Journal of Statistical Software

September 2023, Volume 107, Issue 4. *doi: 10.18637/jss.v107.i04*

DataFrames.jl: Flexible and Fast Tabular Data in Julia

Milan Bouchet-Valat
French Institute for Demographic Studies

Bogumił Kamiński
SGH Warsaw School of Economics

Abstract

DataFrames.jl is a package written for and in the Julia language offering flexible and efficient handling of tabular data sets in memory. Thanks to Julia's unique strengths, it provides an appealing set of features: Rich support for standard data processing tasks and excellent flexibility and efficiency for more advanced and non-standard operations. We present the fundamental design of the package and how it compares with implementations of data frames in other languages, its main features, performance, and possible extensions. We conclude with a practical illustration of typical data processing operations.

Keywords: data frame, tabular data, performance, multi-threading, Julia.

1. Introduction

DataFrames.jl is a package written for and in the Julia language (Bezanson, Edelman, Karpinski, and Shah 2017) offering flexible and efficient handling of tabular data sets in memory. Thanks to Julia's unique strengths, we believe that this new implementation of the classic concept of *data frame* provides an appealing set of features, with rich support for standard data processing tasks, as well as an excellent level of flexibility and efficiency for more advanced and non-standard operations. Among **DataFrames.jl**'s design goals are the ability to write high-level Julia code with C- or Fortran-like speed for any user-defined transformation task, consistent design of basic and high-level operations, strong integration with the Julia ecosystem, efficient and multi-threaded grouped operations, first-class support for custom column types, zero-copy data exchange with other languages via the **Apache Arrow** format, ability to store table- and column-level metadata, and more.

Julia allows developers to write efficient and multi-threaded code to process data without any extra packages. It is relatively common to use a named tuple of vectors as a simple, built-in type to store and process tabular data. Therefore a natural question is: What is the role of **DataFrames.jl** in the Julia ecosystem? The answer is that the main goal is to provide

flexibility and convenience to users while not sacrificing execution speed. This contrasts with the functions that serve e.g., the **data.table** (Dowle and Srinivasan 2023b) package in R or the **Polars** (Vink 2023) bindings in Python, where one *needs* an external package written in a low-level language to benefit from fast and multi-threaded execution of operations on data frames.

This article is structured as follows. In Section 2 we discuss general principles behind the development of the **DataFrames.jl** package, and in Section 3 we present the key design elements of the `DataFrame` object. In Sections 4 and 5 we respectively discuss the low-level imperative API based on indexing and the declarative API using high-level functions. Then Section 6 summarizes other key functionalities provided by the **DataFrames.jl** package. In Section 7 we present how the package integrates with the wider ecosystem of data management packages developed for the Julia language. In Section 8 we discuss the performance of the package. We conclude by giving references to materials available for users wanting to learn how to use **DataFrames.jl** and by discussing limitations and planned developments of the package.

The code presented in this article was run under Julia 1.9.0 and **DataFrames.jl** 1.5.0. The example code from the paper is provided in a companion `README.md` file and, for reader convenience, additional files containing source code for benchmarks of selected tabular data processing packages. Along with these files the `Project.toml` and `Manifest.toml` files are provided to ensure that Julia installs the exact package versions with which the code was run. To learn how to properly set up and run the examples see the instructions provided in Julia's package manager documentation `https://pkgdocs.julialang.org/v1/getting-started/`.

2. About DataFrames.jl

The concept of *data frame* has been well-established for several decades in statistical programming environments. Among its best-known implementations are: (a) R's `data.frame` type, (b) the Python package **pandas** (McKinney 2010; **pandas** Development Team 2023), (c) the R package **tibble** (Müller and Wickham 2023)—companion to **dplyr** (Wickham, François, Henry, and Müller 2023), and (d) the R package **data.table** (Dowle and Srinivasan 2023b).

Data frames are essential building blocks of a majority of data analysis pipelines, so it is not surprising that **DataFrames.jl** is one of the oldest and most popular packages of the Julia ecosystem (Claster and Shah 2021, slide 14). Even though it has been developed and widely used since 2012 (six years before Julia reached version 1.0), **DataFrames.jl** 1.0 was only released in 2021 when it was judged to have reached a sufficient level of maturity, consistency, and integration with Julia design principles.

Data frames are one of the simplest ways of storing tabular data. Just like a spreadsheet, data frames cross observations (rows) with variables (columns). Data frames differ from relational databases in the fact that the order of rows and columns is significant. They are in some ways closer to matrices, with the essential difference that columns have names and can store values of different types efficiently (for example, strings, numbers, or custom user-defined types). Performance is achieved by storing each column as a separate vector. This *column-oriented* storage makes them different from most (but not all) database implementations and particularly suited for the kind of operations involved in statistical work. However, existing data frame packages differ in the extent to which this storage format is exposed to users. In Section 3 we develop the choices made in **DataFrames.jl**.

While the design of **DataFrames.jl** benefited a lot from the experience of various data frame implementations, it is most similar to **dplyr**. Contrary to **pandas** and R's `data.frame`, **DataFrames.jl** does not support a special column holding row names: All columns are treated in the same fashion. This fits in the "tidy data" paradigm (Wickham 2014) and allows for simpler and more general interfaces. A similar choice is made in e.g., **dplyr** or **data.table**, where using row names is discouraged. We know this may disappoint some users, but we invite them to read the below presentations of alternative strategies, which can often prove as convenient and efficient.

Probably the most notable strong point of **DataFrames.jl**, which makes it an original data frame implementation, is due to its thorough and carefully thought application of the Julia programming language design principles. Julia is a high-level, high-performance dynamic programming language for technical computing that compiles to efficient machine code at run time. It combines the ease of use of other dynamic languages such as R and Python with the speed of C, C++ or Fortran. This sets **DataFrames.jl** apart from "classic" data frame implementations since, contrary to them, it is written entirely in the same language as the one its users work in (an approach which has recently been adopted by **Polars** in Rust).

Julia's speed makes it unnecessary to implement the most performance-sensitive parts in a lower-level language. While this may seem an implementation detail relevant only to developers, it also has major implications for end users. Indeed, this means that any custom function written in high-level Julia code can be compiled to machine code and be as efficient as built-in functions. Users do not need to learn a separate lower-level language nor deal with the increased complexity of making two languages interact. This blurs the line between users and developers, making it easier to develop non-standard operations when appropriate. It also frees **DataFrames.jl** from the requirement to implement every particular specialized operation to make it fast: Providing a generic function is enough since it can be combined efficiently with user-defined operations. Therefore, **DataFrames.jl** does not provide implementations of many operations that are typically applied to data frame columns, like plotting, statistical functions, or displaying a data frame in a terminal. All these functionalities are provided by external generic libraries that rely on the abstract table interface **DataFrames.jl** supports.

Furthermore, since it is entirely written in Julia, **DataFrames.jl** does not require columns to fit in one of the few built-in types that the implementation supports. On the contrary, a custom vector type or vector element type implementation provided by another package or even written by the user is treated in the same way as standard ones. Such a new type can be as efficient as the standard arrays and built-in types, which are supported by default. This is in contrast with other high-performance data frame implementations, such as the recently developed **Polars** package, which rely on a specific memory representation of stored columns, often requiring users to convert the columns to other vector types before they can be used with other Python packages. We illustrate in Section 7 how powerful this flexibility is by combining **DataFrames.jl** with other packages e.g., supporting data with a small number of unique values, categorical data, or fast memory-mapped read-only columns. Observe that this flexibility allows users to store objects of any type in a data frame without losing the type information. In particular, this allows a column of a data frame to nest composite types, like arrays or struct types that are available in modern databases like **BigQuery** or **Snowflake**.

3. Fundamental design

In **DataFrames.jl**, data frame objects are of the `DataFrame` type. The internal structure of this type is very simple. It essentially holds a collection of column vectors (which can be any object whose type inherits from `AbstractVector`) and a mapping from column names to these vectors. Columns can be referred to either using a string (typed e.g., `"x"` in Julia), a symbol (typed `:x` or `Symbol("x")`), or just by their position passed as an integer number (starting at 1).

Before we discuss the `DataFrame` object design, let us explain one important difference between Julia and R or Python. Julia is a compiled language, while R and Python are interpreted. The benefit of compilation is that the execution time of Julia code is fast. This, however, comes with a price that for each new data type passed to some function Julia needs to compile it before execution. This impacts the time of the first run of this function (as it includes both compilation and execution time).[1] This cost can be avoided if we *hide* data type from a function. Then the function is executed less efficiently, but one does not have to wait for it to compile. Therefore one of the design patterns that is used in Julia is that in inexpensive functions the data type is hidden (to avoid their recompilation). In contrast, computationally intensive functions are passed information about the type of data they work on (to make sure the code runs as fast as possible since in this case compilation time is expected to be negligible in comparison to execution time).

`DataFrame` objects do not have a fixed schema like tables in traditional databases, which typically have a constant set of column names and defined data types of these columns. Columns can be added, removed, or replaced after constructing a `DataFrame` object. In Julia parlance, this classic behavior of data frames implies that `DataFrame` objects are *type-unstable*. That is, the concrete type of column vectors cannot be determined from the `DataFrame` object at compile time. This avoids over-specialization of the generated code, allowing for efficient storage and processing of thousands of heterogeneous columns without paying a high compilation cost. However, the downside is that this prevents the Julia compiler from generating efficient code unless column vectors are passed to a function that is specialized on their type. This flexibility of `DataFrame` object schema seems to contradict the goal of ensuring the performance of operations performed on them. However, the package uses lazy code specialization to ensure high performance when working with **DataFrames.jl**. We present these solutions in Section 8. The basic idea is that most of the code that operates on `DataFrame` does not have to be maximally fast, as it is cheap anyway: Only the performance-critical parts need and are specialized to be *type-stable* so that the compiler can generate efficient machine code for them.

Several constructors are supported to create `DataFrame` objects. Here let us mention only a few basic ones. `DataFrame()` creates a zero-column, zero-row data frame which can be filled later. `DataFrame(x1 = v1, x2 = v2)` creates a data frame with two columns named `"x1"` and `"x2"` holding copies of vectors `v1` and `v2` (if `v1` and/or `v2` are scalars they are repeated an appropriate number of times). `DataFrame(col => v)` is another syntax that allows passing the column name as a variable (e.g., when `col = "x"`), which is useful when writing a generic code without hard-coded column names. A general constructor also accepts

[1] Julia supports *precompiling* code so that the compilation cost is paid only once when installing a package. Unfortunately, as we explain in the next paragraph, it is impossible to precompile code for all potential combinations of all column types for any number of columns.

any object implementing the **Tables.jl** (Quinn and JuliaData contributors 2023b) interface (see Section 7): This allows constructing data frames from a multiplicity of sources like dictionaries or formats supported by other packages, such as CSV, Arrow or JSON files.

An important design principle of **DataFrames.jl** is that a `DataFrame` object is considered to *take ownership* of its columns. This means that operations implemented in the **DataFrames.jl** package assume that the columns stored in a `DataFrame` are never resized and that their elements are never reordered by an external call. In order to ensure this behavior `DataFrame` constructors by default copy the columns passed to them. This approach is taken because of end-user safety considerations, to avoid unexpected bugs in the code which would corrupt the `DataFrame`. However, in some situations this copying behavior is undesirable. The first case is when the user explicitly wants to avoid copying of the columns for performance or to save memory. For this case **DataFrames.jl** functions consistently accept the `copycols = false` keyword argument that disables copying. The second case is when an external package has already allocated fresh columns, in which case it signals to the `DataFrame` constructor that copying is not needed, e.g., when reading a CSV file with the **CSV.jl** (Quinn and JuliaData contributors 2023a) package.

Despite their internal structure, and contrary to some other implementations like `data.frame` or **tibble** in R, `DataFrame` objects are not exposed to users as collections of columns. Depending on the context, they behave either as two-dimensional structures (like matrices) or as collections of rows (similar to the definition of tables as sets of tuples in relational algebra). This double nature is one of the defining traits of data frames (Petersohn *et al.* 2020), and it proved to be the most useful in practice.

`DataFrame` objects are treated as *two-dimensional objects* in contexts related to indexing, where their behavior is consistent with matrices. This is developed in Section 4. Data frames also behave like two-dimensional objects with the `broadcast` function (a Julia equivalent of vectorization in languages like R or Python), which applies a function to each cell. Finally, the `size` function returns a tuple with the number of rows and of columns, but the functions `nrow` and `ncol` are also provided for convenience.

Additionally, `DataFrame` objects implement all relevant functions defined in Julia Base that operate on collections. For such functions, data frames are treated as *collections of rows*.[2] For example, `sort` orders rows according to specified columns, `repeat` repeats rows, `unique` returns unique occurrences of rows, `first` and `last` return row(s) at the top or at the bottom (respectively), `empty` returns a zero-row data frame with the same column names and types as its argument, `deleteat!` removes specified rows, `push!` adds one or more rows at the bottom, `append!` adds rows contained in a data frame or another table-like object at the bottom.

Operations on columns are supported via distinct functions which are **DataFrames.jl** specific. In particular, `names` returns the column names, `rename!` changes column names, `insertcols!` adds new columns, and `mapcols!` applies a function to each of the column vectors. Higher-level data transformations that operate column-wise by default are described in Section 5.

Before we move to a detailed description of functionalities let us mention that **DataFrames.jl** provides three types which are views of `DataFrame` objects: `SubDataFrame`, `DataFrameRow`,

[2] A notable exception is `length`, which is not implemented to avoid confusing users coming from other implementations where this function returns the number of *columns* and because it is redundant with `nrow`.

and `GroupedDataFrame`. Such views can be constructed very fast and without copying the source data frame, which is especially important when working with large data. Modifications of the contents of a view are propagated back to its parent `DataFrame`, and vice-versa.

In the following two sections we will describe two paradigms of working with data frame objects that **DataFrames.jl** supports. In Section 4 we cover the low-level imperative style based on indexing. In Section 5 we discuss the declarative style using high-level functions. Both paradigms are fully supported in **DataFrames.jl** and users can freely choose between them as needed.

4. Indexing

A classic way of selecting rows or columns in data frames in many programming ecosystems is to rely on indexing, i.e., treating them as two-dimensional structures. Row at position `row` in a `DataFrame` object `df` can be accessed using the `df[row, :]` syntax, and column `col` can be accessed using the `df[:, col]` syntax. This is consistent with the syntax used for matrices in Julia, where `:` indicates that all elements should be retained on the corresponding dimension (rows or columns).

`df[row, :]` requires `row` to be an integer index (as noted above, **DataFrames.jl** does not support row names). It returns a `DataFrameRow` object which is a one-dimensional collection similar to a named tuple, pointing to values in the corresponding row in `df`. `DataFrameRow` objects are views into their parent data frame `df`, they reflect changes that may be applied to it *after* extracting the row.

`df[:, col]` requires `col` to be either a string or symbol giving the column name, or an integer giving its position. As opposed to row indexing,[3] `df[:, col]` returns a *copy* of the column vector. The alternative syntax `df[!, col]` is supported to avoid copying.[4] For convenience, when the name of the column is a literal (rather than stored in a variable like `col`), the syntax `df.x` or `df."x"` can be used to access column `"x"` without copying (equivalent to `df[!, :x]` or `df[!, "x"]`). If a column is extracted without copying, any modification of its contents will be reflected in the data frame.

The contents of a single cell can be accessed using `df[row, col]`, where both indices are single-element selectors.

Indexing can also be used to select multiple rows or columns, by passing a collection of indices or a pattern selector. In this case a new `DataFrame` object is returned. **DataFrames.jl** supports a wide range of selectors, some of which can be combined together (as in `df[rows, cols]`). Here are some selected examples:

- A vector of 1-based indices, like `df[[1, 2], :]` to select the first two rows, or `df[:, [1, 2]]` to select the first two columns, or `df[:, ["x", "y"]]` to select columns `"x"` and `"y"`.
- A Boolean vector with one entry for each row/column set to `true` to retain it, or to

[3] This different behavior was chosen due to performance considerations, as making a copy of a row on indexing would be very slow for data frames with many columns, which would make this operation almost unusable in practice.

[4] The symbol `!` was chosen for similarity with the Julia convention according to which it is used as a suffix to functions that mutate their arguments.

false to exclude it, e.g., df[[true, true, false], :] to select the first two rows of a three-row data frame.

- A regular expression to select columns whose names match, e.g., df[:, r"x"] to select columns containing "x" in their name.
- A Not object indicating which rows/columns should be excluded, e.g., df[Not([1, 2]), :] to take all rows but the first two or df[:, Not(["x", "y"])] to select all columns but "x" and "y".
- A Between object to select columns positioned between two columns (including them), e.g., df[:, Between("x", "y")].
- A Cols object with several selectors to select columns matching at least one selector, e.g., df[:, Cols(1, r"x")] to select the first column and columns containing "x" in their name.

It is worth noting that these multi-element selectors always return a DataFrame object, even if they happen to select a single row/column.[5] This is essential to allow both the user and the compiler to predict the type of the result, leading to faster, more robust, and clearer code.

Consistent with what happens when selecting a single column, all indexing syntaxes which use : return copies of the input data. df[!, cols] can be used to get a new DataFrame holding the original columns without a copy.

Another way to index without copying is to use view(df, rows, cols) or @view df[rows, cols] to create a SubDataFrame object instead of a DataFrame. SubDataFrame objects store a reference to their parent data frame, as well as indices of the rows and/or columns that were selected. They reflect changes made to their parent *after they have been created*, and modifying their contents also affects their parent in return. One should therefore be careful and not reorder or delete rows or columns in the parent data frame while a SubDataFrame pointing to it exists.

The indexing syntaxes presented above can also be used to modify the contents of a data frame. The simplest form is to add a new column or update the contents of an existing one using df[:, col] = v, where col is a single-column selector and v is a vector. Note that this modifies the contents of the column vector if it already exists, and makes a copy of v if it does not. If source and destination vectors have different types, this will imply a conversion, and may even fail for incompatible types (for example string and integer). The df[!, col] = v syntax can be used to *replace* the destination column vector with v instead, without copying it.

It is also possible to update the contents of an existing column only for some rows using df[rows, col] = v. In this case, v must have a length equal to the number of selected rows. One can also update multiple columns at the same time via df[rows, cols] = m, where m can be either a data frame with the same column names as the selected part of df or a matrix with the same number of columns. Finally, a single cell can be updated using df[row, col] = v, where v can be any value that can be converted to the destination type.

[5]This is contrary to R where e.g., the scalar value 1 is represented as the single-element vector c(1) ([1] in the Julia syntax), making it impossible to distinguish between a single- and a multiple-element selector.

A common pattern when working with data frames is to use Boolean indexing to select rows based on conditions on the values of one or more columns. This can be achieved via e.g., `df[df.x .== 1 .&& df.y .> 0, :]`. The dots before operators are required to indicate to Julia that operation must be applied to each element (i.e., row) in the vectors. This syntax can also be used to set the contents of a column based on conditions. In particular, broadcast assignment using operator `.=` allows repeating a single value to assign it to all selected rows, as in `df[df.x .== 1 .&& df.y .> 0, :]  .= "Group 1"`.

Data frame objects can also be used in broadcasting operations, just like matrices. For instance, provided that we store in `df` only numeric columns the code `df .= log.(ifelse.(df .< 0, NaN, df))` is a way to efficiently calculate the `log` of all entries in a data frame, while storing `NaN` for negative values (as applying `log` to a negative value in Julia raises an error). Omitting the `df .=` part returns a newly allocated data frame rather than operating in-place. The same expression can be written more conveniently as `@. df = log(ifelse(df < 0, NaN, df))` (the `@.` macro could have also been used in other examples above involving broadcasting of multiple operations).

Above we have covered only selected major use-cases of indexing. An important property of this functionality in **DataFrames.jl** is that it was carefully designed to consistently support a wide variety of ways of selecting rows and columns of a data frame, covering the numerous combinations of the following alternatives:

1. Extracting part of a data frame
   - (a) Column selection
     - i. Extracting a single column as a vector.
     - ii. Getting a collection of zero or more columns as a data frame.
   - (b) Row selection
     - i. Copying: Allocate new columns.
     - ii. Non-copying: Reuse columns of a source data frame as-is or as a view.
2. Assignment to a data frame
   - (a) Target column update policy (left-hand side of an assignment)
     - i. In-place update of an existing column.
     - ii. Replace column with a copy of source.
     - iii. Replace column without copying the source.
   - (b) Source treatment policy (right-hand side of an assignment)
     - i. Take the object as-is.
     - ii. Broadcast the object into a target data frame.

Such a wide variety of options follows the scenarios that users require when working with real data. The reason all of them are provided is because indexing is considered to be a low-level API of the **DataFrames.jl** package so it must ensure high flexibility.

Let us stress that basic indexing scenarios always make a copy of the underlying data. The reason is that in this way it is ensured that no column aliases (i.e., which point to the same memory) are created, as they can lead to hard-to-catch bugs. However, functions that do

not allocate new copies of the data are provided and can be used for cases where memory is constrained or high performance is required.

In summary, both data frame construction (as described in Section 3) and indexing follow the same policy: *Provide safe operation by default and ensure it is possible to perform an efficient operation if requested.* This approach has proven to be easy to use for users starting to work with **DataFrames.jl**, while providing the flexibility required by experts.

5. Data transformations and grouping

Besides the low level API based on indexing `DataFrame` objects, **DataFrames.jl** provides powerful high-level functions to transform data that allow its users to write their queries in a declarative way. In this area three major functions are provided:

1. `combine` selects columns, optionally transforming them, and returns a data frame with as many rows as are present in the returned columns (typically *combining* several rows of source into one or more rows in the target).
2. `select` selects columns, optionally transforming them, and returns a data frame with as many rows as its input and in the same order.
3. `transform` selects all existing columns and adds new columns by transforming existing ones, and returns a data frame with as many rows as its input and in the same order.

By default the `select` and `transform` functions follow the safety-first policy of **DataFrames.jl** described above and copy all columns of the source data frame. For cases when performance is required the in-place variants `select!` and `transform!` are provided. In particular `transform!` can be much faster and use much less memory than `transform` for large data frames, as the latter makes a copy of all columns in the input data frame by default (though this can be also overridden by appropriately using the `copycols` keyword argument in the same way as described in Section 3 for constructor).

These functions take as arguments a `DataFrame`, a `SubDataFrame`, or a `GroupedDataFrame` (discussed below) followed by a list of column selectors and/or transformations. Column selectors can be of any form used for indexing. Transformations are specified using a special syntax `source => function => target` formed of three parts separated by the `=>` operator:

1. `source`: Selectors indicating source column(s).
2. `function`: A function to apply to the corresponding column vectors.
3. `target`: Name(s) to give to output column(s).

A simple example of the `source => function => target` syntax is `:a => sum => :a_sum`. Where column `:a` is passed to the `sum` function and the result is stored in column `:a_sum`. Note that in this case the `sum` function produces a scalar, which is stored in a single row of the resulting data frame by `combine`, but is broadcast to match the number of rows by `select` and `transform`. Here is an example (in the code examples lines prefixed with `julia>` are Julia code and the lines that follow them are program output):[6]

[6]Copy-pasting the whole block of text in the Julia REPL in Linux will automatically execute the relevant parts and skip others.

```
julia> using DataFrames
julia> df = DataFrame(a = 1:3)

3×1 DataFrame
 Row │ a
     │ Int64
─────┼───────
   1 │     1
   2 │     2
   3 │     3

julia> combine(df, :a => sum => :a_sum)

1×1 DataFrame
 Row │ a_sum
     │ Int64
─────┼───────
   1 │     6

julia> select(df, :a => sum => :a_sum)

3×1 DataFrame
 Row │ a_sum
     │ Int64
─────┼───────
   1 │     6
   2 │     6
   3 │     6
```

Another common pattern is to apply a function to each element in a column. This can be achieved conveniently by wrapping the function in the special `ByRow` object, as in `:a => ByRow(cos) => :a_cos`, which creates a new column `:a_cos` containing the cosine of each element in column `:a`. This kind of syntax is generally not required in R or Python where many functions are vectorized, i.e., they accept vectors as arguments and apply the operation to each of their elements. Julia is stricter and more consistent in that regard, and requires indicating explicitly when vectorization is desired via the broadcasting syntax mentioned above, for which `ByRow` is a shorthand.

Names of output columns are optional and if omitted will either be generated automatically from the function name or extracted from the returned object. For example, `combine(df, :a => sum)` is equivalent to the `combine(df, :a => sum => :a_sum)` example shown above.

One of the important benefits of `source => function => target`, apart from being visually easy to follow, is that it ensures that `function` can be any transformation defined by the user and that the execution of such a query will be fast. The engine that processes such a request ensures that it is compiled to an efficient machine code before its execution.

Additionally, and importantly, `source => function => target` is valid Julia code, so such transformations are easily constructed in a programmatic way. Here is a minimal example that uses broadcasting to apply multiple functions to each of the two selected columns:

```
julia> df = DataFrame(x1 = 1:3, x2 = 4:6)

3×2 DataFrame
 Row │ x1     x2
     │ Int64  Int64
─────┼──────────────
   1 │     1      4
   2 │     2      5
   3 │     3      6

julia> combine(df, [:x1, :x2] .=> [sum minimum maximum])

1×6 DataFrame
 Row │ x1_sum  x2_sum  x1_minimum  x2_minimum  x1_maximum  x2_maximum
     │ Int64   Int64   Int64       Int64       Int64       Int64
─────┼────────────────────────────────────────────────────────────────
   1 │      6      15           1           4           3           6
```

This syntax works because the `.=>` broadcasting operation returns a matrix of pairs listing the expected transformations:

```
julia> [:x1, :x2] .=> [sum minimum maximum]

2×3 Matrix{Pair{Symbol}}:
 :x1=>sum  :x1=>minimum  :x1=>maximum
 :x2=>sum  :x2=>minimum  :x2=>maximum
```

The feature that any transformation specification is valid Julia code also allowed for the development of domain-specific languages that make it possible to specify transformations of data frames more conveniently. This approach is described in Section 6.

The strength of `combine`, `select` and `transform` is most visible when applied to grouped data, via the split-apply-combine strategy (Wickham 2011). The "split" part, i.e., grouping rows according to the values in one or more columns, can be performed via the `groupby` function. This function returns a `GroupedDataFrame` object, which is a view of the source data frame where grouping columns are treated as keys. The most basic way to use a `GroupedDataFrame` is to index into it. Here is an example:

```
julia> df = DataFrame(key1 = ["a", "b", "a", "b"],
                      key2 = [1, 2, 1, 2],
                      value = 1:4)

4×3 DataFrame
 Row │ key1    key2   value
     │ String  Int64  Int64
─────┼──────────────────────
   1 │ a           1      1
   2 │ b           2      2
   3 │ a           1      3
   4 │ b           2      4
```

```
julia> gdf = groupby(df, [:key1, :key2])

GroupedDataFrame with 2 groups based on keys: key1, key2
First Group (2 rows): key1 = "a", key2 = 1
 Row │ key1    key2   value
     │ String  Int64  Int64
─────┼──────────────────────
   1 │ a           1      1
   2 │ a           1      3
⋮
Last Group (2 rows): key1 = "b", key2 = 2
 Row │ key1    key2   value
     │ String  Int64  Int64
─────┼──────────────────────
   1 │ b           2      2
   2 │ b           2      4

julia> gdf[("b", 2)]

2×3 SubDataFrame
 Row │ key1    key2   value
     │ String  Int64  Int64
─────┼──────────────────────
   1 │ b           2      2
   2 │ b           2      4
```

We have shown just one option of indexing into a `GroupedDataFrame`, where one uses a set of values of key columns to pick an appropriate subset of rows as a `SubDataFrame`. It is important to highlight that this operation is guaranteed to be fast as it is non copying (a view is returned) and lookup is performed using a hash table. This functionality is an efficient equivalent of hierarchical indexes provided e.g., by **pandas** (partial indexing with `MultiIndex`). In particular, one can create several `GroupedDataFrame` objects, grouped by different sets of columns, backed by the same data frame without copying the underlying data.

While this kind of manual work with a `GroupedDataFrame` as a collection of groups is often useful, it is somewhat verbose and not fully optimized for speed. Instead, using `combine`, `select`, and `transform` functions ensures that grouped operations are both convenient and efficient. Data transformations can be applied to a `GroupedDataFrame` using exactly the same syntax as for a `DataFrame`, and return an ungrouped `DataFrame` (unless `ungroup = false` is specified, in which case the resulting object is a `GroupedDataFrame`). The only difference is that transformation functions will be applied separately for each group.[7] When transformations return a vector, `combine` returns as many rows as there are values in the vector, and `select` and `transform` require the number of values to match the number of rows in the corresponding group. When transformations return a single scalar value (like

[7] This design ensures an invariant that `DataFrame` is treated exactly the same as a single-group `GroupedDataFrame` with no grouping columns.

sum), combine returns one row per group, and select and transform repeat this value to fill all rows belonging to the corresponding group. Using gdf from the previous example we get:

```
julia> combine(gdf, :value => sum)

2×3 DataFrame
 Row │ key1    key2   value_sum
     │ String  Int64  Int64
─────┼──────────────────────────
   1 │ a           1          4
   2 │ b           2          6

julia> select(gdf, :value => sum)

4×3 DataFrame
 Row │ key1    key2   value_sum
     │ String  Int64  Int64
─────┼──────────────────────────
   1 │ a           1          4
   2 │ b           2          6
   3 │ a           1          4
   4 │ b           2          6
```

Note, in particular, that select and transform ensure an invariant that the order of rows of the parent data frame is respected in the result. Clearly, this invariant does not apply to combine as it changes the number of rows by combining them.

6. Other functionalities

Apart from data frame construction, indexing, and transformation, **DataFrames.jl** provides a range of functions covering all classic operations on data frames (the [!] suffix indicates that the function has two variants: Creating a new object and in-place):

1. Joining: Either creating new data frames (innerjoin, leftjoin, rightjoin, outerjoin, semijoin, antijoin, and crossjoin) or updating a data frame (leftjoin!).
2. Reshaping between long and wide data formats (stack and unstack), transposition (permutedims), and flattening (flatten).
3. Iterating over rows and columns (eachrow and eachcol).
4. Subsetting rows (filter[!], subset[!], deleteat!, empty[!]).
5. Sorting rows (sort[!], issorted, sortperm).
6. Finding or dropping duplicate rows (unique[!], nonunique).
7. Handling missing values (dropmissing[!], disallowmissing[!], allowmissing[!], completecases).

8. Concatenating tables and adding rows (`hcat`, `vcat`, `append!`, `push!`, `repeat[!]`).

9. Manipulating columns (`rename[!]`, `insertcols!`, `mapcols[!]`) and summarizing them (`describe`).

DataFrames.jl fully supports statistical missing values, that are represented using the `missing` object (of type `Missing`) defined by Julia itself (Bouchet-Valat 2018), which is similar to `NA` in R and `NULL` in SQL. In line with the design principles of the package, missing values can therefore be handled with standard Julia functions. Columns which allow for missing values have an element type `Union{T, Missing}` (i.e., they contain either a value of type `T` or a value of type `Missing`), where `T` can be any type and are handled efficiently thanks to compiler optimizations for small `Union` types. For code safety reasons, missing values are never silently skipped: Standard mathematical operators propagate it (returning `missing` if any of the inputs is `missing`), and most functions throw an error if passed `missing`. Missing values can be skipped using the `skipmissing` function, replaced using the `coalesce` function, or propagated using the `passmissing` function.

Another feature of **DataFrames.jl** that is especially useful when working with wide tables is its support of metadata on table and column levels. Table-level metadata are key-value pairs that are attached to a data frame. Column-level metadata are key-value pairs that are attached to a specific column of a data frame. An example application of metadata is to keep descriptive labels of columns stored in a data frame. It is possible to dynamically add and remove metadata from a given data frame. A particularly useful feature is that such metadata can be easily loaded from and stored in files if the data file format used supports it. One of the popular formats giving this possibility is **Apache Parquet**.

Finally, let us mention that while all of the **DataFrames.jl** functionality can be used by writing standard Julia code, for user convenience there have been developed companion packages that provide a domain-specific language (DSL) for specifying transformations of data frames. The three most popular of such packages are the **DataFramesMeta.jl** (Deffebach, Short, and JuliaData contributors 2023), **DataFrameMacros.jl** (Krumbiegel 2023), and **Tidier.jl** (Singh and TidierOrg contributors 2023). All provide a DSL with a similar design to the **dplyr**/**tidyverse** syntax in R. **Tidier.jl** provides a very close equivalent of this syntax, while **DataFramesMeta.jl** and **DataFrameMacros.jl** consist in macros which are (for most of them) more terse equivalents of functions described above: `@select`, `@transform`, `@combine`, `@groupby`, `@subset`, etc.

Here is an example comparison of syntaxes of **DataFrames.jl** and **DataFramesMeta.jl**:

```
transform(df, :a => sum => :a2) # standard Julia code with DataFrames.jl
@transform(df, :a2 = sum(:a)) # DataFramesMeta.jl DSL
```

Using one of these DSLs makes the code even more readable, at the expense that it is not standard Julia code anymore (macros perform non-standard evaluation of the code). An important consideration is that using such DSLs has no impact on the performance of the operations as these packages essentially rewrite DSL statements into standard **DataFrames.jl** queries. The key observation here is that **DataFrames.jl** provides a flexible and fast low-level functionality that can be easily used in a custom DSL taking advantage of Julia's metaprogramming capabilities. In this way end-users can use syntax that is easy for them to learn.

For example **Tidier.jl** was designed in a way that a user who knows **dplyr** can immediately start writing data processing pipelines in Julia.

7. Extensions

Features listed above are tightly linked with the `DataFrame` type and most of them allow modifying such objects in place. However, many data analysis functionalities do not require such tight integration. Here we would like to emphasize another design principle of **DataFrames.jl**. Thanks to the high composability of programs written in the Julia language, the **DataFrames.jl** package does not have to provide many functionalities that are normally bundled into data frame packages in programming languages like R or Python. Instead it is assumed that external packages should provide features that are not directly related to objects provided by **DataFrames.jl**. Two key packages enable the integration with external packages:

- **Tables.jl** (Quinn and JuliaData contributors 2023b) defines an implementation-agnostic interface for objects that should be considered as tables. **DataFrames.jl** implements this interface, which allows creating `DataFrame` objects from sources defined by other packages and passing `DataFrame`s to functions that accept **Tables.jl** objects.
- **DataAPI.jl** (Quinn, Kamiński, and JuliaData contributors 2023a) provides a namespace for data-related generic function definitions to ensure a common API is used across packages without requiring them to depend on each other.

Let us list a few packages that, using these interfaces, interoperate with **DataFrames.jl** without the creation of direct dependencies:

1. Loading from and saving to files in different formats: **CSV.jl** (Quinn and JuliaData contributors 2023a), **Arrow.jl** (Quinn, Savastio, and Apache contributors 2023b), **Avro.jl** (Quinn and JuliaData contributors 2021a), **JSONTables.jl** (Quinn and JuliaData contributors 2021b), **Parquet2.jl** (Savastio 2023), **XLSX.jl** (Noronha 2023), **ReadStatTables.jl** (Chen 2023), **StatFiles.jl** (Anthoff 2019).
2. Plotting: **StatsPlots.jl** (Vertechi, Borregaard, and JuliaPlots contributors 2023).
3. Printing: **PrettyTables.jl** (Chagas 2023).
4. Modelling and machine learning: **GLM.jl** (Bates, Noack, Bouchet-Valat, Kornblith, and JuliaStats contributors 2023) and **MLJ.jl** (Blaom, Kiraly, Lienart, Simillides, Arenas, and Vollmer 2020).

Multiple packages are also often used with **DataFrames.jl**, interoperating with it thanks to interfaces defined by Julia itself. This includes most notably the **Statistics** standard library module, as well as **StatsBase.jl** (Lin, Bouchet-Valat, Noack, Arslan, and JuliaStats contributors 2023), and **Plots.jl** (Breloff, Schwabeneder, Christ, and JuliaPlots contributors 2023).

During the years of development of the Julia ecosystem we have learned that the interface-based design of **DataFrames.jl** has the following benefits:

1. The number of dependencies of packages is lower, which significantly simplifies their maintenance.
2. The code base is more compact, as only one method that relies on the interfaces is needed, which is used for all table types.
3. Adding new packages (or types or functions) to the ecosystem does not require changes in the existing packages as long as new packages conform to the common interface.

It is important to highlight that relying on the interfaces does not lead to performance degradation, as the Julia compiler eventually specializes user code to an efficient machine code.

Some extensions of **DataFrames.jl** deserve to be presented here as they are essential features for a data frame implementation, even though they are implemented in separate packages. They illustrate the flexibility that Julia offers to external packages, which can add features to **DataFrames.jl** without having to merge them into a large monolithic code base.

The **PooledArrays.jl** (Bezanson, Bouchet-Valat, Kamiński, and JuliaData contributors 2023) package allows storing columns in a memory-efficient way in which the proportion of unique values is relatively low. The `PooledArray` type provided by this package behaves exactly like a plain `Array`, but stores values internally as an array of integer reference codes pointing to a pool of all values that appear in the column. Depending on the number of values, codes can be represented using 8, 16, 32 or 64 bits to optimize memory use. `PooledArrays` are particularly efficient to store large types like strings or custom structures.

Besides using less memory, `PooledArrays` are more efficient with `groupby` as the number of unique values is known in advance and computations can be performed directly on integer codes. It can therefore be faster to convert a data frame column to a `PooledArray` before performing repeated `groupby` operations using it as (one of) the grouping key(s). `PooledArray` columns can be created manually, but they are notably created automatically when reading CSV files using **CSV.jl**.[8]

The **CategoricalArrays.jl** (Bouchet-Valat and JuliaData contributors 2023) package is intended for columns which contain categorical data in a statistical sense, either nominal or ordinal. While `CategoricalArrays` use an efficient storage similar to `PooledArrays`, they are semantically different as they carry a set of *levels*, which have a user-specified order and are preserved even if the value is not actually present in the data. Indexing `CategoricalArrays` returns `CategoricalValue` objects which include a reference to these levels and can be compared using operators such as `<` (for ordered arrays). The ability to choose a custom ordering of levels is useful notably when drawing plots when printing summary tables and to define contrasts when fitting statistical models.

The **InlineStrings.jl** (Quinn and JuliaStrings contributors 2023) package allows of an efficient storing of columns with short strings or more generally strings with a fixed length. Instead of storing arrays of strings as references to objects stored elsewhere in memory like the default Julia `String` type, the package's custom string types can be stored inline, in a single contiguous block of memory. Inline strings are allocated on the stack rather than on the heap, avoiding pressure on the garbage collector which would otherwise slow down operations. Inline strings are used by default when reading CSV files via **CSV.jl**.[9]

[8]This can be tweaked using the `pool` keyword argument to the `CSV.read` function.

[9]This can be tweaked using the `stringtype` argument to `CSV.read`. Note in particular, that if a column mixes long and short strings then the standard `String` type might be preferred, as inline strings always take up the space required to fit the longest string in the column.

Finally, the **Arrow.jl** (Quinn *et al.* 2023b) package is an interesting extension not only because it allows reading from and writing to the popular **Apache Arrow** format, but also as a more general illustration of what custom array types can achieve in combination with **DataFrames.jl**. When reading a file, **Arrow.jl** returns a custom `Arrow.Table` object which can easily be converted to a `DataFrame` object thanks to the **Tables.jl** interface described in Section 6. Columns of the resulting `DataFrame` use **Arrow.jl**-specific array types providing read-only views of the backing **Arrow** file, which is made accessible via memory mapping (`mmap`). This means that no copy of the data is made in RAM, leaving the operating system the choice to load the contents of the file to the memory cache or to drop them as appropriate. This zero-copy feature is also valuable to exchange data between Julia and another process. It is worth noting that these features do not require any special handling in **DataFrames.jl**, which simply treats **Arrow.jl** array types just like any other column type. Performing a copying operation (such as `transform`) on the data frame will return a "classic" `DataFrame` in a sense that its columns would use standard Julia types that can be mutated.

8. Performance considerations

While **DataFrames.jl** can be very fast thanks to the Julia compiler generating efficient machine code, some care should be taken to avoid performance traps. The main such trap is due to the type instability of `DataFrame` columns (see section 3). Performance-sensitive code sections should avoid looping over rows of a data frame or over entries of a column vector that has been extracted from a data frame *without first introducing a function barrier*. In what follows we assume that we have some data frame `df` with columns `:x` and `:w`.

For example, the following naive summation will be slow as the type of `df.x` is not known to the compiler:

```
s = 0.0
for xi in df.x
    s += xi
end
```

On the contrary, moving the loop to a separate function which will be specialized on the column vector type will be fast:

```
function mysum(x::AbstractVector)
    s = 0.0
    for xi in x
        s += xi
    end
    return s
end

mysum(df.x)
```

This rule also applies to iteration over rows, either using `eachrow` or using indexing. The following naive implementation of weighted sum is slow (`for row in eachrow(df)` is equivalent to `for i in 1:nrow(df); row = df[i, :]`):

```
s = 0.0
for row in eachrow(df)
    s += row.x * row.w
end
```

Moving the loop to a separate function makes it much more efficient:

```
function mysum(x::AbstractVector, w::AbstractVector)
    s = 0.0
    for (xi, wi) in zip(x, w)
        s += xi * wi
    end
    return s
end

mysum(df.x, df.y)
```

The general rule is to write functions taking vectors as arguments rather than `DataFrame` objects directly. **DataFrames.jl** functions like `combine`, `select`, or `transform` do this internally to ensure the compiler generates specialized code in critical parts. Therefore the user has to think about performance when using the low-level imperative API, while the high-level declarative API handles such issues automatically.

Another important rule to follow to obtain the best performance is to use in-place variants of functions (ending with the `!` suffix) whenever possible to avoid unnecessary memory allocations. It is quite obvious that e.g., `sort!(df)` avoids large allocations compared to `sort(df)`, but this also applies to less expected cases. In particular, a seemingly innocuous operation such as `transform(df, :x => (-) => :minusx)` (which creates a new column containing the opposite of column `:x`) will also have to copy all columns in `df`.[10] On the contrary, `transform!(df, :x => (-) => :minusx)` will only allocate the new column; another option would be to pass the `copycols = false` keyword argument to `transform`. The same applies to `select(df, :x, :y, :z)`, which copies the three selected columns unless `copycols = false` is passed or `select!(df, :x, :y, :z)` is used instead. As explained above, the choice of making copies by default ensures basic uses are always safe, while still allowing efficient operation when necessary.

Grouped operations have been carefully designed to be fast, especially when custom user functions are provided. This is an area where Julia's strengths are apparent. Indeed, while other data frames implementations provide built-in optimizations for common operations written in low-level languages such as C++, more complex or unusual operations written by users in R or Python do not benefit from these optimizations. On the contrary, in **DataFrames.jl**, custom user functions written in Julia are compiled to efficient machine code, with a very low per-group overhead. Like other implementations, optimized methods are provided for common reductions (like `sum`, `mean`, `maximum`, `first`, etc.). Finally, multithreading is used if Julia was configured to use several threads.[11] Thanks to these different strategies, **DataFrames.jl** often ranks among the fastest implementations available.

[10] This behavior differs from R, which uses copy-on-write. While convenient, copy-on-write is not a viable solution for fast languages like Julia, as it requires a relatively expensive check before each write access to a vector.

[11] This can be achieved using the `-t` command line argument when starting `julia` executable, or by setting the `JULIA_NUM_THREADS` environment variable.

Similarly **DataFrames.jl** is shipped with join algorithms that provide a competitive performance in comparison to alternative packages. In particular the implemented algorithms are able to take advantage of the fact that columns on which tables are joined are sorted or have a known number of unique values (e.g., are categorical or pooled, see Section 7).

Providing comprehensive performance benchmarks is a very challenging task and the results depend on multiple factors (available hardware, data characteristics, etc.). Therefore, in this paper we only present a brief comparison against the R packages **data.table** and **arrow** (the latter using **dplyr** to specify queries), and Python packages **Polars** and **pandas** (the latter using the **pyarrow** engine), just to show that **DataFrames.jl** achieves competitive performance for standard operations. We acknowledge that in other benchmarks performance comparisons can give different results. As already noted, the key performance advantage of **DataFrames.jl** is seen with user-defined functions or when non-standard data types are used. However, a fair comparison heavily depends on the particular operation and on its implementation: Therefore we stick to standard functions. For the comparison we have picked benchmarks that were referenced by **data.table** maintainers.[12] The comparison was run using:

- Julia 1.9.0 and **DataFrames.jl** 1.5.0;
- R 4.2.2, **data.table** 1.14.8 and **arrow** 12.0.0 (using **dplyr** to specify queries);
- Python 3.11.3, **Polars** 0.17.13 and **pandas** 2.0.1 (using the **pyarrow** engine).

To simulate a typical usage scenario, the tests were run under Windows 11 on a laptop with 16 GB RAM and an Intel i7-1250U CPU and used the default settings of packages. The chosen hardware has 10 cores, out of which 2 are performance cores and 8 are efficient cores. We picked this setup because it is a typical machine that a developer could use for routine in-memory data processing. After presenting these benchmarks we discuss the server-oriented benchmarks that are maintained at https://duckdblabs.github.io/db-benchmark/.

The benchmarks consist of split-apply-combine and join tests.

First we perform a simple split-apply-combine operation: Compute the grouped sum and count over 50 million rows randomly assigned to one of 500,000 groups. The column to sum (`x`) contains random double-precision floats, and the grouping column (`grp`) contains random integers uniformly distributed between 1 and 500,000.

The join benchmarks apply joins on two tables with 50 million rows each and an integer key column (`x`) ranging between 1 and 50 million, with all values appearing once except one, which is absent from each table at random. Additionally, each table has a column containing random double-precision floats (`y1` and `y2`, respectively). We test inner, left, right, and outer join, except for **Polars**, for which it is not supported.

In Table 1, we see that **DataFrames.jl** offers competitive performance against **data.table** and **Polars** and is significantly faster than **pandas** and **arrow**/**dplyr**. Following (Chen and Revels 2016) we report a minimum run time over 100 executions of a given query (such estimator is most robust concerning such noise as flushing cache lines, task switching to background OS processes, etc.). The code that can be used to reproduce these results is presented in Appendix A.

[12] See Gorecki (2015) and StackOverflow questions referenced there for aggregation (http://stackoverflow.com/a/34167477/2490497) and joins (http://stackoverflow.com/a/34219998/2490497) respectively.

	Aggregation	Inner join	Left join	Right join	Outer join
DataFrames.jl (Julia)	0.22	4.17	5.51	5.01	5.76
data.table (R)	1.02	6.10	7.50	7.50	18.31
arrow/dplyr (R)	1.85	11.90	11.17	13.53	12.78
Polars (Python)	0.69	4.41	4.47	—	8.60
Pandas (Python)	2.09	17.15	17.59	21.12	18.30

Table 1: Summary of performance comparison for different operations between **DataFrames.jl** and selected alternative packages in R and Python. All times are reported in seconds and are a minimum over 100 runs. The result for right join in **Polars** is missing because this package does not provide this functionality directly.

As we have mentioned benchmarking code is a challenging process. We have run the above benchmark also for other hardware settings (with different numbers of available CPUs and operating systems) and the results differed. However, in all of them **DataFrames.jl** performance was competitive. It should be noted though that as the number of available cores increases the relative performance of **Polars** and **arrow/dplyr** improves.

Users interested in additional benchmarks run on a large 40-core machine with 157 GB of RAM can check the `https://duckdblabs.github.io/db-benchmark/` website, which was made available in early 2023. These benchmarks differ from the one presented in our paper as they are executed on a server in a cloud and include larger data volumes. In these benchmarks, in general, **DataFrames.jl** is faster than **pandas** and **dplyr** and generally slower than **Polars**, while relative performance compared with **data.table** and **Arrow** varies depending on operations. The currently biggest areas of required performance improvements indicated by these benchmarks are: (1) caching more standard operations in package precompilation (to make the first run of queries faster), (2) improving the efficiency of grouping operations when there are many small groups (the original design of **DataFrames.jl** concentrated on algorithms that are efficient when there are few large groups), and (3) making multi-threading support more comprehensive. These areas are currently under development. They will only require optimizing code and will not affect the API design.

Summarizing the performance discussion, we would like to conclude that **DataFrames.jl** is competitive in terms of performance compared to alternative data frame ecosystems. However, it should be mentioned that there are still areas that are open for improvement. This is one of the topics that the development of **DataFrames.jl** will concentrate on in the future.

9. Practical illustration

To finish this presentation, let us show a brief example of working with the **DataFrames.jl** package using the "NYC-flights14" dataset from the **data.table** vignette (Dowle and Srinivasan 2023a). It lists flights that departed from New York City airports between January and October 2014. Examples are picked to concentrate on the features that are specific to **DataFrames.jl** and **DataFramesMeta.jl**.

First install and load the the required packages,[13] fetch the data from the Internet and read

[13] Julia will automatically prompt to install packages if they are missing when trying to load them via the `using` command.

it into a DataFrame:

```
julia> using CSV, DataFrames, DataFramesMeta, Chain,
             Dates, HTTP, Plots, Statistics
julia> input = "https://raw.githubusercontent.com/Rdatatable\
                /data.table/master/vignettes/flights14.csv"
julia> flights = CSV.read(HTTP.get(input).body, DataFrame)
```

As the data frame is wide let us start by showing how one can retain only selected columns in-place (i.e., updating the source DataFrame). We use the select! function to achieve this:

```
julia> select!(flights, :year, :month, :origin, :dest, :dep_delay)

253316×5 DataFrame
    Row │ year   month  origin   dest     dep_delay
        │ Int64  Int64  String3  String3  Int64
────────┼───────────────────────────────────────────
      1 │  2014      1  JFK      LAX             14
      2 │  2014      1  JFK      LAX             -3
      3 │  2014      1  JFK      LAX              2
    ... │   ...    ...  ...      ...            ...
 253314 │  2014     10  LGA      RDU             -8
 253315 │  2014     10  LGA      DTW             -4
 253316 │  2014     10  LGA      SDF             -5
                                 253310 rows omitted
```

Let us now apply a multi-step data processing pipeline to this data. Our example involves the following steps:

1. Computing an average departure delay by month.
2. Adding a month name column to the data frame.
3. Printing the resulting data frame sorted by average departure delay (as a forked computation in the pipe).
4. Plotting the relationship between month name and average departure delay and saving it to a PDF file (Figure 1).

We use the @chain macro from the **Chain.jl** (Krumbiegel 2022) package[14] to pass the result of each function as the first argument to the next one (piping).

We show how to perform this operation using two approaches. The first approach uses only **DataFrames.jl** transformation functions:

```
julia> @chain flights begin
           groupby(:month, sort = true)
           combine(:dep_delay => mean)
```

[14] **Chain.jl** is automatically loaded when the **DataFramesMeta.jl** package is used.

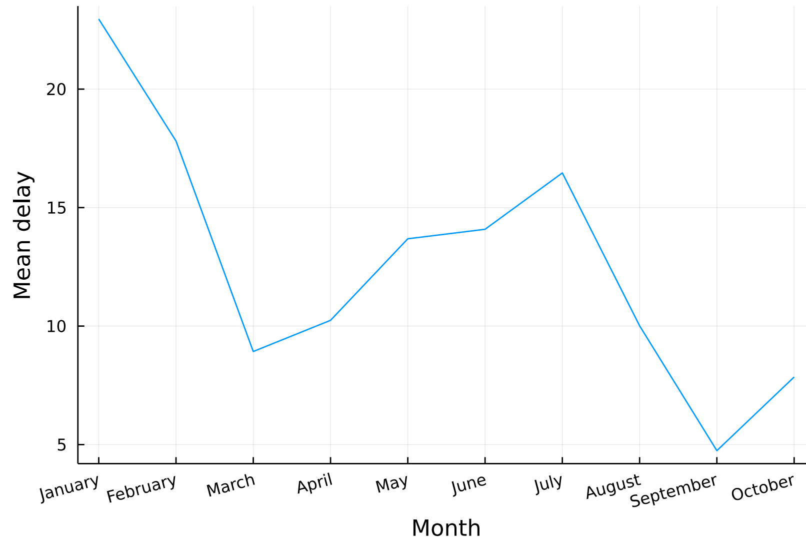

Figure 1: Relationship between a month and average departure delay.

```
    transform(:month => ByRow(monthname) => :month_name)
    @aside show(sort(_, :dep_delay_mean))
    plot(_.month_name, _.dep_delay_mean, label = nothing,
         xlabel = "Month", ylabel = "Mean delay", xrotation = 15)
    savefig("flights.pdf")
end
```

```
10×3 DataFrame
 Row │ month  dep_delay_mean  month_name
     │ Int64  Float64         String
─────┼───────────────────────────────────
   1 │     9         4.74279  September
   2 │    10         7.85055  October
   3 │     3         8.92726  March
   4 │     8        10.0125   August
   5 │     4        10.2431   April
   6 │     5        13.6842   May
   7 │     6        14.0849   June
   8 │     7        16.4631   July
   9 │     2        17.8099   February
  10 │     1        22.9576   January
```

The second approach takes advantage of the **DataFramesMeta.jl** domain-specific language (`@combine` and `@rtransform`) which allows for a terser syntax (we omit showing the output as it is the same as above):

```
@chain flights begin
    groupby(:month, sort = true)
```

```
    @combine(:dep_delay_mean = mean(:dep_delay))
    @rtransform(:month_name = monthname(:month))
    @aside show(sort(_, :dep_delay_mean))
    plot(_.month_name, _.dep_delay_mean, label = nothing,
         xlabel = "Month", ylabel = "Mean delay", xrotation = 15)
    savefig("flights.pdf")
end
```

Let us highlight the following distinctive features of the presented examples:

1. When chaining several operations via `@chain`, by default the value of the previous operation gets passed to the next operation as its first argument, which then can be omitted (exactly like in `%>%` in R). However, one can instead specify the place where this value should be inserted using the `_` (underscore) symbol—we use this feature in the last two commands in the pipe.

2. One can use the `@aside` annotation to fork the pipe; in our example the forked operation was used to `show` the sorted data frame in the terminal; performing this printing has not affected the value passed to the `plot` function as its argument.

3. All operations are wrapped in the `begin-end` block which makes it very easy to add and remove steps in the pipeline interactively (i.e., there is no special piping operator like `%>%` that has to be added or removed at the end of the line when composing the sequence of chained operations).

10. Discussion

As this article shows, **DataFrames.jl** is a feature-complete and flexible implementation of the data frame concept in Julia. It allows for efficient processing of in-memory data sets with single or multiple CPU cores. Moreover, it is important to highlight that the efficiency of **DataFrames.jl** is retained when user-defined data types are stored in the data frame and when user-defined functions are used to transform the data.

No data analysis ecosystem is complete without appropriate reference information and teaching materials. Documentation for **DataFrames.jl** is maintained at https://dataframes.juliadata.org/stable/. A list of curated teaching materials (tutorials, vignettes, cheat-sheet) is included on that page. Storopoli, Huijzer, and Alonso (2021) and Kamiński (2022) are books that can be used as starting points to learn **DataFrames.jl**.

To conclude, we can note two current limitations of **DataFrames.jl** and the perspectives for future improvements.

The first limitation is that while **DataFrames.jl** benefits from Julia's incredible performance, it is also affected by its latency in interactive use. The first time a given operation is run in a Julia session, functions need to be compiled. This usually takes less than a second but can be noticeable. This means that for small data sets, the experience of new users will be that **DataFrames.jl** is less reactive than other data frame implementations. This higher fixed time cost is of course offset by Julia's performance when running similar operations multiple times or for operations that take more than a few seconds to run. Julia developers are well aware

of compilation latency issues, and each new Julia release brings major improvements in this regard. Therefore we do not consider first-run compilation time as a serious and lasting issue for **DataFrames.jl**.

A second limitation is that **DataFrames.jl** is designed to work with in-memory data sets. Other Julia packages have to be used to work with big data in a distributed fashion similar to **Dask** or **Apache Spark**. The `DTable` type has recently been developed to this end in the **Dagger.jl** (Guliński 2021) package. `DTable` implements the **Tables.jl** interface and therefore interoperates perfectly with **DataFrames.jl**, so that e.g., a `DTable` distributed over several machines can use a `DataFrame` object to represent tables on each worker process if desired. Further work is planned to ensure that `DTable` can efficiently take advantage of the functionality already provided by the **DataFrames.jl** package.

Acknowledgments

We would like to thank current and past contributors to **DataFrames.jl** (around one hundred in total, see `https://github.com/JuliaData/DataFrames.jl/graphs/contributors` for a summary), in particular John Myles White and Tom Short who initially developed the package, as well as the authors of packages mentioned in the article.

References

Anthoff D (2019). ***StatFiles.jl**: **FileIO.jl** Integration for Stata, SPSS, and SAS Files.* Julia package version 0.8.0, URL `https://github.com/queryverse/StatFiles.jl`.

Bates D, Noack A, Bouchet-Valat M, Kornblith S, JuliaStats contributors (2023). ***GLM.jl**: Generalized Linear Models in Julia.* Julia package version 1.8.2, URL `https://github.com/JuliaStats/GLM.jl`.

Bezanson J, Bouchet-Valat M, Kamiński B, JuliaData contributors (2023). ***PooledArrays.jl**: A Pooled Representation for Arrays with Few Unique Elements.* Julia package version 1.4.2, URL `https://github.com/JuliaData/PooledArrays.jl`.

Bezanson J, Edelman A, Karpinski S, Shah VB (2017). "Julia: A Fresh Approach to Numerical Computing." *SIAM Review*, **59**(1), 65–98. `doi:10.1137/141000671`.

Blaom AD, Kiraly F, Lienart T, Simillides Y, Arenas D, Vollmer SJ (2020). "MLJ: A Julia Package for Composable Machine Learning." *Journal of Open Source Software*, **5**(55), 2704. `doi:10.21105/joss.02704`.

Bouchet-Valat M (2018). "First-Class Statistical Missing Values Support in Julia 0.7." *The Julia Blog*. URL `https://julialang.org/blog/2018/06/missing/`.

Bouchet-Valat M, JuliaData contributors (2023). ***CategoricalArrays.jl**: Arrays for Working with Categorical Data (Both Nominal and Ordinal).* Julia package version 0.10.8, URL `https://github.com/JuliaData/CategoricalArrays.jl`.

Breloff T, Schwabeneder D, Christ S, JuliaPlots contributors (2023). ***Plots.jl**: Powerful Convenience for Julia Visualizations and Data Analysis.* Julia package version 1.38.12, URL https://github.com/JuliaPlots/Plots.jl.

Chagas RAJ (2023). ***PrettyTables.jl**: Print Data in Formatted Tables.* Julia package version 2.2.4, URL https://github.com/ronisbr/PrettyTables.jl.

Chen J (2023). ***ReadStatTables.jl**: Read and Write Stata, SPSS, and SAS Sata Files with Julia Tables.* Julia package version 0.2.4, URL https://github.com/junyuan-chen/ReadStatTables.jl.

Chen J, Revels J (2016). "Robust Benchmarking in Noisy Environments." *arXiv e-Prints*, arXiv:1608.04295. doi:https://doi.org/10.48550/arXiv.1608.04295.

Claster A, Shah V (2021). "Julia User and Developer Survey 2021." Presentation at JuliaCon, URL https://julialang.org/assets/2021-julia-user-developer-survey.pdf.

Deffebach P, Short T, JuliaData contributors (2023). ***DataFramesMeta.jl**: Metaprogramming Tools for **DataFrames**.* Julia package version 0.14.0, URL https://github.com/JuliaData/DataFramesMeta.jl.

Dowle M, Srinivasan A (2023a). *Introduction to **data.table**.* R package vignette, URL https://CRAN.R-project.org/web/packages/data.table/vignettes/datatable-intro.html.

Dowle M, Srinivasan A (2023b). ***data.table**: Extension of* data.frame. R package version 1.14.8, URL https://CRAN.R-project.org/package=data.table.

Gorecki J (2015). "Solve Common R Problems Efficiently with **data.table**." URL https://jangorecki.github.io/blog/2015-12-11/Solve-common-R-problems-efficiently-with-data.table.html.

Guliński K (2021). "**DTable** – An Early Performance Assessment of a New Distributed Table Implementation." *The Julia Blog.* URL https://julialang.org/blog/2021/12/dtable-performance-assessment/.

Kamiński B (2022). *Julia for Data Analysis.* Manning, Shelter Island. URL https://www.manning.com/books/julia-for-data-analysis.

Krumbiegel J (2022). ***Chain.jl**: A Julia Package for Piping a Value through a Series of Transformation Expressions Using a More Convenient Syntax than Julia's Native Piping Functionality.* Julia package version 0.5.0, URL https://github.com/jkrumbiegel/Chain.jl.

Krumbiegel J (2023). ***DataFrameMacros.jl**: Macros That Simplify Working with **DataFrames.jl**.* Julia package version 0.4.1, URL https://github.com/jkrumbiegel/DataFrameMacros.jl.

Lin D, Bouchet-Valat M, Noack A, Arslan A, JuliaStats contributors (2023). ***StatsBase.jl**: Basic Statistics for Julia.* Julia package version 0.34.0, URL https://github.com/JuliaStats/StatsBase.jl.

McKinney W (2010). "Data Structures for Statistical Computing in Python." In S Van der Walt, J Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 56–61. doi:10.25080/Majora-92bf1922-00a.

Müller K, Wickham H (2023). ***tibble**: Simple Data Frames.* R package version 3.2.1, URL https://CRAN.R-project.org/package=tibble.

Noronha F (2023). ***XLSX.jl**: **Excel** File Reader and Writer for the Julia Language.* Julia package version 0.9.0, URL https://github.com/felipenoris/XLSX.jl.

pandas Development Team (2023). "pandas-dev/pandas: **pandas** v2.1.0." doi:10.5281/zenodo.3509134.

Petersohn D, Macke S, Xin D, Ma W, Lee D, Mo X, Gonzalez JE, Hellerstein JM, Joseph AD, Parameswaran A (2020). "Towards Scalable Dataframe Systems." *Proceedings of the VLDB Endowment*, **13**(12), 2033–2046. doi:10.14778/3407790.3407807.

Quinn J, JuliaData contributors (2021a). ***Avro.jl**: Pure Julia Implementation for Reading/Writing Data in the **Avro** Format.* Julia package version 1.1.0, URL https://github.com/JuliaData/Avro.jl.

Quinn J, JuliaData contributors (2021b). ***JSONTables.jl**: **JSON3.jl** + **Tables.jl**.* Julia package version 1.0.3, URL https://github.com/JuliaData/JSONTables.jl.

Quinn J, JuliaData contributors (2023a). ***CSV.jl**: Utility Library for Working with CSV and Other Delimited Files in the Julia Programming Language.* Julia package version 0.10.10, URL https://github.com/JuliaData/CSV.jl.

Quinn J, JuliaData contributors (2023b). ***Tables.jl**: An Interface for Tables in Julia.* Julia package version 1.10.1, URL https://github.com/JuliaData/Tables.jl.

Quinn J, JuliaStrings contributors (2023). ***InlineStrings.jl**: Fixed-Width String Types for Julia.* Julia package version 1.4.0, URL https://github.com/JuliaStrings/InlineStrings.jl.

Quinn J, Kamiński B, JuliaData contributors (2023a). ***DataAPI.jl**: A Data-Focused Namespace for Packages to Share Functions.* Julia package version 1.15.0, URL https://github.com/JuliaData/DataAPI.jl.

Quinn J, Savastio M, Apache contributors (2023b). ***Arrow.jl**: Official Julia Implementation of **Apache Arrow**.* Julia package version 2.5.2, URL https://github.com/apache/arrow-julia.

Savastio M (2023). ***Parquet2.jl**: Pure Julia Implementation of **parquet** Tabular Data Binary Format.* Julia package version 0.2.15, URL https://github.com/JuliaIO/Parquet.jl.

Singh K, TidierOrg contributors (2023). ***Tidier.jl**: 100% Julia Implementation of the R **tidyverse** Mini-Language.* Julia package version 0.7.6, URL https://github.com/TidierOrg/Tidier.jl.

Storopoli J, Huijzer R, Alonso L (2021). "Julia Data Science." URL https://juliadatascience.io/.

Vertechi P, Borregaard MK, JuliaPlots contributors (2023). ***StatsPlots.jl**: Statistical Plotting Recipes for **Plots.jl***. Julia package version 0.15.5, URL https://github.com/JuliaPlots/StatsPlots.jl.

Vink R (2023). ***polars**: Fast Multi-Threaded DataFrame Library in Rust | Python | Node.js*. Rust crate version 0.29.0, URL https://github.com/pola-rs/polars.

Wickham H (2011). "The Split-Apply-Combine Strategy for Data Analysis." *Journal of Statistical Software*, **40**(1), 1–29. doi:10.18637/jss.v040.i01.

Wickham H (2014). "Tidy Data." *Journal of Statistical Software*, **59**(10), 1–23. ISSN 1548-7660. doi:10.18637/jss.v059.i10.

Wickham H, François R, Henry L, Müller K (2023). ***dplyr**: A Grammar of Data Manipulation*. R package version 1.1.2, URL https://CRAN.R-project.org/package=dplyr.

A. Code for benchmarks

In this appendix, we present the code to reproduce the benchmark results presented in Table 1. This code is also provided as separate files attached to this article. Note that running it takes several hours.

Each code was stored in a file and run using the command given in the comment on top of it. The **DataFrames.jl** code should be run first as it additionally creates CSV files that are later used in other files to ensure that the same data is used in all tests.

A.1. DataFrames.jl (file: `julia_bench.jl`)

Run using: `julia -t auto -project julia_bench.jl`.

```
using CSV, DataFrames, Random

Random.seed!(1234)
n = 50_000_000
k = 500_000
df = DataFrame(x=rand(n), grp=rand(1:k, n))
CSV.write("df.csv", df)
df1 = DataFrame(x = shuffle(1:n-1), y1 = randn(n - 1))
df2 = DataFrame(x = shuffle(2:n), y2 = randn(n - 1))
CSV.write("df1.csv", df1)
CSV.write("df2.csv", df2)

println("Julia aggregation time: ",
        minimum(@elapsed combine(groupby(df, :grp), :x => sum, nrow)
                for _ in 1:100))

println("Julia innerjoin time: ",
        minimum(@elapsed innerjoin(df1, df2, on = :x) for _ in 1:100))
println("Julia leftjoin time: ",
        minimum(@elapsed leftjoin(df1, df2, on = :x) for _ in 1:100))
println("Julia rightjoin time: ",
        minimum(@elapsed rightjoin(df1, df2, on = :x) for _ in 1:100))
println("Julia outerjoin time: ",
        minimum(@elapsed outerjoin(df1, df2, on = :x) for _ in 1:100))
```

A.2. data.table (file: `datatable_bench.r`)

Run using: `RScript datatable_bench.r`.

```
library("data.table")
library("microbenchmark")

dt = fread("df.csv")
bench_agg = microbenchmark(dt[, .(sum(x), .N), grp], times = 100)
```

```
cat("data.table aggregation time:", min(bench_agg$time)/ 10^9, "\n")

dt1 = fread("df1.csv")
dt2 = fread("df2.csv")
bench_innerjoin = microbenchmark(dt1[dt2, nomatch = NULL, on = "x"],
                                 times = 100)
cat("data.table innerjoin time:", min(bench_innerjoin$time)/ 10^9, "\n")
bench_leftjoin = microbenchmark(dt2[dt1, on = "x"], times = 100)
cat("data.table leftjoin time:", min(bench_leftjoin$time)/ 10^9, "\n")
bench_rightjoin = microbenchmark(dt1[dt2, on = "x"], times = 100)
cat("data.table rightjoin time:", min(bench_rightjoin$time)/ 10^9, "\n")
bench_outerjoin = microbenchmark(merge(dt1, dt2, by = "x", all = TRUE),
                                 times = 100)
cat("data.table outerjoin time:", min(bench_outerjoin$time)/ 10^9, "\n")
```

A.3. arrow/dplyr (file: `arrow_bench.r`)

Run using: `RScript arrow_bench.r`.

```
library("dplyr")
library("arrow")
library("microbenchmark")

df = read_csv_arrow("df.csv", as_data_frame=FALSE)
bench_agg = microbenchmark(collect(df %>%
                                   group_by(grp) %>%
                                   summarise(sum(x), n())),
                           times = 100)
cat("Arrow/dplyr aggregation time:", min(bench_agg$time)/ 10^9, "\n")

df1 = read_csv_arrow("df1.csv", as_data_frame=FALSE)
df2 = read_csv_arrow("df2.csv", as_data_frame=FALSE)
bench_innerjoin = microbenchmark(collect(inner_join(df1, df2, on = "x")),
                                 times = 100)
cat("Arrow/dplyr innerjoin time:", min(bench_innerjoin$time)/ 10^9, "\n")
bench_leftjoin = microbenchmark(collect(left_join(df1, df2, on = "x")),
                                times = 100)
cat("Arrow/dplyr leftjoin time:", min(bench_leftjoin$time)/ 10^9, "\n")
bench_rightjoin = microbenchmark(collect(right_join(df1, df2, on = "x")),
                                          times = 100)
cat("Arrow/dplyr rightjoin time:", min(bench_rightjoin$time)/ 10^9, "\n")
bench_outerjoin = microbenchmark(collect(full_join(df1, df2, on = "x")),
                                 times = 100)
cat("Arrow/dplyr outerjoin time:", min(bench_outerjoin$time)/ 10^9, "\n")
```

A.4. Polars (file: polars_bench.py)

Run using: `python polars_bench.r`.

```
import time
import polars as pl

df_polars = pl.read_csv('df.csv')

def test_agg_polars(df):
    begin = time.time()
    df.groupby("grp").agg([pl.sum('x'), pl.count('x').alias('nrow')])
    end = time.time()
    return end-begin

print('Polars aggregation time:',
      min([test_agg_polars(df_polars) for i in range(100)]))

df1_polars = pl.read_csv('df1.csv')
df2_polars = pl.read_csv('df2.csv')

def test_join_polars(df1, df2):
    begin = time.time()
    df1.join(df2, on="x", how="inner")
    end = time.time()
    jinner = end-begin
    begin = time.time()
    df1.join(df2, on="x", how="left")
    end = time.time()
    jleft = end-begin
    # right join is not supported in Polars
    begin = time.time()
    df1.join(df2, on="x", how="outer")
    end = time.time()
    jouter = end-begin
    return (jinner, jleft, jouter)

res_join_pl = [test_join_polars(df1_polars, df2_polars) for i in range(100)]
print('Polars innerjoin time:',
      min([v[0] for v in res_join_pl]))
print('Polars leftjoin time:',
      min([v[1] for v in res_join_pl]))
print('Polars outerjoin time:',
      min([v[2] for v in res_join_pl]))
```

A.5. pandas (file: pandas_bench.py)

Run using: `python pandas_bench.r`.

```
import pandas as pd
import time
df_pandas = pd.read_csv("df.csv", engine='pyarrow')

def test_agg_pandas(df):
    begin = time.time()
    df.groupby("grp").agg({'x': ['sum', 'count']})
    end = time.time()
    return end-begin

print('Pandas aggregation time:',
      min([test_agg_pandas(df_pandas) for i in range(100)]))

df1_pandas = pd.read_csv("df1.csv", engine='pyarrow')
df2_pandas = pd.read_csv("df2.csv", engine='pyarrow')

def test_join_pandas(df1, df2):
    begin = time.time()
    df1.merge(df2, on='x', how='inner')
    end = time.time()
    jinner = end-begin
    begin = time.time()
    df1.merge(df2, on='x', how='left')
    end = time.time()
    jleft = end-begin
    begin = time.time()
    df1.merge(df2, on='x', how='right')
    end = time.time()
    jright = end-begin
    begin = time.time()
    df1.merge(df2, on='x', how='outer')
    end = time.time()
    jouter = end-begin
    return (jinner, jleft, jright, jouter)

res_join_pd = [test_join_pandas(df1_pandas, df2_pandas) for i in range(100)]
print('Pandas innerjoin time:',
      min([v[0] for v in res_join_pd]))
print('Pandas leftjoin time:',
      min([v[1] for v in res_join_pd]))
print('Pandas rightjoin time:',
      min([v[2] for v in res_join_pd]))
print('Pandas outerjoin time:',
      min([v[3] for v in res_join_pd]))
```

Affiliation:

Milan Bouchet-Valat
Institut national d'études démographiques (INED)
9 Cours des Humanités
F-93300 Aubervilliers, France
E-mail: milan.bouchet-valat@ined.fr
URL: http://bouchet-valat.site.ined.fr

Bogumił Kamiński
Decision Support and Analysis Division
SGH Warsaw School of Economics
Al. Niepodległości 162
02-554 Warsaw, Poland
E-mail: bkamins@sgh.waw.pl
URL: http://bogumilkaminski.pl/about/

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
September 2023, Volume 107, Issue 4 *Submitted:* 2022-01-06
doi:10.18637/jss.v107.i04 *Accepted:* 2023-06-08