DTable – an early performance assessment of a new distributed table implementation

02 November 2021 | **Krystian Guliński**

In a recent survey conducted within the Julia community the functionality to process tabular data larger than available RAM came out on top. While Julia already has some tools for that they are not very popular within the community and have been mostly left unmaintained (e.g. `JuliaDB`).

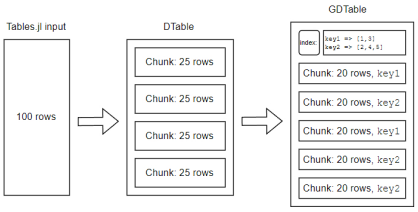
The `DTable` plans to address this popular use case in a composable manner by leveraging the current Julia data ecosystem and our existing distributed computing and memory management capabilities. We hope it's a major step towards a native Julia tool that will handle the out-of-core tabular data processing needs of the Julia community!

What is the `DTable`?

The `DTable` is a table structure providing partitioning of the data and parallelization of operations performed on it in any supported environment. It's built on top of `Dagger.jl`, which enables it to work in any worker and thread setup by taking care of task scheduling and memory management. Any `Tables.jl` compatible source can be ingested by the `DTable` and it can also act as one in case you move the data somewhere else.

A key feature is that the `DTable` doesn't use any dedicated structure for storing the data in memory. Any `Tables.jl` compatible table type can be used for internal storage, which allows for greater composability with the ecosystem. To further support that the set of operations that can be performed on a `DTable` is generic and only relies on interfaces offered by `Tables.jl`.

The diagram below presents a simple visual explaination of how the `DTable` and `GDTable` (grouped `DTable`) are built. Provided input will be partitioned according to either a `chunksize` argument or the existing partitioning (using the `Tables.partitions` interface). After performing a `groupby` operation the data will be shuffled accordingly and new chunks containing only the data belonging to specific keys will be created. Along with an `index` these chunks form a `GDTable`.



Why the DTable?

The `DTable` aims to excel in two areas:

- parallelization of data processing
- out-of-core processing (will be available through future `Dagger.jl` upgrades)

The goal is to become competetive with similiar tools such as `Dask` or `Spark`, so that Julia users can solve and scale their problems within Julia.

By leveraging the composability of the Julia data ecosystem we can reuse a lot of existing functionality in order to achieve the above goals and continue improving the solution in the future instead of just creating another monolithic solution.

Operations available today

Below is a list of functionality generally available today. To post suggestions please comment under this GitHub issue. In the future we hope to provide a roadmap and priority indicators for specific functionality.

- `map`
- `filter`
- `reduce`
- `groupby` (shuffle with full data movement)
- grouped `reduce`
- constructors consuming `Tables.jl` compatible input
- compatibility with `Tables.jl` (`DTable` can be used as a source)

Initial performance comparison (multithreaded)

The benchmarks below present the initial performance assessment of the `DTable` compared to `DataFrames.jl`, which is currently the go-to data processing package in Julia and to `Dask` - the main competitor to the `DTable`. The `DataFrames.jl` benchmarks are here to provide a reference to what the performance in Julia looks like today.

Please note that the benchmarks below were specifically prepared with the focus on comparing the same type of processing activities. That means the benchmark code was accordingly adjusted to make sure the packages are doing exactly the same thing under the hood.

The table below presents the summary of the results obtained in a one machine multithreaded environment (exact setup in the next section). Times from every configuration of each benchmark were compared and summarized in the table. Negative values mean it was slower than the competitor by that percentage.

Operation	avg % faster than Dask	avg % faster than DataFrames.jl
Map	405.7	−50.3
Filter	−11.3	174.9
Reduce (single column)	3012.6	194.2
Reduce (all columns)	2612.3	274.5
Groupby (shuffle)	1700.1	−99.8
Reduce per group (single column)	1921.4	−64.9
Reduce per group (all columns)	2130.0	−14.0

Benchmark configuration

Benchmarks were performed on a desktop with the following specifications:

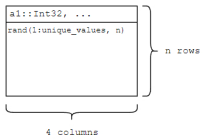
- CPU: Ryzen 5800X 8 cores / 16 threads
- Memory: 32 GB DDR4 RAM
- Julia: master/1.8 (custom branch)

All configurations were ran using an environment with 1 worker and 16 threads.

The data used for experiments was prepared as follows:

- column count: 4 (to allow for a distinction between single and all column benchmarks)
- row count: n
- row value type: `Int32`
- row value range: $1:unique\_values$ (important for `groupby` ops)
- chunksize (`Dask` and `DTable` only): 10^6, 10^7

Diagram below summarizes the above specifications:



Basic operations (`map`, `filter`, `reduce`)

These three operations are the base for the majority of functionality of any table structure. By looking at their performance we can get a good grasp of how the table is doing in the common data transformation scenarios.

These basic operations are unaffected by the count of unique values, so the results of these comparisons are not included here.

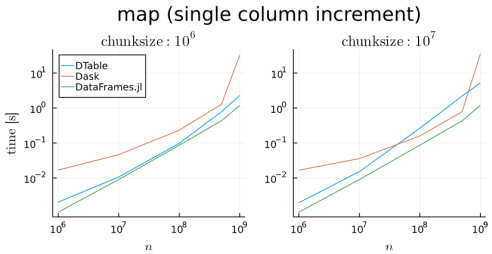
Map (single column increment)

In the first benchmark we're performing a simple `map` operation on the full table.

At first glance it's clear that the overhead coming from the partitioning and parallelization present in the `DTable` and `Dask` is not paying off in this benchmark. The `DataFrames.jl` package is leading here with the `DTable` being on average 50% slower.

At the smaller chunksize (`10^6`) the `DTable` is scaling better than its competitor, which isn't greatly affected by that parameter. Overall the `DTable` managed to offer an average ~4 times speedup compared to `Dask` across all the tested configurations.

DTable command: `map(row -> (r = row.a1 + 1,), d)`



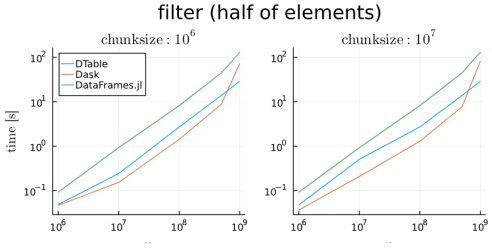
Filter

As the set of values is limited a simple filter expression was chosen, which filters out approximately half of the records (command below).

In this scenario the parallelization and partitioning overhead starts to pay off as both `DTable` and `Dask` are noticably faster than `DataFrames.jl`. When it comes to the comparison of these two the performance looks very similiar with `Dask` being on average 11% faster than the `DTable`.

On top of almost matching the performance of the main competitor the `DTable` offers performance improvements over `DataFrames.jl` by being on average 175% faster.

DTable command: `filter(row -> row.a1 < unique_values ÷ 2, d)`



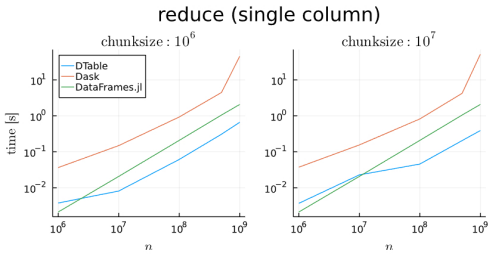
Reduce (single column)

The reduce benchmarks are the place where the `DTable` really shines. This task can easily leverage the partitioning of the data in order to achieve a speed increase.

The `DTable` has not only managed to successfully perform faster than `DataFrames.jl` (on average 194% faster), but it also managed to significantly beat `Dask`'s performance by offering a ~30 times speedup.

Please note that both `DTable` and `DataFrames.jl` are using `OnlineStats` to obtain the variance while `Dask` is using a solution native to it.

DTable command: `reduce(fit!, d, cols=[:a1], init=Variance())`

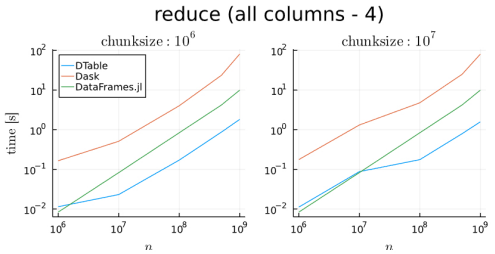


Reduce (all columns)

Similarly to the previous benchmark the `DTable` is performing here very well by offering a ~2.7 times speedup over `DataFrames.jl` and ~26 times speedup over `Dask`.

Additional parallelization can be enabled in the future for wide tables. As of right now the `DTable` is performing the reduction of all columns as a single task.

DTable command: `reduce(fit!, d, init=Variance())`



Grouped operations

A table shuffle is definitely one of the most demanding operations that can be performed on a table, so that's why it was tackled early to evaluate whether the current technology stack makes it feasible to run operations like this.

In the following benchmarks the performance of `groupby` (shuffle) and grouped `reduce` will be put to the test. Other operations like `map` and `filter` are also available for the `GDTable` (grouped `DTable`), but they work in the same way if they were performed on a `DTable`, so previously shown benchmarks have already persented that.

The following benchmarks include results obtained in tests withh varying `unique_values` count, since the number of them directly affects the number of groups generated through the grouping operation.

Please note that the testing scenarios were adjusted specifically to ensure the benchmarks are measuring the same type of activity (data shuffle). Most notably `Dask` benchmarks use `shuffle` explicitly instead of `groupby` to avoid optimized `groupby/reduce` routines, which do not perform data movement. A better comparison can be performed in the future once `DTable` supports these optimized passes as well.

Groupby (shuffle)

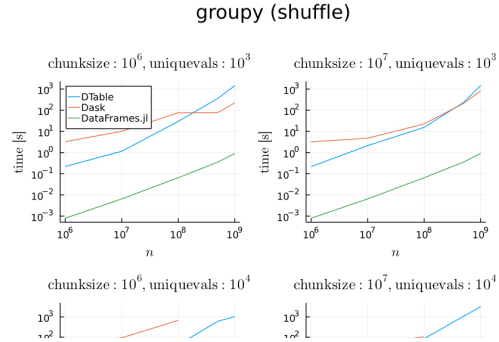
In this experiment we're looking at shuffle performance in various data configurations. `DataFrames.jl` doesn't perform data movement on groupby, so its performance is clearly superior to the other two technologies and is just included for reference purposes.

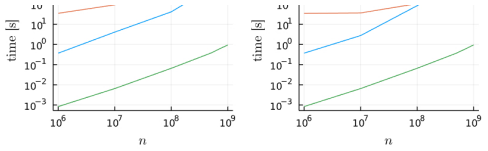
Let's focus on `Dask` and the `DTable`, which are performing the data movement as part of the shuffle. Across the different data configurations we can see a common pattern where the `DTable` is significantly faster than `Dask` at smaller data sizes, which leads to it offering an average ~17 times speedup, but as the data size grows the scaling of `Dask` is better and it eventually matches the speeds of the `DTable`.

However, in the more demanding configurations, in which the `unique_values` count was equal to 10^4, `Dask` was repeatedly failing to finish the shuffle above a certain data size ($n > 10^8$). For that reason the following benchmarks will not include results for these failed tests. Those configurations are also excluded from the average performance comparison.

The `DTable` managed to finish these complex scenarios without any observable hit on scaling, which is a good sign, but future testing needs to be performed on larger data sizes to gain more insight into how well the current shuffle algorithm is performing.

DTable command: `Dagger.groupby(d, :a1)`



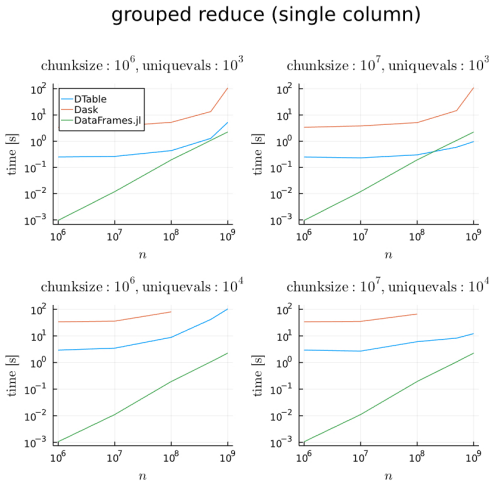


Grouped reduction (single column)

Mimicking the success of reduction benchmarks the `DTable` is performing here better than the direct competition again. For the single column reductions it's an average ~20.2 times speedup over `Dask` and their scaling looks very similiar.

Contrary to the standard reduction benchmarks the `DTable` doesn't offer a speedup compared to `DataFrames.jl` across all the data sizes. It looks like the current algorithm has a significant overhead that can be observed as a lower bound to the performance at smaller data sizes. For the benchmarks with the smaller `unique_values` count the `DTable` manages to catch up to `DataFrames.jl` at bigger data sizes This may indicate that by increasing the data size further we might eventually reach a point where the `DTable` provides a performance improvement over `DataFrames.jl` in this scenario.

DTable command: `r = reduce(fit!, g, cols=[:a2], init=Mean())`

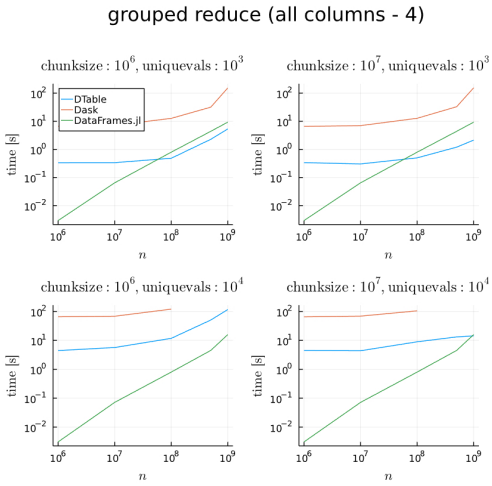


Grouped reduction (all columns)

The results for the all column reduction look very similiar. The `DTable` managed to offer an average ~22.3 times speeup over its main competitor `Dask`.

Again, `DTable` is heavily falling behind on smaller data sizes compared to `DataFrames.jl` due to the significant entry overhead acting as a lower performance bound at smaller data sizes.

DTable command: `r = reduce(fit!, g, init=Mean())`



Implementation details

The `DTable` is built on top of `Dagger` and `Tables.jl` and currently resides within the `Dagger.jl` package. That means it can run in any environment `Dagger` is capable of running in. You should be able to use the `DTable` effectively on your local machine in a threaded environment, on a bigger machine with many workers and threads or have the workflow spread around multiple machines and workers in your cluster.

The `DTable` uses the new `EagerAPI` in `Dagger`, which means all the parallelized calls are done using `Dagger.spawn`. Memory is managed by `Dagger` through the usage of `MemPool.jl`. Upgrades to the related projects in the future will hopefully yield performance and functionality improvements for the `DTable`.

Because of the dependencies of the `DTable` on other projects its focus is completely on delivering `Tables.jl` compatible algorithms and interfaces to address the growing needs for processing big tabular data.

We hope that the `Tables.jl` interface will grow to include an even wider range of functionality while still providing great intercompatibility with other Julia packages.

For more details please visit the Dagger documentation page.

How can I use it?

The `DTable` has successfully passed the proof of concept stage and is currently under active development as a part of the `Dagger.jl` package.

Functionality presented as part of this blogpost is generally available as of today. We highly encourage everyone to have a look at the documentation and to try out the examples included! Due to the fact that the `DTable` is still in early development it's very much possible to provide feedback and affect the future design decisions.

However, there are some pending PRs that haven't been merged into Julia yet that improve the thread safety of `Distributed`, which directly affects `Dagger.jl` stability. User experience may be interrupted when extensively using the `DTable` in a threaded or mixed environment by occasional hangs or crashes.

We hope to include all the necessary fixes in the upcoming Julia 1.7 release.