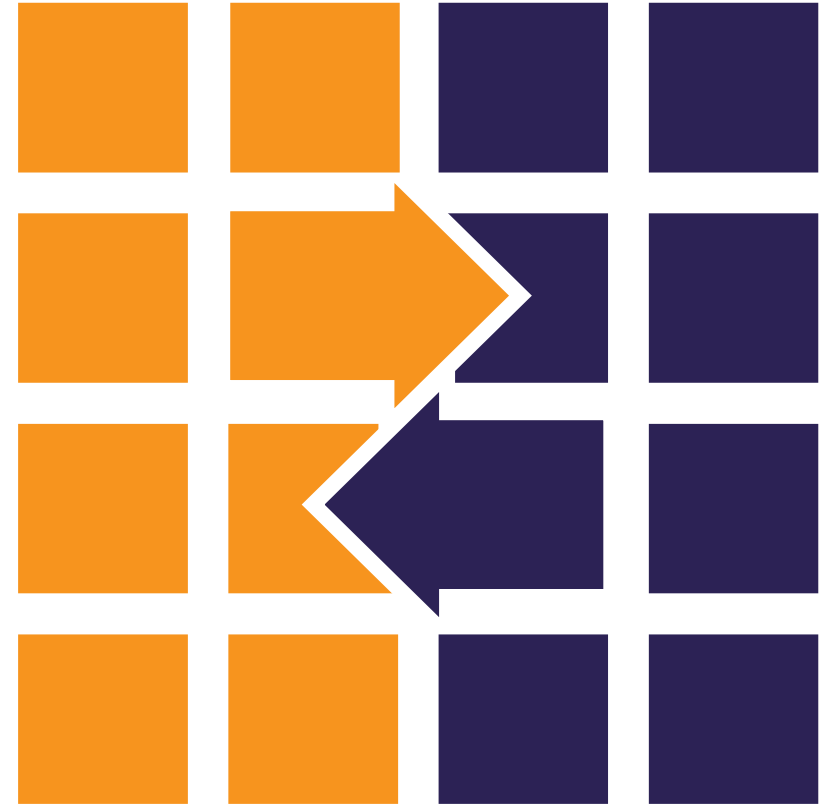# Are You Missing a DataFrame?

The Power of Data Frames in Java

Data Oriented Programming

> " *Data-oriented programming encourages us to model data as (immutable) data, and keep the code that embodies the business logic of how we act on that data separately.*

> — *Brian Goetz*

> *"Data Oriented Programming in Java"*

https://www.infoq.com/articles/data-oriented-programming-java/

Data Oriented vs. Object Oriented

Data/State

- Take my state. Please.
  - Just data – as records, tables, collections
  - Data is immutable

- State? What state?
  - Protect your privates
  - Data is encapsulated
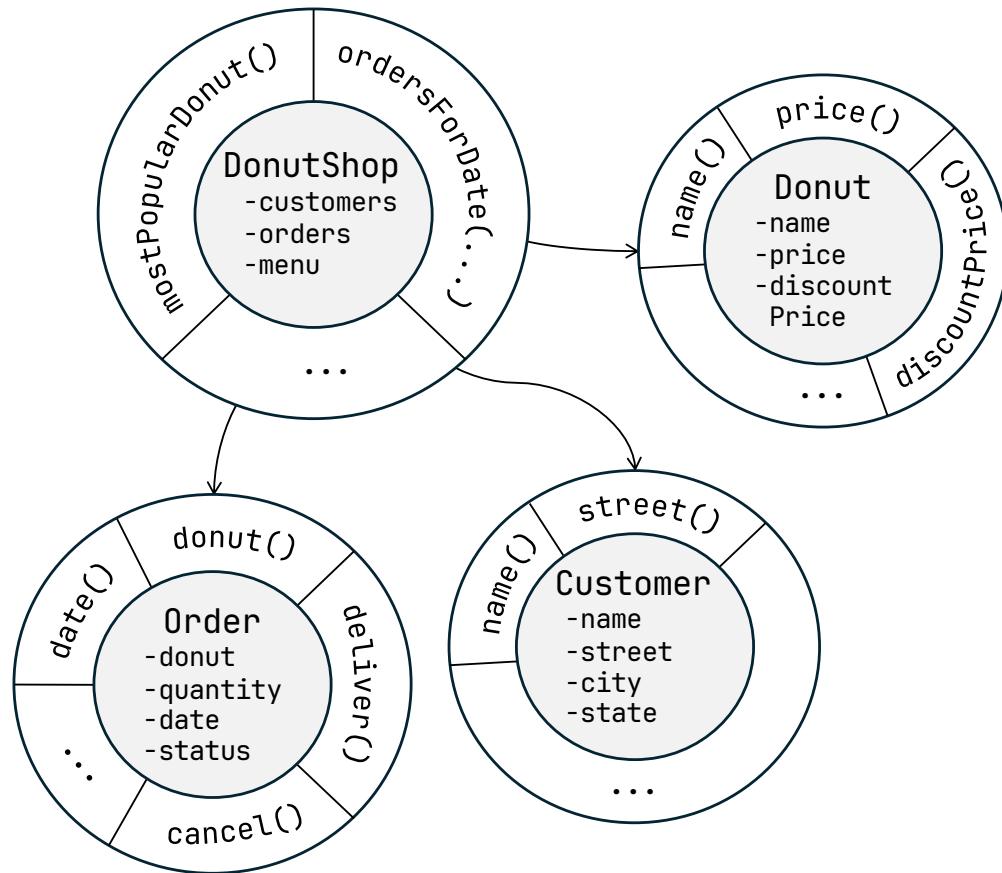
Operations

- Standalone functions operating on data
  - Methods are an exception (e.g., very simple behavior on data elements)

- Objects responding to messages
  - Implemented as methods
- Polymorphism allows objects of different types to respond to the same message

What is it good for?

- Experimentation, exploration, ad hoc calculations
- Simple functions operating on simple data
- Separating business logic from data
  - E.g., business user facing applications, logic needs to be user defined and externalized
- Flexible data models

- Managing complexity of domain structure and behavior
- Managing change
- Taking advantage of OO language features (polymorphism, inheritance)
- Taking advantage of OO modeling techniques

Object Oriented vs. Data Oriented Donuts

VS.

Customers

Name	Street	City	State
Alice	902 S Pacific St	Las Vegas	NM
Bob	405 Main St	Dallas	SD
Carol	12300 State St	Atlanta	MI

Donuts

Name	Price	Discount Price
Old Fashioned	$1.25	$1.00
Blueberry	$1.50	$1.25
Apple Cider	$1.00	$0.90

Orders

Client	Donut	Quantity	Date
Alice	Old Fashioned	12	2024-10-10
Bob	Blueberry	6	2024-10-11
Carol	Apple Cider	6	2024-10-11
Alice	Old Fashioned	4	2024-10-12

Object Oriented vs. Data Oriented Donuts

```java
public List<Donut>
topThreeBestSellers(List<Order> orders)
{
    return orders.stream()
        .collect(
            Collectors.groupingBy(
                Order::donut,
                Collectors.summingInt(Order::quantity))
            )
        .entrySet()
        .stream()
        .sorted(Map.Entry.<~>comparingByValue().reversed())
        .limit(3)
        .map(Map.Entry::getKey)
        .toList();
}
```

Customers

Name	Street	City	State
Alice	902 S Pacific St	Las Vegas	NM
Bob	405 Main St	Dallas	SD
Carol	12300 State St	Atlanta	MI

Donuts

Name	Price	Discount Price
Old Fashioned	$1.25	$1.00
Blueberry	$1.50	$1.25
Apple Cider	$1.00	$0.90

Orders

Client	Donut	Quantity	Date
Alice	Old Fashioned	12	2024-10-10
Bob	Blueberry	6	2024-10-11
Carol	Apple Cider	6	2024-10-11
Alice	Old Fashioned	4	2024-10-12

The One Billion Row Challenge (1BRC)

> **"** *Your mission, should you decide to accept it, is deceptively simple: write a Java program for retrieving temperature measurement values from a text file and calculating the min, mean, and max temperature per weather station. There's just one caveat: the file has 1,000,000,000 rows!*
>
> — *Gunnar Morling (@gunnarmorling)*
> *"The One Billion Row Challenge"*

WOULD YOU LIKE TO KNOW MORE?

Announcement   https://www.morling.dev/blog/one-billion-row-challenge/

Details and Results   https://github.com/gunnarmorling/1brc

A classic Data Oriented Programming problem!

1BRC Results

- Pretty amazing!
  - Top 3 results: 1.535, 1.587, and 1.608 seconds
  - Reference environment: 8 cores, 128 GB RAM

- Most submissions
  - Are many hundreds of lines of (well formatted and well factored) code
  - Have fairly helpful comments (well, some)
  - Use Vector API, low level APIs, the latest language features, the latest JVM features, and dark magic
  - Require intimate understanding of the JVM and the compiler behavior from the developers
  - Took a good amount of the developer's time (spent both writing and reading the code).
  - Do not make it obvious how the algorithms work/what exactly they do due to the solution complexity

This makes sense in the context of 1BRC and doesn't make these solutions "bad code".

Achieving absolute peak performance demands the above "sacrifices".

1BRC – But Optimized for Developers

What if?

instead of optimizing for peak performance we optimized for

- code readability
- software maintainability
- developer time/effort

New requirements!

- The cycles spent developing, understanding, and maintaining this code by humans matter more than achieving the absolute minimum of CPU cycles
- Deliver a working solution quickly and then optimize it if and when needed

---

Enter DataFrame

What Is a DataFrame?

- A DataFrame is tabular data set that can be manipulated programmatically

- It is made up of columns of different types, similar to a relational table

- Tabular data can be loaded as a data frame from

  - a file

  - a database result set

  - anything else that looks like a table

  - anything that can be made look like a table (e.g., a projection of an object graph)

- A data frame can be created programmatically

  - by specifying its values

  - by transforming the existing data frames

Why Is a DataFrame?

- Provides the ability to group data and easily transform and organize data in our code

- Provides the benefits of developer efficiency, flexibility, and code readability

- Leverages the efficiency of collection frameworks

- Can offer memory savings and better performance than the alternative approaches

- Is used in real-world scenarios
  - data transformation
  - data enrichment
  - data validation
  - reconciliation

1BRC With Toy Data

measurements.txt

```
New York City;34.1
New York City;24.3
San Francisco;22.9
Istanbul;5.9
New York City;-2.7
Istanbul;15.0
San Francisco;-5.4
Istanbul;13.2
San Francisco;35.0
Tauranga;17.4
```

Steps

1. Load the data from the file.

2. Perform aggregation and sorting.

3. Show results on the console.

1BRC with Dataframe-EC: Load

```java
CsvSchema msSchema = new CsvSchema()
        .addColumn("Station", STRING)
        .addColumn("Temperature", FLOAT)
        .separator(';')
        .hasHeaderLine(false);

URI measurementFile =
    CalculateAverage.class
        .getClassLoader()
        .getResource(MEASUREMENT_FILE)
        .toURI();

CsvDataSet msDataSet = new CsvDataSet(
    Path.of(measurementFile), "measurements", msSchema);

DataFrame measurements = msDataSet.loadAsDataFrame();
```
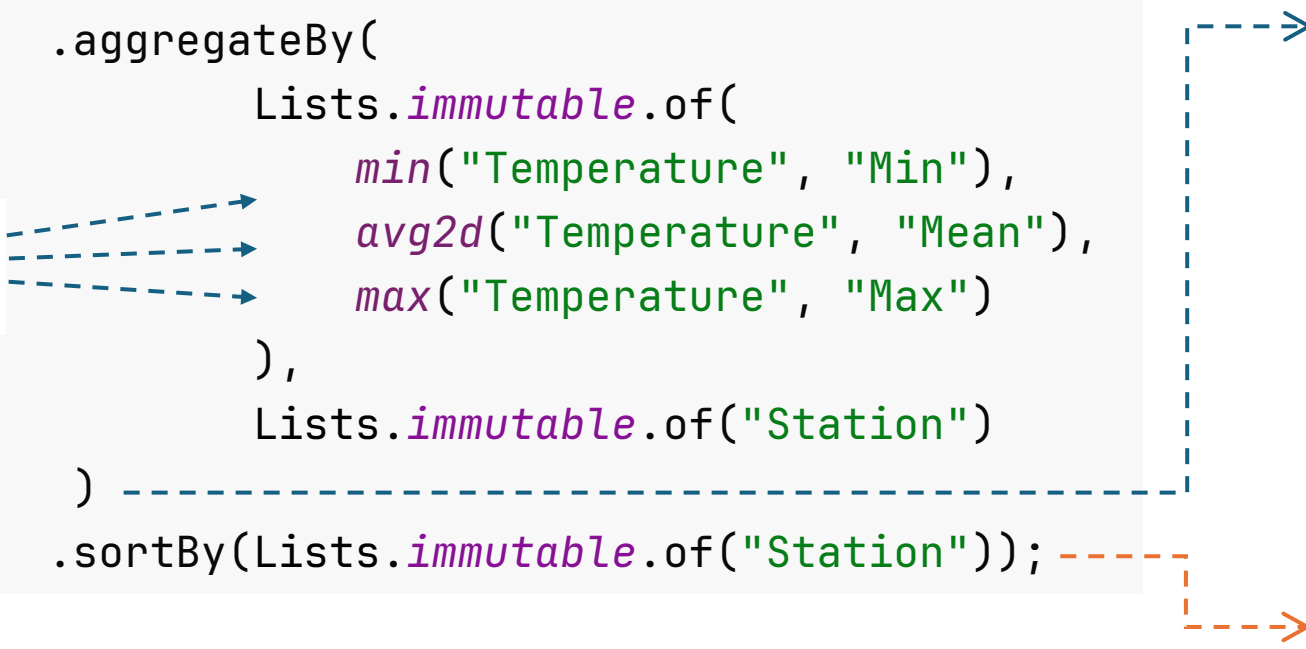
measurements

Station	Temperature
New York City	34.1
New York City	24.3
San Francisco	22.9
Istanbul	5.9
New York City	-2.7
Istanbul	15
San Francisco	-5.4
Istanbul	13.2
San Francisco	35
Tauranga	17.4

1BRC with Dataframe-EC: Process

```
DataFrame aggregated = measurements
    .aggregateBy(
        Lists.immutable.of(
            min("Temperature", "Min"),
            avg2d("Temperature", "Mean"),
            max("Temperature", "Max")
        ),
        Lists.immutable.of("Station")
    )
    .sortBy(Lists.immutable.of("Station"));
```

static imports

Station	Min	Mean	Max
New York City	-2.7	18.56667	34.1
San Francisco	-5.4	17.5	35
Istanbul	5.9	11.36667	15
Tauranga	17.4	17.4	17.4

Station	Min	Mean	Max
Istanbul	5.9	11.36667	15
New York City	-2.7	18.56667	34.1
San Francisco	-5.4	17.5	35
Tauranga	17.4	17.4	17.4

1BRC with Dataframe-EC: Output

```java
aggregated.forEach(row ->
        System.out.printf(
                "%s=%2.1f/%2.1f/%2.1f\n",
                row.getString("Station"),
                row.getFloat("Min"),
                row.getDouble("Mean"),
                row.getFloat("Max"))
);
```

Output:

```
Istanbul=5.9/11.4/15.0
New York City=-2.7/18.6/34.1
San Francisco=-5.4/17.5/35.0
Tauranga=17.4/17.4/17.4
```

1BRC with Tablesaw: Load

```java
URL measurementFile = CalculateAverage.class.getClassLoader()
                            .getResource(MEASUREMENT_FILE);

CsvReadOptions options = CsvReadOptions
        .builder(measurementFile)
        .columnTypes(new ColumnType[] {STRING, FLOAT})
        .separator(';')
        .header(false)
        .build();

Table measurements = Table.read().usingOptions(options);

measurements.column(0).setName("Station");
measurements.column(1).setName("Temperature");
```

1BRC with Tablesaw: Process

static imports

```
Table aggregated = measurements
        .summarize("Temperature", min, mean, max)
        .by("Station")
        .sortOn("Station");
```

1BRC with Tablesaw: Output

```
aggregated.forEach(row ->
        System.out.printf(
                "%s=%2.1f/%2.1f/%2.1f\n",
                row.getString("Station"),
                row.getDouble("Min [Temperature]"),
                row.getDouble("Mean [Temperature]"),
                row.getDouble("Max [Temperature]"))
);
```

About dataframe-ec

- Based on the Eclipse Collections framework (the "-ec" in the name)
- Memory efficient (for practical use cases)
  - Uses highly memory efficient Eclipse Collections
  - Takes advantage of its support for primitive types
- Inspired by Eclipse Collections APIs
- Exposes Eclipse Collections types in its APIs
- Intuitive, humane grammar for the expression DSL used for computed columns, filters, etc.
  - e.g., adding two numbers is expressed as "A + B", as opposed to an internal DSL where you assemble an expression from Java method calls
- Ability to add the expression DSL functions and aggregation functions without touching the core framework

Donut Store Example: Data

Customers

Name	Street	City	State
Alice	902 S Pacific St	Las Vegas	NM
Bob	405 Main St	Dallas	SD
Carol	12300 State St	Atlanta	MI
Dave	102 S Main St	Phoenix	OR

Donuts

Description	Price	Price/Dozen
Blueberry	$1.25	$1.00
Old Fashioned	$1.00	$0.90
Pumpkin Spice	$0.75	$0.65
Jelly	$1.50	$1.25
Apple Cider	$1.50	$1.25

Orders

Customer	DeliveryDate	Donut	Quantity
Alice	2024-05-12	Old Fashioned	12
Alice	2024-05-12	Blueberry	2
Bob	2024-05-12	Old Fashioned	12
Alice	2024-05-13	Apple Cider	12
Alice	2024-05-13	Blueberry	2
Carol	2024-05-13	Old Fashioned	12
Dave	2024-05-14	Old Fashioned	12
Alice	2024-05-14	Jelly	12
Alice	2024-05-14	Blueberry	2
Bob	2024-05-14	Pumpkin Spice	1

Donut Store Example: Use Cases

- List Donuts in Popularity Order
- Priority Orders for Tomorrow
  - Large orders
  - Bob's orders
- Total Spend per Customer
- Donut Count per Customer Per Day

List Donuts in Popularity Order

DataFrame-EC

```
DataFrame donutsInPopularityOrder =
    this.orders
        .aggregateBy(
            Lists.immutable.of(sum("Quantity")),
            Lists.immutable.of("Donut"))
        .sortBy(
            Lists.immutable.of("Quantity", "Donut"),
            Lists.immutable.of(DESC, ASC))
        .keepColumns(Lists.immutable.of("Donut"));
```

Donut	Quantity
Old Fashioned	48
Blueberry	6
Apple Cider	12
Jelly	12
Pumpkin Spice	1

Donut	Quantity
Old Fashioned	48
Apple Cider	12
Jelly	12
Blueberry	12
Pumpkin Spice	1

Donut
Old Fashioned
Apple Cider
Jelly
Blueberry
Pumpkin Spice

List Donuts in Popularity Order
Tablesaw

Donut	Quantity
Old Fashioned	48
Blueberry	6
Apple Cider	12
Jelly	12
Pumpkin Spice	1

```
Table donutsInPopularityOrder = this.orders
     .summarize("Quantity", sum)
     .by("Donut")
     .sortOn("-Sum [Quantity]", "Donut")
     .retainColumns("Donut");
```

Donut	Quantity
Old Fashioned	48
Apple Cider	12
Jelly	12
Blueberry	12
Pumpkin Spice	1

Donut
Old Fashioned
Apple Cider
Jelly
Blueberry
Pumpkin Spice

Priority Orders for Tomorrow

DataFrame-EC

```
DataFrame priorityOrdersTomorrow = this.orders
    .selectBy(
        "DeliveryDate == toDate('%s') and (Quantity >= 12 or Customer == 'Bob')"
        .formatted(TOMORROW)
    );
```

priorityOrdersTomorrow =

Customer	DeliveryDate	Donut	Quantity
Dave	2024-05-14	Old Fashioned	12
Alice	2024-05-14	Jelly	12
Bob	2024-05-14	Pumpkin Spice	1

Priority Orders for Tomorrow

Tablesaw

```java
Table priorityOrdersTomorrow = this.orders.where(
    and(
        t -> t.dateColumn("DeliveryDate").isEqualTo(TOMORROW),
            or(
                t -> t.longColumn("Quantity").isGreaterThanOrEqualTo(12),
                t -> t.stringColumn("Customer").isEqualTo("Bob")
            )
        )
    )
);
```

Total Spend Per Customer

Join order and donut data sets (DataFrame-EC)

```java
this.orders.lookupIn(this.menu)
        .match("Donut", "Donut")
        .select(Lists.immutable.of("Price", "DiscountPrice"))
        .resolveLookup();
```

Customer	DeliveryDate	Donut	Quantity	Price	DiscountPrice
Alice	2024-05-22	Old Fashioned	12	1.0000	0.9000
Alice	2024-05-22	Blueberry	2	1.2500	1.0000
Bob	2024-05-22	Old Fashioned	12	1.0000	0.9000
Alice	2024-05-23	Apple Cider	12	1.5000	1.2500
Alice	2024-05-23	Blueberry	2	1.2500	1.0000
Carol	2024-05-23	Old Fashioned	12	1.0000	0.9000
Dave	2024-05-24	Old Fashioned	12	1.0000	
Alice	2024-05-24	Jelly			

Total Spend Per Customer

Add Order Price column (DataFrame-EC)

```java
this.orders.lookupIn(this.menu)
        .match("Donut", "Donut")
        .select(Lists.immutable.of("Price", "DiscountPrice"))
        .resolveLookup();

this.orders.addColumn("OrderPrice",
    "(Quantity < 12 ? Price : DiscountPrice) * Quantity");
```

Customer	DeliveryDate	Donut	Quantity	Price	DiscountPrice	OrderPrice
Alice	2024-05-22	Old Fashioned	12	1.0000	0.9000	10.8000
Alice	2024-05-22	Blueberry	2	1.2500	1.0000	2.5000
Bob	2024-05-22	Old Fashioned	12	1.0000	0.9000	10.8000
Alice	2024-05-23	Apple Cider	12	1.5000	1.2500	15.0000
Alice	2024-05-23	Blueberry	2	1.2500	1.0000	2.5000
Carol	2024-05-23	Old Fashioned				

Total Spend Per Customer

Aggregate and sort (DataFrame-EC)

```
this.orders.lookupIn(this.menu)
        .match("Donut", "Donut")
        .select(Lists.immutable.of("Price", "DiscountPrice"))
        .resolveLookup();


this.orders.addColumn(
    "OrderPrice",
    "(Quantity < 12 ? Price : DiscountPrice) * Quantity");


DataFrame spendPerCustomer =
    this.orders
        .aggregateBy(
            Lists.immutable.of(sum("OrderPrice", "Total Spend")),
            Lists.immutable.of("Customer"))
        .sortBy(Lists.immutable.of("Customer"));
```

Customer	Total Spend
Alice	45.8000
Bob	11.5500
Carol	13.3000
Dave	10.8000

Donut Count Per Customer Per Day

DataFrame-EC

```
DataFrame donutsInPopularityOrder =
    this.orders
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Conclusions

- Bleep
- Bloop
- Meep
- Moop

Data frame libraries:

Dataframe-EC   https://github.com/vmzakharov/dataframe-ec

Tablesaw   https://github.com/jtablesaw/tablesaw

DFLib   https://github.com/dflib/dflib