Database Management Systems in Social Networking

Andrew Sabot
Computer Science
University of Toronto
andrew.sabot@mail.utoronto.ca

Graeme Stroud
Computer Science
University of Toronto
graeme.stroud@mail.utoronto.ca

Juliana Dean
Computer Science
University of Toronto
juliana.dean@mail.utoronto.

***Abstract*—This paper discuss the challenges of modeling social media data in a database. It surveys the ideas discussed in three papers. It gives an introduction to the structure of social media data, the types of requests made of the dataset, and possible issues and solutions to storing the social data. This paper covers how those solutions perform on certain tests, and also goes into depth on a specific industry case study.**

Keywords—database, social media, useKit, NoSQL, Sharding, JSON/BSON, schema evolution

I. Introduction

Social media has been growing in popularity over the past two decades, and as a result, large quantities of data have been generated. The solutions for storing and accessing such data have become an important topic of study. The type of data generated by social networks spans many different data types and sources. This means that any database solution must be designed to efficiently change with the platform as more functionality is added. This includes the need for quick solutions in schema changes and the ability to integrate with other data sets. Along with the dynamic data environment, there is another important constraint: social media users require real time responses. Therefore, a social network's database system must have fast read and writes, and there must be as little down time as possible in the event of a failure. One of the most crucial issues that all social media database systems will face is the scale of the data. Most databases will not fit on a single machine, and often time the database is distributed across multiple physical sites. This paper does not discuss in depth how each database is distributed, but it does discuss the pros and cons of certain databases' distribution methods in Section IV.

There are many database solutions available for storing and interacting with social network data, each of which excelling at certain database tasks. One way to compare the databases is to measures their performance on the useKit dataset. UseKit is a graph dataset that models a social network dataset containing 50,000 nodes and 100,000 edges[1]. Similarly, there is a Facebook dataset with friendship relations which is commonly used to measure performance. The challenges of converting this dataset into each database will be discussed later in the paper.

The database solutions that will be discussed in this paper are: MySQL, REDis, Riak, MongoDB, CouchDB, Hbase, Cassandra and Neo4j. These databases are tested for performance with UseKit's data set, and are evaluated on their ability to meet the demands of social networks in terms of scalability, ease of integration, and response time.

Finally we will compare how some of the databases performed on the benchmarks and discuss what is currently used by a few social media companies. Some solutions make a lot of sense theoretically, but in practice don't perform well due to the scale, such as Neo4j. We also include a case study of LinkedIn's DBMS, Espresso, as a solution to the limitations of RDBMS in regards to their expanding data ecosystem. Such features include schema evolution, secondary indexing, and an augmented sharding method as means to improve performance and lower the cost of adding new data.

II. Social as Represented by a database

Before we can get into how the databases work, we must first introduce the abstract structure of social media data. In a social network there typically is a user with personal and public information. This includes friends ("the network"), posts, pictures and more. As social media has changed, other things have been added like mobile location, videos watched and much more. A typical way of visualizing a social network is that each user is a node containing information about themselves, and the nodes are connected in a big graph, where each edge represents a relationship between users. This graph grows exponentially with the addition of more users.

The information stored in the node varies significantly for each social network, as well as the technology that the users use. The constant evolution of user data and user technology must be accommodated by the social network technology to keep the platform relevant to users. The example that we keep returning to is that of the user location, since before platforms like Facebook were on mobile, it was not crucial for the database to handle large amounts of location data associated with each user. This demonstrates the issues surrounding the need for schema evolution. Social networks are constantly expanding the data that they store within or related to certain entities, which requires schema changes to accommodate for these. The problem that comes with this is that schema updates can be very costly in their need for down time, time spent in modifying the system, and time spent properly integrating changes with other clients.

GRO-U-P25CS-C443-S/IN/$A!.00 ©2018 IEEE

It's also important to note the scale of data generated by the social networks. For example Twitter generates 155 million tweets per day [1]. This rate has increased since paper 1 was published, but it gives a sense as to the scale of the information coming in. Each tweet contains other information like location, users interactions and more. That number is just the start of the daily data that the database will have to deal with.

III. Current Databases

Social networks use different databases depending on the social network's requirements. Table 1 shows which social media platforms use which databases [2]. According to this table, it is evident that social networks use a mixture of databases, and typically use more than one database type. This is because there are some disjoint collections of data in which one database performs better than another. Social networks may have data pertaining to not just users and other entities in the network, but also crucial data regarding the platform's other features as well as metadata. Since the use cases are different and may require different access patterns, this leads to different database types being used. An example of this may be a social network having both the user connections feature, and also in-platform games, which may result in one's data being more demanding of real-time updates or fast failure recovery than the other. As discussed later in this paper, there are advantages to every databases solution, as well as tradeoffs regarding scalability and logical structure.

TABLE I. NoSQL Databases used in Social Networks

Social Networking sites	NoSQL Database Subcategories used
Facebook	Cassandra, HBase, Neo4j
Twitter	FlockDB, Cassandra, HBase, Neo4j
LinkedIn	Voldemort, MongoDB, HBase, AllegroGraph
Flickr	MongoDB, Neo4j
Friendfeed	HBase, Cassandra, OrientDB
Foursquare	MongoDB, CouchDB, Riak, Cassandra, InfoGrid
MySpace	MongoDB, DynamoDB, Neo4j

There are various advantages and disadvantages between using different databases. Table 2 shows some of the features of different databases [1]. From the table we are able see that all solutions support having keys, but range search is not supported by graph search. Using SQL to query the database is only supported by MySQL and MongoDB. A feature which MongoDB prides itself in having due to its usefulness at generating complex queries.

	Key	Range	SQL	MapReduce
MySQL	+	+	+	-
Redis	+	+	-	-
Riak	+	-	-	+
MongoDB	+	+	+	+
CouchDB	+	-	-	+
HBase	+	+	-	+
Cassandra	+	+	-	+
Neo4j	+	-	-	-

Table 2: Query types supported by storage systems

Another difficulty in comparing databases lies in the different possible data models in representing the same entities and relationships. For example, storing a person with a location and friends can be represented very differently depending on the database. In a graph database each person could be a node with the friendships being represented by edges. On the other hand, in SQL each person might be a row in a person table, and each friendship is a row in a friendship table. Table 3 groups the databases used today by data model, and provides a brief description of the common application of such database depending on its data model. These differences arise do to the difference in each database's level of structure. Structure also impacts a database's ease of scalability. As seen in Figure 1 [2], more structured databases, such as ones using the relational model, are harder to scale. On the other hand, Less structured databases, like key-value ones, are easier to scale due to the simplicity of storing data and representing relationships.

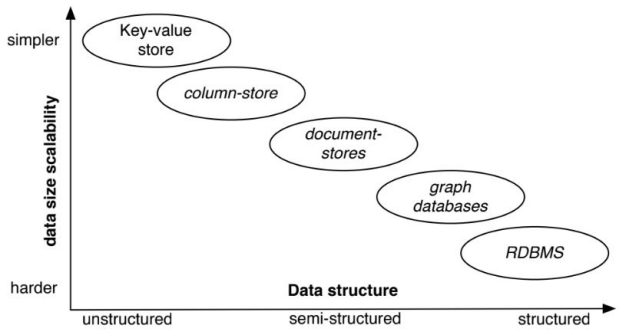

Figure 1: Data size scalability vs data structure

Below is an in-depth evaluation on the types of databases and their advantages in representing social network data.

A. Relational

One of the earliest models is the relational model. Data is stored as a tuple, and multiple tuples are stored in a table representing a relation. An example of a common relational DBMS is MySQL. Difficulties with RDBMS for social media include the strict schema that may not be a good model for social media data. It is considered semi-structured, in that new appearing data types result in a need for structural change. In

the context of ever-expanding social media data, relational DBMSs require a lot of work for schema changes. The resulting integration of these changes into the system, as well as client compatibility adjustments, take a lot of planning and must be manually done. [4]

Accessing a significant quantity of the data collected in social media is also an issue for the relational model. JOIN queries over large data sets are a big constraint of using RDBMS for social data. This is due to the rigidity of keeping elements strictly grouped into tables. For example, finding a user's friends requires joining the user table with a friends table to determine which users share this relationship. The join on such a large table would take significant processing. As the behaviour of social networks require a lot of reads, and read queries require a lot of joins, such overhead with relational databases makes it hard to justify its use as a social network database.

Despite such potential performance constraints, many social networks use relational databases. This can be attributed to the widespread support for relational databases and the longevity of MySQL. What comes with these benefits is an easy setup and a significant amount of documentation. Some companies have augmented existing SQL databases as a work around to the relational model's limitations. Facebook, for instance, has a significant amount of caching on commonly accessed data. [1] This cuts the cost of potential disk requests on expensive queries.

NoSQL databases avoid using the relational model. Triggered by the demand for big data and web applications, NoSQL databases store data using a less strict data structures in order to make insertion operations more efficient. Column store, document store, key-value store and graph store are all forms of NoSQL databases, which are described and evaluated below.

B. Column

Column-oriented databases are semi-structured, yet differ from relational ones in table storage method. Instead of storing all attributes of a record together in a table like RDBMSs, column-oriented databases store each table's columns separately. The purpose of this is to keep attributes for all records together. Each row representing a full record has a row key, which is stored in each column, so a key request is used to access data. An advantage to this method is that one only has to read the columns that are needed for a query. This is especially useful for aggregate queries.

Hbase is an example of a column DBMS, and is based off of the Hadoop Distributed File System. An advantage to this DBMS is that indices created for all row keys automatically. MapReduce queries are supported, as it also is for other column DBMS's like Cassandra. MapReduce is a way of parallelizing queries by separating them into sums.

Cassandra is a hybrid between BigTable and Dynamo storage systems. To access HBase or Cassandra, the Thrift interface is used. Both support super columns, which is where a column can be stored as a value in another column. Range queries are supported, but not all the operations of SQL.

In certain Social Networks the column store databases work well. However, they do have their issues since they are not too flexible due to the need for predefined schema. HBase and Cassandra are good for social data when the queries and information to be stored in is known in advance. They do well on range queries and have mapreduce due to Hadoop. The increased speed gained from column store plays well into the response time criteria.

Database Model	Examples	Uses
Relational	MySQL	Common when data entities have fixed-sized relations e.g. transactions between entities.
Column	Hadoop Hbase Cassandra	Well suited for applications like data warehousing, customer association management systems.
Document	MongoDB CouchDB	Adept for storing, retrieving and manipulating document-oriented information
Key Value Store	DynamoDB Riak Voldemort	Prefered for single query retrieval in real time applications.
Graph DB	Neo4j FlockDB InfoGrid OrientDB AllegroGraph	Supports many applications like Facebook in finding friends, other relations through graph search, etc...

Table 3 : Classifying databases by database model

C. Document Store

Document Store databases are databases where the information is stored in binary JSON (Abbreviated as BSON). Similar documents are grouped together for storage. This can be advantageous for a social network data model, in that requests for related entities are able to be completed quickly due to close proximity. For example, a user's region data could be stored in a user document, so only one request would

need to be made to retrieve this and other user information. This contrasts with the relational model, where retrieving the same data would require a request to the user table and a location table. The two implementations of Document store databases that are measured in the a paper are MongoDB and CouchDB.

There are some differences in the implementations of MongoDB and CouchDB. MongoDB, written in C++, is able to be accessed through BSON objects. These objects are able to represent many queries that can be written in SQL. [1] It also has indices on its collections, allowing for some queries that CouchDB is not able to handle.

CouchDB on the other hand allows access to its contents through a RESTful API. CouchDB also has multiversion concurrency control [1] and this allows it to auto-resolve conflicts. It also has a feature similar to SQL views for queries. There are two versions, one that is more permanent and updates every time a collection is associated with it, and the other is made in memory for each query.

D. Key Value

Key Value databases are databases that store data as key-value pairs. Values in these databases can be retrieved by a simple get operation for the specific key. The simplicity and lack of structure for key-value store databases allow for good scalability. However, this could be detrimental in modelling social network data for representing complex relationships.

Redis is an example of a key value store database with the ability to store sets and collections in addition to simple key-value stores. Redis supports other simple data structures like lists, ordered lists and sets, in addition to the standard key-value features. Master-replication (described in section IV) is supported to help scale reads. Sharding is also used to distribute writes to several server instances. This provides better performance of recovery in the event of node failure. Redis offers a shell client for simple interactions. In practice, Redis is not commonly used for social network databases because of to its lack of joins. Additionally, it's common that updating one value requires multiple changes in database, which has an impact on performance.

Riak is a Dynamo-inspired key-value store. The functionalities of Riak can be accessed through a REST interface. The data is stored based on distributed hash tables. Values often are formatted in JSON. A handy addition is that Riak supports links between two keys, which allows for simple traversal requests. Unlike Redis, Riak offers MapReduce [1]. So in practice Riak tends to work quite well. Since it meets the requirements of being able to scale well and have rapid response times.

E. Graph

Graph databases model relationships between entities with a graph, with the nodes representing entities and the relationship between entities represented with edges. The advantages are that graph databases support common traversal algorithms and graph algorithms, like single and multi source shortest path algorithms. Graph algorithms can be used for finding friends in a friendship network, or other complex relations in other network graphs [2].

Neo4j is a popular example of a graph database. It offers an object oriented API to store and retrieve data. The storage of data is optimized to make graph traversal efficient. Queries for Neo4j are created directly through its API, which allows complex queries. Neo4j supports the query languages SPARQL and Gremlin. It has graph algorithm component with implemented algorithms like Dijkstra's algorithms, which can be used directly in query code. Neo4j uses sharding for scalability, which is described in detail in section IV.

From a theoretical perspective a Graph database makes the most sense for modeling a social network. It is able to represent complex relationships between users, and benefits from the many graph algorithms available. Sadly there are a lot of issues implementation-wise. For example, Neo4j is a perfect fit for social data since it's graph based so it's easy to visualize and is good for the data model. The big downfall is that is does not scale well yet, so it has not been widely used. It also struggles to retrieve the data due to a lack of indices, so while it's good for intuitive storage, it has impacted performance on handling requests. As mentioned above, two major factors of being a good social network is the ease of scaling and the real time responses to users queries.

Additionally, using a graph database for a social network is limited because social networks usually come in conjunction with other critical data that doesn't neatly fit into the graph model. [3] For example, a university database with a student social network requires storage for much other university related data. Although a graph database would neatly store the social network aspect of the platform's data, there is much to be desired on the integration of data that isn't directly related to the network's entities and relationships. This therefore exacerbates the issues in graph database scalability. As seen in Table 1, Facebook, Flickr, and MySpace all use Neo4j in addition to another database, suggesting different storage methods are used depending on the type of data and its needs in regards to entity relationships.

IV. Distributed systems Pros and Cons

While we won't go too far in depth about Distributed Database systems, ease of distribution and its performance is a major factor in determining if a DBMS is an acceptable solution for a social network. This is because most social networks contain enough data that they can't be stored on one

machine, cluster, rack and often times they outgrow one data center. So more often than not the data is distributed over multiple physical locations. This means that the DBMS should maintain good responsiveness while being able to scale for the size of its user base and information stored.

One way that databases are distributed is through sharding. Sharding with the master-slave model consists of distributing data across nodes. All data in a node has a master copy, which indicates the data to be accessed upon requests to it. Each master has a number of copies, denoted as slaves. Upon the failure of the node containing the master, a slave is chosen to become the new master.

The method of failed node recovery is able to affect performance in various ways. This is such that the way masters and slaves are organized into nodes have the ability to prolong blocking requests due to overhead in searching for a slave to promote. [4] For example, organizing only masters into a node and only slaves into a node with no mixing means the need to search for replacement slaves in the event that the node containing all those masters fails. Providing read access to slaves in the event of failure also has the ability to mitigate the request blocking time in slave promotion. Similarly, the work needed to be done per node in the event of a failure can be reduced by evenly distributing masters and slaves across the network.

Additionally, synchronization management of masters and slaves can have an impact on performance. Syncing slaves on every update to the master provides much more work than updating on a schedule or on a delay. [4] However, giving too much of a delay between slave updates could provide discrepancies in data in the event of a master failure.

V. Experiments

Analyzing memory usage in social network databases is important to try to justify intuitions about how well-suited a data model is for social media data. Users also want to be able to find and connect with friends on social media, making it essential for searching and making connections to be fast and efficient. Mathew and Kumar [2] conducted experiments using a dataset obtained from Facebook which simulates a social network. First, they measured memory usage for various NoSQL DBMS's. In addition, for graph databases, they measured the time of operations related to specific graph algorithms, including graph insertions (of nodes and edges) and graph exploration. These operations are needed for users to find and connect with friends, so they are natural operations to analyze to determine which databases work best.

Mathew and Kumar conducted experiments on Ubuntu 14.04 LTS Dell Precision T3610 Tower Workstation with Intel Xeon E5-1620 v2 3.70 GHz. Figure 2 shows the memory use comparison for each of the NoSQL databases that were previously listed. This experiment was evaluated with a single node setup. The analysis found that document-store databases, like MongoDB and CouchDB, use less memory. This contrasts with graph databases, like FlockDB, Infogrid, OrientDB, and AllegroGraph, which use more memory. Neo4j however, is more memory efficient compared to other graph databases. One criticism of this paper is that the authors did not make interesting conclusions from their results. One conclusion we noticed was semi-structured and unstructured databases performed better than the more structured graph databases. However, trying to interpret this result for us was difficult, because the paper doesn't explain what the Facebook looks like or how they got the databases to store the dataset. We feel these could be interesting results that could be explored further by coming up with more precise experiments which o into a deeper analysis of the results.

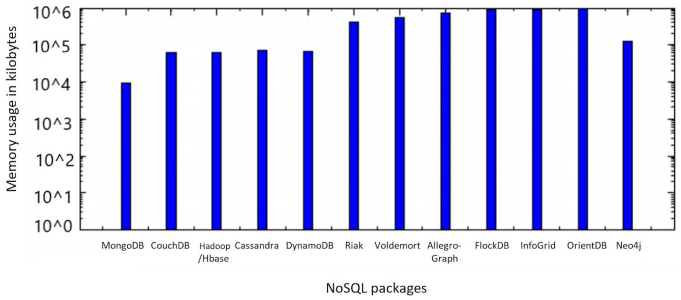

Figure 2: Memory Used by each of the NoSQL subcategories [2]

Mathew and Kumar placed a focus on the NoSQL graph databases in experimentation on the efficiency of different database operations. They focussed on the time it took to perform an insertion of a node. Additionally, they compared the cost of inserting relationships between nodes with multiple properties associated with each edge. Figure 3 shows the time in seconds for nodes to be inserted to the graph. Compared across graph databases, it appears Neo4j takes the least amount of time in general. The rate of change decreases as the number of nodes increased, which the paper did not explain or analyze. We think an easy improvement to this paper would be to extend the results to more nodes to see what the larger trend is, in order to further understand why there is this decrease.

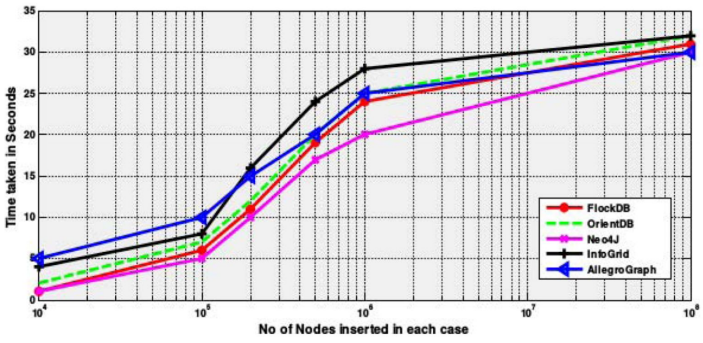

Figure 3 : Time taken during insertion of nodes and their properties to the Graph databases [2]

Figure 4 illustrates the time it takes in seconds to insert edges between nodes in the network graph. This likely models day-to-day operations better than Figure 3, as one would likely add many friends (edges) on social media, but only create one profile (node). Similar to insertions, compared across graph databases, it appears Neo4j takes the least amount of time in general.

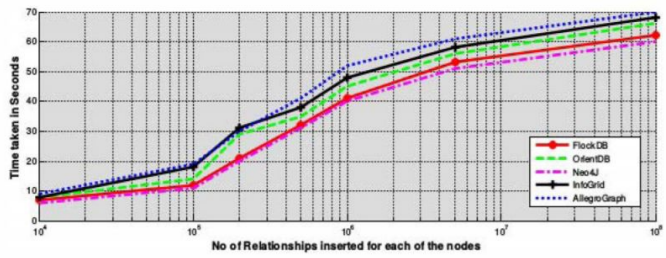

Figure 4 : Time taken during insertion of relationships and their properties to the Graph databases [2]

After comparing the time of insertion operations, Mathew and Kumar looked into the time taken for read operations on graph databases. Figure 5 shows the time in milliseconds it takes to find friends, mutual friends or family member details, for different quantities of nodes. Notice this is significantly less time than the insertion operations, likely because the graph is not modified during the read. One sees Neo4J and FlockDB take less time for read operations compared to other graph databases. When comparing Figure 5 to Figure 3 and 4, the databases that do better with the writing (insertion) operations do worse in the reading operations. This may be due to how the DBMS chooses to optimize insertions at the expense of fast read operations. For example, making it fast to insert new nodes (say, by inserting into memory/disk wherever there is space) may make it slower to explore the graph because it is not optimized for spatial locality (which is one of the principles of locality, suggesting that memory/storage locations near recently accessed locations will likely be accessed soon as well).

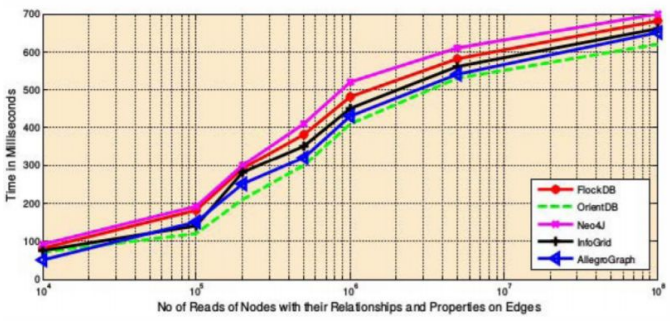

Figure 5: Time taken during reading of nodes with their relationships and properties on edges in Graph databases [2]

Overall, the paper is a good starting point in order to understand the memory usage of NoSQL databases in the context of social media, and how costly insertions and reads are in graph databases. Improvements to how the memory experiments are conducted, as well as more detailed analysis of those results, would greatly improve the memory section of the paper. Graph databases that do better with insertions operations tend to do worse with read operations, though naturally all the databases do worse when more information is read or inserted. Further analyzing how quickly these operations perform as you increase the amount of data inserted may be worth investigating in the future, as the quantity of social media users and data has been increasing rapidly over the past decade.

In *Social-Data Storage Systems* by Nicolas Ruflin, Helmar Burkhart, and Sven Rizzotti, performance experiments were conducted with multiple databases using a common data set to simulate a social network. The results support the discussion about the databases in regards to performance and structure tradeoffs. The benchmark used by this paper was on the dataset useKit. UseKit does a good job of modeling the graph structure of social media. However, it lacks some common data such as "likes," users' personal information, and timelines. One challenge that comes up when using a data benchmark for different databases is that each database structure is different, and forcing them all to work with this data set requires a lot of human effort. The experiment consisted of applying common queries to the data per database type, and recording the number of requests it took for each DBMS to complete them.

For example, the number of queries was compared for different databases to retrieve all information about a user. MySQL required 3 requests to resolve this data, while other databases, such as Cassandra, Riak, and HBase, only required 1 request [1]. There were a few databases that required 2 queries, which were Neo4j and Redis. One issue that comes up with the graph databases is that there will always be the two calls to get the node data and adjacency matrix. For the column store databases and key-value databases, only 1 request is required. This means that they are good choices for networks with a large amount of read requests.

Another result discovered on the useKit data set is that adding more information to the usernode will increase the query time of MySQL, Redis and Neo4j. So if one of these systems is used, then the designer should be aware of the potential costs later on for adding the storage of additional information.

The results discussed about the theoretical issues with the queries on the many DBMS solutions can be helpful in figuring out what is the best fit for a social network's DBMS. However, since the results of this experiment lack numerical evidence, it is difficult to truly extrapolate the best solution for

a social network DBMS. This is because one cannot apply these calculations in the context of their own performance and data model needs. For example, it is hard to know how much SQL is slowed down with the extra operations compared to document store databases, and how these results may change based on scale. It may be that the researchers needed more resources in order to get accurate numbers, but it makes it challenging to really compare the solutions.

VI. Case Study: LinkedIn Espresso

As one can see from the studies above, it is evident that there is no one true DBMS solution for social networking issues regarding performance and data model requirements. In some instances, it is beneficial for a custom DBMS to be developed for a platform, so that optimizations can be finely tuned in conjunction with very particular data models and desired behaviour.

LinkedIn developed their own document-oriented DBMS, called Espresso, to accommodate for highly specialized access patterns and storage criteria pertaining to the nature of the platform's data. This solution arose after identifying the limitations of relational databases in suiting the needs of the expanding data ecosystem. LinkedIn found that their data models did not cleanly fit into schemas suitable for an RDBMS, and required the ability to modify schema with backwards and forwards compatibility. Increasing the scale of the database also proved to be of great value in terms of both price and time. Cost also became an issue for LinkedIn in the amount of expensive hardware and cache support needed for the RDBMS in order to meet scale and latency requirements.

LinkedIn employed the usage of Voldemort, a key-value store DBMS, to handle certain portions of their data set which do not require strict timeline consistency. Based on their experience with this less structured model, the LinkedIn concluded that a NoSQL database structure would be beneficial for handling their data, but with augmentation to allow better timeline consistency. This lead to the decision of using document store for Espresso.

LinkedIn's data follows a common nesting hierarchy amongst similar entities. In Espresso, data is stored in partitions within nodes, where similar entities are grouped together under a common partition key. For example, a mailbox's messages and its statistics share the same partition key mailbox ID, so they are all stored together. Similarly, LinkedIn has company pages, so a company's products, services, and job listings would be stored in the same partition with a common company ID. All index files are in a B+ tree format and are stored within the same node as the data. Espresso improves on standard inverted indexing methods, such as using Apache Lucene, to allow a secondary index search. The choice of using inverted indexing is based in its optimization of text-based searching. This is done by prefixing each index with the element's partition group, with the effect that any requests for this item will be immediately routed to the desired node. Espresso also improves transaction support, as the majority of NoSQL stores' transactions are limited to a single document or record. Espresso allows multiple operations to occur within the same transaction for items within the same partition group. With the ability to atomically update multiple documents, all present in the same location, this aims to greatly reduce the overall time for read and write requests on related content.

In addition to the ability to group documents by group based on common partition keys, Expresso improves on RDBMS by its ability to modify schema in real time. Originally, a schema change required requesting an RDA to make the changes. This would result in potential downtime and extra work in properly integrating the changes with other producers or downstream consumers who interact with the related data. Espresso, on the other hand, allows schema changes to be made on-the-fly, and the schema is backwards compatible in the sense that others interacting with it don't need to immediately acknowledge the change, and operations can still function properly as they did before the change.

In experiments comparing prefix indexing and Lucene's full-text indexing, prefixed indexing consistently performed 2x better than Lucene in making numerous requests to small mailboxes containing 500 messages. [4] This was largely due to the latency of PUT requests, because Lucene is log-structured and therefore provides immutable files, so any update to a document requires it to be replaced and re-indexed. Lucene therefore has much more I/O to do on update whereas prefix indexing allowed the existing document and index file to be updated. In the case of prefix indexing, PUT and GET requests have the same fixed I/O cost. With large mailboxes of 50K messages, prefixed indexing also works better than Lucene, except when the ratio of GETS to PUTS is 0:100.[4] This is because the transaction time is much longer for prefixed indexing due to updating existing large files, whereas Lucene makes a new index file every time.

A primary feature of Espresso is its use of sharding its nodes. In particular, it uses the master-slave model such that all data and their indexes are stored in nodes called the master, and copies of these nodes are then made, called slaves. All requests for data in these nodes goes to the master, but upon failure of the master node, a slave node gets promoted as the new master. Partitions are assigned to a node under the conditions that there is only one master per partition, master and slave partitions are distributed evenly across all storage nodes for load balancing, multiple replicas of the same partition cannot be on the same node, and partition migration is minimized during cluster expansion. The goal in defining these constraints is to mitigate the effects of node failure and

allow quick response in the promotion of slaves to master regarding partitions within the failed node.

The developers at LinkedIn performed experiments between Espresso and sharded MySQL, used on their older system, in order to compare response time in the event of node failure. Espresso outperformed sharded MySQL for every experiment, regardless of the number of partitions. This is due to the partition allocation for each system. With sharded MySQL, a node can either contain master partitions or slaves, not a mix of the two. In the event of a master node failure, a slave node with slaves for all related partitions on the failing node must be promoted to master. When a failure occurs, Espresso has the failed partitions evenly distributed across the cluster so each running node doesn't need to do as much work. Additionally, Espresso allows slave reading to be enabled so a request doesn't need to be fully blocked until the master promotion is completed. Therefore, Espresso's method of gradually handling failed partitions, with immediate request response, outperforms sharded MySQL's recovering method in which requests are blocked until a fully new master node is established.

VII. Further Work

While the papers discussed a lot about the performance of database management systems on a few different solutions, it does not seem like there is much work investigating how the network's population distribution affects the scale. For example, most cities have a higher population and the people within the city are more likely to be connected to each other. So is it possible to have somewhat disjoint databases for distant regions? Is there any gain in performance from that offset by having smaller data sets? It seems significantly less likely that social network users would query people who are distant from their network/ region. So maybe the issue of distribution could be "capped" at a certain number of users since there is some average closed circle network size.

In order to investigate this further, it would require access to multiple social networks' user databases and some way to graphically analyse them. Gaining such access itself would pose a challenge just by the potential confidentiality major social networks would impose on their data patterns and DBMS implementations. But from there it is more of a challenge of organizing the social circles into disjoint groups and seeing how that connects with physical locations.

Conclusion

After surveying multiple papers relating to social network data and storage solutions, it can be concluded that social data is a very hard to categorize. There is currently no solution that works best as far as modeling, accessing and modifying the data. It is also extremely difficult to compare social networks performance on different databases with a common data set due to the amount of data wrangling required to ingest the data set [1].

Graphical databases do a great job in representing the data for analysts to investigate, but due to low use and scaling issues it is not widely adopted. The papers surveyed indicate that there are a lot of pros and cons to each database, showing a general tradeoff between structure and scalability. Based on this information, it is truly up to the social network company to pick the best one to suit their data models, relationships, and access patterns.

From the case study on LinkedIn, we see that the best solution is to take an existing database system and augment it to have the best possible performance on the given network. This can be done in many ways, two of which are Facebook adding a cache on top of an existing db [1] and LinkedIn's development of Espresso [4]. There most likely are newer solutions to this issue that are implemented in practice, but because of the nature of social networks competing it is not yet public information.

Acknowledgment

We would like to thank Professor Sina Meraji for allowing us to use an extra source from before 2013 after a post on piazza (see question 135) by Juliana.

References

[1] Nicolas Ruflin, Helmar Burkhart, and Sven Rizzotti. 2011. "Social-data storage-systems. In Databases and Social Networks" (DBSocial '11). ACM,NewYork,NY,USA,7-12. DOI=http://dx.doi.org.myaccess.library.utoronto.ca/10.1145/1996413.1996415

[2] A. B. Mathew and S. D. Madhu Kumar, "Analysis of data management and query handling in social networks using NoSQL databases," *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Kochi, 2015, pp. 800-806.

[3] Sara Cohen. 2016. Data Management for Social Networking. In Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '16). ACM, New York, NY, USA, 165-177. DOI: https://doi-org.myaccess.library.utoronto.ca/10.1145/2902251.2902306

[4] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, ZhenZhang, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. 2013. "On brewing fresh espresso: LinkedIn's distributed data serving platform." In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13). ACM, New York, NY, USA, 1135-1146. DOI: http://dx.doi.org.myaccess.library.utoronto.ca/10.1145/2463676.2465298