Distributed Database Design for Social Network Graphs

D.T.A. van Beelen, BSc CS

Date of acceptance: August 12, 2008

Master Course Software Engineering

Thesis Supervisor: Jurgen J. Vinju

Internship Supervisor: Henk Punt

Company: Hyves (Startphone Limited)

Availability: public domain

Universiteit van Amsterdam,
Hogeschool van Amsterdam,
Vrije Universiteit

Contents

Abstract

In the area of high volume websites, database scalability is an ongoing challenge. Many solutions exist, using various partitioning techniques, to distribute data over multiple servers. For highly interrelated data however, like social networking graphs, up-scaling is often preferred over out-scaling in order to avoid the latencies caused by collecting data from many different servers.

In this thesis I will propose a new architecture for a distributed database that efficiently handles distribution of graph data over multiple server nodes. The design is optimized so that common queries which are performed on social networking graphs can be completed using a total of three server nodes at most, thereby limiting the overhead that would occur from inter-node communication.

Preface

First of all, I would like to thank Hyves as a whole for the great opportunity of doing an intership at their company. Special thanks go to Koen Kam for offering me this position as an intern and to Henk Punt for being my supervisor during the internship.

Also, I would like to thank Jurgen Vinju and Hans Dekkers for supervising and guiding me on my thesis.

Last, but not least, I would like to thank all the people who supported me the past few months during what has at times been a rough ride.

Chapter 1

Introduction

1.1 Background

Hyves (http://www.hyves.nl/) is the largest social networking website in the Netherlands. Hyves has seen tremendous growth over the past years. Currently over 6.7 million members are subscribed, and the site generates around 150 million pageviews daily.

At the center of a social networking website is the social network that connects its members together. Members can be friends with each other, at which point they enter each other's *friend network*. Apart from direct friends – the first degree friend network – there are also friends of friends – the second degree friend network. Of course, a third degree network also exists, and so on.

At the time of writing, the social network consists of 6.7 million members, and over 245 million friend relations. Needless to say, the network is continuously growing.

Currently, the network is stored in a single MySQL table. This table consists of two columns, *from_member_id* and *to_member_id*, and thus every row in this table represents a directional relation between two members. Two indices exist on this table, making it possible to look-up relations in both directions. The database containing this table is also referred to as the *friend database*.

Using the current database implementation, Hyves is expecting scalability

problems in the (near) future. The large amount of memory required to store the table is main cause for these problems. The indices alone are 12GB, and the data itself accounts for about 4GB. Right now, the servers can still keep all data and indices in main memory at once. However, as the number of relations is growing at a quadratic rate compared to the amount of members, and the amount of members has roughly doubled last year, Hyves should be preparing for a situation where this network does no longer fit into main memory.

The current database is placed in a master-slave setup. Within this setup, the master handles all the writes to the database, while the slaves handle all the read queries. Modifications to the table which are done on the master are replicated to the slaves. Because there are multiple slaves, the read capacity of the database can be scaled virtually without limit. Writes to the database however, are still limited by the capacity of the master.

1.2 Motivation

As Hyves continues to grow, solutions should be sought for the bottlenecks that are being faced. Of course, up-scaling is always an option. Up-scaling, as opposed to out-scaling, is the process of adding more powerful hardware in order to handle a problem. Out-scaling on the other hand, adds more servers, thus spreading the task over multiple machines.

Hyves has a strong commitment to using commodity hardware for its servers. Commodity hardware is generally cheaper, can be ordered on a short period of time, and can be replaced quickly in case of failure. At the moment though, such commodity servers do not scale further than 32GB of main memory, and servers with this capacity are already in use for the friend database.

This means if Hyves were to apply further up-scaling for the friend database, it would have to buy itself some really heavy, and expensive, servers. Not only would this result in increased hardware costs, but costs for server management would also increase due to added complexity in logistics, increased dependence on the chosen supplier, and so on.

Because of the undesirability of further up-scaling, I will be researching solutions to be able to effectively apply out-scaling to the friend database.

1.3 Research Goals

The goal of this research is to find a distributed database setup that will enable Hyves to apply out-scaling to the friend database such that no expected scalability problems remain in the foreseeable future. In particular, I will look for a solution that will not have issues with memory consumption on commodity hardware nor be limited by the capacity of a single master server.

Additionally, I will aim for a solution that is highly efficient and will have as little overhead as possible. While these goals may appear somewhat conflicting, I believe it is possible to find a good compromise that will satisfy both.

With these goals in mind, I have developed the following two hypotheses:

1. Memory consumption of the friend database can be reduced considerably by using an alternative data representation, giving sufficient room for further growth of the social network.
2. Communication overhead between server nodes can be kept at a minimum using the right caching techniques.

Both hypotheses will be validated against the proposed solution, which is described in detail in chapter 3.

1.4 Scope

As mentioned in the previous section, the goal of this research is to find a scalable, distributed database setup for the Hyves friend database. By doing so, I will focus on partitioning the social network over multiple server nodes, implementing caching to avoid communication overhead, and on reducing memory usage.

This research will also strongly focus on the queries on the social network which are important to Hyves. I will not aim to implement a general purpose database, and the database solution will not have to support random queries outside the scope of the network queries.

Also, while it is a goal to partition the social network over multiple nodes, the emphasis here is on partitioning in order to achieve scalability. The partitioning does not have to be particularly smart such as graph partitioning algorithms which aim to minimize cross-set edges [6].

Chapter 2

Context

2.1 The Hyves Network

The Hyves Network is a large social network [14] that is used by its members to stay in touch with each other. Members are connected by creating friendships[1]. From the user's point of view, their direct friends are the people they are most interested in and which they interact the most with. Friends of friends are also interesting, though they are mostly interesting in order to find people to create new friendships with.

Unsurprisingly, this social network can be clearly represented in the form of a graph. Members are represented by the vertices of the graph, while friendships are expressed as edges. Because friendship relations are always bidirectional, this graph is *undirected*.

At the time of writing, the social graph consists of 6.7 million vertices, and 245 million undirected edges (or 420 million directed edges using Hyves' current implementation). Because of the high fan-out of the vertices – vertices on average are connected to about 73 other vertices – and the fact that the total number of edges has grown at a quadratic rate compared to the number of vertices, we can speak of a dense graph [10].

Not all the vertices of the graph are connected, which is easily supported by the fact that there is a considerable amount of members which have made

[1] It should be noted that all relationships on Hyves are considered friendships, though many users will also have relatives, colleagues or other acquaintances in their friends list.

no friendship relations at all. Those are most likely attributed to fresh new members, individuals (most likely from outside the Netherlands) that found no friends on the network, or accounts which have been created in the past by bots. It is likely though there also exist small isolated networks which are not connected to the majority of the other vertices.

While no direct evidence is available, there is also a strong indication that the network has the topology of a scale-free network [1]. In a scale-free network, network nodes are not connected using a random distribution. Instead, there are relatively few nodes, which are called hubs, which have exceptionally many connections to other nodes.

This is supported by the fact that members on average have about 73 friendship relations, and only 3.5% has more than 300 friendships. Furthermore, only 0.03% has more than 1000 friendships, and a mere total of 6 members (which are all celebrities) have over 100,000 friendship relations. This shows a strong correlation to the power law distribution seen in scale-free networks. A graph showing the distribution is given in figure 2.1.

All together, we can state the Hyves network is a dense, undirected, unconnected graph, that exhibits characteristics of a scale-free network.

2.2 Other Social Networks

Hyves is not the only social networking site in existence, and several other, competing sites exist which are faced with similar challenges when it comes to scaling the social network graph. Understandably, little information is disclosed by competitors on how such scalability issues have been resolved. However, some information that could be found includes:

- *Hi5* is the third largest social networking site in the world with a total of over 70 million members[2]. In October 2007, they gave a great presentation on database scalability and their own experiences from the huge growth they have been through. Unfortunately, no specifics on the social network graph were mentioned. [4]
- *LinkedIn* is a more business oriented social networking site. Interesting to note is that with 22 million members they have over 3 times as

[2]While the video from 2007 reported 70 million members, current numbers actually put them at over 80 million members.

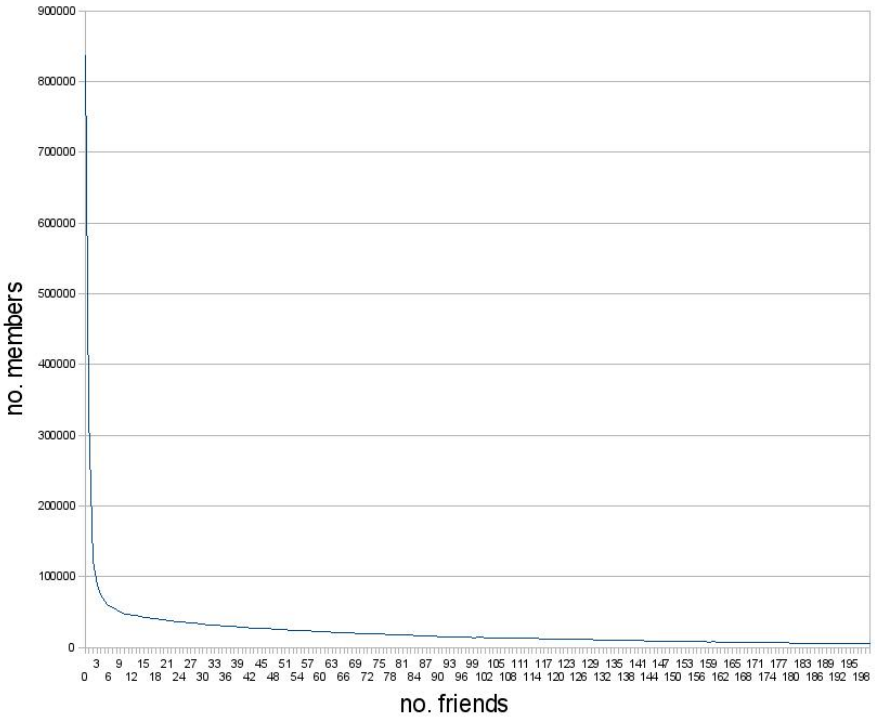

Figure 2.1: Distribution of the amount of friends per member.

much members as Hyves, but their social network "only" contains 120 million edges (though unmentioned, these are likely to be directional), compared to the 420 million edges from Hyves. This makes the social network graph from Hyves many times as dense as the one from LinkedIn. LinkedIn reportedly relies primarily on up-scaling for their friend database, mentioning that partitioning graphs is hard. [3]

2.3 Related Work

Of course, many techniques and solutions exist in the field of database scalability. I will list some of the solutions which are relevant in the context of this research:

- *Sharding* is the technique of splitting a database into mostly independent chunks, or *shards*. This technique is often applied to relational databases, where each shard contains the same set of tables, but a different subset of the rows. Sharding is an easy, and already widely applied method of database scaling, but is not well-suited to highly inter-related data like graphs, because queries can only access data from one shard at a time. [5]

- *Object databases* are an alternative to relational databases. Whereas relational databases store their data in tables, consisting of columns and rows, object databases store data in objects of different classes, just like object oriented programming languages. While controversial for use as a general purpose database, object databases can offer astonishing performance in certain scenarios. [7]

- *BigTable* is Google's approach to building a highly scalable database. BigTable stores its data in big, three-dimensional tables which are transparently stored in a distributed file system (GFS). Unlike graph processing though, where many small amounts of data are queried, BigTable is mostly optimized for high throughput on large amounts of data. [2]

Chapter 3

Proposed Solution

In this chapter, I propose a new architecture for the friend database which will both overcome the scalability issues being faced by Hyves, and at the same time provide outstanding performance.

The architecture I propose is based around three pillars:

- An object-based data representation is chosen over a relational table-based representation. This representation is more compact, obsoletes most memory consuming indices, and provides good data locality.
- Partitioning (sharding) is applied to spread both load and memory usage over multiple server nodes making cheap out-scaling possible.
- Caching of objects is introduced to obsolete the vast majority of internal communication overhead and to reduce query latencies.

An overview of the architecture is given in figure 3.1.

In the next section, I will tell some more about how the object representation works, my motivation for using an object representation and the actual data structures used.

In section 3.2 I will explain how the objects are partitioned over multiple nodes and how objects are located.

In section 3.3 I will talk about the various queries that are supported, and how they are implemented.

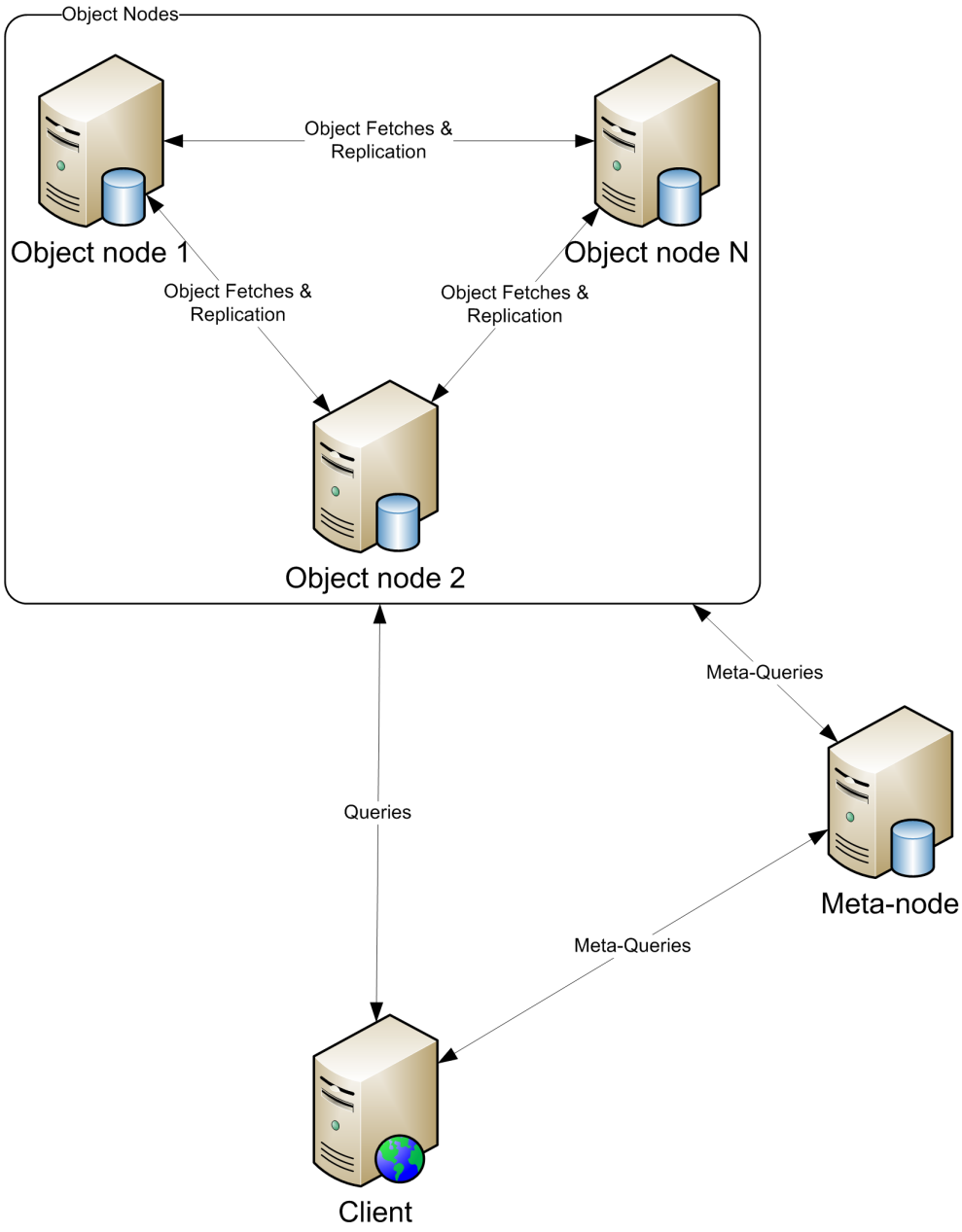

Figure 3.1: Overview of the proposed architecture.

Section 3.4 will cover the caching mechanism, and how it helps to reduce the overhead for query execution.

Finally, in section 3.5 I will explain how persistence is implemented and how data integrity is preserved.

3.1 Object Representation

One of the main design choices for the proposed solution is to use an object-based data representation rather than a relational table-based representation.

Using this representation, every object represents exactly one member, and every object contains a list of IDs, which are nothing more than simple integers, of the first degree friends of that member. The list of friend IDs is always sorted, reducing the time complexity of looking up a specific ID from $O(n)$ to $O(\log_2 n)$.

My main motivation for selecting this type of representation is twofold:

- Memory usage is cut down tremendously. The main reason for this is that not every individual friendship relation is indexed anymore, but only the member objects are indexed. Also, when storing relations in a table, every relation specifies both the source and the target of the relation, whereas all relations in an object share the same source specified in the object.
- Except for the look-up of a single relation, all queries which are interested in one relation from a certain member are also interested in the other relations from that member. Therefore, spatial locality is increased by storing these relations together. Because of this locality of reference, such objects are also good atomic units for distribution.

For a more detailed comparison of object-based representation versus table-based representation, I refer to appendix A.

3.1.1 Data Structure

The data structure used for representing member objects is defined in table 3.1.

Field	Description	Type
memberId	Unique ID of the member	4 bytes unsigned integer
isOnewayMember	Flag indicating whether the member is a oneway member (see section 3.4.2)	1 byte boolean
numFriends	Integer indicating the number of friends of the member	4 bytes unsigned integer
friends	Array with IDs of all friends	*numFriends* * 4 bytes unsigned integer

Table 3.1: Data structure for member objects.

In addition to the memory for the objects themselves, nodes will need to store for every object whether they are the master node for the object and how many references they have to the object (see section 3.4.1 about reference counting). These attributes, a one byte boolean and a four byte integer respectively, are not properties of the actual objects however.

3.1.2 Indexing

An index on the *memberId* field will be necessary for quick look-up of objects. While a B+ Tree index or a hash table could be used, I have chosen to just use a simple one-dimensional array for this. The reason for this is that an array is both extremely simple and extremely fast.

Using such an array may not be very memory efficient if a node contains only relatively few objects, but still compared to the memory required for storing the objects, the memory cost is negligible.

For example, if the array is allocated for 25 million members, the memory cost will be 200MB on 64-bit processors, which accounts for 0.6% of main memory on servers with 32GB RAM.

It should be noted though that the highest member ID is always a bit higher than the total amount of members due to members that deleted their accounts. Therefore, in this example the array should be increased in size when the highest member ID approaches the 25 million.

3.1.3 Memory Usage

Now that we know the data structure used and the type of indexing, we can construct formulas for determining the amount of memory required by a node:

$$index\_size = 1.5 * 8 * m$$
$$data\_size = 20 * n + 4 * f * n$$
$$mem\_size = index\_size + data\_size$$

Where m is the total number of members, f is the average number of friends per member, and n is the number of objects stored on that particular node.

While it may be true that the average number of friends per member does not have to be same across all nodes, in practice this is generally the case. Of course, there might be a small error margin though.

The index size is calculated by allocating 8 bytes for every pointer in the index array (all servers at Hyves have 64-bit processors). The factor 1.5 is used as slack to account for the offset between the highest member ID and the total number of members (which has a factor of approximately 1.1 now), and the fact that enough memory should be allocated to allow for new members to be created.

The data size is calculated by taking the memory required for the data structure listed above, plus the memory required for storing master information and the number of references, which makes a total of 20 bytes in a memory-aligned structure, excluding friends. 4 bytes are added for every friend.

The total memory required on every node for storing its objects is then calculated by summing the index size and the data size. The result is the memory usage in bytes.

3.2 Partitioning

In order to allow for out-scaling, member objects are distributed over multiple nodes. This means that every node is responsible for a different subset of all objects.

I will call the objects for which a node is responsible that node's *master*

objects. The other way around, the node responsible for an object is that object's *master node*.

3.2.1 The Meta-Node

There is no static scheme for determining which node is an object's master node. Therefore one special type of node exists, the *meta-node*. The meta-node contains a list, the *meta-index*, that maps every object to its master node.

When a client wants to submit a query it should therefore first contact the meta-node to find out which node to submit the query to.

Because all clients, as well as other nodes, will submit many queries to the meta-node, there is a risk of the meta-node becoming a new scalability bottleneck. Should a single meta-node not be able to process all queries however, there is always the possibility of using multiple meta-nodes. This is rather easy to do because the object-to-node map is a simple one-dimensional data structure which is trivial to partition over multiple nodes. For this thesis though, I will assume a single meta-node is sufficient.

Finally, because the meta-index is mostly static, this index may even be cached on the object nodes as well.

For new members, new entries are created in the index and occasionally entries get removed for deleted members, but entries for existing members never change (assuming no dynamic redistribution is going on). Therefore, a lazy caching mechanism would be ideally suited for this situation.

For simplicity's sake though, I will assume no caching for the meta-node is implemented in the rest of this thesis.

Meta-Index Consistency

One potential problem with the use of a meta-node is that there is a risk of the meta-index to become inconsistent with the actual state of the object nodes.

To minimize this risk all updates to the meta-index should be performed in a single transaction which is coordinated with the object nodes. This can be achieved using a two-phase commit protocol [11], which will guarantee that

either both nodes complete the transaction, or both will fail, thus keeping a consistent state.

Nevertheless, should a worst-case scenario occur where the meta-index has become corrupted, it would still be possible to rebuild the index by querying all object nodes for the master objects they contain.

3.3 Query Handling

Important to consider are the queries which are performed on the graph, as they are an important influence to the chosen distribution scheme. The queries which are supported can be divided into four categories.

3.3.1 Friend Fetching

One category of queries is the fetching of a list of friend IDs. In fact, two queries are supported in this category:

- Fetch a list of all 1^{st} degree friends of a member.
- Fetch a list of all 2^{nd} degree friends of a member.

The first query is the easiest. It is implemented by looking up the object of the specified member, and returning the list of friend IDs that in the object.

The second query is implemented by locating the object of the specified member, plus all objects of his friends and inserting all the friend IDs in those friend objects in a hash-table-based set, thus eliminating all duplicates. The IDs in this set are then returned as a plain list. Note that this list may also contain first degree friends, but it does not have to.

3.3.2 Pathfinding

Another category is pathfinding. Pathfinding is supported in three flavors:

- Check whether two members are direct friends.
- Find common friends between two members.

- Find paths from one member to another in the third degree network.

The first query again is the easiest. It is simply implemented by looking up the object of one the members and determining, using binary search, whether the ID of the other member is in the friend list of that member.

The second query is still rather easy. The objects of both members are located, and then an intersection is created from the friend lists of both members.

The third query however, is harder. First, the objects of both members are located, plus all friend objects of the first member. The algorithm then performs a depth-first walk through all the friend objects of the first member and the friend IDs in those objects. Every destination of those second degree paths is then looked up in the set of friends of the second member. If the destination is found in the other friend list, a third degree path can be created.

3.3.3 Update Queries

There are also a few queries for performing updates, which are:

- Create a friendship between two members.
- Delete a friendship between two members.
- Create a new member.
- Delete a member.
- Set/unset the *isOnewayMember* flag.

Creating and deleting friendships between members both work about the same way. The objects of both members are located, and the ID of the one member is added to or deleted from the friend list of the other member, and the other way around. Changes to the objects are then replicated, as explained in section 3.4.3.

Creating a new member is also quite easy. A new, empty object is created and registered with the meta-node. The node to store the new object on is determined using a simple modulo calculation on the object ID.

Deleting a member is a bit harder, because all existing friendship relations will be removed before the member object is deleted and unregistered from the meta-node. This triggers updates in all objects which have friend references to the deleted member.

Finally, setting and unsetting the *isOnewayMember* flag is no more work than looking up the object, updating the flag, and replicating the changes.

3.3.4 Friend Suggestions

One special type of query is the so-called friend suggestions query. This query, as its name suggests, is used for offering suggestions to members, about other members, who might likely be friends of them.

In detail, what the query does is construct the second degree friend network, just as with the second degree friends fetch. With one big difference, which is that it also counts how often each second degree friend ID occurs. Finally, when the set is complete, it removes all the first degree friends, and creates a sorted list based on the registered counts. This list is returned as a list of suggestions, with the first member ID being the second degree friend with the highest number of connections to the original member's direct friends.

3.4 Caching the First Degree Network

Having seen how the various queries are implemented in the previous section, it becomes clear that for every query no more objects are required than those of at most 2 members, and the objects of the first degree friends of one of these members.

This means if it is possible for every node to act as a master for a set of objects and also keep in memory copies of all objects in the first degree network of the master objects, then nodes can answer all their queries using at most one additional object from another node.

From this assertion arises the third major design decision: All objects of the first degree friend network of a node's master objects are cached on that node.

As a direct result of this, any node is always able to construct the full second degree friend network of all objects for which it is the master node.

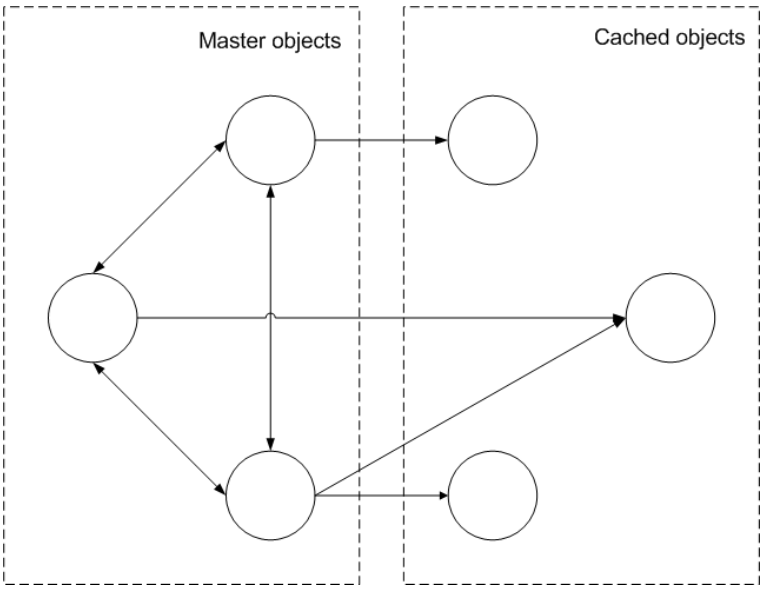

Figure 3.2: Tracking references to first degree friend objects.

Of course, the node may not have all the objects of all second degree friends of all master objects, but it will be guaranteed to know the IDs of those objects.

3.4.1 Reference Counting

In order to determine which objects should be included in the cache, and which objects may be removed from the cache, all objects are reference counted. Reference counting is implemented by counting all references to an object from the master objects on the same node.

Using this mechanism, it is easy to determine which objects should be cached. Any non-master object should be cached if, and only if, it has a reference count higher than 0.

Because only the references from master objects are counted, cycles among the cached objects cannot prevent cached objects from being discarded from the cache when they are no longer referenced by any master objects.

An illustration depicting reference counting is given in figure 3.2.

3.4.2 Directional versus Bidirectional Relations

In order to perform reliable reference counting, it is important that all friend relations are bidirectional.

For example, consider the scenario where objects are redistributed and a node gets assigned an object which was not yet in the cache. At this point it would be an extremely expensive operation to check all master objects and check how many references there are to this particular object. However, if relations are always bidirectional, the same result can be achieved relatively cheap by iterating over the object's friend list and checking for which of those friends the node is registered as the master node.

While this may seem rather trivial, because friendship relations by definition are bidirectional, this is actually a change from the current Hyves implementation, which relies on the usage of directional relations.

In the current Hyves database there is a special group of members called *oneway friendship members*, or just oneway members for short. Relations with these members are always directional (unless two oneway friendship members create a friendship with each other). The reason directional relations are used for these members is that they can have a lot more friends than normal members.

Normal members are limited to having 750 friends at Hyves, while gold-members (which have bought a premium membership) are limited to 1000 friends. Celebrities however, often have many fans who also want to become friends at Hyves. For example, our premier Jan-Peter Balkenende has over 100,000 friends.

Now, with some members having such huge amounts of friends, it is easy to see that certain queries, like friend suggestions, would come to a crawling halt if they had to process such members. Therefore, relations originating from oneway friendship members are left out of the friend graph, so that such queries will automatically ignore them.

In order to both guarantee that all relations are bidirectional and to not let the search space of the queries explode, I introduced the *isOnewayMember* flag that indicates whether a member is a oneway member. Using this flag, queries can ignore such members if desired.

3.4.3 Replication

When an object has been modified by a node, the other nodes that have a cached copy of the object should also receive the update. The nodes that are not master nodes for an object, but which contain a cached copy, are referred to as *slave nodes* for that object.

The replication process is illustrated in figure 3.3. In this illustration Node 1 is the master node of the object for which modifications are being replicated. The other nodes are slaves. As can be seen in the figure, replication messages to slaves are all sent in parallel.

This type of replication of updates does not differ a lot from replication as implemented in Hyves' current master-slave setup. There are two important changes though:

- There no longer is a single master, but every node is a master for a subset of objects.
- Not all slaves receive updates for all replication messages anymore. Instead, nodes act as slaves for a subset of objects.

The first change is not very significant. Any node can simply replicate updates to its master objects to other nodes.

The second change is more difficult however. Before a master node can send replication messages to the slaves, it will first need to find out who the slaves are. But, there is no central registry of which nodes are slaves for which objects.

One option might be to simply replicate to all nodes and to let the nodes which are no slaves of the object simply ignore the message. Using a little trick however, it's possible to avoid these unnecessary messages.

The trick is to send the updated object to the meta-node. The meta-node can then create a list of all the master nodes of the friends of the specified object. Since all relations are bidirectional, the master nodes of an object's friend objects correspond exactly to the slaves of that object. Using this knowledge, the master node can send its replication messages to just the slaves of the object.

Finally, it should be noted that replication messages are always send asynchronously, and therefore do not block the query in which the update is

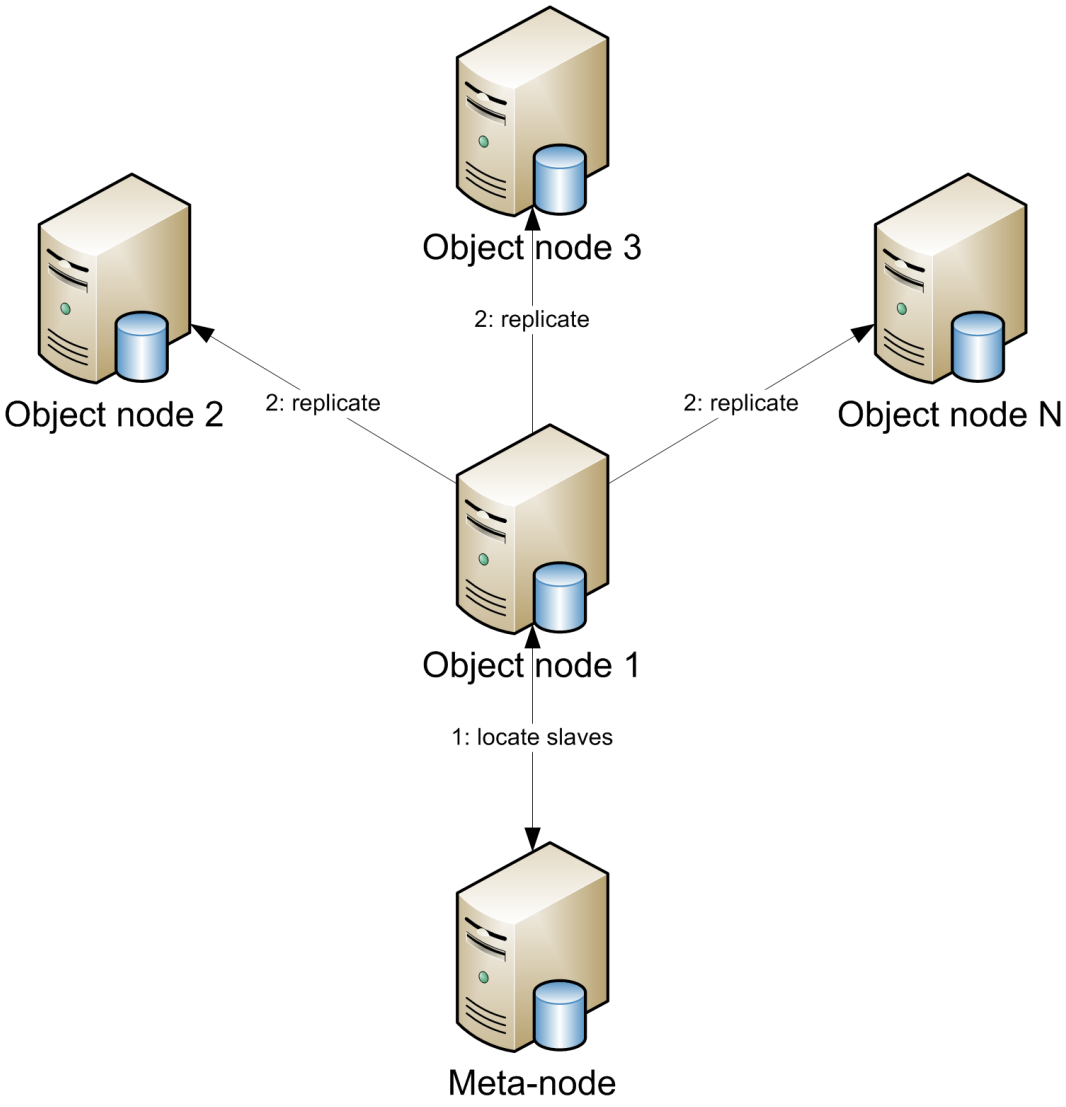

Figure 3.3: Replication overview

performed. This means it is possible for a subsequent query, which is submitted to a different node, to not see the changes performed by the previous query. This fact can lead to results from queries which are not entirely correct and which show outdated information.

However, because the master nodes always do have the latest version of their objects, inaccuracies resulting from replication delays will only occur at the second or third degree friend network, and results which only involve the first degree network will always be accurate. As also stated by LinkedIn [3], inaccuracies outside the first degree friend network will hardly ever be noticed by users. Hyves also considers such inaccuracies to be acceptable.

3.5 Persistence

Whereas the previous sections have dealt extensively with the in-memory data structures, this section will cover how such structures are written to disk in order to preserve data should a crash, or some other failure occur.

Admittedly, because no current scalability bottlenecks exist with the Hyves friend database concerning disk writes or persistence and no such bottlenecks are expected either, this part of the design is described in less detail than the other parts.

3.5.1 Storage Engine

Fortunately, many existing solutions for disk storage exist, and there is no need to re-implement an entire storage engine for this architecture. For this setup I have chosen to use Berkeley DB [9] as the storage engine. Berkeley DB was chosen because it is a proven, fast and well-known database for storing nothing but key-value pairs. As we will see throughout this section, all data that needs to be stored can easily be mapped to such key-value pairs.

3.5.2 Object Node State

The entire state of an object node consists of the objects the node contains, both the master objects and the cached objects. Of these objects, only the master objects are stored on disk.

The reasons for not writing cached objects are threefold. First of all, it would be redundant, as they are already stored on their respective master nodes. Second, it would cost unnecessary disk writes to write all changes to cached objects to disk. The third reason is consistency. While every node is authoritative for its master objects, it is not for the objects it has cached. Therefore, should a node have to reload a cached object, it would be better to ask the respective master node for the object, rather than rely on its own disk storage.

Master objects are saved to disk though, and the in-memory representation and on-disk versions are always kept in sync. This means that updates to master objects will be immediately synchronized to disk before the update is considered complete.

Objects are saved in the Berkeley database using the object ID as key, and a serialized representation of the object data structure as value.

In order for an object node to restore its state after is has been down it will reload all the master objects from the Berkeley DB. All objects referenced from the master objects, but not yet loaded, will be retrieved from other nodes and so the cache is reinitialized. During this process the index will also have been transparently rebuilt, and the node will be ready for operation again.

3.5.3 Meta-Node State

The state of the meta-node consists of the mapping from object IDs to node identifiers (which could be hostnames, IP addresses, or some other method to identify the nodes).

Clearly, this mapping is easily translatable to the Berkeley key-value pairs as well, as object IDs would be used as key and the node identifier as value.

The meta-node, just like the object nodes, keeps its entire data structure in memory, and keeps the on-disk representation in sync upon changes.

Should the meta-node have to recover its state after having been down all it would need to do is load the data structure from the Berkeley DB again.

Chapter 4

Analysis and Validation

In this chapter, I will make an analysis of the scalability of my proposed solution. I will also analyze the communication overhead between server nodes in order to measure the effectiveness of the caching mechanism.

Before I will look at the various properties of the architecture however, I will take a look at how the network has grown in the past. Based on this, combined with current knowledge, I will make a prognosis of the future network growth over the coming 5 years. This is the least amount of time for which the architecture should be sufficiently scalable. This prognosis can be found in the next section.

In section 4.2 I will tell about the prototype I've built for measuring various properties of the proposed architecture.

With the prognosis in mind, and using the prototype I built, I will then analyze the aspects of the proposed architecture most relevant for scalability and performance. I will start by analyzing memory usage in section 4.3. Second, I will analyze the inter-node communication in section 4.4. Finally, I will analyze the actual processing of queries in section 4.5.

4.1 Prognosis

At present, the social network of Hyves consists of 6.7 million members, and 250 million friend relations, but what is the size Hyves should be preparing for? As we can see in appendix B, Hyves should prepare for a linear growth in

the next couple of years, accounting for an increase of 2.75 million members annually, and at most an additional 28.7 friendships per member per year.

If we continue this trend, the Hyves network will have a total of 20.5 million members in just 5 years, each having an average of 216.8 friendships[1]. This would result in a total of about 2.2 billion friendship relations (or 4.4 billion directional relations).

These numbers should be regarded as a minimum scalability requirement for the proposed architecture. If it cannot handle a network of this size, it is probably insufficient. However, scalability should also not stop at this point, and I will also make an attempt at making a prognosis on how much further the current architecture can be scaled. Finally, I will try and propose back-up solutions, should scalability limits be reached.

4.2 Prototype

In order to measure various aspects of the proposed architecture, I've created a simple prototype. This prototype is capable of loading the entire friend graph into memory from a backup copy of the Hyves live database, and perform actual queries on this graph. Unlike a full implementation however, this prototype does not actually use multiple physical nodes, but emulates the use of multiple nodes in memory. For this reason there is no actual communication overhead between nodes, but instead the prototype will register all communication between the nodes through method calls.

Additionally, the prototype does not implement any form of persistence and updates are all performed in memory only.

Using this prototype I have performed various simulations in order to measure how much and the type of communication messages would be sent between nodes, and the amount of memory that would be used per node.

For these simulations a copy of the friend graph was loaded into memory from a snapshot containing just over 5,000,000 members. For some experiments however, the snapshot was capped at 1,000,000 members. In this case, the graph naturally only contained the relations between those first

[1] To be honest, I find it unlikely that members will have an average of 216.8 friends in 5 years time, and expect it to be a lot less. Nevertheless, it's better to be safe than sorry, and I will continue to use this prognosis for the friends ratio. After all, it's the best objective guess we have.

million members.

During simulations, any number of nodes can be emulated, and objects are distributed over those nodes. Distribution is based on a simple modulo of the member ID, which given the law of large numbers [12] results in an even distribution among nodes.

4.2.1 Query Logs

For simulations where measurements were performed on the number of messages passed between nodes during query execution, I replayed a batch of a total of 100,000 queries which were logged from the live website. The queries are executed in the same order as they were performed on the live database, though they only represent a subset of all queries that were executed during the logging period. This is a necessity, as logging of all queries would have flooded the logging servers. Nevertheless, because of the size of the sample, the set of queries is still expected to be statistically representative.

4.2.2 Timing Measurements

In order to be able to do measurements on the performance of queries, I also wrote an algorithm for constructing worst-case graphs of different sizes. By constructing such graphs in incremental sizes and measuring the time needed to complete different queries on those graphs, I was able to measure the time complexity of those queries.

A worst-case graph of size n is constructed as follows: A single member is created as a starting point. Then n friends are created for this member. Finally, new friends for those members are continuously created until all these friends have n friends as well, and this is repeated up to the desired depth of the graph. Cycles in the graph are intentionally avoided within the search space of the queries to make sure all queries will need to process all paths.

Of course, there are some limitations to this approach of time measuring. First of all, a simple workstation machine was used for measurements, and the absolute times recorded are not representative for the more powerful machines at the Hyves server park.

Second, because the entire prototype runs in a single-threaded process, the

timings are only representative for the time needed to do the actual processing of the query, but do not cover latencies for communication overhead. Fortunately, as we will see later on, this overhead is mostly a constant factor.

Finally, we should consider the fact that no concurrency or associated locking mechanisms are implemented in the prototype, which may impact the performance of a real implementation as well. It is believed though that all read queries in the proposed architecture can be implemented without the need for locking.

The prototype was implemented in C++ using the Qt 4.4.0 framework on a Linux platform using GCC 4.2.3, running on an Intel Core2 Duo CPU at 2.4GHz with 4MB cache and 4GB RAM.

4.3 Memory Usage

As explained in section 3.1.3, total memory usage consists of two parts, the memory used for storing objects, and memory used for indices. As we will see, the first is partially scalable using out-scaling, the second does not need any further scaling at all.

4.3.1 Object Memory

The amount of memory required per node for storing objects is directly related to the amount of objects stored on that node, and the size of those objects. There are two types of objects on every node though, master objects and cached objects.

Master objects can easily be distributed evenly over multiple nodes, using a modulo distribution. The amount of memory available for storing master objects can therefore be linearly scaled by adding more server nodes.

The amount of memory required for the cache is a lot harder to assess though. This is because the amount of objects in the first degree network is dependent on the number of master objects, on the number of friends these master objects have and on the ratio in which these master objects share common friends. Finally, for the exact memory consumption it also matters how much friends the objects in the first degree network have.

The main culprit here is the common friends ratio. The reason for this is

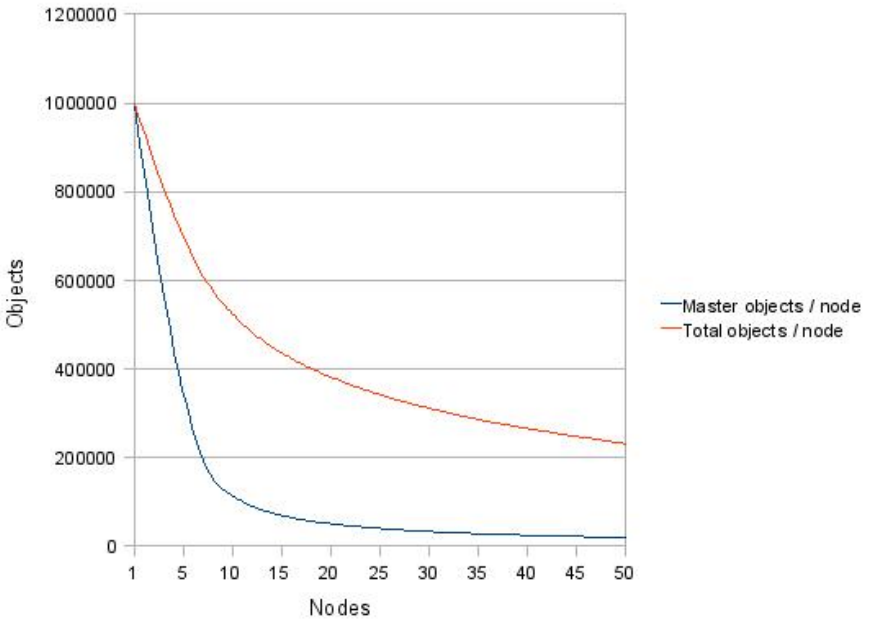

Figure 4.1: Number of objects compared to the number of nodes.

that this ratio is almost impossible to predict, and is highly dependent on the particular partitioning of the objects among the nodes. For this reason I have performed some experiments in the prototype to see how many objects were in the cache under varying circumstances. Because of memory and time constraints, the experiments were performed on a subset of the live friend graph, containing only 1,000,000 members, rather than the just over 5,000,000 members that were present in the snapshot. This subset is still expected to be statistically representative, which is supported by the fact that it shows similar scale-free characteristics.

First of all, I've measured how the amount of objects in the cache reacted to the amount of nodes that were used. The results of which can be found in figure 4.1.

As we can see in the figure, memory usage does not decrease linearly with the number of nodes. This is because as the total number of master objects on a node decreases, the overlap in friends of these objects (the common friends ratio) also decreases. Because of this, the amount of cached objects needed per master object increases.

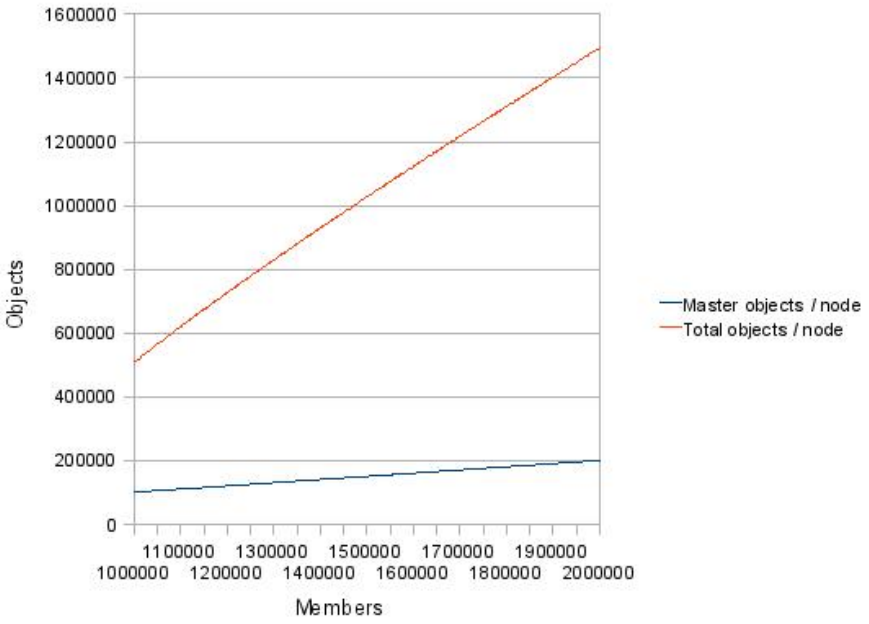

Figure 4.2: Number of objects compared to the number of members.

It should be noted though that the number of cached objects needed for a single master object never exceeds the number of friends of that master object and therefore the average ratio of cached objects per master object will not exceed the average number of friends per members.

Second, I've measured how the amount of objects changes as the number of members increases. For this experiment, more and more members were created, and friendships were created for these members in order to keep the friends per member ratio constant. 10 nodes were used in this experiment. The results can be found in figure 4.2.

In this figure we can see the size of the cache increase almost linearly with the number of master objects as more and more are created. The ratio of increase is not the same though. The number of members doubles, while the total number of objects triples. This already suggests that the growth ratio should slowly decline, as the number of objects per node simply cannot grow larger than the total number of objects in the graph. This is supported by the fact that the figure shows a very slightly flattening curve.

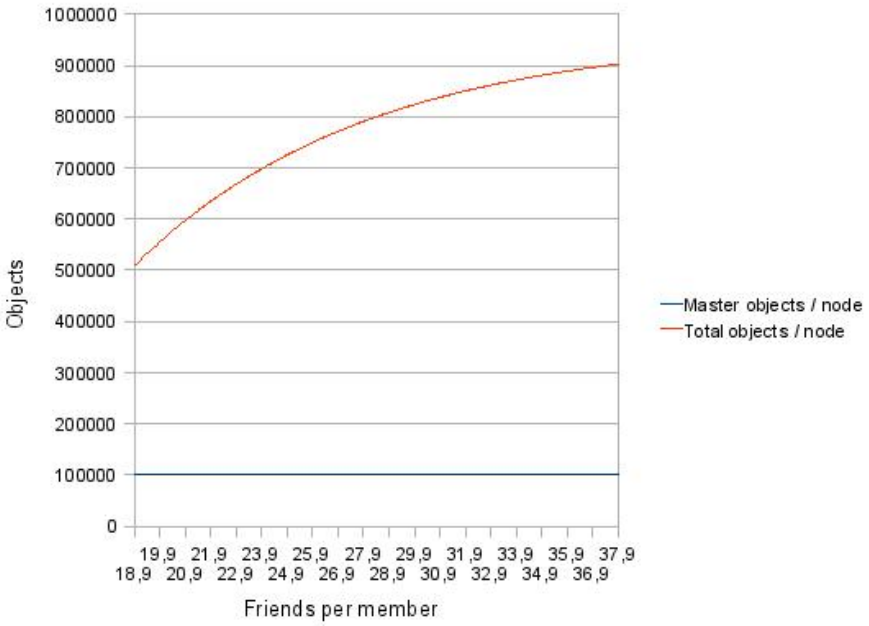

Figure 4.3: Number of objects compared to the number of friends per member.

Third, I've measured how the amount of objects changes as the number of friends per member increases. With the subset of the graphused, members on average had 18,9 friends, and this was increased to 38,9 friends per member during the experiment. Again, 10 nodes were used in this experiment. The experiment was performed by randomly creating additional friendships between members. The results can be found in figure 4.3.

This figure shows us that the total number of objects per node quickly increases as members get more and more friends. This increase flattens however as the number of objects per node approaches the total number of objects in the graph.

4.3.2 Index Memory

The amount of memory required for the index is only dependent on the number of members. Or, to be precise, on the member ID of the newest member. Because of the array implementation of the index, it does not

matter how many nodes there are, how many members are actually located on any given node. Because nothing but the object IDs is being indexed, the number of friends of objects does not have any influence either. All of this implies that the memory requirements for the index cannot be scaled using out-scaling.

Fortunately, the array does not use a lot of memory to begin with, and out-scaling is not expected to be necessary at all. Looking at the expected amount of members in 5 years, 20.5 million, we can calculate the amount of memory needed for this index using the formula given in section 3.1.3. Filling in the formula gives us the following:

$$1.5 * 8 * 20,500,000 = 246,000,000$$

Therefore, 246 million bytes, or almost 235MB, should be reserved for this index. This corresponds to 0.72% of main memory on a server with 32GB RAM.

Even if Hyves would get 10 times as much members as anticipated over the coming 5 years, possibly making them the largest online social network in the world, the memory required for this index would still only take up 7.2% of main memory. I believe it is therefore safe to state that indices will no longer pose any scalability issue in the foreseeable future.

Nevertheless, should all assumptions fail, and should the index still cost too much memory, it is always possible to fall back to a regular hash-based index, at the cost of decreased look-up performance.

4.3.3 Conclusion

As we can see, because of the introduced cache, and to a lesser degree because of the use of a plain array as index, there is actually less memory left for growing the friend network. Additionally, available memory cannot be linearly scaled by adding more nodes, because memory usage per node does not decrease linearly with the added number of nodes.

The good news though is that overall memory requirements have already been reduced considerably. Using the formula from section 3.1.3, even if members on average have 216.8 friends, a single node would be able to

keep a friend graph in memory containing at least 35 million members[2]. A single node therefore can already keep in memory a graph 70% larger than predicted by our prognosis. Given that the graph will not be stored on a single node, but on multiple nodes, the total graph size will be allowed to be even larger.

Furthermore, there is an important side note related to the experiments above. In all the experiments where network growth was simulated, new friendship relations were created randomly. This is not realistic, as we have already found that the network does not have random connectivity, but rather looks like a scale-free network. Randomly creating new friendship relations therefore is not very realistic for natural growth. The upside here is that creating new relations in a scale-free manner (which would have been very hard to simulate truly realistically) would have resulted in a higher common friend ratio, because it would have increased the chances of members becoming friends with people they know indirectly. A higher common friend ratio would have resulted in less objects being cached per node. Therefore the graphs that were presented are most likely more pessimistic than what we would see with real network growth.

Thus as far as we can see, even with the caching, the solution appears sufficiently scalable when it comes to memory requirements for the next 5 years, and probably beyond.

Still, it should be considered what to do if the graph grows faster than anticipated anyway. For one thing, it's probably a safe bet that in 5 years time, commodity servers will be available with 64GB of RAM, or even more. At this point, we would be able to grow the graph at least twice as big again using up-scaling.

Finally, if all else fails, it can still be decided to (partially) disable the cache, at the expense of an increased network load.

4.4 Inter-Node Communication

In this section, I will analyze the communication between nodes.

Basically, there are two areas where communication between nodes is nec-

[2]For this calculation, I make the assumption that a node containing 32GB of RAM, will have 30GB at its disposal just for the friend graph (including index).

essary. The first area is the query pre-processing, where the correct node for processing a query is located and the necessary data is gathered. This is handled in detail in section 4.4.1. The second area is the replication of updates to objects, which is handled in section 4.4.2.

Figure 4.4 gives a detailed overview of the interaction between nodes during the entire process of handling an update query. Note that handling of a read query is exactly the same, minus the replication.

4.4.1 Query Pre-Processing

Before any query can be processed there are a few things that need to be done. First of all, the correct node that will process the query needs to be found. This is achieved by performing a query on the meta-node, which I will simply call a *meta-query*. This is step 1 in figure 4.4.

The purpose of the meta-query is to locate the master node of the object specified by the query[3]. The only exception is the query for creating a new object, in which case there is not yet a defined master node, and the meta-node can return any node. In my simulations, the meta-node will determine the node for a new object based on the modulo of the ID of the new object, but other methods may be used as well.

Once the correct node has been located, the query is submitted to this node. This is step 2 in the figure. For some queries however – the pathfinding queries and some of the update queries – a second object is required which may not be on the same node. If this is the case, the node will submit a meta-query for this object as well (step 3), and fetch this object from the other node (step 4).

At this point, all necessary data is available on the node that will be processing the query.

What makes this interesting is that whatever query we are dealing with, these steps are always the same (of course for some queries the additional object will be required, and for others it won't). This means that whatever query we are dealing with, and no matter how many nodes our setup contains, there are always either 2 or 4 queries between nodes. One meta-query from the client to the meta-node and the actual query from the client to one of the object nodes. And optionally one more meta-query from an internal

[3]Some queries specify two objects, in which case either of those can be used.

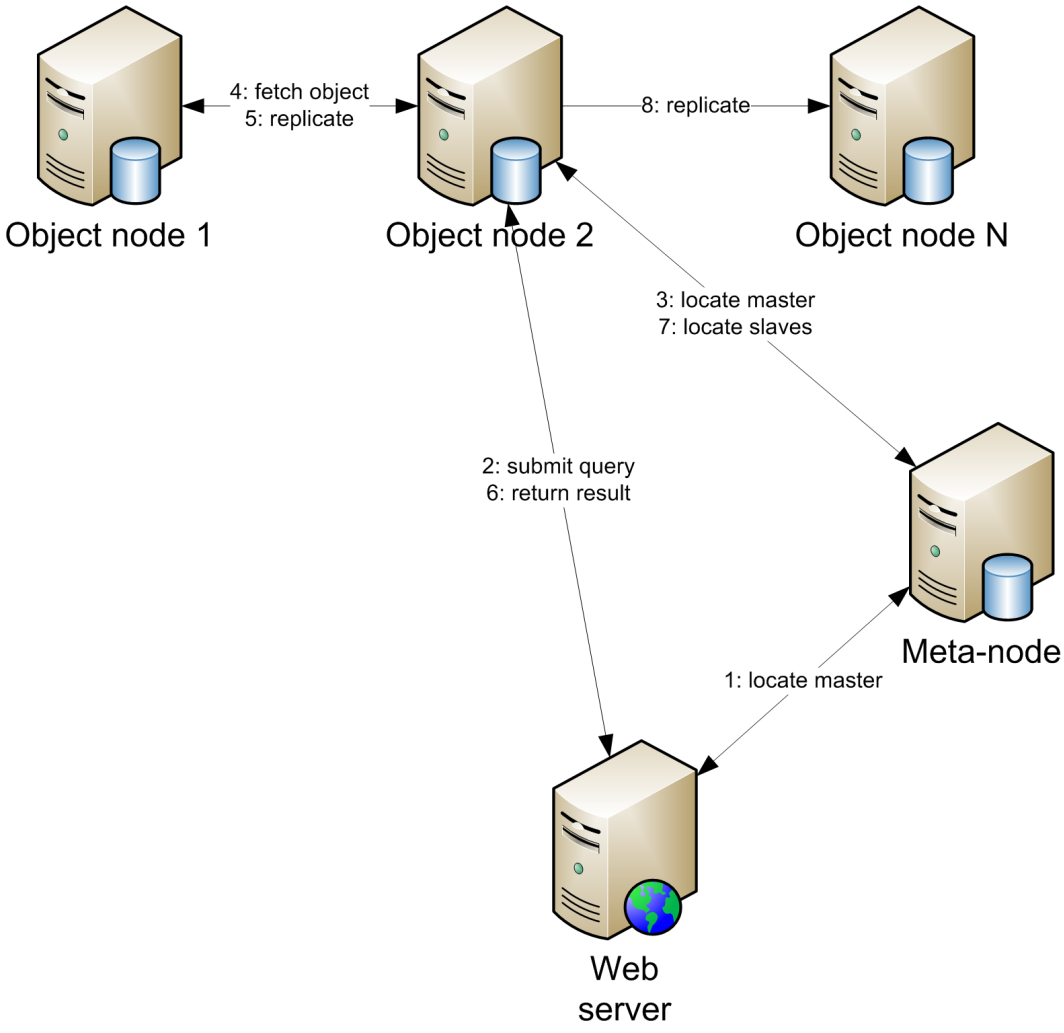

Figure 4.4: Client performing an update query.

node to the meta-node and one object fetch between two internal nodes. While it may occur more often that 4 queries are required rather than 2 as the number of nodes increases, no matter how far the setup is scaled out, more than 4 queries will never be required.

4.4.2 Update Replication

Finally, after the query has been processed and the result returned to the client, any possible changes made to objects should be replicated to other nodes containing the object. This step therefore only applies to update queries.

The first type of replication applies to the creation and deletion of relations. Because these queries always perform updates on two objects, it is possible for the node which is processing the query to only be the master node of one of these objects. In this case, the change to the other object should be immediately committed to the master node of the other object as well. This replication step, step 5 in figure 4.4, requires a two-phase commit protocol [11] to make sure one node does not commit the transaction, while the other fails. Only when both nodes confirm a successful commit of the transaction, will the query return successfully to the client.

Looking at the first replication step, we can conclude that this again is a constant factor, which requires a total of 4 messages between the nodes because of the two-phase commit (or no messages at all if the node happens to be the master of both objects).

The second type of replication applies to all updates to objects, and is performed after the result of the query has been returned to the client. This step concerns replication of updates to all the slave nodes of an object. To do this, first it should be determined which nodes are the slaves of the object. This is step 7 in the figure and the mechanism to determine the slaves nodes is explained in section 3.4.3. When the slaves are known, they are all sent a message with the update.

If we take a close look at this second step, we can see that as the number of nodes increases, the number of slaves for objects will likely increase as well. Therefore, the number of replication messages that have to be sent will increase as well.

To analyze the impact of this I have monitored the amount of replication

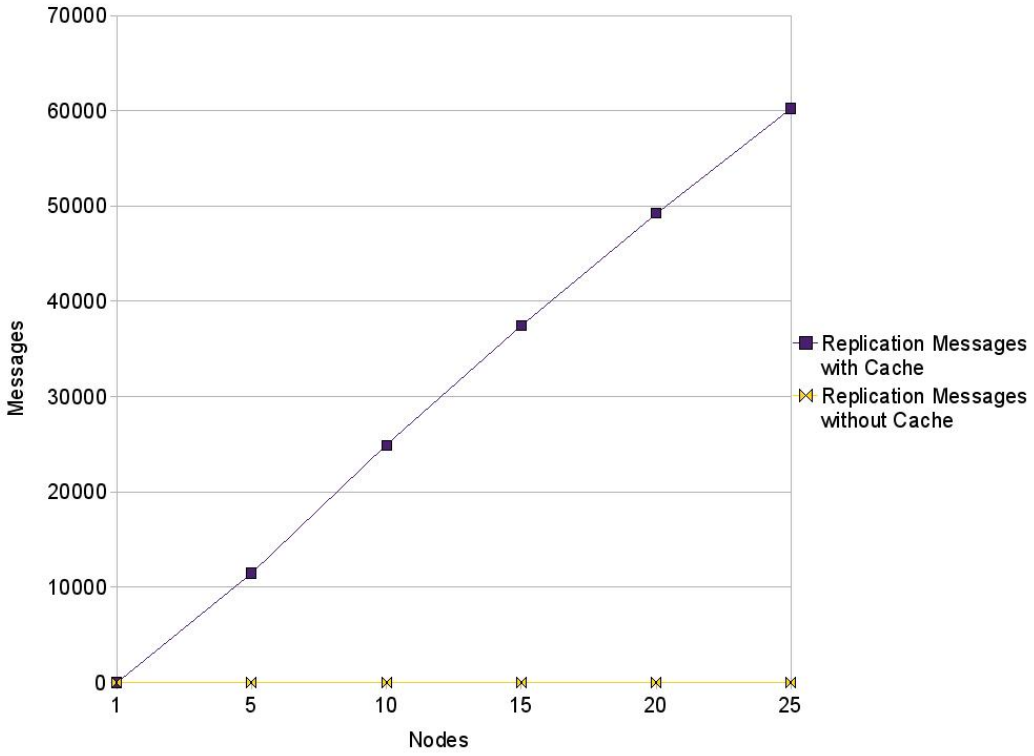

Figure 4.5: Increase of replication messages as more nodes are added.

messages while a query log consisting of 100,000 queries was executed on the backup copy of the Hyves friend graph in the prototype. This experiment was performed with an increasing number of nodes, so as to measure the increase in replication messages as the number of nodes increases. The results of this can be found in figure 4.5.

To put this into perspective, figure 4.6 contains a graph of the overall increase in messages (including meta-queries and object fetches for the query pre-processing).

Additionally, I have split up the different types of messages, and a detailed overview of the increase of various types of messages can be found in tables 4.1 and 4.2. The first table shows the increase in messages when the cache is used, whereas the second table is for comparison with the situation without cache. Note that the last row in both tables includes the 200,000 queries

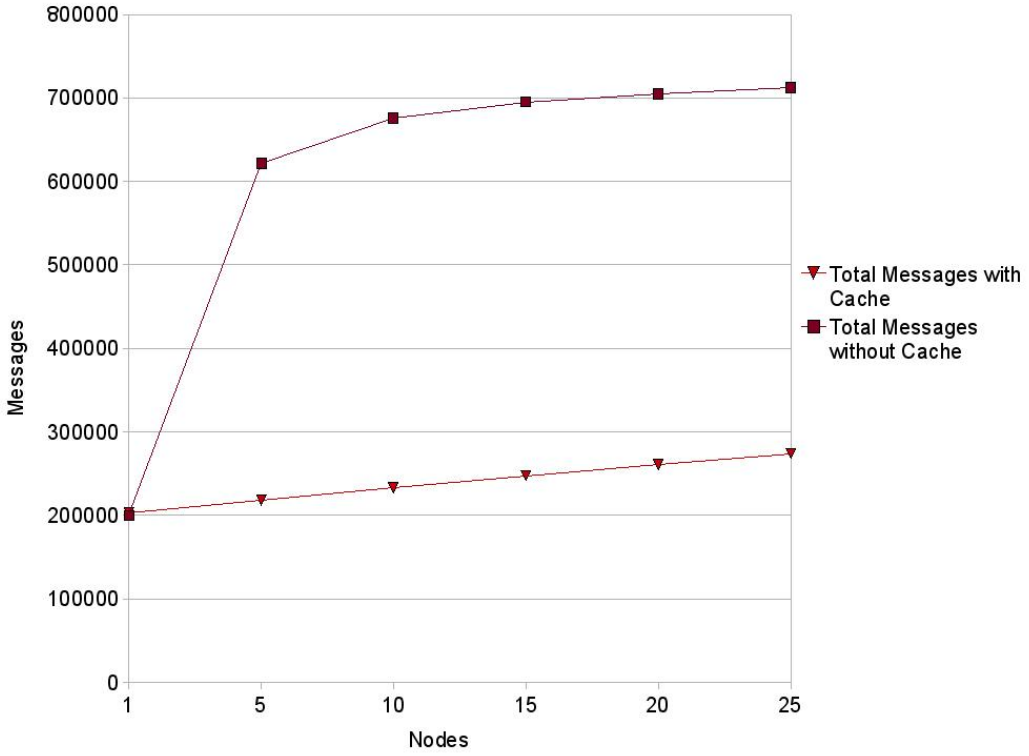

Figure 4.6: Total increase of messages as more nodes are added.

Number of Nodes:	**1**	**5**	**10**	**15**	**20**	**25**
Object Fetches	0	1,895	2,765	3,601	4,410	5,366
Meta-Queries	3,028	4,923	5,793	6,629	7,438	8,390
Replication Messages	0	11,460	24,883	37,444	49,235	60,235
Total Internal Messages	3,028	18,278	33,441	47,674	61,083	73,991
Total Messages	203,028	218,278	233,441	247,674	261,083	273,991

Table 4.1: Messages passed under varying number of nodes with caching.

Number of Nodes:	**1**	**5**	**10**	**15**	**20**	**25**
Object Fetches	0	210,884	237,989	247,543	252,408	256,199
Meta-Queries	0	210,884	237,989	247,543	252,408	256,199
Replication Messages	0	0	0	0	0	0
Total Internal Messages	0	421,768	475,978	495,086	504,816	512,398
Total Messages	200,000	621,768	675,978	695,086	704,816	712,398

Table 4.2: Messages passed under varying number of nodes without caching.

and meta-queries from the client as well.

As we can see, overhead for replication messages increases a little less than linearly with the number of nodes. This increase looks indeed to be caused by the increasing number of slaves that needs to be replicated to. Given that every member has a limited amount of friends, the maximum number of slaves per object is limited as well. Therefore it is expected this graph would flatten more as more nodes are added.

There is one more observation we should make however. As said, there is a constant amount of meta-queries for figuring out who the slaves are. There is one "flaw" in this statement though, which is that it is based on the assumption that there is only one meta-node. In section 3.2.1 it is explained however that it may be necessary to use multiple meta-nodes if a single one cannot take the load. In this case, it may be necessary to query all these meta-nodes in order to determine all slaves of an object. Nevertheless, I believe it is safe to assume there will always be less meta-nodes than other nodes, therefore the overhead caused by using multiple meta-nodes will always still be less than the replication overhead.

4.4.3 Conclusion

As explained in this section, no matter how many nodes there are, there is a constant upper limit to the amount of messages required for any single query. The only exception is replication which increases even less than linearly as the number of nodes increases.

Furthermore, as can be clearly seen in the figures listed above, the total number of messages between nodes is decreased considerably by using a cache. As Hyves has already had past experiences with huge amounts of network messages causing trouble due to overloaded switches, this is a significant gain.

Another thing we can see by looking at the tables, is that most of the overhead when a cache is used is caused by the replication messages. This is actually another advantage for the setup with cache. When caching is not used, all object fetches have to be completed in the query pre-processing stage. With the majority of replication messages however, there is no query waiting for the replication to be completed, and an increase in replication messages does not directly increase query latencies.

For these reasons, I conclude that the introduced cache is an effective mechanism for reducing inter-node communication. Additionally, further outscaling of the setup appears to cause only a little, and less than linear, increase in internal communication. Therefore, when it comes to communication, this architecture appears sufficiently, and well scalable.

4.5 Query Processing

Between the gathering of data and returning a query's result is the actual processing of the query. This processing needed to get the query result is analyzed in this section.

4.5.1 Friend Fetching

The first type of query I explained in section 3.3 was friend fetching. In the first degree, this is nothing more than looking up an object and returning the friends listed in the object. Here, the look-up is a constant time operation, whereas the actual time required for serializing and returning the friend list

is linearly dependent on the size of the friend list. The time complexity thus is $O(n)$, where n is the maximum amount of friends in the worst case.

Fetching friends in the second degree network is already a bit trickier. The potential size of the entire size of the second degree network is n^2, but duplicates have to be removed as well. Because a hash-table-based set is used for storing the network however, checking for duplicates can be performed in amortized constant time[13]. The worst-case time complexity for processing this query therefore is still $O(n^2)$.

In general, we can state that the complexity of friend fetching as implemented is $O(n^d)$ in the worst case, where n is the amount of friends per member and d is the degree of the network being fetched.

To validate this statement, I have measured the time complexity of both first and second degree friend fetching in my prototype. The results can be found in figures 4.7 and 4.8. Along with the measured results, a reference line is plotted which shows the respective expected complexity formulas, corrected with a constant c to fit the measurements. As you can see, the worst-case time complexities are indeed $O(n)$ and $O(n^2)$, respectively.

4.5.2 Pathfinding

Pathfinding is implemented up to the third degree. Analysis of the implementations of each degree follow.

As explained in section 3.3.2, first degree pathfinding is trivial. It involves nothing more than a binary search in the friend list of one of the members for the ID of the other member. The time complexity of this query is therefore determined by the binary search, which is $O(\log_2 n)$ [8]. Again n is the maximum allowed number of friends per member in the worst case.

For second degree pathfinding, an intersection between two friend lists has to be created. Using a hash-based implementation, this can be performed in $O(n)$ time.

Third degree pathfinding again works pretty much the same as second degree pathfinding, except that not a single intersection has to be performed, but an intersection has to be calculated for all n friends of one of the members. Therefore, worst-case time complexity becomes $O(n^2)$.

With the exception of first degree pathfinding, which can be considered a

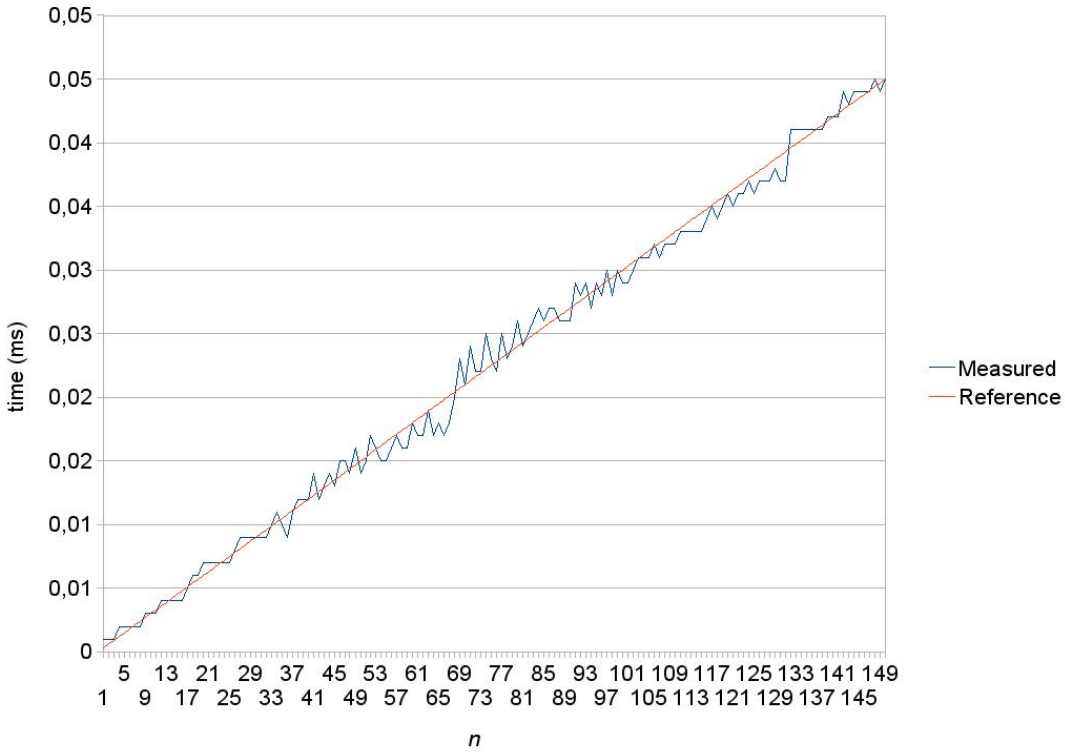

Figure 4.7: Time complexity measurements of 1st degree friend fetching.

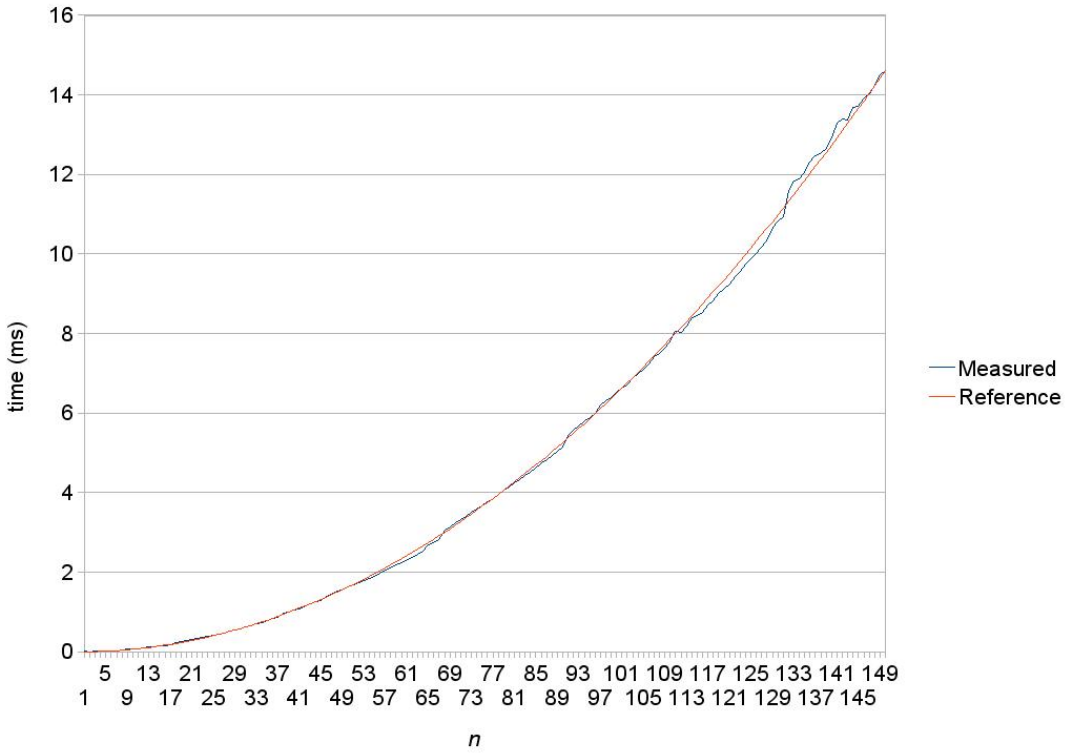

Figure 4.8: Time complexity measurements of 2nd degree friend fetching.

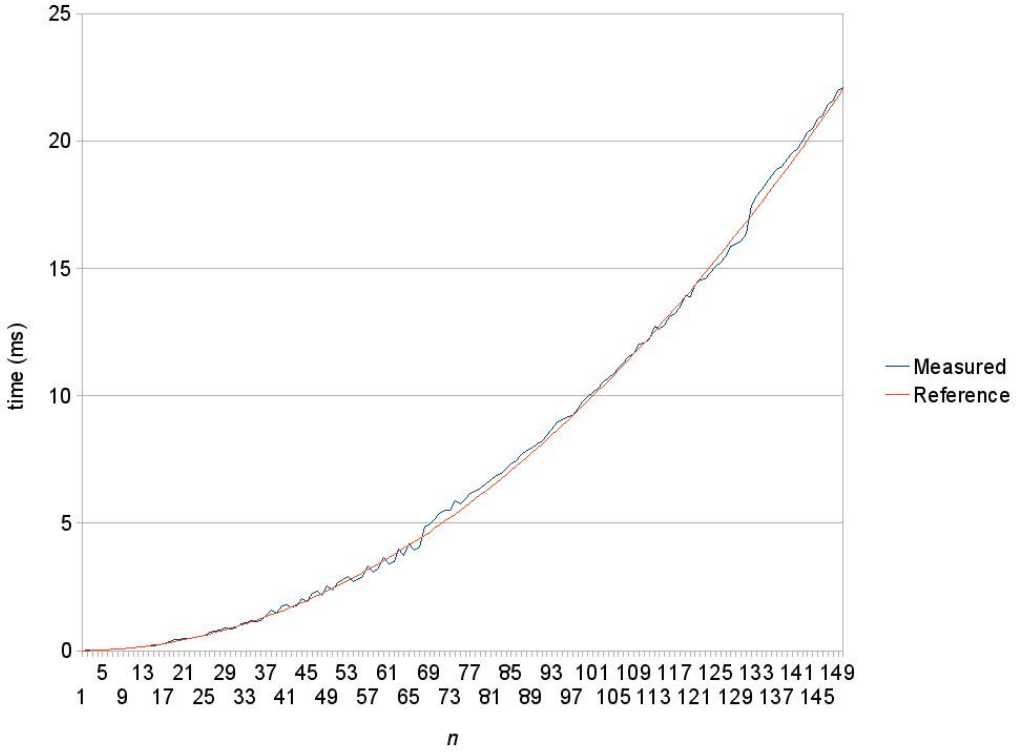

Figure 4.9: Time complexity measurements of 3rd degree pathfinding.

special case, we can say time complexity of this implementation of pathfinding is $O(n^{d-1})$.

In order to validate the worst-time complexity, I have now measured the worst-case time complexity for pathfinding in the third degree. The results of these measurements can be found in figure 4.9. Again, a reference line has been plotted for comparison. As you can see, the figure validates the stated worst-case time complexity for third degree pathfinding.

4.5.3 Update Queries

Four different update queries exist, each of which I will analyze now.

First of all, we have the creation of a relation. In order to create a relation,

the member IDs of two members should be added to each other's friend lists. Insertion in a sorted list is a two-step process: finding the position for insertion, and the actual insertion, which includes moving all items in the list after the insert position. The first has a time complexity of $O(log_2 n)$, the second $O(n)$. Of course, both have to be performed twice, but this has no relevance for the complexity. Therefore the final worst-case time complexity of adding a relation in this setup is $O(n + log_2 n)$.

Now that we know how creating relations work, removing is just as easy. The only difference being that rather than an ID being added, one is removed. The complexity is the same.

Creating a new member is even easier. A new object is created, and registered. A constant operation, therefore the time complexity is $O(1)$.

Deleting a member is the toughest though. When a member is deleted, all his relations should be removed first. Unfortunately, this is not as easy as just dropping the friends list in the member's object, because all the objects of the member's friends should be updated as well. Removing the relations to this member from all his friends results in a worst-case time complexity of $O(n^2 + nlog_2 n)$. Once this is done, the actual removal of the object is a constant time operation though.

4.5.4 Friend Suggestions

Finally, there's the friend suggestions query. This query first constructs the second degree friend network, meanwhile counting the number of occurrences of each member. The complexity of this is $O(n^2)$, just as with a regular second degree friends fetch. Once this network is constructed though, all first degree friends need to be removed from the set and members need to be sorted in order of number of occurrences. Removal can be accomplished in $O(n)$ time, because of the hash that is being used. Sorting is implemented using quicksort, which has an average time complexity of $\Theta(n \log_2 n)$, but a worst case of $O(n^2)$. Therefore, the overall worst-case time complexity is $O(n^2 + n)$.

Unfortunately, it was nearly impossible to generate friend graphs that would consistently trigger completely worst-case behavior for this query. This is because of the quick-sort whose input is extracted from the set containing second degree friends in undefined order. Nevertheless, if we look at figure 4.10, we again see the plotted reference line match pretty closely with the

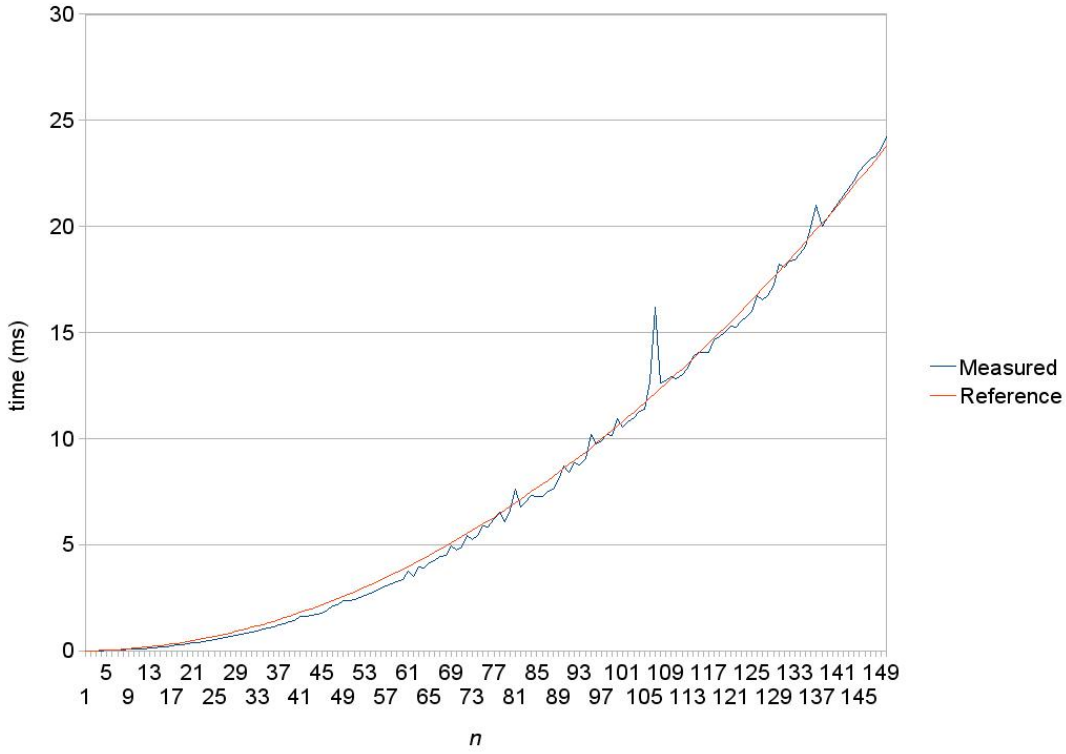

Figure 4.10: Time complexity measurements of friend suggestions.

measured values. Therefore, I believe it's safe to state that worst-case time complexity for friend suggestions is indeed at least close to the order of $O(n^2 + n)$.

4.5.5 Conclusion

In this section I analyzed the worst-case time complexity of various queries which are performed on the Hyves friend database. The time complexity of these queries is actually not influenced by the amount of nodes in the setup, as all actual processing is performed on a single node. Instead, as we have seen, complexity is directly related to characteristics of the network graph on which the queries are performed.

We have seen that the determining factor for a query's worst-case time com-

plexity is the number of friends per member. We have also seen the heaviest queries decrease in performance quadratically to the amount of friends per member, with the friend suggestions query being even slightly worse. For this reason, I have to conclude that while the architecture is rather scalable when it comes to increasing amounts of members or increasing load, Hyves should be careful when it comes to scaling up the number of friends per member. Therefore I would advise to keep limiting the maximum amount of friends per member as is currently the case as well.

If we were to follow figure 4.10, we can estimate that the absolute worst-case for a friend suggestions query on the Hyves live database, where the number of friends per member is limited to 1000, would cost about 1.1s to process[4]. While this is definitely very slow for a single query, it still is believed to be acceptable for an absolute worst case which is highly unlikely to occur in the real friend graph.

Besides worst-case time complexity it's also very interesting to know something about the average cases. We know that members currently have about 73 friends on average. Therefore the easiest way to apply our knowledge about the worst-case time complexity to the average case is to just look at the graphs for worst-case performance and look at the cases where $n = 73$. This will naturally still be a bit pessimistic because real-world second degree friend networks will generally have fewer than 73^2 members due to common friends, but it will still be pretty accurate nevertheless. This would lead us to believe that friend suggestion queries on average can be processed in about 5ms (on the workstation used). This would double to about 10ms if the average number of friends per member were to rise to 100, and so on.

[4]Oneway members are not taken into consideration here, because such heavy queries like third degree pathfinding and friend suggestions are simply disabled for these members.

Chapter 5

Conclusion and Future Work

In this thesis, I have presented a new architecture to replace the current Hyves friend database. The architecture combines advantages of an object-based memory structure, together with sharding and a unique caching mechanism.

These factors combined make for a database that has a low and well scalable communication overhead, and which uses considerably less memory than the current Hyves setup. Measurements indicate that the proposed architecture will be more than sufficiently scalable in the foreseeable future.

Both important bottlenecks with the current Hyves setup, memory usage and the single master, have been addressed.

All in all, I believe the proposed solution can successfully replace the existing solution and is sufficiently capable of resolving the bottlenecks expected by Hyves. Furthermore, I believe that due to the low amount of communication overhead, this solution will be able to provide excellent query performance.

5.1 Partitioning Algorithms

In the future, it would be very interesting to investigate to what extent performance can be further improved using graph partitioning algorithms. I have performed some quick tests, and preliminary results indicate that memory usage required for caching can easily be reduced by 20 to 40% by using a rather simple partitioning algorithm. Furthermore, reducing the size

of the caches also reduces the number of replication messages required to keep the caches up-to-date.

Sophisticated graph partitioning algorithms are still rather complex though, and require a lot of resources to perform the partitioning. Further research would be needed to determine whether the savings in memory and the reduced number of replication messages are worth the expensive of applying extensive graph partitioning.

5.2 Parallelism

Another area to explore is that while the current implementations of the queries are all fully serial, there are many places where parallelization may be applied. For example, pathfinding can be easily parallelized by splitting the space in which to search for paths into chunks and processing these in parallel.

Such parallelizations can easily be implemented using multi-threading on a single-node, after all, that node already contains all the required data. However, if a single query would become too heavy for a single node to process in a reasonable amount of time, it should be considered to split the actual query processing over multiple nodes as well.

Bibliography

[1] Barabási, A.-L., and Bonabeau, E. *Scale-Free Networks*. Scientific American, May 2003.

[2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data, 2006. http://labs.google.com/papers/bigtable.html.

[3] Dellamaggiore, N., and Smith, E. Linkedin: A professional social network built with java™technologies and agile practices. http://hurvitz.org/blog/2008/06/linkedin-architecture.

[4] Garg, A. Hi5: Scaling to 70 million users. http://www.viddler.com/explore/allfacebook/videos/2.

[5] Hoff, T. An unorthodox approach to database design: The coming of a shard, 2007. http://highscalability.com/unorthodox-approach-database-design-coming-shard.

[6] Holzrichter, M., and Oliveira, S. New graph partitioning algorithms, 1998.

[7] Kim, W. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[8] Knuth, D. *The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition*. Addison-Wesley, 1997.

[9] Olson, M. A., Bostic, K., and Seltzer, M. *Berkeley DB*, 1999.

[10] Preiss, B. R. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, 1999.

[11] Skeen, D., and Stonebraker, M. A formal model of crash recovery in a distributed system, 1983.

[12] Tijms, H. *Understanding Probability: Chance Rules in Everyday Life*. Cambridge University Press, 2007.

[13] Trolltech. *Qt 4.4: Generic Containers - Algorithmic Complexity*, 2008. http://doc.trolltech.com/4.4/containers.html#algorithmic-complexity.

[14] Wikipedia. Social network — wikipedia, the free encyclopedia, 2008. http://en.wikipedia.org/w/index.php?title=Social_network&oldid=221023433 [Online; accessed June 23, 2008].

Appendix A

Relational versus Object Based Graph Representation

There are multiple ways to store the adjacency list representation of a graph, two of which are a relational representation and an object based representation:

- In a relational representation, all edges of the graph are stored as rows in a table containing two columns. The first column in this table then identifies the originating vertex and the second column identifies the target vertex. Vertices are not explicitly represented, and a vertex with no edges from or to it is simply never considered to be a part of the graph. A big advantage of a relational representation is that many relational database products are available which use this model. Additionally, indices can easily be applied to columns in tables. By applying an index to the second column, it also becomes fast to look up edges in the reverse direction.
- In the object based representation I selected, every vertex is represented by an object, which contains a list of all outgoing edges. This representation is more compact, because all outgoing edges only need to identify the target vertex, as the originating vertex is represented by the object itself. While objects can exist with no edges from or to it, these objects serve little purpose and may be omitted. Another advantage of this approach is that objects can easily be extended to contain additional information besides just edges.

The choice for a relational or object based graph representation also influences the performance characteristics and memory usage for various operations that are performed on the graph. To make a fair comparison I make the following assumptions:

- When the table based representation is used, the table has two indices both spanning both columns using a B+ Tree. The first index starts with the first column, the second index starts with the second column.
- When the object based representation is used, all objects are referenced from a static array, which is possible because of the relatively small amount of objects. The relations stored in the objects are stored in a sorted dynamic array.
- I use the current graph size of the Hyves friend network for calculating memory sizes. Vertices are identified by member IDs, which are stored in 4 byte integers.

Table A.1 provides a quick overview.

We can see here that because I do no longer need to have two indices on relations, one for each direction, but just a single index on objects only, memory usage decreases considerably. What's more interesting is that memory usage decreases without sacrificing performance. All objects can be accessed using very fast array references.

Additionally, while binary search has a worse time complexity than a B+ Tree index look-up, the binary searches are always performed on small controlled datasets which (except for oneway friend members) are never more than a few kilobytes in size. Because of this, binary searches are arguably faster on these small datasets because they have less memory overhead and therefore cause less CPU cache misses.

Property	**Relational**	**Object Based**
Data size	4GB	2.2GB
Index size	12GB	100MB
Steps required for fetching a single relation	Look-up in B+ Tree spanning 420M rows.	Array reference, plus binary search through the found member's friends.
Steps required for fetching all of a member's relations	Partial look-up in B+ Tree spanning 420M rows, plus iterating and combining the found results.	Array reference.
Steps required for adding or deleting a relation	Look-up in B+ Tree spanning 420M rows, plus adding or deleting the relation at the right position, plus updating the two indices.	Array reference, plus binary search through the found member's friends, plus inserting or removing the relation at the right position.

Table A.1: Comparison between relational and object based representations.

Appendix B

Network Growth

Since the inception of Hyves, the network has grown, both in the amount of members, as well as the amount of friendships. The trend in growth in members is well visible in figure B.1.

Putting some numbers to the graph, Hyves had 2.75 million members at the beginning of 2007, and 5.5 million members at the beginning of 2008, which is almost exactly twice as much. Looking closely though, we can see that this exponential trend does not continue into 2008. In June 2008, there were a total 6.7 million hyvers, and the number continues to grow at a linear pace. This most likely attributed to the fact that Hyves is still mostly oriented towards the Netherlands where the market begins to saturate, while uptake in other countries is much slower. For now, it is safe to assume growth will continue at a linear pace, therefore a growth of 2.75 million members per year is reasonable.

The number of friendships however, has grown in a much higher rate than the number of members, causing the average number of friends per member to increase. Unfortunately, the recorded data that monitors this trend is not fully complete, but some data is available which can be seen in figure B.2.

The graph starts with an average of 32.7 friends per member in February 2007, and ends with 73.3 friends per member at June 2008. Per year, this comes down to an increase of 28.7 friends per member.

Looking at the graph, there actually isn't much to see except for a rather steady, mostly linear increase in the friends/member ratio. To be fair, I don't

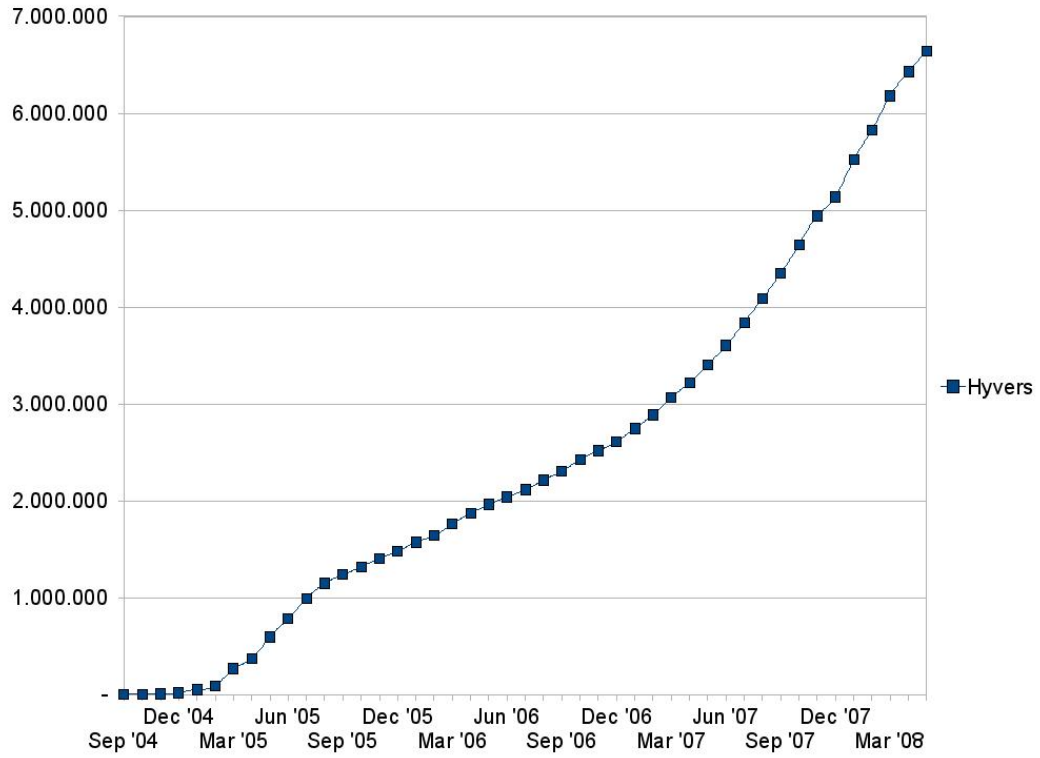

Figure B.1: Growing number of members.

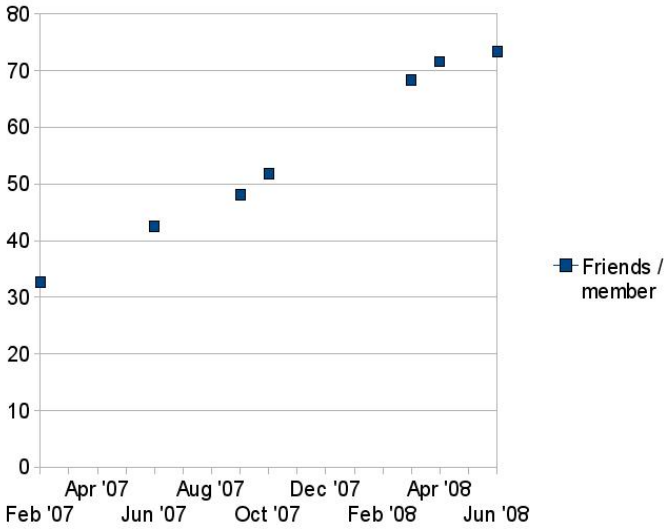

Figure B.2: Growing number of friends per member.

quite know how this trend will continue, but of course a further increase is expected. At the same time, I also believe this can't go on forever, and it's likely there is some natural upper limit. Therefore, if I have to make any prognosis on future development, it will be that the ratio will likely further increase, but it will at most be at the current linear pace, and will likely flatten more at some point in time.