Spark Higher Level APIs

Why Core APIs (RDD) not used widely anymore ?

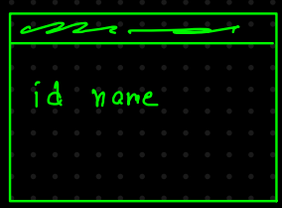
more exploration not required

Schema metadata

1. Verbose!
2. Error-prone : no validation for operations like column access.
3. Performance-limited : Spark engine lacks context about data '1' → 1
   which makes execution inefficient.

header

header != row



What are higher level APIs?

Higher level APIs in spark include Dataframes and Spark SQL which provide schema aware, structured abstraction for distributed data processing.

↳ no dataset in pyspark

1. Schema awareness allows spark to optimize Query execution
2. SQL-like operations makes it accessible to non-programmers.
3. Integration with various data sources (csv, json, Parquet etc.).

Dataframe

- distributed collection of data organized in a structured manner

rdd with some structure

not persistent

Key Benefits

Ease of use

Optimized

Support multiple file formats

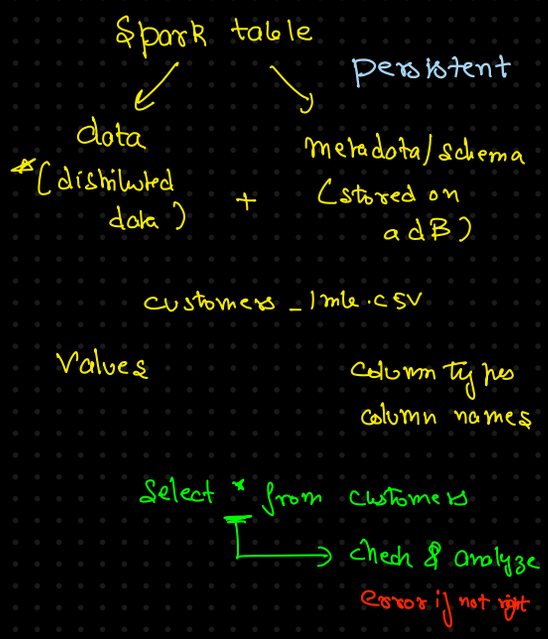
Feature	RDDs	DataFrames
Schema Support	No schema	Schema-aware
Ease of Use	Low	High
Performance	Less optimized	Catalyst-optimized

Spark SQL

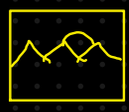
Spark SQL is a module for structured data processing.

It provides an interface for running SQL queries on Spark dataframe and Spark SQL tables.

Feature	DataFrames	Spark SQL
API Type	Programmatic (Python/Scala/Java)	Declarative (SQL)
Persistence	Session-limited	Persistent
SQL Compatibility	Limited	Full SQL support

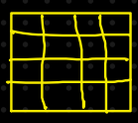


Dataframe in Spark



→ size
→ loc
→ created_at

ls -ltr



columns
data type

a dataframe in sparK is a distributed collection of data organized into named columns, similar to table in a relational database or a spread sheet

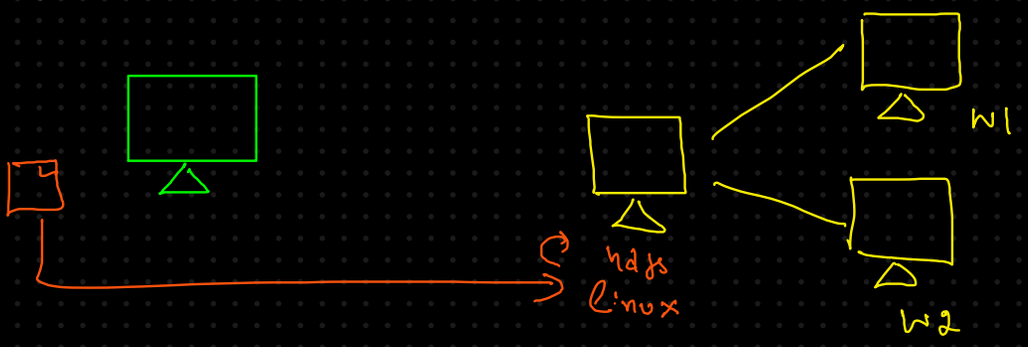
It is an abstraction built on top of rdd with added schema information (metadata).
⇓
data about data

this dataframe is not equal to Pandas Dataframe

Advantage

1. Ease of use
2. Schema awareness
3. Performance
4. Support multiple filetypes → csv, json, parquet

Reading data from HDFS as a dataframe




```
root@my-cluster-m:/# hadoop fs -mkdir /data
root@my-cluster-m:/# ls
'Jupyter Notebooks'      boot            etc      lib32        media                   proc   snap   usr
'Spark Live Class.pdf'   copyright       hadoop   lib64        mnt                     root   srv    var
 Untitled.ipynb          data_type.csv   home     libx32       nested_structure.json   run    sys
 bin                     dev             lib      lost+found   opt                     sbin   tmp
root@my-cluster-m:/# ls
'Jupyter Notebooks'      boot            dev      lib      lost+found              opt    sbin   tmp
'Spark Live Class.pdf'   copyright       etc      lib32    media                   proc   snap   usr
 Untitled.ipynb          data_type.csv   hadoop   lib64    mnt                     root   srv    var
 bin                     dataforhdfs     home     libx32   nested_structure.json   run    sys
root@my-cluster-m:/# hadoop fs -put /dataforhdfs/first_100_customers.csv /data/
root@my-cluster-m:/# hadoop fs -ls /data/
Found 1 items
-rw-r--r--   2 root hadoop       5488 2025-02-03 11:35 /data/first_100_customers.csv
root@my-cluster-m:/#
```

lab
↓
local disk
↓
dataforhdfs
↓ ⇧ → customers.csv
hadoop fs -put ⟵ /data
↓
UI / hadoop fs -ls /data

Option	Description
`format`	Specifies the file format (e.g., `csv`, `json`, `parquet`, etc.).
`header`	Indicates whether the first row contains column names.
`inferSchema`	Automatically infers data types for each column.
`load`	Specifies the file path to load the data.

→ might not always be true

⇒ we should try to provide the schema ourself instead

Reading data in SparK

Interview Question

is reading data an action or a transformation?

When we read we provide options along with format and location

1. header
2. infer schema

1. inferSchema = false but header = true

544mb → 5 partitions

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	load at NativeMethodAccessorImpl.java:0 load at NativeMethodAccessorImpl.java:0	2025/02/03 12:15:02	2 s	1/1	1/1

when we just asked for header, then it ran a small job of collect(1) → to infer the header ⟹ read just a single partition

2. inferSchema = True & header = True

2 Jobs created

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	load at NativeMethodAccessorImpl.java:0 load at NativeMethodAccessorImpl.java:0	2025/02/03 12:20:02	12 s	1/1	5/5
1	load at NativeMethodAccessorImpl.java:0 load at NativeMethodAccessorImpl.java:0	2025/02/03 12:20:01	0.1 s	1/1	1/1

1 job for header
1 for schema

schema

header

3. When we explicitly provide the schema → nothing happens

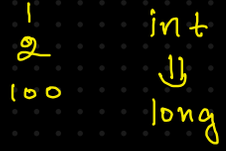
⇒ transformation

Scenario	Behavior	Explanation
Without `inferSchema`, with `header=True`	**Eager Evaluation**	Reads only the first line to determine column names. Triggers a lightweight job (`collect` with `limit 1`).
With `inferSchema=True`	**Eager Evaluation**	Scans the entire dataset to infer schema. Triggers two jobs (1 for column names, 1 for schema inference).
With Explicit Schema Definition	**Lazy Evaluation**	No upfront job is triggered. Schema validation occurs when an action is performed on the DataFrame.

Schema enforcement in Dataframe

Challenges in inferring schema

1. Incorrect inference



2. Performance issue
3. Production - issues ✨

1. Struct Type
2. ddl string

Always make sure to enforce the schema rather than inferring it.

Read Modes in Spark

In production use case, we cannot have a case where columns can be misinterpreted due to corrupt/malformed records which cause issues.

When reading data in Apache Spark, handling corrupt or malformed records is crucial to ensure data quality.

Spark provide 3 read modes to handle such issue.

1. FailFast
   - → stops immediately when a malformed record is encountered.
   - → ensures strict data validation.

Use cases

When strict data integrity is required

2. PERMISIVE (default)
   - → malformed records are processed
   - → allows us to inspect invalid records without stopping execution

When we want to inspect & fix bad data later

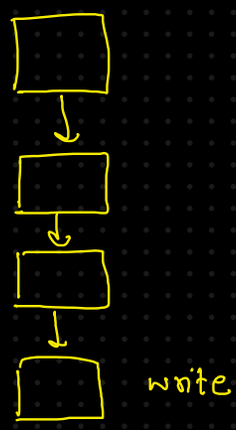
3. DROP MALFORMED
   - → Automatically remove the malformed records
   - → No error, no warning & no trace of dropped records.

When we want to proceed without worrying about incorrect data

Write Operations in SparK

is write an action or transformation?

write operation is considered an action since they trigger computation and save result to external disk.



Spark Dataframe Operation

Spark

Now we have seen how to read and write dataframe with diff options and modes.

Now let us try to understand how exactly a project flows.

Input ⟶ Processing (using spark) ⟶ Output

https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/dataframe.html

Press shift + tab to see fn/operation definition in Jupyter Notebook

Handling different data type in SparK

Different datatypes in spark might require different way of handling.

int float
String
boolean
dates

'mayank'

Data Type	Common Issues
String	Handling inconsistent cases, trimming spaces, and invalid encodings.
Number	Conversion issues, dealing with nulls, and handling precision for floats/doubles.
Date	Parsing formats, handling invalid dates, and timezone mismatches.
Boolean	Variations in input representation (`True/False` , `1/0` , `yes/no`).

Refer the notebooks and ref notebooks for complete code walkthrough.

Date

Spark recognizes the date in format yyyy-mm-dd

1. either we convert every date to above format
2. Read as string and then handle all the formats

Spark Tables

higher level APIs

→ SQL → dataframe

a Spark table provides a structured way to query and store data in a distributed manner.

It is similar to a table in a database but data is distributed across cluster.

data metadata (schema)

df → table

↓

sql

Why spark table:-

1. SQL interface
2. Schema enforcement
3. Temp or a persistent support.

Creating Spark table from dataframe

Type	Description
Temporary Table	Exists only for the duration of the session. Schema and data are stored in memory.
Global Temporary Table	Accessible across all sessions within the same application.
Persistent Table	Stored in a metastore (e.g., Hive), with data saved on disk (HDFS/S3). Accessible across sessions.

metastore → hive

data → hdfs

global temp table

```
df.createOrReplaceGlobalTempView('global_customers')
```

```
spark.sql('show tables').show()
```

```
+---------+---------------+-----------+
|namespace|      tableName|isTemporary|
+---------+---------------+-----------+
|         |temp_customers|       true|
+---------+---------------+-----------+
```

```
spark.sql('show tables in global_temp').show()
```

```
+-----------+----------------+-----------+
|  namespace|       tableName|isTemporary|
+-----------+----------------+-----------+
|global_temp|global_customers|       true|
|global_temp|  temp_customers|       true|
|           |  temp_customers|       true|
+-----------+----------------+-----------+
```

make sure to use global_temp workspace

Spark SQL

data analyst
Non-programmers

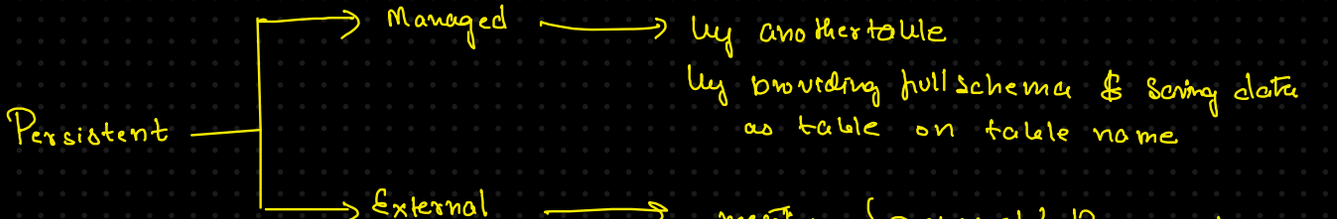
Spark SQL is a module in Apache spark for processing structured data using SQL Queries.

Spark SQL - Managed vs External Table

drop → data ✗ , metadata ✗

Spark SQL supports 2 types of table :-

1. Managed — spark owns both the data and the metadata → restaurant
2. External — spark only owns the metadata . Data is not own and resides at an external location → doordash, swiggy



Please make sure to practice yourself.

Aspect	Managed Table	External Table
Ownership	Spark owns both data and metadata.	Spark owns only the metadata.
Data Location	Stored in the default warehouse directory.	Data resides at a user-defined external path.
Data Deletion	Dropping the table deletes both data and metadata.	Dropping the table deletes only metadata.
Insert Support	Supported	Supported
Update/Delete	Not supported in open-source Spark.	Not supported in open-source Spark.
Use Case	Suitable for single-user or Spark-managed data.	Suitable for shared, externally-managed data.

Creating Dataframe in Spark

Method	Description	Example
`spark.read`	Reads data from external sources.	`spark.read.format("csv").load("/path")`
`spark.sql`	Executes SQL and returns a DataFrame.	`spark.sql("SELECT * FROM table")`
`spark.table`	Accesses an existing table.	`spark.table("customers")`
`spark.range`	Creates a single-column DataFrame with numbers in a range.	`spark.range(0, 10, 2)`
`spark.createDataFrame`	Converts local lists or RDDs to DataFrame.	`spark.createDataFrame(data, schema)`
`rdd.toDF()`	Converts an RDD to a DataFrame.	`rdd.toDF(["col1", "col2"])`
`spark.createDataFrame.toDF()`	Allows renaming columns after DataFrame creation.	`spark.createDataFrame(data).toDF("col1", "col2")`
Explicit Schema	Enforces a schema using `StructType` for better control.	`spark.createDataFrame(data, StructType([...]))`