Week 5: Functional Dependency and Normal Forms

L5.1: *Relational Database Design Part - I*

Features of Good Relational Design

- Reflects *real-world structure* of the problem
- Can represent *all expected data* over time
- Avoids *redundancy* of data
- Provides *efficient access* to data
- Supports the *maintainance of data integrity* over time
- *Clean, consistent, and easy* to understand
- Note: These objectives are sometimes contradictory.
    - For example, redundancy can improve performance, but it can also lead to data integrity problems.

A Good Schema - Example:

- Consider combining relations
    - `sec_class(sec_id, building, room_number)`
    - `section(course_id, sec_id, semester, year)`

into one relation

- `section(course_id, sec_id, semester, year, building, room_number)`
- No redundancy

Redundancy

- Redundancy is the repetition of data in multiple copies of same data in teh database.
    - This problem arises when a database is not normalized.
    - It leads to anomalies.

Anomaly

- Inconsistencies that can arise due to data changes in a database with insertion, deletion, and updation.
    - These problems occurred in poorly planned, un-normalized databases where all the data is stored in one table.

Insertions Anomaly

- When the insertion of data record is not possible without adding some additional unrrelated data to the record.
- Example:
    - We have 3 tables, `courses`, `students` and `enrollements`.
    - We cannot add an enrollement record for a course which is not present in the `courses` table.

Deletion Anomaly

- When deletion of a data record results in losing some unrelated information that was stored as a part of the record that was deleted from a table.
- Example:
    - We have 3 tables, `courses`, `students` and `enrollements`.

    - We cannot delete a course record from the `courses` table if there are enrollements for that course.
    - We have to delete all the enrollements for that course first.

Updation Anomaly

- When a data is changed, which could involve many records having to be changed, leading to the possibility of some changes being made incorrectly.
- Example:
    - We have 3 tables, `courses`, `students` and `enrollements`.
    - We cannot update the `course_name` in the `enrollements` table without updating the `course_name` in the `courses` table.

$$\text{Normalization} \implies \text{Good decomposition} \implies \text{Minimization of Dependency} \implies \text{Redundancy} \implies \text{Anomaly}$$

Normalization is the process of decomposing a relation into smaller relations to minimize redundancy and avoid anomalies.

Decomposition

- Decomposition is the process of breaking down a relation into smaller relations based on functional dependencies.
- Example:
    - We have a relation `customers(customer_id, name, address, city, state, pin_code, phone_number, email_id)`.
    - We can decompose it into 3 relations:
        - `customers(customer_id, name)`
        - `customer_address(customer_id, address, city, state, pin_code)`
        - `customer_contact(customer_id, phone_number, email_id)`

Functional Dependency

- A functional dependency is a relationship between two attributes in a table where the value of one attribute (*determinant*) determines the value of another attribute (*dependent*).
- Example:
    - In the table `students`, the attribute `student_id` determines the values of the attributes `name` and `age`.
    - This is because each student has a unique student ID, and the name and age of a student is determined by their student ID.
- Example 2:
    - In table `customers` `customer_id` determines the values of the attributes `name`, `address`, `city`, `state`, `pin_code`, `phone_number`, `email_id`.

How to make good decompositions?

- Not all decompositions are good.
- A decomposition is good if it is lossless and dependency preserving.
- Example:
    - If we decompose the `customers` table into `customers(customer_id, name` and `customer_address(name, address, city, state, pin_code)`, then this is a **NOT** a good decomposition.

Lossless Decomposition

- A decomposition is lossless if we can join the decomposed relations and get the original relation.

Atomic Domain

- A domain is atomic if it contains values that cannot be divided into smaller components.

- Example:
  - `customer_id` is atomic because it cannot be divided into smaller components.
  - `name` is not atomic because it can be divided into `first_name` and `last_name`.

Frist Normal Form (1NF)

- A relation is in `1NF` if it has no repeating groups, i.e. the domain of each attribute contains only atomic values.
- Example:
  - `customers(customer_id, name, address, city, state, pin_code, phone_number, email_id)` is not in `1NF` because the domain of `name` is not atomic.
  - `customers(customer_id, first_name, last_name, address, city, state, pin_code, phone_number, email_id)` is in `1NF` because the domain of each attribute is atomic.
- Non-atomic values complicate storage and retrieval of data.

L5.2 & 3: *Relational Database Design Part - II & III*

Devise a Theory for Good Relations

- We want to decide whether a particular relation R is in "good" form.
- IN the case that a relation R is not in "good" form, decompose it into a set of relations $\{R_1, R_2, \ldots, R_n\}$ such that:
  - each relation is in ***good*** form
  - the decomposition is ***lossless-join decomposition***.

We have to focus on different types of dependencies that can exist between attributes of a relation.

Functional Dependencies

- A functional dependency is a relationship between two attributes in a table where the value of one attribute (determinant) determines the value of another attribute (dependent).
- Example:
  - In the table `students`, the attribute `student_id` determines the values of the attributes `name` and `age`.
  - This is because each student has a unique student ID, and the name and age of a student is determined by their student ID.

Armstrong's Axioms

- Armstrong's axioms are a set of inference rules used to infer all the functional dependencies on a relational database.

Reflexivity

- If $Y \subseteq X$, then $X \rightarrow Y$.
- It means that if a set of attributes functionally determines another set of attributes, then the original set of attributes also functionally determines the subset.

Augmentation (Additivity)

- If $Y \rightarrow Z$, then $XZ \rightarrow XY$.
- It means if two sets of attributes are functionally determined by the same set of attributes, then their combination is also functionally determined by that set.

Transitivity

- If $X \to Y$ and $Y \to Z$, then $X \to Z$.
- It means that if two sets of attributes are functionally determined by different sets of attributes, and one set is a subset of the other, then the first set also functionally determines the second set.

- By applying these axioms iteratively, you can find all the closure of a set of attributes, which represents all the functional dependencies that can be derived from the given set of functional dependencies.
- These derived functional dependencies are essential for normalization and ensuring data integrity in a database.
- The new FD obtained by applying the axioms is said to the *logically implied* by F.
- The process of generations of FDs terminate after finitely number of steps and we call it the **Closure Set F⁺** and $F \subseteq F^+$
- These axioms are:
  - **Sound**: Generate only functional dependencies that are true in all cases.
  - **Complete**: Eventually generate all functional dependencies that are true in all cases.
    Example:
  - $F = \{A \to B, B \to C\}$
  - $F^+ = \{A \to B, B \to C, A \to C\}$

Additional Derived rules:

Union

- If $X \to Y$ and $X \to Z$, then $X \to YZ$.

Decomposition

- If $X \to YZ$, then $X \to Y$ and $X \to Z$.

Pseudotransitivity

- If $X \to Y$ and $WY \to Z$, then $WX \to Z$.

Closure of Attribute Sets

- The closure of an attribute set α under a set of functional dependencies F (denoted by $\alpha^+$is the set of all attributes that are functionally determined by α under F.
- Algorithm to compute α^+, the closure of α under F

```
result <- a
while (changes to result) do
    for each b -> c in F do
        begin
            If (b is a subset of result) then
                result <- result U c
        end
```

Usecases

- Testing for superkey
- Testing functional dependencies
- Computing the closure of F

Boyce-Codd Normal Form (BCNF)

- A relation R is in BCNF with respect to a set F of FDs if for all FDs in F^+ of the form:

$$a \to b, \text{ where } a \subseteq R \text{ and } b \subseteq R \text{ holds any of:}$$

    - $a \rightarrow b$ is a trivial FD (i.e. $b \subseteq a$)
    - a is a superkey of R
- Example:
    - `instr_dept(ID, name, salary, dept_name, building, budget)` is not in BCNF because the non-trivial dependency `dept_name` $\rightarrow building, budget$ holds and `dept_name` is not a superkey.

Decomposition

- If in schema R and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF, we decompose R into:
    - $\alpha \cup \beta$
    - $R - (\beta - \alpha)$

Loseless Join

- If we decompose a relation R into R_1 and R_2:
    - Decomposition is **lossy** if $R \subset R_1 \bowtie R_2$
    - Decomposition is lossless if $R_1 \bowtie R_2 = R$
- To check for lossless join decomposition using FD set, following must hold:
    - Union of Attributes of R_1 and R_2 must be equal to R

$$R_1 \cup R_2 = R$$

    - Intersection of Attributes of R_1 and R_2 must not be NULL

$$R_1 \cap R_2 \neq \phi$$

    - Common attribute must be a key for at least one relation

$$R_1 \cap R_2 \rightarrow R_1 \text{ or } R_1 \cap R_2 \rightarrow R_2$$

Third Normal Form (3NF)

- A relation schema R is in **third normal form (3NF)** if for all:

$$a \rightarrow b \in F^+$$

    - at least one of the following holds: - $a \rightarrow b$ is a trivial FD ($b \subseteq a$) - a is a superkey of R - Each attribute A in $b - a$ is contained in a candidate key for R
- If a relation is in both BCNF then it is also in 3NF

Goals of Normalization

- Decide whether a relation schema R is in "good" form
- In the case that R is not in "good" form, decompose R into a collection of schemas that are in "good" form such that:
    - each relation schema is in good form
    - the decomposition is lossless-join decomposition
    - Preferably, the decomposition is dependency preserving

Problems with Decomposition

- Maybe impossible to reconstruct the original relation
- Dependency checking may require joins
- Some queries are more expensive

Tutorial 5.3: *Closure of Attribute*

L5.4: *Relational Database Design Part - IV*

Attribute set closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B,\ A \rightarrow C,\ CG \rightarrow H,\ CG \rightarrow I,\ B \rightarrow H\}$
- $(AG)^+$
  - a) $result = AG$
  - b) $result = ABCG$ $(A \rightarrow C \text{ and } A \rightarrow B)$
  - c) $result = ABCGH$ $(CG \rightarrow H \text{ and } CG \subseteq AGBC)$
  - d) $result = ABCGHI$ $(CG \rightarrow I \text{ and } CG \subseteq AGBCH)$
- Is AG a candidate key?
  - a) Is AG a super key?
    - i) Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$
  - b) Is any subset of AG a superkey?
    - i) Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
    - ii) Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$

Extraneous Attributes

- The attributes in a relation that do not contribute to the functional dependencies of the relation.
- In other words, these attributes are not part of any candidate key or do not play a role in determining other attributes.
- These are also known as *non-prime attributes*.
- During the process of normalization, one of the goals is to eliminate redundancy and ensure that each attribute is fully functionally dependent on the candidate keys. Attributes that are not functionally dependent on the candidate keys are considered extraneous and should be removed to achieve a higher normal form.
- Consider a set F of FDs and the FD $\alpha \rightarrow \beta$ in F.
  - Attribute A is **extraneous** in α if $A \in \alpha$ and F logically implies:

$$(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$$

- Example:
  - Consider a relation **Employee** with attributes `{EmployeeID, EmployeeName, Department, DepartmentLocation}`. If we find

that the attribute `DepartmentLocation` is not functionally dependent on `EmployeeID` or `EmployeeName` and it is not part of any candidate key, then `DepartmentLocation` is considered an extraneous attribute and can be removed from the **Employee** relation to eliminate redundancy.

Tests

- Consider a set F of FDs and the FD $a \rightarrow b$ in F.
- To test if attribute $A \in a$ is extraneous in a.
  1. Compute $(\{a\} - A)^+$ using the dependencies in F.
  2. Check that $(\{a\} - A)^+$ contains b, if it does, A is extraneous in a.
- To test if attribute $X \in b$ is extraneous in b.
  1. Compute a^+ using only the dependencies in:
     $F' = (F - \{a \rightarrow b\}) \cup \{a \rightarrow (b - X)\}$
  2. Check that a^+ contains X, if it does, X is extraneous in b.

Equivalence of Sets of Functional Dependencies

- Equivalence of sets of functional dependencies refers to the concept that two sets of functional dependencies have the same closure, and therefore, they represent the same set of constraints on a relation.
- In other words, if two sets of functional dependencies lead to the same set of attributes being functionally determined, they are equivalent.
- Let $F \& G$ are two FDs.
  - These two sets $F \& G$ are equivalent if $F^+ = G^+$, means:
    $(F^+ = G^+) \iff (F^+ \implies G \text{ and } G^+ \implies F)$
    - Equivalence means that every functional dependency in F can be inferred from G, and every functional dependency in G can be inferred from F.
- F and G are equal only if
  - **F covers G**: Means that all FDs of G are logically members of FDs $F \implies G \subseteq F^+$
  - **G covers F**: Means that all FDs of F are logically members of FDs $G \implies F \subseteq G^+$

Condition	CASES			
F covers G	True	True	False	False
G covers F	True	False	True	False
Result	$F = G$	$G \subset F$	$F \subset G$	No comparison

Example:

Consider a relation **Student** with attributes `{StudentID, StudentName, Course, CourseInstructor}`.
We want to determine the functional dependencies in this relation and check if two sets of functional dependencies are equivalent.

Set of Functional Dependencies F:
F: `{StudentID → StudentName, Course → CourseInstructor}`

Set of Functional Dependencies G:
G: `{StudentID → StudentName, StudentID → Course, Course → CourseInstructor}`

Step 1: Find `Closure(F)` or F^+

F^+ = `{StudentID → StudentName, Course → CourseInstructor, StudentID → Course}`
`(StudentID → StudentName)` is already in F.

Using Transitivity `(Course → CourseInstructor and StudentID → Course)`, we can derive `StudentID → Course`.
Therefore,
F^+ = `{StudentID → StudentName, Course → CourseInstructor, StudentID → Course}`

Step 2: Find `Closure(G)` or G^+

G^+ =
`{StudentID → StudentName, Course → CourseInstructor, StudentID → Course, Course → StudentName, Course → CourseInstructor}`
`(StudentID → StudentName, Course → CourseInstructor, StudentID → Course)` are already in G.
Using Transitivity `(Course → StudentName)`, we can derive `Course → StudentName`.
Therefore,
G^+ =
`{StudentID → StudentName, Course → CourseInstructor, StudentID → Course, Course → StudentName, Course → CourseInstructor}`

Step 3: Check Equivalence

Since both F^+ and G^+ are identical, F and G are equivalent sets of functional dependencies. This means that the two sets represent the same set of constraints on the **Student** relation.

Canonical Cover

- Sets of FDs may have redundant dependencies that can be inferred from the others.
- A **canonical cover** is a set of dependencies that is equivalent to the original set of dependencies, but has no redundant dependencies.
- A **canonical cover** for F is a set of dependencies F_c, such that **ALL** the following properties are satisfied:
  - $F^+ = F_c^+$:
    - F logically implies all dependencies in F_c.
    - F_c logically implies all dependencies in F.
  - No FD in F_c contains extraneous attributes.
  - Each left side of an FD in F_c is unique.

Intuitively, a **Canonical cover** of F is a *minimal* set of FDs:

- Equivalent to F.
- Having no redundant dependencies.
- No redundant parts of FDs.

- For example: $A \to C$ is redundant in: $\{A \to B, B \to C, A \to C\}$
- Parts of a functional dependency may be redundant
  - For example: on RHS: $\{\boldsymbol{A \to B, B \to C, A \to CD}\}$ can be simplified to $\{\boldsymbol{A \to B, B \to C, A \to D}\}$
    - In the forward: (1) $A \to CD \Rightarrow A \to C$ and $A \to D$
      (2) $A \to B, B \to C \Rightarrow A \to C$
    - In the reverse: (1) $A \to B, B \to C \Rightarrow A \to C$
      (2) $A \to C, A \to D \Rightarrow A \to CD$
  - For example: on LHS: $\{\boldsymbol{A \to B, B \to C, AC \to D}\}$ can be simplified to $\{\boldsymbol{A \to B, B \to C, A \to D}\}$
    - In the forward: (1) $A \to B, B \to C \Rightarrow A \to C \Rightarrow A \to AC$
      (2) $A \to AC, AC \to D \Rightarrow A \to D$

- In the reverse: $A \rightarrow D \Rightarrow AC \rightarrow D$

Computing Canonical Cover

```
repeat
    Use the union rule to replace any dependencies in F
        a1 → b1 and a1 → b2 with a1 → b1b2
    Find a FD a → b with an extraneous attribute either in a or in b
    If an extraneous attribute in found, delete in from a → b
until  F does not change
```

Prime attributes

Attribute set that belongs to any candidate key are called *Prime attributes*.

- If is union of all the candidate key attribute $\{CK_1 \cup CK_2 \cup \dots\}$
- If Prime attribute determined by other attribute set, then more than one candidate key is possible.
- Example: If A is a Candidate key and $X \rightarrow A$, then X is a also a candidate key.

Non-Prime attributes

Attribute set does not belong to any candidate key are called *Non-Prime attributes*.

Tutorial 5.4: *Super Keys and Candidate Keys*

L5.5: *Relational Database Design Part - V*

Lossless Join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R:
  $r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r)$

Example:

- - Consider **Supplier_Parts** schema: **Supplier_Parts(S#, Sname, City, P#, Qty)**
  - Having dependencies: **S#** $\rightarrow$ **Sname**, **S#** $\rightarrow$ **City**, **(S#, P#)** $\rightarrow$ **Qty**
  - Decompose as: **Supplier(S#, Sname, City, Qty): Parts(P#, Qty)**
  - Take Natural Join to reconstruct: **Supplier** $\bowtie$ **Parts**

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10

S#	Sname	City	Qty
3	Smith	London	20
5	Nick	NY	50
2	Steve	Boston	10
5	Nick	NY	40
5	Nick	NY	10

P#	Qty
301	20
500	50
20	10
400	40
301	10

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
5	**Nick**	**NY**	**20**	**10**
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10
2	**Steve**	**Boston**	**301**	**10**

- We get extra tuples! **Join is Lossy!**
- Common attribute **Qty** is not a superkey in **Supplier** or in **Parts**
- Does not preserve **(S#, P#) → Qty**

Example 2:

- Consider **Supplier_Parts** schema: **Supplier_Parts(S#, Sname, City, P#, Qty)**
- Having dependencies: **S# → Sname, S# → City, (S#, P#) → Qty**
- Decompose as: **Supplier(S#, Sname, City): Parts(S#, P#, Qty)**
- Take Natural Join to reconstruct: **Supplier ⋈ Parts**

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10

S#	Sname	City
3	Smith	London
5	Nick	NY
2	Steve	Boston
5	Nick	NY
5	Nick	NY

S#	P#	Qty
3	301	20
5	500	50
2	20	10
5	400	40
5	301	10

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10

- We get back the original relation. **Join is Lossless.**
- Common attribute **S#** is a superkey in **Supplier**
- Preserves all dependencies

Dependency Preservation

- Left F_i be the set of dependencies F^+ that include only attributes in R_i
  - A decomposition is **dependency preserving**, if $(F_1 \cup F_2 \cup \cdots \cup F_n)^+ = F^+$
    - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Example:

- **R** (*A*, *B*, *C*, *D*, *E*, *F*)
  F = $\{A \to BCD, A \to EF, BC \to AD, BC \to E, BC \to F, B \to F, D \to E\}$
- Decomposition: **R1**(A, B, C, D) **R2**(B, F) **R3**(D, E)
  - $A \to BCD$, $BC \to AD$ are preserved on table R1
  - $B \to F$ is preserved on table R2
  - $D \to E$ is preserved on table R3
  - We have to check whether the remaining FDs: **A→E**, **A→ F**, **BC→E**, **BC→** F are preserved or not.

R1

F_1={**A** → *ABCD*, **B** → *B*, **C** → *C*, **D** → *D*,
AB → *ABCD*, **BC** → *ABCD*, **CD** → *CD*, **AD** → *ABCD*
ABC → *ABCD*, **ABD** → *ABCD*, **ACD** → *ABCD*
BCD → *ABCD*}

  - $F' = F_1 \cup F_2 \cup F_3$.
  - Checking for: **A** → *E*, **A** → *F* in F'^+
    - ▷ $A \to D$ (from R1), $D \to E$ (from R3) : **A→ E** (By Transitivity)
    - ▷ $A \to B$ (from R1), $B \to F$ (from R2) : **A→ F** (By Transitivity)
  - Checking for: **BC** → *E*, **BC** → *F* in F'^+

- Checking for: $BC \rightarrow E$, $BC \rightarrow F$ in F
  - ▷ $BC \rightarrow D$ (from R1), $D \rightarrow E$ (from R3) : **BC→ E** (By Transitivity)
  - ▷ $B \rightarrow F$ (from R2) : **BC→ F** (By Augmentation)

Hence all dependencies are preserved.

Tutorials

5.1: ***Lossless Join Decomposition***

5.2: ***Canonical Cover***

5.3: ***Dependency Preservation***