WHAT IS SQL

Database query language, which can also:

Define structure of data
Modify data
Specify security constraints
DATA DEFINITION

Data-definition language (DDL) provides commands for:

- Defining relation schemas,
  - Determine the domain for each attribute (char, int, etc.)
  - Determine integrity constraints (not null, primary key etc.)
- Deleting them,
- Modifying them
BASIC RELATION DEFINITION

CREATE TABLE <name_of_relation> (  
    <attribute_name>    <data_type>    <constraint>*,
    <attribute_name>    <data_type>    <constraint>*,
    ...
    <integrity_constraint>,
    <integrity_constraint>,
    ...
);

E.g:

CREATE TABLE department (  
    dept_name varchar (20),
    building varchar (15),
    budget numeric (12,2),
    PRIMARY KEY (dept_name));
DATA TYPES

**char(n):** A fixed-length character string with user-specified length n. The full form, character, can be used instead.

**varchar(n):** A variable-length character string with user-specified maximum length n. The full form, character varying, is equivalent.

**int:** An integer (a finite subset of the integers that is machine dependent). The full form, integer, is equivalent.

**smallint:** A small integer (a machine-dependent subset of the integer type).

**numeric(p, d):** A fixed-point number with user-specified precision. The number consists of p digits (plus a sign), and d of the p digits are to the right of the decimal point. Thus, numeric(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.

**real, double precision:** Floating-point and double-precision floating-point numbers with machine-dependent precision.

**float(n):** A floating-point number, with precision of at least n digits.
INTEGRITY CONSTRAINTS

**PRIMARY KEY**(<attribute_name>, <attribute_name>, …):

Given attributes form primary key of relation
nonnull and unique

E.g: CREATE TABLE teachers (  
    teacher_id int,  
    first_name varchar(50),  
    last_name varchar(50),  
    PRIMARY KEY (teacher_id)  
)  

OR  

E.g: CREATE TABLE teachers (  
    teacher_id int PRIMARY KEY,  
    first_name varchar(50),  
    last_name varchar(50)  
)
INTEGRITY CONSTRAINTS

**FOREIGN KEY(<attribute_name>, <attribute_name>,…) REFERENCES <table_name>):**

Given attributes must be a primary key in the given table.

E.g: CREATE TABLE teachers (  
    teacher_id int,  
    first_name varchar(50),  
    last_name varchar(50),  
    department_id int,  
    PRIMARY KEY (teacher_id),  
    FOREIGN KEY (department_id) REFERENCES departments  
)
INTEGRITY CONSTRAINTS

NOT NULL

E.g: CREATE TABLE teachers (  
    teacher_id int,  
    first_name varchar(50) NOT NULL,  
    last_name varchar(50),  
    department_id int,  
    PRIMARY KEY (teacher_id),  
    FOREIGN KEY (department_id) REFERENCES departments  
)
CREATE TABLE teachers (  
    teacher_id int PRIMARY KEY,  
    first_name varchar(50),  
    last_name varchar(50)  
)  

INSERT INTO teachers  
VALUES( 1, "Stan", "Zdonik" )  

DELETE FROM teachers  
WHERE <condition>  

INSERT INTO teachers(teacher_id, last_name)  
VALUES( 1, "Zdonik" )
CREATE TABLE teachers (  
  teacher_id int PRIMARY KEY,  
  first_name varchar(50),  
  last_name varchar(50)  
)  

ALTER TABLE teachers  
  ADD department_id int;  

DROP TABLE teachers;  

ALTER TABLE teachers  
  DROP department_id;
DATA MANIPULATION

Data-manipulation Language (DML) provides commands for:

- Querying information about data
- Insert/Delete/Update tuples

Commands are usually like:

```
SELECT <attribute_name>*
FROM <relation>*
WHERE <condition>;
```

Result of a SQL query is always a relation!
SELECT

Projection operation \( \prod_{a_1, a_2, \ldots, a_n} (R) \)

E.g:

- SELECT first_name
  FROM teachers

- SELECT first_name, last_name
  FROM teachers

- SELECT *
  FROM teachers

**Relational algebra does not allow duplicates, SQL does!**

- SELECT DISTINCT first_name
  FROM teachers

Can contain arithmetic operations (+, -, *, /) on attributes of tuples or constants

E.g:

- SELECT first_name, weight / height, salary * 12
  FROM teachers
FROM

Indicates the relations involved
Cartesian product: $R_1 \times R_2$

E.g: Find every possible pair of left and right socks

```
SELECT *
FROM left_socks, right_socks
```

Will contain all the attributes of the given relations
WHERE

Selection operation \( \sigma_{\text{condition}}(R) \)

SELECT teacher_id
FROM teachers
WHERE age > 50 AND NOT department_name = "computer science"

The condition can contain AND, OR, and NOT

BETWEEN Clause:

SELECT teacher_id
FROM teachers
WHERE age <= 50 AND age >= 30

=  

SELECT teacher_id
FROM teachers
WHERE age BETWEEN 30 AND 50
NULL VALUES

Any arithmetic operation involving null results in null. (E.g: 2 + null = null)

To check for null values, you can use IS NULL predicate

--E.g: SELECT first_name FROM teachers WHERE last_name IS NULL

Any comparison with null returns unknown (E.g: 2 < null)

<table>
<thead>
<tr>
<th>AND</th>
<th>true</th>
<th>false</th>
<th>unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>unknown</td>
<td>false</td>
<td>unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>true</th>
<th>false</th>
<th>unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>true</td>
<td>unknown</td>
<td>unknown</td>
</tr>
</tbody>
</table>

**NOT unknown = unknown**

“<Condition> IS UNKNOWN” evaluates to true if <Condition> evaluates to unknown

**WHERE clause treats conditions resulting in unknown as false**
<ATTRIBUTE> LIKE <PATTERN>

Used for pattern matching in strings
• % for matching any substring
• _ for matching any character

E.g: Find all student names starting with ‘A’
SELECT name
FROM students
WHERE name LIKE ‘A%’

E.g: Find all student names consisting of 3 letters
SELECT name
FROM students
WHERE name LIKE ‘___’

Patterns are case sensitive!
JOINS

Find the first and last name of the students who work under a department

```
SELECT first_name, last_name
FROM students, departments
WHERE students.DepartmentID = departments.DepartmentID
```

OR

```
SELECT first_name, last_name
FROM students JOIN departments ON students.DepartmentID = departments.DepartmentID
```

OR

```
SELECT first_name, last_name
FROM students NATURAL JOIN departments
```
<RELATION1> NATURAL JOIN <RELATION2>

Relational algebra: <relation1> <relation2>

Merges tuples with the same values for all the attributes that have the same name and type. Keeps only 1 copy of each common attribute

SELECT * FROM students NATURAL JOIN departments

<table>
<thead>
<tr>
<th>ID</th>
<th>FirstName</th>
<th>LastName</th>
<th>DepartmentID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>D</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>CSCI</td>
</tr>
<tr>
<td>20</td>
<td>MUSC</td>
</tr>
</tbody>
</table>
SELECT *
FROM students JOIN departments ON students.DepartmentID = departments.DepartmentID

<table>
<thead>
<tr>
<th>ID</th>
<th>FirstName</th>
<th>LastName</th>
<th>DepartmentID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>D</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>CSCI</td>
</tr>
<tr>
<td>20</td>
<td>MUSC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>FirstName</th>
<th>LastName</th>
<th>DepartmentID</th>
<th>DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>10</td>
<td>CSCI</td>
</tr>
</tbody>
</table>
<\textit{RELATION1}> LEFT JOIN <\textit{RELATION2}> ON <\textit{CONDITION}>

Relational algebra: \( <\textit{relation1}> \ \bowtie <\textit{relation2}> \)

SELECT *
FROM students
LEFT JOIN departments ON students.DepartmentID = departments.DepartmentID

<table>
<thead>
<tr>
<th>ID</th>
<th>FirstName</th>
<th>LastName</th>
<th>DepartmentID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>D</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>CSCI</td>
</tr>
<tr>
<td>20</td>
<td>MUSC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>FirstName</th>
<th>LastName</th>
<th>DepartmentID</th>
<th>DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>10</td>
<td>CSCI</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>D</td>
<td>\textit{null}</td>
<td>\textit{null}</td>
</tr>
</tbody>
</table>
RENAMED

Relational algebra: \( \rho_{<\text{new\_name}>/\text{<old\_name>}} (R) \)

<old_name> AS <new_name>

E.g:

SELECT name, salary/12 AS monthly_salary
FROM employees

SELECT first_name, last_name
FROM students AS S, departments AS D
WHERE S.DepartmentID = D.DepartmentID
ORDER BY <ATTRIBUTE>

E.g: List every student’s name and height by ordering them in height from shortest to tallest

SELECT name, height
FROM students
ORDER BY height

E.g: List every student’s name and height by ordering their names alphabetically. The students with the same name should be ordered by their heights in descending order

SELECT name, height
FROM students
ORDER BY name, height DESC

ORDER BY <attribute> ASC/DESC
Default is ASC
SET OPERATIONS

**UNION** (U operation), **INTERSECT** (\( \cap \) operation), **EXCEPT** (\( - \) operation)

The relations these operations work on must have same attributes with same domain

E.g.

(SELECT course_id FROM section WHERE sem = ‘Fall’ AND year = 2009) **UNION** (SELECT course_id FROM section WHERE sem = ‘Spring’ AND year = 2010)

Above operations automatically eliminates duplicates.
In order to retain the duplicates, use ALL (E.g. **UNION ALL**)
AGGREGATE FUNCTIONS

Provide more information about an attribute

**AVG:** average value

**MIN:** minimum value

**MAX:** maximum value

**SUM:** sum of values

**COUNT:** number of values
AGGREGATE FUNCTIONS

E.g. Find the average salary of teachers in computer science department

```
SELECT AVG(salary)
FROM teachers
WHERE department = "Computer Science"
```

E.g. Find the total number of teachers in computer science department

```
SELECT COUNT(ID)
FROM teachers
WHERE department = "Computer Science"
```

E.g. How many different classes do teachers in computer science department teach?

```
SELECT COUNT(DISTINCT class_name)
FROM teachers
WHERE department = "Computer Science"
```
GROUP BY <ATTRIBUTE>

Enables you to use aggregate functions on certain groups
E.g. Find the average salary of teachers in every department

```
SELECT department, AVG(salary)
FROM teachers
GROUP BY department
```

**Attributes inside of SELECT clause and outside of aggregate function must be included in GROUP BY**

E.g. False query

```
SELECT department, department_ID, AVG(salary)
FROM teachers
GROUP BY department
```
HAVING

WHERE clause of aggregate functions

Difference: Predicates in WHERE clause are applied BEFORE formation of groups, unlike GROUP BY

E.g. Find the departments who are employing more than 20 teachers

```sql
SELECT department, COUNT( DISTINCT teacher_ID )
FROM teachers
GROUP BY department
HAVING COUNT( DISTINCT teacher_ID ) > 20
```
NULLS IN AGGREGATES

All aggregate operations except COUNT(*) ignore null values on aggregated attributes

E.g.

SELECT SUM( salary )
FROM people

Will return 22

If an attribute has only null values:

COUNT will return 0
others will return *null*
NESTED SUBQUERIES

Common usage: checking set membership, set comparisons, set cardinality

E.g. Find teachers who work in both math and cs department

```sql
SELECT DISTINCT teacher_ID
FROM teachers
WHERE department = "Computer Science"
    AND teacher_ID IN (SELECT DISTINCT teacher_ID
                        FROM teachers
                        WHERE department = "Mathematics")
```
NESTED SUBQUERIES

E.g. Find teachers who work in cs department and NOT a student

```
SELECT DISTINCT teacher_ID
FROM teachers
WHERE department = "Computer Science"
    AND teacher_ID NOT IN (SELECT DISTINCT student_ID
                               FROM students)
```
NESTED SUBQUERIES

E.g. Find students who gain more than at least one teacher

SELECT student_name
FROM working_students
WHERE salary > SOME (SELECT salary
                      FROM teachers)

E.g. Find students who gain more than ALL teachers

SELECT student_name
FROM working_students
WHERE salary > ALL (SELECT salary
                      FROM teachers)
NESTED SUBQUERIES

E.g. Find students who gain more than at least one teacher

```
SELECT student_name
FROM working_students AS W
WHERE EXISTS (SELECT salary
              FROM teachers AS T
              WHERE W.salary > T.salary)
```

EXISTS returns true if the given subquery does not result in an empty relation

NOT EXISTS performs in the opposite way
NESTED SUBQUERIES

E.g. Find all students who have taken all courses offered in the Biology department

\[
\text{SELECT DISTINCT } S.ID, S.name \\
\text{FROM student AS } S \\
\text{WHERE NOT EXISTS ( } \text{(SELECT course_id} \\
\quad \text{FROM course} \\
\quad \text{WHERE dept_name = “Biology”)} \\
\quad \text{EXCEPT} \\
\quad \text{(SELECT T.course_id} \\
\quad \text{FROM takes AS } T \\
\quad \text{WHERE } S.ID = T.ID));
\]