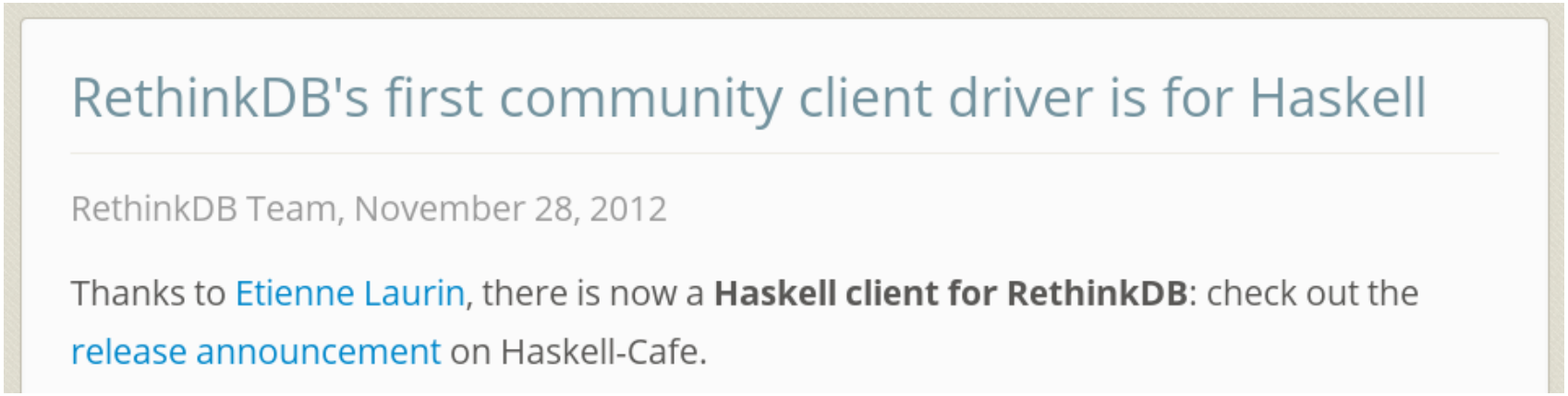
A RethinkDB driver for Haskell

Étienne Laurin

2014-11-13

History



RethinkDB's first community client driver is for Haskell

RethinkDB Team, November 28, 2012

Thanks to Etienne Laurin, there is now a **Haskell client for RethinkDB**: check out the release announcement on Haskell-Cafe.

Figure : Blog Post

One of Many

Installing RethinkDB client drivers

Official drivers »

- JavaScript
- Ruby
- Python

Community-supported drivers »

These drivers have been updated to at least the RethinkDB v1.11 protocol.

- C# / .NET
- Clojure
- Common Lisp
- Dart
- Go
- Haskell
- Java by @npiv
- Java by @dkhenry
- Node.js
- Perl
- PHP
- Scala by @kclay

Why Haskell?

- Abstractions
- Strong static typing
- GHC

Why RethinkDB?

- Powerful, composable query language (ReQL)
- Joins, changefeeds, geo indexes, HTTP, ...
- Sharding and replication

RethinkDB and Functional Programming

RethinkDB is switching over to Lisp

RethinkDB Team, April 01, 2010

Over the past few months we've had many architectural discussions about the future of database technology. It quickly became apparent to us that C++, the language we used to develop RethinkDB, is not sufficiently expressive to build the next-generation database product. We realized that in order to design the database system of the future, we need to use a programming language of the future as well.

Figure : Announcement

Setup

```
$ cabal update
$ cabal install rethinkdb -j
$ ghci

> import Database.RethinkDB.NoClash
> import qualified Database.RethinkDB as R
> :set -XOverloadedStrings
> default (Datum, ReQL, Integer, String)

> h <- connect "localhost" 28015 def
> run h dbList
["test","muni"]
```

Download These Slides

http://atnnn.github.io/rethinkdb/haskell-driver-2014-11-13.pdf

http://goo.gl/UOX4iD

Sample Data

```
> run h $ table "routes" ! 1
{"display_name":"18-46th Avenue","id":"18"}

> run h $ table "runs" ! 1
{"display_name":"Inbound to Fisherman's Wharf",
"stops":["15926","14882",...],
"direction_name":"Inbound",
"id":"08BX_IB",
"route_id":"8BX"}

> run h $ table "stops" ! 1
{"display_name":"Merchant Rd & Golden Gate Br.",
"location":Point<[-122.47587,37.8066699]>,
"id":"114776"}
```

Compared to JavaScript

- Asynchronous queries
- Most commands have the same name

http://rethinkdb.com/api/javascript/

Different Syntax

In JavaScript:

```
r.expr({foo: "bar"})('foo')
```

In Haskell:

```
["foo" := "bar"] ! "foo"
```

- ! to access fields
- := to build objects
- expr can usually be omitted

```
(!) :: Expr obj => obj -> ReQL -> ReQL
```

Function Composition

In JavaScript:

```
r.table("runs")
 .map(r.row("stops").count())
 .sum()
```

In Haskell:

```
R.sum
 . R.map (\row -> count (row!"stops"))
 . table $ "runs"

(.) :: (b -> c) -> (a -> b) -> a -> c
R.map :: (Expr a, Expr seq) => (ReQL -> a) -> seq -> ReQL
```

Function Composition

In JavaScript:

```
r.table("runs")
 .map(r.row("stops").count())
 .sum()
```

In Haskell:

```
table "runs"
 # R.map (\row -> (row!"stops") # count)
 # R.sum

(#) :: a -> (a -> b) -> b
```

Types

ReQL: an operation that can be performed on the server

```
> table "routes" # get "N" # (!"display_name") :: ReQL
get(table("routes"), "N")["display_name"]

> run h it
"N-Judah"
```

Types

`Expr`: build queries from other types

```
class Expr e where
  expr :: e -> ReQL

table "routes" :: Table
get "N" :: Expr s => s -> ReQL
table "routes" # get "N" :: ReQL
```

Types

`Num`, `Fractional` and `Floating` instances: overload the built-in math operators:

```
> count (table "routes") / 2 + 1 :: ReQL
```

No `Ord` instance:

```
> run h $ count (table "stops") > 3000
Error: Could not deduce (Ord ReQL) arising from a use of >
> run h $ count (table "stops") R.> 3000
true
```

Types

`Datum`: query results

```
data Datum
  = Null
  | Bool Bool
  | String Text
  | Number Double
  | Array Array
  | Object Object
  | Time ZonedTime
  | Point LonLat
  | Line Line
  | Polygon Polygon
  | Binary ByteString
```

Running Queries

```
> run h $ table "stops" ! "display_name" :: IO [String]
["19th Ave & Holloway Ave",
 "Merchant Rd & Golden Gate Br.", ...]

> c <- run h $
    table "stops" ! "display_name" :: IO (Cursor String)
> next c
Just "19th Ave & Holloway Ave"

> run h $ table "stops" ! "display_name"
      # sample 2 :: IO (String, String)
("Geary Blvd & Laguna St","City College Bookstore")

> run h $ table "stops" ! 1 :: IO (Map String Datum)
```

Optional Arguments

In JavaScript:

```
r.circle([-122.411017, 37.773589], 500)
r.circle([-122.411017, 37.773589], 500, {unit: 'm'})
```

In Haskell:

```
circle [-122.411017, 37.773589] 500
ex circle [unit Meter] [-122.411017, 37.773589] 500
```

For example:

```
> let heavybit = [-122.411017, 37.773589]
> run h $ table "stops" # R.filter (\stop ->
    ex circle [unit Meter] heavybit 200
    # includes (stop!"location"))
  # R.map (!"display_name")
["Harrison St & 9th St","Folsom St & 9th St"]
```

Aggregations

```
> run h $ table "runs" # group (!"route_id") count
[{"group":"1","reduction":4},
 {"group":"10","reduction":2}, ...]

> run h $ table "runs"
    # group (!"route_id") count
    # group (!"reduction") count
[{"group":1,"reduction":1},
 {"group":2,"reduction":76},
 {"group":3,"reduction":1},
 {"group":4,"reduction":3}]

> run h $ table "runs"
    # group (!"route_id") (avg . R.map count . (!"stops"))
[{"group":"1","reduction":25.25},
 {"group":"10","reduction":66}, ...]
```

Multiple Aggregations

Not supported natively by RethinkDB

```
> run h $ table "runs"
    # group (!"route_id")
      ((\x -> [avg x, R.sum x, R.max x])
       . R.map count . (!"stops"))
[{"group":"1","reduction":[25.25,101,49]},
 {"group":"10","reduction":[66,132,67]}, ...]
```

Multiple Aggregations

```
> expr $ mapReduce
    ((\x -> [avg x, R.sum x, R.max x])
     . R.map count . (!"stops"))

(\b -> ((\g -> [div(g[0][0], g[0][1]), g[1], g[2]]))(
  reduce(map(b, (\h -> [
        [((\d -> count(d)))(h["stops"]), 1],
        ((\e -> count(e)))(h["stops"]),
        ((\f -> count(f)))(h["stops"])])),
    (\i j -> [
      [add(i[0][0], j[0][0]), add(i[0][1], j[0][1])],
      add(i[1], j[1]),
      branch(gt(i[2], j[2]), i[2], j[2])]))))
```

Importing Data

```
> let api path = str . (++ path) $
    "http://proximobus.appspot.com/agencies/sf-muni"

> run h $ tableCreate "routes"
> run h $ table "routes" #
    insert (http (api "/routes.json") def ! "items")
{"inserted":81}

> run h $ table "routes" # ex update [nonAtomic] (\route ->
    http (api "/routes/" + (route!"id") + ".json") def)
{"replaced":81}

> run h $ tableCreate "stops"
> run h $ table "routes" ! "id" # forEach (\id ->
    table "runs" # insert (
    http (api "/routes/" + id + "/runs.json") def ! "items"))
{"inserted":168}
```

Importing Data

```
> run h $ createTable "stops"
> run h $ table "runs" # forEach (\run ->
    flip apply [
      http (api "/routes/" + (run!"route_id") + "/runs/"
        + (run!"id") + "/stops.json") def ! "items"]
    $ \stops -> expr [
      table "runs" # get (run!"id") #
        update (const ["stops" := stops!"id"]),
      table "stops" # insert stops])
{"inserted":3691}

> run h $ table "stops" # ex update [nonAtomic] (\stop -> [
    "latitude" := remove,
    "longitude" := remove,
    "location" := point (stop!"longitude") (stop!"latitude")])
{"replaced":3691}
```

Visualising Data

```
import Geodetics.Grid
import Geodetics.Geodetic
import Geodetics.TransverseMercator

project :: LonLat -> (Double, Double)
project (LonLat lon lat) = let
  gd lat lon =
    Geodetic (lat*~degree) (lon*~degree) (0*~meter) WGS84
  pos = gd lat lon
  sf_sw = gd 37.614775 (-122.522278)
  offset = GridOffset (0*~meter) (0*~meter) (0*~meter)
  pt = toGrid (mkGridTM sf_sw offset _1) pos
  convert f = fromRational $ toRational (f pt /~ meter)
  in (convert eastings, convert northings)
```

Visualising Data

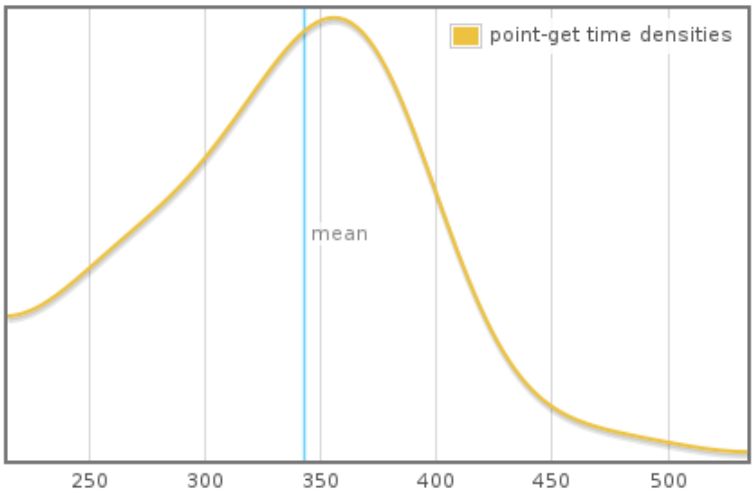
```
import Diagrams.Prelude
import Diagrams.Backend.SVG.CmdLine
import Data.Colour.SRGB

main = do
  h <- fmap (use "muni") $ R.connect "localhost" 28015 def
  runs <- run h $ flip R.map (table "runs") $ \run -> [
    (\r -> expr [r!"bg_color", r!"fg_color"]) `R.apply`
      [get (run!"route_id") (table "routes")],
    run!"stops" # R.map (\stop ->
      table "stops" # get stop # (!"location"))]

  let lines = flip map runs $ \((bg, fg), pts) ->
    let line = fromVertices (map (p2 . project) pts)
    in (line # lineColor (sRGB24read fg) # lw medium) `atop`
         (line # lineColor (sRGB24read bg) # lw thick)
  mainWith (mconcat lines :: Diagram B R2)
```

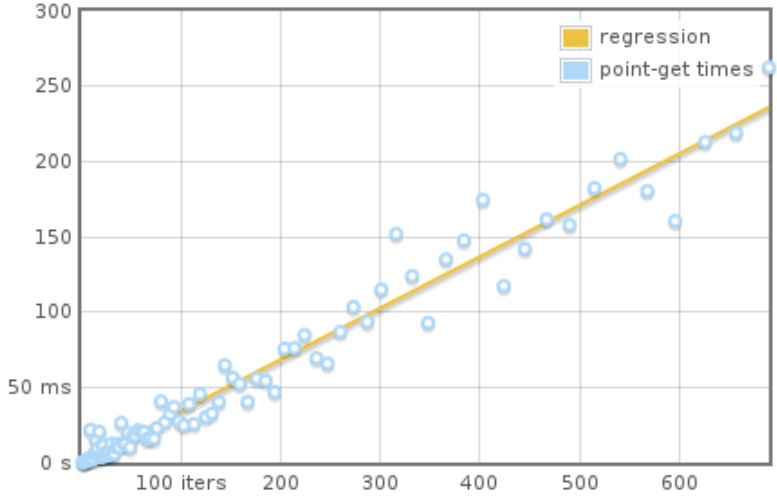
Rendered

Figure : All Routes

Benchmarks

point-get



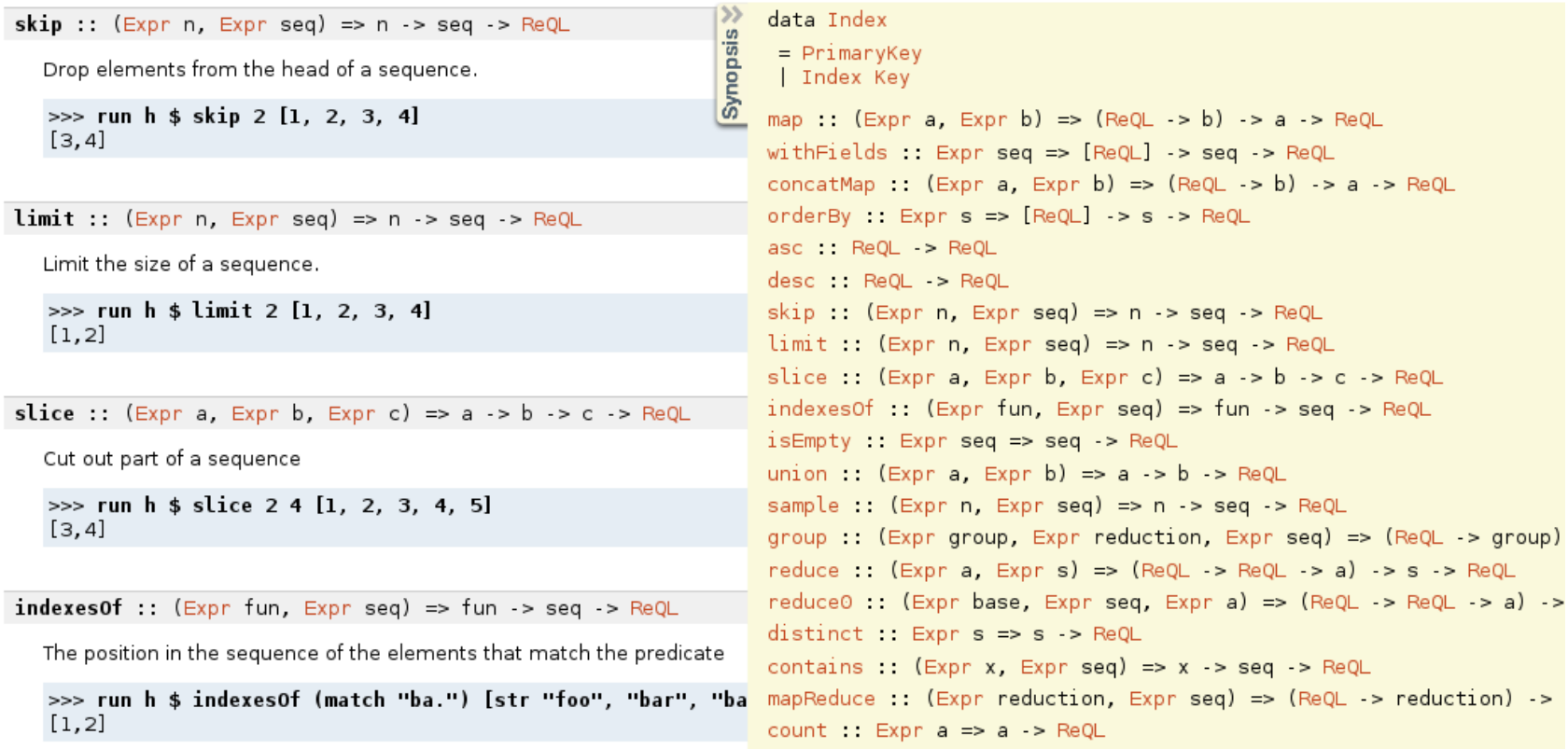


	lower bound	estimate	upper bound
OLS regression	319 µs	342 µs	364 µs
R² goodness-of-fit	0.944	0.964	0.981
Mean execution time	323 µs	343 µs	361 µs
Standard deviation	49.1 µs	62.0 µs	80.8 µs

Figure : Criterion Benchmarks

Documentation

https://hackage.haskell.org/package/rethinkdb/



skip :: (Expr n, Expr seq) => n -> seq -> ReQL

Drop elements from the head of a sequence.

```
>>> run h $ skip 2 [1, 2, 3, 4]
[3,4]
```

limit :: (Expr n, Expr seq) => n -> seq -> ReQL

Limit the size of a sequence.

```
>>> run h $ limit 2 [1, 2, 3, 4]
[1,2]
```

slice :: (Expr a, Expr b, Expr c) => a -> b -> c -> ReQL

Cut out part of a sequence

```
>>> run h $ slice 2 4 [1, 2, 3, 4, 5]
[3,4]
```

indexesOf :: (Expr fun, Expr seq) => fun -> seq -> ReQL

The position in the sequence of the elements that match the predicate

```
>>> run h $ indexesOf (match "ba.") [str "foo", "bar", "ba
[1,2]
```

Synopsis

```
data Index
 = PrimaryKey
 | Index Key

map :: (Expr a, Expr b) => (ReQL -> b) -> a -> ReQL
withFields :: Expr seq => [ReQL] -> seq -> ReQL
concatMap :: (Expr a, Expr b) => (ReQL -> b) -> a -> ReQL
orderBy :: Expr s => [ReQL] -> s -> ReQL
asc :: ReQL -> ReQL
desc :: ReQL -> ReQL
skip :: (Expr n, Expr seq) => n -> seq -> ReQL
limit :: (Expr n, Expr seq) => n -> seq -> ReQL
slice :: (Expr a, Expr b, Expr c) => a -> b -> c -> ReQL
indexesOf :: (Expr fun, Expr seq) => fun -> seq -> ReQL
isEmpty :: Expr seq => seq -> ReQL
union :: (Expr a, Expr b) => a -> b -> ReQL
sample :: (Expr n, Expr seq) => n -> seq -> ReQL
group :: (Expr group, Expr reduction, Expr seq) => (ReQL -> group)
reduce :: (Expr a, Expr s) => (ReQL -> ReQL -> a) -> s -> ReQL
reduce0 :: (Expr base, Expr seq, Expr a) => (ReQL -> ReQL -> a) ->
distinct :: Expr s => s -> ReQL
contains :: (Expr x, Expr seq) => x -> seq -> ReQL
mapReduce :: (Expr reduction, Expr seq) => (ReQL -> reduction) ->
count :: Expr a => a -> ReQL
```

Figure : Haddock Documentation

Questions?

- IRC: Freenode #RethinkDB
- Source code and Issue tracker: https://github.com/AtnNn/haskell-rethinkdb/
- Downloads and documentation: https://hackage.haskell.org/package/rethinkdb/

Questions?

- IRC: Freenode #RethinkDB
- Source code and Issue tracker: https://github.com/AtnNn/haskell-rethinkdb/
- Downloads and documentation: https://hackage.haskell.org/package/rethinkdb/

- Why are there no Monads in this presentation?