

SUBQUERIES AND VIEWS

CS121: Relational Databases
Fall 2018 – Lecture 6

String Comparisons and GROUP BY

2

- Last time, introduced many advanced features of SQL, including **GROUP BY**
- Recall: string comparisons using `=` are *case-insensitive* by default

```
SELECT 'HELLO' = 'hello'; -- Evaluates to true
```
- This can also cause unexpected results with SQL grouping and aggregation
- Example: table of people's favorite colors
 - ▣

```
CREATE TABLE favorite_colors (  
    name    VARCHAR(30) PRIMARY KEY,  
    color  VARCHAR(30)  
);
```

String Compares and GROUP BY (2)

3

- Add data to our table:

```
INSERT INTO favorite_colors VALUES ('Alice', 'BLUE');  
INSERT INTO favorite_colors VALUES ('Bob', 'Red');  
INSERT INTO favorite_colors VALUES ('Clara', 'blue');  
...
```

- How many people like each color?
 - ▣ `SELECT color, COUNT(*) num_people
FROM favorite_colors GROUP BY color;`
 - ▣ Even though “BLUE” and “blue” differ in case, they will still end up in the same group!

Null Values in SQL

4

- Like relational algebra, SQL represents missing information with *null* values
 - ▣ **NULL** is a keyword in SQL
 - ▣ Typically written in all-caps
- Use **IS NULL** and **IS NOT NULL** to check for *null* values
 - ▣ **attr = NULL** is *never* true! (It is *unknown*.)
 - ▣ **attr <> NULL** is also *never* true! (Also *unknown*.)
 - ▣ Instead, write: **attr IS NULL**
- Aggregate operations ignore **NULL** input values
 - ▣ **COUNT** returns 0 for an empty input multiset
 - ▣ All others return **NULL** for an empty input (even **SUM** !)

Comparisons and Unknowns

5

- Relational algebra introduced the *unknown* truth-value
 - ▣ Produced by comparisons with *null*
- SQL also has tests for *unknown* values
 - comp* IS UNKNOWN
 - comp* IS NOT UNKNOWN
 - ▣ *comp* is some comparison operation

NULL in Inserts and Updates

6

- Can specify **NULL** values in **INSERT** and **UPDATE** statements

```
INSERT INTO account  
VALUES ('A-315', NULL, 500);
```

- ▣ Can clearly lead to some problems...
- ▣ Primary key attributes are not allowed to have **NULL** values
- ▣ Other ways to specify constraints on **NULL** values for specific attributes

Additional Join Operations

7

- SQL-92 introduces additional join operations
 - ▣ natural joins
 - ▣ left/right/full outer joins
 - ▣ theta joins
- Syntax varies from the basic “Cartesian product” join syntax
 - ▣ All changes are in **FROM** clause
 - ▣ Varying levels of syntactic sugar...

Theta Join

8

- One relational algebra operation we skipped
- Theta join is a generalized join operation
 - ▣ Sometimes called a “condition join”
- Written as: $r \bowtie_{\theta} s$
- Abbreviation for: $\sigma_{\theta}(r \times s)$
- Doesn't include project operation like natural join and outer joins do
- No *null*-padded results, like outer joins have

SQL Theta Joins

9

- SQL provides a syntax for theta joins
- Example:

Associate customers and loan balances

```
SELECT * FROM borrower INNER JOIN loan ON  
    borrower.loan_number = loan.loan_number;
```

- ▣ Result:

customer_name	loan_number	loan_number	branch_name	amount
Smith	L-11	L-11	Round Hill	900.00
Jackson	L-14	L-14	Downtown	1500.00
Hayes	L-15	L-15	Perryridge	1500.00
Adams	L-16	L-16	Perryridge	1300.00
Jones	L-17	L-17	Downtown	1000.00
...

SQL Theta Joins (2)

10

- Syntax in **FROM** clause:

```
table1 INNER JOIN table2 ON condition
```

- ▣ **INNER** is optional; just distinguishes from outer joins

- No duplicate attribute names are removed

- ▣ Can specify relation name, attribute names

```
table1 INNER JOIN table2 ON condition  
AS rel (attr1, attr2, ...)
```

- Very similar to a derived relation

Theta Joins on Multiple Tables

11

- Can join across multiple tables with this syntax
- Example: join customer, borrower, loan tables

- ▣ Nested theta-joins:

```
SELECT * FROM customer AS c
      JOIN borrower AS b ON
          c.customer_name = b.customer_name
      JOIN loan AS l ON
          b.loan_number = l.loan_number;
```

- ▣ Generally evaluated left to right
- ▣ Can use parentheses to specify join order
- ▣ Order usually doesn't affect results or performance
(if outer joins are involved, results can definitely change)

Theta Joins on Multiple Tables (2)

12

Join customer, borrower, loan tables: take 2

- ▣ One Cartesian product and one theta join:

```
SELECT * FROM customer AS c
      JOIN borrower AS b JOIN loan AS l
      ON c.customer_name = b.customer_name
         AND b.loan_number = l.loan_number;
```

- ▣ Database will optimize this anyway, but it really isn't two theta joins

Join Conditions

13

- Can specify *any* condition (including nested subqueries) in **ON** clause
 - ▣ Even conditions that aren't related to join itself

- Guideline:
 - ▣ Use **ON** clause for join conditions
 - ▣ Use **WHERE** clause for selecting rows
 - ▣ Mixing the two can cause lots of confusion!

Cartesian Products

14

- Cartesian product can be specified as **CROSS JOIN**
 - ▣ Can't specify an **ON** condition for a **CROSS JOIN**
- Cartesian product of *borrower* and *loan*:
 - `SELECT * FROM borrower CROSS JOIN loan;`
 - ▣ Same as a theta join with no condition:
 - `SELECT * FROM borrower INNER JOIN loan
ON TRUE;`
 - ▣ Or, simply:
 - `SELECT * FROM borrower JOIN loan;`
 - `SELECT * FROM borrower, loan;`

Outer Joins

15

- Can specify outer joins in SQL as well:

```
SELECT * FROM table1  
LEFT OUTER JOIN table2 ON condition;
```

```
SELECT * FROM table1  
RIGHT OUTER JOIN table2 ON condition;
```

```
SELECT * FROM table1  
FULL OUTER JOIN table2 ON condition;
```

- OUTER is implied by LEFT/RIGHT/FULL, and can therefore be left out

```
SELECT * FROM table1 LEFT JOIN table2 ON  
condition;
```

Common Attributes

16

- **ON** syntax is clumsy for simple joins
 - Also, it's tempting to include conditions that should be in the **WHERE** clause
- Often, schemas are designed such that join columns have the same names
 - e.g. *borrower.loan_number* and *loan.loan_number*
- **USING** clause is a simplified form of **ON**

```
SELECT * FROM t1 LEFT OUTER JOIN t2
  USING (a1, a2, ...);
```

 - *Roughly* equivalent to:

```
SELECT * FROM t1 LEFT OUTER JOIN t2
  ON (t1.a1 = t2.a1 AND t1.a2 = t2.a2 AND ...);
```

Common Attributes (2)

17

- **USING** also eliminates duplicate join attributes
 - ▣ Result of join with **USING (a1, a2, ...)** will only have one instance of each join column in the result
 - ▣ This is fine, because **USING** requires equal values for the specified attributes
- Example: tables $r(a, b, c)$ and $s(a, b, d)$
 - ▣ **SELECT * FROM r JOIN s USING (a)**
 - ▣ Result schema is: $(a, r.b, r.c, s.b, s.d)$
- Can use **USING** clause with **INNER / OUTER** joins
 - ▣ No condition allowed for **CROSS JOIN**

Natural Joins

18

- SQL natural join operation:
 - `SELECT * FROM t1 NATURAL INNER JOIN t2;`
 - **INNER** is optional, as usual
 - No **ON** or **USING** clause is specified
- *All* common attributes are used in natural join operation
 - To join on a *subset* of common attributes, use a regular **INNER JOIN**, with a **USING** clause

Natural Join Example

19

Join borrower and loan relations:

```
SELECT * FROM borrower NATURAL JOIN loan;
```

□ Result:

loan_number	customer_name	branch_name	amount
L-11	Smith	Round Hill	900.00
L-14	Jackson	Downtown	1500.00
L-15	Hayes	Perryridge	1500.00
L-16	Adams	Perryridge	1300.00
L-17	Jones	Downtown	1000.00
L-17	Williams	Downtown	1000.00
L-20	McBride	North Town	7500.00
L-21	Smith	Central	570.00
L-23	Smith	Redwood	2000.00
L-93	Curry	Mianus	500.00

□ Could also use inner join, **USING (loan_number)**

Natural Outer Joins

20

- Can also specify natural outer joins
 - ▣ **NATURAL** specifies how the rows/columns are matched
 - ▣ All overlapping columns are used for join operation
 - ▣ Unmatched tuples from (left, right, or both) tables are **NULL**-padded and included in result

- Example:

```
SELECT * FROM customer
    NATURAL LEFT OUTER JOIN borrower;
```

```
SELECT * FROM customer
    NATURAL LEFT JOIN borrower;
```

Outer Joins and Aggregates

21

- Outer joins can generate **NULL** values
- Aggregate functions ignore **NULL** values
 - ▣ **COUNT** has most useful behavior!
- Example:
 - ▣ Find out how many loans each customer has
 - ▣ Include customers with *no* loans; show 0 for those customers
 - ▣ Need to use *customer* and *borrower* tables
 - ▣ Need to use an outer join to include customers with no loans

Outer Joins and Aggregates (2)

22

- First step: left outer join customer and borrower tables

```
SELECT customer_name, loan_number
FROM customer LEFT OUTER JOIN borrower
      USING (customer_name);
```

- Generates result:

- ▣ Customers with no loans have **NULL** for *loan_number* attribute

customer_name	loan_number
Adams	L-16
Brooks	NULL
Curry	L-93
Glenn	NULL
Green	NULL
Hayes	L-15
...	

Outer Joins and Aggregates (3)

23

- Finally, need to count number of accounts for each customer
 - ▣ Use grouping and aggregation for this
 - ▣ Grouping, aggregation is applied to *results* of **FROM** clause; won't interfere with join operation
- What's the difference between **COUNT (*)** and **COUNT (loan_number)** ?
 - ▣ **COUNT (*)** simply counts number of tuples in each group
 - ▣ **COUNT (*)** won't produce any counts of 0!
 - ▣ **COUNT (loan_number)** is what we want

Outer Joins and Aggregates (4)

24

- Final query:

```
SELECT customer_name,  
       COUNT(loan_number) AS num_loans  
FROM customer LEFT OUTER JOIN borrower  
       USING (customer_name)  
GROUP BY customer_name  
ORDER BY COUNT(loan_number) DESC;
```

- ▣ Sort by count, just to make it easier to analyze

customer_name	num_loans
Smith	3
Jones	1
Curry	1
McBride	1
Hayes	1
Jackson	1
Williams	1
Adams	1
Brooks	0
Lindsay	0
...	

Views

25

- So far, have used SQL at logical level
 - ▣ Queries generally use actual relations
 - ▣ ...but they don't need to!
 - ▣ Can also write queries against derived relations
 - Nested subqueries or **JOINS** in **FROM** clause
- SQL also provides view-level operations
- Can define views of the logical model
 - ▣ Can write queries directly against views

Why Views?

26

- Two main reasons for using views
- Reason 1: Performance and convenience
 - ▣ Define a view for a widely used derived relation
 - ▣ Write simple queries against the view
 - ▣ DBMS automatically computes view's contents when it is used in a query
- Some databases provide materialized views
 - ▣ View's result is pre-computed and stored on disk
 - ▣ DBMS ensures that view is "up to date"
 - Might update view's contents immediately, or periodically

Why Views? (2)

27

- Reason 2: Security!
 - ▣ Can specify access constraints on both tables and views
 - ▣ Can specify strict access constraints on a table with sensitive information
 - ▣ Can provide a view that excludes sensitive information, with more lenient access
- Example: employee information database
 - ▣ Logical-level tables might have SSN, salary info, other private information
 - ▣ An “employee directory” view could limit this down to employee name and professional contact information

Creating a View

28

- SQL syntax for creating a view is very simple
 - ▣ Based on **SELECT** syntax, as always

```
CREATE VIEW viewname AS select_stmt;
```
 - ▣ View's columns are columns in **SELECT** statement
 - ▣ Column names must be unique, just like any table's columns
 - ▣ Can specify view columns in **CREATE VIEW** syntax:

```
CREATE VIEW viewname (attr1, attr2, ...) AS  
select_stmt;
```
- Even easier to remove:

```
DROP VIEW viewname;
```

Example View

29

- Create a view that shows *total* account balance of each customer.
 - The **SELECT** statement would be:

```
SELECT customer_name,  
       SUM(balance) AS total_balance  
FROM depositor NATURAL JOIN account  
GROUP BY customer_name;
```
 - The view is just as simple:

```
CREATE VIEW customer_deposits AS  
  SELECT customer_name,  
         SUM(balance) AS total_balance  
  FROM depositor NATURAL JOIN account  
  GROUP BY customer_name;
```
- With views, good attribute names are a *must*.

Updating a View?

30

- A view is a derived relation...
- What to do if an **INSERT** or **UPDATE** refers to a view?
- One simple solution: Don't allow it! 😊
- Could also allow the database designer to specify what operations to perform when a modification is attempted against a view
 - ▣ Very flexible approach
 - ▣ Default is still to forbid updates to views

Updatable Views

31

- Can actually define updates for certain kinds of views
- A view is updatable if:
 - ▣ The **FROM** clause only uses one relation
 - ▣ The **SELECT** clause only uses attributes in the relation, and doesn't perform any computations
 - ▣ Attributes not listed in the **SELECT** clause can be set to **NULL**
 - ▣ The view's query doesn't perform any grouping or aggregation
- In these cases, **INSERTs**, **UPDATEs**, and **DELETEs** can be performed

Updatable Views (2)

32

□ Example view:

- All accounts at Downtown branch.

```
CREATE VIEW downtown_accounts AS
  SELECT account_number, branch_name, balance
  FROM account WHERE branch_name='Downtown';
```

□ Is this view updatable?

- **FROM** uses only one relation
- **SELECT** includes all attributes from the relation
- No computations, aggregates, distinct values, etc.
- Yes, it is updatable!

Updatable Views?

33

- Issue a query against the view:

```
SELECT * FROM downtown_accounts;
```

account_number	branch_name	balance
A-101	Downtown	500.00

- Insert a new tuple:

```
INSERT INTO downtown_accounts  
VALUES ('A-600', 'Mianus', 550);
```

- Look at the view again:

```
SELECT * FROM downtown_accounts;
```

account_number	branch_name	balance
A-101	Downtown	500.00

- ▣ Where's my tuple?!

Checking Inserted Rows

34

- Can add **WITH CHECK OPTION** to the view declaration
 - ▣ Inserted rows are checked against the view's **WHERE** clause
 - ▣ If a row doesn't satisfy the **WHERE** clause, it is rejected
- Updated view definition:

```
CREATE VIEW downtown_accounts AS
  SELECT account_number, branch_name, balance
  FROM account WHERE branch_name='Downtown'
WITH CHECK OPTION;
```