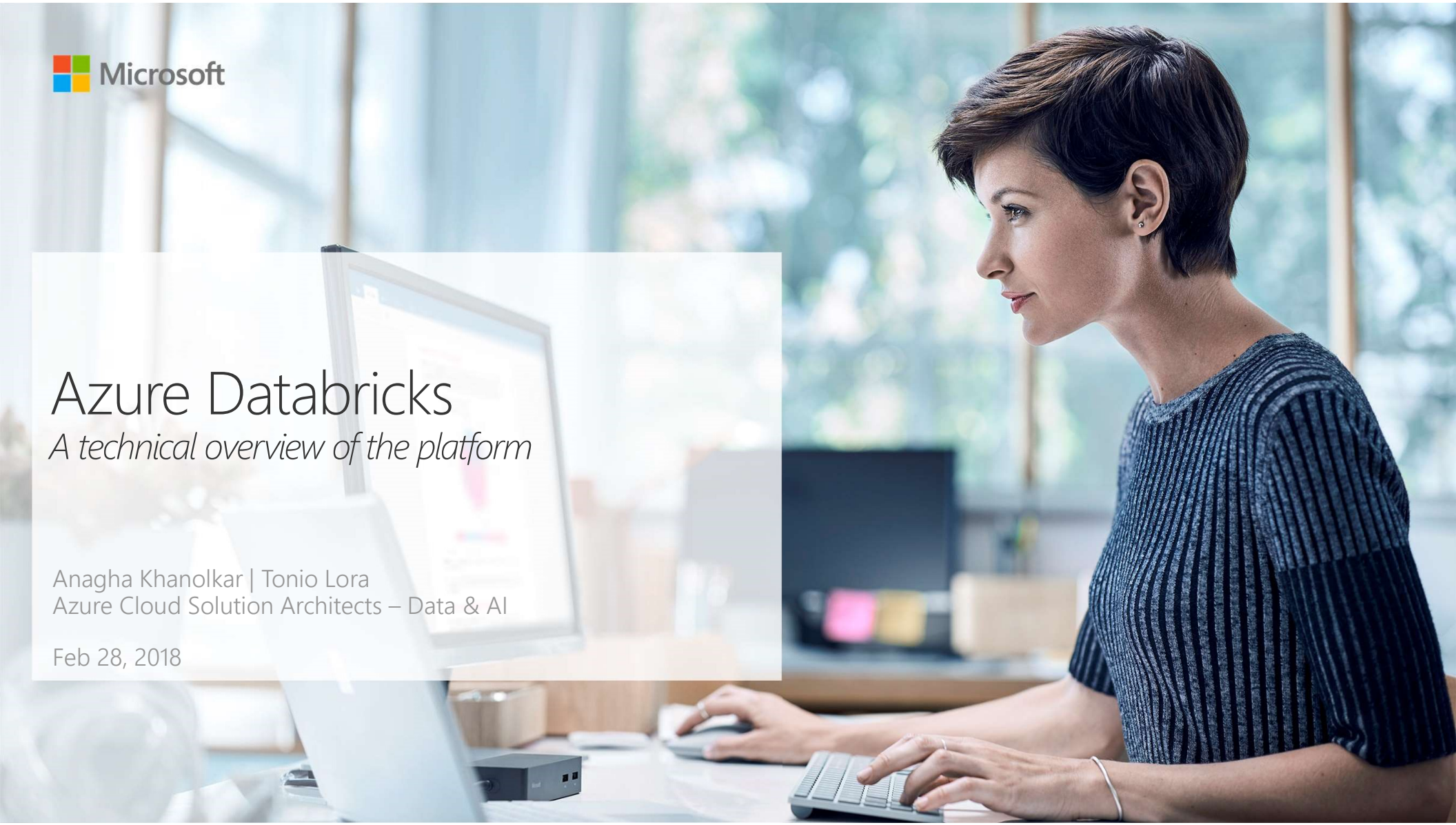


Microsoft

Azure Databricks

A technical overview of the platform

Anagha Khanolkar | Tonio Lora
Azure Cloud Solution Architects – Data & AI

Feb 28, 2018

Apache Spark

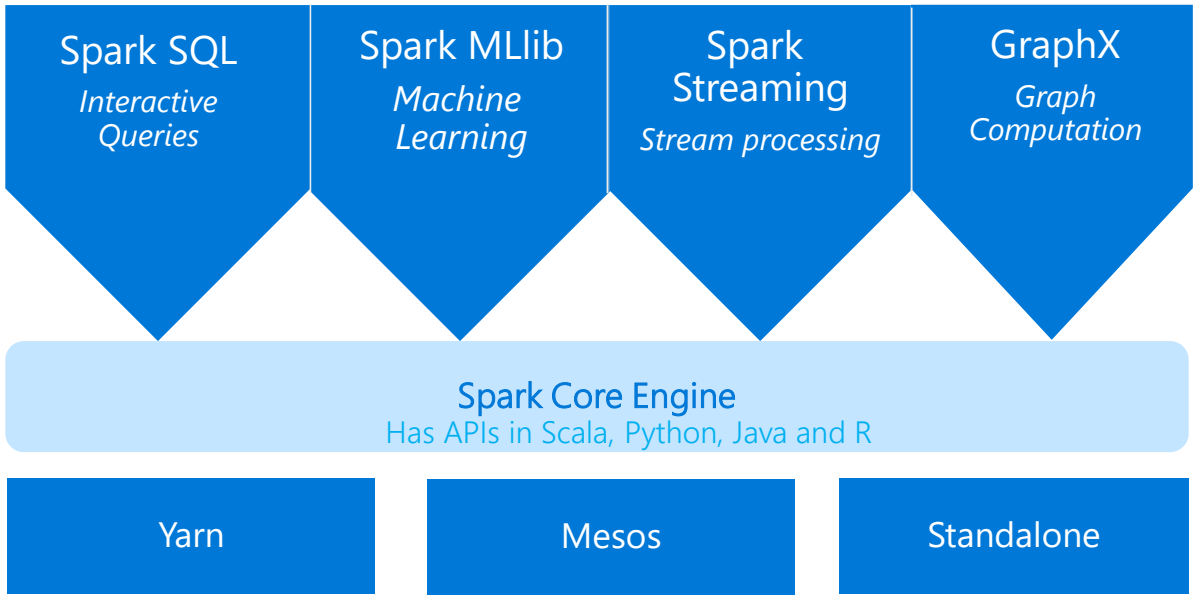
A gentle introduction

APACHE SPARK

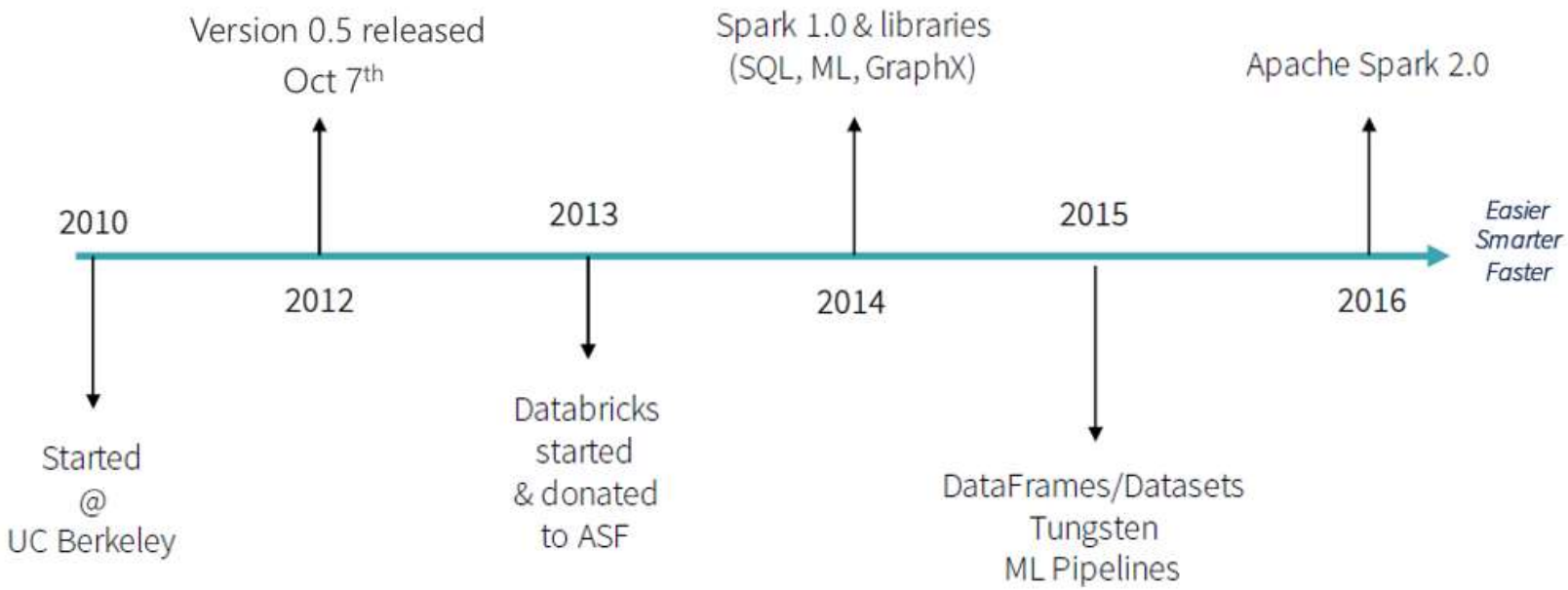
An open source, distributed, data processing framework for Big Data Analytics

Spark unifies:

- Batch Processing
- Interactive SQL
- Real-time processing
- Machine Learning
- Graph Processing



SPARK: A BRIEF HISTORY



SPARK - BENEFITS

PERFORMANCE

Using in-memory computing, Spark is considerably faster than Hadoop (100x in some tests)

PRODUCTIVITY

- Multiple languages supported – Scala, Python, Java
- Less code than map-reduce
- Easy-to-use APIs for processing large datasets
- Includes 100+ operators for transforming

MULTI-PURPOSE

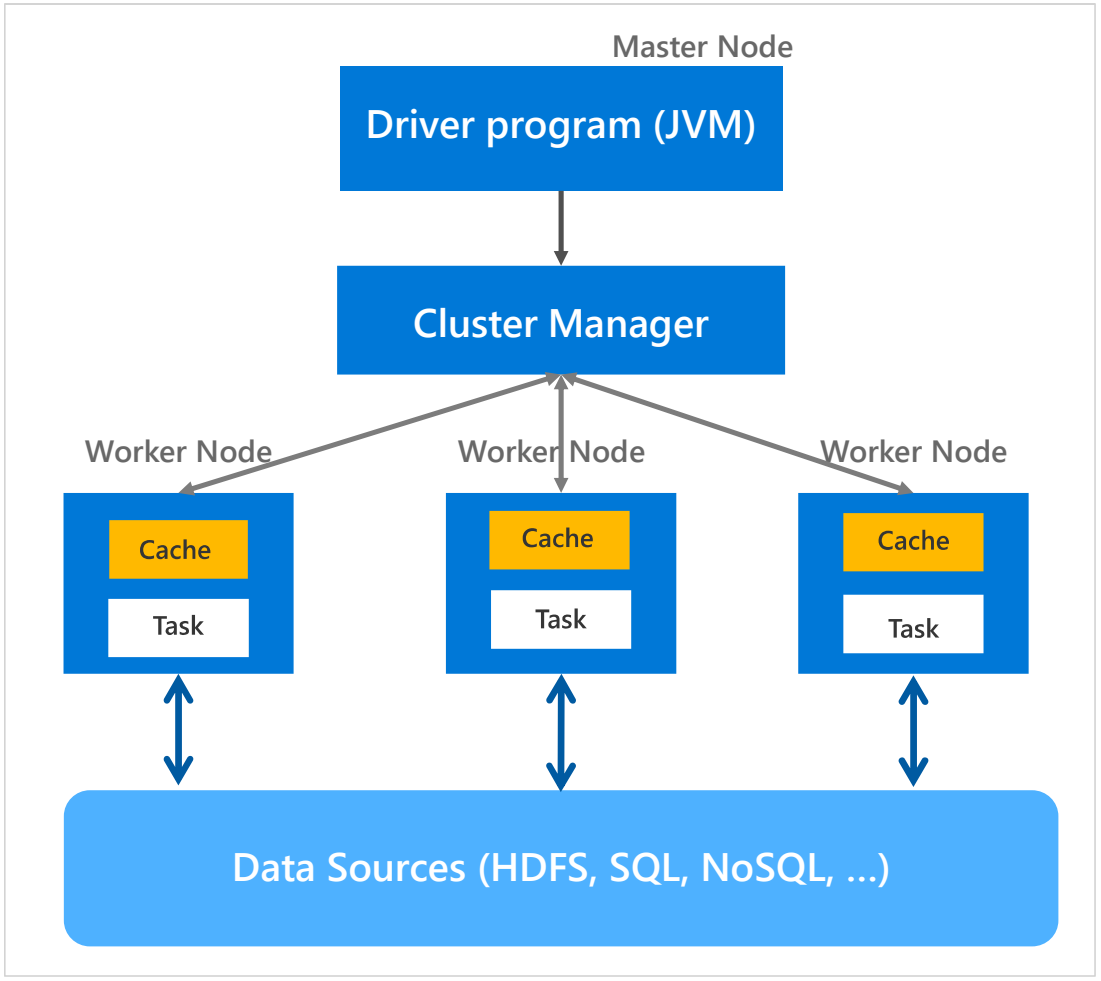
Supports multiple "at scale" processing needs - SQL/interactive/batch/ streaming/machine learning/graphs

CONNECTOR-RICH

Spark has built-in support for many data sources, has a rich ecosystem of ISV applications and a large contributor community

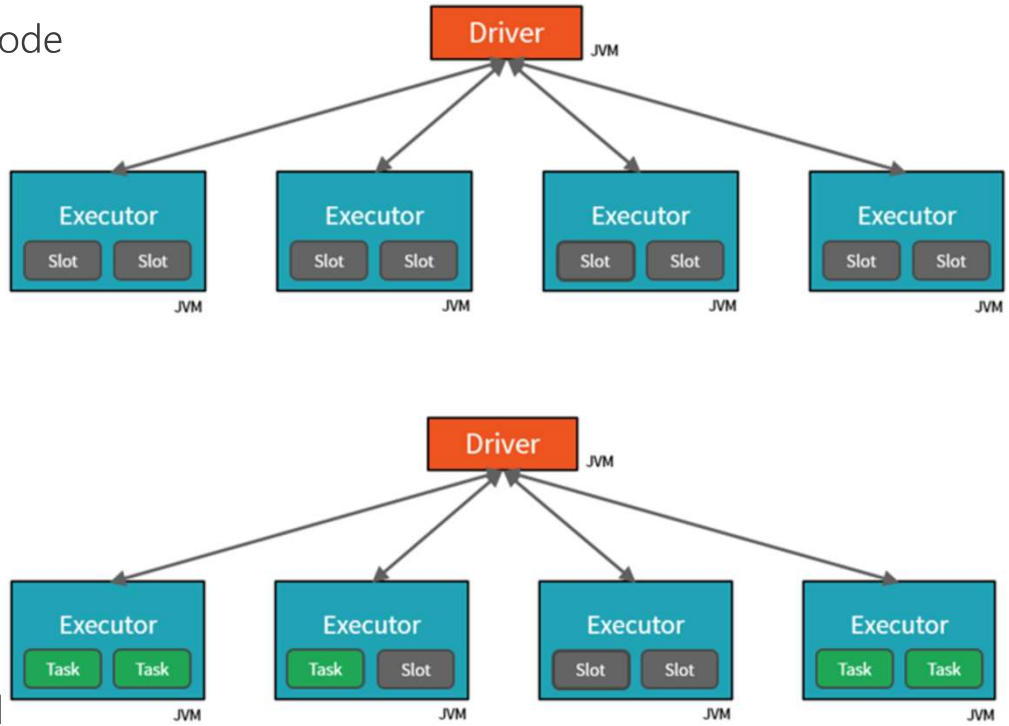
SPARK: CORE CONCEPTS – ARCHITECTURE

SPARK: ARCHITECTURE



SPARK: ARCHITECTURE- BASICS

- Driver is a JVM that runs your application; Runs on a node
- Executor is a JVM running on a node, typically, one instance per node (a unit of parallelism)
- Each executor has slots – determined by cores, CPUs per node (a unit of parallelism)
- Driver divides an application submitted into units of work or tasks and assigns to executor slots for parallel execution
- Driver determines how to partition data so it can be parallel processed across available resources
- Driver assigns a partition of data to each task
- When you read a file, based on instructions from drive each task reads the partition of data assigned to it and processes
- There is a scheduler component (FIFO default), fair scheduler commonly used



SPARK: INITIALIZING SPARK

- SparkSession provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with DataFrame and Dataset APIs.

Spark 2.x

```
// Create a SparkSession. No need to create SparkContext
// You automatically get it as part of the SparkSession
val warehouseLocation = "file:${system:user.dir}/spark-warehouse"
val spark = SparkSession
   .builder()
   .appName("Serengeti-Test-App")
   .config("spark.sql.warehouse.dir", warehouseLocation)
   .enableHiveSupport()
   .getOrCreate()
```

```
// read the json file and create the dataframe
val jsonFile = args(0)
val zipsDF = spark.read.json(jsonFile)
//filter all cities whose population > 40K
zipsDF.filter(zipsDF.col("pop") > 40000).show(10)
```

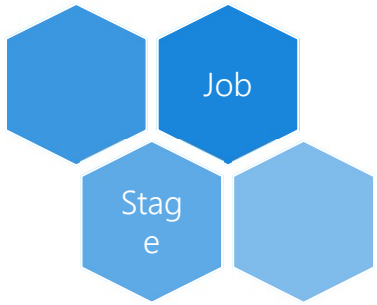
```
// Now create an SQL table and issue SQL queries against it without
// using the sqlContext but through the SparkSession object.
// Creates a temporary view of the DataFrame
zipsDF.createOrReplaceTempView("zips_table")
zipsDF.cache()
val resultsDF = spark.sql("SELECT city, pop, state, zip FROM zips_table")
resultsDF.show(10)
```

```
//drop the table if exists to get around existing table error
spark.sql("DROP TABLE IF EXISTS zips_hive_table")
//save as a hive table
spark.table("zips_table").write.saveAsTable("zips_hive_table")
//make a similar query against the hive table
val resultsHiveDF = spark.sql("SELECT city, pop, state, zip FROM zips_hive_table WHERE pop > 40000")
resultsHiveDF.show(10)
```

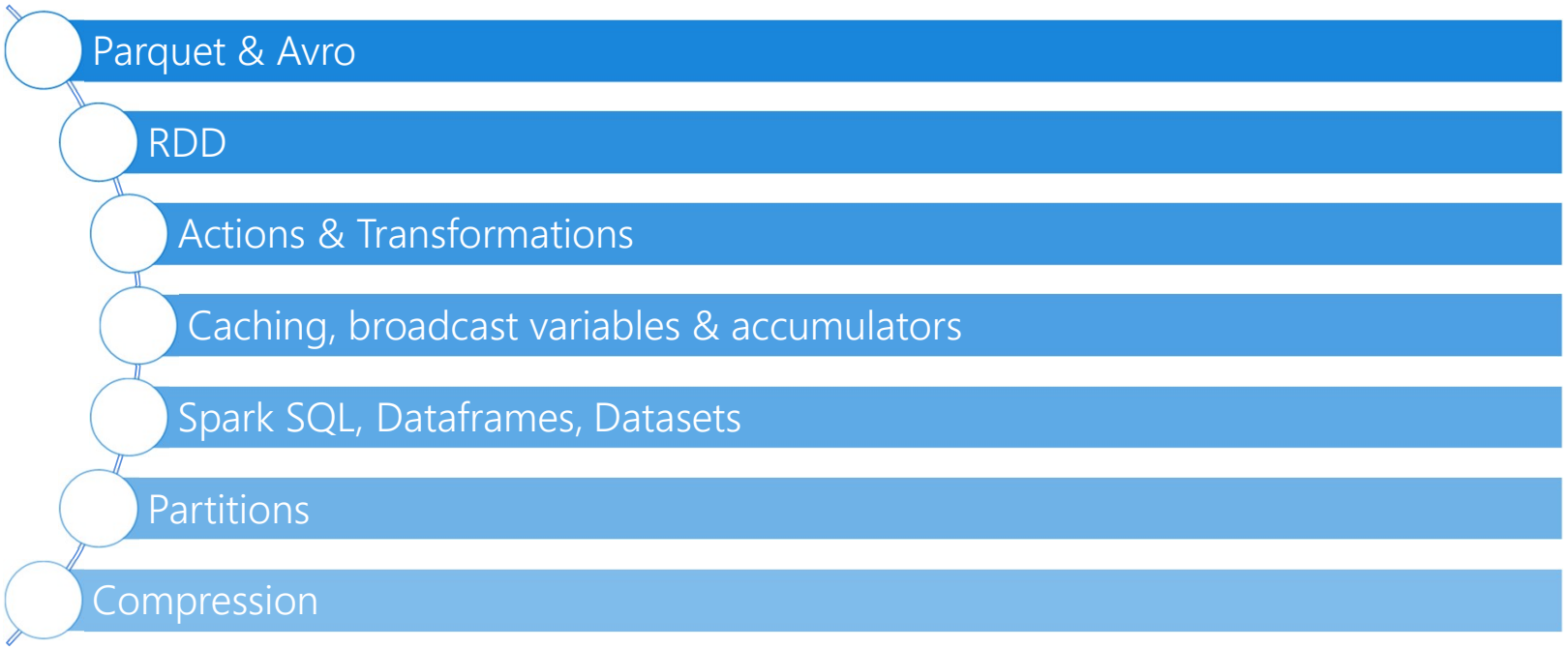
https://databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html

SPARK: ARCHITECTURE- JOBS & STAGES

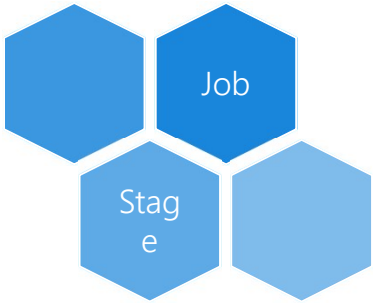
- Job is each parallelization action – results are returned to the driver
- A process you submit may spawn multiple jobs
- A job is broken into 1..many stages
- You can visualize your jobs and stages in the Spark UI – a valuable tool to tune your job



SPARK: CORE CONCEPTS – DATA ENGINEERING



PARQUET AND AVRO



- Are persistent formats
- Are space-efficient
- Are query-efficient
- Are self-describing – schemas are part of the data files
- Support schema evolution
- Support nested structures
- Are splittable

- Are the defacto standard with big data on Hadoop/Spark
- Prefer Avro for row-level access
- Prefer Parquet for columnar, analytical workloads
- Parquet is default in Spark

SPARK: RDD API

- Resilient Distributed Dataset
- RESILIENT: Fault tolerant
- DISTRIBUTED: Computed across nodes in a cluster
- DATASET: Collection of partitioned data

- Parallelized operations on collections of data
- Immutable
- Lazily evaluated
- Lineage tracked
- Assembly language of Spark/lowest level API
- Use for most complex unstructured data processing
- Scala, Python and Java - no R yet

Parallelized collections:

```
val arrData = Array(1, 2, 3, 4, 5)
val distDataRDD = sc.parallelize(arrData)
```

External datasets:

```
val extDataRDD = sc.textFile("nyc_taxi_dataset.txt")
extDataRDD: org.apache.spark.rdd.RDD[String]
```

RDD- ACTIONS & TRANSFORMATIONS

- ACTIONS

Are commands that are computed by Spark right at the time of their execution.

They consist of running all of the previous transformations in order to get back an actual result.

An action is composed of one or more jobs which consists of tasks that will be executed by the workers in parallel where possible

- TRANSFORMATIONS

Are operations that will not be completed at the time you write and execute the code in a cell - they will only get executed once you have called a action.

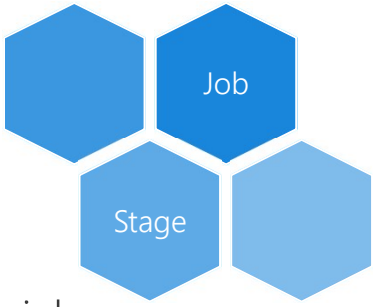
An example of a transformation might be to convert an integer into a float or to filter a set of values.

Transformations *(lazy)*	Actions
select	show
distinct	count
groupBy	collect
sum	save
orderBy	
filter	
limit	

SPARK: ARCHITECTURE- JOBS & STAGES

Now that you know what actions and transformations are:

- Job is each parallelization action – results are returned to the driver
- A process you submit may spawn multiple jobs – each action spawns a job
- A job is broken into 1..many stages – a stage for each transformation step needed
- You can visualize your jobs and stages in the Spark UI – a valuable tool to tune your job



CACHING

A feature in Spark to optimize performance by persisting in memory

Ideal where you need to iteratively read a dataset in a program

Various configurations available in Spark based on architectural considerations

Caching is lazy, need to materialize RDD with an action

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

```
val greenTaxiTripsDF = spark.sql("select * from taxi_db.green_taxi_trips")
greenTaxiTripsDF.cache()
```

BROADCAST VARIABLES & ACCUMULATORS

BROADCAST VARIABLES

Broadcast variables is a performance optimization that allows the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. E.g. a small reference dataset like US state code-name mapping

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

ACCUMULATORS

Accumulators are counters for global count across partitions

```
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(My Accumulator), value: 0)

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

scala> accum.value
res2: Long = 10
```

SPARK SQL MODULE

- Spark SQL is a Spark module for structured data processing.
- The interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed – Spark can optimize
- There are several ways to interact with Spark SQL including SQL and the Dataset API.
- When computing a result the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation.

SQL

- Spark SQL is used to execute SQL queries
- It can be used to interact with a Hive installation
- Can be used from Spark SQL CLI
- And over JDBC/ODBC

DATASET API

- A Dataset is a distributed collection of data
- Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine
- A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.)
- The Dataset API is available in Scala and Java

```scala
// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit,
// you can use custom classes that implement the Product interface
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
// +----+---+
```

```scala
// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name
val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

DATAFRAME API

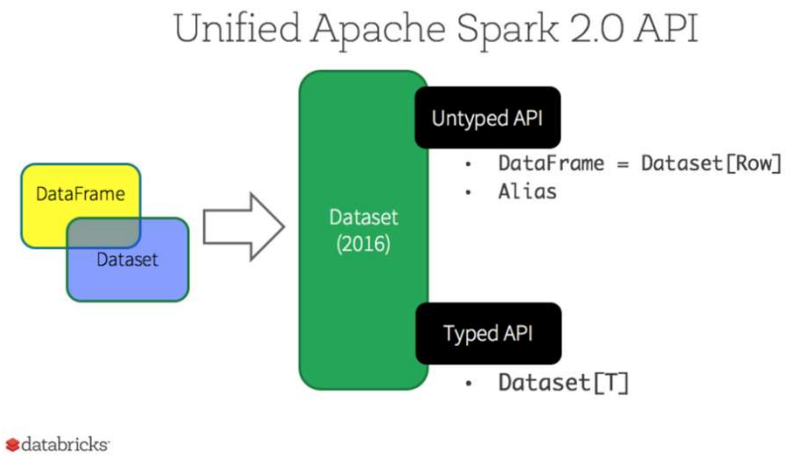
- A DataFrame is data organized into named columns
- It is the most performant API (Tungsten/Catalyst optimizations)
- Has uniform API across languages
- Can build ML pipelines
- It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood
- DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs
- The DataFrame API is available in Scala, Java, Python, and R
- In Scala and Java, a DataFrame is represented by a Dataset of Rows

```scala
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

```scala
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

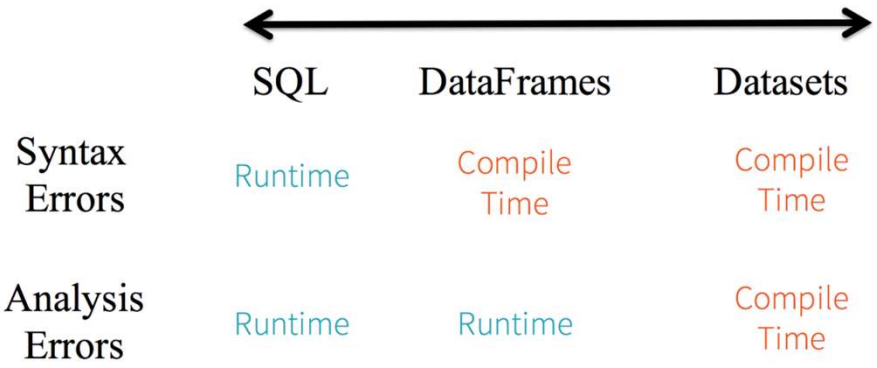
```scala
// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

DATASET VERSUS DATAFRAME API



Typed and Un-typed APIs

Language	Main Abstraction
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python*	DataFrame
R*	DataFrame



RUNNING SQL QUERIES PROGRAMMATICALLY

- TEMPORARY VIEW

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates.

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

- GLOBAL TEMPORARY VIEW

If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view.

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

GENERIC LOAD/SAVE FUNCTIONS

- DEFAULT

Parquet – unless otherwise specified

```
val usersDF = spark.read.load("examples/src/main/resources/users.parquet")
usersDF.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

- OTHER

json, parquet, jdbc, orc, libsvm, csv, text

```
val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

```
val sqlDF = spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`")
```

- SAVE MODE

Multiple options

Scala/Java	Any Language	Meaning
`SaveMode.ErrorIfExists` (default)	`"error"` (default)	When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.
`SaveMode.Append`	`"append"`	When saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data.
`SaveMode.Overwrite`	`"overwrite"`	Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame.
`SaveMode.Ignore`	`"ignore"`	Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected to not save the contents of the DataFrame and to not change the existing data. This is similar to a `CREATE TABLE IF NOT EXISTS` in SQL.

HIVE

- HIVE/PERSISTENT TABLES

dataFrame.saveAsTable

```
df.write.option("path", "/user/akhanolk/ref_data/us_states").saveAsTable("us_states_codeset")
```

- TABLE PARTITION

Split your data into directory trees based on access patterns – performance optimization

```
taxiDF.write.partitionBy("trip_month").format("parquet").save("trips_by_month.parquet")
```

```
//Sync partitions between file system and Hive table
spark.sql("MSCK REPAIR TABLE taxi_db.yellow_taxi_trips")
```

- BUCKETING

Partition, and optionally sort, the data based on a subset of columns while it is written out (a one-time cost), while making successive reads of the data more performant for downstream jobs if the SQL operators can make use of this property.

```
taxiDF
  .write
  .partitionBy("trip_month")
  .bucketBy(10, "payment_type")
  .saveAsTable("trips_months_by_payment_type")
```

- BROADCAST JOINS

Are default in Spark SQL; Small tables are broadcasted

SPARK SQL & JDBC

```scala
// Loading data from a JDBC source
val jdbcDF = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .load()

val connectionProperties = new Properties()
connectionProperties.put("user", "username")
connectionProperties.put("password", "password")
val jdbcDF2 = spark.read
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)

// Saving data to a JDBC source
jdbcDF.write
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .save()

jdbcDF2.write
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)

// Specifying create table column data types on write
jdbcDF.write
  .option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)")
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)
```

SPARK - PARTITIONS

- PARTITIONS DURING READS

In Spark 2.x, there are a number of optimization because of which Spark intelligently determines an optimal number of partitions based on slots and data size

```
val partitions = initialDF.rdd.partitions.size
printf("Partition count: ", partitions)
```

- PARTITIONS IN GENERAL

Unit of parallelism; Tune for performance; Generally 200 MB/partition is considered optimal
Too much is not good, too little is not good
Ensure every slot is used up

- NUANCES

Parquet is highly compressed and compact on disk but uncompressed in RAM
CSV is large on disk, smaller in RAM
So using disk size is not a good practice

- THEN, HOW?
  Read in data, cache
  Check RDD partition length and size & tune

PARTITIONS

- COALESCE(numPartitions)

Returns a new Dataset that has exactly numPartitions partitions, when fewer partitions are requested
If a larger number of partitions is requested, it will stay at the current number of partitions.

Is a narrow transformation and can only be used to reduce the number of partitions – performant because it does not involve shuffle

But does not guarantee even distribution

- REPARTITION(numPartitions)

Returns a new Dataset that has exactly numPartitions partitions

Is a wide transformation because of shuffle

Less performant compared to coalesce

Guarantees even distribution

Can be used to increase or decrease partition count

COMPRESSION

- Compression reduces space usage and improves performance of read/write/shuffle
- The tradeoff is between – compression ratio, speed, CPU usage, splittability
- Depending on the type of persistence format – choose the right compression codec

- AVRO:
  deflate and snappy compression supported;
  snappy is faster, but deflate is slightly more compact

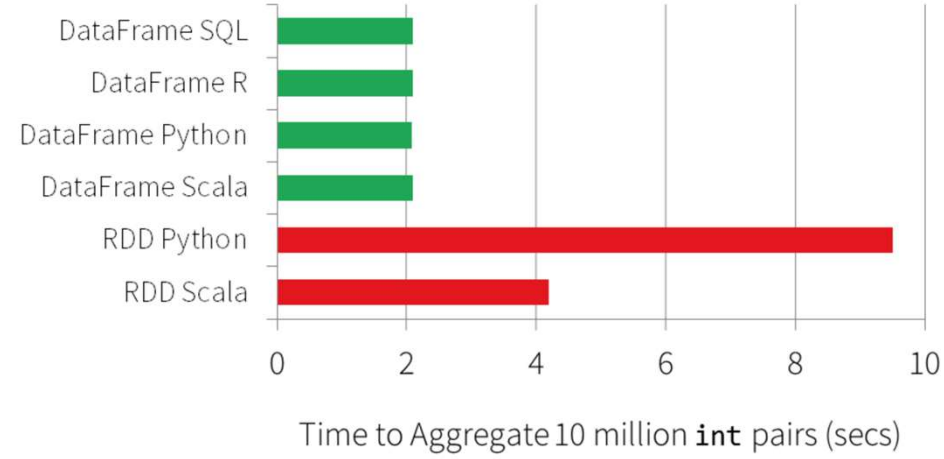
- PARQUET:
  lzo, gzip and snappy (default) compression supported;
  gzip – slow, uses most CPU, but compresses best
  lzo, snappy – splittable
  snappy – overall better than lzo

LANGUAGE CHOICE- RAMIFICATIONS

SOME ARCHITECTURAL CONSIDERATIONS/TRADEOFFS

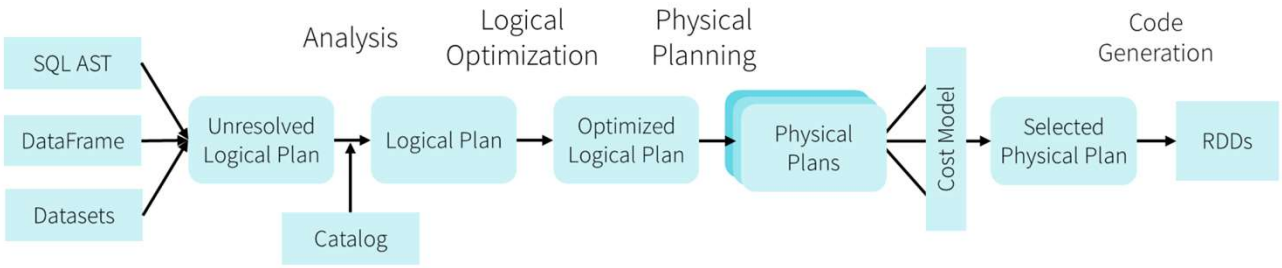
- Feature availability
- Feature rollout
- Performance
- Skill-set of team
- Other

Performance:



CATALYST & TUNGSTEN

- **CATALYST OPTIMIZER**
  Catalyst Optimizer is a query optimizer fundamental to Spark SQL



- **TUNGSTEN**
  Project Tungsten aims at improving the performance of Spark SQL, from a CPU, memory perspective and across other dimensions. Multiple generations of Tungsten have been rolled out

STREAM PROCESSING

- Two APIs – Spark Streaming (2013) and Structured Streaming (2017, now GA)

- SPARK STREAMING
  Referred to by some as legacy spark streaming
  Streaming data abstraction is called Dstream for discretized stream- RDD based
  Can be used for streaming structured and unstructured data
  Lower level API, support for windowing, stateful streaming

- STRUCTURED STREAMING
  High-level streaming API built on Apache Spark SQL engine
  Supports structured data only, dataframe based
  More performant due to Catalyst optimizer
  Think of stream as a continuously growing table
  Run arbitrary queries on the streaming table & sink when needed
  Abstracts out reasoning for streaming making it only slightly different from batch processing
  Sessions, sources, sinks
  Event time capture from within payload for late data handling
  Windowing, aggregations, improved stateful streaming,

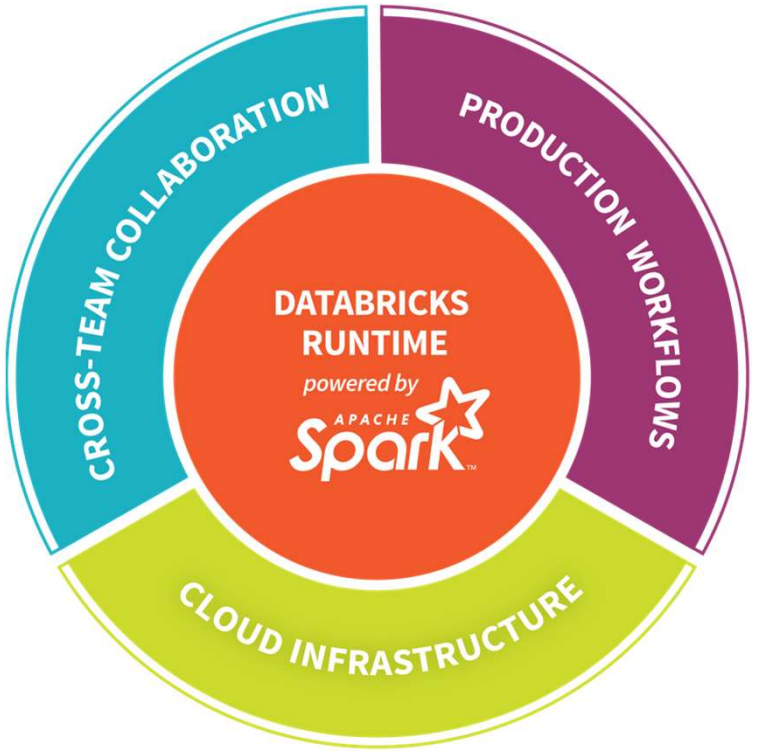
SCALA

- Invented by Martin Odersky
- Sca (scalable) la (language)
- Functional programming – immutable data, pure functions (deterministic, stateless, self-contained), implicit looping and iteration
- Runs on JVM, compiles to bytecode
- More expressive than Java
- Interoperable with Java programs and libraries
- Strong typing, catch runtime errors
- Integration wit popular IDEs
- Support for multiple programming paradigms – functional, imperative (Java – seq of statements to reach a set goal), object-oriented
- Spark is written in Scala – features are released first in Scala

About Databricks – the company

DATABRICKS - COMPANY OVERVIEW

- Founded in late 2013
- By the creators of Apache Spark, original team from UC Berkeley AMPLab
- Largest code contributor code to Apache Spark
- Provides certifications such as Databricks Certified Application, Databricks Certified Distribution and Databricks Certified Developer
- Main Product: The Unified Analytics Platform
- In Oct 2017, introduced Databricks Delta (currently in private preview) - a unified data management system to simplify large-scale data management



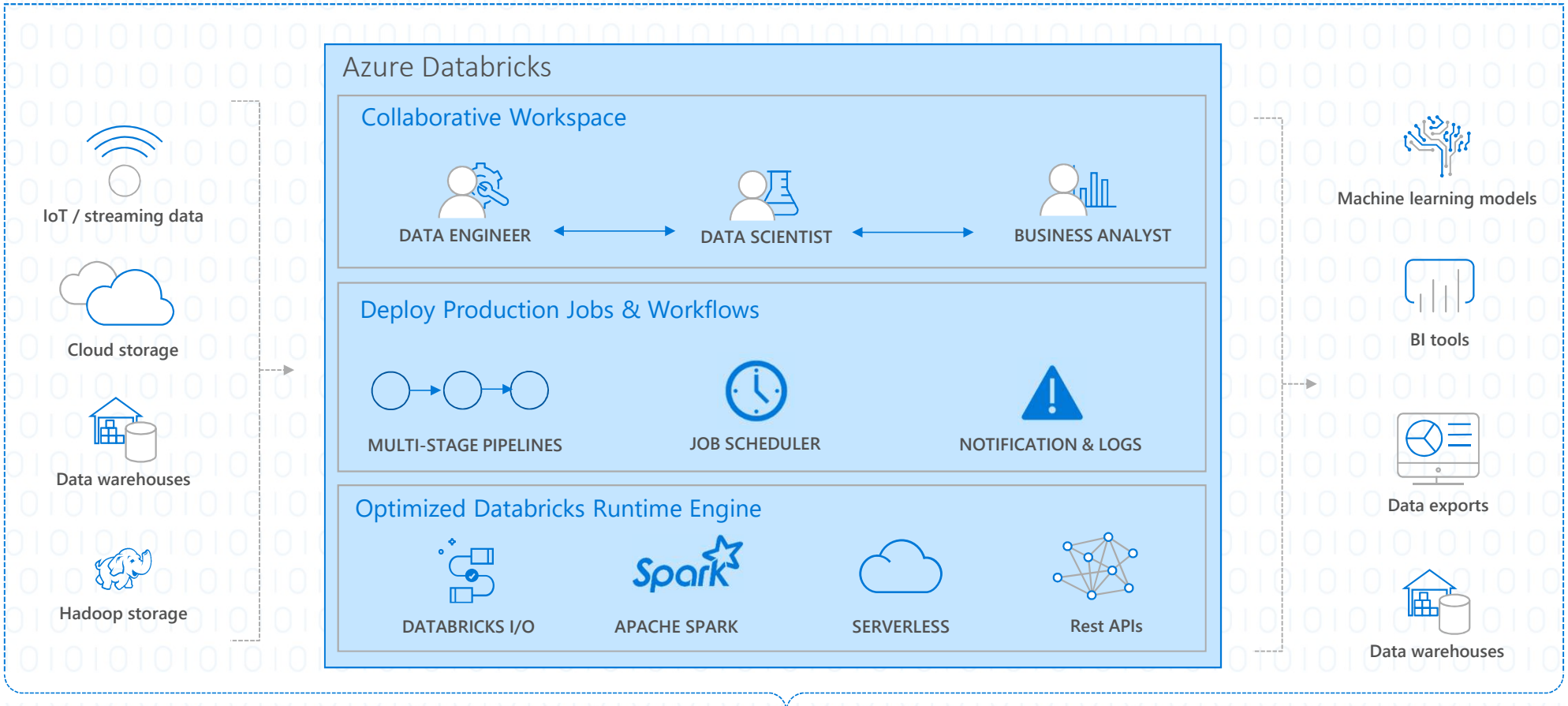
Azure Databricks

Spark as a managed service on Azure

AZURE DATABRICKS

- Azure Databricks is a **first party** service on Azure
- Databricks on Azure has the following Azure services integration:
  - Azure Active Directory: for user authentication
  - Azure Storage Services: Support for Azure Blob Storage and Azure Data Lake Store as persistence layer
  - Azure SQL Datawarehouse: Optimized connector
  - Azure Cosmos DB: Optimized connector
  - Apache Kafka for HDInsight: General Spark streaming Kafka connector
  - Azure Power BI: For rich data visualization

AZURE DATABRICKS



Enhance Productivity

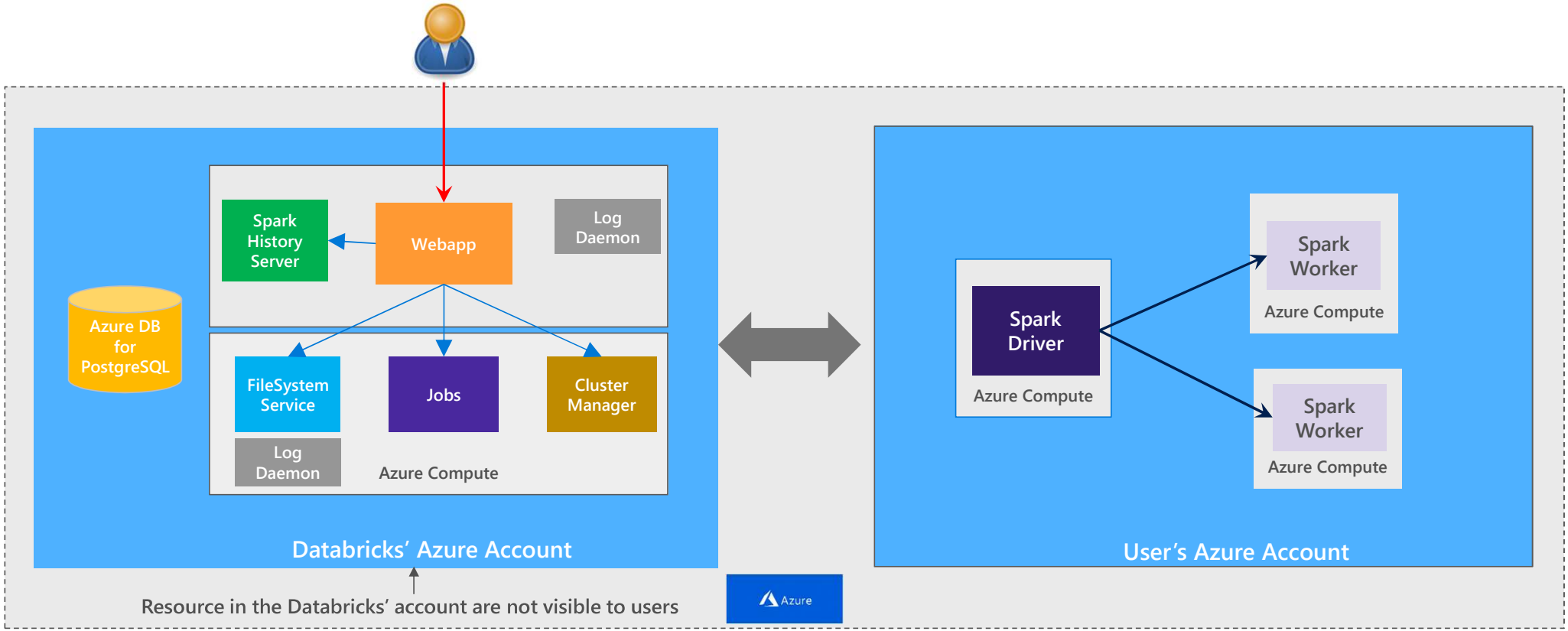
Build on secure & trusted cloud

Scale without limits

Azure Databricks

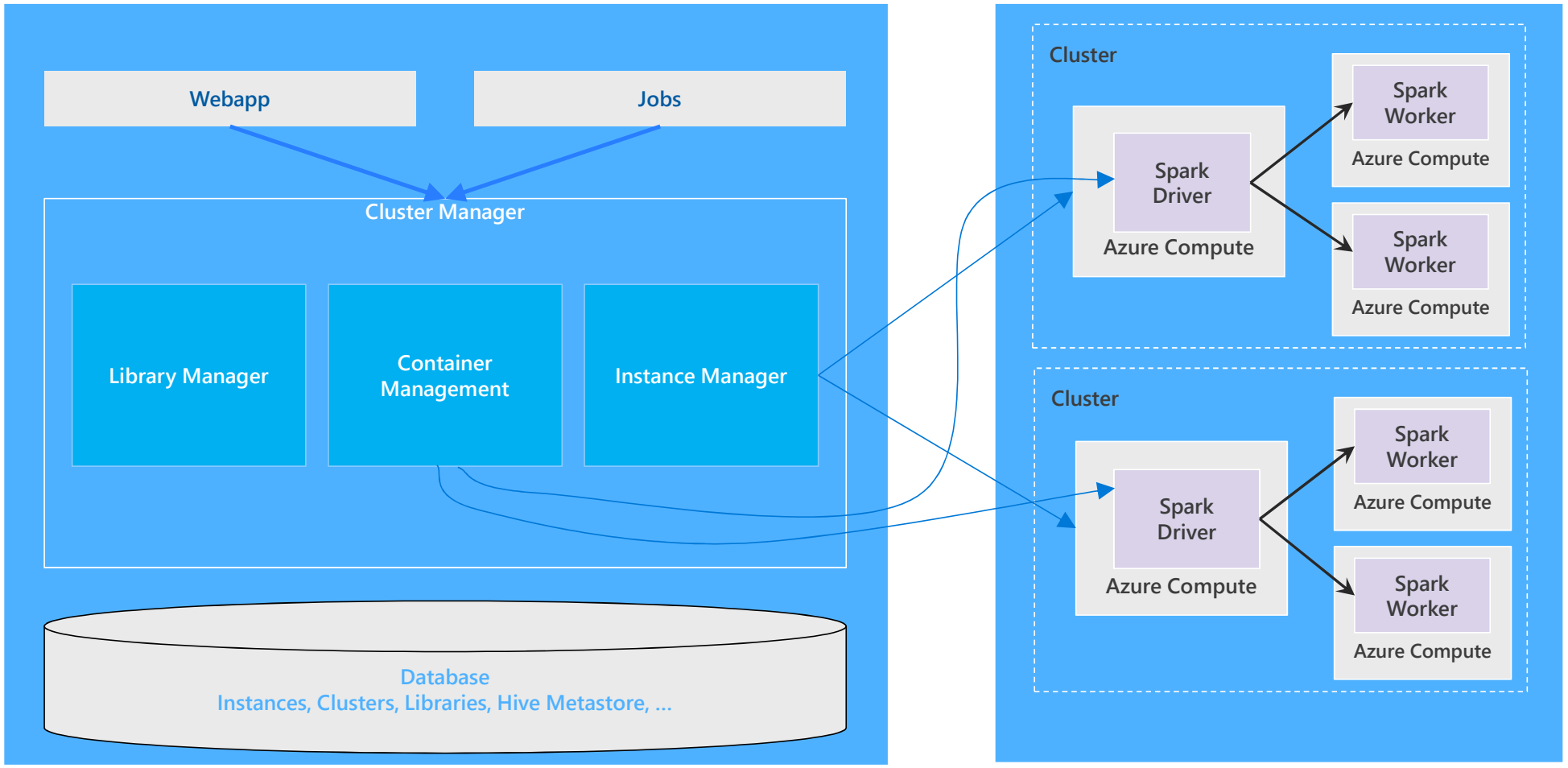
Core concepts

AZURE DATABRICKS CLUSTER ARCHITECTURE



Update: Today you can bring your own metastore

CLUSTER MANAGER ARCHITECTURE



Secure Collaboration

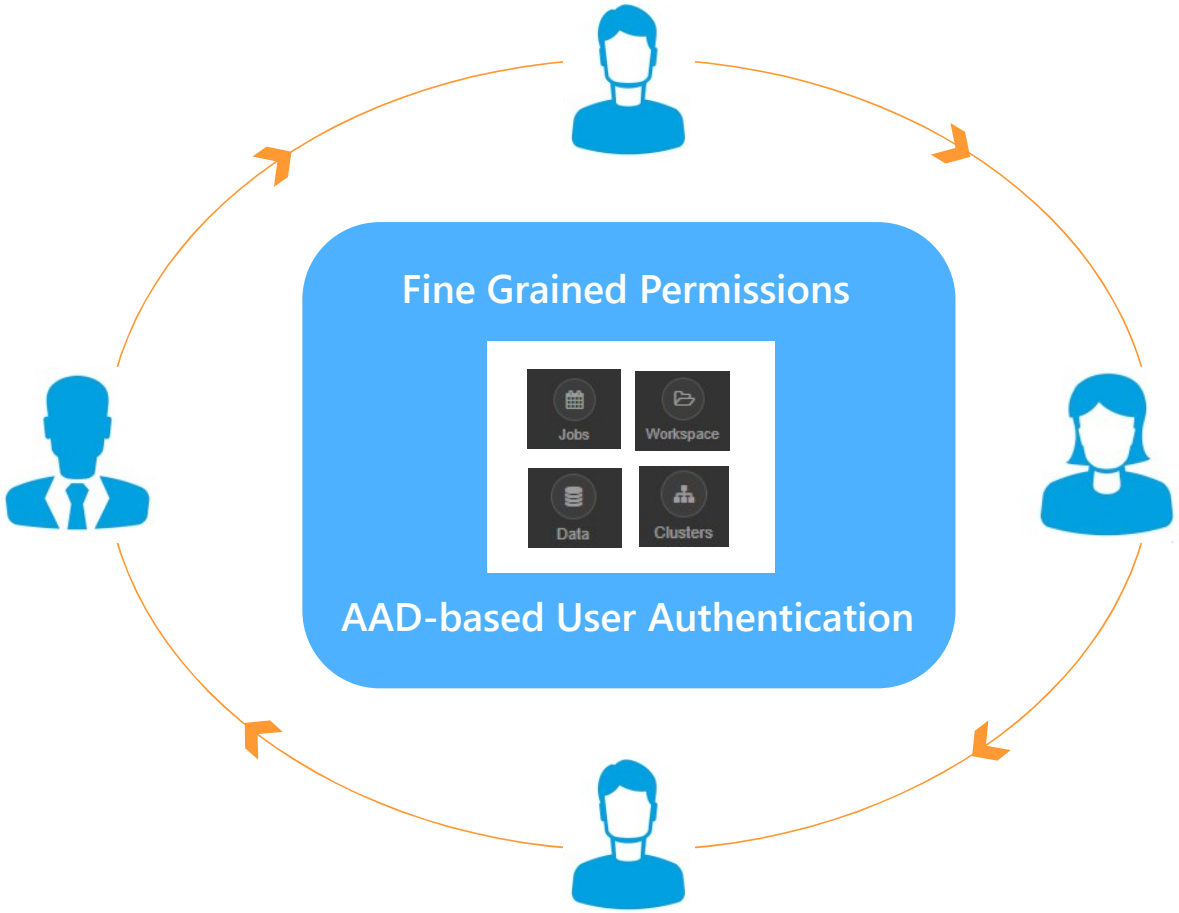
SECURE COLLABORATION

Azure Databricks enables *secure* collaboration between colleagues

- With Azure Databricks colleagues can *securely share* key artifacts such as Clusters, Notebooks, Jobs and Workspaces
- Secure collaboration is enabled through a combination of:

Fine grained permissions: Defines who can do what on which artifacts (access control)

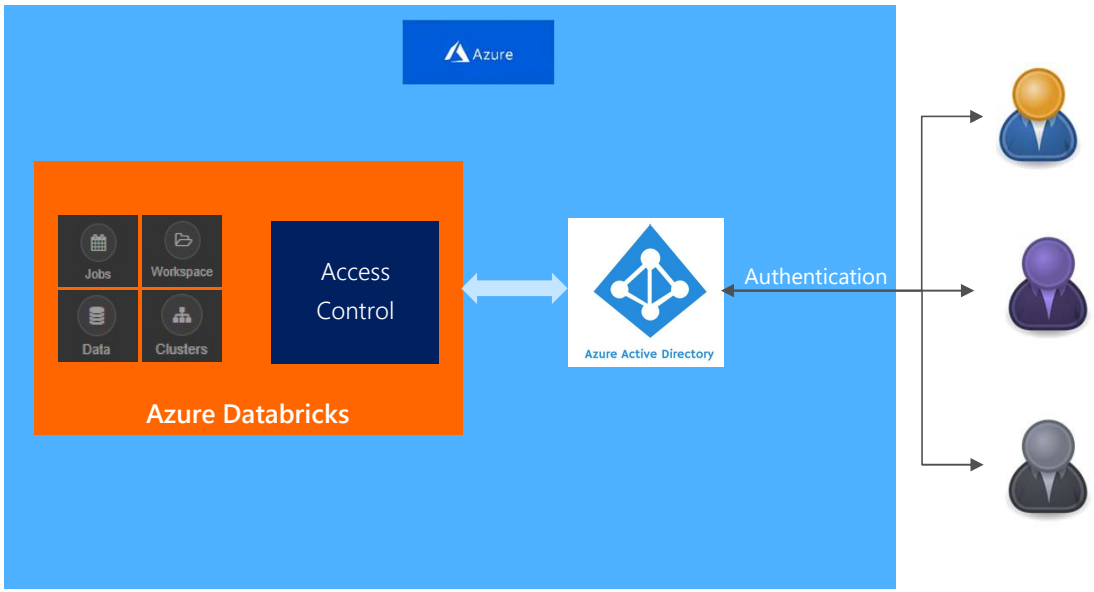
AAD-based authentication: Ensures that users are actually who they claim to be



AZURE DATABRICKS INTEGRATION WITH AAD

Azure Databricks is integrated with AAD—so Azure Databricks users are just regular AAD users

- There is no need to define users—and their access control—separately in Databricks.
- AAD users can be used directly in Azure Databricks for all user-based access control (Clusters, Jobs, Notebooks etc.).
- Databricks has delegated user authentication to AAD enabling single-sign on (SSO) and unified authentication.
- *Notebooks, and their outputs, are stored in the Databricks account*. However, AAD-based access-control ensures that only authorized users can access them.



DATABRICKS ACCESS CONTROL

Access control can be defined at the user level via the Admin Console

Access Control can be defined for Workspaces, Clusters, Jobs and REST APIs

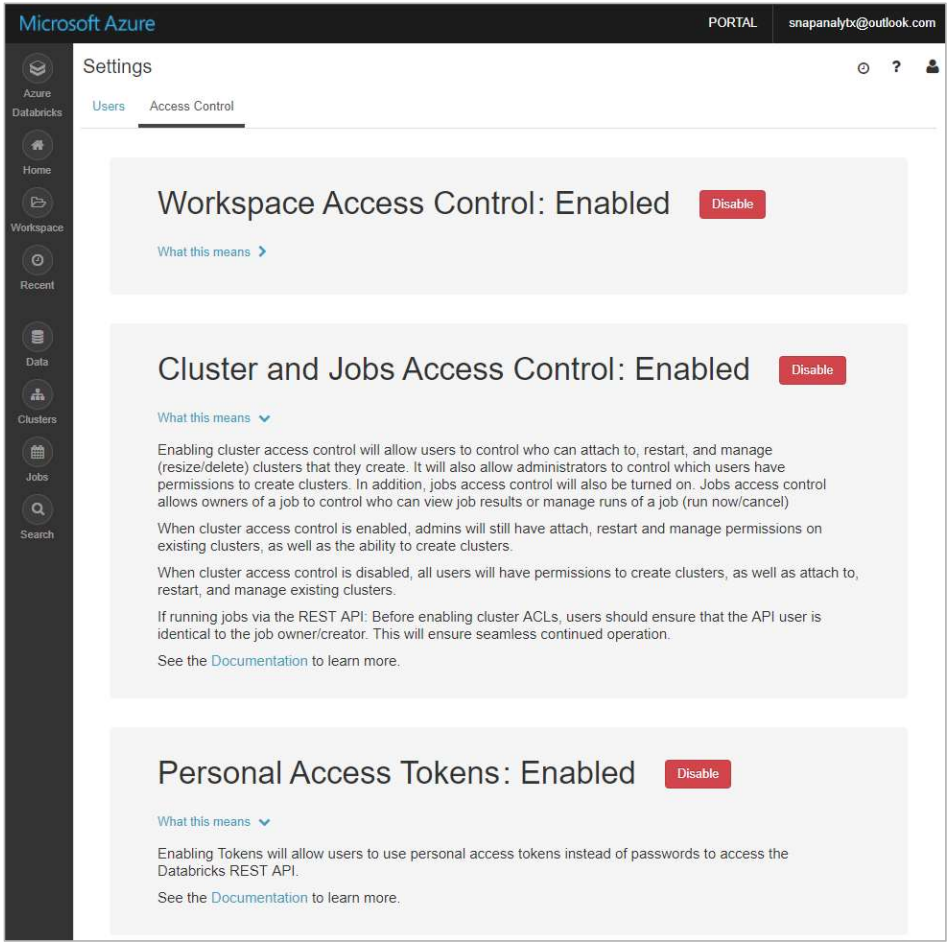
Databricks Access Control

Workspace Access Control	Defines who can who can view, edit, and run notebooks in their workspace
Cluster Access Control	Allows users to who can attach to, restart, and manage (resize/delete) clusters. Allows Admins to specify which users have permissions to create clusters
Jobs Access Control	Allows owners of a job to control who can view job results or manage runs of a job (run now/cancel)
REST API Tokens	Allows users to use personal access tokens instead of passwords to access the Databricks REST API

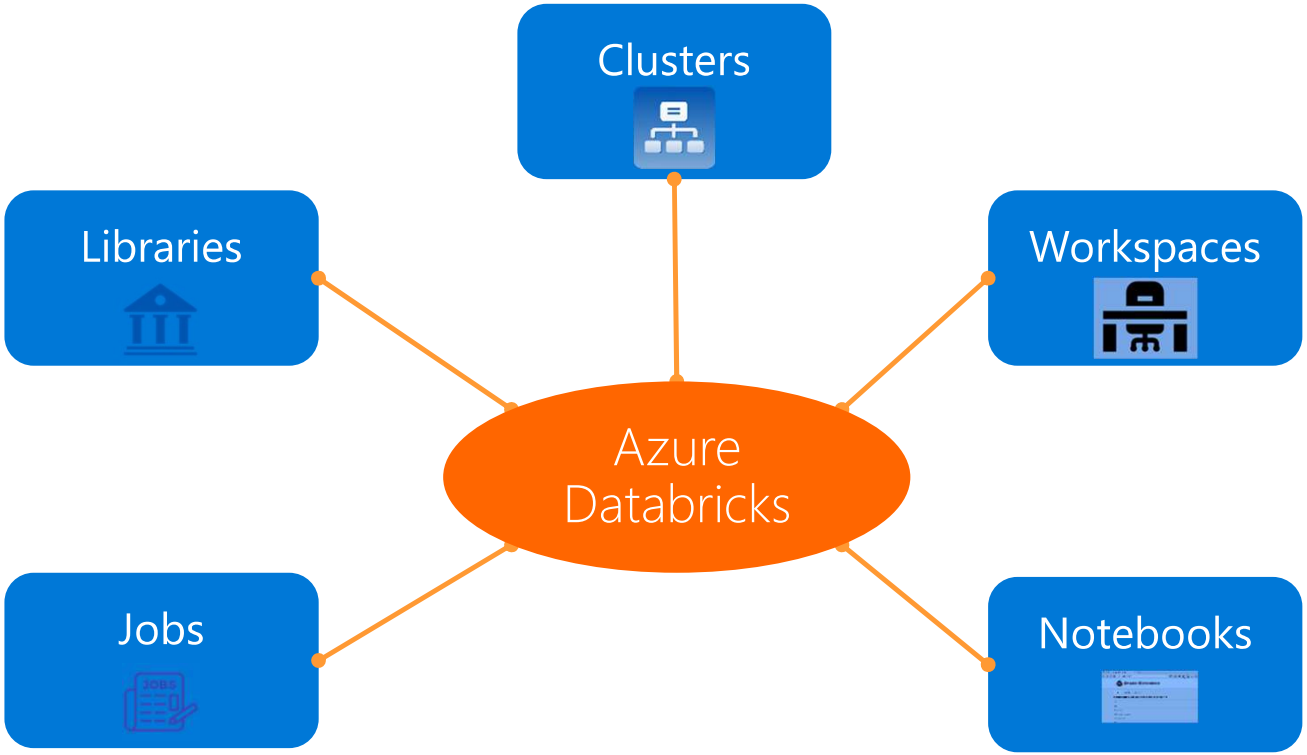
ENABLE/DISABLE ACCESS CONTROL

Access Control can be selectively enabled or disabled for:

- Workspaces,
- Clusters,
- Jobs
- REST APIs



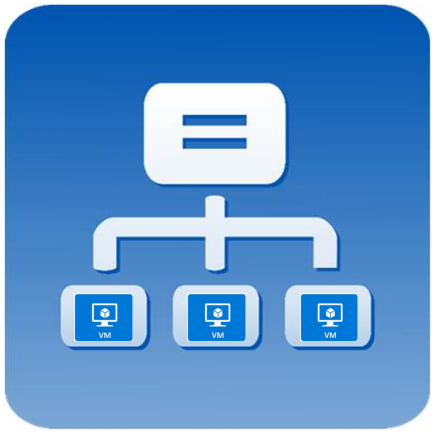
AZURE DATABRICKS CORE ARTIFACTS



Clusters

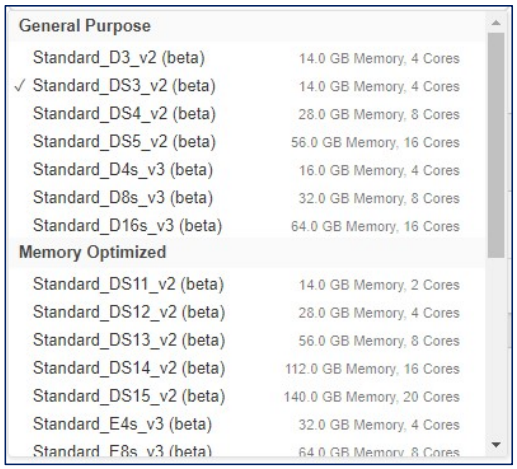
CLUSTERS

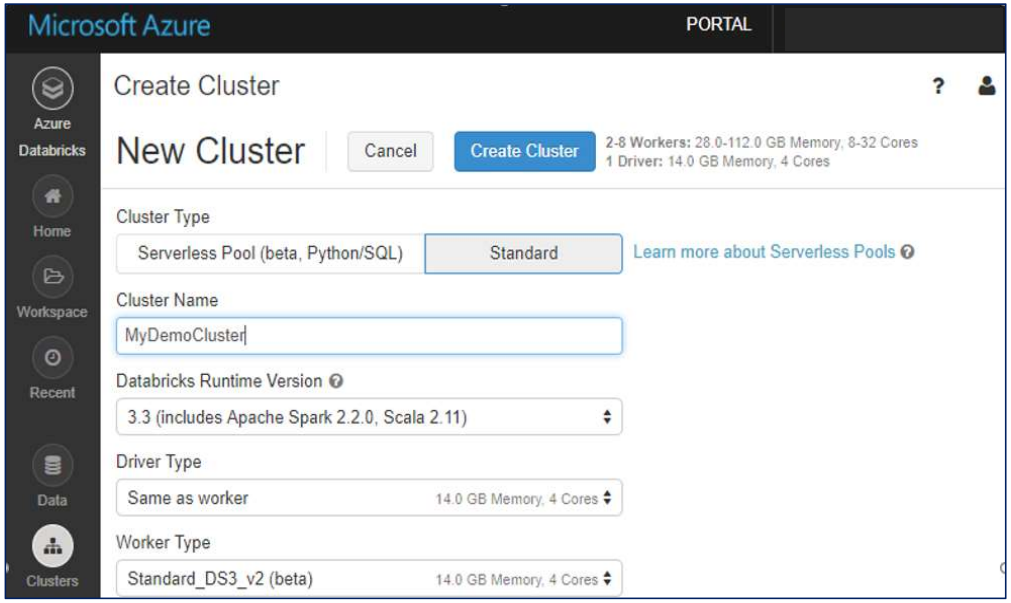
- Azure Databricks clusters are the set of Azure Linux VMs that host the Spark Worker and Driver Nodes
- Your Spark application code (i.e. Jobs) runs on the provisioned clusters.
- Azure Databricks clusters are launched in your subscription—but are managed through the Azure Databricks portal.
- Azure Databricks provides a comprehensive set of graphical wizards to manage the complete lifecycle of clusters—from creation to termination.



CLUSTER CREATION

- You can create two types of clusters – *Standard* and *Serverless Pool* (see next slide)
- While creating a cluster you can specify:
  - Number of nodes
  - Autoscaling and Auto Termination policy
  - Auto Termination policy
  - Spark Configuration details
  - The Azure VM instance types for the Driver and Worker Nodes





Graphical wizard in the Azure Databricks portal to create a Standard Cluster

CLUSTERS: AUTO SCALING AND AUTO TERMINATION

Simplifies cluster management and reduces costs by eliminating wastage

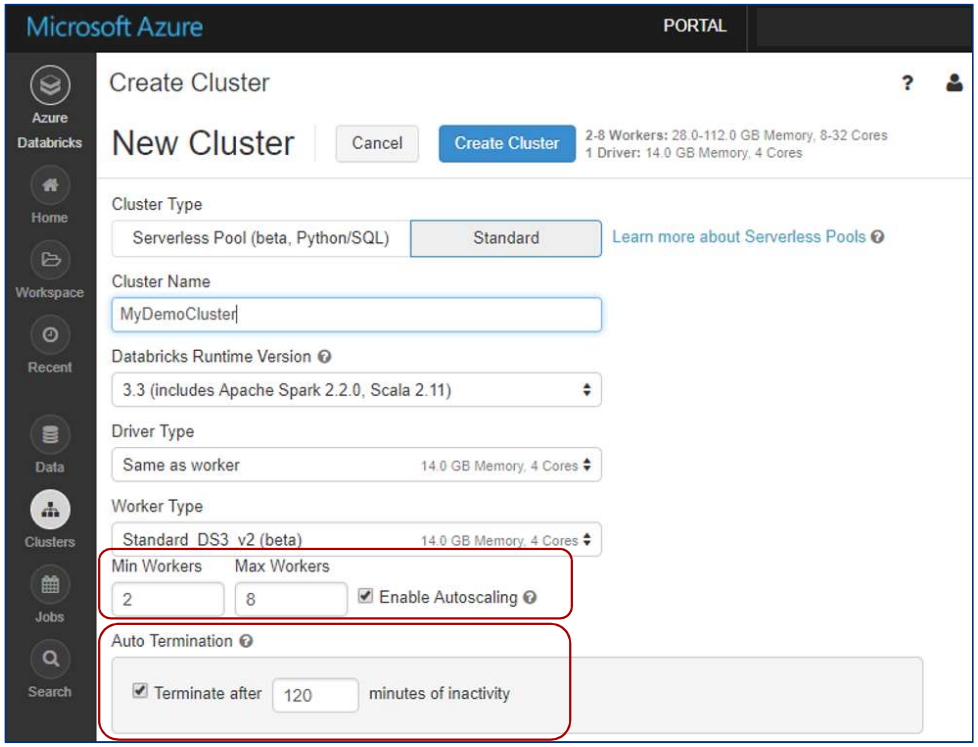
When creating Azure Databricks clusters you can choose Autoscaling and Auto Termination options.

Autoscaling: Just specify the min and max number of clusters. Azure Databricks automatically scales up or down based on load.

Auto Termination: After the specified minutes of inactivity the cluster is automatically terminated.

Benefits:

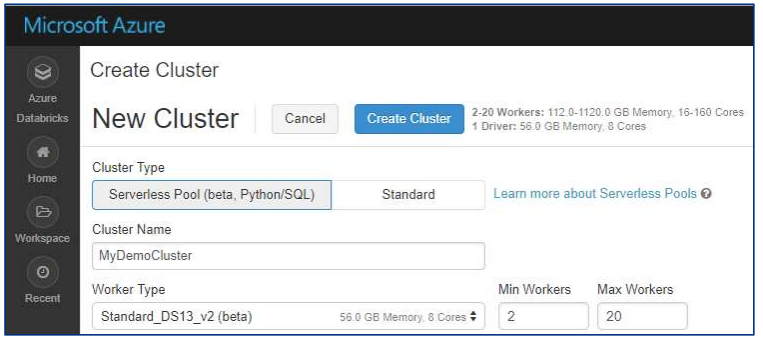
- You do not have to guess, or determine by trial and error, the correct number of nodes for the cluster
- As the workload changes you do not have to manually tweak the number of nodes
- You do not have to worry about wasting resources when the cluster is idle. You only pay for resource when they are actually being used
- You do not have to wait and watch for jobs to complete just so you can shutdown the clusters



SERVERLESS POOL (BETA)

A self-managed pool of cloud resources, auto-configured for interactive Spark workloads

- You specify only the **minimum** and **maximum** number of nodes in the cluster—Azure Databricks provisions and adjusts the compute and local storage based on your usage.
- Limitation: Currently works only for SQL and Python.



Benefits of Serverless Pool	
Auto-Configuration	▪ Databricks chooses the best configuration for Spark to get the best performance ▪ Users don't need to worry about providing any of the Databricks runtime version or Spark configuration. ▪ Databricks also chooses the best cluster parameters to save cost on infrastructure
Elasticity	▪ Automatically scales the compute and local storage, independently, based on usage
Fine grained Sharing	▪ Offers maximum resource utilization and minimum query latencies • *Preemption:* Databricks proactively preempts Spark tasks from over-committed users to ensure all users get their fair share of cluster time and their jobs complete in a timely manner even when contending with dozens of other users. Uses the "Task Preemption for High Concurrency" feature of Spark in Databricks. • *Fault isolation:* Databricks sandboxes the environments belonging to different notebooks from one another.

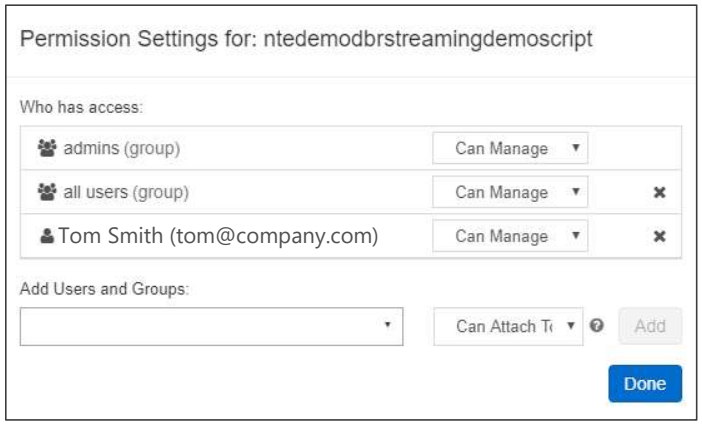
CLUSTER ACCESS CONTROL

- There are two configurable types of permissions for Cluster Access Control:
  - *Individual Cluster Permissions* - This controls a user's ability to attach notebooks to a cluster, as well as to restart/resize/terminate/start clusters.
  - *Cluster Creation Permissions* - This controls a user's ability to create clusters

- Individual permissions can be configured on the Clusters Page by clicking on Permissions under the 'More Actions' icon of an existing cluster

- There are 4 different individual cluster permission levels: *No Permissions*, *Can Attach To*, *Can Restart*, *and Can Manage*. Privileges are shown below

Abilities	No Permissions	Can Attach To	Can Restart	Can Manage
Attach notebooks to cluster		x	x	x
View Spark UI		x	x	x
View cluster metrics (Ganglia)		x	x	x
Terminate cluster			x	x
Start cluster			x	x
Restart cluster			x	x
Resize cluster				x
Modify permissions				x



Jobs

J O B S

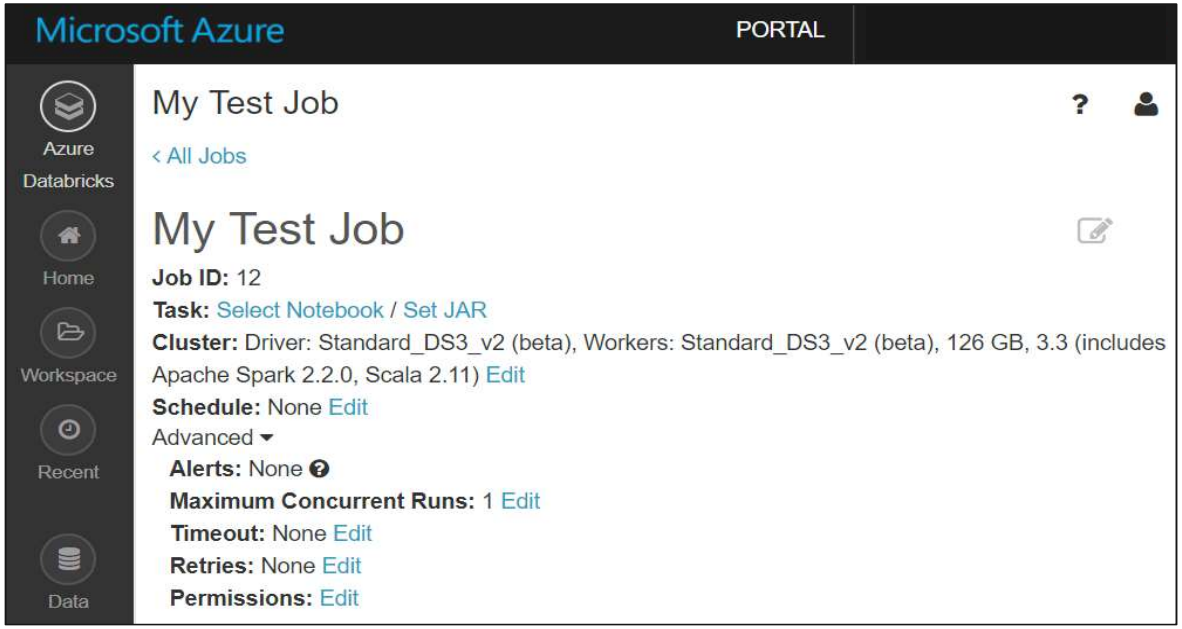
Jobs are the mechanism to submit Spark application code for execution on the Databricks clusters

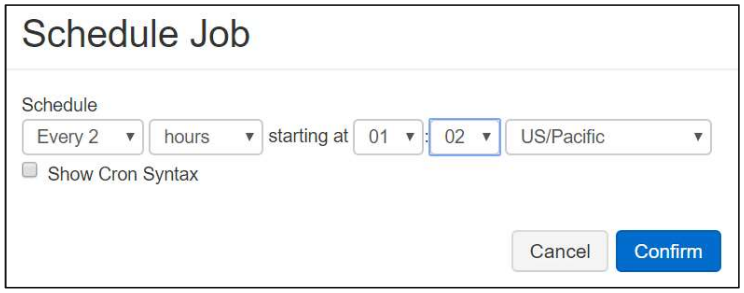
- Spark application code is submitted as a 'Job' for execution on Azure Databricks clusters
- Jobs execute either 'Notebooks' or 'Jars'
- Azure Databricks provide a comprehensive set of graphical tools to create, manage and monitor Jobs.

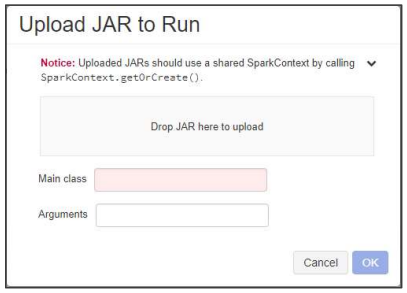
CREATING AND RUNNING JOBS (1 OF 2)

When you create a new Job you have to specify:

- The Notebook or Jar to execute
- Cluster: The cluster on which the Job execute. This could be an exiting or new cluster.
- Schedule i.e. how often the Job runs. Jobs can also be run one time right away.



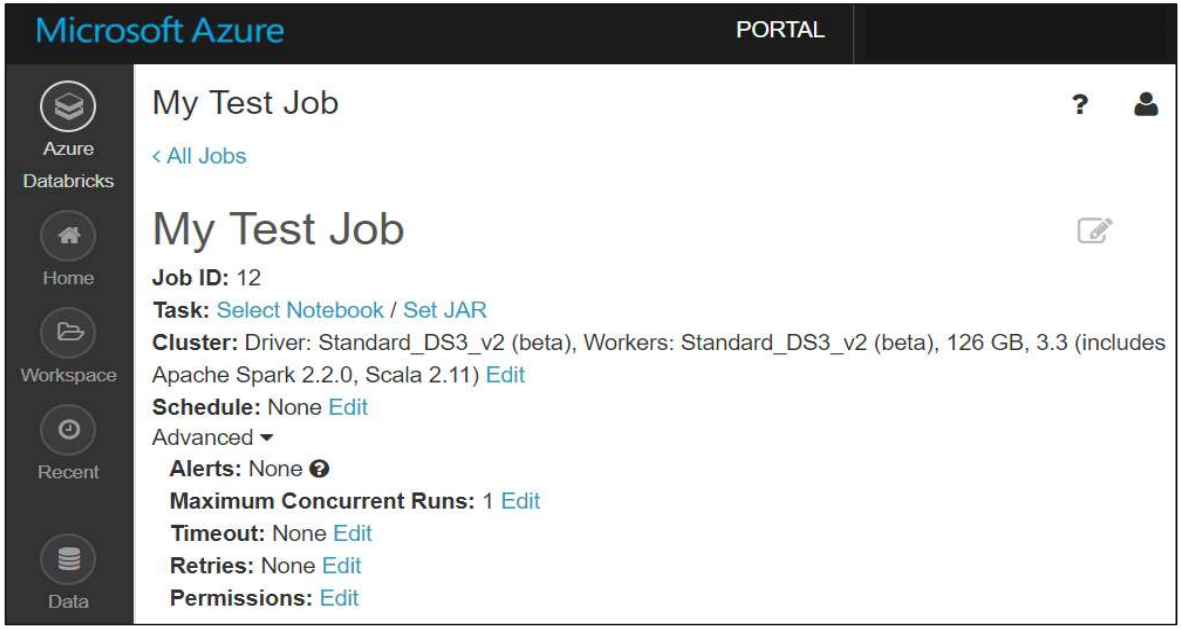


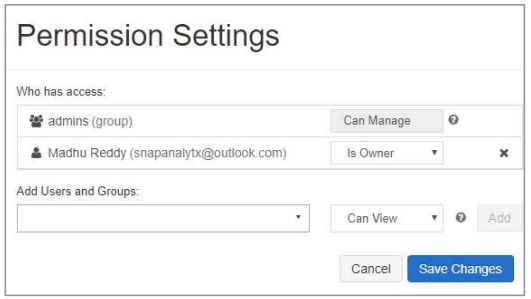


CREATING AND RUNNING JOBS (2 OF 2)

When you create a new job you can optionally specify advanced options:

- Maximum number of concurrent runs of the Job
- Timeout: Jobs still running beyond the specified duration are automatically killed
- Retry Policy: Specifies if—and when—failed jobs will be retried
- Permissions: Who can do what with jobs. This allows for Job definition and management to be *securely shared* with others (see next slide)





JOB ACCESS CONTROL

Enables job owners and administrators to grant fine grained permissions on their jobs

- With Jobs Access Controls job owners can choose which other users or groups can view results of the job.
- Owners can also choose who can manage runs of their job (i.e. invoke run now and cancel.)
- There are 5 different permission levels for jobs:
  - No Permissions
  - Can View
  - Can Manage Run
  - Is Owner and
  - Can Manage

Abilities	No Permissions	Can View	Can Manage Run	Is Owner	Can Manage (admin)
View job details and settings	Yes	Yes	Yes	Yes	Yes
View results, Spark UI, logs of a job run		Yes	Yes	Yes	Yes
Run now			Yes	Yes	Yes
Cancel run			Yes	Yes	Yes
Edit job settings				Yes	Yes
Modify permissions				Yes	Yes
Delete job				Yes	Yes
Change owner					Yes

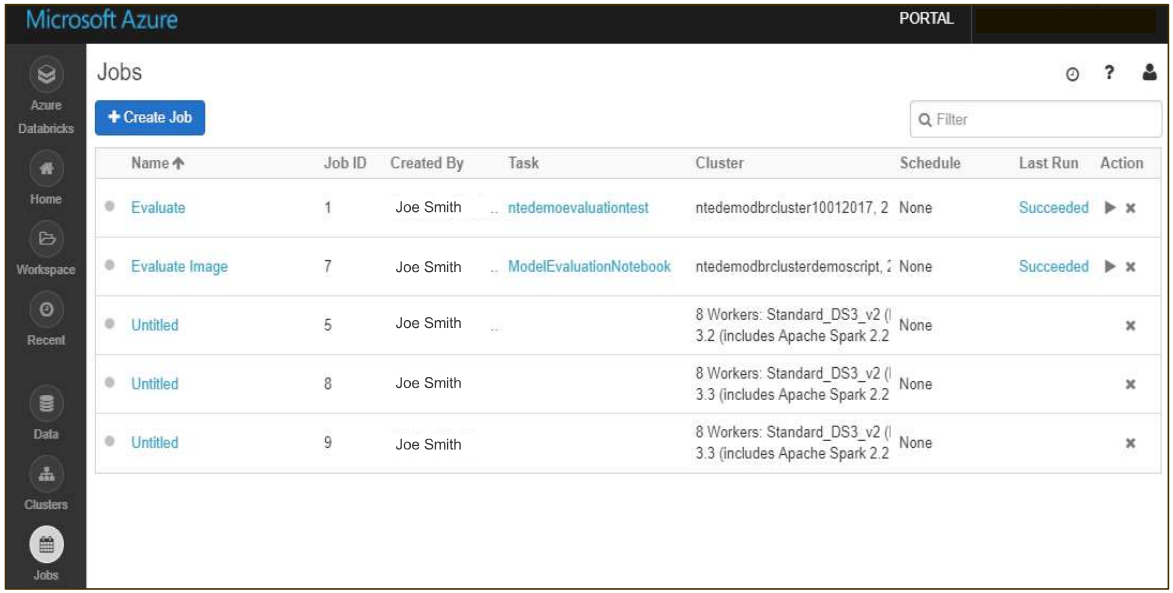
Note: 'Can Manage' permission is reserved for administrators.

VIEWING LIST OF JOBS

In the Portal you can view the list of all jobs you have access to

You can click on “Run Now” icon ▶ to run the job right away

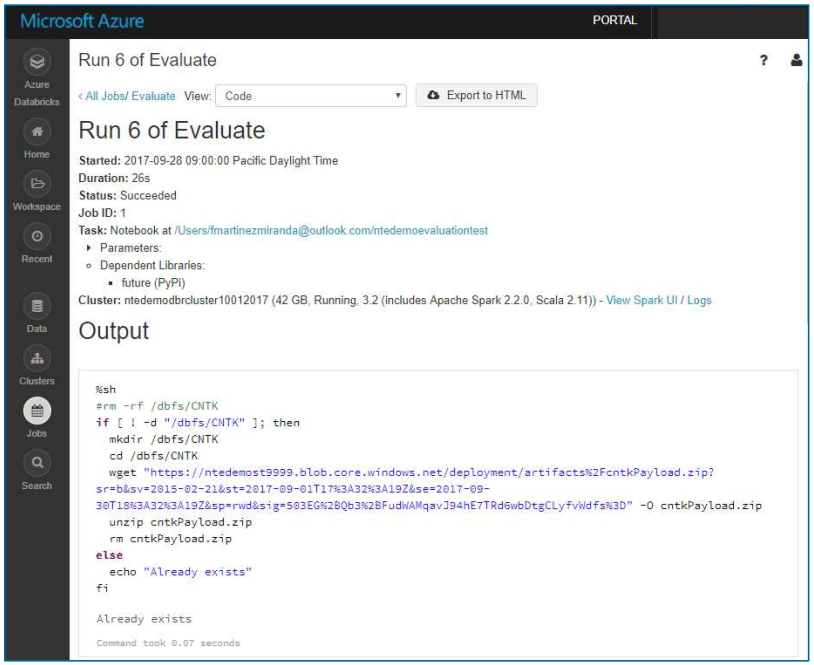
You can also delete a job from the list

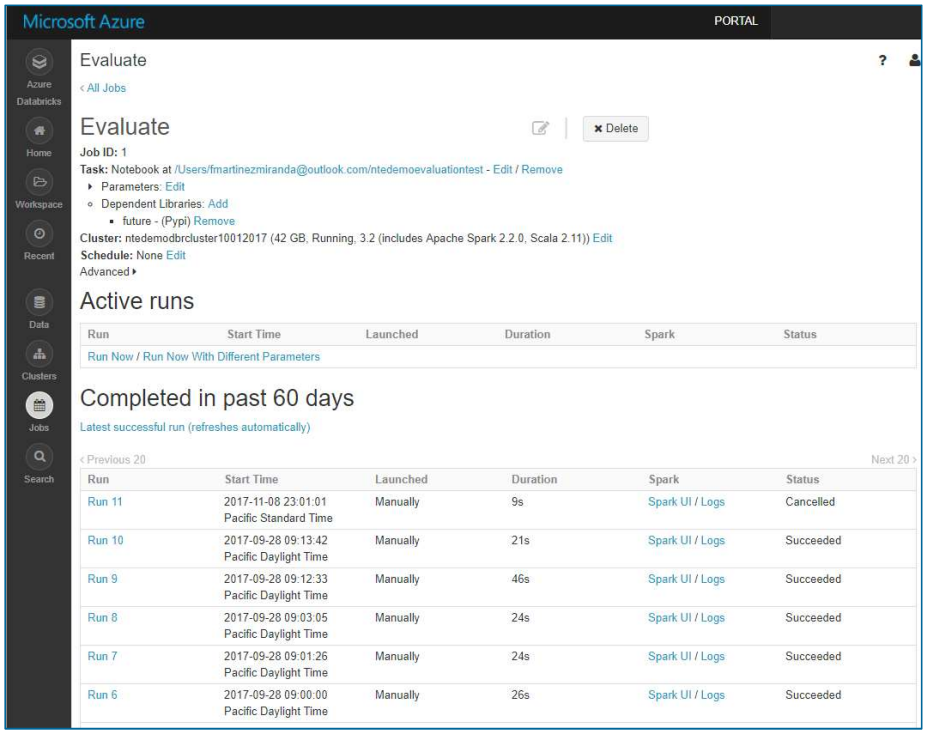


VIEWING JOBS HISTORY

In the Azure Databricks Jobs Portal you can view:

- The list of currently running (Active) Jobs
- History of old Job runs (for up to 60 days)
- The output of a particular Job run (including standard error, standard output, Spark UI logs)



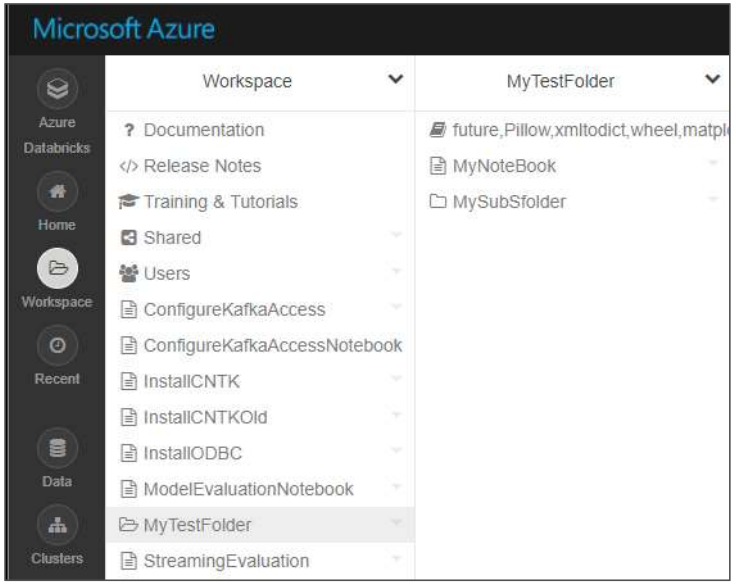


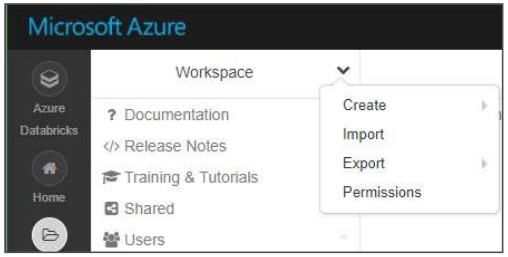
Workspaces & Directories

WORKSPACES

Workspaces enables users to organize—and share—their Notebooks, Libraries and Dashboards

- Workspaces—sort of like Directories— are a convenient way to organize an user's Notebook, Libraries and Dashboards.
- Everything in a workspace is organized into hierarchical folders. Folders can hold Libraries, Notebooks, Dashboard or more (sub) folders.
  - Icons indicate the type of the object contained in a folder
- Every user has one directory that is private and unshared.
  - By default, the workspace and all its contents are available to users.
- Fine grained access control can be defined on workspaces (next slide) to enable *secure collaboration with colleagues*.



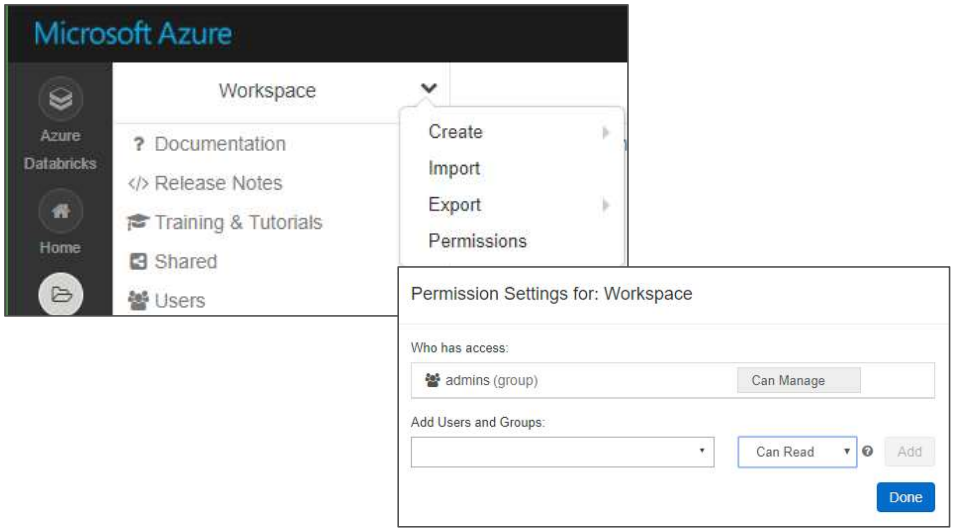


WORKSPACE OPERATIONS

You can search the entire Databricks workspace

In the Azure Databricks Portal, via the Workspaces drop down menu, you can:

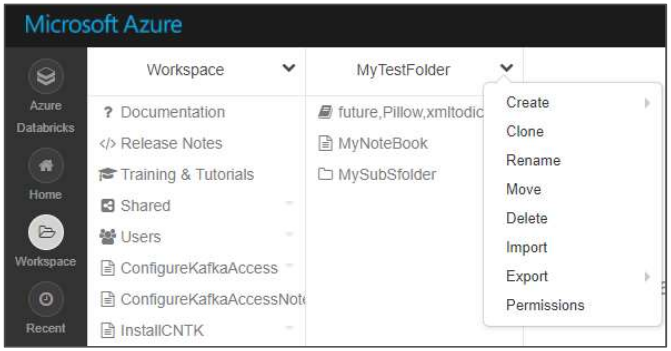
- Create Folders, Notebooks and Libraries
- Import Notebooks into the Workspace
- Export the Workspace to a database archive
- Set Permissions. You can grant 4 levels of permissions
  - Can Manage
  - Can Read
  - Can Edit
  - Can Run



FOLDER OPERATIONS AND ACCESS CONTROL

In the Azure Databricks Portal, via the Folder drop down menu, you can:

- Create Folders, Notebooks and Libraries within the folder
- Clone the folder to create a deep copy of the folder
- Rename or delete the folder
- Move the folder to another location
- Export a folder to save it and its contents as a Databricks archive
- Import a saved Databricks archive into the selected folder
- Set Permissions for the folder. As with Workspaces you can set 5 levels of permissions: *No Permissions, Can Manage, Can Read, Can Edit, Can Run*



Abilities	No Permissions	Read	Run	Edit	Manage
Create items					✅
Delete items					✅
Move/rename items					✅
Change permissions					✅

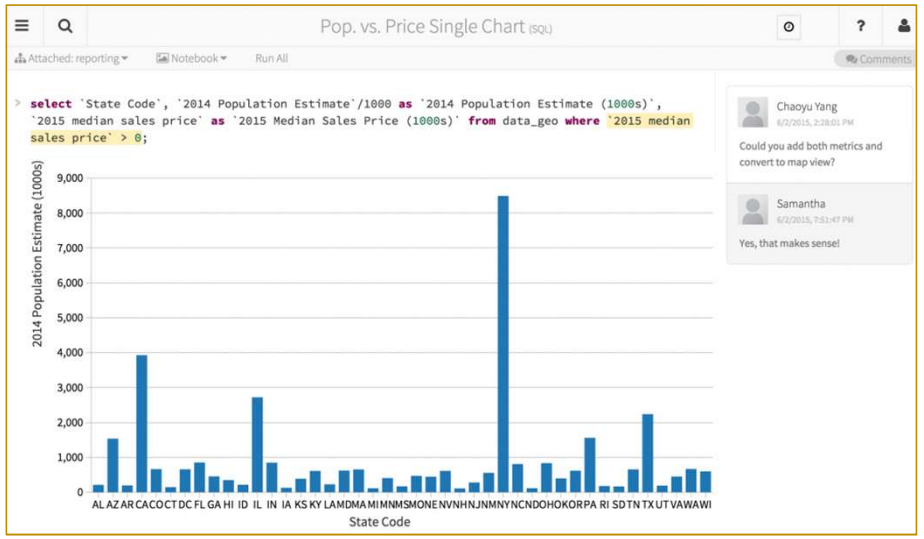
Abilities associated with each permission level

Notebooks, Libraries, Visualization

AZURE DATABRICKS NOTEBOOKS OVERVIEW

Notebooks are a popular way to develop, and run, Spark Applications

- Notebooks are not only for authoring Spark applications but can be *run/executed directly* on clusters
  - Shift+Enter
  - click the ▶ at the top right of the cell in a notebook
  - Submit via Job
- Notebooks support fine grained permissions—so they can be *securely shared* with colleagues for collaboration (see following slide for details on permissions and abilities)
- Notebooks are well-suited for prototyping, rapid development, exploration, discovery and iterative development



Notebooks typically consist of code, data, visualization, comments and notes

MIXING LANGUAGES IN NOTEBOOKS

You can mix multiple languages in the same notebook

Normally a notebook is associated with a specific language. However, with Azure Databricks notebooks, you can mix multiple languages in the same notebook. This is done using the language magic command:

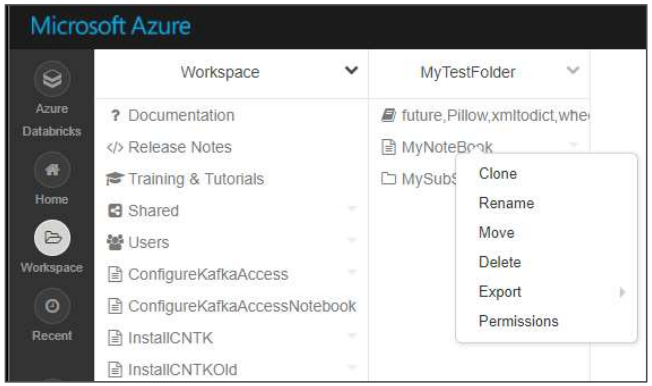
- %python Allows you to execute python code in a notebook (even if that notebook is not python)
- %sql Allows you to execute sql code in a notebook (even if that notebook is not sql).
- %r Allows you to execute r code in a notebook (even if that notebook is not r).
- %scala Allows you to execute scala code in a notebook (even if that notebook is not scala).
- %sh Allows you to execute shell code in your notebook.
- %fs Allows you to use Databricks Utilities - dbutils filesystem commands.
- %md To include rendered markdown

NOTEBOOK OPERATIONS AND ACCESS CONTROL

You can create a new notebook from the Workspace or the folder drop down menu (see previous slides)

From a notebook's drop down menu you can:

- Clone the notebook
- Rename or delete the notebook
- Move the notebook to another location
- Export a notebook to save it and its contents as a Databricks archive or IPython notebook or HTML or source code file.
- Set Permissions for the notebook As with Workspaces you can set 5 levels of permissions: *No Permissions, Can Manage, Can Read, Can Edit, Can Run*
- You can also set permissions from notebook UI itself by selecting the Permissions menu option.



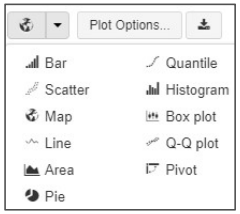
Abilities	No Permissions	Read	Run	Edit	Manage
View cells		✓	✓	✓	✓
Comment		✓	✓	✓	✓
Run Commands			✓	✓	✓
Attach/detach notebooks			✓	✓	✓
Edit cells				✓	✓
Change permissions					✓

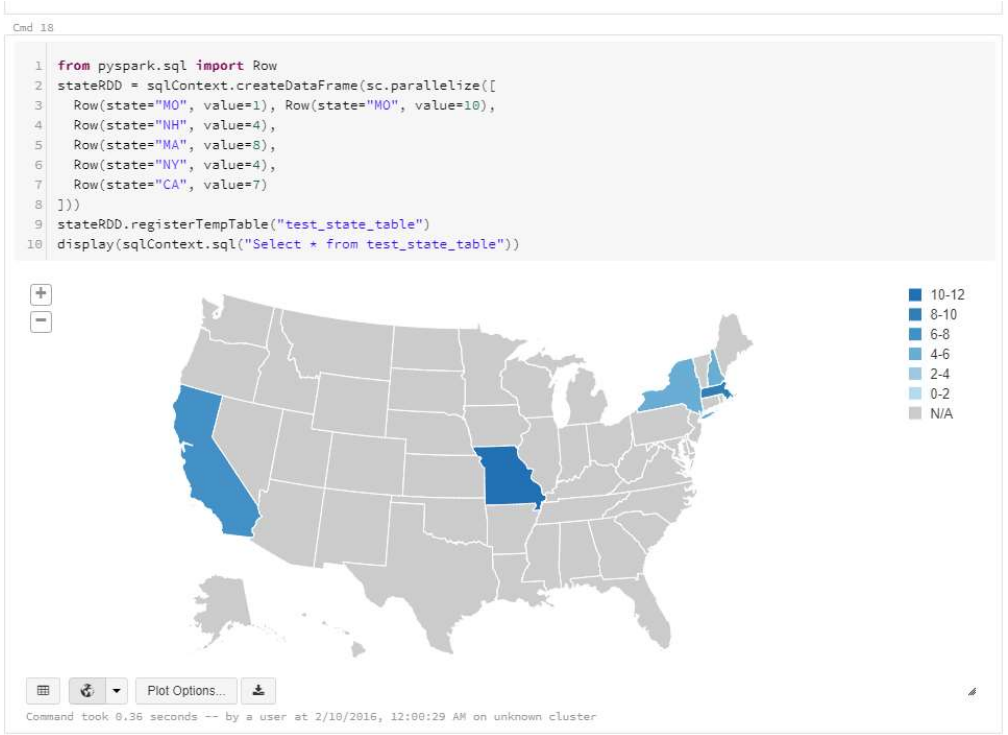
Abilities associated with each permission level

VISUALIZATION

Azure Databricks supports a number of visualization plots out of the box

- All notebooks, *regardless of their language*, support Databricks visualizations.
- When you run the notebook the visualizations are rendered inside the notebook in-place
- The visualizations are written in HTML.
  - You can save the HTML of the entire notebook by exporting to HTML.
  - If you use Matplotlib, the plots are rendered as images so you can just right click and download the image
- You can change the plot type just by picking from the selection

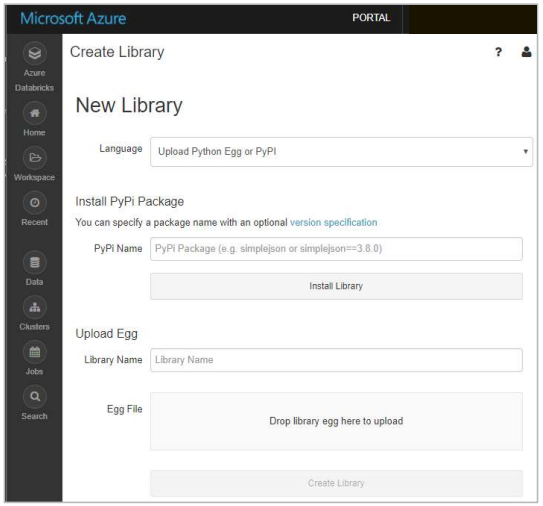


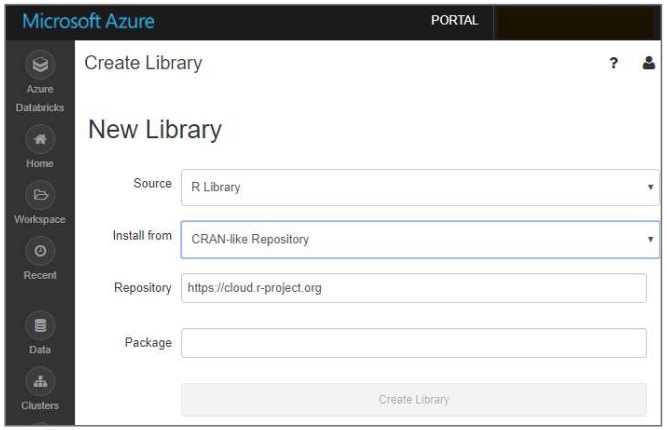


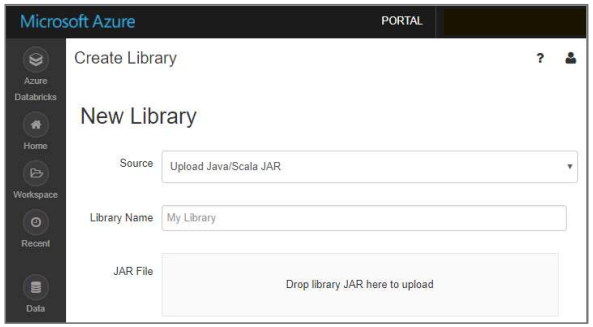
LIBRARIES OVERVIEW

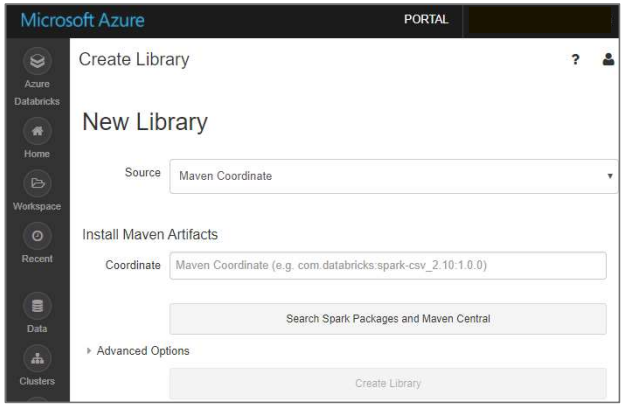
Enables external code to be imported and stored into a Workspace

- Libraries are containers to hold all your *Python, R, Java/Scala* libraries.
- Libraries resides within workspaces or folders.
- Libraries are created by importing the source code
- After importing libraries are immutable—can be deleted or overwritten only.
- You can customize installation of libraries via Init Scripts by writing custom UNIX scripts
- Libraries can also be managed via the Library API





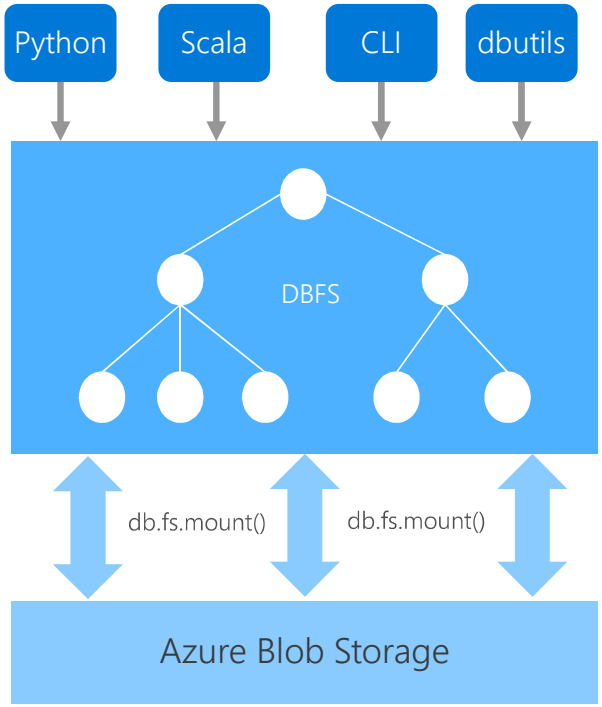




DATABRICKS FILE SYSTEM (DBFS)

Is a distributed File System (DBFS) that is a layer over Azure Blob Storage

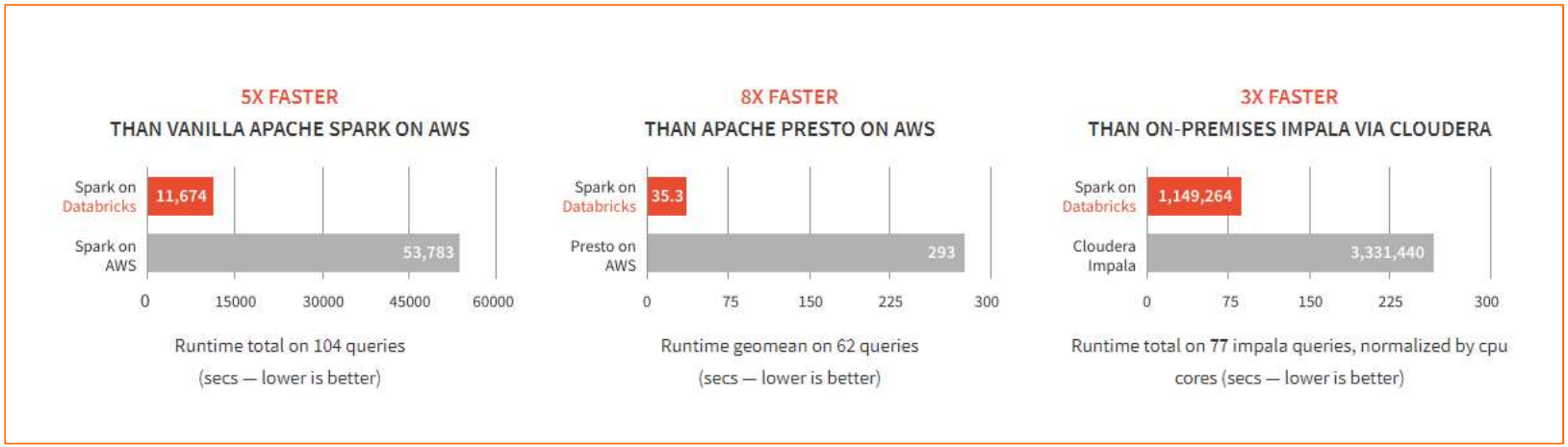
- Azure storage containers can be mounted in DBFS so that users can directly access them without specifying the storage keys
- DBFS mounts are created using *dbutils.fs.mount()*
- Azure storage data can be cached locally on the SSD of the worker nodes
- Available in both Python and Scala and accessible via a DBFS CLI
- Data persisted in Azure Blob Storage – is not lost even after cluster termination



Azure Databricks Performance

DATABRICKS SPARK IS FAST

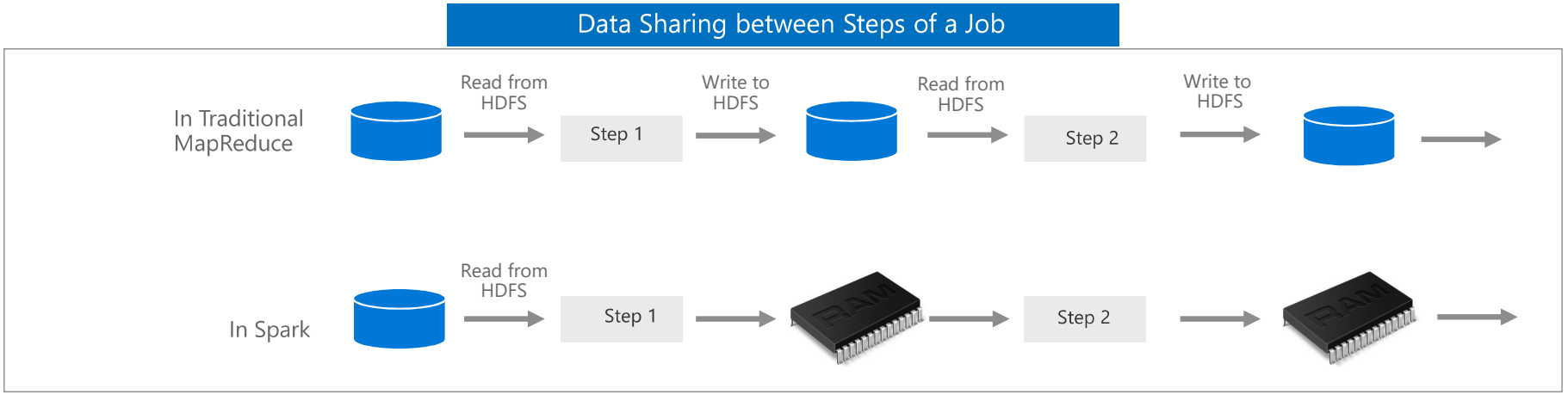
Benchmarks have shown Databricks to often have better performance than alternatives



SOURCE: Benchmarking Big Data SQL Platforms in the Cloud

WHAT MAKES SPARK FAST?(1OF 2)

- **In-memory cluster computing:** Spark provides primitives for *in-memory* cluster computing. A Spark job can *load and cache* data into memory and query it repeatedly (iteratively) much quicker than disk-based systems.
- **Scala Integration:** Spark integrates into the Scala programming language, letting you manipulate distributed datasets like local collections. No need to structure everything as map and reduce operations
- **Faster Data-sharing:** Data-sharing between operations is faster as data is in-memory:
  - In (traditional) Hadoop data is shared through HDFS which is expensive. HDFS maintains three replicas.
  - Spark stores data in-memory *without any replication*.



WHAT MAKES SPARK FAST? (2 OF 2)

Databricks IO Cache automatically caches 'remote' data on 'local nodes' to accelerate data reads

- A copy of the remote file is created in the node's local storage
  - Local data is stored in a fast intermediate format
  - Currently *Parquet* file format is supported
- Remote data is cached automatically
- Supports *DBFS, HDFS, Azure Blob Storage and Azure Data Lake store*
- DBIO Cache lets you"
  - Enable or disable caching at anytime
  - Cache only a select subset of the data
- DBIO Cache has to be configured during cluster creation. The '*max disk space per node reserved for cached data*' must be specified during cluster creation

You can Monitor the state of the DBIO cache in the Portal

Storage

Parquet IO Cache

Host	Disk Usage	Max Disk Usage Limit	Percent Disk Usage	Metadata Cache Size	Max Metadata Cache Size Limit	Percent Metadata Usage
10.0.185.226	8.3 GB	442.4 GB	1 %	6.8 MB	8.8 GB	0 %
10.0.194.201	8.2 GB	442.4 GB	1 %	6.8 MB	8.8 GB	0 %
10.0.199.229	8.2 GB	442.4 GB	1 %	6.9 MB	8.8 GB	0 %
10.0.215.147	8.1 GB	442.4 GB	1 %	7.0 MB	8.8 GB	0 %
Total	32.8 GB	1769.5 GB	1 %	27.4 MB	35.4 GB	0 %

RDDS AND DBIO CACHE - DIFFERENCES

DBIO cache and RDDs are both caches that can be used together

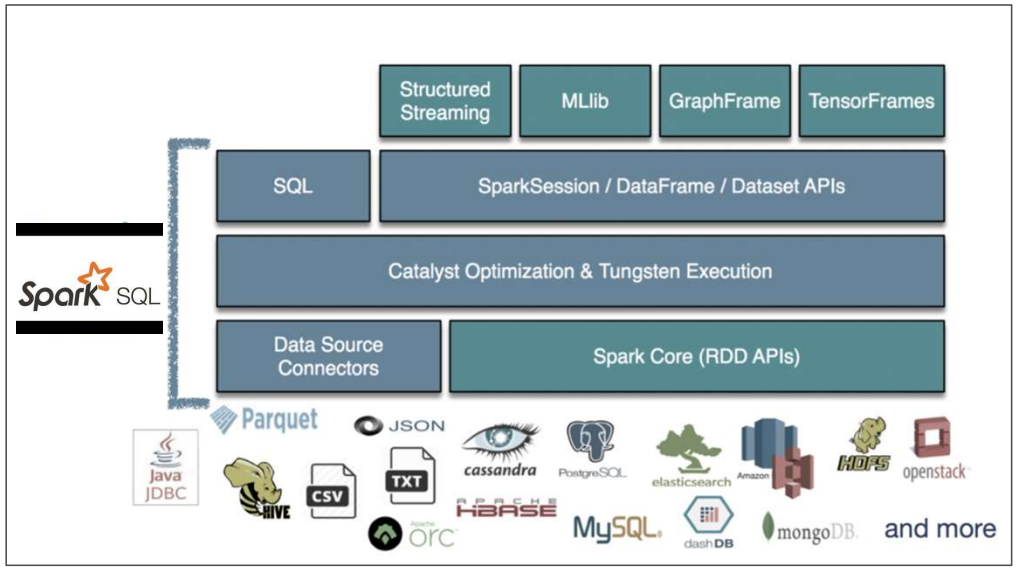
Capability	Comment
Availability	• RDD is part of Apache Spark • Databricks IO cache is available only to Databricks customers.
Type of data stored	• The RDD cache can be used to store the result of any subquery. • The DBIO cache is designed to speed-up scans by creating local copies of remote data. It can improve the performance of a wide range of queries, but cannot be used to store results of arbitrary subqueries.
Performance	• The data stored in the DBIO cache can be read and operated on faster than the data in the RDD cache. This is because the DBIO cache uses efficient decompression algorithms, and outputs data in the optimal format for further processing using whole-stage code generation.
Automatic vs manual control	• When using the RDD cache it is necessary to manually choose tables or queries to be cached. • When using the DBIO cache the data is added to the cache automatically whenever it has to be fetched from a remote source. This process is fully transparent and does not require any action from the user.
Disk vs memory-based	• Unlike the RDD cache, the DBIO cache is stored entirely on the local disk.

Data Analytics

SPARK SQL OVERVIEW

Spark SQL is a distributed SQL query engine for processing structured data

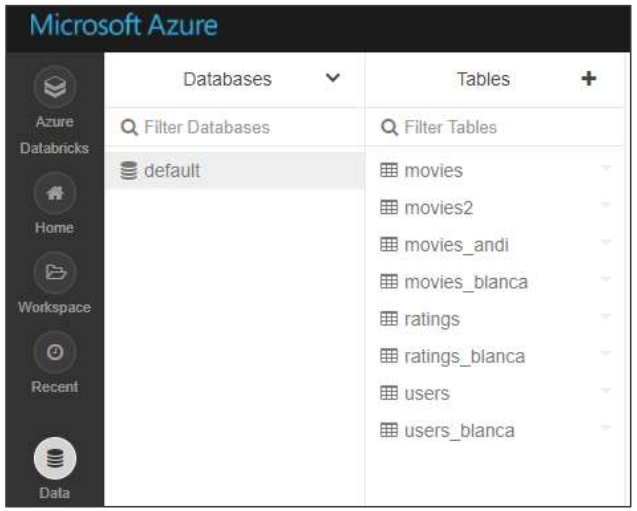
- Can query data stored in wide variety of data sources—external databases, structured data files, Hive tables and more.
- Data can be queried using either SQL or HiveQL
- Has bindings in Python, Scala and Java
- Has built-in support for structured streaming.
- Built using the Catalyst optimizer and Tungsten execution



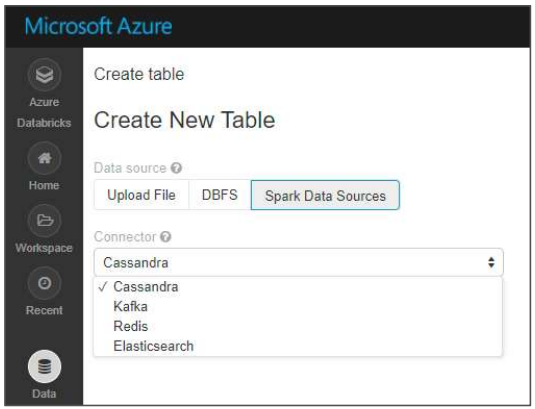
DATABASES AND TABLES OVERVIEW

Tables enable data to be structured and queried using Spark SQL or any of the Spark's language APIs

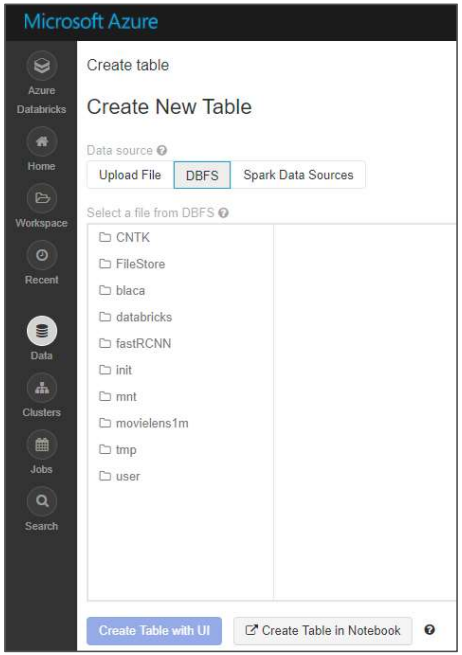
- Databases are a collection of related tables
- Tables are defined using the GUI in the console or programmatically using APIs or Notebooks
- Databricks uses the Hive metastore to manage tables, and supports all file formats and Hive data sources.
- There are multiple ways to create tables (see next slide).
- Like Apache Spark DataFrames, any Spark operation can be applied to Tables (including caching, filtering).
- Partitioned Tables and Partition Pruning: Spark SQL is able to dynamically generate partitions at the file storage level to provide partition columns for tables. When the table is scanned, Spark pushes down the filter predicates involving the partitionBy keys for partition pruning.



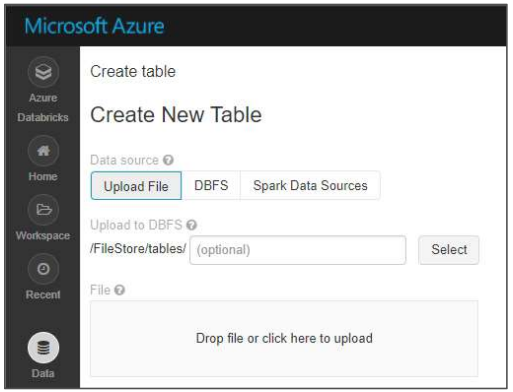
WAYS TO CREATE TABLES



From Spark Data Sources



From data in DBFS



From local files (in CSV, JSON or Avro formats)

Note: You can also create tables programmatically (CREATE TABLE tablename ...)

TABLE OPERATIONS

Azure Databricks tables support the following operations

- **Listing** database and tables
- **Viewing** table details including its schema and sample data
- **Reading** from tables
- **Updating** tables: Table schema is immutable. However, a user can update table data by changing the underlying files.
- **Deleting** tables: A user can delete tables either through the UI or programmatically

From SQL:

```
SELECT * FROM diamonds
```

From Python, use one of these examples:

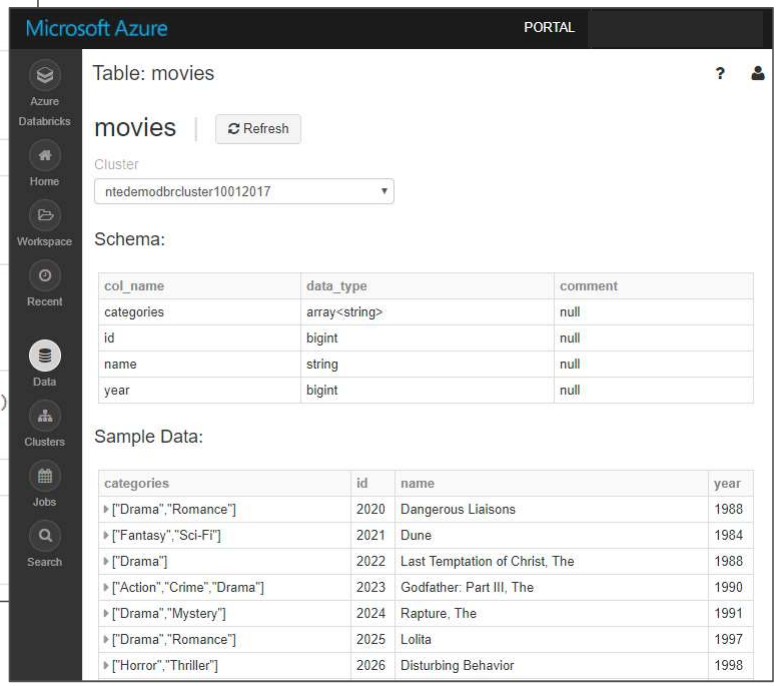
```
diamonds = spark.sql("SELECT * FROM diamonds")
display(diamonds.select("*"))
```

```
diamonds = spark.table("diamonds")
display(diamonds.select("*"))
```

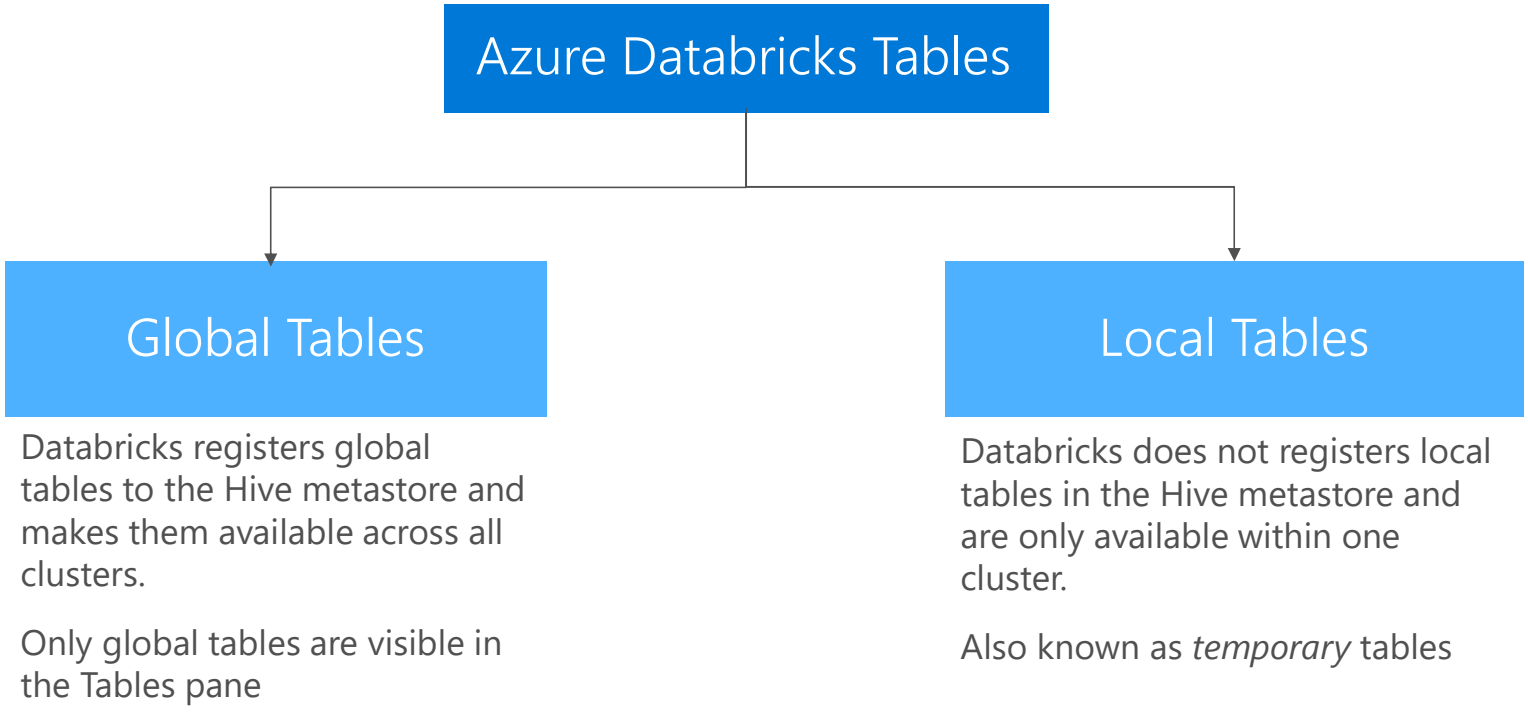
From Scala, use one of these examples:

```
val diamonds = spark.sql("SELECT * FROM diamonds")
display(diamonds.select("*"))
```

```
val diamonds = spark.table("diamonds")
display(diamonds.select("*"))
```



LOCAL AND GLOBAL TABLES

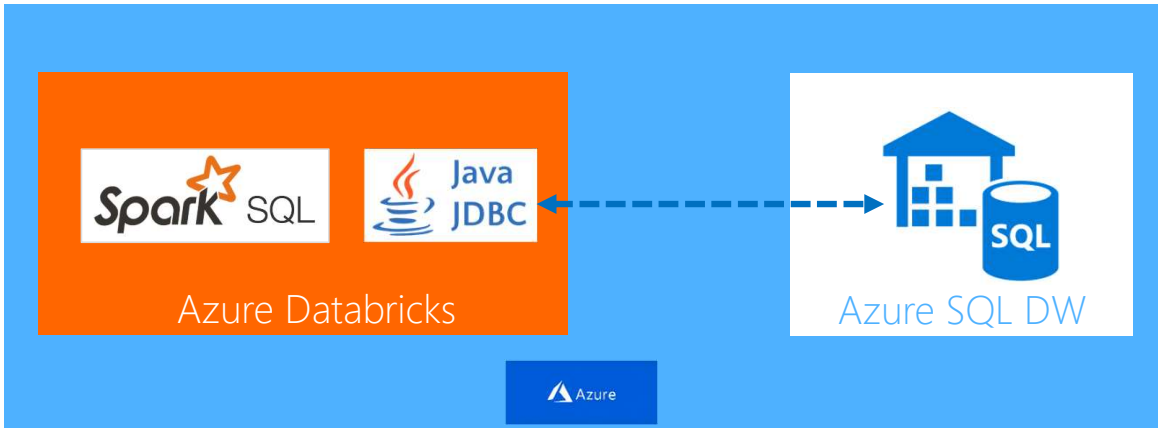


AZURE SQL DW INTEGRATION

Integration enables structured data from SQL DW to be included in Spark Analytics

Azure SQL Data Warehouse is a SQL-based fully managed, petabyte-scale cloud solution for data warehousing

- You can bring in data from Azure SQL DW to perform advanced analytics that require both structured and unstructured data.
- Currently you can access data in Azure SQL DW via the JDBC driver. From within your spark code you can access just like any other JDBC data source.
- If Azure SQL DW is authenticated via AAD then Azure Databricks user can seamlessly access Azure SQL DW.



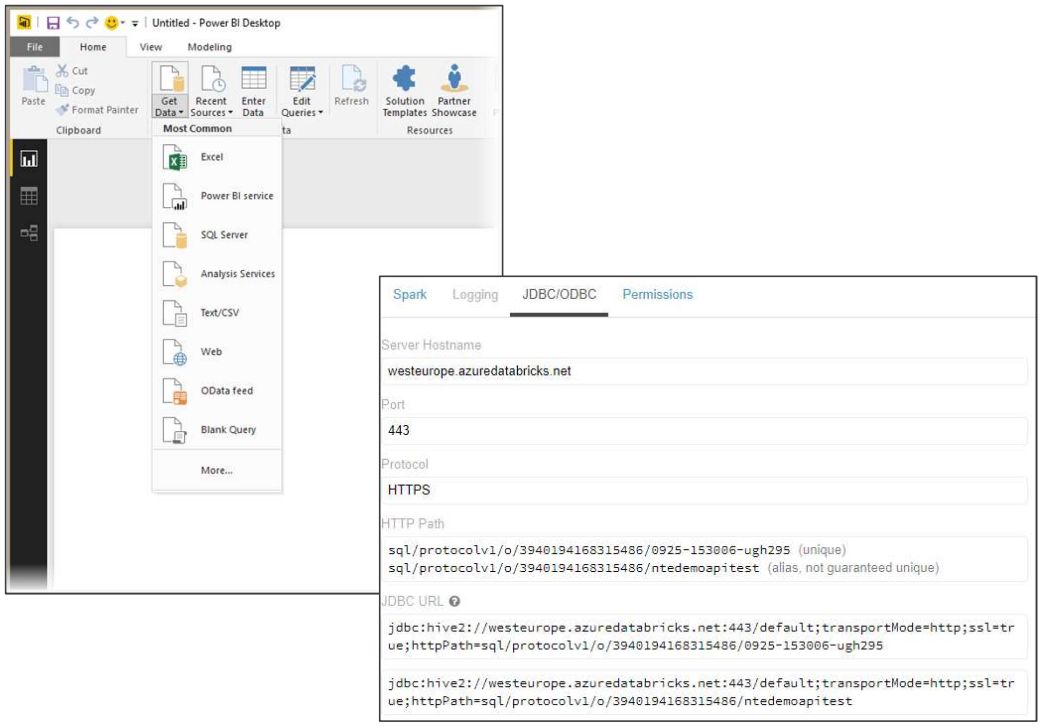
POWER BI INTEGRATION

Enables powerful visualization of data in Spark with Power BI

Power BI is a business analytics tool that provides data Visualization, Report and Dashboard throughout an organization

Power BI Desktop can connect to Azure Databricks clusters to query data using JDBC/ODBC server that runs on the driver node.

- This server listens on port 10000 and it is not accessible outside the subnet where the cluster is running.
- Azure Databricks uses a public HTTPS gateway
- The JDBC/ODBC connection information can be obtained from the Cluster UI directly as shown in the figure.
- When establishing the connection, you can use a Personal Access Token to authenticate to the cluster gateway. Only users who have attach permissions can access the cluster via the JDBC/ ODBC endpoint.
- In Power BI desktop you can setup the connection by choosing the ODBC data source in the "Get Data" option.



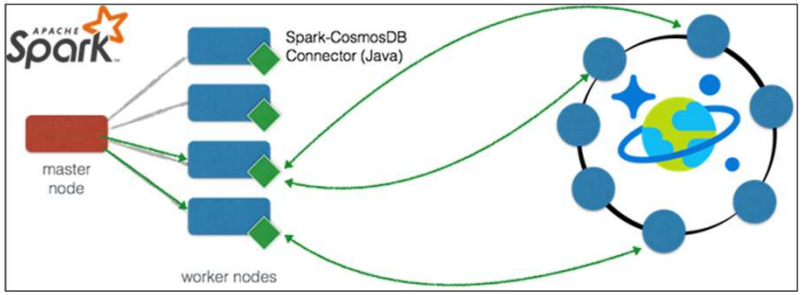
COSMOS DB INTEGRATION

The Spark connector enables real-time analytics over globally distributed data in Azure Cosmos DB

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service for mission-critical applications

- With Spark connector for Azure Cosmos DB, Apache Spark can now interact with all Azure Cosmos DB data models: *Documents, Tables, and Graphs*.
  - efficiently exploits the native Azure Cosmos DB managed indexes and enables updateable columns when performing analytics.
  - utilizes push-down predicate filtering against fast-changing globally-distributed data
- Some use-cases for Azure Cosmos DB + Spark include:
  - Streaming Extract, Transformation, and Loading of data (ETL)
  - Data enrichment
  - Trigger event detection
  - Complex session analysis and personalization
  - Visual data exploration and interactive analysis
  - Notebook experience for data exploration, information sharing, and collaboration

The connector uses the Azure DocumentDB Java SDK and moves data directly between Spark worker nodes and Cosmos DB data nodes



AZURE BLOB STORAGE INTEGRATION

Data can be read from Azure Blob Storage using the Hadoop FileSystem interface. Data can be read from public storage accounts without any additional settings. To read data from a private storage account, you need to set an account key or a Shared Access Signature (SAS) in your notebook

Setting up an account key

```
spark.conf.set ( "fs.azure.account.key.{Your Storage Account Name}.blob.core.windows.net", "{Your Storage Account Access Key}")
```

Setting up a SAS for a given container:

```
spark.conf.set( "fs.azure.sas.{Your Container Name}.{Your Storage Account Name}.blob.core.windows.net", "{Your SAS For The Given Container}")
```

Once an account key or a SAS is setup, you can use standard Spark and Databricks APIs to read from the storage account:

```
val df = spark.read.parquet("wasbs://{Your Container Name}@m{Your Storage Account  name}.blob.core.windows.net/{Your Directory Name}")
dbutils.fs.ls("wasbs://{Your ntainer Name}@{Your Storage Account Name}.blob.core.windows.net/{Your Directory Name}")
```

AZURE DATA LAKE INTEGRATION

To read from your Data Lake Store account, you can configure Spark to use service credentials with the following snippet in your notebook

```
spark.conf.set("dfs.adls.oauth2.access.token.provider.type", "ClientCredential")
spark.conf.set("dfs.adls.oauth2.client.id", "{YOUR SERVICE CLIENT ID}")
spark.conf.set("dfs.adls.oauth2.credential", "{YOUR SERVICE CREDENTIALS}")
spark.conf.set("dfs.adls.oauth2.refresh.url", "https://login.windows.net/{YOUR DIRECTORY ID}/oauth2/token")
```

After providing credentials, you can read from Data Lake Store using standard APIs:

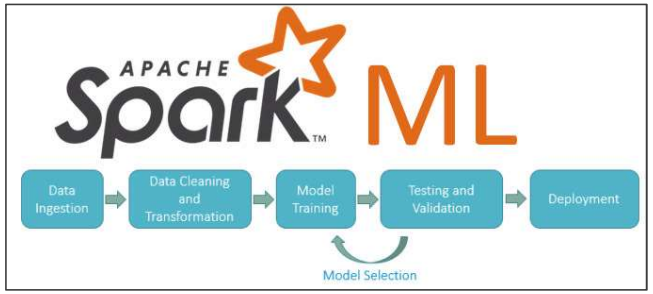
```
val df = spark.read.parquet("adl://{YOUR DATA LAKE STORE ACCOUNT NAME}.azuredatalakestore.net/{YOUR DIRECTORY NAME}")
dbutils.fs.list("adl://{YOUR DATA LAKE STORE ACCOUNT NAME}.azuredatalakestore.net/{YOUR DIRECTORY NAME}")
```

Machine Learning and Deep Learning

SPARK MACHINE LEARNING(ML) OVERVIEW

Enables Parallel, Distributed ML for large datasets on Spark Clusters

- Offers a set of parallelized machine learning algorithms (see next slide)
- Supports Model Selection (hyperparameter tuning) using Cross Validation and Train-Validation Split.
- Supports Java, Scala or Python apps using DataFrame-based API (as of Spark 2.0). Benefits include:
  - An uniform API across ML algorithms and across multiple languages
  - Facilitates ML pipelines (enables combining multiple algorithms into a single pipeline).
  - Optimizations through Tungsten and Catalyst
- Spark MLlib comes pre-installed on Azure Databricks
- 3rd Party libraries supported include: H20 Sparkling Water, SciKit-learn and XGBoost

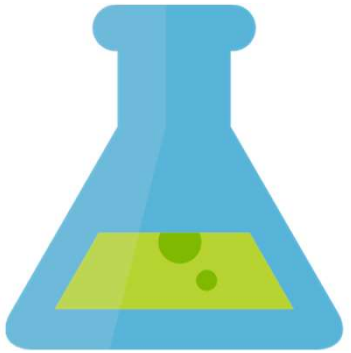


MMLSPARK

Microsoft Machine Learning Library for Apache Spark (MMLSpark) lets you easily create scalable machine learning models for large datasets.

It includes integration of SparkML pipelines with the Microsoft Cognitive Toolkit and OpenCV, enabling you to:

- Ingress and pre-process image data
- Featurize images and text using pre-trained deep learning models
- Train and score classification and regression models using implicit featurization



SPARK ML ALGORITHMS

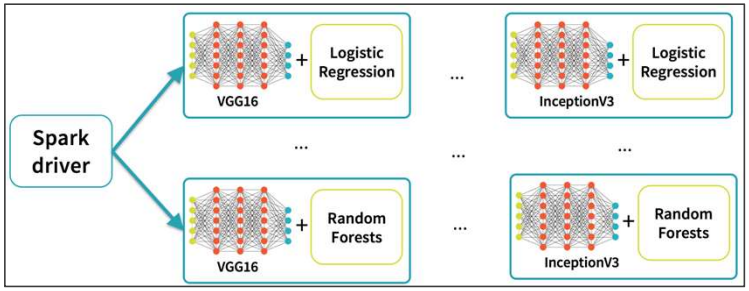
Spark ML Algorithms

Classification and Regression	• Linear Models (SVMs, logistic regression, linear regression) • Naïve Bayes • Decision Trees • Ensembles of trees (Random Forest, Gradient-Boosted Trees) • Isotonic regression
Clustering	• k-means and streaming k-means • Gaussian mixture • Power iteration clustering (PIC) • Latent Dirichlet allocation (LDA)
Collaborative Filtering	• Alternating least squares (ALS)
Dimensionality Reduction	• SVD • PCA
Frequent Pattern Mining	• FP-growth • Association rules
Basic Statistics	• Summary statistics • Correlations • Stratified sampling • Hypothesis testing • Random data generation

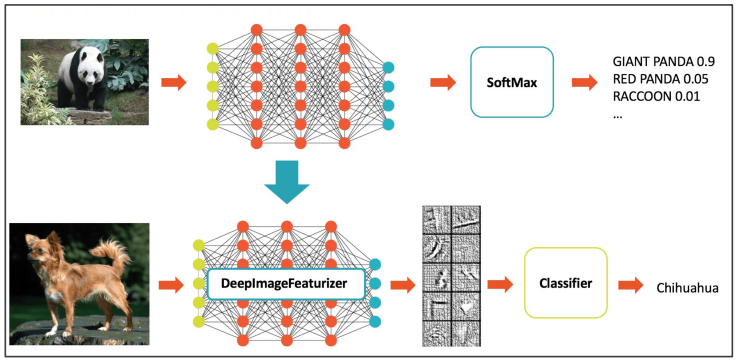
DEEP LEARNING

Azure Databricks supports and integrates with a number of Deep Learning libraries and frameworks to make it easy to build and deploy Deep Learning applications

- Supports Deep Learning Libraries/frameworks including:
  - Microsoft Cognitive Toolkit (CNTK).
    - Article explains how to install CNTK on Azure Databricks.
  - TensorFlowOnSpark
  - BigDL
- Offers Spark Deep Learning Pipelines, a suite of tools for working with and processing images using deep learning using transfer learning. It includes high-level APIs for common aspects of deep learning so they can be done efficiently in a few lines of code:
  - Image loading
  - Applying pre-trained models as transformers in a Spark ML pipeline
  - Transfer learning
  - Distributed hyperparameter tuning
  - Deploying models in DataFrames and SQL



Distributed Hyperparameter Tuning

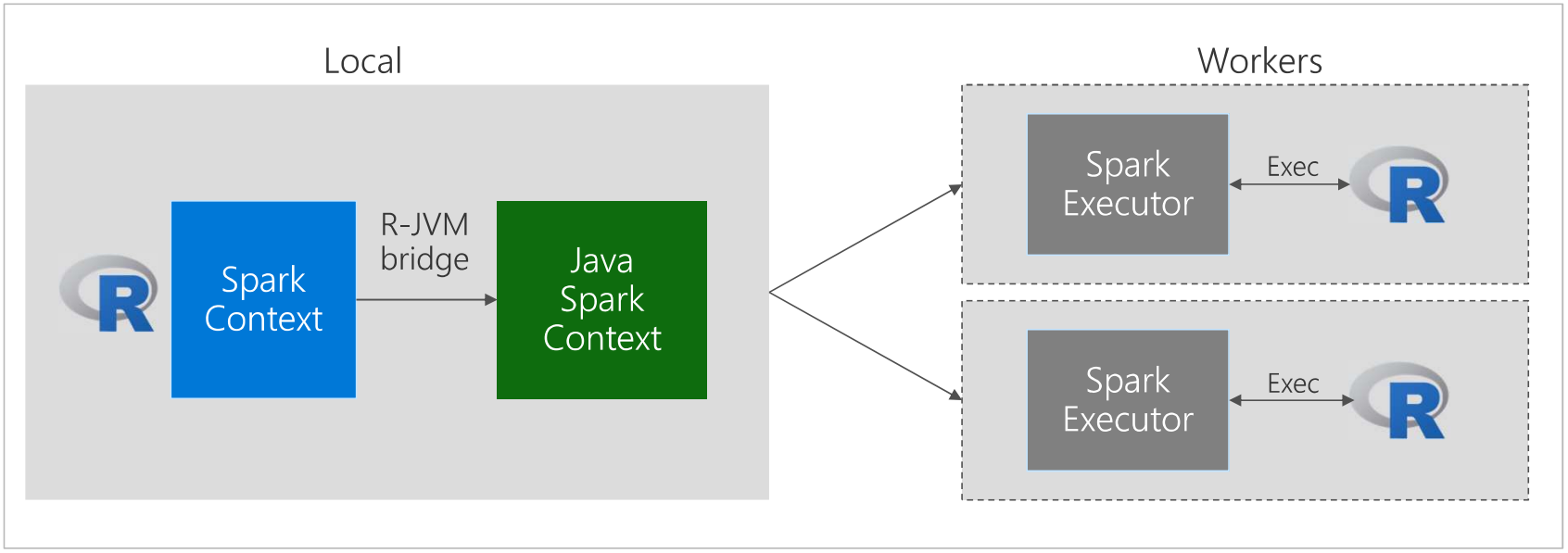


Transfer Learning

SPARKR OVERVIEW

An R package that provides a light-weight frontend to use Apache Spark from R

- Provides a distributed DataFrame implementation that supports operations like selection, filtering, aggregation etc (similar to R data frames, dplyr)
- Supports distributed machine learning using Spark MLlib.
- R programs can connect to a Spark cluster from RStudio, R shell, Rscript or other R IDEs.

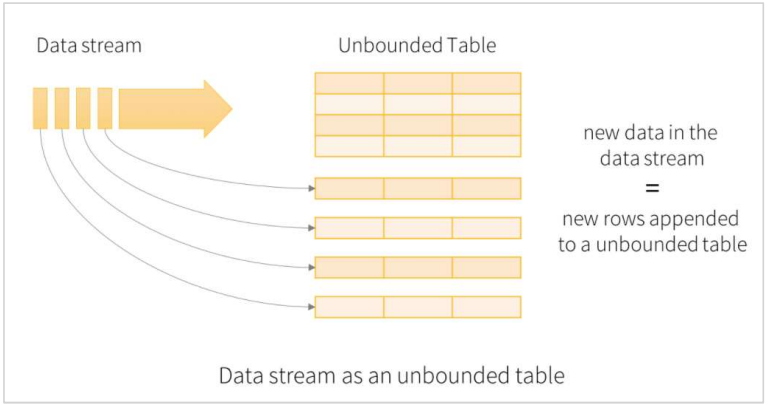


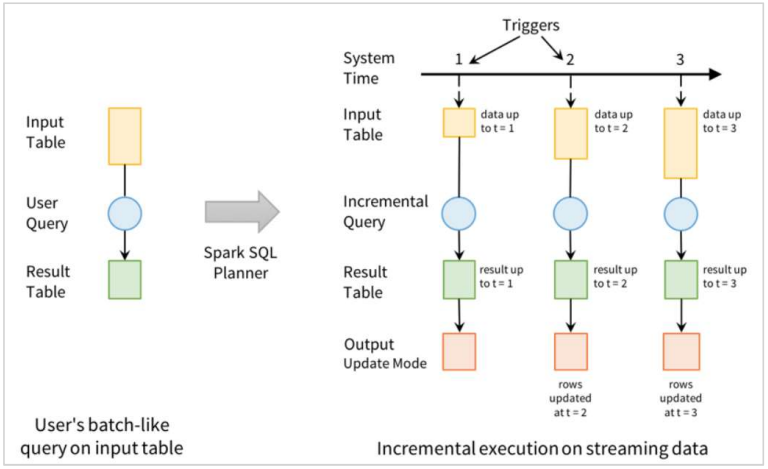
Stream Analytics & Graph Processing

SPARK STRUCTURED STREAMING OVERVIEW

A unified system for end-to-end fault-tolerant, exactly-once stateful stream processing

- Unifies streaming, interactive and batch queries—a single API for both static bounded data and streaming unbounded data.
- Runs on Spark SQL. Uses the Spark SQL Dataset/DataFrame API used for batch processing of static data.
- Runs incrementally and continuously and updates the results as data streams in.
- Supports app development in Scala, Java, Python and R.
- Supports streaming aggregations, event-time windows, windowed grouped aggregation, stream-to-batch joins.
- Features streaming deduplication, multiple output modes and APIs for managing/monitoring streaming queries.
- Built-in sources: Kafka, File source (json, csv, text, parquet)

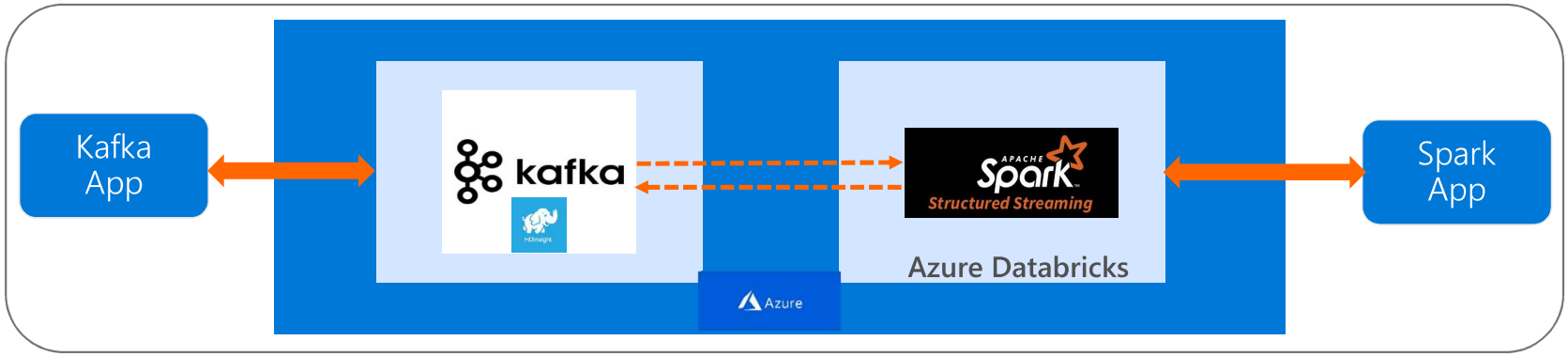




APACHE KAFKA FOR HDINSIGHT INTEGRATION

Azure Databricks Structured Streaming integrates with Apache Kafka for HDInsight

- Apache Kafka for Azure HDInsight is an enterprise grade streaming ingestion service running in Azure.
- Azure Databricks Structured Streaming applications can use Apache Kafka for HDInsight as a data source or sink.
- No additional software (gateways or connectors) are required.
- Setup: Apache Kafka on HDInsight does not provide access to the Kafka brokers over the public internet. So the Kafka clusters and the Azure Databricks cluster must be located in the same Azure Virtual Network.



Note: Azure Databricks Structured Streaming integration with **Azure Event Hubs** is forthcoming

SPARK GRAPHX OVERVIEW

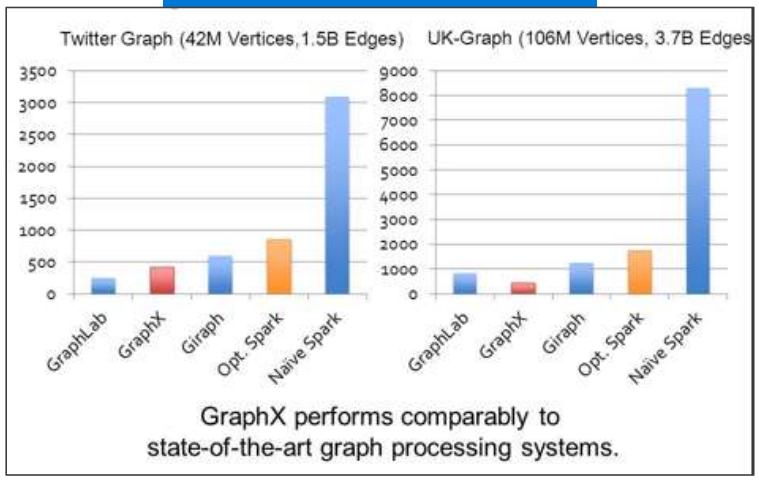
A set of APIs for graph and graph-parallel computation.

- Unifies ETL, exploratory analysis, and iterative graph computation within a single system.
- Developers can:
  - view the same data as both graphs and collections,
  - transform and join graphs with RDDs, and
  - write custom iterative graph algorithms using the Pregel API.
- Currently only supports using the Scala and RDD APIs.

Algorithms

- PageRank
- Connected components
- Label propagation
- SVD++
- Strongly connected components
- Triangle count

PageRank Benchmark

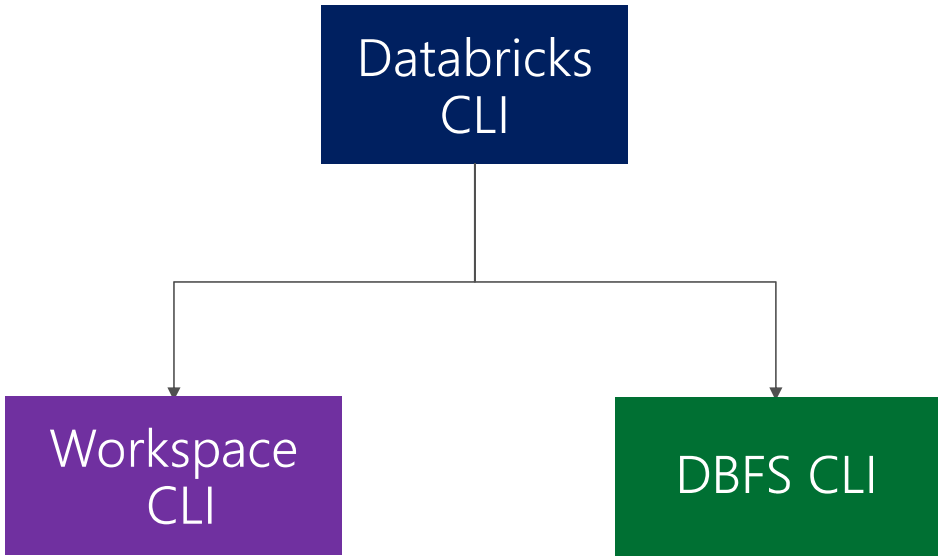


Source: AMPLab

CLI and REST APIs

DATABRICKS CLI

An easy to use interface built on top of the Databricks REST API



Currently, the CLI fully implements the DBFS API and the Workspace API

DATABRICKS WORKSPACE CLI

Databricks Workspace CLI Commands

delete	Deletes objects from the Databricks...
export	Exports a file from the Databricks workspace...
export_dir	Recursively exports a directory from the...
import	Imports a file from local to the Databricks...
import-dir	Recursively imports a directory from local to...
list / ls	List objects in the Databricks Workspace
mkdirs	Make directories in the Databricks Workspace
rm	Deletes objects from the Databricks...

Workspace CLI Example

```
$ Databricks workspace ls /Users/example@Databricks.com/example -l

NOTEBOOK   a  PYTHON
NOTEBOOK   b  SCALA
NOTEBOOK   c  SQL
NOTEBOOK   d  R
DIRECTORY  e
```

D B F S C L I

Leverages the DBFS API to provide an easy Command Line Interface to DBFS

DBFS CLI Commands:

cp	Copy files to and from DBFS.
ls	List files in DBFS.
mkdirs	Make directories in DBFS.
mv	Moves a file between two DBFS paths.
rm	Remove files from dbfs.

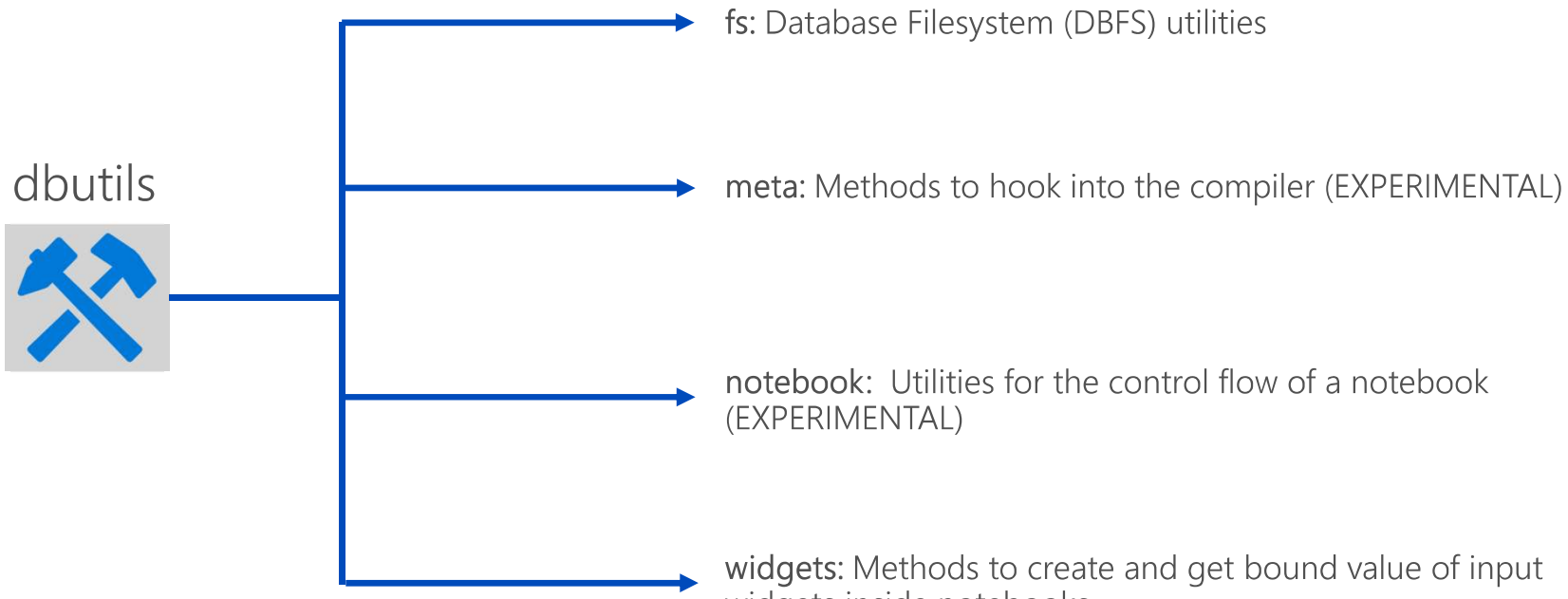
DBFS CLI examples

```
# List files in DBFS
dbfs ls

# Put local file ./foo.txt to dbfs:/foo.txt
dbfs cp ./foo.txt dbfs:/foo.txt

# Get dbfs:/foo.txt and save to local file ./foo.txt
dbfs cp dbfs:/foo.txt ./foo.txt

# Recursively put local dir ./foo to dbfs:/foo
dbfs cp -r ./foo dbfs:/foo
```

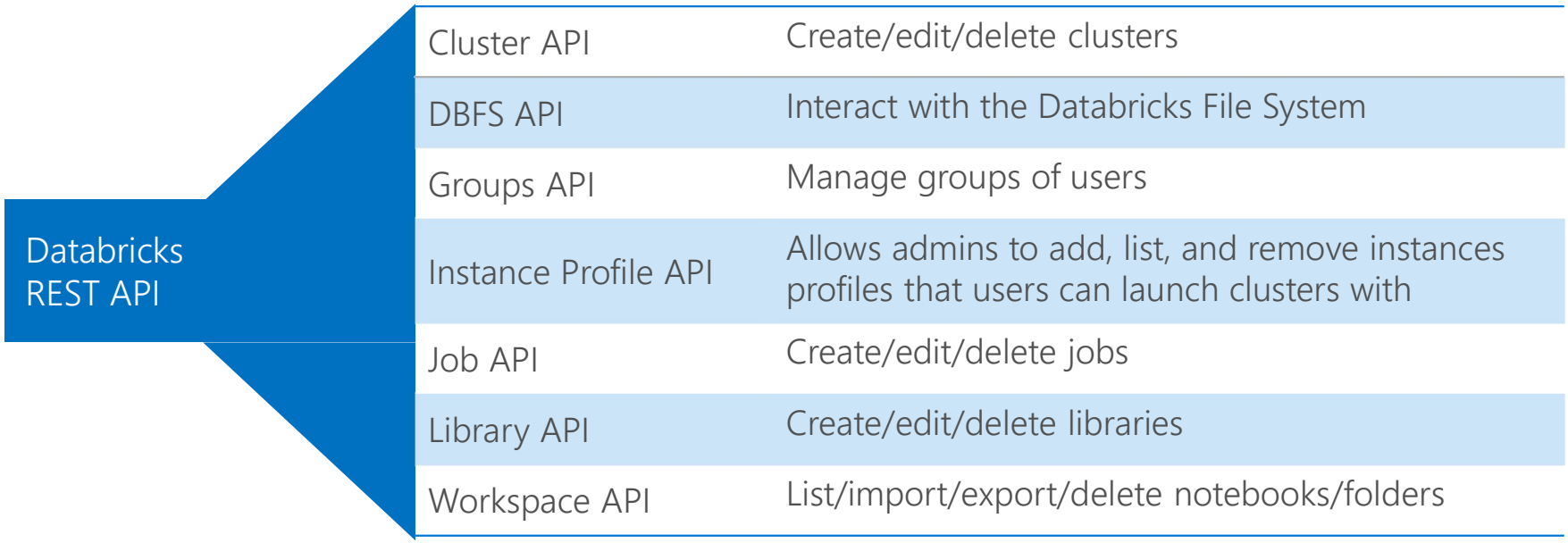
Note: All dbfs paths should be prefixed with dbfs://

DATABRICKS UTILITIES (DBUTILS)

Set of tools that make it easy to perform combinations of tasks



DATABRICKS REST API



Cluster API	Create/edit/delete clusters
DBFS API	Interact with the Databricks File System
Groups API	Manage groups of users
Instance Profile API	Allows admins to add, list, and remove instances profiles that users can launch clusters with
Job API	Create/edit/delete jobs
Library API	Create/edit/delete libraries
Workspace API	List/import/export/delete notebooks/folders

DATABRICKS API - AUTHENTICATION

Personal access tokens or passwords can be used to authenticate and access Databricks REST APIs

- Tokens can be *generated* and *revoked* from the Databricks Portal Token Management Page.
- Tokens have an expiration time
- In the REST call, the token is placed in the header as

-H "Authorization: Bearer TOKEN_VALUE"