Raft & RethinkDB: A thorough Examination

Nikos Koufos

Antonis Matakos

Thanos Pappas

January 21, 2017

Contents

1 Raft **2**
1.1 Introduction 2
1.2 Consensus algorithm 2
1.3 Leader election 3
1.4 Log replication 3
1.5 Safety 4
1.6 Reconfiguration 5

2 Raft vs Paxos **7**
2.1 Intro 7
2.2 Purposes 7
2.3 Differences 7
2.4 Similarities 8
2.5 Conclusion 8

3 RethinkDB - Basic Experience **9**
3.1 Features 9
3.2 When is RethinkDB not a good choice? 10
3.3 Practical considerations 10

4 System Comparison **12**
4.1 RethinkDB vs Niobe vs Petal vs SMART 12
4.2 RethinkDB vs Cassandra vs HBase 13

5 Limitations **16**

6 Technical Implementation **17**
6.1 Overview 17
6.2 RethinkDB's Raft Implementation 18

7 Experiments **20**
7.1 Read Performance 20
7.2 Write Performance 23
7.3 RAM Usage Statistics 23
7.4 Availability 25

8 Conclusion **27**

1 Raft

1.1 Introduction

Raft[1] is a consensus algorithm for managing a replicated log. It produces a result equivalent to Paxos[2] and is as efficient, but its structure is quite different. Raft was designed to provide a better understandability and a foundation for building practical systems. To achieve that, Raft separates the key elements of consensus (leader election, log replication, safety) to reduce the number of possible states.

Raft is similar in many ways to existing consensus algorithms, but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms, which simplifies the replication log management and makes Raft more understandable.
- **Leader election:** Raft applies randomized timers to the heartbeat process to achieve leader election. This technique adds a small amount of overhead in exchange for rapid and simple conflict resolution.
- **Reconfiguration:** In order for the cluster to continue operating normally during the membership changes, Raft uses a joint consensus approach where the majorities of two different configurations overlap during transitions.

1.2 Consensus algorithm

Raft implements consensus by first electing a distinguished leader. The leader accepts log entries from clients, while having complete responsibility for managing the replicated log. Given the leader-based approach, Raft decomposes the consensus problem into three relatively independent sub-problems (leader election, log replication, safety) which will be discussed subsequently.

A raft cluster which contains n servers, can tolerate up to $\frac{n-1}{2}$ failures, where n is typically odd. At any given time each node is in one of three states: leader, follower, or candidate. Under normal circumstances there is only one leader and all of the other servers are followers, which redirect client requests to the leader. Followers issue no requests on their own but simply respond to requests from leaders and candidates. The candidate state is used to elect a new leader which will be described in the following sections.

Raft divides time into terms of arbitrary length, which is equal to Paxos epochs. Terms are numbered with consecutive integers. Each term begins with a leader election. If a candidate wins the election, then it serves as leader for the rest of the term. In the case of a split vote between candidates, the term will end with no leader, as a result, a new term (with a new election) will follow. Raft ensures that there is at most one leader in a given term. Each server stores a current term number, which increases over time. If a candidate or leader discovers that its term is out of date, it immediately reverts to follower state.

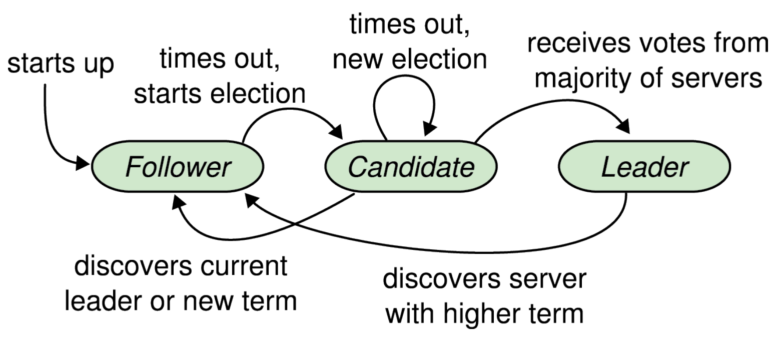
starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 1.1: Server states. Followers only respond to requests from other servers. If a follower receives no communication it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

1.3 Leader election

Upon system initiation, servers begin as followers. Leaders send periodic heartbeats to all followers in order to maintain their authority. If a follower receives no communication over a period of time, called the election timeout, then it assumes there is no viable leader and begins an election to choose a new leader. To begin an election, a follower increments its current term and transitions to candidate state. It then votes for itself and requests votes from the other servers in the cluster. This might lead to three possible scenarios.

(i) **The candidate wins the election:** A candidate becomes leader if it receives votes from a majority of the servers. Each server will vote for the candidate whose request was received first. The majority rule ensures that at most one candidate can win the election for a particular term. Once a candidate wins an election, it becomes leader and then sends heartbeat messages to all other servers to establish its authority and prevent new elections.

(ii) **Another server establishes itself as leader:** While waiting for votes, a candidate may receive a message from another server claiming to be leader. If the leader's term is larger or equal to the candidate's current term, then the candidate recognizes the server as the legitimate leader and then returns to follower state. Otherwise, the candidate rejects the message and continues in candidate state.

(iii) **A period of time goes by with no winner:** If many followers become candidates at the same time, it is possible that no candidate obtains a majority (split vote). In this case, each candidate will time out and start a new election by incrementing its term. To prevent the indefinite repetition of split voting, Raft uses randomized timeouts chosen from a time interval.

1.4 Log replication

The leader appends the client's command to its log as a new entry, then sends it to the other servers to replicate the entry. When the entry has been safely replicated (a majority of servers have replicated it), the leader applies the entry to its state machine and returns the result of that execution to the client. If followers crash or run slowly, the leader retries to send the command indefinitely until all followers eventually store all log entries. The leader makes a decision whether it is safe to apply a log entry to the state machines (committed).

Raft satisfies the following properties, which together constitute the Log Matching Property:

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

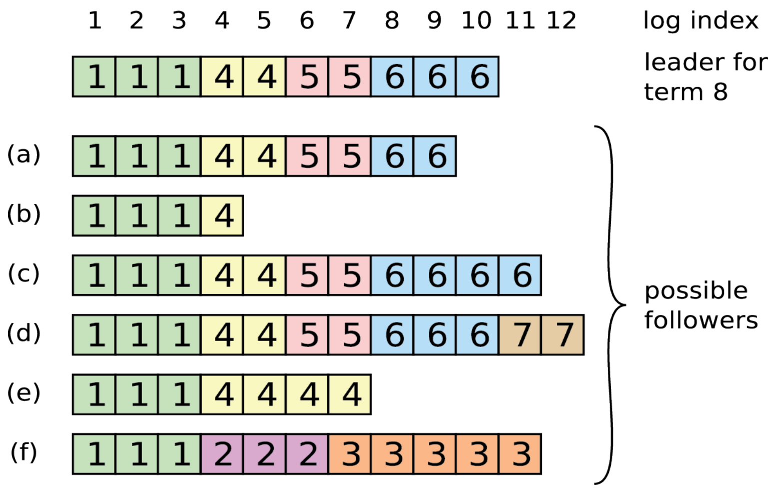

Figure 1.2: When the leader at the top comes to power, it is possible that any of scenarios (a-f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a-b), may have extra uncommitted entries (c-d), or both (e-f). For example, scenario(f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

The leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log. The leader maintains a nextIndex for each follower, which is the index of the next log entry the leader will send to that follower.

As the log increases in size, it occupies more space and takes more time to replay. Therefore it needs to discard unnecessary information. In order to achieve this Raft uses a simple snapshotting technique.

1.5 Safety

The Raft algorithm restricts which servers may be elected as leader. The restriction ensures that the leader for any given term contains all of the entries committed in previous terms.

Raft uses a simple approach which guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election, without the need to transfer those entries to the leader. This means that log entries only flow in one direction, from leaders to followers, and leaders never overwrite existing entries in their logs.

If a leader crashes before committing an entry, future leaders will attempt to finish replicating the entry. However, a leader cannot immediately conclude that an entry from a previous term is committed, once it is stored on a majority of servers.

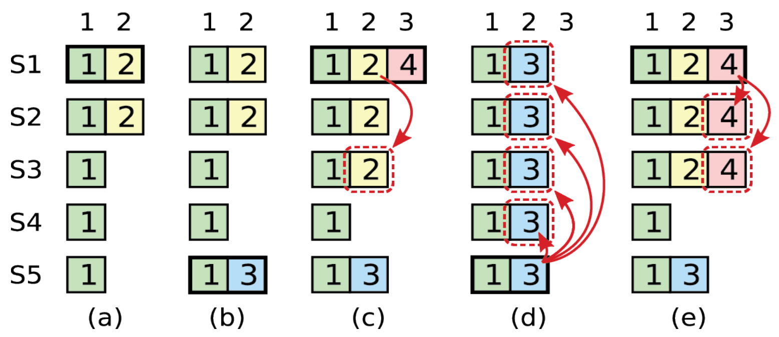

Figure 1.3: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

1.6 Reconfiguration

Raft was designed to make configuration changes online, without shutting down the whole system. To ensure safety, Raft uses a two-phase approach for reconfiguration. In the first step the system switches to a transitional configuration called joint consensus. Once the joint consensus has been committed, the system then transitions to the new configuration. The joint consensus combines both the old and new configurations:

- Log entries are replicated to all servers in both configurations.
- Any server from either configuration may serve as leader.
- Agreement (for elections and entry commitment) requires separate majorities from both the old and new configurations.

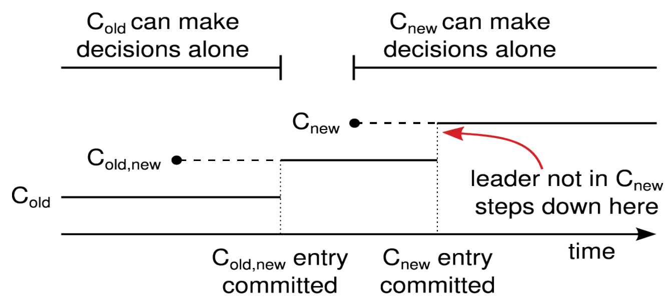
C_{old} can make decisions alone
C_{new} can make decisions alone
C_{new}
$C_{old,new}$
C_{old}
leader not in C_{new} steps down here
time
$C_{old,new}$ entry committed
C_{new} entry committed

Figure 1.4: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{new,old}$ configuration entry in its log and commits it to $C_{old,new}$ (a majority of and a majority of C_{new}) Then it creates the entry and commits it to a majority of C_{new}. There is no point in time in which C_{old} and C_{new} can both make decisions independently.

The aforementioned properties allow the system to service clients' requests while performing reconfiguration. This approach has three main issues.

- **New servers may not initially store any log entries**
  If they are added in this state, it could take a while for them to catch up, which could lead to a period where committing new log entries might not be possible. In order to avoid that, Raft introduces an additional phase before the configuration change, in which the new servers join the cluster as non-voting members.

- **The leader may not be part of the new configuration**
  In this case, the leader returns to follower state once it has committed the C_{new} log entry. This means that there will be a period of time when the leader is managing a cluster of servers that does not include itself. It replicates log entries but excludes itself from majorities. The leader transition occurs when C_{new} is committed.

- **Removed servers can disrupt the cluster**
  These servers will not receive heartbeats, so they will time out and start new elections. They will then become candidates with new term numbers, and this will cause the current leader to revert to follower state. This will cause constant leader elections, resulting in poor availability. To prevent this problem, servers ignore the vote requests when they assume a viable leader.

2 | Raft vs Paxos

2.1 Intro

In this chapter we will thoroughly compare both Raft and Paxos and outline their respective advantages and drawbacks. Both are widely used distributed consensus algorithms, which can be described as the act of reaching agreement among a collection of machines cooperating to solve a problem. Paxos has historically dominated both academic and commercial discourse but due to its complexity a more understandable algorithm was needed to serve as a foundation for building more practical systems. Raft was designed to satisfy this need. Both protocols have different purposes depending on the application.

2.2 Purposes

The Paxos algorithm was designed for implementing an asynchronous, reliable and fault-tolerant distributed system. It made a largely academic contribution and was a big step towards establishing a theoretical foundation for achieving consensus in distributed systems. However, the actual requirements necessary to build a real system, including areas such as leader election, failure detection, and log management, are not present in the first Paxos specification. A subsequent paper by Lamport in 2001[3] does an excellent job of making the original protocol more accessible, and to some degree, proposes techniques for designing a replicated log.

On the other hand, Raft was designed to be easily understandable and to provide a solid foundation for building practical systems. It's fault-tolerance and performance are equivalent to Paxos. The main difference between the two protocols is that Raft can be decomposed into relatively independent sub-problems, while Paxos handles the various elements as one. Additionally, Raft introduced several new concepts in distributed consensus such as a mechanism for online system reconfiguration, which was described in section 1.6.

2.3 Differences

Despite both Paxos and Raft being distributed consensus protocols, they have several key differences in design and structure, which are described below:

- **Simplicity vs Complexity**
  Paxos was built to provide a reliable consensus algorithm for distributed systems. Many argue that its biggest drawback is its high complexity in spite of many attempts to make it more approachable. On the other hand, Raft was designed with simplicity and understandability as its main goal.
- **Decomposition**
  The main reason for Paxos' complexity, is that many of the challenges towards reaching consensus are not clearly defined. This issue is tackled by Raft, which has clear separation of its components, such as leader election and reconfiguration.

- **Role Separation**
  Adding to Paxos' complexity is the fact that the replica roles are incoherent. Specifically, a single node can simultaneously function as proposer, acceptor and listener. On the contrary, Raft has a more clear-cut distribution of roles, where a node can only be assigned one role (follower, candidate, leader).
- **Log Management**
  Raft manages and stores a sequence of log entries, and is centered around log management. This results in additional complexity to Raft's log management, compared to Paxos' single consensus value handling.
- **Client Interaction**
  In Paxos all nodes can accept requests from clients and start servicing them. However, due to Raft's strong leader property all request must be forwarded and handled by the leader.
- **Leader Election**
  In the first phase of (multi) Paxos, the replica that owns the proposal with the highest ID number is elected leader and can indefinitely propose until it is challenged by another node. This approach was implemented to eliminate the first phase. Unlike Paxos, Raft uses randomized timers to elect leader. To maintain leadership, Raft uses a heartbeat mechanism.
- **Reconfiguration**
  Multi-Paxos reconfiguration introduces a "change set of servers" command which determines the servers that will be in the new configuration. In addition, leader may propose α commands ahead of the last executed command. Raft's approach to reconfiguration uses a new "joint consensus" technique which requires two sets of majorities, one from the old set and one from the new one. This approach allows an online reconfiguration.

2.4 Similarities

Raft is unique in many ways compared to typical Paxos implementations, but despite those differences, both are grounded in a similar set of core principles. Although both protocols have a different implementation of leader election, they both acknowledge that the leader election process is imperative to ensure progress.

Moreover, both protocols can normally operate with $2n + 1$ servers and can tolerate up to n server failures. However, either protocol cannot guarantee availability in case of system partition, with no majority in any particular partition of the network.

Finally, despite the differences in the system reconfiguration, both algorithms ensure the system stays online during the reconfiguration process. This makes the system available to the clients.

2.5 Conclusion

We have highlighted the similarities and key differences between both Raft and Paxos which led us to the following conclusion. We believe that while Paxos was an important stepping stone in the evolution of consensus in distributed systems, Raft's simplicity and understandability make it both a better educational tool for academics and practical foundation for building distributed systems.

3 | RethinkDB - Basic Experience

RethinkDB[1] is a distributed, open-source, scalable JSON database built to service web applications in real time. It exhibits a novel access model which diverges from the traditional database architecture. Instead of polling for changes, RethinkDB allows the developers to continuously push updated query results to applications in real-time. Due to the replacement of the polling technique, which consumes a lot of CPU power and network I/O, RethinkDB dramatically reduces the time and effort necessary to build scalable real-time apps.

3.1 Features

RethinkDB provides a wide variety of features to the users, which are listed below.

- **Query Language Support(API)**
  In addition to being designed from scratch for real-time apps, RethinkDB offers a new query language(ReQL), which can be easily setup and learn. The language is fully documented.
- **User Interface**
  RethinkDB provides a web interface for managing a server cluster, controlling sharding and replication and running ReQL queries. In addition, you can perform administration tasks using ReQL commands.
- **Data Backup**
  Using the *dump* and *restore* commands, RethinkDB allows the user to take hot backups on a live cluster.
- **Importing Data**
  RethinkDB allows the user to import already existing data into RethinkDB databases, using either JSON files or CSV.
- **Data Migration**
  In RethinkDB, data is migrated to the new version automatically resulting in databases that are not backward-compatible.
- **Third-Party Compatibility**
  The system supports third-party tools, either on administration or on deployment level.
- **Sharding and Replication**
  RethinkDB allows you to shard and replicate each table of your cluster. Settings can be accessed from the web console, as previously mentioned.

[1] http://rethinkdb.com/

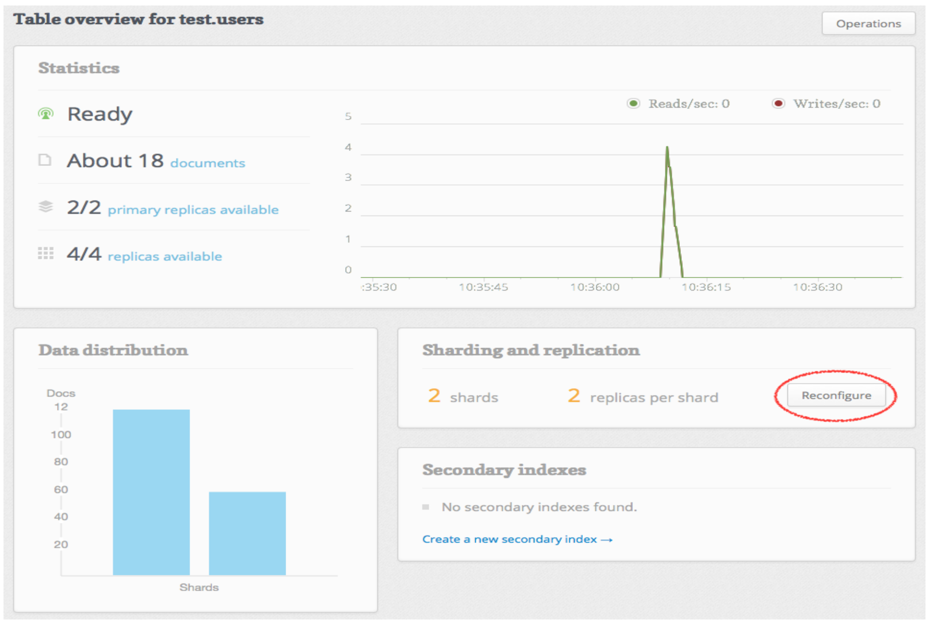

Figure 3.1: Web Console GUI

3.2 When is RethinkDB not a good choice?

RethinkDB is not always the perfect choice. We present some cases below.

- If the user needs full ACID support or strong schema enforcement then relational databases such as MySQL or PostgreSQL would be a better choice.
- For computationally intensive analytics a system like Hadoop or a column-oriented store like Vertica may be a better option.
- In some cases RethinkDB trades off write availability in favor of data consistency.

3.3 Practical considerations

There are some practical consideration regarding the use of RethinkDB.

- **Language support**
  Officially, drivers for Ruby, Python, and JavaScript/Node.js are provided, as well as many community-supported drivers such as C# and PHP.
- **SQL support**
  There is no sql support, however Rethink provides its own query language, which supports many of SQL's functions.
- **Atomicity**
  All write operations that are deterministic, involving a single document in RethinkDB, are guaranteed to be atomic. On the other hand, non deterministic operations(such as random values) cannot be performed atomically. In addition, multiple documents are not updated in an atomic fashion.
- **Query routing**
  Queries in RethinkDB are routed to the proper destination, independent of which node received the request.

- **Write durability**
  RethinkDB comes with strict write durability, which means that no write is ever acknowledged until it's safely committed to disk.

4 | System Comparison

In this chapter we will study the key elements of the selected systems one by one and highlight their differences and similarities compared to RethinkDB. To achieve that, we will make a distinction into to two other system categories. The first one includes state of the art distributed replication systems, such as SMART, Niobe and Petal, and the second category includes widely used database replication systems. We made this separation in order to achieve a clear distinction and due to the systems' common features.

4.1 RethinkDB vs Niobe vs Petal vs SMART

Quorum. Due to the fact that SMART [4], Niobe [5] and Petal [6] use Paxos as their GSM, a majority of servers who are able to communiciate with each other, is required in order to ensure progress. The same applies to our system, with the main difference that RethinkDB uses Raft as its GSM.

Regarding the heartbeat process SMART, Niobe and RethinkDB use the classic heartbeat technique where the leader exchanges messages periodically with other replicas. Petal uses a similar technique called liveness module.

Reads. All four systems implement different techniques regarding data reading. RethinkDB offers three possible read modes. The default way is the "single" value, which returns values that are in memory (but not necessarily written to disk) on the primary replica. There is also the "majority" mode which only returns values that are safely committed on disk on a majority of replicas. This requires sending a message to every replica on each read, so it is the slowest but most consistent(ensures safety even in the case of a system partition). Finally, the "outdated" mode will return values that are in memory on an arbitrarily-selected replica. This is the fastest but least consistent.

In Niobe, if a replica receives a Read request from a front-end, it first acquires a lock for reading, guaranteeing that no Update requests are being processed(see section 4.2 in John MacCormick et al). Then, the replica checks that it believes itself to be the primary, and if not, replies with an appropriate hint. Otherwise, it checks that it holds a read lease from every secondary it believes is alive. The leasing mechanism employs an extra parameter, which is the maximum amount of network or other delay that is permitted before we consider a fault has occurred.

Petal uses three of its modules(liveness, data access, virtual to physical translator) to service client reads. This process is separated into three main steps: (1) The virtual disk directory translates the client-supplied virtual disk identifier into a global map identifier. (2) The specified global map determines the server responsible for translating the given offset. (3) The physical map at the specified server translates the global map identifier and the offset to a physical disk and an offset within that disk.

Finally, in SMART when the client sends a read request, and after some communication delay the leader receives it and sends a proposal. The leader logs this proposal immediately, while the other replicas must wait until they receive it. Once the leader receives LOGGED messages from a quorum, it begins executing the request.

Recovery mechanisms. RethinkDB, SMART and Petal use different variations of the snapshooting technique to ensure data safety. On the other hand, Niobe uses a recovery mechanism based on a dirty-region log, and the GSM ensures data is consistent before clients are allowed access to the affected data item.

Reconfiguration. RethinkDB uses a slight variation of Raft's reconfiguration mechanism, which will be explained in a following section. Petal and SMART use a similar approach to reconfiguration based on the multi-paxos protocol. However, Petal's GSM also enforces two additional restrictions. First, requests are not pipelined($\alpha = 1$). Second, only two types of migration are allowed: adding one machine, or removing one machine. This ensures quorums from consecutive configurations always overlap. On the contrary, SMART does not require any overlap at all between consecutive configurations. Finally, Niobe uses a centralized GSM to store and update the system configuration, but does not specify the implementation of the GSM.

4.2 RethinkDB vs Cassandra vs HBase

In this section we will compare three commercially used distributed database systems. To start with, HBase[1] is wide-column store based on Apache Hadoop and on concepts of BigTable. Apache HBase is a NoSQL key/value store which runs on top of HDFS (Hadoop Distributed File System) and its operations run in real-time on its database rather than MapReduce jobs. HBase is partitioned to tables, and tables are further split into column families.

Cassandra[2] is Wide-column store based on ideas of BigTable and DynamoDB. Apache Cassandra is a leading NoSQL, distributed database management system which offers continuous availability, high scalability and performance, strong security, and operational simplicity.

Indexing. In RethinkDB when the user creates a table, they have the option of specifying the attribute that will serve as the primary key. When the user inserts a document into the table, if the document contains the primary key attribute, its value is used to index the document. Otherwise, a random unique ID is generated for the index automatically. The primary key of each document is used by RethinkDB to place the document into an appropriate shard, and index it within that shard using a B-Tree data structure. Querying documents by primary key is efficient, because the query can immediately be routed to the right shard and the document can be looked up in the B-Tree. RethinkDB also supports both secondary and compound indexes, as well as indexes that compute arbitrary expressions.

In HBase, data is stored in tables, which have rows and columns. A row in HBase consists of a row key and one or more columns with values associated with them. HBase uses a single index that is lexicographically sorted on the primary row key. Access to records, in any way other than through the primary row, requires scanning over potentially all the rows in the table to test them against your filter. Hbase does not natively support secondary indexes, but with the trigger mechanism applied on the input, can automatically keep a secondary index up-to-date, and therefore not to burden the application.

Cassandra is a partitioned row store, where rows are organized into tables with a required

[1]https://hbase.apache.org/
[2]http://cassandra.apache.org/

primary key. The first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the PK. Other columns may be indexed independent of the PK. Cassandra also provides a native indexing mechanism in secondary indexes. Secondary indexes work off of the columns values. The user has to declare a secondary index on a Column Family.

Cap theorem. In theoretical computer science, the CAP theorem also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees: (1) Consistency (all nodes see the same data at the same time). (2) Availability (a guarantee that every request receives a response about whether it succeeded or failed). (3) Partition tolerance (the system continues to operate despite arbitrary partitioning due to network failures)

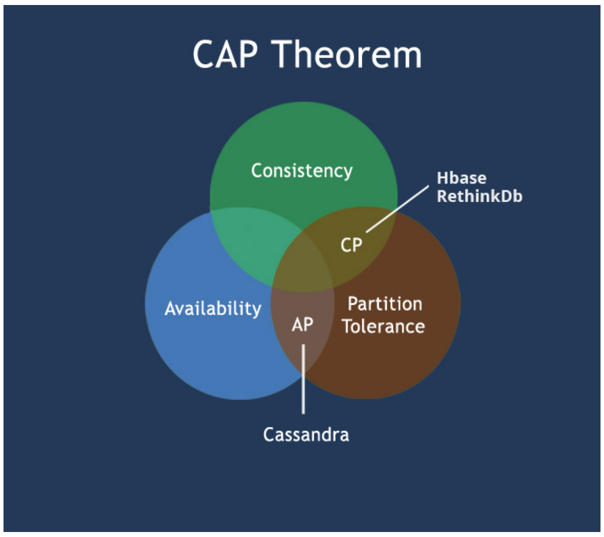

Figure 4.1: CAP Theorem

Both RethinkDB and HBase sacrifice availability in favor of consistency and partition tolerance. On the other hand, Cassandra aims to optimize performance first and foremost, therefore goes for availability and partition tolerance instead of consistency.

ACID. (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction.

- ***Atomicity:*** In HBase all mutations are atomic within a row. That mutate several rows will not be atomic across the multiple rows. Any write will either wholly succeed or wholly fail. However, operations that mutate several rows will not be atomic across the multiple rows. In Cassandra, a write is atomic at the partition-level, meaning inserting or updating columns in a row is treated as one write operation. Cassandra does not support transactions in the sense of bundling multiple row updates into one all-or-nothing operation. In RethinkDB, write atomicity is supported on a per-document basis - updates to a single JSON document are guaranteed to be atomic. However, RethinkDB operations are never atomic across multiple keys.

- ***Consistency:*** HBase seeks to maintain consistency. In contrast to "eventually consistent" Cassandra, which allows to tune between availability and consistency, HBase does not offer various consistency level settings. Thus, the price of HBase's strong consistency is that writes can be slower. On the other hand, in RethinkDB data always remains immediately consistent and conflict-free, and a read that follows a write is always guaranteed to see the write.

- ***Isolation:*** Cassandra and HBase offer row-level isolation, which means that writes to a row are isolated to the client performing the write and are not visible to any other user until they are complete. By default, RethinkDB can return data from concurrent writes that have not been committed to disk yet. The *read_mode* option to table allows control of the isolation level.

- ***Durability:*** All three systems are strictly durable in a way that data that have been committed to disk will survive permanently.

Below is a table that sums up the various properties of the three database systems.

Feature	**RethinkDB**	**Cassandra**	**HBase**
Database Model	Document store	Wide column store	Wide column store
Written in	C++	Java	Java
Partitioning methods	Sharding	Sharding	Sharding
MapReduce	Yes	Yes	Yes
Replication methods	Master-slave	Selectable factor	Selectable factor
Replication mode	Asynchronous & Synchronous	Peer to Peer & Asynchronous	Asynchronous
Locking model	MVCC	Lock Free Model	MVCC
Secondary index	Yes	Restricted	No
Best used	Applications where you need constant real-time updates.	Web analytics, to count hits by hour, by browser, by IP, etc. Transaction logging. Data collection from huge sensor arrays.	Hadoop is probably still the best way to run Map/Reduce jobs on huge datasets.

Table 4.1: Comparison Table

5 | Limitations

Like all other systems, RethinkDB is not without its flaws. In this section we will describe the the most important limitations we came across while examining the system. To achieve a clear representation of the limitations we categorized them into their respective categories.

System requirements. When it comes to system requirements, there are no strict limitations, except that RethinkDB servers must have at least 2GB of RAM. In addition, although RethinkDB runs on several systems (Linux 32bit and 64bit, OS X 10.7) and has experimental ARM support, it does not run on Windows.

Table and Sharding limitations. RethinkDB does not limit the number of databases that can be created, but it limits the number of shards to 32. It also doesn't limit the number of tables that can be created per database or cluster. Each table requires a minimum of approximately 10MB disk space on each server in a cluster, and adds an overhead of 8MB RAM to the server it's replicated on. While there is no hard limit on the size of a single document, there is a recommended limit of 16MB. The maximum size of a JSON query is 64M. Finally, primary keys are limited to 127 characters.

Failover. When a server fails, it may be because of a network availability issue or something more serious, such as system failure. In a multi-server configuration, where tables have multiple replicas distributed among multiple physical machines, RethinkDB will be able to maintain availability automatically in many cases. To perform automatic failover for a table, the following requirements must be met: (1) The cluster must have three or more servers, (2) The table must be configured to have three or more replicas, (3) A majority (greater than half) of replicas for the table must be available.

In Rethinkdb if the primary replica for a table fails, as long as more than half of the table's voting replicas and more than half of the voting replicas for each shard remain available, one of those voting replicas will be arbitrarily selected as the new primary. There will be a brief period of unavailability, but no data will be lost. If the primary replica specified in a table's configuration comes back online after a failure, it will return to being the primary.

Other considerations. As mentioned beforehand, RethinkDB does not have full ACID support. Furthermore, advanced computations are generally slow compared to a column-oriented system. Finally, RethinkDB trades off write availability in favor of data consistency.

6 | Technical Implementation

In this section we will go into detail with the technical aspects of the system's implementation.

6.1 Overview

Basic primitives. Each RethinkDB process has a thread pool with a fixed number of threads. Each of these threads is running an event loop that processes IO notifications from the OS and messages from other threads.

Each RethinkDB server in a cluster maintains a TCP connection to every other server in the cluster. The messages sent across these TCP connections fit into two basic paradigms which are called "mailboxes" and "directory". A mailbox is an object that is created dynamically on a specific server. It has an address, which is serializable; if one server has the address of a mailbox on another server, it can send messages to that mailbox. The directory consists of a set of values that each server broadcasts to every other server it's connected to. A server sets up a mailbox to process a certain type of request and then sends that mailbox's address over the directory so other servers can discover it. These mailbox addresses in the directory are called "business cards".

Executing queries. During start-up one of the constructed objects is an instance, which listens for incoming TCP connections from client drivers. When it receives a message, it converts the message from the client into a tree of ReQL terms. An operation on a table is represented as a read or write object; the result of the operation is represented as a read or write response object. Every RethinkDB table is represented as a B-tree. The blocks of the B-tree are stored in RethinkDB's custom page cache. When the store command finishes executing the read or write against the B-tree, it returns a read or write response, which is passed back via the same way it arrived.

Table configuration. The table's configuration describes which servers should be performing which role for which part of the table. When the user changes the table's configuration, objects on various servers must be created and destroyed to implement the user's new configuration, while making sure that the tables' contents are correctly copied from server to server. There is one multi-table manager per server, created upon initiation. It keeps a record for each table that is known to exist, whether that table has a replica on that server or not. If the server is supposed to be a replica for a given table, the multi-table manager will construct a table manager for that table.

RethinkDB uses the Raft consensus algorithm to manage table meta-data, but not to manage user queries. RethinkDB uses a custom implementation of Raft that's designed to work with the RethinkDB event loop and networking primitives. There is one Raft cluster per RethinkDB table. Each table manager instantiates an object for each raft member. Among other things, the state contains the table's current configuration and a collection of contract objects which describe what each server is supposed to be doing with respect to each region of the table.

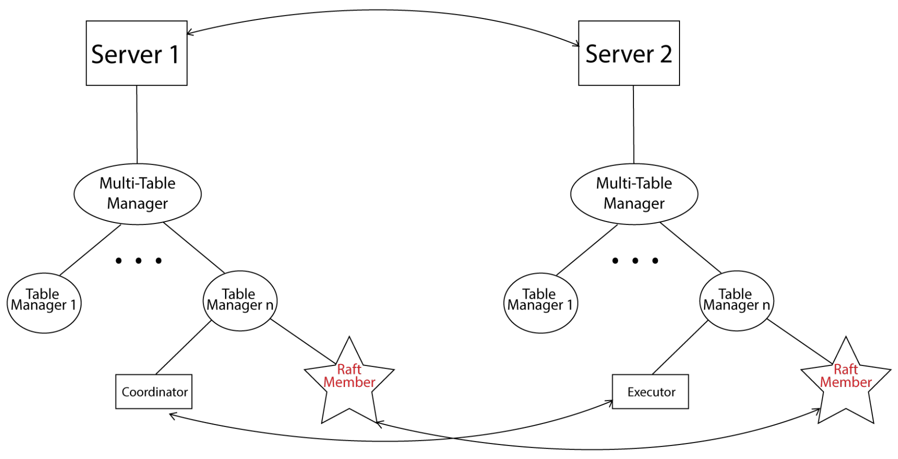

Figure 6.1: Table Configuration

6.2 RethinkDB's Raft Implementation

As mentioned above RethinkDB uses a custom implementation for the Raft algorithm, which has several differences with the one presented in the Raft paper. These differences are listed below.

- **Server initiation.** In RethinkDB's implementation, when a new peer joins the cluster it is initialized by copying the log, snapshot, etc. from an existing peer, instead of starting with a blank state.
- **Snapshot.** RethinkDB's Raft implementation deviates from the Raft paper in the order of snapshots. The Raft paper proposes that the first step should be to save the snapshot, but because RethinkDB only stores one snapshot at a time and requires that snapshot to be right before the start of the log, it saves the snapshot if and when the log is updated instead.
- **Committing index.** The implementation deviates from the Raft paper in that they persist the commit index to disk whenever it changes.
- **Heartbeat process.** Their implementation deviates significantly from the Raft paper in that they don't actually send a continuous stream of heartbeats from the leader to the followers. Instead, they send a "virtual heartbeat stream"; They send a single "start virtual heartbeats" message when a server becomes leader and another "stop virtual heartbeats" message when it ceases to be leader, and they rely on the underlying networking layer to detect if the connection has failed.
- **Log entries.** RethinkDB's implementation slightly differs from the Raft paper in that when the leader commits a log entry, it immediately sends an append-entries message so that clients can commit the log entry too. This is necessary because RethinkDB uses "virtual heartbeats" in place of regular heartbeats, and virtual heartbeats don't update the commit index.
- **Leader election.** In RethinkDB a candidate sends out request-vote messages and waits to get enough votes. It blocks until it is elected or a "cancel signal" message is received. The candidate is responsible for detecting the case where another leader is elected and also for detecting the case where the election times out, and pulsing the "cancel signal" message.

- **Watchdog.** In RethinkDB there is no way for the leader to reply to a virtual heartbeat. So the leader needs some other way to find out that another peer's term is higher than its term. In order to make this work, RethinkDB's implementation varies from the Raft paper in how leaders handle RequestVote messages; Using an additional "watchdog" mechanism RethinkDB detects whether a peer has heard from a leader by using a fixed timer which checks periodically.
- **Resending Messages.** In the Raft paper, servers retry messages if they do not receive a response in a timely manner. RethinkDB implementation deviates from the Raft paper slightly in that it does not retry the exact same message necessarily. If the snapshot has been updated, it sends a newer snapshot on the next try.

7 | Experiments

Before we conclude our examination of RethinkDB, we will conduct some experiments regarding availability and performance. Specifically we will test the system's performance and availability in the following categories:

- Read Performance
- Write Performance
- Ram Usage Statistics
- Availability

All tests were executed on typical personal computer with Intel(R) Core(TM) i7-4510U CPU (dual core with hyper-threading) running at 2.00GHz, 4GB RAM running at 1600MHz and a solid state disk with reading speed up to 540 MB/sec and writing speed up to 500 MB/sec. For all experiments we use threes servers to achieve the majority constraint and separated the data into three shards with three replicas each.

7.1 Read Performance

The goal of this experiment is to measure reading rates under various circumstances such as increasing traffic in varying table size. We considered three different table sizes, small, medium and large which are explained in numbers below.

Table	**Small**	**Medium**	**Large**
Records	0-1000	1000-10000	>10000

Table 7.1: Tables Sizes

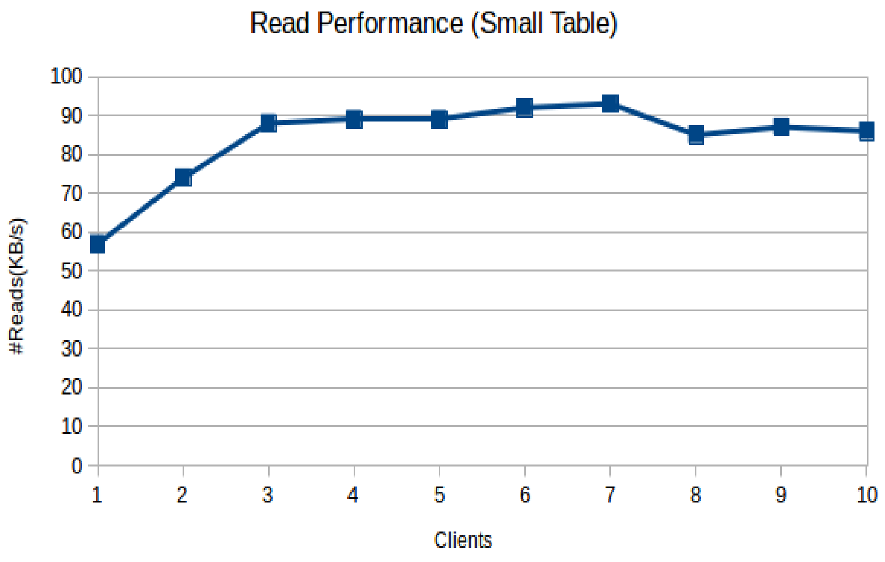

Figure 7.1: Read performance for the small table size category

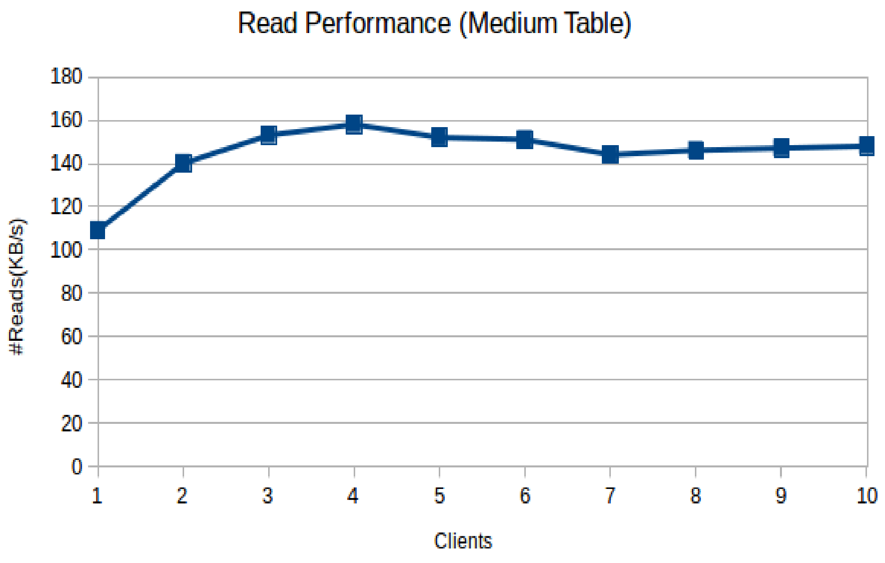

Figure 7.2: Read performance for the medium table size category

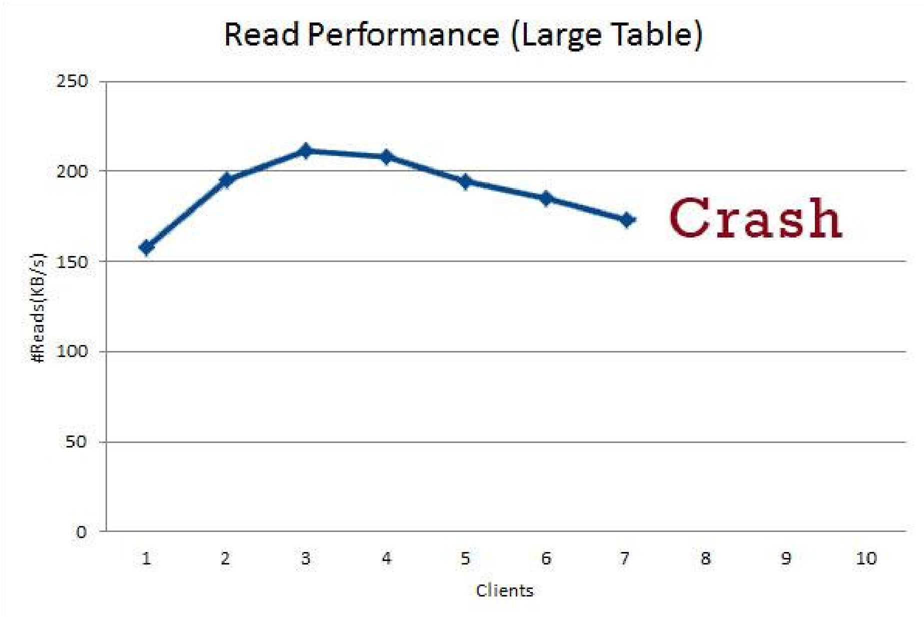

Figure 7.3: Read performance for the large table size category

Observations. We observed that read rates show fluctuations possibly because of the way the system distributes the machine's resources. As we can see in figures 7.1 and 7.2 read performance increases with a steady rate for up to four clients and then stabilizes. This can be explained due to our machine's specifications (4 threads). For large tables as seen in figure 7.3, the results are inconclusive because the machine runs out of available resources which leads to total breakdown.

7.2 Write Performance

To measure the writing performance of the system we simply continuously increase the number of client requests and evaluate the system's response.

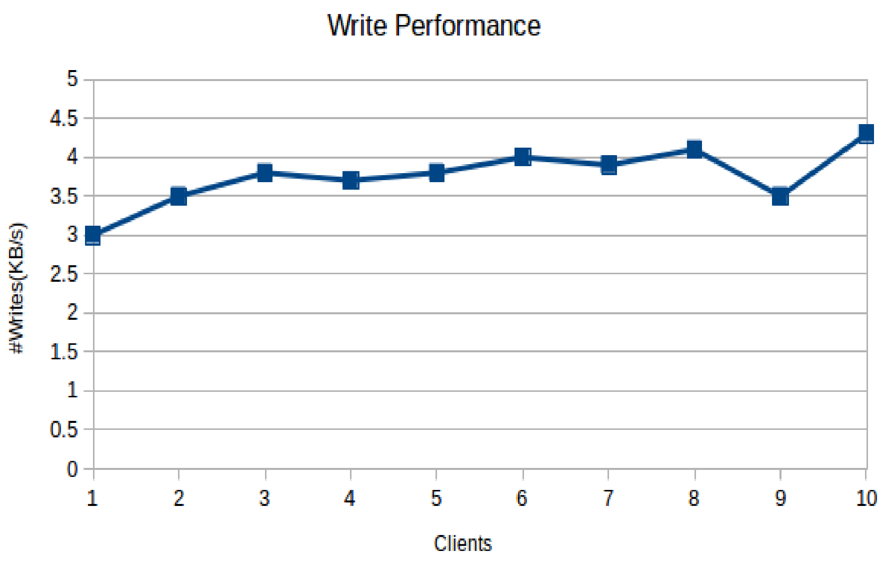

Figure 7.4: Write performance

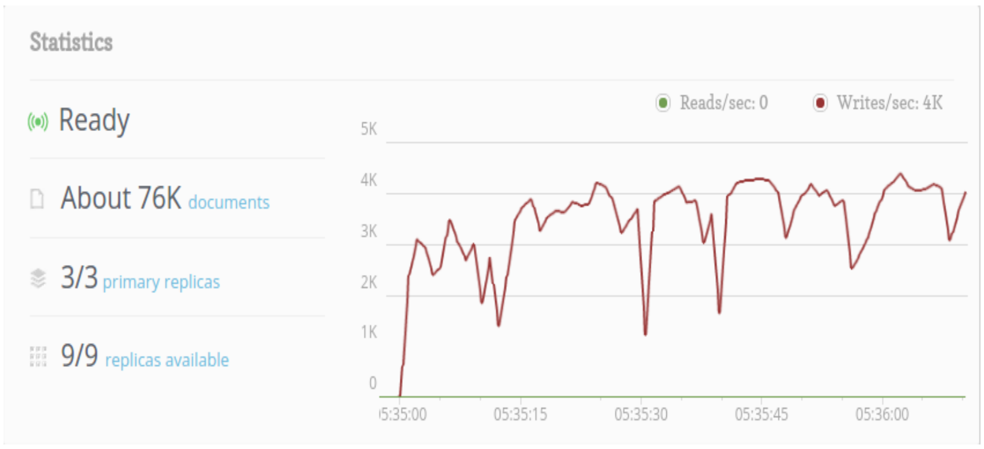

Figure 7.5: Write performance for a single client

Observations. As explained earlier, write speeds exhibit a steady increase up to four clients peaking at approx. 4KBps. It is worth mentioning that writing performance in general follows a spiky behavior as shown in figure 7.5.

7.3 RAM Usage Statistics

In this experiment we will measure the RAM usage while increasing the amount of client requests. Similarly to section 7.1 we used varying table sizes.

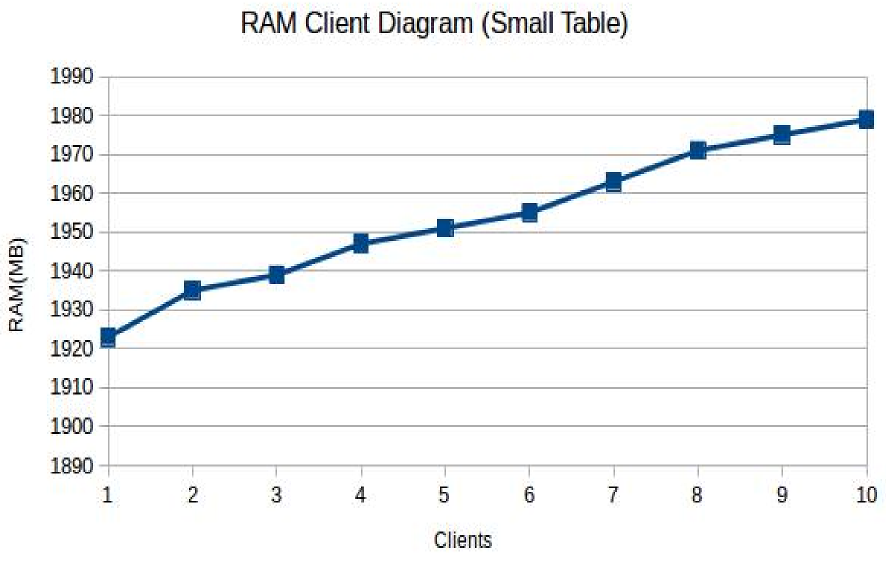

Figure 7.6: Ram usage in the small table size category

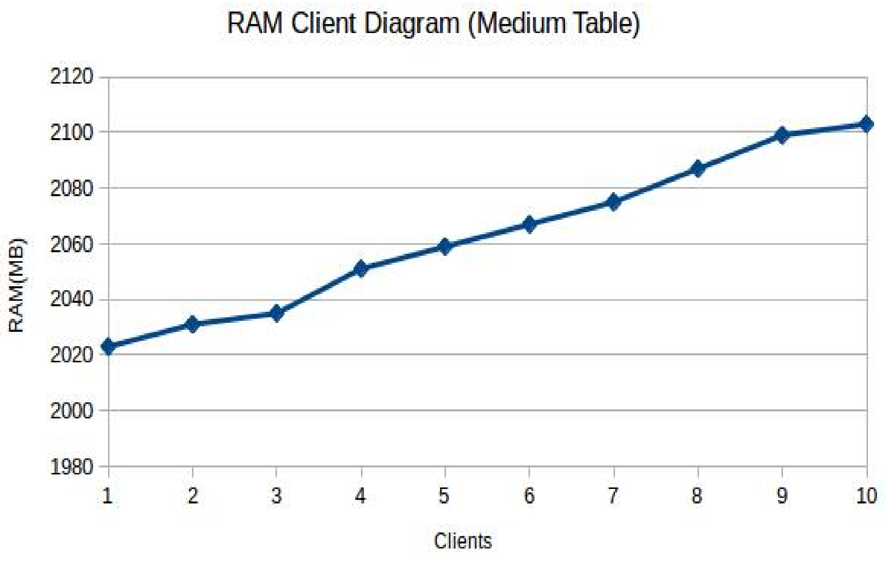

Figure 7.7: Ram usage in the medium table size category

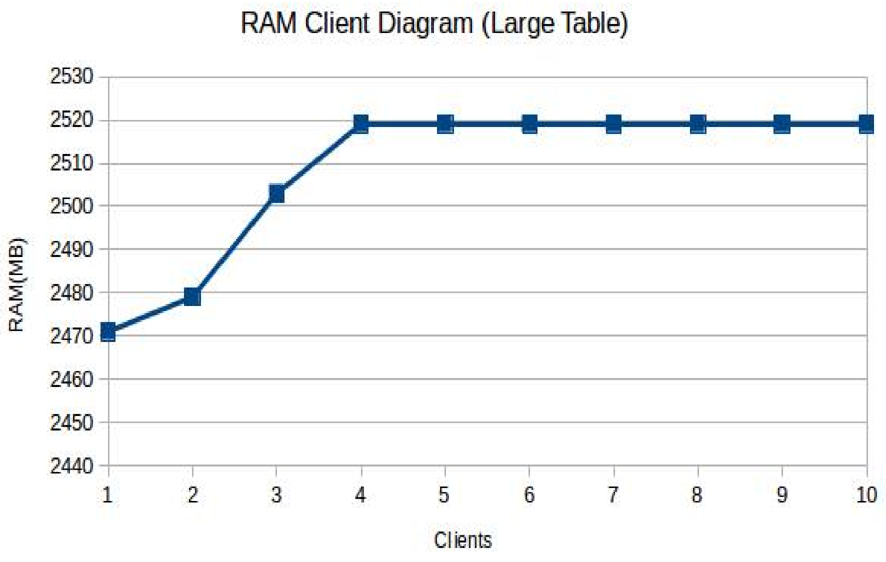

Figure 7.8: Ram usage in the large table size category

Observations. We observe that both for small and medium table sizes, the increase in RAM usage seems to be linear. As expected, medium size tables require more RAM than the small ones. Finally, in the case of large tables the system drastically consumes all available memory leading the machine to a failure state.

7.4 Availability

In the final experiment, we tested the system's availability under certain circumstances such as server failure and online reconfiguration. In the following experiments the recorded delay during reconfiguration is the delay that is experienced from a client during constant read requests. The delay figures reported below do not consider network delay due to the servers running locally. Also, the numbers are measured in seconds.

Table	**Small**	**Medium**	**Large**
Delay(3,3)→(1,1)	0.21235	0.7823	4.1044
Delay(1,1)→(3,3)	0.32643	0.4910	0.8345 & CRASH

Table 7.2: $Delay(x_1, y_1) \rightarrow Delay(x_2, y_2)$ represents a state transfer where x_1 is the number of shards in the previous state and y_1 is the number of replicas, x_2, y_2 are the new states respectively

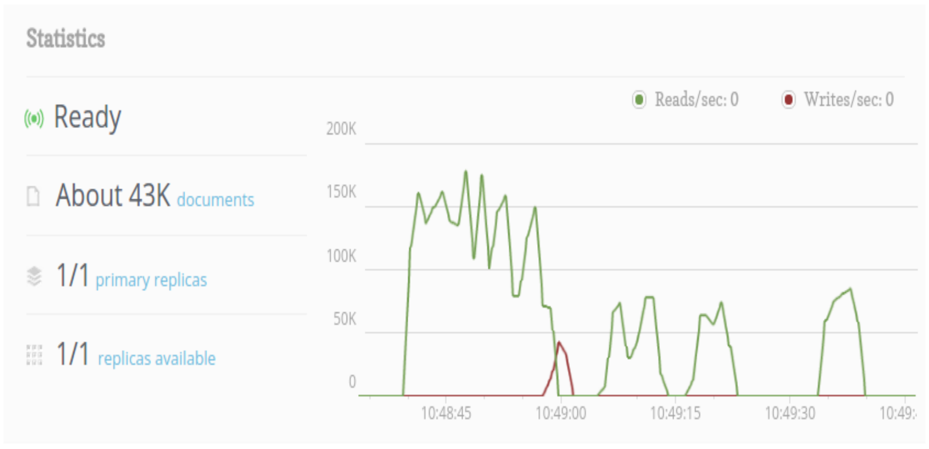

Figure 7.9: System read and write activity during reconfiguration. The red spike signals the start of the state transition. The green spikes are the reads by a single client.

Table	**Small**	**Medium**	**Large**
Delay	0.26731	0.27392	CRASH

Table 7.3: Delay figures under server failure. The reported number are the same for all users and independent of their number.

Observations. As we can see in table 7.2 the number do not seem to follow any pattern in the small and medium table size categories. In the large size category there is a big delay when performing a state transfer into a smaller number shards and replicas and the system starts to behave abnormally as can be seen figure 7.9. Before the start of the reconfiguration (indicated by the red spike in writes) the system functions more or less normally. However, after reconfiguration the system becomes only periodically available to clients. Worse still, in the case of transfer into a bigger number of shards and replicas the system becomes unresponsive and unable to finish the configuration process. This is further proof that the system requires a lot of resources in order to function properly during reconfiguration.

Finally, delay in case of a server crash is the same for small and medium table sizes. The experiment was impossible to carry out in large tables due to the system running out of resources, as mentioned previously.

8 Conclusion

In this project we thoroughly examined every core aspect of RethinkDB and the Raft consensus algorithm which is the underlying foundation for RethinkDB. Regarding Raft we tried to break it down into separate parts to obtain a better understanding of each component. We then proceed to compare it to the benchmark consensus algorithm of distributed systems, Paxos, and highlight their main differences and similarities. After this in depth study, we reached the conclusion that Raft's simplicity and role separation are key elements that will contribute to the future establishment as building and study foundation.

After studying the underlying mechanism we went on to an upper level of the system. To gain a deeper insight of the system's functionalities, we examined the provided features, the practical considerations and the system's caveats. We then compared the system to two basic system categories; widely used database replication and state-of-the-art distributed replication. Considering all the above, we form the conjecture that RethinkDB belongs to the top shelf of distributed database systems. However, it is not always the optimal choice depending on the user's requirements.

Finally, to test RethinkDB's availability and performance we conducted a series of benchmark-like simulations. Although some experiments were inconclusive, the general picture we obtained is that the system operates normally under relatively regular sized data. On the other hand, for big amount of data our personal computer was unable to handle the system's resources requirements. This leads us to the assumption that a typical personal computer is not suitable to manage large scale data. It would be interesting to furtherly test the system on a high-end computer or on a cluster of personal computers.

References

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 305–320, USENIX Association, 2014.

[2] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998.

[3] L. Lamport, "Paxos Made Simple," pp. 1–14, Nov. 2001.

[4] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The smart way to migrate replicated stateful services," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 103–115, Apr. 2006.

[5] J. Maccormick, C. A. Thekkath, M. Jager, K. Roomp, L. Zhou, and R. Peterson, "Niobe: A practical replication protocol," *Trans. Storage*, vol. 3, pp. 1:1–1:43, Feb. 2008.

[6] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," *SIGOPS Oper. Syst. Rev.*, vol. 30, pp. 84–92, Sept. 1996.