SWAN Use Cases

Diogo C., Prasanth K., Enric T.
On behalf of the SWAN team

https://cern.ch/swan

Feb 3rd, 2021
CERN Academic Training

Physics Analysis

CERN

Physics analysis

> SWAN can be used for interactive physics analysis

- Usually medium/small size analysis (input data ~ a few GBs)
- Last steps of analysis (e.g. nanoAOD, ntuples)

> All the **data** you need

- CERNBox: user files
- EOS: project spaces, experiment data

> All the **software** you need

- ROOT
- Python data science libraries: numpy, pandas, matplotlib, seaborn, ...
- Python ML libraries: tensorflow, keras, pytorch, ...
- R kernel
- Plus anything you can find in an LCG release (~ 500 packages)

CERN

Dimuon spectrum analysis in SWAN

> Physics analysis is one of the most common use cases of SWAN
  - Refer to lecture 1 for examples

> Analysis producing a dimuon spectrum from CMS Open Data
  - Uses ROOT (RDataFrame), NumPy, pandas and matplotlib
  - Accesses public CMS data on EOS
  - Written in C++ and Python
  - Creates interactive plots in a notebook

Open the SWAN training material

CERN

ROOT's RDataFrame: columnar data

pt_x	pt_y	pt_z	theta

columns
or “branches”
can contain any kind
of c++ object

entries
or events
or rows

CERN

RDataFrame: quick how-to

1. Build an RDataFrame object by specifying your dataset

2. Apply a series of transformations to your data
   - filter (e.g. apply some cuts) or
   - define new columns

3. Apply actions to the transformed data to produce results (e.g. fill a histogram)

CERN

RDataFrame: simple code example

1. Build RDataFrame

```
ROOT::RDataFrame d("t", "f.root");
auto h = d.Filter("theta > 0").Histo1D("pt");
h->Draw();
```

2. Cut on theta

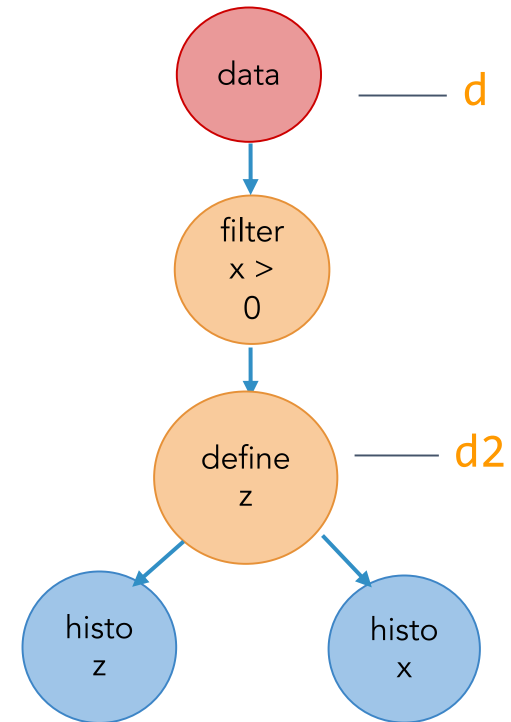
3. Fill histogram with pt

CERN

RDataFrame: analysis as a data-flow

```
// d2 is a new data-frame, a transformed version of d
auto d2 = d.Filter("x > 0")
           .Define("z", "x*x + y*y");

// make multiple histograms out of it
auto hz = d2.Histo1D("z");
auto hx = d2.Histo1D("x");
```



CERN

Spark/NXCals

CERN

Apache Spark

> Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters.

- Unified: provide a unified platform for writing big data applications
  - Data loading, SQL queries, streaming computation and machine learning
- Computing engine: limits the scope to computations
  - Only handles loading data from storage systems and performing computations on it
  - Can be used with wide variety of persistent storage systems, including HDFS, EOS, Amazon S3 and Azure Storage
- Libraries: In addition to standard libraries, spark supports wide array of external libraries (spark-packages.org)

CERN

Spark Applications

> Spark Applications consist of a driver process and a set of executor processes

> **Driver Process**

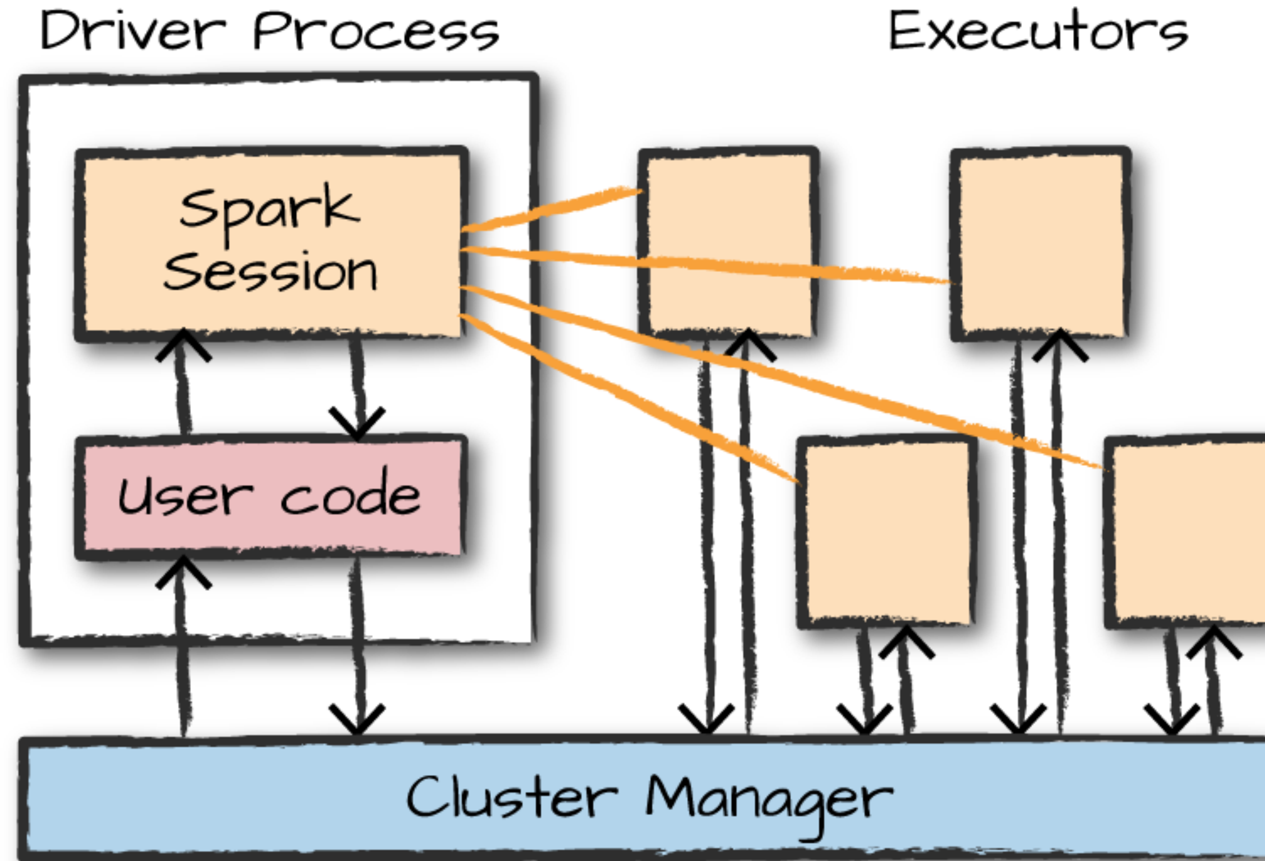
- Maintaining information about the Spark Application
- Responding to a user’s program or input
- And analysing, distributing, and scheduling work across the executors

> **Executor Process**

- Executing code assigned to it by the driver
- And reporting the state of the computation on that executor back to the driver node

CERN

Architecture of Spark Applications



CERN

Spark’s APIs - DataFrame

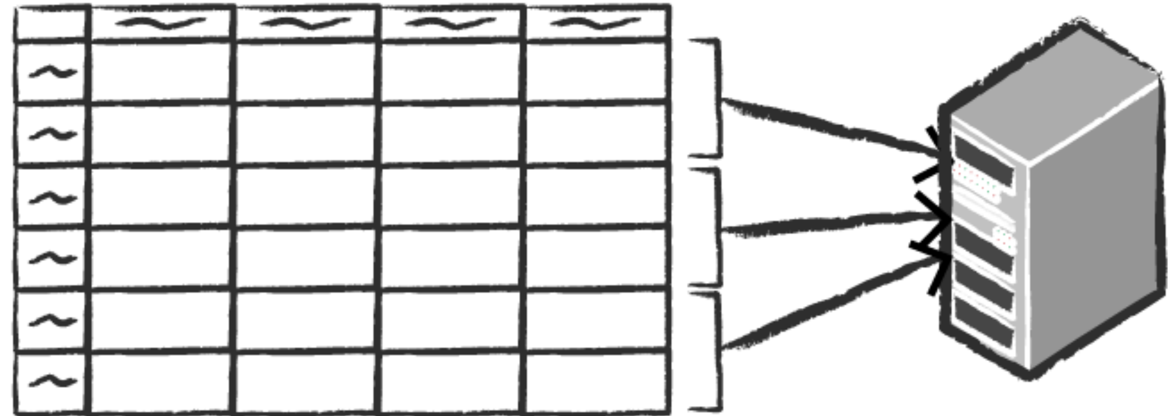
- Spark has two fundamental sets of APIs: the **low-level “unstructured” APIs**, and the **higher-level structured APIs**
- In this talk we only focus on one such higher-level structured APIs - **DataFrame**
- A DataFrame is the most common Structured API and simply represents a **table of data with rows and columns**.
- The list that defines the columns and the types within those columns is called the **schema**.
- You can think of a DataFrame as a spreadsheet with named columns

CERN

Spark’s APIs - DataFrame

Spreadsheet on
a single machine

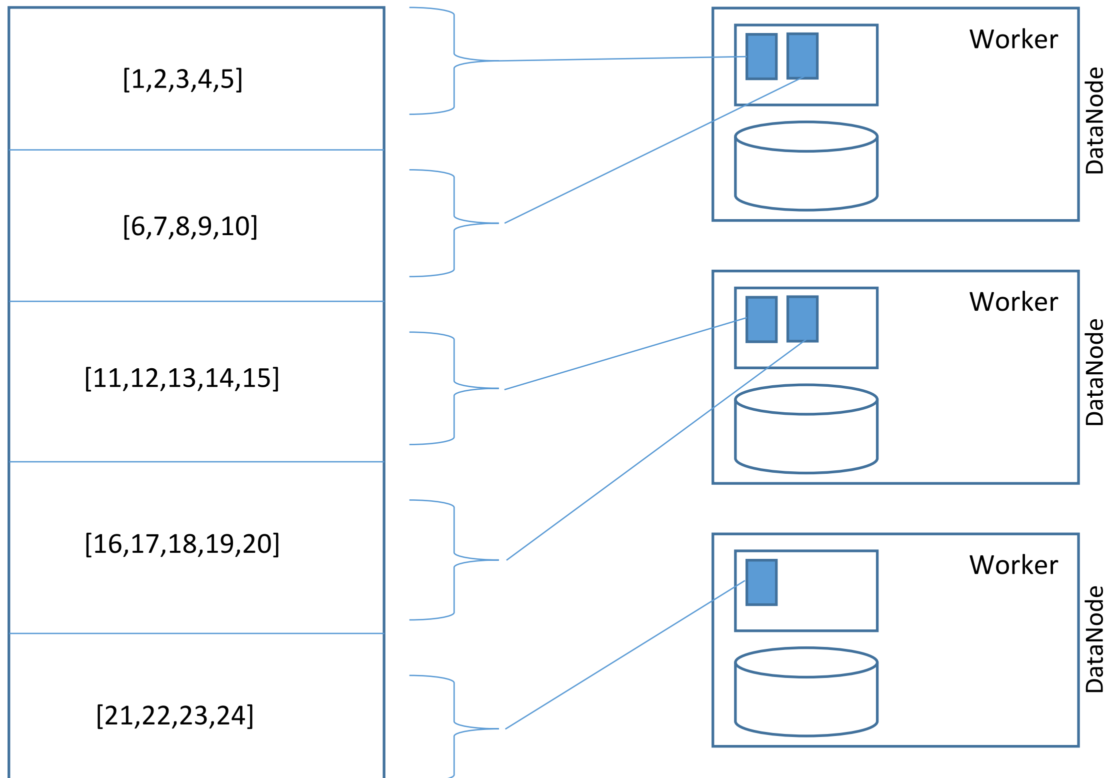
Table or Data Frame
partitioned across servers
in a data center



DataFrame - Partitions

- To allow every executor to perform work in parallel, Spark breaks up the data into chunks called **partitions**.
- A partition is a **collection of rows** that sit on one physical machine in your cluster.
- A DataFrame’s partitions represent how the data is **physically distributed** across the cluster of machines during execution.
- If you have **one partition**, Spark will have a **parallelism of only one**, even if you have thousands of executors.
- If you have **many partitions** but only one executor, Spark will still have a parallelism of only one because there is only one computation resource.

DataFrame - Example

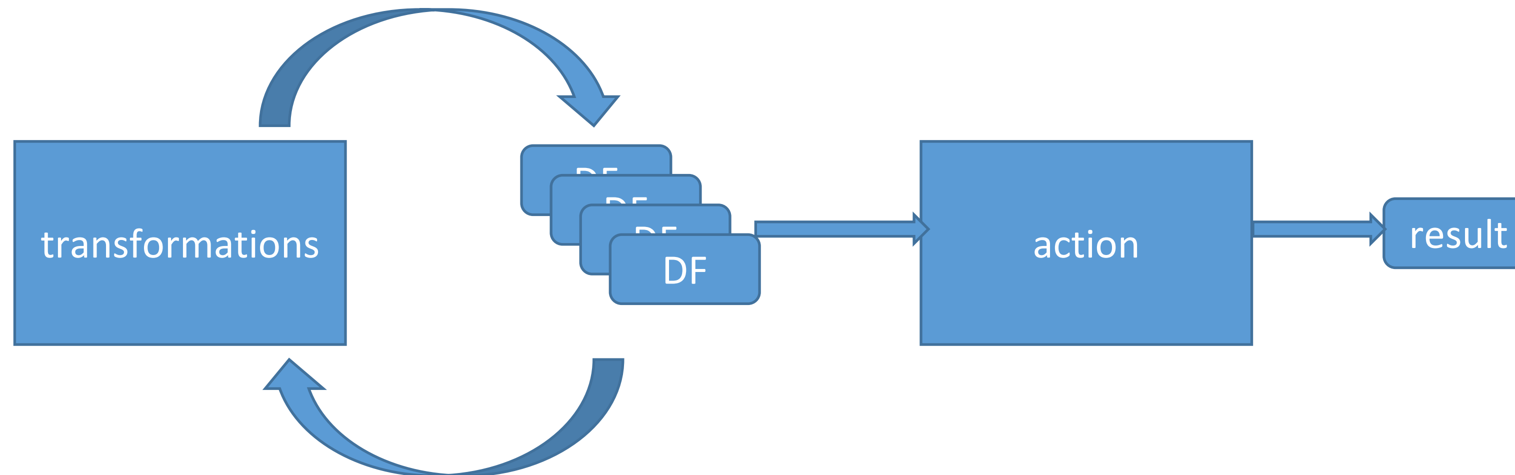


- **Partitioned**: DF is partitioned and distributed across worker nodes of the cluster
- **Immutable**: does not change once created, can only be transformed into new DFs
- **Cacheable**: hold all the data in a persistent storage like memory (preferrable) or disk
- **Lazy Evaluation**: Data inside DF is not available or transformed until an action is executed

CERN

DataFrame – Actions and Transformations

> Two types of **operations** on DFs:
- **Transformations** – lazy operations that return another DF
- **Actions** – operations that trigger computation and return value



CERN

DataFrame - Transformations

> Instructing Spark on how you would like to modify the DF are called transformations.

Example:

```
even_df = df.where("n % 2 = 0")
```

> Transformations are the core of how we ***express logic*** using spark

> Transformations are ***lazy*** and only actions will trigger computations

> Two Types of transformations: ***narrow*** and ***wide***

DataFrame - Actions

> An action instructs Spark to compute a result from a series of transformations

Example: counting number of records in a dataframe:

```
df.count()
```

> There are three kinds of actions

- Actions to view the data in the console
- Actions to collect data to native objects in respective languages
- Actions to write data to storage systems (HDFS, S3, EOS etc)

CERN

Use case – DataCenter Operations

Do the heavylifting in spark and collect aggregated view to panda DF

In [7]:
```
df_loadAvg_pandas = spark.sql("SELECT substring(submitter_host,7,length(submitter_host)) as host, \
                                      avg(body.LoadAvg) as avg, \
                                      hour(from_unixtime(timestamp / 1000, 'yyyy-MM-dd HH:mm:ss')) as hr \
                               FROM loadAvg \
                               WHERE submitter_hostgroup like 'hadoop_ng/nxcals%' \
                               AND dayofmonth(from_unixtime(timestamp / 1000, 'yyyy-MM-dd HH:mm:ss')) = 22 \
                               GROUP BY hour(from_unixtime(timestamp / 1000, 'yyyy-MM-dd HH:mm:ss')), submitter_host")\
                    .toPandas()
```

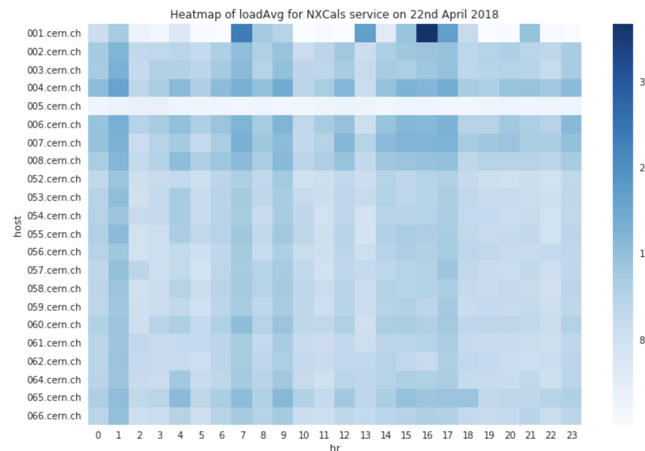
Apache Spark: 54 EXECUTORS 108 CORES Jobs: 1 COMPLETED

Job ID	Job Name	Status	Stages	Tasks	Submission Time	Duration
1	toPandas	COMPLETED	2/2	307 / 307	3 minutes ago	14s

Visualize with seaborn

In [9]:
```
# heatmap of LoadAvg
plt.figure(figsize=(12, 8))
ax = sns.heatmap(df_loadAvg_pandas.pivot(index='host', columns='hr', values='avg'), cmap="Blues")
ax.set_title("Heatmap of loadAvg for NXCals service on 22nd April 2018")
```

Out[9]: Text(0.5,1,u'Heatmap of loadAvg for NXCals service on 22nd April 2018')



https://swan-gallery.web.cern.ch/notebooks/apache_spark1/analytix-hostmetrics-example.html

Use case – NXCals

Inspect data

```
In [2]: df1.select('acqStamp','voltage_18V','current_18V','device','pt100Value').toPandas()[:5]
```

Apache Spark: 2 EXECUTORS 4 CORES Jobs: 1 COMPLETED

Job ID	Job Name	Status	Stages	Tasks	Submission Time	Duration
1	toPandas	COMPLETED	1/1		a few seconds ago	7s

Out[2]:

	acqStamp	voltage_18V	current_18V	device	pt100Value
0	[illegible]	NaN	37.301794	RADMON.PS-10	106.578911
1	[illegible]	NaN	NaN	RADMON.PS-10	107.246742
2	[illegible]	NaN	37.560840	RADMON.PS-10	106.504707
3	[illegible]	20.099066	NaN	RADMON.PS-10	107.068854
4	[illegible]	20.111261	37.698135	RADMON.PS-10	106.578911

Draw a plot with matplotlib

```
In [3]: import matplotlib
        import pandas as pd
        %matplotlib inline
```

```
In [4]: p_df = df1.select('acqStamp','current_18V').toPandas()
        p_df.sort_values(by='acqStamp').plot(pd.to_datetime(p_df['acqStamp'],unit='ns'),'current_18V',figsize=(15,5))
```

Apache Spark: 2 EXECUTORS 4 CORES Jobs: 1 COMPLETED

Job ID	Job Name	Status	Stages	Tasks	Submission Time	Duration
2	toPandas	COMPLETED	1/1		a few seconds ago	3s

```
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f51735d8110>
```

https://swan-gallery.web.cern.ch/notebooks/apache_spark1/NXCals-example.html

References

> Further Study on Spark

- Hadoop: The Definitive Guide, 4th Edition, ISBN: 9781491901632
- Access with CERN account - https://www.oreilly.com/library/view/temporary-access/

> Contact IT Hadoop and Spark Service

- https://cern.service-now.com/service-portal?id=service_element&name=Hadoop-Service

> NXCals Project and API

- http://nxcals-docs.web.cern.ch/current/

CERN

SWAN Use Cases

Thank you

Diogo Castro (diogo.castro@cern.ch)
Enric Tejedor (etejedor@cern.ch)
Prasanth Kothuri (prasanth.kothuri@cern.ch)

CERN