SQL: Data Manipulation Language

csc343, Introduction to Databases
Diane Horton
Winter 2016

UNIVERSITY OF TORONTO

Introduction

- So far, we have defined database schemas and queries mathematically.
- SQL is a formal language for doing so with a DBMS.
- "Structured Query Language", but it's for more than writing queries.
- Two sub-parts:
    - DDL (Data Definition Language), for defining schemas.
    - DML (Data Manipulation Language), for writing queries and modifying the database.

PostgreSQL

- We'll be working in PostgreSQL, an open-source relational DBMS.
- Learn your way around the documentation; it will be very helpful.
- Standards?
  - There are several, the most recent being SQL:2008.
  - The standards are not freely available. Must purchase from the International Standards Organization (ISO).
  - PostgreSQL supports most of it SQL:2008.
  - DBMSs vary in the details around the edges, making portability difficult.

A high-level language

- SQL is a very high-level language.
  - Say "what" rather than "how."
- You write queries without manipulating data. Contrast languages like Java or C++.
- Provides physical "data independence:"
  - Details of how the data is stored can change with no impact on your queries.
- You can focus on readability.
  - But because the DMBS optimizes your query, you get efficiency.

Heads up: SELECT vs σ

- In SQL,
  - "SELECT" is for choosing columns, *i.e.*, π.
  - Example:

```
SELECT surName
FROM Student
WHERE campus = 'StG';
```

- In relational algebra,
  - "select" means choosing rows, *i.e.*, σ.

Basic queries

[Slides 8-16 are essentially covered by Prep4]

UNIVERSITY OF TORONTO

Meaning of a query with one relation

```
SELECT name
FROM Course
WHERE dept = 'CSC';
```

$\pi_{name}\ (\sigma_{dept="csc"}\ (Course))$

… and with multiple relations

```
SELECT name
FROM Course, Offering, Took
WHERE dept = 'CSC';
```

$$\pi_{name}\,(\sigma_{dept=\text{"csc"}}\,(Course \times Offering \times Took))$$

Temporarily renaming a table

- You can rename tables (just for the duration of the statement):

```
SELECT e.name, d.name
FROM employee e, department d
WHERE d.name = 'marketing'
AND e.name = 'Horton';
```

- Can be convenient vs the longer full names:

```
SELECT employee.name, department.name
FROM employee, department
WHERE department.name = 'marketing'
AND employee.name = 'Horton';
```

- This is like ρ in relational algebra.

Self-joins

- As we know, renaming is *required* for self-joins.
- Example:

```
SELECT e1.name, e2.name
FROM employee e1, employee e2
WHERE e1.salary < e2.salary;
```

* In SELECT clauses

- A * in the SELECT clause means "all attributes of this relation."
- Example:

```
SELECT *
FROM Course
WHERE dept = 'CSC';
```

Renaming attributes

- Use `AS «new name»` to rename an attribute in the result.
- Example:

```
SELECT name AS title, dept
FROM Course
WHERE breadth;
```

Complex Conditions in a WHERE

- We can build boolean expressions with operators that produce boolean results.
  - comparison operators: `=, <>, <, >, <=, >=`
  - and many other operators:
    see section 6.1.2 of the text and chapter 9 of the postgreSQL documentation.
- Note that "not equals" is unusual: `<>`
- We can combine boolean expressions with:
  - Boolean operators: `AND, OR, NOT`.

Example: Compound condition

- Find 3rd- and 4th-year CSC courses:

```
SELECT *
FROM Offering
WHERE dept = 'CSC' AND cnum >= 300;
```

ORDER BY

- To put the tuples in order, add this as the final clause:
  `ORDER BY «attribute list» [DESC]`
- The default is ascending order; DESC overrides it to force descending order.
- The attribute list can include expressions: e.g., `ORDER BY sales+rentals`
- The ordering is the last thing done before the SELECT, so all attributes are still available.

Case-sensitivity and whitespace

- Example query:

```
SELECT surName
FROM Student
WHERE campus = 'StG';
```

- Keywords, like `SELECT`, are not case-sensitive.
  - One convention is to use uppercase for keywords.
- Identifiers, like `Student` are not case-sensitive either.
  - One convention is to use lowercase for attributes, and a leading capital letter followed by lowercase for relations.
- Literal strings, like `'StG'`, are case-sensitive, and require single quotes.
- Whitespace (other than inside quotes) is ignored.

Expressions in SELECT clauses

- Instead of a simple attribute name, you can use an expression in a SELECT clause.
- Operands: attributes, constants
  Operators: arithmetic ops, string ops
- Examples:

```
SELECT sid, grade-10 AS adjusted
FROM Took;

SELECT dept||cnum
FROM course;
```

Expressions that are a constant

- Sometimes it makes sense for the whole expression to be a constant (something that doesn't involve any attributes!).
- Example:

```
SELECT sID,
   'satisfies' AS breadthRequirement
FROM Course
WHERE breadth;
```

Pattern operators

- Two ways to compare a string to a pattern by:
  - *«attribute»* `LIKE` *«pattern»*
  - *«attribute»* `NOT LIKE` *«pattern»*
- Pattern is a quoted string
  - `%` means: any string
  - `_` means: any single character
- Example:

```
SELECT *
FROM Course
WHERE name LIKE '%Comp%';
```

Aggregation

Computing on a column

- We often want to compute something across the values in a column.
- `SUM`, `AVG`, `COUNT`, `MIN`, and `MAX` can be applied to a column in a SELECT clause.
- Also, `COUNT(*)` counts the number of tuples.
- We call this aggregation.
- Note: To stop duplicates from contributing to the aggregation, use `DISTINCT` inside the brackets.
- Example: aggregation.txt

Grouping

- Example: group-by.txt
- If we follow a SELECT-FROM-WHERE expression with GROUP BY <attributes>
  - The tuples are grouped according to the values of those attributes, and
  - any aggregation gives us a single value per group.

Restrictions on aggregation

- If any aggregation is used, then each element of the SELECT list must be either:
  - aggregated, or
  - an attribute on the GROUP BY list.
- Otherwise, it doesn't even make sense to include the attribute.

HAVING Clauses

- Example: having.txt
- WHERE let's you decide which tuples to keep.
- Similarly, you can decide which *groups* to keep.
- Syntax:

```
. . .
GROUP BY «attributes»
HAVING «condition»
```

- Semantics:
  Only groups satisfying the condition are kept.

Requirements on HAVING clauses

- Outside subqueries, HAVING may refer to attributes only if they are either:
  - aggregated, or
  - an attribute on the GROUP BY list.
- (The same requirement as for SELECT clauses with aggregation).

Set operations

Tables can have duplicates in SQL

- A table can have duplicate tuples, unless this would violate an integrity constraint.
- And SELECT-FROM-WHERE statements leave duplicates in unless you say not to.
- Why?
  - Getting rid of duplicates is expensive!
  - We may want the duplicates because they tell us how many times something occurred.

Bags

- SQL treats tables as "bags" (or "multisets") rather than sets.
- Bags are just like sets, but duplicates are allowed.
- {6, 2, 7, 1, 9} is a set (and a bag)
  {6, 2, 2, 7, 1, 9} is not a set, but is a bag.
- Like with sets, order doesn't matter.
  {6, 2, 7, 1, 9} = {1, 2, 6, 7, 9}
- Example: Tables with duplicates

Union, Intersection, and Difference

- These are expressed as:

  (*«subquery»*) UNION (*«subquery»*)

  (*«subquery»*) INTERSECT (*«subquery»*)

  (*«subquery»*) EXCEPT (*«subquery»*)

- The brackets are mandatory.
- The operands must be queries; you can't simply use a relation name.

Example

```
(SELECT sid
 FROM Took
 WHERE grade > 95)
           UNION
(SELECT sid
 FROM Took
 WHERE grade < 50);
```

Operations $\cup$, $\cap$, and $-$ with Bags

- For $\cup$, $\cap$, and $-$ the number of occurrences of a tuple in the result requires some thought.
- (But it makes total sense.)

Operations $\cup$, $\cap$, and $-$ with Bags

- Suppose tuple t occurs
  - m times in relation R, and
  - n times in relation S.

Operation	Number of occurrences of t in result
$R \cap S$	min(m, n)
$R \cup S$	m + n
R - S	max(m-n, 0)

UNIVERSITY OF TORONTO

Bag vs Set Semantics: which is used

- We saw that a SELECT-FROM-WHERE statement uses bag semantics by default.
  - Duplicates are kept in the result.
- The set operations use set semantics by default.
  - Duplicates are *eliminated* from the result.

Motivation: Efficiency

- When doing projection, it is easier not to eliminate duplicates.
  - Just work one tuple at a time.
- For intersection or difference, it is most efficient to sort the relations first.
  - At that point you may as well eliminate the duplicates anyway.

Controlling Duplicate Elimination

- We can force the result of a SFW query to be a set by using `SELECT DISTINCT ...`
- We can force the result of a set operation to be a bag by using `ALL`, e.g.,

```
(SELECT sid
 FROM Took
 WHERE grade > 95)
          UNION ALL
(SELECT sid
 FROM Took
 WHERE grade < 50);
```

- Examples: controlling-dups.txt, except-all.txt

Views

The idea

- A view is a relation defined in terms of stored tables (called base tables) and other views.
- Access a view like any base table.
- Two kinds of view:
  - Virtual: no tuples are stored; view is just a query for constructing the relation when needed.
  - Materialized: actually constructed and stored. Expensive to maintain!
- We'll use only virtual views.
  - PostgreSQL did not support materialized views until version 9.3 (which we are not running).

Example: defining a virtual view

- A view for students who earned an 80 or higher in a CSC course.

```
CREATE VIEW topresults as
SELECT firstname, surname, cnum
FROM Student, Took, Offering
WHERE
   Student.sid = Took.sid AND
   Took.oid = Offering.oid AND
   grade >= 80 AND dept = 'CSC';
```

Uses for views

- Break down a large query.
- Provide another way of looking at the same data, e.g., for one category of user.

Outer Joins

The joins you know from RA

These can go in a FROM clause, or can be stand-alone queries:

Expression	**Meaning**
`R, S`	$R \times S$
`R cross join S`	
`R natural join S`	$R \bowtie S$
`R join S on Condition`	$R \bowtie_{condition} S$

In practise natural join is dangerous

- A working query can be broken by adding a column to a schema.
  - Example:

```
SELECT sID, instructor
FROM Student NATURAL JOIN Took
                NATURAL JOIN Offering;
```

  - What if we add a column called `campus` to `Offering`?
- Also, having implicit comparisons impairs readability.
- Best practise: Don't use natural join.

Dangling tuples

- With joins that require some attributes to match, tuples lacking a match are left out of the results.
- We say that they are "dangling".
- An outer join preserves dangling tuples by padding them with `NULL` in the other relation.
- A join that doesn't pad with `NULL` is called an inner join.

Three kinds of outer join

- `LEFT OUTER JOIN`
  - Preserves dangling tuples from the relation on the LHS by padding with nulls on the RHS.
- `RIGHT OUTER JOIN`
  - The reverse.
- `FULL OUTER JOIN`
  - Does both.

Example: joining R and S various ways

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL JOIN S

A	B	C
1	2	3

Example

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL FULL JOIN S

A	B	C
1	2	3
4	5	NULL
NULL	6	7

Example

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL LEFT JOIN S

A	B	C
1	2	3
4	5	NULL

Example

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL RIGHT JOIN S

A	B	C
1	2	3
NULL	6	7

Summary of join expressions

Cartesian product

`A CROSS JOIN B` same as `A,B`

Theta-join

`A JOIN B ON C`

✓`A {LEFT|RIGHT|FULL} JOIN B ON C`

Natural join

`A NATURAL JOIN B`

✓`A NATURAL {LEFT|RIGHT|FULL} JOIN B ON C`

✓ indicates that tuples are padded when needed.

Keywords INNER and OUTER

- There are keywords `INNER` and `OUTER`, but you never need to use them.
- Your intentions are clear anyway:
  - You get an outer join iff you use the keywords `LEFT`, `RIGHT`, or `FULL`.
  - If you don't use the keywords `LEFT`, `RIGHT`, or `FULL` you get an inner join.

Impact of having null values

Missing Information

- Two common scenarios:
  - Missing value.
    E.g., we know a student has some email address, but we don't know what it is.
  - Inapplicable attribute.
    E.g., the value of attribute spouse for an unmarried person.

Representing missing information

- One possibility: use a special value as a placeholder. E.g.,
  - If age unknown, use 0.
  - If StNum unknown, use 999999999.
- Implications?
- Better solution: use a value not in any domain. We call this a null value.
- Tuples in SQL relations can have `NULL` as a value for one or more components.

Checking for null values

- You can compare an attribute value to `NULL` with
  - `IS NULL`
  - `IS NOT NULL`
- Example:

```
SELECT *
FROM Course
WHERE breadth IS NULL;
```

In SQL we have 3 truth-values

- Because of `NULL`, we need three truth-values:
  - If one or both operands to a comparison is `NULL`, the comparison always evaluates to `UNKNOWN`.
  - Otherwise, comparisons evaluate to `TRUE` or `FALSE`.

Combining truth values

- We need to know how the three truth-values combine with `AND`, `OR` and `NOT`.
- Can think of it in terms of the truth table.
- Or can think in terms of numbers:
  - `TRUE` = 1, `FALSE` = 0, `UNKNOWN` = 0.5
  - `AND` is min, `OR` is max,
  - `NOT` x is (1-x), i.e., it "flips" the value

The three-valued truth table

A	B	A and B	A or B
T	T	T	T
TF or FT		F	T
F	F	F	F
TU or UT		U	T
FU or UF		F	U
U	U	U	U

UNIVERSITY OF TORONTO

Thinking of the truth-values as numbers

A	B	as nums	A and B	min	A or B	max
T	T	1, 1	T	1	T	1
TF or FT		1, 0	F	0	T	1
F	F	0, 0	F	0	F	0
TU or UT		1, 0.5	U	0.5	T	1
FU or UF		0, 0.5	F	0	U	0.5
U	U	0.5, 0.5	U	0.5	U	0.5

Surprises from 3-valued logic

- Some laws you are used to still hold in three-valued logic. For example,
  - `AND` is commutative.
- But others don't. For example,
  - The law of the excluded middle breaks: `(p or (NOT p))` might not be `TRUE`!
  - `(0*x)` might not be `0`.

Impact of null values on WHERE

- A tuple is in a query result iff the WHERE clause is `TRUE`.
- `UNKNOWN` is not good enough.
- "WHERE is picky."
- Example: where-null

Impact of null values on DISTINCT

- Example: select-distinct-null
- This behaviour may vary across DBMSs.

Impact of null values on aggregation

- Summary: Aggregation ignores `NULL`.
  - `NULL` never contributes to a sum, average, or count, and
  - can never be the minimum or maximum of a column (unless *every* value is `NULL`).
- If there are no *non*-`NULL` values in a column, then the result of the aggregation is `NULL`.
  - Exception: `COUNT` of an empty set is 0.

Aggregation ignores nulls

	some nulls in A	All nulls in A
`min(A)`	ignore the nulls	null
`max(A)`		
`sum(A)`		
`avg(A)`		
`count(A)`		0
`count(*)`	all tuples count	

Example: aggregation-nulls