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)

### Aggregates

Other common aggregates:
- max, min, sum, count, stdev, ...

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

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: Group By

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, 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

```sql
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

```sql
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)

Also for COUNT(assets) -- returns 3

But COUNT (*) returns:

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

Given

\[
\text{branch} = \begin{array}{ccc}
\text{bname} & \text{bcity} & \text{assets} \\
\end{array}
\]

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

- AVG, MIN, MAX also returns NULL
- 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 b3,country WHERE country.code = b3.country1;
```
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)
  - More Joins

Join Types

- “cross join” forms the $M \times N$ Cartesian product
  - SELECT * FROM T1 CROSS JOIN T2 or 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, more specific, 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.
• Both natural and inner variants
• Uses null values:
  • null signifies that the value is unknown or does not exist
  • All comparisons involving null are (roughly speaking) false by definition.
• Computes the inner join and then adds tuples from relations that do not match tuples in the other relation.

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>
<tr>
<td>15151</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>76766</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- left outer join

SELECT * FROM instructor NATURAL 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>
<tr>
<td>76766</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- right outer join

SELECT * FROM instructor NATURAL 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>
<tr>
<td>15151</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- full outer join

SELECT * FROM instructor NATURAL FULL 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>
<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>
## Outer Join – Example

SELECT * FROM instructor NATURAL FULL 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>

SELECT * FROM instructor INNER FULL JOIN teaches

ON instructor.ID = teaches.ID

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

*inner* is the default!

## Example queries

- Download [uni.sql](#) and [cmds.sql](#)
- `createdb uni`
- `psql uni < uni.sql`
- Type in lines from `cmds.sql` one at a time.
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

- Number of students who took instructor ID 3199

```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 = '3199');
```

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

```sql
select count(distinct a.id)
from takes a inner join teaches b
on b.id='3199'
    and a.course_id=b.course_id
    and a.semester=b.semester
    and a.year=b.year
    and a.sec_id=b.sec_id;
```

Definition of Some Clause

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

(5 < some 0 5 6 ) =
(5 < some 0 5 ) =
(5 = some 0 5 ) =
(5 != some 0 5 ) =

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

- Same query using join

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

Definition of all clause

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

\[
\begin{array}{c|c|c}
  (5 < \text{all} & 0 & 0 \\
  5 & 5 & 1 \\
  6 & 1 & 0 \\
\end{array} =
\]

\[
\begin{array}{c|c|c}
  (5 < \text{all} & 6 & 0 \\
  10 & 5 & 1 \\
\end{array} =
\]

\[
\begin{array}{c|c|c}
  (5 = \text{all} & 4 & 0 \\
  5 & 1 & 0 \\
\end{array} =
\]

\[
\begin{array}{c|c|c}
  (5 \neq \text{all} & 4 & 0 \\
  6 & 1 & 0 \\
\end{array} =
\]

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:` \( r \text{ intersect } s \neq \emptyset \)
- `not exists:` \( r \text{ intersect } s = \emptyset \)
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 F
where semester = 'Fall' and year = 2009 and
exists (select *
    from section S
    where semester = 'Spring' and year = 2010
    and F.course_id = S.course_id);
```

• Correlation name or correlation variable

Not Exists

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

```sql
select S.ID, S.name
from student 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));
```

■ Note that $X - Y = \emptyset \iff X \subseteq Y$
Test for Absence of Duplicate Tuples

- Find all courses that were offered exactly once in 2009:

**WRONG:** unique used only during table creation.

```sql
select T.course_id
from course T
where unique (select R.course_id
               from section R
               where T.course_id = R.course_id)
```

**RIGHT:**

```sql
select T.course_id from course T
where 1 = (select count(R.course_id)
           from section R
           where T.course_id = R.course_id and R.year = 2009);
```

Derived Relations

- SQL allows a subquery expression to be used in the `from` clause
- Find the average instructors’ salaries of those departments where the average salary is greater than $42,000.”

```sql
select dept_name, avg_salary
from (select dept_name, avg(salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

- Note that the following is equivalent:

```sql
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name
having avg(salary) > 42000;
```
Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructor’s name and department, but not the salary. This person should see a relation described, in SQL, by:

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

- A view provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

---

Views

- A view of instructors without their salary:

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

- Find all instructors in the Biology department

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

- Create a view of department salary totals

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

- View definition is not the same as creating a new relation by evaluating the query expression: Rather, a view definition **causes the saving of an expression**; the expression is substituted into queries using the view.
Views Defined Using Other Views

- create view physics_fall_2009 as
  select course.course_id, sec_id, building, room_number
  from course, section
  where course.course_id = section.course_id
    and course.dept_name = 'Physics'
    and section.semester = 'Fall'
    and section.year = '2009';

- create view physics_fall_2009_watson as
  select course_id, room_number
  from physics_fall_2009
  where building = 'Watson';

Views Defined Using Other Views

- Derived views include complete definitions of used views

  create view physics_fall_2009_watson as
  select course_id, room_number
  from physics_fall_2009
  where building = 'Watson';

  create view physics_fall_2009_watson as
  (select course.course_id, building, room_number
   from course, section
   where course.course_id = section.course_id
     and course.dept_name = 'Physics'
     and section.semester = 'Fall'
     and section.year = '2009')
  where building = 'Watson';
View Expansion (implementation)

Views can be used in defining other views:

- View $v_i$ directly depends on a view $v_j$ if $v_j$ is used in $v_i$’s definition.
- View $v_i$ can depend on view $v_j$ either directly or indirectly.
- View $v$ is recursive if it depends on itself.

  This would be bad.

View Expansion (implementation)

- A way to interpret queries w/ views...
  - Let view $v_i$ be defined by an expression $e_i$ that may itself contain uses of view relations.
  - View expansion of an expression $e$ repeats the following replacement step:
    repeat
      Find any view relation $v_i$ in $e$
      Replace the view relation $v_i$ by expression $e_i$
    until no more view relations are present in $e$
  - As long as the view definitions are not recursive, this loop will terminate.
Update of (through) a View

- Add a new tuple to faculty view which we defined earlier
  \[\text{insert into faculty values ('30765', 'Green', 'Music');}\]
  This insertion must be represented by the insertion of the tuple
  \[('30765', 'Green', 'Music', null) \quad \leftrightarrow \text{salary column}\]
  into the instructor relation.

Some Updates cannot be Translated Uniquely

- create view instructor_info as
  \[
  \text{select ID, name, building}
  \text{from instructor, department}
  \text{where instructor.dept_name = department.dept_name;}
  \]
- insert into instructor_info values ('69987', 'White', 'Taylor');
  - which department, if multiple departments in Taylor?
  - what if no department is in Taylor?
- Most SQL implementations allow updates only on simple views:
  - The from clause has only one database relation.
  - The select clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
  - Any attribute not listed in the select clause can be set to null
  - The query does not have a group by or having clause.
And Some Not at All

• **create view** `history_instructors as``
  `select *``
  `from instructor``
  `where dept_name= 'History';``

• Insert ('25566', 'Brown', 'Biology', 100000) into `history_instructors` (no)

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`

**Triggers**

- A *trigger* is a statement that is executed automatically by the system as a side effect of a modification to the database.

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  
  1. setting the account balance to zero
  2. creating a loan in the amount of the overdraft
  3. giving this loan a loan number identical to the account number of the overdrawn account
Trigger Example in SQL:1999

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
    actions to be taken
end
```

(just an example, as databases have wildly different syntaxes)
Triggers...

- External World Actions
  - How does the DB order something if the inventory is low?

- Syntax
  - Every system has its own syntax (including postgres)

- Careful with triggers
  - Cascading triggers, Infinite Sequences...

- More Info/Examples:
  - https://www.tutorialspoint.com/postgresql/postgresql_triggers.htm