Module 4

1. Construct Functional Dependency. Give Inference rules for Functional Dependencies.

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

X → Y

The left side of FD is known as a determinant, the right side of the production is known as a dependent.
For example, assume we have an employee table with attributes:
Emp_Id, Emp_Name, Emp_Address
Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.
Functional dependency can be written as:

Emp_Id → Emp_Name

We can say that Emp_Name is functionally dependent on Emp_Id.
Functional Dependencies:

- Are used to specify formal measures of the "goodness" of relational designs .
- And keys are used to define normal forms for relations.
- Are constraints that are derived from the meaning and interrelationships of the data attributes.

The Inference Rules for Functional Dependencies are:

- IR1: Reflexive Rule: If Y subset-of X, then X -> Y
- IR2: Augmentation Rule: If X -> Y, then XZ -> YZ
- IR3: Transitive Rule: If X -> Y and Y -> Z, then X -> Z
- IR4: Decomposition Rule: If X -> YZ, then X -> Y and X -> Z
- IR5: Union Rule: If X -> Y and X -> Z, then X -> YZ
- IR6: Psuedotransitivity Rule: If X -> Y and WY -> Z, then WX -> Z

The rules IR1, IR2 and IR3 are known as Armstrong's Inference Rules, they are the basic and complete rules on which IR4, IR5 and IR6 are derived from.

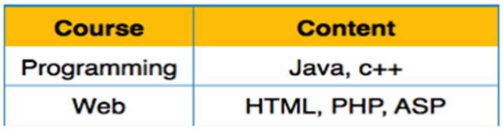
2. Explain 1NF with example.

A table is considered to be in 1NF if all the fields contain only scalar/atomic values (as opposed to list of values).
The First Normal Form disallows:

- composite attributes
- multivalued attributes
- nested relations; attributes whose values for an individual tuple are non-atomic

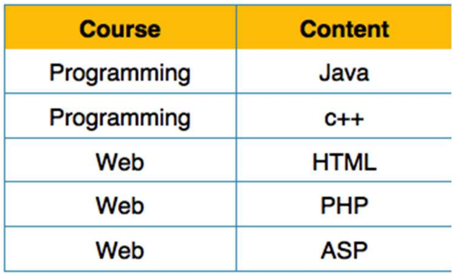
To convert a given Schema into 1NF:

- Place all items that appear in the repeating group on a new table.
- Designate a primary key for each new table produced.
- Duplicate in the new table the primary key of the table from which the repeating group was extracted or vice versa.



Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

Not in 1NF



Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

Converted to 1NF

3. Explain 2NF with example.

A relation schema R is in second normal form (2NF) if every non-prime attribute A in R is fully functionally dependent on the primary key. R can be decomposed into 2NF relations through 2NF normalization.

For a table to be in 2NF, there are two requirements:

- The database is in first normal form.
- All non-key attributes in the table must be functionally dependent on the entire primary key.

To convert a given table to 2NF:

- If a data item is fully functionally dependent on only a part of the primary key, move that data item and that part of the primary key to a new table.
- If other data items are functionally dependent on the same part of the key, place them in the new table also.
- Make the partial primary key copied from the original table the primary key for the new table. Place all items that appear in the repeating group on a new table.

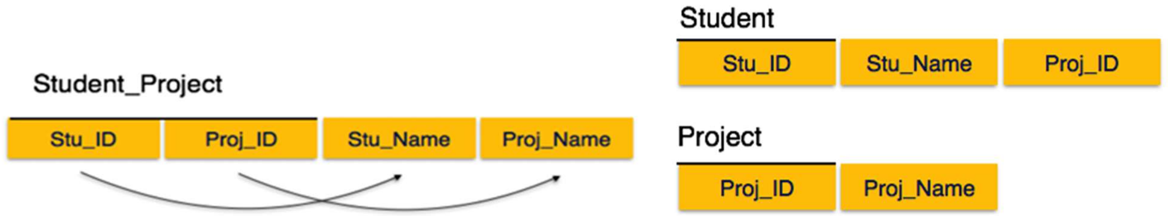
Example 1 (Convert to 2NF)

Old Scheme → {Title, PubId, AuId , Price, AuAddress}

New Scheme → {Title, PubId, AuId , Price}

New Scheme → {AuId , AuAddress}

Example 2



We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attributes individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called partial dependency, which is not allowed in Second Normal Form. We broke the relation in two. So there exists no partial dependency.

4. Explain 3NF and BCNF with example.

3NF

A relation schema R is in third normal form (3NF) if it is in 2NF, and no non-prime attribute A in R is transitively dependent on the primary key. R can be decomposed through 3NF normalization.

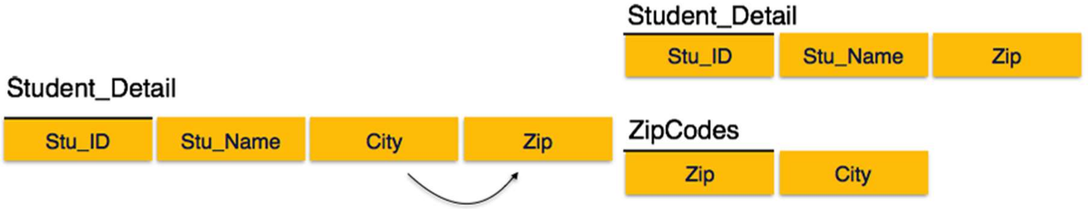
Transitive functional dependency: a FD X -> Z that can be derived from two FDs X -> Y and Y -> Z

For a table to be in 3NF, there are two requirements:

- The table should be second normal form.
- No attribute is transitively dependent on the primary key

To convert a given schema into 3NF:

- Move all items involved in transitive dependencies to a new entity.
- Identify the primary key for the new entity.
- Place the primary key for the new entity as a foreign key on the original entity.



BCNF

A relation schema R is in Boyce-Codd Normal Form (BCNF) if whenever an FD X -> A holds in R, then X is a super key of R. There exist relations that are in 3NF but not in BCNF.

In the above image, Stu_ID is the super-key in the relation Student_Detail and Zip is the super-key in the relation ZipCodes. So, Stu_ID → Stu_Name, Zip and Zip → City Which confirms that both the relations are in BCNF.

Third normal form and BCNF are not same if the following conditions are true:

- The table has two or more candidate key.
- At least two of the candidate keys are composed of more than one attribute.
- The keys are not disjointed i.e. The composite candidate keys share some attributes.

To convert a schema into BCNF:

- Place the two candidate primary keys in separate entities.
- Place each of the remaining data items in one of the resulting entities according to its dependency on the primary key.

Example 1 (Convert to BCNF)

Old Scheme → {City, Street, ZipCode }

New Scheme1 → {ZipCode, Street}

New Scheme2 → {City, Street}

Loss of relation {ZipCode} → {City}

Alternate New Scheme1 → {ZipCode, Street }

Alternate New Scheme2 → {ZipCode, City}

5. Consider the below Relation

R{City, Street, HouseNumber, HouseColor, CityPopulation}
Assume key as {City, Street, HouseNumber}
The Dependencies are:
{City, Street, HouseNumber} →{HouseColor}
{City} →{CityPopulation}

Check whether the given R is in 2NF? If not convert into 2NF.

Scheme → {City, Street, HouseNumber, HouseColor, CityPopulation}
Key →{City, Street, HouseNumber}
{City, Street, HouseNumber} →{HouseColor}
{City} → {CityPopulation}
CityPopulation does not belong to any key.
CityPopulation is functionally dependent on the City which is a proper subset of the key

Old Scheme → {City , Street, HouseNumber, HouseColor, CityPopulation}
New Scheme → {City, Street, HouseNumber, HouseColor}
New Scheme → {City, CityPopulation}

6. Consider the relation

Emp-Proj ={SSN, Pnumber, Hours, Ename, Pname, Plocation}
Assume {SSN, Pnumber } as Primary key
The dependencies are:
{SSN, Pnumber}->Hours
SSN->Ename
Pnumber->{Pname, Plocation}

Normalize the above relation to 3NF

Old Scheme →{SSN, Pnumber, Hours, Ename, Pname, Plocation}
New Scheme →{SSN, Pnumber, Hours}
New Scheme → {SSN, Ename}
New Scheme → {Pnumber, Pname, Plocation}

7. Consider the following relation

R {Title, PubId, AuId, Price, AuAddress}
Assume primary key as {Title, PubId, AuId}
The Dependencies are
{Title, PubId, AuID} → {Price}
{AuID} → {AuAddress}

Check whether the given R is in 2NF? If not convert into 2NF.

Scheme → {Title, PubId, AuId, Price, AuAddress}
Key → {Title, PubId, AuId}
{Title, PubId, AuID} → {Price}
{AuID} → {AuAddress}
AuAddress does not belong to a key
AuAddress functionally depends on AuId which is a subset of a key

Old Scheme → {Title, PubId, AuId, Price, AuAddress}
New Scheme → {Title, PubId, AuId, Price}
New Scheme → {AuId, AuAddress}

8. Consider the following relation

R {Studio, StudioCity, CityTemp}
Assum e Primary Key as {Studio}
The Dependencies are:
{Studio} → {StudioCity}
{StudioCity} → {CityTemp}

Check whether the given R is in 3NF? If not convert into 3NF

Scheme → {Studio, StudioCity, CityTemp}
Primary Key → {Studio}
{Studio} →{StudioCity}
{StudioCity} → {CityTemp}
{Studio} →{CityTemp}
Both StudioCity and CityTemp depend on the entire key hence 2NF
CityTemp transitively depends on Studio hence violates 3NF

Old Scheme → {Studio , StudioCity, CityTemp}
New Scheme → {Studio , StudioCity}
New Scheme → {StudioCity , CityTemp}

9. Give the minimal cover Algorithm. Find the minimal cover using the minimal cover algorithm for the following functional dependency.

F = {B->A, D->A,AB->D}

The minimal cover (or canonical cover) for a set of functional dependencies F is an equivalent set of dependencies that is minimal, meaning that it has no extraneous attributes, and each functional dependency has a single attribute on the right-hand side. Here is the algorithm to find the minimal cover:

- ✓ **Decompose the FDs**: Ensure that every functional dependency in F has a single attribute on the right-hand side.
- ✓ **Remove extraneous attributes**: For each functional dependency X→A in F, check if any attribute in A is extraneous.
- ✓ **Remove redundant dependencies**: For each functional dependency X→A in F, check if it is redundant by computing the closure of the remaining dependencies and verifying if it still implies X→A.

The Algorithm is given as follows:

- Set F:=E
- Replace each FD
    - X → {A1, A2, A3 … An} in F by n FD’s
    - X → A1
    - X → A2
    - …
    - X → An
- For each FD X→A in F
    - For each attribute B that is an element of X
        - If F→{X→A}U (X-{B}→A) == F
            - Replace X→A with X-{B}→A in F
- For each remaining FD X→A in F
    - If {F – {X→A}} == F, then
        - Remove X→ A from F

- ✓ **Decompose the FDs**

The given FDs are already decomposed as each has a single attribute on the right-hand side:

F={B→A, D→A, AB→D}.

- ✓ **Remove Extraneous Attributes**

Check each FD for extraneous attributes.

Check AB→D.

- Consider A in AB→D. We need to check if (AB−A)→D (i.e., B→D) holds.
- Comparing AB → D with other FD in F.
  - B → A, use augmentation rule.
  - BB → AB which can be written as B → AB.
- Now, using the transitive property, we consider AB→D and B → AB.
- We get, B → D, therefore we can replace AB→D, with B→D in F.

- ✓ **Remove Redundant Dependencies**

Check if any FD is redundant by computing the closure of the remaining dependencies and seeing if it still implies the FD.

Check B→D:

- Remove B→A from F. The remaining set is F′={D→A, B→D}.
- Compute the closure of {B} using F′:
  - Start with {B}.
  - Dependencies in F′ can be applied since B→A, can be derived by applying transitive property on D→A and B→D .

Thus, B→A is redundant.

Minimal Cover

The minimal cover for F={B→A, D→A, AB→D} is: {D→A, B→D}

Module 5

1. Explain the desirable properties of Transactions.

Atomicity:

- **Definition**: Atomicity ensures that a transaction is treated as a single, indivisible unit of work. This means that either all the operations within the transaction are executed, or none of them are.
- **Importance**: This property ensures that the database remains in a consistent state even if a transaction fails. For example, if a transaction involves transferring money from one account to another, atomicity ensures that either both the debit and credit operations are completed or neither is, preventing scenarios where money could be lost or created from nothing.

Consistency Preservation:

- **Definition**: Consistency ensures that a transaction takes the database from one valid state to another, maintaining the integrity constraints defined in the database schema.
- **Importance**: Consistency guarantees that the database adheres to the rules and constraints defined, such as unique keys, foreign keys, and other business rules. For example, in a banking system, the total amount of money in all accounts should remain constant after a transaction if no money is being added or removed from the system.

Isolation:

- **Definition**: Isolation ensures that the operations of one transaction are not visible to other transactions until the transaction is committed. This property prevents transactions from interfering with each other.
- **Importance**: Isolation avoids the temporary update problem, where one transaction sees an intermediate state of another transaction. For instance, if two transactions are updating the same account balance, isolation ensures that one transaction's updates are not visible to the other until the first transaction is completed, thus preventing incorrect computations based on partial data.

Durability or Permanency:

- **Definition**: Durability ensures that once a transaction is committed, its changes are permanent and will not be lost, even in the event of a system failure.
- **Importance**: Durability guarantees that committed transactions are saved permanently. For example, in an e-commerce application, once an order is placed and committed, the order details should be stored permanently, ensuring that the order can be processed and fulfilled even if there is a subsequent system crash.

2. Explain Different states of Transactions with Diagram.

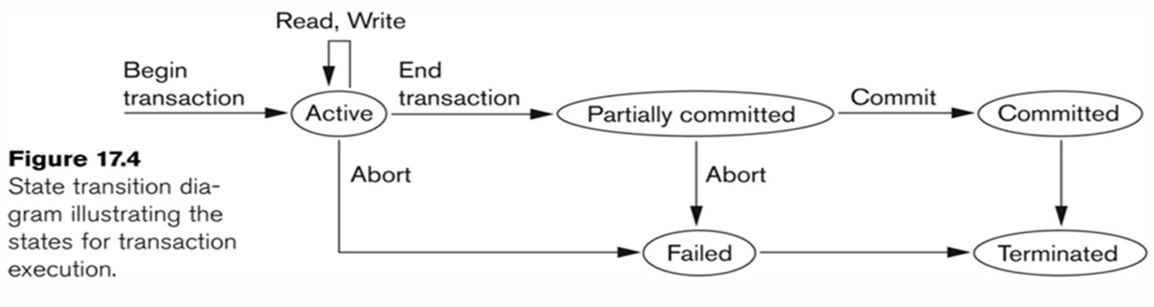


Figure 17.4
State transition diagram illustrating the states for transaction execution.

Active state: The transaction is currently executing its operations.

Partially committed state: The transaction has finished executing but has not yet made its changes permanent.

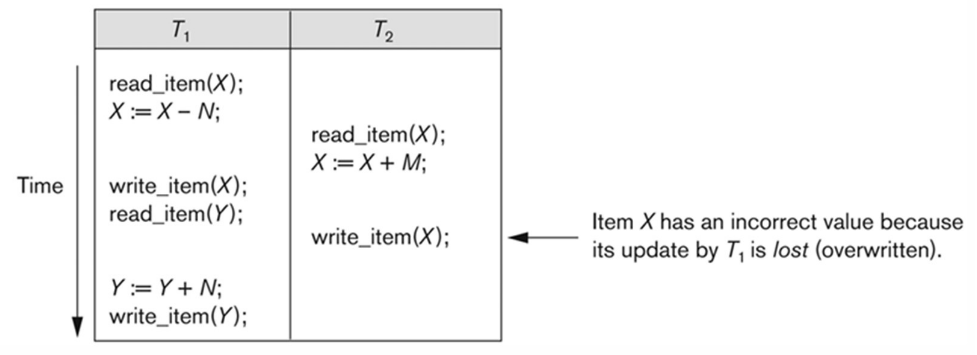
Committed state: The transaction has successfully completed, and all its changes are now permanent.

Failed state: The transaction has encountered an error or was aborted and cannot proceed.

Terminated state: The transaction has finished its execution, either by committing successfully or by failing and rolling back.

3. Explain Why concurrency is needed?

- The Lost Update Problem
    - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.



Time

T_1	T_2

read_item(X);
X := X − N;	

	read_item(X);
	X := X + M;

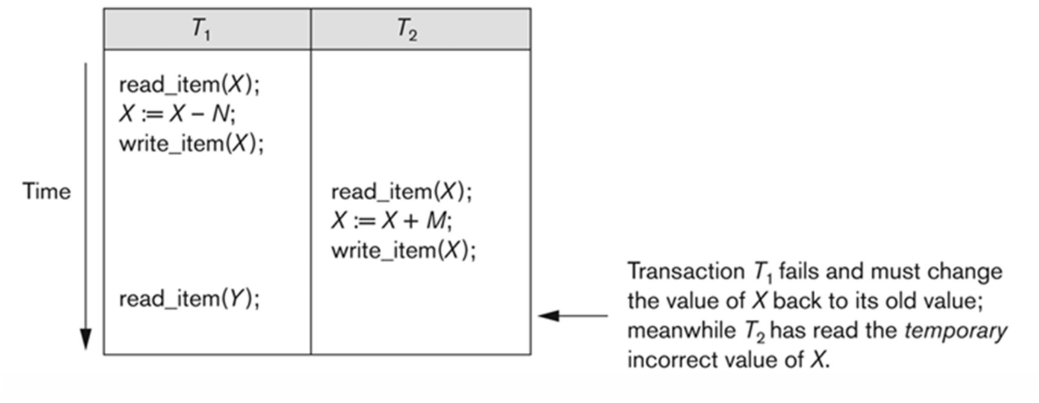
write_item(X);
read_item(Y);	

	write_item(X);

Y := Y + N;
write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

- The Temporary Update (or Dirty Read) Problem
    - This occurs when one transaction updates a database item and then the transaction fails for some reason.
    - The updated item is accessed by another transaction before it is changed back to its original value.



Time

T_1	T_2

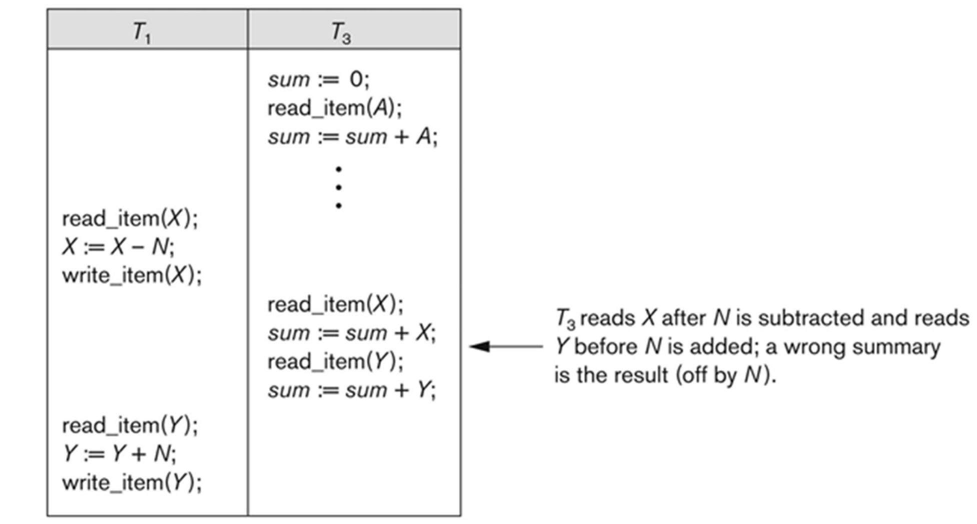
read_item(X);
X := X − N;
write_item(X);	

	read_item(X);
	X := X + M;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

- The Incorrect Summary Problem
    - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.



T_1	T_3

	sum := 0;
	read_item(A);
	sum := *sum* + A;
	⋮

read_item(X);
X := X − N;
write_item(X);	

	read_item(X);
	sum := *sum* + X;
	read_item(Y);
	sum := *sum* + Y;

read_item(Y);
Y := Y + N;
write_item(Y);	

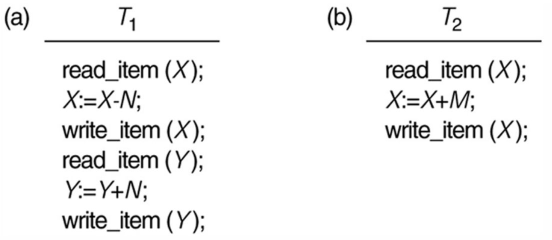
T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

4. **Develop the steps involved in read and write operations of transactions.**

READ AND WRITE OPERATIONS

Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

- read_item(X): Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
- read_item(X) command includes the following steps:
    - Find the address of the disk block that contains item X.
    - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
    - Copy item X from the buffer to the program variable named X.
- write_item(X): Writes the value of program variable X into the database item named X.
- write_item(X) command includes the following steps:
    - Find the address of the disk block that contains item X.
    - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
    - Copy item X from the program variable named X into its correct location in the buffer.
    - Store the updated block from the buffer back to disk (either immediately or at some later point in time).



```
(a)       T1                    (b)       T2
     ________________               ________________
     read_item (X);                 read_item (X);
     X:=X-N;                        X:=X+M;
     write_item (X);                write_item (X);
     read_item (Y);
     Y:=Y+N;
     write_item (Y);
```

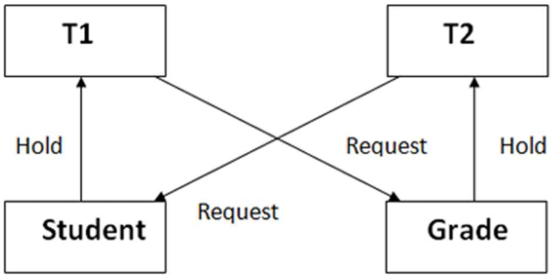
5. **Explain the reasons for the failure of transactions.**

- **A computer failure (system crash):** A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.
- **A transaction or system error**: Some operations in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.
- **Local errors or exception conditions detected by the transaction:** Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled. A programmed abort in the transaction causes it to fail.
- **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.
- **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
- **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

6. Explain Deadlock and Starvation.

Deadlock:

- **Definition**: Deadlock occurs when a set of transactions are waiting for each other in a circular chain, where each transaction holds a resource that the next transaction in the chain is waiting to acquire. This results in all transactions being stuck indefinitely.
- **Scenario**: For example, if transaction T1 holds a lock on resource R1 and is waiting for resource R2, while transaction T2 holds a lock on resource R2 and is waiting for resource R1, both transactions will be stuck in a deadlock.



- **Detection and Prevention**:
  - **Prevention Protocols**: Strategies include locking all required resources in advance or following a strict ordering of resource acquisition to prevent circular wait.
  - **Timestamp-Based Protocols**:
    - **Wait-Die Scheme**: Older transactions wait for resources held by younger ones, but younger transactions are killed and restarted if they request resources held by older ones.
    - **Wound-Wait Scheme**: Older transactions can preempt resources held by younger transactions, forcing the younger transactions to roll back.
  - **No Waiting Algorithm**: Immediately aborts and restarts transactions that cannot obtain a lock.
  - **Deadlock Detection**: Uses mechanisms like wait-for graphs to detect cycles indicating deadlock and then resolves them by aborting one or more transactions.
  - **Victim Selection**: Chooses which transaction to abort based on criteria like the age or priority of transactions.
  - **Timeouts**: Automatically abort transactions that wait longer than a predefined period.

Starvation:

- **Definition**: Starvation occurs when a transaction is repeatedly delayed indefinitely because other transactions are continually given preference. This can happen if a transaction is continually chosen as a victim during deadlock resolution.
- **Scenario**: For instance, if a low-priority transaction is consistently aborted in favor of higher-priority transactions, it may never get a chance to execute.
- **Solution**:
  - **Fairness Mechanisms**: Implementing first-come-first-served queues or ensuring fair scheduling can mitigate starvation by giving each transaction a chance to execute.
  - **Wound-Wait Consideration**: In the Wound-Wait scheme, ensure that younger transactions are not perpetually wounded by long-running older transactions to prevent starvation.

7. **Explain the characteristics of NoSQL Databases.**

- **Schema-less Data Model**:
  - **Definition**: NoSQL databases do not require a predefined schema, allowing for flexible and dynamic data structures.
  - **Example**: In a document database like MongoDB, each document can have a different structure, allowing for easy updates and changes to the data model without requiring a schema migration.
- **Horizontal Scalability**:
  - **Definition**: NoSQL databases are designed to scale out by adding more servers or nodes, rather than scaling up by adding more power to a single server.
  - **Example**: Distributed databases like Cassandra and HBase can handle large amounts of data and high transaction volumes by distributing the load across multiple servers.
- **Distributed Architecture**:
  - **Definition**: NoSQL databases use a distributed architecture, spreading data across multiple servers to improve performance and reliability.
  - **Example**: Apache Cassandra uses a peer-to-peer distributed system across its nodes, avoiding single points of failure and enabling efficient data distribution and replication.
- **Variety of Data Models**:
  - **Definition**: NoSQL databases support various data models, including key-value, document, column-family, and graph.
  - **Example**:
    - **Key-Value Stores**: Redis, where data is stored as a collection of key-value pairs.
    - **Document Stores**: MongoDB, where data is stored in documents (usually JSON or BSON).
    - **Column-Family Stores**: Cassandra, where data is stored in columns and rows.
    - **Graph Databases**: Neo4j, which stores data in nodes and edges, representing relationships.
- **Eventual Consistency**:
  - **Definition**: Many NoSQL databases embrace eventual consistency, meaning data updates will eventually propagate to all nodes, but may not be immediately consistent.
  - **Example**: Amazon DynamoDB ensures that all copies of data will converge to the same state eventually, allowing for high availability and partition tolerance.
- **Flexible and Efficient Storage**:
  - **Definition**: NoSQL databases are designed to handle large volumes of unstructured or semi-structured data efficiently.
  - **Example**: In Hadoop's HBase, data is stored in a highly compressed format, optimized for read and write operations on large datasets.
- **Support for Big Data and Real-Time Applications**:
  - **Definition**: NoSQL databases are often used in big data and real-time web applications due to their ability to handle large-scale, high-velocity data.
  - **Example**: Apache Cassandra is used by companies like Netflix to manage massive amounts of streaming data in real time.
- **Developer-Friendly**:
  - **Definition**: NoSQL databases provide APIs and query languages that are often more intuitive and easier to use for developers compared to traditional SQL.
  - **Example**: MongoDB uses a JSON-like query language that is simple for developers to understand and use, aligning closely with modern web development practices.

8. Explain Types of databases of NoSQL.

Document-Based Databases

Store data in documents using formats like JSON, BSON, or XML. Each document is a self-contained unit with flexible schema.

- **Features**:
  - Schema-less design allows for dynamic, flexible data structures.
  - Supports nested documents and arrays.
  - Often includes powerful querying capabilities, including indexing and aggregation.

Advantages	Disadvantages
Flexibility in data modeling.	Can be less efficient for complex transactions involving multiple documents.
Easy to scale horizontally.	Potential for data duplication due to denormalization.
Efficient for storing and retrieving complex hierarchical data.	
Good for applications with evolving schemas.	

- **Examples**: MongoDB, CouchDB

Key-Value Stores

Store data as a collection of key-value pairs, where each key is unique and maps directly to a value.

- **Features**:
  - Simple data model with high performance.
  - Efficient for quick lookups and simple data retrieval.
  - Often includes in-memory storage for fast access.

Advantages	Disadvantages
High performance and low latency.	Limited querying capabilities.
Easy to scale horizontally.	Inefficient for complex queries or relationships.
Flexible for various use cases, especially caching and session management.	Typically lacks built-in data integrity constraints.

- **Examples**: Redis, Riak

Column-Oriented Databases

Store data in columns rather than rows, with each column family containing rows with different numbers of columns.

- **Features**:
  - Optimized for read and write performance on large datasets.
  - Supports high compression and efficient storage.
  - Can handle large-scale data across distributed systems.

Advantages	Disadvantages
High performance for read/write operations.	Complex data modeling compared to other NoSQL types.
Efficient storage and retrieval for large datasets.	Potentially high learning curve for setup and maintenance.
Suitable for analytical applications and time-series data.	Can be inefficient for transactional applications.

- **Examples**: Apache Cassandra, HBase

Graph-Based Databases

- **Description**:
  - Store data in nodes and edges, with nodes representing entities and edges representing relationships between entities.
- **Features**:
  - Optimized for managing and querying relationships.
  - Supports complex queries on interconnected data.
  - Often includes ACID transaction support for data integrity.

Advantages	Disadvantages
Ideal for applications with complex relationships.	Less efficient for large-scale bulk data operations.
High performance for graph traversal operations.	May require specialized knowledge for effective use.
Flexible schema for evolving data structures.	Potentially more complex to scale horizontally.

- **Examples**: Neo4j, Amazon Neptune

9. Define the Graph database. List the advantages and Disadvantages of Graph databases.

Graph-Based Data Model:

A data model focusing on building relationships between data elements. Each element is stored as a node, and the associations between these nodes are represented by edges. Properties provide additional information about nodes and edges.

- **Nodes**: Represent instances of data or objects that need to be tracked.
- **Edges**: Represent the relationships or connections between nodes.
- **Properties**: Store information associated with nodes and edges.

Advantages of Graph Data Model

- **Structure**:
  - **Agility**: Graph structures are highly flexible and adaptable to changing data requirements.
- **Explicit Representation**:
  - **Relationship Clarity**: Relationships between entities are explicitly represented, making complex queries straightforward.
- **Real-Time Output**:
  - **Efficient Queries**: Queries return real-time results, providing quick insights into interconnected data.

Disadvantages of Graph Data Model

- **No Standard Query Language**:
  - **Varied Syntax**: Different graph databases use different query languages (e.g., Cypher for Neo4j, GraphQL for DGraph), leading to a lack of standardization.
- **Limited Suitability for Transactions**:
  - **Transactional Challenges**: Graph databases may not be well-suited for systems requiring high transaction throughput, compared to traditional relational databases.
- **Small User Base**:
  - **Support Issues**: The smaller user base may lead to challenges in finding support and resources for troubleshooting and development.

10. Compare NoSQL and RDBMS.

Aspect	**NoSQL**	**RDBMS**
Data Model	Schema-less or flexible schema (key-value, document, column, graph)	Structured schema (tables, rows, columns)
Schema Flexibility	Dynamic schema, accommodates unstructured data	Fixed schema, requires predefined structure
Scalability	Horizontal scaling (distributed across multiple servers)	Primarily vertical scaling (upgrading hardware), with some horizontal scaling options
Transaction Support	Eventual consistency; some support for ACID transactions	Strong ACID compliance (Atomicity, Consistency, Isolation, Durability)
Query Language	Custom or varied query languages (e.g., MQL, Cypher, GraphQL)	SQL (Structured Query Language)
Data Integrity	Limited support for constraints; focus on flexibility	Strong support for constraints; data normalization
Use Cases	Big data, real-time analytics, dynamic or evolving data	Structured data, complex queries, transactional systems
Replication	Distributed replication across multiple nodes or data centers	Master-slave or primary-replica replication for redundancy and scaling reads

11. Explain the need of Schema less database.

- **Flexibility in Data Storage**
    - **Dynamic Data**: Schemaless databases allow for the storage of diverse and evolving data types without a predefined schema. This flexibility is crucial for applications where the data model is not fully known or changes over time.
    - **Example**: In a document database like MongoDB, you can store documents with different structures in the same collection, adapting to changes in data requirements without schema migrations.
- **Handling Non-Uniform Data**
    - **Varied Records**: Schemaless databases handle records with different sets of fields effectively. This is useful for datasets where each record may have a different structure.
    - **Example**: In a key-value store like Redis, each key can hold data of any type, accommodating varying data formats without needing a uniform schema.
- **Simplified Data Management**
    - **Avoids Sparse Tables**: Traditional relational schemas can lead to sparse tables with many null columns when records have different attributes. Schemaless databases avoid this by allowing each record to contain only the data it needs.
    - **Example**: In a column-family database like Cassandra, each column family can store rows with different columns, avoiding the issue of unused or null columns.
- **Adaptability to Change**
    - **Evolving Requirements**: Schemaless databases are well-suited for projects where the data model evolves frequently. You can add or remove fields without worrying about schema modifications or data migrations.
    - **Example**: In a graph database like Neo4j, you can dynamically add new types of relationships or properties without changing the overall schema.
- **Implicit Schema in Application Code**
    - **Code Integration**: Although schemaless databases do not enforce a schema, the application code that interacts with the database often relies on an implicit schema. This means that data assumptions and structures are managed at the application level.
    - **Example**: An application might expect a field named email to always contain a string value. Even though the database is schemaless, the application code enforces this expectation.
- **Integration with Web Services**
    - **Encapsulation**: Encapsulating database interactions within a single application and using web services for integration can mitigate the challenges of a schemaless database. This approach centralizes schema management within the application layer.
    - **Example**: An application that provides an API that can handle schema changes internally and present a consistent interface to other applications.
- **Controlled Schema Changes**
    - **Schema Evolution**: While relational databases require schema changes to be explicitly defined, schemaless databases naturally accommodate evolving data requirements. However, it is still essential to manage schema changes carefully to maintain data consistency.
    - **Example**: For a document database, you can introduce new fields in documents as needed without schema constraints, but you must ensure that applications handling the data are updated accordingly.