Data 101: Data Engineering Note 3 – September 15, 2022

More SQL, Integrity, and Constraints

Lecturer: Lakshya Jain *Scribe: Shreya Shankar*

1 Introduction

In the last note, we discussed more SQL syntax around views, subqueries, and aggregatien, In this note, we'll cover even more SQL—including windowing, sampling, string manipulation, updating, integrity, and constrains.

2 Window Functions

So far, when we performed aggregation, we did so over non-overlapping groups of tuples. But what if we want to consider overlapping groups or tuples/rows beyond the current row without grouping in our queries? Examples of this include:

- Compute a cumulative sum, not just individual group sums
- Determine the rank within a group
- Normalize a row using a group average

We can use **window functions** to consider rows beyond the current row in our calculations. The syntax requires 3 ingredients:

- PARTITION BY: what key specifies the larger group
- ORDER: how to order rows in the group or partition
- RANGE: how big the window/group/partition should be

The syntax looks like:

```
SELECT id, location, age, AVG(age) OVER (
   PARTITION BY location ORDER BY age RANGE BETWEEN UNBOUNDED PRECEDING AND 1
      PRECEDING
) AS avg_age
FROM Stops
```

In the above query, for each row, we compute the average age with respect to the tuples in the same location with a smaller/preceding age. There are many different window aggregation functions we could use—age, rank, lead, lag are some examples. Check the lecture slides for more.

3 String Manipulation

SQL has many string manipulation functions, such as substring, strpos (starting position of a substring), concatenation, regex_replace, and more. Refer to the lecture slides for examples on how to use these.

4 Sampling

We can also *sample* tuples, or select a subset of tuples. This is desirable in data science—for example, the original dataset might be too large, and you want to experiment quickly before running operations on the full dataset. There are 2 sampling functions discussed in this class: https://www.overleaf.com/project/630c2928dcc8365e71

- `SELECT * FROM R TABLESAMPLE BERNOULLI(percentage p)`: takes a $p\%$ uniform random sample
- `SELECT * FROM R TABLESAMPLE SYSTEM(percentage p)`: tuples are grouped in pages on disk, so $p\%$ of pages are uniformly randomly selected

BERNOULLI is slower due to more random accesses; SYSTEM is faster but less "random."

5 Updates

Suppose we want to update a table—e.g., insert tuples, delete tuples, modify or update existing tuples. The syntax to insert is:

```
-- Inserts multiple values
INSERT INTO STOPS VALUES
(5001, 'Hispanic', 37, 'Rockridge'),
(5002, 'White', 68, 'MacArthur');
```

The syntax to delete is:

```
-- Deletes all rows
DELETE FROM Stops;

-- Deletes some rows
DELETE FROM Stops WHERE age <18;
```

The syntax to update is:

```
-- Set null ages
UPDATE Stops
SET age = 18
WHERE age IS NULL;
```

6 Keys and Constraints

Tuples can have *keys*, which make it easy to identify them in a relation. For example, a table of students could have a student ID key. A set of attributes is said to form a key if no two tuples can have the same values for that combination of attributes. We call this a **PRIMARY KEY**, defined in a table definition:

```
CREATE TABLE Stops(
stopID INTEGER, race VARCHAR(10), location VARCHAR(20),
age INTEGER, arrest BOOLEAN,
PRIMARY KEY (stopID)
);
```

When we define a primary key, we are also defining a **constraint** on its uniqueness. For example, in the above statement, each tuple must have a unique stopID—otherwise when inserting a tuple with a duplicate stopID into the table, we'll get an error. Additionally, no attribute of a PRIMARY KEY can be NULL.

There can be only one PRIMARY KEY (hence the name!). What if we want multiple unique keys? We can use the UNIQUE key syntax:

```
CREATE TABLE Stops(
  stopID INTEGER, personID INTEGER, stopTime TIMESTAMP,
  race VARCHAR(10), location VARCHAR(20),
  age INTEGER, arrest BOOLEAN,
  PRIMARY KEY (stopID),
  UNIQUE (personID, stopTime)
);
```

Here, there are effectively 3 unique keys: stopID, personID, stopTime. UNIQUE keys can be null-valued. We could also declare attributes to have default values or not be null, e.g.,:

```
CREATE TABLE Stops(
  stopID INTEGER, personID INTEGER, stopTime TIMESTAMP,
  race VARCHAR(10), location VARCHAR(20) NOT NULL,
  age INTEGER, arrest BOOLEAN DEFAULT False,
  PRIMARY KEY (stopID),
  UNIQUE (personID, stopTime)
);
```

This gives us, so far, multiple different kinds of constraints—uniqueness via PRIMARY KEY or UNIQUE, and attribute-based constraints (DEFAULT, NOT NULL). The final constraint type we'll cover is FOREIGN KEY, or cross-table integrity. Suppose we have two tables, an Actor table and a Cast_Info table. We'll have actor IDs in the Cast_info table, but we want them to reference the actor IDs in the Actor table. We can declare this constraint as follows:

```
CREATE TABLE Actor (id INTEGER, name TEXT, PRIMARY KEY actor_id);

CREATE TABLE Cast_info (person_id INTEGER, movie_id INTEGER,
  FOREIGN KEY (person_id) REFERENCES Actor (id),
  movie_id INTEGER);
```

With foreign key constraints, if we try to insert a tuple in `Cast_info` that has a `person_id` that isn't an id in the `Actor` table, we'll get an error. Moreover, if we delete a tuple in the `Actor` table that has an id in the `Cast_info` table, we'll also get an error. Refer to the lecture slides for more information on foreign key constraints.