Internal data structure:

Graph db performance comes from the fact that it uses native graph storage. It means that graph database is not using any underlying global index to associate nodes but stores nodes in a graph like manner. We say that graph database has native processing capabilities if it has **index-free adjacency** which means that each node has a micro-index of other nearby nodes.

Let's look at some files that are located in neo4j data/garph.db directory. We immediately notice that they are all binary and there is a clear hierarchy of files. We have .db, .db.id, .name, .index and others. These files are **store files** and are divided into node store files, relationship store files, label store files, property store files. Each store file is composed of **fixed size records**. The size of record varies depending on the file store. For example, record size in node store file is 15 byte and record size of relationship store is 33. These values may change depending on neo4j version. Note that because our record size is fixed within store file it is possible to access any record in O(1) if we know record id.

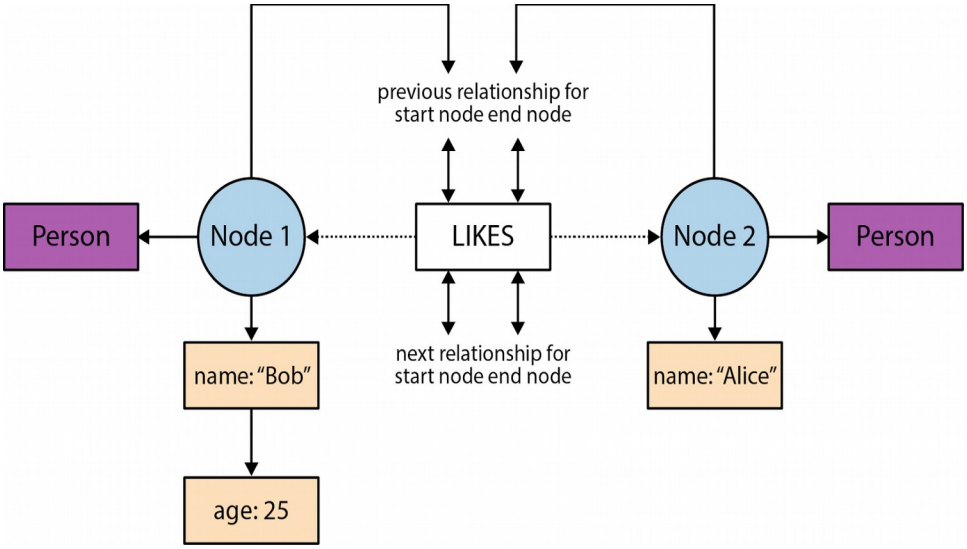

fig 1

Let's look at node and relationship store files since they are the files that define a structure of our graph. Node store file with .db extension is made of fixed size records. Each record has inUse flag, id of relationship, first proprety id, and id of a first label, and some extra flags that are used for different neo4j optimizations.

Relationship store file has a pointer to first node, second node, to the previous and next relationship of the first node and second node. Note that relationship is implemented as doubly linked list on a disk. We have also a pointer to relationship type which is used to determine relationship name and we also have a pointer to first property. From this it follows that our relations and nodes can be similar and still have different properties.

Property store can handle all primitive types of Java, strings and arrays of strings and primitive types. It is also made of fixed size records. It has a pointer to a type of value which it stores and then a pointer to dynamic store where variable size values such as strings and arrays are held. Primitive types are stored directly in property record. Note that property record has a link to the next property so it is implemented as a linked list.

Dynamic stores are used to store strings and arrays. Theses store files are also composed of fixed size records but of bigger size than any other. These records are implemented as doubly linked lists and can be accessed by neo4j only at the beginning of the list. In other words, id manager will give id of the first block in a chain.

Id manager recycles ids of records that has been deleted and gives new ids if there are no ids to recycle. Recycled ids are kept in .id file for fast access. This file will never be empty and will have at least one id available. In case .id file is corrupted, it can be rebuilt by scanning .db file and checking where inuse flag is set to zero.

Example in fig 1:

We get our node. From this node, we can access at O(1) list of properties, list of labels, list of properties, and list of relationships of the node and its neighbors. The last part is true because each relationship stores pointers to relationships of first node and to relationships of second node.

Disclaimer:

- These notes should be used in conjunction with slides since slides have more technical information on actual implementation of different records. I would recommend to read the slides to have an intuition about internal functioning of neo4j and then jump to slides to explore how for example property name is stored or how id files are structured.