The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- Also: other information such as
  - The set of indices to be maintained for each relation.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.
SQL Constructs: Data Definition Language

CREATE TABLE <name> ( <field> <domain>, ... )

create table department
    dept_name varchar(20),
    building varchar(15),
    budget numeric(12,2) check (budget > 0),

    primary key (dept_name)
);

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
)
SQL Constructs: Data Definition Language

- drop table student
- delete from student
  - Keeps the empty table around
- alter table
  - alter table student add address varchar(50);
  - alter table student drop tot_cred;

SQL Constructs: Insert/Delete/Update Tuples

- INSERT INTO <name> (<field names>) VALUES (<field values>)
  insert into instructor values (‘10211’, ‘Smith’, ‘Biology’, 66000);
  insert into instructor (name, ID) values (‘Smith’, ‘10211’);
    -- NULL for other two
  insert into instructor (ID) values (‘10211’);
    -- FAIL

- DELETE FROM <name> WHERE <condition>:
  delete from department where budget < 80000;
  - Syntax is fine, but this command may be rejected because of referential integrity constraints.
SQL Constructs: Insert/Delete/Update Tuples

- **DELETE FROM** `<name> WHERE <condition>`
  
  **delete from** department **WHERE** budget < 80000;

<table>
<thead>
<tr>
<th>dept_name</th>
<th>building</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biology</td>
<td>Watson</td>
<td>90000</td>
</tr>
<tr>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>Elec. Eng.</td>
<td>Taylor</td>
<td>85000</td>
</tr>
<tr>
<td>Finance</td>
<td>Painter</td>
<td>120000</td>
</tr>
<tr>
<td>History</td>
<td>Painter</td>
<td>50000</td>
</tr>
<tr>
<td>Music</td>
<td>Packard</td>
<td>80000</td>
</tr>
<tr>
<td>Physics</td>
<td>Watson</td>
<td>70000</td>
</tr>
</tbody>
</table>

*Figure 2.5* The `department` relation.

We can choose what happens:

1. Reject the delete, or
2. Delete the rows in Instructor (may be a cascade), or
3. Set the appropriate values in Instructor to NULL

---

**SQL Constructs: Insert/Delete/Update Tuples**

- **DELETE FROM** `<name> WHERE <condition>`
  
  **delete from** department **WHERE** budget < 80000;

```sql
create table instructor
(ID varchar(5),
 name varchar(20) not null,
 dept_name varchar(20),
 salary numeric(8,2) check (salary > 29000),
 primary key (ID),
 foreign key (dept_name) references department
 on delete set null
);
```

We can choose what happens:

1. Reject the delete (nothing), or
2. Delete the rows in Instructor (on delete cascade), or
3. Set the appropriate values in Instructor to NULL (on delete set null)
SQL Constructs: Insert/Delete/Update Tuples

- DELETE FROM <name> WHERE <condition>
  - Delete all classrooms with capacity below average
    ```sql
delete from classroom where capacity < (select avg(capacity) from classroom);
```
  - Problem: as we delete tuples, the average capacity changes
  - Or: delete the smallest classroom

- Solution used in SQL:
  - First, compute \text{avg} capacity and find all tuples to delete
  - Next, delete all tuples found above (without recomputing \text{avg} or retesting the tuples)

UPDATE <name> SET <field name> = <value> WHERE <condition>

- Increase all salaries’s over $100,000 by 6%, all other receive 5%.
- Write two update statements:
  ```sql
  update instructor
  set salary = salary * 1.06
  where salary > 100000;

  update instructor
  set salary = salary * 1.05
  where salary \leq 10000;
  ```
  - The order is important
  - Can be done better using the \text{case} statement
SQL Constructs: Insert/Delete/Update Tuples

- UPDATE <name> SET <field name> = <value> WHERE <condition>
  - Increase all salaries's over $100,000 by 6%, all other receive 5%.
  - Can be done better using the case statement
    UPDATE instructor
    SET salary =
    CASE
      WHEN salary > 100000
        THEN salary * 1.06
      WHEN salary <= 100000
        THEN salary * 1.05
    END;

Outline

- Overview of modeling
- Relational Model (Chapter 2)
  - Basics
  - Keys
  - Relational operations
  - Relational algebra basics
- SQL (Chapter 3)
  - Basic Data Definition (3.2)
  - Basic Queries (3.3-3.5)
  - Null values (3.6)
  - Aggregates (3.7)
Basic Query Structure

```
select A_1, A_2, ..., A_n
from r_1, r_2, ..., r_m
where P
```

Attributes or expressions
Relations (or queries returning tables)
Predicates

Find the names of all instructors:
```
select name
from instructor
```

Apply some filters (predicates):
```
select name
from instructor
where salary > 80000 and dept_name = 'Finance';
```

Remove duplicates:
```
select distinct name
from instructor
```

Order the output:
```
select distinct name
from instructor
order by name desc
```

Basic Query Constructs

Select all attributes:
```
select *
from instructor
```

Expressions in the select clause:
```
select name, salary < 100000
from instructor
```

Find the names of all instructors:
```
select name
from instructor
```

More complex filters:
```
select name
from instructor
where (dept_name != 'Finance' and salary > 75000)
or (dept_name = 'Finance' and salary > 85000);
```

A filter with a subquery:
```
select name
from instructor
where dept_name in (select dept_name from department where budget < 100000);
```
Basic Query Constructs

Renaming tables or output column names:

```sql
select i.name, i.salary * 2 as double_salary
from instructor i
where i.salary < 80000 and i.name like '%g_';
```

Find the names of all instructors:

```sql
select name
from instructor
```

More complex expressions:

```sql
select concat(name, concat(', ', dept_name))
from instructor;
```

Careful with NULLs:

```sql
select name
from instructor
where salary < 100000 or salary >= 100000;
```

Wouldn’t return the instructor with NULL salary (if any)

Multi-table Queries

Use predicates to only select “matching” pairs:

```sql
select *
from instructor i, department d
where i.dept_name = d.dept_name;
```

Cartesian product:

```sql
select *
from instructor, department
```

Identical (in this case) to using a natural join:

```sql
select *
from instructor natural join department;
```

Natural join does an equality on common attributes – doesn’t work here:

```sql
select *
from instructor natural join advisor;
```

Instead can use “on” construct (or where clause as above):

```sql
select *
from instructor join advisor on (i_id = id);
```
Multi-table Queries

3-Table Query to get a list of instructor-teaches-course information:

```sql
select i.name as instructor_name, c.title as course_name
from instructor i, course c, teaches
where i.ID = teaches.ID and c.course_id = teaches.course_id;
```

Beware of unintended common names (happens often)
You may think the following query has the same result as above – it doesn’t

```sql
select name, title
from instructor natural join course natural join teaches;
```

I prefer avoiding “natural joins” for that reason

Set operations

Find courses that ran in Fall 2009 or Spring 2010

```sql
(select course_id from section where semester = ‘Fall’ and year = 2009)
union
(select course_id from section where semester = ‘Spring’ and year = 2010);
```

In both:

```sql
(select course_id from section where semester = ‘Fall’ and year = 2009)
intersect
(select course_id from section where semester = ‘Spring’ and year = 2010);
```

In Fall 2009, but not in Spring 2010:

```sql
(select course_id from section where semester = ‘Fall’ and year = 2009)
except
(select course_id from section where semester = ‘Spring’ and year = 2010);
```
Set operations: Duplicates

Union/Intersection/Except eliminate duplicates in the answer (the other SQL commands don’t) (e.g., try ‘select dept_name from instructor’).

Can use “union all” to retain duplicates.

NOTE: The duplicates are retained in a systematic fashion (for all SQL operations)

Suppose a tuple occurs \( m \) times in \( r \) and \( n \) times in \( s \), then, it occurs:

- \( m + n \) times in \( r \) union all \( s \)
- \( \min(m,n) \) times in \( r \) intersect all \( s \)
- \( \max(0, m - n) \) times in \( r \) except all \( s \)

Outline

- Overview of modeling
- Relational Model (Chapter 2)
  - Basics
  - Keys
  - Relational operations
  - Relational algebra basics
- SQL (Chapter 3)
  - Basic Data Definition (3.2)
  - Basic Queries (3.3-3.5)
  - Null values (3.6)
  - Aggregates (3.7)
SQL: Nulls

The “dirty little secret” of SQL
(major headache for query optimization)

Can be a value of any attribute

e.g: branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

What does this mean?

(unknown) We don’t know Waltham’s assets
(inapplicable) Waltham has a special kind of account without assets
(withheld) We are not allowed to know

SQL: Nulls

Arithmetic Operations with NULL

n + NULL = NULL  (similarly for all arithmetic ops: +, -, *, /, mod, …)

e.g: branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

```
SELECT bname, assets * 2 as a2
FROM branch
```

<table>
<thead>
<tr>
<th>bname</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>18M</td>
</tr>
<tr>
<td>Perry</td>
<td>3.4M</td>
</tr>
<tr>
<td>Mianus</td>
<td>.8M</td>
</tr>
<tr>
<td>Waltham</td>
<td>NULL</td>
</tr>
</tbody>
</table>
SQL: Nulls

Arithmetic Operations with **NULL**

\[ n + \text{NULL} = \text{NULL} \]

(similarly for all arithmetic ops: +, -, *, /, mod, ...)

```sql
SELECT *
FROM branch
WHERE assets IS NULL
```

```
<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```

**Counter-intuitive:** NULL * 0 = NULL

Counter-intuitive: select * from movies
where length >= 120 or length <= 120
**SQL: Unknown**

Boolean Operations with Unknown

\[ n < \text{NULL} = \text{UNKNOWN} \quad \text{(similarly for all boolean ops: \( >, \leq, \geq, \neq, =, \ldots \))} \]

\[ \text{FALSE OR UNKNOWN} = \text{UNKNOWN} \]
\[ \text{TRUE AND UNKNOWN} = \text{UNKNOWN} \]

Intuition: substitute each of TRUE, FALSE for unknown. If different answer results, results is unknown

\[ \text{UNKNOWN OR UNKNOWN} = \text{UNKNOWN} \]
\[ \text{UNKNOWN AND UNKNOWN} = \text{UNKNOWN} \]
\[ \text{NOT (UNKNOWN)} = \text{UNKNOWN} \]

Can write:

```
SELECT ...
FROM ...
WHERE booleanexp IS UNKNOWN
```

UNKNOWN tuples are not included in final result

**Outline**

- Overview of modeling
- Relational Model (Chapter 2)
  - Basics
  - Keys
  - Relational operations
  - Relational algebra basics
- SQL (Chapter 3)
  - Basic Data Definition (3.2)
  - Basic Queries (3.3-3.5)
  - Null values (3.6)
  - Aggregates (3.7)
Other common aggregates:
- max, min, sum, count, stdev, ...

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2010
```

Find the average salary of instructors in the Computer Science
```
select avg(salary)
from instructor
where dept_name = 'Comp. Sci';
```

Can specify aggregates in any query.
Find max salary over instructors teaching in S'10
```
select max(salary)
from teaches natural join instructor
where semester = 'Spring' and year = 2010;
```

Aggregate result can be used as a scalar.
Find instructors with max salary:
```
select *
from instructor
where salary = (select max(salary) from instructor);
```

Following doesn’t work:
```
select *
from instructor
where salary = max(salary);
```
```
select name, max(salary)
from instructor
where salary = max(salary);
```
Split the tuples into groups, and compute the aggregate for each group

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

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
</tbody>
</table>

```

Attributes in the select clause must be aggregates, or must appear in the group by clause. Following wouldn't work

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

“having” can be used to select only some of the groups.

```
select dept_name, ID, avg (salary)
from instructor
group by dept_name
having avg(salary) > 42000;
```
Aggregates and NULLs

Given

```
branch =

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horseneck</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```

Aggregate Operations

```
SELECT SUM(assets) = 11.1 M
FROM branch

NULL is ignored for SUM

Same for AVG (3.7M), MIN (0.4M), MAX (9M)

But COUNT(*) returns

| COUNT | 4 |
```

Aggregates and NULLs

Given

```
SELECT SUM(assets) = NULL
FROM branch

Same as AVG, MIN, MAX

But COUNT(assets) returns

| COUNT | 0 |
```
Summary

- Relational Model (Chapter 2)
  - Basics
  - Keys
  - Relational operations
  - Relational algebra basics
- SQL (Chapter 3)
  - Setting up the PostgreSQL database
  - Data Definition (3.2)
  - Basics (3.3-3.5)
  - Null values (3.6)
  - Aggregates (3.7)
  - Advanced operators

With Clause

- The **with** clause provides a way of defining a temporary table (or “view”) whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```sql
with max_budget (value) as
  (select max(budget)
   from department)
select *
from department, max_budget
where department.budget = max_budget.value;
```
**With Clause, cont**

- WITH
  - b AS ((SELECT * FROM borders) UNION (SELECT country2,country1...)
  - cd AS (SELECT code FROM country WHERE name='Germany'),
  - b1 AS (SELECT UNIQUE b.country1 FROM b,cd WHERE b.country2 = cd.code),
  - b2 AS (SELECT UNIQUE b.country1 FROM b,b1 WHERE (b.country2 = b1.country1)),
  - b3 AS ((select * from b2) minus (select * from b1))
  - SELECT name FROM b21,country WHERE country.code = b3.country1;

**String Operations**

- SQL includes a string-matching operator for comparisons on character strings. The operator “like” uses patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.

- Find the names of all instructors whose name includes the substring “dar”.
  ```sql
  select name
  from instructor
  where name like '%dar%'
  ```

- Match the string “100 %”
  ```sql
  like '100 \%
  ```

- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.
Ordering the Display of Tuples

- List in alphabetic order the names of all instructors
  
  ```sql
  select distinct name
  from instructor
  order by name
  ```

- We may specify `desc` for descending order or `asc` for ascending order, for each attribute; ascending order is the default.
  
  Example: `order by name desc`

- Can sort on multiple attributes
  
  Example: `order by dept_name, name`

---

Joins

- “cross join” forms the $M \times N$ Cartesian product
  
  ```sql
  SELECT * FROM T1 CROSS JOIN T2
  ```

- “natural join” joins two tables on common columns

- “inner join” joins two tables using an “on clause”
  
  Can be thought of as a generalized natural join

- “outer join” (left| right | full)
  
  Includes rows that did not match w/ NULL values
Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values:
  - *null* signifies that the value is unknown or does not exist
  - All comparisons involving *null* are (roughly speaking) *false* by definition.

Outer Join – Example

- Relation *instructor1*

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
</tr>
</tbody>
</table>

- Relation *teaches1*

<table>
<thead>
<tr>
<th>ID</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>CS-101</td>
</tr>
<tr>
<td>12121</td>
<td>FIN-201</td>
</tr>
<tr>
<td>76766</td>
<td>BIO-101</td>
</tr>
</tbody>
</table>
Outer Join – Example

• natural join

SELECT * FROM instructor NATURAL JOIN teaches

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>CS-101</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>FIN-201</td>
</tr>
</tbody>
</table>

• left outer join

SELECT * FROM instructor LEFT JOIN teaches

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>CS-101</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>FIN-201</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>null</td>
</tr>
</tbody>
</table>

• right outer join

SELECT * FROM instructor RIGHT JOIN teaches

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>CS-101</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>FIN-201</td>
</tr>
<tr>
<td>76766</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

• full outer join

SELECT * FROM instructor FULL OUTER JOIN teaches

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>CS-101</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>FIN-201</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>null</td>
</tr>
<tr>
<td>76766</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>