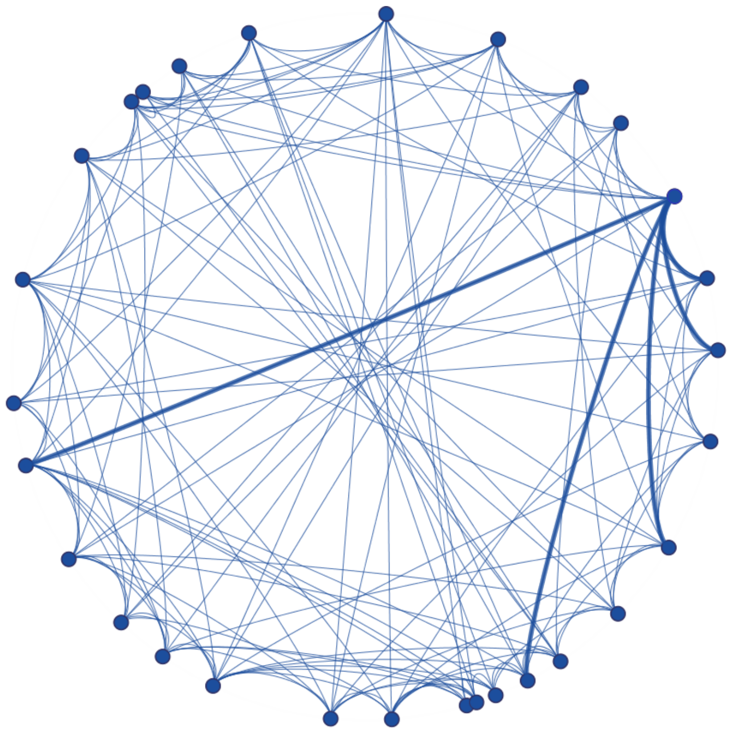
CHALMERS | UNIVERSITY OF GOTHENBURG



A Distributed Key-Value Store

Implementation and evaluation of two strategies for lookup in distributed databases

Bachelor of Science Thesis

OLLE BERGKVIST
PETER ERIKSSON
JULIA JARMAR
KRISTIAN SÄLLBERG
JOEL ÖDLUND

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden June 2013

A Distributed Key-Value Store
Implementation and evaluation of two strategies for lookup in distributed databases

OLLE BERGKVIST
PETER ERIKSSON
JULIA JARMAR
KRISTIAN SÄLLBERG
JOEL ÖDLUND

Examiner: Sven-Arne Andréasson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: Visualization of a distributed system, where nodes are organized and interconnected according to the Chord algorithm.

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

This report describes the design, implementation and testing of a database; more specifically, a distributed, scalable key-value store able to handle many concurrent queries. A central issue when building this type of system is how to search for a specific node in the distributed data space. Two different algorithms, solving this issue in different ways, have been implemented and compared against each other through benchmark testing. One of these algorithms is the Chord algorithm described by Stoica et al in 2001, the other an altered version which is referred to as FCN (Fully Connected Network) throughout this report. While they build upon the same general idea of how to distribute data over the available space, they differ greatly in the amount of connections between nodes in the network.

The report also describes the testing environment used to test these algorithms against each other. Both the database and the testing environment were implemented in Erlang.

Within the confines of the tests, the database showed good signs of scalability and concurrent query handling. The tests also showed great differences in query-per-second performance for the two algorithms, with FCN having a much higher throughput. Chord, on the other hand, adapted more quickly to changes in large networks. This was not measured empirically but concluded from simulation and analysis.

It became clear that FCN was more well suited than Chord in an application where quick lookups are treasured and the network is relatively stable.

Acknowledgements

We want to thank our supervisor K.V.S Prasad for his inspiration and guidance throughout the project, and for his encouragement in times of adversity. We also want to thank Arne Linde for his general support. Chalmers University of Technology provided us with tools necessary to conduct this project.

Glossary

Word	Description
ACID	Atomicity, Consistency, Isolation, Durability. Basic properties that are expected to hold in a traditional database.
CAN	Content Addressable Network, a decentralized infrastructure providing lookup service.
Cache	Storing data in short term memory to get low latency for frequently requested data.
Chord	An algorithm for distribution of a hash-table.
ETS	An Erlang data structure providing constant time access to data.
FCN	Fully Connected Network. Our name for a variant of Chord where all nodes keeps references to all other nodes.
Hash-table	A data structure where elements of data are stored along with, and accessed by a key.
Java	Widely used general-purpose programming language.
Key-value store	Simple database, similar to a hash table.
LAN	Local Area Network.
Latency	How long it takes for data to travel between two nodes.
OTP	Open Telecom Platform.
Pastry	A decentralized infrastructure providing lookup service.
Processing	A graphics library expressed in Java.
Read/Write	The operation of storing or fetching from the memory of the computer a single time.
SQL	Structured Query Language.
WAN	Wide Area Network.

Contents

1 Introduction

Traditional relational databases, generally known as SQL databases, while good at storing large amounts of data, tend to experience problems when exposed to massive amounts of concurrent reads and writes.

NoSQL (Not Only SQL) is a class of database systems that aims to address the issues of high concurrent data volumes through the use of distribution and simplified data models. Major information technology companies such as Facebook, Amazon and LiveJournal have been driving forces in the development of this new set of databases. In 2008, Facebook developed a decentralized storage system named Cassandra [1] to deal with steadily increasing data throughput. One of their problems was real-time messaging (large amounts of data simultaneously travelling to and from the databases). This is also known as the Inbox Search problem, where the write throughput is billions of writes per day and is constantly growing. Amazon, facing the challenge of storing petabytes of data, created Dynamo [2] to be able to leverage the power of commodity hardware rather than having to invest in supercomputers.

The advantages of NoSQL databases have made them increasingly popular among a wide variety of applications. Their use is however not without its drawbacks. A major compromise of using a NoSQL approach is loss of core database features such as relations and ACID properties. There is an ongoing debate about the usefulness of NoSQL, and whether its supposed performance gains justify the significant compromise in features and consistency [3].

Despite these drawbacks, this type of database still remains popular in the industry.

1.1 Purpose

At the heart of this project lies the design and implementation of a database that addresses the issues present in traditional relational databases. The database should be distributed, scalable and optimized for high volumes of concurrent queries. Different approaches to these problems will be researched; those found most appropriate will be implemented and compared against each other through benchmark testing. Ideally, this should yield some clear results on which approach is most suitable for our type of database. This investigation forms the main objective of this project. The testing process should also reveal to what degree the database fulfills its specification above.

2 Technical Background

The implementation of a distributed storage system necessitates a rich theoretical knowledge base. This chapter aims to shed light on the technical terms central to the subject area, and explain the concepts that make up the foundation of the project. Implementation-specific details will not be covered here, just background information relevant to understanding the implementation.

2.1 Databases

A *database* is able to store great amounts of data in an organized manner. Here follow brief descriptions on three database classes of some importance to the project.

SQL This is the traditional database type. It is based around relations between data and adheres to ACID properties. This class is time proven and SQL databases are highly optimized. However they have been designed to be deployed on just one computer, which places limits on concurrent performance.

NoSQL NoSQL is a relatively (at the time of writing) new set of databases. The common denominator is a simplified data and query model, and the sacrifice of ACID relations. This allows for simple division of the data space over many nodes. NoSQL databases generally enjoy increased read and write throughput as well as increased total storage capacity. Typically, they are distributed systems.

NewSQL There are times when being limited to the simplified data models of NoSQL databases makes such databases inconvenient to use. NewSQL places itself somewhere inbetween the first two sets of databases. Allowing databases to be distributed, they still support ACID properties and SQL queries.

2.2 Distribution

A *distributed system* is a collection of independent computers, communicating through a network while cooperating on some task. It should be transparent and appear as a single coherent system to its users. The design of such a system can be challenging and there are several issues to handle, such as nodes or networks going down or nodes acting erroneously. The advantages of a distributed system is its wide availability, reliability and strong performance. Figure 1 shows a general illustration of a distributed system.

A distributed system can be centralized or decentralized. In the former case, one node is assigned special responsibilities and thus acts as a manager for all other nodes. This model is simple to implement, but is error prone due to having one single point of failure. The more complicated decentralized model eliminates this problem by distributing responsibilites equally over nodes.

2.2.1 Fallacies

The designer of a distributed system faces many challenges. In 1994, Peter Deutsch formulated seven false assumptions the designer is likely to make which, with an addition by James Gosling

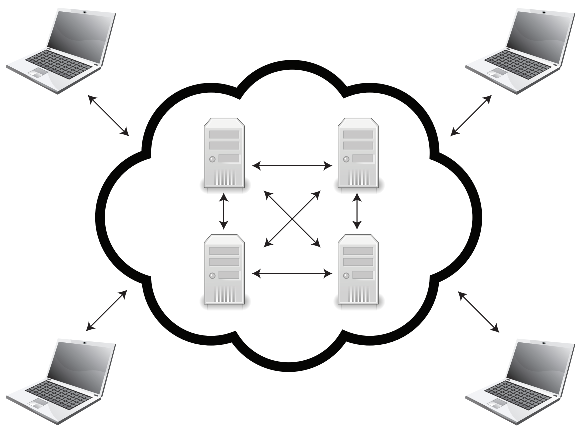


Figure 1: Distributed system.

in 1997, form the *Eight Fallacies of Distributed Computing* [4]. They are stated as follows. Brief descriptions have been added.

1. **The network is reliable:** Many failures are possible in a network. Power failures occur at times. Network cables can be pulled or break. Routers can be broken or incorrectly configured.

2. **Latency is zero:** In LANs the latency is usually low, but in WANs it may turn out to be very high.

3. **Bandwidth is infinite:** Even though the total bandwidth capacity of internet is increasing, new applications such as video and audio streaming send increasingly large amounts of data across networks.

4. **The network is secure:** There are a number of known and unknown attacks targeting networks and computers in networks.

5. **Topology doesn't change:** Servers are removed and added. Dynamic IP addresses have the inherent issue that networks and routers will receive new IP addresses when restarted.

6. **There is one administrator:** As network messages move through a number of networks, the risk of networks being incompatible and the risk of experiencing errors grows.

7. **Transport cost is zero:** It isn't free to use bandwidth, so applications using the internet might turn out to be more expensive than thought at first glance. Neither is it free (in terms of processing resources) to move from application level to network level.

8. **The network is homogeneous:** Devices connected to networks are not all the same, and they probably don't all speak the same language. It might prove hard to make them communicate.

2.3 Distributed Databases

A *distributed database* is a database running on multiple machines. It is typically implemented to be fault-tolerant, meaning that it is able to keep up operations even in the event of node failure.

This section will cover two different approaches to distributing data over a set of nodes: replication and partitioning.

Replicating data A replicated distributed database keeps full replicas of the entire database at each node. The major problem these databases face is how to keep replicas consistent. If one replica is updated, it must be ensured that all other replicas are updated as well. To handle this issue, some consistency protocol is usually followed. Based on the importance of keeping the data consistent, this protocol could be strong or weak. The former guarantees all replicas are always consistent, the latter that all replicas will eventually, in the absence of updates, become consistent.

An obvious drawback to the replication approach is that it requires a lot of space. It is also troublesome to update, since new data must be written to many locations. In other words it is not very scalable. An advantage is that data can be found quickly and easily.

Partitioning data Partitioning (also known as fragmenting) the database means that data is scattered among the nodes. Each node holds a subset of the data. This eliminates the problem of having to keep replicas consistent, and also resolves the scaling issues present in the replication model. However, the process of finding data becomes more complicated. The main challenge is making sure the node responsible for the requested data can be looked up in reasonable time.

2.4 Key-value Store

A *key-value store* is a database with a very simple data model, similar to a hash table. There are no relations in the traditional sense, and no complex queries. Data elements are simply stored as a value associated with a key. As a result, a key-value store supports very few operations, just the insertion and deletion of key-value pairs, and the fetching of a value using its key.

Despite this simplicity, there are many practical applications for such a database. In many scenarios where data volumes are very large, relations within the data are often relatively simple. The key-value store can be an appropriate choice in this case. For example, Dynamo, the distributed database behind Amazon's shopping cart, is a key-value store.

This type of database is less complicated to distribute than one with a fully relational model, and can therefore take full advantage of the benefits of distribution, such as speed and scalability. Some applications may need to adapt their data model on the client side in order to be compatible with such a simple database. This may be acceptable if performance and scalability is a priority.

2.5 The Concurrent Paradigm

Closely linked to the notion of distribution, is the notion of *concurrency.* When running a system on several machines at once, several processes will execute simultaneously, and possibly interact with each other. The concurrent programming paradigm embodies this concept.

There are a few problems inherent to the concurrency model, particularly when it comes to ensuring the integrity of data. Many programming languages allow memory to be shared across threads, effectively granting an arbitrary number of actors direct writing access to the same variables. Additional abstractions are needed in order to control access to variables, and avoid corrupted values. This introduces considerable overhead.

Another approach is to forbid memory sharing altogether. In this case, processes will have their own private memory and pass messages between each other to convey state. The sending and receiving of messages leads to significant overhead as well; however, disallowing memory sharing effectively removes the risk of corrupting shared variables [5].

2.6 Erlang

Erlang is a functional programming language with many built in features for distribution and concurrency [6]. It supports a large amount of concurrent processes, due to processes being extremely lightweight and scheduled by the Erlang virtual machine instead of the underlying operating system. Each process has its own memory and they communicate through message passing. Another benefit to the Erlang distribution model is its built in fault tolerance: processes can monitor the status of other processes, even on remote nodes, and detect if and why a process dies.

OTP Many distributed programs follow a common pattern, with operations such as spawning, registering and monitoring processes. This led to the development of *Open Telecom Platform*, which is a set of Erlang modules and design patterns to help build these systems. OTP supports a generic model for a process (called a supervisor) to monitor other processes (called workers) and perform different actions should some of them go down. A typical way of using workers is the so-called generic server model, which handles client-server behaviour. All OTP generic models have a standard set of interface functions; using these provides stability to operations such as spawning and monitoring processes. This considerably streamlines the development of server applications, and is widely used in real-world implementations.

2.7 The Chord Protocol

The *Chord protocol* [7] is central to the project, and is discussed in detail in this section. It is a distributed, decentralized and scalable protocol for looking up the location of a particular data element in a distributed system.

In the Chord protocol, each node and data element is assigned a key using a hash function. These keys are then organized in a logical ring, where each node is responsible for all data elements in the interval immediately preceding the node. As the number of nodes grow larger, this strategy provides a natural load balancing.

All nodes operate in the same way, and no node is more important than any other. Nodes can join and leave the network dynamically. This way, the Chord network is fully decentralized.

Each node is only aware of a small subset of the other nodes in the network. Because of this, the search for a node can involve querying several other nodes before the correct one is found. By distributing the routing information this way, the network becomes scalable. When the network topology changes, not all nodes need to update their information, only the small subset that is affected by the change. The algorithm is also designed so that the correct node can always be

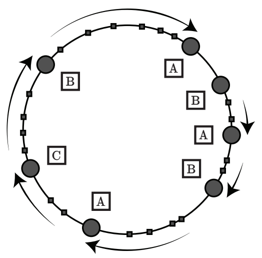


Figure 2: Consistent hashing.

found, even if all routing information has not yet been propagated. Lookup will then be slower, but the data will still be available. Below, we treat the different aspects of the protocol in more detail.

Consistent hashing For the hashing of keys, Chord utilizes a variation of a hashing strategy known as *consistent hashing*.

A naive approach to hashing would be to use a conventional mathematical hash function h, modulo the number of entries in the hash table or the number of nodes. However, this creates severe problems when the hash table is resized, or a node is added or removed. A majority of the keys would have to be moved to another node in this case. To solve this problem, Karger et al introduced the so called consistent hashing method [8].

The key feature in consistent hashing is the relative ability to retain the mapping of keys to hash table entries or nodes. In average, only K/N keys have to be moved when a new node is inserted, where K is the number of keys and N is the previous number of nodes in the system.

This is accomplished by mapping all hash values to a different point at the edge of a circle. Each node is mapped to many randomly distributed points on the same circle. When finding the appropriate node for a key, the system steps through the nodes on the circle until it finds a node point that succeeds the hashed key on the circle. This node is chosen.

When node A is added to such a circle, some of the keys previously assigned to other nodes will now instead be handled by A. These nodes have to transfer these key-value pairs to A. Similarly, when node A disconnects from the system, these other nodes will regain responsibility for the values previously handled by A. These values have to be transferred from A to the other nodes before A disconnects, and in case of a non-graceful disconnect (i.e. network disruption or a system crash) this requires data replication in addition to data partitioning. However, these issues are outside the scope of this section.

As illustrated by Figure 2, a node occupies several positions on the ring. This design provides a high degree of load balancing. Karger et al show that, with high probability, each node will be responsible for at most $(1+\epsilon)K/N$ keys. ϵ can be arbitrarily small if each node runs $O(\log N)$ virtual nodes, each with its own position on the ring.

Stoica et al mention that Chord can be implemented with virtual nodes, similar to consistent hashing as proposed by Karger et al. However, Chord can also be implemented without the use of virtual nodes. This eases the presentation and implementation. In this case, each node has only one position on the ring. In this case, with high probability, the load on a node may exceed the average by (at most) an $O(\log N)$ factor. This is still a reasonable load balance.

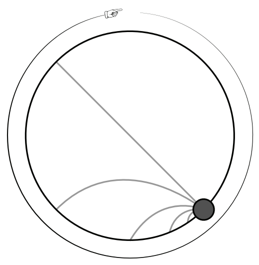


Figure 3: Principal illustration of the Chord finger table.

Routing and lookup Each node maintains a routing table to other nodes known as a *finger table*. The size of this table is determined by the size of the key space being used. If P is the size of the key space, the finger table will have $M = \log P$ fingers. This also corresponds to the number of bits in the hash, so if the 160-bit SHA hash function is used, the finger table will have 160 entries. Each finger refers to a position on the ring relative to the node. The first entry refers to the immediately following position on the ring. The next finger doubles the distance, referring two positions away. The third doubles the distance again, pointing four positions away, and so on. Since we have M fingers, the last finger will be referring to the position directly opposite on the ring. Figure 3 shows an illustration of a finger table when $M = 5$.

Each finger keeps a reference to the first node that comes immediately after the position the finger is referring to. When a lookup for a key is performed, the node searches its finger table for the closest node preceding the key being searched. The search is then passed on to that node recursively, until the node immediately preceding the key is found. This node is now able to return the correct node that is responsible for the key, namely its most immediate successor.

Due to the design of the finger table, where each finger covers exponentially larger intervals, the distance to the target node will decrease by at least half for each step in the search. This implies that the number of queries required to find the correct node for a key is bounded above by $O(\log N)$, where N is the number of nodes in the system. This means that the search time in the network grows only logarithmically as the network becomes larger.

Joining the Chord ring, the straightforward way At any time, a new node may join the ring. In order to do so, the joining node contacts any of the nodes already in the network. The joining node is assigned a key, typically by hashing its hostname, IP address or some other unique identifier. Using the lookup function described above, the correct place on the ring is determined. Specifically, the node that will immediately follow the new node is returned by the lookup. This node knows about its own predecessor, so new node can now notify both its new successor and predecessor of its existence. Using the same lookup function, the finger table of the new node is built.

The new node must also be a part of some other nodes' finger tables. This is performed using a variant of the lookup function to find the nodes that might need to update their finger tables.

From here on, this approach is referred to as *aggressive*.

A more stable way of joining The above join method is simple, but has problems. In a large network, the system will experience several nodes joining at the same time. In such scenarios, the pointers between nodes may become corrupted if they are updated by different joining nodes simultaneously. Also, this method does not provide a way for the routing information to stay up-to-date when a node voluntarily or involuntarily leaves the network. A more sophisticated joining method, addressing these issues, is described below.

Here, when a node is joining, it does not notify its predecessor but only its successor. The predecessor will then update itself in a so-called stabilization event. During stabilization, each node asks its successor for the successor's predecessor. Normally this will be the asking node itself. But if a new node has joined just preceding the successor, the asking node has to accept the new node as its new successor. This way, several nodes can join concurrently. They will be fully accepted, one after another, as the stabilization events occur.

The new node must also be a part of some other nodes' finger tables. This is performed by periodically making sure the finger table of each node is up-to-date, using the previously described lookup function.

Once the node has joined the ring, all elements belonging to this node will be transferred from its successor.

Leaving the ring, gracefully or not When a node leaves the network, it notifies its immediate neighbors of its departure and transfers all its keys to the appropriate node. If the node does not leave voluntarily, but experiences failure, no such procedure will take place. Still, the keys held by the failing node must not be lost. In order to handle failures, copies of data elements are replicated on other nodes, namely the r most immediate successors. A procedure similar to stabilize will keep this list up to date. If a node detects that its successor has failed, it will instead direct queries to the next available node in the predecessor list. This node will have a replicated version of the failed node's data. Eventually, the stabilize routine will correct all pointers to the new responsible node. The behaviour for leaving the ring requires that the stabilize routine is running. Therefore, a Chord implementation using the aggressive join mechanism described above does not handle departures in a well defined way.

2.8 Fully Connected Network (FCN)

A design similar to Chord is the *Fully Connected Network*, or FCN. FCN is a name the authors of this report have given to the system. It is similar to Chord in that nodes and key-value pairs are placed on a logical ring, and that each element is stored on its clockwise successor node on the ring.

Unlike Chord, FCN nodes do not use a finger table. Instead, each node keeps a reference to all other nodes in the network. This makes the lookup problem trivial, and it can be solved locally on each node. However, the routing information on each node grows at the same rate as the number of nodes in the network, implying that the accumulated routing information in the system grows at a quadratic rate. This results in a large amount of connections even for small numbers of nodes. Figure 4 illustrates this, showing a comparison of Chord and FCN's numbers of connections in a network with 32 nodes.

When a new node joins the network, it needs to contact every other node in order to have them update their finger tables. This makes the join operation more expensive by a factor of $O(n)$, where n is the number of nodes in the system, as the network size increases.

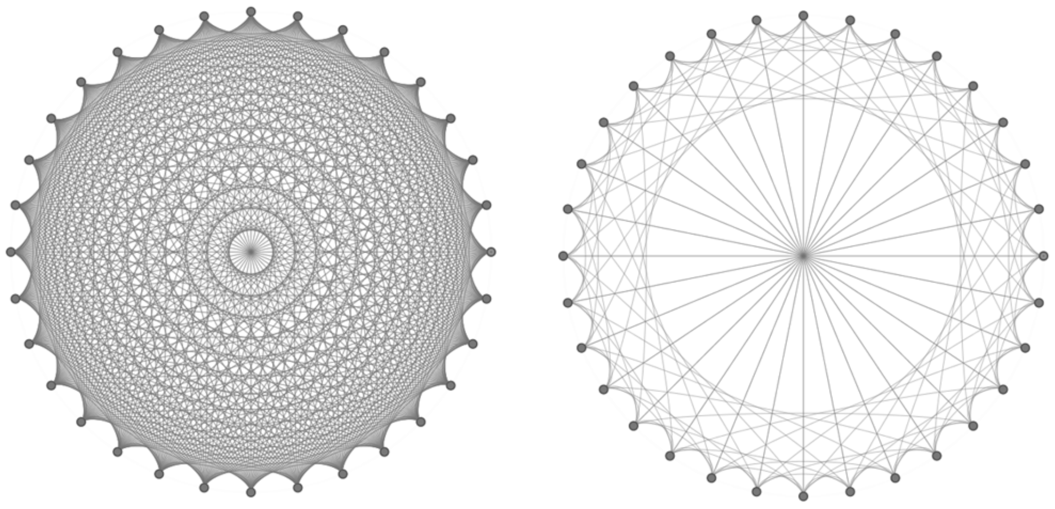


Figure 4: Different numbers of connections in FCN (left) and Chord (right).

3 Method

This section concretizes the objectives and goals laid out in the project purpose. The different parts of the project are outlined in more detail and ambitions and limitations are defined. The tools and languages chosen for these tasks are also described.

3.1 Main Project Parts

The implementation is divided into three main areas: the database itself, a testing environment, and a visualization application.

3.1.1 Database

Several data models were considered, for example the key-value store described in Section 2.4, and the more complicated document store which models data as loosely stored documents. The key-value store was selected due to its relative simplicity, which allowed the project to focus more on distribution.

In order to have a scalable, flexible, fault tolerant distributed system, a decentralized approach was adopted. In this architecture, responsibilities are equal amongst all database nodes, with no single point of failure or obvious bottleneck. This is also known as a peer-to-peer architecture.

Since a very scalable database was desired, it was natural to discard the replicated data approach in favor of partitioning. In a decentralized, partitioned system, algorithms are needed to structure the system and make sure data can be stored and fetched correctly.

Finding an algorithm The search for possible solutions was narrowed down to three algorithms: Chord, CAN [9], and Pastry [10]. CAN is the most complex of the three, but it was clear that any potential performance gain over the others would not be apparent for smaller numbers of nodes, and no interesting differences would be possible to measure in our testing. Out of the two remaining algorithms, the choice was not obvious, however Chord seemed more intuitive and simple to implement, and was therefore chosen.

Two implementations Along with the pure Chord implementation, its simpler variation FCN was also implemented. An approach similar to the latter is used by Facebook's Cassandra [1] and Amazon's Dynamo [2]. It is clear that lookups will be faster in a system where all nodes know about each other, but it should be interesting to see how large this difference is, and what the trade-offs are to this advantage.

Additional challenges When storing data in a decentralized system, two of the greatest challenges are how to keep any replicated data consistent, and how to make sure no data is lost in the event of failure. Naturally, this had to be considered when designing the system.

Once data storage is reliable, an important problem to solve is how to partition the data. Chord's consistent hashing (see Section 2.7) offers a built-in solution for this problem. The same approach was used for the FCN.

3.1.2 Test Bed

To be able to compare the two implementations against each other, and also measure whether the database met the requirements outlined in the purpose, a testing environment was implemented. In order to simulate a realistic usage scenario for the database, this test bed would itself have to be distributed over several machines. It should be able to stress test a database with any number of queries from any number of nodes, collecting data on performance. The test bed should be general enough for many kinds of tests to be run without any modifications to the test bed itself. This means it could potentially be used for any system. Finally, in order to be user friendly, it should be possible to start and stop processes on all test hosts from just one computer.

More details on the testing process will follow in Section 3.2.

3.1.3 Visualization

A distributed system is inherently difficult to overview and debug. The algorithms are executed on different machines concurrently, and there is no obvious way to get a picture of what is going on. A distributed system is also difficult to demonstrate in a presentation, except for very small instances. For these reasons, there was a need for a separate visualization application. This application should, in real time, visualize the state of the database, as well as how the algorithms operate in the system. This will be useful not only for debugging and testing, but also for showing during a presentation. For this reason, it should be aesthetically pleasing and intuitive to understand. Besides representing the state of an actual database, the visualization should also be able to simulate the system independently. This is appropriate for implementation purposes, but it also allows the visualization to simulate scenarios that are not feasible to test in practice. Such an application should be valuable as a complementary tool for the database as well as an interesting product in itself.

3.2 Experiments

There are three aspects which should be tested for: distribution, query throughput, and scalability. If the database is good at these things, the implementation has been a success.

Distribution To see whether data is distributed fairly among nodes, data elements were counted on each node and the standard deviation was calculated. Assuming there is a correlation between the amount of data elements on a node and the amount of traffic that node has to handle, this should give an indication of how well work is distributed over nodes in the network. While the results here could in part be predicted by analyzing the algorithms, it would also be interesting to see the numbers in practice.

Throughput The important thing here is whether the database can handle queries in a quick and concurrent manner. This was tested by measuring the number of queries handled per second.

Scalability It is important that the database operates quickly not only when receiving many queries per second, but also when the database itself has grown very large. The former was measured by increasing the number of test clients in the test bed, the latter by increasing the number of database nodes in the network. Since scaling the number of machines should give some hints on how the database would perform in a real world setting, this has also been done.

3.3 Limitations

Due to the extensive nature of the subject, it was important to narrow down the project scope. The following areas have been compromised or left uncovered.

Scale Naturally, it would be impossible to simulate the kind of traffic generated at real-world sites such as Facebook. However, the amount of queries from each client can be scaled up quite far, which would be comparable to a normal amount of queries from a large number of clients. This solution was chosen as a compromise.

Security In a real world situation, some security measures would be necessary in a database product, but this is beyond the defined scope of this project.

Disk Access As with the security aspect mentioned above, disk access is also outside of the project scope. Instead, data is cached in local memory, which would roughly mirror the performance rates of extremely well optimized enterprise databases.

Testing There are many outside factors that could affect the outcome of the tests. In the scope of this project it is not practically possible to consider them all. Among those not considered are Erlang runtime parameters. Changing these may have considerable impact on performance [11], but this has not been taken into account.

3.4 Development Tools

Erlang After outlining the project, it was clear that extensive support for distribution and concurrency was needed. A distributed environment benefits from advanced fault tolerance handling, so some support for this would be necessary as well. Evaluation of possible platforms pointed towards Erlang and OTP which has well developed support for these demands. After some initial experimental attempts it became obvious that this was the right choice. While it is likely that the database would have been faster if implemented in for instance C, the convenience of Erlang simply could not be beaten. In fact, some of the most prominent databases in the NoSQL category are successfully using Erlang as their main tool. Among them are CouchDB [12], RIAK [13] and Erlang's built in NoSQL database mnesia [14]. The creation of the mnesia database was motivated by fast key/value lookup, need to distribute data and high fault tolerance; goals that lie close to the stated purpose of this project.

Processing Erlang has no proper support for rendering real time graphics. When planning the visualization it became apparent another platform was needed for that. There are many possible options for displaying graphics, among them libraries such as openFrameworks [15], libcinder [16] and Processing [17]. The first two are implemented in C++. Processing uses Java.

In order to interface the Erlang Database implementation with the Java visualization, an existing java-erlang library was used.

Processing allows developers to use a simplified version of Java. But it can also been used as just a Java library simplifying access to graphics rendering libraries such as openGL. The latter approach was used in this project.

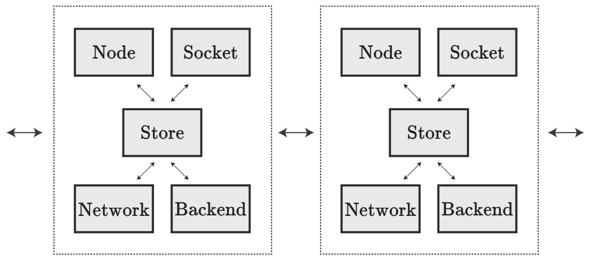


Figure 5: Database modules.

4 Results

This project has yielded results in two categories: implementation and experiments. The former will be described in Sections 4.1 through 4.4, focusing on which problems had to be solved and how this was done. The latter will follow in Section 4.5, presenting the data obtained from tests.

4.1 Database

As mentioned in Section 3.1.1, there is a number of issues inherent to the decentralized, partitioned model of distribution. They all had to be considered in the database implementation. Also, there was the challenge of how to sufficiently modularize the database. Considering the objective of performing tests on different implementations, switching between these implementations had to be a smooth and painless process. There was also a need for an efficient concurrency model. The system must process queries in parallel; if it was to handle them sequentially, total processing time would scale linearly with the amount of queries, and quickly become unacceptably high.

The following subsections will explain how these problems were tackled and solved. Many of the issues can be categorized under some headings in the fallacy list presented in Section 2.2.1. Therefore, the final subsection will refer back to the fallacies and sum up their impact on the implementation.

4.1.1 General Details

The database was implemented in Erlang, taking advantage of many of the features built into the language; one of the most important ones being the ability to link processes together and have them monitor each other. The project makes heavy use of a proven and reliable approach to this concept: the OTP supervision tree, which divides supervising and working responsibilities between processes in a structured manner. Another Erlang feature used by the database is the OTP application model, a very useful project file structure and configuration handling standardization. Adhering to this standard means greater control for the user, for instance when it comes to starting and stopping the system. A drawback to using Erlang was the fact that it does not enforce any type checking. This made development and debugging more time consuming. However, the benefits of the language greatly outweighed this disadvantage.

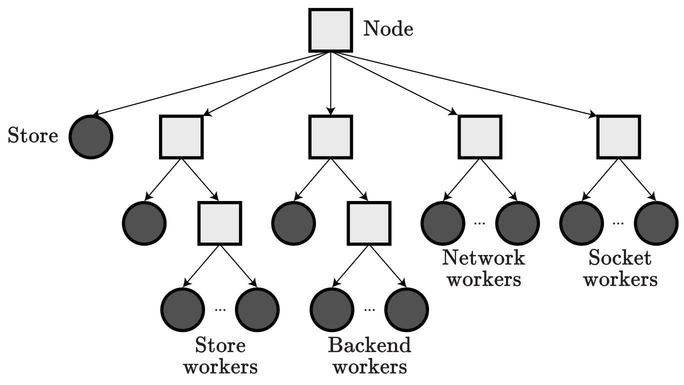


Figure 6: OTP supervision tree.

4.1.2 Modularity

Every database node consists of the same collection of modules. Figure 5 shows two database nodes, illustrating how modules communicate internally inside a node, and also external communication between the two. All modules are structured according to the OTP supervision tree mentioned above.

Node The Node module is located at the very top of the supervision tree, and thus acts as the master module for a single database node. It starts and monitors all other modules. It also doubles as a client interface, providing some simple commands such as starting the database, joining an existing database network, and sending queries. Figure 6 visualizes the supervision tree, showing supervisor processes as squares and worker processes as circles. It can be seen that nearly all direct children of Node are actually supervisors monitoring their own worker processes.

Network All connections between nodes are handled by the Network module. Its main responsibility is to provide a function that finds and returns the node associated with a given id. This function is called by a number of other modules. These modules neither know nor care how lookup has been implemented, as long as it does its job. Hence, any number of Network modules could be implemented and easily switched out for each other, as long as they export the same basic functionality. In this project, two different implementations were made: Chord and FCN. Details about them can be found under Sections 4.1.5 and 4.1.6.

Store As can be seen in Figure 5, the Store module exchanges messages with all other modules, and can be considered the middleman between the Node/Socket and Network/Backend parts of the database. Requests for inserting or fetching data, as well as for a node to join or leave the network, all pass through the Storage module. In the former case, it will query the network to find the responsible node, and then propagate the data request to the backend at that node. In the latter case, it will send the join/leave request to the network, then query the backend to make sure any necessary moving of data is carried out.

Backend This module is responsible for storing the actual data. To accomplish this it uses a so-called Erlang ETS table, a native Erlang data structure supporting storage of large quantities of data. Among the benefits of ETS is constant access time and built-in support for concurrent data access. Just as with the Network module, the ETS implementation could easily be switched out for some other method of storage.

Socket The Socket module listens on a TCP port for connections from clients. Each client connection is handled by a Socket worker process, which makes sure the client's request is passed on to the appropriate module.

A database must be able to communicate with other systems; hence, a communication protocol using these socket connections has been developed. The protocol is used to communicate between database nodes and clients. It is described in detail in Appendix A.1.

4.1.3 Concurrency

The concurrency issue starts at socket level, where multiple connections must be handled at once, and continues through storage level all the way to the backend, where the reading and writing of data must proceed in parallel. It is crucial that the handling of one query does not block other queries from being processed.

In solving these issues, the processes of Erlang become a major advantage. They run concurrently by design, using the message passing model mentioned in Section 2.5. Also, they are lightweight enough to be spawned to handle a single task, and die once it is done [6].

The socket level concurrency issue is easily solved by letting multiple Erlang processes wait for connections. Each incoming connection is assigned a storage worker process; each of these processes passes its query on to the network and backend modules, both of which will spawn worker processes of their own. The processes will execute in parallel, and die once they have completed their task.

4.1.4 Fault-tolerance and Consistency

In a fault tolerant database, the data on each node must to some extent be replicated. In the event of failure, the responsibility of a failing node should be inherited by some other node.

As a solution, each database node stores and updates information about its potential inheritors. As local data becomes modified, each inheritor is notified, and updates a maintained replica accordingly, making sure the data stays consistent. If node failure occurs, the affected nodes should be updated. By using monitoring, as provided by Erlang, each node can be linked to each other and hence be notified in case of a failure.

4.1.5 Chord Implementation

Implementing Chord has been a central part of the project. However, implementing the algorithm in Erlang faced some particular design and implementation challenges. These will be outlined in this section. In Section 2.7, it was noted that several aspects of Chord can be varied. These different variations were explored in the implementation.

A four part design was selected for the Chord module. Server, Setup, Lookup, and a Library module.

Four part design The *server* module takes care of the general infrastructure needed to communicate with higher-layer software and other nodes. The server delegates tasks to the Setup and Lookup modules. The *setup* module provides functionality for joining the ring. The *lookup* module provides the lookup function for mapping a key to a node. Finally, the *library* module collects common utility functions such as math, logic and functions related to the finger table.

Parallelism The Chord module, like other parts of the database, was parallelized. Once a Chord query is received, a new process is spawned to handle the call. This way, several queries can be handled concurrently.

The routing information (the finger table and the predecessor pointer) is located in the Server module. Changes to this routing information is only performed by the server process itself, ensuring that data integrity is not compromised by concurrent updates.

The parallelism of queries conveniently solved another general problem, where nodes were deadlocked trying to query each other. For example, joins often led to situations where a node A, while handling a request to join the ring, caused another node C to send additional queries to node A, which was already busy. Without parallel queries, solving this issue would have required a separate construct.

Message passing The Erlang concurrency model based on message passing was particularly suitable for the implementation of Chord. This solved most of the concurrency-related problems otherwise associated with a distributed system. Still, there were some issues adapting the algorithm to this paradigm.

A central issue in the implementation was that the Chord algorithm does not distinguish between a node querying itself and it querying other nodes. For example, a node's finger table may contain references to the node itself; that node will then query itself as part of the algorithm. Such scenarios imposes particular challenges in Erlang. Queries are implemented by message passing, and when a message handler in a process naively messages the own process, there will be a deadlock. In order to mitigate this, a series of wrapper functions were added. Now, an actual message is not sent unless the recipient is another node. If the recipient is the node itself, the local equivalent function is called instead.

Intervals on a ring All nodes are responsible for the interval on the ring immediately preceding it. The calculation of these intervals proved to be somewhat complicated, in particular when reaching around the end of the circle, when the succeeding node has a position close to zero. Three different cases were identified, and the logic for these had to be carefully implemented.

Recursive versus iterative lookup In [7], it is mentioned that the lookup function that resolves keys to nodes can be implemented in two different ways, iterative and recursive. Both approaches have been implemented. The iterative version is still implemented in the functional style of Erlang, but executes the search algorithm locally on the node, and queries other nodes doing so. The recursive implementation had a difficult deadlock issue which occurred when two nodes were simultaneously querying each other. This issue was not fully resolved, and the iterative version was used in testing.

Maintaining routing information The routing information can be kept up to date with two different algorithms, aggressive and stabilized. The aggressive approach was fully implemented, and is the main method used by the database. As noted in Section 2.7, this approach can not handle several nodes joining concurrently in a reliable way.

The stabilized version was also implemented, but faced a difficult race condition that was not fully solved. The effect of this is that the database does not handle concurrent joins, and care was taken during testing to allow each join procedure to finish before launching a new one.

4.1.6 FCN Implementation

The fully connected network, being a simplification of the Chord protocol, was less complicated to implement than its counterpart. The search algorithm is straightforward, and is implemented locally on the node. Because of this, a modularized design like Chord was not necessary for the FCN implementation, and much of the issues faced by the Chord implementation were not present for FCN.

Lookup Since each node keeps a reference to all other nodes, and their respective placement on the key ring, the lookup problem is trivial in FCN. The FCN node responsible for the lookup just looks through its data structure holding the references to other nodes, until it finds the proper node for a given key. Similarly to Chord, the FCN module was required to handle many queries concurrently. Here, a solution similar to the Chord module was implemented.

Joining the ring When node N joins the network, it needs the path to another arbitrary node B in the network. N sends a *join request* to B. B then proceeds to inform all other existing nodes in the network, and returns its list of all the other nodes to N.

Leaving the ring When a node leaves the network, it is responsible for notifying all the other nodes about its absence. If a node crashes, or the network connection is interrupted, the other nodes are automatically notified by a standard Erlang process monitor.

4.1.7 Handling Fallacies

Out of the eight fallacies listed in Section 2.2.1, some have been very central to the project. These will be outlined below. The ones not mentioned have not been applicable, or ignored due to time limitations.

The Network is Reliable In the context of this database, handling this fallacy comes down to how to handle losing connection to nodes. In the general case, this is taken care of by the fault tolerance in the system, mentioned in Section 4.1.4. However more extreme cases, for example if a power outage caused half of the nodes to lose their connection, have not been accounted for in this implementation.

Topology Doesn't Change A partitioned distributed storage system must be prepared to handle this issue. It is simply in the nature of these systems that the number of nodes is not constant. Accordingly, this has been covered thoroughly, both in the case of Chord and FCN. For more detailed information, refer back to Sections 4.1.5 and 4.1.6.

Transport Cost is Zero In order to keep transport costs down, transported data must be kept at a minimum. In the case of this implementation, this refers to messages passed between nodes. Chord is specifically designed to minimize the amount of message passing, which solves this problem nicely.

The Network is Homogeneous A low level protocol is necessary in order to let different types of devices communicate through the network. This has been implemented and described on page 15.

4.2 Test Bed

As outlined in the Method chapter, the test bed should be distributed, general, and user friendly. Apart from these design details, there also exists the problem of how the actual tests should be designed and performed. They need to be designed to measure relevant data, and performed in such a way that these data are properly isolated and reliably returned. Large scale testing of a distributed system is very time consuming. If all nodes have to be manually configured, and tests are supposed to be executed with different configurations, the time spent on configuring and re-configuring the nodes will become unmanageable. Some automatization will obviously be required.

4.2.1 Implementation

A centralized distribution architecture, where one node acts as master, was chosen. This allows the user to enter commands into the master node, and have that node replicate the commands over all other nodes, without the slave nodes having to know about each others existence. This is known as the master/slave architecture. Due to the distributed nature of this solution, Erlang was the obvious choice of development platform.

Master The master node keeps track of all slave nodes, sends tests to the slave nodes when prompted by the user, and evaluates test results once returned from the slaves.

It has functionality to start a given number of slaves partitioned over the available nodes, and subsequently maintain a list of all these slaves. When asked to perform a test, this test must somehow be distributed over all slaves in the list, and performed by them to yield a result. This is achieved by sending a message containing an executable test file to each slave. By switching out the executable, the user can ask the test bed to carry out any kind of test. When all slaves have reported their results, the master will perform some calculations to make sense of these results. To achieve greater generality, these calculations could have been carried out by a separate module, but this was not a big priority.

Slave Slave processes follow a very simple structure. They wait for commands from the master node, execute the test once a file is sent, and report back the results once finished. When running a test, the slaves measure the total time it takes to run it. The master is notified of the elapsed time and is also given a list of all results from all tests.

4.2.2 Testing

It is very important that the delay and traffic associated with the setup of test cases does not interfere with the actual testing. This means that the setup process must be completely separated from the testing process. This separation is achieved by enforcing an interface pattern where all test cases must provide special functionality for setup, testing, and teardown.

As an example, when running a test that requires a number of sockets to be started, the sockets must all be set up before any tests are executed. This should be taken care of by the setup functionality mentioned above.

4.3 Visualization

Although Erlang has support for graphical user interfaces, it was deemed inappropriate for a full visualization application. Instead, an object-oriented design and a Java implementation was selected. For the actual rendering, the Java library Processing was used.

Implementation design To represent the database, an object oriented model was designed. The individual database nodes were modeled, along with the data elements. The Chord and the FCN algorithms were then independently implemented in this model.

Graphical design Since the Chord key space is a logical ring, it was natural to also graphically represent the database as a ring. Each node is displayed on its logical position on the ring, and each data element is displayed in the same fashion. One can pan and zoom in on the display, and holding the mouse over a node displays information about it, along with its current connections to other nodes.

Care was taken to make the visualization intuitive and aesthetically pleasing, since it was intended to be used in the presentation of the project. Several color schemes were designed, and graphical details such as the thickness of lines were made customizable. This allowed the application to be quickly adapted to a given setting. The individual parts of the visualization can also be turned on and off interactively, which allows a pedagogic presentation and reduces clutter.

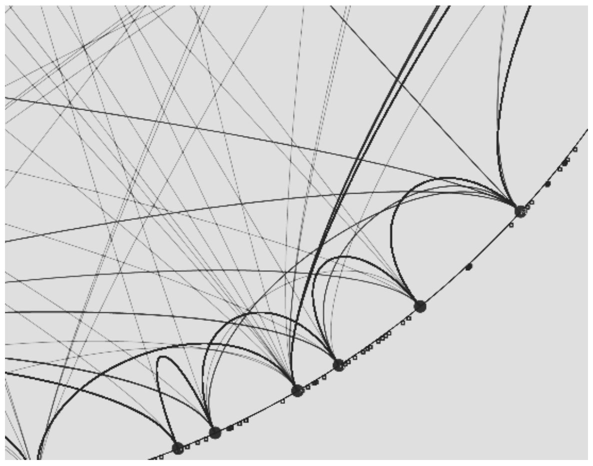


Figure 7: Section of the Chord Ring as represented by the Visualization.

Interactive and absolute mode The visualization has two main modes of operation, interactive and absolute. In interactive mode, the database is simulated locally and is not a representation of an actual database. In this mode one can interactively manipulate the system, add and remove nodes or data elements and view the results in real time.

In the absolute mode, the visualization displays the state of an actual instance of the Erlang implementation. Here, a dedicated thread is utilizing a Java-Erlang interface and acts as an Erlang node that communicates with the test bed, as it is updating the database. If the database is set up to report visualization information, it passes a message to the Java node when an event occurs.

Algorithm selection In the simulation mode, the user can select three different algorithms. In addition to Chord and FCN, a "simple" mode is also available where the query is passed between the nodes in a linear fashion. This is used to introduce the other algorithms and motivate the need for them. One can switch arbitrarily between the three modes without resetting the state of the simulation.

Views The data elements can be displayed in two different ways. Either they are displayed on their logical position on the ring, or on their "physical" position. In the physical view, data elements are clustered around the node they are actually stored on. One can switch interactively between these two modes.

State change animations Each change in the database is animated in the visualization. When a new node or element is added, its graphical representation is moved to its correct position in an animation. This way the viewer is unlikely to miss any changes, and can more intuitively follow what is going on, for example in a presentation. The animation uses a simple spring physics simulation, so an object bounces around a little bit around its location before it comes to rest, further helping the viewer to track changes. Also, the transition between views is animated, so one can follow individual data elements as the view changes.

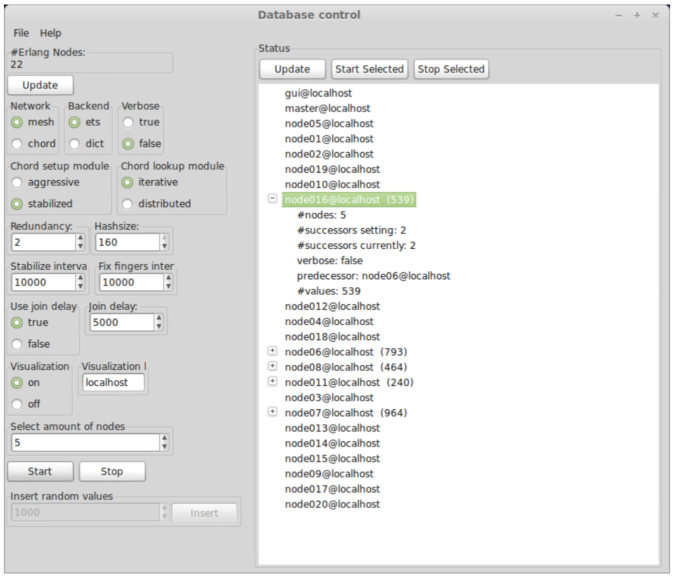
Query Animations The query algorithms themselves can be animated in the visualization. If activated, a graphical representation of the query can be seen travelling between the nodes. All queries take the same amount of time to finish, independent of the distance between the nodes in the ring. This causes some queries to appear slower than others, but since locations on the ring are logical rather than physical, this is reasonable. The speed of the queries are customizable, and are not intended to be realistic. Rather, they illustrate the relative differences in query times between different configurations in an intuitive way.

4.4 Utilities

Some helpful tools have been implemented on top of the main components of the project. They will be outlined in this section.

Database Client The client can be seen as a middle layer between the test bed and the database. It communicates with the database using the protocol described in Appendix A.1.

For the time being, only an Erlang client has been implemented. However, it would be possible to implement clients in other languages, as long as they obey the communication protocol. The



#Erlang Nodes: 22

Network: [✓ mesh] chord
Backend: [✓ ets] dict
Verbose: true [✓ false]
Chord setup module: aggressive [✓ stabilized]
Chord lookup module: [✓ iterative] distributed
Redundancy: 2
Hashsize: 160
Stabilize interva: 10000
Fix fingers inter: 10000
Use join delay: [✓ true] false
Join delay: 5000
Visualization: [✓ on] off
Visualization l: localhost
Select amount of nodes: 5
Insert random values: 1000

Status

```
gui@localhost
master@localhost
node05@localhost
node01@localhost
node02@localhost
node019@localhost
node010@localhost
node016@localhost  (539)
  #nodes: 5
  #successors setting: 2
  #successors currently: 2
  verbose: false
  predecessor: node06@localhost
  #values: 539
node012@localhost
node04@localhost
node018@localhost
node06@localhost  (793)
node08@localhost  (464)
node011@localhost  (240)
node03@localhost
node07@localhost  (964)
node013@localhost
node014@localhost
node015@localhost
node09@localhost
node017@localhost
node020@localhost
```

Figure 8: Database GUI.

Erlang client has a simple interface with commands for starting and stopping the database, and inserting and requesting data.

Database GUI The GUI was implemented in order to create a simple, intuitive tool for starting and stopping database nodes in an existing network of Erlang nodes. Nodes can be set up with a number of different parameters. Figure 8 shows an example of a running database instance of five nodes.

Test Automatization Large scale testing can be performed on an arbitrary number of machines. Once a network of Erlang nodes has been established, these nodes can be fully controlled from one place. This is made possible by a mix of native Erlang tools and custom made ones. Using these tools tests can be configured and run in sequence as needed. The results will be printed out to a file.

4.5 Experiments

This section outlines the results of the experiments performed on the database. After listing the preconditions in hardware and test parameters, the results themselves are presented and explained. Detailed analysis of the charts will follow in the next chapter.

4.5.1 Testing Environment

The tests were carried out over a distributed net of Chalmers computers[1], and are to be considered a simulation of a large number of clients querying the database. Each computer runs 10 instances

[1]Intel Core 2 Duo CPU @ 3 GHz, 3.7 GB RAM

of Erlang, all of which symbolize a single node in the network. This is not quite optimal, since these instances cannot truly be executed in parallel without 10 cores actually working simultaneously on each computer. However this is the most reasonable way of running the desired amount of nodes, due to limitations in the supply of machines. At most 30 machines were used.

Since the system is running in a protected LAN environment, the variations in network bandwidth and latency can be assumed to be small. To the extent these issues do arise, it can be perceived as part of a real world scenario.

The system is not aware of the network topology and neither should it be, it should be able to operate regardless of underlying topology. The tests should not be considerably distorted by this.

The test suites are designed to fail in case one or several computers disconnect from the others. This implies a test case will either complete with the correct amount of computers still running or fail. However it is not guaranteed all Erlang nodes on all computers will stay connected. Before and after each test, it was manually confirmed all nodes were still up.

4.5.2 Testing Parameters

The test cases are designed to be flexible in how many queries, test clients and database nodes are used. However, when measuring queries per second performance across configurations, all tests are set to generate one million queries in total. Half of the queries are *put* and the other half are *get*. All tests use randomization when generating test data and choosing which node to send queries to.

While the number of queries is kept constant across tests, the numbers of clients and database nodes are scaled up incrementally. However, the limited amount of machines placed restrictions on both client and node scaling. When testing with 800 clients, some of them occasionally experienced timeouts, meaning that the test had to be re-run. Since it was likely this would only have gotten worse with more clients, they were not scaled any further beyond this point.

As mentioned, each computer runs 10 instances of Erlang, implying that in the maximum configuration of 30 machines, at most 300 database nodes are possible. When testing for a lower amount, the machines to run them on were chosen at random, however never with more than 10 nodes on a single machine.

Each individual test configuration is run three times, and the average is output as a result. They are run under two separate implementations: Chord and FCN. Because of the low number of tests on each configuration, the results are not to be viewed as final, but rather an indication of what the results would have been under more exhaustive testing. Since one single deviating result will have a large impact on the average, diverging values should be viewed with particular caution.

One aspect that has not been tested for is the performance of the database as it grows and has to update the routing tables at each node. This was excluded partly due to time limitations, but also the fact that there existed some problems with the node joining function in the Chord implementation (see last paragraph of Section 4.1.5), which made it hard to compare Chord and FCN fairly in this context. Because of this, "performance" here simply refers to queries per second in an already stable network of some given size.

4.5.3 Results: Throughput and Scaling

This subsection will present, in chart form, performance results obtained from tests performed under the circumstances outlined above. Performance is measured in queries per second, and it will be noted how this performance varies as the number of clients, database nodes, and/or machines is increased.

The charts' X-axes denote the amount of database nodes present in the tests. Queries per second are shown on the Y-axes. Each line in the chart represents a test carried out with a given number of test clients, where one client symbolizes one simultaneous request to the database.

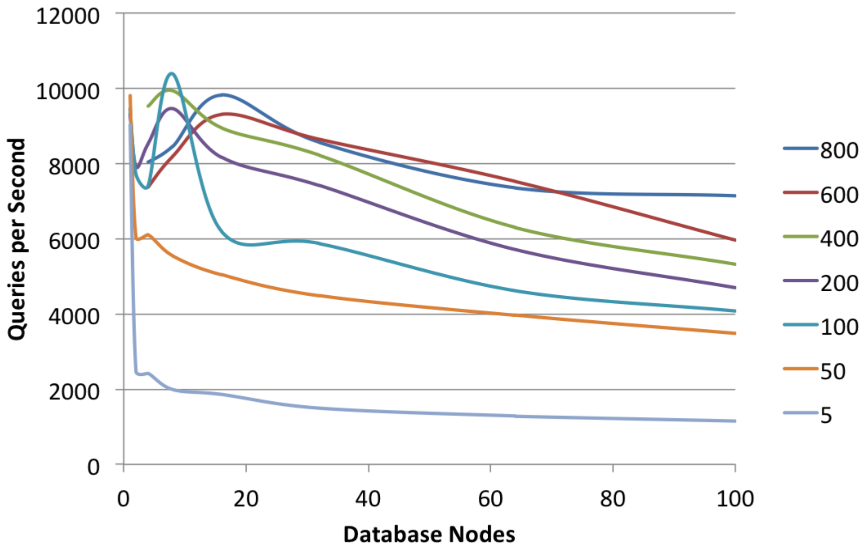


Figure 9: Query frequency performance, Chord on 10 computers.

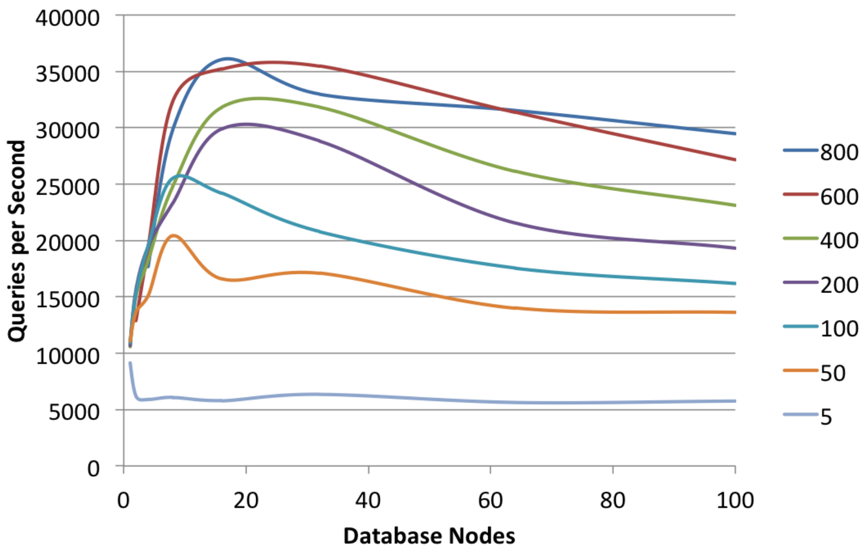


Figure 10: Query frequency performance, FCN on 10 computers.

Figures 9 and 10 show the performance of the Chord and FCN implementations when run on ten computers. (Note that the y-axes are scaled differently.) As can clearly be seen, there is a positive correlation between performance and the number of test clients. In contrast, when scaling the number of database nodes, performance peaks quite early.

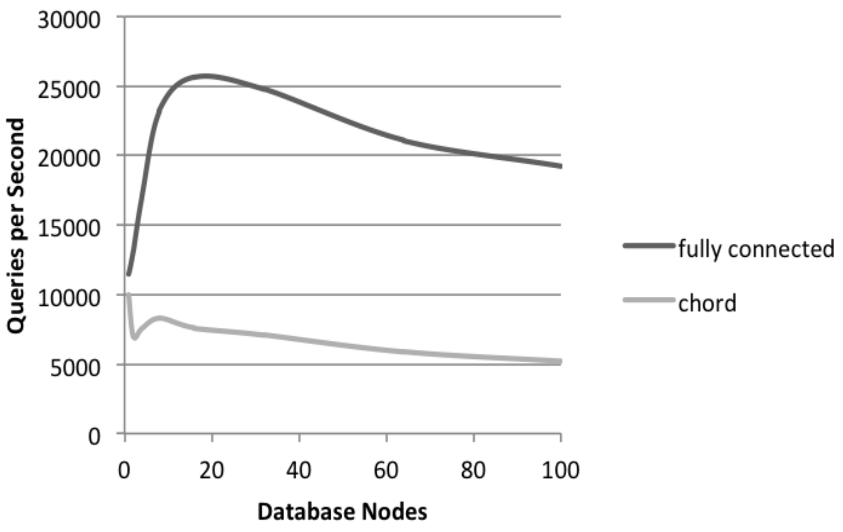


Figure 11: Average performances of the fully connected network and Chord on 10 computers.

Upon closer examination of figures 9 and 10, it is apparent that the Chord implementation is substantially slower than the fully connected network. Figure 11 clarifies this relationship. It can be seen that the fully connected network configuration has its peak around 20 database nodes. The Chord curve peaks as early as the one node mark.

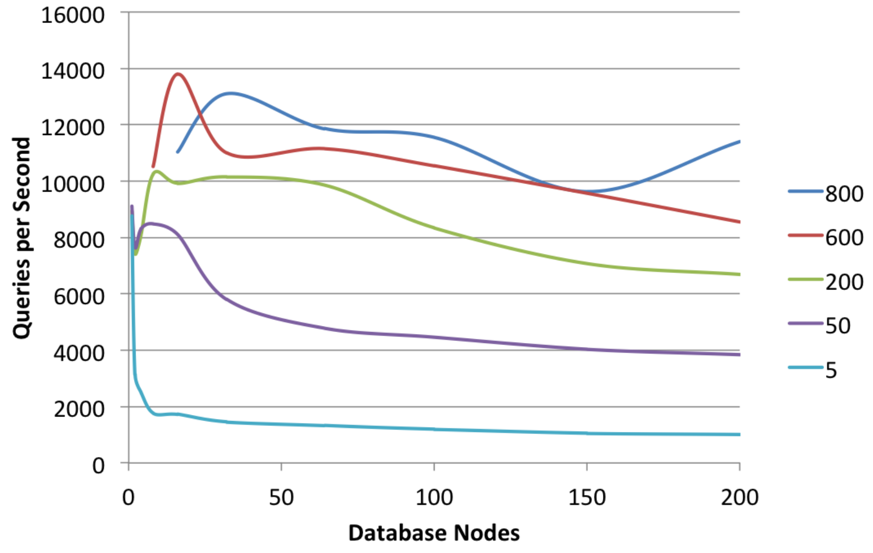


Figure 12: Query frequency performance, Chord on 20 computers.

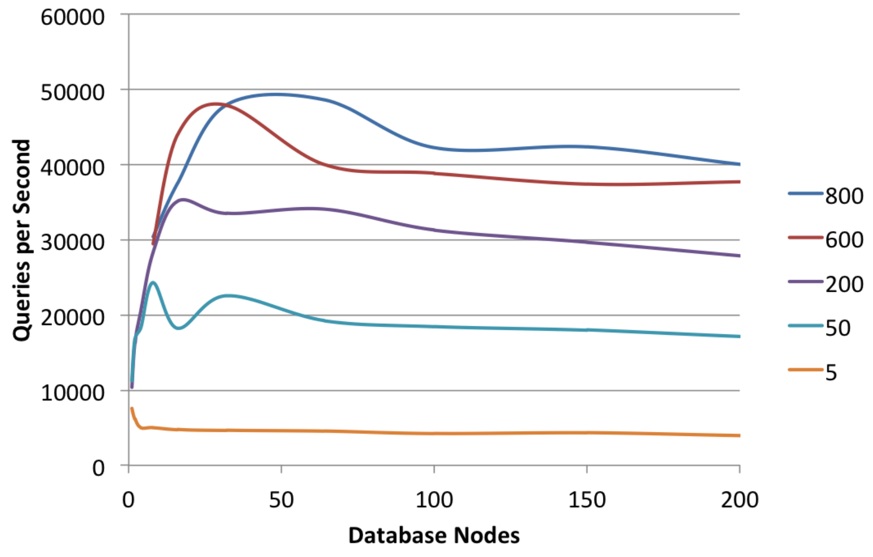


Figure 13: Query frequency performance, fully connected network on 20 computers.

Figures 12 and 13 parallel the previous charts, except run on 20 computers. Note that overall performance has improved considerably for both implementations.

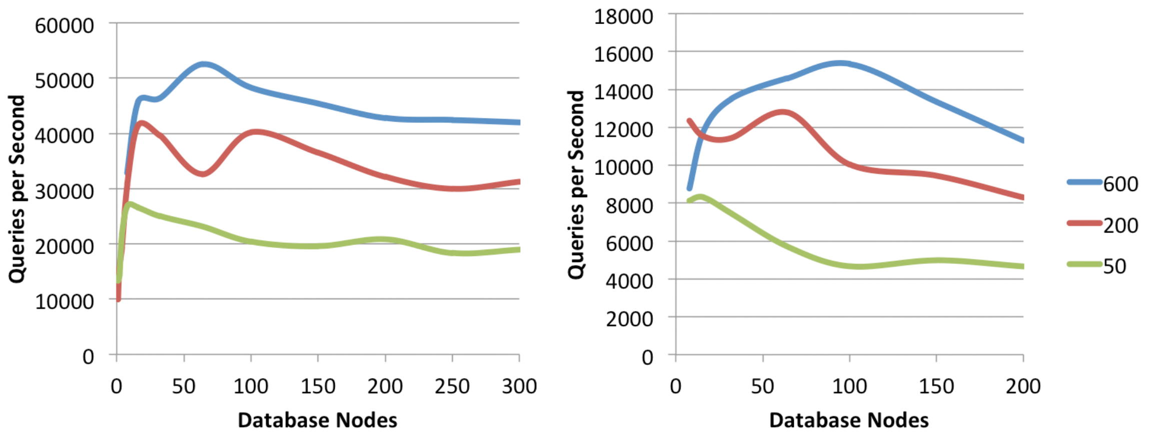
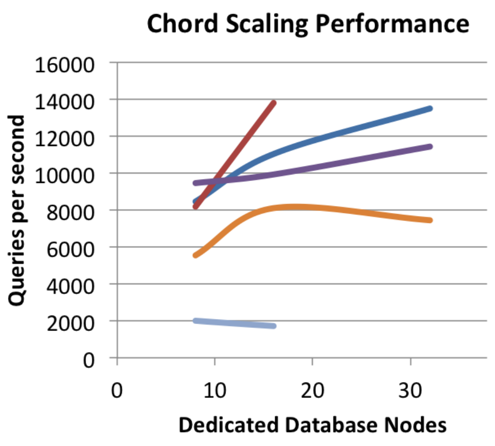


Figure 14: Query frequency performances, 30 computers. Fully connected network to the left, and Chord to the right.

In Figure 14, the number of computers has been increased to 30. Again, overall performance has improved for both Chord and FCN.

Scaling Performance It seems clear that performance improves as machines are added. Figures 15 and 16 should clarify this even further. Here, each node runs on a dedicated machine, and can therefore take maximum advantage of the available hardware.



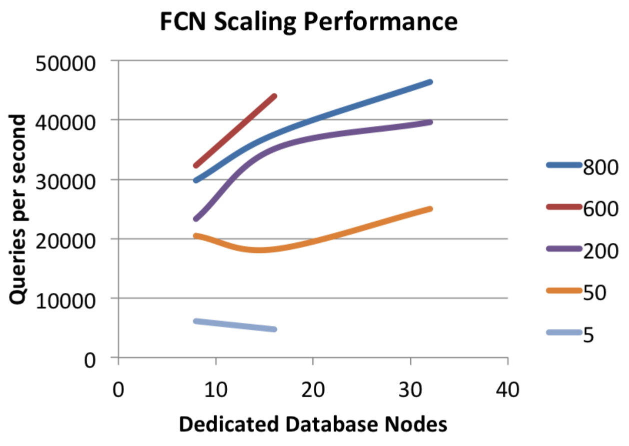


Figure 15: Scaling performance of the Chord implementation.

Figure 16: Scaling performance of the FCN implementation.

It can be seen that a doubling of the number of database nodes represents roughly a 30% increase in performance when testing against 800 clients. The relative scaling performance seems quite similar between the two implementations.

4.5.4 Results: Load Distribution

During each performance test, the test bed also measured the distribution of data elements over all database nodes and calculated the standard deviation. This section will show the standard deviation averaged over all tests, including both lookup implementations. This is reasonable, as they divide the key space according to the same specification, and deviations in the distributions are expected to be similar.

Since the nodes are assigned a random position on the ring, different nodes will be responsible for differently sized sections of the key space. This implies that the work distribution between nodes may not be even. The standard deviation of the number of data elements at each node is a measurement of this. If the standard deviation is high, it implies that some nodes are being used much more than other nodes. This in turn limits the gains of parallelism.

The amount of database nodes is represented on the X-axis, and the standard deviation on the Y-axis.

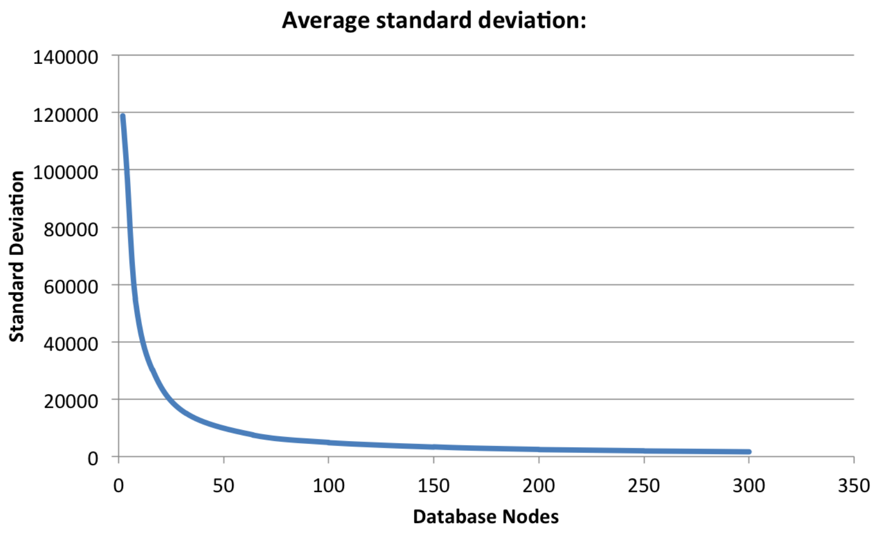


Figure 17: Average standard deviation of all tests combined.

When having just one node, there can be no distribution. Thus the measurement begins at two nodes. This is the highest point at 120 000; after this, the curve decreases exponentially.

One might think this equals an actual improvement in terms of distribution as the number of nodes increases, but this is not the case. The graph shows a decrease in workload per node, but this does not guarantee workload is evenly distributed across nodes.

5 Discussion

Two aspects will be central to this discussion: the comparison between Chord and FCN, and the evaluation of whether the database lives up to its specification. The test results from the previous chapter will be analyzed with these questions in mind.

One of the main goals with the database was that it should be able to handle a high volume of concurrent queries. Looking at the results, performance can be said to be high in this regard, with a maximum throughput of roughly 50 000 queries per second. However it is difficult to fully put this number into context, since there were many variables that could not be controlled for, for example when it comes to hardware.

The following subsections will analyze the test results from the Results chapter in greater detail, focusing on aspects central to the database implementation, and comparing Chord and FCN along the way. Finally, Section 5.4 will summarize this comparison and conclude which scenarios would do the different implementations most justice.

5.1 Lookup Speed

One of the most obvious takeaways from the charts in the Results chapter should be that the Chord algorithm runs a lot slower than FCN on the tested configurations. Even when scaling the number of machines, this performance difference remains relatively constant.

It is reasonable to attribute this to the different amount of network traffic generated by the implementations. Chord has a distributed lookup function, which queries up to log N other nodes over the network in order to find the target node. FCN keeps track of all other nodes, and does the lookup internally as a linear search in a list. No network messages are sent during this process. Chord is much slower in practice simply due to the high cost of network messages. During lookup, the only network traffic generated by FCN is the transfer of the actual data element.

This would also explain the differing peak points of the curves in Figure 11. When only one node exists, Chord does not have to send any messages which makes performance more comparable to that of the FCN. After this, with the exception of a local maximum point at ten nodes, the graph steadily declines. This is likely related to the overhead introduced by message passing.

5.2 Scaling

In order to analyze how well the database scales, one must first define what "scaling well" actually means. In this case, it is taken to mean that after some quantity has been increased, the database performs better than it did before the increase. This does not take into account whether the database can operate efficiently as it is growing and the nodes must update their routing tables. The implications of this will be discussed in the final subsection below. Before that, the impact of scaling different parameters (clients, machines, and database nodes) will be analyzed. Chord and FCN will be compared here as well.

5.2.1 Test Clients

In the context of these tests, one test client symbolizes one simultaneous request to the database, i.e. having 200 clients means that the database has to handle 200 concurrent requests.

All performance charts show the same pattern: increasing the number of clients leads to performance improvement. This can be explained by a number of reasons, all linked to the notion of parallelism, which is central to the foundation of this project.

The greatest advantage to increasing the number of clients is that the number of nodes in use will increase as well. This means that the total work performed will be distributed over a greater number of machines, resulting in a smaller number of queries per CPU unit. In a non-parallelized system, this would no longer be an advantage when the number of clients exceeds the available resources. However, making use of parallelism, more clients can be served at the same time, greatly increasing the capacity of query handling. As a concrete example, consider the fact that when the number of clients is 800 on 20 nodes, there still has been no sign of decreased performance, despite there being 40 times as many clients as nodes. This pattern can be observed on all performance charts.

Looking at figures 9 and 10, and considering the data points at which the number of clients has been measured, the improvement appears to be roughly linear on a logarithmic scale. In other words, throughout the charts, doubling the number of clients has a fairly consistent outcome in terms of performance betterment. However, it can be assumed that this increase would not stay constant forever, due to hardware limits on parallelism.

While these results are likely not up to industry standards, and there was no opportunity to test for more than 800 clients, the fact that performance steadily improved as clients increased still has to be viewed as a success.

5.2.2 Nodes and Machines

When scaling the amount of database nodes, performance peaks quite early and goes down steadily after that point. However, as the amount of machines increase the peak shifts to the right. To see an example of this, compare Figures 10 and 13. A possible explanation for this is that this configuration makes optimal use of the available CPU power. In Figure 10, where the test is run on 10 computers, the curve peaks around 20 nodes; when the number of computers is doubled (Figure 13) the peak is around 40. Since each machine runs on a dual core processor, these peaks clearly correlate closely with the number of available cores. In other words, while the database is not scalable in terms of database nodes, it does look a lot better when considering the number of machines as well. This can be seen clearly in charts 15 and 16, where each node runs on a dedicated machine.

It would have been interesting to further increase the number of computers (and therefore the number of cores) and see whether this pattern would have persisted. If so, it would be clear the hardware limitations posed a major drawback. Since the optimal configuration appears to be one node per core, it is very possible that scaling the number of cores would have had a great impact on performance.

It should be noted that while the curves in Figures 15 and 16 do see a positive progression, it is dubious whether this can truly be referred to as "scalable". In a perfect scenario, when doubling the number of machines, performance should come close to doubling as well. In this case, the increase is 30%. Part of this is likely due to overhead. This is especially true for the Chord implementation. While its relative scaling performance is comparable to that of the FCN, in actual numbers it lags far behind, handling about a third as many queries per second as FCN. Taking into account the aspects discussed in Section 5.1, it can be assumed that Chord never reaches the point where hardware limitations dictate its performance, but is rather held back by implementation-specific overhead.

In any case, a 30% increase is a fairly good result considering the simplicity of the database implementation.

5.2.3 Dynamic Scaling

As mentioned at the end of Section 4.5.2, it was not possible to test for dynamic scaling performance, in other words how efficiently the database can operate during the time when new nodes are being added. However, despite there being no empirical results, some patterns could still be observed by simulating the algorithms in the visualization application. Analysis of the algorithms' intrinsic properties also helped in this regard.

Dynamic scalability comes down to the cost of updating the routing information at existing nodes as new nodes join the network. Ideally, this cost should be low, and grow much more slowly than the network size. In FCN, where all nodes must be notified of changes, one joining node yields N network messages in order to update other nodes. The Chord algorithm, using the aggressive join strategy, requires only $M \log N$ such operations, where M is the constant size of each node's finger table [7]. Sending messages over the network is expensive, so naturally the implementation which sends the fewest will be fastest in practice. As long as N is smaller than $M \log N$, FCN will have the advantage; however Chord grows at a much slower rate, which makes it inherently more scalable. In very large networks, using a FCN-like implementation is likely not reasonable.

When using the stabilized join strategy in Chord, finger table updates are spread out over time at a constant rate, regardless of whether network changes are occuring. Since a node always, at least, knows about its neighbors, it can always look up any other node even if its routing information has not yet been updated. It might take longer, but data will always be available. FCN, in contrast, does not spread out updates, increasing the simultaneous stress on the system when many nodes join the network in a short amount of time. Also, there will be a short period for each node where its routing information is incomplete.

In short, whether Chord uses aggressive or stabilized join, it is clearly more dynamically scalable than FCN.

5.3 Load Distribution

Due to the way Chord and FCN randomly place the nodes on the logical ring, the distribution of traffic to each node will not necessarily be the same. Ideally, this inequality should diminish as the number of nodes increases, however in these implementations this has not been the case. While the standard deviation from the expected load on each node decreases in absolute numbers, so does the expected value. In fact, they both go down at roughly the same rate, meaning that relatively, standard deviation stays about the same no matter how many nodes are added. Obviously, this is not optimal. Implementing the more sophisticated hashing method described in section 2.7, where each node is represented on several places on the ring, would likely have led to considerable improvement in this area. Another solution would be to manually assign positions to the nodes. This would definitely assure fair distribution. However, this would compromise the dynamic scalability of the database.

5.4 Chord or FCN?

As noted in Section 5.1, FCN clearly performed better in terms of throughput. However it should be noted that all tests were carried out in a small and very stable environment. The advantages of the Chord algorithm would be more visible in a larger network, and when database nodes join and leave the network more often. The join operation sends messages to other nodes in order to update their routing tables. As described in Section 5.2.3, Chord will send significantly less such messages, assuming that network size is larger than $M \log N$. Due to the high costs of network traffic, Chord would likely have been advantageous in a large and dynamic environment. However, this has not been part of the experiments.

For an intended application of the database, for example a backend to a web service, it can be expected that the number of nodes is mostly constant and that joins and leaves occur infrequently. In this case, it does not make sense to optimize for performance during joins and leaves. Rather, what should be optimized for is fast lookup performance in an already stable network. In other words, the FCN option is the most reasonable choice. The pure Chord model would be more suitable for a large, loosely connected network, such as a peer-to-peer file sharing system. However one can also imagine cases where the advantages of Chord come to use even in a fully controlled environment. For example, a database may need to be quickly scaled in order to meet quick changes in demand. A scenario where this may apply is when resources are dynamically allocated to different time zones during the course of a day.

6 Conclusion

The fully connected network turned out to be more appropriate in the scenario envisioned for this database. A database like this is usually run on a relatively low and constant number of nodes, which means that Chord's advantages are not put to use. Instead, Chord is hampered by the network messages necessary for lookup.

As for the performance of the database itself, it did turn out to handle concurrent queries to some satisfaction, performance still improving even as the number of clients rose to 40 times the number of database nodes. While there is likely, due to hardware, some upper limit to this improvement, no ceiling was discernible in our tests.

The database also scales fairly well, considering the simplicity of its implementation.

Load distribution over the nodes in the network, on the other hand, did not turn out to be satisfactory. A more sophisticated Chord implementation is likely to have improved this considerably.

References

[1] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[2] 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. In *IN PROC. SOSP*, pages 205–220, 2007.

[3] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12, 2011.

[4] Max Goff. *Network Distributed Computing: Fitscapes and Fallacies*. Prentice Hall Professional Technical Reference, 2003.

[5] M. Ben-Ari. *Principles of concurrent and distributed programming, second edition*. Addison-Wesley, second edition, 2006.

[6] Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[7] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[8] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001.

[10] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[11] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. A scalability benchmark suite for erlang/otp. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, Erlang '12, pages 33–42, New York, NY, USA, 2012. ACM.

[12] Apache couchdb. `http://couchdb.apache.org/`.

[13] Riak website. `http://basho.com/products/riak-overview/`.

[14] Håkan Mattsson, Hans Nilsson, Claes Wikström, and Ericsson Telecom Ab. Mnesia - a distributed robust dbms for telecommunications applications. In *In Practical Applications of Declarative Languages: Proceedings of the PADL'1999 Symposium, G. Gupta, Ed. Number 1551 in LNCS*, pages 152–163. Springer, 1999.

[15] openframeworks website. `http://www.openframeworks.cc/`.

[16] lib cinder website. `http://www.libcinder.org/`.

[17] B. Fry. *Visualizing Data: Exploring and Explaining Data with the Processing Environment*. O'Reilly Media, 2008.

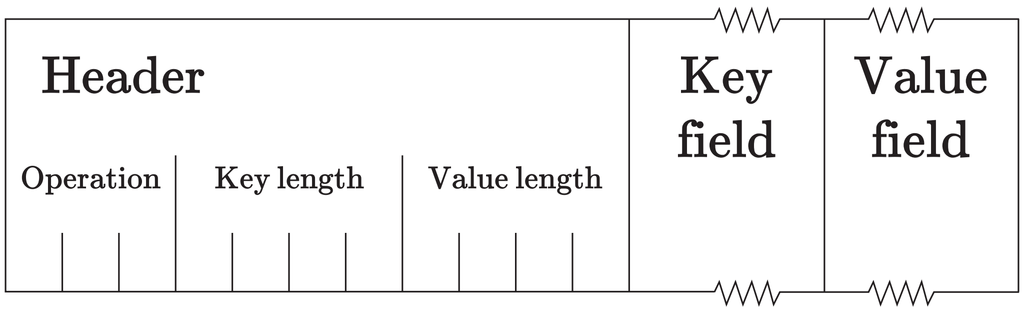


Figure 18: Message structure.

A Technical Details

A.1 Database Client Protocol

The protocol used to communicate between any one of the database nodes and a client application is an application layer protocol that supports three basic operations: Put, get and remove. The message from the client to the server consists of a header, a key field, and optionally a value fields. The reply from the server to the client follows the same format.

The first three bytes of the header is a text string used to distinguish between the operations, and between the server replies. The next four bytes is an unsigned integer representing the length of the key field. Likewise, the following four bytes represents the length of the value field. Figure 18 and the table below illustrates this.

Header			Key field	Value field	Function	Possible replies
Operation	Key	Value				
GET	Variable	Zero	Requested key	Empty	Send a ”get” query	GOK, GER
PUT	Variable	Variable	Key to insert	Value to insert	Send a ”put” command	POK, PER
DEL	Variable	Zero	Key to remove	Empty	Send a ”delete” command	DOK, DER
GOK	Variable	Variable	Requested key	Requested value	Reply to a get query	
POK	Variable	Zero	Requested key	Empty	Normal reply to a put command	
DOK	Variable	Zero	Requested key	Empty	Normal reply to a delete command	
GER	Variable	Zero	Requested key	Empty	Error reply to a get query	
PER	Variable	Zero	Requested key	Empty	Error reply to a put command	
DER	Variable	Zero	Requested key	Empty	Error reply to a delete command	

B Project Planning

Here follows some deliberation on some soft parts of the project.

Initial Planning This project was conceived by Kristian and Peter.

Once the project was started, there was a considerable amount of work to decide how to further define the project. The final shape of the project and the specific tasks that we wanted to complete was not clear at the beginning. There were discussions about what kind of database we were going to build, and what aspects of this wide field we wanted to focus on. Eventually we reached a consensus to focus on distribution, and the relatively simple chord algorithm. This decision process was unfortunately time consuming and delayed the actual project start. We were also somewhat conflicted by the initial plans to "just" build a database, and the academic expectations that became clear after the project had begun.

The visualization and the testing environment were not initially a part of the project, but was added later for the reasons given in the report.

Work Distribution We did not have any formal work assignments. However, as the project advanced informal groups were formed on different areas based on personal interest. This worked out fairly well but might also have given some lack of responsibility. The actual personal contributions are listed separately.

We decided early on to have a rotating project manager and a rotating secretary. These roles were changed every two weeks. This was chosen because no one wanted to put substantial focus on project management, but at the same time there was a need for these roles to be filled.

Decision Making All decisions in the group were taken in a democratic fashion. Although we had temporary project managers, the actual decisions were taken by the group, sometimes by vote.

Time Planning Once the initial planning phase was complete, there was some time set apart for literature studies. Once the actual development began, the work was divided into seven two-week iterations, with a "bi-weekly build" session at each deadline.

Report Writing The report sections were divided between the group members in a similar fashion as the development tasks. Most parts have several contributing writers, and the individual contributions are listed separately. For the actual document we used LaTeX, and the cloud service ShareLatex that allows collaborative editing.

Resource Management All code for the project was managed by a Git repository at Google Code. It is available at http://code.google.com/p/group13bachchalmers/

C Individual Contributions

C.1 Report Contributions

A = Main Author
CA = Contributing Author
E = Editing

Sections:	Joel	Julia	Kristian	Olle	Peter
Abstract	CA	A			CA
1. Introduction			A		
1.1 Purpose	A	A			
2. Technical Background					
2.1 Database		CA	A		
2.2 Distribution		CA	A		A
2.3 Distributed Database		CA			A
2.4 Key-value Store	CA	A			
2.5 The Concurrent Paradigm		A			
2.6 Erlang		E			A
2.7 The Chord Protocol	A	E		A	
2.8 Fully Connected Network (FCN)	A	CA		A	
3. Method					
3.1 Main Project Parts	CA	A			A
3.2 Experiments		A			
3.3 Limitations	CA	CA	A		E
3.4 Development Tools		A	A		CA
4. Results					
4.1 Database		A			A
4.1.1-4.1.4,4.1.7		A		E	A
4.1.5 Chord Implementation	A	E		A	
4.1.6 FCN Implementation	CA			A	
4.2 Test Bed		CA	A		CA
4.3 Visualization	A	E			
4.4 Utilities		CA			CA
4.5 Experiments	CA	A	CA		A
5. Discussion					
5.1 Lookup Speed	CA	A	CA	CA	CA
5.2 Scaling	CA	A	CA	CA	CA
5.3 Load Distribution	CA	A	CA	CA	CA
5.4 Chord or FCN?	CA	A	CA	CA	CA
6. Conclusion	CA	A	CA	CA	CA
Appendix A. Client Protocol				A	
Appendix B. Project Planning	A		A		
Illustrations	CA		A		

C.2 Code Contributions

A = Main Author
CA = Contributing Author
E = Editing

Modules:	Joel	Julia	Kristian	Olle	Peter
Database					
Node Module				E	A
Store Module					A
Network Module, FCN				E	A
Network Module, Chord	CA			A	
Backend Module					A
Socket Module					A
Socket Library			A	A	
Client			A		A
GUI				CA	A
Visualization support			CA	A	A
Test Bed					
Master		A	A		CA
Slave			A		
Tests		A			A
Automatization Tools		CA	CA		A
Visualization					
Graphics	A		E	CA	
Logics	A			CA	
Erlang Interface			A	A	A

D Test Results

This appendix shows the test results yielded form the test bed. Beginning backwards, FCN and Chord are compared with 30, 20 and 10 computers on the following pages.

Table 1: FCN running on thirty computers.

Slaves	**Nodes**	**Queries/Sec**	**Deviations**
50	1	13317.19	0.0
50	2	14331.49	120822.33
50	4	21660.94	102186.75
50	8	26870.08	61677.57
50	16	26624.73	32250.05
50	32	25019.97	14943.07
50	64	23157.65	7561.92
50	100	20404.17	5684.1
50	150	19564.93	3494.95
50	200	20847.25	2643.92
50	250	18329.45	2008.73
50	300	18970.37	1657.37
200	1	9967.25	0.0
200	2	15836.59	98805.67
200	4	19452.58	124817.69
200	8	30850.71	70736.56
200	16	41439.19	30122.33
200	32	39620.02	14246.74
200	64	32648.41	7103.09
200	100	40182.43	5060.5
200	150	36513.75	3458.34
200	200	32127.01	2737.04
200	250	29967.85	2013.72
200	300	31262.51	1659.51
600	8	32854.19	55078.0
600	16	45708.77	27579.36
600	32	46389.86	15229.68
600	64	52565.64	7755.33
600	100	48301.91	4935.67
600	150	45422.34	3323.87
600	200	42773.68	2459.83
600	250	42461.01	5324.94
600	300	42000.96	1689.74

Table 2: Chord running on thirty computers.

Slaves	**Nodes**	**Queries/Sec**	**Deviations**
50	8	8136.07	50760.96
50	16	8309.59	26087.89
50	32	7450.88	17307.35
50	64	5715.12	8273.95
50	100	4662.93	5213.96
50	150	4979.48	3503.27
50	200	4656.89	2387.68
200	8	12344.57	33166.46
200	16	11522.97	31669.72
200	32	11439.43	16513.47
200	64	12790.96	8258.23
200	100	10034.5	5312.52
200	150	9446.48	3393.77
200	200	8299.09	2603.2
600	8	8772.99	51520.4
600	16	11816.34	24701.32
600	32	13503.39	13175.02
600	64	14594.38	7736.08
600	100	15357.12	4986.87
600	150	13347.87	3338.66
600	200	11300.08	2563.44

Table 3: FCN running on twenty computers.

Slaves	Nodes	Queries/Sec	Deviations
5	1	7559.37	0.0
5	2	6135.33	146048.0
5	4	5014.52	105590.26
5	8	5022.59	49046.41
5	16	4738.15	27012.28
5	32	4644.49	15073.55
5	64	4555.36	7660.09
5	100	4227.95	5474.89
5	150	4334.11	3537.19
5	200	3961.52	2699.87
50	1	11198.97	0.0
50	2	16810.84	92896.33
50	4	18363.69	103457.12
50	8	24307.44	48838.17
50	16	18215.37	29933.74
50	32	22562.14	16353.65
50	64	19250.01	7974.73
50	100	18441.65	4579.55
50	150	17973.43	3290.54
50	200	17142.43	2381.36
200	1	10333.92	0.0
200	2	16304.97	41500.67
200	4	20623.27	88776.77
200	8	28363.9	52923.56
200	16	35118.43	34506.09
200	32	33493.07	13870.4
200	64	34075.28	6953.7
200	100	31301.31	4450.09
200	150	29671.5	3141.17
200	200	27867.79	2379.39
400	1	9956.28	0.0
400	2	14503.77	127137.67
400	4	19004.7	84767.59
400	8	30522.33	58387.56
400	16	39862.18	24634.42
400	32	43068.25	14531.12
400	64	36408.1	7546.1
400	100	38041.51	5213.85
400	150	37782.12	3606.74
400	200	35636.73	2698.52
600	8	29457.46	57690.42
600	16	43997.64	26000.4
600	32	47899.86	15722.33
600	64	39995.54	7356.06
600	100	38852.21	4970.49
600	150	37393.97	3266.92
600	200	37704.93	2415.87
800	8	30376.39	61623.93
800	16	37534.96	35438.46
800	32	47929.87	14447.17
800	64	48607.93	8136.19
800	100	42242.26	4747.28
800	150	42346.87	2977.08
800	200	40034.49	2418.64

Table 4: Chord running on twenty computers.

Slaves	Nodes	Queries/Sec	Deviations
5	1	8773.61	0.0
5	2	3141.47	206426.33
5	4	2512.98	103665.12
5	8	1770.48	68057.24
5	16	1721.71	29897.8
5	32	1461.31	14711.31
5	64	1323.53	7605.36
5	100	1200.1	5496.73
5	150	1055.94	3512.76
5	200	1009.5	2717.83
50	1	9123.76	0.0
50	2	7640.54	108980.33
50	4	8297.01	90410.15
50	8	8487.2	52264.41
50	16	8108.08	28668.71
50	32	5801.49	16471.32
50	64	4783.86	7421.43
50	100	4462.06	4775.58
50	150	4036.02	3166.61
50	200	3842.72	2425.87
200	1	8951.83	0.0
200	2	7429.89	105119.0
200	4	8075.97	70676.56
200	8	10249.63	51709.82
200	16	9931.32	33610.03
200	32	10156.15	14234.51
200	64	9863.72	8729.96
200	100	8344.07	5599.69
200	150	7074.95	3688.96
200	200	6691.03	2717.0
600	8	10533.62	55998.43
600	16	13803.75	35251.23
600	32	11003.88	17885.63
600	64	11157.36	7216.08
600	100	10543.43	4831.36
600	150	9571.49	3396.21
600	200	8552.97	2678.87
800	16	11916.77	28191.01
800	32	11499.83	15417.36
800	64	11406.21	7174.74
800	100	85390.79	3334.89
800	150	9244.09	3429.51
800	200	90799.06	1722.33
800	16	11039.47	29458.38
800	32	13104.54	15765.02
800	64	11855.9	7587.55
800	100	11556.04	5078.41
800	150	9628.54	3649.24
800	200	11406.25	2509.59

Table 5: FCN running on ten computers.

Slaves	Nodes	Queries/Sec	Deviations
5	1	9177.14	0.0
5	2	6202.8	124748.33
5	4	5889.31	111702.58
5	8	6093.25	52808.81
5	16	5802.81	31143.55
5	32	6356.62	13022.66
5	64	5633.76	7658.74
5	100	5758.16	4812.61
50	1	11056.57	0.0
50	2	13646.48	34637.33
50	4	15123.45	85098.02
50	8	20439.64	37491.08
50	16	16595.8	33228.02
50	32	17093.18	14957.01
50	64	13994.26	7480.38
50	100	13629.03	4990.87
100	1	10955.0	0.0
100	2	15299.09	133002.0
100	4	19527.15	125219.96
100	8	25525.89	68129.25
100	16	24190.09	26768.54
100	32	20803.23	16864.87
100	64	17563.56	7443.86
100	100	16191.96	4987.25
200	1	10685.44	0.0
200	2	15631.78	80735.0
200	4	19457.62	75179.06
200	8	23309.29	66527.58
200	16	29902.6	35483.8
200	32	28847.51	13361.64
200	64	21553.75	7484.73
200	100	19324.09	4808.27
400	1	10576.52	0.0
400	2	15069.86	118662.0
400	4	18247.61	102457.12
400	8	24766.23	40648.47
400	16	31803.96	27791.53
400	32	31844.25	14492.37
400	64	26135.42	8078.72
400	100	23124.44	4810.94
600	2	12888.61	99134.67
600	4	19396.14	88107.5
600	8	32319.23	40882.5
600	16	35235.05	26869.99
600	32	35466.08	16755.66
600	64	31371.9	7392.25
600	100	27171.14	5100.69
800	4	17680.5	121846.57
800	8	29794.83	49773.1
800	16	36062.52	32227.01
800	32	32996.52	12972.55
800	64	31542.6	7620.91
800	100	29481.3	5094.75

Table 6: Chord running on ten computers.

Slaves	Nodes	Queries/Sec	Deviations
5	1	9041.04	0.0
5	2	2475.9	123636.33
5	4	2418.59	118144.38
5	8	1991.36	70686.34
5	16	1860.39	28432.86
5	32	1494.87	12709.03
5	64	1288.66	7873.74
5	100	1151.62	4861.0
50	1	9805.09	0.0
50	2	6042.4	134600.33
50	4	6107.42	96133.42
50	8	5547.98	58613.26
50	16	5040.27	31865.62
50	32	4483.05	16611.71
50	64	3965.12	7564.55
50	100	3485.24	4943.42
100	1	9461.79	0.0
100	2	7716.42	141521.0
100	4	7412.95	76314.2
100	8	10382.22	49265.91
100	16	6199.03	31530.88
100	32	5876.33	15912.56
100	64	4619.3	7370.33
100	100	4078.89	4942.45
200	1	9316.94	0.0
200	2	7921.58	191961.33
200	4	8528.41	95412.84
200	8	9461.45	54728.56
200	16	8161.05	30095.58
200	32	7413.7	14561.36
200	64	5712.88	7290.45
200	100	4698.98	4950.3
400	4	9524.85	106936.93
400	8	9937.83	57967.84
400	16	8950.41	32097.44
400	32	8223.01	14803.9
400	64	6304.3	7579.86
400	100	5322.85	4951.48
600	4	7382.48	95000.61
600	8	8187.9	51554.87
600	16	9305.37	31721.54
600	32	8643.97	17104.31
600	64	7526.72	7285.43
600	100	5964.85	4941.4
800	4	8031.14	109850.29
800	8	8459.57	46569.37
800	16	9818.89	27863.9
800	32	8565.74	15230.26
800	64	7355.95	8188.49
800	100	7143.02	4928.84