❖ Week Recap

- Basic notions of Relational Database Models
  - Attributes and their types
  - Mathematical structure of relational model
  - Schema and Instance
  - Keys, primary as well as foreign
- Relational algebra with operators
- Relational query language
  - DDL (Data Definition)
  - DML (Basic Query Structure)
- Detailed understanding of basic query structure
- Set operations, null values, and aggregation

❖ Select distinct

From the classroom relation in the figure, find the names of buildings in which every individual classroom has capacity less than 100 (removing the duplicates).

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure: classroom relation

➔Query:

select distinct building **from** classroom
where capacity < 100;

➔Output:

building
Painter
Taylor
Watson

❖ Select all

From the classroom relation in the figure, find the names of buildings in which every individual classroom has capacity less than 100 (without removing the duplicates).

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure: classroom relation

➔Query:

select all building **from** classroom
where capacity < 100;

➔Output:

building
Painter
Taylor
Watson
Watson

Cartesian Product

Find the list of all students of departments which have a budget < $0.1million

select name, budget
from student, department
where student.dept name = department.dept name
and budget < 100000;

name	*budget*
Brandt	50000.00
Peltier	70000.00
Levy	70000.00
Sanchez	80000.00
Snow	70000.00
Aoi	85000.00
Bourikas	85000.00
Tanaka	90000.00

- The above query first generates every possible student department pair, which is the Cartesian product of student and department. Then, it filters all the rows with student.dept name = department.dept name **and** budget < 100000.

- The common attribute dept name in the resulting table are renamed using the relation name - student.dept name and department.dept name)

Rename AS Operation

- The same query in the previous slide can be framed by renaming the tables as shown below.

  select S.name **as** studentname, budget **as** deptbudget
  from student **as** S, department **as** D
  where S.dept name = D.dept name **and** budget < 100000;

name	*budget*
Brandt	50000.00
Peltier	70000.00
Levy	70000.00
Sanchez	80000.00
Snow	70000.00
Aoi	85000.00
Bourikas	85000.00
Tanaka	90000.00

- The above query renames the relation student **as** S and the relation department **as** D
- It also displays the attribute name as StudentName and budget as DeptBudget.
- Note that the budget attribute does not have any prefix because it occurs only in the department relation.

Where: AND and OR

- From the instructor and department relations in the figure, find out the names of all instructors whose department is Finance or whose department is in any of the following buildings: Watson, Taylor.

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
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

department

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

➔ Query:

select name
from instructor I, department D
where D.dept name = I.dept name
and (I.dept name = 'Finance'
or building **in** ('Watson','Taylor'));

➔ Output:

name
Srinivasan
Wu
Einstein
Gold
Katz
Singh
Crick
Brandt
Kim

String Operations

- From the course relation in the figure, find the titles of all courses whose course id has three alphabets indicating the department.

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure: course relation

➔ Query:

```
select title
from course
 where course_id like '___-%';
```

➔ Output:

title
Intro. to Biology
Genetics
Computational Biology
Investment Banking
World History
Physical Principles

- The course id of each department has either 2 or 3 alphabets in the beginning, followed by a hyphen and then followed by a 3-digit number. The above query returns the names of those departments that have 3 alphabets in the beginning.

Order By

- From the student relation in the figure, obtain the list of all students in alphabetic order of departments and within each department, in decreasing order of total credits.

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

Figure: student relation

- The list is first sorted in alphabetic order of dept name.
- Within each dept, it is sorted in decreasing order of total credits.

➔ Query:

```
select name, dept name, tot cred
from student
order by dept name ASC, tot cred DESC;
```

➔ Output:

name	dept_name	tot_cred
Tanaka	Biology	120
Zhang	Comp. Sci.	102
Brown	Comp. Sci.	58
Williams	Comp. Sci.	54
Shankar	Comp. Sci.	32
Bourikas	Elec. Eng.	98
Aoi	Elec. Eng.	60
Chavez	Finance	110
Brandt	History	80
Sanchez	Music	38
Peltier	Physics	56
Levy	Physics	46
Snow	Physics	0

In Operator

- From the teaches relation in the figure, find the IDs of all courses taught in the Fall or Spring of 2018

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

Figure: teaches relation

➔Query:

```
select course id
from teaches
where semester in ('Fall','Spring') and year=2018;
```

➔Output:

course_id
CS-315
FIN-201
MU-199
HIS-351
CS-101
CS-319
CS-319

Note: We can use **distinct** to remove duplicates.

Set Operations: union

- For the same question in the previous slide, we can find the solution using union operator as follows.

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

Figure: teaches relation

➔Query:

```
select course id
from teaches
where semester='Fall'
    and year=2018
union
select course id
from teaches
where semester='Spring'
    and year=2018
```

➔ Output:

course_id
CS-101
CS-315
CS-319
FIN-201
HIS-351
MU-199

- Note that union removes all duplicates. If we use union all instead of union, we get the same set of tuples as in previous slide.

Set Operations (2): intersect

- From the instructor relation in the figure, find the names of all instructors who taught in either the Computer Science department or the Finance department and whose salary is < 80000.

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

➔Query:

```
select name
from instructor
where dept name in ('Comp. Sci.','Finance')
intersect
select name
from instructor
where salary < 80000;
```

⇨ Figure: instructor relation

➔Output:

name
Srinivasan
Katz

- Note that the same can be achieved using the query: select name from instructor where dept name in('Comp. Sci.', 'Finance') and salary < 80000;

Set Operations (3): except

- From the *instructor* relation in the figure, find the names of all instructors who taught in either the Computer Science department or the Finance department and whose salary is either ≥ 90000 or ≤ 70000.

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

Figure: *instructor* relation

  - Query:

```
select name
from instructor
where dept_name in ('Comp. Sci.','Finance')
except
select name
from instructor
where salary < 90000 and salary > 70000;
```

  - Output:

name
Srinivasan
Brandt
Wu

  - Note that the same can be achieved using the query given below:

```
select name from instructor
where dept_name in('Comp. Sci.', 'Finance')
and (salary >= 90000 or salary <= 70000);
```

Aggregate functions: avg

- From the *classroom* relation given in the figure, find the names and the average capacity of each building whose average capacity is greater than 25.

building	room_number	capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure: *classroom* relation

  - Query:

```
select building, avg (capacity)
from classroom
group by building
having avg (capacity) > 25;
```

  - Output:

building	avg
Taylor	70.00
Packard	500.00
Watson	40.00

Aggregate functions (2): min

- From the instructor relation given in the figure, find the least salary drawn by any instructor among all the instructors.

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

➔Figure: instructor relation

➔Query:

```
select min(salary) as least salary
from instructor;
```

➔Output:

least_salary
40000.00

Aggregate functions (3): max

- From the student relation given in the figure, find the maximum credits obtained by any student among all the students.

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

→Figure: student relation

➔Query:

select max(tot cred) as max credits
from student;

➔ Output:

max_credits
120

Aggregate functions (4): count

- From the *section* relation given in the figure, find the number of courses run in each building.

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

Figure: section relation

- Query:

select *building*,
count(*course_id*) **as** *course_count*
from *section*
group by *building*;

- Output:

building	*course_count*
Taylor	5
Packard	4
Painter	3
Watson	3

Aggregate functions (5): sum

- From the *course* relation given in the figure, find the total credits offered by each department.

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure: *course* relation

- Query:

select *dept_name*,
sum(*credits*) **as** *sum_credits*
from *course*
group by *dept_name*;

- Output:

dept_name	*sum_credits*
Finance	3
History	3
Physics	4
Music	3
Comp. Sci.	17
Biology	11
Elec. Eng.	3

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries
- A subquery is a select-from-where expression that is nested within another query
- The nesting can be done in the following SQL query

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

  as follows:
  - A_i can be replaced by a subquery that generates a single value
  - r_i can be replaced by any valid subquery
  - P can be replaced with an expression of the form:

    B <operation> (subquery)

    where B is an attribute and <operation> to be defined later

Subqueries in the Where Clause

- Typical use of subqueries is to perform tests:
  - For set membership
  - For set comparisons
  - For set cardinality

Set Membership

- Find courses offered in Fall 2009 and in Spring 2010. (**intersect** example)

```
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. (**except** example)

```
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);
```

Set Membership (2)

- 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 simpler manner. The formulation above is simply to illustrate SQL features.

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');
```

Definition of "some" Clause

- F <comp> **some** r ⇔ ∃t ∈ r such that (F <comp> t)
  where <comp> can be: <, ≤, >, ≥, =, ≠
- **some** represents existential quantification

(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**

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 <comp> **all** r ⇔ ∀t ∈ r such that (F <comp> t)
  Where <comp> can be: <, ≤, >, ≥, =, ≠
- **all** represents universal quantification

(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**

Test for Empty Relations: "exists"

- The exists construct returns the value true if the argument subquery is nonempty
  - exists r ⇔ r ≠ ∅
  - not exists r ⇔ r = ∅

Use of "exists" Clause

- 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);
```

- **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: X − Y = ∅ ⇔ X ⊆ Y
- Note: Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples: "unique"

- 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 2009

```
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 = 2009);
```

Subqueries in the From Clause

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;
```

Subqueries in the Select Clause

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

Modifications 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 in a given relation

Deletion

- Delete all instructors
  delete from *instructor*
- Delete all instructors from the Finance department
  delete from *instructor*
  where *dept_name*= 'Finance';
- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building
  delete from *instructor*
  where *dept_name* **in** (**select** *dept_name*
  from *department*
  where *building* = 'Watson');

Deletion (2)

- 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 deposit, the average salary changes
- Solution used in SQL:
  a) First, compute **avg** (salary) and find all tuples to delete
  b) 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 (2)

- Add all instructors to the *student* relation with *tot_creds* 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

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

would cause problem

Updates

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

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

- The order 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_creds = (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 Expressions

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

Types of Join between Relations

- Cross join
- Inner join
  - Equi-join
    - Natural join
- Outer join
  - Left outer join
  - Right outer join
  - Full outer join
- Self-join

Cross Join

- CROSS JOIN returns the Cartesian product of rows from tables in the join
  - Explicit

```
select *
from employee cross join department;
```

  - Implicit

```
select *
from employee, department;
```

Join operations – Example

- Relation *course*

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

- Relation *prereq*

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

- Observe that
  prereq information is missing for CS-315 and
  course information is missing for CS-347

Inner Join

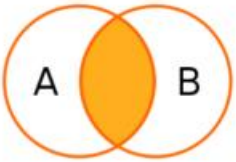
- *course* **inner join** *prereq*

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

- If specified as **natural**, the 2nd course_id field is skipped

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



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

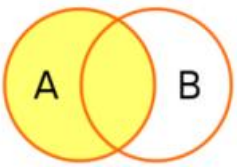
Left Outer Join

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

course_id	title	dept_name	credits	prere_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
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



Right Outer Join

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

course_id	title	dept_name	credits	prere_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
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 types
inner join
left outer join
right outer join
full outer join

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

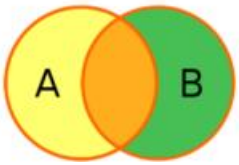
Full Outer Join

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

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

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

course_id	*title*	*dept_name*	*credits*	*prere_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 (equi_join), and a natural join?
- *course* **left outer join** *prereq* **on** *course.course_id* = *prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prere_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*

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by
  select *ID, name, dept_name*
  from *instructor*
- A **view** provides a mechanism to hide certain data from the view of certain users
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form
  create view *v* **as** < query expression >
  where < query expression > is any legal SQL expression
- The view name is represented by *v*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates
- 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');
```

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v1 is said to depend directly on a view relation v2 if v2 is used in the expression defining v1
- A view relation v1 is said to depend on view relation v2 if either v1 depends directly on v2 or there is a path of dependencies from v1 to v2
- A view relation v is said to be recursive if it depends on itself

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

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 of 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');
  - 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

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?

Materialized Views

- **Materializing a view**: create a physical table containing all the tuples in the result of the query defining the 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

Transactions

Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (for example, using API)
  - In SQL:1999, can use: **begin atomic ... end**
    - Not supported on most databases

Integrity Constraints

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 Rs. 10,000.00
  - A salary of a bank employee must be at least Rs. 250.00 an hour
  - A customer must have a (non-null) phone number

Integrity 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**
- **unique** $(A_1, A_2, \ldots, A_m)$
  - The unique specification states that the attributes $A_1, A_2, \ldots, A_m$ form a candidate key
  - Candidate keys are permitted to be null (in contrast to primary keys).

The check clause

- check(P), where P is a predicate
- 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'))
);
```

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. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S

Cascading Actions in Referential Integrity

- With cascading, you can define the actions that the Database Engine takes when a user tries to delete or update a key to which existing foreign keys point
-
```
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: **no action, set null, set default**

Integrity Constraint Violation During Transactions

- **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 (will discuss later)

SQL Data Types and Schemas

Built-in Data Types in SQL

- **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

Index Creation

- **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)*
- Indices are data structures used to speed up access to records with specified values for index attributes

  select *
  from *student*
  where *ID* = '12345'
  - Can be executed by using the index to find the required record, without looking at all records of student
  - *More on indices in Chapter 9*

User-Defined Types

- **create type** construct in SQL creates user-defined type (alias, like typedef in C)

  create type *Dollars* **as numeric** (12,2) **final**
  - **create table** *department* (
    dept_name **varchar** (20),
    building **varchar** (15),
    budget Dollars);

Domains

- **create domain** construct in SQL-92 creates user-defined domain types
  create domain *person_name* **char**(20) **not null**
- Types and domains are similar
- Domains can have constraints, such as **not null**, specified on them
  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object – object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object – object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself

Authorization

Authorization

- Forms of authorization on parts of the database:
  - **Read** - allows reading, but not modification of data
  - **Insert** - allows insertion of new data, but not modification of existing data
  - **Update** - allows modification, but not deletion of data
  - **Delete** - allows deletion of data
- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices
  - **Resources** - allows creation of new relations
  - **Alteration** - allows addition or deletion of attributes in a relation
  - **Drop** - allows deletion of relations

Authorization Specification in SQL

- The **grant** statement is used to confer authorization
  grant <privilege list>
  on <relation name or view name> **to** <user list>
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator)

Privileges in SQL

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

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization
  revoke <privilege list>
  on <relation name or view name> **from** <user list>
- Example:
  revoke select on *branch* **from** U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold
- If <revokee-list> includes **public**, all users lose the privilege except those granted it explicitly
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation
- All privileges that depend on the privilege being revoked are also revoked

Roles

- **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 on Views

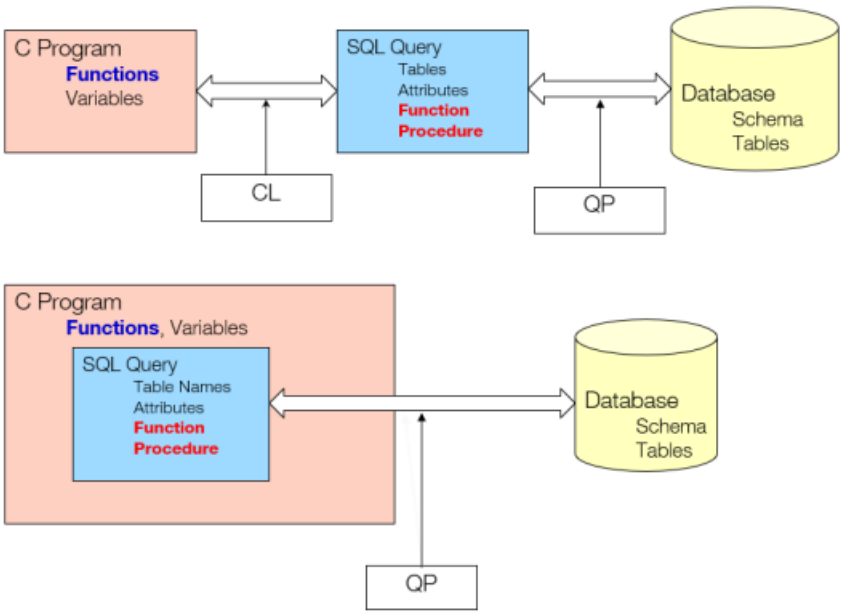
- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
  grant select on *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  select *
  from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor*?
  - creator of view did not have some permissions on *instructor*?

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**;

Functions and Procedural Constructs

Native Language← →Query Language



Functions and Procedures

- Functions / Procedures and Control Flow Statements were added in SQL:1999
  - **Functions/Procedures** can be written in **SQL itself**, or in an **external programming language** (like C, Java)
  - Functions written in an external languages are particularly useful with specialized data types such as images and geometric objects
    - ▷ Example: Functions to check if polygons overlap, or to compare images for similarity
  - Some database systems support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including **loops**, **if-then-else**, and **assignment**
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department:

```
create function dept_count (dept_name varchar(20))
        returns integer
        begin
        declare d_count integer;
                select count (*) into d_count
                from instructor
                where instructor.dept_name = dept_name
        return d_cont;
        end
```

- The function *dept_count* can be used to find the department names and budget of all departments with more that 12 instructors:

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

SQL functions (2)

- Compound statement: **begin** ... **end**
  May contain multiple SQL statements between **begin** and **end**.
- **returns** – indicates the variable-type that is returned (for example, integer)
- **return** – specifies the values that are to be returned as result of invoking the function
- SQL function are in fact **parameterized views** that generalize the regular notion of views by allowing parameters

Table Functions

- **Functions that return a relation as a result** added in SQL:2003
- Return all instructors in a given department:

```
create function instructor_of (dept_name char(20))
        returns table (
                ID varchar(5),
                name varchar(20),
                dept_name varchar(20)
                salary numeric(8, 2) )
        returns table
                (select ID, name, dept_name, salary
                from instructor
                where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *
from table  (instructor_of ('Music'))
```

SQL Procedures

- The dept_count function could instead be written as procedure:

```
create procedure dept_count_proc (
                in dept_name varchar (20), out d_count integer)
        begin
                select count(*) into d_count
                from instructor
                where instructor.dept_name = dept_count_proc.dept_name
        end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

- Procedures and functions can be invoked also from dynamic SQL
- SQL:1999 allows **overloading** - more than one function/procedure of the same name as long as the number of arguments and / or the types of the arguments differ

Language Constructs for Procedures and Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - *Warning*: **Most database systems implement their own variant of the standard syntax**
- Compound statement: **begin ... end**
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements

Language Constructs (2): while and repeat

- **while** loop:

```
while boolean expression do
        sequence of statements;
end while;
```

- **repeat** loop:

```
repeat
        sequence of statements;
 until boolean expression
 end repeat;
```

Language Constructs (3): for

- **for** loop
  - Permits iteration over all results of a query
- Find the budget of all departments:

```
declare n integer default 0;
for r as
    select budget from department
do
    set n = n + r.budget
end for;
```

Language Constructs (4): if-then-else

- Conditional statements
  - **if-then-else**
  - **case**
- **if-then-else** statement

```
if boolean expression then
        sequence of statements;
elseif boolean expression then
        sequence of statements;
 ...
else
        sequence of statements;
end if;
```

- The **if** statement supports the use of optional **elseif** clauses and a default **else** clause.
- Example procedure: registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
  - See book (page 177) for details

Language Constructs (5): Simple case

- Simple **case** statement

```
case variable
        when value1 then
                sequence of statements;
        when value2 then
                sequence of statements;
         ...
        else
                sequence of statements;
end case;
```

- The **when** clause of the **case** statement defines the value that when satisfied determines the flow of control

Language Constructs (6): Searched case

- Searched **case** statement

```
case
        when sql-expression = value1 then
                sequence of statements;
        when sql-expression = value2 then
                sequence of statements;
        ...
        else
                sequence of statements;
end case;
```

- Any supported SQL expression can be used here. These expressions can contain references to variables, parameters, special registers, and more.

Language Constructs (7): Exception

- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
        ...
        signal out_of_classroom_seats
        ...
end
```

  - The handler here is **exit** – causes enclosing **begin** . . . **end** to be terminate and exit
  - Other actions possible on exception

External Language Routines*

- SQL:1999 allows the definition of functions / procedures in an imperative programming language, (Java, C#, C or C++) which can be invoked from SQL queries
- Such functions can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions
- Declaring external language procedures and functions

```
create procedure dept_count_proc(
                in dept_name varchar(20),
                out count integer)
        language C
        external name '/usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
        language C
        external name '/usr/avi/bin/dept_count'
```

External Language Routines (2)*

- Benefits of external language functions/procedures:
  - More efficient for many operations, and more expressive power

External Language Routines (3)*: Security

- To deal with security problems, we can do one of the following:
  - Use **sandbox** techniques
    - That is, use a safe language like Java, which cannot be used to access/damage other parts of the database code
  - Run external language functions/procedures in a separate process, with no access to the database process' memory
    - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space

Triggers

Triggers

- A **trigger** defines a set of actions that are performed in response to an **insert**, **update**, or **delete** operation on a specified table
  - When such an SQL operation is executed, the trigger is said to have been **activated**
  - Triggers are **optional**
  - Triggers are defined using the **create trigger** statement
- Triggers can be used
  - To enforce data integrity rules via referential constraints and check constraints
  - To cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions to perform tasks such as issuing alerts
- To design a trigger mechanism, we must:
  - Specify the **events** / (like **update**, **insert**, or **delete**) for the trigger to executed
  - Specify the **time** (**BEFORE** or **AFTER**) of execution
  - Specify the **actions** to be taken when the trigger executes
- **Syntax of triggers may vary across systems**

Types of Triggers: BEFORE

- **BEFORE triggers**
  - Run before an **update**, or **insert**
  - Values that are being updated or inserted can be modified before the database is actually modified. You can use triggers that run before an update or insert to:
    - Check or modify values before they are actually updated or inserted in the database
      - Useful if user-view and internal database format differs
    - Run other non-database operations coded in user-defined functions
- **BEFORE DELETE triggers**
  - Run before a **delete**
    - Checks values (a raises an error, if necessary)

Types of Triggers (2): AFTER

- **AFTER triggers**
  - Run after an **update**, **insert**, or **delete**
  - You can use triggers that run after an update or insert to:
    - ▷ Update data in other tables
      - – Useful for maintain relationships between data or keep audit trail
    - ▷ Check against other data in the table or in other tables
      - – Useful to ensure data integrity when referential integrity constraints aren't appropriate, or
      - – when table check constraints limit checking to the current table only
    - ▷ Run non-database operations coded in user-defined functions
      - – Useful when issuing alerts or to update information outside the database

Row Level and Statement Level Triggers

There are two types of triggers based on the level at which the triggers are applied:

- **Row level triggers** are executed whenever a row is affected by the event on which the trigger is defined.
  - Let Employee be a table with 100 rows. Suppose an **update** statement is executed to increase the salary of each employee by 10%. Any row level **update** trigger configured on the table Employee will affect all the 100 rows in the table during this update.
- **Statement level triggers** perform a single action for all rows affected by a statement, instead of executing a separate action for each affected row.
  - Used for each **statement** instead of for each **row**
  - Uses **referencing old table** or **referencing new table** to refer to temporary tables called **transition tables** containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

Triggering Events and Actions in SQL

- Triggering event can be an **insert, delete or update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
    begin atomic
      set nrow.grade = null;
    end;
```

Trigger to Maintain credits earned value

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <>'F' and nrow.grade is not null
        and (orow.grade = 'F' or orow.grade is null)
begin atomic
        update student
        set tot_cred= tot_cred +
            (select  credits
            from course
            where  course.course_id=nrow.course_id)
        where student.id = nrow.id;
end;
```

How to use triggers?

- The optimal use of DML triggers is for short, simple, and easy to maintain write operations that act largely independent of an applications business logic.
- Typical and recommended uses of triggers include:
  - Logging changes to a history table
  - Auditing users and their actions against sensitive tables
  - Adding additional values to a table that may not be available to an application (due to security restrictions or other limitations), such as:
    - Login/user name
    - Time an operation occurs
    - Server/database name
  - Simple validation

How not to use triggers?

- Triggers are like Lays: *Once you pop, you can't stop*
- One of the greatest challenges for architects and developers is to ensure that
  - triggers are used only as needed, and
  - to not allow them to become a one-size-fits-all solution for any data needs that happen to come along
- Adding triggers is often seen as faster and easier than adding code to an application, but the cost of doing so is compounded over time with each added line of code

How to use triggers? (2)

- Triggers can become dangerous when:
  - There are too many
  - Trigger code becomes complex
  - Triggers go cross-server - across databases over network
  - Triggers call triggers
  - Recursive triggers are set to ON. This database-level setting is set to off by default
  - Functions, stored procedures, or views are in triggers
  - Iteration occurs