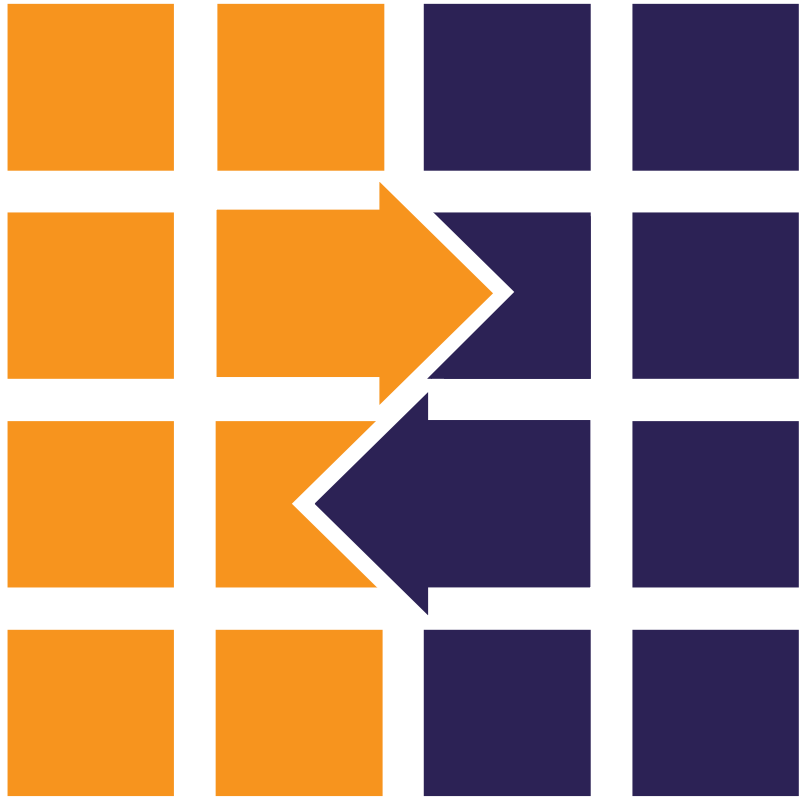
Are You Missing a DataFrame?

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/

See also the DOP article series by Nicolai Parlog:

https://inside.java/2024/05/23/dop-v1-1-introduction/

Object-Oriented vs. Data-Oriented

Data/State

- Protect your privates
- Everything is ~~awesome~~ encapsulated
- There is no ~~spoon~~ state

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

Operations

- Objects responding to messages
- Polymorphism (allows objects of different types to respond to the same message)

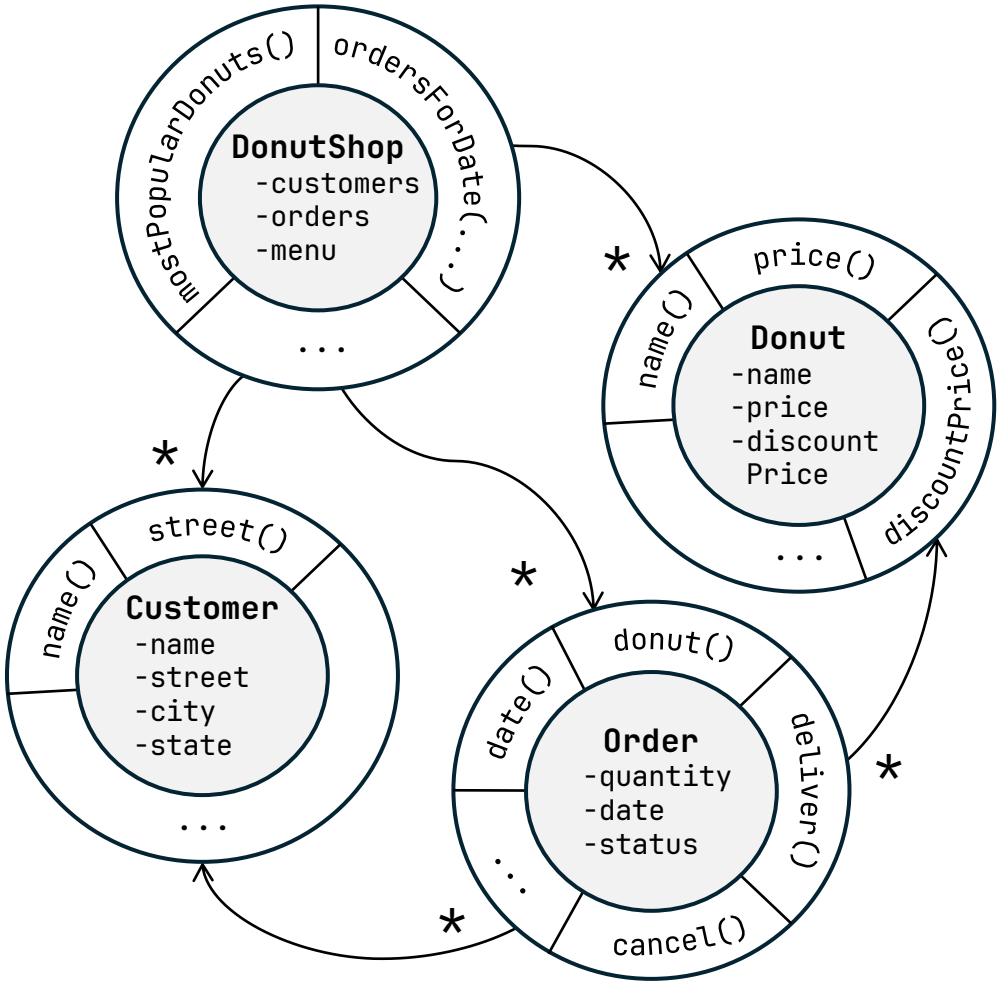
- Standalone functions operating on data

What Is It Good For?

- Managing complex structures and behaviors
- Separation of concerns
- Taking advantage of OO language features
- Applying OO modeling techniques

- Data exploration, ad hoc calculations
- Simple functions operating on simple data
- Separation of business logic from data
- Flexible data models

Object Oriented vs. Data Oriented Donuts



VS.

Menu

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

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

Orders

Customer	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

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

Menu

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

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

Orders

Customer	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

Data Oriented Programming in Java

- Records, sealed classes, and pattern matching
  - Records model data aggregates and take advantage of static typing
  - Sealed classes – prevent illegal states
  - Pattern matching for `instanceof` and `switch`
- Collections/streams/maps
- Record drawbacks
  - Memory overhead
    - For example, object header + object alignment, collected using JOL: `Customer` – 16 bytes, `Donut` – 12 bytes, `Order` – 16 bytes
  - Stream code readability is not always amazing
  - Brittle-ish when it comes to refactoring
- Map Oriented Programming [in movie quotes]
  - Memory overhead—"*It's large. Large. Large. So large*"
  - Readability and maintainability—"*The horror… the horror…*"

What Is a DataFrame?

- A tabular data set that can be manipulated programmatically
- Made up of columns of different types, similar to a relational table
- Can be created
  - from tabular data
    - a csv 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)
  - programmatically
    - by specifying its values
    - by transforming the existing data frames

Why Is a DataFrame?

- Provides the ability to easily group, transform, and organize data in our code
- Offers the benefits of developer efficiency, flexibility, and code readability
- Leverages the efficiency of underlying 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/data quality
  - reconciliation

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
  - Use Vector API, low level APIs, the latest language features, the latest JVM features, and dark magic
  - Do not make it obvious how the algorithms work
  - Do not make it obvious what the code does functionally
  - Require intimate understanding of the JVM/compiler behavior from the developers
  - Took a good amount of the developer's time (spent both writing and reading the code)

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

Achieving absolute peak performance requires 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

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 Python/Pandas: Load

```
df = pd.read_csv(
    FILE,
    sep=";",
    header=None,
    names=["Station", "Temperature"],
    engine='pyarrow’
)
```

	Station	Temperature
0	New York City	34.1
1	New York City	24.3
2	San Francisco	22.9
3	Istanbul	5.9
4	New York City	-2.7
5	Istanbul	15
6	San Francisco	-5.4
7	Istanbul	13.2
8	San Francisco	35
9	Tauranga	17.4

1BRC With Python/Pandas: Process

```
df = (df.groupby("Station", sort=False)
      .agg(["min", "mean", "max"]))

df = df.sort_values("Station")
```

	Temperature		
Station	**min**	**mean**	**max**
New York City	-2.7	18.56667	34.1
San Francisco	-5.4	17.50000	35.0
Istanbul	5.9	11.36667	15.0
Tauranga	17.4	17.40000	17.4

OR

```
df = df.groupby("station").agg(["min", "max", "mean"])
```

	Temperature		
Station	**min**	**mean**	**max**
Istanbul	5.9	11.36667	15.0
New York City	-2.7	18.56667	34.1
San Francisco	-5.4	17.50000	35.0
Tauranga	17.4	17.40000	17.4

NOTE: `groupby()` sorts by default, to override this behavior use the `sort=False` parameter and then make an explicit call to `sort_values()` if needed

1BRC With Python/Pandas: Output

Station	min	mean	max
New York City	-2.7	18.56667	34.1
San Francisco	-5.4	17.50000	35.0
Istanbul	5.9	11.36667	15.0
Tauranga	17.4	17.40000	17.4

```
df.columns = df.columns.droplevel()

for index, row in df.iterrows():
    print(f"{index}={row['min']:0.1f}/{row['mean']:0.1f}/{row['max']:0.1f}")
```

```
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 Dataframe-EC: Load

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

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

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

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

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"));
```

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

```
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"))
);
```

```
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

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

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 Tablesaw: Process

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

Station	Min [Temperature]	Mean [Temperature]	Max [Temperature]
New York City	-2.70000	18.56667	34.10000
San Francisco	-5.40000	17.50000	35
Istanbul	5.90000	11.36667	15
Tauranga	17.40000	17.40000	17.40000

Station	Min [Temperature]	Mean [Temperature]	Max [Temperature]
Istanbul	5.90000	11.36667	15
New York City	-2.70000	18.56667	34.10000
San Francisco	-5.40000	17.50000	35
Tauranga	17.40000	17.40000	17.40000

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]"))
);
```

```
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 Kotlin: Load

```
val measurementFile = object {}::class.java.classLoader.getResource(FILE)

val measurements = DataFrame.readCSV(
    measurementFile!!,
    header = listOf("Station", "Temperature"),
    delimiter = ';'
);
```

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 Kotlin: Process

```
val aggregated = measurements.groupBy("Station")
    .aggregate {
        min("Temperature") into "min"
        mean("Temperature") into "mean"
        max("Temperature") into "max"
    }
    .sortBy("Station")
```

Station	group
New York City	[3 x 2]
San Francisco	[3 x 2]
Istanbul	[3 x 2]
Tauranga	[1 x 2]

Station	Temperature
New York City	34.1
New York City	24.3
New York City	-2.7

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

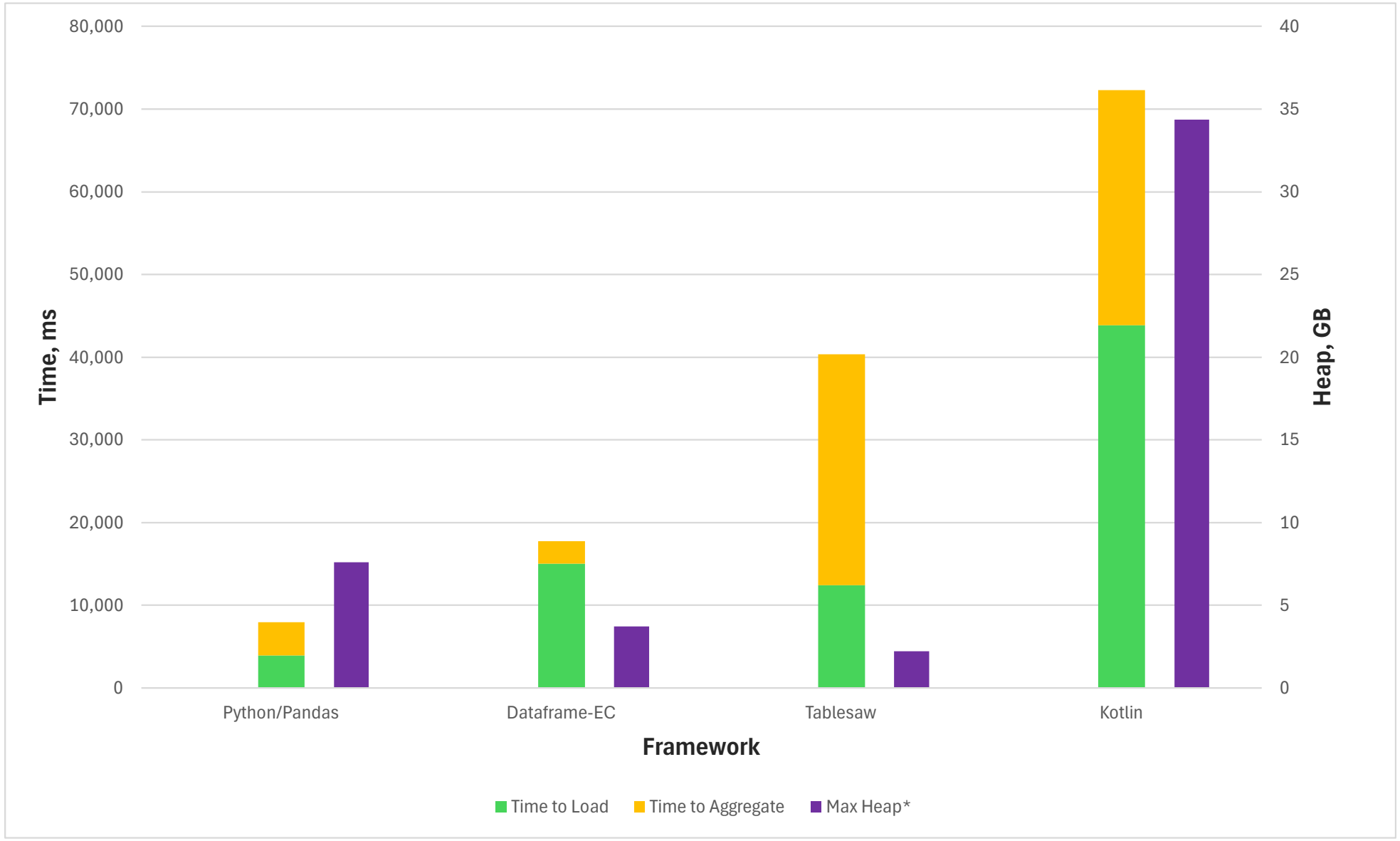
Station	Min	Mean	Max
Istanbul	5.9	11.36667	15
New York City	-2.7	18.56667	34.1
San Francisco	-5.4	17.5	35
Tauranga	17.4	17.4	17.4

1BRC With Kotlin: Output

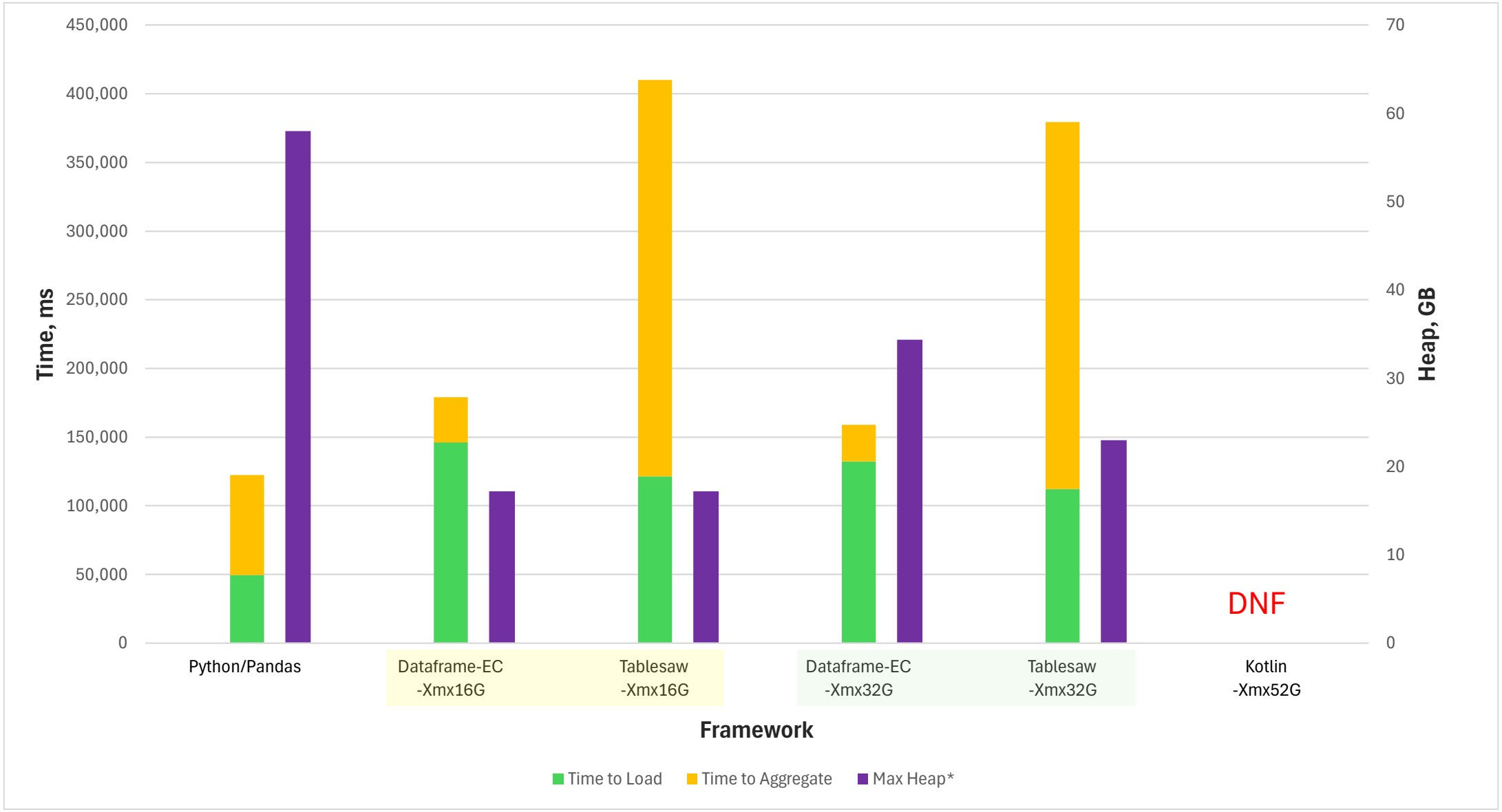
```
aggregated.forEach {
    println(
        "%s=%2.1f/%2.1f/%2.1f".format(it["Station"], it["min"], it["mean"], it["max"])
    )
}
```

```
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
```

Performance: 100MRC



Performance: 1BRC



About dataframe-ec

- Intuitive, simple grammar for the expression DSL used for computed columns, filters, etc.
- Ability to add the expression DSL functions and aggregation functions without touching the core framework
- Dealing with nulls: acceptance, high tolerance, flexibility
- Built 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

Eclipse Collections – What Is It Good For?

- Rich functional API with good symmetry (*productivity*) ✓
- Memory Efficiency (*performance*) ✓
- Optimized Eager API (*performance*) ✓
- Primitive Collections (*productivity* + *performance*) ✓
- Immutable Collections (*safety* + *performance*) ✓
- Lazy and Parallel Lazy API (*performance*) ✓
- Multimap, Bag, BiMap, Pool, Interval (*productivity* + *type safety*) ✓
- Hashing Strategy Data Structures (*productivity* + *performance*)
- Multi-reader data structures (*safety* + *performance*)
- Mutable and Immutable Collection Factories (*productivity*) ✓

Solid Foundation: Eclipse Collections Types

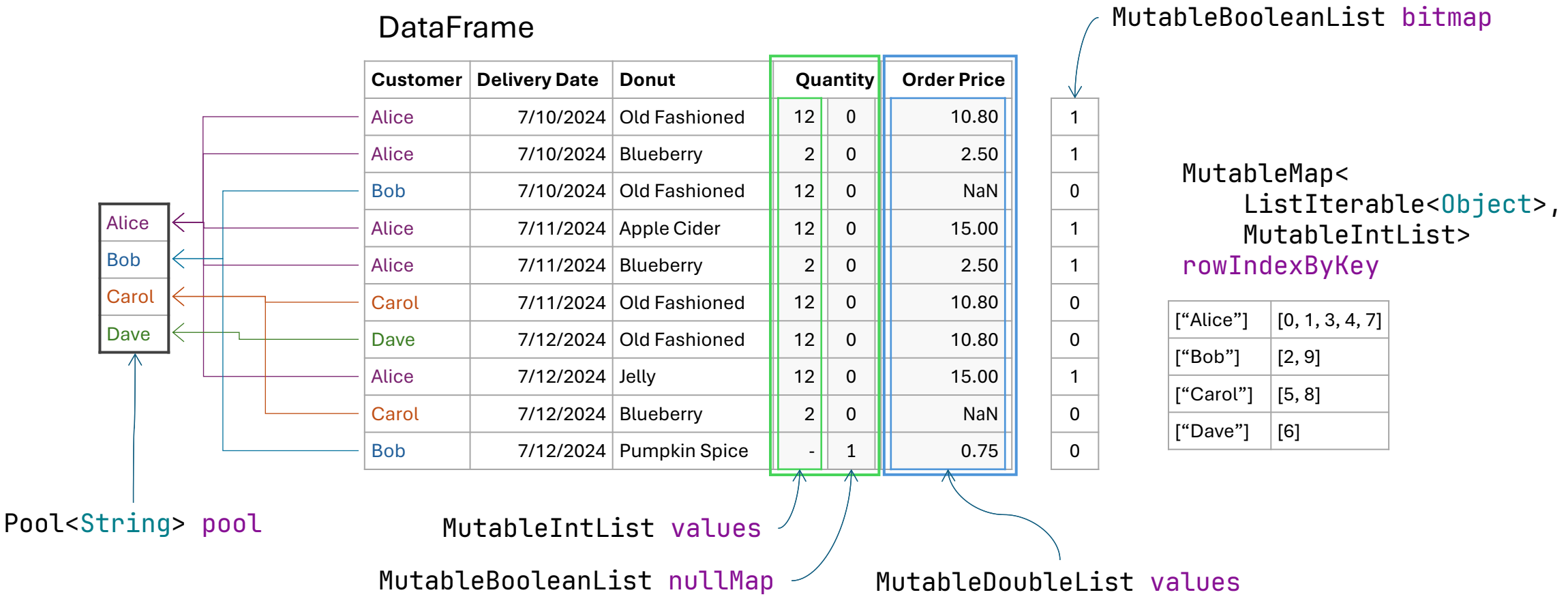
…the foundation, and walls, and joints, and studs, and…

DataFrame

Customer	Delivery Date	Donut	Quantity	Order Price
Alice	7/10/2024	Old Fashioned	12	10.80
Alice	7/10/2024	Blueberry	2	2.50
Bob	7/10/2024	Old Fashioned	12	null
Alice	7/11/2024	Apple Cider	12	15.00
Alice	7/11/2024	Blueberry	2	2.50
Carol	7/11/2024	Old Fashioned	12	10.80
Dave	7/12/2024	Old Fashioned	12	10.80
Alice	7/12/2024	Jelly	12	15.00
Carol	7/12/2024	Blueberry	2	null
Bob	7/12/2024	Pumpkin Spice	null	0.75

Solid Foundation: Eclipse Collections Types

…the foundation, and walls, and joints, and studs, and…



Solid Inspiration: Eclipse Collections APIs

Eclipse Collections Patterns

```
this.donutOrders.partition(
  order -> simpleOrder.quantity < 12
)
```

```
this.donutOrders.distinct(
  HashingStrategies.fromFunctions(
    Order::customer, Order::deliveryDate
  )
);
```

```
this.donutOrders.select(
  order -> order.customer().equals("Carol")
)
```

```
this.donutOrders.sortThis(Comparator
  .comparing(Order::customer)
  .thenComparingDouble(Order::orderPrice)
);
```

DataFrame-EC Patterns

```
this.donutOrders.partition("Quantity < 12")
```

```
this.donutOrders.distinct(
    Lists.immutable.of("Customer", "DeliveryDate")
)
```

```
this.donutOrders.selectBy("Customer == 'Carol'")
```

```
this.donutOrders.sortBy(
    Lists.immutable.of("Customer", "OrderPrice")
)
```

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

Menu

Donut	Price	DiscountPrice
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
Carol	2024-05-14	Blueberry	2
Bob	2024-05-14	Pumpkin Spice	1

Donut Store Example: Use Cases

1. List Donuts in the Popularity Order
2. Priority Orders for Tomorrow:

   Large orders (Quantity >= 12) or Bob's orders
3. Total Spend per Customer
4. 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	6
Pumpkin Spice	1

Donut
Old Fashioned
Apple Cider
Jelly
Blueberry
Pumpkin Spice

List Donuts in Popularity Order

Tablesaw

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

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

Donut	Sum [Quantity]
Old Fashioned	48
Apple Cider	12
Jelly	12
Blueberry	6
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)
    );
```

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

```
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")
        )
    )
);
```

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

Total Spend per Customer: Steps

- Calculate the dollar amount of each order:
  - Join Donuts data set to Orders data set to get donut prices
  - Compute order amounts:

$$Order\ Amount = \begin{cases} Regular\ Price \times Quantity, Quantity < 12 \\ Discount\ Price \times Quantity, Quantity \geq 12 \end{cases}$$

- Group by customer and add up order amounts
- Sort by customer

Total Spend per Customer

Join order and donut data sets

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

```
Table ordersWithPrices = this.orders
        .joinOn("Donut")
        .inner(this.menu);
```

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			

Total Spend per Customer

Add Order Price column (DataFrame-EC)

```
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	12	1.0000	0.9000	10.8000
Dave	2024-05-24	Old Fashioned	12	1.0000	0.9000	10.8000
Alice	2024-05-24	Jelly	12	1.5000		
[illegible]	2024-05-24	Blueberry				

Total Spend per Customer

Add Order Price column (Tablesaw, Option 1)

```
DoubleColumn orderPrice = DoubleColumn.create("OrderPrice");
ordersWithPrices.addColumns(orderPrice);
LongColumn quantity = ordersWithPrices.longColumn("Quantity");

orderPrice.set(quantity.isLessThan(12),
    ordersWithPrices.doubleColumn("Price").multiply(quantity)
);

orderPrice.set(quantity.isGreaterThanOrEqualTo(12),
    ordersWithPrices.doubleColumn("DiscountPrice").multiply(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	12	1.0000	0.9000	10.8000
Dave	2024-05-24	Old Fashioned	12	1.0000		
Alice	2024-05-24	Jelly				

Total Spend per Customer

Add Order Price column (Tablesaw, Option 2)

```
DoubleColumn orderPrice = DoubleColumn.create("OrderPrice");
ordersWithPrices.forEach(
        row -> {
            long quantity = row.getLong("Quantity");
            orderPrice.append(
                quantity < 12
                        ? quantity * row.getDouble("Price")
                        : quantity * row.getDouble("DiscountPrice")
            );
        }
);

ordersWithPrices.addColumns(orderPrice);
```

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	12	1.0000	0.9000	10.8000
Dave	2024-05-24	Old Fashioned	12	1.0000		
Alice	2024-05-24	Jelly				

Total Spend per Customer

Aggregate and sort

```
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

```
Table spendPerCustomer = ordersWithPrices
    .summarize("OrderPrice", sum)
    .by("Customer")
    .sortOn("Customer");
```

Customer	Sum [OrderPrice]
Alice	45.8000
Bob	11.5500
Carol	13.3000
Dave	10.8000

Donut Count per Customer per Day

```
DataFrame donutsPerCustomerPerDay =
    this.orders.pivot(
        Lists.immutable.of("Customer"),
        "DeliveryDate",
        Lists.immutable.of(sum("Quantity"))
    );
```

```
Table donutsPerCustomerPerDay =
    this.orders.pivot(
        "Customer",
        "DeliveryDate",
        "Quantity", sum
    );
```

Customer	2024-05-23	2024-05-24	2024-05-25
Alice	14	14	12
Bob	12	0	1
Carol	0	12	2
Dave	0	0	12

Customer	2024-05-23	2024-05-24	2024-05-25
Alice	14	14	12
Bob	12		1
Carol		12	2
Dave			12

Takeaways

- Java data frames are a useful addition to your data manipulation toolkit
  - Work well with Data-Oriented Programming Paradigm
  - Make filtering, aggregation, transformation, enrichment easy!
- Ad hoc manipulation of tabular data
  - The types of things you might want to use Excel for
- Programmatically transforming, querying, analyzing data in your application
  - Maybe you don’t need that Spark cluster after all
- Data Science – notebooks and visualization

The Last Slide

This Talk + Sources

https://github.com/vmzakharov/missing-dataframe-talk

Dataframe-EC

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

Eclipse Collections

https://github.com/eclipse/eclipse-collections

Other Java Dataframe Libraries

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

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

Appendix

Donut Store Examples: Kotlin

List Donuts in Popularity Order

Kotlin

```
val donutsInPopularityOrder = this.orders
    .groupBy("Donut")
    .sum("Quantity")
    .sortBy { it["Quantity"].desc() and it["Donut"] }
    .select("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	6
Pumpkin Spice	1

Donut
Old Fashioned
Apple Cider
Jelly
Blueberry
Pumpkin Spice

Priority Orders for Tomorrow

Kotlin

```
val priorityOrdersTomorrow = orders.filter {
    ("DeliveryDate"<LocalDate>() == TOMORROW)
    &&
    ("Quantity"<Int>() >= 12 || "Customer"<String>() == "Bob")
}
```

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

Total Spend per Customer

Kotlin

```
val spendPerCustomer = orders
    .join(menu, "Donut")
    .add("OrderPrice") {
        (if ("Quantity"<Int>() < 12) "Price"<Double>()
        else "DiscountPrice"<Double>()) * "Quantity"<Int>()
    }
    .select("Customer", "OrderPrice")
    .groupBy("Customer")
    .sum {"OrderPrice"<Double>() named "Total Spend"}
```

Customer	Total Spend
Alice	45.8000
Bob	11.5500
Carol	13.3000
Dave	10.8000

Donut Count per Customer per Day

Kotlin

```kotlin
val donutsPerCustomerPerDay =
    orders
        .pivot("DeliveryDate", inward = false)
        .groupBy("Customer")
        .aggregate { sum("Quantity") }
```

Customer	2024-05-23	2024-05-24	2024-05-25
Alice	14	14	12
Bob	12	null	1
Carol	null	12	2
Dave	null	null	12