SQL: Part I
Introduction to Databases
CompSci 316 Spring 2017

Announcements (Mon. Feb. 06)
• Homework #1 due today 11:59pm
• Homework #2 to be assigned today
  • Due Friday, 02/17
  • Problem 6 and extra credit problem due after midterm
  • Gradiance problems and problem 4 to be released after respective lectures
  • Get started on it asap! you will need to learn these topics for midterm
• If you are unable to form/join a project group by Wednesday’s class (02/08), send me an email
• Meet me after the class if you can’t submit using WebSubmit

SQL
• SQL: Structured Query Language
  • Pronounced “S-Q-L” or “sequel”
  • The standard query language supported by most DBMS
• A brief history
  • IBM System R
  • ANSI SQL89
  • ANSI SQL92 (SQL2)
  • ANSI SQL99 (SQL3)
  • ANSI SQL 2003 (added OLAP, XML, etc.)
  • ANSI SQL 2006 (added more XML)
  • ANSI SQL 2008, ...

Creating anddropping tables
• CREATE TABLE table_name
  (... column_name column_type, ...);
• DROP TABLE table_name;
• Examples
  create table User(uid integer, name varchar(30), age integer, pop float);
  create table Group(gid char(10), name varchar(100));
  create table Member(uid integer, gid char(10));
  drop table Member;
  drop table Group;
  drop table User;
  /* everything from -- to the end of line is ignored.
  -- SQL is insensitive to white space.
  -- SQL is insensitive to case (e.g., GROUP is equivalent to ...GROUP...)

Basic queries: SFW statement
• SELECT $A_1$, $A_2$, ..., $A_n$
  FROM $R_1$, $R_2$, ..., $R_m$
  WHERE condition;
• Also called an SPJ (select-project-join) query
• Corresponds to (but not really equivalent to) relational algebra query:
  $\pi_{A_1,A_2,...,A_n}(\sigma_{condition}(R_1 \times R_2 \times \ldots \times R_m))$

Example: reading a table
• SELECT * FROM User;
  • Single-table query, so no cross product here
  • WHERE clause is optional
  • * is a short hand for “all columns”
Example: selection and projection

• Name of users under 18
  • SELECT name FROM User WHERE age<18;
• When was Lisa born?
  • SELECT 2016-age FROM User
    WHERE name = 'Lisa';
• SELECT list can contain expressions
  • Can also use built-in functions such as SUBSTR, ABS, etc.
• String literals (case sensitive) are enclosed in single quotes (case sensitivity may vary for DBMSs)

Example: join

• ID's and names of groups with a user whose name contains “Simpson”
  • SELECT Group.gid, Group.name
    FROM User, Member, Group
    WHERE User.uid = Member.uid
    AND Member.gid = Group.gid
    AND User.name LIKE '%Simpson%';
• LIKE matches a string against a pattern
  • % matches any sequence of zero or more characters
• Okay to omit table_name in table_name.column_name if column_name is unique

Example: rename

• ID's of all pairs of users that belong to one group
  • Relational algebra query:
    \[ \pi_{3, JED, 3}.JED \times \rho_{3, M\epsilon\mbox{m}1, 3, JED} \]
  • SQL:
    SELECT m1.uid AS uid1, m2.uid AS uid2
    FROM Member AS m1, Member AS m2
    WHERE m1.gid = m2.gid
    AND m1.uid > m2.uid;
• AS keyword is completely optional

A more complicated example

• Names of all groups that Lisa and Ralph are both in

Tip: Write the FROM clause first, then WHERE, and then SELECT

Why SFW statements?

• Out of many possible ways of structuring SQL statements, why did the designers choose SELECT-FROM-WHERE?
  • A large number of queries can be written using only selection, projection, and cross product (or join)
  • Any query that uses only these operators can be written in a canonical form: \[ \pi_L (\sigma_{\varphi} (R_1 \times \cdots \times R_m)) \]
  • Example: \[ \pi_{r,s,t} (\sigma_{p,q} (R_{s,t} \times R_{r,s,t} \times R_{p,q} \times R_{p,q} \times R_{m})) \]
  • SELECT-FROM-WHERE captures this canonical form

Set versus bag semantics

• Set
  • No duplicates
  • Relational model and algebra use set semantics
• Bag
  • Duplicates allowed
  • Number of duplicates is significant
  • SQL uses bag semantics by default
Set versus bag example

<table>
<thead>
<tr>
<th>uid</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>dps</td>
</tr>
<tr>
<td>123</td>
<td>gov</td>
</tr>
<tr>
<td>857</td>
<td>abc</td>
</tr>
</tbody>
</table>

SELECT gid FROM Member;

A case for bag semantics

- Efficiency
  - Saves time of eliminating duplicates
- Which one is more useful?
  - \( \pi_{\text{age}} \text{User} \)
  - SELECT age FROM User;

Besides, SQL provides the option of set semantics with \text{DISTINCT} keyword

Forcing set semantics

- ID’s of all pairs of users that belong to one group
  - SELECT m1.uid AS uid1, m2.uid AS uid2 FROM Member AS m1, Member AS m2 WHERE m1.gid = m2.gid AND m1.uid > m2.uid;
  - Say Lisa and Ralph are in both the book club and the student government
  - SELECT \text{DISTINCT} m1.uid AS uid1, m2.uid AS uid2 ...
  - With \text{DISTINCT}, all duplicate (uid1, uid2) pairs are removed from the output

Semantics of SFW

- SELECT [\text{DISTINCT}] E_1, E_2, \ldots, E_n FROM R_1, R_2, \ldots, R_m WHERE condition;
- For each \( t_1 \) in \( R_1 \):
  - For each \( t_2 \) in \( R_2 \): ...
  - For each \( t_m \) in \( R_m \):
    - If condition is true over \( t_1, t_2, \ldots, t_m \):
      - Compute and output \( E_1, E_2, \ldots, E_n \) as a row
    - If \text{DISTINCT} is present
      - Eliminate duplicate rows in output
- \( t_1, t_2, \ldots, t_m \) are often called tuple variables

SQL set and bag operations

- \text{UNION, EXCEPT, INTERSECT}
  - Set semantics
    - Duplicates in input tables, if any, are first eliminated
    - Duplicates in result are also eliminated (for \text{UNION})
    - Exactly like set \( \cup, \cap \) in relational algebra
  - \text{UNION ALL, EXCEPT ALL, INTERSECT ALL}
    - Bag semantics
      - Think of each row as having an implicit \text{count} (the number of times it appears in the table)
      - Bag union: \text{sum} up the counts from two tables
      - Bag difference: \text{proper-subtract} the two counts
      - Bag intersection: take the \text{minimum} of the two counts

Examples of bag operations

\begin{tabular}{|l|l|}
\hline
Bag1 & Bag2 \\
\hline
apple & apple \\
apple & orange \\
orange & \\
\hline
\end{tabular}

(SELECT * FROM Bag1) \text{UNION ALL} (SELECT * FROM Bag2)
Examples of set versus bag operations

- `(SELECT uid1 FROM Poke) EXCEPT (SELECT uid2 FROM Poke);`
- `(SELECT uid1 FROM Poke) EXCEPT ALL (SELECT uid2 FROM Poke);`

SQL features covered so far

- SELECT-FROM-WHERE statements (select-project-join queries)
- Set and bag operations
- Next: how to nest SQL 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;

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 (some) Bart
  - `SELECT * FROM User WHERE age = (SELECT age FROM User WHERE name = 'Bart');`
- Runtime error if subquery returns more than one row
- What if the subquery returns no rows?
  - The answer is treated as a special value NULL, and the comparison with NULL will fail
  - (more on NULL later)

IN subqueries

- `x IN (subquery) checks if x is in the result of a subquery`
  
- Example: users at the same age as (some) Bart
  - `SELECT * FROM User WHERE age IN (SELECT age FROM User WHERE name = 'Bart');`
  - NOTE: set membership – the attribute being checked must be the same as that in the nested query result

EXISTS subqueries

- `EXISTS (subquery) checks if the result of a 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
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
  - 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

Another example

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

- Q. What does this query return?
- A. 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 $x$ op ALL(subquery) ...
    - True iff for all $t$ in the result of subquery, $x$ op $t$
  - Existential quantification (exists):
    ... WHERE $x$ op ANY(subquery) ...
    - True iff there exists some $t$ in subquery result such that $x$ op $t$

- Beware
  - In common parlance, “any” and “all” seem to be synonyms
  - In SQL, ANY really means “some”

Examples of quantified subqueries

- Which users are the most popular?

More ways to get the most popular

- Which users are the most popular?
SQL features covered so far

• SELECT-FROM-WHERE statements
• Set and bag operations
• Subqueries
  • Subqueries allow queries to be written in more declarative ways (recall the “most popular” query)
  • But in many cases they don’t add expressive power
    • Try translating other forms of subqueries into [NOT] EXISTS, which in turn can be translated into join (and difference)
    • Watch out for number of duplicates though

Next: aggregation and grouping

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);

• Returns one number
• Next: how to return aggregates for a set of tuples

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

1. Compute FROM (x) – cross product
2. Compute WHERE (σ) – apply to each row
3. Compute GROUP BY – group rows according to the values of GROUP BY columns
4. Compute SELECT for each group (π) – output specified columns

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;

Group rows according to the values of GROUP BY columns

γ = Extended relational algebra symbol for aggregation:
Aggregates with no GROUP BY

- An aggregate query with no \texttt{GROUP BY} clause = all rows go into one group
  
  \begin{verbatim}
  SELECT AVG(pop) FROM User;
  \end{verbatim}

Restriction on SELECT

- If a query uses aggregation/group by, then every column referenced in \texttt{SELECT} must be either
  - Aggregated, or
  - A \texttt{GROUP BY} column

  \[ \text{This restriction ensures that any \texttt{SELECT} expression produces only one value for each group} \]

Examples of invalid queries

- \texttt{SELECT uid, age FROM User GROUP BY age;}
  
- \texttt{SELECT age FROM User GROUP BY uid, age;}

- \texttt{SELECT uid, MAX(pop) FROM User;}

  \[ \text{Next: How to filter out some groups} \]

\textbf{HAVING}

- Used to filter groups based on the group properties (e.g., aggregate values, \texttt{GROUP BY} column values)

  \begin{verbatim}
  1. SELECT .. FROM .. WHERE .. GROUP BY ..
     HAVING condition;
  \end{verbatim}

  \begin{enumerate}
  \item Compute \texttt{FROM \times} – cross product
  \item Compute \texttt{WHERE (\sigma)} – apply to each row
  \item Compute \texttt{GROUP BY} – group rows according to the values of \texttt{GROUP BY} columns
  \item Compute \texttt{HAVING} – another \texttt{\sigma} but over the groups!
  \item Compute \texttt{SELECT (\pi)} – for each group that passes \texttt{HAVING}
  \end{enumerate}

\textbf{HAVING examples}

- List the average popularity for each age group with more than a hundred users

  \begin{verbatim}
  2. SELECT age, AVG(pop) FROM User
     GROUP BY age
     HAVING COUNT(*) > 100;
  \end{verbatim}
  
  - Can be written using \texttt{WHERE} and table subqueries

- Find average popularity for each age group over 10

  \begin{verbatim}
  3. SELECT age, AVG(pop) FROM User
     GROUP BY age
     HAVING age > 10;
  \end{verbatim}
  
  - Can be written using \texttt{WHERE} without table subqueries

SQL features covered so far

- \texttt{SELECT-FROM-WHERE} statements
- Set and bag operations
- Subqueries
- Aggregation and grouping

  \[ \text{More expressive power than relational algebra} \]

  \[ \text{Next: ordering output rows} \]
ORDER BY

• SELECT [DISTINCT] ...
  FROM ... WHERE ... GROUP BY ... HAVING ...
ORDER BY output_column [ASC|DESC], ...;
• ASC = ascending, DESC = descending
• Semantics: After SELECT list has been computed and optional duplicate elimination has been carried out, sort the output according to ORDER BY specification

ORDER BY example

• List all users, sort them by popularity (descending) and name (ascending)
  • SELECT uid, name, age, pop
  FROM User
  ORDER BY pop DESC, name;
  • ASC is the default option
  • Strictly speaking, only output columns can appear in ORDER BY clause (although some DBMS support more)
  • Can use sequence numbers instead of names to refer to output columns: ORDER BY 4 DESC, 2;

SQL features covered so far

• SELECT-FROM-WHERE statements
• Set and bag operations
• Subqueries
• Aggregation and grouping
• Ordering

❖ Next: NULL’s, outerjoins, data modification, constraints, ...