Data 101: Data Engineering Note 2 – September 8, 2022

Views, Subqueries, and Aggregation

Lecturer: Lakshya Jain *Scribe: Shreya Shankar*

1 Introduction

In the last note, we introduced some basic SQL syntax in the `SELECT FROM WHERE` format. In this note, we'll discuss some more advanced SQL commands.

2 Views

Suppose we want the result of a SQL query to act as a new table that we can query. For example, the following query turns a SELECT statement into another table:

```
CREATE TABLE CitationStops AS
(SELECT gender, citation
FROM Stops
WHERE citation = True)
```

After running this command, we now can query `CitationStops` directly. But if the **base table**, or the `Stops` table changes (e.g., gets new records), we have to recreate `CitationStops` to reflect the new changes!

To avoid manually recreating derived tables, we can instead define a **view**, as follows:

```
CREATE VIEW CitationStops AS
(SELECT gender, citation
FROM Stops
WHERE citation = True)
```

In views (also known as virtual views), outputs are not stored, and any time a user queries the view (i.e., `CitationStops`), the output is computed on demand. Think of this as a variable or as a virtual relation that is more convenient to query than the base table.

Sometimes, we may only want to create a view and query it once—as part of breaking down a larger SQL query. We can define an inline view or **common table expression** (CTE), as follows:

```
WITH CitationStops AS
(SELECT gender, citation
FROM Stops
WHERE citation = True)
SELECT * FROM CitationStops
```

Like virtual views, CTEs are computed on demand. But unlike virtual views, their lifetime is restricted to the query.

On the flip side, suppose we want to create a view and query it a lot. The naive method of creating a view might be slow because the view is computed on demand every time a query is run. To solve this issue, we can define a **materialized view**, as follows:

```
CREATE MATERIALIZED VIEW CitationStops AS
(SELECT gender, citation
FROM Stops
WHERE citation = True)
```

In materialized views, outputs are stored just like they are in regular tables (but not like in virtual views). Materialized views are typically automatically updated as base tables change, but since they must be rematerialized frequently, they might add unnecessary overhead to base table updates. So, data engineers (you!) must be thoughtful about what views to keep materialized and what views to keep virtual.

3 Subqueries

A parenthesized SQL query statement (a **subquery**) can be used as a value in various places of a larger SQL query. We could use subqueries as scalars or sets.

3.1 Subquery as a Scalar

If a subquery returns a single tuple with a single attribute value, it can be treated as a scalar in expressions. Suppose we want to collect all the stops that happened at the same location as `id = 123`. We could issue a query as follows, with the parethesized statement as the subquery:

```
SELECT S1.id, S1.race, S1.location, S1.arrest
FROM Stops S1
WHERE S1.location = (SELECT S2.location FROM Stops S2 WHERE S2.id = 123)
```

Note that when using subqueries, it's important to define relations with relevant variables (i.e., S1 and S2) such that when accessing attributes, it's clear which relation's attribute is being accessed. We could also rewrite this query to use a CTE (i.e., `WITH` statement) instead of a subquery—this exercise is left to the reader.

3.2 Subquery as a Set

We can use the `EXISTS` or `NOT EXISTS` keywords in `WHERE` clauses to use results of a subquery in set form. For example, suppose we want to determine all the Stops that don't have a corresponding zipcode in Zips:

```
SELECT Stops.location FROM Stops
WHERE NOT EXISTS
(SELECT * FROM Zips WHERE Zips.location = Stops.location)
```

In the above query, the subquery returns a set of all zips that have a corresponding location in Seps, which is then used with the `NOT EXISTS` keyword to return locations of stops that do not have an entry in the subquery set. There are other useful keywords to use in multiset operations:

- (subquery) `UNION ALL` (subquery)
- (subquery) `EXCEPT ALL` (subquery)
- (subquery) `INTERSECT ALL` (subquery)

Refer to the lecture for more information on these operations.

4 Aggregations

We can also **aggregate** a column in a `SELECT` clause, such as perform `SUM`, `MIN`, `MAX`, `AVG`, or `COUNT`. `COUNT(*)` is a special syntax to count the number of tuples, for example:

```
SELECT COUNT(*) FROM Stops
```

returns the number of tuples in the Stops relation. Similarly, `SELECT MAX(age), AVG (age) FROM Stops` selects the min and max age from the stops relation.

4.1 Handling NULLs

Tuples can have NULL values for attributes, which we need to take note of when performing queries. Generally, NULLs do not satisfy conditions—for example, if a tuple value is NULL, `age > 40` and `age <= 40` will both evaluate to `FALSE`. This leads to some unintuitive behavior, for example:

```
SELECT * FROM Stops WHERE age > 40 OR age <= 40
```

will return all tuples that don't have a NULL age value, not all the tuples in the relation! If we want all the tuples, we need to explicitly test for NULL:

```
SELECT * FROM Stops WHERE age > 40 OR age <= 40 OR age IS NULL
```

For aggregations, NULL values are not involved. For example, the average of a column will be the average of all the non-null values in that column. However, if all the values in the column are NULL, the aggregation will also return NULL.

4.2 Grouping

In some cases, we may want to compute an aggregate for each "group" of tuples as opposed to an overall COUNT, MAX or SUM. To do so, we add a `GROUP BY` clause after the SELECT-FROM-WHERE. For example, say we wanted to find average and minimum ages for each location:

```
SELECT location, AVG(age) AS avgage, MIN (age) as minage
FROM Stops
GROUP BY location
```

will return the intended result. Note that if aggregation is used, then each element of the SELECT clause must either be an aggregate or an attribute in the GROUP BY list. This is because if an attribute is not being aggregated or being grouped, we have no way to "squish" the values down per group.

We can also use more sophisticated syntax in GROUP BYs; for example, the following query returns race and median age:

```
SELECT race, PERCENTAGE_DISC(0.5)
WITHIN GROUP (ORDER BY age)
FROM Stops
```

Note the usage of ORDER BY, which enforces an ordering of the result. The syntax of an ORDER BY clause is `ORDER BY <attr> ASC | DESC`. Refer to the lecture for more information and GROUP BY functions.

4.3 HAVING Clauses

Suppose we want to filter a GROUP BY on some condition. We can use a HAVING clause, which typically precedes a GROUP BY. The HAVING condition is applied to each group, and groups not satisfying the condition are eliminated. For example, say we wanted to compute the locations with at least 30 stops:

```
SELECT location, COUNT (*)
FROM Stops
GROUP BY location
HAVING COUNT (*) > 30
```

Similarly to SELECT clauses, each attribute mentioned in a HAVING clause must either be part of the GROUP BY or be aggregated.

4.4 LIMIT and OFFSET

Sometimes we want to limit the result to a few tuples via `LIMIT <val>`. For example, say we want to order Stops by descending age, return top 15:

```
SELECT *
FROM Stops
ORDER BY age DESC LIMIT 15;
```

Sometimes we want to start the output at a particular point via `OFFSET <val>`. For example, say we want to order Stops by descending age, return positions 11 to 15:

```
SELECT *
FROM Stops
ORDER BY age DESC LIMIT 5 OFFSET 10
```

Refer to the lecture for more information on these keywords.