DSA5104
Principles of Data Management and Retrieval

Lecture 3: SQL

Recap

- Database Systems
  - Data abstraction
  - Physical data Independence
  - Database languages
  - Transactions (ACID)
- Relational Model
  - Relation, table, tuple, attribute
  - Domain, null values
  - Keys, primary/foreign key constraints
  - Relational algebra

Recap

- SQL Parts
  - DDL, DML, Integrity constraints
- SQL Data Definition
  - CREATE TABLE (integrity constraints), Domain types
- Basic Query Structure of SQL Queries
  - SELECT, FROM, WHERE
- Additional Basic Operations
  - rename, string, ORDER BY
- Set Operations
- Null Values
  - Result of arithmetic expression -> Null
  - Result of comparison / boolean operation -> Unknown
  - WHERE clause
- Aggregate Functions
  - GROUP BY, HAVING
- Nested Subqueries
  - Where can a nesting query be used?

Aggregation with Null Values

- Rule - All aggregate functions **except count (*)** ignore null values in their input collection.

- As a result of null values being ignored, the collection of values may be empty.
  - The count of an empty collection is defined to be **0**
  - All other aggregate operations return a value of **null** when applied on an empty collection.

- count(*expr*) - returns the number of rows **where *expr* is not null**.
  - You can count either all rows, or only distinct values of expr.
- count(*) - returns all rows, **including duplicates and nulls**.

- count never returns null.

Aggregation with Null Values (Cont.)

- SQL

  alter table *relation* **add** *column_name D*;

  select count(*column_name***) from** *relation*

  where *column_name* **is null;**

- What is the result of the above query?

Aggregation with Null Values (Cont.)

- SQL

  alter table *relation* **add** *column_name D*;

  select count(*column_name***) from** *relation*

  where *column_name* **is null;**

- What is the result of the above query?

- "Find the number of rows with column_name's value being null"

P

ART ONE

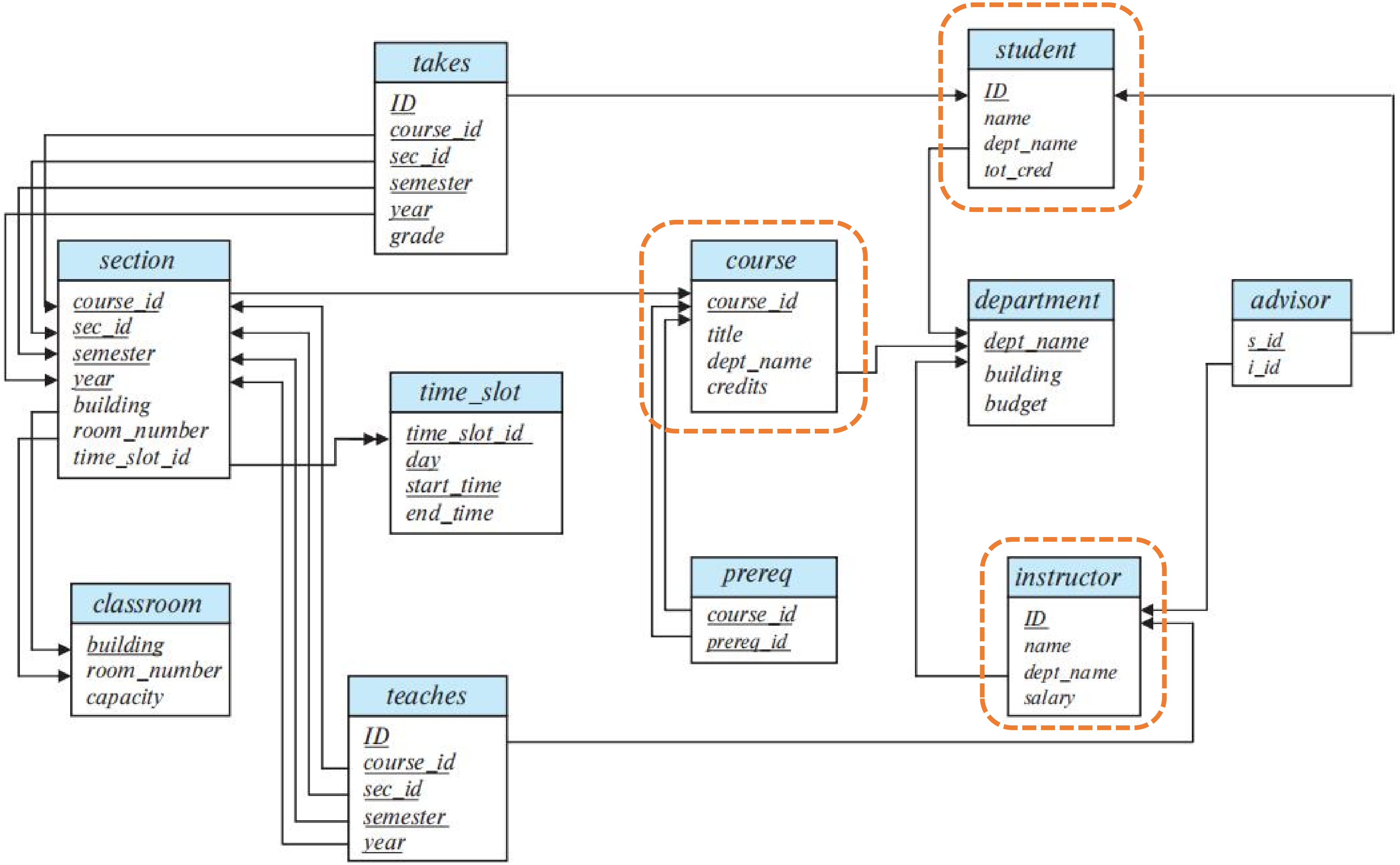
SQL

01

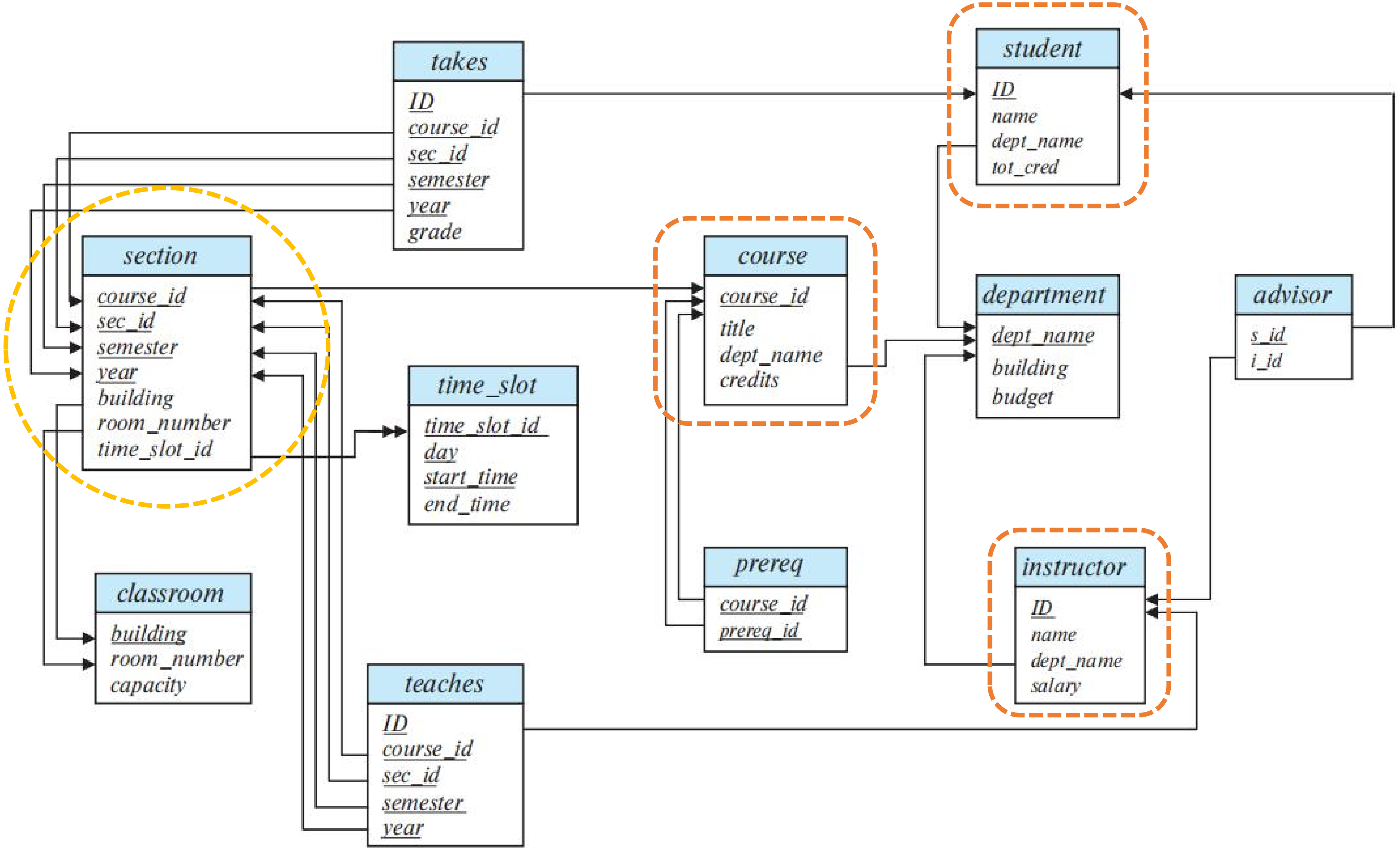
Outline

- Nested Subqueries
- Modification of the Database
- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

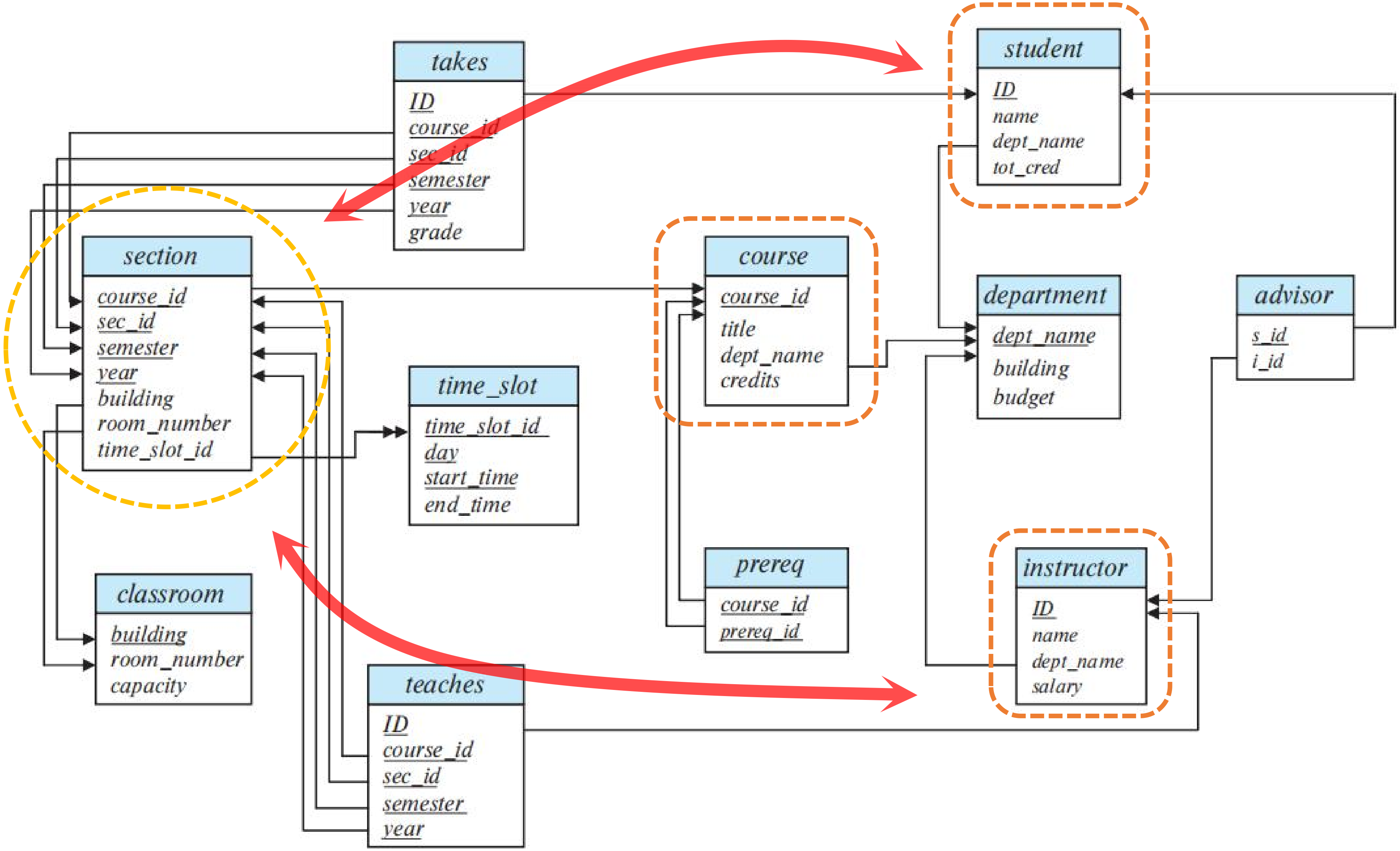
Schema Diagram for University Database



Schema Diagram for University Database



Schema Diagram for University Database



The *Section* Relation

- Each course in a university may be offered multiple times, across different semesters, or even within a semester.
- *Section* - a relation to describe each individual offering, or section, of the class.
- Instance

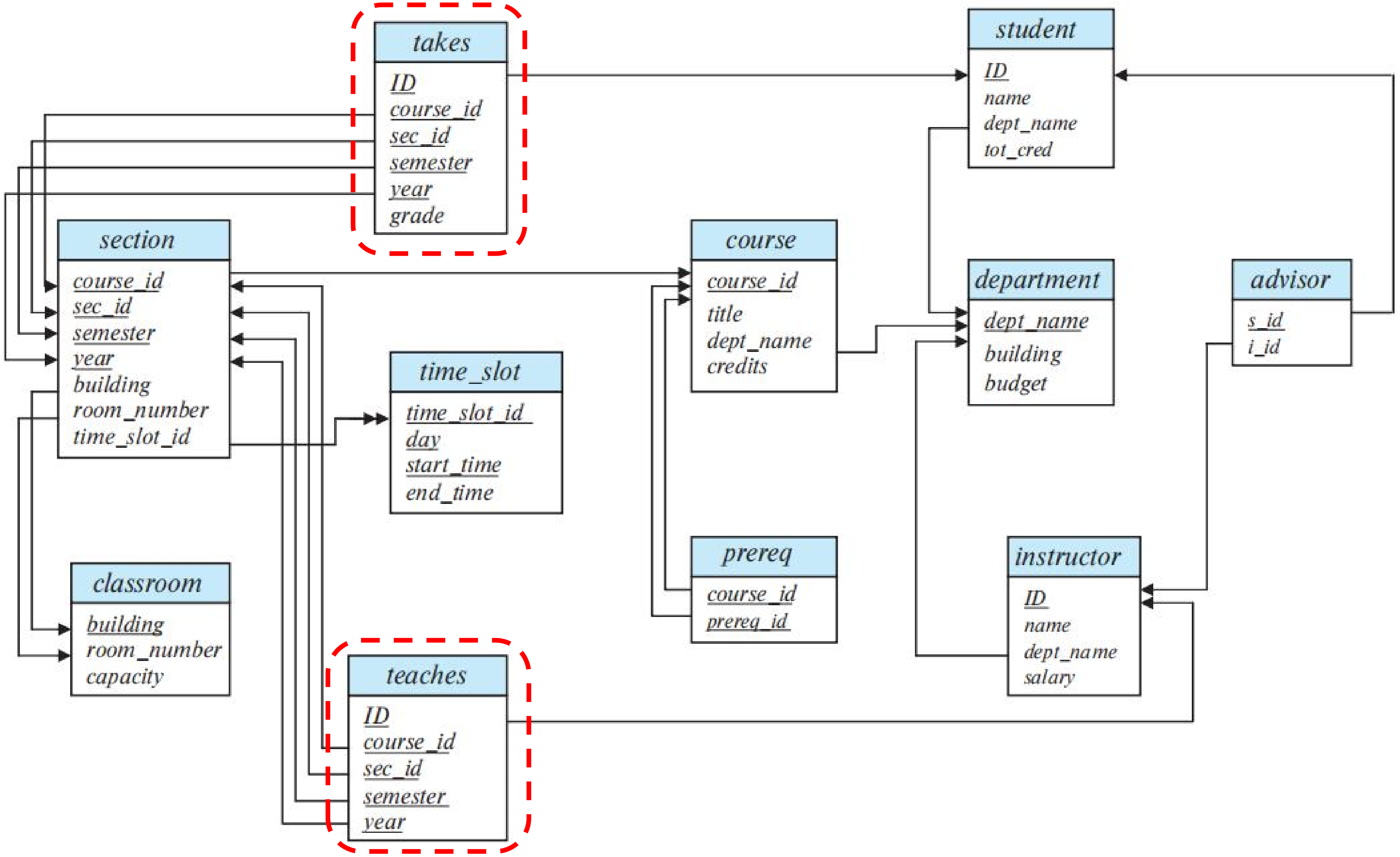
Schema
section (course_id, sec_id, semester, year, building, room number, time slot id)

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

The *takes* & *teaches* Relations

- *student* (<u>ID</u>, name, dept_name, tot_cred)
- *instructor* (<u>ID</u>, name, dept_name, salary)
- *section* (<u>course_id</u>, <u>sec_id</u>, <u>semester</u>, <u>year</u>, building, room number, time slot id)
- *takes* (<u>ID</u>, <u>course_id</u>, <u>sec_id</u>, <u>semester</u>, <u>year</u>, grade)
  - Integrity constraints: FK to *PK* of *student, FK to PK* of *section*
- *teaches* (<u>ID</u>, <u>course_id</u>, <u>sec_id</u>, <u>semester</u>, <u>year</u>)
  - Integrity constraints: FK to *PK* of *instructor, FK to PK* of *section*

Schema Diagram for University Database



Test for Empty Relations

- *Purpose*: for testing whether a subquery has any tuples in its result.
- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- Let *r* denote the result relation of a subquery
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Test for Empty Relations (Cont.)

- "Find courses offered in Fall 2017 and in Spring 2018"
- SQL 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);
```

- Relational Algebra:

$$\Pi_{course\_id}\,(\sigma_{semester=\text{"Fall"}\ \wedge\ year=2017}\,(section)) \cap$$
$$\Pi_{course\_id}\,(\sigma_{semester=\text{"Spring"}\ \wedge\ year=2018}\,(section))$$

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 semester = 'Spring' and year= 2018
                      and S.course_id = T.course_id);
```

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

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"

Outer Query

```
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);
```

Inner Query

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

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"

Outer Query

Correlation Name: S

```
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);
```

Inner Query

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

Scoping Rule for Correlation Name

- In a subquery, according to the rule, it is legal to use only correlation names defined in the subquery itself or in any query that contains the subquery.

- If a correlation name is defined both locally in a subquery and globally in a containing query, the local definition applies.
  - Analogous to the usual scoping rules used for variables in programming languages.

How to Write this Query?

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

Use of “not exists” Clause

- *“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) );
```

Use of “not exists” Clause

- *“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) );
```

First nested query lists all courses offered in Biology

Second nested query lists all courses a particular student took

Use of "not exists" Clause

- *"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) );
```

First nested query lists all courses offered in Biology

Second nested query lists all courses a particular student took

- Note that $X - Y = \varnothing \iff X \subseteq Y$
- Note: Cannot write this query using = all and its variants

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”*

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 T.course_id= R.course_id
                     and R.year = 2017);
```

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 T.course_id= R.course_id
                     and R.year = 2017);
```

- If $r = \emptyset$, **unique** r = *true* or *false?*

Test for Existence of Duplicate Tuples

- The **not unique** construct tests the existence of duplicate tuples in a subquery.

- *"Find all courses that were offered at least twice in 2017"*

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

“unique” Test with Null Values

- The **unique** test on a relation is defined to **fail** if and only if the relation contains two distinct tuples t_1 and t_2 such that $t_1 = t_2$.

- If any of the attributes of t_1 or t_2 are null, will the test $t_1 = t_2$ fail or not?

“unique” Test with Null Values

- The **unique** test on a relation is defined to **fail** if and only if the relation contains two distinct tuples t_1 and t_2 such that $t_1 = t_2$.

- If one of the attributes of t_1 or t_2 is null, the test $t_1 = t_2$ will fail or not?

- It is possible for unique to be true even if there are multiple copies of a tuple, as long as at least one of the attributes of the tuple is null.

Subqueries in the From 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;
```

Subqueries in the From 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

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

Subqueries in the From Clause (Cont.)

- 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."*
- 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;

We give the ***subquery result relation*** a name, and rename the attributes, using the as clause.

Subqueries in the From Clause (Cont.)

- *"Find the maximum across all departments of the total of all instructors' salaries in each department."*

  select max (*tot_salary*)
  from (**select** *dept_name*, **sum**(*salary*)
  　　　from *instructor*
  　　　group by *dept_name*) **as** *dept_total* (*dept_name*, *tot_salary*);

  - The **having** clause does not help in this task

“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 Subqueries

- 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 occurs if subquery returns more than one result tuple

Scalar without a "from" Clause

- Certain queries require a calculation but no reference to any relation.
- "*find the average number of sections taught (regardless of year or semester) per instructor*"

(**select count** (*) **from** *teaches*) / (**select count** (*) **from** *instructor*);

select (**select count** (*) **from** *teaches*) / (**select count** (*) **from** *instructor*);
from *dual*;

Oracle

Outline

- Nested Subqueries
- Modification of the Database
- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

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 *instructor*, the average salary changes
- Solution used in SQL:
  1. First, compute **avg** (salary) and find all tuples to delete
  2. 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);

  - The *attribute values* for inserted tuples must be members of the corresponding attribute's **domain**.
  - Tuples inserted must have the correct number of attributes.

Insertion

- Add a new tuple to *course*

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

  - The *attribute values* for inserted tuples must be members of the corresponding attribute's **domain**.
  - Tuples inserted must have the correct number of attributes.

- or equivalently

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

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

Specify the attributes in random order as part of the insert statement.

Insertion (Cont.)

- 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;

  Insert tuples on the basis of the result of a query

- The **select** 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

Insertion (Cont.)

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

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

- Insert a large set of tuples into a relation by reading from formatted text files

```
CREATE TABLE discounts (
    id INT NOT NULL AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    expired_date DATE NOT NULL,
    amount DECIMAL(10 , 2 ) NULL,
    PRIMARY KEY (id)
);
```

discounts.csv

```
id,title,expired date,amount
1,"Spring Break 2014",20140401,20
2,"Back to School 2014",20140901,25
3,"Summer 2014",20140825,10
```

```
LOAD DATA INFILE 'c:/tmp/discounts.csv'
INTO TABLE discounts
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

https://www.mysqltutorial.org/**import-csv-file-mysql-table**/

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

- 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*);

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

Updates (Cont.)

- 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

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

- The general form of the case statement

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

Updates with Scalar Subqueries

- Recompute 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);
```

Updates with Scalar Subqueries

- Recompute 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);
```

- *tot_creds* is set to null for students who have not taken any course

- Instead of **select sum**(*credits*), use (set *tot_creds* to 0)s:

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

Outline

- Nested Subqueries
- Modification of the Database
- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

Joined Relations

- **Join operations** take two 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 as subquery expressions in the **from** clause
- Three types of joins:
  - Natural join
  - Inner join
  - Outer join

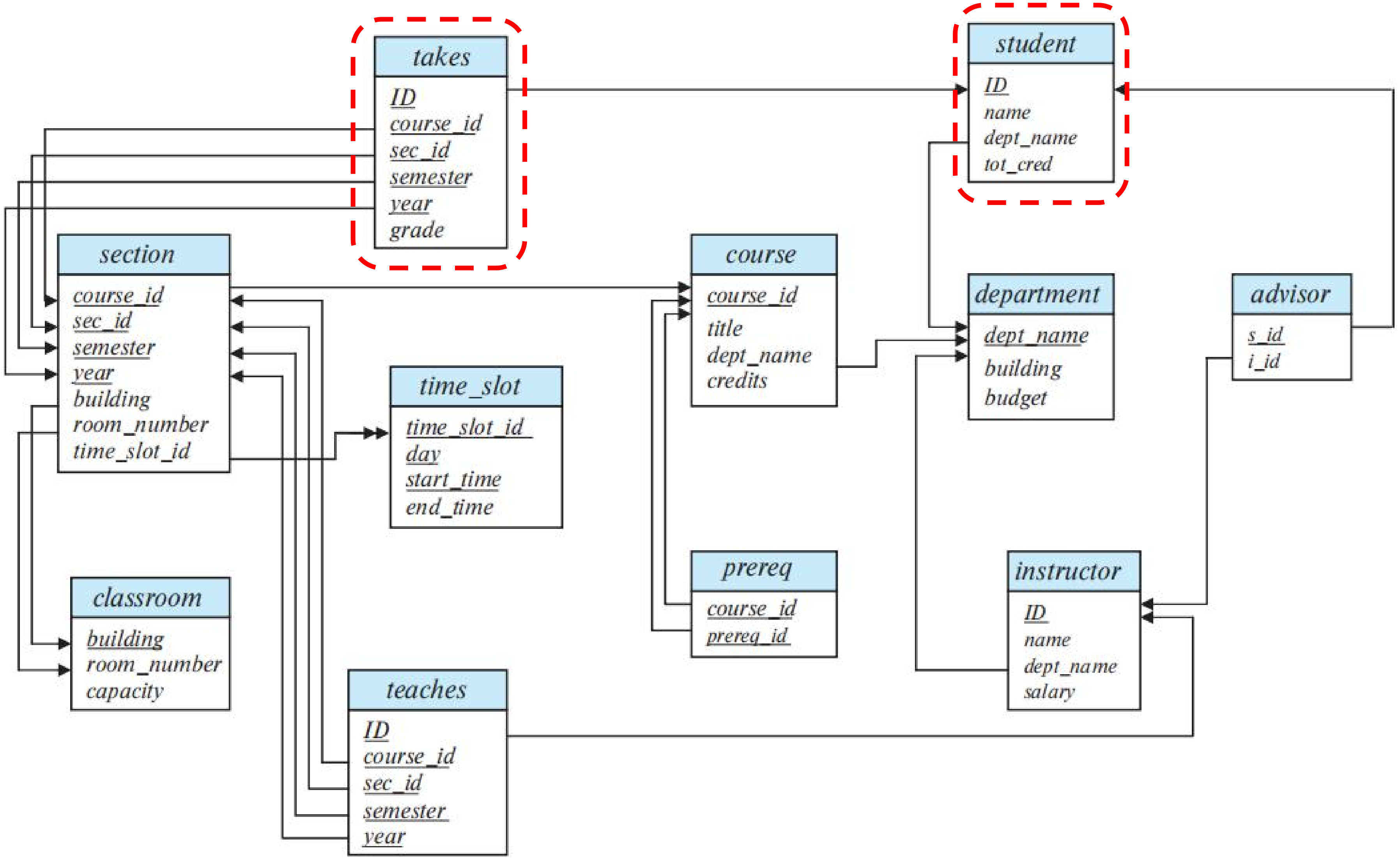
Student Relation

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*

Schema Diagram for University Database



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***.

- *"List the names of students along with the course ID of the courses that they have taken"*
  - **select** *name*, *course_id*
    from *students, takes*
    where *student.ID* = *takes.ID*;

- Same query in SQL with "natural join" construct
  - **select** *name*, *course_id*
    from *student* **natural join** *takes*;

Natural Join in SQL (Cont.)

- 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 ;

student natural join *takes*

The **Cartesian product** of two relations **concatenates** each tuple of the first relation with every tuple of the second

The **natural join** of two relations does not repeat common attributes. Only one copy is kept.

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*

student natural join *takes*

The **Cartesian product** of two relations **concatenates** each tuple of the first relation with every tuple of the second

The **natural join** of two relations does not repeat common attributes. Only one copy is kept.

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*

Attributes in Natural Join Result

Common Attributes

Unique Attributes of First Relation

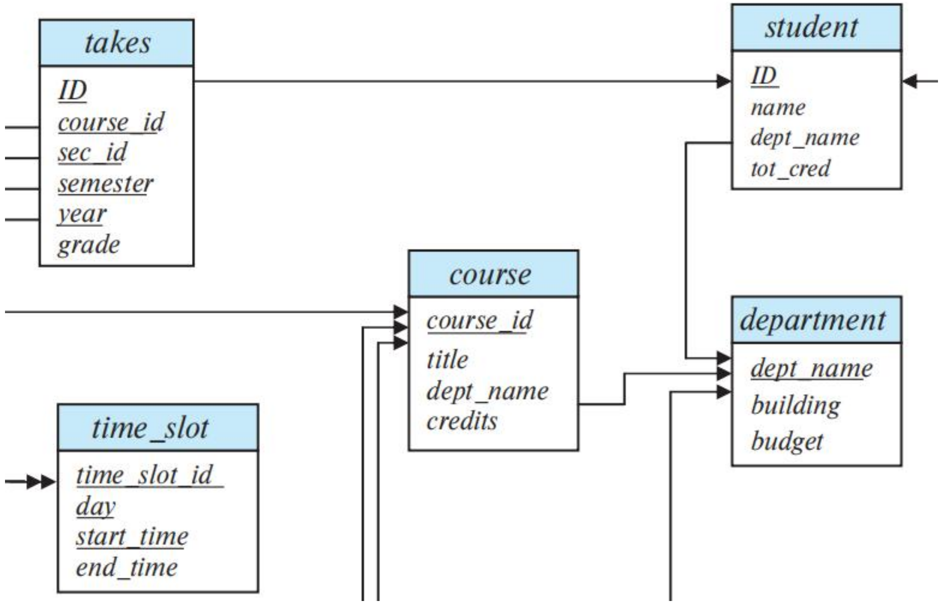
Unique Attributes of Second Relation

Dangerous in Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly

- *"List the names of students along with the titles of courses that they have taken"*

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



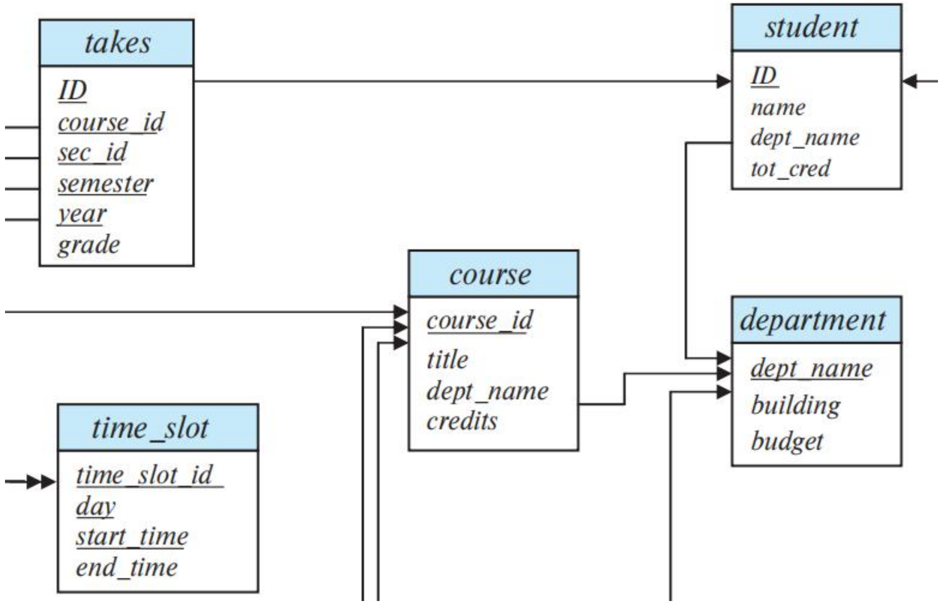
Dangerous in Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly

- *"List the names of students along with the titles of courses that they have taken"*

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

  - Natural join: *student.dept_name* = *course.dept_name*



Dangerous in Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly

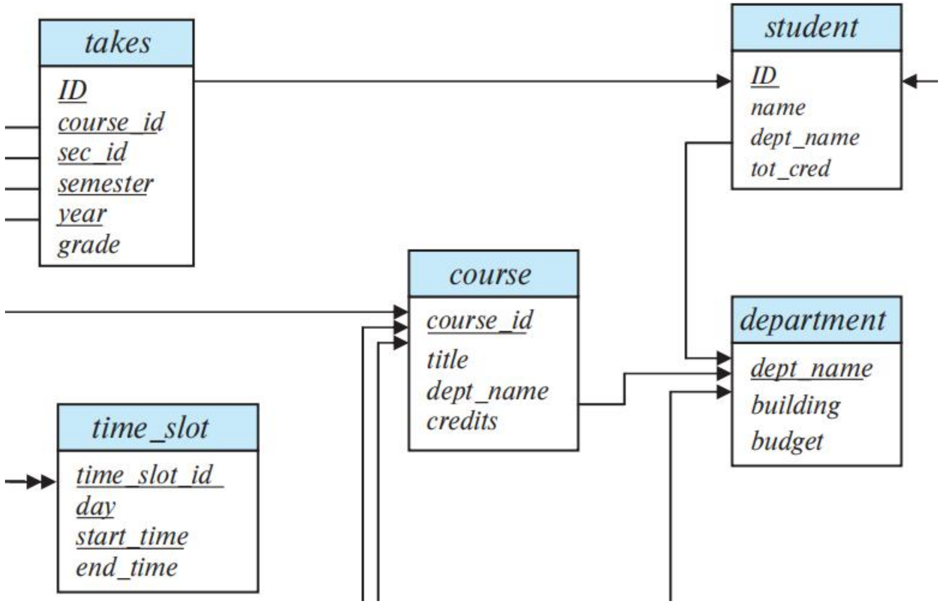
- *"List the names of students along with the titles of courses that they have taken"*

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

  - Natural join: *student.dept_name* = *course.dept_name*

  - Correct version

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



Natural Join with Using Clause

- To avoid the danger of equating attributes erroneously, we can use the "**using**" construct that allows us to specify exactly which columns should be equated.

- *"List the names of students along with the titles of courses that they have taken"*

select *name*, *title*

from (*student* **natural join** *takes*) **join** *course* **using** (*course_id*);

Join Conditions

- The **on** condition allows a general predicate 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 Conditions

- The **on** condition allows a general predicate 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*

  What is the difference between this query and natural join?

  - 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 Conditions

- The **on** condition allows a general predicate 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.

What is the difference between this query and natural join?

*The **ID** attributes are listed twice, in the join result.*

- **Equivalent to:**

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

select *student.ID* **as** *ID, name, dept name, tot cred, course id, sec id, semester, year, grade*
from *student* **join** *takes* **on** *student.ID* = *takes.ID*;

Why Outer Join?

- *"Display a list of all students, displaying all their information, along with the courses that they have taken."*

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

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

student

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*

student **natural join** *takes*

Why Outer Join?

- *"Display a list of all students, displaying all their information, along with the courses that they have taken."*

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

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

"Lost"

student

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*

student **natural join** *takes*

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.
- *vs.* inner join (normal join)
- Three forms of outer join:
  - left outer join
  - right outer join
  - full outer join

Outer Join Examples

- *"Display a list of all students, displaying all their information, along with the courses that they have taken."*

select *
from *student* **natural left outer 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-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
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*

Outer Join Examples

- *"Display a list of all students, displaying all their information, along with the courses that they have taken."*

  select *
  from *student* **natural left outer join** *takes*;

- *"Find all students who have not taken a course"*

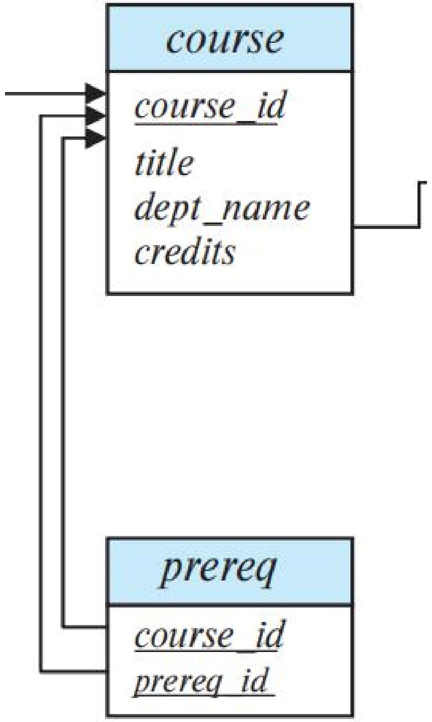
  select *
  from *student* **natural left outer join** *takes*
  where *course_id* **is null;**

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-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
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*

Outer Join More 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 CS-347 (*FK constraint is not satisfied!*)

  prereq information is missing CS-315

Left Outer Join

- *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*

- In relational algebra: *course* ⟕ *prereq*

Right Outer Join

- *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

- In relational algebra: *course* ⟖ *prereq*

Full Outer Join

- *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

- In relational algebra: *course* ⟗ *prereq*

Outer Join using on Clause

- *“Display a list of all students, displaying all their information, along with the courses that they have taken.”*

  select *
  from *student* **natural left outer join** *takes* **on** *student.ID* = *takes.ID*;

  select *
  from *student* **natural left outer join** *takes*;

- What is the different between these two queries?

Outer Join using on/where Clause

- *"Display a list of all students, displaying all their information, along with the courses that they have taken."*

  select *
  from *student* **natural left outer join** *takes* **on** *student.ID = takes.ID*;

  select *
  from *student* **natural left outer join** *takes* **on** *true*
  where *student.ID = takes.ID*;

- What is the different between these two queries?

Outer Join using on/where Clause

- *"Display a list of all students, displaying all their information, along with the courses that they have taken."*

  select *
  from *student* **natural left outer join** *takes* **on** *student.ID* = *takes.ID*;

  select *
  from *student* **natural left outer join** *takes* **on** *true*
  where *student.ID* = *takes.ID*;

The on *condition is part of the outer join specification, but a* where *clause is not.*

- What is the different between these two queries?

- (*70557, Snow, Physics, 0, null, null, null, null, null, null*) is in the result of the first query, but not the second

Join Types and Conditions

- **Join operations** take two relations and return as a result another relation.

- **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 $(A_1, A_2, \ldots, A_n)$

  - The default join type, when the **join** clause is used without the **outer** prefix, is the **inner join**.
  - Similarly, **natural join** is equivalent to **natural inner join**.

Joined Relations – Examples

- *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

- *course* **full outer join** *prereq* **using** (*course_id*)

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

Joined Relations – Examples

- *course* **inner join** *prereq* **on** *course.course_id* = *prereq.course_id*

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

- What is the difference between the above, and a natural join?

- *course* **left outer join** *prereq* **on** *course.course_id* = *prereq.course_id*

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

Joined Relations – Examples

- *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

- *course* **full outer join** *prereq* **using** (*course_id*)

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

Outline

- Nested Subqueries
- Modification of the Database
- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

Why Views?

- In some cases, it is not desirable for all users to see the entire ***logical model***
  - That is, to hide the actual relations stored in the database for **security purpose**

- Consider a clerk who needs to know an instructor's ID, name, and department name, but does not have authorization to see the instructor's salary amount. 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**.

Why Views?

- Aside from security concerns, we may wish to create a personalized collection of "virtual" relations that is better matched to a certain user's intuition of the structure of the enterprise.
  - That is, to hide the complexity of logical model and to simplify users' interaction with the system

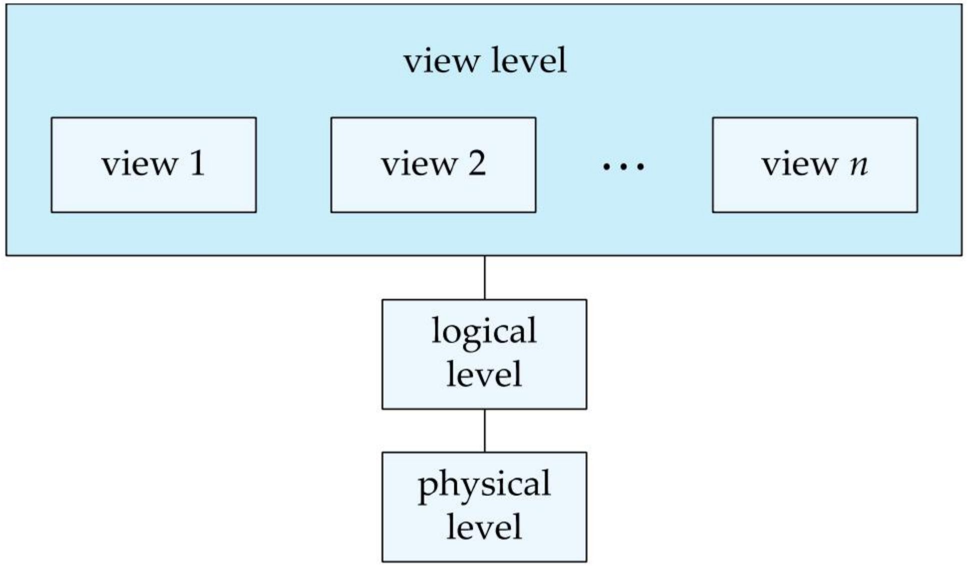
- Consider in our university example, we may want to have *"a list of all course sections offered by the Physics department in the Fall 2017 semester"*, with the building and room number of each section. The relation that we would create for obtaining such a list is:

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;

Views

- A **view** provides a mechanism to hide certain data from the view of certain users.
- In general, it is a bad idea to *compute* and *store* query results (as those in the previous examples)
  - May need to update everytime the original relations change
- 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 of Data



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.

View Definition (Cont.)

- Consider a clerk who needs to know an instructor's ID, name, and department name, but does not have authorization to see the instructor's salary amount.

```
create view faculty as
    select ID, name, dept_name
    from instructor;
```

- Consider in our university example, we may want to have *"a list of all course sections offered by the Physics department in the Fall 2017 semester"*, with the building and room number of each section.

```
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;
```

Using Views in SQL Queries

- Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates.

- *"Find all Physics courses offered in the Fall 2017 semester in the Watson building"*

```
select course_id
from physics_fall_2017
where building = 'Watson';
```

- Create a view of department salary totals, with attribute names of the view be specified explicitly

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

Using Views in SQL Queries (Cont.)

- When we define a view, the database system *stores the definition of the view*.
  - Rather than the result of evaluation of the query expression that defines the view.
  - To avoid out of date data whenever the relations used to define the view are modified.

- When a view relation appears in a query, it is replaced by the stored query expression.
  - Whenever we evaluate the query, the view relation is recomputed.

Views Defined Using Other Views

- One view may be used in the expression defining another view.

- Create a view for *"finding all Physics courses offered in the Fall 2017 semester in the Watson building"*

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

Views Defined Using Other Views

- One view may be used in the expression defining another view.

- Create a view for *"finding all Physics courses offered in the Fall 2017 semester in the Watson building"*

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

- Equivalent 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';
```

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**.

- **Materialized view maintenance** (or, view maintenance)
  - When relations used in the view definition are updated, the view is kept up-to-date.

- Purpose - to increase application performance for those using a view frequently
  - E.g., Applications that demand fast response to certain queries that compute aggregates over large relations can also benefit greatly by creating materialized views corresponding to the aggregation queries.
  - The materialized view is likely to be much smaller than the underlying large relations on which the view is defined. → Avoid reading large relations

Update of a View

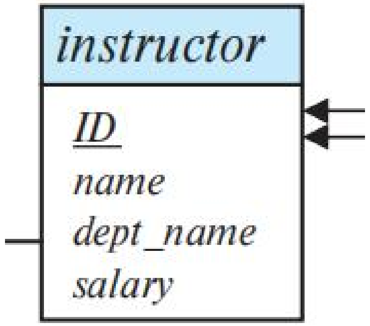
- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty*

  values ('30765', 'Green', 'Music');

```
create view faculty as
    select ID, name, dept_name
    from instructor;
```

- This insertion must be represented by the insertion into the *instructor* relation
  - Must have a value for salary.



- Two approaches
  - Reject the insertion
  - Insert the tuple

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

  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');

- Issues
  - If there is no instructor with ID 69987, and no department in the Taylor building?

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*

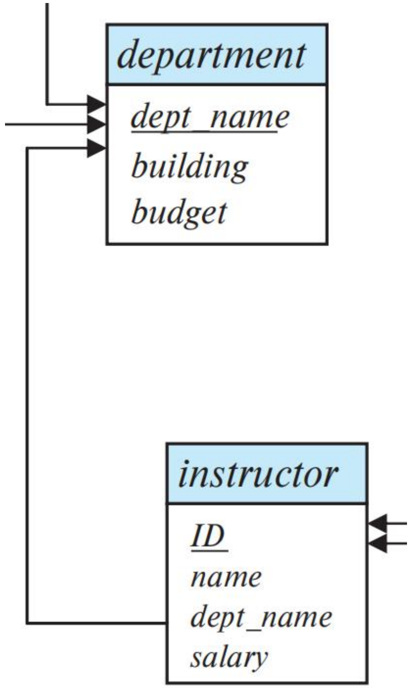
  values ('69987', 'White', 'Taylor');

- Issues
  - If there is no instructor with ID 69987, and no department in the Taylor building?
  - Insert ('69987', 'White', null, null) into *instructor* ?
  - Insert (null, 'Taylor', null) into *department* ?

instructor & *department* Relations

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*



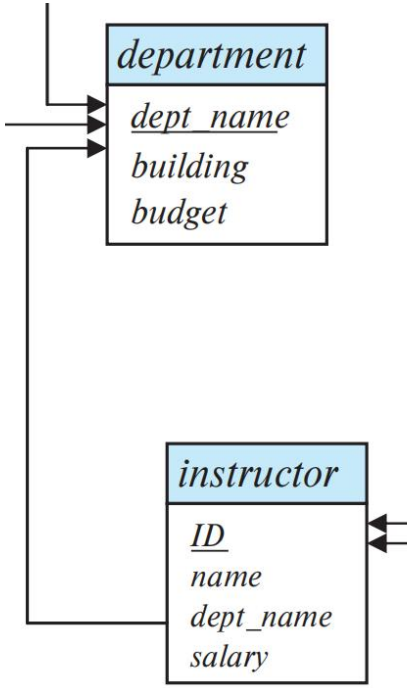
Does this update have the desired effect?

instructor & *department* Relations

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

The view *instructor_info* still does not include the tuple ('69987', 'White', 'Taylor') after the updte.



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
  - If there is no instructor with ID 69987, and no department in the Taylor building?
  - ~~Insert ('69987', 'White', null, null) into *instructor* ?~~
  - ~~Insert (null, 'Taylor', null) into *department* ?~~

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

- 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.

Outline

- Nested Subqueries
- Modification of the Database
- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a “unit” of work.
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit work -** The updates performed by the transaction become permanent in the database.
  - **Rollback work -** All the updates performed by the transaction are undone.
- Atomic transaction
  - Either fully executed or rolled back as if it never occurred

Transactions (Cont.)

- *Automatic commit* of individual SQL statements
  - In SQL implementations like MySQL, by default each SQL statement is taken to be a transaction on its own, and it gets committed as soon as it is executed.

- SQL:1999 standard
  - Allow multiple SQL statements to be enclosed between the keywords

begin atomic … end

- MySQL

```
START TRANSACTION;
```

…

```
COMMIT;
```

- Teradata

```
BEGIN TRANSACTION;
 DELETE FROM employee
 WHERE name = 'Reed C';
 UPDATE department
 SET emp_count = emp_count -1
 WHERE dept_no = 500;
END TRANSACTION;
```

Integrity Constraints

- **Integrity constraints** ensure that changes made to the database by authorized users do not result in a *loss of data consistency*.

- **Integrity constraints** guard against *accidental damage* to the database.
  - Security constraints guard against access to the database by *unauthorized users*.

- Examples of integrity constraints are:
  - An instructor name cannot be null.
  - No two instructors can have the same instructor ID.
  - Every department name in the course relation must have a matching department name in the department relation.
  - The budget of a department must be greater than $0.00.

Constraints on a Single Relation

- **primary key**

- **not null**
- **unique**
- **check** (P), where P is a predicate

Not Null Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**

- The **not null** constraint
  - Prohibits the insertion of a null value for the attribute.
  - An example of a **domain constraint**

Unique Constraints

- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes A_1, A_2, …, A_m form a *superkey*.
  - Candidate keys are permitted to be **null** (in contrast to primary keys).

The check Clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.

- Example: ensure that semester is one of 'Fall', 'Winter', 'Spring', or 'Summer'.

```
create table section
    (course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6),
     year           numeric (4,0),
     building       varchar (15),
     room_number    varchar (7),
     time_slot_id   varchar (4),
     primary key (course_id, sec_id, semester, year),
     check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```

```
create table department
    (dept_name    varchar (20),
     building     varchar (15),
     budget       numeric (12,2) check (budget > 0),
     primary key (dept_name));
```

- A check clause may appear on its own, as shown above, or as part of the declaration of an attribute.

Referential Integrity

- Ensures that a value that appears in one relation (the *referencing* relation) for a given set of attributes also appears for a certain set of attributes in another relation (the referenced relation).

  - Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".

- *Foreign keys* are a form of a referential integrity constraint.
  - The referenced attributes form a primary key of the referenced relation.

Referential Integrity (Cont.)

- *Foreign keys* can be specified as part of the SQL **create table** statement.
  - E.g., in *course* relation definition:

    foreign key (*dept_name*) **references** *department*

- By default, a foreign key references the primary-key attributes of the referenced table.

- SQL allows a list of attributes of the referenced relation to be specified explicitly.

  foreign key (*dept_name*) **references** *department* (*dept_name*)

  - The specified list of attributes must be declared as a **superkey** of the referenced relation, using either a **primary key** constraint or a **unique** constraint.
  - The foreign key must reference a **compatible** set of attributes, i.e., the number of attributes must be the same and the data types of corresponding attributes must be compatible.

Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

  create table *course* (
  (…
  dept_name **varchar**(20),
  foreign key (*dept_name*) **references** *department*
  on delete cascade
  on update cascade,
  . . .)

If a delete or update action on the referenced relation violates the constraint, the system must take steps to change the tuple in the referencing relation to restore the constraint.

- Instead of **cascade** we can use :
  - **set null**,
  - **set default**

Integrity Constraint Violation During Transactions

- Consider:

  create table *person* (
  ID **char**(10),
  name **char**(40),
  mother **char**(10),
  father **char**(10),
  primary key *ID,*
  foreign key *father* **references** *person,*
  foreign key *mother* **references** *person*)

- How to insert a tuple without causing constraint violation?
  - Insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
  - For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
  - An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:

  create assertion <assertion-name> **check** (<predicate>);

Outline

- Nested Subqueries
- Modification of the Database
- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length n.
- **varchar(n).** Variable length character strings, with user-specified maximum length n.

- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).

- **numeric(p,d).** Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least n digits.

Date & Time Types in SQL

- **date:** Dates, containing a (4 digit) year, month and day of the month
  - Example: **date** '2022-08-26'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp:** A combination of **date** and **time**
  - Example: **timestamp** '2022-08-26 21:00:30.75'
- **interval:** period of time
  - Example: interval 1 day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values
  - Example: if x and y are of type **date**, then $x - y$ is an **interval** whose value is the number of days from date x to date y.

Default Values

- Specify **default** value for an attribute in **create table** statement:

```
create table student
    (ID          varchar (5),
     name        varchar (20) not null,
     dept_name   varchar (20),
     tot_cred    numeric (3,0) default 0,
     primary key (ID));
```

- Insertion to *student* can omit the value for the *toto_cred* attribute

```
insert into student(ID, name, dept_name)
    values ('12789', 'Newman', 'Comp. Sci.');
```

Large-Object Types

- Large data items (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob -** binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)

    image **blob**(10MB)
  - **clob -** character large object -- object is a large collection of character data

    book_review **clob**(10KB)
  - **lob - L**arge **OB**ject
- When a query returns a large object, a pointer is returned rather than the large object itself.

User-Defined Types

- **create type** construct (SQL:1999) in SQL creates user-defined type

  create type *Dollars* **as numeric (12,2) final;**

  create type *Pounds* **as numeric (12,2) final;**

User-Defined Types

- **create type** construct (SQL:1999) in SQL creates user-defined type

  create type *Dollars* **as numeric (12,2) final;**

  create type *Pounds* **as numeric (12,2) final;**

  - An attempt to assign a value of type *Dollars* to a variable of type *Pounds* results in a compile-time error

- Example:

  create table *department*
  (*dept_name* **varchar** (20),
  building **varchar** (15),
  budget Dollars);

Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. But,
  - Domains can have constraints, such as **not null**, specified on them.
  - Domains can have default values specified on them.
  - Values of one domain type can be assigned to values of another domain type as long as the underlying types are compatible.

- Example:

create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
    check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Outline

- Nested Subqueries
- Modification of the Database
- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

Index Creation

- Many queries reference only a small proportion of the records in a table.
  - *“Find all instructors in the Physics department”*
  - *“Find the salary value of the instructor with ID 22201”*
  - It is inefficient for the system to read every record to find a record with particular value

- An **index** on an attribute of a relation is a **data structure** that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
  - Part of the ***physical schema*** of the database

- We create an index with the **create index** command (*DDL command*) and drop an index with the **drop index** command

  create index <name> **on** <relation-name> (<attribute-list>);

  drop index <index-name>;

Index Creation Example

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))

- **create index** *studentID_index* **on** *student*(*ID*)

- The query:

  select *
  from *student*
  where *ID* = '12345'

  can be executed by using the index to find the required record, without looking at all records of *student*

Authorization

- We may assign a user several forms of authorizations on data of the database.
  - **Read** - allows reading, but not modification of data.
  - **Insert** - allows insertion of new data, but not modification of existing data.
  - **Update** - allows modification, but not deletion of data.
  - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**.
- We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list>

  on <relation or view >

  to <user/role list>;

- <user/role list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Example:

  grant select on *department* **to** Amit, Satoshi;

- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

    grant select on *student* **to** U_1, U_2, U_3**;**

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list> **on** <relation or view> **from** <user list>

  - Example:

  revoke select on *student* **from** U_1, U_2, U_3;

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
  - Each instructor must have the same types of authorizations on the same set of relations.

- To create a role we use:

  create role <name>;

- Example:

  create role *instructor*;

- Once a role is created we can assign “users” to the role using:
  - **grant** <role> **to** <users>

Roles Examples

- Create a role: **create role** *instructor*;
- Grant roles to users: **grant** *instructor* **to** Amit;
- Privileges can be granted to roles: **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles

  create role *teaching_assistant;*

  grant *teaching_assistant* **to** *instructor*;

  - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles

  create role *dean*;

  grant *instructor* **to** *dean*;

  grant *dean* **to** Satoshi;

Roles Examples

- Create a role: **create role** *instructor*;
- Grant roles to users: **grant** *instructor* **to** Amit;
- Privileges can be granted to roles: **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles

  create role *teaching_assistant;*

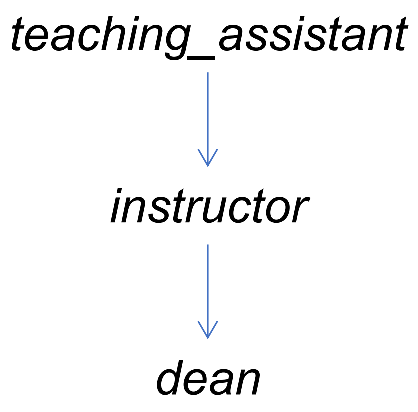
  grant *teaching_assistant* **to** *instructor*;

  - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles

  create role *dean*;

  grant *instructor* **to** *dean*;

  grant *dean* **to** Satoshi;



Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');

- **grant select on** *geo_instructor* **to** *geo_staff*

- Suppose that a *geo_staff* member issues

  select *
  from *geo_instructor*;

- What if
  - *geo_staff* does not have permissions on *instructor?*
  - Creator of view did not have some permissions on *geo_instructor* / *instructor?*

Other Authorization Features

- **references** privilege to create foreign key

  grant reference (*dept_name*) **on** *department* **to** Mariano;

  - This grant statement allows user Mariano to create relations that reference the key *dept_name* of the *department* relation as a foreign key
  - *Why is this required?*

- Transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
    - Grant a privilege and to allow the recipient to pass the privilege on to other users
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
  - **revoke grant option for select on** *department* **from** Amit;

Other Authorization Features

- **references** privilege to create foreign key

  grant reference (*dept_name*) **on** *department* **to** Mariano;

  - This grant statement allows user Mariano to create relations that reference the key *dept_name* of the *department* relation as a foreign key
  - *Why is this required?*
  - Foreign-key constraints restrict deletion and update operations on the referenced relation.

- Transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
    - Grant a privilege and to allow the recipient to pass the privilege on to other users
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
  - **revoke grant option for select on** *department* **from** Amit;