UNIT - 5:

Transaction Concept: Transaction State, ACID properties, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability, lock based, time stamp based, optimistic, concurrency protocols, Deadlocks, Failure Classification, Storage, Recovery and Atomicity, Recovery algorithm.

Storage and File Structure: Overview of Physical Storage Media, RAID, File Organization.

Introduction to Indexing Techniques: B+ Trees, operations on B+ Trees, Hash Based Indexing.

A transaction is a unit of program execution that accesses and possibly updates various data items.
E.g., transaction to transfer $50 from account A to account B:

1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions

Example of Fund Transfer
Transaction to transfer $50 from account A to account B:

1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

- Atomicity requirement - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

Failure could be due to software or hardware
The system should ensure that updates of a partially executed transaction are not reflected in the database

- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.
- Consistency requirement in above example:

The sum of A and B is unchanged by the execution of the transaction
In general, consistency requirements include:
Explicitly specified integrity constraints such as primary keys and foreign keys
Implicit integrity constraints - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
A transaction must see a consistent database.
During transaction execution the database may be temporarily inconsistent.
When the transaction completes successfully the database must be consistent - Erroneous transaction logic can lead to inconsistency

- Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum A + *B* will be less than it should be).

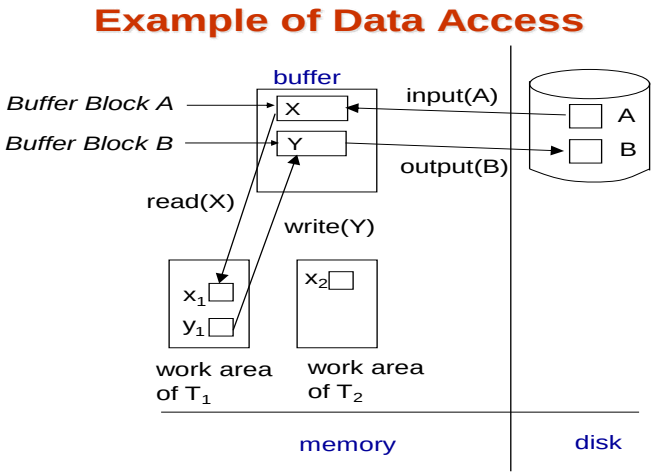
	T1	**T2**
1.	**read**(*A*)	
2.	*A* := *A* – 50	

3. **write**(A)

read(A), read(B), print(A+B)

4. **read**(B)
5. $B := B + 50$
6. **write**(B

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits.



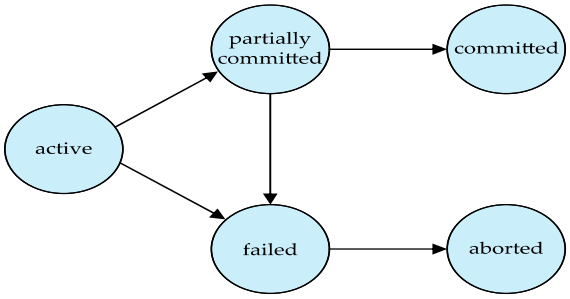
ACID Properties

A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$).

T_1	T_2
read (A) $A := A - 50$	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B)
write (A) read (B) $B := B + 50$ write (B) commit	
	$B := B + temp$ write (B) commit

Serializability

Basic Assumption – Each transaction preserves database consistency.

Thus, serial execution of a set of transactions preserves database consistency.

A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. Conflict serializability
2. View Serializability

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
    1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
    2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
    3. l_i = **write**(Q), l_j = **read**(Q). They conflict
    4. l_i = **write**(Q), l_j = **write**(Q). They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
- If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule
- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Anomalies with Interleaved Execution

1. Reading Uncommitted Data (WR Conflicts, "**dirty reads**"):

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
	C
read (B)	
write (B)	
Abbort	

2. Unrepeatable Reads (RW Conflicts):

T_1	T_2
read (A)	
	read (A)
	write (A)
	C
read (A)	
write (B)	
C	

3. Overwriting Uncommitted Data (WW Conflicts):

T_1	T_2
write (A)	
	write (A)
	write (B)
	C
write (B)	
C	

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
  1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**

Other Notions of Serializability

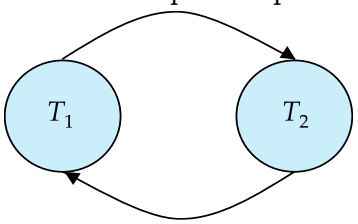
- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B$ - 10
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

- Determining such equivalence requires analysis of operations other than read and write.

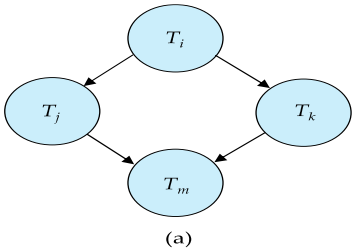
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- Precedence graph — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- Example of a precedence graph:

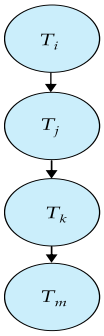


Test for Conflict Serializability

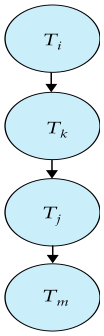
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which takes order n^2 time, where n is the number of vertices in the graph.
    - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
    - This is a linear order consistent with the partial order of the graph.
    - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
        - Are there others?



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus, existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur;
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.
- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- connection.setAutoCommit(false);
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
  - E.g. In SQL **set transaction isolation level serializable**
  - E.g. in JDBC - connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)

Implementation of Isolation Levels

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a “snapshot” of the database

Transactions as SQL Statements

- E.g., Transaction 1:
  select *ID, name* **from** *instructor* **where** *salary* > 90000
- E.g., Transaction 2:
  insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000)
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?

- Instance of the **phantom phenomenon**

- Also consider T3 below, with Wu's salary = 90000

  update *instructor*
  set *salary* = *salary* * 1.1
  where *name* = 'Wu'

- Key idea: Detect "**predicate**" conflicts, and use some form of "**predicate locking**"

Concurrency Control

Purpose of Concurrency Control

To enforce Isolation (through mutual exclusion) among conflicting transactions.
To preserve database consistency through consistency preserving execution of transactions.
To resolve read-write and write-write conflicts.

Example:
In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Two-Phase Locking Techniques

- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock (X). Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

Two-Phase Locking Techniques: Essential components

- Two locks modes:
  - (a) shared (read) (b) exclusive (write).
- Shared mode: shared lock (X)
  - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive mode: Write lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
- Conflict matrix

	S	X
S	true	false
X	false	false

Lock-Based Protocols

A lock is a mechanism to control concurrent access to a data item
Data items can be locked in two modes :

1. exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using lock-X instruction.
2. shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction.

Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.
Lock-compatibility matrix

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

```
T2: lock-S(A);
    read (A);
    unlock(A);
    lock-S(B);
    read (B);
    unlock(B);
    display(A+B)
```

Locking as above is not sufficient to guarantee serializability — if A and B get updated in-between the read of A and B, the displayed sum would be wrong.
A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

The Two-Phase Locking Protocol
This protocol ensures conflict-serializable schedules.
Phase 1: Growing Phase
Transaction may obtain locks
Transaction may not release locks
Phase 2: Shrinking Phase
Transaction may release locks
Transaction may not obtain locks

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).
There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
Given a transaction Ti that does not follow two-phase locking, we can find a transaction Tj that uses two-phase locking, and a schedule for Ti and Tj that is not conflict serializable.

Lock Conversions
Two-phase locking with lock conversions:

- First Phase:
  - can acquire a lock-S on item
  - can acquire a lock-X on item
  - can convert a lock-S to a lock-X (upgrade)
- Second Phase:
  - can release a lock-S
  - can release a lock-X
  - can convert a lock-X to a lock-S (downgrade)

This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks
A transaction Ti issues the standard read/write instruction, without explicit locking calls.
The operation read(D) is processed as:

```
if Ti has a lock on D
  then
        read(D)
  else begin
        if necessary wait until no other
            transaction has a lock-X on D
        grant Ti a  lock-S on D;
        read(D)
       end
```

write(D) is processed as:

```
if Ti has a  lock-X on D
  then
   write(D)
 else begin
     if necessary wait until no other transaction has any lock on D,
     if Ti has a lock-S on D
        then
          upgrade lock on D  to lock-X
       else
          grant Ti a lock-X on D
       write(D)
  end;
```

All locks are released after commit or abort

Deadlocks

Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

Neither T3 nor T4 can make progress — executing lock - S(B) causes T4 to wait for T3 to release its lock on B, while executing lock-X(A) causes T3 to wait for T4 to release its lock on A.
Such a situation is called a deadlock.
To handle a deadlock one of T3 or T4 must be rolled back and its locks released.

Two-phase locking does not ensure freedom from deadlocks.
In addition to deadlocks, there is a possibility of starvation.
Starvation occurs if the concurrency control manager is badly designed.

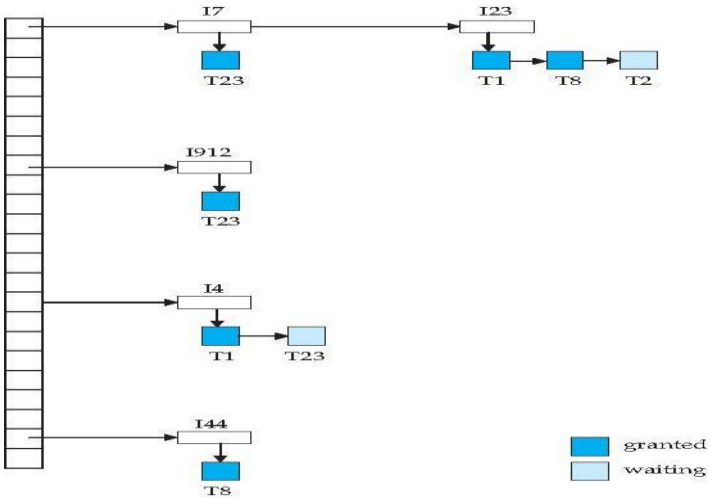
For example:
A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
The same transaction is repeatedly rolled back due to deadlocks.
Concurrency control manager can be designed to prevent starvation.

The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
When a deadlock occurs there is a possibility of cascading roll-backs.
Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking -- a transaction must hold all its exclusive locks till it commits/aborts.
Rigorous two-phase locking is even stricter. Here, all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Implementation of Locking

- A lock manager can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a lock table to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



- Dark blue rectangles indicate granted locks; light blue indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
- lock manager may keep a list of locks held by each transaction, to implement this efficiently

Deadlock Handling

System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
Deadlock prevention protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:

- Require that each transaction locks all its data items before it begins execution (predeclaration).
- Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order.

More Deadlock Prevention Strategies

Following schemes use transaction timestamps for the sake of deadlock prevention alone.

wait-die scheme — non-preemptive

- older transaction may wait for younger one to release data item. (older means smaller timestamp) Younger transactions never Younger transactions never wait for older ones; they are rolled back instead.
- a transaction may die several times before acquiring needed data item

wound-wait scheme — preemptive

- older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
- may be fewer rollbacks than wait-die scheme.

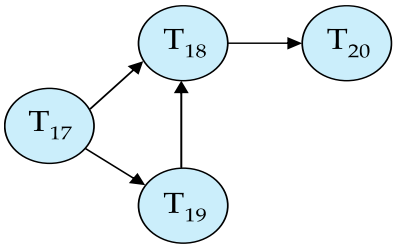
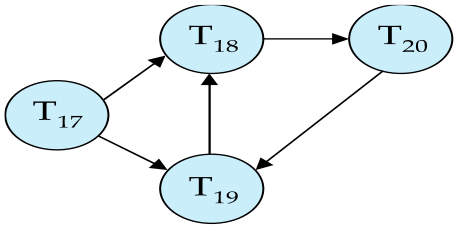
Both in wait-die and in wound-wait schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Timeout-Based Schemes:

- A transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
- Thus, deadlocks are not possible
- simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection

- Deadlocks can be described as a wait-for graph, which consists of a pair G = (V,E),
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair Ti -> Tj.
- If Ti -> Tj is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.
- When Ti requests a data item currently being held by Tj, then the edge Ti -> Tj is inserted in the wait-for graph. This edge is removed only when Tj is no longer holding a data item needed by Ti.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

	
Wait-for graph without a cycle	Wait-for graph with a cycle

Deadlock Recovery

When deadlock is detected: Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.

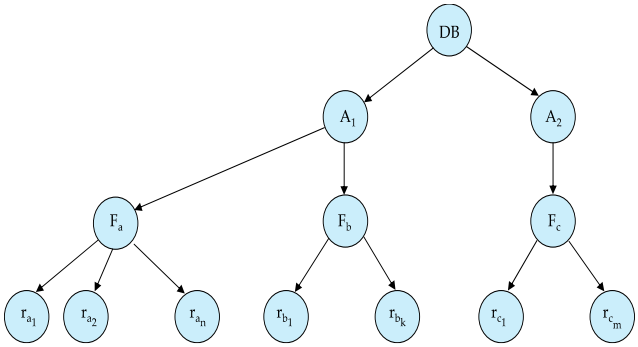
Rollback -- determine how far to roll back transaction

- Total rollback: Abort the transaction and then restart it.
- More effective to roll back transaction only as far as necessary to break deadlock.

Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree.
- When a transaction locks a node in the tree explicitly, it implicitly locks all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - fine granularity (lower in tree): high concurrency, high locking overhead
  - coarse granularity (higher in tree): low locking overhead, low concurrency



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  - The lock compatibility matrix must be observed.
  - The root of the tree must be locked first, and may be locked in any mode.

- A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
- A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
- T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
- T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)
  - If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
    - Hence, the **read** operation is rejected, and T_i is rolled back.
  - If TS(T_i) ≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).
- Suppose that transaction T_i issues **write**(Q).
  - If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the **write** operation is rejected, and T_i is rolled back.
  - If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
    - Hence, this **write** operation is rejected, and T_i is rolled back.
  - Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

- A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by T_j must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.
  1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
  2. **Validation phase**: Transaction T_i performs a "validation test'' to determine if local variables can be written without violating serializability.
  3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation
- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase

RECOVERY ALGORITHMS

1. Introduction to Recovery in DBMS

A Database Management System (DBMS) ensures data integrity and consistency. However, failures such as system crashes, transaction failures, or disk crashes can lead to data loss or inconsistency. Recovery algorithms help restore the database to a consistent state while maintaining the ACID (Atomicity, Consistency, Isolation, Durability) properties.

2. Types of DBMS Failures

- Transaction Failure – Due to logical errors, deadlocks, or system constraints.
- System Crash – Due to hardware or software failure, causing loss of volatile memory.
- Disk Failure – Due to disk corruption or hardware failure leading to permanent data loss.

3. Recovery Techniques

A. Log-Based Recovery

- A log file stores transaction details before changes are applied to the database.
- Write-Ahead Logging (WAL): Changes must be recorded in the log before being written to the database.
- Types of Log-Based Recovery:
- Deferred Update (No-Undo/Redo): Updates are applied only after a transaction commits.
- Immediate Update (Undo/Redo): Changes are applied before the commit, requiring undo and redo operations.

B. Checkpointing

- A checkpoint is a saved database state to speed up recovery.
- The system does not need to scan the entire log during recovery, only from the last checkpoint.

C. Shadow Paging

- Maintains two copies: current page table and shadow page table.
- The shadow page table remains unchanged until the transaction is committed.
- No need for logs, but inefficient for large databases.

D. ARIES Recovery Algorithm (Used in modern DBMS)

- Analysis Phase: Identifies active transactions at the time of failure.
- Redo Phase: Reapplies committed transactions.
- Undo Phase: Rolls back uncommitted transactions.

STORAGE AND FILE STRUCTURE

Storage types in DBMS:

The records in databases are stored in file formats. Physically, the data is stored in electromagnetic format on a device. The electromagnetic devices used in database systems for data storage are classified as follows:

1. Primary Memory
2. Secondary Memory
3. Tertiary Memory

Types of Memory

1. Primary Memory

The primary memory of a server is the type of data storage that is directly accessible by the central processing unit, meaning that it doesn't require any other devices to read from it. The primary memory must, in general, function flawlessly with equal contributions from the electric power supply, the hardware backup system, the supporting devices, the coolant that moderates the system temperature, etc.

- The size of these devices is considerably smaller and they are volatile.
- According to performance and speed, the primary memory devices are the fastest devices, and this feature is in direct correlation with their capacity.
- These primary memory devices are usually more expensive due to their increased speed and performance.

The cache is one of the types of Primary Memory.

- **Cache Memory:** Cache Memory is a special very high-speed memory. It is used to speed up and synchronize with a high-speed CPU. Cache memory is costlier than main memory or disk memory but more economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU.

2. Secondary Memory

Data storage devices known as secondary storage, as the name suggests, are devices that can be accessed for storing data that will be needed at a later point in time for various purposes or database actions. Therefore, these types of storage systems are sometimes called backup units as well. Devices that are plugged or connected externally fall under this memory category, unlike primary memory, which is part of the CPU. The size of this group of devices is noticeably larger than the primary devices and smaller than the tertiary devices.

- It is also regarded as a temporary storage system since it can hold data when needed and delete it when the user is done with it. Compared to primary storage devices as well as tertiary devices, these secondary storage devices are slower with respect to actions and pace.
- It usually has a higher capacity than primary storage systems, but it changes with the technological world, which is expanding every day.

3. Tertiary Memory

For data storage, Tertiary Memory refers to devices that can hold a large amount of data without being constantly connected to the server or the peripherals. A device of this type is connected either to a server or to a device where the database is stored from the outside.

- Due to the fact that tertiary storage provides more space than other types of device memory but is most slowly performing, the cost of tertiary storage is lower than primary and secondary. As a means to make a backup of data, this type of storage is commonly used for making copies from servers and databases.
- The ability to use secondary devices and to delete the contents of the tertiary devices is similar.

Memory Hierarchy

A computer system has a hierarchy of memory. Direct access to a CPU's main memory and inbuilt registers is available. Accessing the main memory takes less time than running a CPU. Cache memory is introduced to minimize this difference in speed. Data that is most

frequently accessed by the CPU resides in cache memory, which provides the fastest access time to data. Fastest-accessing memory is the most expensive. Although large storage devices are slower and less expensive than CPU registers and cache memory, they can store a greater amount of data.

1. Magnetic Disks

Present-day computer systems use hard disk drives as secondary storage devices. Magnetic disks store information using the concept of magnetism. Metal disks are coated with magnetizable material to create hard disks. Spindles hold these disks vertically. As the read/write head moves between the disks, it de-magnetizes or magnetizes the spots under it. There are two magnetized spots: 0 (zero) and 1 (one). Formatted hard disks store data efficiently by storing them in a defined order. The hard disk plate is divided into many concentric circles, called tracks. Each track contains a number of sectors. Data on a hard disk is typically stored in sectors of 512 bytes.

2. Redundant Array of Independent Disks(RAID)

In the Redundant Array of Independent Disks technology, two or more secondary storage devices are connected so that the devices operate as one storage medium. A RAID array consists of several disks linked together for a variety of purposes. Disk arrays are categorized by their RAID levels.

FILE ORGANIZATION

File organization is the way data is stored in a database system. It concerns the organization of records on a storage media in a way that makes retrieval of information and storage possible. They may use the storage medium as the core storage, which could be the hard disk, that is the SSD, or any other media.

The primary goals of file organization are:

1. **Quick Data Access:** Cutting down the time it takes to search for some information.
2. **Efficient Storage Utilization:** Making sure that there is little wastage of space in storage compartments.
3. **Ease of Maintenance:** Reduction of the complexity of either inserting, deleting, or modifying data.

Types of File Organization in DBMS

File organization methods are broadly categorized into the following types:

1. Heap (Unordered) File Organization

Heap file organization doesn't organize records in any specific manner; the records are also called run units. New data is added more at the end of the file this is also known as appending. This is a typical and simple approach but when it comes to the aspect of trying to find a particular record, it is not efficient.

Advantages:

1. Simple to implement.
2. Supplied for when lots of records have been inserted at one time.

Disadvantages:

1. Inefficient due to the fact that the whole file must be searched.
2. Not suitable when dealing with large amounts of data.

Use Cases:

1. Most appropriate in cases where the database is compact, or the search operations do not take place regularly

2. Sequential File Organization

Here, records are stored in place and order either in relation to a specific key or in some other pre-planned way. It is best suited for use when data need to be sorted before being used in an application.

Advantages:

1. For range-based queries, it is fast as well.
2. Easy to maintain sorted data.

Disadvantages:

1. While deletions and insertions can also be carried out, they are usually a little bit tricky and time-consuming.
2. It will not be appropriate for dynamic data.

Use Cases:

1. Applicable in conditions where data is processed in large portions such as payroll or inventory control.

3. Clustered File Organization

In clustered file organization, related records are stored on the same storage block. This makes a reduction of the I/O operations used to fetch related data from the disk.

Advantages:

1. Improves retrieval for the search string, which involves data that is connected in some way.
2. Reduces the time and cost of join operations.

Disadvantages:

1. Hard to implement and sustain.
2. Not suitable for data that is not likely to be clustered; and not good for large sets of data with little or no such correlations.

Use Cases:

1. Operated in contexts with JOIN expression commonly employed.

4. Hashed File Organization

In hashed file organization, keys are mapped with locations of the storage medium by using a hash function. Files are kept at such hashed places or records are usually kept at such locations.

Advantages:

1. Efficient access for point queries, in circumstances when cold storage may be possible.
2. Uniform distribution reduces on wastage of storage space.

Disadvantages:

1. This means that their performance for range queries is quite poor.
2. Collision mostly makes data retrieval challenging.

Use Cases:

1. Ideally used where point access is frequent such as lookup tables.

5. Indexed File Organization

This method involves the use of an index that stores pointers toward the record. Actual data is kept in another file while the index has pointers to the data records.

Advantages:

1. Improves search efficiency.

2. Can be used to identify objects in a range as well as point queries.

Disadvantages:

1. It also needs extra space for the index.
2. This causes insert and delete operations to take longer time because of work done by the index.

Use Cases:

1. Popular in databases even for quick search operations like that of search engines.

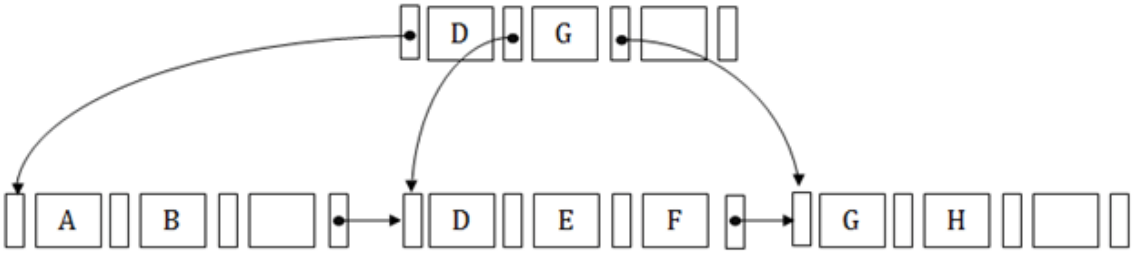
INTRODUCTION TO INDEXING TECHNIQUES

B+ Tree

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

Structure of B+ Tree

- In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.
- It contains an internal node and leaf node.



Internal node

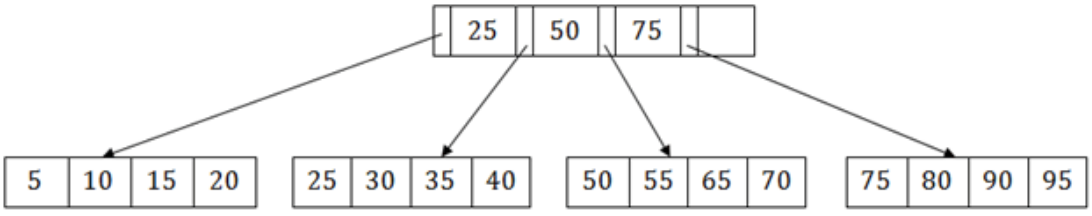
- An internal node of the B+ tree can contain at least n/2 record pointers except the root node.
- At most, an internal node of the tree contains n pointers.

Leaf node

- The leaf node of the B+ tree can contain at least n/2 record pointers and n/2 key values.
- At most, a leaf node contains n record pointer and n key values.
- Every leaf node of the B+ tree contains one block pointer P to point to next leaf node.

Searching a record in B+ Tree

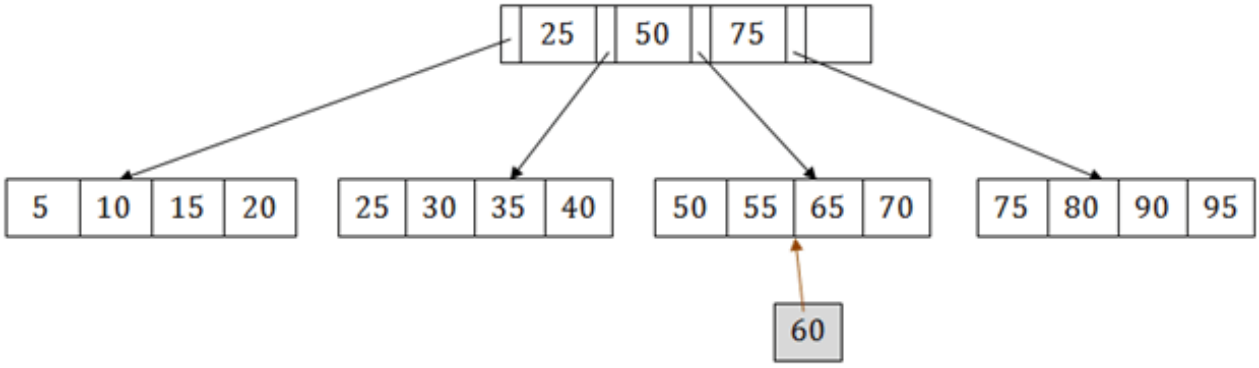
Suppose we have to search 55 in the below B+ tree structure. First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55.
So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



B+ Tree Insertion

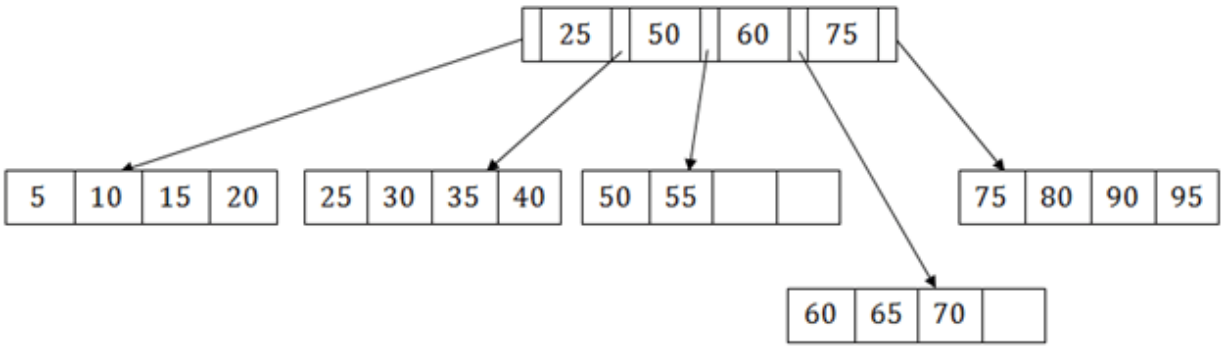
Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

In this case, we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order.



The 3rd leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to a new leaf node.

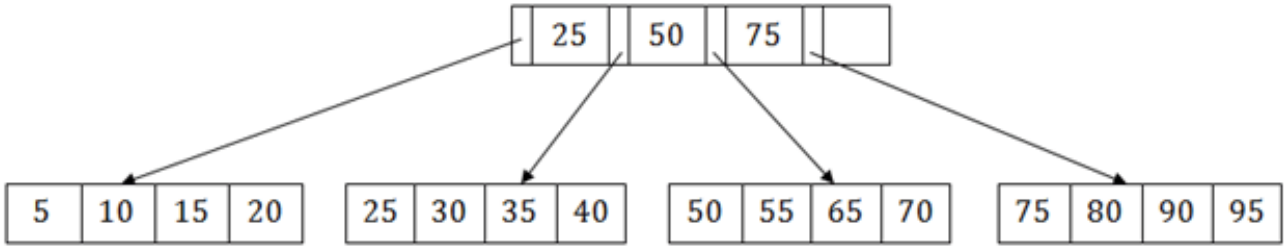


This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

B+ Tree Deletion

Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as well as from the 4th leaf node too. If we remove it from the intermediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.

After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:



Hashing

Hashing in DBMS is a technique to quickly locate a data record in a database irrespective of the size of the database. For larger databases containing thousands and millions of records, the indexing data structure technique becomes very inefficient because searching a specific record through indexing will consume more time. This doesn't align with the goals of DBMS, especially when performance and data retrieval time are minimized. So, to counter this problem hashing technique is used. In this article, we will learn about various hashing techniques.

What is Hashing?

The hashing technique utilizes an auxiliary hash table to store the data records using a hash function. There are 2 key components in hashing:

- **Hash Table:** A hash table is an array or data structure and its size is determined by the total volume of data records present in the database. Each memory location in a hash table is called a '***bucket***' or hash indice and stores a data record's exact location and can be accessed through a hash function.
- **Bucket:** A bucket is a memory location (index) in the hash table that stores the data record. These buckets generally store a disk block which further stores multiple records. It is also known as the hash index.
- **Hash Function:** A hash function is a mathematical equation or algorithm that takes one data record's primary key as input and computes the hash index as output.

Hash Function

A hash function is a mathematical algorithm that computes the index or the location where the current data record is to be stored in the hash table so that it can be accessed efficiently later. This hash function is the most crucial component that determines the speed of fetching data.

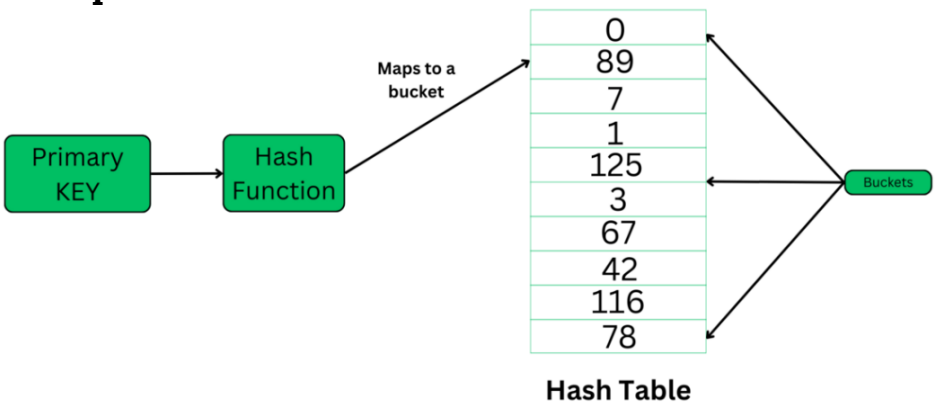
Working of Hash Function

The hash function generates a hash index through the primary key of the data record.

Now, there are 2 possibilities:

1. The hash index generated isn't already occupied by any other value. So, the address of the data record will be stored here.
2. The hash index generated is already occupied by some other value. This is called collision so to counter this, a collision resolution technique will be applied.
3. Now whenever we query a specific record, the hash function will be applied and returns the data record comparatively faster than indexing because we can directly reach the exact location of the data record through the hash function rather than searching through indices one by one.

Example:



Hash Table

Dynamic Hashing

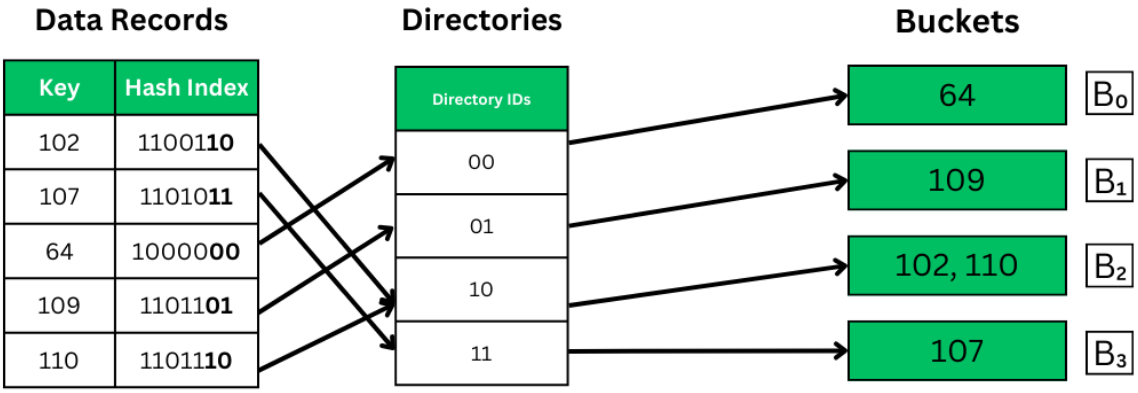
Dynamic hashing is also known as extendible hashing, used to handle database that frequently changes data sets. This method offers us a way to add and remove data buckets on demand dynamically. This way as the number of data records varies, the buckets will also grow and shrink in size periodically whenever a change is made.

Properties of Dynamic Hashing

- The buckets will vary in size dynamically periodically as changes are made offering more flexibility in making any change.
- Dynamic Hashing aids in improving overall performance by minimizing or completely preventing collisions.
- **It has the following major components:** Data bucket, Flexible hash function, and directories
- A flexible hash function means that it will generate more dynamic values and will keep changing periodically asserting to the requirements of the database.
- Directories are containers that store the pointer to buckets. If bucket overflow or bucket skew-like problems happen to occur, then bucket splitting is done to maintain efficient retrieval time of data records. Each directory will have a directory id.
- **Global Depth:** It is defined as the number of bits in each directory id. The more the number of records, the more bits are there.

Working of Dynamic Hashing

Example: If global depth: k = 2, the keys will be mapped accordingly to the hash index. K bits starting from LSB will be taken to map a key to the buckets. That leaves us with the following 4 possibilities: 00, 11, 10, 01.



Dynamic Hashing - mapping

As we can see in the above image, the k bits from LSBs are taken in the hash index to map to their appropriate buckets through directory IDs. The hash indices point to the directories, and the k bits are taken from the directories' IDs and then mapped to the buckets. Each bucket holds the value corresponding to the IDs converted in binary.