1. Discuss the practical exercises for graph databases. Do you understand the queries and what they are doing?

   For the first query; the task is to select all actors who have co-acted with Jennifer Lawrence more than once and output their names and how many times they have co-acted with her.
   The way to do this is to select all the actors who have co-acted with Jennifer Lawrence, group them by name (using a with clause) and return only the actors who have more than one occurrence.
   Because all the actors who have been selected have 1 or more occurrences, we can use "<>" rather than ">".

```
match (p1:Person)-[:ACTS_IN]->(:Movie)<-[:ACTS_IN]-(p2:Person)
where p1.name = 'Lawrence, Jennifer (III)'
  and p2.name <> 'Lawrence, Jennifer (III)'
with p2.name as name, count(*) as total
where total <> 1
return name, total
order by name, total;
```

   The second query asks us to find the distance between every genre and every other genre. The query to do this first matches all genres. Then works out the shortest path from that genre to every other genre. This, however will contain paths from g1 to g2 **and** from g2 to g1. To remove this we use an ordering and only take the relations which satisfy that (alphabetical) ordering (where genre1 < g2.genre). Then, since every path has two HAS_GENRE relationships between them for every film in the path; the length is length(path)/2.

```
match (g:Genre)
with g.genre as genre1
match path = allshortestpaths( (g:genre)-[:HAS_GENRE*]-(g2:genre) )
where genre1 < g2.genre
return distinct genre1, g2.genre as genre2, length(path)/2 as length
order by length desc, genre1, genre2;
```

   The third query asks us to calculate a "similarity" score between two movies based on the number of genres and actors that the movies have in common.

   To do this we first have to establish all the movies which have actors and genres in common. We can do this by two consecutive match clauses. The first will select all movies which have a common actor, the second will select all movies from the first set which have a common genre. The third match clause then filters out all those movies in that group which do not have a common keyword.
   The with clause renames the variables so that they can be returned in the next line. The only thing of particular note is count(). This returns the number of distinct paths between m1 and m2 in the path it is passed. In path1, count(path1) represents the number of common actors while count(path2) is the number of genres that the two movies have in common. There is no need to count path3 – matching it is enough to filter out any movies which do not have any keywords in common.

```
match (m1:Movie)
where m1.title = 'Skyfall (2012)'
match path1 = (m1:Movie)-[:ACTS_IN*2]-(m2:Movie)
match path2 = (m1:Movie)-[:HAS_GENRE*2]-(m2:Movie)
match path3 = (m1:Movie)-[:HAS_KEYWORD*2]-(m2:Movie)
with m2.title as title, (10 * count(path1) + count(path2)) as score
return title, score
order by score desc;
```

I'm aware that this query will not return movies which do not have BOTH a common actor and a common genre. But: I spent a substantial amount of time on this and am happy that this does something close enough for me to talk about it.

2. In SQL, the creator of a database needs to define the schema up front: that is, what columns each table is going to have, and the datatype of each column. This means that all rows in the table have the same set of columns (although some of them may be set to NULL if the value is unknown).
   On the other hand, Cypher does not have an explicit schema: a node can have any set of properties, and you can always add a property with a new name to an existing node.
   Discuss the pros and cons of these two approaches.

   In SQL; defining the schema on creation means that the database is very rigid and inflexible. This is good if the purpose of the database is known before creation. However, it does limit the table in certain situations – if there are lots of optional fields then the database will either have to create many tables or accept a high proportion of NULL values. This *can be* okay however is either wasteful of space OR means queries will require lots of joins.
   Take the example where a movie has alternative names in different countries. In a SQL database the options are either to create a new table containing the fields

   ```
   ALTNAMES (\overline{movie_id}, \overline{country_id}, alt_name)
   ```

   or to add fields onto the movies table for each country for which the movie could have an alternative name for.
   The first approach would be preferred by OLTP databases (since it has lower redundancy and is easier to write to) while the second would be preferred by OLAP databases (since it doesn't require joins). However, both have downfalls and are not ideal.

   Compared to a graph database which can add any properties to a node: if the movie node wants to have an alternate name for that country then it can simply add a new property with the alternate name. This approach does not fill the whole node with NULLs or force expensive joins for basic queries.

   SQL's schema enforcement does have positives too: a well-designed schema can result in very low data redundancy. It's impossible to implement the same thing using Cypher since users can simply add properties to any node – if this data is stored elsewhere in the database then there is redundancy.

   SQL also is more resilient to user-error. Take an numeric field (ie for the movies table: "star_rating"). In a SQL database this would be integer. So any other data type would not be allowed. This means if a user was to input totally invalid data (say "four"), the database would not allow it to be entered. This keeps the database in a consistent state.

   However, with a graph database since properties are defined individually by the user they could create the property "star_rating" and set it to "four". Despite this data being invalid, since the graph database did not have a defined schemaa it would be accepted.

3. In the graph databases tutorial we showed a Cypher query for find the number of co-actors for Jennifer Lawrence. How would you write the same query in SQL?

   ```
   select distinct p1.name
   from people as p1
   join plays_role as pr1 on pr1.person_id = p1.person_id
   join movies as m on m.movie_id = pr1.movie_id
   join plays_role as pr2 on pr2.movie_id = m.movie_id
   ```

```
join people as p2 on p2.person_id = pr2.person_id
where p2.name = 'Jennifer Lawrence'
order by p1.name;
```

4. We discussed several examples of Cypher queries that have close equivalents in SQL, and some examples that are hard to express in SQL. Discuss: would it make sense to move existing SQL-based applications to Cypher? In what circumstances would you choose one query language over the other?

   While Cypher has positives when compared to SQl: I believe that SQL is better suited for most applications.

   In general: most of the largest queries executed on SQL-based databases are done for business analytics on OLAP databases. These sort of queries aim to only touch each fact once and then proceed. Graph-oriented databases do not offer any speedup on analytics such as these. They would likely slow-down the queries – on a sufficiently large database (such as one which analytics queries would be executed on), you cannot load the whole database into RAM. In a SQL-based application you load the facts which you are currently processing into RAM. However, for a graph-database you would not be able to know in advance which nodes connect to the node you are analysing. So would be unable to load them into RAM. This means that each node would require multiple fetches from main memory – greatly slowing down the query.

   For some queries Cypher is far faster than SQL-based applications. It is especially good at finding routes from one node to another – something which SQL requires many $\Theta(n^2)$ joins to do. However, this sort of query is uncommon and there is very little demand for it in industy – so little that recursive queries were not included in the SQL standard until 1999.

   Another argument for not changing existing SQL-based applications to Cypher is that SQL is well-known. Cypher is nicher and there would be large integration issues. In addition to this: there are many different SQL providers all of which would require a different implementation to change their product to a graph-oriented database.

   SQL-based OLTP databases are faster to write to and update than graph-based applications. For most day-to-day usage all you care about is the speed of writing and updating and in this regard SQL is superior to Cypher.

   Since SQL defines the schema when creating the table; a good OLTP implementation enforces low data redundancy. This reduces the amount of data that you have to store when compared to a graph-based database. It also removes the possibility of inconsistent states – again something which Cypher struggles to do.

   Also: since SQL is very established many companies who offer SQL databases have very optimised implementations – which collect data about the database and can greatly decrease the amount of time taken on many queries. While Cypher can also be optimised: in general it is not optimised to the same extent – and collecting analytical data on the database would be impractical since properties are defined when creating the nodes.

5. In the example graph database, there are 22 nodes of label Genre, but in the relational database, the genres table has 273 rows. How do you explain this discrepancy? What are the consequences of this difference in data model?

   In the graph database, different "HAS_GENRE" relations point to the same genre node. This means that the number of genre nodes is equal to the number of genre relations.

   While in the *old, bad and totally un-normalised* relational database schema: each genre has a foreign key pointing to one movie associated. This means that in the relational database: the number of records in the genres table is equal to the number of genre

relations NOT the number of genres. In short: this discrepancy is caused by a bad data model.

The consequences of this difference is that the relational database is very un-normalised and stores many duplicate copies of the name of the genre. So there could be a situation in the relational database where the name of a genre changes and the whole table would be in an inconsistent state, or where the name of one genre was input incorrectly. This problem would not happen with the graph database.

In more general differences between the graph database schema and the relational database: the graph database also makes it easier to query for movies with a specific genre. In the relational database, you have to search through the genres table for a record with a specific genre, then search the movies table for the movie with that movie_id. This will take time Θmn where m is the number of movies and n is the number of movies with that genre.

While in the graph database you only have to navigate to the genres node and then select all movies who have a pointer to that node. This will take $\Theta(n)$ time with respect to either (depending on the implementation) the number of movies in the database OR the number of movies with that genre. Either is better than the time complexity of the same query in the relational database.

6. What do you think the database software is doing internally when you ask it to find the shortest path between two nodes? Describe it in words (no code required).

   To find the path; the software must search the graph. It can either do this by a breadth-first-search or by iterative deepening. The space complexity of a breadth - first-search may be prohibitive for larger graphs though. So I think it likely that the software would use a heuristic to decide whether to use an iterative deepening strategy or a breadth-first-search.

   The software first uses some heuristic to find out whether the branching factor and the depth of the tree will make the space complexity of a bfs prohibitive. If not: the software will perform a bfs until it finds a path between the nodes. If the space complexity of a bfs is prohibitive, then the software would run a series of dfs's using an iterative deepening strategy to search for the shortest path between two nodes.

   Due to the nature of a breadth-first-search; any path found using one is guaranteed to be a shortest path (since all possible shorter paths have already been searched).
   However, the iterative deepening strategy may lead to cases where the first path found is not the shortest path. This would only occur if the depth was increased by more than one at each iteration.