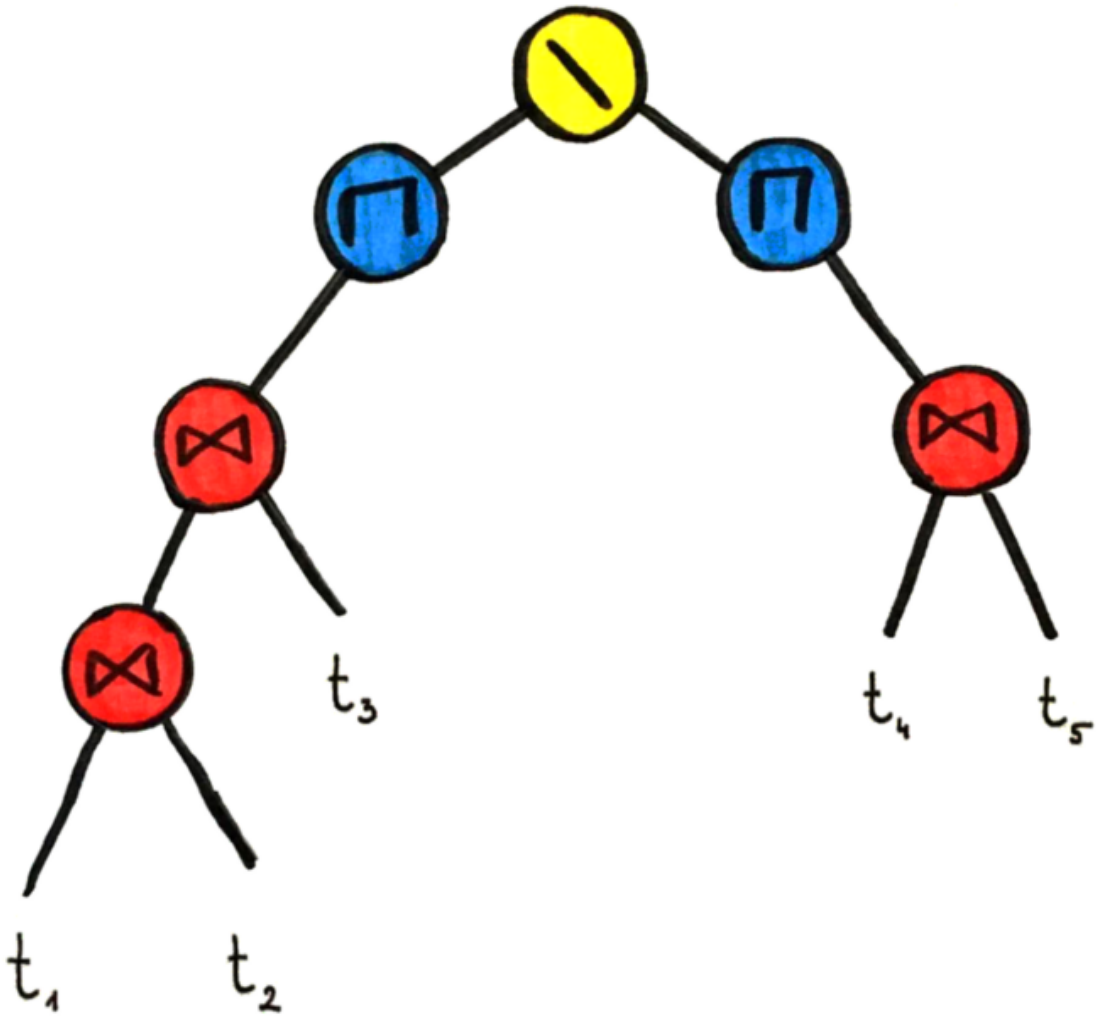
medium.com

How to Build a Relational Database From Scratch - The Startup - Medium

Tivadar Danka

11-14 minutes



Opening the black box of relational databases

Relational databases are amazing. They are not only very effective at storing data, but they also have a beautiful underlying mathematical theory as well. However, especially when one is using a database through an ORM, it can seem like a black box. Opening the black boxes is an excellent way to master a subject, so I have decided to go into the details and build a minimal but functional relational database system on my own. In this post, I am sharing my journey with you.

Instead of trying to follow how SQL works under the hood, my aim is to show the fundamental principles behind relational databases. Even if one wants to be more specific, there are several SQL dialects like PostgreSQL, MySQL, and many more, which can have significant differences.

Before we dive deep into the implementation, we should have a solid understanding of the underlying mathematical objects. So, let's talk about what relations really are!

(If you would like to follow interactively and play around with the code, you can find the repository on GitHub at https://github.com/cosmic-cortex/relational-databases-from-scratch!)

Relations

> "Mathematicians are like Frenchmen: whatever you say to them they translate into their own language and forthwith it is something entirely different." — Johann Wolfgang von Goethe

To understand relational databases, first, we shall define what is a *relation*. Mathematically speaking, a relation R over sets

$$X_1, X_2, \dots, X_n$$

is a subset of their Cartesian product:

$$R \subseteq X_1 \times X_2 \times \cdots \times X_n.$$

To give you an example, the < operator is a relation. You can think of it as a set of all tuples of natural numbers where the first element in the tuple is less than the second. This is how actually < is defined in mathematics.

A more concrete example will help grasp this abstract definition. Suppose that we are talking about employees in a company. Each employee has a unique id, a name, position, and a salary:

$$X_1 = \mathbb{N}$$

$$X_2 = \text{set of names}$$

$$X_3 = \text{set of positions}$$

$$X_4 = \mathbb{R}$$

In this model, an employee is a vector of four dimensions:

$$(0, \text{"Michael Scott"}, \text{"Regional Manager"}, 100000)$$
$$(1, \text{"Dwight K. Schrute"}, \text{"Assistant to the Regional Manager"}, 65000)$$

Employee relation is a set of these. Using relational databases terminology, a relation corresponds to a *table* and an element of a relation is a *record* in that table. In the following, I will use these terms interchangeably.

In Python, we are going to model records with special dictionaries and tables with a set of records. We could go with something more

complex (like `pandas` data frames for instance) but these will be perfect. Note that by going with sets, duplicate records are not allowed.

Because dictionaries are unhashable by default, they cannot be inserted into sets. To circumvent this, I have added a hash method for the `Record` class, which returns the hash of the tuple obtained by the dict.

This is dangerous to do, so we should be careful. Because dictionaries are mutable, their “hash” (as I defined above) can change. Thus, if you put this into a set and change one of its value, the hash changes and because sets in Python use hashes under the hood, this can mess things up. So, to avoid this, I have explicitly disabled modifying the record by overwriting the `__setitem__` special method.

To have some more data to play around with, we shall also add a table for tasks with the following columns: `id`, `employee_id`, `completed`. The `employee_id` will describe which employee does the task with id `id` belongs to, and `completed` is simply a Boolean, indicating its status.

Our tables are going to be sets of records, which we create manually for now.

Now that we have laid a proper foundation, it is time to look at what makes relational databases work: the operations.

Relational algebra: operations on relations

A relational database would be practically useless if we wouldn’t have ways to retrieve and structure information within the

database. This task is achieved by applying operators on relations. The relational algebra is the mathematical structure defined by these operations. To be precise, elements of the relational algebra are functions whose inputs and outputs are relations.

If this sounds too abstract for you, let's see some concrete examples!

Selection

One of the most fundamental operators is the selection operation, which filters the relation based on certain conditions. It is denoted by

$$\sigma_C,$$

where *C* denotes the conditions. In our example database, a condition can be that *"the employee's salary is more than 60000"*. It can be thought of as a function that takes a record and returns a Boolean indicating whether the condition has been met.

If we apply the condition *"salary is more than 60000"* for our `employees` table, this is what we get.

```
In [1]: select(employees, [lambda x: x['salary'] > 60000])
Out[1]:
{{'id': 0,
  'name': 'Michael Scott',
  'position': 'Regional Manager',
  'salary': 100000},
 {'id': 1,
  'name': 'Dwight K. Schrute',
  'position': 'Assistant to the Regional Manager',
  'salary': 65000}}
```

Projection

The `select` operator is used to select *records* in our table. However, we might wish to apply filters to the columns as well for removing unnecessary information. This can be done with the projection operator. It is denoted with

$$\Pi_S,$$

where *S* denotes the list of columns which should remain. This is also pretty simple to implement in our setting.

This is what happens when we project the `employees` table to the `id` and `name` columns.

```
In [1]: project(employees, ['id', 'name'])
Out[1]:
{{'id': 0, 'name': 'Michael Scott'},
 {'id': 1, 'name': 'Dwight K. Schrute'},
 {'id': 2, 'name': 'Pamela Beesly'},
 {'id': 3, 'name': 'James Halpert'},
 {'id': 4, 'name': 'Stanley Hudson'}}
```

Rename

Renaming columns is often very useful. For instance, both of our tables have an `id` column, which might introduce some ambiguity, so renaming it is needed.

```
In [1]: employee_names = project(employees, ['id', 'name'])
In [2]: rename(employee_names, {'name': 'full name'})
Out[2]:
{{'full name': 'Michael Scott', 'id': 0},
 {'full name': 'Dwight K. Schrute', 'id': 1},
 {'full name': 'Pamela Beesly', 'id': 2},
 {'full name': 'James Halpert', 'id': 3},
 {'full name': 'Stanley Hudson', 'id': 4}}
```

Cross-product

This is where things start to get interesting. The true strength of relational databases is the fact that you are able to combine information within several tables and answer complicated questions by querying these combined tables. The simplest method to do this is taking the cross product, which merges records together in all of the possible ways. I am going to show an example before the implementation to make sure it is clear.

```
In [1]: cross_product(left=employees, right=tasks)
Out[1]:
{{'right.id': 0,
  'right.employee_id': 0,
  'right.completed': False,
  'left.name': 'Michael Scott',
  'left.salary': 100000,
  'left.id': 0,
  'left.position': 'Regional Manager'},
 {'right.id': 1,
  'right.employee_id': 0,
  'right.completed': False,
  'left.name': 'Michael Scott',
  'left.salary': 100000,
```

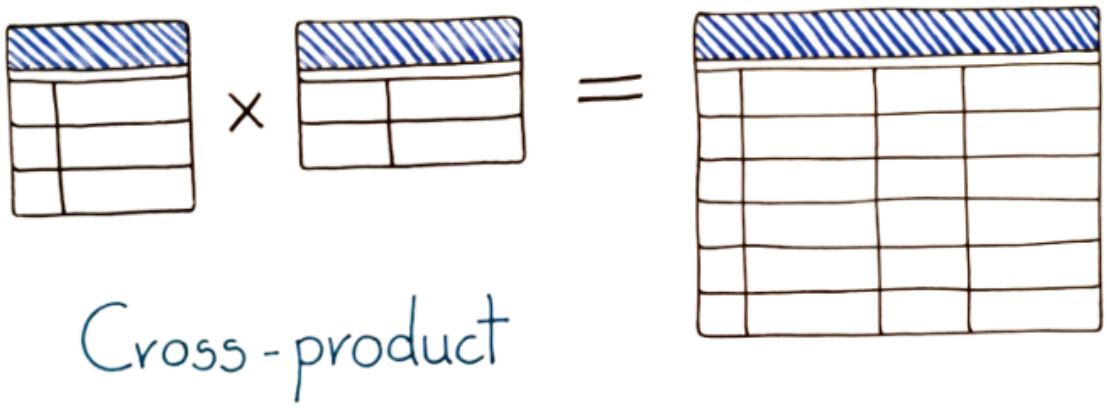
```
  'left.id': 0,
  'left.position': 'Regional Manager'},
 {'right.id': 2,
  'right.employee_id': 1,
  'right.completed': True,
  'left.name': 'Michael Scott',
  'left.salary': 100000,
  'left.id': 0,
  'left.position': 'Regional Manager'},
  .
  .
  .
```

Here is where we see that colliding column names can become an issue. Before the cross product is taken, each column is prefixed with the table name for every table.

The cross-product operator is denoted with

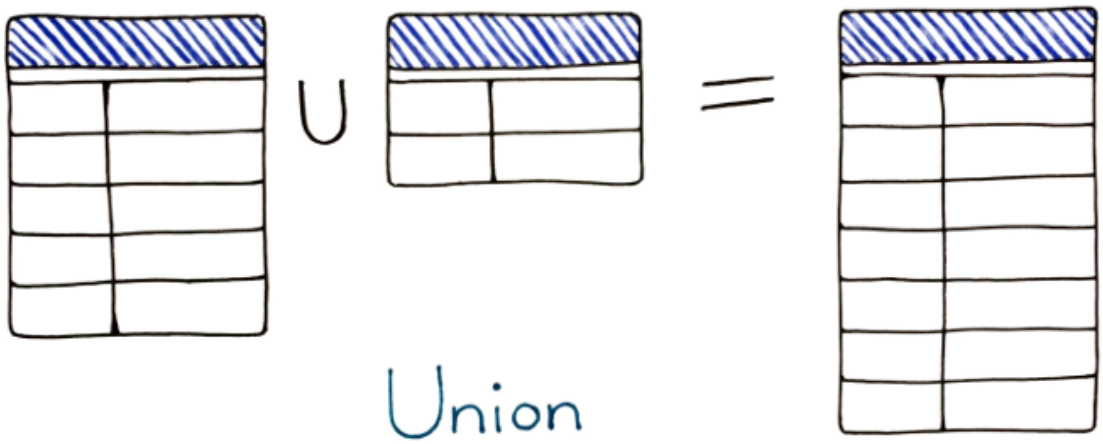
$$A \times B.$$

You have probably observed that cross-product combines information without regarding logical coherence. Some records in the cross-product table shows an employee and a task which belongs to another employee. We will fix this later when we are using joins.



Union

So, we have seen that with the cross-product, we can combine tables “horizontally”. It is natural to expect the same “vertically”, as was the case with filtering rows (`select`) and columns (`project`). This can be done with the `union` operator.



Strictly speaking, this operator doesn’t really make sense when the tables have distinct columns. However, there is a solution for this: the missing columns can be padded with null values such as `None`.

Difference

Our last operator is the `difference`, which is again similar to the set-theoretic difference. It simply eliminates the rows of one table from another.

Because tables in our implementation are sets, we can simply use the built-in method.

You might notice that the intersection is left out. It is because intersection can be expressed with a difference by

$$A \cap B = A \setminus (A \setminus B),$$

or

difference(left, difference(left, right))

in code.

The case of the intersection operator is not unique. In fact, all relevant operators can be obtained as a combination of the previous six operators, as we will see later.

SQL queries in terms of relational algebra

Now that we have all these seemingly abstract operations, we can begin to see how SQL queries translate to relational algebra. Suppose that we want to query the names of all employees who have a salary larger than 60k USD. In SQL, this would look like

SELECT name FROM employees WHERE salary > 60000

In our implementation, this is equivalent to

```
temp_table = select(employees, [lambda x: x['salary'] > 60000])
result = project(temp_table, ['name'])
```

So, we can see that our tools are powerful enough to express these type of queries. However, the neatest things are still ahead of us. To be able to answer more complex questions regarding our data (like what is the average salary of those employees who have an incomplete task), we need to be able to intelligently combine information between tables. These are done with joins.

Joins

There are several methods to combine tables together in the relational algebra. So far, we have seen one: the cross-product, which combined all records together, even when they were

logically not consistent. However, combining tables horizontally can be done such that the resulting table will contain meaningful information. Again, there are several methods for this. We will begin with the most fundamental one: the *theta join*.

Theta join

The theta join operator is simply the combination of the cross-product and the select operations. It is denoted with

$$A \bowtie_\theta B,$$

where θ denotes the condition of the select. For instance, the theta join of *employees* and *tasks* on the condition that `employee['id']` and `task['employee_id']` match looks like the following.

```
In [1]: theta_join(
    left=employees,
    right=tasks,
    conditions=[lambda x, y: x["id"] == y["employee_id"]]
)
Out[1]:
{{'left.id': 0,
  'left.name': 'Michael Scott',
  'left.position': 'Regional Manager',
  'left.salary': 100000,
  'right.id': 0,
  'right.employee_id': 0,
  'right.completed': False},
 {'left.id': 0,
  'left.name': 'Michael Scott',
  'left.position': 'Regional Manager',
  'left.salary': 100000,
  'right.id': 1,
  'right.employee_id': 0,
  'right.completed': False},
 {'left.id': 1,
  'left.name': 'Dwight K. Schrute',
  'left.position': 'Assistant to the Regional Manager',
  'left.salary': 65000
```

As you can see, this solves the problem we had for the cross-product, where the task in a given record did not necessarily belong to the employee in the same record.

Outer joins

In some occasions, it is useful to preserve some information during a join even if there is no logically corresponding element in the other table. This is what the left/right/full outer join operators do. We are not going to go into detail, but these operators take the rows in the left/right/both tables which are missing from the theta join and union them together.

Queries as elements of the relational algebra

We have seen that most queries, even joins, can be expressed with only six operators:

- Select
- Project
- Rename
- Cross-product
- Union
- Difference

To see a more complex query, let's introduce a third table, containing the clients of Dunder Mifflin Scranton. Each client has an `id`, a `name` and a `contact_id`, which is the contact employee's id.

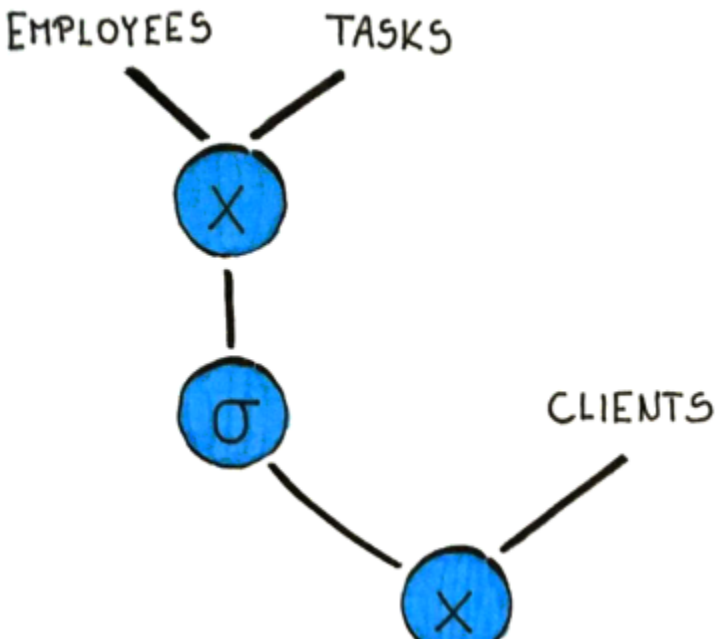
Now consider the following query: *"what are the names of the clients whose contacts have at least one incomplete task?"*

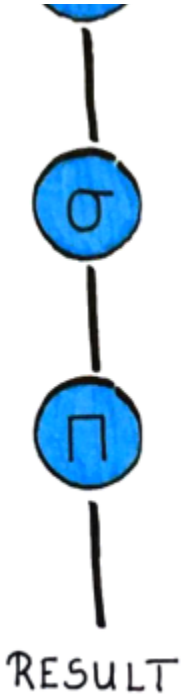
To answer this, we have to combine information from all three tables. It is useful to think through the steps we need to take. Here, we are going to

1. find the employees who have incomplete tasks by joining the *employees* and *tasks* tables together,
2. join the resulting table to the *clients* table to match client names to contact employees,
3. select the client names by projecting to the client name column.

In code, this would look like the following.

No matter how complicated, every query can be represented as a graph called the *expression tree.*





Expression tree for the query *“what are the names of the clients whose contacts have at least one incomplete task?”*

In fact, this is not the only possible way to do this query, there are alternative solutions. For example, we could create the cross-product for all three tables right away and filtered out the required records using a single `select`. When using SQL, the engine tries to optimize the query by estimating the required cost for a given execution plan and choosing the most optimal for you.

From relational algebra to SQL

Relational algebra is just the tip of the iceberg. It gives us the building blocks to represent queries, however, it does not give us the tools to find how a query can be optimally represented. Under the hood, SQL actually uses relational calculus, which essentially a declarative language based on relational algebra. This means that you only describe what you want, not how you want it. The latter part is figured out by the engine.

We are missing core SQL functionalities like for example aggregates, i.e. functions mapping relations to elements, such as the averaging function. So, there is still a long way to go for a fully functional relational database. However, the fundamentals are laid and the path towards it is clear.

Resources

During my quest, I had two excellent resources helping me to build a relational database:

- Databases MOOC by Jennifer Widom
- The Theory of Relational Databases by David Maier