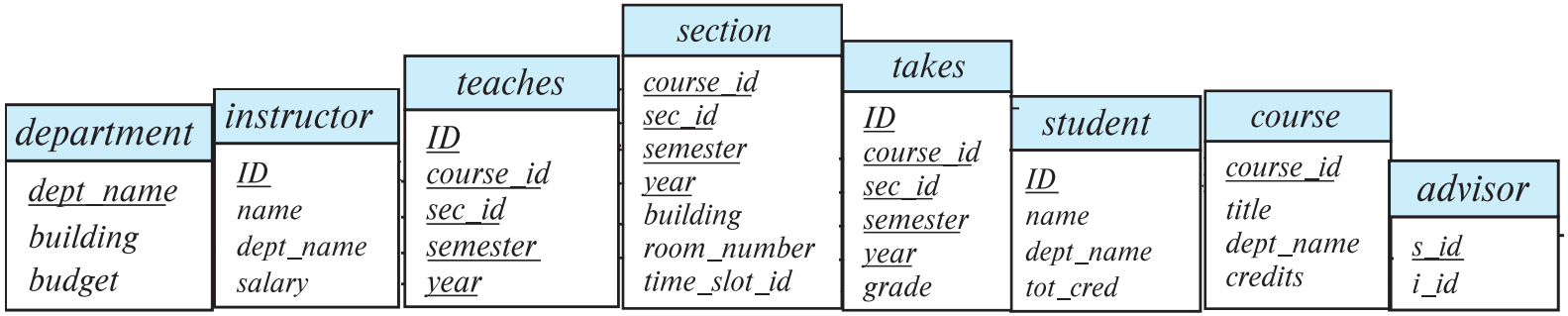
Introduction to SQL

Miao Qiao
The University of Auckland

Basic Query Structure

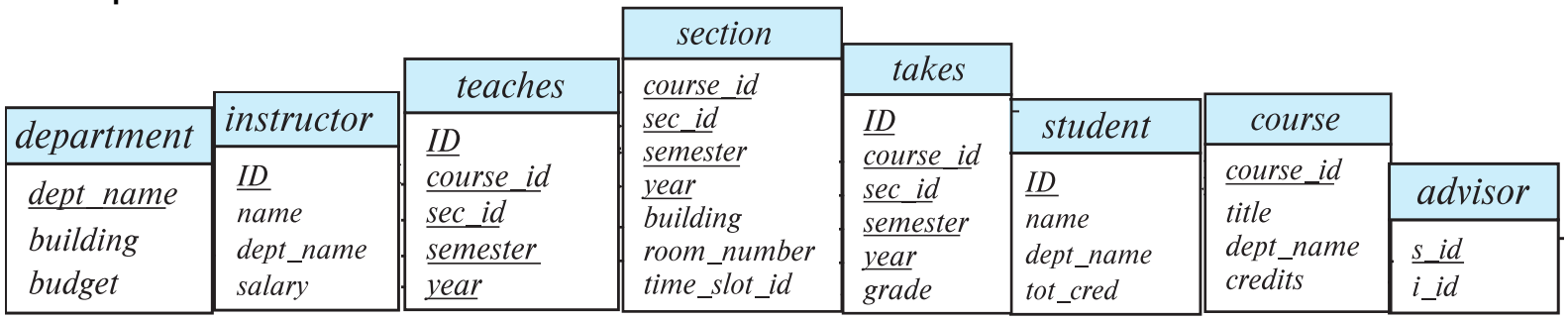
- A typical SQL query has the form:

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P



Basic Query Structure (demo)

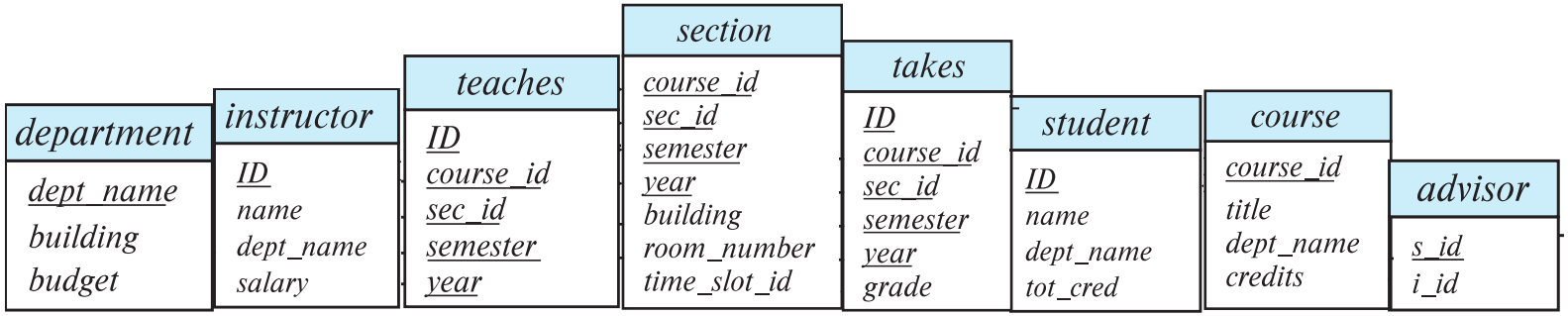
1. Find the names of all instructors
2. Find the department names of all instructors, and remove duplicates
3. Find the department names of all instructors, not removing duplicates
4. Find all attributes of instrctor show the entire instructor table
5. Find a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12
6. Find all instructors in Comp. Sci. dept with salary > 70000
7. Find the names of all instructors who have taught some course and the course_id
8. Find the names of all instructors in the Comp. Sci. department who have taught some course and the course_id
9. Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.



Basic Query Structure (demo)

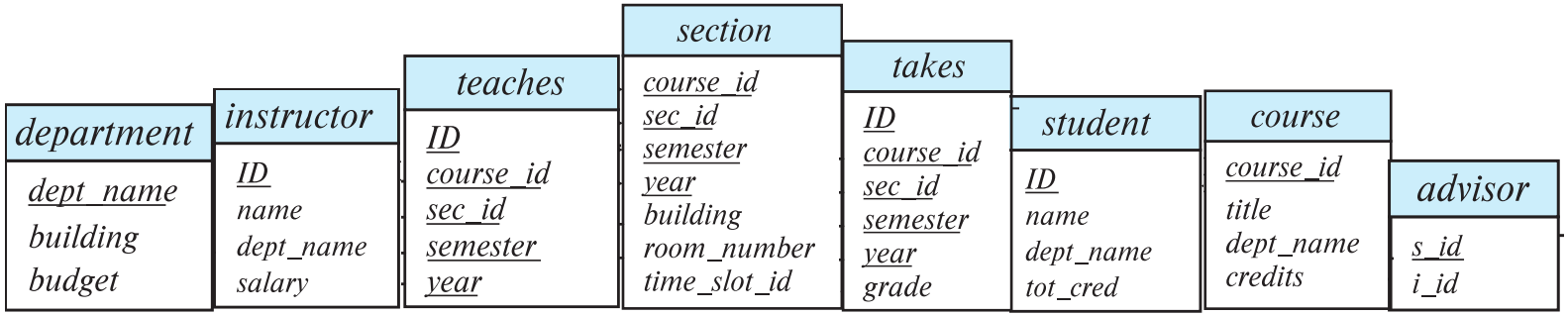
- Find the names of all instructors whose name includes the substring “in”.
- String Operations
  - The operator **like** uses patterns that are described using two special characters:
    - percent (%). The % character matches any substring.
    - underscore (_). The _ character matches any character

- Find the names of all instructors whose name has 4 characters.
- Find the names of all instructors whose name has at leaset 4 characters.



Basic Query Structure (demo)

1. List in alphabetic order the names of all instructors
2. List in descending alphabetic order the names of all instructors
3. List in order of the combination of the names and salary of all instructors
4. Find the names of all instructors with salary between $90,000 and $100,000
5. Find courses that ran in Fall 2017 or in Spring 2018
6. Find courses that ran in Fall 2017 and in Spring 2018
7. Find courses that ran in Fall 2017 but not in Spring 2018
8. Find courses that ran in Fall 2017 or in Spring 2018, retain all duplications
9. Find all instructors whose salary is null
10. Find all instructors whose salary is not null
11. Null under and, or, with true/false



Basic Query Structure (demo)

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

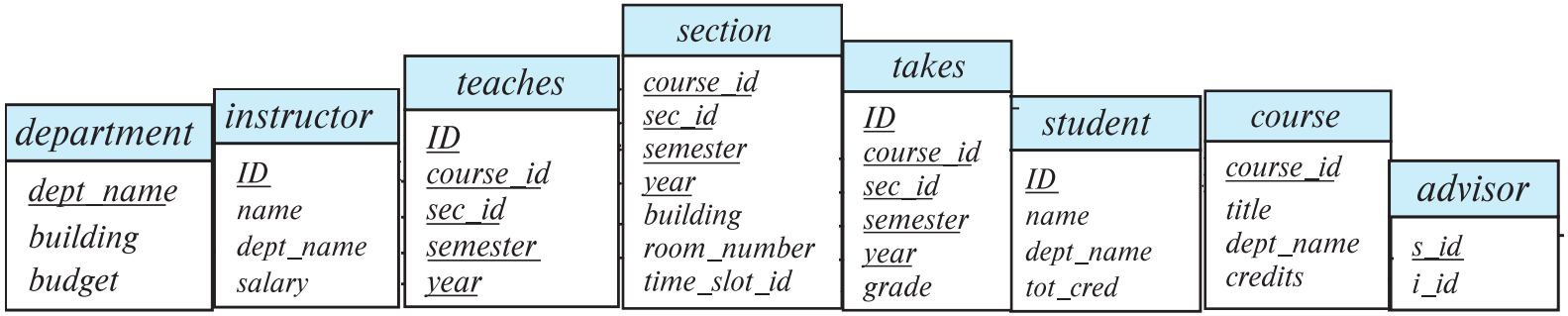
select A_1, A_2, ..., A_n ← Aggregation function over values over multiple rows

from r_1, r_2, ..., r_m

where P

group by columns ← New clauses

having condition

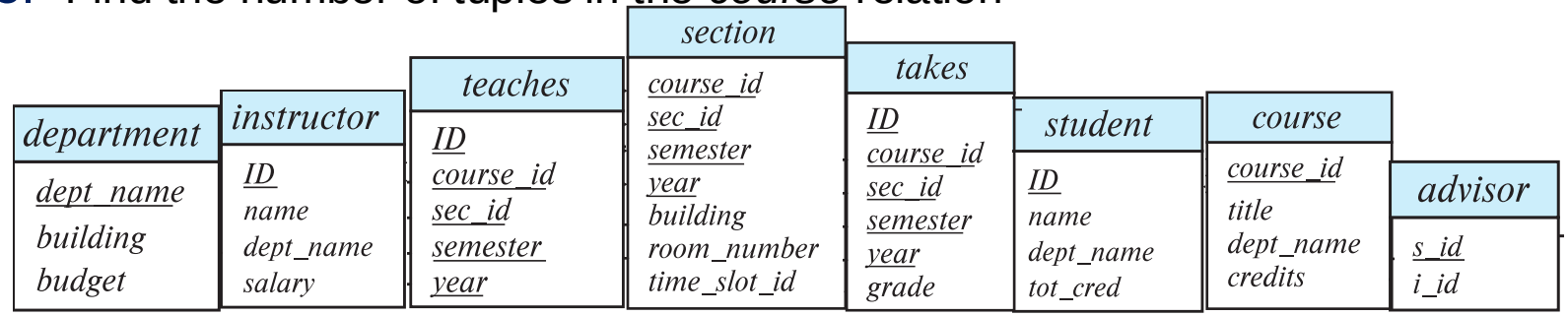


Basic Query Structure (demo)

- These functions operate on the multiset of values of a column of a relation, and return a value

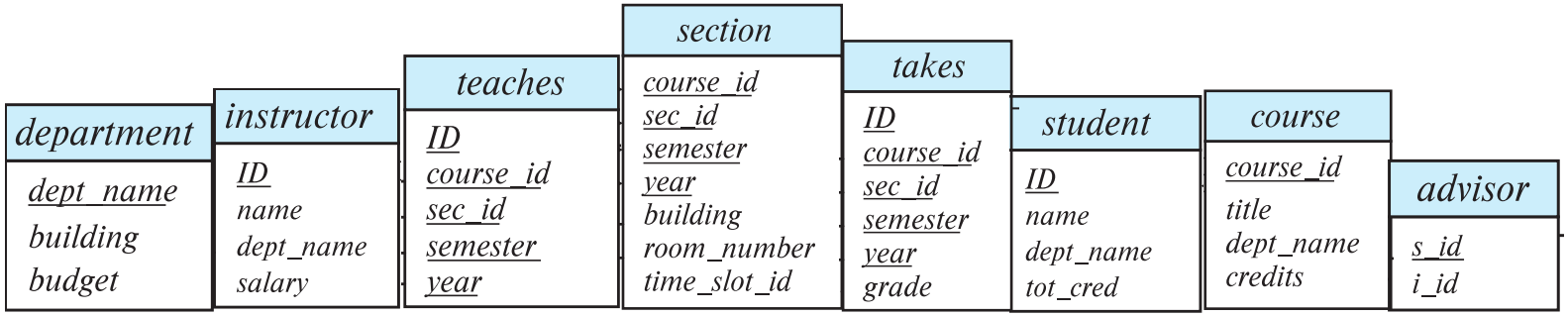
avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

1. Find the highest salary of any instructor
2. Find the average salary of instructors in the Computer Science department
3. Find the lowest salary of an instructor who have taught a course
4. Find the total number of instructors who teach a course in the Spring 2018 semester
5. Find the number of tuples in the *course* relation



Aggregate – Group By - Having

1. Find the average salary of instructors in each department
2. Find the names and average salaries of all departments whose average salary is greater than 42000
3. Find the names and average salaries of all departments over instructors whose salary is greater than 70000
4. Find the names and average salaries of all departments whose average salary is greater than 70000
5. Find the average salaries of instructors who have taught a course



Nested 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 $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P

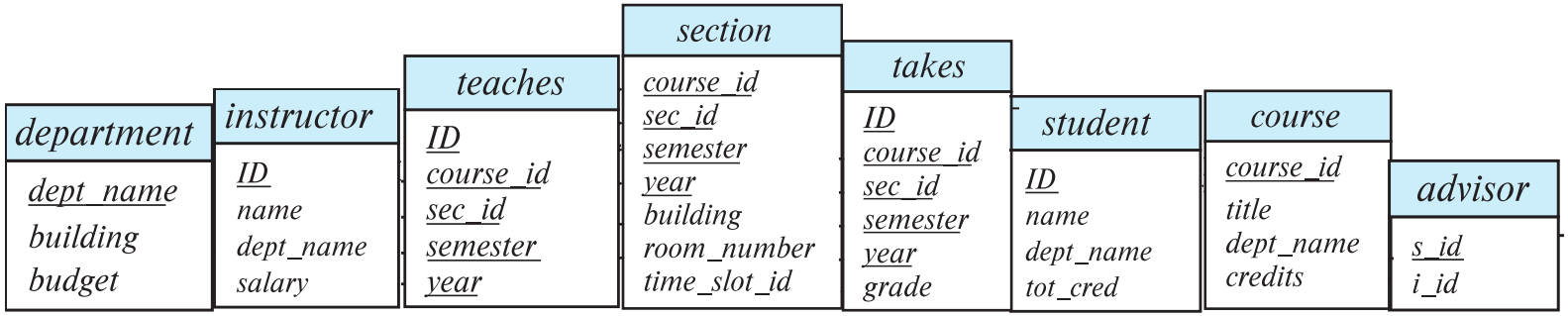
  as follows:

  - **From clause:** r_i can be replaced by any valid subquery
  - **Where clause:** P can be replaced with an expression of the form:

    B <operation> (subquery)

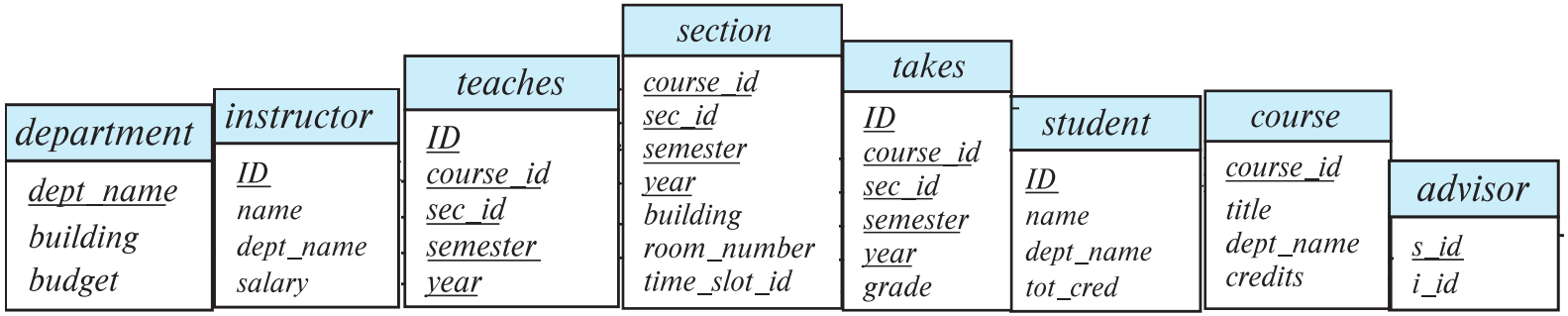
    B is an attribute and <operation> to be defined later.
  - **Select clause:**

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



Subquery in Where Clause

1. Name all instructors whose name is either “Mozart” or Einstein”
2. Name all instructors whose name is neither “Mozart” nor Einstein”
3. Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101 or 12121 or 15151
4. Find names of instructors with salary greater than that of some (at least one) instructor in the Computer Science department.
5. Find all instructors earning the highest salary (there may be more than one with the same salary).
6. Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.



Definition of "some" Clause

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

(5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

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

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

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

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

Definition of "all" Clause

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

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

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

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

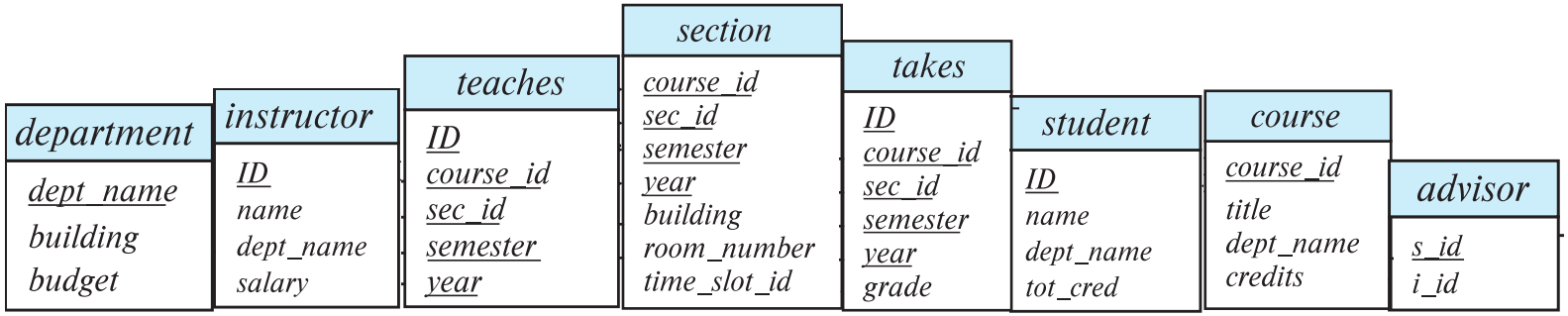
(5 ≠ **all** | 4, 6 |) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≠ **in**

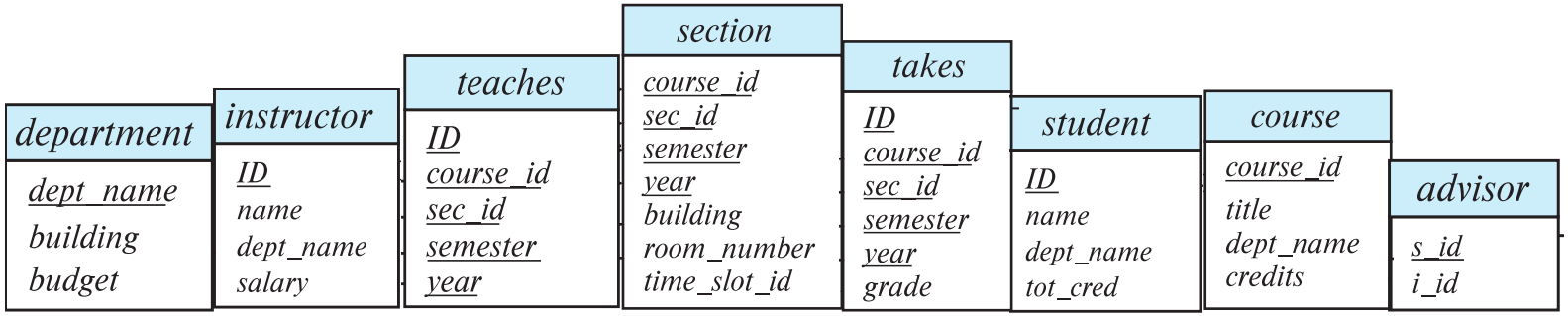
Subquery in Where Clause

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
  - Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester
- 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



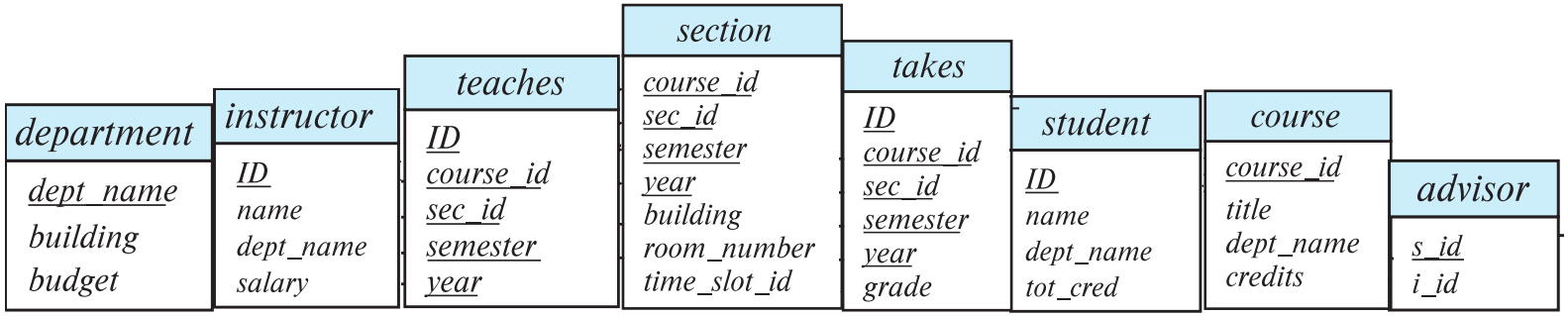
Query Quest

1. Find names of instructors with salary greater than that of some (at least one) instructor in the Computer Science department.
   - Use self-join, some, exists, aggregation
2. Find all instructors earning the highest salary (there may be more than one with the same salary).
   - Use self-join, all, exists, aggregation



Subquery in From and Select Clauses

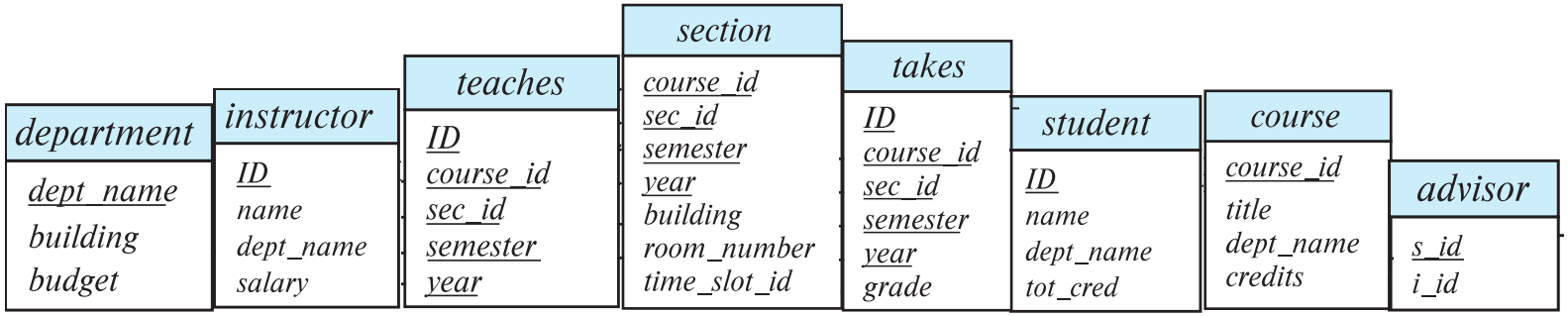
1. Find the average instructors' salaries of those departments where the average salary is greater than $42,000"
2. Find all departments with the maximum budget (with clause)
3. Find all departments where the total salary is greater than the average of the total salary at all departments
4. List all departments along with the number of instructors in each department



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;
```



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

- **delete from** *a relation* **where** condition

Deletion

- Delete all instructors

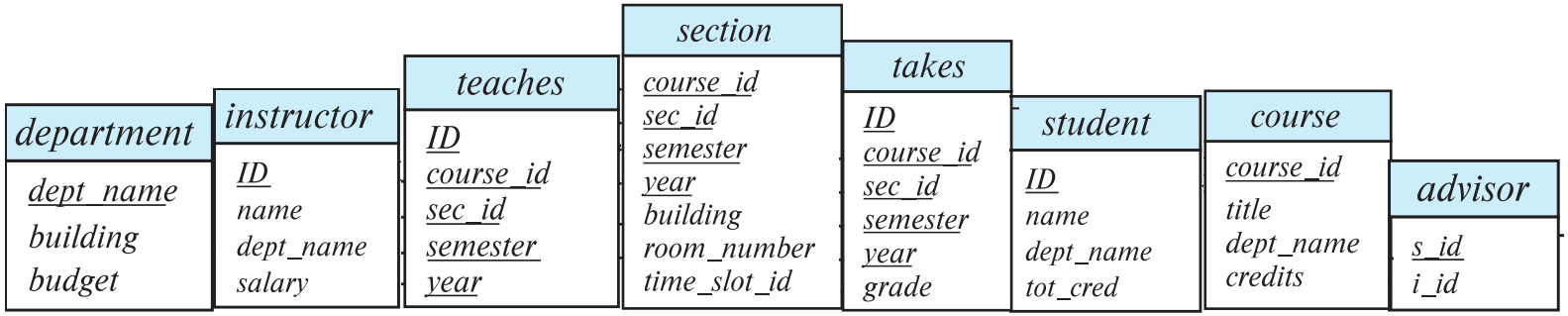
  delete from *instructor*

- 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');

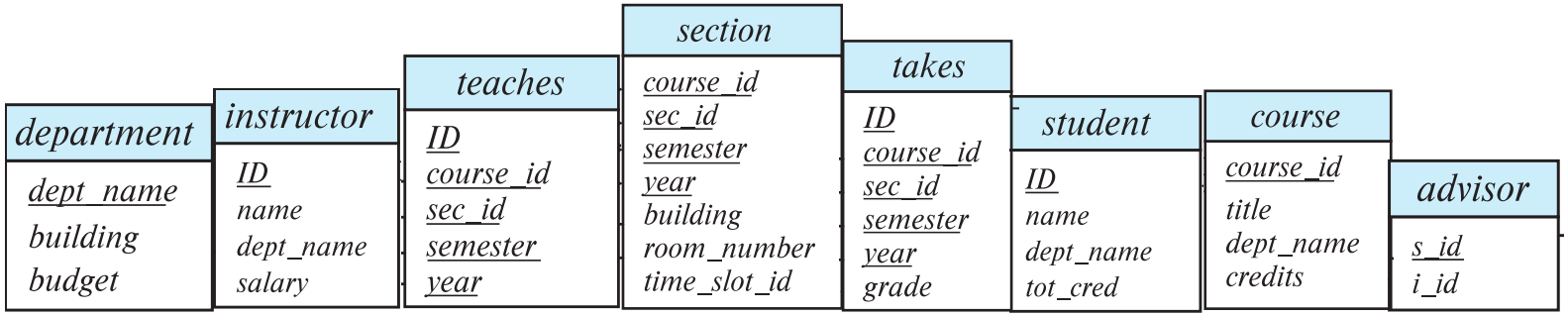


Deletion (Cont.)

- 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*

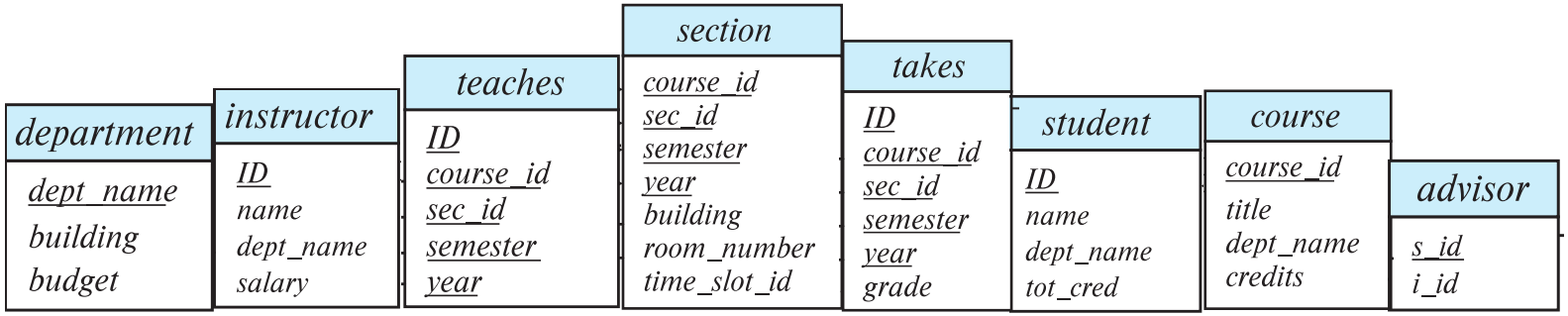
  insert into *course*
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

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

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

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



Insertion (Cont.)

- 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.

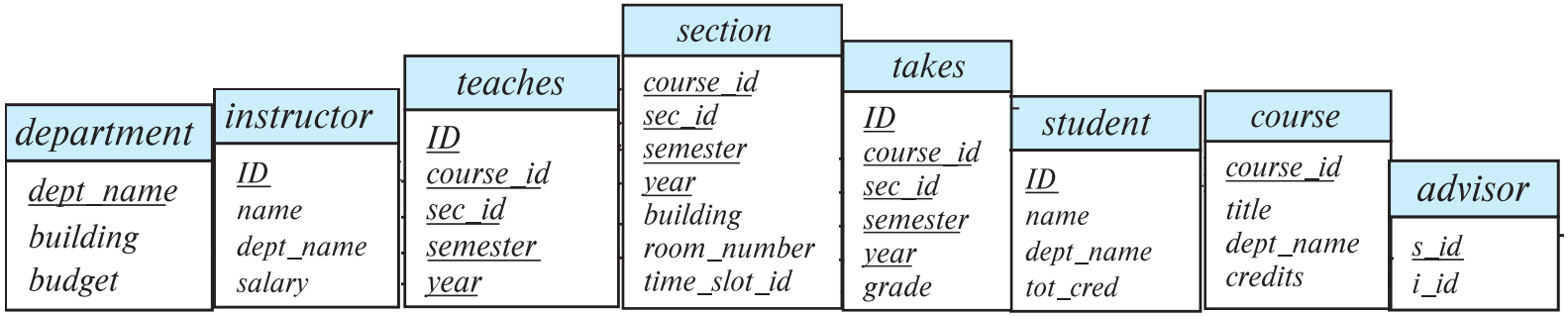
```
insert into instructor
    select ID, name, dept_name, 18000
    from   student
    where  dept_name = 'Music' and total_cred > 144;
```

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

Otherwise queries like

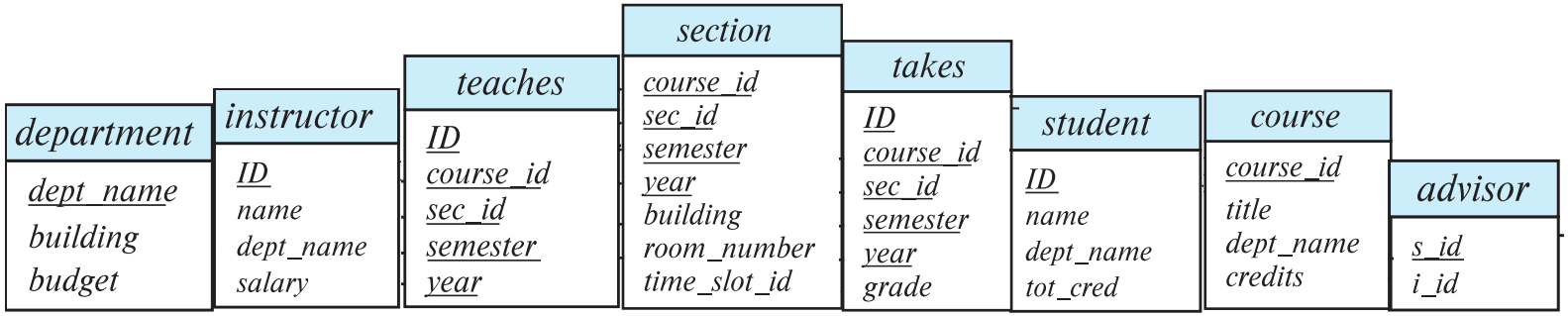
```
insert into table1 select * from table1
```

would cause problem



Updates

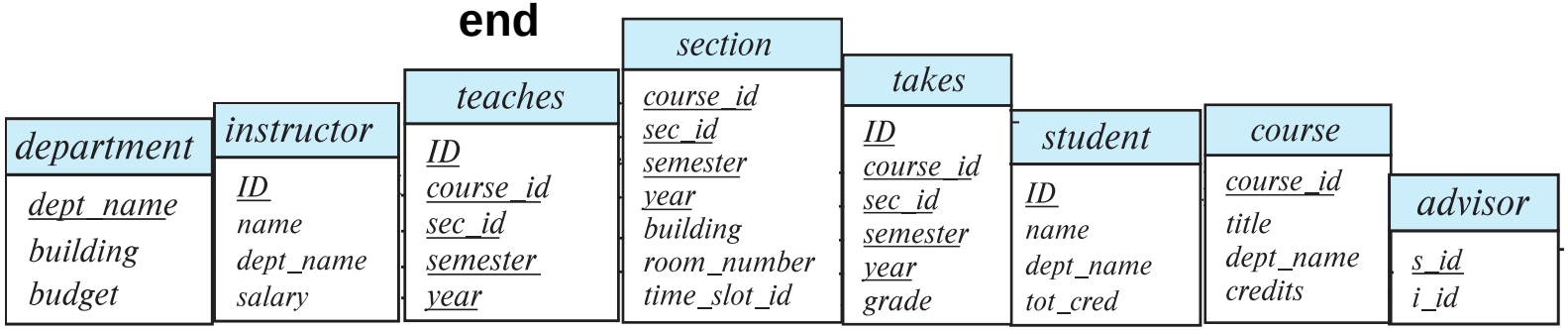
1. Give a 5% salary raise to all instructors
2. Give a 5% salary raise to those instructors who earn less than 70000
3. Give a 5% salary raise to instructors whose salary is less than average



Updates (Cont.)

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

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



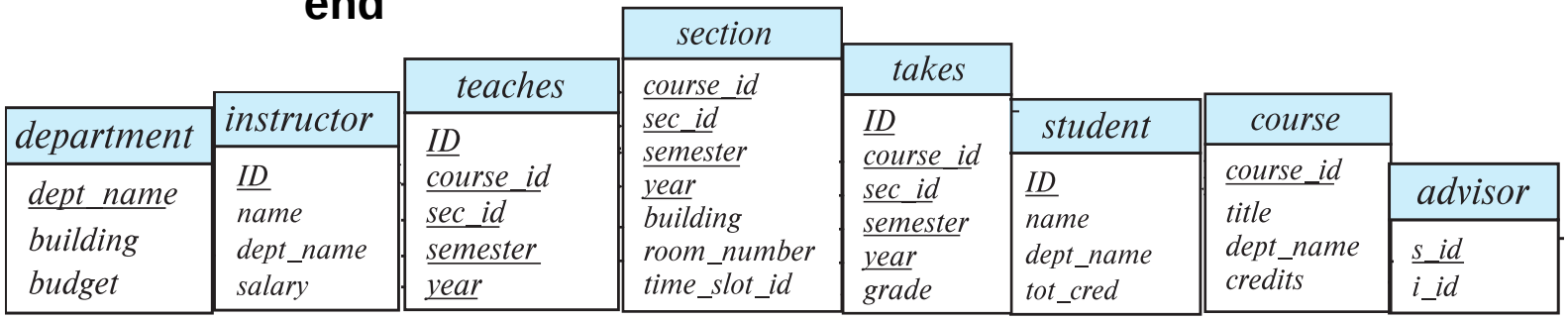
Updates with Scalar Subqueries

- 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);
```

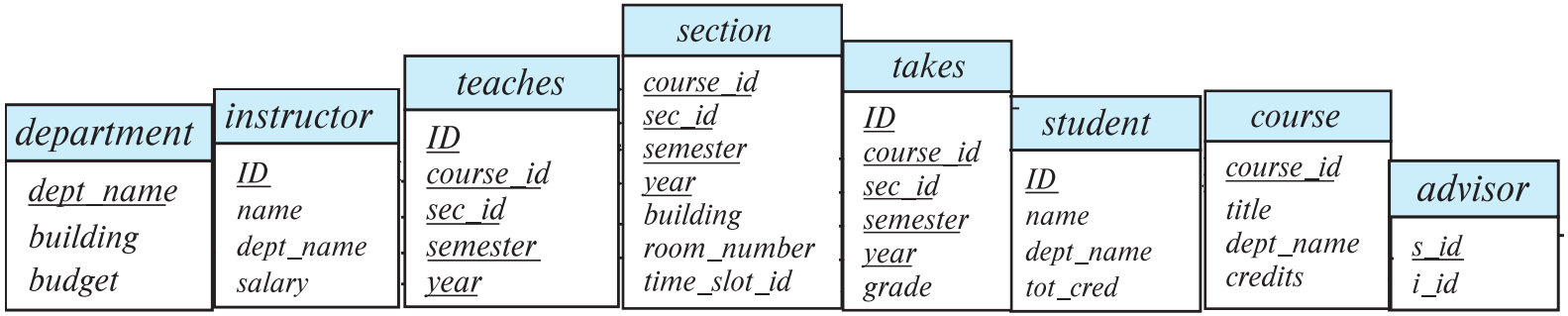
- Sets *tot_creds* to null for students who have not taken any course
  - Instead of **sum**(*credits*), use:

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



Join conditions

- List the names of students instructors along with the titles of courses that they have taken
  - Natural Join with **Using** Clause
  - Join condition with **on** condition



Join types

- Three forms of outer join:
  - Natural left outer join *course* ⟕ *prereq*
  - Natural right outer join *course* ⟖ *prereq*
  - Natural full outer join *course* ⟗ *prereq*

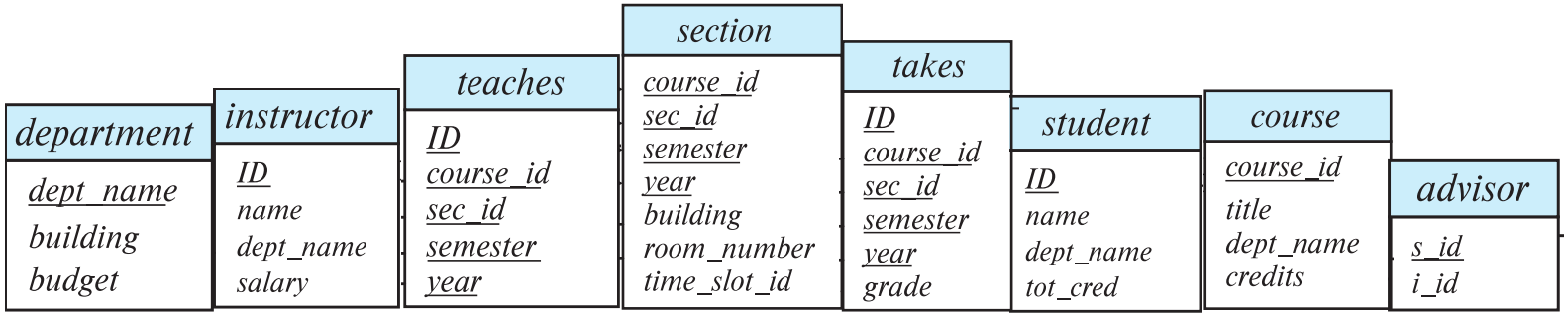
course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

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_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

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



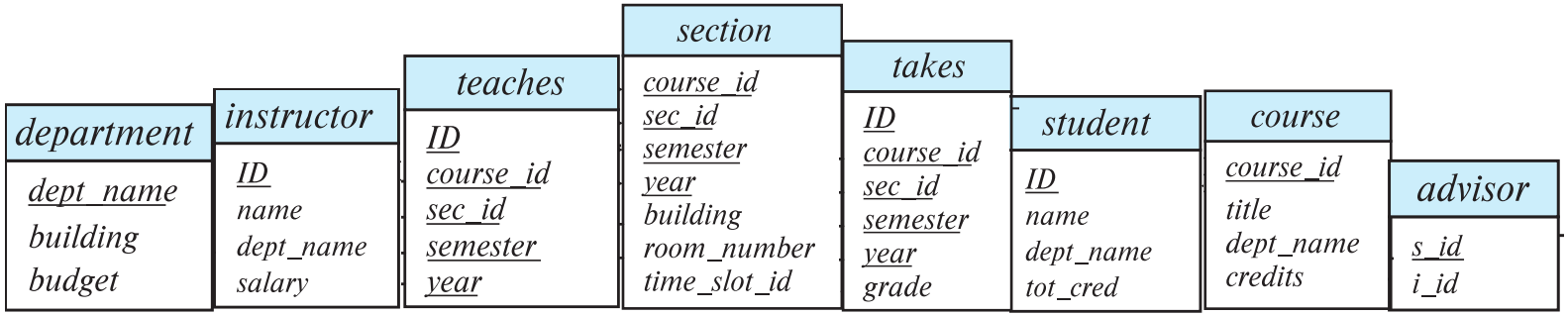
Join

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

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

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



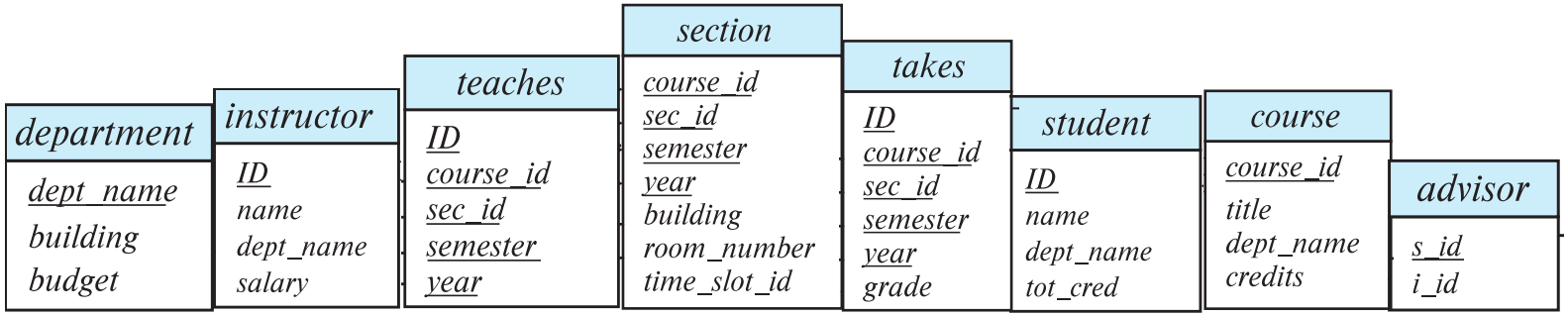
Joined Relations – Examples

- *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



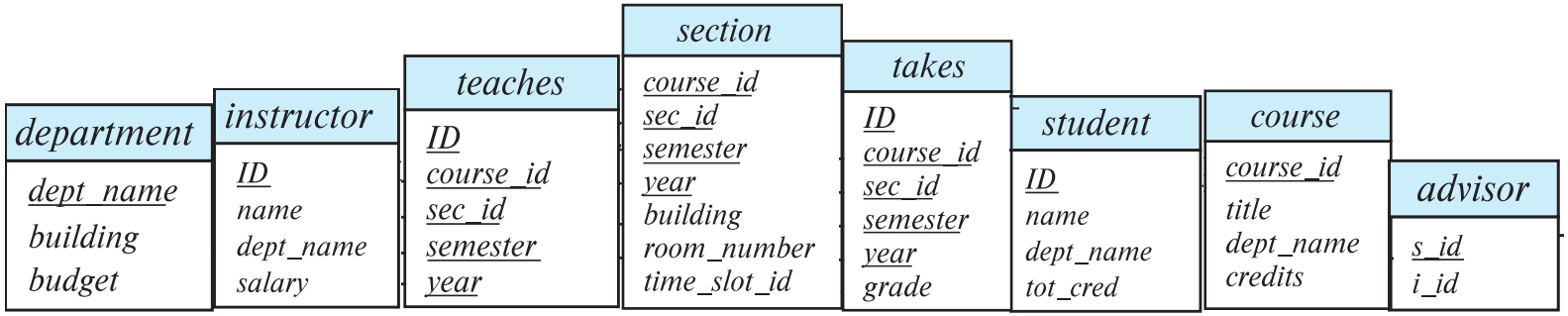
Joined Relations – Examples

- *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*



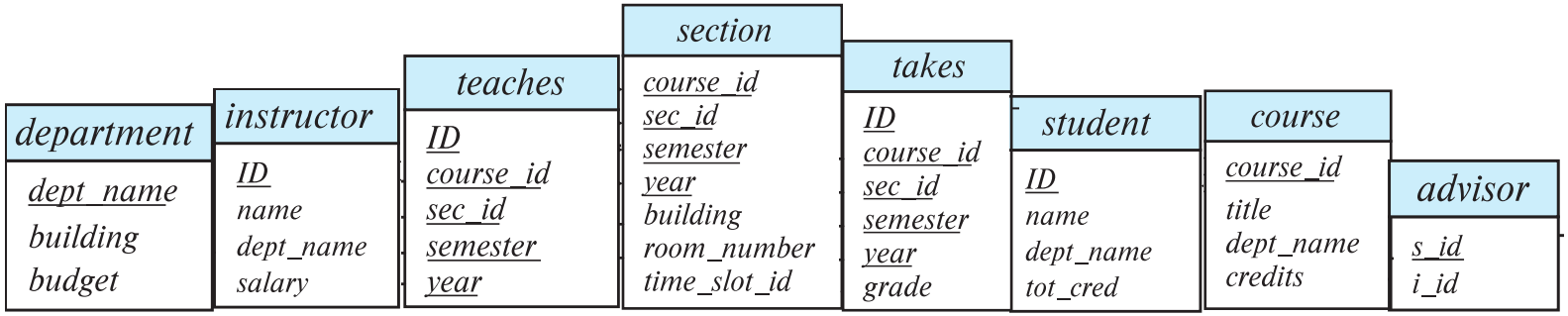
Joined Relations – Examples

- *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

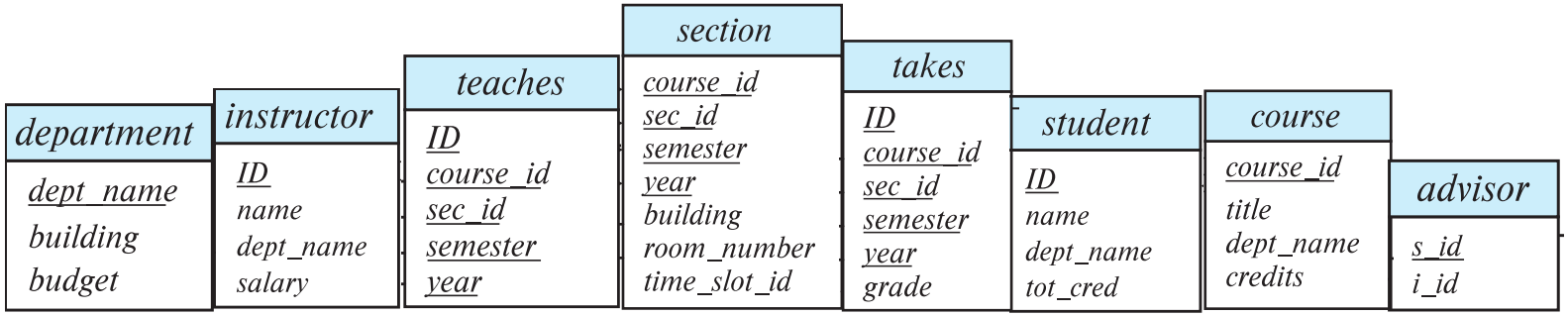


View

- A **view** provides a mechanism to hide certain data from the view of certain users.
  - 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*
- **create view** *v* **as** < query expression >

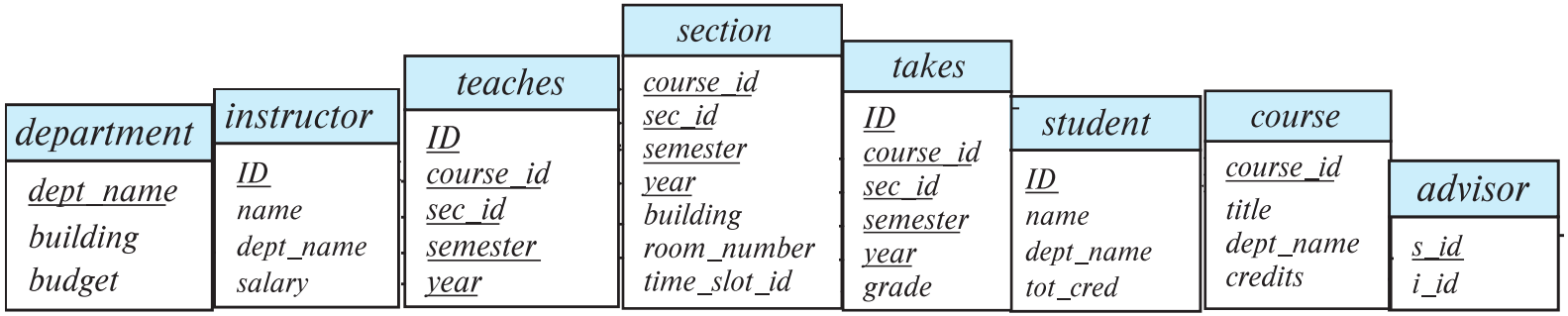
1. A view of instructors without their salary
2. Find all instructors in the Biology department
3. Create a view of department salary totals



Define Views using other Views

- **create view** ***physics_fall_2017*** **as**
  select *course.course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course.course_id = section.course_id*
  and *course.dept_name* = 'Physics'
  and *section.semester* = 'Fall'
  and *section.year* = '2017';

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



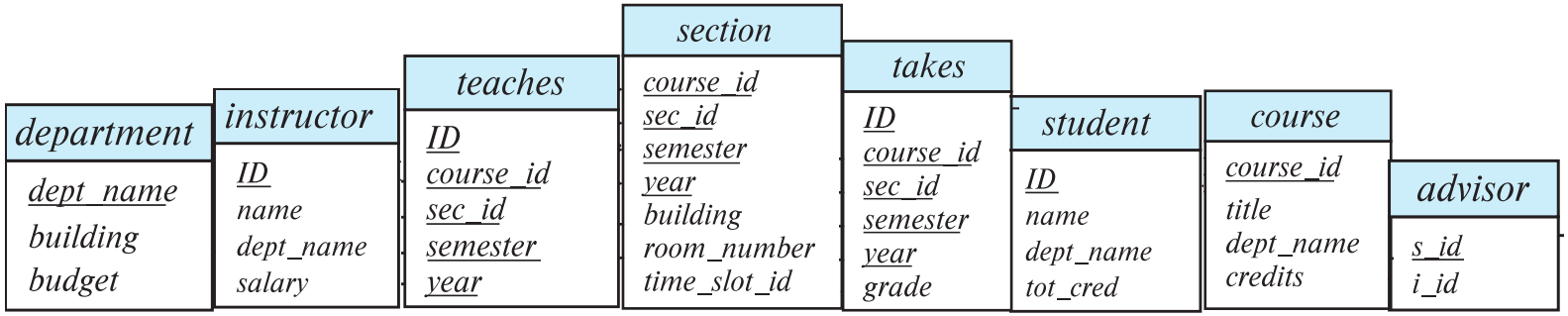
View Expansion

- Expand the view :

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

- To:

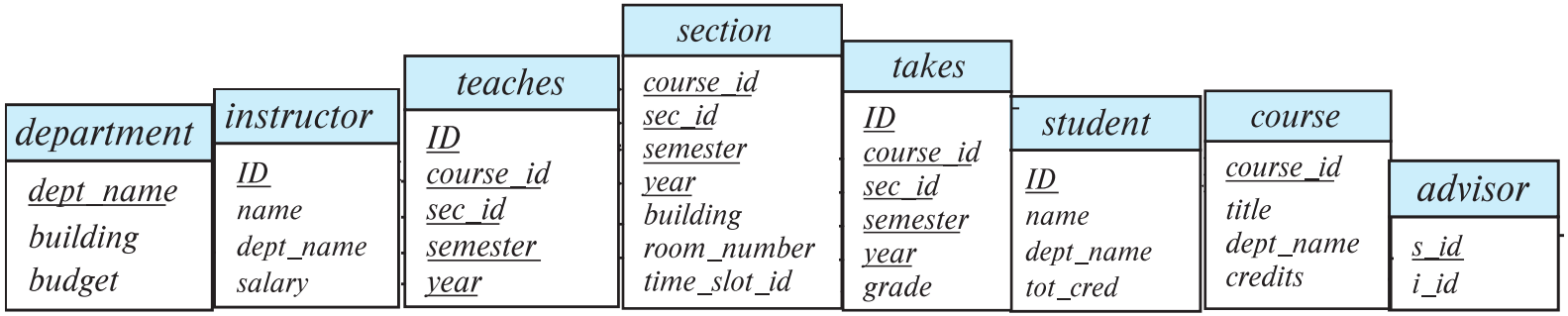
```
create view physics_fall_2017_watson as
    select course_id, room_number
    from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2017')
    where building= 'Watson';
```



Views

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

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

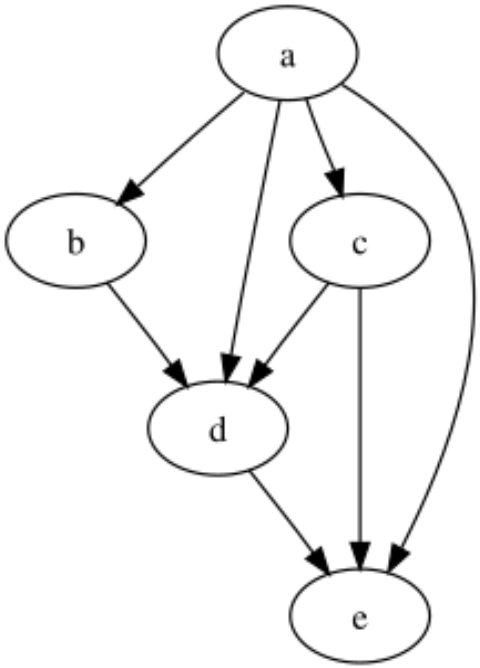


Some Updates Cannot be Translated Uniquely

- **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 in Taylor?
  - What if no department is in Taylor?

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

Dependency Graph of Views should be a DAG



department
- dept_name
- building
- budget

instructor
- ID
- name
- dept_name
- salary

teaches
- ID
- course_id
- sec_id
- semester
- year

section
- course_id
- sec_id
- semester
- year
- building
- room_number
- time_slot_id

takes
- ID
- course_id
- sec_id
- semester
- year
- grade

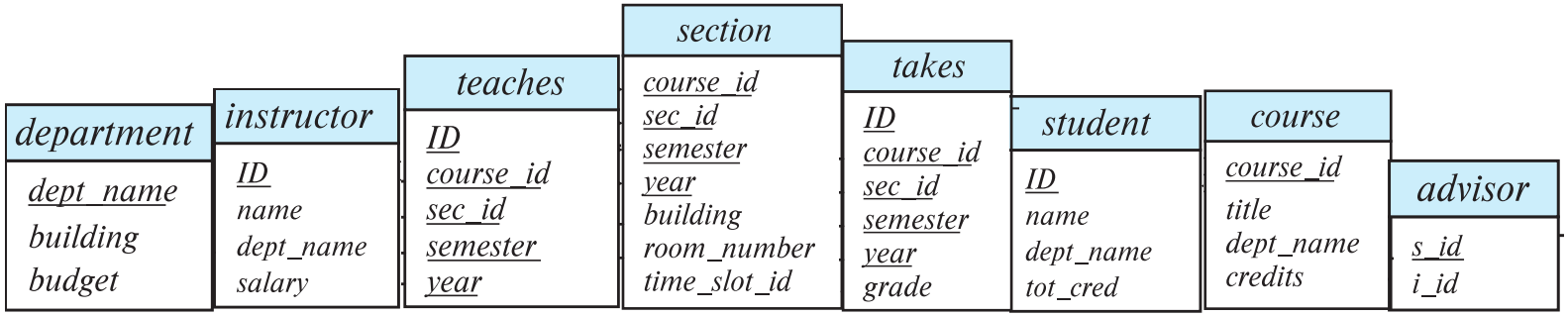
student
- ID
- name
- dept_name
- tot_cred

course
- course_id
- title
- dept_name
- credits

advisor
- s_id
- i_id

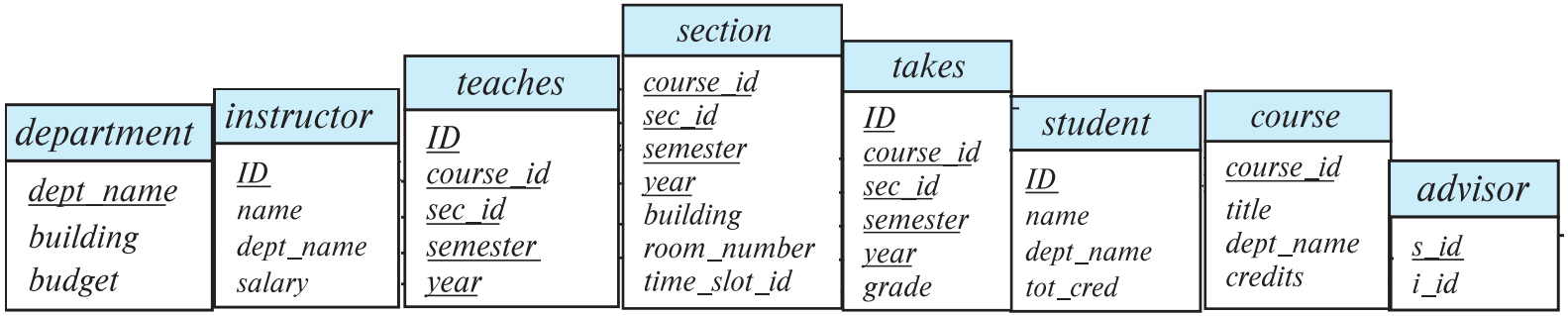
Query Quest

1. Display a list of all instructors, showing their ID, name, and the number of sections that they have taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use scalar subqueries.
2. Outer join expressions can be computed in SQL without using the SQL outer join operation. To illustrate this fact, show how to rewrite each of the following SQL queries without using the outer join expression.
    1. **select** * **from** student **natural left outer join** takes
    2. (hard) **select** * **from** student **natural full outer join** takes



Integrity Constraints

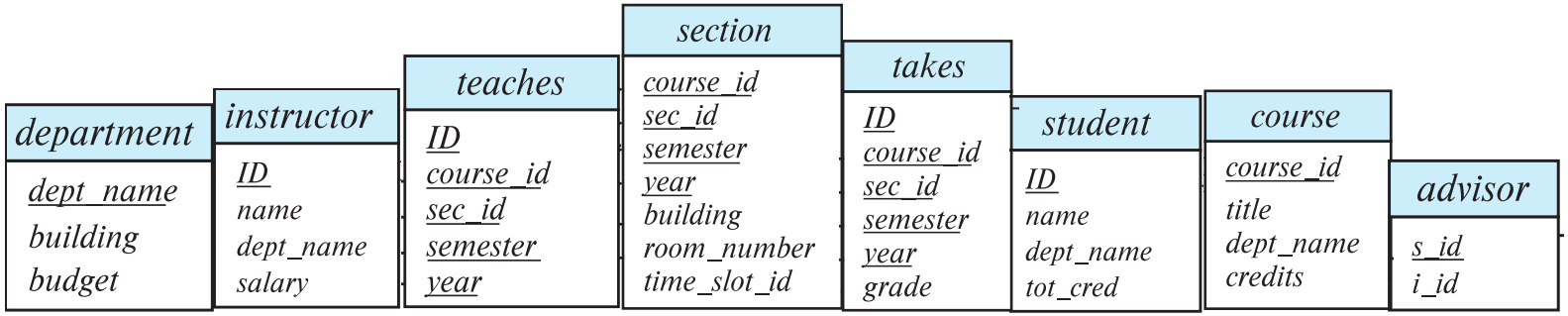
- A checking account must have a balance greater than $10,000.00
- A salary of a bank employee must be at least $4.00 an hour
- A customer must have a (non-null) phone number



Constraints on a Single Relation

- **not null**
  - *name* **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**
- **primary key**
- **unique** (A_1, A_2, …, A_m)
  - Candidate keys are permitted to be null (in contrast to primary keys).
- **check** (P), where P is a predicate (read DDL.sql)

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



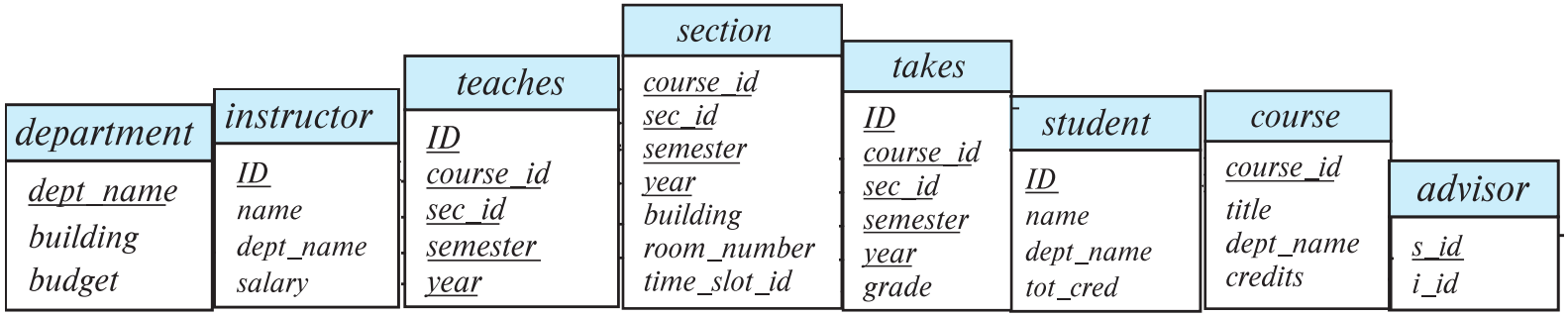
The check clause

- 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')))
```

Complex check conditions in **section** (not supported by sqlite)

```
check (time_slot_id in (select time_slot_id from time_slot))
```

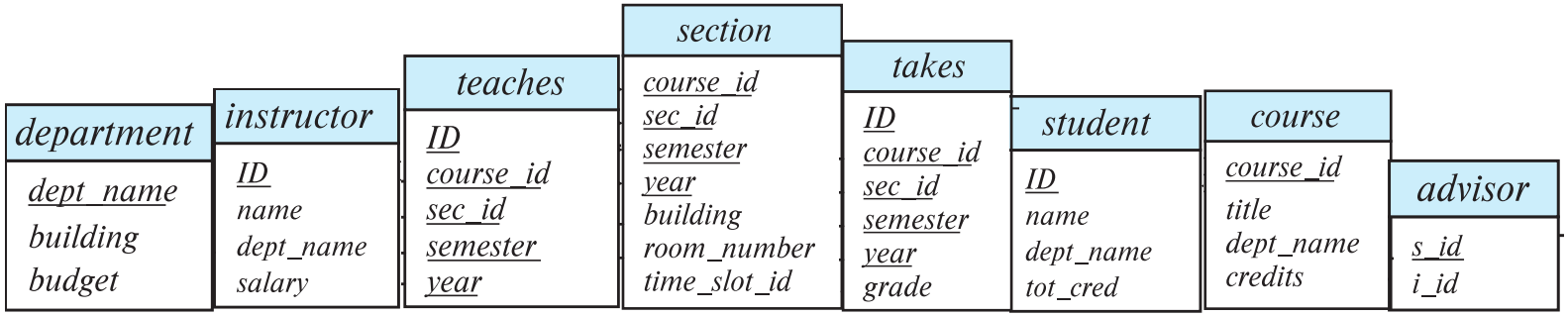


Referential Integrity

- **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.

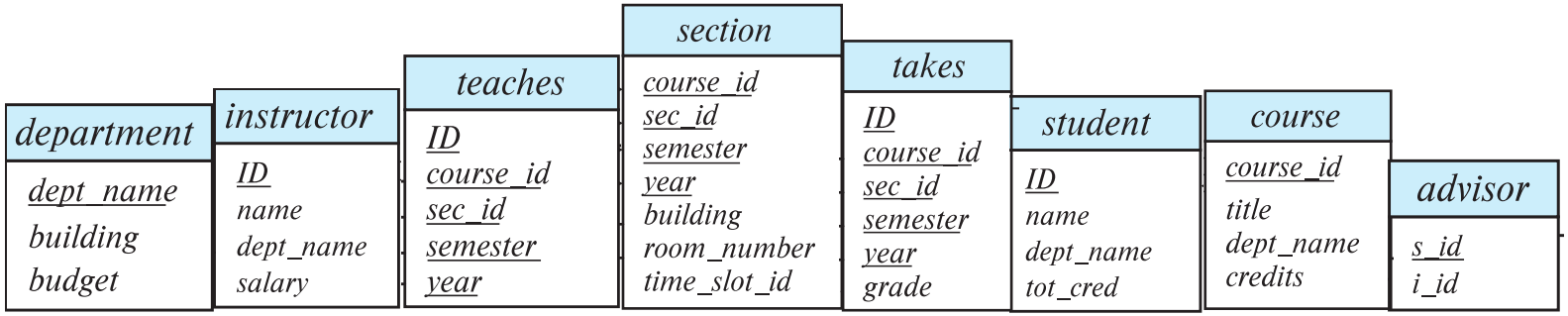


Actions in Referential Integrity

- 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

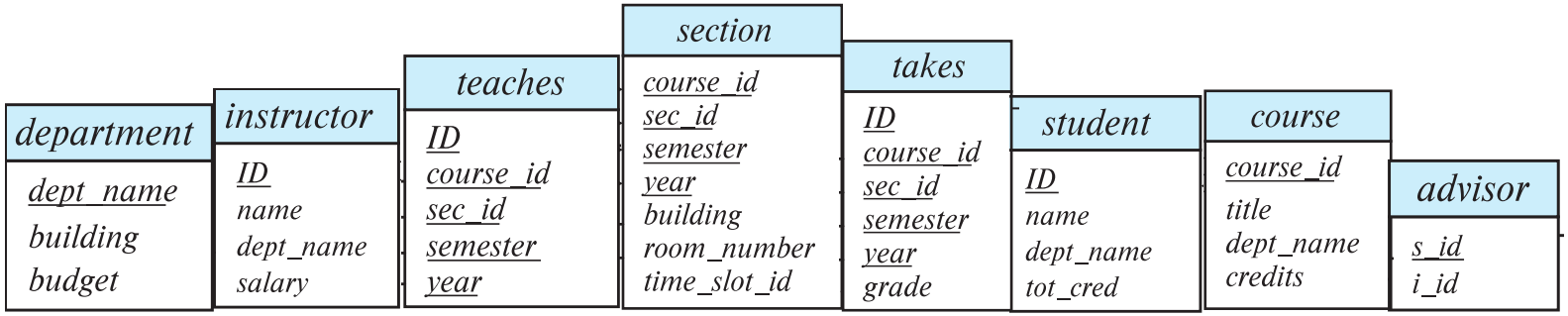
```
create table course (
        (…
         dept_name varchar(20),
         foreign key (dept_name) references department
              on delete cascade
              on update cascade,
          . . .)
```

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



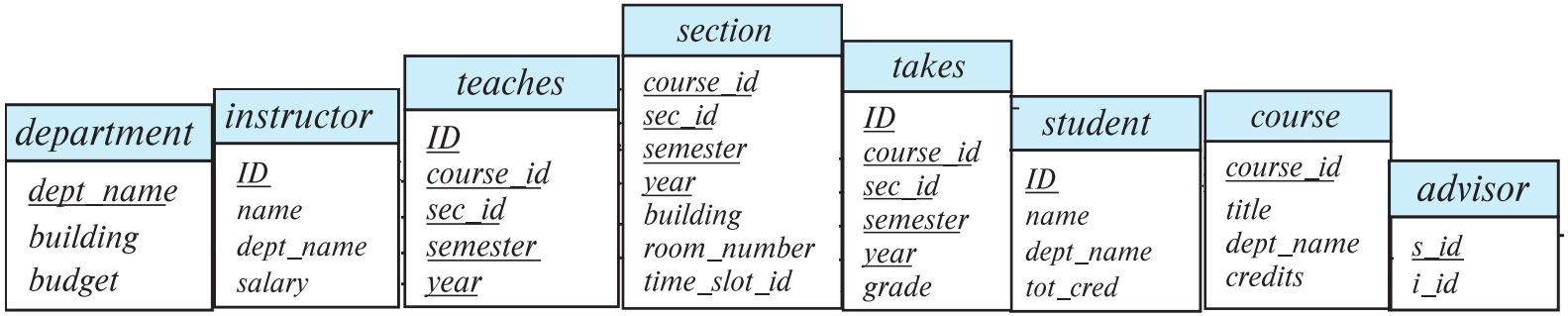
Integrity Constraint Violation

- **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



Built-in Data Types in SQL

- **date:** Example: **date** '2005-7-27'
- **time:** Example: **time** '09:00:30' **time** '09:00:30.75'
- **Timestamp** Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** Subtracting a date/time/timestamp value from another gives an interval value
- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object
  - **clob/text**: character large object
  - When a query returns a large object, a pointer is returned rather than the large object itself.



Index Creation

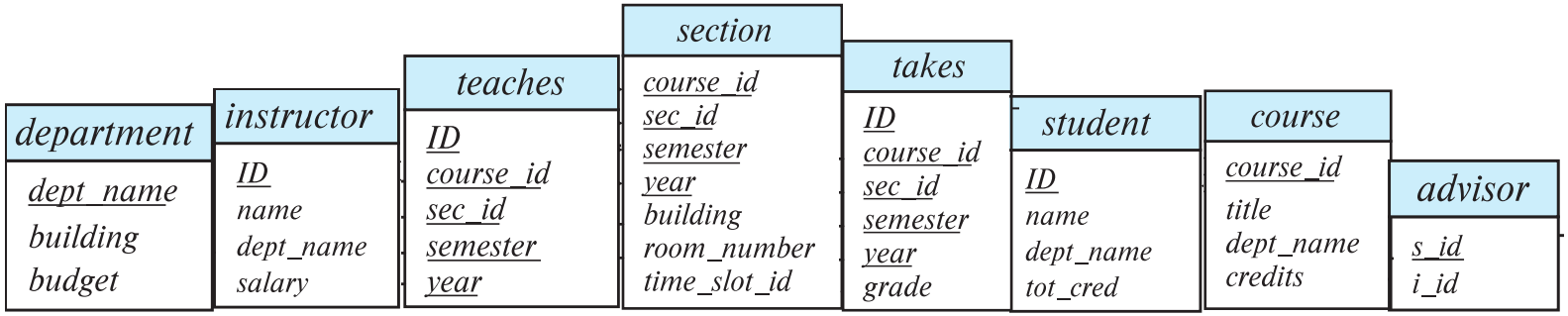
- We create an index with the **create index** command

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

- **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'
```

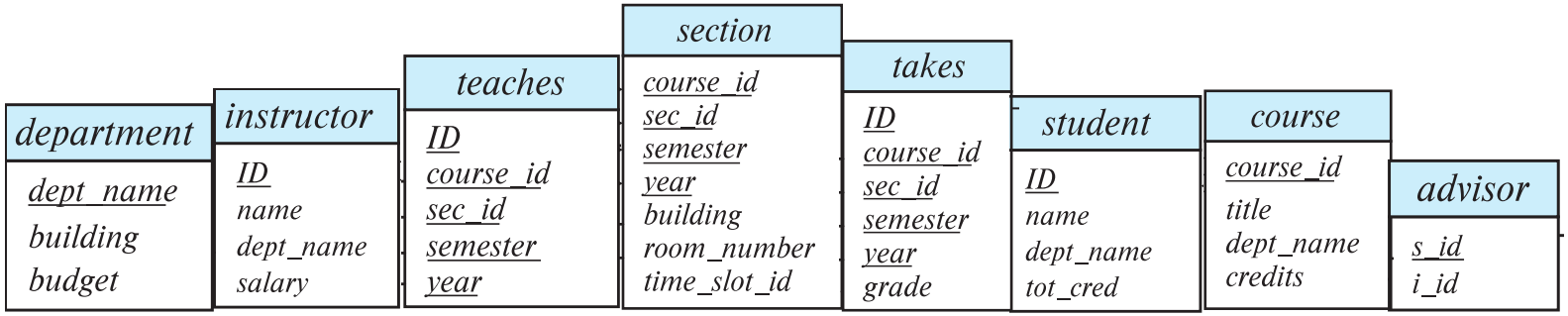
Demo: EXPLAIN QUERY PLAN



Transactions

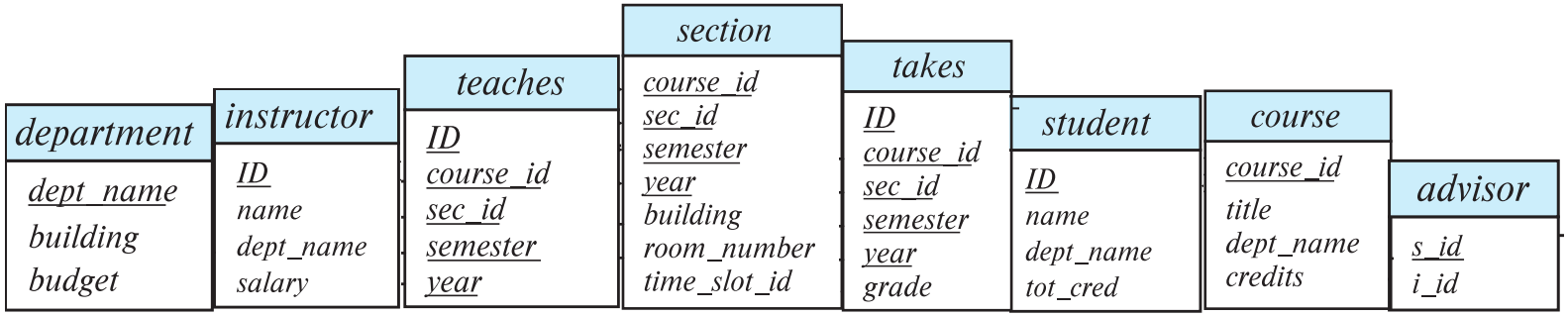
- The transaction must end with one of the following statements:
  - **Commit work**.
  - **Rollback work**.
- **A**tomic transaction
- **I**solation from concurrent transactions
- ACID
  - **C**onsistency and **D**urability.

- Demo: transaction



Authorization

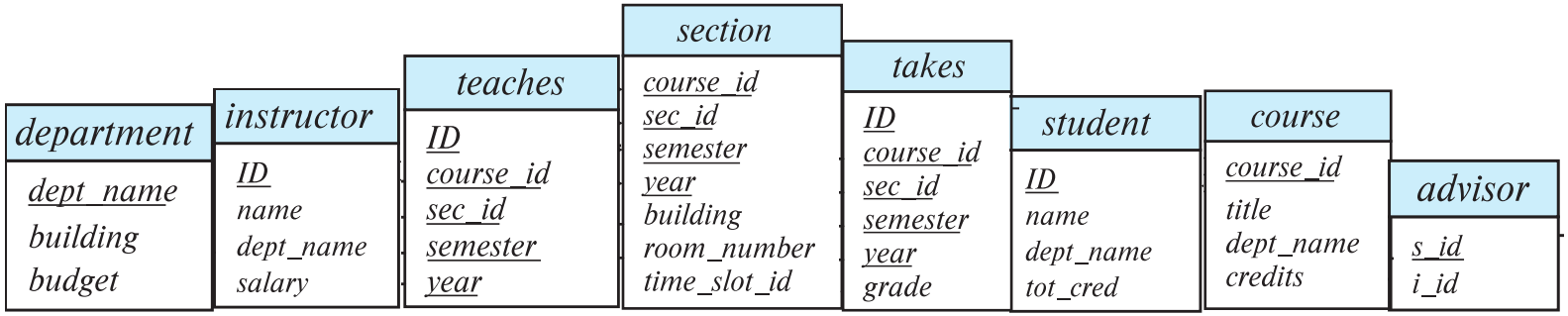
- Make sure that the users see the data that they are supposed to see
- Guard the database against modifications by malicious users
- Users have **privilege**; user can only operate on data they are authorized



Privileges in SQL

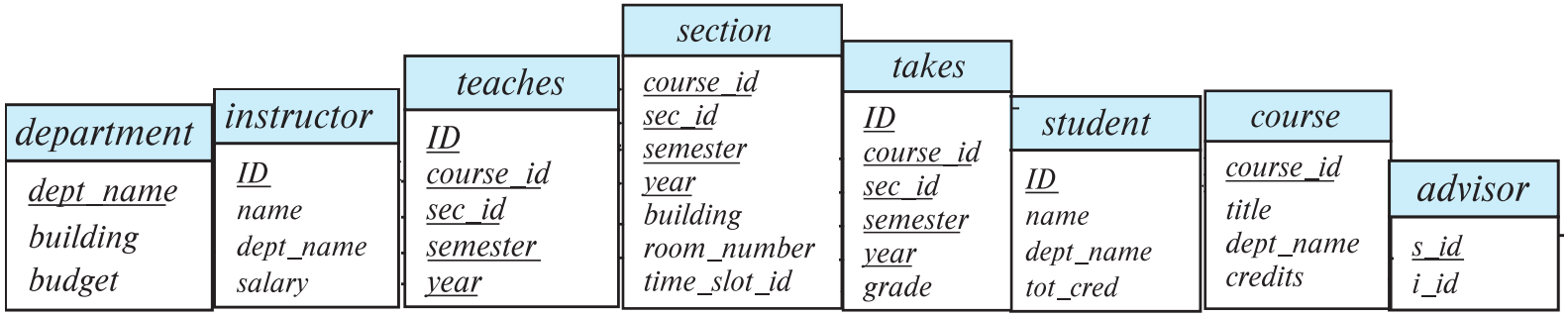
- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

    grant select on *instructor* **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



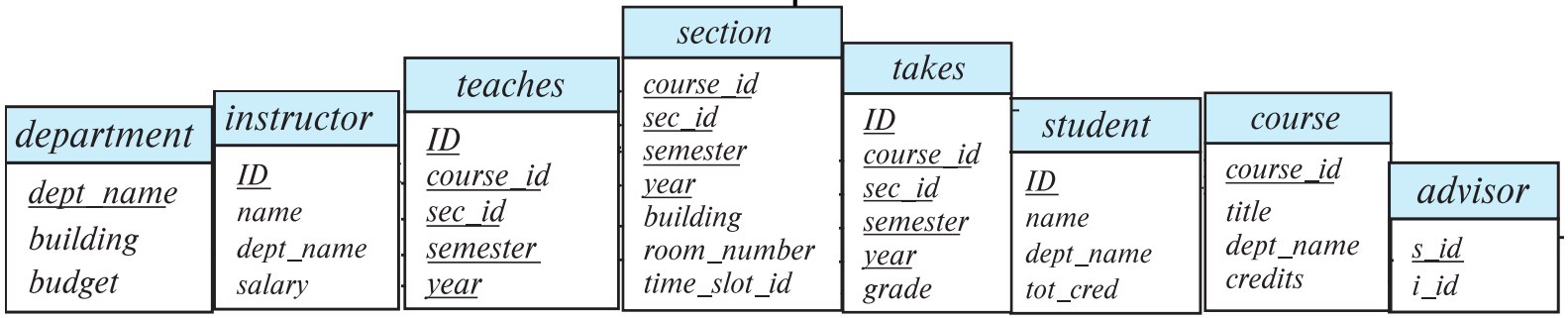
Authorization

- **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**);



Authorization to select a set of tuples/attributes

- Authorization to a user to instructors from Geology
- **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 *instructor?*

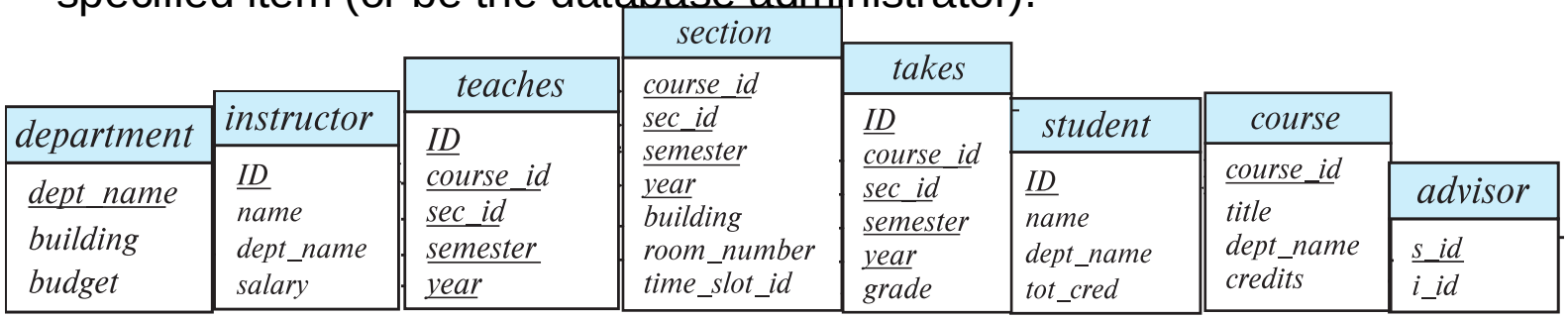


Authorization Specification in SQL

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

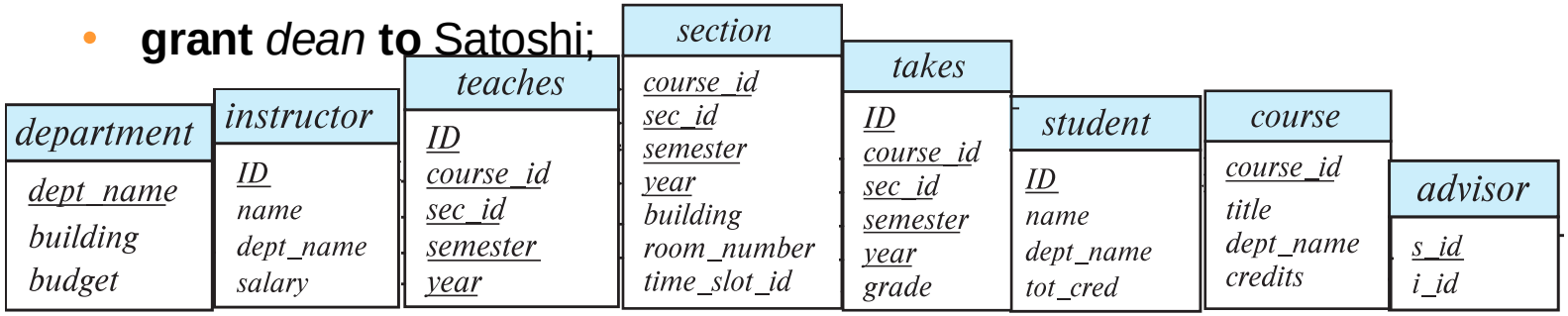
  grant <privilege list> **on** <relation or view > **to** <user list>

- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role
- 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).



Roles Example

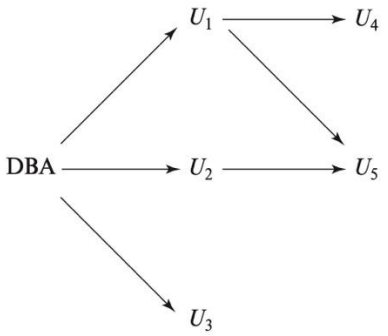
- **create role** instructor;
- **grant** *instructor* **to** Amit**;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;



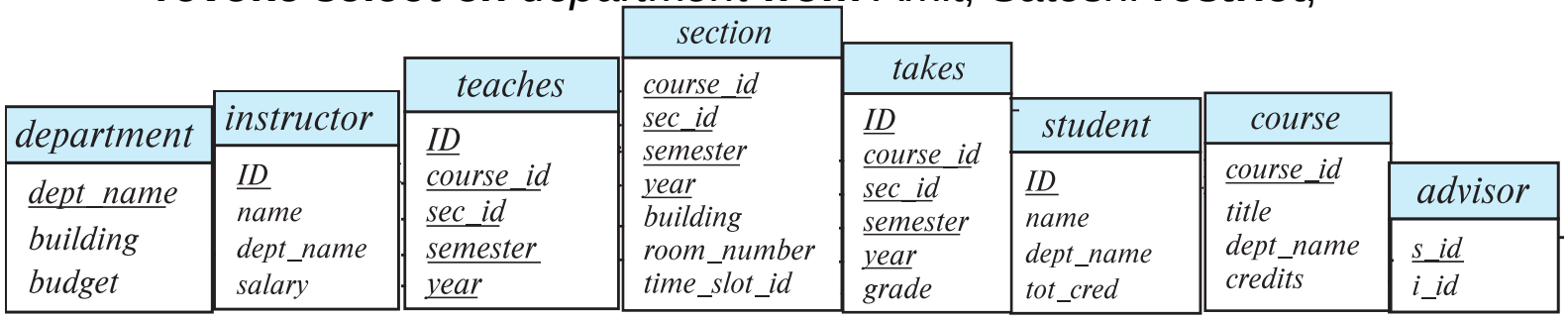
Authorization

- Relation creator is the owner
- Owner has all privileges and may grant privileges

  Example: grant select on *department* **to** Amit, Satoshi
- Grant the same or lesser privileges to other users



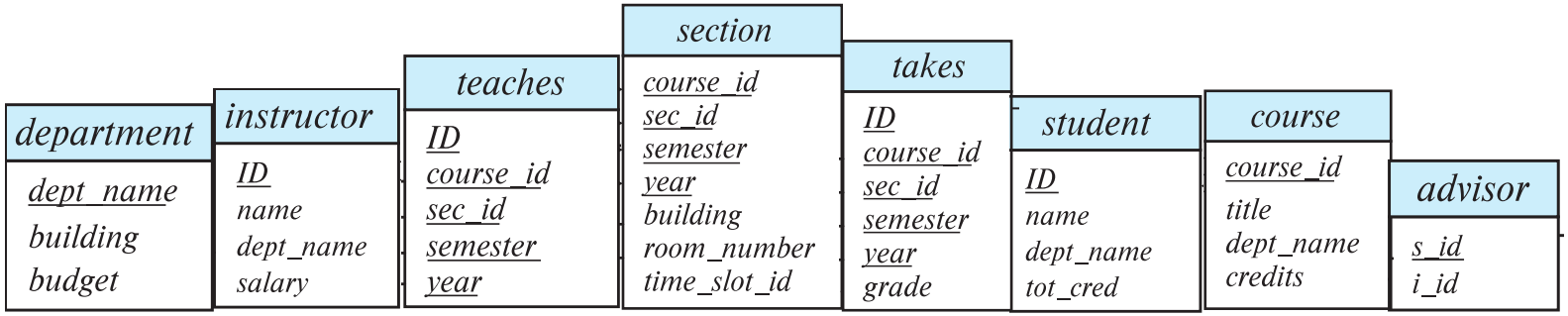
- Revoke privileges
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;



Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:

  create a role <name>
- Example:
  - **create role** *geo_staff*
- Once a role is created we can assign "users" to the role using:
  - **grant** <role> **to** <users>



Revoking Authorization in SQL

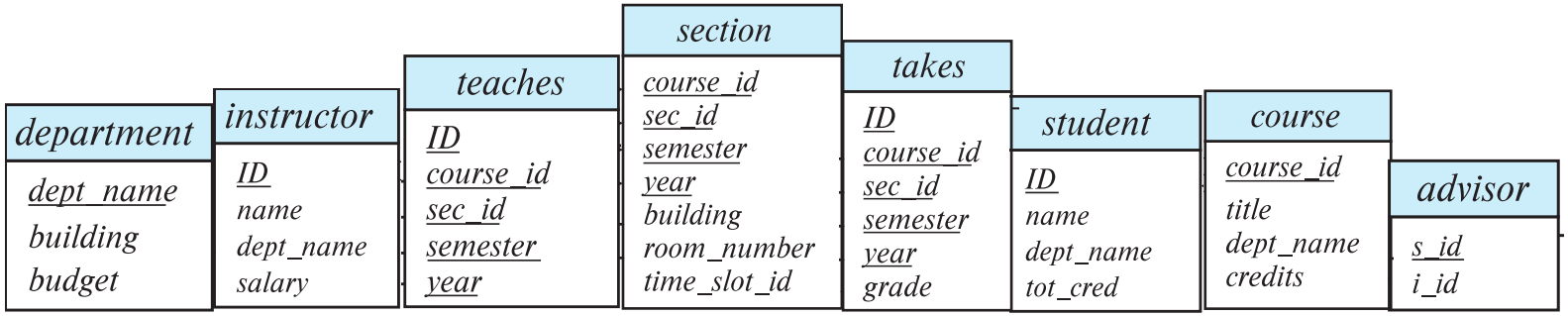
- The **revoke** statement is used to revoke authorization.

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

- Example:

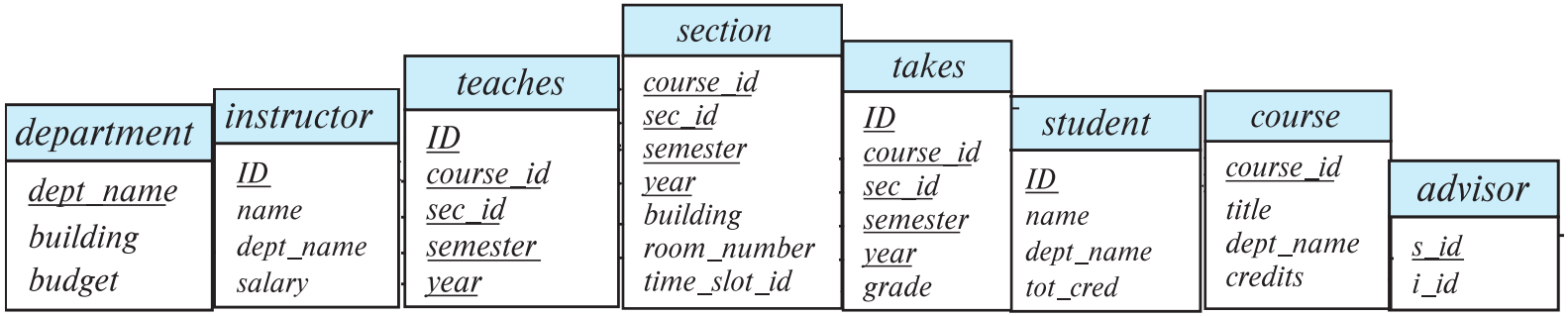
  revoke select on *student* **from** U_1, U_2, U_3

- <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.



Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - Why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict.**



Wrap up

- **Basic Queries**: SELECT A1, A2, ..., An FROM r1, r2, ..., rm WHERE P
- **String Operations**: LIKE '%pattern%'. Wildcards: _ (single character), % (multiple characters)
- **Aggregation**
  - GROUP BY A1, A2, ..., An
  - Filtering groups: HAVING P
- **Nested Queries**
  - Used in SELECT, FROM, WHERE clauses
- **Data Modification**
  - INSERT, DELETE, UPDATE
- **Joins**
  - NATURAL JOIN
  - INNER JOIN
  - (LEFT, RIGHT, FULL) OUTER JOIN
  - JOIN ... ON condition
  - JOIN ... USING (column)
- **Data Types**: Ensuring correct storage formats
- **Views**: Virtual tables for abstraction
- **Integrity Constraints**: Enforcing data consistency
- **Authorization**: Managing user access and permissions
  - GRANT (Assign privileges)
  - REVOKE (Remove privileges)

Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language for at least two reasons

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval.
- Model for communicating with the database:
  - Open a connection
  - Create a “statement” object
  - Execute queries using the statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
  {
    try (Connection conn = DriverManager.getConnection(
          "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
         Statement stmt = conn.createStatement();
        )
    {
         … Do Actual Work ….
    }
    catch (SQLException sqle) {
       System.out.println("SQLException : " + sqle);
    }
}
```

NOTE: Above syntax works with Java 7, and JDBC 4 onwards.
Resources opened in “try (….)” syntax (“try with resources”) are automatically closed at the end of the try block

JDBC Code (Cont.)

- Update to database

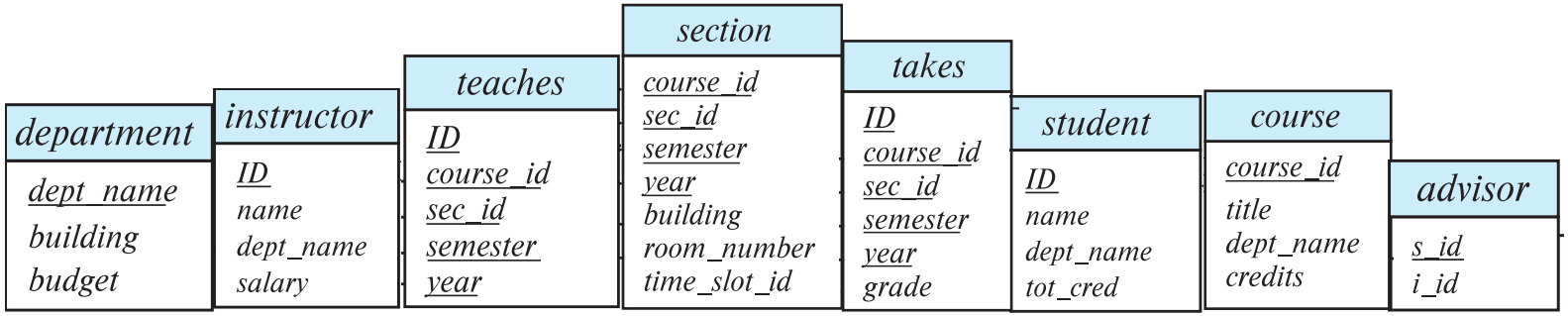
```
try {
    stmt.executeUpdate(
        "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
} catch (SQLException sqle)
{
   System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

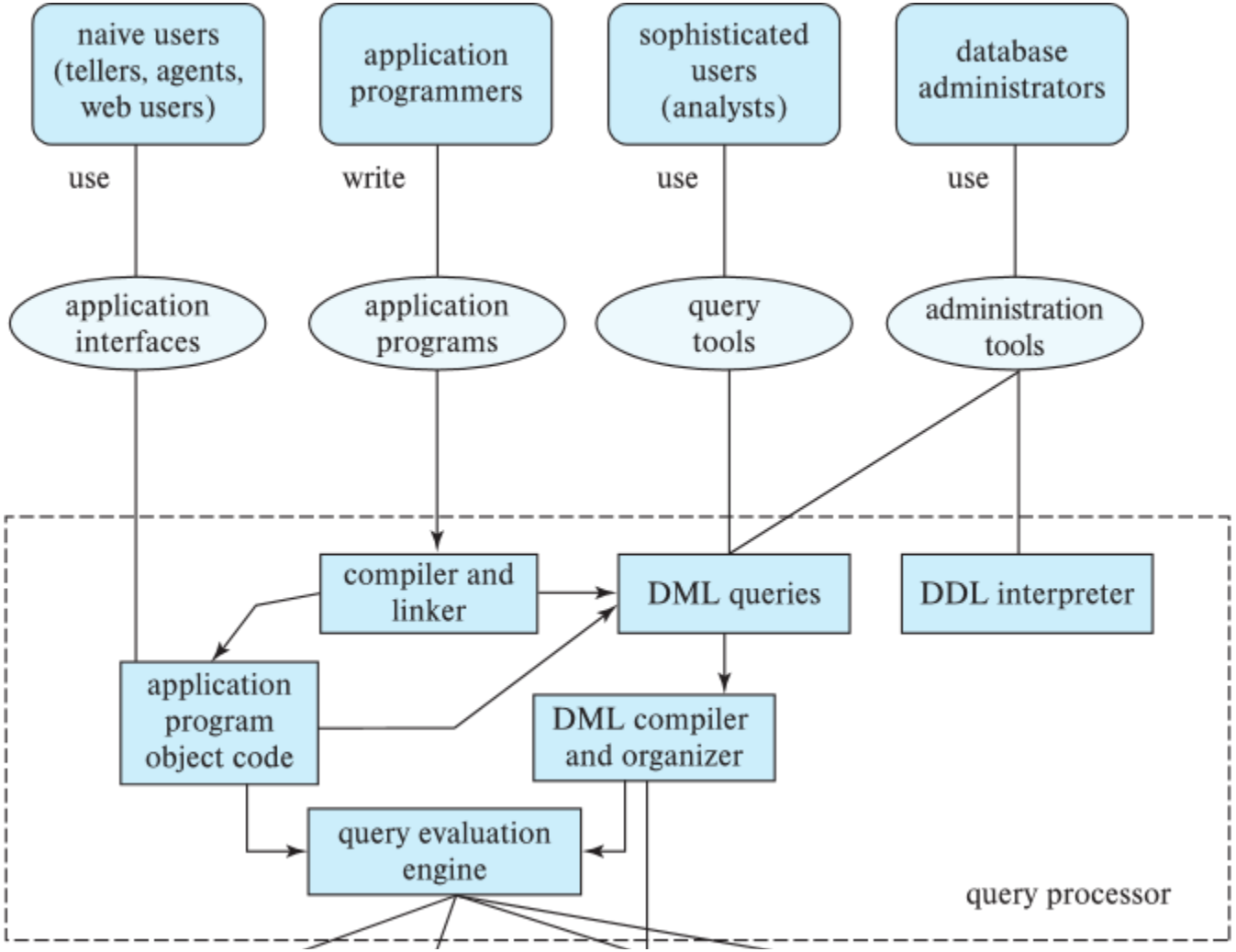
```
ResultSet rset = stmt.executeQuery(
                          "select dept_name, avg (salary)
                           from instructor
                           group by dept_name");
while (rset.next()) {
      System.out.println(rset.getString("dept_name") + " " +  rset.getFloat(2));
}
```

JDBC

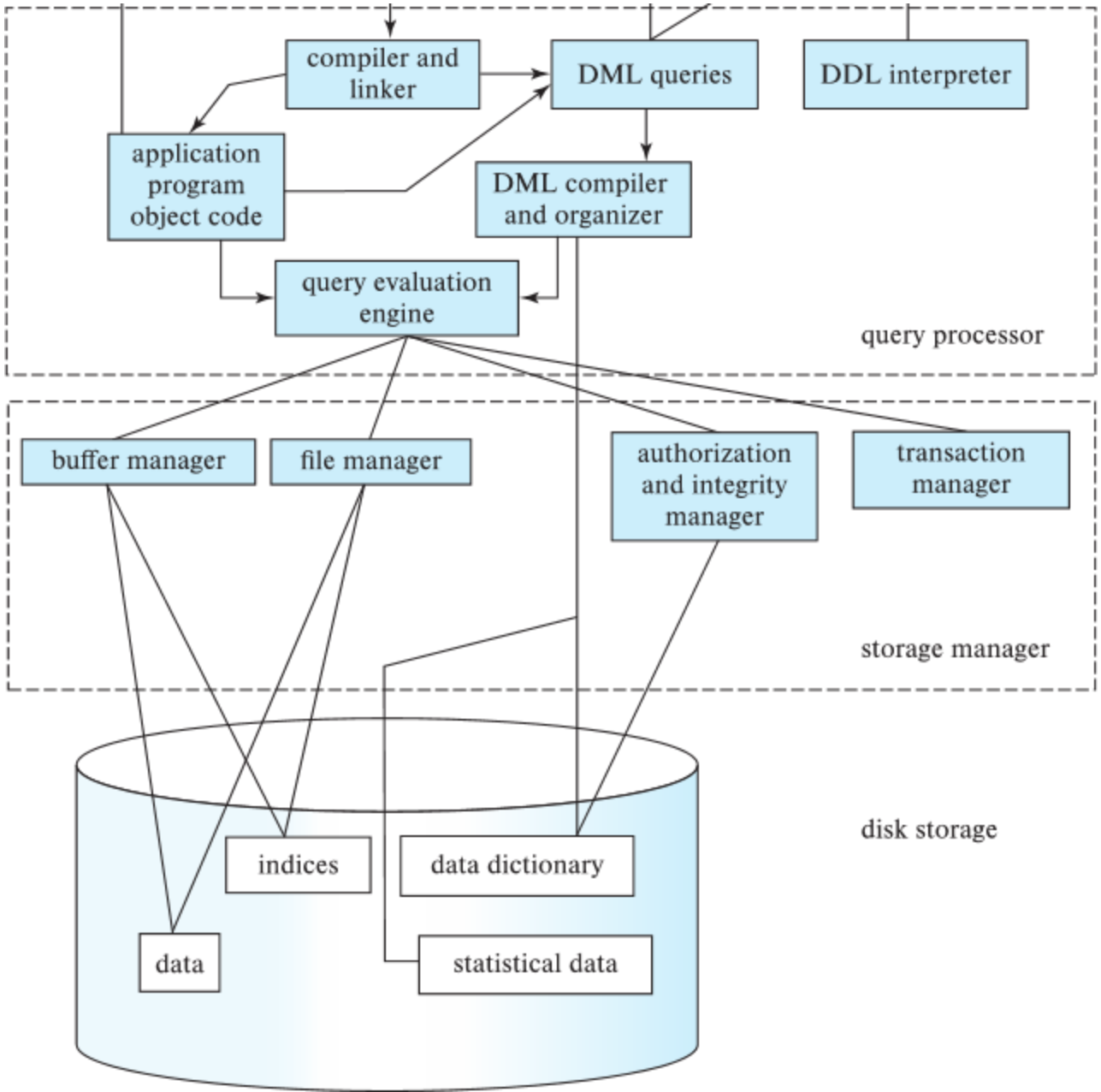
- Connecting to the Database
- Shipping SQL Statements to the Database System
- Exceptions and Resource Management
- Retrieving the Result of a Query
- Prepared Statements
- Callable Statements
- Metadata Features
- Other Features
- JDBC Basics Tutorial
  - https://docs.oracle.com/javase/tutorial/jdbc/index.html



Database Users



Database Architecture (Centralized/Shared-Memory)



THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE

FIN

Any questions?