DBMS ASSIGNMENT -5

Ramya Sri Achanta
(2300293)

1.

Avoiding NULL values in a relational database is crucial during the design phase due to several reasons:

1. Uncertainty: NULLs introduce uncertainty in data interpretation. They could signify unknown, inapplicable, or missing values, leading to confusion.
2. Query Complexity: Dealing with NULLs in SQL queries adds complexity. NULLs behave differently in comparisons and calculations, often causing unexpected outcomes and making queries harder to comprehend.
3. False Tuples: Presence of NULLs can result in false data, creating misleading combinations that don't accurately represent real-world information, leading to erroneous results in operations like joins.
4. Indexing Inefficiency: NULLs can hinder index efficiency. Some databases don't include NULLs in indexes by default, impacting query performance, especially when NULLs are involved.

To prevent false tuples and mitigate the issues caused by NULLs, you can follow these best practices during the database design process:

1. Use NOT NULL constraints: When establishing table columns for attributes that must always have a value, use NOT NULL constraints. By ensuring that NULLs are prohibited in those columns, this constraint helps to avoid false tuples that result from missing or unknown data. Hence that helps to maintain data integrity.

2. Normalize the database: Normalization: To lessen dependencies and redundancies, normalize the database schema. Normalization reduces the likelihood of NULL values and contributes to the preservation of data integrity by arranging data into distinct tables according to their functional relationships.

3. Use default values: Consider setting default values for columns where ever it’s appropriate, rather than permitting NULL entries. When explicit values are not provided during data insertion, default values offer sensible replacements, preventing NULLs and ensure that columns always contain correct data.

4. Data Validation: To guarantee that only accurate and comprehensive data is entered into the database, implement strong data validation procedures. Checks for data validation can assist in identifying and preventing the insertion of NULL values in places where they are not allowed.

5. Handle NULLs explicitly in queries: Use the relevant comparison operators, such as IS NULL or IS NOT NULL, to explicitly handle NULLs in queries. You may guarantee accurate querying and stop false tuples from influencing query results by explicitly controlling NULL values.

By putting these precautions in place, database designers may lessen the issue of false tuples and keep the database accurate and dependable. Achieving a balance between eliminating NULL-related problems and allowing for valid use cases in which NULL values could be required or appropriate is crucial.

2. The fourth phase of database normalization known as the Fourth Normal Form (4NF) improves on the ideas presented in the Third Normal Form (3NF). It deals with specific kinds of dependence and redundancy problems that could still be present in a 3NF-compliant structure.

Definition : A relation R is said to be in the Fourth Normal Form (4NF) if it is in the Third Normal Form (3NF) and has no multi-valued dependencies (MVDs). When redundancy results from two or more independent multi-valued dependencies between attributes in a table, an MVD is present.

Violation of 4NF: If a relation has a non-trivial multivalued dependency that isn't caused by the candidate keys, it breaks the Fourth Normal Form. In simpler terms, a relation is not in 4NF if there is a multivalued dependency with a determinant that is not a superkey and that is not inferred by the candidate keys.

Applicability: The Fourth Normal Form is typically applicable in scenarios where there are multivalued dependencies in the data. Multivalued dependencies occur when the presence of one or more independent multivalued attributes in a relation leads to the existence of multiple related rows. These dependencies can cause data redundancy and update anomalies.

Examples could include:

1.Modeling scenarios with numerous independent multivalued properties, such as a student enrolling in multiple courses and having multiple phone numbers, is one common example where 4NF is appropriate.
2. Depicting many-to-many relationships using attributes that are dependent on just one of the relationship's entities.

You can guarantee data integrity and remove redundancy brought on by multivalued dependencies by using the Fourth Normal Form. It lessens update anomalies, minimizes redundant data, and enhances the database's overall architecture.

3. Analysis of Relation R with Functional Dependencies

Given:

R = {A, B, C, D, E, F, G, H, I, J}

F = {{A, B}→{C}, {A}→{D, E}, {B}→{F}, {F}→{G, H},{D}→{I, J}}

a) Finding a superkey for the relation: A superkey is a set of attributes that uniquely identifies a tuple in the relation.

closure of {A,B}={A,B,C,D,E,F,G,H,I,J} (using {AB}->{C} , {A}->{DE},{B}->{F} and {F}->{GH})

From the given functional dependencies, we can see that {A, B} determines all other attributes. Therefore, {A, B} is a superkey for relation R.

b) Finding the candidate key for R: A candidate key is a minimal superkey, meaning it doesn't contain any redundant attributes. In this case, {A, B} is the only candidate key for R because no proper subset of {A, B} can determine all other attributes.

A+={DEIJ} !=R

B+={FGH}!=R

Hence AB is candidate key.

c) Checking whether H is redundant in F → GH: To check if H is redundant, we need to see if F → G holds Given {F}→{G, H}, we can see that H is not redundant because removing H from the dependency would change its closure, and it is necessary for determining the closure of F.

d) Checking whether B is redundant in {A, B} → C: To check if B is redundant, we need to see if A → C holds. From the given functional dependencies, we cannot deduce A → C. Therefore, B is not redundant in {A, B} → C.

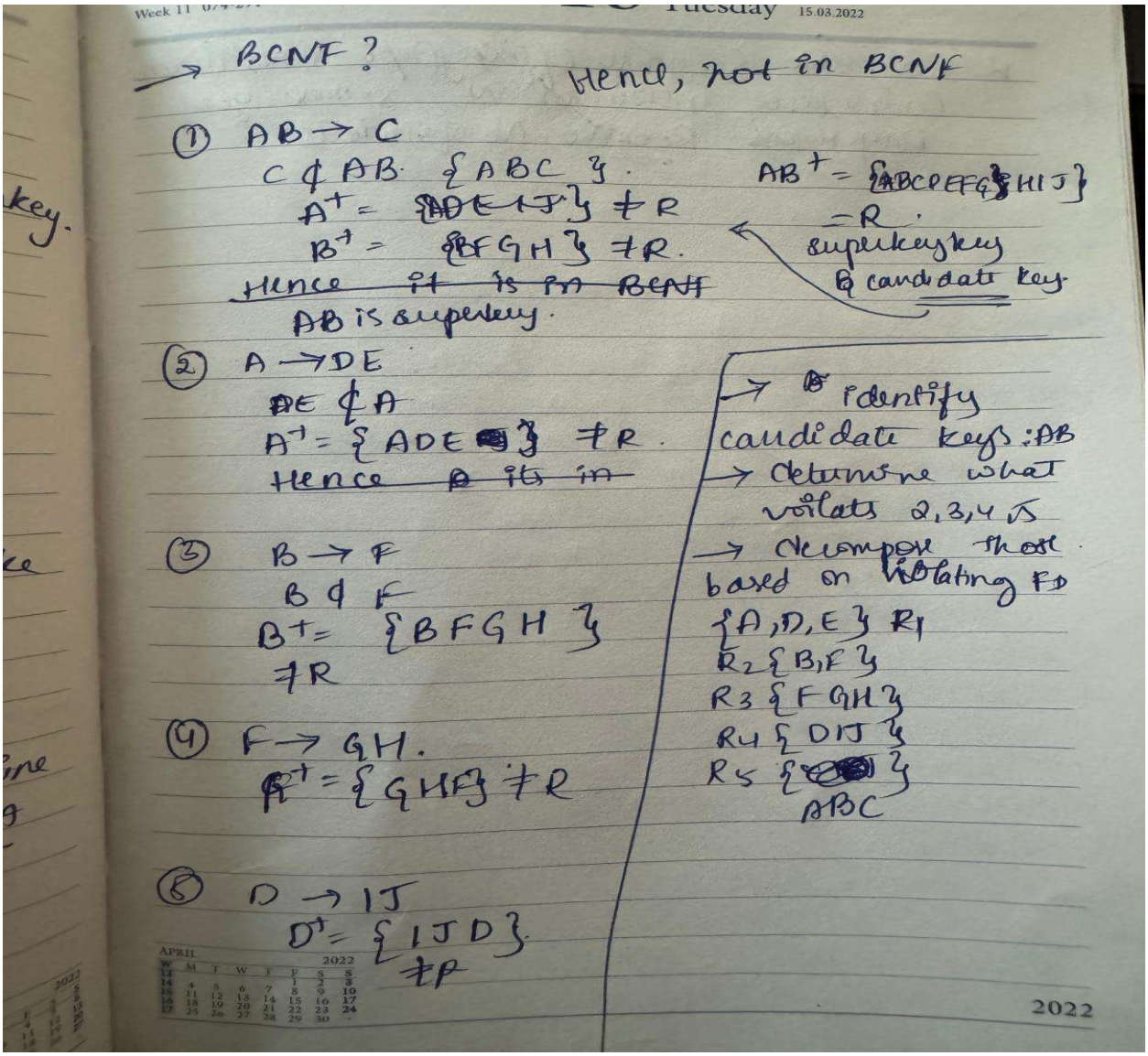
e) Checking whether R is in BCNF: A relation is in BCNF if, for every non-trivial functional dependency X → Y, X is a superkey. Let's check each functional dependency:

- {A, B} → C: {A, B} is a superkey, so this dependency satisfies BCNF.
- A → {D, E}: A is not a superkey, so this violates BCNF.

- B → F: B is not a superkey, so this violates BCNF.
- F → {G, H}: F is not a superkey, so this violates BCNF.
- D → {I, J}: D is not a superkey, so this violates BCNF.

Since not all the dependencies satisfy BCNF. Therefore, R is not in BCNF.

Below image is rough work of d) and e)



f) Attribute-preserving and lossless join decomposition:

we can perform a decomposition based on the minimal cover of given dependencies.
Minimal cover:

- {A, B}→{C}
- {A}→{D, E}
- {B}→{F}
- {F}→{G, H}
- {D}→{I, J}

We can decompose R into smaller relations based on the functional dependencies to achieve a lossless join decomposition. From the minimal cover we can get possible decomposition as follows:

- R1 = {A, B, C}
- R2 = {A, D, E}
- R3 = {B, F}
- R4 = {F, G, H}
- R5 = {D, I, J}

This decomposition is attribute-preserving because all attributes from the original relation are present in the decomposed relations. It is also a lossless join decomposition because the common attributes between the relations form a superkey in at least one of the relations.

g) Dependency-preserving decomposition: A decomposition is dependency-preserving if each functional dependency in the original relation is represented in at least one of the decomposed relations. Using the decomposition from the previous step:

- {A, B} → C is preserved in R1(A,B,C)
- A → {D, E} is preserved in R2(A,D,E)
- B → F is preserved in R3(B,F)
- F → {G, H} is preserved in R4(F,G,H)
- D → {I, J} is preserved in R5(D,I,J)

Therefore, the decomposition is dependency-preserving

h) Decomposition the R into 3NF:

Therefore, the decomposition is dependency-preserving.we can find out that there aren't any transitive dependencies in the minimal cover so the relation obtained from the decomposition are already in 3NF.

- R1({A, D, E})
- R2({F, G, H})
- R3({A, B, C})

- R4({B, F})
- R5({D, I, J})

Each relation contains dependencies where the determinant is a key of the relation, which is a requirement for 3NF. These relations are in 3NF, as each non-prime attribute is fully functionally dependent on the candidate keys.

4) Analysis of University Schedule Relation R

Given:

R = {CourseNo, SectionNo, OfferingDept, CreditHours, CourseLevel, InstructorSSN, Semester, Year, DaysHours, RoomNumber, NumberOfStudents}

F = {{CourseNo → OfferingDept, CreditHours, CourseLevel},{CourseNo, SectionNo, Semester, Year →

DaysHours, RoomNumber, NumberOfStudents, InstructorSSN},{RoomNumber, DaysHours, Semester, Year → InstructorSSN, CourseNo, SectionNo}}

a) Computing the candidate key for the relation R:

closure of {CourseNo, SectionNo, Semester, Year}+={ CourseNo, SectionNo, OfferingDept, CreditHours, CourseLevel, InstructorSSN, Semester, Year, DaysHours, RoomNumber, NumberOfStudents }

closure of {CourseNo}+={ CourseNo, SectionNo, OfferingDept, CreditHours, CourseLevel, InstructorSSN, Semester, Year, DaysHours, RoomNumber, NumberOfStudents }
From the given functional dependencies, we can see that the combination of {CourseNo, SectionNo, Semester, Year} and {CourseNo} uniquely determines all other attributes in the relation. Therefore, {CourseNo, SectionNo, Semester, Year} and {CourseNo} is a candidate key for R.

b) Applying Armstrong's axioms to check functional dependencies:

1.CourseNo → OfferingDept, CourseLevel:

From the given functional dependencies, we have CourseNo → OfferingDept, CreditHours, CourseLevel.

By applying the decomposition rule of Armstrong's axioms, we can infer that CourseNo → OfferingDept and CourseNo → CourseLevel hold.

2.CourseNo, Semester → OfferingDept, CreditHours, CourseLevel:

We can use the given functional dependencies and Armstrong's axioms to check if this dependency holds.

1.1 CourseNo → OfferingDept, CreditHours, CourseLevel (given)

1.2. CourseNo, Semester → CourseNo (reflexivity)

1.3. CourseNo, Semester → OfferingDept, CreditHours, CourseLevel (transitivity, using 1 and 2)

Therefore, CourseNo, Semester → OfferingDept, CreditHours, CourseLevel holds.

c) Checking if {CourseNo, SectionNo} can be a superkey and if CourseNo can be a candidate key:

1. Closure of {CourseNo,SectionNo}+= {CourseNo, OfferingDept, CreditHours, CourseLevel}.=! R

{CourseNo, SectionNo} cannot be a superkey because it does not uniquely determine all other attributes in the relation. For example, it does not determine the Semester and Year attributes.

2.closure of {CourseNo}+={ CourseNo, SectionNo, OfferingDept, CreditHours, CourseLevel, InstructorSSN, Semester, Year, DaysHours, RoomNumber, NumberOfStudents }

From the given functional dependencies, we can see that the {CourseNo} uniquely determines all other attributes in the relation. Therefore {CourseNo} is a candidate key for R.

.

d) Checking if the relation R is in BCNF or 3NF:

To check if R is in BCNF, we need to ensure that for every non-trivial functional dependency X → Y, X is a superkey. Let's check each functional dependency:

1.CourseNo → OfferingDept, CreditHours, CourseLevel: CourseNo is a candidate key ({CourseNo}).Therefore, this functional dependency satisfies the conditions for both BCNF and 3NF.

2.CourseNo, SectionNo, Semester, Year → DaysHours, RoomNumber, NumberOfStudents, InstructorSSN: {CourseNo, SectionNo, Semester, Year} is a superkey, so this satisfies BCNF and 3NF.

3.RoomNumber, DaysHours, Semester, Year → InstructorSSN, CourseNo, SectionNo: {RoomNumber, DaysHours, Semester, Year} is not a superkey, so this violates BCNF.

Therefore, R is not in BCNF.

To check if R is in 3NF, we need to ensure that for every non-trivial functional dependency X → Y, either X is a superkey or Y is a prime attribute (part of a candidate key). Let's check each functional dependency:

- RoomNumber, DaysHours, Semester, Year → InstructorSSN, CourseNo, SectionNo: {RoomNumber, DaysHours, Semester, Year} is not a superkey, and {InstructorSSN, CourseNo, SectionNo} are not prime attributes, so this violates 3NF.

Therefore, R is not in 3NF.

Since the third functional dependency violates the conditions for both BCNF and 3NF, the relation R is not in BCNF or 3NF. To ensure the relation is in BCNF or 3NF, we need to decompose it further to remove the dependency violating the normal form conditions.

e) Giving a lossless join, dependency-preserving decomposition into BCNF or 3NF for the schema R:

To achieve a lossless join and dependency-preserving decomposition, we can decompose R into smaller relations based on the functional dependencies. Here's a possible decomposition into 3NF:

- R1 = {CourseNo, OfferingDept, CreditHours, CourseLevel}

- R2 = {CourseNo, SectionNo, Semester, Year, DaysHours, RoomNumber, NumberOfStudents, InstructorSSN}

- R3 = {RoomNumber, DaysHours, Semester, Year, InstructorSSN, CourseNo, SectionNo}

This decomposition is lossless join because the common attributes between the relations form a superkey in at least one of the relations. It is also dependency-preserving because each functional dependency is represented in at least one of the decomposed relations.

5.

a) DTD for the XML document:

```
<!ELEMENT CATALOG (PLANT | FOOD | BOOK | CUSTOMER)*>

<!ELEMENT PLANT (COMMON, BOTANICAL, ZONE, LIGHT, PRICE, AVAILABILITY)>

<!ELEMENT FOOD (NAME, PRICE, DESCRIPTION, CALORIES)>

<!ELEMENT BOOK (AUTHOR, TITLE, GENER, PRICE, DATE, QTY)>

<!ELEMENT CUSTOMER (NAME, ITEM+, TOTAL, PHONE)>

<!ATTLIST PLANT pid ID #REQUIRED>

<!ELEMENT COMMON (#PCDATA)>

<!ELEMENT BOTANICAL (#PCDATA)>

<!ELEMENT ZONE (#PCDATA)>

<!ELEMENT LIGHT (#PCDATA)>

<!ELEMENT PRICE (#PCDATA)>

<!ELEMENT AVAILABILITY (#PCDATA)>

<!ELEMENT NAME (#PCDATA)>

<!ELEMENT DESCRIPTION (#PCDATA)>

<!ELEMENT CALORIES (#PCDATA)>

<!ATTLIST BOOK isbn ID #REQUIRED>

<!ELEMENT AUTHOR (FNAME, MI?, LNAME)>

<!ATTLIST AUTHOR aid ID #REQUIRED>

<!ELEMENT TITLE (#PCDATA)>

<!ELEMENT GENER (#PCDATA)>
```

<!ELEMENT DATE (#PCDATA)>

<!ELEMENT QTY (#PCDATA)>

<!ELEMENT FNAME (#PCDATA)>

<!ELEMENT MI (#PCDATA)>

<!ELEMENT LNAME (#PCDATA)>

<!ATTLIST CUSTOMER cid ID #REQUIRED>

<!ELEMENT NAME (FNAME, MI?, LNAME)>

<!ELEMENT ITEM (#PCDATA)>

<!ELEMENT TOTAL (#PCDATA)>

<!ELEMENT PHONE (#PCDATA)>

<!ELEMENT FNAME (#PCDATA)>

<!ELEMENT MI (#PCDATA)>

<!ELEMENT LNAME (#PCDATA)>

b. XML to RDMS mapping:

- PLANT table: pid (primary key), COMMON, BOTANICAL, ZONE, LIGHT, PRICE, AVAILABILITY
- FOOD table: pid (primary key), NAME, PRICE, DESCRIPTION, CALORIES
- BOOK table: isbn (primary key), TITLE, GENER, PRICE, DATE, QTY
- AUTHOR table: aid (primary key), FNAME, MI, LNAME
- BOOK_AUTHOR table: isbn (foreign key referencing BOOK), aid (foreign key referencing AUTHOR)
- CUSTOMER table: cid (primary key), FNAME, MI, LNAME, TOTAL, PHONE
- CUSTOMER_ITEM table: cid (foreign key referencing CUSTOMER), ITEM

The PLANT, FOOD, BOOK, and CUSTOMER elements are mapped to separate tables with their respective attributes as columns. The AUTHOR element is mapped to a separate table, and the relationship between BOOK and AUTHOR is captured using the BOOK_AUTHOR table. The CUSTOMER_ITEM table captures the many-to-many relationship between CUSTOMER and ITEM.

c. XPath query to find names of plants with Shady light environment and availability > 0:

```
/CATALOG/PLANT[LIGHT='Shady' and AVAILABILITY>0]/COMMON
```

d. XQuery to find books sold more than 50 copies:

```
 <books> {
for $i in /CATALOG/BOOK
where $i/QTY > 50
return <book>{$i/TITLE}</book>
 } </books>
```

e. XQuery to find the customer with the highest amount of catalog items:

```
        let $customer-items :=
           for $customer in /CATALOG/CUSTOMER
           return
              <customers-name-total>
                 <NAME>{string-join($customer/NAME/(FNAME, LNAME), ' ')}</NAME>
                 <TOTAL>{sum($customer/ITEM)}</TOTAL>
              </customers-name-total>
        return
           $customer-items[./TOTAL = max($customer-items/TOTAL)]
```