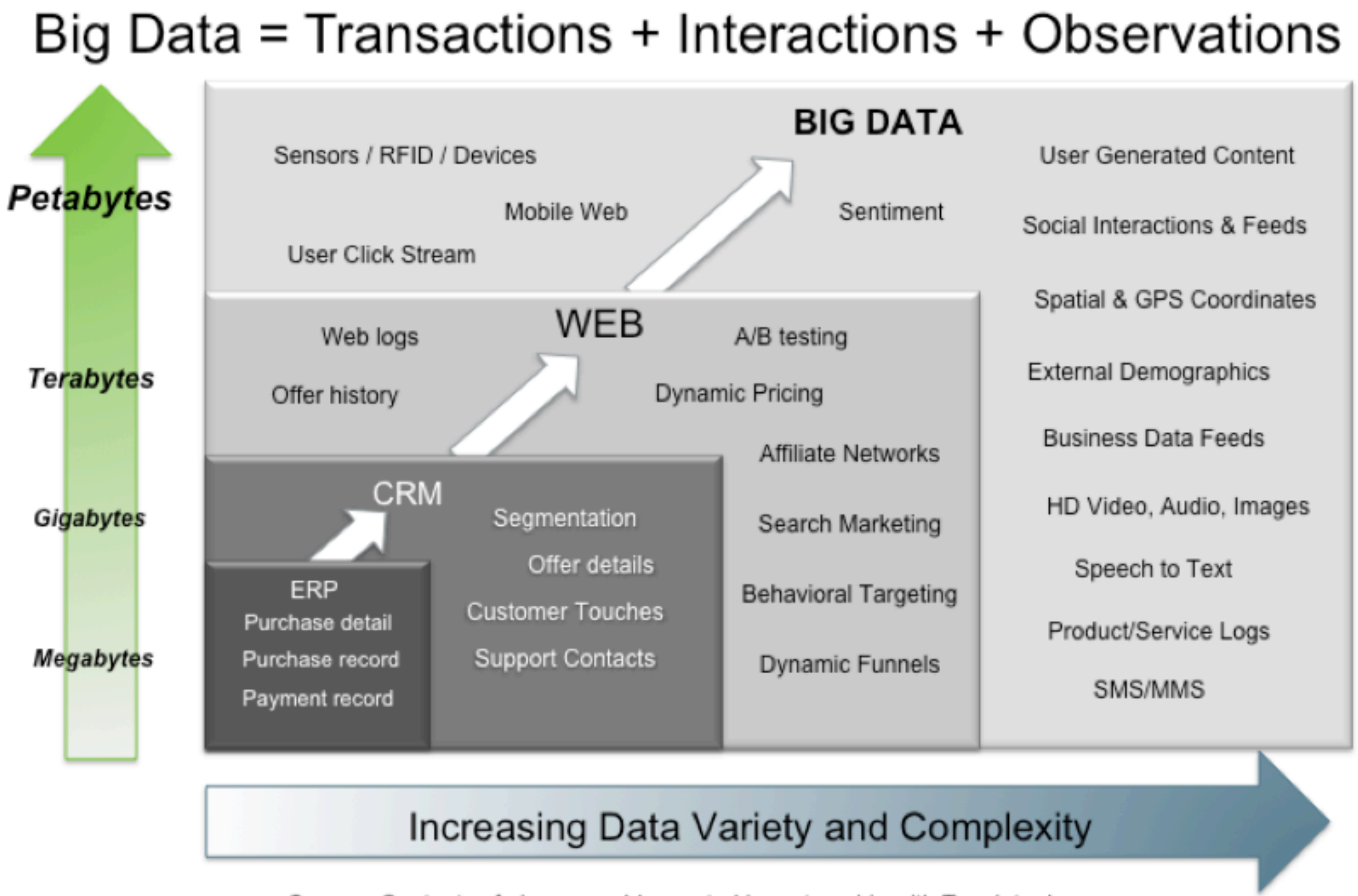
# Data Science in the Wild

## Lecture 11: In-memory Parallel Processing in Spark

Eran Toch

CORNELL TECH

# The Scale of Big Data




***Source**: Contents of above graphic created in partnership with Teradata, Inc.*

# Agenda

1. Spark
2. Spark DataFrames
3. Spark SQL
4. Machine Learning on Spark
5. ML Pipelines

# Technological Architecture

In Memory Data Flow
APACHE Spark

Data Warehouse
HIVE

NoSQL
APACHE HBASE

Scripting
Pig

Processing

MapReduce / YARN

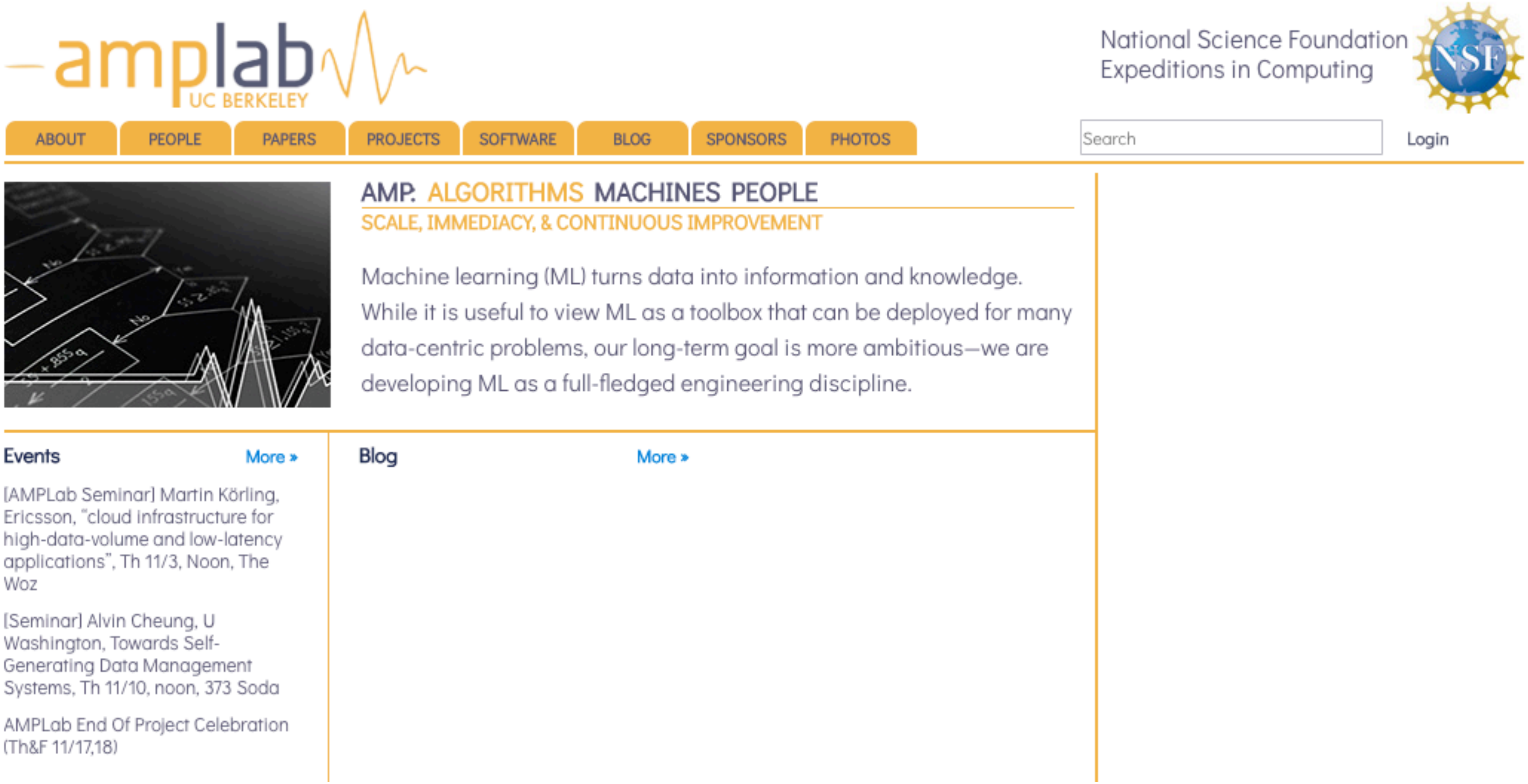
Storage
hadoop

Hadoop Distributed File System (HDFS)

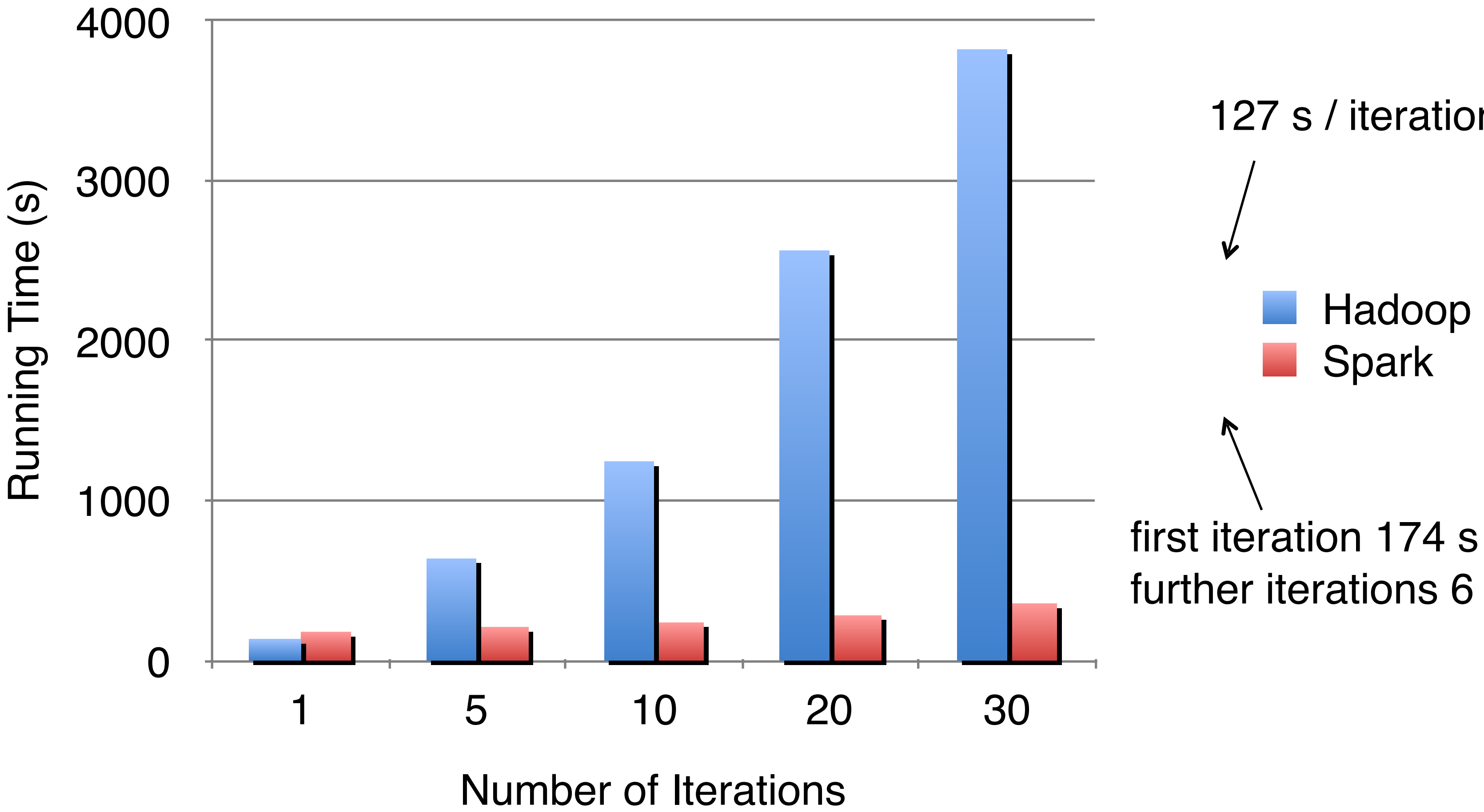
# Motivation

- Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a working set of data:
  - Iterative algorithms (many in machine learning)
  - Interactive data mining tools (R, Excel, Python)
- Spark makes working sets a first-class concept to efficiently support these apps

# History


amplab
UC BERKELEY
National Science Foundation
Expeditions in Computing
NSF
ABOUT
PEOPLE
PAPERS
PROJECTS
SOFTWARE
BLOG
SPONSORS
PHOTOS
Search
Login
AMP: ALGORITHMS MACHINES PEOPLE
SCALE, IMMEDIACY, & CONTINUOUS IMPROVEMENT
Machine learning (ML) turns data into information and knowledge. While it is useful to view ML as a toolbox that can be deployed for many data-centric problems, our long-term goal is more ambitious—we are developing ML as a full-fledged engineering discipline.
Events
More »
[AMPLab Seminar] Martin Körling, Ericsson, "cloud infrastructure for high-data-volume and low-latency applications", Th 11/3, Noon, The Woz
[Seminar] Alvin Cheung, U Washington, Towards Self-Generating Data Management Systems, Th 11/10, noon, 373 Soda
AMPLab End Of Project Celebration (Th&F 11/17,18)
Blog
More »

# Logistic Regression Performance




127 s / iteration

first iteration 174 s
further iterations 6 s

# Spark

- Provides distributed memory abstractions for clusters to support apps with working sets
- Retain the attractive properties of MapReduce:
    - Fault tolerance (for crashes & stragglers)
    - Data locality
    - Scalability

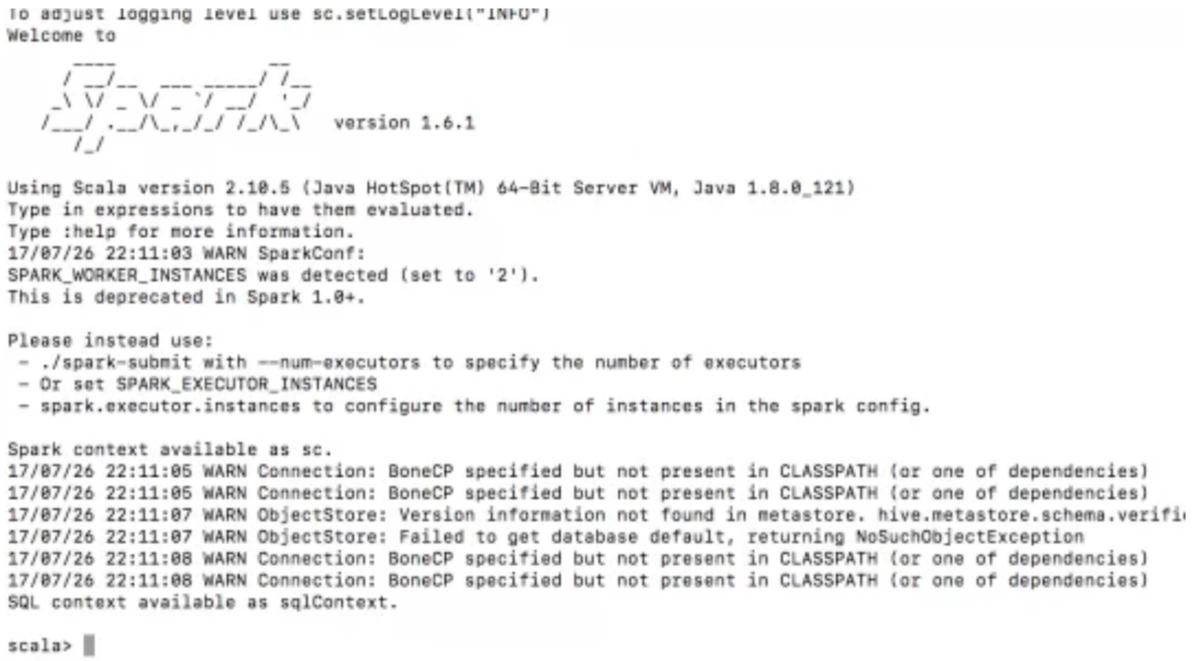
# Languages

- Scala: Spark is primarily written in Scala, making it Spark's "default" language. This book will include Scala code examples wherever relevant.
- Java
- Python
- SQL: Spark supports ANSI SQL 2003 standard
- R: Spark has two commonly used R libraries, one as a part of Spark core (SparkR) and another as an R community driven package (sparklyr

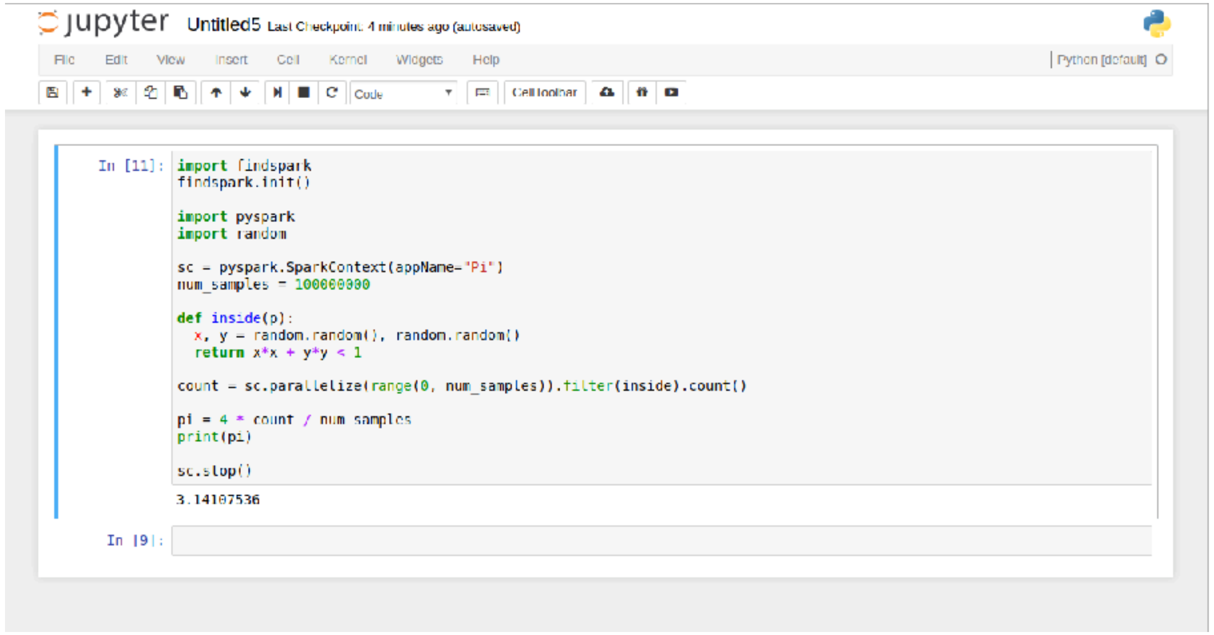
# Running Spark

- Self Hosted: You can set up a cluster yourself using bare metal machines or virtual machines
- Cloud Providers: Most cloud providers offer Spark clusters: AWS has EMR and GCP has DataProc.
- Vendor Solutions: Companies including Databricks and Cloudera provide Spark solutions

# Environments



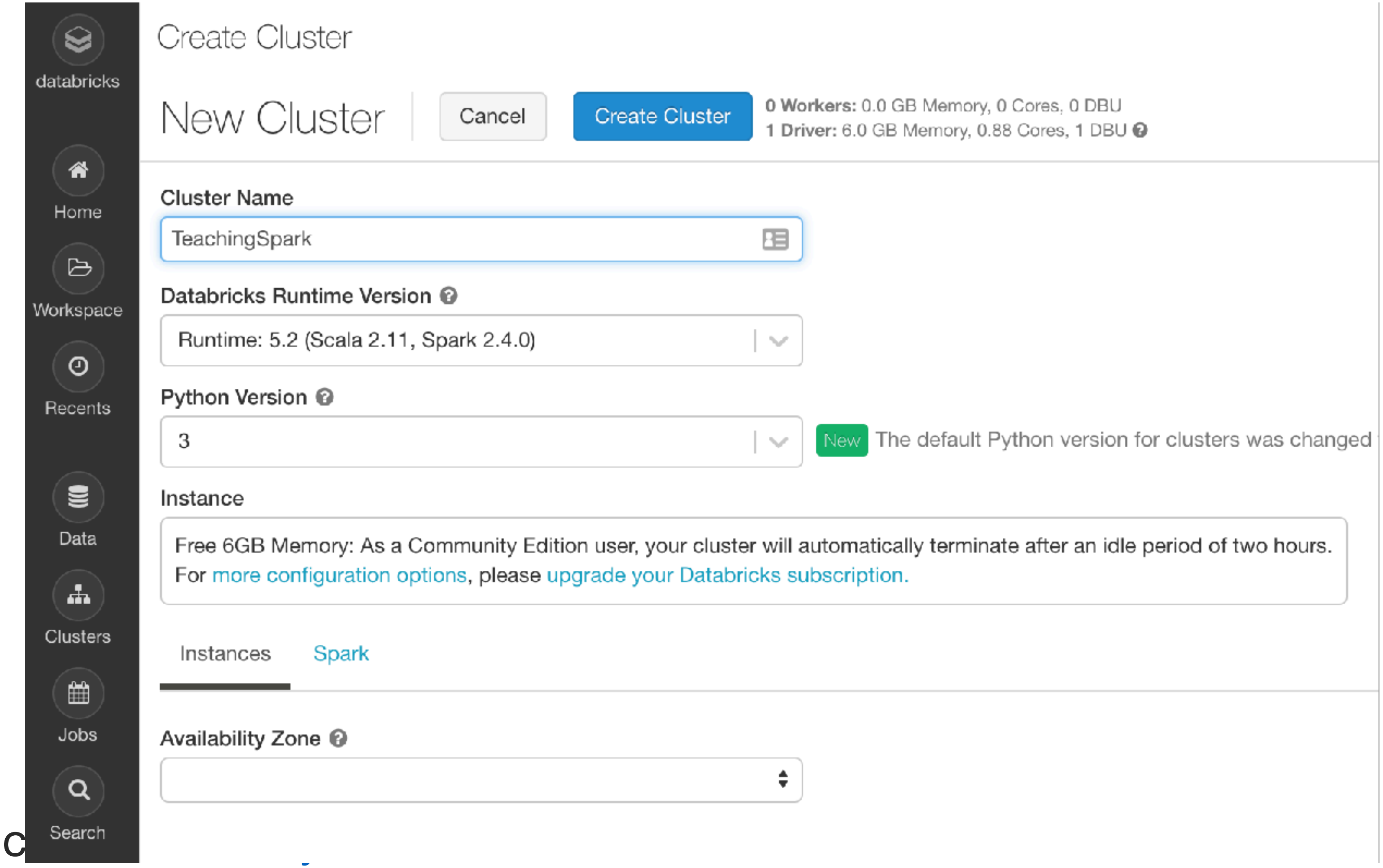

Spark Shell



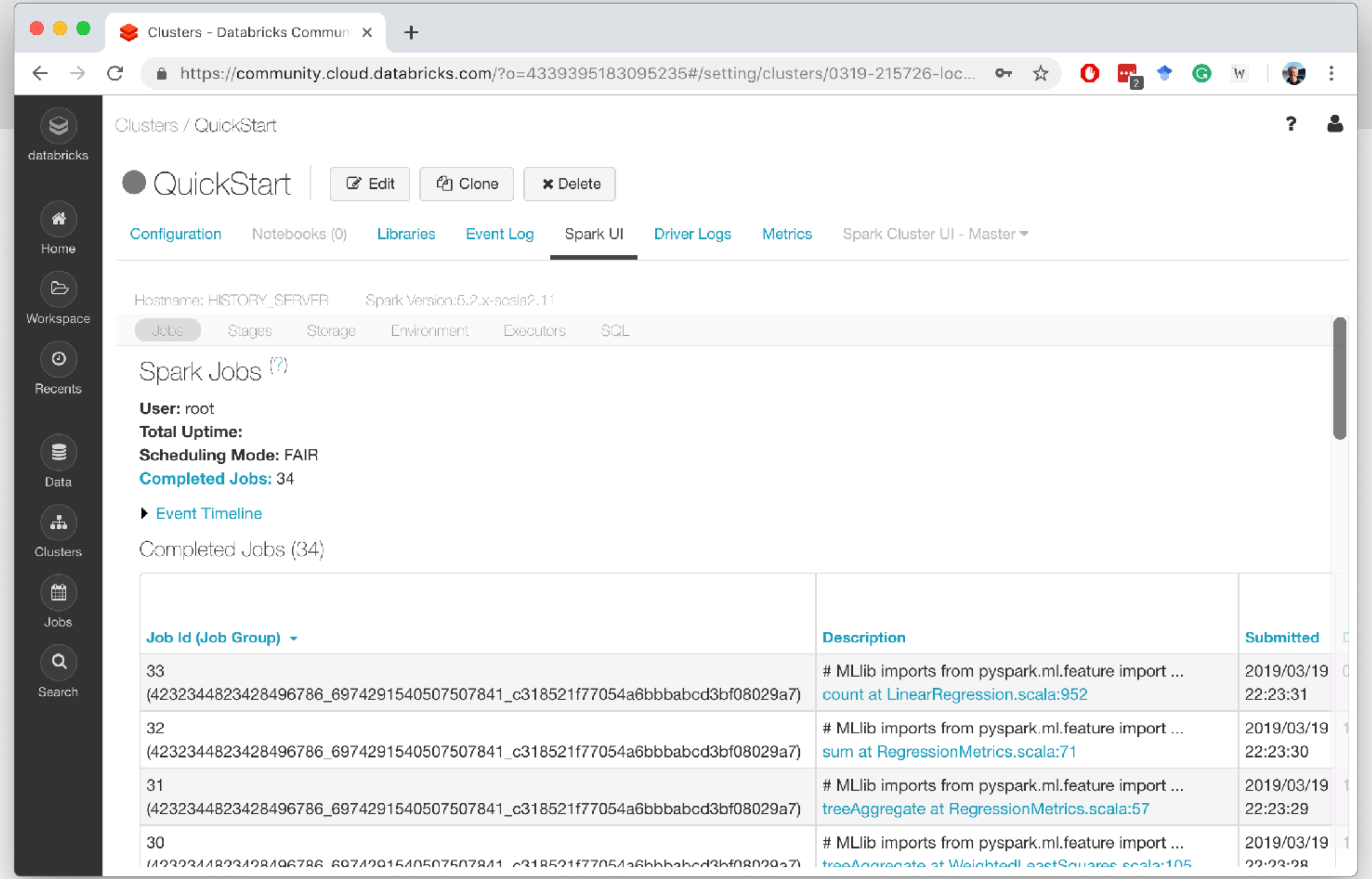

Jupyter Integration

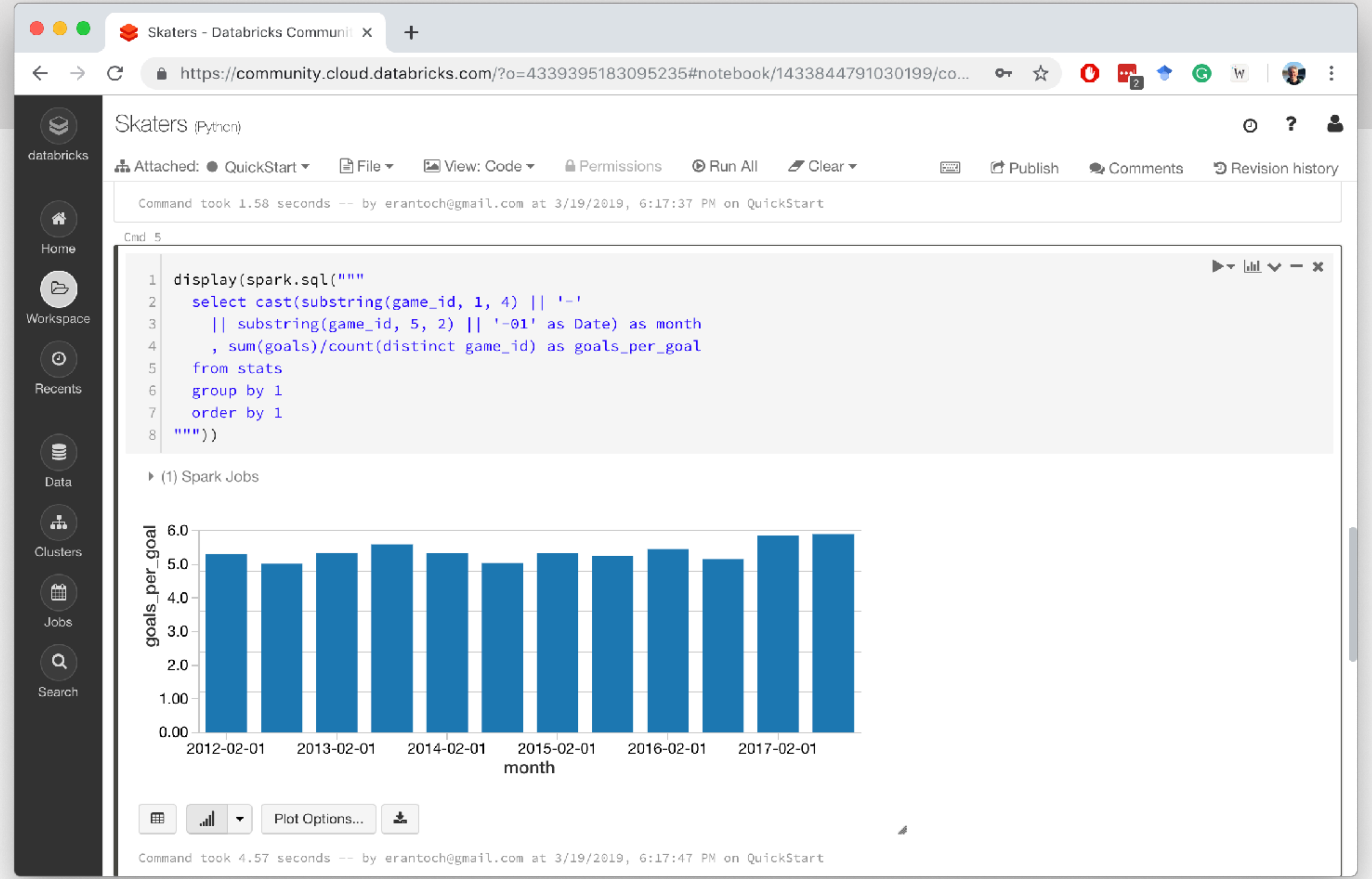
https://medium.com/@yajieli/installing-spark-pyspark-on-mac-and-fix-of-some-common-errors-355a9050f735

# DataBricks

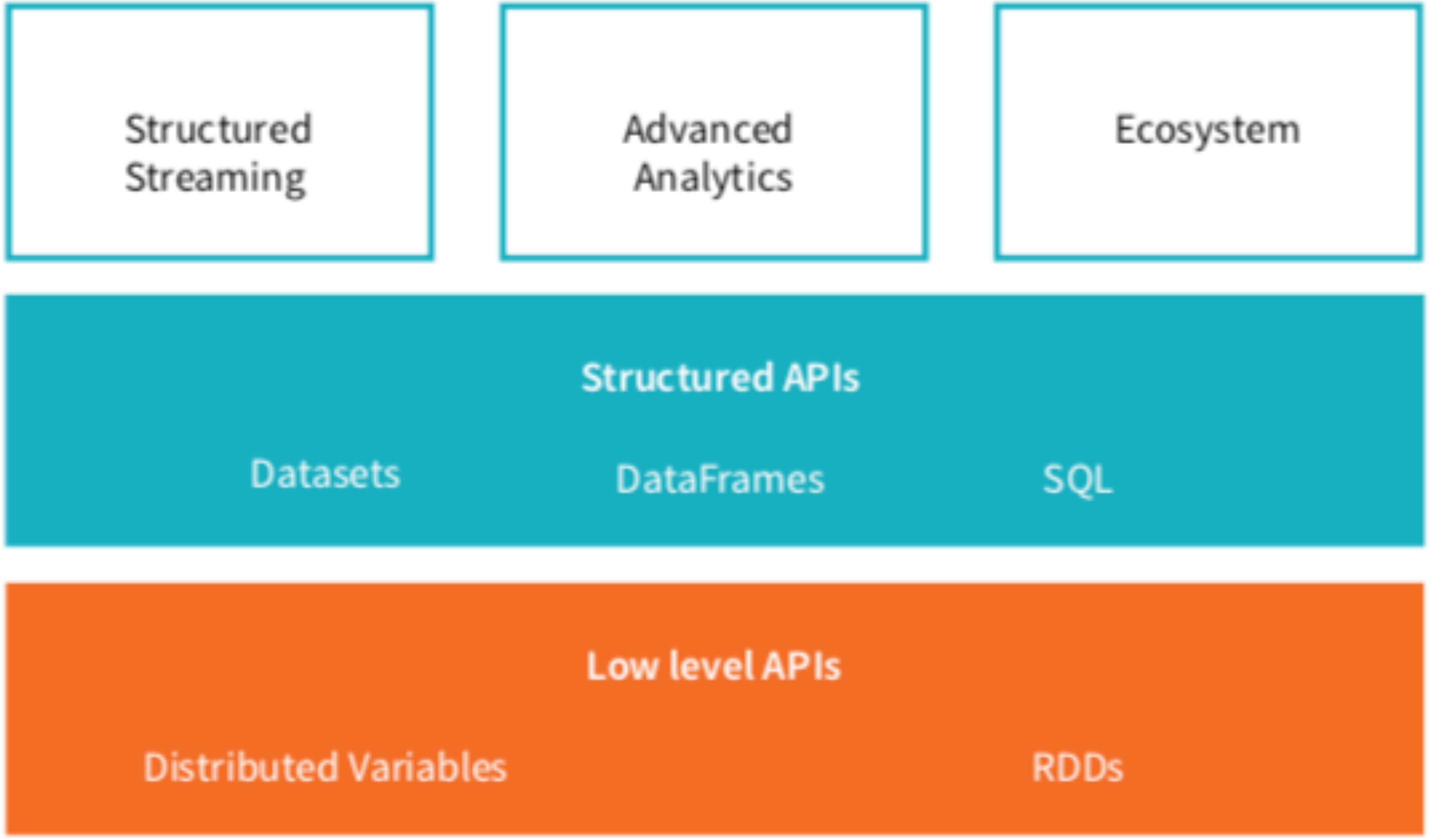



Databric


Clusters - Databricks Commun
https://community.cloud.databricks.com/?o=4339395183095235#/setting/clusters/0319-215726-loc...
databricks
Home
Workspace
Recents
Data
Clusters
Jobs
Search
Clusters / QuickStart
QuickStart
Edit
Clone
Delete
Configuration
Notebooks (0)
Libraries
Event Log
Spark UI
Driver Logs
Metrics
Spark Cluster UI - Master
Hostname: HISTORY_SERVER
Spark Version:5.2.x-scala2.11
Jobs
Stages
Storage
Environment
Executors
SQL
Spark Jobs (?)
User: root
Total Uptime:
Scheduling Mode: FAIR
Completed Jobs: 34
Event Timeline
Completed Jobs (34)
Job Id (Job Group)
Description
Submitted
33 (4232344823428496786_6974291540507507841_c318521f77054a6bbbabcd3bf08029a7)
# MLlib imports from pyspark.ml.feature import ...
count at LinearRegression.scala:952
2019/03/19 22:23:31
32 (4232344823428496786_6974291540507507841_c318521f77054a6bbbabcd3bf08029a7)
# MLlib imports from pyspark.ml.feature import ...
sum at RegressionMetrics.scala:71
2019/03/19 22:23:30
31 (4232344823428496786_6974291540507507841_c318521f77054a6bbbabcd3bf08029a7)
# MLlib imports from pyspark.ml.feature import ...
treeAggregate at RegressionMetrics.scala:57
2019/03/19 22:23:29
30
# MLlib imports from pyspark.ml.feature import ...
2019/03/19


Skaters - Databricks Communit
https://community.cloud.databricks.com/?o=4339395183095235#notebook/1433844791030199/co...
databricks
Home
Workspace
Recents
Data
Clusters
Jobs
Search
Skaters (Python)
Attached: QuickStart
File
View: Code
Permissions
Run All
Clear
Publish
Comments
Revision history
Command took 1.58 seconds -- by erantoch@gmail.com at 3/19/2019, 6:17:37 PM on QuickStart
Cmd 5
display(spark.sql("""
  select cast(substring(game_id, 1, 4) || '-'
    || substring(game_id, 5, 2) || '-01' as Date) as month
    , sum(goals)/count(distinct game_id) as goals_per_goal
  from stats
  group by 1
  order by 1
"""))
(1) Spark Jobs
goals_per_goal
6.0
5.0
4.0
3.0
2.0
1.00
0.00
2012-02-01
2013-02-01
2014-02-01
2015-02-01
2016-02-01
2017-02-01
month
Plot Options...
Command took 4.57 seconds -- by erantoch@gmail.com at 3/19/2019, 6:17:47 PM on QuickStart

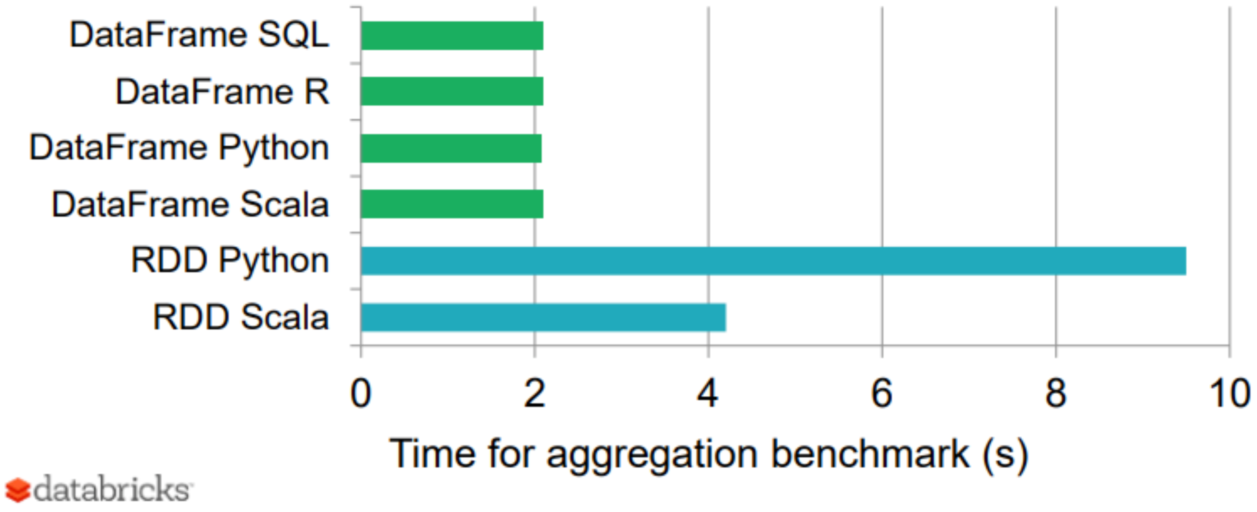
# Spark Structure




https://pages.databricks.com/gentle-intro-spark.html

# Spark Programming

- Resilient Distributed Datasets (RDDs)
    - Phased out
- DataFrames
- Spark SQL




https://towardsdatascience.com/sql-at-scale-with-apache-spark-sql-and-dataframes-concepts-architecture-and-examples-c567853a702f

# <2> Spark DataFrames

# Data Frames

- A DataFrame is the most common Structured API and simply represents a table of data with rows and columns
- The list of columns and the types in those columns the schema
- A Spark DataFrame can be parallelized across thousands of computers

| game_id | player_id | team_id | timeOnIce | assists | goals | shots | hits | powerPlayGoals | powerPlayAssists | penaltyMinutes | fac |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2012030221 | 8471958 | 3 | 1925 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 |
| 2012030221 | 8471339 | 3 | 1597 | 1 | 0 | 2 | 3 | 0 | 0 | 0 | 0 |
| 2012030221 | 8471873 | 3 | 1695 | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 0 |
| 2012030221 | 8473432 | 3 | 957 | 0 | 0 | 3 | 5 | 0 | 0 | 2 | 0 |
| 2012030221 | 8470192 | 3 | 859 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 |
| 2012030221 | 8474151 | 3 | 1919 | 0 | 1 | 3 | 5 | 0 | 0 | 0 | 0 |
| 2012030221 | 8475184 | 3 | 697 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 |
| 2012030221 | 8475186 | 3 | 933 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 |

# 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 our cluster
- Programming with Dataframes means that we specify high-level transformations of data and Spark determines how this work will actually execute on the cluster.
- Lower level APIs do exist (via the Resilient Distributed Datasets interface)

DataFrame with 4 partitions

# Loading Data

```
file_location = "/FileStore/tables/game_skater_stats.csv"
df = spark.read.format("csv").option("inferSchema",
        True).option("header", True).load(file_location)
display(df)
```

▸ (3) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [game_id: integer, player_id: integer ... 20 more fields]

| game_id | player_id | team_id | timeOnIce | assists | goals | shots | hits | powerPlayGoals | powerPlayAssists | penaltyMinutes |
|---|---|---|---|---|---|---|---|---|---|---|
| 2012030221 | 8471958 | 3 | 1925 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| 2012030221 | 8471339 | 3 | 1597 | 1 | 0 | 2 | 3 | 0 | 0 | 0 |
| 2012030221 | 8471873 | 3 | 1695 | 0 | 0 | 1 | 2 | 0 | 0 | 0 |
| 2012030221 | 8473432 | 3 | 957 | 0 | 0 | 3 | 5 | 0 | 0 | 2 |
| 2012030221 | 8470192 | 3 | 859 | 0 | 0 | 1 | 0 | 0 | 0 | 2 |
| 2012030221 | 8474151 | 3 | 1919 | 0 | 1 | 3 | 5 | 0 | 0 | 0 |
| 2012030221 | 8475184 | 3 | 697 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| 2012030221 | 8475186 | 3 | 933 | 0 | 0 | 1 | 0 | 0 | 0 | 2 |

Showing the first 1000 rows.

Command took 3.25 seconds -- by erantoch@gmail.com at 3/19/2019, 6:14:21 PM on QuickStart

https://towardsdatascience.com/a-brief-introduction-to-pyspark-ff4284701873

# Ways to Read Data

- Reading from CSV is done in an “eager” mode: the data is immediately loaded to the memory
- Lazy initialization is generally preferred with Spark
- It is possible with parquet files

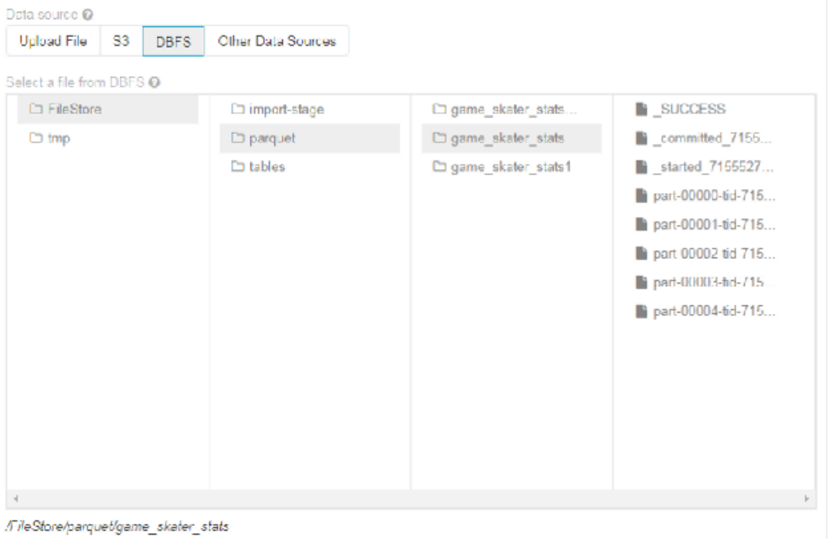
```
df = spark.read .load(“s3a://my_bucket/game_skater_stats/*.parquet")
```

# Writing Data

- Writing to Parquet:

```
# DBFS (Parquet)
df.write.save(‘/FileStore/parquet/game_stats',format='parquet')

# S3 (Parquet)
df.write.parquet("s3a://my_bucket/game_stats", mode=“overwrite")
```




- Writing to CSV:

```
# DBFS (CSV)
df.write.save('/FileStore/parquet/game_stats.csv', format='csv')
# S3 (CSV)
df.coalesce(1).write.format("com.databricks.spark.csv")
   .option("header", "true").save("s3a://my_bucket/
game_sstats.csv")
```

# Schemas

- df.printSchema()
-

```
root
 |-- game_id: integer (nullable = true)
 |-- player_id: integer (nullable = true)
 |-- team_id: integer (nullable = true)
 |-- timeOnIce: integer (nullable = true)
 |-- assists: integer (nullable = true)
 |-- goals: integer (nullable = true)
 |-- shots: integer (nullable = true)
 |-- hits: integer (nullable = true)
 |-- powerPlayGoals: integer (nullable = true)
 |-- powerPlayAssists: integer (nullable = true)
 |-- penaltyMinutes: integer (nullable = true)
 |-- faceOffWins: integer (nullable = true)
 |-- faceoffTaken: integer (nullable = true)
 |-- takeaways: integer (nullable = true)
 |-- giveaways: integer (nullable = true)
 |-- shortHandedGoals: integer (nullable = true)
 |-- shortHandedAssists: integer (nullable = true)
 |-- blocked: integer (nullable = true)
 |-- plusMinus: integer (nullable = true)
 |-- evenTimeOnIce: integer (nullable = true)
 |-- shortHandedTimeOnIce: integer (nullable = true)
 |-- powerPlayTimeOnIce: integer (nullable = true)
```

# Operations

```
unionDF = df1.unionAll(df2)
display(unionDF)

df = unionDF.select(explode("employees").alias("e"))
explodeDF = df.selectExpr("e.firstName", "e.lastName", "e.email", "e.salary")

filterDF = explodeDF.filter(explodeDF.firstName == "xiangrui").sort(explodeDF.lastName)
display(filterDF)
```

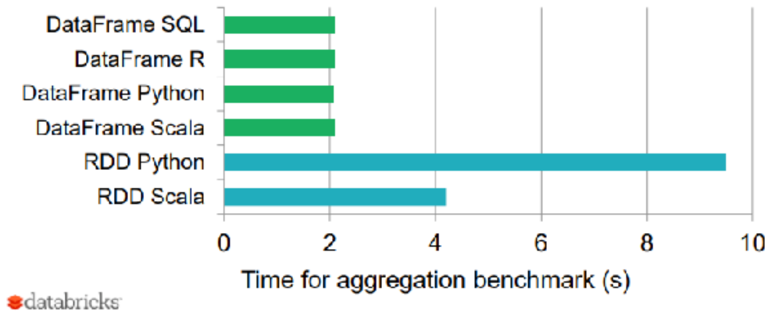
- Replace null values with -- using DataFrame Na function

```
nonNullDF = explodeDF.fillna("--")
display(nonNullDF)
```

# <4> Spark SQL

# Spark SQL

- SQL: Structure Query Language was defined for relational databases
- Spark SQL is borrowed from HIVE's implementation of a limited language for Hadoop-based datasets
- Spark SQL provides a DataFrame API that can perform relational operations on both external data sources and Spark's built-in distributed collections




https://towardsdatascience.com/sql-at-scale-with-apache-spark-sql-and-dataframes-concepts-architecture-and-examples-c567853a702f

# Running SQL

- SQL runs as a “language inside language” model
- Databases and tables can be created independently or from DataFrames

spark.sql("show databases")

```
df.createOrReplaceTempView("stats")
display(spark.sql("""
  select player_id, sum(1) as games, sum(goals) as goals
  from stats
  group by 1
  order by 3 desc
  limit 5
"""))
```

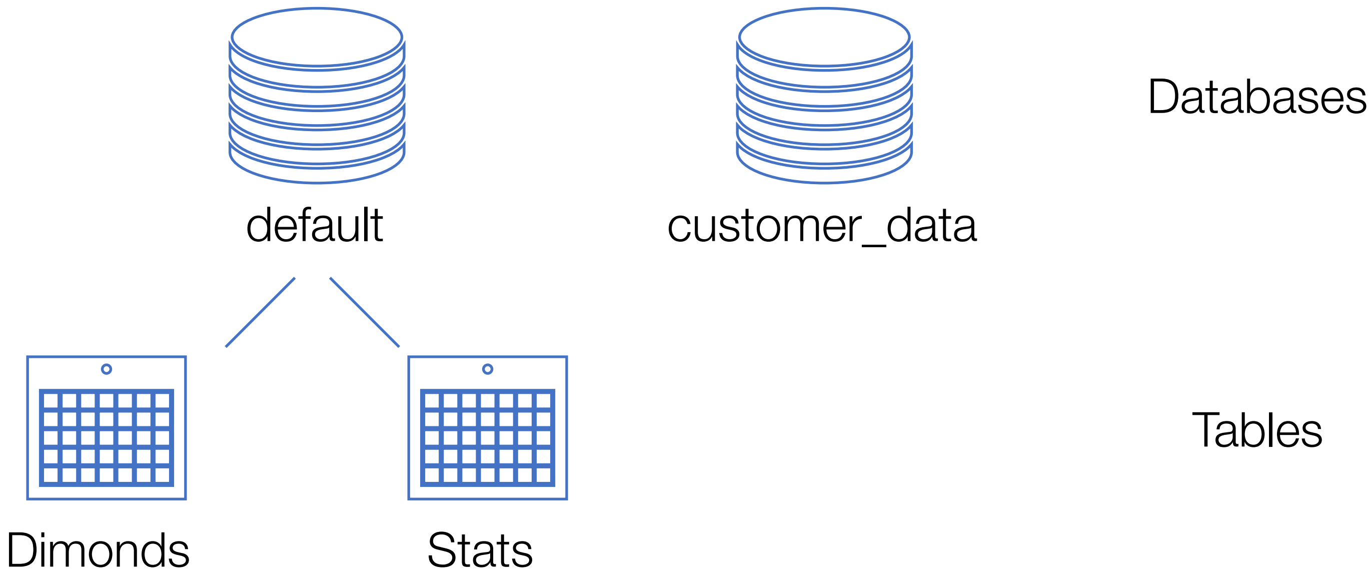
# Commands

- Show databases
- Show tables
- Create Database
- Alter Database
- Drop Database
- Create Table / View / Function
- Drop Table / View / Function
- Select
- Insert
- Alter
- …

# Basic Data Structure

spark.sql("Create Database customer_data")
display(spark.sql("show databases"))

| databaseName |
|---|
| customer_data |
| default |




display(spark.sql("show tables"))

| database | tableName | isTemporary |
|---|---|---|
| default | diamonds | false |
| | stats | true |

# Creating Tables

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [db_name.]table_name
    [(col_name1 col_type1 [COMMENT col_comment1], ...)]
    USING datasource
    [OPTIONS (key1=val1, key2=val2, ...)]
    [PARTITIONED BY (col_name1, col_name2, ...)]
    [CLUSTERED BY (col_name3, col_name4, ...) INTO num_buckets BUCKETS]
    [LOCATION path]
    [COMMENT table_comment]
    [TBLPROPERTIES (key1=val1, key2=val2, ...)]
    [AS select_statement]
```

```
CREATE TABLE boxes (width INT, length INT, height INT) USING CSV
```

TEMPORARY
The created table will be available only in this session and will not be persisted to the underlying metastore

# Example

```
CREATE TABLE rectangles
    USING PARQUET
    PARTITIONED BY (width)
    CLUSTERED BY (length) INTO 8 buckets
    AS SELECT * FROM boxes
```

USING <data source>
The file format to use for the table. One of TEXT, CSV, JSON, JDBC, PARQUET, ORC, HIVE, DELTA, and LIBSVM

PARTITIONED BY
Partition the created table by the specified columns. A directory is created for each partition.

CLUSTERED BY
Each partition in the created table will be split into a fixed number of buckets by the specified columns. This is typically used with partitioning to read and shuffle less data. Support for SORTED BY will be added in a future version.

LOCATION
The created table uses the specified directory to store its data. This clause automatically implies EXTERNAL.

AS <select_statement>
Populate the table with input data from the select statement. This may not be specified with TEMPORARY TABLE or with a column list. To specify it with TEMPORARY, use CREATE TEMPORARY VIEW instead.

# Select Example

```
df.createOrReplaceTempView("stats")
display(spark.sql("""
  select player_id, sum(1) as games, sum(goals) as goals
  from stats
  group by 1
  order by 3 desc
  limit 5
"""))
```

| player_id | games | goals |
|---|---|---|
| 8471214 | 520 | 299 |
| 8471675 | 522 | 221 |
| 8474141 | 499 | 216 |
| 8470794 | 515 | 207 |
| 8475765 | 465 | 200 |

# Select

```
SELECT [hints, ...] [ALL|DISTINCT] named_expression[, named_expression, ...]
    FROM relation[, relation, ...]
    [lateral_view[, lateral_view, ...]]
    [WHERE boolean_expression]
    [aggregation [HAVING boolean_expression]]
    [ORDER BY sort_expressions]
    [CLUSTER BY expressions]
    [DISTRIBUTE BY expressions]
    [SORT BY sort_expressions]
    [WINDOW named_window[, WINDOW named_window, ...]]
    [LIMIT num_rows]

named_expression:
    : expression [AS alias]

relation:
    | join_relation
    | (table_name|query|relation) [sample] [AS alias]
    : VALUES (expressions)[, (expressions), ...]
          [AS (column_name[, column_name, ...])]

expressions:
    : expression[, expression, ...]

sort_expressions:
    : expression [ASC|DESC][, expression [ASC|DESC], ...]
```

# Examples

```
SELECT * FROM boxes
SELECT width, length FROM boxes WHERE height=3
SELECT DISTINCT width, length FROM boxes WHERE height=3 LIMIT 2
SELECT * FROM boxes ORDER BY width
```

- DISTINCT: select all matching rows from the relation then remove duplicate results.
- WHERE: Filter rows by predicate.
- ORDER BY: Impose total ordering on a set of expressions. Default sort direction is ascending. You cannot use this with SORT BY, CLUSTER BY, or DISTRIBUTE BY.
- SORT BY: Impose ordering on a set of expressions within each partition. Default sort direction is ascending. You cannot use this with ORDER BY or CLUSTER BY.
- LIMIT: Limit the number of rows returned.

# Casting and Functions

select cast(goals/shots * 50 as int)/50.0 as Goals_per_shot
,sum(1) as Players

| | | | | | |
|---|---|---|---|---|---|
| ! | char | floor | map_keys | schema_of_json | uuid |
| % | char_length | format_number | map_values | second | var_pop |
| & | character_length | format_string | max | sentences | var_samp |
| * | chr | from_json | md5 | sequence | variance |
| + | coalesce | from_unixtime | mean | sha | weekday |
| - | collect_list | from_utc_timestamp | min | sha1 | weekofyear |
| / | collect_set | get_json_object | minute | sha2 | when |
| < | concat | greatest | mod | shiftleft | window |
| <= | concat_ws | grouping | monotonically_increasing_id | shiftright | xpath |
| <=> | conv | grouping_id | month | shiftrightunsigned | xpath_boolean |
| = | corr | hash | months_between | shuffle | xpath_double |
| | cos | hex | named_struct | sign | xpath_float |
| > | cosh | hour | nanvl | signum | xpath_int |
| >= | cot | hypot | negative | sin | xpath_long |
| ^ | count | if | next_day | sinh | xpath_number |
| abs | count_min_sketch | ifnull | not | size | xpath_short |
| acos | covar_pop | in | now | skewness | xpath_string |
| add_months | covar_samp | initcap | ntile | slice | year |
| aggregate | crc32 | inline | nullif | smallint | zip_with |
| and | cube | inline_outer | nvl | sort_array | \| |
| approx_count_distinct | cume_dist | input_file_block_length | nvl2 | soundex | ~ |
| approx_percentile | current_database | input_file_block_start | octet_length | space | |
| array | current_date | input_file_name | or | spark_partition_id | |
| array_contains | current_timestamp | instr | parse_url | split | |
| array_distinct | date | int | percent_rank | sqrt | |

https://docs.databricks.com/spark/latest/spark-sql/la

# Examples

```
SELECT * FROM boxes DISTRIBUTE BY width SORT BY width
SELECT * FROM boxes CLUSTER BY length
SELECT * FROM boxes TABLESAMPLE (3 ROWS)
SELECT * FROM boxes TABLESAMPLE (25 PERCENT)
```

**`HAVING`**
Filter grouped result by predicate.

**`DISTRIBUTE BY`**
Repartition rows in the relation based on a set of expressions. Rows with the same expression values will be hashed to the same worker. You cannot use this with `ORDER BY` or `CLUSTER BY`.

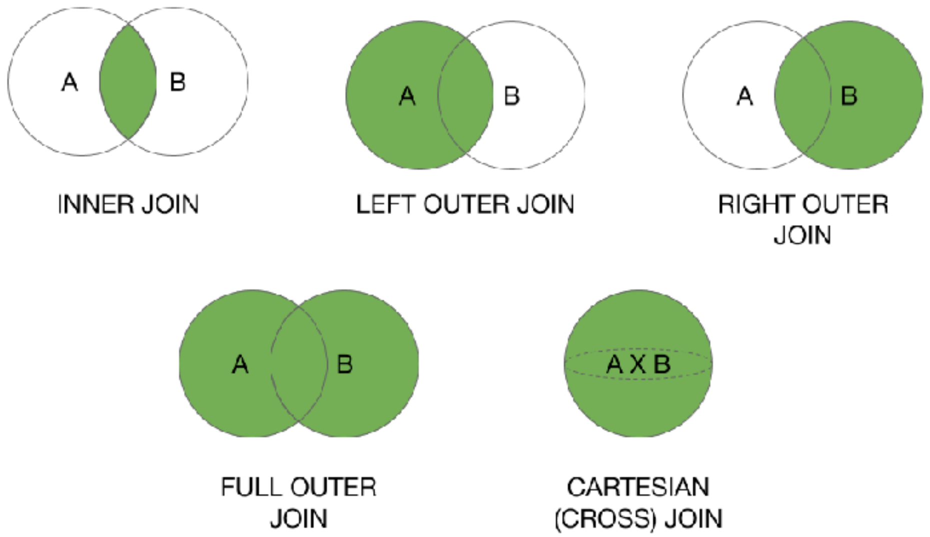
**`CLUSTER BY`**
Repartition rows in the relation based on a set of expressions and sort the rows in ascending order based on the expressions. In other words, this is a shorthand for `DISTRIBUTE BY` and `SORT BY` where all expressions are sorted in ascending order. You cannot use this with `ORDER BY`, `DISTRIBUTE BY`, or `SORT BY`.

**`Sample`**
Sample the input data. This can be expressed in terms of either a percentage (must be between 0 and 100) or a fixed number of input rows.

# Joins

```
SELECT * FROM boxes INNER JOIN rectangles ON boxes.width = rectangles.width
SELECT * FROM boxes FULL OUTER JOIN rectangles USING (width, length)
SELECT * FROM boxes NATURAL JOIN rectangles
```

- 




**`INNER JOIN`**
Select all rows from both relations where there is match.

**`OUTER JOIN`**
Select all rows from both relations, filling with null values on the side that does not have a match.

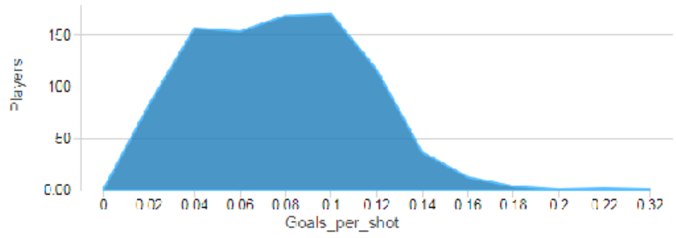
**`SEMI JOIN`**
Select only rows from the side of the `SEMI JOIN` where there is a match. If one row matches multiple rows, only the first match is returned.

**`LEFT ANTI JOIN`**
Select only rows from the left side that match no rows on the right side.

# Aggregation

Group by a set of expressions using one or more aggregate functions. Common built-in aggregate functions include count, avg, min, max, and sum.

```
display(spark.sql("""
  select cast(goals/shots * 50 as int)/50.0 as Goals_per_shot
      ,sum(player_id) as Players
  from (
    select player_id, sum(shots) as shots, sum(goals) as goals
    from stats
    group by player_id
    having goals >= 5
  )
  group by Goals_per_shot
  order by Goals_per_shot
"""))
```




Spark also provides different ways to group by, with ROLLUP, CUBE, and GROUPING SETS

# Explain

- Provide detailed plan information about statement without actually running it

```
display(spark.sql("""exp
lain select player_id,
sum(shots) as shots,
sum(goals) as goals
   from stats
   group by player_id
   having goals >=
5"""))
```

```
== Physical Plan ==
*(2) Filter (isnotnull(goals#1725L) && (goals#1725L >= 5))
+- *(2) HashAggregate(keys=[player_id#27],
functions=[finalmerge_sum(merge sum#1735L) AS
sum(cast(shots#32 as bigint))#1728L, finalmerge_sum(merge
sum#1737L) AS sum(cast(goals#31 as bigint))#1729L])
   +- Exchange hashpartitioning(player_id#27, 200)
      +- *(1) HashAggregate(keys=[player_id#27],
functions=[partial_sum(cast(shots#32 as bigint)) AS sum#1735L,
partial_sum(cast(goals#31 as bigint)) AS sum#1737L])
         +- *(1) FileScan csv [player_id#27,goals#31,shots#32]
Batched: false, DataFilters: [], Format: CSV, Location:
InMemoryFileIndex[dbfs:/FileStore/tables/game_skater_stats.csv],
PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<player_id:int,goals:int,shots:int>
```

# Summary

- SQL provides a standard way to analyze data
- Select
- Join
- Group By

# <5> Machine Learning on Spark

# MLib

- MLlib is Spark's machine learning (ML) library
    - ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
    - Featurization: feature extraction, transformation, dimensionality reduction, and selection
    - Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
    - Utilities: linear algebra, statistics, data handling, etc.

# Example

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import Correlation

data = [(Vectors.sparse(4, [(0, 1.0), (3, -2.0)]),),
        (Vectors.dense([4.0, 5.0, 0.0, 3.0]),),
        (Vectors.dense([6.0, 7.0, 0.0, 8.0]),),
        (Vectors.sparse(4, [(0, 9.0), (3, 1.0)]),)]
df = spark.createDataFrame(data, ["features"])

r1 = Correlation.corr(df, "features").head()
print("Pearson correlation matrix:\n" + str(r1[0]))

r2 = Correlation.corr(df, "features", "spearman").head()
print("Spearman correlation matrix:\n" + str(r2[0]))
```

```
Pearson correlation matrix:
DenseMatrix([[ 1.        ,  0.05564149,         nan,  0.40047142],
             [ 0.05564149,  1.        ,         nan,  0.91359586],
             [        nan,         nan,  1.        ,         nan],
             [ 0.40047142,  0.91359586,         nan,  1.        ]])
Spearman correlation matrix:
DenseMatrix([[ 1.        ,  0.10540926,         nan,  0.4       ],
             [ 0.10540926,  1.        ,         nan,  0.9486833 ],
             [        nan,         nan,  1.        ,         nan],
             [ 0.4       ,  0.9486833 ,         nan,  1.        ]])
Command took 3.24 seconds -- by erantoch@gmail.com at 3/25/2019, 4:07:09 PM on TeachingSpar
```

# Hypothesis Testing

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import ChiSquareTest

data = [(0.0, Vectors.dense(0.5, 10.0)),
        (0.0, Vectors.dense(1.5, 20.0)),
        (1.0, Vectors.dense(1.5, 30.0)),
        (0.0, Vectors.dense(3.5, 30.0)),
        (0.0, Vectors.dense(3.5, 40.0)),
        (1.0, Vectors.dense(3.5, 40.0))]
df = spark.createDataFrame(data, ["label", "features"])

r = ChiSquareTest.test(df, "features", "label").head()
print("pValues: " + str(r.pValues))
print("degreesOfFreedom: " + str(r.degreesOfFreedom))
print("statistics: " + str(r.statistics))
```

Table 1

| | |
|---|---|
| pValues: [0.687289278791 | 0.682270330336] |
| degreesOfFreedom: [2 | 3] |
| statistics: [0.75 | 1.5] |

# Extracting Features

- Feature Extractors
    - TF-IDF
    - Word2Vec
    - CountVectorizer
    - FeatureHasher
- Feature Transformers
    - Tokenizer
    - StopWordsRemover
    - n-gram
    - PCA
    - Imputer
-

# TF/IDF

```
from pyspark.ml.feature import HashingTF, IDF, Tokenizer
sentenceData = spark.createDataFrame([
    (0.0, "Hi I heard about Spark"),
    (0.0, "I wish Java could use case classes"),
    (1.0, "Logistic regression models are neat")
], ["label", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
wordsData = tokenizer.transform(sentenceData)

hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures",
numFeatures=20)
featurizedData = hashingTF.transform(wordsData)
# alternatively, CountVectorizer can also be used to get term
frequency vectors

idf = IDF(inputCol="rawFeatures", outputCol="features")
idfModel = idf.fit(featurizedData)
rescaledData = idfModel.transform(featurizedData)

rescaledData.select("label", "features").show()
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  0.0|(20,[0,5,9,17],[0...|
|  0.0|(20,[2,7,9,13,15]...|
|  1.0|(20,[4,6,13,15,18...|
+-----+--------------------+
```

# Word2Vec

```
from pyspark.ml.feature import Word2Vec

# Input data: Each row is a bag of words from a sentence or
document.
documentDF = spark.createDataFrame([
    ("Hi I heard about Spark".split(" "), ),
    ("I wish Java could use case classes".split(" "), ),
    ("Logistic regression models are neat".split(" "), )
], ["text"])

# Learn a mapping from words to Vectors.
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text",
outputCol="result")
model = word2Vec.fit(documentDF)

result = model.transform(documentDF)
for row in result.collect():
    text, vector = row
    print("Text: [%s] => \nVector: %s\n" % (", ".join(text),
str(vector)))
```

```
Text: [Hi, I, heard, about, Spark] =>
Vector: [-0.0159335330129,0.0215295135975,0.00646775923669]

Text: [I, wish, Java, could, use, case, classes] =>
Vector: [-0.0109682194889,-0.0309452622065,0.00577214998858]

Text: [Logistic, regression, models, are, neat] =>
Vector: [-0.0435343801975,0.0350369662046,0.0243757784367]
```

# PCA

```python
from pyspark.ml.feature import PCA
from pyspark.ml.linalg import Vectors

data = [(Vectors.sparse(5, [(1, 1.0), (3, 7.0)]),),
        (Vectors.dense([2.0, 0.0, 3.0, 4.0, 5.0]),),
        (Vectors.dense([4.0, 0.0, 0.0, 6.0, 7.0]),)]
df = spark.createDataFrame(data, ["features"])

pca = PCA(k=3, inputCol="features", outputCol="pcaFeatures")
model = pca.fit(df)

result = model.transform(df).select("pcaFeatures")
result.show(truncate=False)
```

```
+------------------------------------------------------------+
|pcaFeatures                                                 |
+------------------------------------------------------------+
|[1.6485728230883807,-4.013282700516296,-5.51655055421941]   |
|[-4.645104331781532,-1.1167972663619032,-5.516550554219409]|
|[-6.428880535676488,-5.337951427775355,-5.51655055421941]   |
+------------------------------------------------------------+
```

# Classification and Regression

- Classification
    - Logistic regression
    - Decision tree classifier
    - Random forest classifier
    - …
- Regression
    - Linear regression
    - Generalized linear regression
    - Decision tree regression
    - …
- Linear methods
- Decision trees
- Tree Ensembles
    - Random Forests
    - Gradient-Boosted Trees (GBTs)

# Linear Regression

```
from pyspark.ml.regression import LinearRegression

# Load training data
training = spark.read.format("libsvm")\
    .load("data/mllib/sample_linear_regression_data.txt")

lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for linear regression
print("Coefficients: %s" % str(lrModel.coefficients))
print("Intercept: %s" % str(lrModel.intercept))

# Summarize the model over the training set and print out some metrics
trainingSummary = lrModel.summary
print("numIterations: %d" % trainingSummary.totalIterations)
print("objectiveHistory: %s" % str(trainingSummary.objectiveHistory))
trainingSummary.residuals.show()
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)
```

# Random Forest

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

# Load and parse the data file, converting it to a DataFrame.
data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data
# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = data.randomSplit([0.7, 0.3])
# Train a RandomForest model.
rf = RandomForestRegressor(featuresCol="indexedFeatures")
# Chain indexer and forest in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, rf])
# Train model.  This also runs the indexer.
model = pipeline.fit(trainingData)
# Make predictions.
predictions = model.transform(testData)
# Select example rows to display.
predictions.select("prediction", "label", "features").show(5)
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
rfModel = model.stages[1]
print(rfModel)  # summary only
```

# Other Functions

- Clustering
  - K-means
  - Latent Dirichlet allocation (LDA)
  - Bisecting k-means
  - Gaussian Mixture Model (GMM)
- Collaborative Filtering
  - Explicit vs. implicit feedback
  - Scaling of the regularization parameter
  - Cold-start strategy
- Frequent Pattern Mining
  - FP-Growth
  - PrefixSpan
- Model quality
  - Model selection (a.k.a. hyperparameter tuning)
  - Cross-Validation
  - Train-Validation Split

# ML Pipelines

# ML Pipelines

- In machine learning, it is common to run a sequence of algorithms to process and learn from data. E.g., a simple text document processing workflow might include several stages:
  - Split each document's text into words.
  - Convert each document's words into a numerical feature vector.
  - Learn a prediction model using the feature vectors and labels.
- MLlib represents such a workflow as a Pipeline, which consists of a sequence of PipelineStages (Transformers and Estimators) to be run in a specific order

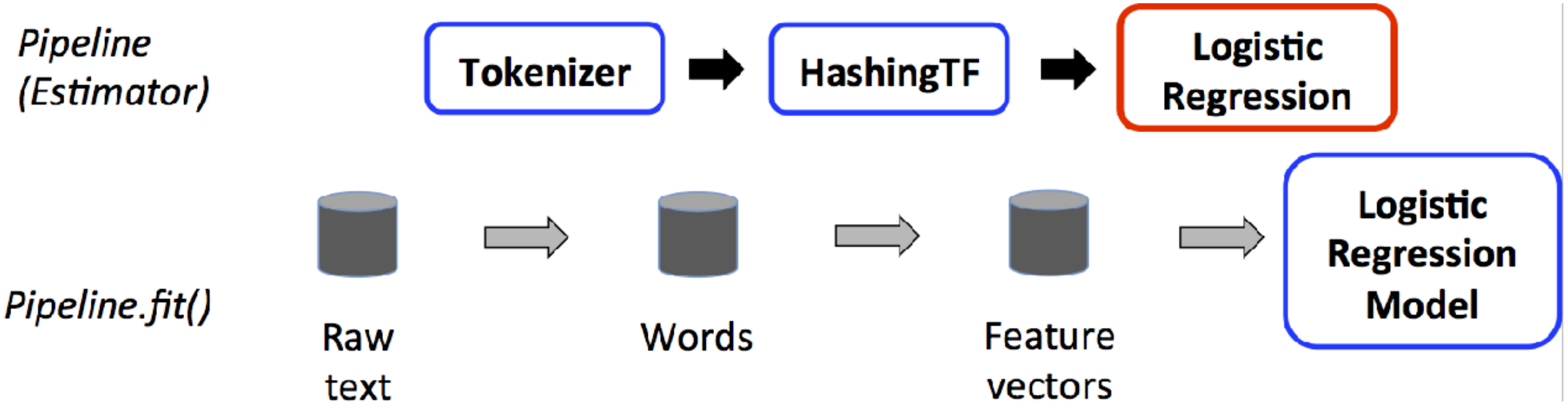
# Transformers

- A Transformer is an abstraction that includes feature transformers and learned models. Technically, a Transformer implements a method transform(), which converts one DataFrame into another, generally by appending one or more columns. For example:
- A feature transformer might take a DataFrame, read a column (e.g., text), map it into a new column (e.g., feature vectors), and output a new DataFrame with the mapped column appended.
- A learning model might take a DataFrame, read the column containing feature vectors, predict the label for each feature vector, and output a new DataFrame with predicted labels appended as a column.

# Estimators

- An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on data.
- An Estimator implements a method fit(), which accepts a DataFrame and produces a Model.
- For example, a learning algorithm such as LogisticRegression is an Estimator, and calling fit() trains a LogisticRegressionModel, which is a Model and hence a Transformer.

# Pipelines




- The first two stages (Tokenizer and HashingTF) are Transformers (blue), and the third (LogisticRegression) is an Estimator (red)

# Example

```python
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

# Cont'd

```python
# Fit the pipeline to training documents.
model = pipeline.fit(training)

# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s, prediction=%f" % (rid, text, str(prob), prediction))
```

# Summary

- The challenges of big data
- HDFS
- MapReduce
- Spark