Benchmarking Apache Flink and Apache Spark DataFlow Systems on Large-Scale Distributed Machine Learning Algorithms

Advisor Prof. Sonia Bergamaschi

Co-Advisor Dr. Tilmann Rabl

Co-Advisor Christoph Boden

Candidate Andrea Spina

Dipartimento di Ingegneria "Enzo Ferrari" - Laurea Magistrale di Ingegneria Informatica - MODENA

Where, When, How and Why

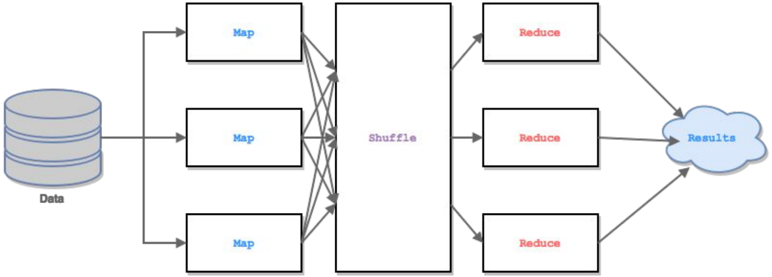
- Berlin, DE
- 5 months Traineeship - MAY - OCT ‘16
- *Database Systems and Information Management Group,* Technische Universität
- Team Project - Systems Performance Research Unit

Agenda

- Background
- Experiments Definition
- Benchmarking and Results Analysis
- Insights by Results

Background

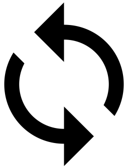
Why Distributed Machine Learning?



Mutliple Sources

MapReduce

Iterative Algorithms



Because represents innovation …

SOLUTION

Mutliple Sources

Massive DataFlow Engines

Iterative Algorithms

One of the goals - ***fairness***

- give code open-source
- keep jobs reproducible
- make benchmark exhaustive
- …
- model systems ***as same as possible***

Another Goal - include more and more systems

APACHE Spark™

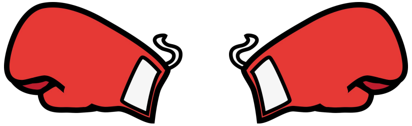
MPI

ML

My Goal - *“OK, may I start with a couple of those?”*

Apache Flink vs. Apache Spark

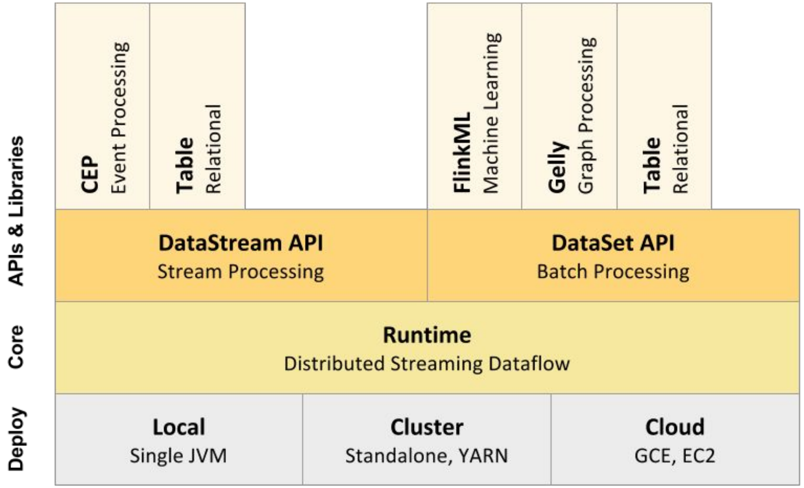
Apache Flink

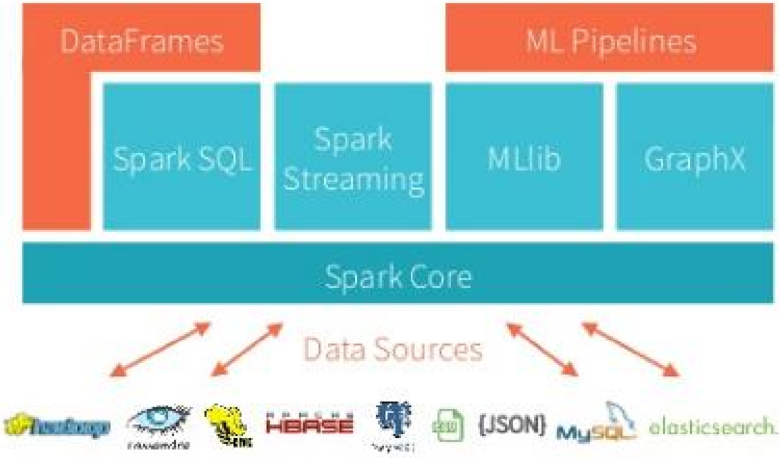


Apache Spark

THE GOAL - Benchmark on Performance and Scalability

Systems similarities - they are **stacks**





Systems similarities - they do **batch** and **streaming**

JAVA

SCALA

PYTHON

JAVA

SCALA

PYTHON

R

Systems similarities - they do **batch** and **streaming**

JAVA

SCALA

PYTHON

JAVA

SCALA

PYTHON

R

Apache Spark vs Apache Flink - *differences*

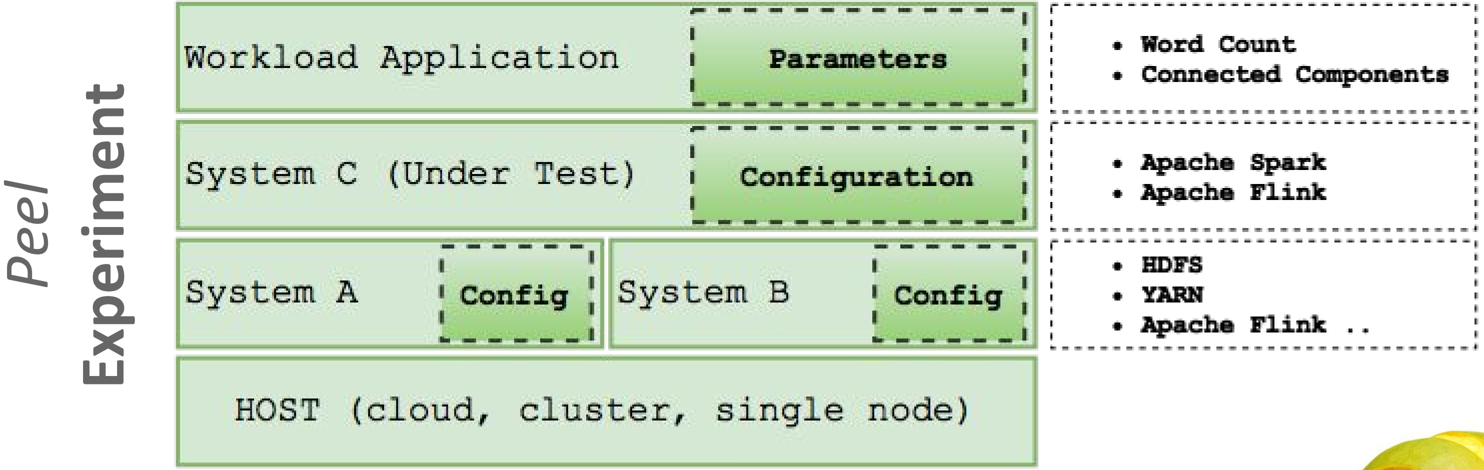
batch to Streaming

streaming to Batch

we do **batch**

...
and iterations, memory management,
user policies
...

Peel Framework - The Benchmarking Software



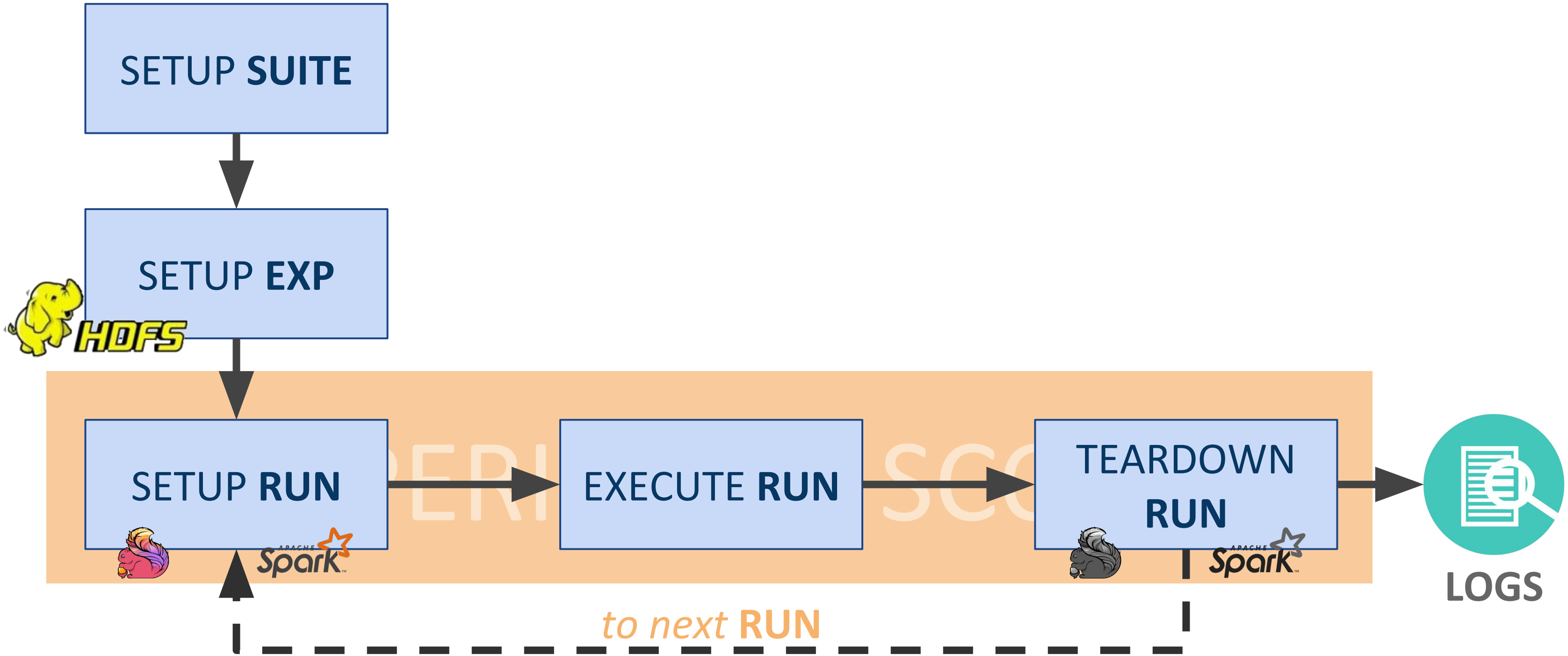
- submits config by *dependency injection*
- packages together by *peel* **bundles**

Peel execution flow - the `suite:run` command

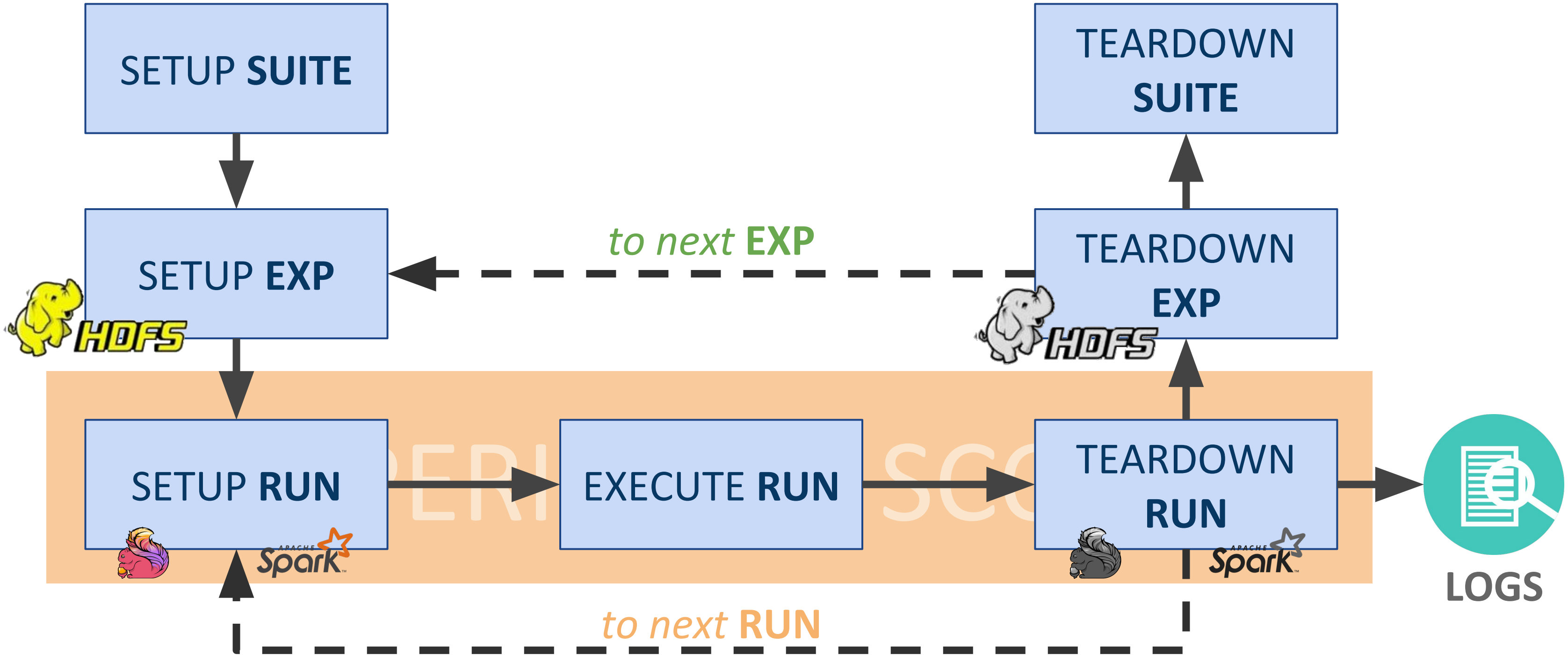
SETUP **SUITE**

Peel execution flow - turn on systems

Peel execution flow - collect logs and run again



Peel execution flow - turn off systems



shee - fast and furios *peel* data visualization tool

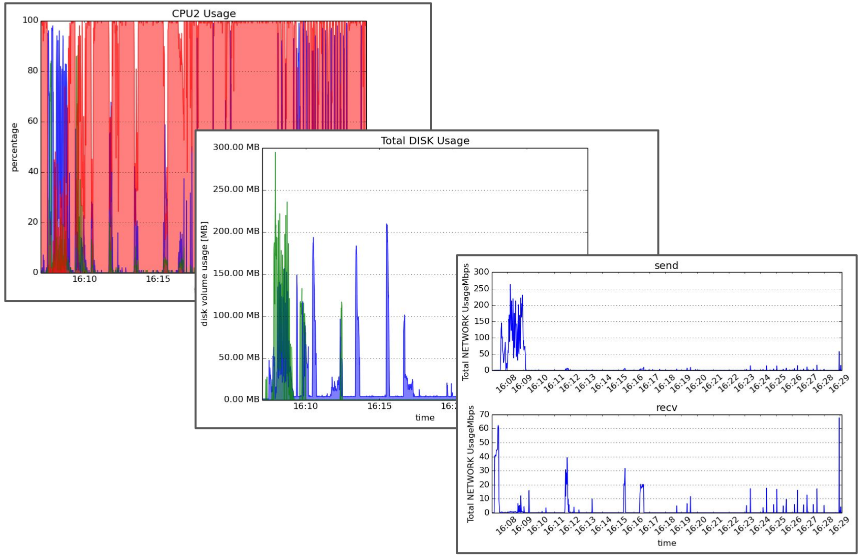
https://github.com/spi-x-i/shee

- built on top of Python, Pandas and matplotlib
- **APIs**
  - node - level
  - cluster - level
- web **UI**

shee - fast and furios *peel* data visualization tool

https://github.com/spi-x-i/shee

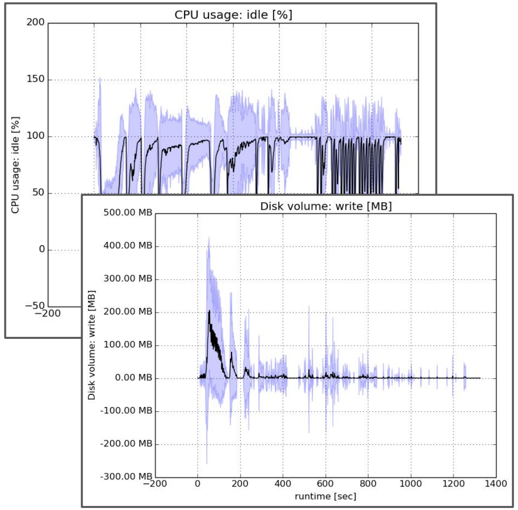
- built on top of Python, Pandas and matplotlib
- **APIs**
  - **node - level**
  - cluster - level
- web **UI**

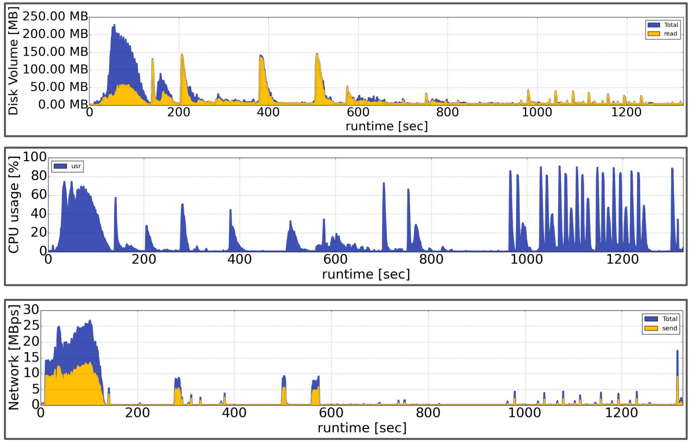


shee - fast and furios *peel* data visualization tool

https://github.com/spi-x-i/shee

- built on top of Python, Pandas and matplotlib
- **APIs**
  - node - level
  - **cluster - level**
- web **UI**

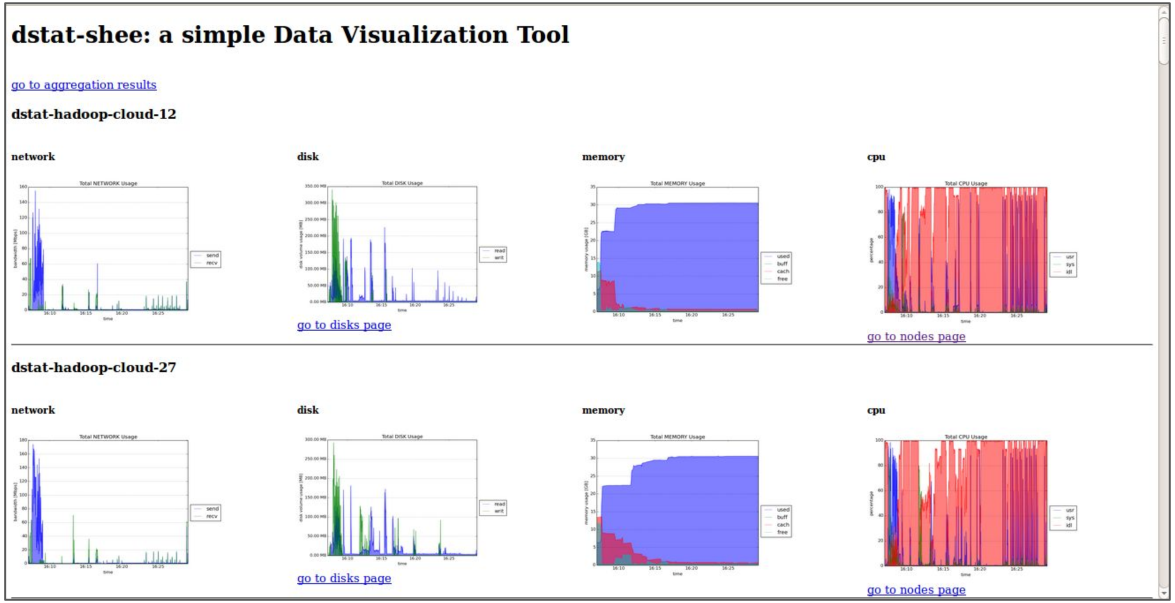




shee - fast and furios *peel* data visualization tool

https://github.com/spi-x-i/shee

- built on top of Python, Pandas and matplotlib
- **APIs**
  - node - level
  - cluster - level
- web **UI**



shee - fast and furios *peel* data visualization tool

https://github.com/spi-x-i/shee

- built on top of Python, Pandas and matplotlib
- **APIs**
  - node - level
  - cluster - level
- web **UI**

1 ★ Star | 48 Commit | 1 Fork | 3 Contributors

Defining Experiments

https://gitlab.tubit.tu-berlin.de/andrea-spina/MLBenchmark

The **fairness** constraint

- Apache Spark 1.6.2 - Apache Flink 1.0.3
- **We want the same** (*as much as possible*) **...**
  - data *structures*
  - *pipeline* for solvers
  - *operators*
  - configuration
  - parameters
  - environment

Guaranteed by Peel

Experiments Overview - **Four** applications

APPROACH We want to cover **many** applications	Regression	Supervised Learning	Not Supervised Learning	Recommendation System
ALGORITHM Choosed by a **Tradeoff** between complexity and fairness	Multiple Linear Regression	Support Vector Machine *	KMeans	Alternating Least Squares
DATA GENERATION Writing Data **on-demand** by Peel Framework	Apache SystemML	Apache Spark	Apache Spark	Apache Flink

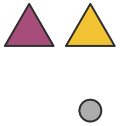
Building the Experiment Pipeline - **KMeans Example**

PREMISE

We always evaluate **Training Phase Performance**

Building the KMeans Pipeline - **Studying**

KMEANS *clustering*
find new classes from
unlabeled data by grouping



$$C = \{c_1, c_2, \ldots, c_k\}$$

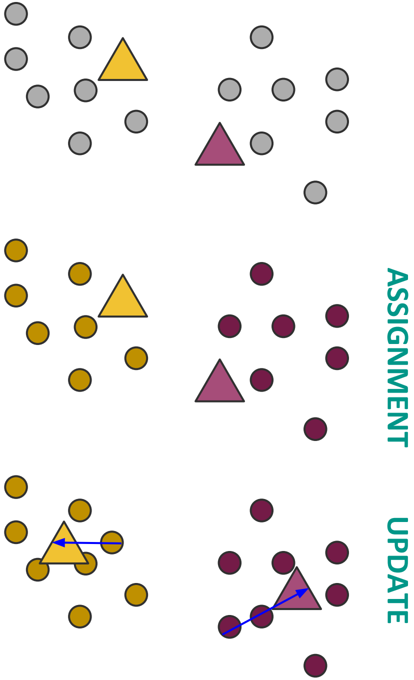
$$X = \{x_1, x_2, \ldots, x_n\}$$

ASSIGNMENT STEP
re-partition datapoints according to
centroids

$$\varphi_X(C) = \sum_{x \in X} d(x, C)^2$$

UPDATE STEP
retrieve new centroids by
datapoints location mean

$$c_i^{(t+1)} = \frac{1}{S_i^{(t)}} \sum_{x_j \in S_i^{(t)}} x_j$$

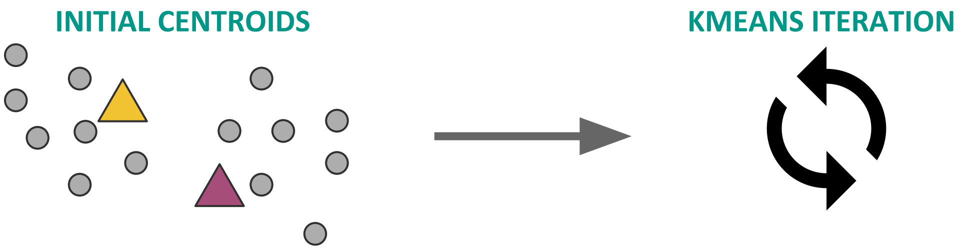


Building the KMeans Pipeline - **Studying**

1. Explore systems machine learning libraries
2. Do research!

E.g. keeping smarter initial **k** centroids choice

- *random*
- KMeans ++
- KMeans ||



What do we want to compare? What keeps Systems on Stress!

Building the KMeans Pipeline - **Data Structures**

INIT CENTERS

DATA

EXPERIMENT SCOPE

Dataset → Point vectors

p_0	x_0	x_1	…	x_{n-1}
p_1	x_0	x_1	…	x_{n-1}

Init centers → (id, vector)

c_0	id_0	x_0	x_1	…	x_{n-1}
c_1	id_1	x_0	x_1	…	x_{n-1}

We need to:

- model data
- operate on data

We employed:

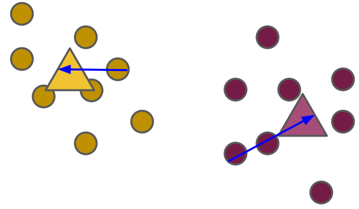
- ~~Flink Vectors~~
- ~~Spark Vectors~~
- **Breeze Vectors**
- ~~Scala Arrays~~

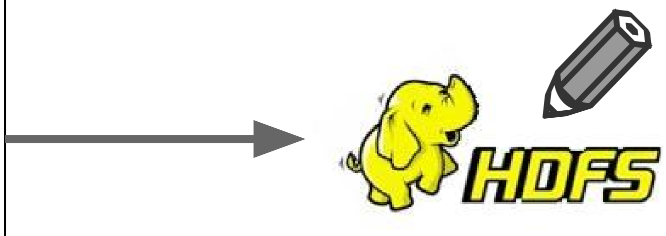
Building the KMeans Pipeline - **KMeans Iteration**

	ASSIGNMENT STEP	UPDATE STEP		
SPARK	**Map** Function	**ReduceByKey**	Map Function Aggregate by **AVG**	**Broadcasting** Function
FLINK	**RichMapFunction**	**GroupBy** **Reduce**	Map Function Aggregate by **AVG**	**Broadcasting** Function

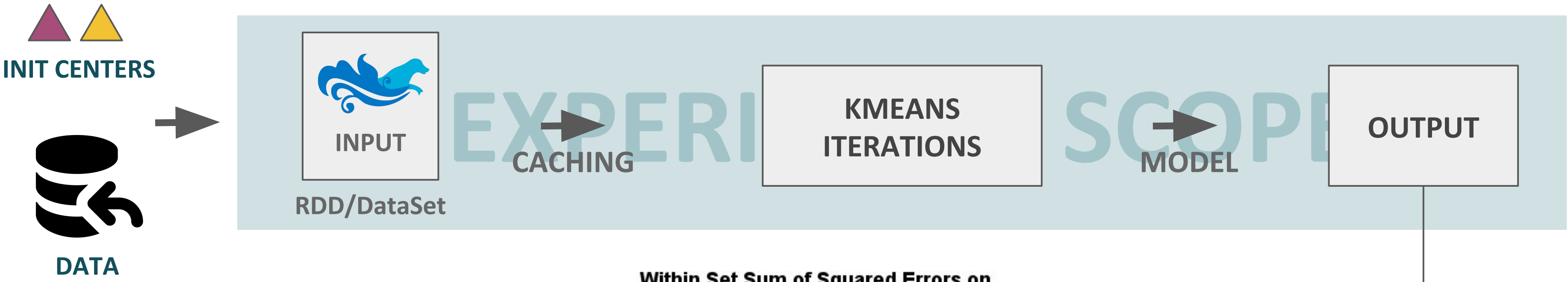
Building the KMeans Pipeline - **Materializing**

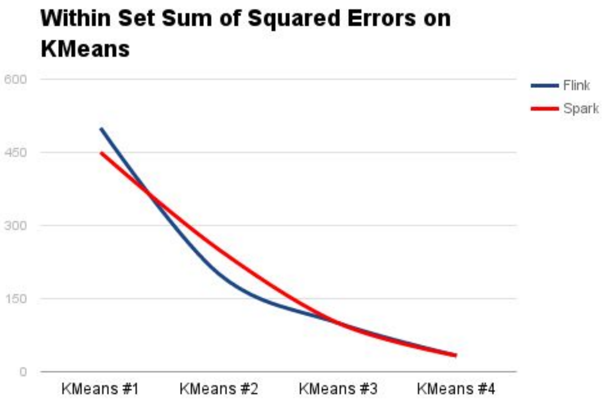
Model → **final centers**

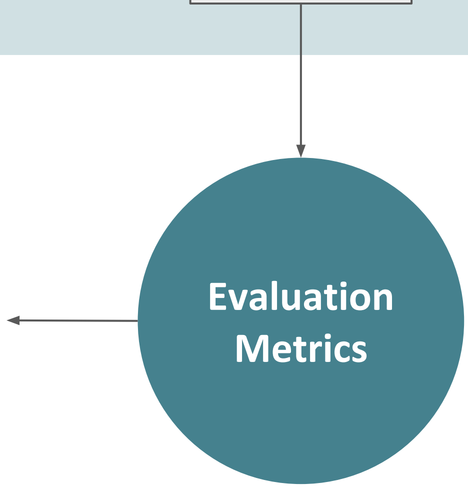




Building the KMeans Pipeline - **Validation**







Model Evaluation

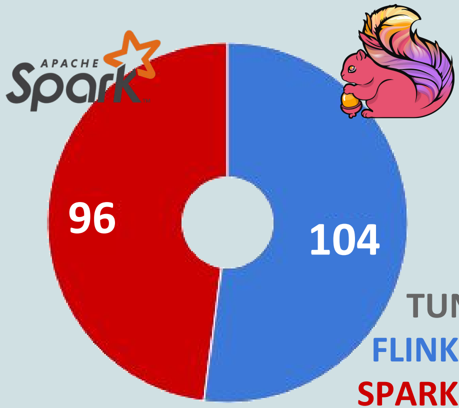
- *fairness* **first**
- convergence of metrics
- good-enough model **then**

Benchmarking and Results Analysis

Some general **insights**

TOTAL RUNTIME [h]

200



TUNING
FLINK 22%
SPARK 30%

CLUSTERS

2

WALLY

nodes	**30**	CPUs/node	**8**
RAM/node	**16GB**	Storage/no.	**3x1TB**
Eth	**1Gbit**		

CLOUD-11

nodes	**25**	CPUs/node	**16**
RAM/node	**32GB**	Storage/no.	**2x1TB**
Eth	**1Gbit**		

DATASETS

28

AVERAGE SIZE

275GB

EVALUATIONS

strong scale
weak scale
data scale

Spark versus Flink Summary

34 

8

Multiple Linear Regression

Spark v Flink
8 - 1

- Spark 63% outperforms Flink
- Flink 74% faster on critic resources
- FlinkML provides better runtimes

Support Vector Machine

Spark v Flink
16 - 0

- Spark 71% outperforms Flink
- Flink likes MORE Data
- Good Scalability Behavior

KMeans

Spark v Flink
10 - 7

- Similar Performance
- Flink definitely likes MORE data
- Flink 11% faster on critic resources

Reccomendation System

NOT COMPARABLE

Spark versus **Flink** Summary

Spark 34 | RUNTIME WINS | 8 Flink

Multiple Linear Regression

Spark v Flink

8 - 1

- Spark 63% outperforms Flink
- Flink 74% faster on critic resources
- FlinkML provides better runtimes

Support Vector Machine

Spark v Flink

16 - 0

- Spark 71% outperforms Flink
- Flink likes MORE Data
- Good Scalability Behavior

KMeans

Spark v Flink

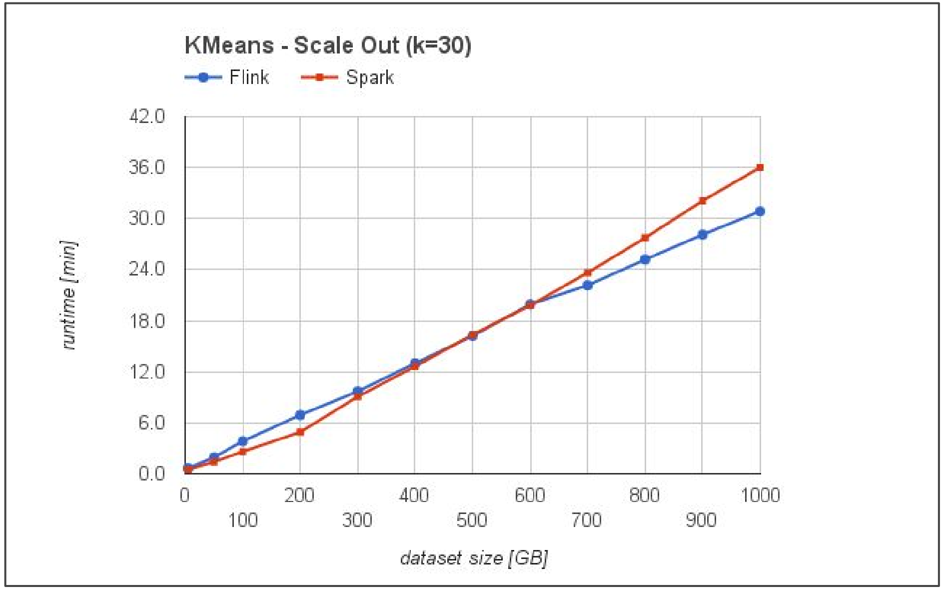
10 - 7

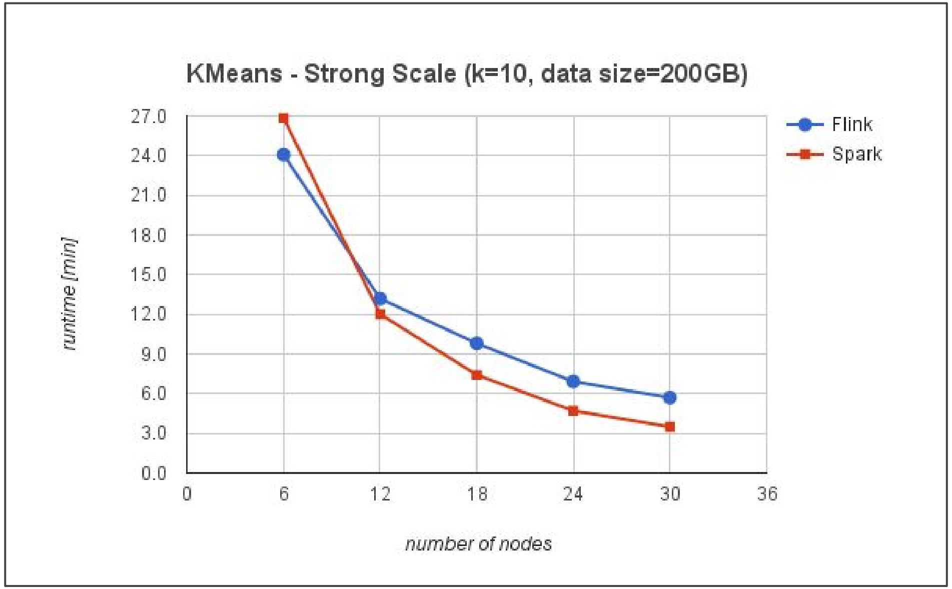
- Similar Performance
- Flink definitely likes MORE data
- Flink 11% faster on critic resources

Alternating Least Squares

NOT COMPARABLE

KMeans **strong scale** and **scale data**

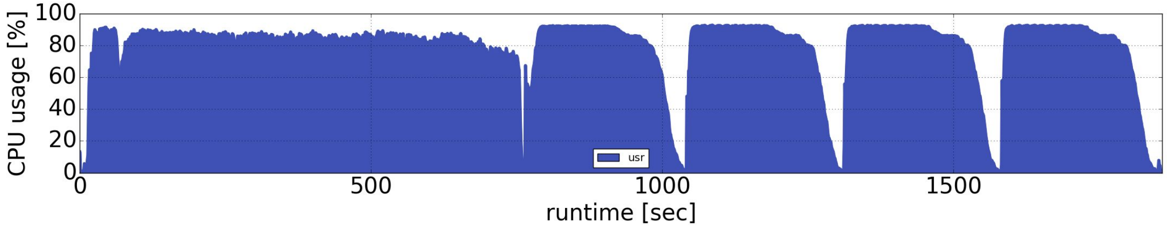




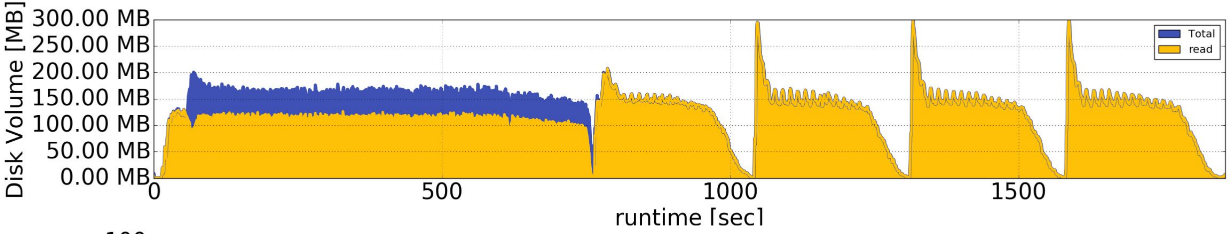
- **12GB** RAM per node
- **8** core CPU per node
- sparsity **0%**
- model size **100**

Insights from Executions

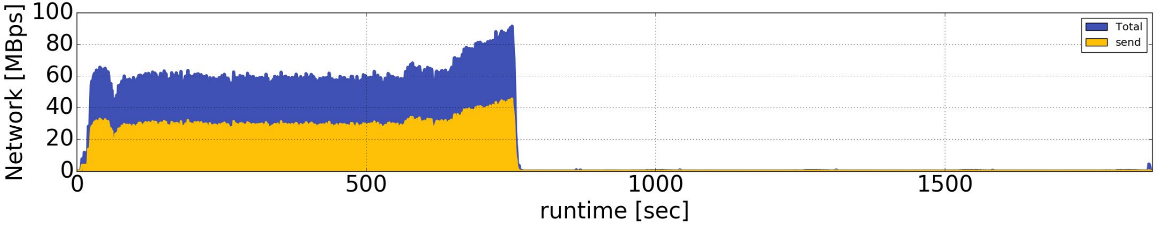
How should a large-scale processing engine **work** ?



CPU
user



DISK
read / write



NETWORK
send / recv

Flink KMeans

30 centroids

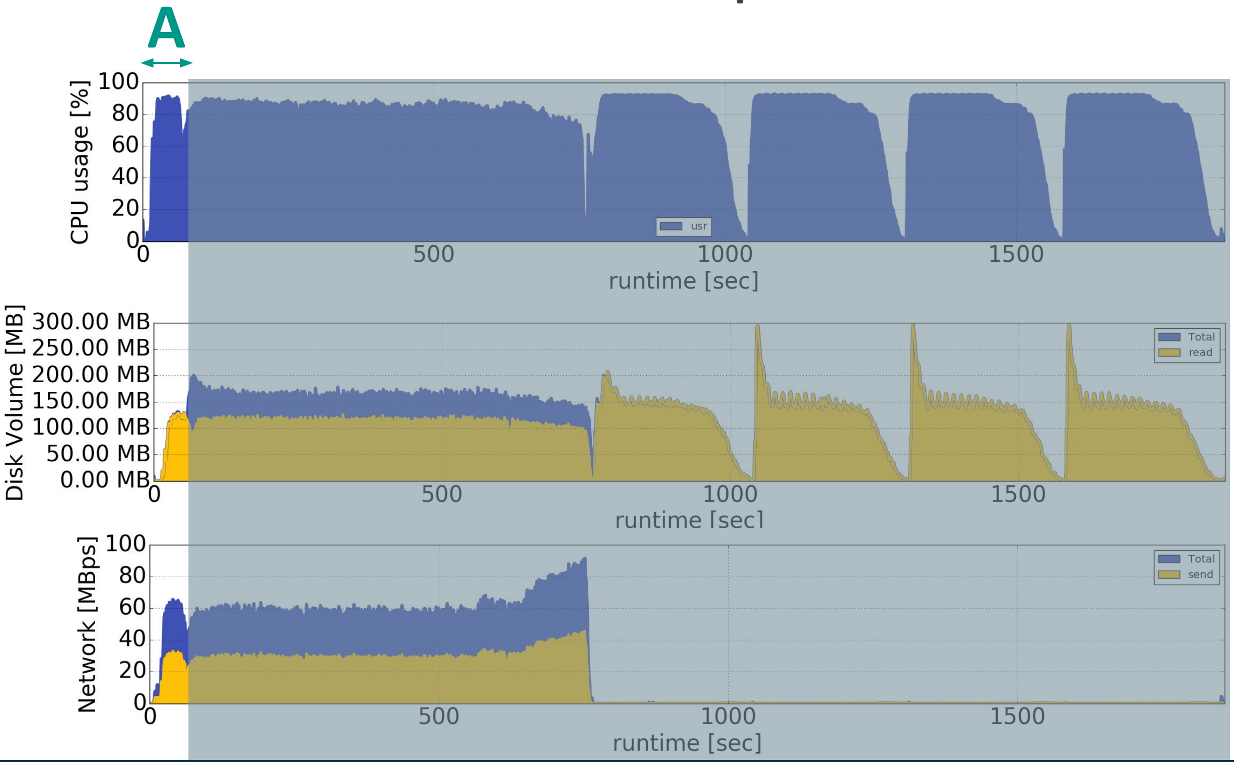
5 iterations

1TB data

30 nodes

12GB RAM / node

Like This KMeans Experiment! Step A



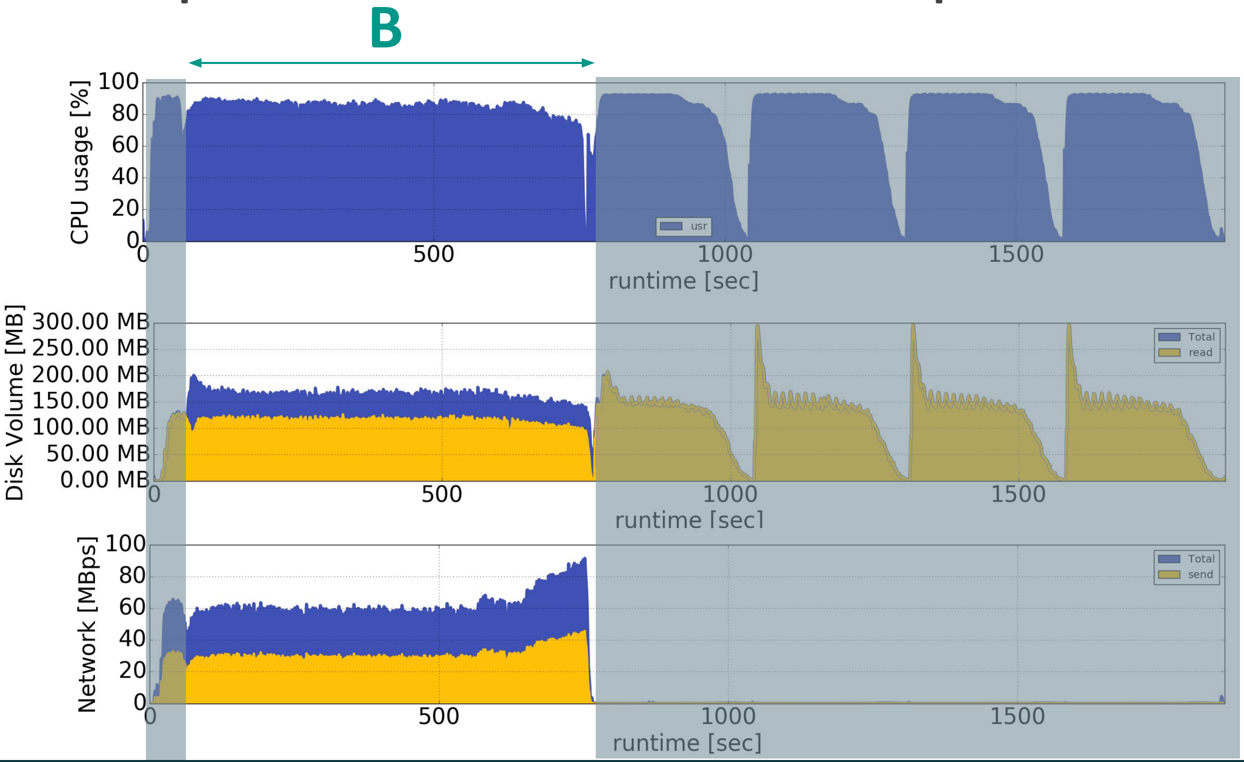
A:

Building Execution Pipeline

Reading from Source

Map Points to Breeze

Step **B** - The master sends partitions across the cluster



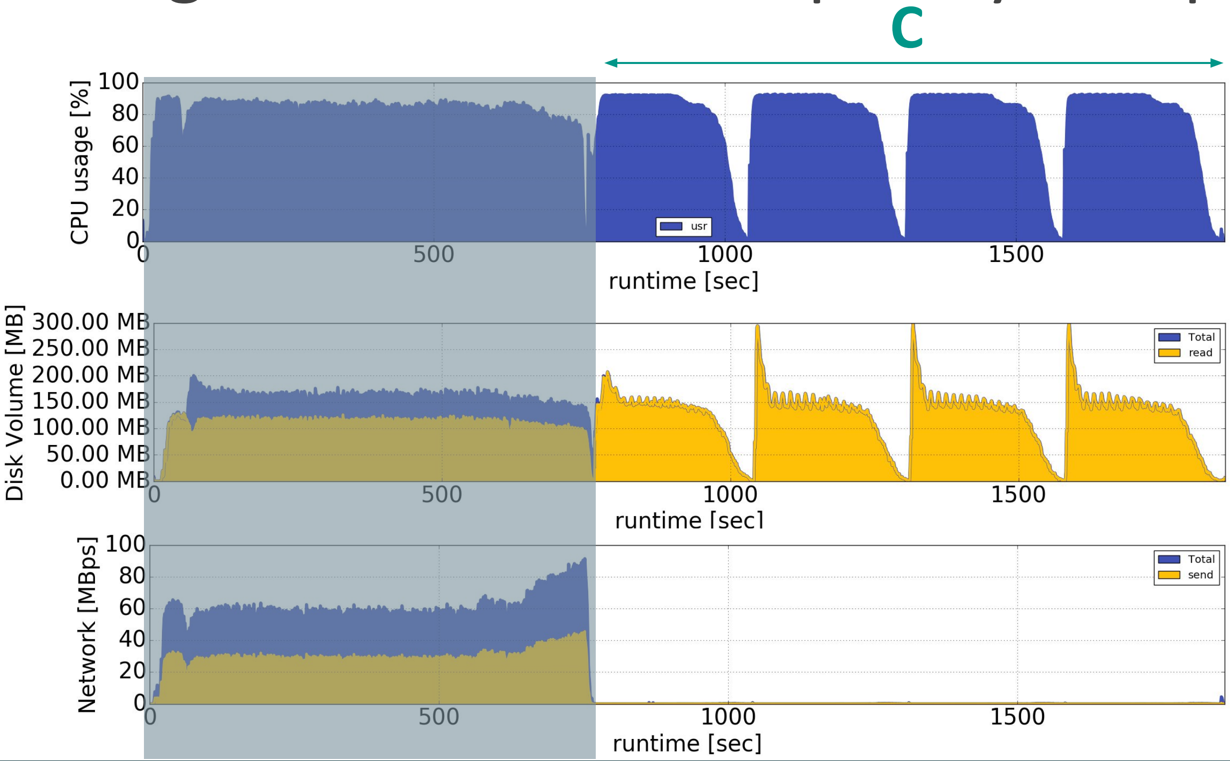
B:

Repartition Data

Cache Data and **spill** to Disk

First Iteration Execution

Regular Iterations Frequency - Step C



C:

Next Iterations (2 to 5)

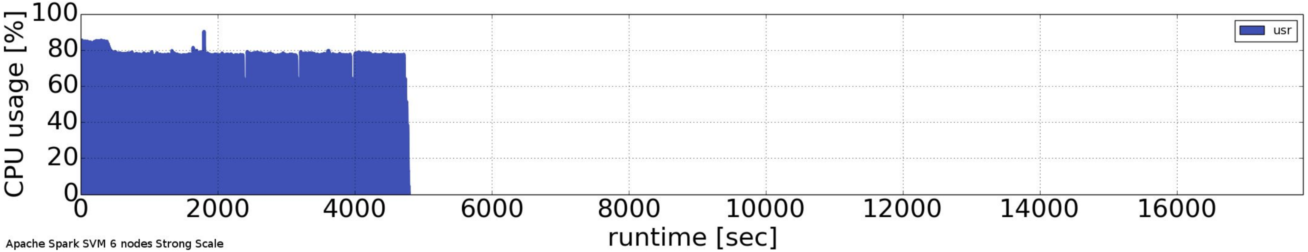
Read **spilled** data

Broadcast the model

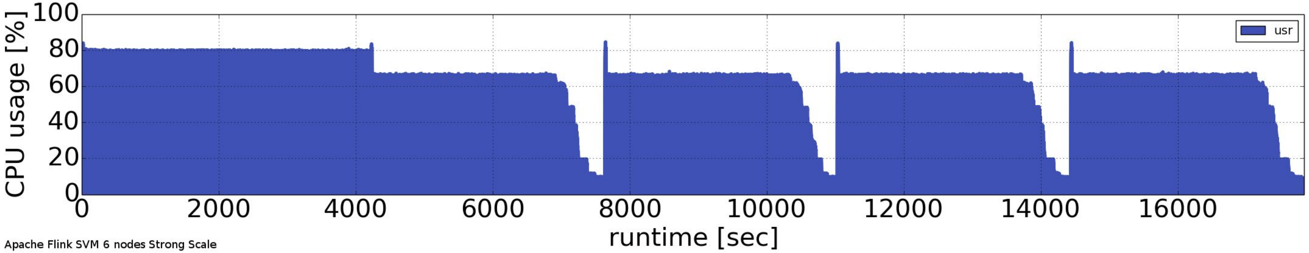
Produce **Sink** (write to disk)

How Spark outperforms Flink - #1 Repartitioning

SVM - 6 Nodes - 212GB Dataset - 5 iterations - 30 nodes - 12GB RAM / node

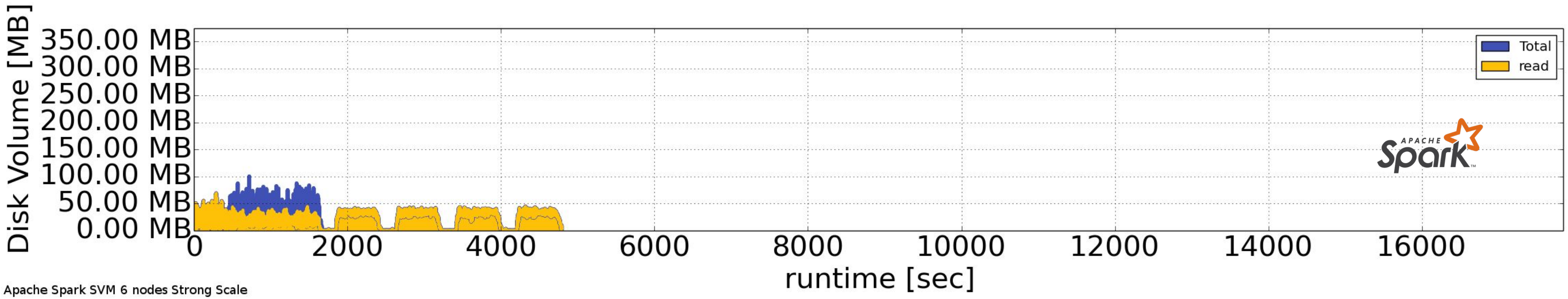


Apache Spark SVM 6 nodes Strong Scale

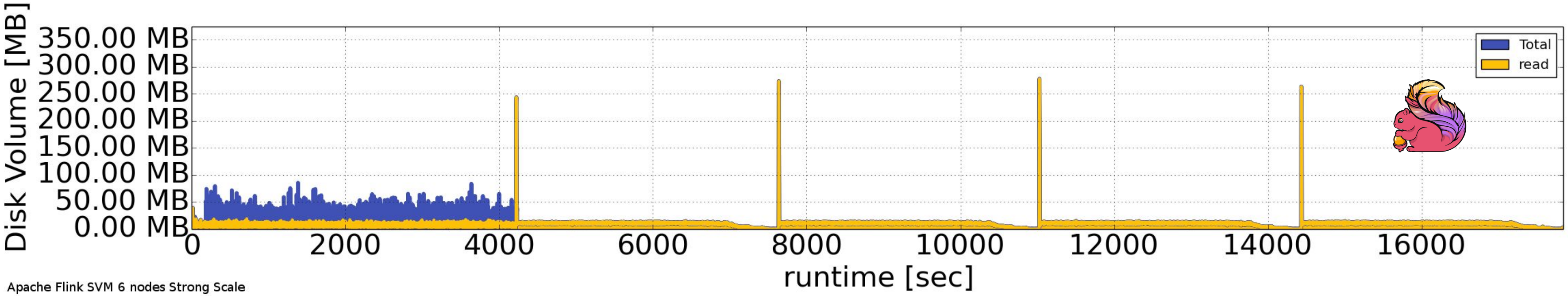


Apache Flink SVM 6 nodes Strong Scale

#1 Repartitioning - Distributed-to-Distributed

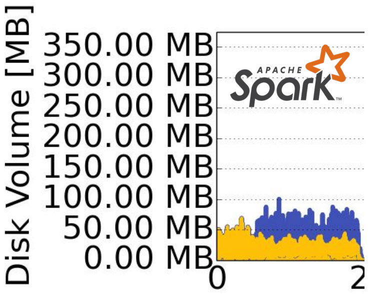


Apache Spark SVM 6 nodes Strong Scale

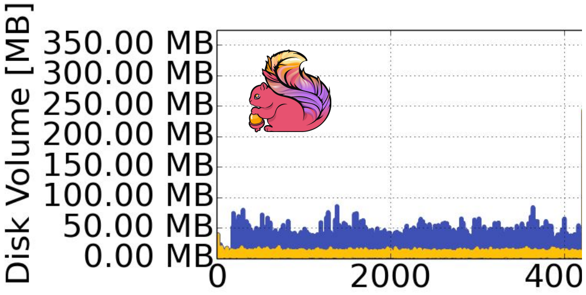


Apache Flink SVM 6 nodes Strong Scale

#1 Repartitioning - What Spark does



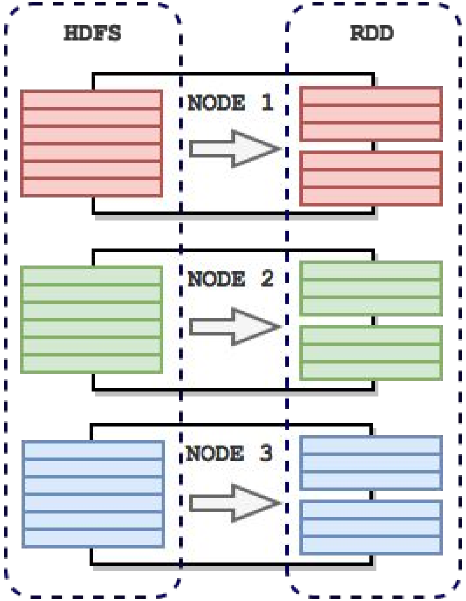
Apache Spark SVM 6 nodes Strong Scale



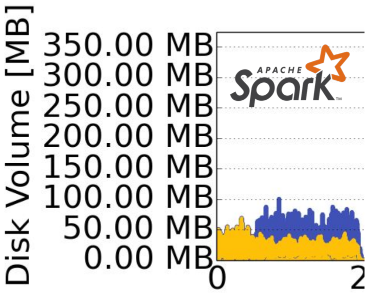
Apache Flink SVM 6 nodes Strong Scale

Narrow Transformation

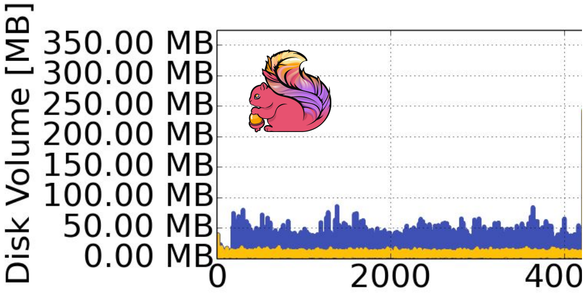
GOAL - no data movements



#1 Repartitioning - What Flink does

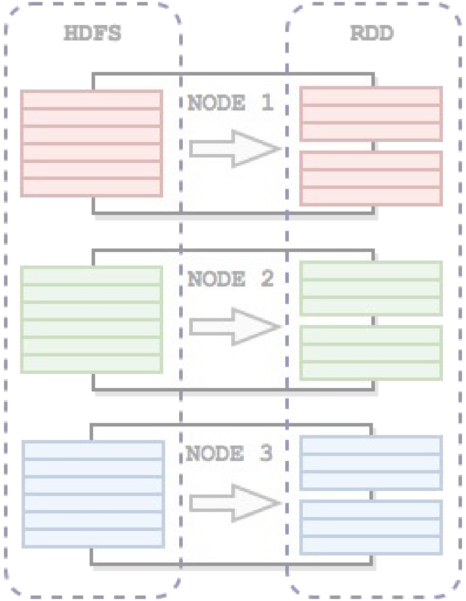


Apache Spark SVM 6 nodes Strong Scale

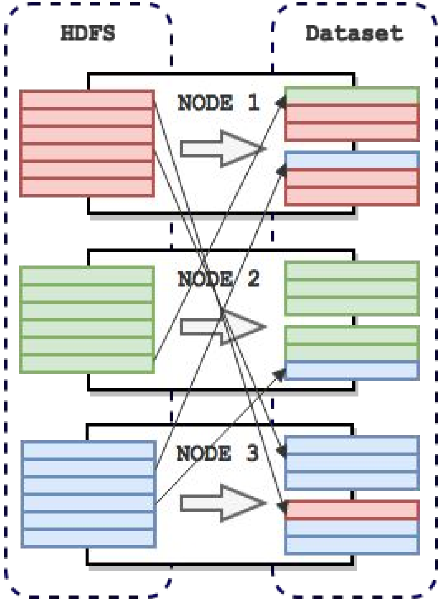


Apache Flink SVM 6 nodes Strong Scale

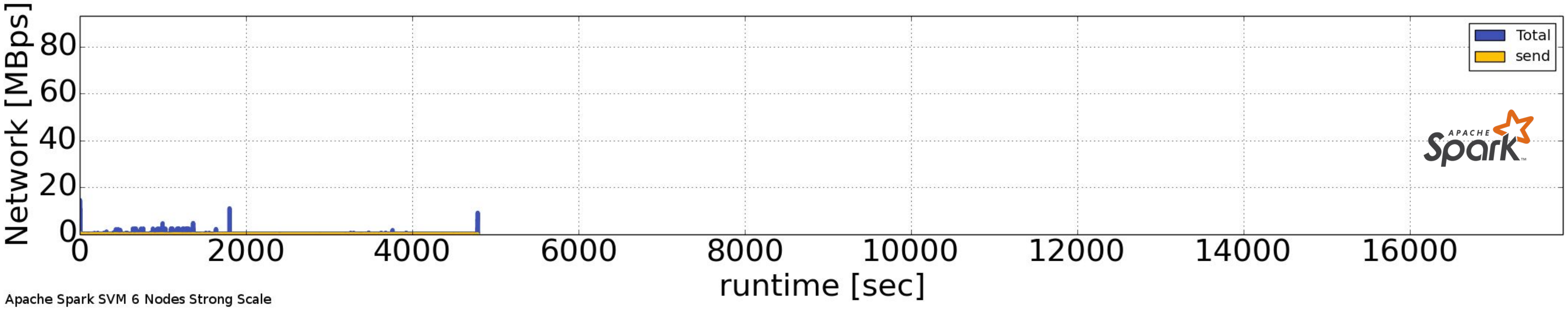
Narrow Transformation
GOAL - no data movements



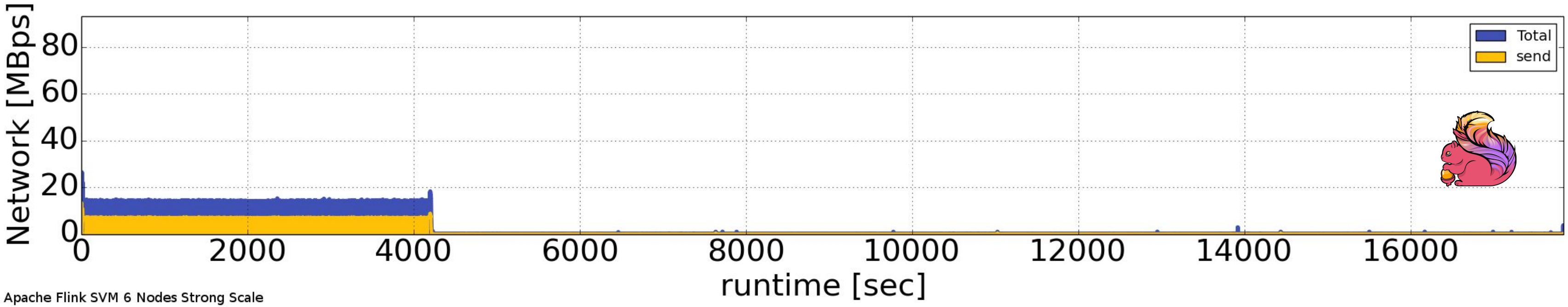
Shuffling data between nodes



#1 Repartitioning - Flink Network Overhead



Apache Spark SVM 6 Nodes Strong Scale



Apache Flink SVM 6 Nodes Strong Scale

Additional Relevants

- **#2 Caching** Flink Issue - FLINK-1730
  https://issues.apache.org/jira/browse/FLINK-1730
  - Spark - user-defined caching returns faster intra-iteration timing
  - Flink manages caching internally (**Bulk Iterations**) and it is slower when the data is not **Big**
- **#3 Broadcasting** Flink Improvements Proposal - FLIP-5
  https://cwiki.apache.org/confluence/display/FLINK/FLIP-5%3A+Only+send+data+to+each+taskmanager+once+for+broadcasts
  - Flink Broadcast brings communication overhead
  - Anyway it was not critical to this benchmark

Conclusions

Main Considerations

- Currently Spark is the right choice for **batch** purposes
  - and now Spark 2.0 …
- Flink was *born to stream* and is **growing** along streaming
  - need to find a *tradeoff*
- Flink put first **robustness** and **availability**
  - and it masters ***join, hashing, grouping***
- Spark put first **performance** and **efficiency**

Thank You

Media

- https://blog.websummit.net/berlin-the-startup-city-guide/ - background image - pag.2
- https://whatsthebigdata.com/2013/03/18/processing-big-data-the-google-way/ - background image - pag.4
- https://www.mapr.com/sites/default/files/blogimages/Spark-core-stack-DB.jpg - spark stack - pag.11
- https://flink.apache.org/img/flink-stack-frontpage.png - flink stack -pag.11
- http://www.hostingtalk.it/wp-content/uploads/2016/04/machine_learning.png - background image - pag. 28
- http://static.wixstatic.com/media/53defd_17c4b53bdda34dd89eed13867b9cc1aa~mv2.jpg - background image - pag.51
- http://www.trustsecurity.co.uk/admin/resources/monitoring-w680h300.jpg - background image - pag.61


BONUS SLIDES
YES
memegenerator.net

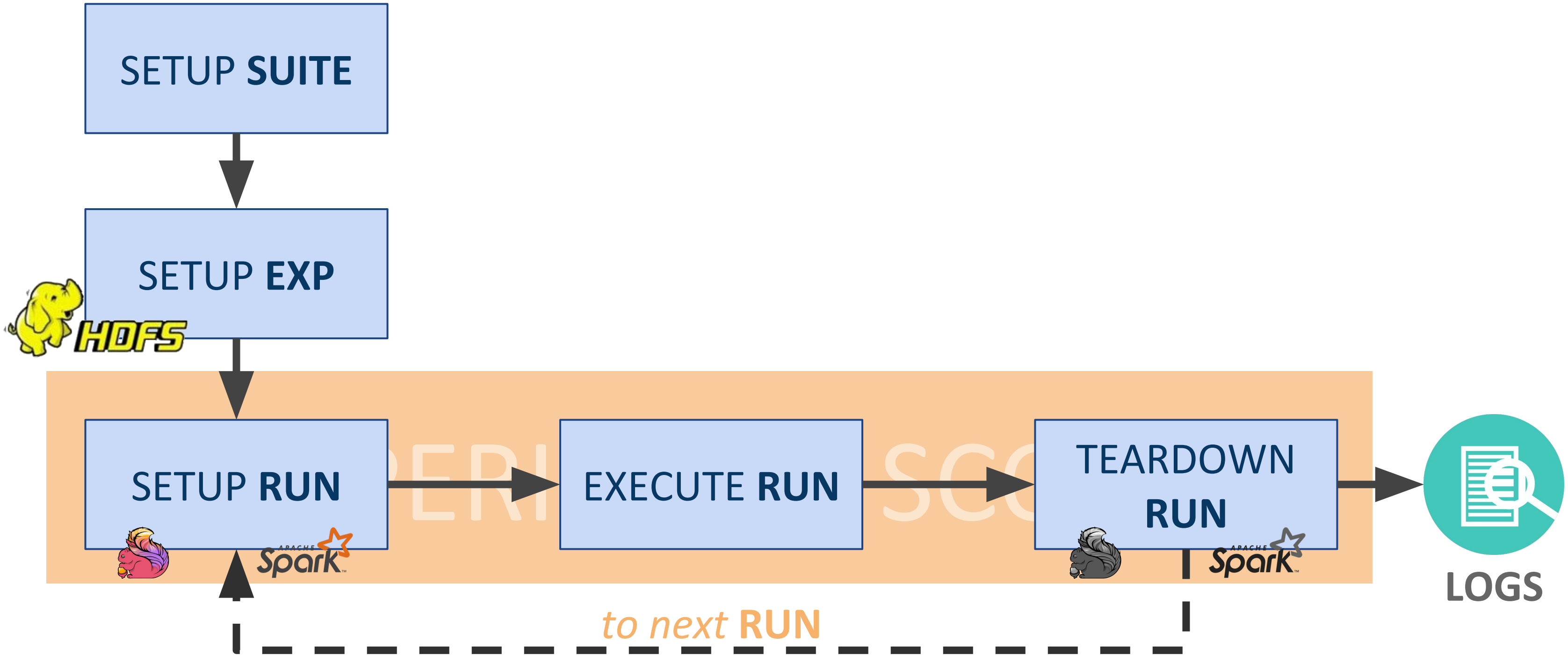
Peel Framework Deploy Flow

Peel execution flow - the `suite:run` command

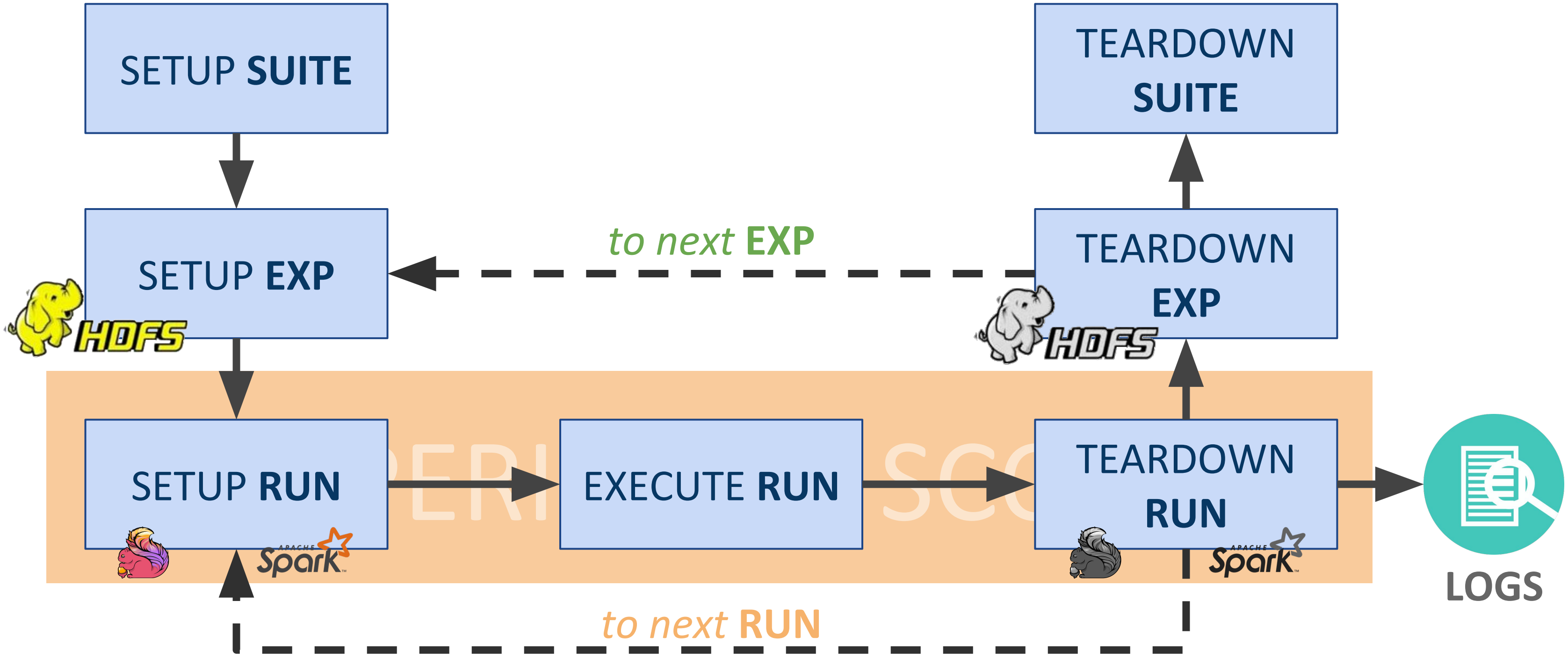
SETUP **SUITE**

Peel execution flow - turn on systems

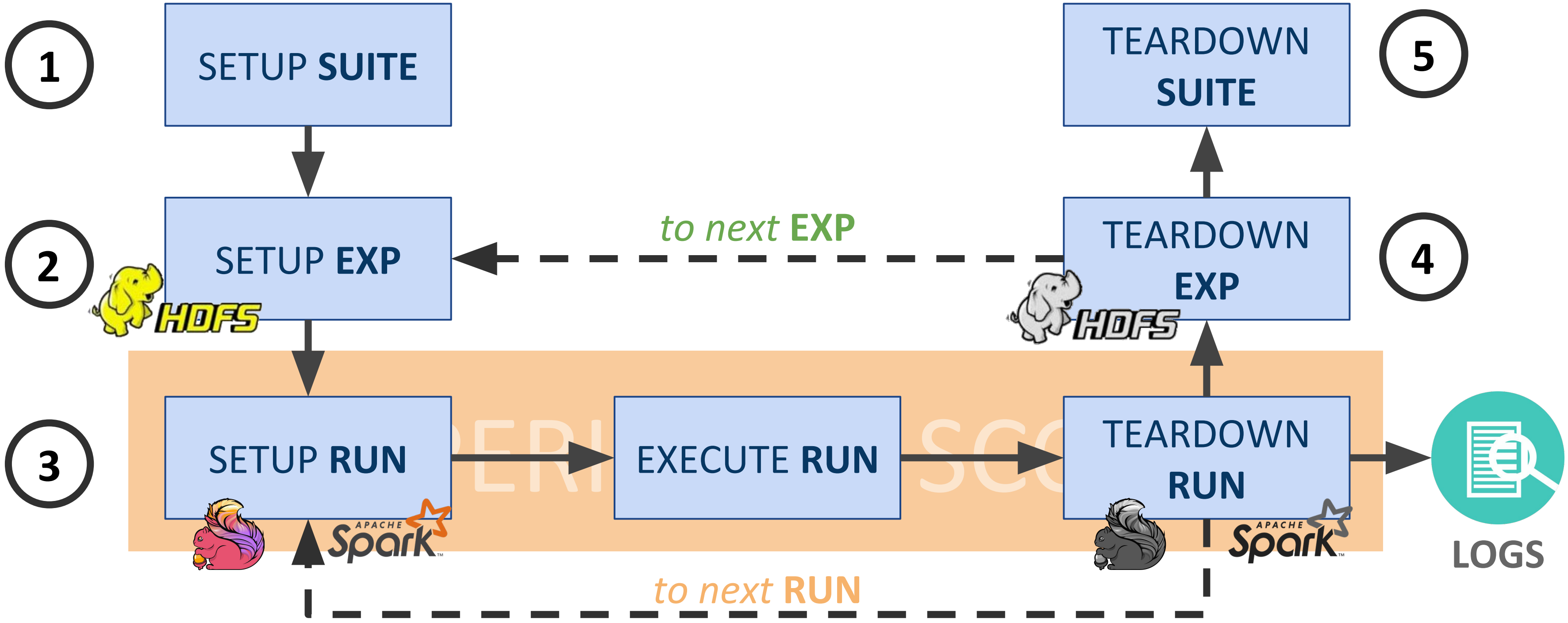
Peel execution flow - collect logs and run again



Peel execution flow - turn off systems



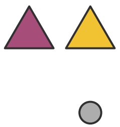
Peel execution flow - It enables **context** fairness



The KMeans Theory

Building the Experiment Pipeline - **KMeans Example**

KMEANS *clustering*
find new classes from unlabeled data by grouping



$$C = \{c_1, c_2, \ldots, c_k\}$$

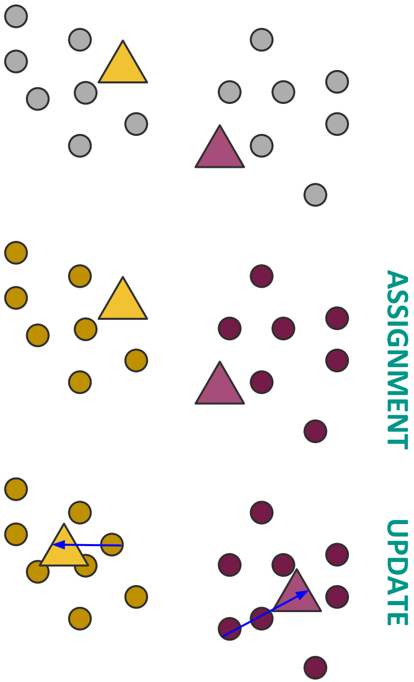
$$X = \{x_1, x_2, \ldots, x_n\}$$

ASSIGNMENT STEP
re-partition datapoints according to centroids

$$\varphi_X(C) = \sum_{x \in X} d(x, C)^2$$

UPDATE STEP
retrieve new centroids by datapoints location mean

$$c_i^{(t+1)} = \frac{1}{S_i^{(t)}} \sum_{x_j \in S_i^{(t)}} x_j$$



KMeans Workload Code

Building the KMeans Pipeline - **Data Structures**

INIT CENTERS

DATA

EXPERIMENT SCOPE

```
def getBreezeDataSet(sc: SparkContext, inputPath: String, numPartitions: Int): RDD[BDVector[Double]] = {
  sc
    .textFile(inputPath, numPartitions)
    .map(s => {
      val point: BDVector[Double] = BDVector[Double](s.split(",").map(_.toDouble))
      point
    })
}
```

```
def getBreezeDataSet(env: ExecutionEnvironment, inputPath: String): DataSet[BDVector[Double]] = {
  env
    .readTextFile(inputPath)
    .map(s => {
      val point: BDVector[Double] = BDVector[Double](s.split(",").map(_.toDouble))
      point
    })
}
```

* the project is developed in Scala

KMeans Iteration #1

```scala
@ForwardedFields(Array("*->_2"))
final class CommonSelectNearestCenter extends RichMapFunction[BDVector[Double], (Int, BDVector[Double], Long)] {
 private var centroids: Traversable[(Int, BDVector[Double])] = null

 /** reads centroids and indexing values from the broadcasted set **/
 override def open(parameters: Configuration): Unit = {
  centroids = getRuntimeContext.getBroadcastVariable[(Int, BDVector[Double])]("centroids").asScala
 }

 override def map(point: BDVector[Double]): (Int, BDVector[Double], Long) = {
  var minDistance: Double = Double.MaxValue
  var closestCentroidId: Int = -1
  for ((idx, centroid) <- centroids) {
   val distance = squaredDistance(point, centroid)
   if (distance < minDistance) {
    minDistance = distance
    closestCentroidId = idx
   } }
  (closestCentroidId, point, 1L)
 } }

val finalCentroids: DataSet[(Int, BDVector[Double])] = centroids.iterate(iterations) { currentCentroids =>
 val newCentroids = points.map(new CommonSelectNearestCenter).withBroadcastSet(currentCentroids, "centroids")
/** … **/
```

```
while(iterations < maxIterations) {

 val bcCentroids = data.context.broadcast(currentCentroids)

 val newCentroids: RDD[(Int, (BDVector[Double], Long))] = data.map (point => {
  var minDistance: Double = Double.MaxValue
  var closestCentroidId: Int = -1
  val centers = bcCentroids.value

  centers.foreach(c => { // c = (idx, centroid)
  val distance = squaredDistance(point, c._2)
   if (distance < minDistance) {
    minDistance = distance
    closestCentroidId = c._1
   }
  })

  (closestCentroidId, (point, 1L))
 })

/** … **/
```

KMeans Iteration #1

```scala
val finalCentroids: DataSet[(Int, BDVector[Double])] = centroids.iterate(iterations) { currentCentroids =>
 val newCentroids = points
   .map(new CommonSelectNearestCenter).withBroadcastSet(currentCentroids, "centroids")
   .groupBy(0)
   .reduce((p1, p2) => {
     (p1._1, p1._2 + p2._2, p1._3 + p2._3)}).withForwardedFields("_1")
```

KMeans Iteration #2

```scala
/** … **/
(closestCentroidId, (point, 1L))
}).reduceByKey(mergeContribs)

type WeightedPoint = (BDVector[Double], Long)
def mergeContribs(x: WeightedPoint, y: WeightedPoint): WeightedPoint = {
 (x._1 + y._1, x._2 + y._2)
}
```

KMeans Iteration #2

```scala
val avgNewCentroids = newCentroids
 .map(x => {
   val avgCenter = x._2 / x._3.toDouble
   (x._1, avgCenter)
 }).withForwardedFields("_1")

avgNewCentroids
```

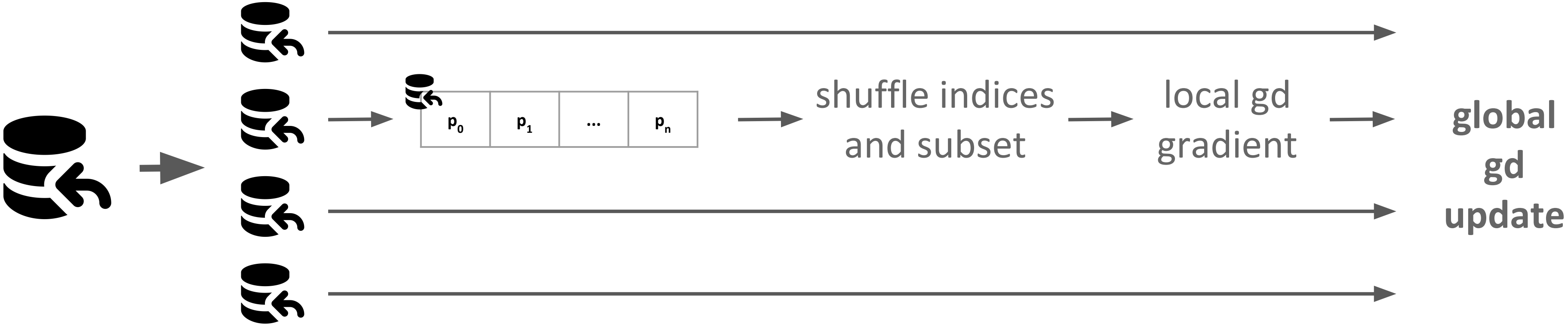
```scala
currentCentroids = newCentroids
 .map(x => {
   val (center, count) = x._2
   val avgCenter = center / count.toDouble
   (x._1, avgCenter)
 }).collect()

iterations += 1
```

When Experiment Definition Goes Wrong …

SVM and Gradient Descent: what we wanted to do

ORIGINAL IDEA → *Gradient Descent + mini-batching*

1. sampling not comparable → Custom and Common Sampler
2. `mapPartitions` over mini-batches

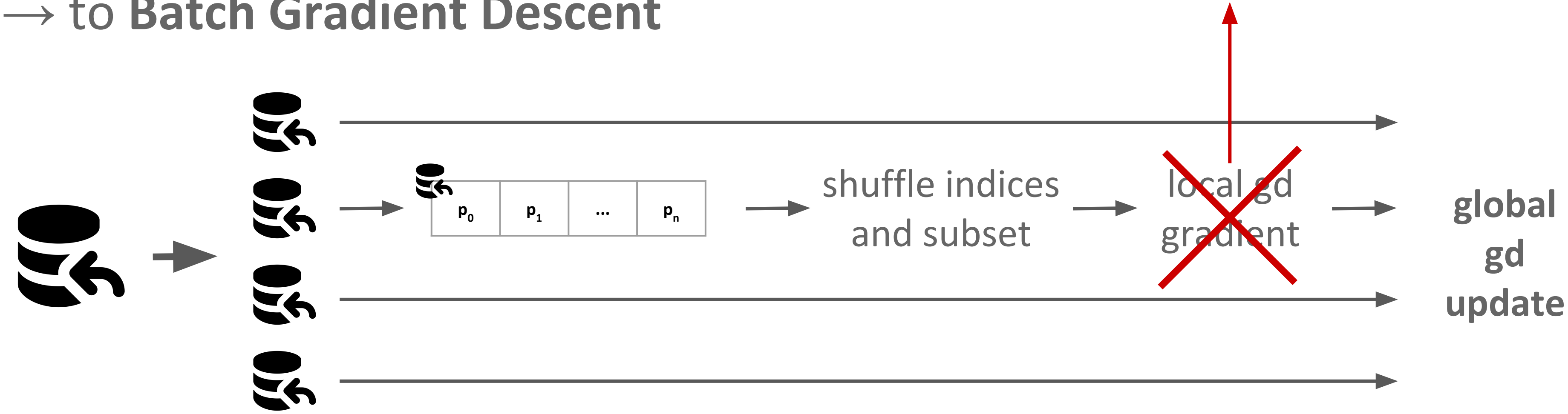


SVM and Gradient Descent: what we did

ISSUE

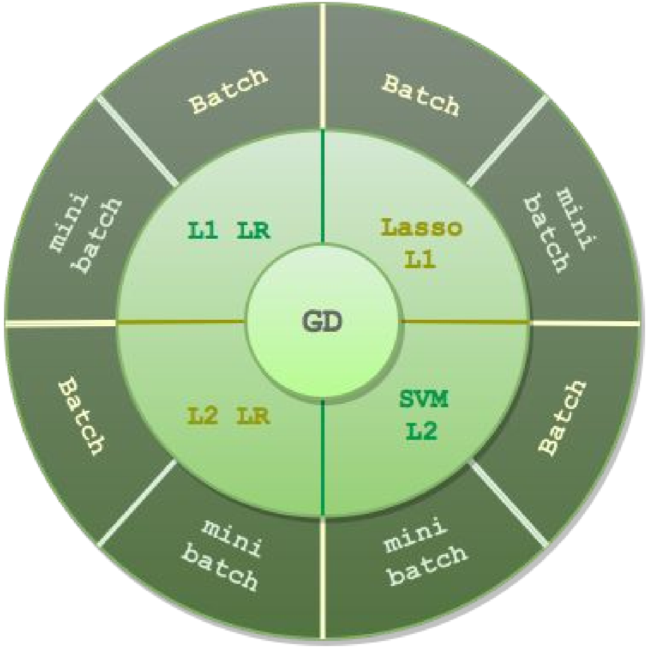
Spark not able to Run `mapPartitions` → **OutOfMemory Exception**

→ to **Batch Gradient Descent**



When Experiment Definition Goes Wrong …

The Supervised Learning Framework



Other Results

Spark versus Flink Summary

Multiple Linear Regression

Spark v Flink
8 - 1

- Spark 63% outperforms Flink
- Flink 74% faster on critic resources
- FlinkML provides better runtimes

Support Vector Machine

Spark v Flink
16 - 0

- Spark 71% outperforms Flink
- Flink likes MORE Data

KMeans

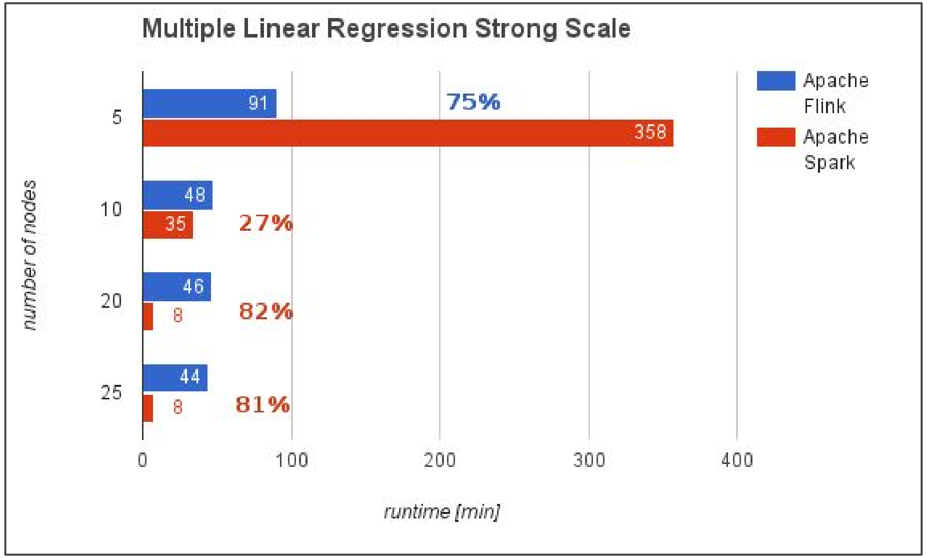
Spark v Flink
10 - 7

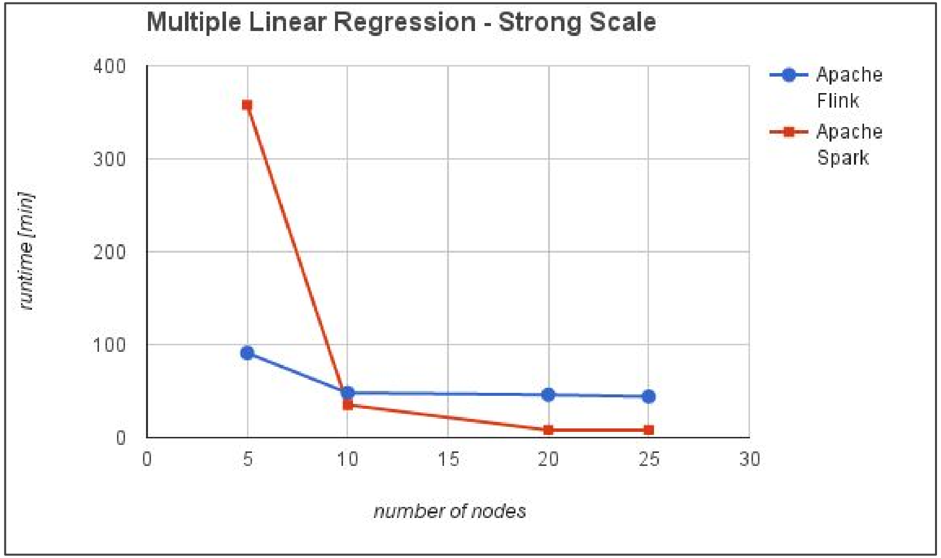
- Similar Performance
- Flink definitely likes MORE data
- Flink 11% faster on critic resources

Alternating Least Squares

NOT COMPARABLE

Multiple Linear Regression **strong scale**





CLOUD-11 (25nodes)

- **28GB** RAM per node
- **16** core CPU per node

DATASET INFO

- no. datapoints **10^7**
- model size **1000**
- data size **80GB**
- sparsity **30%**

ALGORITHM INFO

- Iterations **100**

Spark versus **Flink** Summary

34 RUNTIME WINS **8**

Multiple Linear Regression

Spark v Flink

8 - 1

- Spark 63% outperforms Flink
- Flink 74% faster on critic resources
- FlinkML provides better runtimes

Support Vector Machine

Spark v Flink

16 - 0

- Spark 71% outperforms Flink
- Flink likes MORE Data

KMeans

Spark v Flink

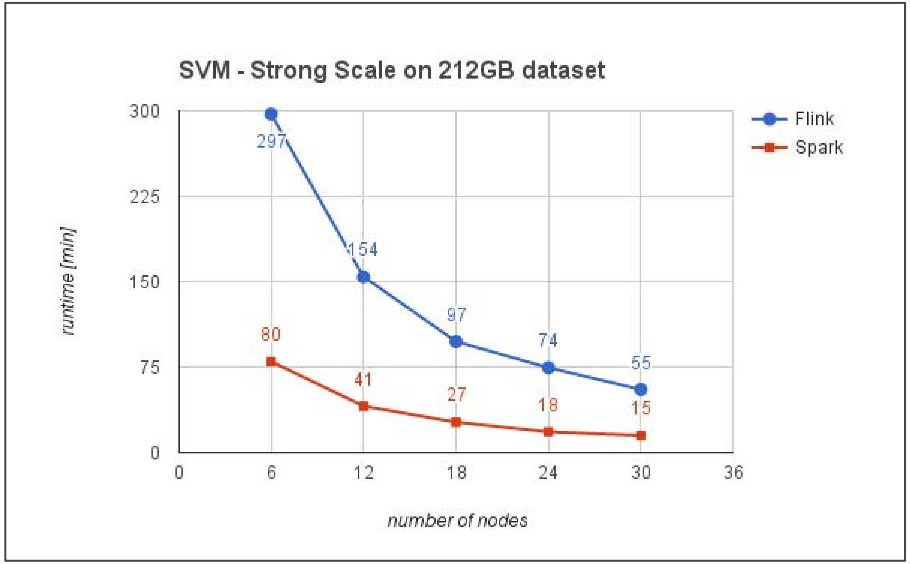
10 - 7

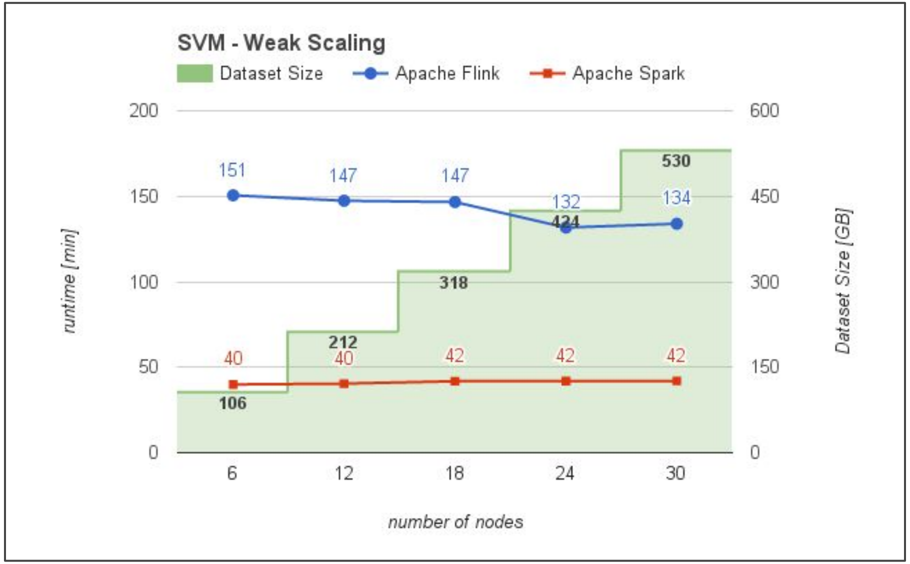
- Similar Performance
- Flink definitely likes MORE data
- Flink 11% faster on critic resources

Alternating Least Squares

NOT COMPARABLE

Support Vector Machine **strong scale** and **weak scale**





- **12GB** RAM per node
- **8** core CPU per node
- sparsity **0%**
- model size **1000**

Future Developments

Future Improvements

- Complete not comparable benchmarking
- Redefine ALS benchmarking
- Add not Included Systems
- Improve *shee* and integrate it in *peel* framework