CHAPTER 24
Databases and SQL

Memory is man's greatest friend and worst enemy.
—Gilbert Parker

The data you need will often live in *databases*, systems designed for efficiently storing and querying data. The bulk of these are *relational* databases, such as PostgreSQL, MySQL, and SQL Server, which store data in *tables* and are typically queried using Structured Query Language (SQL), a declarative language for manipulating data.

SQL is a pretty essential part of the data scientist's toolkit. In this chapter, we'll create NotQuiteABase, a Python implementation of something that's not quite a database. We'll also cover the basics of SQL while showing how they work in our not-quite database, which is the most "from scratch" way I could think of to help you understand what they're doing. My hope is that solving problems in NotQuiteABase will give you a good sense of how you might solve the same problems using SQL.

CREATE TABLE and INSERT

A relational database is a collection of tables, and of relationships among them. A table is simply a collection of rows, not unlike some of the matrices we've been working with. However, a table also has associated with it a fixed *schema* consisting of column names and column types.

For example, imagine a `users` dataset containing for each user her `user_id`, `name`, and `num_friends`:

```
users = [[0, "Hero", 0],
         [1, "Dunn", 2],
         [2, "Sue", 3],
         [3, "Chi", 3]]
```

In SQL, we might create this table with:

```
CREATE TABLE users (
    user_id INT NOT NULL,
    name VARCHAR(200),
    num_friends INT);
```

Notice that we specified that the `user_id` and `num_friends` must be integers (and that `user_id` isn't allowed to be `NULL`, which indicates a missing value and is sort of like our `None`) and that the name should be a string of length 200 or less. We'll use Python types in a similar way.

> SQL is almost completely case and indentation insensitive. The capitalization and indentation style here is my preferred style. If you start learning SQL, you will surely encounter other examples styled differently.

You can insert the rows with `INSERT` statements:

```
INSERT INTO users (user_id, name, num_friends) VALUES (0, 'Hero', 0);
```

Notice also that SQL statements need to end with semicolons, and that SQL requires single quotes for its strings.

In NotQuiteABase, you'll create a `Table` by specifying a similar schema. Then to insert a row, you'll use the table's `insert` method, which takes a `list` of row values that need to be in the same order as the table's column names.

Behind the scenes, we'll store each row as a `dict` from column names to values. A real database would never use such a space-wasting representation, but doing so will make NotQuiteABase much easier to work with.

We'll implement the NotQuiteABase `Table` as a giant class, which we'll implement one method at a time. Let's start by getting out of the way some imports and type aliases:

```
from typing import Tuple, Sequence, List, Any, Callable, Dict, Iterator
from collections import defaultdict

# A few type aliases we'll use later
Row = Dict[str, Any]                        # A database row
WhereClause = Callable[[Row], bool]         # Predicate for a single row
HavingClause = Callable[[List[Row]], bool]  # Predicate over multiple rows
```

Let's start with the constructor. To create a NotQuiteABase table, we'll need to pass in a list of column names, and a list of column types, just as you would if you were creating a table in a SQL database:

```python
class Table:
    def __init__(self, columns: List[str], types: List[type]) -> None:
        assert len(columns) == len(types), "# of columns must == # of types"

        self.columns = columns         # Names of columns
        self.types = types             # Data types of columns
        self.rows: List[Row] = []      # (no data yet)
```

We'll add a helper method to get the type of a column:

```python
    def col2type(self, col: str) -> type:
        idx = self.columns.index(col)      # Find the index of the column,
        return self.types[idx]             # and return its type.
```

And we'll add an `insert` method that checks that the values you're inserting are valid. In particular, you have to provide the correct number of values, and each has to be the correct type (or `None`):

```python
    def insert(self, values: list) -> None:
        # Check for right # of values
        if len(values) != len(self.types):
            raise ValueError(f"You need to provide {len(self.types)} values")

        # Check for right types of values
        for value, typ3 in zip(values, self.types):
            if not isinstance(value, typ3) and value is not None:
                raise TypeError(f"Expected type {typ3} but got {value}")

        # Add the corresponding dict as a "row"
        self.rows.append(dict(zip(self.columns, values)))
```

In an actual SQL database you'd explicitly specify whether any given column was allowed to contain null (`None`) values; to make our lives simpler we'll just say that any column can.

We'll also introduce a few dunder methods that allow us to treat a table like a `List[Row]`, which we'll mostly use for testing our code:

```python
    def __getitem__(self, idx: int) -> Row:
        return self.rows[idx]

    def __iter__(self) -> Iterator[Row]:
        return iter(self.rows)

    def __len__(self) -> int:
        return len(self.rows)
```

And we'll add a method to pretty-print our table:

```python
    def __repr__(self):
        """Pretty representation of the table: columns then rows"""
        rows = "\n".join(str(row) for row in self.rows)
```

```
        return f"{self.columns}\n{rows}"
```

Now we can create our `Users` table:

```
# Constructor requires column names and types
users = Table(['user_id', 'name', 'num_friends'], [int, str, int])
users.insert([0, "Hero", 0])
users.insert([1, "Dunn", 2])
users.insert([2, "Sue", 3])
users.insert([3, "Chi", 3])
users.insert([4, "Thor", 3])
users.insert([5, "Clive", 2])
users.insert([6, "Hicks", 3])
users.insert([7, "Devin", 2])
users.insert([8, "Kate", 2])
users.insert([9, "Klein", 3])
users.insert([10, "Jen", 1])
```

If you now `print(users)`, you'll see:

```
['user_id', 'name', 'num_friends']
{'user_id': 0, 'name': 'Hero', 'num_friends': 0}
{'user_id': 1, 'name': 'Dunn', 'num_friends': 2}
{'user_id': 2, 'name': 'Sue', 'num_friends': 3}
...
```

The list-like API makes it easy to write tests:

```
assert len(users) == 11
assert users[1]['name'] == 'Dunn'
```

We've got a lot more functionality to add.

UPDATE

Sometimes you need to update the data that's already in the database. For instance, if Dunn acquires another friend, you might need to do this:

```
UPDATE users
SET num_friends = 3
WHERE user_id = 1;
```

The key features are:

- What table to update
- Which rows to update
- Which fields to update
- What their new values should be

We'll add a similar `update` method to NotQuiteABase. Its first argument will be a `dict` whose keys are the columns to update and whose values are the new values for those fields. Its second (optional) argument should be a `predicate` that returns `True` for rows that should be updated, and `False` otherwise:

```
    def update(self,
               updates: Dict[str, Any],
               predicate: WhereClause = lambda row: True):
        # First make sure the updates have valid names and types
        for column, new_value in updates.items():
            if column not in self.columns:
                raise ValueError(f"invalid column: {column}")

            typ3 = self.col2type(column)
            if not isinstance(new_value, typ3) and new_value is not None:
                raise TypeError(f"expected type {typ3}, but got {new_value}")

        # Now update
        for row in self.rows:
            if predicate(row):
                for column, new_value in updates.items():
                    row[column] = new_value
```

after which we can simply do this:

```
assert users[1]['num_friends'] == 2             # Original value

users.update({'num_friends' : 3},               # Set num_friends = 3
             lambda row: row['user_id'] == 1)   # in rows where user_id == 1

assert users[1]['num_friends'] == 3             # Updated value
```

DELETE

There are two ways to delete rows from a table in SQL. The dangerous way deletes every row from a table:

```
DELETE FROM users;
```

The less dangerous way adds a `WHERE` clause and deletes only rows that match a certain condition:

```
DELETE FROM users WHERE user_id = 1;
```

It's easy to add this functionality to our `Table`:

```
    def delete(self, predicate: WhereClause = lambda row: True) -> None:
        """Delete all rows matching predicate"""
        self.rows = [row for row in self.rows if not predicate(row)]
```

If you supply a `predicate` function (i.e., a `WHERE` clause), this deletes only the rows that satisfy it. If you don't supply one, the default `predicate` always returns `True`, and you will delete every row.

For example:

```
# We're not actually going to run these
users.delete(lambda row: row["user_id"] == 1)  # Deletes rows with user_id == 1
users.delete()                                 # Deletes every row
```

SELECT

Typically you don't inspect SQL tables directly. Instead you query them with a `SELECT` statement:

```
SELECT * FROM users;                             -- get the entire contents
SELECT * FROM users LIMIT 2;                     -- get the first two rows
SELECT user_id FROM users;                       -- only get specific columns
SELECT user_id FROM users WHERE name = 'Dunn';  -- only get specific rows
```

You can also use `SELECT` statements to calculate fields:

```
SELECT LENGTH(name) AS name_length FROM users;
```

We'll give our `Table` class a `select` method that returns a new `Table`. The method accepts two optional arguments:

- `keep_columns` specifies the names of the columns you want to keep in the result. If you don't supply it, the result contains all the columns.
- `additional_columns` is a dictionary whose keys are new column names and whose values are functions specifying how to compute the values of the new columns. We'll peek at the type annotations of those functions to figure out the types of the new columns, so the functions will need to have annotated return types.

If you were to supply neither of them, you'd simply get back a copy of the table:

```
    def select(self,
               keep_columns: List[str] = None,
               additional_columns: Dict[str, Callable] = None) -> 'Table':

        if keep_columns is None:         # If no columns specified,
            keep_columns = self.columns  # return all columns

        if additional_columns is None:
            additional_columns = {}

        # New column names and types
        new_columns = keep_columns + list(additional_columns.keys())
        keep_types = [self.col2type(col) for col in keep_columns]
```

```
        # This is how to get the return type from a type annotation.
        # It will crash if `calculation` doesn't have a return type.
        add_types = [calculation.__annotations__['return']
                     for calculation in additional_columns.values()]

        # Create a new table for results
        new_table = Table(new_columns, keep_types + add_types)

        for row in self.rows:
            new_row = [row[column] for column in keep_columns]
            for column_name, calculation in additional_columns.items():
                new_row.append(calculation(row))
            new_table.insert(new_row)

        return new_table
```

> Remember way back in Chapter 2 when we said that type annotations don't actually do anything? Well, here's the counterexample. But look at the convoluted procedure we have to go through to get at them.

Our `select` returns a new `Table`, while the typical SQL `SELECT` just produces some sort of transient result set (unless you explicitly insert the results into a table).

We'll also need `where` and `limit` methods. Both are pretty simple:

```
    def where(self, predicate: WhereClause = lambda row: True) -> 'Table':
        """Return only the rows that satisfy the supplied predicate"""
        where_table = Table(self.columns, self.types)
        for row in self.rows:
            if predicate(row):
                values = [row[column] for column in self.columns]
                where_table.insert(values)
        return where_table

    def limit(self, num_rows: int) -> 'Table':
        """Return only the first `num_rows` rows"""
        limit_table = Table(self.columns, self.types)
        for i, row in enumerate(self.rows):
            if i >= num_rows:
                break
            values = [row[column] for column in self.columns]
            limit_table.insert(values)
        return limit_table
```

after which we can easily construct NotQuiteABase equivalents to the preceding SQL statements:

```
# SELECT * FROM users;
all_users = users.select()
```

```
assert len(all_users) == 11

# SELECT * FROM users LIMIT 2;
two_users = users.limit(2)
assert len(two_users) == 2

# SELECT user_id FROM users;
just_ids = users.select(keep_columns=["user_id"])
assert just_ids.columns == ['user_id']

# SELECT user_id FROM users WHERE name = 'Dunn';
dunn_ids = (
    users
    .where(lambda row: row["name"] == "Dunn")
    .select(keep_columns=["user_id"])
)
assert len(dunn_ids) == 1
assert dunn_ids[0] == {"user_id": 1}

# SELECT LENGTH(name) AS name_length FROM users;
def name_length(row) -> int: return len(row["name"])

name_lengths = users.select(keep_columns=[],
                            additional_columns = {"name_length": name_length})
assert name_lengths[0]['name_length'] == len("Hero")
```

Notice that for the multiline "fluent" queries we have to wrap the whole query in parentheses.

GROUP BY

Another common SQL operation is `GROUP BY`, which groups together rows with identical values in specified columns and produces aggregate values like `MIN` and `MAX` and `COUNT` and `SUM`.

For example, you might want to find the number of users and the smallest `user_id` for each possible name length:

```
SELECT LENGTH(name) as name_length,
 MIN(user_id) AS min_user_id,
 COUNT(*) AS num_users
FROM users
GROUP BY LENGTH(name);
```

Every field we `SELECT` needs to be either in the `GROUP BY` clause (which `name_length` is) or an aggregate computation (which `min_user_id` and `num_users` are).

SQL also supports a `HAVING` clause that behaves similarly to a `WHERE` clause, except that its filter is applied to the aggregates (whereas a `WHERE` would filter out rows before aggregation even took place).

You might want to know the average number of friends for users whose names start with specific letters but see only the results for letters whose corresponding average is greater than 1. (Yes, some of these examples are contrived.)

```
SELECT SUBSTR(name, 1, 1) AS first_letter,
 AVG(num_friends) AS avg_num_friends
FROM users
GROUP BY SUBSTR(name, 1, 1)
HAVING AVG(num_friends) > 1;
```

Functions for working with strings vary across SQL implementations; some databases might instead use `SUBSTRING` or something else.

You can also compute overall aggregates. In that case, you leave off the `GROUP BY`:

```
SELECT SUM(user_id) as user_id_sum
FROM users
WHERE user_id > 1;
```

To add this functionality to NotQuiteABase `Table`s, we'll add a `group_by` method. It takes the names of the columns you want to group by, a dictionary of the aggregation functions you want to run over each group, and an optional predicate called `having` that operates on multiple rows.

Then it does the following steps:

1. Creates a `defaultdict` to map `tuple`s (of the group-by values) to rows (containing the group-by values). Recall that you can't use lists as `dict` keys; you have to use tuples.
2. Iterates over the rows of the table, populating the `defaultdict`.
3. Creates a new table with the correct output columns.
4. Iterates over the `defaultdict` and populates the output table, applying the `having` filter, if any.

```
    def group_by(self,
                 group_by_columns: List[str],
                 aggregates: Dict[str, Callable],
                 having: HavingClause = lambda group: True) -> 'Table':

        grouped_rows = defaultdict(list)

        # Populate groups
        for row in self.rows:
            key = tuple(row[column] for column in group_by_columns)
            grouped_rows[key].append(row)
```

```
    # Result table consists of group_by columns and aggregates
    new_columns = group_by_columns + list(aggregates.keys())
    group_by_types = [self.col2type(col) for col in group_by_columns]
    aggregate_types = [agg.__annotations__['return']
                       for agg in aggregates.values()]
    result_table = Table(new_columns, group_by_types + aggregate_types)

    for key, rows in grouped_rows.items():
        if having(rows):
            new_row = list(key)
            for aggregate_name, aggregate_fn in aggregates.items():
                new_row.append(aggregate_fn(rows))
            result_table.insert(new_row)

    return result_table
```

An actual database would almost certainly do this in a more efficient manner.)

Again, let's see how we would do the equivalent of the preceding SQL statements. The `name_length` metrics are:

```
def min_user_id(rows) -> int:
    return min(row["user_id"] for row in rows)

def length(rows) -> int:
    return len(rows)

stats_by_length = (
    users
    .select(additional_columns={"name_length" : name_length})
    .group_by(group_by_columns=["name_length"],
              aggregates={"min_user_id" : min_user_id,
                          "num_users" : length})
)
```

The `first_letter` metrics:

```
def first_letter_of_name(row: Row) -> str:
    return row["name"][0] if row["name"] else ""

def average_num_friends(rows: List[Row]) -> float:
    return sum(row["num_friends"] for row in rows) / len(rows)

def enough_friends(rows: List[Row]) -> bool:
    return average_num_friends(rows) > 1

avg_friends_by_letter = (
```

```
    users
    .select(additional_columns={'first_letter' : first_letter_of_name})
    .group_by(group_by_columns=['first_letter'],
              aggregates={"avg_num_friends" : average_num_friends},
              having=enough_friends)
)
```

and the user_id_sum is:

```
def sum_user_ids(rows: List[Row]) -> int:
    return sum(row["user_id"] for row in rows)

user_id_sum = (
    users
    .where(lambda row: row["user_id"] > 1)
    .group_by(group_by_columns=[],
              aggregates={ "user_id_sum" : sum_user_ids })
)
```

ORDER BY

Frequently, you'll want to sort your results. For example, you might want to know the (alphabetically) first two names of your users:

```
SELECT * FROM users
ORDER BY name
LIMIT 2;
```

This is easy to implement by giving our Table an order_by method that takes an order function:

```
    def order_by(self, order: Callable[[Row], Any]) -> 'Table':
        new_table = self.select()       # make a copy
        new_table.rows.sort(key=order)
        return new_table
```

which we can then use as follows:

```
friendliest_letters = (
    avg_friends_by_letter
    .order_by(lambda row: -row["avg_num_friends"])
    .limit(4)
)
```

The SQL ORDER BY lets you specify ASC (ascending) or DESC (descending) for each sort field; here we'd have to bake that into our order function.

JOIN

Relational database tables are often *normalized*, which means that they're organized to minimize redundancy. For example, when we work with our users' interests in Python, we can just give each user a `list` containing his interests.

SQL tables can't typically contain lists, so the typical solution is to create a second table called `user_interests` containing the one-to-many relationship between `user_id`s and `interests`. In SQL you might do:

```
CREATE TABLE user_interests (
    user_id INT NOT NULL,
    interest VARCHAR(100) NOT NULL
);
```

whereas in NotQuiteABase you'd create the table:

```
user_interests = Table(['user_id', 'interest'], [int, str])
user_interests.insert([0, "SQL"])
user_interests.insert([0, "NoSQL"])
user_interests.insert([2, "SQL"])
user_interests.insert([2, "MySQL"])
```

> There's still plenty of redundancy—the interest "SQL" is stored in two different places. In a real database you might store `user_id` and `interest_id` in the `user_interests` table and then create a third table, `interests`, mapping `interest_id` to `interest` so you could store the interest names only once each. Here that would just make our examples more complicated than they need to be.

When our data lives across different tables, how do we analyze it? By `JOIN`ing the tables together. A `JOIN` combines rows in the left table with corresponding rows in the right table, where the meaning of "corresponding" is based on how we specify the join.

For example, to find the users interested in SQL you'd query:

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

The `JOIN` says that, for each row in `users`, we should look at the `user_id` and associate that row with every row in `user_interests` containing the same `user_id`.

Notice we had to specify which tables to `JOIN` and also which columns to join `ON`. This is an `INNER JOIN`, which returns the combinations of rows (and only the combinations of rows) that match according to the specified join criteria.

There is also a LEFT JOIN, which—in addition to the combinations of matching rows—returns a row for each left-table row with no matching rows (in which case, the fields that would have come from the right table are all NULL).

Using a LEFT JOIN, it's easy to count the number of interests each user has:

```
SELECT users.id, COUNT(user_interests.interest) AS num_interests
FROM users
LEFT JOIN user_interests
ON users.user_id = user_interests.user_id
```

The LEFT JOIN ensures that users with no interests will still have rows in the joined dataset (with NULL values for the fields coming from user_interests), and COUNT counts only values that are non-NULL.

The NotQuiteABase join implementation will be more restrictive—it simply joins two tables on whatever columns they have in common. Even so, it's not trivial to write:

```
    def join(self, other_table: 'Table', left_join: bool = False) -> 'Table':

        join_on_columns = [c for c in self.columns           # columns in
                           if c in other_table.columns]      # both tables

        additional_columns = [c for c in other_table.columns # columns only
                              if c not in join_on_columns]   # in right table

        # all columns from left table + additional_columns from right table
        new_columns = self.columns + additional_columns
        new_types = self.types + [other_table.col2type(col)
                                  for col in additional_columns]

        join_table = Table(new_columns, new_types)

        for row in self.rows:
            def is_join(other_row):
                return all(other_row[c] == row[c] for c in join_on_columns)

            other_rows = other_table.where(is_join).rows

            # Each other row that matches this one produces a result row.
            for other_row in other_rows:
                join_table.insert([row[c] for c in self.columns] +
                                  [other_row[c] for c in additional_columns])

            # If no rows match and it's a left join, output with Nones.
            if left_join and not other_rows:
                join_table.insert([row[c] for c in self.columns] +
                                  [None for c in additional_columns])

        return join_table
```

So, we could find users interested in SQL with:

```
sql_users = (
    users
    .join(user_interests)
    .where(lambda row: row["interest"] == "SQL")
    .select(keep_columns=["name"])
)
```

And we could get the interest counts with:

```
def count_interests(rows: List[Row]) -> int:
    """counts how many rows have non-None interests"""
    return len([row for row in rows if row["interest"] is not None])

user_interest_counts = (
    users
    .join(user_interests, left_join=True)
    .group_by(group_by_columns=["user_id"],
              aggregates={"num_interests" : count_interests })
)
```

In SQL, there is also a `RIGHT JOIN`, which keeps rows from the right table that have no matches, and a `FULL OUTER JOIN`, which keeps rows from both tables that have no matches. We won't implement either of those.

Subqueries

In SQL, you can `SELECT` from (and `JOIN`) the results of queries as if they were tables. So, if you wanted to find the smallest `user_id` of anyone interested in SQL, you could use a subquery. (Of course, you could do the same calculation using a `JOIN`, but that wouldn't illustrate subqueries.)

```
SELECT MIN(user_id) AS min_user_id FROM
(SELECT user_id FROM user_interests WHERE interest = 'SQL') sql_interests;
```

Given the way we've designed NotQuiteABase, we get this for free. (Our query results are actual tables.)

```
likes_sql_user_ids = (
    user_interests
    .where(lambda row: row["interest"] == "SQL")
    .select(keep_columns=['user_id'])
)

likes_sql_user_ids.group_by(group_by_columns=[],
                            aggregates={ "min_user_id" : min_user_id })
```

Indexes

To find rows containing a specific value (say, where `name` is "Hero"), NotQuiteABase has to inspect every row in the table. If the table has a lot of rows, this can take a very long time.

Similarly, our `join` algorithm is extremely inefficient. For each row in the left table, it inspects every row in the right table to see if it's a match. With two large tables this could take approximately forever.

Also, you'd often like to apply constraints to some of your columns. For example, in your `users` table you probably don't want to allow two different users to have the same `user_id`.

Indexes solve all these problems. If the `user_interests` table had an index on `user_id`, a smart `join` algorithm could find matches directly rather than scanning the whole table. If the `users` table had a "unique" index on `user_id`, you'd get an error if you tried to insert a duplicate.

Each table in a database can have one or more indexes, which allow you to quickly look up rows by key columns, efficiently join tables together, and enforce unique constraints on columns or combinations of columns.

Designing and using indexes well is something of a black art (which varies somewhat depending on the specific database), but if you end up doing a lot of database work it's worth learning about.

Query Optimization

Recall the query to find all users who are interested in SQL:

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

In NotQuiteABase there are (at least) two different ways to write this query. You could filter the `user_interests` table before performing the join:

```
(
    user_interests
    .where(lambda row: row["interest"] == "SQL")
    .join(users)
    .select(["name"])
)
```

Or you could filter the results of the join:

```
(
    user_interests
    .join(users)
    .where(lambda row: row["interest"] == "SQL")
    .select(["name"])
)
```

You'll end up with the same results either way, but filter-before-join is almost certainly more efficient, since in that case `join` has many fewer rows to operate on.

In SQL, you generally wouldn't worry about this. You "declare" the results you want and leave it up to the query engine to execute them (and use indexes efficiently).

NoSQL

A recent trend in databases is toward nonrelational "NoSQL" databases, which don't represent data in tables. For instance, MongoDB is a popular schemaless database whose elements are arbitrarily complex JSON documents rather than rows.

There are column databases that store data in columns instead of rows (good when data has many columns but queries need few of them), key/value stores that are optimized for retrieving single (complex) values by their keys, databases for storing and traversing graphs, databases that are optimized to run across multiple datacenters, databases that are designed to run in memory, databases for storing time-series data, and hundreds more.

Tomorrow's flavor of the day might not even exist now, so I can't do much more than let you know that NoSQL is a thing. So now you know. It's a thing.

For Further Exploration

- If you'd like to download a relational database to play with, SQLite is fast and tiny, while MySQL and PostgreSQL are larger and featureful. All are free and have lots of documentation.
- If you want to explore NoSQL, MongoDB is very simple to get started with, which can be both a blessing and somewhat of a curse. It also has pretty good documentation.
- The Wikipedia article on NoSQL almost certainly now contains links to databases that didn't even exist when this book was written.