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`

`SELECT` $A_1, A_2, ..., A_n$
`FROM` $R_1, R_2, ..., R_m$
`WHERE` $condition$;

- Called a SPJ (select-project-join) query
- Equivalent (more or less) to relation algebra query:
  $\pi_{A_1,...,A_n}(\sigma_{condition}(R_1 \times R_2 \times ... \times R_m))$

$\leadsto$ Returns an unnamed table with columns $A_1, A_2, ..., A_n$

Example schema:

`Student(`<u>`SID`</u>`, name, age, GPA)`
`Take(`<u>`SID`</u>`, `<u>`CID`</u>`)`
`Course(`<u>`CID`</u>`, 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 `SELECT` $*$ to output all columns in the cross product
  Example: `SELECT * FROM Student`;
  $\leadsto$ Note that `WHERE` clause is optional
  Example: `SELECT * FROM Course, Take WHERE` ...;
- Use `AS` in `SELECT` clause to rename output columns
  Example: `SELECT name AS studName FROM Student` ...;
- Use *tuple variables* in `FROM` clause to rename input tables
  Example: SID's of all pairs of classmates

  $\leadsto$ SQL2 permits an optional `AS` between the table and its tuple variable; Oracle does not
- `SELECT` list may also contain expressions
  Example: when was Lisa born?
- Use `LIKE` in `WHERE` clause for string matching
  Example: ID's of all students whose names start with the letter B
- Use `ORDER BY` clause to sort result rows
  Example: ID's of students over 18, sorted by GPA (descending) then name (ascending)

Operational Semantics of SPJ Queries

`SELECT` $E_1, \ldots\ldots, E_n$
`FROM` $R_1\ \ t_1, \ldots\ldots, R_m\ \ t_m$
`WHERE` $condition$;

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

By default, SQL has *bag semantics*, i.e., duplicate rows are retained

- Different from relational algebra, which has *set semantics*
  Example: $\pi_{\texttt{SID}}(\texttt{Take}) \not\equiv$ `SELECT SID FROM Take;`
- Use `DISTINCT` after `SELECT` to force set semantics
  Example: $\pi_{\texttt{SID}}(\texttt{Take}) \equiv$ `SELECT DISTINCT SID FROM Take;`
- Why bag semantics?
  - Saves time of eliminating duplicates
  - Which one is more useful? `SELECT GPA FROM Student;` or `SELECT DISTINCT GPA FROM Student;`?

UNION, EXCEPT, INTERSECT

Example schema: add another table `ClubMember(`<u>`club`</u>`,` <u>`SID`</u>`)`
Example: SID's of students who are taking classes and/or involved in clubs

- `UNION`, `EXCEPT`, `INTERSECT` eliminate duplicates (set semantics)
  - Exactly like set $\cup$, $-$, $\cap$
- `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

$\rightsquigarrow$ Oracle calls difference `MINUS` instead of `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
  $\rightsquigarrow$ 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
  $\rightsquigarrow$ 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
  $\rightsquigarrow$ Also: `NOT EXISTS`
  Example: students who are at the same age as Bart

  $\rightsquigarrow$ This example uses *correlated subquery*, i.e., a subquery that refers to values from a surrounding query
  $\rightsquigarrow$ 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

- Quantified subqueries:
  ANY—existential quantifier
  ALL—universal quantifier
  $\leadsto$ 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, ANY really means "some"
  Example using ALL: which students have the highest GPA?

  Example using ANY: which students have the highest GPA?

  Example using EXISTS: which students have the highest GPA?

Aggregates

SUM, AVG, MIN, MAX, COUNT
$\leadsto$ Clearly goes beyond relational algebra in expressiveness
Example: number of students under 18, and their average GPA
$\leadsto$ COUNT($*$) counts the number of rows
$\leadsto$ Duplicates do matter!

Example: how many students are taking classes?
$\leadsto$ Use 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., σ) is *grouped* according to the values of `GROUP BY` columns, and aggregates are computed within each group

- $\leadsto$ Number of groups = number of rows in the output
- $\leadsto$ 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?
$\leadsto$ a tempting, but incorrect way

$\leadsto$ 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

- $\leadsto$ *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)

$\leadsto$ 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