Université Mohammed V- Rabat
Ecole Mohammadia d'Ingénieurs
Département Génie Informatique
Filière Génie Informatique et Digitalisation

Traitement Big Data

Pr. N. EL FADDOULI
nfaddouli@gmail.com

2023-2024

Chapitre 3: Spark SQL

- ❑ Présentation et architecture de Spark SQL
- ❑ Les différents types de sources de données.
- ❑ SQL, DataFrames et Datasets.
- ❑ Interopérabilité avec les RDD.
- ❑ Performance de Spark SQL.
- ❑ JDBC/ODBC server et Spark SQL CLI.

TP:

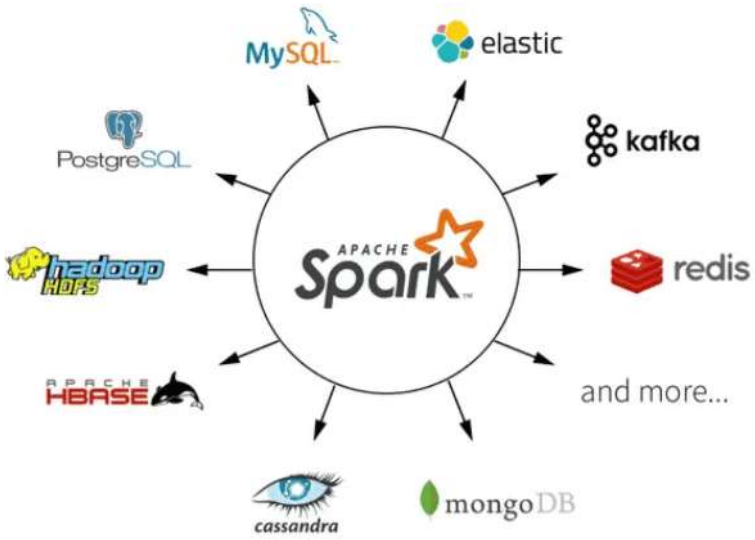
- Manipulation de Datasets via des requêtes SQL.
- Connexion avec une base externe via JDBC.

Présentation

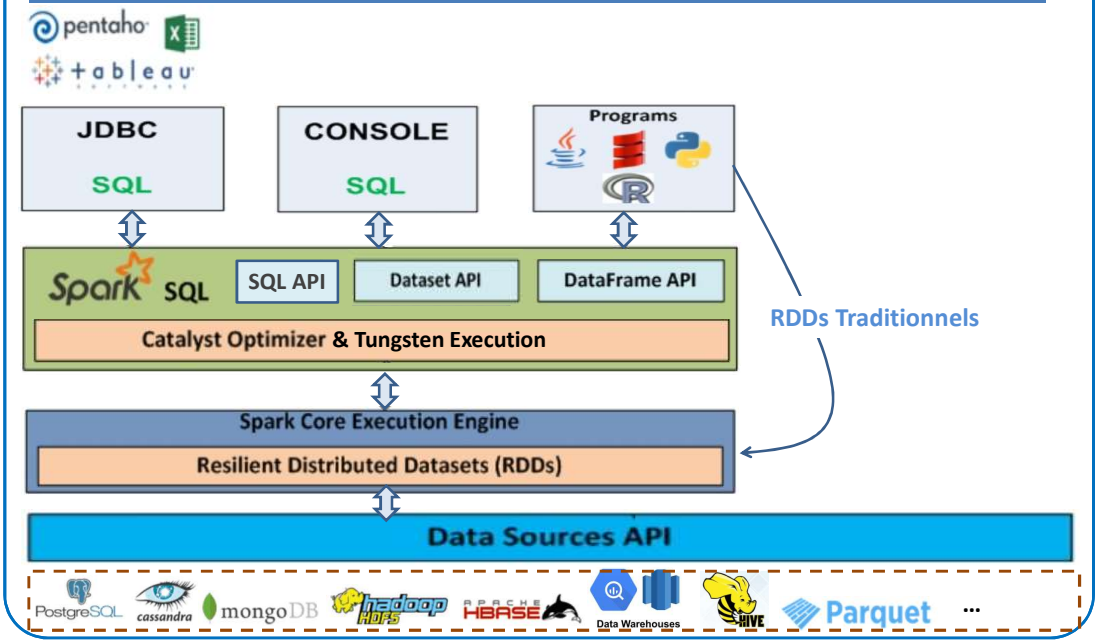
- **Spark SQL** est un module construit sur **Spark Core** et conçu pour le traitement de données **massives structurées** et **semi-structurées** ➔ schéma tabulaire
- Il apporte un nouveau niveau de **facilité**, de **flexibilité** d'utilisation et de **performances**.
- **Facilité d'utilisation** offerte à travers un haut niveau d'abstraction (*schéma tabulaire*) à travers l'**API Dataframe**, et l'utilisation de **SQL**
- **Flexibilité** à travers son **intégration avec des sources de données diverses** comme BDR et Hive, et plusieurs format (Parquet, Avro, ORC, JSON, CSV).
- **Performances** d'exécution assurées par l'optimisation automatique des requêtes par les composants **Catalyst et Tungsten** .

Sources de Données

- Toute source où on peut manipuler ses données selon le module tabulaire (lignes/colonnes).



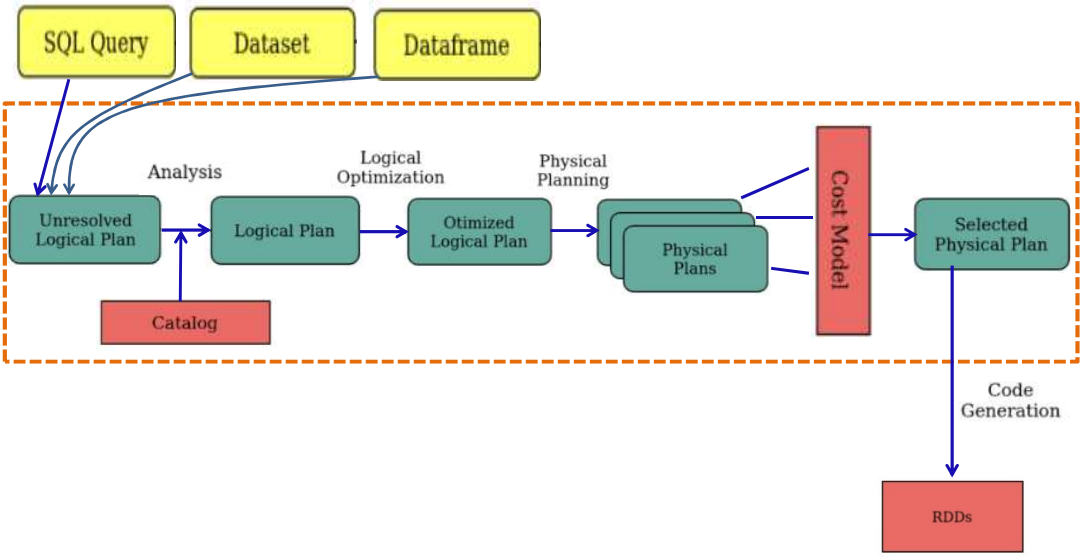
Architecture (1/2)



Architecture (2/2)

- Il expose une interface SQL utilisant **JDBC/ODBC** pour les applications de **Data Warehousing**. Ainsi, tous les outils de Business Intelligence (BI) peuvent se connecter à **Spark SQL** pour effectuer des analyses en mémoire rapides.
- Il expose aussi une **console** interactive pour exécuter le service de Metastore Hive et exécuter des requêtes en ligne de commande.
- Il fournit également l'**API Dataset** et l'**API DaFrame**, qui sont prises en charge dans **Java**, **Scala**, **Python** et **R**. Les requêtes **SQL** effectuées à travers ces deux API sont optimisées par L'optimiseur **Catalyst**.
- On peut utiliser ces API pour **lire** et **écrire** des données depuis et vers **diverses sources** et plusieurs format (***Parquet,Avro, JSON, …***) afin de créer un **DataFrame** ou un **DataSet**.
- Ces **sources de données** sont accessibles via l'**API Data Sources** qui permet la lecture et l'écriture des données dans ces sources.
- On peut également, comme on l'a déjà vu, manipuler les données à travers la création de **RDD** depuis les langages de programmation jusqu'au moteur Spark.

Exécution de requête dans Spark SQL (1/2)



Ref: https://www.data-transitionnumerique.com/spark-sql/

Exécution de requête dans Spark SQL (2/2)

ANALYSE: Plan logique non résolu → Plan Logique

- **Plan logique non résolu** : Le **parseur SQL** ou l'**API Dataframe** crée un *AST(Abstract Syntax Tree*) qui représente la **structure syntaxique** de la requête et qui sera utilisée pour générer le **plan logique non résolu** dans lequel **manquent des détails** (*informations sur les colonnes, leurs types*, ...)
- **Plan Logique**: **Catalyst** génère un **plan logique** de la requête en appliquant une série de règles et en utilisant l'**objet Catalogue** qui peut accéder aux **métadonnées en mémoire** (*colonnes, types*, …) des schémas des données traitées (*tables dans les BD sources ou Dataframe*). On peut ainsi valider les noms et les types des objets utilisés.

Exécution de requête dans Spark SQL (2/2)

OPTIMISATION LOGIQUE: Plan logique → Plan Logique optimisé

- **Plan Logique optimisé**: **Catalyst** applique des **règles d'optimisation** sur le **plan logique** pour l'optimiser. Ces règles peuvent être la **poussée des prédicats**, l'**écrêtage des projections**,..etc.

PLAN PHYSIQUE: Plan logique optimisé → Plan Physique

- **Plan physique**: **Spark** va générer plusieurs plans d'exécution physiques, à partir du plan logique optimisé, dont le coût de chacun sera calculé par le modèle de coût. Ce coût sert à choisir le bon plan physique.

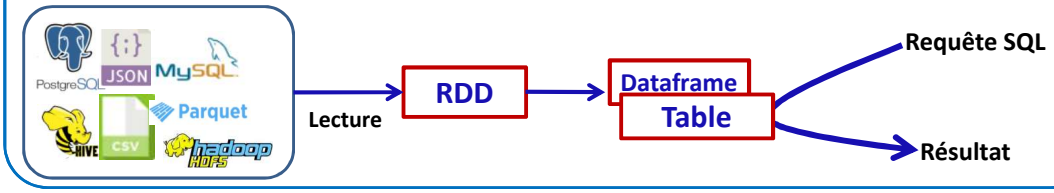
  Le **plan d'exécution physique** prend en compte les caractéristiques spécifiques de la distribution des données et de la configuration du cluster.

GENERATION DE CODE

- L'étape finale consiste à **générer le bytecode Java** qui est envoyé aux nœuds du cluster pour être exécuté et qui manipules des RDDs.

API SQL (1/3)

- L'**API SQL** permet d'exprimer des requêtes en utilisant directement le **langage SQL** avec la fonction **sql()** de la classe **SparkSession**.
- Elle est basée sur le concept de tables et permet d'effectuer des opérations de type **SQL** sur ces **tables distribuées**.
- Elle offre une **abstraction plus élevée** que celle des **RDDs** qui sont **bas niveau**, ce qui signifie que l'utilisateur n'a pas besoin de spécifier **comment les opérations sont exécutées** physiquement.
- On peut enregistrer un **RDD** en tant que **table temporaire** et lui appliquer ensuite des requêtes SQL. (*La table est juste une vue (abstraction) du le RDD*)



API SQL (2/3)

Exemple 1:

```
from pyspark.sql import Row
from pyspark.sql import SparkSession
# Créer un objet SparkSession
spark = SparkSession.builder.appName("Exemple API SQL")\
                            .master("local[*]").getOrCreate()
sc = spark.sparkContext
# RDD des  données (tuples) traitées
rdd = sc.parallelize([(2, "Amine", 26), (3, "Douaa", 34), (1, "Saad", 20)], 2)
# Créer un nouvel RDD où chaque élément est un objet ROW (ligne de table)
row_rdd = rdd.map(lambda x: Row(id=x[0], name=x[1], age=x[2]))
print(row_rdd.collect())
```

```
[Row(id=2, name='Amine', age=26), ....]
```

```
# Créer un Dataframe sur le RDD
df = spark.createDataFrame(row_rdd)
# Créer une table temporaire
df.createOrReplaceTempView("Personnel")
```

```
+---+-----+---+
| id| name|age|
+---+-----+---+
|  2|Amine| 26|
|  1| Saad| 20|
+---+-----+---+
```

```
# Consulter la table temporaire (SQL)
resultat = spark.sql("SELECT * FROM Personnel WHERE id< 3")
# Afficher le résultat
resultat.show()
```

```
for e in resultat.collect():
    print("|", e.id, "|", e.name, "|")
```


API SQL (3/3)

Exemple 2:

```
from pyspark.sql import SparkSession
# Créer un objet SparkSession
spark = SparkSession.builder.appName("Exemple API SQL")\
                            .master("local[*]").getOrCreate()
# Création d'un DataFrame de 3 colonnes sur des données distribuées
data = [(2, "Amine", 26), (3, "Douaa", 34), (1, "Saad", 20),\
        (5, "Laila", 35), (4, "Karima", 21), (6, "Khalid", 17)]
columns = ["id", "name", "age"]
df = spark.createDataFrame(data, columns)
```

```
df = spark.createDataFrame(data)
```

```
Colonnes:  _1 , _2, ...
```

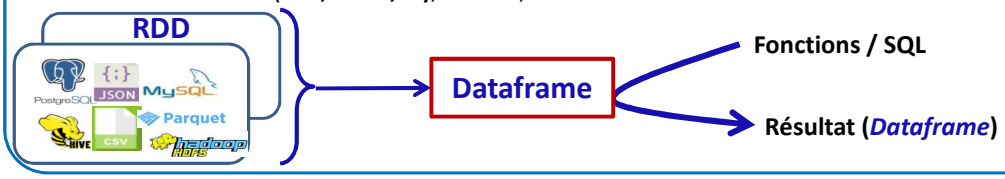
```
# Enregistrer le DataFrame en tant que table temporaire
df.createOrReplaceTempView("Personnel")
# Consulter la table temporaire en SQL
resultat = spark.sql("SELECT * FROM Personnel WHERE id< 4 order by id")
# Afficher le résultat
resultat.show()
```

```
SELECT * FROM Personnel WHERE _1< 4 order by _1
```

```
+---+-----+---+
| id| name|age|
+---+-----+---+
|  1| Saad| 20|
|  2|Amine| 26|
|  3|Douaa| 34|
+---+-----+---+
```

API DataFrame (1)

- L'**API DataFrame** est une interface de **programmation plus orientée objet** qui permet de manipuler des **données structurées** sous forme de **DataFrames**, similaires à ceux que l'on trouve dans d'autres langages comme **R** ou **Python** (*pandas*).
- Un **DataFrame** est une structure organisée en **colonnes nommées** et **non typées.**
- Un **Dataframe** représente une collection de données distribuées **immuables** (*c'est une couche au-dessus des RDD*) avec plus d'**optimisations avancées**.
- Un **Dataframe** contient des métadonnées comme des **informations de schéma** (*colonnes, distribution, …*).
- Les **DataFrames** peuvent être créés à partir de **sources de données variées** telles que des fichiers structurées (**CSV, JSON, …)**, des **BD**, etc ou des **RDD existants.**



API DataFrame (2)

- **Exemple 1**: Charger un fichier **CSV** dans **DataFrame**

```
from pyspark.sql import SparkSession
# Créer une session Spark
spark = SparkSession.builder.appName("Exemple Dataframe").getOrCreate()
# Charger un fichier CSV en tant que DataFrame
df = spark.read.csv("data/foot1.csv", header=True, inferSchema=True, sep=";")
# Afficher le schéma du DataFrame
df.printSchema()
# Afficher les 20 premières lignes du DataFrame
df.show()
```

```
root
 |-- date: string (nullable = true)
 |-- home_team: string (nullable = true)
 |-- away_team: string (nullable = true)
 |-- home_score: integer (nullable = true)
 |-- away_score: integer (nullable = true)
 |-- tournament: string (nullable = true)
 |-- city: string (nullable = true)
 |-- country: string (nullable = true)
```

- **La première ligne contient les noms des colonnes**
- **Spark va inférer le schéma des données à partir d'un échantillon de données au lieu de lui indiquer le schéma.**
- **Le séparateur des champs est ;**

API DataFrame (3)

❑ **Exemple 2**: Charger un fichier **JSON** dans **DataFrame**

```
from pyspark.sql import SparkSession
# Créer une session Spark
spark = SparkSession.builder.appName("exemple").getOrCreate()
# Charger le fichier JSON dans un DataFrame Spark
df = spark.read.json("data/people.json")
# Afficher le schéma du DataFrame
df.printSchema()
# Afficher les premières lignes du DataFrame
df.show()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Spark va inférer automatiquement schéma des données

API DataFrame (4)

❑ **Exemple 3**: Créer un **DataFrame** à partir d'une **collection de données**

```
from pyspark.sql import SparkSession
# Créer un objet SparkSession
spark = SparkSession.builder.appName("Exemple Dataframe")\
                            .master("local[*]").getOrCreate()
# Création d'un DataFrame de 3 colonnes sur des données distribuées
data = [(2, "Amine", 26), (3, "Douaa", 34), (1, "Saad", 20),\
        (5, "Laila", 35), (4, "Karima", 21), (6, "Khalid", 17)]
columns = ["id", "name", "age"]
df = spark.createDataFrame(data, columns)
# Afficher le schéma du DataFrame
df.printSchema()
# Afficher les premières lignes du DataFrame
df.show()
```

```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
```

Spark va inférer automatiquement les types des colonnes à partir de leurs valeurs

API DataFrame (5)

❑ **Exemple 4**: Créer un **DataFrame** à partir d'un **RDD**

```
from pyspark.sql import Row
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Exemple Dataframe")\
                            .master("local[*]").getOrCreate()
sc = spark.sparkContext
# RDD des  données (tuples) traitées
rdd1 = sc.parallelize([(2, "Amine", 26), (3, "Douaa", 34), (1, "Saad", 20)], 2)
# Créer un nouvel RDD où chaque élément est un objet ROW (ligne)
rdd2 = rdd1.map(lambda x: Row(id=x[0], name=x[1], age=x[2]))
# Créer un Dataframe sur le RDD et afficher son schéma
df = spark.createDataFrame(rdd2)
df.printSchema()
# Afficher les premières lignes du DataFrame
df.show()
```

```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
```

On convertit les tuples du RDD en objets de la classe ROW ayant 3 champs nommés que Spark prendra comme colonnes du Dataframe et inférera automatiquement leurs types des à partir de leurs valeurs.

API DataFrame (6)

❑ **Exemple 5**: **StructType** permet de spécifier le schéma du **Dataframe** en cas de **champs complexes** ou pour **imposer la compatibilité des données** par rapport au **schéma indiqué**.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, LongType
spark = SparkSession.builder.master("local[*]") \
    .appName('SparkByExamples.com') \
    .getOrCreate()
data = [("Amine", "Alaoui", "Développeur", 17000),
        ("Imane", "Tahiri", "Architecte",  18000),
        ("Saad", "Sobhi", "Designer",  17000),
        ("Maria", "Faridi", "Graphiste", 16000),
        ("Anas", "Manari", "Développeur",  15000)
        ]
schema = StructType([ \
    StructField("prenom", StringType(), False), \
    StructField("nom", StringType(), False), \
    StructField("fonction", StringType(), True), \
    StructField("salaire", LongType(), True), \
    ])
df = spark.createDataFrame(data=data, schema=schema)
df.printSchema()
```

Si la valeur de l'un des 4 champs a un type différent de celui indiqué, on aura une erreur au moment de la création du DataFrame (exécution).

On n'aura pas cette erreur si on indique juste les noms des colonnes (schema['prenom', ...])

API DataFrame (7)

❑ **Exemple 5 (suite)**: **StructType** pour spécifier un schéma ayant des **champs imbriqués.**

```
Data = [(("Amine", "Alaoui"), "Développeur", 17000),
        (("Imane", "Tahiri"), "Architecte", 18000),
        (("Saad", "Sobhi"), "Designer", 17000),
        (("Maria", "Faridi"), "Graphiste", 16000),
        (("Anas", "Manari"), "Développeur", 15000) ]
structureSchema = StructType([
        StructField(
            'name', StructType([
             StructField('firstname', StringType(), True),
             StructField('lastname', StringType(), True) ])),
        StructField('job', StringType(), True),
        StructField('salary', IntegerType(), True)
         ])
df = spark.createDataFrame(data=Data, schema=structureSchema)
df.printSchema()
df.show( )
df.filter(df.name['firstname'].contains('ne')).show()
```


API DataFrame (8)

Les actions:

❑ **show:** Affiche les première lignes du Dataframe (***par défaut 20***).

Exemple: df.**show()**

df.**show(**10**)**

❑ **count:** Retourne le nombre total de lignes dans le DataFrame.

Exemple: n = **df.count()**

❑ **collect:** Récupère toutes les lignes du DataFrame en tant que tableau local.

Exemple: for **ligne** in df.**collect()**:

print(**ligne.id**, " | ", **ligne.name**, " | ", **ligne.age**)

❑ **write:** sauvegarder le contenu d'un DataFrame dans un fichier

Exemple: df.**write.parquet("**D:\\personnel.parquet**")**:

df.**write.mode("overwrite").csv("**D:\\personnel.parquet**")**:

Mode = **append, overwrite, ignore, error**

API DataFrame (9)

Les transformations:

- **Select:** Sélectionne une ou plusieurs colonnes d'un DataFrame.

  Exemple: df1 = df.**select**("***name***", ***df.age***) ➔ ***df.age , df["age"]*** ou ***"age"***

- **Filter/Where:** Filtre les lignes du DataFrame en fonction d'une condition.

  Exemple: df1 = df.**filter**((df.age > 23) **&** **lower**(df.name).**contains**("am"))

  df.**filter**((df.age > 23) **&** lower(df.name).**like**("%am%")).**show()**

- **GroupBy:** Regroupe les données en fonction de valeurs spécifiques.

  N.B: • Cette transformation retourne un objet **GroupedData** (**pas un Dataframe**)

  • Un groupement est accompagné par un calcul (***fonctions d'agrégation ou autres***)

  Exemple: df1 = df.**groupBy(**"name"**).sum(**"age"**)** ➔ df1[**name, sum(age)**]

  df1 = df.**groupBy(**df.name, df.age**).count()**

On doit importer les fonctions sum et count

df1 = df.**groupBy(**df.name**).agg(sum(**"age"**).alias(**"Somme_Age"**) , count(**"*"**).alias(**"Nombre"**))**

df.groupBy(df.name).**agg({**"age"**: "sum", "*": "count"}).show()**

API DataFrame (10)

- **OrderBy/Sort:** Trie le DataFrame en fonction d'une ou plusieurs colonnes (**asc()** *par défaut*)

  Exemple: df1 = df.**orderBy(**df.name.**desc()**, df.age**)**

- **Distinct:** Retourne les lignes distinctes du DataFrame.

  Exemple: df1 = df.**distinct()**

- **dropDuplicates:** Retourne les lignes distinctes du DataFrame selon une ou plusieurs colonnes

  Exemple: df1 = df.**dropDuplicates([**"name"**,** "age"**])**

- **Join:** Effectue une jointure entre deux DataFrames.

  Exemple: df1.**join(**df2, df1.**id1** == df2.**id2**).**show()**

- **Union:** Combine deux DataFrames ayant le même schéma.

  N.B: • On doit avoir le même nombre de colonnes dans les deux Dataframes

  • Leurs noms et leurs types peuvent être différent, une conversion automatique sera effectuée

  Exemple: df = df1.**union(**df2**)**

API DataFrame (11)

- **Drop:** Supprime une ou plusieurs colonnes du DataFrame.

  Exemple: df1 = df.**drop(**"age", "id"**)**

- **Describe:** calcule des statistiques descriptives pour les colonnes numériques (***max, min, moyenne, nombre et l'écart-type***)

  Exemple: df.**describe(**["age"]**).show()**

- **withColumn: Ajoute** une nouvelle colonne.

  Exemple:

  df. **withColumn(**"identifiant", df.id**).show()** ➔ ajouter **identifiant** de même valeurs que **id**

  df. **withColumn(**"duree", df.age * 2 **).show()** ➔ ajouter la colonne **duree**.

- **Limit:** Limite le nombre de lignes retournées.

  Exemple: df1 = df.**limit(**10**)**

API DataSet(1/3)

- Les **Dataframes** sont une couche au dessus des **RDDs** pour faciliter la manipulation des données sous **format tabulaire** avec **SQL** ou avec des **transformations spécifiques**.
- Un **Dataframe** est basé sur le **typage faible**: il peut **manipuler divers types de données** de manière **flexible**, mais cela peut parfois conduire à des **erreurs lors de l'exécution**.

  Par exemple: on peut faire l'union de deux **DataFrames**, un ayant 3 colonnes de type **entier** et un autre de 3 champs **string**. Le **Dataframe** résultat aura trois champs de type **string**.
- La **vérification des types** n'est faite pas dans la phase de **compilation** (***génération du code***), il n'est faite qu'à l'exécution.
- Un **Dataset** est une collection distribuée de données **typées.** C'est une amélioration du **Dataframe** en y ajoutant le **typage des données**.
- Un **DataFrame** est un **Dataset** contenant des lignes de type **Row** (**DataFrame** = **Dataset**[Row])) dont le schéma précis n'est pas connu.
- Comme un **Dataset** est fortement typé, l'**API Dataset** n'est pas prise en charge dans **Python** et **R** qui sont deux langage faiblement typés pour lesquels l'**API DataFrame** est plus adaptée.
- L'**API Dataset** est disponible dans **Java** et **Scala**.

API DataSet(1/3)

Le Dataset Spark offre plusieurs avantages par rapport au DataFrame, comme:

- **Typage Fort et Statique**: Le **Dataset** est basé sur un **typage fort et statique**, les erreurs liées aux types sont alors **détectées lors de la compilation** plutôt qu'à l'exécution. Cela permet de **réduire les bugs** et d'améliorer la robustesse du code.
- **Performance**: En raison de son typage fort, le Dataset bénéficie d'optimisations de performances supplémentaires, car le compilateur peut générer un code plus efficace en exploitant les informations de type au moment de la compilation.
- **API plus riche**: L'API du Dataset offre des opérations plus riches et expressives par rapport au **DataFrame**, notamment des opérations de type **fonctionnel** telles que **map**, **flatMap**, et **filter**.

TP4: Manipulation des DataFrames et Dataset

Exercice 1:

- Soit le fichier **foot1.csv** de match de foot contenant les colonnes suivantes séparées ";": **date;home_team;away_team;home_score;away_score;tournament;city**
- Ecrire les programmes Python utilisant des **DataFrames** permettant de voir (Sans
  1. Le nombre de matchs par tournoi
  2. Le nombre de match joués dans chaque ville.
  3. Le nombre maximal de buts marqués dans un seul match par pays.
  4. Le nombre de matchs gagnés par pays (équipe)
  5. Les noms des pays ayant marqué le nombre maximal de buts.
- Utilisez un Dataframe pour enregistrer toutes les lignes sous format parquet dans un dossier D:**foo1.parqquet**
- Consulter le contenu de ce dossier.

TP4: Manipulation des DataFrames et Dataset

Exercice 2:

- Lancer le shell spark (pour Scala)
- Importer les classes nécessaires dans Spark-shell, et définir le schéma des données

```
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType};
val customSchema = StructType(Seq(StructField("date", StringType, true),
                    StructField("home", StringType, true),
                    StructField("away", StringType, true),
                    StructField("home_score", IntegerType, true),
                    StructField("away_score", IntegerType, true),
                    StructField("tournament", StringType, true),
                    StructField("city", StringType, true),
                    StructField("country", StringType, true)))
```


TP4: Manipulation des DataFrames et Dataset

- -Créer une classe associée à ce schéma

```
case class match2(date: String, home: String, away: String, home_score:Integer, away_score:Integer, tournament: String, city: String, country: String)
```

- Lire le fichier comme étant un document CSV en y associant la classe créée:

```
val result = spark.read.schema(customSchema).option("delimiter", ";").csv("D:\\foot1.csv").as[match2]
```

- Grouper, faire la somme et afficher le résultat

```
val finalResult = result.groupBy("homee").sum("home_score")
finalResult.collect()
finalResult.show()
```

- Lire le dossier au format parquet créé dans l'exercice 1:

```
val result = spark.read.schema(customSchema).parquet("D:\\foot1.parquet").as[match]
```