Data Engineering Notes

ibrahim.nasser@fau.de

May 23, 2025

Disclaimer

These are my personal notes and are not official course documents. They may contain inaccuracies or omissions, hence, they should not be considered as a substitute for official course materials or as comprehensive preparation for examinations.

Contents

1 Introduction and Basic Data Types

We call the data **Tabular** when there are no modelled dependencies between attributes, for example, demographic attributes such as age, gender, ZIP code, etc. (also called *Nondependency-Oriented Data*). Otherwise it is **Non-Tabular**, e.g. social networks, time series, etc.

Matrix Representation of Data

A set $X = \{X_i \mid i \in \{1 \dots n\}\}$, with n records (samples) is a d-dimensional dataset iff each sample X_i is a set of $\{x_j \mid j \in \{1 \dots d\}\}$ attributes (features). X is tabular if it is invariant w.r.t shuffling of samples and features. Each feature x_j has its own domain $\mathcal{D}_j$

Quantitative vs. Categorical: A variable x is quantitative (numeric) if its domain $\mathcal{D}_x$ is numeric. Otherwise, Categorical. *Examples (Q):* age, weight, height, BMI, Date of Birth. *Examples (C):* name, gender, country, ZIP Code, weather, ID, day.

Nominal vs. Ordinal: A categorical variable x is ordinal if its domain $\mathcal{D}_x$ has a natural ordering. Otherwise, Nominal. *Examples (N):* weather, name, gender, country, ZIP, ID, day *Examples (O):* heat level, textual gpa.

Finite vs. Infinite: A variable x has a finite domain iff $|\mathcal{D}_x| = N, N \in \mathbb{N}$. Otherwise, Infinite. *Examples (F):* age (years), country, ZIP, ID, gender, day. *Examples (I):* BMI, height, Date of Birth.

> Note
>
> All categorical variables have finite domains, not the other way around.

Discrete vs. Cont.: A Quantitative variable x is continuous iff $\forall z, y \in \mathcal{D}_x \exists w \in \mathcal{D}_x, z < w < y$. Otherwise, Discrete. *Examples (D):* age (years, months, days, hours, etc). *Examples (C):* age (unitless, number), Date of Birth (point in cont. time), BMI.

> Note
>
> By **rounding** quantitative data, we can transform cont. domains into discrete ones.

> Note
>
> Age is quantitative finite discrete if it is computed as whole years, months, days, hours. However, it is quantitative infinite continuous it is computed as precise value including fractions

> Note
>
> Date of Birth is quantitative infinite continuous since it is a point in a continuous endless time

Binary: We call a variable x binary iff $|\mathcal{D}_x| = 2$

Temporal: We call a variable x temporal iff $\mathcal{D}_x$ represents time points or intervals. *Examples:* day, month, Date of Birth

Encoding: Data Encoding refers to the technique of converting data into a form that allows it to be properly used by different systems.

Binning: Binning is an encoding technique that is a function $f : \mathcal{D} \to \{1 \dots K\}$

Example: Equal-Width Binning: Size (width) of each bin is calculated as $W = \frac{\text{Max}(x) - \text{Min}(x)}{K}$ where K is the number of bins.

One-Hot Encoding

To mitigate the problem of label encoding for nominal variables.
How? Create a fixed-size vector with size = |unique(x)|, where each position corresponds to a unique category value. Assign a `1` to the position representing the category and `0`s elsewhere.

Example: Suppose unique(x) = {Red, Green, Blue}

- Red → [1, 0, 0]
- Green → [0, 1, 0]
- Blue → [0, 0, 1]

Note

One-hot encoding avoids the problem of implying ordinal relationships. However, it increases dimensionality significantly, especially when the number of categories is large (curse of dimensionality).

Cyclic Encoding

Some categorical variables are *ordinal* and have a natural *cyclic* structure. A classic example is the months of the year:

$$\mathcal{D}_x = \{\text{Jan}, \text{Feb}, \ldots, \text{Dec}\}$$

This variable has both an order (Jan < Feb < ... < Dec) and a cyclic relationship (Dec is followed by Jan).

To encode this properly, we use the index i of each category in the ordered list, where $i = 1, 2, \ldots, k$, and k is the total number of categories.

Encoding Function:

$$\text{enc}(c_i) = (x_i, y_i)$$

$$x_i = \cos\left(\frac{2\pi(i-1)}{k}\right), \quad y_i = \sin\left(\frac{2\pi(i-1)}{k}\right)$$

This maps each category to a unique point on the unit circle, preserving both order and cyclicity.

Note

Cyclic encoding is useful when the first and last categories are conceptually adjacent (e.g., December and January). This is not possible with standard label or one-hot encoding.

Optional: Normalize to Unit Square

$$\text{enc}(c_i) = \left(\frac{x_i + 1}{2}, \frac{y_i + 1}{2}\right)$$

This scaled version maps points to the square $[0, 1] \times [0, 1]$, which can be useful when input normalization is required for machine learning models. Note that this transformation alters the original unit circle geometry.

Note

Use raw unit circle encoding when preserving angular distance is important. Use the normalized version when the model expects features in the range $[0, 1]$.

Non-Tabular Data

Such as Spatial data, images, time series, string, graphs.

A set $X = \{x_i \mid i \in \{1 \ldots n\}\}$ is a d-dimensional **spatial** dataset with n samples if each sample x_i contains a set of $\{x_j \mid j \in \{1 \ldots d\}\}$ features AND each data point x_{ij} is associated with a specific spatial location l.

A spatial location l can be a point $(l_x, l_y) \in \mathbb{R}^2$ (2D spatial data) or $(l_x, l_y, l_z) \in \mathbb{R}^3$ (3D spatial data), etc.

Tokenization (Character-Level)

Tokenization is the process of converting raw text into smaller units called tokens. In character-level tokenization, each unique character from the corpus is treated as a token.

Example: Consider the corpus consisting of a single sentence: `"hi ai"`

- Unique characters: {`h, i,  , a`}
- Assign token IDs: `h:0, i:1, :2, a:3`
- Tokenized sentence: `"hi ai"` → `[0, 1, 2, 3, 1]`

Each character in the sentence is replaced by its corresponding token ID.

Graphs

A graph is a mathematical structure used to model pairwise relations between objects.

- A graph G is defined as $G = (V, E)$, where:
  - V is a set of *vertices* (or *nodes*).
  - $E \subseteq V \times V$ is a set of *edges*.

Types of Graphs:

- **Undirected Graph:** An edge $(u, v) \in E$ implies a bidirectional connection:

  $$(u, v) \in E \Rightarrow (v, u) \in E$$

- **Directed Graph (Digraph):** Edges have direction:

  $$(u, v) \in E \not\Rightarrow (v, u) \in E$$

Graph Representations

Adjacency Matrix:

A $|V| \times |V|$ matrix A, where:

$$A[u][v] = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

- **Space consumption:** $\mathcal{O}(|V|^2)$
- **Edge access:** $\mathcal{O}(1)$
- **Neighbor iteration:** $\mathcal{O}(|V|)$

Adjacency List:

Each vertex $u \in V$ maintains a list of its neighbors.

- **Space consumption:** $\mathcal{O}(|V| + |E|)$
- **Edge access:** $\mathcal{O}(|V|)$ (worst-case search)
- **Neighbor iteration:** $\mathcal{O}(\deg(u))$, where $\deg(u)$ is the degree of vertex u

Weighted Graphs:

In some graphs, each edge $(u, v) \in E$ is associated with a numerical value called a *weight*, often representing cost, distance, capacity, etc.

- For weighted graphs, the edge set becomes:

  $$E \subseteq V \times V \times \mathbb{R}$$

  or we define a weight function:

  $$w : E \to \mathbb{R}$$

- In the adjacency matrix, $A[u][v]$ stores the weight instead of a binary 0 or 1.
- In the adjacency list, each neighbor can be stored along with its edge weight as a tuple: $(v, w(u, v))$.

2 Conceptual Modeling

ER Model: Entity-Relationship Model is a high-level, conceptual framework to describe entities, their attributes, and the relationships between them.

Entity: Basic concept of the Entity-Relationship (ER) model. It is an object in the real world. E.g. e1 (some employee).

Attribute: Entities have attributes that are the properties that describe them.

Entity Type: All entities that have the same entity type share the same attributes. E.g. EMPLOYEE (type), e1 (Entity).

Attribute Value: A particular entity has a specific value for each of its attributes.

Composite: An attribute is composite if it is described in terms of its smaller parts. E.g. Name, some databases consider name as a composite attribute consisting of two **atomic** attributes First Name and Last Name.

Atomic/Simple: Cannot be divided into smaller parts.

Note

These days, we store **date** as a single value attribute of the type *DATE*. Earlier, date was considered as a composite attribute consisting of atomic attributes *day, month, year*.

Derived: An attribute is derived if its value is calculated using other **stored** attributes, e.g. age.

Stored: An attribute is stored if it cannot be derived from other attributes.

Note

Age is both derived and atomic. DateOfBirth is both composite and stored.

Single-Valued: An attribute is single-valued if it can have only one value. E.g. DateOfBirth is single-valued composite. Biological sex is single-valued atomic.

Multi-Valued: An attribute is multivalued if it can have several values. E.g. college degrees is multivalued atomic and can have BSc, MSc, BEng, etc.

Note

Affiliation of an entity type RESEARCHER is multivalued (because one can have different affiliations) and composite because an affiliation could be represented as (Org. Name, Dept., Address, Role, Start Date, End Date).

Entity Set: Collection of entities of a particular entity type in a database in a given time point.

Entity Type	Blueprint/Description
Entity Set	Actual set of entities (entity instances) at a point in time

Note

In ancient logic and philosophy we refer to the definition or conceptual content of a term as an *intension*. However, the set of actual things that satisfy a concept is called *extension*. Hence, Entity Type is called intension, Entity Set is called extension.

Candidate Key (Key Attribute): A candidate key is an attribute (or set of attributes) that **uniquely** and **minimally** identifies each entity in an entity set. E.g. StudentID, studentEmail.

Primary Key: A primary key is the chosen candidate key that will be used to uniquely identify entities in the database.

Foreign Key: A foreign key is an attribute in one table/entity that references the primary key of another table/entity. It expresses a relationship between two entity sets.

Composite Primary Key: A composite primary key is a primary key that consists of two or more attributes combined together to uniquely identify a record in a table. Neither attribute alone is sufficient to guarantee uniqueness — but together, they do.

This is common in relationship tables, for example: enrollment relationship between students and courses (M:M):

StudentID	CourseID	Grade
101	CS101	A
101	MATH201	B
102	CS101	B+

Weak Entity Types: Entity types without key attributes.

Strong Entity Types: Entity types with key attributes.

Relations

If we want to model 1:M or M:1 relations, we use the idea of foreign key (modeling the relation with single value attribute). Examples:

PersonID	PersonName
1	Alice
2	Bob

categoryID	categoryName
1	Sport
2	Science

catID	catName	ownerID
1	Daisy	1
2	Smart	2
3	Sweet	1

articleID	title	categoryID
1	title	1
2	title	2
3	title	1

We model M:M Relations by creating an entity representing that relation (usually with composite primary key).

Relationship Type: The definition / template / blueprint of the relationship

Relationship Instance: A single actual link between entities.

Relationship Set: The collection of all relationship instances at a given time.

Participation: We say that entity types $E_1 \dots E_n$ participate in the **relationship type** R.

Relationship Degree: Number of participating entity types in the relation. E.g. consider a relation SUPPLY that models which suppliers supply which projects and what parts are supplied, the degree here is 3 due the three entity types (SUPPLIER, PROJECT, PART).

Role Name: The name describing the part an entity plays in a relationship.

Recursive Relationship: A relationship where the same entity type participates more than once with different roles. E.g. SUPERVISION.

Cardinality Ratio: Specifies the maximum number of entities of one type that can be associated with an entity of another type in a relationship. Examples (E_1 BINARY_RELATION E_2):

Let E_1 and E_2 be the sets of entities of type E_1 and E_2, respectively. Let $R \subseteq E_1 \times E_2$ be the binary relation between them: $R = \{(e_1, e_2) \mid e_1 \in E_1,\ e_2 \in E_2\}$

- **1:1** each entity of type E_1 can be related to at most one entity of type E_2 and vice versa

  $$\forall e_1 \in E_1, \quad |\{e_2 \in E_2 \mid (e_1, e_2) \in R\}| \leq 1$$

  $$\forall e_2 \in E_2, \quad |\{e_1 \in E_1 \mid (e_1, e_2) \in R\}| \leq 1$$

- **1:M** one entity of type E_1 can be related to many entities of type E_2. Each entity of type E_2 can be related to at most one entity of type E_1

  $$\forall e_1 \in E_1, \quad |\{e_2 \in E_2 \mid (e_1, e_2) \in R\}| \geq 0$$

  $$\forall e_2 \in E_2, \quad |\{e_1 \in E_1 \mid (e_1, e_2) \in R\}| \leq 1$$

- **M:1** Inverse of **1:M**

  $$\forall e_1 \in E_1, \quad |\{e_2 \in E_2 \mid (e_1, e_2) \in R\}| \leq 1$$

  $$\forall e_2 \in E_2, \quad |\{e_1 \in E_1 \mid (e_1, e_2) \in R\}| \geq 0$$

- **M:M** many entities of type E_1 can be related to many entities of type E_2 and vice versa

  $$\forall e_1 \in E_1, \quad |\{e_2 \in E_2 \mid (e_1, e_2) \in R\}| \geq 0$$

  $$\forall e_2 \in E_2, \quad |\{e_1 \in E_1 \mid (e_1, e_2) \in R\}| \geq 0$$

Total vs. Partial Participation

- **Total Participation of** E_1 in R
  $\forall e_1 \in E_1,\ \exists e_2 \in E_2 : (e_1, e_2) \in R$ (Every entity of E_1 is related to at least one entity of E_2.)
- **Partial Participation of** E_1 in R
  $\exists e_1 \in E_1 : \forall e_2 \in E_2, (e_1, e_2) \notin R$ (There exists an entity in E_1 that does not participate in R.)

Migrating attributes from relations to entities:

- 1:1 relationship types: Attributes can be migrated to either participating entity. (e.g. EMP MANAGES DEPT, start_date)
- 1:N or N:1 relationship types: Attributes should be migrated to the entity that participates at most once. (e.g. EMP WORKS_FOR DEPT, start_date)
- M:N relationship types: Attributes cannot be migrated to the participating entities and must remain on the relationship itself.

Identifying Relationships: a special relationship where a weak entity is identified by its relationship with a strong entity. The weak entity cannot exist without the strong entity, and the relationship plays a crucial role in providing the weak entity with a composite key.

Existence Dependency: A weak entity depends on the strong entity for its existence. It cannot exist without being related to a strong entity.

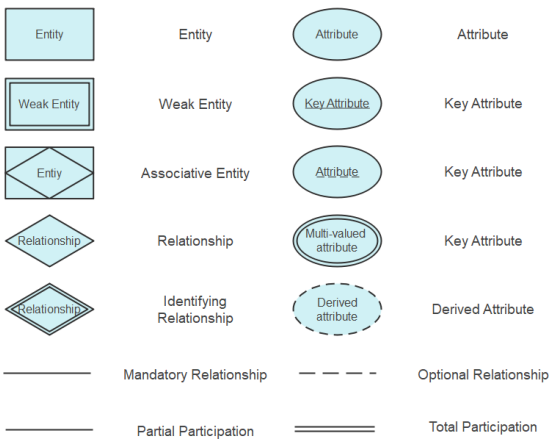
> **Note**
>
> A weak entity type has total participation in its identifying relationship. This means that every instance of the weak entity must be associated with at least one instance of the strong entity. If it doesn't, the weak entity doesn't exist ("existence dependency").

Example:

- Consider we have the following strong entities, customer, product.
- To manage orders, we have two entities, order, orderItem.
- Order is a strong entity since each order has its ID
- However, OrderItem entity can have OrderID, LineNumber, ProductID, quantity, price, discount, etc.
- In this case, OrderItem entity is a weak one, it cannot exist unless an order exists, therefore, the primary key is composite (OrderID, LineNumber)

Min-Max Modeling: Given an entity E participating in Relation R. If at least min and at most max instances of E must participate in R with $min >= 0, max >= 1, max >= min$, then we say E respects min-max constraint (min, max) w.r.t R.

ER Diagram



3 The Relational Data Model

Set: A set is a well-defined collection of distinct objects, considered as an object in its own right. The objects in a set are called elements or members. Sets are usually denoted by capital letters like A, B, or S, and elements are listed within curly braces. For example, $A = \{1, 2, 3\}$ is a set containing the numbers 1, 2, and 3.

Element of a Set: If x is an element of set A, we write $x \in A$. If x is not in A, we write $x \notin A$.

Set-builder Notation: Set-builder notation is a shorthand used to describe a set by stating the properties that its elements must satisfy. For example: $\{x \in \mathbb{N} \mid x \text{ is even}\}$ describes the set of even natural numbers.

Cardinality: The cardinality of a set is the number of elements in the set, denoted $|A|$. For example, if $A = \{1, 2, 3\}$, then $|A| = 3$.

Cartesian Product: Let A and B be sets, then $A \times B = \{(a, b) \mid a \in A, b \in B\}$, e.g. $A = \{1, 2\}, B = \{x, y\}, A \times B = \{(1, x), (1, y), (2, x), (2, y)\}$

Subset: $A \subseteq B \Leftrightarrow \forall X.X \in A \Rightarrow X \in B$

Proper Subset: $A \subset B \Leftrightarrow A \subseteq B \wedge A \neq B$

Relation: $R \subseteq A \times B$. If $(a, b) \in R$, we say that a is related to b via R

Left Total Relation: A relation $R \subseteq A \times B$ is left total (total on A) iff each element in A is related to at least on element in B. $\forall a \in A.\exists b \in B.(a, b) \in R$

Right Unique Relation: A relation $R \subseteq A \times B$ is right unique iff each element in A is related to at most one element in B. $\forall a \in A, \forall b1, b2 \in B, ((a, b1) \in R, (a, b2) \in R) \Rightarrow b1 = b2$

Function: A function is left total and right unique relation. $f : A \to B$

Partial Function: A partial function is a right unique relation, **not** necessarily left total. $f \rightharpoonup B$

Set Union: $x \in A \cup B \Leftrightarrow x \in A$ or $x \in B$

Set Intersection: $x \in A \cap B \Leftrightarrow x \in A$ and $x \in B$

Disjoint: We say two sets A, B are disjoint iff $A \cap B = \phi$

Relation Schema: A declaration $R\big(A_1{:}D_1, \ldots, A_n{:}D_n\big)$ consisting of a name R, a finite, non-empty attribute set $\{A_i\}$ and, for each attribute, its domain $\text{dom}(A_i) = D_i$.

Schema Satisfaction: A *tuple* $t = (v_1, \ldots, v_n)$ *satisfies* the schema if $v_i \in D_i \ \forall i$.

Types: *Classes* of atomic values that share representation and operations, e.g. `Int`, `Real`, or `String`.

Domain: A *set* of atomic values with application-specific semantics whose underlying implementation type is fixed. Domains may define default values. Example: $\mathit{EmployeeAge} = \texttt{Int}[18, 65]$.

> Note
>
> Domain declaration examples: `Name = String(20)`, `DollarPrice = Decimal(5,2)`.

Instance: A *finite set* of tuples that all satisfy a given relation schema. While the schema is comparatively stable (static), its instance is *dynamic*: it evolves through insertions, deletions, and updates.

Two Equivalent Views on Tuples

- **Positional (Cartesian-product) view:** t is an *ordered list* $\big(v_1, \ldots, v_n\big)$. Column order carries meaning; attribute names are implicit.
- **Functional view:** Fix $A = \{A_1, \ldots, A_n\}$ and $D = \bigcup_i D_i$. Then a tuple is a *function* $t : A \to D$ with $t(A_i) \in D_i$. Here, order is irrelevant and attribute names are explicit.

Domain Constraint: Each attribute value must lie in its declared domain D_i. Usually enforced by the DBMS type checker.

Functional Dependency (FD): For attribute sets $X, Y \subseteq A$, the notation $X \to Y$ states: for any two tuples t_1, t_2, equality of X-values implies equality of Y-values. Written out: $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$.

Superkey: An attribute set K with $K \to A$ (it functionally determines the whole tuple).

Candidate Key: A *minimal* superkey — removing any attribute from it destroys the functional determination of A.

Primary Key: The candidate key chosen by the database designer to serve as the principal identifier of tuples in a relation. Remaining candidate keys are called *alternate keys*.

Example

- Relation schema *Employee*(*EmpID*, *SSN*, *Email*, *Name*, *Dept*).
- *Superkeys* include any attribute set that uniquely identifies tuples, e.g $\{EmpID\}$, $\{SSN\}$, $\{EmpID, Name\}$. The third set still determines the whole tuple but is *not* minimal.
- *Candidate keys*: the minimal superkeys $\{EmpID\}$ and $\{SSN\}$. Each is irreducible.
- *Primary key*: suppose we designate *EmpID* as the primary key. The other candidate becomes an *alternate key* available for unique look-ups.

Foreign Key: Attribute(s) in relation R whose values must also appear as the primary-key values of another relation S (ensuring referential integrity).

Note

When an insertion, deletion or modification would break any constraint, the DBMS may (i) reject the change or (ii) repair it automatically ("cascade", insert default/null, etc.). The exact behaviour is part of the schema definition.

A word on modeling different cardinalities

Relational databases use foreign keys (FKs) to represent associations between entities. The modeling depends on the cardinality:

One-to-One (1:1)

A FK is placed in one of the tables.

Person	ID (PK)	Name	PassportID (FK)
	1	Alice	101

Passport	ID (PK)	Number	DateOfIssue
	101	X1234	2020-01-01

One-to-Many (1:M) or Many-to-One (M:1)

The FK is placed in the table on the "many" side, referencing the "one" side.

Department	ID (PK)	Name
	1	Human Resources

Employee	ID (PK)	Name	DepartmentID (FK)
	101	John	1

Many-to-Many (M:M)

Modeled via a relation table with two FKs, each referencing one of the related tables. The combination of FKs often serves as the primary key.

StdID	StdName
1	Jane
2	Mark
3	Sara

CourseID	CourseTitle
10	Database Systems
11	Operating Systems
12	Algorithms

StdID	**CourseID**
1	10
1	12
2	10
2	11
3	11

4 Functional Dependencies

Prime Attribute: An attribute that is part of any candidate key.

Nonprime Attribute: An attribute that is not part of any candidate key.

Example: consider the simple relation `STUDENT(ID, Email, Name, Phone, CourseID)`. Possible super keys:

$$\{\texttt{ID}\}, \{\texttt{Email}\}, \{\texttt{ID}, \texttt{Name}\}, \{\texttt{ID}, \texttt{Email}, \texttt{Phone}\}$$

Only $\{\texttt{ID}\}$ and $\{\texttt{Email}\}$ are prime attributes.

Trivial FD: We say that a functional dependency $X \to Y$ is **trivial** iff $Y \subseteq X$.

Full FD: A functional dependency $X \to Y$ is full iff for any $A \in X$, $(X - \{A\}) \to Y$ does not hold, i.e. you cannot remove any attribute from X without breaking the dependency. Example: {studentID, CourseID} → Grade.

Partial FD: A functional dependency $X \to Y$ is partial iff $\exists A \in X.\ (X - \{A\}) \to Y$ holds.

Transitive FD: A functional dependency $X \to Y$ is transitive in a relation R iff $\exists$ **Nonprime set of attributes** $Z \in R$ and both $X \to Z$ and $Z \to Y$ hold.

Inference: A functional dependency $X \to Y$ is **inferred** from a set of functional dependencies F on a relation R iff $X \to Y$ holds in every instance of R that satisfies all dependencies in F

Armstrong's Inference Rules for Functional Dependencies

- **IR1**: $(Y \subseteq X) \Rightarrow (X \to Y)$ (reflexive)
- **IR2**: $(X \to Y) \Rightarrow (X \cup Z \to Y \cup Z)$ (augmentation)
- **IR3**: $((X \to Y) \wedge (Y \to Z)) \Rightarrow X \to Z$ (transitive)

Closure: The closure of attribute set X under a set of functional dependencies F, denoted as X_F^+ is the set of all attributes that X can determine using FDs in F. $X_F^+ = \{A \mid X \to A \in F \text{ or can be inferred from it}\}$

Closure Algorithm

- input: a set F of FDs on a relation R, and a set of attributes X contained in R
- initialization: $X_F^+ = X$
- changed = True
- while changed:
  1. changed = False
  2. for each FD $Y \to Z \in F$:
     (a) If $(Y \subseteq X_F^+) \wedge (Z \notin X_F^+)$:
        i. $X_F^+ = X_F^+ \cup \{Z\}$
        ii. changed = True
- Output: X_F^+

FDs Verification: F implies $X \to Y$ *iff* $Y \subseteq X_F^+$

Superkeys Verification: X is a super key for R with attribute set U *iff* $X_F^+ = U$

Finding Candidate Keys

```
Input: Relation (R) over set of attributes (U), set of FDs (F)
initialization: K:= U
minimal = False
while not minimal
    minimal = True
    for each attribute A in K
        compute closure of (K-A) under F
            if the closure = U
                set K := K - {A}
                minimal = False
Return: K
```

Coverage: For any two sets of functional dependencies F_1, F_2, we say that F_1 covers F_2, *iff* $\forall X \to Y \in F_2.Y \subseteq X^+_{F_1}$.

Equivalence: For any two sets of functional dependencies F_1, F_2, we say they are equivalent, *iff* they cover each other. Example: verify whether the following FDs are equivalent.

- $F_1 = \{A \to C, AC \to D, E \to AD, E \to H\}$ and
- $F_2 = \{A \to CD, E \to AH\}$

We check first if F_1 covers F_2

- considering $A \to \{C, D\}$ $\quad \{C, D\} \in A^+_{F_1} = \{A, C, D\}$
- considering $E \to \{A, H\}$ $\quad \{A, H\} \in E^+_{F_1} = \{E, A, D, H, C\}$
- Hence F_1 covers F_2

Next, we check first if F_2 covers F_1

- considering $A \to C$ $\quad C \in A^+_{F_2} = \{C, D\}$
- considering $\{A, C\} \to D$ $\quad D \in \{A, C\}^+_{F_2} = \{A, C, D\}$
- considering $E \to \{A, D\}$ $\quad \{A, D\} \in E^+_{F_2} = \{A, H, C, D\}$
- considering $E \to H$ $\quad H \in E^+_{F_2} = \{A, H, C, D\}$
- Hence F_2 covers F_1

Therefore, they are equivalent

Redundancy: A functional dependency $f = X \to A$ is redundant in FDs set F *iff* $A \subseteq X^+_G$ where $G = F - \{X \to A\}$, i.e. $F - \{f\}$ implies f.

Extraneous: Given a set F of FDs and one $f = AX \to B \in F$, then A is extraneous if $B \subseteq X^+_F$

Minimal cover: A set of FDs F is a minimal cover of a set of FDs E iff F covers E and there is no $f \in F$. $\quad F - \{f\}$ covers E

Canonical: A functional dependency $f = X \to Y$ is in a canonical form iff $|Y| = 1$

Minimal set of FDs: A set F of FDs is minimal iff it satisfies the following conditions: (i) All FDs in a canonical form. (ii) No extraneous attributes. (iii) No redundant FDs.

Steps to Obtain a Minimal Set of Functional Dependencies

1. Transform to Canonical form
2. Remove Extraneous Attributes
3. Remove Redundant FDs

5 Relational Algebra and SQL

Data Model: In relational databases, the data model specifies how data is structured and how it can be manipulated. i.e. it says that data is organized into tables (called relations) with columns (attributes) and rows (tuples).

Relational model: In relational databases, the relational model represent data as relations (tables). Each relation has constraints, such as keys or data types, to ensure data integrity.

Relational Algebra: The formal system for manipulating relations. It provides a theoretical foundation for **Query** operations used in relational databases

Algebra: A formal system in which expressions are constructed using operators and atomic operands. These expressions can be evaluated, and two expressions are considered equivalent if they yield the same result for all possible values of their operands.

Relational Algebra: A type of algebra where the operands are relations (tables), and the operators are defined for any instance of those relations. Operations can be combined to form complex expressions, and evaluating an expression produces a result schema (the structure of the output) and a result instance (the actual data produced).

SQL: The Standard Query Language (SQL) is the language used to interact with relational databases. It is a **declarative** language, meaning that when you write a query, you describe what result you want, not how the database should compute it. This contrasts with procedural languages, where you must specify every step.

SQL Structure

SQL is organized into several sub-languages, each serving a distinct purpose:

- Data Definition Language (DDL): used to define or alter the structure of database objects. Commands used such as: `CREATE, ALTER, DROP`
- Data Manipulation Language (DML): used to retrieve and manipulate data. Command used such as: `SELECT, UPDATE, INSERT, DELETE`
- Data Control Language (DCL): manages user permissions and access control. Commands used such as: `GRANT, REVOKE`
- Transaction Control Language (TCL): manages database transactions. Commands used such as: `COMMIT ROLLBACK`

Example DML Queries:

```
SELECT name, age FROM Student WHERE age >= 18 ORDER BY name ASC;

INSERT INTO Student (stdId, name, age) VALUES (101, "Alice", 20);

UPDATE Student SET age = age + 1 WHERE stdId = 101;

DELETE FROM Student WHERE stdId = 101;
```

Examples on `COUNT, DISTINCT, EXIST, IN`

```
SELECT COUNT(*) FROM Student;

SELECT COUNT(DISTINCT stdID) FROM Student;

SELECT * FROM employees e
WHERE EXISTS (
    SELECT 1
    FROM bonus b
    WHERE b.employee_id = e.employee_id
);

SELECT * FROM employees WHERE department_id IN (1, 2, 5);
```

Example DDL Queries:

```
    CREATE TABLE Course (courceID INT PRIMARY KEY,
                         title    VARCHAR(100)
                        );

    ALTER TABLE Course ADD COLUMN credits INT;

    DROP TABLE Course;
```

Example DCL Queries:

```
    GRANT SELECT, INSERT ON Student TO user1;

    REVOKE INSERT ON Student FROM user1;
```

Example TCL Queries:

```
    BEGIN;
    UPDATE Account SET balance = balance - 100 WHERE id = 1;
    COMMIT;
```

Example Primary Key and Foreign Key:

```
CREATE TABLE Department (
    deptID INT PRIMARY KEY,
    deptName VARCHAR(100)
);

CREATE TABLE Employee (
    empID INT PRIMARY KEY,
    empName VARCHAR(100),
    deptID INT,
    FOREIGN KEY (deptID) REFERENCES Department(deptID)
);
```

Example Composite Primary Key:

```
    CREATE TABLE Student (
    stdID INT PRIMARY KEY,
    stdName VARCHAR(100)
);

CREATE TABLE Course (
    crsID INT PRIMARY KEY,
    crsName VARCHAR(100)
);

CREATE TABLE Enrollment (
    stdID INT,
    crsID INT,
    grade CHAR(2),
    PRIMARY KEY (stdID, crsID),
    FOREIGN KEY (stdID) REFERENCES Student(stdID),
    FOREIGN KEY (crsID) REFERENCES Course(crsID)
);
```

Example Domain Constraints:

```
CREATE TABLE Product (
    id INT PRIMARY KEY,
    price DECIMAL(10,2) CHECK (price >= 0),
    category VARCHAR(50) NOT NULL
);
```

Set Operators

Arity / Degree: let $R(A_1, \cdots, A_n)$ be a relation schema, the arity of R is $\text{arity}(R) = n$

Union Compatible: We say that relations $R(A_1, \cdots, A_n)$ and $S(B_1, \cdots, B_n)$ are union compatible if $\text{artiy}(R) = \text{arity}(S)$ and $\text{dom}(A_i) = \text{dom}(B_i) \quad \forall i \in \{1, \cdots, n\}$

Relation Union: $R_1 \cup R_2 = \{t \mid t \in R_1 \vee t \in R_2\}$. SQL Equiv.: `SELECT * FROM R UNION SELECT * FROM S`

Relation Intersection: $R_1 \cap R_2 = \{t \mid t \in R_1 \wedge t \in R_2\}$. SQL: `SELECT * FROM R INTERSECT SELECT * FROM S`

Relation Difference: $R_1 - R_2 = \{t \mid t \in R_1 \wedge t \notin R_2\}$. SQL: `SELECT * FROM R EXCEPT SELECT * FROM S`

Relation Cartesian Product: $R_1 \times R_2 = \{t_1 \circ t_2 \mid t_1 \in R_1,\ t_2 \in R_2\}$. SQL: `SELECT * FROM R CROSS JOIN S`

Relation Operators (Unary)

Rename: Changes the schema of the relation R by renaming attribute A_1 to B_1 and so on. $\rho_{(B1,\cdots,B_n \leftarrow A_1,\cdots,A_n)}(R)$. SQL: `SELECT a AS b FROM R AS R1`

Selection: $\sigma_C(R) = \{t \in R \mid C(t)\}$ is the set of all tuples in R that satisfy the condition C.
The condition C is a Boolean expression composed of predicates:

$$C = P_1\ op_1\ P_2\ op_2\ \cdots\ op_{n-1}\ P_n$$

where each operator $op_i \in \{\text{AND},\ \text{OR},\ \text{NOT}\}$, and each predicate P_i has the form:

$$P_i ::= A\ \theta\ B \quad \text{or} \quad A\ \theta\ c$$

with attributes A, B, constant c, and comparison operator $\theta \in \{=,\ <,\ \leq,\ >,\ \geq,\ \neq\}$.

SQL Equiv. : `SELECT * FROM R WHERE C`

Projection: $\Pi_Y(R) = \{t[Y] \mid t \in R\}$. SQL: `SELECT Y FROM R`

Idempotent: An operator $\mathcal{O}$ is called *idempotent* if applying it multiple times has the same effect as applying it once: $\mathcal{O}(\mathcal{O}(x)) = \mathcal{O}(x) \quad \forall x$. **Projection** is *Idempotent*

Relation Operators (Binary)

Theta Join: $R_1 \bowtie_C R_2 = \{t_1 \circ t_2 \mid t_1 \in R_1 \wedge t_2 \in R_2 \wedge C(t_1 \circ t_2)\} = \sigma_C(R_1 \times R_2)$

Example: `SELECT * FROM Employee JOIN Bonus ON Employee.salary > Bonus.threshold`

id	salary
1	50000
2	30000

threshold
20000
40000

id	salary	threshold
1	50000	20000
1	50000	40000
2	30000	20000

Equi-Join: Theta Join with C consists only of equality comparison.

Example: `SELECT * FROM Orders JOIN Customers ON Orders.cust_id = Customers.id`

order_id	cust_id
101	1
102	2
103	4

id	name
1	Alice
2	Bob
3	Carol

order_id	cust_id	id	name
101	1	1	Alice
102	2	2	Bob

Natural Join: Equi-Join where C is quality on common attributes and duplicate common attributes are removed from the result.

Example: `SELECT * FROM Employee NATURAL JOIN Department`

emp_id	dept_id
1	10
2	20
3	10
4	40
5	50

dept_id	name
10	HR
20	IT
30	Sales
60	Legal

emp_id	dept_id	name
1	10	HR
2	20	IT
3	10	HR

Outer Join: Natural Join but preserves unmatched tuples by padding them with NULL values. That can be done on the relation on the left, right or both.

Left Outer Join: Includes all tuples from the left relation, padding unmatched right-side tuples with NULLS.

Example: `SELECT * FROM Orders LEFT OUTER JOIN Customers ON Orders.cust_id = Customers.id`

order_id	cust_id
101	1
102	2
103	4

id	name
1	Alice
2	Bob
3	Carol

order_id	cust_id	id	name
101	1	1	Alice
102	2	2	Bob
103	4	NULL	NULL

Right Outer Join: Includes all tuples from the right relation, padding unmatched left-side tuples with NULLS.

Example: `SELECT * FROM Orders RIGHT OUTER JOIN Customers ON Orders.cust_id = Customers.id`

order_id	cust_id
101	1
102	2
103	4

id	name
1	Alice
2	Bob
3	Carol

order_id	cust_id	id	name
101	1	1	Alice
102	2	2	Bob
NULL	NULL	3	Carol

Full Outer Join: Includes all tuples from both relations, padding unmatched tuples from either side with NULLS.
Example: `SELECT * FROM Orders FULL OUTER JOIN Customers ON Orders.cust_id = Customers.id`

order_id	cust_id
101	1
102	2
103	4

id	name
1	Alice
2	Bob
3	Carol

order_id	cust_id	id	name
101	1	1	Alice
102	2	2	Bob
103	4	NULL	NULL
NULL	NULL	3	Carol

Operators Properties

Operator	**Result Schema**	**Result Size**	**Comm.**	**Assoc.**	**Idem.**	**Duplicates**
Union	Same as inputs	$\leq \lvert R\rvert + \lvert S\rvert$	Yes	Yes	No	No
Intersection	Same as inputs	$\leq \min(\lvert R\rvert, \lvert S\rvert)$	Yes	Yes	Yes	No
Difference	Same as R	$\leq \lvert R\rvert$	No	No	No	No
Cartesian Product	$R \cup S$	$\lvert R\rvert \cdot \lvert S\rvert$	Yes	Yes	No	Yes
Rename	Same	$\lvert R\rvert$	N/A	N/A	N/A	N/A
Selection	Same as R	$\leq \lvert R\rvert$	Yes	Yes	No	Yes
Projection	Subset of R	$\leq \lvert R\rvert$	No	No	Yes	No
Theta Join	$R \cup S$	$[0,\ \lvert R\rvert \cdot \lvert S\rvert]$	No	No	No	Yes
Equi-Join	$R \cup S$	$[0,\ \lvert R\rvert \cdot \lvert S\rvert]$	No	No	No	Yes
Natural Join	$R \cup S \setminus C$	$[0,\ \lvert R\rvert \cdot \lvert S\rvert]$	Yes	Yes	No	Yes
Left Outer Join	$R \cup S$	$\geq \lvert R\rvert$	No	No	No	Yes
Right Outer Join	$R \cup S$	$\geq \lvert S\rvert$	No	No	No	Yes
Full Outer Join	$R \cup S$	$\geq \max(\lvert R\rvert, \lvert S\rvert)$	No	No	No	Yes

Common Datatypes

Category	**Common Data Types**
Numeric	INT, BIGINT, DECIMAL(p,s), FLOAT, DOUBLE
Character	CHAR(n), VARCHAR(n), TEXT
Date/Time	DATE, TIME, TIMESTAMP, DATETIME
Boolean	BOOLEAN / BOOL

Algebra to SQL Example

$$\pi_{\text{supplier\_name},\ \text{product\_name}}\left(\sigma_{\text{price}>50}\left((\text{Suppliers} \bowtie_{\text{supplier\_id}} \text{Products}) \bowtie_{\text{category\_id}} \text{Categories}\right)\right)$$

```
WITH SupplierProducts AS (
    SELECT
        s.supplier_name,
        p.product_name,
        p.price,
        p.category_id
    FROM Suppliers s
    JOIN Products p ON s.supplier_id = p.supplier_id
),
Filtered AS (
    SELECT * FROM SupplierProducts WHERE price > 50
)
SELECT f.supplier_name, f.product_name
FROM Filtered f
JOIN Categories c ON f.category_id = c.category_id;
```