
Big Data Analytics With JuliaDB

Vitali Marenny

F. Hoffman-La Roche

PhUSE US Connect 2020

Agenda

- Introduce Julia and JuliaDB
- Getting started
- JuliaDB concepts
- Data manipulation
- Data persistence
- Parallel computing
- Concluding remarks
- QA

What is Julia ?

- Julia is an Open Source (MIT Licensed) programming language
- Aims to combine the productivity of Python with the speed of a statically-typed, compiled language like C.
- Julia has its natural usage in the domain of scientific, statistical and high-performance programming.
- Matured to version 1.X as of Sep 2018.

What is JuliaDB ?

- JuliaDB is a package for working with large persistent datasets.
- The high level goals of JuliaDB are to:
 - quickly load multidimensional datasets
 - index the data and perform filter, aggregate and sort operations
 - efficient binary store
 - parallelism to utilize multiple cores or multiple machines in a cluster.
- JuliaDB is implemented entirely in Julia and can utilize Julia's ecosystem of packages such as OnlineStats to achieve out-of-core processing.
- JuliaDB is an analytical database, not traditional relational or no-sql database.

What is JuliaDB ?

- JuliaDB combines aspects of R's Tidyverse and Python's Pandas, to allow for first class data manipulation and transformation functions
- Integration with OnlineStats package, JuliaDB allows for out-of-core statistical analysis for datasets that are too large to fit into RAM all at once.
- The basic building blocks of JuliaDB are the: `IndexedTable` and `NDSparse` structures. The structures store data in typed columns, making operations on entire columns such as aggregations fast (Analytical).
- This means that JuliaDB is not suitable for OLTP (Transactional) type store, i.e. where rows are being added/removed in a fast manner.
- IndexedTable and NDSparse structures are comparable to R's Data Frames (and are in fact interoperable).

Getting Started

- Julia binaries can be downloaded from <https://www.julialang.org/downloads/>.
- Juno, an Atom based plugin, or Jupyter are excellent interfaces to program in Julia.
- JuliaDB is a package that needs to be installed into the Julia depot.
- In the Julia REPL, press the right square bracket “]” and enter Julia’s built-in package manager:

```
add JuliaDB

# Alternatively in Jupyter, or from an open .jl file execute

using Pkg
pkg.add("JuliaDB")
```

Generating Data

- Let's generate a relatively medium sized dataset of biometrics. The dataset will contain 31 Million rows, and two columns date-time and heart rate at one second interval.

```
using JuliaDB, Distributions, Dates

start_dt = Dates.DateTime(Dates.now())
end_dt = Dates.DateTime(Dates.now() + Dates.Year(1))
dates = collect(start_dt:Dates.Second(1):end_dt)
hr = round.(Int64, rand(Normal(90,10),length(dates)))
hr_tracker = table(dates, hr, names = [:date, :hr])
hr_tracker = reindex(hr_tracker, :date)

hr_tracker
```

Table with 31622401 rows, 2 columns:

date	hr
2020-01-17T11:33:55.268	74
2020-01-17T11:33:56.268	92
...	
2021-01-17T11:33:54.268	79
2021-01-17T11:33:55.268	89

Selecting, Sorting, Filtering

- JuliaDB's syntax is very intuitive, selecting column(s) achieved with select function, sorting rows with sort and filtering with filter. Note that sort and filter are part of base Julia, demonstrating interoperability of different packages with the `IndexedTable` and `NDSparse` structures.

```
filter((:hr => x -> x .> 100), hr_tracker) # show heart rates above 100
```

Table with 4647471 rows, 2 columns:

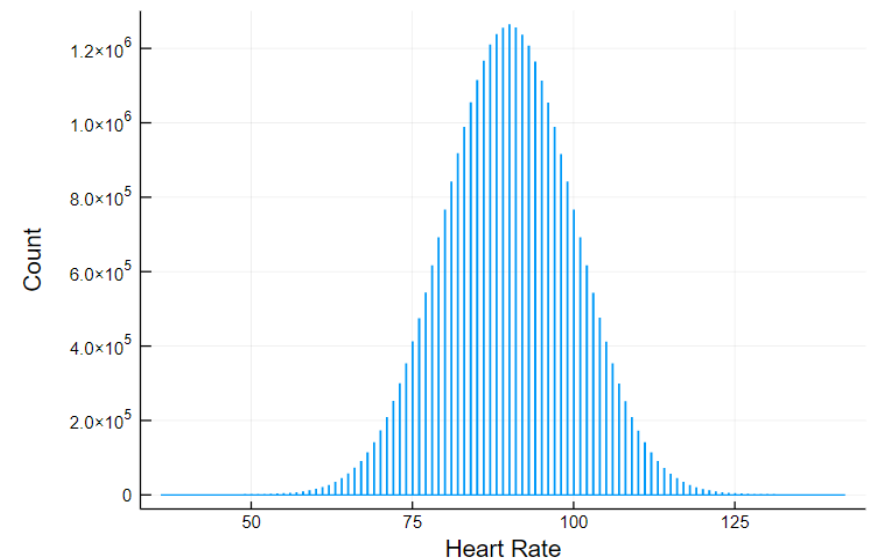
date	hr
2020-01-17T11:34:01.268	109
2020-01-17T11:34:04.268	102
2020-01-17T11:34:19.268	109
...	

```
sort(hr_tracker, :date, rev=true) # sort by Descending Date
```

Table with 31622401 rows, 2 columns:

date	hr
2021-01-17T11:33:55.268	89
2021-01-17T11:33:54.268	79

```
using Plots; gr()
histogram(select(hr_tracker, :hr))
```



File I/O - Saving

- Persisting and loading datasets are an integral part of every data science workflow. JuliaDB has a fast built in delimited file parser (CSV) and an efficient binary storage format.

```
@benchmark writedlm("./test.csv", hr_tracker, ",")
```

```
BenchmarkTools.Trial:
  memory estimate: 25.31 GiB
  allocs estimate: 569308042
  -----
  minimum time:      32.567 s (20.08% GC)
  median time:       32.567 s (20.08% GC)
  mean time:         32.567 s (20.08% GC)
  maximum time:      32.567 s (20.08% GC)
  -----
  samples:           1
  evals/sample:      1
```

```
@benchmark JuliaDB.save(hr_tracker, "./test.jlb")
```

```
BenchmarkTools.Trial:
  memory estimate: 18.08 KiB
  allocs estimate: 296
  -----
  minimum time:      895.218 ms (0.00% GC)
  median time:       922.242 ms (0.00% GC)
  mean time:         950.682 ms (0.00% GC)
  maximum time:      1.124 s (0.00% GC)
  -----
  samples:           6
  evals/sample:      1
```

File I/O - Loading

- Loading the data back to Julia.

```
@benchmark t = loadtable("./test.csv")
```

```
BenchmarkTools.Trial:
 memory estimate:  924.36 MiB
  allocs estimate: 3842
-----
 minimum time:      9.849 s (0.39% GC)
 median time:      9.849 s (0.39% GC)
 mean time:        9.849 s (0.39% GC)
 maximum time:     9.849 s (0.39% GC)
-----
 samples:            1
 evals/sample:      1
```

```
@benchmark t=JuliaDB.load("./test.jlb")
```

```
BenchmarkTools.Trial:
 memory estimate:  14.22 KiB
  allocs estimate: 232
-----
 minimum time:      134.585 μs (0.00% GC)
 median time:      157.792 μs (0.00% GC)
 mean time:        189.004 μs (2.49% GC)
 maximum time:     14.540 ms (58.38% GC)
-----
 samples:           10000
 evals/sample:      1
```

- The Binary file is 41% smaller at 483M vs 820 MB for CSV, and at it takes a max time of 14 ms vs 9849 ms to load.
- There are significant computational and storage advantages in using the binary native JuliaDB format. The x700 speedup in loading time we observe is because JuliaDB memory maps the binary file, which reduces I/O bottlenecks.

Out-of-core computation

- OnlineStats is a package used for statistical computation in an online (one observation at a time) manner.
- Its strengths are in processing streaming and big data that is too large to fit into memory. It integrates with JuliaDB's distributed data structures like IndexTable.

```
using OnlineStats
@time reduce(Mean(), hr_tracker; select = 2)
```

0.154114 seconds (13 allocations: 480 bytes)

Mean: n=31622401 | value=90.0025

Parallel/Distributed Computing

- JuliaDB integrates with Distributed package and can take advantage of both a clustered environment and multicore CPUs.

```
using Distributed
addprocs(4) # local mode, so should not exceed the number of cores in the system
@everywhere using JuliaDB, OnlineStats

@distributed merge for i in 1:4
    df = fit!(Series(Mean()), randn(10000000))
end
```

- addprocs, can be called for both multiprocessing and distributed computations, demonstrating capabilities of multiple dispatch, i.e. local computations that take advantage of multiple cores or distributed computation that can take advantage of clusters of nodes.
- There are many options available and Julia's documentation is comprehensive and helpful.
<https://docs.julialang.org/en/v1/manual/parallel-computing/>

Conclusion

- JuliaDB in combination with OnlineStats and other Julia packages provides a cohesive workflow for data analytics tasks.
- JuliaDB efficiently processes files, computes statistics with tens of millions of observations on modest hardware,
- Julia can be scaled to take advantage of multicore/thread computation with relative ease.
- Julia and JuliaDB can be a valuable addition to the data science toolbox.

Thank you