Datasets

- Ce que la communauté Spark apprécie :
  - En DataFrame : l'aspect compression et la possibilité d'associer un schéma.
  - En RDD : nature de l'interface de programmation orientée objet à sécurité de type
- Databricks a proposé une extension de l'API DataFrame qui fournit une interface de programmation orientée objet et sécurisée, nommée Datasets.
- Un ensemble de données est une collection immuable et fortement typée d'objets mappés à un schéma relationnel.

Datasets

- Une collection fortement typée d'objets spécifiques à un domaine qui peuvent être transformés en parallèle à l'aide d'opérations fonctionnelles ou relationnelles. En Scala, chaque ensemble de données a également une vue non typée appelée DataFrame, qui est un ensemble de données de Row .

Datasets

- Au cœur de l'API Dataset se trouve un nouveau concept appelé encodeur, qui est responsable de la conversion entre les objets JVM et la représentation tabulaire.
- La représentation tabulaire est stockée à l'aide du format binaire interne Tungsten de Spark, permettant des opérations sur des données sérialisées et une utilisation améliorée de la mémoire.

Datasets

- Spark 1.6 prend en charge la génération automatique d'encodeurs pour une grande variété de types, y compris les types primitifs (par exemple, String, Integer, Long), les case classe de Scala et les Java Beans.
- Les utilisateurs de RDD trouvent l'API Dataset assez familière, car elle fournit bon nombre des mêmes transformations fonctionnelles que RDD (par exemple, map, flatMap, filter).

Datasets

RDD

```
val lines = sc.textFile("/wikipedia")
val words = lines.flatMap(_.split(" ")).filter(_ != "")
```

Datasets

```
val lines = spark.read.text("/wikipedia").as[String]
val words = lines.flatMap(_.split(" ")).filter(_ != "")
```

Datasets

- En Spark 2.0, les améliorations apportées à Dataset sont:

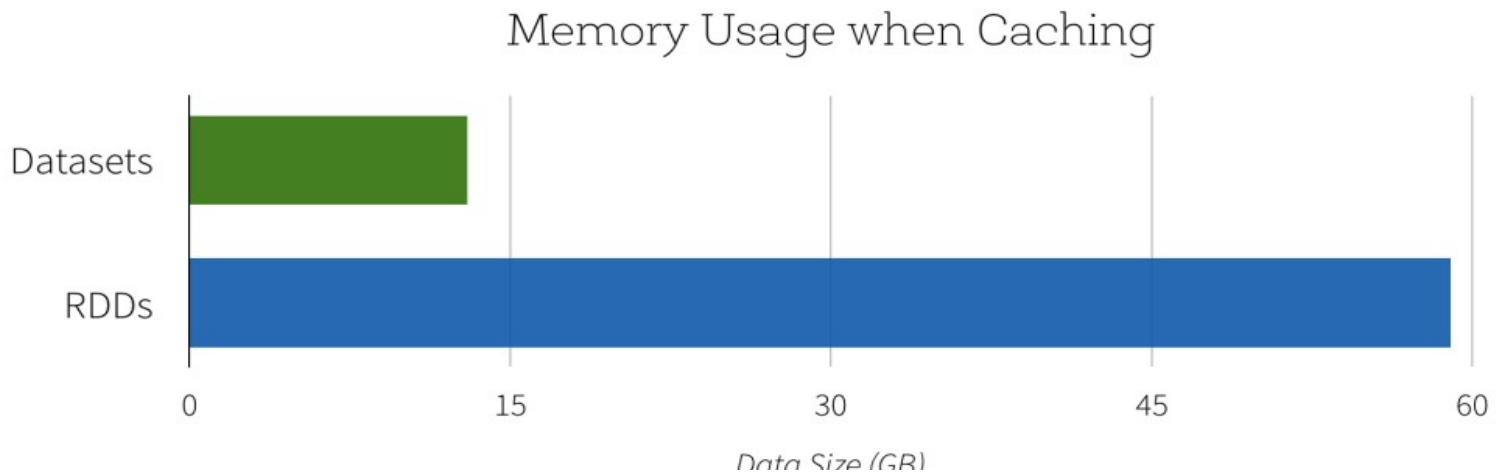
- Optimisations des performances - Dans de nombreux cas, la mise en œuvre actuelle de l'API Dataset n'exploite pas encore les informations supplémentaires dont elle dispose et peut être plus lente que les RDD. Au cours des prochaines versions, Databricks travaillera à l'amélioration des performances de cette nouvelle API.
- Encodeurs personnalisés – alors que Databricks génére actuellement automatiquement des encodeurs pour une grande variété de types, Databricks ouvrira une API pour les objets personnalisés.

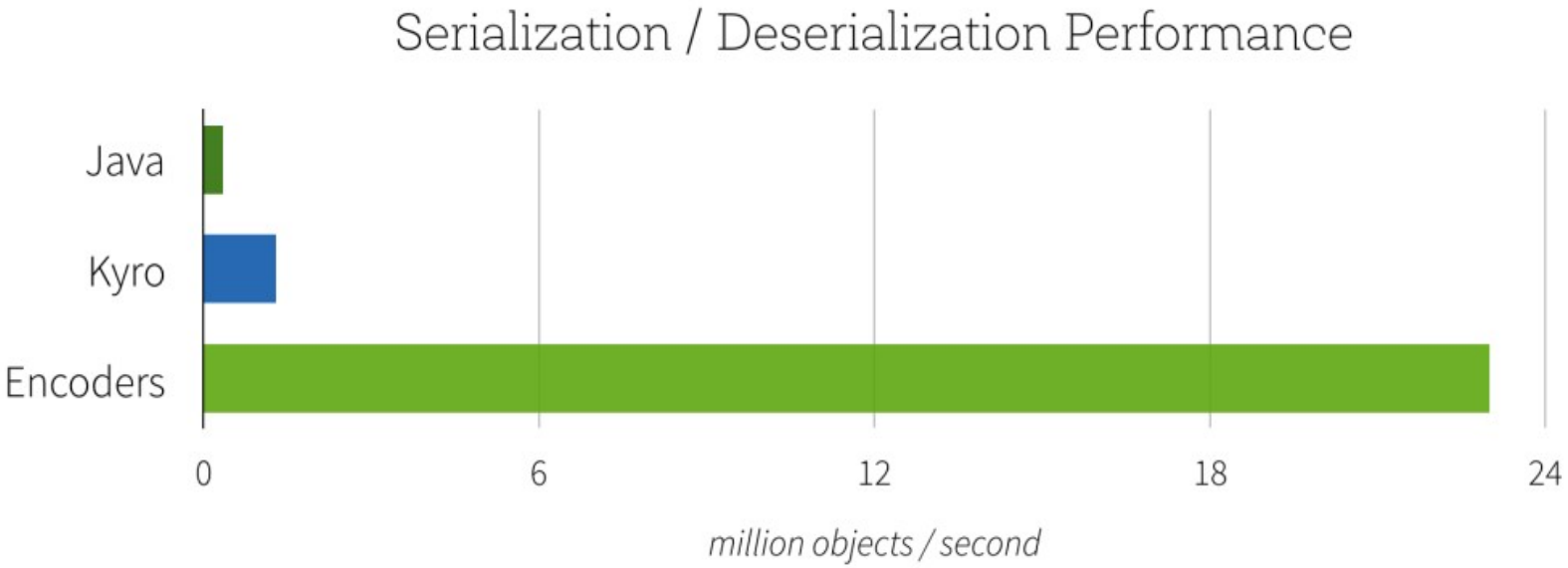
Datasets

- En Spark 2.0, les améliorations apportées à Dataset sont:

- Prise en charge de Python.
- Unification des DataFrames avec les ensembles de données - en raison des garanties de compatibilité, les DataFrames et les ensembles de données ne peuvent actuellement pas partager une classe parente commune. Depuis Spark 2.0, Spark est en mesure d'unifier ces abstractions avec des modifications mineures de l'API, ce qui facilite la création de bibliothèques qui fonctionnent avec les deux abstractions.

Datasets


Memory Usage when Caching
Datasets
RDDs
0
15
30
45
60
Data Size (GB)


Serialization / Deserialization Performance
Java
Kyro
Encoders
0
6
12
18
24
million objects / second

RDD, DataFrames & Datasets

Lire un fichier texte avec textFile

RDD

```
val data = sc.textFile(" nom de fichier ").map( ligne => objet)
```

DataFrame

```
val data = sc.textFile(" nom de fichier "). ").map( ligne => nuplet).toDF(“att1”,””att2”,..)
```

Dataset

```
val data = sc.textFile(" nom de fichier "). map( ligne => objet).toDS()
```

RDD, DataFrames & Datasets

Original file : 151.2MB

Lines, resp. DS, DF and RDD

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
Scan ExistingRDD[_1#57,_2#58,_3#59]	Memory Deserialized 1x Replicated	5	100%	20.3 MB	0.0 B
*Project [_1#3 AS s#7, _2#4 AS p#8, _3#5 AS o#9] +- Scan ExistingRDD[_1#3,_2#4,_3#5]	Disk Serialized 1x Replicated	5	100%	20.3 MB	16.9 MB
rdd	Memory Deserialized 1x Replicated	4	80%	277.9 MB	0.0 B

RDD, DataFrames & Datasets

Transformations

DF vers RDD

```
val data = myDF.rdd
```

DF vers DS

Dataset of tuples

```
val ds = data.as(String, Long, String)
```

Dataset of Objects

```
case class Triple (s: String, p: Long, o: String)
val ds= data.as(Triple)
```

Schéma d’une collection

- Pour obtenir le nom et le type des attributs d’un DataFrame ou DataSet
  - `data.columns`
- Pour renommer un attribut
  - `val data2 = data.withColumnRenamed(“old”,”new”)`

Quelques opérations sur les Dataset

- Actions
  - count, describe, reduce, show
- Transformations
  - distinct, map, filter, flatMap, orderBy, groupByKey, except, join, select
- Fonctions
  - rdd, dtypes, printSchema

Usage

- Utiliser RDDs quand vous avez besoin
  - De fonctionnalités non présentes dans les API structurées
  - pour maintenir une base de code héritée écrite à l'aide de RDD
  - pour faire une manipulation de variable partagée personnalisée
- Pour la grande majorité des cas d'utilisation, les DataFrames seront plus efficaces, plus stables et plus expressifs que les RDD

Shark

Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, Ion Stoica:
Shark: SQL and rich analytics at scale.
SIGMOD Conference 2013: 13-24

http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-214.pdf

Présentation rapide

- Un système d'analyse de données qui associe le traitement des requêtes à des analyses complexes (ML itératif) sur de grands clusters.
- Tolérance aux pannes fine.
- Stockage en mémoire orienté colonne et replanification dynamique des requêtes intermédiaires.
- "Exécutez des requêtes SQL jusqu'à 100 fois plus rapidement qu'avec Apache Hive".
- Shark = "Spark on Hive"

SparkSQL

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia:

SparkSQL: Relational Data Processing in Spark.
SIGMOD Conference 2015: 1383-1394

https://web.eecs.umich.edu/~prabal/teaching/resources/eecs582/armbrust15sparksql.pdf

Introduction

- SparkSQL s'appuie sur l'expérience de conception de Shark.
- Avec une intégration plus étroite entre le traitement relationnel et procédural, grâce à une API DataFrame déclarative qui s'intègre au code procédural Spark.
- Inclut un optimiseur hautement extensible, **Catalyst**

Motivation

- Problèmes avec Shark :
  - ne pouvait être utilisé que pour interroger des données externes stockées dans le catalogue Hive.
  - La seule façon d'appeler Shark depuis Spark était d'écrire une chaîne de caractères contenant la requête SQL.
  - L'optimiseur Hive était trop spécifique à MapReduce et difficile à étendre.

Objectifs de Spark SQL

4 objectifs pour Spark SQL :

- Prend en charge le traitement relationnel à la fois dans les programmes Spark et les sources de données externes.
- Fournir des performances élevées en utilisant des techniques de SGBD établies.
- Prise en charge facile de nouvelles sources de données (base de données externe semi-structurée, fédération de requêtes).
- Activez l'extension avec des algorithmes d'analyse avancés tels que le traitement de graphes et le ML.

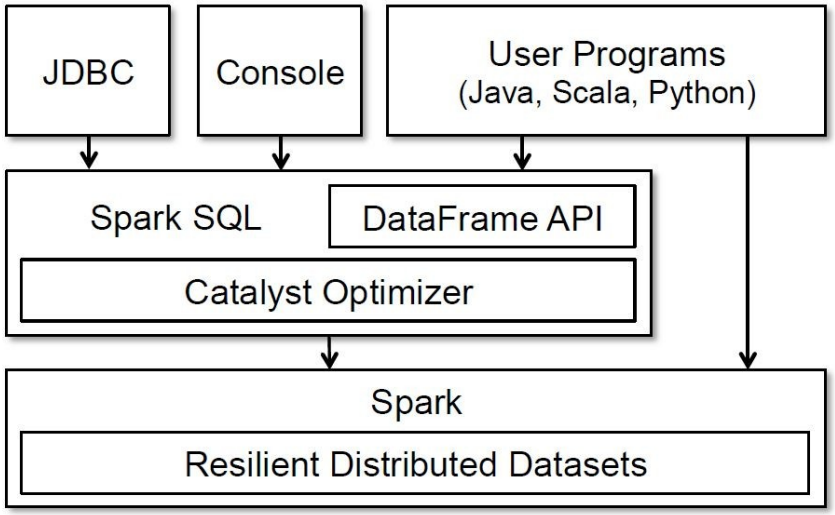


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

Caractéristiques

- Spark SQL utilise un modèle de données imbriqué basé sur Hive pour les tables et les DataFrames.
- Prend en charge tous les principaux types de données SQL et certains types complexes : structures, tableaux, cartes et unions.
- Les types de données complexes peuvent être imbriqués ensemble.
- Prend en charge les UDF.

Spark SQL - example

```
import sqlContext._
import org.apache.spark.sql.types._
import
org.apache.spark.sql.catalyst.expressions._

import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

import sqlContext.implicits._
import org.apache.spark.sql._
```

Spark SQL - example

```
// Not necessary anymore, use spark in spark-
shell
val sqlContext = new
org.apache.spark.sql.SQLContext(sc)


val schemaTriple =  StructType(
  StructField("s", LongType, false)::
  StructField("p", LongType, false)::
  StructField("o", LongType, false)::
  Nil)

val triples =
sc.textFile("/user/olivier/dbpedia/infobox_proper
ties").map(x=>x.split(" ")).map(t=>(t(0).toLong,
(t(1).toLong,t(2).toLong)))
```

Spark SQL - example

```
val rowTriples = triples.map{case(s,
(p,o))=>Row(s,p,o)}

val tripleSchemaRDD =
sqlContext.applySchema(rowTriples, schemaTriple)
tripleSchemaRDD.registerTempTable("triple")

val triplesDF = triples.toDF(“s”,”p”,”o”)
TriplesDF.createOrReplaceTempView(“triple”)


val res = spark.sql("SELECT t1.s, t3.o FROM
triple t1, triple t2, triple t3 WHERE
t1.p=1446244352 AND t2.p =1446133760 AND t3.p
=5755 AND t1.o=t2.s AND t2.o=t3.s")

res.count   // res.show
```

SQL tables and views

- 2 types de tables:
  - Managed où Spark gère à la fois les métadonnées et les données dans le système de fichiers
  - Unmanaged où Spark gère uniquement les métadonnées
- Dans les 2 cas, les métadonnées sont gérés dans le metastore de Hive

SQL tables et vues

- Les vues sont temporaires, elles disparaissent une fois l'application Spark terminée
- Les vues peuvent être
  - Global, visible sur toutes les SparkSessions d'un cluster donné
  - Session-scoped, n'est visible que pour une seule SparkSession
- In SQL

```
CREATE OR REPLACE [GLOBAL] TEMP VIEW .. AS
SELECT ..
```

- In DataFrame API:

```
val df = “SELECT ..”
df.createOrReplaceTempView(“viewName”)
df.createOrReplaceGlobalTempView(“vienName”)
```

Métadonnées

- Métadonnées accessibles via le catalogue

```
spark.catalog.listDatabases
spark.catalog.listTables
spark.catalog.listColumns(“tbaleName”)
```

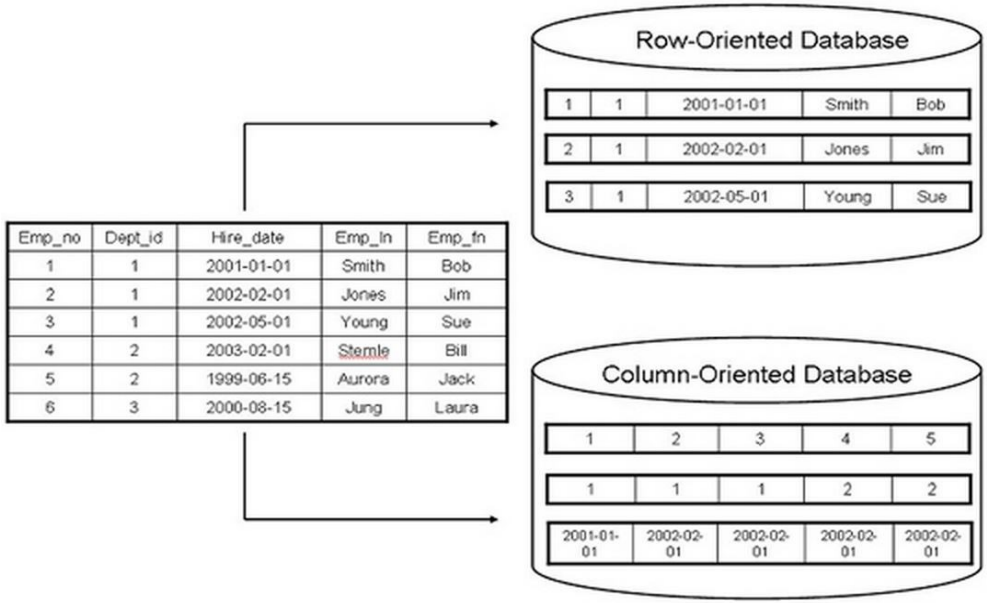
Caching de tables et de vues

- Dans Spark 3, une table peut être mise en cache paresseusement : elle ne doit être mise en cache que lorsqu'elle est utilisée pour la première fois au lieu de l’être immédiatement

```
CACHE [LAZY] TABLE <tableName>
UNCACHE TABLE <tableName>
```

In-Memory Caching

- Mettez en cache les données chaudes par :
- Columnar cache(), réduit l'empreinte mémoire en appliquant un schéma de compression tel que : l'encodage de dictionnaire et l'encodage de longueur d'exécution.



Columnar Storage

Avantages de stockage columnar storage:

1)Limitation des opérations d'IO, pas besoin de passer sur les colonnes inutiles
2)Applique des méthodes de compression adaptées
3) Meilleure performance pour le scan des tables (par rapport à un stockage en ligne), supporte des méthodes vectorielles de calcul

Catalyst Optimizer

- Spark's optimizer
- Spark 1.x : rules
- Spark 2.x : rules + cost
- Spark 3.x : rules + cost + runtime

Catalyst Optimizer

- **Two goals of Catalyst:**

1)Make it easy to add new optimization technique and features to Spark SQL.

2)Enable external developers to extend the optimizer (e.g. adding data source specific rules, support new data types).

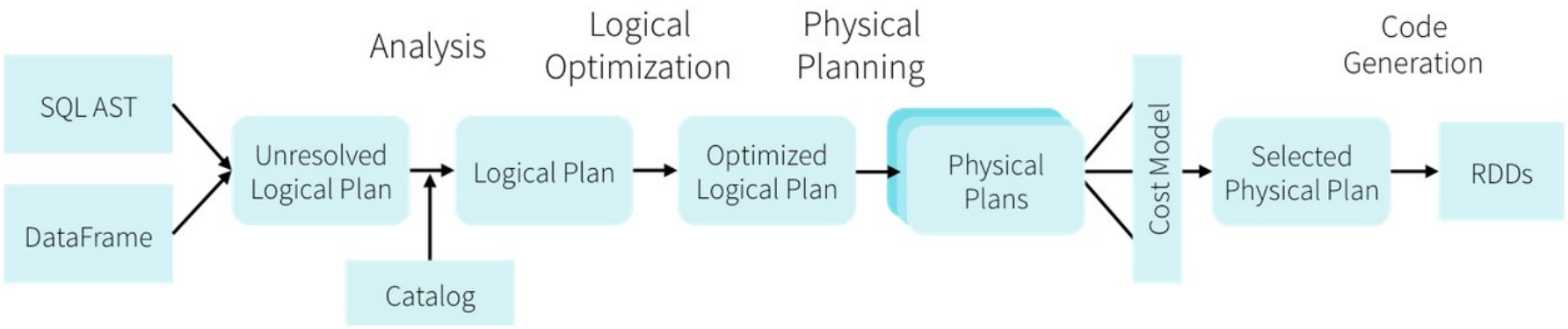
- At the core of Catalyst, it contains a general library for representing **trees** and applying **rule** to manipulate them. For example:

Cost-based optimization is performed by generating multiple plans using rules, and then computing their costs.

Catalyst

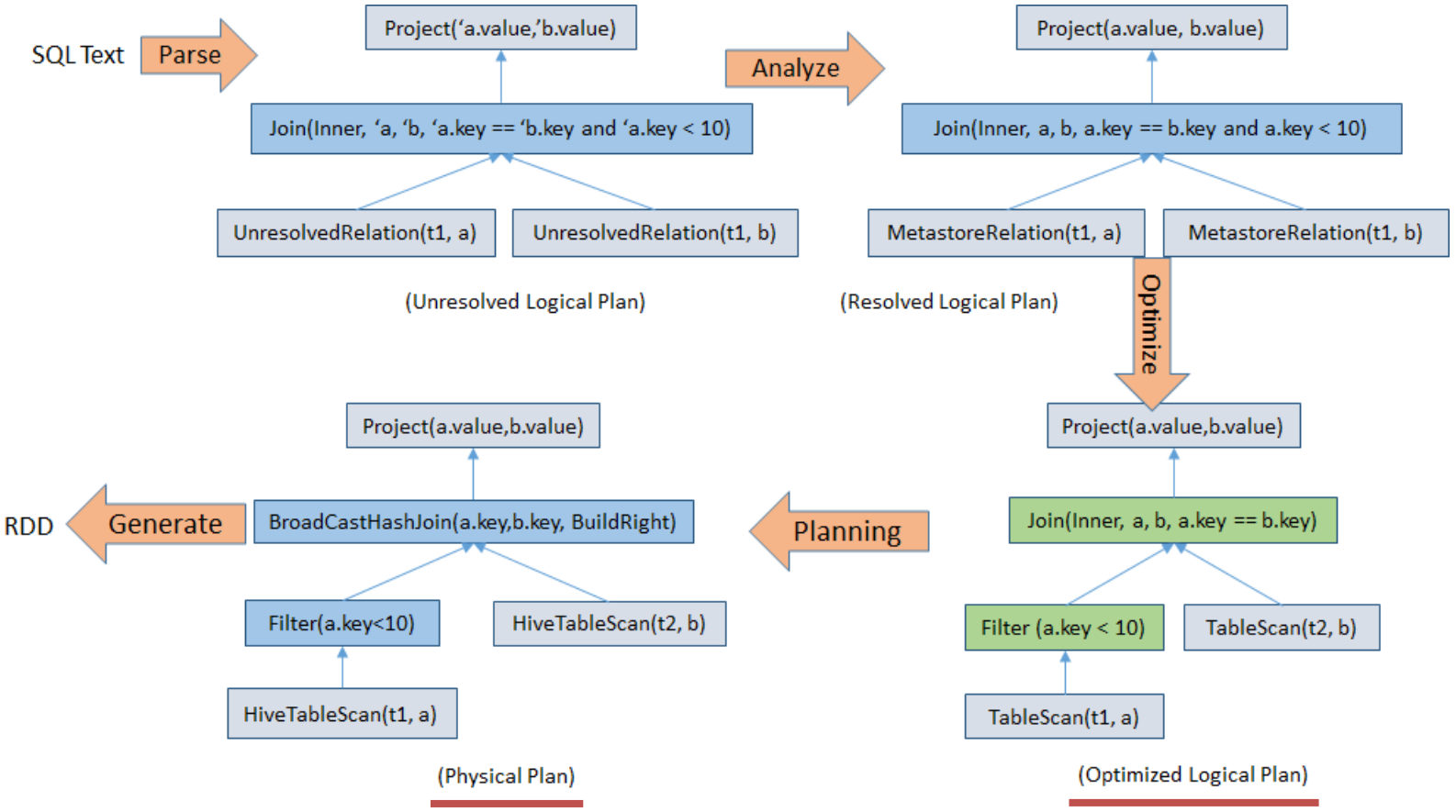
- Optimizer for SQL and DataFrames.
- Makes it easy to add data sources, (e.g., JSON), optimization rules and data types.
- Uses features of Scala (pattern-matching).

Catalyst


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

Catalyst Optimizer – Using Catalyst in Spark SQL

- Exemple:



Catalyst Optimizer – Logical Optimization

- Applique une optimisation standard basée sur des règles au plan logique, comme le pliage constant, le refoulement de prédicat, l'élagage de projection, la simplification d'expression booléenne, etc.

Catalyst Optimizer – Physical Planning

- Pour un plan logique, Spark SQL génère un ou plusieurs plans physiques et sélectionne un plan à l'aide d'un modèle de coût : **select join algo.**

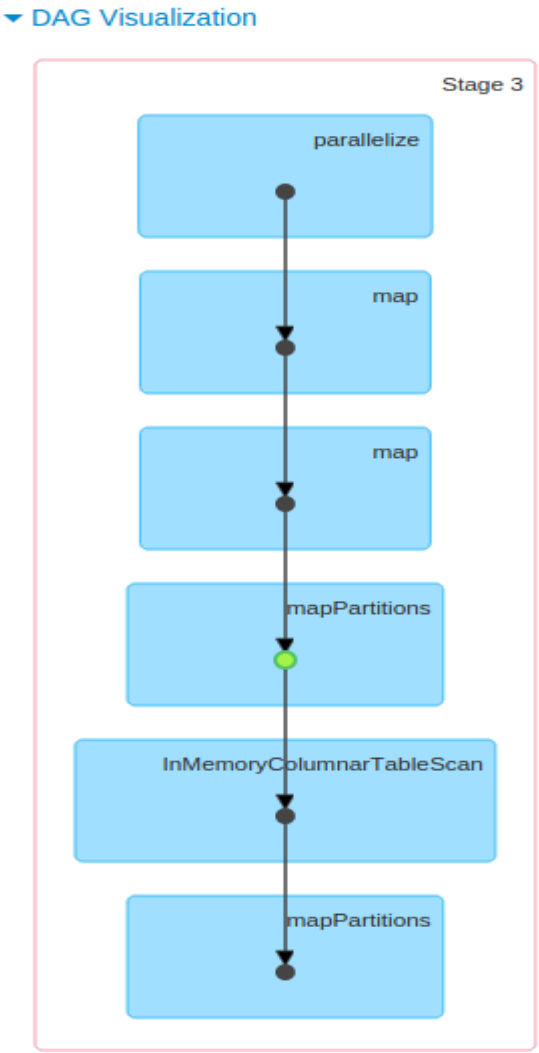
Catalyst Optimizer – Code Generation

- Phase finale de l'optimisation des requêtes, génère du byte-code Java à exécuter sur chaque machine

Visualization / Explain

- Possibilité de visualiser le plan de requête exécuté sur l'interface HTTP Spark.
- Visualization avec la méthode 'explain'.

```
scala> res.explain
== Physical Plan ==
InMemoryColumnarTableScan
[s#6L,p#7L,o#8L], (InMemoryRelation
[s#6L,p#7L,o#8L], true, 10000,
StorageLevel(true, true, false,
true, 1), (Scan
PhysicalRDD[s#6L,p#7L,o#8L]), None)
```



Offre de jointures de Spark SQL

- Broadcast hash join
  - Nécessite qu'une table soit petite
  - No shuffle, pas de tri, très rapide
- Sort-merge join
  - Robuste
  - Peut gérer toutes les tailles de table
  - Besoin de shuffle et trier les données, moins rapide dans la plupart des cas quand une table est petite
- Shuffle Hash join
  - Shuffle mais pas de tri
  - Peut gérer des tables larges mais limite si les clés de jointures ne sont pas équilibrées
- Shuffle nested loop join
  - Ne nécessite pas de clés de jointure

Conclusion

- RDD va devenir l'épine dorsale de l'abstraction de Spark
- DataFrame/DataSet deviendra l'abstraction la plus utilisée par les développeurs Spark