Notes – first part of the presentation (Hans Christian Gregersen)

**Intro**
Neo4j is a graph database management system (GDMS) that both stores and processes data as graphs. Neo4j offers several advantages when we want to model application domains that are highly connected, variably structured, and dynamically changing.

In our presentation, we were covering:
1) The motivation for using Graph Databases instead for example a traditional RDMS
2) The graph data model
3) The query language Cypher
4) The internals and implementation of Neo4j
5) APIs and Tools
6) A demonstration of an app built with neo4j

There are several examples of graphs with such properties in the real-world:

Social Networks such as Facebook and LinkedIn are most intuitively modelled as a graph. They are highly connected, and variably structured as they might support a vast array of different types of relationships. The application domain is also constantly changing as development teams are introducing new features all the time.

Other examples include telecom networks, the web and recommendation systems.

The **motivation** for ditching RDMS in favour of graph databases like Neo4j are many.

Firstly, RDMS stores and models data as tables analogous to the way in which files and records are stored in physical archives. When we're interested in the connections between records more than the actual records themselves, RDMS becomes problematic both from a modelling and performance point of view.

From a modelling point of view, it obscures the model that relationships are not conspicuous, but rather inferred from foreign keys and intermediary tables. In graph databases, relationships are *first class citizens,* and occupy a role just as prominent as the records themselves. It also introduces a lot of unnecessary complexity both conceptually and computationally when the RDMS designer needs to introduce intermediary tables to capture relationships. Furthermore, the E/R model is an unnatural discourse in the context of graphs as illustrated by the fact that it has to be 'learned' by stakeholders. It can be argued that when modelling a graph, the best option is to use a graph….

From a performance perspective, RDMS struggles with queries on relationships as they require the expensive JOIN operation.  As we'll see later, recursively applying JOIN for queries on "friends of friends of…" will quickly lead to sloppy performance and at worst intractable computations.

Finally, schemas are problematic when dealing with variably structured and dynamically changing data. If we want to introduce a new type of relationship in e.g. a social network, then we're required to make changes to the schema, which is both risky and cumbersome. Furthermore, it's hard to design a schema that fits everything in a variably structure domain, and often this means introducing schemas that are too brittle, which in turn complicates future migrations.

**Performance case and example**

Consider a social network which amongst other types of relationships support a 'friendship' relation. Suppose that 'friendship' is not necessarily a symmetric relation e.g. "Bob is friends with Alice" does not imply that "Alice is friends with Bob". In a relational database, the implementation of this type of relationship would normally require two tables. A person table, and a personFriend table (see slide 5 for the tables and the queries).

If we want to find everyone who Bob considers a friend, we would need to join the person table with the personFriend table filtering on name='Bob', which is both syntactically and computationally an easy thing to do. However, if we're interested in the reciprocal query "who considers Bob a friend", we still don't need any advanced syntax, though computationally it's a lot harder as we need a full-scan of the PersonFriend table. So reciprocal queries are hard in RDMS. This is typically addressed using indexing, yet as we've seen in class indexing is a complicated business and requires that you know in advance what type of queries you expect to be most frequent.

and it gets worse if we make queries such as:

"Who are the friends of my friends?"
or
"Who are the friends of my friends of my friends…"

In SQL, we construct such queries by recursively joining tables, but this significantly raises both the syntactic in computational complexity of the query. In fact, Partner and Vukotic demonstrated it in their book *Neo4j in Action* that graph databases are substantially quicker at handling these types of queries.

In their experiment, they stored a social network of 1,000,000 people each with an avg of 50 friends, and tested the execution time of RDMS vs. Neo4j. The results are listed in table in the slides, and it was clear that already at depth 3 i.e. a query of "friends of friends of friends" it the execution time for the RDMS was more than 30 secs, whilst Neo4J could handle this in less 0.168 seconds.

**The Graph Data Model**

In the slides we showed an example of a graph modelling a social network and amongst the relationships in the graph was ‘friend_of’. In constrast to the relational example, the “people who consider Bob a friend” is not more complicated than the reciprocal query. This is because queries like these are expressed easily in terms of patterns on paths through the graph, and because querying a graph (walking paths) is much faster than joining tables. In addition, we only need to look at the relevant parts of the graph, as is further elaborated by Ivan covering the internals.

Neo4J is based on the **labelled property graph model:**

A graph consists of nodes, relationships, properties and labels.
A node contains properties similar to fields in RDMS, and may contain one or more labels for categorisation.
Relationships represent the edges between nodes. Every relationship has a direction, a single name, a start and an end node and may also have properties.

This data model allows us to store connected data as connected data. Returning to our earlier example, it’s easy to see how graphs are the most natural way of expressing a relationship like friendship, which in addition also facilitates some performance advantages on certain queries.

The graph database solution is better for modelling graphs as they offer an intuitive model where relationships are explicitly stated and easy to understand. They offer better performance as the implementation is accomplished by walks through the graph. They offer flexibility and agility as graphs are naturally additive, meaning that we can add new types of relationships and nodes to the graph without disturbing the outcome of existing queries. We’re also not restricted by any schema and this allows us to dynamically adapt to a potentially changing application domain.