CS145 Lecture Notes #7
SQL Query & Modification

Introduction

SQL—Structured Query Language
- Pronounced “S-Q-L” or “sequel”
- The query language of every commercial RDBMS
Evolution of SQL standard: SQL89 → SQL92 (SQL2) → SQL3
Components of SQL:
- DDL: Data Definition Language
  - CREATE TABLE, DROP TABLE, etc.
- DML: Data Manipulation Language
  - Query: SELECT
  - Modification: INSERT, DELETE, UPDATE

Basic SELECT

$$\text{SELECT } A_1, A_2, ..., A_n \text{ FROM } R_1, R_2, ..., R_m \text{ WHERE condition;}$$
- Called a SPJ (select-project-join) query
- Equivalent (more or less) to relation algebra query:
  $$\pi_{A_1,...,A_n}(\sigma_{\text{condition}}(R_1 \times R_2 \times ... \times R_m))$$
  $$\leadsto$$ Returns an unnamed table with columns $$A_1, A_2, ..., A_n$$

Example schema:
- Student(SID, name, age, GPA)
- Take(SID, CID)
- Course(CID, title)
Example: names of students under 18

Example: SID’s and names of students taking Calculus
$$\leadsto$$ String literals are enclosed in single quotes
$$\leadsto$$ SQL is case insensitive; case only matters in quoted strings and strings stored in database
● Use \textit{SELECT} * to output all columns in the cross product
  \textbf{Example: } \texttt{SELECT * FROM Student;}

  \textit{Note that WHERE clause is optional}
  \textbf{Example: } \texttt{SELECT * FROM Course, Take WHERE ...;}

● Use \texttt{AS} in \textit{SELECT} clause to rename output columns
  \textbf{Example: } \texttt{SELECT name AS studName FROM Student ...;}

● Use \textit{tuple variables} in \textit{FROM} clause to rename input tables
  \textbf{Example: } \texttt{SID's of all pairs of classmates}

\textit{SQL2} permits an optional \texttt{AS} between the table and its tuple variable; Oracle does not

● \textit{SELECT} list may also contain expressions
  \textbf{Example: } when was Lisa born?

● Use \texttt{LIKE} in \textit{WHERE} clause for string matching
  \textbf{Example: } ID's of all students whose names start with the letter B

● Use \texttt{ORDER BY} clause to sort result rows
  \textbf{Example: } ID's of students over 18, sorted by GPA (descending) then name (ascending)

\textbf{Operational Semantics of SPJ Queries}

\begin{verbatim}
SELECT $E_1, ..., E_n$
FROM $R_1$ $t_1, ..., R_m$ $t_m$
WHERE \textit{condition};

For each $t_1$ in $R_1$: ...
For each $t_m$ in $R_m$:
  If \textit{condition} is true for $t_1, ..., t_m$,
  Compute and output $\langle E_1, ..., E_n \rangle$
\end{verbatim}

Jun Yang 2  CS145  Spring 1999
By default, SQL has \textit{bag semantics}, i.e., duplicate rows are retained

- Different from relational algebra, which has \textit{set semantics}
  
  Example: $\pi_{\text{SID}}(\text{Take}) \not\equiv \text{SELECT SID FROM Take}$;

- Use \texttt{DISTINCT} after \texttt{SELECT} to force set semantics
  
  Example: $\pi_{\text{SID}}(\text{Take}) \equiv \text{SELECT DISTINCT SID FROM Take}$;

- Why bag semantics?
  
  - Saves time of eliminating duplicates
  
  - Which one is more useful? \texttt{SELECT GPA FROM Student;} or \texttt{SELECT DISTINCT GPA FROM Student;}?

\textbf{UNION, EXCEPT, INTERSECT}

Example schema: add another table \texttt{ClubMember}($\text{club, SID}$)

Example: SID’s of students who are taking classes and/or involved in clubs

- \texttt{UNION, EXCEPT, INTERSECT} eliminate duplicates
  
  (set semantics)
  
  - Exactly like set $\cup$, $\setminus$, $\cap$

- \texttt{UNION ALL, EXCEPT ALL, INTERSECT ALL} retain duplicates
  
  (bag semantics)
  
  - Bag union: sum the times an element appears in the two bags
  
  - Bag difference: proper-subtract the times an element appears in the two bags
  
  - Bag intersection: take the minimum of the times an element appears in the two bags

$\leadsto$ Oracle calls difference \texttt{MINUS} instead of \texttt{EXCEPT}

Example: SID’s of students who are in clubs but not in any classes

Example: SID’s of students who are in more clubs than classes
Subqueries

Subqueries in FROM Clause

(Not covered in book)
Provides an easy way to “nest” queries
Example: names of students who are in more clubs than classes

Subqueries in WHERE Clause

- Simplest case: subquery returns a single row
  ~ Runtime error if subquery returns more than one row
  Example: students who are at the same age as Bart

- IN subquery: checks if something is in the table returned by the subquery
  ~ Also: NOT IN
  Example: students who are at the same age as Bart

- EXISTS(subquery): checks if the table returned by the subquery is nonempty
  ~ Also: NOT EXISTS
  Example: students who are at the same age as Bart

~ This example uses correlated subquery, i.e., a subquery that refers to values from a surrounding query
~ Notice the scoping rule: to find out which table a column belongs to, start with the immediately surrounding subquery; if not found, look in the one surrounding that, and so on

Jun Yang 4 CS145 Spring 1999
• Quantified subqueries:
  - \textit{ANY}—existential quantifier
  - \textit{ALL}—universal quantifier

\textbf{~ Beware:} in common parlance, “any” and “all” seem to be synonyms, e.g., “Bill has more money than any of us” \(\equiv\) “Bill has more money than all of us”; however, in SQL, \textit{ANY} really means “some”

Example using \textit{ALL}: which students have the highest GPA?

Example using \textit{ANY}: which students have the highest GPA?

Example using \textit{EXISTS}: which students have the highest GPA?

\begin{center}
\textbf{Aggregates}
\end{center}

\textbf{SUM, AVG, MIN, MAX, COUNT}

\textbf{~} Clearly goes beyond relational algebra in expressiveness

Example: number of students under 18, and their average GPA

\textbf{~} \texttt{COUNT(*)} counts the number of rows

\textbf{~} Duplicates do matter!

Example: how many students are taking classes?

\textbf{~} Use \texttt{DISTINCT} to eliminate duplicates when computing aggregates
GROUP BY Clause

Syntax: follow SELECT-FROM-WHERE by GROUP BY and a list of columns

Semantics: the table that is the result of the FROM (i.e., \( \times \)) and WHERE (i.e., \( \sigma \)) is grouped according to the values of GROUP BY columns, and aggregates are computed within each group

\( \Rightarrow \) Number of groups = number of rows in the output

\( \Rightarrow \) Without the GROUP BY clause, everything is in one big group

Example: find the average GPA for each age group

Note: If any aggregate is used, then every element of the SELECT clause must either be aggregated or appear in the GROUP BY clause

Example: which students have the highest GPA?

\( \Rightarrow \) a tempting, but incorrect way

\( \Rightarrow \) a correct way

HAVING Clause

Syntax: follow SELECT-FROM-WHERE-GROUP BY by HAVING and a condition

Semantics: for each group, evaluate the HAVING condition; if false, the group will not appear in the output

\( \Rightarrow \) Every column referenced by the HAVING clause must either be aggregated or appear in the GROUP BY clause (just like the rule for SELECT)

Example: SID’s of students who are in more clubs than classes
Summary of **SELECT** Statement

- **SELECT** expressions (columns, aggregates)
- **FROM** tables
- **WHERE** condition (no aggregates)
- **GROUP BY** columns (no aggregates)
- **HAVING** condition (only aggregates and/or **GROUP BY** columns)
- **ORDER BY** columns (if the query has no aggregates), or aggregates and/or **GROUP BY** columns (if the query has aggregates)

⇒ Everything is optional except **SELECT** and **FROM**

**Data Modification**

**INSERT**

- Insert one row
  - Example: Milhouse takes CS145

- Insert the result of a subquery
  - Example: force everybody to take CS145

**DELETE**

- Delete according to a condition
  - Example: Milhouse drops CS145
  - Example: CS145 students must not join “Database Haters’ Club”

- Delete everything
  - Example: DELETE FROM Take;

**UPDATE**

Example: student 123 changes name to “Barney”

Example: set 4.0 as the maximum GPA