SQL: Part I

Introduction to Databases
CompSci 316 Fall 2016

Announcements (Tue. Sep. 20)

- Homework #1 due today 11:59pm
- Homework #2 assigned
- Project mixer next Tuesday in class; details to follow in email

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

Creating and dropping tables

- `CREATE TABLE` *table_name* `(…,` *column_name column_type*`, …);`
- `DROP TABLE` *table_name*`;`
- Examples

```
create table User(uid integer, name varchar(30),
                  age integer, pop float);
create table Group(gid char(10), name varchar(100));
create table Member(uid integer, gid char(10));
drop table Member;
drop table Group;
drop table User;
-- everything from -- to the end of line is ignored.
-- SQL is insensitive to white space.
-- SQL is insensitive to case (e.g., ...Group... is
-- equivalent to ...GROUP...)
```

Basic queries: SFW statement

- `SELECT` A_1`,` A_2`, …,` A_n
  `FROM` R_1`,` R_2`, …,` R_m
  `WHERE` *condition*`;`
- Also called an SPJ (select-project-join) query
- Corresponds to (but not really equivalent to) relational algebra query:

$$\pi_{A_1,A_2,\ldots,A_n}\left(\sigma_{condition}(R_1 \times R_2 \times \cdots \times R_m)\right)$$

Example: reading a table

- `SELECT * FROM User;`
  - 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 users under 18
  - `SELECT name FROM User WHERE age<18;`
- When was Lisa born?
  - `SELECT 2016-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

- 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 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

Example: rename

- ID's of all pairs of users that belong to one group
  - Relational algebra query:

    $\pi_{m_1.uid,m_2.uid}(\rho_{m_1}Member \bowtie_{m_1.gid=m_2.gid \wedge m_1.uid>m_2.uid} \rho_{m_2}Member)$
  - 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

10

A more complicated example

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

```
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;
```

Tip: Write the `FROM` clause first, then `WHERE`, and then `SELECT`

11

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: $\pi_L\left(\sigma_p(R_1\times\cdots\times R_m)\right)$
    - Example: $\pi_{R.A,S.B}(R \bowtie_{p_1} S) \bowtie_{p_2} (\pi_{T.C}\sigma_{p_3}T) = \pi_{R.A,S.B,T.C}\sigma_{p_1\wedge p_2\wedge p_3}(R\times S\times T)$
  - `SELECT-FROM-WHERE` captures this canonical form

12

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

13

Set versus bag example

Member

uid	gid
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
…	…

$\pi_{gid} Member$

gid
dps
gov
abc
…

```
SELECT gid
FROM Member;
```

gid
dps
gov
abc
gov
abc
gov
…

14

A case for bag semantics

- Efficiency
  - Saves time of eliminating duplicates
- Which one is more useful?
  - $\pi_{age} User$
  - `SELECT age FROM User;`
  - The first query just returns _____
  - The second query returns _____
- Besides, SQL provides the option of set semantics with `DISTINCT` keyword

15

Forcing set semantics

- ID's of all pairs of users that belong to one group
  - ```
    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;
    ```
  - `SELECT DISTINCT m1.uid AS uid1, m2.uid AS uid2 …`
    - With `DISTINCT`, all duplicate (`uid1`, `uid2`) pairs are removed from the output
    ```

16

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 $t_1, t_2, \ldots, t_m$:
  Compute and output $E_1, E_2, \ldots, E_n$ as a row
  If `DISTINCT` is present
  Eliminate duplicate rows in output
- $t_1, t_2, \ldots, t_m$ are often called tuple variables

17

SQL set and bag operations

- `UNION`, `EXCEPT`, `INTERSECT`
  - Set semantics
    - Duplicates in input tables, if any, are first eliminated
    - Duplicates in result are also eliminated (for `UNION`)
  - Exactly like set ∪, −, and ∩ 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

18

Examples of bag operations

Bag1

fruit
apple
apple
orange

Bag2

fruit
apple
orange
orange

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

fruit
apple
apple
orange
apple
orange
orange

```
(SELECT * FROM Bag1)
EXCEPT ALL
(SELECT * FROM Bag2);
```

fruit
apple

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

fruit
apple
orange

19

Examples of set versus bag operations

Poke (*uid1*, *uid2*, *timestamp*)

- `(SELECT uid1 FROM Poke)`
  `EXCEPT`
  `(SELECT uid2 FROM Poke);`
  - Users who _____
- `(SELECT uid1 FROM Poke)`
  `EXCEPT ALL`
  `(SELECT uid2 FROM Poke);`
  - Users who _____

20

SQL features covered so far

- `SELECT-FROM-WHERE` statements (select-project-join queries)
- Set and bag operations

☞Next: how to nest SQL queries

21

Table subqueries

- Use query result as a table
  - In set and bag operations, `FROM` clauses, etc.
  - A way to "nest" queries
- Example: names of users who poked others more than others poked them
  - ```
    SELECT DISTINCT name
    FROM User,
         ((SELECT uid1 AS uid FROM Poke)
          EXCEPT ALL
          (SELECT uid2 AS uid 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
  - What's Bart's age?

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

  - Runtime error if subquery returns more than one row
    - Under what condition will this error never occur?
  - What if the subquery returns no rows?
    - The answer is treated as a special value `NULL`, and the comparison with `NULL` will fail

`IN` subqueries

- *x* `IN (`*subquery*`)` checks if *x* is in the result of *subquery*
- Example: users at the same age as (some) Bart
  - What's Bart's age?

```
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

```
SELECT *
FROM Users 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

25

Semantics of subqueries

- ```
  SELECT *
  FROM Users AS u
  WHERE EXISTS (SELECT * FROM User
                WHERE name = 'Bart'
                AND age = u.age);
  ```
- For each row `u` in `User`
  - Evaluate the subquery with the value of `u.age`
  - If the result of the subquery is not empty, output `u.*`
- The DBMS query optimizer may choose to process the query in an equivalent, but more efficient way (example?)

26

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

27

Another example

- ```
  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));
  ```
- Users who _____

28

Quantified subqueries

- A quantified subquery can be used syntactically as a value in a `WHERE` condition
- Universal quantification (for all):
  … `WHERE` *x op* `ALL(`*subquery*`)` …
  - True iff for all *t* in the result of *subquery*, *x op t*
- Existential quantification (exists):
  … `WHERE` *x op* `ANY(`*subquery*`)` …
  - True iff there exists some *t* in *subquery* result such that *x op t*
  - ☞Beware
    - In common parlance, "any" and "all" seem to be synonyms
    - In SQL, `ANY` really means "some"

29

Examples of quantified subqueries

- Which users are the most popular?

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

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

☞Use `NOT` to negate a condition

30

More ways to get the most popular

- Which users are the most popular?

```
SELECT *
FROM User AS u
WHERE NOT EXISTS
    ____________________
    ____________________;
```

```
SELECT * FROM User
WHERE uid NOT IN
    (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
    - Try translating other forms of subqueries into `[NOT] EXISTS`, which in turn can be translated into join (and difference)
      - Watch out for number of duplicates though

☞Next: aggregation and grouping

Aggregates

- Standard SQL aggregate functions: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`
- Example: number of users under 18, and their average popularity
  - ```
    SELECT COUNT(*), AVG(pop)
    FROM User
    WHERE age < 18;
    ```
  - `COUNT(*)` counts the number of rows

Aggregates with `DISTINCT`

- Example: How many users are in some group?
  - ```
    SELECT COUNT(DISTINCT uid)
    FROM Member;
    ```

  is equivalent to:
  - ```
    SELECT COUNT(*)
    FROM (SELECT DISTINCT uid FROM Member);
    ```
    ```

34

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;`

35

Semantics of GROUP BY

`SELECT … FROM … WHERE … GROUP BY …;`

- Compute `FROM` (×)
- Compute `WHERE` (σ)
- Compute `GROUP BY`: group rows according to the values of `GROUP BY` columns
- Compute `SELECT` for each group (π)
  - For aggregation functions with `DISTINCT` inputs, first eliminate duplicates within the group

☞ Number of groups =
number of rows in the final output

36

Example of computing GROUP BY

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

uid	name	age	pop
142	Bart	10	0.9
857	Lisa	8	0.7
123	Milhouse	10	0.2
456	Ralph	8	0.3

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

uid	name	age	pop
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3

Compute SELECT for each group

age	avg_pop
10	0.55
8	0.50

37

Aggregates with no GROUP BY

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

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

Group all rows into one group

Aggregate over the whole group

uid	name	age	pop
142	Bart	10	0.9
857	Lisa	8	0.7
123	Milhouse	10	0.2
456	Ralph	8	0.3

uid	name	age	pop
142	Bart	10	0.9
857	Lisa	8	0.7
123	Milhouse	10	0.2
456	Ralph	8	0.3

avg_pop
0.525

38

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

39

Examples of invalid queries

- `SELECT uid, age FROM User GROUP BY age;` WRONG!
  - Recall there is one output row per group
  - There can be multiple *uid* values per group
- `SELECT uid, MAX(pop) FROM User;` WRONG!
  - Recall there is only one group for an aggregate query with no GROUP BY clause
  - There can be multiple *uid* values
  - Wishful thinking (that the output *uid* value is the one associated with the highest popularity) does NOT work

☞Another way of writing the "most popular" query?

40

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 (π) for each group that passes HAVING

41

HAVING examples

- List the average popularity for each age group with more than a hundred users
  - ```
    SELECT age, AVG(pop)
    FROM User
    GROUP BY age
    HAVING COUNT(*) > 100;
    ```
  - Can be written using WHERE and table subqueries
- 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

42

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

43

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

44

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;

45

SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Subqueries
- Aggregation and grouping
- Ordering

☞Next: NULL's, outerjoins, data modification, constraints, …