Imperial College
Business School

Imperial means
Intelligent Business

Lecture 7
Query planning and development

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

Reading

Video lectures:

7.2.2 - Loading data with pg_dump and pg_restore.mp4
7.3.1 - Importing the movies database.mp4

8.2.1 - Types of interview questions.mp4
8.2.2 - SQL interviews (Guest speaker).mp4

Postgres documentation on indexes:
https://www.postgresql.org/docs/10/static/sql-createindex.html
Postgres documentation on EXPLAIN:
https://www.postgresql.org/docs/10/static/sql-explain.html
Postgres documentation on ANALYZE:
https://www.postgresql.org/docs/10/static/sql-analyze.html

Reading

SEARCH and CYCLE keywords allowing recursive queries: https://www.depesz.com/2021/02/04/waiting-for-postgresql-14-search-and-cycle-clauses/

Why SELECT * can be bad for performance https://tanelpoder.com/posts/reasons-why-select-star-is-bad-for-sql-performance/

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

What do rows represent?

Rows can represent:

- People
- Real objects
- Imaginary objects
- Concepts
- Events (sales, rentals)
- Contracts
- Facts
- Debts
- … etc

Getting to know a new database

- Find out what all the tables are called *(here we only have one)*
- In each table, look at the rows: what do they represent?
- In *most* tables, each row represents an entity, person, or object or some kind *(but this is not always true)*
- In each table, look at the columns: what do they represent?
- Keep the tables, rows and columns (the **schema**) where you can see them easily

Columns: attributes (fixed for each table)

Rows: entities or objects (any number of these)

First Name	Last Name	Address	City	Age
Mickey	Mouse	123 Fantasy Way	Anaheim	73
Bat	Man	321 Cavern Ave	Gotham	54
Wonder	Woman	987 Truth Way	Paradise	39
Donald	Duck	555 Quack Street	Mallard	65
Bugs	Bunny	567 Carrot Street	Rascal	58
Wiley	Coyote	999 Acme Way	Canyon	61
Cat	Woman	234 Purrfect Street	Hairball	32
Tweety	Bird	543	Itotltaw	28

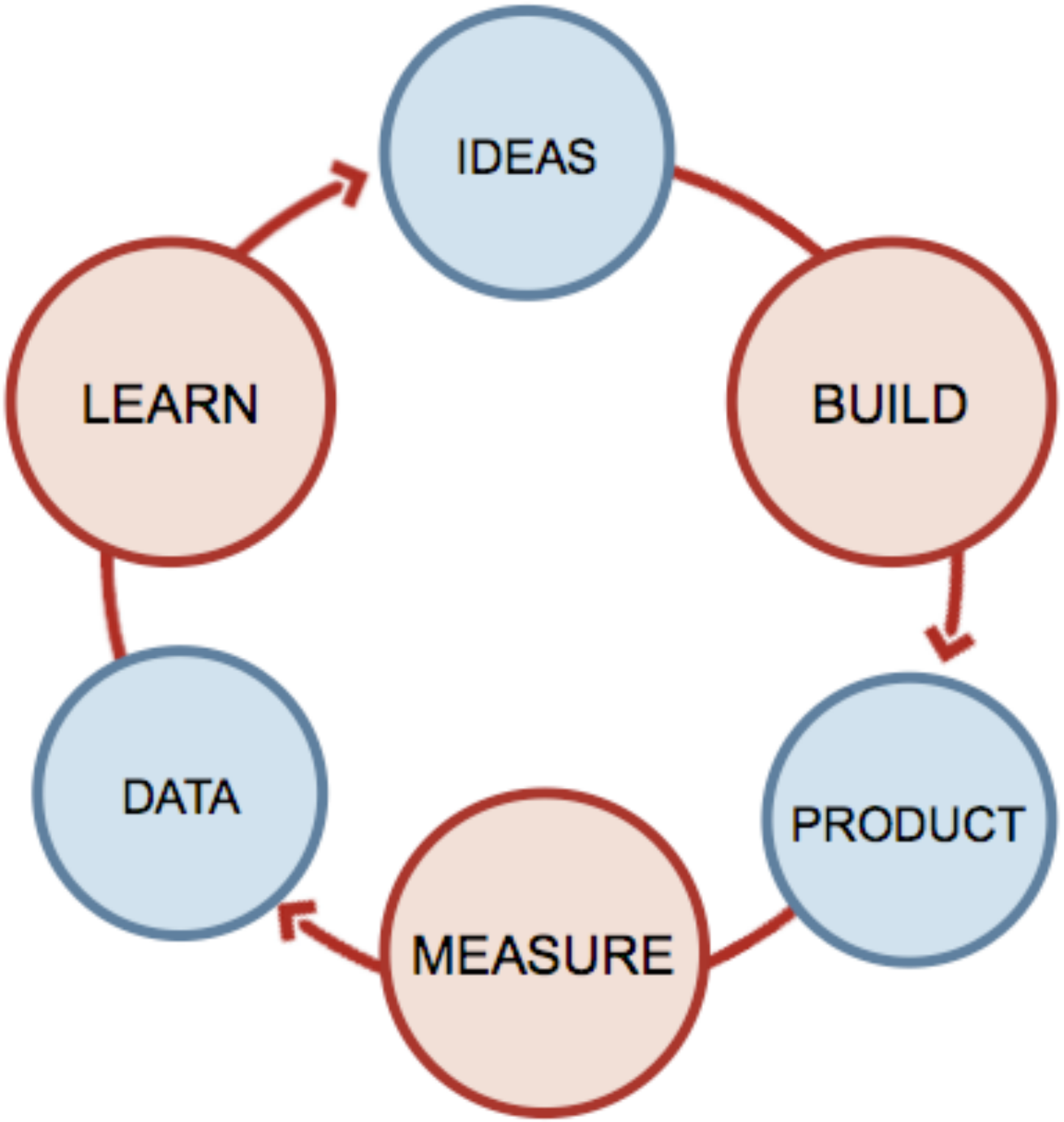
How to write a query

1. Which tables contain the information you need?
   Check you understand what all the tables are for.

2. What result do I want exactly?
   What columns are present in the result?
   Approximately how many rows do you expect to return?
   What does each row represent?

3. What intermediate tables do I need to construct along the way?
   Should I use subqueries, CTEs or views?

The development loop



Always check the number of rows

Whenever you run a query, note the number of rows.

- Is it what you expect?
- Has the row number increased or decreased?

Splitting up queries

- There are three ways to split up queries:

- **Subqueries**
  Smaller queries nested within the main query.

- **Common Table Expressions** (CTEs)
  using the **WITH** keyword
  Smaller queries placed at the top of the main query.

- **Views**
  Saved queries which you can access as if they were tables.

Developing with subqueries

If you are going to use a subquery, start by taking your original query, wrapping it in a subquery and selecting everything. Then move forwards.

Putting the opening and closing brackets on their own on separate lines can help; or you can indent:

```
SELECT * FROM
(
SELECT MAX(rental_date) FROM rental
WHERE staff_id = 2
) AS most_recent
```

```
SELECT * FROM
        (SELECT MAX(rental_date) FROM rental
        WHERE staff_id = 2
        ) AS most_recent
```

Revenue example

Revenue example (Northwind)

Do a month-by-month analysis of revenue growth.

Revenue example (Northwind)

Do a month-by-month analysis of revenue growth.

We start by working our the price of all order details and joining to the orders table.

```
SELECT *, (order_details.UnitPrice * order_details.Quantity) -
order_details.Discount AS price
FROM orders
INNER JOIN order_details ON orders.orderID = order_details.orderID
```

Revenue example (Northwind)

Now we can use this as a subquery orders_with_price

```
SELECT * FROM
(
    SELECT *, (order_details.UnitPrice * order_details.Quantity) -
    order_details.Discount AS price
    FROM orders
    INNER JOIN order_details
    ON orders.orderID = order_details.orderID
) as orders_with_price
```

Revenue example (Northwind)

Now we can GROUP BY to give us the total price of each order.

```
SELECT  orderid, orderDate, SUM(price) FROM

(SELECT order_details.orderid, OrderDate, (order_details.UnitPrice *
order_details.Quantity) - order_details.Discount AS price
FROM orders
INNER JOIN order_details
ON orders.orderID = order_details.orderID) as priced_orders

GROUP BY orderid, orderDate
```

Revenue example (Northwind)

We can extract the month (as a number) with date_part:

```
date_part('month',orderdate)
```

If we want to do a year-on-year revenue analysis, we can't ORDER BY this month column as it doesn't contain the year, so each value of month makes reference to multiple years.
date_part only extracts the month.

How do we group by month AND year?

Revenue example (Northwind)

The **date_trunc** function sets all *smaller* parts of the date to zero, so the year is kept, and all dates within a particular month and year will be set to the same value. This means they can be used with **GROUP BY**.

date_trunc('month', date)

sets everything smaller than month to its smallest value;
here, sets all dates in August to August 1, 00h 00m 00s 00ms

Build up a large query

Let's add our year_and_month column:

Start with just the priced orders:

```
SELECT order_details.orderid, date_trunc('month', OrderDate) as
year_and_month, OrderDate, (order_details.UnitPrice *
order_details.Quantity) - order_details.Discount AS price
FROM orders
INNER JOIN order_details
ON orders.orderID = order_details.orderID
```

Building up a large query

Make this bit into a subquery – add the brackets and alias:

```
(
SELECT order_details.orderid, date_trunc('month', OrderDate) as
year_and_month, OrderDate, (order_details.UnitPrice *
order_details.Quantity) - order_details.Discount AS price
FROM orders
INNER JOIN order_details
ON orders.orderID = order_details.orderID
) AS priced_orders
```

(this won't run yet – we need to SELECT from the subquery first)

Building up a large query

Now we can SELECT from the subquery and do the GROUP BY:

```
SELECT orderid, orderDate, year_and_month, SUM(price)
AS order_price
FROM
(
    SELECT order_details.orderid, date_trunc('month', OrderDate) as
    year_and_month, OrderDate, (order_details.UnitPrice *
    order_details.Quantity) - order_details.Discount AS price
    FROM orders
    INNER JOIN order_details
    ON orders.orderID = order_details.orderID
) as priced_orders
GROUP BY orderid, orderDate, year_and_month
```

Trick: as orderDate and year_and_month are the same for each order, adding them to the GROUP BY won't affect the groups, but will let us SELECT them.

We still want each row to represent an order – we're not grouping by year_and_month yet.

Building up a large query

Finally, we use another subquery to add a window function for a running total:

```
SELECT *, SUM(order_price) OVER(ORDER BY orderdate)
FROM
(
    SELECT orderid, orderDate, year_and_month, SUM(price)
    AS order_price
    FROM
    (
        SELECT order_details.orderid, date_trunc('month', OrderDate) as
        year_and_month, OrderDate, (order_details.UnitPrice *
        order_details.Quantity) - order_details.Discount AS price
        FROM orders
        INNER JOIN order_details
        ON orders.orderID = order_details.orderID
    ) as priced_orders
    GROUP BY orderid, orderDate, year_and_month
) AS orders ORDER BY orderdate
```

Building up a large query

Finally, save as a view:

```
CREATE OR REPLACE VIEW view_orders AS
(
SELECT *, SUM(order_price) OVER(ORDER BY orderdate)
FROM
      (SELECT orderid, orderDate, year_and_month, SUM(price)
      AS order_price
      FROM
            (SELECT order_details.orderid, date_trunc('month', OrderDate) as
            year_and_month, OrderDate, (order_details.UnitPrice *
            order_details.Quantity) - order_details.Discount AS price
            FROM orders
            INNER JOIN order_details
            ON orders.orderID = order_details.orderID
      ) as priced_orders
GROUP BY orderid, orderDate, year_and_month
) AS orders ORDER BY orderdate
)
```

Revenue example (Northwind)

Now we have more power.

We can calculate monthly totals using our view **view_orders**:

```
SELECT SUM(order_price),
year_and_month
FROM view_orders
GROUP BY year_and_month
```

Revenue example (Northwind)

We can also calculate the running total of monthly revenues:

```
SELECT *, SUM(order_price) OVER (ORDER BY
year_and_month)
FROM view_orders
```

Revenue example (Northwind)

We can calculate year-on-year differences using LAG(sum,12), which looks 12 rows ago (1 year ago, as there is 1 row per month)

LAG: look at previous rows
LEAD: look at following rows

LAG(col, 12): look 12 rows ago
LEAD(cl, 12): look 12 rows ahead

```
SELECT *, month_revenue - lagged AS year_increase
FROM

    (SELECT *, LAG(month_revenue,12) OVER(ORDER BY
    year_and_month) AS lagged
    FROM

        (SELECT SUM(order_price) AS month_revenue, year_and_month
        FROM view_orders GROUP BY year_and_month
        )AS t
    )AS t2
```

Query optimisation

Can we make this any simpler?

- We don't need to join to the orders table until we need the order date
- We could do a GROUP BY in the innermost query without having to use a subquery
- priced_orders seems like a very useful intermediate table; we could save it as a view.

Note that we could have saved *any* of the intermediate results as views (with sensible names!) and selected from them.

Query optimisation

Looking at the query plan

EXPLAIN: show query plan

EXPLAIN ANALYZE: show query plan
as well as executing and timing query

https://www.postgresql.org/docs/9.4/using-explain.html

```
EXPLAIN ANALYZE SELECT *
FROM view_orders
```

```
EXPLAIN ANALYZE SELECT *
FROM cumulative_orders
```

a Output | Explain | Messages | Notifications

QUERY PLAN text
WindowAgg (cost=266.66..304.37 rows=2155 width=22) (actual time=4.164..5.041 rows=830 loops=1)
-> Sort (cost=266.66..272.04 rows=2155 width=14) (actual time=4.151..4.330 rows=830 loops=1)
Sort Key: orders.orderdate
Sort Method: quicksort Memory: 63kB
-> Subquery Scan on orders (cost=104.24..147.34 rows=2155 width=14) (actual time=3.205..3.806 rows=830 loops=1)
-> HashAggregate (cost=104.24..125.79 rows=2155 width=14) (actual time=3.204..3.472 rows=830 loops=1)
Group Key: order_details.orderid, orders_1.orderdate
-> Hash Join (cost=32.67..71.92 rows=2155 width=16) (actual time=0.570..2.270 rows=2155 loops=1)
Hash Cond: (order_details.orderid = orders_1.orderid)
-> Seq Scan on order_details (cost=0.00..33.55 rows=2155 width=12) (actual time=0.007..0.524 rows=2155 loops=1)
-> Hash (cost=22.30..22.30 rows=830 width=6) (actual time=0.545..0.546 rows=830 loops=1)
Buckets: 1024 Batches: 1 Memory Usage: 39kB
-> Seq Scan on orders orders_1 (cost=0.00..22.30 rows=830 width=6) (actual time=0.006..0.283 rows=830 loops=1)
Planning Time: 0.573 ms
Execution Time: 5.411 ms

Query optimisation

What takes a long time?

- Sequential scanning through records on the disk
  to find a particular record
  or to find a record matching a criterion
  (can use an index to avoid having to do this)

- Joins (checking the join condition, producing the intermediate table)

Bear in mind that operations often multiply.

- Cross joining a table with 10 rows to itself: 100 rows in the result
- for a table of 100 rows: 100,000 rows in the result

Recent rental example

Recent rental example (dvdrental)

How do we get the most recent rental processed by a member of staff?

```
SELECT * FROM rental
ORDER BY rental_date DESC
LIMIT 1
```

However, what happens if there are two most recent rentals for that member of staff, happening at the same time?

Recent rental example (dvdrental)

How do we get the most recent rental processed by a member of staff?

```
SELECT * FROM rental
ORDER BY rental_date DESC
LIMIT 1
```

However, what happens if there are two most recent rentals for that member of staff, happening at the same time?

One of them will be nondeterministically missed out.

Recent rental example (dvdrental)

How do we get both of them?

We can easily get the *date* of both of them: this query will work even if many records are tied for most recent.

SELECT MAX(rental_date) **FROM** rental
WHERE staff_id = 2

What identifies these records?

- staff ID
- date

This is all we need to uniquely identify these records.

Recent rental example (dvdrental)

So, we can take this data and join it to rentals to get the rest of the information.

First wrap in a subquery and select everything:

```
SELECT * FROM
(
SELECT MAX(rental_date) FROM rental
WHERE staff_id = 2
) AS most_recent
```

Recent rental example (dvdrental)

Now we can join to the rentals table:

```
SELECT * FROM
(
SELECT MAX(rental_date) AS max_date FROM rental
WHERE staff  id = 2
) AS most_recent
INNER JOIN rental
ON most_recent.max_date = rental.rental_date
```

However this joins rentals which don't belong to staff ID 2!

Recent rental example (dvdrental)

So we restrict to staff ID 2:

```
SELECT * FROM
    (
    SELECT MAX(rental_date) AS max_date FROM rental
    WHERE staff  id = 2
    ) AS most_recent
INNER JOIN rental
ON most_recent.max_date = rental.rental_date
AND rental.staff_id = 2
```

Note that we need the restriction in the subquery too so that we apply MAX to the right employee's rentals.

Recent rental example (dvdrental)

This could also be done with WHERE:

```
SELECT * FROM
    (
    SELECT MAX(rental_date) AS max_date FROM rental
    WHERE staff  id = 2
    ) AS most_recent
INNER JOIN rental
ON most_recent.max_date = rental.rental_date
WHERE rental.staff_id = 2
```

There seems to be no difference in performance.

Recent rental example (dvdrental)

What about seeing the most recent rental for all staff?

First we get the most recent rental for each member of staff:

```
SELECT staff_id, MAX(rental_date)
FROM rental
GROUP BY staff_id
```

(this will work no matter how many staff there are)

Recent rental example (dvdrental)

Wrap in a subquery and give sensible names:

```
SELECT * FROM
(
SELECT staff_id, MAX(rental_date) AS last_date
FROM rental
GROUP BY staff_id
) AS last_dates
```

Recent rental example (dvdrental)

Wrap in a subquery and give sensible names:

```
SELECT * FROM
(
SELECT staff_id, MAX(rental_date) AS last_date
FROM rental
GROUP BY staff_id
) AS last_dates
```

Recent rental example (dvdrental)

Now join the last_dates table to the rentals table:

```
SELECT * FROM
rental
INNER JOIN
        (
        SELECT staff_id, MAX(rental_date) AS last_date
        FROM rental
        GROUP BY staff_id
        ) AS last_dates
ON rental.staff_id = last_dates.staff_id
AND rental.rental_date = last_dates.last_date
```

This shows all most recent rentals (no matter how many) for all staff.

Recent rental example (dvdrental)

Finally, we tidy up by joining to inventory and then film so we can get the film title, and restrict the columns.

```
SELECT last_dates.staff_id, last_dates.last_date, film.title FROM
rental
INNER JOIN
                (
                SELECT staff_id, MAX(rental_date) AS last_date
                FROM rental
                GROUP BY staff_id
                ) AS last_dates
ON rental.staff_id = last_dates.staff_id
AND rental.rental_date = last_dates.last_date
INNER JOIN inventory
ON rental.inventory_id = inventory.inventory_id
INNER JOIN film
ON inventory.film_id = film.film_id
```

CSV files

CSV files

Comma-separated-value files are one of the most common formats for data exchange. As a data analyst, CSV files are a core part of importing and exporting data.

CSV files are separated into **lines** by the carriage return character.

The **delimiter** is what splits up a line into cells. Usually it is the comma; the tab character (invisible) can also be used, or you can specify a custom delimiter. The delimiter **must not** appear in the data or errors will result.

There is often a header row with column titles.

CSV files

Example CSV

File Edit View Insert Format Data Tools Add-ons Help All changes saved in Drive

100% $ % .0 .00 123 Arial 10

	A	B	C	D	E	F
1	Email	First Name	Last Name	Company	Snippet 1	
2	example1@domain.com	John	Smith	Company 1	Snippet Sentence1	
3	example2@gmail.com	Ma				
4	example3@outlook.com	Ja				
5						
6						
7						
8						

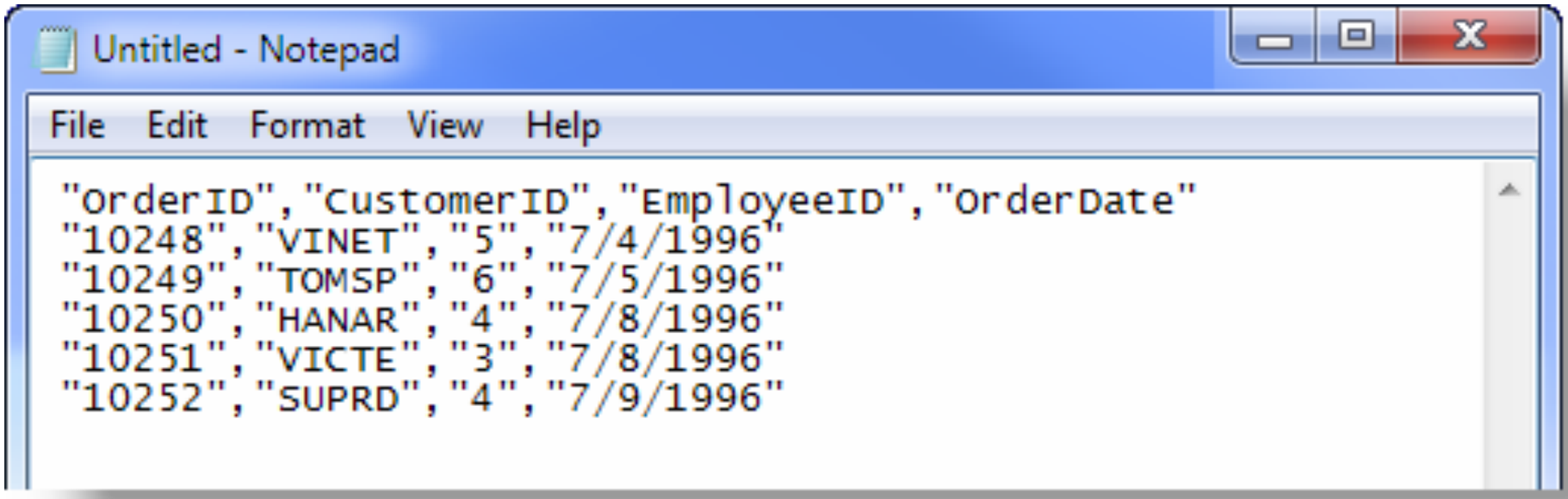
CSV file

Example CSV - Sheet1 — Notatnik

Plik Edycja Format Widok Pomoc

```
Email,First Name,Last Name,Company,Snippet 1
example1@domain.com,John,Smith,Company 1,Snippet Sentence1
example2@gmail.com,Mary,Blake,Company 2,Snippet Sentence 2
example3@outlook.com,James,Joyce,Company 3,Snippet Sentence 3
```

CSV files



Here quotes are used around every value, but this is not necessary.

The comma is used to delimit columns.
The carriage return (invisible) delimits rows.

CSV files

The **COPY** command loads a CSV file into a table which has already been created.

DELIMITER specifies the column delimiter (symbol between columns); usually it is comma or tab.

CSV HEADER says that there is a header row (the first row) which should be ignored.

COPY movie **FROM** '/Users/fintan/Dropbox/Imperial/Databases Online MBA/files/movie_metadata.csv'
DELIMITER ',' **CSV HEADER**;

CSV files

If the **COPY** command does not work, try the **\copy** command.

COPY: SQL command (use in psql or pgAdmin)

\copy: psql command; use only within psql.

\copy has more power and can often resolve permissions issues with COPY.

CSV files

Before loading a CSV file you need to

- have a database ready (make a new one with CREATE DATABASE)

  e.g. CREATE DATABASE kennels

- have a table ready (make a new one with CREATE TABLE)

  e.g. CREATE TABLE dogs (ID integer, name text, breed text)

Do not load any CSV files onto the Imperial database server. Do this only on your own machine.

Backing up with pg_dump

pg_dump dumps your database as a single file.

pg_restore loads a dump file back into the Postgres server as a database.

These are both command line tools like psql, and should be on your PATH.

There are two formats:

- Dump as SQL (default) – here you can open and read the SQL file
- Dump as Postgres binary format (--format c)

Backing up with pg_dump

Dump as SQL:

```
pg_dump dvdrental > dvdrental_dump.sql
```

Load as SQL:

```
psql -d shakespeare -f shakespeare.sql
```

Dump in Postgres binary format:

```
pg_dump dvdrental --format c
```

Restore Postgres binary format:

```
pg_restore -h localhost -U postgres -v -d
shakespeare  shapespeare.pgdump
```

Dump as CSV:

```
COPY film TO '/Users/fintan/film.csv' DELIMITER ',' CSV HEADER;
```

Loading .sql files

A database can be loaded from an .sql file by using the –f option with psql (process file):

CREATE DATABASE shakespeare; (run this in psql, it's an SQL command)

```
psql -d shakespeare -f shakespeare.sql

pg_dump dvdrental --format c

pg_restore -h localhost -U postgres -v -d shakespeare
shakespeare.pgdump
```

Export as CSV:
COPY film TO '/Users/fintan/film.csv' DELIMITER ',' CSV HEADER;

Interview questions

Attacking an interview question

- **Read and understand** the question completely and clearly. Do not move to the next stage until you are **sure** you understand what the question is asking.

- **Look at the schema** (draw out an entity-relationship diagram if there isn't one available) and think about **where the information** you need is to be found. Which tables is it in?

- Think about how to **combine** the required information back together to get the result. Which **language features** will you require?

- Build up the query **in stages**.

- If stuck, go through language features and imagine whether they can help you. Joins? GROUP BY and aggregate functions? Window functions? WITH/CTEs? Subqueries?

Attacking an interview question

- Explain your thought processes as you go along. Mention concepts that you understand.

- Do not name-drop concepts which you don’t understand fully.