Big Time Series Analysis with JuliaDB

Dr. Josh Day of Julia Computing takes a look into the multi-indexed database of the future.

The next generation of data analysis requires the next generation of tools. The most popular open-source packages for data analysis (Python's pandas and various R packages) are designed to work with small files of basic data types, but 'small' and 'basic' do not describe the data landscape of the future. The amount of data in the world is growing exponentially, and as The Economist observes, it's changing as it grows:

"The quality of data has changed, too. They are no longer mainly stocks of digital information – databases of names and other well-defined personal data, such as age, sex, and income. The new economy is more about analyzing rapid real-time flows of often unstructured data: the streams of photos and videos generated by users of social networks, the reams of information produced by commuters on their way to work, the flood of data from hundreds of sensors in a jet engine."[1]

The current generation tools therefore face a number of difficulties in analyzing the next generation of data. The first is that of scale, which can be achieved with distributed computing systems like Hadoop and Spark, but loses the ease of use that make Python and R tools attractive. Scaling an analysis also adds costs in the form of gluing together tools that may not support the same data types or operations (e.g., Spark DataFrame to Pandas DataFrame to numpy array to scikit-learn model). Another issue for current databases is storing non-standard data types. A database can sometimes work around unsupported types (e.g., units and currencies) by attaching metadata to a field, but the same approach is harder to apply to more complicated data like images and video. The next-generation database should therefore offer the features that are lacking in the current generation:

- **Scalability** (works equally well on Small and Big Data)

- **Ease of use** (no need to glue together different formats)
- **Flexibility** (stores data types that may not exist yet).

Introducing JuliaDB

JuliaDB aims to be the analytics database of the future. It is implemented entirely in Julia, a high-performance language for technical computing designed around modern technologies such as just-in-time compilation, type inference, and parallelism. With Julia at its foundation, JuliaDB allows one to:

1. Store any data type. In JuliaDB, you are not restricted to only a small set of allowed types like software in Python or R. JuliaDB is Julia all the way down, meaning that data types from external packages or user defined are just as fast as built-ins. JuliaDB makes it easy to store and analyze anything, from currencies[2] to images.[3]

2. Compile queries. User-defined functions (UDFs) are often strongly discouraged in other languages as they can take big performance hits. Taking advantage of Julia's just-in-time compilation, UDFs in JuliaDB are fast.

3. Avoid glue. JuliaDB is built on analysis-friendly data structures from IndexedTables.jl. Avoid complicated pipelines and write your analysis in the same language as your database.

4. Work at scale. Julia solves the two-language problem, the idea that prototype code must be rewritten in a faster language for production. JuliaDB works seamlessly with both Small and Big Data, as it can operate in parallel and with data larger than memory.

5. Get results fast. JuliaDB is designed on a 'batteries included' philosophy, allowing quick access to statistical analysis with OnlineStats[4] and machine learning with Flux.[5]

JuliaDB for time series

JuliaDB allows multiple sorted index variables (e.g., time and location) that aid in efficient querying, making it an ideal candidate for time-based data. JuliaDB performs extremely fast operations on indexed variables, and data can be selected, filtered, aggregated, joined, etc., all through Julia's clean syntax and JuliaDB's straightforward API.

Time series packages at a glance

The current platforms for working with time-based data have their limitations. Closed-source options like Kx System's kdb+[6] offers promise of performance at the cost of commercial licenses, whereas open-source packages like Python's pandas[7] and R's xts (extensible time series)[8] offer ease of use at the cost of performance. In addition, none of the standard platforms make it easy to compile UDFs, which adds speed bottlenecks for nonstandard methods of analysis. JuliaDB, as a next-generation database, brings both the promise of performance and ease of use (see Table 1).

Figure 1: Loading 7,163 CSVs as a distributed table

```
addprocs()

using JuliaDB

stocks = loadtable(stockdata; indexcols = [:Symbol, :Date], filenamecol = :Symbol)
```

```
Distributed Table with 14887665 rows in 8 chunks:
Symbol      Date        Open    High    Low     Close   Volume    OpenInt
"a.us.txt"  1999-11-18  30.713  33.754  27.002  29.702  66277506  0
"a.us.txt"  1999-11-19  28.986  29.027  26.872  27.257  16142920  0
"a.us.txt"  1999-11-22  27.886  29.702  27.044  29.702  6970266   0
"a.us.txt"  1999-11-23  28.688  29.446  27.002  27.002  6332082   0
"a.us.txt"  1999-11-24  27.083  28.309  27.002  27.717  5132147   0
"a.us.txt"  1999-11-26  27.594  28.012  27.509  27.807  1832635   0
"a.us.txt"  1999-11-29  27.676  28.65   27.38   28.432  4317826   0
"a.us.txt"  1999-11-30  28.35   28.986  27.634  28.48   4567146   0
"a.us.txt"  1999-12-01  28.48   29.324  28.273  28.986  3133746   0
"a.us.txt"  1999-12-02  29.532  30.375  29.155  29.786  3252997   0
"a.us.txt"  1999-12-03  30.336  30.842  29.909  30.039  3223074   0
"a.us.txt"  1999-12-06  30.547  31.348  30.505  30.883  2385046   0
"a.us.txt"  1999-12-07  30.883  31.052  29.909  30.547  2348161   0
"a.us.txt"  1999-12-08  30.547  30.795  30.249  30.505  2000481   0
"a.us.txt"  1999-12-09  30.547  31.012  30.547  30.924  2150096   0
"a.us.txt"  1999-12-10  30.842  31.012  30.209  30.209  1764043   0
"a.us.txt"  1999-12-13  30.713  31.221  29.958  30.713  4260349   0
"a.us.txt"  1999-12-14  30.635  30.635  28.391  29.027  2467856   0
"a.us.txt"  1999-12-15  28.35   28.561  27.676  28.142  3091820   0
"a.us.txt"  1999-12-16  28.35   31.896  28.35   31.896  2738063   0
"a.us.txt"  1999-12-17  31.308  31.808  30.674  31.012  3929255   0
"a.us.txt"  1999-12-20  31.221  31.687  31.134  31.646  1268225   0
"a.us.txt"  1999-12-21  31.517  31.517  31.052  31.47   2394232   0
"a.us.txt"  1999-12-22  31.47   32.104  31.261  32.104  2019439   0
⋮
```

Figures 2 and 3: Plotting the historical closing price of Amazon (AMZN)

```
using StatPlots

amzn = collect(filter(x -> x.Symbol == "amzn.us.txt", stocks))

@df amzn plot(:Date, :Close, label = "AMZN", ylabel = "Closing Price")
```

Table 1: Comparison of time series platform features

Feature	JuliaDB	kdb+	pandas	xts
Open source	Yes	No	Yes	Yes
Distributed	Yes	Yes	No	No
Multiple indexes	Yes	Yes	Yes	No
Any index type	Yes	Builtins only	Builtins only	Time only
Any value type	Yes	Builtins only	Builtins only	Builtins only
Different column types	Yes	Yes	Yes	No
Compiled UDFs	Yes	No	No	No

Big time series example

A common form of time series is OHLC (open, high, low, close) data. In the following code examples, we use JuliaDB to examine Kaggle's Huge Stock Market Data,[9] which contains OHLC and volume information for 7,163 US-based stocks traded on NYSE, NASDAQ, and NYSE MKT. The dataset is split across many files, as each stock's data is contained in a separate CSV file.

Loading data

On a modern laptop, it takes JuliaDB less than a minute to read in all 7,163 CSVs into a distributed tabular data structure with sorted indexes for ticker symbol and date (see Figure 1). JuliaDB's fast loading times are, in part, due to Julia's built-in parallel computing features. Parallelism is available at the 'flip of a switch,' addprocs(), which allows JuliaDB to distribute datasets over several processors. The results from loading

Figures 4 and 5: Plotting the historical volume of AMZN

```
using OnlineStats
partitionplot(amzn, :Volume, ylab = :Volume, stat = Extrema())
```

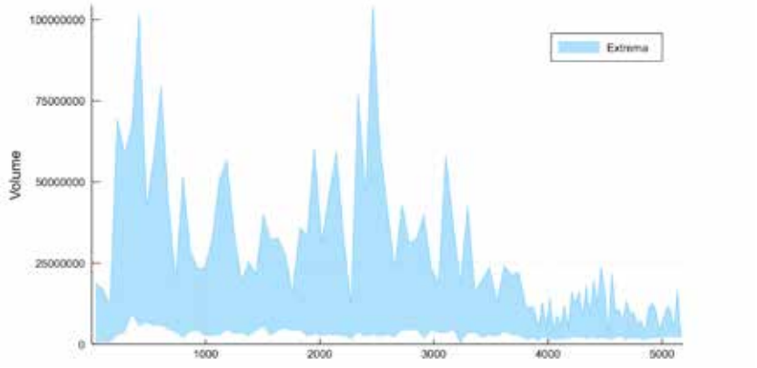


Figure 6: Average volume by stock ticker symbol

```
groupby(mean, stocks, :Symbol, select = :Volume)
```

```
Distributed Table with 7163 rows in 8 chunks:
Symbol          mean
"a.us.txt"      3.9935e6
"aa.us.txt"     3.33776e6
"aaap.us.txt"   1.49989e5
"aaba.us.txt"   2.3218e7
"aac.us.txt"    2.20441e5
"aal.us.txt"    1.00454e7
"aamc.us.txt"   17644.4
"aame.us.txt"   6715.76
"aan.us.txt"    7.13972e5
"aaoi.us.txt"   7.94087e5
"aaon.us.txt"   2.04258e5
"aap.us.txt"    1.22519e6
"aapl.us.txt"   1.06642e8
"aat.us.txt"    2.15145e5
"aau.us.txt"    1.86901e5
"aav.us.txt"    4.28732e5
"aaww.us.txt"   2.64112e5
"aaxn.us.txt"   1.40883e6
"ab.us.txt"     5.60349e5
"abac.us.txt"   11096.8
"abax.us.txt"   2.02161e5
"abb.us.txt"    3.30961e6
"abbv.us.txt"   8.8072e6
"abc.us.txt"    2.97489e6
⋮
```

Figure 7: Average volume by stock ticker symbol

```
ac = reduce(AutoCov(6), amzn, select = :Close)
autocor(ac)
```

```
5-element Array{Float64,1}:
 1.0
 0.998198
 0.996372
 0.994543
 0.99276
```

Figure 8: Time to load taxi data by platform and index column data type

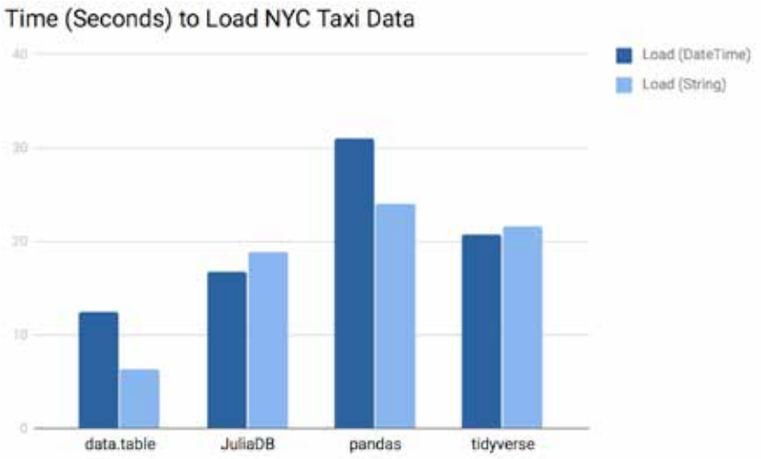
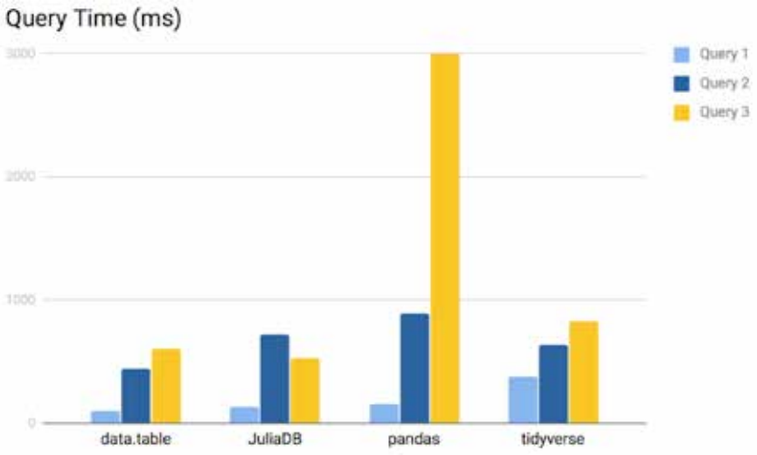


Figure 9: Average time to run example queries by platform



data the first time can also be cached in an efficient binary format that allows reloading in future Julia sessions to be extremely fast. We are also working on adding functionality to import data directly from other formats, such as SQL/ODBC, SAS, and Excel files.

Visualization

Visualizing data is often the first step in any data analysis. In Figure 2 we plot the historical closing prices for AMZN using the StatsPlots.jl[10] package with the GR.jl[11] backend. First, the data subset is retrieved using the filter function, followed by select to return the date and the closing price (see Figure 3).

JuliaDB's integration with OnlineStats makes it possible to plot infinitely sized data (see Figure 4). The partitionplot function is used to create summaries of data partitioned into approximately equal-sized sections. The underlying data structure uses fixed-size memory and is updated incrementally, and can therefore be applied to datasets of any size. In Figure 5, we plot AMZN's historical volume, summarized by its extrema.

Analytics and querying

JuliaDB offers a variety of powerful operations for common data manipulation tasks beyond select and filter, including groupby, map, reduce, groupreduce, join, and groupjoin. In Figure 6 we calculate the mean volume for all 7,163 stocks.

Any statistic in OnlineStats can be used by JuliaDB's reduce and groupreduce functions to compute statistics both online and in parallel. The example in Figure 7 shows how to calculate autocorrelation of a time series.

Performance benchmarks

Some of the top open-source contenders for managing tabular data are Python's pandas and R's data.table[12] or tidyverse[13] (which can then be transformed into xts objects). Here, we take a look at how JuliaDB compares against these three packages, in terms of loading and querying data. In the interest of fairness, we run the benchmarks with the distributed features of JuliaDB turned off.

Benchmark 1: Loading data

Our test for loading data is an 815 MB CSV file from the NYC Taxi & Limousine Commission's records[14] on every yellow cab trip in January 2017. Figure 8 is a comparison of loading the data where the date-time fields are parsed as time objects, or simply as strings.

Benchmark 2: Querying data

Our tests for querying involve three different tasks (see Figure 9):

1. Get the average 'fare amount' by type of vendor. This is a standard 'groupby' operation, calculating a statistic (mean) grouped by the unique values in another column (type of vendor).
2. Get the distribution of 'number of passengers' by 'day of week'. Here, we wish to count the number of trips where the number of passengers is one, two, three, etc., grouped by the day of the week.
3. Get the distribution of 'number of passengers' by whether the weekday number is even (Monday, Wednesday, Friday) or odd (Sunday, Tuesday, Thursday, Saturday). Here, we perform a similar 'groupby' operation to the previous task, but use an arbitrary 'groupby' variable that needs to depend on a user-defined function. JuliaDB is the only platform that does not have a slowdown associated with a UDF, with pandas taking the biggest hit.

The future

This introduction gives only a brief glimpse at JuliaDB's power and flexibility. On Small Data, it is competitive in performance/ease of use compared to its open-source peers, yet scales to Big Data without gluing together additional tools. JuliaDB also integrates with Julia Computing's JuliaRun to seamlessly run in a data center or in the cloud.

Julia Computing is dedicated to making JuliaDB the go-to platform for analytics and machine learning. The unique ability of JuliaDB to store arbitrary data types and compile UDFs makes new types of analysis possible that have yet to be explored, and we are excited for what the future holds.

The example code and benchmarks are at www.juliacomputing.com/wilmott/juladb.

ENDNOTES

1. https://www.economist.com/news/briefing/21721634-how-it-shaping-up-data-giving-rise-new-economy.
2. https://github.com/JuliaFinance/Currencies.jl.
3. https://github.com/joshday/OnlineStats.jl.
4. https://github.com/FluxML/Flux.jl.
5. https://kx.com.
6. https://pandas.pydata.org.
7. https://cran.r-project.org/web/packages/xts/xts.pdf.
8. https://cran.r-project.org/web/packages/xts/xts.pdf.
9. https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs.
10. https://github.com/JuliaPlots/StatPlots.jl.
11. https://github.com/jheinen/GR.jl.
12. https://cran.r-project.org/web/packages/data.table/index.html.
13. https://www.tidyverse.org.
14. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

FURTHER READING

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. 2017. Julia: A fresh approach to numerical computing. SIAM Review 59(1), 65–98.
Wickham, H. 2014. Tidy data. Journal of Statistical Software 59(10), 1–23.