Intension and Extension in Database

- **Intension**: The database schema or blueprint. It defines the structure, data types, constraints, and relationships. It's relatively permanent.
    - ***PDF Visual (Page 2):*** An example shows the SQL definition for a *customer* table (*id INT PRIMARY KEY, name VARCHAR(50)*), representing the intension.
- **Extension**: The actual data (tuples/rows) present in the database at a specific point in time. It changes as data is added, modified, or deleted.
    - ***PDF Visual (Page 3):*** Shows snapshots of an *Employee* table at different time instances (t1, t2, t3) with varying data, illustrating the changing extension.

RDBMS (Relational Database Management System)

- **Summary**: A type of DBMS that stores data in tables (relations).
- **Description**: Data is organized into tables with predefined connections. SQL (Structured Query Language) is used for data manipulation and querying. Examples include MySQL, Oracle, MariaDB. This contrasts with Non-relational databases (like MongoDB) where data isn't stored in tables.
    - ***PDF Visual (Page 4 & 6):*** Basic diagrams show the interaction: User -> Software Application (DBMS) -> Database.
    - ***PDF Visual (Page 5):*** A diagram categorizes Databases into Relational and Non-relational, listing examples for each.

Normalization

- **Summary**: The process of organizing data in a database to reduce redundancy (duplication) and improve data integrity and consistency.
- **Description**: It typically involves dividing large tables into smaller, well-structured tables. The goal is to eliminate undesirable characteristics like Insertion, Deletion, and Updation Anomalies caused by redundancy. Row-level duplication is handled by primary keys, while normalization addresses redundancy within columns across multiple rows.

Anomalies (Caused by Redundancy)

- **Summary**: Errors or inconsistencies that can occur in databases that are not properly normalized.

- **Description**:
    - **Insertion Anomaly**: Difficulty inserting new data because other required data is missing. (e.g., Cannot add a new department if no employees are assigned to it yet in a single large table).
        - ***PDF Visual (Page 9):*** An *Employee* table illustrates that adding a new department requires adding employee details simultaneously.
    - **Deletion Anomaly**: Unintended loss of other data when a record is deleted. (e.g., Deleting the last employee in a department might also delete the only record of that department's existence, manager, etc.).
        - ***PDF Visual (Page 10):*** The *Employee* table shows how deleting all employee records could lose department information.
    - **Updation Anomaly**: Changes to redundant data require updates in multiple places, risking inconsistency if not all are updated. (e.g., Changing a department manager's name requires updating every row for employees in that department).
        - ***PDF Visual (Page 11):*** The *Employee* table demonstrates needing multiple updates to change the salary for the 'HR' department.

How Normalization Helps

- **Summary**: Normalization addresses anomalies by decomposing large tables into smaller, linked tables.
- **Description**: By separating concepts into different tables (e.g., Employee details in one, Department details in another), redundancy is reduced, and anomalies are prevented.
    - ***PDF Visual (Page 12):*** Shows the problematic *Employee* table decomposed into a smaller *Employee* table and a *Department* table, linked by the 'department' attribute.

Types of Normalization

- **Summary**: Normalization involves progressing through different normal forms, each with stricter rules.

- **Description**: The main types covered are First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), and Boyce-Codd Normal Form (BCNF). 4NF and 5NF are also mentioned.

Denormalization

- **Summary**: The process of intentionally introducing redundancy into a normalized database, typically to improve query performance.
- **Description**: It's the opposite of normalization. While normalization reduces redundancy often requiring joins to retrieve related data, denormalization combines tables (re-introducing some redundancy) to make frequently needed data accessible from a single table, potentially speeding up queries.
    - **Benefits**: Faster and simpler queries (fewer joins).
    - **Disadvantages**: Increased redundancy, potential for inconsistency, reduced flexibility.
    - ***PDF Visual (Page 14):*** Illustrates denormalization by showing the previously separated *Employee* and *Department* tables being joined back into a single table to avoid joins when querying, for example, Rahul's salary.

Functional Dependency (FD)

- **Summary**: A constraint between two sets of attributes in a relation, where the value of one set of attributes (determinant) determines the value of another set (dependent).
- **Description**: Notation: X -> Y (read as "X determines Y" or "Y is functionally dependent on X"). If two tuples have the same value for X, they must also have the same value for Y.
    - ***PDF Visual (Page 16):*** A simple diagram shows an arrow from X (determinant) to Y (dependent) within a relation R.
    - ***PDF Visual (Page 17):*** An *Employee* table example shows *EmpId -> EmpFirstName*, meaning EmpId determines the first name.
- **Properties**: Reflexivity (X->X), Augmentation (If X->Y, then XZ->YZ), Transitivity (If X->Y and Y->Z, then X->Z), Union (If X->Y and X->Z, then X->YZ), Decomposition (If X->YZ, then X->Y and X->Z).
- **Types**:

- **Trivial FD**: X -> Y where Y is a subset of X (e.g., {EmpID, EmpFirstName} -> {EmpID}). Always holds true.
- **Non-Trivial FD**: X -> Y where Y is not a subset of X (e.g., {EmpID} -> {EmpFirstName}). Represents a real constraint.

Attribute Closure (X^+)

- **Summary**: The set of all attributes that are functionally determined by a given attribute set X, based on a set of FDs.
- **Description**: Calculated using the given FDs and their properties (reflexivity, transitivity, etc.). It's used to find candidate keys and check dependencies.
  - ***PDF Visual (Pages 22-25):*** A detailed example calculates the closure for attributes A, B, C, D, E given FDs A->B, B->C, C->D, D->E. For instance, A^+ = {A,B,C,D,E}.

Keys (Derived using Attribute Closure)

- **Super Key**: An attribute set whose closure includes *all* attributes in the relation. It uniquely identifies a tuple. Any set containing a candidate key is a super key.
  - ***PDF Visual (Pages 26-31):*** Explains how to find all super keys by calculating closures for all subsets (power set) of the relation's attributes.
- **Candidate Key (CK)**: A *minimal* super key, meaning no proper subset of it is also a super key. A relation can have multiple candidate keys.
- **Primary Key**: One candidate key chosen by the database designer to uniquely identify tuples.
- **Prime Attribute**: An attribute that is part of *any* candidate key.
- **Non-Prime Attribute**: An attribute that is not part of *any* candidate key.

Normal Forms (Detailed Rules & Examples)

First Normal Form (1NF)

- **Rule**: All attribute values must be atomic (indivisible). No repeating groups or multi-valued attributes. Each row must be unique (usually ensured by a primary key).
- **Description**: The most basic level. Focuses on eliminating repeating groups within single rows.
    - ***PDF Visual (Pages 36-39):*** Shows a table with a multi-valued *Order* attribute (*Muffin, Sugar*). It demonstrates three ways to achieve 1NF: 1) Repeating rows for each value, 2) Creating separate columns (Order1, Order2), 3) Decomposing into two tables (Person, Order).

Second Normal Form (2NF)

- **Rules**: Must be in 1NF. Must have *no partial dependencies*.
- **Partial Dependency**: A non-prime attribute is functionally dependent on only a *part* of a candidate key (applies only when CK is composite).
- **Description**: Ensures that non-key attributes depend on the *entire* candidate key.
    - ***PDF Visual (Pages 41-42):*** An example table *(CustomerId, OrderId, OrderName)* has CK *{CustomerId, OrderId}*. The FD *OrderId -> OrderName* shows *OrderName* (non-prime) depends only on part of the CK (*OrderId*), violating 2NF. Decomposition into *(CustomerId, OrderId)* and *(OrderId, OrderName)* resolves this.
    - ***PDF Visual (Pages 43-44):*** An abstract example R(A,B,C,D) with CK={A,B} and FD B->C shows a partial dependency because B is part of the CK and C is non-prime.

Third Normal Form (3NF)

- **Rules**: Must be in 2NF. Must have *no transitive dependencies*.
- **Transitive Dependency**: A non-prime attribute depends on another non-prime attribute, which in turn depends on the candidate key (X->Y->Z where X is CK, Y and Z are non-prime).
- **Alternative Rule**: For every non-trivial FD X -> Y, *either* X is a super key, *or* Y is a prime attribute.
- **Description**: Eliminates dependencies where non-key attributes depend on other non-key attributes.

- ***PDF Visual (Pages 46-47):*** An abstract example R(A,B,C,D) with CK={A,B} and FDs AB->C, C->D shows a transitive dependency (AB->C->D). C is non-prime and determines D (non-prime). This violates 3NF. Applying the alternative rule: for C->D, C is not a superkey, and D is not prime.

Boyce-Codd Normal Form (BCNF)

- **Rules**: Must be in 3NF. For *every* non-trivial functional dependency X -> Y, X must be a super key.
- **Description**: A stricter version of 3NF. It handles certain rare anomalies not addressed by 3NF. The key difference is that BCNF does not allow the "Y is a prime attribute" exception permitted in 3NF.
    - ***PDF Visual (Pages 49-50):*** An abstract example R(A,B,C,D) with CK={A,B} and FDs AB->C, AB->D is shown to be in BCNF because the determinant (LHS) in both FDs (AB) is a superkey (specifically, the candidate key).

Fourth Normal Form (4NF)

- **Rules**: Must be in BCNF. Must have *no multi-valued dependencies* (MVDs).
- **Multi-valued Dependency (MVD)**: Occurs when the presence of one row implies the presence of other rows. If X ->-> Y holds, it means that for a given X value, the set of associated Y values is independent of the set of associated values for other attributes Z (where Z = R - X - Y). Represented as X ->-> Y.
- **Description**: Addresses redundancy arising from independent multi-valued facts stored in the same table.
    - ***PDF Visual (Pages 62-63):*** An example *Student(StudentID, Course, PhoneNbr)* shows MVDs *StudentID ->-> Course* and *StudentID ->-> PhoneNbr*. A student can have multiple courses and multiple phone numbers independently. This causes redundancy. Decomposition into *(StudentID, Course)* and *(StudentID, PhoneNbr)* achieves 4NF.

Fifth Normal Form (5NF) / Project-Join Normal Form (PJNF)

- **Rules**: Must be in 4NF. Must have *no join dependencies* that are not implied by the candidate keys. A table is in 5NF if every join dependency is implied by the candidate keys.

- **Join Dependency (JD)**: A generalization of MVD. A table R satisfies a join dependency *(R1, R2, ..., Rn)* if R is always equal to the join of its projections R1, R2, ..., Rn.
- **Description**: The highest level of normalization, dealing with complex redundancies that can only be removed by decomposing into three or more tables. Ensures the decomposition is lossless when joined back.
    - ***PDF Visual (Pages 65-66):*** Repeats the lossless decomposition example R(A,B,C) -> R1(A,B), R2(A,C) to illustrate the concept of being able to losslessly join decomposed tables back, a requirement for 5NF.

Dependency Preserving Decomposition

- **Summary**: A decomposition where all original functional dependencies can still be enforced by checking dependencies within the individual decomposed tables.
- **Description**: It ensures that the constraints represented by the FDs are not lost after breaking down the table. If the union of FDs logically implied by the schemas of the decomposed tables is equivalent to the original set of FDs, the decomposition is dependency preserving.
    - ***PDF Visual (Pages 51-52):*** Examples show checking if FDs from the original relation hold within the decomposed relations.

Lossy vs. Lossless Decomposition

- **Lossy Decomposition**: When a table is decomposed, and joining the decomposed tables back results in spurious (extra, incorrect) tuples or fails to reconstruct the original table exactly. Data/Associations are lost.
    - ***PDF Visual (Pages 54-55):*** An example R(A,B,C) is decomposed into R1(A,B) and R2(B,C). Joining them back produces extra rows not in the original R, indicating a lossy decomposition.
- **Lossless (or Lossless-Join) Decomposition**: When a table is decomposed, the natural join of the decomposed tables yields exactly the original table, without loss of information or creation of spurious tuples.
    - **Conditions for Lossless Join (for decomposition into R1, R2)**:
        1. The union of attributes in R1 and R2 must equal the attributes in R (R1 ∪ R2 = R).
        2. The intersection of attributes (R1 ∩ R2) must not be empty.

3. The common attribute(s) (R1 ∩ R2) must be a super key for *either* R1 *or* R2.

- ***PDF Visual (Pages 59-60):*** An example R(A,B,C) with CK=A is decomposed into R1(A,B) and R2(A,C). The common attribute A is the CK for both R1 and R2. Joining them back correctly reconstructs the original R, demonstrating a lossless decomposition.

Finding the Highest Normal Form

- **Steps**:
  1. Identify all Candidate Keys (using attribute closure).
  2. Identify Prime and Non-Prime attributes.
  3. Check for 1NF (assumed).
  4. Check for 2NF (no partial dependencies).
  5. Check for 3NF (no transitive dependencies OR use X->Y rule).
  6. Check for BCNF (for all X->Y, X must be a super key).
  7. (If applicable) Check for 4NF (no MVDs).
  8. (If applicable) Check for 5NF (no problematic join dependencies).
- **Example**:
  - ***PDF Visual (Pages 68-71):*** A detailed walkthrough finds the highest normal form for R(A,B,C) with FDs A→BC, B→C, A→B, AB→C, B→A. It identifies CKs {A}, {B}, checks each normal form's conditions, and concludes the relation is in BCNF.

How to Normalize a Table

- **Steps**:
  1. Start with the table and its FDs. Identify CKs, prime/non-prime attributes.
  2. Ensure 1NF (atomic values).
  3. Identify and eliminate partial dependencies (for 2NF) by decomposing. Create new tables where non-prime attributes depend on the part of the key they are determined by.

4. Identify and eliminate transitive dependencies (for 3NF) by further decomposing. Create new tables to separate attributes that depend on other non-key attributes.
5. Identify and eliminate remaining anomalies for BCNF (where determinant X is not a superkey) by decomposing.
6. Ensure decompositions are lossless (typically by ensuring common attributes form a key for one of the resulting tables) and ideally dependency preserving.

- **Example**:
    - ***PDF Visual (Pages 73-75):*** Normalizes R(A,B,C,D) with FDs A→B, B→C, C->D. It finds CK=A, identifies transitive dependencies B->C and C->D violating 3NF, and decomposes into R1(A,B), R2(B,C), R3(C,D), showing these are in BCNF.

Minimal Cover (Canonical Cover) of FDs

- **Purpose**: To find the smallest equivalent set of FDs with no redundancy. Simplifies dependency analysis and normalization.
- **Properties**: No extraneous attributes on LHS, no redundant FDs, singleton RHS.
- **Steps**:
    1. **Decompose RHS**: Ensure every FD has only a single attribute on the right side (using the decomposition property).
    2. **Remove Redundant FDs**: For each FD X->Y, check if it can be derived from the *other* FDs in the set (by computing X^+ using the other FDs). If Y is in X^+ without using X->Y itself, then X->Y is redundant and can be removed.
    3. **Remove Extraneous LHS Attributes**: For each FD with a composite LHS (e.g., XZ -> Y), check if any attribute on the LHS is extraneous. Check if Z can be removed (i.e., does X -> Y hold based on the current FD set?). Check if X can be removed (i.e., does Z -> Y hold?). Remove any extraneous attributes.
- **Example**:
    - ***PDF Visual (Page 78):*** Finds the minimal cover for {A→BC, B→C, A→B, AB→C}. Step 1 yields {A→B, A→C, B→C, A→B, AB→C}. Step 2 removes redundant A→C and AB→C. Step 3 (not needed here as LHS are single attributes). Final minimal cover: {A→B, B→C}.

Equivalence of Functional Dependencies

- **Definition**: Two sets of FDs, F and G, are equivalent if F can derive all FDs in G (F covers G), *and* G can derive all FDs in F (G covers F).
- **How to Check**:
    1. For every FD X->Y in G, compute X^+ using *only* the FDs in F. Check if Y is in X^+. If this holds for all FDs in G, then F covers G.
    2. For every FD X->Y in F, compute X^+ using *only* the FDs in G. Check if Y is in X^+. If this holds for all FDs in F, then G covers F.
    3. If both steps pass, F and G are equivalent.
- **Example**:
    - ***PDF Visual (Page 81):*** Checks if F={A→B, B→C} and G={A→C, A→B} are equivalent. Finds A^+ using F is {A,B,C}, so F covers G (A->C and A->B hold). Finds A^+ using G is {A,B,C}, B^+ using G is {B}, C^+ using G is {C}. Since B^+ using G does not contain C, G does *not* cover F (specifically, B->C is not implied by G). Therefore, F and G are not equivalent.