# uptasticsearch

An R data frame client for Elasticsearch

/jameslamb @_jameslamb

# My friends and I wrote an R package called uptasticsearch.

/jameslamb

/austin3dickey

/ngparas

# Let’s talk about Elasticsearch

# Elasticsearch is a document-based, non-relational, schema-optional, distributed, highly-available data store

- **Document-based** → Single “record” is a JSON object which follows some schema (called a “mapping”) but is extensible and whose content varies within an index
- **Non-relational** → Documents are stored in indices and keyed by unique IDs, but explicit definition of relationships between fields is not required
- **Schema-optional** → You can enforce schema-on-write restrictions on incoming data but don’t have to
- **Distributed** → data in ES are distributed across multiple shards stored on multiple physical nodes (at least in production ES clusters)
- **Available** → Query load is distributed across the cluster without the need for a master node. No single point of failure

Let’s go through each of these points...

# Document stores are databases that store unstructured or semi-structured text

## Each “record” in Elasticsearch is a JSON document.

Information on how the cluster responded. In this case, 4 shards participated in responded to the request.

This tells you how many documents matched your query.


```
{
  "took": 195,
  "timed_out": false,
  "_shards": {
    "total": 4,
    "successful": 4,
    "failed": 0
  },
  "hits": {
    "total": 1517,
    "max_score": 1,
    "hits": [
      {
        "_index": "customers",
        "_type": "customer",
        "_id": "CN00124568",
        "_score": 1,
        "_source": {
          "name": "Mr. Blue",
          "serviceLevel": "Platinum",
          "siteVisits": 110
        }
      },
      {
```


The “hits.hits” portion of the response contains an array of documents. Each document in this array is equivalent to one “record” (think 1 row in a relational DB)

The fields starting with “_” are default ES fields, not data we indexed into the cluster

# Schemas are optional but strongly encouraged in Elasticsearch

Elasticsearch is “schema-optional” because you can enforce type restrictions on certain fields, but the databases will not reject documents that have additional fields not present in your mapping

This applies to the customer index. For now, just think of: index in ES = table in RDBMS

```
{
"customer": {
  "mappings": {
    "_default_": {
      "_timestamp": {
        "enabled": true,
        "store": true
      },
      "properties": {
        "firstContactDate": {
          "type": "date",
          "store": true,
          "fields": {
            "search": {
              "type": "date",
              "store": true,
              "format": "dateOptionalTime"
            }
          },
          "format": "dateOptionalTime"
        }
```

This block tells ES to index a timestamp with every new document passed to this index. Can be user-generated or auto-generated by ES

Example mapping for a field called firstContactDate

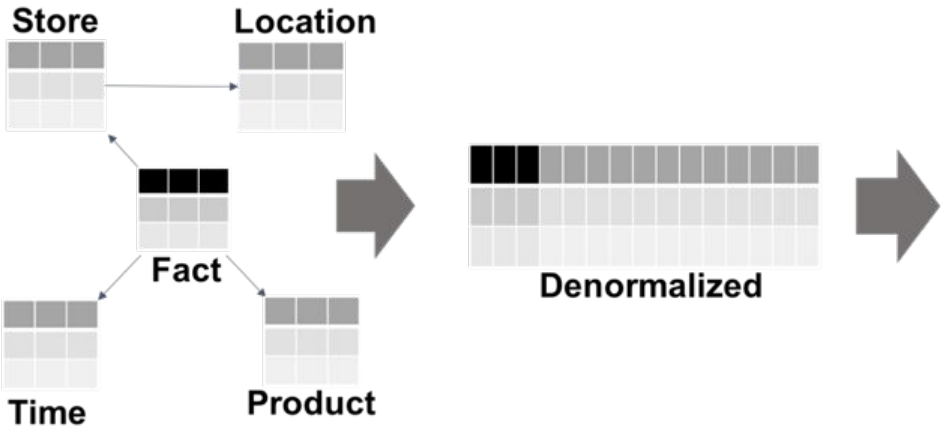
**store: true** = tells Elasticsearch to store the raw values of this field, not just references in an index

**fields: {}** = additional alternative fields to create from raw values passed to this one. In this case, a field called firstContactDate.search will exist that users can query with the “dateOptionalTime” format

# Non-relational = No Joins!

**Elasticsearch has no support for query-time joins.**

**Data that need to be used together by applications must be stored together. This is called "denormalization".**




**Elasticsearch mapping:**

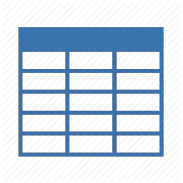
Time.field1
Time.field2
Store.field1
Store.Location.field1
Store.Location.field2
Store.Field2
Product.field1
Product.field2
Fact1
Fact2
…

Image credit:
Contactually

# NoSQL involves “denormalizing” your data. This makes these databases very efficient for serving certain queries, but inefficient for arbitrary questions

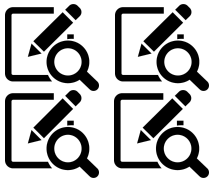
**RDBMS Workflow**


Execute Query
(DB handles joins)

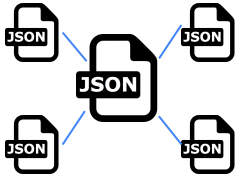


Train Model

**NoSQL Workflow**

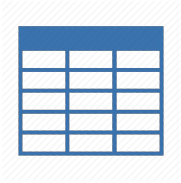


Execute several queries




(join results)


(Make a rectangle)



Train Model

# We wrote an R package called “uptasticsearch” to reduce friction between data scientists and data in Elasticsearch. We wanted data scientists to say “give me data” and get it


```
[
  {
    "_source": {
      "dateTime": "2017-01-01",
      "userName": "Gauss",
      "details": {
        "interactions": 400,
        "userType": "active",
        "appData": [
          {"appName": "farmville","minutes": 500},
          {"appName": "candy_crush","minutes": 350},
          {"appName": "angry_birds","minutes": 422}
        ]
      }
    }
  },
  {
    "_source": {
      "dateTime": "2017-02-02",
      "userName": "Will Hunting",
      "details": {
        "interactions": 5,
        "userType": "very_active",
        "appData": [
          {"appName": "minesweeper","minutes": 28},
          {"appName": "pokemon_go","minutes": 190},
          {"appName": "pokemon_stay","minutes": 1},
          {"appName": "block_dude","minutes": 796}
        ]
      }
    }
  }
]
```


```r
# Load dependencies
library(uptasticsearch)
library(data.table)

# Read in your query (could be specified as an R string instead)
SEARCH_QUERY <- paste0(readLines("query1.json"), collapse = "")

# Execute with uptasticsearch
resultDT <- uptasticsearch::es_search(es_host = "http://mydb.mycompany.com:9200"
                                      , es_index = "gameplay"
                                      , query_body = SEARCH_QUERY)
# Unpack arrays
resultDT <- uptasticsearch::unpack_nested_data(resultDT, "details.appData")
View(resultDT)
```

resultDT

Filter

| | appName | minutes | userName | details.interactions | details.userType |
|---|---|---|---|---|---|
| 1 | farmville | 500 | Gauss | 400 | active |
| 2 | candy_crush | 350 | Gauss | 400 | active |
| 3 | angry_birds | 422 | Gauss | 400 | active |
| 4 | minesweeper | 28 | Will Hunting | 5 | very_active |
| 5 | pokemon_go | 190 | Will Hunting | 5 | very_active |
| 6 | pokemon_stay | 1 | Will Hunting | 5 | very_active |
| 7 | block_dude | 796 | Will Hunting | 5 | very_active |

# uptasticsearch's API is intentionally less expressive than the Elasticsearch HTTP API. We wanted to narrow the focus to make it easy to use for people who are not sys admins or engineers

## uptasticsearch

```r
# Read in query
SEARCH_QUERY <- paste0(readLines("query1.json"), collapse = "")

# Go get data
resultDT <- es_search(es_host = "http://es.mycompany.com:9200"
                      , es_index = "gameplay"
                      , query_body = SEARCH_QUERY)
```

## ropensci/elastic:

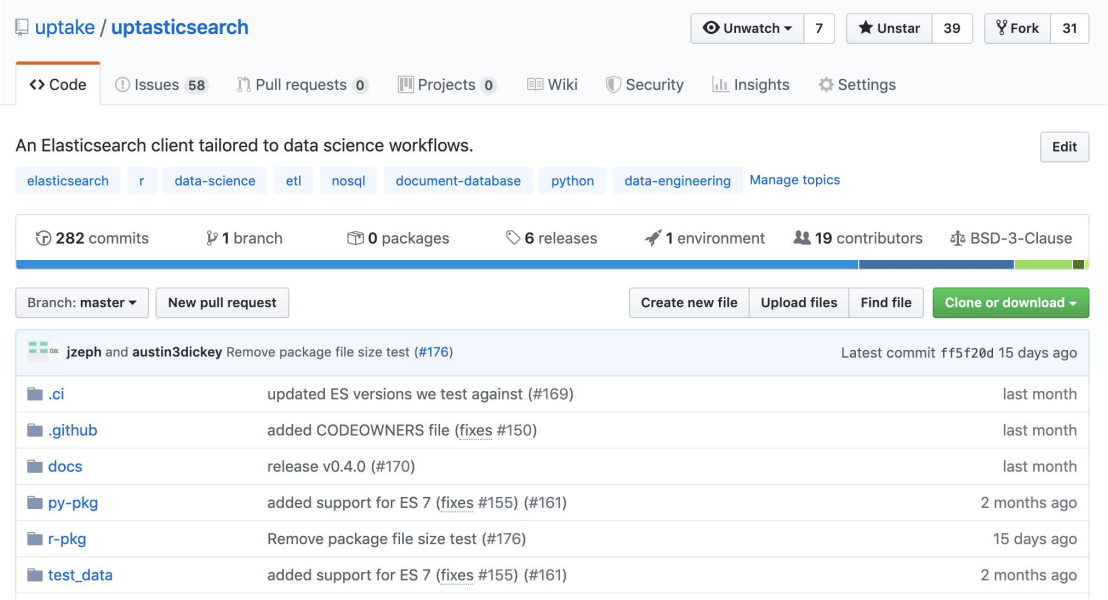
```r
# Connect to ES
esInfo <- elastic::connect(es_host = "es.mycompany.com")

# Build a search URL
searchURL <- paste0("http://es.mycompany.com:9200/gameplay/_search?size=1000&scroll=2m")

# Read in query
SEARCH_QUERY <- paste0(readLines("query1.json"), collapse = "")

# Grab the first result
firstResult <- httr::POST(url = searchURL, body = SEARCH_QUERY) %>%
    httr::content(., as = "text")

# Process into a data.table
jsonList <- jsonlite::fromJSON(firstResult, flatten = TRUE)
resultDT <- data.table::as.data.table(jsonList[["hits"]][["hits"]])

# Grab the scroll ID
scrollId   <- enc2utf8(firstResult$`_scroll_id`)

# Scroll over additional pages
hitsPulled <- nrow(resultDT)
while (TRUE){

    # Grab a page of hits, break if we got back an error
    result  <- httr::POST(url = searchURL, body = scrollId)
    resultJSON  <- httr::content(result, as = "text")

    # Parse to JSON to get total number of documents + new scroll_id
    resultList <- jsonlite::fromJSON(resultJSON, simplifyVector = FALSE)

    # Break if we got nothing
    hitsInThisPage <- length(resultList[["hits"]][["hits"]])
    if (hitsInThisPage == 0){
        break
    }

    # If we have more to pull, get the new scroll_id
    scroll_id <- resultList[['_scroll_id']]

    # Combine with existing table
    resultDT <- data.table::rbindlist(
        list(resultDT, data.table::as.data.table(resultList))
        , fill = TRUE
    )

    # Increment the count
    hitsPulled <- hitsPulled + hitsInThisPage
}
```

# Help us make it better, please

https://github.com/uptake/uptasticsearch/issues

**Things you may learn**

- docker
- R
  - covr, data.table, httr, jsonlite, roxygen2, testthat
- Python
  - json, pandas, requests
- bash
- make
- Travis CI
- Appveyor
- Elasticsearch



# Thanks for your time!