Neo4j: A NOSQL Graph Database Management System

Yuen Hsi Chang

In the 1970s, Edgar F. Codd publicized the relational model, a database model wherein data is represented in terms of tuples, grouped into relations. This model is built upon relational algebra, a theoretical basis which consists of five primitive operators (selection, projection, Cartesian product, set union, and the set difference), and allows queries that model data in databases to be defined. The relational model gained widespread popularity, is generally preferred over its alternatives, namely the hierarchical model and network model, and appropriately handled the first wave of data storage applications in the twentieth century.

In the twenty-first century, applications for data storage increased drastically, and volume of stored data, as well as the frequency in which data is accessed, both skyrocketed. Traditional relational databases, which do not take advantage of todays cheap storage solution and processing power, are getting less appealing as a new variety of database systems emerges. NOSQL, or Not Only SQL, have many types, and is motivated by the concept that DBMS should be picked according to the type of stored data, and the desired performance; generic SQL is no longer all-encompassing.

There are mainly four data models NOSQL databases categorize into, namely key-value store, document store, wide column store, and the graph database model. This paper focuses on Neo4j, a DBMS that implements the graph database model, and aims to demonstrate some advantages Neo4js implementation has over the traditional relational models implementations. Being a data model that is still under development, this paper will also discuss some of its shortcomings, as well as Neo4js potential improvements.

Graph databases, a data model based on graph theory, use nodes, edges, and properties to represent and store data in a graph-like structure, and aim to make the relational model more expressive and meaningful. For instance, consider the traditional relations Person(**p_id**, name) and Car(**license_no**, brand). A person can own cars, sell cars, or get hit by cars, and in order to make the relationship between these two relations explicit, a third relation, such as Owns(**p_id**, license_no), has to get created. In the graph data model, such relationships are made explicit by having two separate nodes for the person and for the car, and connecting them using a directed, named edge. For instance, we can have (John, 1) pointing to (430, Farrari), with an edge labeled Owns, such that a separate relationship set is not required to connect the two entities. This way, the human reader can gain insight and see the connection between the Person and the Car at a glance, without having to read through relationship sets and attribute names.

The graph-like structure also allows the human database manager to easily structure his data model in a clear and presentable manner. Traditionally, designing a database using the relational model requires constructing an Entity-Relationship Model, an abstract diagram that describes the dependencies the database has to handle, and meticulously reducing the E-R model to a relational schema. Graph databases avoids this step, as graph-like structures are themselves very whiteboard friendly; implementing a graph schema is a direct translation process, where one simply parses through the hardcopy data model of interconnected nodes.

We now shift our focus unto Neo4j, an open source solution that is currently the most popular graph database in the world. It is sometimes understood that, since graph databases are unlike other NOSQL data models, which generally aim to simplify the relational model to maximize scaling potential, they are not

desired if performance is a primary concern. This is somewhat true, as web scalability of Neo4j is definitely inferior to that of MongoDB, a document-oriented database, but given appropriate conditions, Neo4j still provides great performance.

Consider a situation where we are to model a social network, consisting of 1,000 people, each of whom has 50 friends, and two people are connected if a path of or under length four connects them. According to a benchmark given in Jim Webbers talk, checking whether two people are connected in a relational database would take 2,000ms, whereas Neo4j could process this in 2ms. Furthermore, if we extend this model such that 1,000,000 people are represented, Neo4j would still require only 2ms. This is because graph databases dont generally suffer from problems joining large datasets, since each person is represented as a standalone node that does not get categorized together within a relation.

Indeed, queries that can be solved via traversing the graph and searching through nodes get processed very quickly, even when a large dataset is involved. However, this comes at the price of storing relationships between nodes explicitly. In the relational model, relationship sets are implicit and are not made obvious until a natural join of entities is made. In Neo4j, by representing every relationship set using connected nodes, joins need not be computed at runtime, yet significantly more data must be stored.

Neo4j is fairly efficient at storing data objects internally, using record IDs to reference the location of nodes, relationships and types, and linked lists to connect different relations a node has. Primarily, Neo4j stores three types of primitives, namely nodes, relationships, and properties. Each node object consists of two RIDs, which in turn point to a linked list of relationships and a linked list of properties. Without going into excessive detail, the relationship list describes the type of the relationship, points to the two nodes that are involved, and references one other relationship the nodes partakes. This reference allows for fast traversal, since it is common that the database manager is interested in all relationships a node is a part of. The property list, on the other hand, stores properties, or key-value pairs, of nodes and relationships, and again references other properties the node or relationship has, for the same reason as to speed up traversal time.

A difference Neo4j has in comparison to other NOSQL databases is that it supports fully ACID transactions, with a key emphasis on durability. Neo4j is committed to manage any data that is pushed towards them, and any transactions that are made get immediately written to nonvolatile memory. Data therefore doesnt fluctuate between various soft states, and the database manager is guaranteed that the system always returns a consistent and final value, instead of a temporary one. This way, even if the system suffers a failure and crashes, any data that has been written does not get lost, and all committed transactions survive permanently.

Nevertheless, most other NOSQL prefer emphasizing BaSE over ACID, and for good reason. This is mainly because horizontal scaling requires connecting databases across multiple servers, some of which may not be fully in sync with one another, such that atomicity and strong consistency must be foregone if web scaling potential is to be maximized. Naturally, by supporting ACID transactions, the web scaling proponent of Neo4j is not as strong as other NOSQL data models.

Typically, data models that support horizontal scaling have data distributed across various servers, and when large scale transactions are received, changes get appropriately handled by corresponding servers. Communications between servers are required to compute certain algorithms and create updates, but many transactions can also be handled independently, or concurrently. Unfortunately, graph data models usually consist of a large amount of deep interconnected nodes, and regular horizontal distribution of data is suboptimal. This is because when graph data is distributed across machines, it is ideal that cross-server edges or

relationships be minimized, and nodes across machines maximized. This way, cross server communication following relationship paths would not assume too much time overhead, while space is evenly used among the servers.

Yet, it is hard to distribute graph data across servers in a way that adheres to optimal standards. Most graph databases consist of one or two highly popular nodes that other nodes gravitate towards. As data gets added and removed from this server, distribution would tend towards a pattern where most of the data is stored in the machines having these highly popular black hole nodes, such that space is not evenly distributed. This results with an isolated, overloaded machine that has to deal with most transaction requests, defeating the purpose of horizontal scaling.

On the other hand, if the database manager forces the nodes to be evenly distributed among the servers, connected nodes would have to get separated, such that cross server relationships become very common. This way, even though space is evenly distributed among machines, transaction times may take significantly longer, as a number of cross-server graph traversals may be associated with each request.

Despite its shortcomings, Neo4j is currently the most popular open source database that follows the graph data model, and solutions to this models problems associated with web scaling are already being devised. To speed up information retrieval, Neo4j uses a technique called cache sharding, wherein data objects that are frequently accessed get cached in a blade server. Retrieving data from this server is significantly faster than retrieving data from disk; millions of traversals per second can be made if the caches are warm. A blade stores approximately 128GB in memory, and when large data sets up to terabytes in scale are involved, these blades get partitioned across multiple servers, wherein each blade contains part of the dataset. This way, accessing data across multiple servers is not nearly as slow, as ideally, most reads would not require an I/O seek.

Regardless, data distributed across multiple servers stills tend to suffer from the problems discussed above, where nodes are not optimally distributed among servers. Algorithmically, Graph defragmentation requires generating a minimal point cut, which takes polynomial time. In actuality, a lot of streamlining is still needed for graph defragmentation to get efficiently implemented. Hopefully, a cost-effective implementation would be made in the near future, so graphs data models would benefit immensely, and Neo4j would evolve as one of the only databases that supports fully ACID transactions and can scale horizontally.

This marks the end of our paper describing various features of Neo4j. Attached with the paper is a short five minute video tutorial, featuring us creating nodes, relationships, and indices on some basic data. Hopefully this encourages the reader to try out this DBMS, which comes with a very expressive GUI and supports multiple query languages, namely Cypher, and Gremlin.

References:

[1] Adell, Josh. "Neo4jphp." *GitHub*. N.p., 11 Dec. 2013. Web. 20 Nov. 2014.

[2] "Cypher Query Language." *The Neo4j Manual V2.1.5*. Neotechnology, n.d. Web. 20 Nov. 2014.

[3] Gioran, Chris. "Neo4j Internals: File Storage." *A Digital Stain*. Blogger, 9 Oct. 2010. Web. 20 Nov. 2014.

[4] Lindaaker, Tobias. *An Overview of Neo4j Internals*. Sideshare.net, 21 May 2012. Web. 21 Nov. 2014.

[5] Webber, Jim. "A Programmatic Introduction to Neo4j." *Vimeo*. GeeCON Conference, 15 Aug. 2011. Web. 21 Nov. 2014.