Big Data, Techniques and Platforms
Cours 3/8: Dataframes

Francesca Bugiotti

CentraleSupélec

October 17, 2023

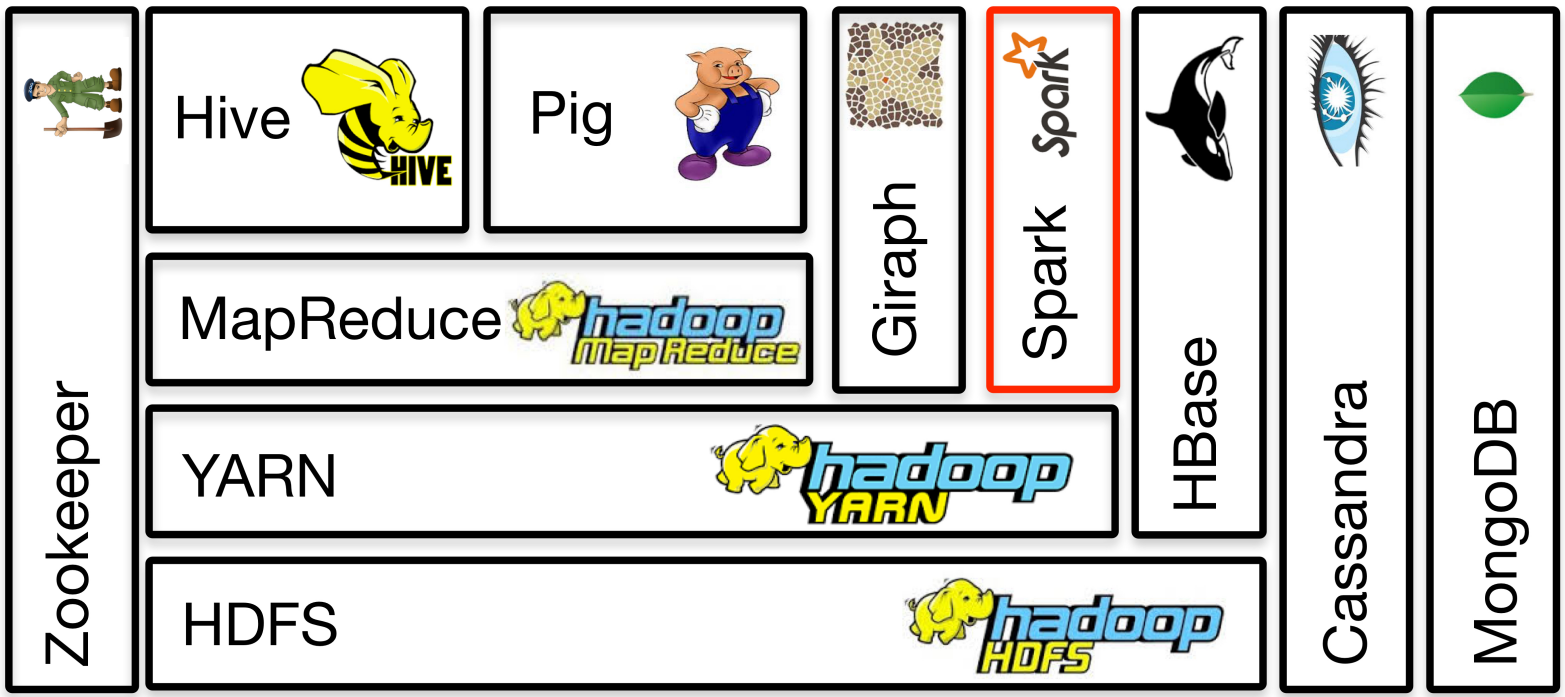
Objectives

- PySpark

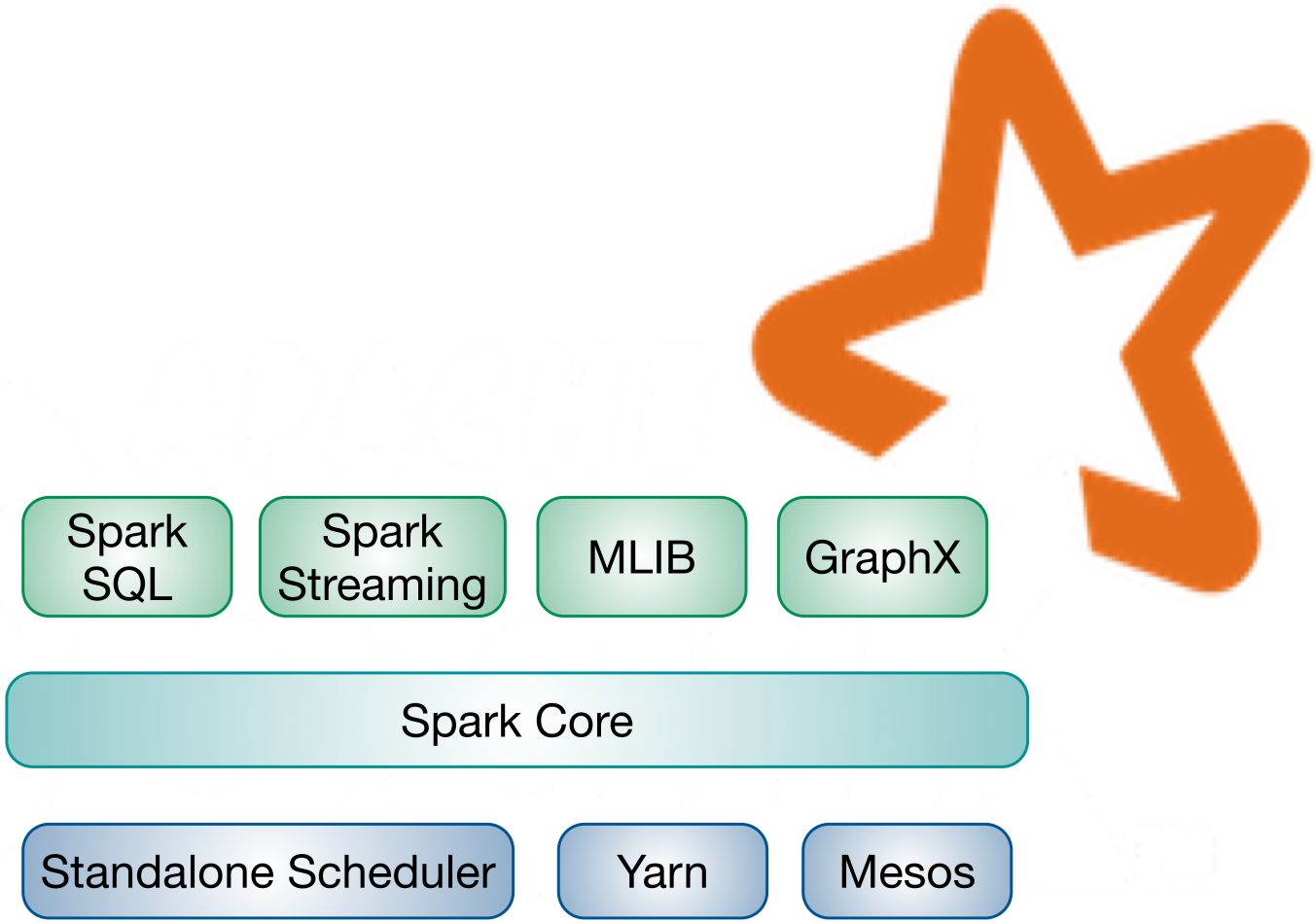
Topics

- PySpark DataFrames
- Pandas DataFrames

Plan


Zookeeper
Hive
HIVE
Pig
Giraph
Spark
Spark
HBase
Cassandra
MongoDB
MapReduce
hadoop
MapReduce
YARN
hadoop
YARN
HDFS
hadoop
HDFS

Spark Stack


Spark SQL
Spark Streaming
MLIB
GraphX
Spark Core
Standalone Scheduler
Yarn
Mesos

RDD: Resilient Distributed Dataset

Currently two types of RDDs

- Parallelized collections: created by executing operators on an existing programming collection
  - Developer can specify the number of slices to cut the dataset into
  - Ideally 2-3 slices per CPU
- Hadoop Datasets: created from any file stored on HDFS or other storage systems supported by Hadoop (S3, Hbase, etc.)
  - These are created using SparkContext's textFile operator
  - Default number of slices in this case is 1 slice per file block

Operators over RDD

Programmer can perform three types of operations:

1. Transformations
2. Actions
3. Persistence

How does Spark work?

- User applications create RDDs, transform them, and run actions
- This results in a DAG (Directed Acyclic Graph) of operators
- DAG is compiled into stages
- Each stage is executed as a series of Tasks (one Task for each Partition)
- Actions drive the execution

RDD are useful when:

- low-level transformation and actions and control on your dataset
- data is unstructured
  - stream
- usage of functional programming
- schema-less data

Big data opened perspectives more and more challenging

- explore the power of distributed data processing
- exploit the Spark opportunities

Spark Dataframe

Version considered of Spark: starting 3.1.1[2]

- DataFrames in R
- Python (Pandas)
- Spark opportunities

Spark DataFrame

Contract

- Scale from kilobytes of data on a single laptop to petabytes on a large cluster
- Support for a wide array of data formats and storage systems
- Exploit Spark SQL Catalyst optimizer
- Offer high integration
- Rely on Spark APIs for Python, Java, Scala, and R

What is a Spark DataFrame?

Definition

distributed collection of data organized into named columns

What is a Spark DataFrame?

Definition:

distributed collection of data organized into named columns

Conceptually equivalent to a table in a relational database

What is a Spark DataFrame?

Optimization

Spark distributed environment and architecture

Creation

- structured data files
- tables in Hive
- external databases
- existing RDDs

Spark Session

The entry point into all functionality in Spark DataFrames:

```
from pyspark.sql import SparkSession

spark = SparkSession \
        .builder \
        .appName("Python Spark SQL basic example") \
        .config("spark.some.config.option", "some-value") \
        .getOrCreate()
```

Example in Python

```
spark is an existing SparkSession

dfPeople = spark.read.json("../people.json")


Displays the content of the DataFrame to stdout

dfPeople.show()
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Example in Python

Once built, Spark provides a domain-specific language for distributed data manipulation of DataFrames.

Example in Python

```
// Select only the "name" column

dfPeople.select("name").show()

// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+
```

Example in Python

```
// Select everybody, but increment the age by 1

dfPeople.select(df['name'], df['age'] + 1).show()

// +-------+----------+
// |   name|(age + 1)|
// +-------+----------+
// |Michael|      null|
// |   Andy|        31|
// | Justin|        20|
// +-------+----------+
```

Example in Python

```
// Select people older than 21

dfPeople.filter(df['age'] > 21).show()

// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+
```

Example in Python

```
// Count people by age

dfPeople.groupBy("age").count().show()

// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

Example in Python

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")

sqlDF.show()

// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

注册为一个临时sql视图

Example in Python

- create a view
- run SQL queries on the view

Example in Python

Temporary views in Spark SQL are session-scoped.

```
// Register the DataFrame as a global temporary view
dfPeople.createGlobalTempView("people")

// Global temporary view is tied to a system preserved
// database `globalTemp`
spark.sql("SELECT * FROM globalTemp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Example in Python

`withColumn()` function

Conceptually add a new column to a Spark DataFrame using the `withColumn()` function: pay attention, a new DataFrame is created as output
The function will take 2 parameters:

- The column name
- The value to be filled across all the existing rows

We want to add the `Country` column

```
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

Using a Spark DataFrame

Update a DataFrame

- a DataFrame is immutable
- It is necessary to:
  - create a view
  - update the view

Example in Python

```
df.withColumn("Country" , lit("France")).show()
```

```
// +----+-----+-----------+
// | age|count|  country|
// +----+-----+-----------+
// |  19|    1|France  +
// |null|    1|France  +
// |  30|    1|France  +
// +----+-----+-----------+
```

Spark DataFrames in action

Similar to RDDs, DataFrames are evaluated lazily:

- Computation only happens with actions
- Optimized by applying specific techniques:
  - such as predicate push-downs
  - bytecode generation
- DataFrame operations are also automatically parallelized and distributed on clusters

Main challenges in BigData

Challenges with BigData:

- apply advanced algorithms
- analyze/distribute computation
- store BigData
- integrate/distribute multiple heterogebeiys sources
- standard
- automatic scale

Main challenges in Big Data

Challenges with BigData:

- apply advanced algorithms → Pandas conversions
- analyze/distribute computation → Spark computation
- store big data → optimized data structure is necessary: Parquet
- integrate/distribute multiple big data sources → intermediate exchange formats (Arrows)
- standard → Docker
- automatic scale → Kubernetes

Python and DataFrames = Pandas

Python and DataFrames

Union between the richness of Python libraries and Spark DataFrame structure

Pandas

Pandas is an open source, BSD-licensed library written for the Python programming language that provides fast and adaptable data structures, and data analysis tools

- built on the Numpy package

Pandas

Formats and Sources supported by DataFrames

Pandas

Characteristics

- Tabular data with heterogeneously-typed columns
  - SQL table
  - Excel spreadsheet
- Ordered and unordered time series data
- Arbitrary matrix data with row and column labels
- Any other form of observational or statistical data sets

Pandas

Characteristics

The two primary data structures of Pandas

- Series
- DataFrames

Pandas DataFrame

Definition

Pandas DataFrame is a 2-D labeled data structure with columns of potentially different type.

- Recall of tabular data
- Flexible table

Pandas DataFrame

Import from:

- CSV or TSV file
- SQL database
- Python structures

The returned objects, the DataFrames, are quite similar to tables.

Example

```
peopleDict = {
      'name' : ["Michael", "Andy", "Justin"],
      'age' : [null, 30, 19]}

import Pandas as pd

pandasdf = pd.DataFrame(peopleDict)
```

```
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Example

- Columns are associated with types
- The type is fixed

Pandas

Provides different functionalities for data analysis.
You can look at data following:

- data in rows
- data columns

Instructions for:

Given a DataFrame `df`:

- display rows `df.head()`, `df.tail()`
- access based on the index of a column
- run math operations on data columns

Example in Python

In Pandas we can also apply the MapReduce programming paradigm:

- `apply()`
  - is used to apply a function along an axis of the DataFrame or on values of Series
- `applymap()`
  - is used to apply a function to a DataFrame element-wise

Pandas

Pandas enables advanced computation:

- complex map-reduce programs
- in the `apply` any kind of function can be specified
- panda open the access to the power of Python libraries
  - scikit-learn AI functions

Pandas

- allows data computation in Pandas is in memory
- provides data structures for in-memory analytics
- analyzing datasets that are larger than memory is somewhat tricky

Solutions:

- manually cut fraction of data
  - difficult to predict
  - data changes and grows
- attention: some Pandas operations need to make intermediate copies $\rightarrow$ same problem

Solution

Look for a methodology that integrates:

1. the flexibility of resource management and data storage of Spark
2. the Pandas library

Conversion

It is possible to convert directly between Pandas DataFrame and Spark DataFrame:

```
//Convert Spark DataFrame to Pandas

pandasdf = people.toPandas()

//Create a Spark DataFrame from Pandas
people = context.createDataFrame(pandasdf)
```

Steps

Communication between Pandas and Spark:

- create a Spark DataFrame even so big that is not possible to read it in memory in Pandas
- split according to a Spark operation the DataFrame in smaller pieces
- apply inside each piece of data stored in a different DataFrame the power of Pandas Python computation
- re-combine data thanks to Spark

Steps

Create a Spark DataFrame:

- easy, we use the standard procedure

Steps

Split according to a Spark operation the DataFrame in smaller pieces

- easy, we use the dedicated operations with `groupby()` operation

Steps

Apply Pandas computation in each piece of data:

- Pandas function API

Pandas + Spark: function APIs

Pandas function APIs[1] enables to directly apply a Python native function to a PySpark DataFrame.

The Python native function:

- input: Pandas instances
- output: Pandas instances

The actors:

- Spark handles the data distribution
- Arrow offers the data exchange intermediate format
- Pandas works with data

Pandas + Spark: function APIs

There are three types of Pandas function APIs:

- Grouped map
- Map
- Co-grouped map

Behind these function a split-apply-combine pattern is present.

Split-apply-combine pattern

Split-apply-combine consists of three steps:

1. Split the data into groups
2. Apply a function on each group
3. Combine the results

groupBy()

Implemented by `groupBy().applyInPandas()`

1. Split the data into groups `DataFrame.groupBy`
2. Apply a function on each group:
   - The input and output of the function are both pandas.DataFrame
   - The input data contains all the rows and columns for each group
3. Combine the results into a new Spark `DataFrame`

Split-apply-combine pattern

The key points:

1. the `Python` function on each group
   - powerful
2. the output schema defined in a `StructType` object or a `String`

Split-apply-combine pattern

You remember that a DataFrame is associated with the columns and the computation

The column labels of the returned `pandas.DataFrame`:

1. match the field names in the defined output schema
2. match the field data types by position

Split-apply-combine pattern

From the computation point of view:

- All data for a group is loaded into memory before the function is applied.

Big Data in-memory split data process.

Example in Python

```
dfs = spark.createDataFrame(
    [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
    ("id", "v"))

def subtractMean(pdf):
    // pdf is a pandas.DataFrame
    v = pdf.v
    return pdf.assign(v=v - v.mean())

dfs.groupby("id").applyInPandas(
subtractMean, schema="id long, v double").show()
```

```
//  | id|   v|
// +---+----+
//  |  1|-0.5|
//  |  1| 0.5|
//  |  2|-3.0|
//  |  2|-1.0|
//  |  2| 4.0|
// +---+----+
```

More

Of course not all is split-apply-combine:

- `mapInPandas()`
- `cogroup()`

Map

You map an operation on the instances by a `pandas.DataFrame` that represents the current `PySpark DataFrame` and returns the result as a `PySpark DataFrame`.

Example in Python

```
dfs = spark.createDataFrame([(1, 21), (2, 30)],
      ("id", "age"))

def filter_func(iterator):
    for pdf in iterator:
        yield pdf[pdf.id == 1]

dfs.mapInPandas(filter_func, schema=df.schema).show()
// +---+---+
// | id|age|
// +---+---+
// |  1| 21|
// +---+---+
```

Cogrouped map

It is a functions that takes in input two DataFrames:

1. analyses the common key and shuffles the data
2. the DataFrames to be cogrouped by a common key
3. a function applied to each cogroup

Split-apply-combine pattern

Implemented by
`DataFrame.groupby().cogroup().applyInPandas()`

The input of the function is two `pandas.DataFrame`

- analyses the common key and shuffles the data
- the `pandas.DataFrame` to be cogrouped by a common key
- a `Python` function applied to each cogroup

Combine the results into a new `DataFrame`

Example in Python

```
import pandas as pd

dfs1 = spark.createDataFrame(
    [(20000101, 1, 1.0), (20000101, 2, 2.0),
        (20000102, 1, 3.0), (20000102, 2, 4.0)],
    ("time", "id", "v1"))

dfs2 = spark.createDataFrame(
    [(20000101, 1, "x"), (20000101, 2, "y")],
    ("time", "id", "v2"))
```

Example in Python

We want to join two DataFrames on the nearest key

```
def asofjoin(l, r):
    return pd.merge_asof(l, r, on="time", by="id")

dfs1.groupby("id").cogroup(dfs2.groupby("id")).
        applyInPandas(
    asofjoin, schema="time int, id int,
        v1 double, v2 string").show()
```

```
+--------+---+---+---+
|    time| id| v1| v2|
+--------+---+---+---+
|20000101|  1|1.0|  x|
|20000102|  1|3.0|  x|
|20000101|  2|2.0|  y|
|20000102|  2|4.0|  y|
+--------+---+---+---+
```

Plan

Parquet

Parquet is an open source file format available to any project in the Hadoop ecosystem

Parquet

efficient and performant flat columnar storage format of data compared to row based files

- CSV
- JSON
- TSV files

Implementation

- record shredding
- assembly algorithm

- efficient compression
- efficient encoding
- easy to read selected columns
- available APIs

Distribution

- groups of rows
- partitioning by columns values
- easy compression

Parquet APIs and frameworks

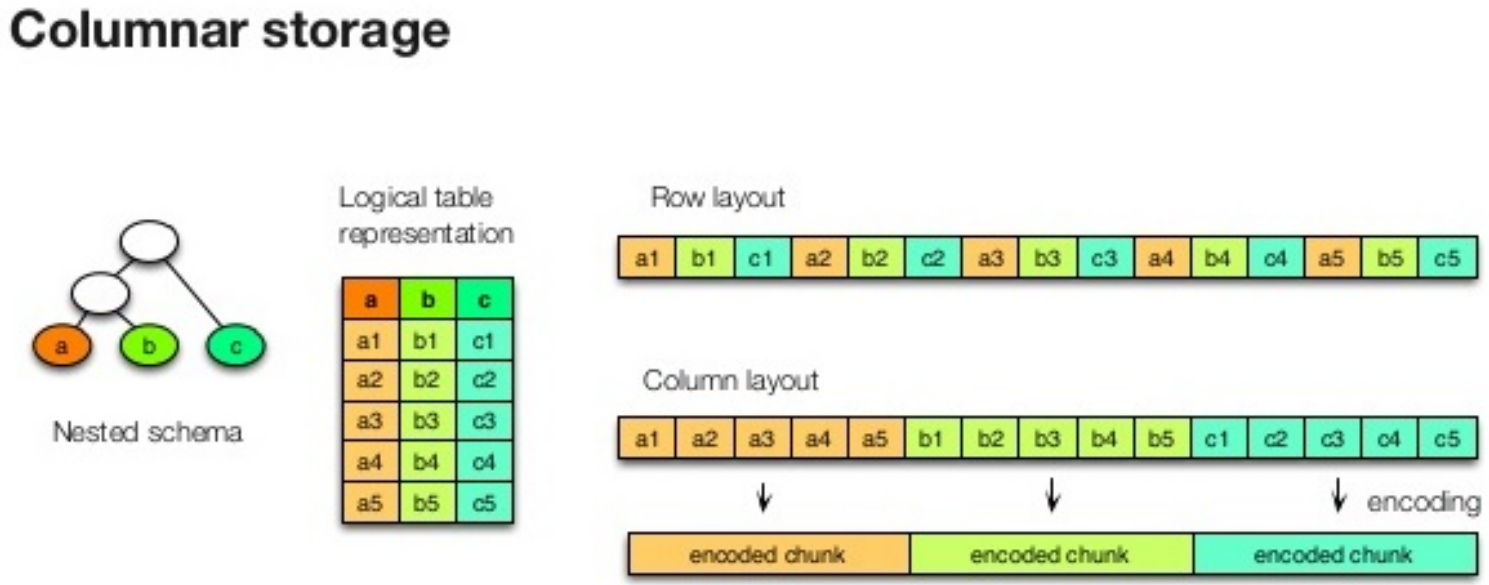
Frameworks

- Spark
- MapReduce
- Scalding (Scala library)
- etc.

Query engines

- Hive
- Presto
- Pig
- etc.

Parquet



Columnar data independent format [a]

[a] https://community.ptc.com/t5/IoT-Tech-Tips/Parquet-Data-Format-used-in-ThingWorx-Analytics/td-p/535228

DataFrames and Parquet

- save DataFrame in Parquet with Spark
- read DataFrame in Parquet with Spark
- save DataFrame in Parquet with Pandas
- read DataFrame in Parquet with Pandas

Example in Python

```
//DataFrames can be saved as Parquet files,
//maintaining the schema information.
peopleDF.write.parquet("people.parquet")

//Read in the Parquet file created above.
//Parquet files are self-describing
//so the schema is preserved.
//The result of loading a parquet file is also a DataFrame.
parquetFile = spark.read.parquet("people.parquet")
```

Example in Python

```
import pandas as pd
//Pandas DataFrames can be saved as Parquet files,
//maintaining the schema information.
pandasdf.to_parquet('people.parquet')

//Read in the Parquet file created above.
//Parquet files are self-describing
//so the schema is preserved.
//The result of loading a parquet in this case
//is a Pandas DataFrame
pandasdf = pd.read_parquet('df.parquet.gzip')
```

DataFrames and Parquet

It is interesting to look at how data are stored in Spark given the Parquet data format:

- The Table is partitioned
- Data is stored in a tree structure that encodes and represents the partitioning
  - the partitioning is encoded in the `path` going to each directory

DataFrames and Parquet

All built-in file sources (including Text/CSV/JSON/ORC/Parquet) are able to discover and infer partitioning information automatically

Tables

In a partitioned table, data are usually stored in different directories, with partitioning column values encoded in the path of each partition directory

DataFrames and Parquet

"name":"Michael", "gender":"male", "country":"US"{"name":"Andy", "age":30
"name":"Justin", "age":19, "gender":"female", "country":"CN" path to table
gender=male ... country=US data.parquet country=CN data.parquet ...
gender=female ... country=US data.parquet country=CN data.parquet ...

DataFrames and Parquet

By passing `path/to/table` to `SparkSession.read.parquet` Spark will automatically extract the partitioning information from the paths.

```
root
|-- name: string (nullable = true)
|-- age: long (nullable = true)
|-- gender: string (nullable = true)
|-- country: string (nullable = true)
```

DataFrames and Parquet

Spark partitioning take advantage of this partitioned/folder data structure.
Data are in the leaf of the folder in parquet encoding but the `path/to/table/column=value` pattern can be used to access data in a optimized way when:

- `basePath` is equal to `path/to/table/`

```
result = dfpeople.read_parquet("people/age=34")
```

DataFrames and Parquet

We are taking advantage of the smart partitioning of Spark and the Parquet data format

Plan

Kubernetes + Spark

Kubernetes + Docker = de facto Container Orchestrator

From Kubernetes point of view:

- running Apache Zeppelin[a] inside Kubernetes
- creating your Apache Spark cluster inside Kubernetes
  - from the official Kubernetes organization on GitHub
- referencing the Spark workers in stand-alone mode

[a] Apache Zeppelin is a web-based notebook that enables interactive data analytics.

Kubernetes + Spark

From Spark point of view:

- Apache Mesos
- Yarn
- Standalone
- ...

Why not introducing a new cluster manager?

- Apache Mesos
- Yarn
- Standalone
- Kubernetes
- ...

Spark

Starting with Spark 2.3 Kubernetes can be used to run and manage Spark resources.

- abstractly Kubernetes is "equivalent" conceptually to a Spark Cluster
- usage of the native Kubernetes scheduler added to Spark
  - allocating resources, giving elasticity, simpler interface to manage Apache Spark workloads
- run of the Spark driver and pod on demand
  - no dedicated Spark cluster

Spark

A native Spark Application in Kubernetes acts as a custom controller:

- it creates Kubernetes resources in response to requests made by the Spark scheduler
- Docker images will be used for Spark driver and Spark executors

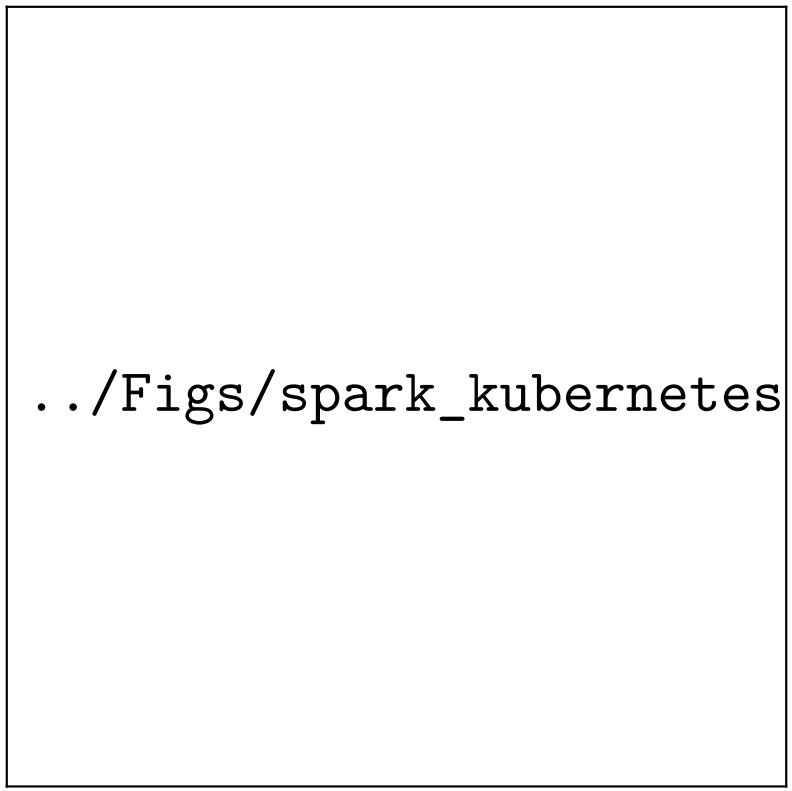
The native approach offers fine-grained management of Spark Applications and enables managing streaming workloads

- Argo, Kubeflow

Architecture

../Figs/spark_kubernetes.png

Architecture



1. submit a Spark application → talk directly to Kubernetes
2. the API server will schedule the driver pod
3. the Spark driver container, the Spark driver, and the Kubernetes Cluster will talk to each other to request and launch Spark executors

Spark

There are two ways to run Spark on Kubernetes:

- using `spark-submit`
- using Spark operator

Spark submit

By using the `spark-submit` client it is possible to submit Spark jobs to Kubernetes

```
spark-submit
```

Spark submit

Delegates the job submission to Spark driver pod on Kubernetes, and finally creates relevant Kubernetes resources by communicating with Kubernetes API server

Spark operator

Spark application configs are written in one place:

- YAML file

Spark app management becomes a lot easier because as the operator comes with:

- tooling for starting/killing
- scheduling apps
- logs capturing

Spark operator

Complex sometimes but rich:

- Details for advanced classes

Objectives

- PySpark
- Arrow for scaling data

Topics

- PySpark DataFrames
- Pandas DataFrames
- Parquet optimized data format
- Arrow for data integration
- Kubernetes for scaling up

Plan

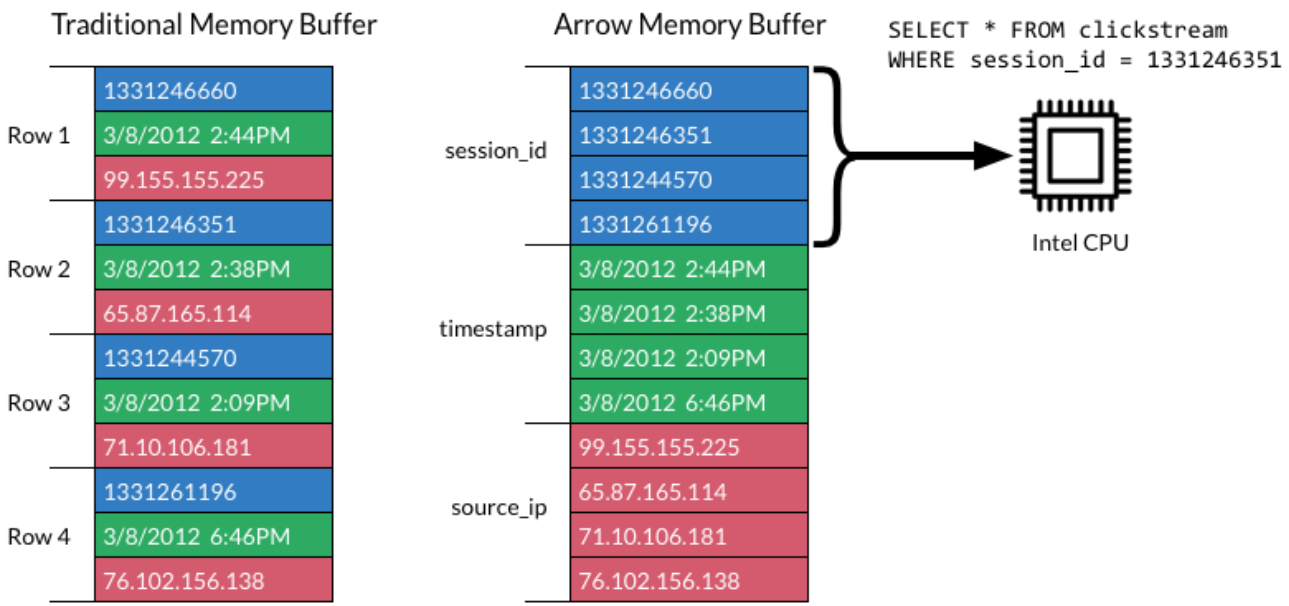
Apache Arrow

- Announced 2016
- Feather package: interoperable DataFrame storage for R and Python, prototype of Arrow format
- Built on lessons of existing DataFrame libraries and databases
- Shared foundation for data analysis
- Designed to take advantage of modern hardware
- `https://arrow.apache.org/`

Arrow

Columnar data independent format

	session_id	timestamp	source_ip
Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138



Apache Arrow

Arrow can be used with:

- Apache Parquet
- Apache Spark
- PySpark
- Pandas
- NumPy
- Other data processing libraries.

Apache Arrow

At the base of the on-disk columnar data formats.

Main objective

- Structure
- Uniform logic
- in-memory data processing

Apache Arrow

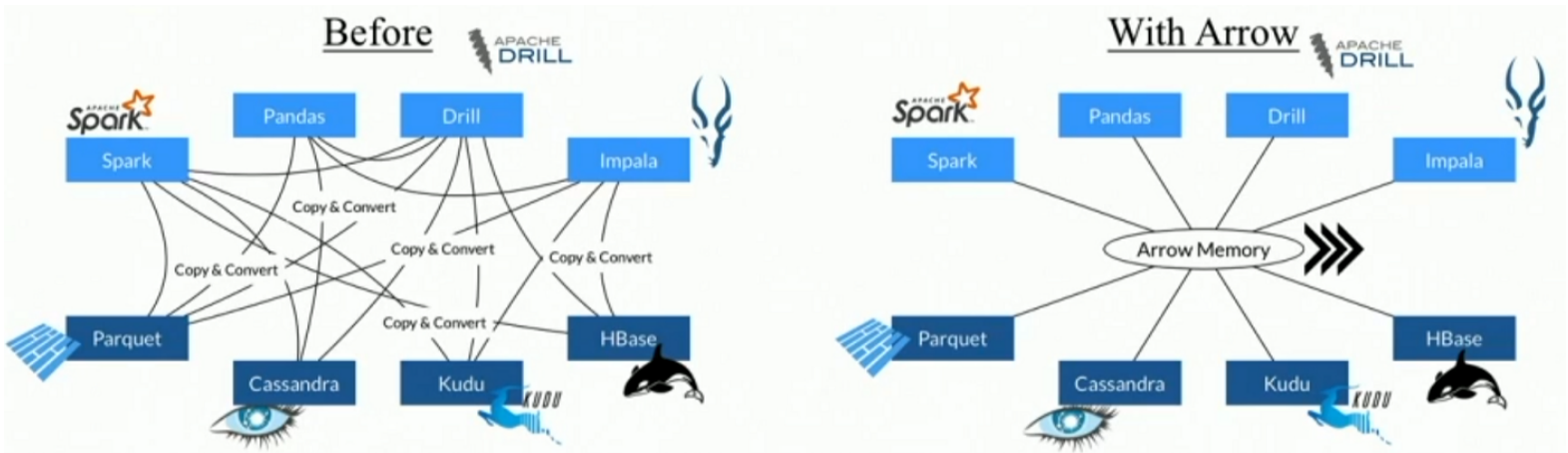
At the base of the on-disk columnar data formats.

Main objective

- Structure
- Uniform logic
- in-memory data processing

Arrow

Accessing large datasets over a network.



Apache Arrow

Arrow Columnar Format

includes a language-agnostic in-memory data structure specification, metadata serialization, and a protocol for serialization and generic data transport.

Arrow

Array

Array or Vector: a sequence of values with known length all having the same type.

Arrays are defined by a few pieces of metadata and data:

- A logical data type.
- A sequence of buffers.
- A length as a 64-bit signed integer. Implementations are permitted to be limited to 32-bit lengths, see more on this below.
- A null count as a 64-bit signed integer.
- An optional dictionary, for dictionary-encoded arrays.

Arrow

Slot

a single logical value in an array of some particular data type

References I


Pandas function.
https://docs.databricks.com/spark/latest/spark-sql/pandas-function-apis.htmll.
Accessed 2021.


pySpark 3.1.1.
https://spark.apache.org/docs/3.1.1/api/python/index.html.
Accessed 2021.