L05: SQL 2

DSAN 6300/PPOL 6810: Relational Databases and SQL Programming

Irina Vayndiner

September 28, 2023

Logistics

- HW1 was due on Tue
- Lab05 is due on Tue 10/3 for everyone
- Q02 and HW2 will be posted today
  - Q02 Due: Tue 10/10
  - HW2 Due: **Mon, 10/16** (with grace period till Tue, 10/17)
    - Answers to HW2 will be available as soon as everyone turns it in
      - Planned: Wed 10/18
    - 100% does **not** include Bonus
      - Not the easiest problem ☺
- No class on 10/9 (Mid Semester Holiday)
- If submitting an assignment within grace period (usually on Wed)
  - Canvas closed for submission
  - Email to me, cc Peijin, and ask her to upload
- In-Class Mid-term **closed books**: Mon, 10/23 and Thu 10/19 (2 diff versions!)
  - Additional "cleanup" DB to do ahead of midterm
  - OHs will cover:
    - **Updating** your MySQL env

Agenda for today's class

- Lecture: SQL 2
- Lab: SQL 1& 2

Outline: Proceeding with SQL

Last week: SQL 1

- Overview of The SQL Query Language
- SQL Data Definition
- Basic Query Structure of SQL Queries
- Aggregate Functions

Today: SQL 2

- Nested Subqueries in WHERE, FROM and SELECT clauses
- Modification of the Database
- Join Expressions
- 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:

  - **Where clause:** P can contain an expression of the form:

    B <operation> (subquery)

    B is an attribute and <operation> is **in, not in, all, some,** etc.
  - **From clause:** r_i can be replaced by any valid subquery
  - **Select clause:**

    A_i can be replaced by a subquery that generates a single value.

Set Membership

- The **in (**and **not in)** operator tests for set membership, where the set is a collection of values produced by a select clause.
- Example: Find courses offered in Fall 2017 and in Spring 2018 **using nested subquery**
  - We start with finding all courses taught in Spring 2018, and we write the subquery
    (**select** *course_id*
    from *section*
    where *semester* = 'Spring' **and** *year*= 2018)

- We then need to find those courses that were taught in the Fall 2017 and that appear in the set of courses obtained in the subquery.
- We do so by nesting the subquery in the where clause of an outer query.

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
        course_id in (select course_id
                      from section
                      where semester = 'Spring' and year= 2018);
```

Set Membership (continued)

- We tested membership in a one-attribute relation
  - It is also possible to test for membership in an arbitrary relation in SQL
- Example: Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 110011

```
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= '10101');
```

- That feature is not implemented in all DBMSs
- The **in** and **not in** operators can also be used on sets of constants. Example: Name all instructors whose name is neither “Mozart” nor “Einstein”

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

Set Comparison – "some" Clause

- Constructs **>some, <=some** , etc represents "greater then some", etc. semantics
- Example "Find names of instructors with salary greater than that of *some (at least one)* instructor in the Biology department."

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

- The > some comparison in the where clause of the outer select is true if the salary value of the tuple is greater than <u>at least one member</u> of the set of all salary values for instructors in Biology.
- Note: we did it before in a different way:

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

Definition of "some" Clause

- F <comp> **some** *r* ⇔ ∃ *t* ∈ *r* such that (F <comp> *t*)
  Where <comp> can be: $<, \leq, >, =, \neq$

$$(5 < \textbf{some} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$$

(read: 5 < at least one tuple in the relation)

$$(5 < \textbf{some} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 = \textbf{some} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$$

$$(5 \neq \textbf{some} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$$

- Note that (**=some**) has the same semantics as **in**
- However, (<>**some**) is different from **not in**

Set Comparison – “all” Clause

- Constructs **>all, <=all**, etc represents “greater then all”, etc. semantics
- Find the names of 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');

Definition of "all" Clause

- F <comp> **all** *r* ⇔ ∀ *t* ∈ *r* (F <comp> *t)*

$$(5 < \textbf{all} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$$

$$(5 < \textbf{all} \begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$$

$$(5 = \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 <> \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6\text{)}$$

- Note that (<>all) has the same semantics as **not in**
- However, (=all) is different from **in**

Test for Empty Relations

- SQL includes a feature for testing whether a subquery has any tuples in its result set
- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Use of "exists" Clause + correlated subqueries

- Consider again the example "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 *semester* = 'Spring' **and** *year*= 2018
  and *S*.*course_id* = *T*.*course_id*);

- This is an example of a **correlated subquery**
  - **Correlation name** – variable S in the outer query
  - Subquery uses a variable *S.course_id* that comes from the outer query.
  - Subquery is evaluated every time a **where** condition is checked in the outer query (with a different value of *S.course_id)*
  - It can be slow!

Use of "not exists" Clause

- We can test for the nonexistence of tuples in a subquery by using the **not exists** construct.
- Example: 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));
```

- Set B: First nested query lists all courses offered in Biology
- Set S: Second nested query lists all courses <u>a particular student</u> (*S.ID*) took
- We want to find students for whom B $\subseteq$ S

Test for Absence of Duplicate Tuples (*unique* construct)

- The **unique** construct
  - tests whether a subquery has any duplicate tuples in its result.
  - evaluates to “true” if a given subquery contains no duplicates or subquery is empty.
- Q: Compare to **distinct**
- 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 T.course_id= R.course_id
                      and R.year = 2017);
```

- This is how it works
  - For each row from T (T.course_id):
    - We select all sections of that course that was offered in 2017

```
select R.course_id
       from section as R
       where T.course_id= R.course_id
             and R.year = 2017
```

    - Then we apply **unique** to check if there are duplicates

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;
```

- The main query treats *max_budget* (*value*) as any other relation
  - In this example it has one row (and one attribute), that is a maximum budget of all departments
  - Then the main query selects all departments that have that value of a budget attribute

Complex Queries using With Clause

- Main motivation to use **with** clause is to improve readability of queries
- It is important for really complicated queries
- **with** clause permits this temporary relation to be used in multiple places within a query
- Example: Find all departments where the total salary is greater than the average 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 the one that returns **only one tuple** containing a single attribute (a scalar)
- It can be used wherever an expression returning a value is permitted, in particular in **select, where,** and **having** clauses
- Example: 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

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

- Note: You may want reload University database data (DML) after you have done any of these type of statements
  - Definitely for HWs and Labs and Tests
  - Attend OHs to reload University Database if you do not know how

Deletion

- We can delete only whole tuples; you cannot delete values on only particular attributes. SQL expresses a deletion by:

  delete from *r*
  where *P*;

  where P represents a predicate and r represents a relation.
- The **delete** statement first finds all tuples t in r for which P(t) is true, and then deletes them from r.
- A delete command operates on only **one** relation

Deletion (continued)

- The where clause can be omitted, in which case **<u>all</u>** tuples in r are deleted. (*Safe mode in mysql prevents that*)
  Delete all instructors:

  delete from *instructor*

- Delete all instructors from the Finance department

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

- Delete instructors associated with departments located in the Watson building.

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

Deletion (continued)

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

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

  - Possible problem: as we delete tuples from *instructor*, the average salary changes
  - Solution used in some DBMSs (not supported in some others):
    - First, compute **avg** (salary) and find all tuples to delete
    - Next, delete all tuples found in the step 1 (without recomputing **avg** or retesting the tuples)

Insertion

- To insert data into a relation, you either specify
- Method 1) a tuple to be inserted

  or
- Method 2) write a query whose result is a set of tuples to be inserted.
- Method 1): Add a new tuple to *course*

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

  - Important: Attributes should be in the same order as listed in the schema
- Or attributes can be explicitly specified

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

- **null** is a valid value (if nulls are allowed): add a new tuple to *student* with *tot_creds* set to null

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

Insertion (continued)

- 2nd method is to insert tuples on the basis of the result of a query
- Example: Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

  insert into *instructor*
  select *ID, name, dept_name, 18000*
  from *student*
  where *dept_name* = 'Music' **and** *total_cred* > 144;

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

  insert into *table*1 **select** * **from** *table*1

  would cause problem (looping)

  Note: Many RDBMS do not allow that
- 3rd method: Most relational database products have special "bulk loader" utilities to insert a large set of tuples into a relation
  - Much faster!

Updates

- The **update** statement allows to change a value in a tuple without changing all values in that tuple.
- Give a 5% salary raise to all instructors

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

- May have a **where** clause, that contains any legal construct in the where clause of the select statement (including nested selects).
- Example: Give a 5% salary raise to those instructors who earn less than 70000

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

- As with insert and delete, a nested select within an update statement may reference the relation that is being updated.
- SQL first tests all tuples in the relation to see whether they should be updated, and it carries out the updates afterward.
  - Example: 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*);

  - In many RDBMS is not supported!

Case Construct for Conditional 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 of updates is important here
- Can be written as one **update** with **case** construct

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

General Form of Case Statement

- The general form of the case statement is as follows:

  case
  when $pred_1$ **then** $result_1$
  when $pred_2$ **then** $result_2$
  ...
  when $pred_n$ **then** $result_n$
  else $result_0$
  end

- The operation returns $result_i$, where i is the first of $pred_1$, $pred_2$,..., $pred_n$ that is satisfied; if none of the predicates is satisfied, the operation returns $result_0$.
- Do not forget **else**!
  - Otherwise it might return null

Updates with Scalar Subqueries

- Scalar subqueries are useful in SQL update statements, where they can be used in the set clause.
- Example: Recompute and update tot_creds value for all students (we assume that a course is successfully completed if the student has a grade that is neither 'F' nor null.)

```
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);
```

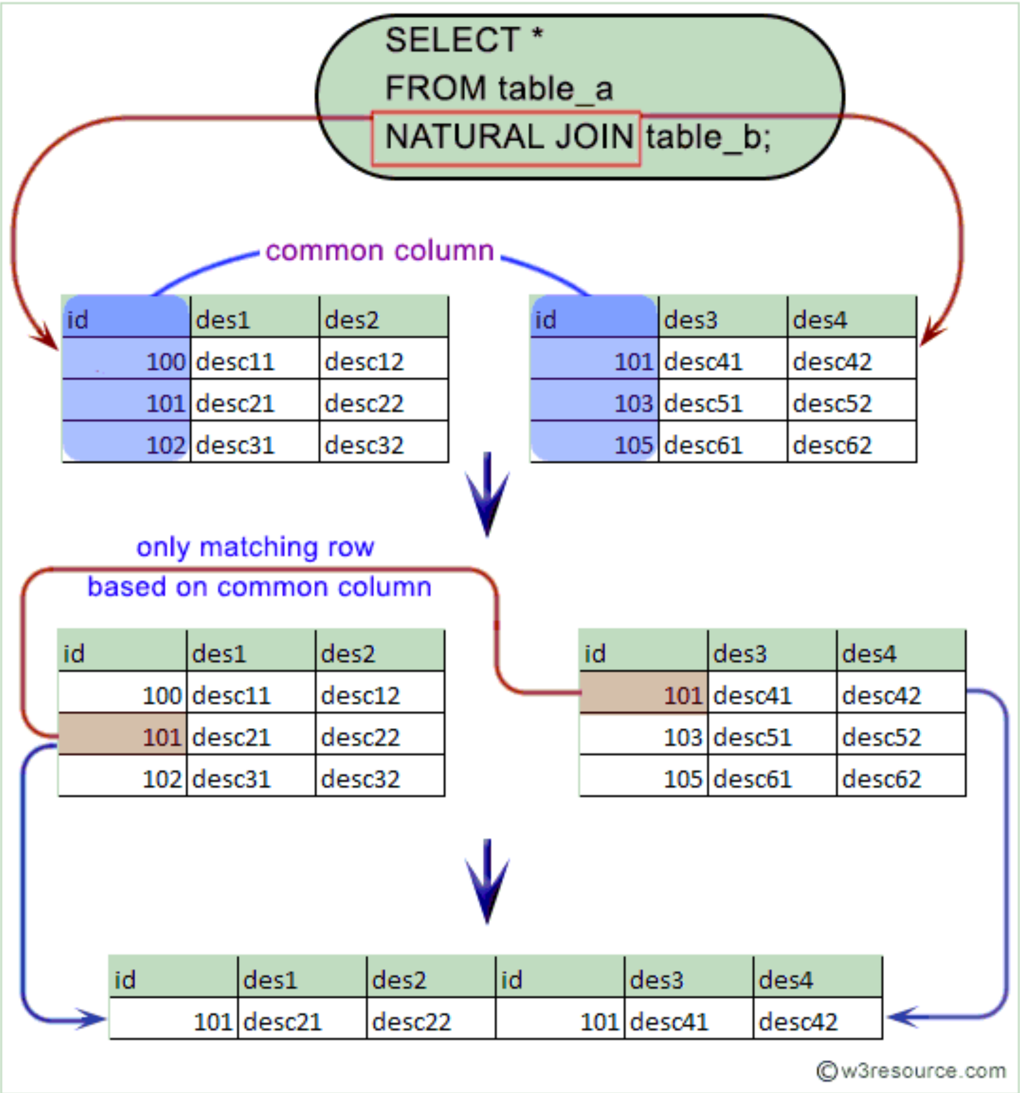
- That will set *tot_creds* to null for students who have not taken any course
- If that is not desirable, then instead of **sum**(*credits*), we can use:

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

Joined Relations

- **Join operations** take two or more relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used in the **from** clause
- Three types of joins (not mutually exclusive):
  - **Natural** join
  - **Inner** join
  - **Outer** join

Natural Join in SQL



Natural join matches tuples with the same values for *all common attributes*, and retains only one copy of each common column.

Example: List the names of students along with the course ID of the courses that they took

- **select** *name*, *course_id*
  from *students*, *takes*
  where *student.ID* = *takes.ID*;

Same query rewritten in SQL with "natural join" construct

- **select** *name*, *course_id*
  from *student* **natural join** *takes*;

General Form of Natural Join in SQL

- The **from** clause can have multiple relations combined using natural join:

  select $A_1, A_2, \ldots A_n$
  from r_1 **natural join** r_2 **natural join .. natural join** r_n
  where P ;

- Natural joins can be combined with other relations in the from clause

  from E_1 , E_2 **, ...** E_n

  where each E_i can be a single relation or an expression involving natural joins.

Student Relation

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Takes Relation

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

student **natural join** *takes*

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	*null*

Pitfalls in Natural Join

- Beware of unrelated attributes (to your specific query) with same name in diff tables!
- Example: List the names of students along with the titles of courses that they have taken
  - Correct version

    select *name*, *title*
    from ***student*** **natural join** ***takes***, *course*
    where *takes*.*course_id* = *course*.*course_id*;

  - student **natural join** takes (ID, name, dept_name, tot_cred, course_id, sec_id)
  - course (course_id, title, dept_name, credits)
    - Has course's dept name (not student's dept name!)
  - Incorrect version

    select *name*, *title*
    from *student* **natural join** *takes* **natural join** *course*;

    - This query implicitly enforces student.dept_name=course.dept_name and thus filters out all (student name, course title) pairs where the student *takes a course in a department other than the student's own department.*
    - The correct version (above), correctly outputs such pairs.

Natural Join with Using Clause

- To avoid using *unrelated* attributes in a natural loin, we can use the "**using**" construct that allows us to specify exactly which columns should be equated.
- r_1 **join** r_2 **using** (A_1, …, A_n).
  - Same as natural join, but match only by A_1, …, A_n
- Works well for the previous example

  select *name*, *title*
  from (*student* **natural join** *takes*) **join** *course* **using** (*course_id*)

equivalent to

  select *name*, *title*
  from *student* **natural join** *takes*, *course*
  where *takes*.*course_id* = *course*.*course_id*;

Join ON Condition

- To generalize a natural join we use a construct

  r_1 **join** r_2 **on** *P*

- The **on** condition allows a general predicate *P* over the relations being joined
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**
- Query example

  select *
  from *student* **join** *takes* **on** *student.ID* = *takes.ID*

  - The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.

- Equivalent to:

  select *
  from *student* , *takes*
  where *student.ID* = *takes.ID*

Join Condition (continued)

- As in the previous example, any query using a join expression with an **on** condition can be replaced by an equivalent expression with the predicate in the on clause moved to the **where** clause.
- Why use **on** condition?
  - A SQL query is often more readable by humans
    - the join condition is specified in the **on** clause
    - the rest of the conditions appear in the where clause

select *
from *student* **join** *takes* **on** *student.ID* = *takes.ID*
where grade = ‘A’

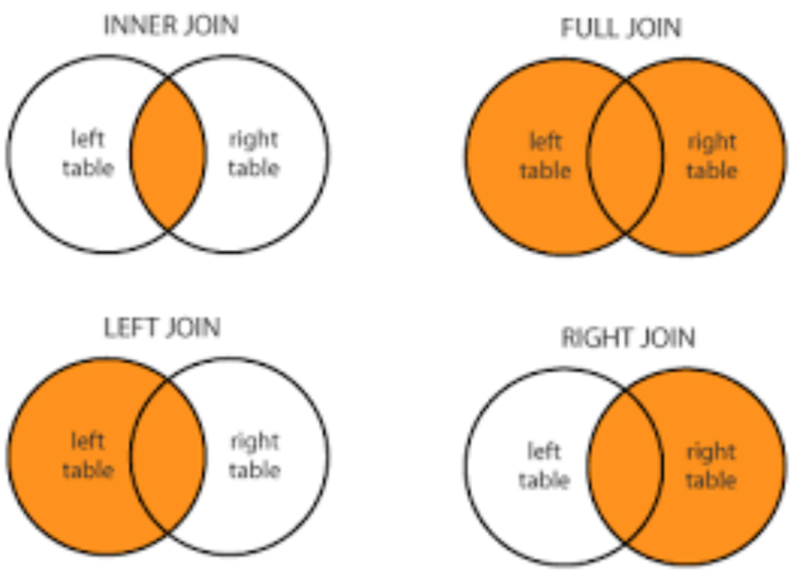
Outer Join

- An extension of the join operation that deals with missing information.
- Say, we want a list of all students, along with the courses that they have taken

  select *

  from *student* **natural join** *takes*;

- Problem: students who took *no courses* will not be in the result set at all
  - Outer joins – designed to fix that problems



- Three forms of outer joins:
  - **left outer** join
  - **right outer** join
  - **full outer** join

Outer Join Examples

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

  course information is missing for CS-347

  prereq information is missing for CS-315

Left Outer Join

- Outer join:
  - Computes the regular (or inner) join
  - Then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
  - Fills in missing data with null values.
- **Left** outer join preserves tuples in the relation named ***before*** (**to the left of**) the join operation.

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- **select** * **from** *course* **natural left outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Right Outer Join

- Outer join:
  - Computes the regular (or inner) join
  - Then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
  - Fills in missing data with null values.
- **Right** outer join preserves tuples in the relation named ***after*** (**to the right of**) the join operation.

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- **select** * **from** *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Full Outer Join

- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Fill in missing data with null values.
- Full outer join preserves tuples in both relations

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- **select** * **from** *course* **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Summary: Joined Types and Conditions

- **Join operations** take two or more relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

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 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.
- Conceptually a view contains the tuples in the query result
- 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.
  - <u>Whenever the view relation is accessed, its tuples are actually calculated</u>

View Definition and Use

- 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 using view we just created

  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 Can be Defined Using Other Views

- View names may appear in a query any place where a relation name may appear
- 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

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';

- The following view depends directly on the view ***physics_fall_2017*** :

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

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';
```

View Expansion (continued)

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- 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

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy is created when the view is defined.
  - Such views are called **Materialized view**
- 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.
  - This is done automatically by DBMS
- The main benefit to materialize a view is performance.
- Key considerations:
  - 1. How often the view accessed
  - 2. How often the view needs to be re-evaluated
- Not the same as a regular table
- The benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.

Update of a View

- Views are very intuitive for select queries, but if used in update, insert or delete statements may cause problems
- Consider a view:

  create view ***faculty*** **as**
  select *ID*, *name*, *dept_name*
  from *instructor*

- Add a new tuple to *faculty* view

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

- This insertion must be implemented by the insertion into the *instructor* relation
  - Must have a value for salary
  - Two approaches
    1. Reject the insert into instructor

    or

    2. Inset the tuple

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

       into the *instructor* relation if nulls are allowed

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');

- **Issues**
  - Which department, if multiple departments are in the building called Taylor?
  - What if no department is in building called Taylor?

And Some Not at All

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= 'History';
- What happens if we insert

  ('25566', 'Brown', 'Biology', 100000)

  into *history_instructors?*

View Updates in SQL: Limitations

- Because of issues, such as those we just discussed, modifications are generally not permitted on view relations, except in limited cases.
- Most SQL implementations allow updates only on **updatable** views, that is when all of the following conditions are satisfied by the query defining the view:
  - 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.

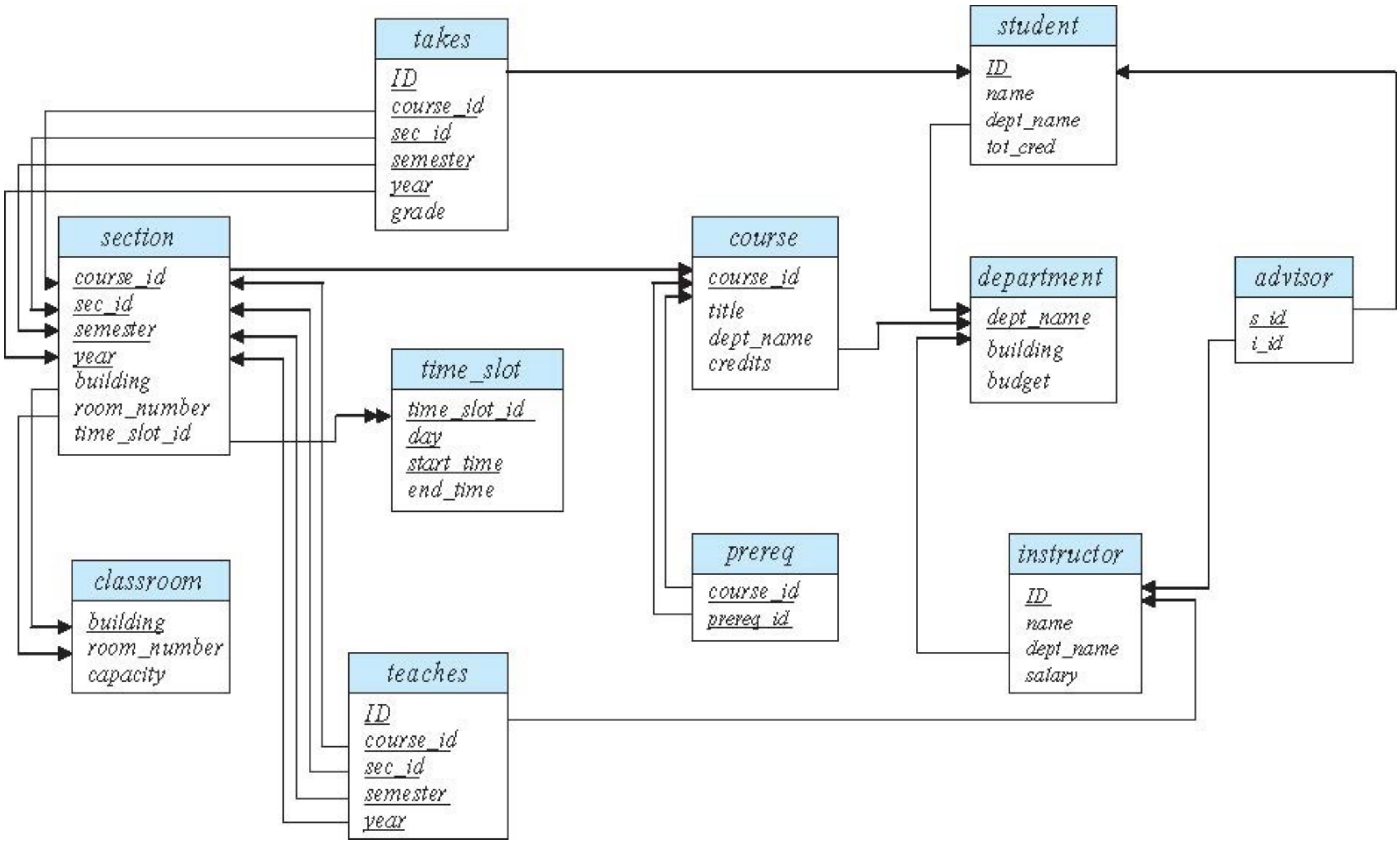
Your MySQL Environment

- *Updating your MySQL environment:*
  - Update "University" Database: Both DDL and DML
  - Needed if you ran any **drop**, **delete** or **update** statements
    - To get good grades for HWs, Tests ☺
  - See Canvas for DDL, DML
    - Run DDL first
    - Then reload DML
  - OHs can go through these steps

University Schema is on Canvas

  - Reference for MySQL syntax: https://dev.mysql.com/doc/refman/8.0/en/

Use in Labs: Schema Diagram for University Database: On Canvas



“What if my Query is Different?” How to check

1) Quick and dirty check
Compare counts
select count (*) **from** (Query1)
and
select count (*) **from** (Query2)

2) Export the results sets as CSV files and use diff

3) Run SQL statement that will compare results of Query 1 and Query 2, for example (ID is some <u>unique set of attributes</u>, e.g. primary key)

select ID **from** (
    select ID **from** . . . /* Query 1 */
    union all
    select ID **from** …   /* Query 2 */
) as tbl
group by ID
having count(*) = 1;

To submit for the Lab05: problems #3 and #4

Use MySQL to run the queries

To Submit

- **The problem formulation & your query**
- **Your Result set**

Format:

- **For SQL file,** upload individual .sql file with problem formulation commented out BEFORE your sql statement, and # of rows commented out
- **For result set,** upload individual .cvs files

File names:

- **Firstname_lastname_net_id_Lab05_problem03.sql**
- **Firstname_lastname_net_id_Lab05_problem04.sql**
- **Firstname_lastname_net_id_Lab05_problem03_results.cvs**
- **Firstname_lastname_net_id_Lab05_problem04_results.cvs**

THERE IS MORE THAN ONE WAY TO WRITE A QUERY!