CS 4400 - Databases I

Jerry Chen

Contents

1 Intro to Databases

1.1 Databases

1.1.1 Definition and Properties

A **database** represents one aspect of the real world, is logically coherent, is is built with a specific purpose for a specific group of users. It is essentially a collection of related data. - An entire **database system** consists of both the data store and the management system. - Database languages have two types: **Database Definition Language** (creation of schemas) and the **Database Manipulation Language** (manipulation of data schema). MySQL encompases both.

A **relational database** has the following strengths/attributes: - Self-describing: names of entities make sense logically. - Program-data independence: the data can change without the program needing to change (as long as logic stays same). - Multiple views: different users (e.g., admin vs. client) see different views of the same data. - Reduces data redundancy: relations across entities can be optimized so that there is only a single source of truth for any piece of information

Database state refers to a snapshot of the contents of a database at some point in time. At all points, it must remain in a **valid state**, which is enforced according to the **database schema** defined (e.g., can a field be null). - Conceptual schema: global “official” design of the database (which tables/entities exist, what their attributes are, etc.). - External schema: specific view exposed to applications and their end users.

2 Concepts of Database Architecture

2.1 Database Concepts

2.1.1 Models, Schemas, and Instances

A **conceptual model** is created by the database designer to describe the data at a high level, independent of any specific database software. It is often represented with an **entity–relationship (ER) diagram**, which identifies entities, their attributes, and relationships between entities, including constraints like cardinalities (how many instances of one entity can be associated with one instance of another entity in a relationship).

A **schema** is the concrete structural definition of a database. It specifies tables, columns, column data types, and constraints that define which database states are valid (for example, primary keys, foreign keys, uniqueness constraints, and nullability rules). An **instance** is the actual data stored in the database at a particular point in time; the schema describes the structure, while the instance is the current contents.

2.1.2 3-Schema Architecture

The **3-schema architecture** organizes database descriptions into three abstraction layers: 1. **Internal schema**: Describes physical storage details, such as file organization and access paths. This includes decisions like whether indexing uses a B-tree or hashing and how records are laid out on disk. 2. **Conceptual schema**: Describes the logical structure of the database, including entities/tables, attributes/columns, relationships, and constraints, without focusing on physical storage. 3. **External schema**: Describes user-facing views tailored to applications or end users. External schemas are often subsets of the conceptual schema and can present the same underlying data in different formats for different consumers.

These layers function as abstraction boundaries: external views should not depend on physical storage details, and physical storage can change without rewriting the logical design.

2.1.3 Data Independence

Data independence is the ability to change one level of the database architecture without forcing changes at higher levels.

Logical data independence is the ability to modify the conceptual schema without requiring changes to external schemas or the applications that depend on them. This is difficult in practice, especially when changes remove or fundamentally alter fields that an external schema expects (for example, deletion or restructuring). The intended effect is that external schemas and applications that do not use the changed parts of the conceptual schema should not need to change.

Physical data independence is the ability to change the internal schema (physical storage, indexing strategy, file layout) without affecting the conceptual schema. This is generally feasible because the conceptual schema describes what data means and how it is related, not how it is stored.

2.2 Entities, Attributes, and Relationships

An **entity** represents a distinguishable real-world object or concept (for example, Student, Employee, Course). Entities are described using attributes, and attributes come in several common types:

Simple (atomic) vs. composite attributes - A **simple (atomic) attribute** cannot be meaningfully subdivided (for example, Age). - A **composite attribute** can be broken into smaller components (for example, Name $\rightarrow$ FirstName + LastName). - Example: First name + last name versus full name. - Note: Composite attributes are not stored as a single merged column in the entity table. Each component becomes its own column, and the combined/composite version is typically not stored as a column at all; it remains conceptual.

Single-valued vs. multi-valued attributes - A **single-valued attribute** has one value per entity instance (for example, Age). - A **multi-valued attribute** can have multiple values for one entity instance (for example, a person may

have multiple college degrees earned). In an ER model this is conceptualized as a set/list; in a relational schema, this usually requires a separate table rather than a single column holding multiple values.

Stored vs. derived attributes - A **stored attribute** is explicitly stored in the database (for example, Birthdate). - A **derived attribute** is computed from other stored attributes (for example, Age derived from Birthdate and the current date). - Example: Birthdate can be stored, while age can be derived from the stored birthdate. - Often it is preferable to avoid storing derived attributes because they create update maintenance overhead (the derived value can become stale and must be kept consistent).

2.2.1 Keys, Strong Entities, and Weak Entities

A **key** is an attribute (or set of attributes) whose values uniquely identify each entity instance within an entity set. Entities commonly have one or more candidate keys, and one of them is chosen as the primary key in the relational schema.

A **strong entity** has at least one key attribute (or key attribute set) that uniquely identifies each instance on its own.

A **weak entity** does not have sufficient attributes to uniquely identify its instances independently. A weak entity must be identified through an association with an "owning" (identifying) strong entity. - Example: Dependents of a company employee may not have an employee ID or any attribute that uniquely identifies them across all dependents. A dependent is identified only when linked to the specific employee (the owner). The dependent's own weak key attribute (like dependent name) is not enough by itself; it becomes identifying only when combined with the owner's key.

2.2.2 Participation and Cardinalities

A **relationship** describes an association between entity types (for example, EMPLOYEE works in DEPARTMENT, STUDENT enrolls in COURSE).

Relationship constraints determine how entities can participate.

A **cardinality ratio** specifies the maximum number of relationship instances an entity can participate in with regards to a separate entity (one entity may have multiple different relationships, but this defines the *mapping* to another entity. - Common maximum-cardinality patterns: - **1:1** (one-to-one): Each entity instance relates to at most one instance of the other entity. - Ex: One person owns one phone, and a phone has one owner. - **1:N** (one-to-many): One instance on the "1" side can relate to many instances on the "N" side, while each "N" side instance relates to at most one "1" side instance. - Ex: One department has many employees, but an employee is only in one department at a time. - **M:N** (many-to-many): Instances on both sides can relate to multiple instances on the other side. - Ex: One employee can work on multiple projects, and one project can have multiple employees working on it.

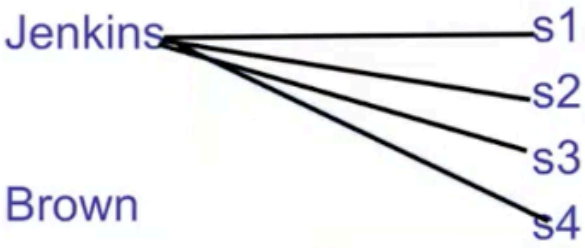
Participation (minimum cardinality / existence constraint)
Participation indicates whether an entity must participate in a relationship. - **Total participation**: Every entity instance must participate in at least one relationship instance. - **Partial participation**: Some entity instances may exist without participating in the relationship. - Example questions that reflect participation constraints: - Does a student need to have a degree? - Does a degree have to have at least one student?

2.2.3 Min/Max Result Set Sizes

The min/max result set size is the smallest possible and largest possible number of tuples (rows) returned from a database query.

This is done with a combination of cardinalities, participation constraints (partial/total), and knowledge of table size (number of rows in specific entity table).

Calculating the Minimum Result Set size



What if the student participation were total? Could the minimum be zero? Now the minimum is 4.

Pay particular attention to the total participation scenarios.

1) Maximum

- **1:1**: max = `min(|A|, |B|)`
- **1:N** (A is "1", B is "N"): max = |B| (one per B, since each B can match at most one A)
- **M:N**: max = |A| * |B|

2) Minimum
Minimum depends on **participation** (single vs double line), not on 1/N alone.

- If **both sides partial**: min = 0
- If **A total, B partial**:
  - **1:1**: min = $|A|$ (each A must match exactly one B, but possible only if $|A| \leq |B|$)
  - **1:N** (A is 1-side): min = $|A|$ (each A must be related to ≥1 B; can reuse multiple B per A)
  - **1:N** (A is N-side): min = $|A|$ (each A (on N-side) needs one B on 1-side)
  - **M:N**: min = $|A|$
- If **both sides total**: min = $\max(|A|, |B|)$ for **M:N**, but for **1:1** it requires $|A| = |B|$ or else impossible.

2.2.4 Relationship Type Attributes

A **relationship-type attribute** is an attribute that belongs to the association between entities rather than to either entity alone. These attributes describe the state/details of the relationship instance. Example: STUDENT and COURSE connected by an ENROLLS relationship. - Relationship-type attributes: Grade, enroll date, section number. - These attributes do not inherently belong exclusively to STUDENT or to COURSE; they describe the enrollment relationship itself.

2.2.5 Entity-Relationship Diagramming Rules and Notation

ER diagrams use standardized shapes and line styles to encode meaning:

Core shapes - **Square (rectangle)**: Entity - **Double square (double rectangle)**: Weak entity - **Single diamond** with text: Relationship - **Double diamond** with text: Identifying relationship (links a weak entity to its identifying/owning entity)

Attributes - **Oval**: Attribute - **Underlined attribute**: Key attribute - **Dotted underline**: Partial key (weak/partial key attribute) used with weak entities. NOTE: does not count as a key attribute. - Example: A dependent's name might be a partial key; it uniquely identifies a dependent only when combined with the owning employee's primary key. - **Dotted oval**: Derived attribute - Note: A derived attribute is not stored as its own column; it is computed from one or more stored columns. - **Double oval**: Multi-valued attribute - **Composite attribute**: A circle with multiple smaller circles connected (components of the composite)

Participation lines - **Single line**: Partial participation - **Double line**: Total participation

2.2.6 Naming and Reading Conventions

Common stylistic conventions for ER diagrams include: - Use **singular** names for entity types. - Entity and relationship type names are often written in **UPPERCASE**. - Attribute names often use an initial capital letter. - Role names are written in lowercase letters. - Diagrams are typically laid out to be read left-to-right or top-to-bottom to keep relationships interpretable as complexity grows.

3 Extended Entity Relationship Diagrams

3.1 The EERD Model

3.1.1 Inheritance, Supertypes, and Subtypes

The **Enhanced Entity-Relationship Diagram (EERD)** model extends the ER model by adding **inheritance** through **supertypes** and **subtypes** (also called subclasses). An entity type can be a **supertype** (top-level, more general), a **subtype** (more specific), or both (a subtype that is also a supertype of another subtype).

Any entity that is a member of a **subtype** (meaning it inherits from a supertype) inherits **all attributes** and **all relationships** of the supertype. A subtype may also introduce **additional attributes** and **additional relationships** that do not apply to the entire supertype.

3.1.2 Notation for Inheritance Direction

The **"U" symbol** indicates the **direction of inheritance**. It visually communicates that membership flows "into" the subtype, similar to how an object can be considered "in" a set (i.e., the subtype represents a subset of the supertype's instances).

3.1.3 Constraints on Identity of Subclass Entities

EERD specialization can impose constraints on whether membership across subtypes is mutually exclusive or allowed to overlap.

- **Disjoint constraint** (shown as **"d"** in a circle): a supertype instance can belong to **at most one** of the subtypes in that specialization.
  - Example: A **Person** is either a **Student** or an **Employee** (cannot be in school full time and also working full time).
  - Note: Disjointness is about subtype membership, not real-world feasibility; the "full time" rationale is an interpretation, but the modeling rule is simply "not both."
- **Overlapping constraint** (shown as **"o"** in a circle): a supertype instance may belong to **multiple** subtypes in that specialization.
  - Example: A **Person** can be both an **Engineer** and a **Parent**.

3.1.4 Participation Constraints in Subclass Entities

A specialization can also specify whether every supertype instance must belong to at least one subtype.

- **Total participation**: **every** supertype instance must be in **some** subtype within the specialization.
  - Example: A **Student** must be some subtype like **Undergraduate** or **Graduate**.
- **Partial participation**: some supertype instances may belong to **no** subtype in the specialization.
  - Example: A **MathStudent** does not need to be a **DiscreteMathStudent** (having that concentration is optional).

3.1.5 Specialization and Generalization in Design

Specialization and generalization describe opposite directions for designing subtype/supertype hierarchies.

- **Specialization** is **top-down** design: start with a general entity type and refine it into more specific subtypes.
  - Example: Start with **Person**, then specialize into **Student** and **Employee**.
- **Generalization** is **bottom-up** design: start with multiple specific entity types and combine shared features into a common supertype.
  - Example: **Car** and **Truck** share common properties, so generalize them into **Vehicle**.

3.1.6 Union (Category)

A **union (category)** creates a single entity type whose members come from the **union of multiple entity types**. This models a group/category that includes objects of different types under one entity type.

3.1.7 Result Set Size

4 MySQL

4.1 Basic SQL Operations

4.1.1 Syntax Conventions

SQL Keywords: ALL CAPS Table names: lowercase nouns, plural Column names: lowercase noun, singular

Keep one clause per line and indent joins and nested conditions.

Comments use either –– prefix or # prefix.

Use backticks to enclose column name aliases, particularly when there may be spaces or collisions with a MySQL keyword.

4.1.2 Reading (Querying) Data

`SELECT` is the universal command for retrieving any sort of data, regardless of whether it's columns, rows, or a scalar value.

```
SELECT *
FROM employee
```

The general structure of a query:

```
SELECT <col_list>
FROM <table_list>
WHERE <condition>
GROUP BY <col_list>
HAVING <condition>
ORDER BY <col_list>
LIMIT <value> OFFSET <value> -- OFFSET must always follow LIMIT
```

Unique Outputs

```
SELECT DISTINCT fname
FROM <table>
```

Fulfilling Conditions

```
SELECT name
FROM Students
WHERE NOT LEN(name) < 5
    AND grade > 90;
```

Condition operators include:

```
LIKE
BETWEEN
IN
=, !=, <>
>, >=, <, <=
```

Ordering

```
SELECT Name
FROM STUDENTS
WHERE Marks > 75
ORDER BY RIGHT(NAME, 3) ASC, ID ASC;
```

Each `ORDER BY` column can independently be `ASC` (default) or `DESC`.

Query Result Set `LIMIT <nrows> OFFSET <n>` restricts the number of rows returned and optionally skips the first `<n>` rows.

Aliasing

```
SELECT address AS 'Address', salary * 1.05 AS 'New Salary'
FROM employee;
```

Aliases rename columns in the output. They can be referenced in `ORDER BY` but not in `WHERE` (because `WHERE` is evaluated before `SELECT`).

4.1.3 Arithmetic and Aggregate Functions

Arithmetic operators: `+`, `–`, `*`, `/` - Division always returns a non-integer type (`DECIMAL`/`DOUBLE`).

Aggregation functions reduce a set of values to a single scalar: - `COUNT()`, `SUM()`, `MAX()`, `MIN()`, `AVG()`

Other numeric functions: - `ROUND(value, decimals)`, `LENGTH(string)`

4.1.4 GROUP BY and HAVING

`GROUP BY` collapses rows into groups based on one or more columns. Every column in the `SELECT` list must either appear in the `GROUP BY` clause or be wrapped in an aggregate function. This is because each group produces exactly one output row, so any non-grouped column would have multiple candidate values with no way to pick one.

Invalid (salary is not grouped or aggregated):

```
SELECT dno, sex, salary
FROM employees
GROUP BY dno, sex
```

Valid (salary is aggregated):

```
SELECT dno, sex, MAX(salary)
FROM employees
GROUP BY dno, sex
```

Another example:

```
SELECT Dno, AVG(salary)
FROM employee
GROUP BY Dno;
```

HAVING vs WHERE `WHERE` filters individual rows *before* grouping. `HAVING` filters groups *after* aggregation. The following two queries produce identical results:

```
SELECT dno AS `Department`, COUNT(*) AS `Count`
FROM employee
WHERE dno != 1
GROUP BY dno
ORDER BY `Count` DESC;
```

```
SELECT dno AS `Department`, COUNT(*) AS `Count`
FROM employee
GROUP BY dno
HAVING dno != 1
ORDER BY `Count` DESC;
```

HAVING becomes *required* when the filter condition depends on an aggregate. Since WHERE runs before grouping, aggregates like COUNT(*) don't exist yet at that stage.

```
SELECT dno, COUNT(*) AS cnt
FROM employee
GROUP BY dno
HAVING COUNT(*) >= 5;
```

Combining both: filter rows first with WHERE, group, then filter groups with HAVING:

```
SELECT dno, COUNT(*) AS n
FROM employee
WHERE sex = 'F'
GROUP BY dno
HAVING COUNT(*) >= 3;
```

An equivalent way to express a HAVING filter is to wrap the grouped query as a subquery and filter with WHERE in the outer query:

```
SELECT *
FROM (
  SELECT dno, AVG(salary) AS avg_salary
  FROM employee
  GROUP BY dno
) g
WHERE g.avg_salary > 80000;
```

4.2 String and Regex Operations

4.2.1 String Functions

Attribute information: - LEN(string)

Capitalization transformations: - LOWER(<col>) - UPPER(<col>)

Extract left/right substring of certain length: - LEFT(string, nchars) - RIGHT(string, nchars) - If nchars > LEN(string), the whole string is returned. If nchars <= 0, an empty string is returned.

Concatenation: - CONCAT(s1, s2...)

4.2.2 LIKE

LIKE performs simple pattern matching in WHERE clauses. - % matches zero or more of any character. - _ matches exactly one character.

4.2.3 RLIKE

RLIKE (or equivalently REGEXP) enables full regex matching in WHERE clauses.

- ^ anchors start of string
- $ anchors end of string
- . matches any single character except newline (equivalent to _ in LIKE)
- * quantifier for zero or more of the preceding element
- + quantifier for one or more of the preceding element
- ? quantifier for zero or one of the preceding element (makes it optional)

[] defines a character class, matching any single character within the brackets (case-sensitive). - ^ inside brackets denotes negation: [^aeiou] matches any non-vowel.

| acts like a logical OR between patterns.

4.3 Nested Queries, Set Comparisons, and Explicit Sets

4.3.1 Nested Queries

A **nested query** (subquery) is a `SELECT` statement embedded within another SQL statement. It is used when some intermediate result needs to be computed in a separate context before the main query can proceed.

For example, to find employees earning above the company average, we first need to compute that average:

```
SELECT *
FROM employee
WHERE salary > (SELECT AVG(salary) FROM employee);
```

Subqueries also work with `IN` to check membership against a derived set. To list employees whose department has more than 3 employees:

```
SELECT *
FROM employee
WHERE dno IN (
    SELECT dno
    FROM employee
    GROUP BY dno
    HAVING COUNT(*) > 3
);
```

4.3.2 ANY and ALL

`ANY` and `ALL` compare a single value against an entire set returned by a subquery.

```
expr op ANY (subquery)
expr op ALL (subquery)
```

`ANY` returns true if the comparison holds for *at least one* value in the set. `ALL` returns true only if the comparison holds for *every* value in the set.

For example, to find employees earning more than at least one employee in department 1:

```
SELECT *
FROM employee
WHERE salary > ANY(
    SELECT salary
    FROM employee
    WHERE dno = 1
);
```

This is equivalent to comparing against the `MIN()`:

```
SELECT *
FROM employee
WHERE salary > (
    SELECT MIN(salary)
    FROM employee
    WHERE dno = 1
);
```

Similarly, `> ALL(subquery)` is equivalent to `> (SELECT MAX(...) ...)`.

4.3.3 Explicit Sets

Explicit sets let you list values directly in the query instead of deriving them from a subquery.

```
SELECT fname, lname, dno
FROM employee
WHERE dno IN (1, 4, 5);
```

4.4 Table JOINs

4.4.1 Motivation

Normalizing data into separate tables avoids redundancy and makes relationships/entities more modular. However, when querying, we often need to reconstruct data across multiple tables.

Referencing multiple tables without a join condition produces a **cartesian product**: if Table A has n rows and Table B has m rows, the result has $n \times m$ pairings.

```
SELECT col1, col2
FROM table1, table2
-- also called CROSS JOIN
```

JOINs constrain this product by specifying a matching condition, returning only the meaningful row combinations.

4.4.2 Inner JOINs

Consider two tables:

```
Student(StudentID, Name, Major)
Enrollment(StudentID[fk], CourseID, Grade)
fk: StudentID -> Student(StudentID)
```

An **inner join** returns only rows where a match exists in both tables:

```
SELECT Student.StudentID, Enrollment.CourseID, Enrollment.Grade
FROM Student
JOIN Enrollment ON Student.StudentID = Enrollment.StudentID;
```

If a student exists in `Student` but has no rows in `Enrollment` (not enrolled in any courses), they will not appear in the result.

4.4.3 Left JOINs

A **left join** guarantees that all rows in the left table (the `FROM` table) will appear. For rows in the left table with no matching row in the right table, the right table's columns appear as `NULL`.

Student

StudentID	Name
1	Alice
2	Bob
3	Carol

Enrollment

StudentID	Course
1	Math
1	History
2	Math

LEFT JOIN

```sql
SELECT s.Name, e.Course
FROM Student s
LEFT JOIN Enrollment e ON s.StudentID = e.StudentID;
```

Name	Course
Alice	Math
Alice	History
Bob	Math
Carol	NULL

4.4.4 Right JOINs

A **right join** guarantees all rows in the right table will appear. If no corresponding row is present in the left table, the left table's columns appear as `NULL`.

RIGHT JOIN

Now flip the table order and use a RIGHT JOIN:

```sql
SELECT s.Name, e.Course
FROM Student s
RIGHT JOIN Enrollment e ON s.StudentID = e.StudentID;
```

But to make this interesting, let's add an orphan row to `Enrollment` — say an enrollment for StudentID 99 who doesn't exist in `Student`:

StudentID	Course
1	Math
1	History
2	Math
99	Physics

Now the RIGHT JOIN result is:

Name	Course
Alice	Math
Alice	History
Bob	Math
NULL	Physics

Every row from the **right** table (`Enrollment`) appears guaranteed. The Physics enrollment for StudentID 99 has no matching student, so `Name` comes back as `NULL`. Carol is gone entirely now — she's on the left side with no match, so she gets dropped.

4.4.5 Natural Join

A **natural join** is an inner join that automatically joins on all columns sharing the same name in both tables. No `ON` clause is written.

```
-- employees(emp_id, dept_id, name)
-- departments(dept_id, dept_name)

SELECT *
FROM employees NATURAL JOIN departments;

-- is the same as
```

```sql
SELECT employees.emp_id, employees.dept_id, employees.name, departments.dept_name
FROM employees
JOIN departments
  ON employees.dept_id = departments.dept_id;
```

NOTE: Natural joins are fragile. If a column is later added to one table that happens to share a name with a column in the other table, the join condition silently changes. Explicit `JOIN ... ON` is generally preferred.

4.5 Set Operations

Set operations combine the results of multiple `SELECT` statements into a single result using set logic (union, intersection, difference). They fill a gap left by JOINs: JOINs require a common key and expand *column-wise*, whereas set operations expand *row-wise* while keeping column count fixed.

4.5.1 Rules

- Each `SELECT` must return the same number of columns (same data shape).
- All corresponding columns must have compatible types.
- Output column names come from the first `SELECT`.
- `ORDER BY` goes once at the very end, after all set operations.
- `INTERSECT` binds tighter than `UNION` and `EXCEPT` (like multiplication before addition). Use parentheses to override.
- A "duplicate" means two rows match on every column in the result tuple.
- `NULL`s are treated as equal for deduplication purposes (unlike normal `WHERE` comparisons).

4.5.2 UNION

`UNION` combines rows from both queries and removes duplicates, equivalent to $A \cup B$.

Example A product has two signup paths stored in separate tables. Analytics wants a unified dataset.

```sql
CREATE TABLE web_signups (
  user_id        BIGINT PRIMARY KEY,
  created_at     TIMESTAMP NOT NULL,
  utm_campaign   VARCHAR(100),
  landing_path   VARCHAR(200)
);

CREATE TABLE referral_signups (
  user_id          BIGINT PRIMARY KEY,
  signup_ts        TIMESTAMP NOT NULL,
  referrer_user_id BIGINT NOT NULL,
  referral_code    VARCHAR(40)
);
```

```sql
SELECT
  ws.user_id                          AS user_id,
  ws.created_at                       AS signup_ts,
  'web'                               AS channel,
  ws.utm_campaign                     AS campaign,
  NULL                                AS referrer_user_id
FROM web_signups ws

UNION

SELECT
  rs.user_id                          AS user_id,
```

```
  rs.signup_ts                          AS signup_ts,
  'referral'                            AS channel,
  NULL                                  AS campaign,
  rs.referrer_user_id                   AS referrer_user_id
FROM referral_signups rs

ORDER BY signup_ts DESC;
```

4.5.3 UNION ALL

`UNION ALL` concatenates all rows from both `SELECT`s with no deduplication. It is faster than `UNION` because MySQL skips the sort/hash step needed to detect duplicates. Use `UNION ALL` when duplicates are acceptable or the two sets are known to be disjoint.

Example A company tracks login and purchase events in separate tables. To build a combined activity feed, every event from both tables should be preserved.

```
-- login_events
-- +---------+---------------------+
-- | user_id | event_ts            |
-- +---------+---------------------+
-- | 101     | 2026-03-01 09:00:00 |
-- | 102     | 2026-03-01 09:05:00 |
-- | 101     | 2026-03-01 09:10:00 |  <-- user 101 logged in twice
-- +---------+---------------------+
--
-- purchase_events
-- +---------+---------------------+
-- | user_id | event_ts            |
-- +---------+---------------------+
-- | 101     | 2026-03-01 09:10:00 |  <-- same user_id + timestamp as a login
-- | 103     | 2026-03-01 09:20:00 |
-- +---------+---------------------+

SELECT user_id, event_ts, 'login'    AS event_type FROM login_events
UNION ALL
SELECT user_id, event_ts, 'purchase' AS event_type FROM purchase_events
ORDER BY event_ts;

-- Result: all 5 rows preserved, nothing removed
-- +---------+---------------------+------------+
-- | user_id | event_ts            | event_type |
-- +---------+---------------------+------------+
-- | 101     | 2026-03-01 09:00:00 | login      |
-- | 102     | 2026-03-01 09:05:00 | login      |
-- | 101     | 2026-03-01 09:10:00 | login      |
-- | 101     | 2026-03-01 09:10:00 | purchase   |  <-- kept despite same user+ts
-- | 103     | 2026-03-01 09:20:00 | purchase   |
-- +---------+---------------------+------------+
--
-- With UNION instead, the row (101, 2026-03-01 09:10:00, 'login') and
-- (101, 2026-03-01 09:10:00, 'purchase') would both survive anyway because
-- event_type differs. But if we had dropped the event_type column, UNION
-- would collapse those two into one row while UNION ALL would keep both.
```

4.5.4 INTERSECT

`INTERSECT` returns $A \cap B$: only the rows that appear in both queries. Like `UNION`, it deduplicates by default. Added in MySQL 8.0.31.

Example A university wants to find students enrolled in both CS 1332 and CS 2110.

```
-- cs1332_roster
-- +------------+
-- | student_id |
-- +------------+
-- | 904100001  |
-- | 904100002  |
-- | 904100003  |
-- | 904100004  |
-- +------------+
--
-- cs2110_roster
-- +------------+
-- | student_id |
-- +------------+
-- | 904100002  |
-- | 904100004  |
-- | 904100005  |
-- +------------+

SELECT student_id FROM cs1332_roster
INTERSECT
SELECT student_id FROM cs2110_roster;

-- Result: only students present in BOTH rosters
-- +------------+
-- | student_id |
-- +------------+
-- | 904100002  |
-- | 904100004  |
-- +------------+
```

4.5.5 EXCEPT

`EXCEPT` computes $A - B$: all rows in the first query that do not appear in the second. It deduplicates by default. Order matters: A EXCEPT $B \neq B$ EXCEPT A. Added in MySQL 8.0.31.

Example The university wants students in CS 1332 who are not in CS 2110.

```
-- cs1332_roster                  cs2110_roster
-- +------------+                 +------------+
-- | student_id |                 | student_id |
-- +------------+                 +------------+
-- | 904100001  |                 | 904100002  |
-- | 904100002  |                 | 904100004  |
-- | 904100003  |                 | 904100005  |
-- | 904100004  |                 +------------+
-- +------------+

SELECT student_id FROM cs1332_roster
EXCEPT
SELECT student_id FROM cs2110_roster;
```

```
-- Result: 1332 students minus anyone also in 2110
-- +------------+
-- | student_id |
-- +------------+
-- | 904100001  |
-- | 904100003  |
-- +------------+
--
-- 904100002 and 904100004 are removed (they appear in the 2110 roster).
--
-- Flipping the order gives a DIFFERENT result:
-- SELECT student_id FROM cs2110_roster
-- EXCEPT
-- SELECT student_id FROM cs1332_roster;
-- => {904100005}  (2110-only students)
```

4.6 Data Manipulation Language (DML)

The four core DML statements are `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. `SELECT` is covered above.

4.6.1 INSERT

`INSERT` adds one or more new rows to a table.

Single row, all columns (order must match the table's column definition):

```
INSERT INTO employee
VALUES (1, 'John', 'Smith', 50000, 1);
```

Single row, named columns (recommended for clarity and resilience to schema changes):

```
INSERT INTO employee (fname, lname, salary, dno)
VALUES ('John', 'Smith', 50000, 1);
```

Columns not listed receive their default value (or `NULL` if no default is defined and the column is nullable).

Multiple rows in a single statement:

```
INSERT INTO employee (fname, lname, salary, dno)
VALUES
  ('John', 'Smith', 50000, 1),
  ('Jane', 'Doe', 55000, 2),
  ('Bob', 'Lee', 48000, 1);
```

Insert from a subquery (bulk insert from existing data):

```
INSERT INTO high_earners (fname, lname, salary)
SELECT fname, lname, salary
FROM employee
WHERE salary > 100000;
```

The column count and types of the `SELECT` output must match the target table's column list.

4.6.2 UPDATE

`UPDATE` modifies existing rows in a table.

```
UPDATE table_name
SET col1 = value1, col2 = value2
WHERE condition;
```

Without a `WHERE` clause, every row in the table is updated.

4.6.3 DELETE

`DELETE` removes one or more rows from a table.

```
DELETE FROM table_name
WHERE condition;
```

Without a `WHERE` clause, every row in the table is removed (but the table structure remains).

4.6.4 DROP

`DROP` removes entire database objects (not individual rows). It deletes both the data and the schema definition.

```
DROP TABLE table_name;
DROP DATABASE db_name;
DROP VIEW view_name;
DROP PROCEDURE procedure_name;
```

These throw an error if the specified object does not exist. Use `IF EXISTS` as a safe alternative:

```
DROP TABLE IF EXISTS table_name;
```

NOTE: `DELETE` removes rows from a table. `DROP` removes the table (or other object) itself.

4.7 Views

A **view** is a named, saved `SELECT` query that behaves like a table. It provides a convenient abstraction: because database design stores different entities in separate tables, we frequently need to join or filter across them in the same way. A view lets us define that query once and `SELECT` from it as if it were a table.

4.7.1 Definition and Properties

All tables created with `CREATE TABLE` are **base tables**: named relations backed by a **physical table** (on-disk storage). Views are **virtual tables**. You can `SELECT` from a view just like a base table, but at runtime the database engine expands the view definition and executes the underlying query. A view does not store data on disk and does not cache computation.

A view is not a precompiled routine; it is a macro/text substitution for the SQL query string it wraps. Every time the view is queried, the full underlying query is re-executed against the current state of the base tables.

```
CREATE VIEW emp_dept (first_name, last_name, salary, dept_name) AS
SELECT e.fname, e.lname, e.salary, d.dname
FROM employee e
JOIN department d ON e.dno = d.dnumber;
```

Now instead of rewriting the join each time:

```
SELECT *
FROM emp_dept
WHERE salary > 50000;
```

4.7.2 Modifying and Dropping Views

`CREATE OR REPLACE VIEW` creates a new view or overwrites the definition of an existing one:

```
CREATE OR REPLACE VIEW emp_dept AS
SELECT e.fname, e.lname, e.salary, e.ssn, d.dname
FROM employee e
JOIN department d ON e.dno = d.dnumber;
```

```
DROP VIEW emp_dept;
DROP VIEW IF EXISTS emp_dept;
```

4.8 Database Program Modules

4.8.1 Stored Procedures

4.8.1.1 Definition and Motivation A **stored procedure** is a named, precompiled set of SQL statements and procedural logic stored within the database, invoked explicitly with `CALL`. - **Precompilation:** When a raw SQL query is sent to the database engine, it must first be parsed and optimized before execution. For procedures (and functions), these parsing and optimization steps happen once, at creation time. Subsequent calls skip that overhead. - **Procedural logic:** Normal SQL is declarative (you describe the desired result). Procedural logic adds explicit control flow: `IF/ELSEIF/ELSE`, `WHILE`, `LOOP`, variable assignments, and sequential steps.

Procedures are *imperative*: you use them to execute a series of actions (inserts, updates, deletes, conditional logic), not to return a result set. They are a way to enforce application-level logic within the database server itself, similar to how a driver executes SQL from another language (Java, Python), but running entirely server-side.

4.8.1.2 Syntax

```
DELIMITER //

CREATE PROCEDURE procedure_name (
    IN param1 dtype1,    -- IN: caller provides, procedure reads (default if unspecified)
    IN param2 dtype2,
    OUT param3 dtype3,   -- OUT: procedure sets, caller reads after
    INOUT param4 dtype4  -- INOUT: caller passes in, procedure modifies, caller sees new value
)
BEGIN
    -- DECLARE statements must appear first (before any executable statements)
    DECLARE local_var1 dtype1;
    DECLARE local_var2 dtype2;

    -- Procedure body
    SELECT COUNT(*) INTO local_var1 FROM table1;
    -- If the SELECT returns 0 rows, the variable is set to NULL.
    -- If more than 1 row is returned, an error is thrown.
END //

DELIMITER ;
```

- `DELIMITER //` temporarily changes the statement delimiter so MySQL doesn't terminate the `CREATE PROCEDURE` at the first `;` inside the body.
- If a parameter name shadows a table column name, the parameter wins unless the column is disambiguated with a table name or alias prefix.

To invoke a procedure:

```
CALL procedure_name(arg1, arg2, @out_var, @inout_var);
-- After the call, read output values:
```

```
SELECT @out_var, @inout_var;
```

4.8.2 Functions

4.8.2.1 Definition and Motivation A **stored function** (also called a user-defined function, UDF) is another type of database program module stored and executed by the DBMS. While procedures are imperative (they take *actions*), functions are computational (they *return a value*). A function always returns exactly one scalar value and can be called anywhere an expression is valid: inside `SELECT`, `WHERE`, `SET`, or even within another function or procedure.

4.8.2.2 Syntax

```
DELIMITER //

CREATE FUNCTION function_name (
    param1 dtype1,
    param2 dtype2
)
RETURNS return_dtype
DETERMINISTIC -- or NOT DETERMINISTIC; required declaration
BEGIN
    DECLARE result return_dtype;

    -- Function body (compute and assign to result)
    SELECT some_col INTO result
    FROM some_table
    WHERE condition
    LIMIT 1;

    RETURN result;
END //

DELIMITER ;
```

Key differences from procedures: - Parameters are input-only (no `IN`/`OUT`/`INOUT` keywords; all parameters are implicitly `IN`). - Must include a `RETURNS` clause declaring the return type. - Must end with a `RETURN` statement. - Must declare `DETERMINISTIC` (same inputs always produce same output) or `NOT DETERMINISTIC`. This lets the optimizer know whether it can cache results. - Cannot use `SELECT` statements that return result sets to the caller; all `SELECT` usage must be `SELECT ... INTO` a variable.

4.8.2.3 Usage Functions are called inline within SQL expressions, not with `CALL`:

```
SELECT fname, lname, function_name(salary, dno) AS bonus
FROM employee;
```

```
SELECT *
FROM employee
WHERE function_name(salary, dno) > 5000;
```

4.8.3 Local and Session Variables

Local variables exist within stored procedures/functions and are created with `DECLARE`. They are scoped to the `BEGIN ... END` block they appear in and are strongly typed (the type must be stated at declaration).

```
BEGIN
    DECLARE name VARCHAR(255);
END
```

User-defined (session) variables are session-scoped: they persist for the lifetime of the database connection, similar to global variables in code. They are dynamically typed and prefixed with `@`.

Assignment via `SET`:

```
SET @name = "Jerry";
```

`SET` is the only context where = is treated as assignment; everywhere else = means comparison.

Assignment via `SELECT ... INTO`:

```
SELECT gpa, major INTO @gpa, @major
FROM students
WHERE gtid = 12345;
```

If this returns more than 1 row, an error is thrown.

Session variables are useful for passing scalar values between queries without recomputing:

```
SELECT gpa INTO @gpa FROM students WHERE gtid = 904102029;

SELECT name, gpa FROM students
WHERE gpa > @gpa;
```

4.8.4 Control Flow

4.8.4.1 Conditionals

`CASE` statements work within queries to return a value based on conditions.

The first type of case is purely evaluative with a relative comparison to other values. It is similar to an if-else conditional but inside of queries.

```
CASE
    WHEN condition1 THEN value1
    WHEN condition2 THEN value2
    ELSE value3
END
```

```
SELECT
    name,
    score,
    CASE
        WHEN score >= 90 THEN 'A'
        WHEN score >= 80 THEN 'B'
        WHEN score >= 70 THEN 'C'
        ELSE 'F'
    END AS grade
FROM students
```

The second type of case is more similar to a switch-case statement in programming.

```
CASE expr
    WHEN expr1 THEN value1
    WHEN expr2 THEN value2
    ELSE value3
END
```

A certain type of `IF` statements are also available in query syntax:

```
SELECT
    name,
    IF(score >= 60, "pass", "fail") AS result
FROM students;
```

Conditional statements which use `IF`, `ELSEIF`, and `ELSE` work only within stored procedures/functions.

```
IF condition1 THEN
    statement1
ELSE IF condition2 THEN
    statement2
ELSE
    statement3
END IF
```

4.8.4.2 Iteration

`LOOP` A `LOOP` runs until it's explicitly stopped by a `LEAVE` statement.

```
label: LOOP
    statement;

    IF condition THEN
        LEAVE label;
    END IF;
END LOOP;
```

`WHILE` A `WHILE` loop runs checking a condition before each iteration.

```
label: WHILE condition DO
    statement;
END WHILE;
```

`REPEAT` A `REPEAT` loop first runs the body, then checks a condition to determine whether to repeat or not.

```
label: REPEAT
    statements;
UNTIL condition -- loop ends when UNTIL condition becomes true
END REPEAT;
```

`LEAVE` In SQL, a label can be provided to any code block (e.g., body of a procedure, or a loop). The `LEAVE` keyword paired with a particular label exits that block of code. - If there is nesting, the level of the particular label matters which block is exited. - There must be a `label_name` provided. Otherwise it is a syntax error.

```
label_name: BEGIN
    ...
    IF condition THEN
        LEAVE label_name;
    END IF;
    ...
END
```

`ITERATE` `ITERATE` is the equivalent of the `continue` statement in Python/Java. It's used to skip one iteration (skip any remaining code in current iteration of the loop and move onto the next iteration).

Just like `LEAVE`, `ITERATE` also needs a label to reference. This means that it can technically skip an iteration not just in the innermost loop, but at any level of enclosing loop.

```
DELIMITER //
CREATE PROCEDURE iterate_ex ()
BEGIN
    DECLARE x INT DEFAULT 0;

    while_label: WHILE x < 5 THEN
        SET x = x + 1;

        IF x = 3 THEN
            ITERATE while_label; -- since this takes a label, you can technically skip an
    iteration of any level of enclosing loop
        END IF;

        SELECT x;
    END WHILE;

END //
DELIMITER ;
```

4.9 Misc Built-In Functions

`FIND_IN_SET(target_val, comma_separated_vals)` - Returns 1-indexed position of target value in a string of comma-separated values.

```
SELECT *
FROM products p
WHERE FIND_IN_SET('red', p.tags) > 0;
```

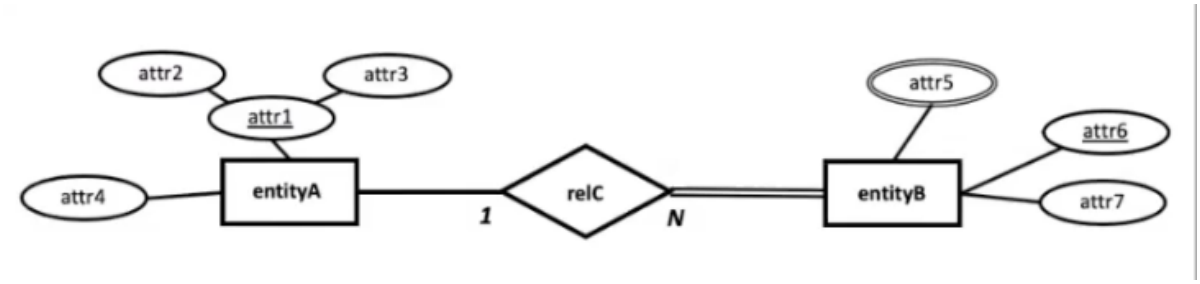
- `tags` is a column name containing values which are essentially arrays (strings of comma-separated values).
- This is essentially equivalent to “if ‘red’ in tags”.

5 Schema Mapping and Table Creation

5.1 Schema Mapping Procedure

5.1.1 Text Schema Notation

Entities, column attributes in parentheses as list. Underline primary key, or multiple if composite primary key. Denote different foreign keys and specify references at bottom.



Example

Each row describes a relation (table).

```
entityA = (_attr2_, _attr3_ attr4) # MUST write composite attribute as its attribute components
entityB = (_attr6_, attr5, attr7)
entityM = (_attr9_[fk], _attr5_) # multivalued attr
fk: attr
```

5.1.2 Entities

5.1.2.1 Step 1: Strong Entities A **strong entity** will get its own table (table name = entity name), where its attributes are columns. Its unique identifier becomes the primary key.

```
employee(_ssn_, fname, lname, age)
```

5.1.2.2 Step 2: Weak Entities A **weak entity** will also get its own table, acting similarly as a strong entity, but it must also have a foreign key (FK) which is the PK of the strong entity it is tied to.

```
dependent(_essn_ [fk], _name_, age)
fk: essn -> employee (ssn) # <weak_attr> -> <strong_entity> (<strong_attr>)
```

5.1.3 2-Entity Relationships

5.1.3.1 Step 3: 1:1 Relationships **Foreign Key Approach**: If one side has partial participation and the other side has total participation, then the entity with partial participation should store the foreign key as well as any relationship attributes in their table. - The goal here is to minimize NULLs. Since it has total participation, that entity is almost like a weak entity, where it must be linked with this other entity who may or may not participate through a relationship. If there is an entry in this table, then the foreign key must reference a valid row in the other entity's table.

Merged Relation Approach: If both sides have total participation, then the above approach also works, but it is even better to merge both entities into a single relationship table since there is a perfect 1:1 correspondance between their rows (tightly coupled together).

Cross-Reference Table (Relationship Relation): If both sides have partial participation, then no matter which side keeps the FK, there will be NULL values in that column. NULL values introduce more friction and require guards in queries, so we want to avoid them. The best choice in this scenario is to create a third table based on the relationship and include only the rows where the two entities are in the relationship.

This third table would contain the foreign keys of both entities (really, the PKs of both entities), as well as any relationship attributes. NOTE: It only needs to pick either foreign key to act as its primary key since the relationship is 1:1 (no value can repeat, so there's no need to pick out a combination for uniqueness).

5.1.3.2 Step 4: 1:N Relationships Now, we can no longer use a merged relation approach because that would result in the duplication of information stored.

Foreign Key Approach: If the N side has total participation, then we can place the foreign key + relationship attributes in that entity's table. This is because the NOT NULL condition is guaranteed.

Relationship Relation: If the N side has partial participation, then we can still create a separate table, similar to above. NOTE: We no longer get to pick which entity's foreign key to take as the primary key. We must pick the N side's identifier as the primary key.

5.1.3.3 Step 5: N:M Relationships **Relationship Relation:** Same as above, but this time we need to pick the combination (composite key) of both foreign keys to act as the PK. The combination including all foreign keys naturally acts as a superkey. However, we must reduce to a candidate key / primary key (maximally reduced superkey) by removing all of the "1" side entity foreign keys from consideration. - I.e., if the relationship is (1,N,M), then (N,M) combinations are able to represent all combinations uniquely.

5.1.4 Step 6: Multivalued Attributes

Consider an entity owning a multivalued attribute. It is similar to a 1:N relationship, but the specific value within the multivalued attribute doesn't have its own table.

The solution is again, a **relationship relation**. The PK of the owner entity is used as a foreign key, and the actual value forms a PK through a combination of (FK, value). - The combination of the two is necessary, because the owner may appear multiple times (not unique), and the same value may appear across different owners. However, an owner will only have one such value within the multivalued attribute, so the combination is unique.

5.1.5 Step 7: N-ary (Multiway) Relationships

We stick with a cross-reference table (relationship relation) with however many entities are involved as foreign keys (+ any relationship attributes). The PK is the combination of FKs from the "many" side.

5.1.6 Step 8: Superclass/Subclass Specialization

There are a few different approaches, each appropriate for difference scenarios

Multiple Relations - Superclass Table and Subclass Tables: Create one table for the superclass containing all shared attributes (including unique identifier) and create a table for the subclasses which contain their local attributes + FK which is superclass PK. - This works because all entities are guaranteed to have a unique identifier in the EERD, including the superclass. - Works for ANY overlapping/disjoint and partial/total specialization.

Multiple Relations - Subclass Tables Only: Avoid creating a table for superclass and create individual tables for each subclass containing local attributes and shared attributes. - Only works when inheritance is disjoint and there is total specialization from the superclass (double line from superclass). This way, the entity must exist in ONE state at a time, and there are no generic instances of the superclass (imagine abstract base class; no instances of general type).

Single Relation with 1 Type Attribute: Avoid any subclass tables and create only a single table for the superclass containing the shared attributes and *all local attributes across all subclasses*, AS WELL AS a "type" column distinguishing which child a row instance belongs to, if any. There will be NULLs across columns which are not applicable (e.g., if there is Device parent and Phone/Computer child, a row of type Computer would just have NULL in a local attribute of Phone (such as isAndroid)). - Works only when inheritance is disjoint, because within the type column

only one value can be stored at a time. - In practice, it can be useful when subclasses have few local attributes, which leads to few NULLs, but generally avoided due to storage wastefulness.

Single Relation with Multiple Type Attributes: Same as above, but there is one boolean flag attribute column per subclass (similar to one-hot encoding) which allows for it to be just as flexible as the first option, enabling it to handle any type of participation and specialization. If a superclass doesn't specialize, all flag columns are 0. If it does for one or more, those columns are set to 1.

5.1.7 Step 9: Union Types

When there is a union, it is kind of the reverse of superclass-subclass relationship. There are multiple superclasses and on "result" or union child. The union type is guaranteed to be exactly one of the participating superclasses. - The issue is that the resulting union type may have any one of the participating superclasses's PKs as its foreign key. But we don't know which superclass entity to look in regarding the reference (essentially 1 to many reference). The superclass PKs could even be different data types for all we know, which would make it impossible to store in a single column.

The solution is the creation of a **surrogate key** with no business meaning. There is a new relation for the union type itself, and this relation's PK is the surrogate key. Then each participating subclass gets this surrogate key as a FK (the "multiple" side gets the FK). - Depending of union type total/partial participation, the FK column may or may not contain NULLs.

5.2 DDL Syntax

5.2.1 Terminology

The following terminology maps relational model concepts to SQL constructs: a **table** corresponds to a relation, a **row** to a tuple, and a **column** to an attribute.

```
CREATE DATABASE IF NOT EXISTS <dbname>;
```

5.2.2 Core Statements

`CREATE TABLE` defines a new relation and its schema, including column definitions, constraints, and foreign key references.

```
CREATE TABLE department (
    Dname VARCHAR(15) NOT NULL UNIQUE,
    Dnumber INT PRIMARY KEY, -- automatically carries NOT NULL + UNIQUE
    Mgr_ssn INT NOT NULL,
    Mgr_start_date DATE NOT NULL,
    FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
);
```

`INSERT` adds tuples to a relation. When all columns are provided in order, the column list can be omitted. When a column list is specified, only the listed columns receive explicit values; all remaining columns must either allow `NULL` or have a `DEFAULT` value defined.

```
-- All columns, values in schema order
INSERT INTO employee
VALUES (<values>);

-- Partial column list
INSERT INTO employee (<col_list>)
VALUES (<values>);
```

DELETE removes tuples from a relation. Tuples are deleted from one table at a time unless referential integrity constraints (e.g., CASCADE) propagate the deletion to other tables. All rows satisfying the WHERE condition are deleted. The operation is **idempotent**: if no rows match the condition, nothing happens, and executing the same statement again produces no further change.

```
DELETE FROM employee
WHERE <condition>;
```

The referential constraint behavior on delete defaults to RESTRICT, but can be set to CASCADE, SET NULL, or SET DEFAULT as described in the referential integrity section below.

5.2.3 Data Types

SQL provides several built-in data types for column definitions: - VARCHAR(n): Variable-length character string with a maximum of n characters. - CHAR(n): Fixed-length character string of exactly n characters. - INT, FLOAT: Numeric types for integers and floating-point numbers. - DATE, TIME, TIMESTAMP: Temporal types for dates, times, and combined date-time values. - BOOLEAN: Logical true/false values.

Default values for a column are specified with the DEFAULT keyword in the column definition, providing a fallback value when an INSERT omits that column.

5.2.4 Constraints

SQL enforces several categories of constraints to ensure that the data in a database remains valid and consistent. These fall into the following types: domain, key, null, entity integrity, referential integrity, and check constraints.

Checking order: 1. NOT NULL and PK constraints (entity integrity, null constraints). 2. Check values (key integrity, domain constraints). 3. Referential integrity (check other table).

5.2.4.1 Domain Constraints (Values) **Domain constraints** restrict the set of allowed values for an attribute column. In other words, any valid value must fall within a domain of values. This manifests in the data type and CHECK clauses (e.g., age must be an integer, and should be non-negative).

5.2.4.2 Key Constraints (Uniqueness) A **relation** consists of a set of tuples, so all tuples in a relation must be distinct: the combination of attribute values in every row must be unique.

A **superkey** is a subset of attributes of a relational schema such that no two tuples in any valid state of the relation have the same combination of values for those attributes. Using all attributes of the relation trivially forms a superkey. A superkey may contain **redundant attributes**, meaning some attributes could be removed and the remaining set would still uniquely identify every tuple.

A **candidate key** is a superkey with no redundant attributes. Removing any single attribute from a candidate key causes it to lose the uniqueness guarantee, so it would no longer be a superkey. A **primary key** is the candidate key chosen to serve as the table's main identifier for locating and referencing tuples. It is declared with the PRIMARY KEY constraint, which automatically enforces both NOT NULL and UNIQUE.

Key constraints refer only to primary keys: it must be unique.

5.2.4.3 Entity Integrity Constraint (PK) **Entity integrity constraints** require that no primary key is NULL. In other words, each entity must have a valid identifier.

5.2.4.4 Referential Integrity Constraint (FK) **Referential integrity constraints** dictates that all non-null foreign keys must reference a valid PK in a different table (in the referenced relation). When a referenced parent row is deleted, SQL provides several policies for handling the child rows that depend on it, specified with ON DELETE: - RESTRICT (default): Reject the delete; the parent row cannot be removed until all referencing child rows are deleted

first. - `CASCADE`: Automatically delete all child rows that reference the deleted parent row. - `SET NULL`: Set the foreign key column in each referencing child row to `NULL`. - `SET DEFAULT`: Set the foreign key column in each referencing child row to that column's defined default value.

A referential integrity error can only be triggered by `ON DELETE RESTRICT`, but may also occur from downstream effects of `ON DELETE CASCADE` (i.e., deletion cascades down until it hits a restrict clause elsewhere). - Essentially, there may be a parent and a child. The children are the rows referencing the parent via a foreign key. If the child's table definition has a referential integrity constraint, and we attempt to delete the parent row before deleting all children, then there will be an error.

5.2.4.5 Null Constraints (Also Values) They are exactly what they sound like: some values cannot be NULL. This is somewhat overlapping with domain constraints but somehow has its own category.

5.2.5 Writing CHECK Constraints

6 Functional Dependencies and Normal Forms

6.1 Functional Dependencies

Up to this point in database design, the semantics of attributes have been clear in the schema. The goals of good relational design are to reduce redundant information stored in tuples, reduce the number of NULL values in columns (through better schema mapping choices), and avoid generating spurious tuples from bad decompositions.

`R1(Student, Course)`
`R2(Course, Instructor)`

Joining on `Course` is fine if each course has one instructor.

But now suppose you instead have:

`R1(Student, Department)`
`R2(Department, Advisor)`

and department is not unique in either way you need for the intended fact.

Example data:

`R1`

- (Alice, CS)
- (Bob, CS)

`R2`

- (CS, Prof. Lee)
- (CS, Prof. Kim)

Join on `Department` gives:

- (Alice, CS, Prof. Lee)
- (Alice, CS, Prof. Kim)
- (Bob, CS, Prof. Lee)
- (Bob, CS, Prof. Kim)

6.1.1 Definition and Motivation

A **functional dependency** (FD) is a constraint of the form $X \to Y$, meaning that if two tuples agree on the value of attribute set X, they must also agree on the value of attribute set Y. The set X is called the **determinant** and Y is the **dependent**.

Consider a relation with attributes `student_id`, `student_name`, `major`. The FD `student_id` $\to$ `student_name`, `major` holds because `student_id` uniquely determines the student's name and major. The reverse does not hold: knowing a student's name does not determine their `student_id`, since multiple students can share the same name.

Another way to think about it: the determinant X is more "specific" than Y. Knowing X pins down Y, but not necessarily the other way around.

6.1.2 Determining Functional Dependencies

Given a populated relation, we cannot determine which FDs hold and which do not unless we understand the meaning of and the relationships among the attributes. All we can say from looking at data is that a certain FD *may* exist if it holds in that particular extension (i.e., the current set of tuples). We cannot guarantee its existence until we understand the semantics of the corresponding attributes. We can, however, definitively state that a certain FD *does not* hold if there exist tuples that violate it: two tuples that agree on X but disagree on Y are a concrete counterexample.

6.1.3 Implications for Database Design

The goal is to group together columns that are facts about the same entity, identified by the same determinant X.

What good design looks like. If $X \to Y$, that suggests Y is a fact about X. For example:

- `student_id` $\to$ `name`, `major`, `graduation_year`
- `course_id` $\to$ `course_title`, `credits`
- `instructor_id` $\to$ `office`, `department`

These feel natural because the right-hand-side attributes are all properties of the entity identified by X.

What you do not want. You do not want to force unrelated facts into one table just because some dependency chain connects them. For example, suppose:

- `class_id` $\to$ `instructor_id`
- `instructor_id` $\to$ `office`

By transitivity, `class_id` $\to$ `office`. But `office` is not really a direct fact about the class; it is a fact about the instructor. Collapsing this into one table introduces redundancy (the same office repeated for every class that instructor teaches) and update anomalies.

6.2 Normalization

Normalization is the process of restructuring a relational schema so that data is stored in a simple, consistent form that minimizes redundancy and avoids anomalies.

6.2.1 Key Terminology

A **superkey** is any set of attributes whose values uniquely identify a tuple in the relation. A **candidate key** is a minimal superkey: no proper subset of it is also a superkey. A relation can have multiple candidate keys; one is chosen as the **primary key**.

A **prime attribute** is an attribute that belongs to *some* candidate key. A **non-prime attribute** is an attribute that does not belong to any candidate key.

6.2.2 Anomalies

Anomalies are problems that arise when a schema is poorly designed, and they may result from insertions, deletions, and updates.

- **Insertion anomaly:** you cannot record a new fact without also supplying unrelated data. For instance, if instructor and office are stored in a class table, you cannot record an instructor's office until that instructor is assigned to a class.
- **Deletion anomaly:** removing a tuple causes unintended loss of other facts. Deleting the last class taught by an instructor also destroys the record of that instructor's office.
- **Update anomaly:** a fact is stored in multiple tuples, so changing it requires updating every copy. If an instructor's office changes, every row for that instructor's classes must be updated.

Anomalies Example

classes

class_name	time	classroom	instructor	office	department
Algebra I	8:00	C01	Mr. Reyes	B24	Math
Geometry	10:00	C01	Mr. Reyes	B24	Math
World History	9:00	C15	Ms. Tan	A11	Humanities
English Literature	10:00	C09	Ms. Larsen	A05	Humanities
Physics	11:00	C06	Ms. Musa	B22	Science
Chemistry	13:00	C17	Ms. Musa	B22	Science
Music	9:00	C25	Mr. Pal	A03	Humanities

The key for the relation shown above is `class_name`. The functional dependencies are:

- `class_name` $\longrightarrow$ `classroom`
- `class_name` $\longrightarrow$ `time`
- `class_name` $\longrightarrow$ `instructor`
- `instructor` $\longrightarrow$ `office`
- `instructor` $\longrightarrow$ `department`

Because `instructor` $\longrightarrow$ `office` and `instructor` $\longrightarrow$ `department` hold but `instructor` is not a candidate key, these transitive dependencies cause the anomalies described above. The attributes `office` and `department` are facts about the instructor, not about the class, yet they are stored in the class table.

6.2.3 First Normal Form (1NF)

All valid relations R are in **1NF**, where every nonprime attribute is functionally dependent on a key of R. What typically causes things to not be a relation / not be 1NF is that attributes have values which do not come from a domain of atomic numbers.

For example, a `student` relation where `phone_numbers` stores a comma-separated list like `"555-1234, 555-5678"` violates 1NF because the attribute is not atomic. The fix is to either create a separate `student_phones` relation or store one phone number per row.

6.2.3.1 Identifying Violation Suppose that we have relation `Student(_student_id_, name, phones)` with the following data:

student_id	name	phones
101	Jerry	404-1111, 404-2222
102	Maya	678-3333

We spot a violation of 1NF because the values in `phones` are not atomic. To fix this, we make a new table for phone numbers.

6.2.3.2 Normalizing to 1NF The new relations are: - `Student(_student_id_, name)` - `StudentPhone(_student_id_, phone)`

student_id	name
101	Jerry
102	Maya

student_id	phone
101	404-1111
101	404-2222
102	678-3333

6.2.4 Second Normal Form (2NF)

A relation is in **2NF** if it is in 1NF and every non-prime attribute is dependent on either the whole or a subset of the composite key. "Fully" means no non-prime attribute depends on just a proper subset of any candidate key. - A non-prime key *may* be indirectly dependent on a full candidate key. For example, if `key -> A` and `A -> B` are functional dependencies, and `key` is the full candidate key while `A` and `B` are non-prime attributes, then this satisfies 2NF because `B` is still dependent on the candidate key.

2NF only matters when a candidate key is composite (has more than one attribute). If every candidate key is a single attribute, the relation is automatically in 2NF.

6.2.4.1 Identifying Violation Consider a relation `Enrollment(student_id, course_id, student_name, course_name, grade)` with the following dependencies: - $student\_id \rightarrow student\_name$ - $course\_id \rightarrow course\_name$ - $(student\_id, course\_id) \rightarrow grade$

And with the following candidate key: - $(student\_id, course\_id)$

Since $(student\_id, course\_id)$ is a composite key and some non-prime attributes are dependent on only part of the key (`student_name`, `course_name`), there is a violation.

6.2.4.2 Normalizing to 2NF In order to fix the violation, we need to split one relation/table into multiple: - `Student(_student_id_, student_name)`: Now non-prime attribute `student_name` is dependent on whole candidate key `student_id`. - `Course(_course_id_, course_name`: Now non-prime attribute `course_name` is dependent on whole candidate key `course_id`. - `Enrollment(_student_id_, _course_id_, grade)`: Non-prime attribute `grade` is dependent on whole candidate key (composite key).

6.2.5 Third Normal Form (3NF)

A relation is in **3NF** if it is in 2NF and no non-prime attribute is *transitively dependent* on any candidate key. In other words, for every FD $X \rightarrow A$ where A is a non-prime attribute, either X is a superkey or A is part of some candidate key.

The intuition: every non-prime attribute must depend on "the key, the whole key, and nothing but the key."

6.2.5.1 Identifying Violation Consider a relation `Employee(emp_id, emp_name, dept_id, dept_name)` with the following dependencies: - $emp\_id \rightarrow emp\_name, dept\_id$ - $dept\_id \rightarrow dept\_name$

And with the following candidate key: - emp_id

The issue is that `dept_name` is a non-prime attribute which is transitively dependent on a candidate key. It is still dependent on the candidate key, so it's 2NF, but not 3NF.

6.2.5.2 Normalizing to 3NF The solution is to break that table up into two relations, so that one relation can be the department with id/name, and the other can be about the employee. By having multiple candidate keys, we ensure all non-prime attributes depend only on the candidate key. - `Employee(_emp_id_, emp_name, dept_id)` - `Department(_dept_id_, dept_name)`

6.2.6 Boyce-Codd Normal Form (BCNF)

A relation is in **BCNF** if for every non-trivial FD $X \rightarrow Y$, X is a superkey. This is a stricter version of 3NF: the requirement applies to *all* attributes on the right-hand side, not just non-prime ones.

3NF allows an FD $X \rightarrow A$ where A is a prime attribute and X is not a superkey. BCNF does not permit this exception. In practice, 3NF and BCNF coincide for most schemas; they differ only when a relation has multiple overlapping composite candidate keys.

6.2.6.1 Identifying Violation Consider a relation `Teaching(student, course, instructor)` with the following dependencies: - $(student, course) \rightarrow instructor$ - $instructor \rightarrow course$

We have the following candidate keys (minimal superkeys): `student, course` and `student, instructor`. Looking at the attributes, we actually see that all attributes are prime!

However, the current relation violates BCNF because even though all attributes are prime, BCNF requires the left side of the functional dependency to be a superkey (some set of fields that uniquely idenfies any row in the relation).

This fulfills 3NF because no non-prime attribute is transitively dependent on any candidate key. However, this violates BCNF because just having `instructor` alone is not sufficient for a superkey, and BCNF requires EVERY functional dependency to have a superkey on the left side.

6.2.6.2 Normalizing to BCNF Example where 3NF $\neq$ BCNF. Consider `Tutoring(student_id, subject, tutor)` with the constraint that each tutor teaches exactly one subject, and each subject can be taught by multiple tutors: - Candidate key: `{student_id, subject}` (a student picks one tutor per subject) - FD: `tutor` $\rightarrow$ `subject`

This satisfies 3NF because `subject` is a prime attribute (part of the candidate key), so the FD `tutor` $\rightarrow$ `subject` is allowed under 3NF's exception. But it violates BCNF because `tutor` is not a superkey.

To achieve BCNF, decompose into: - `TutorSubject(tutor, subject)` - `StudentTutor(student_id, tutor)`

NOTE: BCNF decomposition does not always preserve all original FDs (dependency preservation). In the example above, the constraint "a student has one tutor per subject" is lost after decomposition. This is a known trade-off: BCNF guarantees no redundancy from FDs, but may sacrifice dependency preservation. 3NF always has a dependency-preserving decomposition.

7 Database Design Concepts

7.1 Other Types of Databases

7.1.1 Choosing a Type of Database

Different types of databases optimize for different data models and access patterns.

Key-value store databases like Redis or DynamoDB use key lookups for fast access (imagine a giant distributed hash map), while values are BLOBs (binary large objects). The values can be pretty diverse, like in JSON format.

Document-based store databases like MongoDB store data in collections of documents, which are JSON-shaped. Documents are semi-structured records (allows for objects that can be queried). This is suitable for when objects are naturally self-contained but have varying fields (schema is not as set in stone as MySQL).

Graph databases model data as nodes (entries) and edges (relationships between entities). This should be used when relationships are the main thing. For example, if we wanted to find the recursive chain of bosses of a particular employee, it's super easy to follow a graph's nodes upward, whereas MySQL would struggle with this.

7.1.2 Scalability

Vertical scaling means making one database machine stronger / faster, so we upgrade a single server.

Horizontal scaling means adding more machines to spread load/data across them.

7.1.3 ACID Properties

ACID stands for atomicity, consistency, isolation, and durability. These are properties that MySQL database transactions have. A transaction refers to a group of operations that may be treated as one unit.

Atomicity means that either the whole set of operations within a transaction occurs, or nothing happens. It's an all-or-nothing concept, because if the transaction is interrupted, then the database should never get stuck in a partial state. - This is done through transactions, which group operations in a single block.

```
START TRANSACTION;

UPDATE accounts
SET balance = balance - 100
WHERE id = 1;

UPDATE accounts
SET balance = balance + 100
WHERE id = 2;

IF something_is_wrong THEN -- for example, if you expect only 1 row to be affected but multiple
    were affected
    ROLLBACK;
ELSE
    COMMIT;
END IF;
```

Consistency means that changes to the database state should not violate existing rules in the defined relationship. The database starts in a valid state and ends in a valid state. - This is enforced through a combination of database-enforced consistency (UNIQUE, CHECK, NOT NULL) as well as procedural checks (business logic in stored procedures).

Isolation means that concurrent transactions should behave as if they do not interfere with each other. For example, if a bank account has $100, and two transaction requests are received at the same time, withdrawing $80 and $30 respectively. If each computation occurred relative to the state of $100, then both withdrawls would be permitted,

and the final written value would either be $20 or $70. - This is enforced at the database level (not through SQL or application code) through locks (i.e., first transaction locks and second transaction is not started until the first has committed or rolled back).

Durability means that once data is committed, the database should retain that state even if it crashes. This way, commits are reliable and a "successful commit" is truly successful. - This is also implemented at the database level. For example, if a transaction is committed, then all procedures are already logged to disk even if the internal database state has not been updated yet. At this point, if power goes out, the database can still read the log to completely ensure an update. Furthermore, uncommited partial transactions are rolled back and undone.

7.2 Indexing

7.2.1 Motivation

An index in a database helps MySQL find rows without scanning the entire table. An index is an auxilliary data structure built on one or more columns so the database can locate matching rows faster. It is essentially the key for the B-tree underlying the search engine.

If we frequently need to perform lookups by a particular column's value (e.g., SSN for a person in a government database), then it is a good idea to create an index for SSN in the database for faster lookups. The tradeoff is that writes are more expensive, using extra disk space and a little more time (inserts/updates/deletes must also maintain index structure).

7.2.2 Overview of Primary Indexing

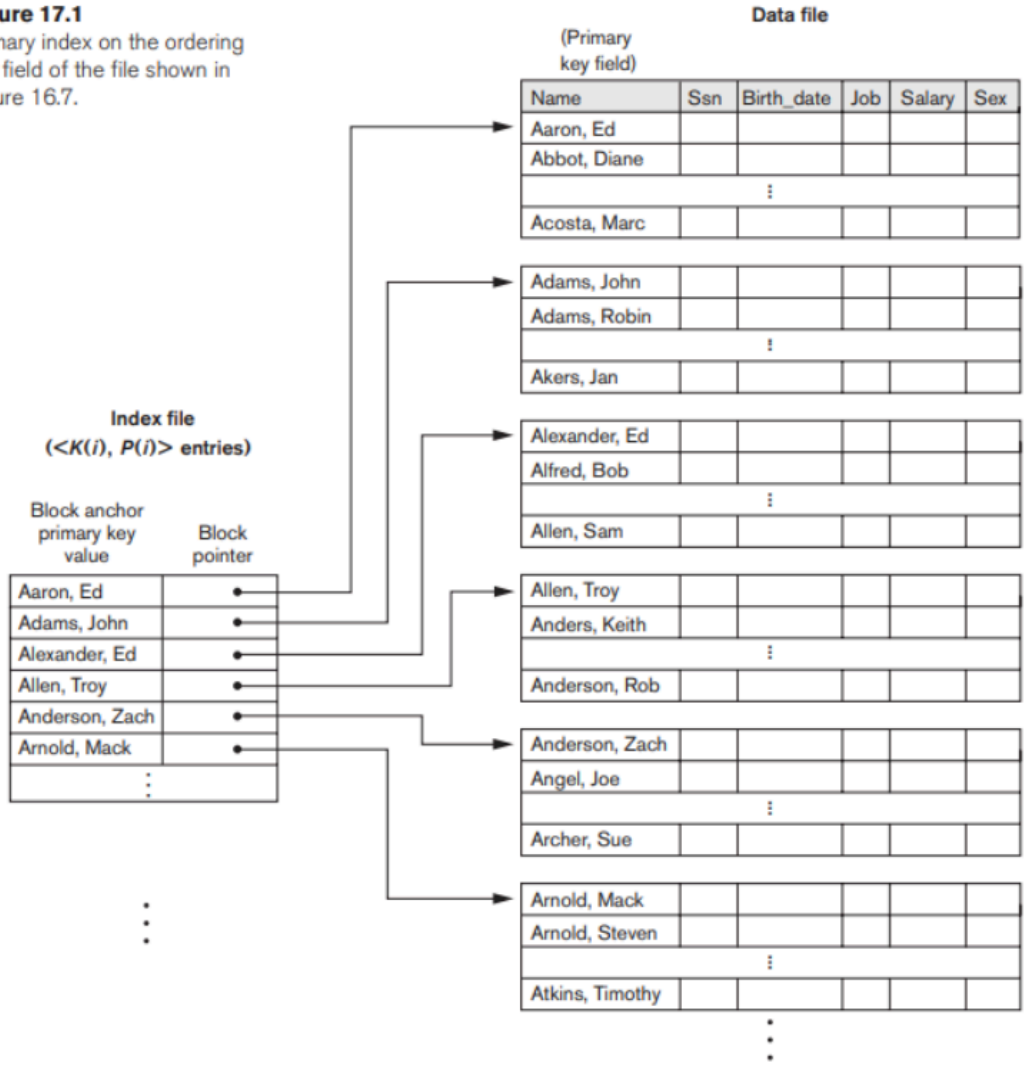
A high-level overview of how indexes work is that they help narrow down the search space. Table data on disk is stored in blocks; suppose that the numbers below are the ordered indexes.

```
10 -> block A
25 -> block B
40 -> block C
55 -> block D
70 -> block E
```

Since the index is much smaller than the table itself (i.e., each index block contains many entries), scanning an index for a key like "40" first and then looking within that block is much faster than looking through all rows (and since it's sorted, we can use something like binary search).

We say there is **primary indexing** if an index is created on an ordered column. Importantly, **in physical disk storage, rows are stored in the order of the column**.

When we use a primary index, we use block pointers. We go to the closest memory block with the index key that is <= our search value. For example, if we try to find Abbot, then we use the block pointer that Aaron maps to.



7.2.3 Overview of Clustering Indexes

A clustering index is used when the table is physically stored in the order of the clustering field, yet that field is not necessarily unique. It is useful when that field is relevant to searches (e.g., queries narrowing down entries with a particular `deptId`), but not necessarily unique amonst all rows.

Similar to primary indexing, the pointer doesn't immediately take us to the entry we're looking for. Instead, it takes us to the start of the block which contains the first appearance of an entry with the clustering field

value we're looking for. For example, we would get pointed to the first block if we looked for `Dept_number` of 2.

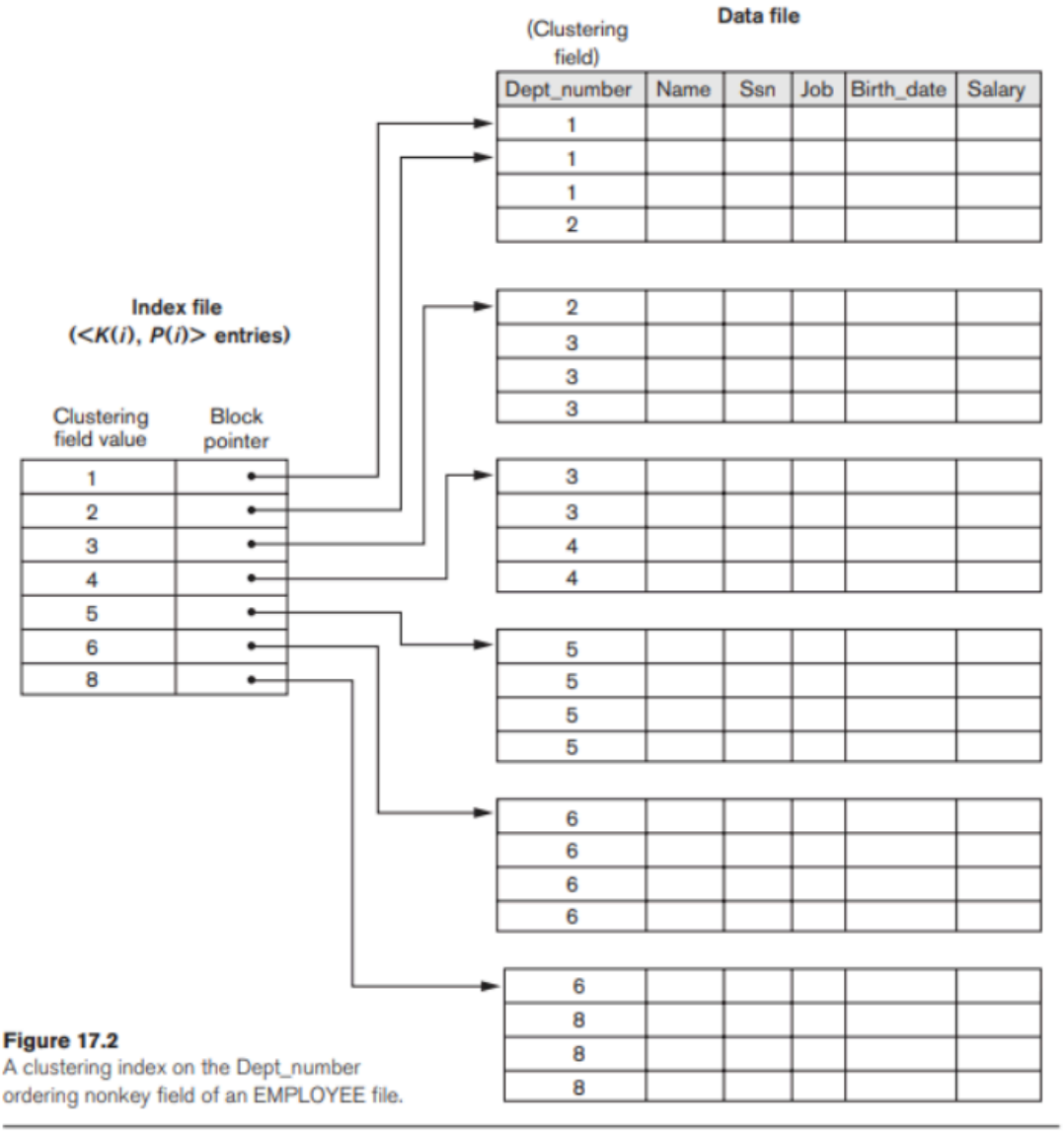


Figure 17.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

7.2.4 Types of Indexes

7.2.4.1 Types of Indexing (By Columns)

There are **unique**, **non-unique**, and **composite** indexes.

For example, if we index the SSN of a person, that is a **unique** index. No two rows share the same indexed key value. In MySQL, all primary keys are automatically indexed as a unique index, guaranteeing uniqueness and non-null.

A **non-unique** index allows for duplicate values in that column. When we need to search for particular elements within a smaller group, non-unique indexes work well. For example, setting `status` column as non-unique so it's easy to find rows which have a status of "pending".

A **composite index** works across more than one column depending on the order it is defined. For example, if the index is `INDEX(fname, lname)`, then it makes querying people by both columns easier (first finds all people with that `fname`, then finds all people with that `lname`). - The order matters when we define a composite index. For example, if I define `INDEX(A, B)`, then writing queries with conditions that target either `A` or `CONDITION A AND CONDITION B` are okay, but doing something like `WHERE CONDITIONB` is slow.

7.2.5 Creating Indexes in MySQL

```
CREATE [UNIQUE] INDEX idx_name
ON table_name (col1, [col2])
```

```
CREATE TABLE table_name
...
INDEX idx_name (cols)
```

```
ALTER TABLE table_name ADD INDEX idx_name (cols)
```