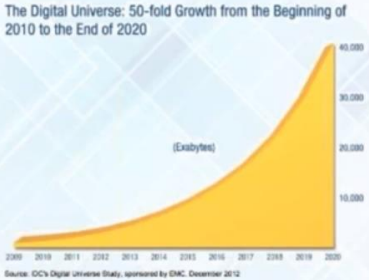
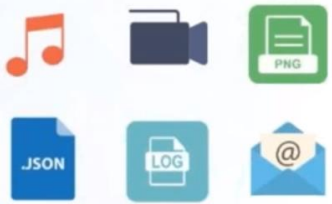
Spark Basic

5 V's of Big Data



Volume



Different kinds of data is being generated from various sources

Variety

Data is being generated at an alarming rate

Velocity

Mechanism to bring the correct meaning out of the data

Value

Min	Max	Mean	SD
4.3	?	5.84	0.83
2.0	4.4	3.05	50000000
15000	7.9	1.20	0.43
0.1	2.5	?	0.76

Uncertainty and inconsistencies in the data

Veracity

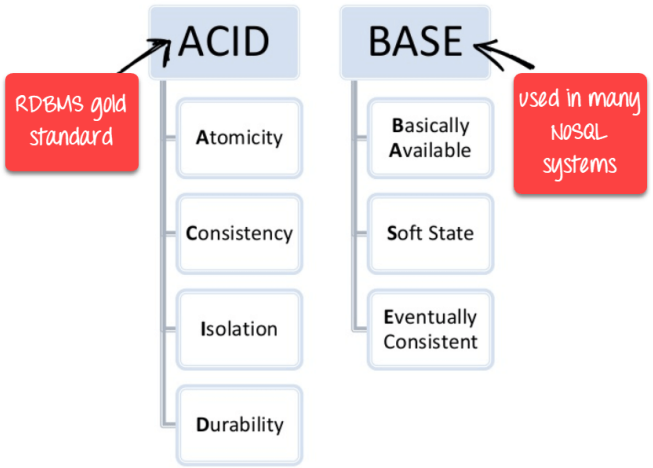
... ...

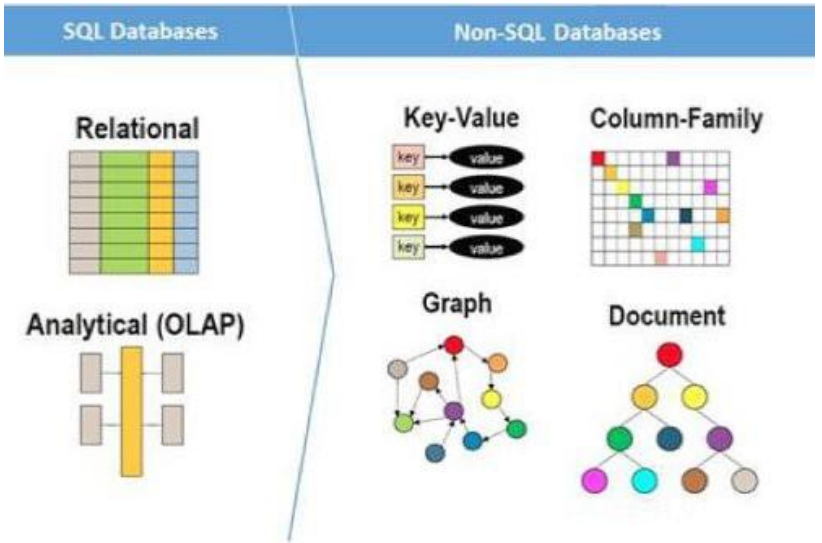
V's associated with Big Data may grow with time

SQL VS NOSQL

SQL		NoSQL
Relational	Model	Non-relational
Structured tables	Data	Semi-structured
Strict schema	Flexibility	Dynamic schema
ACID	Transactions	Mostly BASE, few ACID
Strong	Consistency	Eventual to Strong
Consistency prioritized	Availability	Basic Availability
Vertically by upgrading hardware	Scale	Horizontally by data partitioning

NoSQL:
- **Key-Value** — Dictionary or Hash Table
- **Wide Column** — 2-D Versioned Key-Value
- **Document** — Nested Objects (XML, JSON, YAML)
- **Graph** — Entity-Relationships





DATA LAKE vs DATA WAREHOUSE

DATA LAKE

Data

unstructured

Users

Data Scientists,
Data Analysts

Use cases

Stream Processing,
Machine Learning,
Real time analysis

Raw

Data Lakes contain unstructured, semi structured and structured data with minimal processing. It can be used to contain unconventional data such as log and sensor data

Large

Data Lakes contain vast amounts of data in the order of petabytes. Since the data can be in any form or size, large amounts of unstructured data can be stored indefinitely and can be transformed when in use only

Undefined

Data in data lakes can be used for a wide variety of applications, such as Machine Learning, Streaming analytics, and AI

DATA WAREHOUSE

Data

Structured

Users

Business Analysts

Use cases

Batch Processing,
BI, Reporting

Refined

Data Warehouses contain highly structured data that is cleaned, pre-processed and refined. This data is stored for very specific use cases such as BI.

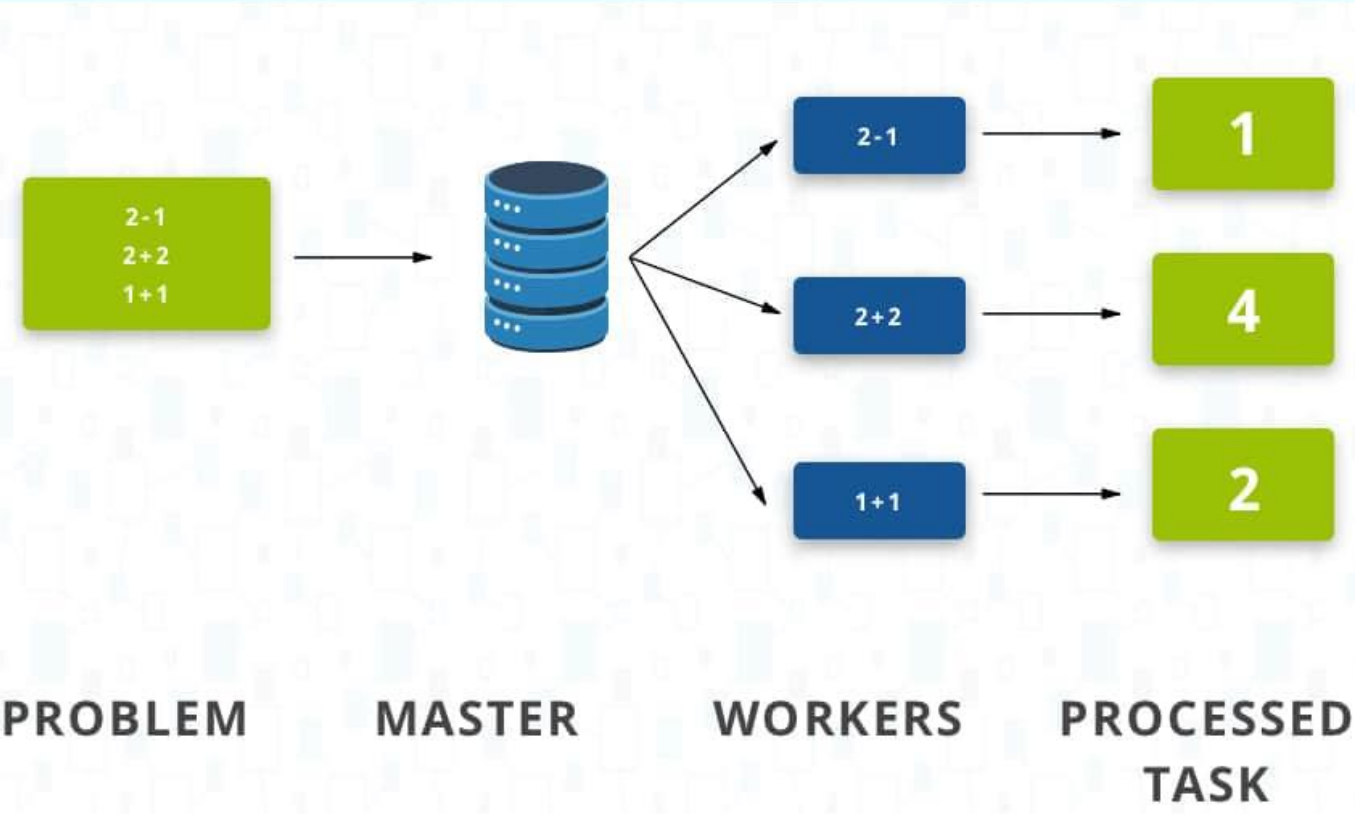
Smaller

Data Warehouses contain less data in the order of terabytes. In order to maintain data cleanliness and health of the warehouse, Data must be processed before ingestion and periodic purging of data is necessary

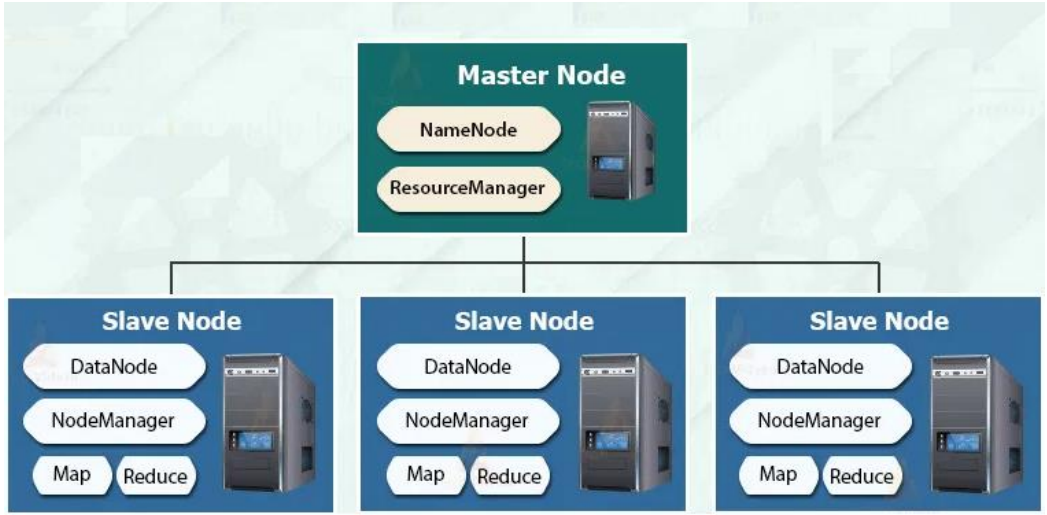
Relational

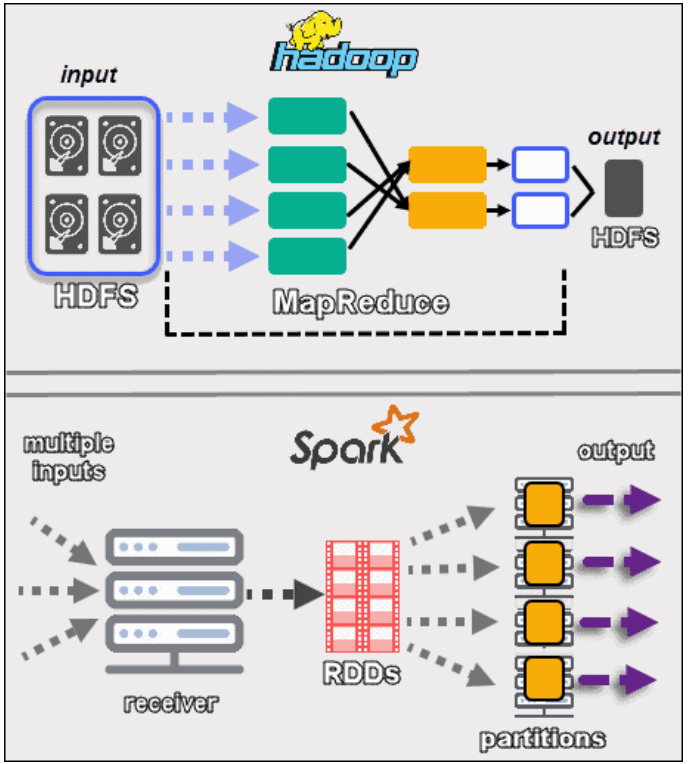
Data Warehouses contain historic and relational data, such as transaction systems, operations etc

Distributed Computing



Master-Slave Architecture



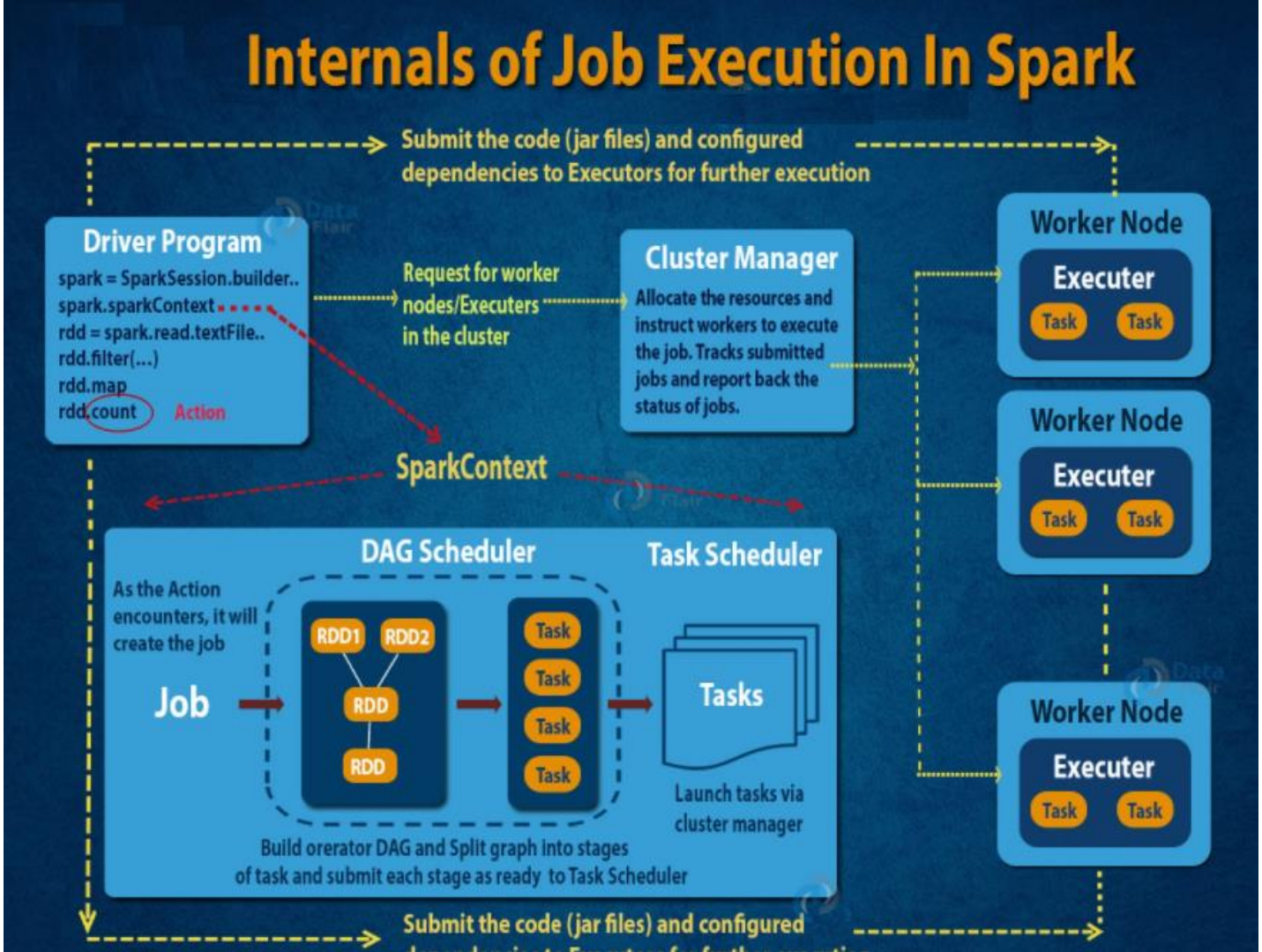


Cluster Manager

- Cluster Manager Types
- The system currently supports several cluster managers:
- **Standalone** – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- **Hadoop YARN** – the resource manager in Hadoop 2.

- **Kubernetes** – an open-source system for automating deployment, scaling, and management of containerized applications.

- **AWS Glue/ Databricks** - event-driven, serverless computing platform

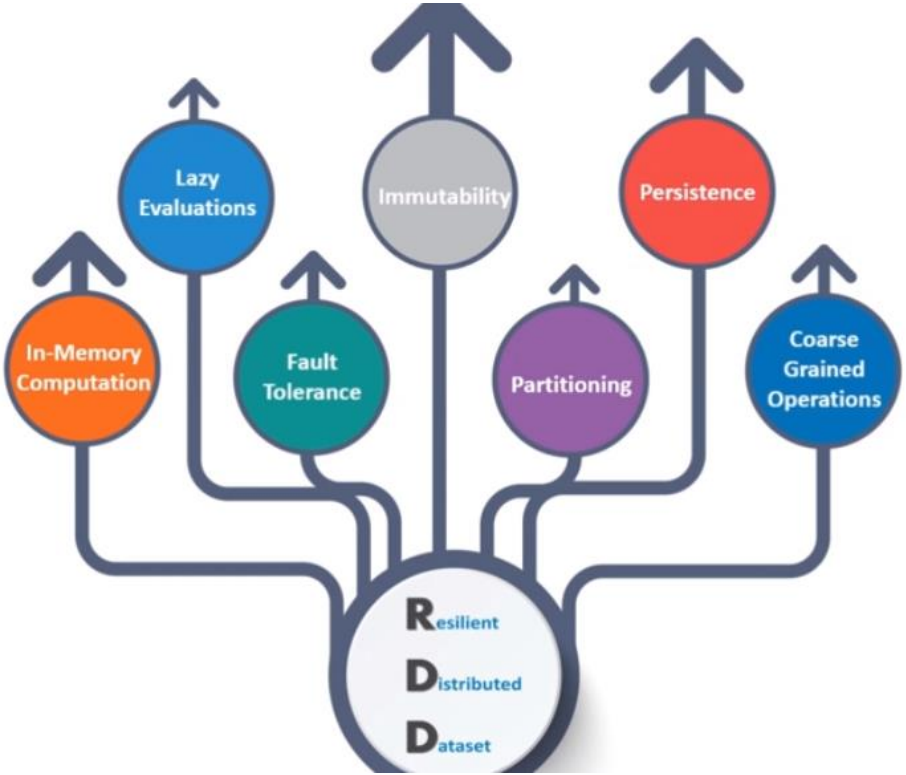

Internals of Job Execution In Spark
Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Task
Task
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Submit the code (jar files) and configured

Spark

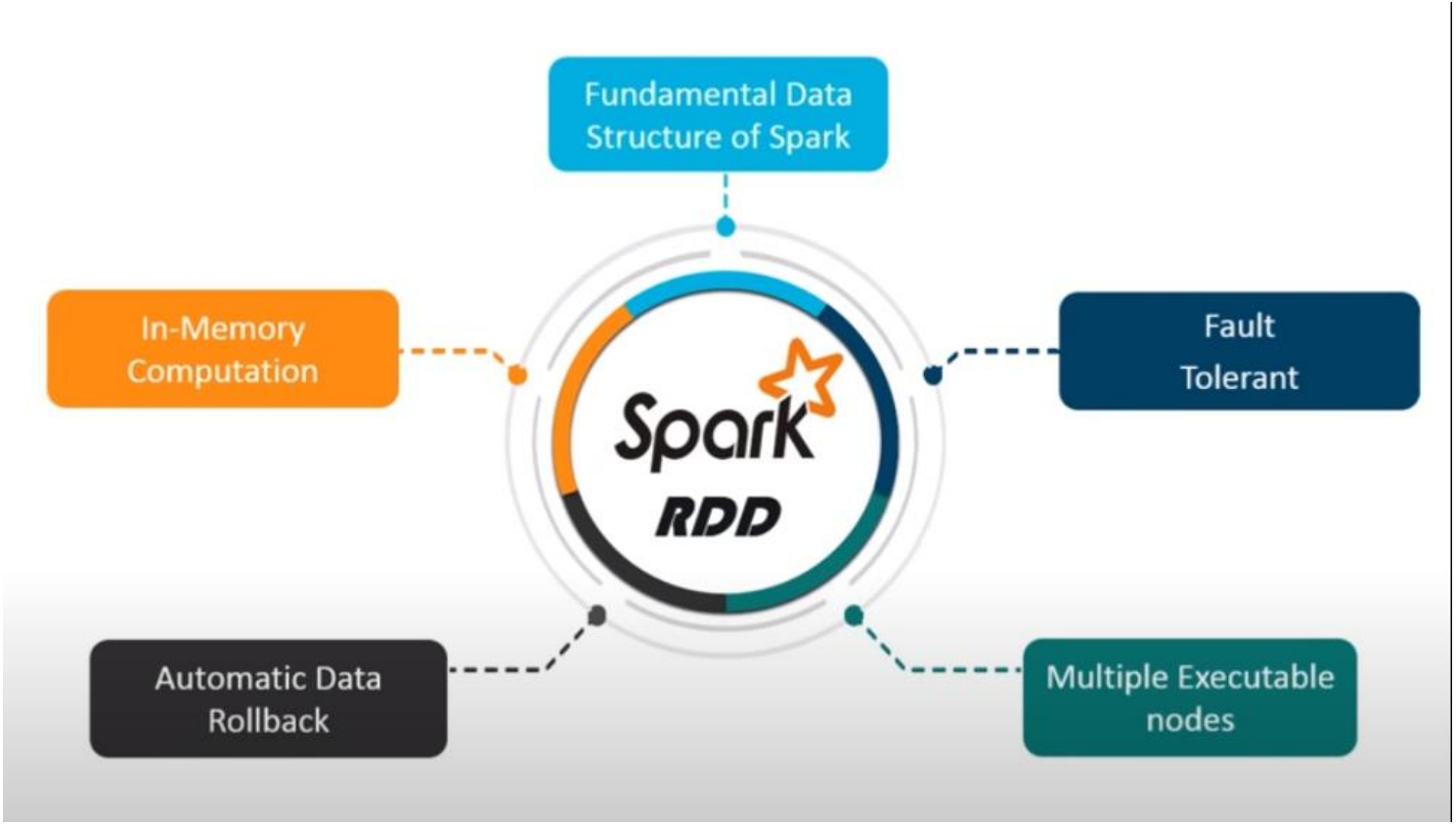
- Spark: A fast and general compute engine for Hadoop data.
- Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.

RDD

- In-Memory Computation
- Lazy Evaluation
- Fault Tolerance
- Immutability
- Partitioning
- Persistence
- Coarse Grained Operation

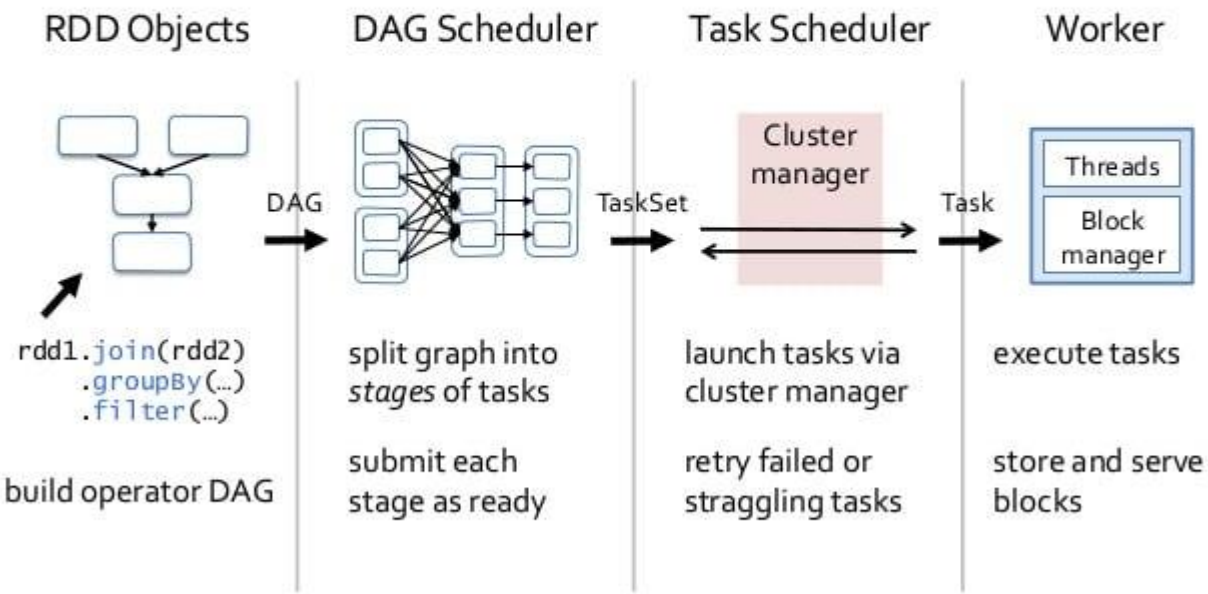


RDD


Fundamental Data Structure of Spark
In-Memory Computation
Fault Tolerant
Spark
RDD
Automatic Data Rollback
Multiple Executable nodes

RDD Transformation

RDD → Stages → Tasks



 = easy  = medium

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

ACTIONS

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

RDD Operations

 = easy  = medium

Essential Core & Intermediate PairRDD Operations

TRANSFORMATIONS

General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey

Math / Statistical

- sampleByKey

Set Theory / Relational

- cogroup (=groupWith)
- join
- subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Data Structure

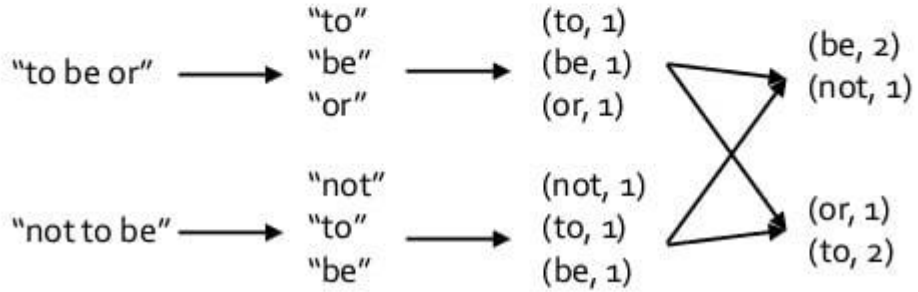
- partitionBy

ACTIONS

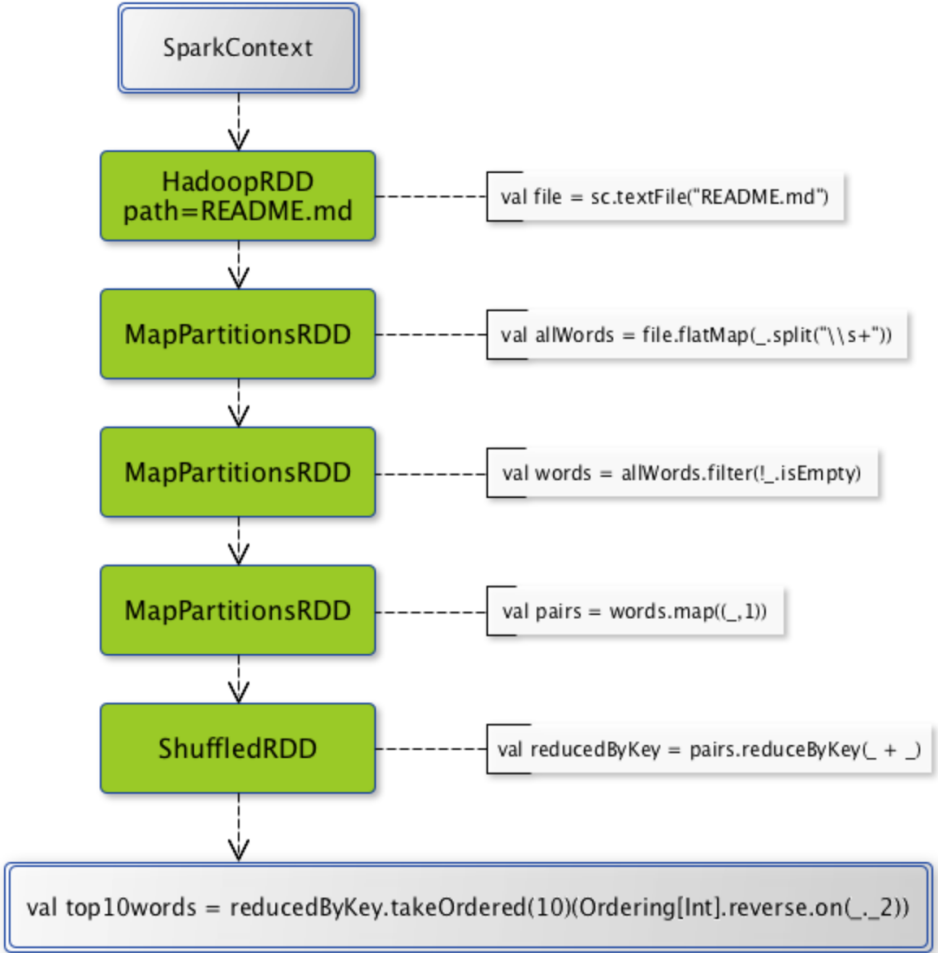
- keys
- values
- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact

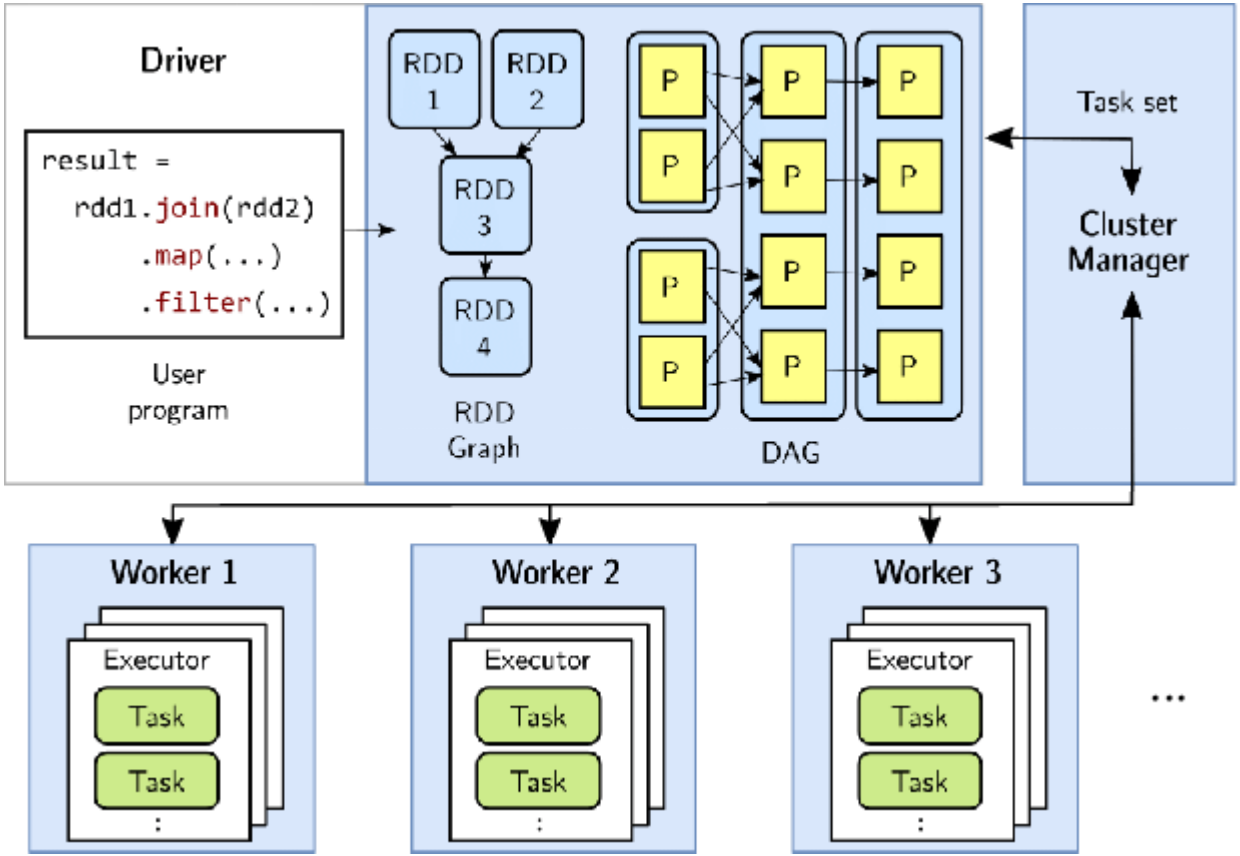
Example: Word Count

```
> lines = sc.textFile("hamlet.txt")
> counts = lines.flatMap(lambda line: line.split(" "))
                .map(lambda word => (word, 1))
                .reduceByKey(lambda x, y: x + y)
```



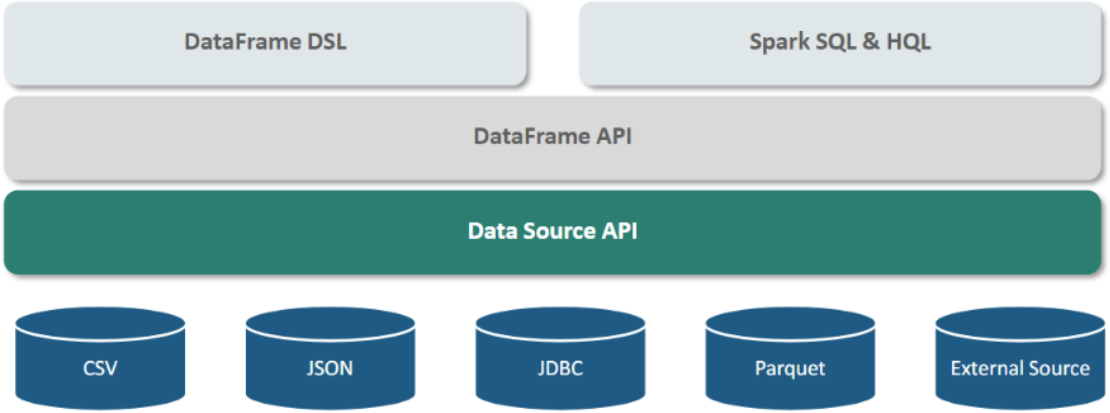
YAHOO!


SparkContext
HadoopRDD
path=README.md
val file = sc.textFile("README.md")
MapPartitionsRDD
val allWords = file.flatMap(_.split("\\s+"))
MapPartitionsRDD
val words = allWords.filter(!_.isEmpty)
MapPartitionsRDD
val pairs = words.map((_,1))
ShuffledRDD
val reducedByKey = pairs.reduceByKey(_ + _)
val top10words = reducedByKey.takeOrdered(10)(Ordering[Int].reverse.on(_._2))


Driver
result =
rdd1.join(rdd2)
.map(...)
.filter(...)
User
program
RDD
1
RDD
2
RDD
3
RDD
4
RDD
Graph
P
DAG
Task set
Cluster
Manager
Worker 1
Worker 2
Worker 3
Executor
Task
Task
:
...

Spark SQL Architecture

Architecture Of Spark SQL



Spark SQL has the following libraries:

1. Data Source API
2. DataFrame API
3. Interpreter & Optimizer
4. SQL Service

What is Spark SQL?

Spark SQL is a Spark module for structured data processing

1. It provides a DataFrame abstraction in Python, Java, and Scala to simplify working with structured datasets. DataFrames are similar to tables in a relational database
2. It can read and write data in a variety of structured formats (e.g., JSON, Hive Tables, and Parquet)
3. It lets you query the data using SQL, both inside a Spark program and from external tools

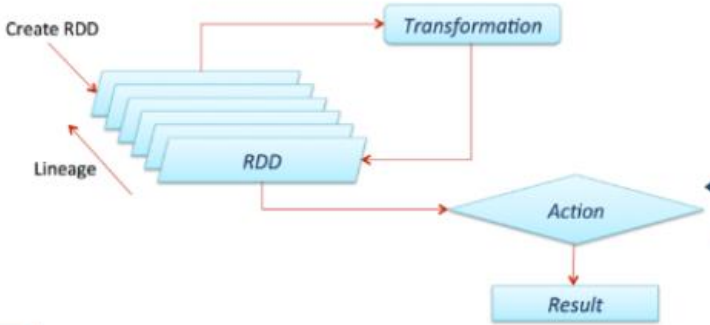
SQL Interpreter and Optimizer

SQL Interpreter & Optimizer handles the *functional programming* part of *Spark SQL*. It transforms the *DataFrames RDDs* to get the *required results* in the *required formats*

Features:

- Functional programming
- Transforming trees
- Faster than RDDs
- Processes all size data

e.g. Catalyst: A modular library for distinct optimization

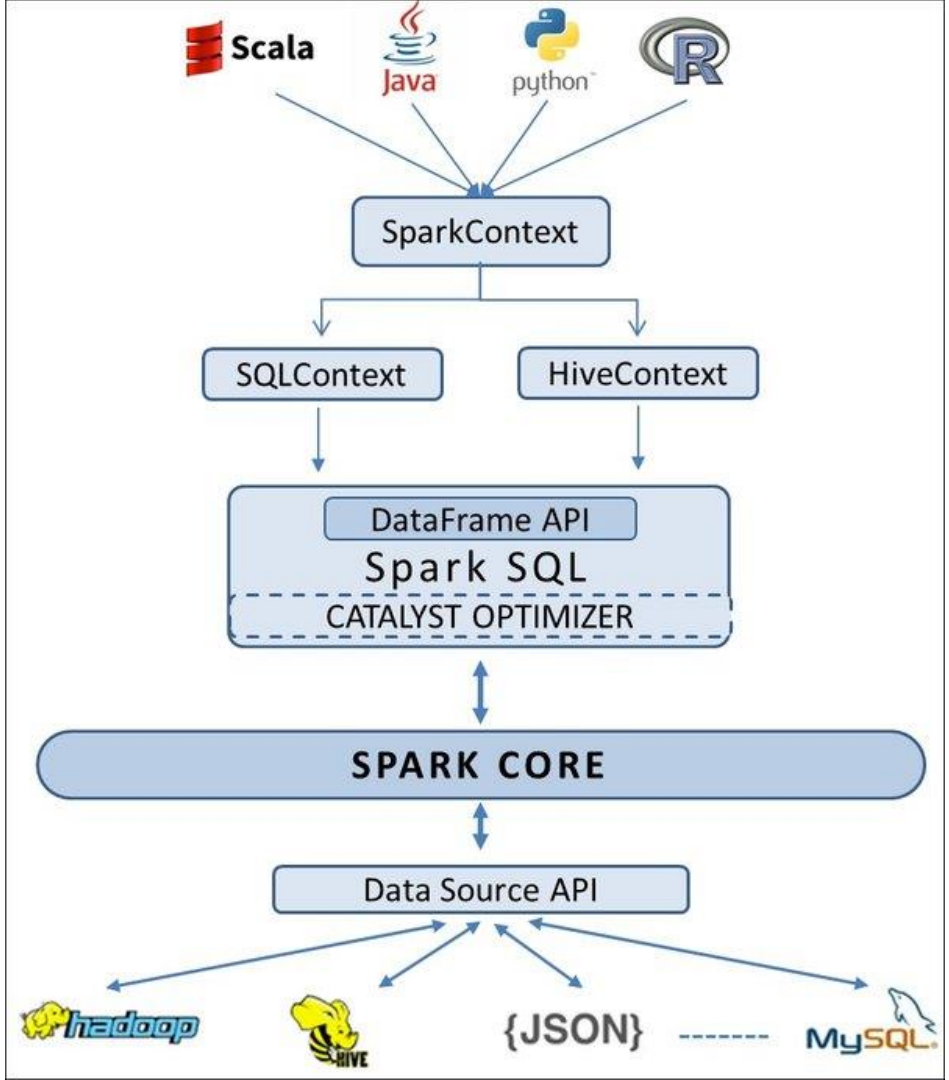


Interpreter & Optimizer

Data Source API

DataFrame API

Named Columns

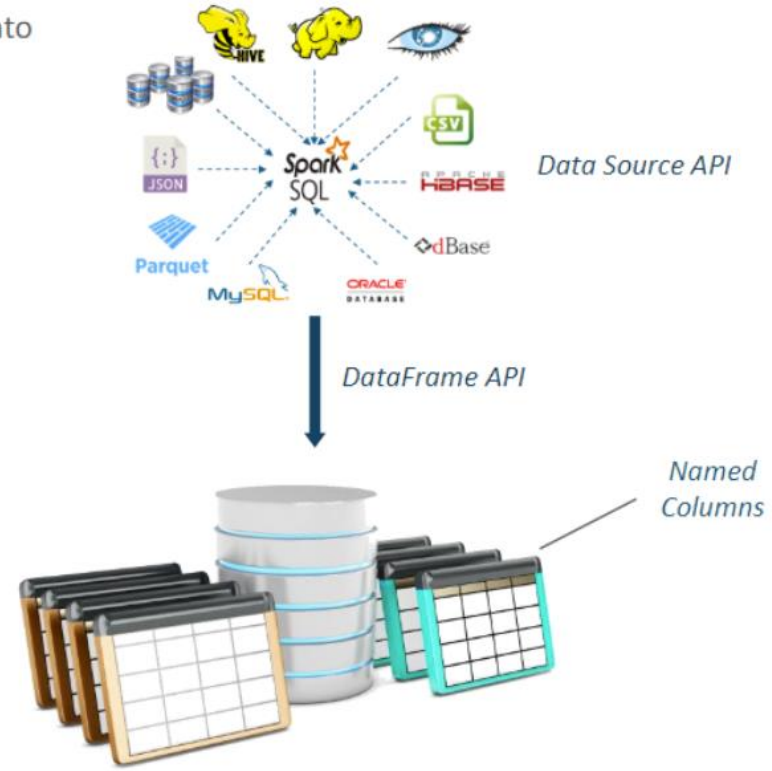

Scala
Java
python
R
SparkContext
SQLContext
HiveContext
DataFrame API
Spark SQL
CATALYST OPTIMIZER
SPARK CORE
Data Source API
hadoop
HIVE
{JSON}
MySQL

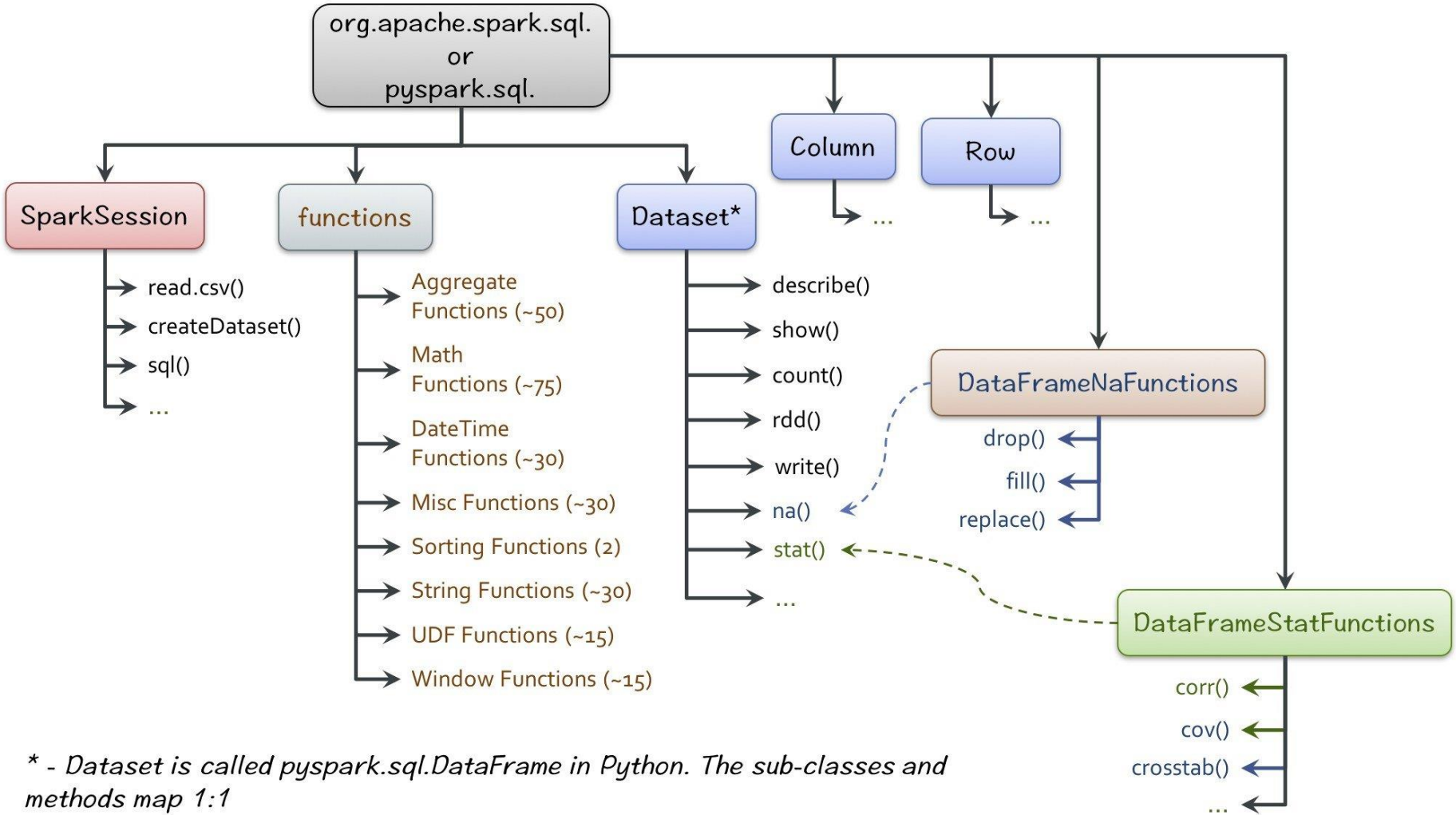
DataFrame API

DataFrame API converts the data that is read through Data Source API into tabular columns to help perform SQL operations

Features:

- Distributed collection of data organized into named columns
- Equivalent to a relational table in SQL
- Lazily evaluated

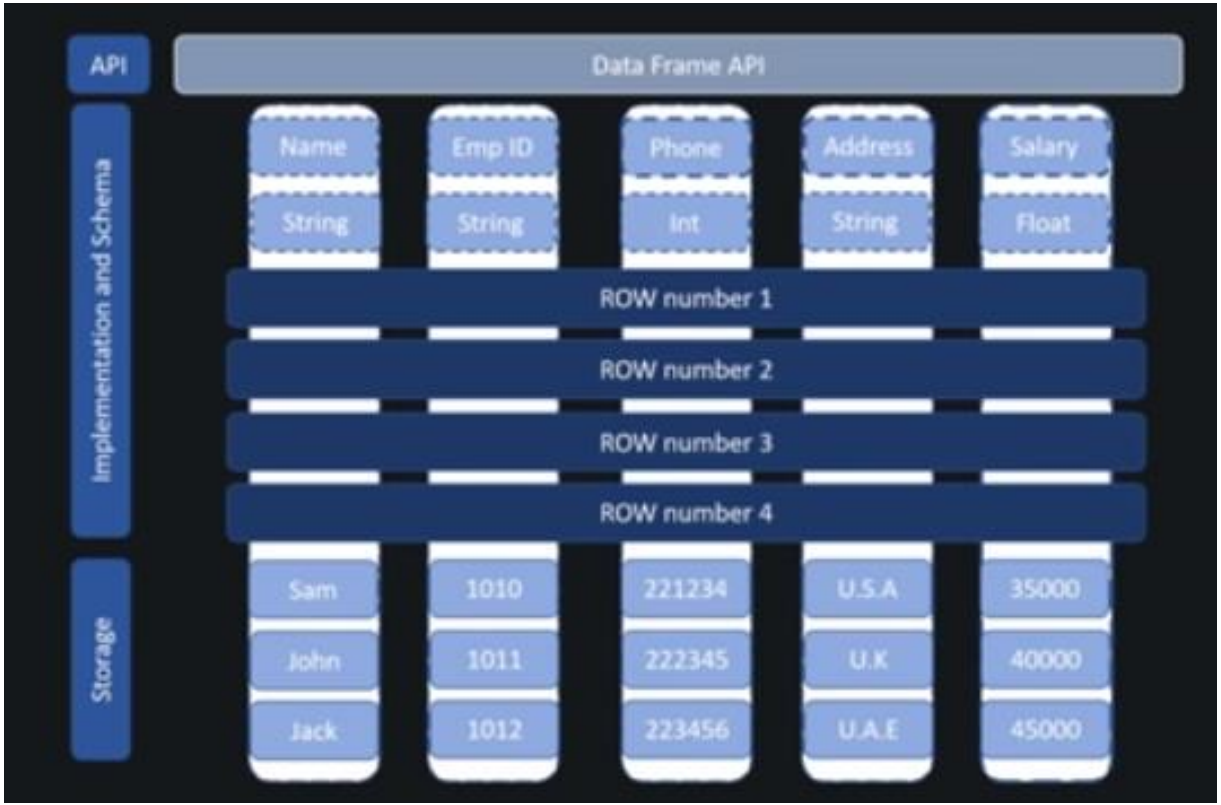




** - Dataset is called pyspark.sql.DataFrame in Python. The sub-classes and methods map 1:1*

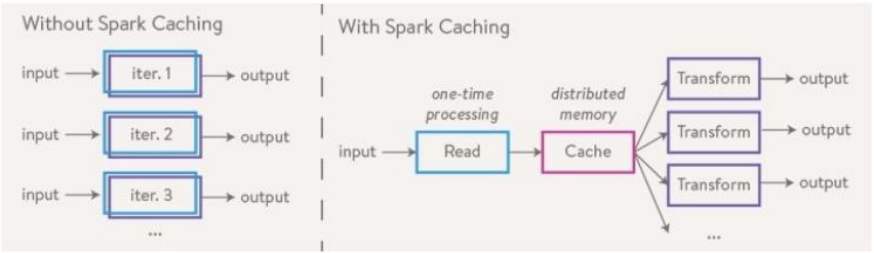
	Python Arrays	Spark RDDs
Loading Data	file = open("data.csv", "r") array = file.readlines() file.close()	rdd = sc.textFile(dbfs://FileStore/tables/data.csv)
Creating Array/RDD	array = ["a", "b", "c"]	rdd = sc. parallelize(["a", "b", "c"])
Printing All Rows	print(array)	rdd.collect()
Printing First *n* Rows	print(array[:5])	rdd.take(5)
Counting No. of Rows	print(len(array))	rdd.count()
Appending Rows	array.append("d")	rdd.union("d")
Unique Elements	numpy.unique(array)	rdd.distinct()
Saving Data	file = open("data.csv", "w") file.write(array) file.close()	rdd.saveAsTextFile(dbfs://FileStore/tables/data.csv)
Iterable Functions	lambda, map(*func*), filter(*func*), reduce(*func*) can be used in the same ways	

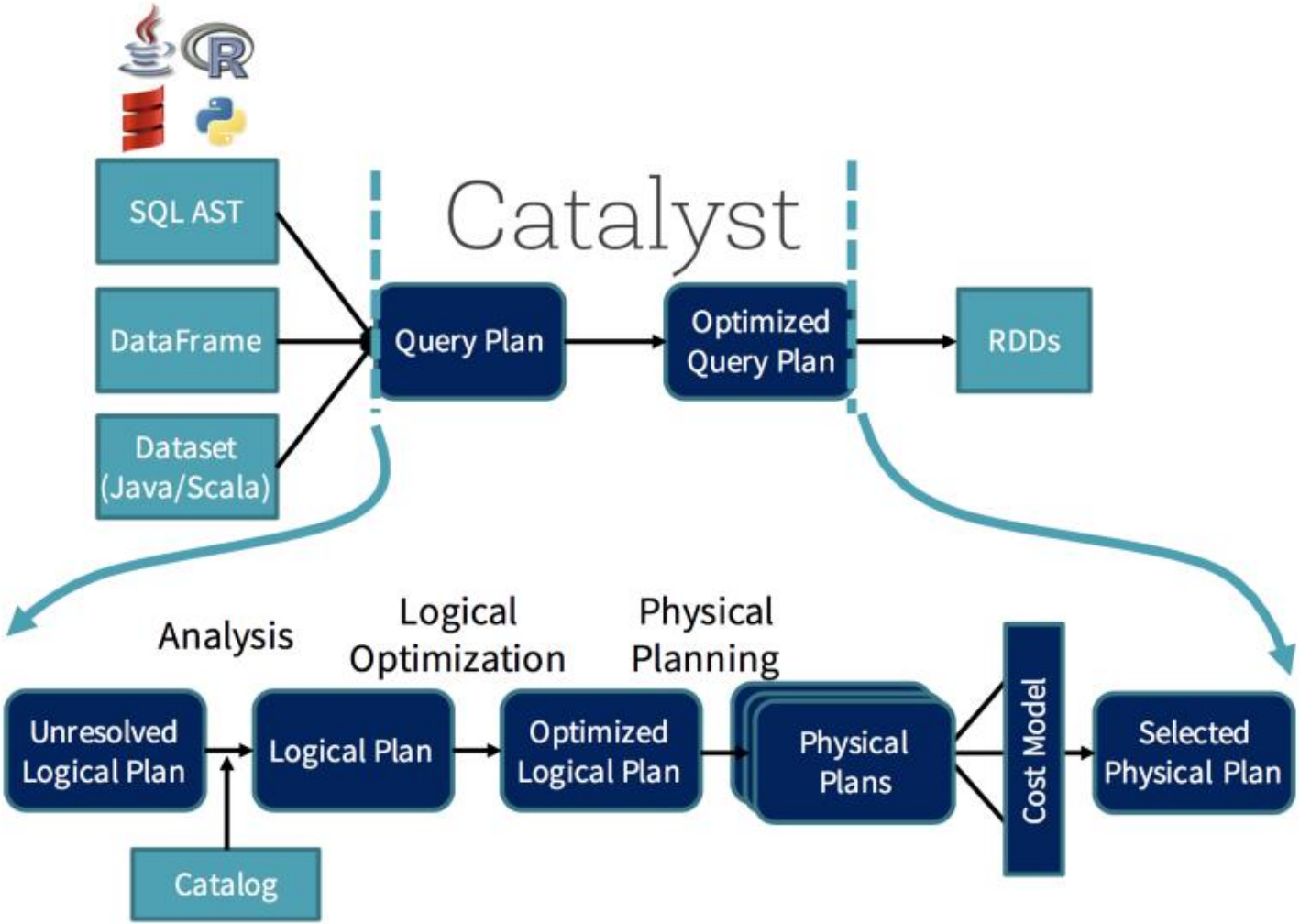
	Pandas DataFrames	Spark DataFrames
Loading Data	import pandas as pd df = pd.read_csv("data.csv", header=True, sep = ',')	from pyspark.sql import * df = spark.read.format("csv").options(header=True, sep=',').load("dbfs://FileStore/tables/data.csv")
DataFrame Schema	df.info()	df.printSchema()
Printing DataFrame	print(df)	df.show()
Printing First *n* Rows	df.head(5)	df.show(5)
Selecting Columns	df[["col1", "col2"]]	df.select("col1", "col2")
Filtering Columns	df[df["col1"] < 10]	df.filter(df["col1"] < 10)
Appending Rows	df.append(df2)	df.union(df2)
Sorting Columns	df.sort_values("col1")	df.sort("col1")
Dropping Null Rows	df.dropna()	df.na.drop()
Saving Data	df.to_csv("data.csv")	df.write.format("csv").save("data.csv")
Statistics Operators	mean(), max(), min() can be used in the same ways	

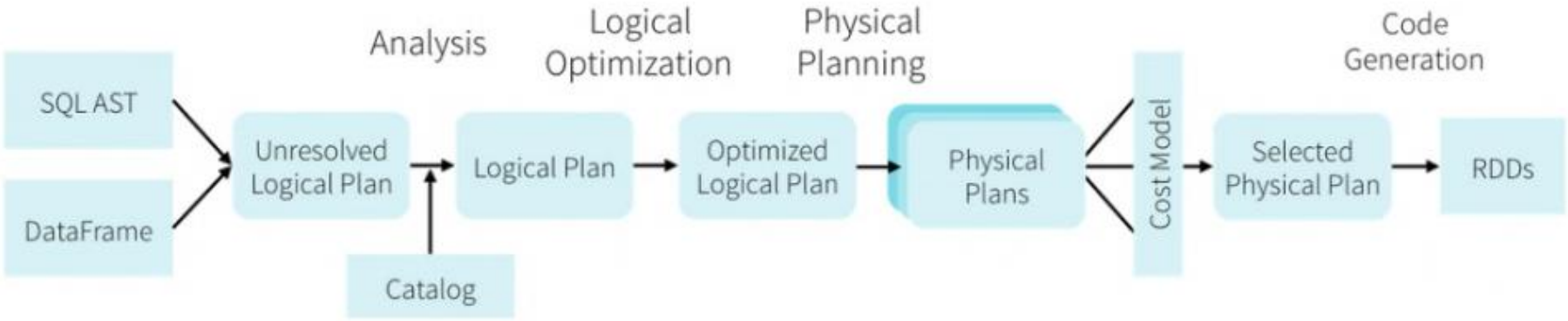

API
Data Frame API
Implementation and Schema
Storage
Name
Emp ID
Phone
Address
Salary
String
String
Int
String
Float
ROW number 1
ROW number 2
ROW number 3
ROW number 4
Sam
John
Jack
1010
1011
1012
221234
222345
223456
U.S.A
U.K
U.A.E
35000
40000
45000

Tungsten

- Unsafe memory, Memory Management and Binary Processing
- NO GC overhead
- Cache-aware computation
- No virtual function dispatches
- Intermediate data in memory vs CPU registers
- Loop unrolling and SIMD



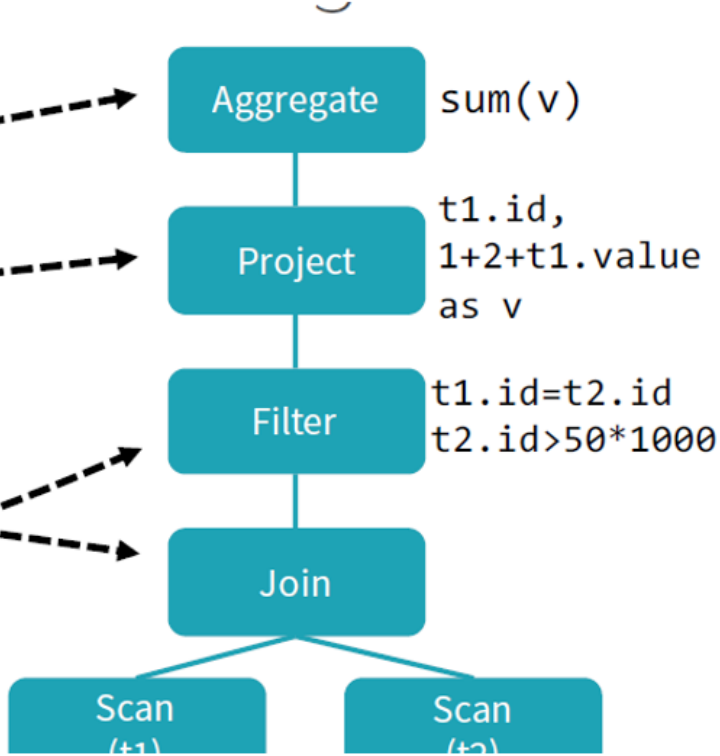

SQL AST
DataFrame
Dataset
(Java/Scala)
Catalyst
Query Plan
Optimized
Query Plan
RDDs
Analysis
Logical
Optimization
Physical
Planning
Unresolved
Logical Plan
Logical Plan
Optimized
Logical Plan
Physical
Plans
Cost Model
Selected
Physical Plan
Catalog


Analysis
Logical Optimization
Physical Planning
Code Generation
SQL AST
DataFrame
Unresolved Logical Plan
Catalog
Logical Plan
Optimized Logical Plan
Physical Plans
Cost Model
Selected Physical Plan
RDDs

Trees: Abstractions of Users' Programs

Query Plan

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

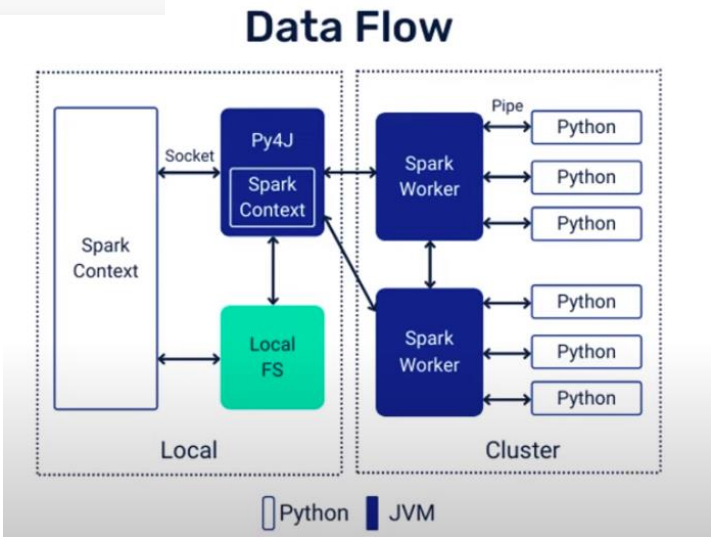


PYSPARK UDFs

UDFs allow us to write **our own custom transformations** using Python or Scala and to **use external libraries**. UDFs can take and return one or more columns as input.

We need to register these custom functions with Spark so that we can use them on all of the worker machines.

Spark serializes the function on the driver and transfers it over the network to all executor processes.



PANDAS UDFs

Pandas UDFs, also known as vectorized UDFs, use Apache Arrow to transfer data between JVM and Python processes and Pandas to work with the data.

We can define a Pandas UDF using the keyword pandas_udf as the decorator.

When using Apache Arrow in memory, columnar format, there is no need to serialize/pickle the data as it is already in a format consumable by the Python process. Instead of operating on individual inputs row by row, we are operating on a Pandas Series or DataFrame.

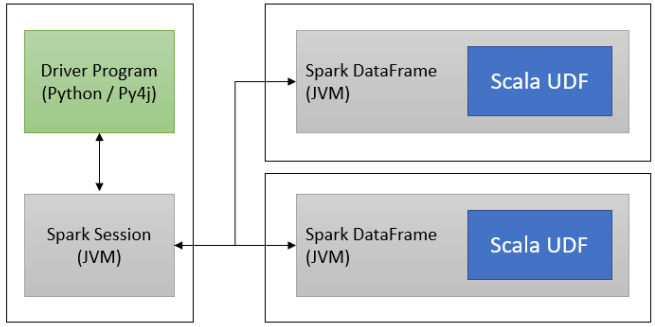
FUNCTIONALITY	SPARK VERSION	Year
Python UDFs for SQL	1.2	2014
Java UDFs in Python API	2.1	2017
Pandas UDFs	2.3	2018
New Pandas UDFs with Python type hints	3.0	2020

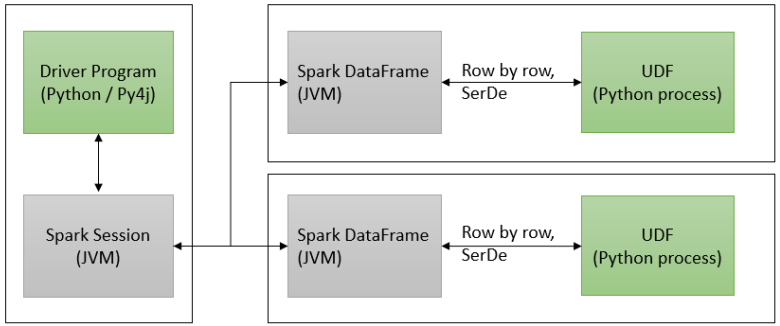
VECTORIZED PANDAS UDFs

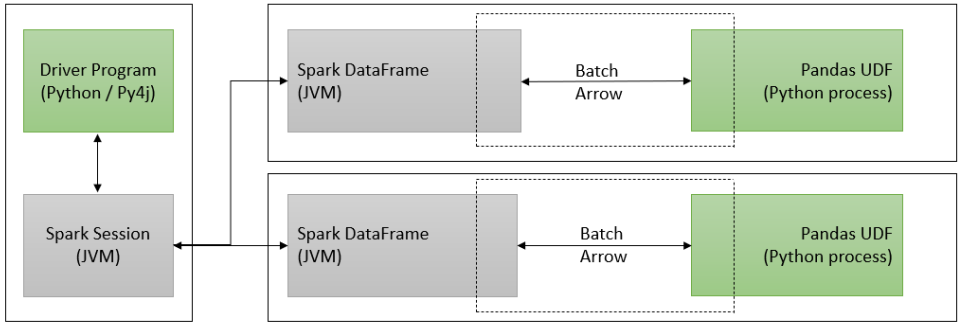
Plus one example

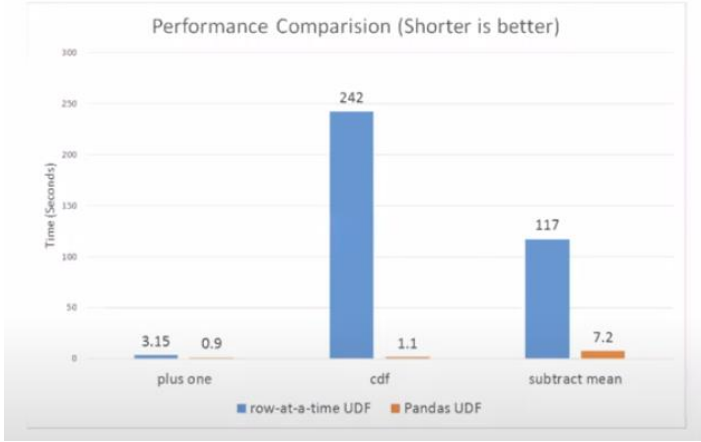
```
from pyspark.sql.functions import udf

# Use udf to define a row-at-a-time udf
@udf('double')
# Input/output are both a single double value
def plus_one(v):
      return v + 1

df.withColumn('v2', plus_one(df.v))
```

```
from pyspark.sql.functions import pandas_udf, PandasUDFType

# Use pandas_udf to define a Pandas UDF
@pandas_udf('double', PandasUDFType.SCALAR)
# Input/output are both a pandas.Series of doubles

def pandas_plus_one(v):
    return v + 1

df.withColumn('v2', pandas_plus_one(df.v))
```


Driver Program
(Python / Py4j)
Spark Session
(JVM)
Spark DataFrame
(JVM)
Scala UDF
Spark DataFrame
(JVM)
Scala UDF


Driver Program
(Python / Py4j)
Spark Session
(JVM)
Spark DataFrame
(JVM)
Row by row,
SerDe
UDF
(Python process)
Spark DataFrame
(JVM)
Row by row,
SerDe
UDF
(Python process)


Driver Program
(Python / Py4j)
Spark Session
(JVM)
Spark DataFrame
(JVM)
Batch
Arrow
Pandas UDF
(Python process)
Spark DataFrame
(JVM)
Batch
Arrow
Pandas UDF
(Python process)


Performance Comparision (Shorter is better)
Time (Seconds)
300
250
200
150
100
50
0
3.15
0.9
242
1.1
117
7.2
plus one
cdf
subtract mean
row-at-a-time UDF
Pandas UDF

Spark Join

LEFT JOIN

Everything on the left
+
anything on the right that matches

```
SELECT *
FROM TABLE_1
LEFT JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

ANTI LEFT JOIN

Everything on the left
that is NOT on the right

```
SELECT *
FROM TABLE_1
LEFT JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
WHERE TABLE_2.KEY IS NULL
```

RIGHT JOIN

Everything on the right
+
anything on the left that matches

```
SELECT *
FROM TABLE_1
RIGHT JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

ANTI RIGHT JOIN

Everything on the right
that is NOT on the left

```
SELECT *
FROM TABLE_1
RIGHT JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
WHERE TABLE_1.KEY IS NULL
```

OUTER JOIN

Everything on the right
+
Everything on the left

```
SELECT *
FROM TABLE_1
OUTER JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

ANTI OUTER JOIN

Everything on the left and right
that is unique to each side

```
SELECT *
FROM TABLE_1
OUTER JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
WHERE TABLE_1.KEY IS NULL
OR TABLE_2.KEY IS NULL
```

INNER JOIN

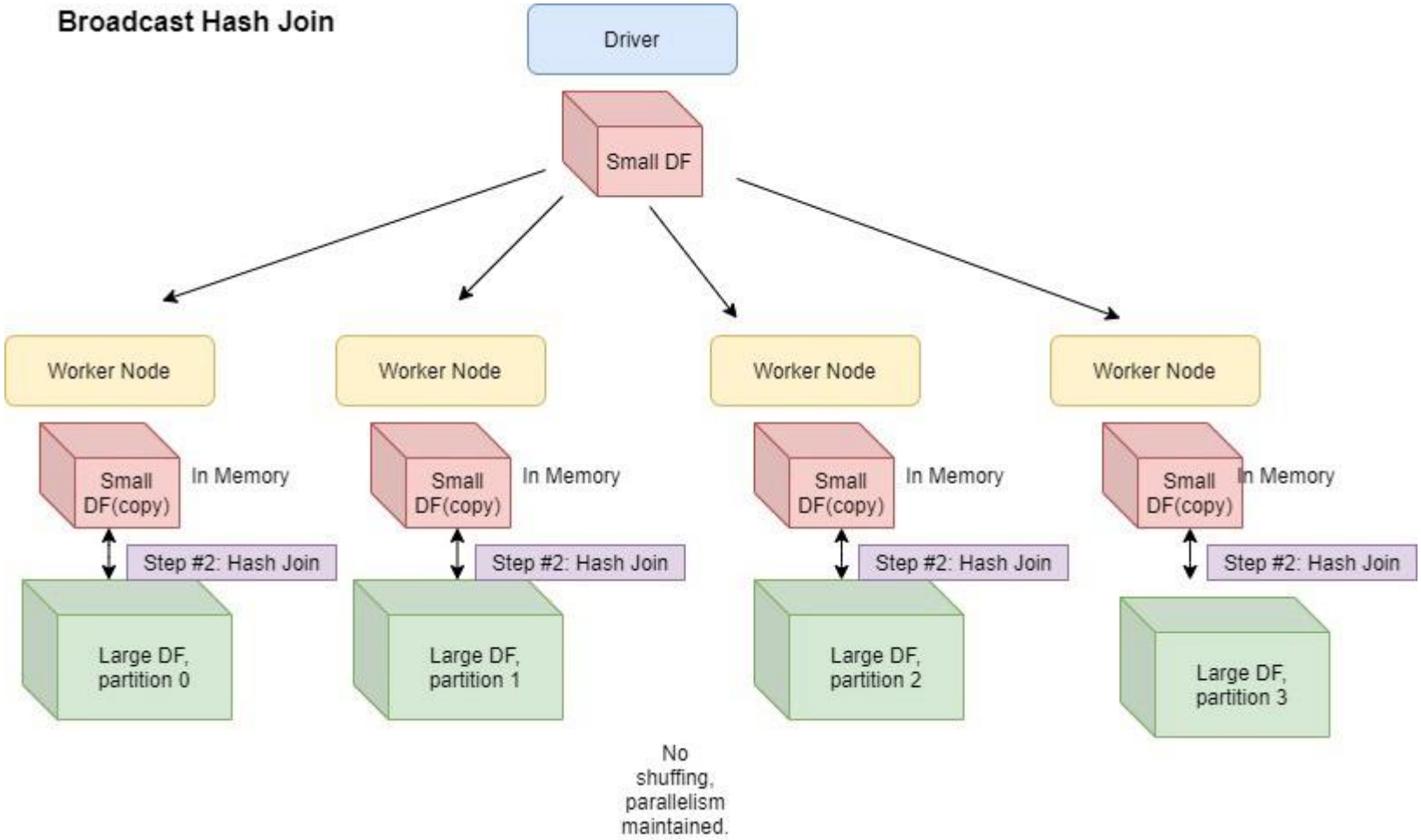
Only the things that match on the left AND the right

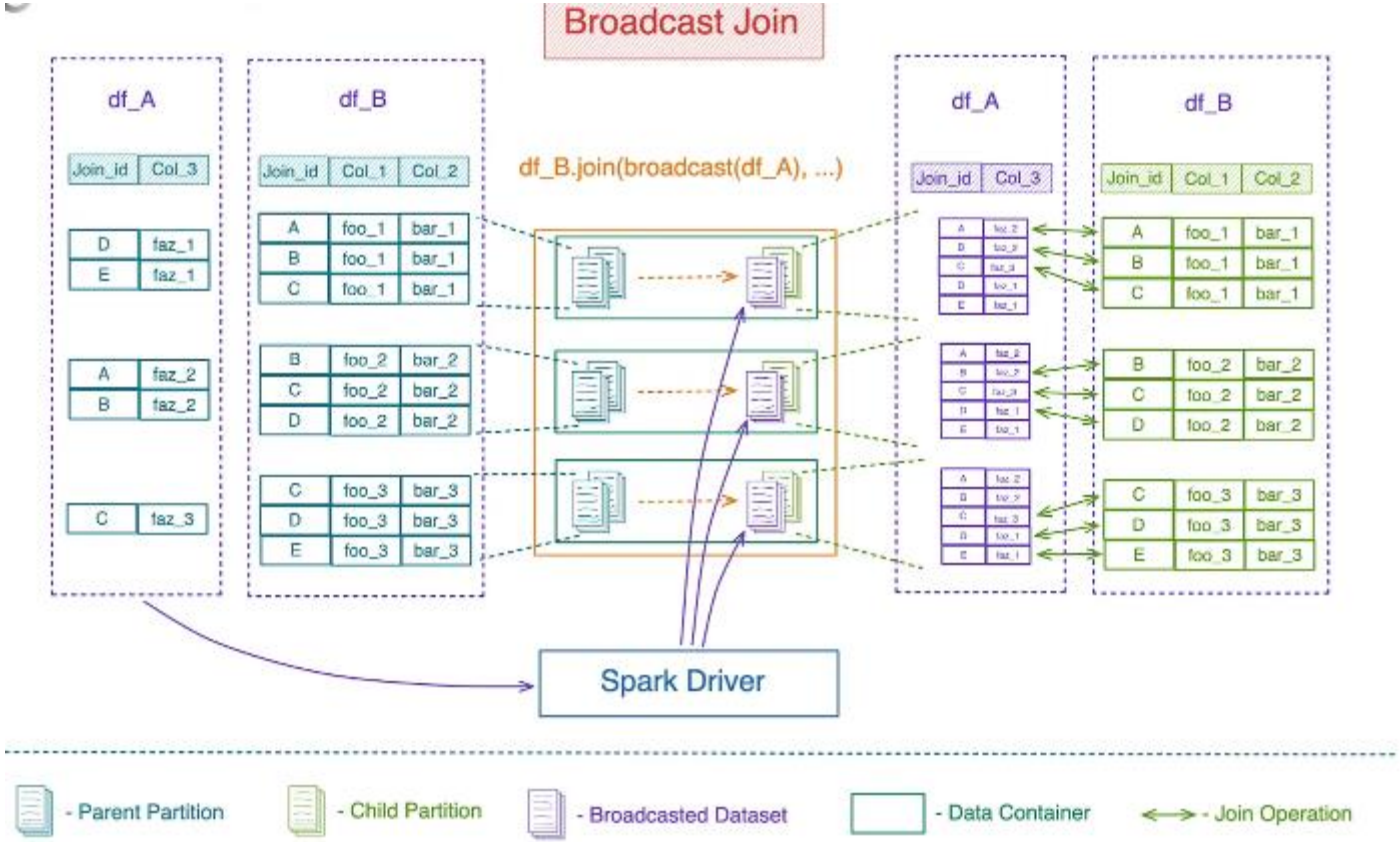
```
SELECT *
FROM TABLE_1
INNER JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

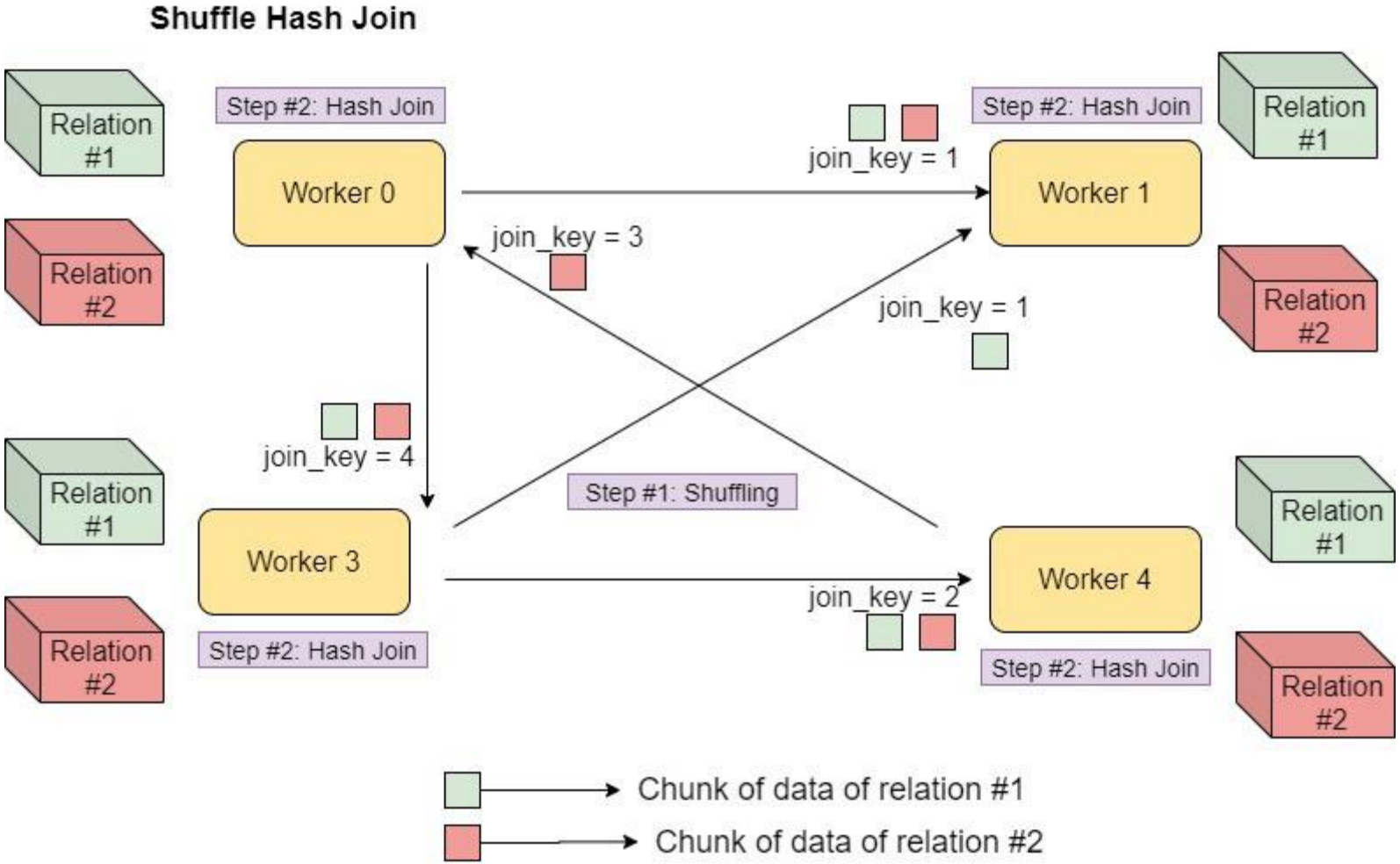
CROSS JOIN

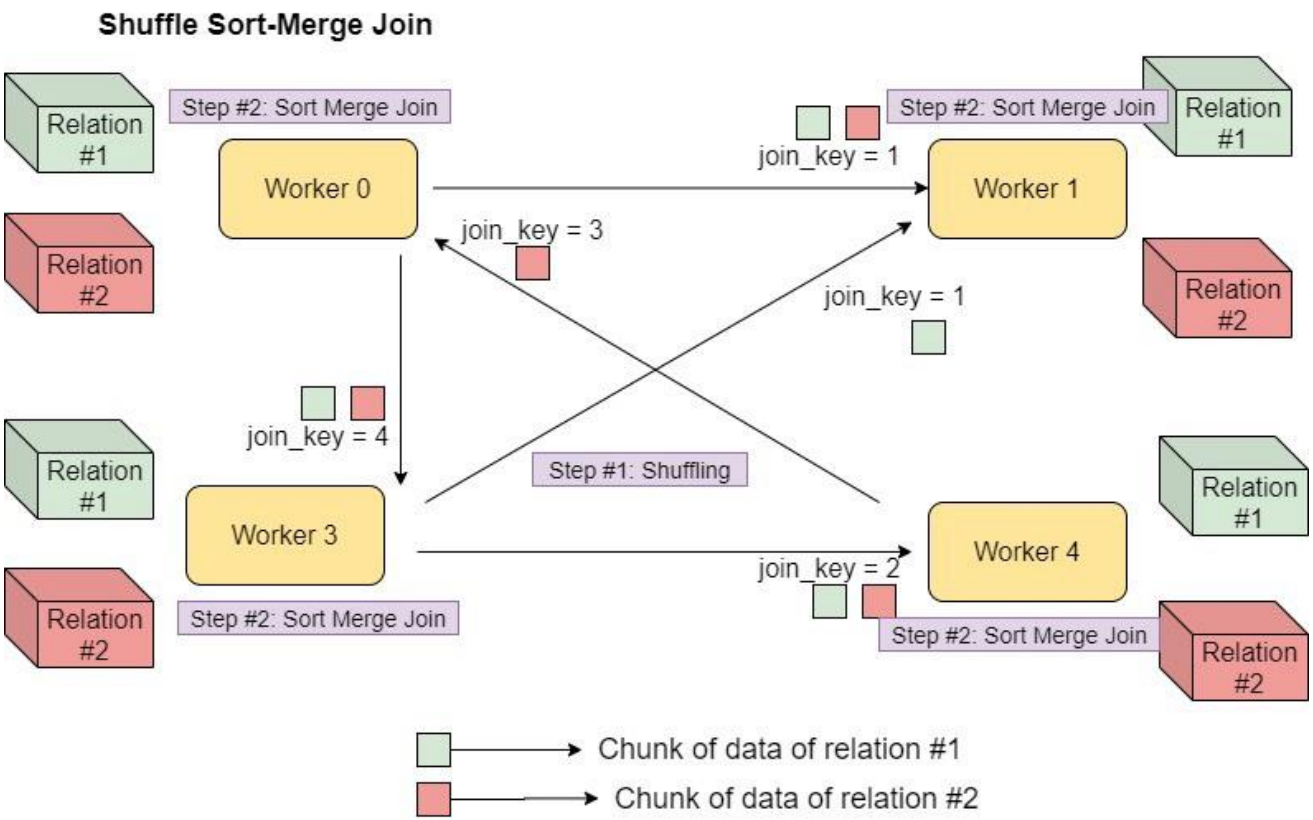
All combination of rows from the right and the left (cartesean product)

```
SELECT *
FROM TABLE_1
CROSS JOIN TABLE_2
```

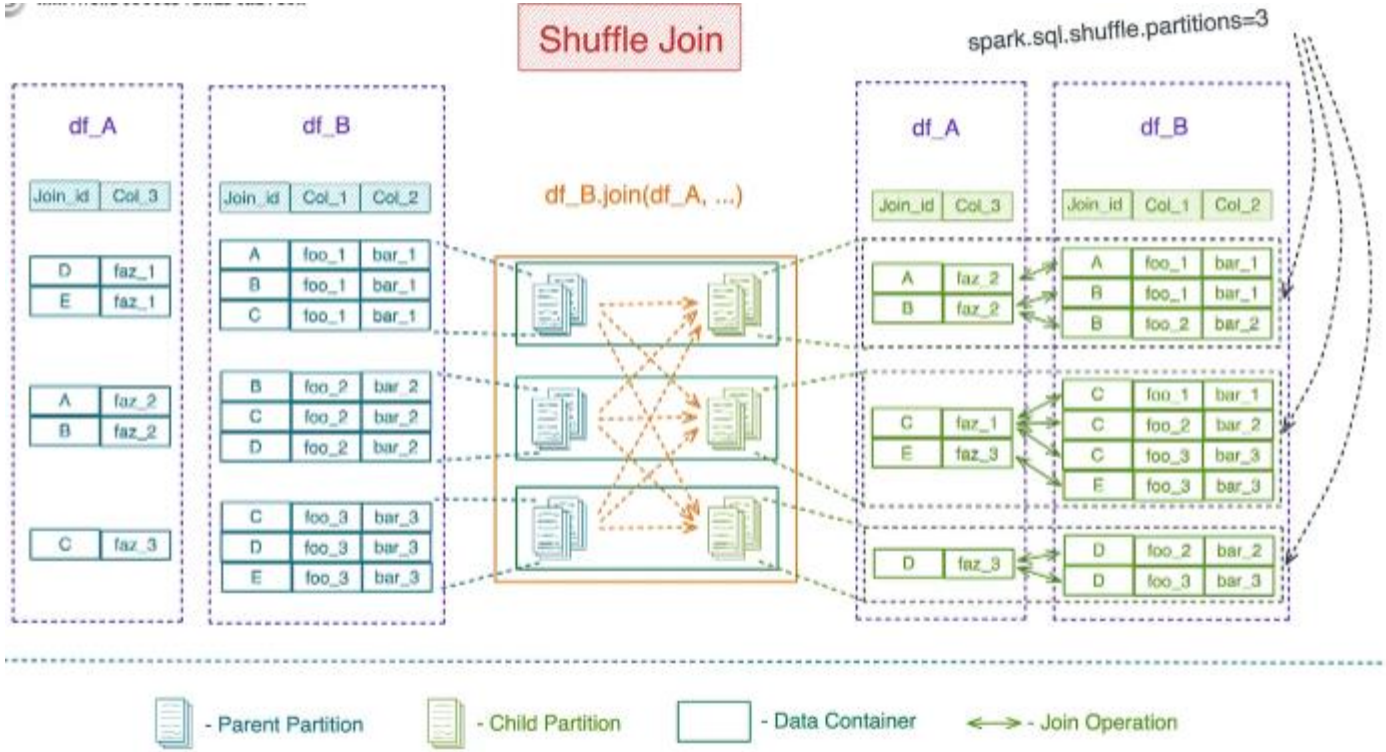

Broadcast Hash Join
Driver
Small DF
Worker Node
Worker Node
Worker Node
Worker Node
Small DF(copy)
In Memory
Small DF(copy)
In Memory
Small DF(copy)
In Memory
Small DF(copy)
In Memory
Step #2: Hash Join
Step #2: Hash Join
Step #2: Hash Join
Step #2: Hash Join
Large DF, partition 0
Large DF, partition 1
Large DF, partition 2
Large DF, partition 3
No shuffing, parallelism maintained.


Broadcast Join
df_A
Join_id
Col_3
D
faz_1
E
faz_1
A
faz_2
B
faz_2
C
faz_3
df_B
Join_id
Col_1
Col_2
A
foo_1
bar_1
B
foo_1
bar_1
C
foo_1
bar_1
B
foo_2
bar_2
C
foo_2
bar_2
D
foo_2
bar_2
C
foo_3
bar_3
D
foo_3
bar_3
E
foo_3
bar_3
df_B.join(broadcast(df_A), ...)
df_A
Join_id
Col_3
df_B
Join_id
Col_1
Col_2
Spark Driver
- Parent Partition
- Child Partition
- Broadcasted Dataset
- Data Container
- Join Operation

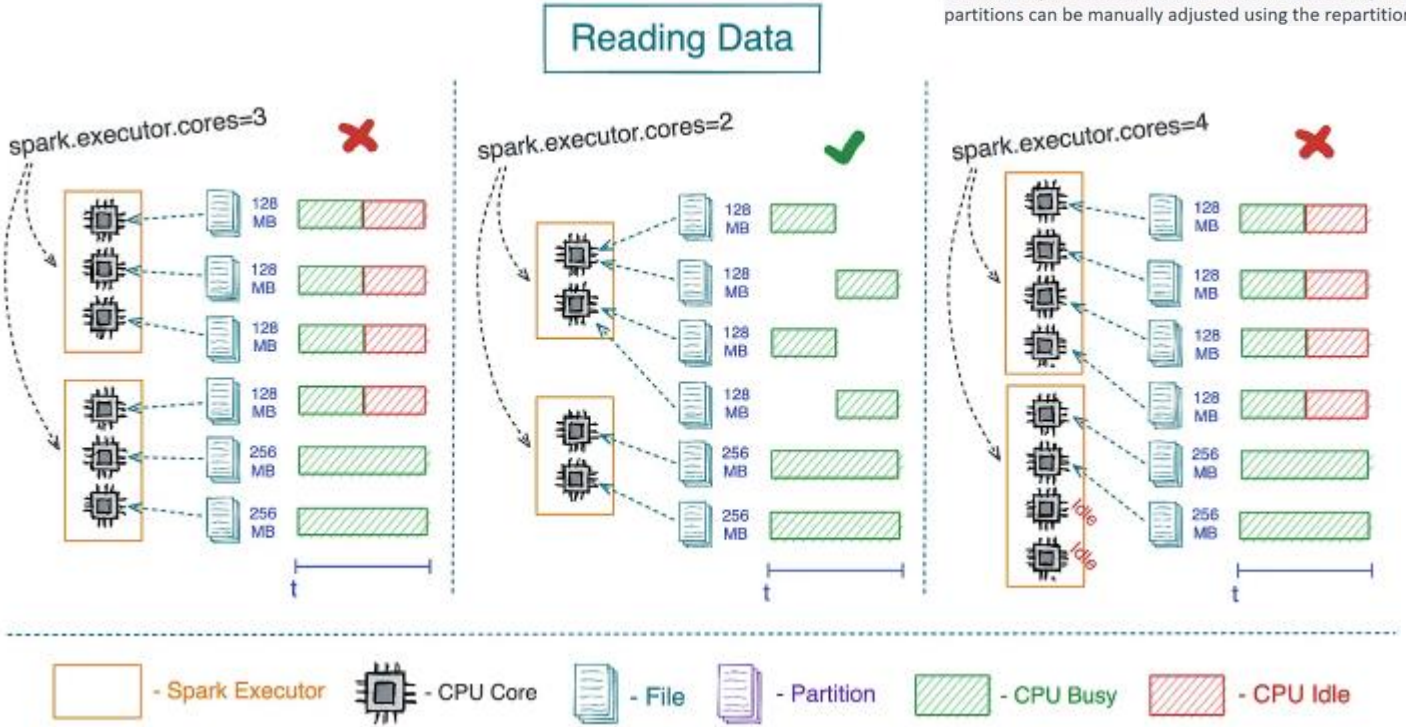

Shuffle Hash Join
Relation #1
Relation #2
Step #2: Hash Join
Worker 0
join_key = 1
Step #2: Hash Join
Worker 1
Relation #1
Relation #2
join_key = 3
join_key = 1
join_key = 4
Step #1: Shuffling
Relation #1
Relation #2
Worker 3
Worker 4
join_key = 2
Step #2: Hash Join
Step #2: Hash Join
Relation #1
Relation #2
Chunk of data of relation #1
Chunk of data of relation #2



How does Spark perform data shuffling and repartitioning when joining two large datasets? Answer: When joining two large datasets in Spark, Spark will first partition both datasets based on the join key. This helps to minimize the amount of data that needs to be shuffled between nodes during the join operation. After the datasets are partitioned, Spark will then perform a shuffle operation to redistribute the data based on the join key, ensuring that all data for a given join key is co-located on the same node. Finally, Spark will perform the join operation by processing the data in parallel on each node, producing the final result.

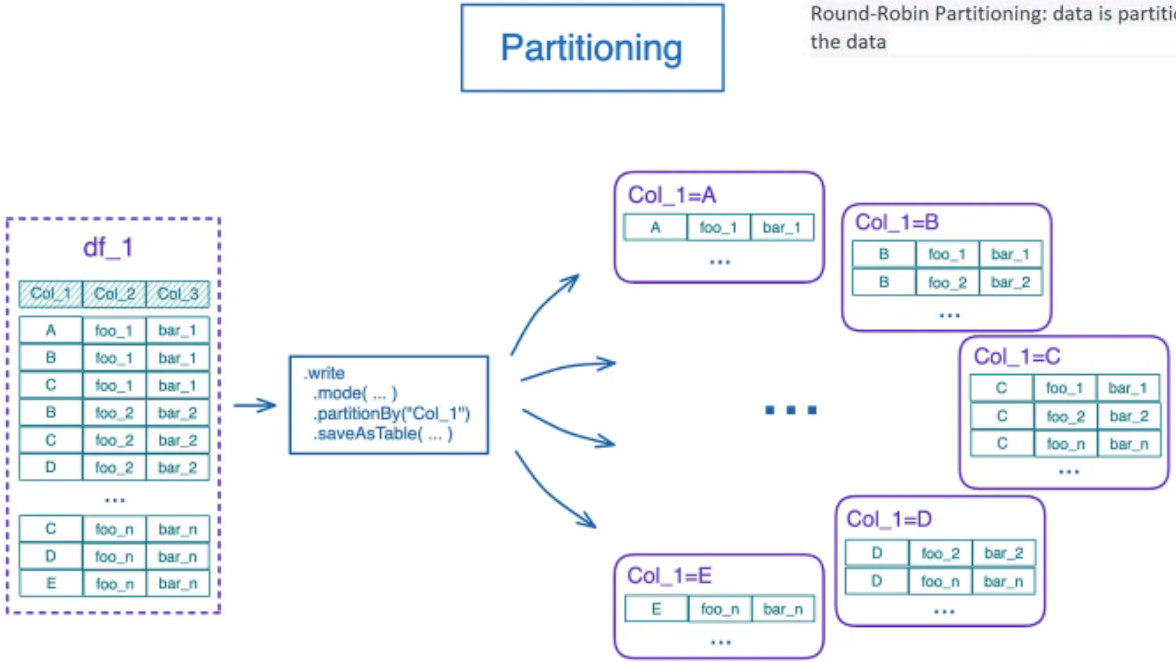

Shuffle Join
spark.sql.shuffle.partitions=3
df_A
df_B
df_B.join(df_A, ...)
Join_id
Col_3
Col_1
Col_2
- Parent Partition
- Child Partition
- Data Container
- Join Operation

Answer: Spark determines the number of partitions for a given RDD or DataFrame based on the configuration settings for the Spark cluster, the amount of available memory, and the size of the data. **By default, Spark will create one partition for each block of data in HDFS, which is typically 128 MB**. However, the number of partitions can be manually adjusted using the repartition or coalesce methods.



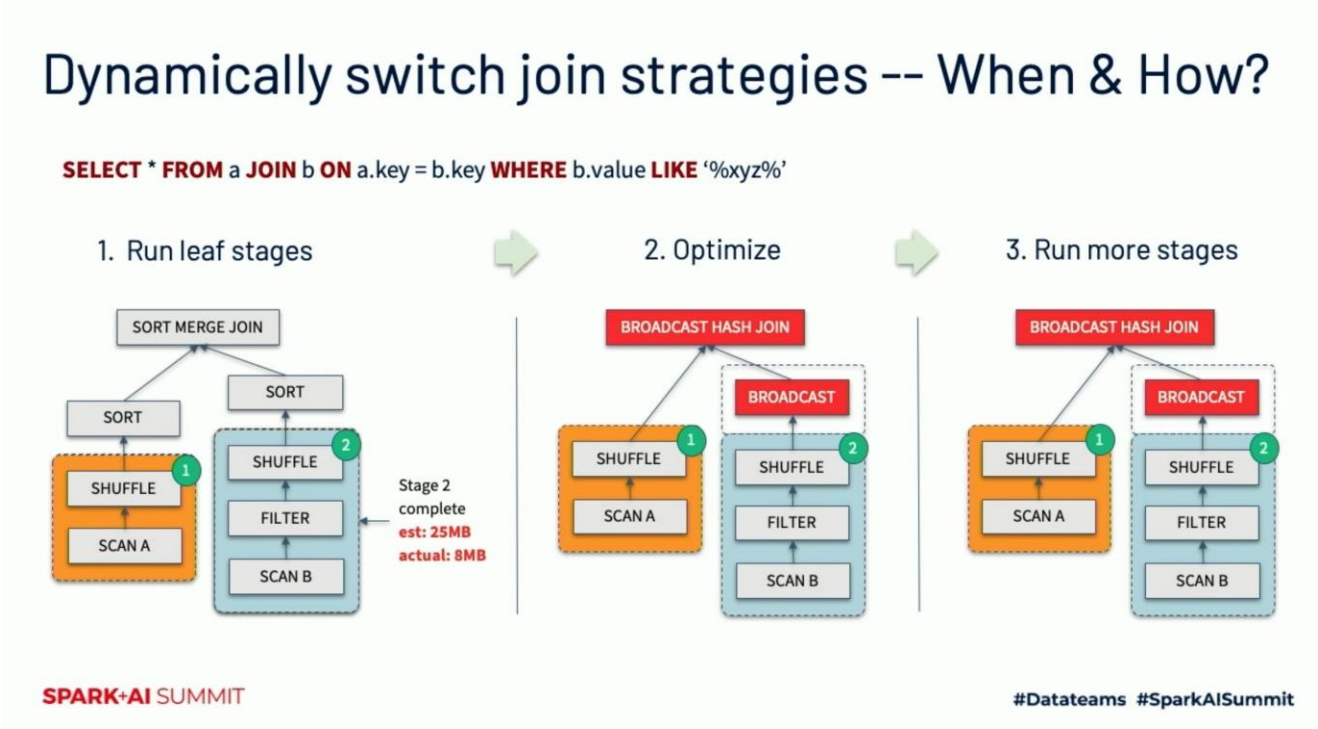
Partition (or task) refers to a unit of work. If you have a 200G hadoop file loaded as an RDD and chunked by 128M (Spark default), then you have ~2000 partitions in this RDD. The number of cores determines how many partitions can be processed at any one time, and up to 2000 (capped at the number of partitions/tasks) can execute this RDD in parallel.

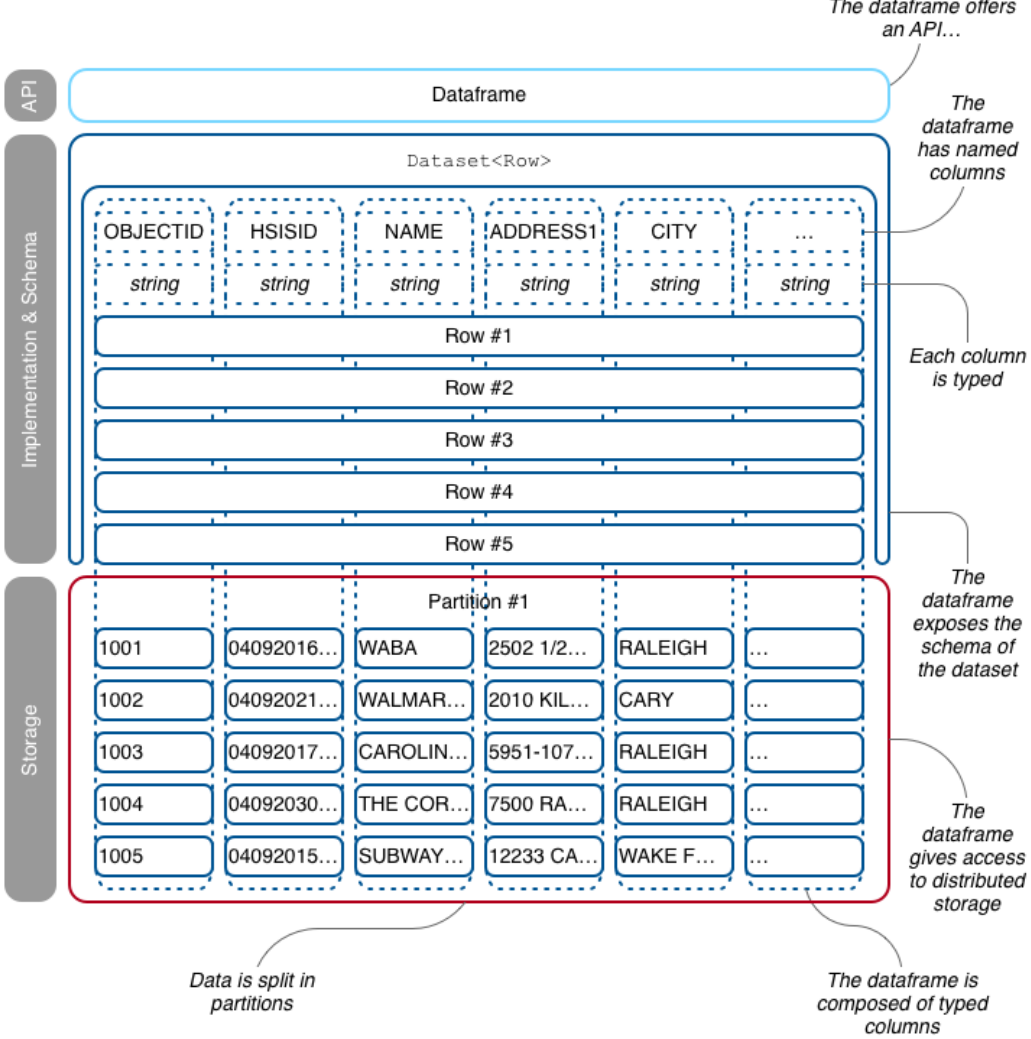
Answer: Spark partitions data in a cluster by assigning a portion of the data to each node in the cluster. There are several partitioning methods available in Spark, including:
Hash Partitioning: data is partitioned based on a hash value of a specified column
Range Partitioning: data is partitioned based on the range of values in a specified column
Round-Robin Partitioning: data is partitioned evenly across all nodes in the cluster, regardless of the values in the data



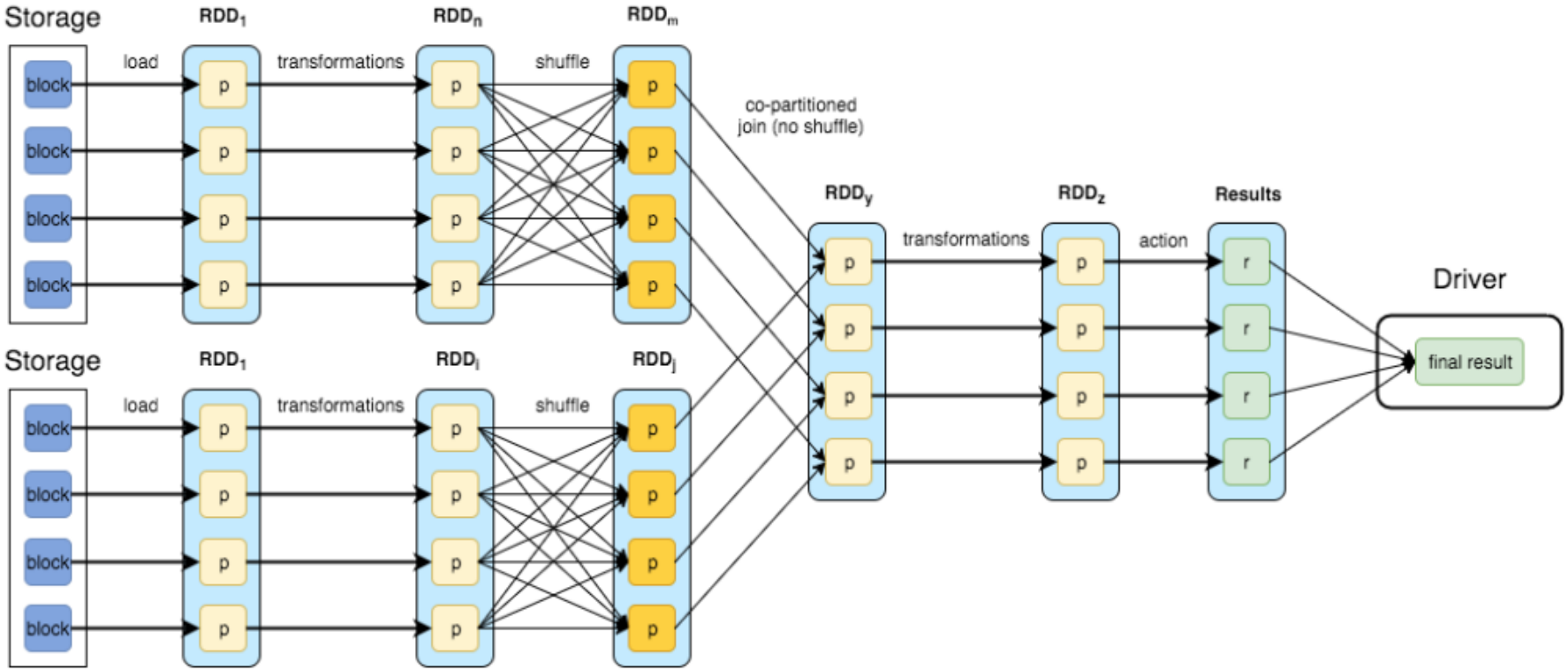
- **Too many partitions** — slower data reads
- **Too many small partitions** — waste of resource
- **Overly large partitions** can even cause executor "out of memory" errors.
- **A small number of large partitions** may leave some worker cores idle.
- **Few partitions:** long computation and write times. Also, it can cause skewed data and inefficient resource use. Skewed partition may lead to slow stage/tasks, data spilling to disk, and OOM errors.

Adaptive Query Execution

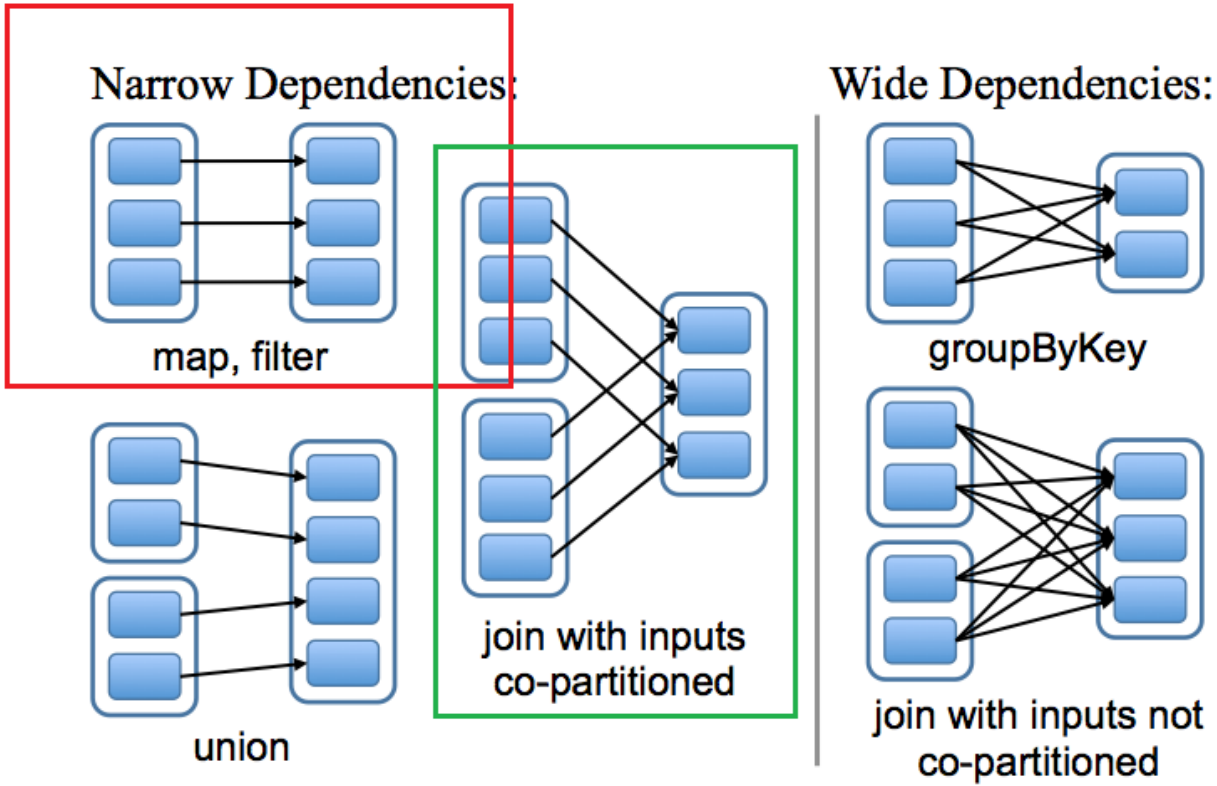



The dataframe offers an API…
API
Dataframe
The dataframe has named columns
Implementation & Schema
Dataset<Row>
OBJECTID
HSISID
NAME
ADDRESS1
CITY
…
string
string
string
string
string
string
Row #1
Row #2
Row #3
Row #4
Row #5
Each column is typed
The dataframe exposes the schema of the dataset
Storage
Partition #1
1001
04092016…
WABA
2502 1/2…
RALEIGH
…
1002
04092021…
WALMAR…
2010 KIL…
CARY
…
1003
04092017…
CAROLIN…
5951-107…
RALEIGH
…
1004
04092030…
THE COR…
7500 RA…
RALEIGH
…
1005
04092015…
SUBWAY…
12233 CA…
WAKE F…
…
The dataframe gives access to distributed storage
Data is split in partitions
The dataframe is composed of typed columns

DAG

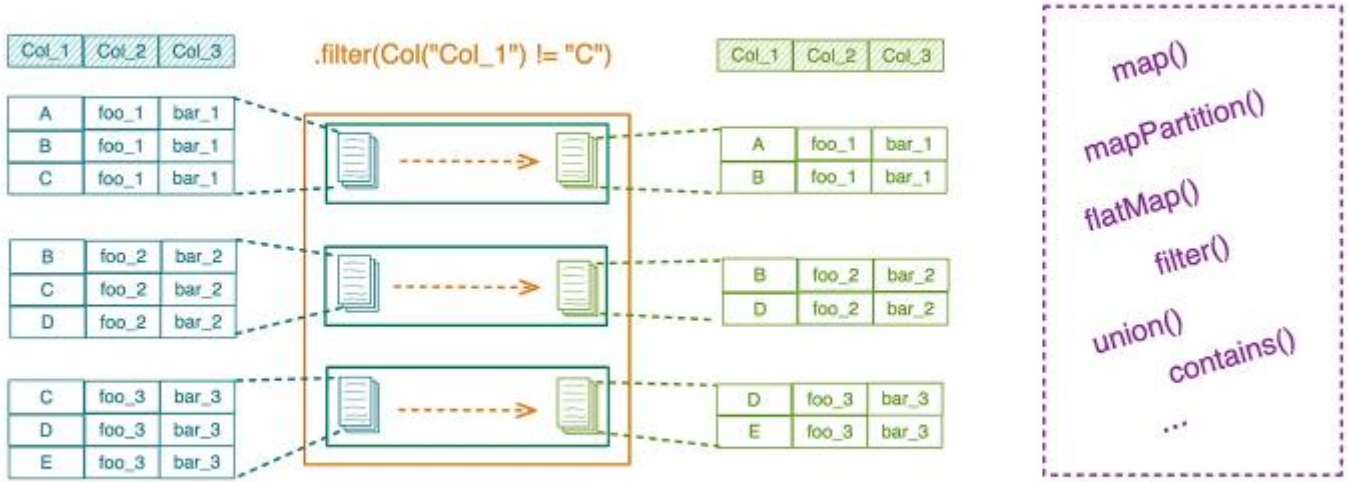

Storage
RDD1
RDDn
RDDm
load
transformations
shuffle
block
p
co-partitioned
join (no shuffle)
RDDy
RDDz
Results
transformations
action
r
Driver
final result
Storage
RDD1
RDDi
RDDj
load
transformations
shuffle

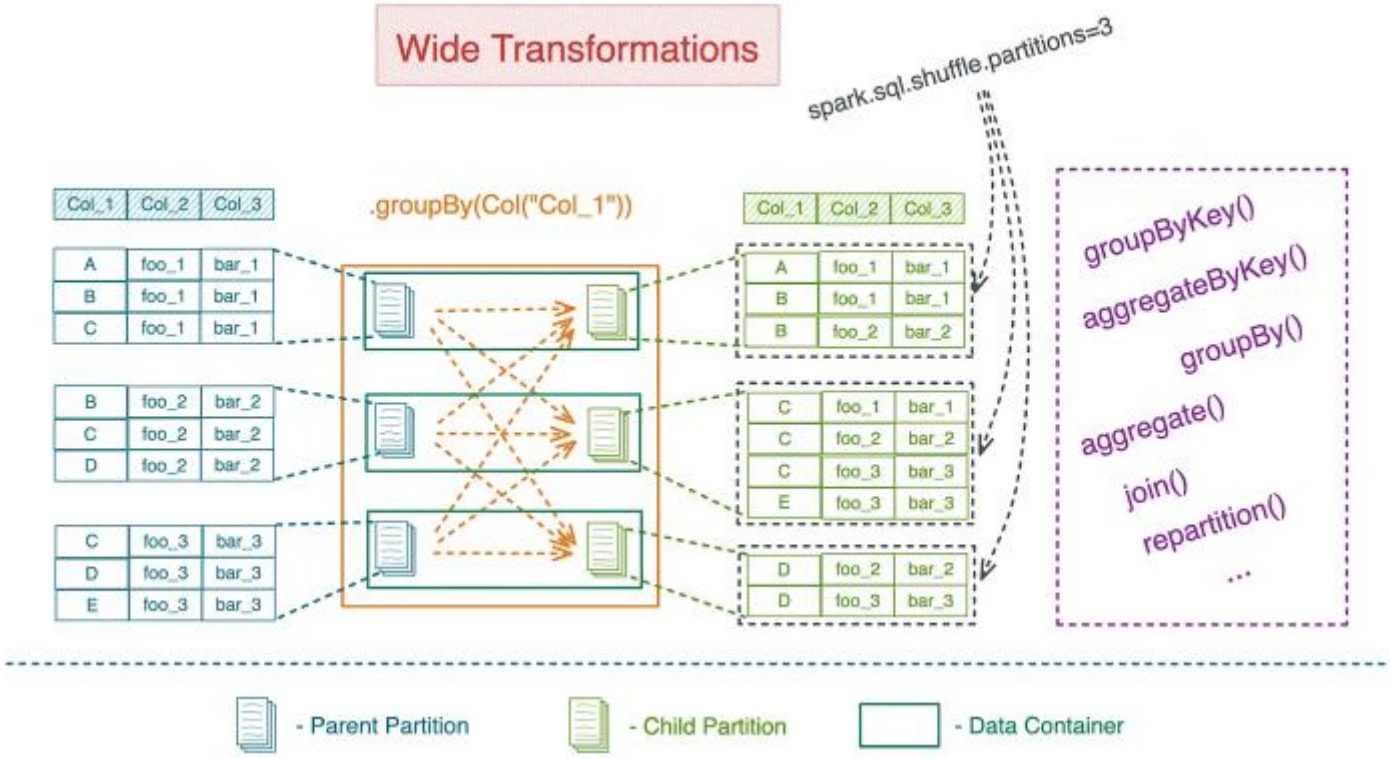
DAG

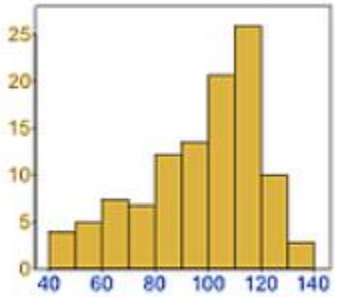


Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

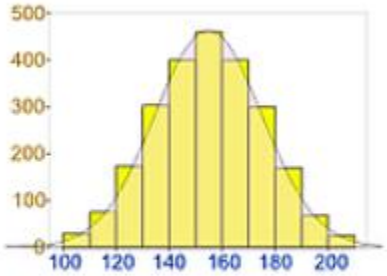
Narrow Transformations



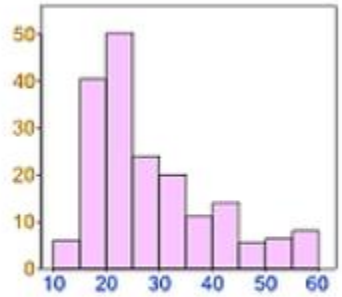

Wide Transformations
spark.sql.shuffle.partitions=3
.groupBy(Col("Col_1"))
Col_1 Col_2 Col_3
A foo_1 bar_1
B foo_1 bar_1
C foo_1 bar_1
B foo_2 bar_2
C foo_2 bar_2
D foo_2 bar_2
C foo_3 bar_3
D foo_3 bar_3
E foo_3 bar_3
Col_1 Col_2 Col_3
A foo_1 bar_1
B foo_1 bar_1
B foo_2 bar_2
C foo_1 bar_1
C foo_2 bar_2
C foo_3 bar_3
E foo_3 bar_3
D foo_2 bar_2
D foo_3 bar_3
groupByKey()
aggregateByKey()
groupBy()
aggregate()
join()
repartition()
...
- Parent Partition
- Child Partition
- Data Container



Negative Skew



No Skew



Positive Skew

Ankara

İstanbul

İzmir

Salting

For simplicity, lets say ▢ represents a record.

Ankara_1

Ankara_2

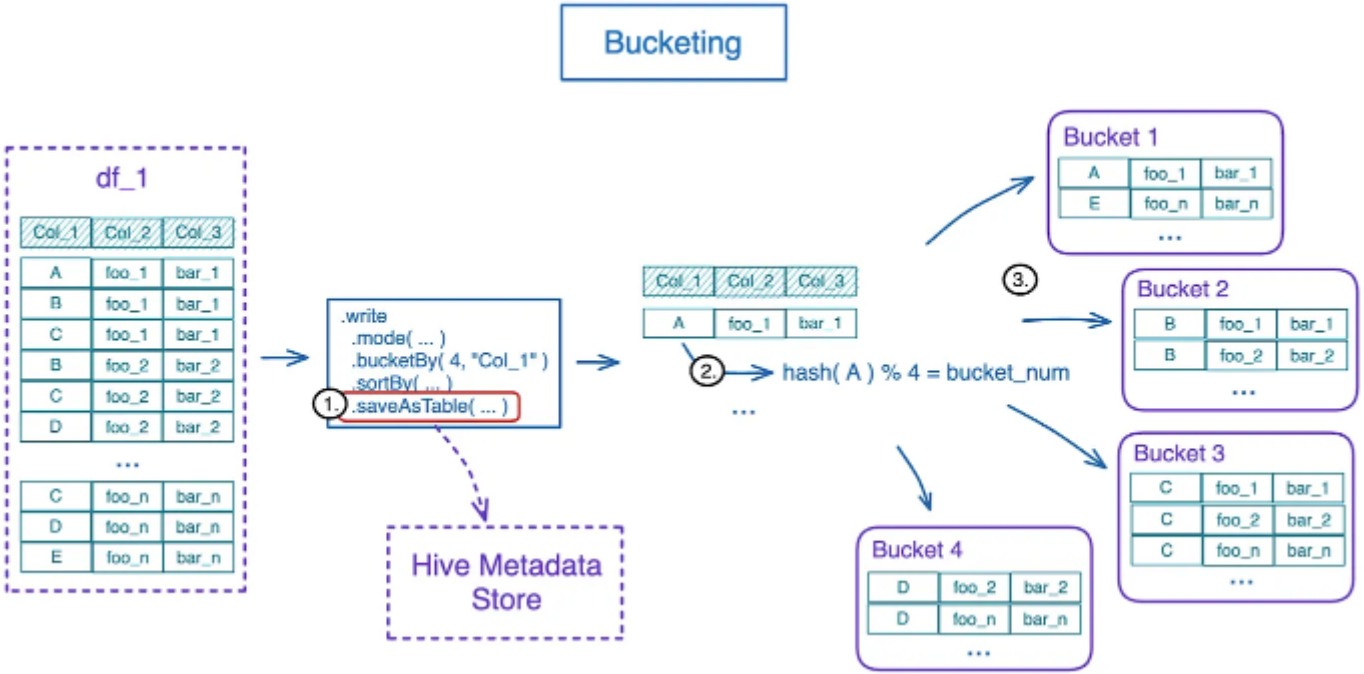
Ankara_3

Ankara_4

Ankara_5

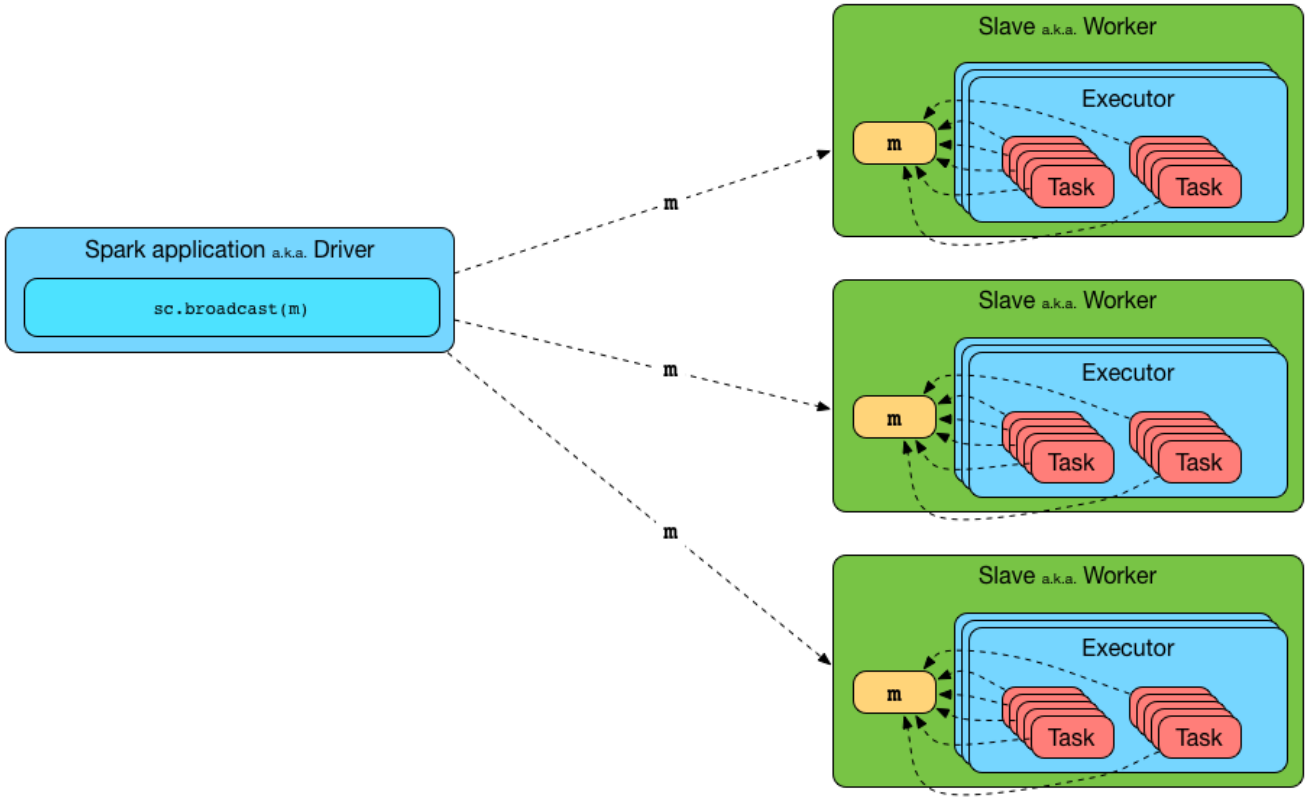
İstanbul

İzmir

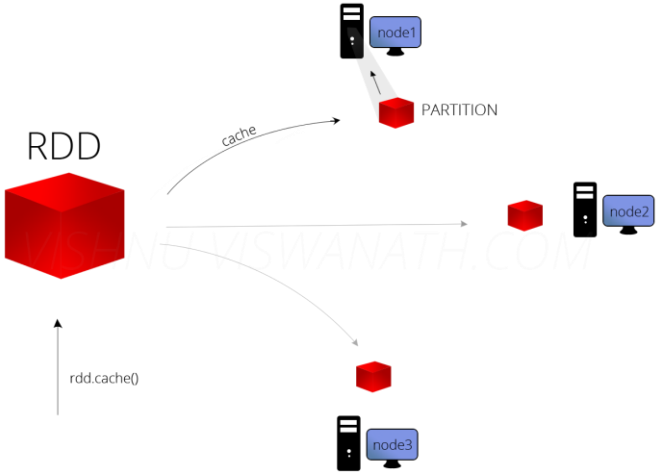


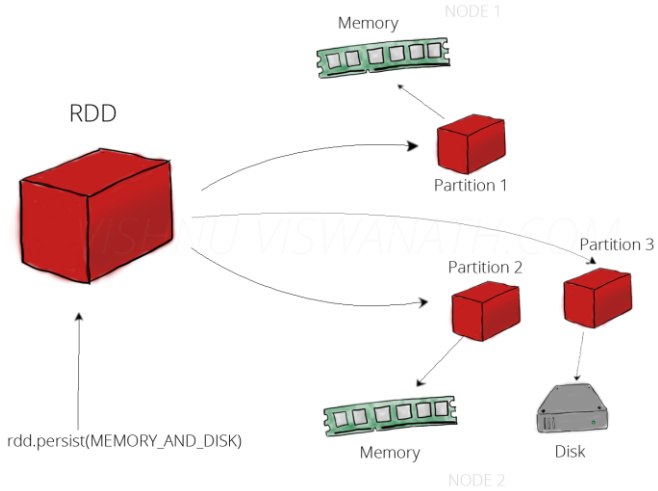
Can you describe a scenario where bucketing can lead to performance optimization in Spark? Answer: Bucketing can lead to performance optimization in Spark when joining large datasets with a skewed distribution of the join key. For example, if you are joining a large fact table with a small dimension table, bucketing the dimension table can help to reduce the amount of data that needs to be shuffled during the join operation. By bucketing the dimension table, you can ensure that the data is evenly distributed across the nodes in the cluster, which can help to reduce the amount of time required for the join operation.

Broadcast Variable Vs Accumulator



Cache vs Persist





1. MEMORY_ONLY (Default level)
2. MEMORY_AND_DISK
3. MEMORY_ONLY_SER
4. MEMORY_ONLY_DISK_SER
5. DISC_ONLY

PARQUET File Format

- Parquet, an open-source file format for Hadoop, stores nested data structures in a flat columnar format.

- Compared to a traditional approach where data is stored in a row-oriented approach, Parquet file format is more efficient in terms of storage and performance.

- It is especially good for queries that read particular columns from a “wide” (with many columns) table since only needed columns are read, and IO is minimized.

parquet

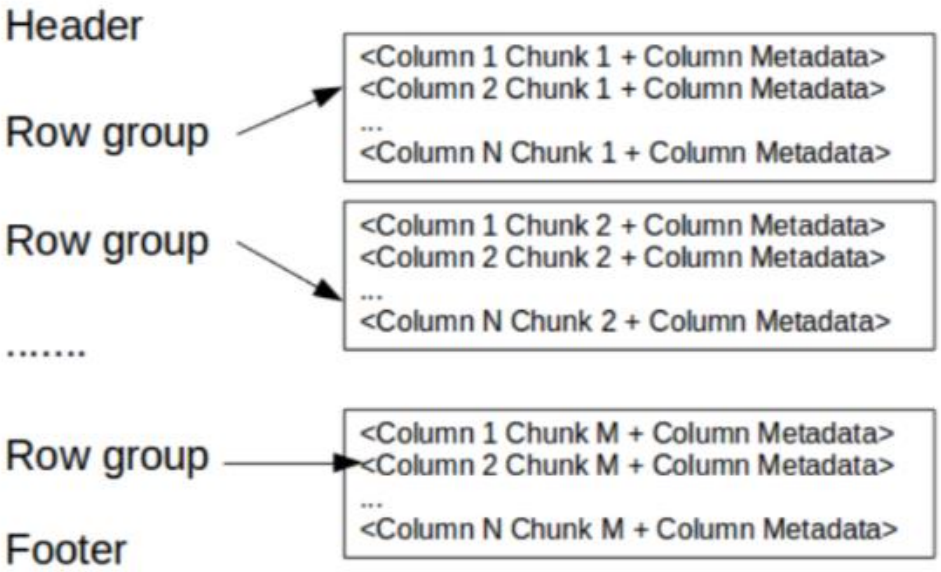
```
path
└── to
    └── table
        ├── gender=male
        │   ├── ...
        │   │
        │   ├── country=US
        │   │   └── data.parquet
        │   ├── country=CN
        │   │   └── data.parquet
        │   └── ...
        └── gender=female
            ├── ...
            │
            ├── country=US
            │   └── data.parquet
            ├── country=CN
            │   └── data.parquet
            └── ...
```

PARQUET File Format

- Support Predicate push down
- column-oriented format increases the query performance as less seek time is required to go to the required columns, and less IO is required as it needs to read only the columns whose data are required.
- One of the unique features of Parquet is that it can store data with nested structures in a columnar fashion too. This means that in a Parquet file format, even the nested fields can be read individually without reading all the fields in the nested structure.

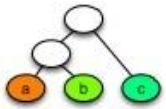
PARQUET File Format

- Row group: A logical horizontal partitioning of the data into rows. A row group consists of a column chunk for each column in the dataset.
- Column chunk: A chunk of the data for a particular column. These column chunks live in a particular row group and are guaranteed to be contiguous in the file.
- Page: Column chunks are divided up into pages written back to back. The pages share a standard header and readers can skip the page they are not interested in.



PARQUET File Format

Columnar storage

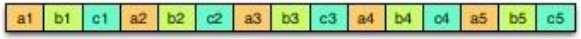


Nested schema

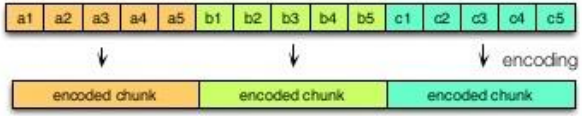
Logical table representation

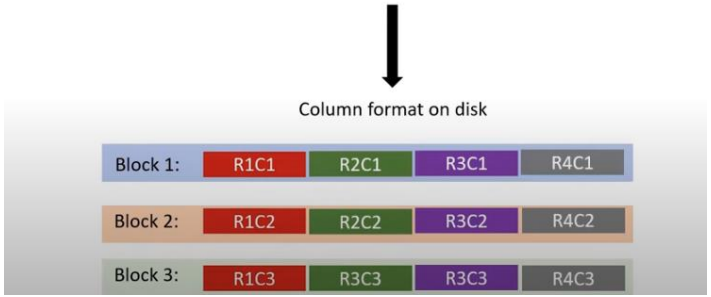
a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

Row layout



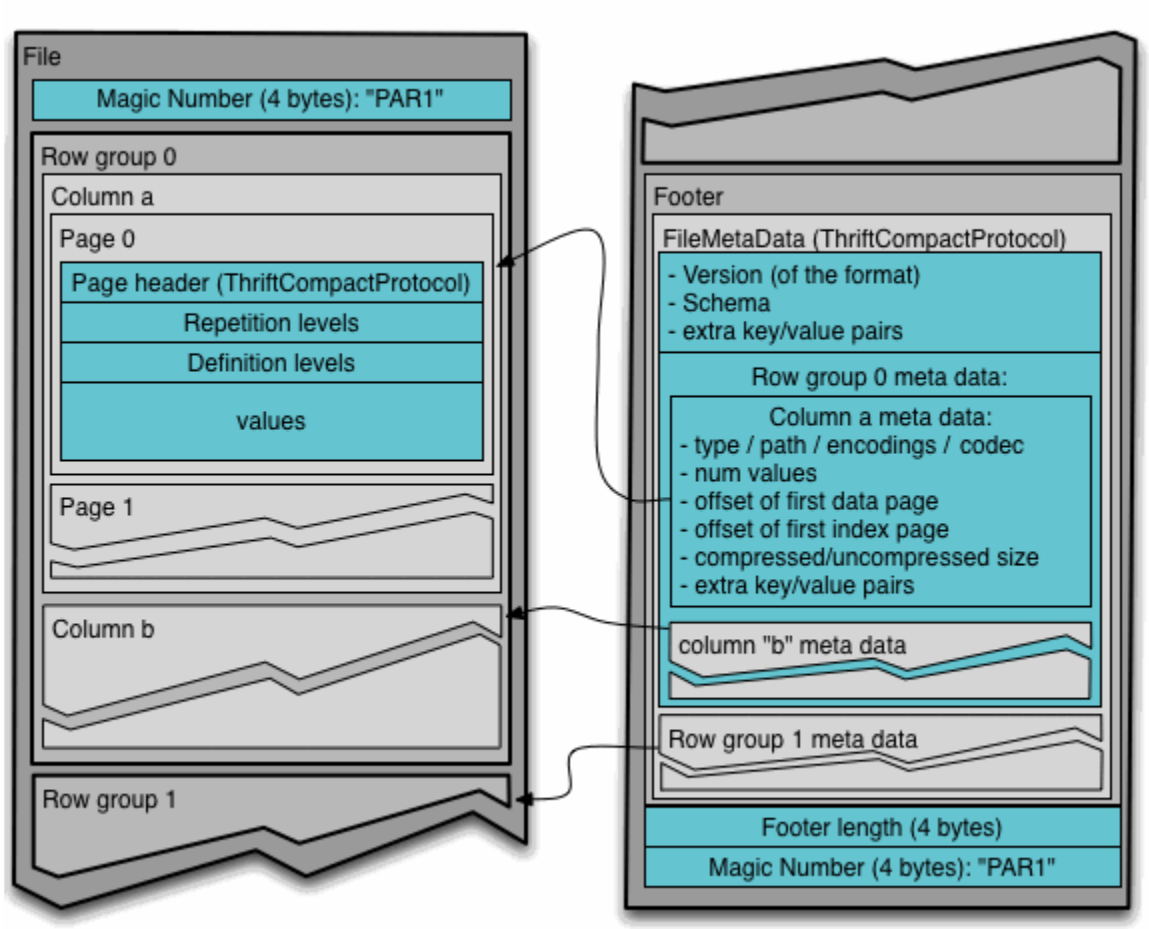
Column layout





select col1, col2 from table where col1 in (R2C1, R3C1)

PARQUET File Format



AVRO vs. PARQUET

1. AVRO is a row-based storage format, whereas PARQUET is a columnar-based storage format.
2. PARQUET is much better for analytical querying, i.e., reads and querying are much more efficient than writing.
3. Writiing operations in AVRO are better than in PARQUET.
4. AVRO is much matured than PARQUET when it comes to schema evolution. PARQUET only supports schema append, whereas AVRO supports a much-featured schema evolution, i.e., adding or modifying columns.
5. PARQUET is ideal for querying a subset of columns in a multi-column table. AVRO is ideal in the case of ETL operations, where we need to query all the columns.

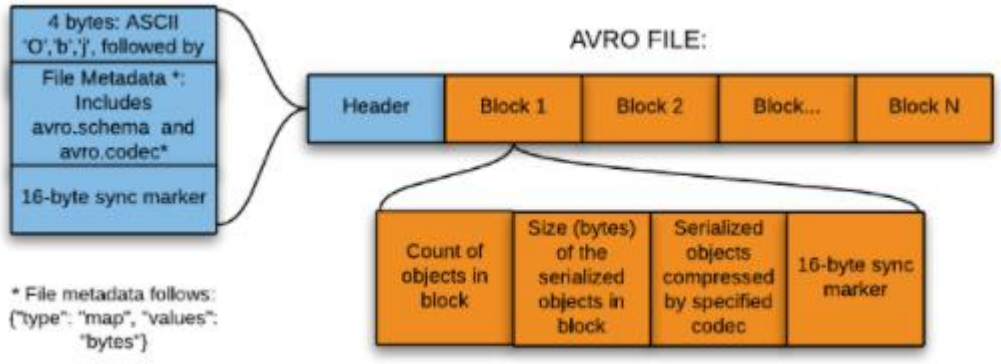
Storage layout difference between row- and column-oriented formats

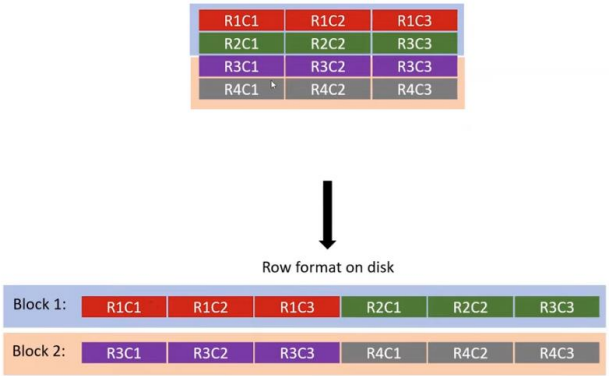
AVRO File Format

- Avro format is a row-based storage format for Hadoop, which is widely used as a serialization platform.
- Avro format stores the schema in JSON format, making it easy to read and interpret by any program.
- Avro format is a language-neutral data serialization system. It can be processed by many languages (currently C, C++, C#, Java, Python, and Ruby).
- A key feature of Avro format is the robust support for data schemas that changes over time, i.e., schema evolution. Avro handles schema changes like missing fields, added fields, and changed fields.

AVRO File Format

- Avro format provides rich data structures. For example, you can create a record that contains an array, an enumerated type, and a sub-record.





JSON
(Schemaless)

```
{user_id: 53,
timestamp: 1497842472,
address: "2 Elm St. Chattanooga, TN"}
```

74 bytes

Schema +
Avro Payload

```
{   "type": "record",
    "name": "Person",
    "fields": [
        {"name": "user_id",   "type": "long"},
        {"name": "timestamp", "type": "long"},
        {"name": "address",   "type": "string"}]}
```

204 bytes

35	59474328	19

```
3220456c6d2053
742e2043686174
74616e6f6f6761
    2c20544e
```

34 bytes

Schema ID +
Avro Payload

21

35	59474328	19

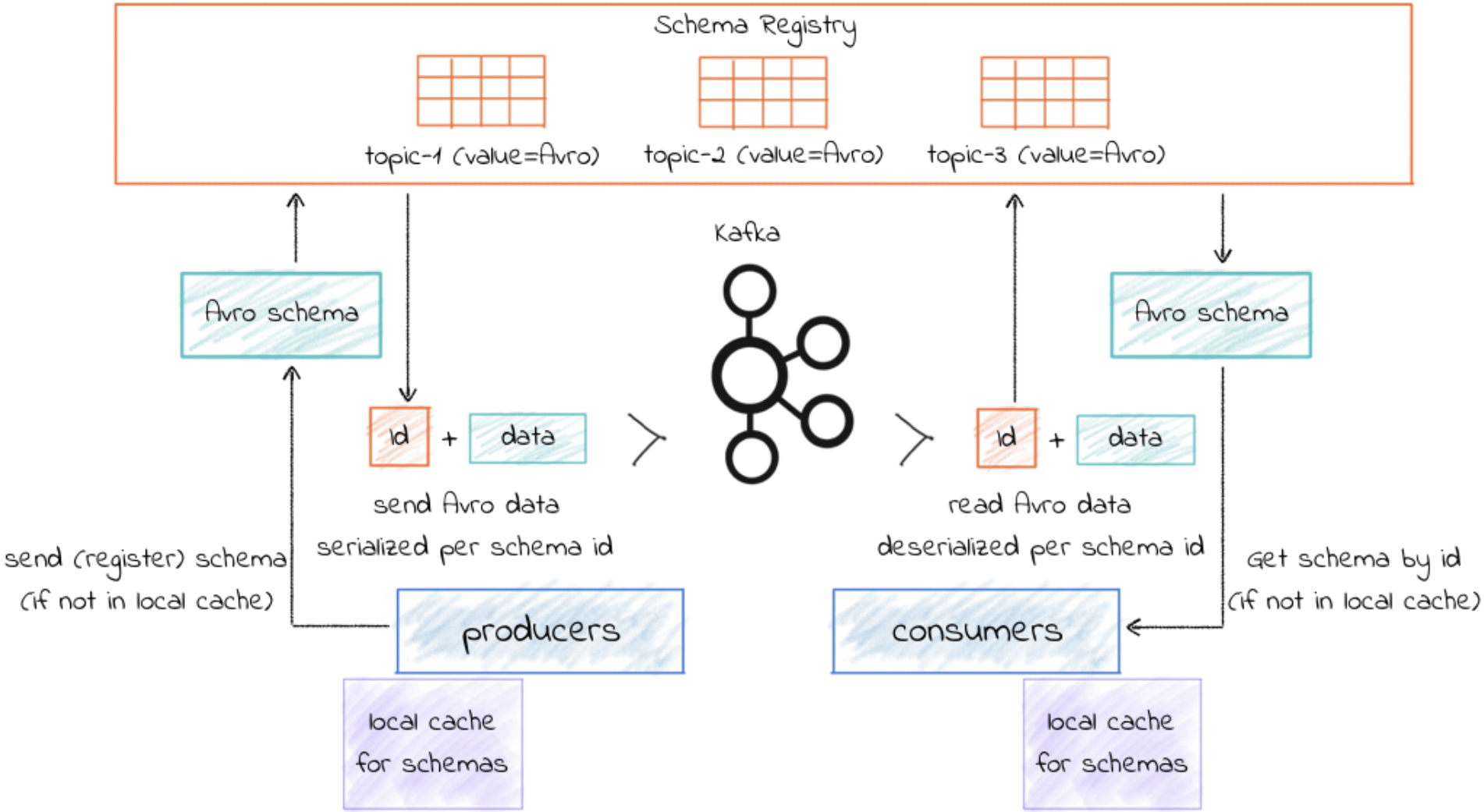
```
3220456c6d2053
742e2043686174
74616e6f6f6761
    2c20544e
```

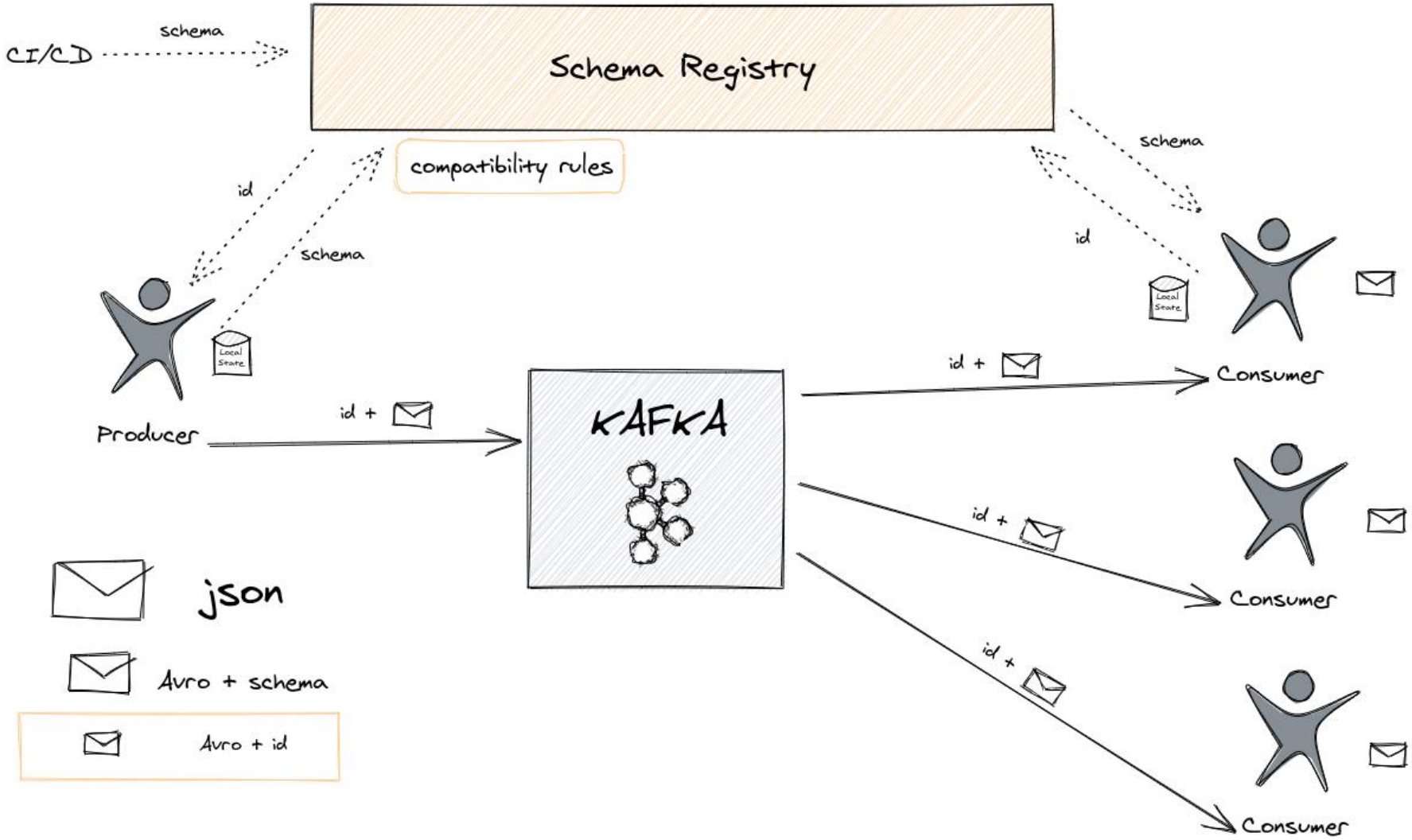
4 bytes

34 bytes

AVRO File Format

- The Avro format is the ideal candidate for storing data in a data lake landing zone because:
- Data from the landing zone is usually read as a whole for further processing by downstream systems (the row-based format is more efficient in this case).
- Downstream systems can easily retrieve table schemas from Avro files (there is no need to store the schemas separately in an external meta store).
- Any source schema change is easily handled (schema evolution).


Schema Registry
topic-1 (value=Avro)
topic-2 (value=Avro)
topic-3 (value=Avro)
Kafka
Avro schema
Avro schema
id
+
data
id
+
data
send Avro data
serialized per schema id
read Avro data
deserialized per schema id
send (register) schema
(if not in local cache)
Get schema by id
(if not in local cache)
producers
consumers
local cache
for schemas
local cache
for schemas


schema
CI/CD
Schema Registry
compatibility rules
id
schema
schema
id
Local State
Local State
Producer
id +
KAFKA
id +
Consumer
id +
Consumer
id +
Consumer
json
Avro + schema
Avro + id

Connected Car Usecase

Personal Data of Drivers
- driver ID (license #)
- name
- home address
- email address
- credit card

who is speeding at night? (20% and more above limit)

Usage-based Insurance Service

Raise premium, Notify by email

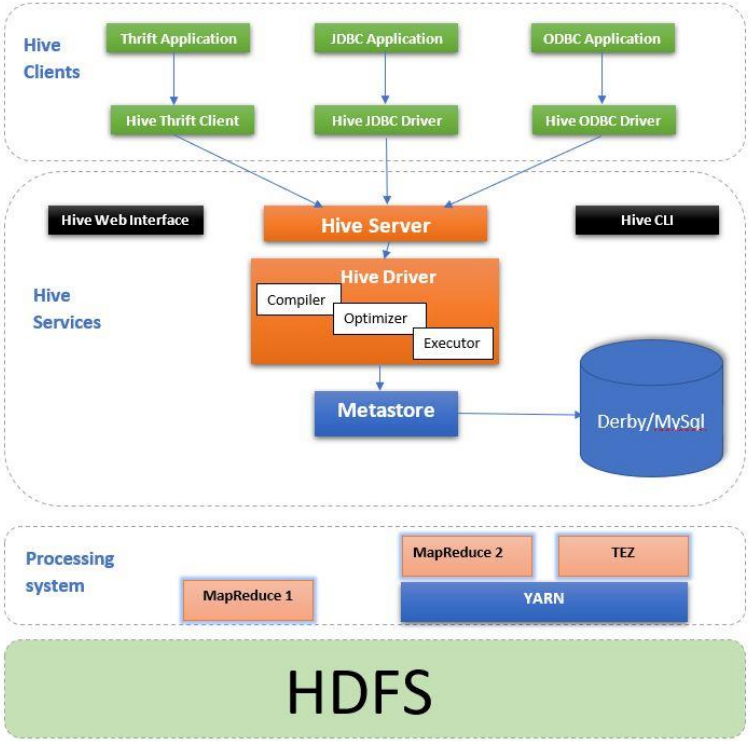
Car Monitoring Events
- car ID (plate #)
- driver ID (license #)
- speed
- location

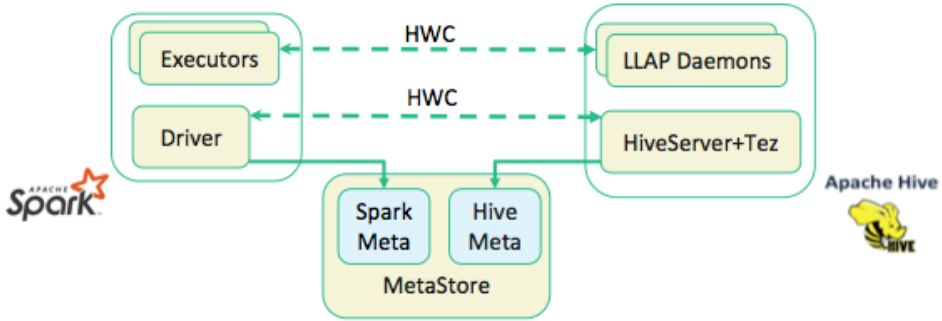
Gateway

Spark

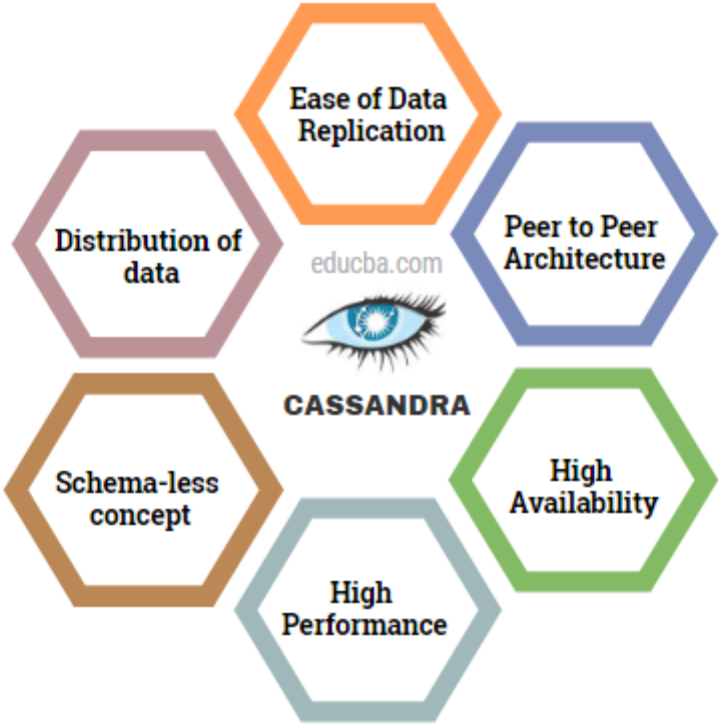
Parquet

Hive

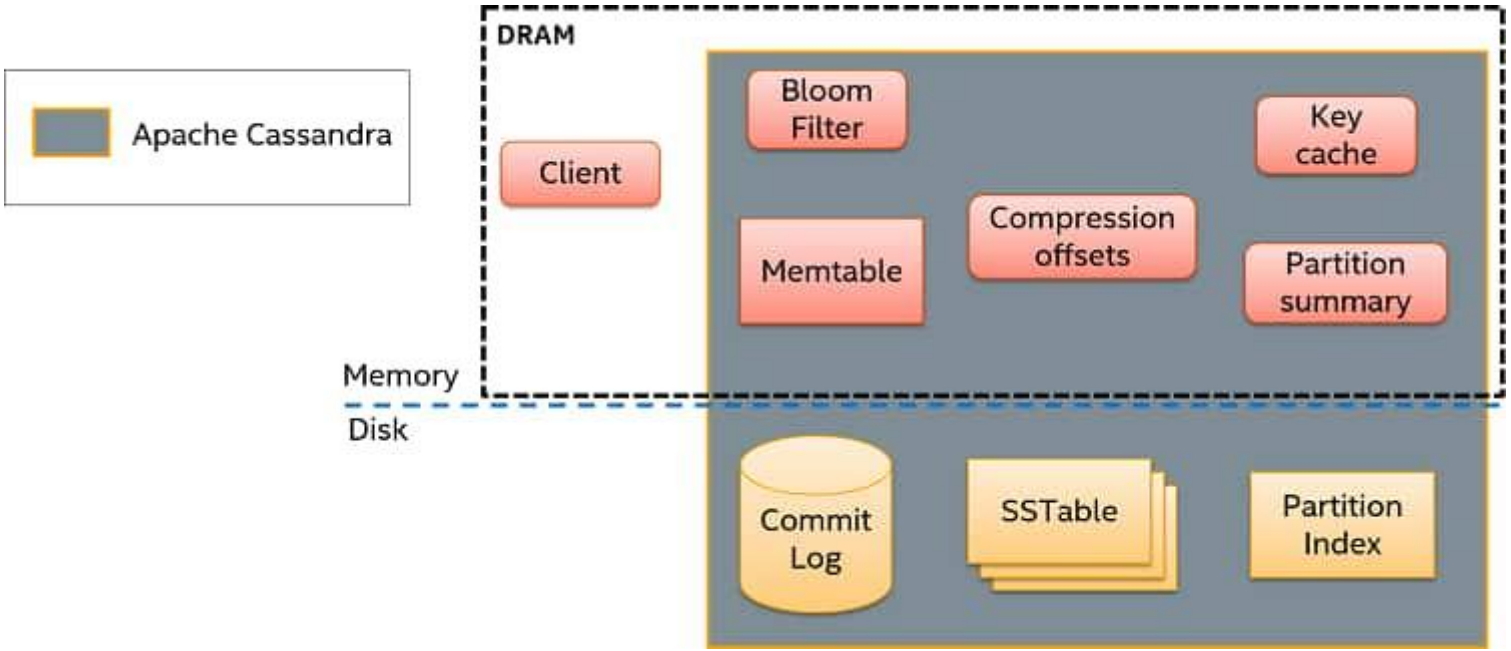

Hive Clients
Thrift Application
JDBC Application
ODBC Application
Hive Thrift Client
Hive JDBC Driver
Hive ODBC Driver
Hive Web Interface
Hive Server
Hive CLI
Hive Services
Hive Driver
Compiler
Optimizer
Executor
Metastore
Derby/MySql
Processing system
MapReduce 1
MapReduce 2
TEZ
YARN
HDFS


Executors
Driver
HWC
HWC
LLAP Daemons
HiveServer+Tez
Spark Meta
Hive Meta
MetaStore
Spark
Apache Hive

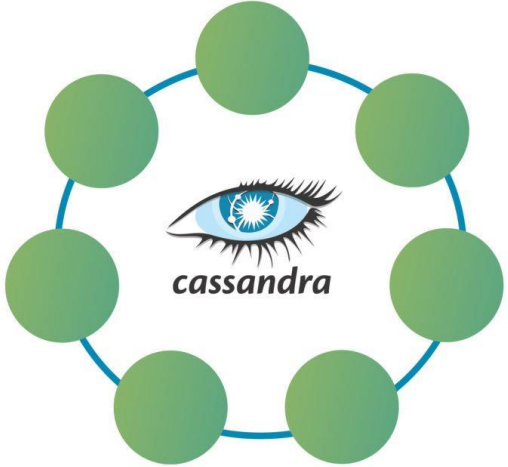
What is Cassandra



Cassandra

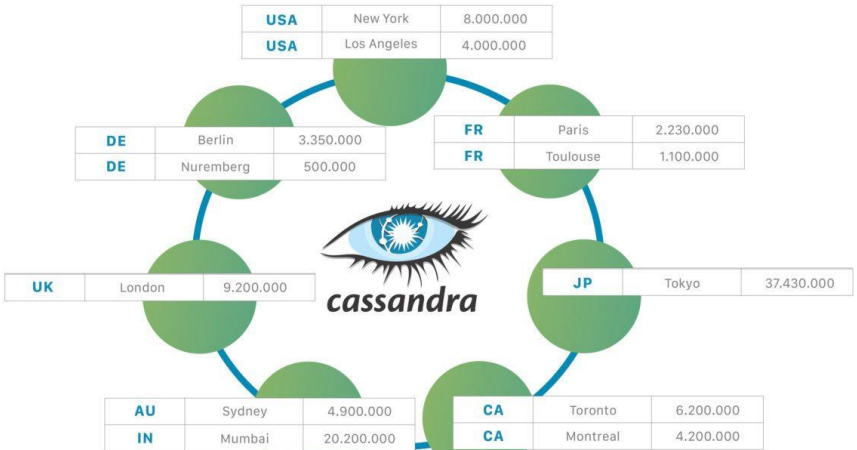

Apache Cassandra
DRAM
Client
Bloom Filter
Key cache
Compression offsets
Memtable
Partition summary
Memory
Disk
Commit Log
SSTable
Partition Index

Cassandra



COUNTRY	CITY	POPULATION
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000

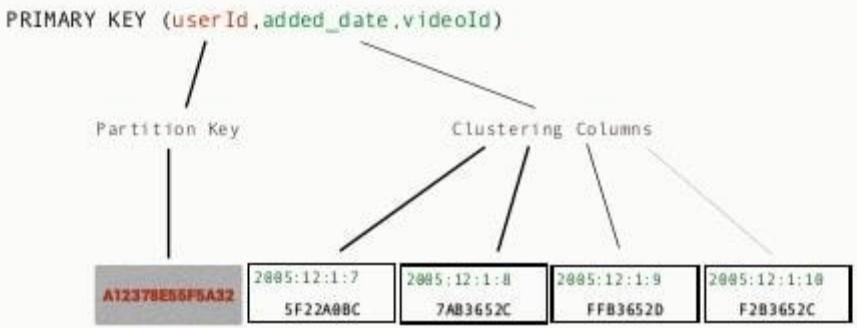
Partition Key



Cassandra

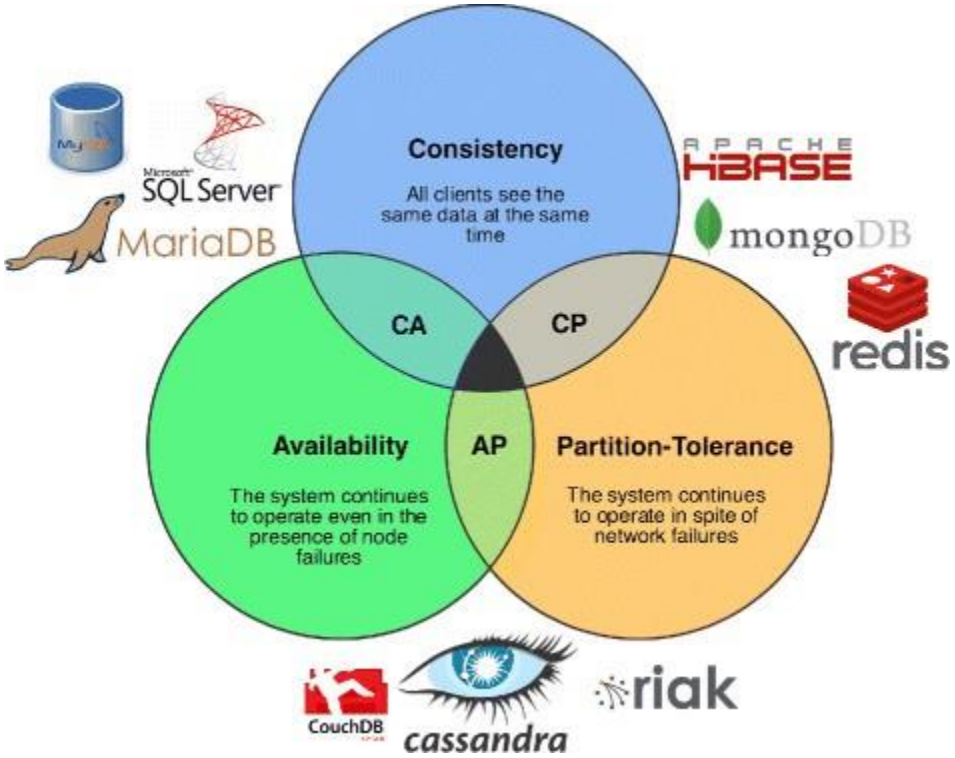
Primary key relationship

DATASTAX

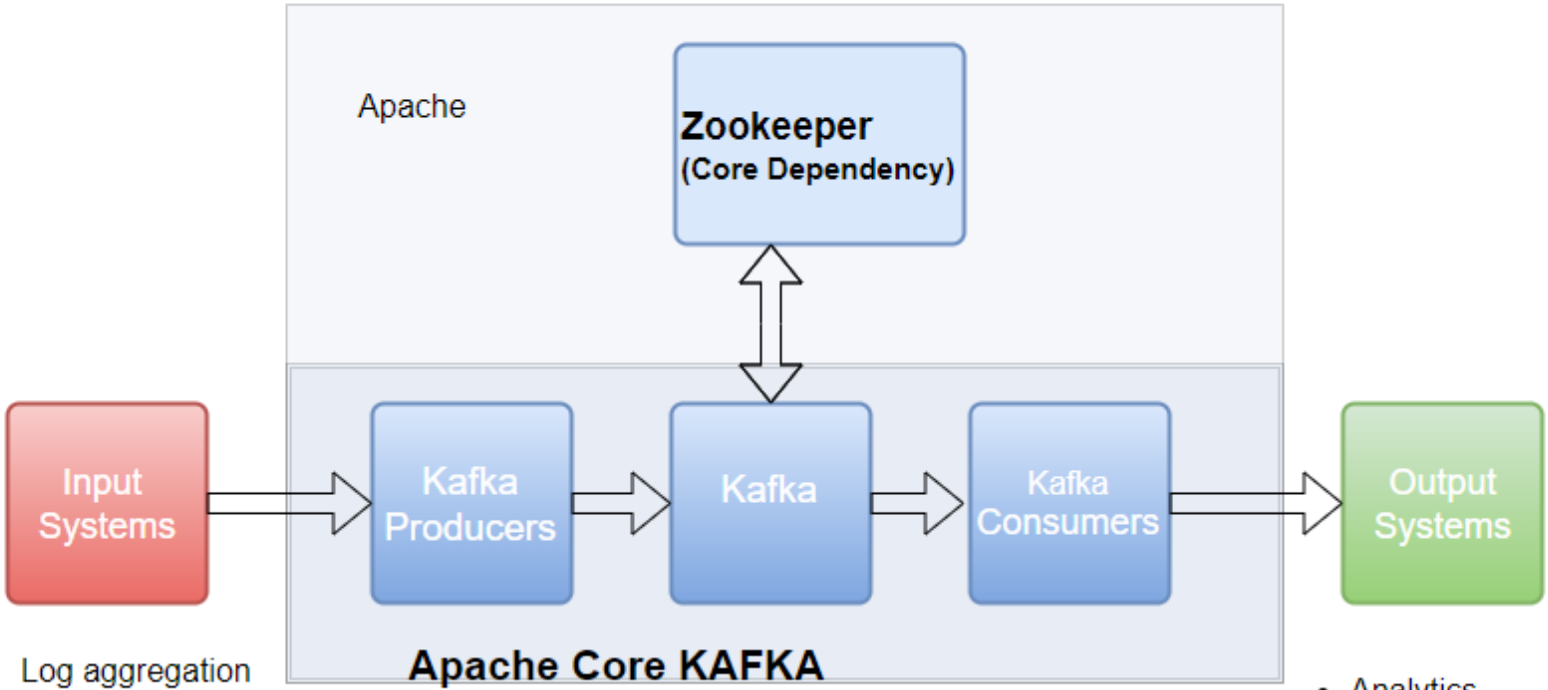



```
SELECT videoId FROM user_videos
WHERE userId = A12378E55F5A32
AND added_date = '2005-12-1'
AND videoId = 5F22A0BC
```

CAP Theorm


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

Kafka Architecture: Core Kafka



- Log aggregation
- Metrics
- KPIs
- Batch imports
- Audit trail
- User activity logs
- Web logs

Not part of core

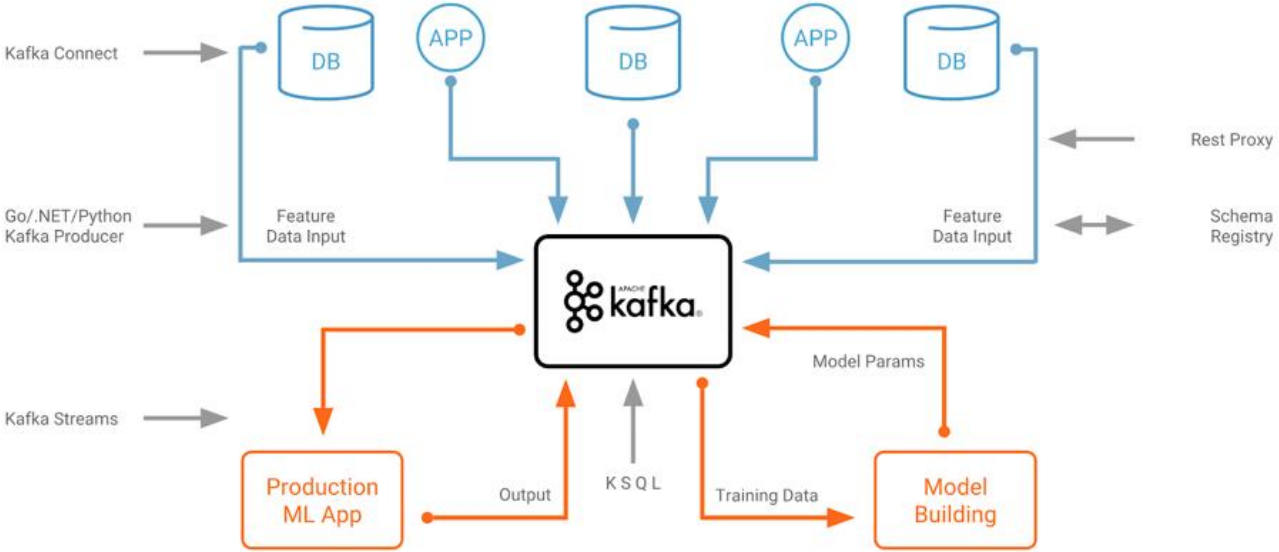
- Schema Registry
- Avro
- Kafka REST Proxy
- Kafka Connect
- Kafka Streams

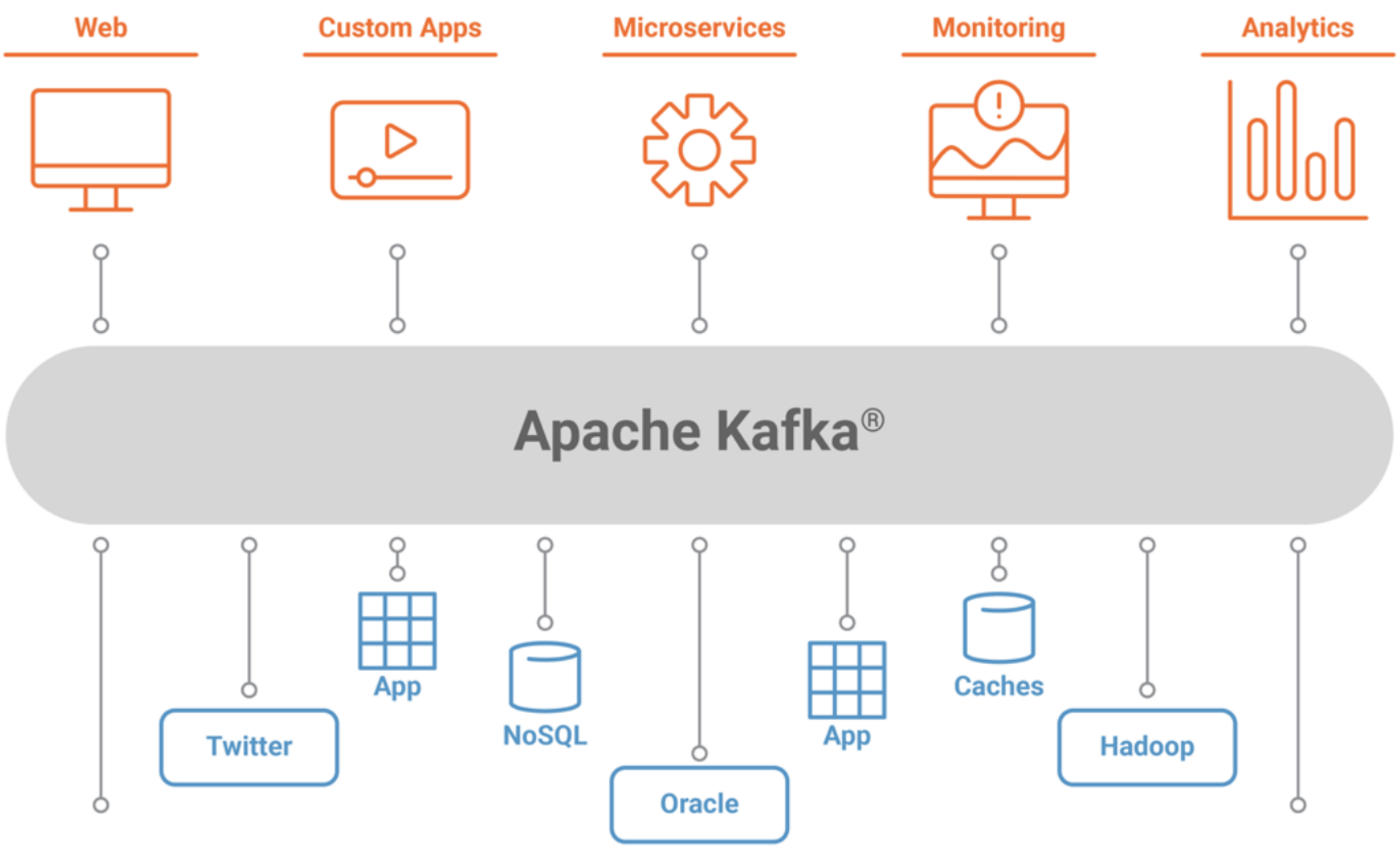
Apache Kafka Core

- Server/Broker
- Scripts to start libs
- Script to start up Zookeeper
- Utils to create topics
- Utils to monitor stats

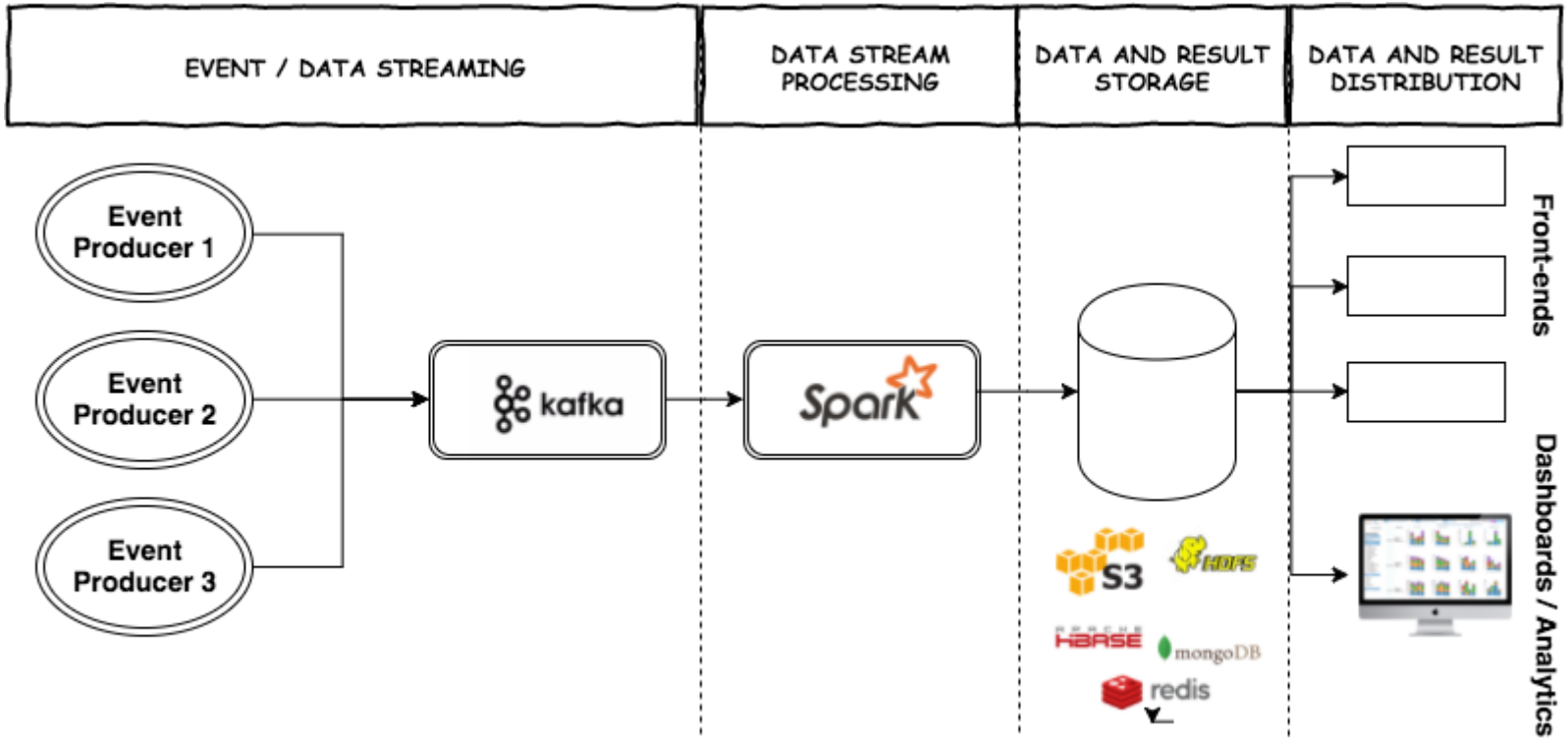
- Analytics
- Databases
- Machine Learning
- Dashboards
- Indexed for Search
- Business Intelligene

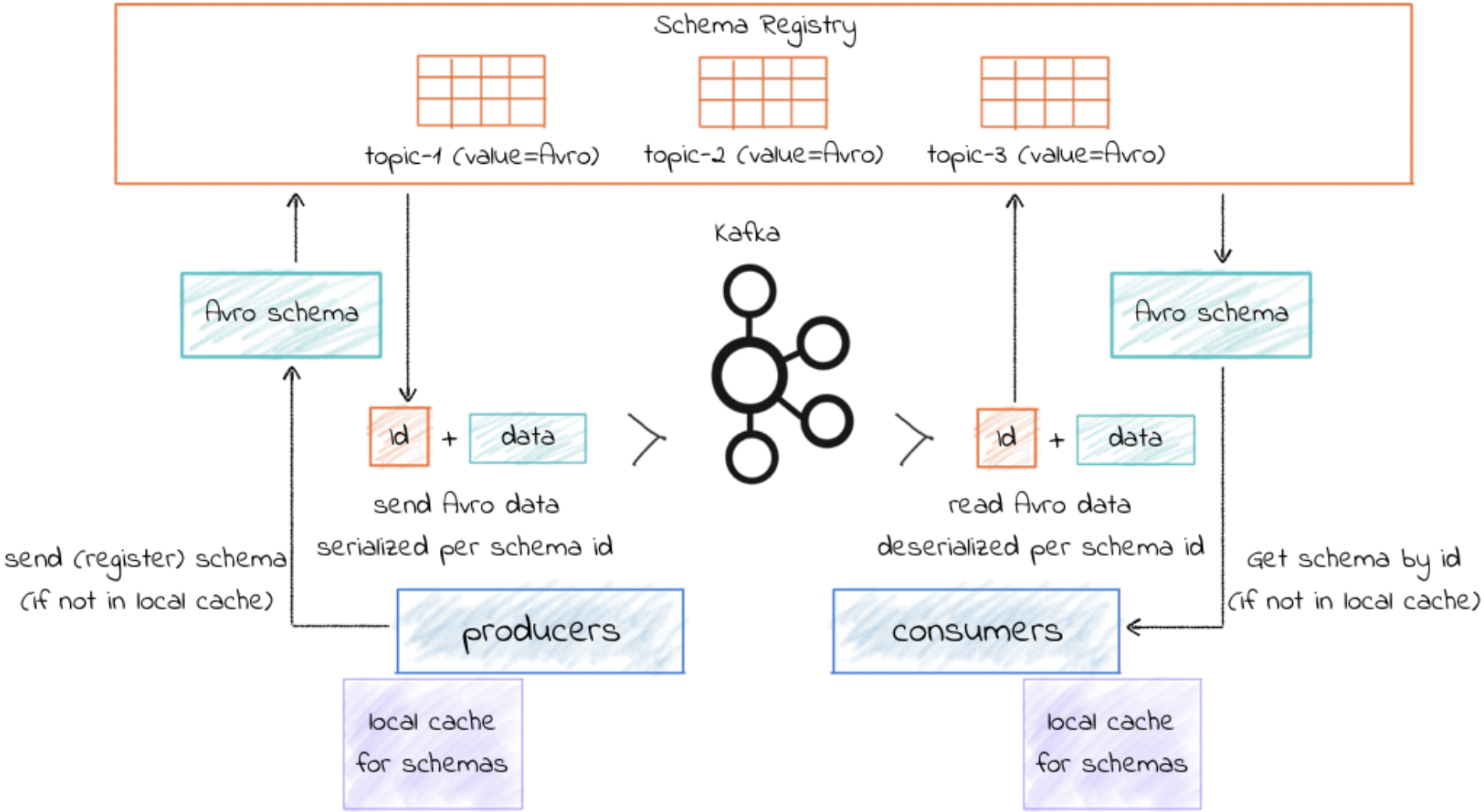
Kafka

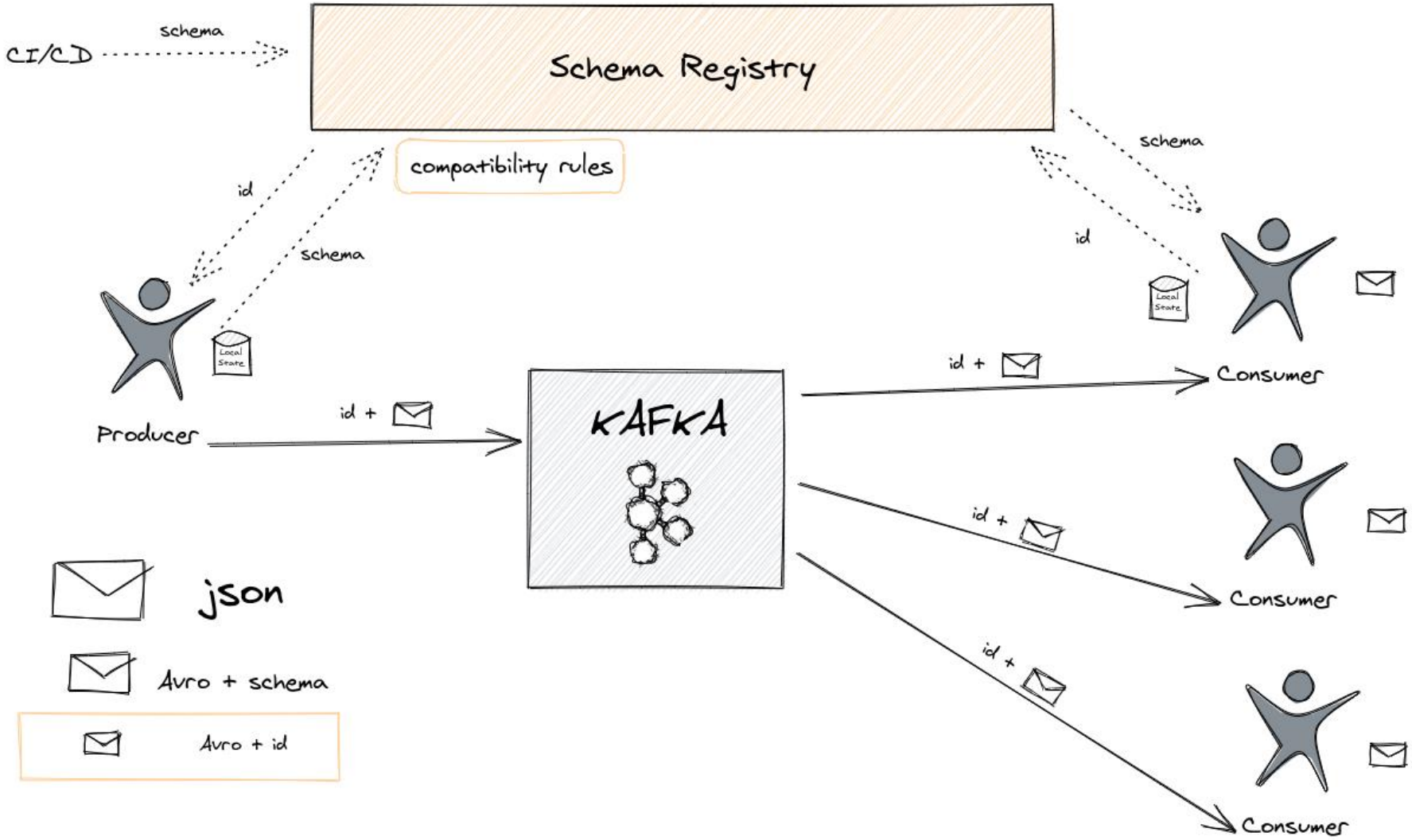

Kafka Connect
DB
APP
DB
APP
DB
Rest Proxy
Go/.NET/Python
Kafka Producer
Feature
Data Input
Feature
Data Input
Schema
Registry
kafka
Model Params
Kafka Streams
Production
ML App
Output
KSQL
Training Data
Model
Building

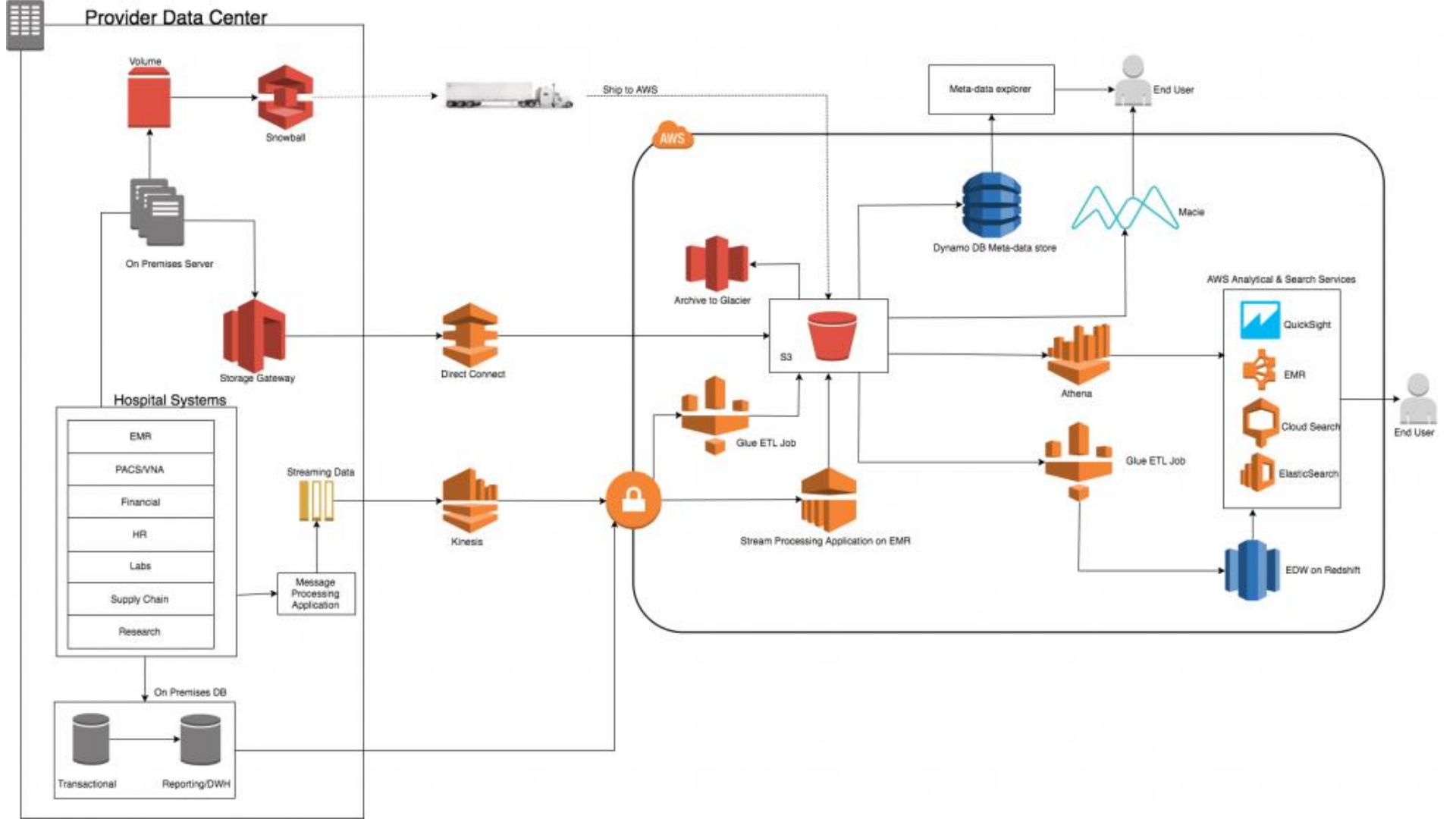

Web
Custom Apps
Microservices
Monitoring
Analytics
Apache Kafka®
Twitter
App
NoSQL
Oracle
App
Caches
Hadoop

Kafka


EVENT / DATA STREAMING
DATA STREAM PROCESSING
DATA AND RESULT STORAGE
DATA AND RESULT DISTRIBUTION
Event Producer 1
Event Producer 2
Event Producer 3
kafka
Spark
S3
HDFS
HBASE
mongoDB
redis
Front-ends
Dashboards / Analytics

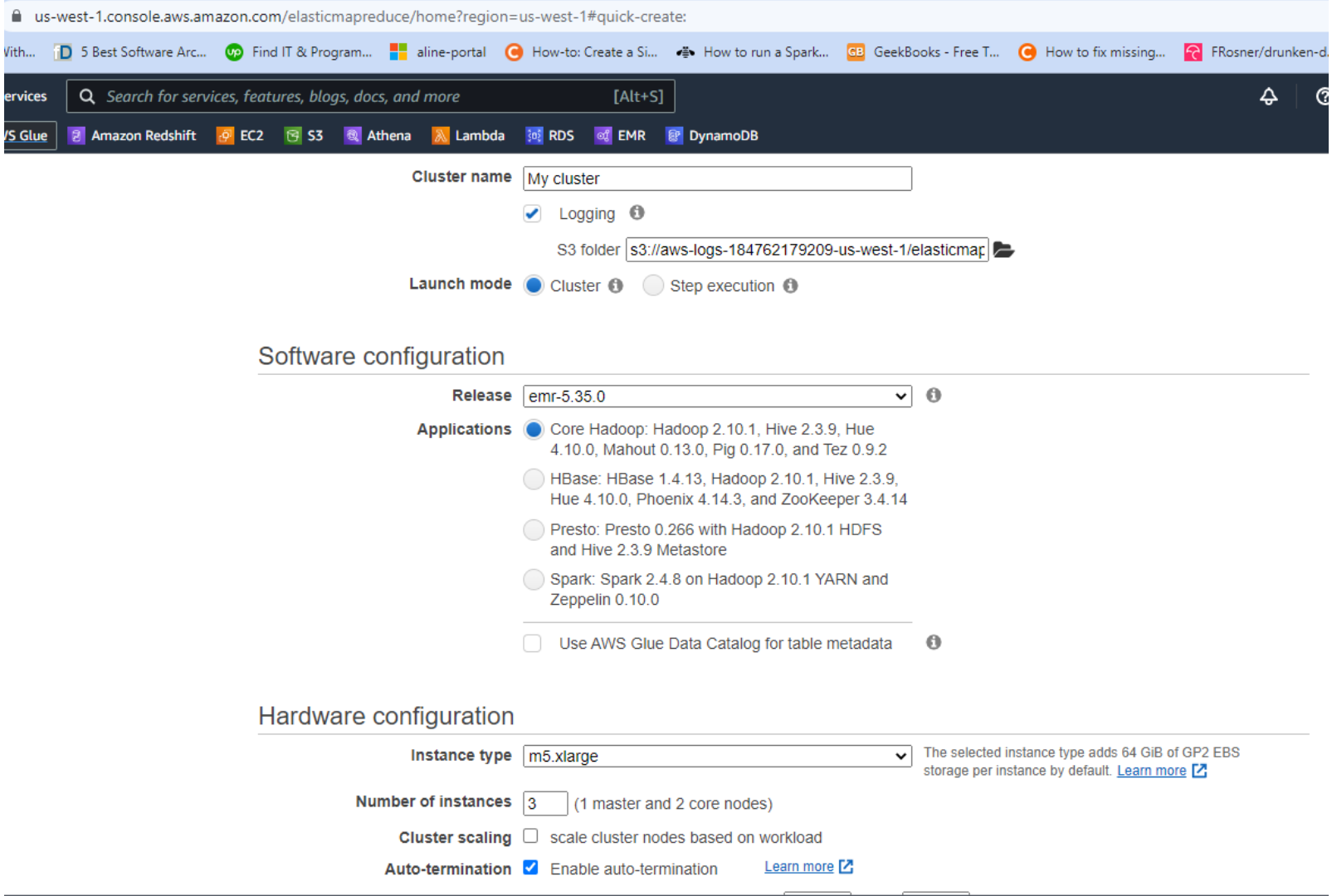

Schema Registry
topic-1 (value=Avro)
topic-2 (value=Avro)
topic-3 (value=Avro)
Kafka
Avro schema
Avro schema
id
+
data
id
+
data
send Avro data
serialized per schema id
read Avro data
deserialized per schema id
send (register) schema
(if not in local cache)
Get schema by id
(if not in local cache)
producers
consumers
local cache
for schemas
local cache
for schemas


schema
CI/CD
Schema Registry
compatibility rules
id
schema
schema
id
Local State
Local State
Producer
id +
KAFKA
id +
Consumer
id +
Consumer
id +
Consumer
json
Avro + schema
Avro + id

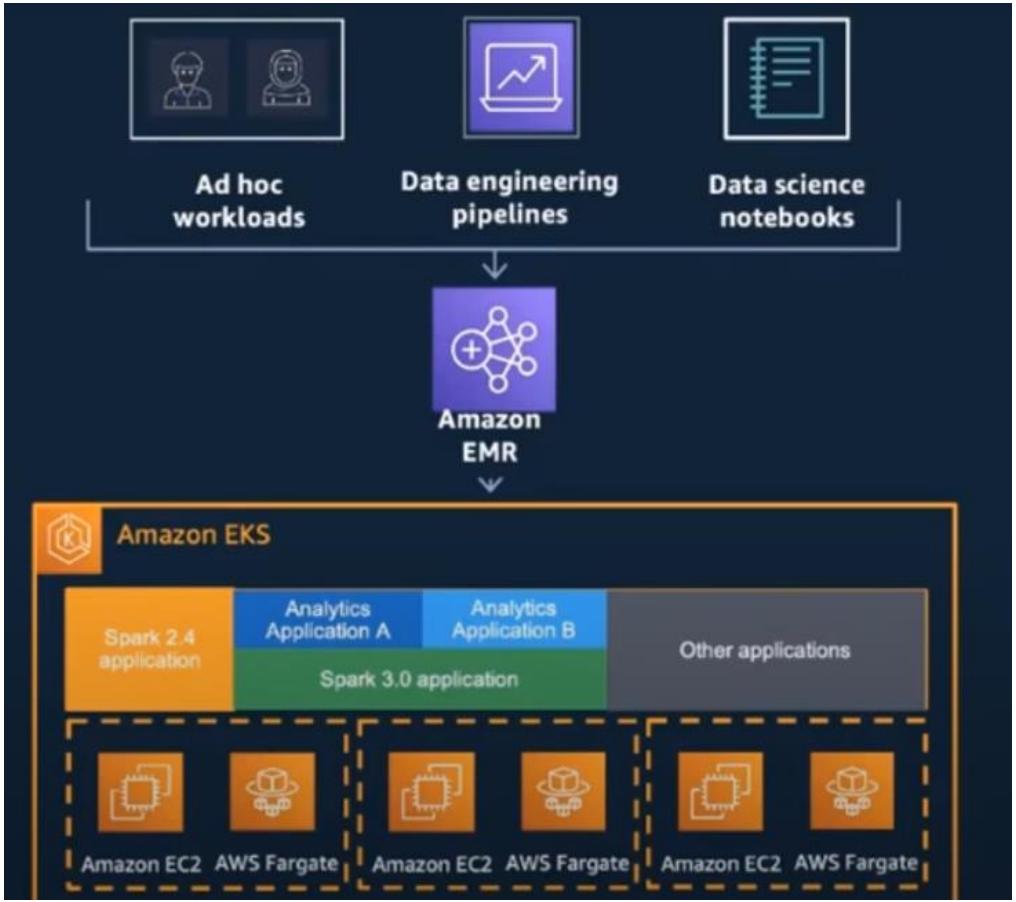

Provider Data Center
Volume
Snowball
Ship to AWS
Meta-data explorer
End User
AWS
On Premises Server
Dynamo DB Meta-data store
Macie
Archive to Glacier
AWS Analytical & Search Services
QuickSight
EMR
Cloud Search
ElasticSearch
Storage Gateway
Direct Connect
S3
Athena
End User
Hospital Systems
EMR
PACS/VNA
Financial
HR
Labs
Supply Chain
Research
Glue ETL Job
Streaming Data
Kinesis
Glue ETL Job
Stream Processing Application on EMR
Message Processing Application
EDW on Redshift
On Premises DB
Transactional
Reporting/DWH

EMR - CLI

- AWS CLI
  - aws emr create-default-roles
  - aws ec2 describe-subnets \
    --filters "Name=availabilityZone,Values=us-east-2b“
  - aws emr create-cluster --name “Cluster” --release-label emr-5.9.0 \
    --instance-type m4.large — instance-count 3 --applications Name=Spark \
    --steps Type=Spark,Name=”Spark Program”,ActionOnFailure=CONTINUE,Args=[—class,com.jeanr84.sparkjob.SparkJob,s3://my-second-emrbucket/tutorialEMR/spark-job-0.0.1-SNAPSHOT.jar,s3://my-second-emrbucket/tutorialEMR/input.txt,s3://my-second-emrbucket/tutorialEMR/output] \
    --use-default-roles \
    --ec2-attributes SubnetId=subnet-e7ba9d9c \
    --auto-terminate
  - $ aws s3 sync s3://my-second-emr-bucket/tutorialEMR/output output


us-west-1.console.aws.amazon.com/elasticmapreduce/home?region=us-west-1#quick-create:
5 Best Software Arc...
Find IT & Program...
aline-portal
How-to: Create a Si...
How to run a Spark...
GeekBooks - Free T...
How to fix missing...
FRosner/drunken-d...
Search for services, features, blogs, docs, and more
[Alt+S]
Amazon Redshift
EC2
S3
Athena
Lambda
RDS
EMR
DynamoDB
Cluster name
My cluster
Logging
S3 folder
s3://aws-logs-184762179209-us-west-1/elasticmap
Launch mode
Cluster
Step execution
Software configuration
Release
emr-5.35.0
Applications
Core Hadoop: Hadoop 2.10.1, Hive 2.3.9, Hue 4.10.0, Mahout 0.13.0, Pig 0.17.0, and Tez 0.9.2
HBase: HBase 1.4.13, Hadoop 2.10.1, Hive 2.3.9, Hue 4.10.0, Phoenix 4.14.3, and ZooKeeper 3.4.14
Presto: Presto 0.266 with Hadoop 2.10.1 HDFS and Hive 2.3.9 Metastore
Spark: Spark 2.4.8 on Hadoop 2.10.1 YARN and Zeppelin 0.10.0
Use AWS Glue Data Catalog for table metadata
Hardware configuration
Instance type
m5.xlarge
The selected instance type adds 64 GiB of GP2 EBS storage per instance by default. Learn more
Number of instances
3
(1 master and 2 core nodes)
Cluster scaling
scale cluster nodes based on workload
Auto-termination
Enable auto-termination
Learn more

EMR on EKS



EMR on EKS Tech Talk : Demo : Job Creation

Create a simple job using a built-in PySpark script.

```
aws emr-containers start-job-run \
    --virtual-cluster-id ${EMR_EKS_CLUSTER_ID} \
    --name sample-pi \
    --execution-role-arn ${EMR_EKS_EXECUTION_ARN} \
    --release-label emr-6.2.0-latest \
    --job-driver '{
        "sparkSubmitJobDriver": {
            "entryPoint": "local:///usr/lib/spark/examples/src/main/python/pi.py",
            "sparkSubmitParameters": "--conf spark.executor.instances=2 --conf spark.executor.memory=2G --cc
        }
    }'
```

AWS Glue

AWS Glue is a fully managed ETL (extract, transform, and load) service that makes it simple and cost-effective to categorize your data, clean it, enrich it, and move it reliably between various data stores and data streams.

AWS Glue is serverless and supports pay-as-you-go model. There is no infrastructure to provision or manage. AWS Glue handles provisioning, configuration, and scaling of the resources required to run the ETL jobs on a fully managed, scale-out Apache Spark environment.

You pay only for the resources your jobs use while running.

AWS Glue features :

- Discover and organize data Connect to a wide variety of data sources – Tap into multiple data sources, both on premises and on AWS, using AWS Glue connections to build your data lake.

Glue Pricing

ETL Jobs – Apache Spark as a Glue job type that runs for 10 minutes and consumes 6 DPUs. The Price of 1 Data Processing Unit (DPU) – Hour is $0.44. Since your job ran for 10 Minutes of an hour and consumed 6 DPUs, you will be billed 6 DPUs X 10 minutes at $0.44 per DPU-hour or $0.44.

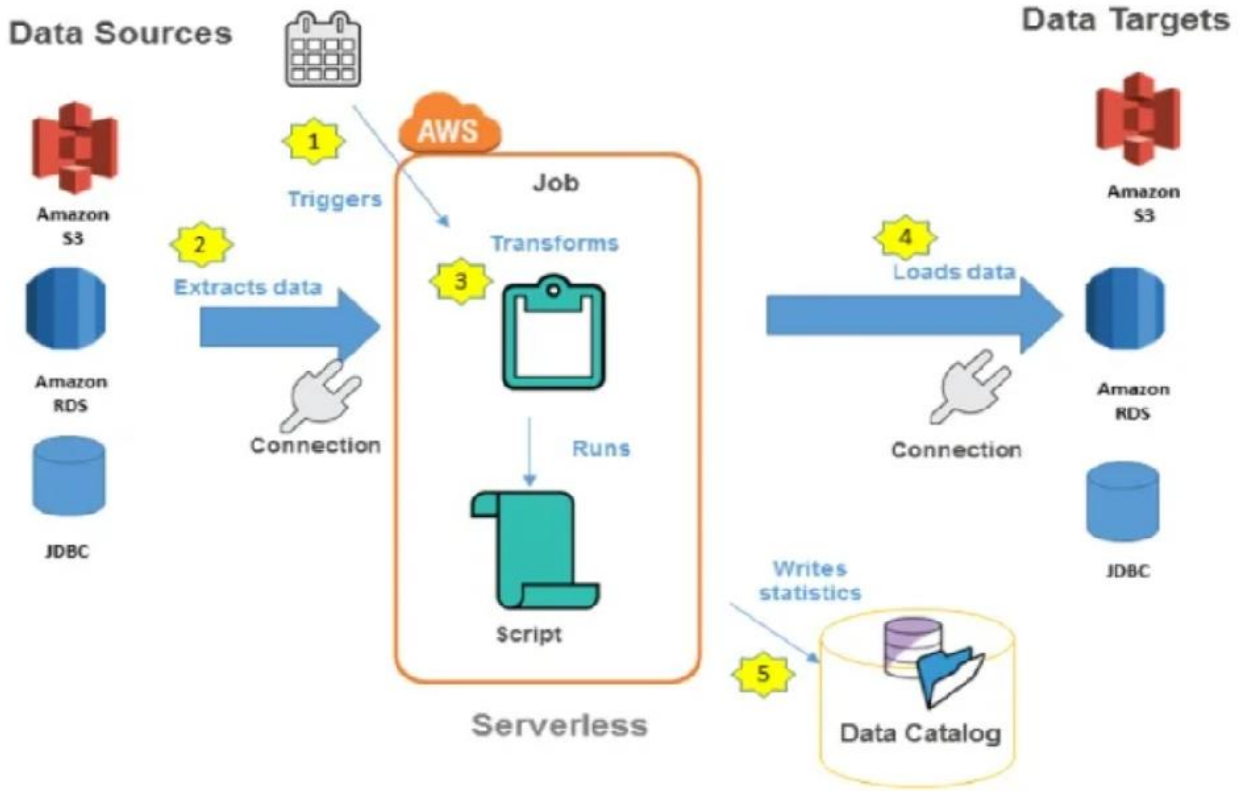
Worker type

The following worker types are available:

- `Standard` – When you choose this type, you also provide a value for **Maximum capacity**. Maximum capacity is the number of AWS Glue data processing units (DPUs) that can be allocated when this job runs. A DPU is a relative measure of processing power that consists of 4 vCPUs of compute capacity and 16 GB of memory. The `Standard` worker type has a 50 GB disk and 2 executors.
- `G.1X` – When you choose this type, you also provide a value for **Number of workers**. Each worker maps to 1 DPU (4 vCPU, 16 GB of memory, 64 GB disk), and provides 1 executor per worker. We recommend this worker type for memory-intensive jobs. This is the default **Worker type** for AWS Glue Version 2.0 or later jobs.
- `G.2X` – When you choose this type, you also provide a value for **Number of workers**. Each worker maps to 2 DPU (8 vCPU, 32 GB of memory, 128 GB disk), and provides 1 executor per worker. We recommend this worker type for memory-intensive jobs and jobs that run machine learning transforms.
- `G.025X` – When you choose this type, you also provide a value for **Number of workers**. Each worker maps to 0.25 DPU (2 vCPU, 4 GB of memory, 64 GB disk), and provides 1 executor per worker. We recommend this worker type for low volume streaming jobs. This worker type is only available for AWS Glue version 3.0 streaming jobs.

Number of DPUs	Cost Per Hour	Job Run Duration (minutes)	Cost Estimate – 1 Run	Cost Estimate – 5k runs
2	$0.44	3	$0.15	$733.33
10	$0.44	1	$0.73	$3,666.67
40	$0.44	1	$2.93	$14,666.67

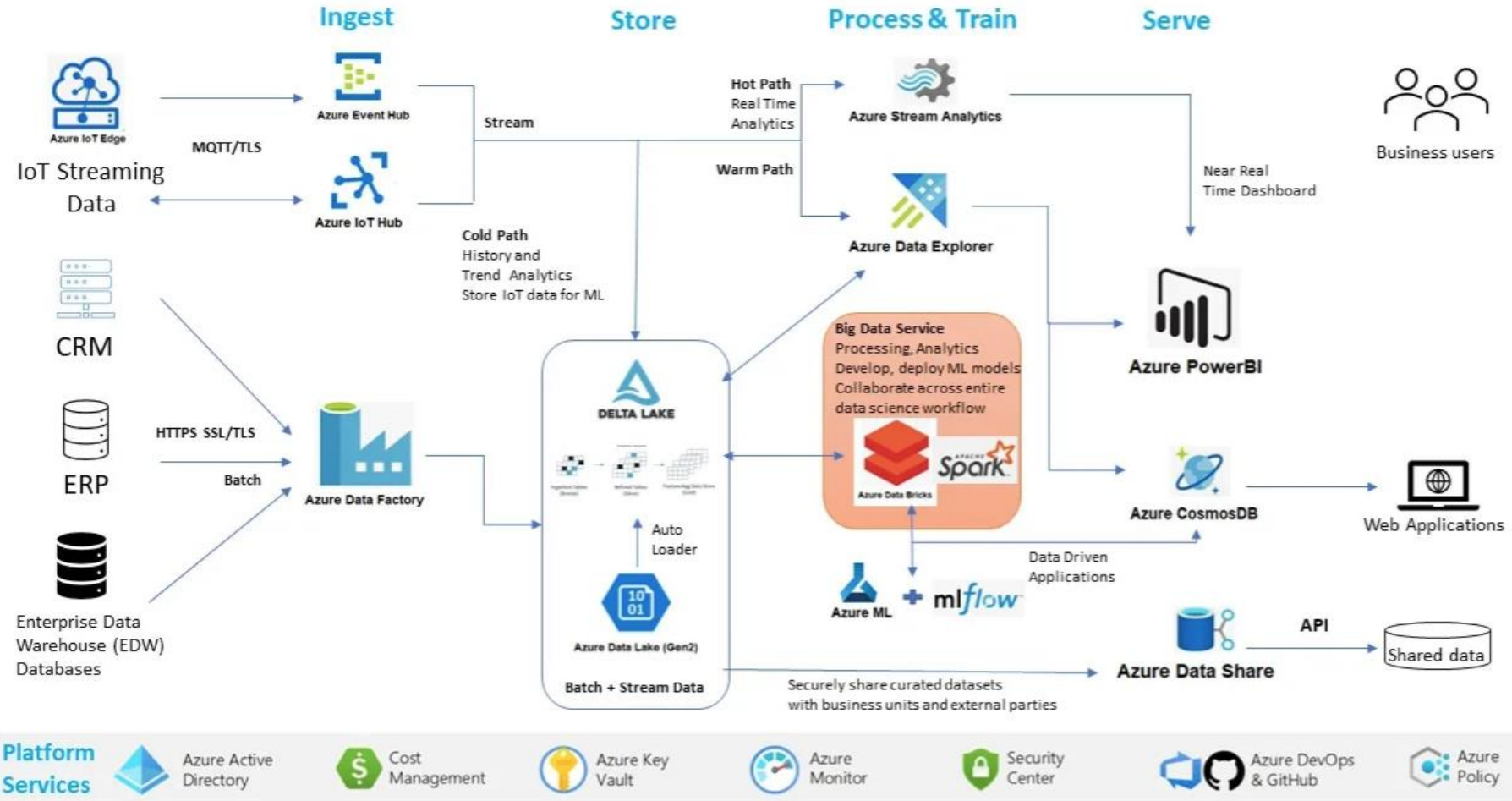
Glue Architecture



AWS Glue Components

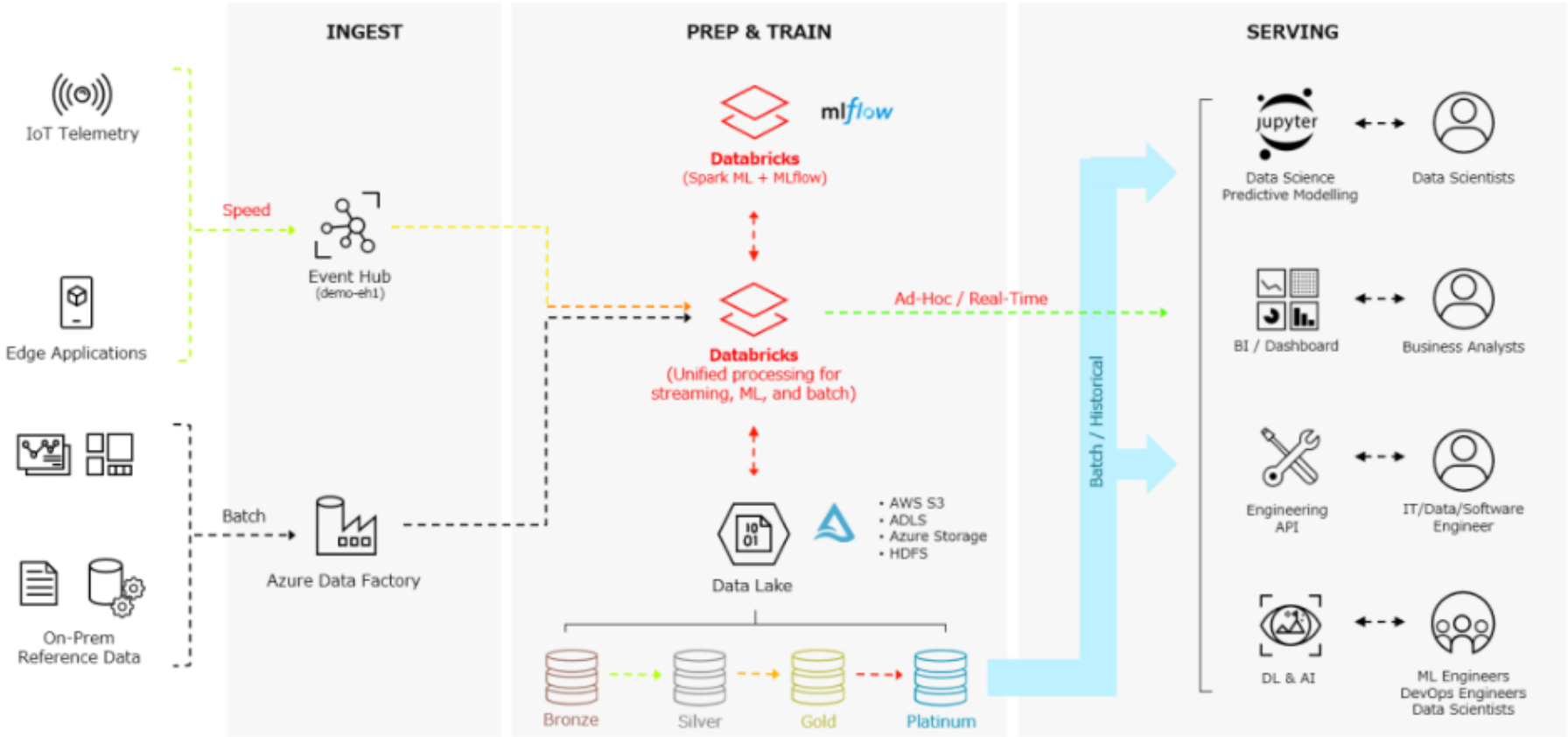
- AWS Glue console to define, monitor and orchestrate your ETL workflow.
- Data Catalog, which is a central metadata repository.
- AWS Glue connection is a Data Catalog object that stores login credentials, URI strings, virtual private cloud (VPC) information, and more for a particular data store. AWS Glue crawlers, jobs, and development endpoints use connections in order to access certain types of data stores.
- ETL engine that can automatically generate Scala or Python code,
- Flexible scheduler that handles dependency resolution, job

Cloud Solution Architecture Data Analytics with Azure Data Bricks



Unified Analytics Pipeline

Delta Architecture with Databricks



ML Data Scientist

Specialty Badges

Apache Spark Developer Associate

Hadoop Migration Architect

Platform Administrator Accreditation

Minimally Qualified Candidate

The minimally qualified candidate should be able to:

- Understanding the basics of the Spark architecture, including Adaptive Query Execution
- Apply the Spark DataFrame API to complete individual data manipulation task, including:
  - selecting, renaming and manipulating columns
  - filtering, dropping, sorting, and aggregating rows
  - joining, reading, writing and partitioning DataFrames
  - working with UDFs and Spark SQL functions

While it will not be explicitly tested, the candidate must have a working knowledge of either Python or Scala. The exam is available in both languages.

Duration

Testers will have 120 minutes to complete the certification exam.

Questions

There are 60 multiple-choice questions on the certification exam. The questions will be distributed by high-level topic in the following way:

- Apache Spark Architecture Concepts – 17% (10/60)
- Apache Spark Architecture Applications – 11% (7/60)
- Apache Spark DataFrame API Applications – 72% (43/60)

Databricks Certification

Spark Architecture — Conceptual

- Cluster architecture: nodes, drivers, workers, executors, slots, etc.
- Spark execution hierarchy: applications, jobs, stages, tasks, etc.
- Shuffling
- Partitioning
- Lazy evaluation
- Transformations vs Actions
- Narrow vs Wide transformations

Spark Architecture — Applied

- Execution deployment modes
- Stability
- Storage levels
- Repartitioning
- Coalescing
- Broadcasting
- DataFrames

Spark DataFrame API

- Subsetting DataFrames (select, filter, etc.)
- Column manipulation (casting, creating columns, manipulating existing columns, complex column types)
- String manipulation (Splitting strings, regex)
- Performance-based operations (repartitioning, shuffle partitions, caching)
- Combining DataFrames (joins, broadcasting, unions, etc)
- Reading/writing DataFrames (schemas, overwriting)
- Working with dates (extraction, formatting, etc)
- Aggregations
- Miscellaneous (sorting, missing values, typed UDFs, value extraction, sampling)

Databricks Certified Data Engineer Associate

The Databricks Certified Data Engineer Associate certification exam assesses an individual's ability to use the Databricks Lakehouse Platform to complete introductory data engineering tasks. This includes an understanding of the Lakehouse Platform and its workspace, its architecture, and its capabilities. It also assesses the ability to perform multi-hop architecture ETL tasks using Apache Spark SQL and Python in both batch and incrementally processed paradigms. Finally, the exam assesses the tester's ability to put basic ETL pipelines and Databricks SQL queries and dashboards into production while maintaining entity permissions. Individuals who pass this certification exam can be expected to complete basic data engineering tasks using Databricks and its associated tools.

Minimally Qualified Candidate

The minimally qualified candidate should be able to:

- Understand how to use and the benefits of using the Databricks Lakehouse Platform and its tools, including:
  - Data Lakehouse (architecture, descriptions, benefits)
  - Data Science and Engineering workspace (clusters, notebooks, data storage)
  - Delta Lake (general concepts, table management and manipulation, optimizations)
- Build ETL pipelines using Apache Spark SQL and Python, including:
  - Relational entities (databases, tables, views)
  - ELT (creating tables, writing data to tables, cleaning data, combining and reshaping tables, SQL UDFs)
  - Python (facilitating Spark SQL with string manipulation and control flow, passing data between PySpark and Spark SQL)
- Incrementally process data, including:
  - Structured Streaming (general concepts, triggers, watermarks)
  - Auto Loader (streaming reads)
  - Multi-hop Architecture (bronze-silver-gold, streaming applications)
  - Delta Live Tables (benefits and features)
- Build production pipelines for data engineering applications and Databricks SQL queries and dashboards, including:
  - Jobs (scheduling, task orchestration, UI)
  - Dashboards (endpoints, scheduling, alerting, refreshing)
- Understand and follow best security practices, including:
  - Unity Catalog (benefits and features)
  - Entity Permissions (team-based permissions, user-based permissions)

Duration

Testers will have 90 minutes to complete the certification exam.

Q & A