Imperial College
Business School

Imperial means
Intelligent Business

Lecture 2
Our first SQL queries

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

Reading

Video lectures:

2.4.1 - SELECT, WHERE and LIMIT syntax.mp4
2.4.2 - Formatting conventions.mp4
3.2 - Making counting queries.mp4

Reading

Postgres documentation on the client/server model:
https://www.postgresql.org/docs/10/static/tutorial-arch.htm I
Postgres documentation on aggregate functions:
https://www.postgresql.org/docs/10/static/tutorial-agg.html
Postgres documentation on SELECT, including GROUP BY:
https://www.postgresql.org/docs/current/static/sql-select.htm I
Postgres documentation on ORDER BY:
https://www.postgresql.org/docs/current/static/queries-order.html
NULL and the billion-dollar mistake:
https://www.lucidchart.com/techblog/2015/08/31/the-worst-mistake-of-computer-science/

Postgres documentation on datatypes:
https://www.postgresql.org/docs/current/static/datatype.html

Vital documents

- SQL cheat sheet (available in the exam)
- ER diagrams for Northwind
- and dvdrental

PostgreSQL cheat sheet

Our practice databases

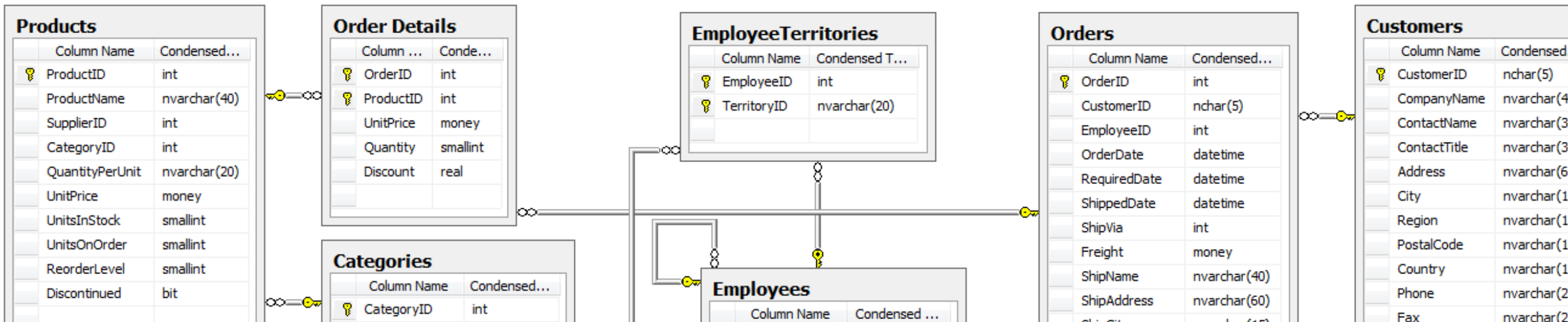
- Host: imperial.ckp3dl3vzxoh.eu-west-2.rds.amazonaws.com
- Username: imperial
- Password: imperial
- Port: 5432
- Database (maintenance DB): either **dvdrental** or **northwind**

In Postgres

- Single quotes only - no double quotes
- This is a Postgres cheat sheet. MySQL or other versions of SQL may operate slightly differently.

Standard query structure

- SELECT (including window functions)
- FROM
- JOINs, each with an ON
- WHERE
- ORDER BY
- LIMIT



General tips for writing queries

General SQL tips

Use newlines liberally to separate the parts of queries.

```
SELECT * FROM employees INNER JOIN roles ON employees.role_id =
roles.id INNER JOIN managers ON employees.manager_id = managers.id
WHERE employees.age age > 50 ORDER BY employees.age DESC
```

```
SELECT * FROM
employees INNER JOIN roles
ON employees.role_id = roles.id
INNER JOIN managers
ON employees.manager_id = managers.id
WHERE employees.age age > 50
ORDER BY employees.age DESC
```

General programming/data science tips

- Try it out! You learn best in front of a live console window.

- Set yourself challenges!
  “I wonder if I can find all the students with more than two supervisors...”

- There are many online resources for SQL:
  Google “SQL” or “Postgres” along with your problem or error message
  Stack Overflow is very useful – if you post a clear question (that hasn’t been asked before), it’s usually answered within a day

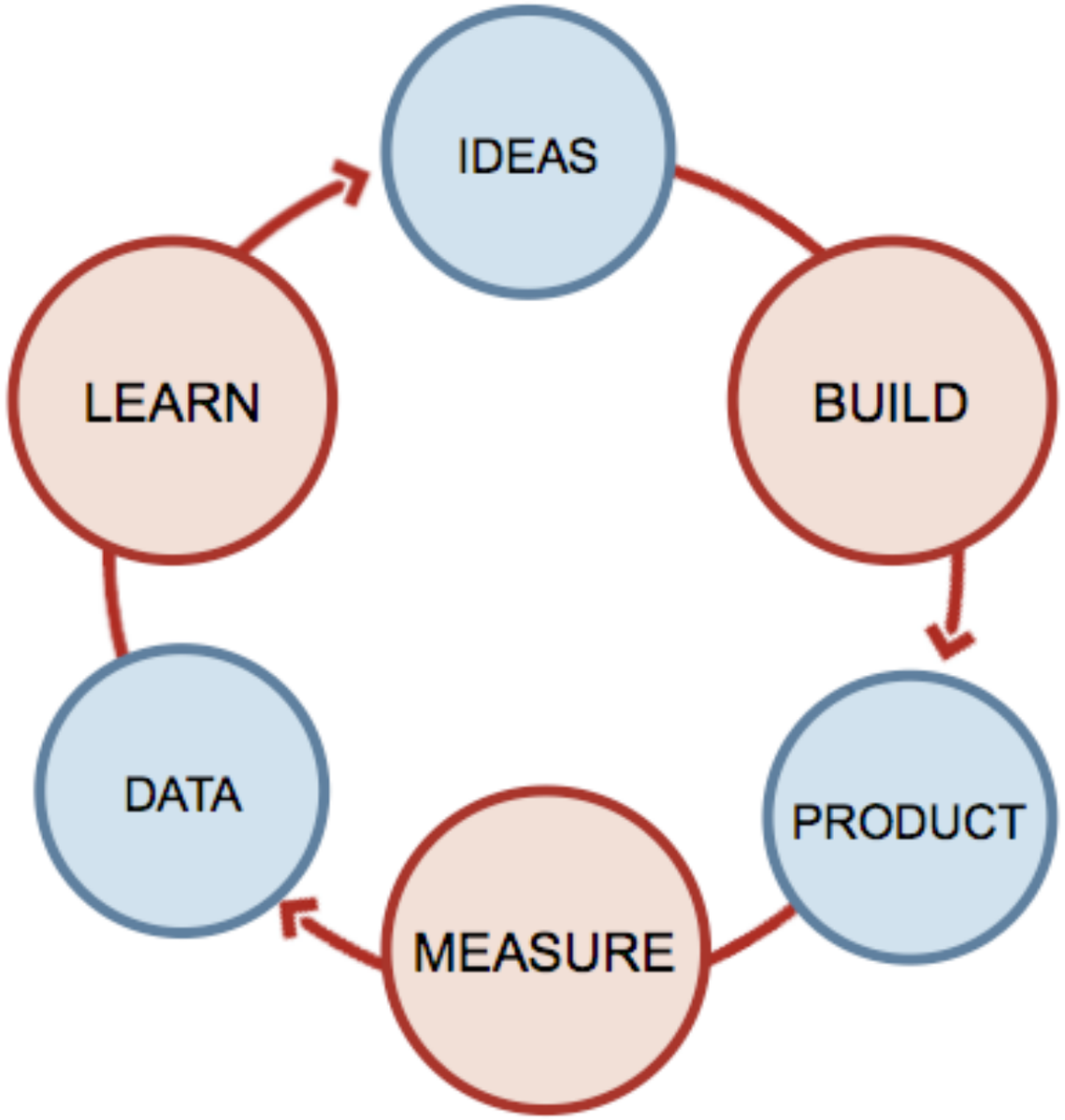
- Make a personal cheat sheet to help you remember syntax

General SQL tips

- Postgres: single quotes only, no double quotes

Developing queries (the development loop)

- Get a simple version of the query to run first
- Check the results
- Extend and tune the query



Naming rules

- Always use the singular (movie, not movies)
- Have good case rules:

```
CamelCaseMeansMoreCognitiveLoad
snake_case_is_better
```

Vital documents

- SQL cheat sheet (available in the exam)
- ER diagrams for Northwind
- and dvdrental

PostgreSQL cheat sheet

Our practice databases

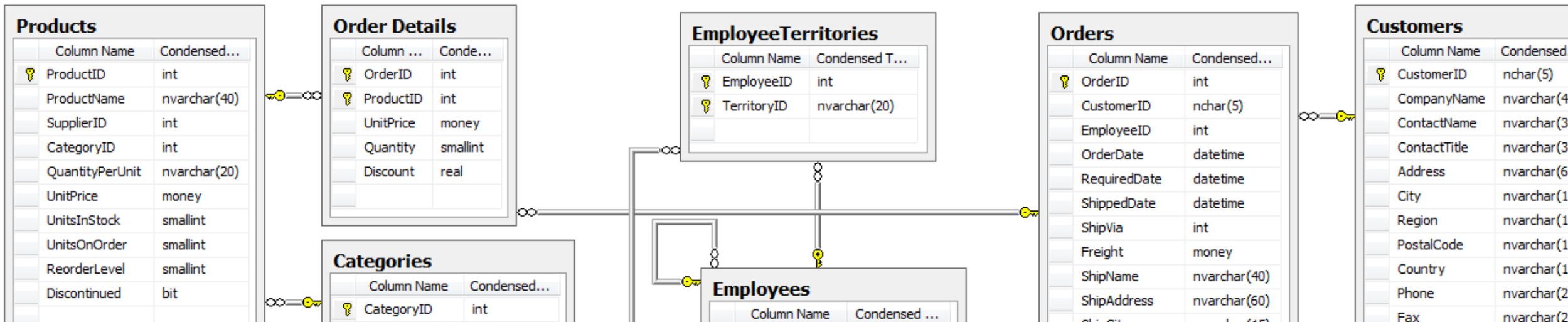
- Host: imperial.ckp3dl3vzxoh.eu-west-2.rds.amazonaws.com
- Username: imperial
- Password: imperial
- Port: 5432
- Database (maintenance DB): either **dvdrental** or **northwind**

In Postgres

- Single quotes only - no double quotes
- This is a Postgres cheat sheet. MySQL or other versions of SQL may operate slightly differently.

Standard query structure

- SELECT (including window functions)
- FROM
- JOINs, each with an ON
- WHERE
- ORDER BY
- LIMIT



NULL

- NULL is a special code indicating that a value is missing, empty, or not present.
- Always in capitals in Postgres.
- When Tony Hoare introduced NULL to the Algol language in 1965, it eventually caused so many problems that he called it "my billion-dollar mistake."
- We can check for TRUE using **WHERE** attribute = TRUE
  However we can't use **WHERE** attribute = NULL
  because **nothing is ever equal to NULL.**
  So we have to use **WHERE** attribute **IS NULL**

Always consider NULLs!

Guess at the result of these comparisons, and then try them out in your console. Remember that nothing is ever equal to NULL, and that NULL compared to anything is NULL.

1. **SELECT** (**TRUE** = **TRUE**)

2. **SELECT** (**TRUE** = **FALSE**)

3. **SELECT** (**3** < **10**)

4. **SELECT** (**NULL** = **TRUE**)

5. **SELECT** (**NULL** = **FALSE**)

6. **SELECT** (**NULL**< **10**)

7. **SELECT** (**NULL** <> **5**)

7. **SELECT** (**NULL** <> **5**) IS NULL

Always consider NULLs!

Guess at the result of these comparisons, and then try them out in your console. Remember that nothing is ever equal to NULL, and that NULL compared to anything is NULL.

1. **SELECT** (**TRUE** = **TRUE**) true

2. **SELECT** (**TRUE** = **FALSE**) false

3. **SELECT** (**3** < **10**) true

4. **SELECT** (**NULL** = **TRUE**) null

5. **SELECT** (**NULL** = **FALSE**) null

6. **SELECT** (**NULL**< **10**) null

7. **SELECT** (**NULL** <> **5**) null

7. **SELECT** (**NULL** <> **5**) IS NULL true

Always consider NULLs!

NULL, when included in any comparison (using = or > or < or <>),
makes the result NULL.

Keys: uniquely identifying rows

Keys

- Generally, a **key** is an attribute (or group of attributes) that uniquely identifies a row. For example, student ID is a key for a student table; however, DOB and last name could be a workable key too.

- Someone has to make sure that the keys are unique (we can ask Postgres to check this)

Are rows uniquely identifiable?
A

dogs

name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

Are rows uniquely identifiable?
B

dogs

name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10
Fido	companion	4
Rover	companion	6

Are rows uniquely identifiable?
C

dogs

id	name	role	age
1	Artemis	working	2
3	Fido	companion	4
4	Rover	companion	6
5	Roger	working	3
7	Buddy	working	7
8	Daisy	show	3
12	Shep	working	10

Keys

name	breed
Rover	Great Dane
Fido	Dalmatian
Woody	Mutt
Buddy	Poodle
Jones	Great Dane

Here name could be a key

Keys

name	breed
Rover	Great Dane
Fido	Dalmatian
Rover	Mutt
Buddy	Poodle
Jones	Great Dane

The key allows UNIQUE identification of rows.

It can be a **composite key** (more than one column).

There must be a guarantee that the key will always identify rows uniquely.

This is one of the jobs of Postgres: ensuring data integrity.

Here name + breed looks like it could be a composite key

Keys

name	breed
Rover	Great Dane
Fido	Dalmatian
Rover	Mutt
Buddy	Poodle
Jones	Great Dane
Rover	Great Dane

Now name + breed is obviously no good, so we need to add another key.

The key allows UNIQUE identification of rows.

It can be a **composite key** (more than one column).

There must be a guarantee that the key will always identify rows uniquely.

This is one of the jobs of Postgres: ensuring data integrity.

Keys

dog_id	name	breed
1	Rover	Dalmatian
2	Fido	Dalmatian
3	Daisy	Mutt
4	Daisy	Poodle

The SELECT keyword
Choosing columns

We can show all columns from a table:

SELECT * **FROM** dogs

Equivalently:

SELECT dogs.* **FROM** dogs
This is useful later on when queries refer to more than one table.

We can also request just the columns we want:

SELECT name, age, breed **FROM** dogs

Renaming columns

Any columns can be renamed:

SELECT name AS name_of_dog **FROM** dogs

SELECT name AS name_of_dog, id AS dog_number **FROM** dogs

In Postgres you can actually leave out the AS; these queries give the same results as the two above:

SELECT name name_of_dog **FROM** dogs

SELECT name name_of_dog, id dog_number **FROM** dogs

This only renames the columns in the results of your query, not in the actual database.

WHERE: filtering rows by certain criteria

WHERE is used to restrict rows. We will only see the rows which match our criteria.

SELECT * **FROM** employees
WHERE age < 40

You can use the operators <, >, =, !=.

You can also use AND and OR (with brackets for clarity):

SELECT * **FROM** employees
WHERE age > 40 OR (role = 'manager' AND authorised = TRUE)

LIMIT: Restricting the number of rows

The LIMIT keyword restricts the number of rows.

It always comes last.

The LIMIT keyword is very useful for getting a quick look at some example data.

SELECT * FROM orders
If the orders table is very big, this could take minutes to run, and return millions of rows!

SELECT * FROM orders
LIMIT 100
This will run in milliseconds. Why?

Inside the SQL server

The SQL processor's job is to read your query and work out how to produce your result.

This might be a big computational job!

In the end, the SQL server always has to translate your query into *disk commands* for the hard drive or SSD. This is where most of your data is stored (SQL does cache some in RAM, which is faster – but RAM is small and expensive).

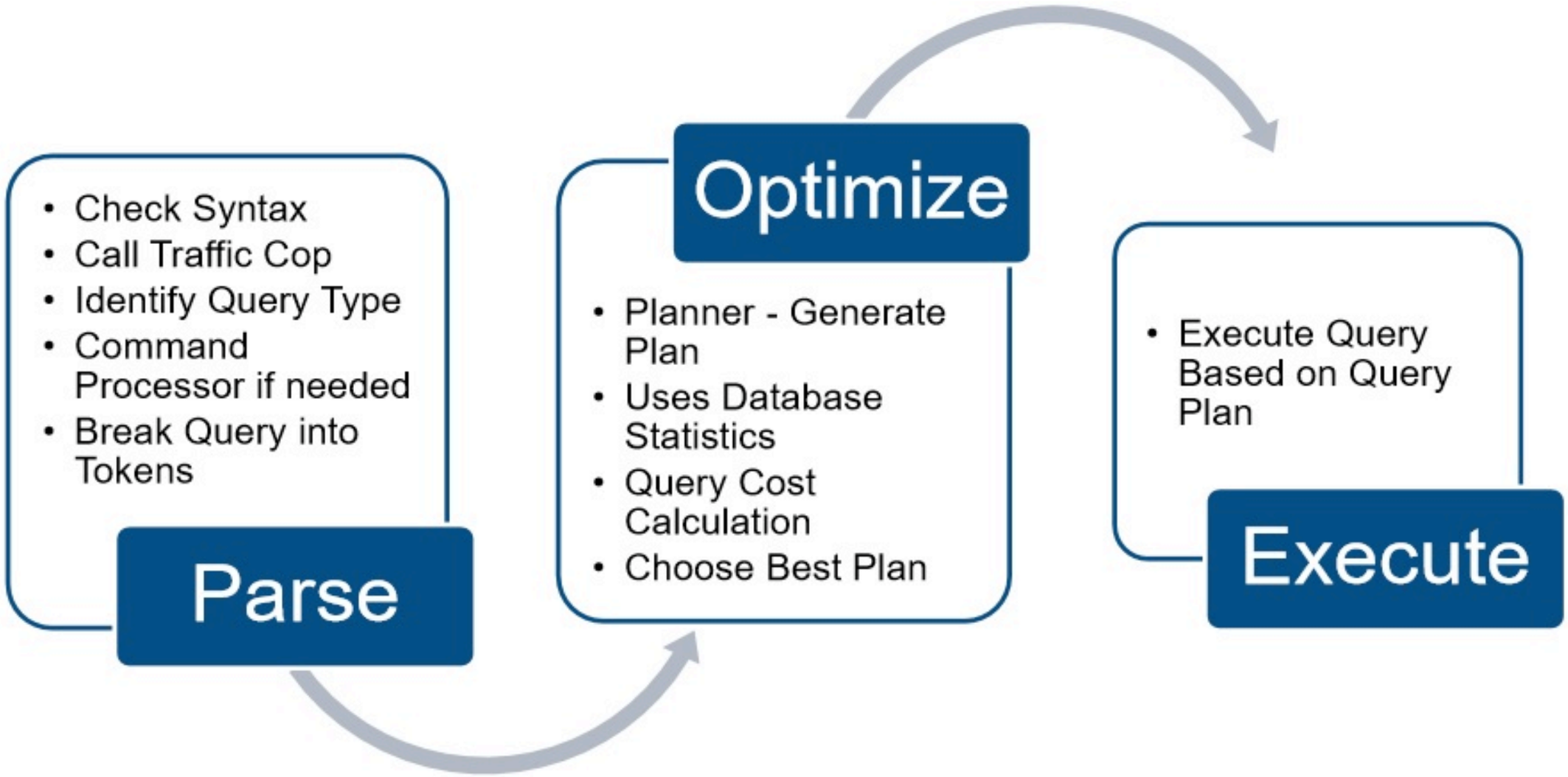
Query optimisations

The SQL processor can make **optimisations** to save time.

For example, if your query says LIMIT 10, the processor will not have to scan the entire table, so the query will be much faster. However, if you request the MAX(age) of all employees, each row has to be checked – the whole table has to be read from disk and checked.

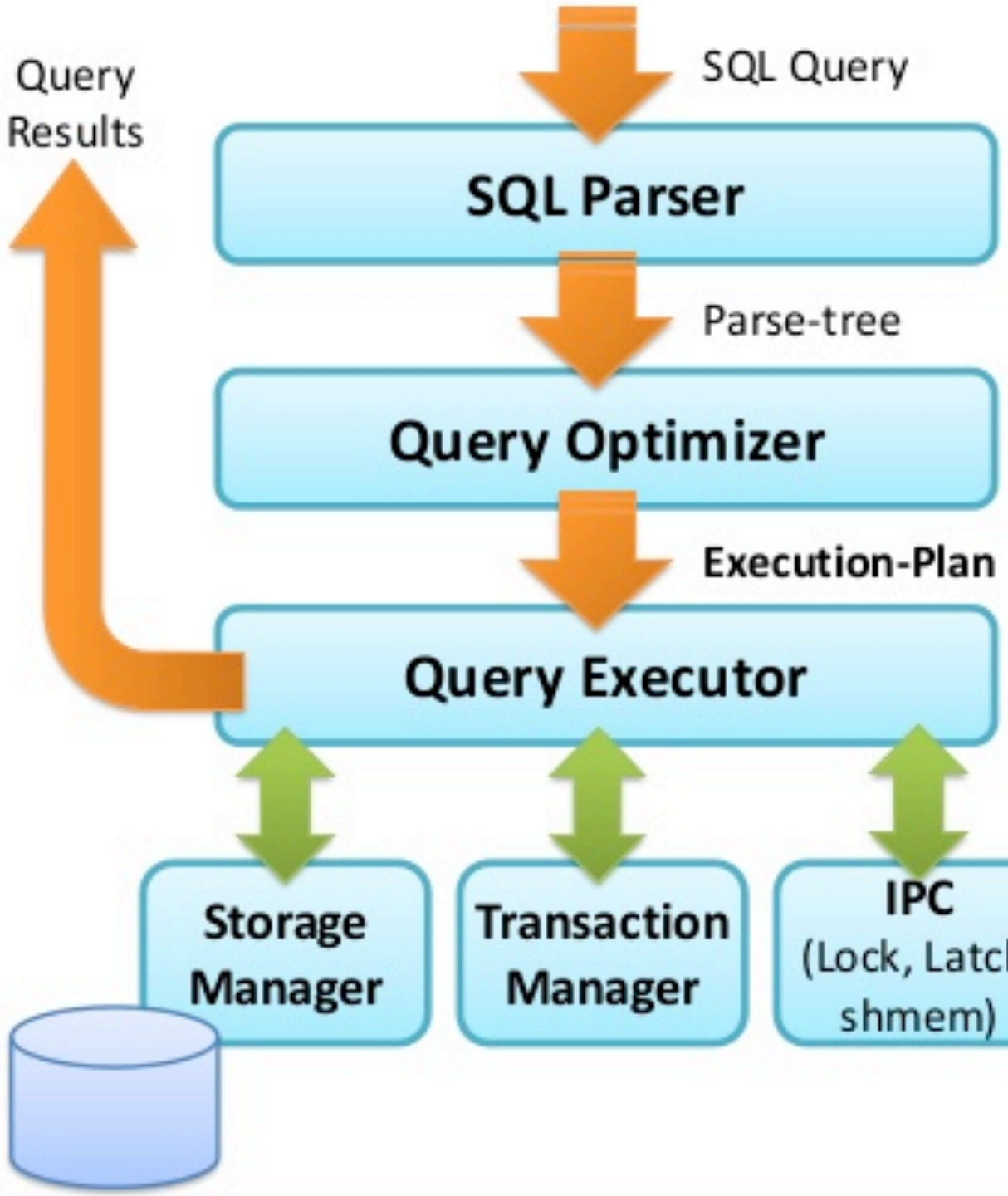
The SQL processor is not as clever as a human analyst. Later, we will see how to optimise queries ourselves.

Inside the SQL server



("The traffic cop is the agent that is responsible for differentiating between simple and complex query commands.")

PostgreSQL Internals



- Parse the supplied SQL, then transform to an internal data structure (parse-tree)
- Detection of syntax errors

- Cost estimation based on the statistics
- Construction of the most optimal query execution-plan

- Run the workloads (like Scan, Join, Sort, ...) according to the query execution plan.
- Utilization of PostgreSQL internal infrastructures; like storage manager

Counting

A note about COUNT

- **COUNT(*)** counts the number of rows, even if they have null cells
- **COUNT(column)** counts the number of rows where that column is not null
- **COUNT(DISTINCT column)** counts the number of distinct values in that column (nulls not counted)

- Usually, you either want **COUNT(*)** or **COUNT(DISTINCT column)**
- Whether you use **COUNT(*)** or **COUNT(column)** can be important in non-inner joins, where rows may have missing values.

Other aggregate functions

An aggregate function is a function which takes a group of rows, and outputs *a single number*.

- SUM
- MIN
- MAX
- AVG
- COUNT
- CORR
- STDDEV
- VARIANCE

https://www.postgresql.org/docs/9.5/functions-aggregate.html

String matching

This is done with **LIKE**. **ILIKE** is the case-insensitive version.

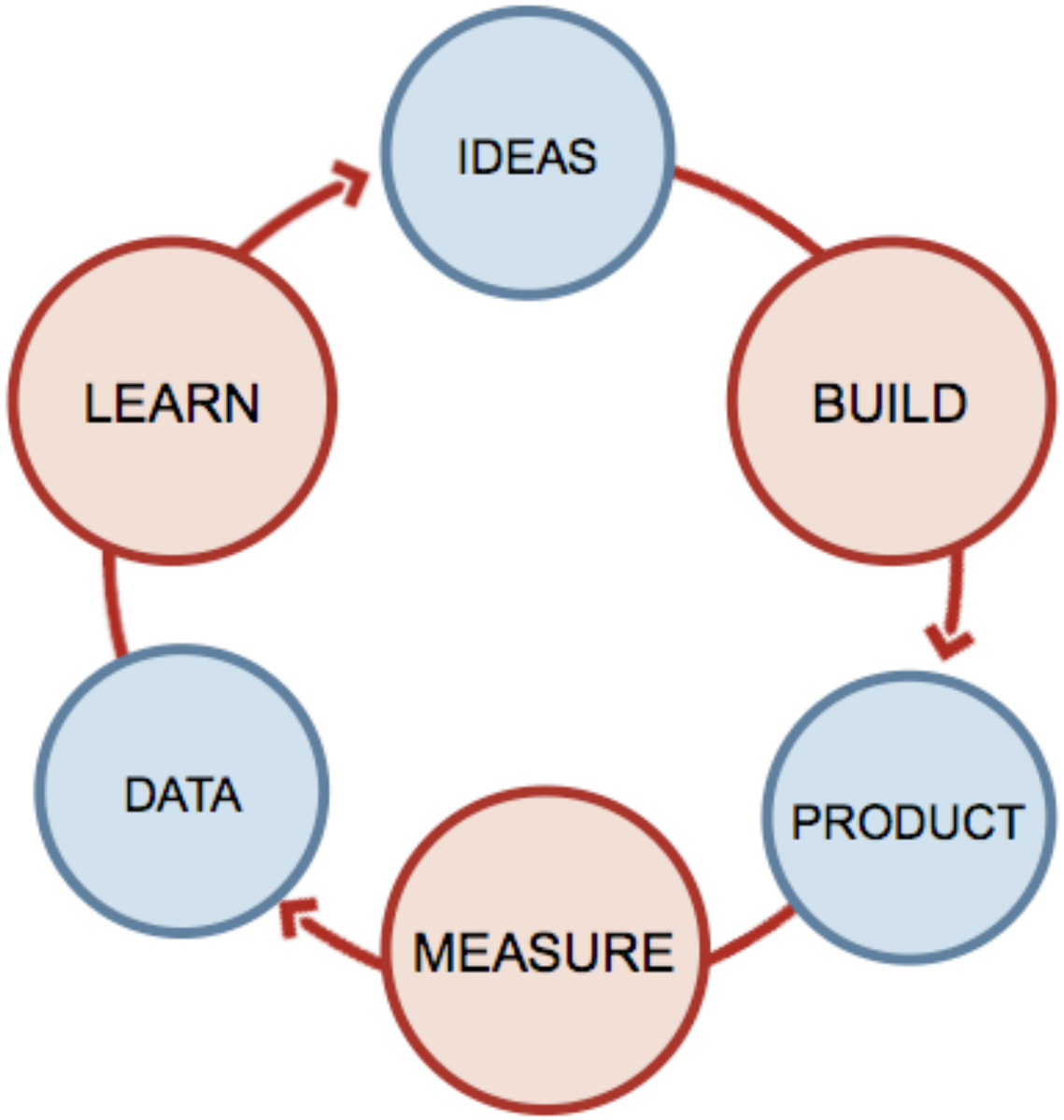
SELECT * **FROM** employees
WHERE name **LIKE** 'Frank'

This is just equality checking! The real power comes with from including wildcards:

1. **WHERE** name **LIKE** '%Frank'
   matches anything, then Frank (words ending with Frank)
2. **WHERE** name **LIKE** 'Frank%'
   matches Frank, then anything (names starting with Frank)
3. **WHERE** name **LIKE** '%Frank%'
   matches names containing Frank anywhere

Developing queries (the development loop)

- Get a simple version of the query to run first
- Check the results
- Extend and tune the query



The structure of a query

A simple query:

- SELECT
- FROM
- WHERE
- ORDER BY
- LIMIT

A more complex query:

- SELECT
- FROM
- **JOINs, each with an ON**
- WHERE
- **GROUP BY**
- ORDER BY
- LIMIT