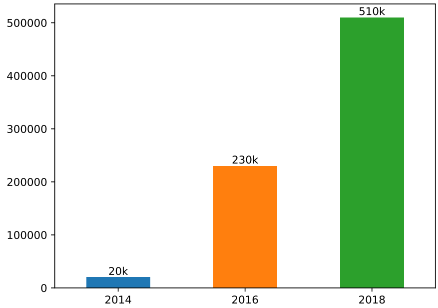
Spark

Chapter-content:

- Spark: the original data model (RDDs)
- Spark: Dataframes
- Spark: Execution
- Pandas Dataframes

Table of content

- Apache Spark: a *distributed computation* framework, like MapReduce.
  Developped at Berkeley by Matei Zaharia, then Apache.
- Similarly to MapReduce, other modules were added to the main API (SQL, Streaming, Machine Learning).
- Natively supports: Scala, Java, Python, and R.
- Written in Scala (needs JVM on every node).



Spark Meetup members

- $>$1000 contributors since 2009.
- 10-100$\times$ faster than Hadoop for some programs.

Spark characteristics

Main features:

- distributed computation (not real time DB)
- computation model remains mostly that of MapReduce : embarassingly parallel job split data into independant tasks (shared-nothing).
- DAG of tasks
- lazy evaluation of operations : allows pipelining

- language-integrated API : computation = functional programming operations over RDDs (or Datasets). RDD = class that represents a collection of data.

- cluster manager: standalone, YARN, Mesos or Kubernetes
- distributed storage system: HDFS or others
- main APIs: SQL, Streaming, Graph, Machine Learning

RDDs

Spark was designed to address 2 shortcomings of MapReduce:

- the limitation to 2 operations Map and Reduce

  (not expressive enough compared to programming or querying languages)

- Data are read/written on disk (rather than RAM), and no way to share sub-tasks

  (slow, unsuited for algorithms that reuse data through successive *iterations*)

Obvious solution: keep data in RAM. But hard to make it fault-tolerant. Spark's approach: RDDs:

- ***Resilient Distributed Dataset*** (RDD): a collection of read-only data, sharded but not necessarily materialized.
- an RDD is defined from one or more sources (DB, RDD, stream) and can be stored on RAM, "persistent" storage on disk, or kept *non-materialized*.
- Spark records the transformations that define the RDD.
- if an RDD partition is lost, Spark will recompute that partition.

an RDD

RDD

```
# Python
lines = sc.textFile("data.txt").map(lambda s: (s, 1))
```

```
// Scala
val lines = sc.textFile("data.txt").map(s => (s, 1))
```

data.txt

Hickory dickory dock.
The mouse ran up the clock.
The clock struck one,
The mouse ran down,
Hickory dickory dock.

materialization of sc.textfile(data.txt)

"Hickory dickory dock."
"The mouse ran up the clock."
"The clock struck one"

"The mouse ran down,"
"Hickory dickory dock."

materialization of lines

("Hickory dickory dock.", 1)
("The mouse ran up the clock.", 1)
("The clock struck one", 1)

("The mouse ran down,", 1)
("Hickory dickory dock.", 1)

Transformations and actions

In short: RDD = high-level abstraction of a computation (its result = materialization of RDD).

2 categories of operations on RDDs: transformations and actions.

- ***transformations*** define RDDs from other RDDs or data sources. Can be composed (and evaluated as pipelines). Evaluation always *lazy*.
- ***actions*** return a value from an RDD to the program (or write to external storage). Each action triggers (by default) the (re)computation of the RDDs it uses. Typically aggregates.

```
# Python
lines = sc.textFile("data.txt")
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
counts.take(2)
counts.count()
```

transformation (map)
transformation (reduceByKey)
action(take)
action(count)

sc : SparkContext, the entry point for RDD API.

actions (RDD)

`.count()`	returns nb of items in RDD
`.reduce(f)`	function must be associative-commutative
`.collect()`	transforms RDD into collection Scala
`.take(n)`	returns array containing n first items
`.first()`	returns first item $\simeq$ take(1)
`.saveAsTextFile(path)`	
`.foreach(f)`	applies function to each item (generally for side-effect)
`...`	

https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

transformations (RDD)

Scala *collection iterators*, and others (set operations, etc.):

`.map(f : A => B)`	
`.filter(f : A => Boolean)`	returns items that evaluate to True
`.flatMap(f : A => list[B])`	concatenates lists obtained through map
`.distinct()`	eliminate duplicates
`.union(otherRDD)`	
`.join(otherRDD)`	on (K,V) and (K,W) pair RDDs, returns (K,(V,W))
`.cogroup(otherRDD)`	like join, but does not flatten: returns (K, Iterator(V), Iterator(W))
`.groupByKey()`	on (key, value) pair RDD
`.reduceByKey(f)`	on (key, value) pair RDD: merges values having same key using f. Merges both before and (unlike groupByKey) after the shuffle
`...`	

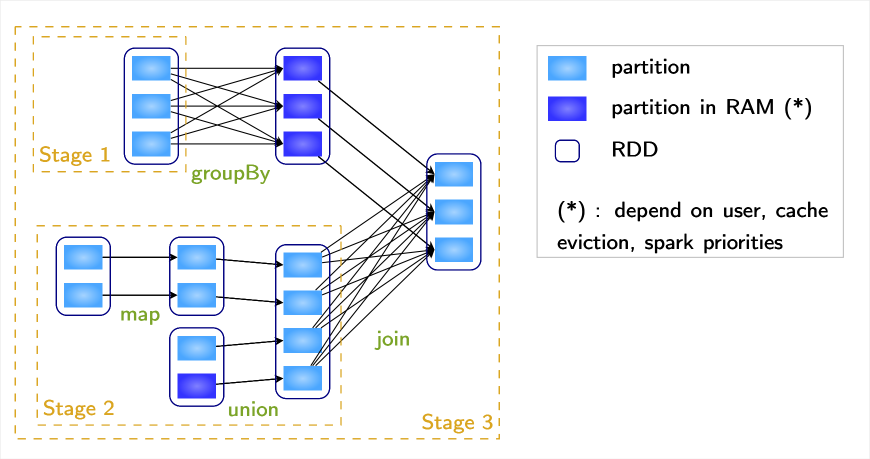
`https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations`

Lazy evaluation, triggered by actions!

Spark Jobs

Like MapReduce, Spark decomposes operations into elementary tasks. Each task confined to a cluster node.

But Spark operations are grouped into more general *stages* delimited by shuffle operations. Spark jobs are DAG of stages.



https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Core API (RDD)

- Methods of RDD interface:

`dependencies`	List of RDDs parents
`getNumPartitions()`	the number of partitions the RDD has been split into
`partitionBy(nb, function)`	specifies how to partition (to overload default)
`repartition(nb)`	specifies how many partition (to overload default)

`toLocalIterator()` iterates over RDD.

- Methods of SparkContext:

`textFile(uri)`	reads a local or HDFS file and returns RDD of strings
`uiWebUrl()`	returns URL of the SparkUI for this context.
	Spark launches a GUI to monitor jobs. First on port 4040, by def
`setLogLevel(level)`	to curtail verbose logs (DEBUG, WARN, ERROR...)

Above is Python API, minor variations for Scala etc.

https://spark.apache.org/docs/latest/api/python/reference/pyspark.html

Storing RDDs

MEMORY_ONLY : (RDD default) Java object in JVM. If no space left in RAM, partition is computed on the fly.

MEMORY_AND_DISK : (DF default) same, but partition stored on disk if no space left

MEMORY_ONLY_SER : serialized storage (1 byte array per partition) *

MEMORY_AND_DISK_SER : ... *

DISK_ONLY : ...

MEMORY_ONLY_2... : specifies how many copies

* Only for Java & Scala : in Python tuples always serialized.

https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

Serialization: requires less space, but access is slower as we need to deserialize.

```
# Python
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
lineLengths.persist()

# lineLengths.cache() is an alias for
# lineLengths.persist(StorageLevel.MEMORY_ONLY)
```

Table of content

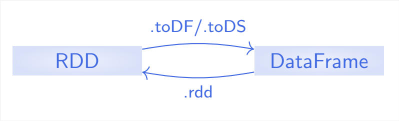
Spark SQL : API Dataset/Dataframe

A ***Dataframe*** is a structured RDD (conforms to a schema): can be viewed as a table with named columns, like in Pandas or R.

Java and Scala API (not Python nor R) also admits statically typed version: Dataset API. *DataFrame* is a *DataSet* of *Row*.

To create a DataFrame : RDD, external source (file, DB table), or object.

```
# Entry point for DF = SparkSession, not SparkContext.
df = spark.read.json("dir/fichier.json")
df.show()     // displays the dataframe
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// +----+-------+
```



Advantage: additional operations, much faster.

Since Spark 2.0, we should use “by default” *Dataset* ($\simeq$ forget about RDD)!

Spark SQL: Advantages of Dataframes/Datasets

Using strict structure as in relational tables allows:

- high level library
- operations similar to SQL
- optimizations similar to DBMS (query plan, code generation)
- can materialize dataframe with column storage :
  (serialization (Arrow) needs 10$\times$ less space than JVM object)
- can perform some operations (filters, sorts, hash) without deserializing
- faster than RDDs for agregation tasks

Since Spark 2.0, use "default" *Dataset* ($\simeq$ forget RDD)!

Spark SQL : examples

- We use a dedicated language (DSL)similar to R or Pandas dataframes. Thanks to DSL, Spark can build AST with *Scala Quasiquotes* (used for query plan optimization and code generation).
- Also supports "real" SQL.

```
df = spark.read.json("dir/fichier.json")
#df.printSchema()
#root
# |-- name: string (nullable = true)
# |-- age: long (nullable = true)

df.filter(df.age > 21).select(df.age+1, df.name).show()

df.select(df['name'], df['age'] + 1)\
  .withColumnRenamed("(age + 1)", "c")\
  .show()
df.groupBy(df.age).count().sort('age',col("count").desc()).show()

// Register a df as a view that can be queried with SQL.
df.createOrReplaceTempView("people")
df2 = spark.sql("SELECT * FROM people")
```

Spark SQL : examples (2)

See https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.join.html
for join syntax.
Also, supports full SQL2003 (including windows).

Transformations and actions (DF)

Transformations : lazy evaluation triggered by actions!

`.orderBy()`	alias pour .sort()
`.filter(condition)`	only keeps lines satisfying condition
`.groupBy(cols)`	returns GroupedDataset on which we compute agregates
`.select(cols)`	projection
`.join(DS, conditions)`	join
`.agg(`f_1`(`col_1`),` f_2`(`col_2`)...)`	
`...`	

Actions : mostly same as RDD, plus:

`.show(n)`	displays n (default 20) first lines as a table
`.head(n)`	returns n first lines as Array[T] (with T=Row)
`...`	

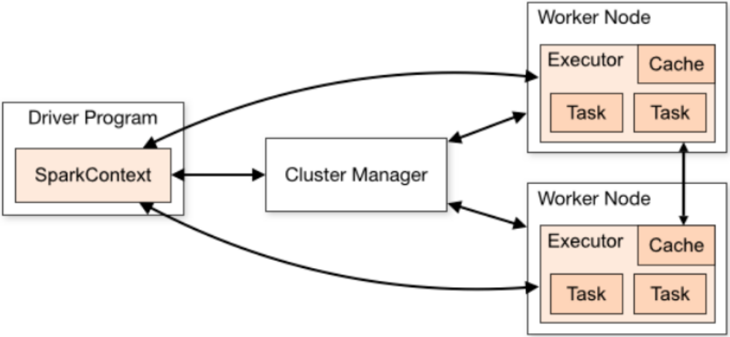
DF do not support `.saveAsTextFile()`: use `.rdd.saveAsTextFile()` or better:

```
df.write().parquet("/path/to/file")#1 file per partition, column format
df.coalesce(1).write.csv("path/to/file")
```

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset

Table of content

Execute a Spark app on a cluster



- Driver acquires ressources (executors) from cluster manager, sends code files (JAR, Python) to executors.
- Driver splits operations (transformations, actions) into tasks and sends tasks to executors.
- Driver communicates with Workers,so they must be connected through network (local as far as possible).

https://spark.apache.org/docs/latest/cluster-overview.html

Spark concepts

Application Your program that uses the Spark API, to run jobs. Uses (1-1 mapping) a dedicated `SparkSession` (incl. `SparkContext`), with its own processes : Driver+Executors.

Ex: a Python program that uses Spark for some computation

Job A computation, triggered by an action.

Ex: `spark.createDataFrame(mylist).distinct().count()`

Stage Each Job is a sequence (DAG) of stages ; a stage is a sequence (DAG) of successive operations, such as map phases, delimited by shuffle operations.

Ex: `spark.createDataFrame(mylist).map(function1).filter(function2)`

Task A unit of work performed by an executor (executing an operation such as map on its part of the data).

Ex: compute 1 partition of df in `df = df_source.map(f)`

Running a Spark program

2 ways to launch a Spark job:

- use an interpreter
  Scala (`spark-shell`), Python (`pyspark`), or R (`SparkR`)
- run standalone application with `spark-submit`.
  Can run `.jar` or `.py` application
  - localely
  - on Spark's native cluster
  - on YARN cluster
  - on Mesos cluster
  - on Kubernetes cluster

https://spark.apache.org/docs/latest/submitting-applications.html

```
# Run application locally on 8 cores
./bin/spark-submit \
  --class MaClasse \
  --master local[8] \   # <master-url>
  /path/to/examples.jar \
  100          # <arguments de l'appli>


# many other possible options...
```

```
# Run application on YARN cluster
./bin/spark-submit \
  --class MaClasse \
  --master yarn \
  --deploy-mode cluster \
  --num-executors 50 \
  --executor-memory 20G \
  /path/to/pi.py \
  100
```

PySpark: how can a Scala app run Python code?

- Need both Python and Java on driver and executors.
- Spark uses Py4J to manipulate objects in JVM from the Python driver.
- On executors, transformations/actions based on Spark functions run entirely in JVM.
- When data (RDD...) must be processed with a native Python function, the code is executed outside the JVM. The executor lazily spawns a Python process, serializes the RDD partitition, sends serialized data and function bytecode to Python process. Python process returns the result to the JVM.
- Communication btw. Python and Java apps induces serialization/deserialization overhead. On driver and on executors.

https://stackoverflow.com/a/61818471

https://medium.com/analytics-vidhya/how-does-pyspark-work-step-by-step-with-pictures-c011402ccd57

https://spark.apache.org/docs/3.2.0/api/python/development/debugging.html

Spark: variables

Each task receives a copy of the variable it nees.
If a task writes, only local copy is modified.

Spark still supports 2 (restricted) kinds of shared variables:

- **Broadcast variables :** one copy per machine (not per task), *read-only*
- **Accumulators :** a global *write-only* variable, modified through a commutative et associative add operation (which can be overloaded)

```
broadcastVar = sc.broadcast([1, 2, 3])
broadcastVar.value  # [1, 2, 3]

accum = sc.accumulator(0)
sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
accum.value    # only usable on driver, not within task
```

https://spark.apache.org/docs/latest/rdd-programming-guide.html#accumulators

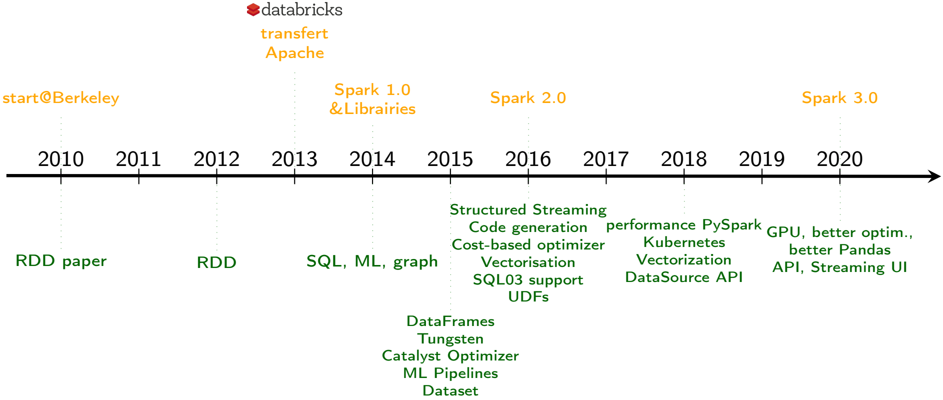
Spark: barrier execution mode

Still experimental, very limited, and poorly documented !

Difference with mapreduce :

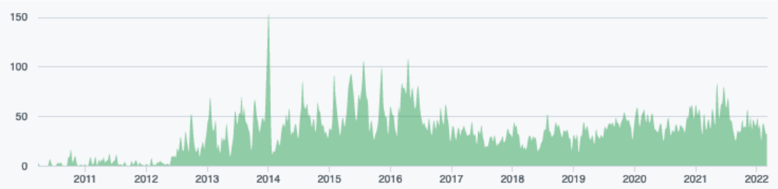
- parallel tasks within a barrier stage are not independent (can communicate)
- all tasks are launched together
- if a task fails, all are restarted
- Spark needs enough threads (one per task)

Spark: timeline

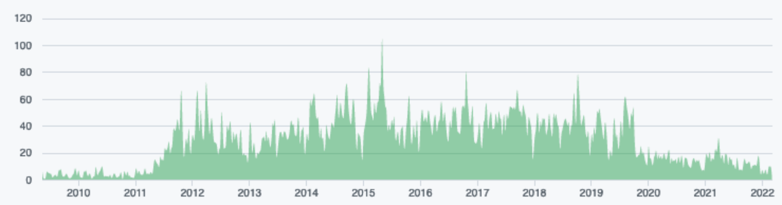


Sources: Databricks et `https://spark.apache.org`

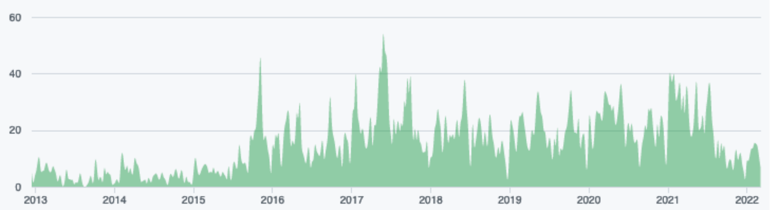
Spark is still highly active.



Spark
github contribs



Hadoop
github contribs



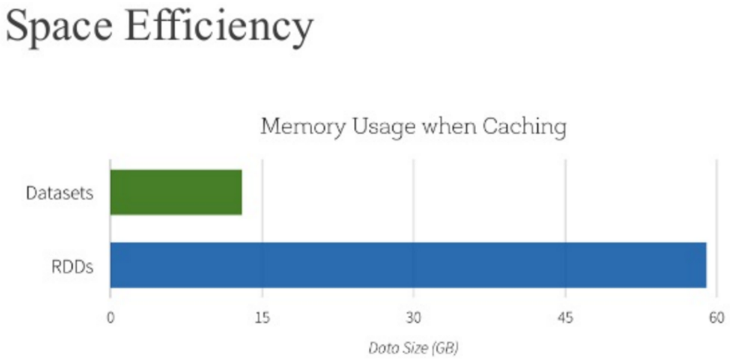
Kafka
github contribs

RDD vs Dataframes: error detection. . .

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

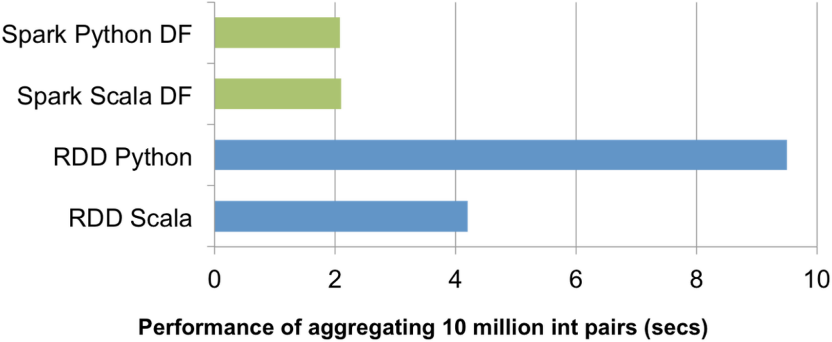
https:
//databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

RDD vs Dataframes: space efficiency. . .



https:
//databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

RDD vs Dataframes: aggregations. . .



https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

Spark: formats for source file...

Spark's file-based data sources

• TEXT	The simplest one with one string column schema
• CSV	Popular for data science workloads
• JSON	The most flexible one for schema changes
• PARQUET	The only one with vectorized reader
• ORC	Popular for shared Hive tables

https://www.slideshare.net/Hadoop_Summit/orc-improvement-in-apache-spark-23-95295487

Spark: Orc sources. . .

```
df.write
  .format("org.apache.spark.sql.execution.datasources.orc")
  .save(path)
```

Read Dataset

```
spark.read
  .format("org.apache.spark.sql.execution.datasources.orc")
  .load(path)
```

Write Dataset

```
CREATE TABLE people (name string, age int)
USING org.apache.spark.sql.execution.datasources.orc
```

Create ORC Table

https://www.slideshare.net/Hadoop_Summit/orc-improvement-in-apache-spark-23-95295487

Alternatives...

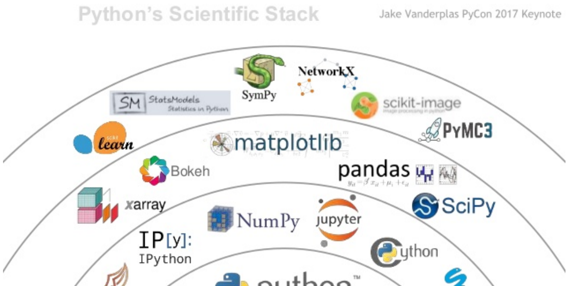
- Python stack (particulièrement Pandas+Dask).

  Pandas is not for distributed computation. Dask adds distributed computation. Still young.
- Apache Flink, Apache Storm

  (Spark arguably richer. Much larger community)
- HPC still relies on other tools

  (OpenMP: shared memory, MPI: distributed memory)
- current issue for Spark: deep learning.

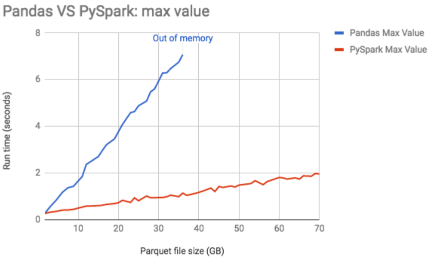
  (can be used as ETL for TensorFlow)

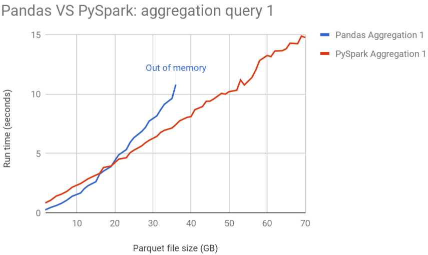


Alternatives: Pandas vs PySpark

On single node.

Machine: 244GB RAM, Xeon E5 2.3GHz
PySpark (2.3.0): local cluster mode, 10GB, 16threads
Pandas: 0.20.3






```
SELECT max(price) FROM sales
```

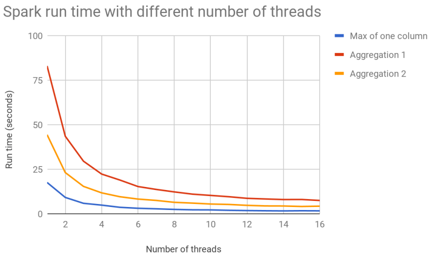
```
SELECT count(distinct customer_sk) FROM sales
```

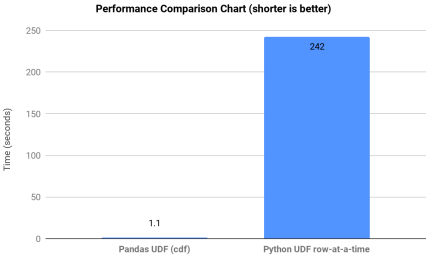
Nb Rows	Parquet size	Uncompressed	Pandas RSS
770MB	39GB	142GB	240GB

Pandas failed for 39GB Parquet, or 60GB CSV (91GB uncompressed).

Alternatives: Pandas vs PySpark (2)

Machine: 244GB RAM, Xeon E5 2.3GHz
PySpark (2.3.0): local cluster mode, 10GB, 16threads
Pandas: 0.20.3





https://databricks.com/blog/2018/05/03/benchmarking-apache-spark-on-a-single-node-machine.html

Bibliography...

Spark:

https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf original RDD paper

https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf original paper (Spark 1)

https://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf original "Dataframe" paper.

https://cdn2.hubspot.net/hubfs/438089/DataBricks_Surveys_-_Content/2016_Spark_Survey/2016_Spark_Infographic.pdf

https://fr.slideshare.net/databricks/jump-start-on-apache-spark-2x-with-databricks quite recent tuto on Spark 2 new features

https://fr.slideshare.net/jozefhabdank/extreme-apache-spark-how-in-3-months-we-created-a-pipeline-that-can-process-25-billion-rows-a-day

Cours (Hadoop, Spark etc):

https://openclassrooms.com/courses/realisez-des-calculs-distribues-sur-des-donnees-massives?status=published C.Hudelot et al. à Centrale-Supélec

http://b3d.bdpedia.fr/ P.Rigaux et al. au CNAM. Complet et précis techniquement.

Table of content

Dataframes. . .

Pandas: data manipulation library in python.

Pandas library in Python

```python
import pandas as pd

# Dataframe: similar to SQL table
df = pd.DataFrame({'animal': ['Raccoon', 'Rabbit', 'Cat', 'Raccoon', 'Human'],
                   'poids': [4, 2.5, 3.3, 4.4, 62],
                   'poids_cerveau': [39, 12.1, 25.6, 39.4, 1320]},
                   columns=['animal', 'poids', 'poids_cerveau'])

# projection:
df1 = df[['animal','poids']]
df2 = df[['animal','poids_cerveau']]

# selection:
df1[df1['poids']>4]
df1[(df1['poids']>4) & (df1['poids']<10)]

# join:
pd.merge(df1, df2)

df.groupby('animal').median()
df.groupby('animal').aggregate({'poids':['max','min'], 'poids_cerveau': 'mean'})
```

Pandas Dataframes

```
>>> df

    animal  poids  poids_cerveau
0  Raccoon    4.0           39.0
1   Rabbit    2.5           12.1
2      Cat    3.3           25.6
3  Raccoon    4.4           39.4
4    Human   62.0         1320.0
```

```
>>> df[['animal','poids']]

    animal  poids
0  Raccoon    4.0
1   Rabbit    2.5
2      Cat    3.3
3  Raccoon    4.4
4    Human   62.0
```

```
>>> df.groupby('animal').aggregate({
          'poids':['max','min'],
          'poids_cerveau': 'mean'})

        poids_cerveau poids
                 mean   max   min
animal
Cat              25.6   3.3   3.3
Human          1320.0  62.0  62.0
Rabbit           12.1   2.5   2.5
Raccoon          39.2   4.4   4.0
```