**2023**

**Find out the candidate keys from the given functional dependency**

**B-> D; H->BC; E-> F G; G-> A; F->E; A->H**

To find the **candidate keys** for a relation schema given a set of **functional dependencies (FDs)**, we follow these steps:

**Step 1: Given Functional Dependencies**

B→D

H→BC

E→FG

G→A

F→E

A→H

**Step 2: Find the Closure of Attributes**

The **closure** of an attribute set X+ is the set of all attributes that can be determined using the given functional dependencies.

We first determine the **set of all attributes**:

{A,B,C,D,E,F,G,H}

To find the **candidate key**, we must find a **minimal set of attributes** that determines all attributes in the schema.

**Step 3: Compute Attribute Closures to Identify Candidate Keys**

We systematically check different attribute combinations to determine which ones are **superkeys** and identify the **minimal ones (candidate keys)**.

**Compute H+ (Closure of H)**

1. H→BC ⇒ Add B,C
2. B→D ⇒ Add D
3. C does not provide any new attributes.
4. H+={H,B,C,D}

H+ does not cover all attributes, so H **alone** is not a candidate key.

**Compute E+ (Closure of E)**

1. E→FG ⇒ Add F,G
2. F→E (Already included)
3. G→A ⇒ Add A
4. A→H ⇒ Add H
5. H→BC ⇒ Add B,C
6. B→D ⇒ Add D

Now, E+={E,F,G,A,H,B,C,D}, which includes all attributes in the schema.

Since E+ covers all attributes, **E is a superkey**.

**Step 4: Check Minimality**

A **candidate key** is a **minimal superkey**, meaning we cannot remove any attributes from it while still being able to determine all attributes.

* The closure of **E alone** determines **all attributes**, so **E is a candidate key**.
* Checking any proper subset of **E** (such as just F or just G) shows that no smaller subset alone can derive all attributes.

Thus, **E is the only candidate key**.

**Final Answer**

The **candidate key** is:

{E}

This means **E uniquely identifies all attributes in the relation**.

**(b) What are the benefits of controlled redundancy? How could you manage controlled redundancy in**

**a relational model?**

**Benefits of Controlled Redundancy**

**Controlled redundancy** in a relational database involves deliberately duplicating certain data elements to achieve specific advantages while minimizing potential downsides. The benefits of controlled redundancy include:

1. **Improved Performance:**
   * **Faster Query Performance:** Redundant data can reduce the need for complex joins and improve read performance. For example, storing a customer's full address in the order table avoids joining with the customer table to retrieve this information.
   * **Optimized Reporting and Aggregation:** Redundancy can help precompute aggregated or summary data, speeding up report generation.
2. **Enhanced Data Availability and Reliability:**
   * **Backup and Recovery:** Redundant data can act as a safeguard in case of partial database corruption, improving fault tolerance and recovery options.
   * **Distributed Databases:** In distributed databases, redundancy can ensure data availability in case of network partition or node failure.
3. **Simplified Application Logic:**
   * **Easier Access:** Storing frequently accessed information in multiple places can simplify the development of certain application features, reducing the need for complex queries.
4. **Historical Tracking and Auditing:**
   * **Versioning:** Controlled redundancy can help maintain historical versions of data for auditing or compliance purposes, allowing easy comparison over time.

**Managing Controlled Redundancy in a Relational Model**

While controlled redundancy has its benefits, it must be managed carefully to avoid inconsistencies and maintain data integrity. Here are some strategies for managing controlled redundancy:

1. **Normalization with Denormalization for Performance:**
   * Start by normalizing the database schema to eliminate unnecessary redundancy and ensure data integrity.
   * Then, selectively denormalize tables to introduce controlled redundancy for performance optimization.
   * Example: Normalize customer and order data into separate tables, but add customer name and address directly into the order table for faster query performance.
2. **Use of Database Constraints:**
   * Employ **foreign keys**, **unique constraints**, and **check constraints** to ensure data consistency between redundant data.
   * Example: Use foreign keys to ensure that any redundant customer ID in the order table matches an existing customer ID in the customer table.
3. **Triggers for Automatic Synchronization:**
   * Utilize **triggers** to automatically update or synchronize redundant data whenever a change occurs in the primary source.
   * Example: Create an AFTER INSERT or AFTER UPDATE trigger to update redundant address data in an orders table whenever the customer address changes.
4. **View and Materialized View:**
   * Use **views** or **materialized views** to present redundant data logically without physically storing it, providing redundancy at the query level.
   * Example: A materialized view combining customer and order data for faster reporting while keeping the underlying tables normalized.
5. **Stored Procedures and Functions:**
   * Use **stored procedures** and **functions** to manage redundancy, ensuring that all data modifications occur through controlled, centralized logic.
   * Example: A stored procedure that updates customer information in all relevant tables whenever a customer record is modified.
6. **Data Warehousing and ETL Processes:**
   * In data warehouses, controlled redundancy is used to pre-aggregate data for reporting. **ETL (Extract, Transform, Load)** processes can manage redundancy between operational databases and data warehouses.
   * Example: Regularly extract transactional data, transform it into aggregated summaries, and load it into a reporting database.
7. **Documentation and Policies:**
   * Maintain **clear documentation** and **policies** regarding the purpose and management of redundancy, including how and when redundant data should be updated.
   * Example: Document the rationale for redundant fields in each table and provide guidelines for developers on how to handle updates.

**Conclusion**

Controlled redundancy, when properly managed, can provide significant benefits such as improved performance, data availability, and simplified application logic. Effective management requires careful planning, appropriate use of constraints, triggers, views, and clear documentation to maintain data integrity and consistency in the relational model.

**Consider the relation r(X, Y, Z, B, M, N) and the given functional dependency is**

**X->YZ; Z-> B; Y-> MN**

**decomposed into 2 relations r1(Y, Z, B) and r2(Y, X, M, N).**

**Tell whether this decomposition is lossless or not.**

To determine whether the decomposition of relation r(X,Y,Z,B,M,N) into **r1​(Y,Z,B) and r2​(Y,X,M,N)** is **lossless**, we use the **lossless decomposition condition** from **functional dependency theory**.

**Step 1: Given Information**

**Relation:** r(X,Y,Z,B,M,N)

**Functional Dependencies (FDs):**

1. X→YZ
2. Z→B
3. Y→MN

**Decomposed Relations:**

* r1​(Y,Z,B)
* r2​(Y,X,M,N)

**Step 2: Find the Closure of Common Attributes**

The decomposition is **lossless** if the intersection of the decomposed relations **can determine the entire original relation**.

* **Common attribute in r1​ and r2​**:
  + **Y is the only common attribute**.
* **Closure of Y (Y+)**:
  + From Y→MN, we get M,N.
  + Since Y does not directly determine X, Z, or B, we check if Y+ includes all attributes.

**Y+={Y,M,N}**, which **does not** include all attributes X,Z,B.

Since Y+ does **not** cover all attributes of the original relation, **the decomposition is not lossless**.

**Final Answer:**

The decomposition **r1​(Y,Z,B) and r2​(Y,X,M,N) is not lossless**, because the common attribute Y does not functionally determine all attributes of the original relation.

**(d) Why is Armstrong's axiom is said to be sound and complete?**

**Why is Armstrong’s Axioms Said to Be Sound and Complete?**

**Armstrong’s Axioms** are a set of **inference rules** used to derive all valid functional dependencies (FDs) in a relational database. These axioms are considered **sound and complete** because they satisfy two fundamental properties:

**1. Soundness (Correctness)**

A **set of inference rules** is **sound** if it only derives functional dependencies that are **logically correct** (i.e., they must always hold in the given relation).

* Armstrong’s axioms are sound because they **never infer incorrect dependencies**.
* They only generate FDs that are **logically implied** by the given set of functional dependencies.
* This ensures that no false or invalid dependency is introduced.

**Example of Soundness:**

Given the functional dependencies:

A→B,B→C

Armstrong’s axioms allow us to infer:

A→C

which is **logically correct** by the **transitivity rule**.

Thus, **only true dependencies** are derived, proving **soundness**.

**2. Completeness**

A **set of inference rules** is **complete** if it can derive **all possible** functional dependencies that logically follow from a given set of dependencies.

* Armstrong’s axioms are **complete** because they allow us to infer **all** valid dependencies that exist in the relation.
* No dependency that logically holds in the relation is left out.

**Example of Completeness:**

Given:

A→B,B→C,C→D

We can derive A→D using **transitivity**, ensuring that all possible functional dependencies are covered.

Since Armstrong’s axioms generate **all and only valid dependencies**, they are **both sound and complete**.

**Conclusion**

Armstrong’s axioms are said to be **sound and complete** because:

* **Soundness**: They never derive incorrect or false dependencies.
* **Completeness**: They can derive all possible valid dependencies from a given set of functional dependencies.

Thus, they form a **perfect inference system** for reasoning about functional dependencies in relational databases. 🚀

**How are spurious tuples generated? How this could be prevented?**

**How Are Spurious Tuples Generated?**

Spurious tuples are **incorrect or extra tuples** that appear when performing a **natural join** or **Cartesian product** on decomposed relations. These tuples are not present in the original relation and lead to incorrect query results.

**Causes of Spurious Tuples**

1. **Lossy Decomposition**
   * If a relation is **decomposed improperly** (i.e., not a **lossless decomposition**), joining the decomposed tables may generate tuples that **did not exist in the original relation**.
   * This typically happens when **the common attribute between decomposed relations does not functionally determine the entire relation**.
2. **Incorrect Join Conditions**
   * When performing a **join operation** (especially natural join) on attributes that do not have a **functional dependency relationship**, unrelated tuples may combine, leading to spurious results.
3. **Loss of Functional Dependency Information**
   * If an important **functional dependency is lost** during decomposition, it can result in incorrect tuple combinations when the relations are joined.

**Example of Spurious Tuple Generation**

**Original Relation:**

| **Emp\_ID** | **Emp\_Name** | **Dept\_ID** | **Dept\_Location** |
| --- | --- | --- | --- |
| 101 | John | D1 | NY |
| 102 | Alice | D2 | LA |

**Decomposed Relations (Improper Decomposition):**

**Relation 1: Employee\_Info (Emp\_ID, Emp\_Name, Dept\_ID)**

| **Emp\_ID** | **Emp\_Name** | **Dept\_ID** |
| --- | --- | --- |
| 101 | John | D1 |
| 102 | Alice | D2 |

**Relation 2: Department\_Info (Dept\_ID, Dept\_Location)**

| **Dept\_ID** | **Dept\_Location** |
| --- | --- |
| D1 | NY |
| D3 | SF |

**Incorrect Natural Join Result:**

Joining **Employee\_Info** and **Department\_Info** on **Dept\_ID**:

| **Emp\_ID** | **Emp\_Name** | **Dept\_ID** | **Dept\_Location** |
| --- | --- | --- | --- |
| 101 | John | D1 | NY |
| 102 | Alice | D2 | NULL |
| 102 | Alice | D3 | SF |

Here, the tuple (102, Alice, D3, SF) **did not exist in the original table**, but was created due to an improper decomposition and incorrect join.

**How to Prevent Spurious Tuples?**

1. **Ensure Lossless Decomposition**
   * A decomposition should satisfy the **lossless join property**, meaning that when the relations are joined, no extra tuples are introduced.
   * **Test for lossless decomposition using the Chase Test or the Lossless Join Condition**.
2. **Preserve Functional Dependencies**
   * Ensure that important functional dependencies are not lost when decomposing a relation.
   * The **decomposed schema should preserve functional dependencies** so that original relationships remain intact.
3. **Use Correct Join Conditions**
   * When performing joins, ensure that attributes used in the join condition have a **proper functional dependency relationship**.
   * Avoid joining relations on attributes that do not uniquely identify records.
4. **Use BCNF or 3NF for Proper Decomposition**
   * Decomposing relations into **Boyce-Codd Normal Form (BCNF)** or **Third Normal Form (3NF)** helps prevent redundancy while ensuring that joins do not create spurious tuples.

**Conclusion**

Spurious tuples arise due to **lossy decomposition, incorrect join conditions, or loss of functional dependencies**. To prevent them, always **ensure lossless decomposition, preserve functional dependencies, and use proper join conditions**. Following **BCNF or 3NF decomposition** techniques can also help eliminate this issue. 🚀

**Give example of a relation which is in 2NF but not in 3NE**

**Example of a Relation in 2NF but Not in 3NF**

To understand this, let's recall the definitions of **2NF (Second Normal Form)** and **3NF (Third Normal Form)**:

* **A relation is in 2NF** if:
  1. It is in **1NF** (i.e., it has no repeating groups or multivalued attributes).
  2. It has **no partial dependency**, meaning **no non-prime attribute** should depend on a **proper subset of any candidate key**.
* **A relation is in 3NF** if:
  1. It is in **2NF**.
  2. It has **no transitive dependency**, meaning **every non-prime attribute** must be functionally dependent only on a **super key**, not on another non-prime attribute.

**Example Relation:**

Consider the following relation **Student\_Info**:

R({Student\_ID, Course\_ID, Instructor, Instructor\_Office})

**Functional Dependencies:**

1. **(Student\_ID, Course\_ID) → Instructor** (Each course has a specific instructor)
2. **Instructor → Instructor\_Office** (Each instructor has a specific office)

**Step 1: Identify Candidate Keys**

* The **candidate key** here is **(Student\_ID, Course\_ID)** because:
  + A student can enroll in multiple courses.
  + Each course has one instructor.
  + Instructor details are not unique per student.

**Step 2: Check for 2NF**

* The relation is **in 1NF** (assumption: atomic values).
* There is **no partial dependency**:
  + No non-prime attribute (Instructor, Instructor\_Office) depends only on **part** of the composite key **(Student\_ID, Course\_ID)**.
  + Both depend on the **entire key**, so it satisfies **2NF**. ✅

**Step 3: Check for 3NF**

* The dependency **Instructor → Instructor\_Office** is a **transitive dependency** because:
  + **Instructor** is **not a candidate key**.
  + **Instructor\_Office** depends on **Instructor**, not directly on the primary key **(Student\_ID, Course\_ID)**.
  + This violates **3NF**, making the relation **not in 3NF**. ❌

**Solution: Decomposing into 3NF**

To convert the relation into **3NF**, we decompose it into two relations:

1. **Student\_Course\_Info(Student\_ID, Course\_ID, Instructor)**
2. **Instructor\_Info(Instructor, Instructor\_Office)**

Now:

* **Student\_Course\_Info** is in 3NF because all attributes depend only on the **primary key** (Student\_ID, Course\_ID).
* **Instructor\_Info** is in 3NF because **Instructor** is a key.

**Conclusion**

The original relation **Student\_Info(Student\_ID, Course\_ID, Instructor, Instructor\_Office)** is **in 2NF** (no partial dependencies) **but not in 3NF** (due to the transitive dependency **Instructor → Instructor\_Office**). 🚀

**Why is cautious waiting better than no waiting? Explain**

**Why is Cautious Waiting Better Than No Waiting?**

**Cautious waiting** and **no waiting** are strategies used in **deadlock avoidance** in operating systems, particularly in the context of resource allocation and process synchronization. Here's why **cautious waiting** is generally considered better than **no waiting**:

**1. Cautious Waiting:**

In cautious waiting, processes are allowed to wait for resources, but the system takes measures to **prevent deadlock** and ensure that the waiting does not lead to a cyclic dependency. For example, a process may wait for resources only if it can still make progress later, avoiding situations where it would be indefinitely blocked. This method typically uses algorithms like **Wait-Die** or **Wound-Wait**, which take into account priorities and timestamped requests to ensure no circular waits are formed.

**Benefits of Cautious Waiting:**

* **Prevents Deadlocks:**  
  Cautious waiting ensures that deadlock is avoided by carefully managing how processes wait for resources. It ensures that the system doesn't get stuck in a state where no process can make progress.
* **Efficient Resource Utilization:**  
  Processes can still be granted resources while ensuring that they don't block each other indefinitely, leading to more efficient resource utilization. The system does not unnecessarily waste resources waiting for conditions to be perfect.
* **Fairness and Order:**  
  Algorithms like **Wait-Die** and **Wound-Wait** prioritize older processes, giving them a better chance of acquiring resources. This helps prevent starvation and ensures fairness in resource allocation.
* **Less Resource Wastage:**  
  Processes that are carefully managed through cautious waiting avoid situations where resources are held indefinitely or inefficiently.

**2. No Waiting:**

In the **no waiting** strategy, a process must release any resources it holds before waiting for additional resources. Essentially, a process cannot wait for resources; if it needs a resource and it's not available, it has to give up the ones it already holds.

**Disadvantages of No Waiting:**

* **Higher Overhead:**  
  If processes continually release and reacquire resources as they try to get the necessary resources, this can result in **high overhead**. There’s a lot of unnecessary switching between resources, which can reduce overall system efficiency.
* **Increased Contention:**  
  Since processes must release resources to wait, they might repeatedly release and request resources, leading to higher contention and inefficiency.
* **Resource Fragmentation:**  
  Resources may be fragmented as processes give up resources they don't currently need, leading to an inefficient distribution of resources.
* **Potential Starvation:**  
  Some processes might be perpetually delayed because they have to release resources before they can proceed, potentially leading to **starvation**, where certain processes never get to complete.

**Comparison: Cautious Waiting vs No Waiting**

* **Deadlock Prevention:**  
  Cautious waiting ensures deadlock is avoided while still allowing processes to wait for resources. In contrast, no waiting may reduce the chance of deadlock but often at the cost of efficiency and resource utilization.
* **Efficiency and Fairness:**  
  Cautious waiting is designed to optimize resource allocation while ensuring fairness (by considering timestamps or priorities). No waiting may lead to inefficient use of resources, and processes can be unfairly delayed or starved.
* **Flexibility:**  
  Cautious waiting allows the system to wait when appropriate, which can result in better overall resource management. No waiting, by contrast, forces processes to release resources, which may not always be desirable or efficient.

**Conclusion:**

**Cautious waiting** is generally considered better than **no waiting** because it **prevents deadlock** while maintaining efficient resource utilization and fairness. **No waiting**, while it might reduce the possibility of deadlock in some cases, often leads to inefficiency, contention, and potential starvation, making it less practical for many systems that require optimal resource management.

**What is strict schedule? Discuss about the recovery policy of strict schedule.**

**What is a Strict Schedule?**

In database transaction management, a **strict schedule** refers to a type of transaction schedule (or sequence of operations) where **a transaction is not allowed to release any locks on data items until it has committed or aborted**. This means that:

* A **transaction holds locks on all the data items it accesses until it is finished** (committed or aborted).
* Once a transaction commits, all its locks are released. If it aborts, the changes made by that transaction are rolled back, and any locks it held are also released.

This behavior ensures that no other transaction can access or modify the data that is being processed by an uncommitted transaction, thereby avoiding certain concurrency issues.

**Strict Schedule Characteristics:**

* **No read/write operations of uncommitted data**: In a strict schedule, if a transaction has modified a data item, no other transaction can access it until the transaction commits.
* **Prevents Cascading Aborts**: Since transactions must be fully committed before releasing any locks, it ensures that if a transaction is rolled back, it does not affect other transactions that have read its uncommitted data (no cascading aborts).

**Recovery Policy of Strict Schedules**

The **recovery policy** associated with strict schedules focuses on **ensuring that the database maintains consistency** even in the event of a failure, by making use of the fact that no transaction will release its locks until it commits. The policy is typically **log-based**, and recovery is achieved through mechanisms like **undo** and **redo** operations. Here's how recovery works in a strict schedule:

**1. No Cascading Rollbacks**

Since **no transaction releases its locks until it commits**, transactions are less likely to affect each other in the event of an abort. Specifically:

* If **Transaction T1** reads a value written by **Transaction T2**, and T2 has not committed yet, T1 would not be able to access that value until T2 commits.
* If **T2 aborts**, T1 will not be affected because T1 has not seen any uncommitted changes. This eliminates the risk of **cascading rollbacks**, where a failure in one transaction could cause a chain reaction of rollbacks in other transactions.

**2. Rollback (Undo)**

In case of a **transaction failure** (e.g., system crash, manual abort), the **undo** operation is used to restore the database to its state before the transaction started:

* Since transactions in a strict schedule do not release their locks until they commit, all modifications made by a transaction are **guaranteed to be either fully applied or not applied at all**.
* The system can **rollback** the transaction by undoing the changes made by that transaction (using the transaction's log).
* This ensures that the database remains in a consistent state after a failure, and the effects of an incomplete transaction are removed.

**3. Log-Based Recovery (Write-Ahead Logging - WAL)**

Strict schedules typically rely on **Write-Ahead Logging (WAL)** to ensure that **log records are written to disk before actual data updates**. This helps in recovering from failures:

* **Before any data item is modified**, the transaction’s log record is written to disk.
* If the system crashes, the log is used to **reapply (redo)** changes from committed transactions or to **undo** changes from aborted transactions.
* The log provides a persistent record of all operations that occurred during the transaction, which allows for complete recovery of the database state after a failure.

**4. No Lost Updates or Dirty Reads**

In a strict schedule:

* **Lost updates** are prevented because a transaction cannot overwrite uncommitted changes from another transaction.
* **Dirty reads** are prevented because transactions must wait for the commit of another transaction before reading its data.

This makes **strict schedules** useful for ensuring that a system remains in a consistent state even during failures, as all changes made by incomplete transactions are not visible to others.

**Advantages of Strict Schedules in Recovery:**

* **Avoidance of Cascading Rollbacks**: As no uncommitted data is visible to other transactions, rolling back one transaction does not affect others.
* **Easier Recovery**: Since all data modifications made by a transaction are committed only after it has completed, recovery is straightforward.
* **Consistency**: The system ensures consistency by preventing dirty reads and lost updates.

**Disadvantages of Strict Schedules:**

* **Reduced Concurrency**: Strict schedules can limit concurrency because transactions have to hold locks until they commit, which can cause delays for other transactions trying to access the same data.
* **Lock Contention**: Since locks are held longer, other transactions may be blocked, which could result in **deadlocks** or **long wait times** for resources.

**Conclusion**

A **strict schedule** is a transaction scheduling method that ensures transactions hold their locks until they are either committed or aborted, preventing uncommitted changes from affecting other transactions. The recovery policy of strict schedules focuses on undoing or redoing transactions to maintain consistency and avoid cascading rollbacks, ensuring database integrity even in the case of system failures. While it provides strong consistency guarantees, it may lead to reduced concurrency and potential lock contention.

**3. (a) Consider the following schedule written in log**

**<T0, Start>**

**<To, A, 96>**

**<To, B, 40>**

**< To, Commit>**

**<T1, Start>**

**<T1, A, 32>**

**<T1, B, 201>**

**What actions will be taken for recovery if (i) Deferred Database Modification, (ii) Immediate**

**Database Modification applied?**

**(b) What is the benefit of using checkpoint?**

**(c) Write the algorithm to check whether a schedule is conflict serializable or not.**

**(d Give an example of a schedule which is recoverable but has lost update problem**

**(a) Recovery Actions for Different Database Modification Approaches**

Given the schedule:

<T0, Start>

<T0, A, 96>

<T0, B, 40>

<T0, Commit>

<T1, Start>

<T1, A, 32>

<T1, B, 201>

**(i) Deferred Database Modification**

* In **Deferred Database Modification**, changes are **not** written to the database immediately. Instead, updates are kept in a log and applied **only after** a transaction commits.
* **Recovery actions:**
  + If the system crashes before **T0 commits**, then **T0's changes are ignored** since they were not applied to the database yet.
  + If the system crashes after **T0 commits but before T1 commits**, **T0's changes are redone** (because committed transactions must persist).
  + If the system crashes before **T1 commits**, **T1's updates are ignored**.

**(ii) Immediate Database Modification**

* In **Immediate Database Modification**, changes are written to the database **before** a transaction commits.
* **Recovery actions:**
  + If the system crashes before **T0 commits**, the updates to **A and B by T0 must be undone** (because uncommitted changes should not persist).
  + If the system crashes after **T0 commits but before T1 commits**, **T0's updates stay, but T1’s updates are undone**.
  + If the system crashes after **T1 updates but before commit**, **T1's changes to A and B must be undone**.

**(b) Benefit of Using Checkpoints**

A **checkpoint** in a database system is a mechanism to reduce the recovery time after a crash. The main benefits are:

1. **Faster Recovery** – The system does not have to scan the entire log from the beginning; it only needs to process logs after the last checkpoint.
2. **Minimizes Work** – It writes the latest committed state to disk, reducing the number of redo and undo operations.
3. **Reduces Log Size** – Older log entries can be safely discarded after a checkpoint.
4. **Ensures Data Consistency** – A checkpoint helps in ensuring that all previous transactions are properly stored before processing new ones.

**(c) Algorithm to Check Conflict Serializability**

To check if a schedule is **conflict serializable**, we use a **precedence (or dependency) graph**:

**Algorithm**

1. **Create a node** for each transaction in the schedule.
2. **Identify conflicting operations:**
   * Conflicts occur when two transactions access the **same data item**, and at least one of them is a **write**.
   * Types of conflicts:
     1. Read-Write (RW)
     2. Write-Read (WR)
     3. Write-Write (WW)
3. **Draw edges in the graph:**
   * If **T1** executes a conflicting operation before **T2**, then add a directed edge from **T1 → T2**.
4. **Check for cycles:**
   * If the graph has a **cycle**, the schedule is **not conflict serializable**.
   * If the graph is **acyclic**, the schedule **is conflict serializable**.

**(d) Example of a Recoverable Schedule with a Lost Update Problem**

A schedule is **recoverable** if no transaction commits before all transactions it depends on commit. A **lost update** occurs when an uncommitted update is overwritten.

**Example Schedule**

T1: R(A) ----> W(A) ----> Commit

T2: R(A) ----> W(A) ----> Commit

Assume the initial value of **A = 100**.

* **T1 reads A (100)**.
* **T2 reads A (100)**.
* **T1 writes A = 120**.
* **T2 writes A = 110** (Overwrites T1's update).
* **T1 commits**.
* **T2 commits**.

**Why is this Recoverable?**

* **T2 commits after T1**, so T2 does not commit before reading an uncommitted value.

**Why does it have a Lost Update Problem?**

* **T1 updates A to 120**, but **T2 overwrites A with 110**, **losing T1's update**.

This problem can be prevented using **locking mechanisms** or **timestamp ordering**.

**4. (a) Consider a fact table consists of 4 Dimensions namely Supplier, Location, Time and Product. The Dimensions Supplier , Location, Time and Product are consisting of 4, 5, 3, 3 values respectively. Assume the values measures (as required). Draw the MOLAP structure which represents this 4 dimensional cube.**

**(b) Represent a starnet model, where the following concept hierarchies are present in the dimensions :**

**(i) City ->State -> Country**

**(ii) Day-> Month ->Quarter -> Year**

**(iii) Item\_name -> Item\_type -> Item brand**

**(a) MOLAP Structure (4-Dimensional Cube)**

MOLAP (Multidimensional OLAP) stores data in a pre-computed multidimensional cube for fast retrieval. The given fact table consists of **four dimensions**:

* **Supplier** (4 values)
* **Location** (5 values)
* **Time** (3 values)
* **Product** (3 values)

The total number of possible data points in the cube:

4×5×3×3=180

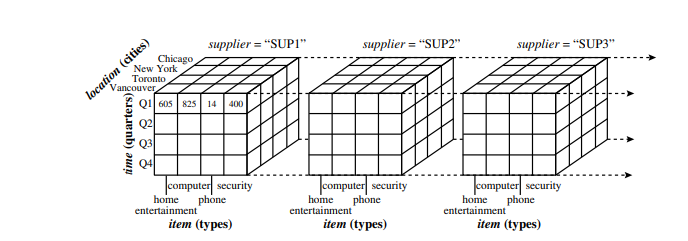
Each cell in this **4D cube** stores an **aggregated measure** (e.g., sales, revenue, quantity sold).

**MOLAP Cube Representation**

A **4D cube** is difficult to visualize directly, but we can break it down into **3D slices**:

1. **3D Representation (Supplier × Location × Product)**
   * Each slice represents a **specific Time value**.
   * Example: If **Time = Q1**, we have a **3D cube** with Supplier, Location, and Product.
2. **Multiple 3D Slices (Time Dimension)**
   * If we add the **Time** dimension, the cube stacks these 3D slices.

Graphically, the MOLAP cube looks like multiple **3D data cubes** stacked along the **Time** axis.



**(b) Starnet Model Representation**

The **starnet model** represents multidimensional data as a graph with a **central fact node** connected to **dimensional hierarchies**.

**Given Concept Hierarchies:**

1. **Location Hierarchy**:
   * City → State → Country
2. **Time Hierarchy**:
   * Day → Month → Quarter → Year
3. **Product Hierarchy**:
   * Item Name → Item Type → Item Brand

**Graph Representation**

A **starnet model** is represented as a **star graph**, where:

* The **Fact Table (Sales, Revenue, etc.)** is in the center.
* Dimensions expand outward in hierarchical levels.

**Starnet Model Structure**

┌───────────┐

│ Country │

└────▲──────┘

│

┌──────────┐

│ State │

└────▲─────┘

│

┌──────────┐

│ City │

└────▲─────┘

│

┌─────────────────────┼───────────────────────┐

│ │ │

┌───▼───┐ ┌───▼───┐ ┌───▼───┐

│Supplier│ │ Product│ │ Time │

└────┬───┘ └────┬───┘ └───┬───┘

│ │ │

│ ┌─────▼─────┐ ┌──▼──┐

│ │ Item Brand│ │ Year│

│ └─────▲─────┘ └──▲──┘

│ │ │

┌────▼────┐ ┌────▼────┐ ┌────▼────┐

│ Fact │ │ Item Type│ │ Quarter │

│ Table │ └────▲─────┘ └───▲────┘

│ (Sales, │ │ │

│ Revenue)│ ┌─────▼─────┐ ┌────▼─────┐

└─────────┘ │ Item Name │ │ Month │

└───────────┘ └────▲─────┘

│

┌────▼───┐

│ Day │

└────────┘

**Explanation of Starnet Model**

* The **fact table** is central, storing measures like revenue and sales.
* **Location hierarchy** expands from **City → State → Country**.
* **Time hierarchy** expands from **Day → Month → Quarter → Year**.
* **Product hierarchy** expands from **Item Name → Item Type → Item Brand**.

This structure helps in **drill-down and roll-up operations** in OLAP, allowing users to analyze data at different levels of granularity.

**Conclusion**

* The **MOLAP cube** organizes data in a 4D array for fast retrieval.
* The **starnet model** provides a flexible hierarchical view of dimensions.

**5. (a) Explain slicing and drill-across operation with an example**

**(b) Compute the number of cuboids for a OLAP System which is represented in terms of lattice of cuboids where Dimension S has 2 abstractions, Q has 3 abstractions and dimensions P and R have**

**no abstractions.**

**(c) Explain the functionality of Virtual data Warehouse with example.**

**(a) Slicing and Drill-Across in OLAP**

In **OLAP (Online Analytical Processing)**, slicing and drill-across operations help in analyzing data effectively.

**1. Slicing**

* **Definition:** Slicing is the process of extracting a **subcube** from a larger multidimensional cube by fixing a value for one dimension.
* **Example:**  
  Suppose we have a **Sales data cube** with dimensions **Time, Product, and Location**.
  + If we **slice the cube** for Time = Q1, we get a **2D slice (Product × Location)** showing sales data only for **Quarter 1**.

**Original Cube (3D)**:

Time × Product × Location

**Sliced Cube (2D View for Q1)**:

Product × Location

**2. Drill-Across**

* **Definition:** Drill-Across is used to combine and compare data from **multiple fact tables** that share some **common dimensions**.
* **Example:**  
  Consider two fact tables:
  1. **Sales\_Fact** (Product, Location, Time, Sales)
  2. **Profit\_Fact** (Product, Location, Time, Profit)
  3. Using **drill-across**, we can **join both tables on common dimensions (Product, Location, Time)** to **compare Sales vs. Profit**.

**(b) Number of Cuboids in a Lattice**

The total number of cuboids in an **OLAP lattice** is computed as:

Total Cuboids=(Number of Abstractions in S+1)×(Number of Abstractions in Q+1)×(Number of Abstractions in P+1)×(Number of Abstractions in R+1)

Given:

* **S has 2 abstractions** → (2 + 1 = 3)
* **Q has 3 abstractions** → (3 + 1 = 4)
* **P has no abstractions** → (0 + 1 = 1)
* **R has no abstractions** → (0 + 1 = 1)

Total Cuboids=3×4×1×1=12

Thus, the OLAP system contains **12 cuboids** in the lattice.

**(c) Functionality of Virtual Data Warehouse**

**Definition**

A **Virtual Data Warehouse (VDW)** is a **logical, real-time** data warehouse that does **not physically store** data but provides a **virtual view** of data from multiple databases.

**Functionality**

1. **Data is not stored in a separate warehouse** but retrieved on-demand from different sources.
2. **Uses middleware or a data federation approach** to integrate data.
3. **Query execution is real-time**, fetching data from operational databases instead of storing it separately.

**Example**

Consider a retail company that has:

* **Sales database**
* **Inventory database**
* **Customer database**

Instead of physically merging them, a **Virtual Data Warehouse** creates a **logical layer** allowing queries like:

SELECT Sales.Amount, Inventory.Stock, Customer.Region

FROM Sales\_DB, Inventory\_DB, Customer\_DB

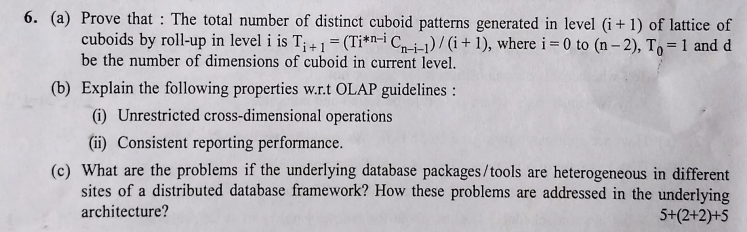
WHERE Sales.Product\_ID = Inventory.Product\_ID

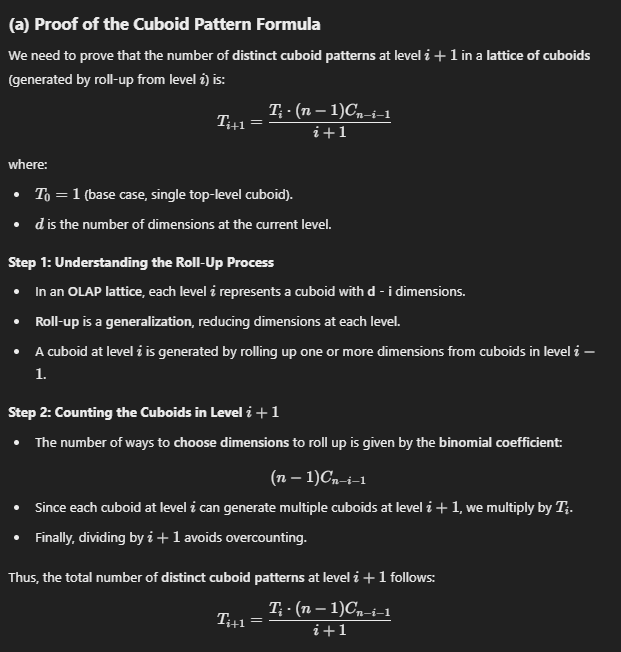
AND Sales.Customer\_ID = Customer.Customer\_ID;

Here, data is **fetched in real-time** without being **stored separately**.

**Advantages**

* **Lower storage cost** since data is not duplicated.
* **Real-time data access**.
* **Flexibility** to integrate multiple data sources dynamically.



****

**(b) OLAP Guidelines**

OLAP systems follow several **key properties** to ensure efficient multi-dimensional analysis.

**(i) Unrestricted Cross-Dimensional Operations**

* **Definition:**
  + Users should be able to **freely analyze** data across **any combination of dimensions** without constraints.
  + OLAP should support **slice, dice, drill-down, roll-up, and pivot** operations across multiple dimensions.
* **Example:**
  + A business analyst might want to analyze **Sales Revenue** based on **Region** and **Time**.
  + Later, they might want to **add a new dimension (e.g., Product Category)** without restrictions.
  + **Unrestricted** means no predefined paths—users should dynamically combine any dimensions.

**(ii) Consistent Reporting Performance**

* **Definition:**
  + The **query response time** should remain **consistent** regardless of the **level of aggregation**.
  + Whether querying **highly detailed data** (e.g., daily sales) or **aggregated data** (e.g., yearly revenue), performance should be stable.
* **Challenges & Solutions:**
  + Large OLAP queries can be **slow** at high granularity.
  + **Solution:** **Pre-aggregation** and **caching** improve performance.
  + MOLAP uses **precomputed cubes** to ensure fast response times.

**(c) Problems & Solutions in Heterogeneous Distributed Databases**

In a **distributed database framework**, different sites may use **heterogeneous database tools** (e.g., MySQL, PostgreSQL, Oracle, SQL Server). This causes several **problems**:

**Problems Due to Heterogeneity**

1. **Data Model Differences**
   * Some databases use **Relational Models (SQL-based)**, while others use **NoSQL** or **Hierarchical Models**.
   * **Solution:** Use a **middleware** layer to standardize queries.
2. **Query Language Incompatibility**
   * Different databases use different SQL dialects.
   * **Solution:** Use **query translation layers** (e.g., an API that converts all queries to a common SQL format).
3. **Data Format Differences**
   * Some databases store dates as **"YYYY-MM-DD"**, while others use **"DD-MM-YYYY"**.
   * **Solution:** Define **global schema mapping** to ensure consistency.
4. **Transaction Management Issues**
   * Different databases follow different **ACID properties**.
   * **Solution:** Use **two-phase commit (2PC)** to ensure consistency.
5. **Security & Authentication Issues**
   * Different sites may use different authentication mechanisms.
   * **Solution:** Use **single sign-on (SSO)** or **federated identity management**.

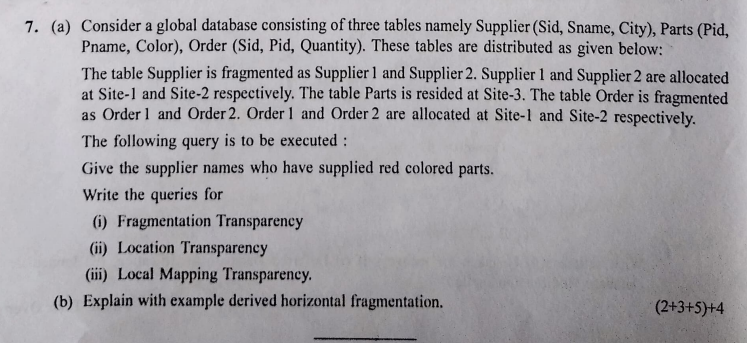
**Architectural Solutions**

To address these challenges, **middleware-based architectures** are commonly used:

* **Federated Database System (FDS):** A global schema integrates different databases.
* **ETL (Extract, Transform, Load) Pipelines:** Convert data into a common format.
* **Data Warehouses:** Consolidate data from different sources.

**Conclusion**

* **(a)** We proved the cuboid formula using combinatorics.
* **(b)** We explained two key OLAP properties (**cross-dimensional operations** and **consistent performance**).
* **(c)** We discussed challenges in **heterogeneous distributed databases** and architectural solutions.

****

**(a) Query Transparency in Distributed Databases**

We are given a **global database** consisting of three tables:

1. **Supplier**: (Sid, Sname, City)
   * **Fragmented into:** Supplier1 at **Site-1**, Supplier2 at **Site-2**.
2. **Parts**: (Pid, Pname, Color)
   * **Stored entirely at:** **Site-3**.
3. **Order**: (Sid, Pid, Quantity)
   * **Fragmented into:** Order1 at **Site-1**, Order2 at **Site-2**.

**Query to Execute:**

*Find supplier names who have supplied red-colored parts.*

The equivalent **global SQL query** would be:

SELECT DISTINCT S.Sname

FROM Supplier S, Parts P, Order O

WHERE S.Sid = O.Sid

AND O.Pid = P.Pid

AND P.Color = 'Red';

**(i) Fragmentation Transparency**

* **Definition:** The user should be **unaware** that tables are fragmented.
* **Solution:** Write a global query that applies to **all fragments**.

Query considering **fragmented tables**:

SELECT DISTINCT S.Sname

FROM (SELECT \* FROM Supplier1 UNION ALL SELECT \* FROM Supplier2) S,

Parts P,

(SELECT \* FROM Order1 UNION ALL SELECT \* FROM Order2) O

WHERE S.Sid = O.Sid

AND O.Pid = P.Pid

AND P.Color = 'Red';

* The UNION ALL combines the fragments of Supplier and Order, making fragmentation **transparent**.

**(ii) Location Transparency**

* **Definition:** The user should not need to know **where tables are stored**.
* **Solution:** A **Distributed Database System (DDBMS)** resolves the table locations internally.

The query remains the **same as the global SQL query**:

SELECT DISTINCT S.Sname

FROM Supplier S, Parts P, Order O

WHERE S.Sid = O.Sid

AND O.Pid = P.Pid

AND P.Color = 'Red';

* The **DDBMS automatically** retrieves data from **Site-1, Site-2, and Site-3**.

**(iii) Local Mapping Transparency**

* **Definition:** The user should not need to worry about **how fragments are mapped to physical storage**.
* **Solution:** The **query processor** ensures the correct retrieval of records.

At each site, queries operate on local fragments:

* **At Site-1** (operates only on Supplier1 and Order1):

SELECT DISTINCT S.Sname

FROM Supplier1 S, Parts P, Order1 O

WHERE S.Sid = O.Sid

AND O.Pid = P.Pid

AND P.Color = 'Red';

* **At Site-2** (operates only on Supplier2 and Order2):

SELECT DISTINCT S.Sname

FROM Supplier2 S, Parts P, Order2 O

WHERE S.Sid = O.Sid

AND O.Pid = P.Pid

AND P.Color = 'Red';

* The **final result** is the **union of results** from both sites.

**(b) Derived Horizontal Fragmentation**

**Definition**

* **Derived Horizontal Fragmentation** occurs when a table is fragmented based on the **fragments of another table** using **foreign key relationships**.

**Example**

**Global Tables:**

1. **Supplier (Sid, Sname, City)**
2. **Parts (Pid, Pname, Color)**
3. **Order (Sid, Pid, Quantity)**
   * Here, Order.Sid is a **foreign key** referencing Supplier.Sid.

**Derived Fragmentation**

If **Supplier** is **horizontally fragmented** based on **City**:

* **Supplier1** (Suppliers in 'Kolkata') → Site-1
* **Supplier2** (Suppliers in 'Mumbai') → Site-2

Then, **Order** is **derived fragmented** based on the **Supplier** fragments:

* **Order1** (Only orders from Supplier1) → Site-1
* **Order2** (Only orders from Supplier2) → Site-2

Thus, **Order** is fragmented based on the fragmentation of **Supplier**.

**Advantages**

* **Improves query performance** by storing related data together.
* **Minimizes network overhead**, as joins occur locally.

**Conclusion**

* **(a) We wrote queries for different types of transparency in a distributed database.**
* **(b) We explained derived horizontal fragmentation with an example.**

**2022**

**1. (a) Consider the following Relation :**

**r(R)= {A, B, C, D, E, F, G, H, I}**

**The set of Functional dependency is given below :**

**F= {A->B, C-> DE,F->G, B-> GH, AF -> C, E-> I}**

**Determine the current normal form of the given relation. Decompose it upto 3NF.**

**(b) Give an example where a relation is in 3NF but not in BCNF.**

**(c) Give an example where a relation is in 3NF as well as in BCNF.**

**(d) What do you understand by attribute preservation? Explain.**

To solve this, we need to:

1. **Identify the current normal form** of the relation.
2. **Decompose the relation step by step** until it reaches the Third Normal Form (3NF).

**Step 1: Determine the current normal form**

We have the relation r(R)={A,B,C,D,E,F,G,H,I} and the set of functional dependencies F as follows:  
F={A→B,C→DE,F→G,B→GH,AF→C,E→I}

We will check for violations of **1NF**, **2NF**, and **3NF**.

* **1NF (First Normal Form)**: The relation must have atomic values only.  
  The relation is assumed to be in 1NF since no multivalued attributes are mentioned.
* **2NF (Second Normal Form)**: There should be no partial dependency on a candidate key.  
  We need to find the candidate key to verify this.

**Step 2: Find the candidate key(s)**

A candidate key is a minimal set of attributes that can determine all other attributes in the relation.

We analyze the functional dependencies:

* A→B: A can determine B
* C→DE: C can determine D and E
* F→G: F can determine G
* B→GH: B can determine G and HH
* AF→C: AF can determine C
* E→I: E can determine I

**Step-by-step derivation of the candidate key:**

* Attributes covered by dependencies: A→B,B→GH implies A→BGH  
  AF→C,C→DE,E→I implies AF→CDEI

Thus, AF can determine all the attributes (A,B,C,D,E,F,G,H,I). Therefore, AF is the **candidate key**.

**Step 3: Check for 2NF violations**

A relation is in 2NF if it is in 1NF and there are no partial dependencies (i.e., no non-prime attribute should be dependent on part of a candidate key).

**Candidate key:** AF  
**Prime attributes:** A and F  
**Non-prime attributes:** B,C,D,E,G,H,I

Check the functional dependencies:

* A→B: Partial dependency (violates 2NF)
* F→G: Partial dependency (violates 2NF)

**Step 4: Decompose into 2NF**

We will decompose the relation to remove partial dependencies.

**Decomposition:**

1. R1​(A,B) with A→B
2. R2​(F,G) with F→G
3. R3​(A,F,C,D,E,H,I) with the remaining attributes

**Step 5: Check for 3NF violations**

A relation is in 3NF if it is in 2NF and there are no transitive dependencies (i.e., a non-prime attribute should not be transitively dependent on the candidate key).

In R3​, we have:

* C→DE: Transitive dependency (violates 3NF)
* E→I: Transitive dependency (violates 3NF)

**Step 6: Decompose into 3NF**

We decompose further to remove transitive dependencies.

**Final decomposition:**

1. R1​(A,B) with A→B
2. R2​(F,G) with F→G
3. R3​(B,G,H) with B→GH
4. R4​(A,F,C) with AF→C
5. R5​(C,D,E) with C→DE
6. R6​(E,I) with E→I

**Conclusion:**

The given relation r(R) was initially in **1NF**, and after decomposition, we achieved **3NF** with the relations R1​,R2​,R3​,R4​,R5​,R6​.

**(b) Example of a relation in 3NF but not in BCNF**

A relation is in **3NF but not in BCNF** when it has a **functional dependency involving a non-candidate key that determines part of a candidate key**.

**Example:**  
Consider the relation R(A,B,C) with functional dependencies:

1. A→B
2. B→C

**Candidate key:** A (since A→BC)

**Explanation:**

* The relation is in **3NF** because:
  + A→B (B is a non-prime attribute, but A is a candidate key, so this satisfies 3NF).
  + B→C (C is a non-prime attribute, but B is a superkey for the subrelation (B,C), so it also satisfies 3NF).
* The relation is **not in BCNF** because:
  + B→C violates BCNF since B is not a candidate key, but it determines another attribute C.

**To convert to BCNF**, we decompose the relation into:

1. R1​(B,C) with B→C
2. R2​(A,B)

**(c) Example of a relation in 3NF and BCNF**

A relation is in **3NF and BCNF** when every determinant is a candidate key.

**Example:**  
Consider the relation R(A,B) with the functional dependency:  
A→B

**Candidate key:** A

**Explanation:**

* The relation is in **3NF** because A is a candidate key and it determines B.
* The relation is also in **BCNF** because the only functional dependency is A→B, and A is a candidate key.

**(d) Attribute Preservation**

**Attribute preservation** is an important property of decomposition in relational databases. When we decompose a relation into two or more sub-relations, **attribute preservation ensures that all original attributes of the relation are present in the decomposed relations**.

In other words, the union of attributes in the decomposed relations must be equal to the set of attributes in the original relation.

**Why is it important?**

* Without attribute preservation, we might lose some attributes during decomposition.
* This could lead to incorrect or incomplete query results after decomposition.

**Example:**  
Consider the relation R(A,B,C) with functional dependency A→B.

If we decompose R into two relations:

1. R1​(A,B)
2. R2​(B,C)

Here, the attribute A is lost in R2​, so the decomposition does **not preserve attributes**.

To ensure attribute preservation, a valid decomposition would be:

1. R1​(A,B)
2. R2​(A,C)

Now, the set of attributes (A,B,C) is fully preserved.

**2. (a) Write an algorithm to find out the F+ for a given relation r, whose functional dependency is F.**

**(b) Consider the following relational schema**

**r1(A, B, C, D, E, F, G)**

**The following functional dependency is hold in the relation**

**A ->B; B -> C; C -> DE; F -> G**

**Maintaining the functional dependency the relation r1 is broken as follows**

**r2(A,B); r3(B, C); r4(C, D, E); r5(F, G)**

**Verify whether this decomposition is lossless or not.**

**(c) With example explain (i) full functional dependency, (ii) partial functional dependency.**

**2(a) Algorithm to Find F⁺ (Closure of Functional Dependencies)**

The closure F+ of a set of functional dependencies F is the set of all functional dependencies that can be **logically inferred** from F. To compute F+, follow these steps:

**Algorithm to Compute F⁺:**

**Input:**  
A set of attributes R and a set of functional dependencies F.

**Output:**  
The closure F+, which contains all functional dependencies logically implied by F.

**Steps:**

1. **Initialize** F+ as F (i.e., start with the given set of functional dependencies).
2. **Repeat until no new functional dependency can be added:**  
   a. For each functional dependency X→Y in F, apply Armstrong’s axioms:
   * **Reflexivity:** If Y⊆X, then X→Y holds.
   * **Augmentation:** If X→Y, then XZ→YZ for any Z.
   * **Transitivity:** If X→Y and Y→Z, then X→Z.  
     b. Add all newly derived dependencies to F+.
3. **Stop** when no more functional dependencies can be inferred.

**2(b) Lossless Decomposition Verification**

We need to verify if the decomposition of relation r1(A,B,C,D,E,F,G) into relations r2,r3,r4,r5 is **lossless**.

**Functional dependencies (FDs):**

1. A→B
2. B→C
3. C→DE
4. F→G

**Decomposed relations:**

* r2(A,B)
* r3(B,C)
* r4(C,D,E)
* r5(F,G)

**Steps to Check for Lossless Decomposition:**

A decomposition is **lossless** if the common attributes between the sub-relations act as a **superkey** in at least one of the sub-relations. We will check for this property.

1. **Common attributes:**
   * r2(A,B) and r3(B,C) have B in common.
   * r3(B,C) and r4(C,D,E) have C in common.
   * r5(F,G) has no common attributes with other relations.
2. **Superkey Verification:**
   * B is a superkey for r3(B,C) because B→C in r3.
   * C is a superkey for r4(C,D,E) because C→DE in r4.
   * F is a superkey for r5(F,G) because F→G.

Thus, the decomposition is **lossless** because every shared attribute between sub-relations acts as a superkey in at least one of the relations.

**2(c) Explanation with Examples:**

**(i) Full Functional Dependency**

A functional dependency X→Y is a **full functional dependency** if all attributes in X are necessary to determine Y. Removing any attribute from X will break the dependency.

**Example:**  
Relation: R(A,B,C)  
Functional dependency: AB→C  
Here, AB→C is a **full functional dependency** because neither A→C nor B→C holds individually.

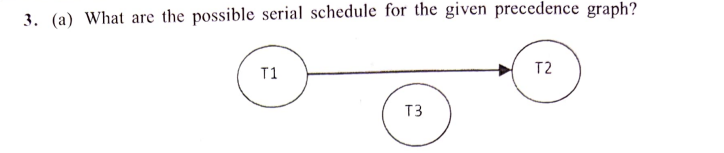
**(ii) Partial Functional Dependency**

A functional dependency X→Y is a **partial functional dependency** if some attribute in X can be removed without breaking the dependency.

**Example:**  
Relation: R(A,B,C)  
Functional dependencies:

* A→C
* AB→C

Here, AB→C is a **partial dependency** because A→C already holds, so B is not necessary.

****

**3(a) Precedence Graph Explanation**

In a **precedence graph (or serializability graph)**:

* **Nodes** represent transactions T1​,T2​,T3​.
* **Edges** represent a precedence (i.e., a dependency) where one transaction must execute before another due to conflicting operations (read/write).

**Given Precedence Graph:**

* T1​→T2​ (There is a dependency from T1​ to T2​)
* T3​ is **not connected** to T1​ or T2​. This means T3​ can execute at **any position** without violating serializability.

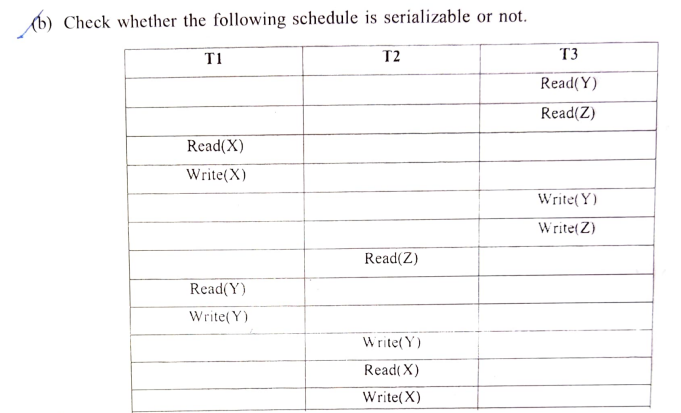
**Possible Serial Schedules**

In a serial schedule, transactions must execute in a complete sequence without interleaving operations.

1. **T1​→T2​ order must be maintained** because of the directed edge.
2. T3​ can appear **before, between, or after T1​ and T2​**.

Thus, the possible serial schedules are:

1. T3​,T1​,T2​
2. T1​,T3​,T2​
3. T1​,T2​,T3​

****

**To check if the given schedule is serializable, we need to:**

1. **Identify the transactions and operations.**
2. **Construct a precedence graph** to detect conflicts.
3. **Check for cycles**—if there is no cycle, the schedule is conflict-serializable.

**Step 1: Transactions and Operations**

Given schedule:

| **Operation** | **Transaction** |
| --- | --- |
| T3: Read(y) | T3 |
| T3: Read(z) | T3 |
| T1: Read(x) | T1 |
| T1: Write(y) | T1 |
| T3: Write(y) | T3 |
| T3: Write(z) | T3 |
| T2: Read(z) | T2 |
| T1: Read(y) | T1 |
| T1: Write(y) | T1 |
| T2: Write(y) | T2 |
| T2: Read(x) | T2 |
| T2: Write(x) | T2 |

**Step 2: Identify Conflicts**

Conflicting operations occur when:

1. **One transaction writes to a variable and another transaction reads/writes the same variable.**
2. The operations happen on the same data item but in different transactions.

**Conflicts:**

1. T1 (Write(y)) → T3 (Write(y)) → T2 (Write(y))
2. T1 (Write(y)) → T2 (Read(y))
3. T1 (Read(x)) → T2 (Write(x))
4. T3 (Read(z)) → T2 (Read(z)) — not a conflict since both are reads.

**Step 3: Build the Precedence Graph**

* **T1 → T3:** T1 writes to y, and T3 subsequently writes to y.
* **T3 → T2:** T3 writes to y, and T2 subsequently writes to y.
* **T1 → T2:** T1 reads and writes to x, and T2 writes to x.

**Precedence graph:**

* **T1 → T3 → T2**

**Step 4: Check for Cycles**

Since the precedence graph is **acyclic**, the schedule is **conflict-serializable**.

**Conclusion:**

The given schedule is **serializable**, and it is equivalent to the serial schedule **T1→T3→T2**.

**c. Write the algorithm pf Read\_lock(X) and Unlock(x) for exclusive lock**

**Algorithm for Read\_Lock(X) and Unlock(X)**

In a locking mechanism, **Read\_Lock(X)** allows **shared access** to the data item X, while **Unlock(X)** releases the lock.

However, **exclusive locks** are different from shared locks. An **exclusive lock (X-Lock)** ensures that **only one transaction** can access the data item for both reading and writing at a time. Below are the algorithms for exclusive lock operations.

**1. Algorithm for Read\_Lock(X) (Exclusive Lock)**

Input: Data item X

1. If X is already locked in exclusive mode by another transaction, then:

a. Wait until the lock is released (block the transaction).

2. If X is not locked, then:

a. Lock X in exclusive mode.

3. Mark the transaction as holding the lock on X for reading.

**Explanation:**

* If X is not locked, the transaction obtains an exclusive lock and reads the value of X.
* If X is locked by another transaction, the transaction must wait.

**2. Algorithm for Unlock(X)**

Input: Data item X

1. Check if the transaction holds a lock on X.

a. If yes, release the lock on X.

b. Remove the transaction from the list of holders of X.

2. If there are other waiting transactions for X, wake them up.

**Explanation:**

* Once a transaction has completed its operations on X, it releases the exclusive lock, allowing other transactions to access X.
* If there are waiting transactions, they are notified and allowed to proceed.

**Example of Exclusive Lock**

Consider two transactions T1 and T2 that want to access data item X:

1. **T1 requests an exclusive lock (Read\_Lock(X)):** It is granted the lock and reads the value of X.
2. **T2 requests an exclusive lock (Read\_Lock(X)):** It must wait because T1 holds the exclusive lock.
3. **T1 executes Unlock(X):** The lock is released, and T2 can now obtain the exclusive lock.

**4. a. the following schedule is written in log :**

**<T0, start>**

**<T0, x, 1002>**

**<T0, y , 801>**

**<T0, commit>**

**<T1, start>**

**<T1,x,687>**

**<T1, y, 300>**

**What actions will be taken if (i) Deferred database modification and (ii) Immediate database modification?**

**4(a) Explanation of the Schedule and Recovery Actions**

The given schedule in the log:

| **Log Entry** | **Description** |
| --- | --- |
| <T0, start> | Transaction T0 starts |
| <T0, x, 1002> | T0 updates x to 1002 |
| <T0, y, 801> | T0 updates y to 801 |
| <T0, commit> | T0 commits its changes |
| <T1, start> | Transaction T1 starts |
| <T1, x, 687> | T1 updates x to 687 |
| <T1, y, 300> | T1 updates y to 300 |

**(i) Deferred Database Modification (Deferred Update)**

* **Updates are not applied to the database immediately.**
* Changes are kept in a temporary log or buffer and applied to the database only when the transaction **commits**.
* If a transaction fails before committing, **no changes** are applied to the database.

**Recovery Actions:**

1. **For committed transaction T0:** Apply changes to the database for x=1002 and y=801.
2. **For uncommitted transaction T1:** No changes are applied because T1 has not committed yet.

**Result:**

* x=1002
* y=801

**(ii) Immediate Database Modification (Immediate Update)**

* **Updates are written to the database immediately**, but a log is maintained for recovery purposes.
* If a transaction fails, the log is used to undo or redo the changes.

**Recovery Actions:**

1. **For T0 (Committed):** No action required because changes are already in the database.
2. **For T1 (Uncommitted):**
   * Use the log to **undo** the changes made by T1. Rollback x to 1002 and y to 801.

**Result after Undo:**

* x=1002
* y=801

**Summary of Actions:**

* **Deferred Update:** Only committed changes are applied.
* **Immediate Update:** Undo the changes of uncommitted transactions.

**(b) Give an example of a schedule which is view equivalent but not conflict equivalent.**

**(c) What is the recovery policy in Strict Schedule?**

**(d) Explain Dirty-Read problem.**

**4(b) Example of a View Equivalent but Not Conflict Equivalent Schedule**

Two schedules are **view equivalent** if:

1. **Same Initial Reads:** Both schedules read the same initial value for each data item.
2. **Same Final Writes:** The final write on each data item is from the same transaction in both schedules.
3. **Same Reads:** If a transaction reads a value written by another transaction in one schedule, it must read the same value in the other schedule.

Two schedules are **conflict equivalent** if they have the **same conflicting operations in the same order**.

**Example:**  
Schedule S1​:

T1: R(x), W(x)

T2: R(x), W(x)

Schedule S2​:

T2: R(x), W(x)

T1: R(x), W(x)

**Conflict Equivalence Check:**

* S1​ and S2​ are **not conflict equivalent** because the order of conflicting operations W(x) differs.

**View Equivalence Check:**

* Both schedules produce the same final write on x, and every read operation retrieves the same value.
* Therefore, S1​ and S2​ are **view equivalent**.

**4(c) Recovery Policy in Strict Schedule**

In a **strict schedule**, the recovery policy ensures that:

1. **No data item is updated by a transaction until the transaction that last wrote to it has committed or aborted.**
2. **Undo is never required**, only **redo** may be needed.

**Advantages:**

* Simple and efficient recovery.
* Ensures the database is always in a consistent state.
* Prevents **cascading rollbacks**, improving reliability.

**4(d) Dirty-Read Problem (Uncommitted Dependency Problem)**

A **dirty read** occurs when a transaction reads a value written by another **uncommitted** transaction. If the uncommitted transaction later fails or rolls back, the data read by the first transaction becomes **invalid**.

**Example of Dirty Read:**

1. T1 updates x to 100 but has not committed yet.
2. T2 reads x=100 (dirty read).
3. T1 rolls back, reverting x to its original value (say 50).

Now, T2 is using an **invalid value (100)** that no longer exists in the database. This can lead to incorrect results.

**6. (a) Prove that the number of cuboid patterns produced by Drill-down from a particular cuboid in the**

**current level i is i.**

**(b) Why is holistic type of measure not preferred over other types of measures in data warehouse?**

**(c) How is the concept of data mart over the time extended to data warehouse?**

**(d) A data warehouse comprises of dimensions M, N, O, P, Q and R. Out of these O and Q does**

**not maintain any concept hierarchy. M and P have 2 concept hierarchy and N has 3 concept**

**hierarchy. Compute the total number of cuboids to represent this in the form of lattice of cuboids.**

**6(a) Prove that the number of cuboid patterns produced by drill-down from a particular cuboid in the current level iii is iii**

In a **drill-down operation**, we go from a higher level of aggregation to a lower level by splitting one dimension into its detailed components (sublevels). The number of new cuboid patterns produced at each step depends on the number of levels available to drill down.

**Proof:**

* Assume we have a cuboid at level i, which contains i dimensions at a high aggregation level.
* When we perform a drill-down on each dimension, we create a more detailed cuboid for that dimension.
* Since there are i dimensions in the cuboid, each one can be drilled down, producing i new cuboids at a more detailed level.

Thus, **the number of new cuboid patterns produced by drill-down from a cuboid at level i is i**.

**6(b) Why is holistic type of measure not preferred over other types of measures in a data warehouse?**

**Holistic measures** are aggregation functions where **the size of the result cannot be determined without scanning all records**, and they cannot be computed incrementally or in parts.

**Examples:**

* **Median, Mode, Rank**

**Reasons why they are not preferred:**

1. **Complex to Compute:** Requires processing the entire data set, making them computationally expensive.
2. **No Incremental Computation:** Cannot be computed incrementally or in distributed systems.
3. **High Storage and Memory Requirement:** Holistic measures require maintaining detailed records to compute the result accurately.
4. **Low Performance:** Slower performance compared to distributive (e.g., Sum, Count) or algebraic (e.g., Average) measures.

**6(c) How is the concept of a data mart extended to a data warehouse?**

A **data mart** is a subset of a data warehouse, typically focusing on a specific business line or department (e.g., sales, finance, marketing). Over time, multiple data marts can be integrated into a **centralized data warehouse**.

**Steps in Extension:**

1. **Departmental Focus (Data Marts):** Initial focus is on individual departments with small-scale data marts.
2. **Integration:** Data marts are gradually combined to form an enterprise-wide data warehouse.
3. **Consolidation of Data Sources:** The data warehouse consolidates data from multiple sources, providing a comprehensive and unified view.
4. **Centralized Data Management:** The data warehouse becomes the central repository for all organizational data, offering advanced analytics and reporting.

**Advantages of Extension:**

* Provides a global view of the organization.
* Reduces data redundancy.
* Improves data consistency and decision-making.

**6(d) Compute the total number of cuboids to represent the given dimensions in the form of a lattice of cuboids.**

We have six dimensions: **M, N, O, P, Q, R**.

* **O and Q have no concept hierarchy**, so they remain as a single level each.
* **M and P have 2 levels (1 base level + 1 additional concept hierarchy).**
* **N has 3 levels (1 base level + 2 additional concept hierarchies).**

The total number of cuboids is computed by multiplying the number of levels for each dimension.

Total Number of Cuboids=(2)×(3)×(1)×(2)×(1)×(1) =12 cuboids

**7. (a) Explain semi-join with example.**

**(b) How is derived horizontal fragmentation obtained by using semi-join? Explain with an example.**

**(c) Consider a database EMP(ENO, SAL, DEPTNO, LOC, JOB)**

**It is fragmented into 2 databases EMP1 and EMP2 based on the values of LOC. IF LOC =**

**'KOLKATA' then the data is stored into EMP1 and for LOC = 'MUMBAI' data is stored into**

**EMP2. EMP1 is stored in Site-1 and Site-2 and EMP2 is stored in Site-3 and Site-4. An employee**

**having ENO= 'E01 located in MUMBAI is shifted to KOLKATA. Write the query to execute this**

**for Local Mapping Transparency.**

**(d) What do you understand by 'graceful degradation' of distributed database?**

**7(a) Explain Semi-Join with Example**

A **semi-join** is used in distributed databases to reduce the amount of data transferred between sites. It helps in performing a join by first filtering rows from one relation that have matching values in another relation.

**Definition:**  
Given two relations R and S, the **semi-join** R⋉S returns the rows from R that have a matching row in S, but only the columns from R are retained.

**Example:**  
Relation **EMP** (Employee details):

| **ENO** | **ENAME** | **DEPTNO** |
| --- | --- | --- |
| E01 | Alice | 10 |
| E02 | Bob | 20 |
| E03 | Carol | 30 |

Relation **DEPT** (Department details):

| **DEPTNO** | **DNAME** |
| --- | --- |
| 10 | HR |
| 20 | Finance |
| 40 | Marketing |

**Semi-join EMP⋉DEPT on DEPTNO:**

* It filters the rows from **EMP** where **DEPTNO** exists in **DEPT**.
* Result (only rows from **EMP** with matching **DEPTNO**):

| **ENO** | **ENAME** | **DEPTNO** |
| --- | --- | --- |
| E01 | Alice | 10 |
| E02 | Bob | 20 |

**7(b) Derived Horizontal Fragmentation using Semi-Join**

**Derived Horizontal Fragmentation** partitions a relation based on the **semi-join** with another related relation. This helps in minimizing data distribution and improving query performance in distributed databases.

**Example:**  
Consider two relations:

1. **EMP(ENO, ENAME, DEPTNO)**
2. **DEPT(DEPTNO, DNAME, LOC)**

Suppose we want to horizontally fragment **EMP** based on **LOC** in **DEPT**:

* **LOC = 'KOLKATA'** → Assign employees whose departments are located in Kolkata to one fragment.
* **LOC = 'MUMBAI'** → Assign employees from Mumbai to another fragment.

We perform a **semi-join** between **EMP** and **DEPT** on **DEPTNO**, filtering employees based on the location from **DEPT**.

**Resulting fragments:**

* **EMP1** (employees in Kolkata)
* **EMP2** (employees in Mumbai)

**7(c) Query for Local Mapping Transparency**

To move an employee with **ENO = 'E01'** from **MUMBAI (EMP2)** to **KOLKATA (EMP1)**:

-- Delete the employee record from EMP2 (Site-3 or Site-4)

DELETE FROM EMP2 WHERE ENO = 'E01';

-- Insert the employee record into EMP1 (Site-1 or Site-2) with updated LOC = 'KOLKATA'

INSERT INTO EMP1 (ENO, SAL, DEPTNO, LOC, JOB)

VALUES ('E01', <SAL>, <DEPTNO>, 'KOLKATA', <JOB>);

*Note:* Ensure you retrieve the appropriate values for **SAL**, **DEPTNO**, and **JOB** before the insert.

**7(d) What is Graceful Degradation in Distributed Databases?**

**Graceful degradation** refers to the ability of a **distributed database system** to continue functioning, though with reduced performance or limited functionality, when some components fail.

**Key Characteristics:**

1. **Partial Failure Handling:** When a failure occurs in one part of the system, the unaffected parts continue to operate.
2. **No System-wide Crash:** The entire system does not crash; only certain services may be affected.
3. **Fault Tolerance:** Graceful degradation ensures that the system remains available for critical operations.
4. **Example:** If one database site becomes unavailable, queries can be processed by other sites, although response times might increase.

**Example:**

* In a distributed e-commerce application, if a data node storing customer reviews fails, the main product catalog can still be available to users, though the review feature may be temporarily disabled.