MDA104 Introduction to Databases

4. SQL

Vlastislav Dohnal

Contents

- History of The SQL Query Language
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Join Expressions
- Views
- Modification of the Database
- Data Definition Language
  - SQL Data Types and Schemas
  - Integrity Constraints

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed to *Structured Query Language (SQL)*
- ANSI and ISO standard SQL:
  - SQL-86; SQL-89
  - SQL-92
  - SQL:1999 (recursive queries, triggers, Y2K compliant!)
  - SQL:2006 (better XML support, XQuery, …)
  - SQL:2008
  - SQL:2011 (adds support for temporal databases)
  - SQL:2016 (operation on JSON in a varchar attribute)
  - SQL:2019 (multidimensional arrays)
  - SQL:2023 (JSON data type, Property Graph Queries (SQL/PGQ))
- Commercial systems offer most SQL-99 features
  - plus, varying feature sets from later standards and special proprietary features
  - sometime varying in syntax.

Basic Query Structure

- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where C

  - A_i represents an attribute
  - R_i represents a relation
  - C is a condition.
- The result of an SQL query is a relation.

The select Clause

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example:
  - Relation

    instructor (*id, name, dept_name, salary*)
  - Find the names of all instructors:

    select *name*
    from *instructor*
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g., *Name* ≡ *NAME* ≡ *name*
  - Some people use upper case wherever we use bold font.

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates,
  - insert the keyword **distinct** after select**.**

- Find the names of all departments of instructors, and remove duplicates

  select distinct *dept_name*
  from *instructor*

- The keyword **all** specifies that duplicates not to be removed.

  select all *dept_name*
  from *instructor*

  - It is also an implicit behavior when the keyword all is omitted.

The select Clause (Cont.)

- Relation

  instructor (*id, name, dept_name, salary*)

- An asterisk in the select clause denotes “all attributes”

  select *
  from *instructor*

- The **select** clause can contain arithmetic expressions
  - Involving the operations: +, –, ∗, and /,
  - Operating on constants or attributes of tuples.
  - Also function can be used (nullif(), upper(), to_char(), …)
- The query:

  select *id, name, dept_name, salary/12*
  from *instructor*

  would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.

- To find all instructors in 'Comp. Sci.' department with salary > 80000

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.' **and** *salary* > 80000

- Comparison results can be combined using the logical connectives
  - **and, or, not**

- Comparisons can be applied to results of arithmetic expressions.

  select *name*
  from *instructor*
  where *salary* / 12 > 6000

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor* $\times$ *teaches*

select $*$
from *instructor, teaches*

  - Generates every possible instructor-teaches pair, with all attributes from both relations.
- Cartesian product not very useful directly,
  - but useful when combined with a where-clause condition (selection operation in relational algebra).

Cartesian Product

instructor

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

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

instructor × *teaches*

Inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…
12121	Wu	Physics	95000	10101	CS-101	1	Fall	2009
12121	Wu	Physics	95000	10101	CS-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
…	…	…	…	…	…	…	…	…

Joins

- Relations:
  - *instructor* (*id*, *name*, *dept_name*, *salary*)
  - *course* (*course_id*, *title*, *dept_name*)
  - *section* (*course_id, sec_id*, *semestr*, *year*)
  - *teaches* (*id*, *course_id*, *sec_id*)
- For all instructors who teach courses, find their names and the course id of the courses they teach.

  select *name, course_id*
  from *instructor, teaches*
  where *instructor.id* = *teaches.id*

- Find the course id, title, semester and year of each course offered by the "Comp. Sci." department

  select *course.course_id, title, semester, year*
  from *course, teaches, section*
  where *course.course_id* = *teaches.course_id* **and**
  teaches.course_id = *section.course_id* **and**
  teaches.sec_id = *section.sec_id* **and**
  dept_name = 'Comp. Sci.'

Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column
- For relations:
  - *instructor* (*id*, *name*, *dept_name*, *salary*)
  - *teaches* (*id*, *course_id*, *sec_id*, *semestr*, *year*)
- **select** * **from** *instructor* **natural join** *teaches*;

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010

Natural Join (Cont.)

- Danger in natural join:
  - beware of unrelated attributes with same name which get equated incorrectly
- Relations:
  - *instructor* (*id*, *name*, *dept_name*, *salary*)
  - *course* (*course_id*, *title*, *dept_name*)
  - *section* (*course_id*, *sec_id*, *semester*, *year*)
  - *teaches* (*id*, *course_id*, *sec_id*)
- List the names of instructors along with the titles of courses that they teach.
  - Incorrect version (equates *course.dept_name* with *instructor.dept_name*)
    - **select** *name*, *title*
from (*instructor* **natural join** *teaches*) **natural join** *course*;
  - Correct version
    - **select** *name*, *title*
from (*instructor* **natural join** *teaches*), *course*
where *teaches*.*course_id*= *course*.*course_id*;
  - Another correct version
    - **select** *name*, *title*
from (*instructor* **natural join** *teaches*) **join** *course* **using**(*course_id*);

The Rename Operation

- The SQL allows renaming <u>relations</u> and <u>attributes</u> using the **as** clause:

  old-name **as** *new-name*

- E.g.,
  - **select** *id, name, salary/12* **as** *monthly_salary*
    from *instructor*

- Find the names of all instructors who have a salary higher than some instructor in ‘Comp. Sci.’
  - **select distinct** *T.name*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary* > *S.salary* **and** *S.dept_name* = *‘Comp. Sci.’*

- Keyword **as** is optional and may be <u>omitted in renaming relations</u>

  instructor **as** *T* ≡ *instructor T*

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors
  select *name*
  from *instructor*
  order by *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute.
  - Ascending order is the default.
  - Example: … **order by** *name* **desc**

- Can sort on multiple attributes
  - Example: … **order by** *dept_name, name*
    or … **order by** *dept_name* **desc**, *name* **asc**

Where Clause Predicates

- SQL includes a **between** comparison operator
- Example:
  - Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, $\geq$ \$90,000 and $\leq$ \$100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000

- Tuple comparison
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor*.*ID*, *dept_name*) = (*teaches*.*ID*, 'Biology');

String Operations

- SQL includes a string-matching operator for comparisons on character strings.
  - 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 includes the substring "dar".

  select *name*
  from *instructor*
  where *name* **like** '%dar%'

- Match the string containing "100 %"

  … **like** '%100 \%%' **escape** '\'

- SQL supports a variety of string operations such as
  - concatenation (using "**||**")
  - converting from upper to lower case (and vice versa)
    - functions **upper**() and **lower**()
  - finding string length, extracting substrings, etc.

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an *unknown* value or that a value does not exist.

- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns *null*

- The predicate **is null** can be used to check for *null* values.
  - Example: Find all instructors whose salary is null.

    select *name*
    from *instructor*
    where *salary* **is null**

Null Values and Three-valued Logic

- Any comparison with *null* returns *null*
  - Example*: 5 < null   or   null <> null   or   null = null*
- Three-valued logic using the truth value *null*:
  - OR: (*null* **or** *true*)   = *true*
    (*null* **or** *false*) = *null*
    (*null* **or** *null) = null*
  - AND: *(true* **and** *null)* = *null*
    (false **and** *null)* = *false*
    (null **and** *null)* = *null*
  - NOT*:* (**not** *null)* = *null*
- Result of **where** clause predicate is treated as *false* if it evaluates to *null*

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators
  - Given multiset relations r_1 and r_2:

1. $\boldsymbol{\sigma_\theta(r_1)}$**:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies the selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
2. $\boldsymbol{\Pi_A(r)}$**:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$
   where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.
3. $\boldsymbol{r_1 \times r_2}$**:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \cdot c_2$ copies of the tuple $t_1 . t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

- Example:
  Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:

  $r_1 = \{(1, a), (2,a)\}$ $\qquad r_2 = \{(2), (3), (3)\}$
- Then $\Pi_B(r_1)$ would be
  {(a), (a)}
- While $\Pi_B(r_1) \times r_2$ would be

  $\{(a,2), (a,3), (a,3), (a,2), (a,3), (a,3)\}$
- SQL duplicate semantics:

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

  is equivalent to the *multiset* version of the rel. alg. expression:

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

Set Operations (union, intersect, except)

- Relation:

  teaches (*id, course_id, sec_id, semester, year*)

- Find courses that ran in Fall 2009 or in Spring 2010

(**select** *course_id* **from** *teaches* **where** *semester* = ‘Fall’ **and** *year* = 2009)
union
(**select** *course_id* **from** *teaches* **where** *semester* = ‘Spring’ **and** *year* = 2010)

- Find courses that ran in Fall 2009 and in Spring 2010

(**select** *course_id* **from** *teaches* **where** *semester* = ‘Fall’ **and** *year* = 2009)
intersect
(**select** *course_id* **from** *teaches* **where** *semester* = ‘Spring’ **and** *year* = 2010)

- Find courses that ran in Fall 2009 but not in Spring 2010

(**select** *course_id* **from** *teaches* **where** *semester* = ‘Fall’ **and** *year* = 2009)
except
(**select** *course_id* **from** *teaches* **where** *semester* = ‘Spring’ **and** *year* = 2010)

Set Operations

- Set operations **union, intersect,** and **except**
  - Each of the above operations automatically eliminates duplicates

- To retain all duplicates, use the corresponding multiset versions
  - **union all, intersect all** and **except all.**

- Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:
  - $m + n$ times in *r* **union all** *s*
  - min(m, n) times in *r* **intersect all** *s*
  - max(0, $m - n$) times in *r* **except all** *s*

Aggregate Functions

- 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

Aggregate Functions (Cont.)

- Relations:
  - *instructor* (*id, name, dept_name, salary*)
  - *teaches* (*id, course_id, sec_id, semestr, year*)
- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= 'Comp. Sci.';

$$\mathcal{G}_{avg(salary)}\left(\sigma_{dept_{name}='Comp.Sci.'}(instructor)\right)$$

- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **select count** (**distinct** *id*)
    from *teaches*
    where *semester* = 'Spring' **and** *year* = 2010
- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;

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

dept_name	*avg*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions <u>must</u> appear in **group by** list
  - Erroneous query:
    select *dept_name*, *id*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;

Aggregate Functions – Having Clause

- Relations:
  - *instructor* (*id, name, dept_name, salary*)

- Find the names and average salaries of all departments whose average salary is greater than 42,000

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

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups.

Note2: so aggregate functions cannot be used in **where** clause.

Null Values and Aggregates

- Total all salaries

  select sum (*salary*)
  from *instructor*

  - Above statement ignores *null* amounts
  - Result is *null* if there is no non-null amount

- All aggregate operations except **count(*)** ignore tuples with *null* values on the aggregated attributes
- What if collection has only *null* values?
  - **count** returns 0
  - all other aggregates return *null*

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
  - scalar query
  - derived relation
  - nested subquery
- A common use of nested subqueries is to perform tests for
  - set membership,
  - set comparisons, and
  - set cardinality.

Example Query: set membership

- Operators: IN   NOT IN
- Relations:
  - *teaches* (*id, course_id, sec_id, semester, year*)
- Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year = 2009 and
        course_id in (select course_id
                      from section
                      where semester = 'Spring' and year = 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year = 2009 and
        course_id  not in (select course_id
                      from section
                      where semester = 'Spring' and year = 2010);
```

Example Query: set membership (cont.)

- Relations:
  - *instructor* (*id, name, dept_name, salary*)
  - *teaches* (*id, course_id, sec_id*, *semester*, *year*)
  - *takes* (*id, course_id, sec_id*, *semester*, *year*)
  - *student* (*id*, *name*)

Attribute ID is a reference to *student*, here.

- Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in
                              (select course_id, sec_id, semester, year
                               from teaches
                               where teaches.ID = 10101);
```

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

Set Comparison

- Relations:
  - *instructor* (*id, name, dept_name, salary*)

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

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

- Same query using **> some** clause

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

Definition of **some** Clause

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

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

Read: 5 < some tuple in the relation

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

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

(5 != **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true (since 0 ≠ 5)

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

any is an equivalent of **some**

Definition of **all** Clause

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

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

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

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

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

(!= **all**) $\equiv$ **not in**

However, (= **all**) $\not\equiv$ **in**

Set comparison and NULL values

	0	
(5 **in**	5	) = true
	null	

	0	
(2 **in**	5	) = false
	null	

	0	
(2 **not in**	5	) = false!!!
	null	

	0	
(null **in**	5	) = false
	6	

	0	
(null **in**	5	) = false!!!
	null	

Example Query

- Relations:
  - *instructor* (*id, name, dept_name, salary*)
- Find the names of instructors whose salary is greater than the salary of <u>all</u> instructors in the Biology department.

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

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists (** r **)** $\Leftrightarrow$ $r \neq \emptyset$
- **not exists (** r **)** $\Leftrightarrow$ $r = \emptyset$

Correlation Variables

- Relations:
  - *section* (<u>*course_id*</u>, <u>*sec_id*</u>, *semestr, year*)
- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"

select *course_id*
from *section* **as** *S*
where *semester* = 'Fall' **and** *year* = 2009 **and**
    exists (**select** *
        from *section* **as** *T*
        where *semester* = 'Spring' **and** *year* = 2010
            and *S.course_id* = *T.course_id*);

- **Correlated subquery**
- **Correlation name** or **correlation variable**

Not Exists

- Relations:
  - *student* (*id, name*)
  - *takes* (*id, course_id, sec_id*, *semester*, *year*)
  - *course* (*course_id, title, dept_name*)
- Find 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));
```

- Remark that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using **= all** and its variants

Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- Find the departments where the average salary is greater than $42,000. Print the average salary too.

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

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

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

Scalar Subquery

- Relations:
  - *instructor* (*id, name, dept_name, salary*)
  - *department* (*dept_name*, *building, budget*)

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

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 sub-query expressions in the **from** clause.

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.

Left Outer Join

- *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

prereq

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

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

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

Right Outer Join

- *course* *prereq*

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

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

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

Full Outer Join

- *course* *prereq*

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

- *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

Joined Relations

- **Join operations** take two relations and return as a result another relation.
  - These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **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 type
inner join
left outer join
right outer join
full outer join

Join condition	**Usage**
natural	r_1 natural <join_type> r_2
on <predicate>	r_1 <join_type> r_2 on <predicate>
using $(A_1,A_2,\ldots A_n)$	r_1 <join_type> r_2 using $(A_1,A_2,\ldots,A_n)$

Joined Relations – Examples

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

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

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

course_id	*title*	*dept_name*	*credits*	*course_id*	*prereq_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

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructor's name and department, but not the salary. This person should see a relation described, in SQL, by

  select *id*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

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

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

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

Example Views

- A view of instructors without their salary

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

- Find all instructors in the Biology department

  select *name*
  from *faculty*
  where *dept_name* = 'Biology'

- Create a view of department salary totals

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

Views Defined Using Other Views

- **create view** *physics_fall_2009* **as**
  select *course*.*course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course*.*course_id* = *section*.*course_id*
  and *course*.*dept_name* = ’Physics’
  and *section*.*semester* = ’Fall’
  and *section*.*year* = ’2009’;

- **create view** *physics_fall_2009_watson* **as**
  select *course_id*, *room_number*
  from *physics_fall_2009*
  where *building* = ’Watson’;

View Expansion

- Expand use of a view in a query/another view

```
create view physics_fall_2009_watson as
  select course_id, room_number
  from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2009')
  where building = 'Watson';
```

View Expansion

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

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

- As long as the view definitions are not recursive, this loop will terminate.
  - Recursive views/queries are typically limited to the construct:
    - WITH RECURSIVE myquery (A, B, …) AS (
      SELECT A, B, … FROM table WHERE …
      UNION
      SELECT A, B, … FROM myquery, table, …
) SELECT * FROM myquery

Non-recursive part of query

Recursive part

Modification of the Database – Deletion

- Relations:
  - *instructor* (*id, name, dept_name, salary*)
  - *department* (*dept_name, building, budget*)
- 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');

Example Query

- Relations:
  - *instructor* (*id, name, dept_name, salary*)

- 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:
  - First, compute **avg** salary and find all tuples to delete
  - Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- Relations:
  - *course* (*course_id*, *title*, *dept_name*, *credits*)

- Add a new tuple to *course*

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

- or equivalently (this is a recommended variant!)

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

Modification of the Database – Insertion

- Relations:
  - *student* (*id*, *name*, *dept_name*, *tot_credits*)

- Add a new tuple to *student* with *tot_credits* set to *null*
  - **insert into** *student*
    values (’3003’, ’Green’, ’Finance’, *null*);
- or equivalently
  - **insert into** *student* (*id*, *name*, *dept_name*)
    values (’3003’, ’Green’, ’Finance’);
  - The value for the unspecified attribute is automatically set to *null*
    - or the default value assigned to the attribute is used instead.

Modification of the Database – Insertion

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

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

- The **select-from-where** statement is evaluated fully before any of its results are inserted into the relation
  - Otherwise queries like this would cause problems

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

Modification of the Database – Updates

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

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

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

Case Statement for Conditional Updates

- Same query as before but with case statement

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

Updates with Scalar Subqueries

- Re-compute and update *tot_credits* value for all students

update *student*
  set *tot_credits* = (**select sum**(*credits*)
                from *takes* **natural join** *course*
                where *student*.*ID*= *takes*.*ID* **and**
                    takes.*grade* <> 'F' **and**
                    takes.*grade* **is not null**);

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

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

- Or, use the function **COALESCE**
  … (**select** coalesce(sum(*credits*), 0) **from** …

Modification of the Database – Views

- Modifications of views must be translated to modifications of the actual relations in the database.
- Consider the view *faculty* where instructors' salary is hidden:

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

Recall: instructor (*id, name, dept_name, salary*)

- Since we allow a view name to appear wherever a relation name is allowed, the user may write:

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

Modification of the Database – Views (cont.)

- The previous insertion must be represented by an insertion into the actual relation *instructor* from which the view *faculty* is constructed.

- An insertion into *instructor* requires a value for *salary*. The insertion can be dealt with by either
  - rejecting the insertion and returning an error message to the user; or
  - inserting the tuple

    ('3003', 'Green', 'Finance', *null*)

    into the *instructor* relation.

Data Definition Language

- Allows the specification of not only a set of relations but also information about each relation, including:
  - The schema for each relation.
  - The domain of values associated with each attribute.
  - Integrity constraints
  - The set of indices to be maintained for each relation.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  integrity-constraint$_1$,
  ...,
  integrity-constraint$_k$)

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of values in the domain of attribute A_i

- Example:

  create table *instructor* (
  ID **char**(5),
  name **varchar**(20)**,**
  dept_name **varchar**(20),
  salary **numeric**(8,2),
  primary key (id))

  - **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
  - **insert into** *instructor* **values** (‘10211’, null, ’Biology’, 66000);

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.
- **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.

Domain Types in SQL (cont.)

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** ‘2005-07-27’
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** ‘09:00:30’ **time** ‘09:00:30.75’
- **timestamp**: date plus time of day
  - Example: **timestamp** ‘2005-07-27 09:00:30.75’
- **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

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - 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**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

Not Null and Unique Constraints

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

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**
- **primary key** (A_1, A_2, …, A_m)
  - Attributes A_1, A_2, … A_m forms the relation’s primary key.
  - Equals to unique and not null.
- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the values in attributes A_1, A_2, … A_m cannot repeat within the relation.

The check clause

- **check** (P)

  where P is a predicate

Example: Ensure that semester is one of fall or spring:

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', 'Spring'))
);

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another 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".

- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S.

  E.g.: S(<u>A</u>,…) R(<u>X</u>, …, A, …)

  A is said to be a **foreign key** of **R** if for any value of **A** appearing in **R** it also appears in **S**.
  - $\Pi_A(R) \subseteq \Pi_A(S)$

Referential Integrity in Create Table

- **foreign key** (A_m, ..., A_n) **references** *r*

Example: Declare *dept_name* as the foreign key referencing *department* relation

create table *instructor* (
ID **char**(5),
name **varchar**(20) **not null,**
dept_name **varchar**(20),
salary **numeric**(8,2),
primary key (*ID*),
foreign key *(dept_name*) **references** *department*
);

Notice: Schema of *department* is (*dept_name, building*).

Cascading Actions in Referential Integrity

- **create table** *course* (
  course_id **char**(5) **primary key**,
  title **varchar**(20),
  dept_name **varchar**(20) **references** *department*
)
- **create table** *course* (
  …
  dept_name **varchar**(20),
  foreign key (*dept_name*) **references** *department*
  on delete cascade
  on update cascade,
  . . .
)
- Alternative actions to cascade: **set null**, **set default**
  - E.g. … ON DELETE CASCADE SET NULL

Complex Check Clauses

- Assume table *section*(<u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>, *time_slot_id, building, room_number*)
- We define this constraint:
  check (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))
  - Why not use a foreign key here?
    - If *time_slot_id* is not the primary key in *time_slot*
- Every section has at least one instructor teaching the section.
  - How to write this?
  - By a subquery…
  - Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers

Drop and Alter Table Constructs

- **drop table** *r*

  DROP TABLE *instructor*;

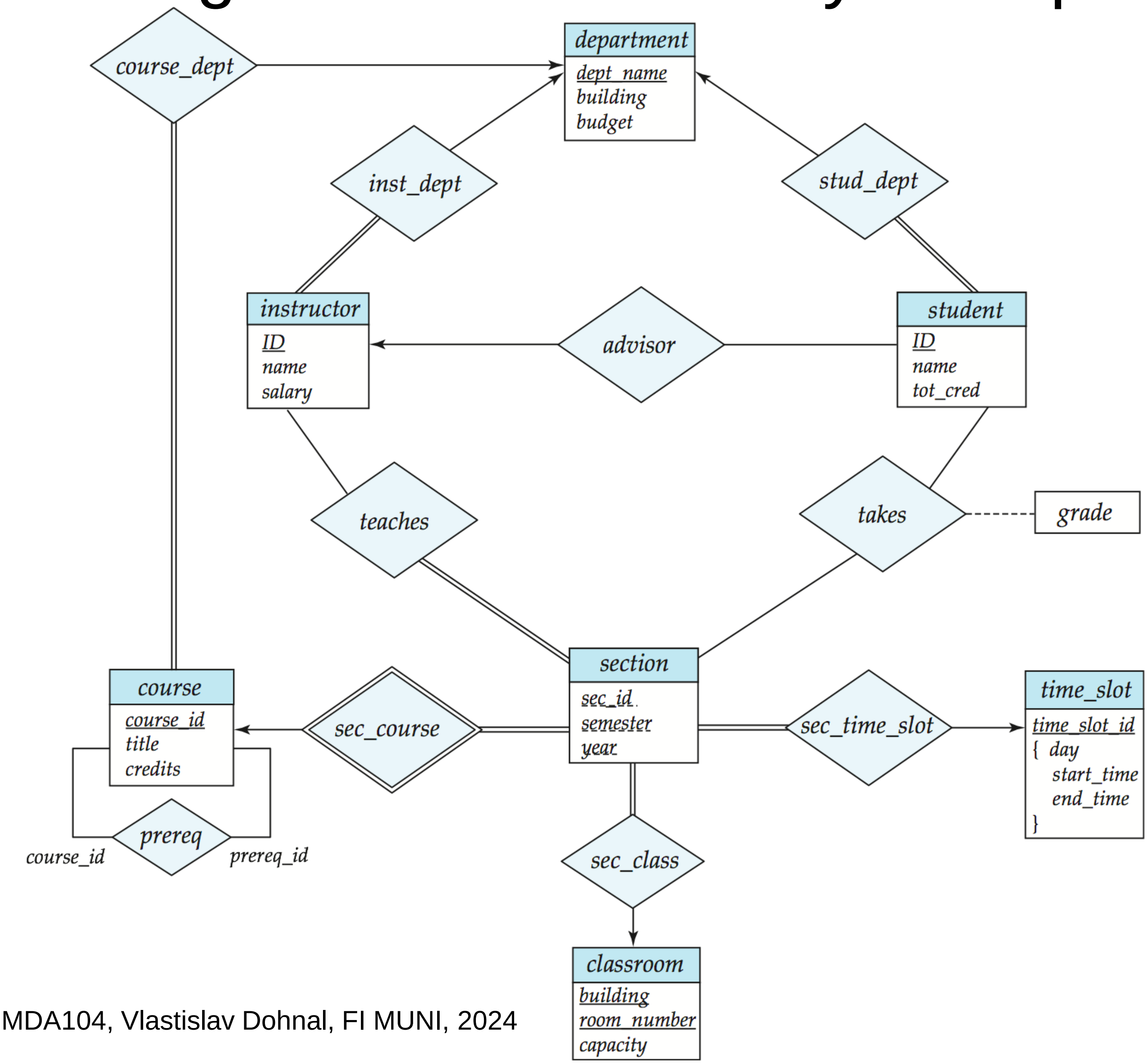
- **alter table** *r* …
  - **alter table** *r* **add** *A D*
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
    - All tuples in the relation are assigned *null* as the value for the new attribute.

    ALTER TABLE *instructor* ADD *rating* CHAR(1);

  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases.

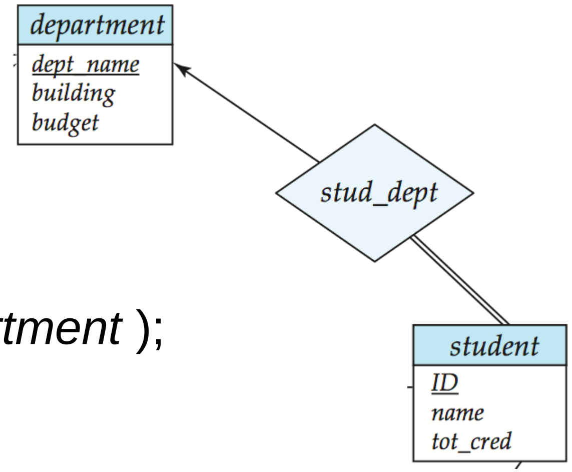
    ALTER TABLE *instructor* DROP *rating*;

E-R Diagram for a University Enterprise



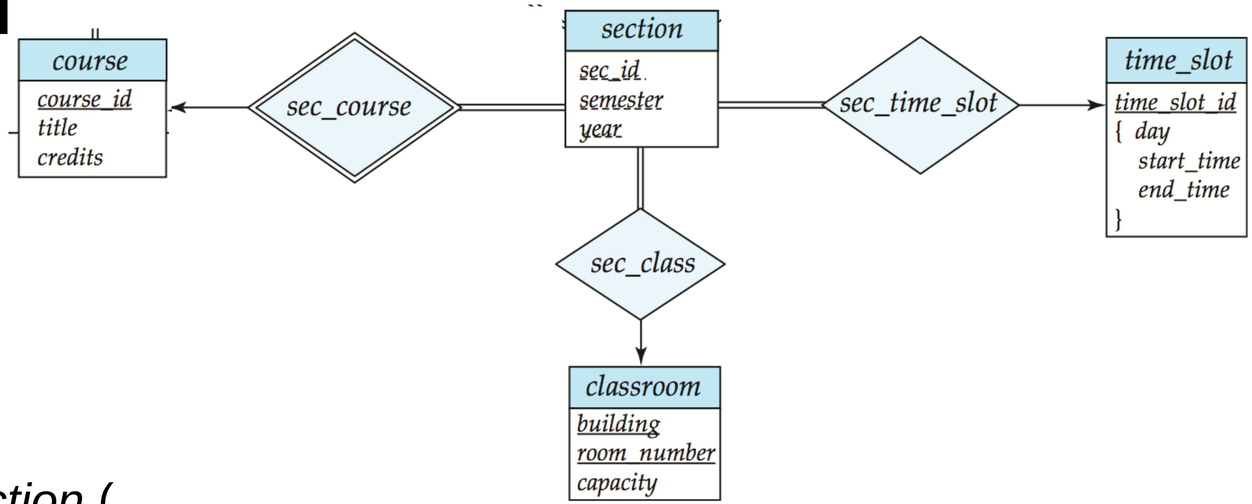
And a Few More Relation Definitions

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



- **create table** *course* (
  course_id **varchar**(8) **primary key,**
  title **varchar(**50),
  dept_name **varchar**(20) **not null**,
  credits **numeric**(2,0),
  foreign key (*dept_name*) **references** *department*);

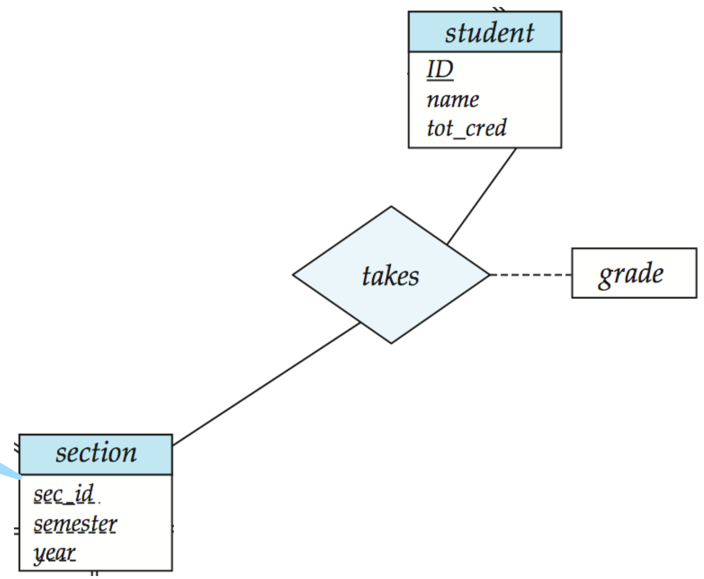
And more still



- **create table** *section* (
  course_id **varchar**(8) **references** *course*,
  sec_id **varchar**(8),
  semester **varchar**(6),
  year **numeric**(4,0),
  building **varchar**(8),
  room_number **varchar**(8),
  time_slot_id **integer references** *time_slot*,
  primary key (*course_id, sec_id, semester, year*),
  foreign key (*building*, *room_number*) **references** *classroom*);

And more…

This is a weak entity set!



- **create table** *takes* (
  ID **varchar**(5),
  course_id **varchar**(8),
  sec_id **varchar**(8),
  semester **varchar**(6),
  year **numeric**(4,0),
  grade **varchar**(2),
  foreign key (*ID*) **references** *student,*
  foreign key (*course_id, sec_id, semester, year*) **references** *section,*
  primary key (*ID*, *course_id*, *sec_id*, *semester*, *year*)
  );

Summary (Takeaways)

- What is SQL and what parts is it divided into
  - DDL, DML, DCL (controls transactions…)
- Be able to formulate an SQL query
  - SELECT, INSERT, UPDATE, DELETE
  - CREATE TABLE, CREATE VIEW
- Understand the joins of relations – to be able to explain differences
- Nested queries (what it is and where it can be used)
- Aggregate functions
- NULL values and their impact on query evaluation
- Types of integrity constraints