Imperial College
Business School

Imperial means
Intelligent Business

Lecture 8
Query optimisation and database maintenance

Dr Fintan Nagle
f.nagle@imperial.ac.uk

Query optimisation

Slow queries can usually be made faster.

However, speed doesn't depend just on the syntax of the query: the query planner is clever and is constantly improving.

Sometimes, queries written in very different ways can compile to the same execution plan – and thus the same speed.

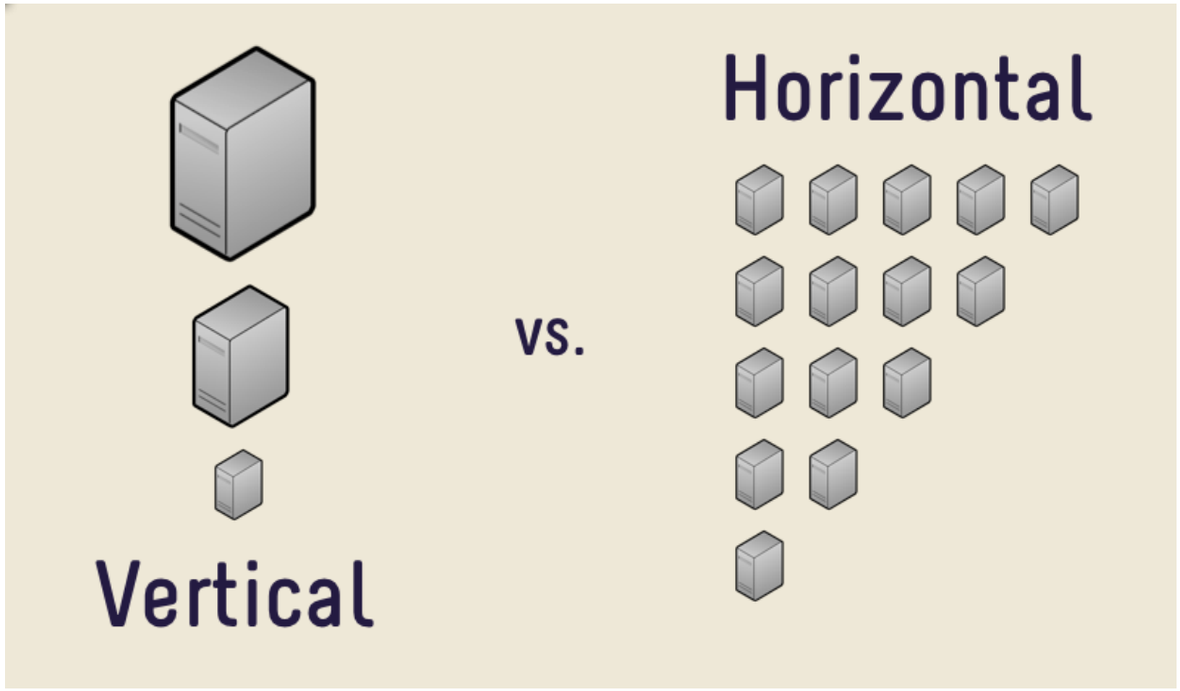
Database maintenance

Varieties of SQL

What follows is specific to Postgres. However, all database products have some kind of maintenance tools, and other SQL products use similar techniques.

Provisioning hardware

There is a tradeoff between spending money on database hardware, and fast queries.



We can scale either

- vertically (a bigger database server)
- horizontally (more database servers of the same size, working together)

There is also a tradeoff between spending money on database hardware, and spending time on database design and query optimisation.

As hardware and computing resources get cheaper, there is a temptation to *not optimise, just rely on fast hardware.* This is dangerous.

Data cleaning

It's important that only clean, tidy data is allowed into the database. We can clean data manually or using Python.

- **Null values**
- **Outliers (data range)**
- **When averaging, a large enough number of data points must be used, or the average will not be reliable**
- **With all analysis methods: junk in, junk out!**
- **Graphs: the axes must be sensible (graph display range)**

Maintenance

- Indexes
  Allow rows to be found without sequential scanning

- Vacuuming
  Reclaim unused or deleted storage

- ANALYZE
  Work out statistics about the database, which the query planner can use to optimise queries.

VACUUM

"VACUUM reclaims storage occupied by dead tuples [records]. In normal PostgreSQL operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a VACUUM is done. Therefore it's necessary to do VACUUM periodically, especially on frequently-updated tables."

https://www.postgresql.org/docs/9.1/sql-vacuum.html

VACUUM: vacuum the entire database
VACUUM film: vacuum just the film table

The **autovacuum daemon** can vacuum automatically.

Indexes

Indexes allow rows to be found (by a particular column) much faster than by scanning the disk.

" CREATE INDEX constructs an index on the specified column(s) of the specified table. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance)."

https://www.postgresql.org/docs/9.1/sql-createindex.html

Create index:
CREATE INDEX title_idx **ON** films (title);

Create index which also enforces uniqueness:
CREATE UNIQUE INDEX title_idx **ON** films (title);

ANALYZE

"ANALYZE collects statistics about the contents of tables in the database, and stores the results in the **pg_statistic** system catalog. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries."

https://www.postgresql.org/docs/9.1/sql-analyze.html

ANALYZE: vacuum the entire database
ANALYZE film: vacuum just the film table

ANALYZE VERBOSE: show progress
Note that analyse shows no output; it stores its conclusions in the pg_statistic table.

ANALYZE

"In the default PostgreSQL configuration, the **autovacuum daemon** (see Section 23.1.5) takes care of automatic analyzing of tables when they are first loaded with data, and as they change throughout regular operation.

When autovacuum is disabled, it is a good idea to run ANALYZE periodically, or just after making major changes in the contents of a table.

Accurate statistics will help the planner to choose the most appropriate query plan, and thereby improve the speed of query processing. A common strategy is to run VACUUM and ANALYZE once a day during a low-usage time of day."

Backups

It is **absolutely key** to

- Maintain automated backups
- Keep them offsite
- Test the restoration process frequently

pg_dump and pg_restore can be used to do manual backups and restore them.

Services like AWS Relational Database Service (RDS) do automated backups for you, and make restoring easy.

Schema migrations

What happens when you change the database? For example, we could add a new table, delete a table, or change an attribute name.

Problems:

Schema migrations

Example migration in Ruby on Rails:

```ruby
class CreateUserCourses < ActiveRecord::Migration[6.0]
  def change
    create_table :user_courses do |t|
      t.references :course, foreign_key: true
      t.references :user, foreign_key: true
      t.datetime :schedule_start
      t.datetime :schedule_end
      t.boolean :completed, default: false

      t.timestamps
    end
  end
end
```

Can be applied (or undone) automatically.

Data migrations

We usually have three versions of an application database:

- **Production** (live customer-facing version)
- **Staging** (test environment on the Web in a real server
- **Development** (programming environment on developers' machines, not accessible online)

Apart from the migrations, **data** needs to be kept in sync – we may need to move data from production to staging, or from development to staging.

Often, specific subsets of data need to be inserted (rather than wiping out the entire DB) and this process needs to be managed.

Fault and disaster recovery

There must be a **process** for recovering from disasters, restoring backups, and checking the system.

There must be a further process for:

- doing a post-mortem to isolate the cause
- putting measures in place so that the same problem does not reoccur

Counting and numbering

Row numbering

Postgres

```
SELECT row_number() OVER (ORDER
BY BirthDate) as n, * FROM employees
```

SQL Server / MS Access Syntax:

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE condition;
```

Rows have no intrinsic number

MySQL Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

Oracle Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number;
```

Even rows (Postgres)

Select even-numbered rows from the employees table

Even rows (Postgres)

Select even-numbered rows from the employees table

```
SELECT * FROM

    (SELECT
    row_number() OVER (ORDER BY BirthDate) as n, *
    FROM employees)
    AS t

WHERE mod(n,2)=0
```

Northwind

Show the top 3 oldest employees (Northwind)

Northwind

Show the top 3 oldest employees (Northwind)

```
SELECT * FROM employees
ORDER BY BirthDate ASC
LIMIT 3
```

Northwind

Show the first half of the products table

Northwind

Show the first half of the products table

```
SELECT * FROM

(SELECT row_number() OVER (ORDER
BY ProductID) as n, *
FROM products)
AS t

WHERE n < (SELECT COUNT(*)

FROM products)/2
```

Northwind

Select the first half of the products table, ordered by ID.

Northwind

Select the first half of the products table,
ordered by ID.

```
SELECT * FROM Products

WHERE ProductID < (SELECT COUNT(*) FROM products)/2
```

Why don’t we do this with IDs?

IDs

The ID is not the same as the row number.

The row count may be less than the highest ID number, since some rows may be deleted.

Counting example

Question

For a table orders having a column defined simply as customer_id VARCHAR(100), consider the following two query results:

```
SELECT count(*) AS total FROM orders;
```

total
100

```
SELECT count(*) AS cust_123_total FROM orders
WHERE customer_id = '123';
```

cust_123_total
15

Given the above query results, what will be the result of the query below?

```
SELECT count(*) AS cust_not_123_total FROM orders
WHERE customer_id <> '123';
```

Question

Intuitively, it is 85

However, values could be NULL, in which case they will never be <> ‘123’