Outline

- Overview of modeling
- Relational Model (Chapter 2)
  - Basics
  - Keys
  - Relational operations
  - Relational algebra basics
- SQL (Chapter 3)
  - Basic Data Definition (3.2)
  - Setting up the PostgreSQL database
  - 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 account without assets
(withheld) We are not allowed to know

SQL: Nulls

Arithmetic Operations with

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

Boolean Operations with Null

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

e.g: \[ \text{branch} = \begin{array}{|c|c|c|} \hline \text{bname} & \text{bcity} & \text{assets} \\
\text{Downtown} & \text{Boston} & 9M \\
\text{Perry} & \text{Horseneck} & 1.7M \\
\text{Mianus} & \text{Horseneck} & .4M \\
\text{Waltham} & \text{Boston} & \text{NULL} \\
\hline \end{array} \]

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

Counter-intuitive: NULL \* 0 = NULL

Counter-intuitive: select * from movies
where length \( \geq \) 120 or length \( \leq \) 120
SQL: Unknown

Boolean Operations with Unknown

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

\[
\begin{align*}
\text{FALSE OR UNKNOWN} &= \text{UNKNOWN} \\
\text{TRUE AND UNKNOWN} &= \text{UNKNOWN}
\end{align*}
\]

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

\[
\begin{align*}
\text{UNKNOWN OR UNKNOWN} &= \text{UNKNOWN} \\
\text{UNKNOWN AND UNKNOWN} &= \text{UNKNOWN} \\
\text{NOT (UNKNOWN)} &= \text{UNKNOWN}
\end{align*}
\]

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)
  - Setting up the PostgreSQL database
  - Basic Queries (3.3-3.5)
  - Null values (3.6)
  - Aggregates (3.7)
Aggregates

Find the average salary of instructors in the Computer Science

```sql
select avg(salary)
from instructor
where dept_name = 'Comp. Sci';
```

Aggregate result can be used as a scalar.

Find instructors with max salary:

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

Can specify aggregates in any query.

Find max salary over instructors teaching in S’10

```sql
select max(salary)
from teaches natural join instructor
where semester = 'Spring' and year = 2010;
```

Following doesn’t work:

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

```sql
select name, max(salary)
from instructor
where salary = max(salary);
```
Split the tuples into groups, and computer the aggregate for each group

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

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

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

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

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

Given

\[ \text{branch} = \]

<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>0.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Aggregate Operations

\[
\text{SELECT SUM (assets) =}
\]

\[
\text{FROM branch}
\]

\[
\begin{array}{c}
\text{SUM} \\
11.1 \text{ M}
\end{array}
\]

\text{NULL is ignored for SUM}

\text{Same for AVG (3.7M), MIN (0.4M), MAX (9M)}

\text{But COUNT (*) returns}

\[
\begin{array}{c}
\text{COUNT} \\
4
\end{array}
\]

\text{Also for COUNT(assets) -- returns 3}

Aggregates and NULLs

Given

\[ \text{branch} = \]

<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>0.4M</td>
</tr>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

\[
\text{SELECT SUM (assets) =}
\]

\[
\text{FROM branch}
\]

\[
\begin{array}{c}
\text{SUM} \\
\text{NULL}
\end{array}
\]

\text{• Same as AVG, MIN, MAX}

\text{• But COUNT (assets) returns}

\[
\begin{array}{c}
\text{COUNT} \\
0
\end{array}
\]

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)
  - Other

With Clause

- The `with` clause allows temporary tables (or “views”) available only to the surrounding query.
- 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 (...)),
  b21 AS ((select * from b2) minus (select * from b1))
SELECT name FROM b21,country WHERE country.code = b21.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”.
  ```
  select name
  from instructor
  where name like '%dar%'
  ```

- Match the string “100 %”
  ```
  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`

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

- Join

  *instructor1* × *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

  *instructor1* × *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**
  \[ \text{id} \times_{\cdot} \text{teaches} \]

<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**
  \[ \text{id} \bowtie_{\cdot} \text{teaches} \]

<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>

Outer Join using Joins

- Outer join can be expressed using basic operations
  - e.g. \( r \bowtie s \) can be written as
    \[
    (r \bowtie s) \cup (r - \Pi_R(r \bowtie s) \times \{null, \ldots, null\})
    \]
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);
```
Example Query

- Find the total number of (distinct) students who have taken course sections taught by the 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);
```

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

```
SELECT count (distinct takes.ID)
FROM takes, teaches
WHERE takes.course_id=teaches.course_id AND
takes.sec_id=teaches.sec_id AND
takes.semester=teaches.semester AND
takes.year=teaches.year AND
teaches.ID='10101';
```

Definition of Some Clause

- \( F \text{some} r \iff \exists t \in r \text{ such that } (F <\text{comp}> t) \)

Where \( <\text{comp}> \) can be: \(<, \geq, >, =, !=, <>\)

\[
\begin{array}{c|ccc}
 & 0 & 5 & 6 \\
\hline
(5 < \text{some}) & \text{true} \\
(5 \geq \text{some}) & \text{false} \\
(5 = \text{some}) & \text{true} \\
(5 \neq \text{some}) & \text{true} (\text{since } 0 \neq 5)
\end{array}
\]

(\(= \text{some}) = \text{in} \)

However, \((\neq \text{some}) \neq \text{in}\)
Set Comparison

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

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

Same query using \(>\) some clause:

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

Definition of all Clause

- \( F \text{ <comp> all } r \leftrightarrow \forall t \in r \ (F \text{ <comp> } t) \)

\[
\begin{array}{c}
(5 < \text{ all } 5) = \text{false} \\
(5 < \text{ all } 6) = \text{true} \\
(5 = \text{ all } 5) = \text{false} \\
(5 \neq \text{ all } 6) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6) \\
\end{array}
\]
Example Query

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

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

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.

  - exists  \( r \leftrightarrow r \neq \emptyset \)
  - not exists  \( r \leftrightarrow r = \emptyset \)
Correlated Subqueries

- Yet another way of specifying the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”

```sql
select course_id
from section S
where semester = 'Fall' and year = 2009 and
  exists (select *
    from section T
    where semester = 'Spring' and year = 2010
      and S.course_id = T.course_id);
```

- **Correlation name** or **correlation variable**