NTNU
Norwegian University of
Science and Technology

Graph-Based Storage In Social Networks

Kristine Steine

Master of Science in Computer Science
Submission date: May 2016
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

How do we efficiently store an ever-growing amount of data? How do we retrieve and analyze relationships between data quickly? These are among the concerns faced by companies such as Google, Yahoo, Amazon, and Facebook today. As the world's leading tech companies are tackling the challenge of their data aggressively increasing in size, NoSQL databases are emerging as their weapon of choice. Immediate consistency is sacrificed to gain higher scalability and availability.

In the case of social networking services, NoSQL graph databases are obvious contenders for the role currently held by relational databases. This thesis explores the suitability of graph databases as the main storage for these services.

Our research suggests that while using graph databases for social networking services eases the task of software development, today's leading graph databases are not as robust and mature as these services require.

Sammendrag

Hvordan kan vi effektivt lagre en stadig voksende mengde data? Hvordan kan vi raskt hente ut og analysere relasjoner mellom data? Dette er bekymringer selskaper som Google, Yahoo, Amazon og Facebook møter i dag. Samtidig som verdens ledende teknologiselskaper takler utfordringen med at dataene deres øker aggressivt i størrelse, vokser NoSQL-databaser frem som deres redning. Umiddelbar konsistens ofres for å oppnå bedre skalerbarhet og tilgjengelighet.

For tilfellet sosiale nettverk er NoSQL-grafdatabaser åpenbare utfordrere for rollen relasjonsdatabaser har i dag. Denne masteroppgaven utforsker hvor egnet grafdatabaser er som hovedlagringssystem for slike tjenester.

Vår forskning antyder at selv grafdatabaser letter oppgaven med å utvikle programvare for sosiale nettverk, er dagens ledende grafdatabaser ikke tilstrekkelig robuste og modne med tanke på kravene som stilles for slike systemer.

Acknowledgements

I would like to thank my supervisor Professor Svein Erik Bratsberg for supporting and believing in me through the past year of research. Thank you for giving me flexibility, motivation, thoughtful advice, and the belief that I could complete this thesis in time – even as the odds were looking very slim towards the end of these 20 weeks.

I would like to thank my friends who have made the past five years a fantastic time. We might not have passed all of our exams on the first try but at least we had fun trying! Special thanks to my officemates Juul Arthur and Hans Kristian for feeding me chocolate and hanging up memes on the walls to make tough days much better. Thank you, Tibor, for patiently allowing me to squat in your house, for all the inappropriate laughs and for comforting me when I need it the most.

Finally, I would like to thank my family for cheering for me through five years of studies – especially my Dad who has continually lifted my spirits by admitting he wasn't a perfect student either. At least I finished in time!

Contents

1 Introduction **1**
- 1.1 Purpose . . . 1
- 1.2 Hypothesis . . . 2
- 1.3 Research process . . . 2
- 1.4 Product . . . 2
- 1.5 Thesis structure . . . 3

2 Graph databases **5**
- 2.1 What are graph databases . . . 5
- 2.2 NoSQL . . . 6
- 2.3 Applications . . . 8
- 2.4 Neo4j . . . 10
  - 2.4.1 Data model . . . 10
  - 2.4.2 Cypher . . . 11
  - 2.4.3 Marketing towards social networks . . . 12
- 2.5 OrientDB . . . 12
  - 2.5.1 Data model . . . 12
  - 2.5.2 Query language . . . 13
  - 2.5.3 Marketing comparison with Neo4j . . . 14

3 Social Networks **15**
- 3.1 Facebook . . . 15
  - 3.1.1 Functionality . . . 15
  - 3.1.2 Social graph . . . 16
  - 3.1.3 Architecture . . . 17
- 3.2 Tumblr . . . 20
  - 3.2.1 Functionality . . . 21
  - 3.2.2 Social graph . . . 21

3.2.3 Architecture 21

4 Use of Graph Databases for Facebook 25
4.1 Experiment motivation 25
4.2 Design of experiment 26
4.2.1 Simplified data model 26
4.2.2 Chosen queries 26
4.3 MySQL 27
4.4 OrientDB 33
4.5 Neo4j 35

5 Evaluation 39
5.1 Compatibility of the data model and the programming model 39
5.2 Usability 40
5.3 Maturity 40
5.4 Cost of migration 41
5.5 ACID compliance 42
5.6 Use within Facebook 42

6 Conclusions 43
6.1 Advantages 43
6.2 Disadvantages 43
6.3 Conclusion to hypothesis 44

Bibliography 45

Chapter 1

Introduction

This chapter introduces the research project. In the following sections, the motivation and goals of the study, as well as the research process, are presented.

1.1 Purpose

Online social networking services are increasingly becoming a part of daily life, with a 2013 study showing that 79 percent of U.S. citizens over the age of 18 are actively using a social networking service [1]. The online social network *Facebook* has 1.65 billion monthly active users as of March 2016 [2], making it the largest social network in the world [3].

Social networks allow users to connect with, interact with, and browse the activity of, other users of the network. Because of this, one could say that the usage of the networks is largely concentrated around connections. Storing, indexing and querying these connections can be done in several ways. Traditionally the data would be structured in relational database tables, and when querying for "names of friends of user x" three tables and two join operations would be involved. Graph databases are specifically designed to efficiently traverse connections. The graph database Neo4j claims to offer better performance, more flexibility and agility than traditional databases when applied to highly connected data [4].

With this research, we aim to study the current database solutions of the social networking services Facebook and Tumblr with regards to their use of graph databases. In addition, we aim to understand the applicability of

graph databases for these social networking services.

1.2 Hypothesis

The hypothesis which serves as the basis for this thesis is as follows:

Social networking services would benefit from storing their data in graph-based databases as opposed to traditional SQL databases.

With this, we seek to discover any improvements in qualities such as usability, performance, maintainability, and scalability when applying a graph-based model and database instead of a traditional SQL database on social networking services. Throughout this paper, the terms *SQL database* and *relational database* are used to describe traditional SQL databases.

1.3 Research process

This research was initially motivated by an interest in graph databases and NoSQL solutions for distributed services. In order to decide the hypothesis, a pre-study of the current database infrastructures of Facebook and Tumblr was conducted. The pre-study found that Facebook currently stores their data in a relational database, supported by several custom made systems which allow developers to treat the data as though it were stored in a graph database. This result initiated a new study into whether using graph databases would be better suited for social networking services such as Facebook than their current solutions.

1.4 Product

The main product of this research is to test the aforementioned hypothesis. The pre-study provided a survey of graph databases and an analysis of Facebook and Tumblr's data structures. In addition, an analysis of how a graph database could be implemented for Facebook and an evaluation of the benefits or disadvantages this implementation would give are provided in this thesis. The products seen as a whole can provide answers to the hypothesis.

1.5 Thesis structure

Chapter 2 presents background theory about graph databases and describes how these differ from traditional relational databases.

Chapter 3 contains a study of the social networks Facebook and Tumblr, analysing their storage and querying needs and understanding how they currently store their data.

A study of how a graph database could be used in Facebook is presented in Chapter 4, and evaluated in Chapter 5.

Chapter 6 concludes the research and re-evaluates the hypothesis.

Note that Chapter 2 and 3 are largely based on the pre-study thesis which was delivered in December 2015 [5].

Chapter 2

Graph databases

Before delving into the storage solutions Facebook and Tumblr have implemented, a look into graph database technology is relevant.

This chapter begins with a presentation of how graph databases are defined, followed by a description of the NoSQL context in which graph databases live. A discussion of relevant and popular application areas for graph database implementations leads into a presentation of the well-known graph database system Neo4j.

2.1 What are graph databases

Graph databases are databases in which the data is modeled as a directed graph or as graph-like structures. Contrasting how relational databases are optimized for aggregated data, graph databases are optimized for highly interconnected data, especially for when the relationships between data are more interesting than the data itself. [6]

A graph is a data structure made up of nodes (or vertices) and edges [7]. Objects or entities are represented by nodes, while connections and relationships between nodes are represented by edges [8]. This structure implies that the network of nodes can be traversed without using indexes, called "index-free node adjacency" [6]. In addition, there are key-value properties which can be connected to edges or nodes [9]. Node metadata is stored within the node, including the ID of the node as well as information about the edges going inwards or outwards. Edges keep their ID, information about the type of connection they represent, as well as the head (incoming) and tail (outgo-

ing) nodes involved. In addition to this, nodes and edges can keep custom properties defined by the database system.

Nearly anything can be modeled as a graph. Real-world examples include Soccer World Cups [10], hierarchical organization charts [11], and logistics networks [12], among other applications presented in section 2.3.

Because graph databases are optimized for graph traversals, they can efficiently process datasets with close interconnections. The graph design allows for constructing systems for pattern detection, predictive models, and complex relationship analyses. [13] [7]

2.2 NoSQL

Graph databases are categorized as NoSQL databases, a term which is used to describe non-relational databases as well as "not only SQL" hybrid solutions [14].

Traditional relational databases share some main properties, named in short as ACID. These include; *Atomicity*, the promise that transactions are either performed in full or aborted in full; *Consistency*, requiring transactions to find and leave the database in a consistent state; *Isolation*, ensuring the isolated execution of each transaction without interference from concurrently executed transactions; and *Durability*, securing the persistence of data in the case of a power loss. [15]

Although these properties are viewed as desirable, many modern applications and services have needs which are hard to meet with the traditional data model. Services such as LinkedIn, Google Maps, and Yahoo Mail store enormous amounts of data, and require a less constrained DBMS in which large amounts of data can be accessed correctly and securely, processed quickly, and stored in a service-specific format. Consistency, structured storage, and strict schemas are lesser-valued properties for these systems while high performance, high scalability, and high availability are top priority. This weighting of DBMS properties resulted in the emergence of application-specific NoSQL systems. [15]

The term NoSQL generally applies to databases with a set of key features which contrast the abovementioned traditional qualities [14] [16]:

- A *shared-nothing architecture* in which the servers do not share processors, memory or storage, and communicate through a shared messaging network.

- *Horizontal scaling*: scaling the system by adding new nodes for storage and processing, as opposed to increasing the power of the existing nodes. The shared-nothing architecture is a key part of this scaling strategy.

- *Distributed indexes* are used to store key data values to increase search efficiency.

- *Flexible schemas* allowing new attributes to be added to data records dynamically. This is also possible in some RDBMSs but still uncommon.

- *Simple interfaces* for data search and procedure calls (SCRUD).

- *Eventual consistency*, guaranteeing consistency at some point in the near future, contrasting the immediate consistency required by ACID. Returned data may not be up-to-date, but all updates are guaranteed to propagate to all nodes. Locking is not always supported, weakening the concurrency model further.

With these features, NoSQL databases sidestep strict ACID in favor of an approach called BASE: Basically Available, Soft-state, Eventually Consistent. This approach allows for higher scalability and availability but sacrifices immediate consistency. The degree to which they avoid ACID guarantees varies, as some NoSQL databases offer stronger consistency through data versioning, logging, and locks. [14][17]

In addition to graph databases, several other database types are categorized as NoSQL [16][17]:

- **Document databases** store standardised documents (e.g. XML or JSON) containing data in key-value pairings, where the system can interpret the values and thereby query these as well as keys. Documents and attributes can be added and removed at runtime as document databases are schema-free. Even without set schema restrictions, multi-attribute lookups can be performed on records with unequal key sets. These stores are popularly used for schema migration, real-time analytics, logging, websites in continuing development, and storage for small systems with yet-to-be-determined data schemes.

- **Pure key-value stores** also allow schema-less storage and efficient search in large data sets of key-value data, but the values are opaque to the system. These stores are very useful for simple key-based operations because of the simplicity of the data structure and are popularly used for caching of complex SQL queries as the dataset will often be held in memory.

- **Tabular stores** extend traditional relational database design by splitting rows and columns over multiple nodes, supporting data partitioning horizontally and vertically. Unlike in document databases, the system is not able to interpret values and other data types than strings are not natively supported. This data model is most suitable for analytical applications and for applications requiring very high scalability because of the efficient partitioning mechanisms. Tabular stores are also known as Column Family stores or Column-oriented stores, terms which also include more traditional relational databases such as C-Store.

- **Multi-model databases** are combinations of the above database types and combine several data models to take advantage of the abilities of each. [8]

Arguably, the most well-known graph database today is Neo4j, which will be further described in section 2.4. In addition, the multi-model database OrientDB supports graph models which we will look more closely into in section 2.5.

Well-known examples of non-graph NoSQL databases include the key-value and column-oriented hybrid Apache Cassandra [18], Amazon's key-value store Dynamo [19], Google's column-oriented database Bigtable [20], Apache's column-oriented database HBase [21], Apache's document database CouchDB [22], and the open-source document-oriented database MongoDB [23].

2.3 Applications

Below is a presentation of some application areas for graph databases. Some are more obvious than others, and some are more useful than others, but still these examples show that graph databases are not "one-trick ponies".

Recommendation systems

Recommendation systems predict user interests and behaviour based on behaviour similarities between users (collaborative filtering), or between new items and items which the user previously has shown interest in (content based filtering). Matching patterns in user behaviour is implemented in graph databases by traversing interaction edges along a network of users and items. [7]

Fraud detection

Fraud can be difficult to discover if relationships are not obvious. Graph databases can be helpful in visualizing and searching for hidden relationships in cases of tax fraud [24] or insurance fraud [25].

Access control

The leading Norwegian telecom provider Telenor implements resource authorization using Neo4j. A graph containing customers, subscriptions, price plans, accounts, and agreements is used to find out e.g. whether a given user is qualified for a given price plan. Because this involves traversal of a network of connected data the graph implementation is favorable to their previous SQL implementation. [26]

Biological networks

Biological information can be represented as networks with regards to chemical structure, metabolic pathways, food webs, neural networks. In such networks analysis of connections is very interesting, e.g. in discovering how proteins interact. [6]

Social networks

Perhaps the most obvious use case for graph databases is storage of social networks. In such networks, nodes can represent people or groups while edges represent connections or flows. Online social networks require high scalability and availability as well as the ability to traverse the network several edges at a time [27]. Relational databases which provide the former involve costly

join operations for the latter, while complex and deep edge queries are easily implemented in graph databases [7].

When to avoid graph databases

The main strength of graph databases is traversal of relationship networks, so data with a low degree of connections would gain less from such a database. Simple data models without significant interconnections, or applications where data relationships are less relevant, would have more to gain from other storage solutions.

2.4 Neo4j

Neo4j is a commercial open-source graph database implemented in Java. Neo Technologies [28], the developers of Neo4j, claim the database to be the world's leading graph database. According to their website, Neo4j is highly available, scalable to the level of billions of nodes and edges, and offers full support of ACID. Neo4j is free to use for non-commercial projects under the GPL v3 license. [29]

2.4.1 Data model

The data model is very similar to the general data model of graph databases. Nodes are called *nodes* and edges are called *relationships*. Nodes can optionally be assigned any number of *labels* which describe the node's affiliation to sets of nodes. These sets are used in schema definitions and in filtered queries. Labels can be added or removed during runtime and can be used to describe a temporary state as well as a node type. Relationships must be assigned a *relationship type*, which categorizes the relationship much like labels do for nodes. Relationships are always directed but can be traversed bidirectionally. Nodes and relationships can have *properties* which are named values: either numeric, boolean, string type or collections of other value types.

Schemas are not required and graphs may be created and used without one, but an optional schema in the form of constraints and indexes is available. Indexes may be created on a property for all nodes with a given label to improve node lookup time. Note that indexes are *eventually available* in Neo4j, meaning an index is created immediately upon the operation being

called, but will not be available for querying immediately as it is populated in the background. When the index is fully populated it will *come online* and be ready to be used in queries. Data integrity may be enforced using constraints. Constraints can be made for nodes or relationships, to ensure node property uniqueness or to require property existence. A property uniqueness constraint will create an index on that property, which will be dropped if the constraint is dropped. A combined existence and uniqueness constraint may be set on a property, and several constraints may be set on the same label. Adding a constraint is an atomic operation which does not return immediately, and the constraint will not be active until the operation is fully performed.

2.4.2 Cypher

Neo4j comes with its own declarative graph query language, Cypher, inspired by SQL. Cypher is meant to express what data should be returned without describing exactly how the data should be retrieved. By moving query optimization to the implementation level, the creators intend to relieve the user of some data structure awareness. The creators intend for Cypher to be easy to use for users, with self-explanatory queries.

Edges are traversed using an arrow syntax. `(n)--(m)` describes any relationship between two nodes, while `(n)-->(m)` describes any directed relationships from one node to another. The relationship type (i.e. label), direction, and relationship properties can optionally be added to this syntax by placing a bracket in the middle of the arrow, e.g. `()-[:FRIEND {BFF: true}]->()`. Nodes can also be labelled by adding `:LABEL` inside the parentheses. Properties are expressed as map-constructs and can be added to both nodes and edges.

Below is an example query written in Cypher, searching for a Person node with the name property value "John", and the names of friends of friends of that node. In order to retrieve information about nodes or relationships in the `RETURN` statement, these must be given context variable names which exist only within the query. Here, the first node is given the variable name `j` and the friends of his friends are given the variable name `fof`. [28] [30]

```
MATCH (j:PERSON {name: 'John'})-[:FRIEND]->()-[:FRIEND]->(fof)
RETURN j.name, fof.name
```

2.4.3 Marketing towards social networks

Social applications are highlighted as one of the main use cases for Neo4j on the Neo4j website [31]. The arguments for this are mainly the same as for graph databases in general, along with the claim that Neo4j is the leading, most documented, and best supported graph database in the world. The comparisons supporting the marketing are made between Neo4j and relational databases, but not between Neo4j and other NoSQL databases.

2.5 OrientDB

OrientDB is an open-source multi-model DBMS implemented in Java, which is free to use for any purpose under the Apache License 2.0. [32] [8] Academic documentation of OrientDB is limited, and for that reason most of this section is based upon the documentation provided by Orient Technologies. In addition, [16] gives a basic overview of the original release of OrientDB in 2012, but this information is mostly outdated and cannot be fully applied to today's version.

2.5.1 Data model

OrientDB supports four data models: graph, document, key-value, and object oriented. All models are supported in the core engine.

The graph model greatly resembles Neo4j's data model. Nodes are called *vertices* and are connected by *edges*. A vertex must hold its own identifier as well as sets of incoming and outgoing edges. Edges must hold their identifier, links to the head (incoming vertex) and tail (outgoing vertex) of the edge, and a label describing the type of relationship between the vertices. In addition to these mandatory properties, vertices and edges may keep custom properties defined by the user.

The OrientDB engine is built to support all four abovementioned data models, and may be accessed through both a document API and a graph API. The developers claim that 80 % of a user's database needs can be handled through the graph API. In the graph API relationships are bidirectional edges, while in the document API relationships are mono-directional links. Documents in the document API are atomic units, allowing relationship operations to be performed without involving the target document, which suggests that the document API may be better suited for multi-threading.

The graph API may be run in schema-less, schema-full or hybrid modes. Schemas are defined at class level and may be used to constrain all fields or a subset of data. Schema-less mode creates classes without properties and allows fields to be arbitrary. Hybrid mode defines some properties on class creation and allows custom fields to be added to records. In schema-full mode all fields are mandatory and must be predefined – custom properties may not be added to records.

2.5.2 Query language

The developers have chosen to use SQL as OrientDB's query language, arguing that it remains the most widely recognized standard and that a majority of developers are comfortable using it. In order to enable graph functionality, some extensions have been added. The OrientDB version of SQL is case insensitive for keywords and class names, but case sensitive for values and field names. Any field may be queried, regardless of whether it is indexed. The engine will automatically detect whether any indexes may be applied to improve the performance of a query, but indexes may also be used directly within a query as shown in the example below.

```
SELECT FROM INDEX:myIndex WHERE key = 'Jay'
```

Relationships are represented by `LINKS` in place of JOINs, and the traditional JOIN syntax is not supported. The dot notation that in SQL is used to note properties from tables which are joined, is used in OrientDB to traverse links. This is illustrated below by two equivalent queries: the first is written in ordinary SQL, the other in OrientDB's version of SQL.

```
SELECT *
FROM Person P, Dog D
WHERE P.id = D.owner AND P.name = "John"

SELECT * FROM Dog WHERE owner.name = "John"
```

Other differences from traditional SQL include the star character `*` being optional in projections wherein all fields are included – the developer is allowed to simply write `SELECT FROM Dog`. In addition, OrientDB does not support the `HAVING` keyword and recommends using nested queries instead.

2.5.3 Marketing comparison with Neo4j

As described in 2.4.3, one of the main use cases Neo4j is promoted for is social networking. While we saw that other NoSQL databases are not named in the promotional material for Neo4j, OrientDB names Neo4j as a competitor in much of its marketing material. [33] [34] [35]

One of the main arguments Orient Technologies presents for choosing OrientDB over Neo4j is pricing: The community edition of OrientDB is free for any purpose under the Apache License while Neo4j is only free for non-commercial open-source projects. OrientDB is also available in an extended enterprise edition which includes distributed clustering configuration, metrics analysisproduction support, more advanced security features, and more advanced backup options. The enterprise edition is licensed commercially with a discount option for startup companies. [36]

In addition to pricing, Orient Technologies lists several features supposedly supported in OrientDB but not in Neo4j, including record level security and access control, runtime garbage collection, more advanced schema handling, custom data types, and using a developer-friendly query language with familiar data model options. They claim to be better at crash recovery thanks to write-ahead logging, and promote their multi-master replication as a better solution than Neo4j's master-slave architecture by featuring linear scalability. Sharding is also presented as a feature supported by OrientDB which is not supported by the community edition of Neo4j, but in the current version of OrientDB sharding is only available to a limited degree: auto-sharding, sharded indexes, distributed aggregation, and backups are not fully supported yet.

While Neo4j is OrientDB's main competitor within the world of graph databases, Orient Technologies also names MongoDB as a competitor. The main benefits presented in OrientDB's favor are the multi-model engine, graph handling of relationships, schema handling, SQL as a query language, ACID compliance, multi-master replication, and direct management of disk pages (avoiding memory mapping).

The multi-model engine is among the most important features in the marketing of OrientDB. Instead of storing parts of a system in MySQL, and others in MongoDB, HBase or Cassandra, Orient Technologies argue that the combined features of the different storage solutions can be found in OrientDB. By using only one system, companies can be relieved of expensive licensing, increased system administration, and communication between systems.

Chapter 3

Social Networks

This chapter presents the two social networks Facebook and Tumblr, and their choices of architecture.

Each section contains a presentation of the social networks, their user demographics and social graph, the functionality provided, and the storage architecture implemented.

3.1 Facebook

Facebook is an online social networking service that was launched in February 2004. As of September 2015, Facebook has 1.55 billion monthly active users, of which 1.01 billion are active daily. Users represent all age groups, though a small majority of users are under the age of 35. [37] [2]

3.1.1 Functionality

Facebook is an online social network on which users register a personal profile and connect with other users (such connections are called *friends*). Users can add text posts, photos and videos, events and invitations, and personal information to their profiles, and can publish text or media posts to friends' profiles. Users can also create and join public or private groups to which such posts can be added. Businesses may register business profiles called *pages* which can be used similarly to personal profiles. Users can control what parts of their profile other users can see and interact with. Profiles can be referenced (*tagged*) in text or media posts by their friends. Users can *like* or

comment on posts they have access to. In addition to this core functionality, users can chat privately, and businesses can advertise to select user types (this is how Facebook creates revenue). [38] [39]

3.1.2 Social graph

The Facebook social graph has a variety of entities and connections. As the functionality is complex, so is the graph. The graph is visualized in Figure 3.2. In this visualization, the main entities are users, posts, comments, pages, groups, and events. In addition there are several entity types in the full graph, such as photos, videos, message threads, messages, notifications, life events, places, apps, and ads, and further edge types such as tagged-in, follows, checked-into, read, has-seen, and much more [40]. The graph is directed, and while most edges are unidirectional, the friend-edge is bidirectional. The graph is sparse but highly clustered. Most users have less than 200 friends, and the median friend count is 99 [41]. 92.7% of users have fewer friends than the average friend count of their friends [41]. The giant connected component includes 99.91% of nodes in the graph [42].

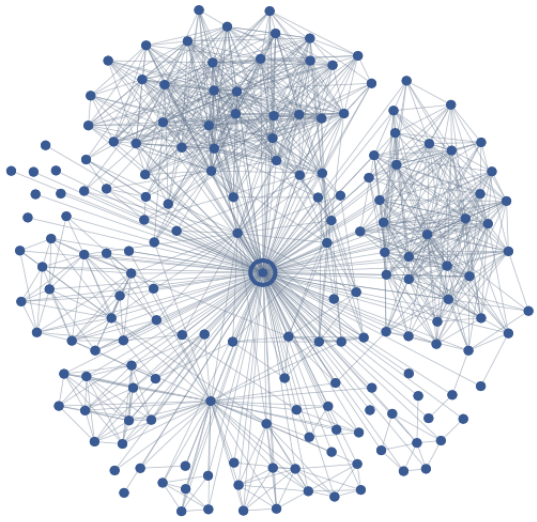

Figure 3.1: Example of friend relationships in Facebook. Each friend of the center user is represented by a shaded dot, and each friend relationship among this user's friends is represented by a line. Retrieved from [43]

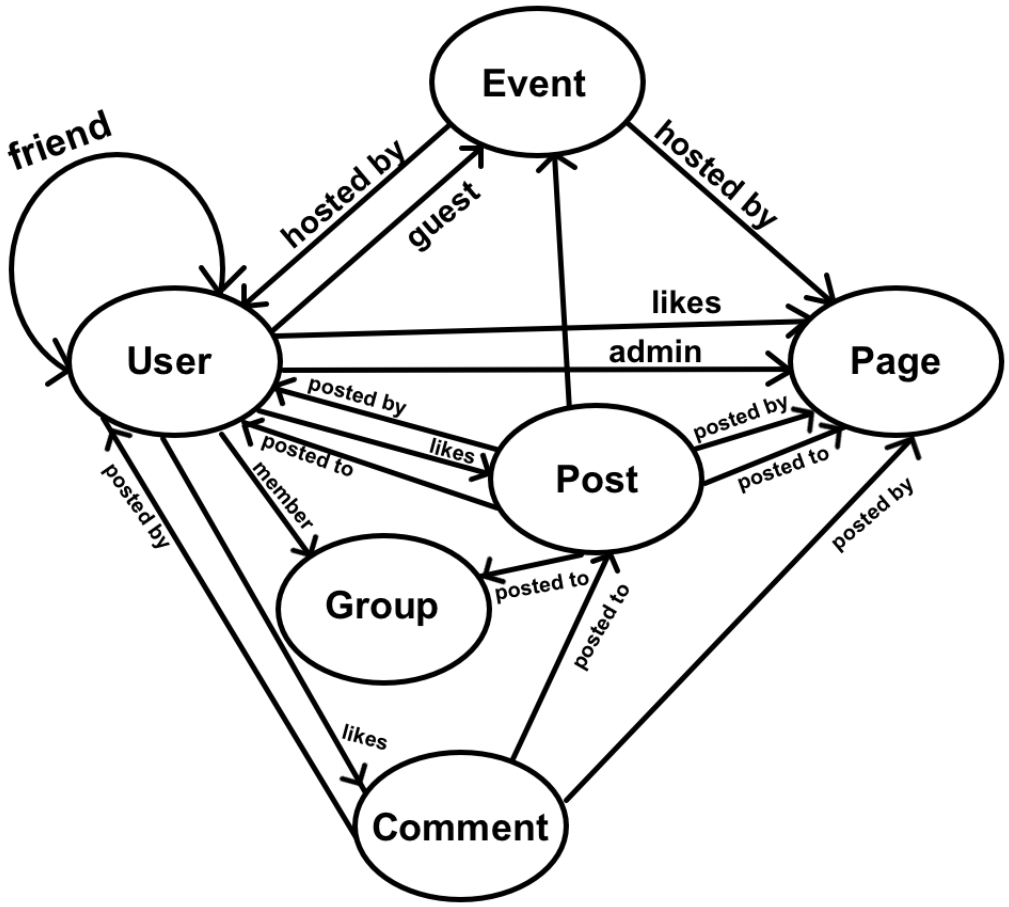

Figure 3.2: Facebook Social Graph (simplified) [40]

3.1.3 Architecture

Facebook originally stored all data in MySQL. Today data for different applications within Facebook are stored in a number of storage solutions. Data for applications that need low-latency database accesses are stored in RocksDB [44], Facebook's open-source key-value store built upon Google's LevelDB [45]. Data for Messages, Nearby Friends, search indexing and data scraping is stored in HBase [46] [47]. Still, not all data has been moved away from

MySQL [48] [27]. Nodes and edges that are added to the social graph are first written to a single MySQL-instance, before being replicated and pushed to the various services within Facebook [49].

Despite storing data mostly in relational or key-value stores, Facebook implements graph-aware layers on top of the storage layer in order to serve the social graph to their applications.

Graph API

Facebook serves the social graph through TAO [27], a read-optimized graph abstraction implemented on top of their MySQL storage. TAO serves the API between Facebook's web servers and MySQL, replacing the previous solution of direct access between the servers and storage. It favors availability and efficiency over strong consistency. TAO is only deployed as a single geographically distributed instance, and can handle a billion reads per second as the multi-petabyte data set continually changes.

In TAO, nodes are *objects* and edges are *associations*. Associations are directed but are often coupled with inverse associations. Bidirectional edges are represented by two oppositely directed associations. Associations from an object towards itself are allowed. Through the API, objects can be created, partially or fully updated, retrieved, or deleted. Associations can be created, updated, retrieved, or deleted. If an association with a coupled inverse association is operated upon, the same operation is automatically performed on the inverse. Atomicity is not guaranteed for write operations performed on coupled associations, and failures in such operations are repaired asynchronously. Association queries are performed on association lists, e.g. a list of all of the comment-associations from a photo object, which can handle queries such as *"10 most liked comments on Bob's photo"*.

The API is mapped to a set of SQL queries, as the underlying storage is MySQL. The data is highly sharded, and objects are bound to a shard for their lifetime. Shards have one table for objects and one table for associations. Associations are stored on the same shard as their ingoing object, to allow any association query to be served from a single server. The shard ID is embedded in the object ID to allow simple identification.

In addition to the storage layer, TAO has a caching layer which implements the entire API for clients. Multiple caching servers work together in tiers to be able to respond to any API request. The in-memory cache holds objects, association lists, and association counts, in a least-recently-used pol-

icy. This separation of caching and storage is highlighted by Facebook as useful in the design, operation, and scaling of each, allowing for different tradeoff decisions for the two layers.

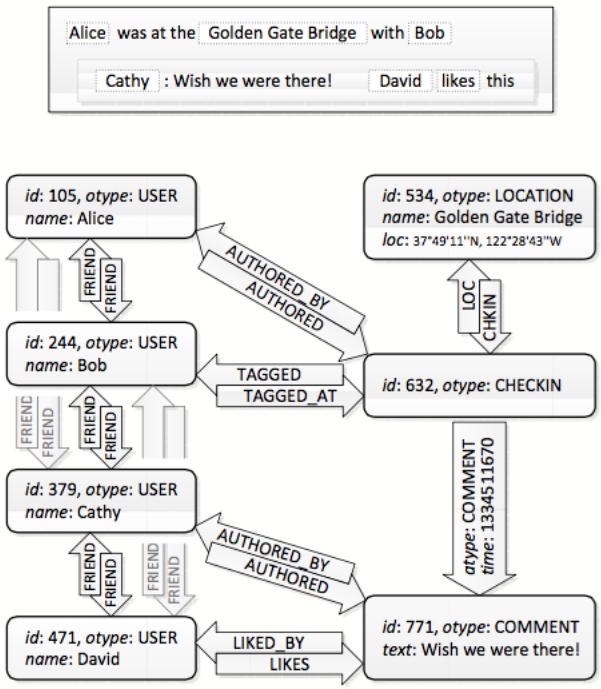

Figure 3.3: An example of how a check-in post can be mapped into TAO objects and associations [27]

Graph search

Searches within Facebook are also graph-aware. The social graph is indexed by Unicorn [43], Facebook's online, in-memory, graph-aware indexing system. Unicorn is designed to handle searches on trillions of edges and billions of nodes. Data scrapes accessed through Hive [50] are built into indices using Hadoop [51]. Real-time updates to the indices are supported by a pipeline from the front-end cluster through Thrift [52] into Unicorn index servers.

Unicorn can handle queries that involve several round-trips between the top aggregator and index servers, such as friend-of-friend queries. Nested

queries are also handled internally by Unicorn to simplify client code. Nodes are stored in entity-type specific verticals along with edges of the same result-type, e.g. all users and user-id yielding edges are held in the same vertical, so that edge traversals between nodes of the same kind do not have to cross into other verticals. This reduces latency as set operations can be performed at the leaf level.

Unicorn serves the lineage of the graph traversals involved in a search result along with the results so that the client can investigate whether the user performing the search is authorized to view the necessary edges leading to the search result. Privacy is as such not handled within Unicorn, but privacy controls are supported through this lineage presentation.

Graph processing

Large-scale analysis of the social graph is handled by a custom version of Apache Giraph [53] [54]. Giraph is an iterative graph processing system that scales to thousands of machines and can process trillions of edges. Data is provided from various sources, including MySQL, Hadoop, and Hive. Facebook applications that use analyses of large subgraphs or even the entire social graph include content rankings, content recommendations, and evaluating ad preferences.

Facebook's People You May Know service is one of these applications. Evaluating relevant friend suggestions based on a user's friends' friend lists can in a worst case require traffic to 25 million machines. Facebook implements an algorithm involving graph partitioning on a set of Giraph machines and optimizing these partitions to maximize the number of local edges within each partition, before performing the graph analysis. This algorithm improved the bandwidth utilization twofold when first implemented. [55]

3.2 Tumblr

Tumblr is an online social network and microblogging platform that was launched in February 2007. Tumblr has more than 230 million active users as of November 2015 [3]. These users manage a total of 268.1 million blogs with more than 125 billion blog posts [56]. The majority of users are under the age of 35 [57] and the number of users is growing very rapidly [58].

3.2.1 Functionality

Tumblr allows users to create blogs to which multimedia can be posted. Posts can be categorized (*tagged*) with an arbitrary amount of keywords, and such tags can be browsed individually either within a blog or site-wide. Blogs are categorized as *primary* or *secondary*: primary being the blog which is created upon user registration, and secondary being blogs which the user has created in addition to the primary. Users have one primary blog and any number of secondary blogs. Secondary blogs can be password-protected while primary blogs must be public. Users can add other users as contributing members or admins of secondary blogs. Posts from blogs which a user follows are shown chronologically on the user's *Dashboard*, which is the landing page upon login. Users can also interact in several ways: by sending *Fan Mails* which are private messages, sending *Ask* messages to which the recipient can choose to answer publicly on their blog or privately through direct messages, or through *Submissions* to other users' blogs. Ask and Submissions are optional features to a blog while Fan Mail is by default and immutably enabled. [59]

3.2.2 Social graph

Tumblr's social graph is simpler than Facebook's, as the functionality is more limited. For the purpose of this visualization, the messaging functionality has been disregarded. The main entities are users, posts, blogs, and tags. The *follows* connection is unidirectional in contrast to Facebook's bidirectional friend connection, and 29% of such connections are reciprocated which is very high in the blogosphere. The giant connected component covers 99.61% of the nodes in the social graph. [60] [42]

3.2.3 Architecture

A 2012 interview with the then VP of Engineering at Tumblr, Blake Matheny, is the main source for the following presentation of the architecture of Tumblr's storage systems. At the time, Tumblr was already a very large application, with 500 million page views each day. The users consume more data than they produce: in February 2012 about 50 GB of new blog posts were added each day, while user inbox updates (posts from blogs a user follows) totaled at 2.7 TB a day. [61]

The data is stored in a variety of systems. The majority is stored in a

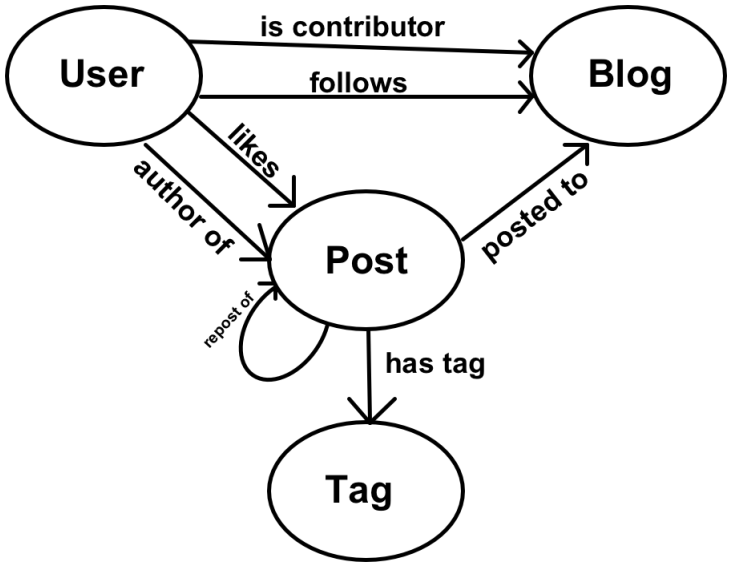

Figure 3.4: Tumblr Social Graph (simplified) [60]

greatly partitioned MySQL architecture [62], a remnant of the original LAMP (Linux, Apache, MySQL, PHP) stack Tumblr was built upon. For situations requiring write-optimization such as the Dashboard (which receives many million writes per second), HBase [21] is used. HBase was considered as a replacement for MySQL, but Tumblr decided against this because of insecurities regarding deploying HBase without their engineers being experienced with how it works.[61]

Notifications (alerting a user that someone liked one of their posts, or that their Dashboard is updated, etc.) are stored in the open-source NoSQL key-value store Redis [63]. Redis was considered acceptable for notification storage despite its lack of persistency guarantees, as notifications are ephemeral and the loss of these would not be critical. Redis is also used as the first level cache for the Tumblr URL shortener (with HBase as permanent storage) and for the secondary index of Dashboard.[61]

At the time of the interview, Tumblr was moving its Dashboard storage from a scatter-gather model to a cell-based model. The scatter-gather model involves data being stored without regards to the locality of related data, then being read from these different locations. The cell-based model stores

all posts in a user's Dashboard together chronologically. This reduces the cost involved in traversing all of the user's *follows* connections to retrieve posts significantly. All users are mapped into cells, which are self-contained structures containing all data for a body of users. Each cell has a Redis caching cluster as well as an HBase cluster. When a post is published by any user, it is written to Kafka [64], the internal firehose. All cells consume posts from the firehose and write them to their HBase. Cells then determine whether any of the author's followers are mapped to the cell, in which case the post ID is pushed to the followers' inboxes. This is to say that all posts are stored in all cells. When a user opens their Dashboard, all the data which is being read comes from their own cell. The goals of the cell based model are to improve parallelization, enable rolling upgrades and beta testing of features on subsets of users, and isolate failures. If one cell fails, only the users in that cell are affected and all other users can still interact with all posts. The cell-based model is regarded as very robust. [61]

Tumblr's model of moving into new architecture designs is to gradually use new technologies in pilot projects so that engineers can get familiar with them, before implementing larger projects with more critical potential damage. This model is a result of Tumblr being a slowly growing startup with a low number of employees – in 2012 there were only 20 engineers out of 100 employees [65] [61].

Chapter 4

Use of Graph Databases for Facebook

This chapter looks into what an implementation of a graph database for Facebook can look like. The background and design of the experiment are presented first, before we look more closely at specific examples of implementations.

4.1 Experiment motivation

We have seen that graph data models are suited to represent real-world social networks and that the modern graph databases Neo4j and OrientDB provide several features required by large-scale social network services. Still, neither Facebook nor Tumblr use graph databases for their storage, though Facebook does implement graph-like systems on top of their relational or key-value stores.

Our hypothesis is that Facebook would benefit from using a graph-based database. To understand whether this is the case, we want to demonstrate how an implementation in Neo4j or OrientDB would compare to an implementation in MySQL. For the purposes of this demonstration, any secondary graph-like systems used in conjunction with MySQL are disregarded.

4.2 Design of experiment

We examine how the Facebook data model may be implemented and queried in a graph database, by implementing a simplified version of the data model and performing some key queries unto it. This is also demonstrated in MySQL for the purpose of comparison.

4.2.1 Simplified data model

The implemented data model, shown in Figure 4.1 is a simplified version of Figure 3.2 from Section 3.1.2. Groups and Events are not part of this version, and while User, Page, Post, and Comment remain the functionality is reduced. In addition to the illustrated node network, Users hold several properties such as birth year, gender, and current city, and hold additional relationship links for family and romantic relationships.

4.2.2 Chosen queries

As described in Section 3.1.3 Facebook offers their users graph-aware searching. Queries can range from simply searching for a named user, to more complex searches such as "photos of friends of people who like bacon". Some example searches are illustrated in Figures 4.2 and 4.3.

In addition to graph search queries, Facebook must support queries for basic functionality such as viewing lists of mutual friends between two users, suggesting relevant users or Pages to follow, and generating the News Feed. The latter is quite intricate, as it involves retrieving relevant recent posts by a user's friends and followed pages in addition to posts that the user's friends recently have interacted with.

To demonstrate the varied queries Facebook should support, the queries below have been chosen.

$Q1$: Retrieving the names of all mutual friends of two users.

$Q2$: Retrieving the names of a user's friends that live in Bergen, Hordaland and like NTNU's Facebook page.

$Q3$: Retrieving all posts by a user's friends and followed pages from the past 24 hours.

$Q4$: Retrieving photos of a user's friends who have friends who like the Bacon Facebook page.

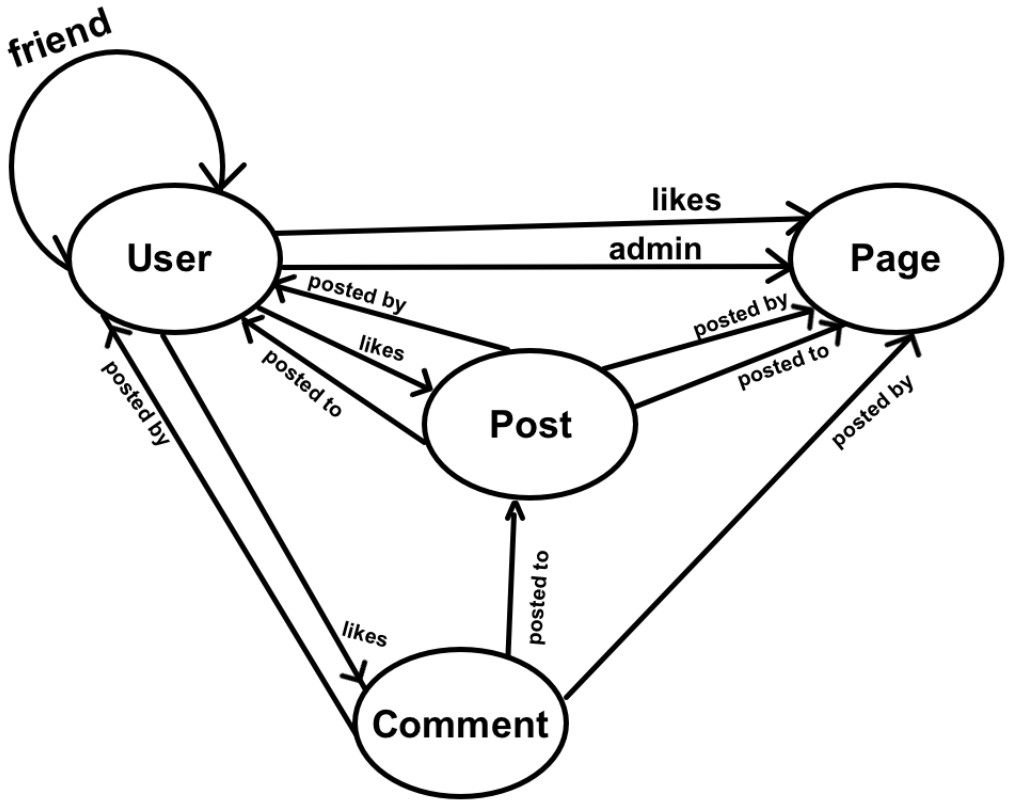

Figure 4.1: The simplified Facebook data model which is implemented

4.3 MySQL

As MySQL is relational, edge traversals must be performed using JOINs or nested queries. Simply joining two tables can be performed quite easily, while longer edge traversals can be equally complex.

In our case, the nodes in Figure 4.1 are stored in tables. Relationship storage depends on the cardinality of the relationships: one-to-one relationships are stored as properties at each end of the edge; one-to-many relationships are stored as a property of the tail node of the edge (e.g. a mother-to-child relationship is stored in the child's node as the child may only have one mother); many-to-many relationships are stored as separate tables containing the ID property of the nodes at each end of the edge. [15]

Figure 4.2: The results of the Facebook graph search "photos of friends of people who like bacon"

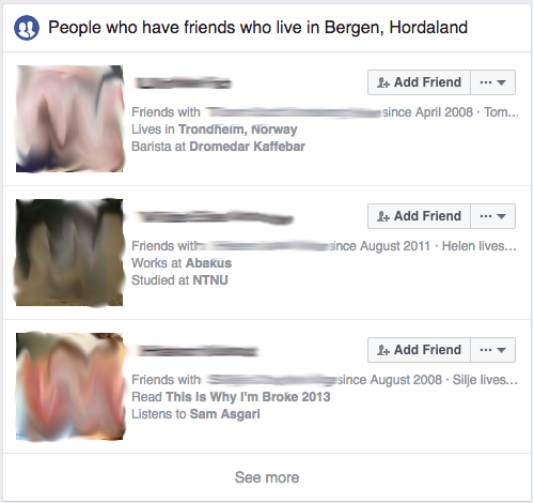

Figure 4.3: The results of the Facebook graph search "friends of people who live in Bergen"

In the following queries, we assume the existence of the tables Person, Posts, Page, Friends, LikesPage, and Tagged. The first three represent nodes in the social graph, while the latter three represent many-to-many relationships.

Q1: *Retrieving the names of all mutual friends of two users.*

To implement $Q1$ we query for the names of friends of the person with username "jane.doe" who are also friends with the person with username "john.doe". This must be performed using a triple-nested query, as we need to resolve the ID of the persons, find their mutual friends' IDs, and resolve the names of those mutual friends.

```
SELECT firstname, lastname
FROM Person
WHERE personid IN (
  SELECT personidB
  FROM Friends
  WHERE personidA IN (
    SELECT personid
    FROM Person
    WHERE username = "jane.doe"
  )
)
AND personid IN (
  SELECT personidB
  FROM Friends
  WHERE personidA IN (
    SELECT personid
    FROM Person
    WHERE username = "john.doe"
  )
)
```

Q2: *Retrieving the names of a user's friends that live in Bergen, Hordaland and like NTNU's Facebook page.*

$Q2$ is performed by joining the Person and LikesPage tables and filtering for the page with the alias "ntnu.no", then filtering the persons' current city as well as only selecting persons who are friends with the person holding the username "john.doe". This produces a nested query to find the correct `pageid` for the NTNU page, and a double-nested query to filter friends of "john.doe".

```
SELECT firstname, lastname
FROM Person, LikesPage
WHERE Person.personid = LikesPage.personid
AND LikesPage.pageid IN (
  SELECT pageid
  FROM Pages
  WHERE aliasURL = "ntnu.no"
)
AND Person.currentcity = "Bergen, Hordaland"
AND Person.personid IN (
  SELECT personidB
  FROM Friends
  WHERE personidA IN (
    SELECT personid
    FROM Person
    WHERE username = "john.doe"
  )
)
```

Q3: *Retrieving all posts by a user's friends and followed pages from the past 24 hours.*

$Q3$ is similarly complex. We assume the Posts table to be indexed on the `time_posted` property. To find the desired posts, two double-nested queries are required: one to find a list of friends of "john.doe", and one to find a list of pages "john.doe" likes. Only posts with a timestamp from the past 24 hours are returned, in descending order.

```
SELECT postid
FROM Posts USE INDEX (time_index)
WHERE time_posted > DATE_SUB(CURDATE(), INTERVAL 1 DAY)
AND (postedbyperson IN (
  SELECT personidB
  FROM Friends
  WHERE personidA IN (
    SELECT personid
    FROM Person
    WHERE username = "john.doe"
  )
)
OR postedbypage IN (
  SELECT pageid
  FROM LikesPage
  WHERE personid IN (
    SELECT personid
    FROM Person
    WHERE username = "john.doe"
  )
))
ORDER BY time_posted DESC
```

Q4: *Retrieving photos of a user's friends who have friends who like the Bacon Facebook page.*

To resolve $Q4$ we query for friends of the person "john.doe" who like the page holding the alias "Bacon", then join this list of persons with the Tagged table to find photos the persons have been tagged in. As shown below, this results in a query containing an inner query containing two double-nested queries.

```
SELECT photoid
FROM Tagged
WHERE personid IN (
  SELECT personidB
  FROM Friends
  WHERE personidA IN (
    SELECT personid
    FROM LikesPage
    WHERE pageid IN (
      SELECT pageid
      FROM Pages
      WHERE aliasURL = "Bacon"
    )
  )
  AND personidB IN (
    SELECT personidB
    FROM Friends
    WHERE personidA IN (
      SELECT personid
      FROM Person
      WHERE username = "john.doe"
    )
  )
)
```

4.4 OrientDB

In OrientDB, the data model can be implemented as a graph, which would be preferable as the data will be queried as a graph. Using the graph-based data model and the graph API enables bidirectional edge traversal and allows the developer to choose between SQL-like queries and declarative pattern-matching. [66]

As shown in Section 2.5.2 `LINK`s are used to avoid JOINs: one-to-one relationships are represented as `LINK`s pointing directly from one record to the other; one-to-many relationships can be represented as `LINKSET`s which are unordered sets of distinct pointers from a record, as `LINKLIST`s which are ordered sets of pointers from a record and allows duplicates, or as `LINKMAP`s which are Java Maps storing a key for each pointer; many-to-many relationships are represented by one-to-many pointers stored at both ends of the edge. In the graph API one-to-many relationships are not allowed: to implement such relationships the developer must instead create multiple edges.

Q1: *Retrieving the names of all mutual friends of two users.*

$Q1$ is implemented below as a pattern-matching query, finding the person whose username is "john.doe" and their friends, filtering those who also have a friend relation with the person whose username is "jane.doe". This query format resembles Neo4j's query language Cypher although the path traversal is presented differently. Path traversal may also be represented by a shortened arrow format which will be shown in the next query.

```
MATCH {class: Person, where: (username="john.doe")}
  .both("friend"){as: mutual}
  .both("friend"){class: Person, where (username="jane.doe")}
RETURN mutual.firstname, mutual.lastname
```

Q2: *Retrieving the names of a user's friends that live in Bergen, Hordaland and like NTNU's Facebook page.*

Below, $Q2$ is shown implemented both using OrientDB's version of SQL and as a pattern-matching query. In the first query, we resolve the record ID of the vertex holding the person whose username is "john.doe" and create a context variable called `$friend` holding the friends of that person. Then, those friends are filtered on the `currentcity` property, and the record ID of the page with the "ntnu.no" alias is required to be among the pages the friends like.

In the second version of the query, the pattern is spelled out more directly. We query for the vertex holding the person whose username is "john.doe", traverse his `friend` relationships to find friends who live in Bergen, create an alias for the nodes at the other end of those relationships, and traverse their `likes` relationships to find whether they like the "ntnu.no" page. Note that the shortened arrow format is used to represent the edge traversal in this query. The traversal shown as `{...}-friend-{...}` represents a bidirectional "friend" edge, while the `{...}-likes->{...}` traversal is a directed edge.

```
SELECT $friend.firstname, $friend.lastname
FROM (SELECT @rid FROM Person WHERE username = "john.doe")
LET $friend = out("friend")
WHERE $friend.currentcity = "Bergen, Hordaland"
AND (SELECT @rid FROM Page WHERE aliasURL = "ntnu.no") IN
  $friend.out(likes)

MATCH {class: Person, where: (username="john.doe")}-friend-
  {as: friends, where: currentcity="Bergen, Hordaland"}
  -likes->{class: Page, where: (aliasURL="ntnu.no")}
RETURN friends.firstname, friends.lastname
```

Q3: *Retrieving all posts by a user's friends and followed pages from the past 24 hours.*

The implementation of $Q3$ shows how the two query formats can be combined in queries. Two separate pattern matches are used to find the posts made in the last 24 hours (timestamps in OrientDB are milliseconds since the Unix Epoch) by "john.doe"'s friends and liked pages, and a `SELECT` query handles merging the two result collections and returning them sorted by time.

```
SELECT unionall(friendsposts, pagesposts)
FROM (MATCH {class: Person, where: (username="john.doe"),
  as: john}-friend-{class: Person}<-postedby-{class: Post,
  where: posted_time >(sysdate().asLong()-24*60*60*1000),
  as: friendsposts}),
  (MATCH {as: john}-likes->{class: Page}<-postedby{class: Post,
  where: posted_time >(sysdate().asLong()-24*60*60*1000),
  as: pagesposts})
ORDER BY posted_time desc
```

Q4: *Retrieving photos of a user's friends who have friends who like the Bacon Facebook page.*

*Q*4 is performed as a pure pattern match with multiple paths. Two pattern matches find photos of friends of "john.doe" and filter those of his friends who have friends who like the "Bacon" page. In order for nodes with the alias `photos` to end up in the query results they must have a `tagged` edge to a node given the `johnsfriends` alias which has passed both expressions.

```
MATCH {class: Person, where: username = "john.doe"}-friend-
  {as: johnsfriends}<-tagged-{as: photos},
  {as: johnsfriends}-friend-{}-likes->{class: Page,
  where: aliasURL = "Bacon"}
RETURN photos
```

4.5 Neo4j

In Neo4j the data is modelled as a graph just as in Figure 4.1. Note that all edges must be directed but may be traversed bidirectionally. Bidirectional edges should be implemented as two oppositely directed edges.

As mentioned in the previous section, pattern-matching in Cypher queries is similar to OrientDB's pattern match expressions. Cypher matches node and relationship patterns to the graph to perform SCRUD operations. Nodes are assigned labels which can be indexed. Because this syntax is assumed to be less familiar to readers than SQL might be, the Cypher syntax is described shortly below. See also Section 2.4.2 for further explanation of the syntax.

The Cypher syntax represents nodes with parentheses that can optionally be filled with property value filters, labels, or context variables. In the following example, we match for a node labeled as a Person, with the `firstname` property value "Alice". In order to retrieve information from this node we must assign it a variable within the query – in this case we set the variable `alice` to represent the node.

```
MATCH (alice:PERSON {firstname: 'Alice'}) RETURN alice.lastname
```

Relationships are traversed using an arrow syntax, where variables, labels, and properties can be specified in a bracket in the middle of the arrow. Using variables we can retrieve property values from relationships as well as nodes. See an example relationship query below.

```
MATCH ()-[rel:WROTE]->() RETURN rel.year
```

Q1: *Retrieving the names of all mutual friends of two users.*

$Q1$ is implemented in Cypher by matching nodes that have `FRIEND` relations with both a Person node holding the username "john.doe" and a Person node holding the username "jane.doe". The relationship direction is not specified as the `FRIEND` relationship can be considered as bidirectional. Nodes that match the pattern are assigned the context variable `mutual` so that their properties may be extracted.

```
MATCH (:PERSON {username: "john.doe"})-[:FRIEND]-
  (mutual)-[:FRIEND]-(:PERSON {username: "jane.doe"})
RETURN mutual.firstname, mutual.lastname
```

Q2: *Retrieving the names of a user's friends that live in Bergen, Hordaland and like NTNU's Facebook page.*

$Q2$ is implemented by assigning the context variable `friend` to nodes holding the `PERSON` label and the `currentcity` value "Bergen, Hordaland", with a `FRIEND` relation to the `PERSON` node with the username "john.doe" and a `LIKES` relation to a node with the `PAGE` label and the `aliasURL` value "ntnu.no". The names of these `friend` nodes are returned.

```
MATCH (:PERSON {username: "john.doe"})-[:FRIEND]-
  (friend:PERSON {currentcity: "Bergen, Hordaland"})
  -[:LIKES]->(:PAGE {aliasURL: "ntnu.no"})
RETURN friend.firstname, friend.lastname
```

Q3: *Retrieving all posts by a user's friends and followed pages from the past 24 hours.*

Implementing $Q3$ in Cypher does not require multiple pattern expressions, as Cypher allows operators within the pattern. Thus, we are able to match for posts by pages "john.doe" follows as well as his friends, using only one pattern-matching expression. The query searches for the node holding the username "john.doe" and traverses his `FRIEND` and `LIKES` relationships to find nodes that have incoming `postedby` edges from `POST` nodes. The `POST` nodes that match the pattern are filtered on their `posted_time` value, requiring it to be a timestamp within the past 24 hours (Neo4j uses milliseconds since the Unix Epoch as timestamps). Posts that match the pattern and pass the filter are returned, sorted by time in descending order.

```
MATCH (:PERSON {username:"john.doe"})
  -[:FRIEND|:LIKES]-()<-[:POSTEDBY]-(p:POST),
WHERE p.posted_time > (timestamp() - 24*60*60*1000)
ORDER BY p.posted_time DESC
RETURN p
```

Q4: *Retrieving photos of a user's friends who have friends who like the Bacon Facebook page.*

While $Q3$ could be resolved using one pattern, $Q4$ must use two patterns as it involves three incoming edge patterns on the same node – "john.doe"'s friends. The first pattern seeks out friends of "john.doe" who have friends who like the "Bacon" page, while the second finds the photos these friends of his have been tagged in. The two patterns are separated by a comma and share the nodes assigned the variable `n` – these nodes must match with both patterns.

```
MATCH (:PERSON {username:"john.doe"})-[:FRIEND]-(n)-[:FRIEND]-()
  -[:LIKES]->(:PAGE {aliasURL: "Bacon"}), (photos)-[:TAGGED]->(n)
RETURN photos
```

Chapter 5

Evaluation

In the previous chapter, we showed how MySQL, OrientDB, and Neo4j could be used to store and query Facebook's social graph. This chapter presents an evaluation of the three databases with regards to the use cases of Facebook's social graph and graph search.

5.1 Compatibility of the data model and the programming model

Among the most important qualities of the data model of a database are the similarity of the data model with the real-world structure of the data, and the similarity of the data model with the data structure implemented in the application software.

There is a clear mismatch between the relational model of MySQL and the graph model used in the software for Facebook's social graph. Mapping between the two delays and complicates the process of performing transactions and queries. The issues associated with the dissimilarity between the model used by the software and the model used in the database are described in the literature using the term *impedance mismatch*. [15]

The mismatching issue is avoided in Neo4j and OrientDB as the social graph may be mapped more directly into their data models. The social graph holds highly connected data, which these databases are primarily made for. Comparing the two using non-distributed implementations, [67] found that Neo4j handled big graphs more efficiently than OrientDB.

5.2 Usability

With regards to querying the data, OrientDB and Neo4j provide the ability to query using pattern-matching expressions, which we find to be the most comfortable way to perform these queries. The relational queries MySQL applies are better suited for aggregation, while deep graph traversal and pattern-matching strategies result in complex JOINs and nested queries. Relational databases are not designed to perform deep traversal queries. [68]

Querying the social graph using Cypher is on the other hand very comfortable and efficient. Relevant queries such as deep graph traversal, computing the distance between two nodes along a relationship type, and matching patterns within the graph are easily implemented. Meanwhile, Cypher is not as well known among developers as SQL is. Using Cypher in a large application requires recruiting developers who have experience with the language or training previously hired developers.

OrientDB seems to be a golden mean between the two - making use of SQL as their query language which most developers would be familiar with while also supporting syntax for traversing the edges of the graph. This combination allows developers with SQL experience to ease into declarative graph queries.

5.3 Maturity

Relational databases, including MySQL, are well established and widely used all over the world. There is no lack of academic literature on topics concerning relational databases, neither is there a lack of tools or software related to managing such databases. Both academic and commercial projects are known to adopt relational databases, ranging from small school projects to critical projects within large corporations. The fact that large corporations are willing to trust relational databases with their most valuable data means that they have reason to invest time and money into maintenance, support, and further development of the technology. Oracle, the owners of MySQL, facilitate a large online community for MySQL users including forums, documentation, blogs, podcasts, and the opportunity for other developers to contribute code. [69] [68] Searching for “mysql” on the website for the npm registry [70], a platform for JavaScript developers to share code and tools, returns 2095 results. The same search performed on the MvnRepository [71],

a search engine for Maven packages, returns 159 results.

Neo4j has a much less established community. Most of the online community for Neo4j is hosted by Neo Technologies, while support outside of the company's website is sparse. [68] There is a notable amount of academic papers discussing and analyzing the database, though much of the academic comment was based on the 1.x.x release of Neo4j and published before the release of version 2.0 in late 2013. At the time of writing this thesis, the newest stable release was version 3.0.0 in late April 2016. A general search for related tools on the npm.js website returns 162 results. The MvnRepository returns 298 results for the same search.

OrientDB's user community is also mostly facilitated by their parent company, Orient Technologies. The online user community is somewhat active, with a handful of discussion threads being posted daily to the forum provided by the company. Academic papers discussing NoSQL or graph databases will often mention OrientDB among other well-known databases, but few papers discuss OrientDB in-depth. Searching for "OrientDB" on the npm.js website returns 49 results at the time of writing. The MvnRepository found 67 results for the same search.

5.4 Cost of migration

Migrating data from one database solution to another is costly, as both practically moving the data and recreating the complex data structure in another database are time-consuming projects. In addition, the process of rewriting the software that communicates with the database can be very expensive for large applications such as Facebook. The high cost of migration is a concern which should not be taken lightly when considering a move from relational databases into NoSQL.

The migration would have to be performed while the database is online because of the uptime requirements of social networking services. Such a migration could be performed on a few database sites at a time by moving a site's load to a neighbouring site while the migration is run, then rebalancing the load once the migration for that site has completed. As only some sites are migrated at a time this would take longer but would ensure data availability during the process.

5.5 ACID compliance

Neo4j claims to be ACID compliant, a claim supported by [7]. OrientDB claims to fully support ACID transactions, which [72] and [73] confirm.

Durability and replication are critical requirements for Facebook's social graph, and while they can afford to sacrifice immediate consistency, these concerns are non-negotiable. The fact that both OrientDB and Neo4j offer ACID transactions makes them more attractive as prospective storage solutions. Sharding seems to be supported better by the Enterprise edition of Neo4j than by the current version of OrientDB, though the replication opportunities in the community edition of OrientDB far exceed those offered by the community edition of Neo4j.

The availability and scalability offered by NoSQL solutions are very interesting for applications such as Facebook, where availability requirements are extremely high. The BASE approach would be appealing for this reason, but guarantees of data persistence simply cannot be deprioritized. In [47] Facebook comments that scaling their MySQL clusters is a task which is hard to combine with their availability demands, and that the administration costs of the clusters are higher than they expect from NoSQL services.

5.6 Use within Facebook

Facebook needs their main storage to be compliant with several very different applications. Graph Search, Messenger, Insights, and News Feed have diverging needs regarding read/write optimization, indexing, replication, and availability. [47]

Should Facebook migrate their main storage into a graph database, some of the applications within Facebook would need to perform a reverse mapping into a relational model. This task is not entirely straightforward as a graph model in many cases can badly misrepresent relational data. [68]

A graph-based database would be more appropriate as a storage layer between the main storage and social graph applications within Facebook, simplifying the communication between graph-based software and a relational database.

Chapter 6

Conclusions

In this chapter, we present improvements and disadvantages to choosing a graph database rather than a relational database for social networks, and conclude the project.

6.1 Advantages

By using a complete graph implementation as their main storage in place of a relational database, social network services would see an improvement in the ease of development and performance of graph-based applications such as graph search, graph analytics, and pattern recognition.

The multi-model engine makes OrientDB more useful than Neo4j in graph-based applications that also offer a combination of non-graph services such as messaging, data aggregation, and managing documents.

In services where the data model is not finally determined and may be subject to change, many NoSQL databases have the benefit of flexible schemas which can lead to lower administration costs of iterative changes.

Largely, NoSQL databases claim to offer better scalability with regards to administration costs and flexibility than what relational databases are able to offer.

6.2 Disadvantages

Sharding in OrientDB is not well documented, and Neo4j only offers cache sharding in their Enterprise edition. Well-documented replication strategies

are needed for large-scale applications such as Facebook where availability and data integrity are critical concerns.

Because of the high cost of data migration, the level of improvement from today's architecture must be very high in order for a migration to be worth performing.

The developer communities for Neo4j and OrientDB are significantly less established than for MySQL, and the tools and literature available come up short in comparison. Developers must to a larger degree rely on self-made tools than when working with MySQL.

While mapping from relational data to a graph model is non-optimal, the reverse is also true. An application based on graph storage that also handles relational data may experience this poorly represented in a graph structure.

6.3 Conclusion to hypothesis

While there are some great advantages to using graph-based storage for graph-based applications, the current infrastructure around the presented graph databases does not appear to be robust enough for them to be used as the main storage for social networking services such as Facebook and Tumblr.

Using NoSQL databases as a layer between relational storage and the application software offers many of the same advantages while conserving the advantages of relational storage but involves extra implementation and administration costs. Facebook have implemented this architecture by using Tao as a layer between the front-end software and their MySQL servers.

Bibliography

[1] Maeve Duggan, Nicole B. Ellison, Cliff Lampe, Amanda Lenhart, and Mary Madden. Social Media Update 2014. `http://www.pewinternet.org/2015/01/09/social-media-update-2014/`, January 2015. Last checked: 24.11.2015.

[2] Facebook. Company Info. `https://newsroom.fb.com/company-info/`, March 2016. Last checked: 31.05.2016.

[3] Statista The Statistics Portal. Leading social networks worldwide as of November 2015, ranked by number of active users (in millions). `http://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/`, November 2015. Last checked: 24.11.2015.

[4] Neo Technology. Why Graph Databases? `http://neo4j.com/why-graph-databases/`, 2015. Last checked: 24.11.2015.

[5] Kristine Steine. Data Storage In Social Networks, December 2015. unpublished prestudy, conducted as part of the MTDT program at NTNU.

[6] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008.

[7] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA March 23rd-24th*, 2013.

[8] Orient Technologies. OrientDB Multi-Model Open Source NoSQL DBMS. `http://orientdb.com/docs/last`, 2016. Last checked: 15.5.2016.

[9] Rachel Roumeliotis. The Future Is Graph Databases. http://radar.oreilly.com/2013/06/the-future-is-graph-databases-2.html, 2013. Last accessed: 6.12.2015.

[10] Neo Technology. World Cup Fun with Neo4j. http://worldcup.neo4j.org/, 2014. Last checked: 16.12.2015.

[11] U.S. Department of State. Department of State Organization Chart. http://www.state.gov/s/d/rm/rls/perfrpt/2007/html/98613.htm#BackFromLD, 2007. Last checked: 16.12.2015.

[12] Neo Technology. Transforming Logistics - Real-time Routing and Tracking with Neo4j. http://neo4j.com/case-studies/global-500-logistics/, 2015. Last checked: 16.12.2015.

[13] Adrian Silvescu, Doina Caragea, and Anna Atramentov. Graph Databases, 2002.

[14] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[15] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015.

[16] Konstantinos Barmpis and Dimitrios S. Kolovos. Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models. *Journal of Object Technology*, 13(3):3:1–26, July 2014.

[17] Robin Hecht and Stefan Jablonski. NoSQL evaluation: A use case oriented survey. In *2012 International Conference on Cloud and Service Computing*, CSC 2011, pages 336–341. CSC, 2011.

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

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[21] The Apache Software Foundation. Apache HBase. `http://hbase.apache.org/`, 2015. Last checked: 10.12.2015.

[22] The Apache Software Foundation. CouchDB: A Database for the Web. `http://couchdb.apache.org/`, 2015. Last checked: 10.12.2015.

[23] MongoDB, Inc. The MongoDB 3.2 Manual. `https://docs.mongodb.org/manual/`, 2015. Last checked: 9.12.2015.

[24] MAPS Student Organization. Med BEKK: Workshop i grafdatabaser. `http://foreninger.uio.no/maps/arrangementer/med-bekk%3A-workshop-i-grafdatabaser.html`, 2014. Last checked: 9.12.2015.

[25] Linkurious. Whiplash for cash : using graphs for fraud detection. `https://linkurio.us/whiplash-for-cash-using-graphs-for-fraud-detection/`, 2014. Last checked: 9.12.2015.

[26] Neo Technology. Telenor's Resource Authorization Challenge. `http://neo4j.com/case-studies/telenor/`, 2015. Last checked: 9.12.2015.

[27] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[28] Neo Technology. Neo4j Manual. `http://neo4j.com/docs/`, 2015. Last checked: 9.12.2015.

[29] Neo Technology. Neo4j Licensing. `http://neo4j.com/licensing/`, 2016. Last checked: 19.5.2016.

[30] Florian Holzschuher and René Peinl. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4J. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 195–204, New York, NY, USA, 2013. ACM.

[31] Neo Technology. Use Case: Social Network. http://neo4j.com/use-cases/social-network/, 2015. Last checked: 9.12.2015.

[32] Orient Technologies. OrientDB on Github. https://github.com/orientechnologies/orientdb, 2016. Last checked: 15.5.2016.

[33] Orient Technologies. OrientDB vs Neo4j. http://orientdb.com/orientdb-vs-neo4j/, 2016. Last checked: 19.5.2016.

[34] Orient Technologies. OrientDB: Why OrientDB? http://orientdb.com/why-orientdb/, 2016. Last checked: 19.5.2016.

[35] Orient Technologies. OrientDB: Success Stories. http://orientdb.com/success/, 2016. Last checked: 19.5.2016.

[36] Orient Technologies. OrientDB Enterprise. http://orientdb.com/orientdb-enterprise/, 2016. Last checked: 19.5.2016.

[37] Maeve Duggan, Nicole B. Ellison, Cliff Lampe, Amanda Lenhart, and Mary Madden. Demographics of Key Social Networking Platforms. http://www.pewinternet.org/2015/01/09/demographics-of-key-social-networking-platforms-2/, January 2015. Last checked: 14.12.2015.

[38] Facebook, Inc. Facebook Privacy Basics. https://www.facebook.com/about/basics/, 2015. Last checked: 12.12.2015.

[39] Facebook, Inc. Facebook Help Centre. https://www.facebook.com/help/, 2015. Last checked: 12.12.2015.

[40] Facebook, Inc. Facebook For Developers — The Graph API. https://developers.facebook.com/docs/graph-api, 2015. Last checked: 14.12.2015.

[41] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The Anatomy of the Facebook Social Graph. *CoRR*, abs/1111.4503, 2011. http://arxiv.org/abs/1111.4503.

[42] Yi Chang, Lei Tang, Yoshiyuki Inagaki, and Yan Liu. What is Tumblr: A Statistical Overview and Comparison. *ACM SIGKDD Explorations Newsletter*, 16(1):21–29, 2014.

[43] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, et al. Unicorn: A System for Searching the Social Graph. *Proceedings of the VLDB Endowment*, 6(11):1150–1161, 2013.

[44] Dhruba Borthakur. Under the Hood: Building and open-sourcing RocksDB. `https://code.facebook.com/posts/666746063357648/under-the-hood-building-and-open-sourcing-rocksdb/`, November 2013. Last checked: 15.12.2015.

[45] Sanjay Ghemawat and Jeff Dean. LevelDB. `https://github.com/google/leveldb`, 2015. Last checked: 15.12.2015.

[46] Zelaine Fong and Rishit Shroff. HydraBase – The evolution of HBase@Facebook. `https://code.facebook.com/posts/321111638043166/hydrabase-the-evolution-of-hbase-facebook/`, June 2014. Last checked: 15.12.2015.

[47] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[48] Mark Callaghan. MySQL and Database Engineering. `https://code.facebook.com/posts/624181104289118/mysql-and-database-engineering-mark-callaghan/`, March 2012. Last checked: 15.12.2015.

[49] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[50] The Apache Software Foundation. Apache Hive. `http://hive.apache.org/`, 2015. Last checked: 15.12.2015.

[51] The Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`, 2015. Last checked: 15.12.2015.

[52] The Apache Software Foundation. Apache Thrift. http://thrift.apache.org/, 2015. Last checked: 15.12.2015.

[53] The Apache Software Foundation. Apache Giraph. http://giraph.apache.org/, 2015. Last checked: 15.12.2015.

[54] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[55] Alessandro Presta and Alon Shalita. Large-scale graph partitioning with Apache Giraph. https://code.facebook.com/posts/274771932683700/large-scale-graph-partitioning-with-apache-giraph/, April 2014. Last checked: 15.12.2015.

[56] Tumblr, Inc. Press Information — Tumblr. https://www.tumblr.com/press, 2015. Last checked: 13.12.2015.

[57] Jason Mander. Tumblr and Instagram have the youngest audiences. http://www.globalwebindex.net/blog/tumblr-instagram-audiences, 2014. Last checked: 13.12.2015.

[58] Jason Mander. TPinterest and Tumblr are the fastest growing social networks. http://www.globalwebindex.net/blog/pinterest-and-tumblr-are-the-fastest-growing-social-networks, 2015. Last checked: 13.12.2015.

[59] Tumblr, Inc. Help Center — Tumblr. https://www.tumblr.com/help/, 2015. Last checked: 12.12.2015.

[60] Tumblr, Inc. API — Tumblr. https://www.tumblr.com/docs/en/api/v2, 2015. Last checked: 13.12.2015.

[61] Todd Hoff. Tumblr Architecture - 15 Billion Page Views a Month and Harder to Scale than Twitter. http://www.highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html, February 2012. Last checked: 13.12.2015.

[62] Tumblr, Inc. Jetpants. `https://github.com/tumblr/jetpants`, 2013. Last checked: 14.12.2015.

[63] Salvatore Sanfilippo. Redis. `https://github.com/antirez/redis`, 2015. Last checked: 14.12.2015.

[64] The Apache Software Foundation. Apache Kafka. `http://kafka.apache.org/`, 2015. Last checked: 14.12.2015.

[65] Josh Halliday. David Karp, founder of Tumblr, on realising his dream. `http://www.theguardian.com/media/2012/jan/29/tumblr-david-karp-interview`, January 2012. Last checked: 14.12.2015.

[66] Orient Technologies. OrientDB SQL - MATCH. `http://orientdb.com/docs/last/SQL-Match.html`, 2016. Last checked: 24.5.2016.

[67] Sotirios Beis, Symeon Papadopoulos, and Yiannis Kompatsiaris. *New Trends in Database and Information Systems II: Selected papers of the 18th East European Conference on Advances in Databases and Information Systems and Associated Satellite Events, ADBIS 2014 Ohrid, Macedonia, September 7-10, 2014 Proceedings II*, chapter Benchmarking Graph Databases on the Problem of Community Detection, pages 3–14. Springer International Publishing, Cham, 2015.

[68] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.

[69] Oracle Corporation. MySQL Developer Zone. `http://dev.mysql.com/`, 2016. Last checked: 27.5.2016.

[70] npm, Inc. npm. `https://www.npmjs.com/`, 2016. Last checked: 27.5.2016.

[71] MvnRepository. The Mvn Repository. `http://mvnrepository.com/`, 2016. Last checked: 27.5.2016.

[72] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. Comparison and classification of nosql databases for big data. In *Proceedings of International Conference on Big Data, Cloud and Applications*, 2015.

[73] Vivek Mishra. *Beginning Apache Cassandra Development*, chapter Titan Graph Databases with Cassandra, pages 123–151. Apress, Berkeley, CA, 2014.