Databases for Data Science

Lecture 07 · 2022-09-19

Today

Starting to wrap up SQL.

- More exercises
- Database programming: stored procedures, functions, triggers
- Indexes and optimization

Working in a Normalized Database

Most of the time, external records (CSV files, etc) will not be normalized.

Inserting into a normalized DB can be tricky.

Importing Normalized Data

Why is this complicated?

- The rows inserted into each table may not be distinct
- New entities may overlap with old entities
- Need to find the foreign keys as we go

Importing Normalized Data

Example: populate `person` and `personal_info`.

- We want to deduplicate `person`s; multiple `personal_info` rows will point to the same person.

Which table do we start with?

Which keys already identify each row?

Step one: collect the input in non-normalized form.

- Each `personal_info` needs both a name and the demographics, so we should join on `arrestees`.

```sql
BEGIN;

CREATE TEMP TABLE import AS (
    SELECT DISTINCT
        a.soid, a.name,
        b.race, b.sex, b.e, b.dob, b.address
    FROM
        arrestees a JOIN bookings b
        ON a.soid=b.soid
);
```

Step two: populate `person`.

- Filter duplicates in input
- Skip anything that would violate the uniqueness constraint

```
INSERT INTO person (soid)
SELECT DISTINCT
    soid
FROM import
ON CONFLICT DO NOTHING;
```

Step three: associate each row with its corresponding `person_id`.

```sql
SELECT
    a.person_id, a.soid,
    b.race, b.sex, b.e, b.dob, b.address
FROM
    person a JOIN import b ON a.soid=b.soid;
```

Step four: populate `personal_info`.

```sql
WITH t1 AS (
    SELECT
        a.person_id, a.soid,
        b.race, b.sex, b.e, b.dob, b.address
    FROM person a JOIN import b ON a.soid=b.soid
)
INSERT INTO personal_info
    (person_id, race, sex, e, dob, address)
SELECT DISTINCT
    person_id, race, sex, e, dob, address
FROM t1
ON CONFLICT DO NOTHING;
COMMIT; -- end transaction
```

- What constraint do we need on `personal_info` for this to work?
    - Is there an alternative?

Importing Normalized Data

How do we find the `personal_info_id` for each booking?

Importing Normalized Data

Group Exercise:

In your group's copy of `mini_homelessness`, migrate all of your data into the new (normalized) tables.

Viewing Normalized Data

Individual exercise:
In your normalized database, write a `SELECT` query that reproduces the original `bookings` table.

Individual exercise:
In your normalized database, write a `SELECT` query to produce the `chargetype`, `charge`, and `releasecode` for every cannabis-related arrest.

Programming in Databases

Sometimes we'll be running the same kind of command many times.

In such cases, it's useful to identify the varying parameters and store the command in a way that is easy to invoke.

Likewise, we may be using the same expression (formula) repeatedly; we'd like to write it down somewhere stable.

Stored Procedures

```sql
CREATE OR REPLACE PROCEDURE insert_student(student_name text)
LANGUAGE PLPGSQL -- This isn't plain SQL!
AS $$

DECLARE -- variable declarations
email TEXT := replace(student_name, ' ', '') || '@ncf.edu';

BEGIN -- start of procedural section

INSERT INTO student(name, email)
VALUES (student_name, email);

END; -- end of procedural section
$$;
```

```sql
CALL insert_student('Xavier Barnes');
```

Stored Procedures

We could instead write scripts in Python or another language.

What are the advantages of stored procedures?

Disadvantages?

Stored Procedures

We could instead write scripts in Python or another language.

What are the advantages of stored procedures?

- May be faster due to caching
- Persists in the DB; can be executed from different contexts

Disadvantages?

- Another complicated language to learn
- Harder to version-control and test
- Other programming languages may be better-suited to a task

Stored procedures

```
CREATE OR REPLACE PROCEDURE insert_student(student_name text)
LANGUAGE PLPGSQL
AS $$
DECLARE email TEXT := replace(student_name, ' ', '') || '@ncf.edu';
BEGIN
    INSERT INTO student(name, email)
    VALUES (student_name, email);
END; -- end of procedural section
$$;
```

PL/pgSQL is a language that extends SQL with procedural programming features.

Exercise: Make your import code into a stored procedure. It should take the filename as its argument.

Functions

Unlike procedures, functions return a value:

```
CREATE FUNCTION increment(a int, b int default 1)
RETURNS INT
AS $$
BEGIN
    RETURN a + b;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT increment(5);
```

Triggers

Triggers run code in response to data being changed.

```
CREATE OR REPLACE FUNCTION archive_student() RETURNS TRIGGER AS $$
    BEGIN
        INSERT INTO alumni (name) SELECT OLD.name;
        RETURN OLD;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER my_trigger
    BEFORE DELETE ON student
    FOR EACH ROW
    EXECUTE FUNCTION archive_student();
```

Triggers

Can happen...

- `before`
- `instead of`
- `after`

...an execution of...

- `insert`
- `update`
- `delete`

Triggers

A trigger can run any user-defined function.

What are some use cases?

Triggers

A trigger can run any user-defined function.

What are some use cases?

- Log any change to a table
- Maintain a summary table
    - (Why not use a view for this?)
- Validate incoming data

Triggers

```
CREATE OR REPLACE FUNCTION archive_student() RETURNS TRIGGER AS $$
    BEGIN
        INSERT INTO alumni (name) SELECT OLD.name;
        RETURN OLD;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER my_trigger
    BEFORE DELETE ON student
    FOR EACH ROW
    EXECUTE FUNCTION archive_student();
```

Individual exercise:

Write a trigger for your `student` table (or something similar) that rejects non-`@ncf.edu` email addresses.

Indexes

PostgreSQL can accelerate our queries by doing some work ahead-of-time.

Indexes are an auxiliary data structure to make certain search operations more efficient.

```sql
CREATE INDEX my_index ON person (name);
```

```sql
CREATE INDEX my_index ON person USING HASH (name);
```

Indexes

Multiple types of index:

- `BTREE` (default) constructs a b-tree.
    - Good for comparisons, e.g. `WHERE value < 500`.
- `HASH` constructs a hash table for the column.
    - Good for equality, e.g. `WHERE school='New College'`
- `GIST`: *Generalized Search Tree*
    - Can handle a broader range of data types than b-trees.

Indexes

What are the downsides to creating an index?

When does an index not help?