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

More Joins

- “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 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 \textit{null} values:
  - \textit{null} signifies that the value is unknown or does not exist
  - All comparisons involving \textit{null} are (roughly speaking) \textbf{false} by definition.

Outer Join – Example

- Relation \textit{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 \textit{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**

```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>
</tbody>
</table>

- **left outer join**

```sql
SELECT * FROM instructor LEFT JOIN teaches
```

```sql
or 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>

- **right outer join**

```sql
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**

```sql
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** (try these)

<table>
<thead>
<tr>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DROP TABLE instructor;</code></td>
</tr>
<tr>
<td><code>DROP TABLE teaches;</code></td>
</tr>
<tr>
<td><code>CREATE TABLE instructor (id INTEGER, name VARCHAR(50), dept_name VARCHAR(50));</code></td>
</tr>
<tr>
<td><code>CREATE TABLE teaches (id INTEGER, course_id VARCHAR(50));</code></td>
</tr>
<tr>
<td><code>INSERT INTO instructor VALUES (10101, 'Srinivasan', 'Comp. Sci.'), (12121, 'Wu', 'Finance'), (15151, 'Mozart', 'Music');</code></td>
</tr>
<tr>
<td><code>INSERT INTO teaches VALUES (10101, 'CS-101'), (12121, 'FIN-201'), (76766, 'BIO-101');</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i CROSS JOIN teaches t;</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i INNER JOIN teaches t ON i.id = t.id;</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i LEFT JOIN teaches t ON i.id = t.id;</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i RIGHT JOIN teaches t ON i.id = t.id;</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i FULL JOIN teaches t ON i.id = t.id;</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i NATURAL JOIN teaches t;</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i LEFT JOIN teaches t USING (id);</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i RIGHT JOIN teaches t USING (id);</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i FULL JOIN teaches t USING (id);</code></td>
</tr>
<tr>
<td><code>SELECT * FROM instructor i SEMI JOIN teaches t;</code></td>
</tr>
</tbody>
</table>

**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

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

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

\[
\begin{array}{c|c|c}
\text{(some)} & 0 & 5 \\
\hline
0 & 5 & 6 \\
\end{array}
\]

(read: \(5 < \text{some tuple in the relation}\))

\[
\begin{array}{c|c|c}
\text{(some)} & 0 & 5 \\
\hline
0 & 5 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{(some)} & 0 & 5 \\
\hline
0 & 5 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{(some)} & 0 & 5 \\
\hline
0 & 5 & 0 \\
\end{array}
\]

\( (= \text{some}) \equiv \text{in} \)
However, \((!= \text{some})\) is not the same as \(\text{not in}\)

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**

- $F \langle \text{comp} \rangle \text{ all } r \iff \forall t \in r \ (F \langle \text{comp} \rangle t)$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(5 &lt; \text{all})$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(6 &lt; \text{all})$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(5 = \text{all})$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(5 \neq \text{all})$</td>
<td></td>
<td></td>
<td></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');
```
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 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.

```
select distinct S.ID, S.name
from student as 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.

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

RIGHT:

```
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
  
  ```sql
  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
  
  ```sql
  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

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

```sql
create view physics_fall_2009_watson as
(select course_id, room_number
from (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 \( v_1 \) directly depends on a view \( v_2 \) if \( v_2 \) is used in \( v_1 \)'s definition.
- View \( v_1 \) can depend on view \( v_2 \) either directly or indirectly.
- View \( v \) is recursive if it depends on itself.
**View Expansion (implementation)**

- A way to interpret queries w/ views...
  - Let view $v$ be defined by an expression $e$, 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

  ```
  insert into faculty values ('30765', 'Green', 'Music');
  ```

  This insertion must be represented by the insertion of the tuple

  ```
  ('30765', 'Green', 'Music', null) ← salary column
  ```

  into the $instructor$ relation.
Some Updates cannot be Translated Uniquely

- `create view instructor_info as
  select ID, name, building
  from instructor, department
  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`
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
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

```sql
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
```
Trigger Example in SQL:1999

```sql
create trigger overdraft-trigger after update on account
    referencing new row as nrow
    for each row
    when nrow.balance < 0
begin atomic
    insert into borrower
        (select customer-name, account-number
            from depositor
            where nrow.account-number = depositor.account-number);
    insert into loan values
        (nrow.account-number, nrow.branch-name, nrow.balance);
    update account set balance = 0
    where account.account-number = nrow.account-number
end
```

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](https://www.tutorialspoint.com/postgresql/postgresql_triggers.htm)
Continuing

- SQL (Chapter 3, 4)
  - Views (4.2)
  - Triggers (5.3)
  - Integrity Constraints (4.4)
  - Functions and Procedures (5.2), Recursive Queries (5.4), Authorization (4.6), Ranking (5.5)
  - Transactions
- Relational Algebra
- E/R Diagrams (7)

- Some Complex SQL Examples

Integrity Constraints

- Predicates on the database
- Must always be true (checked whenever db gets updated)

- There are the following 4 types of IC’s:
  - Key constraints (1 table)
    - e.g., 2 accts can’t share the same acct_no
  - Attribute constraints (1 table)
    - e.g., accts must have nonnegative balance
  - Referential Integrity constraints (2 tables)
    - E.g. bnames associated w/ loans must be names of real branches
  - Global Constraints (n tables)
    - E.g., all loans must be carried by at least 1 customer with a savings acct

  done