An Exploration of Graph Databases Through Neo4j

Visalakshi Gopalakrishnan
Dept. of Computer Engineering
Santa Clara University

Jonathan Jeng
Dept. of Computer Engineering
Santa Clara University

Nisarg Sohagiya
Dept. of Computer Engineering
Santa Clara University

David Wei
Dept. of Computer Engineering
Santa Clara University

Abstract

The concept of NoSQL (Not Only SQL) has grown in recent years to address the evolving and growing needs in this day and age, such as by Internet users. One type of NoSQL database that stands out in particular is that of graph databases, which have many strengths and weaknesses. One example of a graph database management system is Neo4j, which is notable for the fact that it has been around for a long time relative to many other graph databases and thus has a many supporters and users, including Cisco, Walmart, eBay, and HP. Neo4j has thus played a large role in the design of graph databases. As such, in this paper, we will regard databases in general, then study Neo4j in particular as a representative of graph databases.

Background

There exist five main categories of NoSQL databases: key-value stores such as Amazon's DynamoDB, column-based databases such as Google's BigTable, document stores such as MongoDB, graph databases such as Neo4j, and object oriented databases such as db4o.

The pure NoSQL databases typically scale relatively well. In particular, key-value stores scale horizontally especially well. Purely NoSQL databases are typically more flexible, as pure SQL databases have a fixed data model and a static schema. The requisite change implies slower development. Relational databases also suffer from degraded performance as the amount of data increases[1]. In addition, NoSQL databases, though, suffer from a higher probability of errors in searches, as NoSQL database management is decentralized, does not perform some of the checks on data as do relational databases, and often omits some features of relational databases, such as ACID[2] (Atomicity, Consistency, Isolation, Durability) properties. Rather, the NoSQL data model often only guarantees BASE (Basically Available, Soft state, Eventual consistency). Notably, however, graph databases like Neo4j are ACID compliant[3].

In addition, NoSQL has no standard query language, is relatively immature (relational databases have been in production for nearly fifty years[4]), and has fewer users than relational databases due to users being more familiar with SQL and for the above reasons. Furthermore, NoSQL databases do not support encryption of data files and are vulnerable to SQL injection and denial of service attacks[5].

For the reasons outlined previously, NoSQL databases are most typically used in scaled website (e.g., e-commerce) scenarios in which setting less import is placed on relational database features. In other words, the two kinds of databases, NoSQL and relational databases, are chosen for different purposes. Some have come to adopt both, such as Facebook, which uses a combination of NoSQL and MySQL[6]. The former allows for speed and complex queries, whereas the latter guarantees ACID properties and safety from security issues. Graph databases are clearly not ideal for a portion of applications, but are often used for uncovering patterns (e.g., fraud detection, recommendation engines), as query languages such as Cypher, as well shall discuss, function through pattern matching.

Subsequently, we will focus on graph databases in general, then elaborate on one particular type of graph database, Neo4j.

Graph Databases

As stated previously, one type of NoSQL database is the graph database. There are several different data models within the graph data model.

For instance, there is the hypergraph, in which hyper-edges may involve multiple features at one time, as opposed to the typical data model in which relationships are pairwise in nature[7]. For this reason hypergraphs are often, but not always, relatively superior for the purpose of capturing meta-intent.

There is also the property graph data model in which entities are nodes and relationships constitute directed edges that connect said nodes[8]. In this model, both relationships and nodes can be associated with multiple properties, such as a movie name and rating or the number of upvotes and downvotes a reaction to an event has received. Unlike single-relational graphs, property graphs are multi-relational and can contain different types of nodes[9], such as nodes that represent people as well as nodes that represent to events such as concerts or gatherings.

In addition, there is the RDF (Resource Description Framework) graph model. RDF is specified by the W3C (World Wide Web Consortium). RDF databases are notable for the fact that unlike other NoSQL databases, RDF has a query language specification: SPARQL[10] (SPARQL Protocol And RDF Query Language).

The RDF graph database is commonly associated with the Semantic Web movement, which did not gain traction, as is evidenced by the continued use of so-called Web 2.0. Critically, the semantic web depended heavily on Linked Open Data, which, is data available with an open license and linked to other data to provide context[11]. This movement has seemingly resulted in the sharing of large amounts of data, as is suggested by the diagram of datasets (figure 1) published in Linked Data format and maintained by Andrejs Abele and John McCrae. However, in the context of the massive amounts of data collected over the internet, there is still much to be desired in terms of data contributions. Further, as one might imagine, many companies may choose not to publish their data for reasons of protecting users privacy and because of financial motives.

Nevertheless, RDF is used to this day by prominent organizations, such as the BBC, which came up with and implemented the idea of Dynamic Semantic Publishing (DSP), which curates and publishes RDF and HTML aggregations[12]. DSP improves user experience, while automating the processes of aggregation and publishing, thereby promoting the publication of content on a dynamic basis (i.e., a shorter timescale) in events such as the FIFA World Cup 2010.

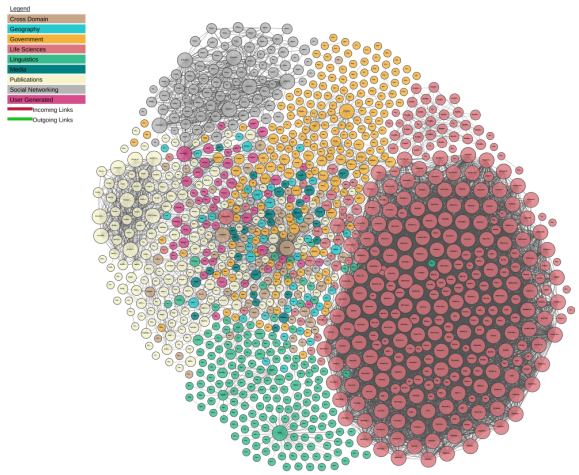

Figure 1: Diagram of Linked Open Data Datasets

In addition, RDF is excellent for the purposes of graph creation and pattern recognition. It is often used, but falls short in

other categories, as is evidenced by its not being adopted for other purposes.

However, we will not go into detail on the concept of RDF databases, as these are based on a data model distinct from the topic at hand. We shall next deliberate on the motivation of all types of graph databases in general.

From Relational to Graph

While relational databases still hold significant advantages over graph databases, relational databases still entail significant weaknesses, and graph databases address these weaknesses of relational databases.

Relational databases structure data by organizing the data into entity tables, and relation tables. At query time, two entity tables are joined together with a relation table in order to create the search space for the result. This is not concerning when joining together small tables, but joining a large number of possibly large tables quickly becomes too costly. Each new relation added to the query may add additional joins. Since each join is a cartesian product between the tables, the size of the solution space increases by a factor equal to the size of the table added to the query. As the size of the table increases it becomes exponentially more difficult to process, requiring multiple disk I/Os to write and read intermediary steps[13].

Furthermore, after creating this immense table, irrelevant rows are simply ignored when creating the result set. While some techniques, such as "push predicate down," can reduce the amount of wasted memory and processing, such techniques cannot completely reduce the wasted memory or processing. As a result, a simple query that only requires querying a small subset of the data now requires processing all records in an entity, related relationships, and other entities.

Finally, relational databases require a strictly defined schema. Within the schema, entities contain primary keys, and relationships contain a mapping between foreign keys of different entities. In order to execute a query, the involved constituents must contain an unbroken chain of primary and foreign keys. If, at any point, the structure of the data changes and/or a new relation between entities is added, a new relation table must be added with the foreign key to primary key connection populated. As a result, any changes to the structure of the data require a change to the rigid schema, and such a process may be quite expensive.

Notably, if one wishes to make use of a recursive query, such as to find friends of friends, doing so with a relational database requires recursive joining of tables. A relational database would suffer from significant query latencies, large memory requirements, and significant expense due to the numerous requisite joins and potentially, the sizes of the tables that must be joined. In the case of graph databases, query latency is merely dependent upon the area of the graph we wish to explore and not how much data is stored. Moreover, Neo4j, in particular, is optimized for graph traversals, which, in the relational database equivalent may be prohibitively expensive. Due to the superior runtime and since graph databases do not compute joins, graph databases are better suited for computing linked and/or recursive relationships.

Graph databases are ideal for and used for processing highly connected and variable relations data. Graph databases offers exceptional performance for reads by traversing through data, which are conceptually modeled as being in graphs,

All of these factors combine to result in relational databases being significantly slower

than their graph database counterparts[14]. Additionally, current trends, such as social networking through platforms such as Twitter and Facebook, often require one to operate on highly linked data and perform recursive queries. In the past decade, the amount of data has grown tremendously such as over the internet and relational databases do not handle well such massive sets of data. One can see the evolving needs by the fact that Oracle includes both a key-value database in Oracle NoSQL Database[15] as well as the Oracle Spatial and Graph option[16] in addition to Oracle Database 12c, which is characterized by Relational Data Structures[17]. This evidences that considering the associated strengths, such as security, maturity, and the strong user base and support, relational databases cannot be abandoned. For some needs such as described above, graph databases must also be considered, so that corporations will make use of both. Graph databases in general compliment relational databases, as graph databases store semi structured data without need for a schema and perform well on particular types of queries. For the purposes of this paper, we will focus on the graph database of Neo4j, as it is more mature than many other graph databases and is currently the most popular option.

Data Storage in Neo4j

As previously mentioned, there exist multiple types of graph data models, including the hypergraph data model and that of RDF. In this paper, however, we focus on the property graph data model used by Neo4j, which consider relations as on the level of entities while eliminating joins by requiring graph traversals instead.

There are two primary store files in Neo4j: node stores, which store nodes, relationship stores, which store relationships, and property store files, which keep user data in key-value pairs and allow properties to be attached to both nodes and relationships as is necessitated in the definition of property graphs. The two are of fixed size (15 bytes and 34 bytes, respectively), which allows for the direct computation of file locations and thus, fast traversals. Each node store file contains only two flags and several pointers to lists of relationships, labels, and properties (e.g., a singly linked list containing the properties of that node), Similarly, relationship store files also contain two flags and pointers to the relationship type, the start and end nodes of the relationship, the next and previous relationship records for the start and end nodes, but also, the IDs of the start and end nodes.

These pointers, for the most part, point to linked lists, as data in Neo4j consists of files of fixed size (as previously stated) consisting of linked lists. Node file store pointers to properties are singly linked lists whereas relationship file store are constituents of doubly linked lists, as each relationship contains pointers to the next and previous relationship in a relationship chain.

Given these pointers, the Neo4j database has index-free adjacency[18]. In other words, the Neo4j database includes pointers between nodes rather than a global index and therefore, neighbors of a node can be found without considering all relationships, relevant and irrelevant. As such, operations on the Neo4j database will conceptually scale well since the time to get relationships corresponding to a node is constant regardless of the size of the dataset, whereas the performance of a relational database decreases as the size of a dataset increases.

The organization of data in Neo4j as largely collections of pointers. As previously stated, This implies the property of index free adjacency and speeds graph traversal.

Given that files consist mostly of records, traversals are in essence, the process of

following pointers, and such can be done quickly given that files are of fixed size[19]. To do so, recall that relationship stores contain pointers to the start and end nodes of the relationship (in the terms of relational databases, these are analogous the entities involved in a relationship). To traverse the graph, travel to the first record of the relationship by computing its offset in the relationship store - such can be done because records are of fixed size - then from the relationship record, follow the pointer to the other node involved in the relationship. In this case, unlike in the case of hypergraphs, edges involve exactly two nodes so from here, all involved entities have been located. From this point on, one follows the linked list of relationships pointed to by the node traversed to find the next node, so to speak, of the path.

As with relational databases, graph databases benefit greatly from the use of SSDs but disk I/O's are still significantly expensive. To address this, Neo4j uses an LRU-K (Least Recently Used) page cache, a page replacement algorithm. This algorithm entails keeping a record of the last *K* references to popular pages and evicting the least frequently used page. This is further nuanced by the concept of page popularity, whereby unpopular pages are preferentially evicted, even if popular pages have not been accessed recently. Both conceptually and in practice, the LRU-K algorithm has little overhead, performs optimal statistical inference, and dynamically adapts to changing patterns of practice. Furthermore, the algorithm is superior to conventional buffering algorithms in the context of discriminating between frequently and infrequently referenced pages[20].

Locality of data

This property follows from allowing read operations locally on slave servers. Not only does each node contain replicated data, but also, these slave servers may be placed near the client, thus providing data locality for client applications[21].

For a comprehensive picture, though, one must consider that Neo4j uses graph data access methods that do not consider data locality and the majority of graph processing makes use of, specifically, random disk access[22]. This disk access becomes an issue when graphs cannot be stored in memory.

Thus, Neo4j enjoys locality of data in concept, allowing affiliated scholars to claim data locality from the allowance of satellite servers to be placed close to end-users[23]. However, the benefits of said data locality are of concern only when the relevant graphs can be stored in memory and even then, the benefits are limited by the fact that graph processing typically uses random disk access.

Clustering

In the context of Neo4j, clustering is accomplished in two solutions known as Causal Clustering and Highly Available Clusters[24].

Causal clusters are characterized by two types of servers: core servers and replica servers. Core servers safeguard data by replicating all transactions and ensure safety by requiring a majority of core servers to acknowledge a write. In the case of a core server cluster suffering enough failures that it cannot process writes, it assumes the property of read-only in order to ensure safety. On the other hand, read replicas act like caches for the data that core servers safeguard. Read replicas are consummate databases and are replicated from core servers. Read replicas periodically poll a core server for new transactions, which can be received by a large number of read replicas from a small number of core servers, thus implying scale.

Furthermore, Neo4j clusters are highly available. This property will be discussed in the next section.

High Availability

Availability refers to the accessibility of an application, service, or function[25] whereas in the context of Neo4j, the concept of "high availability" refers to the scalability package of Neo4j. It is characterized by fault data redundancy, service fault tolerance, and linear read scalability[26].We first address those characterizations, then discuss scalability of graph databases in general.

Firstly, Neo4j addresses the concept of redundancy, in spite of a single instance of Neo4j being constrained to a total of 34 billion nodes, 34 billion relationships, and 68 billion properties (such is sufficient for most users and is only questionable in the cases of companies such as Amazon, Facebook, Google, and IBM). Neo4j accomplishes this redundancy by replicating the full graph to each instance in a cluster. Given the above, it can trivially be concluded that a Neo4j cluster is fault tolerant.

Finally, linear read scalability is accomplished by allowing read operations locally on each slave. Changes in the number of instances in a cluster scale the number of read requests serviced each sentence by a constant amount proportional to the number of instances added or removed.

For graph databases in general, scaling encompasses three key themes:

1. Capacity (graph size)
2. Latency (response time)
3. Read and write throughput

1. Capacity

Some graph database vendors have chosen not to put any upper bounds in graph size in exchange for performance and storage cost. However, Neo4j maintains faster performance and lower storage by optimizing for graph sizes that lie at or below the 95th percentile of use cases. The current release of Neo4j can support single graphs having tens of billions of nodes, relationships, and properties. This allows for graphs with a dataset roughly the size of that corresponding to Facebook, according to some of the individuals responsible for Neo4j[3].

2. Latency

With regard to relational databases, it us usually the case that the more data is stored in tables, the more data is needed for each index. The longer the indices, the longer it takes to perform join operations because said indices must be read. Furthermore, join operations in relational databases are expensive because, as previously stated, said operations involve Cartesian products of relations, resulting in unnecessary data. In the case of more data being stored in relevant relations, joins are more expensive. Both factors give rise to latency issues.

In contrast, graph databases don't suffer the same latency problems. In a graph database, most queries follow a pattern whereby an index is used simply to find a starting node(s). The remainder of the traversal then uses a combination of pointer chasing and pattern matching to search the data store. As a result, performance does not depend on the total size of the dataset, but only on the amount of data in the area of the data store being examined. This leads to performance times that are nearly constant for even as the size of the dataset grows, latency does not increase much, if at all.

3. Throughput

Graph databases do not scale in the same way as other databases. When we look at I/O-intensive application behaviors, we see that

a complex operation typically reads and writes a set of related data, indicating that the application performs multiple operations on a logical subgraph within the overall dataset. With a graph-native store, executing each operation takes less computational effort than the equivalent relational operation. Graphs scale by doing less work for the same outcome.

The most demanding deployments will exceed a single machine's capacity to run queries, and more specifically its I/O throughput. When that happens, Neo4j uses clusters that scale horizontally for high availability and high read throughput.

Cypher

While graph databases can use SQL to query, a more natural way of querying is Cypher. Although Cypher is specific to the graph implementation Neo4j, because it stores graphs programmatically, it can be applied more generally to other graph databases. Instead of relying on the abstract concept of joins, the Cypher query language leverages pattern matching to codify relationships between nodes. Queries in Cypher can be divided into two broad categories, Read queries, and Write queries.

The structure of a Read query in Cypher mirrors its SQL counterpart. A basic Cypher query contains three clauses; Match, Where, and Return.

The Match clause defines the pattern of nodes and relationships to query using an ASCII depiction of the relationship. In order to find nodes that satisfy the condition Node_A In_Rel_With Node_B, the corresponding Cypher query is (Node_A)-[In_Rel_With]->(Node_B). In this case nodes are encapsulated within parenthesis and relationships are encapsulated within square brackets with an arrow pointing in the direction of the relationship.

The Where clause defines conditions and constraints that are applied to the pattern, similar to the WHERE clause in SQL. In Cypher, the Where clause that can exist independent of a Match clause. Unlike the Match clause, the Where clause uses syntax similar to its SQL counterpart. Figure 2 shows a comparison between SQL and Cypher Where clause syntax.

SQL Query

```
SELECT n.title
FROM movie_names n
WHERE n.rating > 7 AND n.ID > 5
```

Cypher Query

```
Match (n)
Where n.rating > 7 AND n.ID > 5
Return n.title
```

Figure 2: SQL vs Cypher Match Syntax

The Return clause defines the data that should be returned. It is analogous to the Select clause in SQL and its similarity is illustrated in Figure 2. Also, unlike the Match clause, the Return clause uses syntax similar to its SQL counterpart

Write queries allow a developer to insert or delete nodes from the graph. Cypher uses queries with Create / Delete, Set / Remove, and Merge clauses to perform these actions.

The Create / Delete clauses are used to create or delete nodes or relationships from a graph.

The Set / Remove clauses are used to set or remove values from properties inside of a node, as well as apply or remove labels from nodes.

The Merge clause allows the developer to specify that a relationship pattern must exist after the creating a node. If the pattern already exists, this clause does nothing. If the pattern does not exist, this clause will create additional

nodes and relationships until the pattern is satisfied[27].

As this shows, from the Cypher query language, we gain a view of sorts of the data model used by the Neo4j graph database. Next, we shall consider how the aforementioned queries are evaluated.

Query Evaluation

To evaluate a query, Cypher must traverse the corresponding graph to collect records of interest. Cypher abstracts the query process into five high level categories of functions, known as operators. The five categories of operators, Starting Point, Expand, Row, Update, and Combine, all handle a specific part of the query execution. Within each category resides several implementations of algorithms or methods to accomplish the task. Each operator in a category can be interchanged with any operator of the same category, and the selection of the operator depends on the nature of the graph and query. The Cypher query engine chooses a series of operators used to evaluate a result called the execution plan. There is no guarantee that the Cypher query engine will choose the same execution plan for the next execution of that same query.

To begin a traversal, Cypher must first determine the node at which the traversal should start. To accomplish this, Cypher uses a category of operators known as Starting Point operators. Starting Point operators identify where the traversals should begin. At the most basic level, Cypher performs a scan of all nodes looking for nodes that match your pattern. More advanced and faster algorithms employ properties of the graph, such as labels and relationships, to reduce the search space.

After identifying the location where the traversal should start, Cypher must begin a search through the nodes based on the pattern. The most basic expand operator is the Expand All operator. This operator follows all node relationships that match the pattern specified. If the start and end points of the graph have already been identified, then Cypher can employ the Expand Into operator, which identifies all connecting relationships linking the start and end nodes.

Additionally, similar to SQL, Cypher contains the ability to push predicates down before expanding into other nodes, thereby improving performance. This is accomplished with the optional Expand All operator.

Row operators allow Cypher to perform SQL-like aggregation operations on rows produced by other operations. Common SQL operations that exist in Cypher include projection, Union, Unique ("Distinct" in Cypher), Sort, Filters, Top, and Count. One particular aggregation operation that exists in graph databases that does not exist in SQL is Eager. Eager allows Cypher to segregate the effects of an operator by forcing an operation to be applied fully to the graph before continuing with another operation. This can result in a significant increase in the number of disk I/Os, but prevents operations from matching data that was just created, which would potentially result in an infinite loop.

Update operators are applied whenever Cypher performs a Write query. These operators allow the developer to update graphs, specify constraints, as well as identify whether or not a relationship exists, and create relationships when a relationship does not already exist.

Finally combine operators in Cypher allows Cypher to combine different operators together to form a complete query. In addition to combining operators, combine operations can also specify whether or not to apply predicates before an operation, check whether a variable returned is null, assert uniqueness, apply hash joins, and solve triangular relationships[28].

Cypher Vs Sql

While graph databases can actually support SQL, Neo4j has opted to create their own query language, Cypher. The decision to do this primarily stems for the complexity of SQL queries required when finding relationships between entities. As mentioned before, unlike relational databases, graph databases consider relations as first class citizen status. Querying in this paradigm, SQL requires multiple join statements to establish a relationship. The complicated joining of private and foreign keys can quickly overwhelm a developer distracting away from the premise of the original query. Cypher, on the other hand, was built using this paradigm. By focusing on the relationships between elements, and defining the query in terms of a pattern of entities and relationships, the query itself becomes more concise and natural[29].

In Neo4j's white paper, Neo4j uses a query to find people who work in an IT department. Figure 3 shows the comparison between the SQL query required to perform this task, versus the equivalent Cypher query[30].

SQL Query

```
SELECT name FROM Person

LEFT JOIN Person_Department
   ON Person.Id = Person_Department.PersonId

LEFT JOIN Department
   ON Department.Id = Person_Department.DepartmentId

WHERE Department.name = "IT Department";
```

Cypher Query

```
MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)
WHERE d.name = "IT Department"
RETURN p.name
```

Figure 3: SQL vs Cypher Query

Performance

Query results in relational databases often are more quickly obtained with indices, horizontal data partitioning, and by use of embedded query optimization tools. However as the amount data increases, performance is impacted by the complexity of table joins and the increase in consumption of computing resource. Delete and Update operations are much slower than are reads.

In contrast the performance of a Neo4j graph database stays relatively constant even if the data is continuously increasing and changing. To some extent that is acceptable, as the data is fundamentally stored at record level. Records are of fixed size and storage is optimized for easy traversal of a graph with the starting point of the traversal as one of the roots, rather than having to perform a global search. This allows for a flexible data storage model and thereby makes Update and Deletes operations faster than in a relational database.

Traversal

As previously mentioned, data in Neo4j is stored as, essentially, collections of pointers. This implies index-free adjacency, meaning that neighbors of a node can be found without having to consult all corresponding relationships in the graph.

Index-free adjacency has significant performance advantages, namely, improved traversal performance, as one need only follow the pointers described, instead of continually performing index lookups, in order to traverse a graph. Queries such as graph traversal thus, in practice, run much faster using the Neo4j system than when using the SQL engine[31]. However, this benefit comes at the cost of other queries being expensive. Queries that do not use traversals are memory intensive, as one cannot

simply follow pointers, and expensive. Additionally, the amount of memory is finite so cache swapping will eventually cause performance to decrease. However, Neo4j addresses this issue by cache-based sharding, wherein requests for users are always sent to a specific, unchanging server. As such, there is a consistent routing of requests so caches of all servers can potentially be used to the utmost extent.

The difference in performance becomes more important as the volume of data increases. In relational databases, the tables will have to be fully scanned and each index scan referencing a key will cost $O(\log 2N)$[32] if using a B-Tree, with N being the number of records in the tabled scanned.

In a graph database such as Neo4j, the first latency will be to find the starting node(s) for traversal and then the rest of the scan for that query is more or less following the links and that implies linear complexity; i.e., depth-first search. Finding the starting node could be faster by using indices on the node property.

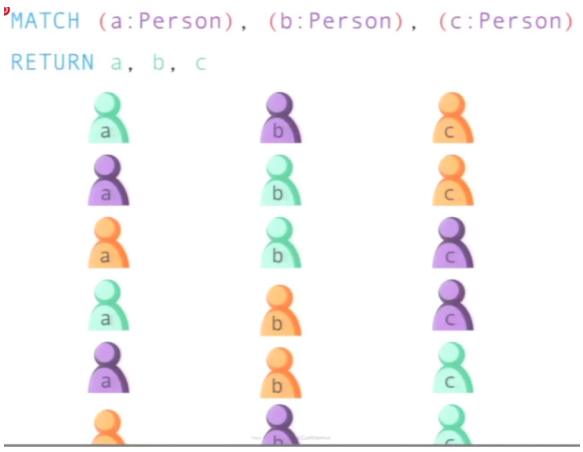

Figure 4: Result Set for a Simple Cypher Query

If the starting node is not indexed properly, faster starting point operators involving indices cannot be used. In the worst case, the Cypher engine might require an "all nodes scan," making a simple query, as seen above, produce a large result set. The large result set could have redundant information or loops requiring the use of an Eager operator, thus greatly diminishing performance and requiring additional disk I/O's. Labeling a node allows that node to be identified as a particular type and using that label in the query will decrease starting latency as it searches for that particular starting point. As a result, Neo4j may be considered fundamentally superior in the context of particular queries, but such performance may be compromised.

Neo4j allows different methods of indexing to improve performance. From earlier years of no indexing to automatic indexing and labeling, Neo4j now allows schema indexing. Schema indexing is similar to indexing in relational databases, where a schema index is based on the label of the certain property of the node. The following exemplifies schema indexing using Cypher.

```
CREATE INDEX ON :Person(name);
```

This statement will create an index for all nodes that carry the 'name' property.

```
MATCH (p:Person {name: 'Stefan'}) RETURN p;
```

Consider the above Cypher query. If no index is created other than label 'person' given to the node, the Cypher engine will scan all nodes with label 'person' and then try to match the 'name' property. However with schema indexing, the engine will now scan only for nodes with label 'person' that also have the 'name' property. This results in better performance than merely labeling nodes. One may consider node labeling as indexing the supergraph subset within the database and schema indexing on the property of the node as the subgraph within the supergraph. However, indexing should be used judiciously as

index files can become significantly large in size and CPU intensive.

Neo4j[29] also has a limitation on the number of relationship types to 65K in order to maintain speed. Properties are organized as Binary Large Objects (BLOB's). These properties are distributed in a file and size limitation makes reading of these files easy.

Comparison of performance between graph databases and relational databases is challenging because of the fundamental difference between how data is stored and accessed. In terms of query performance, graph databases would supersede relational databases if its access path were based on traversal and the relevant data is indexed properly. However the depth of the traversal or poorly modeled data could hurt performance. In addition, graph databases are fundamentally better suited to queries that rely on relationship between entities and any change in the relationship does not affect the way queries are formed, whereas relational databases are better suited for queries that revolve around structured data and has little if not any tolerance for change in the data structure.

This reasoning is borne out in the literature[33]. Such is not surprising and in line with the previous deliberation, so we will not dwell on the point. Rather, in the following, we will elaborate on other factors that contribute to performance.

Architecture and Performance

Neo4j uses a master-slave cluster architecture, wherein all instances in the cluster contains a cluster management component to track instances joining or leaving and write operations are coordinated by the master, which puts an upper limit on the write capacity of the cluster. Thus, the write throughput of a single machine limits Neo4j's dealing with write loads.

This constraining, however, does not preclude high write throughput. Few scenarios are characterized by continuous, high write load; rather, the loads have periods of high and low write load. Therefore, a queue for writes can be used and such ensures that a steady, manageable stream of write operations. This increases performance because it allows write operations to be batched and allows the system to pause write operations even when short periods of maintenance are requested. In the context of extraordinarily high write loads, performance can be improved by vertical scaling, such as by the use of SSDs, which increase I/O performance.

Thus, the architecture of Neo4j is optimized for the expected use cases and thus, fundamentally improves the potential performance of this graph database.

Query Optimizer

Notably, the optimizer used by Neo4j not very mature relative to those for PostgreSQL and for Clingo, which in turn, are not mature relative to those for relational databases. The optimizers of relational databases have gained competence in the dozens of years of development, whereas those of graph databases, particularly that used by Neo4j, do not perform well. This poor relative performance may be attributed to the fact that as of 2014 and in all prior years, the Neo4j query engine likely used rule-based, rather than cost-based query optimization[34].

However matters changed as Neo4j 2.2, released in 2015, included a new query optimizer for Cypher, which was named Ronja. In contrast to the prior rule-based optimizer, the cost-based optimizer of Cypher purportedly uses statistics regarding the graph itself in order to choose an optimal query execution plan[35]. Relative to the rule-based optimizer, the

cost-based optimizer decreases runtime for compilation by up to two orders of magnitude[36].

Unfortunately, the optimizer is, as said, young. Though the cost-based optimizer has been demonstrated to be superior to the previous rule-based optimizer, query optimization techniques remain to be discovered. Furthermore, performance tests remain to be conducted, both using a wide variety of datasets with the existing query optimization techniques, which are clearly inferior to those used by relational databases, and using the query optimization techniques that remain to be contrived and also in conjunction with a wide variety of datasets.

Concepts regarding graph databases in general suggest that the new query optimizer serves to improve queries entailing graph traversal, which already perform at a superior level relative to relational databases. This improvement should not be trivialized, however, there exist a number of applications that do indeed primarily use graph traversals. However, one might question whether an improved optimizer can overcome the hurdles presented by the conceptual foundation of graph databases, as such is ill-suited for queries that do not involve graph traversal. As stated previously, query optimization techniques remain to be contrived and relevant performance tests remain to be conducted. At this point, the existence of the new query optimizer is well established, but the relative performance of it has not been evaluated and/or published.

ACID

ACID is a critical set of properties of databases in general and is well embodied by relational databases wherein these properties define the database transactions and guarantee a safe environment to operate on the data.

Broadly speaking, ACID refers to:

Atomic: All or none. This property ensures that all operation in a transaction must succeed. Even if one of the operations fails, entire transaction is rolled back and the state of the database is unchanged.

Consistent: This property ensures that on the completion of a transaction, the database safely transitions from one stable state to another confined by all constraints defined by the data structure

Isolated: This property ensures that transactions can run concurrently and would result in a stable database state as though the transactions are run sequentially

Durable: This property ensures that once the commit happens, the data written is permanent and is not affected by any form of failure.

For many domains and use cases ACID guarantees data safety, but in some domains in the NoSQL world, ACID properties are not employed and the stringent rules of ACID have been loosened in order to gain benefits such as higher performance, quicker obtaining of result sets and scaling. Specifically, BASE better enables horizontal scaling[37]. The previous result from BASE being a more optimistic approach, assuming that data may not, at one point, be consistent as there are a number of transactions to input to the database in volume, but consistency will eventually be had. ACID, in contrast, assumes the worst and forces consistency after each transaction.

Prior to BASE, the CAP (Consistency, Availability, and Partition Tolerance) theorem, as presented by Eric Brewer in 2000 at ACM symposium, was used in the NoSQL world. The CAP theorem states that at any given time, out of three possible combinations of CAP (i.e.,

Consistency, Availability, and Partition Tolerance), only two are available[38]. In other words, at any given time, a database in an distributed environment could have partition tolerance and consistency or partition tolerance and availability but not all three.(Partitioned tolerance means that even if few nodes fail due to network failure, the remaining nodes can perform to provide service)

BASE has evolved from the CAP theorem and Neo4j embodies the properties BASE. The following are the characteristics referred to by the properties B, A, S, and E.

Basically Available: The database works most of the time. However, there are no assurances that the data retrieved will be in a consistent state or that it will be always available. Sometimes database could fail to retrieve data.

Soft state: Data can change over time as the database only eventually seeks consistency.

Eventual consistency: As the property name suggests, the database will eventually arrive at a consistent stage. In the interim, no transactions are halted because of the inconsistent state of database.

NoSQL databases use the BASE model because they do not usually comply with ACID model, something used by relational databases. However, some NoSQL databases adopt additional approaches and techniques to make the database comply with ACID model.

NoSQL databases essentially universally use the BASE model. Mohan (2013)[39] argues that although NoSQL databases do not guarantee the concept of ACID transactions, they support some form of a smaller transaction to promote data consistency within the database. As such, although NoSQL databases typically do not use the ACID properties, some use techniques so that the database complies with those properties. However, Neo4j is exceptional for its ACID compliance.

Neo4j, in particular, is capable of all of the properties of ACID. Relative to NoSQL databases in general, Neo4j is perhaps the most ACID compliant. As in relational databases, Neo4j enjoys atomicity for all operations must be performed as a transaction and if all do not succeed all fail and the database is not changed. With regard to isolation, Neo4j only makes accessible data that has been committed. As for durability, Neo4j ensures that the database can recover committed transactions by use of an update log. The one area where matters are left to be desired is consistency. Potentially, locks can be manually applied, thereby achieving consistency. However, such impedes performance. As such, Neo4j always guarantees the properties A, I, and D in ACID, but only mostly, rather than always, guarantees consistency[40]. With regard to consistency, Neo4j argues that its enterprise product offers configurable consistency and balances performance and durability[41].”

Notably, other NoSQL databases typically only enjoy atomicity at the lowest level of data structure (e.g., at row-level, single documents, or single items), essentially are only eventually consistent, provide varying levels of isolation, and are only usually durable. Graph databases in general follow this trend, only offering ACID compliance to some degree, but Neo4j is quite ACID compliant for a NoSQL database.

Use cases

Fraud Detection

Traditional methods of fraud detection are based on discrete analyses and thus are susceptible to false positives and false negatives. Knowing this, fraudsters work to develop a

variety of ways to exploit the weaknesses of said fraud detection, creating much difficulty for those being targeted. Graph databases, however, are able to close, to some extent, the opportunity for fraudsters inspired by traditional methods of fraud detection.

Simply by the use of graph databases, one may discover patterns that are buried in tables and difficult to observe. As in the case of data analysis, alternatively representing the relevant data, analogous to visual representation of data, makes hidden patterns obvious. Entries in cells of a table may not allow one to observe abnormal relationships as do relationships between nodes, as in Figure 5[41].

As with relational databases, triggers can be implemented. In the case of graph databases, triggers can cause the corresponding transaction to be evaluated against a fraud graph. Complex graphs may give cause for concern.

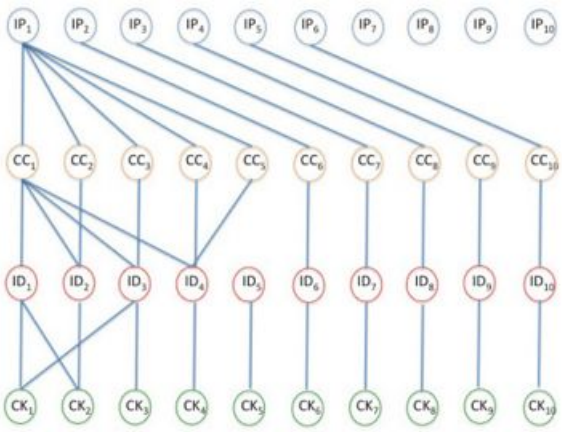

Figure 5: Graph of Transactions to Detect E-Commerce Fraud

Real-Time Recommendation Engines

Key for creating an effective real-time recommendation is the database that understands the relationships between data. Graph databases can efficiently store now entities (i.e., nodes) are related. With a graph database, one may efficiently discover involved entities, tracking relationships according to user, by traversing the graph. In this way, graph databases can efficiently give meaningful insights into customer needs and product trends.

Conclusion

Graph databases are a new and powerful option. As argued, graph databases surpass traditional, relational databases in the case of particular tasks (e.g., when data can be stored in a graph format and queries can entail graph traversals, as well as in the cases fraud detection and recommendation systems). There are different types of graph databases but Neo4j in particular stands out for exceeding other NoSQL databases, such as Cassandra, in its ACID compliance. Neo4j is not without its faults, however, and these ensure that the concept of graph databases, in its current form, will not supplant relational databases. The two compliment each other rather well. Whereas graph databases perform poorly on queries that do not entail graph traversals, graph databases also scale well relative to relational databases and are superior in the case of pattern detection. The latter follows from the fundamental structure of the query language Cypher as matching patterns and the alternative representation of data as graphs, which promotes pattern recognition. Neo4j is quite robust, but its performance shortcomings from query structure and from lack of a mature optimizer hold it back. Furthermore, relational databases are widely supported, well documented, and thus serve well for particular purposes.

It remains to be seen whether the optimizer of Neo4j can be improved to the point that it can be considered as proficient as the optimizers of relational databases, or whether a different type of database will be devised and will supplant both Neo4j and relational databases. With regard to the former possibility, more research must be done on optimization

techniques in the environment of many more datasets. Further, optimization techniques must be devised and investigated. In addition, the inability of Neo4j to handle certain types of queries that do not entail graph traversals is quite compromising. Neo4j may be quite powerful in some realms, but without a solution, it is certain that Neo4j cannot be one's sole database solution.

References

1. Ameya Nayak, Anil Poriya and Dikshay Poojary. Type of NOSQL Databases and its Comparison with Relational Databases. International Journal of Applied Information Systems (IJAIS) Foundation of Computer Science FCS, New York, USA Volume 5– No.4, March 2013.
2. A. B. Vavrenyuk, N. P. Vasilyev, V. V. Makarov, K. A. Matyukhin, M. M. Rovnyagin, A. A. Skitev, "Modified Bloom filter for high performance hybrid NoSQL systems", Life Science Journal, pp. 457-461, 2014.
3. Robinson, Ian, Jim Webber, and Emil Eifrem. Graph Databases. Beijing: O'Reilly, 2015.
4. "Relational Database." Wikipedia. Wikimedia Foundation, 31 May 2017.
5. C. Nance and T. Losser, "NOSQL VS RDBMS - WHY THERE IS ROOM FOR BOTH," in Proceedings of the Southern Association for Information Systems Conference., Savannah, GA, USA, 2013.
6. Zass R, Shashua A. Probabilistic graph and hypergraph matching. International Conference on Computer Vision and Pattern Recognition. 2008:1–8.
7. "What Is a Graph Database? A Property Graph Model Intro." Neo4j Graph Database.
8. Marko A. Rodriguez and Joshua Shinavier. Exposing multi-relational networks to single-relational network analysis algorithms. Journal of Informetrics, 4(1):29–41, 2009.
9. "RDF 1.1 Concepts and Abstract Syntax." W3C, 25 Feb. 2014.
10. Berners-Lee, Tim. "Linked Data." Linked Data - Design Issues. W3C, 27 July 2007.
11. Rayfield, Jem. "Dynamic Semantic Publishing." Springer. Springer Berlin Heidelberg, 01 Jan. 1970.
12. "Ebook: Guide to Graph Databases for the RDBMS Developer." Neo4j Graph Database.
13. C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In Proceedings of the 48th Annual Southeast Regional Conference, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.
14. "Oracle NoSQL Database." Oracle NoSQL Database | Database | Oracle.
15. "Oracle Spatial and Graph." Spatial and Graph | Database | Oracle.
16. "Contents." Database Concepts. N.p., 04 Apr. 2017.
17. K. Barmpis, D. Kolovos. Comparative analysis of data persistence technologies for large-scale models. Extreme Modeling Workshop, ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems (2012).
18. "Relational Databases vs. Graph Databases: A Comparison." Neo4j Graph Database.
19. E. J. O'Neil, P. E. O'Neil, G. Weikum, "The LRU-k Page Replacement Algorithm for Database Disk

Buffering", Proc. ACM SIGMOD Conf., 1993, pp. 297-306.
20. Gupta, Sumit. Neo4j Essentials. N.p.: Packt Limited, 2015.
21. Sakr, Sherif. "Processing Large-scale Graph Data: A Guide to Current Technology." IBM - United States. IBM, 10 June 2013.
22. D. Montag, 2013. "Understanding Neo4j Scalability," White Paper, Neotechnology 2013, retrieved from http://info.neotechnology.com/rs/neotechnology/images/U nderstanding Neo4j Scalability (2) .pdf
23. "Chapter 4. Clustering." Chapter 4. Clustering - The Neo4j Operations Manual V3.2.
24. "Database High Availability Overview." Overview of High Availability. Oracle.
25. "Chapter 3. Cypher." Chapter 3. Cypher - The Neo4j Developer Manual V3.2. Neo4j.
26. "Execution Plans." 3.7. Execution Plans - Chapter 3. Cypher. Neo4j.
27. F. Holzschuher and R. Peinl. Performance of graph query languages - comparison of cypher, gremlin and native access in neo4j. In EDBT/ICDT Workshops, pages 195–204, 2013.
28. Hunger, Boyd, Lyon. The Definitive Guide to Graph Databases for the RDBMS Developer.
29. Welcome to Dark Side: Neo4j worst practices (& how to avoid them). By Stefan Armbruster Field Engineer | February 24, 2016.
30. Xia Y. et al.: Graph analytics and storage. In: IEEE Big Data (2014).
31. A Comparison of a Graph Database and a Relational Database
32. Chad Vicknair, Xiaofei Nan, Michael Macias, Yixin Chen, Zhendong Zhao, Dawn Wilkins. Department of Computer and Information Science University of Mississippi.
33. Andrey Gubichev and Manuel Then. Graph pattern matching: Do we have to reinvent the wheel? In Proceedings of Workshop on Graph Data Management Experiences and Systems, GRADES'14, pages 8:1– 8:7, New York, NY, USA, 2014. ACM.
34. Gillin, Paul. "Neo Claims Big Performance Gains in Update of Top-ranked Graph Database." Silicon Angle.
35. Andrey Gubichev. Query Processing and Optimization in Graph Databases. PhD thesis, M¨unchen, Technische Universit¨at M¨unchen, Diss., 2015.
36. Rabi Prasad Padhy and Deepti Panigrahy , "NoSQL Databases: State-of-the-art and security challenges" , international journal of Recent Engineering Science (IJRES), Volume 16 , October 2015. http://ijresonline.com/archives/volume-16/ IJRES-V16P106.pdf
37. Mason, R. T. (2015), 'NoSQL databases and data modeling techniques for a document-oriented NoSQL database', Proceedings of Informing Science & IT Education Conference (InSITE) 2015, 259-268.
38. Mohan, C. (2013). History repeats itself: Sensible and NonsenSQL aspects of the NoSQL hoopla. EDBT/ICDT 2013 Joint Conference, March 18–22, 2013, Genoa, Italy. ISBN: 978-1- 4503-1597-5.
39. Deepak, G. C. "A Critical Comparison of NOSQL Databases in the Context of Acid and Base." Graduate Faculty of St. Cloud State University (2016).
40. Sebastian Verheughe, Architect and Developer, Telenor, - Volker Pacher, Senior Developer, EBay, and - Marcos

Wada, Software Developer, Walmart. "Unlock the Value of Data Relationships." Neo4j Graph Database. Neo4j.

41. Webber, Jim, and Ian Robinson. "White Paper: The Top 5 Use Cases of Graph Databases." *Neo4j Graph Database*. Neo4j, 2015.