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)
  - Other

Definition of *all* clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> *t)*

(5 < **all** | 0, 5, 6) = false

(5 < **all** | 6, 10) = true

(5 = **all** | 4, 5) = false

(5 != **all** | 4, 6) = true (since 5 ne 4 and 5 ne 6)

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"

```
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 S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                   except
                    (select T.course_id
                     from takes T
                     where S.ID = T.ID));
```

Note that $X - Y = \emptyset$ means $X \subseteq Y$

Test for Absence of Duplicate Tuples

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

WRONG: **unique** is used to define constraints at 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

- Subqueries can even be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
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:

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

Views

- Might not want all users to see the entire logical model (that is, all the actual relations stored in the database.)
- A person who needs to know an instructor's name and department might not need to know the salary. This person should see a relation described, in SQL, by

  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**.

View Definition

- A view is defined using the **create view** statement which has the form:

  create view *v* **as** *<query expression>*

  where <query expression> is any legal SQL expression. The view name is represented by ***v.***

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- 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.

Example Views

- A view of instructors without their salary

  create view *faculty* **as**

  select *ID*, *name*, *dept_name*

  from *instructor*

- Find all instructors in the Biology department

  select *name*

  from *faculty*

  where *dept_name* = 'Biology'

- Create a view of department salary totals

  create view *departments_total_salary*(*dept_name*, *total_salary*) **as**

  select *dept_name*, **sum** (*salary*)

  from *instructor*

  group by *dept_name*;

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

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

- Effect is the following:

```
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

A view may be used to define another view:

- A view relation v_1 is said to ***depend directly*** on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to ***depend on*** view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be ***recursive*** if it depends on itself.

View Expansion

- 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

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

This insertion must be represented by the insertion of the tuple:

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

into the *instructor* relation.

Some Updates Do Not Translate 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*

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

Integrity Constraints

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

- There are 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 exist
  - Global Constraints (***n*** tables)
    E.g., all ***loans*** must be carried by at least 1 ***customer*** with a savings acct

Key Constraints

Idea: specifies that a relation is a set, not a bag

1. Primary Key:

```
CREATE TABLE branch(
        bname  CHAR(15)  PRIMARY KEY,
        bcity      CHAR(20),
        assets     INT);
```

or

```
CREATE TABLE depositor(
        cname   CHAR(15),
        acct_no  CHAR(5),
        PRIMARY KEY(cname, acct_no));
```

2. Candidate Keys:

```
CREATE TABLE customer (
        ssn     CHAR(9)    PRIMARY KEY,
        cname  CHAR(15),
        address CHAR(30),
        city          CHAR(10),
        UNIQUE (cname, address, city));
```


Key Constraints

Effect of SQL Key declarations:
PRIMARY (A1, A2, .., An) or
UNIQUE (A1, A2, ..., An)

Insertions: check for tuples with same values for A1, A2, .., An as inserted tuple. If found, **reject**

Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)

1. 1 primary key per table, several unique keys allowed.
2. Only primary key can be referenced by “foreign key” (ref integrity)
3. DBMS may treat primary key differently
   (e.g.: create an index on PK)

Attribute Constraints

- Idea:
  - Attach constraints to values of attributes
  - Enhances types system (e.g.: >= 0 rather than integer)

1. NOT NULL

e.g.: CREATE TABLE branch(
bname CHAR(15) NOT NULL,
....
)

Note: declaring bname as primary key also prevents null values

2. CHECK

e.g.: CREATE TABLE depositor(
....
balance int NOT NULL,
CHECK(balance >= 0),
....
)

affects insertions, updates in affected columns

Attribute Constraints

Domains:

associate constraints with DOMAINS rather than attributes

Instead of: CREATE TABLE depositor(
....
balance INT NOT NULL,
CHECK (balance >= 0)
)

One can write:

CREATE DOMAIN bank-balance INT (
CONSTRAINT not-overdrawn CHECK (value >= 0),
CONSTRAINT not-null-value CHECK(value NOT NULL));

CREATE TABLE depositor (
.....
balance bank-balance,
)

Advantages?

Attribute Constraints

Associating constraints with domains:

1. can avoid repeating specification of same constraint
   for multiple columns

2. can name constraints
   e.g.: CREATE DOMAIN bank-balance INT (
   CONSTRAINT not-overdrawn CHECK (value >= 0),
   CONSTRAINT not-null-value CHECK (value NOT NULL));

Advantages:
1. Can add or remove:
   ALTER DOMAIN bank-balance
   ADD CONSTRAINT capped CHECK(value <= 10000)
2. report better errors (know which constraint violated)