SQL: Part I

CPS 196.3
Introduction to Database Systems

SQL

- SQL: Structured Query Language
  - Pronounced “S-Q-L” or “sequel”
  - The standard query language support by most commercial DBMS
- A brief history
  - IBM System R
  - ANSI SQL89
  - ANSI SQL92 (SQL2)
  - SQL3 (still under construction after years!)

Creating and dropping tables

- CREATE TABLE table_name (...,
column_name1 column_type1,...);
- DROP TABLE table_name;
- Examples
  - create table Student (SID integer,
    name varchar(30), email varchar(30),
    age integer, GPA float);
  - create table Course (CID char(10), title varchar(100));
  - create table Enroll (SID integer, CID char(10));
  - drop table Student;
  - drop table Course;
  - drop table Enroll;
  -- SQL is insensitive to white space.
  -- SQL is case insensitive; writing ...Course... is equivalent to
  -- writing ...COURSE...;

Basic queries: SFW statement

- SELECT A1, A2, ..., An FROM R1, R2, ..., Rm
  WHERE condition;
- Also called an SPJ (select-project-join) query
- Equivalent (not really!) to relational algebra query
  \[ \pi_{A1,A2,...,An} (\sigma_{condition} (R1 \times R2 \times \ldots \times Rm)) \]

Example: reading a table

- SELECT * FROM Student;
  - 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 students under 18
  - SELECT name FROM Student WHERE age < 18;
- When was Lisa born?
  - SELECT 2002 - age
    FROM Student
    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
Example: join

- SID’s and name’s of students taking courses with the word “Database” in their titles
  - `SELECT Student.SID, Student.name
    FROM Student, Enroll, Course
    WHERE Student.SID = Enroll.SID
    AND Enroll.CID = Course.CID
    AND title LIKE '%Database%';`
  - LIKE matches a string against a pattern
    - % matches any sequence of 0 or more characters
    - Okay to omit `table_name` in `table_name.column_name` if `column_name` is unique

Example: rename

- SID’s of all pairs of classmates
  - Relational algebra query:
    \[ \pi_{e1.SID, e2.SID} (\rho_{e1.CID = e2.CID \land e1.SID > e2.SID} Enroll) \]
    - SQL:
      `SELECT e1.SID AS SID1, e2.SID AS SID2
       FROM Enroll AS e1, Enroll AS e2
       WHERE e1.CID = e2.CID
       AND e1.SID > e2.SID;`
  - AS keyword is completely optional

A more complicated example

- Titles of all courses that Bart and Lisa are taking together
  - `SELECT c.title
    FROM Student sb, Student sl, Enroll eb, Enroll el, Course c
    WHERE sb.name = 'Bart' AND sl.name = 'Lisa'
    AND eb.SID = sb.SID AND el.SID = sl.SID AND eb.CID = el.CID
    AND eb.CID = c.CID;`
  - 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: \( \pi_E \cdot (\sigma_P (R_1 \times \ldots \times R_m)) \)
  - Example: \( \pi_{A,B,C} (R \times S \times T) \) = \( \pi_{A,B,C} (\sigma_{P_1} \sigma_{P_2} \sigma_{P_3} (R \times S \times T)) \)
  - 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

- `\pi_{SID} Enroll`
A case for set semantics

- Efficiency
  - Saves time of eliminating duplicates
- Which one is more useful?
  - π\textsubscript{GPA} \text{Student}
  - SELECT GPA FROM Student;
  - The first query just returns all possible GPA's
  - The second query returns the actual GPA distribution
- Besides, SQL provides the option of set semantics with \textsc{DISTINCT} keyword

Example: forcing set semantics

- SID's of all pairs of classmates
  - SELECT e1.SID AS SID\textsubscript{1}, e2.SID AS SID\textsubscript{2}
    FROM Enroll AS e1, Enroll AS e2
    WHERE e1.CID = e2.CID
    AND e1.SID > e2.SID;
  - Say Bart and Lisa both take CPS196 and CPS114
  - SELECT DISTINCT e1.SID AS SID\textsubscript{1}, e2.SID AS SID\textsubscript{2}
    ...
  - With \textsc{DISTINCT}, all duplicate (SID\textsubscript{1}, SID\textsubscript{2}) pairs are removed from the output

Examples of bag operations

<table>
<thead>
<tr>
<th>Bag1</th>
<th>Bag2</th>
<th>Bag1 UNION ALL Bag2</th>
<th>Bag1 INTERSECT ALL Bag2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruit</td>
<td>Fruit</td>
<td>Fruit</td>
<td>Fruit</td>
</tr>
<tr>
<td>apple</td>
<td>apple</td>
<td>apple</td>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
<td>orange</td>
<td>orange</td>
<td>orange</td>
</tr>
</tbody>
</table>

Examples of set versus bag operations

<table>
<thead>
<tr>
<th>Enroll(SID, CID), ClubMember(club, SID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SELECT SID FROM ClubMember) EXCEPT (SELECT SID FROM Enroll);</td>
</tr>
<tr>
<td>SID's of students who are in clubs but not taking any classes</td>
</tr>
<tr>
<td>(SELECT SID FROM ClubMember) EXCEPT ALL (SELECT SID FROM Enroll);</td>
</tr>
<tr>
<td>SID’s of students who are in more clubs than classes</td>
</tr>
</tbody>
</table>

Operational semantics of SFW

- SELECT \{DISTINCT\} E\textsubscript{1}, E\textsubscript{2}, ..., E\textsubscript{n}
  FROM R\textsubscript{1}, R\textsubscript{2}, ..., R\textsubscript{m}
  WHERE condition;
- For each t\textsubscript{i} in R\textsubscript{i}:
  For each t\textsubscript{j} in R\textsubscript{j}:
  For each t\textsubscript{k} in R\textsubscript{k}:
    If condition is true over t\textsubscript{i}, t\textsubscript{j}, ..., t\textsubscript{k}:
      Compute and output E\textsubscript{i}, E\textsubscript{j}, ..., E\textsubscript{k}
- If \textsc{DISTINCT} is present
  Eliminate duplicate rows in output
- t\textsubscript{1}, t\textsubscript{2}, ..., t\textsubscript{n} are often called tuple variables

SQL set and bag operations

- UNION, EXCEPT, INTERSECT
  - Set semantics
  - 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
Summary of SQL features covered so far

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

Table expression

- Use query result as a table
  - In set and bag operations, FROM clauses, etc.
  - A way to “nest” queries
- Example: names of students who are in more clubs than classes
  ```sql
  SELECT DISTINCT name
  FROM Student,
  (SELECT SID FROM ClubMember)
  AS S
  WHERE Student.SID = S.SID;
  ```

Scalar subqueries

- A query that returns a single row can be used as a value in WHERE, SELECT, etc.
- Example: students at the same age as Bart
  ```sql
  SELECT * 
  FROM Student
  WHERE age = (SELECT age
  FROM Student
  WHERE name = 'Bart');
  ```
  - Runtime error if the subquery returns more than one row

IN subqueries

- `x IN (subquery)` checks if `x` is in the result of subquery
- Example: students at the same age as (some) Bart
  ```sql
  SELECT * 
  FROM Student
  WHERE age IN (SELECT age
  FROM Student
  WHERE name = 'Bart');
  ```

EXISTS subqueries

- `EXISTS (subquery)` checks if the result of subquery is non-empty
- Example: students at the same age as (some) Bart
  ```sql
  SELECT * 
  FROM Student AS s
  WHERE EXISTS (SELECT * FROM Student
  WHERE name = 'Bart'
  AND age = s.age);
  ```
  - It is a correlated subquery—a subquery that references tuple variables in surrounding queries

Operational semantics of subqueries

- For each row `s` in Student
  - Evaluate the subquery with the appropriate value of `s.age`
  - If the result of the subquery is not empty, output `s.*`
- The DBMS query optimizer may choose to process the query in an equivalent, but more efficient way (example?)
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

```sql
SELECT * FROM Student
WHERE EXISTS (SELECT * FROM Enroll e
  WHERE SID = s.SID
  AND EXISTS (SELECT * FROM Enroll
    WHERE SID = s.SID
    AND CID <> e.CID));
```

Students who are taking at least two courses

Quantified subqueries

- A quantified subquery can be used as a value in a `WHERE` condition
- Universal quantification (for all):
  ```sql
  WHERE x op ALL (subquery) ...
  ```
  - True iff for all `t` in the result of subquery, `x op t`
- Existential quantification (exists):
  ```sql
  WHERE x op ANY (subquery) ...
  ```
  - True iff there exists some `t` in the result of subquery 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 students have the highest GPA?
  ```sql
  SELECT *
  FROM Student
  WHERE GPA >= ALL (SELECT GPA FROM Student);
  ```
  ```sql
  SELECT *
  FROM Student
  WHERE NOT (GPA < ANY (SELECT GPA FROM Student));
  ```
  - Use `NOT` to negate a condition

More ways of getting the highest GPA

- Which students have the highest GPA?
  ```sql
  SELECT *
  FROM Student AS s
  WHERE NOT EXISTS (SELECT * FROM Student
    WHERE GPA > s.GPA);
  ```
  ```sql
  SELECT *
  FROM Student
  WHERE SID NOT IN (SELECT s1.SID
    FROM Student AS s1, Student AS s2
    WHERE s1.GPA < s2.GPA);
  ```

Summary of SQL features covered so far

- `SELECT-FROM-WHERE` statements
- Set and bag operations
- Table expressions, subqueries
  - Subqueries allow queries to be written in more declarative ways (recall the highest GPA query)
  - But they do not add any expressive power
    - Try translating other forms of subqueries into [NOT] EXISTS, which in turn can be translated into join (and difference)

- Next: aggregation and grouping
Aggregates

- Standard SQL aggregate functions: \texttt{COUNT}, \texttt{SUM}, \texttt{AVG}, \texttt{MIN}, \texttt{MAX}
- Example: number of students under 18, and their average GPA
  - \texttt{SELECT COUNT(*)}, \texttt{AVG(GPA)}
  - \texttt{FROM Student}
  - \texttt{WHERE age < 18;}
  - \texttt{COUNT(*)} counts the number of rows

Aggregates with \texttt{DISTINCT}

- Example: How many students are taking classes?
  - \texttt{SELECT COUNT(DISTINCT SID)}
  - \texttt{FROM Enroll;}
  - is equivalent to:
  - \texttt{SELECT COUNT(*)}
  - \texttt{FROM (SELECT DISTINCT SID, FROM Enroll);}

GROUP BY

- \texttt{SELECT ... FROM ... WHERE ...}
  - \texttt{GROUP BY list_of_columns;}
- Example: find the average GPA for each age group
  - \texttt{SELECT age, AVG(GPA)}
  - \texttt{FROM Student}
  - \texttt{GROUP BY age;}

Operational semantics of \texttt{GROUP BY}

\texttt{SELECT ... FROM ... WHERE ... GROUP BY ...;}

- Compute \texttt{FROM (\times)}
- Compute \texttt{WHERE (\sigma)}
- Compute \texttt{GROUP BY: group rows according to the values of \texttt{GROUP BY} columns}
- Compute \texttt{SELECT} for each group (\pi)
  - Number of groups = number of rows in the final output

Example of computing \texttt{GROUP BY}

\texttt{SELECT age, AVG(GPA) FROM Student GROUP BY age;}

Aggregates with no \texttt{GROUP BY}

- An aggregate query with no \texttt{GROUP BY} clause
  - represent a special case where all rows go into one group
  - \texttt{SELECT AVG(GPA) FROM Student;}
  - Group all rows into one group
  - Compute aggregate over the group
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
- This restriction ensures that any SELECT expression produces only one value for each group

Examples of invalid queries

- SELECT age FROM Student GROUP BY age;
  - Recall there is one output row per group
  - There can be multiple SID values per group
- SELECT MAX(GPA) FROM Student;
  - Recall there is only one group for an aggregate query with no GROUP BY clause
  - There can be multiple SID values
  - Wishful thinking (that the output SID value is the one associated with the highest GPA) does NOT work

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 (×)
    - Compute WHERE (σ)
    - Compute GROUP BY: group rows according to the values of GROUP BY columns
    - Compute HAVING (another σ over the groups)
    - Compute SELECT for each group that passes the HAVING condition (π)

HAVING examples

- Find the average GPA for each age group over 10
  - SELECT age, AVG(GPA)
    FROM Student
    GROUP BY age
    HAVING age > 10;
  - Can be written using WHERE without table expressions
- List the average GPA for each age group with more than a hundred students
  - SELECT age, AVG(GPA)
    FROM Student
    GROUP BY age
    HAVING COUNT(*) > 100;
  - Can be written using WHERE and table expressions

Summary of SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Table expressions, subqueries
- Aggregation and grouping
  - More expressive power than relational algebra
- Next: ordering output rows

ORDER BY

- SELECT (DISTINCT) ... 
  FROM ... WHERE ... GROUP BY ... HAVING ...
  ORDER BY output_column [ASC | DESC], ...;
- ASC = ascending, DESC = descending
- Operational 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 students, sort them by GPA (descending) and then name (ascending)
  - SELECT SID, name, age, GPA
    FROM Student
    ORDER BY GPA 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 of output columns instead
    ORDER BY 4 DESC, 2;

Summary of SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Table expressions, subqueries
- Aggregation and grouping
- Ordering
  - More expressive power than relational algebra

Next: NULL’s, outerjoins, data modification, constraints, …