Spring 20205 - STAT244 - Julia and Data

DataFrames, ...

Olorundamilola 'Dami' Kazeem

2025-02-27

Spring 2025 - STAT244 - Julia and Data

Contents ... Presentation Outline

```julia
using Pkg

Pkg.activate(".")

Pkg.add("BenchmarkTools")

Pkg.add("DataFrames")
Pkg.add("DataFramesMeta")

Pkg.add("CSV")
Pkg.add("Arrow")

Pkg.add("Distributed")
Pkg.add("Dagger")

Pkg.add("Plots")
Pkg.add("GR")
```

```
  Activating project at `~/Projects/spring2025-stat244-julia-and-data-intro`
   Resolving package versions...
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Project.toml`
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Project.toml`
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Project.toml`
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Project.toml`
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Project.toml`
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Project.toml`
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Project.toml`
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Project.toml`
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Project.toml`
  No Changes to `~/Projects/spring2025-stat244-julia-and-data-intro/Manifest.toml`
```

```julia
using BenchmarkTools

using DataFrames
using DataFramesMeta

using CSV
using Arrow

using Distributed
using Dagger

using Plots
using GR
```

```
WARNING: using Dagger.In in module Main conflicts with an existing identifier.
WARNING: using Dagger.Out in module Main conflicts with an existing identifier.
```

Introduction

What is **data literacy**?

As defined by Wikipedia:

- It is the ability to read, understand, create, and communicate **data** as information.
- It focuses on the competencies involved in working with data.
- It requires certain skills involving reading and understanding data.

What is data?

As defined by Wikipedia:

- A collection of discrete or continuous values that convey information, describing the quantity, quality, fact, statistics, other basic units of meaning
- A sequence of symbols that may be further interpreted formally; and may be used as variables in a computational process, which may represent abstract ideas or concrete measurements

Figure 1: Data_Types

Julia and the Data Ecosystem v0.0 - Initial Outline

How to interact with your data in the Julia Data JuliaData** ecosystem?**

NOTE: There's a lot of ground to cover when it comes to data! Below are just of few directions:

```
# ...
```

Data Analysis

Julia's ecosystem includes mature packages like **DataFrames.jl** and **Query.jl**, which provide robust capabilities for data manipulation, transformation, and exploratory analysis. These tools allow users to work with large datasets in a way that is both intuitive and high-performance.

Data Processing

For tasks involving data ingestion and transformation, Julia offers packages such as **CSV.jl** for fast, efficient reading and writing of CSV files, as well as other tools for parsing and processing different data formats. Its design emphasizes speed and efficiency, making it well-suited for handling big data and real-time processing scenarios.

Data Science

Julia's performance and syntax make it a strong candidate for data science applications. The language supports statistical modeling, machine learning, and scientific computing through libraries like **MLJ.jl** for machine learning and **Flux.jl** for deep learning. This makes it possible to build and deploy predictive models and complex analytical pipelines with ease.

Databases

Interacting with databases in Julia via **Julia Database Interfaces** is streamlined by packages like **LibPQ.jl** for PostgreSQL, **SQLite.jl** for lightweight database management, and **ODBC.jl** for a range of other SQL databases. These libraries enable efficient data retrieval, storage, and manipulation directly within the Julia environment.

Data Miscellaneous

Beyond the core areas, Julia excels in other data-related domains such as: - **Data Cleaning and Wrangling:** Leveraging its high-level syntax and powerful libraries to prepare data for analysis, such as Cleaner.jl a toolbox of simple solutions for common data cleaning problems. - **Data Visualization:** Utilizing packages like **Plots.jl**, **Makie.jl**, and **Gadfly.jl** for creating both static and interactive visualizations. - **Parallel and Distributed Computing:** Julia's native support for parallelism makes it a great choice for scaling data processing and analysis tasks across multiple cores or nodes.

Overall, Julia's growing ecosystem and its blend of high-level expressiveness with low-level performance make it a versatile language for everything from exploratory data analysis to building large-scale data processing and machine learning pipelines.

```
# ...
```

1. Introduction to DataFrames.jl

DataFrames.jl provides a set of tools for working with tabular data in Julia. Its design and functionality are similar to those of `pandas` (in Python) and `data.frame`, `data.table` and `dplyr` (in R), making it a great general purpose data science tool.

To work with tabular datasets and perform common data manipulations

1. What is a DataFrame?
2. Key Features of DataFrames.jl
3. Basic Operations
4. Mock(or Real) World Example

```julia
# Create an empty dataframe

df = DataFrame()
println(df)
```

```
0×0 DataFrame
```

1.1. What is a DataFrame?

- **Definition:**
  - A DataFrame is a tabular data structure, similar to a spreadsheet or SQL table, where data is organized in rows (observations) and columns (heterogeneous types).
- **Purpose:**
  - Used for storing, manipulating, and analyzing structured data.
- **Analogy:**
  - Think of it as a table in Excel or a table in a database.
  - Like a spreadsheet or SQL table, but optimized for programmatic workflows.

```julia
# Create a simple DataFrame

df0 = DataFrame(
    Name = ["Alice", "Bob", "Charlie"],
    Age = [25, 30, 35],
    Salary = [50_000, 75_000, 90_000]
);
```

```julia
println(df0)
```

```
3×3 DataFrame
 Row   Name      Age    Salary
       String    Int64  Int64

   1   Alice       25    50000
   2   Bob         30    75000
   3   Charlie     35    90000
```

```julia
mean(df0.Salary)
```

```
71666.66666666667
```

NOTE

- DataFrames.jl's guiding principles can be found here; however, two core ones are listed below:
  - Stay consistent with Julia' `Base` module functions.
  - Minimize the number of function `DataFrames.jl` provides.
- Columns can have different data types (e.g., String, Int64).
- Rows represent individual observations, and columns represent variables.

1.2. Key Features of DataFrame.jl

- **Columnar Storage:** Efficient for column-based operations. Optimized for column-wise operations (e.g., `mean(df.Salary)`).
- **Missing Data Handling:** Built-in support for Missing type (e.g., `dropmissing(df)`).
- **Integration:** Works seamlessly with other Julia packages (e.g., CSV.jl, Arrow.jl, Plots.jl).
- **Performance:** Optimized for fast data manipulation.
- **Flexibility:** Built for speed (no copies, vectorized operations) Supports a wide range of data types and operations.

```julia
# Handling missing data

df = DataFrame(Name = ["Alice", missing, "Charlie"], Age = [25, 30, missing]);
```

```julia
println(df)
```

```
3×2 DataFrame
 Row   Name     Age
       String?  Int64?

   1   Alice         25
   2   missing       30
   3   Charlie  missing
```

```julia
# Remove rows with missing values
clean_df = dropmissing(df)
println(clean_df)
```

```
1×2 DataFrame
 Row   Name    Age
       String  Int64

   1   Alice      25
```

NOTE

- Missing type allows for explicit handling of incomplete data.
- Use `dropmissing(df)` to remove rows with missing values.
- Use `dropmissing`, `coalesce`, or `replace!` to handle missing data.

1.3. Basic Operations

- **Reading and Writing a DataFrame:** To/From vectors, dictionaries, csv files, or databases.
- **Accessing Data:** Rows, columns, and individual elements.
- **Modifying Data:** Adding, removing, and transforming columns.

- **Filtering and Sorting:** Subsetting rows and ordering data.

Accessing Data

```
# Create a DataFrame with mock data
df = DataFrame(
    ID = 1:5,
    Name = ["Alice", "Bob", "Charlie", "David", "Eve"],
    Age = rand(20:30, 5),       # Random ages between 20 and 30
    Score = round.(rand(5) * 100, digits=2),  # Random scores between 0 and 100
    Active = rand(Bool, 5)      # Random boolean values
)

# Save the DataFrame to a CSV file
CSV.write("data.csv", df)
```

```
"data.csv"
```

```
df = CSV.read("data.csv", DataFrame)
println(df)
```

```
5×5 DataFrame
 Row   ID     Name     Age    Score    Active
       Int64  String7  Int64  Float64  Bool

   1       1  Alice       23    85.37   false
   2       2  Bob         22    22.09    true
   3       3  Charlie     28    24.42    true
   4       4  David       28    24.84   false
   5       5  Eve         28    22.87    true
```

```
# Acessing Data by Column Name
println(df.Name)
```

```
String7["Alice", "Bob", "Charlie", "David", "Eve"]
```

- Rows: `df[1:3, :]` or use Julia's built-in `filter(:Age => >(25), df)`.
- Columns: `df.Name` (copying) or `df[!, :Name]` (non-copying).

```
# Accessing Data by Row Index (and for all Columns)
println(df[5, :])
```

```
DataFrameRow
 Row   ID     Name     Age    Score    Active
       Int64  String7  Int64  Float64  Bool

   5       5  Eve         28    22.87    true
```

Modifying Data

```
# Modifying in-place (non-copy) DataFrame and Data by Column Name

df[!, :Bonus] = df.Score .* 0.1;
```

```
println(df)
```

```
5×6 DataFrame
 Row   ID     Name     Age    Score    Active  Bonus
       Int64  String7  Int64  Float64  Bool    Float64

   1       1  Alice       23    85.37   false    8.537
   2       2  Bob         22    22.09    true    2.209
   3       3  Charlie     28    24.42    true    2.442
   4       4  David       28    24.84   false    2.484
   5       5  Eve         28    22.87    true    2.287
```

```
# Remove a column
select!(df, Not(:Bonus));
```

```
println(df)
```

```
5×5 DataFrame
 Row   ID     Name     Age    Score    Active
       Int64  String7  Int64  Float64  Bool

   1       1  Alice       23    85.37   false
   2       2  Bob         22    22.09    true
   3       3  Charlie     28    24.42    true
   4       4  David       28    24.84   false
   5       5  Eve         28    22.87    true
```

Filtering & Sorting Data

```
# Filter using Base Julia
df_filtered = filter(row -> row.Age >= 25, df);
```

```
println(df_filtered)
```

```
3×5 DataFrame
 Row   ID     Name     Age    Score    Active
       Int64  String7  Int64  Float64  Bool

   1       3  Charlie     28    24.42    true
   2       4  David       28    24.84   false
   3       5  Eve         28    22.87    true
```

```
# Sort by Score
df_sorted_by_score = sort(df, :Score);
```

```julia
println(df_sorted_by_score)
```

```
5×5 DataFrame
 Row   ID     Name     Age    Score    Active
       Int64  String7  Int64  Float64  Bool

   1      2   Bob         22    22.09    true
   2      5   Eve         28    22.87    true
   3      3   Charlie     28    24.42    true
   4      4   David       28    24.84   false
   5      1   Alice       23    85.37   false
```

NOTE - Use `filter` and `sort` for row-wise operations.

- Use `select!`, `transform`, and `combine` for column operations.

```julia
# ...
```

1.4. Mock (or Real World) Example

Scenario: Analyze employee data to calculate average salary by age group.

```julia
# Create a mock student dataset
students = DataFrame(
    ID = 1:8,
    Name = ["Alice", "Bob", "Charlie", "David", "Eve", "Frank", "Grace", "Heidi"],
    Age = [14, 15, 16, 14, 15, 16, 14, 15],
    Score = [85, 92, 78, 88, 95, 81, 90, 87],
    Active = [true, true, false, true, true, false, true, true],
    Grade = [9, 10, 9, 10, 11, 11, 12, 12],
    Attendance = [95, 88, 92, 85, 90, 78, 96, 89],
    Scholarship = [false, true, false, true, false, true, false, true]
);

# Write the DataFrame to a CSV file
CSV.write("students.csv", students);
```

```julia
# Display the dataset
students = CSV.read("students.csv", DataFrame)
println("Student Dataset:")
println(students)
```

```
Student Dataset:
8×8 DataFrame
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1      1   Alice       14     85    true      9          95        false
   2      2   Bob         15     92    true     10          88         true
```

```
   3      3  Charlie    16     78   false     9          92         false
   4      4  David      14     88    true    10          85          true
   5      5  Eve        15     95    true    11          90         false
   6      6  Frank      16     81   false    11          78          true
   7      7  Grace      14     90    true    12          96         false
   8      8  Heidi      15     87    true    12          89          true
```

Analysis Tasks

- Task 1: Calculate the average score by grade.
- Task 2: Identify top-performing students (score 90).
- Task 3: Calculate the average attendance for scholarship vs. non-scholarship students.

```
# Task 1: Average score by grade
avg_score_by_grade = combine(
    groupby(students, :Grade),
    :Score => mean => :Average_Score
);

# Display results
println("\nAverage Score by Grade:")
println(avg_score_by_grade)
```

```
Average Score by Grade:
4×2 DataFrame
 Row   Grade  Average_Score
       Int64  Float64

   1       9           81.5
   2      10           90.0
   3      11           88.0
   4      12           88.5
```

```
# Task 2: Top-performing students (score  90)
top_students = filter(row -> row.Score >= 90, students)

println("\nTop-Performing Students (Score  90):")
println(top_students)
```

```
Top-Performing Students (Score  90):
3×8 DataFrame
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1      2  Bob        15     92    true    10          88          true
   2      5  Eve        15     95    true    11          90         false
   3      7  Grace      14     90    true    12          96         false
```

```
# Task 3: Average attendance for scholarship vs. non-scholarship students
avg_attendance_by_scholarship = combine(
    groupby(students, :Scholarship),
    :Attendance => mean => :Average_Attendance
)

println("\nAverage Attendance by Scholarship Status:")
println(avg_attendance_by_scholarship)
```

```
Average Attendance by Scholarship Status:
2×2 DataFrame
 Row   Scholarship  Average_Attendance
       Bool         Float64

   1        false               93.25
   2         true               85.0
```

NOTE:

- Task 1: Use groupby + combine for grouped calculations.
- Task 2: Use filter to subset rows based on conditions.
- Task 3: Compare groups (e.g., scholarship vs. non-scholarship) using groupby.

4. Introduction to DataFramesMeta.jl

1. What is DataFramesMeta.jl?
2. Key Features of DataFramesMeta.jl
3. Basic Operations
4. Mock (or Real) World Example

Goal: *Show how DataFramesMeta.jl simplifies and streamlines data manipulation with a chainable, expressive syntax.*

4.1. What is DataFramesMeta.jl?

- **Definition:** A meta-package built on DataFrames.jl that provides a concise, chainable syntax for data manipulation.
- **Inspiration:** Borrows ideas from R's `dplyr` and Python's `pandas`.
- **Why Use It?:** Reduces boilerplate code, improves readability, and makes workflows more intuitive.

Key: - DataFramesMeta.jl is not a replacement for DataFrames.jl—it's a **syntactic sugar*** layer on top.

- It is an example of ***metaprogramming tools for DataFrames.jl objects to provide more convenient syntax.***

DataFrames.jl has the functions select, transform, and combine, as well as the in-place select! and transform! for manipulating data frames.

DataFramesMeta.jl provides the macros @select, @transform, @combine, @select!, and @transform! to mirror these functions with more convenient syntax. Inspired by dplyr in R and LINQ in C#.

In addition, DataFramesMeta provides

- @orderby, for sorting data frames
- @subset and @subset!, for keeping rows of a data frame matching a given condition
- Row-wise versions of the above macros in the form of @rtransform, @rtransform!, @rselect, @rselect!, @rorderby, @rsubset, and @rsubset!.
- @rename and @rename! for renaming columns
- @groupby for grouping data
- @by, for grouping and combining a data frame in a single step
- @with, for working with the columns of a data frame with high performance and convenient syntax
- @eachrow and @eachrow! for looping through rows in data frame, again with high performance and convenient syntax.
- @byrow for applying functions to each row of a data frame (only supported inside other macros).
- @passmissing for propagating missing values inside row-wise DataFramesMeta.jl transformations.
- @astable to create multiple columns within a single transformation.
- @chain, from Chain.jl for piping the above macros together, similar to magrittr's %>% in R.
- @label! and @note! for attaching metadata to columns.

```
# ...
```

Coverage:

1. Chainable Syntax: Use @chain to create readable, step-by-step workflows.

2. Symbol-Based Operations: Refer to columns using symbols (:column_name).

3. Common Verbs:

   - @select: Select or rename columns.
   - @transform: Add or modify columns.
   - @subset: Filter rows based on conditions.
   - @groupby: Group data by one or more columns.
   - @combine: Summarize grouped data.
   - @orderby: Sort rows by one or more columns.

Key Point:

- DataFramesMeta.jl is ideal for users familiar with dplyr or pandas.

```julia
# Read in the dataset
students = CSV.read("students.csv", DataFrame)
println("Student Dataset:")
println(students)
```

```
Student Dataset:
```

```
8×8 DataFrame
 Row  ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
      Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1      1  Alice       14     85    true      9          95        false
   2      2  Bob         15     92    true     10          88         true
   3      3  Charlie     16     78   false      9          92        false
   4      4  David       14     88    true     10          85         true
   5      5  Eve         15     95    true     11          90        false
   6      6  Frank       16     81   false     11          78         true
   7      7  Grace       14     90    true     12          96        false
   8      8  Heidi       15     87    true     12          89         true
```

…

```julia
# Select specific columns
selected = @chain students begin
    @select(:Name, :Score, :Grade)
end

# Rename columns
renamed = @chain students begin
    @select(:Student_Name = :Name, :Exam_Score = :Score)
end

println("Selected Columns:")
println(selected)

println("\nRenamed Columns:")
println(renamed)
```

```
Selected Columns:
8×3 DataFrame
 Row  Name     Score  Grade
      String7  Int64  Int64

   1  Alice       85      9
   2  Bob         92     10
   3  Charlie     78      9
   4  David       88     10
   5  Eve         95     11
   6  Frank       81     11
   7  Grace       90     12
   8  Heidi       87     12

Renamed Columns:
```

```
8×2 DataFrame
 Row   Student_Name   Exam_Score
       String7        Int64

   1   Alice                  85
   2   Bob                    92
   3   Charlie                78
   4   David                  88
   5   Eve                    95
   6   Frank                  81
   7   Grace                  90
   8   Heidi                  87
```

```
# @transform: Add or Modify Columns

# Add a new column for pass/fail status
transformed = @chain students begin
    @transform(:Pass = :Score .>= 90)
end

println("Transformed DataFrame:")
println(transformed)
```

```
Transformed DataFrame:
8×9 DataFrame
 Row   ID     Name      Age    Score  Active  Grade  Attendance  Scholarship  Pass
       Int64  String7   Int64  Int64  Bool    Int64  Int64       Bool         Bool

   1       1  Alice        14     85    true      9          95        false  false
   2       2  Bob          15     92    true     10          88         true   true
   3       3  Charlie      16     78   false      9          92        false  false
   4       4  David        14     88    true     10          85         true  false
   5       5  Eve          15     95    true     11          90        false   true
   6       6  Frank        16     81   false     11          78         true  false
   7       7  Grace        14     90    true     12          96        false   true
   8       8  Heidi        15     87    true     12          89         true  false
```

```
# @subset: Filter Rows Based on Conditions

# Filter active students with a score   90
filtered = @chain students begin
    @subset(:Active .&& :Score .>= 90)
end

println("Filtered DataFrame:")
println(filtered)
```

```
Filtered DataFrame:
```

```
3×8 DataFrame
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1       2  Bob         15     92    true     10          88         true
   2       5  Eve         15     95    true     11          90        false
   3       7  Grace       14     90    true     12          96        false
```

```julia
# @groupby: Group Data by One or More Columns

# Group by Grade
grouped = @chain students begin
    @groupby(:Grade)
end

println("Grouped DataFrame:")
println(grouped)
```

```
Grouped DataFrame:
GroupedDataFrame with 4 groups based on key: Grade
Group 1 (2 rows): Grade = 9
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1       1  Alice       14     85    true      9          95        false
   2       3  Charlie     16     78   false      9          92        false
Group 2 (2 rows): Grade = 10
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1       2  Bob         15     92    true     10          88         true
   2       4  David       14     88    true     10          85         true
Group 3 (2 rows): Grade = 11
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1       5  Eve         15     95    true     11          90        false
   2       6  Frank       16     81   false     11          78         true
Group 4 (2 rows): Grade = 12
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1       7  Grace       14     90    true     12          96        false
   2       8  Heidi       15     87    true     12          89         true
```

```julia
# @combine: Summarize Grouped Data
```

```
# Calculate average score by grade
combined = @chain students begin
    @groupby(:Grade)
    @combine(:Average_Score = mean(:Score))
end

println("Combined DataFrame:")
println(combined)
```

```
Combined DataFrame:
4×2 DataFrame
 Row   Grade  Average_Score
       Int64  Float64

   1       9           81.5
   2      10           90.0
   3      11           88.0
   4      12           88.5
```

```
# @orderby: Sort Rows by One or More Columns

# Sort by Score in descending order
sorted = @chain students begin
    @orderby(-:Score)
end

println("Sorted DataFrame:")
println(sorted)
```

```
Sorted DataFrame:
8×8 DataFrame
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1       5  Eve         15     95    true     11          90        false
   2       2  Bob         15     92    true     10          88         true
   3       7  Grace       14     90    true     12          96        false
   4       4  David       14     88    true     10          85         true
   5       8  Heidi       15     87    true     12          89         true
   6       1  Alice       14     85    true      9          95        false
   7       6  Frank       16     81   false     11          78         true
   8       3  Charlie     16     78   false      9          92        false
```

```
# @orderby: Sort Rows by One or More Columns
# Sort by Score in descending order
sorted = @chain students begin
    @orderby(+:Score)
```

```julia
end

println("Sorted DataFrame:")
println(sorted)
```

```
Sorted DataFrame:
8×8 DataFrame
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool

   1      3   Charlie     16     78   false      9          92        false
   2      6   Frank       16     81   false     11          78         true
   3      1   Alice       14     85    true      9          95        false
   4      8   Heidi       15     87    true     12          89         true
   5      4   David       14     88    true     10          85         true
   6      7   Grace       14     90    true     12          96        false
   7      2   Bob         15     92    true     10          88         true
   8      5   Eve         15     95    true     11          90        false
```

Let's revisit the 3 tasks from before using DataFramesMeta.jl!

```julia
# Task 1: Average score by grade
avg_score_by_grade = @chain students begin
    @groupby(:Grade)
    @combine(:Average_Score = mean(:Score))
end

println(avg_score_by_grade)
```

```
4×2 DataFrame
 Row   Grade  Average_Score
       Int64  Float64

   1       9           81.5
   2      10           90.0
   3      11           88.0
   4      12           88.5
```

```julia
# Task 2: Top-performing students (score   90)
top_students = @chain students begin
    @subset!(:Score .>= 90)  # Correct usage of @subset
end

println(top_students)
```

```
3×8 DataFrame
 Row   ID     Name     Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String7  Int64  Int64  Bool    Int64  Int64       Bool
```

```
   1       2  Bob          15      92     true      10           88          true
   2       5  Eve          15      95     true      11           90         false
   3       7  Grace        14      90     true      12           96         false
```

```
# Task 3: Average attendance for scholarship vs. non-scholarship students
avg_attendance_by_scholarship = @chain students begin
    @groupby(:Scholarship)
    @combine(:Average_Attendance = mean(:Attendance))
end

println(avg_attendance_by_scholarship)
```

```
2×2 DataFrame
 Row  Scholarship  Average_Attendance
      Bool         Float64

   1        false                93.0
   2         true                88.0
```

…

5. Introduction to Arrow.jl

1. What is Apache Arrow?
2. Why use Arrow?
3. Arrow.jl in Julia
4. Mock (or Real) World Example

```
# ...
```

5.1 What is Arrow i.e. Apache Arrow?

Motivation: *Who here has struggled with slow CSV files? If you have, then try Arrow!*

Definition: Arrow.jl is a Julia package that provides an interface to the Apache Arrow format, a cross-language development platform for in-memory data. Arrow is designed for high-performance data interchange and storage, making it ideal for working with large datasets and sharing data between different programming languages (e.g., Julia, Python, R, C++).

Key Features: - Columnar Format: Optimized for columnar operations (e.g., analytics, machine learning). - **Zero-Copy Reads:** No data copying between systems (e.g., Python Julia). - **Language Interoperability:** Works seamlessly with Python, R, C++, and more.

Key Point: - Arrow is ideal for big data workflows and multi-language environments.

```
# ...
```

5.2 Why Use Arrow.jl?

- **Performance:** Arrow is faster than CSV/JSON for reading and writing large datasets.
- **Memory Efficiency:** Columnar format reduces memory usage.
- **Interoperability:** Share data between Julia and other languages (e.g., Python, R).
- **Integration:** Works seamlessly with DataFrames.jl and other Julia data tools.

Key Point: - Arrow.jl is the Julia interface to Apache Arrow, enabling high-performance data workflows.

```
# ...
```

CSV vs. Arrow:

- Arrow is 10–100x faster for reading/writing large datasets.
- Arrow uses less memory due to its columnar format.

Example Benchmark:

- Load a 1GB CSV file vs. a 1GB Arrow file.

Key Point:

- Arrow is the best choice for big data workflows.

5.4 Mock (or Real-World) Example

```
# Read in the dataset
students = CSV.read("students.csv", DataFrame)

# Save the DataFrame to an Arrow file
Arrow.write("students_FROMcsv_TO.arrow", students)
```

```
"students_FROMcsv_TO.arrow"
```

```
# Load the Arrow file back into Julia

arrow_table = Arrow.Table("students_FROMcsv_TO.arrow")
loaded_df = DataFrame(arrow_table)
println(loaded_df)
```

```
8×8 DataFrame
 Row   ID     Name      Age    Score  Active  Grade  Attendance  Scholarship
       Int64  String    Int64  Int64  Bool    Int64  Int64       Bool

   1       1  Alice        14     85    true      9          95        false
   2       2  Bob          15     92    true     10          88         true
   3       3  Charlie      16     78   false      9          92        false
   4       4  David        14     88    true     10          85         true
   5       5  Eve          15     95    true     11          90        false
   6       6  Frank        16     81   false     11          78         true
```

7	7	Grace	14	90	true	12	96	false
8	8	Heidi	15	87	true	12	89	true

CSV vs Arrow Benchmark

```
# Create a large dataset (1 million rows)
# n = 1_000_000_000 # trillion is too much for a demo :-)
n = 1_000_000
large_df = DataFrame(
    ID = 1:n,
    Name = rand(["Alice", "Bob", "Charlie", "David", "Eve"], n),
    Age = rand(14:18, n),
    Score = rand(50:100, n),
    Active = rand([true, false], n),
    Grade = rand(9:12, n),
    Attendance = rand(70:100, n),
    Scholarship = rand([true, false], n)
)

# Save the dataset as CSV and Arrow
CSV.write("large_data.csv", large_df);
Arrow.write("large_data.arrow", large_df);
```

```
### Local Machine
# Benchmarking Read Performance:
#   507.539 ms (821 allocations: 68.71 MiB)
#   187.712  s (578 allocations: 28.24 KiB)
###

### Remote Machine Arwen
# Benchmarking Read Performance:
#   442.207 ms (820 allocations: 68.26 MiB)
#   157.131  s (513 allocations: 28.38 KiB)
###

# Benchmark reading
println("Benchmarking Read Performance:")
@btime CSV.read("large_data.csv", DataFrame)
@btime Arrow.Table("large_data.arrow")
```

```
Benchmarking Read Performance:
  446.588 ms (813 allocations: 69.86 MiB)
  162.796  s (513 allocations: 28.38 KiB)
```

```
Arrow.Table with 1000000 rows, 8 columns, and schema:
 :ID           Int64
 :Name         String
```

```
:Age          Int64
:Score        Int64
:Active       Bool
:Grade        Int64
:Attendance   Int64
:Scholarship  Bool
```

```julia
### Local Machine
# Benchmarking Write Performance:
# 1.085 s (33994424 allocations: 888.93 MiB)
# 69.193 ms (324 allocations: 27.04 MiB)
###

### Remote Machine Arwen
# Benchmarking Write Performance:
#   765.590 ms (33994399 allocations: 888.93 MiB)
#   471.580 ms (407 allocations: 19.83 MiB)
###

# Benchmark writing
println("\nBenchmarking Write Performance:")
@btime CSV.write("large_data.csv", large_df)
@btime Arrow.write("large_data.arrow", large_df)
```

```
Benchmarking Write Performance:
  821.423 ms (33994408 allocations: 888.93 MiB)
  477.011 ms (407 allocations: 19.83 MiB)
```

```
"large_data.arrow"
```

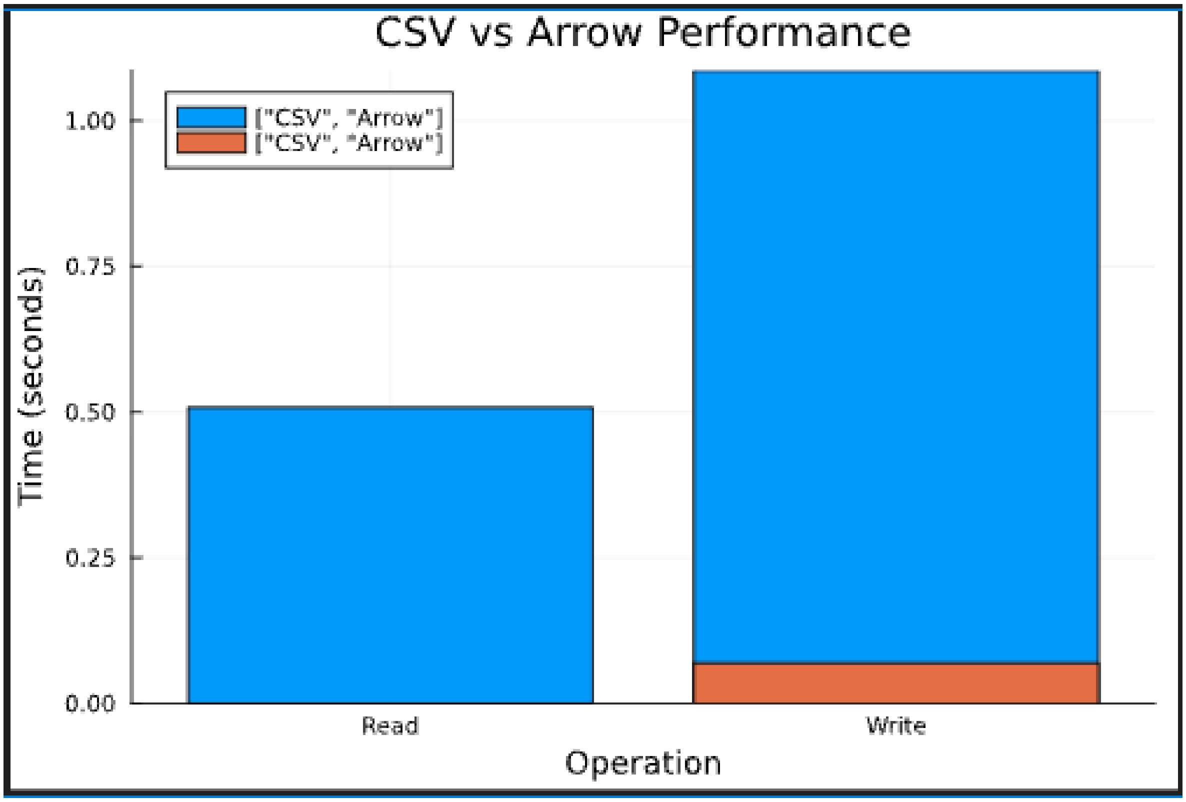
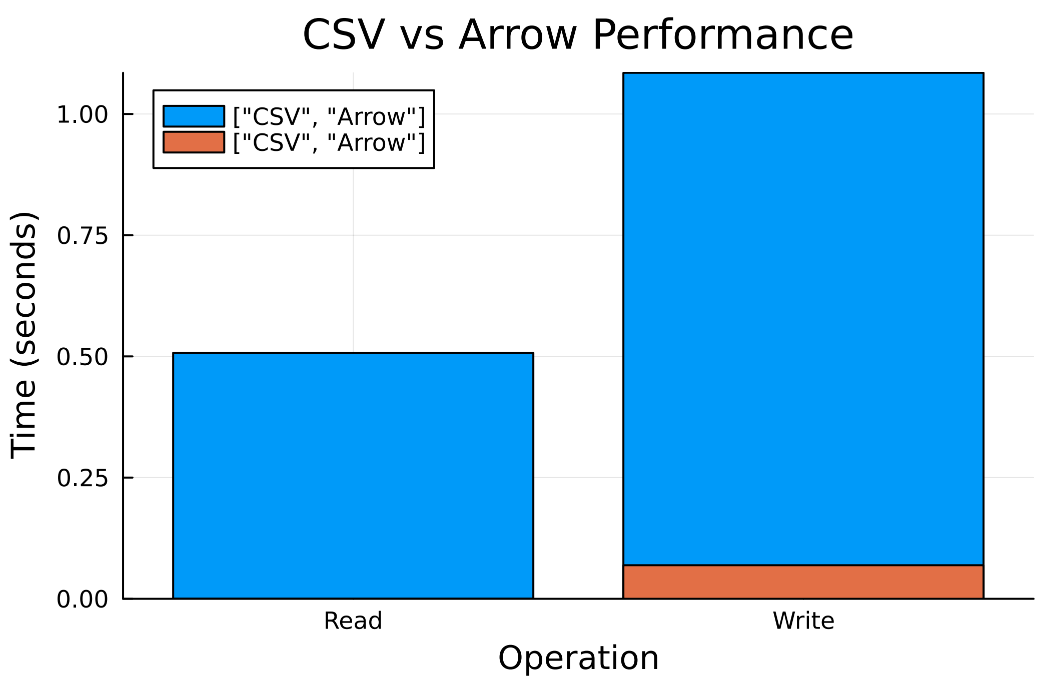


Figure 2: CSV_vs_Arrow

```
using Plots

###
# Plot takes about 1 minutes
###

###
# Benchmarking Read Performance:
#   507.539 ms (821 allocations: 68.71 MiB)  --> 0.507539000 s
#   187.712  s (578 allocations: 28.24 KiB)  --> 0.000187712 s
###

###
# Benchmarking Write Performance:
# 1.085 s (33994424 allocations: 888.93 MiB) --> 1.085000000 s
# 69.193 ms (324 allocations: 27.04 MiB)     --> 0.069193000 s
###
```

```
# Data for plotting

using Plots

# Data for plotting
operations = ["Read", "Write"]
csv_times = [0.507539000, 1.085000000]  # Replace with actual benchmark results
arrow_times = [0.000187712, 0.069193000]  # Replace with actual benchmark results

# Plot
bar(operations, [csv_times arrow_times],
    label = ["CSV", "Arrow"],
    xlabel = "Operation",
    ylabel = "Time (seconds)",
    title = "CSV vs Arrow Performance")
```



6. Introduction to Big Data Analysis in Julia

1. Challenges of Big Data
2. Julia's Approach to Big Data
3. Mention Distributed Computing via Distributed.jl
4. Mention Out-of-Core Computing via Dagger.jl

```
# ...
```

6.1. Challenges of Big Data

Goal: *Demonstrate how Julia handles large datasets using distributed and out-of-core computing.*

- **Volume:** Datasets too large to fit into memory.
- **Velocity:** Data arriving in real-time streams.
- **Variety:** Structured, semi-structured, and unstructured data.
- **Julia's Solution:** Distributed and out-of-core computing.

Key Point:

- Julia provides tools to handle big data efficiently, even on a single machine or a cluster.

```
# ...
```

6.2. Julia's Approach to Big Data

Distributed Parallel Computing: - Use multiple processes or machines to parallelize computations. - Tools: Distributed.jl, MPI.jl.

Out-of-Core Parallel Computing: - Process data that doesn't fit into memory by working in chunks. - Tools: Dagger.jl, CSV.jl (with chunking).

Integration: - Works seamlessly with DataFrames.jl, CSV.jl, Arrow.jl, and other data tools.

Key Point:

- Julia's ecosystem is designed for scalability.

```
# ...
```

6.3. Mention Distributed Computing via Distributed.jl

Distributed.jl is a built-in Julia module that provides tools for parallel and distributed computing. It allows you to distribute computations across multiple processes or machines, enabling you to tackle larger problems and speed up computations by leveraging multiple CPU cores or even clusters of machines.

- **Parallel Loops**: Use `@distributed` to distribute loops across multiple workers, enabling faster computation for large-scale tasks.

- **Task Parallelism**: Use `@spawn` to run tasks asynchronously on workers, ideal for independent or background computations.

```
###
# using Distributed
###

# nprocs()
```

```
###
# Add Worker Processes: Add one Julia worker per CPU core
###

# addprocs(4)  # Add 4 local worker processes
```

```
###
# Parallel Map-Reduce:
###

# @distributed (+) for i in 1:1_000_000
#     i^2
# end
```

```
###
# Load LinearAlgebra on all workers
###

# @everywhere using LinearAlgebra
```

```
###
# Distribute Computations
# Use @distributed to parallelize a loop across workers.
# Sum of squareis using @distributed
###

# result = @distributed (+) for i in 1:1_000_000
#     i^2
# end

# println("Sum of squares: ", result)
```

```
###
# Remote Execution
# Use @spawn to run a task asynchronously on a worker.
###

# Run a task on a worker
# future = @spawn begin
#     sum(i^2 for i in 1:1_000_000)
# end

# # Fetch the result
# result = fetch(future)
# println("Sum of squares: ", result)
```

```
###
# Share Data Between Processes
# Use RemoteChannel to share data between processes.
###

# Create a remote channel
channel = RemoteChannel(()->Channel{Int}(10))

# # Put data into the channel on worker 2
# @spawnat 2 begin
#     for i in 1:10
#         put(channel, i)
#     end
# end

# # Fetch data from the channel on the master process
# for i in 1:10
#     println("Received: ", take(channel))
# end
```

```
RemoteChannel{Channel{Int64}}(1, 1, 1)
```

Mock (or Real) Example

Example: Distributed DataFrames

Here's an example of distributing a DataFrame across workers and processing it in parallel.

```
# using Distributed, DataFrames

# # Add worker processes
# addprocs(4)

# # Load DataFrames on all workers
# @everywhere using DataFrames, Statistics

# # Create a large DataFrame
# n = 1_000_000
# students = DataFrame(
#     ID = 1:n,
#     Name = rand(["Alice", "Bob", "Charlie", "David", "Eve"], n),
#     Age = rand(14:18, n),
#     Score = rand(50:100, n),
#     Grade = rand(9:12, n)
# )

# # Split the DataFrame into chunks
```

```
# chunks = [students[i:min(i + 250_000 - 1, n), :] for i in 1:250_000:n]

# # Distribute chunks to workers
# @everywhere workers() begin
#     global my_chunk = chunks[myid() - 1]  # Assign a chunk to each worker
# end

# # Define a function to calculate average score by grade
# @everywhere function average_score_by_grade(df)
#     return combine(groupby(df, :Grade), :Score => mean => :Average_Score)
# end

# # Compute results in parallel
# results = @distributed (vcat) for i in workers()
#     average_score_by_grade(my_chunk)
# end

# # Combine results (average across all chunks)
# final_result = combine(groupby(results, :Grade), :Average_Score => mean => :Final_Average_Score)

# # Display the final result
# println("Final Average Score by Grade:")
# println(final_result)
```

NOTE:

`Distributed.jl` makes it easy to parallelize computations.

- **Parallel Loops:** Use @distributed to parallelize loops.
- **Task Parallelism:** Use @spawn to run tasks asynchronously.
- **Data Parallelism:** Distribute large datasets across workers.
- **Cluster Computing:** Scale computations across multiple machines.

6.4 Mention Out-of-Core Computing with Dagger.jl

What is Dagger.jl? - A framework for out-of-core and parallel computation. - Works with large datasets by processing them in chunks.

Dagger.jl is a Julia package designed for out-of-core and parallel computation. It allows you to work with datasets that are too large to fit into memory by breaking them into smaller chunks and processing them in parallel. Dagger is particularly useful for big data workflows, where traditional in-memory approaches are not feasible.

```
# Create a large array (1 billion elements)

# large_array = Dagger.ones(1_000_000_000)
```

```
# Perform Out-of-Core Computations
# Dagger automatically breaks the array into chunks and processes them in parallel.

# Compute the sum of the array
# sum_result = sum(large_array)
# println("Sum of large array: ", sum_result)
```

```
# Parallel Map-Reduce
# Dagger supports parallel map-reduce operations, which are useful for big data workflows.

# Map: Square each element
# Reduce: Sum the squared elements

# result = Dagger.@par mapreduce(i -> i^2, +, large_array)
# println("Sum of squares: ", result)
```

```
# Integration with DataFrames.jl
# Dagger can also work with tabular data. Here's an example of processing a large DataFrame:

# using DataFrames

# # Create a large DataFrame
# n = 1_000_000
# large_df = DataFrame(
#     ID = 1:n,
#     Value = rand(1:100, n)
# )

# # Convert the DataFrame to a Dagger table
# dagger_table = Dagger.Table(large_df)

# # Compute the mean of the "Value" column
# mean_value = Dagger.@par mean(dagger_table.Value)
# println("Mean value: ", mean_value)
```

Mock (or Real) Example

Example: Out-of-Core Computation

Here's a complete example of using Dagger to process a large dataset:

```
# using Dagger, DataFrames

# # Create a large DataFrame
# n = 1_000_000
# large_df = DataFrame(
#     ID = 1:n,
```

```
#     Value = rand(1:100, n)
# )

# # Convert the DataFrame to a Dagger table
# dagger_table = Dagger.Table(large_df)

# # Perform a parallel map-reduce operation
# result = Dagger.@par mapreduce(row -> row.Value^2, +, dagger_table)
# println("Sum of squared values: ", result)
```

Integration with DataFrames

- Use Dagger to process large DataFrames in chunks.

Key Point

- Dagger.jl enables you to work with datasets larger than memory.

6.5. Mock (or Real) World Example

Scenario: *Analyze a 10GB dataset of student records.*

NOTE

1. Distributed Computing: Use Distributed.jl to parallelize computations.
2. Out-of-Core Computing: Use Dagger.jl to process large datasets in chunks.
3. Integration: Julia's tools work seamlessly together for big data workflows.

Useful For

- **Big Data Workflows:** Process datasets larger than memory by working with chunks.
- **Parallel Computing:** Automatically parallelize computations across CPU cores.
- **Lazy Evaluation:** Optimize task execution with deferred computation.
- **Integration:** Combine Dagger with DataFrames.jl and Distributed.jl for scalable data analysis.

Key Point

- Julia is a powerful tool for big data analysis, from small datasets to terabytes of data.

```
# using Distributed, Dagger, CSV, DataFrames

# # Step 1: Add worker processes
# addprocs(4)

# # Step 2: Read the dataset in chunks
# df = CSV.read("large_students.csv", DataFrame; chunksize=100_000)

# # Step 3: Process chunks in parallel
# @distributed for chunk in df
#     # Perform analysis on each chunk
```

```
# end

# # Step 4: Combine results
# results = fetch(@distributed (+) for chunk in df
#     sum(chunk.Score)
# end)

# println("Total Score: ", results)
```

Fin!

A Data Project's Logical Flow in Julia and Elsewhere:
- Start simple (DataFrames & DataFramesMeta) → Build complexity (CSV & Arrow) → Scale up to Big Data (Distributed & Dagger).

References

1. JuliaData: Data manipulation, storage, and I/O in Julia
2. Julia Data Science: Data Science using Julia
3. Julia for Data Analysis
4. Wikipedia on Data
5. Wikipedia on Data Literacy