



# Week 3 Lecture 1

▼ Class	BSCCS2001
🕒 Created	@September 25, 2021 9:05 AM
🔗 Materials	
☰ Module #	11
▼ Type	Lecture
☰ Week #	3

## SQL Examples

### SELECT DISTINCT

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

classroom

<u>Aa</u> building	# room_number	# capacity
<u>Packard</u>	101	500
<u>Painter</u>	514	10
<u>Taylor</u>	3128	70
<u>Watson</u>	100	30
<u>Watson</u>	120	50

- Query:

```
SELECT DISTINCT building
FROM classroom
WHERE capacity < 100;
```

- Output:

<u>Aa</u> building
--------------------

<div>Aa</div> building
<u>Painter</u>
<u>Taylor</u>
<u>Watson</u>

### SELECT ALL

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

classroom

<div>Aa</div> building	<div>#</div> room_number	<div>#</div> capacity
<u>Packard</u>	101	500
<u>Painter</u>	514	10
<u>Taylor</u>	3128	70
<u>Watson</u>	100	30
<u>Watson</u>	120	50

- Query:

```
SELECT ALL building
FROM classroom
WHERE capacity < 100;
```

- Output:**

<div>Aa</div> building
<u>Painter</u>
<u>Taylor</u>
<u>Watson</u>
<u>Watson</u>

**NOTE:** The duplicate retention is default and hence it is a common practice to skip **ALL** immediately after **SELECT**

### Cartesian Product

- Find the list of all students of departments which have a budget < \$100K

```
SELECT name, budget
FROM student, department
WHERE student.dept_name = department.dept_name AND budget < 100000;
```

<div>Aa</div> name	<div>#</div> budget
<u>Brandt</u>	50000
<u>Peltier</u>	70000
<u>Levy</u>	70000
<u>Sanchez</u>	80000
<u>Snow</u>	70000
<u>Aoi</u>	85000
<u>Bourikas</u>	85000
<u>Tanaka</u>	90000

- The above query 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 above case can be framed by renaming the table 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;
```

Aa studentname	# deptbudget
Brandt	50000
Peltier	70000
Levy	70000
Sanchez	80000
Snow	70000
Aoi	85000
Bourikas	85000
Tanaka	90000

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

## SELECT: AND and OR

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

instructor

# id	Aa 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

Aa dept_name	≡ building	# budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000

Aa dept_name	≡ building	# budget
<u>Finance</u>	Painter	120000
<u>History</u>	Painter	50000
<u>Music</u>	Packard	80000
<u>Physics</u>	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:

Aa name
<u>Srinivasan</u>
<u>Wu</u>
<u>Einstein</u>
<u>Gold</u>
<u>Katz</u>
<u>Singh</u>
<u>Crick</u>
<u>Brandt</u>
<u>Kim</u>

## String Operations

- From the `course` relation in the figure, find the titles of all the courses whose `course_id` has 3 alphabets indicating the department

course

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

- Query:

```
SELECT title
FROM course
WHERE course_id LIKE '___-%'; -- 3 underscores
```

- Output:

Aa title
<u>Intro. to Biology.</u>
<u>Genetics</u>
<u>Computational Biology.</u>
<u>Investment Banking</u>
<u>World History.</u>
<u>Physical Principles</u>

- 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.

student

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

- Query:

```
SELECT name, dept_name, tot_cred
FROM student
ORDER BY dept_name ASC, tot_cred DESC;
```

- Output:

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

Aa name	dept_name	# tot_cred
<u>Snow</u>	Physics	0

How is this sort happening?

- The list is first sorted in alphabetic order of `dept_name`
- Within each department, it is sorted in decreasing order of total credits

## IN Operator

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

teaches

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

- Query:

```
SELECT course_id
FROM teaches
WHERE semester IN ('Fall', 'Spring')
AND year = 2018;
```

- Output:

course_id
<u>CS-315</u>
<u>FIN-201</u>
<u>MU-199</u>
<u>HIS-351</u>
<u>CS-101</u>
<u>CS-319</u>
<u>CS-319</u>

- **NOTE:** Now we can use **DISTINCT** to remove duplicates

## Set Operations: UNION

- For the same question in the above table, we can find the solution using **UNION** operator as follows:
  - 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:

<u>Aa</u> course_id
<u>CS-101</u>
<u>CS-315</u>
<u>CS-319</u>
<u>FIN-201</u>
<u>HIS-351</u>
<u>MU-199</u>

- **NOTE: UNION** removes all the duplicates. If we use **UNION ALL** instead of **UNION**, we get the same set of tuples as in the above example

### Set Operations: INTERSECT

- From the *instructor* relation in the figure, find the names of all the instructors who taught in either Computer Science department or the Finance department and whose salary is > 80,000

**instructor**

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

- Query:

```
SELECT name
FROM instructor
WHERE dept_name IN ('Comp. Sci.', 'Finance')
INTERSECT
SELECT name
FROM instructor
WHERE salary > 80000;
```

- Output:

<u>Aa</u> name
<u>Srinivasan</u>
<u>Katz</u>

- **NOTE:** The same thing can be achieved by using the query:

```
SELECT name FROM instructor WHERE dept_name IN ('Comp. Sci.', 'Finance') AND salary < 80000;
```

Set Operation: EXCEPT

- From the `instructor` relation in the figure, find the names of all the instructors who taught in either the Computer Science department or the Finance department and whose salary is either  $\geq 90,000$  or  $\leq 70,000$

instructor

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

- Query:

```
SELECT name
FROM instructor
WHERE dept_name IN ('Comp. Sci.', 'Finance')
EXCEPT
SELECT name
FROM instructor
WHERE salary < 90000 AND salary > 70000;
```

- Output:

Aa name
<u>Srinivasan</u>
<u>Brandt</u>
<u>Wu</u>

- NOTE:** The same can be achieved by using the following query

```
SELECT name FROM instructor
WHERE dept_name IN ('Comp. Sci.', 'Finance')
AND (salary >= 90000 OR salary <= 70000);
```

Aggregate function: 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

classroom

Aa building	# room_number	# capacity
<u>Packard</u>	101	500
<u>Painter</u>	514	10
<u>Taylor</u>	3128	70
<u>Watson</u>	100	30
<u>Watson</u>	120	50



- Query:

```
SELECT building, AVG(capacity)
FROM classroom
GROUP BY building
HAVING AVG(capacity) > 25;
```

- Output:

<div>Aa</div> building	<div>≡</div> avg
<u>Taylor</u>	70.00
<u>Packard</u>	500.00
<u>Watson</u>	40.00

Aggregate function: MIN

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

instructor

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

- Query:

```
SELECT MIN(salary) AS least_salary FROM instructor;
```

- Output:

<div>Aa</div> least_salary
<u>40000</u>

Aggregate function: MAX

- From the *instructor* relation given above, find the highest salary drawn by any instructor among all the instructors

- Query:

```
SELECT MAX(salary) AS highest_salary FROM instructor;
```

- Output:

<div>Aa</div> highest_salary
<u>95000</u>

Aggregate function: COUNT

- From the `instructor` relation given above, find the number of instructors in each department
  - Query:

```
SELECT dept_name, COUNT(id) AS ins_count
FROM instructor
GROUP BY dept_name;
```

- Output:

dept_name	ins_count
Comp. Sci.	3
Finance	2
Music	1
Physics	2
History	2
Biology	1
Elec. Eng.	1

Aggregate function: SUM

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

course

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

- 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





# Week 3 Lecture 2

▼ Class	BSCCS2001
🕒 Created	@September 25, 2021 5:30 PM
🔗 Materials	
☰ Module #	12
▼ Type	Lecture
☰ Week #	3

## Intermediate SQL

### Nested sub-queries

- SQL provides a mechanism for the nesting of sub-queries
- A **sub-query** is a **SELECT-FROM-WHERE** expression that is nested within another query
- The nesting can be done in the following SQL query

```
SELECT  $A_1, A_2, \dots, A_n$   
FROM  $r_1, r_2, \dots, r_m$   
WHERE  $P$ 
```

as follows:

- $A_i$  can be replaced by a sub-query that generates a single value
  - $r_i$  can be replace by any valid sub-query
  - $P$  can be replaced with an expression of the form:  
B <operation> (sub-query)  
where B is an attribute and <operation> is to be defined later
- Input of a query → One or more relations
  - Output of a query → Always a single relation

### Subqueries in WHERE clause

- Typical use of subqueries is to perform tests

- For set membership
- For set comparisons
- For set cardinality

## Set Membership

- Find the 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);
```

- 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 simple manner. The formulation above is just to simply 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';
```

- The same above query using **SOME** clause

```
SELECT name
FROM instructor
WHERE salary > SOME (
  SELECT salary
  FROM instructor
  WHERE dept_name = 'Biology');
```

## Definition of "SOME" clause

- $F \text{ <comp> SOME } r \Leftrightarrow \exists t \in r \text{ such that } (F \text{ <comp> } t)$   
where <comp> can be:  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ,  $\neq$
  - SOME** represents existential quantification [The entity in `"()"` is a tuple here]
- $5 < \text{SOME } (0, 5, 6) \rightarrow \text{true}$
- $5 < \text{SOME } (0, 5) \rightarrow \text{false}$
- $5 = \text{SOME } (0, 5) \rightarrow \text{true}$
- $5 \neq \text{SOME } (0, 5) \rightarrow \text{true} \# \text{ as } 0 \neq 5$

(= **SOME**)  $\equiv$  **IN**

However, ( $\neq$  **SOME**)  $\neq$  **NOT IN**

## Set Comparison - "ALL" clause

- Find the names of all the 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

- $F < \text{comp} > \mathbf{ALL} \ r \Leftrightarrow \forall t \in r \text{ such that } (F < \text{comp} > t)$   
where  $< \text{comp} >$  can be:  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ,  $\neq$
- ALL** represents universal quantification [The entity in "()" is a tuple here]

$5 < \mathbf{ALL} (0, 5, 6) \rightarrow \text{false}$

$5 < \mathbf{ALL}(6, 10) \rightarrow \text{true}$

$5 = \mathbf{ALL}(4, 5) \rightarrow \text{false}$

$5 \neq \mathbf{ALL}(4, 5) \rightarrow \text{true}$

( $\neq$  **ALL**)  $\equiv$  **NOT IN**

However, (= **ALL**)  $\neq$  **IN**

## Test for empty relations: "EXISTS"

- The **EXISTS** construct returns the value true if the argument subquery is non-empty
  - $\mathbf{EXISTS} \ r \Leftrightarrow r \neq \emptyset$
  - $\mathbf{NOT EXISTS} \ r \Leftrightarrow r = \emptyset$

## Use of "EXISTS" clause

- Yet another way of specifying the query "Find all the 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 by 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 the courses offered by the Biology department
- Second nested query lists all the courses a particular student has taken
- **NOTE:**  $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- **NOTE:** Cannot write this query string = **ALL** and its variants

## Test for absence of duplicate tuples: "UNIQUE"

- The **UNIQUE** construct tests whether a subquery has any duplicate tuples in its results
- The **UNIQUE** construct evaluates to "true" if a given subquery contains no duplicates
- Find all the courses that were offered at most once in 2009

```
SELECT T.course_id
FROM course AS T
WHERE UNIQUE (
  SELECT R.course_id
  FROM course AS R
  WHERE T.course_id = R.course_id
  AND R.year = 2009);
```

## 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:** We do not need a **HAVING** clause
- Another way to write the 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 the 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: Where a single value is expected
- List all departments along with the number of instructors in each department

```
SELECT dept_name, (
    SELECT COUNT(*)
    FROM instructor
    WHERE department.dept_name = instructor.dept_name)
AS num_instructors
FROM department;
```

- Runtime error occurs if subquery returns more than one result tuple

## Modifications 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 instructors;
```

- 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');
```

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

## Insertion

- Add a new tuple to the 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);
```

- 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 problems

## Updates

- Increase salaries of instructors whose salary is over \$100,000 by 3% and all other by 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

## 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 the 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);
```

- Set `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;
```



# Week 3 Lecture 3

▼ Class	BSCCS2001
🕒 Created	@September 26, 2021 11:24 AM
🔗 Materials	
☰ Module #	13
▼ Type	Lecture
☰ Week #	3

## Intermediate SQL (part 2)

### 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 conditions)
- 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 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*

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

- Relation *prereq*

<div>Aa</div> course_id	<div>≡</div> prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

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

Inner JOIN

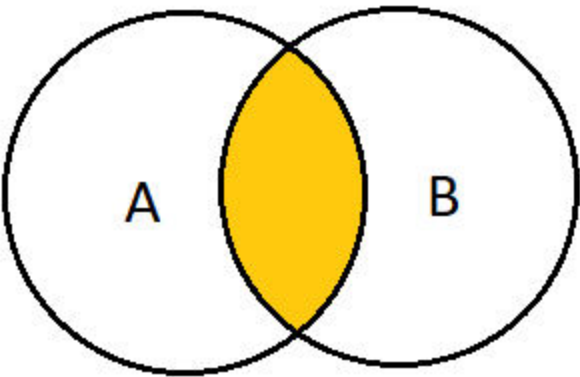
- *course* **INNER JOIN** *prereq*

<div>Aa</div> Name	<div>≡</div> title	<div>≡</div> dept_name	<div>#</div> credits	<div>≡</div> prere_id	<div>≡</div> 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 2<sup>nd</sup> *course\_id* field is skipped

<div>Aa</div> course_id	<div>≡</div> title	<div>≡</div> Column	<div>#</div> credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

<div>Aa</div> course_id	<div>≡</div> 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 results of the join
- Uses *null* values

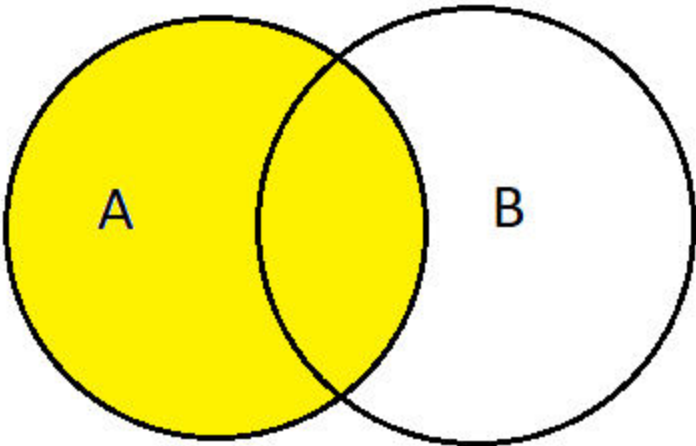
Left Outer JOIN

- *course* **NATURAL LEFT OUTER JOIN** *prereq*

<u>Aa</u> course_id	<u>≡</u> title	<u>≡</u> dept_name	<u>#</u> credits	<u>≡</u> prere_id
<u>BIO-301</u>	Genetics	Biology	4	BIO-101
<u>CS-190</u>	Game Design	Comp. Sci.	4	CS-101
<u>CS-315</u>	Robotics	Comp. Sci.	3	null

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

<u>Aa</u> course_id	<u>≡</u> prereq_id
<u>BIO-301</u>	BIO-101
<u>CS-190</u>	CS-101
<u>CS-347</u>	CS-101



Right Outer JOIN

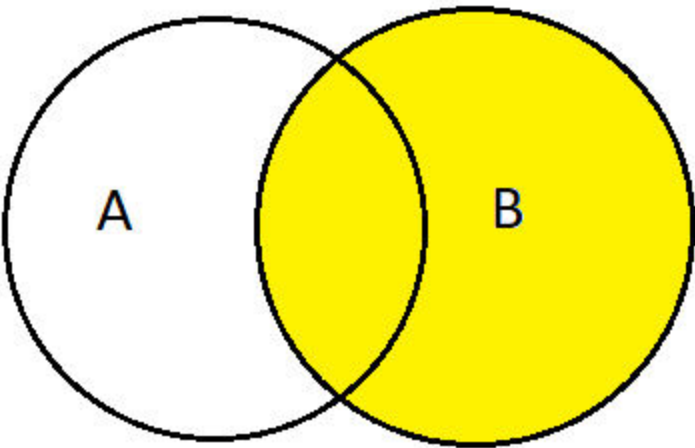
- *course* **NATURAL RIGHT OUTER JOIN** *prereq*

<u>Aa</u> course_id	<u>≡</u> title	<u>≡</u> dept_name	<u>≡</u> credits	<u>≡</u> prere_id
<u>BIO-301</u>	Genetics	Biology	4	BIO-101

<u>Aa</u> course_id	≡ title	≡ dept_name	≡ credits	≡ prere_id
<u>CS-190</u>	Game Design	Comp. Sci.	4	CS-101
<u>CS-347</u>	null	null	null	CS-101

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

<u>Aa</u> course_id	≡ prereq_id
<u>BIO-301</u>	BIO-101
<u>CS-190</u>	CS-101
<u>CS-347</u>	CS-101



### Joined relations

- **Join operations** take two relations and return a relation as the result
- 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 ( $A_1, A_2, \dots, A_n$ )

### Full outer JOIN

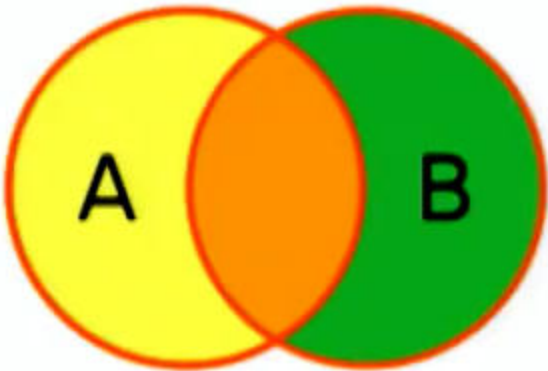
- *course* **NATURAL FULL OUTER JOIN** *prereq*

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

<u>Aa</u> course_id	≡ title	≡ dept_name	≡ credits	≡ prereq_id
<u>CS-347</u>	null	null	null	CS-101

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

<u>Aa</u> course_id	≡ prereq_id
<u>BIO-301</u>	BIO-101
<u>CS-190</u>	CS-101
<u>CS-347</u>	CS-101



### Joined Relations - Example

- *course* **INNER JOIN** *prereq* **ON**  
*course.course\_id = prereq.course\_id*

<u>Aa</u> course_id	≡ title	≡ dept_name	# credits	≡ prere_id	≡ courseid
<u>BIO-301</u>	Genetics	Biology	4	BIO-101	BIO-301
<u>CS-190</u>	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*





<u>Aa</u> course_id	≡ title	≡ dept_name	# credits	≡ prere_id	≡ courseid
<u>BIO-301</u>	Genetics	Biology	4	BIO-101	BIO-301
<u>CS-190</u>	Game Design	Comp. Sci.	4	CS-101	CS-190
<u>CS-315</u>	Robotics	Comp. Sci.	3	null	null

- *course* **NATURAL RIGHT OUTER JOIN** *prereq*

<u>Aa</u> course_id	≡ title	≡ dept_name	≡ credits	≡ prere_id
<u>BIO-301</u>	Genetics	Biology	4	BIO-101
<u>CS-190</u>	Game Design	Comp. Sci.	4	CS-101
<u>CS-347</u>	null	null	null	CS-101

- *course* **FULL OUTER JOIN** *prereq* **USING** (*course\_id*)

<u>Aa</u> course_id	≡ title	≡ dept_name	≡ credits	≡ prere_id
<u>BIO-301</u>	Genetics	Biology	4	BIO-101
<u>CS-190</u>	Game Design	Comp. Sci.	4	CS-101

 course_id	 title	 dept_name	 credits	 prere_id
<u>CS-315</u>	Robotics	Comp. Sci.	3	null
<u>CS-347</u>	null	null	null	CS-101

## Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database)
- Consider a person who needs to know an 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 the 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;
```

## View 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 phsics_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  $v_1$  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 on  $v_2$  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

## 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 VALUE('69987', 'White', 'Taylor');
```

- Which department, if multiple departments in Taylor?

- What if no department is present 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 when we insert `('25566', 'Brown', 'Biology', 100000)` into the `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 data
  - Need to maintain the view, by updating the view whenever the underlying relations are updated



# Week 3 Lecture 4

▼ Class	BSCCS2001
🕒 Created	@September 26, 2021 3:18 PM
🔗 Materials	
☰ Module #	14
▼ Type	Lecture
☰ Week #	3

## Intermediate SQL (part 3)

### Transactions

- It is a unit of work
- Atomic transaction
  - Either something is fully executed or it is rolled back as if it never occurred
  - **Example:** Bank account transactions, when transferring money from one account to another, the transaction should either happen or not happen at all.

It should not fail at a stage where money is deducted from one account and not added to the other account

- 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 guard against accidental damage to the database by ensuring that the 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( $A_1, A_2, \dots, A_m$ )**
  - The unique specification states that the attributes  $A_1, A_2, \dots, A_m$  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  $A$ .
  - Here,  $A$  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
  ...
)
```

- 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

## SQL Data Types and Schemas

### Built-in data types in SQL

- **DATE**: Dates, containing an (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 the 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 students

## User-defined types

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

```
CREATE TYPE Dollars AS NUMERIC(2, 2) FINAL;
```

```
CREATE TABLE department (  
  dept_name VARCHAR(20),  
  building VARCHAR(15),  
  budget Dollars);
```

## Domains

- **CREATE TYPE** 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 than the large object itself

## 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 of 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
- 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 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 U1,U2,U3
```

- **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 U1,U2,U3
```

- `<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 is
  - *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;
```





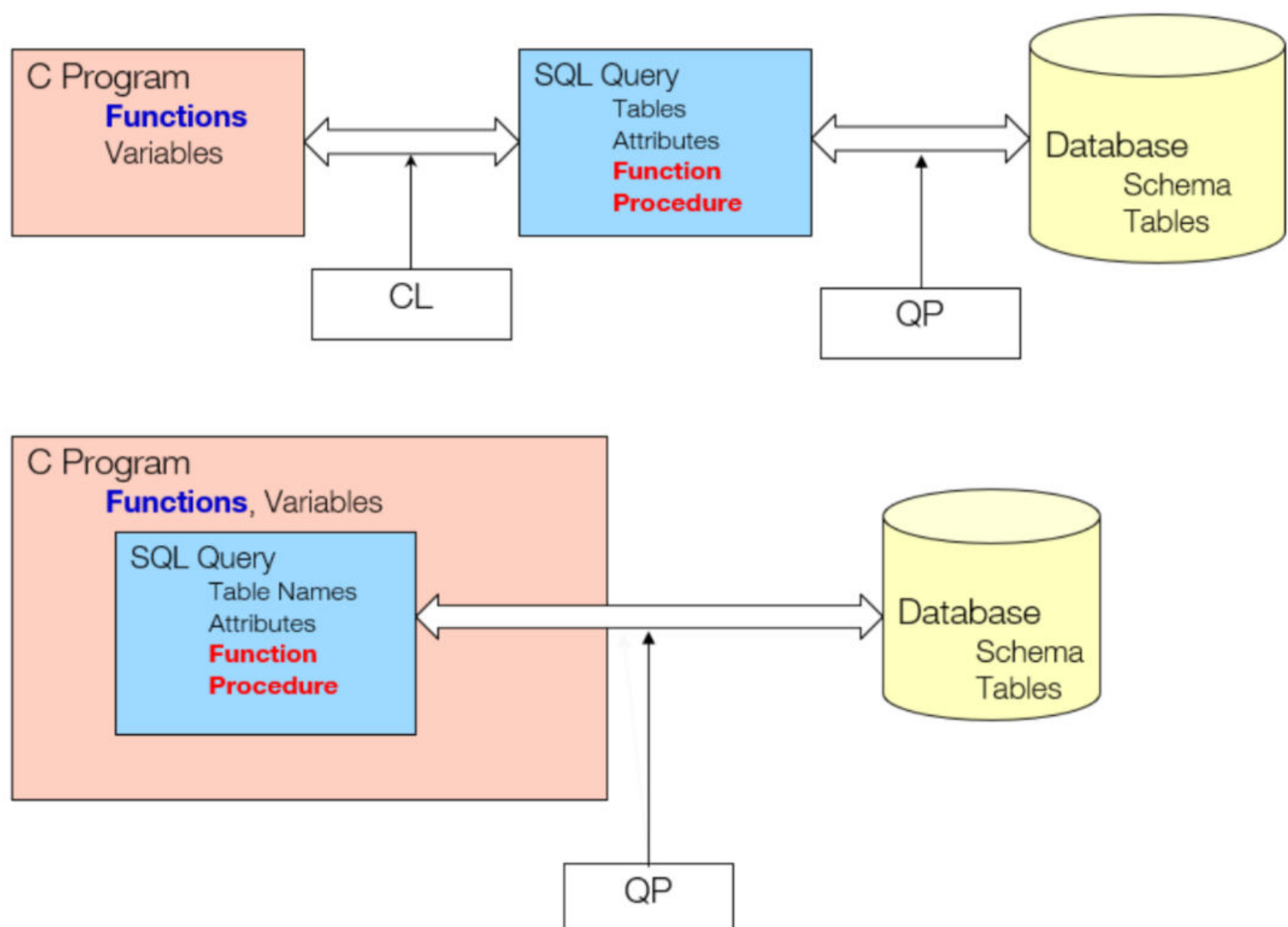
# Week 3 Lecture 5

▼ Class	BSCCS2001
🕒 Created	@September 26, 2021 10:24 PM
🔗 Materials	
☰ Module #	15
▼ Type	Lecture
☰ Week #	3

## Advanced SQL

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, etc
  - Functions written in an external language 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))
RETURN INTEGER
BEGIN
  DECLARE d_count integer;
  SELECT COUNT(*) INTO d_count
  FROM instructor
  WHERE instructor.dept_name = dept_name
  RETURN d_count;
END
```

- The function dept\_count can be used to find the department names and budget of all departments with more than 12 instructors:

```
SELECT dept_name, budget
FROM department
WHERE dept_count (dept_name) > 12;
```

- Compound statement: **BEGIN ... END**

May contain multiple SQL statements between **BEGIN** and **END**

- **RETURNS:** indicates the variable-type that is returned (eg: integer)
- **RETURN:** specifies the values 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))
RETURN 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

- **WHILE** loop:

```
WHILE boolean expression DO
  sequence of statements;
END WHILE;
```

- **REPEAT** loop:

```
REPEAT
  sequence of statements;
UNTIL boolean expression
END REPEAT;
```

- **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;
```

- 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 the capacity is exceeded
- 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
- 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.
- 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 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. The 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_count 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'

```

- Benefits of external language functions/procedures:
  - More efficient for many operations and more expressive power
- Drawbacks:
  - Code to implement function may need to be loaded into the DB system and executed in the DB system's address space
    - Risk of accidental corruption of the DB structures
    - Security risk, allowing users access to unauthorized data
  - There are alternatives, which provide good security at the cost of performance
  - Direct execution in the DB system's space is used when efficiency is more important than security

## External Language Routines: 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 DB code
  - Run external language functions/procedures in a separate process, with no access to the DB process' memory
    - Parameters and results communicated via the inter-process communication
- Both have performance overheads
- Many DB systems support both above approaches as well as direct executing in DB system address space

## 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 DB is actually modified.

You can use triggers that run before an **UPDATE** or **INSERT** to ...

  - Check or modify the values before they are actually updated or inserted in the DB
    - Useful if user-view and internal DB format differs
  - Run other non-DB operations coded in user-defined functions
- **BEFORE DELETE** triggers
  - Run before a **DELETE**
    - Checks value (and raises an error, if necessary)

## Types of Triggers: AFTER

- **AFTER** triggers
  - Run after an **UPDATE**, **INSERT** or **DELETE**
  - You can use triggers than run after an update or insert to:
    - Update data in other tables
      - Useful to 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
      - When table check constraints limit checking to the current table only
    - Run non-DB operations coded in user-defined functions
      - Useful when issuing alerts or to update information outside the DB

## 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 the 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 application 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

**Source:** SQL Server triggers: The good and the scary

## How not to use triggers?

- Triggers are like Lays: *Once you pop, you cannot 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.

**Source:** SQL Server triggers: The good and the scary

### **Alright then, how to use triggers?**

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

**Source:** SQL Server triggers: The good and the scary