Cognixia®

# DataBricks Spark – Azure DataBricks

www.cognixia.com

## Contents

## Spark Applications

**Submit Options**

- The Spark submit script provides many options to specify how the application should run
  - Most are the same as for pyspark2 and spark2-shell

- General submit flags include

– master: local, yarn, or a Mesos or Spark Standalone cluster manager URI

– jars: Additional JAR files (Scala and Java only)

– pyfiles: Additional Python files (Python only)

– driver-java-options: Parameters to pass to the driver JVM

## Spark Applications

– num-executors: Number of executors to start application with

– driver-cores: Number cores to allocate for the Spark driver

– queue: YARN queue to run in

## Spark Applications

- Spark determines how to partition data in an RDD, Dataset, or DataFrame when
    - The data source is read
    - An operation is performed on a DataFrame, Dataset, or RDD
    - Spark optimizes a query
    - You call repartition or coalesce

## Spark Applications

- Partitions are determined when files are read
  - Core Spark determines RDD partitioning based on location, number, and size of files
  - Usually each file is loaded into a single partition
  - Very large files are split across multiple partitions
  - Catalyst optimizer manages partitioning of RDDs that implement DataFrames and Datasets

- You can view the number of partitions in an RDD by calling the function getNumPartitions

```
myRDD.getNumPartitions
```

**Language**: *Scala*

```
myRDD.getNumPartitions()
```

**Language**: *Python*

## Spark Stages and Tasks

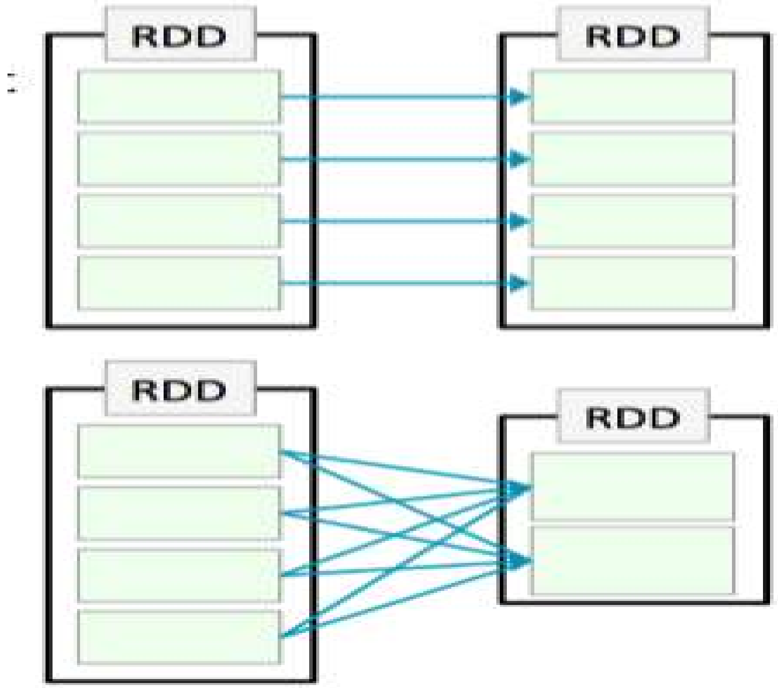
- A task is a series of operations that work on the same partition and are pipelined together
- Stages group together tasks that can run in parallel on different partitions of the same RDD
- Jobs consist of all the stages that make up a query
- Catalyst optimizes partitions and stages when using DataFrames and Datasets
  - Core Spark provides limited optimizations when you work directly with RDDs
  - You need to code most RDD optimizations manually
  - To improve performance, be aware of how tasks and stages are executed when working with RDDs

## Spark Execution Plan

- Spark creates an execution plan for each job in an application
- Catalyst creates SQL, Dataset, and DataFrame execution plans
    - Highly optimized

- Core Spark creates execution plans for RDDs
    - Based on RDD lineage
    - Limited optimization

## Spark Execution Plan



## Spark Execution Plan

- Spark constructs a DAG (Directed Acyclic Graph) based on RDD dependencies

- Narrow dependencies
    - Each partition in the child RDD depends on just one partition of the parent RDD
    - No shuffle required between executors
    - Can be pipelined into a single stage
    - Examples: map, filter, and union

## Spark Execution Plan

- Wide (or shuffle) dependencies
  - Child partitions depend on multiple partitions in the parent RDD
  - Defines a new stage
  - Examples: reduceByKey, join, and groupByKey

## Controlling the Number of partitions

- Partitioning determines how queries execute on a cluster
    - More partitions = more parallel tasks
    - Cluster will be under-utilized if there are too few partitions
    - But too many partitions will increase overhead without an offsetting increase in performance

- Catalyst controls partitioning for SQL, DataFrame, and Dataset queries
- You can control how many partitions are created for RDD queries

## Controlling the Number of partitions

- Specify the number of partitions when data is read
  - Default partitioning is based on size and number of the files (minimum is two)
  - Specify a different minimum number when reading a file

```
myRDD = sc.textFile(myfile,5)
```

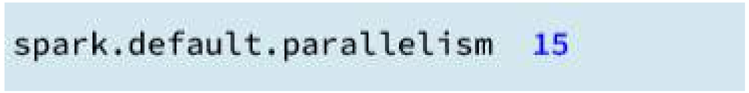
## Controlling the Number of partitions

- Manually repartition
    - Create a new RDD with a specified number of partitions using repartition or coalesce
    - coalesce reduces the number of partitions without requiring a shuffle
    - repartition shuffles the data into more or fewer partitions

```
newRDD = myRDD.repartition(15)
```

## Controlling the Number of partitions

- Specify the number of partitions created by transformations
    - Wide (shuffle) operations such as reduceByKey and join repartition data
    - By default, the number of partitions created is based on the number of partitions of the parent RDD(s)
    - Choose a different default by configuring the spark.default.parallelism property



```
spark.default.parallelism  15
```

## Catalyst Optimizer

- Catalyst can improve SQL, DataFrame, and Dataset query performance by optimizing the DAG to
  - Minimize data transfer between executors
    - Such as broadcast joins—small data sets are pushed to the executors where the larger data sets reside
  - Minimize wide (shuffle) operations
  - Such as unioning two RDDs—grouping, sorting, and joining do not require shuffling
  - Pipeline as many operations into a single stage as possible
  - Generate code for a whole stage at run time
  - Break a query job into multiple jobs, executed in a series

## Catalyst Optimizer

- Execution plans for DataFrame, Dataset, and SQL queries include the following phases
  - Parsed logical plan—calculated directly from the sequence of operations specified in the query
  - Analyzed logical plan—resolves relationships between data sources and columns
  - Optimized logical plan—applies rule-based optimizations
  - Physical plan—describes the actual sequence of operations
  - Code generation—generates bytecode to run on each node, based on a cost model

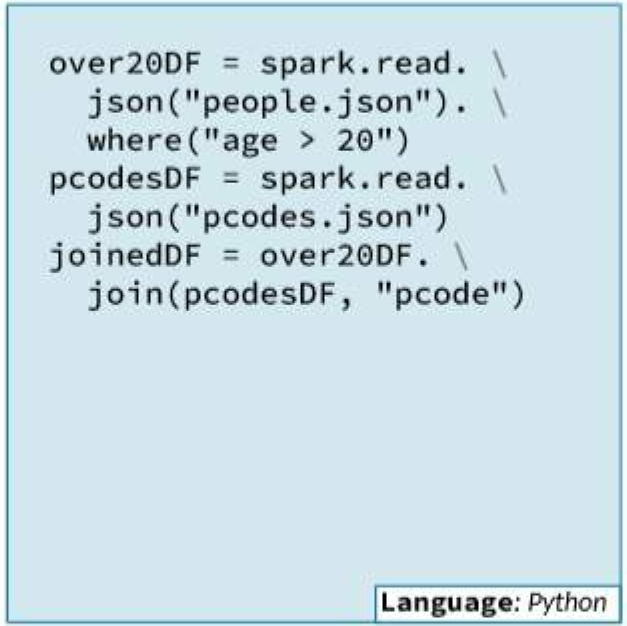
## Spark Persistence

### Spark persistence

- ❑ You can persist a DataFrame, Dataset, or RDD
  - — Also called caching
  - — Data is temporarily saved to memory and/or disk
- ❑ Persistence can improve performance and fault-tolerance
- ❑ Use persistence when
  - — Query results will be used repeatedly
  - — Executing the query again in case of failure would be very expensive
- ❑ Persisted data cannot be shared between applications

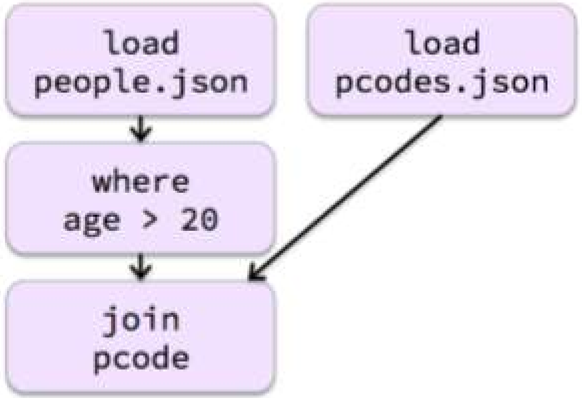
## Spark Persistence

### Spark persistence

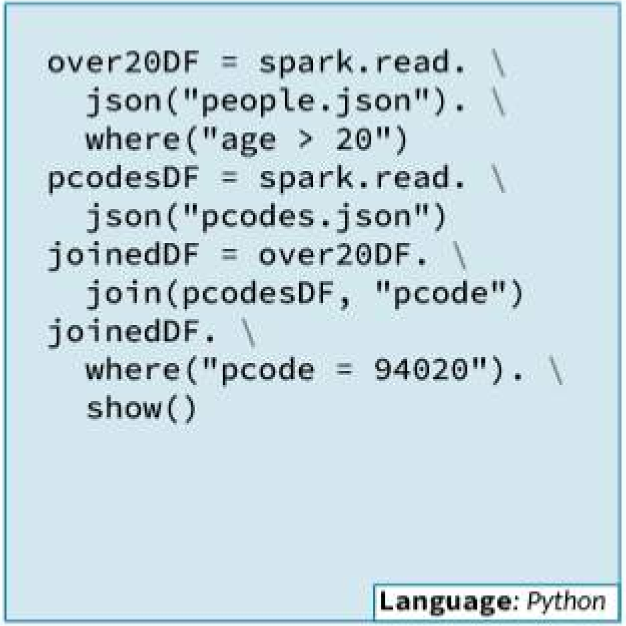


```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode")
```

**Language**: *Python*

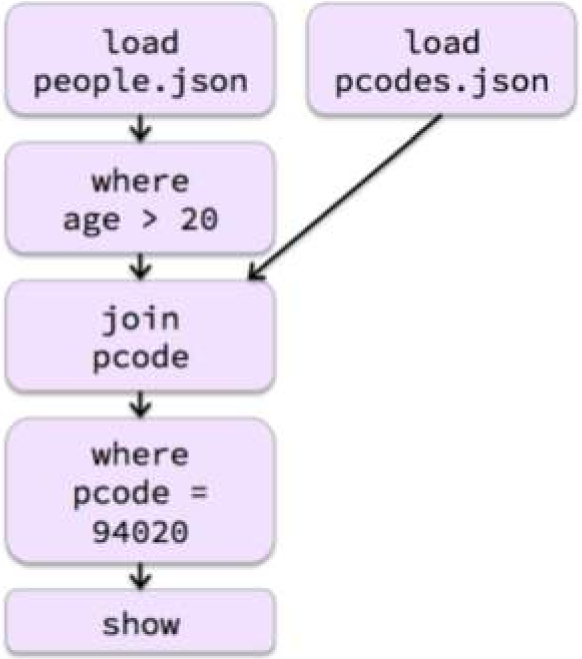


## Spark persistence



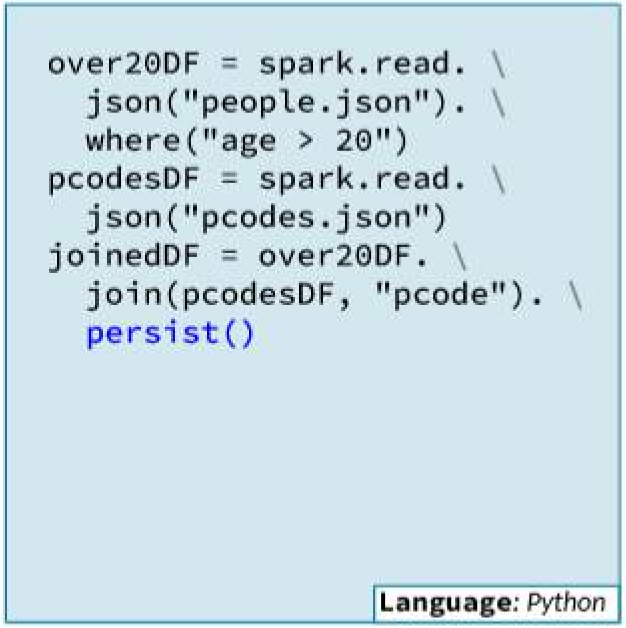
```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode")
joinedDF. \
  where("pcode = 94020"). \
  show()
```

**Language:** *Python*



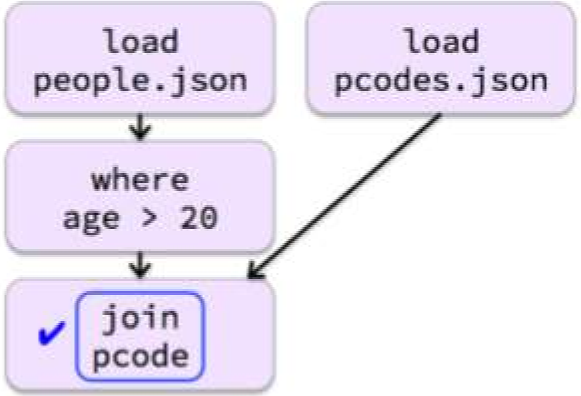
## Spark Persistence

### Spark persistence



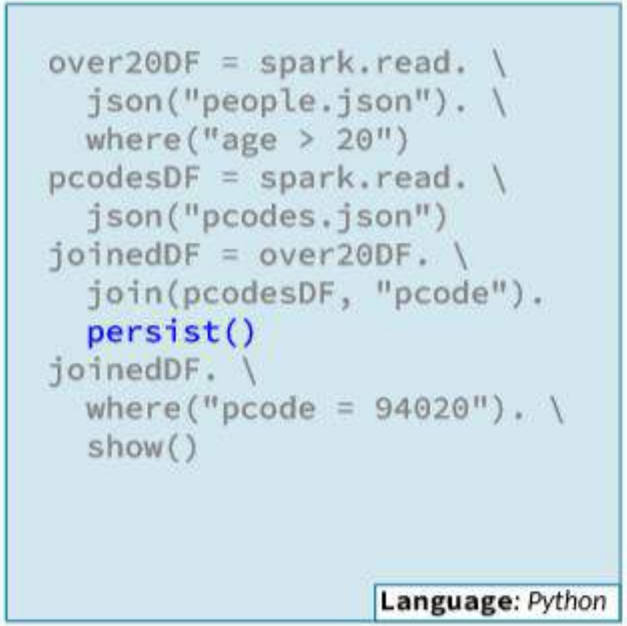
```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode"). \
  persist()
```

**Language:** *Python*



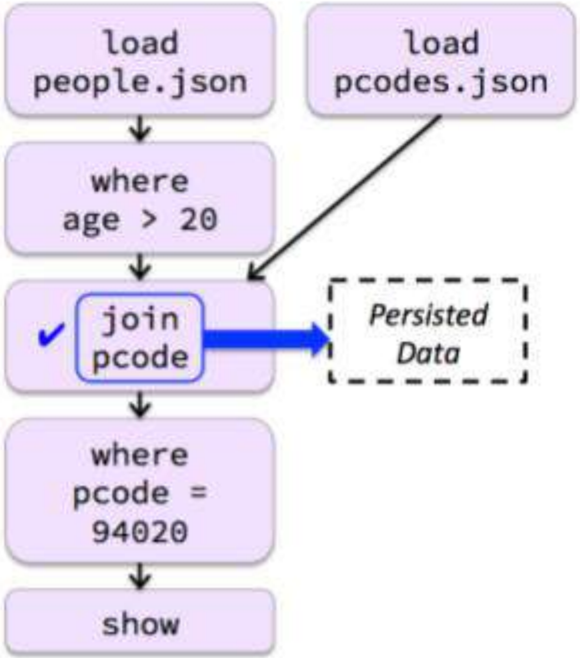
## Spark Persistence

### Spark persistence

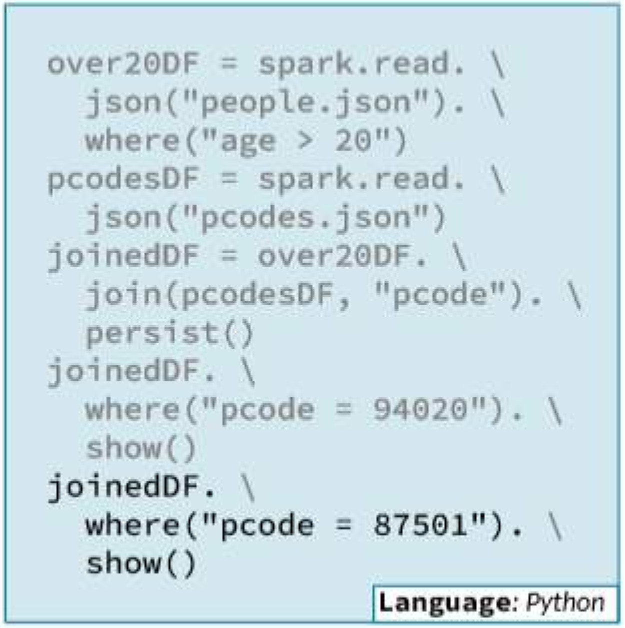


```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode").
  persist()
joinedDF. \
  where("pcode = 94020"). \
  show()
```

**Language:** *Python*

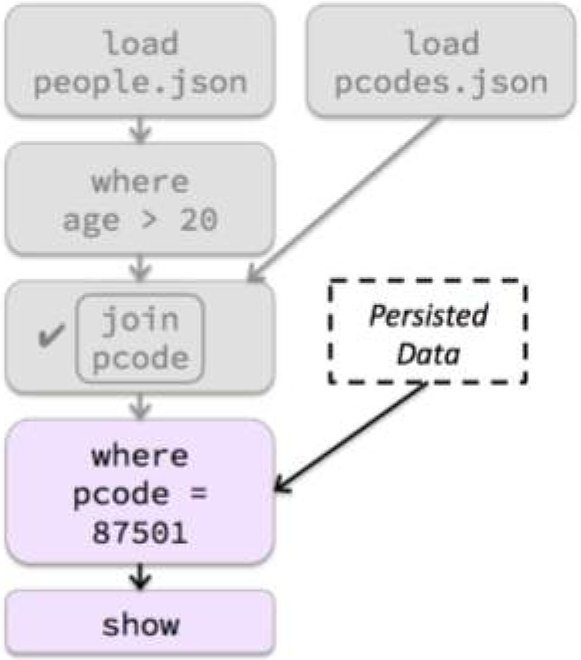


## Spark persistence



```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode"). \
  persist()
joinedDF. \
  where("pcode = 94020"). \
  show()
joinedDF. \
  where("pcode = 87501"). \
  show()
```

**Language:** *Python*



## Table and View Persistence

- ❑ Tables and views can be persisted in memory using CACHE TABLE

  ```
  spark.sql("CACHE TABLE people")
  ```

- ❑ CACHE TABLE can create a view based on a SQL query and cache it at the same time

  ```
  spark.sql("CACHE TABLE over_20 AS SELECT *   FROM people WHERE
  age > 20")
  ```

- ❑ Queries on cached tables work the same as on persisted DataFrames, Datasets, and RDDs
  - – The first query caches the data
  - – Subsequent queries use the cached data

## Spark Persistence

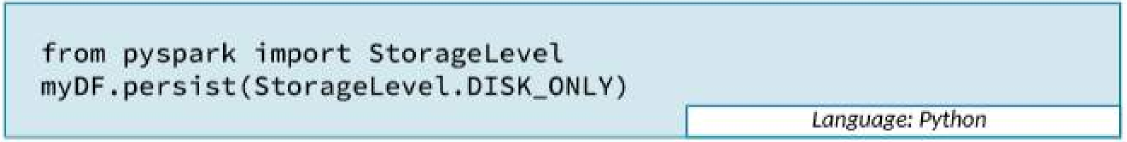
### Persistence Storage levels

- ❑ Storage levels provide several options to manage how data is persisted
  - – Storage location (memory and/or disk)
  - – Serialization of data in memory
  - – Replication
- ❑ Specify storage level when persisting a DataFrame, Dataset, or RDD
  - – Tables and views do not use storage levels
    - – Always persisted in memory
- ❑ Data is persisted based on partitions of the underlying RDDs
  - – Executors persist partitions in JVM memory or temporary local files
  - – The application driver keeps track of the location of each persisted partition’s data

## Spark Persistence

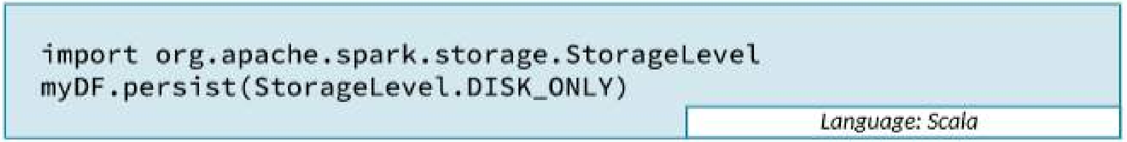
### Persistence Storage levels

- ❑ Storage location—where is the data stored?
  - – MEMORY_ONLY: Store data in memory if it fits
  - – DISK_ONLY: Store all partitions on disk
  - – MEMORY_AND_DISK: Store any partition that does not fit in memory on disk
    - – Called spilling



```python
from pyspark import StorageLevel
myDF.persist(StorageLevel.DISK_ONLY)
```

*Language: Python*



```scala
import org.apache.spark.storage.StorageLevel
myDF.persist(StorageLevel.DISK_ONLY)
```

*Language: Scala*

# Spark Persistence

## Persistence Storage levels - Memory Serialization

- ❑ In Python, data in memory is always serialized
- ❑ In Scala, you can choose to serialize data in memory
  - – By default, in Scala and Java, data in memory is stored objects
  - – Use MEMORY_ONLY_SER and MEMORY_AND_DISK_SER to serialize the objects into a sequence of bytes instead
  - – Much more space efficient but less time efficient
- ❑ Datasets are serialized by Spark SQL encoders, which are very efficient
  - – Plain RDDs use native Java/Scala serialization by default
  - – Use Kryo instead for better performance
- ❑ Serialization options do not apply to disk persistence
  - – Files are always in serialized form by definition

## Persistence Storage levels - Default Storage Levels

- ❑ The storageLevel parameter for the DataFrame, Dataset, or RDD persist operation is optional
  - – The default for DataFrames and Datasets is MEMORY_AND_DISK
  - – The default for RDDs is MEMORY_ONLY
- ❑ persist with no storage level specified is a synonym for cache

  myDF.persist() is equivalent to myDF.cache()

  Table and view storage level is always MEMORY_ONLY

## Persistence Storage levels - When and Where to Persist

- ❑ When should you persist a DataFrame, Dataset, or RDD?
  - – When the data is likely to be reused
    - – Such as in iterative algorithms and machine learning
  - – When it would be very expensive to recreate the data if a job or node fails

## Persistence Storage levels - When and Where to Persist

❑ How to choose a storage level

– Memory—use when possible for best performance

– Save space by serializing the data if necessary

– Disk—use when re-executing the query is more expensive than disk read

– Such as expensive functions or filtering large datasets

– Replication—use when re-execution is more expensive than bandwidth

## Contents

## Spark Streaming

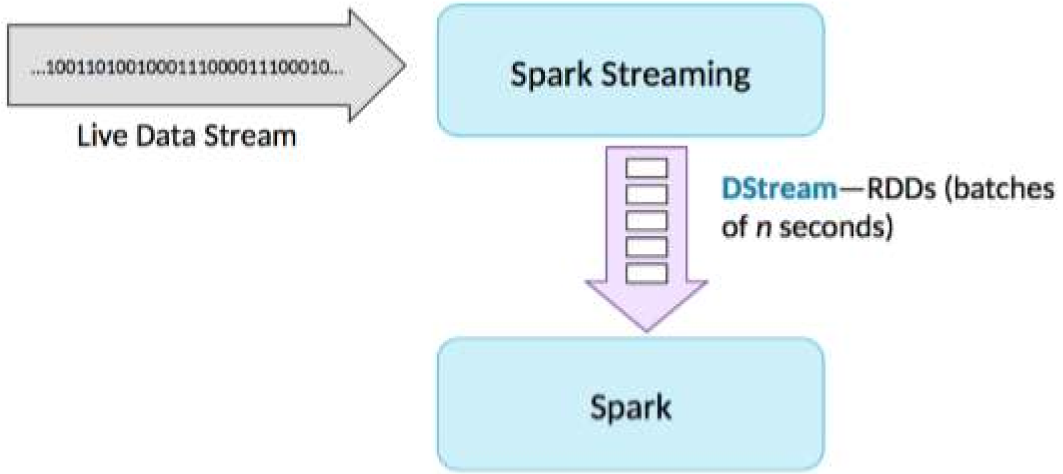
- ❑ An extension of core Spark
- ❑ Provides real-time data processing
- ❑ Segments an incoming stream of data into micro-batches
- ❑ Many big data applications need to process large data streams in real time, such as
  - – Continuous ETL
  - – Website monitoring
  - – Fraud detection
  - – Advertisement monetization
  - – Social media analysis
  - – Financial market trends

## Spark Streaming - Features

- ❑ Latencies of a few seconds or less
- ❑ Scalability and efficient fault tolerance
- ❑ “Once and only once” processing
- ❑ Integrates batch and real-time processing
- ❑ Uses the core Spark RDD API

## DStreams

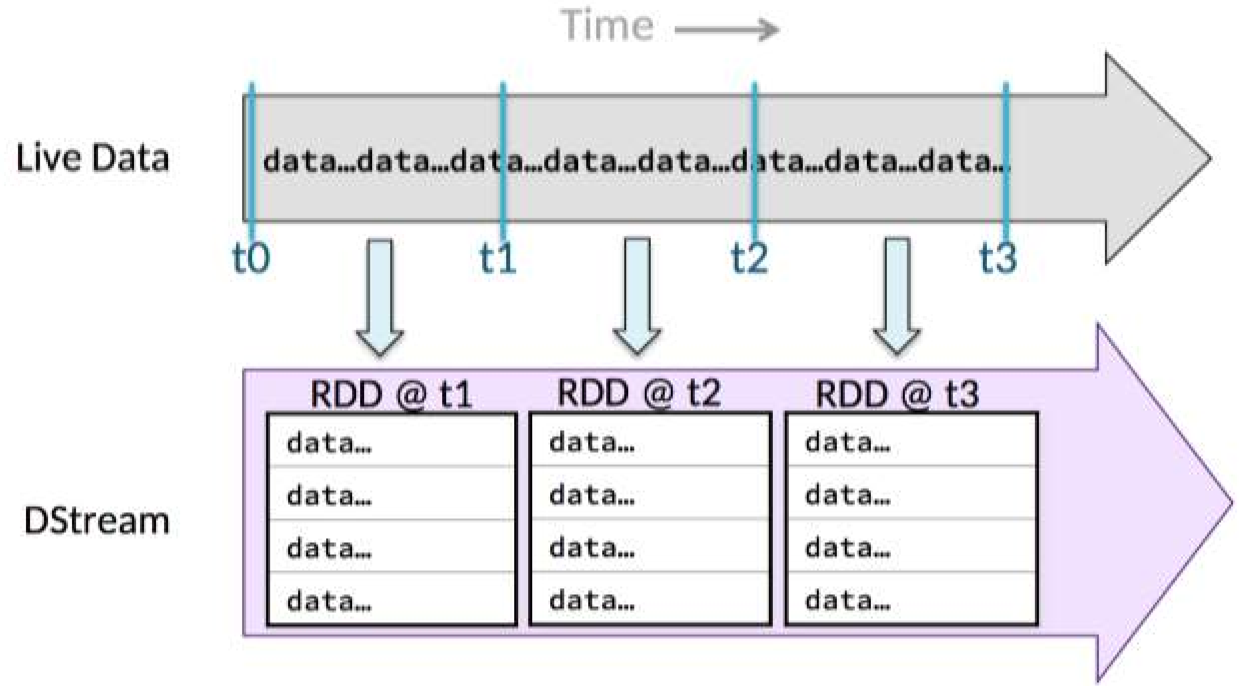
- ❑ DStream—RDDs (batches of n seconds)
- ❑ A DStream (Discretized Stream) divides a data stream into batches of n seconds
- ❑ Processes each batch in Spark as an RDD
- ❑ Returns results of RDD operations in batches



## Spark Streaming

### DStreams

- A DStream is a sequence of RDDs representing a data stream



## DStreams Data Sources

- DStreams are defined for a given input stream (such as a Unix socket) – Created by the Streaming context

  ```
  ssc.socketTextStream(hostname, port)
  ```

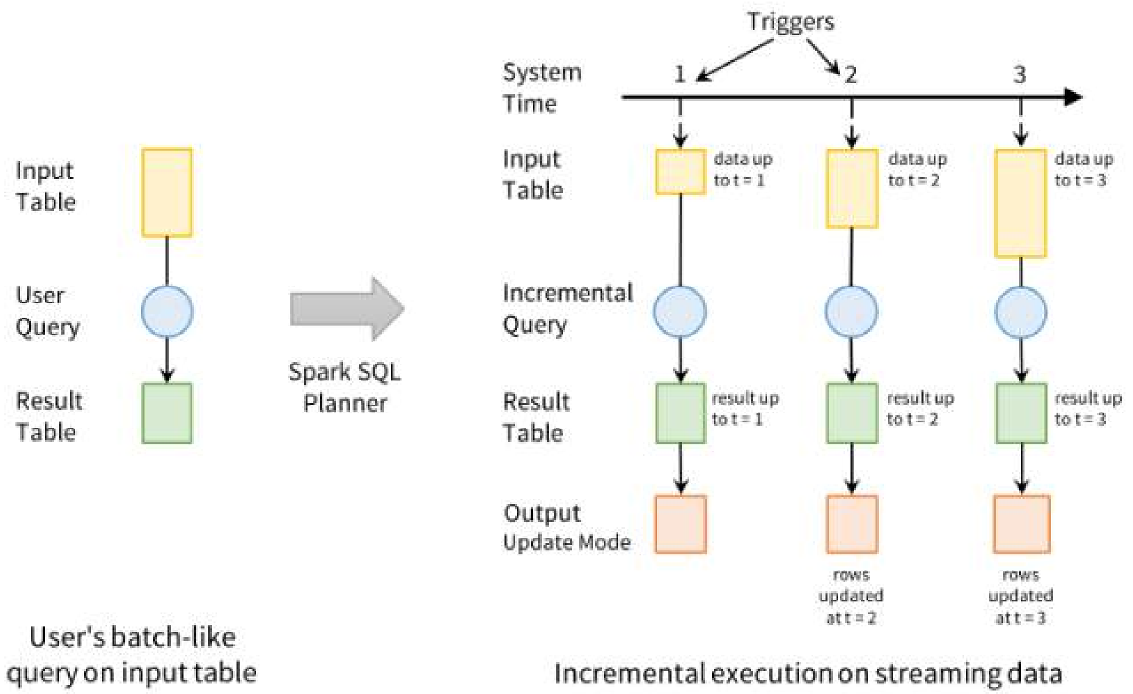
  – Similar to how RDDs are created by the Spark context
- Out-of-the-box data sources
  - – Network
    - – Sockets
    - – Services such as Apache Flume, Apache Kafka, ZeroMQ, or Amazon Kinesis
  - – Files
    - – Monitors an HDFS directory for new content

## DStreams Data Sources

- ❑ DStream operations are applied to every RDD in the stream
  - – Executed once per duration
- ❑ Two types of DStream operations
  - – Transformations
    - – Create a new DStream from an existing one
  - – Output operations
    - – Write data (for example, to a file system, database, or console)
    - – Similar to RDD actions

All the other operations arer same as RDD.

## Spark Structured Streaming



## Spark Structured Streaming

Spark Structured Streaming (aka Structured Streaming or Spark Streams) is the module of Apache Spark for stream processing using streaming queries

Streaming queries can be expressed using a high-level declarative streaming API or  SQL (SQL over stream / streaming SQL). The declarative streaming and SQL are executed on the underlying highly-optimized Spark SQL engine.

## Spark Structured Streaming

Spark Structured Streaming comes with two stream execution engines for executing streaming queries

Micro Batch Execution

Continuous Streaming

## Spark Structured Streaming

Using Structured streaming we can posses

Streaming Aggregation

Streaming Join

Streaming Watermark

Stateful Stream processing

## Spark Structured Streaming

In Structured Streaming, Spark developers describe custom streaming computations in the same way as with Spark SQL.

Internally, Structured Streaming applies the user-defined structured query to the continuously and indefinitely arriving data to analyze real-time streaming data.

Cognixia®

## THANK YOU