Duale Hochschule
Baden-Württemberg

Database Implementation

The NoSQL-Database E-Book

at the Cooperative State University Baden-Württemberg Stuttgart

by

IT14C

11. April 2017

Contents

1 RethinkDB

1.1 RethinkDB

This chapter briefly introduces the major topics around RethinkDB, a real time open source shema-less document-based database. It is widely used by startups and Fortune 500 firms such as NASA, GM and Distractify. Founded in 2009 RethinkDB saw major funding by Y Combinator, had their first release 2012 and went to version 2.0 in 2015. October 2016 the company behind RethinkDB closed down due to the lack of business success but the software got then purchased in February 2017 by a Linux Foundation daughter: The Cloud Native Computing Foundation. It has then been re-licenced under Apache Licence 2.0, going away from their initial copyleft-like licence (CNCF, 2017; Kincaid, 2009).

1.2 How RethinkDB works

1.2.1 Data storage in Rethinkdb

The data structure

RethinkDB stores data as a B-tree-structure. A B-tree is a data structure which is balancing itself and allows filtering and sorting within logarithmic time. This B-tree is saved on a file system instead of the RAM in large structures (Rouse, 2005). This file system is called BTRFS (B-Tree File System), This enables the copy-on-write scheme provided by BTRFS thus making it possible to repair saved data from the copied clone. Other benefits to this filesystem are a garbage compactor to reduce older copies, low cpu-overhead, optimization for solid-state drives, data consistency (more detail about consistency in the ACID and CAP chapter 1.2.3 , b-tree aware caching and multi version control1.2.4. B-Tree aware caching is a way to give RethinkDB the capability of using far more data than available RAM.
RethinkDB does not include a hardware data consistency, this has to be maintained by

the used file system itself, which RethinkDB supports the most commonly available (The Linux Foundation, 2017c)

partioning and multi-datacenter support

Data in RethinkDB can be saved on multiple servers. This is done by replicating databases and providing each of them with a specific tag such as 'de_east' or 'westeros'. A table can have an non-specific amount of replicas on each server. On servers data can be partitioned into shards and furthermore tagged like replicas. The partition is done by a range of specific sharding algorithms and uses the primary key of each table. This means, that a shard key and a primary key is identical. For example if a data set has a primary key containing only letters and is ordered alphabetical, the sharding algorithm will likely split the data around the key 'm'. Thus the new two partitions containing every data with a key from 'a' to 'm' and from 'n' to 'z'. Evidently this algorithm always tries to part at the best pivot to have new partitions in equal size.(The Linux Foundation, 2017g, "How does multidata-center support work")

1.2.2 The atomicity model

Atomicity means, that either the complete stack of operations will be executed or non at all, there is no middle ground (Rouse, 2006). Operations within single json documents are guaranteed to be atomic, queries accessing different keys are not as they may be inconsistent in read and write operations (jepsen.io, 2016). The atomicity differs itself from other NoSQL databases by the way, that every set of operations, every chained query is atomic by the restriction named above. Plus, another limit is, that upon executing a non deterministic operation RethinkDB will nolonger be able to ensure their atomicity. In this case, RethinkDB will automatically throw an error by default. This behaviour may be shut by setting the according flag (The Linux Foundation, 2017e, "How does atomicity work").

1.2.3 ACID and CAP theorems in RethinkDB

RethinkDB's architecture is based, as mentioned in the section above 1.2.2, on the atomicity model. This model is part of the ACID paradigm for databases. ACID is a acronym for atomicity, consistency, isolation, and durability and describes the key desired parameters during a transaction to and off of a Database. ACID is norminized under ISO/IEC 10026-1:1992 Section 4 (Rouse, 2006). RethinkDB has also support for every other ACID paradigm except full isolation and absolute data consistency and therefore might not the best choice if full ACID-Support is needed. But RethinkDB provides a basic consistency as specified within the CAP-theorem. What this theorem is, has been elaborated within the Architectural Basics of Cassandra ??. The basic consistency within RethinkDB has been ensured by the fact, that every shard has a single replication and read and write actions are performed on this replication but not on the shard itself. Data remains immediately consistent and conflict-free. The Database also provides availability needed by the CAP-theorem as data is also accessible both up-to-date and out-of-date. Out-of-date queries are executed on a snapshot and without trying to get the most current data set. This means, that these queries are faster and guarantee availability but may not return nor access the most current data. The up-to-date queries are assuring that they return the latest data consistent and artifact-free. As before mentioned data is not absolutely consistent. This tradeoff roots within the partitioning. RethinkDB assures data to be consistent on the same network but cannot do the same for network partitions if data has to be up-to-date.(The Linux Foundation, 2017a).

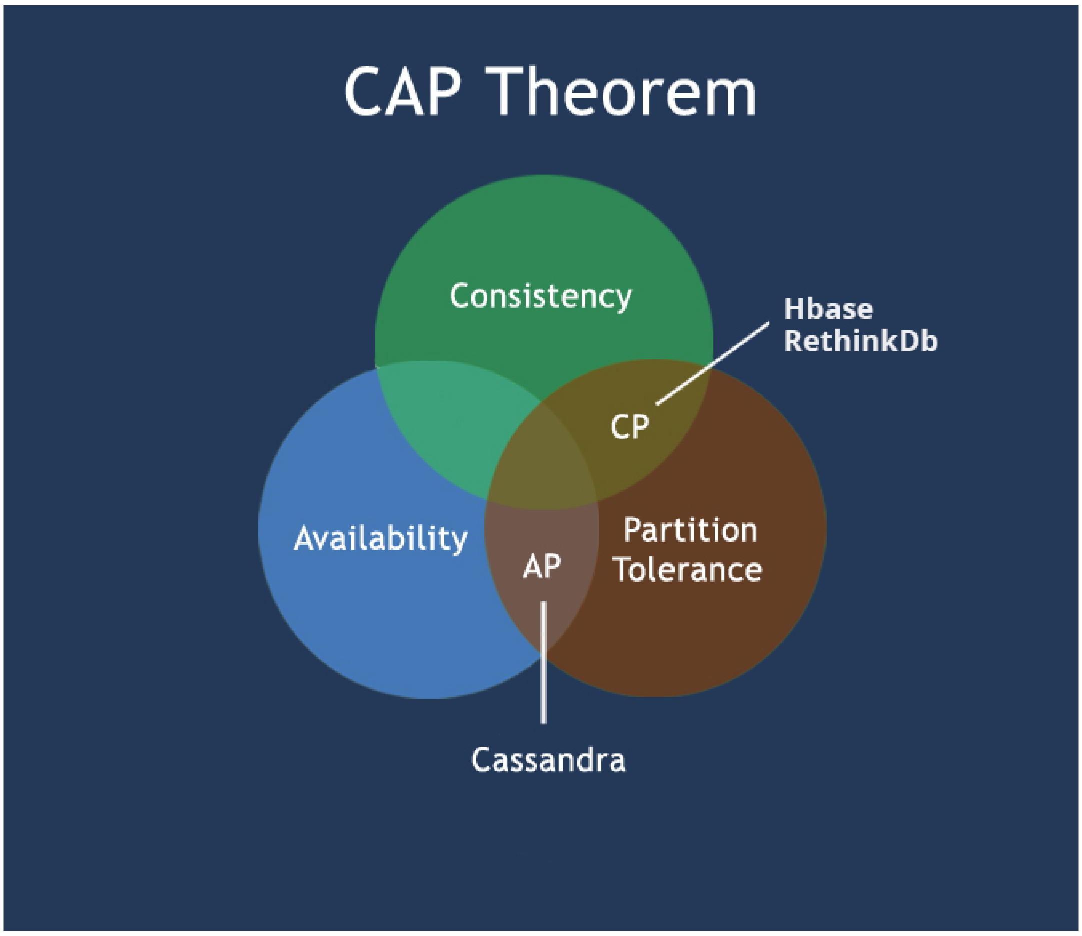

Figure 1.1: Position of RethinkDB in the CAP-Theorem (*CAP Theorem*, n.d.)

1.2.4 multiversion concurrency control

RethinkDB has built-in support for multi version concurrency control. This means, that every writing operation is by default only made after a snapshot, simply a read operation, has been taken off of that tree. This has following benefits:

- Easy roll backing to previous data
- Lock-Free read and write operations
- Enables non-blocking queries making real-time hour long possible

(The Linux Foundation, 2017e, "How are concurrent queries handled? ")

1.3 The query language ReQL

RethinkDB has its own query language called ReQL. A query contains the table, an action and the order to run on a specific database connection. For Example the query to get an id (1) from a table (test) looks like this:

```
r.table("test").get(1).run(connection)
```

Listing 1.1: get document from Database (javascript driver)

One idiomatic aspect of ReQL is already evident in this example. ReQL is a chain of commands. Multiple queries can be written as a one-liner and are thus executed as one without disturbance. If there are dependencies on another query one should use this chain-technique to make sure it is executed in the right order. This works due to the fact, that the query is only parsed and executed on the server while being build on the client. On top of that, RethinkDB is lazy executing the queries. It immediately stops upon satisfaction. Furthermore are those queries functional and allows adding lambda functions as parameter. RethinkDB has build-in support for example for javascript code through the V8-Engine, map-reduce, table-joins and math (The Linux Foundation, n.d.-a).

```
r.db("test").tableCreate("test", options).run(connection)
```

Listing 1.2: create the table 'test'

Every table created gets a primary key for indexing. By default it is id but this can be changed by providing an option:

```
{primaryKey: 'name'}
```

Listing 1.3: options for create table

By setting this option as in the example, the table now is indexing every row under „name“. TableCreate has, as most of the many different options available, accessible in their documentation (The Linux Foundation, n.d.-c).

1.3.1 query executing

RethinkDB has a special way of executing a query. One key point is, that the query is not executed on a client but on the server itself. By receiving the query, the server creates a list of instructions consisting of internal logical operations. RethinkDB now tries to make this list as efficient as possible by executing most basic operations first and more time consuming, such as manipulating data last. Each operation set is called Node and their complexity ranges from single document queries to deep complex subquery commands. After executing, the server returns his result as a datastream not only to the client itself but also to every other relevant server. This has the benefit, that RethinkDB does not really care on which server the query is executed and therefor can parallize this process. (The Linux Foundation, 2017e, "How does RethinkDB execute queries?")

1.4 RethinkDB perfomance analysis

RethinkDB had has a major performance issue since its start but kept on improving on this aspect over the years. For example in a comparison in 2014, MongoDB has been 3 times faster than RethinkDB at executing queries over a large patent data set(juristat.com, 11.04.2017). A newer comparison of 2015 turns this around but only for writing data to the database. Reading, RethinkDB has still been half as fast as MongoDB has been in that benchmark(Thangarajan, 2015). Within an official benchmark test, the RethinkDB Team got this result:

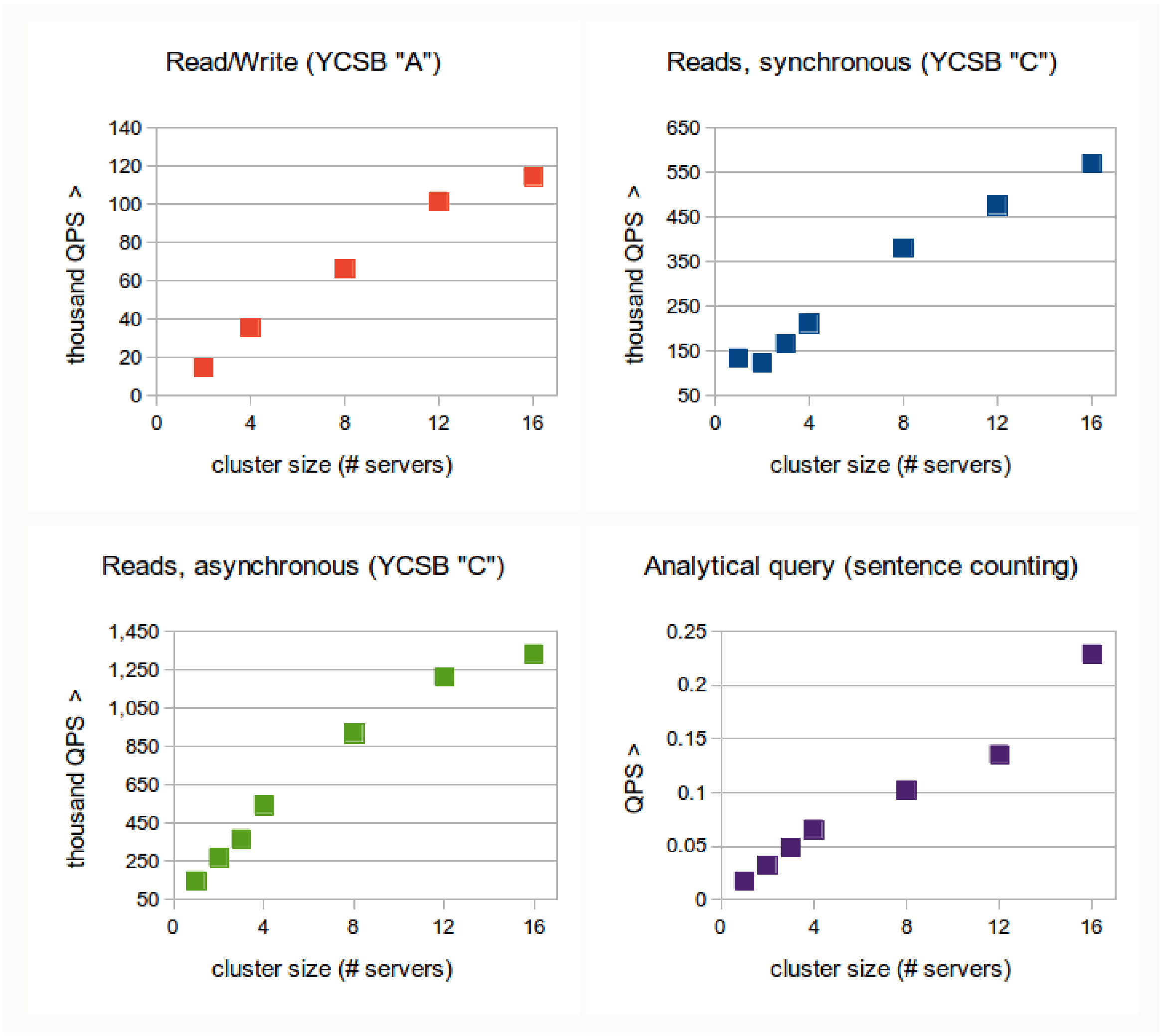

Figure 1.2: Performance Report of RethinkDB (The Linux Foundation, n.d.-b)

They states, that in their working environment they were able to perform around 16 000 queries in a second (QPS). For this benchmark, RethinkDB used "analytic workloads in a simplistic but very common fashion" (The Linux Foundation, n.d.-b).

1.5 Limits

As every existing technique RethinkDB, too, has restictions. These can be hard or soft limitations depending on their context. (The Linux Foundation, 2017d)

It is not possible to create more than 64 Shards in RethinkDB. The number of databases are not limited by anything than diskspace and RAM as is the number of tables. The recommended size limit for a single Document is 16MB and should be respected due to potential perforance issues. Also do Arrays in a RethinkDB Server have a limited length.

This length can be configured per query and by default is at 100.000 elements. Primary keys are limited to 127 characters. But secondary keys aren't hard limited by those 127 characters, if they exceeds their limit, RethinkDB will use linear search for the following characters.

JSON queries are hard limited as well. They can only be 64 MB in size or smaller.

The ordering in RethinkDB is byte-wise. The functions orderBy and between uses the byte representaiton of the carater. This has the result that RethinkDB dose not normalize identical characters with multiple codepoints. For example the character "é" has the UTF-8 representation `\u0065\u0301` and `\u00e9`, the will be grouped separately.

The usage of the cli option `-direct-io` is restricted to supporting file systems only. Typically encrypted and compressed file systems will not support this option.

Numbers are double precision IEEE 754 floating point. Integers are stored precisely from -2^{53} to 2^{53}, outside that range they will be rounded.

1.6 Data Types

RetinkDB has all the basic Data types. Number, string, boolean, object, array and the `null` value. Additionaly RethinkDB stores specific data types like tables, streams, selections, binary objects, time objects, geometry data types, and grouped data. (The Linux Foundation, 2017f)

Numbers can bee Integer or Floatingpoint values. They will be stored with 64-bit percision. `NaN` or Infinit can not be stored.

String are stored with UTF-8 encoding.

Booleans can be `true` or `false`

Null is a special value. It is not the number zero or an empty string. It symbolizes the absence of any other value.

Objects are key-value pairs. Any valid JSON object is a valid RethinkDB object.

```
{
  "key"  : "valueString",
  "key2" : false,
}
```

Listing 1.4: example Object

The keys can only be strings, but the values can be any data type. A RethinkDB Document is a object.

Arrays are lists of values. Arrays can be empty. It is not enforced to use only on type of values in an Array. But it is highly recommended.

```
[
  "valueString",
  false,
]
[]
```

Listing 1.5: example Arrays

The values of an Array can be any data type. If you use arrays for many values, more than 100.000 you should notice that arrays are fully loaded into Server RAM before being send to the client.

Specific Data types

Databases are returned after a `db` function call.

Tables are returned after a `getAll` function call. Their behavior is similar to selections.

Selections are the result of `filter` ot `get` function calls. Selections comes in three variants `Selection<Object>`, `Selection<Array>` and `Selection<Stream>`.

Streams are like arrays, lists. Streams are not writable and are lazy, they give you the next dataset if the current has been processed. With Streams you can read tremendous long lists of Data, without holding everything in Memory.

1.7 Changefeeds

Changefeeds are the main part of RethinkDB's real-time functionality. Almost all queries can be used as a changefeed. With a Changefeed you recive any changes on your query. (The Linux Foundation, 2017b)

```
r.table('project').changes().run(conn, (err, cursor) => {
  cursor.each((err, row) => {
    if(err) throw err;
    processRow(row);
  })
});
```

Listing 1.6: example usage

The `changes()` method returns a cursor. The `each()` method of that cursor object iterates over each change. If the underlaying data of the query changes the callback of `each()` will be called. For example a new project will be added `id:3, name: "awesome Project", priority:` into the project table. The `each()` callback will be called with the changes of the table.

```
{
  old_val: null,
  new_val: {id:3, name: 'awesome Project', priority:1}
}
```

Listing 1.7: example usage

If the callback for `each` returns false, the cursor will stop iterating.

single document changefeeds

The single document changefeed only notifies the subscriber if the document changes.

```
r.table('project').get(3).changes().run(conn, callback);
```

Listing 1.8: example usage

This would listen for changes of the project with id 3.

Filtering Changefeeds

The method `changes()` integrates with the ReQL language. `changes()` can be used with mostly any outher command.

The chaning of ReQL methods can be as complex as you wish. For Example changefeeds can be filter:

```
r.table('project').changes().filter(
    r.row('new_val')('priority').lt(r.row('old_val')('priority'))
)('new_val').run(conn, callback)
```

Listing 1.9: example usage

This will be notified on every change, there the priority is lower as the previous. Chaining methods with changefeeds has some limitations.

handling latency

With some latency on the client side, it is possible that there will be mutliple writes into a table, before the clinet gets all changefeed events. By default a changefeed subscriber will only get one change object. If this is not the desired procedure the option `squash: false` should be used on the `changes()` method. With this the change object will be send for every change.

Changefeeds do not have a delivery guarantee.

1.8 Conclusion

RethinkDB is a highly potential database with many use cases and some valuable customers. Being consolidated under the Linux Foundation, this potential might be more furthered and current problems, like the performance issues reduced. In this future, the lack of documented community drivers will hopefully evaporate and more and more third libaries will be added to RethinkDB. Even though the namespacing, with just the *r*, might be strange to use at first but due to the query chainability, it's many functions and the changefeed make up a lot for the uncertainty resulted in shut down and revival. RethinkDB's easiness in usability and installation-wise, it's simple web interface and the well-elaborated documentation are reason enough to give it a shot.

To synthesize it, RethinkDB is worth considering for web & app projects alike especially if real time support is needed.

References

Cap theorem. (n.d.). Retrieved from http://cs.uoi.gr/~apappas/projects/Raft&Rethinkdb/img/cap.jpg

CNCF. (2017). *Cncf purchases rethinkdb source code and contributes it to the linux foundation under the apache license.* retrieved on 10.04.2017 from https://www.cncf.io/blog/2017/02/06/cncf-purchases-rethinkdb-source-code-contributes-linux-foundation-apache-license/.

jepsen.io. (2016). *Rethinkdb 2.1.5.* retrieved on 11.04.2017 from https://jepsen.io/analyses/rethinkdb-2-1-5.

juristat.com. (11.04.2017). *A comparison of mongodb and rethinkdb with patent data.* retrieved on 10.04.2017 from https://juristat.com/blog/a-comparison-of-mongodb-and-rethinkdb-with-patent-data.

Kincaid, J. (2009). *Yc-funded rethinkdb: A mysql storage engine built from the ground up for solid state drives.* retrieved on 10.04.2017 from https://techcrunch.com/2009/07/28/yc-funded-rethinkdb-a-mysql-storage-engine-built-from-the-ground-up-for-ssds/.

Rouse, M. (2005). *B-tree.* retrieved on 10.04.2017 from http://searchsqlserver.techtarget.com/definition/B-tree.

Rouse, M. (2006). *Acid (atomicity, consistency, isolation, and durability).* retrieved on 10.04.2017 from http://searchsqlserver.techtarget.com/definition/ACID.

Thangarajan, N. (2015). *Mongodb vs. rethinkdb.* retrieved on 11.04.2017 from http://ntvita.com/2015/06/04/mongo-vs-rethink.html.

The Linux Foundation. (n.d.-a). *Introduction to reql.* retrieved on 11.04.2017 from https://rethinkdb.com/docs/introduction-to-reql/.

The Linux Foundation. (n.d.-b). *Rethinkdb 2.1.5 performance & scaling report.* retrieved on 11.04.2017 from https://rethinkdb.com/docs/2-1-5-performance-report/.

The Linux Foundation. (n.d.-c). *Rethinkdb create table.* retrieved on 11.04.2017 from https://rethinkdb.com/api/javascript/table_create/.

The Linux Foundation. (2017a). CAP Theorem. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/architecture/#cap-theorem

The Linux Foundation. (2017b). Changefeeds in RethinkDB - RethinkDB. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/changefeeds/javascript/

The Linux Foundation. (2017c). How is data stored on disk? , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/architecture/#data-storage

The Linux Foundation. (2017d). Limitations in RethinkDB - RethinkDB. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/limitations/

The Linux Foundation. (2017e). Query Execution. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/architecture/#query-execution

The Linux Foundation. (2017f). ReQL data types - RethinkDB. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/data-types/

The Linux Foundation. (2017g). Sharding and replication. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/architecture/#sharding-and-replication