WIZELINE

ACADEMY

Grow your career:
Free courses in Artificial Intelligence, Software Development, User Experience and More

Wi-Fi: Wizeline Academy
Password: academyGDL
Slack Channel: http://bit.ly/slackacademy

@WizelineAcademy

/WizelineAcademy

academy.wizeline.com

Get notified about courses:
tinyurl.com/WL-academy

Week 2 - Session 1/2

Big Data Engineering with Spark

Spark Basics

WIZELINE

In this session...

- Transformations
- Actions
- Caching

By the end of this session, you'll be able to explain:

- ***What are <u>Transformations</u> in Spark***
- ***What are <u>Actions</u> in Spark***
- ***How <u>Caching</u> works in Spark***

TRANSFORMATIONS

Transformations

Defining the Computation

Remember the **`map`** function from our first session?

$$\begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ \ldots \end{pmatrix}.\mathbf{map}(\mathit{fn}) \quad == \quad \begin{pmatrix} \mathit{fn}\,(X_1) \\ \mathit{fn}\,(X_2) \\ \mathit{fn}\,(X_3) \\ \mathit{fn}\,(X_4) \\ \mathit{fn}\,(X_5) \\ \ldots \end{pmatrix}$$

Notice how the output has the same size as the input

```
collection.map(mapper)
```

```
["HELLO", "World!"].map(word => word.toLowerCase) → ["hello", "world!"]
```

Transformations

Defining the Computation

Or the **`filter`** function which discards the elements that don't match a condition?

$$\begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ \dots \end{pmatrix} \textbf{.filter(}\textit{condition}\textbf{)} == \begin{pmatrix} \textit{condition}\,(X_1) \\ \cancel{\textit{condition}\,(X_2)} \\ \textit{condition}\,(X_3) \\ \cancel{\textit{condition}\,(X_4)} \\ \textit{condition}\,(X_5) \\ \dots \end{pmatrix} == \begin{pmatrix} X_1 \\ X_3 \\ X_5 \\ \dots \end{pmatrix}$$

Notice how the output can be smaller than the input

```
collection.filter(condition)
```

```
["map", "filter", "reduce"].filter(word => word.size <= 3)
→ ["map"]
```

Transformations

Defining the Computation

What about **flatMap**?

$$\begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \ldots \end{pmatrix}\mathbf{.flatMap(}\ \mathit{fn}\mathbf{)} \ == \ \mathbf{flatten(}\begin{matrix} \mathit{fn}\,(X_1) \\ \mathit{fn}\,(X_2) \\ \mathit{fn}\,(X_3) \\ \ldots \end{matrix}\mathbf{)} == \mathbf{flatten(}\begin{matrix} [X_{11}, X_{12}, X_{13}, \ldots] \\ [X_{21}, X_{22}, X_{23}, \ldots] \\ [X_{31}, Z_{32}, Z_{33}, \ldots] \\ \ldots \end{matrix}\mathbf{)} == \begin{pmatrix} X_{11} \\ X_{12} \\ \ldots \\ X_{21} \\ X_{22} \\ \ldots \end{pmatrix}$$

```
collection.flatMap(mapper) === flatten(collection.map(mapper))

[“hello world!”, “bye world!”].flatMap(str => str.split(“ ”))
               → [“hello”, “world!”, “bye”, “world!”]
```

Notice how the output can also be larger than the input

Transformations

Defining the Computation

All these operations are called **Transformations** and they have something very important in common...

Transformations

Defining the Computation

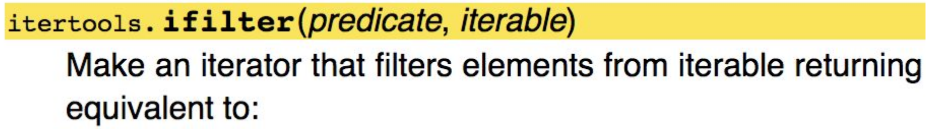
In Spark, **Transformations** **define** how a **computation** should be performed, but **they don't trigger** any **action** in the cluster. In other words, they're "*lazy*" computations.

Let's review an example to make this point completely clear...

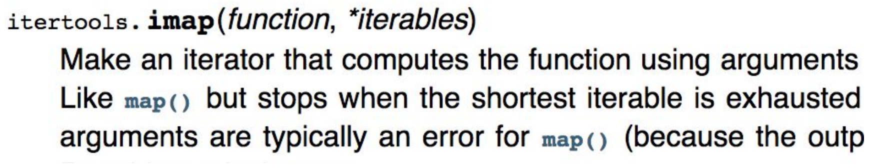
Transformations

Defining the Computation

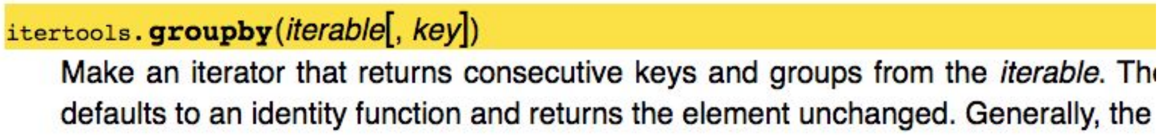
In Python, you might have seen these and similar functions in the `itertools` package:


`itertools.`**`ifilter`**(*predicate*, *iterable*)

Make an iterator that filters elements from iterable returning
equivalent to:


`itertools.`**`imap`**(*function*, **iterables*)

Make an iterator that computes the function using arguments
Like `map()` but stops when the shortest iterable is exhausted
arguments are typically an error for `map()` (because the outp


`itertools.`**`groupby`**(*iterable*[, *key*])

Make an iterator that returns consecutive keys and groups from the *iterable*. Th
defaults to an identity function and returns the element unchanged. Generally, the

Transformations

Defining the Computation in a Functional Style

Transformations follow the functional programing paradigm: They're immutable and declarative.

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that **treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data**. It is a **declarative programming paradigm**, which means programming is done with expressions or declarations instead of statements.

Wikipedia

They're also distributed and fault-tolerant so the model is more restrictive.

Transformations

Functional Programming Paradigm Reminder

Functions as first-class objects

Everything you can do with data can be done with functions e.g; passing functions to functions.

Recursion

Used as a primary control flow structure for looping.

Immutability

An immutable object cannot be modified after it is created.

Statements vs Expressions

Functional programming encourages the use of expressions (functions plus arguments).

What vs How

Let focus and worry about **what** needs to be computed and **not how** to compute it.

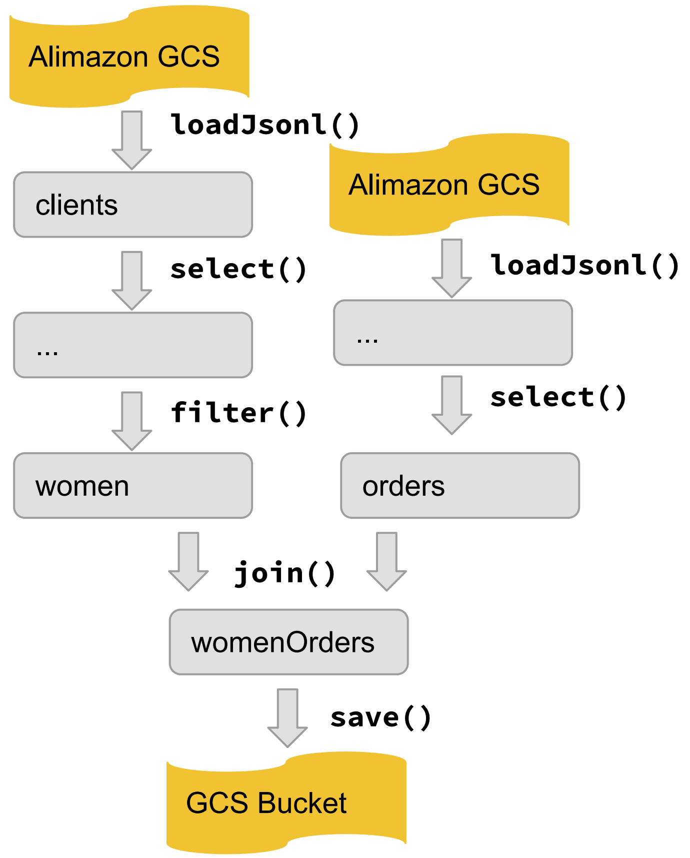
Transformations

Data Lineage Graph

```
val clients = loadJsonl("gs://de-training-input/alimazon/50000/clients/")
val clientOrders = loadJsonl("gs://de-training-input/alimazon/50000/client-orders/")

val orders = clientOrders.select($"client_id", $"id".alias("order_id"), $"product_id", $"total")
val women = clients
    .select($"id".alias("client_id"), $"name", $"registration_date")
    .filter($"gender" === "female")

val womenOrders = women.join(orders, "client_id")


womenOrders.write.
    format("com.databricks.spark.csv").
    option("header", true).
    option("delimiter", ",").
    save("gs://de-training-output-willebaldo/test-alimazon-orders")
```

Transformations

Action



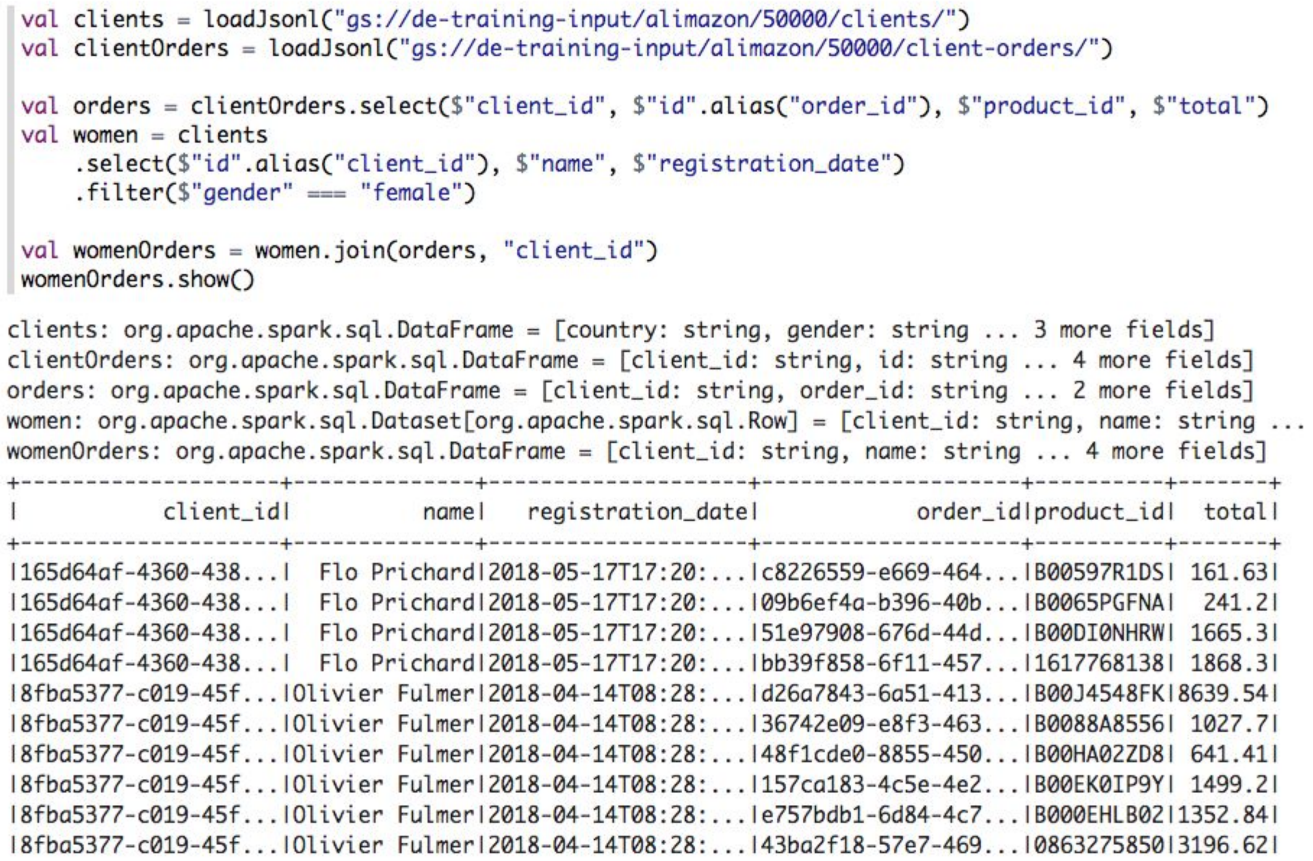
`womenOrders` doesn't contain any data as you might expect in a regular program.

It contains information about "**how to compute the result**" — the so-called "**data lineage graph**" or "**DAG**" of the result.

We'll review **`Actions`** in just a moment.

Transformations

Previewing the Result



```
val clients = loadJsonl("gs://de-training-input/alimazon/50000/clients/")
val clientOrders = loadJsonl("gs://de-training-input/alimazon/50000/client-orders/")

val orders = clientOrders.select($"client_id", $"id".alias("order_id"), $"product_id", $"total")
val women = clients
    .select($"id".alias("client_id"), $"name", $"registration_date")
    .filter($"gender" === "female")

val womenOrders = women.join(orders, "client_id")
womenOrders.show()
```

```
clients: org.apache.spark.sql.DataFrame = [country: string, gender: string ... 3 more fields]
clientOrders: org.apache.spark.sql.DataFrame = [client_id: string, id: string ... 4 more fields]
orders: org.apache.spark.sql.DataFrame = [client_id: string, order_id: string ... 2 more fields]
women: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [client_id: string, name: string ...
womenOrders: org.apache.spark.sql.DataFrame = [client_id: string, name: string ... 4 more fields]
+--------------------+--------------+--------------------+--------------------+----------+-------+
|           client_id|          name|   registration_date|            order_id|product_id|  total|
+--------------------+--------------+--------------------+--------------------+----------+-------+
|165d64af-4360-438...|  Flo Prichard|2018-05-17T17:20:...|c8226559-e669-464...|B00597R1DS| 161.63|
|165d64af-4360-438...|  Flo Prichard|2018-05-17T17:20:...|09b6ef4a-b396-40b...|B0065PGFNA|  241.2|
|165d64af-4360-438...|  Flo Prichard|2018-05-17T17:20:...|51e97908-676d-44d...|B00DI0NHRW| 1665.3|
|165d64af-4360-438...|  Flo Prichard|2018-05-17T17:20:...|bb39f858-6f11-457...|1617768138| 1868.3|
|8fba5377-c019-45f...|Olivier Fulmer|2018-04-14T08:28:...|d26a7843-6a51-413...|B00J4548FK|8639.54|
|8fba5377-c019-45f...|Olivier Fulmer|2018-04-14T08:28:...|36742e09-e8f3-463...|B0088A8556| 1027.7|
|8fba5377-c019-45f...|Olivier Fulmer|2018-04-14T08:28:...|48f1cde0-8855-450...|B00HA02ZD8| 641.41|
|8fba5377-c019-45f...|Olivier Fulmer|2018-04-14T08:28:...|157ca183-4c5e-4e2...|B00EK0IP9Y| 1499.2|
|8fba5377-c019-45f...|Olivier Fulmer|2018-04-14T08:28:...|e757bdb1-6d84-4c7...|B000EHLB02|1352.84|
|8fba5377-c019-45f...|Olivier Fulmer|2018-04-14T08:28:...|43ba2f18-57e7-469...|0863275850|3196.62|
```

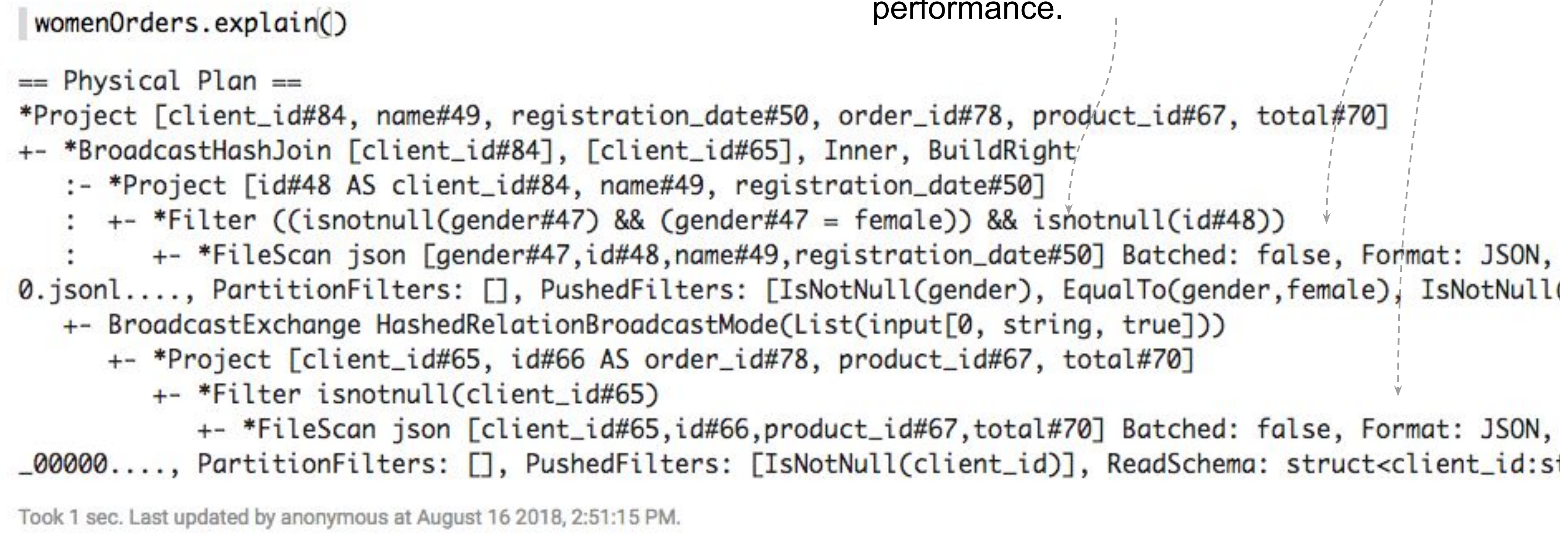
Transformations

Seeing the "Physical Plan" using "**`explain()`**"

Notice how Spark can run independent operations in parallel.

Use "**`explain()`**" to get an idea of how the computation will actually happen.

Notice how Spark automatically placed **`filter`** before **`select`** in order to improve the performance.



```
womenOrders.explain()

== Physical Plan ==
*Project [client_id#84, name#49, registration_date#50, order_id#78, product_id#67, total#70]
+- *BroadcastHashJoin [client_id#84], [client_id#65], Inner, BuildRight
   :- *Project [id#48 AS client_id#84, name#49, registration_date#50]
   :  +- *Filter ((isnotnull(gender#47) && (gender#47 = female)) && isnotnull(id#48))
   :     +- *FileScan json [gender#47,id#48,name#49,registration_date#50] Batched: false, Format: JSON,
0.jsonl...., PartitionFilters: [], PushedFilters: [IsNotNull(gender), EqualTo(gender,female), IsNotNull
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
      +- *Project [client_id#65, id#66 AS order_id#78, product_id#67, total#70]
         +- *Filter isnotnull(client_id#65)
            +- *FileScan json [client_id#65,id#66,product_id#67,total#70] Batched: false, Format: JSON,
_00000...., PartitionFilters: [], PushedFilters: [IsNotNull(client_id)], ReadSchema: struct<client_id:s
```

Took 1 sec. Last updated by anonymous at August 16 2018, 2:51:15 PM.

Transformations

Implications of “Lazy” Computation

Some implications of the “lazy” nature of **Transformations**:

- Operations can be rearranged to optimize the data pipeline (e.g. by pushing “filter” operations to the data sources, if they support it).

- All steps in the computation must be repeated on every evaluation (but we can use *caching* at any point, as we’ll see later).

Transformations

Lazy Computation

Some examples of **Transformations** in the Dataset API:

*(Note that the return type continues to be a **Dataset**, this is, a distributed collection.)*

```
def filter[T](condition: Column): Dataset[T]
```

Returns a new Dataset that contains only the rows that pass the condition.

```
def select[U1](c1: TypedColumn[T, U1]): Dataset[U1]
```

Returns a new Dataset by computing the given Column expression for each element.
Note how this is conceptually “map”.

```
def join(right: Dataset[_], usingColumn: String)
: Dataframe
```

Inner equi-join with another DataFrame using the given column.

Transformations

Lazy Computation

```
def distinct[T](): Dataset[T]
```
Returns a new Dataset that contains only the unique rows from this Dataset.

```
def intersect[T](other: Dataset[T]): Dataset[T]
```
Returns a new Dataset containing rows only in both this Dataset and another Dataset.

```
def groupByKey[T](func: (T) => T):
    KeyValueGroupedDataset[T]
```
*Returns a KeyValueGroupedDataset where the data is grouped by the given key **`func`**.*

Check https://bit.ly/2JPerSu to see more of the API

Transformations

Avoid Premature Optimization

- Spark can –and will– optimize the pipeline as much as it can (e.g. by doing “query folding”) when you use **Datasets** and **Dataframes**.
- As a result, it’s not a good idea to try and optimize “by hand” and prematurely. Learn how the optimizers work instead (**Catalyst** and **Tungsten**) and profile.
- This is another reason to avoid using **RDDs** directly if possible.

Transformations

The Cost of Transformations

- As we'll see in detail in a future session, not all transformations are the same when it comes to their cost.
- Some can be very cheap (e.g. **`map`**, **`flatMap`**) while others can be very costly depending on a number of factors (e.g. **`join`**)
- Stay tuned to the "Shuffling" sessions to learn more about this.

“Transformations” Recap

What We’ve Learned So Far

- What are **Transformations**
- Examples of **Transformations** in Spark
- How **Transformations** define “**data lineage graphs**” (**DAGs**) and the “**lazy**” nature of their computation

QA

WIZELINE

ACTIONS

Actions

Getting Results

So, if **Transformations** only define computation…

How do we actually get results?

Actions

Getting Results

In Spark, **Actions** **trigger** a **computation** in the cluster. These operations are needed when we want to get a result (e.g. to write an output file with the result of our data processing).

Actions

Improving Performance

Some examples of ***Actions*** in the Dataset API:
(Note that the return type is no longer a ***Dataset*** *since the result is no longer distributed.)*

```
def reduce[T](func: (T, T) => T): T
```

Reduces the elements of this **`Dataset`** *using the specified binary function.*

```
def count(): Long
```

Counts the number of elements in this **`Dataset.`**

```
def collect[T](): Array[T]
```

Returns an array with all elements in this **`Dataset.`**

Check https://bit.ly/2JPerSu to see more of the API

“Actions” Recap

What We’ve Learned So Far

- What are **Actions**
- Examples of **Actions** in Spark
- How **Actions** trigger the computation of the DAGs defined by Transformations

QA

WIZELINE

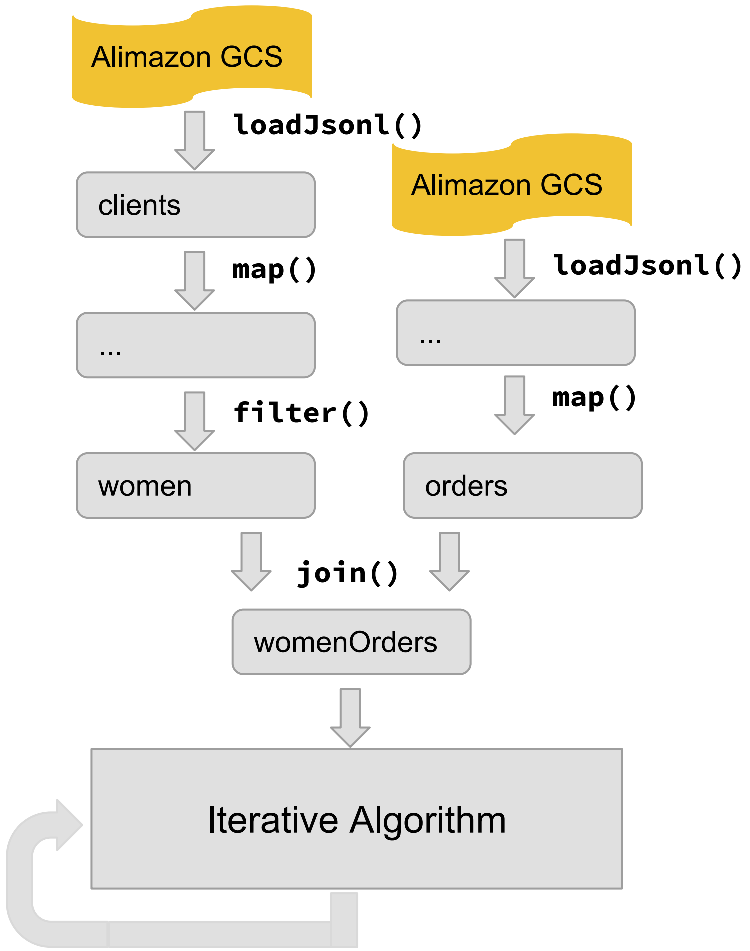
5 minutes

CACHING

Caching

Improving Performance

What would happen if we had to call this workflow **N** number of times (for example, if it were an input to an iterative algorithm)?



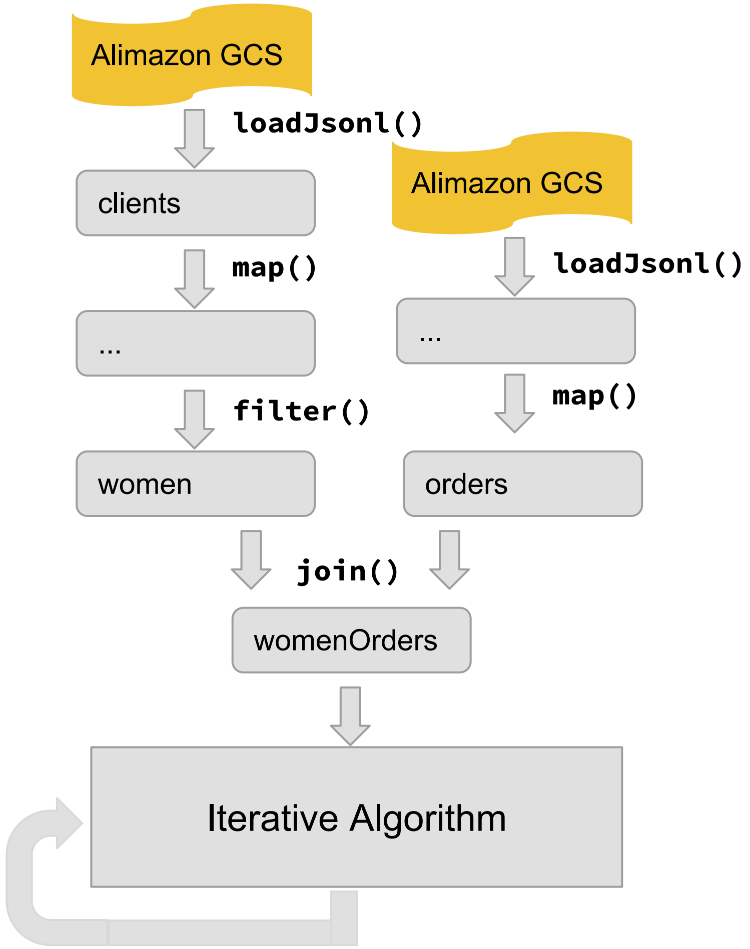
Caching

Improving Performance

Remember that “transformations” are lazy?

That means we would be reading data from the GCS sources **N** times!

Fortunately, there is an easy way to fix that: **`cache()`** or **`persist(...)`**

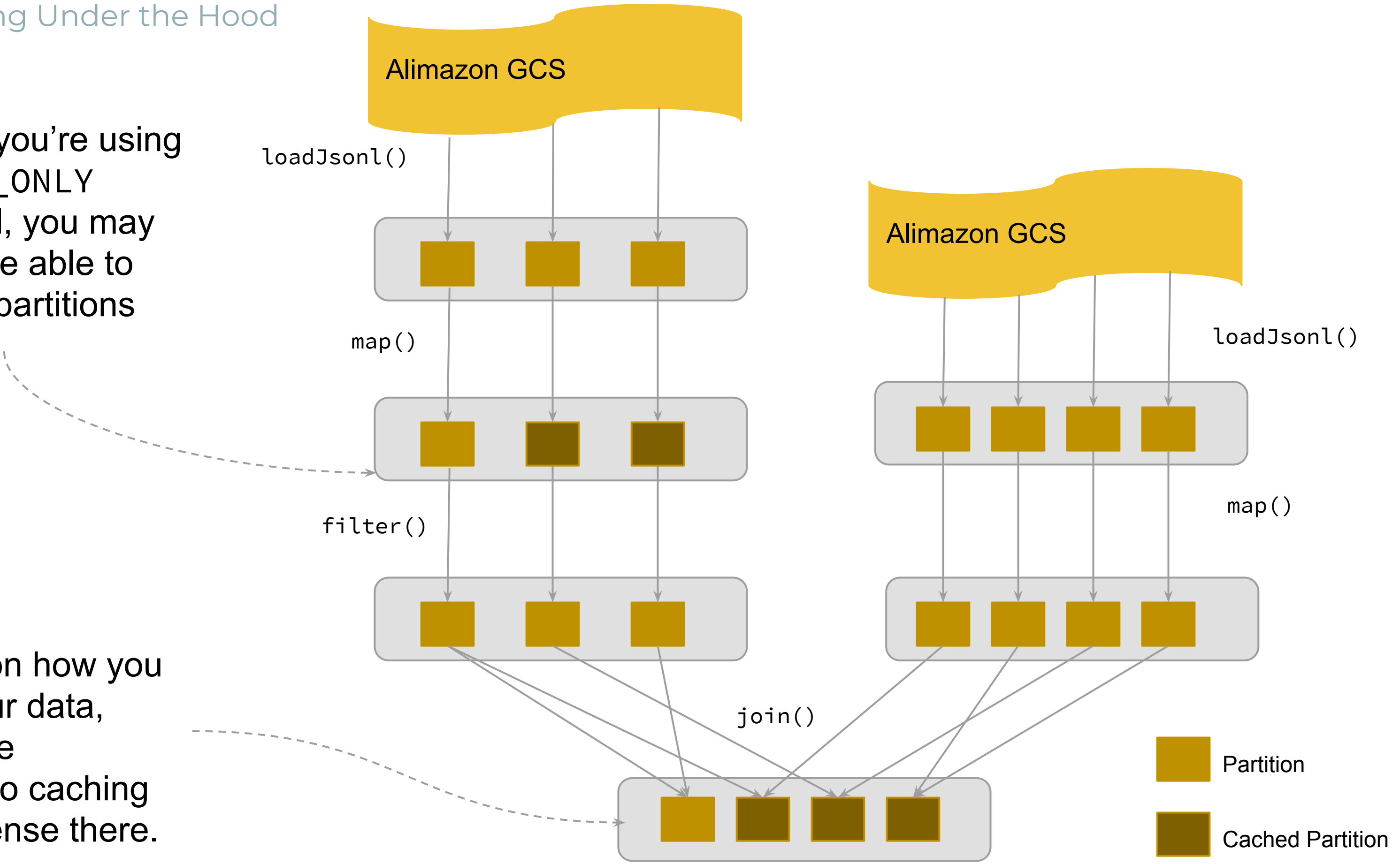


Caching

Looking Under the Hood

Note that, if you're using the `MEMORY_ONLY` storage level, you may not always be able to keep all the partitions cached.

Depending on how you organize your data, "joins" can be expensive, so caching can make sense there.



Caching

Improving performance

The caching API in **`Dataset`**:

```
def persist(newLevel: StorageLevel): Dataset
```

Persist this Dataset with the ***given storage level.***

(`MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, MEMORY_AND_DISK_SER, DISK_ONLY, ...`)

```
def cache(): Dataset
```

Persist this Dataset with the ***default storage level.*** (`MEMORY_AND_DISK`)

Check https://bit.ly/2JPerSu to see more of the API

Caching

Caching Strategies (A.K.A. Storage Levels)

Storage Level	Meaning
MEMORY_ONLY	Store as deserialized Java objects in the JVM. If it does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. **This is the default level.**
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If it does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
DISK_ONLY	Store the RDD partitions only on disk.

Caching

Caching Strategies (A.K.A. Storage Levels)

Storage Level	Meaning
MEMORY_ONLY_SER *(Java and Scala)*	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.

Caching

Is Caching All I Need?

So, is <u>Caching</u> the solution to all my (performance) problems?

Caching

Is Caching All I Need?

So, is <u>Caching</u> the solution to all my (performance) problems?

NO

Though it may seem like you would always want to use <u>Caching</u>, be aware that:

- It may incur a **serialization cost** (depending on the strategy used) and,
- It is **limited by** the amount of **memory and/or disk** that can be stored
- It may **not be the fastest way** to get the data **if you cache excessively in the wrong places** (e.g. you may get in the way of the optimizer).

So it is important to not use it blindly and **be judicious in its use.**

Let’s see when it makes sense to use it...

Caching

To Cache or Not to Cache?

When should I use Caching in general?

- If your **`Datasets`**/**`Dataframes`** have a very **large evaluation cost** and you need to use it **more than once**, caching makes sense.
- If your application is iterative in nature (e.g. iterative machine learning and graph processing algorithms).
- If you’re doing a lot of interactive ad-hoc querying.

Caching

Choosing a Storage Level

How should I choose the <u>Storage Level</u>?

Spark's **storage levels** are meant to **provide** different **trade-offs between memory usage and CPU efficiency**:

- If your data fits comfortably with the default storage level (`MEMORY_ONLY`), leave them that way. This is the most CPU-efficient option, allowing operations to run as fast as possible.

- If not, try using `MEMORY_ONLY_SER` and selecting a fast serialization library to make the objects more space-efficient, but still reasonably fast to access. (Java and Scala)

Caching

Choosing a Storage Level

How should I choose the <u>Storage Level</u>?

- Avoid writing to disk unless you’re running out of memory on your application and the functions that compute your dataset are expensive. Otherwise, recomputing a partition may be as fast as reading it from disk.

- Use the replicated storage levels if you want fast fault recovery (for example, if using Spark to serve requests from a web application).

  - All the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the dataset without waiting to recompute part of the lost data.

Caching

To Cache or Not to Cache?

Caching Gotchas

- **Caching ALL** of the generated **Datasets/Dataframes is not a good strategy** as useful cached blocks may be evicted from the cache well before being re-used.

- **When memory is scarce,** it is recommended to **use `MEMORY_AND_DISK` caching strategy** such that evicted blocks from cache are saved to disk. Reading the blocks from disk is generally faster than re-evaluation (but be sure to profile if you have doubts!)

- **If extra processing cost can be afforded**, you can **use MEMORY_AND_DISK_SER** to further reduce the memory footprint of the cached **Datasets/Dataframes**.

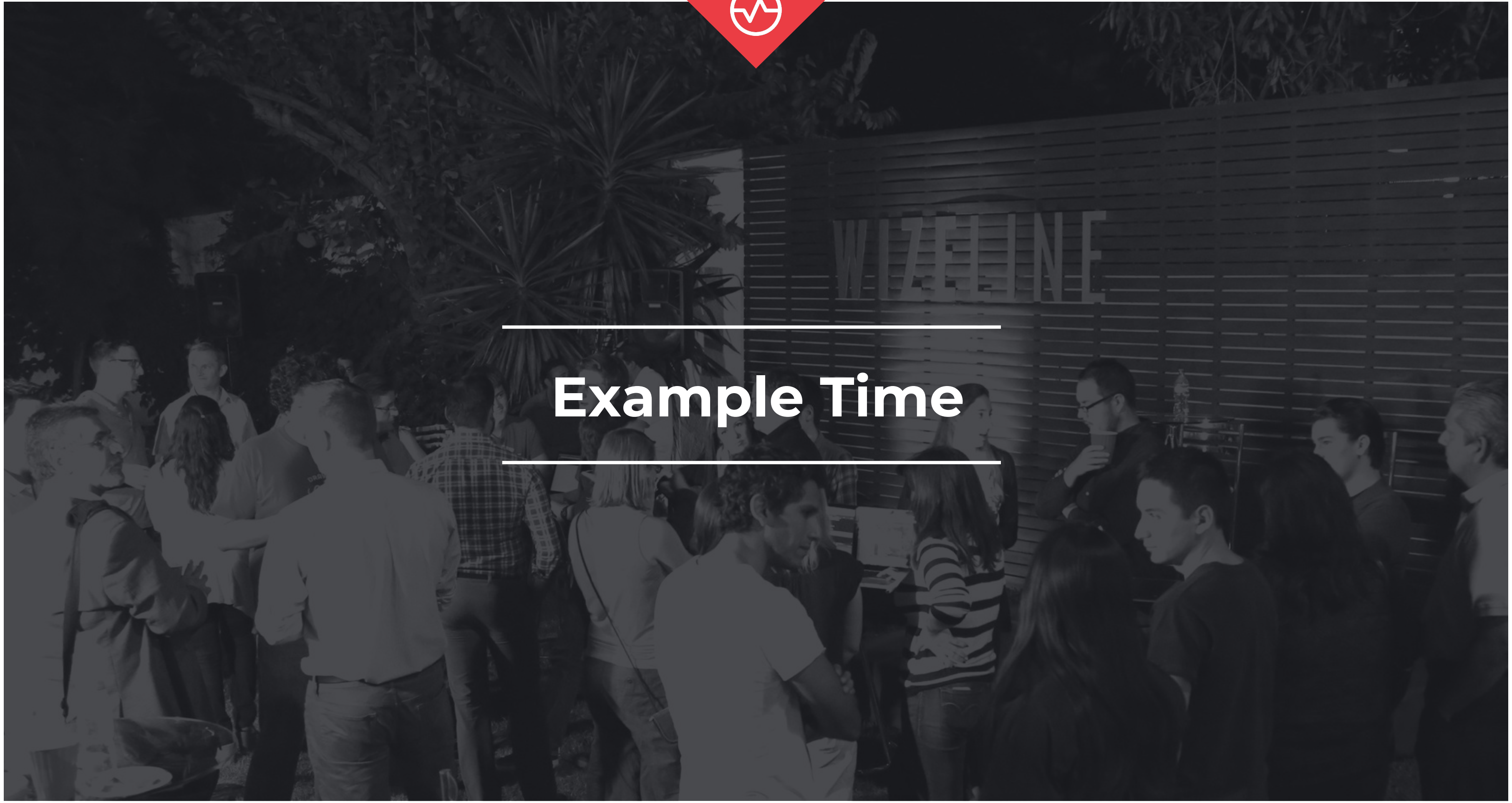
Caching

What We've Learned So Far

- How Caching can help improve performance in certain scenarios
- Which are the different strategies for Caching
- When to use Caching and some of the gotchas to consider

QA

WIZELINE



Example Time

Example

Problem Description

Matrix Polynomials Evaluation

Consider the evaluation of the following polynomials (where ***x*** is a Matrix):

$$3x^3 + x^2$$

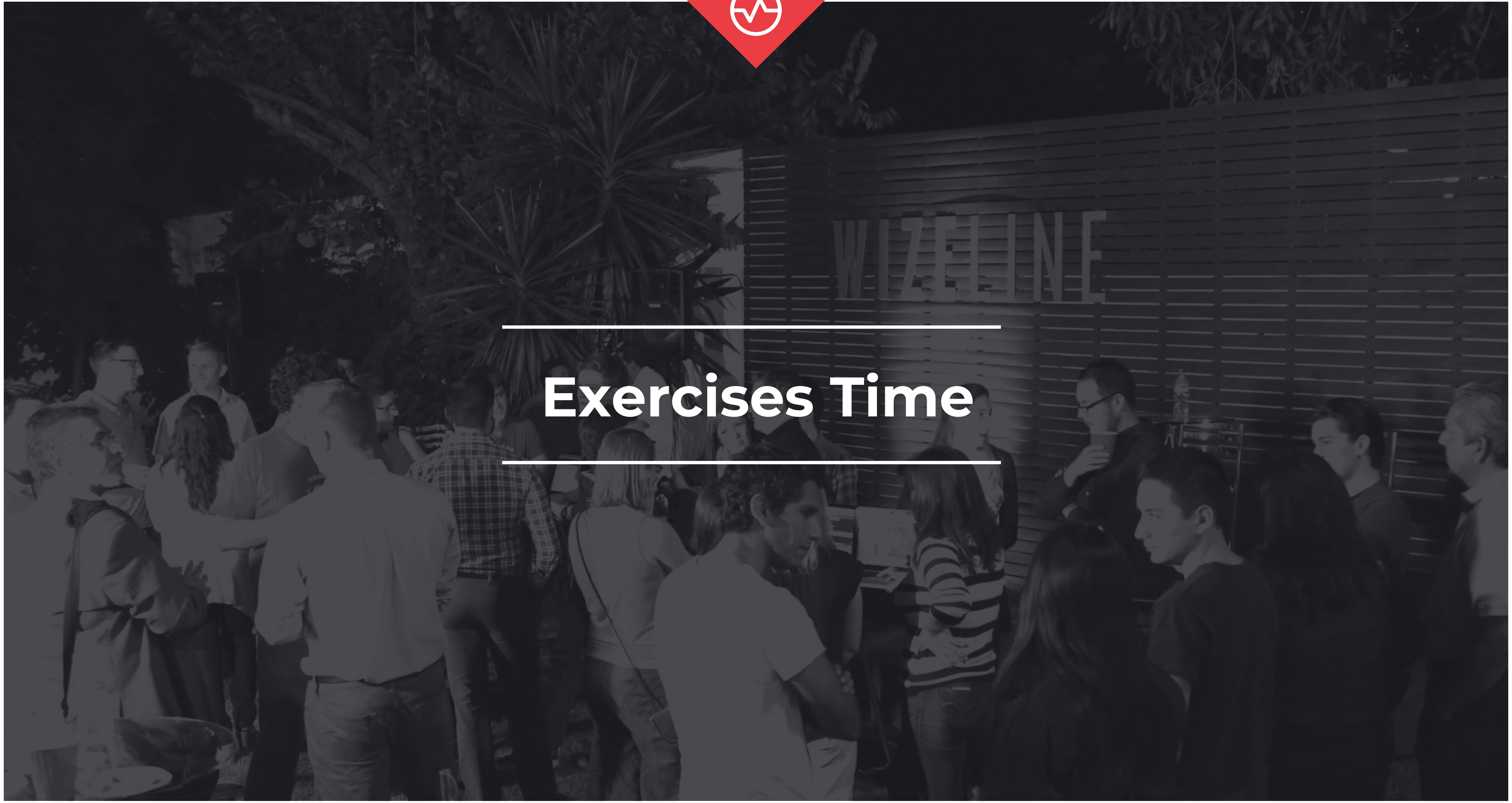
$$x^3 + 2x^2 + x$$

Example

Questions on StackOverflow

Solution Sketch

```
val x = blockBigMatrix
val x2 = x.multiply(x).cache()
val x3 = x2.multiply(x).cache()
x2.toCoordinateMatrix().entries.saveAsTextFile(s"...")
x3.toCoordinateMatrix().entries.saveAsTextFile(s"...")
val result1 = x3.add(x3).add(x3).add(x2)
result1.toCoordinateMatrix().entries.saveAsTextFile(s"...")
val result2a = x2.multiply(x).add(x2).add(x2).add(x)
result2a.toCoordinateMatrix().entries.saveAsTextFile(s"...")
val x3bis = x.multiply(x).add(x).cache()
val result2b = x3bis.multiply(x).add(x3bis)
result2b.toCoordinateMatrix().entries.saveAsTextFile(s"...")
```



Exercises Time

Exercises

Different Caching Strategies

Using the same dataset of the example, **experiment with different caching strategies:**

- **Is there any difference in the resulting execution times?**
- **Can you explain the observed behavior?**

Different Caching Strategies - Hints

Using the same dataset of the example, **experiment with different caching strategies:**

- **Is there any difference in the resulting execution times?**
- **Can you explain the observed behavior?**

Hint 1: *Check the `org.apache.spark.storage.StorageLevel` API*

Exercises

Looking at the Computation Graph

Using the same dataset of the example, **can you get the logical and physical execution plans of the final computation?**

Is there anything that draws your attention? Discuss with your classmates and instructor.

Looking at the Computation Graph

Using the same dataset of the example, **can you get the logical and physical execution plans of the final computation?**

Is there anything that draws your attention?

Hint 1: *Check the two variants of the `explain()` method in the **Datasets/Dataframe** APIs*

Dealing with More Complicated Polynomials

What would happen if you had to deal with larger polynomials? For example:

$$x^{11} + x^9 + x^7 + x^5 + x^4 + x^3 + x^2 + x$$

- How would you partition your evaluation so as to minimize computation time?
- How would you partition your evaluation so as to minimize computation while keeping the amount of memory/disk being used as small as possible?

Exercises Recap

What We've Learned So Far

- How to implement different caching strategies
- How to look at the logical and physical plans for the computation of a Dataset/Dataframe
- How to use the concepts covered in this session in larger datasets

QA

WIZELINE

FURTHER READING

For the Knowledge Hungry...

Transformations/Actions
Big Data Analysis with Scala and Spark
(Coursera MOOC – Week 1 and 2)

Transformations and Actions
(Spark, The Definitive Guide)

Caching
Choosing a Storage Level
(Spark Programming Guide)

To Cache or Not To Cache
(Unravel Data)

Transformations/Actions/Caching
Transformations and Actions in RDDs
(Original Research Paper)

Assignment

To Work on Your Own at Home

Business Report

Assignment

Business Report

Alimazon’s Board of Directors sent a requirement for a full report to have more insights on products and consumers.

The report should be executed as one application and should run with good performance, multiple executions and maintenance of the report for future customizations are expected.

Assignment

Input/Output Description

The dataset you'll need to read is located in the Google bucket:

```
gs://de-training-input-bucket/alimazon/200000/client-orders/
```

Inside the bucket, you'll find many files with the following naming convention:

```
part_timestamp_number.jsonl.gz
```

For example:

```
gs://de-training-input-bucket/alimazon/200000/client-orders/
            part_20180709T170251_00000.jsonl.gz
            part_20180709T170252_00001.jsonl.gz
                           ...
```

Assignment

Subtitle

Question(s) to answer in the assignment:

1. Top 10 selling products by day of week by gross sales and by number of orders
2. Top 10 customers by month by number of orders and spending.

Where Can I Get this Presentation?

C1 - Data Engineering Academy

Your feedback is very valuable!

https://bit.ly/2MCroo9