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 SQL96
  • ANSI SQL89
  • ANSI SQL92 (SQL2)
  • ANSI SQL99 (SQL3)
  • ANSI SQL 2003 (added OLAP, XML, etc.)
  • ANSI SQL 2006 (added more XML)
  • ANSI SQL 2008, …
SQL

• **Data-definition language (DDL):** define/modify schemas, delete relations

• **Data-manipulation language (DML):** query information, and insert/delete/modify tuples

• **Integrity constraints:** specify constraints that the data stored in the database must satisfy

• **Intermediate/Advanced topics:** *(next week)*
  - E.g., triggers, views, indexes, programming, recursive queries
### DDL

- **CREATE TABLE** *table_name*  
  (...,*column_name column_type*, ...);

```
CREATE TABLE User (uid DECIMAL(3,0), name VARCHAR(30), age DECIMAL(2,0), pop DECIMAL(3,2));
CREATE TABLE Group (gid CHAR(10), name VARCHAR(100));
CREATE TABLE Member (uid DECIMAL (3,0), gid CHAR(10));
```

- **DROP TABLE** *table_name*;

```
DROP TABLE User;
DROP TABLE Group;
DROP TABLE Member;
```

-- everything from -- to the end of line is ignored.  
-- SQL is insensitive to white space.  
-- SQL is insensitive to case (e.g., ...CREATE... is  
-- equivalent to ...create...).

**User** *(uid int, name string, age int, pop float)*  
**Group** *(gid string, name string)*  
**Member** *(uid int, gid string)*

---

How does it work with MySQL?
Basic queries for DML: SFW statement

- SELECT $A_1, A_2, \ldots, A_n$
  FROM $R_1, R_2, \ldots, R_m$
  WHERE condition;

- Also called an SPJ (select-project-join) query

- Corresponds to (but not really equivalent to) relational algebra query:
  $$\pi_{A_1, A_2, \ldots, A_n}(\sigma_{\text{condition}}(R_1 \times R_2 \times \cdots \times R_m))$$
Examples

• List all rows in the User table
  
  SELECT * FROM User;
  
  • * is a short hand for “all columns”

• List name of users under 18 (selection, projection)
  
  SELECT name FROM User where age < 18;

• When was Lisa born?
  
  SELECT 2021-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
Example: join

- List 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 ....;
Example: join

• List ID’s and names of groups with a user whose name contains “Simpson”

```sql
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

User (uid int, name string, age int, pop float)
Group (gid string, name string)
Member (uid int, gid string)
Example: rename

- ID’s of all pairs of users that belong to one group
  - Relational algebra query:
    \[ \pi_{m_1.uid, m_2.uid} (\rho_{m_1.Member \bowtie_{m_1.gid = m_2.gid \land m_1.uid > m_2.uid} \rho_{m_2.Member}) \]
  - SQL (not exactly):
    ```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

User (uid int, name string, age int, pop float)
Group (gid string, name string)
Member (uid int, gid string)
A more complicated example

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

```sql
SELECT g.name
   FROM User u1, ..., Member m1, ...
   WHERE u1.name = 'Lisa' AND ...
      AND u1.uid = m1.uid AND ...
      AND ...;
```

**User** (uid int, name string, age int, pop float)
**Group** (gid string, name string)
**Member** (uid int, gid string)
A more complicated example

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

```sql
SELECT g.name
FROM User u1, User u2, Member m1, Member m2, ...
WHERE u1.name = 'Lisa' AND u2.name = 'Ralph'
AND u1.uid = m1.uid AND u2.uid = m1.uid
AND ...;
```

**User** (uid int, name string, age int, pop float)
**Group** (gid string, name string)
**Member** (uid int, gid string)
A more complicated example

- **Names of all groups** that Lisa and Ralph are both in

```sql
SELECT g.name
FROM User u1, User u2, Member m1, Member m2, Group g
WHERE u1.name = 'Lisa' AND u2.name = 'Ralph'
    AND u1.uid = m1.uid AND u2.uid = m2.uid
    AND m1.gid = g.gid AND m2.gid = g.gid;
```

**User** (uid int, name string, age int, pop float)

**Group** (gid string, name string)

**Member** (uid int, gid string)
Why SFW statements?

• Many queries can be written using only selection, projection, and cross product (or join)

• These queries can be written in a canonical form which is captured by SFW:

\[ \pi_L \left( \sigma_p (R_1 \times \cdots \times R_m) \right) \]

• Example: \( \pi_{R.A,S.B}(R \bowtie_{p_1} S) \bowtie_{p_2} (\pi_{T.c} \sigma_{p_3} T) \)
  
  \[ = \pi_{R.A,S.B,T.c} \sigma_{p_1 \land p_2 \land p_3} (R \times S \times T) \]
Set versus bag

User

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Set
• No duplicates
• Relational model and algebra use set semantics

Bag
• Duplicates allowed
• Number of duplicates is significant
• SQL uses bag semantics by default

\[ \pi_{age} \text{User} \]

SELECT age FROM User;

age
10
8
8
8
...
A case for bag semantics

• Efficiency
  • Saves time of eliminating duplicates

• Which one is more useful?
  
  \[ \pi_{\text{age User}} \]

  SELECT age
  FROM User;

  • The first query just returns all possible user ages
  • The second query returns the user age distribution

• Besides, SQL provides the option of set semantics with \textbf{DISTINCT} keyword
Forcing set semantics

- ID’s of all pairs of users that belong to one group

```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;
```

Say Lisa and Ralph are in both the book club and the student government, they id pairs will appear twice

- Remove duplicate (uid1, uid2) pairs from the output

```sql
SELECT DISTINCT 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;
```
Semantics of SFW

• SELECT [DISTINCT] \( E_1, E_2, ..., E_n \)
  FROM \( R_1, R_2, ..., R_m \)
  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 \( t_1, t_2, ..., t_m \):
        Compute and output \( E_1, E_2, ..., E_n \) as a row

If DISTINCT is present
  Eliminate duplicate rows in output

• \( t_1, t_2, ..., t_m \) are often called tuple variables
SQL set and bag operations

- **Set:** UNION, EXCEPT, INTERSECT
  - Exactly like set $\cup$, $-$, and $\cap$ in relational algebra
  - Duplicates in input tables, if any, are first eliminated
  - Duplicates in result are also eliminated (for UNION)

<table>
<thead>
<tr>
<th>Bag1</th>
<th>Bag2</th>
<th>Result UNION</th>
<th>Result EXCEPT</th>
<th>Result INTERSECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>fruit</td>
<td>fruit</td>
<td>(SELECT * FROM Bag1) UNION (SELECT * FROM Bag2);</td>
<td>(SELECT * FROM Bag1) EXCEPT (SELECT * FROM Bag2);</td>
<td>(SELECT * FROM Bag1) INTERSECT (SELECT * FROM Bag2);</td>
</tr>
<tr>
<td>apple</td>
<td>orange</td>
<td>fruit</td>
<td>fruit</td>
<td>fruit</td>
</tr>
<tr>
<td>apple</td>
<td>orange</td>
<td>apple</td>
<td>apple</td>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
<td>orange</td>
<td>orange</td>
<td>orange</td>
<td>orange</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result UNION</th>
<th>Result EXCEPT</th>
<th>Result INTERSECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>fruit</td>
<td>fruit</td>
<td>fruit</td>
</tr>
<tr>
<td>apple</td>
<td>orange</td>
<td>orange</td>
</tr>
<tr>
<td>apple</td>
<td>orange</td>
<td>orange</td>
</tr>
</tbody>
</table>
SQL set and bag operations

- **Set:** UNION, EXCEPT, INTERSECT
  - Exactly like set $\cup$, $\setminus$, and $\cap$ in relational algebra
- **Bag:** UNION ALL, EXCEPT ALL, INTERSECT ALL
  - Think of each row as having an implicit **count** (the number of times it appears in the table)

<table>
<thead>
<tr>
<th>Bag1</th>
<th>Bag2</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>apple</td>
</tr>
<tr>
<td>apple</td>
<td>apple</td>
</tr>
<tr>
<td>orange</td>
<td>orange</td>
</tr>
<tr>
<td>apple: 2 orange:1</td>
<td>apple: 1 orange:2</td>
</tr>
</tbody>
</table>

(SELECT * FROM Bag1)
UNION ALL
(SELECT * FROM Bag2);

sum up the counts from two tables

<table>
<thead>
<tr>
<th>fruit</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
</tr>
<tr>
<td>apple</td>
</tr>
<tr>
<td>apple</td>
</tr>
<tr>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
</tr>
<tr>
<td>apple: 3 orange:3</td>
</tr>
</tbody>
</table>
SQL set and bag operations

• Set: UNION, EXCEPT, INTERSECT
  • Exactly like set $\cup$, $-$, and $\cap$ in relational algebra

• Bag: UNION ALL, EXCEPT ALL, INTERSECT ALL
  • Think of each row as having an implicit count (the number of times it appears in the table)

(\text{SELECT } * \text{ FROM Bag1}) \text{ EXCEPT ALL} (\text{SELECT } * \text{ FROM Bag2};

\begin{array}{|c|c|c|}
\hline
\text{Bag1} & \text{Bag2} & \text{fruit} \\
\text{apple} & \text{apple} & \text{apple} \\
\text{apple} & \text{orange} & \text{proper-subtract the two counts} \\
\text{orange} & \text{orange} & \text{apple: 2} \\
\text{apple: 2} & \text{apple: 1} & \text{orange: 1} \\
\text{orange: 1} & \text{orange: 2} & \text{orange: 2} \\
\hline
\end{array}
SQL set and bag operations

• Set: UNION, EXCEPT, INTERSECT
  • Exactly like set $\cup$, $-$, and $\cap$ in relational algebra
• Bag: UNION ALL, EXCEPT ALL, INTERSECT ALL
  • Think of each row as having an implicit count (the number of times it appears in the table)

<table>
<thead>
<tr>
<th>Bag1</th>
<th>Bag2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fruit</strong></td>
<td><strong>fruit</strong></td>
</tr>
<tr>
<td>apple</td>
<td>apple</td>
</tr>
<tr>
<td>apple</td>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
<td>orange</td>
</tr>
</tbody>
</table>

(SELECT * FROM Bag1) INTERSECT ALL (SELECT * FROM Bag2);

take the minimum of the two counts
Set versus bag operations

Poke (uid1, uid2, timestamp)
  • uid1 poked uid2 at timestamp

Question: How do these two queries differ?

Q1:
(SELECT uid1 FROM Poke)
EXCEPT
(SELECT uid2 FROM Poke);

Q2:
(SELECT uid1 FROM Poke)
EXCEPT ALL
(SELECT uid2 FROM Poke);
Set versus bag operations

Poke (uid1, uid2, timestamp)
• uid1 poked uid2 at timestamp

Question: How do these two queries differ?

Q1:
(SELECT uid1 FROM Poke)
EXCEPT
(SELECT uid2 FROM Poke);

Users who poked others but never got poked by others

Q2:
(SELECT uid1 FROM Poke)
EXCEPT ALL
(SELECT uid2 FROM Poke);

Users who poked others more than others poked them
SQL features covered so far

- SELECT-FROM-WHERE statements
- 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.

• Example: names of *users who poked others more than others poked them*

```sql
SELECT DISTINCT name
FROM User,
    (SELECT uid1 FROM Poke)
EXCEPT ALL
    (SELECT uid2 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 Bart

```
SELECT *
FROM User,
WHERE age = (SELECT age
FROM User
WHERE name = 'Bart');
```

- When can this query go wrong?
  - Return more than 1 row
  - Return no rows
IN subqueries

• $x$ IN (subquery) checks if $x$ is in the result of subquery

• Example: users at the same age as (some) Bart

```
SELECT *
FROM User,
WHERE age IN (SELECT age
    FROM User
    WHERE name = ‘Bart’);
```
EXISTS subqueries

- **EXISTS** *(subquery)* checks if the result of *subquery* is non-empty

- Example: users at the same age as (some) Bart

```sql
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
Another example

• Users who join at least two groups

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

• How to find which table a column belongs to?
  • Start with the immediately surrounding query
  • If not found, look in the one surrounding that; repeat if necessary

User (uid int, name string, age int, pop float)
Group (gid string, name string)
Member (uid int, gid string)

Use

*table_name*. *
*column_name* notation and AS (renaming) to avoid confusion
Quantified subqueries

• Universal quantification (for all):
  • ... WHERE $x$ op $\text{ALL}(\text{subquery})$ ...
  • True iff for all $t$ in the result of $\text{subquery}$, $x$ op $t$

```sql
SELECT *
FROM User
WHERE pop >= \text{ALL}(\text{SELECT pop FROM User});
```

• Existential quantification (exists):
  • ... WHERE $x$ op $\text{ANY}(\text{subquery})$ ...
  • True iff there exists some $t$ in $\text{subquery}$ result s.t. $x$ op $t$

```sql
SELECT *
FROM User
WHERE \text{NOT} (pop < \text{ANY}(\text{SELECT pop FROM User}));
```
More ways to get the most popular

• Which users are the most popular?

Q1. SELECT *
    FROM User
    WHERE pop >= ALL(SELECT pop FROM User);

Q2. SELECT *
    FROM User
    WHERE NOT (pop < ANY(SELECT pop FROM User);

Q3. SELECT *
    FROM User AS u
    WHERE NOT EXISTS (SELECT * FROM User
    WHERE pop > u.pop);

Q4. SELECT * FROM User
    WHERE uid NOT EXISTS (SELECT u1.uid
    FROM User AS u1, User AS u2
    WHERE u1.pop < u2.pop);
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

Next: aggregation and grouping
Aggregates

• Standard SQL aggregate functions: COUNT, SUM, AVG, MIN, MAX

• Example: number of users under 18, and their average popularity
  • COUNT(*) counts the number of rows

  ```sql
  SELECT COUNT(*), AVG(pop)
  FROM User
  WHERE age < 18;
  ```
Aggregates with DISTINCT

• Example: How many users are in some group?

SELECT COUNT(*)
FROM (SELECT DISTINCT uid FROM Member);

Is equivalent to

SELECT COUNT(DISTINCT uid)
FROM Member;
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;
```
Example of computing GROUP BY

```
SELECT age, AVG(pop) FROM User GROUP BY age;
```

Compute GROUP BY: group rows according to the values of GROUP BY columns

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Compute SELECT for each group

```
<table>
<thead>
<tr>
<th>age</th>
<th>avg_pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.55</td>
</tr>
<tr>
<td>8</td>
<td>0.50</td>
</tr>
</tbody>
</table>
```
Semantics of GROUP BY

1. Compute FROM ($\times$)
2. Compute WHERE ($\sigma$)
3. Compute GROUP BY: group rows according to the values of GROUP BY columns
4. Compute SELECT for each group ($\pi$)
   - For aggregation functions with DISTINCT inputs, first eliminate duplicates within the group

Number of groups = number of rows in the final output
Aggregates with no GROUP BY

- An aggregate query with no GROUP BY clause = all rows go into one group

```
SELECT AVG(pop) FROM User;
```
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

Why?

This restriction ensures that any SELECT expression produces only one value for each group

```
SELECT uid, age FROM User GROUP BY age;
```

```
SELECT uid, MAX(pop) FROM User;
```
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;

1. Compute FROM (\times)
2. Compute WHERE (\sigma)
3. Compute GROUP BY: group rows according to the values of GROUP BY columns
4. Compute HAVING (another \sigma over the groups)
5. Compute SELECT (\pi) for each group that passes HAVING
HAVING examples

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

```sql
SELECT age, AVG(pop)
FROM User
GROUP BY age
HAVING COUNT(*)>100;
```

- Can be written using WHERE and table subqueries

```sql
SELECT T.age, T.apop
FROM (SELECT age, AVG(pop) AS apop, COUNT(*) AS gsize
     FROM User
     GROUP BY age) AS T
WHERE T.gsize>100;
```
HAVING examples

• Find average popularity for each age group over 10

```
SELECT age, AVG(pop)
FROM User
GROUP BY age
HAVING age >10;
```

• Can be written using WHERE without table subqueries

```
SELECT age, AVG(pop)
FROM User
WHERE age >10
GROUP BY age;
```
SQL features covered so far

• SELECT-FROM-WHERE statements
• Set and bag operations
• 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

• 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

• Query
  • SELECT-FROM-WHERE statements
  • Set/bag (DISTINCT, UNION/EXCEPT/INTERSECT (ALL))
  • Subqueries (table, scalar, IN, EXISTS, ALL, ANY)
  • Aggregation and grouping (GROUP BY, HAVING)
  • Ordering (ORDER)
  • Outerjoins (and Nulls)

• Modification
  • INSERT/DELETE/UPDATE

• Constraints

Lecture 4