Introduction to Spark

Big Data processing

Fei GAO

Mai 2024

What Is Apache Spark?

Apache Spark is a **unified computing engine** and a set of **libraries** designed for **parallel data processing** on **computer clusters**

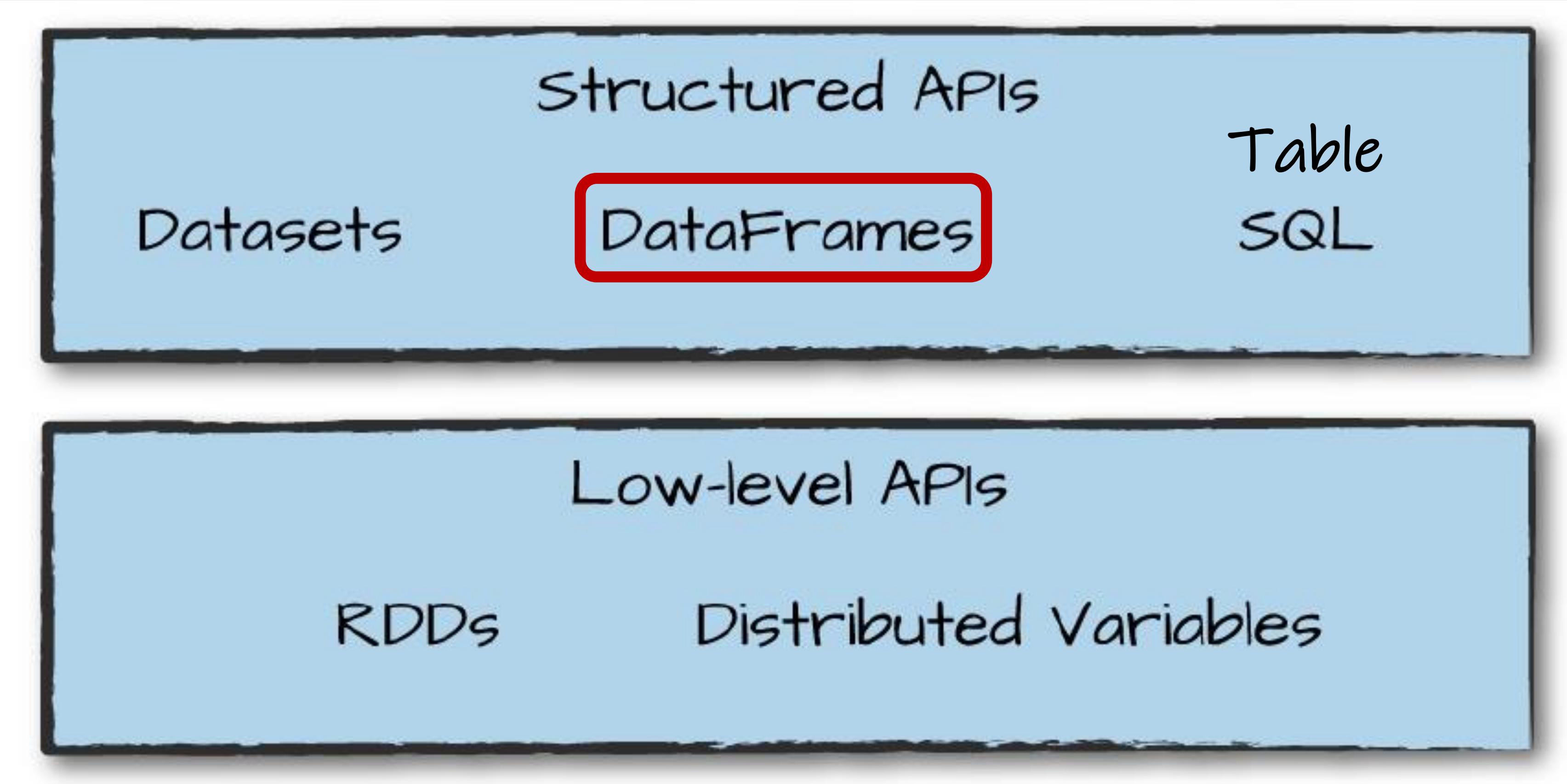
Spark itself is written in **Scala**, and runs on the Java Virtual Machine (**JVM**), You can use Spark from **Python** , **Scala**, **Java or R**.

Libraries incorporated :

- SQL for interactive queries (Spark SQL)
- machine learning (MLlib)
- stream processing (Structured Streaming) for interacting with real-time data
- graph processing (GraphX)

Runs from a laptop to a cluster of thousands of servers

What Is Apache Spark?



Language	Typed and untyped main abstraction	Typed or untyped
Scala	`Dataset[T]` and DataFrame (alias for `Dataset[Row]`)	Both typed and untyped
Java	`Dataset<T>`	Typed
Python	DataFrame	Generic Row untyped
R	DataFrame	Generic Row untyped

Spark's Philosophy

Unified

- Spark's key driving goal is to offer a unified platform for implementing big data applications
- composable APIs : build an application out of smaller pieces or out of existing libraries
- enable high performance by optimizing across the different libraries and functions composed together in a user program
- Langages API
- "structured APIs" (DataFrames, Datasets, and SQL) finalized in Spark 2.0 enable more powerful optimization under user applications

Spark's Philosophy

Computing engine

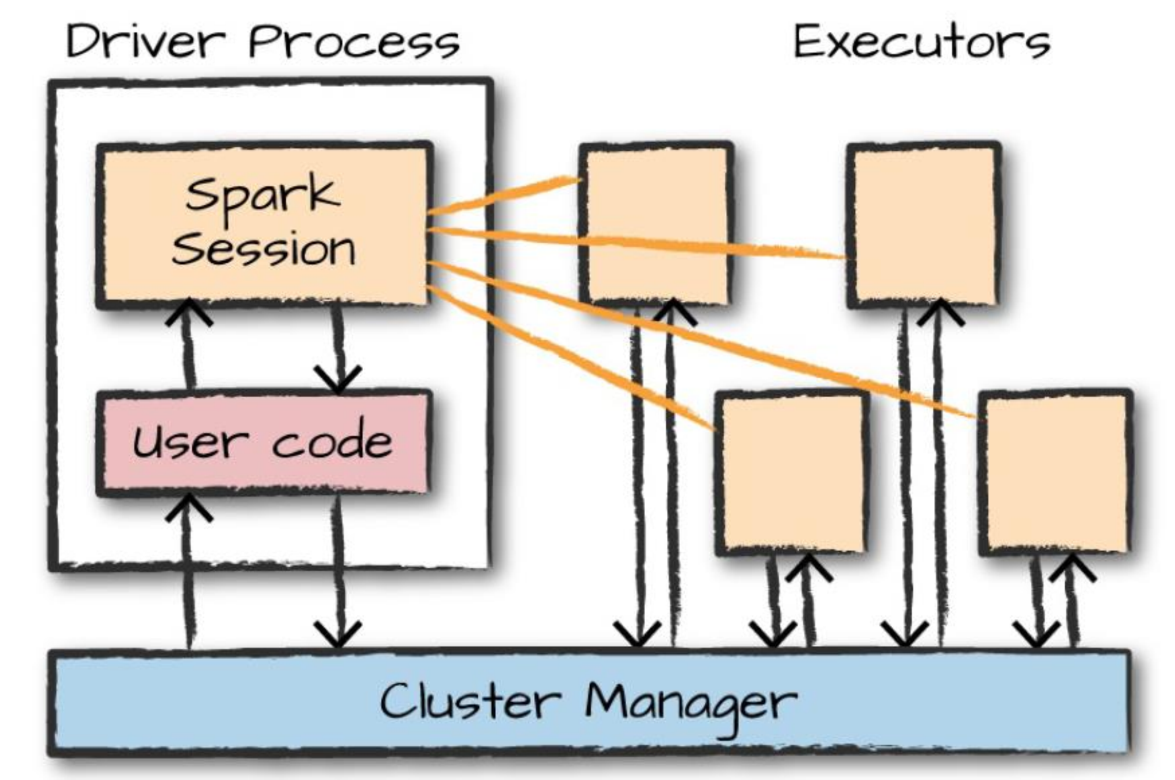
- handles loading data from storage systems
- performing computation
- not permanent storage as the end itself
- Compatible with a wide variety of storage systems (Azure, Amazon, Hadoop, Apache Cassandra, Apache Kafka…)

Librairies

- standard libraries
- a wide array of external libraries published as third-party packages by the open source communities : https://spark-packages.org/

Spark Application

- Spark Applications are the combination of two things: a driver process and a set of executor processes.
- Driver process runs user's code by analyzing, distributing, and scheduling work across the executors.
- The executors are responsible for carrying out the work that the driver assigns them.
- Spark employs a cluster manager that keeps track of the resources available

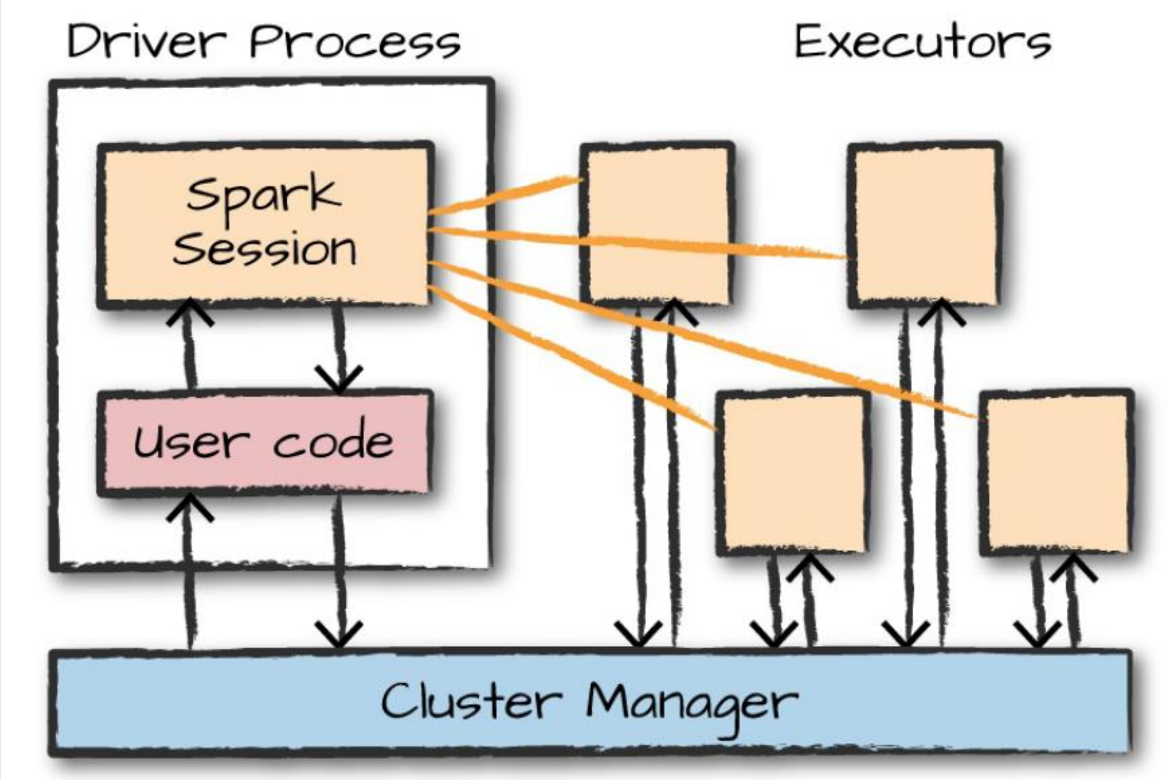


More info : https://blog.knoldus.com/understanding-the-working-of-spark-driver-and-executor/

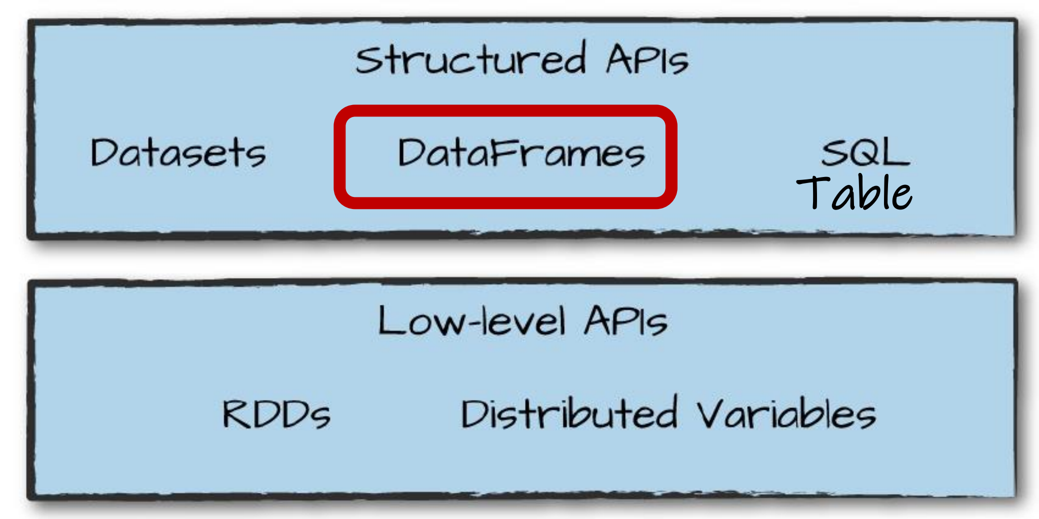
Spark Application

- The **SparkSession** instance is the way Spark executes user-defined manipulations across the cluster

- One-to-one correspondence between a SparkSession and a Spark Application

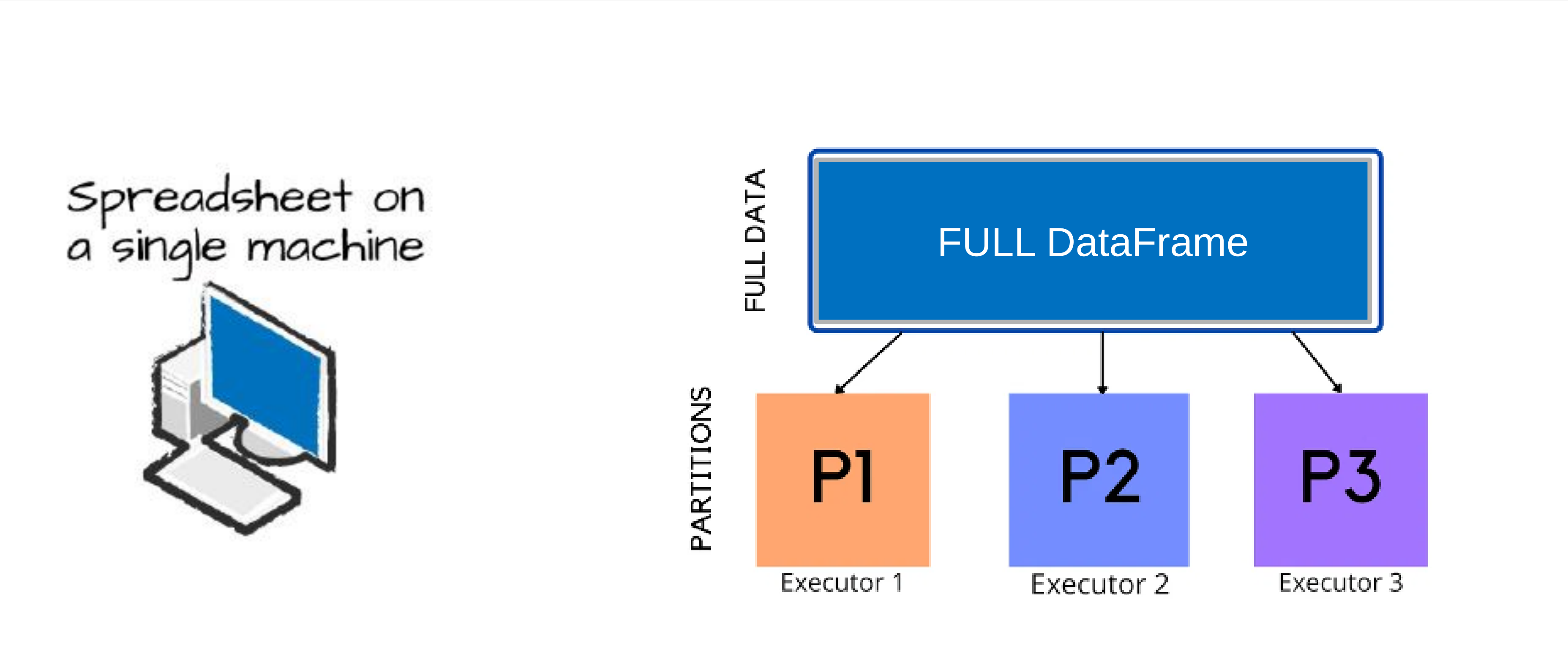


DataFrames : most common Structured API



Language	Typed and untyped main abstraction	Typed or untyped
Scala	`Dataset[T]` and DataFrame (alias for `Dataset[Row]`)	Both typed and untyped
Java	`Dataset<T>`	Typed
Python	DataFrame	Generic Row untyped
R	DataFrame	Generic Row untyped

DataFrames : most common Structured API



Distributed versus single-machine analysis

Spark's core concepts: Partitions

- A partition is a collection of rows that sit on one physical machine in your cluster.
- With DataFrames you do not (for the most part) manipulate partitions manually or individually.
- Spark determines how this work will actually execute on the cluster.

Transformation vs Action

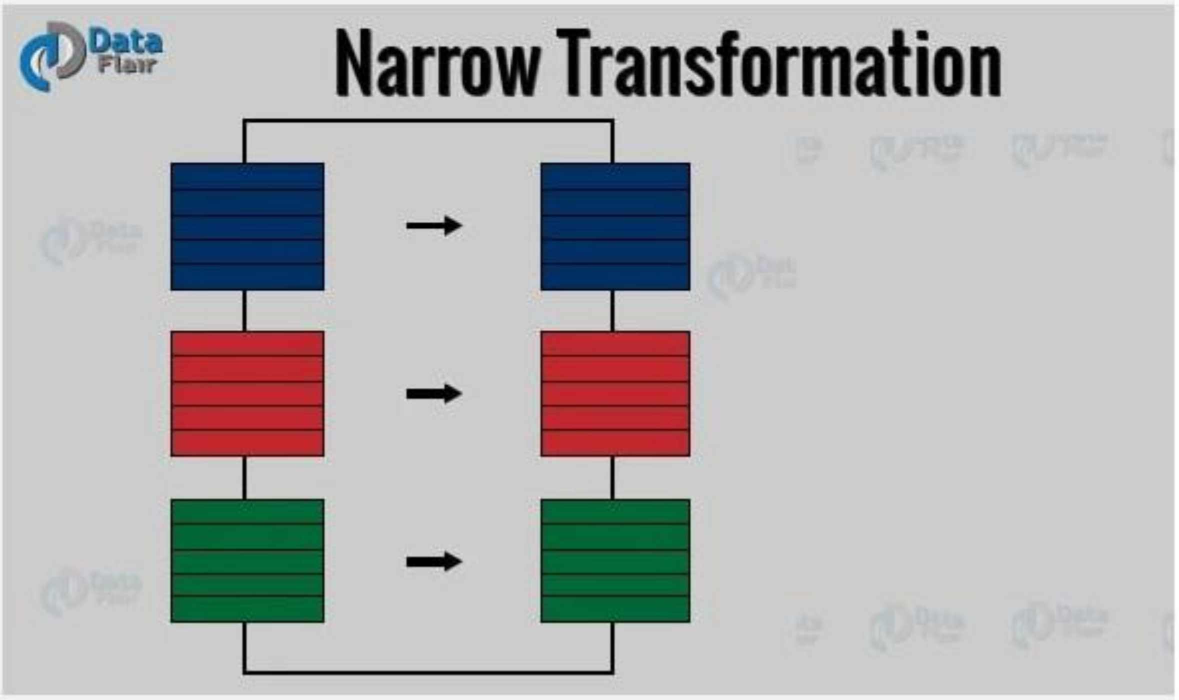
- Spark code essentially consists of **transformations** and **actions**. *How you build these is up to you—whether it's through SQL, low-level RDD manipulation, or machine learning algorithms.*

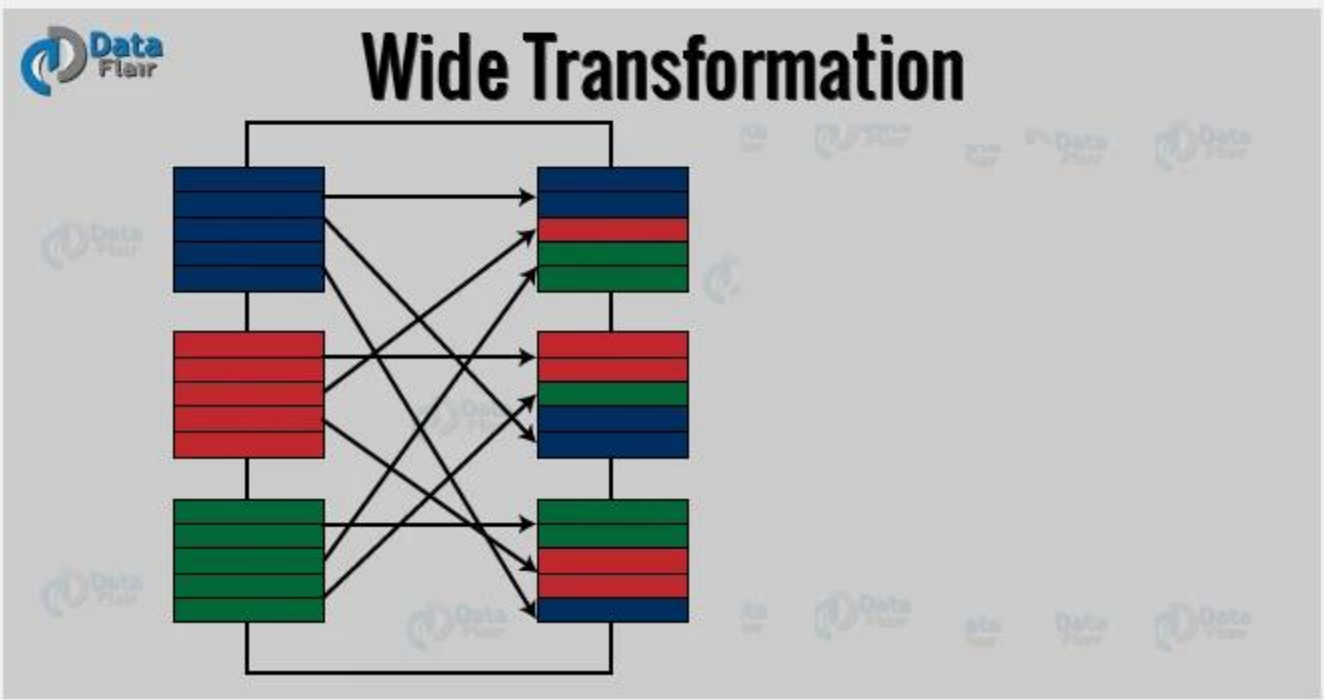
- Transformations : A Spark operation that reads a DataFrame (DS, RDD), manipulates some of the columns, and returns another.

- Transformations allow us to build up our logical transformation plan.

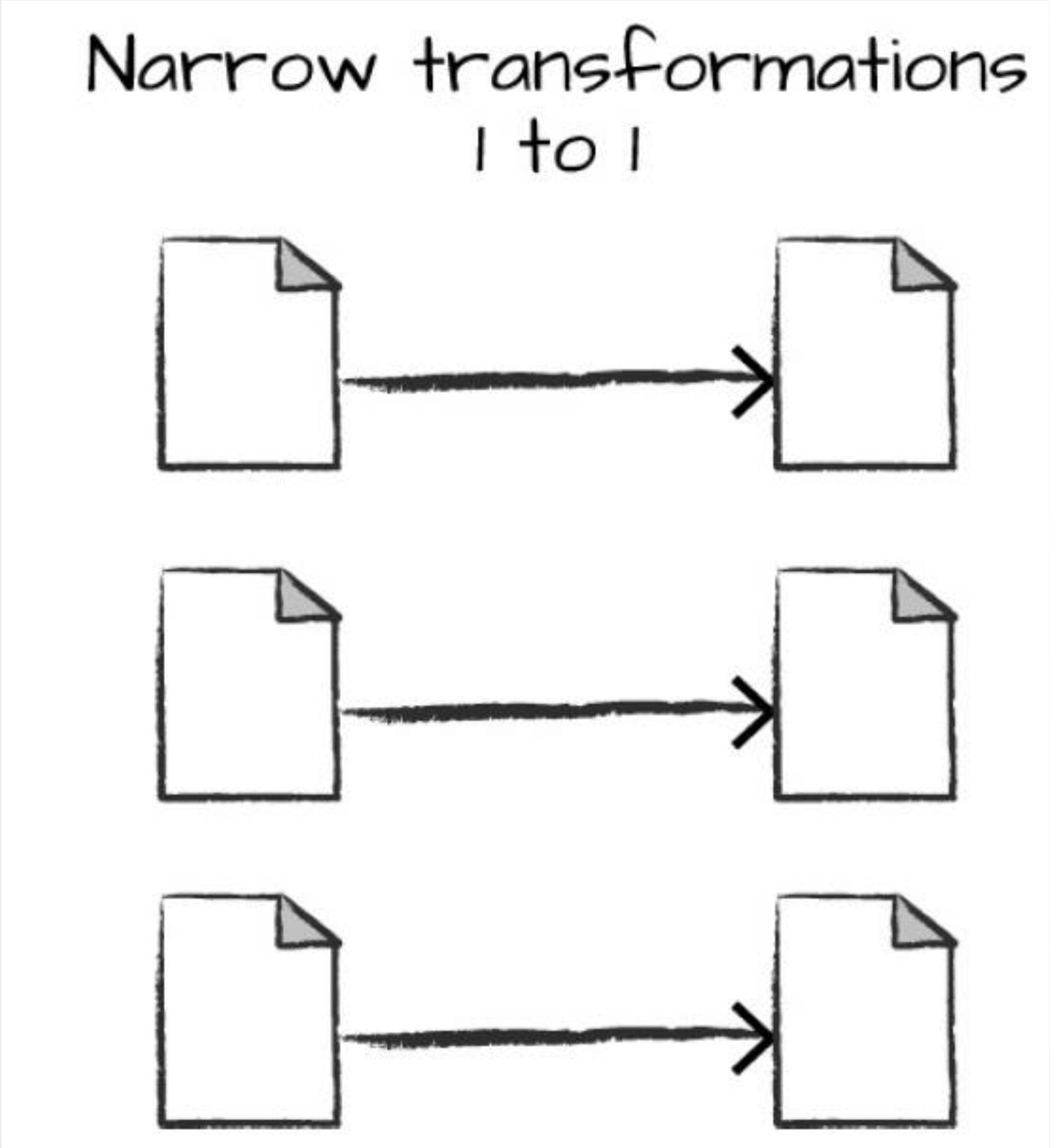
- Two types of transformations : **Narrow dependencies and Wide dependencies**
  1. Narrow Transformations: applies on a single partition, can operate in single partition and no data exchange happens here between partitions.

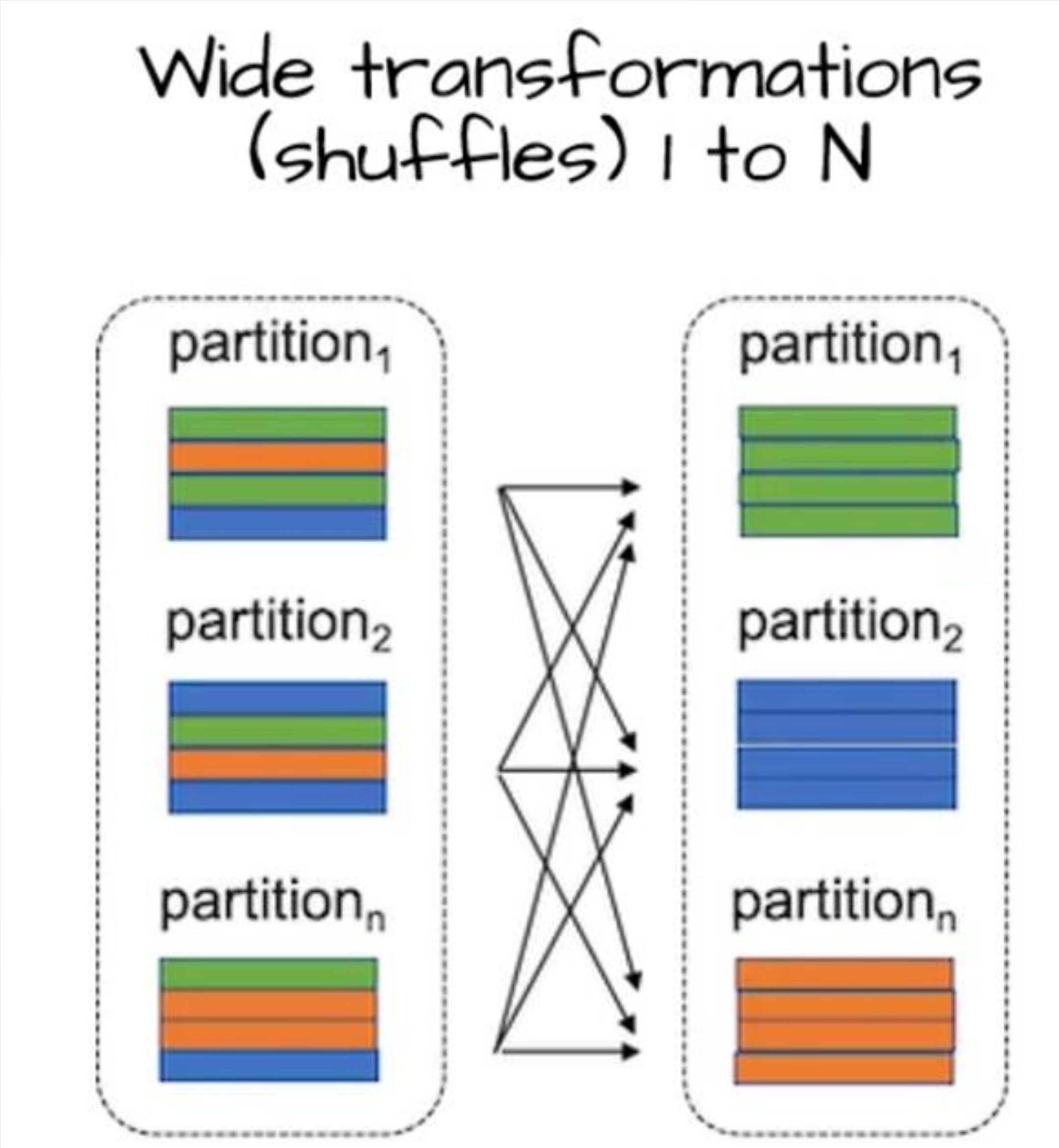
  2. Wide Transformations: applies on a multiple partitions, requires to read other partitions and exchange data between partitions which is called shuffle and Spark has to write data to disk.





Spark's core concepts: Transformations

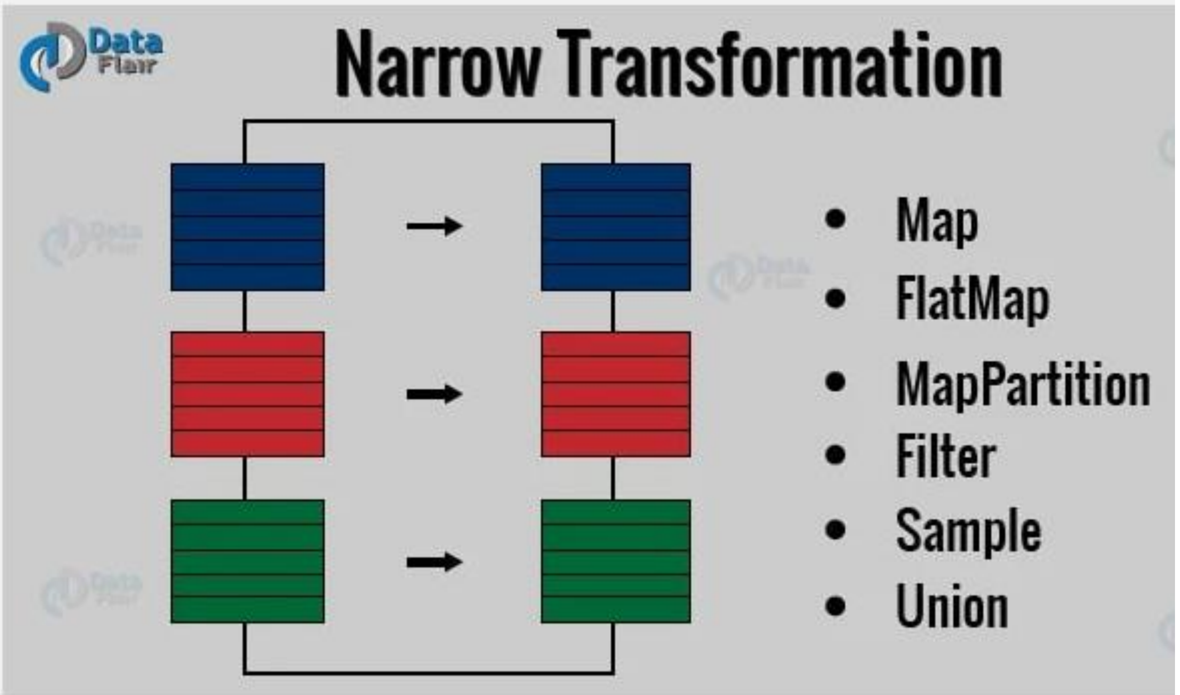


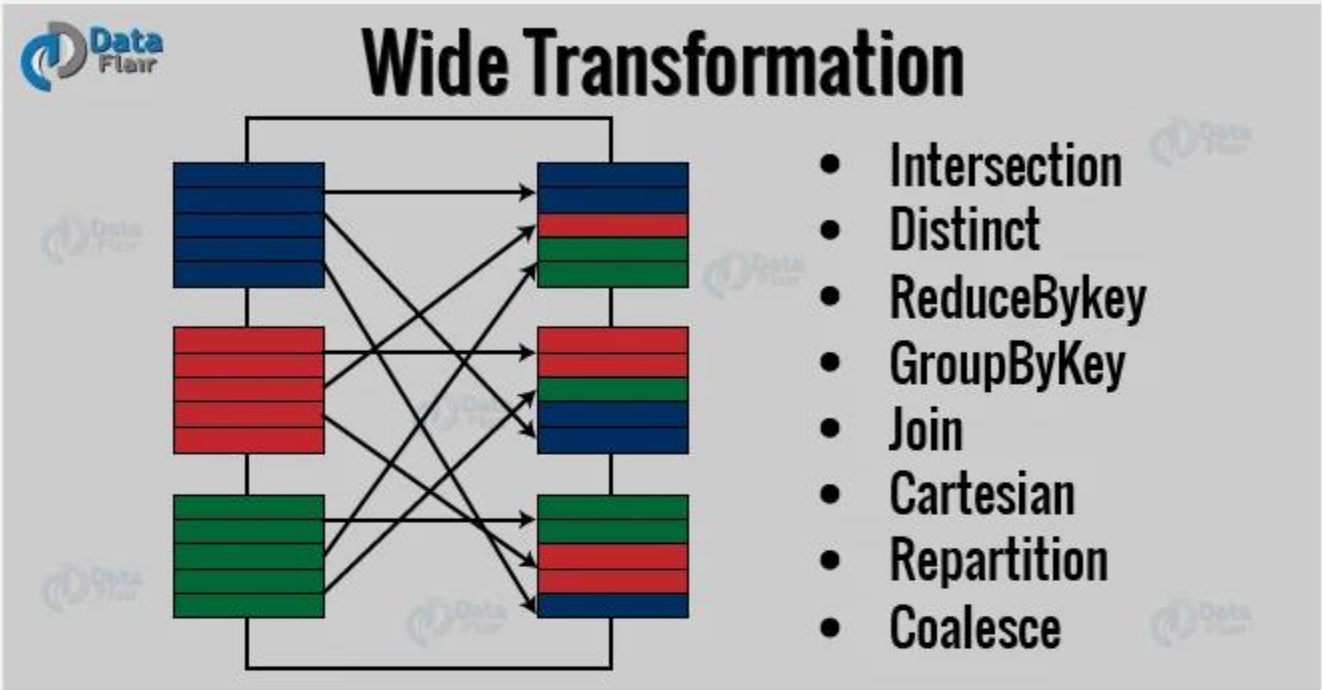


- Two types of transformations : **Narrow dependencies and Wide dependencies**

1. Narrow Transformations: applies on a single partition, for example: filter(), map(), contains() can operate in single partition and no data exchange happens here between partitions.

2. Wide Transformations: applies on a multiple partitions, for example: groupBy(), reduceBy(), orderBy() requires to read other partitions and exchange data between partitions which is called shuffle and Spark has to write data to disk.





Lazy Evaluation

- Spark will wait until the very last moment to execute the graph of computation instructions
- Build up a plan of transformations instead of modifying the data immediately
- Provides immense benefits because Spark can optimize the entire data flow
- Filter example

Spark's core concepts: Actions

- Transformations allow us to build up a **logical transformation plan**

- To trigger the computation of transformation, we run an **action**.
- Action : A spark operation that either returns a result or writes to the disc.

- 3 kinds of actions:
  - Actions to view data in the console
  - Actions to collect data to native objects in the respective language
  - Actions to write to output data sources

- A **Spark job** represents a set of transformations triggered by an individual action, and you can monitor that job from the **Spark UI**

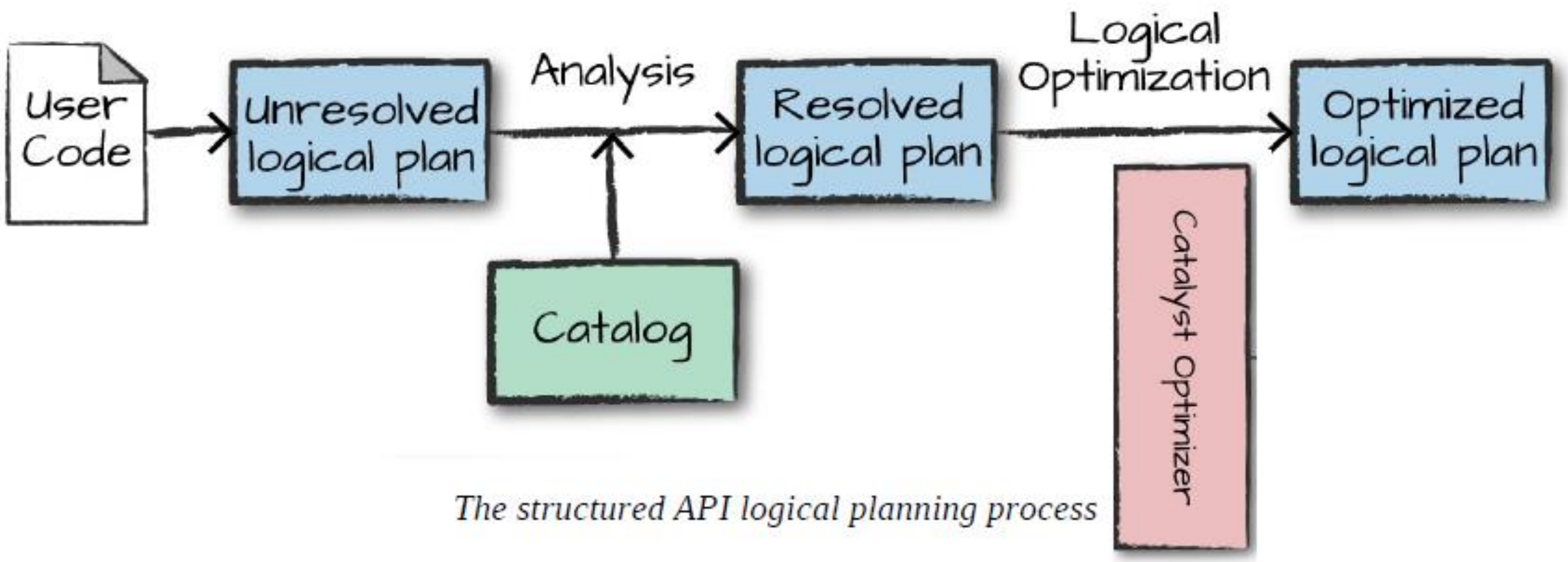
Overview of Structured API Execution

1. Write DataFrame/Dataset/SQL Code.

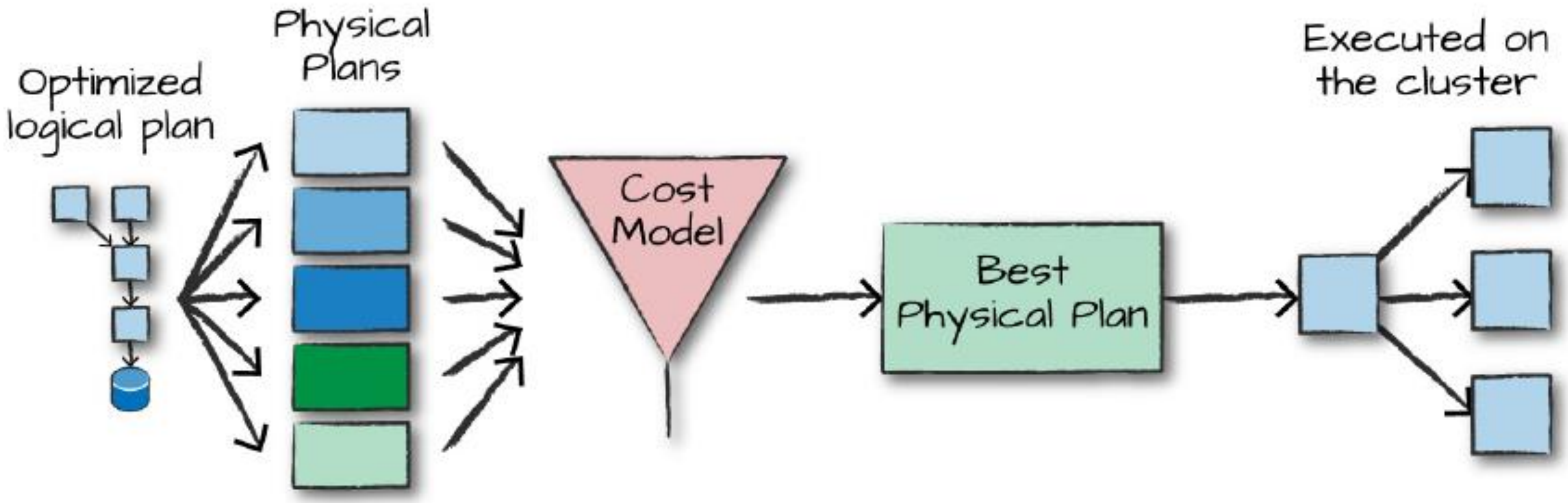
2. If valid code, Spark converts this to a Logical Plan.

3. Spark transforms this Logical Plan to a Physical Plan, checking for optimizations along the way.

4. Spark then executes this Physical Plan (RDD manipulations) on the cluster.



The structured API logical planning process

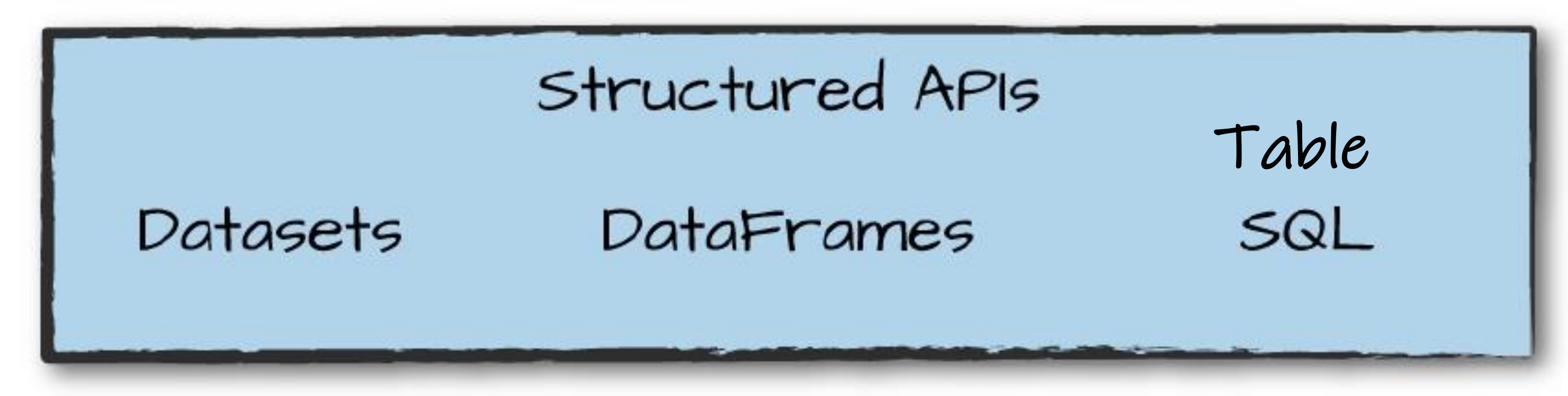


The physical planning process

Spark's core concepts: Spark job, stages, tasks

- **Spark job** represents a set of transformations triggered by an individual action
- **One** Spark job for **one** action. Actions always return results.
- Each job breaks down into a series of **stages**, represent the period during which each partition execute of a series of transformations without needing a shuffle.
- Number of stages in a spark job depends on how many shuffle operations need to take place.
- Each **task** corresponds to a data partition and a set of transformations that will run on a single executor.
- Spark job can be monitored from the Spark UI.

Spark's Structured API



Language	Typed and untyped main abstraction	Typed or untyped
Scala	`Dataset[T]` and DataFrame (alias for `Dataset[Row]`)	Both typed and untyped
Java	`Dataset<T>`	Typed
Python	DataFrame	Generic Row untyped
R	DataFrame	Generic Row untyped

Table Python type reference

Data type	Value type in Python	API to access or create a data type
ByteType	int or long. Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Ensure that numbers are within the range of –128 to 127.	ByteType()
ShortType	int or long. Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Ensure that numbers are within the range of –32768 to 32767.	ShortType()
IntegerType	int or long. Note: Python has a lenient definition of "integer." Numbers that are too large will be rejected by Spark SQL if you use the IntegerType(). It's best practice to use LongType.	IntegerType()
LongType	long. Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Ensure that numbers are within the range of –9223372036854775808 to 9223372036854775807. Otherwise, convert data to decimal.Decimal and use DecimalType.	LongType()
FloatType	float. Note: Numbers will be converted to 4-byte single-precision floating-point numbers at runtime.	FloatType()
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()

```
from pyspark.sql.types import *
b = ByteType()
```

StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	list, tuple, or array	ArrayType(elementType, [containsNull]). Note: The default value of containsNull is True.
MapType	dict	MapType(keyType, valueType, [valueContainsNull]). Note: The default value of valueContainsNull is True.
StructType	list or tuple	StructType(fields). Note: `fields` is a list of StructFields. Also, fields with the same name are not allowed.
StructField	The value type in Python of the data type of this field (for example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]) Note: The default value of nullable is True.

```
from pyspark.sql.types import *
b = ByteType()
```

Data sources

- CSV
- JSON
- Parquet
- Plain-text files
- ORC
- JDBC/ODBC connections

```
spark.read.format("csv")
  .option("mode", "FAILFAST")
  .option("inferSchema", "true")
  .option("path", "path/to/file(s)")
  .schema(someSchema)
  .load()
```

Basics of Reading Data

- DataFrameReader : spark.read (**SparkSession** via the **read** attribute)
- format
- schema
- read mode
- A series of options
- Load('*path*')

Data sources : schemas

- Defines the column names and types of a DataFrame.
- We can either let a data source define the schema (called schema-on-read)
- or define it explicitly ourselves
- A schema is a StructType made up of a number of fields: StructFields
- StructFields have
  - A name,
  - A type,
  - A Boolean flag which specifies whether that column can contain missing or null values

Data sources

Basics of Writing Data

- The DataFrameWriter : DataFrame.write(DataFrame basis via the write attribute)
- format
- ~~schema~~
- read mode
- A series of options
- Load('*path*')

Table 9-1. Spark's read modes

Read mode	Description
`permissive`	Sets all fields to `null` when it encounters a corrupted record and places all corrupted records in a string column called `_corrupt_record`
`dropMalformed`	Drops the row that contains malformed records
`failFast`	Fails immediately upon encountering malformed records

Table 9-2. Spark's save modes

Save mode	Description
`append`	Appends the output files to the list of files that already exist at that location
`overwrite`	Will completely overwrite any data that already exists there
`errorIfExists`	Throws an error and fails the write if data or files already exist at the specified location
`ignore`	If data or files exist at the location, do nothing with the current DataFrame

Basic Structured Operation

Creating DataFrames

With native data

In scala :

-> rows with **Row()**

-> collection with **Seq(**rows**)**

-> RDD with **spark.sparkContext.parallelize(**collection**)**

-> DF with **spark.createDataFrame(**RDD, myManualSchema**)**

In python :

-> rows with **Row()**

-> list with **[**rows**]**

-> -> RDD with **spark.sparkContext.parallelize(**collection**)**

-> DF with **spark.createDataFrame(**RDD, myManualSchema**)**

Basic Structured Operation

select and selectExpr

- Select one column : **.select()**, **.select(col())**, **.select(expr())**

* a column is considered as a expression

- Select multiple columns : **.select()**

- one column or one string manipulation of a column : **.select(expr())**

- One/more column(s) and / or One/more string(s) manipulation(s) of a column : **.selectExpr()***

* most convenent interface for everyday use

Basic Structured Operation

Converting to Spark Types (Literals **lit()**): Using native values

- df.select(expr("*"), **lit(**1**)**.alias("One")).show(2)

Adding Columns :

.withColumn(new column name, expression**)**

- two arguments: the new column name and the expression

Renaming Columns :

.withColumnRenamed(old name, new name**) ***

* Use `` if colname inclus reserved char as in SQL expression

Basic Structured Operation

~~Case Sensitivity in SQL~~

- ~~df.select(expr("*"), lit(1).alias("One")).show(2)~~

Removing Columns :

.drop(col1, col2, …**)**

- two arguments: the new column name and the expression

Changing a Column’s Type (cast) :

col**.cast(**" SQL data type "**)** *

* SQL type not spark type

Basic Structured Operation

Filtering Rows

.filter	(col() ==/!=/>/<...)*
.where	("SQL where clause expression")

Multiple filters : Spark automatically performs all filtering operations at the same time regardless of the filter ordering

* scala : === / =!=

Getting Unique Rows :

.select().distinct(columns)

Random Samples:

.random(bool with replacement, fraction, seed) *

Basic Structured Operation

Random Splits

df.randomSplit(Array(fraction df1, fraction df2…), seed)

Concatenating and Appending Rows (Union) :

df.union(df2)

Sorting Rows

df.sort(col1, col2)

df.orderby(col1, col2)

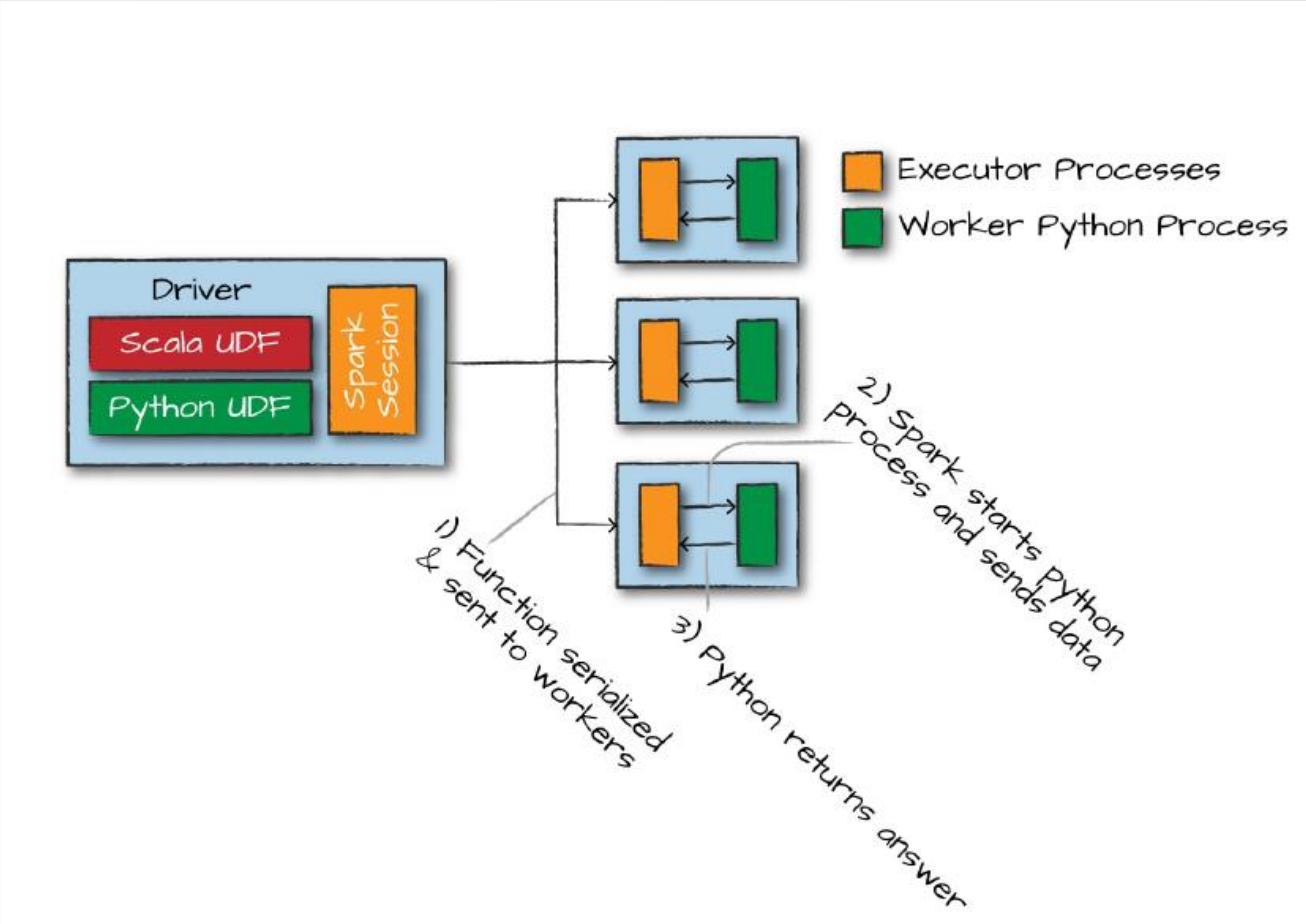
df.orderby(desc(col))

df.orderby(expr(col).desc())

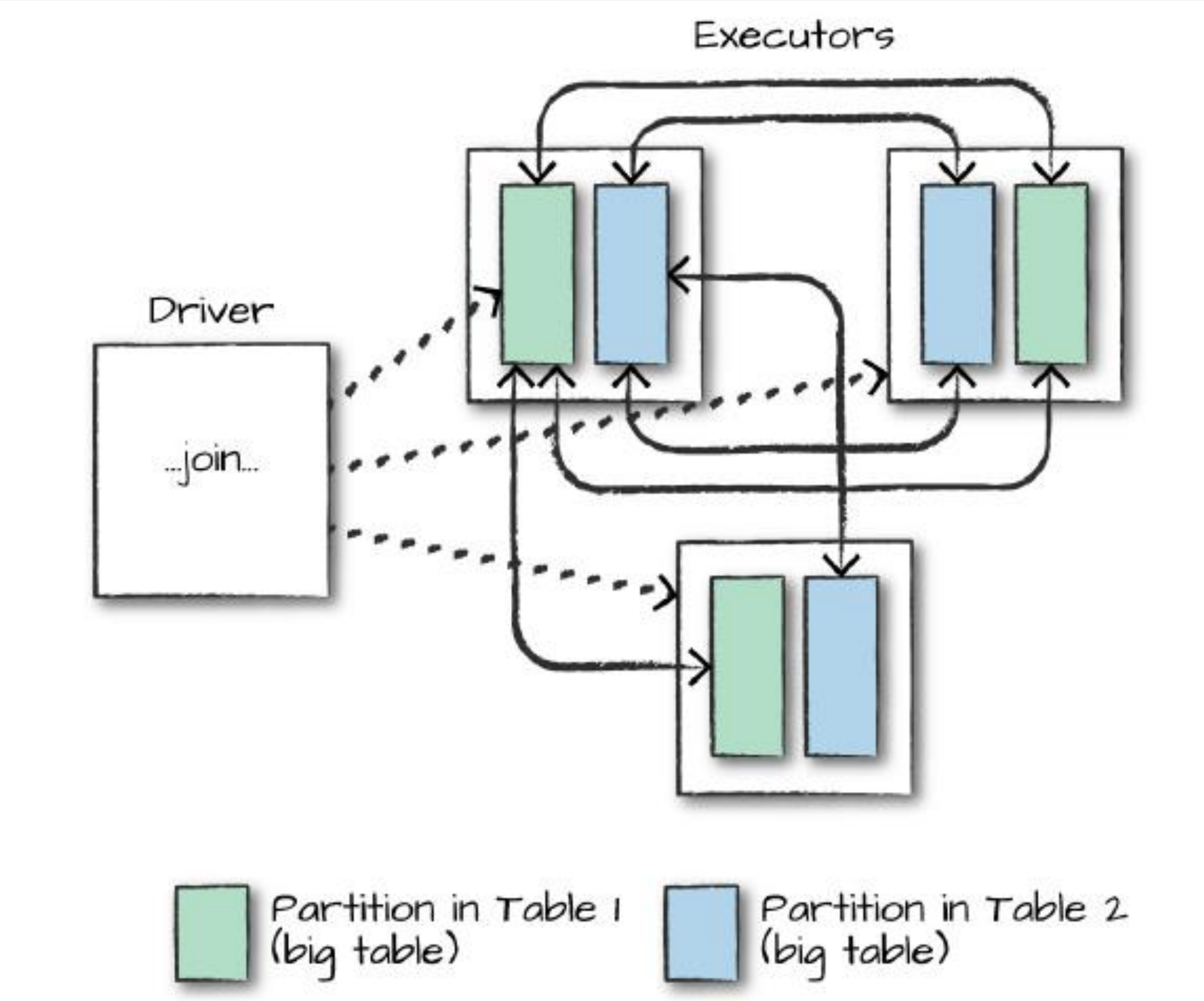
sortWithinPartitions()*

* For optimization purposes, it's sometimes advisable to sort within each partition before another set of transformations

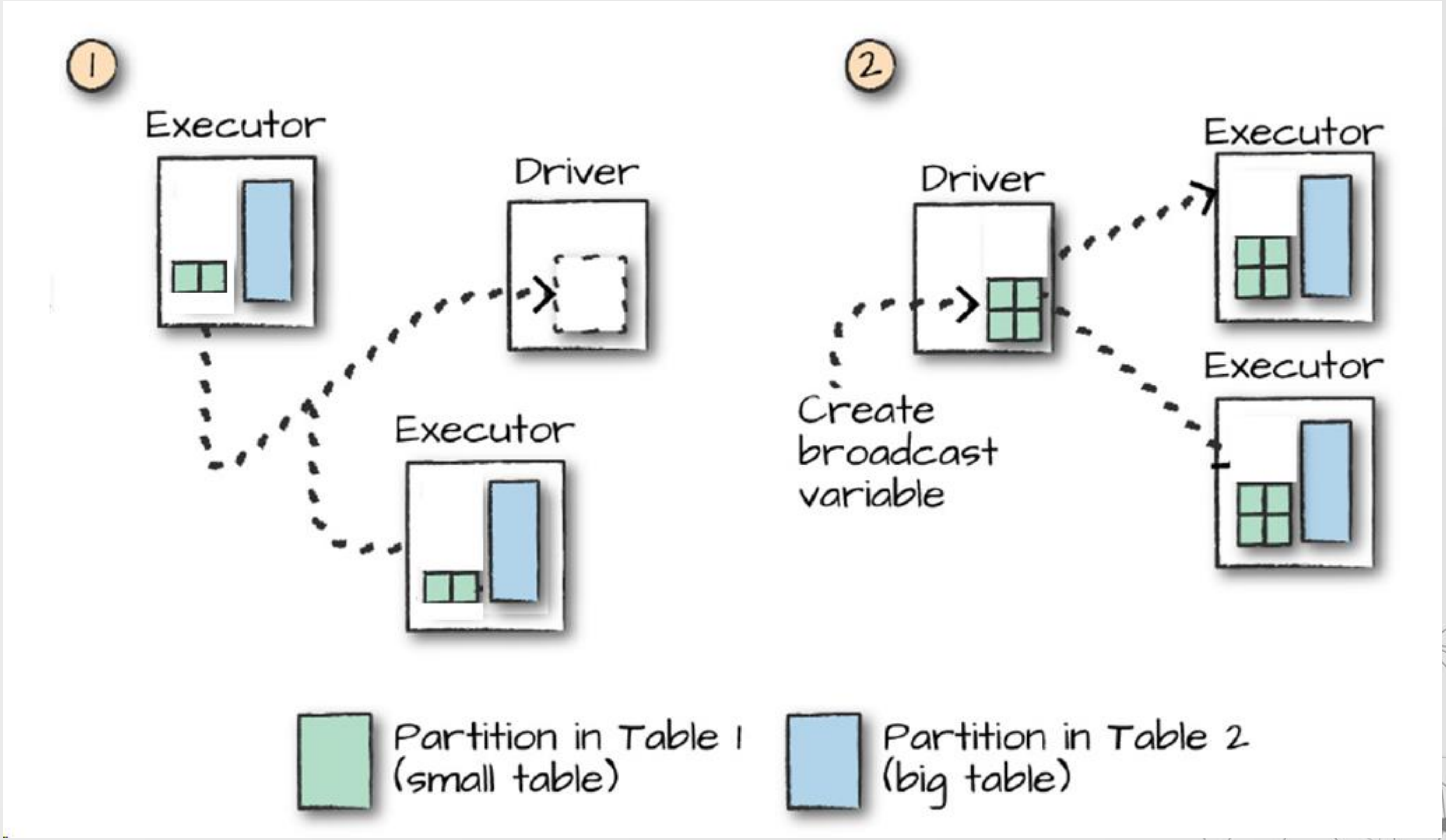
User-Defined Functions



Spark shuffle join : Big table-to-big table



Spark broadcast join : Big table-to-small table



Streaming

- Compute continuously in a production setting (a report about customer activity, or a new machine learning model)
- DStreams API in 2012 (low-level operations)
- In 2016, **Structured Streaming**, a new streaming API built directly on DataFrames that supports both rich optimizations and significantly simpler integration with other DataFrame and Dataset code (which integrates directly with the DataFrame and Dataset APIs)

What Is Stream Processing?

Stream processing is the act of continuously incorporating new data to compute a result.

Stream Processing vs batch processing (computation runs on a fixed-input dataset)

Stream Processing Use Cases

- Notifications and alerting
- Real-time reporting
- Incremental ETL (Extract, Transform, and Load)
- Update data to serve in real time
- Real-time decision making
- Online machine learning

Advantages of Stream Processing

- Enables lower latency : when your application needs to respond quickly or in real time => keep state in memory to get acceptable performance

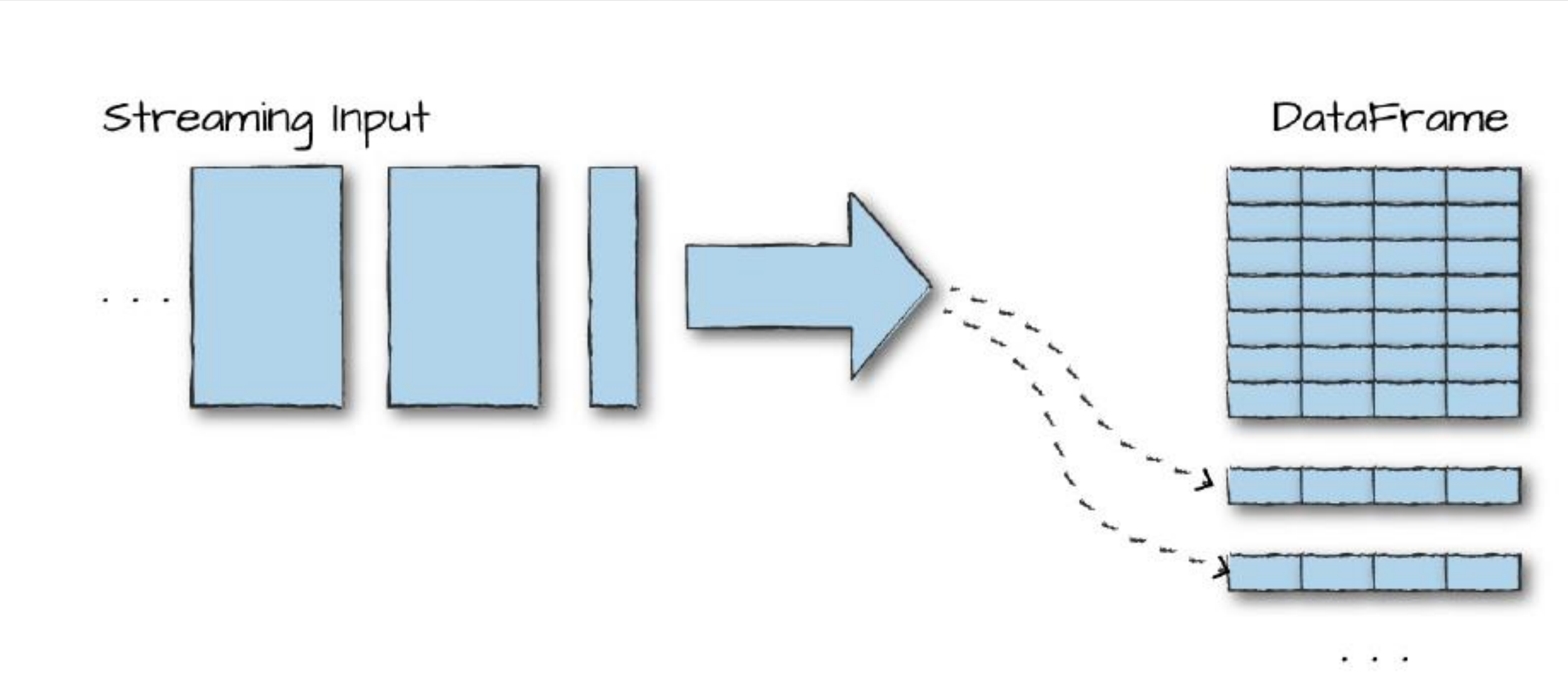
- More efficient in updating a result than repeated batch jobs => Keep state from the previous computation and only count the new data

Structured Streaming Basics

- Stream processing framework built on the Spark SQL engine
- Uses the existing structured APIs in Spark (DataFrames, Datasets, and SQL)
- Users express a streaming computation in the same way they'd write a batch computation on static data.
- Treat a stream of data as a table to which data is continuously appended
- The SS job periodically checks for new input data, process it, updates some internal state located in a state store if needed, and updates its result
- By integrating with the rest of Spark, Structured Streaming enables users to build what we call **continuous applications**

Structured Streaming: Core Concepts

- **Transformations and Actions**
- **Input Sources** Files on file system like HDFS or S3 / Kafka, same as batch
- **Sinks** specify the destination for the result : memory, disk, console…
- **Output Modes** : how we want Spark to write data to that sink
- **Triggers** : define when should check for new input data and update its result
- ***Event-Time Processing*** : *processing data based on timestamps included in the record that may arrive out of order*


Streaming Input
DataFrame

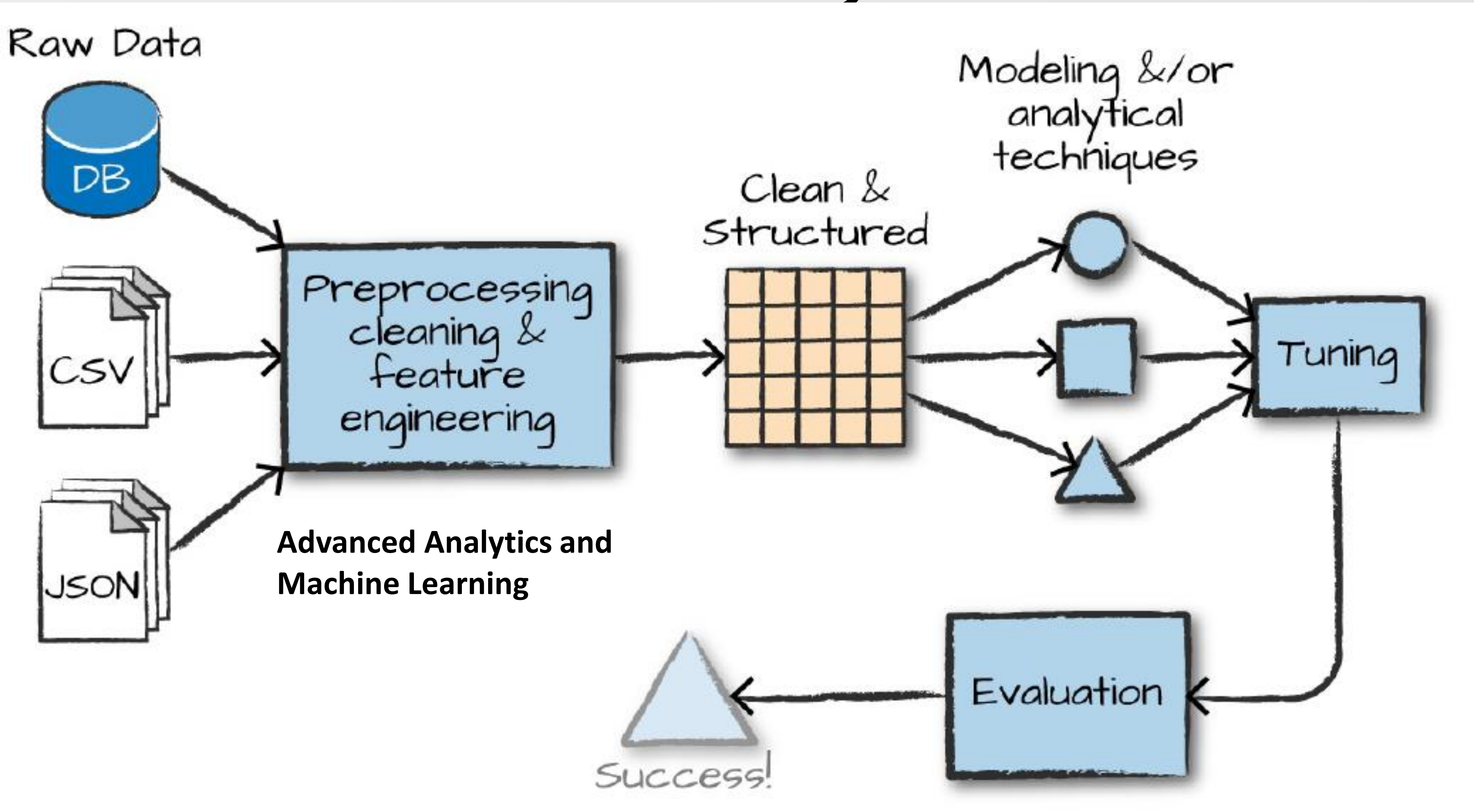
MLlib: Machine Learning and Advanced Analytics

- Package, built on and included in Spark
- provides interfaces for gathering and cleaning data, feature engineering and feature selection, training and tuning
- large-scale supervised and unsupervised machine learning models
- and using those models in production
- Could be used to make predictions in **Strucutred Streaming**
- All machine learning algorithms in Spark take as input a Vector type, which

When and why should you use Mllib ?
versus scikit-learn, TensorFlow, Pytorch…

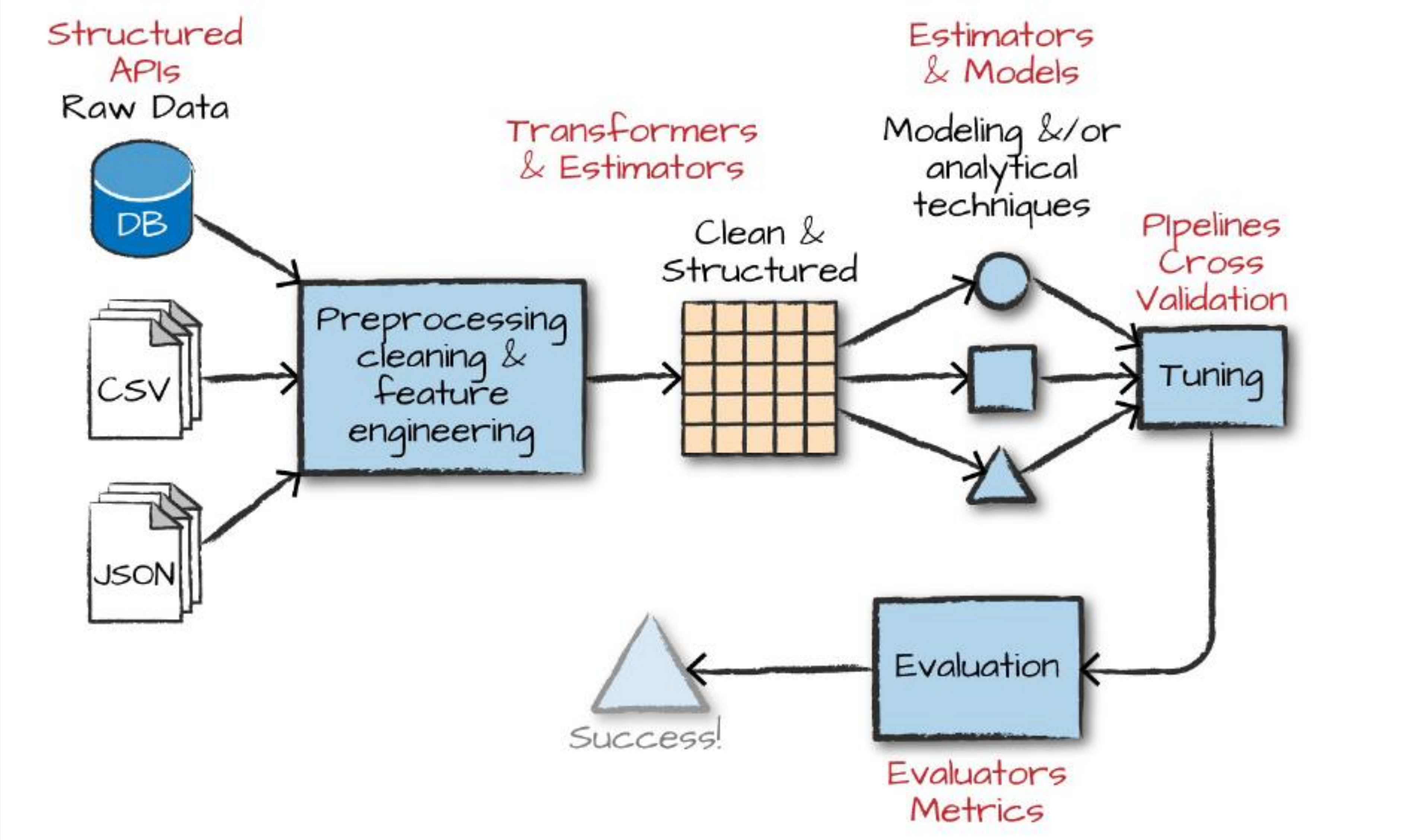
- tools for performing machine learning on a single machine
- have limits either in terms of the size of data you can train on or the processing time
- When you hit those scalability issues, take advantage of Spark's abilities

The machine learning workflow



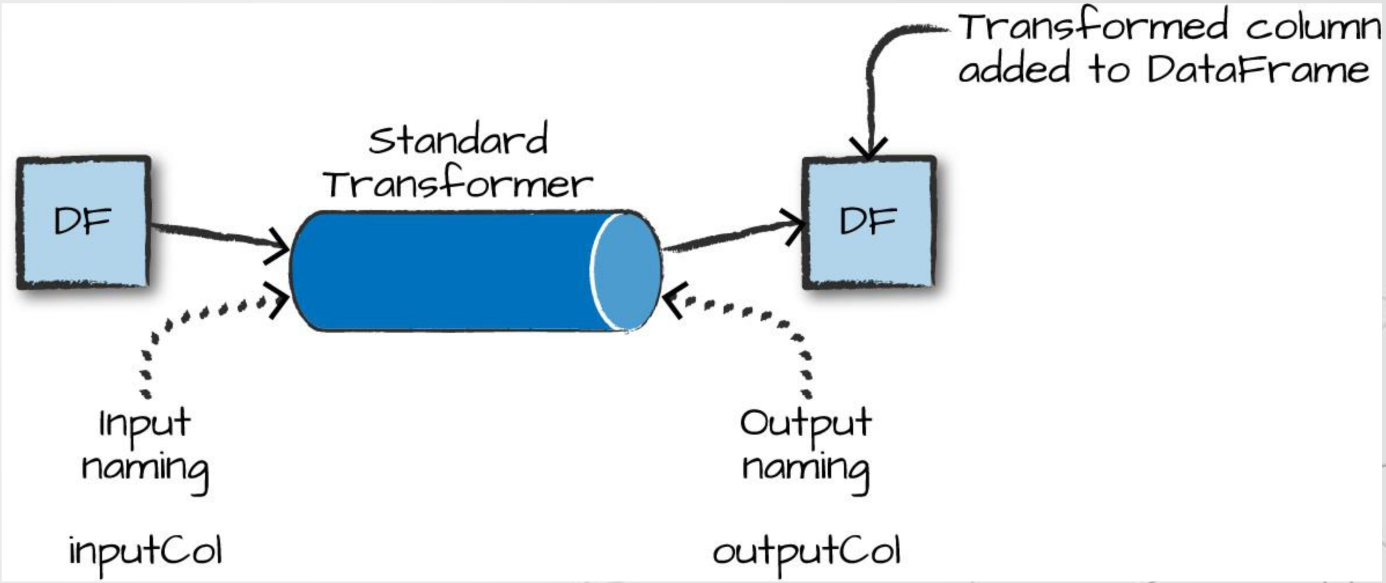
Advanced Analytics and Machine Learning

The machine learning workflow, in Spark



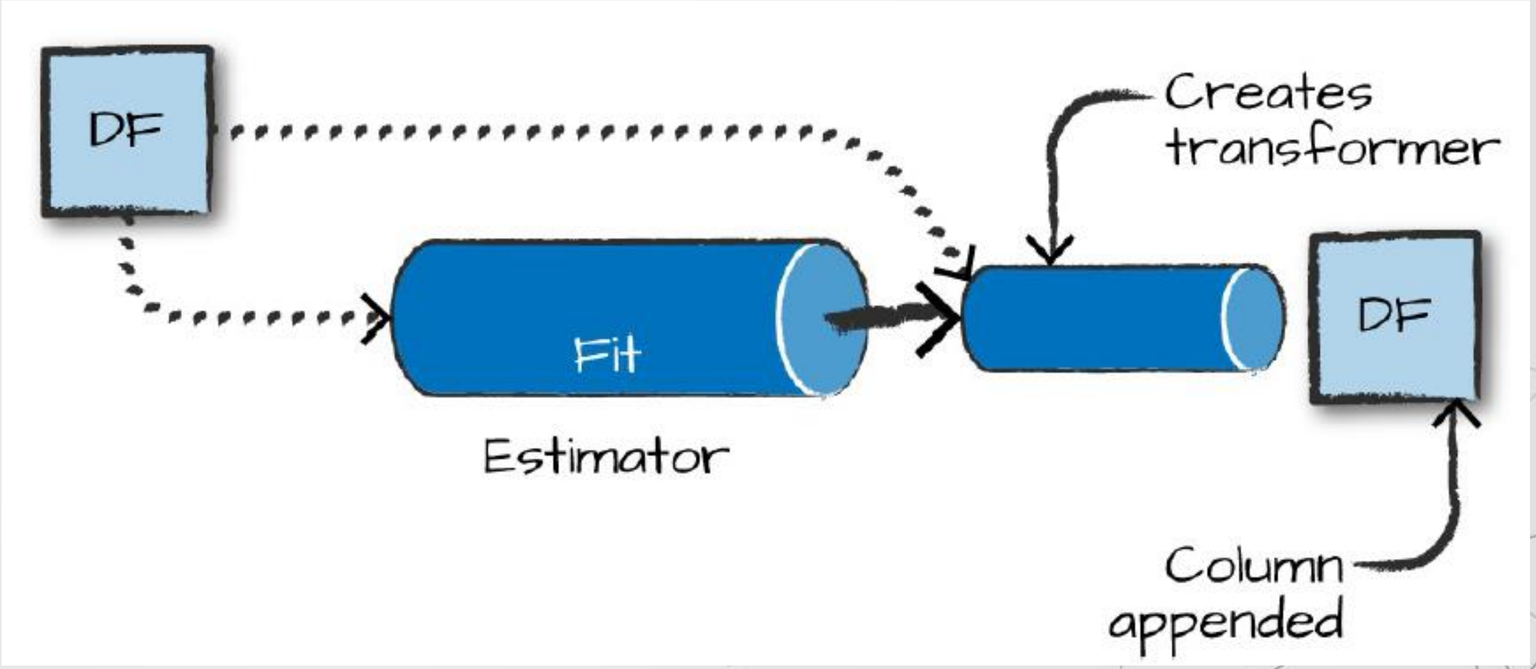
Fundamental "structural" types

- transformers
- estimators
- Evaluators
- pipelines

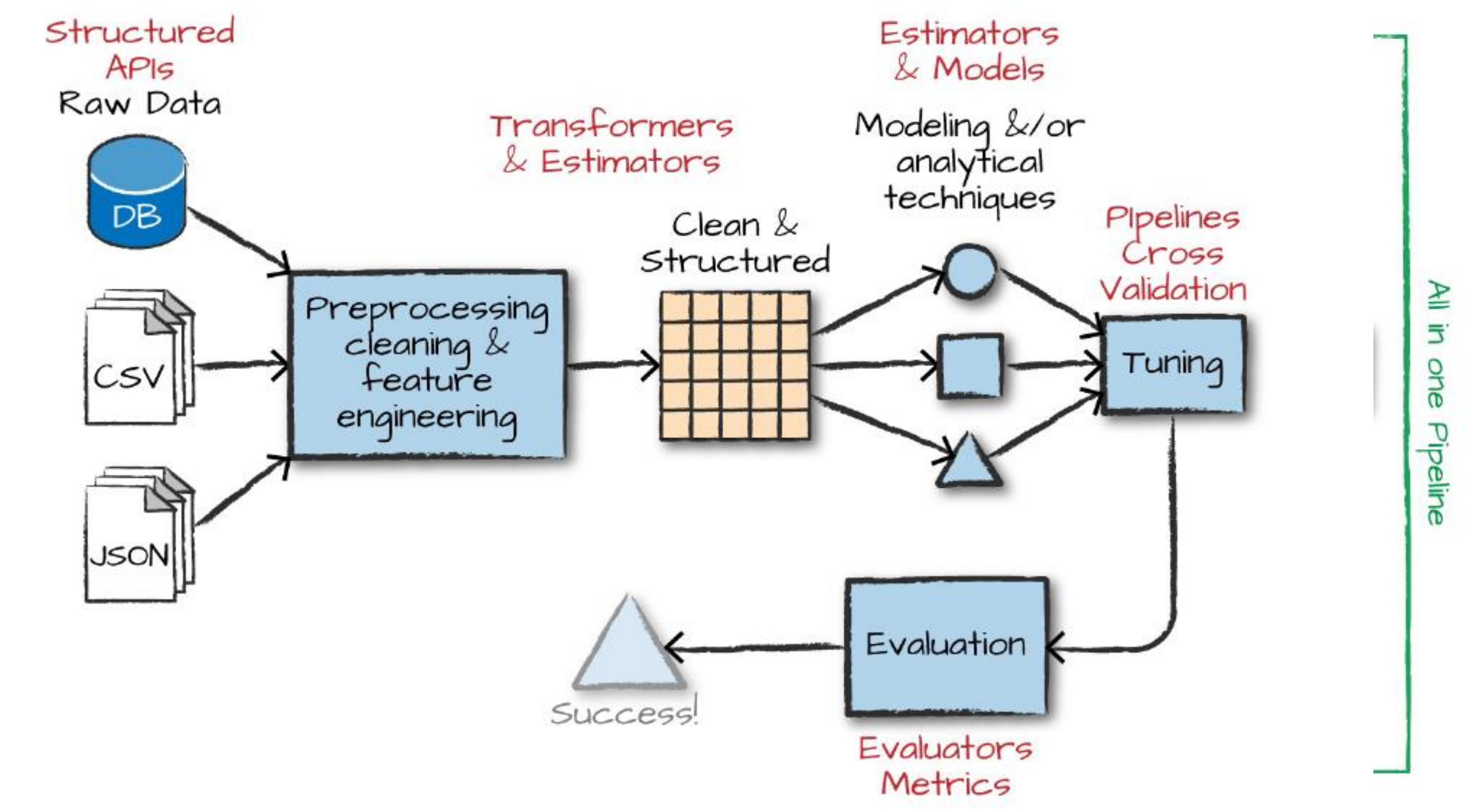


Fundamental "structural" types

- transformers
- estimators
- Evaluators
- pipelines



The machine learning workflow, in Spark



MLlib: Recommendation

- Package, built on and included in Spark
- provides interfaces for gathering and cleaning data, feature engineering and feature selection, training and tuning
- large-scale supervised and unsupervised machine learning models
- and using those models in production
- Could be used to make predictions in **Strucutred Streaming**
- All machine learning algorithms in Spark take as input a Vector type, which must be a set of numerical values.

Regularization

Ordinary least squares regression

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^n} \frac{1}{n} ||\mathbf{X}\boldsymbol{\beta} - \boldsymbol{y}||^2$$

When $\lambda = 0$ (i.e. `regParam` $= 0$), then there is no penalty.

Least Absolute Shrinkage and Selection Operator (LASSO)

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^n} \frac{1}{n} ||\mathbf{X}\boldsymbol{\beta} - \boldsymbol{y}||^2 + \lambda ||\boldsymbol{\beta}||_1$$

When $\lambda > 0$ (i.e. `regParam` > 0) and $\alpha = 1$ (i.e. `elasticNetParam` $= 1$), then the penalty is an L1 penalty.

Regularization

Ridge regression

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^n} \frac{1}{n}||\mathbf{X}\boldsymbol{\beta} - \boldsymbol{y}||^2 + \lambda||\boldsymbol{\beta}||_2^2$$

When $\lambda > 0$ (i.e. `regParam` > 0) and $\alpha = 0$ (i.e. `elasticNetParam` $= 0$) , then the penalty is an L2 penalty.

Elastic net

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^n} \frac{1}{n}||\mathbf{X}\boldsymbol{\beta} - \boldsymbol{y}||^2 + \lambda(\alpha||\boldsymbol{\beta}||_1 + (1-\alpha)||\boldsymbol{\beta}||_2^2), \alpha \in (0,1)$$

When $\lambda > 0$ (i.e. `regParam` > 0) and `elasticNetParam` $\in (0,1)$ (i.e. $\alpha \in (0,1)$) , then the penalty is an L1 + L2 penalty.

Classification scalability reference

Model	Features count	Training examples	Output classes
Logistic regression	1 to 10 million	No limit	Features x Classes < 10 million
Decision trees	1,000s	No limit	Features x Classes < 10,000s
Random forest	10,000s	No limit	Features x Classes < 100,000s
Gradient-boosted trees	1,000s	No limit	Features x Classes < 10,000s

Regression scalability reference

Model	Number features	Training examples
Linear regression	1 to 10 million	No limit
Generalized linear regression	4,096	No limit
Isotonic regression	N/A	Millions
Decision trees	1,000s	No limit
Random forest	10,000s	No limit
Gradient-boosted trees	1,000s	No limit
Survival regression	1 to 10 million	No limit

$$
\begin{array}{c|ccc|}
 & \text{Product 1} & \text{Product 2} & \text{Product 3} \\
\text{user 1} & & & \\
\text{user 2} & & &
\end{array}
\quad
\begin{array}{c}
\text{user 1} \\ \text{user 2}
\end{array}
\begin{bmatrix} 0.5 & ? & 4 \\ 1 & 3 & 5 \end{bmatrix}
=
\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}
\begin{bmatrix} p_1 & p_2 & p_3 \end{bmatrix}
$$

THANKS

Does anyone have any questions?

CREDITS: Fei GAO (EHESP)

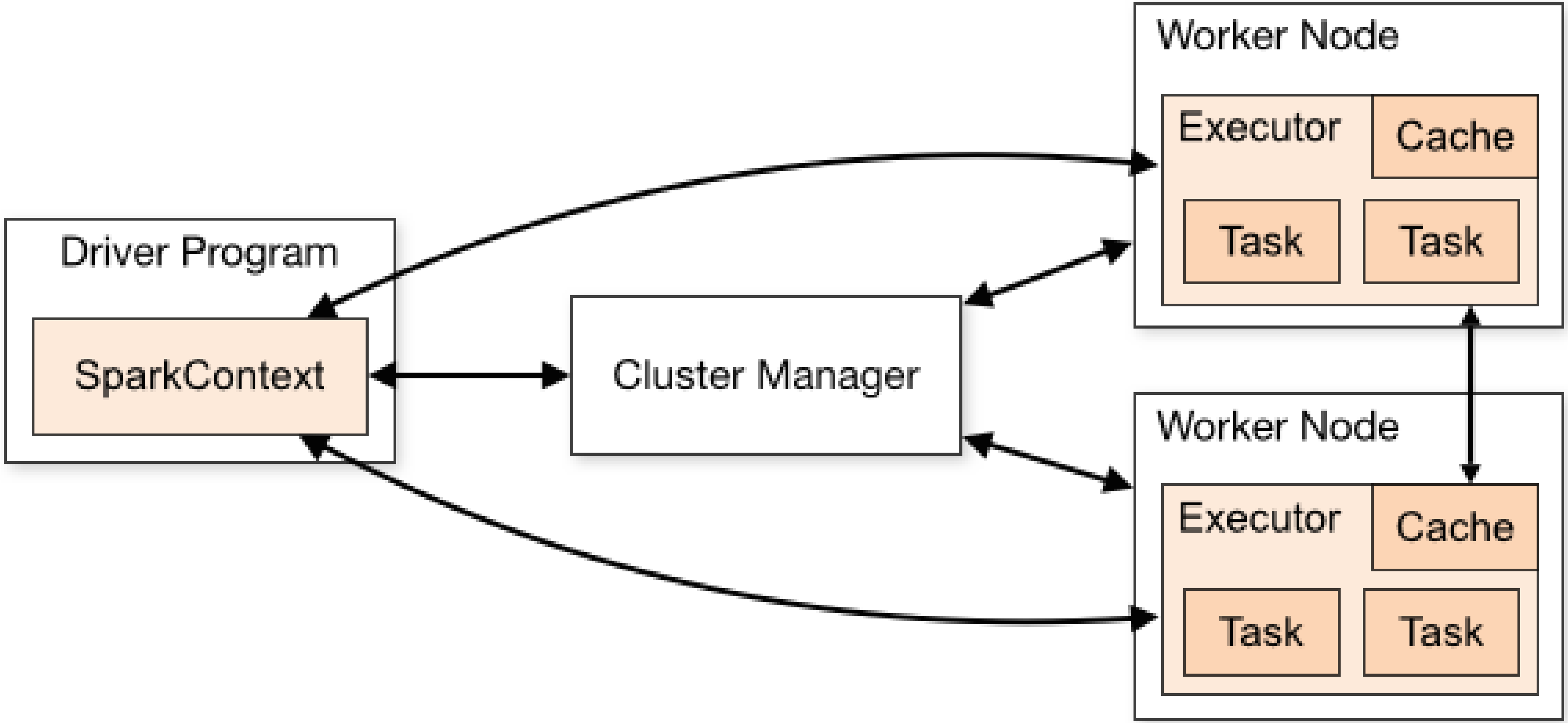
Where to Look for APIs

Spark is a growing project
Where to find functions to transform your data :
Root : https://spark.apache.org/docs/latest/api/scala/index.html

DataFrame (Dataset) Methods : DataFrame is just a Dataset of Row types, so you'll actually end up looking at the Dataset methods
https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/Dataset.html

Specific problems : Dataset submodules like
DataFrameStatFunctions :
https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/DataFrameStatFunctions.html
DataFrameNaFunctions :
https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/DataFrameNaFunctions.html

Worker Node

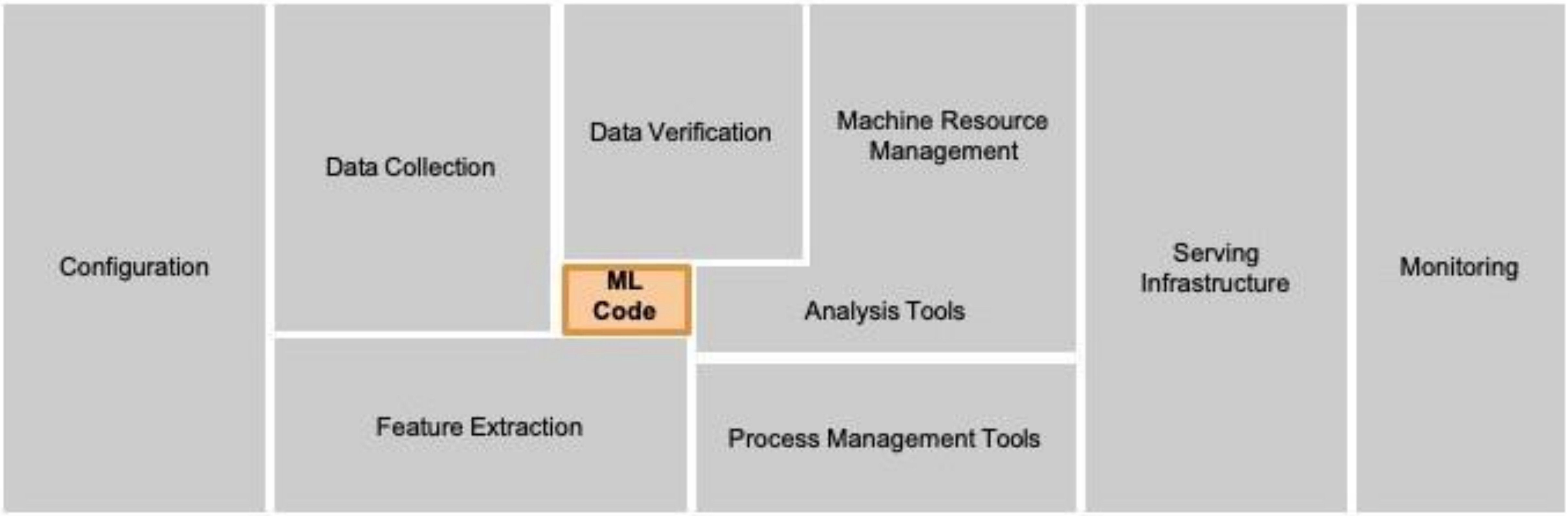


Driver Program
SparkContext
Cluster Manager
Worker Node
Executor
Cache
Task
Task
Worker Node
Executor
Cache
Task
Task

Spark's Ecosystem and Packages

- https://spark-packages.org/

The Requirements Surrounding ML Infrastructure

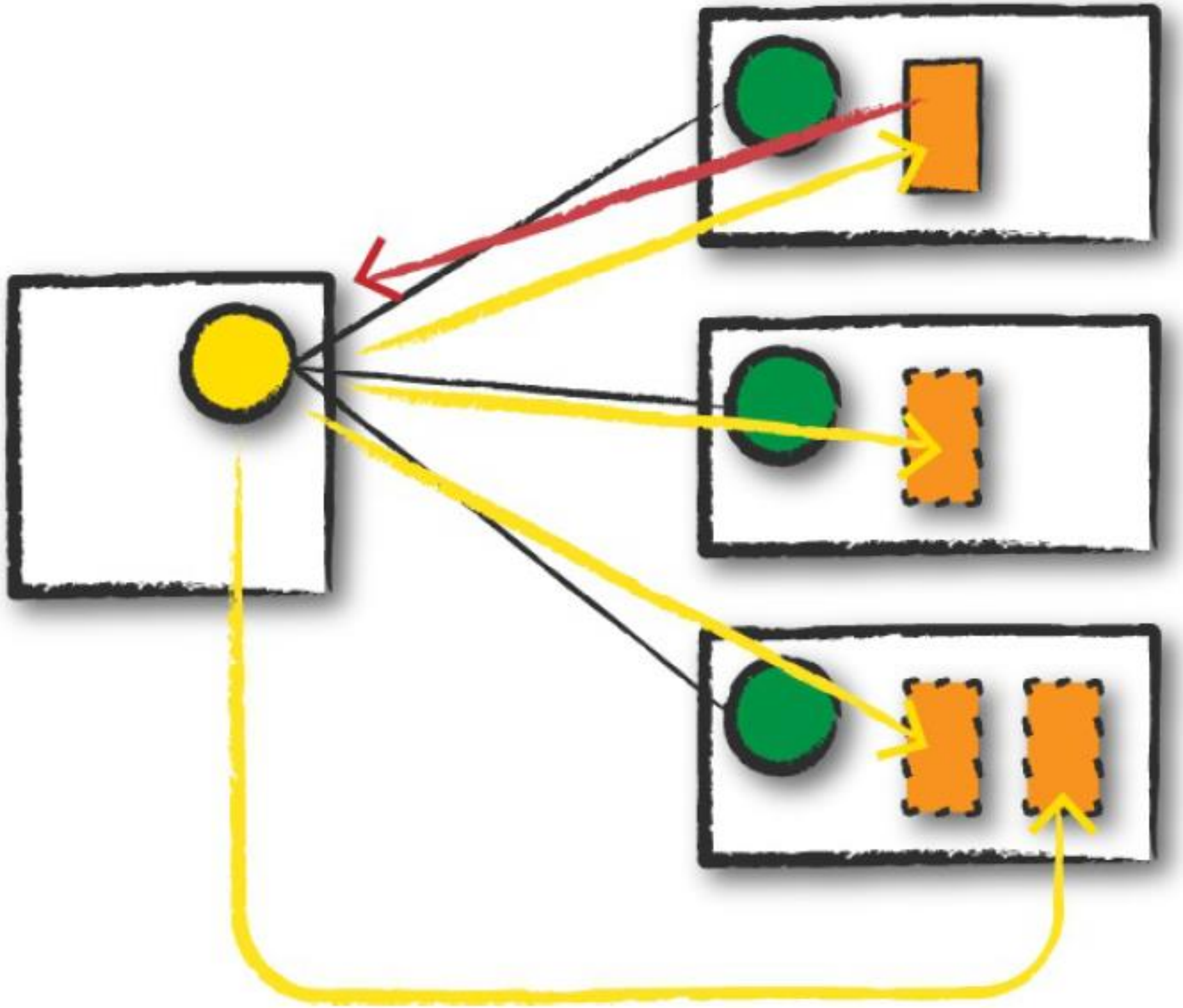


DataFrames vs Datasets:

- **DataFrames**, which chapter, are a distributed collection of objects of type Row that can hold various types of tabular data
- The **Dataset** API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of typed objects, similar to a Java ArrayList or Scala Seq.
- **type-safe** : especially attractive for writing large applications

The architecture of a Spark Application

Cluster mode



The Life Cycle of a Spark Application (Inside Spark)

The Spark Application

- Spark Applications are the combination of two things: a Spark cluster and user-code
- User-code, that defines your Spark Application Exécution
- Each application is made up of one or more Spark jobs
- Spark jobs within an application are executed serially
- In general, one Spark job for one action

The SparkSession

- first step of any Spark Application
- done for you in many interactive modes

The Life Cycle of a Spark Application (Inside Spark)

A Spark Job

- In general, one Spark job for one action
- Actions always return results

Stages

- Each job breaks down into a series of stages
- Group of instructions can be executed together without shuffle
- Instructions creating new data such as read() / range() => new stage

Tasks

- Data and a set of transformations run on a single executor
- Number of tasks in a stage = Number of partitions

The Life Cycle of a Spark Application (Inside Spark)

Pipelining

- Sequence of operations collapsed into a single stage of tasks that do all the operations together
- Much faster than writing the intermediate results to memory or disk after each step
- Transparent to you, the Spark runtime will automatically do it

Shuffle Persistence

- Run this stage later in time than the source stage
- Reduce tasks on failure without rerunning all the input tasks

local mode of Spark?

Spark, in addition to its cluster mode, also has a local mode. The driver and executors are simply processes, which means that they can live on the same machine or different machines. In local mode, the driver and executors run (as threads) on your individual computer instead of a cluster.

The SparkSession

- The SparkSession instance is the way Spark executes user-defined manipulations across the cluster.
- One-to-one correspondence between a SparkSession and a Spark Application

- Languages
- core “concepts” in every language
- These concepts are then translated into Spark code that runs on the cluster of machines.

Programmation très flexible

Exécution des cellules dans l'ordre que l'on veut

Modification de l'ordre des cellules

support des codes, des textes et des images.

Cas d'usage

Interface a utiliser principalement pour l'exploration des données, les Travaux Dirigées

Spark's Streaming APIs

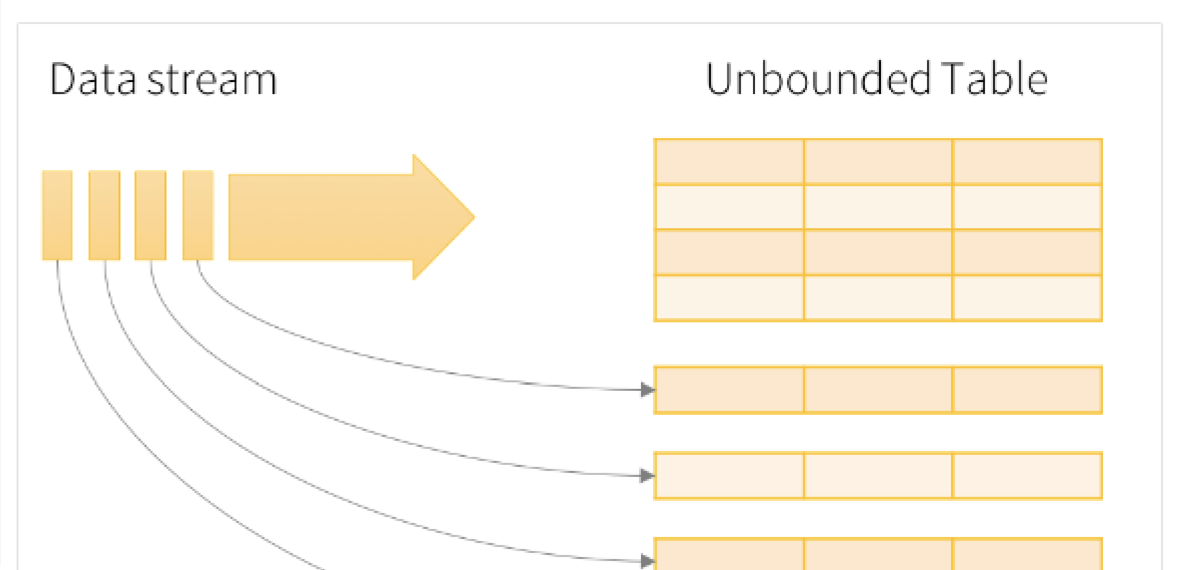
- The earlier **DStream API** in Spark Streaming is purely micro-batch oriented. It is based on Java/Python objects and functions. API but no support for event time.

- The newer **Structured Streaming API** is build on the structured data, adds higher-level optimizations, event time, and support for continuous processing.

- **Structured Streaming** is also designed to make it easy to build end-to-end continuous applications using Apache Spark that combine streaming, batch, and interactive queries

Structured Streaming :

- Sensors, IoT devices, social networks, and online transactions all generate data that needs to be monitored constantly and acted upon quickly. As a result, the need for large-scale, real-time stream processing is more evident than ever before.
- Structured Streaming, the main model for handling streaming datasets in Apache Spark. In Structured Streaming, a data stream is treated as a table that is being continuously appended.

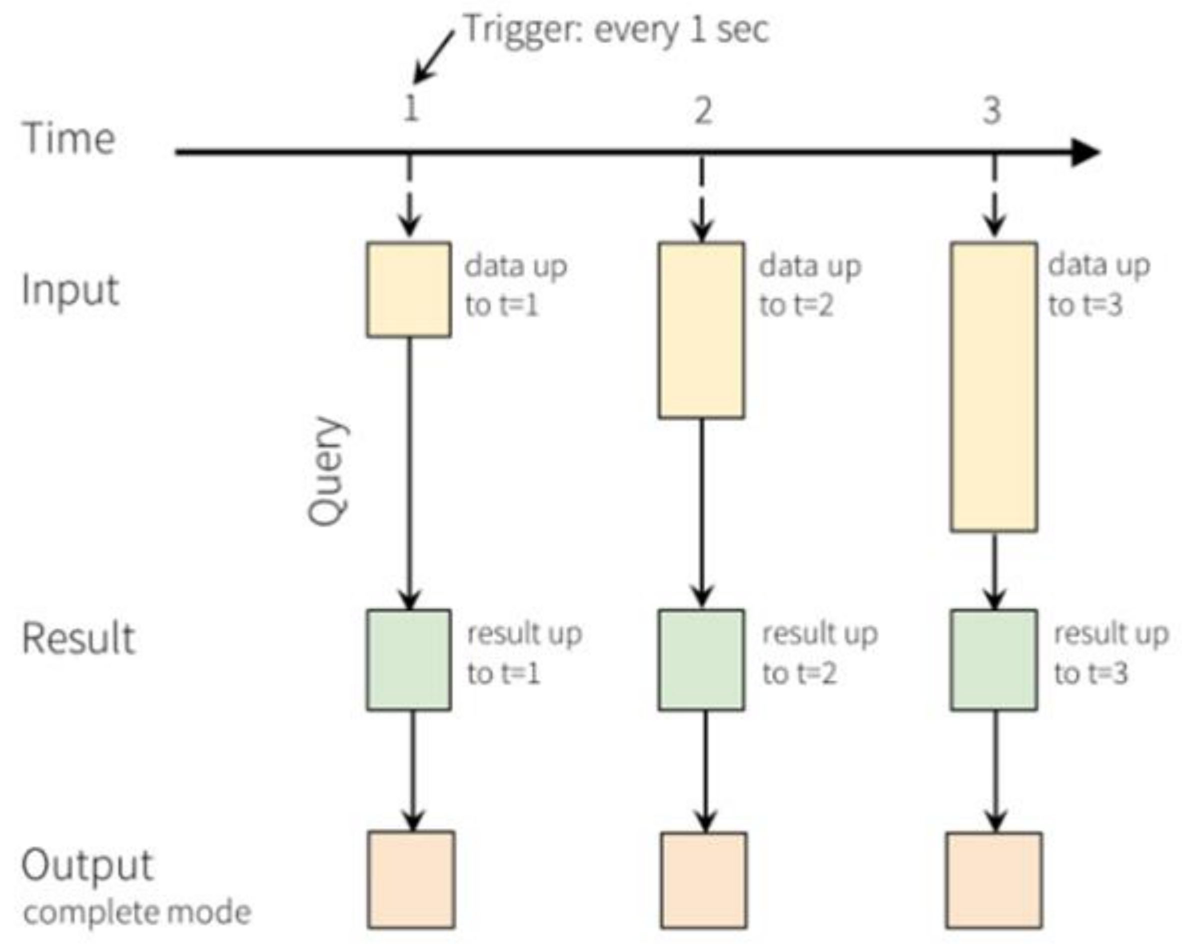


Structured Streaming :

- high-level API for stream processing that became production-ready

- Take the same operations that you perform in batch mode using Spark's structured APIs and run them in a streaming fashion

- Spark permet de traiter des données qui sont figées à un instant T. Grâce au module Spark Streaming, il est possible de traiter des flux de données qui arrivent en continu, et donc de traiter ces données au fur et à mesure de leur arrivée.



Programming Model for Structured Streaming