(More) SQL

Introduction to Databases
CompSci 316 Spring 2020

Announcements (Tue. Feb. 4)

• HW3 posted (all questions now)
  • Due dates: Q1-Q3: Tuesday Feb 11 11:59 pm
  • Q4-Q5: Thursday Feb 13 11:59 pm
  • Many parts, keep working on it!

• Please form your groups by this Thursday Feb 6
  • So that we can help you find a group if needed well before MS1 is due
  • Project formation spreadsheet shared

• 5 members for standard projects please! (otherwise we may have to shuffle later, better if you do it yourself)
• If you want to do an open project, let me know asap

Recap: Basic SQL from Lecture 1-2

• Find addresses of all bars that ‘Dan’ frequents

  • SELECT B.address
    FROM Bar B, Frequents F
    WHERE B.name = F.bar
    AND F.drinker = 'Dan'

We discussed
• SELECT-FROM-WHERE
• DISTINCT
• ORDER BY
• Bag vs. Set semantics (why bag?)
• Semantic of SQL evaluation (?)

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 $\cup$, $\setminus$, 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

Examples of bag operations

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

Examples of set versus bag operations

Poke (uid1, uid2, timestamp)

• (SELECT uid1 FROM Poke)
  EXCEPT
  (SELECT uid2 FROM Poke);

What do these queries return?
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;

Poke

IN subqueries

- x IN (subquery) checks if x is in the result of subquery
- Example: users (all columns) at the same age as (some) Bart

Let's first try without sub-queries

You can use NOT IN too

Semantics of subqueries

- SELECT *
  FROM User 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?)

"WITH" clause – very useful!

- You will find "WITH" clause very useful!
  WITH Temp1 AS
    (SELECT .......
     Temp2 AS
    (SELECT .......
      SELECT X, Y
      FROM TEMP1, TEMP2
      WHERE ....

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

WITH BartAge AS
    (SELECT age
     FROM User
     WHERE name = 'Bart')
    SELECT U.uid, U.name, U.age, U.pop
    FROM User U, BartAge B
    WHERE U.age = B.age

WITH clause not really needed for this query!
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

  ```sql
  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 (later)

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

• 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));

• What does this query return?

Quantified subqueries

• A quantified subquery can be used syntactically 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 subquery result 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 users are the most popular?

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

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

  • Use NOT to negate a condition

More ways to get the most popular

• Which users are the most popular?
Aggregates

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

• Example: number of users under 18, and their average popularity
  
  ```sql
  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?
  
  ```sql
  SELECT COUNT(DISTINCT uid)
  FROM Member;
  ```

  is equivalent to:

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

Grouping

• SELECT ... FROM ... WHERE ...
  GROUP BY list of columns;

• Example: compute average popularity for each age group
  
  ```sql
  SELECT age, AVG(pop)
  FROM User
  GROUP BY age;
  ```

Semantics of GROUP BY

SELECT ... FROM ... WHERE ...
GROUP BY ...;

• Compute FROM (\(\times\))
• Compute WHERE (\(\sigma\))
• Compute GROUP BY: group rows according to the values of GROUP BY columns
• 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

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

Compute SELECT for each group
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?

Examples on blackboard

Examples of invalid queries

- SELECT uid, age
  FROM User GROUP BY age;

- SELECT uid, MAX(pop) FROM User;

Which one is correct?

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 (x)
- 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

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 sub-queries

- 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

Next: incomplete information and nulls!
Incomplete information

- Example: User (uid, name, age, pop)
- Value unknown
  - We do not know Nelson's age
- Value not applicable
  - Suppose pop is based on interactions with others on our social networking site
  - Nelson is new to our site; what is his pop?

Solution 1

Solution 2

Solution 3

SQL’s solution

- A special value NULL
  - For every domain
  - Special rules for dealing with NULL’s

- Example: User (uid, name, age, pop)
  - (789, “Nelson”, NULL, NULL)

Computing with NULL’s

- When we operate on a NULL and another value (including another NULL) using +, −, etc., the result is NULL

- Aggregate functions ignore NULL, except COUNT(*) (since it counts rows)
Three-valued logic

- \( \text{TRUE} = 1, \text{FALSE} = 0, \text{UNKNOWN} = 0.5 \)
- \( x \text{ AND } y = \min(x, y) \)
- \( x \text{ OR } y = \max(x, y) \)
- \( \text{NOT } x = 1 - x \)
- When we compare a NULL with another value (including another NULL) using =, >, etc., the result is UNKNOWN
- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
  - UNKNOWN is not enough

Unfortunate consequences

- SELECT AVG(pop) FROM User;
  - SELECT SUM(pop)/COUNT(*) FROM User;
- \( x \text{ AND } y \) = \( \min(x, y) \)
- \( x \text{ OR } y \) = \( \max(x, y) \)
- \( \text{NOT } x \) = \( 1 - x \)
- When we compare a NULL with another value (including another NULL) using =, >, etc., the result is UNKNOWN
- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
  - UNKNOWN is not enough

Other problem

- Example: Who has NULL pop values?
  - SELECT * FROM User WHERE pop = NULL;
  - (SELECT * FROM User) EXCEPT ALL (SELECT * FROM User WHERE pop = pop);
- SQL introduced special built-in predicates IS NULL and IS NOT NULL
  - SELECT * FROM User WHERE pop IS NULL;

Outerjoin motivation

- Example: a master group membership list
  - SELECT g.gid, g.name AS gname, u.uid, u.name AS uname FROM Group g, Member m, User u WHERE g.gid = m.gid AND m.uid = u.uid;
- What if a group is empty?
  - It may be reasonable for the master list to include empty groups as well
    - For these groups, uid and uname columns would be NULL

Outerjoin flavors and definitions

- A full outerjoin between R and S (denoted \( R \bowtie S \)) includes all rows in the result of \( R \bowtie S \), plus
  - “Dangling” \( R \) rows (those that do not join with any \( S \) rows) padded with NULL’s for \( S \)'s columns
  - “Dangling” \( S \) rows (those that do not join with any \( R \) rows) padded with NULL’s for \( R \)'s columns
- A left outerjoin \( (R \bowtie S) \) includes rows in \( R \bowtie S \) plus dangling \( R \) rows padded with NULL’s
- A right outerjoin \( (R \bowtie S) \) includes rows in \( R \bowtie S \) plus dangling \( S \) rows padded with NULL’s

Outerjoin examples

- SELECT * FROM User;
  - SELECT * FROM User WHERE pop = pop;
- SELECT AVG(pop) FROM User;
  - SELECT SUM(pop)/COUNT(*) FROM User;
  - \( \text{AVG}(pop) = \text{SUM}(pop)/\text{COUNT}(pop) \)

Be careful: NULL breaks many equivalences

Another problem

- Example: Who has NULL pop values?
  - SELECT * FROM User WHERE pop = NULL;
  - (SELECT * FROM User) EXCEPT ALL (SELECT * FROM User WHERE pop = pop);
- SQL introduced special built-in predicates IS NULL and IS NOT NULL
  - SELECT * FROM User WHERE pop IS NULL;

Outerjoin examples

- SELECT AVG(pop) FROM User;
  - SELECT SUM(pop)/COUNT(*) FROM User;
  - \( \text{AVG}(pop) = \text{SUM}(pop)/\text{COUNT}(pop) \)

Be careful: NULL breaks many equivalences
Outerjoin syntax

• SELECT * FROM Group LEFT OUTER JOIN Member
  ON Group.gid = Member.gid;
  = Group.[group_gid]Member.[member_gid]
• SELECT * FROM Group RIGHT OUTER JOIN Member
  ON Group.gid = Member.gid;
  = Group.[group_gid]Member.[member_gid]
• SELECT * FROM Group FULL OUTER JOIN Member
  ON Group.gid = Member.gid;
  = Group.[group_gid]Member.[member_gid]

\[\text{A similar construct exists for regular ("inner") joins:}\]
• SELECT * FROM Group JOIN Member
  ON Group.gid = Member.gid;

\[\text{These are theta joins rather than natural joins}\]
• Return all columns in Group and Member

\[\text{For natural joins, add keyword NATURAL; don't use ON}\]

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...).

INSERT

• Insert one row
  INSERT INTO Member VALUES (789, 'dps');
  -- User 789 joins Dead Putting Society
• Insert the result of a query
  INSERT INTO Member
  (SELECT uid, 'dps' FROM User
   WHERE uid NOT IN (SELECT
     uid FROM Member
     WHERE gid = 'dps'));
  -- Everybody joins Dead Putting Society!

DELETE

• Delete everything from a table
  DELETE FROM Member;
• Delete according to a WHERE condition
  Example: User 789 leaves Dead Putting Society
  DELETE FROM Member
  WHERE uid = 789 AND gid = 'dps';
  Example: Users under age 18 must be removed from United Nuclear Workers
  DELETE FROM Member
  WHERE uid IN (SELECT uid FROM User
    WHERE age < 18)
    AND gid = 'nuk';

UPDATE

• Example: User 142 changes name to “Barney”
  UPDATE User
  SET name = 'Barney'
  WHERE uid = 142;
• Example: We are all popular!
  UPDATE User
  SET pop = (SELECT AVG(pop) FROM User);
  -- But won't update of every row causes average pop to change?
  -- Subquery is always computed over the old table
Constraints

• Restrictions on allowable data in a database
  • In addition to the simple structure and type restrictions imposed by the table definitions
  • Declared as part of the schema
  • Enforced by the DBMS
• Why use constraints?
  • Protect data integrity (catch errors)
  • Tell the DBMS about the data (so it can optimize better)

Types of SQL constraints

• NOT NULL
• Key
• Referential integrity (foreign key)
• General assertion
• Tuple- and attribute-based CHECK’s

NOT NULL constraint examples

• CREATE TABLE User
  (uid INTEGER NOT NULL,
   name VARCHAR(30) NOT NULL,
   twitterid VARCHAR(15) NOT NULL,
   age INTEGER,
   pop FLOAT);

• CREATE TABLE Group
  (gid CHAR(10) NOT NULL,
   name VARCHAR(100) NOT NULL);

• CREATE TABLE Member
  (uid INTEGER NOT NULL,
   gid CHAR(10) NOT NULL);

Key declaration examples

• CREATE TABLE User
  (uid INTEGER NOT NULL PRIMARY KEY,
   name VARCHAR(30) NOT NULL,
   twitterid VARCHAR(15) NOT NULL UNIQUE,
   age INTEGER,
   pop FLOAT);

• CREATE TABLE Group
  (gid CHAR(10) NOT NULL PRIMARY KEY,
   name VARCHAR(100) NOT NULL);

• CREATE TABLE Member
  (uid INTEGER NOT NULL,
   gid CHAR(10) NOT NULL,
   PRIMARY KEY(uid, gid));

This form is required for multi-attribute keys
Referential integrity example

- `Member.uid` references `User.uid`
  - If an `uid` appears in `Member`, it must appear in `User`
- `Member.gid` references `Group.gid`
  - If a `gid` appears in `Member`, it must appear in `Group`

* That is, no “dangling pointers”

```
<table>
<thead>
<tr>
<th>User</th>
<th>Member</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>122</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>456</td>
</tr>
</tbody>
</table>
```

Referential integrity in SQL

- Referenced column(s) must be PRIMARY KEY
- Referencing column(s) form a FOREIGN KEY

Example

```sql
CREATE TABLE Member
(uid INTEGER NOT NULL REFERENCES User(uid),
gid CHAR(10) NOT NULL,
PRIMARY KEY(uid, gid),
FOREIGN KEY (gid) REFERENCES Group(gid));
```

Deferred constraint checking

- No-chicken-no-egg problem
- `CREATE TABLE Dept
  (name CHAR(20) NOT NULL PRIMARY KEY,
  chair CHAR(30) NOT NULL REFERENCES Prof(name));
CREATE TABLE Prof
(name CHAR(30) NOT NULL PRIMARY KEY,
department CHAR(20) NOT NULL REFERENCES Dept(name));`
  - The first `INSERT` will always violate a constraint!
- Deferred constraint checking is necessary
  - Check only at the end of a transaction
  - Allowed in SQL as an option
  - Curious how the schema was created in the first place?
    - `ALTER TABLE ADD CONSTRAINT` (read the manual!)

General assertion

- `CREATE ASSERTION assertion_name
  CHECK assertion_condition;`
  - `assertion_condition` is checked for each modification that could potentially violate it
- Example: `Member.uid` references `User.uid`
  - `CREATE ASSERTION MemberUserRefIntegrity
    CHECK (NOT EXISTS
    (SELECT * FROM Member
    WHERE uid NOT IN
    (SELECT uid FROM User)));`
  - In SQL3, but not all (perhaps no) DBMS supports it

Tuple- and attribute-based CHECK’s

- Associated with a single table
- Only checked when a tuple/attribute is inserted/updated
  - Reject if condition evaluates to FALSE
  - TRUE and UNKNOWN are fine
- Examples:
  - `CREATE TABLE User(...
    age INTEGER CHECK(age IS NULL OR age > 0),
    ...);`
  - `CREATE TABLE Member
    (uid INTEGER NOT NULL,
    CHECK (uid IN (SELECT uid FROM User)),
    ...);`
    - Is it a referential integrity constraint?
    - Not quite; not checked when User is modified
SQL features covered so far

• Query
  • SELECT-FROM-WHERE statements
  • Set and bag operations
  • Table expressions, subqueries
  • Aggregation and grouping
  • Ordering
  • Outerjoins

• Modification
  • INSERT/DELETE/UPDATE

• Constraints

Next: triggers, views, indexes