SQL: Finish Nested Queries & CUD Operation

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. More On SOME & ALL, Introduce EXISTS & NOT EXISTS

2. CUD Operations

3. Bonus Material: "Equivalent Queries"

Some material from lectures created by Professor Greiner: used with permission

Context

- We're solidifying our grasp of SQL essentials ☺
- Previous lecture began the important topic of nested queries
  - Especially important because it gives the ability to develop "iteratively"!
- Today:
  - Finish the topic of nested queries
  - CUD operations

More On SOME & ALL, Introduce EXISTS & NOT EXISTS

Brain-Teasers: I

- The "truth value" of x comparatorOp SOME R ...
  - (R is a relation, *comparatorOp* is one of $<, \leq, >, =, \neq$)
- ...is equivalent to $\exists t \in R$ such that (x comparatorOp t)

> Spend a little time to make sure you agree with the following ...

```
1    (5 < some (0, 5, 6)) = true
2    (5 < some (0, 5)) = false
3    (5 = some (0, 5)) = true
4    (5 <> some (0, 5)) = true
```

- Note: = *some* is equivalent to IN
- Note: <> *some* is not equivalent to NOT IN

Brain-Teasers: II

- The "truth value" of x comparatorOp ALL R ...
  - (R is a relation, *comparatorOp* is one of $<, \leq, >, =, \neq$)
- ...is equivalent to $\forall t \in R$ (x comparatorOp t)

Spend a little time to make sure you agree with the following ...

```
1    (5 < all (0, 5, 6)) = false
2    (5 < all (6, 10)) = true
3    (5 = all (4, 5)) = false
4    (5 <> all (4, 6)) = true
```

- Note: <> *all* is equivalent to NOT IN
- Note: = *all* is not equivalent to IN

Correlated Nested Queries

- When a WHERE clause condition in a nested query references a relation attribute declared in the outer query ...
  - We say that the two queries are correlated
- Semantics: evaluate the nested query once for each tuple in the outer query
  - In contrast to the examples we've been looking at in which the nested query is evaluated independently of the outer query

```
1   -- For each EMPLOYEE tuple, evaluate the nested query, which retrieves the
2   -- Essn values for all DEPENDENT tuples with the same sex and first name
3   -- as that EMPLOYEE tuple ...
4   -- If the Ssn value of the EMPLOYEE tuple is in the result
5   -- of the nested query, then select that EMPLOYEE tuple
6   --
7   -- (Comment by me: "not a very useful query", pedagogic only)
8   SELECT E.Fname, E.Lname
9     FROM EMPLOYEE AS E
10      WHERE E.Ssn IN (SELECT D.Essn
11                        FROM DEPENDENT AS D
12                         WHERE   E.Fname = D.Dependent_name
13                           AND E.Sex = D.Sex);
```

Introducing EXISTS

- EXISTS is a Boolean function and can therefore be used in a WHERE clause
- We use EXISTS to check whether the result of a nested query is empty or not
  - Return true *iff* the nested query result contains at least one tuple
- We use EXISTS (and NOT EXISTS) in conjunction with a correlated nested query
- Example below reimplements the previous query using EXISTS

```
1  SELECT E.Fname, E.Lname
2    FROM EMPLOYEE AS E
3      WHERE E.Ssn IN
4        (SELECT   D.Essn
5          FROM DEPENDENT AS D
6          WHERE E.Fname = D.Dependent_name
7            AND E.Sex = D.Sex);
```

```
1  SELECT  E.Fname, E.Lname
2    FROM EMPLOYEE AS E
3    WHERE EXISTS
4      (SELECT FROM WHERE  *
5         DEPENDENT AS D
6         E.Ssn = D.Essn AND E.Sex = D.Sex
7           AND E.Fname = D.Dependent_name);
```

Empty Relation?

- **exists R** has the same truth value as $R \neq \emptyset$
- **not exists R** has the same truth value as $R = \emptyset$

Reimplement: *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 sub-query: inner sub-query that uses a "correlation name" from an outer query

Using NOT EXISTS to Test For Set Containment ("Superset")

Observation: given sets X and Y

- $X - Y = \varnothing \iff X \subseteq Y$
- In other words: relation $Y \supseteq X$

> Implication: *Is set X a superset of Y? ...*

Can be implemented in SQL as:

```
1       not exists (X except Y)
```

Set Containment: Example

Find all students who have taken all courses offered in the Biology department

```sql
-- Examine each student: does the set of all Biology courses
-- taken by this student contain the set of all Biology
-- courses?
SELECT DISTINCT S.ID, S.name
FROM student AS S
WHERE NOT EXISTS
  -- all courses offered in Biology
  ((SELECT course_id
      FROM course
      WHERE dept_name = 'Biology')
  EXCEPT
  -- all courses taken by a particular student
  (SELECT T.course_id
    FROM takes AS T
    WHERE S.ID = T.ID));
```

Which customers have ordered something?

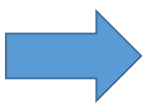
Customer

id	name
1	Joe
4	Mary
3	Scott
6	Elizabeth

Order

id	customer_id	item
105	4	Shoes
107	4	Pants
108	1	Pants
109	3	Tie

$$\{c \mid Customer(c) \wedge (\exists\ o.\ Order(o) \wedge c.id = o.customer\_id)\}$$



id	name
1	Joe
4	Mary
3	Scott

```
SELECT id, name
FROM Customer c
WHERE EXISTS (SELECT *
              FROM Order o
              WHERE c.id = o.customer_id);
```

EXISTS (subquery) returns a Boolean.

Subquery's SELECT irrelevant.

Which customers have not ordered anything?

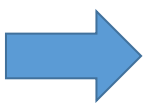
Customer

id	name
1	Joe
4	Mary
3	Scott
6	Elizabeth

Order

id	customer_id	item
105	4	Shoes
107	4	Pants
108	1	Pants
109	3	Tie

$$\{c \mid \text{Customer}(c) \wedge (\neg \exists\, o.\ \text{Order}(o) \wedge c.\text{id} = o.\text{customer\_id})\}$$



id	name
6	Elizabeth

```
SELECT id, name
FROM Customer c
WHERE NOT EXISTS (SELECT *
                  FROM Order o
                  WHERE c.id = o.customer_id);
```

Efficiency

```
SELECT id, name
FROM Customer c
WHERE EXISTS (SELECT *
                    FROM Order o
                    WHERE c.id = o.customer_id);
```

At first satisfying Order, stop and return TRUE.

```
SELECT id, name
FROM Customer c
WHERE NOT EXISTS (SELECT *
                        FROM Order o
                        WHERE c.id = o.customer_id);
```

At first satisfying Order, stop and return FALSE.

Logical equivalence of quantifiers

$$\neg \exists x \,.\, R(x) \;=\; \forall x. \neg R(x)$$

$$\neg \forall x \,.\, R(x) \;=\; \exists x. \neg R(x)$$

Universal quantification – no FORALL()!

Generally a less useful idea in SQL.

- $\exists x.\ R(x)$ – Exists some Order o such that the Customer c ordered it.
- $\forall x\ .\ R(x)$ – For all Orders o, Customer c ordered it.

$$\forall x\ .\ R(x) \;=\; \neg\, \exists x.\ \neg\, R(x)$$

```
SELECT id, name
FROM Customer c
WHERE FORALL (SELECT *
                    FROM Order o
                    WHERE c.id = o.customer_id);
```

```
SELECT id, name
FROM Customer c
WHERE NOT EXISTS (SELECT *
                        FROM Order o
                        WHERE c.id <> o.customer_id);
```

Another form of existential quantification

```
SELECT id, name
FROM Customer c
WHERE EXISTS (SELECT *
                    FROM Order o
                    WHERE c.id = o.customer_id);
```

```
SELECT id, name
FROM Customer
WHERE id = ANY (SELECT customer_id
                          FROM Order);
```

The operator is = ANY()

Another form of universal quantification

```
SELECT *
FROM Product p1
WHERE p1.price >= ALL (SELECT p2.price
                       FROM  Product p2);
```

```
SELECT *
FROM Product p1
WHERE NOT (p1.price < ANY (SELECT p2.price
                           FROM  Product p2));
```

Nested Queries: Can Be Used With FROM Clause

- We've focused our nested query discussion on usage with WHERE clause
- But: can also be used with a query's FROM clause
- After all: SELECT-FROM-where returns a relation
  - So: we can plug the relation returned by a nested query into any place where an SQL query expects to see a relation
  - Including the input to a FROM clause ☺
- Example: *"Find the average instructor salary of those departments where the average salary is greater than $42,000."*

```
1  SELECT dept_name, avg_salary
2    FROM (SELECT dept_name, AVG (salary) AS avg_salary
3            FROM instructor
4            GROUP BY dept_name)
5    WHERE avg_salary > 42000;
```

Naming Nested Query Artifacts

- You can name a nested query's result set
- You can rename the attributes from a nested query's result set

```
1  SELECT dept_name, avg_salary
2    FROM (SELECT dept_name, avg (salary)
3            FROM instructor
4            GROUP by dept_name)
5            AS dept_avg (dept_name, avg_salary)
6    WHERE avg_salary > 42000;
```

Nasty POSTGRESQL Bug

- Q: what's wrong with this query?

```
1  SELECT COUNT (made_only_recharge) AS made_only_recharge
2  FROM (
3      SELECT DISTINCT (identifiant) AS made_only_recharge
4      FROM cdr_data
5      WHERE CALLEDNUMBER = '0130'
6      EXCEPT
7      SELECT DISTINCT (identifiant) AS made_only_recharge
8      FROM cdr_data
9      WHERE CALLEDNUMBER != '0130'
10 )
```

- Per this discussion, the query is OK on Oracle, breaks for POSTGRESQL ☹
- A: there's nothing wrong with the query except that POSTGRESQL insists that each sub-query relation in a FROM clause must be given a name
  - Even if the name is never referenced! ☹
- But the truth is: good SQL style suggests that you always "alias" your sub-query
  - Never hurts to make your code more clear ☺

Nested Queries: Can Be Used With SELECT Clause

- You can even use a nested query in the SELECT clause!
- We refer to this scenario as a scalar sub-query
  - "Scalar": because you expect only a single value as the result
  - "Scalar": because runtime error if sub-query returns more than one result tuple ☺
- Example: *"List all departments along with the number of instructors in each department"*

```
1  SELECT dept_name, (SELECT count(*)
2                     FROM instructor
3                     WHERE department.dept_name = instructor.
                          dept_name)
4                    AS num_instructors
5    FROM department;
```

CUD Operations

Introduction

- We've been polishing our SQL query (the R from CRUD) skills …
- But ignored CUD operations ☹
- One reason: CUD operations are an SQL "add-on" to the theory of relational databases
  - A necessary feature for "commercial" systems
  - But not fundamentally necessary ☺
- Another reason: sophisticated CUD operations incorporate queries to specify the relational context to which the operation will be applied
- We're now ready to tackle this topic

Create

- Simplest form: use to add one or more tuples to a relation
- As we've discussed:
  - Constraints on data types are observed automatically
  - Integrity constraints (part of the DDL specification) are enforced
- Lazy version (don't do this!):
  - Can specify the attribute names implicitly by supplying the attribute values in the same order as they're specified in the CREATE TABLE DDL

```
1  insert into instructor
2      values ('10211', 'Smith', 'Biology', 66000)
```

- Better:
  - Specify attribute names explicitly, such that they are associated one-to-one with the supplied values

```
1  insert into instructor(name, ID, dep_name, salary)
2      values ('Smith', '10211', 'Biology', 66000)
```

Create With Missing Values

- If you omit values of certain columns
  - Default values will be used (per prior DDL statement)
  - Or the NULL value will be used

```
1  insert into instructor(name, ID, dep_name)
2  values ('Smith', '10211', 'Biology')
```

- The above example (implicitly) does not supply a value for the *salary* attribute
- As with previous slide: better to make explicitly state that you're supplying NULL for the *salary* attribute

Inserting From Another Relation

- Example of the beauty that comes from a set-based programming model
- Input to the "insert" operation is defined in terms of tuples
- Queries return tuples
- So: we can insert the result of a query!

```
1  INSERT into student
2    SELECT ID, name, dept_name, 0
3      FROM instructor
```

- This SQL add all instructors to the *student* relation, and sets their *tot_creds* attribute set to 0
- Can insert any result of an query, as long as compatible, into a table

Source of Insert Operation Can Be Any Query

- You can use the full power of a query to drive an "insert" include the SELECT FROM WHERE version
- Example: *"Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000"*

```
INSERT INTO instructor
  SELECT ID, name, dept_name, 18000
    FROM student WHERE dept_name = 'Music' AND
        total_cred > 144;
```

- Semantics: DBMS fully evaluates the query results before inserting results into relation
- Otherwise the following statement could cause an infinite loop ...

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

Adding records to a table

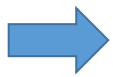
```
INSERT INTO Product
VALUES
   ('MiniGizmo', 15.99, 'GizmoWorks'),
   ('MiniWidget', 21.99, 'WidgetsRUs');

INSERT INTO Product (prod_name, prod_manufacturer)
VALUES
   ('NanoWidget', 'WidgetsRUs');
```

Explicitly listing column names is safer. Protects against table column changes.

Product

prod_name	prod_price	prod_manufacturer
'Gizmo'	$19.99	'GizmoWorks'
'Powergizmo'	$39.99	'GizmoWorks'
'Widget'	$19.99	'WidgetsRUs'
'HyperWidget'	$203.99	'Hyper'



Product

prod_name	prod_price	prod_manufacturer
'Gizmo'	$19.99	'GizmoWorks'
'Powergizmo'	$39.99	'GizmoWorks'
'Widget'	$19.99	'WidgetsRUs'
'HyperWidget'	$203.99	'Hyper'
'MiniGizmo'	$15.99	'GizmoWorks'
'MiniWidget'	$21.99	'WidgetsRUs'
'NanoWidget'	NULL	'WidgetsRUs'

Update

```
1  update instructor set salary = salary * 1.03
2  where salary > 100000
3
4  update instructor set salary = salary * 1.05
5  where salary <= 100000
```

- Use the UPDATE command to modify attribute values of one or more tuples
- Use a WHERE-clause to specify the tuples that will be modified
  - In other words: you use a query to specify the set of tuples that will be updated
- Then use a SET-clause to specify the attributes to be modified and their new values
  - Can selectively update a subset of attributes
- As discussed before, DBMS will enforce the referential and other integrity constraints that were specified as part of the DDL

Queries Are Completely Integrated With Updates

```sql
UPDATE instructor
  SET salary = salary * 1.05
  WHERE salary < (select AVG (salary) FROM instructor);
```

- Just because we're doing declarative programming, don't forget your basic programming skills!
- Example: statement order matters!
- Consider this update: *Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%*

```sql
UPDATE instructor
  SET salary = salary * 1.05 WHERE salary <= 100000;
UPDATE instructor
  SET salary = salary * 1.03 WHERE salary > 100000;
```

Q: can you spot the bug?

Statement Order Matters

```sql
UPDATE instructor
  SET salary = salary * 1.05 WHERE salary <= 100000;
UPDATE instructor
  SET salary = salary * 1.03 WHERE salary > 100000;
```

- A: some instructors may get two raises ☺
- Rather than rearranging the statement order, better to use SQL's conditional update syntax: the CASE statement

```sql
UPDATE instructor
  SET salary = CASE
                 WHEN salary <= 100000 THEN salary *
                     1.05
                 ELSE salary * 1.03
               END
```

Delete

- Use the DELETE command to remove tuples from a relation
- Include a WHERE-clause to specify the tuples to be deleted
- As discussed previously, the DBMS will enforce referential integrity
  - Example: quite difficult to delete tuples from a randomly selected table in the university database
- As with updates, a single statement can only delete tuples from a single table
- As with updates, if you specify CASCADE in the referential integrity constraint, a single statement can effect multiple tables ☺

Delete

- Set theory strikes again!
- If you omit the WHERE-clause from a DELETE statement, you're implicitly specifying "delete all tuples in the relation"
  - Equivalent to: WHERE TRUE, because every tuple satisfies the "empty condition"
  - Result: the table becomes an empty table
  - So: "be careful when you type" ☺
- *Delete all instructor tuples*

```
1 delete from instructor
```

- In general: the number of deleted tuples depends on the number of tuples in the relation that satisfy the WHERE-clause

```
1         delete from instructor where dept_name= 'Finance'
```

Adding data to a table from a file

PostgreSQL

Server file: `COPY Product FROM '/my/path/product.csv' CSV HEADER;`

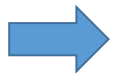
Must be logged in as a superuser, e.g., postgres. Server app must have file permission.

Client file: `\copy Product FROM '/my/path/product.csv' CSV HEADER`

Must use psql.

prod_name	prod_price	prod_manufacturer

```
prod_name, prod_price, prod_manufacturer
Gizmo,19.99,GizmoWorks
Powergizmo,39.99,GizmoWorks
Widget,19.99,WidgetsRUs
HyperWidget,203.99,Hyper
```



prod_name	prod_price	prod_manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$39.99	GizmoWorks
Widget	$19.99	WidgetsRUs
HyperWidget	$203.99	Hyper

Deleting data

```
DELETE FROM Student;

DELETE FROM Student
WHERE …;
```

However – often **never** want to delete!

- E.g., former customers, discontinued products

Soft delete:

- Have column to mark data as “inactive”.

Hard delete:

- Get rid of it!

Create, update, delete

Tables

```
CREATE TABLE Student (…);
```

```
ALTER TABLE Student
RENAME TO Scholar;
```

```
DROP TABLE Student;
```

Columns

```
ALTER TABLE Student
ADD COLUMN zip INT;
```

```
ALTER TABLE Student
RENAME COLUMN zip TO zip_code;
```

```
ALTER TABLE Student
ALTER COLUMN zip VARCHAR(9);
```

```
ALTER TABLE Student
DROP COLUMN state;
```

Constraints

Similar. Later.

Changing schemas affects other code

Obvious: Is a table or column is defined?

More subtle:

- SELECT *
- INSERT INTO – without listing column names

How can you know what's defined?

```
SELECT * FROM Table;
```

psql command	What it does
\d	List defined items. (So far, that's just tables.)
\dt	List defined tables.
\d *tablename*	Give details about table definition.

How to change schemas?

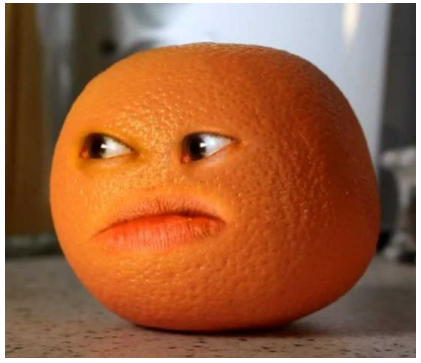
- Create new tables.
- Create new queries using the new tables.
- Copy or compute data for the new tables.
- Switch to using new tables and queries.

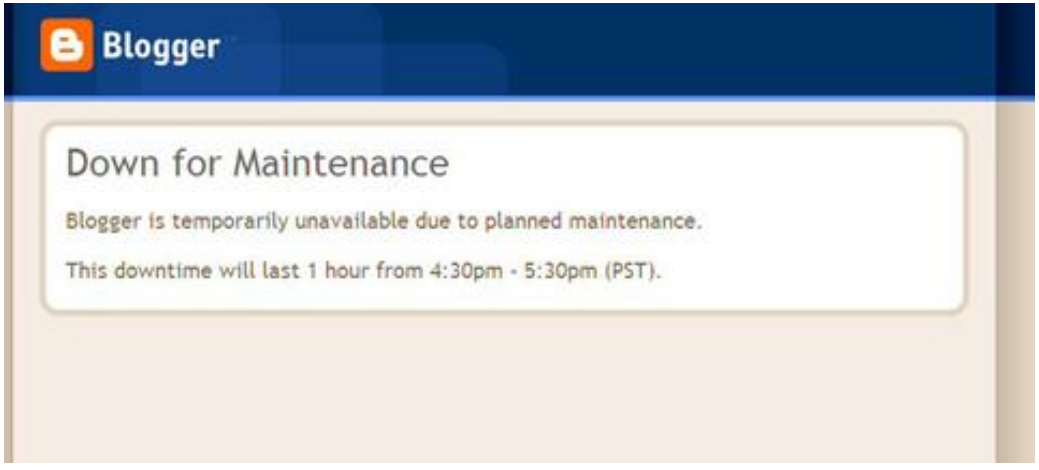
} Can take substantial time. Lots of data. Possibly distributed & replicated.

What about data that arrives during this process?

Traditional – shutdown

Annoying!





Best – invisible to user

A simple version:

- Create *shadow table* that includes schema changes.
- Create *triggers* in original table that forwards data updates to shadow table.
- Copy original table's data to shadow table.
- Rename shadow table to replace original table.

Can choose to only let some users see the new version (A/B testing).

Some DBMSes provide support for such *online* schema changes.

Bonus Material: “Equivalent Queries”

```
SELECT DISTINCT c.id, name
FROM Customer c
INNER JOIN Order o ON c.id = customer_id;
```

```
SELECT id, name
FROM Customer
WHERE id IN (SELECT customer_id
             FROM Order);
```

```
SELECT id, name
FROM Customer c
WHERE (SELECT Count(*)
       FROM Order o
       WHERE c.id = o.customer_id)
      >= 1;
```

```
SELECT id, name
FROM Customer c
WHERE EXISTS (SELECT *
              FROM Order o
              WHERE c.id = o.customer_id);
```

```
SELECT id, name
FROM Customer
WHERE id = ANY (SELECT customer_id
                FROM Order);
```

```
SELECT id
FROM Customer
WHERE EXISTS (SELECT *
                    FROM Order
                    WHERE Customer.id = customer_id);
```

```
SELECT id
FROM Customer
INTERSECT
SELECT customer_id
FROM Order;
```

Today's Lecture: Wrapping it Up

More On SOME & ALL, Introduce EXISTS & NOT EXISTS

CUD Operations

Bonus Material: "Equivalent Queries"

Readings

- Textbook discusses nested queries in Chapter 3.8
- Textbook discussed CUD operations in Chapter 3.9

Do an end-to-end reading of Chapter 3