

DSA5104 Principles of Data Management and Retrieval

AY2025/26 Sem1 By Zhao Peiduo

Lecture 1

Database Systems

- DBMS: interrelated data + programs; convenient and efficient environment.
- Manage data that are highly valuable, large, and concurrently accessed.
- Modern DBMSs manage large, complex collections of data; pervasive in daily life.

Purpose of Database Systems

- Redundancy & inconsistency (multiple file formats); difficulty accessing data (new program per task).
- Data isolation (multiple files/formats) → security challenges.
- Integrity constraints buried in code → hard to add/change.
- *Atomicity*: no partial updates; e.g., fund transfer must be all-or-nothing.
- *Concurrency*: needed for performance; uncontrolled access → inconsistencies (e.g., two withdrawals).
- *Security*: restrict access to subsets of data.
- DB systems address these issues; store & retrieve data safely.

View of Data

- DB system = interrelated data + programs to access/modify.
- Provide abstract view via *data models* (concepts, relationships, semantics, constraints).

Categories of Data Models (high level)

- Relational (tables)
- Entity–Relationship (design).
- Object-based (OO/OR features).
- Semi-structured (XML/JSON).

Instances and Schemas

- *Schema*: overall design. *Instance*: data at a moment.
- Analogy: schema - variable declaration; instance - current value (class/struct blueprint vs object).

Logical vs Physical Schema & Physical Data Independence

- Logical schema: what data/relationships. Physical schema: storage layout.
- Physical data independence: change physical without changing logical; well-defined interfaces.

Data Definition Language (DDL)

- Define schema; DDL compiler → templates in data dictionary (metadata: schema, constraints, auth).
- Example: create table instructor (ID char(5), name varchar(20), dept_name varchar(20), salary numeric(8,2))

Data Manipulation Language (DML)

- Access/update data; *procedural* (what + how) vs *declarative* (what).
- Declarative DMLs easier; query-language part handles retrieval.

SQL Query Language

- Nonprocedural; input tables → one output table.
- Typically embedded or called via APIs (ODBC/JDBC); app code handles I/O/network/UI.

Engine / Components (very high level)

- Storage manager (file/buffer/authorization/transaction). Query processor (DDL interpreter, DML compiler/optimizer, eval engine).
- Query Processing stages: Parsing & translation Optimization Evaluation

Transaction Management

- Transaction = logical unit of work (e.g., transfer \$50: read/update/write A,B).
- Ensure consistency under failures; concurrency control coordinates overlapping txns.

Architectures

- Centralized/shared-memory; Client–server; Parallel (shared-memory/disk/nothing); Distributed (geo, heterogeneity).
- App tiers: two-tier (client-DB) vs three-tier (client-app server-DB); 3-tier aids dev, scale, reliability, security.

Lecture 2

Relation Schema and Instance

- A_1, A_2, \dots, A_n are *attributes*.
- $R = (A_1, A_2, \dots, A_n)$ is a *relation schema*.
- Example: instructor = (ID, name, dept_name, salary).
- A relation instance r defined over schema R is denoted by $r(R)$.
- The current values of a relation are specified by a table.
- An element t of relation r is called a **tuple**, represented by a row in a table.

Attributes

- The set of allowed values for each attribute is its **domain**.
- Attribute values are required to be **atomic** (indivisible).
- Non-atomic example: concatenation of multiple attribute values, e.g. Silberschatz, Korth, Sudarshan for author. This should be broken into several atomic rows with one author each.
- Special value null indicates “unknown”; member of every domain, which could complicate operations.

Relations are Unordered

- Order of tuples is irrelevant; tuples may be stored arbitrarily.
- Example: instructor relation with unordered tuples.

Keys: Let $K \subseteq R$

- **Superkey**: K is a superkey if values for K uniquely identify a tuple (unique identifier).
 - Example: {ID}, {ID, name} are both superkeys of instructor.
 - **SQL**: Every declared PRIMARY KEY or UNIQUE constraint implicitly defines a superkey.
- **Candidate key**: Minimal superkey (only containing elements essential for unique identification).
 - Example: {ID} is a candidate key for instructor.
 - **SQL**: Use UNIQUE to enforce candidate keys.

```
create table instructor (  
    ID varchar(5),  
    name varchar(20),  
    dept_name varchar(20),  
    salary numeric(8,2),  
    unique (name, dept_name) -- candidate key  
);
```

Keys: Let $K \subseteq R$

- **Primary key**: One candidate key chosen to uniquely identify tuples.
 - Example: ID is chosen as the primary key.
 - **SQL**:

```
create table instructor (  
    ID varchar(5) primary key,  
    name varchar(20),  
    dept_name varchar(20),  
    salary numeric(8,2)  
);
```

- **Foreign key**: Attribute in one relation that refers to the primary key in another relation.
 - Example: dept_name in instructor refers to department.dept_name.
 - **SQL**:

```
create table department (  
    dept_name varchar(20) primary key,  
    building varchar(20),  
    budget numeric(12,2)  
);  
  
create table instructor (  
    ID varchar(5) primary key,  
    name varchar(20),  
    dept_name varchar(20),  
    salary numeric(8,2),  
    foreign key (dept_name) references department(dept_name)  
);
```

Relational Query Languages

- SQL is mostly **non-procedural**: user specifies what, DB decides how.
- Three formal relational query languages:
 - Relational algebra (procedural).
 - Tuple relational calculus (non-procedural).
 - Domain relational calculus (non-procedural).

Relational Algebra

- Procedural language: operations on relations produce new relations.
- Six basic operators:
 - **Select** (σ) – filter rows.
 - **Project** (Π) – choose attributes.
 - **Union** (\cup).
 - **Set difference** ($-$).
 - **Cartesian product** (\times).
 - **Rename** (ρ).

Select Operation

- Selects tuples that satisfy a given **predicate**.
- Notation: $\sigma_p(r)$ where p is the **selection predicate**.
- Comparisons: $=, \neq, >, \geq, <, \leq$.
- Predicates can be combined: \wedge (AND), \vee (OR), \neg (NOT).
- Example: $\sigma_{\text{dept_name} = \text{'Physics'}} \wedge \text{salary} > 90000(\text{instructor})$

```
SELECT *  
FROM instructor  
WHERE dept_name = 'Physics' AND salary > 90000;
```

- Example: $\sigma_{\text{dept_name} = \text{building}}(\text{department})$

```
SELECT *  
FROM department  
WHERE dept_name = building;
```

Project Operation (Subsetting)

- RA Notation: $\Pi_{A_1, A_2, \dots, A_k}(r)$.
- Example: $\Pi_{ID, \text{name}, \text{salary}}(\text{instructor})$

```
SELECT DISTINCT ID, name, salary  
FROM instructor;
```

Composition of Operations

- Example: $\Pi_{\text{name}}(\sigma_{\text{dept_name} = \text{'Physics'}}(\text{instructor}))$

```
SELECT DISTINCT name  
FROM instructor  
WHERE dept_name = 'Physics';
```

Cartesian-Product Operation

- Example: instructor \times teaches

```
SELECT *  
FROM instructor  
CROSS JOIN teaches;
```

Join Operation

- Example: $\sigma_{\text{instructor.ID} = \text{teaches.ID}}(\text{instructor} \times \text{teaches})$
- Equivalent: $\text{instructor} \bowtie_{\text{instructor.ID} = \text{teaches.ID}} \text{teaches}$

```
SELECT *
FROM instructor
JOIN teaches
  ON instructor.ID = teaches.ID;
```

Union Operation

- Example: $\Pi_{\text{course_id}}(\sigma_{\text{semester} = \text{"Fall"}} \wedge \text{year} = 2017(\text{section})) \cup \Pi_{\text{course_id}}(\sigma_{\text{semester} = \text{"Spring"}} \wedge \text{year} = 2018(\text{section}))$
- ```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND year = 2017
UNION
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Spring' AND year = 2018;
```

**Set-Intersection Operation**

- Example:  $\Pi_{\text{course\_id}}(\sigma_{\text{semester} = \text{"Fall"}} \wedge \text{year} = 2017(\text{section})) \cap \Pi_{\text{course\_id}}(\sigma_{\text{semester} = \text{"Spring"}} \wedge \text{year} = 2018(\text{section}))$
- ```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND year = 2017
INTERSECT
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Spring' AND year = 2018;
```

Set-Difference Operation

- $\Pi_{\text{course_id}}(\sigma_{\text{semester} = \text{"Fall"}} \wedge \text{year} = 2017(\text{section})) - \Pi_{\text{course_id}}(\sigma_{\text{semester} = \text{"Spring"}} \wedge \text{year} = 2018(\text{section}))$
- ```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND year = 2017
EXCEPT
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Spring' AND year = 2018;
```

**Assignment Operation**

- Assigns the result of an expression to a temporary relation.
- Example:  $C \leftarrow \sigma_{\text{dept\_name} = \text{'Physics'}}(\text{instructor})$
- ```
CREATE TEMPORARY TABLE C AS
SELECT *
FROM instructor
WHERE dept_name = 'Physics';
```

Rename Operation

- Used to rename relations or attributes.
- Example: $\rho_X(E)$ renames the result of E to relation X .
- Example: $\rho_Y(A_1, A_2, A_3)(E)$ renames E to relation Y with attributes A_1, A_2, A_3 .
- ```
-- Rename relation
SELECT *
FROM instructor AS X;

-- Rename relation + attributes
SELECT ID AS A1, name AS A2, salary AS A3
FROM instructor AS Y;
```

**Equivalent Queries**

- Different relational algebra expressions (or SQL queries) can produce the same result.
- Equivalence:  $\sigma_{\text{dept\_name} = \text{'Physics'}}(\sigma_{\text{salary} > 90000}(\text{instructor})) \equiv \sigma_{\text{dept\_name} = \text{'Physics'}} \wedge \text{salary} > 90000(\text{instructor})$
- ```
-- Two equivalent queries
SELECT *
FROM instructor
WHERE dept_name = 'Physics' AND salary > 90000;

SELECT *
FROM (
  SELECT *
  FROM instructor
  WHERE salary > 90000
) AS temp
WHERE dept_name = 'Physics';
```

Design of Database

- Relational algebra is the theoretical foundation of SQL.
- Operators (selection, projection, joins, set operations, rename, etc.) form the core building blocks.
- Database design should enable:
 - Expressive query formulation.
 - Efficient query execution.
 - Logical independence (queries written without depending on physical storage).

SQL Parts

- **DML (Data Manipulation Language)** – query information, insert, delete, and modify tuples.
- **Integrity** – DDL commands for specifying integrity constraints.
- **View Definition** – DDL commands for defining views.
- **Transaction Control** – begin and end transactions.
- **Embedded and Dynamic SQL** – embed SQL within programming languages.
- **Authorization** – specify access rights to relations and views.

Data Definition Language (DDL)

- Schema for each relation.
- Attribute types.
- Integrity constraints.
- Indices to be maintained.
- Security and authorization.
- Physical storage structure.

Domain Types in SQL

- `char(n)` – fixed length string.
- `varchar(n)` – variable length string.
- `int` – machine-dependent integer.
- `smallint` – small integer.
- `numeric(p,d)` – fixed point with precision p and d decimals.
- `real`, `double precision` – floating point, machine dependent.
- `float(n)` – floating point with precision of at least n digits.

Create Table Construct create table r (A_1 D_1 , ..., A_n D_n , constraints...);

- r = relation name.
- A_i = attribute name, D_i = domain type.

Example:

```
create table instructor (
  ID char(5),
  name varchar(20),
  dept_name varchar(20),
  salary numeric(8,2)
);
```

Integrity Constraints

- **Primary Key** (A_1, \dots, A_n)
- **Foreign Key** (A_m) references relation r
- **Not Null**

```
create table instructor (
  ID char(5),
  name varchar(20) not null,
  dept_name varchar(20),
  salary numeric(8,2),
  primary key (ID),
  foreign key (dept_name) references department
);
```

Updates to Tables**Insert:**

```
insert into instructor values ('10211', 'Smith', 'Biology', 66000);
```

Foreign Key Violations (MySQL):

```
SET FOREIGN_KEY_CHECKS = 0; -- disable checks
insert into instructor values (...);
insert into department values (...);
SET FOREIGN_KEY_CHECKS = 1; -- enable checks
```

Foreign Key Violations (PostgreSQL):

- Define FK as *deferrable*.
- Use transactions to defer checks.

Example:

```
begin;
set constraints instructor_dept_name key deferred;
insert into instructor values (...);
insert into department values (...);
commit;
```

Delete:

```
delete from student;
```

Drop Table / Database:

```
drop table r;
drop database university;
```

Updates to Tables (Alter)

Add Attribute:

```
alter table r add A D;
```

- A = attribute name to be added.
- D = domain of A .
- Existing tuples are assigned null for the new attribute.
- Constraint condition must evaluate to TRUE or NULL.

Drop Attribute:

```
alter table r drop A;
```

- A = name of an attribute in relation r .

Basic Query Structure

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P;
```

- A_i = attributes.
- r_i = relations.
- P = predicate.
- Result = relation.

The SELECT Clause

- Lists attributes for result (*projection* in relational algebra).
- SQL names are case-insensitive (implementation-dependent for strings).
- Duplicates allowed; use distinct to eliminate:

```
select distinct dept_name
from instructor;
```

- Use all to explicitly keep duplicates:

```
select all dept_name
from instructor;
```

- Select All Attributes:

```
select *
from instructor;
```

- Select Literals

```
select '437';
select '437' as F00;
select 'A' from instructor;
```

- Arithmetic in Select:

```
select ID, name, salary/12
from instructor;

select ID, name, salary/12 as monthly_salary
from instructor;
```

The WHERE Clause

- Specifies conditions (*selection predicate*).
- Supports logical connectives (and, or, not).
- Comparisons: <, >, <=, >=, =, <>.

Example:

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 70000;
```

The FROM Clause

- Lists relations, corresponds to Cartesian product.
- Example:

```
select *
from instructor, teaches;
```

- Produces all instructor–teaches pairs.
- Common attributes renamed using relation name (e.g., instructor.ID).

The Rename Operation

- Rename relations/attributes using as.
- Syntax: old-name as new-name.
- Example:

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary
and S.dept_name = 'Comp. Sci.';
```

- as keyword is optional: instructor as T \equiv instructor T.

Self Join Example

- Relation emp_super(person, supervisor).
- Query: Find supervisor of Bob's supervisor.

Example:

```
select distinct e2.supervisor
from emp_super e1, emp_super e2
where e1.supervisor = e2.person
and e1.person = 'Bob';
```

String Operations

- Operator like with:
 - % \rightarrow matches substring.
 - _ \rightarrow matches single character.

```
select name
from instructor
where name like '%ar%';
```

- Escape for special characters:

```
like '100 \% escape '\';
```

String Operations (Cont.)

- Pattern examples:
 - 'Intro%' \rightarrow strings starting with Intro.
 - '%Comp%' \rightarrow strings containing Comp.
 - '...' \rightarrow exactly three characters.
 - '...%' \rightarrow at least three characters.
- Other operations:
 - Concatenation (||), or *concat* in MySQL.
 - Case conversion (upper/lower).
 - String length, substring extraction, etc.

```
select upper(concat(name, ' ', dept_name)),
       substring(name, 2, 3) -- start from idx 2 end with 3
from instructor
where name like '%a%T';
```

- Case sensitivity: case-insensitive in MySQL and case-sensitive in PostgreSQL.

Ordering Tuples

```
select distinct name
from instructor
order by name {asc/dsc};
```

Where Clause Predicates

- Use between:

```
select name
from instructor
where salary between 90000 and 100000;
```

- Tuple comparison:

```
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name)
     = (teaches.ID, 'Biology');
```

Set Operations

- Union, intersection and difference:

```
(select course_id from section
 where sem='Fall' and year=2017)
union / intersect / except
(select course_id from section
 where sem='Spring' and year=2018);
```

- union, intersect, except eliminate duplicates.
- To retain duplicates: union all, intersect all, except all.

Null Values

- Tuples may have attributes with null.
- null = unknown value or does not exist.
- Arithmetic with null → result is null.
- Predicate is null checks for nulls.

```
select name
from instructor
where salary is null;
```

- Predicate is not null checks for non-nulls.
- Comparisons with null → result is unknown.
- Example: $5 < \text{null}$, $\text{null} < \text{null}$, $\text{null} = \text{null}$.
- Boolean logic with unknown:
 - true and unknown = unknown
 - false and unknown = false
 - unknown and unknown = unknown
 - unknown or true = true
 - unknown or false = unknown
 - unknown or unknown = unknown
- CHECK constraints: must evaluate to true or unknown.
- WHERE clause predicates evaluating to unknown → treated as false.

Aggregation functions

- Operate on multisets of values, return single value.
- avg, min, max, sum, count.
- Group by clause:

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name;
```

- Attributes outside aggregate functions must appear in group by.
- Example (invalid):

```
select dept_name, ID, avg(salary)
from instructor
group by dept_name;
```

- Having Clause:

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name
having avg(salary) > 42000;
```

- WHERE filters before grouping.
- HAVING filters after grouping.

Nested Subqueries

- A subquery = select-from-where inside another query.
- Can appear in FROM, WHERE, SELECT clauses.
- Example structure:

```
select A1, A2, ...
from r1, r2, ...
where P;
```

- WHERE clause subquery form: $B \langle \text{operation} \rangle (\text{subquery})$.

Set Membership (Subqueries)

- in and not in for WHERE clause
- Courses offered Fall 2017 AND Spring 2018:

```
select distinct course_id
from section
where semester='Fall' and year=2017
and course_id in (
    select course_id
    from section
    where semester='Spring' and year=2018
);
```

- Courses offered Fall 2017 BUT NOT Spring 2018:

```
select distinct course_id
from section
where semester='Fall' and year=2017
and course_id not in (
    select course_id
    from section
    where semester='Spring' and year=2018
);
```

Set Membership (Cont.)

- Instructors not named Mozart or Einstein:

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein');
```

- Count distinct students taught by instructor with ID 10101:

```
select count(distinct ID)
from takes
where (course_id, sec_id, semester, year) in
    (select course_id, sec_id, semester, year
     from teaches
     where teaches.ID = 10101);
```

Set Comparison – SOME Clause

- Instructors with salary greater than *some* Biology instructor:

```
select name
from instructor
where salary > some (
    select salary
    from instructor
    where dept_name = 'Biology');
```

- Semantics: $F \langle \text{comp} \rangle \text{some } r \Leftrightarrow \exists t \in r (F \langle \text{comp} \rangle t)$

Set Comparison – ALL Clause

- Instructors with salary greater than *all* Biology instructors:

```
select name
from instructor
where salary > all (
    select salary
    from instructor
    where dept_name = 'Biology');
```

- Semantics: $F \langle \text{comp} \rangle \text{all } r \Leftrightarrow \forall t \in r (F \langle \text{comp} \rangle t)$

Test for Empty Relations

- exists $r \Leftrightarrow r \neq \emptyset$
- not exists $r \Leftrightarrow r = \emptyset$

Use of EXISTS Clause

```
select course_id
from section as S
where semester='Fall' and year=2017
and exists (select *
            from section as T
            where semester='Spring' and year=2018
            and S.course_id = T.course_id);
```

- Correlated subquery: outer variable (S) used inside subquery.

Use of NOT EXISTS Clause

```
select distinct S.ID, S.name
from student as S
where not exists (
    (select course_id
     from course
     where dept_name='Biology')
except
(select T.course_id
 from takes as T
 where S.ID = T.ID)
);
```

- Finds students who took **all** Biology courses.
- Relies on set difference: $X - Y = \emptyset \Leftrightarrow X \subseteq Y$.

Test for Absence of Duplicate Tuples

- unique(subquery) evaluates to true if no duplicates.
- Example: Courses offered at most once in 2017:

```
select T.course_id
from course as T
where unique (select R.course_id
             from section as R
             where T.course_id = R.course_id
             and R.year = 2017);
```

Subqueries in the FROM Clause

- Subqueries can be used in the FROM clause to create a temporary relation.
- Example: Find average instructor salaries of departments where avg salary > 42000

```
select dept_name, avg_salary
from ( select dept_name, avg(salary) as avg_salary
      from instructor
      group by dept_name )
where avg_salary > 42000;
```

- Equivalent alternative with alias:

```
select dept_name, avg_salary
from ( select dept_name, avg(salary)
      from instructor
      group by dept_name )
  as dept_avg(dept_name, avg_salary)
where avg_salary > 42000;
```

WITH Clause (Common Table Expressions)

- Defines a temporary relation available only to that query.
- Example: Departments with maximum budget

```
with max_budget(value) as (
  select max(budget) from department )
select dept_name
from department, max_budget
where department.budget = max_budget.value;
```

Scalar Subquery

- Scalar subquery returns a single value.
- Example: Departments with number of instructors

```
select dept_name,
  (select count(*)
   from instructor
   where department.dept_name = instructor.dept_name)
  as num_instructors
from department;
```

- Runtime error if subquery returns >1 tuple.

Modification of the Database

- Deletion of tuples from a relation.
- Insertion of new tuples.
- Updating values in some tuples.

Deletion

- Examples:

```
delete from instructor;
delete from instructor
where dept_name = 'Finance';
```

```
delete from instructor
where dept_name in (
  select dept_name
  from department
  where building = 'Watson');
```

- Delete instructors with salary < avg salary

```
delete from instructor
where salary < (select avg(salary)
               from instructor);
```

- Works in PostgreSQL but not in MySQL (error: cannot modify same table). MySQL workaround:

```
set @a = (select avg(salary) from instructor);
delete from instructor where salary < @a;
```

Case Statement for Conditional Updates

```
update instructor
set salary = case
  when salary <= 90000 then salary * 1.05
  else salary * 1.03
end;
```

Insertion

- Examples:

```
insert into course
values ('CS-437','Database Systems','Comp. Sci.',4);
```

```
insert into course(course_id, title, dept_name, credits)
values ('CS-437','Database Systems','Comp. Sci.',4);
```

```
insert into student
values ('3003','Green','Finance',null);
```

- Insert from another table:

```
insert into instructor
select ID, name, dept_name, 18000
from student
where dept_name = 'Music' and total_cred > 144;
```

- select-from-where evaluated fully before insertion.

- Avoid issues like:

```
insert into table1
select * from table1;
```

Updates

- Give a 5% salary raise to all instructors:

```
update instructor
set salary = salary * 1.05;
```

- Give a 5% raise to instructors earning less than 70000:

```
update instructor
set salary = salary * 1.05
where salary < 70000;
```

- Give a 5% raise to instructors earning below average:

```
update instructor
set salary = salary * 1.05
where salary < (select avg(salary)
               from instructor);
```

- SQL standard (PostgreSQL): evaluates condition first, then applies updates.
- MySQL: does not allow updates with the same table inside a subquery.
- Increase salaries with different conditions (Order is important!):

```
update instructor
set salary = salary * 1.03
where salary > 90000;
```

```
update instructor
set salary = salary * 1.05
where salary <= 90000;
```

- Can be replaced with case statement.

Updates with Scalar Subqueries

- Recompute and update tot_cred for all students:

```
update student S
set tot_cred = (select sum(credits)
               from takes, course
               where takes.course_id = course.course_id
                 and S.ID = takes.ID
                 and takes.grade <> 'F'
                 and takes.grade is not null);
```

- If no courses are taken, set tot_cred to null.
- To avoid nulls, use:

```
case
  when sum(credits) is not null then sum(credits)
  else 0
end
```