BDT309

Data Science & Best Practices for Apache Spark on Amazon EMR

Jonathan Fritz, Sr. Product Manager – Amazon EMR
Manjeet Chayel, Specialist Solutions Architect

October 2015

amazon web services

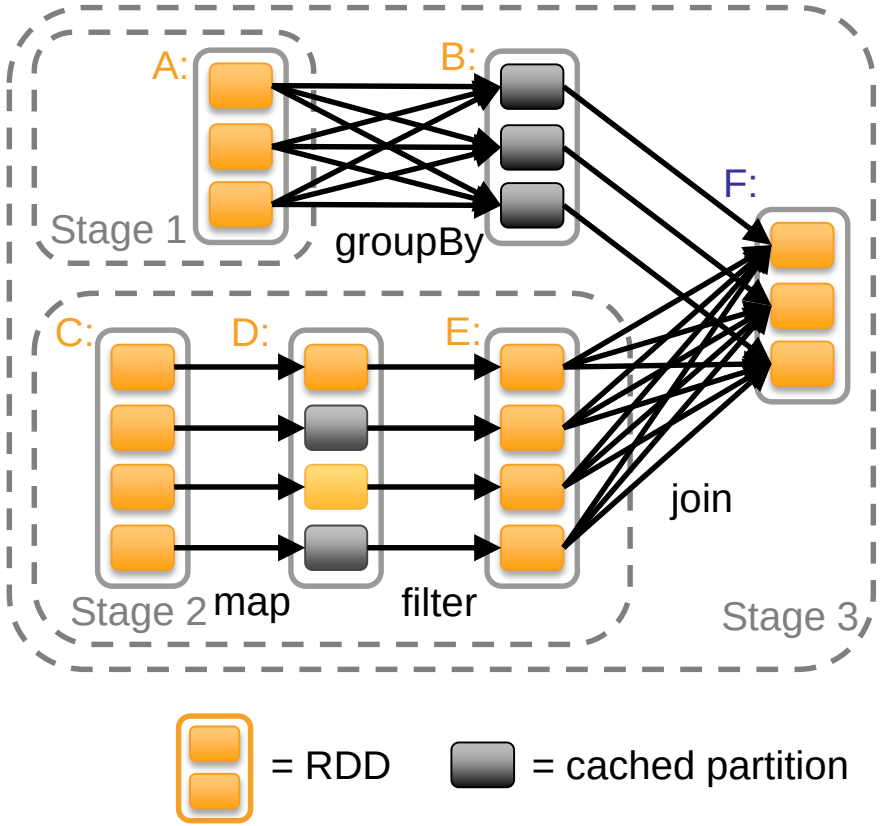
What to Expect from the Session

- Data science with Apache Spark
- Running Spark on Amazon EMR
- Customer use cases and architectures
- Best practices for running Spark
- Demo: Using Apache Zeppelin to analyze US domestic flights dataset

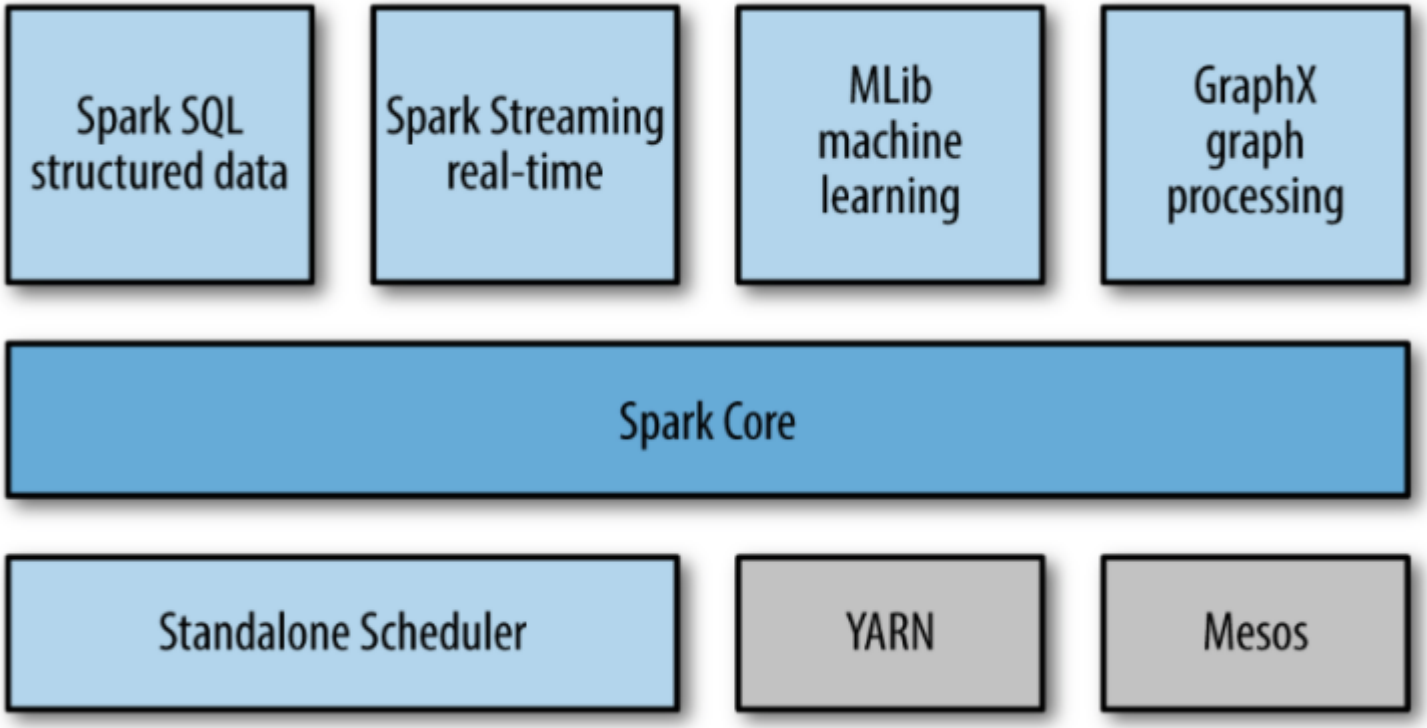

Spark

Spark is fast

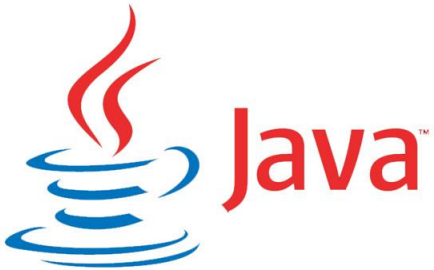
- Massively parallel
- Uses DAGs instead of map-reduce for execution
- Minimizes I/O by storing data in RDDs in memory
- Partitioning-aware to avoid network-intensive shuffle



Spark components to match your use case

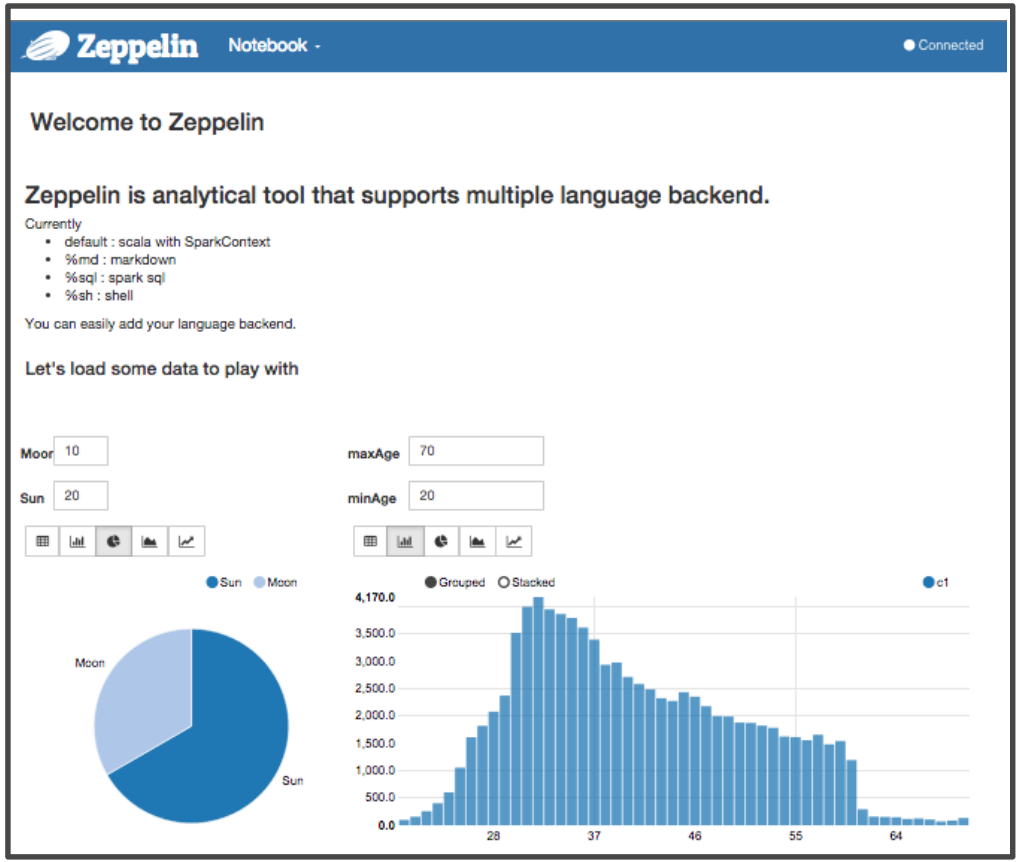


Spark speaks your language



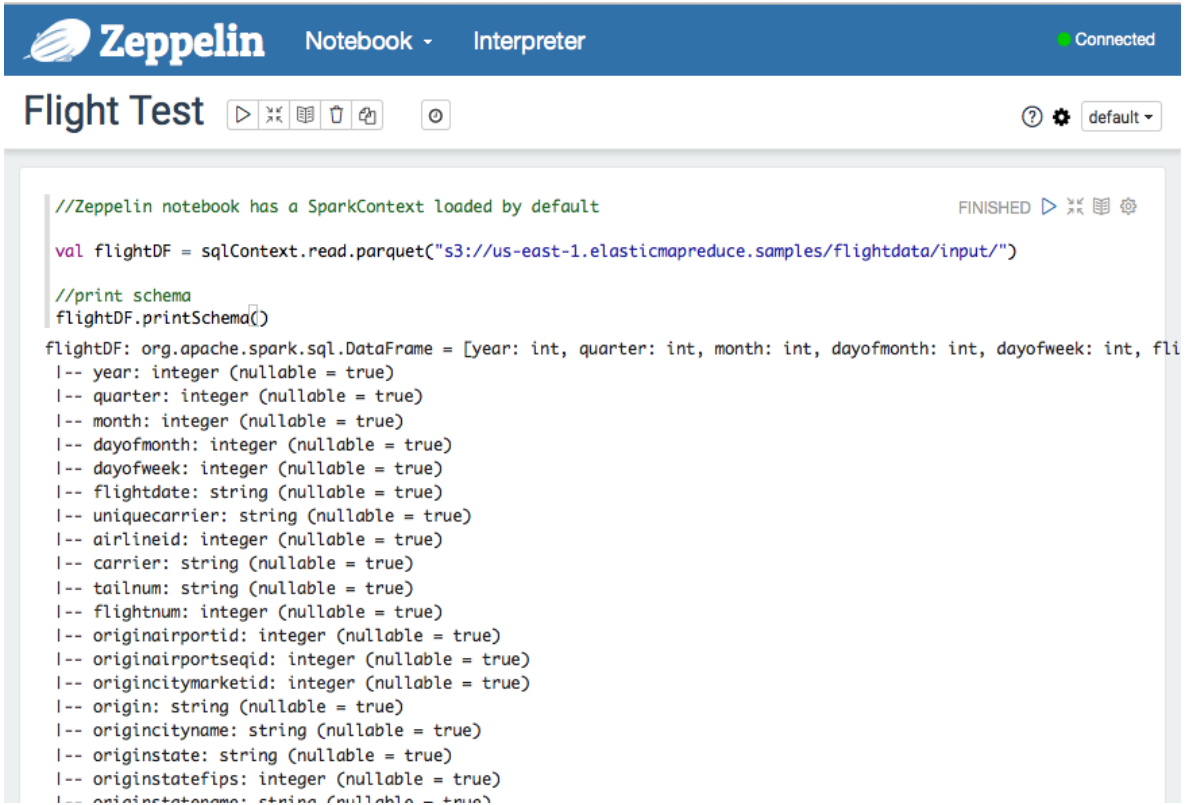
And more!

Apache Zeppelin notebook to develop queries



Now available on Amazon EMR 4.1.0!

Use DataFrames to easily interact with data



- Distributed collection of data organized in columns
- An extension of the existing RDD API
- Optimized for query execution

Easily create DataFrames from many formats

Load data with the Spark SQL Data Sources API

Additional libraries at spark-packages.org

Sample DataFrame manipulations

Zeppelin Notebook Interpreter Connected

Flight Test

default

FINISHED

```
flightDF.groupBy("uniquecarrier").count.show()
```

```
+-------------+--------+
|uniquecarrier|   count|
+-------------+--------+
|           AA|18346054|
|       PA (1)|  316167|
|           TW| 3757747|
|           NK|   26232|
|           TZ|  208420|
|           HA|  721233|
|           AS| 3798315|
|           B6| 2207571|
|           UA|15979963|
|           NW|10585760|
|           HP| 3636682|
|           US|16634929|
|           OH| 1765828|
|           OO| 6824739|
|           PI|  873957|
|           VX|  183212|
|           CO| 8888536|
|       ML (1)|   70622|
|           PS|   83617|
|           WN|23133835|
+-------------+--------+
only showing top 20 rows
```

Took 6 seconds. (outdated)

FINISHED

```
flightDF.describe("distance")
.show()
```

```
+-------+-----------------+
|summary|         distance|
+-------+-----------------+
|  count|        162212419|
|   mean|715.5467073701675|
| stddev|558.8073376162126|
|    min|                0|
|    max|             4983|
+-------+-----------------+
```

Took 10 seconds. (outdated)

FINISHED

```
flightDF.filter(flightDF("depdelay") > 15).
groupBy("origincityname")
.count().sort($"count".desc).show(10)
```

```
+--------------------+-------+
|      origincityname|  count|
+--------------------+-------+
|         Chicago, IL|2082418|
|         Atlanta, GA|1570203|
|Dallas/Fort Worth...|1287501|
|         Houston, TX| 871484|
|          Denver, CO| 848630|
|     Los Angeles, CA| 848212|
|        New York, NY| 826379|
|         Phoenix, AZ| 770063|
|   San Francisco, CA| 679916|
|          Newark, NJ| 657472|
+--------------------+-------+
only showing top 10 rows
```

Took 18 seconds. (outdated)

Use DataFrames for machine learning

```
// Prepare training documents from a list of (id, text, label) tuple
val training = sqlContext.createDataFrame(Seq(
  (0L, "a b c d e spark", 1.0),
  (1L, "b d", 0.0),
  (2L, "spark f g h", 1.0),
  (3L, "hadoop mapreduce", 0.0)
)).toDF("id", "text", "label")


// Configure an ML pipeline, which consists of three stages: tokeni
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")
val hashingTF = new HashingTF()
  .setNumFeatures(1000)
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegre
  .setMaxIter(10)
  .setRegParam(0.01)
val pipeline = new Pipeli
  .setStages(Array(tokeni


// Fit the pipeline to tra
val model = pipeline.fi
```

```
Fit                    local
final                  keyword
finally                keyword
Configure              local
classification         local
NoSuchFieldException   keyword
NoSuchFieldError       keyword
UnsatisfiedLinkError   keyword
```

- Spark ML libraries (replacing MLlib) use DataFrames as input/output for models
- Create ML pipelines with a variety of distributed algorithms

Create DataFrames on streaming data

- Access data in Spark Streaming DStream
- Create SQLContext on the SparkContext used for Spark Streaming application for ad hoc queries
- Incorporate DataFrame in Spark Streaming application

Use R to interact with DataFrames

```
>
> filterFlights <- filter(flights, flights$year > 2010)
> flightCount <- summarize(groupBy(filterFlights, filterFlights$origin), count = n(filterFlights$origin))
> head(arrange(flightCount, desc(flightCount$count)), num = 20L)
15/10/04 19:38:00 INFO MemoryStore: ensureFreeSpace(240536) called with curMem=1081933, maxMem=560993402
```

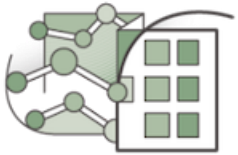
- SparkR package for using R to manipulate DataFrames
- Create SparkR applications or interactively use the SparkR shell (no Zeppelin support yet - ZEPPELIN-156)
- Comparable performance to Python and Scala DataFrames

Spark SQL

- Seamlessly mix SQL with Spark programs
- Uniform data access
- Hive compatibility – run Hive queries without modifications using HiveContext
- Connect through JDBC/ODBC

Running Spark on Amazon EMR

Focus on deriving insights from your data instead of manually configuring clusters



Easy to install and configure Spark

Quickly add and remove capacity

Spark submit or use Zeppelin UI

Hourly, reserved, or EC2 Spot pricing

Secured

Use S3 to decouple compute and storage

Launch the latest Spark version

July 15 – Spark 1.4.1 GA release

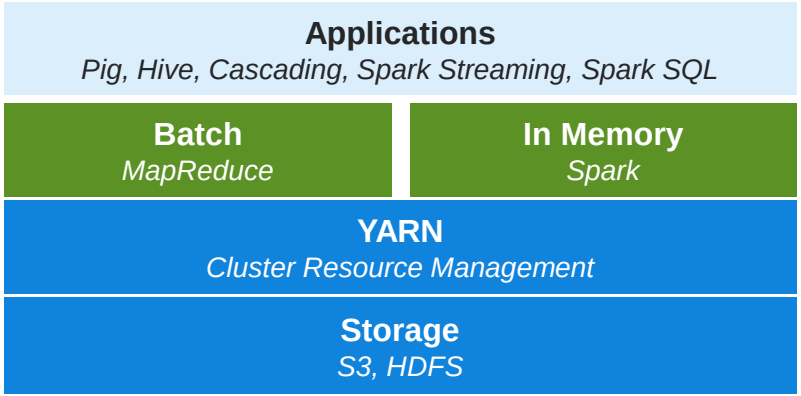
July 24 – Spark 1.4.1 available on Amazon EMR

September 9 – Spark 1.5.0 GA release

September 30 – Spark 1.5.0 available on Amazon EMR

< 3 week cadence with latest open source release

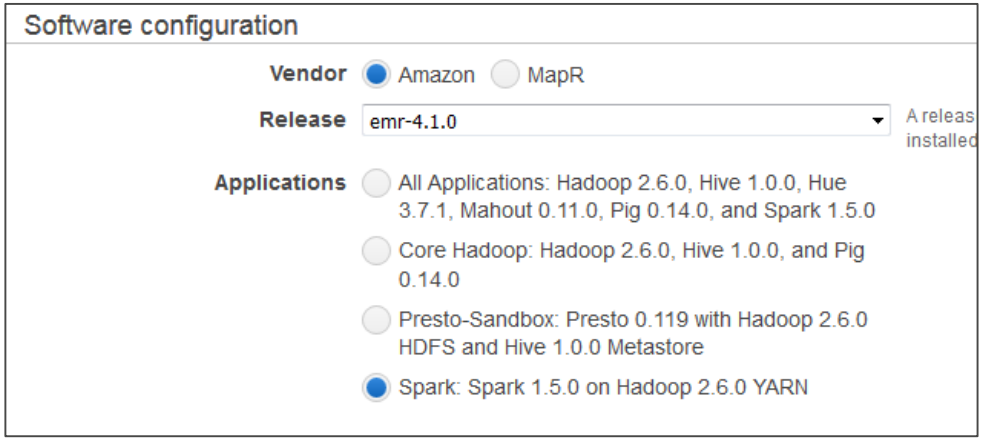
Amazon EMR runs Spark on YARN



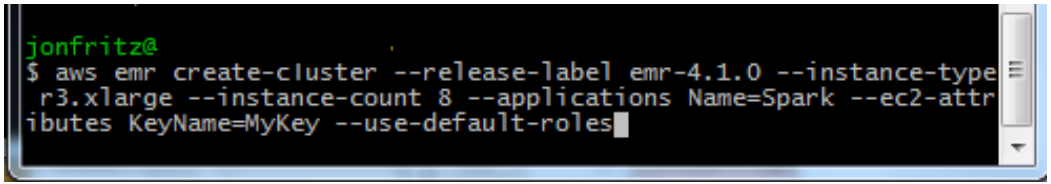
- Dynamically share and centrally configure the same pool of cluster resources across engines
- Schedulers for categorizing, isolating, and prioritizing workloads
- Choose the number of executors to use, or allow YARN to choose (dynamic allocation)
- Kerberos authentication

Create a fully configured cluster in minutes

AWS Management Console

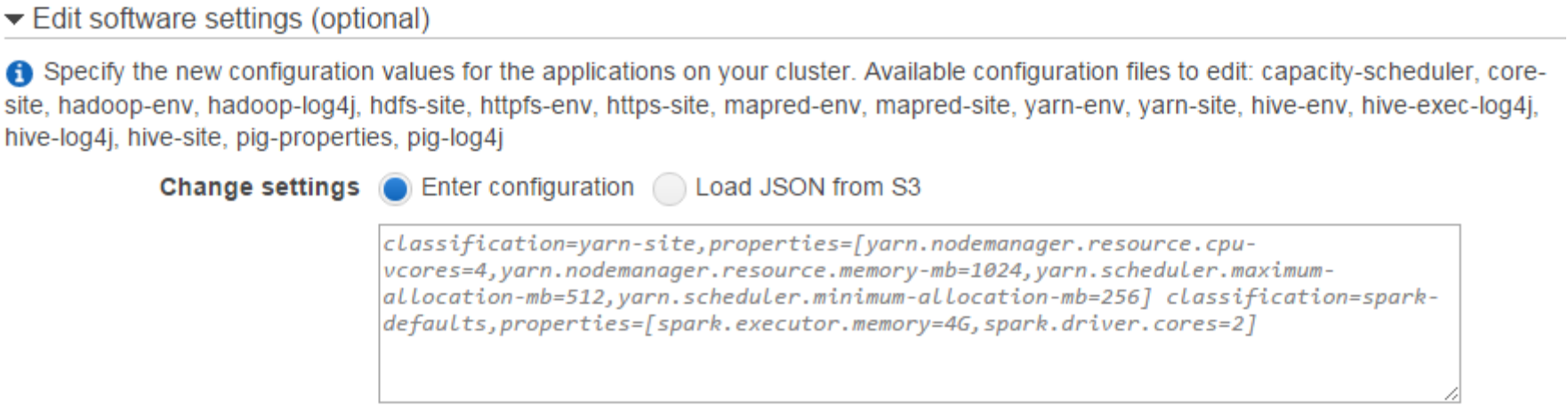


AWS Command Line Interface (CLI)

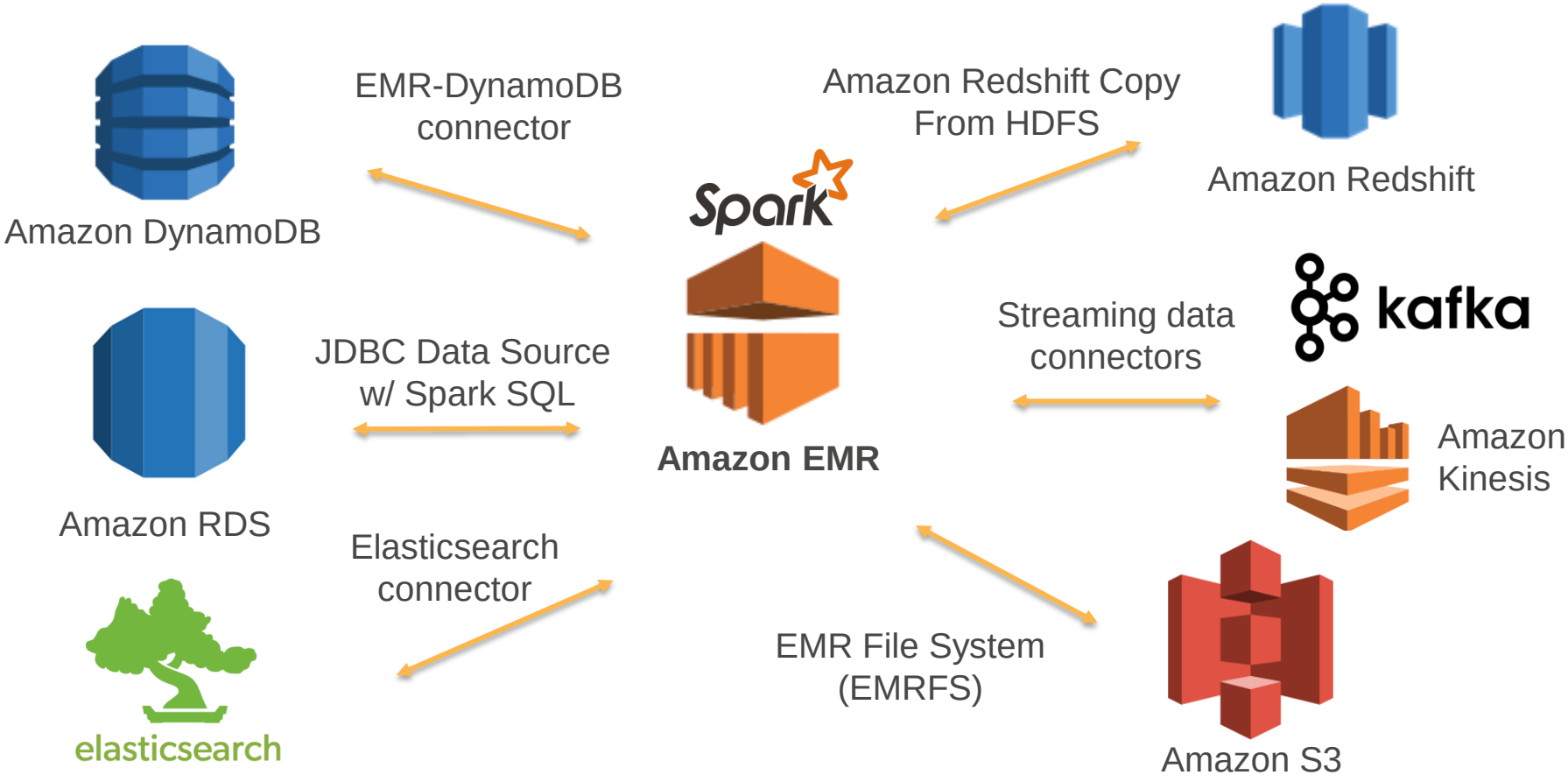


Or use an AWS SDK directly with the Amazon EMR API

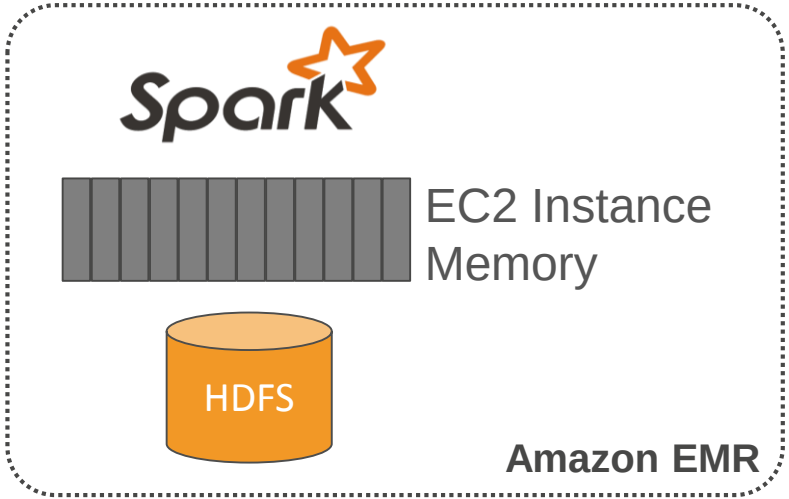
Or easily change your settings

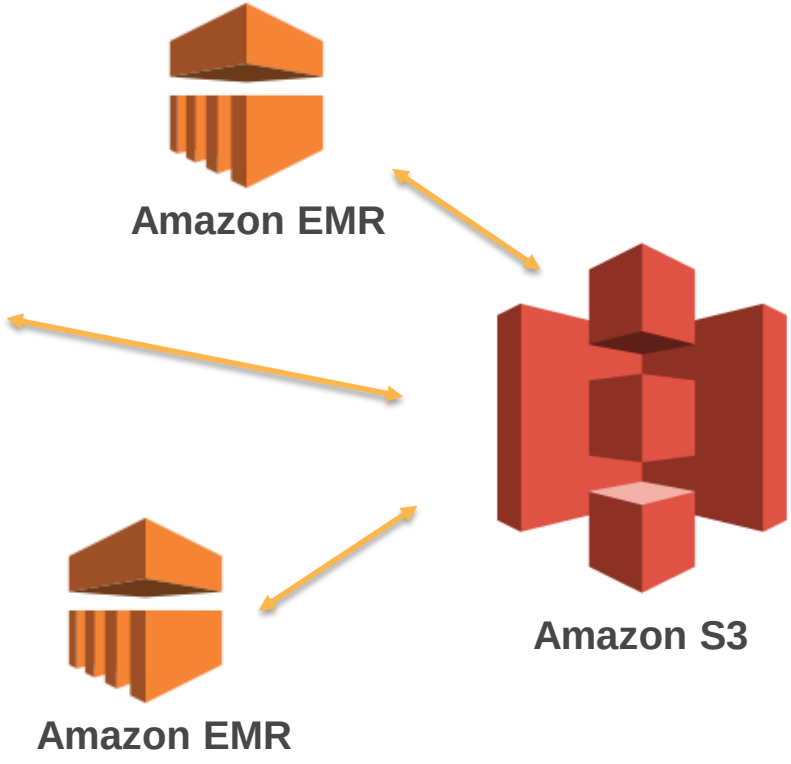


Many storage layers to choose from



Decouple compute and storage by using S3 as your data layer

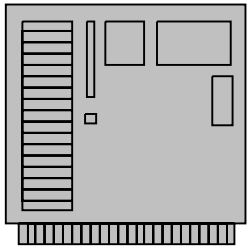




S3 is designed for 11 9's of durability and is massively scalable

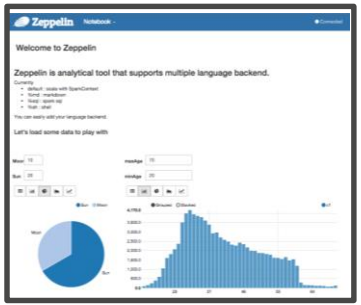
Easy to run your Spark workloads

Submit a Spark application



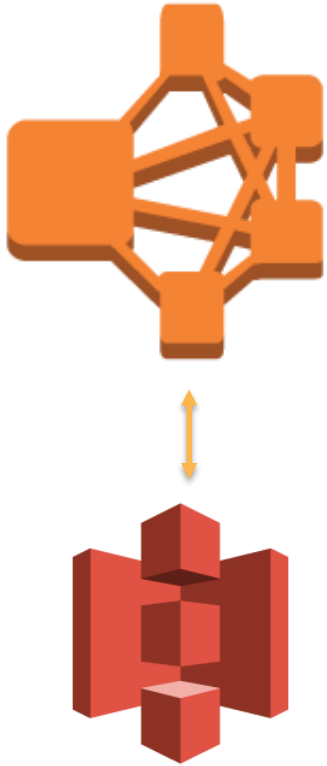
Amazon EMR Step API

Amazon EMR



SSH to master node (Spark Shell)

Secure Spark clusters – encryption at rest

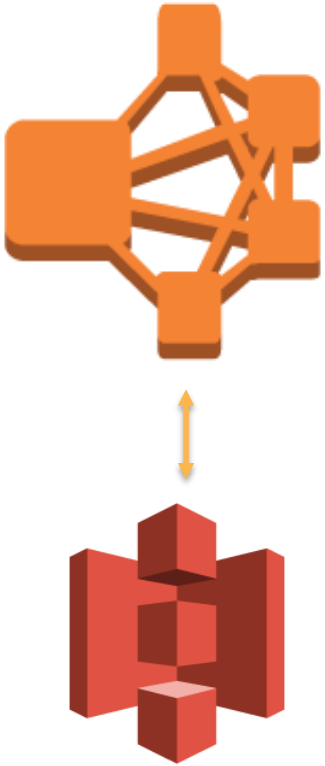


On-Cluster
HDFS transparent encryption (AES 256)
[new on release emr-4.1.0]

Local disk encryption for temporary files
using LUKS encryption via bootstrap action

Amazon S3
EMRFS support for Amazon S3 client-side
and server-side encryption (AES 256)

Secure Spark clusters – encryption in flight



Internode communication on-cluster
Blocks are encrypted in-transit in HDFS when using transparent encryption

Spark's Broadcast and FileServer services can use SSL. BlockTransferService (for shuffle) can't use SSL (SPARK-5682).

S3 to Amazon EMR cluster
Secure communication with SSL

Objects encrypted over the wire if using client-side encryption

Secure Spark clusters – additional features

Permissions:

- **Cluster level**: IAM roles for the Amazon EMR service and the cluster
- **Application level**: Kerberos (Spark on YARN only)
- **Amazon EMR service level**: IAM users

Access: VPC, security groups

Auditing: AWS CloudTrail

Customer use cases

Some of our customers running Spark on Amazon EMR

Machine learning & ad targeting

Web analytics

Ad targeting & recommendations

App search

Security event streaming

Revenue forecasting

Predictive marketing

Personalization

The Washington Post

Kafka

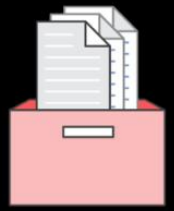
Every click

S3

Clicks, videos, interactives

MLlib on Spark

Customer segments



S3

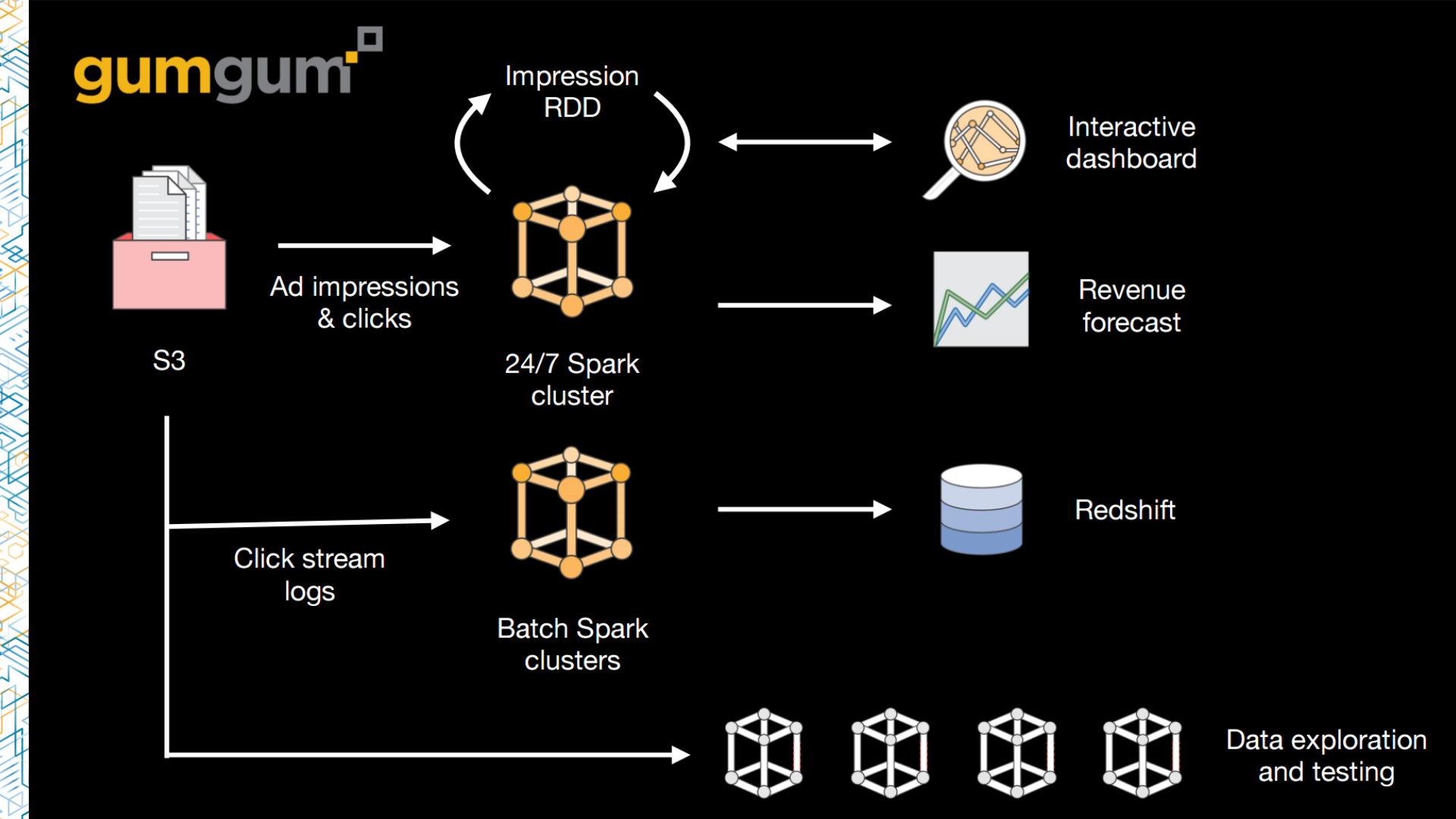
Ad sales

Map user to ads

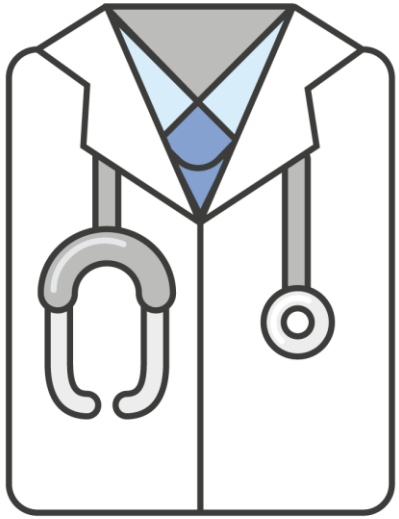
Forecasting

Clicks, videos, interactives

Recommendation Spark cluster


gumgum
Impression RDD
Interactive dashboard
Ad impressions & clicks
S3
24/7 Spark cluster
Revenue forecast
Click stream logs
Batch Spark clusters
Redshift
Data exploration and testing

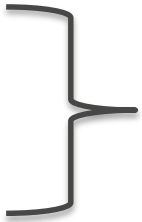
Best Practices for Spark on Amazon EMR



- Using correct instance
- Understanding Executors
  - Sizing your executors
  - Dynamic allocation on YARN
- Understanding storage layers
  - File formats and compression
- Boost your performance
  - Data serialization
  - Avoiding shuffle
  - Managing partitions
  - RDD Persistence
- Using Zeppelin notebook

What does Spark need?

- Memory – lots of it!!
- Network
- CPU
- Horizontal Scaling



Instance

Workflow	Resource
Machine learning	CPU
ETL	I/O

Choose your instance types

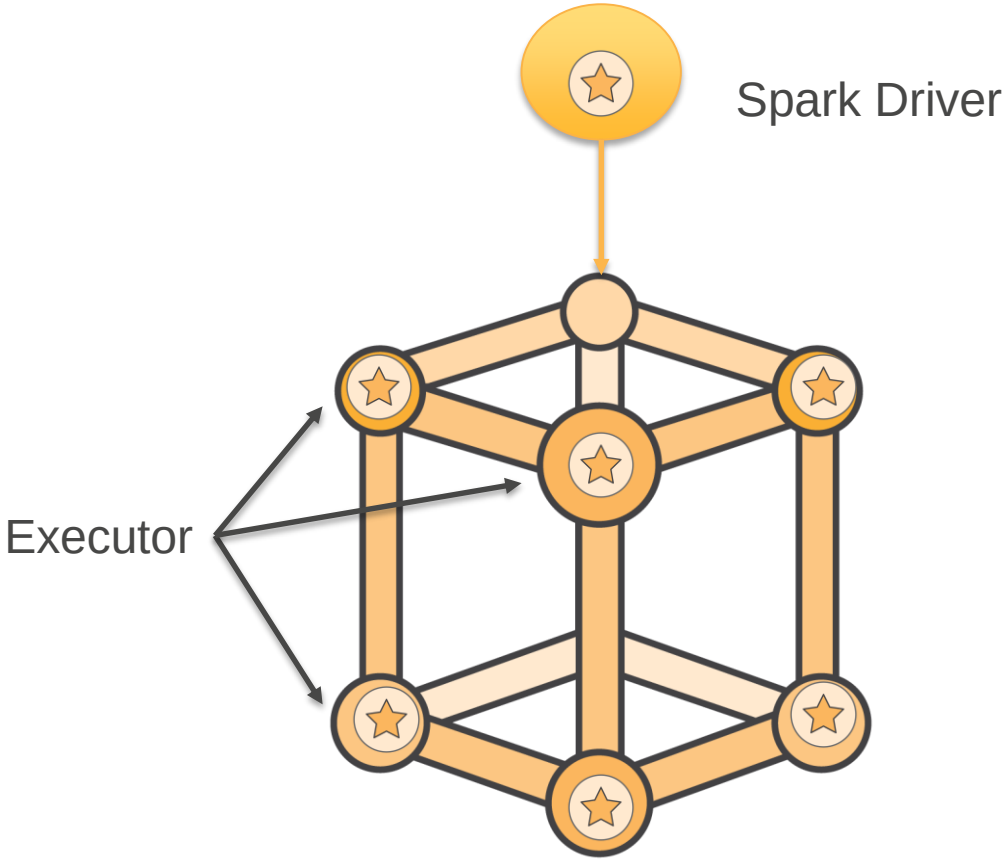
Try different configurations to find your optimal architecture.

General	CPU	Memory	Disk/IO
m1 family m3 family	c1 family c3 family cc1.4xlarge cc2.8xlarge	m2 family r3 family cr1.8xlarge	d2 family i2 family
Batch process	**Machine learning**	**Interactive Analysis**	**Large HDFS**

- Using correct instance
- Understanding Executors
  - Sizing your executors
  - Dynamic allocation on YARN
- Understanding storage layers
  - File formats and compression
  - Caching tables
- Boost your performance
  - Data serialization
  - Avoiding shuffle
  - Managing partitions
  - RDD Persistence

How Spark executes

- Spark Driver
- Executor



Spark Executor on YARN

This is where all the action happens

- How to select number of executors?
- How many cores per executor?
- Example

Sample Amazon EMR cluster

Model	vCPU	Mem (GB)	SSD Storage (GB)	Networking
r3.large	2	15.25	1 x 32	Moderate
r3.xlarge	4	30.5	1 x 80	Moderate
r3.2xlarge	8	61	1 x 160	High
r3.4xlarge	**16**	**122**	**1 x 320**	**High**
r3.8xlarge	32	244	2 x 320	10 Gigabit

Create Amazon EMR cluster


```
$ aws emr create-cluster --name ”Spark cluster” \
   --release-label emr-4.1.0 \
   --applications Name=Hive Name=Spark \
   --use-default-roles \
  --ec2-attributes KeyName=myKey --instance-type r3.4xlarge \
  --instance-count 6 \
  --no-auto-terminate
```

Inside Spark Executor on YARN

Selecting number of executor cores:

- Leave 1 core for OS and other activities
- 4-5 cores per executor gives a good performance
    - Each executor can run up to 4-5 tasks
    - i.e. 4-5 threads for read/write operations to HDFS

Inside Spark Executor on YARN

Selecting number of executor cores:

--num-executors or spark.executor.instances

- $Number\ of\ executors\ per\ node = \frac{(Number\ of\ cores\ on\ node - 1\ \ for\ OS\ )}{Number\ of\ task\ per\ executor}$
- $\frac{16-1}{5} = 3\ executors\ per\ node$

Model	vCPU	Mem (GB)	SSD Storage (GB)	Networking
r3.4xlarge	16	122	1 x 320	High

Inside Spark Executor on YARN

Selecting number of executor cores:

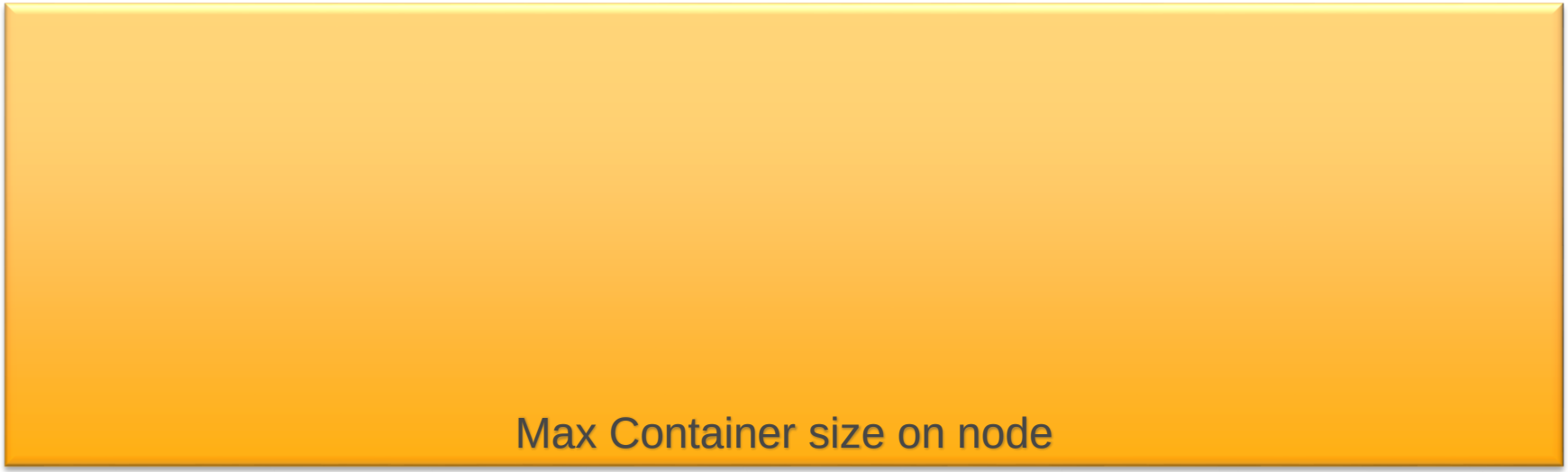
--num-executors or spark.executor.instances

- $\frac{16-1}{5} = 3\ executors\ per\ node$
- *6 instances*

- $num\ of\ executors = (3 * 6) - 1 = \mathbf{17}$

Inside Spark Executor on YARN

YARN Container → Controls the max sum of memory used by the container
yarn.nodemanager.resource.memory-mb

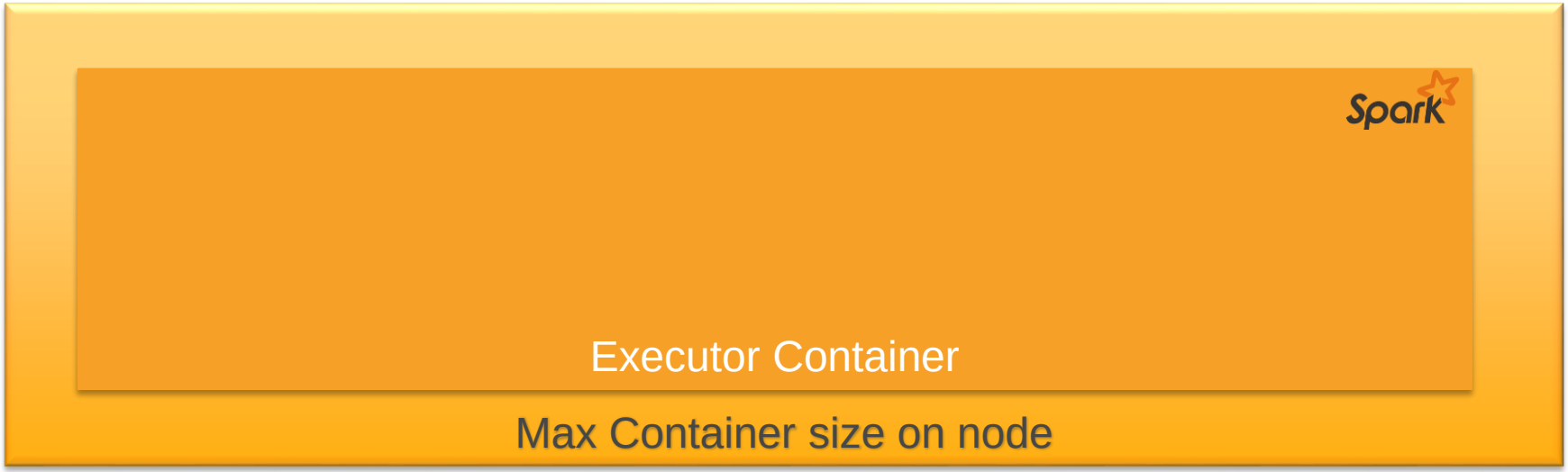


Config File: yarn-site.xml

Default: 116 G

Inside Spark Executor on YARN

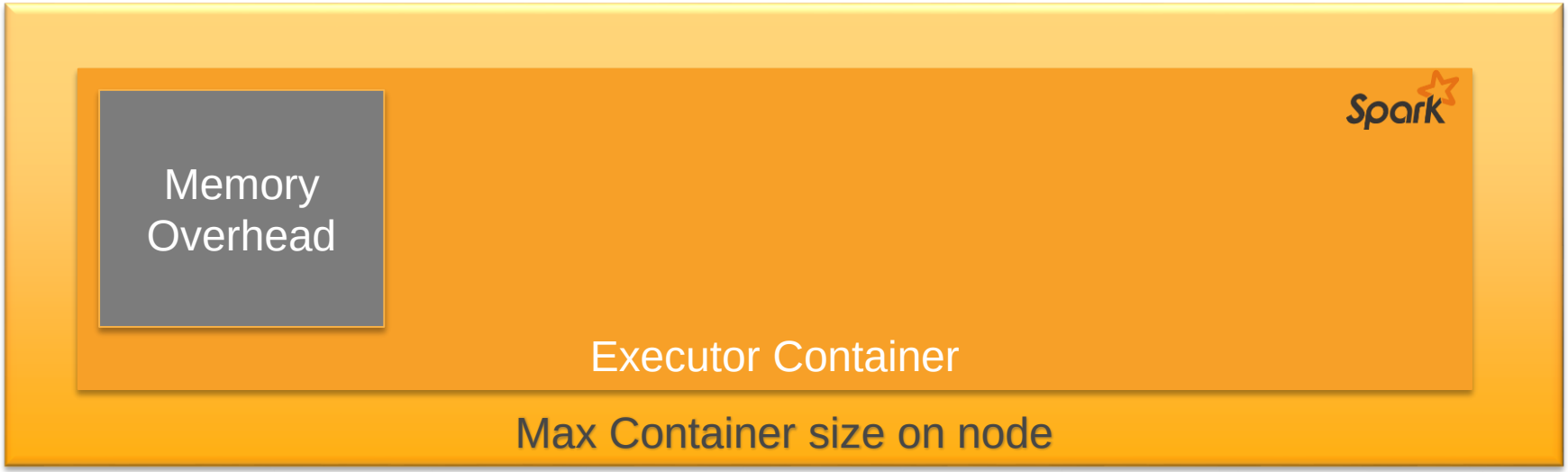
Executor space → Where Spark executor Runs



Inside Spark Executor on YARN

Executor Memory Overhead - Off heap memory (VM overheads, interned strings etc.)

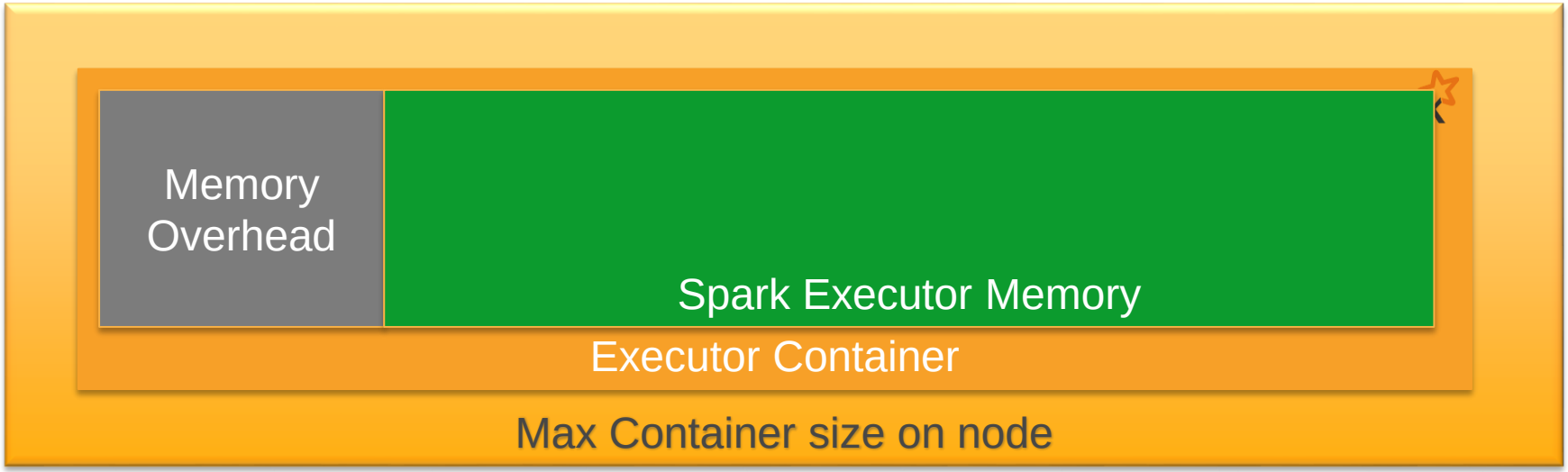
$$spark.yarn.executor.memoryOverhead = executorMemory * 0.10$$



Config File: spark-default.conf

Inside Spark Executor on YARN

Spark executor memory - Amount of memory to use per executor process
spark.executor.memory

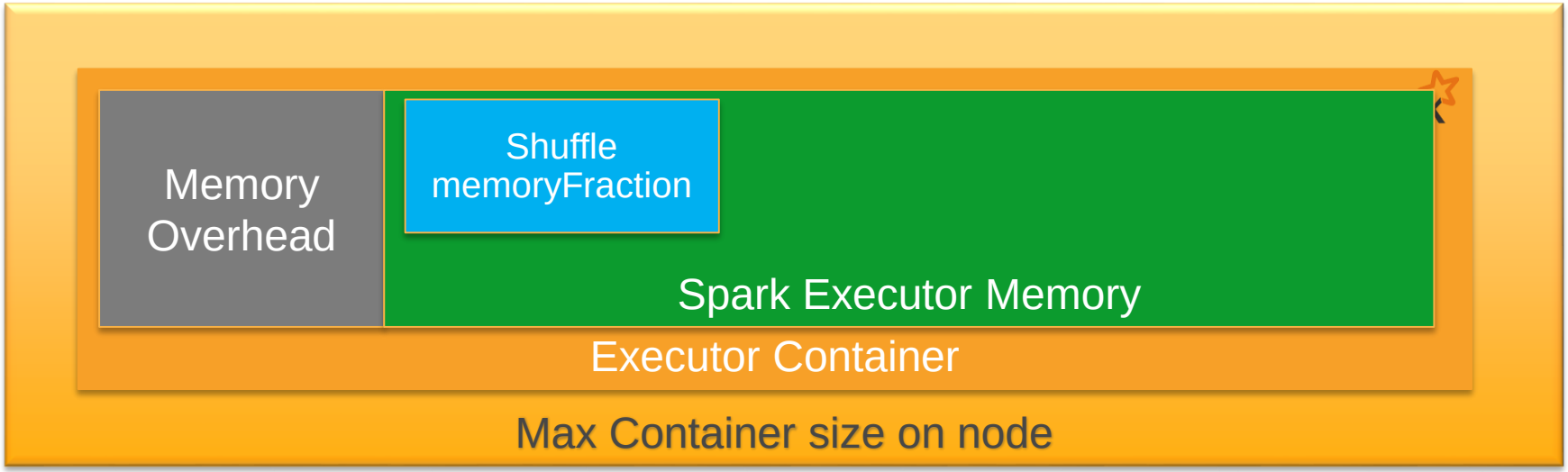


Config File: spark-default.conf

Inside Spark Executor on YARN

Shuffle Memory Fraction- Fraction of Java heap to use for aggregation and cogroups during shuffles

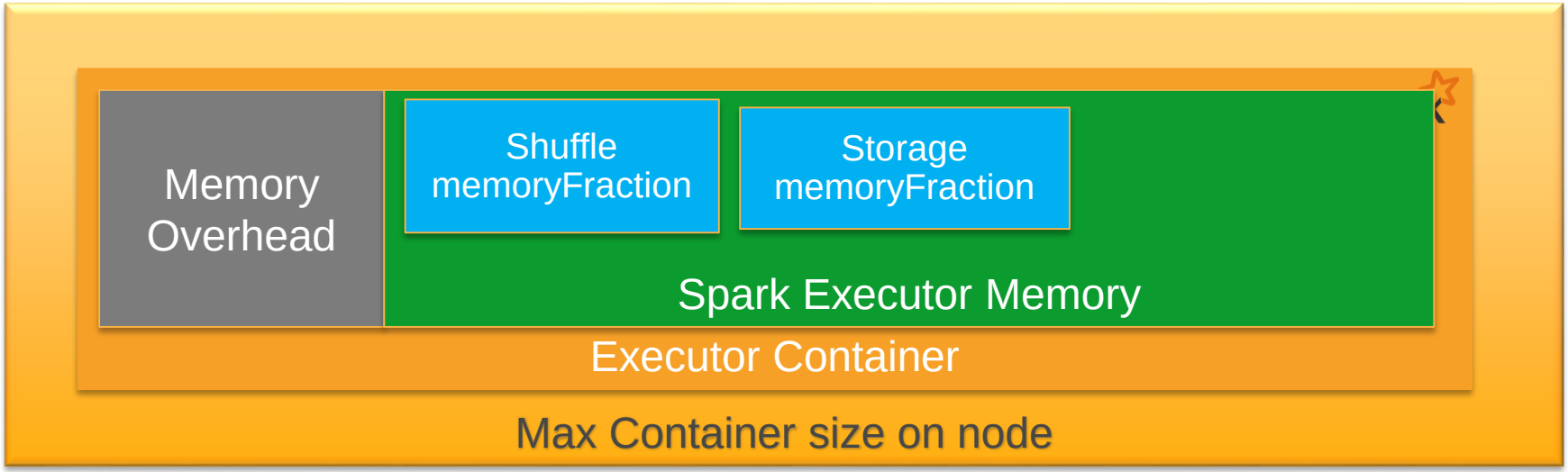
spark.shuffle.memoryFraction



Default: 0.2

Inside Spark Executor on YARN

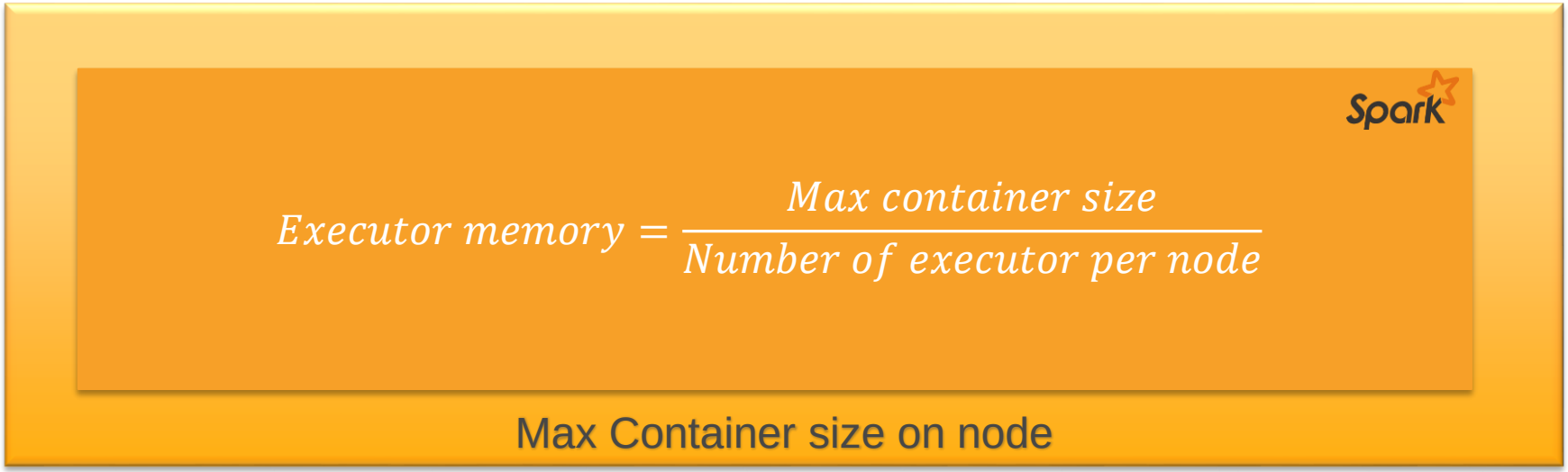
Storage storage Fraction - Fraction of Java heap to use for Spark's memory cache
spark.storage.memoryFraction



Default: 0.6

Inside Spark Executor on YARN

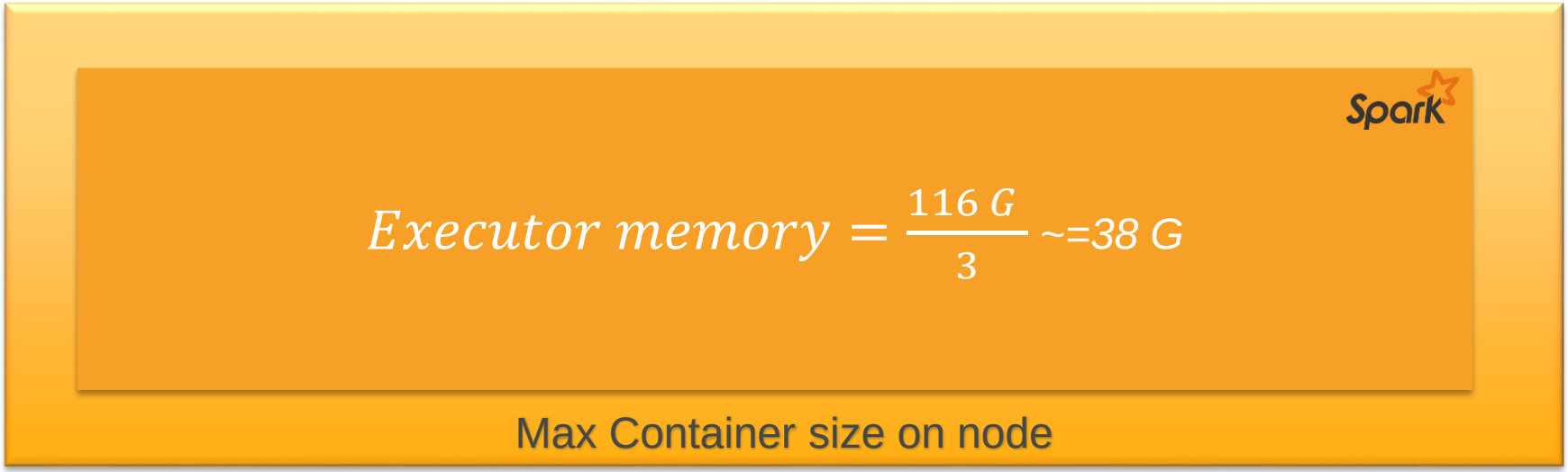
--executor-memory or spark.executor.memory



Config File: spark-default.conf

Inside Spark Executor on YARN

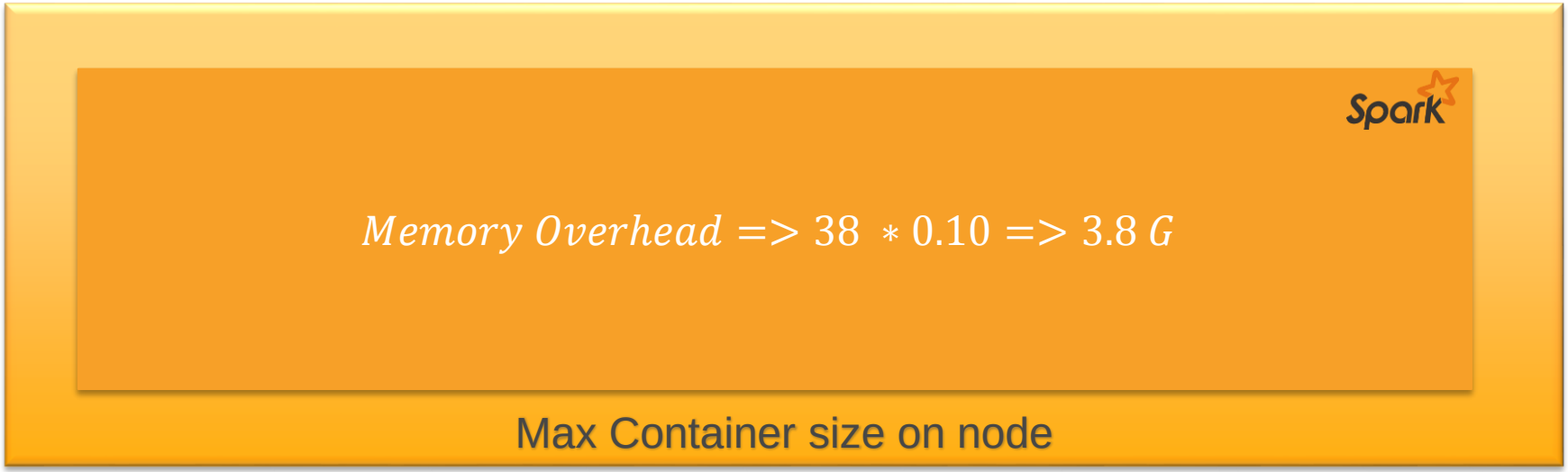
--executor-memory or spark.executor.memory



Config File: spark-default.conf

Inside Spark Executor on YARN

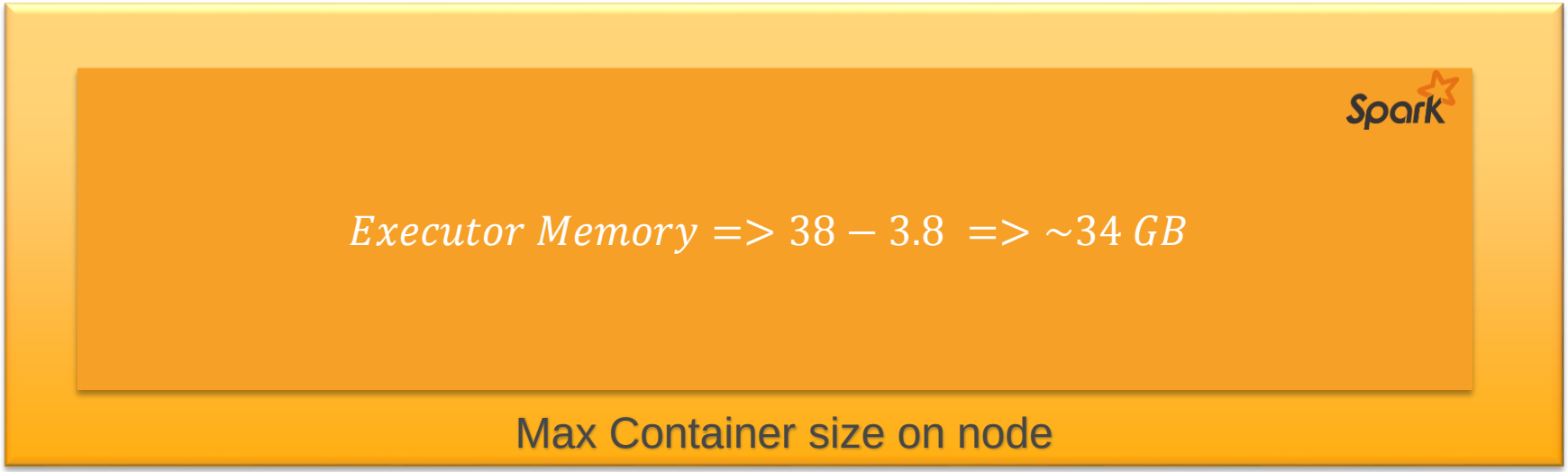
--executor-memory or spark.executor.memory



Config File: spark-default.conf

Inside Spark Executor on YARN

--executor-memory or spark.executor.memory



Config File: spark-default.conf

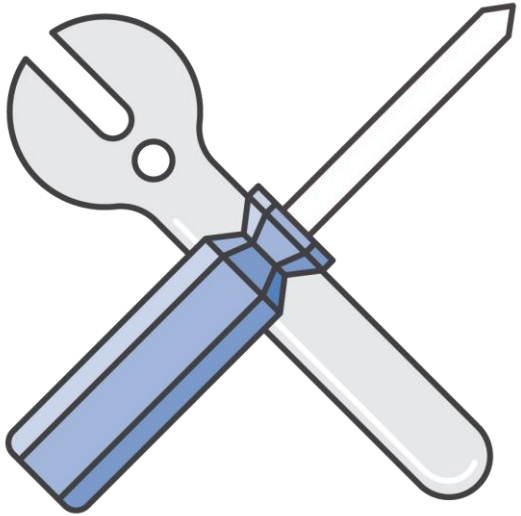
Inside Spark Executor on YARN

Optimal setting:

--num-executors 17

--executor-cores 5

--executor-memory 34G



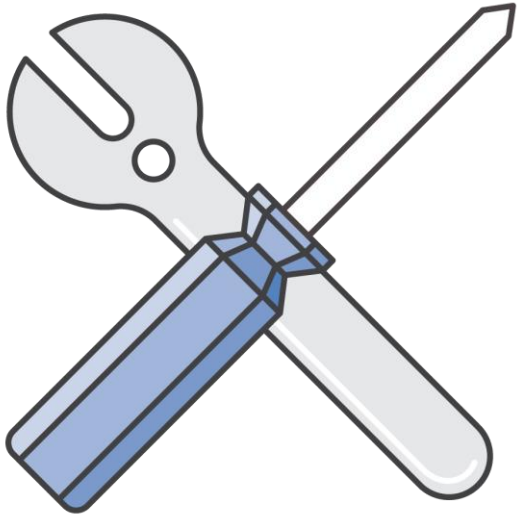
Inside Spark Executor on YARN

Optimal setting:

--num-executors 17

--executor-cores 5

--executor-memory 34G



- Using correct instance
- Understanding Executors
  - Sizing your executors
  - Dynamic allocation on YARN
- Understanding storage layers
  - File formats and compression
- Boost your performance
  - Data serialization
  - Avoiding shuffle
  - Managing partitions
  - RDD Persistence
- Using Zeppelin Notebook

Dynamic Allocation on YARN

… allows your Spark applications to scale up based on demand and scale down when not required.

Remove Idle executors,

Request more on demand

Dynamic Allocation on YARN

Scaling up on executors

- Request when you want the job to complete faster
- Idle resources on cluster
- Exponential increase in executors over time

Dynamic allocation setup

Property	Value
Spark.dynamicAllocation.enabled	true
Spark.shuffle.service.enabled	true
spark.dynamicAllocation.minExecutors	*5*
spark.dynamicAllocation.maxExecutors	*17*
spark.dynamicAllocation.initalExecutors	*0*
sparkdynamicAllocation.executorIdleTime	*60s*
spark.dynamicAllocation.schedulerBacklogTimeout	*5s*
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout	*5s*

Optional

- Using correct instance
- Understanding Executors
  - Sizing your executors
  - Dynamic allocation on YARN
- Understanding storage layers
  - File formats and compression
- Boost your performance
  - Data serialization
  - Avoiding shuffle
  - Managing partitions
  - RDD Persistence
- Using Zeppelin notebook

Compressions

- Always compress data files on Amazon S3
- Reduces storage cost
- Reduces bandwidth between Amazon S3 and Amazon EMR
- Speeds up your job

Compressions

Compression types:

- Some are fast BUT offer less space reduction
- Some are space efficient BUT slower
- Some are split able and some are not

Algorithm	% Space Remaining	Encoding Speed	Decoding Speed
GZIP	13%	21MB/s	118MB/s
LZO	20%	135MB/s	410MB/s
Snappy	22%	172MB/s	409MB/s

Compressions

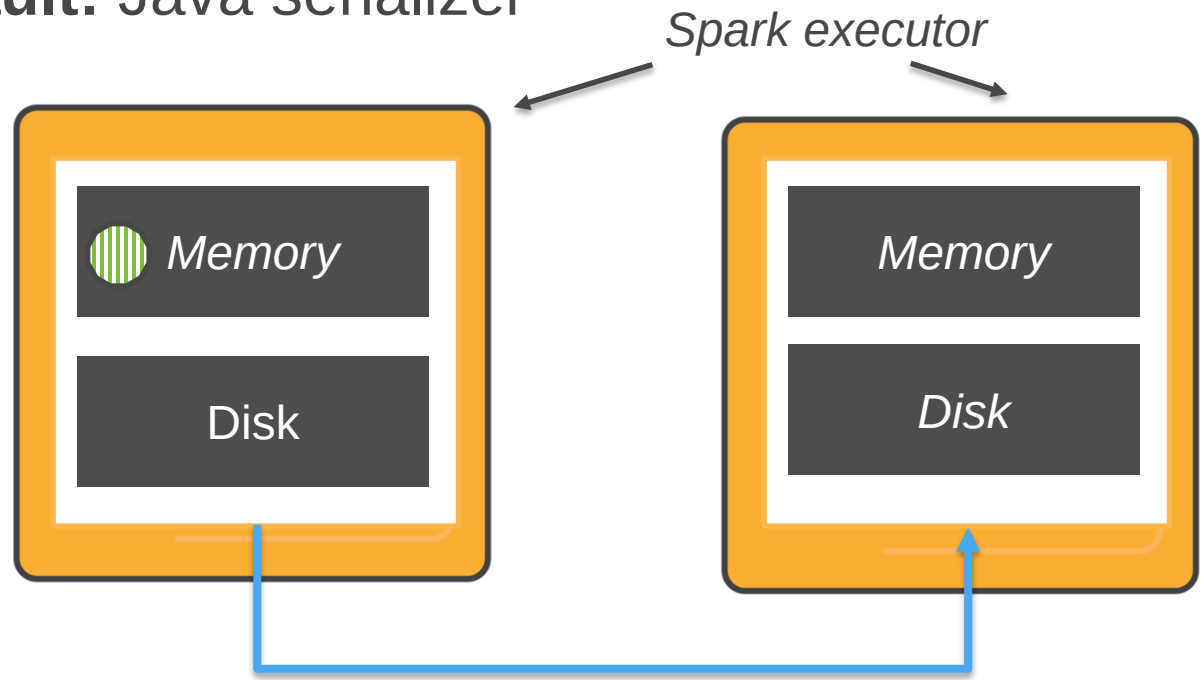
- If you are time-sensitive, faster compressions are a better choice
- If you have large amount of data, use space-efficient compressions
- If you don’t care, pick GZIP

- Using correct instance
- Understanding Executors
  - Sizing your executors
  - Dynamic allocation on YARN
- Understanding storage layers
  - File formats and compression
- Boost your performance
  - Data serialization
  - Avoiding shuffle
  - Managing partitions
  - RDD Persistence
- Using Zeppelin notebook

Data Serialization

- Data is serialized when cached or shuffled

  Default: Java serializer



Data Serialization

- Data is serialized when cached or shuffled

  Default: Java serializer
- **Kyro serialization (10x faster than Java serialization)**
  - Does not support all Serializable types
  - Register the class in advance

Usage: Set in SparkConf

conf.set("spark.serializer","org.apache.spark.serializer.KryoSerializer")

Spark doesn’t like to Shuffle

- Shuffling is expensive
  - Disk I/O
  - Data Serialization
  - Network I/O
  - Spill to disk
  - Increased Garbage collection
- Use aggregateByKey() instead of your own aggregator

  Usage:

  myRDD.aggregateByKey(0)((k,v) => v.toInt+k, (v,k) => k+v).collect
- Apply filter earlier on data

Parallelism & Partitions

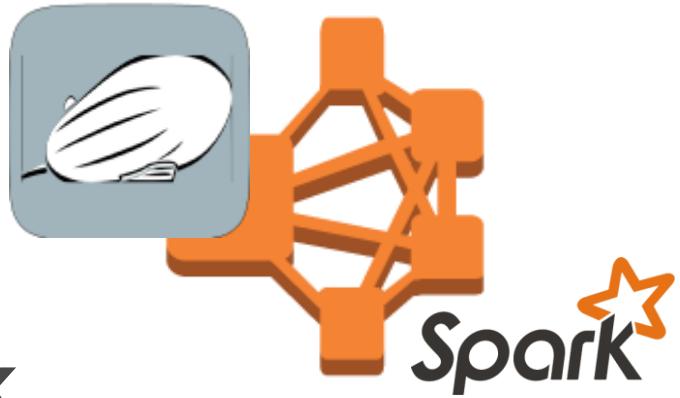
spark.default.parallelism

- **getNumPartitions()**
- If you have >10K tasks, then its good to coalesce
- If you are not using all the slots on cluster, repartition can increase parallelism
- 2-3 tasks per CPU core in your cluster

Config File: spark-default.conf

RDD Persistence

- Caching or persisting dataset in memory
- Methods
  - cache()
  - persist()
- Small RDD → MEMORY_ONLY
- Big RDD → MEMORY_ONLY_SER (CPU intensive)
- Don’t spill to disk
- Use replicated storage for faster recovery



Zeppelin and Spark on Amazon EMR

Remember to complete your evaluations!

Thank you!