

Interrogation de données structurées en Spark

Master DAC – Bases de Données Large Echelle

Mohamed-Amine Baazizi

mohamed-amine.baazizi@lip6.fr

2020-2021

Plan

- API Dataframe
 - Chargement de données
 - Opérations de base
 - Opérations complexes
- Modèle de données Spark SQL

Constat RDD

- Pas de schéma
 - code peu lisible, programmation fastidieuse

Ex. Interrogation Films (Id, Title, Genres)

accès au film identifié par 2

```
spark> val films = sc.textFile().map(_.split(' ')).map(...)
spark> films.filter(x=>x._1==2)
```

1	Toy Story (1995)	Animation Children
2	Jumanji (1995)	Adventure Children
	..	

Possibilité 1 : RDD de Tuples

Constat RDD

- Pas de schéma
 - code peu lisible, programmation fastidieuse

Ex. Interrogation Films (Id, Title, Genres)

accès au film identifié par 2

```
spark> case Class Film(Id:Str, Title:Str, Genres:Str)
spark> val films = sc.textFile().map(_.split( ... )).map(...)
spark> films.filter(x=>x.MovieID==2)
```

Film (1, Toy Story (1995), Animation Children)
Film (2,Jumanji (1995),Adventure Children)
...

Possibilité 2 : RDD d'objets

Constat RDD

- Pas de schéma
 - Code peu lisible, programmation fastidieuse
 - Encapsuler dans des objets reflétant la structure
 - Performances dégradées (sérialisation d'objets, GC)
- Dans tous les cas : absence d'optimisation logique
 - Analyse statique de la requête (vérification des attributs, projection sur attributs non pertinents, ...)

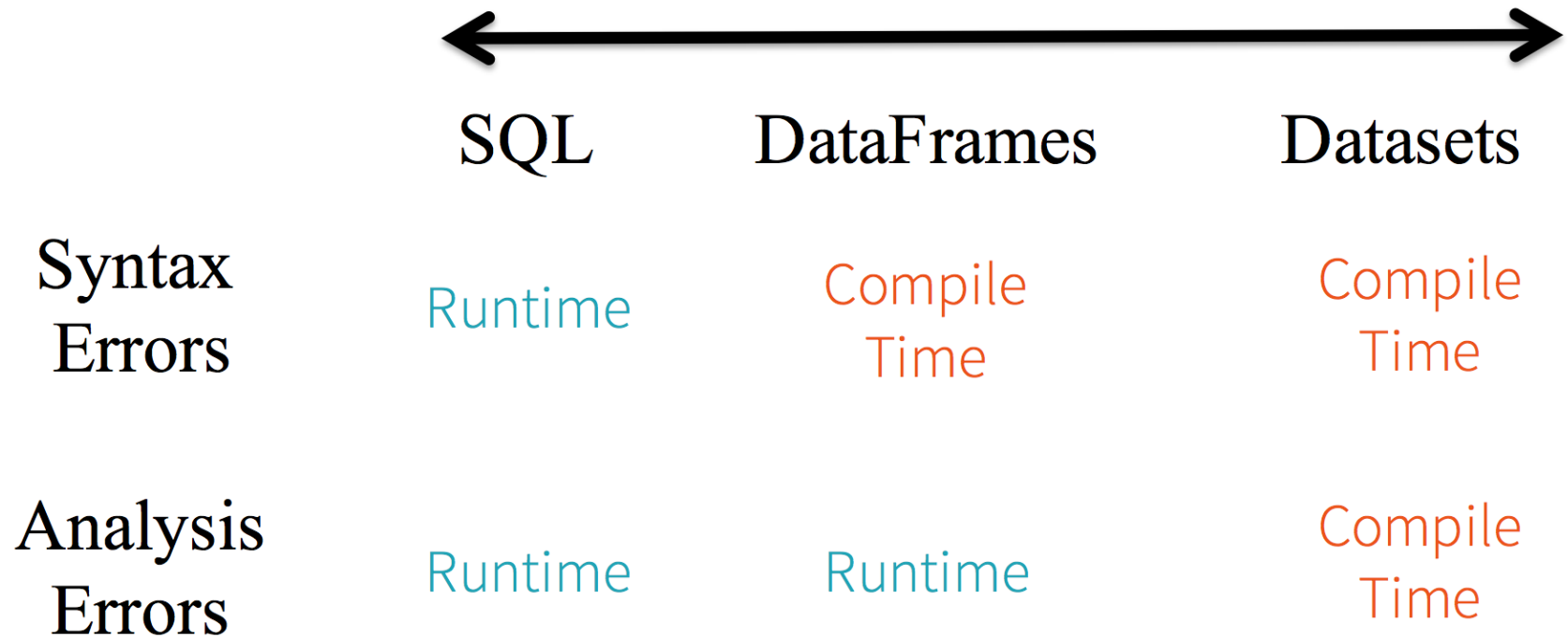
Une API déclarative pour interroger les données structurées?

- **API Dataframe**
 - Collection distribuée de tuples (row) avec un même schéma
 - Inspirée du langage R
- **API Dataset**
 - Collection distribuée d'objets conformes à un type T stockés de manière optimisée
 - Spécifique à Scala
 - Dataframe en est un cas spécifique
 - Dataframe = Dataset[Row]

Avantage des Datasets

- Typage statique
 - détecter erreurs avant exécution
 - Ex. détecter accès à un attribut absent
- Interrogation déclarative
 - Abstraction de l'organisation physique
 - Ex. pas besoin de connaître la position d'un attribut
- Optimisation logique
 - Réécriture de la requête à base de règles
 - Ex. projection des attributs inutiles, ordre des jointures, ajout de sélection, ...

Typage statique



Haut niveau d'abstraction

Films (Id, Title, Genres)

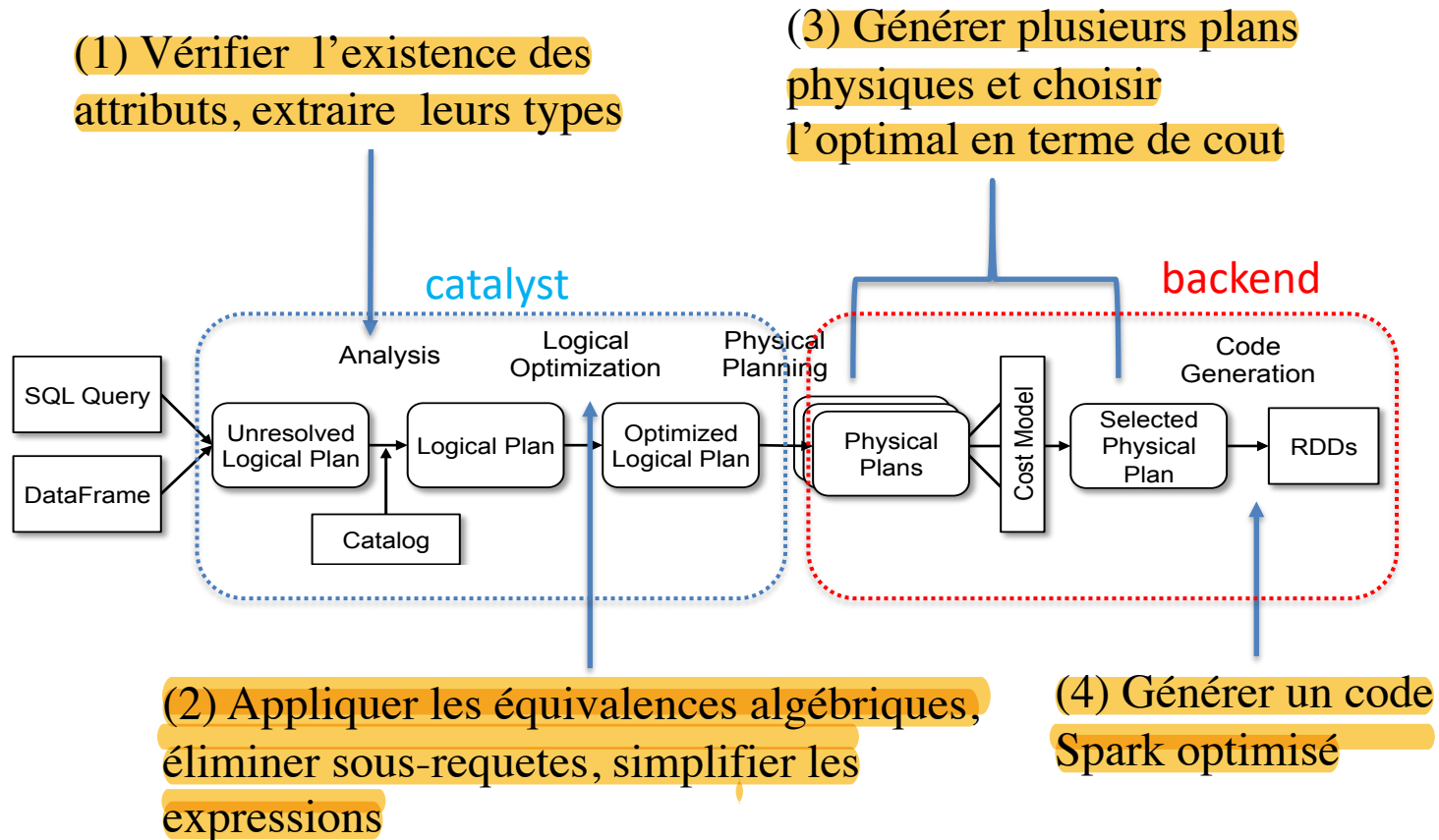
```
spark> val films = sc.textFile().map( _.split(' ..')).map(...)  
spark> films.filter(x=>x._1==2)
```

En RDD

```
spark> val films = spark.load ...  
spark> films.where("`id=2`")
```

En Dataset

Optimisation de SQL sur Spark



Création de Dataset

- Par conversion de RDD
 - Méthode toDS() invoquée depuis une Seq
 - Importer `spark.implicits._`
- Par chargement de nouvelles données
 - Formats : CSV, JSON, Parquet, texte
 - Schéma
 - Fourni par l'utilisateur, pas de 'validation' comme dans SGBD classique (récemment Delta Lakes pour les transactions)
 - Schéma automatiquement inféré
 - Peu précis (ex. détection attributs 'nullable')

Illustration

```
MovieID,Title,Genres
1,Toy Story (1995),Animation|Children...
2,Jumanji (1995),Adventure|Children....
3,Grumpier Old Men (1995),Comedy....
...
```

movies.csv

```
scala> case class Movie(MovieID:String,Title:String,Genres:String)
```

```
scala> val films = spark.read.format("csv").
```

```
    option("header",true).
```

```
    load(path+ "movies.csv").as[Movie]
```

```
films: org.apache.spark.sql.Dataset[Movie] = [MovieID: string, Title: string ... 1
more field]
```

Création de Dataset

- Cas particulier : Dataframe
 - Pas besoin de spécifier le Type (ex. classe Movie)
 - Type Row générique
 - Row = séquence de colonnes ayant un type prédéfini Spark SQL

Création d'un Dataframe : illustration

```
MovieID,Title,Genres
1,Toy Story (1995),Animation|Children...
2,Jumanji (1995),Adventure|Children....
3,Grumpier Old Men (1995),Comedy....
...
```

movies.csv

```
scala> val films = spark.read.format("csv").  
    option("header",true).  
    load(path+ "movies.csv").as[Movie]
```

```
films: org.apache.spark.sql.Dataset[Row]
```

Schéma non fourni : inférence automatique

Types Spark SQL

- Types de base
 - boolean, numeric (integer, decimal, ...), String, null, timestamp
- Tableau
 - ArrayType(type, containsNull)
- Enregistrement
 - StructType(List [StructField])
 - StructField(name, type, nullable)
- Tableau associatif
 - MapType (keyType, valueType, valueContainsNull)

Illustration

Films (Id: num, Title: text, Genres: text)

```
StructType(List(StructField('Id', int, true),  
               StructField('Title', String, true),  
               StructField('Genres', String, true)  
            )  
)
```


Opérateurs

- **Distinction entre actions et transformations**
 - Une action déclenche un traitement
 - Ex. count, show, take
 - Transformations = opérateurs enchainés
 - Ex. select, where, join
- **Distinction entre Dataset et Dataframe**
 - Pour Dataset usage du type plus poussée
 - Pour Dataframe inférence du type SQL
- **Polymorphisme**
 - Différentes versions du même opérateurs
 - Ex. drop(col) et drop(col*)

Transformations Dataframe

- Opérateurs algèbre relationnelle
 - select(col, ...)
 - where (cond)
 - join(autreDataframe)
- Opérateurs modifiant schéma
 - drop(col*)
 - withColumn(nom, def)
 - withColumnRenamed(orig, dest)
- Opérateurs de partitionnement, d'agrégation
 - groupBy(col*) -> produit un *DatasetGroupé*
 - agg(expr)

Transformations *DatasetGroupé*

- Résultat du groupBy
- Application d'une agrégation
 - Type de retour = dataframe
 - min, max, count, sum
 - Ex. `df.groupBy("year").sum`
- Création d'une table pivot
 - Type de retour préservé
 - Ex. `df.groupBy("year").pivot("course").sum("earnings")`
- Plusieurs fonctions d'agrégation

```
import org.apache.spark.sql.functions._  
df.groupBy("department").agg(max("age"), sum("expense"))
```

Fonctions Définies Utilisateur

- Exprimer des traitements non-relationnels
 - Problème classique, traité dans plupart des SGBD
 - Ex. Calculer similarité entre utilisateurs
 - Jacard(u1,u2) = nb_film_communs/nb_films_union
 - Fonction s'exécutant hors du contexte SQL
 - Pas d'optimisation automatique
 - Cout peut dégrader les performances!
 - Possibilité de rajouter règle d'optimisation Catalyseur
 - Spark ML en fait un usage intensif, optimisé
 - Exemple en TME

Dataframe par l'exemple

- Actions et fonctions de base

```
scala> films.show
```

MovieID	Title	Genres
1	Toy Story (1995)	Animation Children's Comedy
2	Jumanji (1995)	Adventure Children's Fantasy
3	Grumpier Old Men ...	Comedy Romance
4	Waiting to Exhale...	Comedy Drama
5	Father of the Bri...	Comedy

```
scala> films.printSchema
```

```
root
```

```
|-- MovieID: string (nullable = true)
```

```
|-- Title: string (nullable = true)
```

```
|-- Genres: string (nullable = true)
```

Dataframe par l'exemple

- Transformations

```
scala> films.map(x=>x.Genres.split("\\|")).show
```

```
+-----+
|          value|
+-----+
|[Animation, Child...|
|[Adventure, Child...|
|  [Comedy, Romance]|
|  [Comedy, Drama]|
```

```
scala> films.orderBy("Title").show
```

```
+-----+-----+-----+
|MovieID|      Title|      Genres|
+-----+-----+-----+
|      5|Father of the Bri...|      Comedy| | |
|     10|GoldenEye (1995)|Action|Adventure|...|
|      3|Grumpier Old Men ...|      Comedy|Romance|
|      6|Heat (1995)|Action|Crime|Thri...|
|      2|Jumanji (1995)|Adventure|Childre...|
|      7|Sabrina (1995)|      Comedy|Romance|
|      9|Sudden Death (1995)|      Action|
```

Dataframe par l'exemple

- Agrégation simple – agrégation avec groupement

```
scala> notes.agg(min("Rating"), max("Rating"), avg("Rating"))
```

```
scala> notes.describe("Rating").show //montre count, stddev en plus
```

```
scala> notes.groupBy("MovieID").agg(count("*")).sort("count(1)").show
```

Dataset par l'exemple

Sélection par critère

```
scala> films.where("MovieID=1")
```

```
scala> films.where("MovieID=1 or Title='Toy Story (1995)'")
```

Projection

```
scala> films.select("MovieID", "Genres")
```

```
scala> films("MovieID") // une colonne à la fois
```

Equi-jointure

```
scala> films.join(notes, "MovieID")
```


Données semi-structurées

- Caractéristiques

- très répandues : crawl api, data-sets publiques, ...
- flexibilité, pas de schéma préétabli (schéma a posteriori)
- imbrication sur plusieurs niveaux, structure variable
- difficiles à manipuler et à cerner

- JSON : modèle le plus répandu

- syntaxe plus simple que XML
- exprime à la fois une séquence de valeurs (arrays) et des tuples (record)

- Quelques modèles plus expressifs utilisent :

- Map (tableaux associatifs)
- Bags (arrays sans la notion d'ordre)

JSON : modèle de données, schémas

- Data model

- Valeurs atomiques : null, bool, number, string
- records : ensemble de paires (clé,valeur)
- arrays : séquence de valeurs

- Spécification du schéma

- pas de standard, propre à chaque système
- quelques similitudes et un candidat en lice (JSON-Schema) pour définir un schéma a priori
- expressivité variable : utilisation d'expression régulière, opérateur d'union, comptage, négation

```
{  
  "email" : "abc@ef",  
  "first" : "li",  
  "coord" : [{  
    "lat" : 45,  
    "long" : 12  
  }],  
  "last" : null  
}
```

un objet JSON

Manipulation de JSON dans Spark

- **Chargement**

- Traduction vers modèle Spark SQL guidée par le schéma
- Schéma fourni ou inféré automatiquement

- **Interrogation**

- Algèbre Dataset
- Notation pointée pour naviguer dans la hiérarchie
- Fonctions prédéfinie pour manipuler les arrays

Inférence du schéma et chargement

```
{
  "person" : {
    "firstname" : "Melena",
    "lastname" : "RYZIK",
    "role" : "reported",
    "rank" : 1,
    "organization" : ""
  }
}
{
  "person" : {
    "firstname" : "other",
    "lastname" : "ABCD",
    "rank" : 1,
    "organization" : "OO"
  }
}
```

Collection de 2 objets

```
testNyt: org.apache.spark.sql.DataFrame
scala> testNyt.printSchema
root
|-- person: struct (nullable = true)
|   |-- firstname: string (nullable = true)
|   |-- lastname: string (nullable = true)
|   |-- organization: string (nullable = true)
|   |-- rank: long (nullable = true)
|   |-- role: string (nullable = true)
```

*role devrait être le
seul attribut
optionnel*

```
scala> testNyt.show
```

person
[Melena, RYZIK, , 1, reported]
[other, ABCD, OO, 1,]

de type
SparkSQL Struct

```
scala> testNyt.select("person.*").show
```

firstname	lastname	organization	rank	role
Melena	RYZIK		1	reported
other	ABCD	OO	1	null

Inférence du schéma et chargement

```
{
  "first" : "al",
  "coord" : [],
  "last" : "jr"
}
{
  "first" : "al",
  "coord" : null,
  "last" : "jr"
}
{
  "email" : "abc@ef",
  "first" : "li",
  "coord" : {
    "lat" : 45,
    "long" : 12
  },
  "last" : null
}
```

Collection de 3 objets

```
scala> test.printSchema
```

```
root
```

```
|-- coord: string (nullable = true)
```

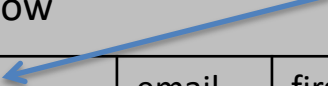
```
|-- email: string (nullable = true)
```

```
|-- first: string (nullable = true)
```

```
|-- last: string (nullable = true)
```

```
scala> test.show
```

de type
SparkSQL String



coord	email	first	last
[]	null	al	jr
null	null	al	jr
{"long":12,"lat":45}	abc@ef	li	null

*impossible de récupérer
long et lat sans parsing
préalable*

Inférence du schéma et chargement

```
{
  "person" : [
    {
      "firstname" : "Melena",
      "lastname" : "RYZIK",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    },
    {
      "firstname" : "derba",
      "lastname" : "OKYZ",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    }
  ]
}
{
  "person" : [
    {
      "firstname" : "other",
      "lastname" : "ABCD",
      "rank" : 1,
      "organization" : "OO"
    }
  ]
}
```

Collection de 2 objets


```
scala> testNyt.printSchema
```

```
root
```

```
|-- person: array (nullable = true)
|   |-- element: struct (containsNull = true)
|       |-- firstname: string (nullable = true)
|       |-- lastname: string (nullable = true)
|       |-- organization: string (nullable = true)
|       |-- rank: long (nullable = true)
|       |-- role: string (nullable = true)
```

```
scala> testNyt.show(truncate=false)
```

de type
SparkSQL Array<Struct>



person
[[Melena, RYZIK, , 1, reported], [derba, OKYZ, , 1, reported]]
[[other, ABCD, OO, 1,]]

Interrogation

```
{
  "person" : {
    "firstname" : "Melena",
    "lastname" : "RYZIK",
    "role" : "reported",
    "rank" : 1,
    "organization" : ""
  }
}
{
  "person" : {
    "firstname" : "other",
    "lastname" : "ABCD",
    "rank" : 1,
    "organization" : "OO"
  }
}
```

Collection de 2 objets

```
scala> testNyt.show
```

person
[Melena, RYZIK, , 1, reported]
[other, ABCD, OO, 1,]

```
scala> testNyt.select("person.*").show
```

firstname	lastname	organization	rank	role
Melena	RYZIK		1	reported
other	ABCD	OO	1	null

Utilisation de l'algèbre Dataset : select, where, join, ...
<https://spark.apache.org/docs/...org.apache.spark.sql.Dataset>

Interrogation

```
{
  "person" : [
    {
      "firstname" : "Melena",
      "lastname" : "RYZIK",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    },
    {
      "firstname" : "derba",
      "lastname" : "OKYZ",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    }
  ]
}
{
  "person" : [
    {
      "firstname" : "other",
      "lastname" : "ABCD",
      "rank" : 1,
      "organization" : "OO"
    }
  ]
}
```

Collection de 2 objets

```
scala> testNyt.show(truncate=false)
```

person
[[Melena, RYZIK, , 1, reported], [derba, OKYZ, , 1, reported]]
[[other, ABCD, OO, 1,]]

```
scala> testNyt.select(explode_outer($"person")).show(truncate=false)
```

col
[Melena, RYZIK, , 1, reported]
[derba, OKYZ, , 1, reported]
[other, ABCD, OO, 1,]

explode dés-imbrique le contenu d'une colonne de type `ArrayType<T>` en retournant une colonne de type `T`

Interrogation

```
{
  "person" : [
    {
      "firstname" : "Melena",
      "lastname" : "RYZIK",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    },
    {
      "firstname" : "derba",
      "lastname" : "OKYZ",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    }
  ]
}
{
  "person" : [
    {
      "firstname" : "other",
      "lastname" : "ABCD",
      "rank" : 1,
      "organization" : "OO"
    }
  ]
}
```

Collection de 2 objets

```
scala> testNyt.show(truncate=false)
```

person
[[Melena, RYZIK, , 1, reported], [derba, OKYZ, , 1, reported]]
[[other, ABCD, OO, 1,]]

```
scala> testNyt.select($"person",explode($"person")).show(truncate=false)
```

person	col
[[Melena, RYZIK, , 1, reported], [derba, OKYZ, , 1, reported]]	[Melena, RYZIK, , 1, reported]
[[Melena, RYZIK, , 1, reported], [derba, OKYZ, , 1, reported]]	[derba, OKYZ, , 1, reported]
[[other, ABCD, OO, 1,]]	[other, ABCD, OO, 1,]

Interrogation

```
{
  "person" : [
    {
      "firstname" : "Melena",
      "lastname" : "RYZIK",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    },
    {
      "firstname" : "derba",
      "lastname" : "OKYZ",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    }
  ]
}
{
  "person" : [
    {
      "firstname" : "other",
      "lastname" : "ABCD",
      "rank" : 1,
      "organization" : "OO"
    }
  ]
}
```

Collection de 2 objets

?



Nécessite l'imbrication dans le select

```
{
  "role" : "reported",
  "persons" : [
    {
      "firstname" : "Melena",
      "lastname" : "RYZIK",
      "rank" : 1,
      "organization" : ""
    },
    {
      "firstname" : "derba",
      "lastname" : "OKYZ",
      "rank" : 1,
      "organization" : ""
    }
  ]
}
```

Bilan

- Expressivité limitée du langage de schéma et du langage de requêtes
 - pas de distinction entre attributs optionnels et obligatoires
 - n'exprime pas l'union : `String + [] + {long: String, lat: String}`
 - quid de l'accès indexé aux éléments d'un Array et de l'imbrication dans le select?
- Autres pistes
 - Extensions SQL : *SQL++* de AsterixDB, *NIQL* de Couchbase, *SQL* de Apache Drill
 - Langages propriétaires : *Aggregation Pipelines* de Mongo, *JSONiq* de Zorba
 - Plusieurs connecteurs possibles avec Spark

Bilan

- Typage statique
- Interrogation déclarative
- Optimisation logique
- Supporte Spark ML
 - faciliter la création et la transformation des *features*
 - Ex. indexation attributs, création vecteurs, décider des *categorical* features, pipeline optimisé ML