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
- $A_1, A_2, \ldots, A_n$ are *attributes*.
- $R = (A_1, A_2, \ldots, A_n)$ 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: $\Pi_{A_1, A_2, \ldots, A_k}(r)$.
- Example: $\Pi_{ID, name, salary}(\text{instructor})$

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

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

Union Operation

- Example: $\Pi_{\text{course\_id}}(\sigma_{\text{semester=“Fall”} \wedge \text{year}=2017}(\text{section})) \cup \Pi_{\text{course\_id}}(\sigma_{\text{semester=“Spring”} \wedge \text{year}=2018}(\text{section})$
-
```
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: $\Pi_{\text{course\_id}}(\sigma_{\text{semester=“Fall”} \wedge \text{year}=2017}(\text{section})) \cap \Pi_{\text{course\_id}}(\sigma_{\text{semester=“Spring”} \wedge \text{year}=2018}(\text{section})$
-
```
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

- $\Pi_{\text{course\_id}}(\sigma_{\text{semester=“Fall”} \wedge \text{year}=2017}(\text{section})) - \Pi_{\text{course\_id}}(\sigma_{\text{semester=“Spring”} \wedge \text{year}=2018}(\text{section}))$
-
```
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: $C \leftarrow \sigma_{\text{dept\_name='Physics'}}(\text{instructor})$
-
```
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: $\rho_{Y(A1,A2,A3)}(E)$ renames E to relation Y with attributes $A1, A2, A3$.
-
```
-- 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: $\sigma_{\text{dept\_name='Physics'}}(\sigma_{\text{salary}>90000}(\text{instructor})) \equiv \sigma_{\text{dept\_name='Physics'} \wedge \text{salary}>90000}(\text{instructor})$
-
```
-- 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).

Lecture 3

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** $(A_1, \ldots, A_n)$
- **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 FOO;
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` ≡ `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:
  - % → matches substring.
  - _ → matches single character.

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

- Escape for special characters:

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

String Operations (Cont.)
- Pattern examples:
  - 'Intro%' → strings starting with Intro.
  - '%Comp%' → strings containing Comp.
  - '___' → exactly three characters.
  - '___%' → 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_bame)),
       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 < null`, `null <> null`, `null = 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 <operation> (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: $F \ <\text{comp}>\ \textbf{some}\ r \Leftrightarrow \exists t \in r\ (F\ <\text{comp}>\ t)$

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: $F \ <\text{comp}>\ \textbf{all}\ r \Leftrightarrow \forall t \in r\ (F\ <\text{comp}>\ t)$

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. Therefore avoid calling select and insert in the same query as select will not get the inserted values.

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
```

Lecture 4

Joined Relations

- **Join operations** – combine two relations and return another relation.
- A join operation is a Cartesian product requiring tuple matches under conditions.
- Specifies attributes in the result of the join.
- Typically used as subquery expressions in the `from` clause.
- Types of joins: Natural join, Inner join, Outer join.

Natural Join in SQL

- Matches tuples with same values for **all common attributes**.
- Retains only one copy of each common column.
- Example:

```
select name, course_id
from students, takes
where student.ID = takes.ID;
```

- Equivalent natural join form:

```
select name, course_id
from student natural join takes;
```

- Multiple relations:

```
select A1, A2, ... An
from r1 natural join r2 natural join ... rn
where P;
```

Dangerous in Natural Join

- Beware of unrelated attributes with same name equating incorrectly.
- Correct example:

```
select name, title
from student natural join takes, course
where takes.course_id = course.course_id;
```

- Incorrect example:

```
select name, title
from student natural join takes natural join course;
```

- Incorrect query omits (name, title) pairs across departments.

Natural Join with Using Clause

- `using` allows explicit column specification to avoid ambiguity.
- Example:

```
select name, title
from (student natural join takes)
join course using (course_id);
```

Join Condition

- `on` condition specifies general predicate for join.
- Equivalent to `where` but uses `on`.
- Example:

```
select *
from student join takes
on student.ID = takes.ID;
```

- Equivalent form:

```
select *
from student, takes
where student.ID = takes.ID;
```

Outer Join

- Extension of join that avoids loss of information.
- Adds non-matching tuples with `null` values.
- Types: Left Outer Join, Right Outer Join, Full Outer Join.

Left Outer Join

- Example: `course natural left outer join prereq`
- Keeps all tuples from left relation, adds nulls if no match.

Right Outer Join

- Example: `course natural right outer join prereq`
- Keeps all tuples from right relation, adds nulls if no match.

Full Outer Join

- Example: `course natural full outer join prereq`
- Keeps all tuples from both relations, filling with nulls if no match.
- MySQL does not support full outer join; requires `union`.

Joined Types and Conditions

- **Join operations** – take two relations and return another relation.
- Used as subquery expressions in the `from` clause.
- **Join condition** – defines which tuples in two relations match.
- **Join type** – defines how unmatched tuples are treated.
- Join types: inner join, left outer join, right outer join, full outer join.
- Join conditions: `natural`, `on <predicate>`, `using(A1, A2, ..., An)`.
- A left outer join preserves tuples in `A`.
- A right outer join preserves tuples in `B`.
- A full outer join preserves tuples in both.
- An inner join does not preserve non-matched tuples.

Views

- Not always desirable to expose full logical model to all users.
- Example: show instructor's ID, name, dept, but hide salary.

```
select ID, name, dept_name
from instructor;
```

- A **view** hides data and acts as a virtual relation.
- Defined using:

```
create view v as <query expression>;
```

- The view name `v` refers to a virtual relation.
- Saves an expression instead of creating a new relation.
- Expression is substituted into queries using the view.

View Definition and Use

- Hide salary:

```
create view faculty as
select ID, name, dept_name
from instructor;
```

- Query Biology instructors:

```
select name
from faculty
where dept_name = 'Biology';
```

- Dept salary totals:

```
create view departments_total_salary
    (dept_name, total_salary) as
select dept_name, sum(salary)
from instructor
group by dept_name;
```

Views Defined Using Other Views

- Views can depend on other views.
- **Depend directly** - v_1 uses v_2 in its definition.
- **Depend on** - if direct or through dependency path.
- **Recursive** - a view depends on itself.
- **Auto-Cascade** - The view nested in another view will always be expanded to its original select clause, thereby whenever we update the fields in the nested view, we will get the outer view updated as well.
- Example:

```
create view physics_fall_2017 as
select course.course_id, sec_id,
       building, room_number
from course, section
where course.course_id = section.course_id
  and dept_name='Physics'
  and semester='Fall'
  and year='2017';

create view physics_fall_2017_watson as
select course_id, room_number
from physics_fall_2017
where building='Watson';
```

View Expansion

- A view can be expanded by substituting definitions.
- Example: expand `physics_fall_2017_watson`.
- Repeat expansion until no view relations remain.
- Terminates if views are not recursive.

Materialized Views

- Some DBMS store physical copies of views (**materialized view**).
- Must be maintained when underlying relations change.
- Requires updates to keep consistent.

Update of a View

- Insert into view must translate into base relation.
- Example:

```
insert into faculty
values ('30765','Green','Music');
```

- Must insert into instructor (salary needed).
- Two options:
  - Reject insert.
  - Insert tuple with null for salary.

Some Updates Cannot be Translated Uniquely

- ```
  create view instructor_info as
  select ID, name, building
  from instructor, department
  where instructor.dept_name = department.dept_name;

  insert into instructor_info
  values ('69987','White','Taylor');
  ```
- Issues:
  - Which department if multiple exist in Taylor?
  - What if no department is in Taylor?

And Some Not at All

- ```
  create view history_instructors as
  select *
  from instructor
  where dept_name='History';
  ```
- Insert issue:

```
insert into history_instructors
values ('25566','Brown','Biology',100000);
```

- With `with check option`, rows must satisfy view condition.

View Updates in SQL

- Updates usually allowed only on **simple views**.
- Rules:
  - `from` clause has only one relation (only one single base table).
  - `select` clause only attributes, no expressions, aggregates, or distinct.
  - Unlisted attributes can be set to `null`.
  - Query has no `group by` or `having`.

Transactions

- A transaction = sequence of queries/updates, a "unit" of work.
- Begins implicitly when an SQL statement executes.
- Must end with:
  - `commit work` – make updates permanent.
  - `rollback work` – undo updates.
- Atomic: all-or-nothing execution.
- Isolated from concurrent transactions.
- In MySQL, `autocommit` is enabled by default.
- Use `start transaction` to disable autocommit, then end with `commit` or `rollback`.

Variables in MySQL

- Three types: **user-defined**, **local**, **system**.
- User-defined (`@var`) – session variables, no declaration needed.

```
set @var=5;
select @var := 5;
```

- Local variables (`var`) – used only in stored procedures, must be declared.
- System variables (`@@var`) – predefined.

Integrity Constraints

- Prevent accidental damage, ensure consistency.
  - Checking account balance > $10,000.
  - Bank salary at least $4.00/hour.
  - Customer must have non-null phone number.

Constraints on a Single Relation

- `not null`
- `primary key`
- `unique`
- `check(P)` where P is a predicate

Not Null Constraints

```
name varchar(20) not null,
budget numeric(12,2) not null
```

Unique Constraints

- `unique(A1, A2, ..., Am)` defines candidate key.
- Candidate keys can be null (unlike primary keys).

Domains

- `create domain` defines user-defined types (SQL-92).

```
create domain person_name char(20) not null;
```

- Domains can include constraints (`not null`, `check`).

```
create domain degree_level varchar(10)
constraint degree_level_test
check (value in ('Bachelors','Masters','Doctorate'));
```

Index Creation

- Index improves query performance by avoiding full scans.
- Command:

```
create index <name>
on <relation-name>(attribute);
```

- MySQL: auto-indexes PK + FK.
- PostgreSQL: does not auto-index FK.

Index Creation Example

```
create table student (
  ID varchar(5),
  name varchar(20) not null,
  dept_name varchar(20),
  tot_cred numeric(3,0) default 0,
  primary key (ID),
  foreign key (dept_name)
    references department(dept_name)
    on delete set null
);
create index studentID_index on student(ID);
```

Query:

```
select * from student where ID='12345';
```

Uses index for efficient lookup.

B^+-Tree Index Files

- Rooted tree; paths root→leaf same length.
- Non-root/leaf node: ceil[n/2] - n children.
- Leaf: ceil[(n-1)/2] - (n-1) values.
- Root:
  - If not leaf then at least 2 children.
  - If leaf then 0 - (n-1) values.

Example B^+-Tree (n=6)

- Leaf: 3–5 values.
- Non-leaf: 3–6 children.
- Root: at least 2 children.

Queries on B^+ Trees

- Search-key values inside nodes kept sorted.

Static Hashing

- Bucket = storage unit (disk block).
- Hash function $h : K \to B$ maps key → bucket.
- Example: $h(76766) = 0, h(10101) = 3, h(45565) = 1$.
- Hash index: bucket stores pointers to records.
- Hash file organization: buckets store records.

Handling Bucket Overflows

- Causes: insufficient buckets, skewed distribution.
- Skew reasons:
  - Many records → same bucket.
  - Poor hash function (non-uniform distribution).
- Solution: use **overflow buckets**.

Authorization

- Privileges on data:
  - **Read** – view only.
  - **Insert** – add new data.
  - **Update** – modify existing data.
  - **Delete** – remove data.
- Privileges on schema:
  - **Index** – create/drop indexes.
  - **Resources** – create new relations.
  - **Alteration** – add/delete attributes.
  - **Drop** – delete relations.

Lecture 5

MySQL with Python

- Use the PyMySQL library to connect Python with MySQL.
- Install via: `pip install PyMySQL`.
- Connection object `conn`:
  - Handles connecting to the database.
  - Sends queries, manages transactions, and creates cursors.
- Cursor object `cur`:
  - Executes queries and fetches results.

```python
import pymysql

conn = pymysql.connect(
    host='127.0.0.1',
    user='root',
    password='ZQSLzwzw100',
    database='university'
)
cur = conn.cursor()

try:
    cur.execute('select * from instructor where salary > 90000')
    results = cur.fetchall()
    for row in results:
        print(row)
finally:
    cur.close()
    conn.close()
```

With Statements: Use `with` blocks to automatically close connections and cursors.

```python
import pymysql

connection_params = {
    'host': '127.0.0.1',
    'user': 'root',
    'password': 'ZQSLzwzw100',
    'database': 'university'
}

with pymysql.connect(**connection_params) as conn:
    with conn.cursor() as cur:
        cur.execute('select * from instructor where salary > 90000')
        results = cur.fetchall()
        for row in results:
            print(row)
```

SQLite with Python

- Use SQLAlchemy and pandas to integrate SQLite into Python.
- Handles resource cleanup automatically.

```python
import pandas as pd
from sqlalchemy import create_engine, text

engine = create_engine('sqlite:///olist.db')

# Load CSV into SQLite
df = pd.read_csv('./kaggle_data/products.csv')
df.to_sql('products', engine, index=False, if_exists='replace')

# Query with context manager
with engine.connect() as connection:
    query = text("SELECT * FROM sellers LIMIT 2")
    result = connection.execute(query).fetchall()
    print(result)
```

Clarification

- The standard SELECT, INSERT, UPDATE, and DELETE statements are **declarative**.
- SQL-92 introduced **SQL/PSM (Persistent Stored Modules)**, a procedural extension:
  - BEGIN...END blocks
  - DECLARE for variables
  - Control flow (IF, CASE, LOOP, WHILE)
  - Exception conditions and declaring handlers

Functions and Procedures

- Encapsulate business logic inside DB.
- SQL syntax is standardized, but implementations vary.
- **Functions**: return something, used in SQL expressions. Cannot contain commit/rollback. Example: `select dept_count('History');`
- **Procedures**: do something (not necessarily return). Called via `call`. Example: `call dept_count_proc('History', return_val);`

Declaring SQL Functions

```sql
create function dept_count(dept_name varchar(20))
returns integer
begin
   declare d_count integer;
   select count(*) into d_count
   from instructor
   where instructor.dept_name = dept_name;
   return d_count;
end;
```

Usage:

```sql
select dept_name, budget
from department
where dept_count(dept_name) > 12;
```

Functions in SQL only:

```sql
create function dept_count(dept_name varchar(20))
returns integer as $$
   select count(*)
   from instructor
   where instructor.dept_name = dept_count.dept_name;
$$ language sql;
```

Table Functions (SQL Standard)

```sql
create function instructor_of(dept_name varchar(20))
returns table(
   ID varchar(5),
   name varchar(20),
   dept_name varchar(20),
   salary numeric(8,2))
return table(
   select ID, name, dept_name, salary
   from instructor
   where instructor.dept_name = instructor_of.dept_name
);
```

Usage: `select * from table(instructor_of('Music'));`

Delimiter in MySQL

- MySQL client treats ; as end of statement. To define multi-statement procedures, redefine delimiter temporarily.
- Example: `delimiter //` procedure body ... `// delimiter ;`
- This ensures the full body (with ;) is passed to server.

SQL Procedures

- Functions can also be written as procedures.
- Example:

```sql
create procedure dept_count_proc (
    in dept_name varchar(20),
    out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name;
end;
```

- `in` = input params, `out` = output params.
- Invoked using `call`:

```sql
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

MySQL Procedures

```sql
drop procedure if exists dept_count_proc;
delimiter //
create procedure dept_count_proc(
    in dept_name_str varchar(20),
    out d_count int)
begin
    select count(*) into d_count
    from instructor
    where dept_name = dept_name_str;
end //
delimiter ;

call dept_count_proc('Physics', @num_count);
select @num_count;
```

Language Constructs

- Compound statement: `begin ... end`.
- While and repeat loops:

```sql
while boolean_expr do
    statements;
end while;

repeat
    statements;
until boolean_expr
end repeat;
```

For Loop (SQL Standard)

```
declare n integer default 0;
for r as
    select budget from department
    where dept_name = 'Music'
do
    set n = n + r.budget;
end for;
```

MySQL: Loop Example

```
drop procedure if exists fib;
delimiter //
create procedure fib(in n int, out answer int)
begin
    declare i int default 2;
    declare p, q int default 1;
    set answer = 1;
    loop1: loop
        if i >= n then leave loop1; end if;
        set answer = p + q;
        set p = q;
        set q = answer;
        set i = i + 1;
    end loop loop1;
end //
delimiter ;
call fib(7, @answer);
select @answer;
```

MySQL While Example

```
create procedure fib(in n int, out answer int)
begin
    declare i int default 2;
    declare p, q int default 1;
    set answer = 1;
    while i < n do
        set answer = p + q;
        set p = q;
        set q = answer;
        set i = i + 1;
    end while;
end;
```

MySQL Repeat Example

```
create procedure fib(in n int, out answer int)
begin
    declare i int default 1;
    declare p int default 0;
    declare q int default 1;
    set answer = 1;
    repeat
        set answer = p + q;
        set p = q;
        set q = answer;
        set i = i + 1;
    until i >= n end repeat;
end;
```

MySQL: Loop

```
delimiter //
create procedure sum_budget()
begin
    declare n int default 0;
    declare r_budget numeric(12,2);
    declare finished integer default 0;
    -- 1. Declare the cursor for the query
    declare cur cursor for
        select budget from department;
    -- 2. Declare a NOT FOUND handler to break the loop
    declare continue handler for NOT FOUND set finished = 1;
    open cur; -- 3. Open the cursor
    get_budget: loop
        fetch cur into r_budget; -- 4. Fetch the row
        if finished = 1 then
            leave get_budget; -- Exit the loop
        end if;
        set n = n + r_budget; -- Accumulate
    end loop get_budget;
    close cur; -- 5. Close the cursor
    select n as total_budget; -- Return result
end //
delimiter ;
call sum_budget();
```

MySQL: While

```
delimiter //
create procedure sum_budget_using_while()
begin
    declare n int default 0;
    declare r_budget numeric(12,2);
    declare finished integer default 0;
    declare cur cursor for
        select budget from department;
    declare continue handler for NOT FOUND set finished = 1;
    open cur;
    fetch cur into r_budget; -- First row
    while finished = 0 do
        set n = n + r_budget; -- Accumulate
        fetch cur into r_budget; -- Next row
    end while;
    close cur;
    select n as total_budget; -- Return
end //
delimiter ;
call sum_budget_using_while();
```

Language Constructs: if-then-else

```
if boolean_expression
    then statement
elseif boolean_expression
    then statement
else
    statement
end if;
```

Example: Handle Exception Condition

```
declare out_of_classroom_seats condition;
declare exit handler for out_of_classroom_seats
begin
   -- Exception handling logic
end;

-- Raise exception
signal out_of_classroom_seats;
```

MySQL Example (from manual)

- Docs: `https://dev.mysql.com/doc/refman/9.4/en/create-procedure.html`
- Error code: 1062, SQLSTATE 23000 (duplicate entry).
- SQLSTATE values come from ANSI SQL/ODBC.

```
drop table if exists test_table;
create table test_table (s1 int, primary key (s1));

drop procedure if exists handlerdemo;
delimiter //
create procedure handlerdemo()
begin
    declare continue handler
        for sqlstate '23000'
        set @x = 1;
    insert into test_table values (1);
    set @x = 2;
    insert into test_table values (1);
    set @x = 3;
end;
//
delimiter ;

call handlerdemo();
select @x;
```

Triggers

- A **trigger** is executed automatically as a side effect of a modification to the database.
- To design a trigger mechanism:
  - Specify the conditions under which the trigger executes.
  - Specify the actions to be taken when it executes.
- Introduced in SQL:1999, but supported earlier with non-standard syntax.
- Syntax may differ depending on the database system.

Triggering Events and Actions in SQL

- Events: `INSERT`, `DELETE`, `UPDATE`.
- Can reference attribute values before/after update:
  - `referencing old row as orow`.
  - `referencing new row as nrow`.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
    set nrow.grade = null;
end;
```

Trigger in MySQL

```
delimiter //
create trigger trigger_name
    trigger_time trigger_event on table_name
    for each row
begin
   ...
end //
delimiter ;
```

Trigger Times and Events in MySQL

- BEFORE: INSERT, UPDATE, DELETE
- AFTER: INSERT, UPDATE, DELETE

Docs: https://dev.mysql.com/doc/refman/9.4/en/trigger-syntax.html

- BEFORE trigger: activated before attempting modification.
- AFTER trigger: activated only if BEFORE triggers succeed.
- Errors during trigger execution fail the entire statement.

Trigger to Maintain credits_earned Value

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
     and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred = tot_cred +
       (select credits from course
        where course.course_id = nrow.course_id)
    where student.id = nrow.id;
end;
```

MySQL Example: Maintain credits_earned

```
delimiter $$
create trigger credits_earned after update on takes
for each row
begin
    if (new.grade <> 'F' and new.grade is not null)
       and (old.grade = 'F' or old.grade is null) then
        update student
        set tot_cred = tot_cred +
            (select credits from course
             where course.course_id = new.course_id)
        where student.id = new.id;
    end if;
end $$
delimiter ;

select * from takes where ID = '98988';
update takes set grade = 'A'
where ID = '98988' and course_id = 'BIO-301';
select * from student where ID = '98988';
```

Recursive Queries (SQL:1999)

- Handle recursive relationships (e.g., prerequisites).
- Use with recursive clause.

```
with recursive rec_prereq(course_id, prereq_id) as (
    select course_id, prereq_id
    from prereq
  union
    select p.course_id, r.prereq_id
    from prereq p, rec_prereq r
    where p.prereq_id = r.course_id
)
select * from rec_prereq;
```

Ranking Functions

- Introduced in SQL:2003.
- Functions: rank(), dense_rank(), ntile(n).

```
select ID, salary, rank() over (order by salary desc) as rnk
from instructor;
```

Dense Rank

```
select ID, salary, dense_rank()
       over (order by salary desc) as drank
from instructor;
```

Ntile Example

```
select ID, salary, ntile(4)
       over (order by salary desc) as quartile
from instructor;
```

MySQL Window Functions (v8+)

```
select name, dept_name, salary,
       rank() over (order by salary desc) as rnk
from instructor;


select name, dept_name, salary,
       dense_rank() over (order by salary desc) as drank
from instructor;


select name, salary,
       ntile(4) over (order by salary desc) as quartile
from instructor;
```

Advanced Aggregation Features

- grouping sets: multiple groupings in one query.
- rollup: hierarchical aggregations.
- cube: all possible groupings.

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


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

Windowing

- Used to smooth out random variations.
- Example (moving average): Average sales over current, previous, and next day.

```
select date, sum(value) over
(order by date rows between 1 preceding and 1 following)
from sales;
```

Other Specifications

- between rows unbounded preceding and current
- rows unbounded preceding
- range between 10 preceding and current row
- range interval 10 day preceding

Windowing: MySQL

```
create table sales (
    s_buyer varchar(12),
    s_date date,
    s_value real
);

insert into sales values('A','2020-01-01',5);
insert into sales values('B','2020-01-03',5);

select s_date,
    sum(s_value) over
    (order by s_date rows between 1 preceding and 2 following)
from sales;
```

Windowing with Partitions

```
select account_number, date_time,
    sum(value) over (
        partition by account_number
        order by date_time
        rows unbounded preceding) as balance
from transaction
order by account_number, date_time;
```

```
select s_buyer, s_date,
    sum(s_value) over (
        partition by s_buyer
        order by s_date
        rows unbounded preceding) as balance
from sales
order by s_buyer, s_date;
```

Window Functions

- Aggregate: SUM, COUNT, AVG, VARIANCE, STDDEV, MIN, MAX
- Non-aggregate: RANK, DENSE_RANK, ROW_NUMBER, NTILE(n)
- Docs: `https://dev.mysql.com/doc/refman/9.4/en/window-function-descriptions.html`

```
window_function(expr) over (
    [partition by ...]
    [order by ...]
    [frame_clause]
)
```

Cross Tabulation (Pivot Table)

- Example: sales by item_name and color
- Rows: dimension attributes
- Columns: dimension attributes
- Cells: aggregate values

```
select item_name,
    sum(case color when 'dark' then quantity end) as dark,
    sum(case color when 'pastel' then quantity end) as pastel,
    sum(case color when 'white' then quantity end) as white
from sales
group by item_name;
```

Data Cube

- A **data cube** is a multidimensional generalization of a cross-tab.
- Can have n dimensions (3 shown as example).
- Cross-tabs can be used as views on a data cube.

Hierarchies on Dimensions

- A **hierarchy** on dimension attributes allows viewing data at different levels of detail.
- Example: DateTime $\rightarrow$ aggregate by hour, date, day of week, month, quarter, year.

Cross Tabulation with Hierarchy

- Cross-tabs can be extended to handle hierarchies.
- Can **drill down** or **roll up** along a hierarchy.

Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations.
- The value `all` is used for aggregates.
- SQL standard uses `null` in place of `all`, despite confusion with normal null values.

Extended Aggregation to Support Data Analytics (Cube)

- `cube` computes union of group by's on every subset of attributes.
- Example relation: `sales(item_name, color, size, quantity)`.

```
select item_name, color, size, sum(quantity)
from sales
group by cube(item_name, color, size);
```

- Produces all subsets: {(item_name, color, size), (item_name, color), (item_name, size), (color, size), (item_name), (color), (size), ()}.

Extended Aggregation (Rollup)

- `rollup` generates union on every prefix of attribute list.

```
select item_name, color, size, sum(quantity)
from sales
group by rollup(item_name, color, size);
```

- Produces: {(item_name, color, size), (item_name, color), (item_name), ()}.

Rollup in MySQL

```
select item_name, color, sum(quantity)
from sales
group by item_name, color with rollup;
```

Data Analytics with `grouping()`

- `grouping(attr)` returns 1 if value is null (aggregate), 0 otherwise.

```
select item_name, color, size, sum(quantity),
       grouping(item_name) as item_name_flag,
       grouping(color) as color_flag,
       grouping(size) as size_flag
from sales
group by cube(item_name, color, size);
```

Lecture 6

Application Programs and User Interfaces

- Most database users do not use query languages like SQL.
- Application programs act as intermediaries between users and the database.
- Applications are split into:
  - Front-end: user interface
  - Middle layer: business logic, security, transformations
  - Backend: data access
- Front-end interfaces: forms, graphical UIs, many are web-based.

The World Wide Web

- Distributed information system based on hypertext.
- Most documents are in HTML.
- HTML contains:
  - Text with font specs and formatting.
  - Hyperlinks to other documents.
  - Forms for user input.

Uniform Resource Locators (URL)

- **Scheme:** protocol (http/https).
- **Domain Name:** which web server is requested.
- **Port:** access gate to resources.
- **Path:** file location on server.
- **Parameters:** e.g., ?key1=value1&key2=value2.
- **Anchor:** jump to specific part in document.

HTML

- Provides formatting, hypertext links, and image display.
- Examople of html script that supports input:
  - Select options (menus, checklists, radios).
  - Enter values (text boxes).
- Input is sent back to the server for processing.

```
<html>
<body>
<table border>
<tr> <th>ID</th> <th>Name</th> <th>Department</th> </tr>
<tr> <td>00128</td> <td>Zhang</td> <td>Comp. Sci.</td> </tr>
....
</table>
<form action="PersonQuery" method="get">
Search for:
<select name="persontype">
    <option value="student" selected>Student</option>
    <option value="instructor">Instructor</option>
</select> <br>
Name: <input type=text size=20 name="name">
<input type=submit value="submit">
</form>
</body>
</html>
```

Client-Side Scripting

- Scripts embedded in web pages, executed in safe mode on client.
- Examples: Javascript, Defunct: Flash, VRML, Applets.
- Allow:
  - Local execution for animations.
  - Input validation.
  - Interactive documents.

Javascript

- Widely used for Web 2.0 rich interfaces.
- Functions:
  - Validate inputs.
  - Modify displayed page using DOM.
- Works with AJAX to fetch/modify data without reload.

Javascript Example

- Example: validate form input.
- Checks that "credits"field is a valid number within range.
- Alerts user if condition fails.

```
<html>
<head>
<script type="text/javascript">
function validate() {
    var credits=document.getElementById("credits").value;
    if (isNaN(credits) || credits<=0 || credits>=16) {
        alert("Credits must be a number greater than 0 and less than 16");
        return false;
    }
}
</script>
</head>
<body>
<form action="createCourse" onsubmit="return validate()">
Title: <input type="text" id="title" size="20"><br>
Credits: <input type="text" id="credits" size="2"><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Client/Server Request/Response Sequence

1. Enter `http://server.com` into browser.
2. Browser consults DNS for IP of `server.com`.
3. Browser issues request for home page.
4. Request crosses internet, reaches web server.
5. Server fetches page from disk.
6. Server detects PHP, passes to PHP interpreter.
7. PHP interpreter executes PHP code.
8. If PHP contains SQL, interpreter sends to MySQL DB.
9. MySQL DB returns result to PHP interpreter.
10. PHP interpreter returns PHP + DB results to server.
11. Web server sends response to client (displayed).

Variables and Functions in PHP

- Demonstrates functions, date formatting, and concatenation.

```
<?php
$temp = "Yesterday is ";
echo $temp . longdate(time() - 1*24*60*60);
function longdate($timestamp) {
    return date("l F jS Y", $timestamp);
}
?>
```

Control Flow in PHP

```
<?php
for ($count = 1; $count <= 3; $count++) {
    echo "Post $count. <br>";
}
$articles = ['First post.', 'Second post.', 'Third post.'];
if (empty($articles)) {
    echo "No article found.";
} else {
    foreach ($articles as $article) {
        echo $article . "<br>";
    }
}
?>
```

PHP Arrays: Numeric and Associative

- **Numeric array**: index = integer, starts at 0.
- **Associative array**: key-value pairs.
- Example:

```
<?php
$array = array("foo"=>"bar", "bar"=>"foo", 100=>-100, -100=>100);
var_dump($array);
?>
```

Output:

```
array(4) {
 ["foo"]=> string(3) "bar"
 ["bar"]=> string(3) "foo"
 [100]=> int(-100)
 [-100]=> int(100)
}
```

HTTP Request Methods

- Client (browser) sends **HTTP request**, server responds.
- Common methods: **GET**, **POST**.
- Forms often send input via GET or POST.

GET

- Retrieves data; parameters added to URL as query string.
- Example: `action.php?name=John&age=30`
- PHP parses with `$_GET`.

POST

- Sends data in body of HTTP request.
- PHP parses with `$_POST`.

Passing Data in the URL

- `$_SERVER['QUERY_STRING']` → query part of URL.
- `$_SERVER['REQUEST_METHOD']` → GET/POST/PUT/HEAD.
- `$_GET` → associative array of query parameters.

Forms and Auto-Increment IDs

- An HTML `<form>` submits data via the **HTTP POST method** if its `method="POST"` attribute is set.
- Example:

```
<form action="insert.php" method="POST">
  Name: <input type="text" name="username">
  <input type="submit" value="Submit">
</form>
```

- PHP script handles the form input using `$_POST`.
- After inserting into a MySQL table with an `AUTO_INCREMENT` column, the function `mysqli_insert_id()` returns the last generated ID.

Lecture 7

The Database Design Journey

- Requirements – what information the app must store.
- Conceptual (ER) – entities & relationships.
- Logical – convert ER diagram into tables & columns.
- Schema Refinement – remove redundancy using FDs & BCNF.
- Physical – performance choices (indexes, storage).
- Security – who can see or change data.

Why Refine a Schema?

- Avoid redundancy.
- Prevent anomalies (update, insert, delete issues).

Functional Dependencies (FDs)

- FD: $X \to Y$ ("X determines Y").
- Means: if two tuples have the same X, then their Y must also be the same.
- Formally: $\forall t_1, t_2 \in r, \pi_X(t_1) = \pi_X(t_2) \implies \pi_Y(t_1) = \pi_Y(t_2)$.
- FD applies to all allowable instances, based on semantics.
- Not symmetric: $X \to Y \not\Longleftrightarrow Y \to X$.

Keys: Superkeys, Candidate Keys, Primary Key

- Superkey (SK): uniquely identifies tuples. $SK \to \{all\ attributes\}$.
- Candidate Key (CK): minimal superkey, no subset can still be a key.
- Primary Key: chosen CK to uniquely identify records.

Detecting Redundancy with FDs

- Example FD: $R(ating) \to W(age\_per\_hour)$.
- R not a key ⇒ (rating, wage) pairs repeat (redundancy).
- S (SSN) is a candidate key ⇒ ensures uniqueness.

Fixing Redundancy by Decomposition

- If FD's determinant isn't a key ⇒ split the table.
- $R \to W$ problematic, so decompose relation to avoid anomalies.

Implication & Closure

- An FD g is implied by a set of FDs F if g holds whenever all FDs in F hold.
- Closure F^+: the set of all FDs implied by F.

Rules of Inference (Armstrong's Axioms)

Reflexivity: If $X \supseteq Y$, then $X \to Y$
Augmentation: If $X \to Y$, then $XZ \to YZ$
Transitivity: If $X \to Y$ and $Y \to Z$, then $X \to Z$
Additional:

- Union: $X \to Y$ and $X \to Z \Rightarrow X \to YZ$
- Decomposition: $X \to YZ \Rightarrow X \to Y$ and $X \to Z$
- Pseudotransitivity: $X \to Y$ and $QY \to Z \Rightarrow QX \to Z$

Computing F^+

```
F+ = F
apply reflexivity
repeat
  for each FD f in F+: apply augmentation
  for each pair f1,f2 in F+: apply transitivity
until F+ stabilizes
```

Attribute Closure

- Closure of F is exponential in attributes.
- X^+: all attributes functionally determined by X.
- Check if $X \to Y \in F^+$ by testing if $Y \subseteq X^+$.

Computing X^+

```
X+ := X
repeat
  for each FD U → V in F
    if U \subseteq X+ then X+ := X+ \cup V
until X+ does not change
```

Uses:

- Test if $X \to Y$ is in F^+.
- Check if X is key: if $X^+ = R$, X is a superkey.

Applications of X^+:

- Superkey test
- Candidate key test
- FD test
- Compute F^+ by checking closures

The Notion of Normal Forms

- If relation has redundancy ⇒ not in normal form.
- In normal form (e.g., BCNF): redundancy/anomalies avoided.

Basic Normal Forms:

- 1NF: attributes atomic, no duplicates.
- 2NF, 3NF: historical.
- BCNF: modern goal.

Boyce-Codd Normal Form (BCNF):

- For all $X \to A \in F^+$, either $A \subseteq X$ (trivial) or X is a superkey.
- Meaning: only key constraints define non-trivial FDs.

Decomposition of a Schema

- To normalize a relation schema, decompose into multiple normalized relation schemas.
- A decomposition of R (attributes $A_1, \ldots, A_n$) replaces R with two or more schemas such that:
  - Each new schema contains a **subset** of attributes of R.
  - Every attribute A_i of R appears in at least one new schema.

Problems with Decompositions

1. Original relation reconstruction may be impossible. *(Not an issue in SNL RWH example).*
2. Dependency checking may require joins. *(Not an issue in SNL RWH example).*
3. Some queries become more expensive. *e.g., How much does M earn?*

Lossless Decomposition

- $R = X \cup Y$
- Lossless if replacing R with $X \cup Y$ preserves all information:

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

- Otherwise, decomposition is lossy.

Lossless Join Decomposition

- Definition Decomposition of R into X and Y is lossless-join wrt F if

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

- Always: $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$.
- To avoid problem #1 (loss), decompositions must be lossless.

Lossless Decomposition & FDs

- **Theorem:** Decomposition of R into X and Y is lossless wrt F if F^+ contains:

$$(X \cap Y) \to X \quad \text{or} \quad (X \cap Y) \to Y$$

- **Corollary:** If $X \to Z$ and $X \cap Z = \emptyset$, then $R - Z$ and XZ is lossless.

Testing for Lossless Join Property (Algorithm)

1. Create matrix S with rows for relations R_i and columns for attributes A_j.
2. Initialize entries $S(i,j)$ with distinct symbols.
3. For each R_i containing A_j, set $S(i,j) = a_j$.
4. Repeat until no change:
   - For each FD $X \to Y$, enforce same symbols in X columns imply same in Y.
5. If any row becomes all a_j, decomposition has nonadditive join property.

Dependency Preserving Decomposition

- Intuitive: If R is decomposed into X, Y, Z, and enforcing FDs on each implies all FDs hold on R.
- **Definition:** Projection of F on X (F_X) is the set of FDs $U \to V \in F^+$ with $U, V \subseteq X$.
- $R \to (X, Y)$ is dependency-preserving if

$$(F_X \cup F_Y)^+ = F^+$$

Testing for Dependency Preservation

1. Compute F^+.
2. For each schema R_i in D, let F_i = projection of F^+ onto R_i.
3. Let $F' := \bigcup_i F_i$.
4. Compute F'^+.
5. If $F'^+ = F^+$ then decomposition is dependency preserving, else not.

Alternative One

- If each FD in F can be checked on one relation in the decomposition, then decomposition is dependency preserving.

Alternative Two

- Even if some FD cannot be tested on a single relation, decomposition might still be dependency preserving.
- Provides an easy sufficient condition to check, but not necessary.

Decomposition into BCNF

- Given relation R with FDs F:
  1. If R is not in BCNF, let $X \to Y$ violate BCNF.
  2. Decompose R into XY and $R - (Y - X)$.
  3. Repeat until all schemas are in BCNF.
- Ensures lossless decomposition.

BCNF & Dependency Preservation

- Decomposition into BCNF is not guaranteed to be dependency preserving.
- May lose ability to enforce some FDs without joins.

Decomposition into 3NF

- Guarantees both:
  - Lossless join decomposition.
  - Dependency preservation.
- Based on a minimal cover of F.

Summary of Schema Refinement

- Lossless join decomposition: must have it.
- Dependency preservation: desirable for enforcement of constraints.
- BCNF: eliminates redundancy but may not preserve dependencies.
- 3NF: weaker but ensures both lossless join and dependency preservation.

How Good is BCNF?

- Removes all redundancy due to FDs.
- But may fail to preserve dependencies.

Higher Normal Forms

- Beyond 3NF and BCNF:
  - 4NF (eliminates redundancy due to MVDs).
  - 5NF (eliminates redundancy due to join dependencies).

Lecture 8

Why Semi-Structured Data

- Schemas evolve; rigid tables slow iteration.
- Natural fit for nested or variable-shape data (e.g., user interests, logs).
- Web/mobile services exchange complex payloads.
- Easier interoperability for data exchange across systems.

Models at a Glance

- Flexible schema; sparse/wide columns possible.
- Multivalued types (sets, arrays, maps).
- Example set: {basketball, cooking, anime, jazz}.
- Example map: {(brand, Apple), (id, "MacBook Air"), (size, 13), (color, "silver")}.

Nested Data Types

- Many apps need hierarchical values (not just single numbers/strings).
- Composite (has sub-attributes): name = {first, last}.
- Multivalued (list/set): hobbies = ["piano", "basketball"].
- Why they matter:
  - Model real-world objects naturally; fewer awkward join tables.
  - Schema-flexible: fields can be added/omitted without changing a rigid schema.
- Common representations:
  - JSON — today's default for web/mobile APIs.
  - XML — mature standard with rich tooling and schemas (DTD/XSD).
- Supports numbers, strings, objects (maps), arrays.

JSON in SQL

- What databases add:
  - Native JSON types (e.g., PostgreSQL json/jsonb, MySQL/SQL Server/SQLite JSON).
  - Path extraction operators/functions.
  - Constructors and aggregates to build JSON from rows.
  - **Caveat:** syntax differs across databases.
- Size note: JSON is verbose; engines may use binary or compact forms (compression).
- Examples: PostgreSQL jsonb, MongoDB BSON.

Knowledge Representation

- RDF – *Resource Description Framework.*
- RDF triple: (subject, predicate, object)
  - Examples: (NBA-2019, winner, Raptors), (Washington-DC, capital-of, USA).
  - ER-like but schema-flexible; natural graph representation.
- Two forms in practice:
  - Attribute facts: (ID, attribute, value)
  - Relationship facts: (ID1, relation, ID2)
- ID = identifiers of entities.

Triple View of RDF Data

- RDF triple: (subject, predicate, object)
- Two forms:
  - Attribute facts: (ID, attribute, value)
  - Relationship facts: (ID1, relation, ID2)

Graph View of RDF Data

- Knowledge graph for part of the University database.
- **Objects**: ovals.
- **Attribute values**: rectangles.
- **Relationships**: edges with associated labels identifying the relationship.
- Note: instance-of relationships omitted for brevity.

Querying RDF – SPARQL

- Goal: find the course id.
- Example triple pattern:
  - ?cid :title "Intro. to Computer Science"
  - A triple pattern is like an RDF triple (subject, predicate, object).
  - ?cid is a variable that can match any value.
  - Binds ?cid to the subject of any triple with predicate :title and that string.
- Joins:
  - ?cid :title "Intro. to Computer Science"
  - ?sid :sec_course ?cid
  - Shared variable ?cid enforces the join between patterns.
- Complete SPARQL query:

```
SELECT ?name WHERE {
  ?cid :title "Intro. to Computer Science" .
  ?sid :sec_course ?cid .
  ?id  :takes ?sid .
  ?id  :name  ?name .
}
```

XML – Extensible Markup Language

- Markup = annotations about a document's structure/meaning that are not printed as part of the content.
  - e.g., a note "make this a large, bold headline" shouldn't appear in the newspaper.
- A **markup language** formally specifies:
  - what is *content*, what is *markup*, and what the markup *means* (semantics).
  - e.g., XML.
- Evolution of markup:
  - From *how to print* ("large, bold") ⇒ to *what it is* (*headline, byline, figure*), i.e., function of the content.

XML – Introduction

- **What it is:** XML = *eXtensible Markup Language*, a W3C standard.
- **Basic structure:**
  - **Elements** use start/end tags and are properly nested: <tag> ... </tag>.
  - **Attributes** carry metadata on the start tag: <course id="CS-101"credits="4≫...</course>.
- **Extensibility vs HTML:**
  - Users can add **new tags** and separately specify how the tag should be handled for display.
  - HTML uses a **predefined vocabulary** of tags for web documents.

XML – Motivation

- **Self-describing:** Tags carry meaning, allowing readers/software to infer structure without a separate schema.
- **Evolvable:** Unknown tags/attributes can be ignored; new optional fields can be added safely (back/forward compatible).
- **Natural for repetition:** Multivalued data = repeated elements (e.g., multiple <item> tags in one order).
- **Nested structures:** Complex objects (order → items → price/qty) map directly to a tree.
- **Validatable when needed:** DTD/XSD can enforce types/keys; otherwise, schema is optional.
- **Trade-off:** Verbose text; mitigate with compression or binary forms when size matters.

Structure of XML

- Tag: label inside < > for a section of data.
- Element: `<tag> ... </tag>` (properly nested).
- Single root element per document.
- Mixed content is legal but discouraged for data representation.

Nested XML Representation of University Information

- **Why nest?**
  - Locality for reads – one fetch gives you instructor + their courses ⇒ fewer joins client-side.
- **Pitfall – Redundancy:**
  - If a course is taught by multiple instructors, full nesting **duplicates** course data ⇒ update anomalies and larger documents.

Attributes

- Elements can have **attributes**.
- **Syntax:** name="value"in the start tag.
  - Values **must be quoted** (single or double).
- **Uniqueness:** An element can have many attributes, but each name appears at most once.
- **Type:** Attribute values are text.

Attributes vs. Subelements

- In document construction:
  - Attributes are part of the markup.
  - Subelement contents are part of basic document contents.
- In data representation, the difference is less clear.
- Use **attributes** for identifiers/metadata: `<course course_id="CS-101"credits="4≫...</course>`.
- Use **subelements** for content:

```
<course>
  <course_id>CS-101</course_id>
  <credits>4</credits>
</course>
```

Elements Containing No Subelements

- An element of the form `<element></element>` that contains no subelements or text can be abbreviated as `<element/>`.
- Abbreviated elements may still contain attributes.

XML Namespaces – Why

- **Problem:** Partners may use the same tag name with different meanings.
- **Idea:** Qualify names with a **prefix**.
  - Bound to a URI ⇒ a globally unique name.
- **Binding syntax:** `xmlns:prefix="URI"` (on an element).
- **Default namespace:** `xmlns="URI"`.
- **Identity rules:** Prefix text is arbitrary; the **URI** is what identifies the vocabulary.

XML Namespaces – How

```
<university
    xmlns="http://example.org/uni"              <!-- default ns for our vocab -->
    xmlns:yale="http://www.yale.edu/ns">        <!-- partner vocab -->
  <course>
    <title>Intro to Computer Science</title>    <!-- in default ns -->
    <code dept="CS" number="101"/>              <!-- attributes are UNqualified -->
    <yale:course>                               <!-- element in yale ns -->
      <yale:title>Intro to CS</yale:title>
      <yale:code yale:dept="CS" yale:number="101"/> <!-- attr needs prefix -->
    </yale:course>
  </course>
</university>
```

XML – CDATA

- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA:

```
<![CDATA[<course> ... </course>]]>
```

- Here, `<course>` and `</course>` are treated as just strings.
- **Syntax**
  - Starts with `<![CDATA[`
  - Ends with `]]>`

XML Schemas

- **Why a schema?**
  - Not required, but crucial for exchange & validation.
  - Defines vocabulary, structure, cardinalities, and (with XSD) data types and keys.
- Two mechanisms for specifying XML schema:
  - Document Type Definition (DTD) – widely used.
  - XML Schema (XSD) – newer, increasing use.

Document Type Definition (DTD)

- **What DTD does:** constrains structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur, and how many times
- **What DTD does not do:**
  - Does not constrain data types
  - All values represented as strings in XML
- **Basic syntax:**

```
<!ELEMENT element (subelements-specification)>
<!ATTLIST element (attributes)>
```

Element Specification in DTD

```
<!ELEMENT element (subelements-specification)>
```

Subelements can be specified as:

- Names of elements
- #PCDATA (parsed character data)
- EMPTY (no subelements) or ANY (anything)

Example:

```
<!ELEMENT department (dept_name, building, budget)>
<!ELEMENT dept_name (#PCDATA)>
<!ELEMENT budget (#PCDATA)>
<!ELEMENT br EMPTY> --> instance <br/>


<!ELEMENT element (subelements-specification)>
```

- Subelement specification may include regular expressions:

```
<!ELEMENT university ((department | course | instructor | teaches)+)>
```

- Special symbols:
  - "," – ordered **sequence** (order matters)
  - "—" – alternatives (i.e., "or")
  - "+" – one or more occurrences
  - "*" – zero or more occurrences
  - "?" – zero or one occurrence

Attribute Specification in DTD

```
<!ATTLIST element attr TYPE DEFAULT>
```

- For each attribute, declare:
  - **Name**
  - **Types:**
    - CDATA – character data
    - ID, IDREF, or IDREFS
  - **Defaults:**
    - #REQUIRED – mandatory
    - #IMPLIED – optional
    - "value"– default
- Example:

```
<!ATTLIST course course_id CDATA #REQUIRED>
```

DTD – ID

- Each element may have **at most one** attribute of type ID.
- Provides a **unique identifier** for the element.
- Rules:
  - ID values must be unique in the same XML document.
  - Attribute name can vary (id, iid, cid, etc.) – the **type** makes it an ID.

```
<!ELEMENT instructor EMPTY>
<!ATTLIST instructor iid ID #REQUIRED>

<instructor iid="i1"/>
<instructor iid="i2"/>
```

DTD – IDREF / IDREFS

- Attributes of type IDREF/IDREFS reference elements (their IDs).
- An IDREF must contain an existing ID value.
- IDREFS may contain a list of IDs, separated by spaces.

```
<!ELEMENT course EMPTY>
<!ATTLIST course cid ID #REQUIRED instructors IDREFS #IMPLIED>

<!ELEMENT instructor EMPTY>
<!ATTLIST instructor iid ID #REQUIRED>

<instructor iid="i1"/>
<instructor iid="i2"/>
<course cid="CS-101" instructors="i1 i2"/>
```

Limitations of DTDs

- Cannot specify type of data (everything is text).
- Cannot constrain numeric ranges (e.g., salary between 0–100000).
- Cannot define cross-element constraints beyond ID/IDREF.
- Cannot define order-insensitive content.
- XML Schema (XSD) addresses these issues.

DTDs and Order of Subelements

- In DTDs, order of subelements is fixed.
- Example:

```
<!ELEMENT instructor (name, dept_name, salary)>
```

- The instance:

```
<instructor>
  <name>Kim</name>
  <dept_name>CS</dept_name>
  <salary>80000</salary>
</instructor>
```

- The following would be invalid (wrong order):

```
<instructor>
  <dept_name>CS</dept_name>
  <name>Kim</name>
  <salary>80000</salary>
</instructor>
```

DTDs – Order-insensitive representation

- To make order unimportant:

```
<!ELEMENT instructor (name | dept_name | salary)*>
```

- This allows any order but does not enforce one occurrence each.
- There is no way in DTDs to say "one of each, any order".
- XML Schema (XSD) introduces unordered groups (via <xs:all>).

Recursion in DTDs

- DTDs allow recursive element definitions.
- Example:

```
<!ELEMENT prerequisite (course*, prerequisite*)>
```

- Allows representation of hierarchies (e.g., prerequisite chains).
- Must be used carefully to avoid cycles in instance data.

XML Schema (XSD)

- XML Schema (W3C) is more expressive than DTD.
- Schema itself is an XML document.
- Provides:
  - Richer data typing system.
  - Support for constraints (keys, value ranges).
  - Namespace-aware definitions.
  - Better compositionality and reuse.

XML Schema – Element Declarations

- <xs:element> defines each element's name and type.
- Simple example:

```
<xs:element name="dept_name" type="xs:string"/>
```

- For nested elements, use complex types:

```
<xs:element name="course">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="course_id" type="xs:string"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="credits" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

XML Schema – Occurrence Constraints

- Attributes minOccurs and maxOccurs define cardinality.
- Examples:

```
<xs:element name="phone" type="xs:string" minOccurs="0" maxOccurs="3"/>
<xs:element name="course" type="CourseType" maxOccurs="unbounded"/>
```

- Default: both = 1 (exactly once).

XML Schema – Simple and Complex Types

- **Simple types:** atomic values (e.g., string, integer, boolean, date).
- **Complex types:** contain nested elements and/or attributes.
- You can define new types using:
  - Restriction – narrows existing type domain.
  - Extension – adds new elements or attributes.

XML Schema – Restriction Example

```
<xs:simpleType name="PositiveInt">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
  </xs:restriction>
</xs:simpleType>
```

XML Schema – Keys and References

- XSD supports relational-style constraints:
  - <xs:key> – defines a unique identifier within a scope.
  - <xs:keyref> – defines a foreign key referencing a key.
- Example:

```
<xs:key name="courseKey">
  <xs:selector xpath="course"/>
  <xs:field xpath="course_id"/>
</xs:key>

<xs:keyref name="takesCourseRef" refer="courseKey">
  <xs:selector xpath="takes"/>
  <xs:field xpath="course_id"/>
</xs:keyref>
```

XML Schema (XSD) – Namespace

- XSD itself uses a namespace declaration:

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

- All schema elements (e.g., xs:element, xs:complexType) come from this namespace.
- Other vocabularies can be mixed by declaring multiple namespaces.
- Example:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:uni="http://example.org/university">
  <xs:element name="instructor" type="uni:InstructorType"/>
</xs:schema>
```

Defining Attributes in XML Schema

- Attributes are specified using the xs:attribute tag.
- **Where:** inside an element's xs:complexType.
- **Presence (use=):** default is optional.
  - Options: optional — required — prohibited.
- **Cardinality:** an attribute may appear at most once on an element; order does not matter.

```
<xs:element name="Order">
  <xs:complexType>
    <xs:attribute name="OrderID" type="xs:int"/>
    <!-- use="optional" would be equivalent -->
  </xs:complexType>
</xs:element>
```

```
<Order/>
<Order OrderID="6"/>
```

```
<xs:complexType name="CourseType">
  <xs:sequence>
    <xs:element name="title" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID" use="required"/>
  <xs:attribute name="dept" type="xs:string" use="optional"/>
</xs:complexType>
```

```
<xs:complexType name="CourseNoDept">
  <xs:complexContent>
    <xs:restriction base="tns:CourseType">
      <xs:sequence>
        <xs:element name="title" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:ID" use="required"/>
      <xs:attribute name="dept" use="prohibited"/> <!-- explicitly forbidden -->
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

Keys and Key References in XML Schema

- Key constraint: dept_name.

```
<xs:key name="deptKey">
  <xs:selector xpath="/university/department"/>
  <xs:field xpath="dept_name"/>
</xs:key>
```

- The selector is a path expression that defines the scope for the constraint.
- The field declarations specify the elements or attributes that form the key.
- Foreign key constraint from course to department.

```
<xs:keyref name="courseDeptFKey" refer="deptKey">
  <xs:selector xpath="/university/course"/>
  <xs:field xpath="dept_name"/>
</xs:keyref>
```

- The refer attribute specifies the name of the key declaration that is being referenced.

Benefits of XML Schema (XSD) over DTDs

- **Rich types** – allows text content to be constrained to specific types (numeric, date, sequences, etc.).
  - xs:int / xs:decimal / xs:date / ...
  - xs:complexType with xs:sequence / xs:choice / xs:all, and minOccurs / maxOccurs.
  - Allows user-defined types to be created.
- **Keys & refs** – supports uniqueness and foreign-key constraints.
  - xs:key, xs:keyref.
- **Namespace-aware** – integrates with namespaces to allow different parts of a document to conform to different schemas.

Lecture 9

Querying and Transforming XML Data

- Two main tasks:
  - **Query:** select/filter/aggregate data from XML
  - **Transform:** translate information from one XML schema to another
- Standard XML querying/translation languages:
  - **XPath** – simple path-based language (building block for XQuery)
  - **XQuery** – standard query language with SQL-like structure (FLWOR)

Tree Model of XML Data

- XML documents are modeled as trees with nodes (elements, attributes).
- Element nodes have **child nodes** (attributes or subelements).
- Text content is modeled as text-node children.
- Each node (except the root) has a **single parent**.
- Node children are **ordered** as in the XML document.

```
<course id="CS-101">
  <title>Intro to CS</title>
</course>
```

XPath – Select Parts of an XML Tree

- XPath selects nodes in XML using path expressions.
- Path expression – sequence of location steps separated by '/'.
- Common syntax:
  - '//' – descendant-or-self (skip levels)
  - '@attr' – select attributes named `attr`
  - '*' – wildcard for any element
- Result: set of nodes (and their containing elements/attributes) matching the path.

XPath – Basics

- Path expressions are evaluated left to right.
- '/' denotes the root of the document.

```
/university-3                    -> <university-3>
/university-3/instructor         -> all <instructor> nodes
/university-3/course/@course_id
```

XPath – Predicates

- **Predicates [ ]** filter node sets.
- **Boolean test:** '[credits]' → elements with a child '<credits>'
- **Attribute test:** '[@course_id]' → elements with that attribute

```
/university-3/course[credits]
/university-3/course[@course_id]
/university-3/course[credits >= 4]
/university-3/course[credits >= 4]/@course_id
```

XPath – Functions

- **Aggregate:** 'count()' counts matching elements
- Example: '/university-2/instructor[count(./teaches/course)>2]'
- **Position test:** '[position()<3]'
- **Boolean connectives:** 'and', 'or'
- **Negation:** 'not()'

XPath – ID Function

- 'id()' selects elements by unique ID value.
- Example: 'id("foo")' returns the node with ID attribute "foo".
- Can apply to references (IDREFS).

```
/university-3/course/id(@dept_name)
/university-3/course/id(@instructors)
```

More XPath Features

- '—' (pipe) – union of expression results Example: '/university-3/course[@dept_name="Comp. Sci"] — /university-3/course[@dept_name="Biology"]'
- '//' skips multiple levels of nodes. Example: '/university-3//name'
- Steps can navigate parents/siblings/ancestors:
  - '//' → all descendants
  - '..' → parent
- 'doc(name)' returns root of named document. Example: 'doc("university.xml")/university/department'

XQuery

- W3C's standard XML query language.
- XPath is a subset used inside XQuery.
- Modeled after SQL with **FLWOR** structure:

FOR · LET · WHERE · ORDER BY · RETURN

- **SQL analogy:**
  - for ⇔ SQL from
  - where ⇔ SQL where
  - order by ⇔ SQL order by
  - return ⇔ SQL select
  - let ⇔ temporary variables (no SQL equivalent)

XQuery – FLWOR Syntax

- **for** – binds a variable to each item of a sequence (XPath result)
- **let** – binds a temporary value
- **where** – filters
- **order by** – sorts
- **return** – constructs the result

Example – Find all courses with credits ¿ 3

```
for $x in /university-3/course
let $courseId := $x/@course_id
where $x/credits > 3
return <course_id>{ $courseId }</course_id>
```

Filter in the for, no let

```
for $x in /university-3/course[credits > 3]
return <course_id>{ $x/@course_id }</course_id>
```

Curly Brackets in Return Clause

- Curly braces {} evaluate expressions inside XML results.
- Items in `return` are treated as XML text unless enclosed in braces.

Alternative Element Construction

- Alternative notation uses **element** and **attribute** constructors.

```
return element course {
    attribute course_id { $x/@course_id },
    attribute dept_name { $x/@dept_name },
    element title { $x/title },
    element credits { $x/credits }
}
```

XQuery – Joins

- Joins are specified in a manner very similar to SQL.

```
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches
where $c/course_id = $t/course_id and $t/IID = $i/IID
return <course_instructor>{ $c $i }</course_instructor>
```

- The same query can be expressed with the selections specified as XPath filters:

```
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches[$c/course_id = $t/course_id
        and $t/IID = $i/IID]
return <course_instructor>{ $c $i }</course_instructor>
```

Comparisons on Sequences

- **General comparisons:** = != < > <= >=

```
$c/credits > 3         (: true if any <credits> > 3 :)
$x/credits = $y/credits   (: true if any credit in $x equals any in $y :)
```

- **Value comparisons:** eq ne lt le gt ge
- These raise an error if either input is a sequence with multiple values.

```
$c/@course_id eq $i/@IID    (: enforces 1-to-1, no multi-values :)
```

Nested Queries

- XQuery FLWOR expressions can be nested inside `return` clauses to create new element hierarchies.

```
<university-1>{
  for $d in /university/department
  return
    <department>
      { $d/* }
      { for $c in /university/course[dept_name = $d/dept_name]
        return $c }
    </department>,
  for $i in /university/instructor
  return
    <instructor>
      { $i/* }
      { for $c in /university/teaches[IID = $i/IID]
        return $c/course_id }
    </instructor>
}</university-1>
```

Aggregate Functions

- XQuery provides a variety of aggregate functions:
  - `fn:count()`, `fn:sum()`, `fn:min()`, `fn:max()`, `fn:distinct-values()`
- Apply on sequences of elements or values.
- Aggregation functions can appear in any XPath expression.
- Namespace: `http://www.w3.org/2005/xpath-functions`, prefix `fn`.

```
fn:count(/university/course)                    (: how many courses :)
fn:distinct-values(/university/course/dept_name)  (: unique departments :)
```

Grouping and Aggregation

- Nested queries are used for grouping.

```
for $d in /university/department
return
  <department-total-salary>
    <dept_name>{ $d/dept_name }</dept_name>
    <total_salary>{
      fn:sum(
        for $i in /university/instructor[dept_name = $d/dept_name]
        return $i/salary
      )
    }</total_salary>
  </department-total-salary>
```

Sorting in XQuery

- Results can be sorted in XQuery using the `order by` clause.
- **Default:** ascending order.

```
for $i in /university/instructor
order by data($i/name)
return <instructor>{ $i/* }</instructor>
```

- Use `order by $i/name descending` to sort in descending order.

```
<university-1>{
  for $d in /university/department
  order by $d/dept_name
  return
    <department>
      { $d/* }
      { for $c in /university/course[dept_name = $d/dept_name]
        order by $c/course_id
        return <course>{ $c/* }</course> }
    </department>
}</university-1>
```

Functions and Types

- User-defined functions use XML Schema types.

```
declare function local:dept_courses($iid as xs:string)
  as element(course)* {
    for $i in /university/instructor[IID = $iid],
        $c in /university/course[dept_name = $i/dept_name]
    return $c
};
```

- Namespace xs: – predefined for XML Schema datatypes.
- Namespace local: – predefined for user-defined functions.

Function Invocation

```
for $i in /university/instructor[name = "Sophie"]
return local:dept_courses($i/IID)
```

- Returns the department courses for instructor(s) named "Sophie".

Other XQuery Features

- XQuery supports **if-then-else** constructs within return clauses.

```
<products>
  <product>
    <name>Widget</name>
    <price>50</price>
  </product>
  <product>
    <name>Gadget</name>
    <price>120</price>
  </product>
</products>

for $p in /products/product[name="Gadget"]
return
  if ($p/price > 100) then
    <result>The product is expensive</result>
  else
    <result>The product is affordable</result>
```

Other XQuery Features (Quantifiers)

- Universal & existential quantification in **where** predicates:
  - **some** $e in path **satisfies** P
  - **every** $e in path **satisfies** P
- Example — *find departments where every instructor has a salary > $50,000*

```
for $d in /university/department
where every $i in /university/instructor[dept_name=$d/dept_name]
     satisfies $i/salary > 50000
return $d
```

- If a department has no instructor, it trivially satisfies the condition.
- Add: fn:exists(/university/instructor[dept_name=$d/dept_name]) to ensure at least one instructor exists in the department.

Scrapy + XPath for Web Crawling

```
class ToScrapeSpiderXPath(scrapy.Spider):
    name = "toscrape-xpath"
    start_urls = ["http://quotes.toscrape.com/"]

    def parse(self, response):
        for quote in response.xpath('//div[@class="quote"]'):
            yield {
                "text":   quote.xpath('./span[@class="text"]/text()').extract_first(),
                "author": quote.xpath('./span/small[@class="author"]/text()').extract_first(),
                "tags":   quote.xpath('./div[@class="tags"]/a[@class="tag"]/text()').extract()
            }
        next_page_url = response.xpath('//li[@class="next"]/a/@href').extract_first()
        if next_page_url is not None:
            yield scrapy.Request(response.urljoin(next_page_url))
```

Storage of XML Data

- **Non-relational** data stores
  - **Flat files**: natural file format for XML; simple
    - Problems: no concurrency, no recovery, etc.
  - **Native XML database**:
    - Built specifically for XML (DOM model, declarative querying)
    - Currently no commercial-grade systems
- **Relational databases** with XML support
  - Data must be translated into relational form
  - **Pros**: mature RDBMS features (ACID, HA, security); integrate with SQL
  - **Cons**: overhead of translation (nested/recurring elements)

Storage of XML in Relational Databases — Alternatives

- String Representation
- Tree Representation
- Map to Relations

String Representation (How)

- Store *small* XML docs as text (CLOB) in an RDBMS
  - CLOB is a SQL:1999 standard type
- For large XML: store each top-level element (e.g., a direct child of <university>) in a string/CLOB column
- Two layouts
  - Single relation for all elements
  - One table per kind (e.g., department_elements, course_elements, instructor_elements, teaches_elements)
- Extract "hot" fields (from subelements/attributes) into separate columns for indexing (e.g., dept_name, course_id, IID)

String Representation (Pros & Cons)

- **Benefits**
  - Can store any XML data even without DTD
  - When many top-level elements exist, strings are small compared to full document
  - Allows fast access to individual elements
- **Drawback**
  - Must parse strings to access values inside elements — parsing can be slow

Tree Representation (Model)

- Model XML as a tree and store in a relation:

```
nodes(id, parent_id, type, label, value)
```

- Each element/attribute gets a unique identifier
- type indicates element vs attribute
- label is the tag/attribute name
- value is the text value
- Optional extra attribute position to record child order

Tree Representation (Pros & Cons)

- **Benefit**: can store any XML data, even without DTD
- **Drawbacks**
  - Data split into many pieces ⇒ increased space overhead
  - Even simple queries may require many joins (slow)

Mapping XML Data to Relations

- Create one table per element type with known schema
- **PK**: synthetic id (or natural key if provided)
- **FK** to parent id for nested elements (preserve hierarchy)
- Attributes & single-occurrence subelements → columns
- Repeating subelements → separate child table
- Add position only if child order matters

Publishing and Shredding XML Data

- Used when exchanging data between business apps
- **Publishing**: convert relational data to XML for export
- **Shredding**: convert XML to normalized relations for storage in RDBMS
- XML-enabled DB systems support automated publishing/shredding
- **Publishing idea**: map each row to an XML element; columns become subelements/attributes
- **Shredding idea**: inverse mapping (or use the mapping rules above)

Native Storage within a Relational Database

- Many systems offer native storage of XML data using the new xml data type.
- XML query languages such as XPath and XQuery are supported to query XML data.
- A relation with an attribute of type xml can store a collection of XML documents; each document is stored as a value of type xml in a separate tuple.
- Allows XQuery queries to be embedded within SQL queries.
- XQuery can be executed on single or multiple XML documents within SQL, each document stored in a separate tuple.

SQL/XML Standard

- Defines SQL extensions to create nested XML output (publishing).
- Each output tuple is mapped to an XML element (row).

SQL Extensions

- Adds operators and aggregates for XML construction directly in SQL.
  - xmlelement — creates XML elements.
  - +xmlattributes — creates attributes.
- Example: Create XML for each course with course id and department as attributes, and title and credits as subelements.

```
select xmlelement(name "course",
       xmlattributes(course_id as course_id, dept_name as dept_name),
       xmlelement(name "title", title),
       xmlelement(name "credits", credits))
from course;
```

SQL Extensions (xmlforest)

- xmlforest — creates a collection (forest) of subelements.

```
SELECT XMLELEMENT("employee",
       XMLFOREST(
          e.empno AS "works_number",
          e.ename AS "name",
          e.job AS "job"))
AS employee
FROM emp e
WHERE e.empno = 7782;
```

Output:

```
<employee>
  <works_number>7782</works_number>
  <name>CLARK</name>
  <job>MANAGER</job>
</employee>
```

SQL Extensions 2 (xmlagg)

- xmlagg — aggregates XML elements into a collection.

```
select xmlelement(name "department",
       dept_name,
       xmlagg(xmlforest(course_id)
       order by (course_id)))
from course
group by dept_name;
```

- Creates one XML element per department, containing all its courses as subelements.
- Since grouped by department, aggregate applies across all courses per department, producing a sequence of course_id elements.

XML Applications

- Storing and exchanging data with complex structures.
  - For structured but non-relational data (e.g., user preferences with many fields or multivalued items).
  - For office data — documents, spreadsheets, etc.
    - Open Document Format (ODF): for OpenOffice.
    - Office Open XML (OOXML): for Microsoft Office.
- Standardized data exchange formats:
  - **ChemML**: for representing chemical information (molecular structure, boiling points, etc.).
  - **RosettaNet**: for e-business XML schemas and message exchange.
- Data mediation:
  - Provides a common data representation format to bridge heterogeneous systems.

XML vs. Relational Data

- **Why XML can be inefficient:**
  - **Verbose/overhead:** repetitive tags increase space vs. rows/columns.
  - **Parsing cost:** converting text to typed values; validation optional.
- **Why XML is better for data exchange:**
  - **Self-describing:** data defined by tags.
  - **Schema-flexible:** easy to evolve (add/remove elements or attributes).
  - **Nested & ordered structures:** represent hierarchies and mixed content.
  - **Ecosystem:** widely accepted across databases, browsers, tools, and applications.

Lecture 10

Taxonomy of NoSQL Databases

- **Key-Value Stores** — e.g., Redis, DynamoDB Fast lookups using key–value pairs.
- **Document Stores** — e.g., MongoDB, CouchDB Flexible schema; stores JSON/BSON documents.
- **Graph Databases** — e.g., Neo4j, Dgraph Represent relationships between entities.
- **Vector Databases** — e.g., Pinecone, Chroma Used for AI similarity search (vector embeddings).

What is MongoDB?

- MongoDB = "**Humongous DB**"
- Document-oriented, stores JSON/BSON data.
- Open-source, schema-flexible.
- High performance and high availability.
- Automatic scaling — **Horizontal Scalability**.
- Classified as a **CP system** under CAP theorem.

Vertical vs. Horizontal Scaling

- **Vertical Scaling (Scale Up):** Add more CPU, RAM, or storage to one server.
- **Horizontal Scaling (Scale Out):** Add more servers/nodes to distribute load.

CAP Theorem

- Proposed by Eric Brewer (2000, Berkeley).
- Proved by Gilbert and Lynch (2002, NUS and MIT).
- In distributed systems, at most two of the following three can be guaranteed:
  1. **Consistency (C):** All nodes see the same data.
  2. **Availability (A):** System always responds to requests.
  3. **Partition Tolerance (P):** System operates despite network failures.
- MongoDB = **CP system** (Consistency + Partition Tolerance).

Why Availability is Important

- Ensures reliable, fast reads/writes — directly linked to business revenue.
- **Amazon (2020):** +100 ms latency = 1% sales; each extra millisecond $6M yearly loss.
- **Google (2020):** +0.5 s latency in search = 20% traffic.
- SLAs mainly concern latency; Netflix relies on AWS for availability.

Why Consistency is Important

- All nodes must see the same data or the latest update.
- Crucial for **accuracy and integrity**.
- Examples:
  - Banking & investment — instant balance updates.
  - Flight booking — all users must see up-to-date seat info.

Why Partition Tolerance is Important

- The system must continue functioning despite network issues.
- Real-world causes: Internet router outages; Undersea cable cuts; DNS or datacenter failures.
- Systems must remain operational even if some nodes are unreachable.

CAP Theorem Fallout

- **Partition Tolerance (P)** is non-negotiable in cloud systems.
- Thus, systems must choose between:
  - **Consistency (C)** or **Availability (A)**.
- **Traditional RDBMS:** Choose Consistency + Partition Tolerance; sacrifice Availability.
- **DynamoDB / NoSQL:** Choose Availability + Partition Tolerance; default = **Eventual Consistency**.

Eventual Consistency

- **Definition:** All replicas eventually converge if no new updates occur.
- **Behavior:**
  - Continuous synchronization across nodes.
  - May return temporary stale reads.
  - Eventually all replicas catch up with the latest state.
- **When Effective:**
  - Systems with short write bursts.
  - Apps tolerant to slightly outdated reads.

CAP Tradeoff

- Starting point for the NoSQL revolution.
- In a distributed system, you can only guarantee two of:
  - **Consistency (C)**
  - **Availability (A)**
  - **Partition Tolerance (P)**
- With Partition Tolerance required in real-world networks, the choice is between:
  - **Consistency + Partition Tolerance (CP)**
  - **Availability + Partition Tolerance (AP)**

CAP Tradeoff: How to Choose

- **Use Case:**
  - Real-time apps (e.g., recommendation engines)
  - Analytics or financial applications
- **Data Requirements:**
  - If accuracy and integrity are critical → prefer CP.
  - If system can tolerate temporary stale reads → prefer AP.
- **Network Conditions:**
  - In unstable networks, Partition Tolerance (P) is unavoidable.
  - Trade-off is always **C vs A** under partitions.

Tunable Consistency & Scalability

- **Tunable Consistency:**
  - Some databases (e.g., DynamoDB) allow adjustment between C and A per operation or dataset.
- **Scalability Impact:**
  - Availability-oriented systems are easier to scale out horizontally.
  - May require conflict resolution mechanisms (e.g., last-write-wins, vector clocks).

Data Model of MongoDB

- **Document:** Basic unit of data.
  - Stored in **BSON (Binary JSON)** → field–value pairs.
  - Maximum document size: **16 MB**.
  - Supports nested fields and arrays.
- **Collection:** Group of documents.
  - Similar to a relational **table**, but **schema-less**.
  - Documents may have different structures.
  - Share common indexes.
- **Database:** Container for collections.

JSON (JavaScript Object Notation)

- **Semi-structured format:**
  - Flexible schema — good for data transfer/exchange.
  - Supports **nesting** (objects inside objects, arrays).
- **Built on:**
  - Field–value pairs.
  - Ordered lists (array order preserved).
- **Advantages:**
  - Easy for humans to read/write.
  - Easy for computers to parse/generate.

BSON (Binary JSON)

- Binary-encoded representation of JSON documents.
- Supports **more data types** than JSON (e.g., Date, Decimal128).
- **Goals:**
  - Lightweight (compact binary format)
  - Traversable (easy field access)
  - Efficient (fast encoding/decoding)
- Optimized for **performance and storage**.

Key Differences: JSON vs BSON

- **Format:** JSON = Text (UTF-8); BSON = Binary (efficient storage)
- **Date Handling:** JSON = String; BSON = Native Date type
- **Binary Data:** JSON = Base64 encoded; BSON = Native support
- **Integer Types:** JSON = Single numeric; BSON = 32/64-bit types
- **Custom Types:** BSON adds ObjectId, Timestamp, Decimal128, etc.
- **Efficiency:** BSON more efficient for complex data, though may add overhead (field names + type metadata)

Documents in MongoDB

- MongoDB stores data as BSON documents — **basic unit of data**.
- Each document is composed of field–value pairs.
- Field values can include:
  - Any BSON data type
  - Other documents
  - Arrays or arrays of documents

Example Document

```
var mydoc = {
  _id: ObjectId("5099803df3f4948bd2f98391"),
  name: { first: "Alan", last: "Turing" },
  birth: new Date("Jun 23, 1912"),
  death: new Date("Jun 07, 1954"),
  contribs: ["Turing machine", "Turing test", "Turinergy"],
  views: NumberLong(1250000)
}
```

- _id → unique ObjectId
- name → embedded document
- birth/death → Date
- contribs → array of strings
- views → 64-bit integer (NumberLong)

BSON Types

- Common types include:
  - Double, String, Object, Array, Binary Data, ObjectId, Boolean, Date, Null, Regex
  - 32-bit and 64-bit integers, Timestamp, Decimal128
- Type numbers/aliases can be queried using the `$type` operator.

The _id Field

- Every MongoDB document contains an **_id** field by default.
- Serves as the **primary key** for the collection.
- Unique, immutable, and may be any non-array type.
- **Default type:** ObjectId (12-byte value).
- Small, fast to generate, ordered.
- Provides approximate ordering by **creation time**.
- If no _id is supplied, MongoDB auto-generates one.

MongoDB vs SQL

- **MongoDB** vs **SQL** analogies:

MongoDB	SQL
Database	Database
Collection	Table
Document	Row/Tuple
Field	Column/Attribute
PK: _id Field	PK: Any Attribute(s)
Uniformity not required	Uniform Relation Schema
Index	Index
Embedded Structure / DBRef	Joins
Shard	Partition

CRUD: Using the Shell

- Display current database: `db`
- Switch databases: `use <database>`
- Show all databases: `show dbs`
- Create new database: `use <new_database>`
- Note: Databases are created only when data is inserted.

CRUD: Creating Collections

- Create a new collection:

```
db.collection.insertOne(<document>)
```

- Equivalent SQL:

```
INSERT INTO <table> VALUES (<attributevalues>);
```

- MongoDB creates databases/collections on first insert.
- Show all collections: `show collections`

CRUD: Inserting Documents

- Insert a single document:

```
db.collection.insertOne(<document>)
```

- Insert multiple documents:

```
db.collection.insertMany([<document1>, <document2>, ...])
```

Example:

```
db.movies.insertOne({
  title: "The Favourite",
  genres: ["Drama", "History"],
  runtime: 121,
  rated: "R",
  year: 2018,
  directors: ["Yorgos Lanthimos"],
  cast: ["Olivia Colman", "Emma Stone", "Rachel Weisz"],
  type: "movie"
})
```

- Returns a document containing:
  - A **boolean** field `acknowledged: true`
  - A field `insertedId` with the new _id value.

CRUD: Querying Documents

- Query syntax:

```
db.collection.find(query, projection, options)
```

- Read all documents in a collection:

```
db.collection.find()
```

- SQL equivalent:

```
SELECT * FROM myCollection;
```

- Read one document:

```
db.collection.findOne()
```

Querying with Conditions

- Match a specific value:

```
db.movies.find({ "title": "Titanic" })
```

- SQL equivalent:

```
SELECT * FROM movies WHERE title = "Titanic";
```

Querying Multiple Values

- Use `$in` operator:

```
db.movies.find({ rated: { $in: ["PG", "PG-13"] } })
```

- SQL equivalent:

```
SELECT * FROM movies WHERE rated IN ("PG", "PG-13");
```

Logical Operators (AND / OR)

- Example: movies released in Mexico with IMDB ≥ 7

```
db.movies.find({ countries: "Mexico", "imdb.rating": { $gte: 7 } })
```

- SQL equivalent:

```
SELECT * FROM movies WHERE countries = "Mexico" AND imdb_rating >= 7;
```

- Return movies released in 2010 that either won at least 5 awards or have the genre "Drama".
- Example:

```
db.movies.find({
  year: 2010,
  $or: [
    { "awards.wins": { $gte: 5 } },
    { genres: "Drama" }
  ]
})
```

- SQL equivalent:

```
SELECT * FROM movies
WHERE year = 2010 AND (awards_wins >= 5 OR genres = "Drama");
```

Querying Arrays

- Return all documents where the field `tags` is an array with exactly two elements: `"red"` and `"blank"` (in that order).

```
db.itemList.find({ tags: ["red", "blank"] })
```

Querying Arrays of Embedded Documents

- Return all documents where an element in the `instock` array matches the specified document.

```
db.inventory.find({ instock: { warehouse: "A", qty: 5 } })
```

- Equivalent query using dot notation:

```
db.inventory.find({ "instock.warehouse": "A", "instock.qty": 5 })
```

Querying Embedded Array Fields by Index

- Use the array index to query a field within an embedded document.

```
db.inventory.find({ "instock.0.qty": { $gte: 20 } })
```

Additional Query Examples

- Combine filters with `$and`, `$or`, and field-level conditions.
- Example:

```
db.inventory.find({
  $and: [
    { "instock.qty": { $gte: 10 } },
    { "instock.warehouse": "B" }
  ]
})
```

- SQL equivalent:

```
SELECT * FROM inventory
WHERE instock_qty >= 10 AND instock_warehouse = "B";
```