SQL: Data Manipulation Language

csc343, Introduction to Databases
Bogdan Simion
Fall 2015
(based on slides from Diane Horton)
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 DBMS optimizes your query, you get efficiency.
Write efficient queries

• Don’t just join everything and hope for the best!
• Big datasets => Big problem, regardless of internal DBMS optimizations!
• We won’t be marking such “just join everything” queries for assignments or future weeks’ prep work!
Heads up: SELECT vs \( \sigma \)

• In SQL,
  • SELECT is for choosing columns, \( i.e. , \pi \).
  • Example:
    ```
    select surName
    from Student
    where campus = 'StG';
    ```

• In relational algebra,
  • “select” means choosing rows, \( i.e. , \sigma \).
Basic queries
[Slides 8-16 are essentially covered by Prep4]
Meaning of a query with one relation

\[
\pi_{\text{name}} (\sigma_{\text{dept} = \text{CSC}} (\text{Course}))
\]
Meaning of a query with one relation

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

$(\sigma_{\text{dept} = \text{'CSC'}}(\text{Course}))$

$(\pi_{\text{name}}(\sigma_{\text{dept} = \text{'CSC'}}(\text{Course})))$
... and with multiple relations

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

$$\Pi_{name} (\sigma_{dept='csc'} (Course \times Offering \times Took))$$
Temporarily renaming a table

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

```sql
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:

```sql
select employee.name, department.name
from employee, department
where department.name = 'marketing'
and employee.name = 'Horton';
```

• **This is like $\rho$** in relational algebra.
Self-joins

• As we know, renaming is required for self-joins.

• Example:
  ```sql
  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:

```sql
SELECT *
FROM Course
WHERE dept = 'CSC';
```
Renaming attributes

• Use `AS «new name»` to rename an attribute in the result.

• Example:
  ```sql
  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.

• We can combine boolean expressions with:
  • Boolean operators: AND, OR, NOT.
Example: Compound condition

• Find 3rd- and 4th-year CSC courses:

```sql
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:
  ```sql
  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:**

  ```sql
  SELECT sid, grade-10 as adjusted
  FROM Took;
  ```

  ```sql
  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:

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

<table>
<thead>
<tr>
<th>name</th>
<th>breadthRequirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Databases</td>
<td>satisfies</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>satisfies</td>
</tr>
<tr>
<td>Intro to Slaying Dragons</td>
<td>satisfies</td>
</tr>
</tbody>
</table>
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

• **Examples:** ...  
  • please focus on understanding the queries, no need to write them down  
  • will be posted on course website anyway

• 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 is applied only within each 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: ...
  - (again, pay attention, these will be posted)
- 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

• 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).
## Order of execution of a SQL query

<table>
<thead>
<tr>
<th>Query order</th>
<th>Execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>FROM</td>
</tr>
<tr>
<td>FROM</td>
<td>WHERE</td>
</tr>
<tr>
<td>WHERE</td>
<td>GROUP BY</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>HAVING</td>
</tr>
<tr>
<td>HAVING</td>
<td>SELECT</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>ORDER BY</td>
</tr>
</tbody>
</table>
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 \(\cup, \cap, \text{ and } -\) with Bags

- For \(\cup, \cap, \text{ and } -\) the number of occurrences of a tuple in the result requires some thought.
- (But it makes total sense.) . . .
- **Exercises:**
  1. \(\{1, 1, 1, 3, 7, 7, 8\} \cup \{1, 5, 7, 7, 8, 8\}\)
  2. \(\{1, 1, 1, 3, 7, 7, 8\} \cap \{1, 5, 7, 7, 8, 8\}\)
  3. \(\{1, 1, 1, 3, 7, 7, 8\} - \{1, 5, 7, 7, 8, 8\}\)
1. \( \{1, 1, 1, 3, 7, 7, 8\} \cup \{1, 5, 7, 7, 8, 8\} \)
   \[= \{1, 1, 1, 3, 7, 7, 8, 1, 5, 7, 7, 8, 8\} \]
   \[= \{1, 1, 1, 1, 3, 5, 7, 7, 7, 7, 8, 8, 8\} \]

2. \( \{1, 1, 1, 3, 7, 7, 8\} \cap \{1, 5, 7, 7, 8, 8\} \)
   \[= \{1, 7, 7, 8\} \]

3. \( \{1, 1, 1, 3, 7, 7, 8\} - \{1, 5, 7, 7, 8, 8\} \)
   \[= \{1, 1, 3\} \]
Operations $\cup$, $\cap$, and $-$ with Bags

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of occurrences of $t$ in result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R \cap S$</td>
<td>$\min(m, n)$</td>
</tr>
<tr>
<td>$R \cup S$</td>
<td>$m + n$</td>
</tr>
<tr>
<td>$R - S$</td>
<td>$\max(m-n, 0)$</td>
</tr>
</tbody>
</table>
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 S-F-W 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.,
  
  ```sql
  (SELECT sid
   FROM Took
   WHERE grade > 95)
   UNION ALL
  (SELECT sid
   FROM Took
   WHERE grade < 50);
  ```

- Examples: ...
Controlling Duplicate Elimination

• We can force the result of a S-F-W 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);
  ```

• More examples: ...
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.

```sql
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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>R, S</td>
<td>R × S</td>
</tr>
<tr>
<td>R cross join S</td>
<td>R × S</td>
</tr>
<tr>
<td>R natural join S</td>
<td>R ⊙ S</td>
</tr>
<tr>
<td>R join S on Condition</td>
<td>$R \bowtie_{\text{condition}} S$</td>
</tr>
</tbody>
</table>
In practice natural join is dangerous

• A working query can be broken by adding a column to a schema.
  • Example:
    ```sql
    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 practice: 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.

• Join syntax summarized on next slides ...
Example: join R and S various ways

R

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

S

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

R NATURAL JOIN S

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

R NATURAL FULL JOIN S

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
### Example

The natural left join of tables R and S:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Table R:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table S:

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Example

\[
R \quad \left|\begin{array}{cc}
A & B \\
1 & 2 \\
4 & 5 \\
\end{array}\right| \quad S \quad \left|\begin{array}{cc}
B & C \\
2 & 3 \\
6 & 7 \\
\end{array}\right|
\]

\[
R \text{ NATURAL RIGHT JOIN } S
\]

\[
\left|\begin{array}{ccc}
A & B & C \\
1 & 2 & 3 \\
\text{NULL} & 6 & 7 \\
\end{array}\right|
\]
Summary of join expressions

• Cartesian product
  A cross join B same as A, B

• Theta-join
  A join B on C no padding of tuples
  A {left|right|full} join B on C padding

• Natural join
  A natural join B no padding of tuples
  A natural {left|right|full} join B padding
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
  • E.g., a student who has not taken any courses yet, does not have a cgpa.
  • **Inapplicable attribute.**
    E.g., the value of attribute spouse for an unmarried person.
  • E.g., the value of attribute criminalRecord for Santa Claus.
Representing missing information

• One possibility: use a special value as a placeholder. E.g.,
  • If age unknown, use 0.
  • If StNum unknown, use 999999999.

• Pros and cons?

• 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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
<th>A or B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>TF or FT</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>TU or UT</td>
<td>U</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>FU or UF</td>
<td>F</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>
Thinking of truth-values as numbers

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>as nums</th>
<th>A and B</th>
<th>min</th>
<th>A or B</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>1, 1</td>
<td>T</td>
<td>1</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>TF or FT</td>
<td>1, 0</td>
<td>F</td>
<td>0</td>
<td>T</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>0, 0</td>
<td>F</td>
<td>0</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>TU or UT</td>
<td>1, 0.5</td>
<td>U</td>
<td>0.5</td>
<td>T</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FU or UF</td>
<td>0, 0.5</td>
<td>F</td>
<td>0</td>
<td>U</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>0.5, 0.5</td>
<td>U</td>
<td>0.5</td>
<td>U</td>
<td>0.5</td>
</tr>
</tbody>
</table>
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: $(p \text{ or } \neg p )$ might not be **TRUE**!
  • $(0 \times x)$ 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

• “Aggregation ignores \texttt{NULL}.”
• \texttt{NULL} never contributes to a sum, average, or count, and can never be the minimum or maximum of a column (unless every value is \texttt{NULL}).
• If there are no \texttt{non-NULL} values in a column, then the result of the aggregation is \texttt{NULL}.
  • Exception: \texttt{COUNT} of an empty set is 0.
## Aggregation ignores nulls

<table>
<thead>
<tr>
<th>Function</th>
<th>Some nulls in A</th>
<th>All nulls in A</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \min(A) )</td>
<td>ignore the nulls</td>
<td>null</td>
</tr>
<tr>
<td>( \max(A) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sum(A) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{avg}(A) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{count}(A) )</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( \text{count}(\ast) )</td>
<td>all tuples count</td>
<td></td>
</tr>
</tbody>
</table>

**Example:** aggregation-nulls
Impact of NULL values on joins

• NULL values are not considered equal, for the purpose of matching values in a join

• Example: ...
NULL values in set operations

• Are they considered equal or no?

• Example: ...