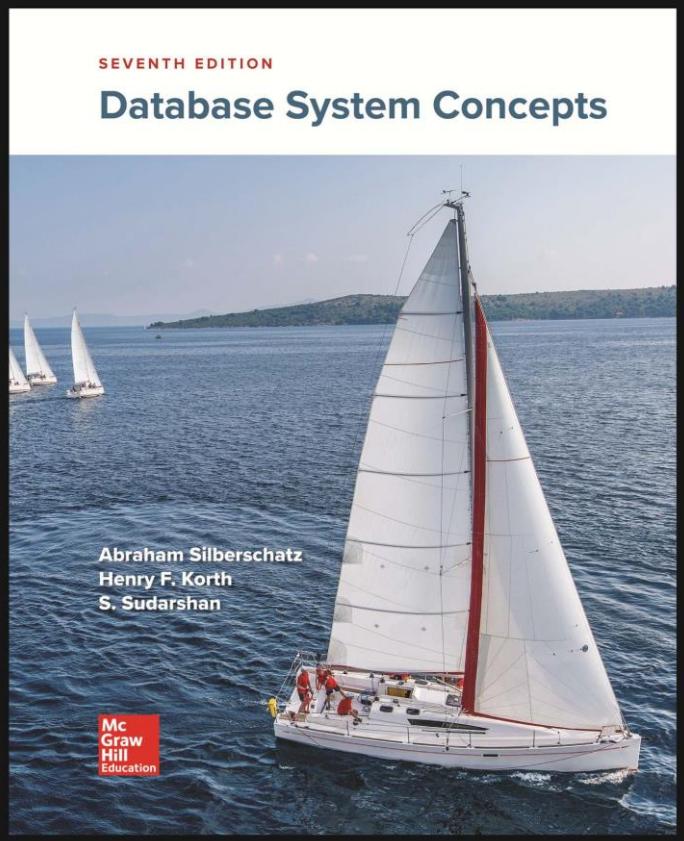


IF2240 – Basis Data SQL (Part 1)



Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 3 : Introduction to SQL

History

IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory

Renamed Structured Query Language (SQL)

ANSI and ISO standard SQL:

- SQL-86
- SQL-89
- SQL-92
- SQL:1999 (language name became Y2K compliant!)
- SQL:2003

Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.

- Not all examples here may work on your particular system.



KNOWLEDGE & SOFTWARE ENGINEERING

SQL Parts

DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.

integrity - the DDL includes commands for specifying integrity constraints.

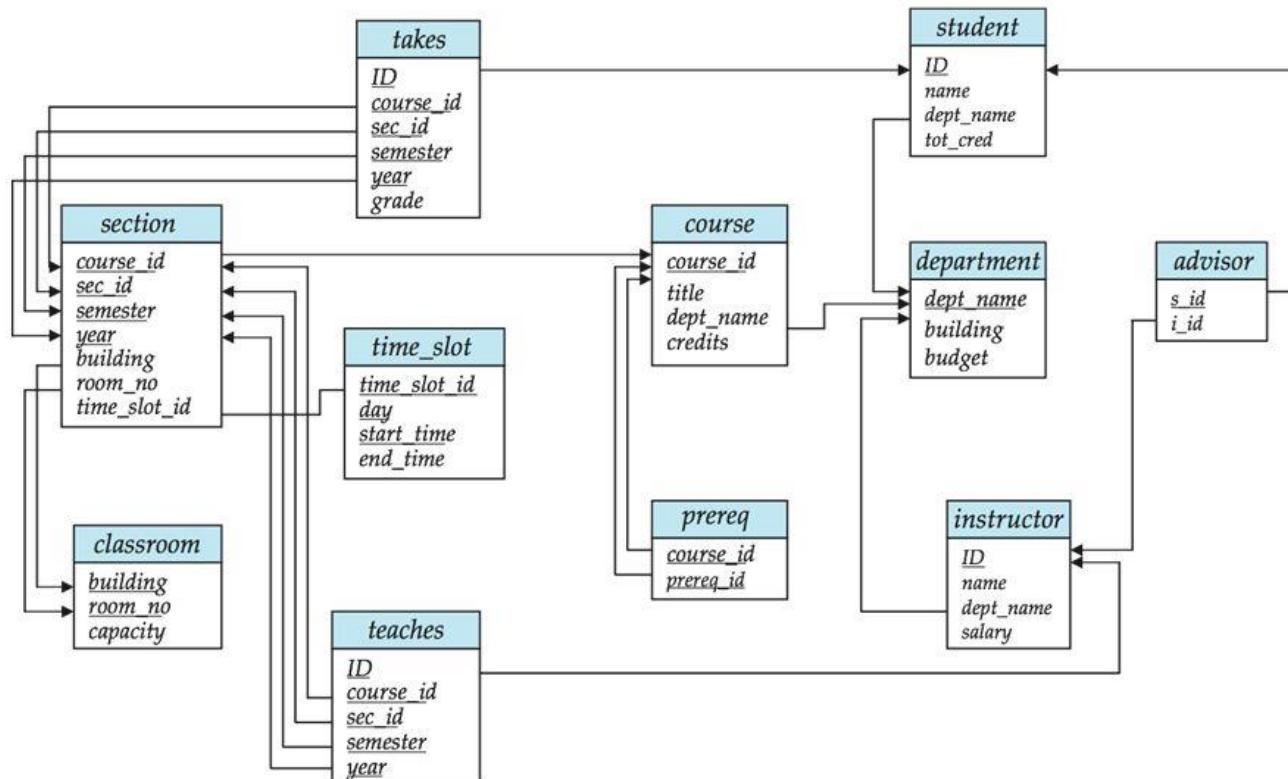
View definition -- The DDL includes commands for defining views.

Transaction control -includes commands for specifying the beginning and ending of transactions.

Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.

Authorization - includes commands for specifying access rights to relations and views.

Schema Diagram



Basic Query Structure

A typical SQL query has the form:

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

- A_i represents an attribute
- R_i represents a relation
- P is a predicate.

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: 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., $Name \equiv NAME \equiv name$
- 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 department names of all instructors, and remove duplicates

```
select distinct dept_name  
from instructor
```

The keyword **all** specifies that duplicates should not be removed.

```
select all dept_name  
from instructor
```

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

The select Clause (Cont.)

An asterisk in the select clause denotes “all attributes”

```
select *  
from instructor
```

An attribute can be a literal with no **from** clause : **select** '437'

- Results is a table with one column and a single row with value “437”
- Can give the column a name using:

```
select '437' as FOO
```

An attribute can be a literal with **from** clause

```
select 'A'  
from instructor
```

- Result is a table with one column and N rows (number of tuples in the *instructors* table), each row with value “A”



KNOWLEDGE & SOFTWARE ENGINEERING

The select Clause (Cont.)

The **select** clause can contain arithmetic expressions involving the operation, +, -, *, and /, and operating on constants or attributes of tuples.

- The query:

```
select ID, 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.

- Can rename “*salary/12*” using the **as** clause:

```
select ID, name, salary/12 as monthly_salary
```

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. dept

```
select name  
from instructor  
where dept_name = 'Comp. Sci.'
```

SQL allows the use of the logical connectives **and**, **or**, and **not**. The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>. Comparisons can be applied to results of arithmetic expressions

To find all instructors in Comp. Sci. dept with salary > 70000

```
select name  
from instructor  
where dept_name = 'Comp. Sci.' and salary > 70000
```

name
Katz
Brandt

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 X teaches*

```
select *
  from instructor, teaches
```

- generates every possible instructor – teaches pair, with all attributes from both relations.
- For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)

Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

Examples

Find the names of all instructors who have taught some course and the course_id

- `select name, course_id
from instructor , teaches
where instructor.ID = teaches.ID`

Find the names of all instructors in the Art department who have taught some course and the course_id

- `select name, course_id
from instructor , teaches
where instructor.ID = teaches.ID
and instructor.dept_name = 'Art'`

<i>name</i>	<i>course_id</i>
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

The Rename Operation

The SQL allows renaming relations and attributes using the **as** clause:

old-name as new-name

Find the names of all instructors who have a higher salary 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 omitted

instructor as T ≡ *instructor T*

Self “Join” Example

Relation *emp-super*

<i>person</i>	<i>supervisor</i>
Bob	Alice
Mary	Susan
Alice	David
David	Mary

Find the supervisor of “Bob”

Find the supervisor of the supervisor of “Bob”

Can you find ALL the supervisors (direct and indirect) of “Bob”?

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 "100%"

```
like '100 \%' escape '\'
```

in that above we use backslash (\) as the escape character.

String Operations (Cont.)

Patterns are case sensitive.

Pattern matching examples:

- 'Intro%' matches any string beginning with "Intro".
- '%Comp%' matches any string containing "Comp" as a substring.
- '_ _ _' matches any string of exactly three characters.
- '_ _ _ %' matches any string of at least three characters.

SQL supports a variety of string operations such as

- concatenation (using "||")
- converting from upper to lower case (and vice versa)
- finding string length, extracting substrings, etc.

Ordering the Display of Tuples

List in alphabetic order the names of all instructors

```
select distinct 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**

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');`

Set Operations

- Find courses that ran in Fall 2017 **or** in Spring 2018

```
(select course_id from section where sem = 'Fall' and year = 2017)
union
(select course_id from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 **and** in Spring 2018

```
(select course_id from section where sem = 'Fall' and year = 2017)
intersect
(select course_id from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 **but not in** Spring 2018

```
(select course_id from section where sem = 'Fall' and year = 2017)
except
(select course_id from section where sem = 'Spring' and year = 2018)
```

Set Operations (Cont.)

Set operations **union**, **intersect**, and **except**

- Each of the above operations automatically eliminates duplicates

To retain all duplicates use the

- **union all**,
- **intersect all**
- **except all**.

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: $5 + \text{null}$ 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
```

The predicate **is not null** succeeds if the value on which it is applied is not null.

Null Values (Cont.)

SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).

- Example: $5 < \text{null}$ or $\text{null} <> \text{null}$ or $\text{null} = \text{null}$

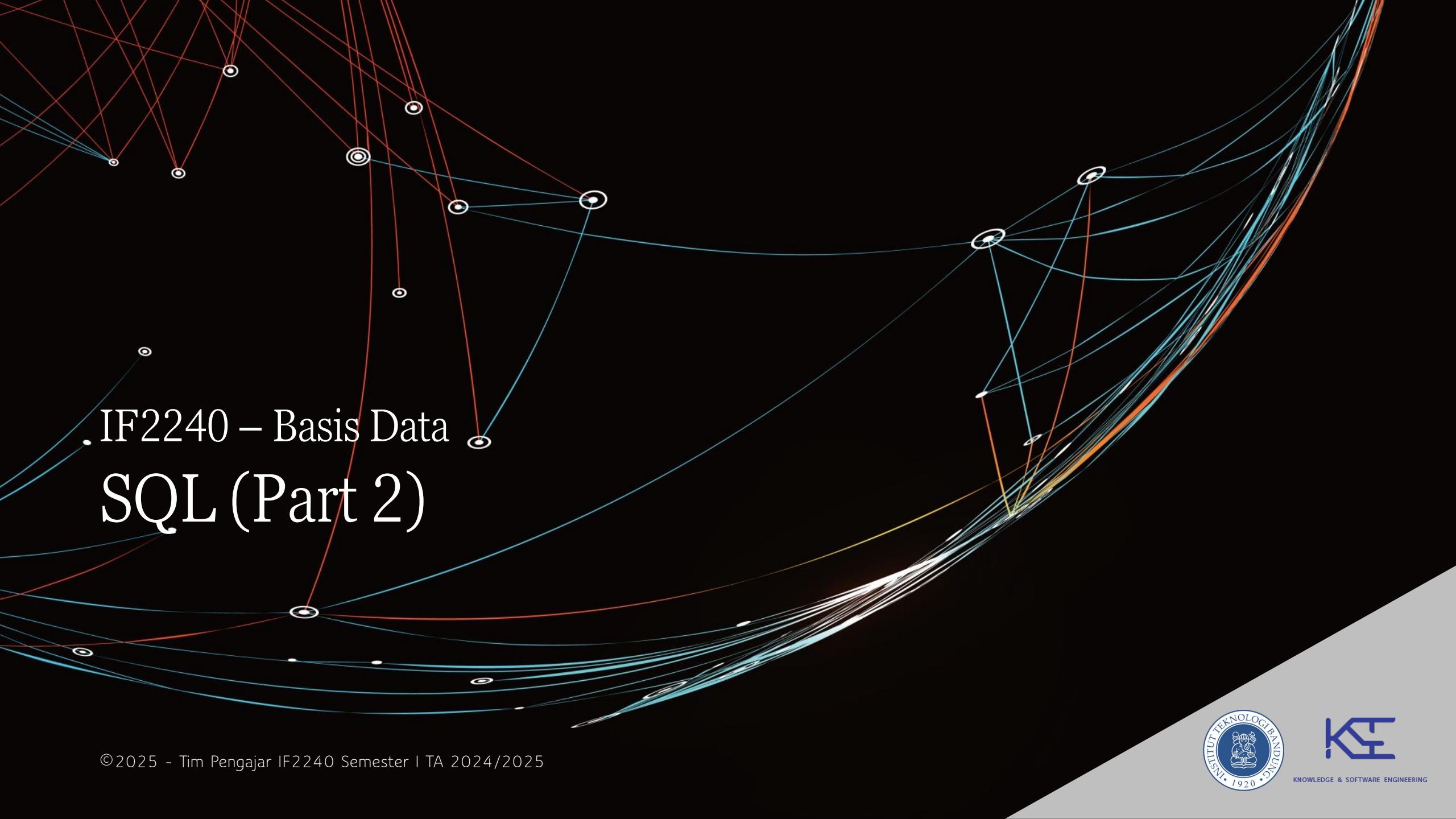
The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.

- **and** : $(\text{true and unknown}) = \text{unknown}$,
 $(\text{false and unknown}) = \text{false}$,
 $(\text{unknown and unknown}) = \text{unknown}$
- **or**: $(\text{unknown or true}) = \text{true}$,
 $(\text{unknown or false}) = \text{unknown}$
 $(\text{unknown or unknown}) = \text{unknown}$

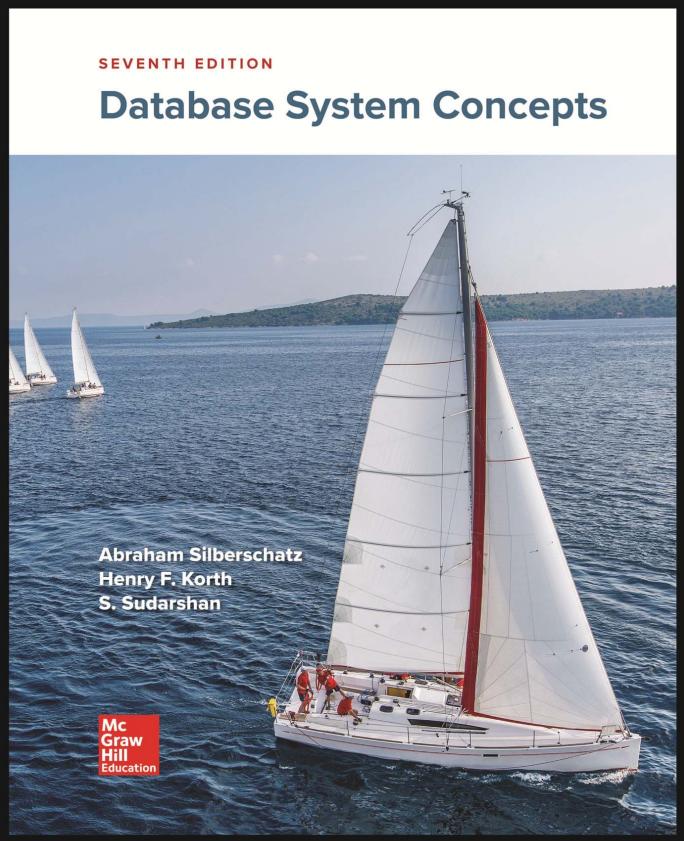
Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Latihan-1

1. Tampilkan data ID dan nama student yang berasal dari departemen “Comp. Sci.”.
2. Tampilkan data instruktur dari departemen “Comp. Sci” yang memiliki gaji antara 7.000.000 hingga 12.000.000 terurut mengecil berdasarkan gaji.
3. Tampilkan data student dengan nama berawalan “Budi” dan terdiri dari sekurang-kurangnya 2 kata.
4. Tampilkan daftar mata kuliah (kode, nama, sks) yang diambil oleh student dengan ID “13521400” pada semester 2 tahun 2022.
5. Tampilkan daftar seluruh kelas yang ditawarkan oleh departemen “Comp. Sci.” pada semester 2 tahun 2022. Informasi yang ingin ditampilkan meliputi kode kuliah, nama, jumlah sks, nomor kelas (sec_id), serta nama dari instruktur kelas tersebut. Daftar terurut berdasarkan kode kuliah dan nomor kelas.



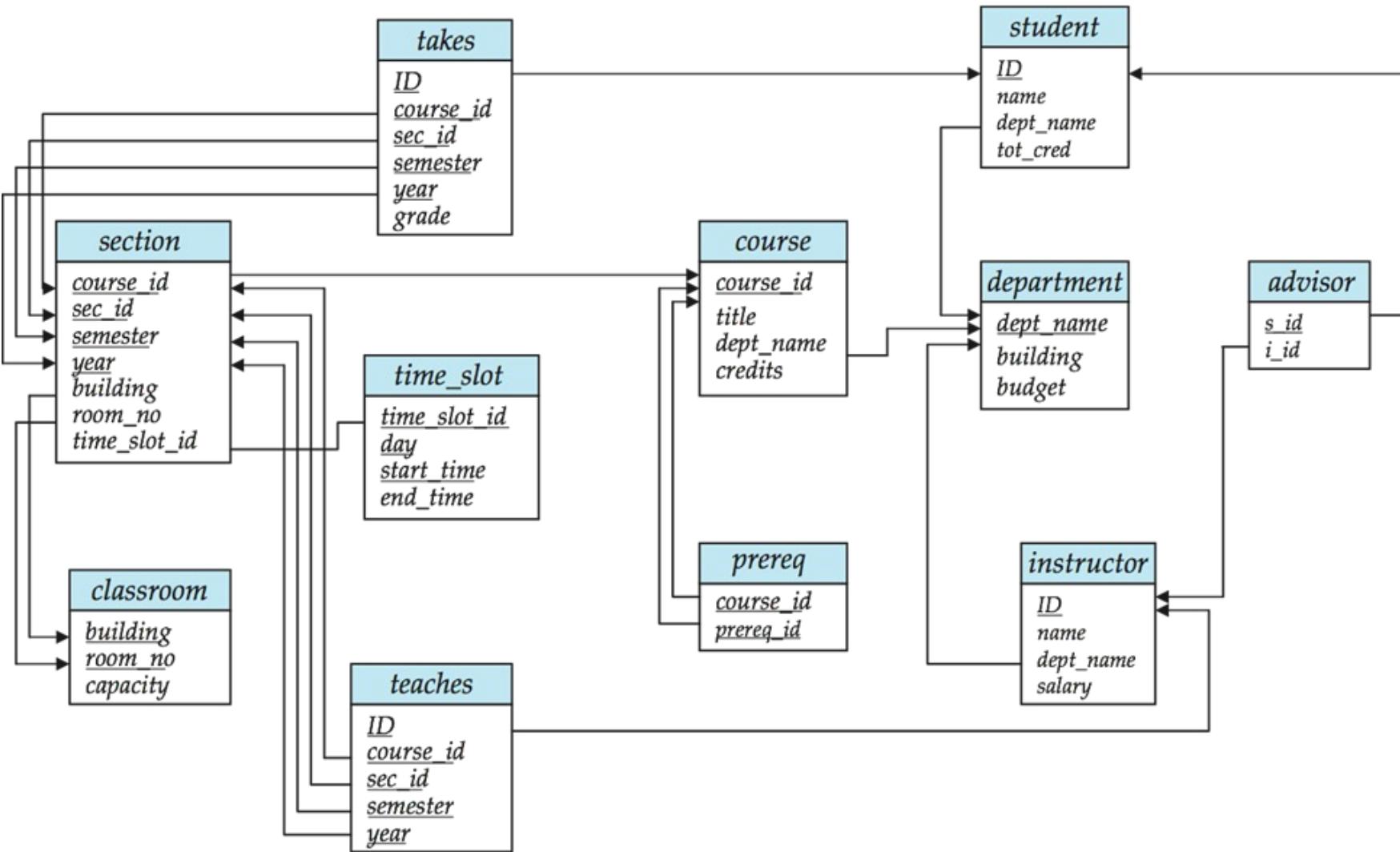
IF2240 – Basis Data SQL (Part 2)



Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 3 : Introduction to SQL



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 Examples

Find the average salary of instructors in the Computer Science department

- ```
select avg (salary)
from instructor
where dept_name= 'Comp. Sci.';
```

Find the total number of instructors who teach a course in the Spring 2018 semester

- ```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;
```

Find the number of tuples in the *course* relation

- ```
select count (*)
from course;
```



KNOWLEDGE & SOFTWARE ENGINEERING

# Aggregate Functions – Group By

Group By: Group tuple that have the same values into summary rows

- Often used with aggregate functions

Find the average salary of instructors in each department

- ```
select dept_name,
       avg (salary) as 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_salary
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 must appear in **group by** list

- /* erroneous query */

```
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```

Aggregate Functions – Having Clause

HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions

Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, avg (salary) as 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



KNOWLEDGE & SOFTWARE ENGINEERING

How About Some More Queries?

6. Find the highest salary earned by instructors from Comp. Sci. Department.
7. Find the enrollment of each section that was offered in Autumn 2009.
8. Find all departments with more than 1000 student enrollments, together with the number of students for each of them. The list should be ordered by dept_name
9. Find all instructors earning the highest salary (there may be more than one with the same salary)

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

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 instructors along with the course ID of the courses that they taught

- `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 A1, A2, .. An
  from r1 natural join r2 natural join .. natural join rn
    where P ;
```

Student Relation

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
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

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
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	<i>null</i>

student natural
join *takes*

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
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	<i>null</i>

Dangerous in Natural Join

Beware of unrelated attributes with same name which get equated incorrectly

Example -- List the names of students instructors along with the titles of courses that they have taken

- Correct version

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

- Incorrect version

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

- This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department.
- The correct version (above), correctly outputs such pairs.

Outer Join

An extension of the join operation that avoids loss of information.

Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.

Uses *null* values.

Three forms of outer join:

- left outer join
- right outer join
- full outer join

Outer Join Examples

Relation *course*

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

Relation *prereq*

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Observe that

course information is missing CS-347

prereq information is missing CS-315

Left Outer Join

course natural left outer join prereq

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

In relational algebra: $course \bowtie prereq$

Right Outer Join

course natural right outer join prereq

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

In relational algebra: $course \bowtie prereq$

Full Outer Join

course natural full outer join prereq

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

In relational algebra: $course \bowtie prereq$

Joined Types and Conditions

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.

- natural
- on <predicate>
- using (A_1, A_2, \dots, A_n)

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.

- inner join
- left outer join
- right outer join
- full outer join

Joined Relations – Examples

course natural right outer join prereq

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

course full outer join prereq using (course_id)

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

Joined Relations – Examples

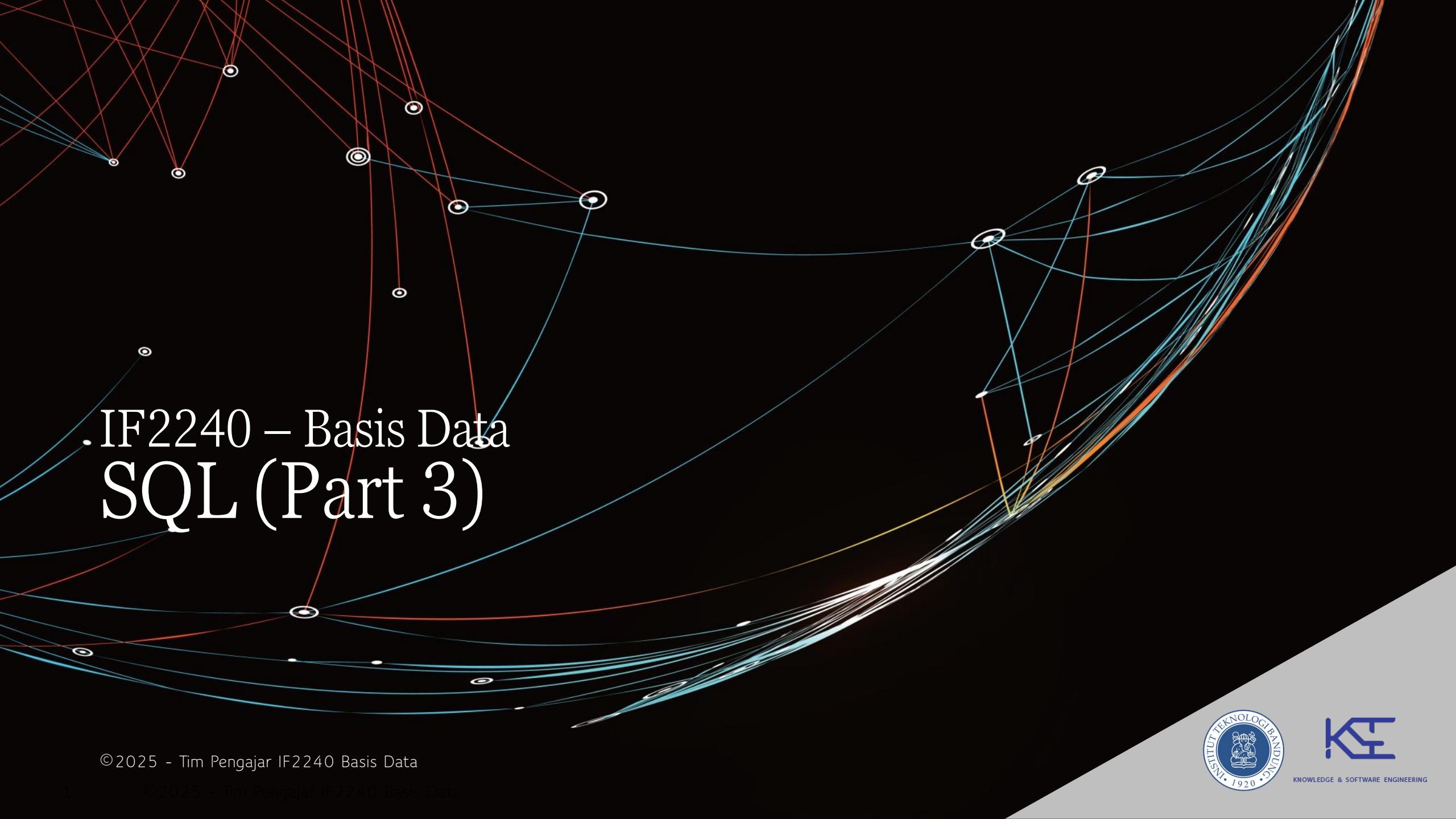
```
course inner join prereq on  
course.course_id = prereq.course_id
```

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

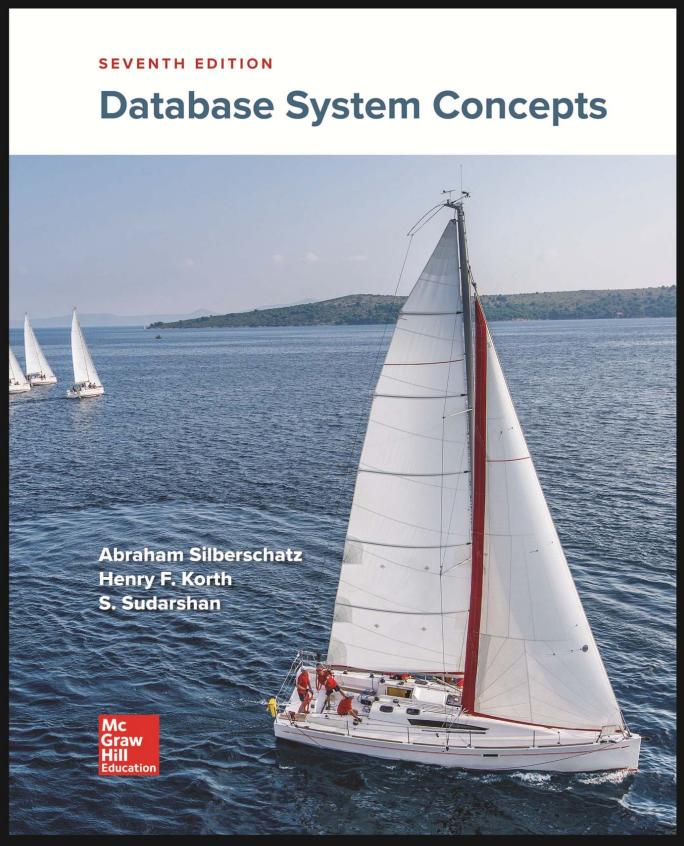
What is the difference between the above, and a natural join?

```
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



IF2240 – Basis Data SQL (Part 3)



Sumber

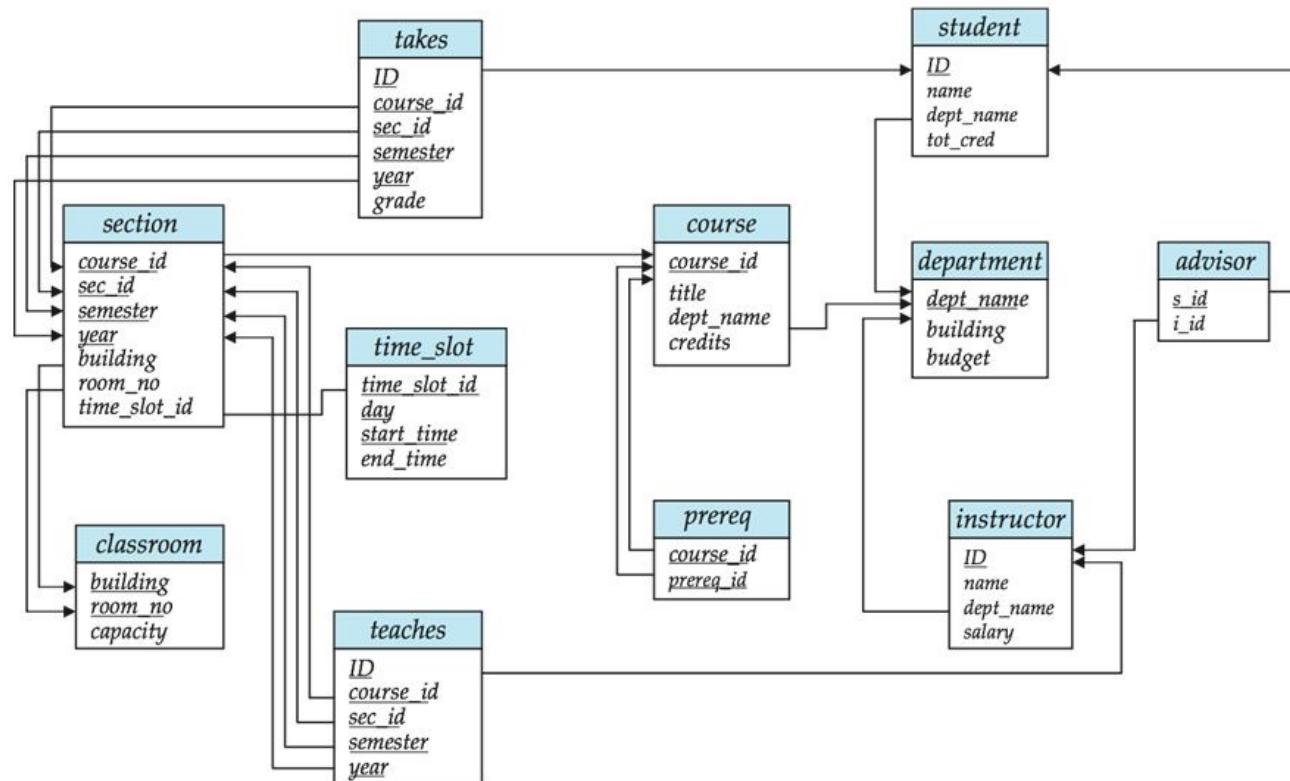
Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 3 : Introduction to SQL
- Chapter 4 : Intermediate SQL



KNOWLEDGE & SOFTWARE ENGINEERING

Schema Diagram



Nested Subqueries

SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.

The nesting can be done in the following SQL query

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P
```

as follows:

- **Where clause:** P can be replaced with an expression of the form:
$$B \langle\text{operation}\rangle (\text{subquery})$$

 B is an attribute and $\langle\text{operation}\rangle$ to be defined later.
- **From clause:** r_i can be replaced by any valid subquery
- **Select clause:**
 A_i can be replaced by a subquery that generates a single value.

Nested Query in Where Clause

- Set Membership (**IN**): to check whether an attribute value is a member of the result set produced by a subquery.
- Set Comparison (**<relational operator> SOME/ALL**): to evaluate whether an attribute value satisfies the **<relational operator>** condition against some or all tuples in the subquery result.
- Test of Empty Relation (**EXISTS**): to determine whether a subquery returns any tuples (i.e., whether the subquery produces a result).
- Test for Absence of Duplicates (**UNIQUE**): to check whether the tuples in the subquery result are unique (i.e., no duplicate values exist).

Set Membership

Find courses offered in Fall 2017 and in Spring 2018

```
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);
```

Find courses offered in Fall 2017 but not in Spring 2018

```
select distinct course_id  
from section  
where semester = 'Fall' and year= 2017 and  
course_id not in (select course_id  
from section  
where semester = 'Spring' and year= 2018);
```



KNOWLEDGE & SOFTWARE ENGINEERING

Set Membership (Cont.)

Name all instructors whose name is neither "Mozart" nor Einstein"

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

Find the total number of (distinct) students who have taken course sections taught by the instructor 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



KNOWLEDGE & SOFTWARE ENGINEERING

Set Comparison – “some” Clause

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

```
select distinct T.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');
```



KNOWLEDGE & SOFTWARE ENGINEERING

Definition of “some” Clause

$F <\text{comp}> \text{some } r \Leftrightarrow \exists t \in r \text{ such that } (F <\text{comp}> t)$

Where comp can be: $<$, \leq , $>$, $=$, \neq

$(5 < \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$ (read: 5 < some tuple in the relation)

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

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

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

$(= \text{some}) \equiv \text{in}$

However, $(\neq \text{some}) \equiv \text{not in}$



Set Comparison – “all” Clause

Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

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

Definition of “all” Clause

$$F \langle \text{comp} \rangle \text{ all } r \Leftrightarrow \forall t \in r (F \langle \text{comp} \rangle t)$$

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

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

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

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

$(\neq \text{all}) \equiv \text{not in}$
However, $(= \text{all}) \not\equiv \text{in}$

Test for Empty Relations

The **exists** construct returns the value **true** if the argument subquery is nonempty.

$$\text{exists } r \Leftrightarrow r \neq \emptyset$$

$$\text{not exists } r \Leftrightarrow r = \emptyset$$

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 “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 $X - Y = \emptyset \Leftrightarrow X \subseteq Y$

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

```
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);
```

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

Another way to write above query

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

With Clause

The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.

Find all departments with the maximum budget

```
with max_budget (value) as
    (select max(budget)
     from department)
select department.name
from department, max_budget
where department.budget = max_budget.value;
```

Complex Queries using With Clause

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

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

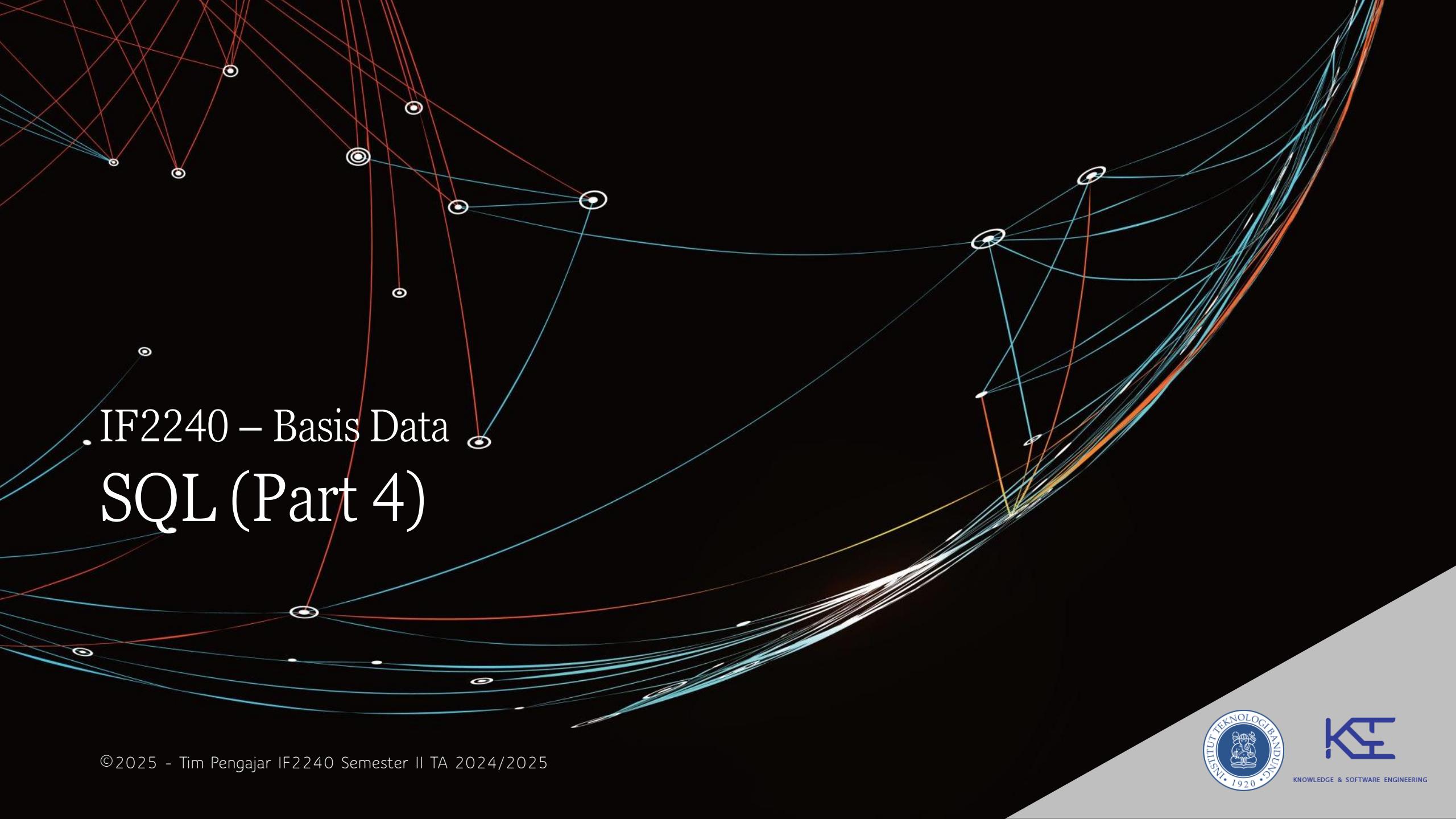
Scalar Subquery

Scalar subquery is one which is used where a single value is expected

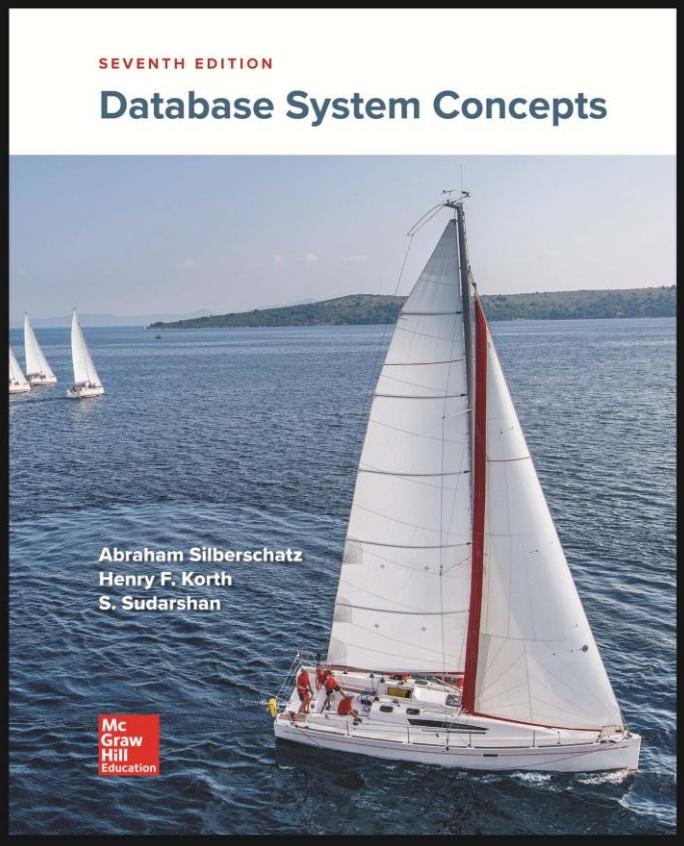
List all departments along with the number of instructors in each department

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

Runtime error if subquery returns more than one result tuple



IF2240 – Basis Data SQL (Part 4)

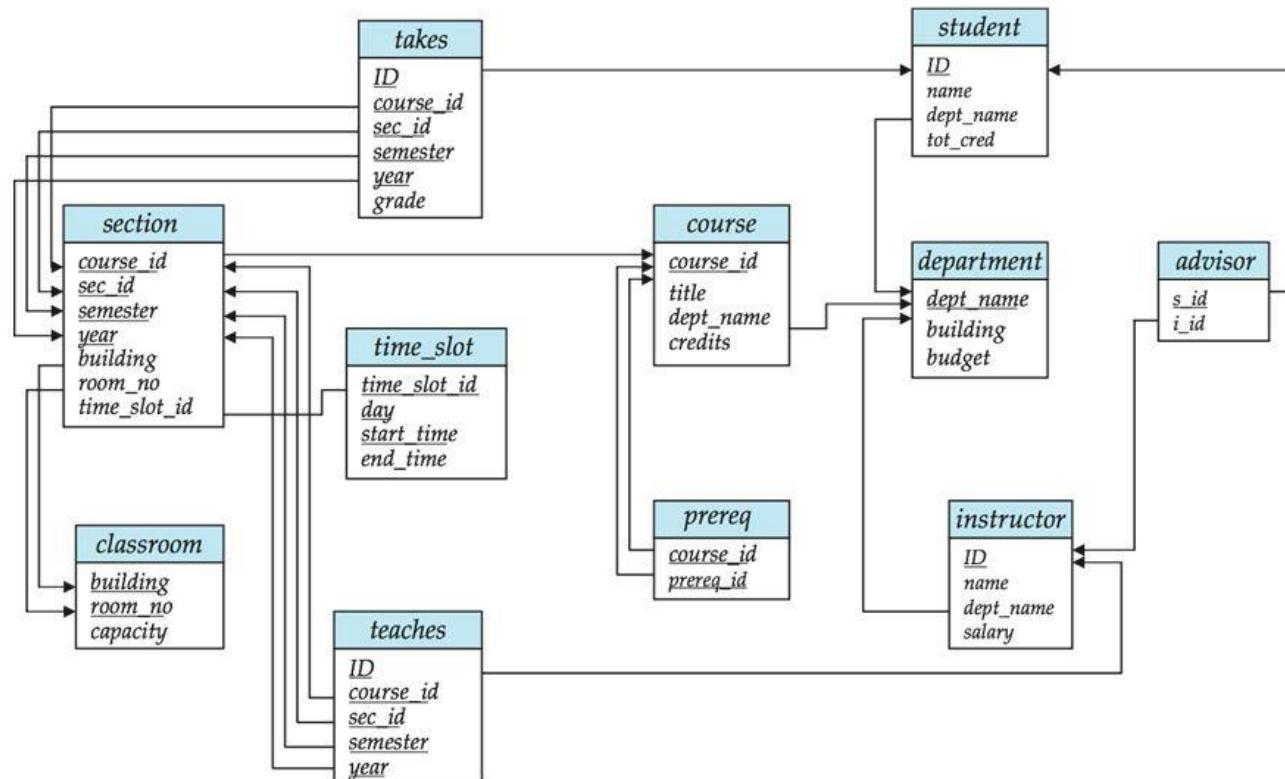


Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 3 : Introduction to SQL
- Chapter 4 : Intermediate SQL

Schema Diagram



View

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**.



KNOWLEDGE & SOFTWARE ENGINEERING

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 and Use

A view of instructors without their salary

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

Find all instructors in the Biology department

```
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;
```



KNOWLEDGE & SOFTWARE ENGINEERING

Views Defined Using Other Views (1/2)

One view may be used in the expression defining another view

A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1

A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2

A view relation v is said to be *recursive* if it depends on itself.

Views Defined Using Other Views (2/2)

```
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';
```



KNOWLEDGE & SOFTWARE ENGINEERING

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**:

If relations used in the query are updated, the materialized view result becomes out of date

- Need to **Maintain** the view, by updating the view whenever the underlying relations are updated.

Modification of the Database

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 of a given relation

Deletion

**delete from <table_name>
where <condition>;**

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.

**conditions involving tuples from other relations
are expressed using subquery.**

```
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

```
insert into <table_name> [(<attributes_list>)]  
values (<attributes_values>);
```

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);
```



KNOWLEDGE & SOFTWARE ENGINEERING

Insertion (Cont.)

```
insert into <table_name> [(<attributes_list>)]  
<query>;
```

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

```
update <table_name>
set <attribute_name> = <expression>
where <condition>;
```

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);
```



KNOWLEDGE & SOFTWARE ENGINEERING

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
```

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
```

Update of a View

Add a new tuple to faculty view which we defined earlier

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

This insertion must be represented by the insertion into the instructor relation

- Must have a value for salary.

Two approaches

- Reject the insert
- 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

- Which department, if multiple departments in Taylor?
- What if no department is in Taylor?

And Some Not at All

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

What happens if we insert

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

into history_instructors?

View Updates in SQL

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.

Data Definition Language

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

1. The schema for each relation.
2. The type of values associated with each attribute.
3. The Integrity constraints
4. The set of indices to be maintained for each relation.
5. Security and authorization information for each relation.
6. The physical storage structure of each relation on disk.

Domain Types in SQL (1/2)

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 stored 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.

Domain Types in SQL (2/2)

date Dates, containing a (4 digit) year, month and date

- Example: `date '2005-7-27'`

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 '2005-7-27 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

Create Table Construct

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

```
create table r
  (A1 D1, A2 D2, ..., An Dn,
   (integrity-constraint1),
   ...
   (integrity-constraintk))
```

- *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))
```

Integrity Constraints in Create Table

Types of integrity constraints

- primary key (A_1, \dots, A_n)
- foreign key (A_m, \dots, A_n) references r
- not null

SQL prevents any update to the database that violates an integrity constraint.

Example:

```
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);
```

And a Few More Relation Definitions

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

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

And more still

```
create table course (
    course_id      varchar(8),
    title          varchar(50),
    dept_name      varchar(20),
    credits         numeric(2,0),
    primary key (course_id),
    foreign key (dept_name) references department);
```

Updates to tables

Drop Table

- `drop table r`

Alter

- `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 existing tuples in the relation are assigned *null* as the value for the new attribute.
- `alter table r drop A`
 - where *A* is the name of an attribute of relation *r*
 - Dropping of attributes not supported by many databases.