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)
  - Joins
  - Null values (3.6)
  - Aggregates (3.7)

Multi-table Queries

Cartesian product:
select *
from *instructor, department*

Use predicates to only select "matching" pairs:
select *
from *instructor i, department d*
where *i.dept_name = d.dept_name;*

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

```
teaches(id, course_id, sec_id, semester, year)
instructor(id, name, dept_name, salary)
course(      id, title, dept_name, credits)
```

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

(**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:

(**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:

(**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 =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

What does this mean?

(not known) *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 =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

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

=

bname	a2
Downtown	18M
Perry	3.4M
Mianus	.8M
Waltham	NULL

Though scalar operations w/ null result in null, aggregate functions operate differently.

SQL: Nulls

Arithmetic Operations with NULL

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

e.g: branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

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

=

bname	bcity	assets
Waltham	Boston	NULL

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*: `>, <=, >=, <>, =, …`)

Intuition: substitute each of TRUE, FALSE for unknown.

If get different answers, result is unknown.

```
FALSE OR UNKNOWN = UNKNOWN
TRUE AND UNKNOWN = UNKNOWN
```

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

note that a predicate with value unknown *is not true…*

Can write:

```
SELECT …
FROM …
WHERE booleanexp IS UNKNOWN
```


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, …

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';*

In a join:
select max(*salary)*
from *teaches* **natural join** *instructor*
where *semester* = 'Spring'
and *year* = 2010;

Aggregates

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

The following do not work:

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

select *name,* **max***(salary)*
from *instructor;*

Aggregates: Group By

Split the tuples into groups, and compute the aggregate for each group
select *dept_name*, **avg** (*salary*)
from *instructor*
group by *dept_name*;

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

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

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

Though scalar operations w/ null *result in* null, *aggregate functions operate differently.*

branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

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

SUM
11.1 M

`NULL` *is ignored for SUM*

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

Also for COUNT(assets) -- returns 3

But `COUNT (*)` *returns*

COUNT
4

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:

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 DISTINCT b.country1 FROM b,cd WHERE b.country2 = cd.code),
-     b2 AS (SELECT DISTINCT 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;

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

  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*

More on Joins

- "cross join" forms the *M x 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" or "using" clause
  - Can be thought of as a generalized natural join
- "outer join" (left|right|full)
  - Effect is **natural join** plus rows that did not match, w/ NULL values
  - Two variations:
    - default requires *explicitly* naming the matching conditions, like inner
    - *natural* variant allows implicit matching conditions

More on Joins

ID	name	dept_name
10101	Srinivasan	Comp. Sci.
12121	Wu	Finance
15151	Mozart	Music

ID	course_id
10101	CS-101
12121	FIN-201
76766	BIO-101

- natural join

```
SELECT * FROM instructor NATURAL JOIN teaches
```

ID	*name*	*dept_name*	*course_id*
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201

- left outer join (or *LEFT JOIN)*

```
SELECT * FROM instructor i LEFT JOIN teaches t on (i.ID = t.ID)
```

```
SELECT * FROM instructor LEFT JOIN teaches USING (ID)
```

ID	*name*	*dept_name*	*course_id*
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201
15151	Mozart	Music	*null*

Outer Join – Example

ID	name	dept_name
10101	Srinivasan	Comp. Sci.
12121	Wu	Finance
15151	Mozart	Music

ID	course_id
10101	CS-101
12121	FIN-201
76766	BIO-101

- right outer join

```
SELECT * FROM instructor RIGHT JOIN teaches using (ID)
```

ID	*name*	*dept_name*	*course_id*
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201
76766	null	null	BIO-101

- full outer join

```
SELECT * FROM instructor FULL JOIN teaches using (ID)
```

ID	*name*	*dept_name*	*course_id*
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201
15151	Mozart	Music	*null*
76766	null	null	BIO-101

Defining an outer join w/ other operators

- Left outer join of instructor and department tables

```
SELECT i.*, d.*
FROM instructor i
INNER JOIN department d
USING (dept_name)

UNION

SELECT i.*, NULL as dept_name, NULL AS building, NULL AS budget
FROM instructor i
WHERE i.dept_name NOT IN (SELECT dept_name FROM department);
```

```
  id   |    name    | dept_name  |  salary  | dept_name  | building |  budget
-------+------------+------------+----------+------------+----------+-----------
 32343 | El Said    | History    | 60000.00 | History    | Painter  |  50000.00
 10101 | Srinivasan | Comp. Sci. | 65000.00 | Comp. Sci. | Taylor   | 100000.00
 33456 | Gold       | Physics    | 87000.00 | Physics    | Watson   |  70000.00
 15151 | Mozart     | Music      | 40000.00 | Music      | Packard  |  80000.00
 83821 | Brandt     | Comp. Sci. | 92000.00 | Comp. Sci. | Taylor   | 100000.00
 76766 | Crick      | Biology    | 72000.00 | Biology    | Watson   |  90000.00
 58583 | Califieri  | History    | 62000.00 | History    | Painter  |  50000.00
 98345 | Kim        | Elec. Eng. | 80000.00 | Elec. Eng. | Taylor   |  85000.00
 45565 | Katz       | Comp. Sci. | 75000.00 | Comp. Sci. | Taylor   | 100000.00
 12121 | Wu         | Finance    | 90000.00 |            |          |
 22222 | Einstein   | Physics    | 95000.00 | Physics    | Watson   |  70000.00
(11 rows)
```


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 LEFT JOIN teaches USING (id);
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);

SELECT * FROM instructor NATURAL LEFT JOIN teaches 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

```
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

```
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* 3199

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.

```
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* $\Leftrightarrow \exists\, t \in r$ such that (F <comp> *t*)
  Where <comp> can be: <, >=, >, =, !=, <>

(5 < **some** [0, 5, 6]) = true   (read: *5 < some tuple in the relation)*

(5 < **some** [0, 5]) = false

(5 = **some** [0, 5]) = true

(5 != **some** [0, 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.

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');