



Best Practices for Using Apache Spark on AWS

Jonathan Fritz, Amazon EMR
Senior Product Manager
July 13, 2016



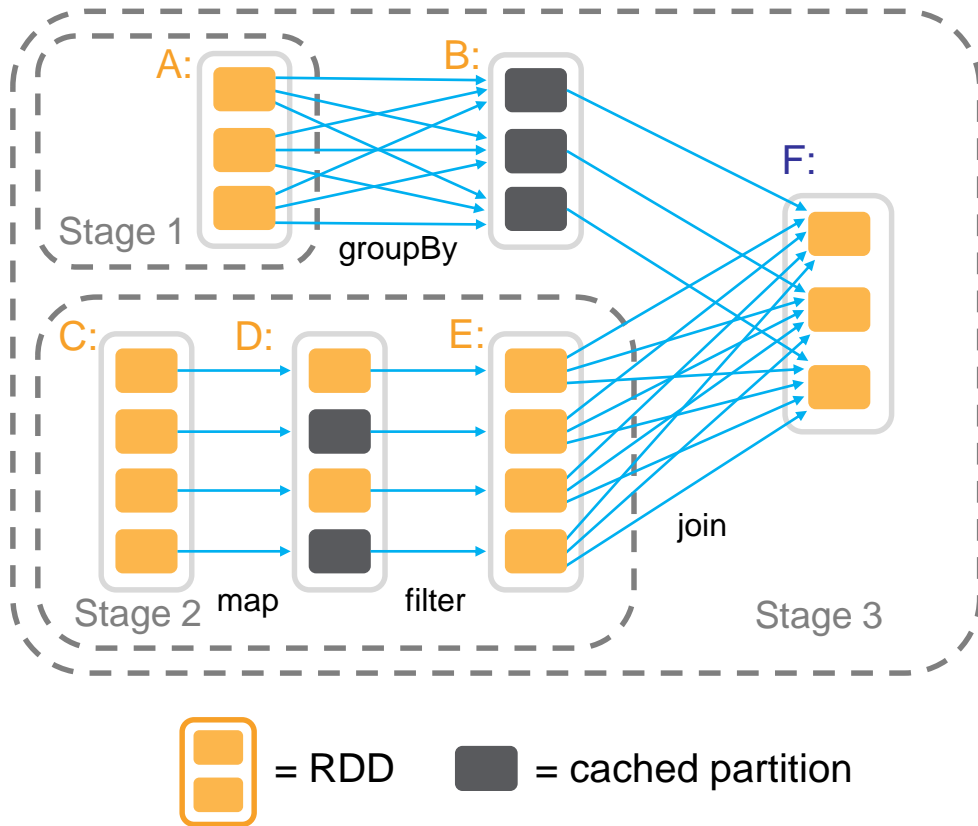
Agenda

- Why Apache Spark?
- Deploying Spark with Amazon EMR
- EMRFS and connectivity to AWS data stores
- Spark on YARN and DataFrames
- Spark security overview

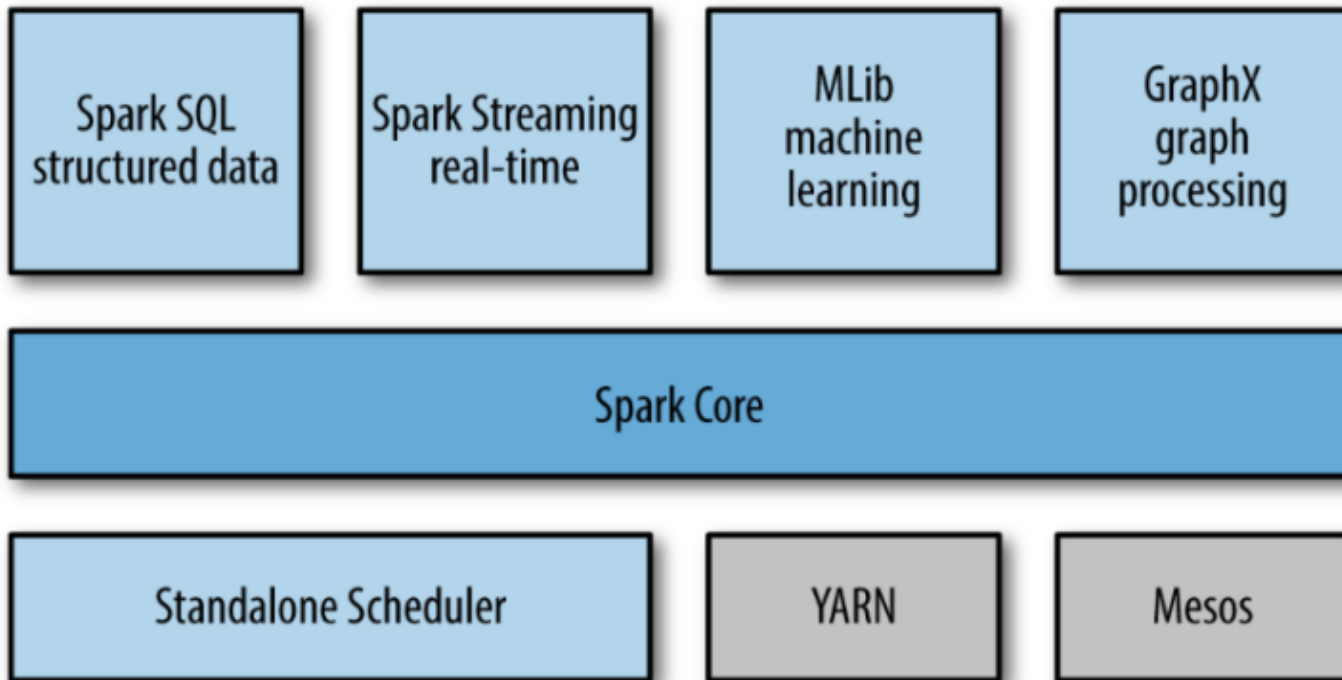


Spark moves at interactive speed

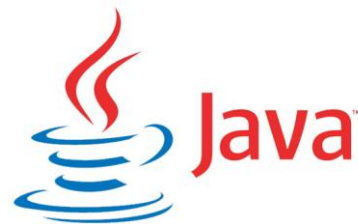
- Massively parallel
- Uses DAGs instead of map-reduce for execution
- Minimizes I/O by storing data in DataFrames in memory
- Partitioning-aware to avoid network-intensive shuffle



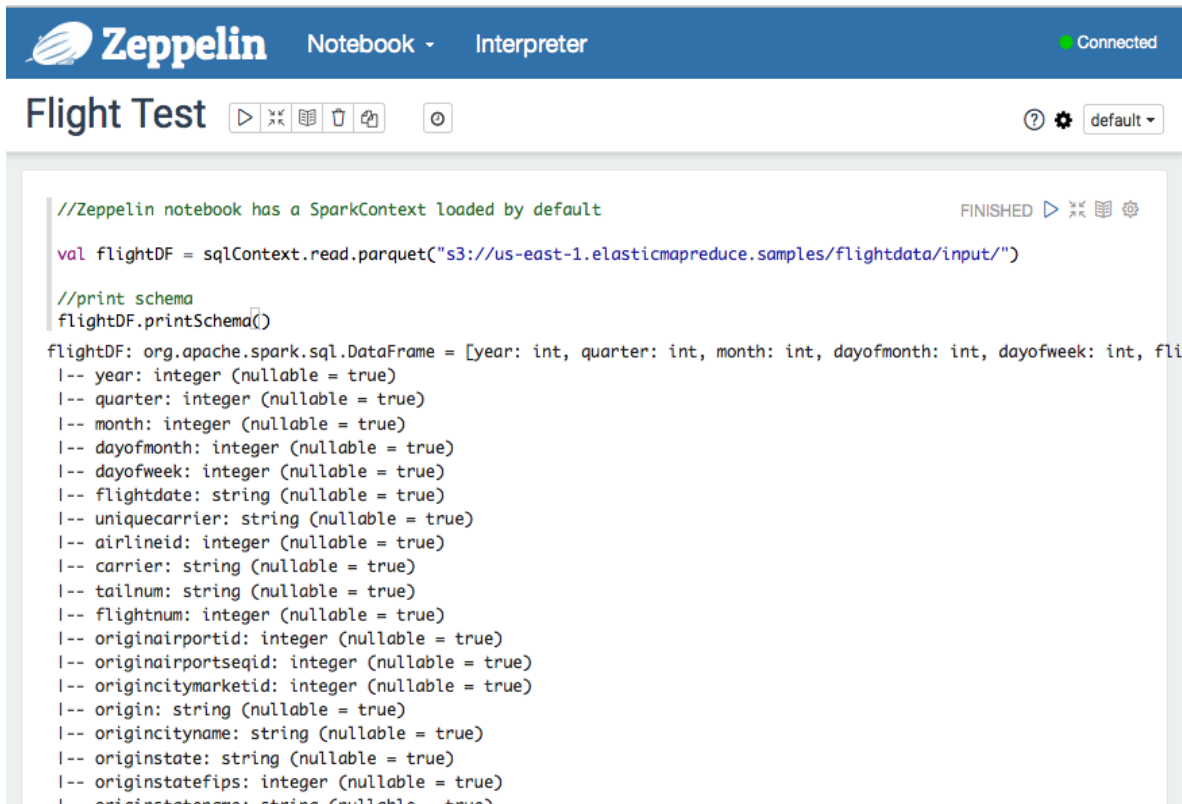
Spark components to match your use case



Spark speaks your language



Use DataFrames to easily interact with data



The screenshot shows the Zeppelin Notebook interface. The top bar is blue with the Zeppelin logo, 'Notebook' and 'Interpreter' tabs, and a 'Connected' status indicator. Below the bar, the notebook title 'Flight Test' is displayed with various icons for execution and management. The main area contains a code block with the following content:

```
//Zeppelin notebook has a SparkContext loaded by default
val flightDF = sqlContext.read.parquet("s3://us-east-1.elasticmapreduce.samples/flightdata/input/")
//print schema
flightDF.printSchema()

flightDF: org.apache.spark.sql.DataFrame = [year: int, quarter: int, month: int, dayofmonth: int, dayofweek: int, fl...
```

The output of the code is a detailed schema for the flight data, listing various fields and their data types and nullability.

- Distributed collection of data organized in columns
- Abstraction for selecting, filtering, aggregating, and plotting structured data
- More optimized for query execution than RDDs from Catalyst query planner
- Datasets introduced in Spark 1.6

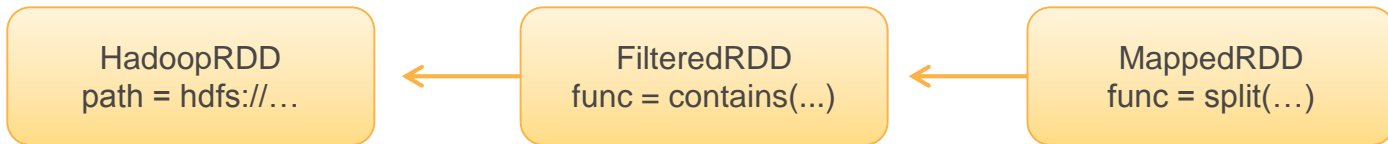
Datasets and DataFrames

- Datasets are an extension of the DataFrames API
(preview in Spark 1.6)
- Object-oriented operations (similar to RDD API)
- Utilizes Catalyst query planner
- Optimized encoders which increase performance and minimize serialization/deserialization overhead
- Compile-time type safety for more robust applications

DataFrames/Datasets/RDDs and fault tolerance

Track the transformations used to build them (their lineage)
to recompute lost data

E.g.: `messages = textFile(...).filter(lambda s: s.contains("ERROR"))
.map(lambda s: s.split('\t')[2])`



Easily create DataFrames from many formats



Parquet



Additional libraries for Spark SQL Data Sources
at spark-packages.org

Load data with the Spark SQL Data Sources API



Amazon Redshift

Additional libraries at spark-packages.org

Use DataFrames for machine learning

```
// Prepare training documents from a list of (id, text, label) tuples
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: tokenizer, hashingTF, and lr.
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")
val hashingTF = new HashingTF()
  .setNumFeatures(1000)
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.01)
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF, lr))

// Fit the pipeline to training documents.
val model = pipeline.fit(training)
```

- Spark ML libraries (replacing MLlib) use DataFrame API as input/output for models instead of RDDs
- Create ML pipelines with a variety of distributed algorithms
- Pipeline persistence to save and load models and full pipelines to Amazon S3

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
- Checkpoint streaming jobs for disaster recovery

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 (SparkR support in Zeppelin 0.6.0—coming soon in EMR)
- Comparable performance to Python and Scala DataFrames

Spark SQL

- Seamlessly mix SQL with Spark programs
- Uniform data access
- Can interact with tables in Hive metastore
- Hive compatibility—run Hive queries without modifications using HiveContext
- Connect through JDBC/ODBC using the Spark Thrift server



Spark 2.0

Coming soon on Amazon EMR

Spark 2.0—API updates

- Increased SQL support with new ANSI SQL parser
- Unifying the DataFrame and Dataset API
- Combining SparkContext and HiveContext
- Additional distributed algorithms in SparkR, including K-Means, Generalized Linear Models, and Naive Bayes
- ML pipeline persistence is now supported across all languages

Spark 2.0—Performance enhancements

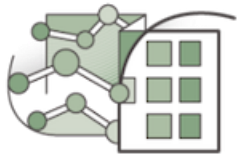
- Second generation Tungsten engine
- Whole-stage code generation to create optimized bytecode at run time
- Improvements to Catalyst optimizer for query performance
- New vectorized Parquet decoder to increase throughput

Spark 2.0—Structured streaming

- New approach to stream data processing
- Structured Streaming API is an extension to the DataFrame/Dataset API (no more DStream)
- Better merges processing on static and streaming datasets
- Leverages Spark to use incremental execution if data is a dynamic stream, and begins to abstract the nature of the data from the developer

Creating Spark clusters with 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, Apache
Oozie, or use Zeppelin UI**



**Hourly, reserved, or
Amazon EC2 Spot pricing**



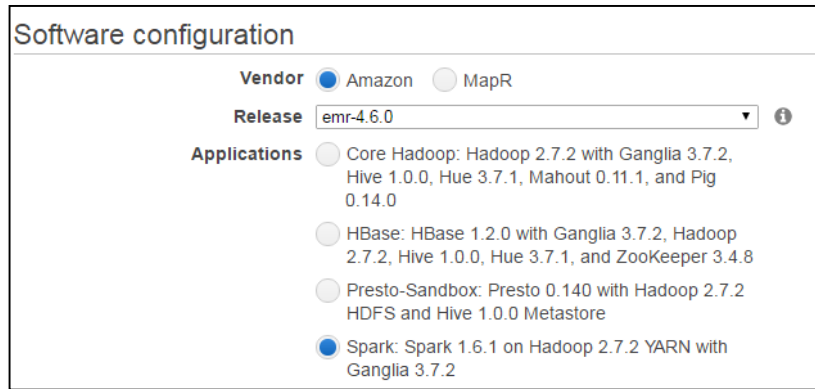
Secured



**Use S3 to decouple
compute and storage**

Create a fully configured cluster with the latest version of Spark in minutes

AWS Management
Console



The screenshot shows the 'Software configuration' section of the AWS Management Console. It includes a 'Vendor' dropdown set to 'Amazon', a 'Release' dropdown set to 'emr-4.6.0', and a list of 'Applications' with radio buttons. The 'Spark' application is selected.

Software configuration

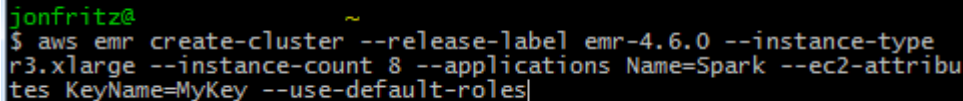
Vendor ☒ Amazon ☐ MapR

Release

Applications

- ☐ Core Hadoop: Hadoop 2.7.2 with Ganglia 3.7.2, Hive 1.0.0, Hue 3.7.1, Mahout 0.11.1, and Pig 0.14.0
- ☐ HBase: HBase 1.2.0 with Ganglia 3.7.2, Hadoop 2.7.2, Hive 1.0.0, Hue 3.7.1, and ZooKeeper 3.4.8
- ☐ Presto-Sandbox: Presto 0.140 with Hadoop 2.7.2 HDFS and Hive 1.0.0 Metastore
- ☒ Spark: Spark 1.6.1 on Hadoop 2.7.2 YARN with Ganglia 3.7.2

AWS Command Line
Interface (AWS CLI)



```
jonfritz@
$ aws emr create-cluster --release-label emr-4.6.0 --instance-type r3.xlarge --instance-count 8 --applications Name=Spark --ec2-attributes KeyName=MyKey --use-default-roles
```

Or use a **AWS SDK** directly with the **Amazon EMR API**

Choice of multiple instances

General

m1 family
m3 family
m4 family

CPU

c3 family
c4 family

Memory

m2 family
r3 family

Disk/IO

d2 family
i2 family
*(or just add EBS
to another
instance type)*

**Batch
processing**

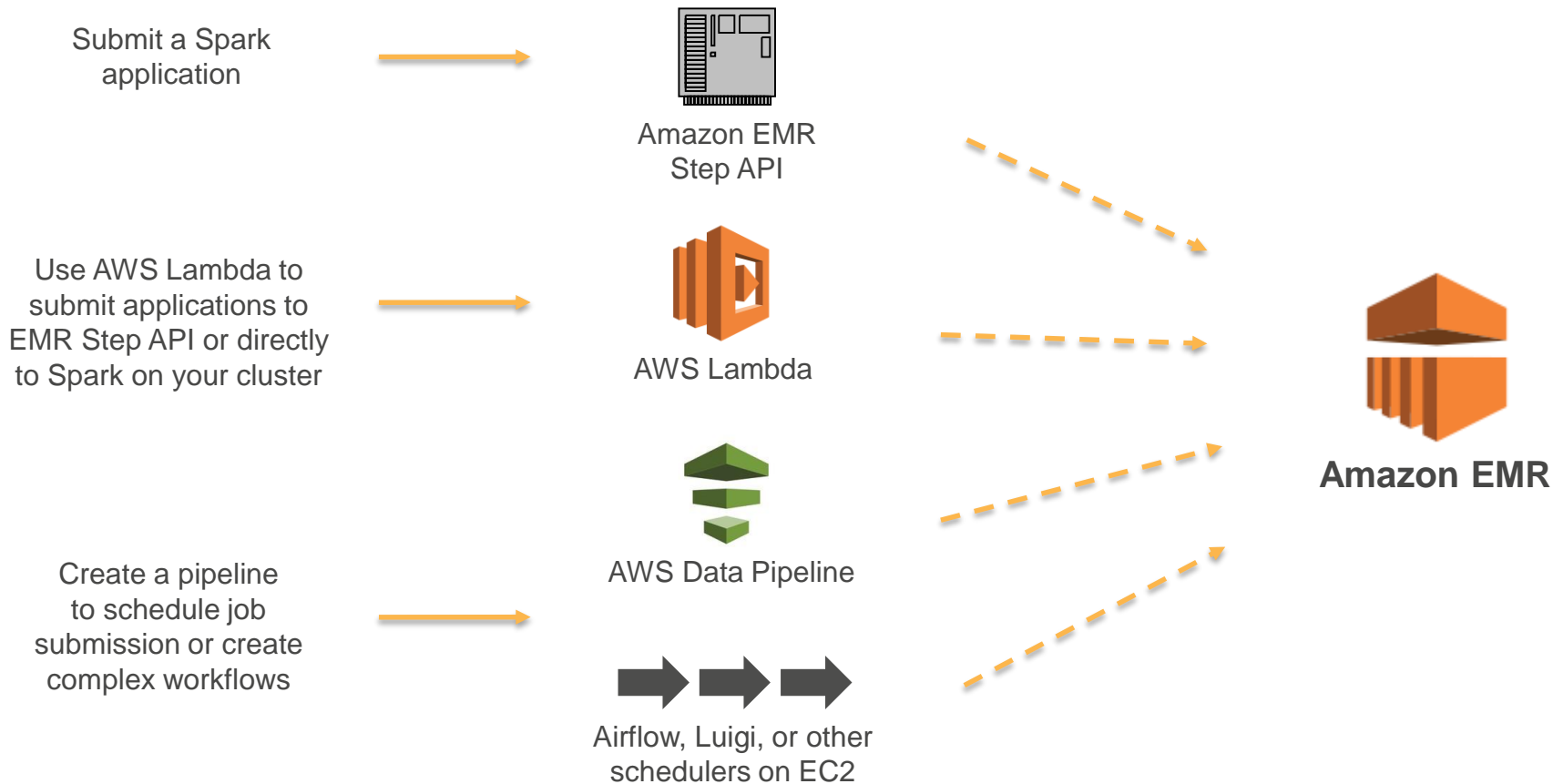
**Machine
learning**

**Cache large
DataFrames**

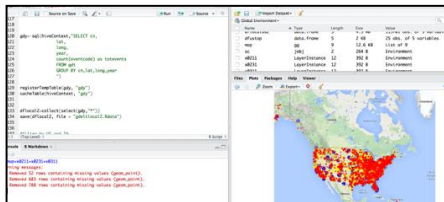
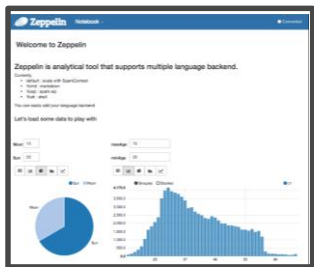
Large HDFS

**Or use EC2 Spot Instances to save up to 90%
on your compute costs.**

Options to submit Spark Jobs—off cluster



Options to submit Spark jobs—on cluster



Web UIs: Zeppelin notebooks,
R Studio, and more!

Use Spark Actions in your Apache Oozie
workflow to create DAGs of Spark jobs.



Connect with ODBC / JDBC
using the Spark Thrift server

Other:

- Use the Spark Job Server for a REST interface and shared DataFrames across jobs
- Use the Spark shell on your cluster

Monitoring and Debugging

- Log pushing to S3
 - Logs produced by driver and executors on each node
 - Can browse through log folders in EMR console
- Spark UI
 - Job performance, task breakdown of jobs, information about cached DataFrames, and more
- Ganglia monitoring
- Amazon CloudWatch metrics in the EMR console

Some of our customers running Spark on EMR



Machine learning &
ad targeting



Web analytics



Ad targeting &
recommendations



App search



Security event
streaming



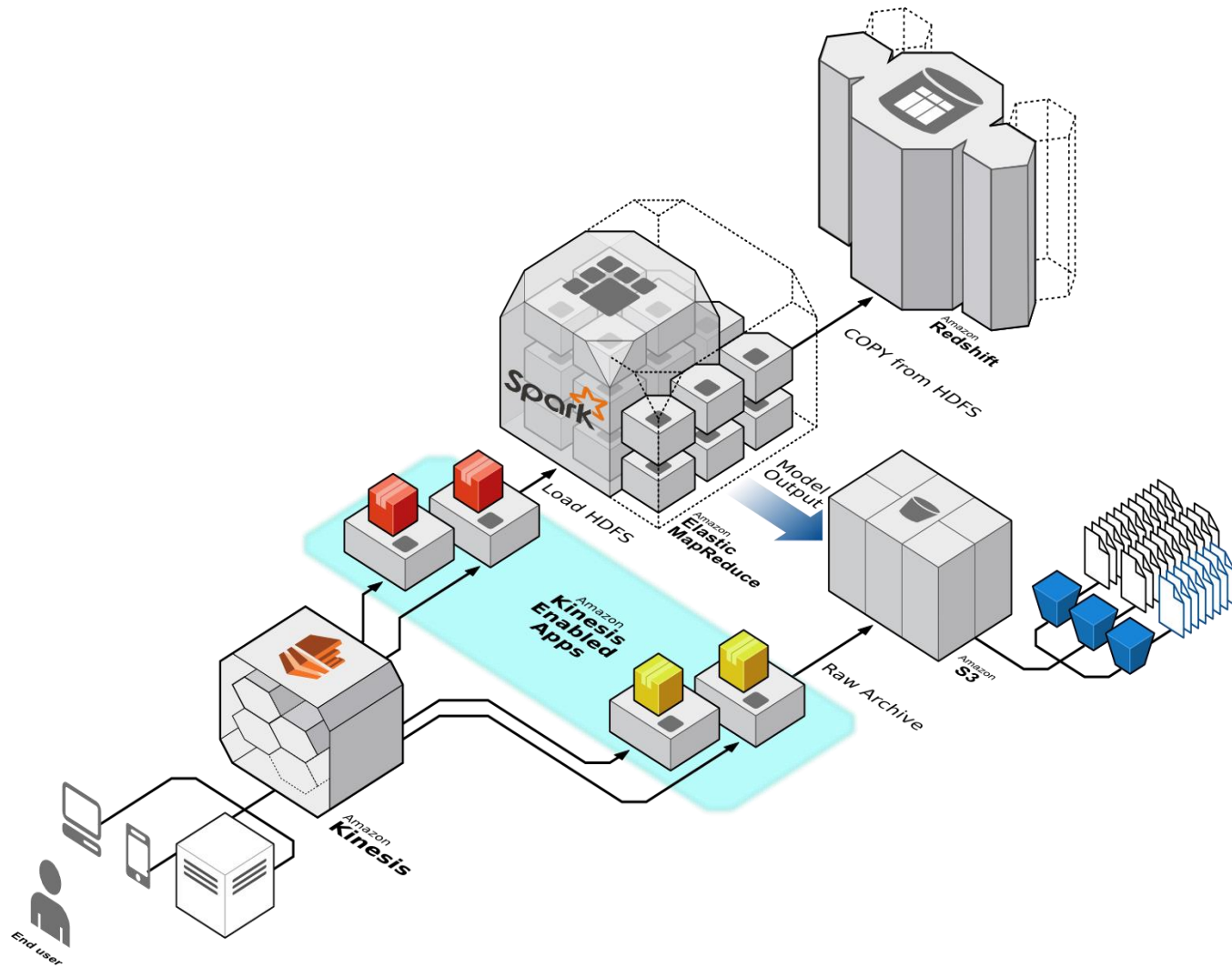
Revenue forecasting



Predictive marketing



Personalization





S3

Ad impressions
& clicks

Impression
RDD



24/7 Spark
cluster



Batch Spark
clusters



Interactive
dashboard

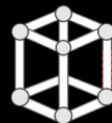
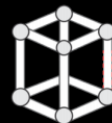
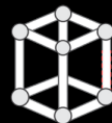
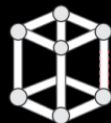
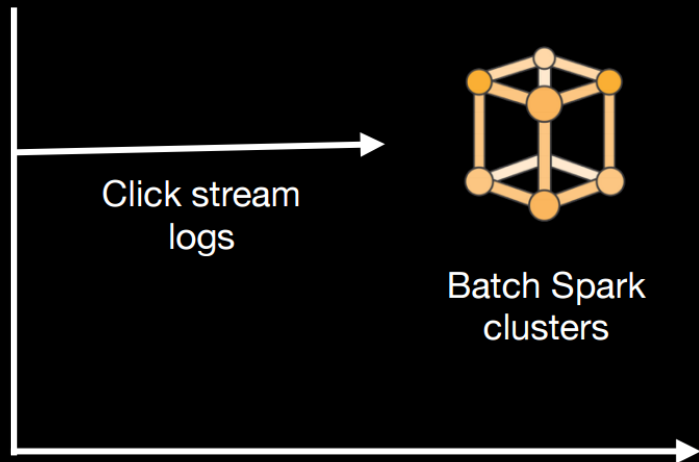


Revenue
forecast

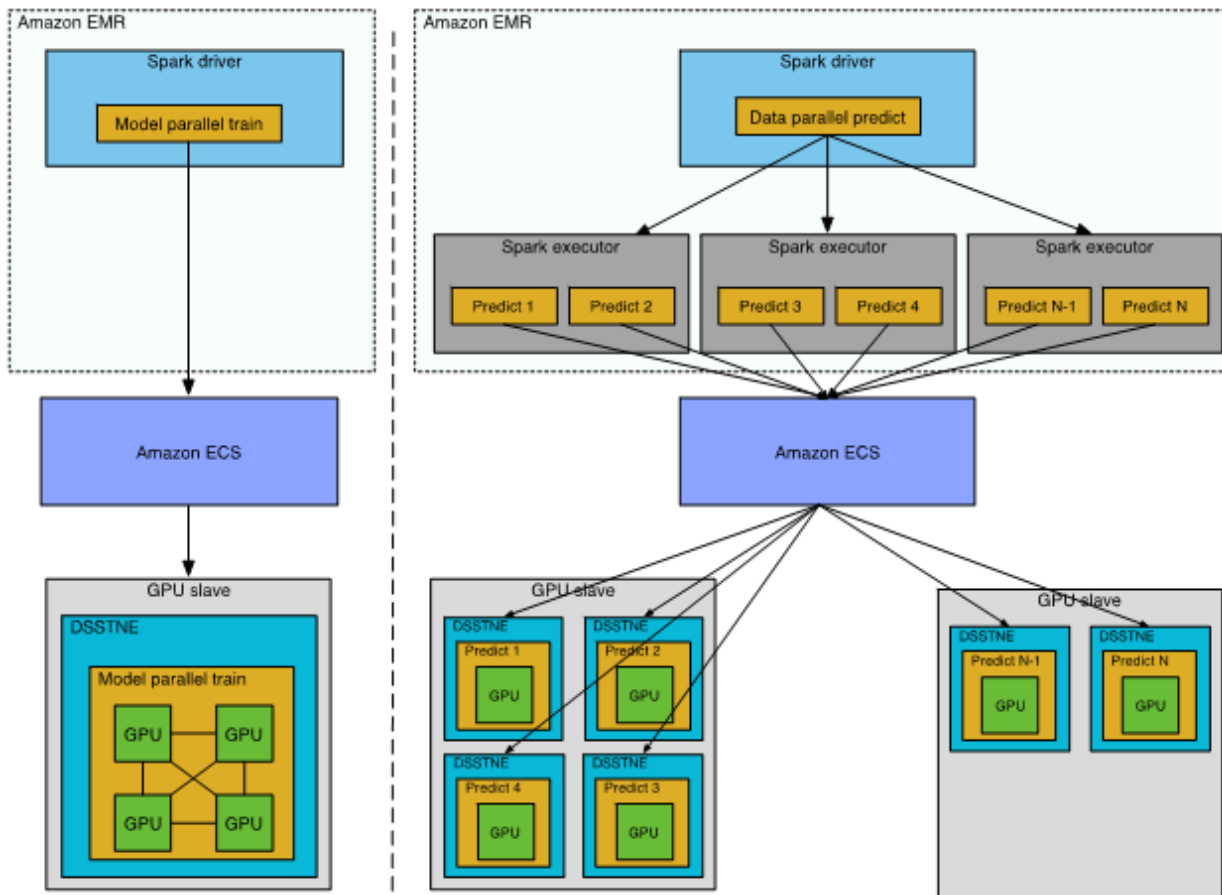


Redshift

Click stream
logs



Data exploration
and testing

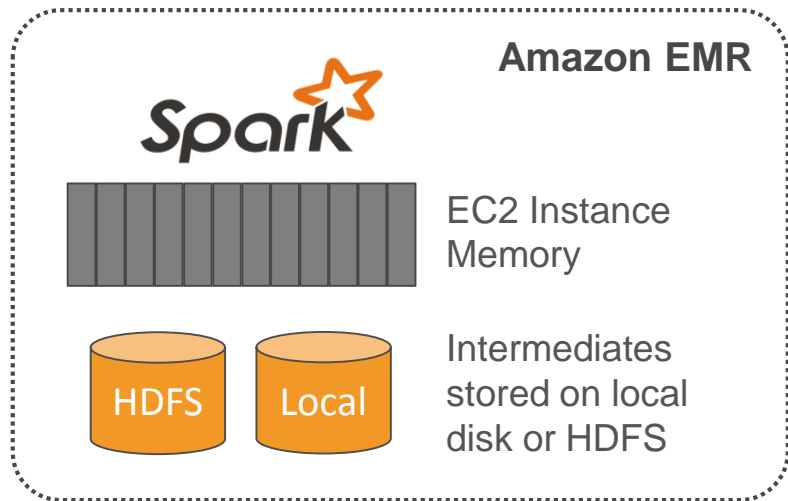


Amazon.com personalization team using Spark + DSSTNE
<http://blogs.aws.amazon.com/bigdata/>

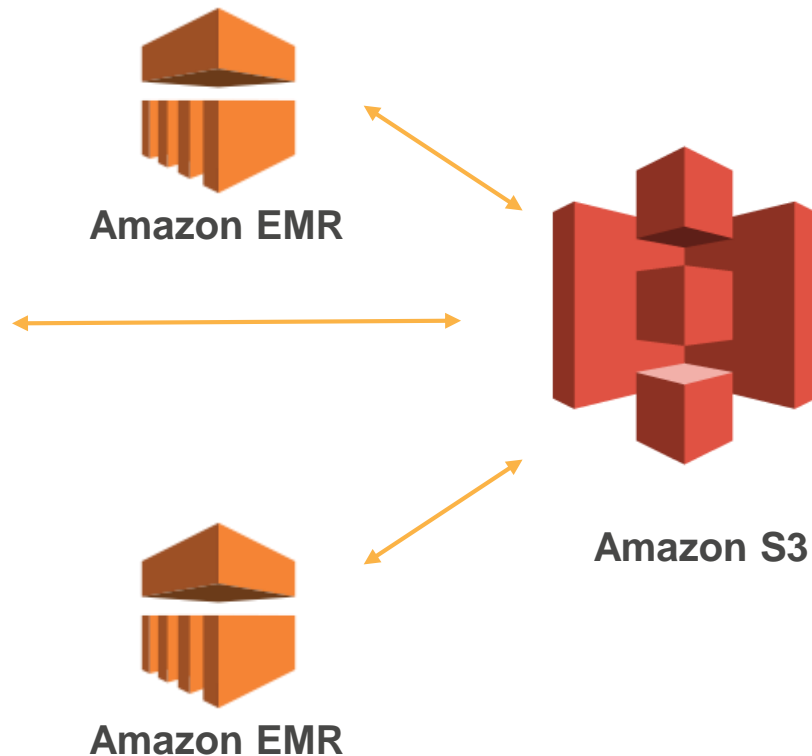
A quick look at Zeppelin and the Spark UI

Using Amazon S3 as persistent storage for Spark

Decouple compute and storage by using S3 as your data layer



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



EMR Filesystem (EMRFS)

- S3 connector for EMR (implements the Hadoop FileSystem interface)
- Improved performance and error handling options
- Transparent to applications—just read/write to “s3://”
- Consistent view feature set for consistent list
- Support for S3 server-side and client-side encryption
- Faster listing using EMRFS metadata

Partitions, compression, and file formats

- Avoid key names in lexicographical order
- Improve throughput and S3 list performance
- Use hashing/random prefixes or reverse the date-time
- Compress data set to minimize bandwidth from S3 to EC2
 - Make sure you use splittable compression or have each file be the optimal size for parallelization on your cluster
- Columnar file formats like Parquet can give increased performance on reads

Use Amazon RDS for an external Hive metastore

Hive metastore with
schema for tables in S3



Amazon RDS



Amazon S3

Set metastore
location in hive-site

Spark 

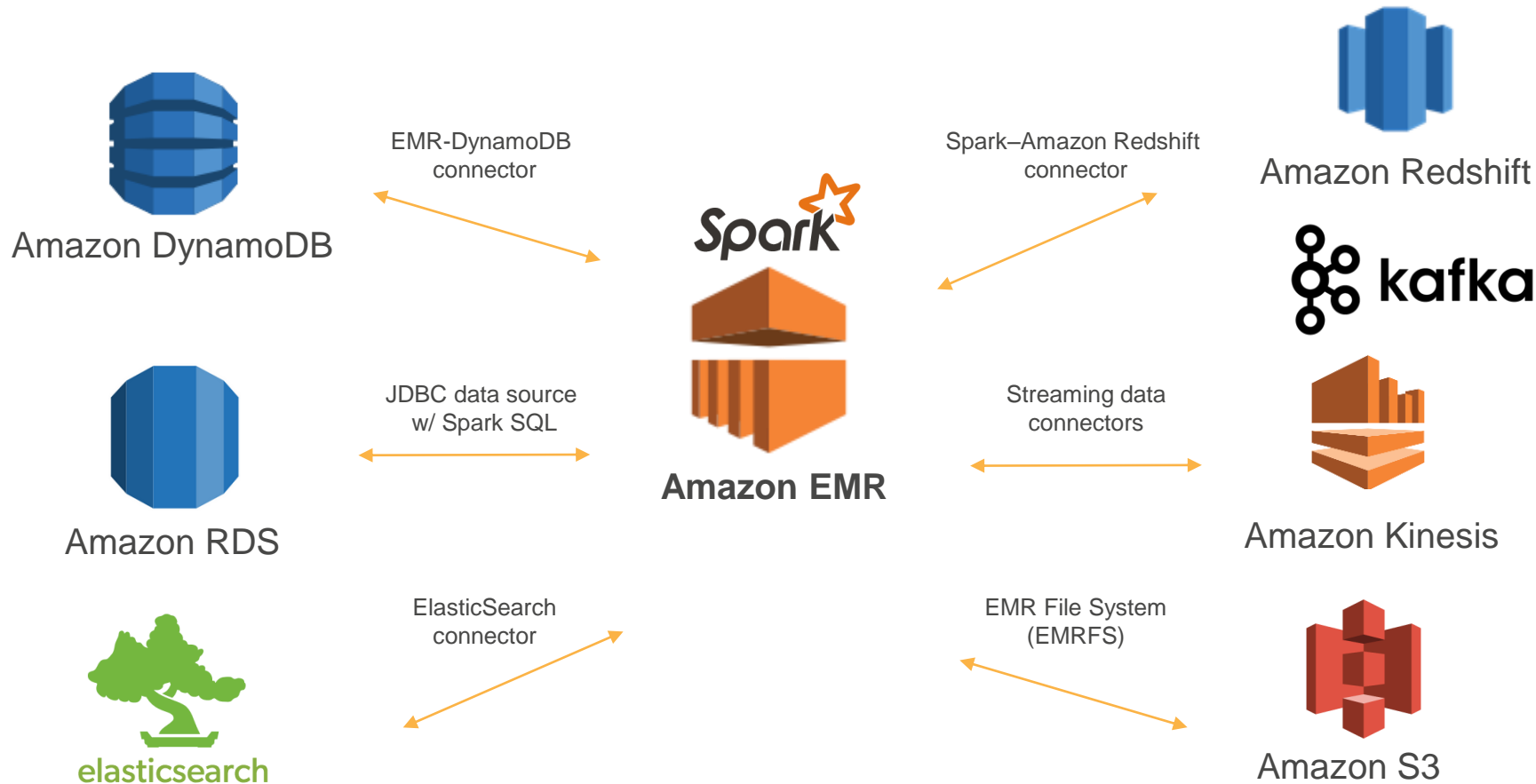


Spark 

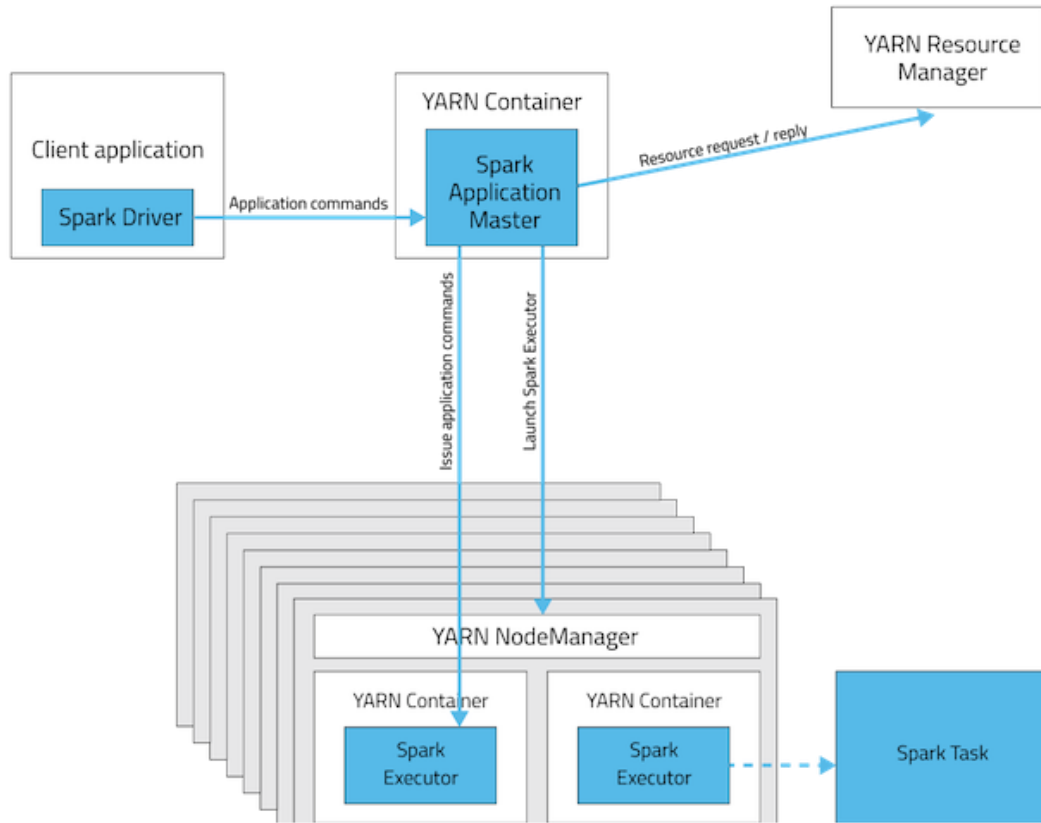


Using Spark with other data stores in AWS

Many storage layers to choose from

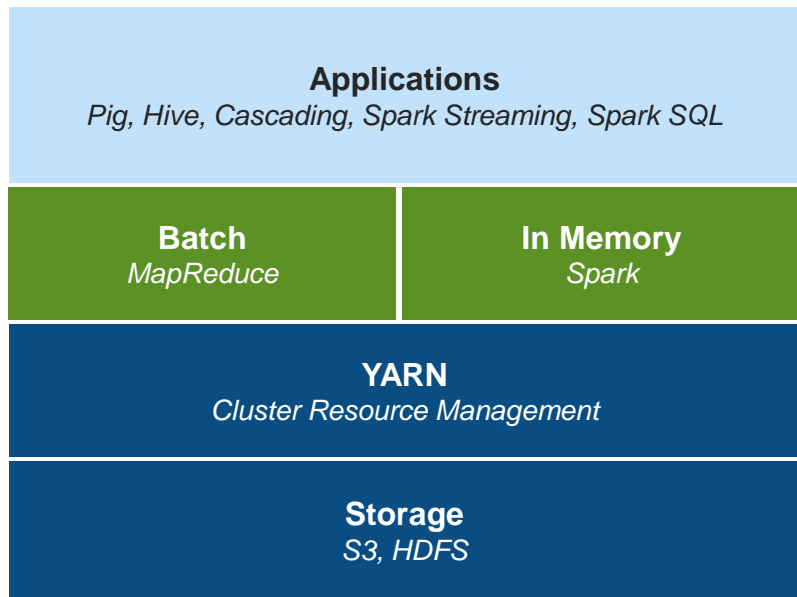


Spark architecture



- Run Spark driver in client or cluster mode
- Spark application runs as a YARN application
- SparkContext runs as a library in your program, one instance per Spark application
- Spark Executors run in YARN Containers on NodeManagers in your cluster

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

YARN schedulers—CapacityScheduler

- Default scheduler specified in Amazon EMR
- Queues
 - Single queue is set by default
 - Can create additional queues for workloads based on multitenancy requirements
- Capacity guarantees
 - Set minimal resources for each queue
 - Programmatically assign free resources to queues
- Adjust these settings using the classification `capacity-scheduler` in an EMR configuration object

What is a Spark executor?

- Processes that store data and run tasks for your Spark application
- Specific to a single Spark application (no shared executors across applications)
- Executors run in YARN containers managed by YARN NodeManager daemons

Inside Spark executor on YARN

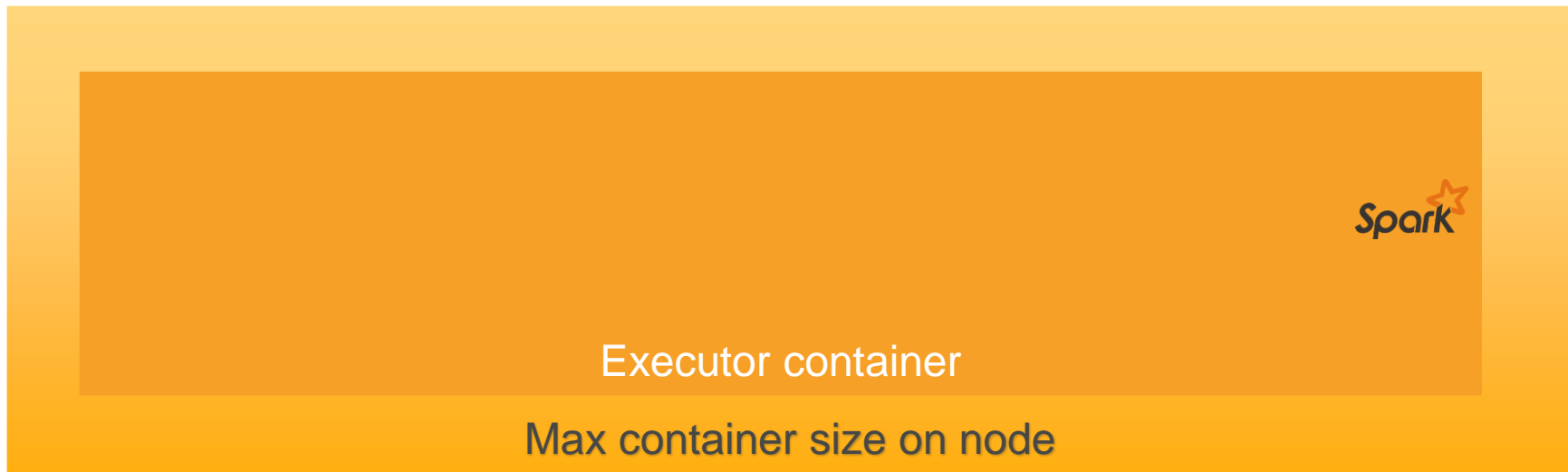
yarn.nodemanager.resource.memory-mb (classification: yarn-site)

- Controls the maximum sum of memory used by YARN container(s)
- EMR sets this value on each node based on instance type

Max container size on node

Inside Spark executor on YARN

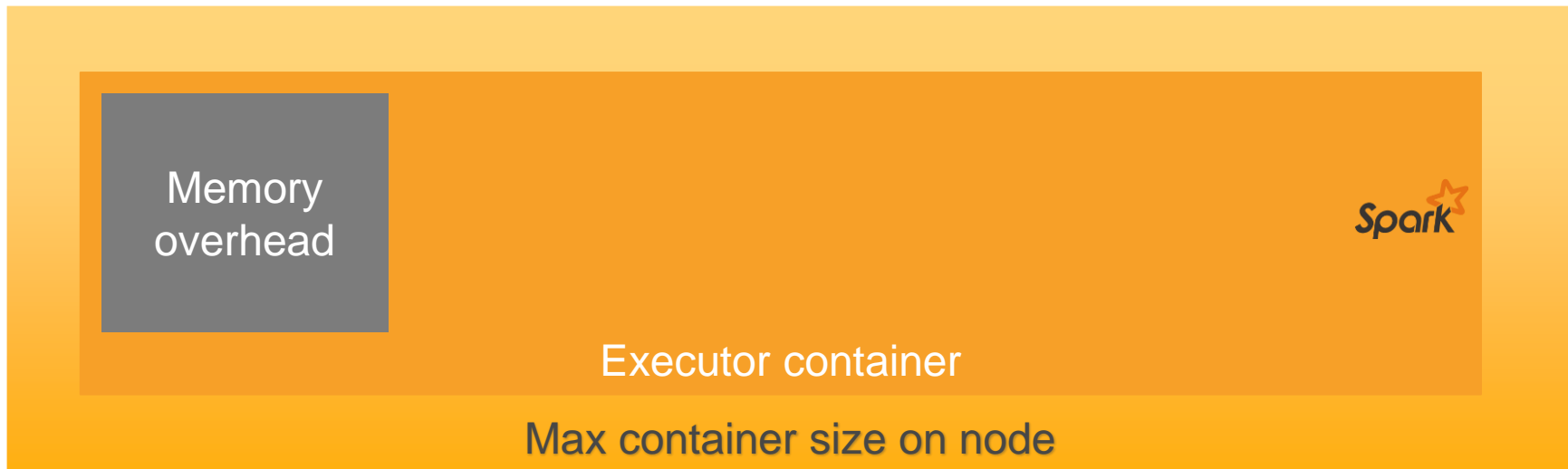
- Executor containers are created on each node



Inside Spark executor on YARN

spark.yarn.executor.memoryOverhead (classification: spark-default)

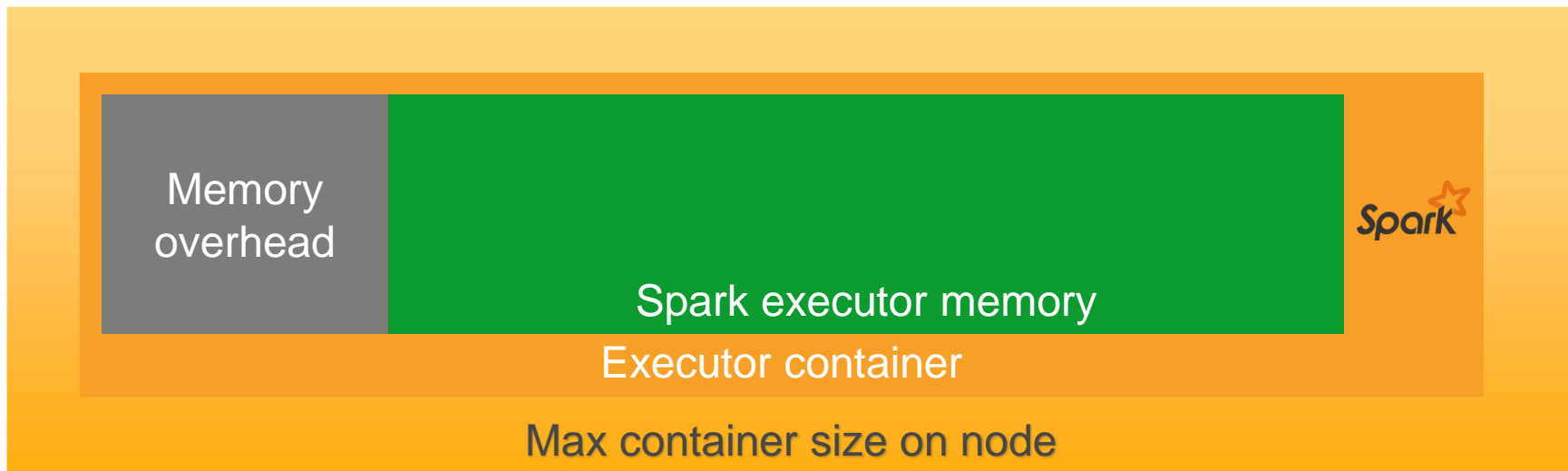
- Off-heap memory (VM overheads, interned strings, etc.)
- Roughly 10% of container size



Inside Spark executor on YARN

spark.executor.memory (classification: spark-default)

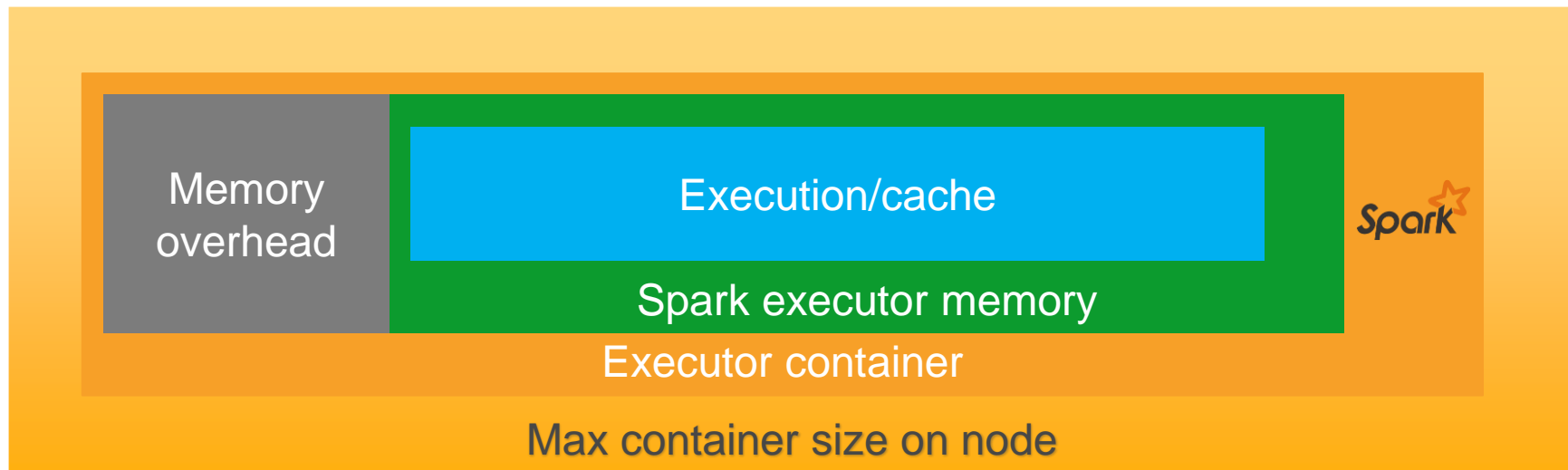
- Amount of memory to use per executor process
- EMR sets this based on the instance family selected for core nodes
- Cannot have different sized executors in the same Spark application



Inside Spark executor on YARN

spark.memory.fraction (classification: spark-default)

- Programmatically manages memory for execution and storage
- `spark.memory.storageFraction` sets percentage storage immune to eviction
- Before Spark 1.6: manually set `spark.shuffle.memoryFraction` and `spark.storage.memoryFraction`




Configuring executors—dynamic allocation

- Optimal resource utilization
- YARN dynamically creates and shuts down executors based on the resource needs of the Spark application
- Spark uses the executor memory and executor cores settings in the configuration for each executor
- Amazon EMR uses dynamic allocation by default (emr-4.5 and later), and calculates the default executor size to use based on the instance family of your core group

Properties related to dynamic allocation

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



Easily override spark-defaults

EMR console:

Edit software settings (optional) ⓘ



Enter configuration



Load JSON from S3

```
classification=spark-defaults,properties=[spark.executor.memory=15g,spark.executor.cores=4]
```

Configuration object:

```
[
  {
    "Classification": "spark-defaults",
    "Properties": {
      "spark.executor.memory": "15g",
      "spark.executor.cores": "4"
    }
  }
]
```

Configuration precedence: (1) SparkConf object, (2) flags passed to Spark Submit, (3) spark-defaults.conf

When to set executor configuration

- Need to fit larger partitions in memory
- GC is too high (though this is being resolved in Spark 1.5+ through work in Project Tungsten)
- Long-running, single tenant Spark Applications
- Static executors recommended for Spark Streaming
- Could be good for multitenancy, depending on YARN scheduler being used

More options for executor configuration

- When creating your cluster, specify `maximizeResourceAllocation` to create one large executor per node. Spark will use all of the executors for each application submitted.
- Adjust the Spark executor settings using an EMR configuration object when creating your cluster
- Pass in configuration overrides when running your Spark application with `spark-submit`

DataFrames

Minimize data being read in the DataFrame

- Use columnar forms like Parquet to scan less data
- More partitions give you more parallelism
 - Automatic partition discovery when using Parquet
 - Can repartition a DataFrame
 - Also you can adjust parallelism using with `spark.default.parallelism`
- Cache DataFrames in memory (StorageLevel)
 - Small datasets: `MEMORY_ONLY`
 - Larger datasets: `MEMORY_AND_DISK_ONLY`

For DataFrames: data serialization

- Data is serialized when cached or shuffled

Default: Java serializer

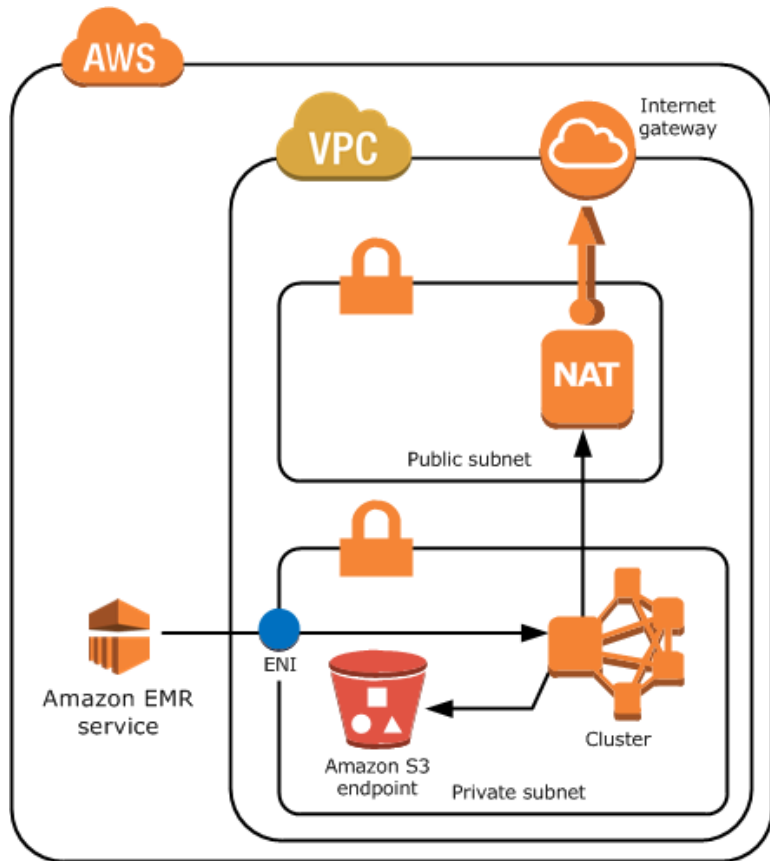
- **Kryo serialization (10x faster than Java serialization)**
 - Does not support all Serializable types
 - Register the class in advance

Usage: Set in SparkConf

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


Spark security on Amazon EMR

VPC private subnets to isolate network



- Use Amazon S3 endpoints for connectivity to S3
- Use Managed NAT for connectivity to other services or the Internet
- Control the traffic using security groups
 - ElasticMapReduce-Master-Private
 - ElasticMapReduce-Slave-Private
 - ElasticMapReduce-ServiceAccess

Spark on EMR security overview



Amazon S3

Encryption At-Rest

- HDFS transparent encryption (AES 256)
- Local disk encryption for temporary files using LUKS encryption
- EMRFS support for S3 client-side and server-side encryption

Encryption In-Flight

- Secure communication with SSL from S3 to EC2 (nodes of cluster)
- HDFS blocks encrypted in-transit when using HDFS encryption
- SASL encryption (digest-MD5) for Spark Shuffle

Permissions

- AWS Identity and Access Management (IAM) roles, Kerberos, and IAM users

Access

- VPC private subnet support, security groups, and SSH keys

Auditing

- AWS CloudTrail and S3 object-level auditing

AWS

S U M M I T

Thank you!

Jonathan Fritz - jonfritz@amazon.com

<https://aws.amazon.com/emr/spark>

<http://blogs.aws.amazon.com/bigdata>

