SQL – Part 1

Running example: assumptions

- We use the following relation schemas:

  Student(Sid, Sname, Major, Byear)
  Course(Cno, Cname, Dept)
  Enroll(Sid, Cno, Grade)

- Sid and Cno are foreign keys in Enroll referencing Sid and Cno in Student and Course, respectively.
- We will also assume that Student has m tuples, Enroll has n tuples, and Course has k tuples.

SQL queries – basic form

- The simplest form of a SQL query is as follows:

SELECT *list of components of tuple variables*
FROM *list of tuple variables associated with relations*
WHERE *condition on components of tuple variables*;

Example of a simple SQL query

- Consider the query “Find the sids and names of students who major in CS.”
- In SQL, the formulation is,

```
SELECT S.Sid, S.Sname
FROM   Student S
WHERE S.Major = ‘CS’;
```

Semantics of a basic SQL query

- In the SQL query

```
SELECT S.Sid, S.Sname
FROM   Student S
WHERE S.Major = 'CS';
```

S is a tuple variable that ranges over all tuples in the Student relation. So, if this relation has 10 tuples, then S ranges over these 10 tuples.

- The semantics of this query is

```
for each tuple S in Student
    if S.Major = 'CS'
        then output (S.Sid, S.Sname)
```

Semantics (continued)

- It is important to realize that, when a tuple variable is associated with a relation, then that tuple variable is assigned, one-at-a-time, to each tuple in that relation.
- E.g., if Student is

Sid	Sname	Major	Byear	
s1	John	CS	1990	← S
s2	Ellen	Math	1995	← S
s3	Eric	CS	1990	← S
s4	Ann	Biology	2001	← S

  then S is assigned, one at the time, to these four tuples.
- The order in which S is assigned to these tuples is not pre-determined since the Student relation is a set!

Semantics (continued)

- For each tuple assigned to the variable S, the (boolean) condition in the WHERE clause is checked.
- If the condition is true for that tuple, then the attributes of that tuple specified in the SELECT clause will be output.
- If the condition is false for that tuple, no attributes of that tuple are output.

SQL has bag semantics

- Consider the query “Find the majors of students.”
- In SQL:
```
SELECT S.Major
FROM   Student S
```
- Semantics: for each tuple S in Student
  output (S.Major)
- Output on our sample database:

Major
CS
Math
CS
Biology

- Observe that ‘CS’ appears twice: the result of this query is a bag (multiset)

Enforcing set semantics

- To coerce the result of the query into a set, use the DISTINCT clause:

```
SELECT DISTINCT S.Major
FROM   Student S
```

- The result is the set

Major
CS
Math
Biology

Time-complexity of DISTINCT

- The time-complexity of the coercion from the bag into the set is at least linear in the size of the bag.
- If hashing is used, it can be done during the output generation.
- One can also use sorting as a technique to accomplish the coercion. In this case, the time-complexity is $O(n \log n)$, where n is the number of tuples in the bag.

EXPLAIN inPostgreSQL

- In PostgreSQL, it is possible to see how the system implements the coercion.
- To see this, issue the EXPLAIN command:

```
EXPLAIN SELECT DISTINCT S.Major
                FROM   Student S;
```

- The system will generate an access plan that indicates whether hashing or sorting is used.
- In general, EXPLAIN is very useful to understand how an SQL query is processed by the system.

EXPLAIN ANALYZE inPostgreSQL

- In PostgreSQL, it is also possible to see prediction and actual time and space complexity associated with a query.
- To see this, issue the EXPLAIN ANALYZE command:

```
EXPLAIN ANALYZE SELECT DISTINCT S.Major
                FROM   Student S;
```

Renaming attributes

- In SQL, it is possible to give names to attributes that appear in a SELECT clause using the AS clause:

```
SELECT S.Sid AS Identifier, S.Sname AS Name
FROM   Student S
WHERE S.Major = 'CS';
```

- This query returns the relation

Identifier	Name
s1	John
s3	Eric

Ordering the output

- In SQL, it is possible the sort the output on the basis of order relation that exists on attribute domains
- This can be done with the ORDER BY clause

```
SELECT S.sid, S.sname
FROM   Student S
WHERE S.Major = 'CS'
ORDER BY sname;
```

sid	sname
s3	Eric
s1	John

Ordering the output (by column)

- The following queries are equivalent

```
SELECT S.sid, S.sname
FROM   Student S
WHERE S.Major = 'CS'
ORDER BY sname;
```

```
SELECT S.sid, S.sname
FROM   Student S
WHERE S.Major = 'CS'
ORDER BY 2;
```

Randomly ordering the output

- In SQL, it is possible to randomly order the output of a query
- This can be done with the ORDER BY RANDOM() clause
- The following executions of the same queries may not give the some result

```
SELECT *
FROM   Student S
ORDER BY RANDOM();
```

```
SELECT *
FROM   Student S
ORDER BY RANDOM();
```

Queries involving multiple relations

- Consider the query "Find the names of students along with the cnos of the courses in which they received a B grade":

```
SELECT S.Sname, E.Cno
FROM   Student S, Enroll E
WHERE S.Sid = E.Sid AND E.Grade = 'B';
```

- We have two tuple tuples variables:
  - S ranges over the tuples in Student;
  - E ranges over the tuples in Enroll.
- In the WHERE clause, conditions are checked on attribute values of tuples assigned to these variables.

Queries on multiple relations (c'ed)

- SQL query:

```
SELECT S.Sname, E.Cno
FROM   Student S, Enroll E
WHERE S.Sid = E.Sid AND E.Grade = 'B';
```

- Semantics:

```
for each tuple S in Student
    for each tuple E in Enroll
        if S.Sid = E.Sid and E.Grade = 'B'
            then output (S.Sname, E.Cno)
```

- Notice the double-nested for-loop!

Queries on multiple relations (c'ed)

- SQL query:

```
SELECT S.Sname, C.Cno
FROM   Student S, Enroll E
WHERE S.Sid = E.Sid AND E.Grade = 'B'
```

- The condition S.Sid = E.Sid is called a join condition: it verifies a condition between two variables
- The condition E.Grade = 'B' is called a constant comparison condition: it verifies a condition on a single variable.

Time-complexity considerations

- If Student has m tuples and Enroll has n tuples, then the if statement

  if S.Sid = E.Sid and E.Grade = 'B'

  is evaluated $m \times n$ times.
- This essentially quadratic time complexity compromises the scalability of this simple query for large values of m and n.

Sub-queries in the FROM clause

- The previous query can also be formulated as follows:

```
SELECT S.Sname, C.Cno
FROM   Student S, (SELECT E.Sid, E.Cno
                   FROM   Enroll E
                   WHERE E.Grade = 'B') C
WHERE S.Sid = C.Sid;
```

- There are 2 global variables (S, C) and 1 local variable (E):
  - S ranges over the tuples in Student;
  - C ranges over the tuples in the relation determined by the sub-query (the part between the brackets);
  - E ranges over the tuples in Enroll.

Subqueries (continued)

- We conclude the following:
  - In SQL, it is permissible to have tuple variables that range over relations, but also tuple variables that range over the tuples of a relation computed by a subquery.
  - This works, because an SQL query always returns a relation.
  - Subqueries are allowed to occur in the FROM clause of a query or another subquery.

Performance improvement

- Query structure:

```
SELECT S.Sname, C.Cno
FROM   Student S, (subquery) C
WHERE S.Sid = C.Sid
```

- Semantics:

```
for each tuple S in Student
    for each tuple C in output (subquery)
        if S.Sid = C.Sid
            then output (S.Sname, C.Cno)
```

- If the subquery yields s tuples, then the if-then statement will be executed $m \times s$ times instead of $m \times n$.
- This can be a considerable gain if s is small compared to m.

- Of course we must also add the time to compute the relation generated by the subquery. This can be done using a linear scan requiring n operations.
- So the total time is $n + m \times s$ instead of $m \times n$.

Multiple tuple variables ranging over the same relation

- Consider the query "Find the name of each student enrolled in at least two courses."
- Since a tuple variable ranging over a relation is only assigned to a single tuple of that relation, it can never hold information about two or more tuples.
- However, if we have two or more tuple variables ranging over the same relation, then this becomes possible.

Multiple tuple variables ranging over the same relation: example

- We want to express the query "Find the sid and name of each student enrolled in at least two courses."
- In SQL, this query can be formulated using two variables ranging over the Enroll relation:

```
SELECT DISTINCT S.Sid, S.Sname
FROM   Student S, Enroll E1, Enroll E2
WHERE S.Sid = E1.Sid AND S.Sid = E2.Sid AND E1.Cno <> E2.Cno;
```

- <> is the symbol for not equal, i.e., ≠.
- Why do we need DISTINCT?

Multiple tuple variables ranging over the same relation: example (continued)

- Semantics:

```
for each tuple S in Student
    for each tuple E1 in Enroll
        for each tuple E2 in Enroll
            if S.Sid = E1.Sid and S.Sid = E2.Sid and E1.Cno ≠ E2.Cno
                then output (S.Sid,S.Sname)
```

- Performance: $m \times n^2$.
- Later, we shall see that there are far more efficient ways to write and solve this query.

Queries using set operations

- We now consider queries involving the set operations union, intersection, and difference:

Operation	Math notation	SQL notation
Union	$\cup$	UNION
Intersection	$\cap$	INTERSECT
Difference	$-$	EXCEPT

Set operations: examples

R

A	B
1	1
1	2
3	2
3	1

S

A	B
1	3
3	1
3	2
2	2
2	3

$R \cup S$

A	B
1	1
1	2
3	2
3	1
1	3
2	2
2	3

$R \cap S$

A	B
3	2
3	1

$R - S$

A	B
1	1
1	2

$S - R$

A	B
1	3
2	2
2	3

Notice that $R \cup S = S \cup R$ and $R \cap S = S \cap R$.
However, in general, $R - S \neq S - R$.

Union

- Consider the query “Find the sids and names of all students who major in CS or who major in Math.”
- In SQL,

```
(SELECT S.Sid, S.Sname
 FROM   Student S
 WHERE S.Major = 'CS')
UNION
(SELECT S.Sid, S.Sname
 FROM   Student S
 WHERE S.Major = 'Math');
```

Union (alternative)

- Consider again “Find the sids and names of all students who major in CS or who major in Math.”
- In SQL,

```
SELECT S.Sid, S.Sname
FROM   Student S
WHERE S.Major = ‘CS’ OR S.Major = ‘Math’;
```

Intersection

- Consider the query "Find the sids of all students who are enrolled in course c1 and course c2."
- In SQL,

```
(SELECT E.Sid
 FROM   Enroll E
 WHERE E.Cno = 'c1')
INTERSECT
(SELECT E.Sid
 FROM   Enroll E
 WHERE E.Cno = 'c2');
```

Wrong alternative for intersection

- Consider again "Find the sids of all students who major in c1 and c2."
- What about the following SQL query?

```
SELECT E.Sid
FROM   Enroll E
WHERE E.Cno = 'c1' AND E.Cno = 'c2';
```

- The result of this query is the empty set since, for no tuple from Enroll assigned to the variable E, E.Cno can be equal to both c1 and c2 at the same time!

Difference

- Consider the query "Find the sids of all students who are enrolled in no courses."
- In SQL,

```
(SELECT S.Sid
 FROM   Student S)
EXCEPT
(SELECT E.Sid
 FROM   Enroll E);
```

- The first subquery yields the set of all student sids.
- The second subquery yields the set of the sids of all students enrolled in some course.
- The difference between both sets yields the result.

Performance considerations

- UNION, INTERSECTION, and DIFFERENCE can be implemented using hashing or using sorting.
- Therefore, these operations can be implemented in linear time $O(m+n)$ or in $O(n\log n + m\log m)$, where m and n are the respective sizes of the relations involved in these operations.

Semantics of set operations in SQL

- In SQL, the result of the UNION, INTERSECT, and EXCEPT operations are sets, even if the inputs to these operations are bags.
- So, the result of

```
(SELECT E.Sid
 FROM   Enroll E)
UNION
(SELECT E.Sid
 FROM   Enroll E)
```

is the same as the result of

```
SELECT  DISTINCT E.Sid
FROM   Enroll E
```

- To retain bag semantics, use UNION ALL, INTERSECT ALL, and EXCEPT ALL.