Unit -3

Functional Dependency

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.

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

The main use of FD is to describe further a relational schema R by specifying constraints on its attributes that must hold all the times.

Functional Dependency (α → β):

- A functional dependency is a relationship between sets of attributes in a relation schema.
- In the given definition, α and β are subsets of the attributes of relation schema R. The notation α → β represents a functional dependency from α to β.
- The functional dependency α → β means that for any two tuples (rows) in an instance of relation schema R, if they have the same values for the attributes in α, then they must also have the same values for the attributes in β.

Functional Dependency Holding on Schema (α → β holds on r(R)):

- This statement means that the functional dependency α → β holds for all legal instances of the relation schema r(R).

- In other words, no matter what data is stored in the relation instance, if the attributes in α are the same for two tuples, then the attributes in β must also be the same for those tuples.

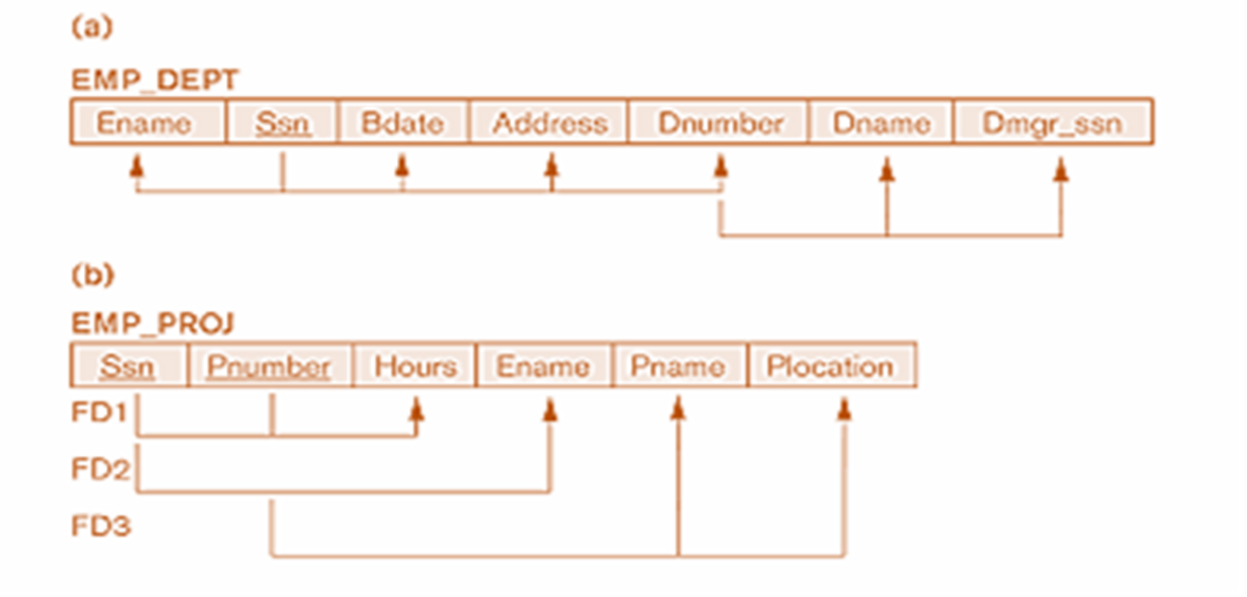
Superkey (K is a superkey for r(R)):

- A superkey is a set of one or more attributes that, taken collectively, uniquely identifies a tuple in a relation.
- In the given context, K is considered a superkey for r(R) if the functional dependency K → R holds on r(R).
- This means that for any legal instance of the relation schema r(R), if two tuples have the same values for the attributes in K, then they must also have the same values for all attributes in R.

To identify Functinonal dependency

Apply Armstrong's Axioms:

Armstrong's axioms are a set of rules that can be used to derive functional dependencies. These axioms include reflexivity, augmentation, and transitivity. Applying these rules can help deduce additional functional dependencies.



Full Functional Dependency:

- **Definition:**
  - A functional dependency $P \rightarrow Q$ is a full functional dependency if and only if the determinant (P) is either a candidate key or a superkey, and the dependent (Q) can be either a prime or non-prime attribute.
- **Explanation:**
  - In a full functional dependency, the determinant uniquely determines the dependent, and removing any attribute from the determinant would break the dependency.
- **Example:**
  - Given the functional dependency $ABC \rightarrow D$.
    - ABC is the determinant.
    - D is the dependent.
    - BC, C, and A individually cannot determine D.
    - Therefore, D is fully functionally dependent on ABC.

Partial Functional Dependency:

- **Definition:**
  - A functional dependency is partial if a non-prime attribute of the relation is derived by only a part of the candidate key. In other words, one or more non-key attributes depend on only a portion of the primary key.
- **Explanation:**
  - In a partial functional dependency, removing any attribute from the determinant would still maintain the dependency, indicating that the dependency is not fully dependent on the entire determinant.
- **Example:**
  - Given the determinants $AC \rightarrow P$, $A \rightarrow D$, $D \rightarrow P$.
    - AC, A, and D are determinants.

- P is the dependent.
- If we find the closure of A (A+), we get A → D, D → P, which implies A → P.
- Since C is not necessary for determining P, P is partially dependent on AC.

Transitive Functional Dependency:(Dervied Dependency)

- **Definition:**
  - A transitive functional dependency occurs when a non-prime attribute of a relation is derived by either another non-prime attribute or the combination of a part of the candidate key along with a non-prime attribute.
- **Explanation:**
  - If X → Y and Y → Z, then it implies a transitive dependency X → Z. In other words, the dependency "transfers" through an intermediary attribute.
- **Example:**
  - Given X → Y and Y → Z, we can determine if X → Z holds. If it does, it signifies a transitive dependency where the dependency flows through the attribute Y.

Trivial Functional Dependency: (Subset)

- **Definition:**
  - A trivial functional dependency is related to the reflexive rule. If X is a set of attributes, and Y is a subset of X, then X → Y holds.
- **Explanation:**
  - Trivial dependencies are those where the dependent attributes are a subset of the determinant attributes. They are often considered "obvious" or self-evident.
- **Example:**

- ABC → BC is a trivial dependency because BC is a subset of the determinant ABC. Here, X is {ABC} and Y is {BC}.

Multi-Valued Dependency:

- **Definition:**
  - In a relation with three fields X, Y, and Z, a multivalued dependency (MVD) exists if, for each value of X, there is a well-defined set of values Y and a well-defined set of values Z, and the set of values Y is independent of the set of values Z. This is denoted as $X \rightarrow Y/Z$.
- **Minimum Attributes Required:**
  - A multivalued dependency requires at least three attributes in the relation: the determinant attribute (X) and two dependent attributes (Y and Z).
- **Explanation:**
  - Multivalued dependency captures a scenario where changes in the determinant attribute (X) uniquely determine sets of values for both Y and Z, and these sets are independent of each other.

Example:

- **Scenario:**
  - Suppose there is a bike manufacturer company producing two colors (white and black) of each model every year. The relevant attributes are BIKE_MODEL, COLOR, and MANUF_YEAR.
- **Dependency:**
  - The example states that COLOR and MANUF_YEAR are dependent on BIKE_MODEL and independent of each other.
- **Representation:**
  - The multivalued dependencies are represented as follows:

BIKE_MODEL →→ MANUF_YEAR
BIKE_MODEL →→ COLOR

  - The double-headed arrow (→→) indicates the multivalued dependency. For each value of BIKE_MODEL, there is a well-defined set of values for

MANUF_YEAR and a well-defined set of values for COLOR, and these sets are independent of each other.

- **Interpretation:**
  - Changes in the model of the bike (BIKE_MODEL) uniquely determine sets of manufacturing years (MANUF_YEAR) and sets of colors (COLOR), and the sets of colors are independent of the sets of manufacturing years.

Non-trivial functional dependency

- A → B has a non-trivial functional dependency if B is not a subset of A.
- When A intersection B is NULL, then A → B is called as complete non-trivial.

$$A^{intersect}B = NULL$$

Example:

- ID → Name,
- Name → DOB

Lossless Decompositon

The concept of lossless decomposition in the context of relational databases ensures that splitting a relation into two relations and then joining them will yield the original relation without any loss of information. In terms of functional dependencies, certain dependencies need to hold for this to be true.

Given a relation schema R, and a decomposition into $R1$ and $R2$, the decomposition is lossless if either of the following functional dependencies holds:

1. $R1 \cap R2 \rightarrow R1$
2. $R1 \cap R2 \rightarrow R2$

This implies that the common attributes ($R1 \cap R2$) between $R1$ and $R2$ functionally determine at least one of the original relations ($R1$ or $R2$). If either of these dependencies holds, the decomposition is lossless.

In SQL terms, you can use the NATURAL JOIN operation to combine the two relations and check if it produces the original relation. Here's how you might express it:

```
sqlCopy code
SELECT * FROM (SELECT R1.* FROM R1 NATURAL JOIN R2) AS Result
```

This SQL statement selects all attributes from the result of the natural join of *R*1 and *R*2, and then compares it with the original relation *R*. If the result matches *R*, the decomposition is lossless.

LOSELESS JOIN

A lossless join property is a desirable characteristic in database design, especially in the context of decomposition. It ensures that if you decompose a relation into smaller relations and then perform a natural join on those smaller relations, you will still be able to obtain the original relation without losing any information. In other words, the join operation will not introduce spurious tuples.

Example of Lossless Join:

Consider a relation schema R(A, B, C, D) with a set of functional dependencies

$$F = AB \rightarrow C, CD \rightarrow A$$

The original relation might look like this:

Original Relation (R):

A	B	C	D
1	2	3	4
5	6	7	8
9	10	11	12

Now, let's decompose \(R\) into two relations

$$(R1(A, B, C))(R2(C, D, A))$$

using the functional dependencies provided:

Decomposed Relations:

\(R1\):

A	B	C
1	2	3
5	6	7
9	10	11

\(R2\):

C	D	A
3	4	1
7	8	5
11	12	9

Now, let's perform a natural join on \(R1\) and \(R2\) to see if we can obtain the original relation \(R\).

Join Operation:

```
SELECT * FROM R1 NATURAL JOIN R2;
```

Result:

A	B	C	D
1	2	3	4
5	6	7	8
9	10	11	12

In this example, the natural join of \(R1\) and \(R2\) produces the original relation \(R\), which demonstrates the lossless join property. The decomposition into \(R1\) and \(R2\) did not lose any information, and the join operation successfully reconstructs the original relation.

Lossless Join Property Explanation:

The lossless join property is maintained when the common attributes

$$(R1 \cap R2)$$

used for the decomposition are a superkey in at least one of the decomposed relations

$$either(R1)or(R2)$$

In this example, (C) is a superkey in (R1) and (R2), ensuring the lossless join property.

Armstrong Axioms

Armstrong's inference rules, also known as Armstrong's axioms, provide a set of rules for inferring additional functional dependencies based on a given set of functional dependencies. These rules are fundamental in the process of deriving the closure of a set of functional dependencies. The three main inference rules are as follows:

IR1-Reflexive , IR2-Augmentation , IR3-Transitive

1. Reflexive Rule (IR1):

- **Rule:**
  - If Y is a subset of X, then $X \rightarrow Y$ always holds.
- **Example:**
  - If $\{Ssn\} \rightarrow \{Ename\}$, then $\{Ssn, Bdate\} \rightarrow \{Ename, Bdate\}$.

2. Augmentation Rule (IR2):

- **Rule:**
  - If $X \rightarrow Y$, then $XZ \rightarrow YZ$ always holds, where XZ stands for the union of sets X and Z.
- **Example:**
  - If $\{Ssn\} \rightarrow \{Ename\}$, then $\{Ssn, Bdate\} \rightarrow \{Ename, Bdate\}$.

3. Transitive Rule (IR3):

- **Rule:**
  - If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$ always holds.
- **Example:**
  - If $\{Ssn\} \rightarrow \{Dnumber\}$ and $\{Dnumber\} \rightarrow \{Dname\}$, then $\{Ssn\} \rightarrow \{Dname\}$.

- Decomposition: If X → YZ, then X → Y and X → Z
- Union/Additive: If X → Y and X → Z, then X → YZ

  Example: if {Ssn} → {Ename} and {Ssn} → {Address} then {Ssn}→ {Ename, Address}
- Pseudo-Transitivity: If X → Y and WY → Z, then WX → Z

Soundness and Completeness:

- These three rules (IR1, IR2, IR3) together form a sound and complete set of inference rules.
- **Soundness:** These rules are sound, meaning that if a functional dependency follows from the rules, it is true in all cases.
- **Completeness:** These rules are complete, meaning that all functional dependencies that hold can be deduced from these rules.

Armstrong's Axioms:

- By applying these rules repeatedly, you can find all dependencies in the closure (F^+) given a set of dependencies (F).
- The collection of these rules is known as Armstrong's axioms, which provide a systematic way of determining all possible functional dependencies that logically follow from a given set.

These axioms are widely used in database normalization and are essential for ensuring the integrity of the relational data model.

example:

Supposing we are given a relation R{A, B, C, D, E, F} with a set of FDs as shown below:

A → BC

B → E

CD → EF

Let us show that the FD AD → F holds for R and is a member of the closure.

(1) A → BC	{Given, A->B & A->C}
(2) A → C	{Decomposition of (1)}
(3) AD → CD	{Augmentation of (2) by adding D}
(4) CD → EF	{Given}
(5) AD → EF	{Transitivity of (3) and (4)}
(6) AD → F	{Decomposition of (5)}

Consider the relation schema <R,F> where R= (ABCDEGHI) and dependencies

F= { AB->E,AG->J, BE->I, E ->G GI ->H }. Show that AB ->GH is derived by F.

Step	Statement	Explanation
1	AB ->E	Given
2	E ->G	Given
3	BE->I	Given
4	GI ->H	Given
5	AB ->G	Transitivity on (1) and (2)
6	AB -> BE	Augmentation (1) by B
7	AB -> I	Transitivity on (6) and (3)
8	AB -> GI	Union on (5) and (7)
9	AB -> H	Transitivity on (8) and (4)
10	AB -> GH	Union on (5) and (9)

Features of good relational design

- GUIDELINE 1: Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes).
  - Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation
  - Only foreign keys should be used to refer to other entities
  - Entity and relationship attributes should be kept apart as much as possible.

Examples of Violating Guideline 1

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn

Mixes attributes of employees and departments

EMP_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation

Mixes attributes of employees and projects and the WORKS_ON relationship

Anomalies - Update , Insert , Delete

ANOMALY-1

EXAMPLE OF AN UPDATE ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Update Anomaly:
  - Changing the name of project number P1 from “Billing” to “Customer-Accounting” may cause this update to be made for all 100 employees working on project P1.

An update anomaly is a situation that can occur in a relational database when data redundancy leads to inconsistencies during the update process. It typically arises in databases that are not well-normalized and contain duplicate or redundant information.

Example of Update Anomaly:

Consider a denormalized table that stores information about employees, including their department and manager. The table might look like this:

```
| EmployeeID | EmployeeName | Department   | Manager      |
|------------|--------------|--------------|--------------|
| 101        | Alice        | HR           | Bob          |
| 102        | Bob          | IT           | Charlie      |
| 103        | Charlie      | Marketing    | Alice        |
```

In this table:

- `EmployeeID` is the primary key.
- `Department` represents the department to which an employee belongs.
- `Manager` represents the manager of each employee.

Now, let's say Alice gets promoted and moves from the HR department to Marketing. To reflect this change, we need to update both her department and her manager. However, because of the denormalized structure, we need to update multiple rows:

1. Update Alice's department to Marketing.
2. Update Alice's manager to the new manager of the Marketing department.

Potential Issues:

1. **Inconsistency:**
   - If the update process is interrupted after updating the department but before updating the manager, the data becomes inconsistent. Alice's manager would be out of sync with her new department.
2. **Complexity:**
   - The need to update multiple rows for a single logical change introduces complexity. This complexity increases with the number of rows referencing the same piece of information.
3. **Increased Risk of Errors:**
   - The likelihood of errors increases as more updates are required. It's easy to forget to update one of the rows, leading to inconsistencies in the data.

Resolution:

A better way to structure the data would be to normalize the table, separating employee information from department information. For example:

Employee Table:

```
| EmployeeID | EmployeeName | DepartmentID | ManagerID   |
|------------|--------------|--------------|-------------|
| 101        | Alice        | 3            | 103         |
| 102        | Bob          | 1            | 104         |
| 103        | Charlie      | 2            | NULL        |
```

Department Table:

```
| DepartmentID | DepartmentName |
|--------------|----------------|
| 1            | HR             |
| 2            | IT             |
| 3            | Marketing      |
```

Manager Table:

```
| ManagerID | ManagerName |
|-----------|-------------|
| 103       | Alice       |
| 104       | Bob         |
```

With this normalized structure, updates to an employee's department or manager are more straightforward and less prone to anomalies. The information is stored in a more modular way, reducing redundancy and improving data integrity.

Anomaly-2

EXAMPLE OF AN INSERT ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Insert Anomaly:
  - Cannot insert a project unless an employee is assigned to it.
- Conversely
  - Cannot insert an employee unless an he/she is assigned to a project.

An insert anomaly occurs in a relational database when it is not possible to add certain information to the database without adding unnecessary data or introducing redundancy. This issue is a consequence of poor database design, specifically when data is not properly normalized.

Let's illustrate the insert anomaly with an example:

Consider a poorly designed table for a university database:

```
Student_Courses (Student_ID, Student_Name, Course_ID, Course_Nam
e)
```

In this table, let's say a student can enroll in multiple courses, and a course can have multiple students. However, the structure of this table is problematic, and it leads to an insert anomaly.

Now, suppose a new student joins the university, and they have not yet enrolled in any courses. In a properly designed database, you should be able to insert information about this student without specifying any courses they are enrolled in. However, with the current design, it is not possible to insert data for this student without creating redundancy or introducing NULL values.

Here is the problem:

```
Student_Courses
| Student_ID | Student_Name | Course_ID | Course_Name |
|------------|--------------|-----------|-------------|
| 101        | Alice        | CSCI101   | Database    |
| 101        | Alice        | CSCI102   | Algorithms  |
| 102        | Bob          | NULL      | NULL        |  <-- New
student, cannot enroll in any courses
```

In this example, the insertion of the new student (Bob) is problematic because the table structure does not allow for students who are not currently enrolled in any courses. Introducing NULL values for Course_ID and Course_Name would be a workaround, but it violates the principles of database normalization and can lead to data inconsistencies.

To address this insert anomaly, the database design should be normalized, and a separate table for students and a separate table for courses should be created. The relationship between students and courses can then be represented using a junction table, allowing for more flexibility and eliminating the insert anomaly.

Anomaly-3

EXAMPLE OF A DELETE ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Delete Anomaly:
  - When a project is deleted, it will result in deleting all the employees who work on that project.
  - Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

A delete anomaly is a situation in a relational database where the deletion of data leads to unintended loss of information. This often occurs when deleting a record (tuple) in a table that has dependencies on other tables.

Example of Delete Anomaly:

Consider a simple example of a database that contains information about employees and their projects. We have two tables: `Employees` and `Projects`. The tables may look like this:

Employees:

EmpID	EmpName	ProjectID
101	Alice	P001
102	Bob	P002
103	Charlie	P001
104	David	P003

Projects:

ProjectID	ProjectName
P001	ProjectA
P002	ProjectB
P003	ProjectC

Now, let's say Charlie (EmpID 103) leaves the company, and we decide to remove his record from the `Employees` table.

Before Deletion:

EmpID	EmpName	ProjectID
101	Alice	P001
102	Bob	P002
103	Charlie	P001
104	David	P003

After Deletion:

EmpID	EmpName	ProjectID
101	Alice	P001
102	Bob	P002
104	David	P003

Delete Anomaly Explanation:

The delete operation leads to a situation where the information about ProjectA (P001) is lost from the `Employees` table, even though Alice is still associated with ProjectA. Charlie was the only employee associated with ProjectA, and by deleting his record, we unintentionally lose information about that project.

Now, if we want to find information about ProjectA and its employees, we can't retrieve it directly from the `Employees` table after the deletion. This is a delete anomaly.

Avoiding Delete Anomalies:

To avoid delete anomalies, one can use normalization techniques, such as breaking the relation into multiple tables and using foreign key relationships. In this case, a separate table for the employee-project relationship would prevent the loss of project information when an employee record is deleted.

- GUIDELINE 2:
  - Design a schema that does not suffer from the insertion, deletion and update anomalies.
  - If there are any anomalies present, then note them so that applications can be made to take them into account.

- GUIDELINE 3:
  - Relations should be designed such that their tuples will have as few NULL values as possible
  - Attributes that are NULL frequently could be placed in separate relations (with the primary key)
- Reasons for nulls:
  - Attribute not applicable or invalid
  - Attribute value unknown (may exist)
  - Value known to exist, but unavailable

- GUIDELINE 4:
  - The relations should be designed to satisfy the lossless join condition.
  - No spurious tuples should be generated by doing a natural-join of any relations.

SPURIOUS TUPLES

A spurious tuple is an unintended or incorrect tuple that is introduced during a join operation in a relational database. This phenomenon occurs when tables are decomposed and then rejoined using a natural join or other join operations, leading to the inclusion of tuples that were not part of the original relations. The presence of spurious tuples violates the lossless join property.

Let's illustrate the concept of a spurious tuple with an example:

Consider a relation $R(A, B, C)$ with the functional dependency $(A \rightarrow B)$ and another functional dependency $(C \rightarrow B)$. This leads to a lossless decomposition into two relations: $R1(A, B)$ with $(A \rightarrow B)$ and $R2(C, B)$ with $(C \rightarrow B)$.

1. **Original Relation (R):**

A	B	C
1	2	3
4	5	6
7	8	9

2. **Decomposed Relations:**
   - $R1(A, B)$:

A	B
1	2
4	5
7	8

   - $R2(C, B)$:

C	B
3	2
6	5
9	8

3. **Join Operation Result:**

A	B	C
1	2	3
1	2	6 ◗
1	2	9 text{(spurious tuple)}
4	5	6
4	5	9 text{(spurious tuple)}
7	8	9

```
| 7 | 8 | 6 text{(spurious tuple)} |
| 7 | 8 | 3 text{(spurious tuple)} |
```

Explanation:

- **Cause of Spurious Tuples:**
  - The spurious tuples are introduced because the common attribute (B) is not a superkey in either (R1) or (R2). As a result, the join operation combines tuples based on common values of (B), leading to unintended combinations.
- **Lossless Join Property Violation:**
  - The presence of spurious tuples violates the lossless join property because the result of the join does not match the original relation (R).

To avoid spurious tuples, it is essential to ensure that the common attributes used for the decomposition and subsequent join form a superkey or candidate key in at least one or both of the decomposed relations. This adherence to the lossless join property helps maintain the integrity of the data during decomposition and recombination.

Normalization and Lossy Decomposition

Normalization:

Normalization is a database design process aimed at organizing data to minimize redundancy and dependency. The goal is to improve data integrity, reduce redundancy, and avoid anomalies during data manipulation. Normalization involves decomposing relations into smaller, well-structured relations based on functional dependencies.

Example of Normalization:

Consider a relation (StudentID, CourseID, Instructor, CourseName) with functional dependencies. To normalize, decompose into two relations:

- (StudentID, CourseID, Instructor)
- (CourseID, CourseName)

Lossy Decomposition:

A lossy decomposition is where decomposing and rejoining results in the loss of some information or inconsistency in tuples violating the lossless join property causing suprious tuples.

Example of Lossy Decomposition:

Consider a relation (A, B, C) with (A → B). If decomposed into (A, B) and (B, C) and then joined, we might lose information.

Original Relation R

A	B	C
1	2	3
4	5	6

Decomposed Relations (Lossy):

- (A, B)

A	B
1	2
4	5

- (B, C)

B	C
2	3
5	6

Join Operation Result (Lossy):

A	B	C
?	2	3
?	5	6

In this case, the lossy decomposition leads to the loss of information in attribute (A), and we cannot completely reconstruct the original relation.

CLOSURE

Closure Definition:

Given a set of attributes X and a set of functional dependencies F, the closure $X+$ is the set of all attributes that can be functionally determined by X based on the dependencies in F. Formally, it is the smallest set of attributes Y such that:

1. $X \subseteq Y$
2. For every functional dependency $Z \rightarrow W$ in F, if $Z \subseteq Y$, then $W \subseteq Y$.

- Formally, the set of all dependencies that include F as well as all dependencies that can be inferred from F is called the closure of F; it is denoted by F+
- F = {Ssn → {Ename, Bdate, Address, Dnumber}, Dnumber → {Dname, Dmgr_ssn} }
- Some of the additional functional dependencies that we can infer from F are the following:
  - Ssn → {Dname, Dmgr_ssn}
  - Ssn → Ssn
  - Dnumber → Dname
- The closure F+ of F is the set of all functional dependencies that can be inferred from F.

Closure should follow Armstrong's Axioms

To compute closure

- To compute the closure of a set of functional dependencies F:

```
F+ = F
repeat
    for each functional dependency f in F+
        apply reflexivity and augmentation rules on f
        add the resulting functional dependencies to F+
    for each pair of functional dependencies f1 and f2 in F+
        if f1 and f2 can be combined using transitivity
            then add the resulting functional dependency to F+
until F+ does not change any further
```

We will get back to this later on, for now you can look into the algorithm that one would use for determining the closure set.

NOTE:
Since a set of size n has 2n subsets, there are a total of

$$2^n \times 2^n = 2^{(2^n)}$$

possible functional dependencies, where n is the number of attributes in R

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- Some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

Equivalent Set

Analogy: Books in a Library

Imagine you have two lists of books, List A and List B, in a library. Each list represents a set of rules about which books you can borrow based on certain conditions.

1. **Covering:**
    - **Definition:** List A is said to "cover" List B if every book that you can borrow according to List B is also mentioned in List A.
    - **Analogy:** If List A covers List B, it means that every book that you are allowed to borrow according to List B is also listed in List A. List A provides more information and includes all the books from List B.
2. **Equivalence:**
    - **Definition:** Lists A and B are considered "equivalent" if they allow you to borrow the same set of books. In other words, the rules in List A and List B are the same.
    - **Analogy:** If Lists A and B are equivalent, it means that the conditions for borrowing books are exactly the same in both lists. Any book you can borrow according to List A, you can also borrow according to List B, and vice versa.
3. **Determining Coverage:**
    - **Process:** To check if List A covers List B, you go through each rule in List B, apply the rules from List A, and see if every book mentioned in List B is also mentioned in List A.
    - **Analogy:** You check each borrowing condition in List B against the more comprehensive rules in List A. If List A includes all the conditions from List B, it covers List B.

Bringing it Back to Functional Dependencies:

Now, let's relate this analogy to functional dependencies:

- **Lists of Books:** Lists A and B represent sets of functional dependencies in a database.
- **Books You Can Borrow:** Books represent attributes in a database table.
- **Borrowing Conditions:** Rules in functional dependencies define conditions under which certain attributes can be determined by others.

So, when we say one set of functional dependencies covers another, it means that every dependency in the second set can be determined using the rules of the first

set. If two sets are equivalent, it means they provide the same rules for determining attributes.

To check if one set of functional dependencies covers another, we calculate the closure of the attributes based on the rules in the first set and see if it includes the attributes specified in the second set.

- Two sets of functional dependencies E and F are equivalent
  - if E+ = F+.
  - equivalence means that every FD in E can be inferred from F, and every FD in F can be inferred from E; that is, E is equivalent to F if both the conditions—E covers F and F covers E—hold.
- We can determine whether F covers E by calculating X+ with respect to F for each FD X → Y in E, and then checking whether this X+ includes the attributes in Y. If this is the case for every FD in E, then F covers E.

example

Sample Questions

Q.1 Let us take an example to show the relationship between two FD sets. A relation R(A,B,C,D) having two FD sets F = {A->B, B->C, AB->D} and G = {A->B, B->C, A->C, A->D}

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

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

As all FDs in set FD1 also hold in set FD2, **FD2 ⊃ FD1** is true.

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

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

As all FDs in set FD2 also hold in set FD1, **FD1 ⊃ FD2** is true.

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

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

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

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

As all FDs in set FD1 also hold in set FD2, **FD2 ⊃ FD1** is true.

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

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

As all FDs in set FD2 do not hold in set FD1, **FD2 ⊄ FD1**.

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

Dependency Preservation

Dependency preservation is a critical concept in database design and normalization. It refers to the property of ensuring that certain **functional dependencies observed in the original relation are retained in the decomposed relations after normalization**.

When a relation is decomposed into smaller relations, it's done to eliminate redundancy and improve the overall design. However, during this process, it's essential to preserve certain functional dependencies to maintain data integrity.

Example:

Let's consider an example to understand dependency preservation:

Original Relation: (A, B, C, D) with functional dependencies (A→B) and (C → D).

1. **Decomposition 1 (Normalization):**
   - Decompose R into R1(A, B) and R2(C, D).

   The functional dependencies are preserved because $(A \rightarrow B)$ is still represented in (R_1) and $(C \rightarrow D)$ is represented in (R_2).

2. **Decomposition 2 (Non-Normalization):**
   - Decompose (R) into R3(A, C) and R4(B, D)

   The functional dependencies are not preserved because A → B is not represented in R3 and (C → D) is not represented in (R4).

Dependency Preservation Techniques:

1. **Lossless Join Decomposition:**
   - Ensures that the decomposed relations can be joined back together without losing information. This helps in preserving functional dependencies.
2. **Projection Decomposition:**
   - Decompose the relation by projecting it onto certain attributes. If a functional dependency involves only the projected attributes, it is preserved.
3. **Synthesis Decomposition:**
   - Decompose the relation by synthesizing it into multiple relations. This is often done when normalization is not feasible.
4. **Normalization:**
   - Applying normalization rules carefully to avoid introducing new functional dependencies and to ensure that the original dependencies are preserved.

Importance of Dependency Preservation:

- **Data Integrity:** Preserving functional dependencies ensures that the relationships between attributes are maintained, preventing inconsistencies in the data.

- **Query Optimization:** Preserving dependencies allows for more efficient query processing by utilizing the existing relationships between attributes.
- **Correctness:** It ensures that the decomposed relations accurately represent the information contained in the original relation.

In summary, dependency preservation is crucial during the decomposition of relations to maintain the reliability and correctness of the database schema. Careful consideration of normalization techniques and decomposition strategies is essential to achieve dependency preservation.

•Consider a schema:

• *dept_advisor(s_ID, i_ID, department_name)*

•With function dependencies:

• *i_ID → dept_name*

• *s_ID, dept_name → i_ID*

•

•In the above design we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship.

•To fix this, we need to decompose *dept_advisor*

•Any decomposition that we will do for dept_advisor will not include all the attributes i.e *s_ID, dept_name, i_ID*

• so following fd will not be preserved at all

• *s_ID, dept_name → i_ID*

•Thus, the decomposition NOT be dependency preserving

Minimal Covers

The minimal cover of a set of functional dependencies (FDs) is a reduced set that captures the same closure as the original set of dependencies. In other words, **it represents an equivalent set of dependencies with the fewest possible dependencies needed to infer the same information about the attributes.**

The process of finding the minimal cover involves eliminating redundancy and unnecessary dependencies from the original set while retaining the closure properties. The minimal cover is important in database design, as it provides a more concise and efficient representation of the dependencies.

Steps to Find the Minimal Cover:

1. **Redundant Dependency Elimination:**

- Remove any redundant dependencies from the original set. Redundancy can occur when one dependency can be inferred from another in the set.

2. **Augmentation:**
   - For each dependency $(X \rightarrow Y)$, check if any attribute in (X) can be removed while preserving the closure. If an attribute can be removed, the dependency is not minimal.
3. **Decomposition:**
   - For each dependency $(X \rightarrow Y)$, check if \(X\) can be further decomposed while preserving the closure. If \(X\) can be decomposed into smaller sets, it indicates redundancy.
4. **Transitivity Check:**
   - Check for transitive dependencies and eliminate unnecessary dependencies. If $(X \rightarrow Y)$ and $(Y \rightarrow Z)$, and $(X \rightarrow Z)$ can be inferred from the original set, then $(X \rightarrow Y)$ is redundant.

Example:

Consider the set of functional dependencies:

1. $A \rightarrow BC$
2. $B \rightarrow AC$
3. $C \rightarrow AB$
4. $AB \rightarrow C$

The minimal cover of this set would be:

1. $A \rightarrow B$
2. $A \rightarrow C$
3. $B \rightarrow A$
4. $C \rightarrow A$

Explanation:

- The original set contains redundancy, such as $A \rightarrow BC$ and $AB \rightarrow C$. These dependencies are eliminated in the minimal cover.
- The dependencies $B \rightarrow AC$ and $C \rightarrow AB$ are decomposed into individual dependencies.
- The transitive dependency $AB \rightarrow C$ is replaced by $A \rightarrow C$ and $A \rightarrow B$.

Importance of Minimal Cover:

1. **Efficiency:** The minimal cover is a more efficient representation, as it avoids unnecessary dependencies and redundancy.
2. **Normalization:** In the context of database normalization, the minimal cover is essential for identifying the key dependencies that must be preserved during decomposition.
3. **Query Optimization:** A minimal set of dependencies helps in optimizing database queries by providing a more streamlined and accurate understanding of the relationships between attributes.

In summary, the minimal cover is a refined set of functional dependencies that preserves the same closure as the original set while eliminating redundancy and unnecessary dependencies.

How to solve

Algorithm Steps:

1. **Initialization:**
   - Set F as a copy of E.
2. **Canonical Form:**
   - Replace each functional dependency $X \rightarrow \{A_1, A_2, ..., A_n\}$ in F with the n individual dependencies $X \rightarrow A_1, X \rightarrow A_2, ..., X \rightarrow A_n$. (*Canonical Form Preparation*)
3. **Extraneous Attribute Removal:**
   - For each functional dependency $X \rightarrow A$ in F:
     - For each attribute B in X:
       - If $\{F - \{X \rightarrow A\}\} \cup \{(X - \{B\}) \rightarrow A\}$ is equivalent to F:
         - Replace $X \rightarrow A$ with $(X - \{B\}) \rightarrow A$ in F.
         - (*Extraneous attribute removal*).
4. **Redundant Dependency Removal:**
   - For each remaining functional dependency $X \rightarrow A$ in F:
     - If $\{F - \{X \rightarrow A\}\}$ is equivalent to F:
       - Remove $X \rightarrow A$ from F.
       - (*Redundant dependency removal*).

examples

Given Set of Functional Dependencies E: {B → A, D → A, AB → D}

Minimum Cover Algorithm:

Step 1: Canonical Form Preparation

- All given dependencies are already in canonical form (having only one attribute on the right-hand side).

Step 2: Extraneous Attribute Removal

- We examine whether AB → D has any extraneous attributes on the left-hand side.
- By applying augmentation (IR2) to B → A, we obtain B → AB.
- Since AB → D is given, by transitivity (IR3), we deduce B → D.
- Therefore, AB → D can be replaced by B → D.
- The new set is E': {B → A, D → A, B → D}.

Step 3: Redundant Dependency Removal

- We check for redundant dependencies in E'.
- By applying transitivity to B → D and D → A, we derive B → A.
- Hence, B → A is redundant in E' and can be eliminated.
- The minimum cover is F: {B → D, D → A}.

Conclusion:

- The original set E can be inferred from the minimum cover F.
- E and F are equivalent sets.

SUPERKEY Algorithm

This algorithm is designed to find a superkey (or candidate key) for a given relation (R) based on a set of functional dependencies (F). A superkey is a set of attributes that uniquely determines all other attributes in the relation.

- **Algorithm 15.2a Finding a Key K for R, given a set F of Functional Dependencies**
  - **Input: A universal relation R and a set of functional dependencies F on the attributes of R.**

1. Set K := R;
2. For each attribute A in K
   { Compute $(K - A)^+$ with respect to F;
   If $(K - A)^+$ contains all the attributes in R,
   then set K := K - {A} };

Let's break down the steps:

Input:

- R: Universal relation.
- F: Set of functional dependencies on the attributes of R.

Algorithm Steps:

1. **Initialization:**
   - Set K to be the set of all attributes of R.
2. **For each attribute A in K:**
   - Compute the closure $(K - A)^+$ with respect to F. This involves determining all the attributes that can be functionally determined by the remaining attributes in K after excluding A.
   - If $(K - A)^+$ contains all the attributes in R, then A is not necessary for K to be a superkey. Therefore, set K to be $K - \{A\}$.

Explanation:

- The algorithm starts with the assumption that all attributes of R form a potential superkey (K).
- It iterates through each attribute A in K and checks whether the closure of $(K - A)$ with respect to F still includes all the attributes of R.
- If removing A maintains the ability to determine all attributes in R, then A is not necessary for K to be a superkey, so it is removed from K.

Objective:

The goal of this algorithm is to iteratively refine the set \(K\) by eliminating attributes that are not essential for forming a superkey. The final result will be a minimal superkey for the given relation \(R\) with respect to the provided set of functional dependencies \(F\).

What is SuperKey

A superkey is a set of attributes in a relation (or table) that, taken together, uniquely identifies each tuple (or row) in that relation. In other words, a superkey is a combination of attributes whose values, when taken collectively, can uniquely determine the values of all other attributes in a tuple.

Here's a more detailed explanation along with an example:

Definition:

A superkey K for a relation R is a set of attributes such that, for every pair of distinct tuples $t1$ and $t2$ in R, if $t1[K] = t2[K]$, then $t1 = t2$. This means that the values of the attributes in K are sufficient to uniquely identify each tuple in the relation.

Example:

Consider a relation Student with the following attributes:

- StudentID
- Name
- DateOfBirth
- Department

Now, let's define a few sets of attributes:

1. **Superkey Example 1:**
    - K1 = {StudentID}
    - Explanation: The StudentID alone is sufficient to uniquely identify each student in the relation. Even if two students have the same name, date of birth, or department, their StudentID will be different.
2. **Superkey Example 2:**
    - K2 = {StudentID, Name}

- Explanation: The combination of StudentID and Name is also a superkey. It is unique for each student, as no two students can have the same StudentID and Name.

3. **Superkey Example 3:**
   - K3 = {DateOfBirth, Department}
   - Explanation: The combination of DateOfBirth and Department can also be a superkey if it uniquely identifies each student. For example, if there are no two students with the same date of birth in the same department.

Key Points:

- A superkey may contain more attributes than necessary to uniquely identify each tuple.
- ***A superkey becomes a candidate key if it is minimal***, i.e., removing any attribute from the superkey would no longer uniquely identify tuples.
- ***A candidate key is a superkey that does not have any unnecessary attributes***.

In summary, a superkey is a concept used in database design to identify sets of attributes that provide a unique identification for each tuple in a relation. It forms the basis for identifying candidate keys and, ultimately, the primary key of a relation.

Find superkey and candidate key

Problem-01: Let R = (A, B, C, D, E, F) be a relation scheme with the following dependencies-
C → F
E → A
EC → D
A → B

Which of the following is a key for R?
CD
EC
AE
AC

Also, determine the total number of candidate keys and super keys.

Solution- We will find candidate keys of the given relation in the following steps-

Step-01: Determine all essential attributes of the given relation.
Essential attributes of the relation are- C and E. So, attributes C and E will definitely be a part of every candidate key.

Step-02: Now, We will check if the essential attributes together can determine all remaining non-essential attributes. To check, we find the closure of CE.
So, we have-
$\{ CE \}^+$
$= \{ C , E \}$
$= \{ C , E , F \}$ (Using $C \rightarrow F$)
$= \{ A , C , E , F \}$ (Using $E \rightarrow A$)
$= \{ A , C , D , E , F \}$ (Using $EC \rightarrow D$)
$= \{ A , B , C , D , E , F \}$ (Using $A \rightarrow B$)

Let's analyze the given set of functional dependencies for the relation scheme (R(E, F, G, H, I, J, K, L, M, N)):

$$(EF \rightarrow G)$$

$$(F \rightarrow I, J)$$

$$(EH \rightarrow KL)$$

$$(K \rightarrow M)$$

$$(L \rightarrow N)$$

Candidate Keys:

A candidate key is a minimal super key, and we can find candidate keys by considering the closure of different attribute sets.

Possible Super Keys:

1. {E, F, H}
2. {E, F, H, K, L}

3. {E, F, H, K, L, M, N}
4. {E, F, H, K, L, M, N, I, J, G}

After evaluating the closure of each of these, we can find that the only candidate key is {E, F, H} because it is a minimal super key, and no proper subset of it is a super key.

Super Keys:

A super key is any set of attributes whose closure contains all attributes of the relation. We already identified some possible super keys above.

Keys:

The candidate key {E, F, H} is also a key.

Closure of {E, F, H}:

$$[\{E, F, H\}^+ = \{E, F, H, K, L, M, N, I, J, G\}]$$

So, the set {E, F, H} is a candidate key.

This analysis indicates that {E, F, H} is the only candidate key for the given set of functional dependencies.

Normalization

The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations

Denormalization

The process of storing the join of higher normal form relations as a base relation—which is in a lower normal form

Candidate Key

When there are multiple SuperKeys the arbitrary primary key is generated it should contain a prime attribute other keys are called secondary keys. Candidate Key has unique values.

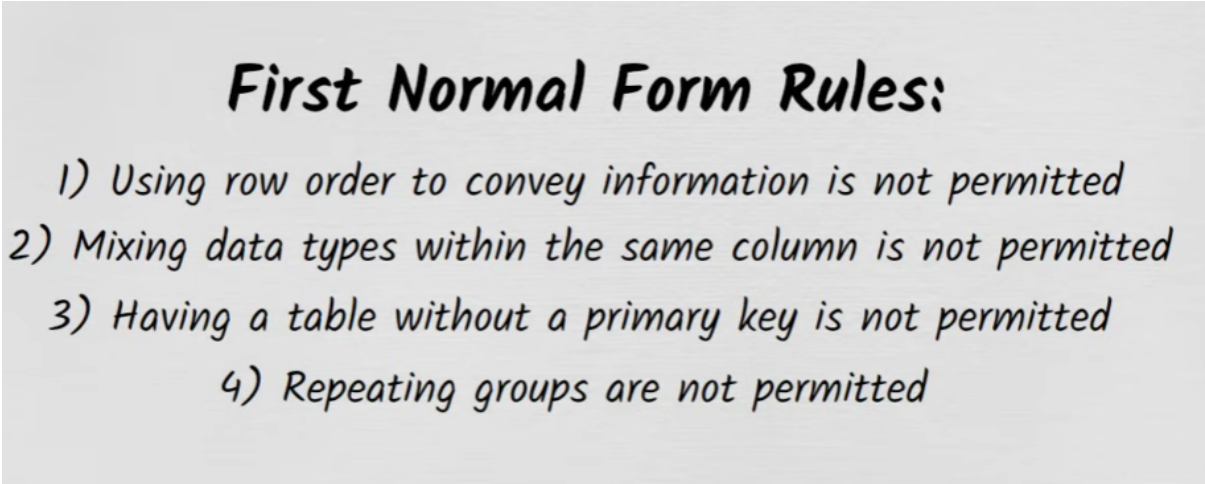
Any attribute involved in a candidate key is a prime attribute

First Normal Form

A relation is in 1NF if it contains only atomic (indivisible) values, and there are no repeating groups or arrays. Each column must have a single atomic value.

Disallows

- multivalued attributes
- composite attributes
- **nested relations**; attributes whose values for an *individual tuple* are non-atomic
- should have one primary key



Considered to be part of the definition of a relation

Most RDBMSs allow only those relations to be defined that are in First Normal Form

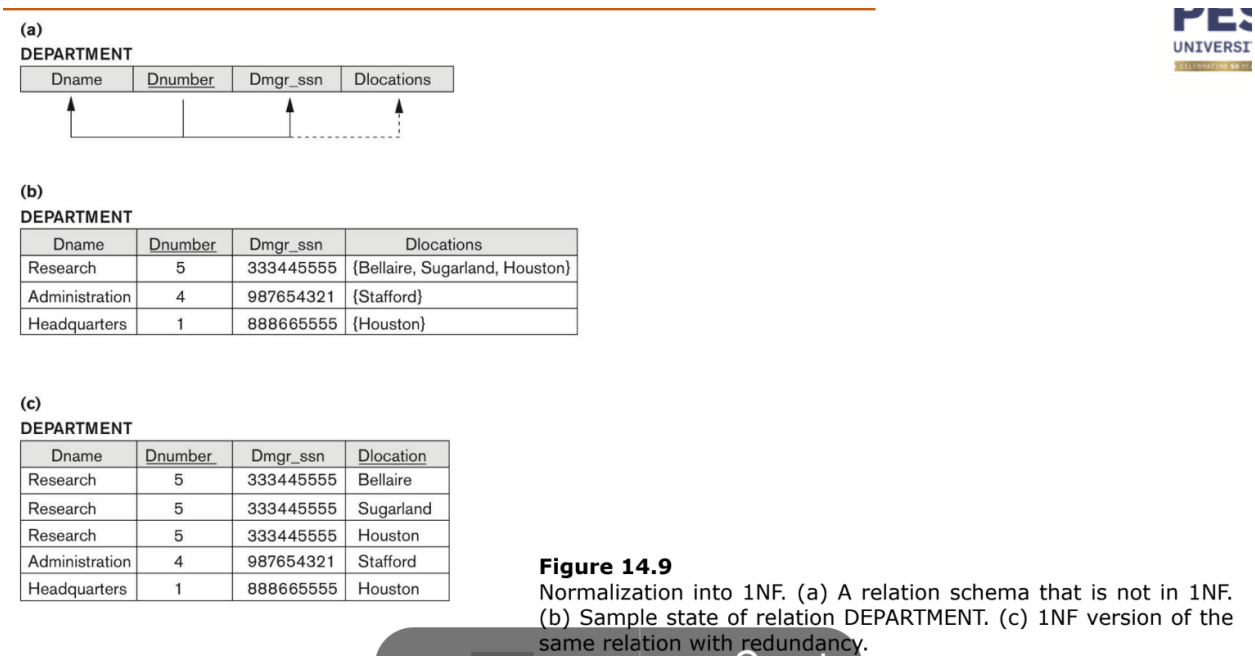
Example:
Consider a table
`Students` with the following columns:

```
| StudentID | Courses                |
|-----------|------------------------|
| 1         | Math, Physics, English |
| 2         | Chemistry, Biology     |
```

This table is not in 1NF because the `Courses` column contains a list of values. To normalize it to 1NF, we create a new table `StudentCourses`:

StudentID	Course
1	Math
1	Physics
1	English
2	Chemistry
2	Biology

Example



(a)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations

(b)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 14.9
Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

No multivalued attributes

Second Normal Form

A relation is in 2NF if it is in 1NF and all non-prime attributes are fully functionally dependent on the primary key.

A relation schema R is in second normal form (2NF) if every non-prime attribute A in R is fully functionally dependent and not partly on the candidate key R.

Whole {Primary Key } →{Secondary Key}

Example:

Consider a table `Orders`:

```
| OrderID | Product   | Category   | Price |
|---------|-----------|------------|-------|
| 1       | Laptop    | Electronics| 800   |
| 2       | Printer   | Electronics| 200   |
| 3       | T-shirt   | Clothing   | 25    |
```

The primary key is `{OrderID, Product}`. Category is dependent only on Product, not on the entire primary key. To normalize it to 2NF, we split it into two tables:

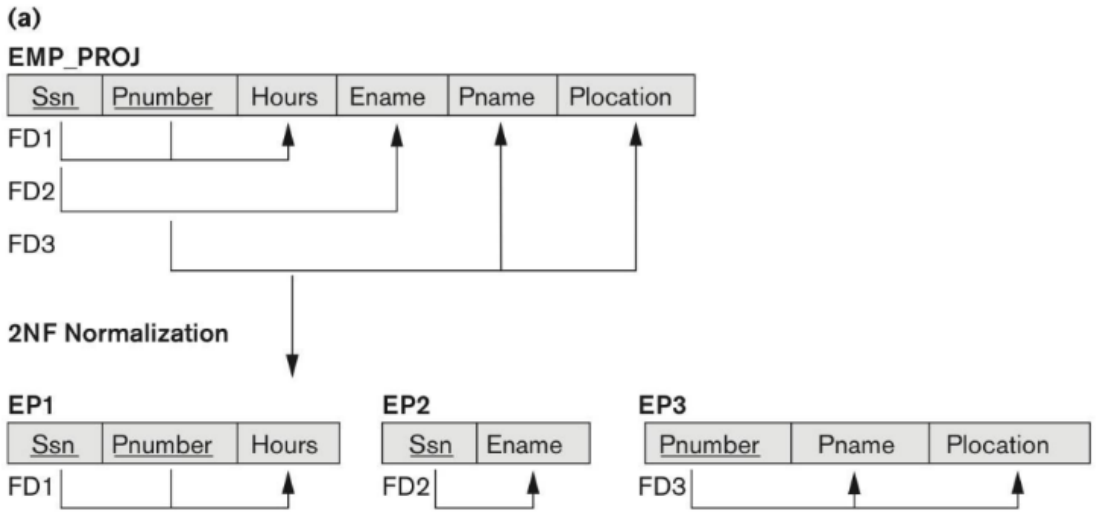
`OrderProducts`:

```
| OrderID | Product   | Price |
|---------|-----------|-------|
| 1       | Laptop    | 800   |
| 2       | Printer   | 200   |
| 3       | T-shirt   | 25    |
```

`ProductCategories`:

```
| Product   | Category    |
|-----------|-------------|
| Laptop    | Electronics|
| Printer   | Electronics|
| T-shirt   | Clothing    |
```

- Examples:
  - {SSN, PNUMBER} -> HOURS is a full FD since neither SSN -> HOURS nor PNUMBER -> HOURS hold
  - {SSN, PNUMBER} -> ENAME is not a full FD (it is called a partial dependency) since SSN -> ENAME also holds



Third Normal Form (3NF):

A relation is in 3NF if it is in 2NF, and no transitive dependencies exist. (It should not depend other non prime attributes)

Example:
Consider a table
`Employees`:

```
sqlCopy code
| EmployeeID | Department   | Location    |
|------------|--------------|-------------|
| 1          | Marketing    | New York    |
| 2          | IT           | San Francisco|
| 3          | HR           | New York    |
```

Here, `Location` is dependent on `Department`, and `Department` is dependent on `EmployeeID`. To normalize it to 3NF, we create two tables:

`EmployeeDepartments`:

```
luaCopy code
| EmployeeID | Department  |
|------------|-------------|
| 1          | Marketing   |
| 2          | IT          |
| 3          | HR          |
```

`DepartmentLocations`:

```
sqlCopy code
| Department  | Location     |
|-------------|--------------|
| Marketing   | New York      |
| IT          | San Francisco|
| HR          | New York      |
```

- Examples:
  - SSN -> DMGRSSN is a **transitive** FD
    - Since SSN -> DNUMBER and DNUMBER -> DMGRSSN hold
  - SSN -> ENAME is **non-transitive**
    - Since there is no set of attributes X where SSN -> X and X -> ENAME

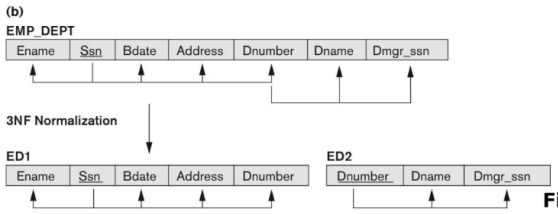


Figure 14.11
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

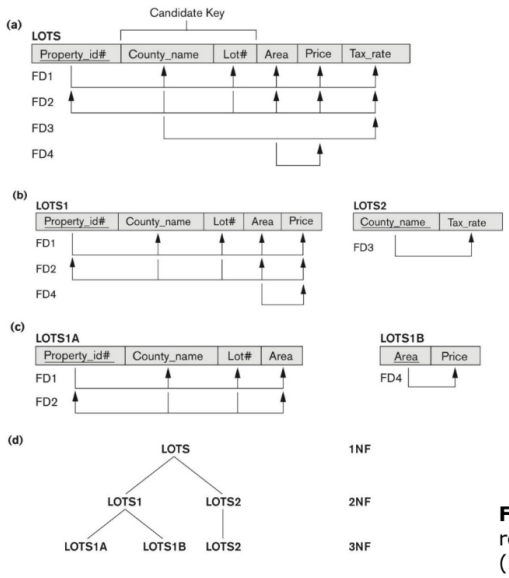
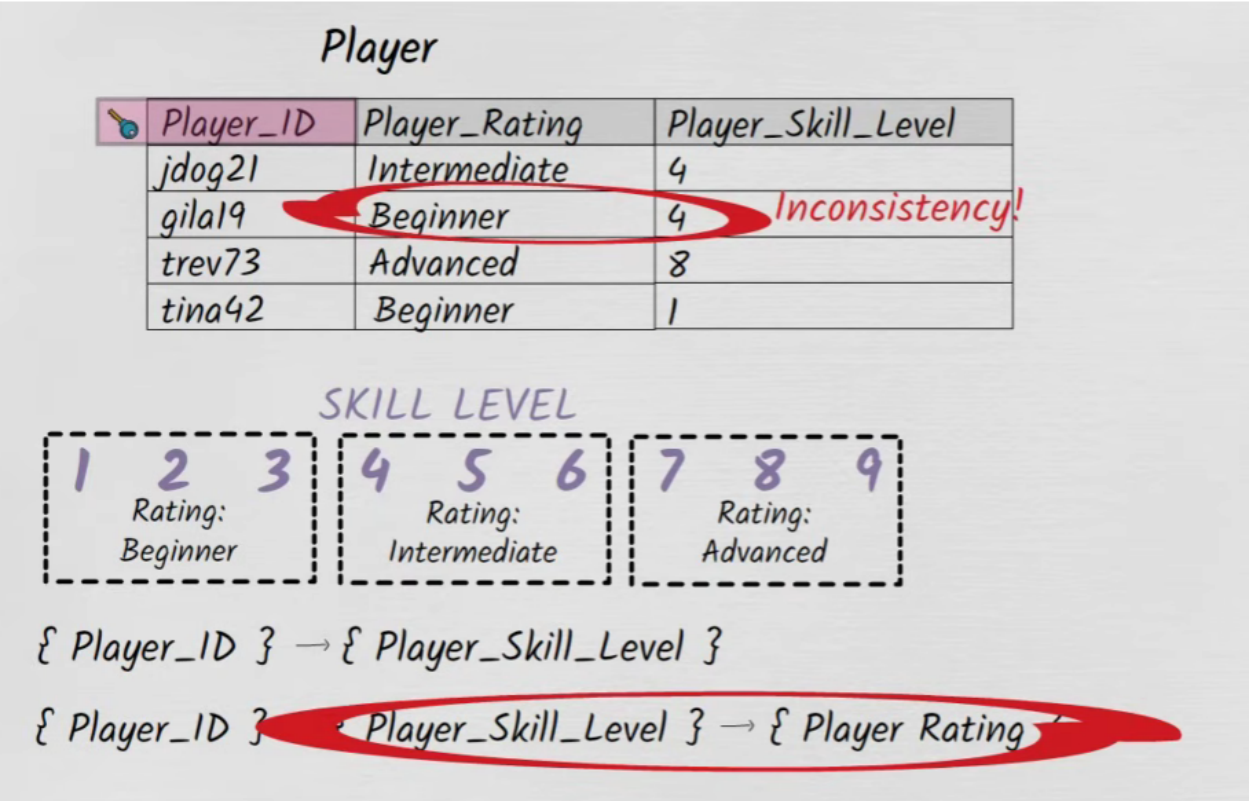


Figure 14.12 Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Progressive normalization of LOTS into a 3NF design.

In X → Y and Y → Z, with X as the primary key, we consider this a problem only if Y is not a candidate key.
When Y is a candidate key, there is no problem with the transitive dependency .

E.g., Consider EMP (SSN, Emp#, Salary).
Here, SSN → Emp# → Salary and Emp# is a candidate key.



CNF using Candidate Keys(Formal Definition)

1. Second Normal Form (2NF):

Simplified Definition:

In a table with multiple ways to uniquely identify rows (multiple candidate keys), we want to make sure that every piece of information (attribute) is fully dependent on all the different ways we can uniquely identify a row.

Example:

Imagine we have a table of properties with property ID, county name, lot number, area, and price. If the tax rate depends on the county name and not on the whole key (combination of property ID, county name, lot number), it violates 2NF. So, we split it into two tables - one with property details and another with county tax rates.

2. Third Normal Form (3NF):

Simplified Definition:

In a table with multiple ways to uniquely identify rows, we want to ensure that no information is indirectly dependent on a unique identifier. Either it directly depends on the unique identifier or it's a fundamental part of the unique identifier.

Example:

Think of a table with property details where the area determines the price, but the area isn't a super unique identifier. This violates 3NF because the price depends on something that's not directly a part of the unique identifier. To fix this, we might create a separate table where area and price are linked.

Additional Insight:

Alternative 3NF Definition:

We can also say a table is in 3NF if every non-prime attribute:

- Is fully dependent on every key of the table.
- Is not indirectly dependent on every key of the table

Boyce-Codd Normal Form (BCNF):

1. **Definition:**
    - A table is in BCNF if, for every functional dependency $(X \rightarrow A)$, (X) must be a superkey of the table. (With exception of trivial functional dependency)
2. **Strength of Normal Forms:**
    - Each normal form is stricter than the one before it. Going from 1NF to BCNF, the rules become more rigorous.
3. **Hierarchy:**
    - Every 2NF table is also in 1NF.
    - Every 3NF table is also in 2NF.
    - Every BCNF table is also in 3NF.
4. **Strength of BCNF:**

- BCNF is considered stronger than 3NF. It imposes stricter rules on how dependencies are handled.

5. **Goal of Normalization:**
   - The ultimate goal is to have each table in BCNF (or at least in 3NF). This ensures a high level of data organization and reduces the risk of anomalies.

Figure 14.14 A relation TEACH that is in 3NF but not in BCNF

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

Figure 14.14
A relation TEACH that is in 3NF but not BCNF.

- Two FDs exist in the relation TEACH:
  - fd1: { student, course} -> instructor
  - fd2: instructor -> course
- {student, course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 14.13 (b).
  - So this relation is in 3NF *but not in* BCNF
- A relation **NOT** in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations.

- Three possible decompositions for relation TEACH
  - D1: {student, instructor} and {student, course}
  - D2: {course, instructor } and {course, student}
  - D3: {instructor, course } and {instructor, student} ✔
- All three decompositions will lose fd1.
  - We have to settle for sacrificing the functional dependency preservation. But we cannot sacrifice the non-additivity property after decomposition.
- Out of the above three, only the 3rd decomposition will not generate spurious tuples after join.(and hence has the non-additivity property).

Testing for Non-Additivity of Binary Relational Decompositions:

1. **Binary Decomposition:**
   - It's like splitting a table into two smaller tables.
2. **NJB Property (Lossless Join):**
   - When we split a table, we want to make sure we can put it back together without losing any information.

- **Testing Binary Decompositions for Lossless Join (Non-additive Join) Property**
  - **Binary Decomposition:** Decomposition of a relation R into two relations.
  - **PROPERTY NJB (non-additive join test for binary decompositions):** A decomposition D = {R1, R2} of R has the lossless join property with respect to a set of functional dependencies F on R *if and only if* either
    - The f.d. ((R1 ∩ R2) → (R1- R2)) is in F^+, or
    - The f.d. ((R1 ∩ R2) → (R2 - R1)) is in F^+.

1. **NJB Test:**
   - We have a test to check if our split (decomposition) is good. It involves some fancy functional dependencies (f.d.), but the key idea is to ensure that certain

dependencies hold.

2. **Example:**
    - Imagine a table about teaching relationships (TEACH). We try different splits (D1, D2, D3) and check if certain dependencies (like Instructor → Course) still hold. If they do, our split is good.

• A relation with only trivial functional dependencies is always in BCNF. In other words, a relation with no non-trivial functional dependencies is always in BCNF.
• BCNF decomposition is always lossless but not always dependency preserving.
Sometimes, going for BCNF may not preserve functional dependencies. So, go for BCNF only if the lost functional dependencies are not required else normalize till 3NF only.

Achieving BCNF:

1. **BCNF Definition:**
    - BCNF is like a higher standard of organization for tables. It ensures minimal redundancies and handles dependencies well.
2. **Procedure for BCNF:**
    - If a table isn't in BCNF, we do a smart split. For example, if Instructor → Course causes trouble, we split into (TEACH - COURSE) and (Instructor u Course). Repeat if needed.
3. **Important Points about BCNF:**
    - Every simple table with only two attributes is always in BCNF.
    - BCNF is great for cutting out redundancies but might not preserve all dependencies.
    - Going beyond BCNF (like 4NF) is possible but often not necessary in practical database systems.
4. **Considerations:**
    - Sometimes, pushing for BCNF may lose some dependencies, so we should balance between normalization and preserving required information.

Multivalued Dependencies and Fourth Normal Form (4)

Multivalued Dependency (MVD) Definition:

- An MVD $X \rightarrow\!\!> Y$ on a relation schema R means that if two tuples $t1$ and $t2$ share the same value for X, then there exist two additional tuples $t3$ and $t4$ with specific properties related to X, Y, and the remaining attributes.

Trivial MVD:

- An MVD is trivial if either Y is a subset of X or the union of X and Y covers all attributes in R.

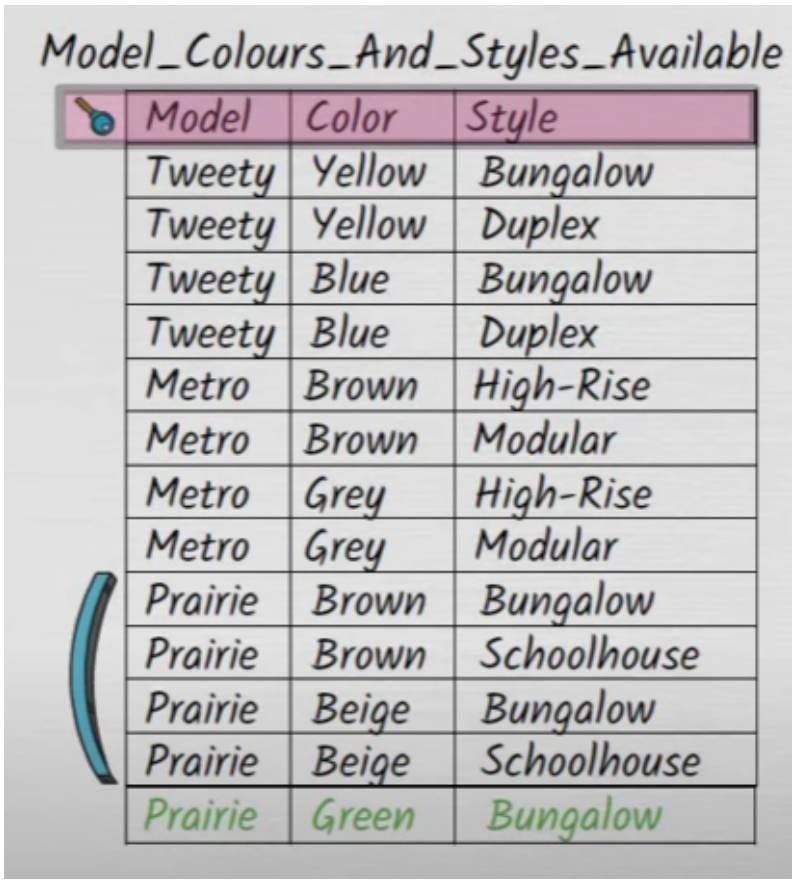
Fourth Normal Form (4NF) Definition:

- A relation schema R is in 4NF with respect to a set of dependencies F if, for every nontrivial MVD $X \rightarrow\!\!> Y$ in F+, X is a superkey for R.

Only the Primary Key can Have Multivalued Dependency

If there exists a table with mutilple Keys with multivalued attributes

Just do decomposition so there is a primary key with MVD secondary key



Key Points:

- An all-key relation is always in BCNF (no functional dependencies).

- A relation without functional dependencies but with non-trivial MVDs may not be in 4NF.
- If a relation violates 4NF due to non-trivial MVD, decomposition is required to eliminate redundancy caused by the MVD.

Example - Fourth Normal Form (4NF):

Consider the EMP relation with two MVDs: *Ename*→>>*Pname* and *Ename*→>>*Dname*. The decomposition into EMP_PROJECTS and EMP_DEPENDENTS resolves the 4NF violation.

Join Dependencies and Fifth Normal Form

Project JOIN Normal Form

Join Dependency (JD) Definition:

- A JD, denoted by JD(R1, R2, ..., Rn), specifies a constraint on relation schema R, stating that every legal state r of R should have a non-additive join decomposition into R1, R2, ..., Rn.

Trivial JD:

- A JD is trivial if one of the relation schemas *Ri* in JD(R1, R2, ..., Rn) is equal to R.

Fifth Normal Form (5NF) Definition:

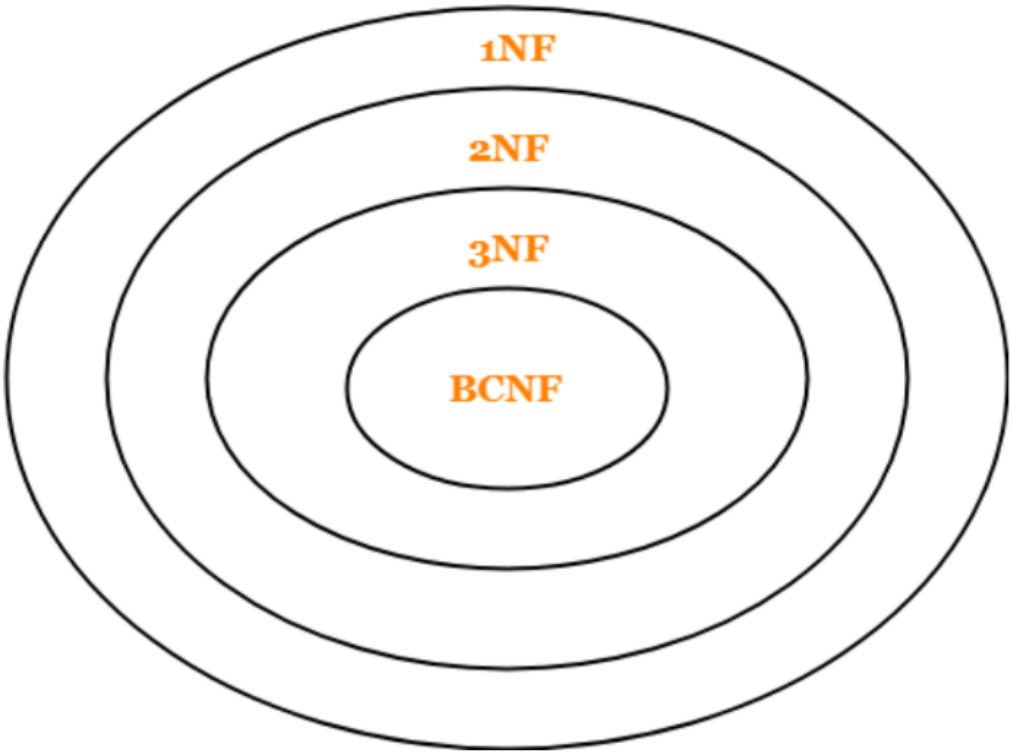
- A relation schema R is in 5NF (or PJNF) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD(R1, R2, ..., Rn) in F+ (implied by F), every *Ri* is a superkey of R.

Key Points:

- JDs are broader than MVDs; an MVD is a special case of a JD with $n=2$.
- A JD is trivial if one of the relation schemas in it is equal to R.
- 5NF is rarely used in practice due to the difficulty of discovering join dependencies in large databases.

Summary

- **Informal Design Guidelines:** Start checking from BCNF, moving towards outer circles (3NF, 2NF, 1NF).
- **Functional Dependencies (FDs):** Define relationships between attributes.
- **Normal Forms:** Progress from 1NF to BCNF ensures proper organization, minimizing redundancy.
- **Fourth and Fifth Normal Forms:** Address more complex dependencies (MVDs, JDs) for advanced database organization.



1NF>2NF>3NF>BCNF>MVD(4NF)>JD(5NF)

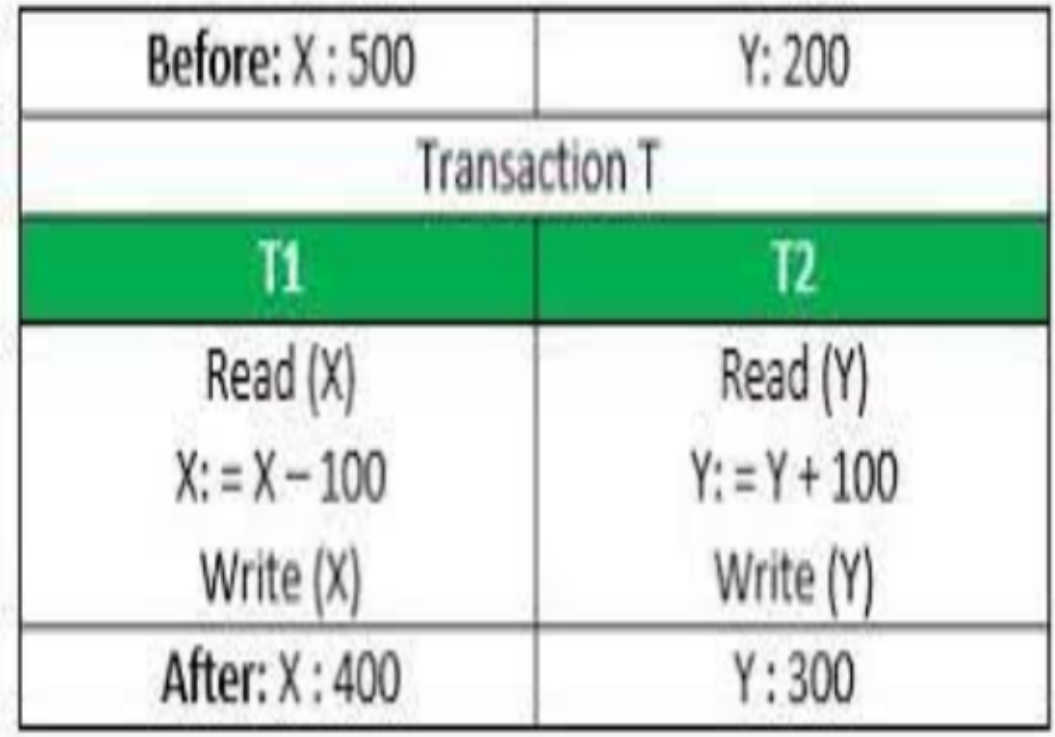
Therefore, 5NF builds upon the concepts of 4NF by additionally ensuring that every relation resulting from a join dependency is a superkey. In essence, 5NF includes the conditions of 4NF and extends the normalization further to handle certain types of join dependencies.

Transaction

Collection of operations that form single logical unit of work is called Transaction

Example: Transferring funds from a checking account to a savings account is considered a single operation from the customer's perspective. However, within the database system, it actually involves multiple operations.

In the database side the information is read(fetched) from database to RAM the operation are executed on the data and stored in RAM it is commited back to the database afterwards.



Steps in Transaction

1. **Definition of a Transaction:**
    - A transaction is a unit of program execution that performs a series of operations on data items.
    - These operations may include reading and updating various data items within a database.
2. **Initiation of a Transaction:**

- A transaction is typically initiated by a user program.
- The user program is often written in a high-level data-manipulation language, such as SQL, or in a general-purpose programming language like C++, Java, etc.
- Embedded database accesses can be made using technologies like JDBC (Java Database Connectivity) or ODBC (Open Database Connectivity).

3. **Example Transaction - Money Transfer:**
    - The example you provided illustrates a transaction that involves transferring $50 from one account (A) to another account (B).
    - The transaction comprises the following steps:
        1. **Read(A):** Retrieve the current balance of account A.
        2. **A := A – 50:** Subtract $50 from the balance of account A.
        3. **Write(A):** Update the database with the new balance of account A.
        4. **Read(B):** Retrieve the current balance of account B.
        5. **B := B + 50:** Add $50 to the balance of account B.
        6. **Write(B):** Update the database with the new balance of account B.

Transaction Boundaries:

- Transactions are delimited by statements (or function calls) of the form `begin transaction` and `end transaction`.
- All operations between the `begin transaction` and `end transaction` statements are considered part of a single transaction.

Read-Only and Read-Write Transactions:

- A transaction can be categorized as:
    - **Read-Only Transaction:** If the database operations in a transaction do not update the database.
    - **Read-Write Transaction:** If the database operations in a transaction involve both retrieval and updates.

Operations in Transactions:

- Transactions access data using two fundamental operations:

- `read(X)`: Transfers the data item X from the database to a variable (also called X) in a buffer in main memory belonging to the transaction.
- `write(X)`: Transfers the value in the variable X in the main-memory buffer of the transaction to the data item X in the database.

Immediate Database Updates:

- In the context presented, the `write` operation is assumed to update the database immediately.

SQL Support for Transactions:

- SQL provides support for the development of transaction-based applications.
- No explicit `begin transaction` statement is needed in SQL; transactions are implicitly managed.
- Every transaction must have an explicit `end` statement, which can be either `COMMIT` (to commit the changes) or `ROLLBACK` (to undo the changes).

Access Modes:

- Access modes for transactions include `READ ONLY` or `READ WRITE`, depending on whether the transaction involves updates to the database.

ACID Properties

- Transactions in a database system must maintain ACID properties:
  - **Atomicity: Transactions are treated as atomic units, either fully completed or fully rolled back.**
  - **Consistency: A transaction brings the database from one valid state to another.**
  - **Isolation: Concurrent execution of transactions does not interfere with each other.**
  - **Durability: Once a transaction is committed, its effects are permanent.**

Example for ACID

Transaction to transfer $50 from account A to account B: Transaction (Ti)

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

1. **Atomicity Requirement:**
   - If a transaction fails after some of its steps but before completion, it can lead to money being "lost" and an inconsistent database state.
   - **The atomicity property ensures that either all actions of the transaction are reflected in the database, or none are.**
   - The database system maintains a log to track the old values of any data on which a transaction performs a write. If the transaction does not complete, the system restores the old values from the log.
2. **Durability Requirement:**
   - **Once the user is notified that the transaction has completed, the updates to the database must persist even in the face of software or hardware failures.**
   - The recovery system of the database is responsible for ensuring durability.
3. **Consistency Requirement:**
   - The sum of A and B should remain unchanged by the execution of the transaction.
   - Consistency requirements include explicitly specified integrity constraints (e.g., primary keys and foreign keys) and implicit integrity constraints.
   - **The database must be consistent after a transaction completes successfully.**
4. **Isolation Requirement:**

- If another transaction (T2) accesses the partially updated database between steps 3 and 6 of the original transaction (T1), it may see an inconsistent database state.
- **Isolation ensures that the concurrent execution of transactions results in a system state equivalent to a state that could have been obtained if the transactions executed one at a time in some order.**
- Ensuring isolation is the responsibility of the concurrency-control system in the database.

Concurrent Execution of Transactions:

- Concurrent execution of transactions provides significant performance benefits.
- The isolation property ensures that concurrent execution does not lead to inconsistent states.
- Solutions, including concurrency-control systems, have been developed to allow multiple transactions to execute concurrently while maintaining isolation

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are. This "all-or-none" property is referred to as atomicity.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability.** After a transaction completes successfully, the changes it has made to the database must persist, even if there are system failures.

Storage Structure

- Volatile storage: Information residing in volatile storage does not usually survive system
  crashes. Access to volatile storage is extremely fast.
  Examples: main memory and cache memory.
- Non-volatile storage: Information residing in non-volatile storage survives system crashes.

Non-volatile storage is slower than volatile storage, particularly for random access.
Examples: magnetic disk and flash storage(for online storage) and optical media and magnetic
tapes(for archival storage)

- Stable storage: Information residing in stable storage is never lost. Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely.

  To implement stable storage, we replicate the information in several non-volatile storage media (usually disk) with independent failure modes.

Aborted - Unsucccessful transaction

Roll Back - Changes made by an aborted transaction is undone

Log - Maintains record of every database transaction

compensating transaction - To roll back a committed transaction

The System log

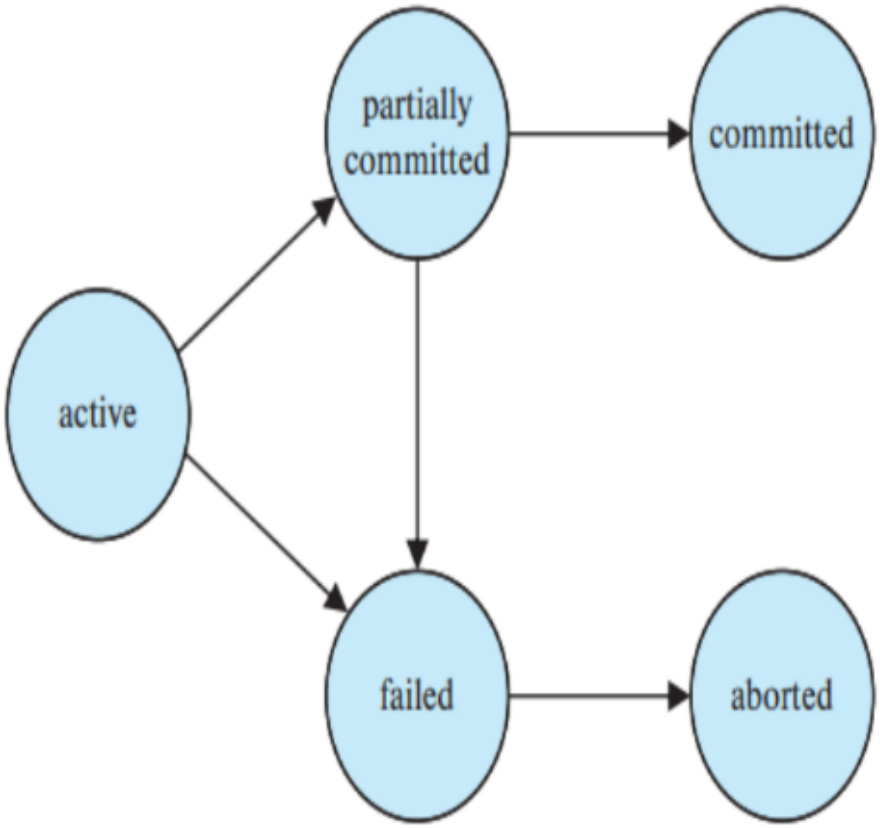
- System log keeps track of transaction operations
- Sequential, append-only file
- Not affected by failure (except disk or catastrophic failure)
- Log buffer
- Log file is backed up periodically
- Undo and redo operations based on log possible

shadow-database scheme

- A transaction that wants to update the database first creates a complete copy of the database.
- All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched.
- If at any point the transaction has to be aborted, the system merely deletes the new

Transaction States

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



A transaction is said to have terminated if it has either committed or aborted.

When transaction finishes its final statement, it enters the partially committed state.

It is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may stop its successful completion.

Transaction in Aborted state can restart or kill the transaction.

Schedules

Schedule – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed

A schedule for a set of transactions must consist of all instructions of those transactions

Must preserve the order in which the instructions appear in each individual transaction.

However, there are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization**
- **Reduced waiting time**.

Scheduling Types

Serial Scheduling

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

Concurrent scheduling

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

Inconsistent scheduling

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

Serialization

If a given non-serial schedule of ‘n’ transactions is equivalent to some serial schedule of ‘n’ transactions, then it is called as a serializable schedule.

Serial Schedules	Serializable Schedules
No concurrency => all the transactions necessarily execute serially one after the other.	**Concurrency =>** multiple transactions can execute concurrently.
Serial schedules lead to **less resource utilization and CPU throughput.**	Serializable schedules **improve both resource utilization and CPU throughput.**
Serial Schedules are **less efficient** as compared to serializable schedules.	Serializable Schedules are always **better** than serial schedules.

We assume that, between a read(Q) instruction and a write(Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction.

- **Conflict Serializability:** Ensures that transactions do not conflict with each other. Conflicts arise when two transactions access the same data item, and at least one of them is a write.
- **View Serializability:** Focuses on the visibility of the database state to transactions. Two schedules are view-equivalent if they produce the same final database state when executed.

Let us consider a schedule S in which there are two consecutive instructions, I and J, of transactions
Ti and Tj , respectively (i ≠ j). If I and J refer to different data items, then we can swap I and J without
affecting the results of any instruction in the schedule.

I and J refer to the same data item Q, then the order of the two steps may matter.

1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
3. I_i = **write**(Q), I_j = **read**(Q). They conflict
4. I_i = **write**(Q), I_j = **write**(Q). They conflict

Conflict Serializability

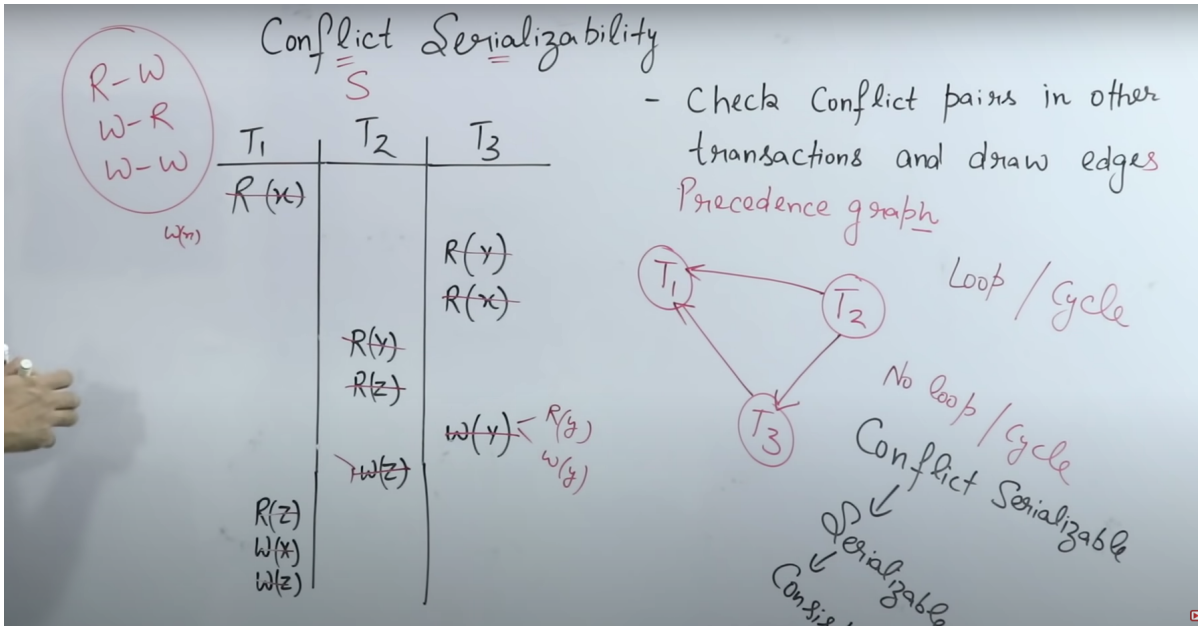
Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

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

Schedule 3

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

Schedule 6



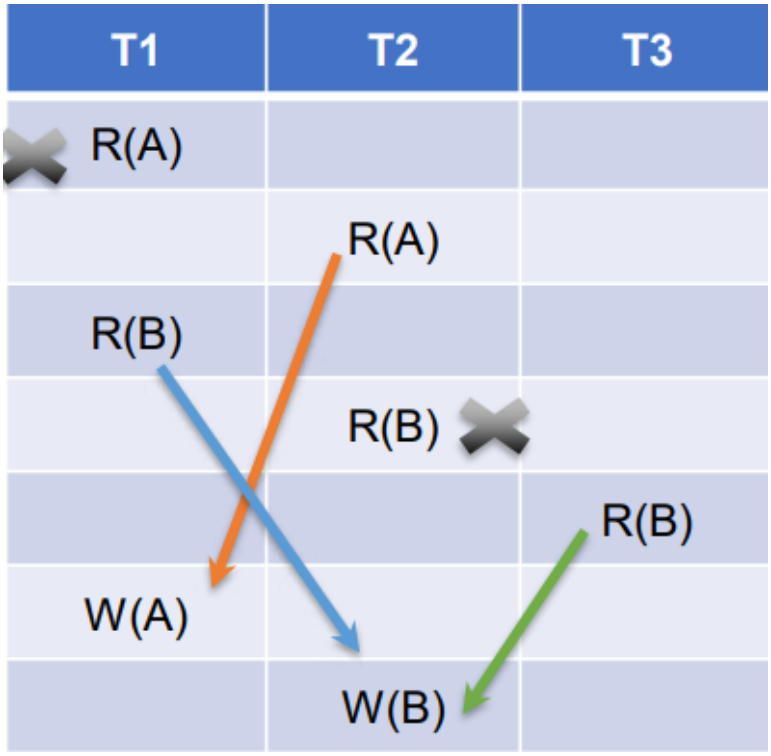
See wether there is r-w w-r or w-w conflict between T1 T2 T3 if cycle can't be made - serializable

To find the serial root check the indegree of the graph least indegree first
t2→t3→t1

When conflict serializable gives loop check for view serializability

Two operations are called as conflicting operations if
all the following conditions hold true for them-

Both the operations belong to different transactions
Both the operations are on the same data item
At least one of the two operations is a write operation



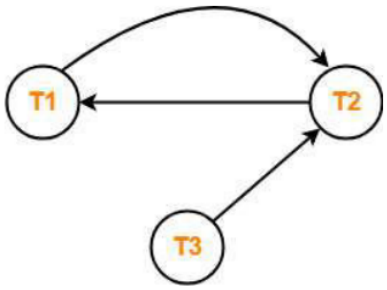
List all the conflicting operations and determine the dependency between the transactions-

$R_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
$R_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
$R_3(B)$, $W_2(B)$ $(T_3 \rightarrow T_2)$

Solution-

Step-02:

Draw the precedence graph-



$(T_2 \rightarrow T_1)$
$(T_1 \rightarrow T_2)$
$(T_3 \rightarrow T_2)$

- Clearly, there **exists a cycle** in the precedence graph.
- Therefore, the given schedule **S is not conflict serializable.**

View Serializability

Two schedules are said to be view equivalent if the order of initial read, final write and update operations is the same in both the schedules.

Consider two schedules S1 and S2 each consisting of two transactions T1 and T2.

Schedules S1 and S2 are called view equivalent if the following three conditions hold true for them-

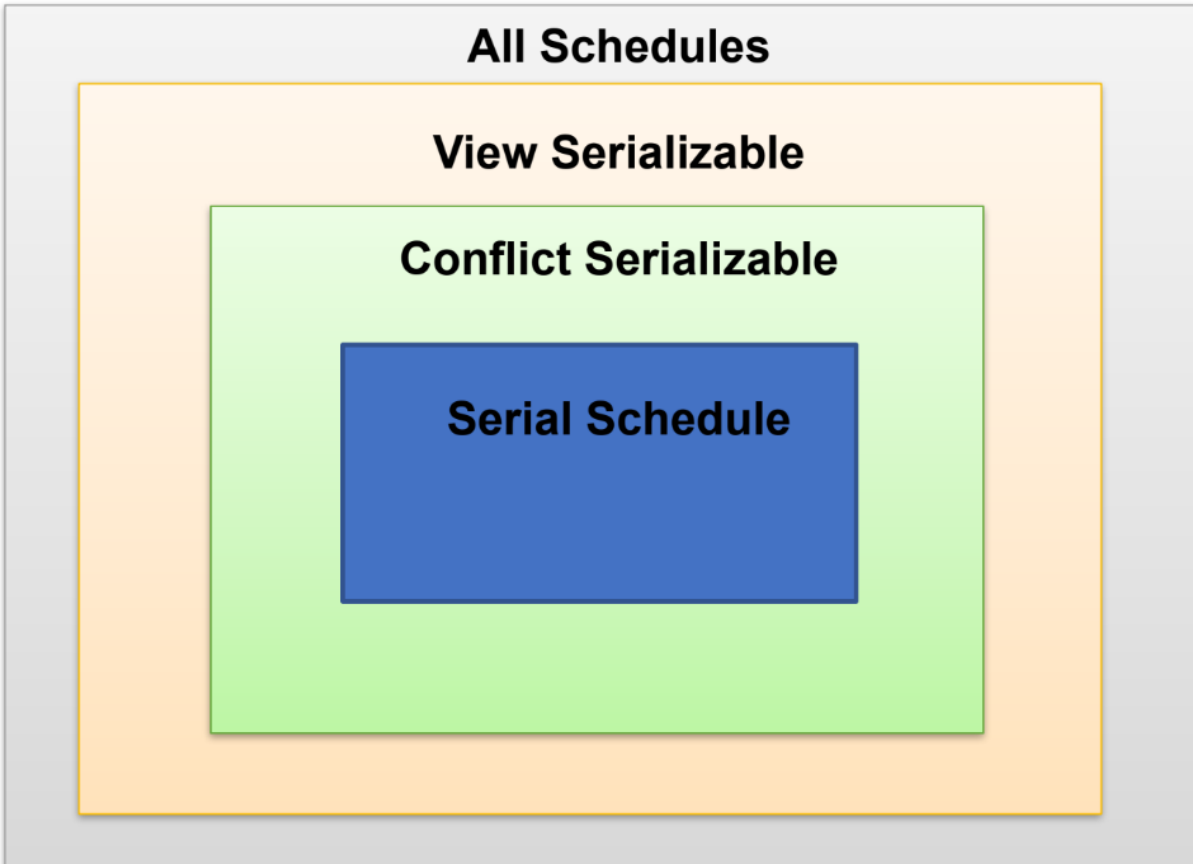
1. "**Initial readers** must be same for all the data items".
2. "**Final writers** must be same for all the data items".
3. "**Write-read** sequence must be same."

Every conflict serializable schedule is also view serializable.

Blind Write :- No read prior to the first write

- **$W_3(X)$** is a blind write, as there is no read before write [$R_3(X)$ before $W_3(X)$]
- **$W_2(X)$** is not a blind write, as a read happens before write [$R_2(X)$ before $W_2(X)$]

T1	T2	T3
	$R_2(X)$	
$R_1(X)$		
		$W_3(X)$
	$W_2(X)$	



Transaction Isolation and Atomicity

If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction

We need to place restrictions on the types of schedules permitted in the system.

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

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

- If T_6 should abort, T_7 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

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

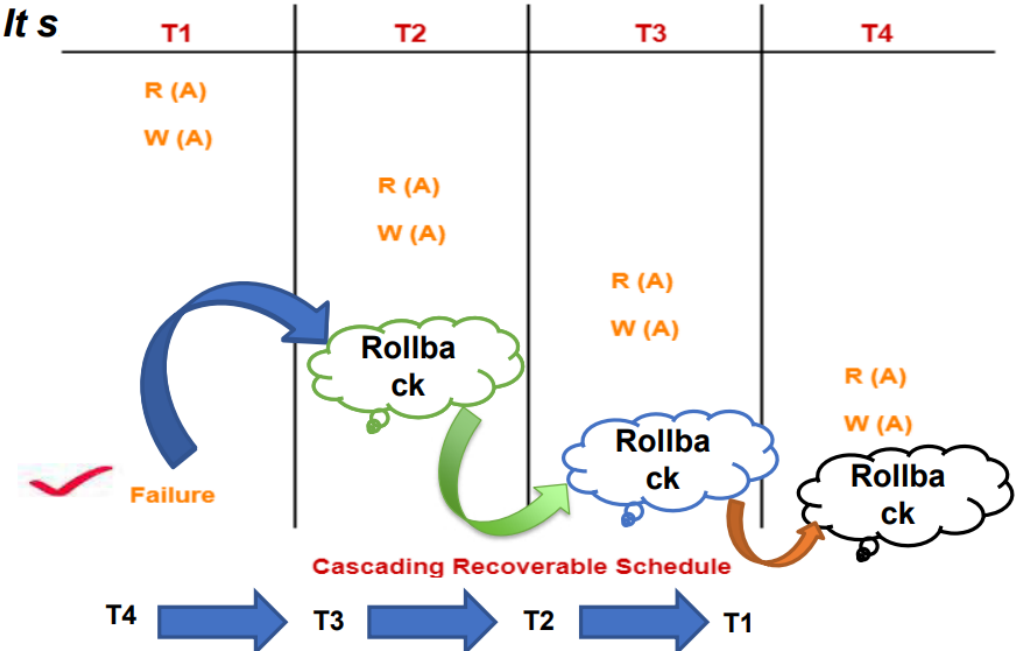
T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

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

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

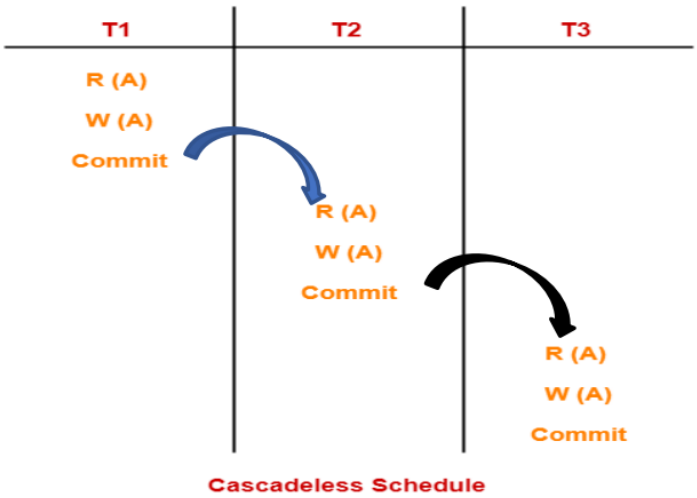
If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a **Cascading Schedule** or **Cascading Rollback** or **Cascading Abort**.

It s



- If in a schedule, a transaction is **not allowed to read a data** item until the last transaction that has written **it is committed** or aborted, then such a schedule is called as a **Cascadeless Schedule**.
- Cascadeless schedule allows only committed read operations.
- It avoids cascading roll back and thus saves CPU time.

Example-



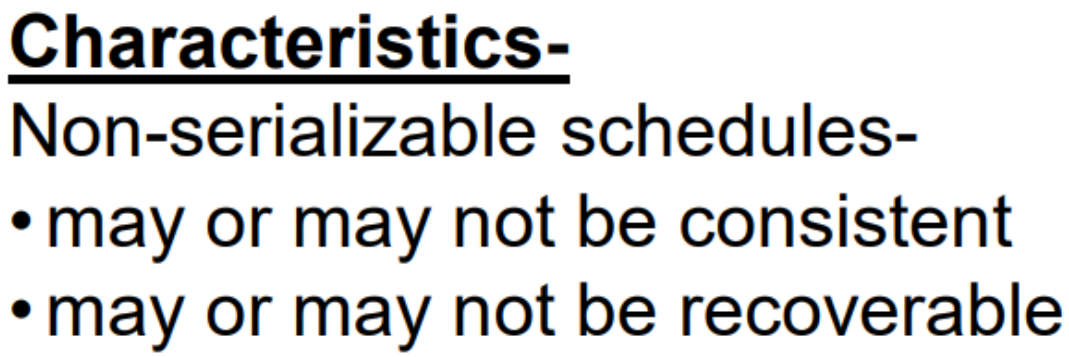
Cascadeless schedule allows only committed read operations.

However, it allows uncommitted write operations

Strict Schedule

If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written is committed or aborted, then such a schedule is called as a Strict Schedule.

- Strict schedule allows only committed read and write operations.
- Strict schedule implements more restrictions than cascadeless schedule



If in a schedule, a transaction performs a dirty read operation from an uncommitted transaction and commits before the transaction from which it has read the value then such a schedule is known as an Irrecoverable Schedule.

T1	T2
Read(A)	
Write(A)	
-	Read(A) ///Dirty Read
-	Write(A)
-	Commit
-	

How to check weather a trasaction is recoverable or not

- Check for conflict Serializibility - If no go for next one
- Check for Dirty read - If no go for next one
- Check for sequence of commits :- Sequence should match the sequence of dirty read if not its unrecoverable

T1	T2	T3
R(X)		
	R(X)	
R(Z)		
		R(X)
		R(Y)
W(X)		
		W(Y)
	R(Y)	
	W(Z)	
	W(Y)	
C		
	C	
		C

Consider a schedule S having 3 transactions.

Check weather it is Recoverable or not?

1. Check for conflict serializibilty: No (cycle exists T1 and T2)
2. Dirty Read : Yes (T3 to T2)
3. *Check the sequence of commit*

a) *T1 -> T2 -> T3*

4. **Check the sequence of Dirty Read**

a) T3 -> T2

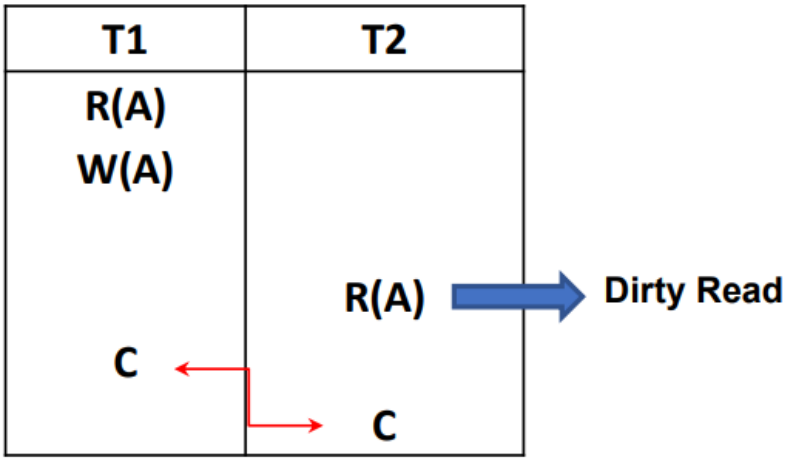
Order does not match.

Hence, its not recoverable.

If in a schedule, A transaction performs a dirty read operation from an uncommitted transaction and its commit operation is delayed till the uncommitted transaction either commits or roll backs then such a schedule is called as a Recoverable Schedule.

T_1	T_2
R(A)	
W(A)	
	R(A)
commit	
	commit

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.



This is a **Recoverable Schedule**
T1 commits before T2,
That makes the value read by T2 correct.

Concurrency control

To ensure the preservation of the isolation property, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called concurrency-control schemes.

Example for inconsistent value

Example :

Time/Transactions	T1	T2
1	Read(X)	
2	X=X-10	
3	Write(X)	
4		Read(X)
5	Rollback	

Inconsistent value of X in Transactions T1 and T2.

Lock based protocol

A transaction can access a data item only if it holds a lock on that item. Transactions acquire locks on the entire database before starting and release them after committing. This locking policy allows only one transaction to execute at a time, while others must wait for the lock to be released.

HAS POOR PERFORMANCE

XS Locking Protocol

Data items can be locked in two modes :

Exclusive (X) mode : Data item can be both read as well as written.

Shared (S) mode. Data item can only be read.

Lock-compatibility matrix

This function is used to define whether a pair of lock modes can peacefully coexist on a shared resource, or if granting a lock of one mode could potentially block another lock of a different mode.

	S	X
S	true	false
X	false	false

Shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously.

The transaction makes a request to the **concurrency-control manager**.

Concurrency-control manager grants the lock to the transaction.

To access a data item, transaction T must first lock that item.

Problem with XS :-

- Maybe irrecoverable
- maybe non serializable
- deadlock
- starvation

Deadlock

Deadlock

- Consider transactions T3 and T4 where they are transactions T1 and T2 respectively with the unlocking delayed.
- Neither T3 or T4 can make progress — executing lock-S(B) causes T4 to wait for T3 to release its lock on B, while executing lock-X(A) causes T3 to wait for T4 to release its lock on A.
- Such a situation is called a **deadlock**.
- One of the two transactions must rollback.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

2-Phase Locking

They are always serializable

Two-phase locking does not ensure freedom from deadlocks.

Cascading rollback

Phase 1: Growing Phase

Transaction may obtain locks
Transaction may not release locks

can convert a lock-S to a lock-X (Shared to Exclusive) - upgrade

Phase 2: Shrinking Phase

Transaction may release locks
Transaction may not obtain locks

can convert a lock-X to a lock-S (Exclusive to Shared) - downgrade

- **Lock Point** - point where a transaction acquired its final lock - end of growing phase

- **Strict two-phase locking** a transaction must hold all its exclusive locks till it commits/aborts.
  Ensures recoverability and avoids cascading roll-backs

  Rigorous two-phase locking: a transaction must hold all locks till commit/abort. Transactions can be serialized in the order in which they commit.
- The lock manager replies to a lock request by sending a lock grant messages
- Each record of the linked list for a data item, notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.

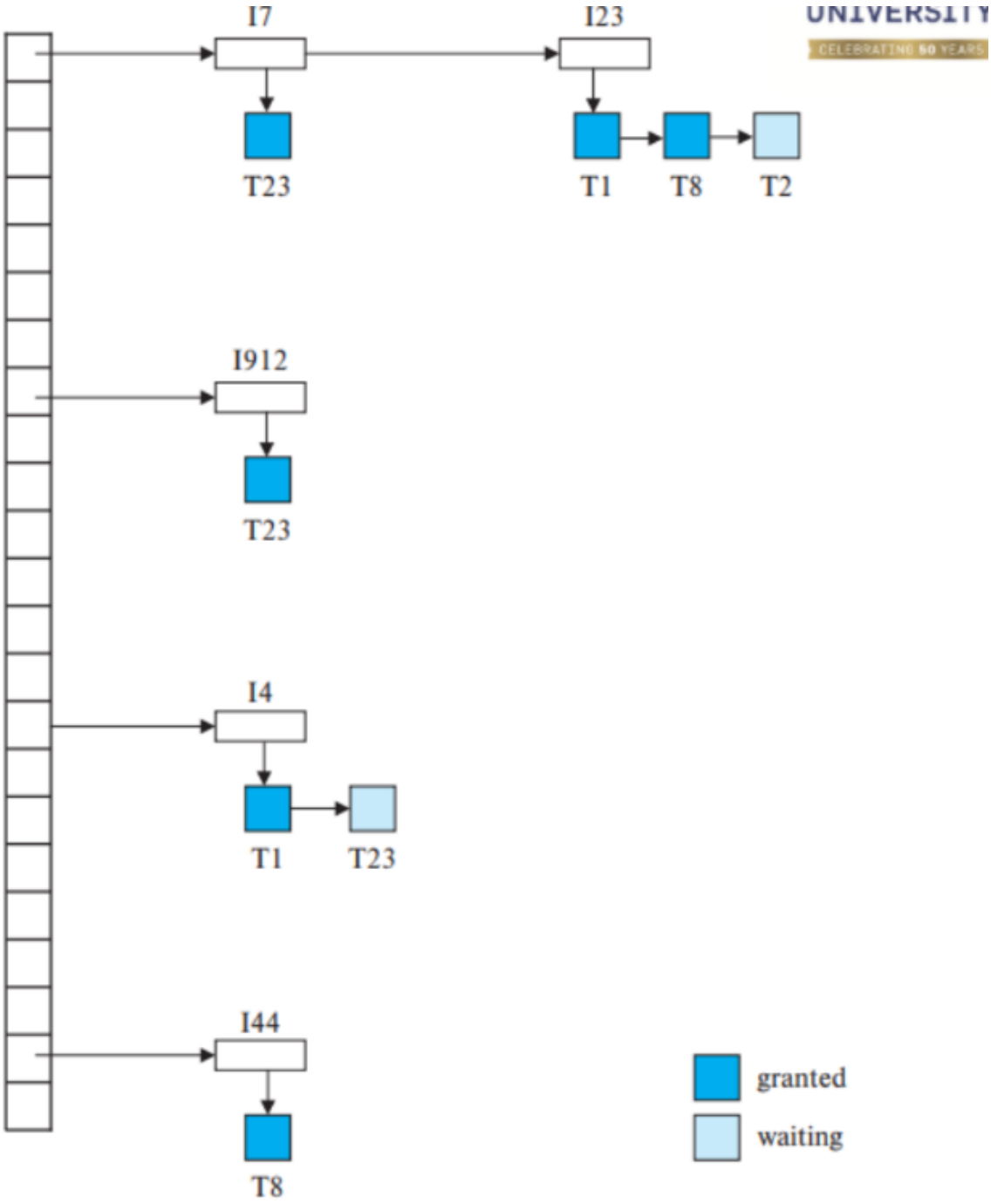
	Strict two-phase locking	Rigorous two-phase locking
Lock Release	Locks are released only at the end of the transaction.	Locks are released as soon as they are no longer needed for a specific data item. Transaction can release some locks before completing its execution.
Deadlock Prevention	Can lead to potential deadlocks if not managed properly.	Reduces the chances of deadlocks by allowing lock release and reacquisition.
Resource Utilization	Can lead to underutilization of resources, as locks are held till the end.	Improves resource utilization by allowing other transactions to use released locks.
Transaction Starvation	Can lead to transaction starvation due to lock holding till the end.	Reduces the likelihood of transaction starvation by allowing lock reacquisition during execution.

Strict 2PL ⇒ Cascadeless, Recoverable
Rigorous 2PL ⇒ Cascadeless, Recoverable, Starvation free

Lock Table

The lock manager maintains an in-memory data-structure called a lock table to record granted locks and pending requests.

Uses Linked List



- Dark rectangles indicate granted locks, light colored ones indicate waiting requests
- Lock table also records the type of lock granted or requested.
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks

- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted.
- It can be seen in the fig. that T23 has been granted locks on I912 and I7 and is waiting for a lock on I4.
- **This algorithm guarantees freedom from starvation**

Deadlock

Prevention Stratergies

- Require that each transaction locks all its data items before it begins execution (predeclaration).
- Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
- performs transaction rollback instead of waiting for a lock whenever the wait could
  potentially result in a deadlock.

Wait-Die(Non-Premptive)

Non-preemptive: The older transaction is allowed to wait, and the younger one is rolled back.

Wound-Wait(Premptive)

Preemptive: The younger transaction is allowed to wait, and the older one is rolled back.

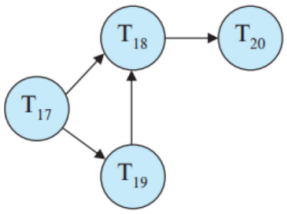
Starvation is thus avoided

Deadlock Detection

- If deadlocks occur frequently, then the detection algorithm should be invoked more frequently.

- Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken.
- In addition, the number of cycles in the graph may also grow. In the worst case, we would
  invoke the detection algorithm every time a request for allocation could not be granted
  immediately

- Transaction T17 is waiting for transactions T18 and T1
- Transaction T19 is waiting for transaction T18.
- Transaction T18 is waiting for transaction T20.
- Since the graph does not contain any cycles, there is no deadlocks



Deadlock Recovery

Deadlock Recovery

On detection of deadlock, three actions have to be taken :

- Select **victim**
  - Select that transaction as victim that will incur **minimum cost** when rollbacked.
  - Many factors may determine the cost of a rollback, including:
    - How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
    - How many data items the transaction has used.
    - How many more data items the transaction needs for it to complete.
    - How many transactions will be involved in the rollback.
- **Rollback**
  - Total rollback: Abort the transaction and then restart it.
  - Partial rollback: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for.
- **Starvation**
  - Oldest transaction in the deadlock set is never chosen as victim.

Recovery System

Types of failure

1) Transaction Failure

Logical error

The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.

System Error

The system has entered an undesirable state (e.g., deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.

2) System Crash

Well-designed systems have numerous internal checks, at the hardware and the software
level, that bring the system to a halt when there is an error.
Hence, the fail-stop assumption is a reasonable one.

3) Disk Failure

A disk block can lose its content due to a head crash or a failure during a data-transfer operation.

In order to determine the appropriate recovery strategy for failures, it is important to identify the failure modes of the devices used to store data.

Next, we need to consider how these failure modes impact the contents of the database.

Based on this analysis, we can propose recovery algorithms that ensure database consistency and transaction atomicity in the event of failures.

These recovery algorithms consist of two parts:

1. During normal transaction processing, actions are taken to ensure that sufficient information is available for recovery in case of failures.
2. After a failure occurs, actions are taken to recover the database contents to a state that guarantees database consistency, transaction atomicity, and durability.

Storage

We identified three categories of storage:

- Volatile storage:
  - Does not survive system crashes
  - Examples: main memory, cache memory
- Non-volatile storage:
  - Survives system crashes
  - Examples: disk, tape, flash memory, non-volatile RAM
  - But may still fail, losing data
- Stable storage:
  - A mythical form of storage that survives all failures
  - Approximated by maintaining multiple copies on distinct non-volatile media

Failure during data transfer can still result in inconsistent copies: Block transfer can result in

- Successful completion: The transferred information arrived safely at its destination.
- Partial failure: A failure occurred in the midst of a transfer, and the destination block has incorrect information.
- Total failure: The failure occurred sufficiently early during the transfer that the destination block remains intact. (destination block was never updated)

Data Access

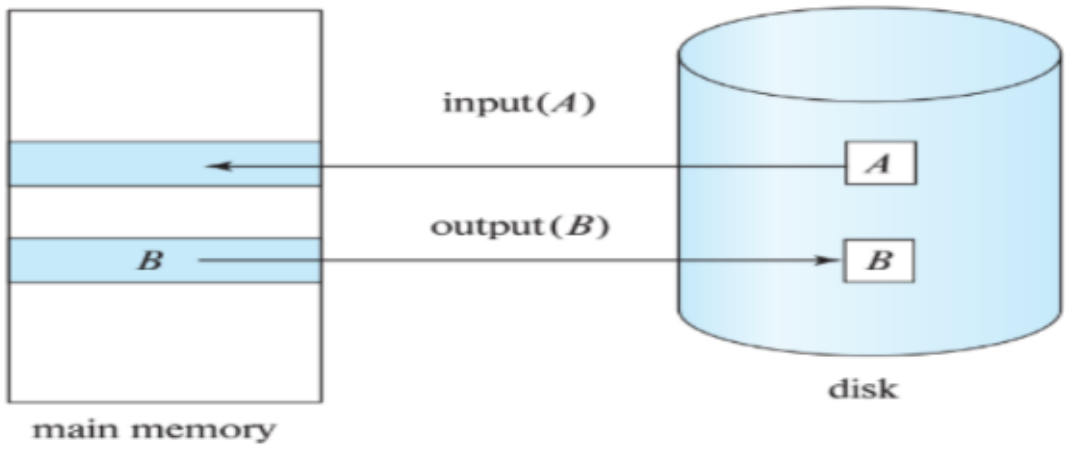


Figure 19.1 Block storage operations.

Physical blocks are those blocks residing on the disk.

Buffer blocks are the blocks residing temporarily in the main memory.

input (B) transfers the physical block B to the main memory.

output (B) transfers the buffer block B to the disk and replaces the appropriate physical block there.

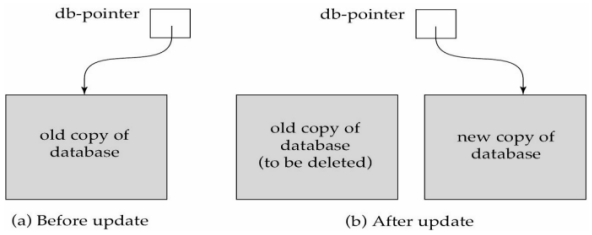
Each transaction Ti has its private work area in which local copies of all data items accessed and
updated by it are kept. Ti 's local copy of a data item X is called xi

•Transferring data items between system buffer blocks and their private work area done by:

- **read**(X) assigns the value of data item X to the local variable x_i.
- **write**(X) assigns the value of local variable x_i to data item $\{X\}$ in the buffer block.
- Note: **output**(B_X) need not immediately follow **write**(X). The system can perform the **output** operation when it deems fit.

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternatives: **shadow-copy** and **shadow-paging** (brief details in the book Database system concepts seventh edition – page: 914)

shadow-copy



Log Records

Log Records

- The most widely used structure for recording database modifications is the log. The log is a sequence of log records, recording all the update activities in the database.
- There are several types of log records. An update log record describes a single database write. It has these fields:

• Transaction identifier, which is the unique identifier of the transaction that performed the write operation.

• Data-item identifier, which is the unique identifier of the data item written. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides and an offset within the block.

• Old value, which is the value of the data item prior to the write.

• New value, which is the value that the data item will have after the write.

We represent an update log record as <Ti, Xj, V1, V2>, indicating that transaction Ti has performed a write on data item Xj. Xj had value V1 before the write and has value V2 after the write. Other special log records exist to

t- transaction identifier

x - data identifier

v1 - old value

v2 - new value

Among the types of log records are:
<Ti start>. Transaction Ti has started.

<Ti commit>. Transaction Ti has committed.

<Ti abort>. Transaction Ti has aborted.

- A **log** is a sequence of **log records**. The records keep information about updated activities on the database.
  - The **log** is kept in stable storage
- When transaction T_i starts, it registers itself by writing a $<T_i$ **start**> log record
- *Before* T_i executes **write**(X), a log record $<T, X, V_1, V_2>$ is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes its last statement, the log record $<T_i$ **commi**t> is written.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification.

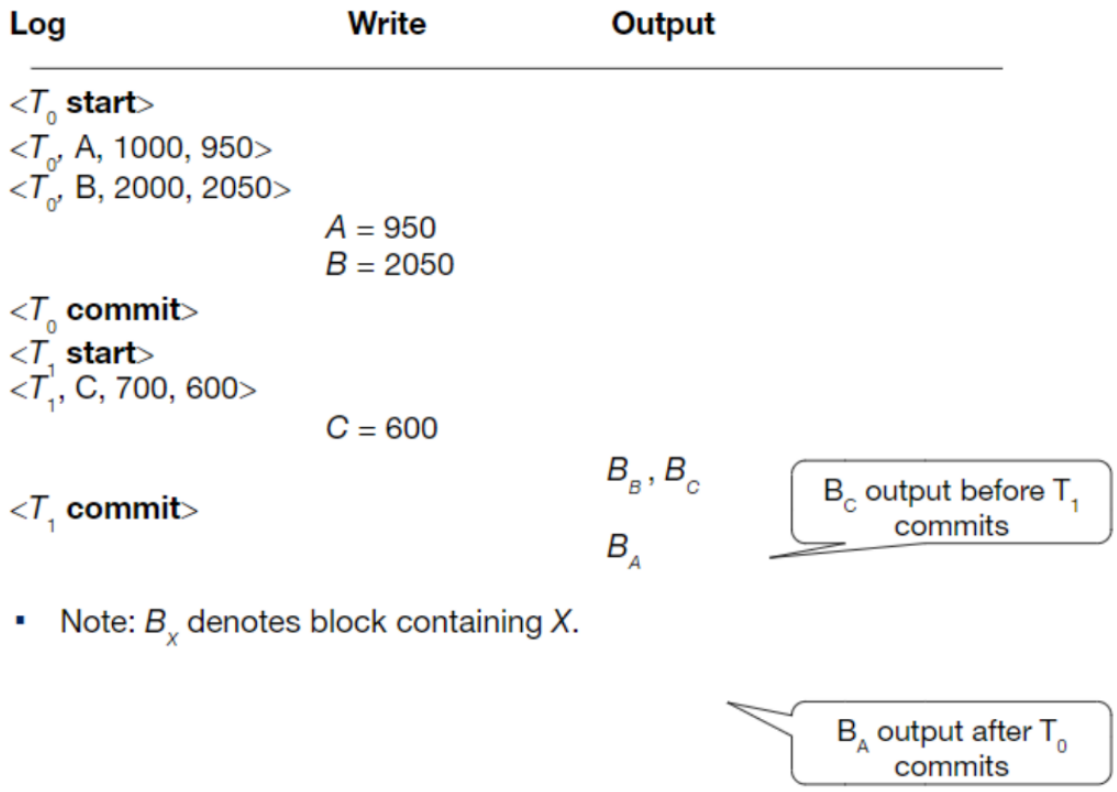
The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted; they allow the system also to redo changes made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk. In order for us to understand the role of these log records in recovery, we need to consider the steps a transaction takes in modifying a data item:

The transaction performs some computations in its own private part of main memory.

The transaction modifies the data block in the disk buffer in main memory holding the data item.

The database system executes the output operation that writes the data block to disk.

immediate-modification scheme allows updates of an uncommitted transaction to be made to the buffer,or the disk itself before the transaction commits



- Note: B_X denotes block containing X.

deferred-modification scheme performs updates to buffer/disk only at the time of transaction commit

• Simplifies some aspects of recovery

• But has overhead of storing a local copy

A recovery algorithm must take into account a variety of factors, including:

• The possibility that a transaction may have committed although some of its database modifications exist
only in the disk buffer in main memory and not in the database on disk.

• The possibility that a transaction may have modified the database while in the active state and, as a result of a subsequent failure, may need to abort.

- **The undo operation using a log record sets the data item specified in the log record to the old value contained in the log record.**

Each time a data item X is restored to its old value V a special log record <Ti, X, V> is written out

- **The redo operation using a log record sets the data item specified in the log record to the new value contained in the log record.**

A transaction is said to have committed when its commit log record is output to stable storage

When a system crash occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. We introduce checkpoints.

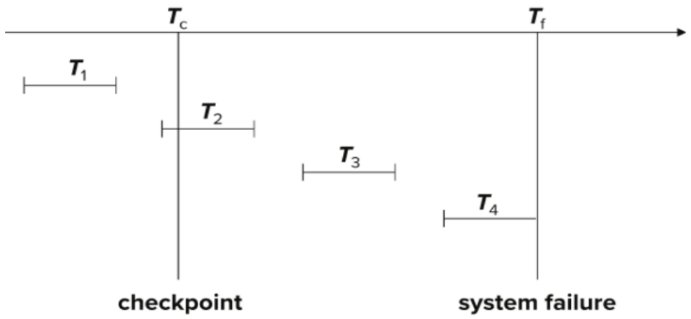
Checkpoint

Output all log records currently residing in main memory onto stable storage.
Output all modified buffer blocks to the disk.
Write a log record < checkpoint L> onto stable storage where L is a list of all transactions active at the time of checkpoint.

All updates are stopped while doing checkpointing

A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undoneT_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Recovery Algorithm

Logging (during normal operation):

1. `<Ti start>` at transaction start.
    - At the beginning of each transaction (T_i) a `<Ti start>` log record is generated.
2. `<Ti, Xj, V1, V2>` for each update.
    - For each update operation in a transaction (T_i) on data item (X_j) , a log record `<Ti, Xj, V1, V2>` is generated, where $(V1)$ is the old value, and$(V2)$ is the new value.
3. `<Ti commit>` at transaction end.
    - When a transaction(T_i)successfully completes, a `<Ti commit>` log record is generated.

Transaction rollback (during normal operation):

1. Let (T_i) be the transaction to be rolled back.
2. Scan log backward from the end, and for each log record of (T_i) of the form `<Ti, Xj, V1, V2>`.
    - For each update operation in (T_i) , perform the undo operation by writing (V1) to (Xj).
    - Write a compensation log record `<Ti, Xj, V1>`. These records are called compensation log records.
3. Once the record `<Ti start>` is found, stop the scan, and write the log record `<Ti abort>`.

Recovery from failure:

Redo phase:

1. Find the last `<checkpoint L>` record, and set undo-list to (L).

- Identify the most recent checkpoint to establish the starting point for recovery.

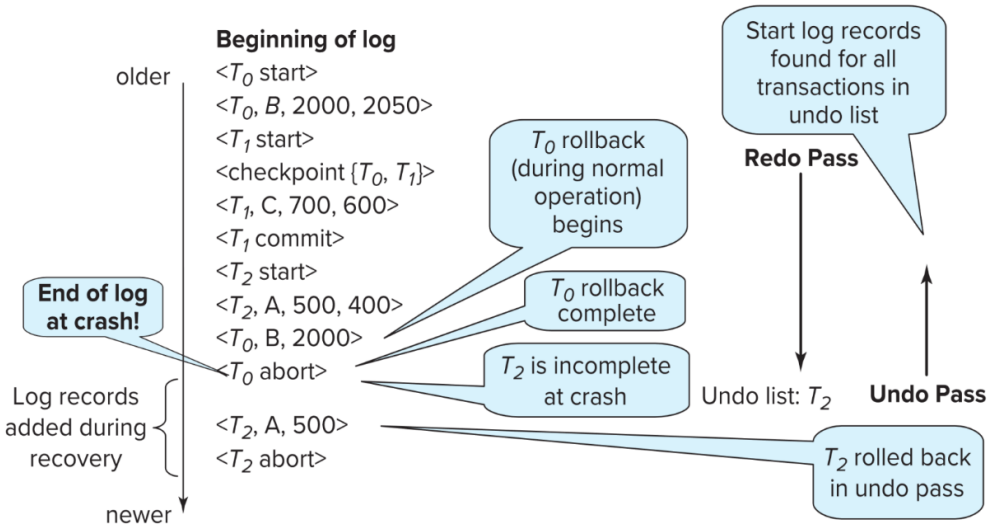
2. Scan forward from the `<checkpoint L>` record.
   - Whenever a record `<Ti, Xj, V1, V2>` or `<Ti, Xj, V2>` is found, redo it by writing (V2) to (Xj).
   - Whenever a log record `<Ti start>` is found, add (T_i) to the undo-list.
   - Whenever a log record `<Ti commit>` or `<Ti abort>` is found, remove (T_i) from the undo-list.

Undo phase:

1. Scan log backward from the end.
   - Whenever a log record `<Ti, Xj, V1, V2>` is found where (T_i) is in the undo-list, perform the same actions as for transaction rollback:
     - Perform undo by writing (V1) to (Xj).
     - Write a log record `<Ti, Xj, V1>`.
   - Whenever a log record `<Ti start>` is found where (T_i) is in the undo-list:
     - Write a log record `<Ti abort>`.
     - Remove (T_i) from the undo-list.
2. Stop when the undo-list is empty.
   - This means that a `<Ti start>` has been found for every transaction in the undo-list.

After undo phase completes, normal transaction processing can commence.

This algorithm ensures that in the event of a failure, the system can recover to a consistent state by redoing committed transactions and undoing incomplete or aborted transactions. The logging mechanism captures the necessary information for recovery.



repeating history

Optimizing commit

Group commit is a technique that optimizes the process of committing transactions by delaying log writes until a group of transactions is ready to be committed. Instead of forcing the log as soon as an individual transaction completes, the system waits until several transactions have completed or a predefined time has passed. It then commits this group of transactions together, resulting in fewer log write operations per committed transaction.

Benefits of Group Commit:

- Reduced Overhead:
- Increased Transaction Rate:
- Reduced Commit Delay:

Batch Processing:

- **Challenge:** Inserting each record as a separate transaction can be limiting, especially in data loading applications.
- **Strategy:** Perform a batch of inserts as a single transaction.

- **Benefits:**
  - Log records are written together on one page.
  - Increases the number of inserts that can be performed per second.
- **Outcome:** This strategy improves the efficiency of data loading processes.

Database Security CIA

- Access Control : Restricting access to the database by unauthorized users. Access control is done by creating user accounts and to control login process by the DBMS.
- Inference Control: This method is known as the countermeasures to statistical database security problem. (**Statistical databases** are used to provide statistical information or summaries of values based on various criteria.)
- Flow Control: This prevents information from flowing in such a way that it reaches unauthorized users.
- Data Encryption: This method is mainly used to protect sensitive data (such as credit card numbers, OTP numbers) and other sensitive numbers.

The DBA's responsibilities include granting privileges to users who need to use the system and classifying users and data in accordance with the policy of the organization.

Whenever a person or a group of persons needs to access a database system, the individual or group must first apply for a user account.

The database system must also keep track of all operations on the database that are applied by a certain user throughout each login session

Sensitive Data and Types of Disclosures

Sensitivity of data is a measure of the importance assigned to the data by its owner for the purpose of denoting its need for protection.

Several factors can cause data to be classified as sensitive:

- **Inherently sensitive**. The value of the data itself may be so revealing or confidential that it becomes sensitive—for example, a person's salary or patient who has HIV/AIDS.
- **From a sensitive source**. The source of the data may indicate a need for secrecy—for example, an informer whose identity must be kept secret.
- **Declared sensitive**. The owner of the data may have explicitly declared it as sensitive.
- **A sensitive attribute or sensitive record**. The particular attribute or record may have been declared sensitive—for example, the salary attribute of an employee or the salary history record in a personnel database.
- **Sensitive in relation to previously disclosed data.** Some data may not be sensitive by itself but will become sensitive in the presence of some other data—for example, the exact latitude and longitude information for a location where some previously recorded event happened that was later deemed sensitive.

Several factors must be considered before deciding whether it is safe to reveal the data. The three
most important factors are:

Data availability
Access acceptability
Authenticity assurance.

Access Control :- security policy specifies who is authorized to do what.

Two A security mechanism allows us to enforce a chosen security policy.
Main mechanisms at the DBMS level:

- Discretionary access control
- Mandatory access control

In many applications, an additional security policy is needed that classifies data and users based
on security classes. This approach, known as
mandatory access control (MAC), the need for multilevel security exists in government, military, and intelligence applications, as well as in many industrial and corporate applications.

Unlike DAC, where the owner can determine the access and privileges and can restrict the resources based on the identity of the users, In MAC, the system only determines the access and the resources will be restricted based on the clearance of the subjects.

Suppose that the DBA creates four accounts—A1, A2, A3, and A4—and wants only A1 to be able to create base relations. To do this, the DBA must issue the following GRANT command in SQL:

GRANT CREATETAB **TO** A1;

Next, suppose that A1 wants to allow account A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts.
A1 can issue the following command:

GRANT SELECT **ON** EMPLOYEE, DEPARTMENT **TO** A3 **WITH GRANT OPTION**;

The clause WITH GRANT OPTION means that A3 can now propagate the privilege to other accounts by using GRANT. For example, A3 can grant the SELECT privilege on the EMPLOYEE relation to A4 by issuing the following command:

GRANT SELECT **ON** EMPLOYEE **TO** A4;

Now suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3; A1 then can issue this command:

REVOKE SELECT **ON** EMPLOYEE **FROM** A3;

The DBMS must now revoke the SELECT privilege on EMPLOYEE from A3, and it must also automatically revoke the SELECT privilege on EMPLOYEE from A4.

This is because A3 granted that privilege to A4, but A3 does not have the privilege any more.