Social-Data Storage-Systems

Nicolas Ruflin
University of Basel
Schanzenstrasse 46
4001 Basel
Switzerland
n.ruflin@stud.unibas.ch

Helmar Burkhart
University of Basel
Schanzenstrasse 46
4001 Basel
Switzerland
helmar.burkhart@unibas.ch

Sven Rizzotti
University of Basel
Schanzenstrasse 46
4001 Basel
Switzerland
sven.rizzotti@unibas.ch

ABSTRACT

The amount of social data produced by a wide variety of social platforms grows every day. Storing and querying this huge amount of data in almost real time presents a challenge to storage systems in order to scale up to hundreds or thousands of nodes. Also the graph structure and the diverse and changing structure of every single node in social data has to be handled by these systems. In this paper, we describe five storage system types on the basis of eight current open source storage system solutions in order to analyze their application potential.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Distributed data structures; H.3.4 [**Information storage and retrieval**]: Information networks

General Terms

Experimentation, Measurement

Keywords

Column store, document store, graph db, key-value store, NoSQL, rdbms, social data, storage system

1. INTRODUCTION

With the proliferation of web 2.0 applications and social platforms, storage systems were faced with new challenges. Google stores and analyzes as many web pages and links as possible in order to provide relevant search results. In 2008, it already passed the mark of more than 1 trillion unique URLs. Twitter already stores more than 155 million tweets per day. Other kinds of so called big data are nowadays produced by location services (mobile phones) and sensor networks in general. These kinds of services are likely to grow massively in the next years [7]. The large amount of data makes it impossible to store and analyze the data with a single machine or even a single cluster. Instead, storage and processing has to be distributed. Due to the need for new storage solutions, Google created BigTable [3] based on Google File System (GFS) and Amazon wrote the Dynamo paper [4], which Riak and other products are based on.

Especially the structure of social data presents a challenge to storage systems. The structure of social data is diverse, differs from use case to use case and changes over time [1]. Not only can the overall network structure and community structure change, but also the structure of every single node can change over time. New entries and data types are necessary when new features are added to the system. Data usage requires parallel processing. Also small companies face the challenge of the big amount of data that is produced by their users and by social interactions. This need lead to a new community, known as NoSQL (Not only SQL) and created different types of new storage systems. Most are based on the ideas of Dynamo (Riak) or BigTable (HBase) or a combination of both (Cassandra).

In this paper, five different storage system types are analyzed: RDBMS, key-value store, column store, document store, and graph databases. Eight different implementations were chosen to analyze these five types of storage systems: MySQL, Redis, Riak, MongoDB, CouchDB, HBase, Cassandra, and Neo4j. The selection of the storage systems is based on their popularity, use cases and development status. We used a real data set from the social content collaboration platform useKit [15] to evaluate the storage systems. This data set consists of around 50'000 nodes, 100'000 edges and has three different node types: User, content and context. All node types can be connected to each other by edges to form an undirected graph. This data set was modeled for all storage systems to better understand how social data can be stored and queried.

2. STORAGE SYSTEMS OVERVIEW

Every kind of system has its use case and has to make tradeoffs at some point. Some of these tradeoffs are shown by the CAP theorem [6]. There are systems that are optimized for massive *writes*, others for massive *reads*. Most of these NoSQL solutions are still in an early development phase, but they evolve fast and already found adoption in different types of applications. The data models of the different solutions are diverse and use-case optimized. Table 1 presents a short overview of the evaluated storage systems showing the data model and query language supported by the system. As this can change, the version of the evaluated system is shown.

DBSocial '11, June 12, Athens, Greece

	Data Model	Query Language	Version
MySQL	Relational	SQL	5.1
Redis	Key-value	-	2.0
Riak	Key-value	MapReduce	0.12
MongoDB	Document	SQL, MapReduce	1.6.5
CouchDB	Document	MapReduce	1.0.1
Cassandra	Column	Range, MapReduce	0.6.2
HBase	Column	Range, MapReduce	0.20.6
Neo4j	Graph	SPARQL, internal	1.2

Table 1: Overview of storage systems evaluated

Most of the new storage systems are optimized to run on several nodes for better performance, implement fault tolerance and are accessible by a wide variety of programming languages. They offer interfaces like Representational State Transfer (REST) [5] or Thrift [14] that allow access based on the standard HTTP Protocol. The communication language can often be chosen by the client and support standards such as XML or JSON.

One goal of NoSQL solutions is massive scaling on demand. Thus, data can be distributed on different computers through the deployment of additional nodes. The main challenge is to efficiently distribute the data on the different machines and also to retrieve the data in an efficient way. Generally, two different techniques are used: Sharding or distributed hash tables (DHT) [2].

2.1 MySQL

MySQL is a very popular relational database. It is used in a wide variety of projects and has proven to be stable. Because of its simple installation and setup procedure, it is used by small companies as well as large production environments such as Twitter where it is used in the way of a key-value store. MySQL supports transactions and relations (foreign keys) depending on the backend.

Replication allows to scale the MySQL database to other machines. Especially *reads* can be scaled with the master-slave architecture. All *writes* go to the master and are replicated to the slaves which serve all the *reads*. Another common technique to scale the MySQL database is horizontal sharding. Sharding can be done based on different tables or by splitting up the data of one table based on row keys. The splitting up of the data and sending it to the right server has to be done on the client side. One disadvantage of sharding is that cross-server queries are expensive.

2.2 Redis

Redis is a key-value store that allows to store sets and collections in addition to simple key-value stores. Redis is written in C and is optimized to keep all values in memory similar to memcached but frequently writes the data to its physical storage. Redis can be used in an append-only mode, which means every change is directly written to disk. Unfortunately, this leads to much lower performance.

In addition to the standard key-value features, Redis supports lists, ordered lists and sets which can be modified with atomic operations. Retrieving data objects stored in Redis is done with a simple key request or one of the special operations that exist for lists and sets as range queries or atomic push/pop operations. To scale *reads*, master-slave replication is supported. To distribute *writes* to several server instances has to be done on the client side through sharding, for example based on key hashes. For simple interaction, Redis offers a shell client.

2.3 Riak

Riak is a Dynamo-inspired key-value store and uses Bitcask [9] as default storage backend. The number of nodes that are necessary for successful reading or writing can be chosen for every request. The programmer can decide whether consistency or availability should be chosen from the CAP theorem. All functionality in Riak can be accessed through a REST interface.

Riak was built as a distributed system from ground up and uses Erlang as the programming language. The data is stored on the Riak nodes based on distributed hash tables (DHT). Depending on the values set in a request, data is read or written from / to several nodes to guarantee consistency or eventual consistency. For internal synchronization between the nodes and different versions of values for the same key, vector clocks [12] are used. All nodes in a Riak cluster are equal and new ones can dynamically join or leave a cluster, which makes it horizontally scalable. There is no dedicated root node.

The Riak storage model supports buckets, which are similar to databases in MySQL, and inside a bucket keys with values. Values can be formatted as JSON arrays or objects. In addition, Riak supports links between two keys, which simplifies traversal requests. Values in Riak can be retrieved by a simple *get* for the specific key. As query language, Riak offers MapReduce where link walking can also be used internally. MapReduce queries can be written in Javascript or Erlang and return JSON objects.

2.4 MongoDB

MongoDB is a document store written in C++. Document data is stored in a binary JSON format, called BSON. Several similar documents are stored together as collections, which are comparable to databases in MySQL. MongoDB allows to define indices based on specific fields in documents. Ad-hoc queries can be used to query and retrieve data that is also based on these indices in order to make querying more efficient. Queries are created as BSON objects and the queries are similar to SQL queries. These kinds of queries are only available for queries which affect only one node. In addition, MapReduce queries and atomic operations on single fields inside documents are supported. This differs from CouchDB where atomic operations are only possible on complete documents. MongoDB follows the goal to be as simple as possible and wants to be the MySQL of NoSQL databases. There is a wide variety of clients for almost every programming language which leverage the REST interface of MongoDB or directly connect to the socket.

Horizontal scaling with MongoDB is done through sharding based on document keys. To have redundancy and to make MongoDB failover save, asynchronous replication is supported. A master-slave replications can be done where only the master is able to accept *writes* and the slaves are read-only replicas.

2.5 CouchDB

CouchDB is a document store written in Erlang and provides a RESTful API to access the CouchDB functionality. ACIDity is provided on a document level. CouchDB uses Multi-Version Concurrency Control (MVCC) instead

of locks to manage concurrent access to the database and documents. This means that every document in CouchDB is versioned. Conflicts can be resolved automatically by the database. The database file on disk is always in a consistent state. CouchDB is already used in research for data management and workflow management at CERN [10].

To query data, CouchDB offers so-called views. There are two types of views: Temporary views which are only loaded into memory and permanent views that are stored on disk and loaded on startup. Views are MapReduce functions written in JavaScript that process the stored data and return the results. A query language similar to SQL and indices on specific fields are not supported, but indices are automatically created for the view results. The results of views are only once completely calculated and then updated every time data is added or updated, which makes the views more efficient. The generated views can be distributed to several CouchDB nodes, which allows for distributed queries.

Scaling *reads* in CouchDB is done through asynchronous incremental replication. For *writes*, CouchDB started to support sharding of data as it is already provided by the CouchDB Lounge extension, but this is still under development. Partitioning data should be supported in the future.

2.6 HBase

HBase is a column storage based on the Hadoop Distributed File System (HDFS) [16] created with the idea of BigTable [3] in mind. HBase is strongly consistent and picks consistency and partition tolerance from the CAP theorem [6]. HBase is – like Hadoop – completely written in Java. To access HBase, the Thrift [14] interface is used, but also REST calls are supported. Indices are automatically created for all row keys and are sorted lexicographically. Internally, B-Trees are used for indices based on which sorting and range queries are made. HBase supports transactional calls with strong consistency in comparison to Cassandra which is based on MVCC. Still, HBase supports versioning and conflict resolution.

The HBase data model is based on the BigTable data model [3]. HBase supports range queries or can be used as key-value store. For data processing and more complex queries of the complete data set, MapReduce queries are supported, which are executed in Hadoop. Scaling HBase can be done by distributing the data to several nodes. HBase clusters have a so-called name node which manages the task and data distribution. There can be a single point of failure if the name node is not redundant. Managing and coordinating HBase data and name nodes is often done with Zookeeper [8], a high-performance coordination service. If HBase is used in combination with Hadoop for the data processing of an additional node, a so-called tasktracker is necessary to distribute tasks to the clients.

2.7 Cassandra

Cassandra is a column-store and a hybrid between BigTable and Dynamo. It uses a data model similar to BigTable and is based on the Dynamo infrastructure. To access Cassandra, the Thrift framework is used. Other APIs such as a REST API are under development.

In addition to the HBase data model, Cassandra supports super columns which are similar to columns but have columns as values. This allows one more data structure nesting level. Data in Cassandra can be accessed by simple key requests. In addition, range queries are supported based on row keys as described before in HBase. Data processing is done through a Hadoop adapter.

Replication of data sent to Cassandra is done based on eventual-consistency, which puts it on the CAP Theorem triangle on the availability and partitioning side, as opposed to HBase, which is on the consistency side and partitioning side. Cassandra is optimized for lots of nodes in one or several data centers that are connected with fibre channel cables and have low latency. This is typical for Dynamo implementations. Concurrent access to the same data is managed with weak concurrency (MVCC) as seen before in CouchDB.

Cassandra is able to scale *writes* horizontally by adding nodes. Distribution on nodes is based on hash tables as described in the Dynamo paper. Nodes can dynamically be added to the cluster which automatically redistributes the data. Every node in a Cassandra cluster is the same, there is no root node.

2.8 Neo4j

Neo4j is a graph database, stored on disk that is fully transactional. Neo4j is written in Java and offers an object-oriented API to store and retrieve data. The data model consists of nodes and edges as known from graph structure. The data storage is optimized for graph traversal. Neo4j can also be used as triple store for RDF data, but it is not optimized for such usage. Neo4j is accessed directly through its Java API. A REST API is under development.

Queries for Neo4j are created directly through its API, which allows complex queries. In addition, SPARQL [11] and Gremlin [13] are supported as query languages. Neo4j offers a wide variety of additional components such as the graph algo component which implements graph algorithms such as Dijkstra and makes it possible to directly use these algorithms in the query code.

To scale Neo4j, sharding is used. This allows to scale *reads* and *writes*, but the time for graph traversal from one node to the next is not necessarily constant because the two nodes can be on two different machines, which requires additional communication.

3. EVALUATION

These eight analyzed systems differ in the structure that data is stored, in the query languages and in how data can be processed. There is no clear result as to which system is the best; it depends on the use case. For the need of processing whole data sets, storage systems which directly implement processing based on MapReduce or other processing languages have a clear advantage. Systems with different indices should be preferred if querying for single nodes with specific attributes is necessary.

3.1 Data Model and Scalability

There are different social data models. Link networks which are analyzed by Google are content-centered and directed graphs. The user relations in Facebook form an undirected graph which is user-centered. useKit with its three different node types forms an undirected graph which is context-centered. The main challenge for all storage systems is how the graph data has to be modeled so querying and processing is as efficient as possible.

The eight different storage system solutions can be grouped

into five different general storage types, but also inside these five storage types, there are considerable differences as to how data has to be modeled and stored. The comparison of Redis and Riak shows two key-value stores with completely different implementations. The data model of a storage system is often directly related to how a storage system can scale *reads*, *writes* and queries. In general it can be said that the less structured the data model is, the simpler it is to scale to large data sets. The following figure 1 compares data size scalability with the data model structure. There is a smooth transition between the different data models. The exact position depends on the concrete implementation of the storage system.

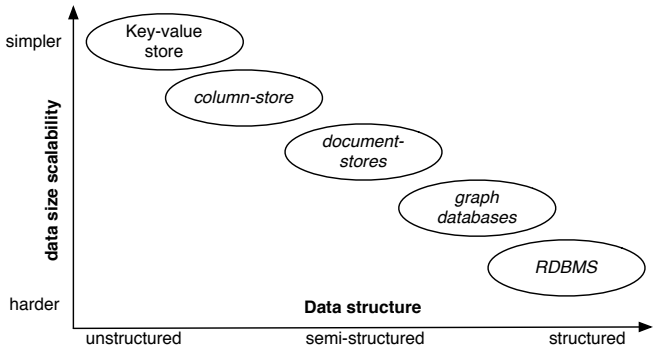

Figure 1: Data size scalability vs data structure

For social data, storage systems with semi-structured or unstructured data models have a clear advantage because of their possibility to scale and also because social data is often only semi-structured and all information for one node can be stored in one location. Column stores are a good fit for much social data because column stores are in this paper considered to be a combination of key-value stores and document stores with tradeoffs. Column stores are almost as scalable as simple key-value stores but are able to store semi-structured data with simple indices, which allows to do simple queries such as range queries. Also, the support for processing the data with MapReduce is important. This does not mean that column stores are the solution for social data. Every implementation is different and has different goals. It has therefore to be evaluated in detail what is needed and where some tradeoffs have to be made. It is important to know in advance what the requirements will be and what kinds of queries have to be used in order to choose the right system.

MySQL with its normalized data model, which is typical for RDBMS, allows to store data in a compact way. MySQL offers good capabilities for querying a specific node in one table based on a row key. One main problem with MySQL and social data are the n to n relations, typical for graph data sets, that exist and that lead to extensive *joins* over multiple tables and server instances in large data sets. To alter the structure in big data sets can take a long time, which makes the systems inflexible. Furthermore, social platforms often focus on *reads* and not on *writes*. *Writes* are cheap in MySQL, because only one table is affected and an entry has to be added to the index tree. *Reads* on the other hand can require queries for lots of keys and joining tables. Nevertheless, MySQL is used in different social platforms. Currently, Facebook and Twitter are probably the most famous ones. The problem is partially solved by Facebook by adding a cache layer in front of MySQL to aggregate different cache results which allows to respond to complex queries by the cache instead of MySQL.

As an enhanced key-value store which supports lists, sets and range queries, Redis requires to model the data differently from MySQL. Modeling data for Redis or key-value stores in general often means creating keys that describe the value stored by concatenating strings. It is important that these keys can be recreated for data retrieval. The same data often has to be stored under several keys because no join queries or queries in general are available. This can lead to the problem that if data entries are updated, they have to be updated in different locations. Redis is not a typical match for social data as it does not support any data processing.

Riak, which is also a key-value store, does not support lists or sets but has a native support for JSON documents. This makes it possible to use Riak similarly to a document store. Less complex key constructs have to be used for Riak than Redis because lists and additional information can directly be stored inside the value of every node. This allows a flexible data model because no indices have to be changed or recreated when the data model changes. Riak supports links between the keys. This makes Riak a good fit for social data especially because of its scalability possibilities and its support for MapReduce queries to process large data sets in a distributed manner.

The two document stores MongoDB and CouchDB use almost the same data model. For both systems, the data can be modeled in JSON/BSON. This makes it possible to model semi-structured data and every node with all its information. With their flexible data model and the support for MapReduce queries, both are a good fit for social data, but have some scalability issues because scaling the data set is based on sharding, which can be complex for large data sets and has to be done on the client side.

HBase and Cassandra as column stores allow to model the data in a semi-structured way. In contrast to document stores, column families for both systems have to be predefined, which makes these systems less flexible for data modeling than document stores. Cassandra, with its support for super columns, allows to store data in an even more structured way and offers more specific queries than HBase. During modeling data for column stores it is important to know in advance what the queries will be so that the data is modeled the way that it can most efficiently be retrieved by queries. Column stores fit social data well because of their semi-structure data model and its ability to scale to large data sets. Both HBase and Cassandra are a good match for social data because they allow basic range queries based on (super) column families and also processing of data based on MapReduce with Hadoop is already supported by the system.

Neo4j has the graph as a storage structure. This makes it obvious to also model the data set as a graph. As social data can normally be represented as a graph, no remodeling is necessary to store nodes and edges. This makes modeling and storing data in graph databases intuitive. Concerning the data model, Neo4j fits social data perfectly. One of the main reasons why graph databases are not largely adopted in social networks is because of their scalability issues and

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

Table 2: Query types supported by storage systems

because most queries are not graph traversal queries. Most queries in social networks are queries for a single node with all its additional data or range queries. Neo4j does not offer special indices for these kinds of queries which are not graph traversal queries for which Neo4j is optimized. The graph storage structure also makes scaling a difficult task. This makes Neo4j a good solution to store social data, but not to retrieve social data in a typical way.

3.2 Queries and Data Processing

The storage systems have different capabilities as to how to query and process data. Data processing is crucial to retrieve aggregated and relevant data from large data sets. In general, four different query types are supported, as can be seen in table 2 where + means the specific query type is supported and − means that is is not supported.

Only the systems that support MapReduce allow large-scale data processing. For all the other systems (MySQL, Redis, Neo4j) the processing has to be done on the client side, hence in the memory. This limits the size of the processable data. Hadoop is currently one of the most used data processing systems and offers adapters for various systems to allow data processing for all kinds of storage systems based on MapReduce.

4. RESULTS

Social data has irregular structures and the storage system requirements differ from use case to use case. Requirements such as data size, consistency, availability or how the data has to be queried, processed or searched have to be known in advance in order to make a proper decision. Three different approaches were used to store the graph nodes and edges depending on the storage system. In RDBMS systems, the nodes are stored in one table and an additional table is used to store the edges (relations). In key-value stores, document stores and column stores, the node information is stored under one key with all the edges as a list directly inside the node – except for Redis, where the list feature was used. Graph databases allow to store the graph data as it is.

A typical query for social data is to retrieve a node with all its additional data. In our useKit data set this could be a user node with its username and the number of content and context items. In MySQL, there are three queries required to resolve this task. One query to fetch the row with all the user data and two additional queries to fetch all the content and context ids. The same applies to Neo4j and Redis, where the user data is stored under a single key and the two adjacency lists are stored as edges in Neo4j or as two lists in Redis. For the column stores HBase and Cassandra document stores MongoDB, CouchDB and Riak, where the data is stored like in a document store, only one key request is needed. All data with all adjacency lists are stored under a single key. If more additional data was added to the user node, like all relations to users, additional queries would be needed in MySQL, Redis and Neo4j. In contrast, the other storage systems would still only need a single request as also this additional data is written into the same document. This shows a clear advantage of document and column stores over RDBMS and graph databases for social data.

MySQL has proven to be stable and reliable, which makes it a good fit for all relational data where transactions or other RDBMS features are needed. Setting up MySQL is simple and relational databases are still best known by most developers, which makes them a first pick candidate for most projects.

Redis is a great addition to other storage systems or as a caching solution. It stores simple key-value data and lists data efficiently and fast. The list feature also allows range queries and atomic operations. The scalability is limited because for good performance the whole data set should fit into memory. Also, automatic sharding is not yet supported.

As a dynamo key-value store implementation, Riak makes it possible to scale large data sets with no single point of failure and automatically scales by just adding nodes. Its implemented support of MapReduce is a necessary addition to process and query data. To store data redundantly in Riak, it is necessary to run Riak, as well as all similarly distributed systems, on more than one node. The support for JSON documents and links, which allows link walking, is a great addition for social data. The main disadvantage of Riak is that no range queries are possible because the data is distributed based on DHT. This problem is partially resolved in version 0.13 of Riak as Lucene is directly integrated into Riak and supports indices and search queries.

MongoDB and CouchDB are two similar document stores. The data model allows to add almost any kind of data to the storage system. MongoDB offers a query language similar to SQL and offers indices that can be set on specific keys. This allows a simple entry point for users that are new to NoSQL solutions. Both systems offer MapReduce queries. CouchDB offers more enhanced MapReduce queries because all queries are based on MapReduce and the system automatically creates indices based on these so-called views. CouchDB is based on the eventual consistency model and MVCC, whereas MongoDB uses strong consistency. Both systems lack the possibility to automatically horizontally scale *writes*. This makes both systems a good fit for social data sets which can be stored on a few machines.

HBase and Cassandra with the column model as storage structure offer a combination of key-value store and document store. Because of its consistency, HBase is better in high read rates and Cassandra in high write rates because of its eventual consistency model. More and more companies pick HBase as part of their storage solution. This can be for pure processing in combination with Hadoop or – in the case of Facebook Mail – as storage system. The community around HBase is growing and has some large committers such as Yahoo or Facebook.

Graph databases have not yet found large adoption in the area of social data, even though their data model fits social data really well. The main constraints are the scalability and processing issues that exist with current graph databases such as Neo4j. This could change in the future.

5. CONCLUSION

The evaluation of storage systems is difficult due to their fast development. New features are added, the speed of *reads*, *writes* and queries are improved and new promising solutions appear on the market which can make evaluation data worthless. The problem of finding the right storage system for a social data set is still tricky. There is no general solution for social data because the structure of every social data set is different and also the requirements of the different social networks differ. In general, it can be said that the problem is not solved by just storing the data efficiently. Retrieving, querying and processing are as important.

Graph databases such as Neo4j are adequate for storing graphs and for representing social data but have constraints in querying and processing social data because they are optimized for graph traversal queries, which are rare in social networks. Also, the scale of social data poses a problem for Neo4j.

The main problem with RDBMS or, more precisely, with MySQL is that social data is semi-structured. The data structure in RDBMS therefore has to change for new appearing data types. JOIN queries over large data sets are one of the main constraints of using RDBMS for social data. Furthermore, RDBMS are not able to process complete data sets which is often necessary for social data. Research is conducted in the area of RDBMS to make it more scalable and VoltDB is one resulting storage system.

Key-value stores are easy to scale and can handle large data sets. The main constraints are their missing capabilities for querying and data processing. But more and more solutions such as Riak offer implemented processing systems which directly access the data that is stored in the key-value store in order to overcome this limitation.

Column and document stores are similar in their basic structure. In column stores, there are more constraints for the data structure and the nesting level is limited. It is easier to model data for document stores but because of its semi-predefined data model, column stores are easier to scale because the storage system can take advantage of the data structure in advance. Column stores only allow range queries based on keys. This requires to model the data in a way that the range queries based on the sorted keys return the expected result. Column stores seem to be one of the best fits for social data at the moment because of their capabilities to scale and to query the semi-structured data set. Also, they are good in processing the data. One main disadvantage of NoSQL solutions such as key-value stores, column stores and document stores is that the several copies of one entity exist in the storage system and the data set has to be kept clean by the programmer instead of the storage system. However, these are tradeoffs that have to be made in order to make it more scalable.

Even though there is already a fair number of storage systems, more research is necessary to create more scalable systems. Especially the combination of storage systems with processing should be researched in more detail in order to provide more efficient processing methods.

6. REFERENCES

[1] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *Proc. VLDB Endow.*, 4(3):173–184, December 2010.

[2] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 205–218, Berkeley, CA, USA, Jan 2007. USENIX Association.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, Jan 2007. ACM.

[5] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, Jan 2000.

[6] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, Jan 2002.

[7] S. Higginbotham. Sensor networks top social networks for big data. `http://gigaom.com/cloud/sensor-networks-top-social-networks-for-big-data-2`, Sept. 2010.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, Berkeley, CA, USA, Jan 2010. USENIX Association.

[9] D. S. Justin Sheehy. Bitcask. a log-structured hash table for fast key/value data. Technical report, Basho Technologies, 04 2010.

[10] S. Metson. Cern: A case study. `http://www.couch.io/case-study-cern`.

[11] E. Prud'Hommeaux and A. Seaborne. Sparql query language for rdf. `http://www.w3.org/TR/rdf-sparql-query/`, 2008.

[12] M. Raynal and M. Singhal. Logical time: Capturing causality in distributed systems. *Computer*, Jan 1996.

[13] M. A. Rodriguez. Gremlin. `https://github.com/tinkerpop/gremlin`, 2011.

[14] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook*, Jan 2007.

[15] useKit. Share and discuss. `http://useKit.com`, march 2011.

[16] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2 edition, Jan 2010.