Paper SD12

Big Data Analytics with JuliaDB

Vitali Marenny, Hoffman La Roche, Mississauga, Canada

ABSTRACT

JuliaDB is an analytical database package, part of the open source Julia language ecosystem, designed for distributed out-of-core processing of data files and statistical processing. Using the Julia programming language and JuliaDB package, data scientists can leverage a single integrated programming language from prototype to production. JuliaDB provides a cohesive workflow for reading, processing and performing data analytics tasks on both small and large datasets. JuliaDB integrates well with other Julia packages such as OnlineStats and Distributed to retain the benefits of an easy to use high-level language with the performance of low-level language.

INTRODUCTION

Julia is an Open Source (MIT Licensed) programming language with an aim 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.

JuliaDB is a package implementing an analytical database that provides distributed table and array data structures with convenient functions to load data from CSV. The high level goals of JuliaDB are to load multidimensional datasets quickly and incrementally, index the data and perform filter, aggregate and sort operations, efficient binary store and use Julia's built in 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.

GETTING STARTED WITH JULIA AND JULIADB

Julia binaries can be downloaded from https://www.julialang.org/downloads/. Julia binaries are available for Linux, FreeBSD, MacOS and Windows. Juno, an Atom based plugin, or Jupyter provide an excellent way 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")
```

Julia's package manager is used to add, remove, query the status, and work with virtual environments. It is worthwhile to spend time reviewing the documentation at: https://docs.julialang.org/en/v1/stdlib/Pkg/index.html

JULIADB BUILDING BLOCKS

JuliaDB combines aspects of R's excellent Tidyverse and Python's Pandas, to allow for first class data manipulation and transformation functions. Through seamless 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). However, 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 very interoperable).

Let's generate a relatively medium sized dataset of biometrics data relating to a heart rate tracker that measures the heart rate of a person each second for a duration of a year. The dataset will contain 31 Million rows, and two columns date-time and heart rate.

```
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
```

To achieve best performance columns used in lookup or joins should be indexed. Indexing can be done during table definition by supplying **pkey** argument or at any point after by using the reindex command as done in the last line in the snippet.

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
```

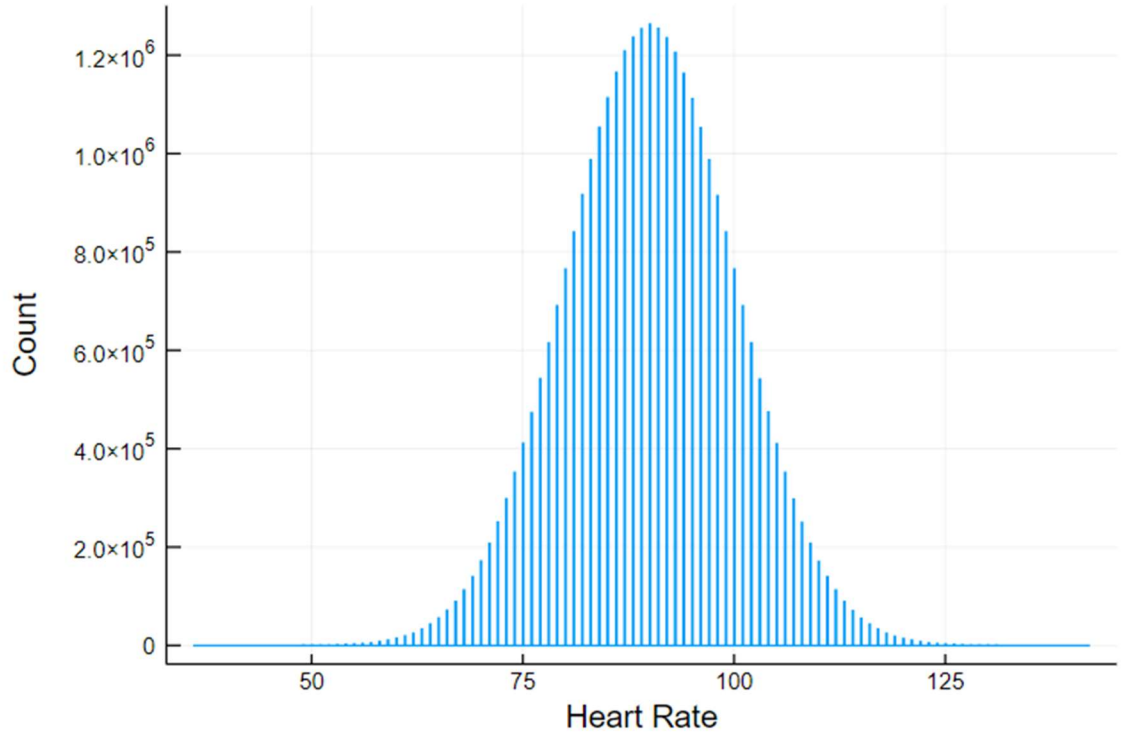
```
2020-01-17T11:34:04.268  102
2020-01-17T11:34:19.268  109
….
```

```
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
```

For the select demo, let's make a plot to show interop of JuliaDB IndexTable with Plots.jl package. Since we sample from normal distribution, and have a large sample size we expect to see normal distribution histogram with the mean around 90.

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



The computation is blazingly fast even on modest hardware, JuliaDB has no issues manipulating 31 Million lines showing the power of its JIT compiler.

FILE LOADING AND 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. Let's persist the hr_tracker dataset to CSV, load it back, then write it in JuliaDB's binary format and load it and see some benchmark comparisons.

```
@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
```

```
@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. We see 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 WITH ONLINESTATS

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. Let's have a look at the reduce operator, that applies a function f (in our case Mean()) to a column in an object (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. Multicore processing is an integral part of Julia and can be achieved with just a few lines of code, namely: importing the Distributed package and invoking the addprocs function with the argument of the total number of workers and using the macro @everywhere for statements that we want to execute on all nodes.

```
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 1:5
    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, and 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.

RECOMMENDED READING

Julia and JuliaDB documentation is comprehensive and worth a close read:
https://docs.julialang.org/en/v1/
https://juliacomputing.github.io/JuliaDB.jl/latest/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:
Vitali Marenny
Hoffmann-La Roche Limited
7070 Mississauga Road
Mississauga, Ontario
L5N 5M8
Email: vitali.marenny@roche.com
Web: https://www.roche.ca