SQL: Part (II)

March 10, 2023

Announcements

- Assignment 2 released. Due: March 26.
- Assignment 1 sample solution released on canvas.

Quick review (1)

SELECT A_1, A_2, ..., A_n
FROM R_1, R_2, ..., R_m
WHERE P;

A basic sql query can be expressed by a SELECT-FROM-WHERE statement as shown above.

- A_1, A_2, ..., A_n: a list of desired attributes in the query.
- R_1, R_2, ..., R_m: a list of tables accessed during the query evaluation.
- P: a filtering predicate involving the attributes from R_1, R_2, ..., R_m.

Quick review (2)

- Aggregation with grouping

```
-- Get the average credit of the students for each department.
SELECT dept_name, AVG(tot_cred)
FROM student
GROUP BY dept_name;
```

- Use HAVING to further filter group-by aggregation result

```
--  Get the average credit of the students for each
--  department with at least 50 students
SELECT dept_name, AVG(tot_cred)
FROM student
GROUP BY dept_name;
HAVING count(*) >= 50;
```

Q&A (1) Conversion between SQL data types

PostgreSQL does implicit type conversion when necessary.

Example

```
CREATE TABLE t (a integer);
INSERT INTO t VALUES ('123');
```

Explicit type conversion with CAST operator or :: syntax.

Example

```
SELECT CAST('123.45' AS REAL); -- result: 123.45
SELECT 123.45::INT; -- result: 123
```

Q&A (1) Conversion between SQL data types

Explicit type conversion is a must for correct results in some cases.

Example

```
SELECT 1/2; -- result: 0
SELECT 1::REAL/2; -- result: 0.5
```

Use PostgreSQL's pg_typeof function to get its actual type of an expression.

Example

```
SELECT pg_typeof(1/2); -- result: integer
SELECT pg_typeof(1::REAL/2); -- result: double precision
```

Q&A (2) PostgreSQL has persistent storage

- Each successful database change will be persisted, even if you encounter a system crash or a power failure.
- No need to import `ddl.sql` or `data.sql` every time you connect to the database.
- PostgreSQL achieves persistent storage by Write-Ahead Logging (WAL).
  - We will discuss about it later in the course.

Null Values

Null values

- Value unknown/inapplicable
- Used for each date type
- Special rules for dealing with NULL's

Example

```
SELECT ID, name
FROM instructor
WHERE salary IS NOT NULL;
```

Special rules for NULL

- Arithmetic operation:

$$\text{NULL } \mathbf{op} \text{ value/NULL} = \text{NULL}$$

- Comparison:

$$\text{NULL } \theta \text{ value/NULL} = \text{UNKNOWN}$$

- Aggregation functions ignore NULL, except COUNT(*).
  - COUNT(*) just conuts rows.
- Evaluating aggregation functions (except COUNT) on an empty bag returns NULL.
  - The count of an empty bag is 0.
- NULL cannot be used explicitly used an operand.
  - Wrong: NULL + 3, `x = NULL`
  - Correct: `x IS NULL`, `x IS NOT NULL`

Three-valued logic of SQL

- TRUE = 1, FALSE = 0, UNKNOWN = 0.5
- x AND $y = \min(x, y)$
- x OR $y = \max(x, y)$
- NOT $x = 1 - x$

- WHERE and HAVING only select rows for output if the condition evaluates to TRUE.

Pitfalls of NULL

NULL breaks many equivalences.

```
-- Not equivalent due to NULL
SELECT AVG(salary) FROM instructor;
SELECT SUM(salary)/COUNT(*) FROM instructor;
```

```
-- Not equivalent due to NULL
SELECT * from instructor;
SELECT * FROM instructor WHERE salary > 5000 OR salary <= 5000;
SELECT * FROM instructor WHERE salary = salary;
```

Joins

SQL join expressions

- An join expression applies an join operation to two relations and produces a new relation.
- They are typically used as subqueries in FROM clauses.

Theta join

```
R JOIN S ON join_condition
```

- The `join_condition` can be a general predicate over the relations being joined.

Example

```
-- student(ID, name, dept_name, tot_cred)
-- takes(ID, course_id, sec_id, semester, year, grade)

SELECT * FROM student JOIN takes ON student.ID = takes.ID;
SELECT * FROM student, takes WHERE student.ID = takes.ID;
```

Question. Is the keyword ON redundant?

Natural join

```
R NATURAL JOIN S
```

- Join tuples with the same values for all common attributes.
- Retain only one copy of each common column.

Example

```
-- student(ID, name, dept_name, tot_cred)
-- takes(ID, course_id, sec_id, semester, year, grade)
SELECT name, course_id
FROM student NATURAL JOIN takes

-- an equivalent query
SELECT name, course_id
FROM student, takes
WHERE student.ID = takes.ID
```

Natural join more relations

```
SELECT A_1,A_2,...,A_n
FROM R_1 NATURAL JOIN R_2 NATURAL JOIN ... R_k
WHERE P;
```

The USING keyword

Example

List the name of each student, along with the title of each course he/she takes.

```
-- A problematic query
SELECT name, title
FROM student NATURAL JOIN takes NATURAL JOIN course;
```

Problem: Attributes with the same name get equated unexpectedly in natural join.

Solution 1: Use WHERE and product to avoid joining on unrelated attributes.

```
SELECT name, title
FROM student NATURAL JOIN takes, course
WHERE takes.course_id = course.course_id;
```

Solution 2: The USING keyword specifies exactly which attributes should be joined.

```
SELECT name, title
FROM (student NATURAL JOIN takes) JOIN course USING (course_id);
```

Outer join motivation

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-325	Robotics	Comp. Sci.	3

Table: Course

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

Table: Prereq

List all the information of each course, along with the id's of its pre-required courses.

```
SELECT * from course NATURAL 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

Table: Course $\bowtie$ Prereq

Left outer join

A left outer join between R and S, denoted as $R \ltimes\!\bowtie S$ includes both

- rows in $R \bowtie S$, and
- dangling R rows padded with NULL's.

Example. `SELECT * from course NATURAL LEFT OUTER JOIN prereq;`

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-325	Robotics	Comp. Sci.	3	NULL

Table: Course $\ltimes\!\bowtie$ Prereq

- `('CS-325', 'Robotics', 'Comp. Sci.', 3)` is a dangling tuple in the relation Course when joining with Prereq, i.e., no tuples from Prereq matche it.

More outer join flavors

- A right outer join between R and S, denoted as $R \rtimes S$, includes rows in $R \bowtie S$ plus dangling S rows padded with NULL's.
- A full outer join, denoted as $R \ltimes\rtimes S$, includes all rows from $R \bowtie S$, plus
  - dangling R rows padded with NULL's
  - dangling S rows padded with NULL's

Example

```
-- Right outer join (1)
SELECT * FROM course NATURAL RIGHT OUTER JOIN  prereq;

-- Right outer join (2)
SELECT * FROM course RIGHT OUTER JOIN prereq
ON course.course_id = prereq.course_id;

-- Right outer join (3)
SELECT * FROM course RIGHT OUTER JOIN prereq
USING course_id;
```

Outer join examples

A	I
3	6
1	3
3	4

Table: $R(A, I)$

I	C	E
6	1	3
4	0	4
2	2	2

Table: $S(I, C, E)$

A	I	C	E
3	6	1	3
3	4	0	4

Table: Natural join $R \bowtie S$

A	I	C	E
3	6	1	3
3	4	0	4
1	3	NULL	NULL

Table: Left outer join $R \mathbin{⟕} S$

A	I	C	E
3	6	1	3
3	4	0	4
NULL	2	2	2

Table: Right outer join $R \mathbin{⟖} S$

A	I	C	E
3	6	1	3
3	4	0	4
1	3	NULL	NULL
NULL	2	2	2

Table: Full outer join $R \mathbin{⟗} S$

ON vs. WHERE

```
-- NULL values are preserved
SELECT * FROM course LEFT OUTER JOIN prereq
ON course.course_id = prereq.course_id;


-- NULL values are left out
SELECT * FROM course LEFT OUTER JOIN prereq ON TRUE
WHERE course.course_id = prereq.course_id;
```

Join recap

Join types

- inner join
- outer join

Join conditions

- on <predicates>
- using <A_1, ..., A_n>
- natural

Subqueries

Nested subqueries

A subquery is a SELECT-FROM-WHERE expression that nested in another query.

Example

List the id's of all courses offered in Fall 2017 but not in Spring 2018.

```
SELECT DISTINCT course_id --------------------------- outer query
FROM section
WHERE semester = 'Fall' AND year = 2017 AND
      course_id NOT IN (SELECT course_id   ---------- inner query
                        FROM section
                        WHERE semester = 'Spring' AND year = 2018);
```

Remark. Subqueries are enclosed by parentheses.

Nested subqueries (cont'd)

A subquery can be nested in a SELECT-FROM-WHERE statement almost anywhere

SELECT A_1, A_2, ..., A_n
FROM R_1, R_2, ..., R_m
WHERE P;

- FROM: every R_i can be replaced by a subquery.
- WHERE: P can include predicates involving subqueries.
- SELECT: every A_i can includes a subquery that generates a single value.

Subqueries in FROM clauses

- Subqueries can be used in FROM clauses since a subquery always return a relation.

```
SELECT dept_name, avg_salary
FROM (SELECT dept_name, avg(salary) AS avg_salary -- subquery
      FROM instructor
      GROUP BY dept_name)
WHERE avg_salary > 42000;
```

- Rename the relation returned by a subquery with keyword AS.

```
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;
```

Common table expression (WITH)

```
WITH R1(A_1, A_2, ...) As       -- a temporary relation R1
     (subquery_1),
     R2(B_1, B_2, ...) AS        -- a temporary relation R2
     (subquery_2),
     ...
SELECT ... FROM ... WHERE ...;  -- the actual query
```

- Defines temporary relations to be used by
  - other relations defined in the same WITH clause
  - the actual query.
- Only the result of the actual query are returned.
- Make queries more clear and readable.

WITH example

```
-- Find all the departments with total salary greater than
-- the average of the total salary of 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 via EXISTS

- `EXISTS (subquery)`: the subquery result is non-empty.

```
-- Find all courses offered in both Fall 2017 and Spring 2018 semester
SELECT course_id
FROM section as S
WHERE semester = 'Fall' AND year = 2017 AND
      EXISTS (SELECT * FROM section as T
              WHERE semester = 'Spring' AND year= 2018
                    AND course_id = S.course_id);
```

- Scoping rule: an attribute refers to the most closely nested relation with that attribute.

Subqueries via UNIQUE

- `UNIQUE (subquery)`: the subquery result contains no duplicates.

```
-- Find all courses that offered at most once in 2017.
SELECT T.course_id
FROM course as T
WHERE UNIQUE (SELECT R.course_id
              FROM section as R
              WHERE T.course_id= R.course_id
                    AND R.year = 2017);
```

Subqueries via IN

- `x IN (subquery)`: x is in the subquery result.
  - x can either an attribute A or a tuple $(A_1, \ldots, A_n)$

```
-- List the course_id's of all courses offered in Fall 2017
-- but not in Spring 2018
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND year = 2017 AND
      course_id NOT IN (SELECT course_id
                        FROM section
                        WHERE semester = 'Spring' AND year = 2018);
```

More subqueries in WHERE

- `x op ALL (subquery)`: x op t for all t in the subquery result.

```
-- Find the name 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');
```

- `x op SOME (subquery)`: x op t for some t in the subquery result.

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

Scalar subquery

- A subquery that returns a single tuple containing a single attribute is a scalar subquery.
- A scalar subquery can be used as a value in WHERE, SELECT and HAVING clauses.

```
-- List the name and ID of each instructor with the highest salary
SELECT name, ID
FROM instructor
WHERE salary = (SELECT MAX(salary)
                FROM instructor);
```

- Runtime error if subquery returns more than one row.
- NULL if subquery returns no rows.

Scalar subquery (cont'd)

```
-- List the name and the number of instructors of each department
SELECT dept_name,
       (SELECT COUNT(*) FROM instructor
        WHERE department.dept_name = instructor.dept_name
       ) AS num_instructors
FROM department;
```