(More) SQL

Introduction to Databases

CompSci 316 Fall 2020
Announcements (Thu. Sept 10)

• HW3 + Gradiance 2 posted (ER diagram)
  • Due dates: Wed September 16 11:59 pm

• Attendance posted up to 09/03 on Sakai
  • First two weeks, everyone will get attendance
  • If your attendance is recorded incorrectly after that, let us know by filling out the regrade form
  • If you are watching/downloading lecture later and getting wrong attendance, take a screenshot
  • Two lowest scores to be dropped

• Discussion grades posted on Sakai

• All regrade requests must appear within one week
Recap: Basic SQL from Lecture 1-2

• Find addresses of all bars that ‘Dan’ frequents

SELECT B.address
FROM Bar B, Frequents F
WHERE B.name = F.bar
AND F.drinker = ‘Dan’

We discussed
• SELECT-FROM-WHERE
• DISTINCT
• ORDER BY
• Bag vs. Set semantics (why bag?)
• Semantic of SQL evaluation (?)

<table>
<thead>
<tr>
<th>Bar</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>108 Morris Street</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequents</th>
<th>drinker</th>
<th>bar</th>
<th>times_a_week</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Dan</td>
<td>The Edge</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Dan</td>
<td>Satisfaction</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
SQL set and bag operations

• **UNION, EXCEPT, INTERSECT**
  - Set semantics
    - Duplicates in input tables, if any, are first eliminated
    - Duplicates in result are also eliminated (for UNION)
  - Exactly like set $\cup$, $-$, and $\cap$ in relational algebra

• **UNION ALL, EXCEPT ALL, INTERSECT ALL**
  - Bag semantics
  - Think of each row as having an implicit count (the number of times it appears in the table)
  - Bag union: sum up the counts from two tables
  - Bag difference: proper-subtract the two counts
  - Bag intersection: take the minimum of the two counts
Examples of bag operations

Bag1

fruit
apple
apple
orange

Bag2

fruit
apple
orange
orange

(SELECT * FROM Bag1) UNION ALL (SELECT * FROM Bag2);

(SELECT * FROM Bag1) EXCEPT ALL (SELECT * FROM Bag2);

(SELECT * FROM Bag1) INTERSECT ALL (SELECT * FROM Bag2);
Examples of set versus bag operations

Poke (uid1, uid2, timestamp)

• (SELECT uid1 FROM Poke) EXCEPT (SELECT uid2 FROM Poke);
  • Users who poked others but never got poked by others

• (SELECT uid1 FROM Poke) EXCEPT ALL (SELECT uid2 FROM Poke);
  • Users who poked others more than others poke them
Next: how to “nest” SQL queries and write sub-queries?
Table subqueries

• Use query result as a table
  • In set and bag operations, FROM clauses, etc.
  • A way to “nest” queries

• Example: names of users who poked others more than others poked them

• SELECT DISTINCT name
  FROM User,
  ((SELECT uid1 AS uid FROM Poke)
   EXCEPT ALL
   (SELECT uid2 AS uid FROM Poke))
  AS T
WHERE User.uid = T.uid;

Poke (uid1, uid2, timestamp)
Announcements (Tue. Sept 15)

• HW3 + Gradiance 2 (ER diagram) due tomorrow
  • Wed September 16 11:59 pm

• MS1 due next Thursday 09/24
  • Check project_details file on sakai *very carefully* for the deliverables
  • Weekly project updates due from next week on private Piazza threads
  • Even if you are doing an open project, read deliverables from the fixed project
  • Use the remaining time in your discussion session after quiz to work on projects
    (or HW problem if your group prefers so)

• Current approximate standing in class and survey to be posted soon

• Another research tool now to help you learn SQL (IRex):
  https://ratest.cs.duke.edu/irex#/ 
  • “Alpha” version, still working on it, many queries are not supported (and does not show a warning)
  • We request you to give us consent to use your data in anonymized/aggregate form to improve the tool!
  • Please report bugs and give comments!
IN subqueries

• $x$ IN (subquery) checks if $x$ is in the result of subquery

• Example: users (all columns) at the same age as (some) Bart

Let’s first try without sub-queries

• SELECT *
  FROM User
  WHERE age IN (SELECT age
                 FROM User
                 WHERE name = 'Bart');

You can use NOT IN too
EXISTS subqueries

• **EXISTS** *(subquery)* checks if the result of *subquery* is non-empty

• Example: users at the same age as (some) Bart

  • SELECT *
    FROM User AS u
    WHERE **EXISTS** (SELECT * FROM User
    WHERE name = 'Bart'
    AND age = u.age);

• This happens to be a **correlated subquery**—a subquery that references tuple variables in surrounding queries

You can use **NOT EXISTS** too
Semantics of subqueries

• SELECT *
  FROM User AS u
  WHERE EXISTS (SELECT * FROM User
    WHERE name = 'Bart'
    AND age = u.age);

• For each row u in User (called “binding”)
  • Evaluate the subquery with the value of u.age
  • If the result of the subquery is not empty, output u.*

• The DBMS query optimizer may choose to process the query in an equivalent, but more efficient way (example?)

Remember SQL evaluation!

FROM-WHERE-SELECT

User(uid, name, age, pop)
“WITH” clause – very useful!

• You will find “WITH” clause very useful!

  WITH Temp1 AS
      (SELECT .... ..),
  Temp2 AS
      (SELECT .... ..)
  SELECT X, Y
  FROM TEMP1, TEMP2
  WHERE....

• Can simplify complex nested queries

  Example: users at the same age as (some) Bart

  WITH BartAge AS
      (SELECT age
       FROM User
       WHERE name = 'Bart')
  SELECT U.uid, U.name, U.age, U.pop
  FROM User U, BartAge B
  WHERE U.age = B.age

  WITH clause
  not really needed
  for this query!
Scalar subqueries

• A query that returns a single row can be used as a value in WHERE, SELECT, etc.

• Example: users at the same age as Bart
  • SELECT *
    FROM User
    WHERE age = (SELECT age
                  FROM User
                  WHERE name = 'Bart');

• Runtime error if subquery returns more than one row
  • Under what condition will this error never occur?

• What if the subquery returns no rows?
  • The answer is treated as a special value NULL, and the comparison with NULL will fail (later)
Scoping rule of subqueries

• To find out which table a column belongs to
  • Start with the immediately surrounding query
  • If not found, look in the one surrounding that; repeat if necessary

• Use `table_name.column_name` notation and AS (renaming) to avoid confusion
Another example

• SELECT * FROM User u
  WHERE EXISTS
    (SELECT * FROM Member m
     WHERE uid = u.uid
     AND EXISTS
       (SELECT * FROM Member
        WHERE uid = u.uid
        AND gid <> m.gid));

• What does this query return?
• Users who join at least two groups
Quantified subqueries

• A quantified subquery can be used syntactically as a value in a WHERE condition

• Universal quantification (for all):
  ... WHERE \( x \text{ op ALL}(\text{subquery}) \) ...
  • True iff for all \( t \) in the result of subquery, \( x \text{ op } t \)

• Existential quantification (exists):
  ... WHERE \( x \text{ op ANY}(\text{subquery}) \) ...
  • True iff there exists some \( t \) in subquery result such that \( x \text{ op } t \)

☞ Beware
  • In common parlance, “any” and “all” seem to be synonyms
  • In SQL, \text{ANY} really means “some”
Examples of quantified subqueries

• Which users are the most popular?

  • SELECT *
    FROM User
    WHERE pop >= ALL(SELECT pop FROM User);

  • SELECT *
    FROM User
    WHERE NOT
      (pop < ANY(SELECT pop FROM User));

☞ Use NOT to negate a condition
More ways to get the most popular

• Which users are the most popular?

• SELECT *
  FROM User AS u
  WHERE NOT EXISTS
    (SELECT * FROM User
     WHERE pop > u.pop);

• SELECT * FROM User
  WHERE uid NOT IN
    (SELECT u1.uid
     FROM User AS u1, User AS u2
     WHERE u1.pop < u2.pop);
Next: aggregates, group-by, having!
Aggregates

• Standard SQL aggregate functions: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`

• Example: number of users under 18, and their average popularity
  • `SELECT COUNT(*), AVG(pop)`
    FROM User
    WHERE age < 18;
  • `COUNT(*)` counts the number of rows
Aggregates with DISTINCT

• Example: How many users are in some group?

  • SELECT COUNT(DISTINCT uid)
    FROM Member;

is equivalent to:

  • SELECT COUNT(*)
    FROM (SELECT DISTINCT uid FROM Member);
Grouping

- SELECT ... FROM ... WHERE ...
  GROUP BY list_of_columns;

- Example: compute average popularity for each age group
  - SELECT age, AVG(pop)
    FROM User
    GROUP BY age;
Semantics of GROUP BY

SELECT ... FROM ... WHERE ... GROUP BY ...;

• Compute FROM ($\times$)
• Compute WHERE ($\sigma$)
• Compute GROUP BY: group rows according to the values of GROUP BY columns
• Compute SELECT for each group ($\pi$)
  • For aggregation functions with DISTINCT inputs, first eliminate duplicates within the group

Number of groups = number of rows in the final output
Example of computing GROUP BY

```
SELECT age, AVG(pop) FROM User GROUP BY age;
```

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Compute GROUP BY: group rows according to the values of GROUP BY columns

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Compute SELECT for each group

<table>
<thead>
<tr>
<th>age</th>
<th>avg_pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.55</td>
</tr>
<tr>
<td>8</td>
<td>0.50</td>
</tr>
</tbody>
</table>
Aggregates with no GROUP BY

• An aggregate query with no GROUP BY clause = all rows go into one group

SELECT AVG(pop) FROM User;

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Group all rows into one group

Aggregate over the whole group

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
<th>avg_pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
<td>0.525</td>
</tr>
</tbody>
</table>
Restriction on SELECT

• If a query uses aggregation/group by, then every column referenced in SELECT must be either
  • Aggregated, or
  • A GROUP BY column

Why?

☞ This restriction ensures that any SELECT expression produces only one value for each group

Examples on blackboard
Examples of invalid queries

- SELECT uid, age
  FROM User GROUP BY age;
  • Recall there is one output row per group
  • There can be multiple uid values per group

- SELECT uid, MAX(pop) FROM User;
  • Recall there is only one group for an aggregate query with no GROUP BY clause
  • There can be multiple uid values
  • Wishful thinking (that the output uid value is the one associated with the highest popularity) does NOT work

Which one is correct?
HAVING

• Used to filter groups based on the group properties (e.g., aggregate values, GROUP BY column values)

• SELECT ... FROM ... WHERE ... GROUP BY ...

HAVING condition:

• Compute FROM (\(\times\))
• Compute WHERE (\(\sigma\))
• Compute GROUP BY: group rows according to the values of GROUP BY columns
• Compute HAVING (another \(\sigma\) over the groups)
• Compute SELECT (\(\pi\)) for each group that passes HAVING
HAVING examples

• List the average popularity for each age group with more than a hundred users
  • SELECT age, AVG(pop)
    FROM User
    GROUP BY age
    HAVING COUNT(*) > 100;
  • Can be written using WHERE and table sub-queries

• Find average popularity for each age group over 10
  • SELECT age, AVG(pop)
    FROM User
    GROUP BY age
    HAVING age > 10;
  • Can be written using WHERE without table subqueries
Views

• A view is like a “virtual” table
  • Defined by a query, which describes how to compute the view contents on the fly
  • DBMS stores the view definition query instead of view contents
  • Can be used in queries just like a regular table
Creating and dropping views

• Example: members of Jessica’s Circle

  • CREATE VIEW JessicaCircle AS
    SELECT * FROM User
    WHERE uid IN (SELECT uid FROM Member
                 WHERE gid = 'jes');

  • Tables used in defining a view are called “base tables”
    • User and Member above

• To drop a view

  • DROP VIEW JessicaCircle;

Why use views?