Université Libre de Bruxelles

INFO-H-415 - Advanced databases
Project report: RethinkDB

Akira BAES, Antoine VANDEVENNE

Contents

1 Introduction

The emergence of real-time databases allowed data to be shared between several applications in an instantaneous manner. Such applications include collaborative real-time document edition, multi-player games, and real-time analytics boards. It is possible to implement this feature in several ways, among which the usage of a database with real-time capabilities.
In order to explore and experiment with this particular solution, we focused our work on RethinkDB, a real-time oriented document store. RethinkDB is a database management system (DBMS) using JSON files as documents and offering a particular data access model, the so-called "changefeeds" which allow applications to receive continuously updated results from the database for specified a query.
In this document, we begin with a deep analysis of RethinkDB in section 2. Section 3 overviews our implementation of a collaborative drawing game. In sections 4 and 5, we respectively analyze the performances and convenience of RethinkDB by respectively reviewing benchmarks and comparing it with similar other databases. Finally, a conclusion is provided in section 6.

2 RethinkDB

RethinkDB is a noSQL, document-store database with real-time capabilities. As such, it is non-relational: the database is document-oriented and the data are stored in JSON documents with no fixed table structure. Its real-time capabilities lies in the feature of changefeeds which aims to compete with classical real-time databases by providing a more flexible concept where mostly any query can be monitored for changes.

2.1 Web Administration Console

RethinkDB comes with a Web Administration Console. The console features a visual overview or the server's activity or database's performances, and permits several manipulations such as the management of sharding and replication on a per-table basis, a data explorer allowing to execute ReQL queries in JavaScript. It even offers a very basic query profiler, though it is difficult to interpret.

2.2 Introduction to ReQL

In order to properly introduce the different concepts of RethinkDB and concretize them with a few examples, a shallow overview of ReQL is provided in this section.
RethinkDB features its own query language, ReQL. It comes with several API to embed itself in multiple officially supported programming languages. To this day, ReQL is available for JavaScript, Python, Ruby and Java but there exist many other community-supported drivers, among which C++ and Go, for instance.
ReQL is focused on the manipulation of JSON documents and the creation of chainable queries. To create a query using ReQL, one must start from the RethinkDB's driver object and append the desired operator to the former in the same fashion as a call to an object's function in most high level programming language. The query can then be refined by successive appending of new operators to the end of the chain.

To get a better grasp on this concept, a simple example is provided in Listing1. The RethinkDB driver is represented by the variable r to which are successively appended two operators. The first one specifies the database to work on (here "Example") and the second one requests that a new table named "Fruits" must be created on this database. This operation returns an object representing the query that is yet to be executed. This is done by establishing a connection with the server and appending the *run* operator to which the connection should be passed. The execution will then entirely take place on the server. As we are not interested about how a query execution is handled as of yet, we will omit the details concerning the connection et the execution in the following examples.

```
query = r.db("Example").tableCreate("Fruits")
query.run(connection)
```

Listing 1: Creation of a table

```
r.table('Fruits').insert([
    { 'name': 'Apple', 'amount': 178 },
    { 'name': 'Orange', 'amount': 75, 'unit_price': 1.5 }
])
```

Listing 2: Insertion of documents (Python)

Listing 2 exhibits the operator *insert* used to load new documents in the database. The selection of the table in which those documents should be loaded is done by the mean of the *table* operator. Several observations on this Listing will be made in the following section but, as of now, its only purpose is to display the content of the documents currently present in the database.

Given the two documents previously inserted, it is now possible to retrieve documents based on arbitrary filters. Where in SQL one would have made use of the "SELECT", "FROM" and "WHERE" keywords to construct a basic query, in ReQL they are replaced by a combination of the "table" and "filter" operators to which can be adjoined one or several conditional operators. Such a query and its result are respectively depicted in Listings 3 and 4. In this case, the *gt* operator is used to retrieve all fruits whose specified amount is strictly greater than 100. As only one entry matches this query, the result consists of a single document containing all field of the matched entry.

2.3 Document store

A document store or document-oriented database is a database that stores all information about a given object in a single document which is semi-structured data. There exists no scheme or inherent structure to the tables. This is in contrast to table-based relational databases in which the information describing an object would be split across several tables. As such, document stores are a main category of the NoSQL databases family, and are a subclass of key-value stores. [27]
In document-oriented databases, the content of data is not opaque to the system.

```
r.table('Fruits').filter(r.row('amount').gt(100))
```

Listing 3: Retrieval of documents (Python)

```
{
    "id": eda155d3-567a-4773-8179-741fd81104fd",
    "amount": 178,
    "name": "Apple"
}
```

Listing 4: Query result of Listing 3

```
{
    "id": "8644aaf2-9928-4231-aa68-4e65e31bf217"
    "CourseName": "Advanced Databases",
    "Mnemonic": "INFO-H-415",
    "AcademicYear": "2017-2018"
}
```

Listing 5: Example of JSON document

Instead, the internal structure of the documents can be exploited to generate metadata used for optimization. [22]

RethinkDB uses JSON as its document structure. In programming languages that support it, JSON-like object structures can be represented at the application level using data types of the language itself. For example, in Python a JSON RethinkDB document will be parsed into a dictionary, a built-in data type of Python where a non-mutable key maps to an entry. It makes managing data very easy, as dictionaries can be explored in Python without further parsing and internal structure can be further extended by nested Python lists or dictionaries. As such, Python dictionaries and RethinkDB document are interchangeable concepts (insertion, query manipulation, ...).

As an example, the JSON file depicted in Listing 5 could be used to represent the course of Advanced Databases and be stored in a document-oriented database. As RethinkDB belong to the class of key-value stores, a primary key is required to identify each document within a table. In this example, and by default, this key is represented by the field *id*. As can be observed in Listing 4, this field is present in the query result although no identifier was provided when inserting the entry in Listing 2. Indeed, if no specific value for the primary key is provided, a new and unique one will be automatically generated upon insertion of the entry. In addition, RethinkDB allows to specify the field or the combination of fields to be used as the primary key when creating a new table.

2.4 Real-time database

Real-time databases are databases that are supposed to work for reactive applications where changes in the database are immediately visible on the user's end. The traditional request-response pull-based query structure does not provide easy tools to do that without implementing them by yourself. Real-time oriented databases offer push-oriented queries natively to facilitate the writing of applications that try to reflect the real-time current state of the database by having the database itself react to modifications and send to the application the changes immediately as they occur. [11] [9]

In a classical database, this can be done by polling the database at regular intervals for changes.
In most real-time oriented databases, you can request the server to send you real-time updates on a file, or keep you synchronized on the state of a document. RethinkDB simply generalizes the concept of real-time updates to any query to make it easier to write, including queries that transform or select the data. [15]

This changefeed approach was unique to RethinkDB [11], but other databases such as Meteor provide similar push-based queries. Even more recently, a new database named Baqend boast to take inspiration from RethinkDB's approach but refine it with self-maintaining queries. One last kind of real-time databases that is currently popular are the cloud-based real-time database like Firebase whose approach are better described as cross-device state synchronization, as you can share one document between several applications. That implementation is thus a lot more limited, dealing with the synchronization of a single document entry rather than arbitrary queries. [9]

2.5 Indexing

As they are not SQL-based, document store databases are not "indexed" in the same way SQL databases records are. It is however possible to create artificial indexes on the data, which can help increasing performances.

In RethinkDB, an index is automatically created on the primary key field which has many speed advantages, as the data is internally stored in a B-Tree, a sorted data structure that allows quick retrieval of stored data. [13]
Secondary indexes on arbitrary entries are available to speed up research at the cost of memory usage. [22] Creating a secondary index also allows for more advanced ordering and data manipulation. [15] Secondary indexes can be based on any field, a combination of fields or even arbitrary expressions.
Upon any write operation, an update operation for all indexes is started. [22]

As an example, Listing 6 shows the process of creating a simple secondary index on the field "amount", in our previous database example. This is done by a simple call to the operator *index_create* followed by a call to the operator

```
r.table("Fruits").index_create("amount")
r.table("Fruits").index_wait("amount")
```

Listing 6: Creation of a simple index (Python)

```
r.table("Fruits").index_create(
    "infos", [r.row["amount"], r.row["unit_price"]]
)
r.table("Fruits").index_wait("infos")
```

Listing 7: Creation of a compound index (Python)

index_wait, this ensures that the index will be ready before one submit a new query.

Listing 7 depicts the creation of a compound index. This type of index is similar to simple indexes but differs in that it applies on multiple fields and returns an array of values rather than a single result.

Listing 8 shows the creation of a multi-index, in which a document can have multiple keys in the same index. This type of index is used to create tags, as one document can show up in different keys (or tag) searches.

Finally, a fourth type of secondary index is made available in RethinkDB: indexes on arbitrary ReQL expressions. By the mean of a anonymous function, it is possible to construct an index on the total value of fruits by multiplying the *amount* and *unit_price* fields, as can be seen in Listing 9.

2.6 Data storage

As was already mentioned earlier, the storage of RethinkDB's data takes place in B-Trees. [13] The storage engine features a log-structure, multi-core operations, data recovery after power failure, full consistency of data [16], but no check for material data corruption. The data is cached into a B-Tree aware structure that uses around 1% of the data size in RAM. [18] For instance, the structure for 1 TB of data would occupy about 10 GB of RAM. [13]
The database allows to dump/restore the database for backup purposes, but it does not have advanced backup capabilities. [14]

2.7 Sharding

RethinkDB allows for easy sharding via the web interface. RethinkDB shards the database based on the primary key. [13] RethinkDB will determine how to

```
{
    "name": "Apple",
    "amount": 178,
    "producers": [ "belgium", "canada", "france"]
}

# Create the multi-index on the tags list
r.table("Fruits").index_create("producers", multi=True)

# Get all results of given tag
r.table("Fruits").get_all("belgium", index="producers")
```

Listing 8: Creation of a multi index (Python)

```
r.table("users").index_create(
    "total_value", lambda fruit:
    fruit["amount"] * fruit["unit_price"])
```

Listing 9: Creation of an index on an arbitrary ReQL expression (Python)

split the table to evenly distribute them between the number of shards you entered, either via the web console or via the API. [20] All the queries are then handled and sent automatically to the relevant shard. Sharding is pretty scalable on RethinkDB for up to 64 shards on the same table. [18]

Database replication is handled the same way, as you enter a number of replicas and RethinkDB takes care of what happens under the hood. While you can change those number anytime and request a re-balance, RethinkDB will not adapt those numbers on the fly. [20]

2.8 Query execution

Queries in RethinkDB are basically "transformed into an execution plan that consists of a stack of internal logical operations". [13] RethinkDB separates operation nodes that deal with lower-level access like table scan, index ranges and document lookup, which can determine to which server the query is sent (parallelisation), and nodes that deal with higher-level operations like data transformations, mapping, grouping, by stacking them in that order. This explains why some combinations are allowed for changefeeds and some are not. [15]
A lazy evaluation is done by evaluating the top of the stack first. The top nodes then ask the node below it for some data recursively, until it has enough to send a response to the client. [13] The lower nodes are thus rarely executed in their entirety. This permits fairly complex queries to be executed efficiently.

However, some query structures cannot be evaluated lazily or parallelised easily. While planned, there is currently no tool in RethinkDB to analyse the query complexity. The documentation advises to reach out and ask for help to the RethinkDB developers in case of problem. [13]

Write atomicity is assured for any combination of deterministic operations on a single document. By default, RethinkDB will not allow non-atomic operations on replace or update. [13]

2.9 ChangeFeeds implementation

It is possible to create a changefeed on nearly every valid query. The database will send any update on the query in real-time to the application. Changefeed push notifications are unidirectional and can not guarantee delivery. [15]
The database servers are responsible to handle the changefeeds send the requests to all corresponding shards, after what they are executed lazily like any normal query. [13]
All shards share together all updates before sending them back. This creates a lot of intra-cluster messages in proportion to the number of involved servers. To reduce this, RethinkDB allows to create what is called a "proxy node", which will centralize all the changefeed-related messages routing and filter the duplicates. [20]
Some query types will not allow for the creation of changefeeds. For example, *min/max/order_by.limit* must be called on an existing sorting index (primary or secondary) to be able to use changefeeds on them. Transformations that do not allow data to end, such as an *order_by* without a result size limit, cannot be transformed into a changefeed. Similarly, transformations that consume the whole feed as *.count* and *.order_by* cannot come after *.change*. [15]

2.10 Advanced use of ReQL

2.10.1 Joins

Like many databases, RethinkDB supports table JOIN commands. Like the rest of the commands, the cluster/shard complexity is hidden by the implementation. RethinkDB has several versions and ways of doing JOIN. The fastest, *eq_join*, can be used on indexed fields only, and returns a table of documents with two fields (left, right) which contain the joined documents. Those two fields can be fused together with a call to a zip command, but for more control about which how fields are treated, the map command can be used.
For joining tables on something else than indexed fields, *inner_join* and *outer_join* can take arbitrary lambda functions to join tables without having to create secondary indexes. They are however a lot slower in consequence and are generally not recommended to use.
One-to-many join on indexed fields can also be done by querying for one "parent" document and merging it with the merge command to a *get_all* query

(which is similar to a *filter* command for indexed fields).
Many-to-many joins on indexed fields can be done by chaining several *eq_join* together, if the field on which we join first is reused for the second join (as it is only fast because data will be ordered on that field).
Other operations can be done on the joined sequences via transformation methods such as map, etc.

2.10.2 Transformations

RethinkDB features a lot of the classical data manipulation functions and allows to use them in queries to manipulate data.

map can be used on one or more sequences to apply a lambda function to manually merge or transform the documents in the sequence(s).

order_by can be used on one sequence to sort the resulting stream either based on an index (primary or secondary), or on a "deterministic" lambda function (no random or user input), which will be slower and is currently limited to sorting 100.000 documents, as it requires to keep them in memory. It is often used with a *limit* statement that ends the sequence after a given number of elements.

union merges two sequences in one longer sequence and can interleave them based on order if an ordering method is passed. As the manipulated elements are documents, there is no constraint on the schemes having to match.

There are other manipulation queries such as *sample*, *slice*, *with_fields* (to filter based on fields), etc.

2.10.3 Aggregations

RethinkDB features all the classic aggregation queries. Most will take lambda functions to aggregate on non-indexed values.

group takes a stream and partitions its documents into multiple groups based on a field or a function. The result will be a document where each field is a group value and contains a list of the grouped documents. With the right parameters, it even allows a document to appear in multiple groups if the grouping value is an array or values.

ungroup can be used to turn a grouping back into an array of elements. It is useful when you apply reductions on the groups and want the result in a list rather than in the fields of one document.

reduce transforms a stream into a single value by applying repeatedly the given function on pair of elements. It can be used along with group and map to explicitly perform the (group-)map-reduce pattern to aggregate large amount of data. [19]

fold can apply a function in order on a sequence and maintains a state in an accumulator. It can either act like a reduce but is guaranteed to work with the elements in their original order, or it can produce a new sequence by consuming all the elements and emiting new ones based on the accumulator's value.

Other classical aggregation queries like count, sum, avg, min, max, distinct (on a given field) are directly available and function as usual.

2.11 Other features

RethinkDB does feature some basic spatial database functions. It does not support multiple reference systems, and is limited to sperical longitude/latitude. [17]
RethinkDB does not feature triggers per-se, but the changefeeds can be used to do the same work trough the server application.

3 Our application

In order to explore the different possibilities offered by RethinkDB, is devised a simple yet interesting application relying heavily on changefeeds.

3.1 Use case

The application consists of a collaborative drawing game taking place on a grid of pixels stored in a persistent database. At the beginning of the game, the database is initially loaded with blank pixels representing the entire game map. As a pixel corresponds to every possible pair of valid coordinates, an instance of a 10.000 x 10.000 game map would require the insertion of 100.000.000 blank pixels in the database before the beginning of the game.

When a user wishes to play, i.e. collaborate in the drawing, he is first asked to enter the coordinates of the top left corner of his game window. After submission of valid coordinates, he is assigned a random color and shown a window containing a grid of 50 x 50 tiles representing the same amount of pixels in the database. He can then click on any tile present in the game window to fill it with the color that was assigned to him, thus changed to color or the corresponding pixel in the database.
As a user is coloring pixels in a section of the game map, any other user whose game window contains those pixels will see their color change in real-time. No particular protections of colored pixels having been set up, any user can draw on any section of the game map without restriction.

Furthermore, it is possible to specify for a client to run in *bot mode* instead of *user mode*. Once valid coordinates are inserted, the client will automatically follow a random linear path within the game window and bounce on the edges of the latter. Any pixel on this path will be colored in a random color. This mode allows for better visualization of the game mechanics and provide a more interactive testing bed to regular users.
As the focus of the use case was on the immediacy of color updates, no particular effort was made to easily move the game window (although it can be done by restarting the client and entering new coordinates) nor provide a complete account management system.

Figure 1 depicts two regular user's windows focused on slightly offset coordinates. Two automatic bots had previously filled square areas corresponding to smaller coordinates than that of the users' windows. Thus, one of those square appears only in the top left part of the second user's window while the second appears in both users' windows.

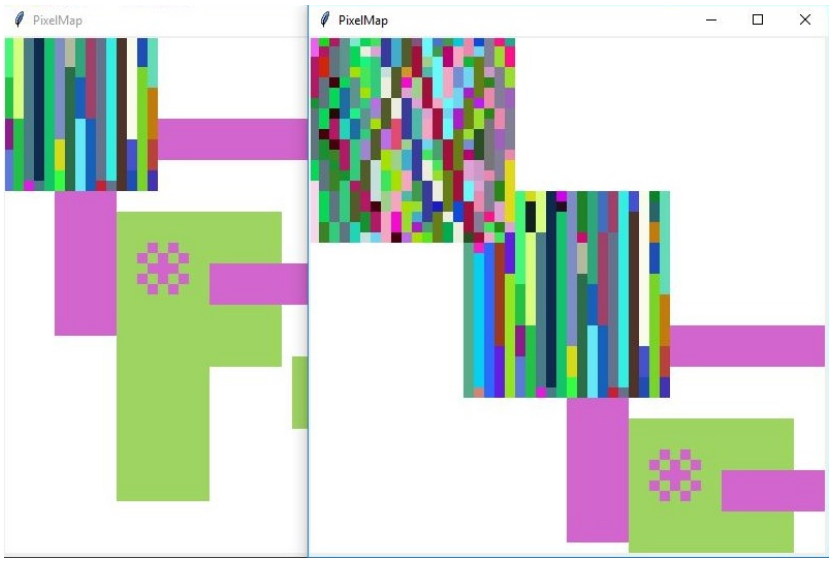

Figure 1: View of two game client focused on slightly offset coordinates

```
Database setup completed.
Tables creation completed.
Game dimensions loaded in the database.
Packing blank pixels in batches : 100.00 %
Loading batches of blank pixels into the database : 100.00 %
Full game grid loaded with blank pixels.
```

Figure 2: Initialization script loading blank pixels in the database

3.2 Practical details

The application is written in Python 3 and makes use of the following non-default packages: *tkinter*, *rethinkdb*. Thus, each of those is required to execute the different modules, in addition to the server executable.

Along with this report, four modules are provided: *main.py*, *init.py*, *drop.py* and *config.py*. The first module contains the entire client application, the second and third modules serve to manage the database and the required tables, and the last module allows to configure database parameters and game settings. Upon execution of the module *init.py*, the insertion in the database of all pixels fitting in the configured game map dimensions starts. As this process may take some time, a progress percentage is provided as shown in Figure 2.

3.3 Implementation choices

Once established, a connection is kept open and used for a given type of communication as long as the client is running. Thus, for a regular user's client, two connections are established: one for the update queries related to the drawing of the game map and another to handle the changefeed.
Because opening of a connection takes a significant amount of time, it was not possible to open and close a connection for every query as it would have induced non-negligible latency in the rendering of the game map.

Different levels of durability and safety were available for sending queries to the server. We choose to use soft durability with replies, meaning that we receive write ACKs before writes are committed to the disk, but we wait for the server's ACK before sending the next query. Our experimentations proved this solution to be the most efficient as a noreply write loop can lead to loss of data in our case.

As was mentioned earlier, the initialization script provided with the module *init.py* proceeds to fill the database with blank pixels when executed. As the query corresponding to each insertion of one pixel can be precomputed (the dimension of the game map are known), individual queries are packed in batches by groups of 200 and sent together to the server for optimality. [21]

In the database, the documents corresponding to a pixel of the game map are structured in two fields: *coords* and *value*. The *coords* field being a compound of the X-coordinate and the Y-coordinate of a pixel, we choose it to be the primary key of the document, thus automatically creating a primary index on all the data concerned by our queries (there is no query involving the color of pixels).
Thanks to this design choice, we are able to exploit the primary key by using the operator *between* instead of a custom filter for the retrieval of pixels within a given area of the game map, and a single *get* operator to update one pixel based on both coordinates.

Since a single query can take some time to be executed, as for the application not to lag behind when the user clicks very fast, the pixel color change queries are bundled into batches when new queries are created while the last one is still waiting. We used a Python Queue object which is thread-safe to store the waiting queries as they are created in case of congestion.

4 Performances

4.1 Benchmarks

Benchmarks run on our own machines, other people's benchmarks for which we could reproduce the results on our machines.

RethinkDB has a report about scaling and performance [10] using for Yahoo Cloud Serving Benchmark (YCSB) [28] [23] which could serve as base to compare with other databases [7]. However most of the technical report work on this seems outdated so we didn't research much further.
Brute write benchmark compared to MongoDB (as both are NoSQL databases with similar query expressiveness) often put MongoDB on top [8], but this is often because MongoDB does not send ack by default on writes, while RethinkDB has four different options and by default uses the slowest (send ack only after finishing writing, do not ignore server reply). However, in the same comparison, RethinkDB was shown to require a lot more disk space to hold the same amount of data.

4.2 Our application under stress

When using queries that filter a specific pixel of the game, a single update queries would take up to two seconds with the drawing robot when the table contained a lot of pixels. However, by retooling the filter into a "get" and fixing the coordinate pair as the primary key, the robot went back to being able to insert pixels immediately. This showed us that working with indexes was very important.

Similarly, asking for a region by doing a filter with four conditions could take considerable time if the region was a lot smaller than the database (because the query had to parse each entry). This is not a problem if the database is sparse in points, but to test the limits of the program we decided that the database would start filled with white pixels. We fixed this by retooling the four filter conditions into a "between" query which works on indexes to isolate the x, then two additional filter conditions for the y. We went back to having immediate results. Unfortunately we cannot use two "between" with different indexes, as it would not make sense (the data being sorted by one or the other), so increasing the database's density in Y would probably make this filter sluggish again.

Creating new connexions is also very time-consuming, sometimes taking five seconds on average. Hence, we re-wrote all our queries to use a few shared connections. Different threads would create queries, but one main thread would maintain the connection and send the queries. To avoid race conditions and batch queries when necessary, we used a thread-safe python queue to solve the producer-consumer problem.

In conclusion a lot of RethinkDB's problems can be traced back to its concistency guarantees which slow down a lot of the writes and the effort it takes to use indexed values, since the primary indexes are by default on a hash key that

is rarely useable by itself, and secondary indexes have to be manually created and take time to be available (for a small 100 x 100 pixels database, it took several minutes to create a secondary index on the y value on a slow computer).

5 Comparison with other databases

5.1 MongoDB with Meteor.js

Meteor is a JavaScript framework built on top of the NoSQL MongoDB which features query update streams. The implementation is called "oplog tailing", and is implemented on top of MongoDB's data replication system, by receiving all write updates destined to the replicates. [5] This technique is similar to RethinkDB's internal broadcast, and scales badly in case of heavy load. [9] [6] When Meteor cannot follow the oplog anymore, it can automatically switch to a polling of the diff of the database at an increased interval instead of following the real-time changes.

5.2 PostgreSQL with listen/notify triggers

It is possible to receive notifications on the changes of a table with listen/notify triggers in PostgreSQL. [25] Of course, this is much lower level and require more work to get advanced results both application and server-side, doesn't take in account sharding etc. So it has nothing in common with RethinkDB's ease of use, but if you can write similar queries, this approach can be several orders of magnitude more efficient.

5.3 Baquend

Baquend is a NoSQL database-as-service currently still in beta which has real-time queries whose structure is inspired by RethinkDB. Baquend separates those queries into resultStream which updates the whole query and eventStream which sends an event for every database write that changes your result. [1] This allows them to extend the type of queries that can be listened compared to RethinkDB's changefeed. [9]

5.4 Firebase

Some commentators said that cloud-based real-time databases were the biggest reason for RethinkDB's downfall. [24] Database services like Firebase [2] allows to synchronize documents between applications, thus allowing database modifications (on a single document) to be pushed in real-time to other users [3], and are built with scaling of large throughput in mind. [4] In Firebase, while you cannot listen to the changes on any advanced query, you can listen to changes on any node of its tree structure. [9] There exist many other databases based on the same concept.

6 Conclusion

In one hand, RethinkDB has several arguments in its favor: it is easy to deploy, manage and use; it boosts a powerful yet simple and secure query language; offers a convenient tool for continuous update of query results; provides an easy interface to shard the database; and most of its complexity is hidden under the hood.
In the other hand, it has some drawbacks among which the apparition of bottlenecks at proxy nodes for big applications, and higher than average memory and RAM consumption. In addition, it does not feature an internal automatic optimization engine and every operation not involving indexes becomes really slow, thus requiring users to be careful when writing their queries. There is no integrated way to manage data backups either.
In conclusion, RethinkDB works best when used in quick mid-sized real-time projects but can get costly on bigger projects due to the extra spending on proxy nodes required to keep things working. [25]

As for the future, in 2016, RethinkDB's company filed bankruptcy and all its code was released. Today, it is an open-source project managed by the Linux Foundation. [12] About the failure of RethinkDB as a product, its developer stated that they focused on the wrong metrics: strict data correctness guarantees, simplicity of interface hiding the complexity of the application, and consistency in the documentation, query language and drivers, while being late to the market and not having high performances or a feature. [26] While RethinkDB is a competent database overall, there exist more popular databases that can efficiently solve the particular problems it addresses.

References

[1] Baqend guide: Real-time queries. https://www.baqend.com/guide-next/topics/realtime/. Accessed: 2017-12-20.

[2] Firebase documentation. https://firebase.google.com/docs/database/. Accessed: 2017-12-16.

[3] Firebase documentation: Read and write data on the web. https://firebase.google.com/docs/database/web/read-and-write. Accessed: 2017-12-20.

[4] Firebase documentation: Realtime database limits. https://firebase.google.com/docs/database/usage/limits. Accessed: 2017-12-20.

[5] Meteor documentation: Oplog observer driver. https://github.com/meteor/docs/blob/version-NEXT/long-form/oplog-observe-driver.md. Accessed: 2017-12-20.

[6] Meteor github issue: Large number of operations hangs server. https://github.com/meteor/meteor/issues/2668. Accessed: 2017-12-20.

[7] Mongodb files for yahoo! cloud system benchmark (ycsb). https://github.com/mongodb-labs/YCSB/tree/master/ycsb-mongodb. Accessed: 2017-12-16.

[8] Mongodb vs rethinkdb: Benchmarks. https://www.amon.cx/blog/rethinkdb-reviewed-by-a-mongo-fan/. Accessed: 2017-12-16.

[9] Real time databases explained: why metero rethinkdb parse and firebase don't scale. https://medium.baqend.com/real-time-databases-explained-why-meteor-rethinkdb-parse-and-firebase-dont-scale-822ff87d2f87. Accessed: 2017-12-16.

[10] RethinkDB Blog: rethinkdb 2.1.5 performance & scaling report. https://rethinkdb.com/docs/2-1-5-performance-report/. Accessed: 2017-12-16.

[11] Rethinkdb blog: Advancing the realtime web. https://www.rethinkdb.com/blog/realtime-web/. Accessed: 2017-12-20.

[12] Rethinkdb blog: Rethinkdb joins the linux foundation. https://www.rethinkdb.com/blog/rethinkdb-joins-linux-foundation/. Accessed: 2017-12-21.

[13] Rethinkdb documentation: Architecture faq. https://www.rethinkdb.com/docs/architecture/. Accessed: 2017-12-20.

[14] Rethinkdb documentation: Backing up your data. https://rethinkdb.com/docs/backup/. Accessed: 2017-12-20.

[15] RethinkDB Documentation changefeeds in python. https://www.rethinkdb.com/docs/changefeeds/python/. Accessed: 2017-12-16.

[16] Rethinkdb documentation: Consistency guarantees. https://rethinkdb.com/docs/consistency/. Accessed: 2017-12-20.

[17] Rethinkdb documentation: Geospatial queries. https://www.rethinkdb.com/docs/geo-support/python/. Accessed: 2017-12-20.

[18] Rethinkdb documentation: Limitations in rethinkdb. https://rethinkdb.com/limitations/. Accessed: 2017-12-20.

[19] Rethinkdb documentation: Map-reduce in rethinkdb. https://rethinkdb.com/docs/map-reduce/. Accessed: 2017-12-20.

[20] Rethinkdb documentation: Scaling, sharding and replication. https://www.rethinkdb.com/docs/sharding-and-replication/. Accessed: 2017-12-20.

[21] Rethinkdb documentation: Troubleshooting common rethinkdb problems. https://rethinkdb.com/docs/troubleshooting/. Accessed: 2017-12-21.

[22] Rethinkdb documentation: Using secondary indexes in rethinkdb. https://www.rethinkdb.com/docs/secondary-indexes/python/. Accessed: 2017-12-20.

[23] Rethinkdb files for yahoo! cloud serving benchmark. https://github.com/rethinkdb/ycsb. Accessed: 2017-12-16.

[24] Rethinkdb is dead, but not because mongodb. instead, the cloud is to blame. https://www.techrepublic.com/article/rethinkdb-is-dead-and-mongodb-isnt-what-killed-it/. Accessed: 2017-12-20.

[25] Rethinkdb versus postgresql: my personal experience. https://blog.sagemath.com/2017/02/09/rethinkdb-vs-postgres.html. Accessed: 2017-12-16.

[26] Rethinkdb: why we failed (post-mortem). http://www.defmacro.org/2017/01/18/why-rethinkdb-failed.html. Accessed: 2017-12-20.

[27] What is a document store database?
http://database.guide/what-is-a-document-store-database/7.
Accessed: 2017-12-16.

[28] Yahoo cloud service benchmark.
https://research.yahoo.com/news/yahoo-cloud-serving-benchmark.
Accessed: 2017-12-16.