SQL (Lecture 2)

- Nested Subqueries
- Modification of the Database
- Views

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.

- The nesting can be done in the following SQL query

  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., r_m
  where P

  as follows:

  - A_i can be replaced be a subquery that generates a single value.
  - r_i can be replaced by any valid subquery
  - P can be replaced with an expression of the form:

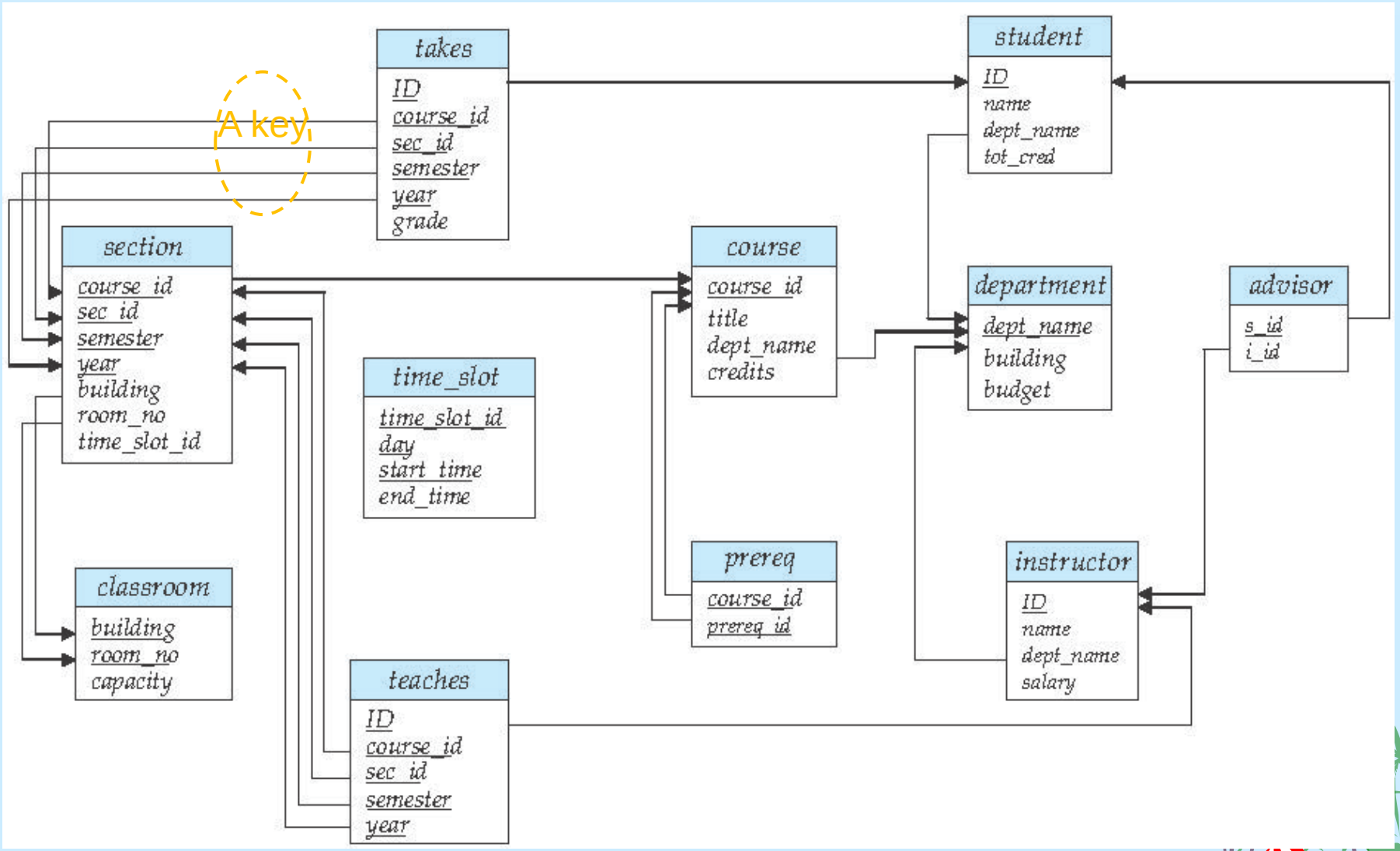
    B <operation> (subquery)

    Where B is an attribute and <operation> to be defined later.

Subqueries in the Where Clause

- The most common use of subqueries is in the where clause, which is to perform tests for
  - set membership in
  - set comparisons
  - Empty relations *most powerful subquery*
  - Absence of duplicate tuples

Schema Diagram for University Database



Set Membership

- Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2019 and
        course_id  in ( select course_id
                         from section
                         where semester = 'Spring' and year= 2020);
```

- Find courses offered in Fall 2017 but not in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2019 and
        course_id  not in ( select course_id
                            from section
                            where semester = 'Spring' and year= 2020);
```

Set Membership (Cont.)

- Name all instructors whose name is neither “Mozart” nor Einstein”

```
select distinct name
from instructor
where  name not in ('Mozart', 'Einstein')
```

- Find the total number of (distinct) students who have taken course sections taught by the instructor named “Einstein”

```
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 in
                                ( select ID
                                  from instructor
                                  where name = ‘Einstein’ )
                    );
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison – “some” Clause

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

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

- Same query using > some clause

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

Definition of Some Clause

- F <comp> **some** $r \Leftrightarrow \exists t \in r$ *s.t.* (F <comp> t)
  Where <comp> can be: $<, \leq, >, =, \neq$

(5< **some** | 0 / 5 / 6 |) = true

5 is less than some tuple in the relation

(5< **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**

However, (≠ **some**) ≡ **not in**

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 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≡ **in**

Set Comparison – “all” Clause

- 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** $r \Leftrightarrow r = \emptyset$ $\Leftrightarrow$ *P is false*

  P is the query condition to return r

Use of “exists” Clause

- Yet another way of specifying the query “Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester”

```
select course_id
from section as S
where semester = 'Fall' and year = 2017 and
        exists (select *
                from section as T
                where T.semester = 'Spring' and year= 2018
                        and T.course_id = S.course_id);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query

Powerful “not exists” Clause

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

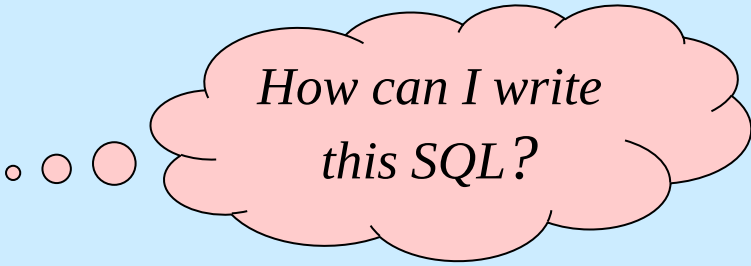
select *S.ID,S.name*
from *student* **as** *S*
where

?? ***// student S have taken all course offered in the Biology department***

Solution

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

```
select 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 T.ID = S.ID)
              );
```



How to get the solution

- Step 1: define predicates:
  - P(S) : All courses that student S has taken;
  - Q : All courses offered in Biology;
- Step 2: define the logical formula
  - Find those student S that $P(S) \supseteq Q$
  - Equivalent formula: Find S that $Q - P(S) = \varnothing$
- Step 3: SQL statement
  - Find S:
    Select S.ID, S.name from Student S
  - Q
    select course_id from course where dept_name = 'Biology'
  - P(S)
    select T.course_id from takes as T where T.ID = S.ID

Solution

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

```
select 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 T.ID = S.ID)
            );
```

To find S

Q

P(S)

Note: Cannot write this query using = all and its variants

Another solution

- Step 1: define predicates:
  - P(S,C) : student S has taken course C;
  - Q(C) : course C is offered in Biology department;
- Step 2: define the logical formula
  - Find those student S that $\forall$ C (Q(C) $\rightarrow$ P(S,C))
  - Equivalent formula: Find S that $\neg\exists$ C (Q(C) $\wedge \neg$P(S,C))
- Step 3: SQL statement
  - To find S:
    Select S.ID, S.name from Student S
  - $\neg\exists$ C Q(C)
    Not Exists Select * from course as C where dept_name = 'Biology'
  - $\neg$P(S,C): student S has NOT taken course C
    Not Exists select * from takes as T
    where T.ID = S.ID and T.course_id = C.course_id

Another solution

- Step 4:

```
Select  S.ID, S.name
from Student S
where Not exists (
        Select * from course as C
        where dept_name = 'Biology'
        And
        Not Exists (
                select * from takes as T
                where T.ID = S.ID and T.course_id = C.course_id
        )
)
```

To find S

$\neg \exists$ C Q(C)

$\neg$ P(S,C)

- Variable from outer level is known as a correlation variable

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
  - The **unique** construct evaluates to “true” if a given subquery contains no duplicates .

- Find all courses that were offered at most once in 2017

```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where R.course_id= T.course_id
                    and R.year = 2017);
```

Example

- Find all courses that were offered at lease two times in 2017

```
select T.course_id
from course as T
where not unique (select R.course_id
                  from section as R
                  where R.course_id= T.course_id
                        and R.year = 2017);
```

Subqueries in the Form Clause

- 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."

```
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 we do not need to use the **having** clause
- Another way to write above query

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

With Clause

- The **with** clause provides a way of defining a temporary relation 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 department.name
from department, max_budget
where department.budget = max_budget.value;
```

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

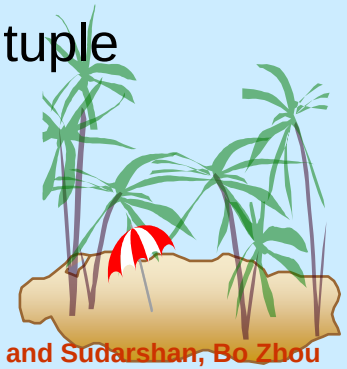
```
with dept _total (dept_name, value) as
        (select dept_name, sum(salary)
         from instructor
         group by dept_name),
dept_total_avg(value) as
       (select avg(value)
       from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,
        (select count(*)
           from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

- Runtime error if subquery returns more than one result tuple
- I never use this…



Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= ’Finance’;

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = ’Watson’);

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

  - Problem: as we delete tuples from deposit, the average salary changes
  - Solution used in SQL:
    - First, compute avg (salary) and find all tuples to delete
    - Next, delete all tuples found above (without recomputing avg or retesting the tuples)

Insertion

- Add a new tuple to *course*

```
insert into course
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- or equivalently ---Strong recommended!

```
insert into course (course_id, title, dept_name, credits)
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- Add a new tuple to *student* with *tot_creds* set to null

```
insert into student
    values ('3003', 'Green', 'Finance', null);
```

Insertion (Cont.)

- Add all instructors to the *student* relation with tot_creds set to 0

```
insert into student
    select ID, name, dept_name, 0
     from   instructor
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

  Otherwise queries like

```
insert into table1 select * from table1
```

  would cause problem

Updates

- Give a 5% salary raise to all instructors

```
update instructor
   set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who earn less than 70000

```
update instructor
   set salary = salary * 1.05
  where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average

```
update instructor
 set salary = salary * 1.05
 where salary <  (select avg (salary)
                  from instructor);
```

Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor
    set salary = case
                    when salary <= 100000 then salary * 1.05
                      else salary * 1.03
                    end
```

Updates with Scalar Subqueries

- Re-compute and update tot_creds value for all students

```
update student S
   set tot_cred = (select sum(credits)
                   from takes, course
                   where takes.course_id = course.course_id and
                         S.ID= takes.ID.and
                         takes.grade <> 'F' and
                         takes.grade is not null);
```

  - The query will set *tot_creds* to null for students who have not taken any course

- Instead of **sum**(*credits*), use:

```
case
    when sum(credits) is not null then sum(credits)
    else 0
end
```

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 instructors name and department, but not 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:

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

  where:
  - <query expression> is any legal 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 by default would NOT create a new relation in the database.
  - When a view is created, the query expression is stored in the database;
  - 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

- One view may be used in the expression defining 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.

Views Defined Using Other Views

- **create view** *physics_fall_2017* **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* = '2017';

- **create view** *physics_fall_2017_watson* **as**
  select *course_id*, *room_number*
  from *physics_fall_2017*
  where *building*= 'Watson';

View Expansion

- One view may be used in the expression defining another view.
- View expansion of an expression repeats the following replacement step:

  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

  - *As long as the view definitions are not recursive, this loop will terminate*

View Expansion

- Expand the view :

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from physics_fall_2017
    where building= 'Watson'
```

- To:

```
create view physics_fall_2017_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 = '2017')
     where building= 'Watson';
```

Update of a View

- Views are a useful tool for query, and it is an important feature to implement the logical data independence. However, it will create some difficulty to update a view.
- Add a new tuple to *faculty* view which we defined earlier

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

This insertion would be represented by the insertion of the tuple

(’30765’, ’Green’, ’Music’, null)

into the *instructor* relation

Update of a View(Cont.)

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

- What happens if we

  insert into *history_instructors* values ('25566', 'Brown', 'Biology', 100000)

  - By default, it would be allowed, however, the inserted tuple is not belong to the view. You can not find it using: select * from *history_instructors*.

- So "With Check Option" is introduced in SQL, to check the WHERE clause condition in view definition before insert a tuple to the view. If we add "With Check Option" to the end of above view definition. The insertion will be rejected.

Updates cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor*.*dept_name*= *department*.*dept_name*;

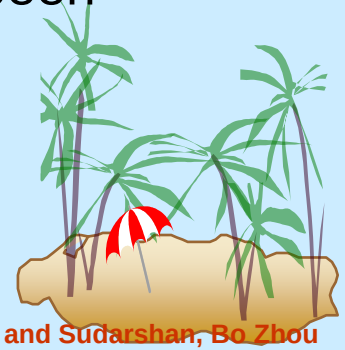
- **insert into** *instructor_info* **values** (’69987’, ’White’, ’Taylor’);
  - which department, if multiple departments in Taylor?
  - what if no department is in Taylor?

Updatable View

- Most SQL implementations allow updates only on simple views defined on a single relation
  - The FROM clause has only one relation;
  - The SELECT clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specifications;
  - Any attributes does not listed in the select clause can be set to null, or has default value;
  - The query does not have a group by or having clause.
- New SQL standard (SQL:1999) allow more views updatable, however, the rules becomes much more complex.

Materialized Views

- **Materializing a view**: create a physical table containing all the tuples in the result of the query defining the view
  - Use materialized view for frequently query over the view, and require high performance.
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.
  - Internal trigger mechanism to maintain the data consistency of materialized view.
- **Materialized view** is not defined in SQL standard, but it been implemented by most of DBMS.



The end of lecture