CMSC 206: Database Management Systems

Query processing and optimisation

Thomas Laurent

31/10/2020

University of the Philippines Open University

Table of contents

Introduction

Introduction

Last time we saw how to use SQL to define, populate and query our DB. This week we will take a (very high level) look at what the RDBMS does when we submit a query.

Overview

When we submit a query to the DBMS, it:

1. validates the query
2. parses and translates the query to relational operations
3. optimises the query
4. runs the query

We will take a look at each of these steps

Validating and parsing the query

Validating and parsing the query

In this step, the RDBMS first checks that the query makes sense: that the query is syntactically correct, and that all the elements (schemas, tables, columns, functions, ...) it refers to exist. The query is also translated to an internal representation used by the RDBMS.

I will not cover how this is done. Parser theory is a whole field of its own and is discussed extensively in the literature.

Optimising the query

Introduction

SQL is a declarative language that we use to tell the RDBMS *what* we want, not *how* it should get it.

There are many ways to (physically) execute an SQL query. i.e. there are different equivalent relational algebra expressions that can represent a query.

The RDBMS tries to find an efficient way to run the query we give it. A way to run a query is called a **query plan**.

Optimising relational algebra expression

The main idea of the optimisation process is that we want to handle as little data as we can.

If two plans perform the same query, the most efficient one will usually:

- Perform selection early
- Use joins rather than cartesian products (taking a condition from the WHERE clause for example)
- Project out useless attributes early
- Perform the most restrictive joins first

Those rules are called *heuristics* and are basically rules of thumb used by the RDBMS.

Simple example

Say we have a table Country(country_id, country_population, country_name) with 200 tuples in it, and a table City(city_id, country_id, city_population, city_name) where country_id is a foreign key to Country with 10 000 tuples in it.

Now let's say we want to query our DB like this:

```
SELECT City.city_name
FROM City NATURAL JOIN Country
WHERE City.city_population > 100 000
AND Country.country_name = "France"
```

How can the DBMS answer this query?

Simple example: The naive way

To answer this query, the DBMS can do the following steps:

1. Cartesian product of Country and City (2 000 000 tuples with 7 columns!)
2. Filter out the tuples where the two country_id are equal
3. Filter out the tuples where city_population $\leq$ 100 000
4. Filter out the tuples where country_name $\neq$ “France”
5. Project on city_name

This will give us the correct result but creates a very big table that will take up a lot of memory, and will be very long to go through when filtering our tuples. It also involves a lot of data we do not need for most of the process.

Simple example: A better way

To answer this query, the DBMS can also do the following steps:

1. Filter out the tuples in City where city_population $\leq$ 100 000 (noted T1, $\leq$ 10 000 tuples)
2. Project T1 on city_id, country_id, city_name
3. Filter out the tuples in Country where country_name $\neq$ “France” (noted T2, hopefully $\leq$ 1 tuple)
4. Join T1 and T2 on country_id ($\leq$ 10 000 tuples, 5 columns)
5. Project on city_name

This might not be the best plan depending on the exact data in our table, what columns are indexed (we will see indexes next week) and how things are implemented in the DBMS but it sure will be better than the naive approach.

Cost estimation

To know which query plan will be most effective without running them all (which would defeat the point) we need to *estimate* the cost of each plan.

To do this, the RDBMS keeps some statistics about the tables and the data they contains, such as size of the table and the distribution of values inside the table. This data is what we call the **catalog information**. Using this data the RDBMS can evaluate which conditions are more selective and the approximate size of joins/cross products and thus have an idea of which plan will be the most efficient.

Be nice to your query optimiser

Writing queries that are easy to optimise

Although the RDBMS will try to find an efficient plan to execute your query, it is always good to keep performance in mind when writing it. If you join big tables together, your query will be slow, even with good optimisation. If you write your condition in a certain way (using columns that are not indexed, using certain functions, ...) or if you use some data types, you might prevent the RDBMS from optimising your query.

A few resources exist on the art of query tuning. For example *here*, or *here*. But experience with the system will also play a big role, as each RDBMS had its own quirks.

Conclusion

Conclusion

This week we only took a very high level look at how a RDBMS processes queries. I chose this approach as you will be more likely to use a RDBMS than to implement one, but it is still good to have some idea of how the system works. This way, when things go wrong/slow, you can have an idea of what is going on, rather than wondering what is wrong with the magic black box.

If you are interested in the subject there is plenty of literature available. As usual, the Additional Materials section gives a starting point.