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:
choose the attributes:
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

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

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

Expressions in the select clause:
```
select name, salary < 100000
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);
```

Renaming tables or output column names:
```
select i.name, i.salary * 2 as double_salary
from instructor i
where i.salary < 80000 and i.name like 'g_';
```

More complex expressions:
```
select concat(name, concat(' ', dept_name))
from instructor;
```

```
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:

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

Cartesian product:
```
select *
from instructor, department
```

Almost same (in this case) to using natural join:
```
select *
from instructor natural join department;
```

Natural join does an equality on common attributes – doesn’t work here:
```
select *
from instructor natural join advisor;
```

Instead can use “on” construct (or where clause as above):
```
select *
from instructor join advisor on (i_id = id);
```

---

Multi-table Queries

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

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

```
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

\[(\text{select course_id from section where semester = ‘Fall’ and year = 2009})\]
\[\text{union}\]
\[(\text{select course_id from section where semester = ‘Spring’ and year = 2010})\];

In both:
\[(\text{select course_id from section where semester = ‘Fall’ and year = 2009})\]
\[\text{intersect}\]
\[(\text{select course_id from section where semester = ‘Spring’ and year = 2010})\];

In Fall 2009, but not in Spring 2010:
\[(\text{select course_id from section where semester = ‘Fall’ and year = 2009})\]
\[\text{except}\]
\[(\text{select course_id from section where semester = ‘Spring’ and year = 2010})\];

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 \(r\) \text{union all} \(s\)
\[\min(m,n)\] times in \(r\) \text{intersect all} \(s\)
\[\max(0, m - n)\] times in \(r\) \text{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

<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

\[ n + \text{NULL} = \text{NULL} \]  
(similarly for all arithmetic ops: \(+, -, \ast, /, \text{mod}, \ldots\)

e.g.: \(\text{branch} = \)

<table>
<thead>
<tr>
<th>\text{bname}</th>
<th>\text{bcity}</th>
<th>\text{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>

\[
\text{SELECT bname, assets} \ast 2 \text{ as a2 } \\
\text{FROM branch}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{bname} & \text{a2} \\
\hline
\text{Downtown} & 18M \\
\text{Perry} & 3.4M \\
\text{Mianus} & .8M \\
\text{Waltham} & \text{NULL} \\
\hline
\end{array}
\]

---

SQL: Nulls

Arithmetic Operations with NULL

\[ n + \text{NULL} = \text{NULL} \]  
(similarly for all arithmetic ops: \(+, -, \ast, /, \text{mod}, \ldots\)

e.g.: \(\text{branch} = \)

<table>
<thead>
<tr>
<th>\text{bname}</th>
<th>\text{bcity}</th>
<th>\text{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>

\[
\text{SELECT * } \\
\text{FROM branch}
\]

\[
\text{WHERE assets IS NULL}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{bname} & \text{bcity} & \text{assets} \\
\hline
\text{Waltham} & \text{Boston} & \text{NULL} \\
\hline
\end{array}
\]
**SQL: Nulls**

Counter-intuitive: NULL * 0 = NULL

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

---

**SQL: Unknown**

Boolean Operations with Unknown

n < NULL = UNKNOWN  (similarly for all boolean ops: >, <=, >=, <>, =, ...)

FALSE OR UNKNOWN = UNKNOWN
TRUE AND UNKNOWN = UNKNOWN

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

UNKNOWN OR UNKNOWN = UNKNOWN
UNKNOWN AND UNKNOWN = UNKNOWN
NOT (UNKNOWN) = UNKNOWN

**Can write:**

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

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

In a join:
```sql
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:
```sql
select *
from instructor
where salary = (select max(salary) from instructor);
```
Aggregates

Aggregate result can be used as a scalar.
Find instructors with max salary:

```sql
select *
from instructor
where salary = (select max(salary) from instructor);
```

The following do not work:

```sql
select *
from instructor
where salary = max(salary);
```

```sql
select name, max(salary)
from instructor;
```

Split the tuples into groups, and compute the aggregate for each group

```sql
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>

<table>
<thead>
<tr>
<th>dept_name</th>
<th>avg_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>Comp. Sci.</td>
<td>77333</td>
</tr>
<tr>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>Finance</td>
<td>85000</td>
</tr>
<tr>
<td>History</td>
<td>61000</td>
</tr>
<tr>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>Physics</td>
<td>91000</td>
</tr>
</tbody>
</table>
Aggregates: Group By

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

Having used to select from aggregated rows
Where used to select non-aggregated rows

Aggregates and NULLs

Given

<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) =
FROM branch
```

<table>
<thead>
<tr>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 M</td>
</tr>
</tbody>
</table>

NULL is ignored for SUM

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

But COUNT (*) returns

<table>
<thead>
<tr>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Aggregates and NULLs

Given

\[
\text{branch} = \begin{array}{|c|c|c|}
\hline
\text{bname} & \text{bcity} & \text{assets} \\
\hline
\end{array}
\]

\[
\text{SELECT SUM (assets) = } \begin{array}{|c|}
\hline
\text{SUM} \\
\hline
\text{NULL} \\
\hline
\end{array}
\]
\[
\text{FROM branch}
\]

- Same as AVG, MIN, MAX
- But COUNT (assets) returns

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 %' escape '\'
  ```

- 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
  - SELECT * FROM T1 CROSS JOIN T2
    - SELECT * FROM T1,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 inner join and then adds tuples from relations that do not match tuples in the other relation.
- 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>

**natural join**

```sql
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>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>null</td>
</tr>
</tbody>
</table>

**left outer join**

```sql
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>
Outer Join – Example

- 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>BIO-101</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>BIO-101</td>
</tr>
</tbody>
</table>

---

Joins in Postgresql

- **T1 CROSS JOIN T2**

- **T1** { [INNER] | [LEFT | RIGHT | FULL] [OUTER] } JOIN **T2** ON boolean_expression
- **T1** { [INNER] | [LEFT | RIGHT | FULL] [OUTER] } JOIN **T2** USING ( join column list )
- **T1 NATURAL** { [INNER] | [LEFT | RIGHT | FULL] [OUTER] } JOIN **T2**

DROP TABLE instructor;
DROP TABLE teaches;
CREATE TABLE instructor (id INTEGER, name VARCHAR(50), dept_name VARCHAR(50));
CREATE TABLE teaches (id INTEGER, course_id VARCHAR(50));

INSERT INTO instructor VALUES (10101, 'Srinivasan', 'Comp. Sci.'), (12121, 'Wu', 'Finance'), (15151, 'Mozart', 'Music');

INSERT INTO teaches VALUES (10101, 'CS-101'), (12121, 'FIN-201'), (76766, 'BIO-101');

SELECT * FROM instructor i cross teaches t;
SELECT * FROM instructor i cross join teaches t;
SELECT * FROM instructor i natural join teaches t;
SELECT * FROM instructor i LEFT JOIN teaches t on (i.id=t.id);
SELECT * FROM instructor i RIGHT JOIN teaches t USING (id);
SELECT * FROM instructor i FULL JOIN teaches t USING (id);
Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A subquery is a select-from-where expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Example Query

- Find courses offered in Fall 2009 and in Spring 2010
  ```sql
  select distinct course_id
  from section
  where semester = 'Fall' and year = 2009 and
  course_id in (select course_id
                 from section
                 where semester = 'Spring' and year = 2010);
  ```
- Find courses offered in Fall 2009 but not in Spring 2010
  ```sql
  select distinct course_id
  from section
  where semester = 'Fall' and year = 2009 and
  course_id not in (select course_id
                     from section
                     where semester = 'Spring' and year = 2010);
  ```

Already did w/ set operations
Example Query

- Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101

```sql
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);
```

- Note: Above query could also be written more efficiently with a join. The formulation above is simply to illustrate SQL features.

Definition of Some Clause

- F <comp> some r ⇔ ∃ t ∈ r such that (F <comp> t)
  Where <comp> can be: <, >=, >, =, !=, <>

(5 < some 5) = true
(5 < some 6) = true (read: 5 < some tuple in the relation)
(5 < some 0) = false
(5 = some 5) = true
(5 != some 5) = true (the 0)
(= some) ≡ in
However, (!= some) is not the same as not in
Set Comparison

• Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```sql
select distinct T.name
from instructor T, instructor S
where T.salary > S.salary and S.dept name = 'Biology';
```

■ Same query using > some clause

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

Definition of all clause

• $F <\text{comp}> \text{ all } r \iff \forall t \in r \ (F <\text{comp}> t)$

<table>
<thead>
<tr>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(5 &lt; \text{ all } 6)$</td>
<td>false</td>
</tr>
<tr>
<td>$(5 &lt; \text{ all } 10)$</td>
<td>true</td>
</tr>
<tr>
<td>$(5 = \text{ all } 5)$</td>
<td>false</td>
</tr>
<tr>
<td>$(5 \neq \text{ all } 6)$</td>
<td>true   (since $5 \neq 4$ and $5 \neq 6$)</td>
</tr>
</tbody>
</table>
Example Query

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

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

END