Databases Supervision 2 (srns2)

Question 1

Exercise 3a

```
MATCH (p:Person) -[:ACTED_IN]-> (m:Movie)
RETURN p.name AS name, COUNT(*) AS total
ORDER BY total DESC, name
LIMIT 10;
```

This query finds every `ACTED_IN` relationship for each actor and each movie they act in, and counts the number of movies each actor acts in.

Exercise 3b

```
  MATCH (m1:Movie {title: 'The Matrix Reloaded'})
    <-[r1:ACTED_IN]- (p:Person) -[r2:ACTED_IN]->
    (m2:Movie {title: 'John Wick'})
  RETURN p.name AS name, r1.roles AS roles1, r2.roles AS roles2
  ORDER BY name, roles1, roles2;
```

In this query, the person (actor) is in the middle, and an `ACTED_IN` relationship is found to two different movies (The Matrix Reloaded and John Wick), and returns the name of the actor as well as the roles they play in each if they acted in both films.

Exercise 3c

```
MATCH path=ALLSHORTESTPATHS(
    (m:Person {name: 'Steven Spielberg'}) -[:PRODUCED*]- (n:Person)
  )
  WHERE n.person_id <> m.person_id
  RETURN LENGTH(path)/2 AS spielberg_number,
         COUNT(DISTINCT n.person_id) AS total
ORDER BY spielberg_number;
```

In this query, the shortest path between Steven Spielberg and another producer is found using a number of `PRODUCED` relationships between nodes, where the producer isn't Steven Spielberg. The length of the path is divided by 2 as the path goes both ways and is twice as long as the 'Spielberg number', and this is then returned as the Spielberg number. The number of producers with each Spielberg number is counted, and the tally for each Spielberg number is returned.

Question 2

This can be rewritten as `(a is null) or (b is null)`. This is because `(a and b)` would evaluate to `null` if either `a` or `b` is `null`, so checking whether either of them is `null` has the same effect.

Question 3

If neither `a` nor `b` is null, then we are effectively back to boolean logic, so the expression `not (a and b)` has the same value as `(not a) or (not b)`. We must consider what happens when either `a` or `b` is `null`. If either `a` or `b` is `null`, `a and b` evaluates to `null`, so `not (a and b) = not null = null` too. If `a` is `null`, `not a` evaluates to `null`, and equivalently for `b` and `not b`. Therefore, if one of `a` and `b` is `null`, one of `not a` and `not b` will be `null`, so `(not a) or (not b)` will also be `null`. Therefore, these two expressions both return `null` if `a` or `b` is `null`. Hence, these two expressions have the same value in 3-valued logic.

A similar approach can be used for `not (a or b)` and `(not a) and (not b)`. If `a` and `b` are both not `null` then these expressions have the same value as this is boolean logic again. Now consider if one of them is `null`. `a or b` will evaluate to `null`, as will `not (a or b)`. One of `not a` and `not b` will also evaluate to `null`, and hence so will `(not a) and (not b)`. Therefore, these two expressions have the same value in 3-valued logic.

`a or (not a)` is not strictly true in 3-valued logic. If `a` is `null`, then this will return `null`. To make this always true, it needs to be extended to `a or (not a) or (a is null)`, which will always be true as `null or true` is `true`.

Question 4

- A pro of Cypher's approach is that if some nodes have specific properties that not all nodes of that type have then you don't need a whole new column (like you have to in SQL) with most of the records being filled with `null` in that field, so it is more memory-efficient and a more intuitive approach.
- Another pro of Cypher's approach is that if the structure and properties of the data you need to store change you don't need to recreate the database with a new schema, you just add properties onto the nodes, while SQL would require a new database schema.
- A con of Cypher's approach and pro of SQL's is that in SQL the data is far more organised into tables, and when you see two nodes of the same time you can be sure that they have the same properties.
- Another pro of SQL's approach is that it forces you to stick to a strict E-R model which simplifies what the data looks like, and helps you keep it more organised.

Question 5

It wouldn't make sense to move all SQL-based applications to Cypher. While Cypher is more powerful when analysing relationships, SQL is closer to relational algebra and is likely far more optimised and efficient. Furthermore, SQL is the standard for databases and has APIs and libraries for virtually any language you wish to use to interact with it, particularly for web applications. Cypher is far more powerful when it comes to analysing the relationships between nodes, and would therefore be a better choice for applications such as data science and statistics.

Question 6 (2008 Paper 6 Question 8)

(a)

A safe query in relational algebra is a query that is guaranteed to return a finite number of results, and won't keep going to infinity under any circumstances.

(b)

(i)

Maximum: `rs` (when every `B` in `R` matches every `B` in `S`)
Minimum: `0` (when every `B` in `R` matches no `B` in `S`)

(ii)

Maximum: `rs`
Minimum: `0`

(in the same situations as in part (i))

(iii)

Maximum: When every `B` in `R` matches a `B` in `S`, the second part is empty, so the size of this expression is the size of the first part, `r`.
Minimum: When no `B` in `R` matches any `B` in `S`, the size of the result of this entire expression evaluates to `0` (the second part - the part in brackets - is equivalent to the first part).

(iv)

Both the maximum and the minimum here are `r`, as the left part of the Venn diagram is shaded.

(v)

Here, the maximum and minimum are both `rs` since all of `R` and `S` are included.

(c)

The third one may be different. For example, consider the following:

R

A	B
-3	2
-4	2
-5	3
-6	3

S

B	C
2	15
2	16
3	17
3	18

Let `b = 3`. The first and second expressions both evaluate to:

A	C
-5	17
-5	18
-6	17
-6	18

The third expression evalutes to:

A	C
-3	15
-4	15
-5	15
-6	15
-3	16
-4	16
-5	16
-6	16