MU5IN852

Bases de Données
Large Echelle

Cours 1
SQL à large échelle

septembre 2024

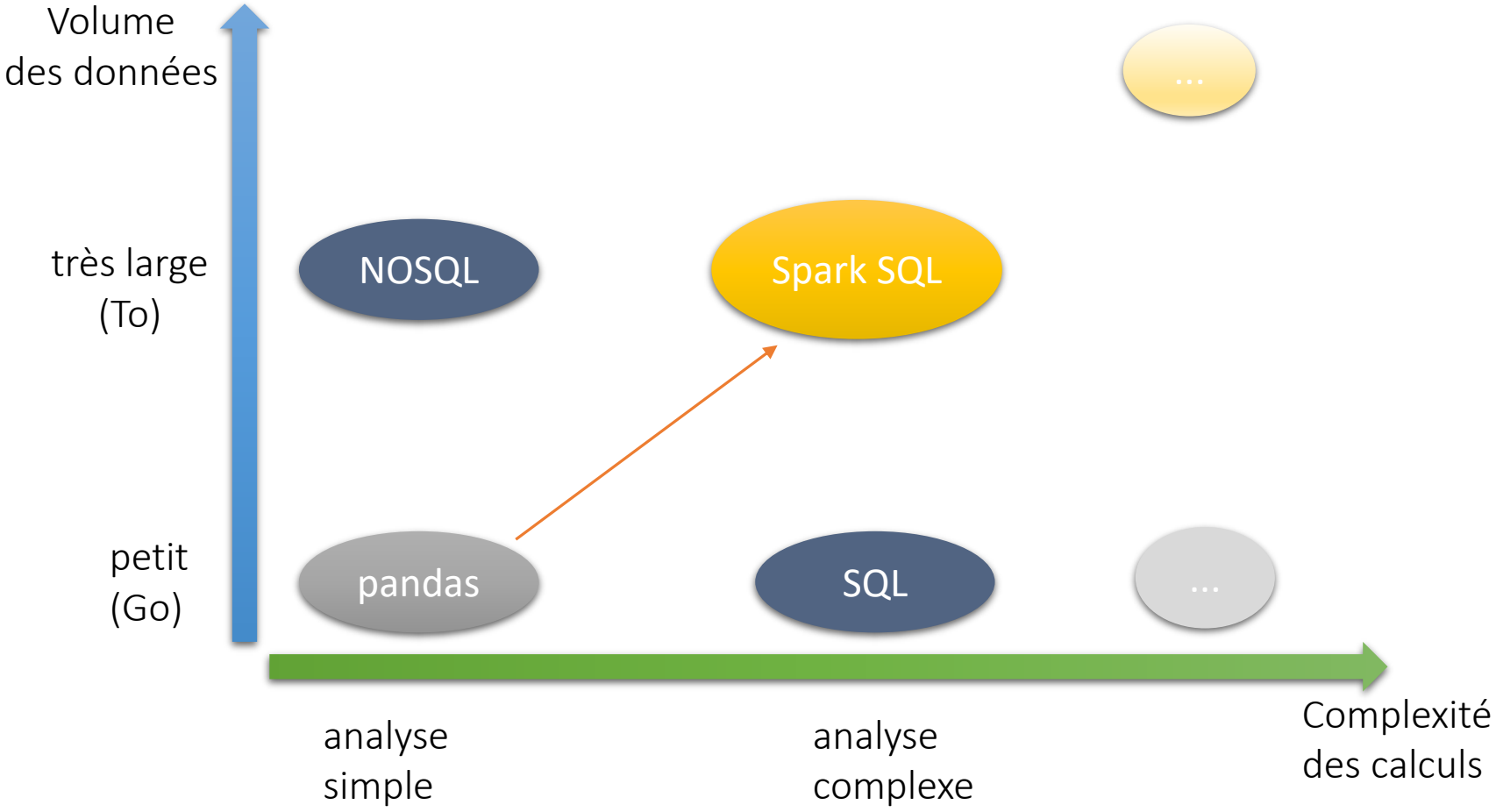
hubert.naacke@lip6.fr

séances BDLE : 3 blocs

- SQL pour analyser des big data
  - Préparation, structuration
  - Données multidimensionnelles, agrégation, fonctions de fenêtres
- Manipulation et préparation de big data
  - JSON : manipulation avec pyspark
  - Qualité : mise à jour de tables, contraintes, qualité des données
  - ML et big data
- Système big data et cluster computing
  - Data streaming
  - Stockage compact
  - Exécution parallèle et distribuée, optimisation de requêtes
- Big graphs
  - Bulk synchronous parallel processing
  - Modèle et langage de requêtes pour des grands graphes
  - Graphes de connaissance, représentation par des embeddings

Commun : savoir faire sur Spark et la gestion de big data

Défis


Volume
des données
très large
(To)
petit
(Go)
NOSQL
Spark SQL
...
pandas
SQL
...
analyse
simple
analyse
complexe
Complexité
des calculs

Motivations et Objectifs

- Volume : analyser un fichier quelle que soit sa taille
  - S'affranchir de la limite imposée par la capacité d'une machine
  - manipuler efficacement un fichier qui ne tient pas dans la mémoire d'une machine
- Diversité des données : analyser des fichiers «bruts»
  - Données peu structurées, incomplètes
- Poser une requête **déclarative** sur plusieurs fichiers
  - Moteur SQL pouvant lire directement et efficacement des fichiers
- Analyse avancée
  - Disposer de nombreuses fonctions d'analyse
- Langage de requête extensible
  - SQL intégrant des fonctions définies par l'utilisateur : UDF

Solution SQL pour le big data

- La plateforme Spark
  - Manipulation efficace de fichiers volumineux
  - Architecture big data orientée service
    - Système offrant un service de requêtes
    - Système distribué : capacité non limitée, haute disponibilité
    - Fonctionne de manière identique avec 1 ou N machines
      - Conception d'un scénario d'analyse sur un 1 machine
      - Mise en production : déploiement sur un cluster
- SQL est un langage standard de manipulation
  - Chaines de traitement extensibles et réutilisables
  - Migration facilité entre les plateformes big data
    - exple migration de/vers google BigQuery, Flink, Hive

SQL via l'interface pyspark

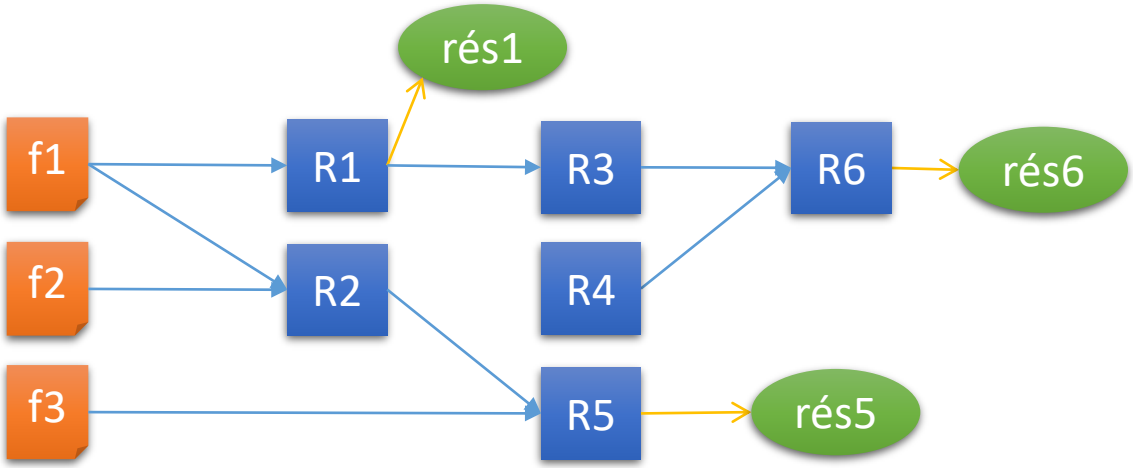
- pyspark : interface pour connecter une appli python au système spark
  - API pour invoquer de requêtes SQL depuis une appli python
- Avantages
  - Langage python familier
  - Intégration facilitée avec les lib python fréquemment utilisées
- Inconvénient
  - Langage différent de scala utilisé nativement dans Spark
    - Inconvénient mineur quand python invoque des opérations natives en scala

Chaine de traitements SQL

Traitement complexe composé de **plusieurs** requêtes SQL

Légende :

Chaine = graphe acyclique de requêtes



Définition d'une chaine de traitements dans Spark

Définir l'accès aux fichiers

```
f1= spark.read( ... )
f1.createOrReplaceTempView("f1")
```

Définir les requêtes
basées sur des fichiers
ou des requêtes

```
R1 =  spark.sql("select *
                 from f1
                 where … ")
```

Visualiser le résultat d'une requête

```
R1.show(… )
```

Manipuler des données **typées**

Définition du schéma des données

Fichier f1

- Approche appelée « Schema on read »
- Un fichier à lire est représenté par une table relationnelle
  - Une ligne correspond à un nuplet
    - Un champ correspond à un attribut typé
- Syntaxe
  - *table1* = spark.read.format(…).load(nom de fichier, schéma)
  - Cette opération définit la façon de lire les données mais ne lit **pas** les données
  - Plusieurs formats supportés
    - csv : nécessite de spécifier le schéma des données à lire
    - json : spécifier le schéma, sinon il sera inféré
    - orc, parquet : schéma auto-défini

Types SQL

- Type simple
  - Numérique : Boolean, Short, Int, Long, Float, Double
  - Chaîne de caractères : String
  - Binaire

Type	Type SQL	Type pyspark
Nombre entier	SHORT, INT, LONG	ShortType() IntegerType() LongType()
Nombre réel	FLOAT, DOUBLE	FloatType() DoubleType()
Boolean	BOOLEAN	BooleanType()
Chaine	STRING	StringType()

Types temporels : définition

- Représenter une date
  - DATE : année mois jour
  - TIMESTAMP plus *fin* que le type DATE
    - année mois jour heure minute seconde
- Représenter une durée : type INTERVAL
  - Représente une plage de n unités de temps consécutives.
  - Il existe deux échelles d'unités de temps
    - year month
      - INTERVAL '4-2' YEAR TO MONTH : durée vaut 4 ans et 2 mois
    - day hour minute second
      - INTERVAL '3' DAY : durée = 3 jours
      - INTERVAL '40:10:59' HOUR TO SECOND :
        - durée vaut 40 heures + 10 minutes + 59 seconds

Types temporels : opérations

- Création depuis une chaîne
  - to_date('2024/10/25', 'y/M/d') as d
  - ou strptime(chaine, format)
  - format = *pattern* défini par des symboles appellés *modifiers*
- Conversion
  - from_unixtime(nombre)
- Addition
  - date_add(date, intervalle)
- Soustraction
  - date_diff(date1, date2)
- Extraction
  - extract(day FROM DATE '2024-02-25');
  - partie à extraire: day, month, year, …

Types complexes

- Définir des types « imbriqués »
  - array : l'attribut est une liste de valeurs
  - tuple : l'attribut est un tuple.
  - map : l'attribut est un *dictionnaire*
    - associe une clé typée à une valeur typée

Type	Type SQL	Type pyspark
Tableau	ARRAY< type_elt >	ArrayType(….)
Tuple	STRUCT<champ$_1$ type$_1$, …., champ$_n$ type$_n$ >	StructType([StructField(… , …), …)]
Dictionnaire	MAP< type_clé, type_valeur>	MapType<…, …>

Définir le schéma d'un fichier

- Syntaxe simplifiée pour le format csv
  - Schéma composé uniquement de types simples
  - "nom_1 $type_1$, …, nom_n $type_n$"
  - Exemple : schema = "photoID Long, userID String"
- Syntaxe générale
  - Le schéma peut contenir des types complexes
  - Exemple de syntaxe SQL
    - schema = "STRUCT< photoID Long, userID String>"
  - Exemple de syntaxe python
    - schema = StructType([StructField("photoID" , LongType()), StructField("userID" , StringType())])

Catalogue des types SQL

- SQL est standard
  - avec quelques « nuances » entre les implémentations
- Spark SQL
  - Types
    - spark.apache.org/docs/latest/sql-ref-datatypes.html
  - Format des dates:
    - spark.apache.org/docs/latest/sql-ref-datetime-pattern.html
- DuckDB SQL
  - Types
    - duckdb.org/docs/sql/data_types/overview
  - Format des dates
    - duckdb.org/docs/sql/functions/dateformat

Requêtes

Graphe de requêtes SQL

- Accès SQL aux fichiers
  - Rendre visible les tables associées aux fichiers dans l'environnement SQL

```
table1.createOrReplaceTempView("Table1")
```

- Requête SQL + nommage de la requête en SQL

```
CREATE OR REPLACE TEMP VIEW R1 AS
        select …
        from Table1 … where … ;
```

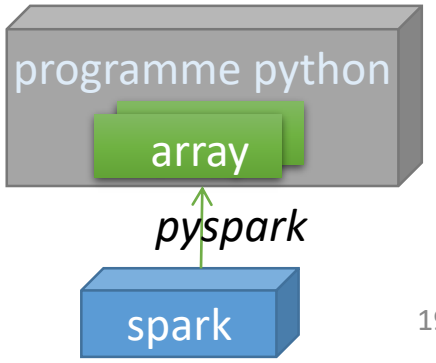
OU

- Requête SQL + nommage de la requête en python

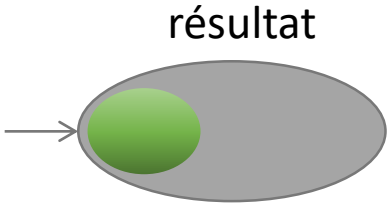
```
R1 = spark.sql(""" select …
                   from Table1 …. where … """)
R1.createOrReplaceTempView("R1")
```

Spark SQL : exécution des requêtes

- Séparation entre la définition et l'exécution d'une requête
  - La définition d'une requête SQL ne déclenche aucun traitement
- Exécution dans Spark
  - Quand ? Exécution différée à la demande dite *lazy*
    - Invoquer une instruction pour produire (une partie) du résultat de la requête
  - Qui ? Exécution par le moteur de requêtes de Spark
    - La requête n'est pas traitée directement dans l'environnement python
  - Quel résultat ? Cela crée un tableau python
- Notion de service de traitement de requêtes



Exécution partielle/complète d'une requête



show : Résultat **tronqué** d'une requête

résultat_partiel = requête.**show**(**n**)

Exécution partielle de la requête

Seul un tableau de **n** tuples est alloué

collect : Résultat **complet** d'une requête

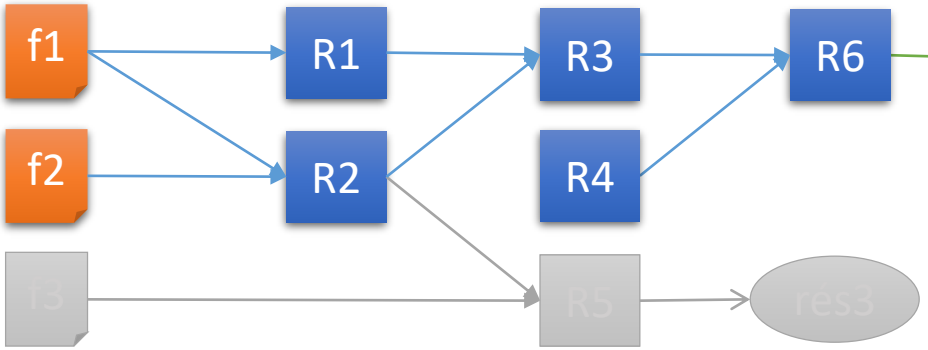
résultat_complet = requête.**collect**()

Exécution complète de la requête

Un tableau contenant tout le résultat est alloué

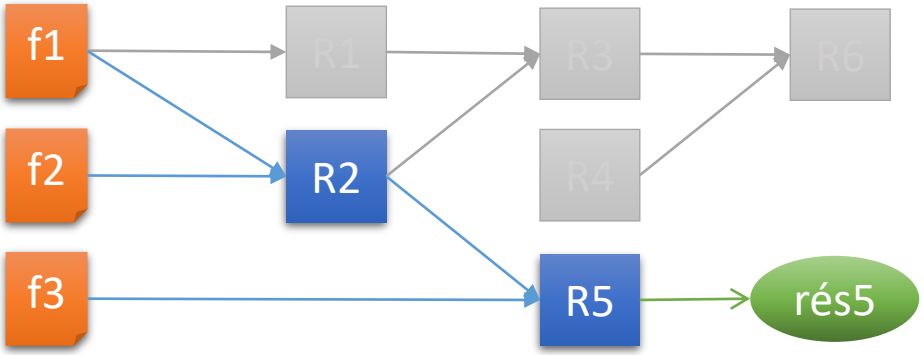
Possible seulement si le résultat est « petit »

Exécution d'une chaine de requêtes



Exécuter la requête R6

Tous les *prédécesseurs* sont exécutés



Exécuter la requête R5

Fonctions SQL

Lire la documentation des fonctions SQL

- spark : spark.apache.org/docs/latest/api/**sql**

ou

- duckDB: duckdb.org/docs/sql/**functions**/overview

Expressions CASE et CAST

- Syntaxe SQL, équivaut à un appel de fonction
- Exprimer une alternative
  - case when *condition* then *valeur1* else *valeur2* end as nom

  Équivalent à la fonction
  - if(condition, valeur1, valeur2) as nom
- Convertion de type
  - Cast(valeur as type)
  - Equivalent à des fonctions to_int(), to_…

Categories of SQL functions

- Numeric functions
- Pattern Matching, Regular Expressions, Text
- Manipulate Date, Time, Timestamp, Interval
  - formating date
- Manipulate Array, List, Map, Struct
  - Lambda Functions
- Aggregate Functions
- Window Functions

Manipuler des chaines de caractères (1/2)

- Découper une chaine
  - split(chaine, séparateur) retourne un tableau
  - Le séparateur peut être une expression régulière regex
    - Exple de séparateur : "[^a-z0-9]+" ou \\W+
- Extraction de motif
  - regexp_extract(,)
  - exple : extraire un nombre entier placé entre parenthèses dans une chaine
    - cast(regexp_extract(att1, '\\((\\d+)\\)', 1) as int) as durée
      - caractères d'échappement pour matcher une parenthèse: \\(\\)
    - regexp_extract(value, '\\"(.+)\\"(\\(([^\\)]*)\\))*', 1) as attr1
      - Le 1er groupe est le 1er terme entre guillemets
    - regexp_extract(value, '\\"(.+)\\"(\\(([^\\)]*)\\))*', 3) as attr3
      - Le 3ème groupe est le terme entre parenthèses contenant tout sauf une parenthèse
  - avec remplacement
    - regexp_replace(chaine, regex, rempl) ou replace(, ,)

Manipuler des chaines de caractères (2/2)

- Changement de casse : lower( ), upper( )
- Nettoyer les espaces : trim(), rtrim(), ltrim()
- Concaténation
  - concat(a1, …), avec séparateur: concat_ws(séparateur, chaine1, …)
- Prefixe, suffixe : left(,), right(,)
- Conversion de type
  - cast(attribut as int)

Manipuler des attributs complexes

- But
  - Gérer des données semi-structurées
  - Attributs de type list, set, map ou struct
- Définir un attribut complexe
  - Imbrication par agrégation : **group by**
    - collect_list(attr) : génère un tableau contenant les valeurs d'attr pour chaque groupe.
    - collect_list(**(**attr1, attr2**)**) : génère un tableau de tuples
    - collect_set(….) : l'attribut est un ensemble
      - Les valeurs sont uniques
- L'ordre des éléments est quelconque

Créer un attribut complexe: collect_list ou collect _set

- Exemple

  Select couleur, collect_list(nom) as noms

  From T

  Group by couleur

T

Couleur	Nom
vert	pomme
vert	avocat
rouge	raisin
rouge	choux

Couleur	Noms
vert	pomme,avocat
rouge	raisin,choux

Manipuler un tableau

- Accès : element_at(tab, position), find_in_set( )
- Sélection :
  - slice(t, début, longueur)
  - filter(t, condition)
- opérations ensemblistes
  - array_intersect, array_union, array_zip, …
- tri : array_sort
- Si t est un tableau de tableaux : union des tableaux
  - Le résultat est un tableau de valeurs ayant un type simple
  - flatten(t)
- Exprimer une condition à partir d'un tableau: quantification
  - every(tableau, x -> x = 2)
  - exists(….)

Transformer un tableau: transform

```
SELECT couleur, transform(noms, n -> substring(n,1,3)) as prefixes
FROM T
```

Couleur	Noms
vert	pomme, avocat
rouge	raisin, choux

Couleur	Prefixes
vert	pom, avo
rouge	rai, cho

Tableau: désimbrication

- Désimbriquer un attribut complexe
  - explode(tableau)
  - génère un tuple pour chaque valeur de l'attribut tableau
- Exemple
  - Select couleur, explode(noms) as nom
  - From T

Couleur	Noms
vert	pomme,avocat
rouge	raisin,choux

Couleur	Nom
vert	pomme
vert	avocat
rouge	raisin
rouge	choux

User Defined Functions (UDF)

UDF : définition et déclaration

But: Appliquer une fonction sur chaque tuple d'une table

- Définir une fonction python

  def maFonction ($param_1$, ..., $param_n$) :

  result = ... exemple: result = (min(param1, param2), max(param1, param2))

  return result

  - $param_1$, ..., $param_n$: les attributs du tuple sur lequel la fonction est invoquée
- Définir le type du résultat de la fonction
  - Voir les types slide 14
  - Exple : type = StructType([StructField("mesure1" , FloatType()), StructField("mesure2" , FloatType())])
- Déclarer la fonction pour permettre son invocation
  - dans une requête SQL
    - spark.udf.register("maFonction", maFonction, type)
  - Ou dans une requête pyspark
    - ma_fonction_udf = udf(maFonction, type)

UDF : invocation

- Exple: table T(a, b), calculer un attribut c = f(b)
- Invoquer la fonction en SQL
  - t.selectExpr("a", "maFonction(b) as c")

  SELECT a, maFonction(b) as c
  FROM T

- Ou en syntaxe pyspark
  - T.select(col(a), maFonction(col(b)).alias(c))

- UDF exécutée dans l'environnement SQL
  - doit être capable d'exécuter une fonction écrire en python

Matérialisation des résultats

Quel résultat matérialiser ?

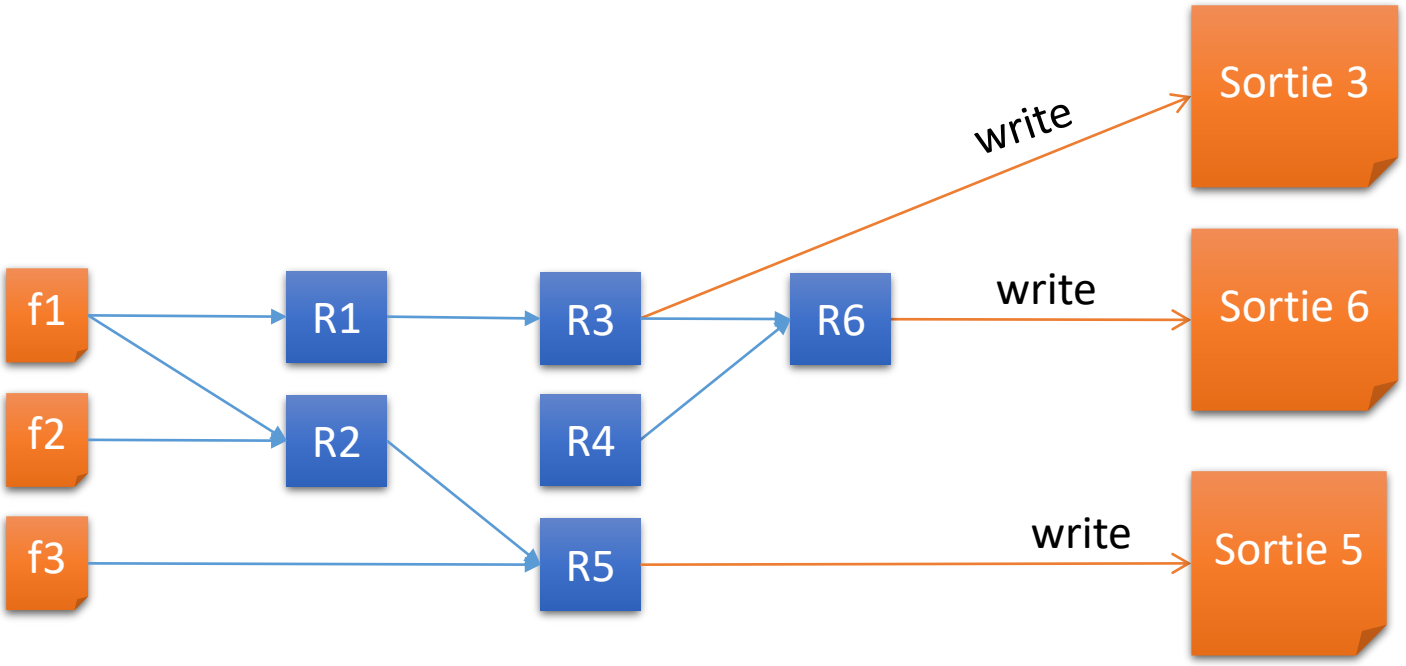
Quand le matérialiser ?

Matérialiser les résultats d'une chaine de requêtes

Rendre persistent certains résultats d'une chaine SQL

- Stocker le résultat d'une requête dans un fichier

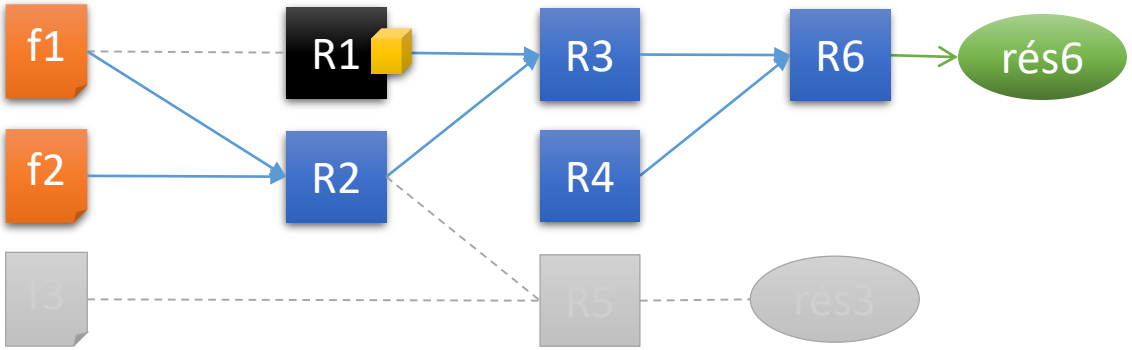
  table.write.option(…).format(….).save(…)



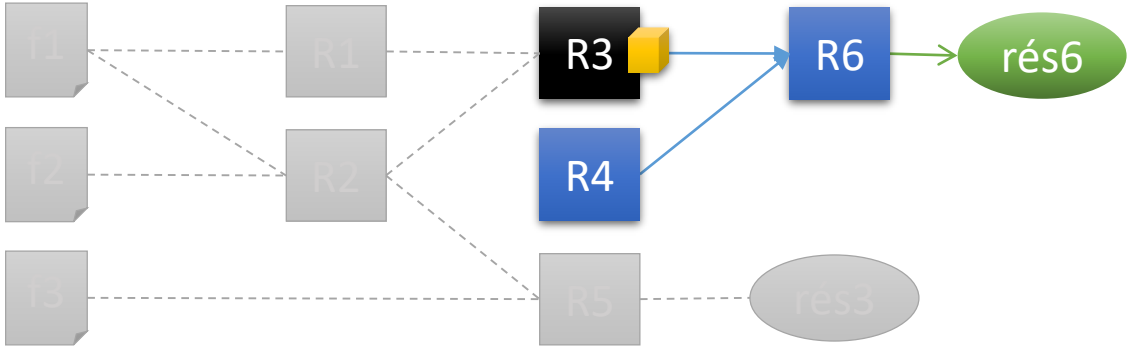
Matérialiser des résultats intermédiaires

- Objectif de performance
  - accès plus rapide au résultat d'une requête sans exécuter les requêtes dont elle dépend
    - Ne pas exécuter les prédecesseurs d'une requête dans le graphe
- Méthode
  - Matérialiser en mémoire ou stocker sur disque le résultat d'une requête intermédiaire
- Syntaxe
  - SQL : `CACHE TABLE Table1;`
  - Python : `table1.cache()` ou `table1.persist()`
- L'instruction *cache* ne déclenche **pas** l'exécution d'une requête

Exécution avec matérialisation des résultats intermédiaires



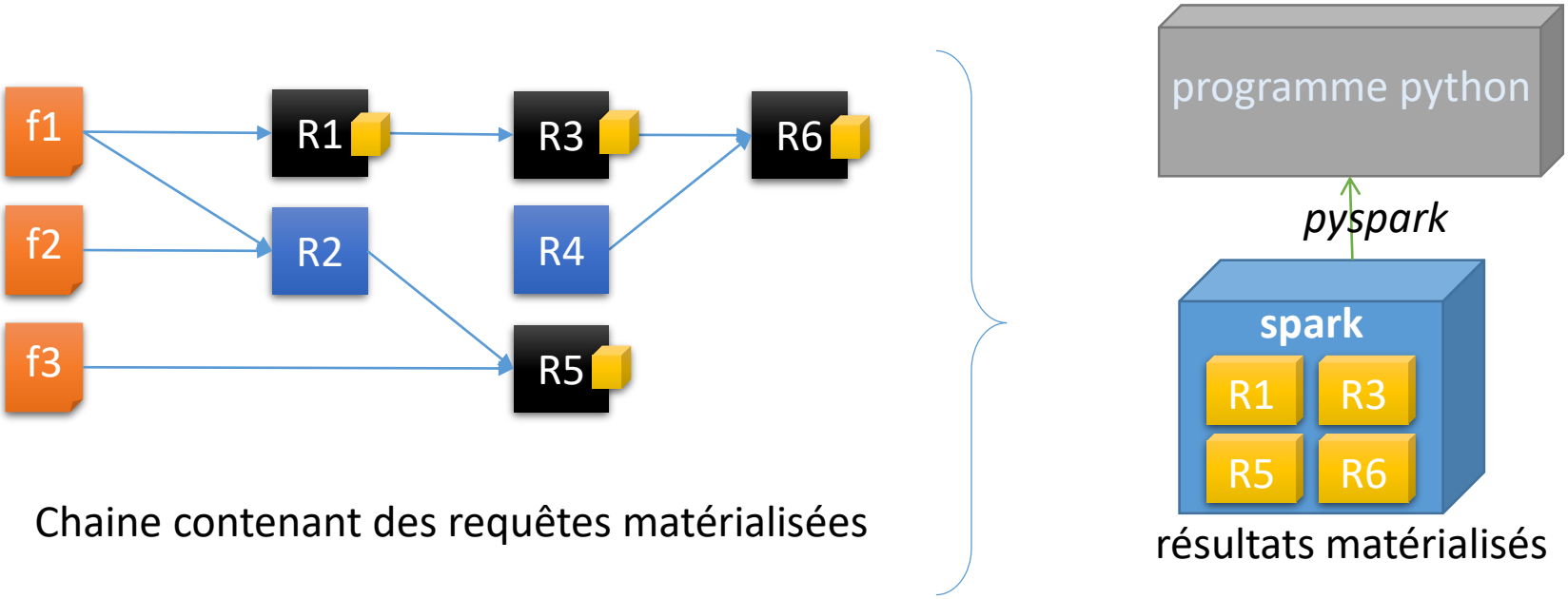
Exécuter la requête R6 avec R1 matérialisée



Exécuter la requête R6 avec R3 matérialisée

Ignorer les *prédécesseurs* d'une requête matérialisée = enlever les arcs entrants

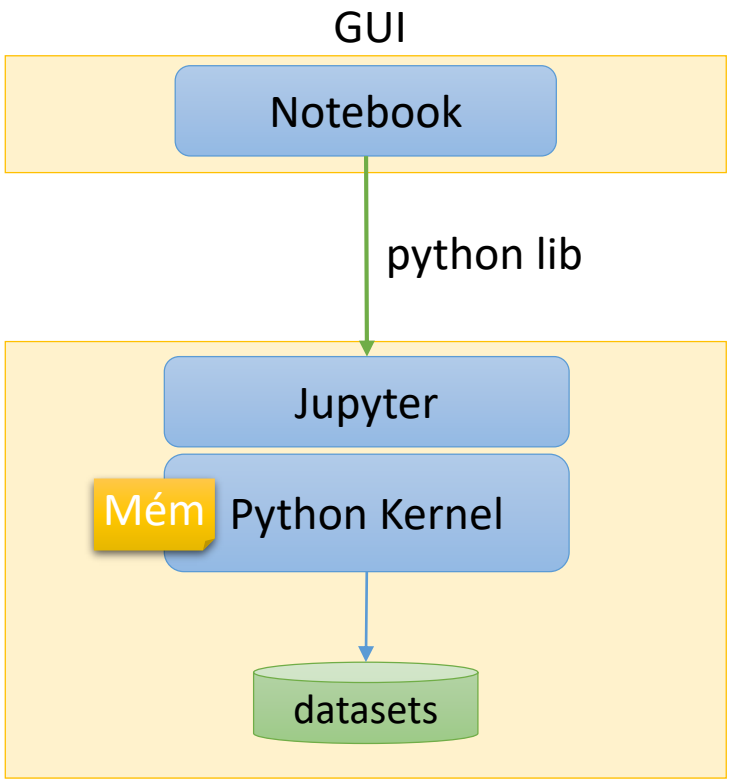
Matérialisation des résultats intermédiaires dans Spark



Environnement de TP

Analyse de données en python

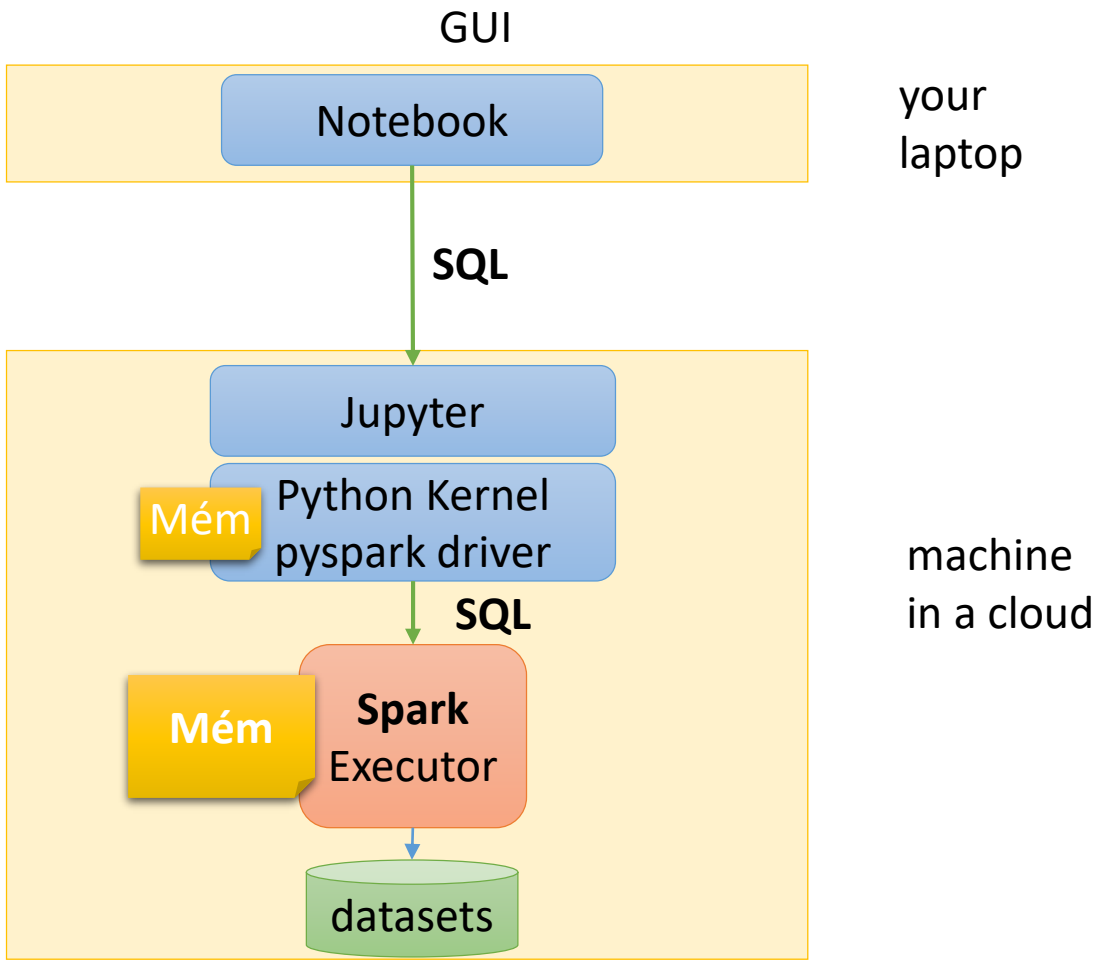
- Architecture d'un notebook colab



your
laptop

machine
in a cloud

Analyse de données avec Spark



Récap des fonctions pyspark utiles pour le 1er TP

- Lire un fichier
  - user_visits = spark.read.option("header", "True").option("delimiter", ";").format("csv").load(fichier.csv)
- Matérialiser le résultat d'une requête :
  - *req*.persist() ou req.cache()
- Afficher les n premiers tuples
  - *req*.show(n, False)
- Exécuter une requête en entier
  - req.collect() ou req.toPandas()
- Schéma : structure d'une table
  - *req*.printSchema()
- Décrire les valeurs des attributs
  - *req*.describe("att1", "att2", ...).show()
  - Calcule pour chaque attribut : min, max, avg, sdtdev, count
- Ecrire le résultat d'une requête dans un fichier
  - req.write.format('csv').mode('overwrite').option("header","true").save(fichier)

TP1

- Objectif
  - Définir des chaines de traitement complexes en SQL
- Outil
  - Notebook colab
  - Cellule SQL : %%sql
    - Peut contenir plusieurs expressions SQL séparées par **;**
    - Exécuter une cellule a pour effet de
      - Définir les expressions
      - Exécuter la dernière expression de la cellule
      - Afficher les 100 premiers tuples dans un tableau pandas
    - Bonne pratique :
      - Cellule %%sql contenant 2 expressions : définition et requête
        - Create or replace temp view table1 as select … from … ;
        - Select * from table1

Biblio

- **Spark SQL : Relational Data Processing in Spark**
  - **SIGMOD 2015**
  - Databricks, MIT CSAIL, UC Berkeley AmpLab

    https://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

    https://dl.acm.org/doi/pdf/10.1145/2723372.2742797