UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

WeST People and Knowledge Networks
Institute for Web Science and Technologies

Scale-out evaluation of news feed retrieval algorithms on Neo4j and Titan clusters

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Informationsmanagement

vorgelegt von
Sebastian Schlicht

Erstgutachter:	Prof. Dr. Steffen Staab Institute for Web Science and Technologies
Zweitgutachter:	Rene Pickhardt Institute for Web Science and Technologies

Koblenz, im April 2015

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	☑	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☑	☐
Der Text dieser Arbeit ist unter der Creative Commons Lizenz CC-BY-SA 3.0 verfügbar.	☑	☐
Der Quellcode ist unter der GNU GPL v2 Lizenz verfügbar.	☑	☐
Die erhobenen Daten sind unter der Open Database Lizenz ODbL verfügbar.	☐	☑

.....

(Ort, Datum) (Unterschrift)

Abstract

News feed platforms, such as Facebook and Twitter, are growing continuously. Their primary requirements are scalability, low request latencies and high availability [7] for read and write requests. This requires to scale out the system to multiple machines [10]. STOU and Graphity were reported as high performant algorithms to power a news feed system. However, their evaluation took place on a single machine with one thread, the algorithms didn't support to run in a distributed system. We identified a proper experimental setup for the evaluation of *scale-out*. We implemented versions of STOU and Graphity, that support a distributed execution. We evaluated the news feed algorithms on two distributed graph databases, that use different distribution strategies: a) Neo4j with its database replication and b) Titan using Cassandra, that bases upon distributed hash-tables. Cassandra is an eventually-consistent database. Neo4j shows good read *scale-out* properties, for both STOU and Graphity, but its database replication approach can't provide write *scale-out*. On Titan Graphity provides better read *scale-out* than STOU, while STOU shows good write *scale-out* properties. We weren't able to implement Graphity in a manner, that would support concurrent writes in a distributed system. Graphity seems too complex, to be used on eventually-consistent databases. With caching enabled, its read performance is similar to STOU, while its runtime complexity for writes is higher. We conclude that STOU is more suitable to run in a distributed system. Both distribution strategies have strengths that can be used to power a news feed system and a combination of both approaches will show the better *scale-out* properties.

Zusammenfassung

News Feed Plattformen, wie Facebook und Twitter, erfahren ein stetiges Wachstum. Hauptanforderungen sind Skalierbarkeit, geringe Latenzen und hohe Verfügbarkeit [7]. Insbesondere muss ein hoher Durchsatz an Schreibanfragen erreicht werden. Dies erfordert, das System horizontal zu skalieren, auf mehrere Maschinen zu verteilen [10]. STOU und Graphity wurden als hoch performante Algorithmen beschrieben, mit denen ein News Feed System betrieben werden kann. Allerdings fand ihre Evaluation auf einer einzelnen Maschine mit einem Thread statt. Die Ausführung in einem verteilten System wurde in ihrer ursprünglichen Form nicht unterstützt. Wir haben ein Versuchsaufbau erarbeitet, um *scale-out* zu evaluieren. Wir haben STOU und Graphity so implementiert, dass sie eine verteilte Ausführung unterstützen. Wir haben die Algorithmen auf zwei verteilten Graphdatenbanken evaluiert, die unterschiedliche Verteilungsstrategien verwenden: a) Neo4j mit ihrer Datenbankreplikation und b) Titan mit Cassandra, das auf verteilten Hash-Tabellen basiert. Cassandra ist eine Datenbank, die keine unmittelbare Konsistenz der Daten garantiert. Neo4j zeigt gute Lese-*Scale-out*-Eigenschaften für STOU und Graphity, aber die Datenbankreplikation unterstützt den *Scale-out* von Schreibvorgängen

nicht. Auf Titan zeigt Graphity bessere Lese-*Scale-out* als STOU, während STOU gute Schreib-*Scale-out* beweist. Wir konnten Graphity nicht so implementieren, dass simultane Schreibvorgänge im verteilten System möglich wurden. Graphity scheint zu komplex zu sein, um auf einer Datenbank ohne garantierter Konsistenz zu laufen. Mit aktiviertem Caching ist dessen Lese-Performanz vergleichbar mit der von STOU, während die Laufzeitkomplexität höher ist. Wir folgern, dass STOU geeigneter für ein verteiltes System ist. Beide Verteilungsstrategien haben Stärken, die zum Betreiben eines News Feed Systems genutzt werden können. Die Kombination beider Ansätze wird bessere *Scale-out*-Eigenschaften zeigen, als der jeweilige Ansatz für sich allein.

Contents

1 Introduction

News feed systems are typical features of social networking platforms, such as Twitter and Facebook [17]. Primary requirements for these platforms are scalability, low request latencies and high availability [7]. Social networking platforms globally experienced growth in the last few years. For example, the number of monthly active users of Facebook worldwide grew from 100 million in 2008 to 1.1 billion in 2013 according to estimations.[1,2] Table 1 shows the growth of the Facebook graph from 2007 to 2012. The statistics support the estimations and show that the total number of users and friendships increased as well. Of course, next to users and friendship edges social network graphs include content items and their relationships to users. If including meta data, social network graphs with such dimension clearly exceed the memory limit of servers. The data size requires considerable storage and can grow by 100 GB per day [13]. A platform needs to be highly scalable to support such continuous growth [10], [15]. Scale, in this context, more precisely means scale out. One strategy to scale out a storage system is to replicate the database and keep the replicas up-to-date. This strategy provides ACID transactions and can support any level of data consistency. Traditional relational databases are capable of the database replication approach, but the support is still limited [10]. Another solution is to use distributed hash-tables that distribute the key management responsibility. Storage systems using this distribution strategy are eventually-consistent and aim to provide high write throughput [15].

STOU and Graphity were reported as high performant data structure algorithms to power a news feed system that eliminate the data redundancy [17]. However, the algorithm evaluation was restricted to the response time aspect of performance on one single-threaded machine. It didn't reveal if the algorithms scale out. Neither STOU nor Graphity were implemented in a manner that would support a multi-

Year	User nodes	Friendships	User node degree	News items / s[3]
2007	13M	644.6M	50	1.4K
2008	56M	2.1G	38	2.9K
2009	139.1M	6.2G	46	5.8K
2010	332.3M	18.8G	57	11.6K
2011	562.4M	47.5G	84	23.1K
2012[4]	721.1M	68.7G	95	46.3K

Table 1: Development of the number of user nodes and friendships, the average user node degree [2] and the estimated number of news items per second in Facebook.

[1] `http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/` (last access on 2015/04/28)

[2] `http://www.statisticbrain.com/facebook-statistics/` (last access on 2015/04/28)

[3] Estimations according to the law of sharing, includes shared items and media items.

[4] The data points belong to an unknown time in 2012.

threaded or distributed execution at all. Our contributions are the following:

- We surveyed evaluation approaches of the ability of distributed systems to scale out. In this context we clearly separated scalability, *scale-up* and *scale-out*. We identified four key factors of the evaluation approaches that can be categorized into:

  1. Metrics (benchmark metrics)
  2. Requests (request generation and issuing)
  3. Data size (total data size)
  4. Cluster setup (system architecture and hardware)

  We discussed proper choices regarding to these key factors and summarized a suitable strategy to evaluate *scale-out*.

- We reimplemented STOU and Graphity in a manner that supports multi-threaded distributed execution. The algorithms were implemented on two graph databases, Neo4j and Titan on Cassandra, that use different distribution strategies, in order to scale out the news feed service.

- We implemented an evaluation framework that allowed to issue multiple requests to nodes in a cluster concurrently, as demanded by our *scale-out* evaluation strategy.

- In our *scale-out* evaluation on a social network graph, we show that the read throughput of STOU and Graphity increases linearly with the cluster size. We show that Graphity doesn't provide higher read throughput than STOU in a *scale-out* scenario, if caching is enabled. We show that the write throughput of STOU on Titan increases linearly with the cluster size. Database replication, as in Neo4j, doesn't provide write *scale-out*.

The remainder is organized as follows: The survey on scalability evaluation approaches for distributed systems is presented in section 2. This includes a summary of existing approaches, key factors extracted from the approaches and the description of a proper strategy to evaluate *scale-out*. In section 3 we formalize the news feed retrieval problem and introduce and compare the news feed retrieval algorithms STOU and Graphity. Section 4 describes common strategies to distribute a database, in order to scale out. We present technologies used for and details of the implementations of STOU and Graphity, that support a distributed execution. The experimental setup and the results of the *scale-out* evaluation are presented in section 6. We conclude the results in section 7 and discuss our next steps in section 8.

Please note that the algorithm implementations, the server plugins and the benchmark code are available on Github[5] under an open-source license, together with all configuration files that were used.

[5] `https://github.com/sebschlicht/bachelor-thesis`

2 A Survey on the Scalability Evaluation of Distributed Systems

At first there is a need for a proper experimental setup, to evaluate the scalability of distributed systems. We look at existing evaluations of distributed systems, in order to design a proper setup. Several distributed storage systems were released in the last decade. Examples are Amazon Dynamo [10], Cassandra [15], Yahoo! PNUTS [7] and Google Bigtable [5]. Main goals of the distributed storage systems were high-availability and scalability. Papers that introduced a distributed storage system thus often evaluated the scalability of the new technology. In most cases the evaluation was a short section in the work, since the focus was on the introduction of a new storage technology. The evaluation approaches are varying and many choices were made without any statements supporting the one or the other option. At first we define and point out the aspects of scalability, to compare related work without confusing equivocally usages of terms related to scalability. We summarize the evaluation approaches, in order to identify key factors. We present the key factors identified and discuss available options to find the most appropriate option for our *scale-out* evaluation. The section concludes with an evaluation strategy based upon the choices made per key factor.

2.1 Scalability: Scale-Up vs. Scale-Out

Scalability is the ability to scale, to process an increasing volume of work and/or the ability to enlarge [3]. A system is called "scaling", if it shows scalability. There are two variants of scalability: 1.) *Scale-up* and 2.) *Scale-out*. *Scale-up* is the ability to scale up, to scale by increasing the hardware resources available on the machine. *Scale-up* is also called the ability to scale vertically. *Scale-out*, in contrast, is the ability to scale out, to scale by adding a machine to the system. The machines are nodes in a cluster with a size n, where the cluster size n refers to the number of nodes. To scale out means to increase n, in particular beyond the value 1. Programs are executed in parallel on each cluster node. Thus the algorithms have to support distributed execution, in order to allow the system to scale out. Analogous to *scale-up*, *scale-out* is called the ability to scale horizontally. When referring to scalability, we don't explicitly specify which variant of scalability is used and want to express, that any form of scalability is agreeable. If we want to specify the variant, we will use *scale-up* or *scale-out*. We transform statements from related work to separate the variants of scalability, where applicable. In the context of distributed systems most of the work actually used the term scale in the meaning of scale out.

2.2 Approaches to Evaluate Scale-Out

To evaluate the scalability of Dynamo, Amazon used it as the storage backend for several services in their production environment. For one month, in the peak season

December, Amazon measured the average and the 99.9th percentile of Dynamo's read and write request latencies. The requests were caused by their shopping customers. During the measurement Dynamo ran on a cluster with a couple hundred of nodes spread across multiple data centers.

To evaluate the *scale-out* of PNUTS Yahoo! measured its average request latency. Since Yahoo! did not instrument its customers in the evaluation of PNUTS, they had to fire requests on their own. Three clients with 100 threads each, resulting in 300 client threads, fired requests with a fixed request rate of 1,200 requests/s against the PNUTS cluster. The database contained synthetically generated 1 kB records. 90% of the requests were read requests that retrieved a record, the remaining requests updated one half of a record. Instead of a large cluster with hundred of nodes PNUTS ran on a cluster with 6 up to 15 nodes, spread across multiple data centers. $^2/_3$ of the nodes ran on machines with dual-core CPUs and 4 GB memory. The remaining nodes ran on machines with quad-core CPUs instead. PNUTS is split up into several service layers and each node is part of one of these layers. Two of the layers are the storage layer and the controller layer. The PNUTS system thus consists of storage nodes, tablet controllers and nodes that are part of one of the other service layers. In the evaluation they varied the size of the storage cluster. 80% of the requests targeted the local data center.

Google evaluated Bigtable's *scale-out* according to the cluster throughput. It is also split up into multiple service layers. Bigtable consists of GFS cells that are part of a storage layer and tablet servers that store meta data. In the evaluation the storage cluster of GFS cells had a fixed size of 1786 nodes. Instead the number of tablet servers was varied from 1 to up to 500. One client per tablet server fired requests against the Bigtable cluster. Read and write requests were separated and targeted a 1 kB record each. In every benchmark 1 GB of data were read or wrote per tablet server. The machines in use had dual-core CPUs and enough memory to hold the working set of the applications running. Some machines were shared between different applications. Each machine ran a GFS cell and at maximum one tablet server or one client. In addition the cluster nodes weren't isolated from the Google network. Some machines ran applications that weren't related to the benchmark. The round trip time in the network was below 1 ms.

In addition to these evaluation approaches, Yahoo! developed YCSB, a benchmark framework for cloud serving systems. The framework aims to evaluate the performance and scalability of cloud serving systems, that provide online read and write access to data. Cloud serving systems aim for *scale-out*, high availability and elasticity, where elasticity is the ability of a system to scale out while running [8]. To evaluate the performance of the database under load, the framework measures request latencies and the throughput. The framework uses read-heavy and write-heavy workloads that include the insertion of data. In an experimental setup, the framework was used to evaluate several distributed databases. The number of storage nodes of the database was varied from 2 to 12. A single benchmark client issued synthetically generated requests. The generated workload was read-heavy and each

	Metric		Requests						
	Latency	Throughput	Read <> Write	Inserts Data	Request Rate	Client Count	Data Generation	Data Size	Cluster Varied
Dynamo	✓		✓	✓	?	?	users	?	storage
Bigtable		✓	✓	?	prop.	n	syn. 1kB	prop.	service
PNUTS	✓				static	3	syn. <1 kB	static	storage
YCSB	✓	✓	(✓)	✓	optimized	1	syn. 1kB	prop.	storage

Table 2: Key factors in the *scale-out* evaluation approaches of distributed storage systems introduced in the last decade. Proportional is proportional to n, n refers to the cluster size.

request read, updated or inserted 1 kB of data. The request rate and the size of the data stored in the storage cluster was increased proportionally with the number of storage nodes. Quad core machines with 8 GB memory were used for the benchmarks. The evaluated databases were tuned.

2.3 Key Factors of Approaches to Evaluate Scale-Out

Approaches to evaluate *scale-out* were summarized in subsection 2.2. In this section we present key factors of the evaluation approaches. An overview of the key factors and the choice made in related work is given in Table 2. We discuss the options available for each key factor, in order to find the most appropriate choice to evaluate *scale-out* of a news feed systems.

2.3.1 Metrics

The most prominently used metric in the evaluation approaches is the average request latency.

$$\bar{L} = \frac{1}{n_r} \sum_{i=1}^{n_r} L_i \tag{1}$$

The average request latency $\bar{L}$ is the arithmetic medium over the latency L of every request. n_r refers to the total number of requests that were processed in the benchmark. Request latencies are measured between the client and the service cluster. Amazon also measured the 99.9th percentiles of request latencies, since they consider it an indicator of high service quality for all customers [10]. Google measured

the throughput of the service cluster instead of request latencies.

$$\bar{T} = \frac{1}{t} \sum_{i=1}^{t} T_i \tag{2}$$

The average throughput $\bar{T}$ is the arithmetic medium over the throughput T of each second of the benchmark. The throughput of a second is the number of requests processed by the cluster during this second. t refers to the benchmark duration in seconds. In fact there is a trade-off between request latencies and throughput. Request latencies of a system increase when its load increases. Some see the request latency as an indicator for service quality [14] and service level agreements of Amazon and Yahoo! most prominently include the expected request latency distribution [10], [7]. The YCSB evaluation increased the potential throughput until the actual throughput did not further increase, while measuring request latencies as well. The platform of a news feed has to be highly scalable in order to support continuous growth [10]. One cause for higher throughput is a higher number of active users, which we consider an aspect of a growing platform. Thus throughput is a key metric to evaluate the scalability of a system, including *scale-out*. Request latencies, in contrast, indicate the service quality for a user. In this evaluation we will use both metrics, in order to evaluate the *scale-out* qualitatively. A database with a good ability to scale out should provide a linear increase of throughput with acceptable request latencies, when adding more nodes to the cluster [8].

2.3.2 Requests

The evaluation approaches greatly differ in the benchmark requests. This includes:

- request types
- request rate
- number of clients issuing requests
- data and data size read or wrote per request

Request Types Most of the evaluation approaches separate between read and write requests. Note that a write request can either be an insertion, an update or a deletion. In conclusion we identify the four request types of CRUD. The composition of the request types is fundamental for the evaluation, but depends on the service. In the evaluation of PNUTS Yahoo! used a fixed request composition, where 80% of the request were read requests, which makes it hard to state the *scale-out* of PNUTS for reads and writes separately. On the other hand, there might be interdependencies between read and write requests when executed concurrently, e.g. due to schema violations. Since there are read heavy and write heavy services, it is important to state the *scale-out* for reads and writes separately, instead of using any or an average

request composition. A hybrid approach is to issue read and write requests concurrently, but repeat the benchmarks for read heavy and write heavy workloads, an approach used by the YCSB framework. This approach still allows to make statements for read and write heavy services separately without strictly separating the requests. It is very unlikely that a system will run in a production environment, where reads and writes are strictly separated. However, it is more clean to strictly separate the requests in a scientific evaluation. *Scale-out* evaluations therefore should strictly separate read and write requests. The write request composition depends on the service as well, which makes it impossible to define an optimal solution for every case. An insertion request may lead to a growth of the data size, while an update request, that overrides an existing value, may not. A growing platform is likely to produce more data, thus a *scale-out* evaluation at least should include insertion requests.

Request Rate The request rate r is the maximum number of pending requests at any point in time.

$$P(t) \leq r \tag{3}$$

$P(t)$ refers to the number of concurrently pending requests at time t. Thus the request rate equals the maximum number of concurrently executed requests and the maximum number of open connections to the cluster. The request rate can affect the maximum throughput of a system. A higher request rate may enable the system to parallelize more work internally and provide higher throughput. For example, read requests may trigger disc hits, that the OS can buffer and batch process. However, there will be a point where a higher request rate won't further increase the throughput, but solely increase request latencies. In the disc hit example this point is reached when the disc buffer runs full. Thus there will be an optimal request rate per service and cluster size, where the throughput is at maximum. One approach to identify the maximum throughput is to increase the request rate as long as the throughput is increasing [8]. As stated, throughput is the key metric in a *scale-out* evaluation. Thus one should maximize the throughput in each benchmark of a distributed system.

Number of Clients The number of clients limits the maximum request rate, especially if requests are issued serially. While Google uses as many clients as service nodes, Yahoo! uses very few clients or a single client. Yahoo! stated, that the single client wasn't a bottleneck in their evaluation, since the client threads were waiting for database responses most of the time [8]. This evaluation noted a maximum update throughput of 12,000 write requests per second on a Cassandra cluster with 6 nodes. However, the Cassandra cluster was tuned. Due to differences in hardware setup, software configuration and workloads we don't expect a higher throughput and should be able to use a single client as well. However, when maximizing the throughput by increasing the request rate we should watch the client load to make sure it wasn't a bottleneck.

Request Data The network bandwidth available between the clients and the cluster limits the maximum throughput. Thus the data size read or wrote should match the average size of a read or write request, to get meaningful results. One approach is to use user-generated requests that show this data size naturally. For example, Amazon used user-generated requests to evaluate Dynamo in a production environment. Another approach is to use synthetically generated data for the requests and to aim for a data size matching the (expected) average request size. If the system being evaluated is a storage system that will power other systems, it may be hard to find a proper value for this data size. To repeat the evaluation for all systems the storage system could power may be very time consuming. Synthetic request generation was used in most of the evaluation approaches. The data size used in the storage system evaluation approaches was 1 kB per request. PNUTS, for example, was evaluated with $^1/_2$ kB data per write and 1 kB data per read request. To our best knowledge there is no reason for this particular data size, but we state that 1 kB data is sufficient for most (atomic) storage system requests in a service oriented architecture.

2.3.3 Data Size

The evaluation approaches differ in the total data size and if the data size is kept constant or increases proportionally with the cluster size. This applies to initial data, as well as the data read or wrote during a benchmark. Initial data is necessary to run read benchmarks but can be necessary to run write benchmarks if insertion requests are lacking. The data read or wrote during a benchmark depends on the total number of requests and the data read or wrote per request. In the evaluation of PNUTS the data size was kept constant. Bigtable was evaluated with a data size that was increased proportionally with the cluster size n. The YCSB framework increases the data size proportionally to n as well. Continuous growth of a platform requires scalability. Continuous growth includes growth of the data size, due to a larger number of active users, an increasing usage of the service and new features. A news feed with user-generated content obviously contains more data when growing. Thus the data size should scale proportionally with the cluster size, to evaluate *scale-out* of a news feed system. In contrast, the data size scale is independent of the *scale-out* of a system. Though a large data scale can be recommended, as this is the use case for a scale out scenario.

2.3.4 Cluster Setup

The cluster was a key factor in the evaluation of distributed systems. This includes the service architecture, the network topology and the hardware setup. We identified two layers in the service architecture, that we had to separate. At least there is a service layer. We call nodes of the service layer service nodes and they form a service node cluster. Service nodes accept requests from clients, collect data from the storage system and execute the business logic of the application. Finally they

respond to the clients with the computed results. In some services the storage system is included, but in other services it is separated from the service nodes. We thus identify a second layer, the storage layer, that isn't present in all services. Storage nodes can be seen as standalone nodes that form a second cluster, parallel to the service node cluster. Storage nodes accept requests from service nodes and provide read and write access to the data stored. An increase of the cluster size is undefined, if both clusters are present. We can either increase the size of the service cluster, the size of the storage cluster or the size of both clusters. When increasing the service cluster size, there will be a point when the storage layer is saturated. An additional service node might not further increase the throughput, but increase storage request latencies. When increasing the storage cluster size, there will be a point when the service layer is saturated. An additional storage node will lower the storage request latency but will not increase the throughput of the service layer. Depending on the complexity of the business logic and if there are additional requests necessary, e.g. to an index service, one or the other scenario is more likely. Most of the evaluation approaches varied the number of the storage nodes. In the evaluation of Bigtable Google varied the number of service nodes. To avoid the saturation of one of the layers, we should always increase the number of service nodes and the number of storage nodes simultaneously.

Another aspect of the cluster is the hardware configuration of a node. The performance per node in the cluster doesn't affect the evaluation of the *scale-out*. Though to scale up the nodes increases the throughput, the increase of the throughput when adding nodes should still be proportional to the cluster size. However, there might be a minimal hardware configuration. We don't want the OS and the service to fight for resources. Thus we should at least use dual-core CPUs to ensure that at minimum one core is available for the service. The other core(s) will still have to switch contexts between threads of the OS and the service. We should provide enough memory to run at least the OS and the service without any caching. Besides these very basic rules we can not define a hardware setup that is suitable to evaluate every service and its technology stake. Deep knowledge of the technologies in use can be necessary, in order to tune a hardware setup for a certain service with and without caching enabled. Tuning the hardware setup might end up in different hardware setups for different technologies evaluated. The setup's cost effectiveness could help to make the evaluations comparable in such cases. However, we suggest to keep the default settings and provide an amount of memory common to modern computers, that matches the basic criteria above.

2.4 Summarized Scale-Out Evaluation Strategy

In subsection 2.3 we discussed characteristics of a proper strategy to evaluate *scale-out*. We now describe a strategy, that takes these characteristics into account. In an evaluation one should measure the throughput of the service and the average request latency. Request latencies should always be acceptable, but throughput is

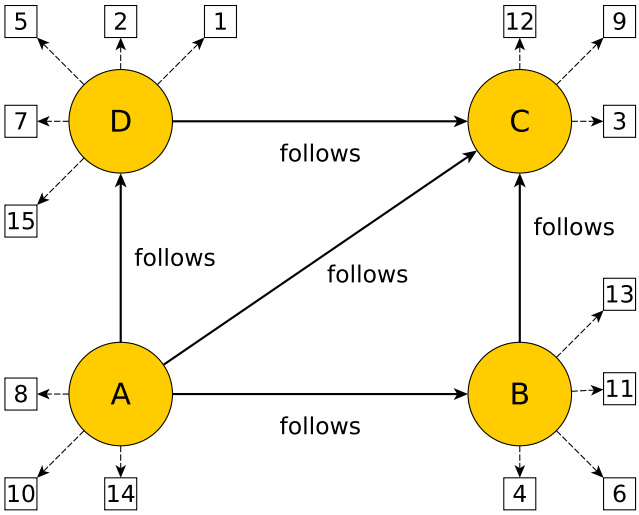


Figure 1: Social network of user A including news items

the key metric. The throughput should scale proportionally with the cluster size n, where n describes the number of service nodes and storage nodes simultaneously. Each node should, at minimum, run on a dual-core machine with enough memory to run OS and the technology in use. Additional memory is necessary if caching is enabled. The size of the data loaded before or wrote in a single benchmark will scale proportionally with the cluster size n, to indicate the continuous growth of the service. A single client can fire requests against the service cluster. All CRUD request types should be instrumented, but read (R) and write (C, U, D) requests should be strictly separated. The request rate should be increased in order to find the maximum throughput. The request size should match the average size of a user-generated request when evaluating a service. The request size should be 1 kB when evaluating a general-purpose storage system.

3 Fundamental

Fundamental to our evaluation is the news feed retrieval problem, formalized in subsection 3.1. The news feed retrieval algorithms STOU and Graphity are solutions to this problem. We describe the two retrieval algorithms and their graph models in subsection 3.2. We compare their theoretical runtime complexities and describe our expectations concerning the evaluation results. In subsection 4.1 we describe approaches to distribute a system. Rather than implementing distribution approaches ourself, we used existing distributed graph databases that are described as well.

3.1 Formalization of the News Feed Retrieval Problem

Figure 1 shows a small social network. There are two kind of elements that can be related to other elements. Suitable for this setting is the graph notation where elements are called vertices and relations are called edges. Circular vertices represent users such as user a. Two users can be linked by a $follows$ edge. This directed edge represents a directed friendship. For example, user a follows user b. Rectangular nodes are news items. The number of a news item represents its creation time, a higher value indicates a more recently created item. Examples for these news items are generated items due to profile updates, the usage of apps or changes in the user's subscriptions. In addition, there are user-generated items such as photos, music, videos or simple text messages. The sharing of an existing news item can also be a new item. The news feed of a user is the sorted and filtered aggregation of news items created by the user's friends. Again, friendships are indicates by the directed $follows$ edges. We call the friends of a user its ego network. For instance, the ego network of user a is the set b, c, d. A simple sorting of the news feed is to sort the news items by their creation time. The most simple filtering is using no filter at all. Typically only the first k news items in the news feed are retrieved and items at position $k_1 > k$ are cut off.

We define some notation in order to formalize the problem of retrieving the news feed for a user, based on the formalization in the Graphity paper [17].

- $G(V, L, E)$ is a graph with a finite set of vertices V, a finite set of edge labels L and a set of edges $E \subset V \times L \times V$.
- The set of vertices V is a complete partitioning $V = A \cup C, A \cap C = \emptyset$ into aggregating vertices A and news item vertices C.
- We map the vertices to a property map $V \rightarrow$ *Map*<*Object*, *Object*>. Thus each vertex $v \in V$ is assigned a property function f_v to retrieve its properties. The property function maps property keys to property values, formally f_v : *Object* $\rightarrow$ *Object*. Edges can have properties, too. This is known as the property graph model.[6,7]
- Valid property keys are defined per vertex/edge type. The property key set of aggregating vertices is $\emptyset$. The property key set of news item vertices is $\{d, t\}$, where the property value for d indicates the data (or content) of the news item. t is mapped to the time stamp of the news item's creation. We can expand each property map if necessary.
- L is the set $\{follows, published\}$ and these edge labels reflect the partitioning of $V = A \cup C$. $follows$ edges are directed edges from an aggregating vertex a to an aggregating vertex b such that $b \neq a$. Every aggregating vertex

[6] `http://neo4j.com/developer/graph-database/#property-graph`
[7] `https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model`

a can have any number of incoming and outgoing $follows$ edges. In contrast $published$ edges lead from an aggregating vertex a to a news item vertex $c \in C$. Each news item vertex c must have at least one incoming $published$ edge, that indicates its author.

- We define $NewsBy(a) = \{c \in C | (a, published, c) \in E\}$ to be the set of all news item vertices created by an aggregating vertex a.
- We define $Subscriptions(a) = \{b \in A | (a, follows, b) \in E\}$ to be the set of subscriptions of an aggregating vertex a. d_{out} refers to the size of this set.
- We define $Followers(a) = \{b | a \in Subscriptions(b)\}$. d_{in} is the size of this set for an aggregating vertex a.
- We call $NewsFor(a) = \{c \in C | \exists b \in Subscriptions(a) : c \in NewsBy(b)\}$ the news items for an aggregating vertex a.
- For the news item sorting, we define a sort function $Sort : C \to \mathbb{N}$. It maps a news item to its position in the news item set, formally $c \mapsto i_c$.
- In order to filter the news items, we define a filter function $Filter : C \to C$.

With this notation we can define the news feed for an aggregating vertex $a \in A$ as

$$NewsFeed(a) = Filter(Sort(NewsFor(a))) \tag{4}$$

where $Filter$ and $Sort$ are generic functions. A basic $Filter$ function is the identity: $c \mapsto c$. The result is the input set C, as if we didn't filter the news feed at all. To retrieve the top-k news items we need a filter function, that cuts off news items with a position $i > k$ and thus limits the news feed to the first k news items. A basic $Sort$ function could sort the news items descending by their creation time stamp $f_c(t)$.

However, sorting of the news feed is not limited to the creation time stamp. Facebook for example takes the interaction of friends (number of comments and likes) and the type of the news item into account [11]. There are four main factors that Facebook looks at, when sorting the news feed [1], [6]:

1. your interaction with the author (e.g. number of likes)
2. the interaction with the news item of your friends and all users (e.g. number of likes/complaints)
3. your interaction with news items of the same type (e.g. number of likes of other photos)
4. complaints by any user targeting either the news item or its author

Multiple users complained about the new sorting and preferred the simple time sorting. Thus in 2015 the user can decide whether to use the more complex sorting introduced in 2012 or the simple time sorting. However, the new sorting algorithm still sorts the news items by time: Crucial is the time when a news item got visible for a user, rather than its creation time stamp.

With properties in vertices and edges we can model a sort function that takes Facebook's new sorting criteria into account. For example, we could introduce an edge label $likes$, that indicates a user's attitude towards a news item or an user.

$$L = \{follows, published, likes\} \tag{5}$$

To retrieve the number of likes to the author's news items, we can define:

$$LikesToItemsFrom_a(b) = |\{c|c \in NewsBy(b) \wedge \exists(a, likes, c) \in E\}| \tag{6}$$

We can define

$$LikesOfFriendsTo_a(c) = |\{b|b \in Subscriptions(a) \wedge \exists(b, likes, c) \in E\}| \tag{7}$$

to retrieve the number of your friends' likes to a particular news item. In addition we could add a property y to news item vertices, that represents the type of the news item. To retrieve the number of likes to news items of a specific type, we can define:

$$LikesToType_a(type) = |\{c|c \in NewsFor(a) \wedge f_c(y) = type \wedge \exists(a, likes, c) \in E\}| \tag{8}$$

Of course, these are absolute values that include every news item. Since user preferences may change over time, it may be better to ignore old news item and limit the number of news items considered for the sorting. Consider a function $LastRecent$ that reduces the view port to at most 100 news items created during the previous 30 days. We now can calculate ratios instead of absolute values. The recent ratio of news items of an author liked by user a, to the total number of the author's news items is:

$$RatioLikesToItemsFrom_a(b) = \frac{|LastRecent(LikesToItemsFrom_a(b))|}{|LastRecent(NewsBy(b))|} \tag{9}$$

We can define the ratio of friends that liked a particular news item:

$$RatioLikesOfFriendsTo_a(c) = \frac{|LikesOfFriendsTo_a(c)|}{|\{b|b \in Subscriptions(a) \wedge c \in NewsFor(b)\}|} \tag{10}$$

The recent ratio of the number of likes to the total number of news items, in respect to a specific news item type, can be defined as:

$$RatioLikesToType_a(type) = \frac{|LastRecent(LikesToType_a(type))|}{|LastRecent(\{c|c \in NewsFor(a) \wedge f_c(y) = type\})|} \tag{11}$$

These ratios can be used to calculate a relevance factor per news item. The news items in the news feed of a user could be sorted according to their relevance factor. To our best knowledge, there is no information concerning the weight of each ratio in Facebook's sorting algorithm.

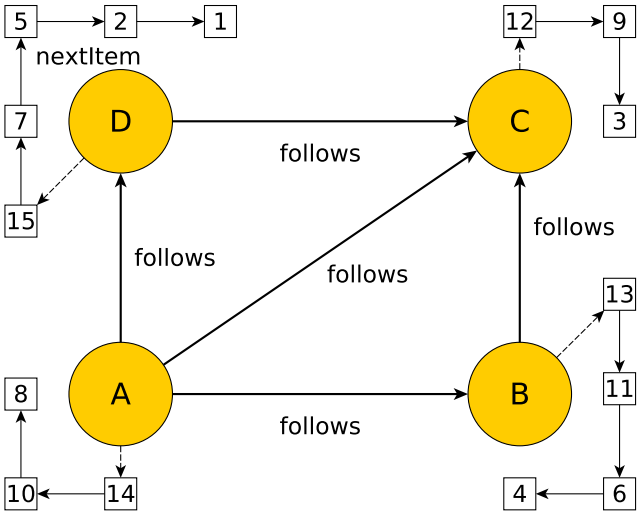


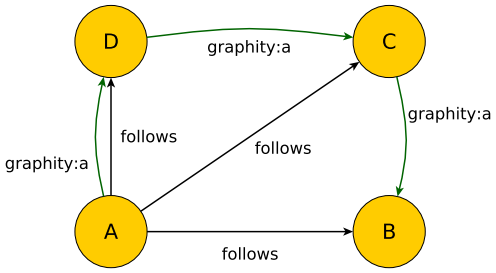
Figure 2: Graph model of STOU.

3.2 Graphity News Feed Retrieval Algorithms

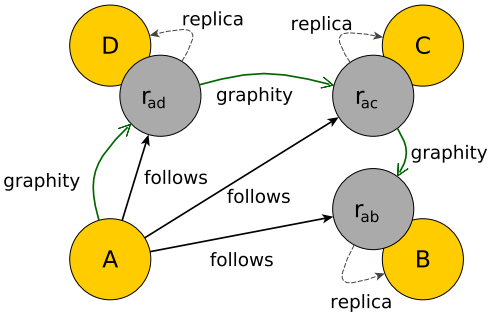
After formalizing the news feed retrieval problem the Graphity paper introduces two graph models for efficient retrieval of the top-k news items. These are STOU and Graphity, where STOU is optimized for write heavy and Graphity for read heavy cases.

3.2.1 STOU

Consider a social network graph as described in subsection 3.1. To retrieve the news feed with length k for a user a a breadth search of depth 2 would be necessary: Visit all users a follows, collect the news items they created and sort the items before reducing the feed length to k news items. Note that all users were visited and all news items per user had to be retrieved. This is necessary while it's impossible to know if a news item will be relevant, before sorting the news items that may be relevant. STOU is a graph model that faces this problem. For each aggregating vertex a the news items in $NewsBy(a)$ get sorted by their creation time stamp. The graph model STOU is illustrated in Figure 2. Starting from the graph model of the graph in Figure 1 little is needed to achieve this sorting. We introduce a new edge label $nextItem$ to link the news items. These edges link from one news item to the news item created subsequently and form a linked list. Formally we link two news items $c_1, c_2 \in c : f_{c_1}(t) > f_{c_2}(t) \wedge \nexists c_3 \in C : f_{c_1}(t) > f_{c_3}(t) > f_{c_2}(t)$. The $published$ edges are now used to link from an aggregating vertex to its most recently created news item. Per aggregating vertex, the first k news items have to be visited using STOU, in order to retrieve the top-k news items. Any item at position $k + 1$ would be cut off by the filter anyway.



(a) Basic Graphity graph model with the Graphity index of user a. Users are linked with an edge type specific to user a.



(b) Replica layer of user a, it contains a copy of every user that a follows - replicated for user a exclusively.

Figure 3: Graph models of Graphity.

3.2.2 Graphity

The graph model Graphity bases upon STOU and introduces an index structure to further reduce the costs of the news feed retrieval. One problem of STOU is that all aggregating vertices a follows have to be considered. An aggregating vertex is considered even if the result feed doesn't contain any of its news items. In Graphity the aggregating vertices are sorted. For each aggregating vertex a we link all aggregating vertices in $b \in Subscriptions(a)$, sorted by the time stamp of their most recently created news item. This linked list is called the Graphity index of a. Figure 3a shows the Graphity model. We call b_{c1} the most recent news item of b. Formally we link two aggregating vertices $b, g \in Subscriptions(a) : f_{b_{c1}}(t) > f_{g_{c1}}(t) \wedge \nexists l \in Subscriptions(a) : f_{b_{c1}}(t) > f_{l_{c1}}(t) > f_{g_{c1}}(t)$. With Graphity at maximum k aggregating vertices have to be visited, in order to retrieve a news feed with length k.

The initial version of Graphity used an unique edge label for each aggregating vertex to link its subscriptions. In fact only a limited number of edge labels is available in a graph database technology such as Neo4j. If we need one edge label per aggregating vertex, the maximum number of edge label is the upper limit of $|A|$, the number of aggregating vertices, as well. Since Graphity's initial release a replica layer was added to the graph model to overcome this limitation. The replica layer is illustrated in Figure 3b. For each aggregating vertex a we create a replication vertex $r_{a,b}$ for each aggregating vertex $b \in Subscriptions(a)$. These replication vertices form the replica layer of a. We introduce an edge label $graphity$, that substitutes the per-user edge labels for the Graphity index. Edges with this label link the replica layer of an aggregating vertex a like the aggregating vertices were linked in the original Graphity model. This of course increases the storage complexity of Graphity but doesn't increase its runtime complexity. For example, the number of edges per user increases by 100% and the number of vertices by $d_{out} \cdot A$. However, on the long term the number of user vertices is less than the number of news items.

Algorithm	Retrieve	Post	Subscribe	Unsubscribe
STOU	$\mathcal{O}(d_{out} \cdot log(d_{out}))$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Graphity	$\mathcal{O}(k \cdot log(k))$	$\mathcal{O}(d_{in})$	$\mathcal{O}(d_{in})$	$\mathcal{O}(1)$

Table 3: Theoretical runtime complexity of the news feed retrieval algorithms.

Thus the storage complexity class is not increased.

3.2.3 Runtime Complexity Comparison

After STOU and Graphity were described we now compare the runtime complexities of the two models according to their operation types:

- retrieve news feed
- update subscription state
- create news item

The deletion of news items is not considered, due to the low ratio of such requests reported in an existing social network platform [17].

The first operation is to retrieve the news feed of an aggregating vertex a with length k. In STOU we have to consider the news items of the d_{out} aggregating vertices that a follows. Each aggregating vertex in $Subscriptions(a)$ has its news items sorted, represented by the linked list the $nextItem$ edges form. To retrieve the news feed we have to merge the d_{out} sorted lists of the users in $Subscriptions(a)$. Dominant in this merge sort is to add the aggregating vertices to a priority queue. The runtime complexity of this operation is $d_{out} \cdot log(d_{out})$. In Graphity the aggregating vertices are sorted. We have to consider the news items of at most k aggregating vertices. The sorted news item lists of the k aggregating vertices have to be merged in order to retrieve a news feed with length k. Compared to STOU the runtime complexity improves to $k \cdot log(k)$, due to the sorted aggregating vertices. For this reason Graphity is considered read optimized. Table 3 gives an overview of the runtime complexities of STOU and Graphity.

The second operation is to update the subscription state of an aggregating vertex a. This includes to either add or remove an aggregating vertex b to or from $Subscriptions(a)$. In STOU we have $follows$ edges that indicate a subscription. To add a subscription for a we have to add a $follows$ edge from a to b. To remove a subscription of a we have to remove the $follows$ edge from a to b. Both operations have a runtime complexity of $\mathcal{O}(1)$. In addition to the STOU model Graphity has the Graphity index, stored in the replica layer. To add a subscription for a we have to add a $follows$ edge from a to b and create a replica vertex $r_{a,b}$ of b. The replica's position in the Graphity index of a has to be found, similar to an insertion sort. Since the replica layer of a was sorted before adding b the runtime complexity of the insertion sort is $\mathcal{O}(n)$ in the linked list. Thus we can add subscriptions in $\mathcal{O}(d_{out})$.

To remove a subscription of a we have to remove the $follows$ edge from a to b and remove the replica vertex $r_{a,b}$. Starting from the $follows$ edge that is to be removed, we can access $r_{a,b}$ in $\mathcal{O}(1)$. The removal includes an update of the Graphity index. Graph edges are doubly linked lists, therefore it's sufficient to link the predecessor with the successor of $r_{a,b}$. The runtime complexity of the subscription removal is $\mathcal{O}(1)$.

The third operation is to create a news item for an aggregating vertex a, to add it to $NewsBy(a)$. In STOU we have $nextItem$ edges that link the news items of an aggregating vertex. To create a news item we have to create a vertex for the news item and insert it at the top of the list. This has a runtime complexity of $\mathcal{O}(1)$. For this reason STOU is considered write optimized. In Graphity additional effort is necessary. The aggregating vertex a has to be moved to the first position of the Graphity index of each follower. This index update has a runtime complexity of $\mathcal{O}(1)$, but the update has to be repeated for the d_{in} aggregating vertices in $Followers(a)$. Thus the create news item operation has a runtime complexity of $\mathcal{O}(d_{in})$ in Graphity.

4 Distribute STOU and Graphity

In order to evaluate the ability of STOU and Graphity to scale out, we had to distribute the news feed service. According to the CAP theorem, a distributed system can have at most two of three properties, that are all desirable [4]: 1. data consistency (C), 2. high data availability (A) and 3. tolerance of network partitions (P). Data consistency means to have a single up-to-date copy of the data. Highly available services can always respond to a request. Actually a request doesn't need to success. Partition tolerance implies to be able to continue operation though some nodes loose connection due to network and disk failures, natural disasters, cooling or power problems. As an example, consider two nodes sharing data. Allowing at least one node to update the data, leads to data inconsistency. If data consistency has to be preserved, the other node has to act as unavailable. The only way to update both nodes is to communicate and thus forfeiting P. In fact, it is not a two-of-three choice, but it is a matter of degree. Each of the three CAP properties is more continuous than binary. However, the general belief is that you can't forfeit P. Distributed systems dealing in an infrastructure with hundreds or thousands of components need to treat (network) failures as the norm, rather than the exception [10], [15]. The modern CAP goal is to maximize the combinations of data consistency and high availability according to the application requirements and NoSQL systems typically choose availability over data consistency [4]. A common approach to visualize the CAP theorem is the CAP triangle shaped by points representing C, A and P. Systems then can be represented in this triangle.

In this section we describe two common strategies to distribute a database and two technologies that each implement one of these strategies. We class the distribution technologies with the CAP triangle. It is important to classify at technology level, because most distributed storage systems implement own approaches to over-

come weaknesses of the distribution strategy they base upon. In fact configuration options further allow adjustments of the system's position in the CAP triangle. Thus a system can be represented as a region of possible configurations.

4.1 Distribution Strategies

There exist several strategies to distribute a storage system. We present two common and basic distribution strategies for storage systems: a) Database replication and b) distributed hash-tables. However, there are several variants of these strategies, that try to overcome some weaknesses of the basic approaches. In this work we focus on the basic approaches and variants that we use in our implementations presented in subsection 4.2.

4.1.1 Database Replication

One approach to distribute a storage system is to replicate the entire database. Typically we separate eager replication and lazy replication that both improve availability and performance [12]. Eager replication keeps all cluster nodes completely synchronized and each update is a single atomic transaction involving all nodes. Typically a locking scheme is used to detect concurrent updates that are converted to waits or deadlocks. For this reason there are no data inconsistencies and transactions are ACID. However, this leads to a problem called "scaleup pitfall" [12]: Each write transaction involves all nodes and both response times and deadlocks increase with the cluster size. This offers high availability and data consistency but lacks of partition tolerance. In contrast lazy replication propagate updates to the cluster asynchronously. Multiple transactions that access the same data objects lead to multiple object versions that have to be reconciled e.g. at application level. Of course, there are more techniques to replicate a service. These techniques often combine eager and lazy replication ideas to overcome weaknesses of the simple replication approaches. As an example there are master copy schemes that reduce the deadlock and latency problems of eager replication, without sacrificing the ACID transactions. A single master node owns the data and is the authoritative data store. Only the master node is allowed to write data, slave nodes have read-only access. If a write request reaches a slave node, it is delegated to the master.

4.1.2 Distributed Hash-Tables

Distributed hash-tables distribute data blocks over multiple servers. The distribution achieves load balancing and block replication improves availability, fault tolerance and performance [9]. In contrast to database replication, distributed hash-tables support an increasing amount of data stored in the cluster. They can store files that are larger than a single server's capacity and support heterogeneous hardware configurations via virtual nodes [9]. However, data distribution increases the number of intra-cluster network requests for writes and reads, since each block must

be looked up separately. Routing tables actually can reduce this problem to a single hop [15]. In a system based upon distributed hash-tables, nodes and blocks have keys where node keys form a ring. Each node is responsible for a certain number of keys, depending on the total number of blocks and the cluster size. For a block b it is easy to look up the responsible node: It is the first node with a key greater than the key of block b. This successor node is responsible for the block and its replicas. The replicas are placed on nodes in the ring immediately after the successor. Client requests get routed to the predecessor, that returns the successor and the replica nodes. However, these details may vary between implementations. Clients specify consistency levels for their write and read requests. The consistency levels refers to the minimum number of nodes in the cluster, that responded to the request. Only nodes with a copy of the requested block can respond to the request. Thus the consistency level needs to be less than the replication factor, the number of copies per block, that can be configured as well. Possible consistency levels are QUORUM or an absolute number. QUORUM refers to the majority ($n/2+1$) of all nodes that hold a replica of the requested block.

4.2 Distribution Technologies and Implementation

In order to distribute STOU and Graphity, there were three aspects to concern:

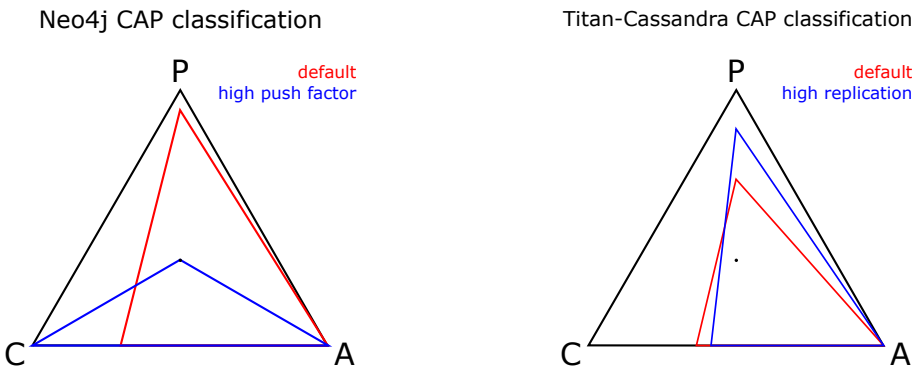
- get access to a distributed graph database
- implement STOU and Graphity with respect to distributed execution
- provide an endpoint for benchmark client requests

We used existing technologies to get access to a distributed graph database. The first technology is Neo4j, the graph database used in the initial evaluation of STOU and Graphity. The second technology is a composition of Titan and Apache Cassandra. While Neo4j implements a database replication approach to distribute the database, Cassandra is an eventually-consistent database based upon distributed hash-tables. Both distributed databases provide endpoints for plugins. Thus we implemented STOU and Graphity as plugins for the two technologies. We present the two technologies and key aspects of our distributed algorithm implementations.

4.2.1 Neo4j

Neo4j[8] is a transactional open source graph database written in Java. Neo4j offers ACID transactions. The Enterprise version of Neo4j supports a high availability (HA) mode. In HA mode Neo4j replicates the whole data on each node. Thus the data is limited to the capacity of the node with the lowest storage capacity. The distribution strategy can be seen as delayed eager replication with a master copy scheme: A single node is elected as master node and becomes authoritative. Slave

[8] `http://neo4j.com/`



(a) Neo4j (b) Titan-Cassandra

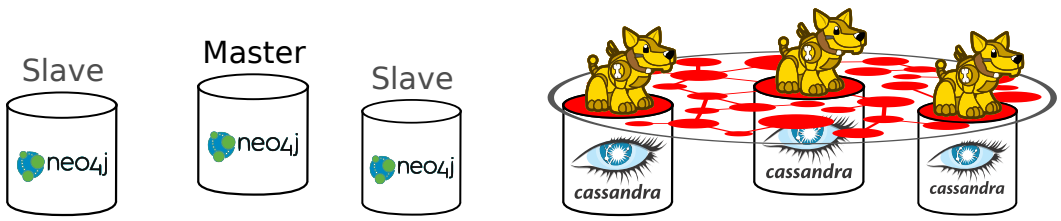
Figure 4: Classification of Neo4j and Titan-Cassandra configurations in the CAP triangle.

nodes poll updates in a fixed interval. If write requests reach a slave node, it synchronizes the graph elements involved and locks them through the master node. The write transaction is executed on the master and applied to the slave on success. Write requests that reach the master directly can be pushed to a number of slaves on success, without the need for remote locking.

Each Neo4j node is capable of accepting read and write requests. The master holds the up-to-date data but slaves poll updates in a configurable interval, that defaults to 10s. If the master goes down or is unreachable, e.g. due to network partitions, a new master is elected automatically. Thus Neo4j continues to handle read and write requests after some delay for the election. However, this leads to multiple data branches that have to be merged later. Thus Neo4j is highly available, provides a strong network partition tolerance and a weak level of data consistency by default. Neo4j is classified with the CAP triangle in Figure 4a. The default configuration of Neo4j is labelled *default*. Neo4j can be configured to push write transactions to a number of slaves, for example to all slaves. This leads to high availability and high data consistency but scarifies the network partition tolerance. In Figure 4a this configuration is named *high push factor*.

We use Neo4j Enterprise in version 2.1.2. Each cluster node runs one Neo4j instance. The Neo4j server accepts clients requests and executes the application logic in a server plugin that we implemented. The plugin can be queried by its REST endpoints. One node is configured as master node, while the other nodes are configured to work as slave. The cluster node setup is shown in Figure 5a. The plugin code and the configuration files for Neo4j are available on Github.[9] A central concept in Neo4j is locking, which we had to consider in our plugin code. Write requests automatically lock graph elements they write to. Concurrent write requests won't be able to write to locked graph elements and have to wait for the locking requests

[9] `http://github.com/sebschlicht/bachelor-thesis`



(a) Neo4j (b) Titan-Cassandra

Figure 5: Setup of Neo4j and Titan-Cassandra clusters with $n = 3$ nodes.

to finish. The automatic locking locks graph elements chronologically by their access. This can lead to deadlocks in the algorithm implementations. For example, if two users try to subscribe to each other simultaneously: One request would try to create a subscription edge from user a to user b in a transaction t_1. Since this edge starts from user a, this user would be locked at first. The second request would try to create a subscription edge from user b to user a in a transaction t_2. In contrast to the first transaction, t_2 would lock user b at first. In the next step t_1 wants to lock the edge's destination vertex user b. This vertex has already been locked by t_2, thus t_1 waits for t_2 to finish. At the same time t_2 wants to lock user a, the destination vertex of its new subscription edge. User a has already been locked by t_1 and thus t_2 waits for t_1 to finish. This is a deadlock, as both transactions would wait for each other infinitely. Instead of a chronological locking, we lock user vertices descending by their unique identifier. Such consistent locking order prevents transactions from running into deadlocks. However, in practice it wasn't sufficient to use a consistent locking order. We still experienced deadlocks and thus a percentage of requests kept failing.

4.2.2 Titan-Cassandra

Titan[10] is a graph database that supports a range of storage backends, such as Apache HBase, Oracle Berkeley DB and Apache Cassandra. We call Titan Titan-Cassandra, when Cassandra is used as its storage backend. Cassandra[11] is a distributed eventually-consistent storage system basing upon distributed hash-tables [15]. With no single point of failure, Cassandra aims to run on clusters with hundred of nodes and to provide high write throughput without sacrificing read efficiency. In addition to the natural load balancing of distributed hash-tables, Cassandra analyses load information to improve load balancing. Cassandra serves multiple services in Facebook, such as the Inbox Search. Rexster[12] is a graph server to make a graph accessible via REST or a Rexster-specific binary protocol.

Each Titan-Cassandra node accepts read and write requests. By default Titan

[10] https://github.com/thinkaurelius/titan/wiki
[11] http://cassandra.apache.org/
[12] https://github.com/tinkerpop/rexster/wiki

queries Cassandra with a write consistency level of 1 and a read consistency level of QUORUM. Thus every block is replicated on one node only. Read requests that access a block must be answered by the node with the exclusive copy of the block. Data won't be available if the network partitions, but the nodes will continue to operate and to execute write requests. Nevertheless, Cassandra is eventually-consistent and can't provide a high consistency level, no matter how few or many replicas of a block exist. With its default configuration Titan-Cassandra provides high availability, a basic network partition tolerance and a weak level of data consistency. Titan-Cassandra is classified with the CAP triangle in Figure 4b. The default Titan-Cassandra configuration is labelled *default*. Titan-Cassandra is able to successfully execute read requests if the network partitions as well, if a higher block replication factor is used. Titan-Cassandra stays network partition tolerant for writes, as long as the number of nodes in the resulting clusters is higher than the replication factor. However, an increasing replication factor slightly decreases the data consistency, as there are more replicas of a block that can be out-of-sync. In Figure 4b this configuration is referred as *high replication*.

We use Titan 0.5.3, embedded in Rexster 2.5.0, using Cassandra 2.0.2 as its storage backend. Each node in the cluster runs all the three tools simultaneously. Rexster accepts client requests and queries the embedded Titan instance. Titan uses the Cassandra cluster to store the graph. Titan thus acts as a layer between the Rexster service cluster and the Cassandra storage cluster. This setup is show in Figure 5b. Note that we used OpenJDK 1.7 instead of Oracle Java. Oracle Java is reported to increase the performance of Cassandra significantly. The application logic is executed in a Rexster extension that provides REST endpoints. The extension code and the configuration files for Rexster, Titan and Cassandra are available on Github as well. A central concept in the Rexster extension using Titan-Cassandra, is that transactions are non-atomic and lock free on the graph API level. If concurrent writes involve the same vertices, both writes can succeed. This leads to violations of the graph schemas that we would expect for STOU or Graphity. The solution is to store the request time stamp in the edges created by the corresponding request. Schema violations still can occur and we may have multiple $graphity$ or $nextItem$ edges leading out from or in a certain vertex. However, we can ignore all the edges except of the edge with the most recent time stamp. Edges that we ignore can be removed on-read to improve the runtime of future read and write requests. This is working properly in STOU. A $follows$ edge is either existing or not. News items are sorted in a list and new items are added exclusively to the head of this list. Graphity orders the users in a user list. In contrast to the news item list, we also move users from any position of the user list to its head, whenever a user posts a news item. Graphity requires the ordered user list, in order to work properly. An eventually-consistent storage backend can't ensure a consistent state, where the user list is ordered at any moment. Due to write operations, the user list regularly enters an unordered state. It is possible to detect this state, but the state prohibits concurrent writes. Until the changes are made persistent in Cassandra, read requests will return incorrect results

as well, but this would be acceptable. A possible solution is to rebuild the Graphity index in each write request, that modifies the Graphity index: For all subscribers of a news item author and thus for every single write request. The runtime complexity of all write request types would change to $\mathcal{O}(d_{out} \cdot d_{out})$. In the end, we weren't able to fully implement Graphity on the eventually-consistent Cassandra, at least with the runtime complexities listed in Table 3.

5 Hypothesis

According to the runtime complexities we expect STOU to support a higher throughput of write requests than Graphity with the same conditions. In contrast we expect Graphity to handle more read requests than STOU. A typical value for the news feed length k is 15, since most users are interested in the latest news items only [17]. The mean node degree of users in social networks like Facebook or Twitter is of magnitude 10^2 [16], [18]. On a graph with this properties Graphity can be expected to speed up the core retrieval process roughly by a magnitude. However, runtime complexity are not sufficient to predict the results of the evaluation, due to the overall setup. Both algorithms will run distributed, in a graph database on a cluster with multiple machines. Requests will be executed by one or more clients on different machines. We have to consider network latency from the client to the cluster and there may be intra-cluster network requests in addition. The graph database will be wrapped in a web server technology, in order to accept client requests. The performance of the algorithm in the graph database depends on multiple factor as well. In the theoretical runtime analysis we always started from the vertices and edges, where we would execute the one or the other retrieval algorithm. In fact we need to look up the corresponding edge or vertex at first. Usually this implies an index lookup, which can imply additional network requests if an external index backend is used. The runtime complexity of the retrieval algorithm might have a minor effect on the evaluation results.

On Titan-Cassandra, with its distributed hash-tables, graph elements will be balanced across the entire cluster. STOU involves all the d_{out} user vertices in a read request. This is a result of its runtime complexity ($\mathcal{O}(d_{out} \cdot log(d_{out}))$). In contrast, Graphity involves fewer graph elements, only the first k user vertices have to be considered. Analogous to STOU, this is a result of the runtime complexity of Graphity ($\mathcal{O}(k \cdot log(k))$). Due to intra-cluster communication, read requests will be slow with both retrieval algorithms, but the read throughput will increase linearly with the cluster size. Since every graph element has to be looked up in the Cassandra cluster and leads to network requests, Graphity will show a better read *scale-out* than STOU. It will provide significantly higher read throughput and lower request latencies. However, Graphity won't provide 10 times higher read throughput, as the runtime complexity suggests. Index lookups and intra-cluster communication will cause the major part of the request latencies and reduce the effect of the runtime complexity. On Titan-Cassandra we expect both algorithms to support an increas-

ing number of write requests when scaling out. STOU will scale out write requests better than Graphity. Graphity needs to update and retrieve more graph elements to execute a write request. Thus we expect STOU to support a higher write throughput, though write requests are just acknowledged in Titan-Cassandra.

We expect both STOU and Graphity to support an increasing number of read requests when scaling out the Neo4j cluster. Graphity will support a higher read throughput than STOU. This is a result of its runtime complexity. Due to the distribution strategy implemented in Neo4j, we expect neither STOU nor Graphity to support an increasing number of write requests when scaling out.

6 Evaluation

We ran a series of benchmarks to evaluate the *scale-out* of STOU and Graphity on Neo4j and Titan. Our evaluation strategy based upon the *scale-out* evaluation strategy described in subsection 2.4. However, we couldn't follow all the suggestions. This section describes the actual strategy and points out major differences. In every benchmark we measured the total throughput of the cluster and the average request latency. We used a clean database for each benchmark. From benchmark to benchmark we increased the cluster size n. In this case n refers to the number of nodes, where each node is a service node and a storage node simultaneously. Starting from a single node we added one node per benchmark, until the cluster size reached eight nodes. However, Neo4j doesn't support a cluster with two nodes, thus data points for $n = 2$ are unavailable. In every benchmark the number of requests was kept proportionally with n. For this reason the total data size increased proportionally as well. In contrast to the suggestions in subsection 2.4 we didn't try to maximize the offered throughput for each cluster size and database technology. We ran a large number of benchmarks in limited time and thus had to enter into compromises. Instead of maximizing the throughput, the request rate increases proportionally with n. We compared the throughput of STOU and Graphity per graph database. A system with a good *scale-out* should offer proportionally increasing throughput. At the same time the request latencies should remain constant and should always be acceptable. Read and write requests were strictly separated in the evaluation. Subsection 6.1 describes the experimental setup including the data set used for the request generation and the hardware configurations, in order to make the results reproducible. The results of the read request evaluation are presented in subsection 6.2, the results of the write request evaluation in subsection 6.3.

6.1 Experimental Setup

A single benchmark client issued requests to the cluster with an asynchronous HTTP client library. The load was balanced across the cluster using round-robin scheduling. For the initial Graphity evaluation the authors used a dump of the german Wikipedia as data set. This dump is a stream with all changes from its emergence in

2004 until the 13th of August in 2011. Each change can be translated into a request in a social network. However, some requests depend on others and we would run into problems when firing request concurrently. Instead we synthetically generate requests to reach a node degree similar to today's social network platforms. Table 4 illustrates the write request distribution that we used. Table 5 shows the statistics of the resulting graphs used in the read benchmarks. The data size of the Neo4j and Titan graphs roughly equalled, thus only a single value is provided. Write benchmarks result in graphs an order of magnitude smaller. The user node degree equals the user node degree of Facebook in 2011 and is 10% too low, compared to Facebook and Twitter today [18], [16]. As stated in the introduction, a social network graph with all its meta data won't fit into memory. However, if your cluster is large enough and you can use the memory of each cluster node, you can fit the graph into memory at some point. This is more easy to achieve, if the network structure and the meta data are separated, e.g. if you store the meta data in an external service. In our benchmarks we stored the network structure and the meta data in the graph, as our graphs were small enough to fit into memory.

The evaluation took place in the University of Koblenz cloud, with a total of 168 cores and 1.75 TB memory on 14 compute nodes. Please note that the storage in the cloud is virtualized via ceph. Every machine we used was a virtual machine in this cloud. Table 6 shows the machine configurations. The benchmark client had enough resources to handle the increasing request rate. In fact the CPU utilization did not reach more than 70%. Cluster nodes had four cores and 8 GB memory, enough to run both the OS and the graph database in use. All virtual machines, the client and each Neo4j and Titan node, were in the same virtual network. The round-trip time between any pair of virtual machines was less than a millisecond.

6.2 Read Requests

The evaluation client fires 100,000 read requests per node, with at maximum 50 requests per node simultaneously. When the client generates a read request, it picks

Request Type	Ratio
Create user	0.5%
Post news item	40.0%
Subscribe	51.5%
Unsubscribe	8.0%

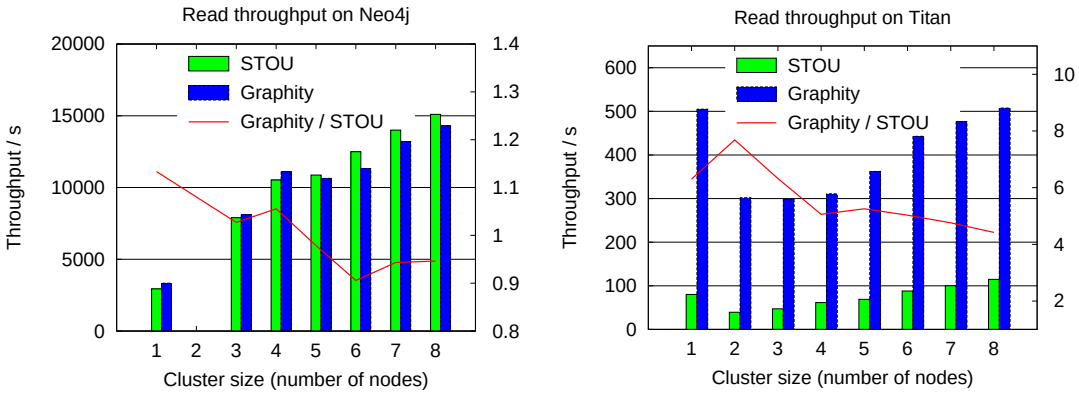
Table 4: Request distribution used for write benchmarks in the *scale-out* evaluation.

#	$\|A\|$	$\|C\|$	$\overline{d_{out}}$	$E_{A>k}$	Data size
1m	5.2k	1.6m	84.2	95.4%	0.4 GB
3m	15.0k	4.8m	87.3	95.3%	1.3 GB
4m	19.9k	6.4m	87.5	95.5%	1.7 GB
8m	39.9k	12.7m	87.3	95.6%	3.4 GB

Table 5: Statistics of the graphs resulting from a number of generated requests that were used in read benchmarks. The table shows the number of users and news items, the average node degree, the ratio of users with more than $k = 15$ items in their news feed and the total data size.

Machine	CPU	RAM
Client	8 cores @ 2.9 GHz	16 GB
Cluster node	4 cores @ 2.9 GHz	8 GB

Table 6: Virtual machine configuration.

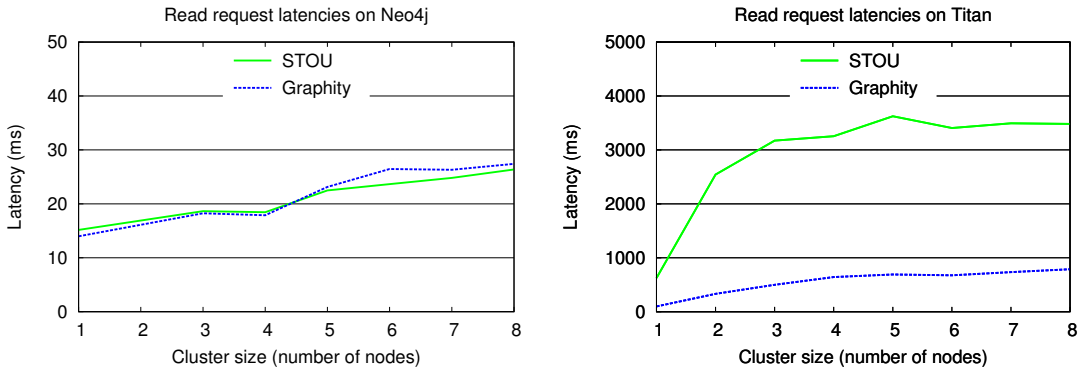


(a) Neo4j (b) Titan

Figure 6: Read throughput in requests per second using STOU and Graphity, together with the throughput ratio between Graphity and STOU, separated for Neo4j and Titan with caching enabled.

a random existing user and retrieves its news feed. The maximum length of the news feed is $k = 15$, since users typically are interested in the most recent news items only [17]. However, chances are that the news feed of the user doesn't contain any news items. As shown in Table 5 this chance is very low. The server responds with a JSON object containing the news feed. A news item contains 140 characters, thus we can expect the response data size to be less than 2 kB. In contrast to write requests read requests require existing data in the database. Therefore we bulk load 1,000,000 write requests per node prior to each read benchmark. All caches are enabled.

Figure 6a presents the read throughput achieved on Neo4j. STOU processes about 3k read requests per second on a single Neo4j node. The read throughput increases almost linearly with the cluster size for $n \geq 3$. An additional node increases the throughput by about 1,500 requests per second. On 8 Neo4j nodes, STOU processes 15k read requests per second. However, from $n = 4$ to $n = 5$ the increase is slightly. The average request latencies are shown in Figure 7a. The latencies of STOU increase at most linearly. Graphity processes about 3.3k read requests per second on a single Neo4j node. This is 15% more than STOU can process on a single node. The read throughput increases with the cluster size, except from $n = 3$ to $n = 4$. We consider the data point $n = 4$ to be a measurement inaccuracy. The throughput ratio of Graphity compared to STOU is shown by the red line. With an increasing cluster size the throughput ratio decreases. For $n \geq 5$ the throughput

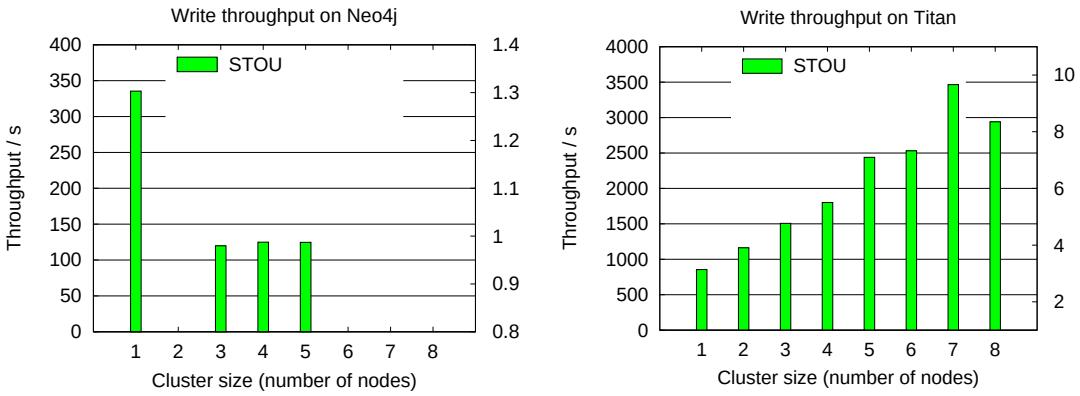


(a) Neo4j (b) Titan

Figure 7: Read request latencies of STOU and Graphity on Neo4j or Titan.

of Graphity is lower than the throughput of STOU. A Neo4j cluster with 8 nodes running Graphity provides a throughput of about 14.3k read requests per second. This is 5% less than STOU processes on 8 Neo4j nodes. However, the throughput ratio doesn't tend to decrease further. The average read request latencies of Graphity increase at most linearly and roughly equal the latencies of STOU.

The read throughput of STOU and Graphity on Titan is presented in Figure 6b. STOU processes about 80 read requests per second on a single Titan node. From $n = 1$ to $n = 2$ the read throughput of STOU halfs to 40 requests per second. Network requests become necessary in order to respond to read requests. The read throughput increases with the cluster size for $n \geq 2$ by about 10 requests per second with each additional node. STOU processes 115 read requests per second on 8 Titan nodes. Figure 7b shows the average read request latencies on Titan. The latencies quickly increase, but the increase is at most linear with the cluster size. For $n \geq 6$ the read latencies roughly keep constant. On a single Titan node, Graphity can process about 500 read requests per second. This is more than 600% of the read throughput that STOU achieves on a single Titan node. Analogous to STOU the read throughput drops to 300 requests per second when switching to two cluster nodes. For $n \geq 2$ the read throughput increases linearly by 34 requests per seconds with each additional node. The throughput ratio of Graphity compared to STOU is shown by the red line. With an increasing cluster size the throughput ratio decreases, except for $n = 2$. The throughput of Graphity doesn't drop below the throughput of STOU for the cluster sizes that we used. However, the throughput ratio tends to decrease further. A Titan cluster with 8 nodes processes about 500 read requests per second. This is 440% of the read throughput of STOU on the same Titan cluster. The read request latencies increase at most linearly with the cluster size. On Titan the request latencies of Graphity are more than 70% lower than the latencies of STOU.



(a) Neo4j

(b) Titan

Figure 8: Write throughput in requests per second using STOU, separated for Neo4j and Titan.

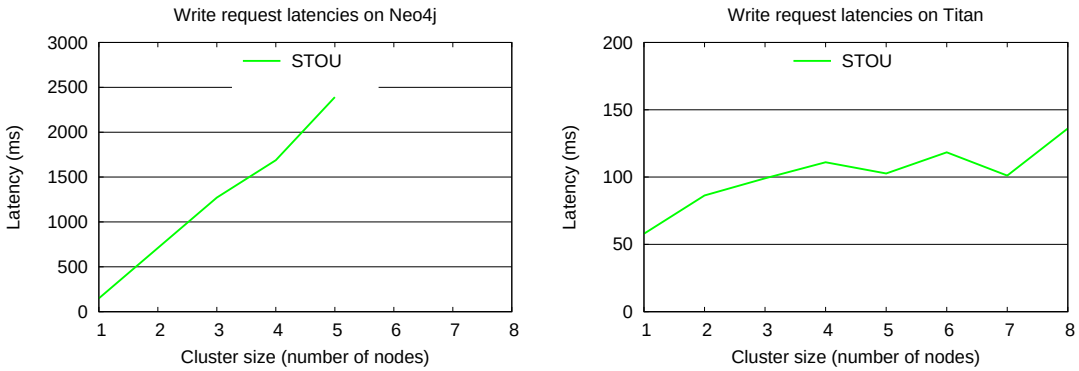
6.3 Write Requests

Our evaluation client issues 100,000 write requests per node, with at maximum $r = 50$ requests per node simultaneously. There are three write request types, as described in subsubsection 3.2.3:

1. post
2. subscribe
3. unsubscribe

Whenever the client generates a new write request, it selects the request type according to the request distribution shown in Table 4. At first we pick a random user, that is the target of the write operation. If the client posts a news item, we need to generate content. We generate a sequence of 140 random characters, as many as the maximum length of a Twitter feed. If the client adds a subscription, we need a user to subscribe to. We pick this user according to the subscription distribution of the data set. The subscription distribution follows a power-law. If the client removes a subscription, it always removes the least recent subscription of the target user. The requests are UTF-8 encoded and wrapped in JSON. We can expect a data size of less than 200 B per request. Please note that the write requests are non-atomic. Each change in a user's subscription state will trigger the creation of a new news item for this user. These news items are generated by the server automatically and don't require additional requests. Thus each subscribe and each unsubscribe request will create a news item for each of the two users involved. In addition we add users lazily. Whenever a user is involved in a news item creation or subscription and is not yet existing in the database, it will be created automatically.

Figure 8a illustrates the write throughput of STOU on Neo4j. STOU processes about 330 write requests per second on a single Neo4j node. The write throughput



(a) Neo4j (b) Titan

Figure 9: Write request latencies of STOU and Graphity on Neo4j or Titan.

doesn't increase with the cluster size, instead it drops if multiple nodes process write requests. STOU processes about 120 write requests per second on three Neo4j nodes. The write throughput stays roughly constant when further increasing the cluster size. Figure 9a presents the average latencies of write requests in STOU on Neo4j, for each cluster size n. The request latencies increase proportionally with the cluster size. This resulted in an increasing benchmark duration and thus the Neo4j cluster wasn't further enlarged. For $n \geq 3$ the average write request latency is higher than a second.

The write throughput of STOU on Titan is presented in Figure 8b. STOU processes 850 write requests per second on a single Titan node. The write throughput increases roughly linearly with the cluster size. However, the throughput increase from $n = 5$ to $n = 6$ is slightly. From $n = 7$ to $n = 8$ there is a throughput decrease. We consider the data point at $n = 7$ a measurement inaccuracy. STOU processes about 3k write requests per second on 8 Titan nodes. The average write request latencies are shown in Figure 9b. The latencies increase disproportionately low, at most linearly with n.

7 Conclusions

On Neo4j we have seen that STOU and Graphity provide similar read *scale-out*. The read throughput and the average read request latencies of both algorithms roughly equal. For clusters with size $n \geq 5$ STOU provides a higher read throughput and lower request latencies than Graphity. On smaller clusters it's the other way round. Both effects may have been caused by the cloud utilization and measurement inaccuracies. We weren't able to examine the write throughput of Graphity on Neo4j, but STOU can be expected to show better write throughput and *scale-out*, as described in section 5. We conclude that STOU is the better algorithm to power a distributed news feed system using Neo4j, that offers limited write throughput.

On Titan Graphity shows better read *scale-out* than STOU. In contrast STOU shows a more linear increase of throughput when scaling out. Thus the throughput ratio between Graphity and STOU is decreasing with the cluster size. If the ratio decrease is linear, STOU and Graphity would provide equal read throughput on a cluster with a size of about 20. However, it is uncertain if the ratio decreases further. We weren't able to examine the write throughput of Graphity on Titan, but the throughput can be expected to equal STOU's write throughput, as described in section 5. However, graph schema violations have to be repaired on read, an effect that we didn't measure. Schema violations will happen significantly more often in Graphity than in STOU. In particular, because concurrent news posts of different users lead to schema violations in Graphity. We conclude that Graphity shows better read *scale-out* on Titan, while Graphity can't be implemented on an eventually-consistent database with the runtime complexities described in subsubsection 3.2.3. STOU is the better algorithm to power a distributed news feed system using Titan, that shows read and write *scale-out*.

A distributed news feed system needs to support high write throughput. On Neo4j we have seen a maximum write throughput of below 350 requests per second. The write throughput of Titan-Cassandra increases with the cluster size. On the other hand, it has to support a read throughput that is even higher. The read performance on Titan is far too low to power a news feed system. We suggest the usage of STOU on Titan to power a large scale news feed system and tune Titan and Cassandra as much as possible, to improve read throughput and latencies. For example, the replication factor could be increased and additional cluster nodes will reduce the latencies as well. If Cassandra doesn't support the desired read throughput, a caching layer can be used. For example a Neo4j server running STOU, which provides a high read throughput. Titan-Cassandra would be the authoritative database that would process write requests in real-time. Neo4j would bulk the write requests, in order to keep the read-only database up-to-date - with an acceptable update delay. This combination of distributed hash-tables and database replication, would support high read throughput and high write throughput.

8 Future Work

At first the evaluation should be repeated on dedicated hardware, in order to remove the side-effects of virtualization. At least one should avoid the usage of virtualized storage. For example, read latencies were much higher than reported [8] and the cloud might be the cause of this behaviour. Though Titan offered increasing write throughput, it is uncertain if we reached the limit. It would be better to maximize the throughput per benchmark, as suggested in [8]. It would be interesting to evaluate the algorithms with workloads, where write and read requests occur simultaneously. For example, requests may fail, be incorrect or the simultaneous occurrence of writes and reads may affect the system performance. In an eventually-consistent database there is a delay between write requests and the vis-

ibility of their changes in read requests, that should be measured. Another interesting aspect would be an evaluation on a cluster with 50 up to hundred nodes. However, we couldn't scale out beyond 8 nodes, as our tools weren't suitable for larger clusters. The bootstrap process has to be redesigned as well, in order to support larger clusters. As Neo4j and Titan-Cassandra differ in their service architecture and their caching options, we should implement the two distribution strategies on our own. This would allow more valid statements concerning the strengths and weaknesses of the distribution strategies. However, these next steps are out of the scope of a bachelor thesis and could be concerned in a subsequent master thesis.

References

[1] Lars Backstrom. News Feed FYI: A Window Into News Feed. https://www.facebook.com/business/news/News-Feed-FYI-A-Window-Into-News-Feed, 2013. [Online; last access on 2015/02/18].

[2] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four Degrees of Separation. In *Proceedings of the 4th Annual ACM Web Science Conference*, pages 33–42. ACM, 2012.

[3] André B Bondi. Characteristics of Scalability and Their Impact on Performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.

[4] Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer*, 45(2):23–29, 2012.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[6] Josh Constine. Facebook Explains The Four Ways It Sorts The News Feed And Insists Average Page Reach Didn't Decrease. http://techcrunch.com/2012/11/16/facebook-page-reach/, 2012. [Online; last access on 2015/02/18].

[7] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[8] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[9] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, 2001.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[11] Facebook. How News Feed Works. https://www.facebook.com/help/327131014036297, 2014. [Online; last access on 2015/02/18].

[12] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM, 1996.

[13] Jake Hofman. Social network analysis with Hadoop. `http://jakehofman.com/talks/hadoopworld_20091002.pdf`, 2009. [Online; last access on 2015/04/28].

[14] Prasad Jogalekar and Murray Woodside. Evaluating the scalability of distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 11(6): 589–603, 2000.

[15] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[16] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. Information Network or Social Network? The Structure of the Twitter Follow Graph. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, pages 493–498. International World Wide Web Conferences Steering Committee, 2014.

[17] Rene Pickhardt, Thomas Gottron, Ansgar Scherp, Steffen Staab, and Jonas Kunze. Efficient Graph Models for Retrieving Top-k News Feeds from Ego Networks. In *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, pages 123–133. IEEE, 2012.

[18] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The Anatomy of the Facebook Social Graph. *arXiv preprint arXiv:1111.4503*, 2011.