Simply RethinkDB

A Database with a query language for human

Vinh Quốc Nguyễn

Simply RethinkDB

A database with a query language for human

Vinh Quốc Nguyễn

This book is for sale at http://leanpub.com/simplyrethinkdb

This version was published on 2015-10-12

Tweet This Book!

Please help Vinh Quốc Nguyễn by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Simply RethinkDB: Starting with RethinkDB in a week

The suggested hashtag for this book is #simplyrethinkdb.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#simplyrethinkdb

This books was written as an effort to learn more about RethinkDB. At some points, I almost abandon it because it is lots of work.

A long the way, I started to write Relang, a RethinkDB driver for Erlang which gives me more deep understanding of RethinkDB and make me want to finish this book.

All of this will not be published without my wife supporting. An awesome girl who without her support I won't be able to finish this.

I also thank so much for your purchase. If you happen to live in San Jose, please visit and see us in person. I want to say thank to you in person.

This books mark an important period in my life when I married. The "Rethink" word also reminds me to rethink when we have trouble in our relationship in the future. Rethinking and understanding.

Contents

1. Welcome

Introduction

Welcome to my readers. I appreciate your purchase. This will help me continue improving the book content.

Before we go into the technical details, I have something to say.

Firstly, I'm not a RethinkDB expert at all. I'm just an average guy who loves programming and new technologies. To me, RethinkDB is a pleasure to use. However, due to its age, there are not many books and documents about it comparing to other database systems. While the RethinkDB cocumentation and API is very good, it can be hard to know where to start. So this guide is for all those in mind who are unsure about taking the plunge into RethinkDB as something totally new. I hope this helps them ease into the learning process.

The purpose of this book is to organize the concepts of RethinkDB in order to help you to read and understand the RethinkDB API directly. Upon finishing the book, you will have a foundational knowledge in which to extend your knowledge with many other RethinkDB videos and blog posts out on the Internet.

Secondly, I'm a fan of Mixu's[1] writing style[2]. I won't cover deeply things like installing RethinkDB, fine-tuning, extra function parameters, and so on. Those topics are covered very well on RethinkDB's documention itself. What I want you to take away from this book is a good grasp on RethinkDB usage in practice. and how to apply commands in real scenarios.

Third, I'm not fluent in English. If you find any mistakes, you can report the issue on repository[3] or email me directly.

Fourth, RethinkDB is changing so fast that things in this book may not reflect its current state. Once again, I'd be very grateful for any errata you may point out, via my email or Github. Since this is a LeanPub book, once I update you may download it again free of charge.

And finally, due to my limited knowledge with RethinkDB, I want to keep this book short and straight to the point. Expect a book of around 200 pages. My goal is for this to be a book that you can pick up, read on the train while riding to work and after a week you can sit down and actually start your first RethinkDB project without hesitation.

---

[1] http://blog.mixu.net/

[2] http://blog.mixu.net/2012/07/26/writing-about-technical-topics-like-its-2012/

[3] https://github.com/kureikain/simplyrethink

Why learn RethinkDB?

RethinkDB is mind-blowing to me. I like the beauty and nature of ReQL which is built into the language. It is also very developer friendly with its own administrator UI. RethinkDB is very easy to learn, because its query language is natural to how we think when constructing a query. We can easily tell what ReQL will do and what is the execution order of the query.

Take this SQL query:

```
SELECT * FROM users WHERE name="Vinh" ORDER BY id DESC LIMIT 10,100
```

This query is passed as a string and occaisionally you may sometimes forget the ordering or syntax. Will we put **ORDER** before or after **LIMIT**? Where the `WHERE` clause should appear? We also can't be certain if an index will be used. Because SQL is a string, the order of execution is defined by the syntax. Memorizing that syntax is essential.

Compare this with ReQL (RethinkDB Query Language):

```
r.table('users').getAll('vinh', {index: 'name'}).order_by(r.desc(id)).limit(10)
```

We can easily ascertain (or 'grok[4]') immediately what will result from this query, and the order of execution is clear to us. This is because the methods are chained, one after another, from left too right. ReQL was designed with the intention of a very clear API but without the ambiguity that comes with an ORM.

We can also see that it will use an index **name** when finding data. The way the query is constructed, feels similiar to `jQuery` if you are a front-end developer who never works with databases. Or if you are a functional programming person, you probably see the similarity immediately.

If the above example hasn't convinced you, then check this out:

```
SELECT *
FROM foods as f
INNER JOIN compounds_foods as c ON c.food_id=f.id
WHERE f.id IN (10, 20)
ORDER By f.id DESC, c.id ASC
```

The same query represented as ReQL would look like this:

[4] https://en.wikipedia.org/wiki/Grok

```
r.db("food")
  .table("foodbase")
  .filter(function (food) {
    return r.expr([10, 20]).contains(food("id"))
  })
  .eqJoin("id", r.db("foodbase").table("compound_foods"), {index: "food_id"})
```

Even if you are not completely familar with the syntax, you can guess what is going to happen. In ReQL, we are taking the `foodbase` database, and table `foods`, and filtering them and filtering the result with another table called `compound_foods`. Within the filter, we pass an anonymous function which determines if the "id" field of document is contained in the array `[10, 20]`. If it is either `10` or `20` then we join the results with the `compound_foods` table based on the `id` field and use an index to efficiently search. The query looks like a chain of API call and the order of execution is clear to the reader.

RethinkDB really makes me rethink how we work with database. I don't have to write a query in a language that I don't like. As well, I'm no longer forced to use a syntax that I don't like because I have no choice. And further, if something does go wrong, I don't have to slowly tear apart the entire string to find out which clause has the issue. The resulting error from a ReQL query allows me to more precisely determine the cause of error.

Furthermore, RethinkDB is explicit. Later on, you will also learn that in RethinkDB you have to explicitly tell it to do some *not-very-safe* operations. Such as when a non-atomic update is required, you clearly set a flag to do it. RethinkDB by default has sensible and conservative settings as a database should to help you avoid shooting yourself in the foot.

In my opinion, RethinkDB forces us to understand what we are doing. Everything is exposed on the query. No magic, no "why did this query fail on production but work as expected on my local machine", no hidden surprises.

In Vietnamese culture, we usually follow a rule of three in demonstrations before we conclude. Being Vietnamese, let me end by showing you this third example.

Do you understand the query below?

```
r
  .db('foodbase')
  .table('foods')
  .filter(r.row('created_at').year().eq(2011))
```

This query finds all foods which were inserted in the year **2011**. I cannot even provide an equivalent SQL example, because it just cannot be as beautiful and concise as the above query.

Feedback

I appreciate all of your feedbacks to improve this book. Below is my handle on internet:

- twitter: http://twitter.com/kureikain[5]
- email: kurei@axcoto.com[6]
- twitter book hashtag: #simplyrethinkdb

[5] http://twitter.com/kureikain
[6] kurei@axcoto.com

Credit

- Sample dataset: foodb.ca/foods[7]
- Book cover: Design by my friend, aresta.co[8] helps to create the cover for this book

[7] http://foodb.ca/foods
[8] http://aresta.co/

2. Getting to know RethinkDB

Let's warm up with some RethinkDB concepts, ideas and tools. In this chapter, thing may a bit confuse because sometime to understand concept A, you need to understand B. To understand B, you need C, which is based on A. So plese use your feeling and don't hestitate to do some quick lookup on offical docs to clear thing out a bit.

From now on, we will use term **ReQL** to mean anything related to RethinkDB query language, or query API.

Getting Started

It's not uncommon to see someone write an interactive shell in browser for evaluation purpose such as mongly, tryRedis. This isn't applied for RethinkDB because it comes with an excellent editor where you can type code and run it.

Install RethinkDB by downloading package for your platform http://rethinkdb.com/docs/install/. Run it after installing.

The Ports

By default, RethinkDB runs on 3 ports

8080 this is the web user interface of RethinkDB or the dashboard. You can query the data, check performance and server status on that UI.

28015
this is the client drive port. All client drive will connect to RethinkDB through this port. If you remember in previous chapter, we used a `tcpdump` command to listen on this port to capture data send over it.

29015
this is intracluster port; different RethinkDB node in a cluster communicates with eath others via this port

The dashboard

Open your browser at http://127.0.0.1:8080 and welcome RethinkDB. You can play around to see what you have:

Navigate to the Explorer tab, you can type the command from there. Let's start with

```
1 r.dbList()
```

Run it and you can see a list of database.

RethinkDB object

Similar to traditonal database system, we also have database in RethinkDB. A database contains many tables. Each table contains your JSON document. Those JSON document can contains any fields. A table doesn't force a schema for those fields.

A JSON document is similar to a row in MySQL. Each of field in the document is similar to column in MySQL. When I say *JSON document*, I mean an JSON object with fields, not a single number, an array or a string. However, each field can content whatever JSON data type.

More than that, the same field can accept whatever data type. On same table, two document can contains diferent data type for same field.

Durability

You will see an option/argument call `durability` `*durability` appear a lot in many option of ReQL. Because it's so common and it's very important, I want to address it here. Durability accepts value of *'soft'* or *'hard'*.

soft means the writes will be acknowledge by server immdediately and data will be flushed to disk in background.

hard The opposite of soft. The default behaviour is to acknowledge after data is written to disk. Therefore, when you don't need the data to be consitent, such as writing a cache, or an important log, you should set durability to soft in order to increase speed

Atomicity

According to RethinkDB docs [^atomic], write atomicity is supported on a per-document basis. So when you write to a single document, it's either succesfully or nothing orrcur instead of updating a couple field and leaving your data in a bad shape. Furthermore, RethinkDB guarantees that any combination of operation can be executed in a single document will be write atomically.

However, it does comes with a limit. To quote RethinkDB doc, Operations that cannot be proven deterministic cannot update the document in an atomic way. That being said, the unpredictable value won't be atomic. Eg, randome value, operation run by using JavaScript expression other than ReQL, or values which are fetched from somewhere else. RethinkDB will throw an error instead of silently do it or ignore. You can choose to set a flag for writing data in non-atomic way.

Multiple document writing isn't atomic.

[^atomic] http://www.rethinkdb.com/docs/architecture/#how-does-the-atomicity-model-work

Command line tool

Besides the dashboard, RethinkDB gives us some command line utility to interactive with it. Some of them are:

- import
- export
- dump
- restore

import

In the spirit of giving users the dashboard, RethinkDB also gives us some sample data. You can download the data in file input_polls and country_stats at https://github.com/rethinkdb/rethinkdb/tree/next/demos/electi and import them into test database

```
rethinkdb import -c localhost:28015 --table test.input_polls --pkey uuid -f inpu\
t_polls.json --format json
rethinkdb import -c localhost:28015 --table test.county_stats --pkey uuid -f cou\
nty_stats.json --format json
```

Notice the `--table` argument, we are passing the table name in format of *database_name*.*table_name". In our case, we import the data into two tables: `input_polls` and `county_stats` inside database `test`.

Basically you can easily import any file contains a valid JSON document.

export

`export` exports your database into many JSON files, each file is a table. The JSON file can be import using above import command.

dump

dump will just export whole of data of a cluster, it's similar to an `export` command, then follow by *gzip* to compress all JSON output file. Syntax is as easy as.

```
rethinkdb dump -c 127.0.0.1:28015
```

Here is an example output when I run this command:

```
rethinkdb dump -c 127.0.0.1:28015
NOTE: 'rethinkdb-dump' saves data and secondary indexes, but does *not* save
 cluster metadata.  You will need to recreate your cluster setup yourself after
 you run 'rethinkdb-restore'.
Exporting to directory...
[========================================] 100%
764509 rows exported from 9 tables, with 21 secondary indexes
  Done (157 seconds)
Zipping export directory...
  Done (5 seconds)
```

The dump result is a gzip file whose name is in format `rethinkdb_dump_{timestamp}.tar.gz` It's very useful when you want to try out something and get back your original data. Note it here because you will need it later.

restore

Once we got the dump file with `dump` command. We can restore with:

```
rethinkdb restore -c 127.0.0.1:28015 rethinkdb_dump_DATE_TIME.tar.gz
```

Import sample data

It's much nicer to work with real and fun data than boring data. I found a very useful dataset call FooDB[9]. It's a data about food constituents, chemistry and biolog. To quote their about page:

> What is FooDB FooDB is the worldâ€™s largest and most comprehensive resource on food constituents, chemistry and biology. It provides information on both macronutrients and micronutrients, including many of the constituents that give foods their flavor, color, taste, texture and aroma

I import their data into RethinkDB, and generate some sample tables such as users table. At the end, I used the `dump` command to generate sample data which you can download using below links[10]

https://www.dropbox.com/s/dy48el02j9p4b2g/simplyrethink_dump_2015-08-11T22%3A15%3A51.tar.gz?dl=0[11]. Once you download it, you can import this sample dataset:

```
rethinkdb restore -c 127.0.0.1:28015 simplyrethink_dump_2015-08-11T22:15:51.tar.\
gz
```

The output looks like this:

```
Unzipping archive file...
  Done (1 seconds)
Importing from directory...
[                                        ]   0%
[                                        ]   0%
[                                        ]   0%
[                                        ]   0%
[========================================] 100%
764509 rows imported in 9 tables
  Done (2166 seconds)
```

Once this processing is done, you should have a database call **foodb** which contains the data we play throught the book. At any point, if you messed up data, you can always restore from this sample data. Also, I encourage to back up data if you build many interesting data to experiment yourself.

[9] http://foodb.ca/about

[10] https://www.dropbox.com/s/dy48el02j9p4b2g/simplyrethink_dump_2015-08-11T22%3A15%3A51.tar.gz?dl=0

[11] https://www.dropbox.com/s/dy48el02j9p4b2g/simplyrethink_dump_2015-08-11T22%3A15%3A51.tar.gz?dl=0

3. Reading Data Basic

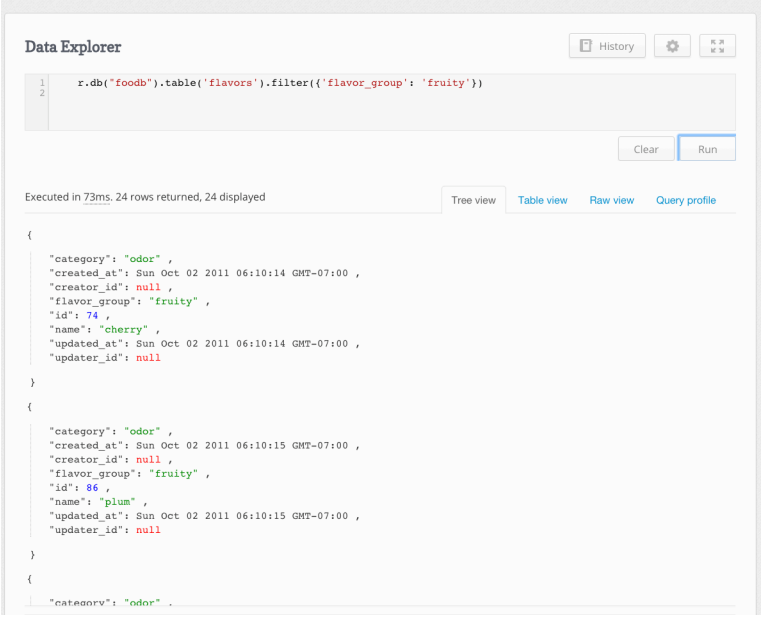
If you are lazy(just like me) and skip straigh to this chapter, please go back to the end of previous chapter to import sample dataset. Once you did it, let's start. Oh, before we start, let me tell you this, sometime if you see an . . . it means, we have more data returning, but I cannot paste them all into the book. I used . . . to denote for more data available.

Getting to Know ReQL

RethinkDB uses a special syntax call *ReQL* to interact with the data. ReQL is chainable. You start with a database, chain to table, and chain to other API to get what you want, in a way very natural. Type this into data explorer:

```
1  r.db("foodb").table('flavors').filter({'flavor_group': 'fruity'})
```

You should see some interesting data now.



Result of filter command by flavor_group

Don't worry about the syntax, just look at it again and even without any knowledge you know what it does and easily remember it. A way, for me, to understand ReQL is that every command return an object which share some API which we can call those API as if method of an object.

ReQL is particular binding to your language. Though, of course, they will look familiar between different language to maintain consitent look and feel. But, they are different. Those querie are constructed by making function call of your language, not by concat SQL String, or not by special JSON object like MongoDB. Therefore, it feel very natualy to write ReQL, as if the data we manipulate is an object or data type in our language. But everything comes with a trade off. On the downside, we have to accept differences of ReQL betweens many language. No matter how hard we try, different language has different syntax, especially when it comes to anonymous function.

What is `r`? `r` is like a special namespace which is all RethinkDB is exposed via it. It's just a normal variable in your language, or a namespace, a package name, a module. Think of `r` like `$` of jQuery. If you don't like `r`, assign another variable to it is possible.

We will call all method of `r`, or of any method of return resulted from other method are command for now. Think of it like a method in jQuery world.

Here is example, with this HTML structure:

```
<div class="db">
  <div class="table" data-type="anime">Haru River</div>
  <div class="table" data-type="anime">Bakasara</div>
  <div class="table" data-type="movie">James Bond</div>
</div>
```

To filter only anime movie, we can use this jQuery:

```
$('.db').find('.table').filter('[data-type="anime"]')
```

If we have a database call `db`, and a table call `table`, with 3 records:

```
{type: 'anime', title: 'Haru River'}
{type: 'anime', title: 'Bakasara'}
{type: 'movie', title: 'James Bond'}
```

The equavilent ReQL use to find only anime is:

```
r.db('db').table('table').filter({type: 'anime'})
```

Notice how similar the structure between them? Because of those concept, I find ReQL is easy to learn. If you can write jQuery, you can write ReQL.

Another way to understand is considering ReQL like the pipe on Linux, you select the data, passing into another command:

```
$ cd db; ls -la table/* | grep 'type: anime'
```

Drivers

This section deep down a bit on how the drivers work, you can skip if you are not interested in how RethinkDB driver works at low level. But I really hope you keep reading it. Let's start.

ReQL is binded to your language. Therefore, the API is implemented totally by the driver itself. You won't work directly with RethinkDB server. You write the query using the API of driver, the driver will built it into a real query to send to server, receive data, parse it to return data as a native data of your language.

Internaly, all client driver will turn the query that you write in driver language into an AST tree, then serialize them as JSON and send to server.

If you curious, you can fire up `tcpdump` and watch the raw query in JSON

```
1  tcpdump -nl -w - -i lo0 -c 500 port 28015|strings
```

An example of what above `tcpdump` return when I run those command(in Ruby):

```
1  r.db("foodb").table("users").with_fields("address").run
```

Once I ran this command, I see this via `tcpdump`:

[1,[96,[[15,[[14,[“foodb”]],”users”]],”address”]],{}]

So basically, the whole query is turned into a special JSON object by client driver. If you would like to dig deeper, the above query is actually translate in to this:

[QUERY_START, [WITH_FIELDS, [[TABLE, [[DB, [“foodb”]],”users”]],”address”]],{}]

Each of numbers is equivalent to a command in RethinkDB. Those number is predefined in RethinkDB. So basically, whatever you write using client driver API will be turn into an JSON array. Each of element often takes this form:

```
1  COMMAND, [Argument Array], {Option object}
```

It's similar to a function call when we have *function* name, follow by its argument and the last is an option object.

You can quickly sense a downside is that each of driver have different API to construct query. When you come to another language, you may feel very strange. The driver hide the real query behinds its API. It's kind of similar to how you use an ORM in SQL world to avoid writing raw SQL string. But

it's different because the ORM usually has its own API to turn query into raw query string, which in turns send to server using another driver with that database protocol. Here, we are having power of an ORM, but happen at driver level, because RethinkDB protocol is a powerful JSON protocol to help model query like function call, with argument and follow by parameter. In fact, ReQL is modeled after functional languages like Lisp or Hashkell.

If you would like to know more about ReQL at lower level, you should read more in official documents[12]

Using drivers

RethinkDB supports 3 official drives:

- Ruby
- NodeJS
- Python

These support all driver specifications[13]. The community drives such as Go, PHP probably won't support them all, if you used a different language and find something isn't right, it is probably not your fault.

All ReQL starts with `r`, its top level module to expose its public API. In NodeJS, we can use

```
var r = require('rethinkdb')
```

or Ruby

```
require 'rethinkdb'
include RethinkDB::Shortcuts
puts r.inspect
```

or in Go Lang

```
import (
  r "github.com/dancannon/gorethink"
)
```

Once we constructed ReQL with `r`, we have to call `run` method to execute it. The command will be submit to an active database connection. The database connection can be establish with `connect`.

[12] http://rethinkdb.com/docs/writing-drivers/
[13] http://rethinkdb.com/docs/driver-spec/

```
var r = require('rethinkdb')
var connection = r.connect({
  host: '127.0.0.1',
  port: '28015',
  db: 'test'
}, function (err, conn) {
  r.db('db').table('table').filter({type: 'anime'})
})
```

When creating the connection with `r.connect`, you can pass an `db` parameter to specify a default database to work on once connecting succesfully. It's similar to the current database of MySQL. Without setting this parameter, RethinkDB assumes `test` as default database.

To understand more about difference of API in different language, let looks at Go Lang driver[14]

Notice that we don't have any host, or database parameter now. They are `Address`and `Database` in Go Lang driver. Therefore, by using an un-official language, the API will be totally different with official API.

That's how beautiful it is because each language has its own design philosophy. Such as in Go lang, we cannot have a lower case field of a struct and expect it publicly available to outside. Using names such as `host` or `db` for connection option is impossible in Go lang.

Default database

Similar to MySQL, when you can issue `use database_name` to switch to another database. We can do that in RethinkDB by calling `use` command on a connection object.

```
connection.use('another_db')
```

In this small book, most of the time, we will use Data Exploer. Therefore we can use `r` without initialization and calling `run` method. *Data Exploer* will do that for us. Just keep in mind when you write code, you have to connect, and explicitly call `run` to, obviously, run the query.

Note that you don't have to switch to another database to access its table, you can just call `r.db('another_db')` before building query.

Repl

Repl means read-eval-print loop. To avoid burden of manually call `run` and passing connection object. Some driver offers a `repl` to help call `run` without any parameter.

Such as in Ruby:

[14] https://github.com/dancannon/gorethink
var connection *r.Session
connection, err := r.Connect(r.ConnectOpts{ Address: "localhost:28015", Database: "test", })
if err != nil { log.Fatalln(err.Error()) }

```
r.connect(:db => 'marvel').repl
r.table("test").run
```

JavaScript doesn't have a repl. I think because we can already used the Data Explorer.

Data Type

Why do we have to discuss about data type? We use dynamic language and we almost dicuss about Ruby and JavaScript most of time. But understanding data type allow us to read API document better. It helps us to understand why we can can `r.table.insert` but we cannot call like `r.table.filter().insert`. Aren't we still selecting data from table, so we should be able to insert data to it?

Data type helps us know that we can only call some method on some of data type

Each ReQL method can be call on one or many above data types. Take `update` command, when you browser the API document, you see

```
table.update(json | expr[, {durability: "hard", returnVals: false, nonAtomic: fa\
lse}]) → object

selection.update(json | expr[, {durability: "hard", returnVals: false, nonAtomic\
: false}]) → object

singleSelection.update(json | expr[, {durability: "hard", returnVals: false, non\
Atomic: false}]) → object
```

It means the command can be invoked on a table, or selection (eg: first 30 element of tables), or a single selection - a document is an example of single selection. The behaviour maybe different based on data type, even the command is same.

In RethinkDB, we have several data types. We will focus into those 2 kind for now:

- Basic data type
- Composite data type

Basic data type

These are usually the native data type in your language too:

```
* Number: any real numbers. RethinkDB uses double precision (64-bit) floating po\
int numbers internally
* String
* Time: This is native RethinkDB date time type. However, they will be converted\
 automatically to your native data type in your language by the driver.
* Boolean: True/False
* Null: Depend on your language, it can be nil, null,..
* Object: any valid JSON object. In JavaScript, it will be a normal object. In R\
uby, it can be a hash.
* Array: any valid JSON array.
```

The data type of a field or column can be change. If you assign a `number` to a field, you can still assign an value with different data type to that same field. So we don't have a static schema for tables.

We have a very useful command to get the type of any vaue. It's `typeOf`. Example:

```
r.db('foodb').table('foods')
  .typeOf()
//=>
"TABLE"

r.db('foodb').table('foods')
  .filter(r.row("name").match('A^'))
  .typeOf()
//=>
"SELECTION<STREAM>"
```

It's seems not very important to understand about data type at first. I really hope you should invest some time to use it frequently to understand the data type of a value.

To give a story. In MariaDB10.0/MySQL5.6, when data type doesn't match, an index may not be used. Let's say you have a field **name** with type `VARCHAR(255)` when you define it, then you create an index on that column. Query on that column with exact data type will make index kicked in. Let's come back MySQL a bit.

First I insert below records.

```
INSERT INTO foods(name) VALUES("100");
Query OK, 1 row affected, 1 warning (0.00 sec)
```

Below query will use index:

```
MariaDB [food]> EXPLAIN SELECT * FROM foods WHERE name="100";
+------+-------------+-------+-------+--------------------+--------------------\
-+---------+-------+------+-------+
| id   | select_type | table | type  | possible_keys      | key                \
 | key_len | ref   | rows | Extra |
+------+-------------+-------+-------+--------------------+--------------------\
-+---------+-------+------+-------+
|    1 | SIMPLE      | foods | const | index_foods_on_name | index_foods_on_name\
 | 257     | const |    1 |       |
+------+-------------+-------+-------+--------------------+--------------------\
-+---------+-------+------+-------+
```

But this query won't:

```
EXPLAIN select * from foods where name = 9;
MariaDB [food]> EXPLAIN SELECT * FROM foods WHERE name=100;
+------+-------------+-------+------+--------------------+------+---------+----\
--+------+-------------+
| id   | select_type | table | type | possible_keys      | key  | key_len | ref\
 | rows | Extra       |
+------+-------------+-------+------+--------------------+------+---------+----\
--+------+-------------+
|    1 | SIMPLE      | foods | ALL  | index_foods_on_name | NULL | NULL    | NUL\
L |  890 | Using where |
+------+-------------+-------+------+--------------------+------+---------+----\
--+------+-------------+
1 row in set (0.00 sec)
```

When we pass string *'9'*, the index is used. When we pass number *9*, the index isn't used.

Of if you have a date time column and you passing time as string, the index won't kicked in either.

The lesson here is we aboslutely should understand about data type.

Composite data type

We have 3 composite data types.

- Streams

:are list or array, but they're loaded in a lazy fashion. Instead of returning a whole array at once, meaning all data are read into memory, a cursor is return. A cursor is a pointer into the result set.

We can loop over cursor to read data when we need. Imagine instead of an array, and loop over it, you know iterate over the cursor to get next value. It allows you to iterator over a data set without building an entire array in memory. It's equivalent to PHP iterator, or Ruby iterator, or JavaScript iterator. Stream allows us access current element and keep track of current position so that we can, ideally call `next()` on a cursor to move to next element, until we reach to the end of array, it returns nil and iterator can stop. Because of that, we can work with large data set because RethinkDB doesn't need to load all of data and return to client. The nature of stream make it read-only, you cannot change the data while iterating over it.

- Selections

:represent subsets of tables, for example, the return values of **filter** or **get**. There are two kinds of selections, **Selection

- Tables

:are RethinkDB database tables. They behave like selections. However, they're writable, as you can insert and delete documents in them. ReQL methods that use an index, like `getAll`, are only available on tables. Because `index` are created on table level.

In short, you cannot modify streams, you can update or change value of selection but you cannot remove existed document, or insert new one. Tables allows you insert new document or remove existed one.

Sequence

RethinkDB document use `sequence` in lots of places. It's a particular data type. You can think of it as an shortwords for all: streams, table, seletion

Remember data types seems not much important but you should understand them well because it helps us understand the efficient of a query. If a query returns an array, it consumes lot of memory to hold the array.

Sorting data

When talking about data type, let think of how we sort them. It really doesn't matter in the order, what is important is the definition of sorting data.

Understanding sorting is important in RethinkDB because of its schemaless. The primary key may not be a numeric field, it can be a string. Moreover than that, a field can have whatever data type, how are we going to compare an object to a string when sorting.

Here is sorting order:

Arrays (and strings) sort lexicographically. Objects are coerced to arrays before sorting. Strings are sorted by UTF-8 codepoint and do not support Unicode collations.

Mixed sequences of data sort in the following order:

- arrays
- booleans
- null
- numbers
- objects
- binary objects
- geometry objects
- times
- strings

That mean array < booleans < null < numbers < objects < binary objects < geometry objects < times < strings.

Selecting data

In this section, we will learn how to get data out of RethinkDB. Most of the time, we will choose a db to work with, and chain into command `table`.

Select the whole table

Let's find all foods. This is same as `SELECT * FROM foods` in SQL.

```
r.db('foodb').table('foods')
//=>

[{
    "created_at": Wed Feb 09 2011 00: 37: 17 GMT - 08: 00,
    "creator_id": null,
    "description": null,
    "food_group": "Herbs and Spices",
    "food_subgroup": "Spices",
    "food_type": "Type 1",
    "id": 43,
    "itis_id": "29610",
    "legacy_id": 46,
    "name": "Caraway",
    "name_scientific": "Carum carvi",
    "picture_content_type": "image/jpeg",
    "picture_file_name": "43.jpg",
    "picture_file_size": 59897,
    "picture_updated_at": Fri Apr 20 2012 09: 38: 36 GMT - 07: 00,
    "updated_at": Fri Apr 20 2012 16: 38: 37 GMT - 07: 00,
    "updater_id": null,
    "wikipedia_id": null
}, {
    "created_at": Wed Feb 09 2011 00: 37: 18 GMT - 08: 00,
    "creator_id": null,
    "description": null,
    "food_group": "Herbs and Spices",
    "food_subgroup": "Spices",
    "food_type": "Type 1",
```

```
    "id": 67,
    "itis_id": "501839",
    "legacy_id": 73,
    "name": "Cumin",
    "name_scientific": "Cuminum cyminum",
    "picture_content_type": "image/jpeg",
    "picture_file_name": "67.jpg",
    "picture_file_size": 73485,
    "picture_updated_at": Fri Apr 20 2012 09: 32: 32 GMT - 07: 00,
    "updated_at": Fri Apr 20 2012 16: 32: 33 GMT - 07: 00,
    "updater_id": null,
    "wikipedia_id": null
},
...
]
```

You should get back an array of JSON object. By default, the data explorer will automatically paginate it and display a part of data.

Typing `r.db(db_name)` all the time is insane. We can drop it to use `r.table()` without calling `r.db()` if the table is in current selected database. Without any indication, the default database is `test`. On Data Exploer, without a `r.db` command, RethinkDB will use `test` as default database. Unfortunately we cannot set a default database with data exploer[15]

Counting

We can also `count` the table or any sequence by calling `count` command.

```
r.db('foodb').table('foods').count()
//=>
863
```

Select a single document by its primary key

To select a single element, we call `get` on a table, and passing its primary key value.

[15] https://github.com/rethinkdb/rethinkdb/issues/829

```
r.db('foodb').table('foods')
  .get(108)
//=>
{
    "created_at": Wed Feb 09 2011 00: 37: 20 GMT - 08: 00,
    "creator_id": null,
    "description": null,
    "food_group": "Herbs and Spices",
    "food_subgroup": "Herbs",
    "food_type": "Type 1",
    "id": 108,
    "itis_id": "32565",
    "legacy_id": 115,
    "name": "Lemon balm",
    "name_scientific": "Melissa officinalis",
    "picture_content_type": "image/jpeg",
    "picture_file_name": "108.jpg",
    "picture_file_size": 30057,
    "picture_updated_at": Fri Apr 20 2012 09: 33: 54 GMT - 07: 00,
    "updated_at": Fri Apr 20 2012 16: 33: 54 GMT - 07: 00,
    "updater_id": null,
    "wikipedia_id": null
}
```

Every document in RethinkDB includes a primary key field, its value is unique across cluster and is used to identify the document. The name of primary field is `id` by default. However, when you create a table, you have an option to change name of primary field. We will learn more about it later. Just keep a note here.

In RethinkDB, using of incremental primary key isn't recommended because that's hard in a cluster environment. To make sure the uniqueness of the new value, We have to check in every clusters somehow. RethinkDB team decides[16] to use an universal unique[17] id instead of an incremental value.

`get` command returns the whole document. What if we get a single field? Such as we only care about name? RethinkDB has a command call **bracket** for that purpose. In Ruby it's `[]`, and in JavaScript it's `()`.

We can do this in JavaScript:

[16] http://stackoverflow.com/questions/21020823/unique-integer-counter-in-rethinkdb

[17] http://en.wikipedia.org/wiki/Universally_unique_identifier

```
r.db('foodb').table('foods')
  .get(108)("name")
//=>
"Lemon balm"
```

Or in Ruby

```
r.connect.repl
r.db('foodb').table('foods').get(108)[:name].run
```

What special about **bracket** is that it return a single value of the field. The type of value is same type of value, not a subset of document. We can verify that with `typeOf` command:

```
r.db('foodb').table('foods')
  .get(108)
  ("name")
  .typeOf()
//=>
"STRING"
```

You can even get nested field with bracket:

```
r.db('foodb').table('test')
  .get(108)("address")("country")
```

with assumption that the document has **address** field is an object contains a field name **country**.

If you don't like the using of bracket, you can use `getField`(JavaScript) or `get_field`(Ruby) which have same effect:

```
r.db('foodb').table('foods')
  .get(108)
  .getField('name')
//=>
"Lemon balm"
```

How about getting a sub set of document, we can use `pluck` like this:

```
r.db('foodb').table('foods')
  .get(108)
  .pluck(get"name", "id")
//=>
{
  "id": 108 ,
  "name":  "Lemon balm"
}
```

`pluck` probably existed in many standard library of your favourite language. This example shows you how friendly ReQL is.

Select many documents by value of fields

To select many document based on value of field, We used `filter` method, and passing an object with expected value.

Let's find all food that were inserted into database on `2011`, the year I come to the US.

```
r.db('foodb').table('foods')
  .filter(r.row('created_at').year().eq(2011))
//=>Executed in 59ms. 40 rows returned, 40 displayed, more available
[{
    "created_at": Wed Feb 09 2011 00: 37: 17 GMT - 08: 00,
    "creator_id": null,
    "description": null,
    "food_group": "Herbs and Spices",
    "food_subgroup": "Spices",
    "food_type": "Type 1",
    "id": 43,
    "itis_id": "29610",
    "legacy_id": 46,
    "name": "Caraway",
    "name_scientific": "Carum carvi",
    "picture_content_type": "image/jpeg",
    "picture_file_name": "43.jpg",
    "picture_file_size": 59897,
    "picture_updated_at": Fri Apr 20 2012 09: 38: 36 GMT - 07: 00,
    "updated_at": Fri Apr 20 2012 16: 38: 37 GMT - 07: 00,
    "updater_id": null,
    "wikipedia_id": null
}
```

```
...
]
```

`r.row` is new to you, but no worry, it just means current document. We used `r.row('created_at')` to get value of **created_at** field, similar with how we use **bracket** on `get` command to get a single value. Because **created_at** is a datetime value, I get its year with, well, `year` command, then using `eq` to do an equal compare with 2011. Sound a lot, but above query is really simple and exlain itself. Sometimes I feel redundant to explain query but I have to write this book anyway.

We can also pass an filter object to do matching filter:

```
r.db('foodb').table('foods')
  .filter({
    food_type: 'Type 1',
    food_group: 'Fruits'
  })
//=>
[
{
  "created_at": Wed Feb 09 2011 00:37:15 GMT-08:00 ,
  "creator_id": null ,
  "description": null ,
  "food_group":  "Fruits" ,
  "food_subgroup":  "Tropical fruits" ,
  "food_type":  "Type 1" ,
  "id": 14 ,
  "itis_id":  "18099" ,
  "legacy_id": 14 ,
  "name":  "Custard apple" ,
  "name_scientific":  "Annona reticulata" ,
  "picture_content_type":  "image/jpeg" ,
  "picture_file_name":  "14.jpg" ,
  "picture_file_size": 29242 ,
  "picture_updated_at": Fri Apr 20 2012 09:30:49 GMT-07:00 ,
  "updated_at": Fri Apr 20 2012 16:30:49 GMT-07:00 ,
  "updater_id": null ,
  "wikipedia_id": null
},...
]
```

Passing an object will match exactly document with those field and value. In other words, passing an object is equal to passing multiple `eq` command and `and` command. Above query can re-write using expression:

```
r.db('foodb').table('foods')
  .filter(
    r.and(
      r.row('food_type').eq('Type 1'),
      r.row('food_group').eq('Fruits')
    )
  )
```

The object notation is much cleaner in this case.

From a selection of document, We can use `pluck` to get a subset of documents's field instead of returning the whole document. Similarly to how we use *bracket* to get a particular field

```
r.db('foodb').table('foods')
  .filter({
    food_type: 'Type 1',
    food_group: 'Fruits'
  })
  .pluck('id', 'name', 'food_subgroup')
//=>Executed in 70ms. 40 rows returned, 40 displayed, more available
[
  {
  "food_subgroup":  "Berries" ,
  "id": 75 ,
  "name":  "Black crowberry"
  }, {
  "food_subgroup":  "Tropical fruits" ,
  "id": 150 ,
  "name":  "Guava"
  }, {
  "food_subgroup":  "Tropical fruits" ,
  "id": 151 ,
  "name":  "Pomegranate"
  }, ...
]
```

By passing a list of field to `pluck`, we can get only those field.

Opposite of `pluck` is `without`. We passed a list of fields, and it removes those fiels from document.

```
r.db('foodb').table('foods')
  .filter({
    food_type: 'Type 1',
    food_group: 'Fruits'
  })
  .without("created_at", "picture_content_type", 'picture_file_name', 'picture_f\
ile_size', 'picture_updated_at')
//=> Executed in 52ms. 40 rows returned, 40 displayed, more available
[
{
  "creator_id": null ,
  "description": null ,
  "food_group":  "Fruits" ,
  "food_subgroup":  "Berries" ,
  "food_type":  "Type 1" ,
  "id": 75 ,
  "itis_id":  "23743" ,
  "legacy_id": 81 ,
  "name":  "Black crowberry" ,
  "name_scientific":  "Empetrum nigrum" ,
  "updated_at": Fri Apr 20 2012 16:29:43 GMT-07:00 ,
  "updater_id": null ,
  "wikipedia_id": null
},...
]
```

With simple filterting, we can easily pass an filter object as above. But what up with complex searching? Such as finding all foods whose name starts with character **N**. As you see at the beginning, we used `r.row` command to do a bit complex query.

```
r.db('foodb').table('foods')
  .filter(r.row('created_at').year().eq(2011))
```

Let's dive more into it.

Counting filter result

By calling `count` at the end of filter, we can count the result set of sequence r.db('foodb').table('foods') .filter({food_type: 'Type 1',food_group: 'Fruits'}) .count() //⇒ 122

r.row

`r.row` is our swiss army knife. It refers to current visited document. Literally, it's the document at which RethinkDB is accessing. You can think of it like **this** in a JavScript callback/iterator. Or think of it like current element in an iterator loop. It's very handy because we can call other ReQL command on it to achieve our filtering.

It somehow feel like jQuery filtering command. For an instance, we write this in JavaScript to filter all DOM element whose `data-type` value is *anime*.

```
1 $('.db').find('.table').filter(function() {
2   return $(this).data('type')=='anime'
3 })
```

In ReQL, using `filter` with filter object:

```
1 r.db('foodb').table('foods').filter({food_group: 'Fruits'})
```

We can re-write it with `r.row`

```
1 r.db('foodb').table('foods').filter(r.row('food_group').eq('Fruits'))
```

Breaking it down we have:

- r.row ⇒ current document
- ('type') ⇒ get value of field **type**
- .eq('anime') ⇒ return true if the value is equal to the argument, anime in this case

`r.row` a RethinkDB object, which we can continue call many method to filter or manipulation it. The expression that we pass into `filter` is a normal ReQL expression but evaluate to a boolean result. RethinkDB runs it and if the returned value is `true`, the document is included into result set. Ideally, any function that returns boolean result can used with `filter`. Note that the evaluation of filter expression run on RethinkDB server, therefore they has to be a valid ReQL expression, they cannot be any arbitrary language expression. You cannot write:

```
1 r.db('db').table('table').filter(r.row('type') == 'anime')
```

In manner of filter action, we usually execute comparison or some condition to be matched, RethinkDB gives us some kind of those method. You should refer to its API for extensive command. Usually, we can use `r.row` in combine with `pluck` or `without` or `bracket` command to narrow down data before comparing. Below are some function for that purpose:

- *eq(value)* check equal to value. similar to ==.
- *ne(value)* check not equal to value. similar to `!=`.
- *ge(value)* check greater than or equal value. similar to `>=`.
- *gt(value)* check greater than value. similar to `>`.
- *le(value)* check less than or equal value. similar to `<=`.
- *lt(value)* check less than value. similar to `<`.
- *add(value)* Sum two numbers, concatenate two strings, or concatenate 2 arrays.
- *sub()* Subtract two numbers.

Each of above command can be call on different data type. Eg, when you call `add` on an array, it will append the element to array. when you call on a string, it concat parameter to the original string. Or calling on a number and they just do arithmetic operation.

Run those command in data explorer:

```
r.expr(["foo", "bar"]).add(['forbar'])
//=>
[
  "foo" ,
  "bar" ,
  "forbar"
]

r.expr(2).add(10)
//=>
12

r.expr('foo').add("bar")
//=>
"foobar"
```

Note that the reason we use `r.expr` is that we have to turn the native object(array, number, string in our language) into RethinkDB data type, so that we can call command on those.

However, in Ruby it can be even shorter with r(r([‘foo’, ‘bar’]) + [‘foorbar’])

Basically, you have to remember that everything is evaluated on server, and RethinkDB command only callable on RethinkDB data type

You can find more about those document in RethinkDB doc, in group *Math and logic*[18].

Let’s apply what we learn, by finding al food where its name starts with character `R` and is a tropical fruits.

[18] http://rethinkdb.com/api/javascript/#mod

```
r.db("foodb").table("foods")
  .filter(
      r.row("name").match("^R")
      .and(
        r.row("food_subgroup").eq('Tropical fruits')
      )
  )
//=>
{
  "created_at": Wed Feb 09 2011 00:37:27 GMT-08:00 ,
  "creator_id": null ,
  "description": null ,
  "food_group":  "Fruits" ,
  "food_subgroup":  "Tropical fruits" ,
  "food_type":  "Type 1" ,
  "id": 234 ,
  "itis_id":  "506073" ,
  "legacy_id": 249 ,
  "name":  "Rambutan" ,
  "name_scientific":  "Nephelium lappaceum" ,
  "picture_content_type":  "image/jpeg" ,
  "picture_file_name":  "234.jpg" ,
  "picture_file_size": 71055 ,
  "picture_updated_at": Fri Apr 20 2012 09:43:04 GMT-07:00 ,
  "updated_at": Fri Apr 20 2012 16:43:05 GMT-07:00 ,
  "updater_id": null ,
  "wikipedia_id": null
}
```

Here we are usinbg `match` with an regular expression *^R* means any name starts with R, and using `and` to do an **and** operator with other boolean. Other boolean is result of getting field **food_subgroup** and compare with *tropical fruits*.

`filter` seems handy but it's actually limited. `filter` didn't leverage index. It scan and hold all data in memory. Of course, this isn't scale infinite. Only 100,000 records can be filter. For anything large than that, we have to use `getAll` or `between` which we will learn in chapter 5.

Now, let's try to find all *foods* which has more than 10 foods document in its group. We probably think of a simple solution like this: for each of document, we get its **food_group** and count how many items has that same food group, if the result is greater than 10, we return true, so that it will be included in `filter` result. We may have duplicate result but let's try this naieve soltuion:

```
r.db('foodb').table('foods')
  .filter(
    r.db('foodb').table('foods')
      .filter(
        {food_group: r.row("food_group")}
      )
      .count()
      .gt(10)
  )
```

Query looks good but when we run, we get this:

```
RqlCompileError: Cannot use r.row in nested queries.  Use functions instead in:
r.db("foodb").table("foods").filter(r.db("foodb").table("foods").filter({food_gr\
oup:
r.row("food_group")}).count().gt(10))
```

Basically, we have nested query here, and RethinkDB doesn't know which query `r.row` should belong to, is it parent query, or the sub query? In those case, we have to use filter with function. Let's move to next chapter.

Filter with function

Beside passing an ReQL expression, we can also use a function which return true or false to `filter`.

Let's try previous example.

```
r.db('foodb').table('foods')
  .filter(function (food) {
   return r.db('foodb').table('foods').filter({food_group: food("food_group")}).\
count().gt(10)
  })
```

Now, we no longer using `r.row`, we pass an anonymous function with a single parameter(which we can name whatever), when itereating over the table, RethinkDB call this function, and passing current document as its first argument. By using function, we can still access current document, without using `r.row`, and clearly bind current document to a variable, so that we can access its value and avoid conflicting. Here, we name our argument **food**, instead of writing:

```
filter({food_group: r.row("food_group")})
```

We will write:

```
filter({filter_group: food("food_group")})
```

And we using boolean value, `count().gt(10)` here, as result of function. Filter with function helps us write query with complex logic.

Pagination data

We rarely want a whole sequence of document, usually we care about a subset of data such as pagination data. In this section, we go over commands: order, limit and skip.

Order data

So far, we only select data and accept default ordering. Let's control how they appear:

```
r.db('foodb').table('foods')
  .filter(function (food) {
   return r.db('foodb').table('foods').filter({food_group:
```

food("food_group")}).count().gt(10) }) .orderBy("name") //⇒Executed in 5.69s. 821 rows returned [{ "created_at": { "$reql_type$": "TIME", "epoch_time": 1297240650, "timezone": "-08:00" }, "creator_id": null, "description": null, "food_group": "Aquatic foods", "food_subgroup": "Mollusks", "food_type": "Type 1", "id": 280, "itis_id": "69493", "legacy_id": 307, "name": "Abalone", "name_scientific": "Haliotis", "picture_content_type": "image/jpeg", "picture_file_name": "280.jpg", "picture_file_size": 99231, "picture_updated_at": { "$reql_type$": "TIME", "epoch_time": 1334940073, "timezone": "-07:00" }, "updated_at": { "$reql_type$": "TIME", "epoch_time": 1334965273, "timezone": "-07:00" }, "updater_id": null, "wikipedia_id": null }, ...]

We re-used above `filter` query, but append `orderBy("name")`. If you notice, the above command run quite long **Executed in 5.56s. 821 rows returned** and all rows are returned instead of a streams as usual. When we are calling `orderBy` without specifing an index, it load all data into memory to sort, which is both of slow and in-efficient. We will learn more about ordering with index in chapter 5. For now, let's continue with this method because, well, they are easy to use, at first :D

We can reverse order by applying `r.desc` command:

```
r.db('foodb').table('foods')
  .filter(function (food) {
    return r.db('foodb').table('foods').filter({food_group: food("food_group")})\
.count().gt(10)
  })
  .orderBy(r.desc("name"))
```

We can order on `table` too, not just filter:

```
r.db('foodb').table('foods')
  .orderBy(r.desc("name"))
```

We can order by multiple field, at a time

```
r.db('foodb').table('foods')
  .orderBy(r.desc("name"), r.asc("created_at"))
```

We order by descending order on field `name` and ascending on field `created_at`.

One more thing to note is that RethinkDB doesn't order document based on time they are inserted by default. The order seems in an unpredicted way without explicitly setting an order . In MySQL, for example, even without any index, the default order will be exactly same as you insrted the document. However, in RethinkDB it doesn't. I guess this is because it's distributed.

We can combine some document commands with `orderBy` too. Such as `pluck` only an useful set of fields:

```
r.db('foodb').table('foods')
  .pluck("id", "name", "food_group")
  .orderBy(r.desc("name"), r.asc("created_at"))
//=>Executed in 122ms. 863 rows returned
[
  {
      "food_group": "Milk and milk products",
      "id": 634,
      "name": "Yogurt"
  },
  {
      "food_group": "Milk and milk products",
      "id": 656,
```

```
        "name": "Ymer"
    },
    {
        "food_group": "Aquatic foods",
        "id": 523,
        "name": "Yellowtail amberjack"
    },
    ...
]
```

Limiting data

Once we have an ordering sequence, we usually want to select a limit number of document instead of the whole sequence. We use command `limit(n)` for this purpose. It get *n* elements from the sequence or array.

```
r.db('foodb').table('foods')
  .pluck("id", "name", "food_group")
  .orderBy(r.desc("name"), r.asc("created_at"))
  .limit(4)
//=>Executed in 107ms. 2 rows returned
[{
    "food_group": "Milk and milk products",
    "id": 634,
    "name": "Yogurt"
}, {
    "food_group": "Milk and milk products",
    "id": 656,
    "name": "Ymer"
}, {
    "food_group": "Aquatic foods",
    "id": 523,
    "name": "Yellowtail amberjack"
}, {
    "food_group": "Aquatic foods",
    "id": 522,
    "name": "Yellowfin tuna"
}]
```

`limit` get us a number of document that we want, but it always start from the beginning of sequence. To start selecting data starts from a position, we used `skip`.

Skip

As its name, `skip(n)` ignore a number of element from the head of sequence.

```
r.db('foodb').table('foods')
  .pluck("id", "name", "food_group")
  .orderBy(r.desc("name"), r.asc("created_at"))
  .skip(2)
  .limit(2)
//=> Executed in 97ms. 2 rows returned
[{
    "food_group": "Aquatic foods",
    "id": 523,
    "name": "Yellowtail amberjack"
}, {
    "food_group": "Aquatic foods",
    "id": 522,
    "name": "Yellowfin tuna"
}]
```

Access Nested field

As you know, RethinkDB document is a JSON object. Very likely we have two or more level of data structure. So how we can access those nested field, or to drill down the fields.

Let's beging this chapter by creating some sample data. Just copy and paste, ignore the syntax for now because we save them for chapter 4.

First, create table on test db.

```
r.tableCreate("books")
```

Then, insert sample data:

```
r.table("books")
  .insert([
        {
            id: 1,
            name: "Simply RethinkDB",
            address: {
                country: {
                    code: "USA",
                    name: "The United State of America"
                }
            },
            contact: {
                phone: {
                    work: "408-555-1212",
                    home: "408-555-1213",
                    cell: "408-555-1214"
                },
                email: {
                    work: "bob@smith.com",
                    home: "bobsmith@gmail.com",
                    other: "bobbys@moosecall.net"
                },
                im: {
                    skype: "Bob Smith",
                    aim: "bobmoose",
                    icq: "nobodyremembersicqnumbers"
                }
            }
        },
        {
            id: 2,
            name: "TKKG",
            address: {
                country: {
                    code: "GER",
                    name: "Germany"
                }
            },
            contact: {
                phone: {
                    work: "408-111-1212",
                    home: "408-111-1213",
```

```
                    cell: "408-111-1214"
                },
                email: {
                    work: "bob@gmail.com",
                    home: "bobsmith@axcoto.com",
                    other: "bobbys@axcoto.com"
                },
                im: {
                    skype: "Jon",
                    aim: "Jon",
                    icq: "nooneremembersicqnumbers"
                }
            }
        }
  ])
```

Depend on your language, you will usually have some way to access nested field, by following the nested path. In above example, let's say we want to access ***skype** im, the path is:

contact -> im -> skype

Using JavaScript driver, we will use **bracket** to access field and sub field.

```
r.table('books').get(1)('contact')('im')('skype')
//=>
"Bob Smith"
```

While as, in Ruby driver, bracket notation is `[field]`

```
r.table('books').get(1)['contact']['im']['skype']
```

We can keep calling bracket to get the final nested field follow the path. Not just a single document, we can use bracket on table level too:

```
r.table('books')('address')('country')
[
{
  "code":  "GER" ,
  "name":  "Germany"
}, {
  "code":  "USA" ,
  "name":  "The United State of America"
}
]
```

Or using in combination with filter, on selection:

```
r.table('books')
  .filter({id: 1})('address')('country')('name')
//=>
"The United State of America"
```

Beside using **bracket** command, we can also using `getField` if that feel more nature:

```
r.table('books')
  .getField('contact')('email')
//=>
[
{
  "home": bobsmith@axcoto.com, »
  "other": bobbys@axcoto.com, »
  "work": bob@gmail.com, »
}, {
  "home": bobsmith@gmail.com, »
  "other": bobbys@moosecall.net, »
  "work": bob@smith.com, »
}]
```

At the end of the day all you have to remember is to drill down the path with a chain of bracket command.

Wrap Up

We now have some basic understanding:

```
1. ReQL always starts with `r`.
2. ReQL is tie to your language depend on language driver.
3. Default database
3. Find an document by its primary
4. Access table data and filter data by some condition
5. Access nested field
```

We will learn more about advanced query in other chapter. For now, let's move on and try to write some data into RethinkDB.