SQL Part I
Relational-Algebra Queries with SELECT
THE DEVELOPMENT OF SQL

- SQL is the major query language for commercial applications
- Most implementations are “close” to SQL2 = SQL92
- Basic ideas are elegant, taken from relational algebra
- New versions of the standard “under construction”
- It is becoming an “ugly” patchwork of special cases
SQL for this Course

• We focus on the essential features of SQL2
  – Chapter 5 will serve as our reference
  – More detailed works are cited there

• SQL queries are written with the SELECT statement
  – In its full form, SELECT is complex
  – The sub-form SELECT-FROM-WHERE covers relational algebra queries
Running Example

STUDENT(SID, Name, City, Major)
ENROLL(SID, CourseName)
The Most Common Query Format

• Form the product, filter it down, and then project:
  \[ \pi_{A_1, \ldots, A_m}(\sigma_F(R_1 \times \cdots \times R_n)) \]

• Using SQL syntax, we write:
  SELECT \(A_1, \ldots, A_m\)
  FROM \(R_1, \ldots, R_n\)
  WHERE \(F\);

• Notes
  – SELECT produces a new relation
  – Different order in which the three steps are listed
  – Commas in FROM line are Cartesian products
**Queries on a Single Relation**

- In relational algebra: $\pi_{A_1, \ldots, A_m}(\sigma_F(R))$
- \texttt{SELECT } $A_1, \ldots, A_m$
  \texttt{FROM } $R$
  \texttt{WHERE } $F$;

- Query: find the students living in Boston
  - $\pi_{NAME}(\sigma_{CITY=\text{BOSTON}}(\text{STUDENT}))$
  - \texttt{SELECT NAME}
    \texttt{FROM STUDENT}
    \texttt{WHERE (CITY = 'BOSTON');}
Queries on a single relation (cont’d)

- Produce a table of students in Boston and their majors:
  ```sql
  SELECT NAME, MAJOR
  FROM STUDENT
  WHERE (CITY = 'BOSTON');
  ```

- Produce full data for students in Boston:
  ```sql
  SELECT *
  FROM STUDENT
  WHERE (CITY = 'BOSTON');
  ```

- The asterick * stands for all the attributes, in this case SID, NAME, CITY, MAJOR
Storing the Results of a Query

- INSERT INTO can be used to store the results of a query:

- INSERT INTO RESULT
  
  SELECT *
  FROM STUDENT
  WHERE (CITY = 'BOSTON');

- SELECT NAME, MAJOR
  FROM RESULT;
Tables Versus Relations

- Strictly speaking, SQL processes tables not relations
- Tables have extra structure that relations don’t have:
  - The rows are ordered
  - Tables may contain duplicate rows
- The SELECT statement allows us to control:
  - The order of rows, with ORDER BY
  - The processing of duplicates, with DISTINCT
The ORDER BY Clause

- Alphabetically list the C.S. students and their city:
  SELECT NAME,CITY
  FROM STUDENT
  WHERE (MAJOR = 'CS')
  ORDER BY NAME;

- List the students in Boston, ordered by name within each major:
  SELECT MAJOR,NAME,ADDRESS
  FROM STUDENT
  WHERE (CITY = 'BOSTON')
  ORDER BY MAJOR,NAME;
Removing Duplicate Rows with SELECT DISTINCT

- By default, duplicate rows are not eliminated
- SELECT DISTINCT causes duplicates to be eliminated
- List all cities that C.S. students live in:
  SELECT DISTINCT CITY
  FROM STUDENT
  WHERE (MAJOR = 'CS')
- In general, SELECT DISTINCT is more expensive than SELECT
Union, Intersection, and Difference

- \((\text{SELECT} \ldots )\) UNION \((\text{SELECT} \ldots )\);
- \((\text{SELECT} \ldots )\) INTERSECT \((\text{SELECT} \ldots )\);
- \((\text{SELECT} \ldots )\) EXCEPT \((\text{SELECT} \ldots )\);

Unlike SELECT, these operators do eliminate duplicates.
ILLUSTRATION

Give names for biology students at Harvard and Yale:

- $\pi_{\text{name}}(\sigma_{\text{major}=\text{bio}}(HARV)) \cup \pi_{\text{name}}(\sigma_{\text{major}=\text{bio}}(YALE))$

- (SELECT NAME
  FROM HARV
  WHERE (MAJOR='bio'))
  UNION
  (SELECT NAME
  FROM YALE
  WHERE (MAJOR='bio'));
Membership Constraints

Find all students who major in biology or chemistry.

- $\pi_{\text{name}}(\sigma_{\text{major}=\text{bio \ OR \ major}=\text{chem}}(\text{STUDENT}))$
- $\pi_{\text{name}}(\sigma_{\text{major}\in\{\text{bio,chem}\}}(\text{STUDENT}))$
- SELECT NAME
  FROM STUDENT
  WHERE (\text{MAJOR}=\text{'bio'} \ OR \ \text{MAJOR}=\text{'chem'})
- SELECT NAME
  FROM STUDENT
  WHERE (\text{MAJOR IN (}'\text{bio}','\text{chem}'\text{)})
Range Constraints

The following are equivalent:

• The SQL constraint: $X$ BETWEEN $X_1$ AND $X_2$
• $X \in [X_1, X_2]$
• $X \geq X_1$ & $X \leq X_2$

Example:

• SELECT *
  FROM STUDENT
  WHERE (NAME BETWEEN 'M' AND 'T');
• $\sigma_{name \in ['M', 'T']} (STUDENT)$
**Pattern-Matching Constraints**

- Find all movie titles containing the substring “Cop”:
  
  ```sql
  SELECT TITLE
  FROM MOVIE
  WHERE (TITLE LIKE '%Cop%');
  ```

- Find all movie titles whose third letter is ’z’:
  
  ```sql
  SELECT TITLE
  FROM MOVIE
  WHERE (TITLE LIKE '_z%');
  ```
Renaming the Attributes of a Relation

- Get names, cities and majors of students, and rename the first two attributes to 'lastname' and 'c'
- $\pi_{\text{name} \rightarrow \text{lastname}, \text{city} \rightarrow c, \text{major} } (\text{STUDENT})$
- SELECT
  NAME AS LASTNAME, CITY AS C, MAJOR
FROM STUDENT
JOINS AND OTHER MULTI-TABLE QUERIES

• Find students enrolled in History, given the schema:
  – STUDENT(SID,NAME,CITY,MAJOR)
  – ENROLL(SID,COURSE)

• \( \pi_{s.sid \rightarrow sid, name} (\sigma_{course=\text{hist} \& s.sid=e.sid (\text{student} \times \text{enroll})}) \)

• SELECT S.SID AS SID, NAME
  FROM STUDENT AS S, ENROLL AS E
  WHERE (COURSE=’hist’ AND S.SID=E.SID)

• SELECT SID, NAME
  FROM (STUDENT NATURAL JOIN ENROLL)
  WHERE COURSE=’hist’
Aggregation

- We can aggregate the values of relational attributes
- The basic aggregate operators are:
  - **SUM** — computes the sum
  - **AVG** — computes the average
  - **MAX** — computes the maximum
  - **MIN** — computes the minimum
  - **COUNT** — counts the number of tuples
- With AVG, SUM and COUNT, we must specify whether duplicates should be removed before the operator is applied
Queries with Aggregates

• Consider the table:
  ORDERS(OID, CUSTOMER, AMOUNT, YEAR)

• Compute the average amount of an order from 1990:

  \[ \text{AVG}(\pi_{\text{amount}}(\sigma_{\text{year}=1990}(\text{ORDERS}))) \]

  \[ \text{SELECT AVG(AMOUNT) FROM ORDERS WHERE YEAR=1990;} \]

• We were not allowed to remove duplicates before computing this average
Queries with Aggregates (continued)

- Count the number of orders placed in 1990:
  \[
  \text{SELECT COUNT(OID)} \\
  \text{FROM ORDERS} \\
  \text{WHERE YEAR=1990;}
  \]

- Count the number of different dates on which orders were posted by customer 109:
  \[
  \text{SELECT COUNT(DISTINCT DATE)} \\
  \text{FROM ORDERS} \\
  \text{WHERE (CUSTOMER = 245);}
  \]

- Here we must remove duplicates
**GROUPING**

- **Step 1:** Group the tuples of a relation into *blocks*
- **Step 2:** Form a new relation by applying aggregates to the blocks

- **Example:**
  - Given the table `STUDENT(NAME, DEPT, GPA)`, make a new table showing, for each department, the maximum and average gpa
  - `SELECT DEPT, MAX(GPA), AVG(GPA) FROM STUDENT GROUP BY DEPT`
**Grouping with a WHERE Clause**

- The WHERE clause filters out tuples *before* the grouping is performed.
- For each department, count how many students have GPA greater than 3.0.

```sql
SELECT DEPT, COUNT(NAME)
FROM STUDENT
WHERE GPA > 3.0
GROUP BY DEPT
```
Grouping with a HAVING Clause

- The HAVING clause is a condition that filters down the set of blocks, after the grouping has been done.
- Show the statistics for all departments with average GPA greater than 3.0.
- `SELECT DEPT, MAX(GPA), AVG(GPA)
FROM STUDENT
GROUP BY DEPT
HAVING AVG(GPA) > 3.0`
It is possible to have quite a sophisticated query:

List for each C# the largest order posted on January 1, 1990, if the average order posted on that date by that C# was greater than 100.

```sql
SELECT C#, MAX (AMT) 
FROM ORDERS 
WHERE (DATE = 1990.1.1) 
GROUP BY C# 
HAVING AVG(AMT) > 100; 
```
Some notes on procedurality

SQL is really a procedural language in that different parts of the query are conceptually handled in a definite order. (The implementation may do things differently but with the same result.)

1. Take a cartesian product of FROM list.
2. Apply the selection conditions without doing projections.
3. Apply the grouping conditions without doing projections.
4. Apply the aggregates without removing duplicates unless DISTINCT is used.

5. Apply HAVING on aggregates derived from groups.

6. Output the result without removing duplicates unless DISTINCT is used.