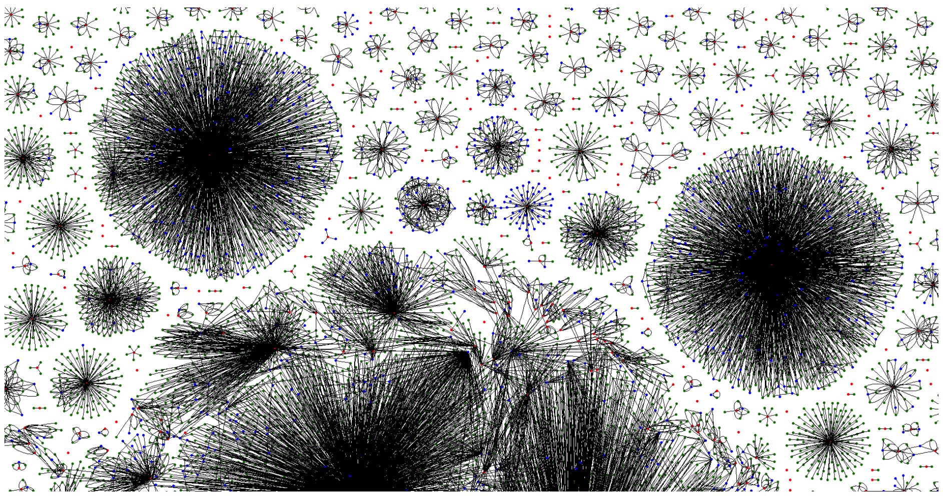
Storing and Analyzing Social Data



University of Basel
Computer Science Department

Master Thesis

Author:
Nicolas Ruflin
Breisacherstrasse 26
4057 Basel
+41 76 435 58 67
n.ruflin@stud.unibas.ch

Thesis Advisors:
Prof. Dr. Helmar Burkhart
Dr. Sven Rizzotti

December 31, 2010

Contents

Abstract

Social platforms such as Facebook and Twitter have been growing exponentially in the last few years. As a result of this growth, the amount of social data increased enormously. The need for storing and analyzing social data became crucial. New storage solutions – also called NoSQL – were therefore created to fulfill this need. This thesis will analyze the structure of social data and give an overview of currently used storage systems and their respective advantages and disadvantages for differently structured social data. Thus, the main goal of this thesis is to find out the structure of social data and to identify which types of storage systems are suitable for storing and processing social data. Based on concrete implementations of the different storage systems it is analyzed which solutions fit which type of data and how the data can be processed and analyzed in the respective system. A focus lies on simple analyzing methods such as the degree centrality and simplified PageRank calculations.

Target reader group: Designers and developers of social networks.

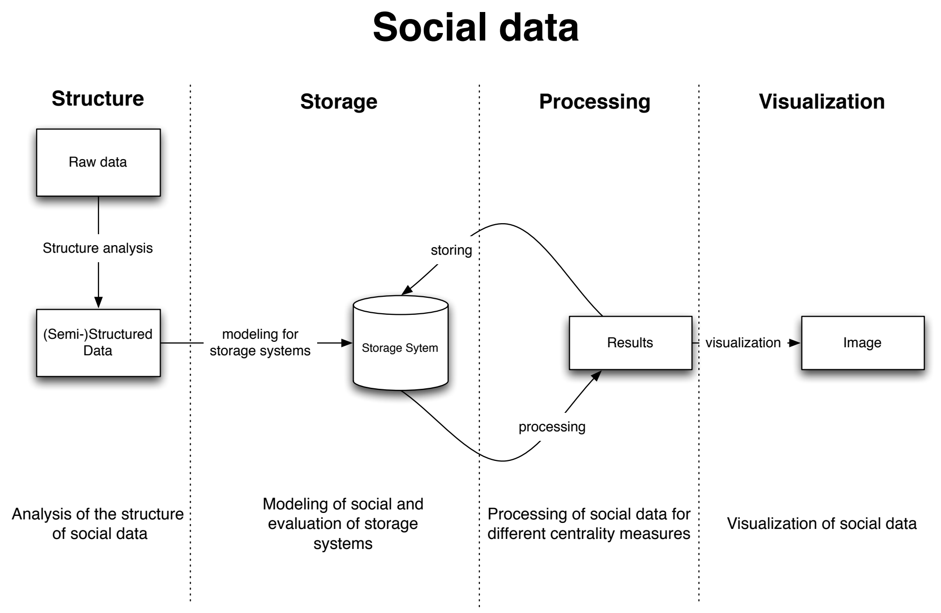
1 Introduction

Today's omnipresence of social data is to some extent associated with the development of the Internet, which is the result of the the Advanced Research Projects Agency Network (ARPANET)[1]. The ARPANET was a computer network which came to be the foundation of the global internet as we know it today. It was created in the United States and launched with a message – or the first form of social data in the internet – that was sent via the ARPANET on 29 October 1969.

Although social data does not exclusively come from the Internet – also telephones, sensors or mobile devices provide social data – the predominant part of social data is in fact internet based. It started with the exchange of information via e-mail. Soon, e-mails could be sent to several people at once. The next steps were mailing lists, fora and blogs with commenting functions. Later, social networks followed. It is then that the amount of data became a mass phenomenon. With social platforms such as Facebook or Twitter, the upswing of social data was as strong that today, it has become a standard medium. As a result of the exponential increase in social data, the need for storing and analyzing it has become a challenge that has yet to be solved.

The goal of this thesis is to analyze the structure of social data and to give an overview of already existing algorithms to analyze and process social data, implemented on different storage systems and based on a specific data set. In addition, the data set will be visualized. As opposed to more research-oriented papers, this thesis combines four different research areas – structure, storage, processing and visualization of social data, as is portrayed in figure 1 – and implements them in a real-world example. Although each of these four areas has been researched extensively and relevant findings have been made, the findings of each area have rarely been brought together to see the overall picture. Thus, researchers in one area only seldomly have access to discoveries that were made in another area. This is what makes the study of social data challenging and fascinating at the same time, since social data is an interdisciplinary field and touches upon all four areas.

To describe the results of the already existing research and to set a general framework, the topic will first be broken down into the four areas of figure 1. In a second step, the results of this thesis will be discussed in a more globalized consolidation.



Social data
Structure
Storage
Processing
Visualization
Raw data
Structure analysis
(Semi-)Structured Data
modeling for storage systems
Storage Sytem
storing
Results
processing
visualization
Image
Analysis of the structure of social data
Modeling of social and evaluation of storage systems
Processing of social data for different centrality measures
Visualization of social data

Figure 1: Four areas of social data

The structure of social data varies greatly for single nodes and edges in a network but also the overall network structure varies depending on the kind of social data. Twitter only has users and tweets as node types and is a directed graph whereas Facebook has users, photos, comments and others and is an undirected network. The common denominator of social data is its graph structure: Data points are always connected with other data points through edges[2]. In the research area of semantic web [3], the structure of network data and how to describe [4] and query [5][6] it in a structured way was studied in detail. The overall structure of social data with its communities and the internal interaction is studied in sociology and graph theory, the communication inside the networks is analyzed in linguistics [7]. In this thesis, the general structure of social data will be analyzed. As a specific example for social data the useKit data set – which will be described below – is analyzed, stored, processed and visualized.

The amount of data produced in social networks grows every day. This requires storage systems that are capable of storing huge amounts of data. The data of a single social network already exceeds the storage capacities of complete data centers. Thus, storage systems are required to fulfill certain criteria when it comes to social data. The structure of social data is diverse and can differ from use case to use case and it can change over time [8]. To work with the stored data, new processing systems are necessary that can process the data in parallel. In this thesis, five different storage system types are analyzed by applying them to the useKit data set in order to evaluate how suitable they are for storing social data. The five storage system types are: key-value stores, column stores, document stores, graph databases and RDBMS.

Processing the stored data is crucial in order to extract relevant data out of the stored data [9]. There are different methods of how data sets can be queried or processed. Parts of the research community use implementations for processing social data which are based on supercomputers such as the Cray [10] [11]. Others focus on providing processing methods that can scale up to thousands of commodity machines to provide the same processing power [12]. The commodity hardware approach is also more adequate for social network companies such as Facebook, Twitter and LinkedIn that have to process large amounts of data. One of the de facto standards for processing large amounts of data on commodity hardware is currently the MapReduce model [13], which is also extensively used in this thesis. However, as the data changes rapidly and constantly, there is a need for improved and more efficient methods. In this thesis, the emphasis lies on processing the nodes for centrality measures. Centrality measures of a node in a network can describe the relevance, influence or power of a node in a network. Two centrality measures – degree centrality and PageRank – are implemented in every storage system with the specific capabilities of every single system to process the data set.

Visualizing information was already used a long time ago by the Greeks and Romans to communicate and to show concrete ideas. One of the main goals of information visualization is the representation of the data in a way that makes it visually appealing and understandable by the consumer [14]. Initially, data visualization in computer science focused on the research area of computer graphics. From there, the visualization of information through computers and algorithms has made its ways in almost every other research area. Nevertheless, visualization is still in its infancy. Basic algorithms based on physical laws are used to create the visual representation [15]. But often, human interaction is still necessary, because the human feeling of how a graph representation is comfortable is not only based on mathematical rules [16]. With the growth of the internet and social data, visualization is used to visualize the relations or the structure of these networks, but also in other ways such as to show how the swine flue spreads over the world, which was done in 2009 by Google. Data visualization often makes data better understandable and can reveal unknown details such as communities or clusters in a network, but also describe the overall data structure. In this thesis, different methods were evaluated to visualize graphs with more than 100'000 nodes and edges. The useKit data set was visualized in order to better understand the overall structure.

Every section of this thesis is split up into four parts as shown in figure 1: data structure, data processing, data storage and data visualization. In the first part of the thesis, the structure of social data 2.1, especially the structure of the useKit data 3.1 is analyzed. The second part evaluates the different types of storage systems 2.3 in detail based on eight different storage systems 3.3 – MySQL, Redis, Riak, MongoDB, CouchDB, HBase, Cassandra, Neo4j – in order to show the advantages of every single system. The useKit data is modeled for every system and stored inside the system. In the next section, processing methods to calculate degree centrality and PageRank 2.2 for the useKit data are shown for every system 3.3. To better understand the complete structure of the useKit data, the data is visualized in the end 3.4.

2 From Raw Graph Data to Visualization

Graph data is an elementary part of most data sets in research, business, social networks and others. Current computers are good in running sequential algorithms and also store data sequentially. To represent graphs in computer memory often adjacency matrices are used which is efficient as long as the complete graph data set fits into memory. Storing graphs that do not fit onto one machine in an efficient way is still an open problem. Until a few years ago there were only a few graph data sets that had millions of nodes. With the growth of social networks more and more huge graph data sets exist which have to be stored and processed. To store, process and analyze social data it is crucial to understand the structure of social data and how it can be represented.

First, the structure and representations of graph data and social data will be described in detail. In a second step, the different algorithms that are used in this thesis to process and analyze the social data will be described. This will be followed by a part on existing storage and processing systems. Then visualization algorithms and methods are described in detail.

2.1 Graph data – Social data

Large quantities of digital data are created every day by people and computers. As soon as the data consists of data points that are related or there are relations within the data, the respective data can be represented as a graph or network [17]. Examples for such data sets are data sets of chemical systems, biological systems, neural networks, DNA, the world wide web or social data created by social platforms. Social data can be represented as a graph because of the relations between its nodes which often means users are connected to other users.

Social data is produced by users that interact with other users, but also machines can create social data. There are coffee machines that tweet every time someone gets a coffee or mobile phones that automatically send the current location of a user to one or several social platforms. Also different social platforms can be connected to each other to automatically update a person's status on other platforms. Adding a twitter account to your own Facebook profile means that every time a message is posted on Facebook, it is automatically tweeted. Thus, a single input produces several data entries.

The term social graph describes the data in social platforms including its relationships to other nodes. The term is also used by Facebook to describe its own data set. There does not appear to be any formal definition of a social graph. In general, every social network has its own social graph but by having more and more social platforms connected to each other these social graphs also start to overlap. Because there are no standards for social graphs, one main concern is the portability of social data between platforms. Brad Fitzpatrick has posted his thoughts on the social graph in the blog post "Thoughts on the Social Graph" [18] and mentions some interesting points about the owner ship of the social graph and if there should be a standard way to access it. He discusses how the storage of a social graph, which means storing social data, could be decentralized and perhaps also on the client side instead of trusting in one large company that stores and manages all data.

Definition of social data There are different approaches to define social data. In the research area of linguistics, social data is better known under the term computer-mediated communication (CMC) and can be defined as "predominantly text-based human-human interaction mediated by networked computers or mobile telephony" [7]. Because I did not find an exact technical definition of social data, in this thesis social data is defined even broader as data that is produced by individuals or indirectly by machines through interaction by individuals (coffee machine) that is published on a data store (social platform, forum etc.) accessible by other individuals and that allows interaction (comments, response, like etc.). Every data point has at least one connection to another data point. Examples of social data are forum entries, comments on social platforms or tweets, but also data like GPS locations that are sent from mobile devices to social platforms.

One of the simplest forms of social data produced in the internet is probably email. Emails are sent daily from one user to several users to communicate and to exchange data. This started in 1970 with the invention of the internet. The next step in evolution were mailing lists, forums, wikis (Wikipedia), e-learning systems and other types of social software. The produced data was limited to a relatively small number of participants. Around 2006 the so-called Web 2.0 started and platforms like Facebook, Orkut and others started to grow rapidly. The number of users that exchanged social data grew exponentially and new requirements to process this massive amount of data appeared. When data was no longer simply exchanged between 5 or 10 users, data started to be exchanged between hundreds or thousands of nodes and could be access by a lot more because most of the data is public available. Currently, Facebook has more than 500 million users [19]. This means that Facebook is a network or a graph with more than 500 million vertices and billions of edges.

As Stanley Milgram found out back in 1960 with his famous and widely accepted small-world experiments [20], the average distance between two people is 5.9. Therefore, most people are on average connected to everyone else over six nodes. It is therefore likely that these social networks have a similar value of connectability – possibly even lower, since Facebook has a tendency to connect more people than real life, since people can be contacted easily and effortlessly.

2.1.1 Structure of Social Data

There are no predefined data models for the structure of social data. Social data appears in a broad variety of forms and its structure differs from use case to use case. Also, the structure might vary within a use case, and it can evolve over time. Let us take location data as an example. Nowadays, lots of mobile devices included GPS but have different capabilities. Some only send data +-100m accurate, others include also sea height and more precise details. The structure of the data depends on the system that sends the data and on its capabilities. Even though the exact structure of the data that is sent is not known in advance, storage systems have to be capable to store the data. One option is to filter all incoming data and discard all information that is not supported by all systems, but this means loosing relevant data. An other approach is to store all data even though the data structure can be different for every package which poses special requirements on the storage systems.

The structure of social network data is different from web link data. Web link data consists of webpages and the connecting links. It can be represented as a directed graph because every link goes only in one direction. There is a distinct difference between social data and web link data as Bahmani describes: "In social networks, there is no equivalent of a web host, and more generally it is not easy to find a corresponding block structure (even if such a structure actually exists)" [8]. This difference has a direct effect on the storing and retrieving of social data, because similar data cannot be stored together on one machine. This again has a direct affect on data accessing and processing, because calculations are even more distributed and it is hard to minimize communication. Typical for the social data structure is the N to N relations that exist for most nodes, which means every node connects to several nodes, and several other nodes are connected to this node.

Social data can be represented as a graph. In the example of Facebook, every node is a user and the connections between the users are edges. Research has shown that social networks in the real wold and also computer-based social networks such as Facebook are fat tailed networks. Thus, they differ from random graphs [17]. This means that at the tails of the network, there are highly connected networks in comparison to the connectivity to the rest of the network. We see the same phenomenon in the real world: it is highly probable that people who know each other also know each other's friends – they are highly connected – and they know considerably less people in the rest of the world – less connectivity in the network overall.

In social networks, there are often some highly connected nodes. In social networks such as Facebook or Twitter, these are often famous people. Based on the power law, these nodes are exponentially more popular than others nodes. In June 2009, the average user on Twitter had around 126 followers[21]. But there are some users with more than two million followers [22], and the number increases fast.

2.1.2 Representation of Social Data

For every data set there are different methods to represent the data. The different representations can be used for different tasks and every representation has its advantages and disadvantages. It is crucial to find the right data representation for a specific task. In the following, different methods of representing social data are described.

One of the simplest ways of representing a graph is with dots (vertices) and lines (edges)[2]. With simple dots and lines, every graph can be represented and visualized. Thus, also social data can be represented with dots and lines. In social networks, every user is a vertex and every connection to another user is an edge. However, additional parameters for vertices and edges are often desired to also represent specific values of the vertices and describe the edges in more detail. This can be done with a so-called property graph [23].

The DOT Language [24] defines a simple grammar to represent graphs as pure text files that are readable by people and computers. Every line in the file consists of two vertices and the edge in between. The DOT Language allows to set properties for edges and vertices and to define subgraph structure. The created files are used by Graphviz to process and visualize graphs.

Lots of research has been done in the area of semantic web – also called the web of data [25]. The goal of the semantic web is to give computer understandable meaning to nodes and the connections in between various types of nodes. It is possible to assign a semantic meaning to most relations in social networks. This makes it possible to use social data as semantic data and process it by computers. One common way to represent relations in graph data is the Resource Description Framework (RDF) [4], which is based on XML from the semantic web movement created by W3C[1]. RDF is a standard model that interchanges semantic data on the web. Every RDF triple consists of subject, predicate and object. To map these properties to a graph, subject and object are represented as vertices, predicate as an edge. This makes it possible to represent every type of graph. Converting social data into a RDF representation makes social networks searchable by SPARQL. A computer is therefore able to query the data based on constraints, even though the computer does not understand the data. The Friends of a Friend (FOAF)[2] project is another approach that is more focused on representing social data in a standardized way and that has as its goal to make sharing and exchanging information about the connections and activities of people easier. FOAF is based on RDF with predefined properties that can be used to name the edges. Another approach in this area is the Web Ontology Language (OWL) [26]. The overall goal of these representations of social data is to make it readable and searchable by computers.

Another representation of graph data often used in high performance computing in combination with matrix calculations is the representation as an adjacency matrix. These matrices are often sparse matrices, which makes it possible to store data in a compressed form as the compressed sparse row (CSR) format and others[27][28]. Much research has been conducted on the processing and storing of sparse matrices [29][30]. Matrices or multidimensional arrays are a forms of data representation that fit computer memory and processor structures well.

Another approach to represent graph data or social data is the JavaScript Object Notation (JSON)[31]. JSON can represent complex structures in a compact way and is also readable by people. JSON has a loose type system and supports objects and arrays. This allows to represent graph data with subarrays and can be used to represent single nodes and edges with their properties or small networks with an initial node. Because of its compact form and its support in most programming languages, JSON has been becoming one of the standards in interchanging data between different services and often also replaces XML. The JSON format has found adoption in storage systems to store data and query data as in Riak or MongoDB.

For people, the representation of a graph that is most easily understandable is the representation as an image. To create such an image, the graph has to be processed first. Visualizing graph data for people is not only based on pure mathematical and physical, rules but also on the sensation of the viewer [14]. This makes it a complex subject that is researched thoroughly. In contrast, image representation is difficult to understand for computers. Thus, other formats that were mentioned above are used for the internal representation and storage.

2.1.3 Data providers

To store, process and analyze social data sets real world examples with a adequate size are necessary. There are different open data sets mostly created by research or the open source community. Most of these public data sets are web link data sets or similar, real world social data sets are rare.

Some of the first open data repositories where data was analyzed in depth were citation networks [32] and the Usenet [33]. There are also open data repositories such as DBPedia [34], Geonames and others which have already been analyzed in research [35] [36]. The Many

[1]The World Wide Web Consortium (W3C): `http://www.w3.org/`
[2]Friends of a Friend project: `http://www.foaf-project.org/`

Eyes project from IBM [3] is an open project that allows daily users and researchers to upload their own data and to visualize it with the existing tools. The platform is also used by the University of Victoria, Canada, for research. A web link data set that was created for research is the ClueWeb09[4] collection [37]. The goal of this collection was to support research on information retrieval and related human language technologies. The data set consists of more than 1 billion pages in 10 different languages and has a size of 5 TB. It was collected in January and February 2009 and was used in different research projects such as for example the TREC 2009 Web Track[38].

For the analysis in this thesis a data set from a social network is needed. Even if a huge amount of social data is produced every day, the access to social data is protected. Every social network secures its own data to protect the privacy of its users and to make it impossible to give insight to competitors. Most social networks have an API as Facebook with the Open Graph protocol [5] or Google with its social graph API [6] and the Google GData Protocol [7] which allows to access parts of the data. There are two main problems when using such a large amount of social data for research. To access hidden user data, every user has to confirm the access first. Moreover, it is often not possible to store data that is retrieved from social networks locally for more than 24 hours, which prevents people from retrieving a large amount of data for processing purposes. Some research institutes are working together with social network data providers, which allows them to access the data directly.

2.2 Social data graph analysis

Social data graph analysis describes the process of analyzing the data of which a social graph consists of. One of the main goals of analyzing a social graph is to better understand the structure of the graph and find important nodes inside the graph. To find important nodes different algorithms can be used as centrality measures like degree centrality or PageRank. To better understand the complete structure of the graph community identification or visualizing the graph can help. Implementing these centrality measures on different storage systems to analyze the stored graph data is a central part of this thesis.

The main challenge with the analysis of social data is the amount of data that already exists and the data that is constantly produced. The amount of data has exceeded the storage capacities of one single machine a long time ago. Thus, the data needs to be stored in larger clusters. The graphs that have to be analyzed have millions to billions of nodes and edges, which also makes it impossible to process them on a single machine. To analyze only a subset of the existing graph data, key problems are finding communities which are dense collections of vertices, splitting up graphs with coarsening and refinement or local clustering. Different approaches to find communities with parallel computers on peta scale graphs have been described by Bader [39]. For research, there are massively multithreaded and shared memory systems such as the Cray computer that are used for large graph processing [10] [11] [40]. Such systems are rare and expensive and also have scalability limits.

[3] IBM ManyEyes Project page: `http://manyeyes.alphaworks.ibm.com/manyeyes/`

[4] ClubWeb09 data set: `http://boston.lti.cs.cmu.edu/Data/clueweb09/`

[5] Facebook API to include Facebook content in other webpages: `http://opengraphprotocol.org/`

[6] Google API to make public connections useful: `http://code.google.com/apis/socialgraph/`

[7] GData APi: `http://code.google.com/apis/gdata/`

Distributed solutions for storing, processing and analyzing social graph data are needed because many graphs of interest cannot be analyzed on a single machine because of its spanning on millions of vertices and billions of edges [41]. Based on this, lots of companies and researchers moved to distributed solutions to analyze this huge amount of data. Similar problems are tackled in the research area of data warehousing . Companies such as Facebook or Google which have large graph data sets store, process and analyze their data on commodity hardware. The most common processing model used is MapReduce [13]. Analyzing data on distributed systems is not a simple task since related nodes are distributed over different machine. This again leads to a lot of communication. The first approach that was tried was storing related nodes on the same machine to minimize queries over several machines. But as Facebook stated the structure of social data is dynamic and not predictable which leads to the constant restructuring of the system. The conclusion was to randomly distribute the nodes on the machines and not do any preprocessing.

Until october 2010 in most areas there was a clear distinction between instantaneous data querying and the delayed data processing. Data processing on large data sets was done based on MapReduce and retrieving the result could take from several minutes up to days. Whenever new data was added to the data set, the complete data set had to be reprocessed which is typical for the MapReduce paradigm. This was changed by Google in October 2010 through the presentation of their live search index system Caffeine[42] which is based on the scalable infrastructure Pregel [12]. Pregel allows to dynamically add and update nodes in the system and the added or modified nodes can be processed without having to reprocess the whole data set.

2.2.1 Graph Theory – Centrality

Every graph consists of edges and vertices. In this thesis, a graph is defined as $G = (V, E)$ where vertices V stand for the nodes in the graph, and edges E for the links or connections between the nodes. This is the standard definition that is mostly used for social data and other graph data. Edges in a graph can be undirected or directed, depending on the graph.

One of the most common tasks for graphs is to find the shortest path between two nodes, which is also called the geodesic distance between two nodes. There are different algorithms to solve the problem of the shortest path. The Dijkstra algorithm solves the single-source shortest path problem. It has a complexity of $O(V^2)$ for connected graphs, but can be optimized to $O(|E| + |V|log|V|)$ for sparse graphs and a Fibonacci heap implementation. The Bellman-Ford algorithm is similar to the Dijkstra algorithm, but is also able to solve graphs with negative edges and runs in $O(|V||E|)$.

To calculate all shortest paths in a graph the Floyd-Warshall algorithm can be used. The Floyd-Warshall algorithm calculates all shortest paths with a complexity of $O(n^3)$. Because of its exponential complexity it cannot be used for large graphs. Most graphs in the area of social data that are analyzed are sparse graphs. The Johnson algorithm is similar to the Floyd-Warshall but it is more efficient on sparse matrices. It uses the Bellman-Ford algorithm to eliminate negative edges and the Dijkstra algorithm to calculate all shortest paths. Is the Dijkstra algorithm implemented as a Fibonacci heap, all shortest paths can be calculated in $O(V^2logV + VE)$.

Because shortest path algorithms are crucial for graph processing such as centrality calculations, many researchers worked on optimizing these algorithms. Brandes has shown that betweenness centrality [43] on unweighted graphs can be calculated with $O(EV)$ and $O(EV + E^2log(E)$ for weighted graphs [44]. This made it possible to extend centrality calculations to larger graphs.

There are various centrality algorithms for a vertex in a graph. The centrality of a person in a social network describes how relevant the person is for the network and, as a consequence, how much power a person has. However, as Hanneman [43] describes, influence does not necessarily mean power.

In the following, degree centrality and PageRank are described in detail. The implementations of these two will be made for data processing. Other centrality measures such as closeness, betweenness and stress centrality or the more complex Katz centrality, which have been described in detail by Hannemann and Riddle[43] have been extensively researched in recent years. Also parallel algorithms were developed to evaluate centrality indices for massive data sets in parallel [11] [45]. Calculating shortest paths between vertices is crucial for most of these algorithms.

Another centrality approach that was proposed by Lempel and Moran is the Stochastic Approach for Link-Structure Analysis (SALSA) [46]. SALSA is based upon Markov chains and is a stochastic approach for link structure analysis. The authors distinguish between so-called hubs and authorities. The goal of the SALSA-algorithm is to find good authorities, which can be compared to nodes that have a high centrality degree.

2.2.2 Degree Centrality

There are two different types of degree centrality, one defined by Freeman [47] and a newer version defined by Bonacich [48]. The latter also takes into account to which nodes a certain node is connected to. In this thesis, it is assumed that a node, which is connected to lots of other nodes with high degree, probably also has a more central position in the network than a node that is connected to lots of nodes without connections. This is an idea that can also be found in the PageRank algorithm.

The Freeman approach is defined as the sum of all edges connected to a node and differentiates between in and out degree for directed graphs. Every edge can have an edge weight which is the value in the sum for the connected vertex. In unweighted and undirected graphs, the Freeman degree centrality is simply the count of all edges that are connected to a node. For normalization reasons, this sum is divided by the total number of vertices in the network minus one. The complexity is $O(1)$, which makes it simple and efficient to calculate. The equation is shown in in figure 1, where $deg(v)$ is the count of edges that vertex v has. This is divided by the total number of edges n in the network minus one. The result is the degree centrality for vertex v.

$$C_D(v) = \frac{deg(v)}{n-1} \tag{1}$$

In figure 2 a small network graph with 15 nodes and directed but unweighted edges is shown. In this example, node 1 would have an in-degree of 2, node 2 and in-degree of 5 and and out-degree of 1 because it has 5 incoming edges from node 3,4,5,6 and 15 and one outgoing edge to node 1. Nodes 2, 6 and 10 are the nodes with the largest in-degree in this network. What this exactly means can be different in every use case, but it could mean these are the most powerful nodes or the nodes with the most influence.

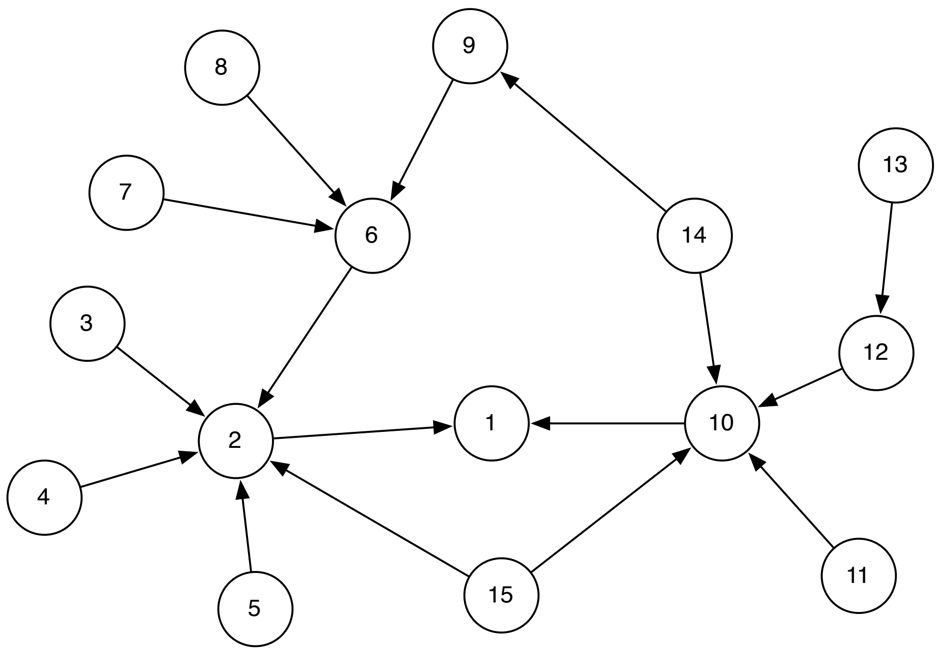


Figure 2: Small network

2.2.3 PageRank

PageRank was developed by Page, Brin et al. [49] [50] and was first used in a large deployment by Google to calculate the relevance of webpages in the global web. The PageRank centrality measure is a variant of the eigenvector centrality [43] and shares its basic ideas with the degree centrality introduced by Bonacich. It gives a more global view on the complete graph to evaluate how central a node is instead of just using the local information of a node for evaluation. PageRank extends the degree centrality that it assumes: A node with many relevant neighbors is also more likely to be relevant than a node with lots of neighbors that only have a few connections. This assumption can also be made in real life. A person that is known by many important people is likely to be important, too. The PageRank of a node describes the likelyhood that a random walk arrives at this node.

The sequential PageRank algorithm is shown in equation 2 where c is the damping factor, N_v the number of outgoing edges from the node v, N the total number of nodes and B_v the set of all nodes that link to v. The damping factor c simulates a frequent jump to a random node. This factor is need to prevent so called sink nodes that do not have any outgoing links. Then the assumption is made, that the user jumps to a random other node. That is the reason that parts of the PageRank value are equally distributed to every node by the right part of the equation. The smaller the damping factor, the more often random jumps happen. Good values for c seem to be between 0.7 and 0.8. The calculation of the PageRank is an iterative process, which means that the previous results of an iteration are always used for the next iteration. Consequently, intermediate results have to be stored. If the damping factor c is set to 1, the PageRank algorithm equates to the eigenvector centrality. The PageRank algorithm converges to a static state.

$$P'(u) = c \sum_{v \in B_v} \frac{P'(v)}{N_v} + \frac{1-c}{N} \tag{2}$$

In figure 2 as described before with the degree centrality, a small network graph is shown. Node 1 in the network had an in-degree of 2, which made it a less central node than 2, 6 or 10. With the PageRank the node 1 gets more important because the importance of neighbor nodes is also taken into account. Node number 1 has two incoming edges from the two central nodes 2 and 10. This makes node number 1 more central then node 6 which has three incoming edges from node 7, 8 and 9. But these three nodes do not have any incoming edges from other important nodes or do not have incoming edges at all.

The personalized PageRank algorithm is a modification of the existing algorithm. Instead of jumping to a random node with a given probability, the jump is always done to the initial node. This creates PageRank values that are more personalized on the focus of the given user. Nodes that are relevant for a user do not need to have the same relevance for others. This makes recommendations and personalized search results possible. Experiments have been done by Twitter in combination with FlockDB[8] to implement a fast and incremental personalized PageRank[8].

2.2.4 Data processing

Data processing plays a key role in all areas where a large amount of digital data is produced and the goal is to understand the data. A quote form Todd Hoff [9] describes that it is not enough to simply process data in order to make sense of it, but also human analysis is necessary:

> "There's just a lot more intelligence required to make sense out of a piece of data than we realize. The reaction taken to this data once it becomes information requires analysis; human analysis."

To process data, a data storage system is needed that allows to load or query the data in an efficient way and is also capable of storing the results that come from the processing system.

All data processed in this thesis has a graph structure. The processing and analyzing of graph structures was already analyzed in detail in the semantic web movement where the graph structure is stored as RDF. Much research has been conducted to find out how RDF data can be stored and queried in an efficient way [51][52][53]. The default query language for these kinds of data sets is SPARQL [5] or one of its enhanced versions [6]. Typical queries are

[8]Distributed and fault tolerant graph db with MySQL as storage engine: `http://github.com/twitter/flockdb`

graph traversal queries and finding nodes with predefined edges in between. An example query could be to find all students that have grades over 5, listen to classical music and like to read books. For this, the focus is on graph traversal. In this thesis, the focus is more on processing complete data sets to identify the structure of the whole network and on calculating centrality indices of every single node.

Efficient processing of large graph data is a challenge in research as well as for many companies. The main challenge is that the complete graph cannot be held in the memory of one machine [41]. In research, one of the approaches is to use massive shared memory systems such as the Cray that can hold the complete graph in memory. Because of the rare availability of such machines, most companies process their data on lots of connected commodity hardware machines. The main challenge is the communication overhead that is produced during processing the graph. Because of the connections between the data nodes that are distributed on different machines communication is necessary as soon as the calculation for one node includes values of other nodes.

New methods to process data at a large scale more efficiently have already appeared, as for example the Pregel [12] system that is used by Google. Pregel is a system for graph processing at large scale. Google uses it interally to keep its search index up-to-date and to deliver almost live search results. Pregel is based on the Google File System (GFS) and supports transactions and functions similar to triggers. In contrast to most systems which are optimized for instantaneous return, the goal of Pregel is not to respond as quickly as possible, but to be consistent. Transactions can take up to several minutes because of locks that are not freed. Before Pregel, the data was processed with MapReduce. Large amounts of data had to be reprocessed every time new nodes and edges were discovered. The new system only updates the affected nodes which makes it much more scalable. To analyze and query the given data, Google developed Dremel [54] which is a read only query system which can distribute the queries to thousands of CPUs and Petabytes of data. There also exist open source variants of this approach like Hama[9], an apache project, or Phoebus[10] that are still in early development and first have to prove scalability to a large amount of data.

It is no longer necessary to build data centers to scale up to this size. Providers such as Amazon or Rackspace offer virtual processing or storage instances that can be rented and have to be paid based on the time the machines are used or based on process cycles. This makes it possible for research to rent a large amount of machines to do some calculations on it and then shut it down without having to spend a large amount of money on super computers or small data centers.

MapReduce One of the common methods for graph processing and data analysis is MapReduce. MapReduce is a programming model based on the idea of functional programming languages using a map and a reduce phase. Dean et al developed MapReduce at Google[13] to handle the large amount of data Google has to process every day. MapReduce can be compared to procedures in old storage system except that it is distributed on multiple machines and can also be thought of as a distributed merge-sort engine. MapReduce is optimized for data parallelism, which is different from task parallelism. In data parallelism, much data with the

[9]Hama project: `http://incubator.apache.org/hama/`
[10]Phoebus project: `http://github.com/xslogic/phoebus`

same task is processed in parallel, but not different tasks are executed on the data in parallel. MapReduce will be used in this thesis for processing the data to calculate PageRank.

The MapReduce programming model can be split up into two phases, the map phase and the reduce phase. The data is loaded from the source, represented in figure 3 as input file, in to the mapping phase. The mapping processes the input data in parallel on several machines and creates the intermediate results which are sent to the reducers by doing per-record computations. Often the number of intermediate results is much larger than the initial input data. Then the reducer processes the intermediate results in parallel on several machines as in the map phase and aggregates the results to create the output. The input and and output for MapReduce can be in general any data source. There are intermediate phases between the map and the reduce phase as combining, partitioning, sorting which are often handled automatically by MapReduce implementations as in Hadoop. To optimize the performance of MapReduce Lin et al propose different optimization of theses phases [41] e.g. to minimize the communication between the different nodes.

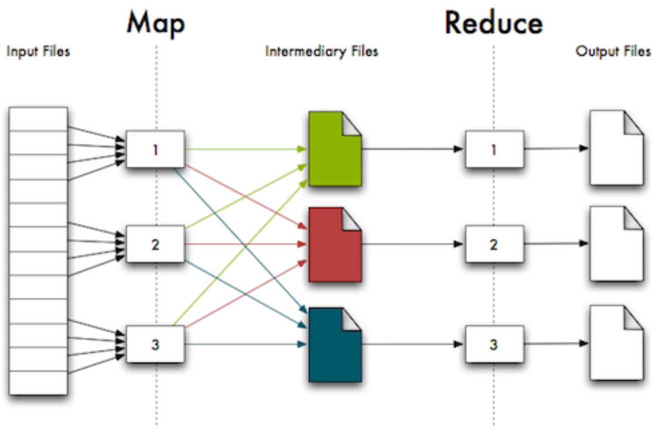


Figure 3: Simplified MapReduce phases [55]

With its functional programming approach, MapReduce “automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disk” as Dean et al. states [13]. Google uses MapReduce for various tasks where it is necessary two crawl and process several petabytes of data per day. One of the main problems of MapReduce is that the complete graph state has to be stored after every final reduce stage. This leads to heavy communication with the storage system. A solution to this problem could be to send the code to the node instead of sending the data of a node to another node for processing which was described before with the Pregel approach.

2.3 Storage systems

With the growth of internet companies such as Google or Amazon and social platforms such as Facebook or Twitter, new requirements for storage systems emerged. Google tries to store and analyze as many webpages and links as possible to provide relevant search results. In 2008, it already passed the mark of more than 1 trillion unique URLs [56]. Assuming every link is 256 Bytes, only storing the links already equals 232 terabyte of storage. Twitter has to store more than 90 million tweets per day [57] which is a large amount of data even if every tweet is only 140 characters long. Other kinds of so called big data are nowadays produced by location services (mobile phones) and sensor networks in general which are likely to grow massively in the next years [58]. The large amount of data makes it impossible to store and analyze the data with a single machine or even a single cluster. Instead, it has to be distributed. With the need for new storage solutions, Google created BigTable [59] based on Google File System (GFS) [60] and Amazon wrote the Dynamo paper [61] based on which SimpleDB and Amazon EC2 were implemented.

Also small companies are challenged by the big amount of data that is produced by its users and by the social interactions. A new community, known as NoSQL (Not only SQL), grew and created different types of storage systems. Most are based on Dynamo (Riak) or BigTable (HBase) or a combination of both (Cassandra). One more solution that was developed by Yahoo is PNUTS [62].

The main problem with current SQL solutions (MySQL, Oracle etc.) is that they are not easy to scale. Mainly, scaling writes and scaling queries to data sets stored and several machines. Current SQL solutions are often used as an all-purpose tool for everything. SQL solutions are perfect where a predefined data model exists, locking of entries is crucial, which means that Atomicity, Consistency, Isolation, Durability (ACID) is necessary and transactions have to be supported. Often, the new solutions relax parts of ACID for better performance or scalability, and in the example of Dynamo also for eventual consistency. Other solutions keep all the data in memory and only seldomly write their data on to disk. This makes them much more efficient, but it could lead to data loss in case of a power outage or crash as long as the data is not stored redundant.

There are use cases for every kind of system. All databases have to make tradeoffs at some point. Some of these tradeoffs are shown by the CAP theorem 2.3.1. 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 adaption in different types of applications. The data models of the different solutions are diverse and optimized for the use cases.

A switch from a SQL solution to one of the NoSQL solutions that were mentioned above is not always easy or necessary. The storage model and structure varies from system to system and querying data can be completely different by using MapReduce or so called views. Setting up these new solutions is often more complicated than simply installing a LAMP stack. SQL has been developed for a long time and many packages have been created to simply copy it to the hard drive to start working.

Most of the new storage systems were built so that they are able to run on several nodes to perform better, be fault tolerant and be accessible by different services, which means by a wide variety of programming languages. They offer interfaces like Representational State Transfer

(REST) [63] or Thrift [64] that allows easy access based on the standard HTTP Protocol over the local network or the internet from various machines. The communication language can often be chosen by the client. Standards as XML or JSON are supported. The efficiency of a system differs on the use case and there is not a system that fits all needs. The no free lunch theorem [65] can also be applied to storage systems.

Every system stores data differently. Data and queries have to be modeled differently for every system so that they fit the underlying storage structure best. Depending on the system and the use case, data has to be stored so that it can be queried as efficiently as possible. Most social data sets have many to many relationships between the nodes which is a challenge in most systems because it can not be represented naturally in the storage systems except for graph databases.

To compare the performance of the different storage system types, Yahoo created Yahoo! Cloud Serving Benchmark (YCSB) [66]. This framework facilitates the performance comparison of different cloud-based data storage systems. An other system to load-test various distributed storage system is vpork[11] coming from the Voldemort project. Measuring performance is crucial for finding the right storage system but is not treated in this thesis because the focus lies on the data structure and not performance comparisons.

One goal of NoSQL solutions is massive scaling on demand. Thus, data can easily be distributed on different computers without heavy interaction of the developer through a simple 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. In general, two different techniques are used: Sharding or distributed hash tables (DHT).

Sharding is used for horizontal partitioning when the data no longer fits on one machine. Based on the data ID, the data is stored on different system. As an example, all user IDs from 1 - 1000 are stored on one machine, all users from 1001 - 2000 are stored on the next machine. This is efficient for data that is not highly connected. With such data, queries can be answered by one single machine. As soon as queries have to query and join data from different machines, sharding becomes inefficient. Scaling based on sharding is efficient for writes but can be a bottleneck for queries. Sharding can use any kind of algorithms to distribute the data on different nodes.

Distributed Hash Table (DHT) [67] is based on the idea of peer-to-peer, which distributes data to the different nodes based on a hash key. DHT was implemented in 2001 in some projects like Chord [68]. Nodes can dynamically join or leave the ring structure and data is reorganized automatically. The same data is stored on different nodes in order to guarantee redundancy. This is of high value for the distributed storage systems, since machines in large clusters regularly fail and have to be replaced without affecting or interrupting the rest of the cluster. Efficient algorithms help to find the data on the different nodes.

Even if most systems look completely new at first sight, most systems recombine existing features. This is especially true for storage backends. Several distributed systems use well-known systems such as Oracle Berkley DB[12] or InnoDB[13] as storage backend on every node. These storage backends have proven to be stable and fast. They can often be exchanged in

[11]Distributed database load-testing utility: `https://github.com/trav/vpork`

[12]Oracle Berkley DB: `http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html`

[13]InnoDB: `http://www.innodb.com/`

order to optimize for a specific use case. Some systems allow the use of memory as storage backend for faster reads and writes. However, with the disadvantage of loosing the data in case of a crash.

In the following, first the technical details about the CAP Theorem, the Dynamo and the BigTable paper will be described. Then the five storage system types which are analyzed in this thesis – as relational database management system, key-value store, column store, document store and graph databases – will be described in detail. So-called triple stores and quad stores like Hexastore [51] or RDF-3X [53] that were created and optimized to store, query and search RDF triples, will not be analyzed because these storage systems are optimized for graph matching and graph traversal queries but not for processing data. Object databases will neither be covered in this comparison because there were no popular object databases found which are currently used for social data projects.

2.3.1 CAP, Dynamo and BigTable

Every storage system has its specific implementations. The next part will give a general overview of technical details that are used in various systems.

Cap Theorem Gilbert et al state that consistency, availability and partition tolerance are the three desirable properties when designing distributed web services[69]. Brewer states that always a trade-off decision has to be made between the three because only two of the three can be picked for every operation. This is also known as the Cap Theorem. Most new storage systems choose between availability and consistency, but do not make tradeoffs for partition tolerance because the systems are built to scale. Making tradeoffs for partition tolerance would mean that a cluster would get unusable as soon as a node fails.

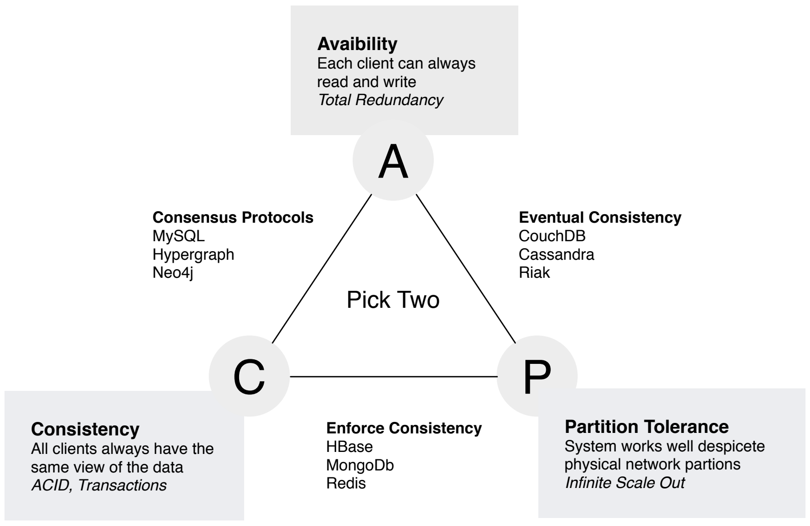


Figure 4: CAP Theorem

In figure 4, the three corners and edges of the CAP theorem in combination with the storage systems that will be discussed in this thesis are shown. The storage systems are always located at the edge where they can be positioned. Several storage types can be positioned on two different edges, since they can choose consistency and availability for every request, as described in the Dynamo paper which will be described more in detail below. A visual guide to NoSQL in combination with the CAP triangle was created by Clark Kent [70]. This shows extensive list of the different storage systems. The next subsections will describe the three states of the CAP theorem in detail.

Consistency Consistency means a service functions completely or not at all. Every operation in the system is atomic. If the same data is stored in two different data centers and one data center looses connectivity, the still available node will be in read only mode and does not allow to update the value until the other node is available again. As stated in the Dynamo paper [61], consistency can also be defined by $R + W > N$ where R are the nodes that can be read from, W the nodes that is written to and N the total number of available nodes.

Availability Availability means that the system is available for reads and writes all the time. This can be achieved by distributing data on different machines and storing it redundantly. Even if some nodes are down, reads and writes are still possible on one or several of the available nodes. Therefore, the same data can be updated on two different nodes with different values, because the nodes are not connected anymore or synchronization is too slow. There are different approaches to identify conflicts between the data sets as soon as the network is connected again. The first approach is a timestamp for every entry that is created. This requires synchronized time on all servers. The second approach includes vector clocks which are used in Dynamo like systems and will be described in more detail below. How the data is merged depends on the storage system. In some systems, conflicts are merged directly in the storage system, in others, the client has to take over this task.

Partition Tolerance Nodes still have to work well even if there are errors in communication between the nodes and the data is partitioned to different nodes. Gilber & Lynch describe partition tolerance as “No set of failures less than total network failure is allowed to cause the system to respond incorrectly.” [69]. To make this possible shared nothing systems are required which is implemented for most scalable data storage systems.

The two basic must read papers that inspired most of the NoSQL solutions are Dynamo from Amazon and BigTable from Google. Dynamo is a key-value store and is optimized for availability. It was created and influenced by the infrastructure that Amazon had at this time: data centers close to each other and connected through fibre channel cables which means low latency. In contrast, BigTable was created based on the idea of data centers which are located in completely different locations and focus more on consistency. Amazon built the Dynamo systems based on the idea that their book store and other services always have to be online, even if parts of the system are down. The user always has to be able to pay and checkout books, which means scaling writes is crucial. In the case of Google, most requests are reading requests (search requests) which means scaling reads is crucial. It is less of a problem if parts of the system are blocked for writes for a certain time.

Dynamo allows to choose for every request (read / write / delete) how many nodes should be queried or written to. Thus, the CAP theorem axis can be different for every request. This enables the developer to decide how redundantly data should be stored. The assumption is made if data is written to for example at least three servers out of five, the data is consistent. This is known as eventual consistency. Still it is possible that different versions of a data set exist on different servers. To differentiate between this different versions Dynamo uses vector clocks [71]. To handle temporary failures, the so-called sloppy quorum approach is used: Instead of using the traditional approach, data can be read and written to the first N healthy nodes instead of writing to the three predefined nodes. Otherwise it would be unavailable during server failures.

2.3.2 Relational database management system (RDBMS)

Relational database management systems are probably the storage systems that are at the moment used the most often. Data is modeled in tables consisting of predefined columns, the data itself is stored in rows. The data structure is often created based on the five normal forms which are designed to prevent update anomalies [72]. To query data that is stored in two tables, joins are necessary. For storing and querying structured data with a predefined model that is not changing, RDBMS are suitable. For every column indices can be set. Retrieving data based in indices is efficient and fast. Inserting data is also fast and efficient because only a row has to be inserted and the corresponding indices (often B+ trees) updated.

One of the main disadvantages of relational databases is that when data grows, joins become more and more expensive because lots of data has to be queried in order to create the results. As soon as joins become larger than the memory size or have to be processed on more than one machine, these joins are getting more and more inefficient the larger and more distributed the data gets.

In the area of RDBMS there is recently also a lot of development and also new RDBMS like VoltDB[14] appeared on the market that promise to be more scalable and still support ACIDity and transactions.

Popular RDBMS are MySQL 3.3.1, PostgreSQL[15], SQLite[16] or Oracle Database[17]. In this thesis, MySQL is used for the analysis.

2.3.3 Key-value stores

The simplest way to store data is with key-value store. Keys are normally strings, values can be any type of objects (binary data). Most key-value stores are similar to the popular memached [73] cache but add a persistence layer in the backend. The range of available key-value stores ranges from simple single list key-value stores to support for lists, ordered lists (Redis) or storing complete documents (Riak). Key-value stores are optimized for heavy reads and writes but only offer minimal querying functionality. The Dynamo implementation for example is basically a key-value store implementation that was implemented as SimpleDB.

[14]Scalable Open Source RDBMS `http://voltdb.com/`
[15]PostgreSQL: `http://www.postgresql.org/`
[16]Server-less SQL database `http://www.sqlite.org/`
[17]Oracle Database: `http://www.oracle.com/de/products/database/index.html`

Modeling complex data for key-value stores is often more complicated because pure key-values stores only support fetching values by keys. No secondary indices are supported. To store and retrieve data, combined key names are often used to make the keys unique and to make it possible to retrieve values again. The basic system does not support any query mechanism or indicies. Because of their simplicity, key-value stores are often used as caching layer.

Popular key-value stores are SimpleDB[18], Redis 3.3.2, membase[19], Riak 3.3.3 or Tokyo Tyrant [20]. For the experiments Redis and Riak are used.

2.3.4 Column stores

In Column stores, data is stored in columns – as opposed to the rows that are used in relational or row oriented database systems. With column stores it is possible to store similar data (that belongs to one row) close to each other on disk. Is is not necessary to predefine how much columns will be in one row. One of the first implementations in research was C-Store [74], which was created as a collaboration project among different universities.

Cassandra and HBase are based on BigTable[59], which is a standard column store implementation by Google. BigTable is used at Google for almost all their applications such search, Google Docs or Google Calendar. For scaling, Google uses the Google File System (GFS) [60]. Figure 5 is an example of how Google stores its search data in a BigTable. All data related to www.cnn.com is stored in one row. In the first column content, the webpage content is stored. In the other columns, all website URLs which have an anchor / URL to www.cnn.com are stored. This data is stored closely together on disk. For processing the data for www.cnn.com, the system has to retrieve only one row of data.

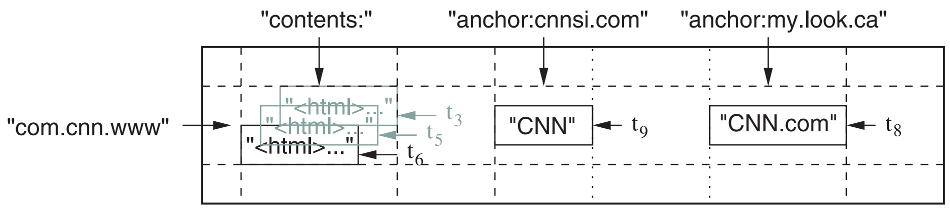


Figure 5: Column store

Column stores are built with the idea of scaling in mind. There are only few indices, most based on the row keys. Also, data is directly sorted and stored based on the row keys as in Big Table. This makes it efficient to retrieve data that is close to each other. There is no specific query language. Some column stores only allow to retrieve documents based on the row key or to make range queries for row keys. Some support more advanced query methods but not with a standardized query language.

[18]Key-value store based on the Dynamo paper: `http://aws.amazon.com/simpledb/`

[19]Simple key-value store based on memcached `http://membase.org/`

[20]Tokyo Tyrant: network interface of Tokyo Cabinet: `http://fallabs.com/tokyotyrant/`

Popular column stores are BigTable, Voldemort[21], HBase 3.3.6, Hypertable or Cassandra 3.3.7. In this thesis, Cassandra and HBase are used for the analysis.

2.3.5 Document stores

In document stores, records are stored in the form of documents. Normally, the structure and form of the data does not have to be predefined. The type and structure of the fields can differ between every document. Every stored document has a unique id to retrieve or update the document. The stored data is semi-structured which means that there is no real separation between data and schema. Thus, the model can change dynamically. Depending on the system values inside a document can be typed. Several document stores support storing the data directly as JSON documents. This facilitates the creating and retrieving of documents. Similar types of documents or documents with similar structures can be stored under buckets or databases. Normally, the storage systems automatically create a unique identifier for every document, but this identifier can also set by the client.

Document stores offer much comfort for the programmer because data can directly be stored in the structure it is. Thus, data does not have to be normalized first, and programmers do not need to worry about a structure that might or might not be supported by the database system.

To query data efficiently, some data stores support indices over predefined values inside the documents. There is no standard query language. Also systems have different strategies of querying the data. There are two different approaches to scale document stores. One is sharding, already known from the RDBMS. The documents are distributed on different server based on the ID. Also queries for documents have to be made on the specific servers. An other approach is based on the Dynamo paper. This says that documents are distributed based on a DHT on different servers. There is for example an approach to implement the storage backend of CouchDB based on Dynamo [75].

Popular document stores are MongoDB 3.3.4, CouchDB 3.3.5 or RavenDB[22]. In this thesis, CouchDB and MongoDB are used for the analysis.

2.3.6 Graph databases

In comparison to other storage systems, graph databases store the data as a graph instead of just representing a graph structure. Marko A. Rodriguez defines graph databases in a more technical way: “A graph database is any storage system that provides index-free adjacency” [76]. According to Rodriguez, moving from one adjacent vertex to the next always takes the same amount of time – regardless where in the graph the vertex is, how many vertices there are and without using indices. In other databases, an index would be needed to determine which vertices are adjacent to a vertex.

General graph databases such as Neo4j are optimized for graph traversal. Other typical graph application patterns are scoring, ranking and search. Triple and quad stores that are often used for storing and querying RDF data are no typical graph databases. Theses systems are optimized for graph matching queries to match subject, predicates and object and can often be used with the SPARQL [5] query language. There are also several different approaches to

[21]A distributed database `http://project-voldemort.com/`

[22]RavenDB: `http://ravendb.net/`

build graph databases based on existing databases such as for example FlockDB [77] which is based on MySQL, created by Twitter. These types of graph databases are often optimized for a concrete use case.

There are different query languages for graph databases. In the area of RDF stores, SPARQL [5] is the de-facto standard. Gremlin [78] uses another approach, which is optimized for property graphs[23]. Current graph databases often have their own implementations of a query or traversal language that is optimized for the storage system. In addition, the default query language SPARQL is often supported.

At the moment, graph databases evolve fast. New databases appear on the market, most of them with the goal of being more scalable and distributed, such as for example InfoGrid[24] or InfiniteGraph.

Popular graph databases are Neo4j 3.3.8, HypergraphDB[25], AllegroGraph[26], FlockDB [77] or Inifinite Graph[27]. In this thesis, Neo4j is used for the analysis. HypergraphDB was also considered, but Neo4j seems to have a simpler implementation, is more extensible and has a better documentation than HypergraphDB.

2.4 Data visualization

Data or information visualization means a graphical representation of data that is mostly generated by computers [14]. The goal of data visualization is to convert data to a meaningful graph [14] and make it unterstandable for people to communicate complex ideas and to gain insights into the data. Both art and science are part of data visualization. Shneidermann defined a task by data type taxonomy based on the Visual Information-Seeking Matra (overview first, zoom and filter, then details on demand) with seven data types and seven tasks [79]. Graph drawing is not only based on mathematical rules, but it also has to respect aesthetic rules: Minimal edge crossing, evenly distributed vertices and symmetry have to be taken into account [15]. It is important to find a good visual representation for the data that can be understood by people. There are different graph drawing models and algorithms. One often used method is the spring electrical model, which is scalable to thousands of vertices. Other approaches are the stress model or the minimal energy model[15].

Hu describes the visualization problem as following [16]:

> The key enabling ingredients include a multilevel approach, force approximations by space decomposition and algebraic techniques for the robust solution of the stress model and in the sparsification of it.

One of the general problems in data visualization is that many graphs are too large to be processed on a single machine. To solve this, some graph partitioning and community detection algorithms [80] have to be used, namely the parallel algorithms that were proposed by Madduri and Bader [81] [11].

[23]Property graph defined by blueprint: `http://github.com/tinkerpop/blueprints/wiki/property-graph-model`

[24]InfoGrid Web Graph Database: `http://infogrid.org/`

[25]HypergraphDB project: `http://www.kobrix.com/hgdb.jsp`

[26]Proprietary RDF store: `http://www.franz.com/agraph/allegrograph/`

[27]Distributed Graph Database: `http://www.infinitegraph.com/`

Visualization is crucial with the growth of social network and social data analysis [82]. Results have to be represented in a human understandable way. Often social graph data is visualized with dots and lines where dots are vertices/actors and lines are the connections in between. A technique often used for visualizing social data is edge bundling [83] to remove clutter in the image. Paul Butler describes a similar approach on how he created a visual representations of friendships on Facebook [84]. From the large Facebook data set he took a sample of about ten million friendships. To remove the clutter on the image and not having too much lines he used colors and weights for the lines between cities based on the number of friendships in between and the distance. The resulting image can be seen in figure 6.

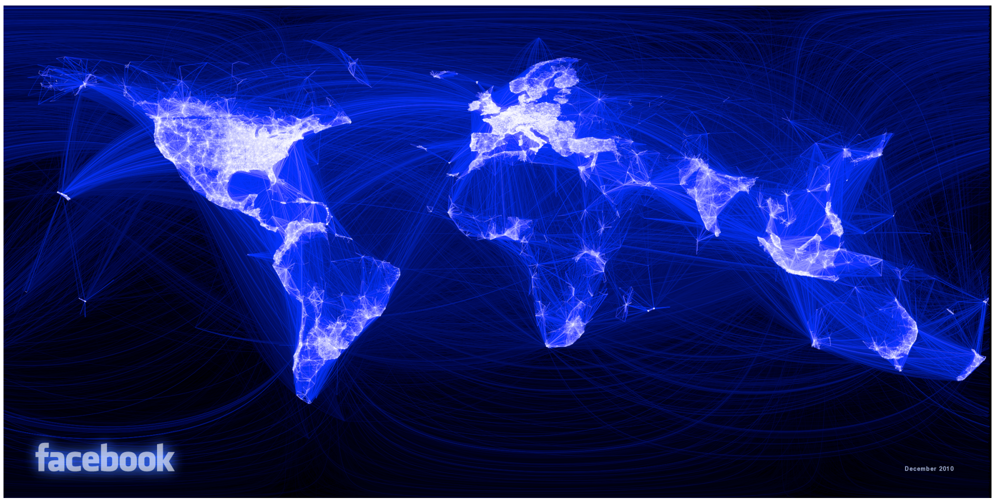


Figure 6: Facebook friendships visualized

There are different tools to visualize data. One of the most used and scalable ones is Graphviz [85] developed by different developers and the open source community. With the DOT language [24] Graphviz offers a simple pure text file structure to store graph data. These simple text files can be read and visualized by Graphviz but are also human readable. There are different algorithms to process and visualize the stored graph data. For example, sfdp is a new scalable visualization algorithm created by Yifan Hu. With Graphviz it is possible to process and visualize large graphs ($>10^6$ nodes).

Furthermore, there are already online platforms that offer visualization of data as a service. One of them is the Many Eyes experiment founded by IBM in cooperation with several universities. Many Eyes offers a wide variety of existing data that can be visualized. Different visualization types can be applied to the data. Furthermore, users are also able to upload their own data and share it with others.

3 Data, Methods & Implementation

For the analysis of social network data and the implementation of the centrality algorithm, a specific data set was needed. The useKit data set from the social ad-hoc collaboration platform useKit.com was chosen. The data was modeled for the following eight storage systems: MySQL, Redis, Riak, MongoDB, CouchDB, HBase, Cassandra and Neo4j. The data model was specifically created and adapted for every single system. To calculate degree centrality and PageRank on the different storage systems, the useKit data set first had to be stored in every system. Then, the degree centrality and PageRank algorithms where implemented on every system if possible. To better understand the global structure of the network, the data was visualized with the help of Graphviz.

In the following, first the exact structure of the useKit data will be described in detail, followed by a description of what the concrete algorithms of degree centrality and PageRank look like. Then, all eight storage systems and their technical background will be described in detail. Last, methods of visualizing network data are described.

3.1 useKit Data

Social data cannot be created simply by connecting random nodes through edges, because social networks are often small world networks. There are different methods to create these kinds of networks, such as the preferential attachment method proposed by Barabasi et al [86]. To not rely on randomly created data, a data set from the social ad-hoc collaboration platform useKit.com[28] was taken. The decision was made for the useKit data set because of the direct connection to the useKit team, the willingness of the useKit team to provide the data set and because other available social data sets are rare as described in section 2.1.3. useKit is a platform that offers organization of content combined with simple collaboration. The useKit data structure was simplified to its basics. Additional nodes such as comments or likes were neglected. The simplified data consists of three different types of nodes: Content, context and user, as can be seen in figure 7. Every node type is described below.

Content Every content item produced by a user in the useKit platform is stored as a content node. A content item can be a link, video, file or every other type of data that can be stored in useKit. Every content object is created by a user. Thus, this user is the creator of the created content object. In addition, several users can own the content object. This means that these users have a symlink to the same content item. A content item exists in the system as long as at least one symlink to the content item exists. These links are the edges between content nodes and user nodes. Because at least one user has to own the object but the number of owners is unlimited, this represents a 1 to N relation which is shown in figure 7. All content items can be in several contexts, but they do not have to be. This equals a 0 to N relation between the content and the context nodes. A content item is indirectly related to other content items over contexts or users. This means that if a content item is in the same context as another content object or the same user owns two content objects, there is an indirect relation between these two content objects.

[28]useKit Platform: `http://useKit.com`

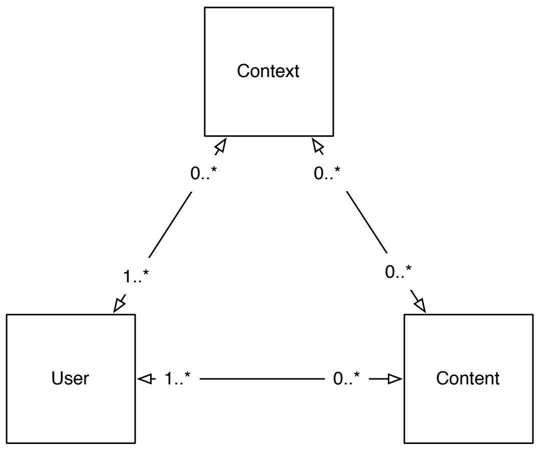


Figure 7: useKit Data Structure

Context A context is the entity that is used in useKit to organize and share content. A context can be added to various content items to organize them, which makes it similar to tags. But because a context object also has a relation to one or several users, the context object is also used for sharing as known from folders. Contexts allow users to organize and share content at the same time. Because different contexts can be assigned to a single content, every user can add its own layer of organization. Initially, a context is empty but an unlimited number of content objects can be added. This describes a 0 to N relationship between the context and content node, as can be seen in figure 7. The same as for a content object, every context object is owned by at least one user but several users can join or be invited to a context. This represents a 1 to N relationship from a context to a user node.

User The user node is every user that exists in the useKit system. A user can create content items and contexts or join contexts. There are no direct edges between two users, only indirect connections over context or content nodes. For reasons of simplicity, in the current useKit data set of this thesis these edges exist for users that invited another user but were ignored. If a context has other users that are in the same context, the user is connected over this context to the other users. If a content object has more than one owner, the user is also indirectly connected to these users. Because a user has initially no content and context items but can create an unlimited number of contexts and content items for both objects, there is a 0 to N relationship from the user node to the context node and content node which is shown in figure 7.

Edges The three types of nodes are connected through six different types of edges. These edges are directed. Every edge type describes an edge from one node type to another node type. Because in the useKit system an edge is always created automatically in both directions, the system can be simplified to three undirected edges. If for example a user joins a context, an edge from the user to the context is created. At the same time, the user is added to the context, which represents an edge from the context to the user. This leads to an undirected graph, which is different from directed link networks as described before where a link from one node to another node is always directed. This is also the case in a social network such as Twitter where a user can follow another user without the user following the current user. In Facebook, on the other hand, a link is always bidirectional. As soon as two users are friends, an undirected edge exist in between.

In the useKit data described above, eight different types of edges exist:

1. user is in context
2. user created context
3. user owns content
4. user created content
5. context has content
6. context has user
7. content has user
8. context is in context

Already in the raw useKit data, the directed edges 1 and 6, 3 and 7, 5 and 8 are combined to an undirected edge. The additional information in edge 2 and 4 about the creator of a content or context are neglected in this thesis for reasons of simplification. Otherwise there would be more than one edge between certain nodes. By ignoring these edges, knowledge about the exact relations is lost. It was tried to add edge weights instead of double edges. This would mean that a creator of a content or context object has a stronger relation to these nodes. It is assumed that this could be true but is not always the case. Based on this, it was decided to go with the simpler solution which leads to an undirected and unweighted graph where all edges have a weight of one.

In this thesis a simplified data structure of the useKit data set is used. There are no edges between user – user, content – content, context – context as described before. This is untypical for social data where nodes of the same type are normally connected directly by an edge. This is for example the case in Facebook, where users are connected with other users by being friends. In the useKit data, all relations between two nodes of the same type involve an additional node which adds some information about the relation between these two nodes. In further experiments it would be interesting to use this to describe the relationships between similar nodes in more detail. For example, if a content object has several contexts and several users, this implies some connections between the different contexts and between the different users. If a content is in different contexts, one assumption that can be made is that the contexts are somehow related to each other because they share the same content item. Is this the case for several content objects that are in the same context, the assumption is made that these contexts have something in common. It could mean that these two contexts have a similar or related meaning. The same can be done for users that share the same context or the same content objects. The assumption can be made that these users share some interests, which could mean these users are close in the network graph and could be recommended as friends to each other. With this data structure it is possible to find users with similar interest, contexts which are similar or content objects that belong to similar subjects which makes it possible to calculate relations and make recommendations based on the results. This could be done in further experiments on the same data.

Figure 8 displays a real world example of the useKit data represented as a visual graph. A small network with users, contexts and content items is shown. There are three different users in the network: Hans, Eva and René. Every user has several content and context objects. There is no direct edge between the users, contexts and content items as described before and the graph is undirected. The users are connected indirectly either through contexts or contents. René and Eva have two connections over one hop: the context Uni Basel or the content object Profs. There are also connections over two or more hops like Hans – Basel – Inkubator – René. Every content node is connected to at least one user node. The same is true for every context node that is connected to at least one user node. The three different types of edges are also shown in the image. It is typical for the useKit data set that the number of content items is much larger than the number of contexts and users.

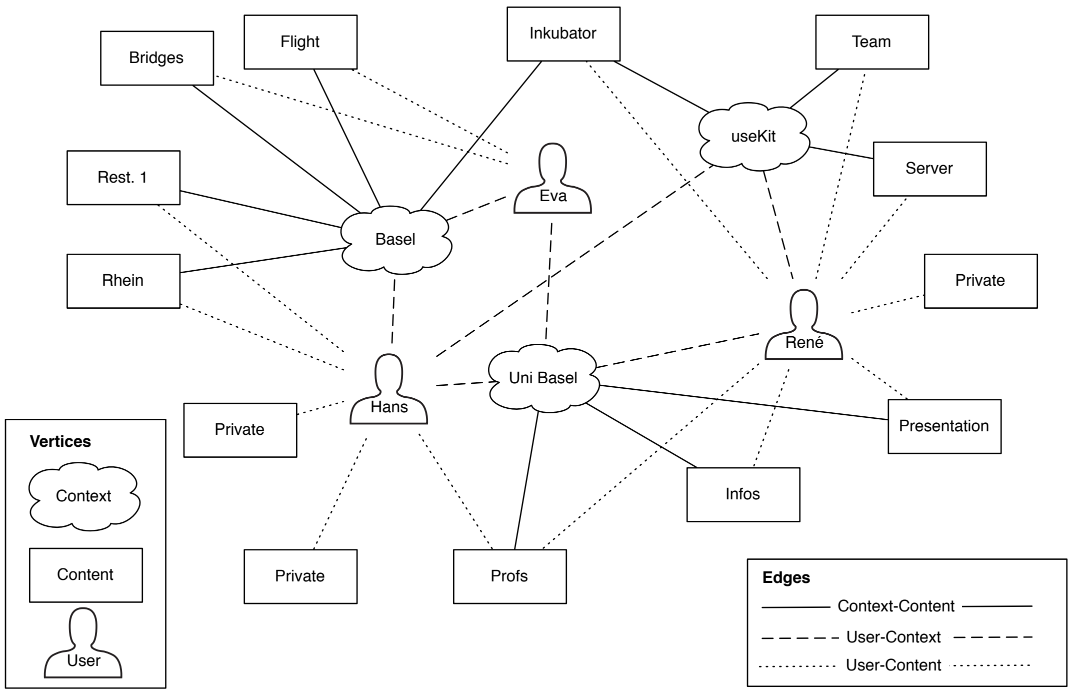


Figure 8: useKit Example data

The data set for the experiments was created from the original data set by processing and filtering the original social network data from the useKit platform and by storing the filtered data in a local MySQL database. All data that is not relevant for the following was filtered out which leaded to the data set described above. To do the experiments on an even larger data set, it was tried to create a random useKit data set based on Barabasi algorithm [86]. The results were too different from the real data set, which leaded to the decision to only use the original data. There was no further investigation in this subject. The existing data set with more than 50'000 nodes and more than 100'000 thousand edges was sufficient for the experiments and made it possible to do all the experiments on a single machine.

3.2 Social graph analysis

The social graph produced by the useKit data was analyzed with different methods. Only a limited set of algorithms was evaluated. The degree centrality was chosen because it has a simple algorithm but still delivers relevant information in most cases even though only a local view on the node is returned. The PageRank algorithm was chosen as an algorithm that makes processing the complete graph necessary and returns a more global view for a single node.

3.2.1 Degree centrality

Degree centrality was calculated for all three node types. Different edge weights were evaluated to receive more relevant results. Because the useKit graph is an undirected graph, this thesis does not make any differentiation between in and out degree. The freeman algorithm where every edge has a weight of one was used. Thus, the degree centrality of a node is exactly the number of edges connected to the node.

To calculate degree centrality, two different types of implementations were made. One is the basic iterative implementation that can be seen in listing 1. All edges are retrieved from a node and a loop is made to sum up all the edge weights. In our case, all edges have weight one.

```
function degree(node) {
    edges = node.getEdges();

    degree = 0;
    foreach edges as edge
        degree += edge.getWeight();

    return degree;
}
```

Listing 1: Iterative degree centrality code

The second type of the implementation was done with MapReduce to make it available for large scale processing. This implementation can be seen in listing 2. The map phase is the same as the iterative calculation that sums up all the edge weights. The reduce phase is optional but is used to order the results based on the degree value.

```
function map(key, node) {
    edges = node.getEdges();

    degree = 0;
    foreach edges as edge
        degree += edge.getWeight();

    emit(key, degree);
}
```

```
function reduce(key, values) {
    emit(values.sort(function(a, b) {
        return b.degree - a.degree;
    }));
}
```

Listing 2: MapReduce degree centrality code

The detailed implementation of the two algorithms for the different storage systems will be discussed in the storage system section 3.3.

3.2.2 PageRank

The standard PageRank calculations are done on directed graphs as web link data sets. Because the useKit data set is an undirected graph, for the PageRank calculation the assumption is made that every undirected edge consists of two directed edges. Thus, every node has the same number of incoming as well as outgoing edges and the standard PageRank algorithm can be applied. As seen in equation 3, PageRank can be calculated iteratively. A simplified version of the original PageRank algorithm is shown in equation 2 in section 2.2.3. This simplified version of the algorithm ignores the damping factor, which means that there are no random jumps to other nodes. These random jumps are partially used to prevent data sinks. This is less of a problem in the useKit network, because the undirected graph was converted to a bidirectional graph as described before, which means every node that has an incoming edge also has an outgoing edge, so no sink nodes exist. Because the main focus in the thesis lies on the data structure, data model and the concrete implementation of the PageRank algorithm and not on finding the best algorithm for the useKit data set, this simplification was made.

$$P'(u) = \sum_{v \in B_v} \frac{P'(v)}{N_v} \tag{3}$$

Instead of summing up the edge weights as done in the degree centrality algorithm, the value of an edge is the sum of all values sent by its neighbor nodes as seen in listing 3. This pseudo code is heavily simplified. The function getIncomingEdgeNodes can be an expensive operation. The same is true for getPartialWeights where calculations on the specific node have to be done first before the value can used in the target node. The damping factor is neglected for simplicity reasons as mentioned before. The result of an iteration for a simple node cannot directly overwrite the existing PageRank value inside the node, otherwise it would affect the calculations for the following nodes that are made after this node. All intermediate results have to be stored in an additional storage location until one complete iteration over all nodes is done. Then the existing PageRank values can be overwritten to every node.

```
function pagerank(node) {
    incomingNodes = node.getIncomingEdgeNodes();

    pageRank = node.getPageRank();
    foreach incomingNodes as incomingNode
        pageRank += incomingNode.getPartialWeight();

    return pageRank;
}
```

Listing 3: PageRank iterative pseudo code

MapReduce at Google was developed to scale the PageRank algorithm to billions of websites. The pseudocode for the map and the reduce phase can be found in listing 4. In the map phase, first the current PageRank value of the node is divided by the number of incoming edges. The resulting value is the amount that is sent to each connected node in the second part of the map phase. In the reduce phase, all these partial values sent for a single node are summed up and stored as new PageRank value. Detailed optimization proposals for this algorithm can be found in "Design Patterns for Efficient Graph Algorithms in MapReduce" from Lin and Schatz[41]. PageRank is an algorithm that converges after multiple iterations. The size of the graph only slightly affects the number of iterations, often ten iterations are already enough for a good approximation of the exact PageRank [87]. For multiple attached iterations of the PageRank algorithms it is necessary to also emit the complete node information during the map phase. Then all information that is necessary for several map and reduce phases directly exists inside the system. This allows us to repeat the PageRank calculation several times. Would the node not be sent, the node information like connected nodes would be lost and multiple iterations would no be possible without accessing the storage system.

```
function map(key, node) {
    partialPageRankValue = node.PageRank / #node.adjacencyList;

    foreach node.adjacencyList as adjacencyNode
        emit(adjacencyNode.id, partialPagenRankValue);
}

function reduce(key, values) {
    pageRank = 0;

    foreach values as value
        pageRank += value;

    emit(key, pageRank);
}
```

Listing 4: PageRank MapReduce code

The detailed implementation for every specific storage system can be found in the section 3.3. Depending on the system, the iterative or the MapReduce implementation was chosen.

3.3 Storage systems and their data models

Since the amount of data is growing, there is a need for new storage systems. The number of different so-called Not only SQL (NoSQL) solutions is growing fast. Thus, in the course of writing this thesis, new systems appeared and the analyzed systems evolved and were improved. During the next years, probably only the best and most stable systems will further be developed and it is likely to become easier to choose the right system for a specific problem. The current development pace for NoSQL solutions is quite high. While writing this thesis, Redis evolved from version 1.0 to version 2.2, several features were added and the speed improved. This fast development pace has lead to confusion in the communities because there are many blog entries or outdated documentations where old systems are compared without a specific version number.

The storage systems which are analyzed in this thesis were chosen based on the following factors: Documentation, usage in projects, community behind the project, stability, active development, open source, available clients and my personal assessment. All the chosen systems are open source. This was an important basis for decision, because it makes it possible to evaluate the systems in detail by studying and analyzing the code when needed. One or two systems of every type were chosen to compare the results and also to guarantee that at least one solution still exists at the end of writing this thesis. One more reason to pick more than one solution per type was that there are completely different implementations for the same type, as can be seen with Riak and Redis that are basically both key-value stores but have completely different implementations. New systems appeared during writing, which are mentioned below but not evaluated in detail. The systems were selected in July 2010.

For simplicity it was decided to make all client implementations in Java. Java was chosen because all storage systems offer a Java client or have a Java API by default. In addition, Java allows us to make the written code cross platform executable. The specific clients that were chosen for the individual storage system are listed in the description of every storage system below. The communication between the client and the server depended on the server, but most systems have REST interface and support communication through JSON, which was also used for data modeling.

The data was modeled for every single storage system to store, query and process the data in an adequate way. For most storage systems, the data had to be denormalized. The advantage of this approach is that data is stored in a way that is ideal to query the data, the disadvantage is that much duplicated data exists in the system and the programmer has to take care of removing duplicated or outdated data which is more a fat client – thin server architecture approach.

In table 1 an overview of the eight evaluated systems can be found where storage type, implementation code language and supported query languages are listed. For a more extensive list of storage systems, Knut Haugen created a blog entry with Google Docs that lists more details [88]. All storage systems support simple key requests based on an index, except Neo4j where this can be added through a Lucene[29] index. The storage systems are written in different languages. In this context, Erlang[30] – as a functional programming language built to create distributed, fault-tolerant systems – should be mentioned. It seems as if Erlang is a good match for distributed systems such as Riak.

In the following, all eight storage systems are described in detail. The initial data set was stored in MySQL. To feed all the other systems with data, scripts were created to query the data from the MySQL database, remodel the data and write it into the specific storage system.

3.3.1 MySQL

MySQL[31] is probably the most popular relational open source database. It is used in a wide variety of projects and has proven to be stable. Because of its easy installation and setup it is used in lots of small projects, but also in large production environments as for example

[29]Lucene project: `http://lucene.apache.org/java/docs/index.html`
[30]Erlang a functional programming language: `http://www.erlang.org/`
[31]MySQL webpage: http://www.mysql.com/

	Data Model	Code Language	Query Language
MySQL	Relational	C	SQL
Redis	Key-value	C	-
Riak	Key-value	Erlang	MapReduce
MongoDB	Document	C++	SQL, MapReduce
CouchDB	Document	Erlang	MapReduce
Cassandra	Column	Java	Range, MapReduce
HBase	Column	Java	Range, MapReduce
Neo4j	Graph	Java	SPARQL, internal

Table 1: Evaluate storage system comparison

Twitter where it is rather used the way of a key-value store. MySQL supports transactions and also relations (foreign keys) depending on the backend. For every table, an individual storage backend can be chosen. The two most popular backends are InnoDB an MyISAM.

To scale the MySQL database to more than one machine, replication is supported. With the master-slave architecture, especially reads can be scaled. All writes are going 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.

MySQL is used to store social data in lots of different projects. Currently, Facebook and Twitter are probably the most famous ones. The main problem for storing social data in MySQL is that social data normally has the form of a graph, which means querying includes lots of joins over different tables with relations. MySQL is not optimized for such queries, which means that such queries are expensive. Furthermore, social platforms often have the 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. This is partially solved by Facebook by adding cache solutions in front of MySQL to aggregate different cache results so complex queries are answered by the cache on not MySQL itself.

As client MySQL Connector/J [32] was used. The used server version was 5.1.50.

Data model The data for MySQL was modeled completely normalized with six tables as shown in figure 9. A table for every node type where the id was stored and a table for each type of edge that represents all N to N relations between the three node types. The tables user, content and context represent the nodes and the tables in between the edges which are the N to N relations. This model only allows to represent an undirected graph. To represent a directed graph, three additional tables with the edge data would be necessary or an additional parameter in the edge table which defines the direction of an edge.

[32]Java MySQL client: `http://dev.mysql.com/doc/refman/5.1/en/connector-j.html`

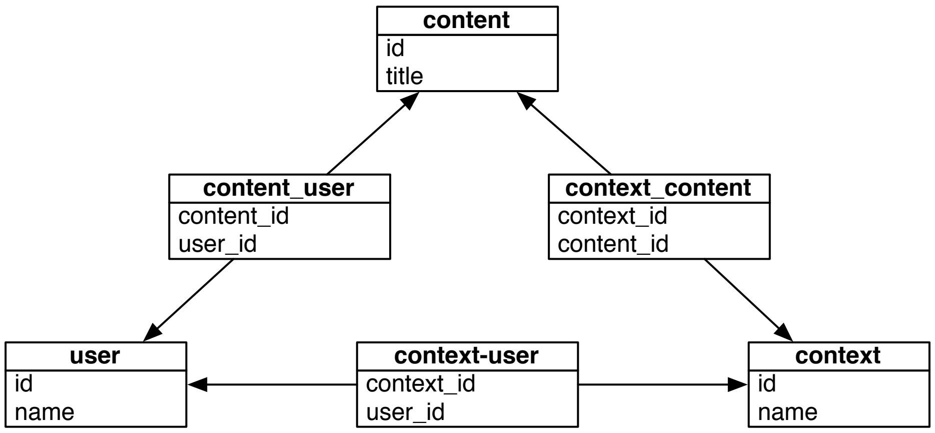


Figure 9: MySQL data model

Queries Degree centrality can be calculated rather easily, as seen in listing 5 for the user with id 1. It is necessary to know the node type and the node id. Based on this, two select count queries on the edge tables can be made for the specific node id and the two results are summed up to the degree centrality of the node.

```
SELECT
    (SELECT COUNT(*) FROM `content_user` WHERE user_id = 1)
        +
    (SELECT COUNT(*) FROM context_user WHERE user_id = 1)
```

Listing 5: Degree centrality for user node 1 in SQL

There is no query for calculating the PageRank in MySQL because it would require to make all calculations on the client side. It could probably be done with stored procedures but this was not done in these experiments. MySQL can only use the sequential PageRank approach (equation 2) because there is no support for MapReduce queries. To calculate PageRank on MySQL, all intermediate results have to be cache/stored until the iteration over all nodes is finished. Then every single value has to be written back to MySQL before the next iteration.

3.3.2 Redis

Redis[33] is a key-value store 2.3.3 that in addition to simple key-value stores also allows to store sets and collections and is written in C. It is optimized to keep all values in memory similar to memcached [73] but frequently writes the data to disk. How often the data is written to disk can be configured. Redis can be used in an append only mode, which means every change is directly written to disk, but 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

[33]Redis key-value store: `http://redis.io/`

queries or atomic push/pop operations. To scale reads, master-slave replication is supported. Distributing writes has to be done on the client side through sharding for example based on key hashes. Redis has a shell client for simple interactions.

For all experiments Redis server version 2.0.2 was used. The Java client used was jredis version 1.0-RC2[34]. A final release was not available during the time of the experiments.

Data model With its support for lists, it is convenient to use the list features for storing the nodes and its adjacency list. Twelve different types of keys were created to store the values for the nodes and the edges as can be seen in listing 6. All ids of one type are stored in a list, additional information to a node is stored under the key type:(id) where id has to be replaced by the object id. The two adjacency list types are stored in sets. The set type was chosen for the edges because retrieving the number of items is directly supported by Redis with the command *scard*. Thus, the information about every edge is stored twice. The information about the edges user – content is stored in user:(id):contents and in content:(id):users. This would allow to also store directed graphs where these values could be different.

```
users => type list
user:(id):user => type string
user:(id):contexts => type set
user:(id):contents => type set
contents => type list
content:(id):user => type string
content:(id):contexts => type set
content:(id):users => type set
contexts => type list
context:(id):user => type string
context:(id):users => type set
context:(id):contents => type set
```

Listing 6: Redis structure

Queries The degree centrality value for a node can be calculated with the two lists of adjacency nodes. The number of items in these two lists have to be added. This equates to the number of edges for a node. Listing 7 shows the Java code of how this can be done with the scard command that directly returns the number of items stored in the list given by the key. The degree centrality of the node is calculated by adding these two values. For a weighted graph it would be necessary to iterate over every list item by retrieving all list values and sum them up.

```
contextCounter = jredis.scard("user:" + userId + ":contexts");
contentCounter = jredis.scard("user:" + userId + ":contents");
degree = contextCounter + contentCounter;
```

Listing 7: Degree centrality in Redis

Because Redis does not support queries inside the storage system, calculating PageRank also means that it is necessary to make all calculations on the client side. The same calculations would be necessary as in the MySQL case and an additional storage structure would be

[34]Java Redis client: `http://code.google.com/p/jredis/`

necessary to store all the intermediate results. This was not done because the main focus is on the storage systems and not the processing on the client side.

3.3.3 Riak

"Riak is a Dynamo-inspired key/value store" [89] and uses Bitcask [90] as default storage backend. For every read it can be chosen from how many nodes the read has to be answered and for every write to how many nodes it has to be successfully written before it assumes that the action was successful. Thus, the programmer can decide for every single request whether consistency or availability should be chosen from the CAP theorem.

Riak was built from ground up as a distributed system. This is the reason that Erlang is used 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 [91] are used. All nodes in a Riak cluster are equal and new nodes can dynamically join or leave a Riak cluster, which makes it horizontally scalable. There is no dedicated root node.

The Riak storage model supports buckets (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. All keys from a specific bucket can be retrieved at once, but this is an expensive request because all nodes have to be queried for all values. As query language, Riak offers MapReduce where internally also link walking can be used. MapReduce queries can be written in Javascript or Erlang. Results are returned as JSON objects.

All functionality in Riak can be accessed through a REST interface. There are different clients for all kinds of language. For the implementation the official Java riak-client[35] was used. The current version from trunk was used because so far, no binaries exist (September 2010). For all experiments Riak version 0.12 was used. During the experiments version 0.13 was released with an implemented search engine based on Lucene and speed improvements.

Data model Because of the support for JSON objects data is stored similar to document stores. All edge informations are stored as arrays inside the node document. It would be possible to use links to connect the nodes instead of arrays with ids, but it was decided that it is simpler and probably also more efficient to store the edges as arrays. Because MapReduce queries can only be done over one bucket, all data was stored in a single bucket.

The id for every object in the Riak bucket was created by combining the type and the document id. All nodes that are connected to another node are stored in an array adjacency-list as seen in listing 8 where a single node is represented. If this would be a user node, the first two (type) would be replaced by user and the (type) in the adjacency list would be replaced by content and context with their specific ids. Inside the adjacency-list array, all content and context ids would be stored. This storage structure is optimized for the MapReduce queries where it is easy to iterate over a single array of adjacency ids. A more optimized structure

[35]Official Riak java client: `http://bitbucket.org/jonjlee/riak-java-client/wiki/Home`

to use the data in a live platform would probably be to split up the adjacency-list into two separate arrays for content and context edge arrays in the case of a user node.

```
id:(type)-(id) {
    id:(type)-(id),
    adjacency-list: [
        (type)-(id),
        (type)-(id),
        ...
    ]
}
```

Listing 8: User document in Riak with id 1

Queries Degree centrality in Riak can be calculated in two different ways. One method is to retrieve the specific document that has all the relevant data for one node as a JSON string and convert it to a JSON object in Java. Then the degree is calculated by getting the length of the adjacencyList array as seen in listing 9. The returned value is the degree of the specific node.

```java
FetchResponse response = riak.fetch("nodes", "user-1");
o = response.getObject();

JSONObject json = new JSONObject(o.getValue());
JSONArray adjacencyList = json.getJSONArray("adjacencyList");

int degree = adjacencyList.length();
```

Listing 9: Riak Degree java code

The second method is to calculate the degree centrality for all nodes at once with MapReduce. This makes it possible to also directly compare the nodes with each other and sort the results. The MapReduce degree centrality algorithm in Riak is shown in listing 10. In the map phase, the degree for every node is calculated and returned similarly to the first method. In the reduce phase, the results are sorted based on the degree value. The map-reduce code is written in JavaScript which is directly supported by Riak.

```javascript
function map(value, keyData) {
    var v =  Riak.mapValuesJson(value);
    var degree = v[0].adjacencyList.length;
    var obj = new Object();
    obj.degree = degree;
    obj.id = v[0].id;
    return [obj];
}

function reduce(valueList) {
    return valueList.sort(function(a, b) {
        return b.degree - a.degree;
    });
}
```

Listing 10: Riak Degree centrality MapReduce function

The PageRank query is implemented as MapReduce query based on the code in listing 4. The modified code for Riak written in JavaScript can be found in listing 11. After every iteration, all the values are written back to storage in the Java code. It was not clear how the reduce phase could write directly to the Riak backend or how several map and reduce phases could be done at once to directly do several iterations in the storage system without client interaction.

```
function(value, keyData) {
    var data =  Riak.mapValuesJson(value)[0];
    var pageRank = data.pageRank;
    var adjacencyCount = data.adjacencyList.length;
    var p = pageRank / adjacencyCount;
    var values = [];

    for (var i = 0; i < adjacencyCount; i++) {
        var node = {};
        node[data.adjacencyList[i]] = p;
        values.push(node);
    }

    var thisNode = {};
    thisNode[data.id] = pageRank;
    values.push(thisNode);
    return values;
}

function(values) {
    values = Riak.filterNotFound(values);
    var r = {};

    for (var i in values) {
        for (var w in values[i]) {
            if (w in r) {
                r[w] += values[i][w];
            } else {
                r[w] = values[i][w];
            }
        }
    }
    return [r];
}
```

Listing 11: Riak MapReduce PageRank

3.3.4 MongoDB

MongoDB[36] is a document store which is developed as an open source project but backed and actively supported by the company 10gen. It is written in C++. The document data is stored in a Binary JSON format called BSON. Several similar documents are stored together as so-called 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

[36]MongoDB document store: `http://www.mongodb.org/`

retrieve data that is also based on these indices 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 are supported. Atomic operations on single fields inside documents are supported. This is different compared to 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 language which leverage the REST interface of MongoDB or directly connect to the socket.

Large files can be stored in MongoDB by using GridFS as storage backend. 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. MongoDB at scale is already used by foursquare[37].

For all experiments the official MongoDB java client version 2.1[38] was used and the MongoDB server version 1.6.3.

Data model For MongoDB the same data model as for Riak is used which can be found in listing 8. MongoDB converts the given JSON object to a BSON object for storing. For simplicity reasons, all objects are stored in one MongoDB collection. It would also be possible to create a collection for every node type to make the data simpler to distribute based on the node type. No indices were set on the data set except the default index on the document id.

Queries Calculating degree centrality in MongoDB is similar to Riak. The document is fetched from the storage system as a string, converted to a JSON object and then the length of the array is checked as seen in listing 12. First a connection to the collection useKit is created, then a query object for the id $user - 1$ is made. The query is executed and the single response object is fetched.

```
DBCollection collection = db.getCollection("usekit");

BasicDBObject query = new BasicDBObject();
query.put("_id", "user-1");

DBObject result = collection.findOne(query);

JSONObject json = new JSONObject(result.toString());
JSONArray adjacencyList = json.getJSONArray("adjacencyList");

int degree = adjacencyList.length();
```

Listing 12: MongoDB degree centrality

In addition, MongoDB also supports MapReduce queries. This allows to calculate the degree centrality on the complete data set directly inside the storage systems and compare the results. In listing 13 the map and reduce functions are shown. The map function emits the degree centrality for a node, the reduce function only returns the value. The results are stored

[37]foursquare: Location check-in service`http://foursquare.com/`

[38]MongoDB Java client: `http://www.mongodb.org/display/DOCS/Java+Language+Center`

in res.results and are queried for the ten first results, ordered descending by the degree value. This code can directly be executed in the mongo console.

```
function map() {
    emit(this._id, this.adjacencyList.length);
}
function reduce(key, values) {
    return values[0];
}
res = db.pagerank.mapReduce(map, reduce);
db[res.result].find().sort({value:-1}).limit(10);
```

Listing 13: Degree centrality in MongoDB MapReduce based

The PageRank implementation is based on MapReduce. As seen in listing 14 in the map phase all partial PageRank values are emitted and then summed up by the reduce function to the final PageRank of the node. The if clause of the values array is necessary because the reduce function has to be idempotent, which means it can be called several times and still has to return the right result.

```
function map() {
    var adjacencyCount = this.adjacencyList.length;

    if (this.adjacencyList.length > 0) {
        var p = this.pagerank / this.adjacencyList.length;
        var values = [];

        for (var i = 0; i < adjacencyCount; i++) {
            emit(this.adjacencyList[i], p);
        }
    }
}

function reduce(key, values) {
    var r = {};
    var sum = 0;

    if (values.length > 0) {
        for (var i = 0; i < values.length; i++) {
            sum += values[i];
        }
    } else {
        sum = values;
    }
    return sum;
}

res = db.pagerank.mapReduce(map, reduce);
db[res.result].find().sort({value:-1}).limit(10);
```

Listing 14: PageRank on MongoDB

3.3.5 CouchDB

CouchDB is a document store written in Erlang and is developed as an open source project under the apache license. A RESTful API exists 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.

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 into the system on startup. Views are MapReduce functions written in JavaScript that process the stored data and return the results. A query language similar to SQL is not supported. 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 in the system, which makes the views much more efficient. The generated views can also be distributed to several CouchDB nodes, which makes it possible to distribute queries. Since version 0.11 CouchDB supports linking documents in view 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.

CouchDB is already used in research for Data Management and Workflow Management at CERN, the European Organization for Nuclear Research[92]. As CouchDB client couchdb4j[39] for Java was used. Because there were no up-to-date releases, the current version from trunk was used. Other Java clients that were evaluated were ektorp[40] and jrelax[41]. The decision was made for couchdb4j because of its simplicity.

Data model The useKit data for CouchDB was modeled in the exact same way as for MongoDB and Riak, which can be found in listing 8. CouchDB directly stores the data in JSON format, similarly to Riak.

Query The degree centrality in CouchDB is calculated with a MapReduce query that creates a view. Based on this view, CouchDB creates indices to query the data more efficiently. To retrieve the top ten nodes based on the degree centrality, the query in listing 15 is used. No reduce function is needed because the results are already sorted by the return key which is the degree with the document id as value.

```
function(doc) {
    emit(doc.adjacencyList.length, doc.id);
}
```

Listing 15: Degree centrality in Riak

The PageRank MapReduce query did not work on CouchDB. The CouchDB MapReduce queries require a specific reduce factor in the reduce phase. The number of results after the first reduce step has to be significantly smaller than in the map phase. This factor is predefined

[39]CouchDB Java client from Marcus Breese: `http://github.com/mbreese/couchdb4j`
[40]Ektorp CouchDB Java client: `http://code.google.com/p/ektorp/`
[41]jrelax CouchDB Java client: `http://wiki.github.com/isterin/jrelax/`

by the CouchDB query system and throws an error if the number of results in the reduce phase does not reduce fast enough. As in the PageRank algorithm, the number of start nodes before the map phase and after the reduce phase are both the total number of nodes and are thus exactly the same. The PageRank query in CouchDB always threw an error because the reduce factor was not large enough, which made it impossible to implement the PageRank algorithm in CouchDB. No configuration parameter was found to disable this check.

3.3.6 HBase

HBase [93] is a column storage based on the Hadoop Distributed File System (HDFS)[42] created with the idea of BigTable [59] in mind. The HBase project is open source. HBase is strongly consistent and picks consistency and partition tolerance from the CAP theorem as seen in figure 4. HBase is like Hadoop completely written in Java[43]. To access HBase, the Thrift[64] 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 as seen in figure 5. A table contains rows where every row is an array of bytes. HBase keeps rows lexicographically ordered. Every row can have several column families which are the same for every row in a table. Every column family can have several column keys with values inside. To retrieve a specific value, the row key, column family, column key and the timestamp have to be known. If the timestamp is left out, the newest version of the value is returned. HBase stores a timestamp in milliseconds with every value.

HBase supports range queries or can be used as key-value stores where the key has to be known to fetch the data. 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[44], 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. More detailed information about the performance of HBase can be found in the paper of Carstoiu et al [94]. Replication in HBase is done based on log files.

HBase in combination with Hadoop for data processing is used by several companies such as Twitter, Facebook or Yahoo in large scale, which means more than 1000 nodes. The company cloudera offers Hadoop in combination with HBase for enterprises. They also offer simple packages for linux that can be installed with the package manager which makes setting up HBase much simpler.

[42]Hadoop Distributed File System: `http://hadoop.apache.org/hdfs/`

[43]HBase Java API `http://hbase.apache.org/docs/current/api/overview-summary.html`

[44]Zookeeper: Centralized managing service: `http://hadoop.apache.org/zookeeper/`

A project that is similar to HBase is Hypertable[45] which is also based on the idea of BigTable. The two systems have been compared by Khetrapal and Ganesh [95]. For all experiments, Hadoop and Hbase version 0.20.6 have been used in combination with the official Java client[46].

Data model For the useKit data set, all data was stored inside the useKit table on a single HBase instance. Every node was represented by a row in the useKit table. The row name was the id of the node for which the node type and the ids are concatenated (example content-17). Inside every row, two different column families were created, one for storing the adjacency list for every node, a second one for storing the PageRank values as it is shown in listing 16. The adjacency column family contains all adjacency nodes which are stored with nodeId as column key, but also as value. The timestamp is not relevant here and is automatically set by the system.

```
useKit {                        // Table
    nodeId {                    // Row name
        adjacencies {           // Column family
            nodeId: nodeId;     // Column key : value
            ...
            ...
        }
        pagerank {              // Column family
            nodeId: pageRank;   // Column key : value
        }
    }
}
```

Listing 16: HBase useKit data model

Queries Degree centrality for a single node can be calculated in HBase as seen in listing 17. First, the table has to be set, then a get query is created where the specific row user-1 is set and the column family adjacencies as defined in the data model. Then the number of entries can directly be retrieved, which is also the degree of the specific node.

```
HTable table = new HTable(config, "useKit");
Get g = new Get(Bytes.toBytes("user-1"));
g.addFamily(Bytes.toBytes("adjacencies"));
Result r = table.get(g);
int degree = r.size();
```

Listing 17: Degree centrality on HBase for 1 node

To calculate degree centrality for all nodes, MapReduce is used. The MapReduce Java code is executed in Hadoop, which directly reads and writes the HBase data. In the map phase, the number of values stored in the adjacency table is read out and sent to the reduce phase in combination with the row key. The reduce phase takes the value and writes it directly into the storage system as seen in listing 18. To order the results based on the degree centrality, the degree centrality is taken as key in bytes format to enable the correct sorting.

[45]Hypertable database: `http://www.hypertable.org/`

[46]HBase Java client API: `http://hbase.apache.org/docs/current/api/org/apache/hadoop/hbase/client/package-summary.html`

```
public void map(...) {
    // Reads out the row key
    ImmutableBytesWritable mapKey = new ImmutableBytesWritable(values.getRow
        ());

    // Sends data to reduce phase
    context.write(mapKey, new IntWritable(values.size()));
}

public void reduce(...) {
    int degree = values.iterator().next().get();

    // Writes data back to db
    Put put = new Put(Bytes.toBytes("degree"));

    // Add degree as column name to sort based on it
    put.add(Bytes.toBytes("degree"), Bytes.toBytes(degree), key.get());
    context.write(key, put);
}
```

Listing 18: Degree centrality on HBase

PageRank is also implemented based on MapReduce with Hadoop. In the map phase, the adjancencyList from HBase is retrieved for every row and the partial values for every single node is sent to the reduce phase as seen in listing 19. In the reduce phase, all values for one row are summed up and directly written back to HBase. As seen in the code, all values are stored as bytes arrays.

```
public void map(...) {
    // Current pagerank value of node
    double pagerank = Bytes.toDouble(values.getValue(Bytes.toBytes("
        adjacencies:pagerank")));

    // Pagerank value sent to every connected node
    pagerank = pagerank / values.size();

    String rowString = new String(values.getRow());

    for (KeyValue key : values.list()) {

        String column = Bytes.toString(key.getColumn());
        column = column.substring(12);

        // Only send not pagerank values
        if (column.compareTo("pagerank") != 0) {
            ImmutableBytesWritable mapKey = new ImmutableBytesWritable(key.
                getValue());

            context.write(mapKey, new DoubleWritable(pagerank));
        }
    }
}

public void reduce(...) {
    double sum = 0;
    for (DoubleWritable val : values) {
        sum += val.get();
    }

    Put put = new Put(key.get());
    put.add(Bytes.toBytes("adjacencies"), Bytes.toBytes("pagerank"), Bytes.
        toBytes(sum));
    context.write(key, put);
}
```

Listing 19: PageRank on HBase

3.3.7 Cassandra

Cassandra[47] is a column-store and is implemented as a hybrid between BigTable and Dynamo. It uses a data model similar to BigTable and is based on the Dynamo infrastructure. Cassandra is written in Java and was published under the apache license. To access Cassandra, the Thrift framework is used. Other APIs such as a REST API are in development. It was first developed by Facebook and later open sourced. Since then it is further developed by the open source community and is used in different projects like Twitter, Digg or Reddis for different use cases.

Cassandra can be used with the same data model as HBase. Tables are named keyspaces and in addition to HBase, Cassandra supports so-called super columns. Super columns are similar to columns but have columns as values. Column families and super columns have to be predefined in the config file. To change the column family and super column structure, a restart of the server is necessary. Both types support different orderings as ASCII, UTF-8, Long or UUID.

Data in Cassandra can be accessed by simple key requests. In addition, range queries are supported based on row keys as seen before in HBase. Data processing is done through a Hadoop adapter.

Replication of data sent to the Cassandra is done based on eventual-consistency which puts it on the CAP Theorem triangle on the availability and partitioning side, in contrast to HBase. Cassandra is optimized for lots of nodes in one datacenter or datacenters that are connected with fibre channel connections and have low latency which 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. The data is distributed on the nodes based on hash tables as described in the Dynamo paper. Nodes can dynamically be added to the cluster and the data is automatically redistributed. Every node in a Cassandra cluster is the same, there is no root node.

For all implementations, Cassandra version 0.6.2 was used in combination with a Java client named Hector[48] version 0.6.0.17.

[47]Cassandra project: `http://cassandra.apache.org/`

[48]Java client from Ran Tavory `http://github.com/rantav/hector`

Data Model For reasons of simplicity, the exact same data model is used for Cassandra as is used for HBase, which can be found in listing 16. Super columns were not used because the useKit data can be modeled well without this additional structure level.

Query The degree centrality for a single node can be directly retrieved from the storage system, as seen in listing 20. First the keyspace – which in this example is useKit – has to be set. All adjacency nodes are stored in the column family adjacencies as described in the data model in listing 16. Then a count of the columns of this specific column family in a row can be done and directly retrieved as is shown for user-1. The retrieved value is the degree centrality.

```
keyspace = client.getKeyspace("useKit");
ColumnParent columnParent = new ColumnParent();
columnParent.setColumn_family("adjacency");
int degree = keyspace.getCount("user-1", columnParent);
```

Listing 20: Degree centrality on Cassandra

No special implementation was done for degree centrality over the complete data set and PageRank because Cassandra also uses Hadoop as data processing. The same algorithms as used for HBase can be applied.

3.3.8 Neo4j

Neo4j[49] is a graph database that is fully transactional and where the data is stored on disk. 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 in development.

Queries for Neo4j are created directly through its API, which makes it possible to create complex queries. In addition, SPARQL[5] and Gremlin[78] are supported as query languages. Neo4j offers a wide variety of additional components such as the graph algo component[50] which implements graph algorithms such as Dijkstra and makes it possible to directly use these algorithms inside the code. For visualization of the graphs, there is the Neoclipse plugin for eclipse which offers predefined visualization based on the stored data. A component for graphviz is under development.

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 still constant because the two nodes can be on two different machines, which requires additional communication.

For all experiments, Neo4j version 1.1 was used with the official Java client API [51].

Data model Because of its support for storing graph data, the data was stored as a graph in Neo4j, with content, context and user as nodes and the connections between them as edges.

[49]Neo4j project: `http://neo4j.org/`

[50]Neo4j graph algo component: `http://components.neo4j.org/neo4j-graph-algo/`

[51]Neo4j Java api library: `http://api.neo4j.org/current/org/neo4j/graphdb/package-summary.html`

A type for each of the three different nodes was created and six different edge types for every edge that can exist between the nodes. That means that every edge is part of a directed graph and exists two times (both directions). This could be improved by always detecting whether a connection between the nodes already exists.

Queries Degree centrality in Neo4j was calculated by counting all relationships of a specific node, as can be seen in listing 21. It was expected that there is already a counter inside every node for the number of relationships, but a function to read out such a value could not be found. To calculate degree centrality over all nodes and compare the values, the Lucene index of the nodes was used. The values were calculated for every single node and then compared and sorted on the client side.

```
for (Relationship r : node.getRelationships(Type.NODE_TYPE)) {
    userCount++;
}
```

Listing 21: Neo4j degree centrality

The PageRank algorithm was not implemented on Neo4j because there is no batch processing system for Neo4j. One way to calculate PageRank for every node would be to calculate it on the client side. Another possibility is over a connector to the Java Universal Network/Graph Framework (JUNG)[52] but it is not completely clear how this combination works. It seems as if it would be necessary to convert the Neo4j graph data to a general property graph model to make it accessible to Gremlin.

3.4 Visualization

To better understand the structure of a graph, a visual representation of the graph can help. The main goal is therefore to visualize the complete graph instead of a subgraph or communities to better understand the structure of the data. To visualize the graph with more than 50’000 vertices and 100’000 edges, different systems were evaluated. The main concern with most systems is that they were not capable of processing graphs of this size even though this is still a moderate size in comparison to other network data. Neoclipse[53] is one of the evaluated tools and is an eclipse plugin which allows to automatically visualize graphs stored in a Neo4j database. The plugin is not capable of processing and visualizing larger graphs in a short time. One main problem is that the system tries to animate the graph and also allows to change the graph in real time. Another library that was evaluated was Processing[54], which makes it possible to also animate graph changes over time. This means that for every point in time a data set is needed. Even though this library looks promising, the decision was made for Graphviz because of its speed and simplicity. It will be described more in detail below. All the visualization was made in two dimensions because no system was found that is capable of visualizing and representing data of this scale in 3D.

One more tool that looked promising was Snap [96] developed by Madduri. Problems occurred during compile time. SNAP would not have been used to visualize the existing graph,

[52]JUNG Framework: `http://jung.sourceforge.net/doc/index.html`
[53]Eclipse plugin to visualize Neo4j graphs `http://wiki.neo4j.org/content/Neoclipse_Guide`
[54]Processing project: `http://processing.org/`

but rather to analyze and partition the graph, which would make it possible to process and split up the graph and visualize only parts of it.

Graphviz Graphviz [85] was used for graph processing because it was capable of processing the large graphs and the usage is very simple. Graphviz is an open source project initiated by AT&T research. A wide variety of different algorithms are available for Graphviz in order to visualize different graph types:

- dot: Optimized for hierarchical or layered drawings of directed graphs
- neato: Spring model layouts. Minimizes global energy.
- fdp: Similar to neato, but reduces forces. sfdp is the multiscale version of fdp
- circo: For graphs with cyclic structures
- twopi: Radial layout

Currently, the only algorithm that is optimized for multiple processors is sfdp developed by Yifan Hu. Only two of the algorithms were considered to provide good results for the given data. The two algorithms that fit the given data best are neato and fdp and of course the multiprocessor version sfdp. The decision for these two was made based on the assumption that social data and also the useKit data does not have a radial, cyclic or hierarchical structure which excludes the algorithms twopi, circo and dot. It is expected that the data structure is more typical to small networks where the spring model and the force model show good results.

The neato and ftp algorithms offer a wide variety of additional parameters that were evaluated to improve the constructions of the graph. The creation of the DOT language [24] files that are used by Graphviz and the different algorithms were produced with Java code that fetched the data from the MySQL database and wrote it to the Graphviz file.

Graphviz offers gvpr as a filter for graph preprocessing. The usage of this tool was not necessary because data preprocessing such as filtering nodes without any connections was made in the creation script. An interesting feature that is supported by Graphviz is graph or egde clustering. There is a parameter to automatically combine multiple edges between the same nodes and a wide variety of parameters are available to influence the graph processing.

useKit data representation The basic DOT language file was created by Java code from the data in the MySQL database. Graphviz allows to set different metrics for every edge and node. Different methods were tried to create the file for receiving good results. Evaluations were made with directed and undirected graphs. For the nodes, different forms and colors were tried to make it visually appealing. For the different nodes the following colors were picked: Context (blue), user (red), content (green). All the lines between the edges are the same (black) and have no special attributes. It is necessary that every node in the DOT language has a unique id, which allows Graphviz to process the data. Thus, the node type name was prepended to the id, e.g. content127 for the content node with id 127 as seen before for the key-value stores. All parameters can be seen in the example dot file in listing 22.

```
graph G {
    concentrate=true
    outputMode=edgesfirst
    user1 [color = red, shape = box, label = u]
    user1 -- context8
    user1 -- context111
    user5 [color = red, shape = box, label = u]
    user5 -- context111
    user5 -- context16
    ...
    context11 [color = blue, label = c]
    context11 -- content238
    ...
    content238 [color = green, shape=point]
    ...
}
```

Listing 22: DOT Language file

3.5 Experiment setup

All installations and experiments were made on two different machines. One was a desktop machine with two Intel Pentium 4 CPU 2.80GHz processors, 2 GB of RAM and a 80GB hard drive. The system was running a default Ubuntu 10.4 server installation. The second machine was a MacBook Pro with an Intel Core 2 Duo 2.53 GHZ processor, 4 GB of RAM and 500GB hard drive (7200RPM). Hardware with more RAM was not available. For the various storage systems, the default setup and default configurations were used. For performance comparisons it would be necessary to configure every system for the specific use case to optimize the performance. For the file storage systems the max number of allowed open files and allowed processes was increased because most storage systems use a large number of open files at the same time. More details about the setup can be found on the wiki pages created for this thesis[55].

To process large graphs and have access to a larger amount of memory, an extra large EC2 instance from Amazon was deployed for a short time. This gave access to faster and a larger number of CPUs. In collaboration with the math institute at the University of Basel, access to a machine with 16 Intel Xeon 2.67 CPUs and 48GB of memory for fast graph processing was possible. This significantly reduced the time to visualize graphs. To display and analyze the svg files produced by Graphviz the ZRGViewer[56] was used.

[55]Thesis documentation with detailed setup: `http://ruflin.com/wiki/master-thesis:index`
[56]ZGRViewer, a GraphViz/DOT Viewer: `http://zvtm.sourceforge.net/zgrviewer.html`

4 Analysis and Results

In this thesis, the structure of social data and social networks was analyzed to compare the different types of social data. The useKit data as an example for social data has been modeled, stored and processed with different storage systems and was analyzed in detail to evaluate how adequate the different storage system are for social data. Every storage system has shown its advantages and disadvantages for the different processing methods. To better understand the global structure of the useKit data, the data set was visualized with Graphviz.

4.1 Data structure

The data structure of different social networks was analyzed based on the functionalities that the social network platforms offer. The useKit data set was analyzed in depth and will be described in detail.

4.1.1 Social data

Analyzing the data structure of different social networks reveals that the network structure is different for every network. The common denominator that can be found for all social data is that all consist of nodes and edges and can be represented as a graph. Every network can consist of different node and edge types. A differentiation between directed and undirected edges has to be made. An example of this can be seen by comparing the social graph of Facebook and Twitter where the users in Facebook build a friend graph which is based on friendships and where the connections are bidirectional, which means that the users are connected by undirected edges. The users in Twitter represent a directed social graph mostly based on interests which also leads to a different overall structure. The social graph in useKit is based on small collaborative networks and interest networks which again results in a different graph structure with lots of small clusters which is shown through the visualization of the data in section 4.4.

Social data differs from web link data in its overall structure and in how the graph changes over time. In figure 10 two small social networks with the same nodes but different edges in between are shown. The social network A consists of two highly connected communities which have few connections in between. The social network B consists of three highly connected communities which also have a few connections in between. The change from social network A to B over time is typical for social networks. The communities change over time, new communities appear, others disappear which means the edges between the nodes change. This is the same in the real world, where new friendships arise and if e.g. a person moves to a different place, some relationships are lost and new communities are joined. This altering overall structure of social networks is especially relevant for storage systems. To minimize queries over several machines, it makes sense to store highly connected communities on the same machine. But because of the changing communities this cannot be done without redistributing the nodes every time a community changes. If for the social network A in figure 10 the community on the left would be stored on machine one and the community on the right side would be stored on machine two, this would be ideal because there are only few connections between these nodes, which means little communication is necessary.

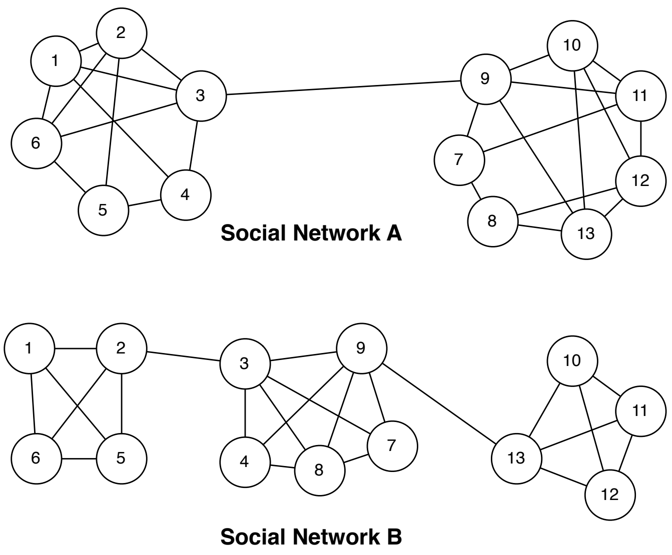


Figure 10: Two social networks with the same nodes

The social network A might change over time to the social network B where new communities were formed. As a consequence, the nodes of the middle community are stored on machine one and two even though it would make sense to store them on the same machine now. Because there are lots of connections between the nodes in the middle community, lots of communication has to be done between the machines which makes it inefficient. This means that special requirements are required for storage systems for social data. The altering community structure is what makes social data different from web link data where the data can be clustered by domains. Domains normally do not change over time. The assumption is made that most links inside a domain also point to the same domain and only a few links are external links to other websites. This is true in most cases. As always, there are special cases like link collections where most links are external links.

4.1.2 useKit Data

As described in section 3.1 the useKit data has three different types of nodes: context, content and user nodes. This makes it different from other typical social data where users are mostly connected directly with other users. This means that only one type of nodes exist. Modeling the useKit data is simple because all connections are known and there are only six relations in a directed graph and three relations in an undirected graph (user - context, context - content, content - user). This directly affects processing and the network analysis. More details can be found in section 4.2.

Even though this basic data structure of the useKit data did not change for the experiments, the real data structure on the useKit platform evolved. New types of edges were added such as direct user to user connections and additional social data such as comments for content objects were added. It is typical for social data that new features are added frequently to the social networks which change the data structure. Every time the structure changes, new processing methods have to be evaluated and the existing ones have to be adapted to the new structure and data.

Only the minimal node information was used for all calculations and processing. All additional information that exists for every node – such as first name and last name for the user nodes or context name for the context node – were ignored. This made the data much more compact for storage and processing.

Prior to the experiments, the structure of the useKit network was mostly unknown. Through the processing, analysis and visualization of the useKit data, a better understanding of the data set was achieved. As can be seen in the visualization section and in figure 13 useKit consists of many small communities that are highly connected. There are large connected clusters with highly connected sub-communities inside. There are no nodes that are central for the complete network, which means that every community has connections to these nodes. This can be explained by how the useKit platform is used, namely for collaboration in small teams. The content, context and user nodes are highly connected among the teams but only few data is important to the outside, which means only a few connections exist outside of the team. Public contexts are the only nodes that can attract lots of users and lots of content which would make it possible that public contexts, like the Music context, could get globally popular.

4.2 Network analysis

Different centrality measures were calculated on the useKit data to analyze the data set. The goal of the analysis was to find central nodes in the system. Only a limited amount of network analysis algorithms were used because of time restrictions. All the processing of the data was done on a snapshot of the data from the beginning of October. Since then, the original data set grew. The two well-known centrality measures degree centrality and PageRank were calculated on the data set. The different algorithms were implement on all storage systems. The results of the calculations were the same on every system as expected even though there were different methods that were used to receive the results.

4.2.1 Degree centrality

The degree centrality calculation was done for all types of nodes. Because of the different node types it was tried to calculate degree centrality with weighted edges. The reason for this was the assumption that a user or a context object are likely to be more important than a content object. It is expected that most users are connected to each other over contexts. This means that a user with many contexts should have a higher degree than a user with many contents. Comparing the values with weighted edges and without weighted edges, it seemed as if the relative values are quite similar. One explanation for this is that a user with many contents is also likely to have many contexts to organize their content and the other way around. Thus, degree centrality was only calculated with unweighted edges.

In this thesis, it was assumed that the degree centrality will not describe the central nodes very well and it will have a lots of false information. The reason for this is that no relations to other users are necessary to receive a high degree centrality. A user with lots of content items and contexts can have a high degree centrality without being important or central in the network. But as we see later, the degree centrality values matched the PageRank values quite well. This could be explained by the fact that in the useKit network there is nothing to gain for the users by having a high degree centrality – in contrast to Google, where it can influence the PageRank and thus it can directly influence how often a website is found. This means that there are no users that try to create as many connections to their own profile as possible. The useKit platform is used to store, retrieve and share data so that all generated content is somehow connected to the user and makes sense for the user. A user with lots of content and context will not gain more power in the useKit network. Instead, if a user creates lots of nonsense content, he will probably loose connections to others. The useKit platform is still in beta stage, which means that most users use the platform in the expected way. As soon as the platform grows this can change and attract also spam bots and other users that try to misuse the system. This could change the values.

In the top ten results for all nodes were seven user nodes, three context nodes and zero content nodes. The top users are users in the system that actively use the system in their everyday life, which means they create lots of content, create contexts to share the content and invite other users to collaborate. One of the top ten users was Nicolas Ruflin, the writer of this thesis. The degree centrality of this user is high in comparison to other users. This can be explained by two facts: He was one of the first people that was able to use the system and he also heavily tests the live system.

All three top contexts in the top ten are public contexts. One of these contexts is the context "Master Thesis Nicolas Ruflin" which was used to collect content for this thesis. This context is in the top ten because of its massive amount of content items. One other public context is the "Music" context which has – in addition to the other contexts – also lots of users. This leads to a high degree centrality value. It makes sense that all contexts with a high degree are public contexts. A public context can be joined by everyone, and as long as it is not set to read only in the system, all users can add content.

There were no content objects in the top 100 results. This can be explained by the fact that normally, content objects only belong to one to four contexts and are owned by only a few users. If a context has some content objects, the degree of the context is already higher than the degree of the content object itself because more connections exist. The main question that should be asked here is how an important or central content can be described. There is no real solution to that with the degree centrality where content objects will always be at the end of the rankings if they are not owned by lots of users.

In figure 11 four plots with the degree centrality of the top 50 nodes for every type are shown. In the top left graph the values for the top 50 users are compared. There are a few users in the system that have a degree centrality larger than 1000. These are very active users. The degree centrality drops fast and then the average user has a degree centrality of around 50. The same can be seen in respect to contexts. There are a few contexts with a degree larger than 100, but the average context has a degree centrality of 28. The peaks for the context objects can be explained by the power law. If a context is popular, even more users will join it and even more content will be added. For the content objects the values are much lower because it is rare that a content object belongs to many users or is in many contexts. In the top 50 content objects, the objects all have a degree centrality around 6, the overall average is around 3. On the right bottom, all graphs are combined into one graph. This makes it obvious that the degree centrality for the different node types cannot be compared. Every node type has to be treated separately.

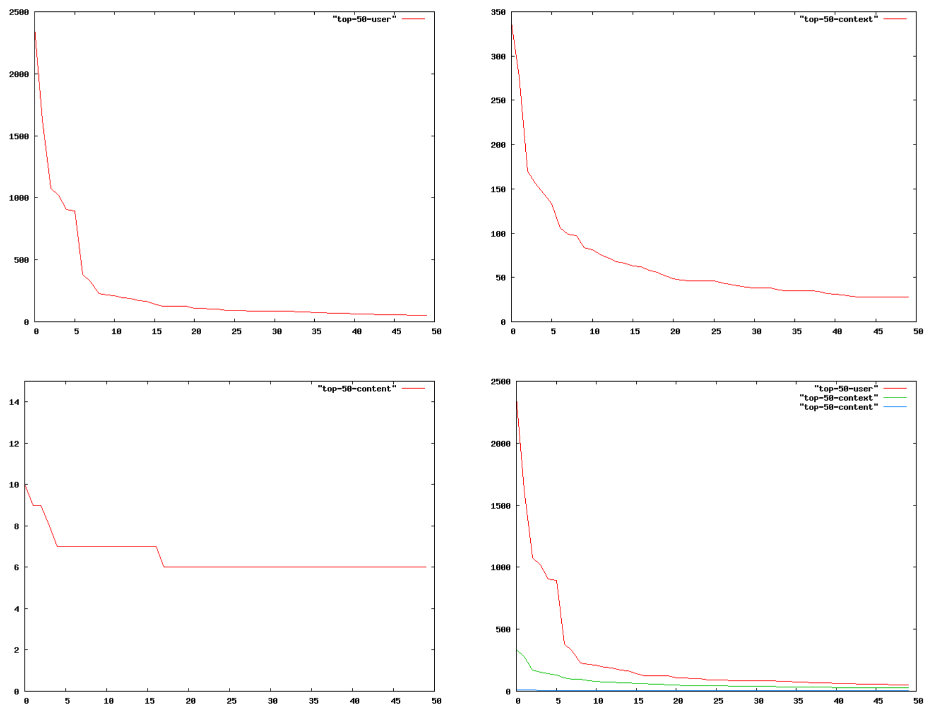


Figure 11: Top 50 degree centrality of the different node types

4.2.2 PageRank

The PageRank algorithm is a more complex algorithm that takes the complete structure of a network into account. It was expected that this algorithm delivers more relevant results than the degree centrality algorithm because it not only counts to how many other nodes a node is connected but also takes into account how relevant these other nodes are in the network. This fits the useKit data well, since the importance of a node cannot only be explained by the connections, but it is relevant how important the connected nodes are.

The effect of the PageRank algorithm on the useKit data is that local networks with high connectivity do not receive top values in the complete network. An example for this could be that three people work together and share much content and context, but they do not connect to the outside. Even if they have lots of connections, after several iterations they get a lower rating than a node that is in the global network and that is highly connected. This describes the main difference from degree centrality.

The useKit data with its 50’000 nodes and more than 100’000 edges is still a manageable size of data that also fits into memory. This is important for the PageRank algorithm because all intermediate results have to be stored. PageRank is an iterative method and already after four to five iterations most top ten nodes were under the top ten. The more iterations were done, the more accurate the results.

To calculate PageRank, the undirected useKit graph was converted to a bidirectional graph where every single edge is replaced by two directed edges. The discovery was made that the PageRank results are similar to the degree centrality results, but not equal. This similarity can partially be explained by the simplified version of the PageRank that was used. Hotho et al. already discovered that the PageRank does not fit undirected graphs that well and proposed an improved algorithm based on the PageRank for undirected graphs, especially for folksonomy networks, the so-called FolkRank [97] algorithm. The useKit data graph with its context nodes which are partially similar to tags is similar to a folksonomy graph. To improve the results, the FolkRank algorithm could be applied to the given data. This was not done anymore because of time restrictions.

Also with the PageRank algorithm, content objects do not appear in the top 100 rankings. With the PageRank algorithm this could be the case if a content object has connections to important users and context objects. Still it is not likely that a content object appears in the top rankings because of its few connections, compared to users and contexts. In order to find relevant or central content objects it could be necessary to only compare the values of the content objects with each other. Having an important or central content would mean this content is relevant for different contexts / subjects and is important for lots of users (owned by the users).

A personalized PageRank on the useKit data could be interesting for delivering relevant search results to the user. Every user receives the results that are more relevant for him, which means in this case data that is close to the user. A personalized PageRank can also be used for recommendations. There can be contexts that have a high PageRank seen from this specific user. If this user did not subscribe yet to the context, this context could be recommended to the users because he is probably interested in this context. But for recommendations, there are more relevant factors, such as closeness or access rights. In an implementation of a personalized PageRank it would be important to only analyze edges to objects where the user has access to. In the current implementations, all edges are analyzed independently of whether the nodes are globally accessible or not.

4.3 Storage systems

Eight different storage systems were analyzed for the storage, querying and processing of social data, specifically the useKit data set. The 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 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 have a clear advantage if querying for single nodes with specific attributes is necessary. The results for every storage system for data model, social data, queries and data processing, degree centrality, PageRank algorithm and scalability are described below.

4.3.1 Data model

The data that had to be stored was the same for all storage systems. It was given by the useKit nodes and their relations. The data was modeled differently for every system, as shown in section 3.3.

MySQL with its normalized data model, which is typical for RDBMS, allows to store data in a compact way. Every data entry is only stored once, whereas in other systems e.g. edges are stored twice or more. MySQL offers good capabilities for querying a specific node in one table. The structure had to be predefined with tables and columns. Changing the structure in big data sets can take a long time, which makes the systems inflexible.

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. Often, it is necessary to store a list of all ids in advance under one specific id to retrieve all ids later. For this, Redis also offers the $keys$ command, which makes it possible to retrieve all keys with a specific prefix. 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. In the useKit data this is the case for the relationships that are stored with every node. This means that every relationship is stored twice. Updating all data entries has to be implemented by the developer and is not supported by the storage system.

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. Also, the used data model is the same as for document stores. Less complex key constructs have to be used for Riak than Redis because lists and additional informations 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. However, this was not used because of limitations that exist with links like the maximum number of links that can be sent in the request header.

The two document stores MongoDB and CouchDB use the exact same data model. For both systems, the data can be modeled in JSON/BSON, which is also used in Riak. This makes it possible to model semi-structured data and every node with all its information. Furthermore, edge information is stored in one single document. Connections between two nodes are stored in both nodes and therefore in two documents. If an edge is removed, two documents have to be updated. Once again, this has to be done by the developer on the client side.

HBase and Cassandra as column stores allow to model the data in a semi-structured way similarly to the document stores and Riak. Every node was stored as one row and all additional information was stored in column families and columns including all edges of a node. In contrast to document stores, column families for both systems had to be predefined, which makes these systems less flexible for data modeling compared to document stores. One column family in every row was used to store all adjacency information, which made it possible to update and retrieve these values without affecting other values. Cassandra with its support for super columns allows to store data in an even more structured way and offers more specific queries then HBase. During modeling data for column stores it is important to know in advance

	Storage in KB
MySQL	1467
Redis	1800
Riak	38900
MongoDB	11182
CouchDB	6600
HBase	36100
Cassandra	23400
Neo4j	12700

Table 2: Storage size for 23000 nodes and 44000 nodes

what the queries will be so that the data is modeled the way that it can most efficiently be retrieved by queries.

Neo4j has the graph as a storage structure which makes it obvious to also model the data set as a graph. Because the useKit data is already graph data, no remodeling was necessary and the data was stored with nodes and edges. This makes modeling and storing data in graph databases intuitive.

Based on the data model, every system needs a different amount of storage for the same number of nodes and edges. A small comparison of the storage size was made for 23000 nodes and 44000 edges. The results can be found in table 2. The results are always measured in combination with the necessary indices. For every database, the minimal dataset with only edges and nodes was stored without any additional information. Data stores such as CouchDB and MongoDB allow to compress and optimize the data set. This was done before measuring the storage size. One of the main reasons that the size of the stored data differs in a factor of ten is that for example with documents stores, every key name has to be stored as well, in contrast to MySQL where the column name is only stored once. The size for document stores could be reduced by using shorter key names such as "c" instead of "context". The same is true for column stores. Riak seems to store by default more than one copy of the data also when running on a single machine which leads to a larger data size. For HBase, the differentiation between log files and real stored values was also unclear. It is not clear why the size of the Neo4j data is this large, because it is assumed that edge and node properties are stored in a compressed form. There was no optimized function called on the Neo4j index after adding the data. This could be one reason. In comparison, MongoDB and CouchDB data had more than five times the size before compaction. It seems as if both storage systems reserve space in advance to guarantee speed. The larger storage size is a tradeoff that is also made for better scalability and performance. Still, the size of MySQL exceeds the size of sparse matrix stored in compressed sparse row (CSR) format[28] based on the adjacency matrix or other optimized sparse matrix storage methods which are used in high performance computing. But it is not known how this kind of data can be scaled horizontally.

The same data was stored in all systems as mentioned before. Every system has different optimization options like bulk inserts to increase the number of writes per second. Also, inserting speed depends on whether the storage system directly writes the data to disk or first stores all values in memory which can also be configured in several systems. All data came from the

MySQL database, which limits the insert rate of all other systems by its reading speed. Thus, the performance of writing the data set to the different storage systems was not measured. The results would not be meaningful without making detailed investigation into optimizing every single system for the use case. To compare the performance in detail, the Yahoo! Cloud Serving Benchmark (YCSB)[57] can be used. This is based on the YCSB paper from Cooper et al.[66].

4.3.2 Social Data

For social data, storage systems with semi-structured data models have a clear advantage because of their possibility to scale and also because social data is often only semi-structured. Column stores are a good fit for lots of social data because column stores are in this thesis 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 like 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 kind of queries have to be used in order to choose the right system. As shown in this thesis, every data can be modeled for every system but it behaves differently for processing, high read or write loads or for simple queries.

Storing social data in MySQL is not a concern because in general, it is always possible to normalize data. Typical n to n relations can be represented with additional tables. The cost for writing data to the storage system stays constant because only a row has to be added and some indices have to be updated. One main concern for using MySQL with social data can be that the data set can change often, which means the structure of the database has to be changed. This can be a major disadvantage on MySQL because changing the structure with much data can take up to hours. Another problem of using MySQL for social data are the queries of n to n relationships. With social data, there are often queries that access several tables which means doing joins in SQL over several tables with large data sets. Because for some joins the complete data set has to be scanned, this becomes more and more inefficient with large data sets. An example for such a query could be "All content that is in context 5 and is owned by user 2". For this, a query on the content with two joins to the content_context and content_user table are necessary. These joins get slower with large data sets, especially when the intermediate results no longer fit into memory.

Redis is not a typical match for social data because it does not directly allow complex data structures or support queries. Because it allows to store any type of data that can be converted to a string as values for a key, it is possible to store JSON strings inside Redis. This makes it possible to use it similarly to Riak. The main problem is that there are no indices or queries that can be used to retrieve the data or even to process the data.

Riak fits social data well, especially because it is easy to scale to a large number of nodes. It is therefore possible to store large data sets and MapReduce queries are supported. The support

[57]Yahoo! Cloud Serving Benchmark project code: `https://github.com/brianfrankcooper/YCSB/wiki/`

for native JSON documents in every value makes it possible to use it partially like a document store and semi-structured data can be stored. In addition, links between keys are possible, which allows link walking which can be typical for some types of data queries in social data. One disadvantage of Riak for social data is that all queries are based on MapReduce, which means that queries are delayed and data cannot be retrieved based on indices or range queries. This will probably change in the future with Riak 0.13 and its Lucene implementation.

With the open data model of MongoDB and CouchDB, almost every type of social data fits the systems. MongoDB with its support for SQL-like queries based on predefined indices and additional MapReduce queries is an interesting match for social data. The same is true for CouchDB where queries are based on MapReduce but are updated in real time, which makes it possible to retrieve data without additional processing. Both systems have to make tradeoffs for scalability because scaling in both systems is based on replication and sharding. However, sharding is often a complex task for social data because of its distributed nature.

Column stores fit social data well because of its semi-structure data model and its ability to scale. 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. Both systems can scale to large data sets.

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 its scalability issues and because most queries are not graph traversal queries. Most queries in social networks are single node queries or range queries. Neo4j does not offer special indices for this kind 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.

4.3.3 Queries & Data Processing

In the implementation, there were no typical social data queries such as the content log on the useKit start page. The query to get this content log described in words is as following: Get all data that is in all contexts the user has and order it by date. For such queries once again a different evaluation of the systems would be necessary. These queries affect only a part of the data set an can be positioned between the simple degree query and the PageRank processing which affects the complete data set. One of the most often used systems for data processing is probably Hadoop, which directly supports MapReduce. But also additional languages such as Pig[58] or Hive[59] exist. More and more storage systems offer a plugin or adapters for Hadoop that allow to send the stored data to Hadoop for efficient processing, which is already done in Cassandra.

The evaluated systems supported four different types of queries, as can be seen in table 3: key requests, SQL queries, range queries and MapReduce queries. MapReduce queries are already data processing and would also allow to model range queries. Key requests are supported by all systems, whereas for Neo4j an additional Lucene index is necessary.

[58]Pig project: `http://pig.apache.org/`
[59]Hive project: `http://wiki.apache.org/hadoop/Hive`

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

Table 3: Query types with storage systems

Only the systems that support MapReduce allow real 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. HBase is powerful especially for data processing because of its direct integration with Hadoop hence MapReduce. This integration allows to directly read from and write to HBase inside the MapReduce queries, which makes processing and storing results efficient. Also distributing the processing is automatically done by the system. In addition, other Hadoop interfaces such as Pig or Hive can be used for data querying. Since release 0.6[60], Cassandra also supports processing its data with Hadoop. This makes it possible to run MapReduce queries on Cassandra data but because the data first has to be loaded into Hadoop, it is assumed that the processing is slower than in HBase.

It is interesting to see that there are different MapReduce implementations. CouchDB – where MapReduce queries are converted to views – uses a rereduce phase where only the values are sent without keys, which makes the implementation different from Hadoop where the keys always exist. The reason CouchDB uses this rereduce phase is probably to be able to update the views on the fly without reprocessing the complete data set. CouchDB also requires a certain reduce rate in the reduce phase, otherwise an error is thrown which does not allow to simply reorder the results in the reduce phase. Every system has its additions and small modifications to the MapReduce implementation like MongoDB, which offers a finalized method. Also, in some systems as Riak the reduce phase is optional, others require the reduce phase. This makes it necessary to adapt the specific MapReduce algorithm, e.g. PageRank, to every single system. The possibilities to debug MapReduce are limited to logging the output in the different phases, which makes debugging MapReduce much more complicated than debugging SQL queries. Also, because the exact output is not known in advance because of the large data set, it is hard to find small errors in the query code.

4.3.4 Degree centrality

Calculating degree centrality for single nodes was a simple task for all storage systems. Every system allows to pick a single node and all its additional data with a simple query or a key request as seen in table 3. The complexity of calculating the degree centrality stays the same

[60]Cassandra 0.6 release notes: `https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces3`

for all systems and also for growing data sets. To compare and order all the results, the storage systems which support processing the data with MapReduce had a clear advantage because on the other systems the processing had to be made on the client side.

MySQL performed well for calculating degree centrality, especially if the degree centrality was requested for only one node. To return degree centrality, only two indices have to be accessed and two counts have to be made. The response time and the complexity stay constant for such queries even if the data set grows. This is different for querying the top ten degree nodes. MySQL therefore first has to execute the two queries mentioned before for every single node, store the results in memory and then order the results. This has to be done every time a request is sent to the server, because intermediate results are not stored. The amount of work that has to be done grows linear with the amount of data. The time needed for processing also grows in a linear way as long as all intermediate results can be kept in memory. As soon as it is necessary to write intermediate results to disk, it gets slower by a large factor. It was tried to create a view based on the presented queries. However, in MySQL these views were even more inefficient than doing the queries directly because it seemed like MySQL does not use all indices correctly to calculate the results. As read this could be different with other SQL solutions like PostgreSQL or Oracle. MySQL does some internal query caching, but this was ignored in these experiments.

In Redis, degree centrality can be calculated quite simply because all links from one node to other nodes are stored in two lists. To calculate degree centrality for a node, only two count requests have to be made. Getting the degree centrality for all nodes is more processing and query intense because it has to be done for every single node and ordering would have to be done on the client side. If the top ten degree nodes would be needed it could be implemented the way that a sorted set is created with every node inside and the degree as the score. Every time a node is updated, the list has to be updated with the new values. Once again this would have to be implemented by the coder on the client side.

In Riak, MongoDB, CouchDB, HBase and Cassandra, degree centrality is easy to calculate for a single node because all data needed for calculating degree centrality is stored in a single value, document or row. With one request, the data can be retrieved and in the Java code the length of the adjacency array can be retrieved. For processing and comparing degree centrality of all nodes in all systems, the same MapReduce queries shown in section 3.2.1 are used. In these systems only the results have to be retrieved by the client and no processing is needed on the client side in contrast to MySQL, Redis and Neo4j.

Degree centrality calculation on Neo4j for one node is simple because every node knows all its adjacencies. Calculating degree for all nodes and comparing it cannot be done in the storage system and has to be done on the client side as with MySQL and Redis.

4.3.5 PageRank

For the PageRank processing, once again the systems which support MapReduce queries have a clear advantage because on the other systems, the calculations have to be done on the client side which leads to much communication between the storage system and the client because the client has to retrieve the complete data set. In addition, the client has to be capable of keeping all data in memory to process it, which is not possible for large data sets.

On MySQL, the PageRank algorithm was partially implemented for testing on the client side. All data was queried from the storage system and then processed by the serial PageRank algorithm. The implementation was made in Java. There are two options for the algorithm. One is to make the calculations for every separated node and write the intermediate results to a special storage place which also means that information for a single node has to be queried several times and at the end of every calculation, the intermediate results have to be written back to the storage. The second option is to load all data in memory and directly make all calculations based on the memory data. This is much more efficient but requires that all data can be loaded into memory which limits the size of the data. In general this does not scale to large data sets. The client side calculations based on these two ideas were only made for proof of concept but are not a solution for large data sets. It would be possible to implement the PageRank algorithm as a stored procedure in MySQL. But then it would still be necessary to call the stored procedure on every single row and it would be necessary to store the intermediate results somewhere. This was not done in the experiments.

On Riak, MongoDB, CouchDB, HBase and Cassandra, the PageRank implementation was done based on MapReduce. The basic MapReduce code is the same in all systems as presented in listing 4 but the parameters and capabilities of every system differ in the details as described before. The most powerful, simplest and most efficient solution is the implementation based on HBase in combination with Hadoop. Data can be read and written to the storage system directly in the MapReduce code and it is possible to add multiple reduce steps or to run multiple iterations with a single call to the system. Because the MapReduce queries are implemented in Java, the complete capabilities of the Java language can be used. Riak, MongoDB and CouchDB in contrast implement the queries in JavaScript which also allows almost all possibilities Hadoop offers but it would not be possible to access external data in some special cases as it is possible with Hadoop. This was used to create the output for the plots. There was no implementation for Cassandra, but the processing would be done in the same way as with HBase and Hadoop. Even though there were not made any performance comparisons, the performance of the PageRank processing was the fastest on Hadoop and was much slower on the other systems. All processing was done on a single machine which makes it impossible to judge how the different systems would behave and perform on multiple machines. One main disadvantage that was found in Riak is that there was no way to directly write back the data to storage from the reduce phase instead of sending it to the client first, which heavily degrades the performance.

4.3.6 Scalability

All installations and tests were made on only one or two machines. Because of this, no clear recommendation can be made which systems scales best in which cases, but assumptions are made based on the knowledge about the storage systems. Here, scaling means scaling to a large amount of data which is stored on several machines. It is important not to confuse it with performance. All systems are able to scale reads by replication. The less structured the data, the simpler scaling is where key-value stores are at the one end and RDBMS on the other end, as can be seen in figure 12.

As can be seen in table 4 the different systems support different methods to scale to large data sets. Replication is supported by all systems by which reads can be scaled and the data is

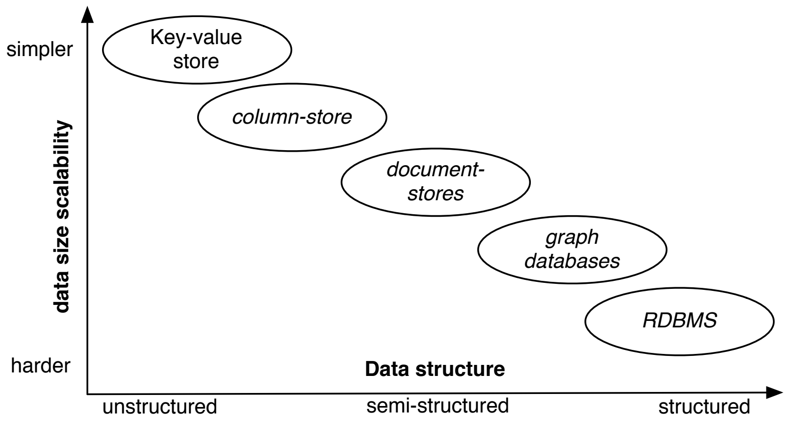


Figure 12: Scalability vs data structure

	Replication	Sharding	Partitioning	DHT
MySQL	+	+	-	-
Redis	+	-	-	-
Riak	+	-	+	+
MongoDB	+	+	-	-
CouchDB	+	+	-	-
HBase	+	-	+	-
Cassandra	+	-	+	+
Neo4j	+	+	-	-

Table 4: Supported methods to scale reads/writes on large data sets

stored with replicas which makes it fail-over safe. Sharding is for scaling large data sets and can be used for scaling writes. Sharding is only listed with a 1 in the table if it is supported by the storage system itself and has not to be done on the client side. The main problem that occurs with sharding is based on which values sharding is done and which entries are stored on which server. This is especially a complex task for social data because the connectivity of nodes changes over time, which is typical for social data as seen before. Based on this fact, large providers such as Facebook decided to store the data randomly on different nodes instead of trying to cluster connected data on the same node. In table 4 partitioning means automatic partitioning: Data is automatically distributed to new nodes if nodes join or leave the cluster. This is necessary to automatically scale up to large data sets without having to redistribute the values on different nodes every time more servers are added to the cluster. With Distributed Hash Tables (DHT) the data is randomly distributed to the nodes without having a master node that decides how the data is distributed and replicated. Cassandra is based on DHT but also allows preserving ordered partitioning to allow range queries which are often necessary. DHT systems are typically shared nothing system where every node can still operate even if the connections to some other nodes are lost.

Every system has its advantages and disadvantages in scaling. It is important to know if only reads or also writes have to be scaled. Redis is limited in scaling to large data sets because the performance of Redis is based on the assumption that all data can be kept in memory. Sharding to different nodes is not supported yet, but should be added in one of the next releases. HBase uses a single master name node to manage the different nodes which adds on single point of failure, but HBase allows to have multiple master nodes which means replicas of the name node exist and an automatic fail-over can happen if the master fails. Cassandra on the other side does not need a master node because all nodes are equal and new nodes are discovered automatically. There is no single point of failure. Cassandra scales better for writes than HBase because of its eventual consistency. This makes it possible to still write to the system also if some nodes are not available but can lead to inconsistent data that has to be merged. One main disadvantage of Cassandra is that when the data structure of (super) column families changes, the cluster has to be restarted to support the new data structure. There were not made any detailed performance measurements but all systems could be improved in speed by improving and adapting the configuration to the specific use case.

4.3.7 General survey

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 supported yet.

As a key-value store dynamo implementation, Riak makes it possible to scale up to large data sets with no single point of failure and automatically scales by just adding nodes. The number of nodes used for a read or write can be chosen separately for every request. 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 on more than one node as all similar distributed systems. 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. In the most recent version (0.13) of Riak 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 throw almost any kind of data into the storage. MongoDB offers a query language similar to SQL (only on not sharded systems) 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 enhance 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.

HBase and Cassandra with the column model as storage structure offer a combination of key-value store and document store. It allows to store semi-structured data but still allows range and filter queries. The data is distributed based on partitioning or DHT. Because of its consistency, HBase is better in high read rates and Cassandra in high writes 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. When Cassandra was published by Facebook, it was one of the first scalable NoSQL data stores that were open source. Because of this a strong company grew around Cassandra and there was a lot of buzz around this project. Still it is under heavy development. But in the meantime other interesting projects were published. It is difficult to predict in which direction the development of Cassandra goes in the future but probably it is going to stay one of the key players in the NoSQL market.

Neo4j is under heavy development and one of the next large community goals is to make it more scalable. Neo4j allows to represent network data 1 to 1 in the storage structure which makes it intuitive to work with graph data. The various components and extensions make Neo4j a powerful solution for all graph data where graph traversal is a key issue. Because of its scalability issues and lack of range queries there is no large adoption of graph databases for large data sets yet. Indices and search queries based on node values can be provided by adding Lucene indices.

4.4 Visualization

Creating valid dots files was done through Java code directly from the MySQL database. Updating the data if a node or edge changes cannot be done without querying the complete file. Because of this every time the graph changes the DOT file has to be recreated. Processing the created DOT file was done with the code seen in listing 23 which writes the graph into the file in the SVG format.

```
dot -Tsvg graph.dot -K neato -o graph-neato.svg
dot -Tsvg graph.dot -K sfdp -o graph-sfdp.svg
```

Listing 23: Graph creation command with neato/sfdp

The neato and the sfdp algorithm offered the best visual results for the useKit data. The spring and the force model approach seem to offer the best results for this kind of social data. Different parameters were tried to optimize the visual output. The parameters which can be found in listing 24 returned the best results. The parameter model=subset is an optimization for large graphs developed by Gansner et al [98] based on stress majorization. The parameter packMode=clust tries to cluster subcommunities, outputMode=edgefirst draws the edges first and then draws the nodes. Otherwise many nodes could not be seen because an edge is on top.

```
graph G {
    model=subset
    outputMode=edgesfirst
    packMode=clust
```

Listing 24: Graphviz params for neato and sfdp

Other parameters such as $overlap = false$ were tried out but this leaded to memory overflow errors. It is also possible to choose different start nodes but with the size of the graph data no real differences were discovered. It was also tried to show some special nodes like the user Nicolas Ruflin or the context Master Thesis. Even though a special form was added to these edges, they cannot be discovered in the visual output. Another problem occurred when trying to convert svg files to png. With some large svg result sets this also leaded to out of memory errors.

The output of this processing can be seen in figure 13. This output shows the complete community structure of useKit. There are some large and heavily connected networks in the center. Between these nodes, there are smaller clusters with several users that share contexts and content items. This structure is typical for useKit and represents how useKit is used. Small groups work together on a project, produce lots of content and share it over contexts. Often inside companies several such groups exist with overlapping users and contexts. Together, these groups form the clusters that can be seen in the figure. On the outside are singular users that probably signed up and do not use the useKit platform anymore.

The time needed for creating one visual representation as SVG file was not relevant for this thesis. The visualization was made of one snapshot of the data set. Most of the processing algorithms for Graphviz except sfpd support only one processor, which makes scaling the processing to large graphs difficult. Throwing lots of memory at the visualization task helps because visualization is much faster as long as all data and intermediate results fit into memory.

One of the main restraints during processing of large graphs was the available memory. For efficient processing, the whole graph has to be in memory for fast access from the processor. The test environments with 2 and 4 GB of RAM were not sufficient for many algorithms and some specific parameters. As soon as the data has to be stored to disk, processing gets much slower. With the growth of fast SSD and growing space of fast memory this will be less of a problem in the next years. One solution that was used to solve this problem was renting extra large EC2 instances for a short time period which offers 15GB of RAM and 8 EC2 compute units. Even larger instances with up to 68.4GB of memory exist. The access to the machine at the math institute with 48GB of RAM first made it possible to also process and visualize the most recent data with more than 100'000 nodes. Even though most processing was done on a single processor, the processing time was reduced from more than one hour to several minutes, which showed how important the amount of RAM is for efficient graph visualization.

A possibility to get better visualization results is to shrink the data to only one cluster and then create the DOT language file based on this data to visualize only one cluster in order to get a more specific view of the data. If all data is connected without separated subgraphs, algorithms for detecting community structures are needed. This can be an expensive task.

The data was processed without time constraints. This means it was ignored when a specific connection between two nodes was created. If a user created lots of content and contexts about a year ago and is now not active anymore, he can still appear on top of all users even if he is no longer active in the system. For better results a time factor should be used that only shows active data or data that was changed or created recently.

Visualizing data well is a time-consuming task. Tools like Graphviz are helpful to visualize large graph sets and also to deliver good basic results. To improve the outputs, lots of parameters can be set. The output after processing cannot be predicated completely for large graphs, which often makes it a trial and error approach. Improving automation of graph visualization would lower the entrance hurdle for lots of other research areas to visualize and better understand their data sets.

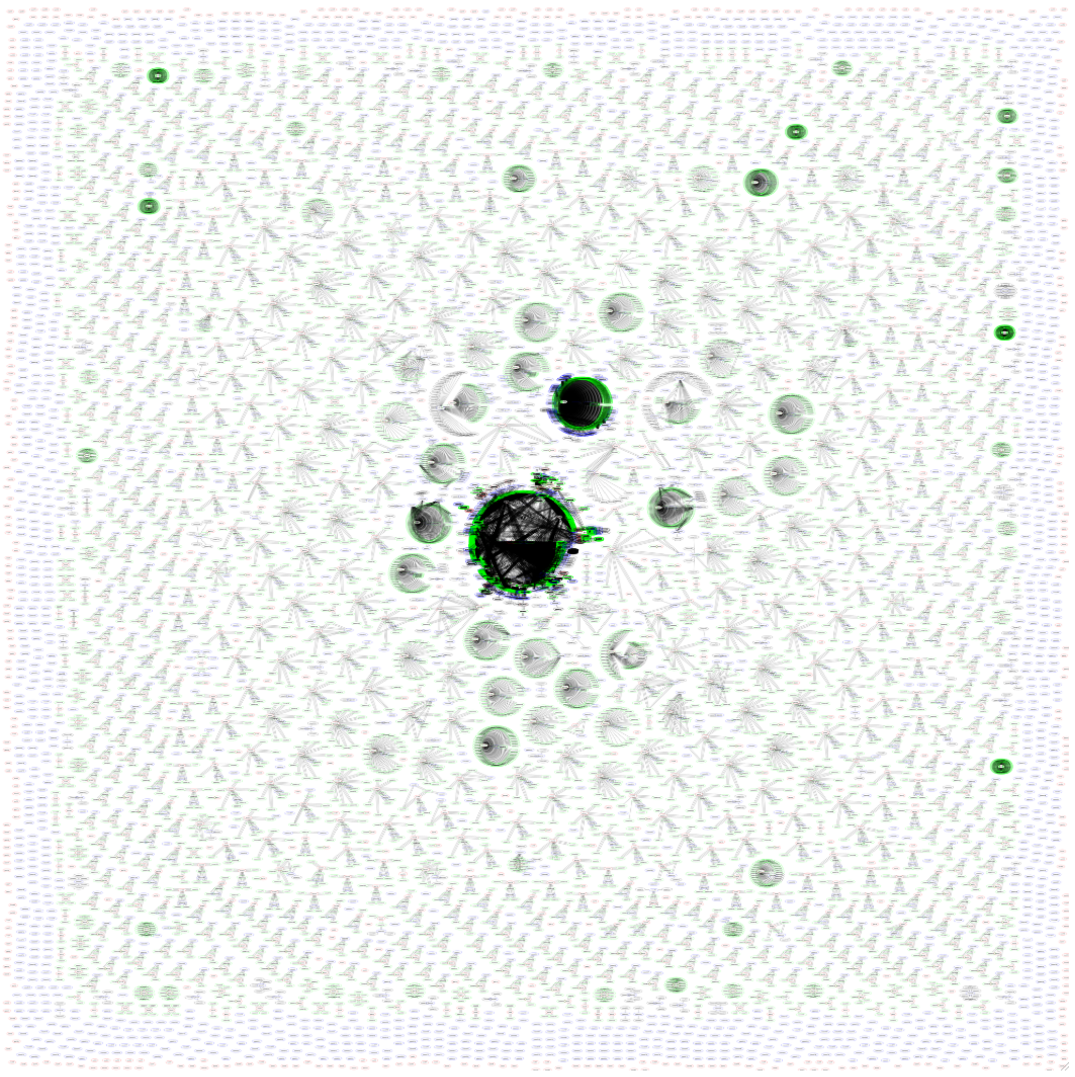


Figure 13: useKit Data visualized. Show seperated highly conntected local clusters.

5 Discussion & Future Work

This section will be concerned with the specific results from the analysis and results part. In addition, some ideas for future work will be discussed for every research area.

5.1 Data

The main challenge with social data is its structure and its size. The structure of single nodes varies from use case to use case, but also the global structure of every network is different. Every social network is unique and to analyze a network, first the structure has to be understood. There are approaches to standardize some representations such as friendships with FOAF. This should also help to make data portable from one platform to another. Much more important to exchange data are the APIs that are offered by most platforms which are not standardized but allow to access the data from a social network in a simple way.

Even if the structure of social data changes and exists in various forms, the basics stay the same: Nodes are connected through edges. New types of social data such as location data from mobile devices but also sensor data, which will also be part of the social data stream, grow fast [58]. The amount of data that is produced daily already exceeds the amount of data that can be stored, and the gap between created data and storage capacity will grow larger over the next few years. Processing the data efficiently and only storing the relevant parts will get more and more important and in general the importance of graphs will rise.

Because there will not be any predefined structure of social data and the amount of data will grow, this poses special requirements on the storage systems. There will be specific and optimized solutions for every social network and probably less standardization because performance is more important and standardization always means making some tradeoffs.

The analyzed useKit data set is only a specific case. As it was already discovered with the useKit data set, the structure changes over time and new types of edges and nodes are added to the data set. This often happens when new features are added to a social platform. More social networks should be analyzed based on the node and overall structure and some classification schemes should be developed to classify the social networks into different groups to find out if some general structures exist.

One main challenge with social data is the querying of social data. This is connected to the structure of social data, but was not discussed in detail in this thesis. Typical news feed streams as can be found on the Twitter or Facebook start page were not discussed in this thesis. These streams consist of data from connected nodes and poses a real challenge to storage systems in order to retrieve this data. It would be interesting to do more research in this area to find out how these feeds can be created in an efficient way.

5.2 Storage systems

Platforms such as Facebook or Google are already capable of storing and processing an impressive amount of data. Facebook has more than 500 million users, only a fraction of the users is online at the same time. There are no web applications yet that allow a large amount of the world population to interact at the same time. This world scalability will become more and more important as more and more users are online at the same time all around the world with their mobile devices.

Five different types of storage systems were evaluated. There is no single system that is best in all parts, since every system has its advantages and disadvantages. This is typical for so-called NoSQL solutions because most systems were developed with a specific use case in mind. A question that is raised frequently concerns standards for NoSQL solutions, for example for a query language similar to SQL. It is assumed that there will not be any query standards that can or will be adopted by most of the systems. One reason is that most of these systems were built because other systems were not good enough in a specific task, so these systems were built to solve these tasks best. Standardization often has the drawback of loosing some performance, which is the opposite of what these systems were built for. One standardization that is already part of most systems is the access interface. REST is chosen in most systems as one of the access methods. This makes it possible to use the systems from all kind of client languages and to use all these systems in service-oriented architectures and over network connections. NoSQL solutions do not directly compete with traditional SQL solutions. The discussion if NoSQL replaces SQL is needless. There are needs for both kinds of systems. The goal is to pick the best system for every task. This can also mean that one platform uses different systems for different data sets. NoSQL solutions are not yet as stable as SQL solutions. The reason for this is that most systems are still in early development and are in alpha or beta stage. It will take some more time to mature these systems. At the moment, a new system appears almost every week in the community. In the long run probably only the best and most stable system will survive.

One main concern with most NoSQL solutions is that there is no or only limited access management. This can be a security issue. However, most systems started to implement rudimentary access management; HBase can be taken as an example. But there won't be the same fine-grained access management like in most SQL solutions based on tables or even rows because checking access rights for every read and write can be a scalability issue in large systems.

An interesting new movement can be seen in CouchDB. CouchDB can be used on mobile devices and can distribute data over several mobile devices or sync the database between different systems. This indicates that in the future, mobile devices will probably also be part of the global storage cloud. This poses new problems such as high latency or disconnectivity to storage systems. This approach is sometimes also mentioned as the ambient cloud [99]. Already now, most scalable implementations of data storage and data processing are based on commodity hardware that is scaled up to thousands of nodes across different data centers instead of using super computers. This approach will even be extended by the ambient cloud where resources could be used and shared across the globe, including mobile devices.

Social data has a diverse structure and the actual need differs heavily from use case to use case. The selection of the storage system depends heavily on the actual problem. The requirements such as data size, consistency, availability or how the data has to be queried, processed or searched have to be known in advance to make a decision for the right system.

Every system had its specific advantages and disadvantages in storing the data as was seen in the results and analysis section. Column stores seem to be one of the systems that fit big data or social data best. Column stores are a combination of document stores and key-value stores. The data structure has to be only partially predefined and data is accessed directly through keys or range queries. No complex query systems exist. Column stores are built to be scalable. HBase with the HDFS file systems as backend, Hadoop for data processing and several extensions such as Hive and Pig to query the data, this seems to be one of the most powerful systems to scale and process big data at the moment.

The main concern with SQL solutions is that scaling is more difficult and the systems are not built to process large data sets. As soon as large data sets have to be stored, several servers or complete data centers are involved. This means that nodes will fail on a regular basis and replication of all data is necessary. Cassandra offers a scaling structure where every node is exactly the same and no central master exists which means no single point of failure. This makes adding and administering nodes simple. On the downside, this also means that there is no central point of control.

Having different storage systems for the same data can be a big advantage as was experienced during the experiment. To process data, it is nice to have systems such as HBase that support MapReduce queries. Doing graph traversal in such a system would be difficult. Having the same data also stored in Neo4j makes it simple to write some graph traversal queries. In real implementations it is not applicable to always have all data in different storage systems. But it is worth thinking about having importers for other systems to process and query the data in other systems instead of writing complex systems to process the data as is necessary.

Another interesting aspect would be to compare the performance of the different systems. However, doing good and relevant benchmarks is a difficult task and was – due to lack of time – not part of this thesis. Benchmarks always have to be done with a specific data set and its requirements. To choose a system that fits the given data and requirements, deep understanding of the data is necessary and it has to be known how the data will be accessed. Some systems are better in inserting/writing data, others in reading. Easy insert does not necessarily mean easy out which is typical for key-value stores where adding data is simple but it has to be thought in advance how to retrieve the data again. Basic questions need to be asked such as: Is it more read or write oriented? Should availability or consistency be chosen? Picking the right systems is a long process. There is not one system that fits all needs. Always the system that fits the problem best should be chosen.

There are strong communities around these new storage solutions that are lead by the highly skilled developers of the storage systems. This often leads to interesting discussions where the systems are compared and where it is seen that similar problems were often solved with different approaches. Because there is also a hype around these systems, a lot of blogging is done where also false information or outdated information is posted. This sometimes leads to confusion in the community. Especially subjects such as the basic CAP theorem is a subject that is often discussed and a lot is written about.

Having the same data in different storage systems makes it possible to always use the most efficient system or the simplest one to implement a certain task. Selecting and evaluating single nodes is simple and efficient in MySQL. This makes debugging and searching single nodes simple. Batch processing with MapReduce is implemented well in HBase/Hadoop which made it simpler to write this code for HBase first and then adapt it to other systems. There are always different problems that have to be solved for one data set. This can make it reasonable to store the data in the different storage systems.

Even though there already is a fair amount of storage systems, more research should be done in this area to create even 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.

5.3 Processing / Analyzing

In this thesis, the aspect of retrieving items from a database and processing the data after retrieving was looked at. One important case that was not discussed is searching inside the data. The de facto standard in this area is Lucene. Searching through data sets once again poses different requirements on the storage systems. Data is analyzed first and then stored in a form that it can be searched easily. Most of the systems mentioned before have plugins or export scripts to send data directly to Lucene.

Different types of centrality were calculated for the different types of vertices. First, degree centrality was calculated. For this multidimensional type of data, it was clear that the degree centrality would only give a basic idea about a vertex. For a user it describes how active this user is in creating content and creating / subscribing to contexts. But it does not describe the role or position of the vertex in the overall network. The reason for this is that to evaluate the position or role of a vertex with the given data, at least the next node is relevant. It is possible that a user created 100 content items and 10 contexts without having a connection to any other user over a context or a content item because all contexts are set private.

The PageRank algorithm is useful in directed graphs data sets like web links, but has its limitations for undirected graphs. The results delivered by the simplified PageRank algorithms are still relevant but could be improved by making modifications on the algorithm in order to better fit the undirected graph and the useKit data set like the FolkRank [97] approach. Facebook – as one of the largest networks with an undirected graph – has not yet revealed any algorithms about the calculation of the centrality of a person in the network, and especially its feed creation, which is a combination of different items from connected users. Which exact items are shown is based on algorithms which Facebook once called EdgeRank[61]. Still a lot of research is needed in the area of centrality measures and undirected graphs in order to provide better centrality measure for undirected graphs or even graphs that consist of a combination of directed and undirected edges. There do not exist any algorithms similar to PageRank which deliver relevant results for graphs that have both types of edges, directed and undirected ones.

The useKit data set was processed for degree centrality and a basic PageRank algorithm. The results of both algorithms were similar and showed the central nodes in the graph for each type of node. All node types in the useKit graph were treated equally. Even though some experiments were made in weighting the edges differently between the different edge types this was not treated concludingly. More research in this direction should be done to evaluate how the different node types and edge weights can affect the graph processing.

For all this processing, only the basic information was used. The useKit data would offer more factors based on which a more fine-grained processing would be possible. For content items, the number of views and additionally the number of comments could be taken into account. For a context node, the number of views could be added. In the original data set, a user is also directly connected to other users which could be added to the processing info. To

[61] f8 developer conference video mentioning EdgeRank: `http://apps.facebook.com/feightlive/`

process this information, advanced algorithms would be needed. This would probably deliver more fine-grained results.

The processing of the nodes was made independently of the time in which a node or an edge was created. In social networks, information often depends on its actuality. So the date of the nodes and the edges should also be a processing factor in future experiments. This additional information would also exist in the original data set. Interesting and more relevant results could be retrieved by extending the PageRank algorithm also with date information, whereas older information would be less relevant.

All the data was processed based on a global view on the graph. By processing local subgraphs, recommendations could be made to show a user contexts that he could be interested in. Another possibility is to recommend users with similar interests which are in the same context but to which the user does not have a direct connection yet. Identifying similar contexts could lead to context recommendations for content items. For this, different processing methods would be needed that were not evaluated in this thesis.

There are different approaches to data processing, such as incremental processing or batch processing. Until recently, large data processing was mostly done with the de facto standard MapReduce. After updating parts of the data, it was necessary to reprocess the complete data set. It was already known that this is not the most efficient way because the same data has to be reprocessed several times. With the release of Google's live search and the Pregel paper [12], Google showed that there are already new methods of how large data sets can be updated in almost real time without having to update the complete data set. Sensor network data will grow and already more data is produced than can be stored. This makes it necessary to process the data in real time and only store the relevant parts. Data processing is a subject that is closely connected with data mining and machine learning which makes it interesting in lots of other areas. New systems such as S4 from Yahoo[62] that is based on streaming data try to lower the entrance barrier to make data processing also available to a broader target group. The same is done with alternative processing languages for Hadoop like Hive that offers more and SQL style of writing queries, or Pig as an alternative that might also simplify the writing of processing queries.

Scaling the processing of large data sets is not easy. There can be different bottlenecks. It can be the storage system, network connections or the processing itself. One general discovery is that it is tried to move the calculations closer to the data which is known from stored procedures in RDBMS. But these calculations are horizontally scalable to not become a new bottleneck. At the moment, the most scalable processing system seems to be Hadoop, which scales up to thousands of nodes.

One of the next steps in data processing is probably to distribute it even more and to use all kinds of idle devices. Mobile devices such as smart phones or tablets have more and more processing power that could also be used. One of the first approaches that showed how calculations can be distributed was Seti@home[100] that used millions of home computers to analyze data. Wuala[63], a swiss start-up that makes it possible to share own disk space to get more web space, already shows how it can be done for storage space with different kind of devices. The same could be done with processing. A user can let his own processing power to

[62]S4 distributed stream computing platform: `http://s4.io/`
[63]Secure online storage: `http://www.wuala.com/`

others that need it [101]. This would make large amounts of processing power available that could be rented when needed.

A new area of research is graph similarity and graph comparison. Several graph similarity algorithms are based on PageRank or other centrality measures. One of the goals is to find out which communities are similar to others and to describe what that means in detail. There are many other appliances of graph comparison in biology, medicine, chemistry and other areas. To calculate the results, efficient graph preprocessing is needed based on some algorithms as PageRank shown in this thesis. Once again, for better graph comparison improved centrality algorithms are need that also deliver relevant result for undirected or mixed graphs and can process nodes based on time values or other specific needs.

5.4 Visualization of Graph Data

Visualization is a complex subject and research in this area has just started. Creating good output images requires a lot of processing power and because it is not only based on algorithmic logic, user input and tweeking the parameters is needed to get good results that are understood by people. Visualizing results and data often helps to understand complex results and data sets.

Different approaches were used to visualize the useKit data. The current results is a global view on all data that differentiates between user, content, context by color. This helped to understand the structure of the useKit network. It would be interesting to visualize only subgraphs or sub communities that are highly connected in order to better understand the local network structure. For this, preprocessing with community detection and subgraph separation would be necessary. Also interesting would be to show some special nodes in the graph such as the most popular or relevant users and contexts. This was tried but no helpful output results were produced. Another subject that is popular at the moment because of the 3D cinema would be to visualize the data in a 3D cube, which makes it possible to fly through the data. Most existing algorithms are optimized to create two-dimensional data. The used formulas in graph visualization are from physics, for example the electrical spring model or the minimal force model. The assumption can be made that these models would also work in the three-dimensional space.

All data was visualized independently of time. Thus, a content that was created a year ago is shown exactly the same as a content object that was created recently. The assumption can be made that data which was not updated in a year is probably less relevant than new data. One possibility could be to fade out such data points more and more with time or move them in the background. Such time-resolved network studies are also called longitudinal studies and are researched by different research groups at the moment.

The useKit data was visualized at one point in time. It would be interesting to show how the network changes over time. This could be done because a copy of the complete useKit data set exists from every six hours. However, there are several problems. First, a lot of processing power is needed to visualize every data set and then to create a movie out of it. A second problem could be that the visualizing algorithm decides to visualize the data completely differently even if only one or two nodes change. For people this would look like a large jump in the data set. So there should be some constraints that single data points do not move too much between every iteration. But to do this, it is necessary to know the previous results and to understand what node was where. A next step would be to visualize the data in real time.

The data set is continuously growing, which is why visualizing the data in real time with the current algorithms and processing capabilities is not possible. More research is needed in the area of live processing data and direct visualization also for large data sets.

Visualizing data is a complex undertaking and a deep understanding of the data is necessary before processing and knowledge in the area of visualize algorithm. But visualizing large data sets should be available also to other areas where these requirements are not met. One first step in this direction is ManyEyes from IBM. This allows people to upload own data sets and visualize these data sets. Still, a lot of research has to be done so that data sets are automatically understood by the algorithm in order to set the right algorithms and optimization parameters. This touches also the research area of machine learning because computers have to learn the human understanding of a pleasant graph.

6 Conclusion

This thesis was concerned with analyzing the structure of social data and discovering which storage system fits which type of data best on the basis of concrete implementations. In the course of these implementations it was discussed how the data can be processed and analyzed in the respective system. Furthermore, the data set was visualized.

Social data is semi-structured and the structure often changes. Social networks evolve, new features are implemented and new devices such as mobile phones feed data with different structure into the social network. This makes it necessary to have adequate storage systems which are capable of evolving with the growth and changes of social networks. To better understand the data, centrality measures are necessary in order to identify important nodes in the network. Through visualization, a complete view on the data set can be produced which gives interesting insights into the graph structure.

The smallest denominator that is shared by all social networks and social data in general is the possibility of representing them as a graph. All social data consists of nodes and edges. Depending on the network, the edges are undirected or directed and they can or cannot have weights. Often, different kinds of nodes with additional properties exist. All these nodes are connected by edges. Most social data sets are different from web link data sets. Web link data sets are directed graphs and because of their domain structure it is possible to cluster the data of one domain closely together in the storage system. This is mostly not possible for social data because communities that could be similar to domains change and evolve over time. Some communities break up and new communities with different connections are built. This makes it almost impossible to store users with similar communities close to each other because their connections change over time. Thus, social data poses special requirements to storage systems.

The problem of finding the right storage system for social data has not yet been solved. There is no general solution for social data because, as mentioned before, the structure of every social data set is different and also the requirements of the different social networks are different. 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. This could change in the future with more scalable graph databases. The main problem with RDBMS or MySQL is that social data is semi-structured. The data structure in RDBMS therefore has to be changed every time new data types appear. JOIN queries over large data sets are one of the main constraints of using RDBMS for social data. Executing JOINs over data sets stored on several machines is inefficient. Furthermore, RDBMS are not able to process complete data sets. However, for social data sets this is often necessary. 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 problem. 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 store and model data for document stores but because of its semi-predefined data model, column stores are easier to scale because the storage system knows more about the data structure in advance. Column stores only allow range queries based on keys. This makes it necessary to model the data the 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 same data is often stored multiple times and the data set has to be kept clean by the programmer instead of the storage system. However, this tradeoff has to be made in order to make it more scalable.

The useKit data set was analyzed in detail, modeled and stored on all eight evaluated storage systems. Also, the degree centrality and a simplified PageRank algorithm were calculated. The useKit data set consists of three node types (context, user, content) which are connected by undirected edges. Thus, it is similar to a folksonomy network. Based on the useKit data set it was shown that it is possible to model social data for all eight storage systems, but the structure that the data is stored in is different for every system. This again leads to different queries. The storage size for the same data set differed heavily between the solutions because in the NoSQL solutions, some data was stored twice and for example in document stores also the key to a value has to be stored. Degree centrality and PageRank were calculated based on the useKit data set. This delivered relevant results for the data set. The results for degree centrality were similar to the PageRank results. The reason for this is the fact that the PageRank algorithm was simplified and had to be modified to handle an undirected graph. PageRank and pseudo-directed graphs deliver similar results as the degree centrality. The results made clear that for further investigation of the useKit data, a differentiation between the three node types is necessary. Content, context and user nodes should not have the same weights because context and user have more relevance in the network.

All storage systems were capable of calculating the degree centrality for a single node in a simple way. Processing the complete useKit data set for comparing degree centrality or PageRank calculation showed the main constraints of pure key-value stores, RDBMS and graph databases which do not offer an internal processing system. For these systems, the processing has to be done on the client side which requires much communication with the server in order to retrieve and store the data after processing. This makes it impossible to process data sets that cannot be processed by a single client. Column and document stores or enhanced key-value stores which offer MapReduce as a processing system and allow processing on the server side with direct access to the data. This makes it possible to scale the processing to several nodes. Because the current implementations were made on a limited data set of only 50'000 nodes and around 100'000 edges calculations and processing was possible on a single machine. It would be interesting to do further experiments on a larger data set that is distributed to several machines. Because of these limitations, no benchmarks were made. For retrieving relevant benchmarks it is necessary to scale the solutions to several nodes because the main advantage of distributed MapReduce queries are their capabilities in distributed systems. In further experiments, more detailed benchmarks might be significant.

The current standard solution for data processing is MapReduce in combination with Hadoop. New solutions were presented recently by Google with Pregel. These new solutions are focused on updating only local parts of the data set instead of reprocessing the complete data set every time new data is added. Currently, lots of research is done to improve the MapReduce model by minimizing communication between the different processing units. Processing large data sets will be used in more and more areas. To store, analyze and process these large data sets, parallel computing is indispensable. This is currently an emerging subject in research. This can be shown by the number of papers that are concerned with these subjects and that appeared in the last few years. The number of connected machines and CPUs is steadily growing. More research in this area is needed, especially to make data processing available to a wider group of people that also have to deal with large data sets.

The complete useKit data set was visualized with the visualization tool Graphviz based on the created DOT-files which was created based on the preprocessed data. The different node types were visualized with different colors. The resulting image revealed new information about the network structure. It clearly shows that the network has lots of small communities with different node types which are heavily connected. The useKit platform is used by small groups and teams to exchange and collaborate on web data or files. The visualization makes it clear that the platform is used as it was intended. There are only few connections between the small communities, mostly over public contexts which can be shared by everyone. No visualization of smaller communities was made. However, it would be interesting to do so in later experiments in order to identify the structure of these communities. For this, preprocessing of the data with community identification algorithms would be needed.

Representing the data in a visual way is crucial in order to explain the results and to understand the data. In this thesis, the data was visualized at one point in time. It would be interesting to visualize how this data changes over time in order to understand how the network evolves over time. Processing large data from one point of time already takes from several minutes up to hours; processing and visualizing data in real time would therefore be even more challenging. There are so far no systems that are capable of doing this in real time. Research in visualization is still in its infancy. It is expected that in the future, systems come to be that make visualization a simpler task and that will give interesting new insights into graph structures and other areas where visualization can be used.

Working with network data is an interdisciplinary task. In the future, more research in all associated areas is necessary. Even more importantly, the information and results have to be exchanged between the different research areas. The amount of network data in all different areas will grow in the future, which indicates that better and more efficient methods are needed in order to store, process and visualize data to better understand the data and retrieve results. To reach these goals in the near future, a close interdisciplinary collaboration – perhaps also with interdisciplinary teams – will be necessary.

References

[1] J. Abbate, "From arpanet to internet: A history of arpa-sponsored computer networks, 1966-1988," *Dissertations available from ProQuest*, Jan 1994. [Online]. Available: http://repository.upenn.edu/dissertations/AAI9503730/

[2] M. Rodriguez and P. Neubauer, "Constructions from dots and lines," *Arxiv preprint arXiv:1006.2361*, Jan 2010. [Online]. Available: http://arxiv.org/pdf/1006.2361

[3] T. Berners-Lee, J. Hendler, and O. Lassila..., "The semantic web," *Scientific american*, Jan 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.9584&rep=rep1&type=pdf

[4] "Resource description framework (rdf)," 8 2010. [Online]. Available: http://www.w3.org/RDF/. cited 16.8.2010

[5] A. S. Eric Prud'hommeaux, "Sparql query language for rdf." [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/." cited 28.9.2010

[6] K. Anyanwu, A. Maduko, and A. Sheth, "Sparq2l: towards support for subgraph extraction queries in rdf databases," *Proceedings of the 16th ...*, Jan 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1242572.1242680

[7] S. Herring, "A faceted classification scheme for computer-mediated discourse," *Language@ Internet*, Jan 2007. [Online]. Available: http://www.languageatinternet.de/articles/2007/761/index_html

[8] B. Bahmani, A. Chowdhury, and A. Goel, "Fast incremental and personalized pagerank," *citeulike.org*, 2010. [Online]. Available: http://www.citeulike.org/user/seeding-films/article/7795698

[9] T. Hoff, "I, cloud." [Online]. Available: http://highscalability.com/blog/2010/10/14/i-cloud.html." cited 26.12.2010

[10] D. Bader..., "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," *Parallel Processing*, Jan 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1690657

[11] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-..., "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," *computer.org*, Jan 2009. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/IPDPS.2009.5161100

[12] G. Malewicz, M. Austern, and A. Bik, "Pregel: a system for large-scale graph processing," *Proceedings of the ...*, Jan 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1807184

[13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, Jan 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1327452.1327492&coll=GUIDE&dl=&idx=J79&part=magazine&WantType=Magazines&title=Communications%2520of%2520the%2520ACM

[14] C. Chen, "Information visualization," *Wiley Interdisciplinary Reviews: Computational Statistics*, 2010. [Online]. Available: /journal/123398028/abstract

[15] Y. Hu, "Efficient, high-quality force-directed graph drawing," *Mathematica Journal*, Jan 2005. [Online]. Available: http://www.mathematica-journal.com/issue/v10i1/contents/graph_draw/graph_draw.pdf

[16] ——, *Algorithms for Visualizing Large Networks*, 2010, ch. 15.

[17] S. Boccaletti, V. Latora, Y. Moreno, and M. Chavez, "Complex networks: Structure and dynamics," *Physics Reports*, Jan 2006. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S037015730500462X

[18] B. Fitzpatrick, "Thoughts on the social graph." [Online]. Available: http://bradfitz.com/social-graph-problem/." cited 1.12.2010

[19] R. Johnson, "Scaling facebook to 500 million users and beyond." [Online]. Available: http://www.facebook.com/notes/facebook-engineering/scaling-facebook-to-500-million-users-and-beyond/409881258919." cited 17.9.2010

[20] S. Milgram, "The small world problem," *Psychology today*, Jan 1967. [Online]. Available: http://dces.essex.ac.uk/staff/hunter/Milgram%2520PsychToday%25201967.pdf

[21] Dr.Clip, "126 average followers per twitter user." [Online]. Available: http://bitbriefs.amplify.com/2009/06/30/126-average-followers-per-twitter-user/." cited 11.9.2010

[22] J. V. Grove, "Ashton hits 2 million followers on twitter." [Online]. Available: http://mashable.com/2009/06/02/ashton-2-million-followers/." cited 8.11.2010

[23] okram, "Property graph model." [Online]. Available: https://github.com/tinkerpop/blueprints/wiki/property-graph-model." cited 17.11.2010

[24] E. Koutsofios and S. North, "Drawing graphs with dot," *phil.uu.nl*.

[25] I. Herman, "W3c semantic web activity." [Online]. Available: http://www.w3.org/2001/sw/." cited 29.9.2010

[26] W. O. W. Group, "Owl 2 web ontology language." [Online]. Available: http://www.w3.org/TR/owl2-overview/." cited 16.10.2010

[27] J. Dongarra, "Sparse matrix storage formats." [Online]. Available: http://www.cs.utk.edu/~dongarra/etemplates/node372.html." cited 4.12.2010

[28] N. Bell. . . , "Efficient sparse matrix-vector multiplication on cuda," *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Jan 2009. [Online]. Available: http://mgarland.org/files/papers/nvr-2008-004.pdf

[29] O. Schenk. . . , "Solving unsymmetric sparse systems of linear equations with pardiso* 1," *Future Generation Computer Systems*, Jan 2004. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0167739X03001882

[30] T. Davis, "University of florida sparse matrix collection," *NA Digest*, Jan 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.1925&rep=rep1&type=pdf

[31] "Introducing json." [Online]. Available: http://www.json.org/." cited 17.11.2010

[32] V. Batagelj, "Efficient algorithms for citation network analysis," *Arxiv preprint cs*, Jan 2003. [Online]. Available: http://arxiv.org/abs/cs.DS/0309023

[33] P. Resnick, N. Iacovou, and M. Suchak, "Grouplens: an open architecture for collaborative filtering of netnews," *Proceedings of the . . .* , Jan 1994. [Online]. Available: http://portal.acm.org/citation.cfm?id=192844.192905&type=series

[34] T. T. Jr, "wiki.dbpedia.org : About." [Online]. Available: http://dbpedia.org/About." cited 28.9.2010

[35] C. Bizer, T. Heath, D. Ayers, and Y. Raimond, "Interlinking open data on the web," *4th European Semantic . . .* , 2010. [Online]. Available: http://people.kmi.open.ac.uk/tom/papers/bizer-heath-eswc2007-interlinking-open-data.pdf

[36] S. Auer, C. Bizer, G. Kobilarov, and J. Lehmann, "Dbpedia: A nucleus for a web of open data," *The Semantic Web*, Jan 2007. [Online]. Available: http://www.springerlink.com/index/rm32474088w54378.pdf

[37] J. Callan, M. Hoy, C. Yoo, and L. Zhao, "Clueweb09 data set," *boston.lti.cs.cmu.edu*, Jan 2009. [Online]. Available: http://boston.lti.cs.cmu.edu/classes/11-742/S10-TREC/TREC-Nov19-09.pdf

[38] I. S. Nick Craswell, Charles Clarke, "Trec 2009 web track guidelines." [Online]. Available: http://plg.uwaterloo.ca/~trecweb/2009.html." cited 17.11.2010

[39] D. Bader, "Petascale computing for large-scale graph problems," *Parallel Processing and Applied Mathematics*, Jan 2008. [Online]. Available: http://www.springerlink.com/index/LL6W243778N017HJ.pdf

[40] K. Madduri and D. Bader, "Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis," *computer.org*, Jan 2009. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/IPDPS.2009.5161060

[41] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," ... *Workshop on Mining and Learning with Graphs*, Jan 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1830263

[42] C. Grimes, "Our new search index: Caffeine." [Online]. Available: http://googleblog.blogspot.com/2010/06/our-new-search-index-caffeine.html." cited 29.9.2010

[43] M. R. Robert A. Hanneman, "Introduction to social network methods - 10. centrality and power." [Online]. Available: http://faculty.ucr.edu/~hanneman/nettext/C10_Centrality.html." cited 27.9.2010

[44] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, Jan 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.2024&rep=rep1&type=pdf

[45] K. D. A. Bader, "Parallel algorithms for evaluating centrality indices in real-world networks," 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1157604

[46] R. Lempel and S. Moran, "The stochastic approach for link-structure analysis (salsa) and the tkc effect1," *Computer Networks*, Jan 2000. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1389128600000347

[47] L. Freeman, "Centrality in social networks conceptual clarification," *Social networks*, Jan 1979. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/0378873378900217

[48] P. Bonacich, "Power and centrality: A family of measures," *American Journal of Sociology*, Jan 1987. [Online]. Available: http://www.jstor.org/stable/2780000

[49] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine* 1," *Computer networks and ISDN systems*, Jan 1998. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S016975529800110X

[50] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," *en.scientificcommons.org*, Jan 1998. [Online]. Available: http://en.scientificcommons.org/42893894

[51] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *Proceedings of the VLDB* ..., Jan 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1453965

[52] D. Abadi, A. Marcus, and S. Madden, "Scalable semantic web data management using vertical partitioning," ... *on Very large data* ..., Jan 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1325851.1325900

[53] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," ... *—The International Journal on Very Large* ..., Jan 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1731351.1731354

[54] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel-interactive-analysis-of-web-scale-datasets," 2010. [Online]. Available: http://www.google.com/buzz/goog.research.buzz/WsARqxc7d7R/Dremel-Interactive-Analysis-of-Web-Scale-Datasets

[55] M. Garfinkel, "Learning hadoop." [Online]. Available: http://blog.maxgarfinkel.com/archives/176." cited 17.12.2010

[56] N. H. Jesse Alpert, "We knew the web was big..." [Online]. Available: http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html." cited 17.11.2010

[57] Twitter, "Tweets per day." [Online]. Available: http://www.flickr.com/photos/twitteroffice/4990581534/." cited 10.11.2010

[58] S. Higginbotham, "Sensor networks top social networks for big data." [Online]. Available: http://gigaom.com/cloud/sensor-networks-top-social-networks-for-big-data-2/." cited 17.11.2010

[59] F. Chang, J. Dean, S. Ghemawat, and W. Hsieh, "Bigtable: A distributed storage system for structured data," *ACM Transactions on* ..., Jan 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1365815.1365816

[60] S. Ghemawat and H. Gobioff..., "The google file system," *ACM SIGOPS Operating* ..., Jan 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=1165389.945450

[61] G. DeCandia, D. Hastorun, and M. Jampani, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS* ..., Jan 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1294261.1294281

[62] B. Cooper, R. Ramakrishnan, and U. Srivastava, "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the* ..., Jan 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1454167

[63] R. Fielding, "Architectural styles and the design of network-based software architectures," *Citeseer*, Jan 2000. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.9164&rep=rep1&type=pdf

[64] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook*, Jan 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.3304&rep=rep1&type=pdf

[65] D. Wolpert..., "No free lunch theorems for optimization," *Evolutionary Computation*, Jan 2002. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=585893

[66] B. Cooper, A. Silberstein, and E. Tam, "Benchmarking cloud serving systems with ycsb," ... *symposium on Cloud* ..., Jan 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1807128.1807152

[67] H. Balakrishnan, M. Kaashoek, and D. Karger, “Looking up data in p2p systems,” . . . *of the ACM*, Jan 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=606299

[68] I. Stoica, R. Morris, D. Karger, and M. Kaashoek, “Chord: A scalable peer-to-peer lookup service for internet applications,” *Proceedings of the . . .* , Jan 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=964723.383071

[69] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *ACM SIGACT News*, Jan 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=564585.564601

[70] C. Kent, “The visual guide to nosql systems.” [Online]. Available: http://paolodedios.com/blog/2010/5/19/the-visual-guide-to-nosql-systems.html.” cited 17.11.2010

[71] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, Jan 1978. [Online]. Available: http://portal.acm.org/citation.cfm?id=359563

[72] W. Kent, “A simple guide to five normal forms in relational database theory,” *Communications of the ACM*, Jan 1983. [Online]. Available: http://portal.acm.org/citation.cfm?id=358024.358054

[73] “memcached - a distributed memory object caching system,” 09 2010. [Online]. Available: http://memcached.org/. cited 11.9.2010

[74] M. Stonebraker, D. Abadi, and A. Batkin. . . , “C-store: a column-oriented dbms,” *Proceedings of the . . .* , Jan 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1083592.1083658

[75] B. Anderson, “Dynamo and couchdb clusters.” [Online]. Available: http://blog.cloudant.com/dynamo-and-couchdb-clusters.” cited 11.9.2010

[76] M. A. Rodriguez, “Graph databases: Trends in the web of data.” [Online]. Available: http://www.slideshare.net/slidarko/graph-databases-trends-in-the-web-of-data.” cited 24.9.2010

[77] R. Pointer, N. Kallen, E. Ceaser, and J. Kalucki, “Introducing flockdb.” [Online]. Available: http://engineering.twitter.com/2010/05/introducing-flockdb.html.” cited 17.9.2010

[78] okram, “gremlin.” [Online]. Available: https://github.com/tinkerpop/gremlin/wiki.” cited 11.10.2010

[79] B. Shneiderman, “The eyes have it: A task by data type taxonomy for information visualizations,” *Visual Languages*, Jan 2002. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=545307

[80] M. Newman, “Detecting community structure in networks,” *The European Physical Journal B-Condensed Matter . . .* , Jan 2004. [Online]. Available: http://www.springerlink.com/index/5GTDACX17BQV6CDC.pdf

[81] K. Madduri, D. Bader, J. Berry, and J. Crobak, "Parallel shortest path algorithms for solving large-scale instances," *9th DIMACS Implementation ...*, Jan 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.7020&rep=rep1&type=pdf

[82] L. Freeman, "Visualizing social networks," *Journal of social structure*, Jan 2000. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.6279&rep=rep1&type=pdf

[83] D. Holten and J. van Wijk, "Force-directed edge bundling for graph visualization," *Computer Graphics Forum*, Jan 2009. [Online]. Available: http://www3.interscience.wiley.com/journal/122523375/abstract

[84] P. Butler, "Visualizing friendships." [Online]. Available: http://www.facebook.com/note.php?note_id=469716398919." cited 17.12.2010

[85] J. E. et al, "Graphviz - graph visualization software." [Online]. Available: http://www.graphviz.org/." cited 6.12.2010

[86] A. Barabási..., "Emergence of scaling in random networks," *Science*, Jan 1999. [Online]. Available: http://www.sciencemag.org/cgi/content/abstract/sci;286/5439/509

[87] T. Haveliwala, "Efficient computation of pagerank," *Citeseer*, Jan 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.3145&rep=rep1&type=pdf

[88] K. Haugen, "Analysis of the nosql landscape." [Online]. Available: http://blog.knuthaugen.no/2010/03/the-nosql-landscape.html." cited 22.11.2010

[89] M. P. Sean Cribbs, "Riak: An open source internet-scale data store." [Online]. Available: http://wiki.basho.com/." cited 11.9.2010

[90] D. S. Justin Sheehy, "Bitcask. a log-structured hash table for fast key/value data," 04 2010. [Online]. Available: http://downloads.basho.com/papers/bitcask-intro.pdf. cited 27.4.2010

[91] M. Raynal and M. Singhal, "Logical time: Capturing causality in distributed systems," *Computer*, Jan 1996. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=485846

[92] S. Metson, "Cern: A case study." [Online]. Available: http://www.couch.io/case-study-cern." cited 11.10.2010

[93] S. Maniyam, "Hbase: Bigtable-like structured storage for hadoop hdfs." [Online]. Available: http://wiki.apache.org/hadoop/Hbase." cited 13.10.2010

[94] D. Carstoiu, A. Cernian, and A. Olteanu, "Hadoop hbase-0.20. 2 performance evaluation," *New Trends in Information ...*, Jan 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5488644

[95] A. Khetrapal and V. Ganesh, “Hbase and hypertable for large scale distributed storage systems,” *Dept. of Computer Science*, 2010. [Online]. Available: http://www.uavindia.com/ankur/downloads/HypertableHBaseEval2.pdf

[96] K. Madduri, “Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks,” *Parallel and Distributed Processing*, pp. 1–12, Jan 2008. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4536261

[97] A. Hotho, R. Jäschke, and C. Schmitz…, “Information retrieval in folksonomies: Search and ranking,” *The Semantic Web:* …, Jan 2006. [Online]. Available: http://www.springerlink.com/index/r8313654k80v7231.pdf

[98] E. Gansner and Y. Koren…, “Graph drawing by stress majorization,” *Graph Drawing*, Jan 2005. [Online]. Available: http://www.springerlink.com/index/JRN52J7CX8GRCY6V.pdf

[99] T. Hoff, “Building super scalable systems: Blade runner meets autonomic computing in the ambient cloud.” [Online]. Available: http://highscalability.com/blog/2009/12/16/building-super-scalable-systems-blade-runner-meets-autonomic.html.” cited 28.9.2010

[100] D. Anderson, J. Cobb, and E. Korpela…, “Seti@ home: an experiment in public-resource computing,” … *of the ACM*, Jan 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=581571.581573

[101] A. Dou, V. Kalogeraki, and D. Gunopulos, “Misco: a mapreduce framework for mobile systems,” *Proceedings of the* …, Jan 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1839294.1839332