Community Experience Distilled

Mastering RethinkDB

Master the capabilities of RethinkDB and implement them to develop efficient real-time web applications

Shaikh Shahid

[PACKT] PUBLISHING open source* community experience distilled

Table of Contents

1
RethinkDB Architecture and Data Model

RethinkDB is a real time open source and distributed database. It stores JSON document (Basically unstructured data) in operable format with distribution (Sharding and Replication). It also provides the real time **"push"** of JSON data to the Server which re-defines the whole real time web application development.

In this chapter we will look over the Architecture and key elements of it in order to understand how RethinkDB supports these awesome features with high performance. We will also look over Data modelling along with SQL operations in NoSQL i.e Joins.

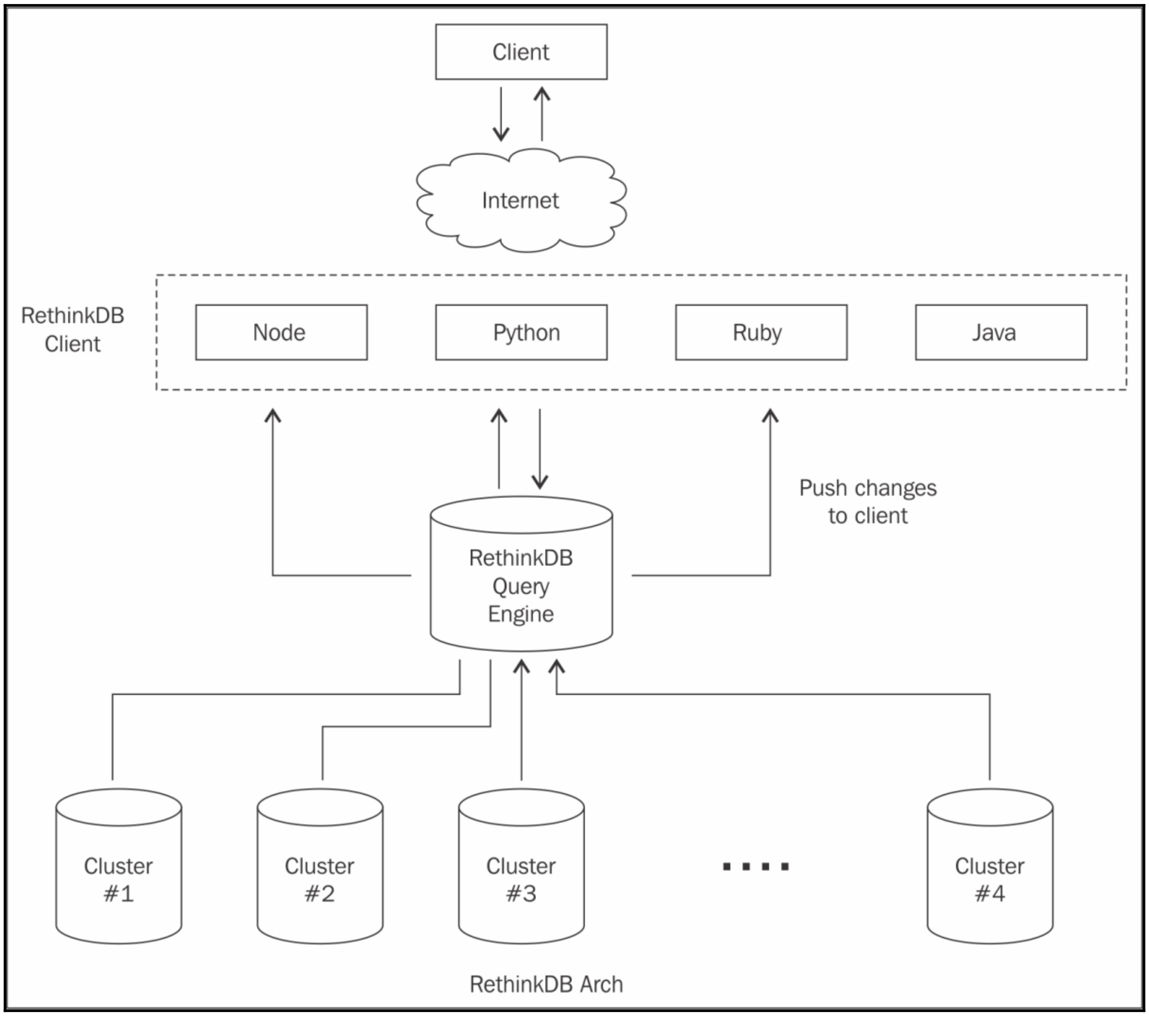
Here is the list of topics we are going to cover in this chapter.

1. RethinkDB Architectural components.
2. Sharding and replication in RethinkDB.
3. RethinkDB failover handling.
4. RethinkDB data model.
5. Data modelling in RethinkDB.

RethinkDB Architectural components

RethinkDB Architecture consists of various components such as cluster, query execution engine, file system storage, push changes (real time feed) and of course RethinkDB client drivers.

Refer below diagram to understand the block level components of RethinkDB.



Client drivers

RethinkDB provides client drivers for Node.js, Python, Ruby and Java. At the time of writing the book only these languages were supported. In this book we will refer code examples with Node.js.

RethinkDB Query Engine

RethinkDB query handler as name implies performs the Query execution and return back the response to the client. It does so by performing lot of internal operations such as sorting, indexing, finding the cluster or merging data from various clusters. All of these operations are performed by RethinkDB query handler. We will look over this in detail in upcoming section.

RethinkDB Clusters

RethinkDB is distributed database designed for high performance real time operations. RethinkDB manages the distribution by clustering (Sharding OR Replication). RethinkDB cluster are just another instance of main process of RethinkDB and stores the data.We will look over Sharding and Replication in detail in upcoming section.

Pushing changes to RethinkDB client

This is revolutionary concept introduced by RethinkDB. Consider this scenario; you are developing an application for stock market where there are too many changes in given amount of time. Obviously we are storing each and every entry in the database and making sure that other connected nodes or clients know about these changes.

In order to do so, conventional way is to keep looking (Polling) for the data in the particular collection or table in order to find some changes. This improves the latency and turn around time of packets and we all know Network call in WAN is really costly.

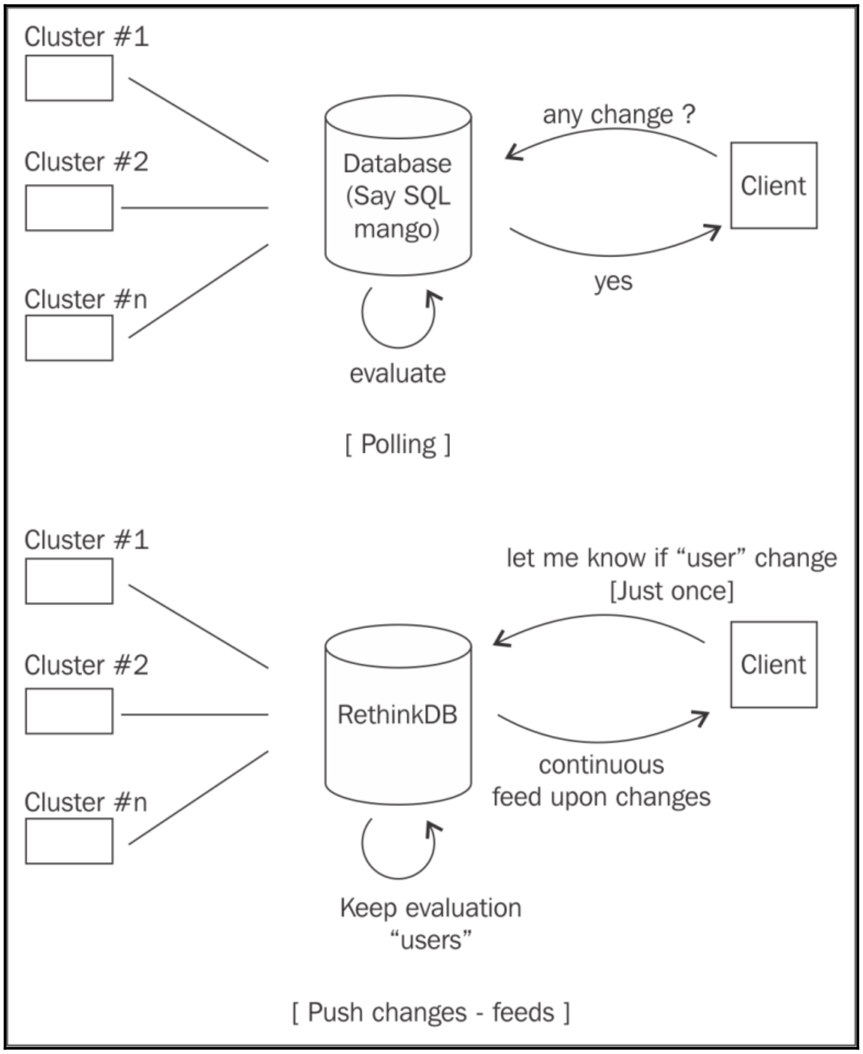
Then came something called Socket, in this we do the Polling operation but from socket layer not HTTP layer. Here the size of network request may get reduced but still we do the Polling.

Note: Socket.io is one of popular project available real time web development.

RethinkDB propose a reverse approach of it, what about the database itself tells you **“Hey there are some changes happen in stock value and here is new and old value”.**

This is exactly what RethinkDB push changes (change feed in technical term) does, once you subscribe to particular table to look for its changes, RethinkDB just keep pushing the old and new value of change happen to connected client, by connected client I meant

RethinkDB client not a web application client. The difference between polling and push changes is shown below:



So you will get the changes data in one of RethinkDB client say Node.js and then you can simply broadcast it over the network using Socket probably.

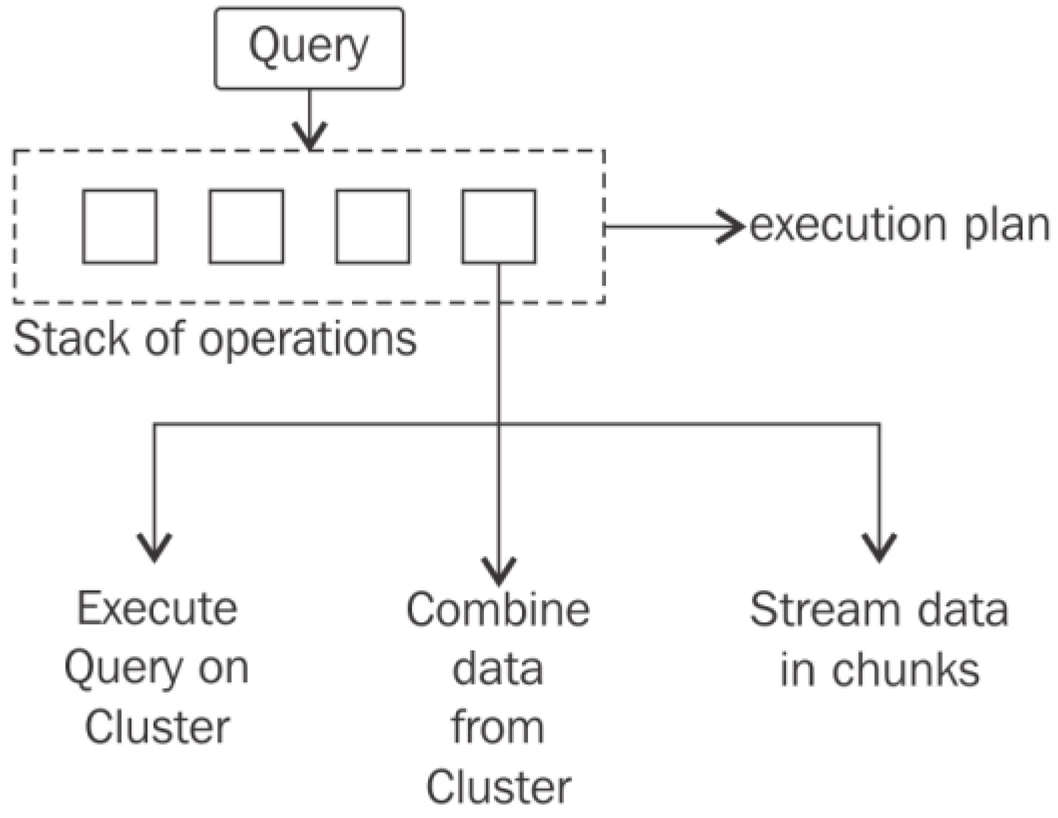
But why are we using the Socket when RethinkDB can provide us changes of data? Because RethinkDB provides it to middle-layer not to client layer, having client layer directly talk to client can be risky and hence it is not allowed yet.

But, RethinkDB team is working on another project called **"horizon"** which solvesthe issue mentioned above, to allow client communicate to database using secure layer of middle-tier. We will look over about **Horizon** in chapter 10, *"Using RethinkDB and horizon "* in detail.

Query execution in RethinkDB

RethinkDB query engine is very critical and important part of it. RethinkDB performs various computation and internal logic operation to maintain the high performance along with good throughput of system.

Refer below diagram to understand query execution.



[fig Query execution]

RethinkDB upon arrival of query, divide it into various stacks. Each stack contains various methods and internal logic to perform its operation. Every stack consists of various methods but there are three core methods which play key role.

1. First method decides how to execute the query or subset of query on each server in particular cluster.

1. Second method decides how to merge the data coming from various clusters in

order to make sense of it.

1. Third method which is very important deals with transmission of those data in streams rather than as whole.

To speed up the process, these stacks are transported to every related server and each server begins to evaluate it in parallel to other servers. This process runs recursively in order to merge up the data to stream to the client.

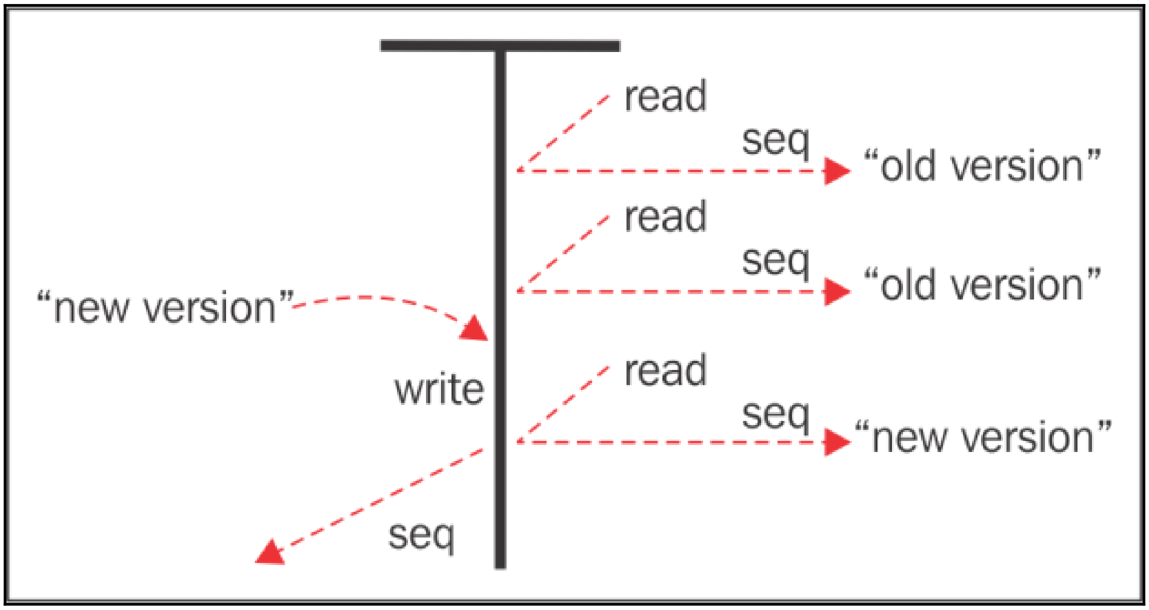
Stack in the node grabs the data from the stack below it and performs its own methods execution and transformation. The data from each server is then combined into a single result set and stream to the client.

In order to speed up the process and maintains high performance, every query is completely parallelized across various relevant clusters. Every cluster thus then perform the query execution and then data is again merged together to make single result set.

RethinkDB query engine maintains the efficiency in the process too, for .eg if client only request certain result which is not in shared or replicated server, it will not execute the parallel operation and just return the result set. This process is also referred as Lazy execution.

To maintain the concurrency and high performance of query execution, RethinkDB uses block-level multi-version concurrency control (MVCC). If one user is reading some data while other users are writing on it, there is a high chance of inconsistent data and to avoid that we use concurrency control algorithm. One of the simplest way and commonly used method by SQL databases is to "lock" the transaction i.e. make the user wait if some write operation is being performed on the data. This slows down the system and since big data promises fast reading time, this just simply won't work.

Multiversion concurrency control (MVCC) takes the different approach, here each user will see the snapshot of the data i.e. child copies of master data and if there are some changes going on in the master copy then child copies or snapshot will not get updated until the change has been committed.



RethinkDB does use block-level MVCC and this is how it works. Whenever there is any update or write operation is being performed during the read operation, RethinkDB takes a snapshot of each shard and maintains different version of a block to make sure every read and write operation works parallelly. RethinkDB does use exclusive locks on block-level in a case of multiple updates happening on the same document. These locks are very short in duration because they all are cached hence it always seems to be lock-free.

RethinkDB provides atomicity of data as per JSON document. This is different than other NoSQL system, most of NoSQL systems provides atomicity to each small operation done on the document before the actual commit. RethinkDB does the opposite, it provides atomicity to a document no matter what combination of operations is been performed.

For.eg user may want to read some data say the first name from one document, change it to upper case, append the last name coming from other JSON document and then update the JSON document. All these operations will be performed automatically in one update operation.

RethinkDB limits this atomicity to few operations, For.eg results coming from JavaScript code cannot be performed atomically. The result of a subquery is also not atomic, replace cannot be performed atomically.

File system and data storage

RethinkDB supports major used file systems such as NTFS, EXT etc. RethinkDB also supports direct I/O file system for efficiency and performance but it is not enabled by default.

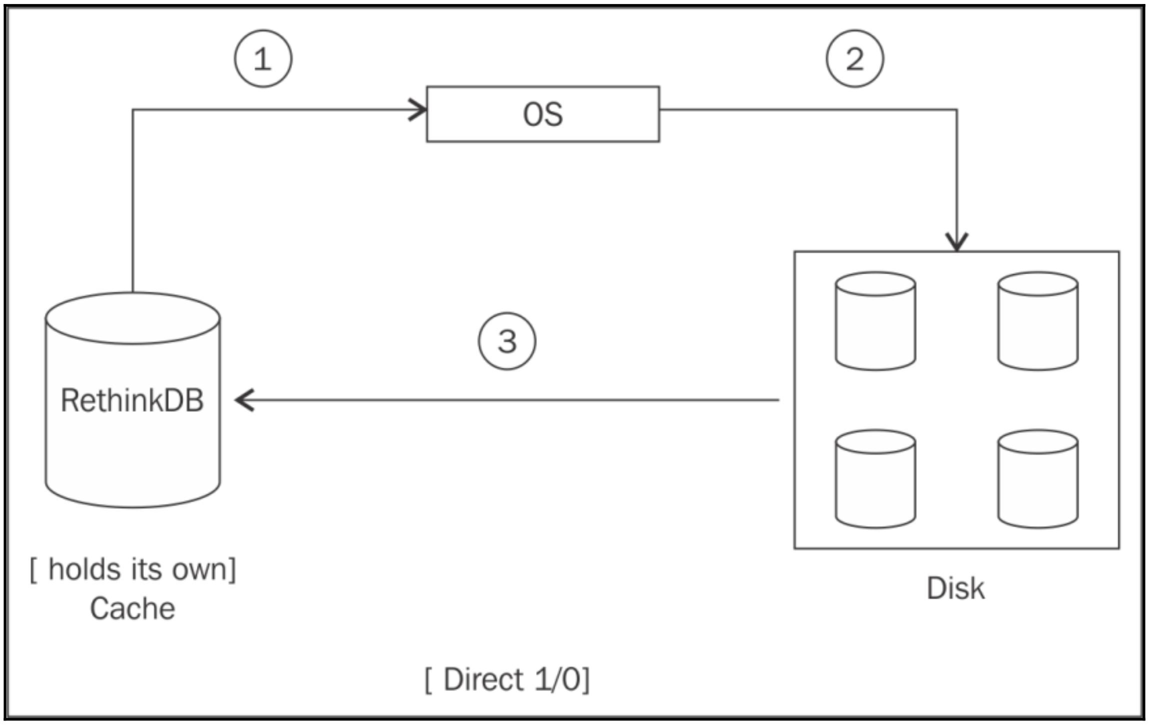
About Direct I/O

File is stored on disk and when it's been requested by any program, operating system first put it into Main Memory for faster read. Operating system can read directly from Disk too but that will slow down the response time because of heavy cost I/O operation. Hence operating system first put it into Main Memory for operation. It's called **"Buffer cache".**

Databases generally manage data caching at the application and they do not need operating system to cache it for them. In such cases, the process of buffering at two places (Main Memory and application cache) becomes overhead since data is first moved to Main Memory and then application cache.

This **"double buffering"** of data results in more CPU consumption and load on Memory too.

Direct I/O is a file system for those application which wants to avoid the buffering at Main memory and directly read files from disk. When Direct I/O is used, data is transferred directly to Application buffer instead of Memory buffer as shown in the below image.



Direct I/O can be used in two ways.

1. Mounting the file system using Direct I/O (Options vary from OS to OS).
2. Opening the file using **O_DIRECT** option specified in **open()** system call.

Direct I/O provides great efficiency and performance by reducing CPU consumption and overhead of managing two buffers.

Data Storage

RethinkDB uses custom built storage engine inspired by BTRFS – Binary tree file system by Oracle. There is not enough information available on RethinkDB custom file system right now, but we have found following promises by it.

1. Fully concurrent garbage compactor.
2. Low CPU overhead.
3. Efficient multi-core operation.

4. SSD optimization.
5. Power failure recovery.
6. Data consistency in case of failures.
7. MVCC supports (Multiversion concurrency control).

Due to these features, RethinkDB can handle large amount of data in very less amount of Memory storage.

Sharding and Replication

Sharding is partitioning where database is splitted across multiple, smaller databases to improve the performance and reading time. In Replication we basically copy the database across multiple database to provide quicker look and fast response time. Content delivery networks are the best examples of this.

RethinkDB, just like other NoSQL databases also uses Sharding and Replication to provide fast response and greater availability. Let's look over it in detail one by one.

Sharding in RethinkDB

RethinkDB makes use of range sharding algorithm to provide sharding feature. It performs the sharding on the table's primary key to partition the data. RethinkDB uses table's primary to perform all sharding operation and it cannot use any other keys to do so. In RethinkDB, shard key and primary key are same.

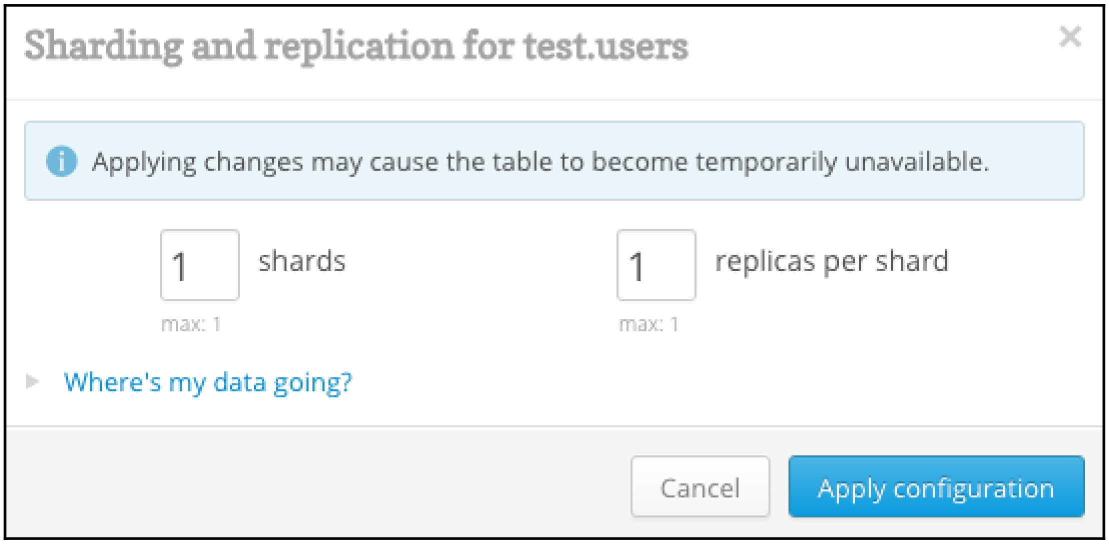
Upon request to create a new shard for particular table, RethinkDB examines the table and tries to find out the optimal break point to create **"even"** number of shards.

For eg. Say you have a table with 1000 rows, primary key ranging from 0 to 999 and you asked RethinkDB to create two shards for you.

RethinkDB will likely find primary key 500 as a breaking point. It will store every entry ranging from 0 to 499 in shard-1, while data with primary key 500 to 999 will be stored in shard-2. Shards will be distributed across clusters automatically.

You can specify the sharding and replication settings at the time of creation of table or alter it later. You cannot specify the split point manually, that's RethinkDB do internally. You cannot have less Server than you shard.

You can always visit the RethinkDB administrative screen to increase the shards or replicas.


Sharding and replication for test.users

Applying changes may cause the table to become temporarily unavailable.

1 shards
max: 1

1 replicas per shard
max: 1

We will look over this more in detail with practical use cases in chapter 5 totally focussed on RethinkDB administration.

Let's look more into detail about how range based sharding works.

Sharding can be basically done in two ways, using vertical partitioning or horizontal partitioning.

1. In vertical partitioning, we store data in different tables having different document in different databases.
2. In horizontal, we store documents of same table in separate databases. Range shared algorithm is dynamic algorithm which determines the breaking point of table and stores data in different shards based on the calculation.

Range Based Sharding

In range sharding algorithm we use a service called **"locator"** to determine the entries in particular table. The locator services finds out the data using range queries and hence it becomes faster than others. If you do not have range or some kind of indicator to know that which data belongs to which shard in which Server, you will need to look over every database to find the the particular document, which no doubt turns into very slow process.

RethinkDB maintains relevant piece of metadata which they refer to as **"directory".**The directory maintains a list of node (RethinkDB instance) responsibilities for each shard. Each node is responsible for maintaining the updated version of directory.

RethinkDB allows users to provide the location of shards.You can again go to web based administrative screens to perform the same. However, you need to setup the RethinkDB Servers manually using command line and it cannot be done via web based interface.

Replication in RethinkDB

Replication provides the copy of data in order to improve performance, availability and failover handling. Each shard in RethinkDB can contain configurable number of replicas. RethinkDB instance (Node) in the cluster can be used as replication node for any shard. You can always change the replication from RethinkDB web console.

Currently, RethinkDB does not allow more than one replica in single RethinkDB instance due to some technical limitation. Every RethinkDB instance stores the metadata of the tables. In case of changes in metadata, RethinkDB sent those changes across other RethinkDB instance in the cluster in order to keep the updated metadata across every shard and replica.

Indexing in RethinkDB

RethinkDB uses primary key by default to index the document in table. If user does not provide the primary key information during the creation of table, RethinkDB uses its default named **id.**

Default generated primary key contains information about the shard location in order to directly fetch the information from appropriate shard. Primary key of each shard is indexed using B-Tree data structure.

One of the example for RethinkDB primary key is as follows.

D0041fcf-9a3a-460d-8450-4380b00ffac0

RethinkDB also provides secondary key and compound keys (combination of keys) features. It also provides Multi indexes features that allow you to have arrays of values acting as keys which again can be single, compound keys.

Having system generated keys for primary is very efficient and fast, because the query execution engine can immediately determines on which shard the data is present and hence need no extra routing while having custom primary key say an Alphabet or a number may force RethinkDB to perform more searching of data on various clusters hence slowing down the performance. You can always use secondary keys of your choice to perform further indexing and searching based upon your application need.

Automatic failover handling in RethinkDB

RethinkDB provides automatic failover handling in multi-server configuration where multiple replicas of table are present. In case of node failure due to any reason, RethinkDB finds out the other node to divert the request and maintain the availability. However, there are some requirements that must be met before considering Automatic failover handling.

1. The cluster must have three or more node (RethinkDB Servers).
2. The table must be set to have three or more replicas set with '**voting**' option**.**
3. During failover, majority of replicas (greater than half of all replicas) for the table must be online.

Every table by default has primary replica created by RethinkDB. You can always change that using **reconfigure ()** command. In case of failure of primary replica of table, as long as more than half of replicas with voting option are available, one of them internally will be selected as primary replica. There will be slight offline scenario while the selection is been going on in RethinkDB but that will be very minor and no data will be lost.

As soon as primary replica gets online, RethinkDB automatically sync it with latest documents and switch the control of primary replica to it automatically.

About voting replicas

By default, every replica in RethinkDB is created with as 'voting' replicas. That means, those replicas will take part in the failover process to perform the selection of next primary replica. You can also change this option using **reconfigure ()** command.

Automatic failover requires at least three server clusters with three replicas for table. Two server clusters will not be covered under automatic failover process and system may go down during the failure of any RethinkDB instance.

In such cases where RethinkDB cannot perform the failover, you need to do it manually using the **reconfigure ()** command by passing emergency repair mode key.

Upon running the emergency repair mode, each of the shards are first examined and classified into three categories.

1. If more than half of total shards are available, it will return **Healthy** status
2. Repairable: In case of **Repairable**, shard is not healthy but has one replica which can be used.
3. Beyond repair.: In case of **Beyond repair,** shard has no available replica and cannot be used

For each and every shard which can be repaired, RethinkDB will first change all of the offline replicas into non voting replicas. If there is not voting replica available, RethinkDB will choose one non voting replica and forcefully convert it into voting replica.

You can specify two options along with emergency repair option.

- **unsafe_rollback It** will leave those shards which are beyond repair during the failover process
- **unsafe_rollback_or_erase It** will delete those shards which are beyond repair and create one on available server that holds another shard for that table.

Here is the command reference.

```
    r.table(users).reconfigure(   {emergencyRepair:
"unsafe_rollback_or_erase"} ).run(conn, callback);
```

Please make a note that emergency failover handling is very critical and should be done by skilled person else you may end up loosing all the data.

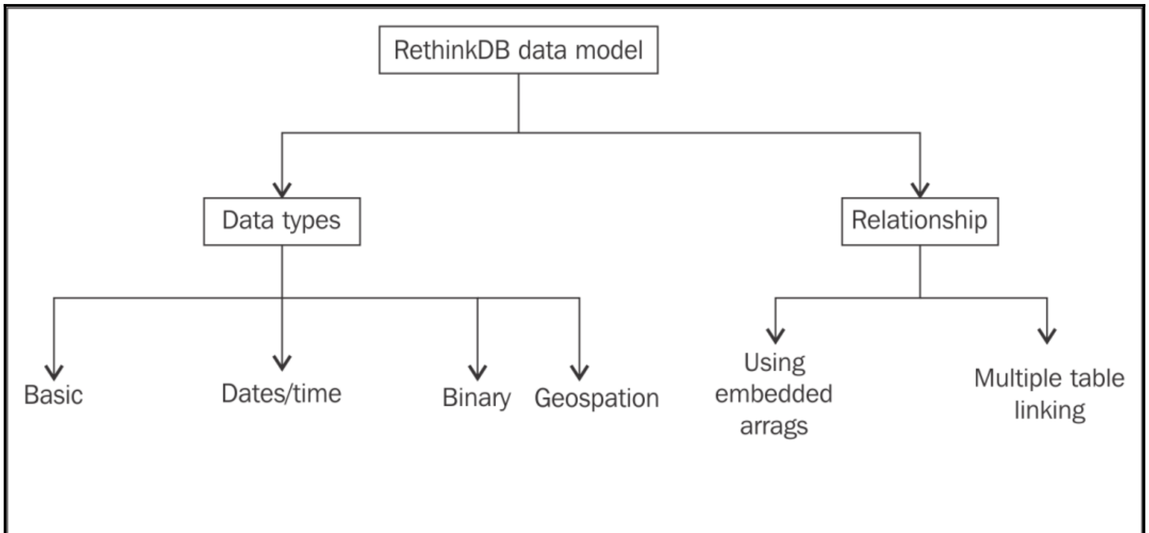
RethinkDB data model

RethinkDB data model consists of two main components.

1. RethinkDB data types.
2. RethinkDB model relationship.

RethinkDB data types are further classified into basic data types, dates, binary object and geospatial queries.

Refer below diagram for more detail.



[RethinkDB data model]

Let's go over to each component in detail.

RethinkDB data types

RethinkDB provides six basic data types, they are represented into tabular format as shown below:

Data Types	**Description**
Numbers	Numbers are stored using double precision floating point numbers.
Strings	Strings are stored in UTF-8 standard.
Objects	It is stored as key value pair, standard JSON format.
Arrays	Stored as list of elements. It supports 100,000 elements by default.
Binary	Binary object includes file, images and other binary data.
Date and time	RethinkDB stores date and time in millisecond precision.
Booleans	True and false
Null	Stores null values.

RethinkDB also provides extended data types supported by RethinkDB. They are as follows:

1. **Tables**: They are RethinkDB tables. You can insert and delete documents in them with proper indexing.
2. **Streams:**Streams provide transfer of large amount of data in chunks so that System should not reach the buffer overflow limit. They are loaded in lazy fashion. Streams provide the navigation point to a chunk called **"cursor"** which you can use to traverse over result set. You can club all of them once the streams are collected using cursor. This makes it easy to read large amount of data. Streams are read only operation.
3. **Selections**: Selections are subset of data such as 10 documents out of 1000 from table. There are two types of Selections, one with object and one with streams. The difference between two is that object can be writable while streams are read only. You can pass object selected by say **get ()** command to other commands to manipulate the data.
4. **Date and time**: Date and time are stored in RethinkDB with millisecond precision along with time zone. Currently minute-precision time offsets from UTC are supported. RethinkDB internally calculate time zone difference hence you don't need to do it in client end.

You can use native date commands supported by RethinkDB drivers such as in Node.js, you can use **Date()** object.

Binary objects

Binary objects are stored similar to BLOB in SQL databases. You can directly read and upload the files, images etc directly into database, Parsing and other dirty task will be dealt by RethinkDB.

One of the amazing functionality of RethinkDB is calling external api's from RethinkDB native drivers. So consider you want to add profile pictures of users directly into database from Gravatar – one common place of avatars, all you need to do is call the api and convert the detail into binary. Consider below official code snippet.

```
    var hash = md5(email);     gravatarUrl = 'http://gravatar.com/avatar/'
+ hash + '?d=retro';     r.table('users').get(userId).update({
gravatar: r.http(gravatarUrl, {resultFormat: 'binary'})     }).run(conn,
callback)
```

Assuming `conn` is RethinkDB connection and we are passing `email` of user to get the avatar. If you notice we are calling gravatar api using **http ()** function and converting result format into binary..

Specifying result format is not mandatory here, if MIME type of calling server is set correctly you can just call the HTTP url and it will be converted to default MIME type say binary. It's better to be on safe side though.

Geospatial queries in RethinkDB

RethinkDB provides and support geospatial features to help you build location based application. By using geospatial queries you can easily parse, convert and perform lots of operations such as computing distance between two locations, finding intersecting location and many more. RethinkDB stores all geographical information in GeoJSON standard format.

RethinkDB uses geometry object to perform geographical queries. Geometry objects are derived by plotting two dimensional objects such as lines, points on the sphere in three dimensional spaces.

RethinkDB stores information in standard geographic system which is addressing a point on a surface in **longitude** and **latitude**. It does not support Elevation yet. The range of longitude is -180 to 180 and range of latitude is -90 to 90. To store same we use **point ()** function of RethinkDB. Here is sample code, assuming **r** is RethinkDB object.

```
r.table('demo').insert([
 {
  name : "Mumbai",
  location : r.point(19.0760,72.8777)
 }
])
```

Supported data types

Geospatial data types and functions are derived from three geometric object data types such as points, lines and polygon

By using these three, RethinkDB provides various Geospatial functions such **circle (), intersect (), getNearest()** etc.

RethinkDB model relationship

RethinkDB, besides being NoSQL database provides one of the most requested features by SQL developers, that is JOINS. RethinkDB allows you to model and structure your data in such a way that allows you to perform JOINS over it. There are two ways to model relationship in RethinkDB.

1. By using Embedded arrays in the document.
2. By linking documents stored in multiple tables.

Let's see how both of them work along with their merits and demerits.

Embedded Arrays

Embedded arrays are basically using the array of objects in the table in each document. Here is simple example.

```
{   "id": "7644aaf2-9928-4231-aa68-4e65e31bf219",   "name": "Shahid",
"chapters": [     {"title": "Chapter 01", "pages": "30+"},     {"title":
"Chapter 02", "pages": "30+"}   ] }
```

Here, `[chapters]`key contains the array of objects, and each object contains name of chapter and pages count. In order to perform the query, here is how we do in RethinkDB.

To display all chapters from table.

```
r.db("book").table("chapters").run()
```

To get all the chapters written by particular author, we can do that by passing the author id.

```
r.db("book").table("chapters").get('7644aaf2-9928-4231-
aa68-4e65e31bf219').run()
```

Here we are passing the id of document which is system generated primary key id by the RethinkDB. Upon passing it, we can access single document from the table. This is very basic way of modelling and not recommended for production uses.

Merits of embedded arrays

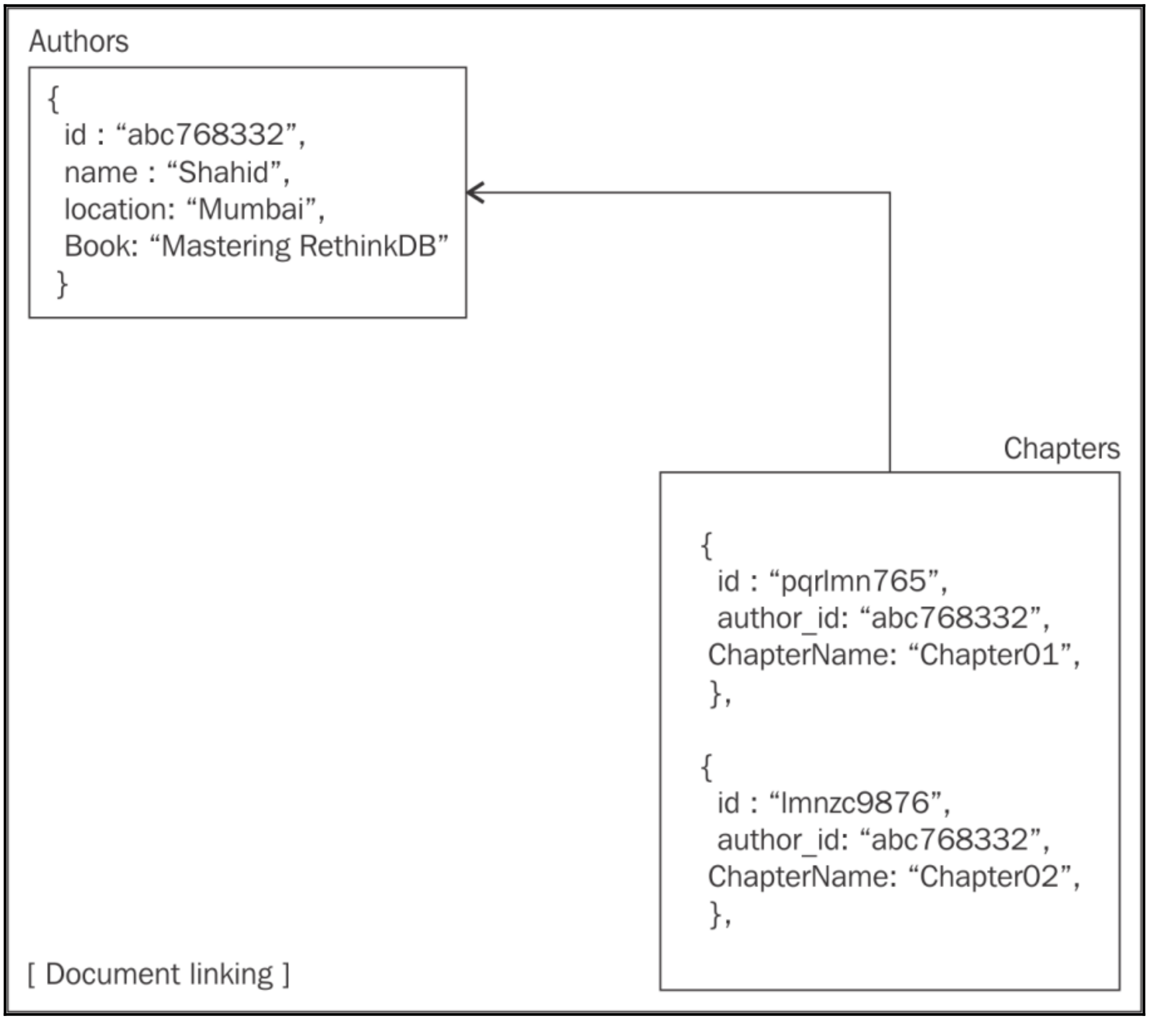
1. Easy to design and query.
2. Updates on single document will automatically updates chapters and authors both. Hence both data will be atomic.

Demerits of embedded arrays

1. As soon as size of embedded array exceeds, it costs so much computing work to load entire array every time when we have to perform any operation.

Document linking in multiple tables

This is similar to foreign key relationship we do in traditional SQL databases. Here we link two or more tables using the index. We can also query the table using functions provided by RethinkDB to perform various joins (Inner, outer etc.).



As shown in above diagram, we have two tables with multiple documents in it. Table **"Authors"** will just contain documents related to Authors, say name, location etc and obviously a primary key.

Table **“chapters”** will contain documents of chapters; each chapter is separated document instead of an array. The only difference is, each document will contain an index with same value as primary key from an **“Author's”** table. This way we link both of them together.

As you can see in diagram, there is a key in every document in “**chapters**” table named “**author_id**” with same value as **id** in “**Authors**” table.

Here is how we query the tables to perform JOINS.

```
r.db("books").table("chapters").   filter({"author_id": "abc768332"}).
run()
```

This will return every chapter for Author having ID as **abc768332.**

You can also use RethinkDB JOINS functions such as **eq_join(),** will look over it next section.

Merits of document linking:

1. Data distribution and very neat design.
2. No need to load all of the data of **chapters** at once in order to perform any operation.
3. No matter the size of documents in **chapters** table, performance won't be affected as it affects in embedded arrays.

Demerits of document linking:

1. No foreign key constraints, so linking of data will be complicated.
2. In case of update in one table, it won't automatically update the data in another table.

Here is the list of JOIN commands provided by RethinkDB.

1. eq_Join.
2. innerJoin.
3. outerJoin.
4. Zip.

Before looking over it, let's populate a data in two tables. We have **authors** and **chapters** in **books** database.

We will adding author detail in **authors** table.

```
r.db('books').table('authors').insert({name : "Shahid", location :
"Mumbai"})
```

We will be adding chapter details in **chapters** table.

```
r.db('books').table('chapters').insert({
   author_id : "521b92b1-0d83-483d-a374-b94a400cf699",
   chapterName : "Chapter 1"
})

r.db('books').table('chapters').insert({
   author_id : "521b92b1-0d83-483d-a374-b94a400cf699",
   chapterName : "Chapter 2"
})
```

Performing eq_Join

```
r.db('books').table('chapters').eqJoin("author_id",r.db('books').table('aut
hors'))
```

Here we are joining **chapters** table with **authors** table and mentioning which field we have mapped to ID of authors i.e `author_id`**.**

It shall return following.

```
[
   {
      "left":{
         "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
         "chapterName":"Chapter 1",
         "id":"f0b5b2f7-1f82-41ef-a945-f5fa8259dd53"
      },
      "right":{
         "id":"521b92b1-0d83-483d-a374-b94a400cf699",
         "location":"Mumbai",
         "name":"Shahid"
      }
   },
   {
      "left":{
         "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
         "chapterName":"Chapter 2",
         "id":"f58826d4-e259-4ae4-91e4-d2e3db2d9ad3"
      },
      "right":{
         "id":"521b92b1-0d83-483d-a374-b94a400cf699",
         "location":"Mumbai",
         "name":"Shahid"
      }
   }
]
```

Left and **right** key supposedly represent the tables on left side of query and right side of query. If you map the keys and value with the ReQL query, you can easily understand.

However, this is not how we really want our data. We want processed, result oriented data and to do that we need to use **zip ()** command. It basically removes all metadata information from the result and gives you only the documents of tables.

```
r.db('books').table('chapters').eqJoin("author_id",r.db('books').table('aut
hors')).zip()
```

It shall return following.

```
[
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 1",
      "id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "location":"Mumbai",
      "name":"Shahid"
   },
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 2",
      "id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "location":"Mumbai",
      "name":"Shahid"
   }
]
```

Performing inner join

Inner join as we all know equate two tables by comparing each row. It is very similar in working with EQJOIN except it compares each document in table against the target table while in EQJOIN we specify which key to compare for.

Here is simple query to perform inner join.

```
r.db('books').table('chapters').innerJoin(r.db('books').table('authors'),fu
nction(chapters,authors) {
  return chapters
}).zip()
```

This function takes target table as a parameter and callback function which contains data of both the tables in callback argument. If you notice, for understanding i named callback parameters same as table name. You can perform lot of other operation like comparison, filtering inside the callback function and then return the result. Since it's a JOIN, in both of the variable data will be similar except with different table id's.

Here is the result for same.

```
[
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 1",
      "id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "location":"Mumbai",
      "name":"Shahid"
   },
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 2",
      "id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "location":"Mumbai",
      "name":"Shahid"
   }
]
```

If you observe, result set is pretty much same as EQJOIN (eq_Join() function) except it provides you result of each document under callback function. This makes it really slow and RethinkDB team does not recommend it to use in production.

Performing outer join

Outer join union the result of left join and right join and returns it to client. So basically result set from both of the table will be combined and return back. Here is sample query.

```
r.db('books').table('chapters').outerJoin(r.db('books').table('authors'),fu
nction(chapters,authors) {
  return authors
}).zip()
```

This will combine the each document of chapters with each document of authors and returns back the result. Again, we can access each document of the query here under callback.

It shall return following.

```
[
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 1",
      "id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "location":"Mumbai",
      "name":"Shahid"
   },
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 2",
      "id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "location":"Mumbai",
      "name":"Shahid"
   }
]
```

In order to check the working of it, let's just create one more author in **authors** table and do not create the chapter document entry for it.

```
r.db('books').table('authors').insert({name : "RandomGuy", location :
"California"})
```

Upon running the outer join query again, here is the result.

```
[
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 1",
      "id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "location":"Mumbai",
      "name":"Shahid"
   },
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 2",
      "id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "location":"Mumbai",
      "name":"Shahid"
   },
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 1",
      "id":"78acabb5-a5b8-434b-acb1-52507b71831d",
      "location":"California",
      "name":"RandomGuy"
   },
   {
      "author_id":"521b92b1-0d83-483d-a374-b94a400cf699",
      "chapterName":"Chapter 2",
      "id":"78acabb5-a5b8-434b-acb1-52507b71831d",
      "location":"California",
      "name":"RandomGuy"
   }
]
```

Hence we get all of the result, union of each document of table present in left table i.e **Authors** with each document present in right table i.e **chapters.**

Zip

This function performs the merging of **left** fields with **right** fields comes in JOIN operation into single data set.

Constraints and limitation in RethinkDB

We have covered various architectural features and data model of RethinkDB. Let's look over some of the constraints of RethinkDB which you need to take into account while architecting your data store.

RethinkDB divides the limitation into **hard** and **soft** limitation. Hard limitations are as follows.

1. Creation of the number of databases has no hard limit.
2. There is limit of shard creation which is maximum of 64 hards.
3. Creation of tables inside the database has no hard limit.
4. Storage size of single document has no hard limit. (However it is recommended to limit it under 16 MB for performance).
5. Maximum size of Query is 64MB.

RethinkDB also has some memory limitation, they are as follows.

1. An empty table will takes up to 4MB.
2. Each table after population of documents requires at least 10MB of disk space on each server wherever it is replicated in the cluster.

3. Each table consumes 8MB of RAM on each server.

RethinkDB, in order to keep the performance high stores some data in RAM. There are basically three source of usage of RAM by RethinkDB.

1. Metadata.
2. Page cache.
3. Running Queries and background process.

RethinkDB stores meta data of tables in main memory in order to ensure fast read access. Every table consumes around 8MB per server for the meta data. RethinkDB organizes the data into blocks with size ranging from 512 Byte to maximum 4 KB. Out of these blocks, approx 10 to 26 bytes per block is kept in memory.

Page cache is very important aspect of performance. It is basically used for storing very frequent accessible data in the RAM rather than reading it from disk (Except in the case of Direct I/O where page cache is in Application buffer than in RAM). RethinkDB uses this formula to calculate the size of the cache.

Cache size = available memory – 1024 MB / 2.

If cache size is less than 1224 MB then RethinkDB set the size of page cache to 100 MB. This is why it is recommended to at least have 2GB of ram allocated for RethinkDB process.

You can also change the size of page cache when you start the Server or later using configuration files.

Every database uses some memory to store results of ongoing running queries. Since queries differ in general there is no exact estimate about the memory usage by running queries; however a rough estimate is between 1 to 20 MB including the background processes such as transferring data between nodes and voting process etc.

Summary

RethinkDB is indeed a next generation database with some amazing features and high performance. Due to these architectural advantages, RethinkDB has been trusted by top level organization such as NASA etc. We have covered the architecture of RethinkDB in detail in order to understand how it works and why it is great for real time databases. We also covered disk storage method, clustering, failover handling of RethinkDB.

Along with architecture, we have also looked over data modelling in RethinkDB. We also looked over one of the most used feature of SQL, JOIN working in NoSQL database i.e

RethinkDB. There is no doubt that RethinkDB is the next big thing.

In next chapter, we are going to study about query language of RethinkDB called **ReQL.** We will go over them with examples. We will also study about **Changefeeds** which is personally my most favourite feature of RethinkDB.

2
RethinkDB Query language

ReQL is RethinkDB query language offers powerful and easy way to perform operation on JSON documents. It is one of the important part of the RethinkDB architecture. This is built on three important principles such as Embedding ReQL in programming language, ReQL queries are chainable and ReQL queries are executed on Server.

Here is the list of topics we are going to cover along with the above mentioned principles:

- Performing conditional queries
- Traversing over nested fields
- Performing string operations
- Performing MapReduce operations
- Calling HTTP APIs using ReQL
- Handling binary objects
- Performing JOINS
- Accessing changefeed (realtimefeed) in RethinkDB
- Performing geolocation operations
- Performing administrative operations

Let's look over each one of them.

Embedding ReQL in programming language.

RethinkDB provides client drivers for various programming language. For explanation I am going to consider Node.js and the following steps are given below:

- You can start the ReQL exploration journey by connecting to database.
- Install RethinkDB client module and make sure you have RethinkDB Server

ready and running listening to default port.

- Make sure you have done **npm install rethinkdb** before running the code.

```
varrethinkdb = require('rethinkdb');
var connection = null;
rethinkdb.connect({host : 'localhost', port : 28015},function(err,conn) {
if(err) {
throw new Error('Connection error');
 } else {
connection = conn;
 }
});
```

Above simple code snippet written in Node.js is importing **rethinkdb** module and connecting to RethinkDB Server on default port. It returns callback function with error and connection variable and upon getting the **connection** object, we are good to go!

By default RethinkDB creates and connects to default database named **test.** We can perform various operations on it such as creating table or performing CRUD operation.

Let's begin with creating a simple table to store blog posts.

```
rethinkdb.db('test').tableCreate('authors')
.run(connection, function(err, result) {     if (err) throw err;
console.log(JSON.stringify(result)); });
```

Upon running the code you should be getting this in your console.

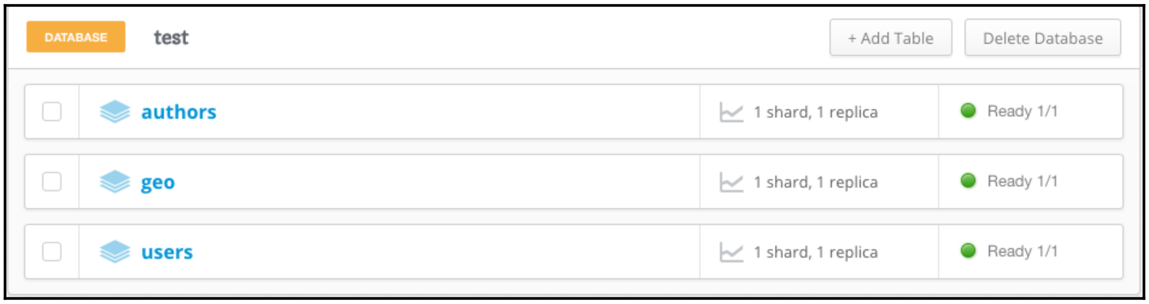
```
{
   "config_changes":[
      {
         "new_val":{
            "db":"test",
            "durability":"hard",
            "id":"a168e362-b7e7-4260-8b93-37ee43430bac",
            "indexes":[

            ],
            "name":"authors",
            "primary_key":"id",
            "shards":[
               {
                  "nonvoting_replicas":[

                  ],
                  "primary_replica":"Shahids_MacBook_Air_local_pnj",
                  "replicas":[
                     "Shahids_MacBook_Air_local_pnj"
                  ]
               }
            ],
            "write_acks":"majority"
         },
         "old_val":null
      }
   ],
   "tables_created":1
}
```

In our code, we need to check **tables_created** key in order to determine whether table was created or not.

We can check whether the table was created or not in administrative web console too. Point your browser to **http://localhost:8080** and go to database section from menu. You should be able to see something like this.



authors	1 shard, 1 replica	● Ready 1/1
geo	1 shard, 1 replica	● Ready 1/1
users	1 shard, 1 replica	● Ready 1/1

In order to run the code in Node.js, since you get the **connection** object in callback and it's asynchronous you need to manage the flow of the code. For now, you can just paste the code to create table inside the closure callback of connection and it should work fine but when writing code for production, make sure you handle the callback flow well and do not end up being in callback hell situation.

One of the best way to manage it and which I use very frequently while coding Node.js program is using **async** node module. It has rich set of functions to properly manage and design your code and of course there will be no chance of callback hell and you can by far manage not to go more than 4 level of nesting (3 is ideal but very tricky to reach).

We have our database up and running and we have created the table using our code. Next is performing very famous CRUD (Create, read, update and delete) operation on the table.

This will not just cover some important functions of RethinkDB query language but also will give quick kick to know how to perform basic operations which you have been probably doing in other NoSQL and SQL databases.

Performing CRUD operation using RethinkDB and Node

Let's begin with creating fresh new table named '**users**' and perform CRUD operation using ReQL queries.

```
varrethinkdb = require('rethinkdb');
var connection = null;
rethinkdb.connect({host : 'localhost', port : 28015},function(err,conn) {
if(err) {
throw new Error('Connection error');
 } else {
connection = conn;
 }
});
```

We will use default database to perform this operation. After getting the connection, we will create table using following code.

```
rethinkdb.db('test').tableCreate('authors').run(connection,
function(err, result) {     if (err) throw err;
console.log(JSON.stringify(result)); });
```

Make sure this piece of code is place right within the **else** section of connection code. Since we have created the table, we can perform the operation on it.

Creating new records

ReQL provides **insert()** function to create new document in the table. Here is the code to do so.

```
rethinkdb.db('test').table('users').insert({
userId : "shahid",
password : "password123",
createdDate : new Date()
  }).run(connection,function(err,response) {
if(err) {
```

```
throw new Error(err);
    }
console.log(response);
  });
```

Upon running the code, RethinkDB returns following.

```
{
deleted: 0,
errors: 0,
generated_keys: [ 'e54671a0-bdcc-44ab-99f3-90a44f0291f8' ],
inserted: 1,
replaced: 0,
skipped: 0,
unchanged: 0
}
```

Providing **db**detail is optional in the query.Key to look for in code is **inserted** which determines whether document was inserted or not. Since we are not providing primary key from our side, RethinkDB generates it automatically and you can find that in **generated_keys** key.

You can add much more complex data in the RethinkDB as you can do in any other NoSQL database. You can also store nested objects, arrays, binary objects etc. We will look over binary objects in upcoming section.

Reading the document data

You can read the documents from table using **get()** or **getAll()** ReQL method. Here is a query to retrieve all documents present in the table.

```
rethinkdb.table('users').run(connection,function(err,cursor) {
if(err) {
throw new Error(err);
    }
cursor.toArray(function(err,result) {
console.log(result);
    });
  });
```

Above code will perform the read operation in the table and returns you cursor. Cursor provides batch of data in sequence instead of whole data at once. We use **toArray()** function to read the information in batch. It should print following in the console.

```
[ {
createdDate: 2016-06-09T08:58:15.324Z,
id: 'e54671a0-bdcc-44ab-99f3-90a44f0291f8',
password: 'password123',
userId: 'shahid'
} ]
```

You can also read specific data based on filter. Let's try to read the data with the uniquely generated id using **get ()** method.

```
rethinkdb.table('users').get('e54671a0-bdcc-44ab-99f3-90a44f0291f8')
 .run(connection,function(err,data) {
if(err) {
throw new Error(err);
    }
console.log(data);
  });
```

It will return one single document from the table without the cursor. You can get the response in callback variable.

As you may notice, we are directly accessing table in the query without passing the database name because **connection** variable contains the database information.

Updating the document

You can perform the update operation on document using **update()** function which accepts the object as an argument. You can at least update one key or whole document (i.e all keys) after fetching the document. Here is the code snippet to do so.

```
rethinkdb.table('users')
    .get('e54671a0-bdcc-44ab-99f3-90a44f0291f8')
    .update({userId : 'shahidShaikh'})
    .run(connection,function(err,response) {
if(err) {
throw new Error(err);
      }
console.log(response);
    });
```

In order to update the document we are first fetching using **get()** method and then running

update() function, we are just updating the **userId** key here but you can pass other keys as well.

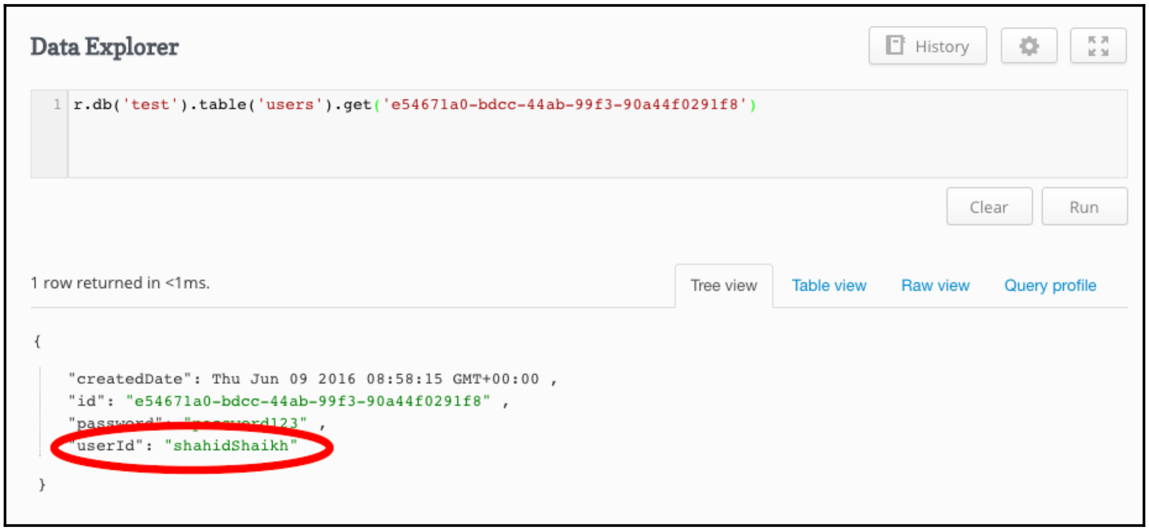
It will return following in the console.

```
{
deleted: 0,
errors: 0,
inserted: 0,
replaced: 1,
skipped: 0,
unchanged: 0
}
```

Key to look for in code is **replaced** which if returns as 1 means update operation is successfully committed else there is an error in case of any other value.

You can validate the data by either running the **get()** method again or running following code in web administrative console.

```
r.db('test').table('users').get('e54671a0-bdcc-44ab-99f3-90a44f0291f8')
```



```
r.db('test').table('users').get('e54671a0-bdcc-44ab-99f3-90a44f0291f8')
```

1 row returned in <1ms.

```
{
    "createdDate": Thu Jun 09 2016 08:58:15 GMT+00:00 ,
    "id": "e54671a0-bdcc-44ab-99f3-90a44f0291f8" ,
    "password": "password123" ,
    "userId": "shahidShaikh"
}
```

Deleting the document

We can use **delete()** ReQL commands in order to perform deletion. Again we need to fetch the record first and then perform the delete operation. You can perform delete all operation as well by first fetching all documents. Here is the code to do so.

```
rethinkdb.table('users')
    .get('e54671a0-bdcc-44ab-99f3-90a44f0291f8')
    .delete()
    .run(connection,function(err,response) {
if(err) {
throw new Error(err);
      }
console.log(response);
    });
```

This should print following in terminal.

{

deleted: 1,

errors: 0,

inserted: 0,

replaced: 0,

skipped: 0,

unchanged: 0

}

You can check **deleted** key to find out the status of delete operation. In case you want to delete all documents from database, you can do by selecting all documents first and appending **delete()** operation.

```
rethinkdb.table('users')
    .delete()
    .run(connection,function(err,response) {
if(err) {
throw new Error(err);
      }
console.log(response);
    });
```

ReQL queries are chainable

Almost all ReQL queries are chainable. You can chain ReQL queries using dot operator just like we do with pipe in Unix. Data flows from left to right and data from one commands passed to next one until query get executed. You can chain queries until your query is done.

Like we performed some queries on previous section, we chained **get()** function with **update()** or **delete()** to perform the query.

For Example.

```
rethinkdb.table('users').delete()

rethinkdb.table('users').get('<<id>>').update({id : 10});

rethinkdb.db('test').table('users').distinct().count();
```

This way of design provides natural way of reading and understanding queries. It's easy to learn, modify and read.

ReQL queries are executed on Server

Queries are formed in the client but will be sent to Server for execution when you run them. This makes sure there is no network round trip and bandwidth allocation. This provides efficiency in query execution.

We also mentioned this in chapter 1, *Query execution in RethinkDB* that RethinkDB executes query in lazy manner. It only fetches the data asked and required for the query to get complete. For.eg.

```
r.db('test').table('users').limit(5)
```

To perform this query, RethinkDB will look over for the five documents only in the **users** table. RethinkDB will perform enough operation to perform the data collection requested in query. This avoids extra computation cost and CPU cycles.

To provide the highest level of efficiency, RethinkDB automatically parallelise the query as much as possible across Server, CPU cores or even data centers. RethinkDB automatically process the complex queries into stages and parallelise them across clusters and collect data from each one of them before returning it to client.

Performing conditional queries

ReQL supports conditional queries using **subqueries, expressions** and **lambda** function. In this section we will look over each one of them using sample code written in Node.js.

In order to perform these queries, I have populated our **users** table in **test** database with some documents. Here is the query executed from RethinkDB web administrative screen.

```
r.db('test').table('users').insert([{
name : "John",
age : 24
}, {
name : "Mary",
age : 32
},{
name : "Michael",
age : 28
}])
```

Note: In web administrative screen, you do not need to provide run function with connection, it automatically append it and executes the query at Server.

Let's run a query to find out documents with age greater than 30. We are going to execute following code after getting the connection to database, same as we did in former section.

```
rethinkdb.table('users').filter(function (user) {
return user("age").gt(30)
}).run(connection,function(err,cursor) {
if(err) {
throw new Error(err);
    }
cursor.toArray(function(err,data) {
console.log(data);
    });
});
```

Upon execution, if data is same as mentioned above, you should be receiving following output.

```
[ {
age: 32,
id: 'c8e12a8c-a717-4d3a-a057-dc90caa7cfcb',
name: 'Mary'
```

```
    } ]
```

Filter() can be considered as **SELECT** clause with **WHERE** condition of SQL. It accepts condition and matches the condition with documents and returns the results in case of match. You cannot use standard comparison operators such as =, >, < etc. Instead ReQL provides functions to do same. Here are list of common comparison operators and their substitute ReQL functions.

Comparison Operators	**Substitute ReQl functions**
==	eq()
>	gt()
<	lt()
>=	ge()
<=	le()

Filter can also support complex queries. Like we mentioned, ReQL queries are chainable hence you can always perform these complex queries by dividing them into subtask.

Consider an example; you need to retrieve all those users whose age is greater than 30 and name should be '**John**'.

We know that we can use **lt()** function to perform the checking of age and **eq()** for checking exact name. In order to chain these two part of query, we will use **and()** function. Here is a code to do so.

```
rethinkdb.table("users").filter(function(user) {
return user("age").lt(30).and(user("name").eq("John"))
}).run(connection,function(err,cursor) {
if(err) {
throw new Error(err);
  }
cursor.toArray(function(err,data) {
console.log(data);
  })
});
```

Upon execution, you should receive these JSON document in console.

```
[ {
age: 24,
id: '664fced5-c7d3-4f75-8086-7d6b6171dedb',
name: 'John'
 } ]
```

You can also use range queries using **between()** or **during()** ReQL queries to solve more complex queries.

Just like the comparison operators, RethinkDB also provides functions to perform math in ReQL queries. You can use them to directly perform math on values and of course manipulate them as well. To avail the use of Math functions, you can **expr()** function as a first parameter or any number coming from latter query. Here is example to add two numbers.

```
rethinkdb.expr(2).add(2).run(connection,function(err,result) {
if(err) {
throw new Error(err);
      }
console.log(result);
    });
```

On console, you should be receiving 4

.

Similarly you can use following math functions for respective operations.

Operations	**Functions**
Subtracting two numbers	sub()
Multiplying two numbers	mul()
Divide two numbers	div()
Mod	mod()

There is of course boolean operation functions too to perform AND / OR operations. First chain in Query should be **expr()** for AND / OR boolean operation as well.

Traversing over nested fields

Consider you have JSON document like this.

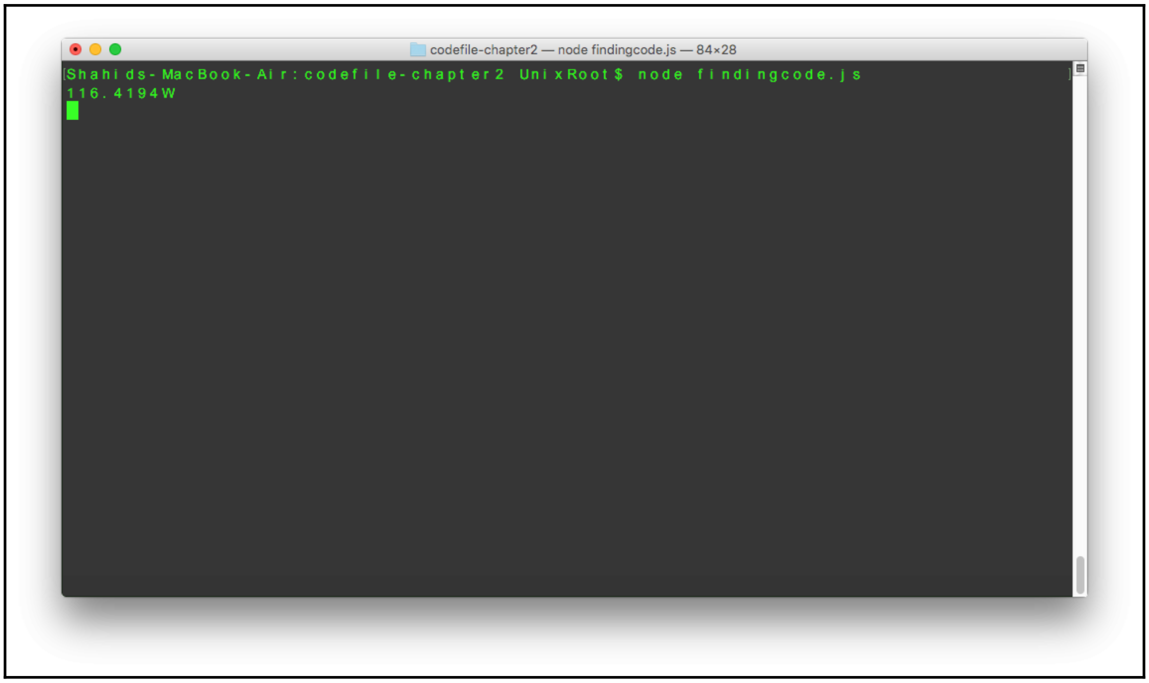
```
{
  "id":"f6f1f0ce-32dd-4bc6-885d-97fe07310845",
  "age":24,
  "name":"John",
   "address":{
      "address1":"suite 300",
      "address2":"Broadway",
      "map":{
         "latitude":"116.4194W",
         "longitude":"38.8026N"
      },
      "state":"Navada",
      "street":"51/A"
   }
}
```

And we want to fetch the value of **latitude** present on **map** key which again resides in **address** key. In order to do this, we will first get the document using primary key then traverse through keys. Here is how you can do this.

```
rethinkdb.table("users").get('f6f1f0ce-32dd-4bc6-885d-97fe07310845')("addre
ss")("map")("latitude").run(connection,function(err,result) {
if(err) {
throw new Error(err);
  }
console.log(result);
});
```

You should be able to see value of **latitude** key in console.

Insert



```
Shahids-MacBook-Air:codefile-chapter2 UnixRoot$ node findingcode.js
116.4194W
```

Performing string operations

ReQL provides following functions to manipulate and search over strings.

- Match() takes regular expression as an input and perform search over the field and in case it matches, it returns back the data in **cursor** which we can loop over to retrieve actual data.

- For example, Find all users whose name starts with **J.** Here is query for same.

```
rethinkdb.table("users").filter(function(user) {
return user("name").match("^J")
}).run(connection,function(err,cursor) {
if(err) {
throw new Error(err);
  }
cursor.toArray(function(err,data) {
console.log(data);
  });
});
```

- Here first we are performing filter and inside it putting our **match()** condition. Filter gives every document to **match()** function and it appends it to cursor. Upon running, you should be able to view the users with name starting with **J.**
    - **split()** takes optional argument as separator to filter out the string and returns back an array contains the splitted part of string. If no argument is present, it separates by space.
        - For example, you have address stored like this, **"Suite 300"** and you want to split them by space, you can do so like this (Assuming the structure of document as we mentioned above).

```
rethinkdb.table("users").get('f6f1f0ce-32dd-4bc6-885d-97fe07310845')("addre
ss")("address1").split().run(connection,function(err,result) {
if(err) {
throw new Error(err);
  }
console.log(result);
});
```

You should be receiving the following output.

```
[ 'suite', '300' ]
```

- Make sure your first input is string and not an object. You can also provide any separator say **|,&,*** etc. You can also provide an optional argument which limits the number of splits.
    - Rest two functions will allow you to change the casing of string. For example, we want name to appear in capital casing only, you can run following query to do so.

```
rethinkdb.table("users").get('f6f1f0ce-32dd-4bc6-885d-97fe07310845')("name"
).upcase().run(connection,function(err,result) {
if(err) {
throw new Error(err);
  }
console.log(result);
});
```

You should be receiving the name in uppercase in console. You can use **downcase()** in same way.

Performing MapReduce operations

Mapreduce is the programming model to perform operations (Mainly aggregation) on distributed set of data across various clusters in different servers. This concept was coined by Google and been used in Google file system initially and later been adopted in open source Hadoop project.

Mapreduce works by processing the data on each server and then combine it together to form a resultset. It actually divides into two operations namely **Map** and **reduce.**

- **Map**: It performs the transformation of the elements in the group or individual sequence
- **Reduce**: It performs the aggregation and combine results from Map into meaningful result set.

In RethinkDB, MapReduce queries operate in three steps.

1. **Group operation**:To process the data into groups. This step is optional.
2. **Map operation**:To transform the data or group of data into sequence.
3. **Reduce operation**: To aggregate the sequence data to form resultset.

So mainly it is GMR (Group – Map – Reduce) operation. RethinkDB spread the mapreduce query across various clusters in order to improve efficiency. There is specific command to perform this GMRoperation; however RethinkDB already integrated them internally to some aggregate functions in order to simplify the process.

Let's perform some aggregation operation in RethinkDB.

Grouping the data

To group the data on basis of field we can use **group()** ReQL function. Here is sample query on our users table to group the data on the basis of name.

```
rethinkdb.table("users").group("name").run(connection,function(err,cursor)
{
if(err) {
throw new Error(err);
  }
cursor.toArray(function(err,data) {
console.log(JSON.stringify(data));
  });
});
```

Here is the output for same.

```
[
   {
      "group":"John",
      "reduction":[
         {
            "age":24,
            "id":"664fced5-c7d3-4f75-8086-7d6b6171dedb",
            "name":"John"
         },
         {
            "address":{
               "address1":"suite 300",
               "address2":"Broadway",
               "map":{
                  "latitude":"116.4194W",
                  "longitude":"38.8026N"
               },
               "state":"Navada",
               "street":"51/A"
            },
            "age":24,
            "id":"f6f1f0ce-32dd-4bc6-885d-97fe07310845",
            "name":"John"
         }
      ]
   },
   {
      "group":"Mary",
      "reduction":[
         {
            "age":32,
            "id":"c8e12a8c-a717-4d3a-a057-dc90caa7cfcb",
            "name":"Mary"
         }
      ]
   },
   {
      "group":"Michael",
      "reduction":[
         {
            "age":28,
            "id":"4228f95d-8ee4-4cbd-a4a7-a503648d2170",
            "name":"Michael"
         }
      ]
   }
]
```

If you observe the query response, data is group by the name and each group is associated with document. Every matching data for the group resides under **reduction** array. In order to work on each **reduction** array, you can use **ungroup()** ReQL function which in turns takes grouped streams of data and convert it into array of object. It's useful to perform the operations such as sorting and so on.on grouped values.

Counting the data

We can count the number of documents present in the table or a sub document of a document using **count()** method. Here is simple example.

```
rethinkdb.table("users").count().run(connection,function(err,data) {
if(err) {
throw new Error(err);
  }
console.log(data);
});
```

It should return the number of documents present in the table. You can also use it count the sub document by nesting the fields and running **count()** function at the end.

Sum

We can perform the addition of the sequence of data. If value is passed as an expression then sums it up else searches in the field provided in the query.

For example, find out total number of ages of users.

```
rethinkdb.table("users")("age").sum().run(connection,function(err,data) {
if(err) {
throw new Error(err);
  }
console.log(data);
});
```

You can of course use an expression to perform math operation like this.

```
rethinkdb.expr([1,3,4,8]).sum().run(connection,function(err,data) {
if(err) {
```

```
throw new Error(err);
  }
console.log(data);
});
```

Should return 16.

Avg

Performs the average of the given number or searches for the value provided as field in query. For example,

```
rethinkdb.expr([1,3,4,8]).avg().run(connection,function(err,data) {
if(err) {
throw new Error(err);
  }
console.log(data);
});
```

Min and Max

Finds out the maximum and minimum number provided as an expression or as field.

For example,

Find out the oldest users in database.

```
rethinkdb.table("users")("age").max().run(connection,function(err,data) {
if(err) {
throw new Error(err);
  }
console.log(data);
});
```

Same way of finding out the youngest user.

```
rethinkdb.table("users")("age").min().run(connection,function(err,data) {
if(err) {
throw new Error(err);
```

```
  }
console.log(data);
});
```

Distinct

Distinct finds and removes the duplicate element from the sequence, just like SQL one.

For example, find user with unique name.

```
rethinkdb.table("users")("name").distinct().run(connection,function(err,dat
a) {
if(err) {
throw new Error(err);
  }
console.log(data);
});
```

It should return an array containing the names.

```
[ 'John', 'Mary', 'Michael' ]
```

Contains

Contains look for the value in the field and if found return boolean response, true in case if it contains the value, false otherwise.

For example, find the user whose name contains **John.**

```
rethinkdb.table("users")("name").contains("John").run(connection,function(e
rr,data) {
if(err) {
throw new Error(err);
  }
console.log(data);
});
```

Should return **true**.

Map and reduce

Aggregate functions such as count (), sum () already makes use of map and reduce internally, if required then group () too. You can of course use them explicitly in order to perform various functions.

Calling HTTP APIs using ReQL

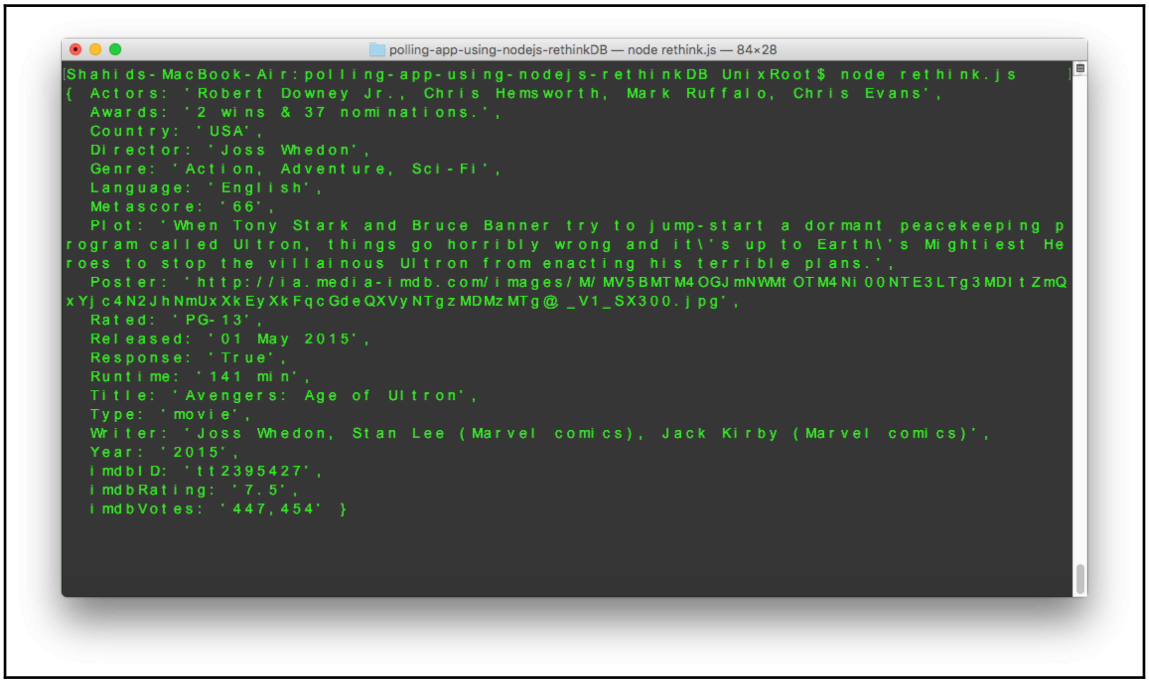
RethinkDB provides support to call external API which returns data in JSON object, which probably most of the large API provider do. You can call HTTP api directly from your database hence no need of writing piece of code to just call an API and then dump into database. RethinkDB also handles it asynchronously so performance won't be affected if API takes longer time.

Let's try one basic API call before moving ahead with storing those in our table. We all know and use **OMDB** for movies review. There is a website called **omdbapi.com** which provides the API's to find out the movie information present in OMDB database. Let's call one to fetch information about **Avengers** movie and see how it goes.

```
rethinkdb.http("http://www.omdbapi.com/?t=avengers&y=2015&plot=short&r=json
").run(connection,function(err,data) {
if(err) {
throw new Error(err);
  }
console.log(data);
});
```

You should be receiving following in the console.

```
{
  Actors: 'Robert Downey Jr., Chris Hemsworth, Mark Ruffalo, Chris Evans',
  Awards: '2 wins & 37 nominations.',
  Country: 'USA',
  Director: 'Joss Whedon',
  Genre: 'Action, Adventure, Sci-Fi',
  Language: 'English',
Metascore: '66',
  Plot: 'When Tony Stark and Bruce Banner try to jump-start a dormant
peacekeeping program called Ultron, things go horribly wrong and it's up to
Earth's Mightiest Heroes to stop the villainous Ultron from enacting his
terrible plans.',
  Poster:
'http://ia.media-imdb.com/images/M/MV5BMTM4OGJmNWMtOTM4Ni00NTE3LTg3MDItZmQx
Yjc4N2JhNmUxXkEyXkFqcGdeQXVyNTgzMDMzMTg@._V1_SX300.jpg',
  Rated: 'PG-13',
  Released: '01 May 2015',
  Response: 'True',
  Runtime: '141 min',
  Title: 'Avengers: Age of Ultron',
  Type: 'movie',
  Writer: 'Joss Whedon, Stan Lee (Marvel comics), Jack Kirby (Marvel
comics)',
  Year: '2015',
imdbID: 'tt2395427',
imdbRating: '7.5',
imdbVotes: '447,454'
}
```



polling-app-using-nodejs-rethinkDB — node rethink.js — 84×28

```
Shahids-MacBook-Air:polling-app-using-nodejs-rethinkDB UnixRoot$ node rethink.js
{ Actors: 'Robert Downey Jr., Chris Hemsworth, Mark Ruffalo, Chris Evans',
  Awards: '2 wins & 37 nominations.',
  Country: 'USA',
  Director: 'Joss Whedon',
  Genre: 'Action, Adventure, Sci-Fi',
  Language: 'English',
  Metascore: '66',
  Plot: 'When Tony Stark and Bruce Banner try to jump-start a dormant peacekeeping p
rogram called Ultron, things go horribly wrong and it\'s up to Earth\'s Mightiest He
roes to stop the villainous Ultron from enacting his terrible plans.',
  Poster: 'http://ia.media-imdb.com/images/M/MV5BMTM4OGJmNWMtOTM4Ni00NTE3LTg3MDItZmQ
xYjc4N2JhNmUxXkEyXkFqcGdeQXVyNTgzMDMzMTg@._V1_SX300.jpg',
  Rated: 'PG-13',
  Released: '01 May 2015',
  Response: 'True',
  Runtime: '141 min',
  Title: 'Avengers: Age of Ultron',
  Type: 'movie',
  Writer: 'Joss Whedon, Stan Lee (Marvel comics), Jack Kirby (Marvel comics)',
  Year: '2015',
  imdbID: 'tt2395427',
  imdbRating: '7.5',
  imdbVotes: '447,454' }
```

Well, it does work. We can now call this API to directly dump them into our database, or may be dump few fields. Here is sample query to add new document into table using HTTP API call. I have created another table named **movies** to add the information.

```
rethinkdb.table("movies").insert(
rethinkdb.http("http://www.omdbapi.com/?t=avengers&y=2015&plot=short&r=json
")
).run(connection,function(err,data) {
if(err) {
throw new Error(err);
  }
console.log(data);
});
```

It will add new document in our table with the information coming from HTTP response.. You can also provide dynamic URL's coming from the table to call the HTTP API.

By default RethinkDB calls the GET http method, however you can specify your own and if needed provides the data too. Here is a sample example.

```
r.http('<< URL >>', { method: POST, data: {userId : 10, name : "Shahid" }
});
```

In case if you receive response in paginated way, RethinkDB also provides a way to solve that. You need to either provide **page** and **pageLimit** parameter depending upon the API you are calling or getting a response. RethinkDB will call the API and provide the result in streams which we can access using cursor API.

There are some API's which requires authentication first, you can do that too using RethinkDB http call by passing **auth** key.

```
r.http('<< URL >>', { auth: {           user: userName,           pass:
password        } });
```

Handling binary objects

As we have mentioned in chapter about RethinkDB binary object support, let's look over how to use it using ReQL. Syntax to store binary objects differs from client to client, in Node.js it uses buffers to convert the stream into binary and we can use RethinkDB to insert that in table.

Let's take example from the above document. There is a key called **Poster** which is official poster of the movie in JPEG image format. We can store the image directly in RethinkDB in binary format.

Consider following code.

```
rethinkdb.http("http://www.omdbapi.com/?t=avengers&y=2015&plot=short&r=json
").run(connection,function(err,data) {
if(err) {
throw new Error(err);
  }
rethinkdb.table("movies").insert({
movieName :data.Title,
posterImage :rethinkdb.http(data.Poster, {resultFormat : 'binary'})
  }).run(connection,function(err,data) {
if(err) {
throw new Error(err);
    }
console.log(data);
  });
});
```

What we are doing here is first fetching the record from OMDB and then putting the **Poster** in our table using **Insert()** command. If you see the result, it looks like this.

```
{
"id": "1890fc6f-1f95-4cbc-8b0e-c62a97884e0c" ,
"movieName": "Avengers: Age of Ultron" ,
"posterImage": <binary, 46.5KB, "ffd8ffe0 00 10...">
}
```

You can store files directly from the file system as well. Use file system API to read the file and then provide the path to **insert()** command. Simple as that, you can also store files in RAW format by passing **raw** key in **resultFormat** object.

Performing JOINS

JOINS are one of the features for NoSQL database. RethinkDB provides the ReQL functions to perform the various types of JOINS such as inner, outer etc. Please refer chapter one to study more in detail.

Accessing changefeed (realtimefeed) in RethinkDB

We had mentioned it in chapter one about the real time feature of RethinkDB called **changefeed.** It is the heart of the RethinkDB real time functionality. RethinkDB changefeed provides the continuous live update about the changes happening in the subscribed table.

In order to avail the feature of changefeed, you just need to attach your listener for particular table and you should be receiving every single minor update happening in the table such as addition, deletion, updation etc.

Let's demonstrate this using our **users** table. Here is the piece of code which will add the listener to the table.

```
rethinkdb.table("users").changes().run(connection,function(err,cursor) {
if(err) {
throw new Error(err);
  }
cursor.each(console.log);
});
```

That's it. Run the code on separate terminal in order to observe the behaviour. Let's try to add some new document into the table and see how it works. I have added new document using web administrative screen of RethinkDB and this is what it prints on console.

```
{
new_val: {
id: '25cf3a63-f750-469a-847f-f43d827289a1', name: 'Shahid'
  },
old_val: null
}
```

Since we have performed addition, we get the **old_val**key as null, however in case of updating and deletion it should return old value of the document. Let's run one update query to see the behaviour of changefeed.

This is my query from web console.

```
r.table("users").get("4228f95d-8ee4-4cbd-a4a7-a503648d2170").update({age :
30});
```

This is response in the terminal.

```
 {
new_val:
   {
age: 30,
id: '4228f95d-8ee4-4cbd-a4a7-a503648d2170',
name: 'Michael'
   },
old_val:
   {
age: 28,
id: '4228f95d-8ee4-4cbd-a4a7-a503648d2170',
name: 'Michael'
    }
 }
```

This is really awesome feature. You don't just get the changes of the table; you get old and new value in order to perform extra operation. Let's try one with deletion and see how changefeed reacts.

```
r.table("users").get("25cf3a63-f750-469a-847f-f43d827289a1").delete()
```

Changefeed prints the following on the console.

```
{
new_val: null,
old_val: {
id: '25cf3a63-f750-469a-847f-f43d827289a1', name: 'Shahid'
    }
 }
```

It returns new value as null because of deletion, fair enough.

You can also perform the changefeed operation over particular or set of document as well. All you need to do is to select those document using get() method and append changes() after it. Example shown below.

```
rethinkdb.table("users").get("4228f95d-8ee4-4cbd-a4a7-a503648d2170")
.changes()
.run(connection,function(err,cursor) {
if(err) {
throw new Error(err);
  }
cursor.each(console.log);
});
```

This will return the changes happening in the document. In order to get the multiple document to find changes, you can use **filter()** with it. Like we said, ReQL queries are chainable.

When we attach our listener to the table, it only provides us the changes when there is any ReQL operation happening on the table, how about getting the initial data in order to show it to client end etc. You can do this too by specifying **includeInitial** key to **true** in changes() function.

There might be some requirement comes where you want the changes in particular way such as without old value etc. You can do this by specifying **includeStates** key to **true** in changes() function. It returns **type** key to explain the type of change in changefeed.

You can specify following in type key.

- add – new value is added.
- remove – value is removed.
- change – update of value.
- Initial – an initial value notification.
- uninitial – not an initial value.
- State – a status information.

If there is fast changes happening in the table, it is possible the one of the changes in the table happens before the invocation of changes() function. In such cases RethinkDB will return you one object containing multiple changes.

However, if you want to receive individual changes for this special case, you can do this by specifying **squash** key to **false** in changes() function. It will buffer all the changes and returns you back, it can buffer max 100,000 changes document.

Application of changefeed

Since this is one of the promising features of RethinkDB you might be thinking where can we apply it in real time. Application of changefeed can range from small scale notification system to large scale news feed of the application. One thing you should keep in mind regarding changefeed is that it works directly with your database, you are not doing any external or WAN http call in order to know what's going on in the table, this saves lots of bandwidth and improves latency and round trip time.

You can design real time social media application using this or a push notification and so on. Application of this feature is limitless and can be applied across various domains.

One limitation of changefeed is that it push changes to middle layer client only. Hence you need to write some code to push this data to client layer. I would suggest socket will perform very best in this scenario. However, there is another project currently in beta development and probably will be available for public after launch of this book called **Horizon.** This project will let you access RethinkDB database from client end. Don't worry about the security, there will be middle layer but that will be pre written and tightly coupled with UI in order to contact database safely. We will cover this in detail in chapter 10.

Performing geolocation operations

RethinkDB does support geolocation operations and you can write rich geographical by combining geospatial queries and changefeed. Here we are going to look over some of the ReQL functions which we can use to perform geolocation operation.

Storing a coordinate

You can store a coordinate using point() ReQL api. It accepts longitude and latitude as an input parameter. Here is the sample example of adding location of Mumbai and Delhi in our table. I am using web administrative console to execute the query.

```
r.table("geo").insert([
  {
place : 'Mumbai',
location :r.point(19.0760,72.8777)
  },
  {
place : 'Delhi',
location :r.point(28.6139,77.2090)
  }])
```

Finding distance between points

You can find out the distance between two points using **distance()** ReQL command. Here is an example.

```
r.table('geo').get("25855d3d-70f6-4480-8472-4c7081b1874a")('location').dist
ance(r.table('geo').get("02e1f1ce-0768-460c-8ddd-9e0cf42ce887")('location')
)
```

Here we are selecting two points and passing the location to the distance function. This should return you the distance in points. You can also specify the units as kilometer, meter etc by passing the unit key in distance function.

Similarly, there are ReQL functions available for finding out the nearest point, finding intersection of a point in circle, creating circle using points and lines etc.

Performing administrative operations

RethinkDB provides administrative functions which you can use to perform various administrative tasks such as granting permission to the table or database, changing size of cluster or shard, re-configuring failover shards etc. You can perform administrative operations either by using web console or by using ReQL commands. In next chapter we will look over these in detail.

Summary

We have covered the basics and advance level of queries in ReQL with client driver (Node.js) code. We covered performing CRUD operation using RethinkDB and also covered handling various special cases such as calling HTTP api's from the query and storing binary blobs in RethinkDB. We also look over one of the promising feature of RethinkDB called **changefeed** in detail.

In next chapter we are going to look over some administrative operations such as handling permission along with configuring clusters and shard in RethinkDB. We will also look over failover and other administrative tool which you can use to manage RethinkDB in detail.