Neo4j

Background

Sometimes, certain data is best represented by a graph - this includes things like social relationships (Facebook friends), map data, or even the results of web crawler links. As a result, certain graph databases have been developed. One of the most popular is Neo4j, which has a very unique architecture.

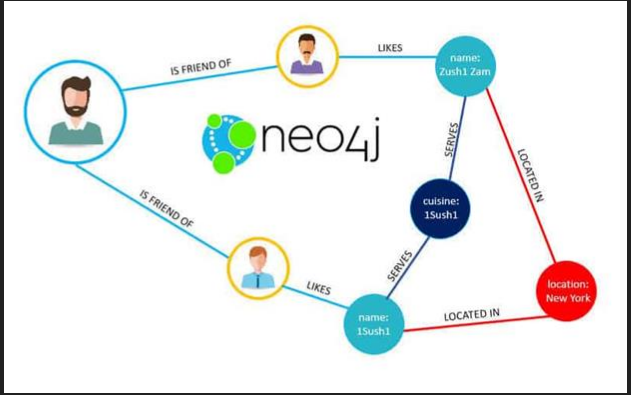
Data Format

Nodes: Points in the graph

Properties: Key value pairs on nodes, do not need to follow a particular structure

Relationships: Directed edges on nodes

Labels: Information corresponding to edges, can also be queried on



Native vs. Non Native Graph Databases

Unlike certain graph databases, which are essentially a graph query language used as an abstraction on top of an existing distributed database, Neo4j is actually known as a native graph database.

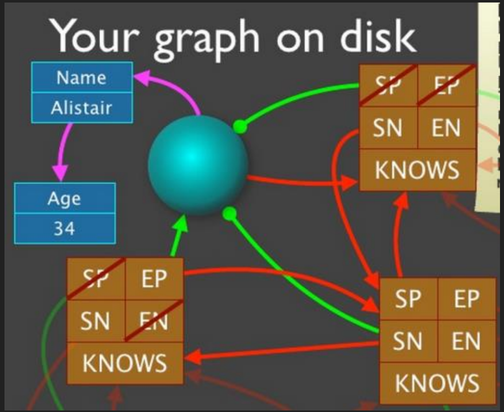
By representing data in its actual graph form on disk, Neo4j is able to have significant performance improvements when dealing with the data.

Data Storage

Nodes, relationships, labels, and properties all kept in different files. Want to optimize for storing similar relationships close to one another on disk for fast traversals.

Each node contains a pointer to its first relationship (which contains pointers to the next relationship from that node), effectively acting as a mini index for them. In this way, the data is stored as tons of linked lists on disk.

This is known as index-free adjacency.



Avoiding Indexes

Recall: If all nodes are being added to an index, finding a node in an index scales logarithmically with the size of the dataset. Hence, every single graph query will take longer if using a huge graph.

By using index-free adjacency, we can ensure that the speed of a query depends only on the number of nodes involved in it, as opposed to the size of the whole graph.

Non native graph solutions have to use many indexes to link nodes together with relationships and as a result, they become slow.

Challenges with Graph Data

We need ACID properties:

Modifying a relationship is not an isolated operation - it also often requires modifying the nodes on both ends of a relationship. As a result, Neo4j has been forced to use a write ahead log for atomicity, as well as locking in order to ensure that two transactions do not get jumbled up, leading to an inconsistent graph.

This becomes even more problematic on a distributed scale, where potentially multiple nodes in the graph that are being modified are on different partitions in a cluster.

Distributed Transaction Ordering

If a two transactions simultaneously overlap on multiple partitions, all involved partitions need to order them the same way (and so do their replicas!).

To do this, Neo4j in a distributed setting uses a very clever method of figuring out which partitions (if any) multiple transactions overlap on, and use one of them as a sort of transaction coordinator in order to ensure that all involved partitions commit at the same time. This way not all transactions need to do this expensive ordering operation, but only overlapping ones.

For replication, Neo4j uses Raft to ensure graph consistency.

Conclusion

Graph databases have become immensely popular as of recent, because there is tons of deeply relational data where scanning the points themselves is an afterthought. As a result, using something like a native graph database can avoid using a centralized index on every single traversal iteration, and therefore greatly speed up the process of gaining insights.