Evaluation of NoSQL databases for large-scale decentralized microblogging

Alexandre Fonseca, Anh Thu Vu, Peter Grman
Universitat Politècnica de Catalunya

***Abstract*—Modern distributed systems have evolved to scales that would have seemed unfeasible a decade ago. Whereas in the past, creating and hosting new content in the Internet required considerable effort, nowadays, blogs, videos, pictures and thousands of other data can be created with single clicks of a button, amounting to several Terabytes of new data per day. While supporting services have managed to atennuate this new load by deploying caching and content distribution components, the central data storage of most of those systems has not experienced significant change and continues to rely on a centralized, monolithic and scaled-up core of database nodes. This centralized storage, in face of frequent full-blown service outages, is clearly becoming insufficient and new alternatives are needed.**

The NoSQL movement has been gaining, in recent years, considerable traction. Defending the use of simple database systems with a focus on distribution, sharding and replication, these databases are built for horizontal scaling and the handling of massive amounts of data. In this paper, we'll examine and evaluate the applicability of 2 of the most popular NoSQL data stores - Cassandra and Couchbase - in the context of a microblogging system. We evaluate not only the operational latency of these data stores but also such things as their replication and sharding capabilities, handling of churn and resource usage.

***Index Terms*—Decentralized Storage, Microblogging, Scalability, Peer-to-peer, NoSQL.**

I. Introduction

In recent years, the concept of microblogging has gained considerable traction. As opposed to traditional blogging where authors post often lengthy and detailed articles about their interests, activities or even thematic subjects (e.g code development or sports), microblogging focuses on the posting of very small and concise posts. These posts may be comprised of short sentences or individual media objects such as links or videos and their small size motivates users to post more often and in real-time.

The focus on numerous small real-time interactions and the growing userbase of these kinds of systems makes them a natural platform for marketing and public relations campaigns in a trend called social media marketing. In addition, these systems are also becoming important sources of real-time updates and news as people can post information about local events and see them reach their audience and spread throughout the network in a matter of seconds. While this may be an extraordinarily useful asset in crisis situations where traditional news coverage suffers significant delays, the lack of a trusted review of such information may lead to accidental or malicious misinformation and possible catastrophes [24]. Also, analysis of generated data in these microblogging networks gives researchers a valuable insight into social perception, interaction and opinion regarding a wide range of events, products and stories.

By far, the most popular service associated with microblogging is Twitter. Created in 2006, Twitter now has over 500 million registered users and handles more than 340 million "tweets" (i.e microblog posts) per day. Other popular microblogging services include Tumblr and identi.ca. Moreover, Facebook and Google+ status updates may also be seen as a form of microblogging.

The gigantic amount of data generated in these microblogging systems and other big-data systems makes central management a gargantuan task. Systems cannot rely solely on vertical scalability but have to expand horizontally and strive to offer a good quality of service at a global scale. In addition, these systems have to be resilient to failures and malicious attacks which also puts great emphasis on replication and security.

Relational Database Systems (RDS) have long been the main choice for storage in enterprise and data management systems. Providing tables following strict relational schemas, ACID transactions, SQL query language and relationships managed through foreign keys and joins, these systems place great priority in database normalization (removing of redundancy) and in data consistency. Examples of these systems include MySQL, MariaDB, PostgreSQL, Microsoft SQL Server, SQLite, among many others. Combined with caching mechanisms, these systems may indeed achieve very good performance provided that all the data fits in a single server or data center (centralized). If that is not the case, the consistency enforcing mechanisms and the limited replication and partitioning techniques may severely increase database operation latencies.

With the increased relevance of big-data, NoSQL database systems have been growing in popularity. These databases relax consistency models, incentive the denormalization of models, aggregating data not by entity type but by operation (minimizing or eliminating all together joins) and provide flexible schema definitions [22]. These properties allow NoSQL database systems to greatly reduce operation latency (specially with writes and even without caching systems) and to provide increased horizontal scaling capabilities at the expense of increased data redundancy and disk space usage. In fact, most NoSQL systems are able to function in a total decentralized manner, being able to do live partitioning topology changes and replication degree modifications. Some of the most popular NoSQL systems include Cassandra, CouchBase, MongoDB, DynamoDB and Riak.

Global scale microblogging systems like Twitter rely heavily on frequent write operations and very big amounts of data, as such, the use of NoSQL databases appears to be a perfect match for their requirements. Immediate consistency which is the strong point of most RDS is usually not a problem as long as content is eventually consistent (with a tolerance of seconds or even minutes), highly available, and performant.

In this paper we discuss the applicability and evaluate 2 of the most popular NoSQL database systems in the context of a decentralized microblogging system: Cassandra and Couchbase. We examine not only the usual metrics [4] of operational latency, time for convergence and dataset size, but also examine the ease of configuration, replication and sharding capabilities, performance under crashes and other individual features of interest of these database systems.

A decentralized microblogging system might be perceived in at least 2 different ways:

- Authority-based — The system is backed by some type of authority (corporation, government, etc.) which leverages the decentralization to better support scalability and robustness but still is, to all effects, the manager of the network, choosing the resources used by the system.
- Community-based — The system is not backed by a central authority but rather managed by a community of multiple users and leverages the resources provided by those users to function. This is akin to how voluntary computing systems are able to obtain computing and storage resources and is characterized by a high-degree of churn and heterogeneity.

In this paper, due to time constraints and evaluation limitations we focus primarily on the first type of systems. Nevertheless we will, when deemed appropriate, make some remarks about the applicability or limitations of these database systems in such heterogeneous and high-churn environments.

The rest of this paper is organized as follows: in section 2 we go through some related work on this area; in section 3 we give a brief overview of the characteristics of NoSQL database systems and look into the main features of each of the 2 considered NoSQL databases; in section 4 we describe a system model for our evaluation; in section 5 we discuss implementation details and considerations for each database directly related to the microblogging application; in section 6 we describe the experiments that were made and discuss the obtained results; in section 7 we summarize the obtained results and provide some final remarks about the evaluation and the database systems.

II. Related Work

To the best of our knowledge, there are no publicly available studies evaluating the implementation of the basis of a decentralized microblogging system over a cluster of NoSQL database nodes. Back in 2010, Twitter itself was looking into porting their entire tweet infrastructure to Cassandra [27] but we were unable to find the results of their evaluations. Reddit [20] and Digg [23] also implemented part of their systems using Cassandra but once again, no concrete evaluation results were reported.

Netflix published, in November 2011, a comprehensive benchmark of the Cassandra scalability on AWS [5] but their usecase is very different from that of microblogging. They also focus solely on highly performant machines and don't analyze how Cassandra behaves in smaller, less powerful machines and how (or if it even can) be used in voluntary computing environments or in a user-based P2P manner.

Pierre et al. examine in [28] the use of P2P VPNs as a way to form decentralized microblogging networks between groups of friends. Besides considering a localized scale instead of a more global one, their evaluation was also limited to simulations and did not consider such aspects of availability or latency.

Cuckoo [29] attempts to provide a distributed and decentralized microblogging system by organizing clients in a Pastry DHT. Data is propagated through the DHT based on the routing tables and neighbour lists at each node using a cache-and-relay mechanism in conjunction with multicast and Gossip protocols. In case information about a certain user cannot be found in the DHT, dedicated servers from the service provider are used as backup. This is an interesting approach to use in combination with NoSQL-based dedicated servers to deploy a microblogging system in a voluntary/user-based P2P system at a very large scale. However, it lacks concrete implementation and experimental results and some issues regarding security and privacy have yet to be resolved.

Megaphone [26] proposes something similar to Cuckoo but eliminating all together the need for central servers. Based on Scribe and Pastry, publishing paths are determined by constructing a tree with the publisher as the root and data is sent back to each publisher using that same route. If the publisher leaves the network, the root responsibility is taken by one of its neighbours. At the moment it is just a proof of concept so its actual applicability and performance are still unknowns.

Regarding NoSQL databases, several surveys have been made focusing on different aspects of these databases. In [17], Hecht et al. survey some NoSQL systems based on use case analysis, data models, query possibilities, replication, etc. In [3], Cattell et al. also create a thorough survey of the most popular NoSQL and SQL database systems. The problem with surveys is that they quickly become outdated when the surveyed technologies are changing and maturing at a very big pace as is what is happening with NoSQL databases.

III. NoSQL Databases

The main idea behind basic NoSQL databases is to focus solely on the storage of arbitrary values indexed by keys and let the application worry about the business logic and entity relationships. This allows NoSQL databases to be significantly less complex and more flexible than traditional RDS databases at the expense of moving complexity, such as transaction systems and schema management, into the application itself. While this might seem like a step back from RDS systems where this functionality is offered by default, the big alure for NoSQL systems is the performance and scalability benefits that such simplification entails, specially for those operations where transactional consistency is not required or when flexible schemas are actually a requirement.

However, key-value data stores are not a new concept. Starting from the MultiValue database created for the Pick operating system in the early 1960s, several other such data stores were created: MUMPS, DBM (and successors), BerkeleyDB, MetaKit and LotusDomino are some examples. What distinguises today's most popular NoSQL databases from the aforementioned systems is that while the focus is still on key-values storage, additional functionality was added to better support and automatically manage data partitioning and replication for increased scalability and robustness, 2 characteristics that became increasingly important over the years with the expansion of the Internet and large scale distributed systems. In fact, it was in the early 2000s, that the NoSQL movement started gaining some momentum against the well established relational database paradigm with the introduction of db4o, memcached, CouchDB, BigTable. Since 2005, several other NoSQL databases were introduced: Dynamo, MongoDB, Cassandra, Voldemort, Couchbase, Riak and HBase are just some of the more than 150 different NoSQL systems available for use [15].

A. *Cassandra*

Cassandra is a NoSQL solution designed to handle a massive amount of data distributed across a cluster of commodity servers with no single point of failure. Initially developed by Facebook to support their messaging infrastructure, it was released in July 2008 and by March 2009 became an Apache Incubator project. By February 2010, it graduated to a top-level Apache Software Foundation project. At the time of this writing, its latest stable release was 1.2.4 (April 11, 2013) and this is the version we used on our evaluation. Most information for Cassandra was collected from [21] and [14].

Cassandra offers a data model which is reminiscent of an hybrid between key-value and tabular data storage. Data in cassandra is organized into column families (resembling a RDS table) which have rows and columns. A row is indexed and identified by a row key and contains 0 or more columns constituted by a triple (name, value, timestamp). Row columns are completely dynamic and flexible in that different rows of the same column family need not have the same set of columns and columns may be added or removed from a row at any time. Column families are also grouped into keyspaces for easy application data separation. Cassandra's oficial client API is based on CQL queries delivered over the Thrift RPC protocol. CQL offers a syntax similar to that of SQL but without such features as *JOINs*, transactions and limited filtering capability. Conflict resolution is made by taking the value with the latest timestamp. For this reason, Cassandra requires nodes with loosely synchronized clocks.

In Cassandra, all nodes play the same role in the system, distributing and forwarding data across the cluster to/from the owner nodes. Starting with version 1.2, Cassandra nodes are composed of a variable number of virtual nodes. The virtual nodes of all real nodes in the cluster are organized into a ring overlay and are automatically assigned tokens. These tokens determine the data-node assignment according to a consistent hashing mechanism over the first column of the primary keys of each row (known as the partition key). Each virtual node is then assigned the range of rows whose key is between its token and the token of its predecessor in the ring. The hash function can be parametrized by using different *Partitioner* classes. Cassandra ships with 3 such partitioners: MurmurHash-based (default), MD5Hash-based and ordered-key-hash based. Information and state of each node is distributed through the cluster using a Gossip protocol that runs every second and communicates with up to 3 other nodes in the cluster.

Topology-awareness can be given to Cassandra clusters through the use of different *Snitch* classes which group nodes into racks and then into datacenters. Included snitches are the *SimpleSnitch* (no topology info), *RackInferringSnitch* (datacenter and rack assigned by IP address), *PropertyFileSnitch* (manual global topology configuration), *GossipingPropertyFileSnitch* (manual local topology configuration) and the *EC2Snitch* and *EC2MultipleRegionSnitch* which automatically get region and availability zone information from the AWS EC2 metadata API. Topology awareness has an impact not only in the routing of read/write requests by trying to keep client communications inside the same rack or datacenter but also in the choice of replicas by ensuring that data is replicated in multiple datacenters for increased reliability.

Replication strategies are determined per keyspace. The *SimpleStrategy* replicates a virtual node's data on the k (configurable value) successor virtual nodes in the ring. On the other hand *NetworkTopologyStrategy* allows the specification of the replication factor per-datacenter and inside each datacenter attempts to put replicas in different racks.

Consistency levels aren't hardcoded in Cassandra and can be specified on a per-query base. Write requests are routed to all replicas owning the written row and the number of responses the coordinator node waits for before confirming the write to the client is dependant on the write consistency level set. On the one end, *ONE* waits for a single write confirmation while on the other end *ALL* waits for a confirmation from all replicas. In between you have such levels as *TWO*, *THREE*, *QUORUM* or *LOCAL_QUORUM*. If one or more replicas are down at the time of the write, they will be repaired later on using hinted handoffs, read repairs or anti-entropy node repairs. Read requests from a client are directed to one of the database nodes and that node becomes the coordinator of the request. The coordinator contacts the k replicas which are responding faster where k is determined by the consistency level similar to what happens with the write requets. If responses aren't all the same, the value returned to the client is that associated with the newest timestamp. Non-considered replicas with different values are sent read-repair requests so they can update their stored values with the one returned to the user.

Cassandra also supports limited filtering of the rows returned by a query. Supported filtering operations are: $=$, $>$, $>=$, $<$, or $<=$. These operations can only be applied by default to columns over which an index was constructed and the less than or greater than operations cannot be applied to partition keys. To be able to filter non-indexed columns or use the forbidden operations on the partition key, the user must explicitly specify in his query that he allows arbitrary filtering. In this case, the user is accepting that Cassandra cannot provide any performance guarantees as such a query might require searching all stored data in sequence.

B. CouchBase

Couchbase is the result of a merger between Membase and CouchDB in Feb 2011. It inherits properties from both the aforementioned data store systems, i.e.: key-value storage where values can be any arbitrary binary data, a combination of in-memory and disk storage for speed and persistency and indexing and querying by applying incremental MapReduce. It is currently managed by Couchbase Inc. and distributed under two different but frequently integrated and synchronized editions: Enterprise and Community. We used the latest Community Edition stable release at the time of this writing (2.0.1, released on April 9, 2013) in our evaluation. Most information about Couchbase was collected from [7] and [6].

Couchbase typically stores its data as JSON documents. These documents are logically grouped into *buckets*. The *buckets* act as a mechanism to manage resources, replication, persistency and security for the group of documents and not necessarily to seperate different types of documents. In fact, it is recommended to treat each *bucket* similarly to a database in traditional relational database systems and separate documents into different *buckets* based on resources, replication requirements and update frequency rather than the types of documents. There are two types of buckets, namely:

- *Couchbase* (or *membase*) : Data in these buckets is stored in disk and cached in memory. These buckets provides a highly-available, distributed and persistent data storage with replication capabilities to cope with several node failures. For the purposes of our experiments, we will be using this type of buckets in our evaluation.
- *Memcached* : Data in these buckets is stored solely in memory. It is designed to be used alongside a relational database as a distributed caching mechanism to speed up access to frequently used data.

Data in one bucket is divided into *vBuckets*, or subsets of key space of the bucket, replicated and distributed among nodes in a cluster. This technique abstracts the logical partitioning of data from the physical topology of the cluster, which can dynamically change. Clients can choose to communicate with any node in the cluster and let the server hash the *key* value to obtain the corresponding *vBucket* identifier or do it themselves and communicate directly with the node responsible for that *vBucket*. The latter is recommended for better performance. In order to enforce strong consistency within a cluster, Couchbase makes sure that, at any point, there is only one active replica of a *vBucket* and all accesses to data belonging to that *vBucket* are handled by only one node. To improve locality and, thus, latency, Couchbase supports replication between different datacenters or clusters by providing a Cross Datacenter Replication (XDCR) functionality to synchronize data between clusters from time to time.

Querying on the data within one bucket can be achieved by using *views*. A *view* is a list of <*key*, *value*> tuples produced by applying a *map* and an optional *reduce* function on every document in the bucket. Typically, the *map* function will produce one or more tuples with the document's field(s) that we want to index as the *key* and the *reduce* function performs computation to summarize the information on this list of tuples. A common use case for *views* is to index the documents and then apply a query to that index since there is no way to query for documents with keys within a specific range as there is with Cassandra for example. Besides, *views* are also used to simulate the joining of relational database tables or to perform ordering, grouping and summarization of the data in the bucket. These *views* are stored in each node's disk in a distributed manner, i.e.: the partial *view* at each node corresponds to its part of data. Each query from client will be scattered to all nodes in the cluster and an aggregated result is returned to the client. The *views* are updated incrementally at each node by applying the two functions only on the changes (add, update, delete) since the last update. An update on a view might be initiated by a predefined schedule or by a query from client with the *stale* parameter set to *false* (update before the query) or *update_after* (update after the query).

The definition of *views* (their *map* and *reduce* functions) are stored in a special type of documents called design documents. A bucket can have multiple design documents, each describing one or more *views*. *Views* within the same design document will have the same user-defined update schedule and, thus, be updated at the same time. This schedule, which is a combination of time, the number of changes since the last update, and the amount of *views* within a design document needs to be taken into consideration when designing the database so as to prevent node imbalances caused by very demanding view generation.

IV. System Model

For the purposes of this evaluation, we consider a microblogging system similar to the one provided by Twitter. In fact, we may, at times, borrow some of the vocabulary traditionally associated with this service to provide a more familiar understanding. This system allows a set of users to post small messages (we will use the limitation of 140 characters imposed by Twitter as a basis and we will, for simplicity, refer to them as *tweets*). All tweets of a certain user can be seen by visiting that user's page (what we refer to as the user's userline). In addition, users may follow one another. When a user follows other users, it has access to a special timeline page in which he may see all messages posted by those users, ordered by posting time. Of these operations, we identify the following 3 as the ones with the most representative and varied workloads and thus the ones in which we will focus for the remainder of the evaluation:

- *post* — A post represents a data write by part of a user. This write will be witnessed by all following users who will see the new message on their timeline or by users that access the user's userline. In a traditional RDS this would imply the writing of a new row on a single table (under the simple model we are considering).
- *userline* — A read operation of all messages posted by a user. This represents a read of relatively aggregated

data: all data created by a user. In a traditional RDS this would imply reading data from a single table. We limit each read to a limited number of tweets as is done in most applications. We considered this limit to be 50.
- *timeline* — A read operation of all messages posted by users which the owner of the timeline is following. This represents a read of relatively scattered data: data created by different users. In a traditional RDS this would imply, at the very least, a join between a table holding information about followers and another table holding information about tweets. It thus represents a more complex type of read. Just as with the userline, we consider that a single timeline operation retrieves up to 50 tweets.

Data describing tweets, users and their relations is stored in a decentralized database cluster composed of multiple servers using commodity hardware. As such, this cluster is vulnerable to internal network and node problems and crashes but also to possible external catastrophes which might result in entire datacenters becoming unavailable. Therefore, this cluster should be geographically distributed and data managed by it should be replicated to ensure maximum availability. In addition, we consider that the amount of data generated may reach such quantities that make it unfeasible to store the entire set of data in a single node and thus also consider the horizontal scalability provided by sharding (partitioning) techniques. This is not an unfundamented consideration as, according to [25], Twitter generated more than 12TB of data per day in 2010.

We further consider that the users who interact with our system are themselves geographically replicated and attempt to interact with our system using the cluster node that is closest to them. However, in the event that that particular node becomes unresponsive, we assume that the user is able to contact another node of the cluster. This can be achieved by having each user know the addresses of at least 2 nodes of the cluster for example.

Since our main objective is to evaluate the performance and characteristics of the data storage systems, we will consider that users access these databases directly, thus removing any interference that could have been caused by a traditional webserver.

V. Implementation

In order to be able to perform the evaluation of the aforementioned database systems, we had not only to setup the database clusters but also populate them with some information that is representative of the type of system we are considering. In addition, in order to generate the required workload to stress the system, we also had to develop a distributed workload manager and generator which deployed clients in different nodes and made them able to communicate with the 2 considered databases. In this section, we detail the steps we took in our implementation of each of the aforementioned components.

A. Data

Initially, our plan was to crawl Twitter using their API and retrieve a sample of real data from that system. However, recent changes to that API limited the amount of operations that could be done per day. In order to extract a sizable sample, it would take us several weeks with the imposed limitations.

Faced with this challenge, we decided to generate fictitional data but following, as close as possible, the data distributions observed on Twitter. According to [19], Twitter's follower and following distributions follow a power law similar to what happens in the rest of the web and, in particular, in the blogosphere. We collected some aggregated twitter data from [18] and checked that indeed a powerlaw distribution had the best fit among several other distributions with a R^2 of 0.88 for the follower distribution and 0.83 for the following distribution. As for the actual distribution for the number of tweets per user, we were unable to find satisfying data sets from which to get that information. [2] suggests that the number of tweets per user is somewhat proportional to the amount of followers but does not provide sample data from which we could try to derive a distribution. Since an accurate number of tweets has less impact than an accurate modelling of the follower/following distributions due to the operations we have chosen and their tweet limit, we ended up using a normal distribution for the number of tweets per user $N(\mu, \sigma)$ with $\mu = 2^{\log(numFollowers+1)}$ and $\sigma = \log((numFollowers + 1) * 10)$.

In the end, our experimentation was done with a dataset containing 400 users totalling 10080 follower relationships and 3540 tweets. In a SQLite normalized format this database amounted to 1.6MB in size.

B. Cassandra

Due to Cassandra's use of column families or tables to store data, the data modelling employed in this database was not so different from that of a normal RDS. However, since we want to take maximum advantage of horizontal scalability and decentralization, we attempted to denormalize data as much as possible so that an operation is more likely to be able to gather all necessary data from a single table in a single or reduced number of servers. In the end, Cassandra's data model included the following tables:

- *Users* — Table containing rows with a single column describing the username of each user. The primary key of this table is, obviously, the only column it contains.
- *Followers* — Table containing rows with 2 columns both describing the username of different users. The first represents a user who is being followed and the second a user that follows the first. The primary key is composed of both columns which creates a primary index over the first column and a secondary index over the second.
- *Tweets* — Table containing rows with 3 columns: id of a tweet, username of the author of the tweet and the actual message. The primary key is the id of the tweet and there is no additional index.
- *Userline* — Table containing rows with the same 3 columns as the *Tweets* table. The difference is that the

positions of the username and tweetid were exchanged and these 2 columns constitute the primary key of this table. Thus this table allows indexing of tweets based on the user who posted them (necessary for userline operations).

- *Timeline* — Table containing rows with 4 columns that describe a tweet in a user's timeline: the username of that user, the id of the tweet, the username of the author of the tweet and the tweet's body. The primary key is composed by the first 2 columns thus creating a primary index by owner of timeline and a seconday index by tweet in such a timeline.

A userline or timeline operation will simply read all tweets associated with a particular user. A tweet operation is more complex in that it requires 2 different queries. In a first query we have to retrieve the list of followers of the poster from the *Followers* table. In a second query we perform a batch addition of tweets. This batch addition encompasses not only the addition to the *Tweets* table and the *Userline* of the author but also to the *Timeline* of each of the followers.

The interfacing with the Cassandra database was made using the recommended CQL interface which borrows much of its syntax from the familiar SQL. For both read and write queries we used the default consistency level of *ONE* which specifies that a response will only be returned to the user when at least one node has confirmed the writing of the new data or has returned a result for the read query. This provides the best possible latency and doesn't constitute a great sacrifice of consistency since the multiple eventual consistency mechanisms (read-repair, broadcast of write to all replicas, etc.) employed by Cassandra make nodes converge quite fast and certainly inside our consistency tolerance period of a few seconds to minutes.

C. CouchBase

Taking a similar denormalization approach as that employed in Cassandra's model (with some Couchbase specific optimizations), we applied a model consisting of one single bucket with four types of documents:

- *Users* — Each document represents one user with its username and an array of all his/her followers. The key of a *users* documents is simply the username.
- *Tweets* — Each document represents a tweet with its id, the author's username and the message body. The key of each *tweets* document is the corresponding tweet id.
- *Userline* — Each document contains the collection of all tweets posted by the user. This collection is updated every time the user posts a tweet. Since each *userline* is updated only by the owner, we don't have to worry about concurrent updates or their chronological order. The key of a *userline* document is the owner's username with a predefined prefix to distinguish it from *users* documents.
- *Timeline* — Each document corresponds to a tweet on a user's timeline. It contains the owner of the timeline's username and all information about the tweet, i.e.: tweet's id, author's username and the message. The key of a *timeline* document is the combination of owner's username and tweet's id.

Initially, our approach for timeline documents was similar to userline. However, due to the high rate of concurrent updates to a timeline, our attempt to put a lock on the document before an update proved to degrade the performance drastically. Thus we decided to separate each entry into a separate document as described. We are also aware that for a real case usage, a userline document will grow unbounded. This can be an issue since a document in a Couchbase bucket is limited to 20 MB [9] and we generally do not want to keep transferring old tweets over the network. The problem can be easily solved by dividing the whole userline into smaller parts according to a period of time, i.e.: a month, a year quarter. The list of all userline documents for a user can be stored in a master userline document and a client will first retrieve this master document before retrieving the actual userline(s).

In our model, querying for the userline of a user simply involves retrieving the corresponding document with a key derived from the user's username. On the other hand, retrieval of a user's timeline is achieved by using Couchbase's view. Our timeline view indexes all the timeline documents by its owner's username and the tweet's timestamp, which is reflected in its id. Each row of the timeline view includes in its *value* the tweet's message and the author's username. Thus we are able to obtain a user's timeline in a single operation.

Since we were expecting a large amount of queries on the timeline view and such queries do not require real-time data, we set *stale=ok* and scheduled the servers to check the number of new changes every second and update the view if changes exceed 5. This is quite an aggressive setting [10], however, due to the short duration of our experiments, we decided to deploy such schedule to enforce updating of timelines and their consistency.

In the case of the tweet operation, it is slightly more complicated as the posting of a new tweet will occur in three steps: adding a new tweet document, updating the corresponding userline document and adding new timeline documents, one for each of the user's followers.

D. Distributed loader and manager

To be able to launch coordinated workloads from multiple slave nodes as well as coordinate the actual deployment of the database clusters over several nodes and study the acquired data, we developed a custom Python script which we called *pydloader* and a set of helper scripts. Pydloader is a Python script which makes use of several python libraries such as Boto (for AWS interaction), Paramiko (for SSH sessions) and RpyC (for RPC communication). It contains 2 main components:

- *Manager* — Run directly on the laptop of the user performing the experiment, it provides an interactive console interface where the user may issue commands to deploy new remote nodes, install slaves on those deployed nodes via SSH and then communicate and perform actions on those slaves through a RPC communication protocol.
- *Slave* — Launched on a remote node by the manager, the slave can operate in one of 2 ways. As a database slave, it is able to setup the various database services running on that node to work in tandem with those running in

other nodes and thus allowing the formation of a database cluster. It also supports the populating of a database cluster with pre-generated data and the termination of the database service or of the entire cluster. As a workload slave, it is able to spawn multiple clients that interact with a database cluster issuing all the operations deemed relevant on that cluster all the while gathering logs and statistics regarding latencies and other times.

VI. Evaluation and Comparison

The evaluation of the chosen NoSQL database systems was performed using the Amazon Web Services (AWS) infrastructure, namely the Elastic Compute Cloud (EC2) and Elastic Block Storage (EBS) services. The choice of AWS as our evaluation platform was based not only on some previous familiarity with the platform but also due to some helpful features it provides (loosely synchronized clocks, snapshots and machine images, different instance types, extensive API). In addition, some of the database systems we evaluated offer special configuration helpers that facilitate the deployment of clusters in this environment.

For the database nodes, we used small standard on-demand (*m1.small*) instances with 1.7GB of RAM, 1 compute unit, 1 small virtual core, 150GB of disk storage and moderate I/O performance. A compute unit is roughly equivalent to a 1.0 to 1.2 GHz 2007 Opteron or 2007 Xeon processor [1]. This is a rather low-end node for a database but it allows us to more easily stress the cluster with lower loads and to study the effectiveness of deployment in multiple cheap nodes instead of a reduced number of very powerful ones. For the workload generating nodes, we used micro on-demand instances (*t1.micro*) which offer the same CPU capacity but sacrifice RAM (615MB) and disk storage (no disk storage on the node, only network-attached storage). These limitations make it about 3 times cheaper than a small node and thus allowed us to deploy a greater number of them to generate an appropriate workload.

All instances were loaded with a custom Amazon Machine Image (AMI) based on the official Amazon Linux Image which is itself based on Red Hat Enterprise Linux 5 and 6. This custom AMI was configured by us to have the 2 database systems (Cassandra, Couchbase) pre-installed along with *pydloader*. The root filesystem of the nodes was backed by this AMI which was loaded into an EBS (and thus network-attached) storage unit with 8GB. To avoid latency degradation and increased contention on the communication links, the database nodes stored the actual database data, commit logs and other data structures in the instance's own storage (the 150GB available to the small instances).

For the testing, we deployed 6 database nodes across 2 datacenters in Ireland. The workload nodes numbered at 12 and were distributed equally among both datacenters.

For replication, Cassandra used a replication factor of 4 in order to have a total of 2 replicas per availability zone. With a consistency level of one, this allows clients to always get data from the zone closer to them even in the event of a node failure per zone. It also ensures that data is available even if an entire zone happens to fail. In Couchbase, due to the limit of the replication factor, we could only deploy a replication factor of 3 in our experiments. This settings means that, when the number of servers in the cluster allows, for any piece of data, there will be one active replica and three other inactive ones. All access to this piece of data will be directed to the active replica to ensure strong consistency. One of the inactive replicas will be activated in the event of a crash in the main replica.

A. Ease of configuration and setup

Usually, the more automated the various internal processes of a database system (replication, partitioning, etc.) are, the easier and faster it is to deploy a working database cluster.

In this aspect, Cassandra is indeed very easy to setup. After installation using the provided packages for RHEL, configuring a Cassandra node is as easy as specifying listen and broadcast addresses, a list of seeds that will help new Cassandra nodes to discover other nodes in the same cluster and the used partitioner and snitch classes. In the case of EC2, snitch configuration is specially easy with Cassandra automatically detecting the topology details with the *Ec2Snitch* or *Ec2MultiRegionSnitch*. Most of the default values for the remaining properties will apply to most systems and do not need to be configured but can be if the administrator so desires. Creation of keyspaces and configuration of the replication options for each keyspace can be made using any of the numerous Cassandra interfaces: CQL via Thrift, the command line script cassandra-cli or a graphical web-based administration console provided by DataStax.

Installing Couchbase is also pretty simple. After installing the provided packages for Linux, we start the software, configure the path for the data directory, RAM quota, administrator credentials and a single-server cluster is ready. To setup a N-servers cluster, we simply start N single-servers and add them, one-by-one into a common cluster. Configuring and monitoring Couchbase servers and clusters can be done via command line tools like couchbase-cli, cbstats,…, through the graphical web-based administrator console or even through its RESTful API.

Both databases also have specially tailored AWS AMIs which make the deployment of single region database clusters on AWS even easier. For multiple region deployments, one still has to resort to manual configurations though.

B. Setup time

In this section, we explore the time it takes for a whole database cluster to start from scratch and populate its storage with data from our SQLite data set. This is a useful metric in that it provides an idea of just how fast one can expect node/data center recovery or data migration to new locations to be.

After starting the 6 Cassandra database instances, it took them approximately 2 minutes to acknowledge the existence of one another, form and stabilize the cluster ring. After the ring stabilized, we then proceeded to the populating of the cluster with data from our SQLite database. This populating phase took an average of 392 seconds (6 minutes and a half) to complete and stabilize. For comparison, a single node deployment took around 1 minute to deploy. Comparing these values, it seems that setup time grows approximately linearly with the number of nodes.

Similarly, the 6 instances took slightly more than 2 minutes to sequentially start the Couchbase Server service and add each other into the cluster. Since all the servers were started with no data, the *rebalance* process, which is a background process required to migrate data from active servers to the newly added ones and allow the new servers to be fully functioning, took almost no time. It is worth mentioning that the *rebalance* process on a cluster with pre-existing data might take quite some time as the cluster has to rebalance and redistribute data according to the new cluster configuration which may involve very big data transfers and rebuilding of views. During this process, all servers remain functional and responsive. However, the increase in intra-cluster traffic and CPU utilization might have some impact on the overall performance. Lastly, the process of populating data on these Couchbase servers took slightly less than 2 minutes, which is only around twice as much as on a single server.

C. *Operational Latency*

In this section, we examine the latency associated with the 3 operations we identified as being the most significant in our system model. To that end, we launched a total of 200 threads distributed across the workload nodes, with each thread simulating a user and performing random operations from the selected 3 for a period of time of approximately 4 minutes. The time between the issuing of the request and the return of the data by the database system is measured as the operational latency for each of these operations and is logged.

Data relevant to tweet latency can be seen in Figure 1. Due to the denormalization process we employed, in Cassandra a tweet will write one row to the *Tweets* and *Userline* tables and as many rows as followers in the *Timeline* tables. While this introduces a lot of load in the tweet operation, it allows the userline and, specially, the timeline operations to have very little latency as can be seen in Figure 2 and Figure 3. If this were not the case, these operations would have to wait for results from several different tables which, due to partitioning, might be spread across several different nodes. In Figure 4, Figure 5 and Figure 6 we show this situation by plotting the latency of the same operations for a more normalized data model of Cassandra where the body of the tweets is only kept at the *Tweets* table and not at the *Userline* and *Timeline* tables. In this way, a userline or timeline operation must execute up to 50 selection operations on the *Tweets* table to be able to retrieve the bodies and authors of the respective tweets which increases both the load and latency of those operations. Unfortunately, this increase in load was sufficient to overload the database servers therefore the latency of the tweet operation was also negatively affected.

In the previous graphs we've also plotted results obtained with Couchbase. For tweet and userline operations, Couchbase yields a slightly higher latency, yet still comparable to that registered with Cassandra. However, due to the fact that timeline operations utilize Couchbase's *views*, which are known to have greater latency than retrieval of documents based on keys, the tweet latency of Couchbase is much worse than that of Cassandra. We also considered a normalized version of Couchbase, analogous to the one described for Cassandra in the last paragraph. The results are plotted alongside those of Cassandra in the relevant figures and show a similar increase in latency for the userline and timeline latencies although not as pronounced as in the Cassandra case because Couchbase was able to operate under this model without overloading the node resources.

In Figure 7 we also compare the tweet latencies obtained during the denormalized experiment and group them according to the number of followers the tweeting user has. It is clear that in both Cassandra and Couchbase the immediate denormalization performed by our implementation has a growing cost with the number of followers. While with a small number of followers this might not be a problem, celebrities or other users who, due to their roles, naturally have hundreds of thousands of followers cannot use this type of denormalization on insertion. For these users, we suggest that this denormalization be done in an asynchronous and periodical manner at the expense of increased convergence/consistency time - the interval from which the tweet is posted until everybody is able to see it.

We were unable to find native mechanisms in Cassandra to implement this so it would have to be done using custom external processes. However, Couchbase's periodic and asynchronous view updates using MapReduce computations appears to be an ideal fit for this. To test this functionality, we considered a new Couchbase model model, where we have only two types of documents: *users* and *tweets* with a small modification to the *tweets* documents to include the list of all followers of the author into each tweet. The timeline view now goes through each tweet document and produces a row for each of the followers in the list. We also need another view for the userline which also goes through each tweet and produces one row with the author's username and tweet's id as the key. The latencies obtained with this model can be seen in Figure 8. The two operations userline and timeline are now very much alike which explains why they have so similar latencies. Interestingly, the timeline performance is actually better than the timeline performance in the denormalized model. For tweet operations, the fact that posting a tweet now doesn't require adding new timeline documents also improves its latency significantly. Note that with this new implementation, our previous problem with the proportional relationship between a user's number of followers and the tweet operation's latency would be solved since the updating of views would be done asynchronously and incrementally.

D. *Reconfiguration latency*

In this section, the time it takes for the database cluster to adapt to changes in the cluster configuration is analyzed. This is important in that it provides an idea of how flexible is the scalability of these database systems and of how well they handle churn. We expect these reconfigurations to be able to

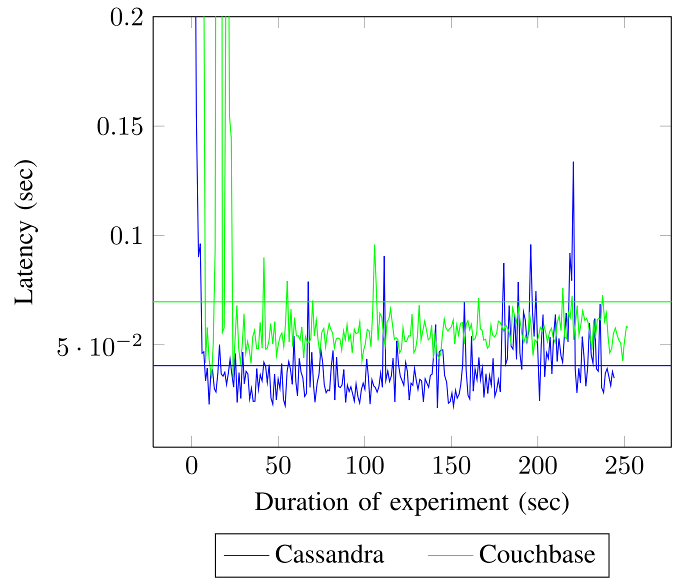

Fig. 1. Operational latency for *tweet* operations in a 4 minute experiment with 200 simultaneous clients using Cassandra and Couchbase clusters

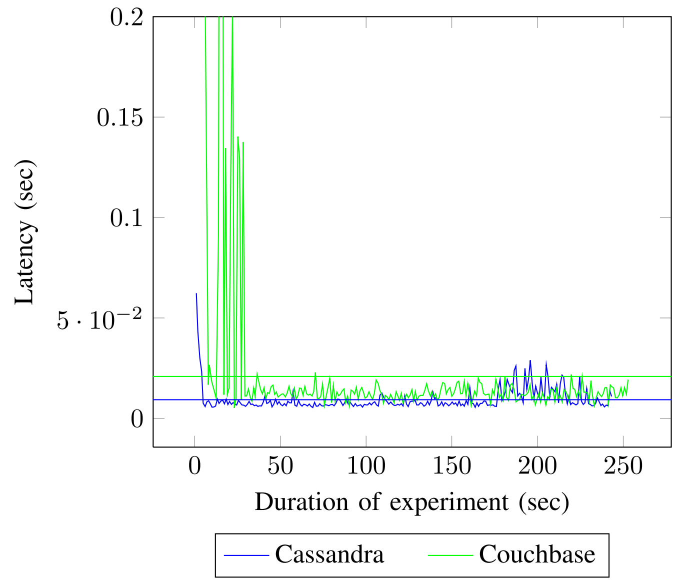

Fig. 2. Operational latency for *userline* operations in a 4 minute experiment with 200 simultaneous clients using Cassandra and Couchbase clusters

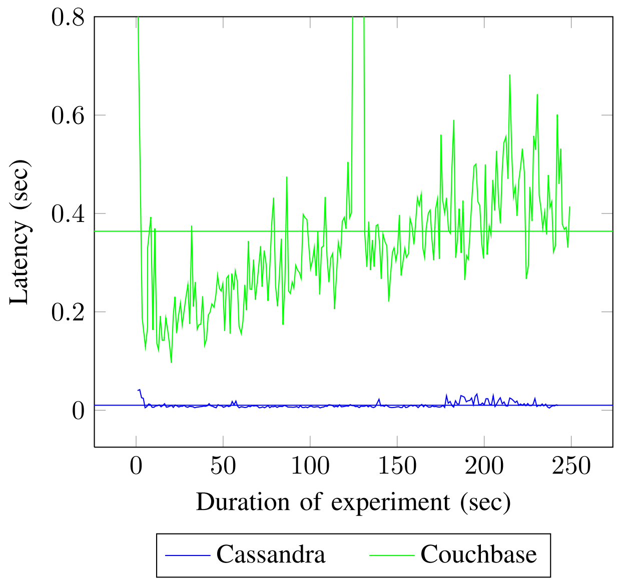

Fig. 3. Operational latency for *timeline* operations in a 4 minute experiment with 200 simultaneous clients using Cassandra and Couchbase clusters

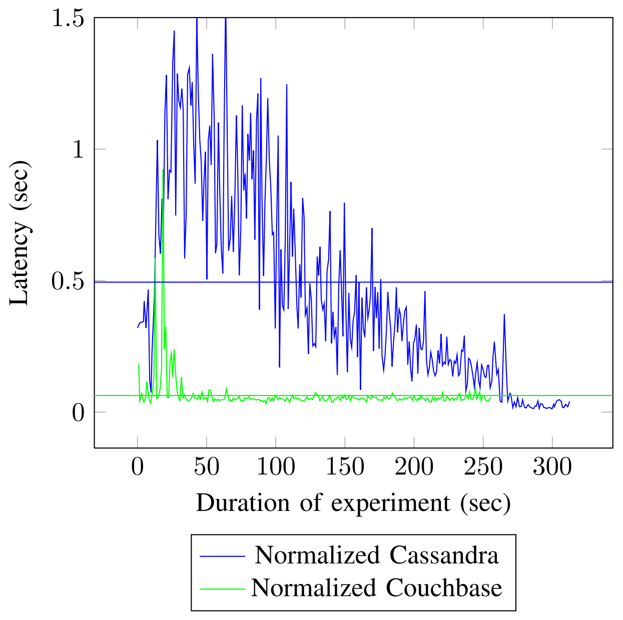

Fig. 4. Operational latency for *tweet* operations in a 4 minute experiment with 200 simultaneous clients using a normalized Cassandra and Couchbase data model.

happen without any downtime and with minimal interference to the provided service so we also analyze what happens to ongoing requests while this reconfiguration is taking place.

In Cassandra, and using our data set, adding a new node to the ring takes an average of 5 minutes. This includes the time to actually join the ring and to bootstrap its share of the data from the other nodes. Figure 9 shows the latency of tweet operations over a small 60 seconds experiment with 200 users done with 4 active cassandra nodes and 2 new joining nodes. Comparing it with Figure 1, it is clear that this bootstrapping and ring restructuring puts the system into some extra load and should not be done during peak hours if latency is an important factor. It also shows that using Cassandra in a high churn environment such as that of voluntary computing may severely adverse the latency of the system as at any one time we'll have multiple disconnections and connections of new nodes which hardly allow the stabilization of the ring structure.

We performed the same experiment with Couchbase and the result can be observed in Figure 10. Although the add new server operation took almost no time in Couchbase, the *rebalance* process required to bring this new server into operation took much longer than Cassandra's bootstrapping process. The process was recorded to take 25 minutes in one of our experiments where we added two new nodes into a 4-node cluster in a very low load condition and with only

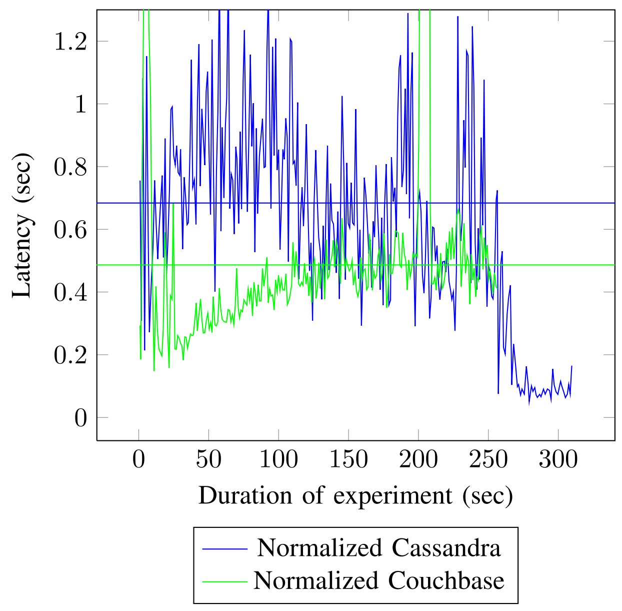

Fig. 5. Operational latency for *userline* operations in a 4 minute experiment with 200 simultaneous clients using a normalized Cassandra and Couchbase data model.

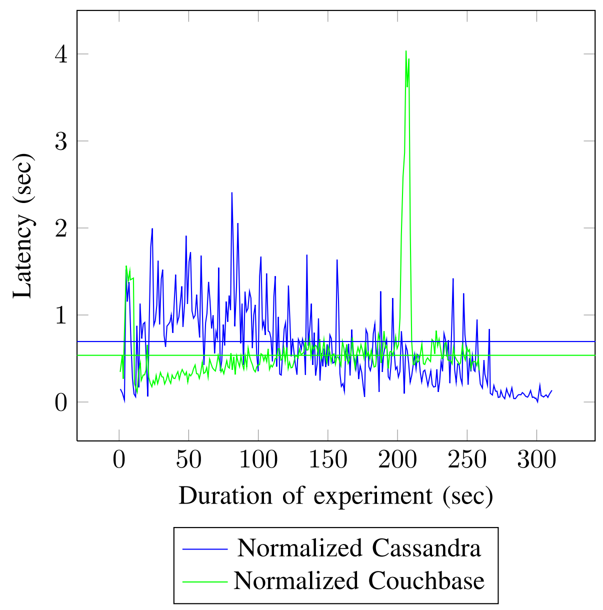

Fig. 6. Operational latency for *timeline* operations in a 4 minute experiment with 200 simultaneous clients using a normalized Cassandra and Couchbase data model.

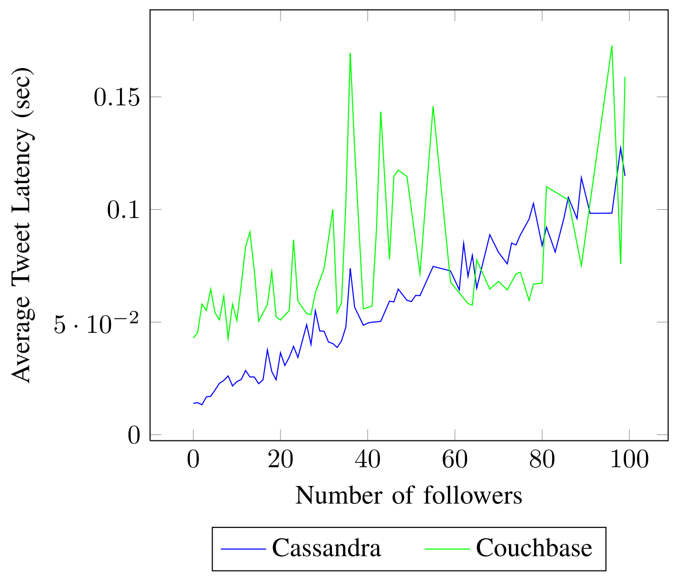

Fig. 7. Average operational latencies for *tweet* operations grouped by number of followers and registered in a 4 minute experiment with 200 simultaneous clients using a denormalized Cassandra and Couchbase data model

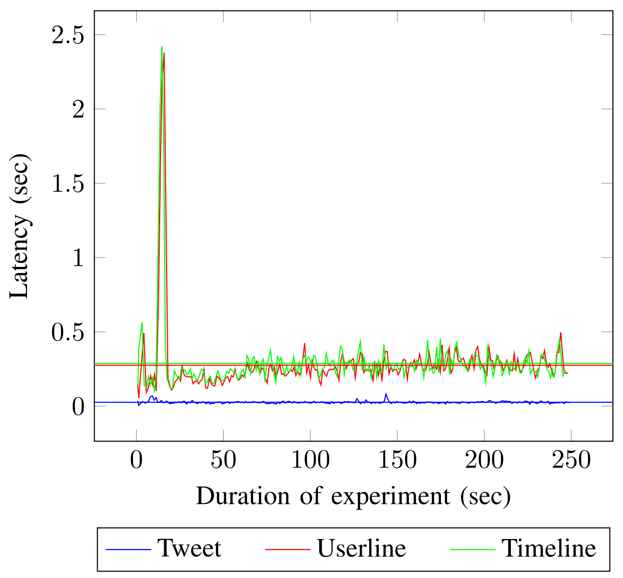

Fig. 8. Operational latency for all operations in a 4 minute experiment with 200 simultaneous clients using the Couchbase model with asynchronous updating of timeline and userline views.

the initial data. The *Rebalance* process is designed to be able to perform *"online"* thus keeping the cluster functional and responsive during this process. However, it is known to have a significant impact on the utilization of resources, including disk IO, CPU, RAM usage and network bandwidth [11]. The result from our experiments indicates that the impact of this *rebalancing* seems negligible for non-view operations, i.e.: tweet and userline while the impact on view operation is very significant. For these reasons, Couchbase should also be used in environments with very little churn.

E. Consistency

In this section, we examine how long it takes for a certain update to the database data to be reflected in all nodes of the cluster. To that end, we create a new tweet and have all workload nodes query their closest cluster nodes looking for a tweet with a matching id and content and register the time it takes for that to happen.

In our testes with Cassandra, the average time for a tweet to be observed by every workload node was of approximately 0.096498 seconds with a standard deviation of 0.096319. The reason for this high deviation is that nodes in the same datacenter tend to notice the write quicker than those in other datacenters. On the other hand, Couchbase with its strong

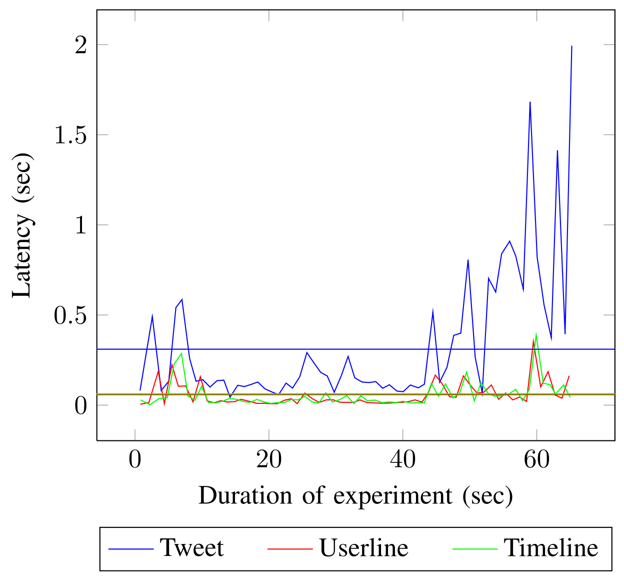

Fig. 9. Operational latency for the 3 operations in a 1 minute experiment with 200 simultaneous clients using 4 active Cassandra database nodes and 2 joining ones.

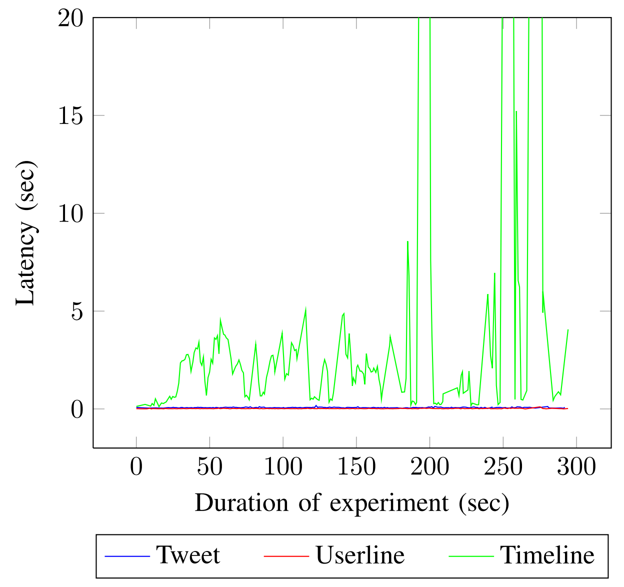

Fig. 10. Operational latency for the 3 operations in a 4 minute experiment with 200 simultaneous clients using 4 active Couchbase database nodes and 2 joining ones.

consistency shows a better result of an average of 0.007501 seconds and a standard deviation of 0.012476 for a new tweet to be seen by other users. However, it is expected to yield a lower performance for timeline operations where the update of timeline view is triggered by a predefined schedule. For a better consistency, one will have to increase this update rate at the cost of higher CPU utilization and higher disk IO.

F. Replication

In this section, we examine and compare replication features of each database system and see how the system behaves with the crashing of database nodes.

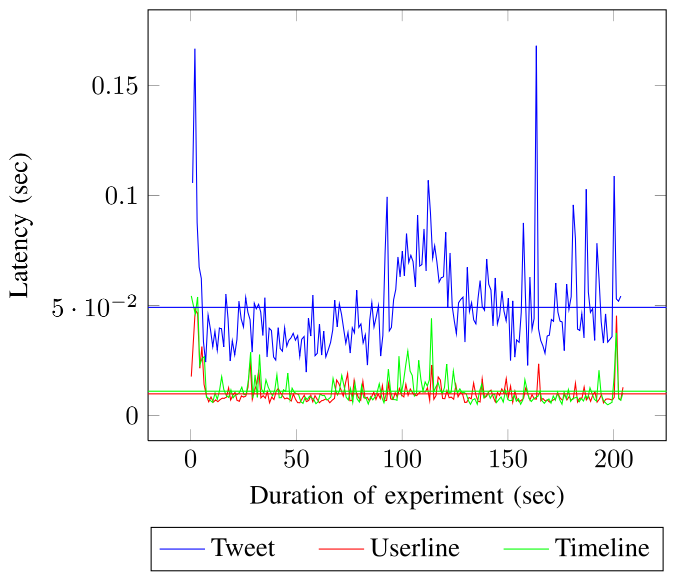

Fig. 11. Operational latency for the 3 operations in a 200 second experiment with 200 simultaneous clients using Cassandra. 100 seconds into the experiment, 2 of the database nodes crash.

In Figure 11 we plot the latencies of the operations in a 200 second experiment where 2 of the 6 nodes in the Cassandra cluster crash after 100 seconds of experiment time. As can be seen this had some effect on the latency of these operations as the ring adapted and compensated for the failures but once it stabilized the latency returned to a relatively normal state. In addition, no errors were registered (for any kind of operation) at any point, indicating that the replicas performed their duty correctly and kept the system responsive.

The experiment was repeated with Couchbase and the result can be seen in Figure 12. The behaviour observed in this experiment is similar to that registered in subsection VI-D where the impact of changes in cluster configuration is more significant in the timeline operation and less in the other two. Except for the two short peaks, the performance of userline and tweet operations remain the same, resulting in a relatively stable cluster. Although there were no records of operation errors, the latency impact in some of the requests during the crash is so severe that in a normal situation these would be registered as errors and a retry would be sent.

Cassandra allows a very flexible configuration of replication options, being able to specify a per-datacenter replication factor with no hardcoded limitations. Of course, the higher the replication factor, the greater the toll an operation takes over the system and the more disk and memory space is wasted. Due to this reason, most Cassandra deployments use a replication factor of 2 or 3 nodes per datacenter, depending on their consistency and reliability requirements [13].

This is not the case with Couchbase. Although the replication factor can be flexibly configured for each data bucket in a cluster, this value is ultimately limited to 3. The replication between different datacenters can be achieved by Cross Datacenter Replication (XDCR) functionality. Using XDCR, changes in a bucket in one datacenter will be transferred to the corresponding bucket in the other datacenter after it

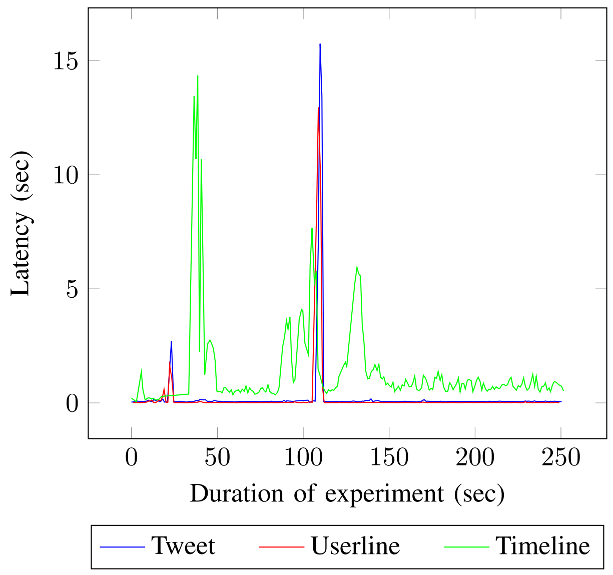

Fig. 12. Operational latency for the 3 operations in a 240 second experiment with 200 simultaneous clients using Couchbase. 2 of the database nodes crash (at 30 seconds and 100 seconds into the experiment).

is persisted on disk in the original one [8]. This process is performed automatically, periodically, and, typically, in batch, with some configurable settings such as number of concurrent streams, size of a batch package and update interval ranging from 1 minute to 4 hours [12]. This implies that changes in one datacenter will undergo a high latency before it is reflected in the other. Therefore, this technique is only applicable for applications without a strict consistency requirement.

G. *Scalability/Load balancing*

In this section, we examine how data is distributed among the database nodes belonging to the cluster. In particular, we are concerned with how fair the load distribution is so that it doesn't overload some nodes while leaving others with very little load. Most of the results shown here were collected from the monitoring tools accompanying the 2 database systems.

According to the *nodetool* program that accompanies Cassandra, the average data ownership percentage of each of the 6 nodes in the cluster ring after the setup phase was of 16.68% with a standard deviation of 1.23%. This indicates quite a fair distribution of data. More surprising is the fact that even with node crashes and rejoins, the ring adapts and ensures that the average ownership continues to be fair with negligible difference from the previously mentioned average.

Using the Web Console, we observed a similar behavior on the Couchbase cluster. The collected data indicates that both active data and inactive replica data are distributed evenly over the cluster with a standard deviation of 0.65% for data on disk and 0.04% for RAM usage. In the event of node joins or crashes, a *rebalance* ensures the almost perfect balance between servers. However, even without rebalancing, after the crash of 2 nodes, our 6-node (now 4) cluster still shows a very well-balanced distribution with a standard deviation of only 0.89% for data on disk and 0.05% for RAM usage. This proves the efficiency of the replication algorithm employed by Couchbase.

H. *System load*

In this section, we are concerned with the analysis of CPU, memory and disk load in a single node throughout the duration of the experiment. To this end, we used the *vmstat* tool to collect CPU, memory and disk usage data every 5 seconds during an experiment.

The collected results from normal denormalized experiments with both Cassandra and Couchbase can be seen in Figure 13, Figure 14, Figure 15 and Figure 16. For the most part, results follow from the use of default configuration values (minus the ones explicitly detailed in this paper). The obtained results are very curious, showing that, with the same load, Cassandra has an average of 67% cpu idle time whereas Couchbase's average idle time is of little more than 1.6%. The difference between block write rates is also very significant with Cassandra doing an average of 298 block writes every 5 seconds and Couchvase doing almost 10 times more with an average block write rate of 2366 writes per 5 seconds.

While we do not have concrete evidence as to where the differences come from, we conjecture that the increased CPU usage and block writes by part of Couchbase are a direct result of the updating of the indexes and views using MapReduce and the immediate persistence of written data to persistent storage. Cassandra, on the other hand, appears to delay the writing of its in-memory column families to disk, risking some data loss upon failure but needing less frequent access to persistent storage. The justification for Couchbase's CPU usage is further backed up by noticing how it does not appear to decrease after the end of the experiment. However, this apparent increased load in Couchbase does not seem to reflect on its response latency suggesting that it is not a result of overloading but rather a smart use of idle resources for asynchronous tasks. If this is so, setups where power consumption is a limiting factor should perhaps attempt to lessen this effect.

Common to both systems, however, is the reduced number of block reads. This shows a very good use of available memory. Since our small dataset managed to fit entirely in memory, both database systems did very little reading from the disk, being able to provide responses directly from data in memory. In a production environment with terabytes of information this will certainly not happen but we expect that both systems will have good cache mechanisms which attempt to keep in memory the most popular data.

Regarding memory, Cassandra appears to be slightly more memory-hungry than Couchbase but the difference isn't very significant. We imagine this might have something to do with the memory efficiency of the JVM used by Cassandra versus the memory efficiency of Couchbase's implementation in C++ and Erlang.

I. *Disk space usage*

The size occupied by data, metadata and other helper structures in a database system is a very important metric in

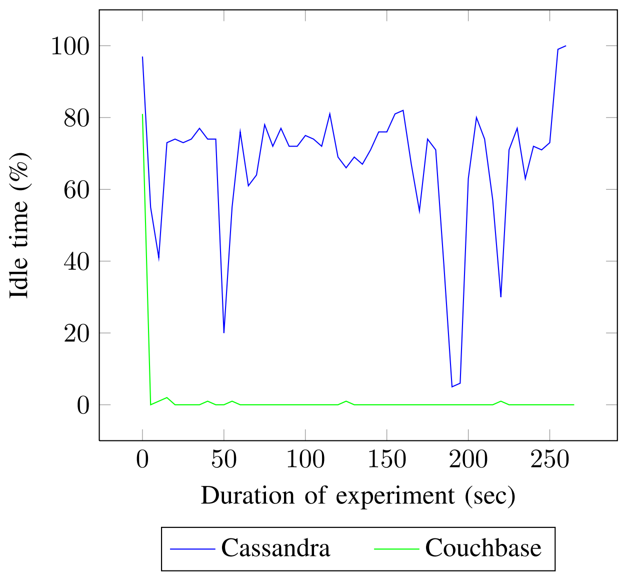

Fig. 13. CPU Idle times for Cassandra and Couchbase under the normal denormalized experiment described in subsection VI-C

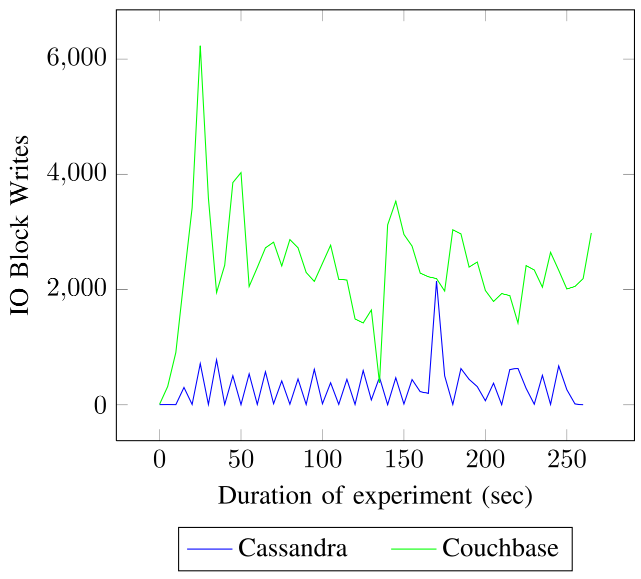

Fig. 15. IO block writes for Cassandra and Couchbase under the normal denormalized experiment described in subsection VI-C

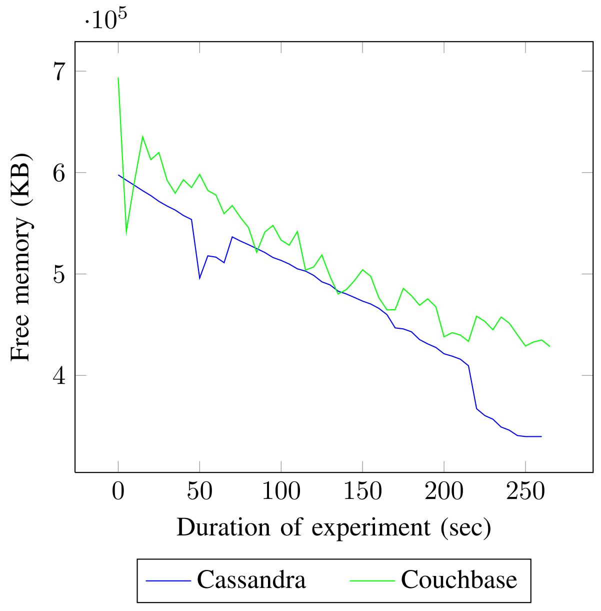

Fig. 14. Free memory for Cassandra and Couchbase under the normal denormalized experiment described in subsection VI-C

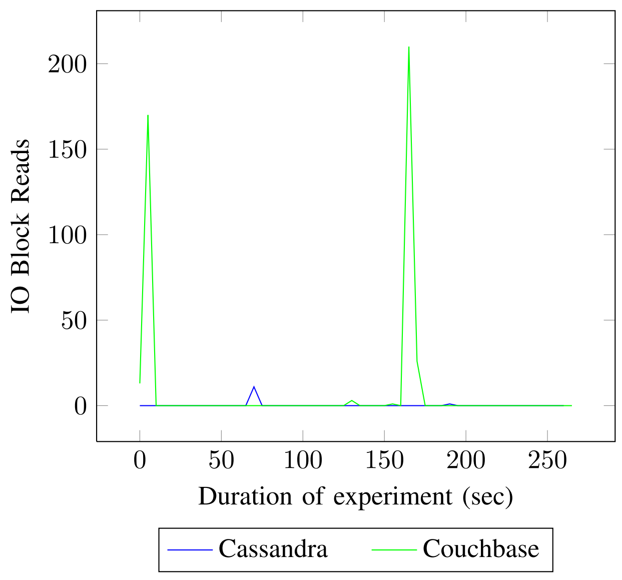

Fig. 16. IO block reads for Cassandra and Couchbase under the normal denormalized experiment described in subsection VI-C

that it is directly correlated with the number of nodes/drives needed to be able to store all system data.

The deployment of our test database in the denormalized Cassandra version occupies 16.31MB of space in a single node (a 10x increase over the normalized SQLite data). Considering the distributed clusters we've been evaluating, we have witnessed an average disk usage of 9.5MB per node. This value reflects the size of the data stored at each node considering both partitioning and replication. For comparison, the normalized Cassandra version occupies 2.8MB in a single node and 1.6MB per node in a distributed environment. Besides direct disk space occupied by data, the commit log can also grow quite large. In fact, Cassandra's commit log after populating occupies a total of 54MB on the injection node. In addition, nodes running Cassandra should, at all times, have, at least, 50% of free disk space because some periodic operations such as compaction of tables may temporarily create a whole copy of stored data. This represents quite a waste of available storage space specially if coupled together with the kind of denormalization we have employed during this evaluation. Cassandra developers justify this waste with the fact that fully copying local data during compactions allows Cassandra to perform these operations faster and consider that storage space is cheap and readily available. In addition, Cassandra does not allow the specification of per-node storage space limits which could be taken into account by the partitioning process. The specification of different number of virtual nodes in each host might achieve this effect but is hardly accurate. Overall, these limitations make Cassandra's unfeasible on decentralized voluntary/community systems where disk storage is often regarded by the users as a precious commodity and is highly heterogeneous.

In our dernormalized Couchbase, after the cluster has set up and populated the *views*, we observed a total of 250MB of data on disk, distributed across the cluster. Since Couchbase also does not allow the control of the amount of data stored at each node, it too is not a feasible option to be deployed in a voluntary computing environment. However, unlike Cassandra, it does not appear to have a requirement on a minimum amount of free space available in the system which allows it to make a more effective use of the nodes' resources.

VII. Conclusion

The applicability of NoSQL database systems, in particular, Cassandra and Couchbase to a microblogging application has shown very interesting results and place both Cassandra and Couchbase as solid alternatives to a more traditional and centralized RDS based storage in this context.

These systems show great promise in terms of data handling and distribution capabilities. We have seen, with Cassandra and Couchbase, how easy it is to setup and deploy clusters based in these systems and how, with some changes to the data modelling and system interaction, we are able to exploit their capabilities to provide a fast, reliable and scalable service which does not rely on a central core of high capacity nodes but rather on several distributed nodes running commodity hardware.

It is also clear that no 2 NoSQL database systems are the same. Cassandra and Couchbase, although sharing a similar philosophy, provide different sets of features and even different data models which lead to different performance and functional advantages and disadvantages. With more than 150 different types of NoSQL databases, careful study is needed to review and select the database which best suits the needs of the specific project.

This is not to say that NoSQL systems are a solution for every problem. Some applications cannot function properly without transactional support and at a smaller scale, NoSQL systems might actually turn out to be less efficient than traditional RDS. In addition, the increased data storage usage resultant from denormalization procedures inherent to NoSQL systems, along with the lack of a common interface as that provided by SQL in RDS and their relative immaturity may deter or slow down the conversion of existing services as it would require significant changes to the service architecture or code base [16]. In addition, from evidence collected in this paper, it seems that these systems also need to improve their churn tolerance and resource usage control if they are ever to be used as reliable and fast data stores in voluntary computing environments.

Given the limitations of RDS and NoSQL systems, it is our firm belief that the future of large-scale data storage will not pass through a victory of one of these 2 types of storage systems over the other but rather on a hybridization of these systems, taking the best features from each side into an harmonious combination. In fact, this is exactly the direction that these systems are going towards, with RDS such as MySQL adopting NoSQL data stores and access as well as flexible schemas [30] and NoSQL databases such as Cassandra moving on to CQL interface which closely mimics the abstraction layer provided by SQL or CouchBase's views that allow a more normalized data model.

References

[1] Amazon. *Amazon EC2 FAQs*. URL: http://aws.amazon.com/ec2/faqs/ (visited on 05/15/2013).

[2] Beevolve. *An Exhaustive Study of Twitter Users Across the World*. URL: http://www.beevolve.com/twitter-statistics/#f3 (visited on 04/26/2013).

[3] Rick Cattell. "Scalable SQL and NoSQL data stores". In: *SIGMOD Rec.* 39.4 (May 2011), pp. 12–27. ISSN: 0163-5808. DOI: 10.1145/1978915.1978919. URL: http://doi.acm.org/10.1145/1978915.1978919.

[4] Peter M. Chen and David A. Patterson. *Storage Performance - Metrics and Benchmarks*. 1993.

[5] Adrian Cockcroft and Denis Sheahan. *Benchmarking Cassandra Scalability on AWS - Over a million writes per second*. URL: http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html (visited on 05/20/2013).

[6] Couchbase. *Couchbase Developer's Guide 2.0*. URL: http://www.couchbase.com/docs/couchbase-devguide-2.0/ (visited on 05/22/2013).

[7] Couchbase. *Couchbase Server 2.0 Manual*. URL: http://www.couchbase.com/docs/couchbase-manual-2.0/index.html (visited on 05/22/2013).

[8] Couchbase. *Cross Datacenter Replication (XDCR) in Couchbase*. URL: http://www.couchbase.com/docs/couchbase-manual-2.0/xdcr-topologies.html (visited on 05/22/2013).

[9] Couchbase. *Document values in Couchbase*. URL: http://www.couchbase.com/docs/couchbase-devguide-2.0/couchbase-values.html (visited on 05/22/2013).

[10] Couchbase. *View Update Schedule in Couchbase*. URL: http://www.couchbase.com/docs/couchbase-manual-2.0/couchbase-views-operation-autoupdate.html (visited on 05/22/2013).

[11] Couchbase. *When to rebalance in Couchbase*. URL: http://www.couchbase.com/docs/couchbase-manual-2.0/couchbase-admin-tasks-addremove-deciding.html (visited on 05/22/2013).

[12] Couchbase. *XDCR Settings in Couchbase*. URL: http://www.couchbase.com/docs/couchbase-manual-2.0/couchbase-admin-restapi-xdcr-change-settings.html (visited on 05/22/2013).

[13] Datastax. *About data distribution and replication*. URL: http://www.datastax.com/docs/1.2/cluster_architecture/data_distribution (visited on 05/20/2013).

[14] Datastax. *Apache Cassandra 1.2 Documentation*. URL: http://www.datastax.com/docs/1.2/index (visited on 05/05/2013).

[15] Prof. Dr.-Ing. Stefan Edlich. *NOSQL Databases*. URL: http://nosql-database.org/ (visited on 05/20/2013).

[16] Twitter Engineering. *Cassandra at Twitter Today*. URL: http://engineering.twitter.com/2010/07/cassandra-at-twitter-today.html (visited on 05/20/2013).

[17] Robin Hecht and Stefan Jablonski. “NoSQL evaluation: A use case oriented survey”. In: *Proceedings of the 2011 International Conference on Cloud and Service Computing*. CSC ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 336–341. ISBN: 978-1-4577-1635-5. DOI: 10.1109/CSC.2011.6138544. URL: http://dx.doi.org/10.1109/CSC.2011.6138544.

[18] Infochimps. *Twitter Census*. URL: http://www.infochimps.com/collections/twitter-census (visited on 04/25/2013).

[19] Akshay Java et al. “Why we twitter: understanding microblogging usage and communities”. In: *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*. WebKDD/SNA-KDD ’07. San Jose, California: ACM, 2007, pp. 56–65. ISBN: 978-1-59593-848-0. DOI: 10.1145/1348549.1348556. URL: http://doi.acm.org/10.1145/1348549.1348556.

[20] David Ketralnis. *She who entangles men*. URL: http://blog.reddit.com/2010/03/she-who-entangles-men.html (visited on 05/20/2013).

[21] Avinash Lakshman and Prashant Malik. “Cassandra: a decentralized structured storage system”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[22] Mike Loukides. *The NoSQL movement*. URL: http://strata.oreilly.com/2012/02/nosql-non-relational-database.html (visited on 05/20/2013).

[23] Om Malik. *Why Digg digs Cassandra*. URL: http://gigaom.com/2010/03/11/digg-cassandara/ (visited on 05/20/2013).

[24] Mark Mardell. *A market-moving fake tweet and Twitter’s trust issue*. URL: http://www.bbc.co.uk/news/world-us-canada-22283209 (visited on 04/23/2013).

[25] Erica Naone. *What Twitter Learns from All Those Tweets*. URL: http://www.technologyreview.com/view/420968/what-twitter-learns-from-all-those-tweets/ (visited on 04/23/2013).

[26] Timothy Perfitt and Burkhard Englert. “Megaphone: Fault Tolerant, Scalable, and Trustworthy P2P Microblogging”. In: *Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*. ICIW ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 469–477. ISBN: 978-0-7695-4022-1. DOI: 10.1109/ICIW.2010.77. URL: http://dx.doi.org/10.1109/ICIW.2010.77.

[27] Alex Popescu and Ryan King. *Cassandra @ Twitter: An Interview with Ryan King*. URL: http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king (visited on 05/20/2013).

[28] Pierre St Juste et al. “Enabling decentralized microblogging through P2PVPNs”. In: *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*. IEEE. 2013, pp. 323–328.

[29] Tianyin Xu et al. “Cuckoo: towards decentralized, socio-aware online microblogging services and data measurements”. In: *Proceedings of the 2nd ACM International Workshop on Hot Topics in Planet-scale Measurement*. HotPlanet ’10. San Francisco, California: ACM, 2010, 4:1–4:6. ISBN: 978-1-4503-0177-0. DOI: 10.1145/1834616.1834622. URL: http://doi.acm.org/10.1145/1834616.1834622.

[30] Rob Young. *MySQL 5.6: What’s New in Performance, Scalability, Availability*. URL: https://blogs.oracle.com/MySQL/entry/mysql_5_6_is_a (visited on 05/22/2013).