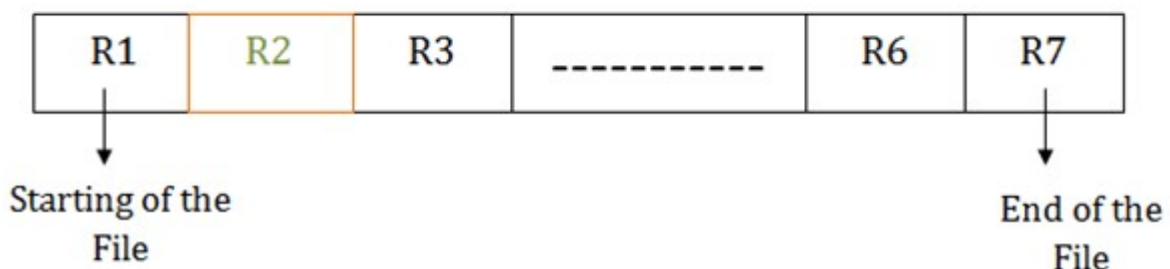
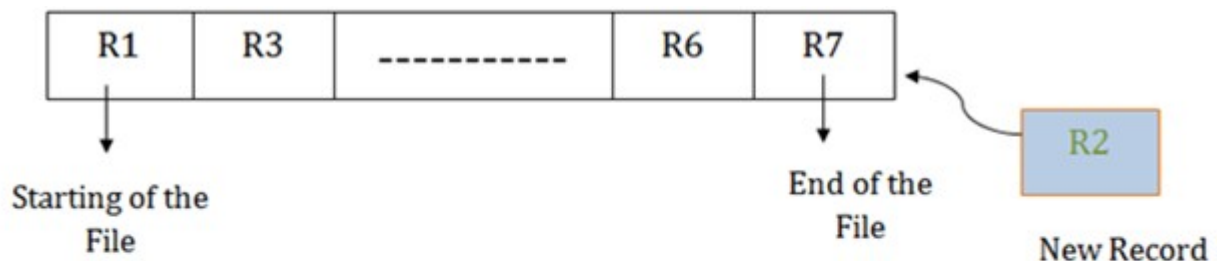
2. Sorted File Method:

- In this method, the new record is always inserted at the file's end, and then it will sort the sequence in ascending or descending order. Sorting of records is based on any primary key or any other key.
- In the case of modification of any record, it will update the record and then sort the file, and lastly, the updated record is placed in the right place.



Insertion of the new record:

Suppose there is a preexisting sorted sequence of four records R1, R3 and so on upto R6 and R7. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file, and then it will sort the sequence.



Pros of sequential file organization

- It contains a fast and efficient method for the huge amount of data.

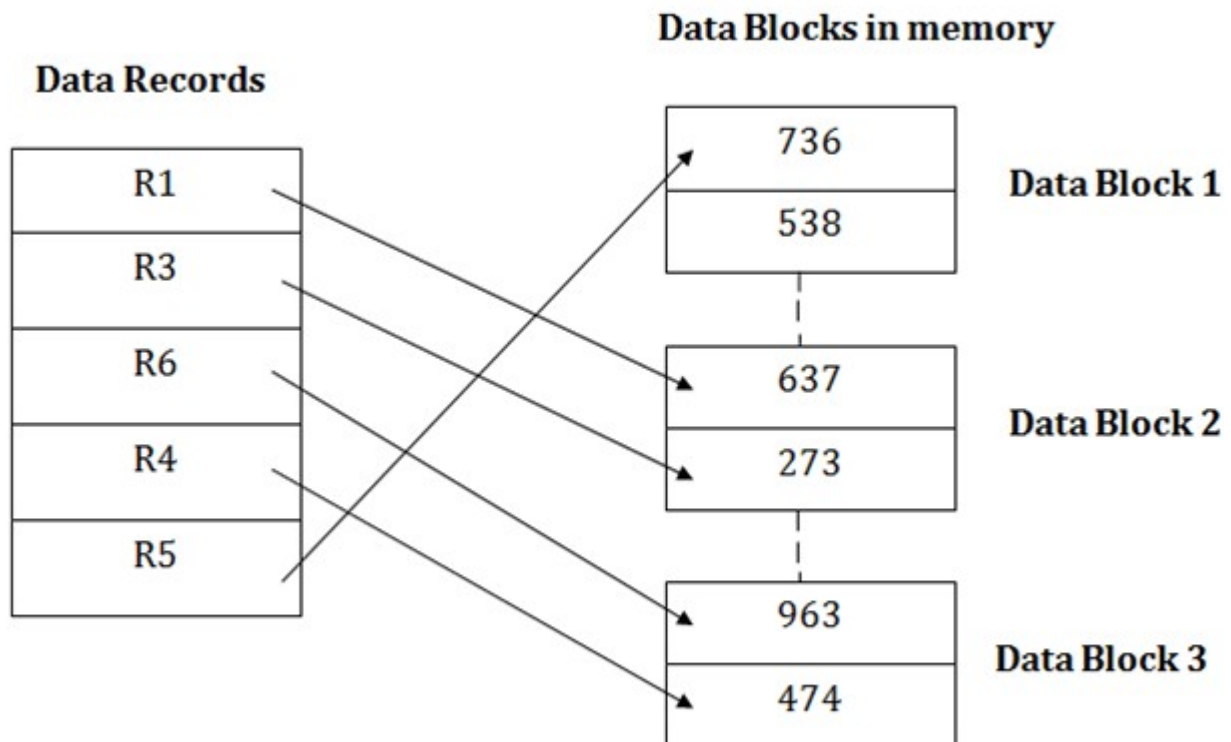
- In this method, files can be easily stored in cheaper storage mechanism like magnetic tapes.
 - It is simple in design. It requires no much effort to store the data.
 - This method is used when most of the records have to be accessed like grade calculation of a student, generating the salary slip, etc.
 - This method is used for report generation or statistical calculations.
-

Cons of sequential file organization

- It will waste time as we cannot jump on a particular record that is required but we have to move sequentially which takes our time.
 - Sorted file method takes more time and space for sorting the records.
-

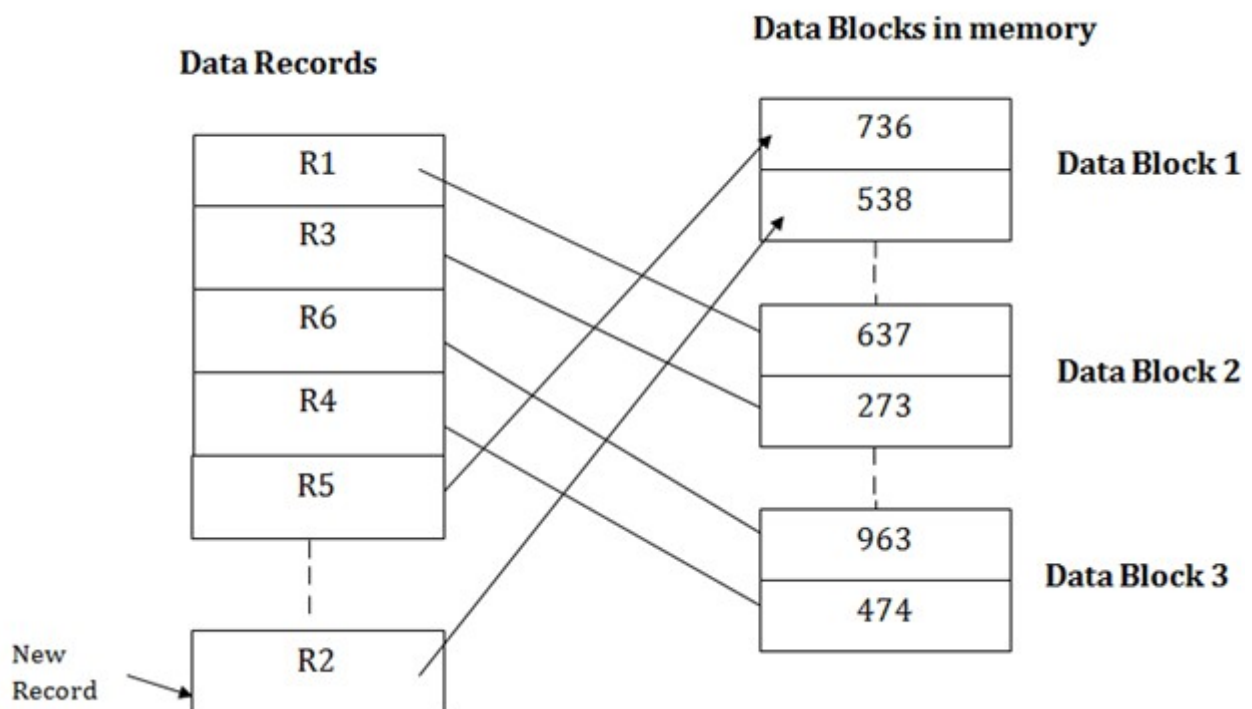
Heap file organization

- It is the simplest and most basic type of organization. It works with data blocks. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.
 - When the data block is full, the new record is stored in some other block. This new data block need not to be the very next data block, but it can select any data block in the memory to store new records. The heap file is also known as an unordered file.
 - In the file, every record has a unique id, and every page in a file is of the same size. It is the DBMS responsibility to store and manage the new records.
-



Insertion of a new record

Suppose we have five records R1, R3, R6, R4 and R5 in a heap and suppose we want to insert a new record R2 in a heap. If the data block 3 is full then it will be inserted in any of the database selected by the DBMS, let's say data block 1.



If we want to search, update or delete the data in heap file organization, then we need to traverse the data from starting of the file till we get the requested record.

If the database is very large then searching, updating or deleting of record will be time-consuming because there is no sorting or ordering of records. In the heap file organization, we need to check all the data until we get the requested record.

Pros of Heap file organization

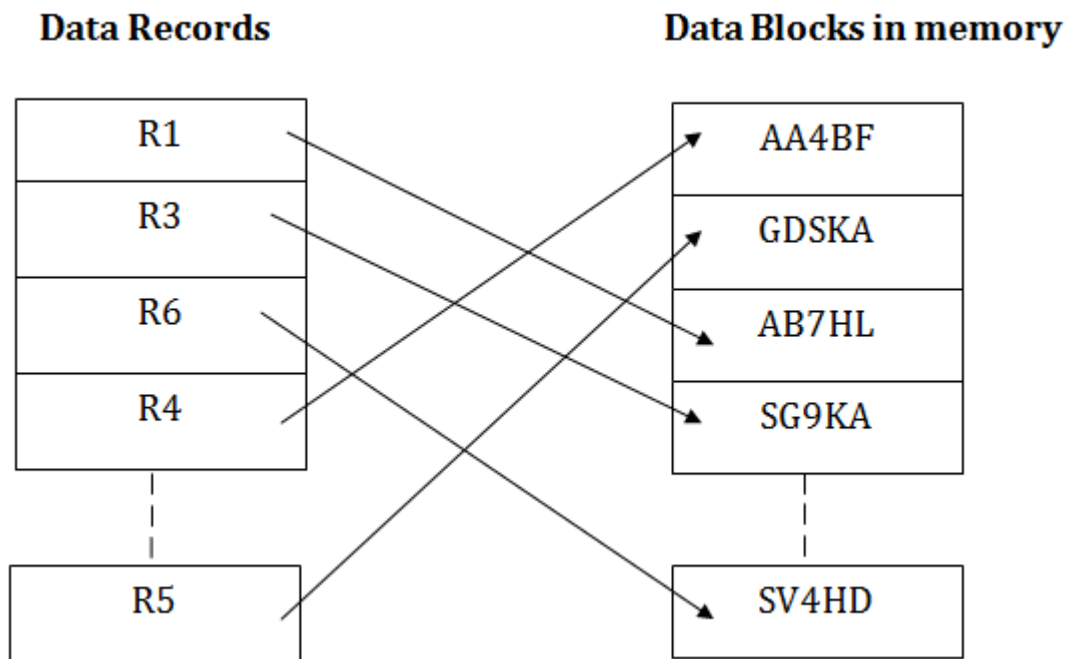
- It is a very good method of file organization for bulk insertion. If there is a large number of data which needs to load into the database at a time, then this method is best suited.
 - In case of a small database, fetching and retrieving of records is faster than the sequential record.
-

Cons of Heap file organization

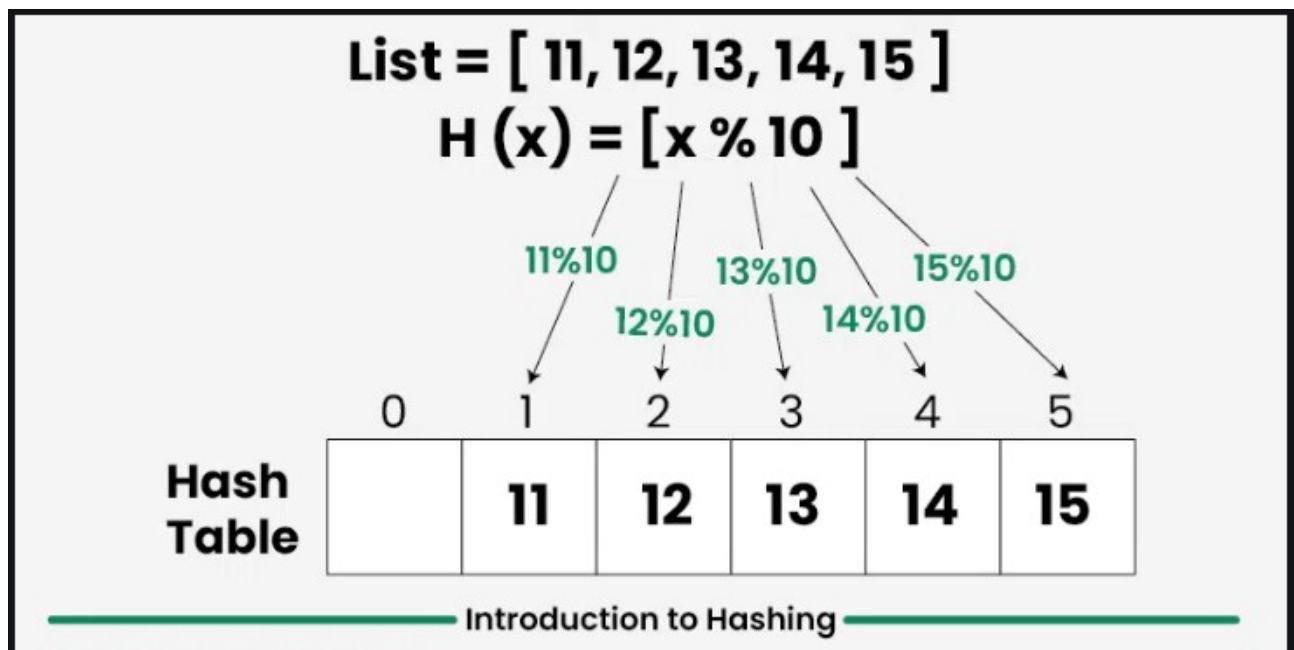
- This method is inefficient for the large database because it takes time to search or modify the record.
 - This method is inefficient for large databases.
-

What is meant by Hash file organization?

It is one of the types of **file organization**. Hash File Organization uses the computation of hash function on some fields of the records. The hash function's output determines the location of disk block where the records are to be placed.



When a record has to be received using the hash key columns, then the address is generated, and the whole record is retrieved using that address. In the same way, when a new record has to be inserted, then the address is generated using the hash key and record is directly inserted. The same process is applied in the case of delete and update



Need for Hash data structure

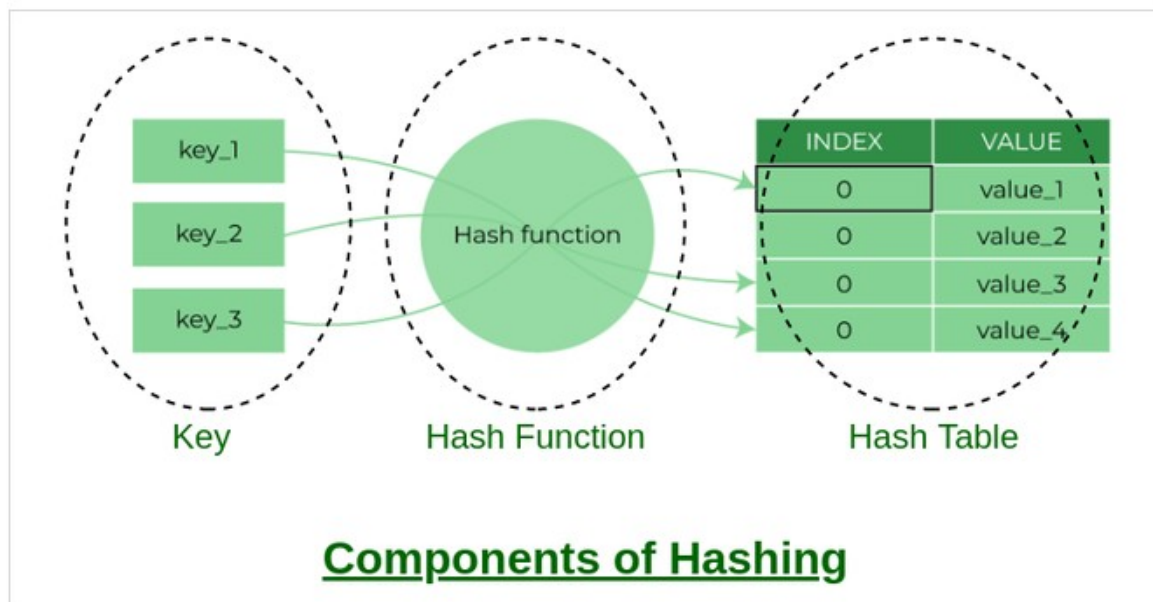
The amount of data on the internet is growing exponentially every day, making it difficult to store it all effectively. In day-to-day programming, this amount of data might not be that big, but still, it needs to be stored, accessed, and processed easily and efficiently. A

very common data structure that is used for such a purpose is the Array data structure.

Components of Hashing

There are majorly three components of hashing:

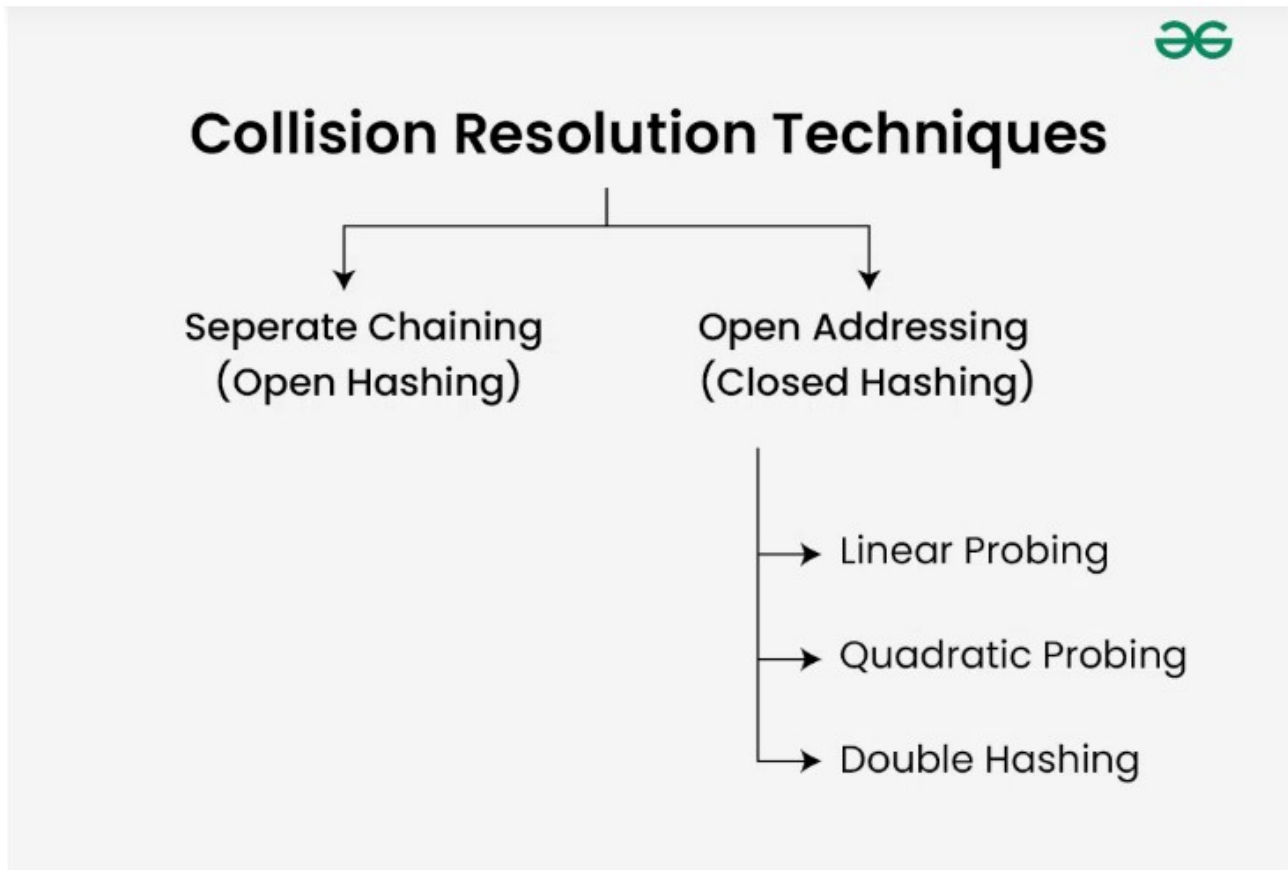
- 1.**Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
- 2.**Hash Function:** The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.
- 3.**Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.



Components of Hashing

What is Collision?

The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



1) Separate Chaining

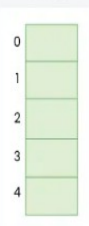
The idea behind **Separate Chaining** is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

Example: We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

Hash function = $\text{key} \% 5$,

Elements = 12, 15, 22, 25 and 37.

Separate Chaining




Slot

0	
1	
2	
3	
4	

Step 01
Empty hash table with range of hash values from 0 to 4 according to the hash function provided.

1 / 5

Separate Chaining



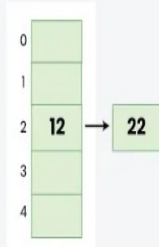
Slot

0	
1	
2	12
3	
4	

Step 02
The first key to be inserted is 12 which is mapped to slot 2 ($12\%5=2$).

2 / 5

Separate Chaining



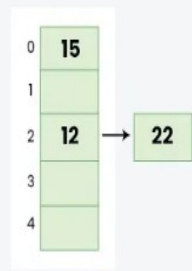
Slot

0	
1	
2	12 → 22
3	
4	

Step 03
The next key is 22 which is mapped to slot 2 ($22\%5=2$) but slot 2 is already occupied by key 12. Separate chaining will handle collision by creating a linked list to slot 2.

3 / 5

Separate Chaining



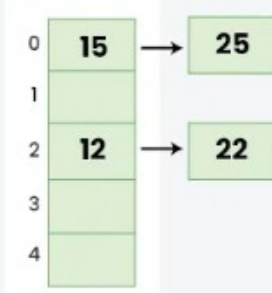
Slot

0	15 → 22
1	
2	12
3	
4	

Step 04
The next key is 15 which is mapped to slot 0 ($15\%5=0$).

4 / 5

Separate Chaining



Slot

0	15 → 25
1	
2	12 → 22
3	
4	

Step 05
The next key is 25 which is mapped to slot 0 ($25\%5=0$). But slot 0 is already occupied by key 25. Again, Separate chaining will handle collision by creating a linked list to slot 2.

5 / 5

2) Open Addressing

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for

an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

2.a) Linear Probing

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

Algorithm:

1. Calculate the hash key. i.e. **$key = data \% size$**
2. Check, if **$hashTable[key]$** is empty
 - store the value directly by **$hashTable[key] = data$**
3. If the hash index already has some value then
 - check for next index using **$key = (key+1) \% size$**
4. Check, if the next index is available $hashTable[key]$ then store the value. Otherwise try for next index.
5. Do the above process till we find the space.

Example: Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

2.b) Quadratic Probing

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

An example sequence using quadratic probing is:

$$H + 1^2, H + 2^2, H + 3^2, H + 4^2, \dots, H + k^2$$

This method is also known as the mid-square method because in this method we look for i^2 -th probe (slot) in i -th iteration and the value of $i = 0, 1, \dots, n - 1$. We always start from the original hash location. If only the location is occupied then we check the other slots.

Let $hash(x)$ be the slot index computed using the hash function and n be the size of the hash table.

If the slot $hash(x) \% n$ is full, then we try $(hash(x) + 1^2) \% n$.

If $(hash(x) + 1^2) \% n$ is also full, then we try $(hash(x) + 2^2) \% n$.

If $(hash(x) + 2^2) \% n$ is also full, then we try $(hash(x) + 3^2) \% n$.

This process will be repeated for all the values of i until an empty slot is found

Example: Let us consider table Size = 7, hash function as $\text{Hash}(x) = x \% 7$ and collision resolution strategy to be $f(i) = i^2$. Insert = 22, 30, and 50

2.c) Double Hashing

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing make use of two hash function,

- The first hash function is **$h1(k)$** which takes the key and gives out a location on the hash table. But if the new location is not occupied or empty then we can easily place our key.
- But in case the location is occupied (collision) we will use secondary hash-function **$h2(k)$** in combination with the first hash-function **$h1(k)$** to find the new location on the hash table.

This combination of hash functions is of the form

$$h(k, i) = (h1(k) + i * h2(k)) \% n$$

where

- i is a non-negative integer that indicates a collision number,
- k = element/key which is being hashed
- n = hash table size.

Complexity of the Double hashing algorithm:

Time complexity: $O(n)$

Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **$h1(k) = k \bmod 7$** and second hash-function is **$h2(k) = 1 + (k \bmod 5)$**

load factor

if there are n entries and b is the size of the array there would be n/b entries on each index. This value n/b is called the **load factor** that represents the load that is there on our map.

Rehashing:

Rehashing is the process of increasing the size of a hashmap and

redistributing the elements to new buckets based on their new hash values. It is done to improve the performance of the hashmap and to prevent collisions caused by a high load factor.

If the load factor exceeds a certain threshold (often set to 0.75), the hashmap becomes inefficient as the number of collisions increases

Extendible Hashing (Dynamic approach to DBMS)

- Extendible Hashing is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

Step 1 - Analyze Data Elements: Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.

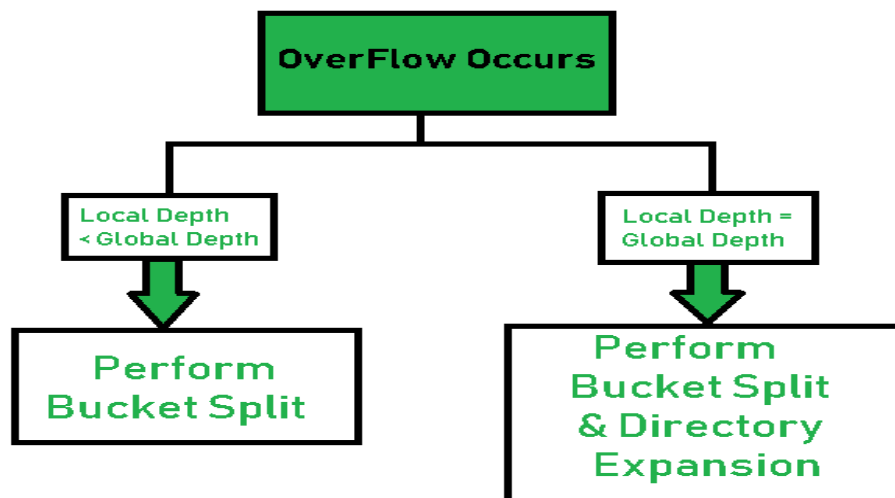
- Step 2 - Convert into binary format: Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.
- Step 3 - Check Global Depth of the directory. Suppose the global depth of the Hash-directory is 3.
- Step 4 - Identify the Directory: Consider the 'Global-Depth' number of LSBs in the binary number and match it to the directory id. Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110001 viz. 001.
- Step 5 - Navigation: Now, navigate to the bucket pointed by the directory with directory-id 001.
- Step 6 - Insertion and Overflow Check: Insert the element and check if the bucket overflows. If an overflow is encountered, go to step 7 followed by Step 8, otherwise, go to step 9.

•Step 7 - Tackling Over Flow Condition during Data Insertion: Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data.

First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.

○Case1: If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers.

Directory expansion will double the number of directories present in the hash structure.



○○Case2: In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.

•Step 8 - Rehashing of Split Bucket Elements: The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.

- Step 9 - The element is successfully hashed.

Example based on Extendible Hashing: Now, let us consider a prominent example of hashing the following elements: 16,4,6,22,24,10,31,7,9,20,26.

Bucket Size: 3 (Assume)

Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

- Solution: First, calculate the binary forms of each of the given numbers.

16- 10000

4- 00100

6- 00110

22- 10110

24- 11000

10- 01010

31- 11111

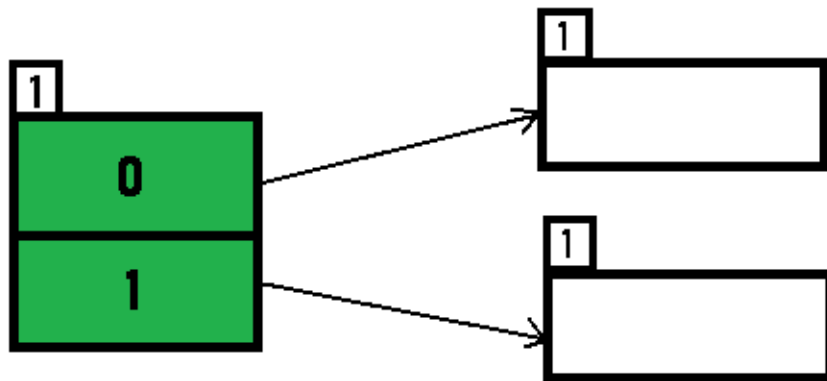
7- 00111

9- 01001

20- 10100

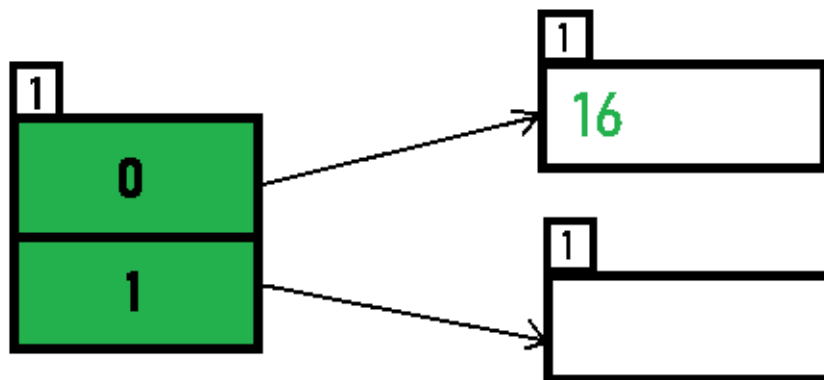
26- 11010

- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



Inserting 16:

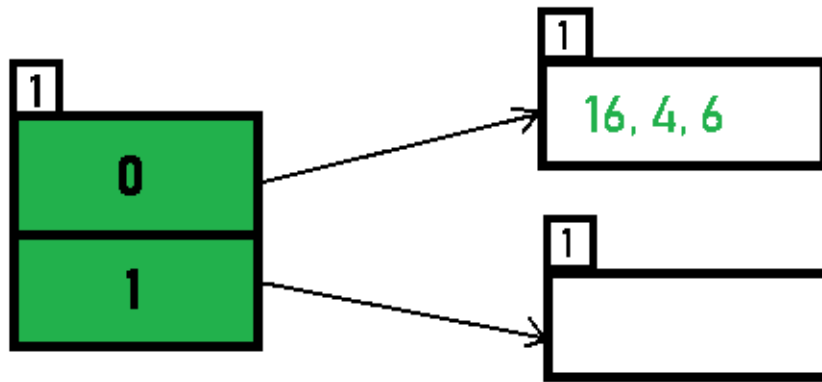
The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 10000 which is 0. Hence, 16 is mapped to the directory with id=0.



$Hash(16) = 10000$

Inserting 4 and 6:

Both 4(100) and 6(110) have 0 in their LSB. Hence, they are hashed as follows:



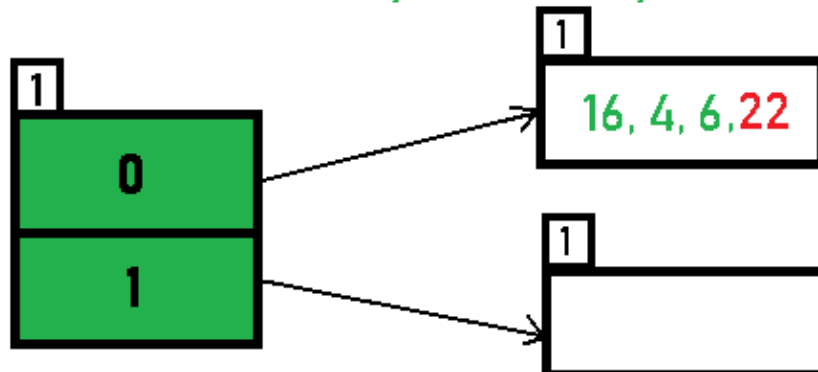
$Hash(4)=10\underline{0}$

$Hash(6)=11\underline{0}$

Inserting 22: The binary form of 22 is 10110. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.

OverFlow Condition

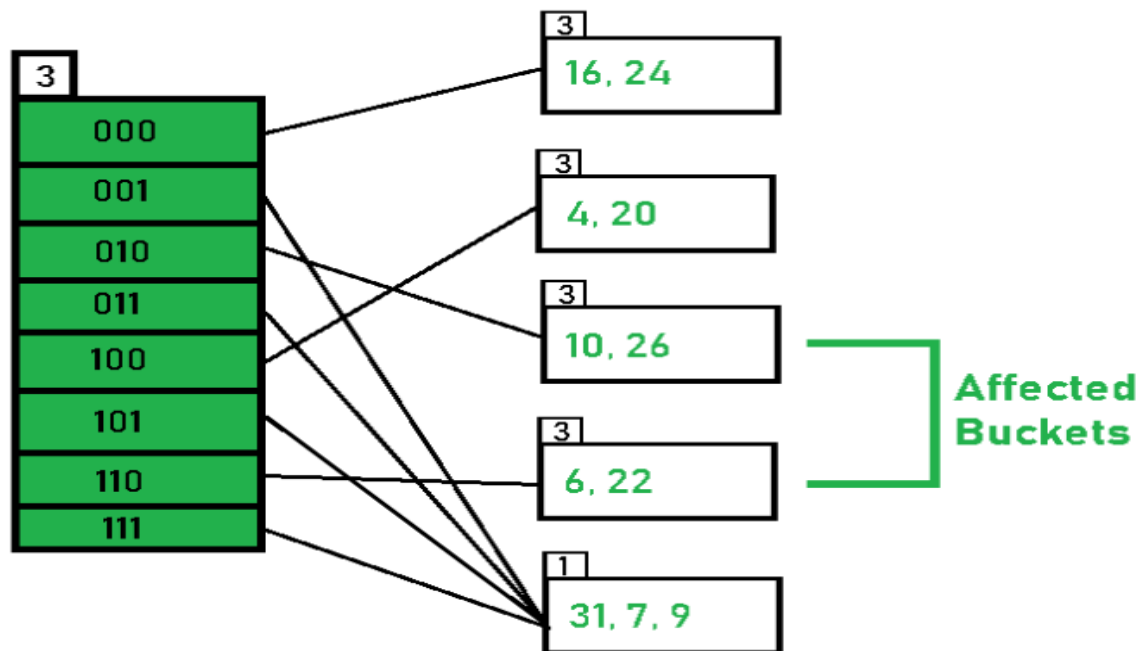
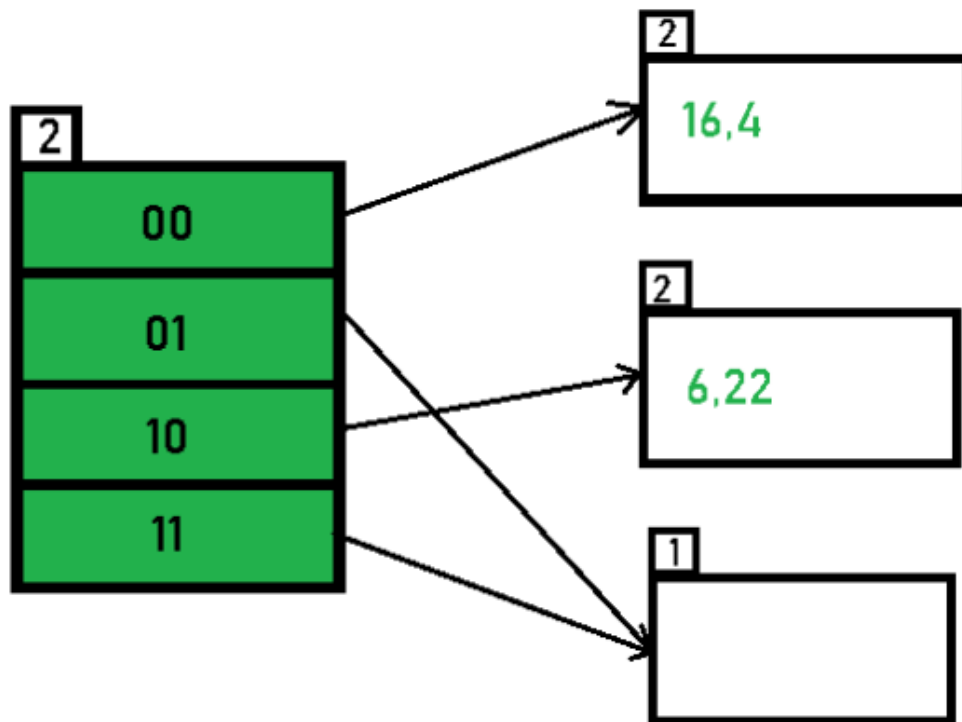
Here, Local Depth=Global Depth



$Hash(22)=1011\underline{0}$

As directed by Step 7-Case 1, Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now, the global depth is 2. Hence, 16, 4, 6, 22 are now rehashed w.r.t 2 LSBs. [16(10000), 4(100), 6(110), 22(10110)]

After Bucket Split and Directory Expansion



Indexed sequential access method (ISAM)

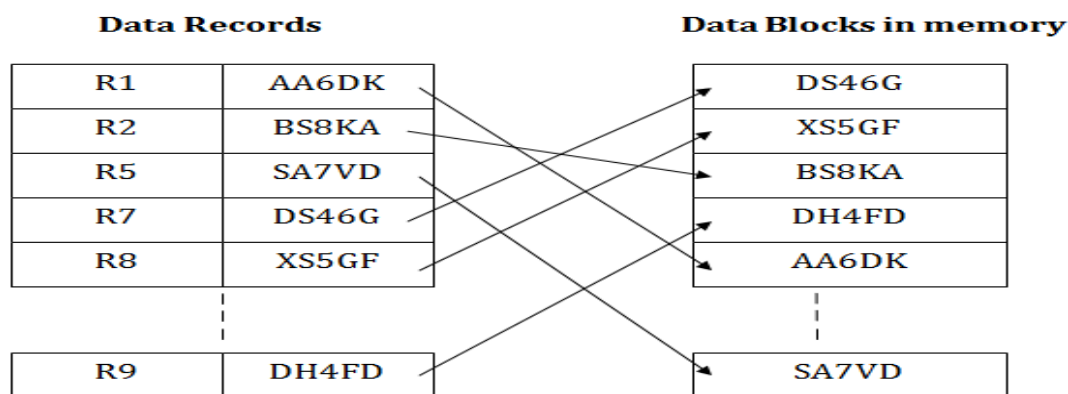
29 Jul 2025 | 3 min read

Introduction

In this article, we will discuss the concept, types, advantages, and disadvantages of hash file organization with the help of its various examples.

What is Indexed Sequential Access Method?

ISAM method is an advanced sequential **file organization**. In this method, records are stored in the file using the **primary key**. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



If any record has to be retrieved based on its index value, then the address of the data block is fetched and the record is retrieved from the memory.

Working of ISAM

It includes indexed characterization as well as sequential data listing in its scheme to achieve more systematic data storage and access.

The functioning of ISAM requires the following steps, which can be summarized as follows:

- **Insertion:** When a new record is specified, the file where the data is stored is updated with the correct sequential order.
- **Retrieval:** A data retrieval procedure requires an index entry that points to the pointer that will directly access the required data file record.

- Deletion:** When a record is deleted, it is flagged as such in the data file, and the corresponding entry in the primary index ceases to exist or is changed.
 - Updation:** It involves checking the index to find the record, upgrading the data file record, and checking the index for required changes.
-

Components of Indexed Sequential Access Method

List the various components of Indexed Sequential Access Method are:

- Data File:** In this component of ISAM records are stored in sorted order based on a key field.
 - Index File:** An index file is a collection of index entries that point to blocks in a data file.
 - Overflow Area:** It is used to store new records which are inserted but cannot fit in the primary data field.
-

Pros of ISAM:

- In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
 - This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.
-

Cons of ISAM

- This method requires extra space in the disk to store the index value.
 - When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
 - When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.
-

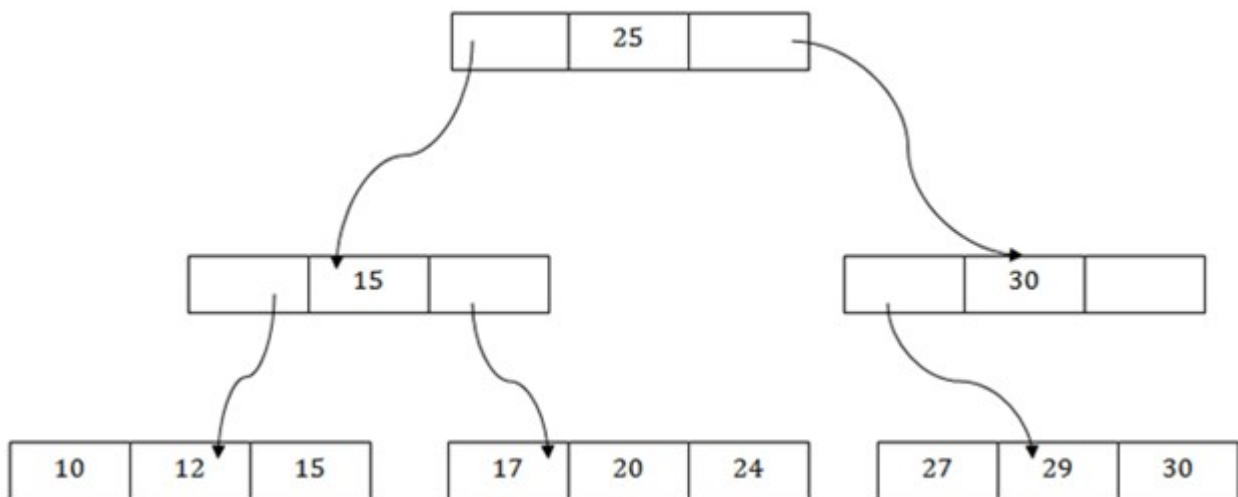
Applications of Indexed Sequential Access Method

Following are the list of applications of Indexed Sequential Access Method is:

- It is used in a mainframe database where there are many data files that need to be accessed quickly and managed effectively.
 - It can be used in inventory systems to manage stock details, product details, order information and facilitate both individual inventory search and sequential inventory checking.
-

B+ File Organization

- B+ tree file organization is the advanced method of an indexed sequential access method. It uses a tree-like structure to store records in File.
 - It uses the same concept of key-index where the primary key is used to sort the records. For each primary key, the value of the index is generated and mapped with the record.
 - The B+ tree is similar to a binary search tree (BST), but it can have more than two children. In this method, all the records are stored only at the leaf node. Intermediate nodes act as a pointer to the leaf nodes. They do not contain any records.
-



The above B+ tree shows that:

- There is one root node of the tree, i.e., 25.
-

- There is an intermediary layer with nodes. They do not store the actual record. They have only pointers to the leaf node.
 - The nodes to the left of the root node contain the prior value of the root and nodes to the right contain next value of the root, i.e., 15 and 30 respectively.
 - There is only one leaf node which has only values, i.e., 10, 12, 17, 20, 24, 27 and 29.
 - Searching for any record is easier as all the leaf nodes are balanced.
 - In this method, searching any record can be traversed through the single path and accessed easily.
-

Pros of B+ tree file organization

- In this method, searching becomes very easy as all the records are stored only in the leaf nodes and sorted the sequential linked list.
 - Traversing through the tree structure is easier and faster.
 - The size of the B+ tree has no restrictions, so the number of records can increase or decrease and the B+ tree structure can also grow or shrink.
 - It is a balanced tree structure, and any insert/update/delete does not affect the performance of tree.
-

Cons of B+ tree file organization

- This method is inefficient for the static method.
-

Functional Dependencies

Formal definition:

- A functional dependency, denoted by $X \twoheadrightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R . The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, we must also have $t_1[Y] = t_2[Y]$.
- That is, the values of the Y component of a tuple in r depend on the values of the X component.
 - Or, there is a functional dependency from X to Y .
 - The set of attributes X is called the **left-hand side** of the FD, and Y is called the **right-hand side**.

- X functionally determines Y in a relation schema R if and only if, whenever two tuples of $r(R)$ agree on their X -value, they must necessarily agree on their Y -value.

Inference Rules for Functional Dependencies

- F is the set of functional dependencies that are specified on relation schema R .
- F^+ is the set of all possible functional dependencies that may hold on R , and is called the **closure of F** .
- That is, F^+ is the set of all functional dependencies that can be inferred from F .
- An FD f is inferred from a set of dependencies F specified on R if f holds in every relation state r that is a legal extension of R .

6 Well-known inference rules for functional dependencies:

IR1 (reflexive rule): If $X \supseteq Y$, then $X \rightarrow Y$.

IR2 (augmentation rule): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

IR3 (transitive rule): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

IR4 (decomposition, or projective, rule): $\{X \rightarrow YZ\} \models X \rightarrow Y$.

IR5 (union, or additive, rule): $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.

IR6 (pseudotransitive rule): $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$.

- Inference rules IR1 through IR3 are known as **Armstrong's inference rules**.

- Inference rules IR1 through IR3 are *sound* and *complete*.

GATE Questions

Q.1: Consider the relation scheme $R = \{E, F, G, H, I, J, K, L, M, N\}$ and the set of functional dependencies $\{\{E, F\} \rightarrow \{G\}, \{F\} \rightarrow \{I, J\}, \{E, H\} \rightarrow \{K, L\}, K \rightarrow \{M\}, L \rightarrow \{N\}\}$ on R . What is the key for R ? (GATE-CS-2014)

- $\{E, F\}$
- $\{E, F, H\}$
- $\{E, F, H, K, L\}$
- $\{E\}$

Solution:

Finding attribute closure of all given options, we get:

$\{E, F\}^+ = \{EFGIJ\}$

$\{E, F, H\}^+ = \{EFHGIJKLMN\}$

$\{E, F, H, K, L\}^+ = \{\{EFHGIJKLMN\}\}$

$\{E\}^+ = \{E\}$

$\{EFH\}^+$ and $\{EFHKL\}^+$ results in set of all attributes, but EFH is minimal. So it will be candidate key. So correct option is (B).

Q.2: How to check whether an FD can be derived from a given FD set?

Solution:

To check whether an FD $A \rightarrow B$ can be derived from an FD set F ,

1. Find $(A)^+$ using FD set F .

2. If B is subset of $(A)^+$, then $A \rightarrow B$ is true else not true.

Q.3: In a schema with attributes A, B, C, D and E following set of functional dependencies are given

$\{A \rightarrow B, A \rightarrow C, CD \rightarrow E, B \rightarrow D, E \rightarrow A\}$

Which of the following functional dependencies is NOT implied by the above set? (GATE IT 2005)

A. $CD \rightarrow AC$

B. $BD \rightarrow CD$

C. $BC \rightarrow CD$

D. $AC \rightarrow BC$

Solution:

Using FD set given in question,
 $(CD)^+ = \{CDEAB\}$ which means $CD \rightarrow AC$ also holds true.

$(BD)^+ = \{BD\}$ which means $BD \rightarrow CD$ can't hold true. So this FD is not implied in FD set. So (B) is the required option.

Others can be checked in the same way.

Q.4: Consider a relation scheme $R = (A, B, C, D, E, H)$ on which the following functional dependencies hold: $\{A \rightarrow B, BC \rightarrow D, E \rightarrow C, D \rightarrow A\}$. What are the candidate keys of R? [GATE 2005]

(a) AE, BE

(b) AE, BE, DE

(c) AEH, BEH, BCH

(d) AEH, BEH, DEH

Solution:

$(AE)^+ = \{ABECD\}$ which is not set of all attributes. So AE is not a candidate key. Hence option A and B are wrong.

$(AEH)^+ = \{ABCDEH\}$

$(BEH)^+ = \{BEHCDA\}$

$(BCH)^+ = \{BCHDA\}$ which is not set of all attributes. So BCH is not a candidate key. Hence option C is wrong.

So correct answer is D.

Easiest way to find the closure set of attribute

Set of all those attributes which can be functionally determined from an attribute set is called **closure of the attribute set** and the closure of the attribute set $\{X\}$ is denoted as $\{X\}^+$. We can only find candidate key and primary keys only with help of closure set of an attribute. So let see the easiest way to calculate the closure set of attributes. In this method you have to do the multiple iteration. **Example-1** : Let $R(A, B, C)$ is a table which has three attributes A, B, C. also their is two functional dependencies :

$A \rightarrow B$

$B \rightarrow C$

So the,

$(A)^+ = \{A, B, C\}$

This is because we can find B with A and when we get B we can also find the C. In this the property of transitivity is used. **Example-2** : Let take table $R(A, B, C, D, E, F, G)$. Functional dependencies are :

$A \rightarrow B$

$BC \rightarrow DE$

$AEG \rightarrow G$

So you have to find the closure of $(AC)^+$. So see the above dependencies. You can see (AC) is already exists and A is already present so by using the above dependencies you can find B with the help of A then you have (ABC) in your set now you also have BC in your set so you can find DE using the above dependencies (see the dependency no.2). Now you have $(ABCDE)$ in your but see you don't have G in your set so you cannot find G using (AEG) dependency. So the closure of the (AC) is $(ABCDE)$. And this mean if you have AC you can find $(ABCDE)$. **Example-3 :** Let take another table $R(A, B, C, D, E)$. Functional dependencies :

$A \rightarrow BC$

$CD \rightarrow E$

$B \rightarrow D$

$E \rightarrow A$

So you have to find the closure of $(B)^+$. Now you have already B in your set. Through B you can find the D (see dependency $B \rightarrow D$) now you have (BD) . You can see that after iterating and checking we can't find any dependency through (BD) . So the closure of B is BD .

Equivalence of Functional Dependencies

Last Updated : 21 Jun, 2025

- Equivalence of functional dependencies means two sets of functional dependencies (FDs) are considered equivalent if they enforce the same constraints on a relation. This happens when every FD in one set can be derived from the other set and vice versa using inference rules like Armstrong's axioms.

Equivalent FDs result in the same set of valid relations and preserve the same data integrity. Identifying equivalent FDs helps in normalization and optimizing database design without losing any constraints.

How To Find the Relationship Between Two Functional Dependency Sets?

Let FD1 and FD2 be two FD sets for a relation R.

- 1.If all FDs of FD1 can be derived from FDs present in FD2, we can say that $FD2 \supset FD1$.
- 2.If all FDs of FD2 can be derived from FDs present in FD1, we can say that $FD1 \supset FD2$.
- 3.If 1 and 2 both are true, $FD1 = FD2$.

Sample Questions

Q.1 Let us take an example to show the relationship between two FD sets. A relation $R(A,B,C,D)$ having two FD sets $FD1 = \{A \rightarrow B, B \rightarrow C, AB \rightarrow D\}$ and $FD2 = \{A \rightarrow B, B \rightarrow C, A \rightarrow C, A \rightarrow D\}$

Step 1: Checking whether all FDs of FD1 are present in FD2

- $A \rightarrow B$ in set FD1 is present in set FD2.
- $B \rightarrow C$ in set FD1 is also present in set FD2.
- $AB \rightarrow D$ is present in set FD1 but not directly in FD2 but we will check whether we can derive it or not. For set FD2, $(AB)^+ = \{A, B, C, D\}$. It means that AB can functionally determine A, B, C, and D. So $AB \rightarrow D$ will also hold in set FD2.

As all FDs in set FD1 also hold in set FD2, **$FD2 \supset FD1$** is true.

Step 2: Checking whether all FDs of FD2 are present in FD1

- $A \rightarrow B$ in set FD2 is present in set FD1.
- $B \rightarrow C$ in set FD2 is also present in set FD1.
- $A \rightarrow C$ is present in FD2 but not directly in FD1 but we will check whether we can derive it or not. For set FD1, $(A)^+ = \{A, B, C, D\}$. It

means that A can functionally determine A, B, C, and D. SO $A \rightarrow C$ will also hold in set FD1.

- $A \rightarrow D$ is present in FD2 but not directly in FD1 but we will check whether we can derive it or not. For set FD1, $(A)^+ = \{A, B, C, D\}$. It means that A can functionally determine A, B, C, and D. SO $A \rightarrow D$ will also hold in set FD1.

As all FDs in set FD2 also hold in set FD1, **$FD2 \supset FD1$** is true.

Step 3: As $FD2 \supset FD1$ and $FD1 \supset FD2$ both are true **$FD2 = FD1$** is true. These two FD sets are semantically equivalent.

Q.2 Let us take another example to show the relationship between two FD sets. A relation $R2(A,B,C,D)$ having two FD sets $FD1 = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$ and $FD2 = \{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$

Step 1: Checking whether all FDs of FD1 are present in FD2

- $A \rightarrow B$ in set FD1 is present in set FD2.
- $B \rightarrow C$ in set FD1 is also present in set FD2.
- $A \rightarrow C$ is present in FD1 but not directly in FD2 but we will check whether we can derive it or not. For set FD2, $(A)^+ = \{A, B, C, D\}$. It means that A can functionally determine A, B, C, and D. SO $A \rightarrow C$ will also hold in set FD2.

As all FDs in set FD1 also hold in set FD2, **$FD2 \supset FD1$** is true.

Step 2: Checking whether all FDs of FD2 are present in FD1

- $A \rightarrow B$ in set FD2 is present in set FD1.,
- $B \rightarrow C$ in set FD2 is also present in set FD1.
- $A \rightarrow D$ is present in FD2 but not directly in FD1 but we will check whether we can derive it or not. For set FD1, $(A)^+ = \{A, B, C\}$. It means that A can't functionally determine D.
- So $A \rightarrow D$ will not hold in FD1.

As all FDs in set FD2 do not hold in set FD1, **$FD2 \not\supset FD1$** .

Step 3: In this case, **$FD2 \supset FD1$** and **$FD2 \not\supset FD1$** , these two FD sets are not semantically equivalent.

How To Find Minimal Set:

If we have a set of functional dependencies, we get the simplest and irreducible form of functional dependencies after reducing these functional dependencies. This is called the **Minimal Cover** or **Irreducible Set** (as we can't reduce the set further). It is also called a **Canonical Cover**.

Let us understand the procedure to find the minimal cover by this example:

The Given Functional Dependencies are - $A \rightarrow B$, $B \rightarrow C$, $D \rightarrow ABC$, $AC \rightarrow D$

1. Steps to Find Minimal Cover

Step 1: First split all the right-hand attributes of all FDs such that RHS contains a single attribute.

Example: $D \rightarrow ABC$ is split into $D \rightarrow A$, $D \rightarrow B$ and $D \rightarrow C$

$A \rightarrow B$, $B \rightarrow C$, $D \rightarrow A$, $D \rightarrow B$, $D \rightarrow C$, $AC \rightarrow D$

[Note: We can't split $AC \rightarrow D$ as $A \rightarrow D$, $C \rightarrow D$]

Step 2: Now remove all redundant FDs.

[Redundant FD is if we derive one FD from another FD]

Let, 's test the redundance of $A \rightarrow B$

$A^+ = A$ (A is only closure contains to A, simply we can derive A from A) So, $A \rightarrow B$ is not redundant.

Similarly, $B \rightarrow C$ is not redundant.

But, $D \rightarrow B$ and $D \rightarrow C$ is redundant

because $D^+ = A$ and $A^+ = B$, So $D^+ = B$ can be derived which means $D \rightarrow B$ is redundant. So, We remove $D \rightarrow B$ from the FDs set.

Now, check for $D \rightarrow C$, it is not redundant because we can't $D^+ = B$ and $B^+ = C$ as we remove $D \rightarrow B$ from the list.

At last, we check for $AC \rightarrow D$. This is also not redundant.

$AC^+ = AC$

So, the final FDs are: $A \rightarrow B$, $B \rightarrow C$, $D \rightarrow A$, $D \rightarrow C$, $AC \rightarrow D$

Step 3: Find the Extraneous attribute and remove it.

In this case, we should only check $AC \rightarrow D$. Simply we can say the right-hand attributes are pointed by only one attribute at one time.

$AC \rightarrow D$, either A or C, or none can be extraneous.

If $A^+ = C$ then C is extraneous and it can be removed.

If $C^+ = A$ then A is extraneous and it can be removed.

So, the final FDs are: $A \rightarrow B$, $B \rightarrow C$, $D \rightarrow A$, $D \rightarrow C$, $AC \rightarrow D$

Hence, we can write it as $A \rightarrow B$, $B \rightarrow C$, $D \rightarrow AC$, $AC \rightarrow D$ this is the minimum cover.

2. Find the Minimal Cover

Given Functional Dependencies

- $A \rightarrow B$
- $B \rightarrow C$
- $D \rightarrow ABC$
- $AC \rightarrow D$

Step 1: Split the Functional Dependencies

- $A \rightarrow B$
- $B \rightarrow C$
- $D \rightarrow A$
- $D \rightarrow B$
- $D \rightarrow C$
- $AC \rightarrow D$

Step 2: Remove Redundant FDs

- $A \rightarrow B$ (not redundant)
- $B \rightarrow C$ (not redundant)
- $D \rightarrow A$ (not redundant)
- $D \rightarrow B$ (redundant, because $D \rightarrow A$ and $A \rightarrow B$)
- $D \rightarrow C$ (not redundant, as $D \rightarrow B$ was removed)
- $AC \rightarrow D$ (not redundant)

After removing redundancies FDs set became

- $A \rightarrow B$
- $B \rightarrow C$
- $D \rightarrow A$
- $D \rightarrow C$
- $AC \rightarrow D$

Step 3: Remove Extraneous Attributes

- In $AC \rightarrow D$, check if A or C is extraneous.
- Compute closures
- $A_+ = \{A, B, C\}$

- $C_+ = \{C\}$

- Since **A** alone does not give D, A is not extraneous

- Since **C** alone does not give D, C is not extraneous

So Minimal cover of $(A \rightarrow B, B \rightarrow C, D \rightarrow ABC, AC \rightarrow D) \Rightarrow (A \rightarrow B, B \rightarrow C, D \rightarrow A, AC \rightarrow D)$

By ensuring all functional dependencies are minimal and non-redundant, we obtain the correct minimal cover.

Canonical Cover of Functional Dependencies in DBMS

Last Updated : 23 Jul, 2025

- Managing a large set of functional dependencies can result in unnecessary computational overhead. This is where the canonical cover becomes useful. A canonical cover is a set of functional dependencies that is equivalent to a given set of functional dependencies but is minimal in terms of the number of dependencies. Canonical Cover of functional dependency is also called minimal set of functional dependency or irreducible form of functional dependency.

Cover of FD:

A set of Fds **F** is said to cover another set of Fds **E**. In **E** every FD also in **F** closure. Every FD in **E** can be inferred from **F**. We can say **E** is covered by **F**.

Definition: Cover of Functional Dependencies

A set of functional dependencies **F** is said to be a **cover** of another set of functional dependencies **E** if:

Every functional dependency in E can be inferred from F,
i.e., $E \subseteq F^+$

This means:

- The closure of **F** (denoted F^+) includes **all dependencies in E**.
- In other words, for every FD in **E**, that FD must also be **implied by F**.

We say:

"E is covered by F" or "F covers E"

What is Decomposition in DBMS?

When we divide a table into multiple tables or divide a relation into multiple relations, then this process is termed Decomposition in DBMS. We perform decomposition in DBMS when we want to process a particular data set. It is performed in a database management system when we need to ensure consistency and remove anomalies and duplicate data present in the database. When we perform decomposition in DBMS, we must try to ensure that no information or data is lost.

Types of Decomposition

There are two types of Decomposition:

- Lossless Decomposition
- Lossy Decomposition

Lossless Decomposition

The process in which where we can regain the original relation R with the help of joins from the multiple relations formed after decomposition. This process is termed as lossless decomposition. It is used to remove the redundant data from the database while retaining the useful information. The lossless decomposition tries to ensure following things:

- While regaining the original relation, no information should be lost.
- If we perform join operation on the sub-divided relations, we must get the original relation.

Example:

There is a relation called R(A, B, C)

A	B	C
55	16	27

A	B	C
48	52	89

Now we decompose this relation into two sub relations R1 and R2
R1(A, B)

A	B
55	16
48	52

R2(B, C)

B	C
16	27
52	89

After performing the Join operation we get the same original relation

A	B	C
55	16	27
48	52	89

Lossy Decomposition

As the name suggests, lossy decomposition means when we perform join operation on the sub-relations it doesn't result to the same relation which was decomposed. After the join operation, we always found some extraneous tuples. These extra tuples generates difficulty for the user to identify the original tuples.

Example:

We have a relation R(A, B, C)

A	B	C
1	2	1
2	5	3

A	B	C
3	3	3

Now , we decompose it into sub-relations R1 and R2

R1(A, B)

A	B
1	2
2	5
3	3

R2(B, C)

B	C
2	1
5	3
3	3

Now After performing join operation

A	B	C
1	2	1
2	5	3
2	3	3
3	5	3
3	3	3

Properties of Decomposition

- Lossless:** All the decomposition that we perform in Database management system should be lossless. All the information should not be lost while performing the join on the sub-relation to get back

the original relation. It helps to remove the redundant data from the database.

- Dependency Preservation:** Dependency Preservation is an important technique in database management system. It ensures that the functional dependencies between the entities is maintained while performing decomposition. It helps to improve the database efficiency, maintain consistency and integrity.

- Lack of Data Redundancy:** Data Redundancy is generally termed as duplicate data or repeated data. This property states that the decomposition performed should not suffer redundant data. It will help us to get rid of unwanted data and focus only on the useful data or information.

Fully Functional Dependency

In full functional dependency an attribute or a set of attributes uniquely determines another attribute or set of attributes. If a relation R has attributes X, Y, Z with the dependencies $X \rightarrow Y$ and $X \rightarrow Z$ which states that those dependencies are fully functional.

7. Partial Functional Dependency

In partial functional dependency a non key attribute depends on a part of the composite key, rather than the whole key. If a relation R has attributes X, Y, Z where X and Y are the composite key and Z is non key attribute. Then $X \rightarrow Z$ is a partial functional dependency in RDBMS.

1. Trivial Functional Dependency

In Trivial Functional Dependency, a dependent is always a subset of the determinant. i.e. If $X \rightarrow Y$ and Y is the subset of X, then it is called trivial functional dependency.

Symbolically: $A \rightarrow B$ is trivial functional dependency if B is a subset of A. The following dependencies are also trivial: $A \rightarrow A$ & $B \rightarrow B$

Example 1 :

- $ABC \rightarrow AB$

- ABC -> A
- ABC -> ABC

Example 2:

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, {roll_no, name} → name is a trivial functional dependency, since the dependent name is a subset of determinant set {roll_no, name}. Similarly, roll_no → roll_no is also an example of trivial functional dependency.

Normalization

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of minimizing redundancy and minimizing the insertion, deletion, and update anomalies

Insertion Anomalies: These anomalies occur when it is not possible to insert data into a database because the required fields are missing or because the data is incomplete. For example, if a database requires that every record has a **primary key**, but no value is provided for a particular record, it cannot be inserted into the database.

Deletion anomalies: These anomalies occur when deleting a record from a database and can result in the unintentional loss of data. For example, if a database contains information about customers and orders, deleting a customer record may also delete all the orders associated with that customer.

Update anomalies: These anomalies occur when modifying data in a database and can result in inconsistencies or errors. For example, if

a database contains information about employees and their salaries, updating an employee's salary in one record but not in all related records could lead to incorrect calculations and reporting.

First Normal Form (1NF)

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385 , 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389 , 8589830302	Punjab

The above EMPLOYEE table is an unnormalized relation as it contains multiple values corresponding to EMP_PHONE attribute i.e. these values are non-atomic. So relations with multi value entries are called unnormalized relations.

To overcome this problem, we have to eliminate the non atomic values of EMP_PHONE attribute.

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar

12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

There are 3 ways to achieve first normal form.

Method 1:

To remove the repeating values for a column, the EMPLOYEE table was converted to a flat relation EMPLOYEE_1 table by repeating the pair (EMP_ID, EMP_NAME) for every entry in the table. Now the new relation does not contain any non-atomic values so the table is said to be normalized and is in First Normal Form.

Method 2:

Another method is to remove the attributes that violate 1NF and place it in a separate relation along with primary key. So the unnormalized relation, EMPLOYEE table is decomposed into two sub-relations **EMP_DETAILS** and **EMP_PERFORMANCE**

EMP_DETAILS

EMP_ID	EMP_NAME
14	John
20	Harry
12	Sam

EMP_PERFORMANCE

EMP_ID	EMP_PHONE	EMP_STATE
14	7272826385	UP
14	9064738238	UP
20	8574783832	Bihar
12	7390372389	Punjab
12	8589830302	Punjab

The main idea of decomposing the relations is to keep the different types of information in their separate relation as first normal form disallows multivalued attribute that are composite in nature. In the **EMP_DETAILS** relation, the attribute (EMP_ID) acts as a primary key and in the **EMP_PERFORMANCE** relation the

attributes (EMP_ID, EMP_PHONE) act as a primary key. Now it satisfies both the conditions for a relation to be in 1NF.

The relation is decomposed according to the following rules:

- One relation consists of the primary key (EMP_ID) of the original relation (i.e. EMPLOYEE) and non repeating attributes of the original relation (i.e. EMP_NAME).
- The other relation consists of copy of the primary key of the original relation and all the repeating attributes of the original relation.

Method 3:

The third method of normalizing a unnormalized relation into 1NF will be explained with following example where skills of an employee of some company are fixed. Suppose an employee can have maximum of five skills.

EMP_SKILL relation

EMP_ID	Skill
14	DBMS, C, C++
20	JAVA, C
12	DBMS, HTML, VB, MS OFFICE

Here the **EMP_SKILL relation** is not 1NF as the skill attribute contains a set of values. So to remove this problem, we define multiple Skill columns as shown.

The above relation is decomposed into 1NF in the following example.

EMP_ID	Skill_1	Skill_2	Skill_3	Skill_4	Skill_5
14	DBMS	C	C++	-	-
20	JAVA	C	-	-	-
12	DBMS	HTML	VB	MS OFFICE	-

The above representation is in 1NF but this technique is not preferred as it may cause problems such as:

- It would be difficult to query the relation. For Example, it would be difficult to answer the queries like “which employee share a skill?”, “Which employees have skill C?”
- Restriction of employee skills to 5. If employee with more skills appears, it would be left unrecorded.

To sum up, all the three approaches are correct because they transform any unnormalized table into a first normal form table. However, the second approach where table is decomposed into relations is more efficient as minimizes the

duplicacy of the data. So for a relation to be in first normal form, each set of repeating groups should appear in its own table and every relation should have a primary key.

Second Normal Form (2NF)

3 Dec 2024 | 3 min read

- In the 2NF, relational must be in 1NF.
- In the second normal form, all non-key attributes are fully functional dependent on the primary key

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	TEACHER_AGE
25	30
47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology

47	English
83	Math
83	Computer

Third Normal Form (3NF)

3 Dec 2024 | 3 min read

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
- 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency $X \rightarrow Y$.

- 1.X is a super key.
- 2.Y is a prime attribute, i.e., each element of Y is part of some candidate key.

To explain the 3NF, let us consider the example of Employee_Detail relation as shown below.

Example: EMPLOYEE_DETAIL table

Example:

EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above:

1. {EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....so on

Candidate key: {EMP_ID}

Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.

Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on

super key(EMP_ID). It violates the rule of third normal form. The reduction of 2NF relation into 3NF consists of splitting the 2NF into appropriate relations such that every non-key attribute is functionally dependent on the primary key not transitively or indirectly of the respective relations.

That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

Fourth normal form (4NF)

26 May 2025 | 3 min read

Introduction:

Before discussing the fourth normal form, one should be aware of **multivalued dependency**. This is a consequence of 1NF which does not allow an attribute in a tuple to have a set of values.

The normal forms developed so far only deal with functional dependencies only. It is still possible for a relation in 3NF or BCNF to have anomalies while updating,

inserting, and deleting. This can happen when multivalued dependencies are not properly taken care of. To eliminate anomalies arising from these dependencies, the concept of Fourth Normal Form was developed.

- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
- For a dependency $A \twoheadrightarrow B$, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

To explain the concept of 4NF, let us consider an "Employee relation" as shown below. This relation includes all the three attributes and acts as a primary key because no single attribute can uniquely identify a record.

Emp_Name	Equipment	Language
Anurag	Personal Computer	English
Anurag	Personal Computer	French
Anurag	Mainframe	English
Anurag	Mainframe	French
Kapil	Personal Computer	English
Kapil	Personal Computer	French
Kapil	Personal Computer	Japanese

The relationship between Empname and Equipment is a multivalued dependency because each pair of (Emp_name,Language) values in the Employee relation, the associated set of Equipment values is determined only by Empname and is independent of language.

Equipment Anurag, English = Equipment Anurag, French = { Personal Computer, Mainframe }

Equipment Kapil, English = Equipment Anurag, French = { Personal Computer, Mainframe } = Equipment Kapil , Japanese = { Personal Computer }

Thus it implies that

Emp_Name \twoheadrightarrow Equipment

Similarly, the relationship between Emp_name and Language is a multivalued dependency which is represented as

Emp_Name \twoheadrightarrow Language

So to transform a table with multivalued dependencies into 4NF, move each multivalued dependencies pair to a new relation which is shown below.

Emp_Equip

Emp_Name	Equipment
Anurag	Personal Computer
Anurag	Mainframe
Kapil	Personal Computer

Emp_Lang

Emp_Name	Language
Anurag	English
Anurag	French
Kapil	English
Kapil	French
Kapil	Japanese

So we see that the source relation Employee is non losslessly decomposed into sub relations.

Emp_Equip (@Emp_name + @Equipment)

Emp_Lang (@Emp_name + @Language)

Where @symbols shows the primary keys.

The resulting relations Emp_Equip (a variety of equipment allocated to an employee) and Emp_Lang (a variety of languages an employee is familiar with) are in 4NF.

Fifth normal form (5NF)

26 May 2025 | 4 min read

- A relation is in 5NF if it is in **4NF** and does not contain any **join dependency** and joining should be lossless.
 - 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
 - 5NF is also known as Project-join normal form (PJ/NF).
-

- A Relation is in 5NF, if for all join dependencies, at least one of the following method applies:
 - (R_1, R_2, \dots, R_n) is a trivial join dependency.
 - Each R_i is a candidate key for R i.e. every join dependency in R is a consequence only of the candidate keys of R .
-

15.4.1

External Sort-Merge Algorithm

Sorting of relations that do not fit in memory is called external sorting. The most commonly used technique for external sorting is the external sort-merge algorithm. We describe the external sort-merge algorithm next. Let M denote the number of blocks in the main memory buffer available for sorting, that is, the number of disk blocks whose contents can be buffered in available main memory.

1. In the first stage, a number of sorted runs are created; each run is sorted but contains only some of the records of the relation.

```

i = 0;
repeat
    read M blocks of the relation, or the rest of the relation,
        whichever is smaller;
    sort the in-memory part of the relation;
    write the sorted data to run file  $R_i$ ;
    i = i + 1;
until the end of the relation
  
```

2. In the second stage, the runs are merged. Suppose, for now, that the total number of runs, N , is less than M , so that we can allocate one block to each run and have space left to hold one block of output. The merge stage

operates as follows:

read one block of each of the N files R_i into a buffer block in memory; repeat

Page 702

choose the first tuple (in sort order) among all buffer blocks;

write the tuple to the output, and delete it from the buffer block;

if the buffer block of any run R_i is empty and not end-of-file(R_i)

then read the next block of R_i into the buffer block;

until all input buffer blocks are empty

The output of the merge stage is the sorted relation. The output file is buffered to reduce the number of disk write operations. The preceding merge operation is a generalization of the two-way merge used by the standard in-memory sort-merge algorithm; it merges N runs, so it is called an N -way merge.

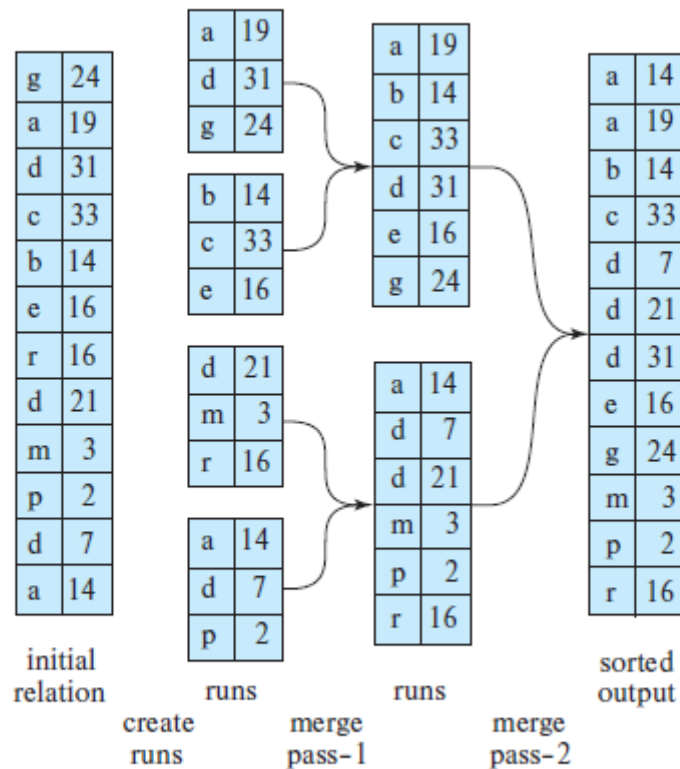


Figure 15.4 External sorting using sort-merge.

Evaluation of **Relational Operators** like **Selection** and **Join** is a fundamental part of **query processing** in **DBMS**. These operations are implemented using **efficient algorithms** to handle large datasets stored on disk.

1. Selection Operator (σ)

Purpose:

Retrieves tuples that satisfy a **predicate/condition** from a relation.

Example:

$\sigma_{\text{age} > 25}$ (Employee)

Algorithms for Selection

a. Linear Search (Brute Force)

- **Scan** each tuple in the relation.
- Check if it satisfies the condition.

Cost:

- I/O = number of blocks in the relation
 - Best if no indexes are available.
-

b. Binary Search

- Used if:
 - The relation is **sorted on the attribute**.
 - Condition is **equality** (e.g., $\sigma_{\text{empID} = 1001}$).

Cost:

- $O(\log_2 n)$ block accesses to find the match
 - Then linear scan for further matches (for duplicates)
-

c. Using Index (Primary/Secondary)

Types:

1. Primary Index on Selection Attribute

- Fast access using index.
- Only one or few disk accesses.

2. Secondary Index

- May point to multiple blocks → more disk I/O.
 - Useful for non-key attributes.
-

d. Index-only Selection

- If index **stores all required fields**, we **don't access the main table**.
 - Highly efficient.
-

1. Linear / Sequential Search

Description:

- Scan each block **one by one**.
- No index required.
- Works for **any attribute** (key or non-key).

Cost:

- **Unsorted file:**
 - **On average:** $b/2$ block accesses
 - **Worst case:** b block accesses
- **Where:**
 b = total number of blocks in the file

Example:

If file has 4000 blocks:

- Avg cost = $4000 / 2 = 2000$
 - Worst case = **4000**
-

2. Binary Search

Description:

- Only works on **sorted files**
- Performs $\log_2(b)$ comparisons
- Efficient for **equality selection on key attribute**

Cost:

- $\log_2(b)$ to locate block
 - 1 block to fetch record from that block

$$\text{Total cost} = \log_2(b) + 1$$

Example:

If file has 4000 blocks:

- $\log_2(4000) \approx 12$
 - Total cost = $12 + 1 = 13$ **block accesses**
-

3. Index-Based Search

Indexes can be:

- **Primary** (on sorted file, on key)
- **Secondary** (on unsorted file or non-key)
- **Single-level or Multilevel**

a. Primary Index (Single-level)

Description:

- Index is built on **sorted key attribute**
- Points to the **first record of each block**

Cost:

- $\log_2(b_{\text{sub}i}) + 1$
 - Where $b_{\text{sub}i}$ = blocks in index
 - +1 for data block access

If index fits in memory, index access cost may be **0**.

b. Secondary Index

Description:

- Built on **non-key** or **unsorted** attributes
- Points to **individual records**, not blocks

Cost:

- $\log_2(b_{\text{sub}i}) + t$
 - t = number of matching tuples (each may be in different block)
-

c. Multilevel Index

Description:

- Index on index → like a tree (B-tree/B+ tree)
- Height = number of levels = **h**

Cost:

- For equality selection:
Cost = h (index block accesses) + 1 (data block)
So, **Total = h + 1**
-

Example (from your earlier question):

- Multilevel index has **4 levels**

- So, cost = 4 + 1 = 5 **block accesses**

2. Join Operator (\bowtie)

Purpose:

Combines tuples from two relations based on a join condition.

Example:

Employee \bowtie Department.dept_id = Employee.dept_id

Types of Join:

- **Theta Join** ($R \bowtie_{\theta} S$) — generic condition
 - **Equi-Join** ($R \bowtie R.A = S.B$) — equality condition
 - **Natural Join** — implicit equality on attributes with the same name
 - **Outer Join** — retains unmatched tuples
 - **Self Join** — relation joined with itself
-

Join Algorithms

Let's assume:

- **R** has **M** blocks, **r** tuples
 - **S** has **N** blocks, **s** tuples
 - **B** is the number of buffer pages available in memory
-

a. Nested Loop Join

i. Naive Nested Loop Join

Query Optimization in DBMS: Heuristic-Based Approach

What is Query Optimization?

Query optimization is the process of **selecting the most efficient way to execute a given query** by considering multiple execution plans and choosing the one with the **lowest cost** (in terms of CPU, memory, and especially disk I/O).

There are **two main approaches**:

1. Heuristic-Based Optimization

2. Cost-Based Optimization

Let's focus on **Heuristic-Based Query Optimization** in detail:

1. What is Heuristic-Based Optimization?

A rule-based method where a set of **predefined transformation rules (heuristics)** are applied to rewrite a query into a more efficient form **without computing the actual cost** of each plan.

- It's **faster** than cost-based optimization
 - Used in **early DBMSs** and still used today as a **preprocessing step** before cost-based optimization
-

2. Common Heuristic Rules

Here are the **common transformation rules** (heuristics) used to optimize relational algebra queries:

Rule 1: Perform Selections Early

- Apply σ (**selection**) operations as early as possible in the query tree.
- Reduces the number of tuples early, improving efficiency.

Example:

Instead of:

Cost-Based Query Optimization in DBMS (Detailed Explanation)

What is Cost-Based Optimization?

Cost-based optimization is a technique in **query optimization** where the **DBMS generates multiple execution plans** for a query and chooses the **one with the lowest estimated cost**, based on:

- Disk I/O
- CPU usage
- Memory usage
- Estimated size of intermediate results

Goal: Minimize total resource usage, especially **disk I/O**, which is typically the most expensive.

Steps in Cost-Based Optimization

1. Translate SQL → Relational Algebra

Convert the SQL query into an **initial relational algebra expression**.

2. Generate Alternative Execution Plans

Using **algebraic equivalences** and transformations (e.g., reordering joins), the optimizer generates different **logical plans** and their **physical implementations**.

Example alternatives:

- Join order: $(A \bowtie B) \bowtie C$ vs. $A \bowtie (B \bowtie C)$
 - Join algorithm: nested-loop join vs. hash join
 - Index vs. full scan
-

3. Estimate Cost of Each Plan

Use **catalog statistics** to estimate:

- **Relation size (number of tuples)**
 - **Selectivity of predicates**
 - **Number of disk I/O operations**
 - **Number of intermediate results**
-

4. Choose the Plan with Minimum Cost

Compare estimated costs and pick the **least expensive** plan.

Cost Model

Let's understand the **cost estimation** with a simplified disk-based model:

- $B(R)$ = number of blocks in relation R
- $T(R)$ = number of tuples in R
- $V(R, A)$ = number of distinct values of attribute A in R
- B = number of available buffer pages

Selection Cost (σ)

Linear scan (no index):

Cost-Based Optimization:

For a given query and environment, the Optimizer allocates a cost in numerical form which is related to each step of a possible plan and then finds these values together to get a cost estimate for the plan or for the possible strategy. After calculating the costs of all possible plans, the Optimizer tries to choose a plan which will have the possible lowest cost estimate. For that reason, the Optimizer may be sometimes referred to as the Cost-Based Optimizer. Below are some of the features of the cost-based optimization-

- 1.The cost-based optimization is based on the cost of the query that to be optimized.
- 2.The query can use a lot of paths based on the value of indexes, available sorting methods, constraints, etc.
- 3.The aim of query optimization is to choose the most efficient path of implementing the query at the possible lowest minimum cost in the form of an algorithm.
- 4.The cost of executing the algorithm needs to be provided by the query Optimizer so that the most suitable query can be selected for an operation.
- 5.The cost of an algorithm also depends upon the cardinality of the input.

Here's a detailed comparison of the **features of various types of NoSQL databases**:

Overview: Types of NoSQL Databases

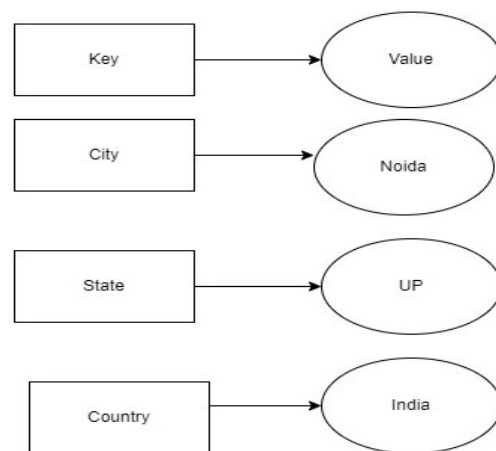
Type	Data Model	Ideal For
Key-Value Store	Key-Value pairs	Caching, session storage, simple lookups

Type	Data Model	Ideal For
Document Store	JSON/XML/BSON	Content management, product catalogs
Column-Family Store	Column-oriented	Analytics, large-scale data warehouses
Graph Database	Nodes & Edges	Social networks, recommendation engines

Key-Value Data Model in NoSQL

Last Updated : 17 Feb, 2022

- A key-value data model or database is also referred to as a key-value store. It is a non-relational type of database. In this, an associative array is used as a basic database in which an individual key is linked with just one value in a collection. For the values, keys are special identifiers. Any kind of entity can be valued. The collection of key-value pairs stored on separate records is called key-value databases and they do not have an already defined structure.



How do key-value databases work?

A number of easy strings or even a complicated entity are referred to as a value that is associated with a key by a key-value database, which is utilized to monitor the entity. Like in many programming paradigms, a key-value database resembles a map object or array, or

dictionary, however, which is put away in a tenacious manner and controlled by a DBMS.

An efficient and compact structure of the index is used by the key-value store to have the option to rapidly and dependably find value using its key. For example, Redis is a key-value store used to track lists, maps, heaps, and primitive types (which are simple data structures) in a constant database. Redis can uncover a very basic point of interaction to query and manipulate value types, just by supporting a predetermined number of value types, and when arranged, is prepared to do high throughput.

When to use a key-value database:

Here are a few situations in which you can use a key-value database:-

- User session attributes in an online app like finance or gaming, which is referred to as real-time random data access.
- Caching mechanism for repeatedly accessing data or key-based design.
- The application is developed on queries that are based on keys.

Features:

- One of the most un-complex kinds of NoSQL data models.
- For storing, getting, and removing data, key-value databases utilize simple functions.
- Querying language is not present in key-value databases.
- Built-in redundancy makes this database more reliable.

Advantages:

- It is very easy to use. Due to the simplicity of the database, data can accept any kind, or even different kinds when required.
- Its response time is fast due to its simplicity, given that the remaining environment near it is very much constructed and improved.
- Key-value store databases are scalable vertically as well as horizontally.
- Built-in redundancy makes this database more reliable.

Disadvantages:

- As querying language is not present in key-value databases, transportation of queries from one database to a different database cannot be done.
- The key-value store database is not refined. You cannot query the database without a key.

Document Data Model:

A Document Data Model is a lot different than other data models because it stores data in JSON, BSON, or XML documents. In this data model, we can move documents under one document and apart from this, any particular elements can be indexed to run queries faster. Often documents are stored and retrieved in such a way that it becomes close to the data objects which are used in many applications which means very less translations are required to use data in applications. JSON is a native language that is often used to store and query data too.

So in the document data model, each document has a key-value pair below is an example for the same.

```
{  
  "Name" : "Yashodhra",  
  "Address" : "Near Patel Nagar",
```



```
"Email" : "yahoo123@yahoo.com",  
"Contact" : "12345"  
}
```

Working of Document Data Model:

This is a data model which works as a semi-structured data model in which the records and data associated with them are stored in a single document which means this data model is not completely unstructured. The main thing is that data here is stored in a document.

Features:

- Document Type Model: As we all know data is stored in documents rather than tables or graphs, so it becomes easy to map things in many programming languages.
- Flexible Schema: Overall schema is very much flexible to support this statement one must know that not all documents in a collection need to have the same fields.
- Distributed and Resilient: Document data models are very much dispersed which is the reason behind horizontal scaling and distribution of data.
- Manageable Query Language: These data models are the ones in which query language allows the developers to perform CRUD (Create Read Update Destroy) operations on the data model.

Examples of Document Data Models :

- Amazon DocumentDB
- MongoDB
- Cosmos DB
- ArangoDB
- Couchbase Server

- CouchDB

Advantages:

- Schema-less: These are very good in retaining existing data at massive volumes because there are absolutely no restrictions in the format and the structure of data storage.
- Faster creation of document and maintenance: It is very simple to create a document and apart from this maintenance requires is almost nothing.
- Open formats: It has a very simple build process that uses XML, JSON, and its other forms.
- Built-in versioning: It has built-in versioning which means as the documents grow in size there might be a chance they can grow in complexity. Versioning decreases conflicts.

Disadvantages:

- Weak Atomicity: It lacks in supporting multi-document ACID transactions. A change in the document data model involving two collections will require us to run two separate queries i.e. one for each collection. This is where it breaks atomicity requirements.
- Consistency Check Limitations: One can search the collections and documents that are not connected to an author collection but doing this might create a problem in the performance of database performance.
- Security: Nowadays many web applications lack security which in turn results in the leakage of sensitive data. So it becomes a point of concern, one must pay attention to web app vulnerabilities.

Applications of Document Data Model :

- Content Management: These data models are very much used in creating various video streaming platforms, blogs, and similar services Because each is stored as a single document and the

database here is much easier to maintain as the service evolves over time.

- Book Database: These are very much useful in making book databases because as we know this data model lets us nest.
- Catalog: When it comes to storing and reading catalog files these data models are very much used because it has a fast reading ability if incase Catalogs have thousands of attributes stored.
- Analytics Platform: These data models are very much used in the Analytics Platform.

Columnar Data Model of NoSQL

Last Updated : 23 Jul, 2025

- The Columnar Data Model of NoSQL is important. NoSQL databases are different from SQL databases. This is because it uses a data model that has a different structure than the previously followed row-and-column table model used with relational database management systems (RDBMS). **NoSQL** databases are a flexible schema model which is designed to scale horizontally across many servers and is used in large volumes of data.

Columnar Data Model of NoSQL :

Basically, the relational database stores data in rows and also reads the data row by row, column store is organized as a set of columns. So if someone wants to run analytics on a small number of columns, one can read those columns directly without consuming memory with the unwanted data. Columns are somehow are of the same type and gain from more efficient compression, which makes reads faster than before. Examples of Columnar Data Model: Cassandra and Apache Hadoop Hbase.

Working of Columnar Data Model:

In Columnar Data Model instead of organizing information into rows, it does in columns. This makes them function the same way that tables work in relational databases. This type of data model is much more flexible obviously because it is a type of NoSQL database. The below example will help in understanding the Columnar data model:

Row-Oriented Table:

S.No.	Name	Cour se	Branch	I D
01.	Tanmay	B- Tech	Compute r	2
02.	Abhishe k	B- Tech	Electroni cs	5
03.	Samridd ha	B- Tech	IT	7
04.	Aditi	B- Tech	E & TC	8

Column - Oriented Table:

S.No.	Name	I D
01.	Tanmay	2
02.	Abhishe k	5
03.	Samridd ha	7

S.No.	Name	I D
04.	Aditi	8

S.No.	Cour se	I D
01.	B- Tech	2
02.	B- Tech	5
03.	B- Tech	7
04.	B- Tech	8

S.No.	Branch	I D
01.	Compute r	2
02.	Electroni cs	5
03.	IT	7
04.	E & TC	8

Columnar Data Model uses the concept of key space, which is like a schema in relational models.

Advantages of Columnar Data Model :

- Well structured: Since these data models are good at compression so these are very structured or well organized in terms of storage.

- Flexibility: A large amount of flexibility as it is not necessary for the columns to look like each other, which means one can add new and different columns without disrupting the whole database
- Aggregation queries are fast: The most important thing is aggregation queries are quite fast because a majority of the information is stored in a column. An example would be Adding up the total number of students enrolled in one year.
- Scalability: It can be spread across large clusters of machines, even numbering in thousands.
- Load Times: Since one can easily load a row table in a few seconds so load times are nearly excellent.

Disadvantages of Columnar Data Model:

- Designing indexing Schema: To design an effective and working schema is too difficult and very time-consuming.
- Suboptimal data loading: incremental data loading is suboptimal and must be avoided, but this might not be an issue for some users.
- Security vulnerabilities: If security is one of the priorities then it must be known that the Columnar data model lacks inbuilt security features in this case, one must look into relational databases.
- Online Transaction Processing (OLTP): Online Transaction Processing (OLTP) applications are also not compatible with columnar data models because of the way data is stored.

Applications of Columnar Data Model:

- Columnar Data Model is very much used in various Blogging Platforms.
- It is used in Content management systems like WordPress, Joomla, etc.
- It is used in Systems that maintain counters.
- It is used in Systems that require heavy write requests.

- It is used in Services that have expiring usage.

Introduction to Graph Database on NoSQL

Last Updated : 23 Jul, 2025

- A graph database is a type of NoSQL database that is designed to handle data with complex relationships and interconnections. In a graph database, data is stored as nodes and edges, where nodes represent entities and edges represent the relationships between those entities.
 1. Graph databases are particularly well-suited for applications that require deep and complex queries, such as social networks, recommendation engines, and fraud detection systems. They can also be used for other types of applications, such as supply chain management, network and infrastructure management, and bioinformatics.
 2. One of the main advantages of graph databases is their ability to handle and represent relationships between entities. This is because the relationships between entities are as important as the entities themselves, and often cannot be easily represented in a traditional relational database.
 3. Another advantage of graph databases is their flexibility. Graph databases can handle data with changing structures and can be adapted to new use cases without requiring significant changes to the database schema. This makes them particularly useful for applications with rapidly changing data structures or complex data requirements.
 4. However, graph databases may not be suitable for all applications. For example, they may not be the best choice for applications that

require simple queries or that deal primarily with data that can be easily represented in a traditional relational database. Additionally, graph databases may require more specialized knowledge and expertise to use effectively

Types of Graph Databases:

- Property Graphs: These graphs are used for querying and analyzing data by modelling the relationships among the data. It comprises of vertices that has information about the particular subject and edges that denote the relationship. The vertices and edges have additional attributes called properties.
- RDF Graphs: It stands for Resource Description Framework. It focuses more on data integration. They are used to represent complex data with well defined semantics. It is represented by three elements: two vertices, an edge that reflect the subject, predicate and object of a sentence. Every vertex and edge is represented by URI(Uniform Resource Identifier).

When to Use Graph Database?

- Graph databases should be used for heavily interconnected data.
- It should be used when amount of data is larger and relationships are present.
- It can be used to represent the cohesive picture of the data.

How Graph and Graph Databases Work?

Graph databases provide graph models They allow users to perform traversal queries since data is connected. Graph algorithms are also applied to find patterns, paths and other relationships this enabling more analysis of the data. The algorithms help to explore the neighboring nodes, clustering of vertices analyze relationships and patterns. Countless joins are not required in this kind of database.

Example of Graph Database:

- Recommendation engines in E commerce use graph databases to provide customers with accurate recommendations, updates about

new products thus increasing sales and satisfying the customer's desires.

- Social media companies use graph databases to find the "friends of friends" or products that the user's friends like and send suggestions accordingly to user.
- To detect fraud Graph databases play a major role. Users can create graph from the transactions between entities and store other important information. Once created, running a simple query will help to identify the fraud.

Advantages of Graph Database:

- Potential advantage of Graph Database is establishing the relationships with external sources as well
- No joins are required since relationships is already specified.
- Query is dependent on concrete relationships and not on the amount of data.
- It is flexible and agile.
- it is easy to manage the data in terms of graph.
- Efficient data modeling: Graph databases allow for efficient data modeling by representing data as nodes and edges. This allows for more flexible and scalable data modeling than traditional relational databases.
- Flexible relationships: Graph databases are designed to handle complex relationships and interconnections between data elements. This makes them well-suited for applications that require deep and complex queries, such as social networks, recommendation engines, and fraud detection systems.
- High performance: Graph databases are optimized for handling large and complex datasets, making them well-suited for applications that require high levels of performance and scalability.

- Scalability: Graph databases can be easily scaled horizontally, allowing additional servers to be added to the cluster to handle increased data volume or traffic.
- Easy to use: Graph databases are typically easier to use than traditional relational databases. They often have a simpler data model and query language, and can be easier to maintain and scale.

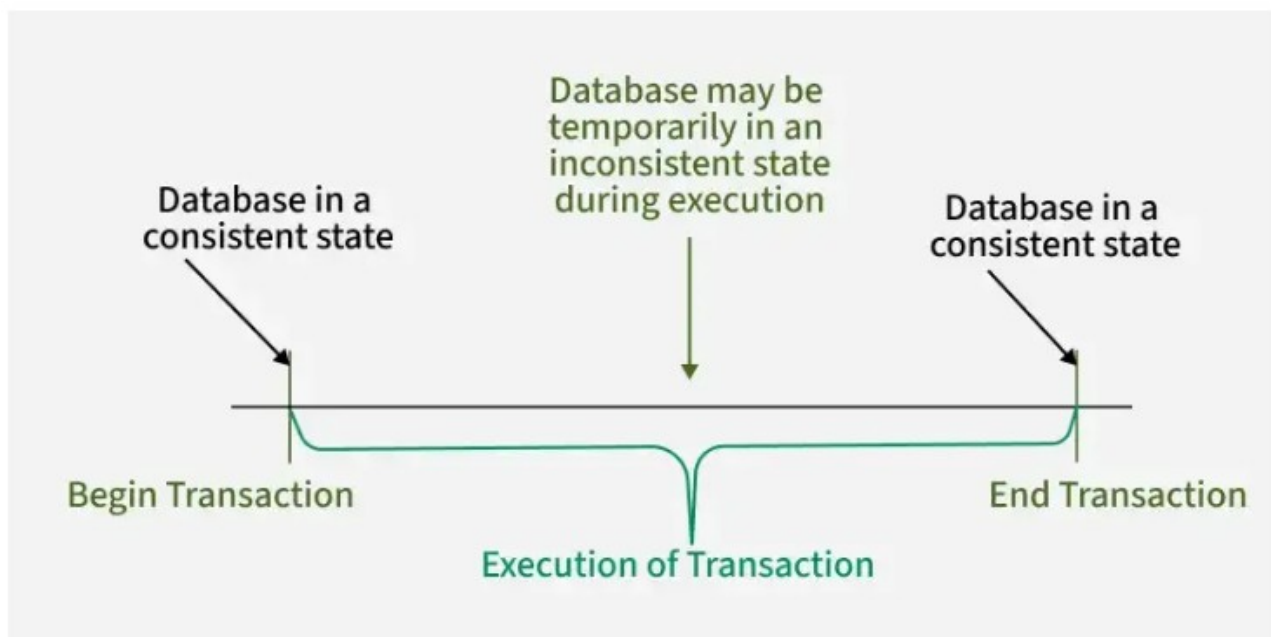
Disadvantages of Graph Database:

- Often for complex relationships speed becomes slower in searching.
- The query language is platform dependent.
- They are inappropriate for transactional data
- It has smaller user base.
- Limited use cases: Graph databases are not suitable for all applications. They may not be the best choice for applications that require simple queries or that deal primarily with data that can be easily represented in a traditional relational database.
- Specialized knowledge: Graph databases may require specialized knowledge and expertise to use effectively, including knowledge of graph theory and algorithms.
- Immature technology: The technology for graph databases is relatively new and still evolving, which means that it may not be as stable or well-supported as traditional relational databases.
- Integration with other tools: Graph databases may not be as well-integrated with other tools and systems as traditional relational databases, which can make it more difficult to use them in conjunction with other technologies.
- Overall, graph databases on NoSQL offer many advantages for applications that require complex and deep relationships between data elements. They are highly flexible, scalable, and performant,

and can handle large and complex datasets. However, they may not be suitable for all applications, and may require specialized knowledge and expertise to use effectively.

Transaction in DBMS

A transaction refers to a sequence of one or more operations (such as read, write, update, or delete) performed on the database as a single logical unit of work. A transaction ensures that either all the operations are successfully executed (committed) or none of them take effect (rolled back). Transactions are designed to maintain the integrity, consistency and reliability of the database, even in the case of system failures or concurrent access.



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.

Example: Let's consider an online banking application:

Transaction: When a user performs a **money transfer**, several operations occur, such as:

- **Reading** the account balance of the sender.
- **Writing** the deducted amount from the sender's account.
- **Writing** the added amount to the recipient's account.

In a **transaction**, all these steps should either complete successfully or, if any error occurs, the database should **rollback** to its previous state, ensuring no partial data is written to the system.

Table Format Summary

Time	T1	T2	Value of X
t0			100
t1	Read(X=100)		100
t2	X = 110		100
t3		Read(X=100)	100
t4		X = 200	100
t5	Write(X=110)		110
t6		Write(X=200)	200

Facts about Database Transactions

- 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.

Operations of Transaction

A user can make different types of requests to access and modify the contents of a database. So, we have different types of operations relating to a transaction. They are discussed as follows:

1) Read(X)

A read operation is used to read the value of a particular database element **X** and stores it in a temporary buffer in the main memory for further actions such as displaying that value.

Example: For a banking system, when a user checks their balance, a Read operation is performed on their **account** balance:

```
SELECT balance FROM accounts WHERE account_id = 'A123';
```

This updates the balance of the user's account after withdrawal.

2) Write(X)

A write operation stores updated data from main memory back to the database. It usually follows a read, where data is fetched, modified (e.g., arithmetic changes), and then written back to save the updated value.

Example: For the banking system, if a user withdraws money, a **Write** operation is performed after the balance is updated:

```
UPDATE accounts SET balance = balance - 100 WHERE  
account_id = 'A123';
```

This updates the balance of the user's account after withdrawal.

3) Commit

This operation in transactions is used to maintain integrity in the database. Due to some failure of power, hardware, or software, etc., a transaction might get interrupted before all its operations are completed. This may cause ambiguity in the database, i.e. it might get inconsistent before and after the transaction.

Example: After a successful money transfer in a banking system, a **Commit** operation finalizes the transaction:

```
COMMIT;
```

Once the transaction is committed, the changes to the database are permanent, and the transaction is considered **successful**.

4) Rollback

A rollback undoes all changes made by a transaction if an error occurs, restoring the database to its last consistent state. It helps prevent data inconsistency and ensures safety.

Example: Suppose during the money transfer process, the system encounters an issue, like insufficient funds in the sender's account. In that case, the transaction is rolled back:

ROLLBACK;

This will undo all the operations performed so far and ensure that the database remains consistent.

Properties of Transaction

Transactions in DBMS must ensure data is accurate and reliable. They follow four key ACID properties:

- 1.**Atomicity:** A transaction is all or nothing. If any part fails, the entire transaction is rolled back. Example: While transferring money, both debit and credit must succeed. If one fails, nothing should change.
- 2.**Consistency:** A transaction must keep the database in a valid state, moving it from one consistent state to another. Example: If balance is ₹1000 and ₹200 is withdrawn, the new balance should be ₹800.
- 3.**Isolation:** Transactions run independently. One transaction's operations should not affect another's intermediate steps. Example: Two users withdrawing from the same account must not interfere with each other's balance updates.
- 4.**Durability:** Once a transaction is committed, its changes stay even if the system crashes. Example: After a successful transfer, the updated balance remains safe despite a power failure.

Dirty Read

Dirty read is a read of uncommitted data. If a particular row is modified by another running application and not yet committed, we also run an application to read the same row with the same uncommitted data. This is the state we say it as a dirty read.

The one main thing is that the dirty reader has to stop reading dirty.

We can try to use the shared locks to prevent other transactions to modify the row, if one is carried out here.

Example 2

Consider another example

Let T2 read the update value of X made by T1, but T1 fails and rolls back. So, T2 read an incorrect value of X.

T1	T2
read(x)	
X=X-5	
write(x)	
	read(x)
	x=x+5
	write(x)
ROLLBACK	
	commit

Transaction Schedules

When multiple transaction requests are made at the same time, we need to decide their order of execution. Thus, a transaction schedule can be defined as a chronological order of execution of multiple transactions. Example: After a successful transfer, the updated balance remains safe despite a power failure.

There are broadly **two types** of transaction schedules discussed as follows:

i) Serial Schedule

In a serial schedule, transactions execute one at a time, ensuring database consistency but increasing waiting time and reducing system throughput. To improve throughput while maintaining consistency, concurrent schedules with strict rules are used, allowing safe simultaneous execution of transactions.

ii) Non-Serial Schedule

Non-serial schedule is a type of transaction schedule where multiple transactions are executed concurrently, interleaving their operations, instead of running one after another. It improves system efficiency but requires concurrency control to maintain database consistency.

Serializability in DBMS

Last Updated : 02 Aug, 2025

Serializability is a concept in DBMS that ensures concurrent transaction execution results in a consistent database, just like some serial (one-by-one) execution.

Non-serial Schedule

A non-serial schedule allows transactions to run concurrently and may access the same data. To ensure database consistency, it must be serializable, meaning it should produce the same result as some serial (one-by-one) execution.

Example:

Transaction-1	Transaction-2
R(a)	
W(a)	
	R(b)
	W(b)
R(b)	
	R(a)
W(b)	
	W(a)

We can observe that Transaction-2 begins its execution before Transaction-1 is finished, and they are both working on the same data, i.e., "a" and "b", interchangeably. Where "R"-Read, "W"-Write

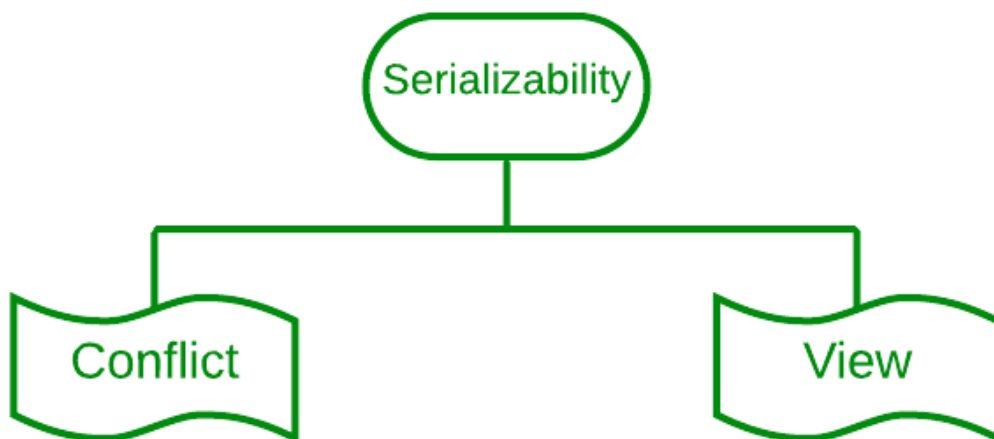
Serializability testing

We can utilize the Serialization Graph or Precedence Graph to examine a schedule's serializability. A schedule's full transactions are organized into a Directed Graph, what a serialization graph is.

It can be described as a Graph $G(V, E)$ with vertices $V = "V1, V2, V3, ..., Vn"$ and directed edges $E = "E1, E2, E3, ..., En"$. One of the two operations—READ or WRITE—performed by a certain transaction is contained in the collection of edges. Where $T_i \rightarrow T_j$, means Transaction- T_i is either performing read or write before the transaction- T_j .

Types of Serializability

There are two ways to check whether any non-serial schedule is serializable.



Types of Serializability - Conflict & View

1. Conflict serializability

Conflict serializability ensures database consistency by checking if a non-serial schedule can be rearranged into a serial order by swapping non-conflicting operations. It prevents conflicting operations (like read/write on the same data) from executing at the same time.

What is Conflict Equivalency?

For two schedules (S1 and S2) to be conflict equivalent, they must satisfy the following:

- **Same Conflicting Operations:** The same conflicting operations (e.g., reads and writes on the same data item) must occur in both schedules in the same order. Example: If t1 writes A before t2 in S1, then t1 must also write A before t2 in S2.
- **Non-Conflicting Operations:** Operations that do not conflict (e.g., reading different data items) should not affect the order of the schedules. Example: If t1 reads B and t2 reads C in S1, then t1 should still read B and t2 should still read C in S2.
- **Preserving Transaction Order:** The order of conflicting operations should be the same in both schedules to maintain consistency. Example: If t1 writes A and t2 reads A in S1, t2 must read A after t1 in S2.

In simple terms, conflict equivalency ensures that conflicting operations happen in the same order in both schedules, while non-conflicting ones can appear in any order.

Example 1: **Conflict Serializable** Schedule

(But **not** view serializable in general unless consistent reads and writes)

Transactions

- **T1:** Read(X), Write(X)
- **T2:** Read(X), Write(X)

Schedule Summary Table (Conflict Serializable)

Time	T1	T2	Value of X
t0			100
t1	Read(X=100)		100
t2	X = 110		100
t3	Write(X=110)		110
t4		Read(X=110)	110
t5		X = 220	110
t6		Write(X=220)	220

Explanation

- **Final X = 220**
- This schedule is **conflict equivalent** to serial schedule:
T1 → T2 (T1 completes all conflicting operations before T2 starts)
- So, it's **conflict serializable**

2. View Serializability

View serializability ensures that a non-serial schedule results in the same final outcome as a serial schedule, maintaining database consistency.

To further understand view serializability in DBMS, we need to understand the schedules S1 and S2. The two transactions T1 and T2 should be used to establish these two schedules. Each schedule must follow the three transactions in order to retain the equivalent of the transaction. These three circumstances are listed below.

1. **Same Transactions:** Both schedules must include the same set of transactions.
2. **Same Writes:** Each data item must be written by the same transaction in both schedules.
3. **Same Reads:** Each read must read the same value (from the same write) in both schedules.

What is View Equivalency?

Schedules (S1 and S2) must satisfy these two requirements in order to be view equivalent:

1. The same data must be read first in both schedules. Example: If t1 reads A in S1, it must do the same in S2.
2. The same data must be used for the final write. Example: If t1 updates A last in S1, it should do the same in S2.
3. The middle sequence should also match. Example: If t1 reads A and t2 updates A in S1, the same order should occur in S2.

Example 2: View Serializable but Not Conflict Serializable

Time	T1	T2	Value of X
t0			100
t1		Read(X=100)	100

Time	T1	T2	Value of X
t2	Read(X=100)		100
t3	X = 110		100
t4		X = 200	100
t5		Write(X=200)	200
t6	Write(X=110)		110

Explanation

- T1 and T2 both read initial value (100) — allowed in view serializability.
- Final value is **110** — last write from T1.
- **This is NOT conflict serializable**, because conflicting operations are out of order:
 - T2 reads X before T1 writes → violates T1 → T2 conflict order.
 - But this is **view serializable** if:
 - Same **read-from** relationships.
 - Same **final write**.

Anomalies with Interleaved Execution (in Transaction Processing)

When transactions are executed **concurrently (interleaved)** without proper control, **anomalies** (i.e., unexpected or incorrect results) may occur.

Here are the **main anomalies**:

1. Lost Update

Problem:

Two transactions read the same data and **both update it**, but the **first update is lost**.

Example:

Step	T1	T2	Value of X
t1	Read(X=100)		100
t2		Read(X=100)	100
t3	X = 100 + 10		
t4		X = 100 × 2	
t5	Write(X=110)		110
t6		Write(X=200)	200 (T1's update lost)

2. Dirty Read (Uncommitted Dependency)

Problem:

A transaction reads **uncommitted changes** made by another transaction, which may later **roll back**.

Example:

Step	T1	T2	Value of X
t1	Write(X=500)		500
t2		Read(X=500)	500
t3	Abort		
t4		Uses invalid data	(used dirty data)

3. Unrepeatable Read

Problem:

A transaction reads the **same data twice** and gets **different values** due to another transaction modifying it.

Example:

Step	T1	T2	Value of X
t1	Read(X=100)		100
t2		Write(X=300)	300
t3	Read(X=300)		(changed mid-way)

4. Phantom Read

Problem:

A transaction gets **different sets of rows** in the **same query** when repeated, because another transaction **added/deleted rows**.

Example:

- T1: SELECT * FROM Employees WHERE salary > 5000
- T2: INSERT INTO Employees VALUES (E99, salary=6000)
- T1: Runs the same query again, sees **extra row** (phantom)

Recoverable Schedule

A recoverable schedule ensures that the database can return to a consistent state after a transaction failure.

In this type of schedule:

1. A transaction cannot use (read) data updated by another transaction until that transaction commits.
2. If a transaction fails before committing, all its changes must be rolled back, and any other transactions that have used its uncommitted data must also be rolled back.

Recoverable Schedule prevents data inconsistencies and ensures that no transaction commits based on unverified changes, making recovery easier and safer.

Example:

S1: R1(x), W1(x), R2(x), R1(y), R2(y), W2(x), W1(y), C1, C2;

The given schedule follows the order of **Ti → Tj ⇒ C1 → C2**.

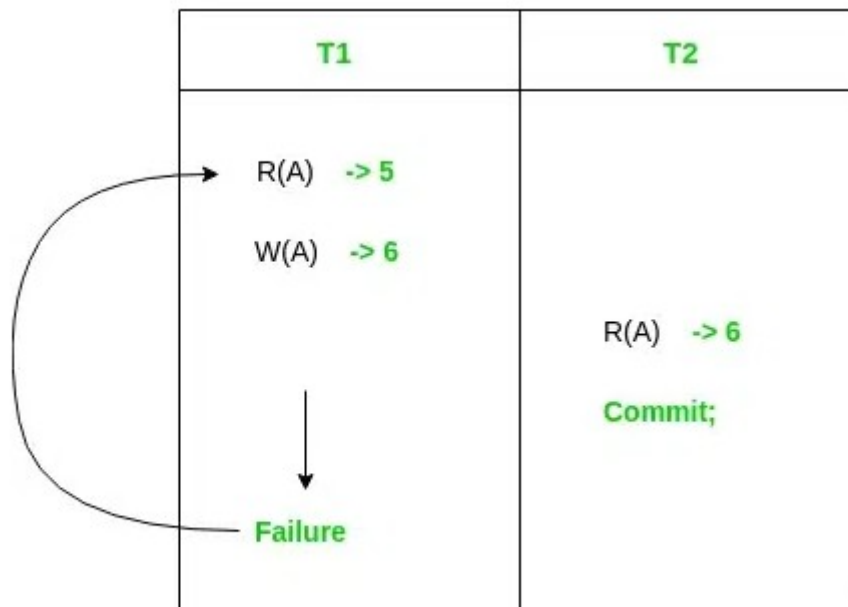
Transaction T1 is executed before T2 hence there is no chance of conflict occurring. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of the first transaction performed the first update on data item x, hence given schedule is recoverable.

Let us see an example of an unrecoverable schedule to clear the concept more.

S2: R1(x), R2(x), R1(z), R3(x), R3(y), W1(x), W3(y), R2(y), W2(z), W2(y), C1, C2, C3;

Ti → Tj ⇒ C2 → C3 but W3(y) executed before W2(y) which leads to conflicts thus it must be committed before the T2 transaction. So given schedule is unrecoverable. if **Ti → Tj ⇒ C3 → C2** is given in the schedule then it will become a recoverable schedule.

Example:



Cascadeless Schedule

A cascade-less schedule ensures that if one transaction fails, it does not cause multiple other transactions to roll back.

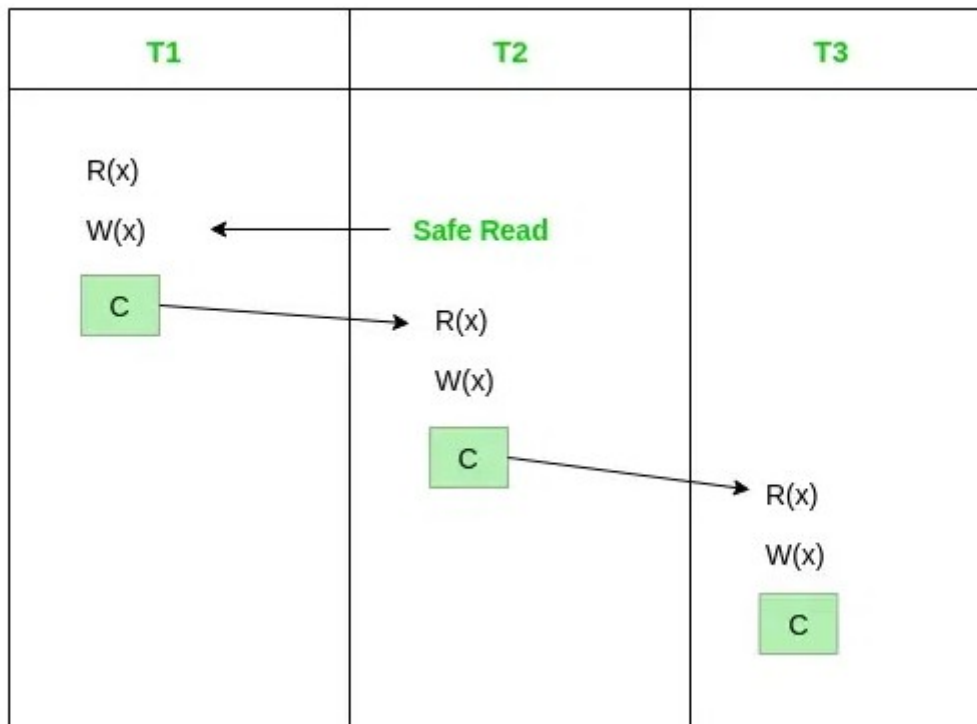
In this type of schedule, a transaction is not allowed to use (read) uncommitted data from another transaction until that transaction is fully committed. This prevents **cascading rollbacks**, where a failure in one transaction forces many others to undo their changes. If a transaction fails before committing, only its own changes are undone, but other transactions that have read its data do not need to roll back, making the recovery process simpler and more efficient.

Example:

S3: R1(x), R2(z), R3(x), R1(z), R2(y), R3(y), W1(x), C1, W2(z), W3(y), W2(y), C3, C2;

In this schedule W3(y) and W2(y) overwrite conflicts and there is no read, therefore given schedule is cascade less schedule.

Special Case: A committed transaction desired to abort. As given below all the transactions are reading committed data hence it's cascadeless schedule.



Strict Schedule

A schedule is strict if it is both recoverable and cascadeless.

A strict schedule ensures strong data consistency by following two key rules:

1. A transaction cannot use (read) uncommitted data from another transaction until that transaction commits.
2. A transaction cannot update (write) the same data modified by another transaction until the first one commits.

This means that if a transaction fails before committing, all its changes must be undone (rolled back), and any other transactions that used its uncommitted data must also be rolled back.

Strict schedule helps in maintaining a reliable and consistent database by preventing data conflicts and minimizing errors.

Example:

S4: R1(x), R2(x), R1(z), R3(x), R3(y), W1(x), C1, W3(y), C3, R2(y), W2(z), W2(y), C2;

In this schedule, no read-write or write-write conflict arises before committing hence its strict schedule:

T1	T2	T3
R1(x) R1(z) W1(x) C1;	R2(x) R2(y) W2(z) W2(y) C2;	R3(x) R3(y) W3(y) C3;

Concurrency Control in DBMS

Last Updated : 23 Jul, 2025

In a database management system (DBMS), allowing transactions to run concurrently has significant advantages, such as better system resource utilization and higher throughput. However, it is crucial that these transactions do not conflict with each other. The ultimate goal is to ensure that the database remains consistent and accurate. For instance, if two users try to book the last available seat on a flight at the same time, the system must ensure that only one booking succeeds. Concurrency control is a critical mechanism in DBMS that ensures the consistency and integrity of data when multiple operations are performed at the same time.

- Concurrency control is a concept in Database Management Systems (DBMS) that ensures multiple transactions can simultaneously access or modify data without causing errors or inconsistencies. It provides

mechanisms to handle the concurrent execution in a way that maintains **ACID properties**.

Concurrent Execution can lead to various challenges:

- Dirty Reads:** One transaction reads uncommitted data from another transaction, leading to potential inconsistencies if the changes are later rolled back.
- Lost Updates:** When two or more transactions update the same data simultaneously, one update may overwrite the other, causing data loss.
- Inconsistent Reads:** A transaction may read the same data multiple times during its execution, and the data might change between reads due to another transaction, leading to inconsistency.

Why is Concurrency Control Needed?

- Without Concurrency Control:** Transactions interfere with each other, causing issues like lost updates, dirty reads or inconsistent results.
- With Concurrency Control:** Transactions are properly managed (e.g., using locks or timestamps) to ensure they execute in a consistent, isolated manner, preserving data accuracy.

Concurrency control is critical to maintaining the accuracy and reliability of databases in multi-user environments. By preventing conflicts and inconsistencies during concurrent transactions, it ensures the database remains consistent and correct, even under high levels of simultaneous activity.

Lock Based Concurrency Control Protocol in DBMS

Last Updated : 02 Aug, 2025

A lock in DBMS controls concurrent access, allowing only one transaction to use a data item at a time. This ensures data integrity and prevents issues like lost updates or dirty reads during simultaneous transactions.

Lock Based Protocols in DBMS ensure that a transaction cannot read or write data until it gets the necessary lock. Here's how they work:

- These protocols prevent concurrency issues by allowing only one transaction to access a specific data item at a time.
- Locks help multiple transactions work together smoothly by managing access to the database items.
- Locking is a common method used to maintain the **serializability** of transactions.
- A transaction must acquire a read lock or write lock on a data item before performing any read or write operations on it.

Types of Lock

1.**Shared Lock (S):** Shared Lock is also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.

2.**Exclusive Lock (X):** Data item can be both read as well as written. This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

Read more about [Types of Locks](#).

Rules of Locking

The basic rules for Locking are given below:

Read Lock (or) Shared Lock(S)

- If a Transaction has a Read lock on a data item, it can read the item but not update it.
- If a transaction has a Read lock on the data item, other transaction can obtain Read Lock on the data item but no Write Locks.
- So, the Read Lock is also called a Shared Lock.

Write Lock (or) Exclusive Lock (X)

- If a transaction has a write Lock on a data item, it can both read and update the data item.
- If a transaction has a write Lock on the data item, then other transactions cannot obtain either a Read lock or write lock on the data item.
- So, the Write Lock is also known as Exclusive Lock.

Lock Compatibility Matrix

- A transaction can acquire a lock on a data item only if the requested lock is compatible with existing locks held by other transactions.
- Shared Locks (S):** Multiple transactions can hold shared locks on the same data item simultaneously.
- Exclusive Lock (X):** If a transaction holds an exclusive lock on a data item, no other transaction can hold any type of lock on that item.

- If a requested lock is not compatible, the requesting transaction must wait until all incompatible locks are released by other transactions.
- Once the incompatible locks are released, the requested lock is granted.

	S	X
S	✓	✗
X	✗	✗

Concurrency Control Protocols

Concurrency Control Protocols are the methods used to manage multiple transactions happening at the same time. They ensure that transactions are executed safely without interfering with each other, maintaining the accuracy and consistency of the database.

These protocols prevent issues like data conflicts, lost updates or inconsistent data by controlling how transactions access and modify data.

Types of Lock-Based Protocols

1. Simplistic Lock Protocol

It is the simplest method for locking data during a transaction. Simple lock-based protocols enable all transactions to obtain a lock on the data before inserting, deleting, or updating it. It will unlock the data item once the transaction is completed.

Example: Consider a database with a single data item $x = 10$.

Transactions:

- T1**: Wants to read and update x.
- T2**: Wants to read x.

Steps:

- 1.T1 requests an exclusive lock on X to update its value. The lock is granted.
- 2.T1 reads $X = 10$ and updates it to $X = 20$.
- 3.T2 requests a shared lock on X to read its value. Since T1 is holding an exclusive lock, T2 must wait.
- 4.T1 completes its operation and releases the lock.
- 5.T2 now gets the shared lock and reads the updated value $X = 20$.

This example shows how simplistic lock protocols handle concurrency but do not prevent problems like deadlocks or limits concurrency.

2. Pre-Claiming Lock Protocol

The Pre-Claiming Lock Protocol avoids deadlocks by requiring a transaction to request all needed locks before it starts. It runs only if all locks are granted; otherwise, it waits or rolls back.

Example: Consider two transactions T1 and T2 and two data items, x and y:

Transaction T1 declares that it needs:

- A write lock on x.
- A read lock on y.

Since both locks are available, the system grants them. T1 starts execution: It updates x. It reads the value of y.

While T1 is executing, Transaction T2 declares that it needs: However, since T1 already holds a write lock on x, T2's request is denied. T2 must wait until T1 completes its operations and releases the locks. A read lock on x

Once T1 finishes, it releases the locks on x and y. The system now grants the read lock on x to T2, allowing it to proceed.

This method is simple but may lead to inefficiency in systems with a high number of transactions.

Two Phase Locking Protocol

Last Updated : 30 Jul, 2025

The Two-Phase Locking (2PL) Protocol is a key technique used in DBMS to manage how multiple concurrent transactions access and modify data. When many users or processes interact with a database, it's important to ensure that data remains consistent and error-free. Without proper management, issues like data conflicts or corruption can occur

The Two-Phase Locking Protocol resolves this issue by defining clear rules for managing data locks. It divides a transaction into two phases:

1. **Growing Phase:** In this step, the transaction gathers all the locks it needs to access the required data. During this phase, it cannot release any locks.
2. **Shrinking Phase:** Once a transaction starts releasing locks, it cannot acquire any new ones. This ensures that no other transaction interferes with the ongoing process.

T 1	T2	
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	

T 1	T2	
8		Unlock(A)
9		Unlock(C)
1 0

This is a basic outline of a transaction that demonstrates how locking and unlocking work in the Two-Phase Locking Protocol (2PL).

Transaction T₁

- The growing Phase is from steps 1-3
- The shrinking Phase is from steps 5-7
- Lock Point at 3

Transaction T₂

- The growing Phase is from steps 2-6
- The shrinking Phase is from steps 8-9
- Lock Point at 6

Example of 2PL

Imagine a library system where multiple users can borrow or return books. Each action (like borrowing or returning) is treated as a transaction. Here's how the Two-Phase Locking Protocol (2PL) works, including the lock point:

User A wants to:

- 1.Check the availability of Book X.
- 2.Borrow Book X if it's available.
- 3.Update the library's record.

Growing Phase (Locks are Acquired):

- 1.User A locks Book X with a shared lock (S) to check its availability.
- 2.After confirming the book is available, User A upgrades the lock to an exclusive lock (X) to borrow it.
- 3.User A locks the library's record to update the borrowing details.

Lock Point: Once User A has acquired all the necessary locks (on Book X and the library record), the transaction reaches the lock point. No more locks can be acquired after this.

Shrinking Phase (Locks are Released):

1. User A updates the record and releases the lock on the library's record.
2. User A finishes borrowing and releases the exclusive lock on Book X.

This process ensures that no other user can interfere with Book X or the library record during the transaction, maintaining data accuracy and consistency. The lock point ensures that all locks are acquired before any are released, following the 2PL rules.

Drawbacks of 2PL

Two-phase locking (2PL) ensures that transactions are executed in the correct order by using two phases: acquiring and releasing locks.

However, it has some drawbacks:

- **Deadlocks:** Transactions can get stuck waiting for each other's locks, causing them to freeze indefinitely.
- **Cascading Rollbacks:** If one transaction fails, others that depend on it might also fail leading to inefficiency and potential data issues.
- **Lock Contention:** Too many transactions competing for the same locks can slow down the system, especially when many users are working at the same time.
- **Limited Concurrency:** The strict rules of 2PL can reduce how many transactions can run at once, resulting in slower performance and longer wait times.

Cascading Rollbacks in 2-PL

	T ₁	T ₂	T ₃
1	Lock-X(A)		
2	Read(A)		
3	Write(A)		
4	Lock-S(B) --->LP	Rollback	
5	Read(B)		Rollback
6	Unlock(A) , Unlock(B)		
7		Lock-X(A)--->LP	
8		Read(A)	
9		Write(A)	
10		Unlock(A)	
11			Lock-S(A) --->LP
12			Read(A)

FAIL — Rollback

LP - Lock Point

Read(A) in T₂ and T₃ denotes Dirty Read because of Write(A) in T₁.

The image illustrates a transaction schedule using the Two-Phase Locking (2PL) protocol, showing the sequence of actions for three transactions T1, T2 and T3.

Key Points:

1. Transaction T1:

- T1 acquires an exclusive lock (X) on data item A, performs a write operation on A and then acquires a shared lock (S) on B.
- T1 reaches its lock point (LP) after acquiring all locks.
- Eventually, T1 fails and a rollback is triggered, undoing its changes.

2. Transaction T2:

- T2 reads A after T1 writes A. This is called a **dirty read** because T1's write is not committed yet.
- When T1 rolls back, T2's operations become invalid, and it is also forced to rollback.

3. Transaction T3:

- T3 reads A after T2 reads A. Since T2 depends on the uncommitted changes of T1, T3 indirectly relies on T1's changes.
- When T1 rolls back, T3 is also forced to rollback even though it was not directly interacting with T1's operations.

Cascading Rollback Problem:

- Here, T2 and T3 both depend on uncommitted data from T1 (either directly or indirectly).
- When T1 fails and rolls back all dependent transactions (T2 and T3) must also rollback because their operations were based on invalid data.
- Cascading rollbacks waste system resources, reduce concurrency and lead to inefficiency.
- In large systems, this can significantly affect performance and make recovery more complex.

Categories of Two Phase Locking (Strict, Rigorous & Conservative)

Last Updated : 02 Aug, 2025

Two-Phase Locking (2PL) is a crucial technique in database management systems designed to ensure consistency and isolation during concurrent transactions. This article will cover the three main types of 2PL: strict 2PL, rigorous 2PL and conservative 2PL highlighting the differences in their locking protocols

There are three categories:

1. Strict 2-PL
2. Rigorous 2-PL
3. Conservative 2-PL

Strict Two-Phase Locking Protocol (Strict 2PL)

The Strict Two-Phase Locking Protocol is a variation of the Two-Phase Locking (2PL) protocol that adds a specific rule for releasing exclusive (X) locks. In addition to following the two phases (growing and shrinking), strict 2PL ensures that:

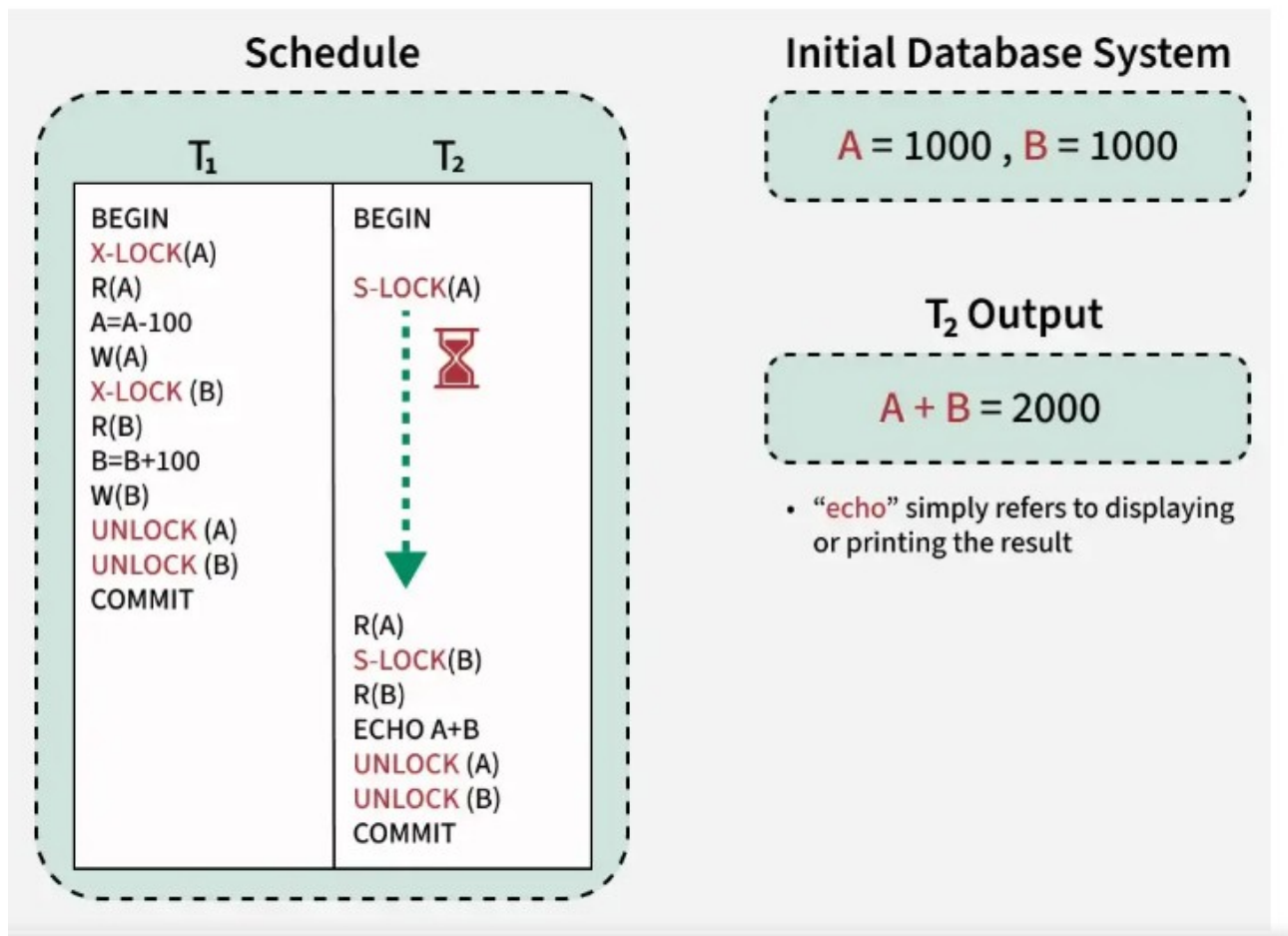
1. **All Exclusive (X) Locks are Held Until the Transaction**

Commits or Aborts:

- A transaction can acquire locks during the growing phase.
- It cannot release any exclusive locks until it has completed its operations and either committed or aborted.
- This rule prevents other transactions from accessing data modified by an ongoing transaction, ensuring data consistency.

2. **Benefits:**

- **Recoverable Schedule:** Ensures that if a transaction fails, no other transactions depend on its uncommitted changes.
- **Cascadeless Schedule:** Prevents cascading rollbacks, as other transactions cannot read uncommitted data.



The image demonstrates a Strict Two-Phase Locking (2PL) Protocol scenario with two transactions, T₁ and T₂ accessing and modifying two database accounts A and B. Here’s a step-by-step explanation:

Initial Database State:

- Account A=1000
- Account B=1000

Steps in the Schedule:

Transaction T₁:

1. **BEGIN:** T₁ starts and acquires an exclusive lock (X-Lock) on A. This means T₁ can read and modify A, and no other transaction can access A during this time.
2. T₁ reads A and subtracts 100 from it (A=900).
3. T₁ acquires an X-Lock on B and adds 100 to B (B=1100).
4. T₁ unlocks A and B only after committing. At this point, the changes made by T₁ become visible to other transactions.

Transaction T2:

- 1.BEGIN: T2 starts and tries to acquire a shared lock (S-Lock) on A to read its value.
- 2.T2 waits because T1 is holding an exclusive lock on A. This prevents T2 from reading the uncommitted data ensuring no dirty reads.
- 3.Once T1 commits and releases the lock on A, T2 acquires the S-Lock on A, reads $A=900$ and moves on.
- 4.T2 acquires an S-Lock on B, reads $B=1100$ and calculates $A+B=2000$.
- 5.T2 unlocks both A and B after committing.

Final Output:

- After T1 commits, T2 reads the updated values of A and B and the result $A+B=2000$ reflects the committed changes.

Rigorous Two-Phase Locking (Rigorous 2PL)

Rigorous 2PL is a stricter version of the Two-Phase Locking (2PL) protocol. In this protocol:

- 1.All locks (both shared and exclusive) are held by a transaction until it either commits or aborts.
- 2.Lockes are only released after the transaction finishes, ensuring that no other transaction can access the locked data until the first transaction is fully complete.

This means that no other transaction can read or write the data being used by the current transaction until it is committed or rolled back. This ensures data consistency and avoids issues like dirty reads or cascading rollbacks.

T1	T2
lock-exclusive(A)	
lock-exclusive(B)	
read(A)	
A = A + 50	
write(A)	
read(B)	
B = B + 100	
write(B)	
	lock-exclusive(A)
	read(A)
	A = A + 50
	write(A)
commit	
unlock(A)	
unlock(B)	
	commit
	unlock(A)

Sequence of transactions

Explanation of the above transactions is given below:

T1 (left column)

- 1.lock-exclusive(A) → T1 locks data item A exclusively.
- 2.lock-exclusive(B) → T1 also locks B exclusively.
- 3.Reads and modifies A: read(A) → A = A + 50 → write(A).
- 4.Reads and modifies B: read(B) → B = B + 100 → write(B).
- 5.commit is performed before releasing any locks.
- 6.unlock(A) and unlock(B) happen after commit.

Follows Rigorous 2PL: No locks are released before the transaction commits.

T2 (right column)

- 1.lock-exclusive(A) → T1 locks A exclusively.
- 2.Reads and modifies A: read(A) → A = A + 50 → write(A).
- 3.commit is executed.
- 4.unlock(A) is done after commit.

Follows Rigorous 2PL: Even with a single item, lock is held until commit.

Hence, it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still, **Deadlocks** are possible!

Note: The difference between Strict 2-PL and Rigorous 2-PL is that Rigorous is more restrictive, it requires both Exclusive and Shared locks to be held until after the Transaction commits and this is what makes the implementation of Rigorous 2-PL easier.

Conservative Two-Phase Locking (Conservative 2PL)

Conservative 2PL is a variation of the Two-Phase Locking protocol designed to prevent deadlocks entirely. It is also known as the Static-2PL. The key feature of Conservative 2PL is that a transaction acquires all the locks it needs at the very beginning of the transaction before it starts executing. If the transaction cannot acquire all the required locks, it does not proceed and waits until all locks become available.

T ₁	T ₂
lock-exclusive(a)	
lock-exclusive(b)	
read(a)	
a=a+50	
write(a)	
unlock(a)	
	lock-exclusive(a)
	read(a)
	a=a+50
	write(a)
	unlock(a)
read(b)	
b=b+100	
write(b)	
unlock(b)	

This example demonstrates Conservative Two-Phase Locking (Conservative 2PL), where each transaction acquires all the locks it needs at the beginning of the transaction to avoid deadlocks. Here's a detailed explanation of how it aligns with the principles of Conservative 2PL:

Transaction T1 (left column):

- 1.**Lock-Exclusive(A):** T1 acquires an exclusive lock on A before performing any operation on it.
- 2.**Lock-Exclusive(B):** Before proceeding further, T1 also acquires an exclusive lock on B ensuring that both resources required by T1 are locked at the start.
- 3.**Operations on A:** T1 reads the value of A, increments it by 50 and writes the updated value back to A.
- 4.**Unlock(A):** After completing all operations on A, T1 releases the lock on A.
- 5.**Operations on B:** T1 reads the value of B, increments it by 100 and writes the updated value back to B.
- 6.**Unlock(B):** T1 releases the lock on B after finishing all operations.

Transaction T2 (right column):

- 1.**Lock-Exclusive(A):** T2 attempts to acquire an exclusive lock on A at the start of the transaction. If T1 still holds the lock on A T2 waits until T1 releases it.
- 2.**Operations on A:** Once T2 acquires the lock it reads A, increments it by 50 and writes the updated value back to A.
- 3.**Unlock(A):** T2 releases the lock on A, after completing its operations.

Advantages of Conservative 2PL:

- 1.**Deadlock-Free:** Deadlocks are avoided because a transaction either acquires all required locks at once or waits until it can.
- 2.**Efficient Use of Locks:** Transactions do not hold partial locks while waiting for others, reducing contention.
- 3.**Consistency and Serializability:** Like all 2PL variants, it ensures consistent and serializable schedules.

Timestamp based Concurrency Control

Timestamp-based concurrency control is a technique used in database management systems (DBMS) to ensure serializability of transactions without using locks. It uses timestamps to determine the order of transaction execution and ensures that conflicting operations follow a consistent order.

Each transaction T is assigned a unique timestamp $TS(T)$ when it enters the system. This timestamp determines the transaction's place in the execution order.

Timestamp Ordering Protocol

The Timestamp Ordering Protocol enforces that older transactions (with smaller timestamps) are given higher priority. This prevents conflicts and ensures the execution is serializable and deadlock-free.

For example:

- If Transaction T_1 enters the system first, it gets a timestamp $TS(T_1) = 007$ (assumption).
- If Transaction T_2 enters after T_1 , it gets a timestamp $TS(T_2) = 009$ (assumption).

This means T_1 is "older" than T_2 and T_1 should execute before T_2 to maintain consistency.

Features of Timestamp Ordering Protocol:

1. Transaction Priority:

- Older transactions (those with smaller timestamps) are given higher priority.

- For example, if transaction T1 has a timestamp of 007 times and transaction T2 has a timestamp of 009 times, T1 will execute first as it entered the system earlier.

2. Early Conflict Management: Unlike lock-based protocols, which manage conflicts during execution, timestamp-based protocols start managing conflicts as soon as a transaction is created.

3. Ensuring Serializability: The protocol ensures that the schedule of transactions is serializable. This means the transactions can be executed in an order that is logically equivalent to their timestamp order.

How Timestamp Ordering Works

Each **data item X** in the database keeps two timestamps:

- **W_TS(X):** Timestamp of the last transaction that wrote to X
- **R_TS(X):** Timestamp of the last transaction that read from X

Basic Timestamp Ordering

The Basic TO Protocol works by comparing the timestamp of the current transaction with the timestamps on the data items it wants to **read/write**:



- Suppose, if an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned timestamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.
- The protocol manages concurrent execution such that the timestamps determine the serializability order.

- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.
- Whenever some Transaction T tries to issue a $R_item(X)$ or a $W_item(X)$, the Basic TO algorithm compares the timestamp of T with $R_TS(X)$ & $W_TS(X)$ to ensure that the Timestamp order is not violated.

Two Basic TO protocols are discussed below:

1. Whenever a Transaction T issues a **$R_item(X)$** operation, check the following conditions:

- If **$W_TS(X) > TS(T)$** → **Abort T** (conflict: a newer write already occurred)
- Else** → **Allow read** and set **$R_TS(X) = \max(R_TS(X), TS(T))$**

2. Whenever a Transaction T issues a **$W_item(X)$** operation, check the following conditions:

- If **$R_TS(X) > TS(T)$** or **$W_TS(X) > TS(T)$** → **Abort T** (conflict: older transaction overwriting newer read/write)
- Else** → **Allow write** and set **$W_TS(X) = TS(T)$**

When conflicts are detected, the younger transaction is aborted and rolled back.

Timestamp Based Protocol

Consider two transactions T1 and T2 where Transaction T2 came after Transaction T1

$TS(T_i)$ = Timestamp of Transaction T_i

$RTS(x)$ = Maximum Timestamp of a Transaction that read x

$WTS(x)$ = Maximum Timestamp of a Transaction that wrote x

$TS(T1) = 10$

$TS(T2) = 20$

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

Timestamp Based Protocol

$TS(T1) = 10$

$TS(T2) = 20$

$RTS(A) = 10$

$WTS(A) = 10$

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

Timestamp Based Protocol

$TS(T1) = 10$

$TS(T2) = 20$

- Conflicting Operation Write(A) and Read (A)
- $WTS(A) < TS(T2)$
∴ $RTS(A) = \max(10, TS(T2)) = 20$
- Also, $WTS(A) = 20$

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

Timestamp Based Protocol

$TS(T1) = 10$

$TS(T2) = 20$

$RTS(B) = 20$

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

Timestamp Based Protocol

$TS(T1) = 10$

$TS(T2) = 20$

- Conflicting Operation Write(B) and Read (A)
- Write(B) operation of Transaction T1 is conflicting with Read (B) operation of transaction T2.
- $RTS(B) > TS(T1) \therefore$ Rollback

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

Rollback



Strict Timestamp Ordering Protocol

The Strict Timestamp Ordering Protocol is an enhanced version that avoids cascading rollbacks by delaying operations until it's safe to execute them.

Key Features

- **Strict Execution Order:** Transactions must execute in the exact order of their timestamps. Operations are delayed if executing them would violate the timestamp order, ensuring a strict schedule.
- **No Cascading Rollbacks:** To avoid cascading aborts, a transaction must delay its operations until all conflicting operations of older transactions are either committed or aborted.
- **Consistency and Serializability:** The protocol ensures conflict-serializable schedules by following strict ordering rules based on transaction timestamps.

Rules for Read Operation $R_item(X)$:

T can read X only if:

- $W_TS(X) \leq TS(T)$ and
- The transaction that last wrote X has committed

Rules for Write Operation $W_item(X)$:

T can write X only if:

- $R_TS(X) \leq TS(T)$ and $W_TS(X) \leq TS(T)$ and
- All previous readers/writers of X have committed

If these conditions aren't met, the operation is delayed (not aborted immediately).

Advantages	Disadvantages
Conflict-Serializable: Maintains a correct execution order	Cascading Rollbacks (in Basic TO protocol)
Deadlock-Free: No locks, so no circular waits	Starvation: Newer transactions may be delayed
Simple Conflict Resolution: Uses timestamps only	High Overhead: Constantly updating R_TS/W_TS
No Locking Needed: Avoids lock management complexity	Lower Throughput under high concurrency
Predictable Execution: Operations follow a known order	Delayed Execution in Strict TO for consistency

What is Multi-Version Concurrency Control (MVCC) in DBMS?

Last Updated : 21 Mar, 2024

Multi-Version Concurrency Control (MVCC) is a database optimization method, that makes redundant copies of records to allow for safe concurrent reading and updating of data. DBMS reads and writes are not blocked by one another while using MVCC. A technique called concurrency control keeps concurrent processes running to avoid read/write conflicts or other irregularities in a database.

Whenever it has to be updated rather than replacing the old one with the new information an MVCC database generates a newer version of the data item.

What is Multi-Version Concurrency Control (MVCC) in DBMS?

Multi-Version Concurrency Control is a technology, utilized to enhance databases by resolving concurrency problems and also data locking by preserving older database versions. When many tasks attempt to update the same piece of data simultaneously, MVCC causes a conflict and necessitates a retry from one or more of the processes.

Types of Multi-Version Concurrency Control (MVCC) in DBMS

Below are some types of Multi-Version Concurrency Control (MVCC) in DBMS

- **Timestamp-based MVCC:** The data visibility to transactions is defined by the unique timestamp assigned to each transaction that creates a new version of a record.
- **Snapshot-based MVCC:** This utilizes the database snapshot that is created at the beginning of a transaction to supply the information that is needed for the transaction.
- **History-based MVCC:** This Keeps track of every modification made to a record, making transaction rollbacks simple.
- **Hybrid MVCC:** This coordinates data flexibility and performance by combining two or more MVCC approaches.

How Does Multi-Version Concurrency Control (MVCC) in DBMS Works?

- In the database, every tuple has a version number. The tuple with the greatest version number can have a read operation done on it simultaneously.
- Only a copy of the record may be used for writing operations.
- While the copy is being updated concurrently, the user may still view the previous version.
- The version number is increased upon successful completion of the writing process.
- The upgraded version is now used for every new record operation and every time there is an update, this cycle is repeated

Advantages of Multi-Version Concurrency Control (MVCC) in DBMS

Below are some advantages of Multi-Version Concurrency Control in DBMS

- **The reduced read-and-write necessity for database locks:** The database can support many read-and-write transactions without locking the entire system thanks to MVCC.
- **Increased Concurrency:** This Enables several users to use the system at once.
- **Minimize read operation delays:** By enabling concurrent read operations, MVCC helps to cut down on read times for data.
- **Accuracy and consistency:** Preserves data integrity over several concurrent transactions.

Disadvantages of Multi-Version Concurrency Control (MVCC) in DBMS

Below are some disadvantages of Multi-Version Concurrency Control in DBMS

- **Overhead:** Keeping track of many data versions might result in overhead.
- **Garbage collecting:** To get rid of outdated data versions, MVCC needs effective garbage collecting systems.
- **Increase the size of the database:** Expand the capacity of the database since MVCC generates numerous copies of the records and/or tuples.
- **Complexity:** Compared to more straightforward locking systems, MVCC usage might be more difficult.

Validation Based Protocol in DBMS

Last Updated : 02 Aug, 2025

Validation-Based Protocol, also known as Optimistic Concurrency Control, is a technique in DBMS used to manage concurrent transactions without locking data during execution. It assumes that conflicts between transactions are rare and delays conflict detection until the validation phase just before committing.

Why It's Called Optimistic Concurrency Control

- Assumes low interference among transactions.
- Does not check for conflicts during execution.
- All updates are done on local copies.
- At the end, a validation check ensures data consistency before writing to the database.

Phases of Validation-Based Protocol

This protocol operates in three main phases:

1. Read Phase:

- The transaction reads data and performs calculations using temporary (local) variables.
- No changes are made to the database yet.

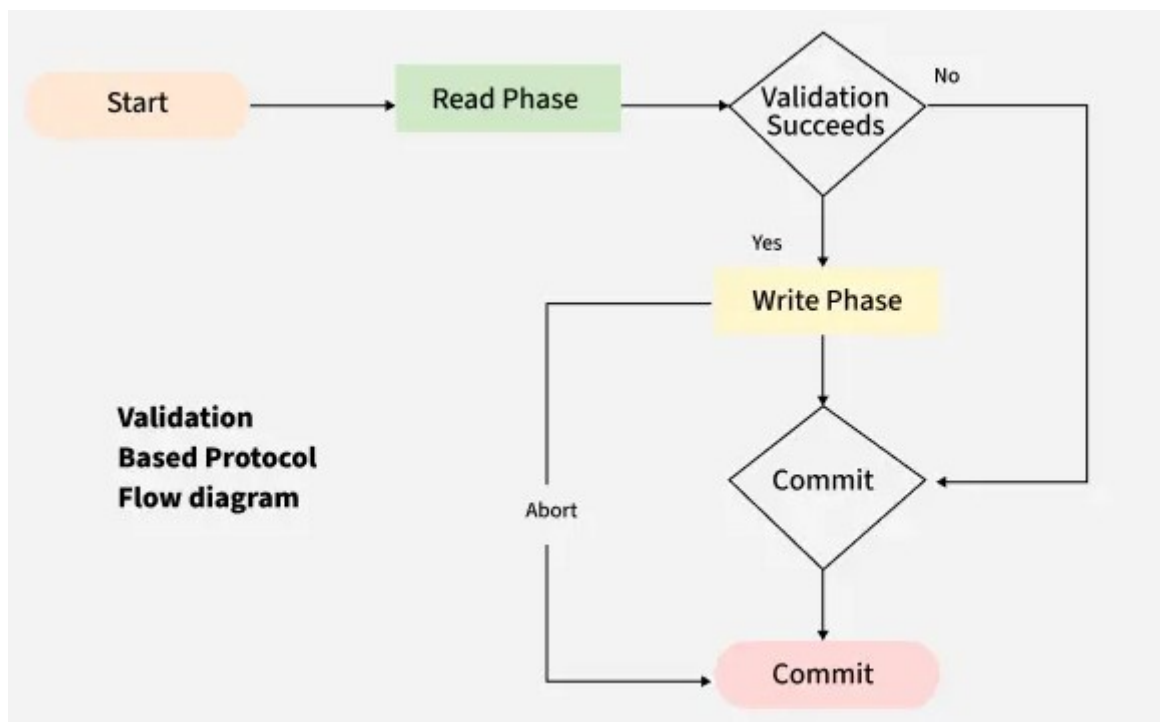
2. Validation Phase:

- Before committing, the transaction checks for conflicts with other concurrently running transactions.
- If no conflict is found (i.e., serializability is preserved), it proceeds to the write phase.
- If validation fails, the transaction is aborted and restarted.

3. Write Phase:

- If validation passes, the transaction writes changes to the database.
- If validation fails, the transaction is aborted and may be restarted.

The diagram below shows the working of the transaction based protocol.



Validation-based protocol delays conflict checks until the end. It works well in systems where most transactions don't interfere with each other.

- In low-conflict environments → high success rate.

- In high-conflict environments → frequent rollbacks, reduced efficiency.

Timestamps Used

Validation based protocols often use timestamps to manage the serialization order of transactions. Each transaction T_i is associated with three important timestamps:

- **Start (T_i):** The time when transaction T_i begins its execution (read phase).
- **Validation (T_i):** Time when T_i starts validation. Often used as its logical timestamp for serialization.
- **Finish (T_i):** The time when transaction T_i completes its write phase (if validation is successful).

Two more terms that we need to know are:

- **Write_set:** A set of transaction that contains all the write operations that T_i performs.
- **Read_set:** A set of transaction that contains all the read operations that T_i performs.

Example:

Let's consider two transactions where T_j starts before T_i , since $TS(T_j) < TS(T_i)$ so the validation phase succeeds in the Schedule A. It's noted that the final write operations to the database are performed only after the validation of both T_i and T_j . Since T_i reads the old values of $x(12)$ and $y(15)$ while print $(x+y)$ operation unless final write operation take place.

Time	T_j (Earlier TS)	T_i (Later TS)	Notes
T0		$r(x) \rightarrow x = 12$	T_i starts and reads $x = 12$
T1	$r(x) \rightarrow x = 12$		T_j starts and reads $x = 12$
T2	$x = x - 10$		T_j computes new value for x
T3		$r(y) \rightarrow y = 15$	T_i reads y

Time	Tj (Earlier TS)	Ti (Later TS)	Notes
T4		$y = y + 10$	Ti computes new value for y
T5	r(x)		Tj re-reads x (local copy still 12)
T6		<validate>	Ti enters validation phase
T7		$\text{print}(x + y) \rightarrow 27$	Uses local $x = 12, y = 15$
T8	<validate>		Tj enters validation phase
T9	w(x)		Tj writes x
T10	w(y)		Tj writes y

Deadlock in DBMS

Last Updated : 30 Jul, 2025

A deadlock occurs in a multi-user database environment when two or more transactions block each other indefinitely by each holding a resource the other needs. This results in a cycle of dependencies (circular wait) where no transaction can proceed.

Necessary Conditions of Deadlock

For a deadlock to occur, all four of these conditions must be true:

- Mutual Exclusion:** Only one **transaction** can hold a particular resource at a time.
- Hold and Wait:** The Transactions holding resources may request additional resources held by others.
- No Preemption:** The Resources cannot be forcibly taken from the transaction holding them.

- Circular Wait:** A cycle of transactions exists where each transaction is waiting for the resource held by the next transaction in the cycle.

Why Deadlocks Are a Problem?

- Transactions are stuck indefinitely.
- System throughput decreases as transactions remain blocked.
- Resources are held unnecessarily, preventing other operations.
- Can lead to performance bottlenecks or even system-wide standstill if not handled.

Real-Life Example

Transaction T1:

- Locks rows in Students
- Wants to update rows in Grades

Transaction T2:

- Locks rows in Grades
- Wants to update rows in Students

Both wait on each other and this results in **deadlock**, and all database activity comes to a standstill.

How to Handle Deadlocks

There are some approaches and by ensuring them, we can handle deadlocks. They are discussed below:

1. Deadlock Avoidance

Plan transactions in a way that prevents deadlock from occurring.

Methods:

- Access resources in the same order. For e.g., always access Students first, then Grades
- Use row-level locking and READ COMMITTED isolation level. It reduces chances, but doesn't eliminate deadlocks.completely

2. Deadlock Detection

If a transaction waits too long, the DBMS checks if it's part of a deadlock.

Method: Wait-For Graph

- Nodes: Transactions
- Edges: Waiting relationships
- If there's a cycle, a deadlock exists. It's mostly suitable for small to medium databases

3. Deadlock Prevention

For a large database, the deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that a deadlock never occurs. The DBMS analyzes the operations whether they can create a deadlock situation or not, If they do, that transaction is never allowed to be executed.

Deadlock prevention mechanism proposes two schemes:

1. Wait-Die Scheme (Non-preemptive)

- Older transactions are allowed to wait.
- Younger transactions are killed (aborted and restarted) if they request a resource held by an older one

For example:

- Consider two transaction- $T1 = 10$ and $T2 = 20$
- If $T1$ (older) wants a resource held by $T2 \rightarrow T1$ waits
- If $T2$ (younger) wants a resource held by $T1 \rightarrow T2$ dies and restarts

Prevents deadlock by not allowing a younger transaction to wait and form a wait cycle.

2. Wound-Wait Scheme (Preemptive)

- Older transactions are aggressive (preemptive) and can force younger ones to abort.
- Younger transactions must wait if they want a resource held by an older one.

For example:

- Consider two transaction- $T1 = 10$ and $T2 = 20$
- If $T1$ (older) wants a resource held by $T2 \rightarrow T2$ is killed, $T1$ proceeds.
- If $T2$ (younger) wants a resource held by $T1 \rightarrow T2$ waits

Prevents deadlock by not allowing younger transactions to block older ones.

The following table lists the differences between Wait-Die and Wound-Wait scheme prevention schemes:

Wait - Die	Wound -Wait
It is based on a non-preemptive technique.	It is based on a preemptive technique.
In this, older transactions must wait for the younger one to release its data items.	In this, older transactions never wait for younger transactions.
The number of aborts and rollbacks is higher in these techniques.	In this, the number of aborts and rollback is lesser.

Multiple Granularity

17 Mar 2025 | 3 min read

Let's start by understanding the meaning of granularity.

Granularity: It is the size of data item allowed to lock.

Multiple Granularity:

- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

For example: Consider a tree which has four levels of nodes.

- The first level or higher level shows the entire database.
- The second level represents a node of type area. The higher level database consists of exactly these areas.
- The area consists of children nodes which are known as files. No file can be present in more than one area.
- Finally, each file contains child nodes known as records. The file has exactly those records that are its child nodes. No records represent in more than one file.
- Hence, the levels of the tree starting from the top level are as follows:
 - 1.Database
 - 2.Area
 - 3.File
 - 4.Record

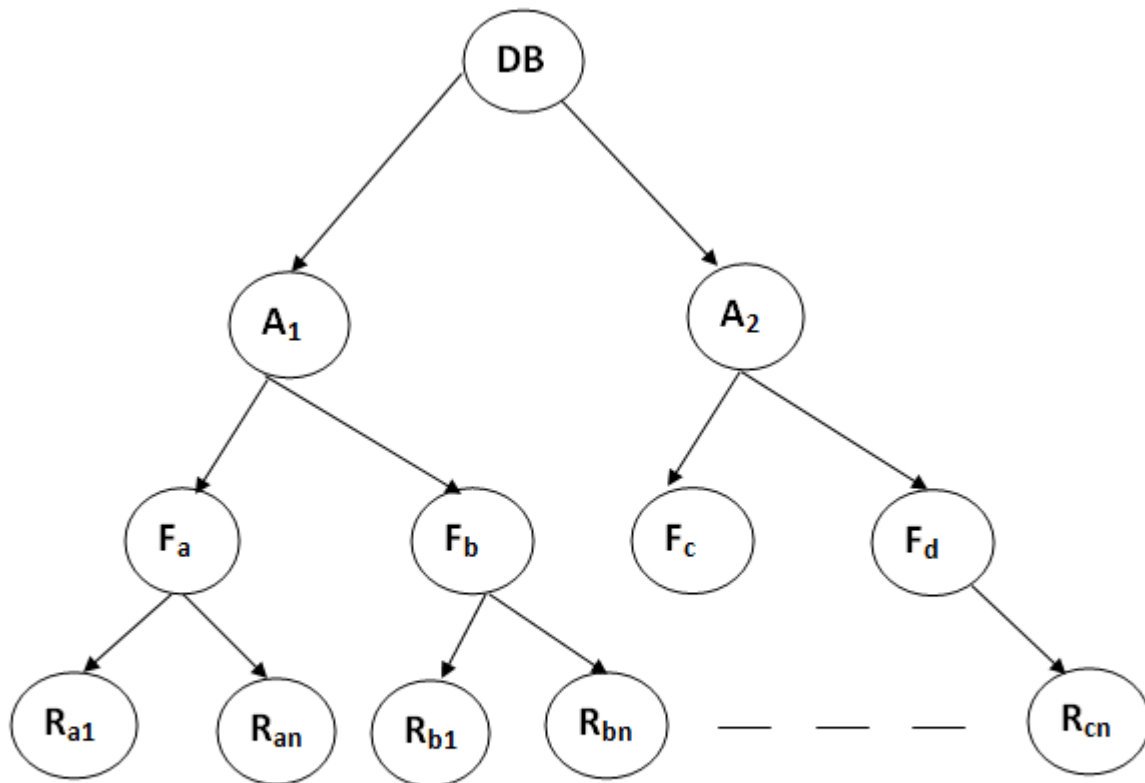


Figure: Multi Granularity tree Hierarchy

In this example, the highest level shows the entire database. The levels below are file, record, and fields.

There are three additional lock modes with multiple granularity:

Intention Mode Lock

Intention-shared (IS): It contains explicit locking at a lower level of the tree but only with shared locks.

Intention-Exclusive (IX): It contains explicit locking at a lower level with exclusive or shared locks.

Shared & Intention-Exclusive (SIX): In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

Compatibility Matrix with Intention Lock Modes: The below table describes the compatibility matrix for these lock modes:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Observe that in multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

- If transaction T1 reads record Ra9 in file Fa, then transaction T1 needs to lock the database, area A1 and file Fa in IX mode. Finally, it needs to lock Ra2 in S mode.
- If transaction T2 modifies record Ra9 in file Fa, then it can do so after locking the database, area A1 and file Fa in IX mode. Finally, it needs to lock the Ra9 in X mode.
- If transaction T3 reads all the records in file Fa, then transaction T3 needs to lock the database, and area A in IS mode. At last, it needs to lock Fa in S mode.
- If transaction T4 reads the entire database, then T4 needs to lock the database in S mode.

Database Recovery Techniques in DBMS

Last Updated : 30 Jul, 2025



Database Systems like any other computer system, are subject to failures but the data stored in them must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

Types of Recovery Techniques in DBMS

Database recovery techniques are used in database management systems (DBMS) to restore a database to a consistent state after a failure or error has occurred. The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss.

There are mainly two types of recovery techniques used in DBMS

- Rollback/Undo Recovery Technique**
- Commit/Redo Recovery Technique**
- CheckPoint Recovery Technique**

1. Rollback/Undo Recovery Technique

Rollback/Undo Recovery reverses the changes made by transactions that failed before completion. Using the transaction log, the system undoes these changes to restore the database to its previous consistent state.

2. Commit/Redo Recovery Technique

Commit/Redo Recovery re-applies the changes of successfully committed transactions after a system failure. It uses the transaction log to redo operations and restore the database to its most recent consistent state.

3. Checkpoint Recovery Technique

Checkpoint Recovery is a method used in databases to save the system's state at regular intervals, called checkpoints. This saved state allows the system to recover quickly after a crash by restoring from the last checkpoint, reducing data loss and downtime. It ensures data consistency while balancing system performance and recovery time.

Database Systems

Database recovery helps restore data after failures like system crashes, wrong commands, virus attacks, etc. Recovery relies on **backups** and **system logs** (a special file that tracks all transaction activities).

System Log Entries

- **start_transaction(T):** Transaction T begins.
- **read_item(T, X):** T reads item X.
- **write_item(T, X, old, new):** T changes X from old to new.
- **commit(T):** T finishes successfully; changes are saved.
- **abort(T):** T fails and stops.
- **checkpoint:** Marks a safe state where all previous transactions are committed and logs are saved.

Recovery Actions

- **Undo:** Roll back changes of failed transactions using log info (set values back to old_value).
- **Redo:** Reapply operations of committed transactions if not written to disk.

Recovery Techniques

1. Deferred Update (No-Undo/Redo)

- Changes are made only after commit.
- If a transaction fails before commit, nothing needs to be undone.
- After a crash, redo may be needed.

2. Immediate Update (Undo/Redo)

- Changes can happen before commit but must be logged first.
- On failure, undo uncommitted changes; redo committed ones.

Other Recovery Mechanisms

- **Buffering/Caching:** Data is updated in memory (cache) before writing to disk. Modified buffers are marked using a *dirty bit*.
- **Shadow Paging:** Maintains a copy (shadow) of the database pages. Updates are made to new pages, and changes become official only after commit.
- **Backward Recovery (Undo/Rollback):** Reverses changes made by failed or incomplete transactions.
- **Forward Recovery (Redo/Roll Forward):** Reapplies saved valid changes to restore the database after a crash.

Backup Techniques

There are different types of Backup Techniques. Some of them are listed below.

- **Full Database Backup:** A complete backup of the entire database, including all data, system metadata, and full text catalogs. It allows full restoration of the database at the time the backup was taken.
- **Differential Backup:** Captures only the data that has changed since the last full backup. It requires the last full backup to restore the database, followed by the latest differential backup.

- **Transaction Log Backup:** Backs up all changes recorded in the transaction log since the last log backup. It includes every transaction and allows recovery to a specific point in time even if the data files are lost—without losing any committed transactions.

Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

Last Updated : 02 Aug, 2025

ARIES is a robust recovery algorithm based on the Write-Ahead Logging (WAL) protocol, widely used in modern DBMS to ensure durability and consistency after system failures.

Types of Log Records in ARIES

Every update operation creates a **log record**, which can be of the following types:

- **Undo-only Log Record:**
Stores only the before image (original data). Used to undo changes.
- **Redo-only Log Record:**
Stores only the after image (new data). Used to redo changes.
- **Undo-Redo Log Record:**
Stores both before and after images. Used for both undo and redo operations.

Log Sequence Numbers (LSN)

- Every log record is assigned a unique, monotonically increasing LSN.
- Every data page has a pageLSN indicating the LSN of the last update applied to it.

Write-Ahead Logging (WAL) Rules

- A log record must reach stable storage before the corresponding data page is written to disk.
- Log writes are buffered in a memory area called the log tail.
- The log tail is flushed to disk when full.
- A transaction is not considered committed until its commit record is flushed to disk.

Checkpoints

Periodically, the system writes a checkpoint record to the log.

A checkpoint contains:

- Transaction Table
- Dirty Page Table

A master log record, stored in stable storage, holds the LSN of the most recent checkpoint.

Recovery Phases in ARIES

When the system restarts after a crash, ARIES performs three recovery phases:

Analysis Phase

- Starts from the latest checkpoint.
- Reconstructs the state of active transactions and dirty pages at the time of the crash.

Redo Phase

- Replays operations forward from the smallest LSN of dirty pages.
- Reapplies necessary changes to bring the database to the state before crash.

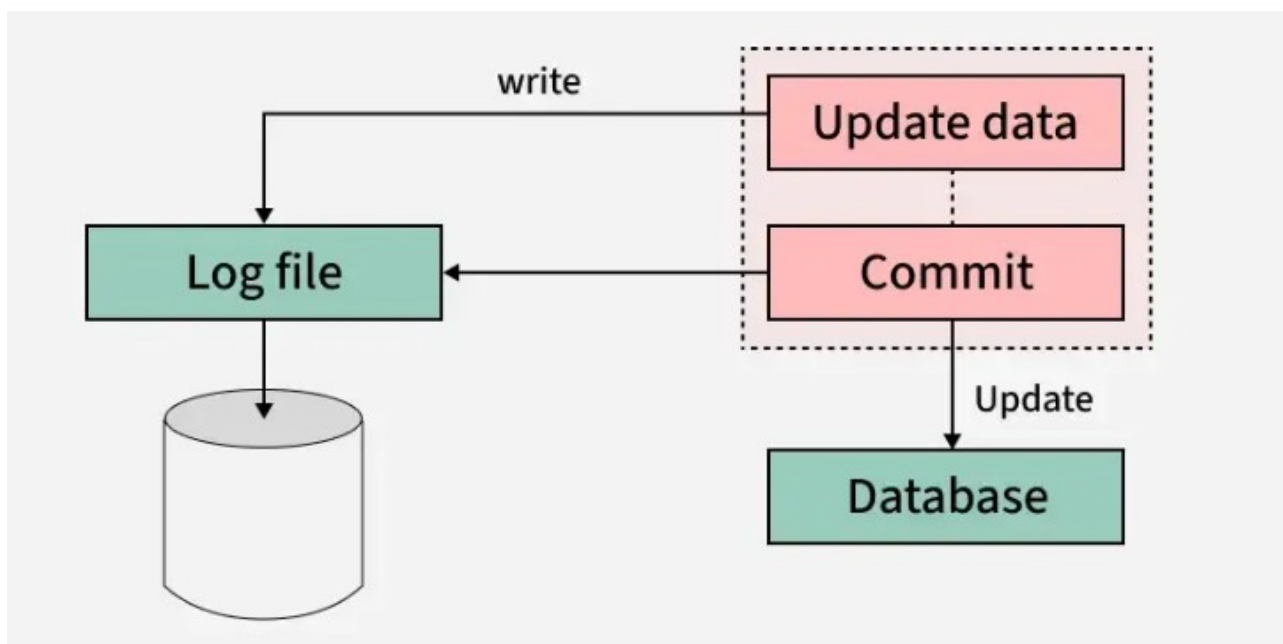
Undo Phase

- Scans the log backward.
- Rolls back the effects of uncommitted (loser) transactions.

Log based Recovery in DBMS

Last Updated : 30 Jul, 2025

Log-based recovery in DBMS ensures data can be maintained or restored in the event of a system failure. The DBMS records every transaction on stable storage, allowing for easy data recovery when a failure occurs. For each operation performed on the database, a log file is created. Transactions are logged and verified before being applied to the database, ensuring data integrity.



Log in DBMS

A log is a sequence of records that document the operations performed during database transactions. Logs are stored in a log file for each transaction, providing a mechanism to recover data in the event of a failure. For every operation executed on the database, a corresponding log record is created. It is critical to store these logs

before the actual transaction operations are applied to the database, ensuring data integrity and consistency during recovery processes.

For example, consider a transaction to modify a student's city. This transaction generates the following logs:

1. Start Log: When the transaction begins, a log is created to indicate the start of the transaction.

- **Format:** <Tn, Start>
- Here, Tn represents the transaction identifier.
- **Example:** <T1, Start> indicates that Transaction 1 has started.

2. Operation Log: When the city is updated, a log is recorded to capture the old and new values of the operation.

- **Format:** <Tn, Attribute, Old_Value, New_Value>
- **Example:** <T1, City, 'Gorakhpur', 'Noida'> shows that in Transaction 1, the value of the City attribute has changed from 'Gorakhpur' to 'Noida'.

3. Commit Log: Once the transaction is successfully completed, a final log is created to indicate that the transaction has been completed and the changes are now permanent.

- **Format:** <Tn, Commit>
- **Example:** <T1, Commit> signifies that Transaction 1 has been successfully completed.

These logs play a crucial role in ensuring that the database can recover to a consistent state after a system crash. If a failure occurs, the DBMS can use these logs to either roll back incomplete transactions or redo committed transactions to maintain data consistency.

Key Operations in Log-Based Recovery

Undo Operation

The undo operation reverses the changes made by an uncommitted transaction, restoring the database to its previous state.

Example of Undo: Consider a transaction T1 that updates a bank account balance but fails before committing:

Initial State: Account balance = 500.

Transaction T1:

- Update balance to 600.
- Log entry:

<T1, Balance, 500, 600>

Failure: T1 fails before committing.

Undo Process:

- Use the old value from the log to revert the change.
- Set balance back to 500.
- Final log entry after undo:

<T1, Abort>

Redo Operation

The redo operation re-applies the changes made by a committed transaction to ensure consistency in the database.

Example of Redo: Consider a transaction T2 that updates an account balance but the database crashes before changes are permanently reflected:

Initial State: Account balance = 300.

Transaction T2:

- Update balance to 400.
- Log entries:

<T2, Start> <T2, Balance, 300, 400> <T2, Commit>

Crash: Changes are not reflected in the database.

Redo Process:

- Use the new value from the log to reapply the committed change.
- Set balance to 400.

Undo-Redo Example:

Assume two transactions:

- T1: Failed transaction (requires undo).
- T2: Committed transaction (requires redo).

Log File:

<T1, Start><T1, Balance, 500, 600><T2, Start><T2, Balance, 300, 400><T2, Commit><T1, Abort>

Identify Committed and Uncommitted Transactions:

- T1: Not committed → Undo.
- T2: Committed → Redo.

Undo T1: Revert balance from 600 to 500.

Redo T2: Reapply balance change from 300 to 400.

Operation	Trigger	Action
Undo	For uncommitted/failed transactions	Revert changes using the old values in the log.
Redo	For committed transactions	Reapply changes using the new values in the log.

These operations ensure data consistency and integrity in the event of system failures.

3. Checkpoint Recovery Technique

Checkpoint Recovery is a method used in databases to save the system's state at regular intervals, called checkpoints. This saved state allows the system to recover quickly after a crash by restoring from the last checkpoint, reducing data loss and downtime. It ensures data consistency while balancing system performance and recovery time.

The **architecture of a Distributed Database Management System (DDBMS)** refers to how the database is logically and physically structured across multiple locations (nodes) and how various components interact to manage and query distributed data.

Overview

A **Distributed DBMS** is a software system that manages a **logically integrated database** over a **network of physically distributed sites**.

Key Components of Distributed DBMS Architecture

1. Global Conceptual Schema (GCS)

- Represents the unified logical view of the entire distributed database.
- Users write queries using this global schema.

2. Local Conceptual Schemas (LCS)

- Represent the database schema at individual sites.
- Each site manages its own schema.

3. Fragmentation and Allocation Module

- Handles **data partitioning**:
 - **Horizontal** (row-wise split)
 - **Vertical** (column-wise split)
 - **Mixed (hybrid)**
- Controls where each fragment is stored (allocation).

4. Distributed Query Processor (DQP)

- Translates global queries into local queries.
- Coordinates data retrieval across multiple sites.

5. Distributed Transaction Manager (DTM)

- Manages atomicity, consistency, isolation, and durability (ACID) for transactions spanning multiple nodes.

6. Distributed Concurrency Control Manager

- Maintains data consistency by managing concurrent access to data across all nodes.

7. Distributed Recovery Manager

- Handles failures and ensures data recovery across distributed nodes.

8. Communication Manager

- Manages communication between different database sites using a network protocol.
-

Types of DDBMS Architectures

1. Client-Server Architecture

- **Client:** Sends queries and receives results.
- **Server:** Processes requests, manages data storage and execution.

Typical in modern web-based applications.

2. Peer-to-Peer (P2P) Architecture

- All nodes are equal – act as both clients and servers.
- Ideal for fault-tolerant and scalable systems.

Example: Blockchain networks, distributed file systems.

3. Multi-Database (Federated) Architecture

- Consists of **independent DBMSs** at each site.
- No global schema necessarily.
- Coordination is done at a higher federated level.

Useful when integrating different database systems.

Data Distribution Approaches

1. Replication

- Data is **copied** and maintained at multiple sites for availability.

2. Partitioning (Fragmentation)

- Data is **divided** among sites to improve performance and reduce redundancy.

3. Hybrid

- Combination of replication and partitioning.

Distribution of Data

► What It Means:

Data in a distributed database is **stored across multiple physical locations** (nodes, sites, or servers), often connected via a network. The goal is to **distribute data efficiently** while maintaining system performance, availability, and scalability.

Key Concepts:

• Fragmentation

Dividing a database into smaller parts called *fragments*:

- **Horizontal Fragmentation** – splitting rows based on conditions
- **Vertical Fragmentation** – splitting columns (attributes)
- **Mixed/Hybrid Fragmentation** – combining both

• Replication

Creating **copies of data** at multiple sites to ensure **fault tolerance**, **availability**, and **faster access**.

• Allocation

Deciding **where** (on which site) each fragment or replica should be stored.

Objective:

- Improve **performance** by localizing access
 - Enhance **reliability** and **availability**
 - Support **scalability**
-

Logical Co-relation (Global Logical Schema)

► What It Means:

Even though data is **physically distributed**, it is managed as a **single logical database**. Users and applications interact with a **unified view** without needing to know where or how the data is stored.

Key Concepts:

• Global Conceptual Schema

A logical schema that provides a **uniform view** of all data, regardless of physical location.

- **Data Independence**

Users are **shielded** from changes in:

- Data location (Location Independence)
- Fragmentation or replication details
- Underlying physical storage

- **Transparency Features:**

- **Location Transparency**
 - **Fragmentation Transparency**
 - **Replication Transparency**
-

Objective:

- Provide a **seamless user experience**
- Simplify application development
- Maintain **data consistency and integrity**

DDB vs Centralized DB – Feature Comparison

Feature	Distributed DBMS (DDBMS)	Centralized DBMS (CDBMS)
Centralized Control	Not centralized — control is distributed across sites.	Fully centralized control at a single site.
Data Independence	High – provides logical and physical data independence .	Also supports data independence, but less flexible than DDBMS.
Distributed Transparency	Supports various transparencies (location, fragmentation, replication).	Not required — all data is in one place.
Reduction of Redundancy	Some redundancy due to replication , but managed to improve reliability/performance.	Minimal redundancy (unless manually introduced), but higher risk of data loss .
Complex Physical Structure	Complex – multiple nodes, data allocation, network communication required.	Simple – single-site storage and access.
Integrity, Recovery, and Concurrency	Complex – needs distributed protocols for integrity, recovery, and concurrency control.	Simpler – easier to maintain ACID properties.
Privacy and Security	More challenging – must secure multiple sites and communication channels .	Easier to implement — centralized access control and single-point security.

Why Use a Distributed Database (DDB)?

A **Distributed Database** is used when a single centralized database is not sufficient to meet the performance, availability, and scalability needs of modern applications, especially in geographically spread systems.

Key Reasons for Using DDBMS

Improved Reliability & Availability

- Data is replicated or distributed across multiple sites.
 - If one site fails, others can continue operation.
 - Ensures **fault tolerance** and **high availability**.
-

Improved Performance

- Data is stored closer to where it is needed (local access).
 - Reduces data transfer time and response latency.
 - Supports **faster query execution** for distributed users.
-

Scalability

- Easy to **add new sites or nodes** as data grows.
 - No need to overhaul the entire system — supports **horizontal scaling**.
-

Modularity (Flexibility of Growth)

- Individual sites can be upgraded or maintained **independently**.
 - System can grow by adding more databases **without disrupting operations**.
-

Distributed Data Processing

- Enables **parallel query execution**.
 - Workload is shared across multiple nodes, reducing bottlenecks.
-

Local Autonomy

- Each site can **control its own data**, useful in organizations with distributed departments or branches.

- Increases **organizational flexibility**.
-

Cost Efficiency

- Can use **low-cost commodity hardware** at multiple sites instead of a high-end centralized server.
 - Reduces bandwidth costs by minimizing data movement.
-

Data Sharing Across Locations

- Useful for **multi-location companies** (e.g., banks, retail chains).
 - Supports **real-time data access** across all branches.
-

Support for Distributed Applications

- Cloud computing, IoT, mobile applications, and web services often demand **distributed data storage**.
 - DDB is the backbone of such systems.
-

Improved Reliability of Backups and Recovery

- Backup and recovery can be **distributed and parallelized**.
 - Reduces **downtime during recovery**.
-

Reference architecture of a distributed DBMS

In chapter 1 we looked at the ANSI_SPARC three-level architecture of a DBMS. The architecture reference shows how different schemas of the DBMS can be organised. This architecture cannot be applied directly to distributed environments because of the diversity and complexity of distributed DBMSs. The diagram below shows how the schemas of a distributed database system can be organised. The diagram is adopted from Hirendra Sisodiya (2011).

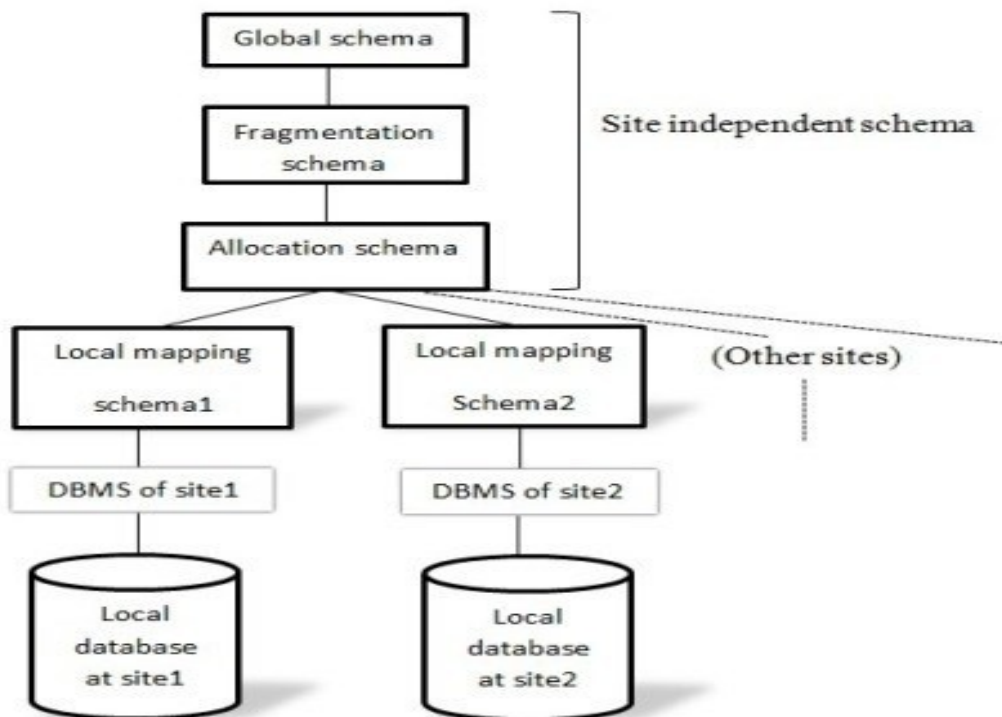


Figure 15.4

Reference architecture for distributed database

1. Global schema

The global schema contains two parts, a global external schema and a global conceptual schema. The global schema gives access to the entire system. It provides applications with access to the entire distributed database system, and logical description of the whole database as if it was not distributed.

2. Fragmentation schema

The fragmentation schema gives the description of how the data is partitioned.

3. Allocation schema

Gives a description of where the partitions are located.

4. Local mapping

The local mapping contains the local conceptual and local internal schema. The local conceptual schema provides the description of the local data. The local internal schema gives the description of how the data is physically stored on the disk.

Fragmentation in Distributed DBMS

Last Updated : 31 Jul, 2025

Fragmentation is the process of dividing a database table into smaller parts called fragments, which are stored on different sites in a

distributed system. Each fragment is a logical unit of data, and together, they must allow the **reconstruction** of the original table without any data loss- typically using **UNION** or **JOIN** operations. Fragments are independent (no fragment can be derived from another), and users don't need to know that the data is fragmented. This is known as **fragmentation transparency**.

We have three methods for data fragmenting of a table:

- Horizontal fragmentation
- Vertical fragmentation
- Mixed or Hybrid fragmentation

Let's discuss them one by one:

Horizontal fragmentation

Horizontal fragmentation splits a table by rows based on conditions on one or more attributes. Each **fragment contains a subset of rows**, which are then stored at different sites. This helps in localizing data access.

In relational algebra, it's represented using the **SELECT (σ)** operation on the table.

$$\sigma_p(T)$$

- σ is relational algebra operator for selection
- p is the condition satisfied by a horizontal fragment

Note that a union operation can be performed on the fragments to construct table T. Such a fragment containing all the rows of table T is called a *complete horizontal fragment*.

For example, consider an EMPLOYEE table (T) :

En o	Ena me	Desi gn	Sala ry	D ep
10 1	A	abc	300 0	1
10 2	B	abc	400 0	1
10 3	C	abc	550 0	2
10 4	D	abc	500 0	2
10 5	E	abc	200 0	2

This EMPLOYEE table can be divided into different fragments like:

- EMP 1 = $\sigma_{Dep = 1}$ EMPLOYEE
- EMP 2 = $\sigma_{Dep = 2}$ EMPLOYEE

These two fragments are:

1. T1 fragment of Dep = 1

E n o	Ena me	Desi gn	Sala ry	D ep
1 0 1	A	abc	300 0	1
1 0 2	B	abc	400 0	1

2. T2 fragment of Dep = 2

E n o	Ena me	Desi gn	Sala ry	D ep
1	C	abc	550	2

0 3			0	
1 0 4	D	abc	500 0	2
1 0 5	E	abc	200 0	2

Now, here it is possible to get back T as **$T = T1 \cup T2 \cup \dots \cup TN$**

Vertical Fragmentation

Vertical fragmentation splits a table by columns (attributes), storing different columns on different sites. Since each site may not need all columns, this improves efficiency. To reconstruct **the** original table, each fragment must include a **common key** (like the primary key or a tuple ID) so that rows can be joined correctly using a **natural JOIN**.

$$\pi_{a1, a2, \dots, an} (T)$$

- **π** is relational algebra operator
- **$a1, \dots, an$** are the attributes of T
- **T** is the table (relation)

For example, for the EMPLOYEE table we have T1 as :

E n o	Ena me	Desi gn	Tuple_ id
1 0 1	A	abc	1
1 0 2	B	abc	2
1 0 3	C	abc	3

1 0 4	D	abc	4
1 0 5	E	abc	5

For the second sub table of relation after vertical fragmentation is given as follows :

Sala ry	D e p	Tuple_ id
3000	1	1
4000	2	2
5500	3	3
5000	1	4
2000	4	5

This is T2 and to get back to the original T, we join these two fragments T1 and T2 as **$\pi_{\text{EMPLOYEE}}(\mathbf{T1} \bowtie \mathbf{T2})$**

Mixed Fragmentation

Hybrid (or mixed) fragmentation combines vertical and horizontal fragmentation-first splitting a table by columns, then dividing those fragments by rows (or vice versa). It uses both **SELECT (σ)** and **PROJECT (π)** operations. This is useful when simple horizontal or vertical fragmentation isn't sufficient for data distribution.

Mixed fragmentation can be done in two different ways:

1. The first method is to first create a set or group of horizontal fragments and then create vertical fragments from one or more of the horizontal fragments.
2. The second method is to first create a set or group of vertical fragments and then create horizontal fragments from one or more of the vertical fragments.

The original relation can be obtained by the combination of JOIN and UNION operations which is given as follows:

$$\sigma_P(\pi_{a1, a2, \dots, an}(\sigma_p(T)))$$

For example, for our **EMPLOYEE** table, below is the implementation of mixed fragmentation is **$\pi_{\text{Ename, Design}}(\sigma_{\text{Eno} < 104}(\text{EMPLOYEE}))$**

The result of this fragmentation is:

Ena me	Desi gn
A	abc
B	abc
C	abc

Advantages:

- As the data is stored close to the usage site, the efficiency of the database system will increase
- Local query optimization methods are sufficient for some queries as the data is available locally
- In order to maintain the security and privacy of the database system, fragmentation is advantageous

Disadvantages:

- Access speeds may be very high if data from different fragments are needed

- If we are using recursive fragmentation, then it will be very expensive

Transparencies in DDBMS

Last Updated : 23 Jul, 2025

Distributed Database Management System (DDBMS) :

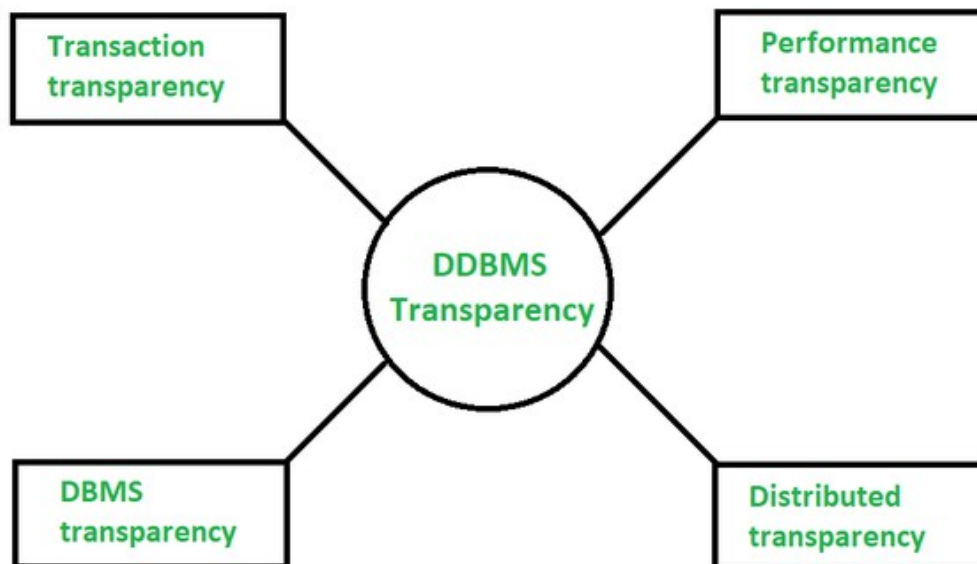
A distributed database is essentially a database that's not limited to at least one system, it covers different sites, i.e, on multiple computers or over a network of computers. A distributed database system is found on various sites that don't share physical components. This may be required when a specific database must be accessed by various users globally. It must be managed such for the users. It's like one single database.

What is transparency?

Transparency in DDBMS refers to the transparent distribution of information to the user from the system. It helps in hiding the information that is to be implemented by the user. Let's say, for example, in a normal DBMS, data independence is a form of transparency that helps in hiding changes in the definition & organization of the data from the user. But, they all have the same overall target. That means to make use of the distributed database the same as a centralized database.

In Distributed Database Management System, there are four types of transparencies, which are as follows -

- Transaction transparency
- Performance transparency
- DBMS transparency
- Distribution transparency



1. Transaction transparency-

This transparency makes sure that all the transactions that are distributed preserve distributed database integrity and regularity. Also, it is to understand that distribution transaction access is the data stored at multiple locations. Another thing to notice is that the DDBMS is responsible for maintaining the atomicity of every sub-transaction (By this, we mean that either the whole transaction takes place directly or doesn't happen in the least). It is very complex due to the use of fragmentation, allocation, and replication structure of DBMS.

2. Performance transparency-

This transparency requires a DDBMS to work in a way that if it is a centralized database management system. Also, the system should not undergo any downs in performance as its architecture is distributed. Likewise, a DDBMS must have a distributed query processor which can map a data request into an ordered sequence of operations on the local database. This has another complexity to

take under consideration which is the fragmentation, replication, and allocation structure of DBMS.

3. DBMS transparency-

This transparency is only applicable to heterogeneous types of DDBMS (Databases that have different sites and use different operating systems, products, and data models) as it hides the fact that the local DBMS may be different. This transparency is one of the most complicated transparencies to make use of as a generalization.

4. Distribution transparency-

Distribution transparency helps the user to recognize the database as a single thing or a logical entity, and if a DDBMS displays distribution data transparency, then the user does not need to know that the data is fragmented.

Distribution transparency has its 5 types, which are discussed below

-

- Fragmentation transparency-

In this type of transparency, the user doesn't have to know about fragmented data and, due to which, it leads to the reason why database accesses are based on the global schema. This is almost somewhat like users of SQL views, where the user might not know that they're employing a view of a table rather than the table itself.

- Location transparency-

If this type of transparency is provided by DDBMS, then it is necessary for the user to know how the data has been fragmented, but knowing the location of the data is not necessary.

- Replication transparency-

In replication transparency, the user does not know about the copying of fragments. Replication transparency is related to concurrency transparency and failure transparency. Whenever a user modifies a data item, the update is reflected altogether in the copies of the table. However, this operation shouldn't be known to the user.

- Local Mapping transparency-

In local mapping transparency, the user needs to define both the fragment names, location of data items while taking into account any duplications that may exist. This is a more difficult and time-taking query for the user in DDBMS transparency.

- Naming transparency-

We already know that DBMS and DDBMS are types of centralized database system. It means that each item in this database must consist of a unique name. It further means that DDBMS must make sure that no two sites are creating a database object with the same name. So to solve the problem of naming transparency, there are two ways, either we can create a central name server to create the unique names of objects in the system, or, differently, is to add an object starting with the identifier of the creator site

What is Database Sharding?

Database Sharding is a database scaling technique where data is partitioned across multiple servers or databases, known as shards. Each shard holds a subset of the entire dataset. This method distributes the load by horizontally splitting the data, meaning each shard manages a different piece of the data.

•Advantages of Database Sharding

- Shards handle smaller amounts of data, reducing query response times.
- Easily scale the system by adding more shards as data grows.

- Shards distribute traffic across multiple servers, avoiding bottlenecks.

•Disadvantages of Database Sharding

- Setting up and managing shards adds complexity to database design and maintenance.
- As data grows unevenly, shards may need to be rebalanced, which can be tricky.
- Running queries across multiple shards can be slow and complicated to implement.

•Features of Sharding

- Horizontal data partitioning.
- Different shards can be hosted on different servers.
- Allows for independent scaling of each shard.
- Shards may be spread across different geographical locations.

What is Database Replication?

Database replication involves copying and maintaining database information in multiple locations. This allows for multiple copies (replicas) of the same data on different servers, ensuring **availability** and **redundancy**. There are typically two types: Master-Slave and Master-Master replication.

•Advantages of Database Replication

- In case one replica fails, others are available, ensuring data availability.
- Read requests can be distributed across multiple replicas to reduce load on a single server.
- Data is replicated across servers, making it less likely to lose data.

•Disadvantages of Database Replication

- Writes might not instantly reflect on all replicas, leading to inconsistency issues.

- Each replica stores the full data, leading to higher storage needs.
- Replicating data across geographical distances may lead to latency in synchronization.

•Features of Replication

- Full data copies on multiple servers.
- Can be synchronous (immediate updates across replicas) or asynchronous (eventual updates).
- Improves data availability and disaster recovery.
- Can be used for both read scaling and failover.

Database Sharding vs. Database Replication

Below the difference between database sharding and replication:

Database Sharding	Database Replication
Divides data into smaller chunks (shards).	Copies the same data to multiple servers.
It is used for Scalability and performance improvement.	It is used for High availability and redundancy.
Each shard contains a portion of the data.	Each replica contains a full copy of the data.
Spreads data and queries across shards.	Spreads read queries across replicas.
Cross-shard queries can complicate consistency.	Can suffer from inconsistency due to lag between replicas.
Low tolerance as failure of one shard affects part of data.	High tolerance, other replicas can take over if one fails.
Complex to implement and manage.	Simpler to implement but requires careful sync management.

What is the CAP Theorem?

The CAP theorem is a fundamental concept in distributed systems theory that was first proposed by Eric Brewer in 2000 and subsequently shown by Seth Gilbert and Nancy Lynch in 2002. It

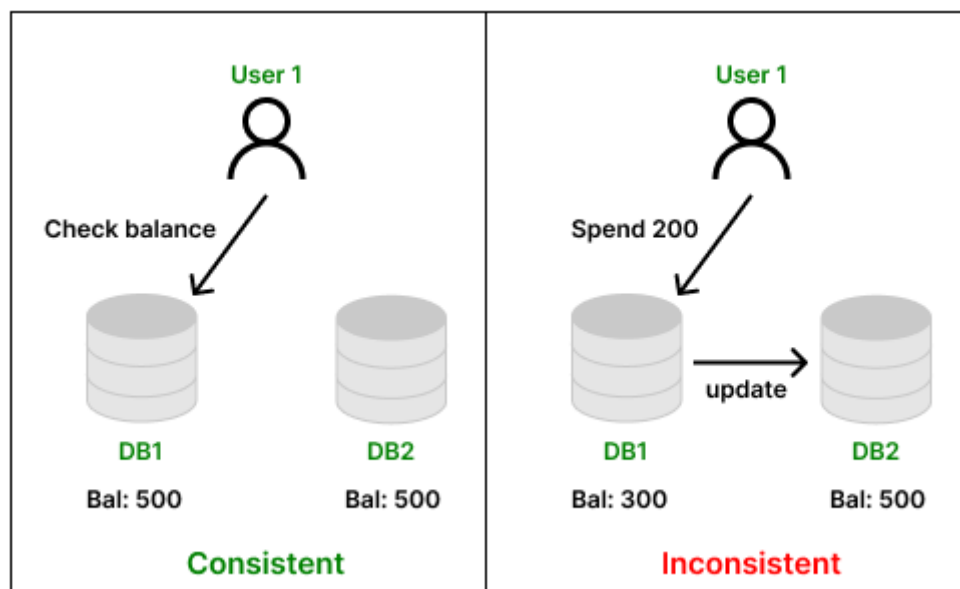
asserts that all three of the following qualities cannot be concurrently guaranteed in any distributed data system:

1. Consistency

Consistency means that all the nodes (databases) inside a network will have the same copies of a replicated data item visible for various transactions. It guarantees that every node in a distributed cluster returns the same, most recent, and successful write. It refers to every client having the same view of the data. There are various types of consistency models. Consistency in CAP refers to sequential consistency, a very strong form of consistency.

Note that the concept of Consistency in ACID and CAP are slightly different since in CAP, it refers to the consistency of the values in different copies of the same data item in a replicated distributed system. In **ACID**, it refers to the fact that a transaction will not violate the integrity constraints specified on the database schema.

For example, a user checks his account balance and knows that he has 500 rupees. He spends 200 rupees on some products. Hence the amount of 200 must be deducted changing his account balance to 300 rupees. This change must be committed and communicated with all other databases that hold this user's details. Otherwise, there will be inconsistency, and the other database might show his account balance as 500 rupees which is not true.

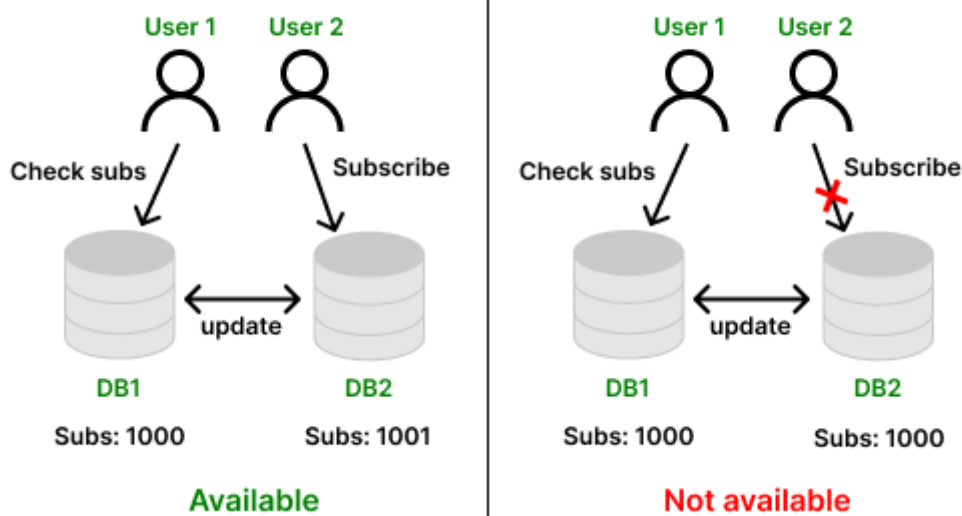


2. Availability

Availability means that each read or write request for a data item will either be processed successfully or will receive a message that the

operation cannot be completed. Every non-failing node returns a response for all the read and write requests in a reasonable amount of time. The key word here is "every". In simple terms, every node (on either side of a network partition) must be able to respond in a reasonable amount of time.

For example, user A is a content creator having 1000 other users subscribed to his channel. Another user B who is far away from user A tries to subscribe to user A's channel. Since the distance between both users are huge, they are connected to different database node of the social media network. If the distributed system follows the principle of availability, user B must be able to subscribe to user A's channel.

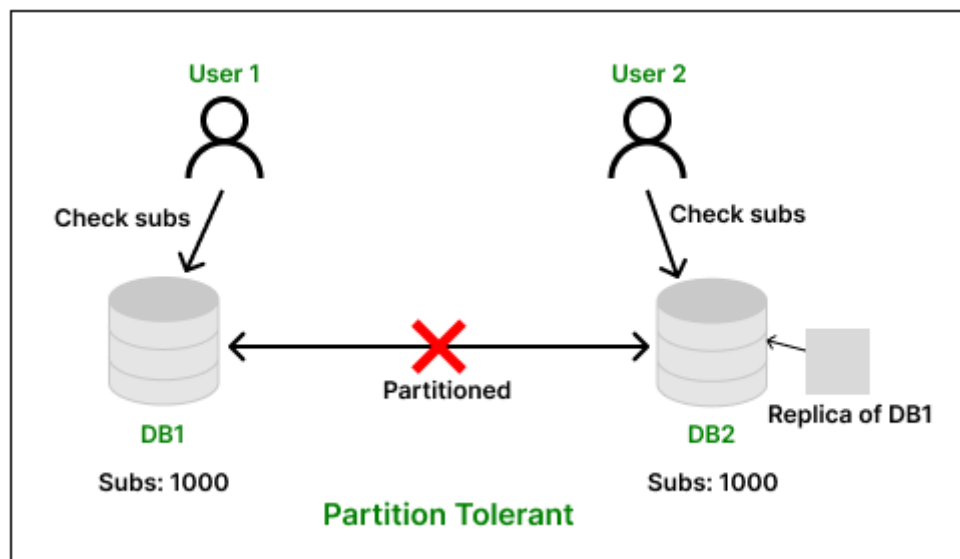


3. Partition Tolerance

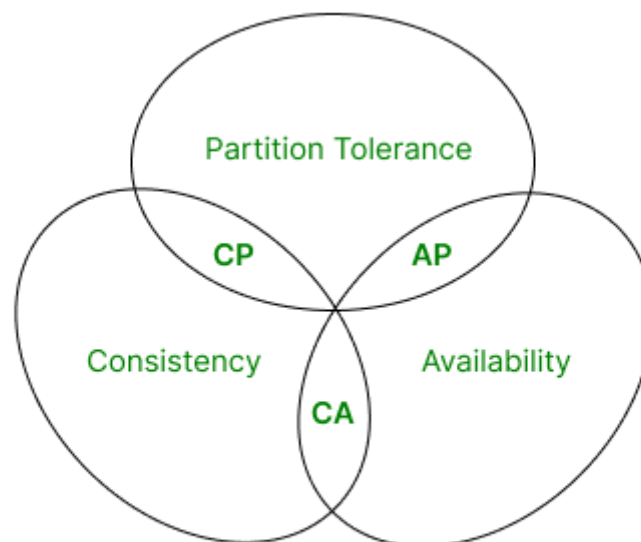
Partition tolerance means that the system can continue operating even if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other. That means, the system continues to function and upholds its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

For example, take the example of the same social media network where two users are trying to find the subscriber count of a particular channel. Due to some technical fault, there occurs a network outage, the second database connected by user B loses its connection with first database. Hence the subscriber count is shown to the user B with the help of replica of data which was previously stored in database 1

backed up prior to network outage. Hence the distributed system is partition tolerant.



The CAP theorem states that distributed databases can have at most two of the three properties: consistency, availability, and partition tolerance. As a result, database systems prioritize only two properties at a time.



Venn diagram of CAP theorem

Weaknesses of RDBMS

Poor Performance with Large-Scale or Complex Data

- RDBMSs can struggle with:
 - **Big Data** (terabytes to petabytes)
 - **High-velocity** read/write operations
 - **Unstructured** or **semi-structured** data (e.g., images, videos, JSON, XML)
-

Rigid Schema

- Schema must be **defined upfront**.
 - Any schema changes require **migrations** that can be complex and error-prone.
 - Not ideal for **agile or dynamic** applications (e.g., startups, evolving data models).
-

Limited Horizontal Scalability

- Traditional RDBMSs scale **vertically** (by upgrading hardware).
 - **Horizontal scaling** (adding more servers) is difficult and often not supported out-of-the-box.
-

Not Ideal for Distributed Systems

- Centralized control limits:
 - **Fault tolerance**
 - **Geo-replication**
 - **Latency optimization**
 - Distributed transaction support (e.g., 2PC) adds **overhead and complexity**.
-

Poor Support for Semi-Structured / NoSQL Data

- JSON, key-value pairs, documents, and graph data are not handled efficiently.
 - Workarounds exist (e.g., PostgreSQL JSONB), but are not native strengths.
-

High Overhead for Joins and Complex Queries

- Join operations on large datasets can be **expensive** and **slow**.
 - Indexing can help but increases **storage and maintenance cost**.
-

Limited Flexibility for Hierarchical or Graph Data

- RDBMSs are **tabular** and don't represent **hierarchies** or **networks** naturally.
 - Graph databases (e.g., Neo4j) or NoSQL are better for such structures.
-

Cost of Licensing and Maintenance

- Some RDBMS platforms (e.g., Oracle, MS SQL Server) have **high licensing costs**.
 - Even open-source RDBMSs (e.g., PostgreSQL, MySQL) require skilled **administration and tuning**.
-

Concurrency and Locking Issues

- Heavy transaction systems may face:
 - **Deadlocks**
 - **Lock contention**
 - **Throughput bottlenecks**
-

Not Cloud-Native by Default

- Traditional RDBMSs were designed for **on-premises environments**.
- Scaling and performance tuning in cloud environments requires extra effort or workarounds.

Object-Oriented Data Model (OODM)

Key Contributions:

a. Encapsulation

- Combines **data** + **methods** (behavior) into a single unit (object).
- Encourages **modularity** and **reusability**.

b. Complex Data Types Support

- Supports **user-defined types (UDTs)** like arrays, sets, lists, tuples, etc.

- Useful for **CAD/CAM, multimedia, GIS, scientific databases**, etc.

c. Inheritance

- Allows one object/class to inherit properties of another.
- Promotes **code reuse, hierarchical modeling**, and **polymorphism**.

d. Object Identity

- Each object has a **unique object identifier (OID)** independent of its values.
- Enhances data linking without relying on keys like in RDBMS.

e. Direct Representation of Real-world Entities

- Models **real-world objects more naturally** than rows and columns.
 - Ideal for applications needing **semantic richness**.
-

Object-Relational Data Model (ORDBMS)

Key Contributions:

a. Hybrid Power: Relational + Object Features

- Combines strengths of RDBMS (e.g., SQL, ACID) with object-oriented features.
- Allows **backward compatibility** with relational databases.

b. User-Defined Types (UDTs)

- Supports custom data types with attributes and methods.
- Enables modeling of **non-atomic** and **complex attributes**.

c. Inheritance in Tables and Types

- Tables can inherit structure and behavior from other tables or types.
- Promotes **data abstraction and reuse**.

d. Table Functions & Object Methods

- Supports **methods/functions** within objects stored in tables.
- Encourages **encapsulation** and **behavioral modeling**.

e. Support for Multimedia and Spatial Data

- Used in applications needing **images, audio, video, GIS data**, etc.
- Extends SQL to handle such complex data.

XML Document (eXtensible Markup Language)

What is it?

- An **XML Document** is a **structured text file** that stores data using custom tags.
- XML is designed to **store and transport data**, not display it.
- It is **both human-readable and machine-readable**.

Example:

```
xml
CopyEdit
<book>
  <title>Learning XML</title>
  <author>Soumodeep Karmakar</author>
  <price>450</price>
</book>
```

DTD (Document Type Definition)

What is DTD?

- DTD defines the **structure** and the **legal elements and attributes** of an XML document.
- It ensures the XML document is **valid** (follows the defined structure).

Types:

- **Internal DTD** – Defined within the XML document
- **External DTD** – Defined in a separate file

Example (Internal DTD):

```
xml
CopyEdit
<?xml version="1.0"?>
<!DOCTYPE book [
  <!ELEMENT book (title, author, price)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
]>
<book>
  <title>Learning XML</title>
  <author>Soumodeep Karmakar</author>
  <price>450</price>
</book>
```

XML Schema (XSD – XML Schema Definition)

What is XML Schema?

- XML Schema defines the **structure**, **data types**, and **constraints** of XML documents.
- More **powerful** and **flexible** than DTD.
- Written in **XML syntax** (unlike DTD).
- Supports:
 - Data types (integer, date, string, etc.)
 - Constraints (minLength, maxOccurs, pattern, etc.)
 - Namespaces

Example:

```
xml
CopyEdit
<!-- XML Document -->
<book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="book.xsd">
  <title>Learning XML</title>
  <author>Soumodeep Karmakar</author>
  <price>450</price>
</book>
```

```
xml
CopyEdit
<!-- XML Schema (book.xsd) -->
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="author" type="xs:string"/>
        <xs:element name="price" type="xs:decimal"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

DTD vs XML Schema

Feature	DTD	XML Schema (XSD)
Syntax	Non-XML	XML-based
Data Types	Limited (only #PCDATA, etc.)	Rich data types (int, date, float...)
Namespace Support	No	Yes
Extensibility	No	Yes
Readability	Simpler	Slightly complex but more powerful
Industry Preference	Obsolete in many cases	Preferred for modern XML validation

XPath (XML Path Language)

What is XPath?

- XPath is a **query language** used to **navigate** and **select nodes** from an XML document.
- It uses **path expressions** to identify parts of an XML tree (like navigating folders).
- XPath is **used in XSLT, XQuery, and XML Schema**.

Example XML:

```
xml
CopyEdit
<library>
  <book>
    <title>XML Fundamentals</title>
    <author>Soumodeep</author>
  </book>
  <book>
    <title>Advanced XML</title>
    <author>Aryan</author>
  </book>
</library>
```

Example XPath Expressions:

Expression	Description
/library/book	Selects all <book> elements under <library>
//title	Selects all <title> elements anywhere
/library/book[1]	Selects the first <book> element
//book[author='Aryan']	Selects <book> where author is 'Aryan'

XQuery (XML Query Language)

What is XQuery?

- XQuery is a **powerful query language for XML** used to:
 - Query
 - Transform
 - Extract
 - Manipulate XML data
 - Think of it as **SQL for XML**.
 - XQuery is **built on XPath** and includes all XPath capabilities.
-

XQuery Syntax Example:

```
xquery
CopyEdit
for $b in /library/book
where $b/author = "Soumodeep"
return $b/title
```

Output:

```
php-template
CopyEdit
<title>XML Fundamentals</title>
```

XPath vs XQuery – Comparison Table

Feature	XPath	XQuery
Purpose	Navigate and select parts of XML	Query, transform, and manipulate XML
Functionality	Limited to path navigation	Full programming features (loops, if, etc.)
Is Based On	XML Tree structure	XPath + FLWOR expressions
Return Type	Node set, string, number, boolean	XML nodes, values, or even HTML output
Complex Queries	Not suitable	Designed for complex operations
Used In	XSLT, XQuery, Schemas	XML DBs, Web services, data extraction
Supports Conditions	Limited	Fully supports <code>if</code> , <code>for</code> , <code>where</code> , etc.
Learning Curve	Easier	More advanced and powerful