ESIEE
PARIS

LAB + PROJET

Sara AISSAOUI & Yannick PRAT

Matière : DATA ENGINEERING

LAB PRATICE

DE1 – Lab 1

Objectifs :

L'objectif principal du Lab 1 était de fournir une première expérience pratique sur la lecture des plans d'exécution Spark. L'utilité principale de cette pratique est d'établir l'habitude d'analyser le coût réel des opérations (I/O et Shuffle) pour optimiser les pipelines de données.

Le laboratoire a comparé trois aspects fondamentaux :

1. **RDD vs DataFrame** : Comparaison de la performance du Shuffle pour la même tâche (Top-N Tokens).
2. **Projection Efficace** : Démonstration de l'optimisation I/O (lecture de colonnes) fournie par l'optimiseur Catalyst des DataFrames.

Partie 1 : RDD (API Bas Niveau)

Cette section a exécuté le calcul du Top-N des tokens en utilisant l'API RDD. L'Action déclencheuse était le .take(10).

```
# RDD pipeline: tokenize 'text' column and count tokens
rdd = df.select("text").rdd.flatMap(lambda row: (row[0] or "").lower().split())
pair = rdd.map(lambda t: (t, 1))
counts = pair.reduceByKey(lambda a,b: a+b)
top_rdd = counts.sortBy(lambda kv: (-kv[1], kv[0])).take(10)
top_rdd
```

Justification du Job et des Métriques

L'exécution RDD a été identifiée dans le Job qui contenait le Stage 25 (reduceByKey) et les Stages suivants (26, 28, 30) pour le tri (sortBy). Le Stage 25 est le plus important car il réalise le brassage des données.

- files_read (2) & input_size_bytes (184 729 octets) : Ces valeurs sont tirées de la colonne Input Size du Stage 25 (180.4 KiB. Ce Stage est le premier à accéder aux données, représentant le coût I/O total de la lecture des 2 fichiers CSV sources.
- shuffle_write_bytes (762 octets) : Cette valeur provient de la colonne Output Size du Stage 25 : 762.0 B. Elle représente le coût d'écriture des paires (token, count) sur le disque des Executors pour préparer le brassage des données (shuffle).
- shuffle_read_bytes (762 octets) : Cette valeur provient des colonnes Shuffle Read Size des Stages 26, 28, 30 : 762.0 B. Ces Stages lisent les données écrites par le Stage 25 pour effectuer le tri final. La symétrie (Write = Read) confirme le coût total du brassage.

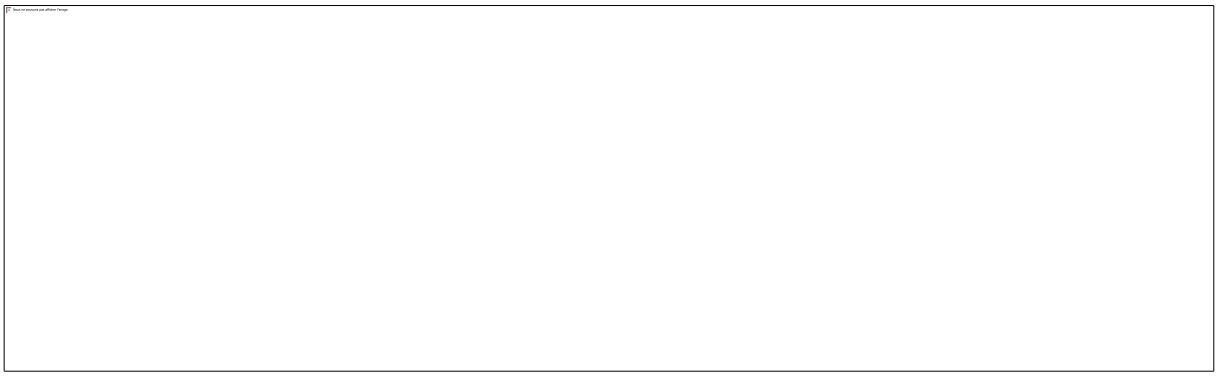

Show 20 entries

Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Shuffle Write Size / Records
driver		10.255.255.254:45209	0.9 s	2	0	0	2	false	180.4 KiB / 2	762 B / 4

Partie 2 : DataFrame (API Haut Niveau)

Cette section a exécuté la même tâche Top-N en utilisant l'API DataFrame, déclenchée par l'Action .write.csv().



Justification du Job et des Métriques

▾ Aggregated Metrics by Executor

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Shuffle Write Size / Records
driver		10.255.255.254:45209	0.2 s	2	0	0	2	false	180.4 KiB / 2	2.9 KiB / 40

Le Job choisi pour l'analyse était le Job 25 (ou similaire) qui présentait le même Input Size et qui était le Job de calcul principal avant l'écriture.

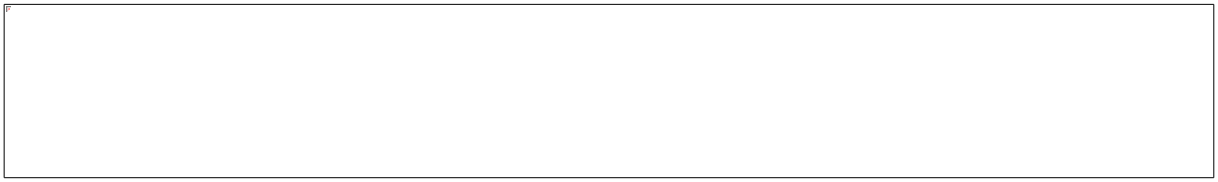
- input_size_bytes (184 729 octets) : Identique au RDD. Le Job lit la même quantité de données brutes des fichiers.
- shuffle_write_bytes et shuffle_read_bytes (2 970 octets) : Cette valeur 2.9 KiB a été tirée de la colonne Shuffle Write Size du Job 25.

On observe que Shuffle du DataFrame 2 970 B est plus lourd que celui du RDD 762 B. Cela est souvent dû à la surcharge des DataFrames (sérialisation des schémas, types) lors du brassage, rendant l'échange de données plus volumineux.

Partie 3 : Expérience de Projection (Optimisation I/O)

Cette section compare l'impact sur l'I/O (lecture disque) entre la sélection de toutes les colonnes (select("*")) et la projection minimale (select("category", "value")). L'Action déclencheuse était le .count() pour les deux Jobs.

```
# Case A: select all columns then aggregate on 'category'
all_cols = df.select("*").groupBy("category").agg(F.sum("value").alias("sum_value"))
all_cols.explain("formatted")
_ = all_cols.count()  # trigger
```



Justification des Jobs

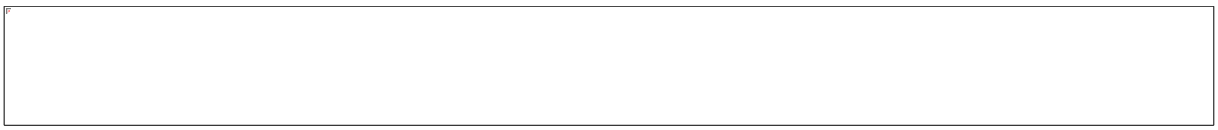
- Cas A (projection_all_cols) : Représenté par le Job 27 (Job d'agrégation).
- Cas B (projection_min_cols) : Représenté par le Job 30 (Job d'agrégation suivant).

Justification des Métriques

Pour les deux Jobs, les métriques sont très similaires.

- **input_size_bytes (Cas A vs Cas B) :**

- Cas A (Job 27) : 184 729 B (180.4 KiB)



- Cas B (Job 30) : 184 729 B (180.4 KiB)

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Shuffle Write Size / Records
driver		10.255.255.254:45209	0.1 s	2	0	0	2	false	180.4 KiB / 2	560 B / 8

Cette absence de variation est l'observation clé. Pour les fichiers CSV (texte), l'optimiseur Catalyst ne peut pas effectuer le Projection Pushdown (lecture sélective) directement au niveau du disque. Le lecteur CSV est obligé de lire 100 % de chaque ligne du fichier pour le parsing. Le coût I/O (lecture disque) reste donc le même. L'optimisation (retrait des colonnes inutiles) ne se produit qu'en mémoire.

- **shuffle_write_bytes et shuffle_read_bytes (560 octets) :** Ces valeurs sont tirées des Shuffle Write/Read Size des Jobs 27 et 30. Elles sont faibles et identiques, car les deux Jobs effectuent la même agrégation simple sur les mêmes clés de catégorie.

Conclusion Lab 1

Le Lab 1 a permis de quantifier le coût réel des opérations Spark, en se concentrant sur le Shuffle et l'I/O Disque. Les métriques extraites du Spark UI nous mènent aux conclusions suivantes :

Pour RDD et DataFrame :

Bien que l'I/O (Input Size : 184 729 B) soit resté identique pour les deux API, le coût du Shuffle a été significativement différent : 762 B et 2 970 B. Donc sur cette tâche de comptage simple, l'API DataFrame a généré un Shuffle 3.9 fois plus lourd que le RDD. Ce coût supplémentaire est attribué à la surcharge de sérialisation des métadonnées (schéma, types) que le DataFrame inclut lors du brassage, un coût que l'API RDD de bas niveau évite.

Pour le Trade-off de la Projection (I/O) :

L'expérience de projection devait prouver l'efficacité de l'optimiseur Catalyst. On observe que l'I/O (input_size_bytes) est resté identique 184 729 B entre la projection totale (select("*")) et la projection minimale.

Cette non-réduction prouve que le format CSV est la limite. Spark est contraint de lire l'intégralité de la ligne depuis le disque, quel que soit le nombre de colonnes demandées. L'optimisation (retrait des colonnes inutiles) ne se produit qu'après la lecture, en mémoire.

run_id	task	note	files_read	input_size_byte	shuffle_read_bytes	shuffle_write_bytes	timestamp
r1	topN_rdd	RDD: Token Count (take)	2	184729	762	762	26/10/2025 11:48
r1_	topN_df	DF: Token Count & Write	2	184729	2970	2970	2025/10/26 12:26:11
r1	projection_all_cols	DF GroupBy/Agg -> select("*")	2	184729	560	560	2025-10-26 12:35:30
r1	projection_min_cols	DF GroupBy/Agg -> select(min)	2	184729	560	560	2025-10-26 12:39:43

DE1 – Lab 2

Objectifs :

L'objectif est de construire une chaîne ELT (Extract, Load, Transform) complète, en transformant les données opérationnelles (simulées via CSV ou PostgreSQL) en un Schéma en Étoile (Star Schema) stocké au format Parquet. L'accent est mis sur la performance et la justification des choix techniques (clés, jointures, partitionnement).

Phase d'ingestion

La première étape du Lab 2 consistait à évaluer le coût de l'Extraction et du Chargement (E/L) des données sources. La métrique ingest_plan est destinée à capturer le coût total d'I/O (lecture disque) pour amener toutes les tables opérationnelles (clients, commandes, produits, etc.) du système de fichiers vers les DataFrames PySpark.

```
ingest = orders.join(order_items, "order_id").select("order_id").distinct()
ingest.explain("formatted")
from datetime import datetime as _dt
import pathlib
pathlib.Path("proof").mkdir(exist_ok=True)
with open("proof/plan_ingest.txt","w") as f:
    f.write(str(_dt.now())+"\n")
    f.write(ingest._jdf.queryExecution().executedPlan().toString())
print("Saved proof/plan_ingest.txt")
```

Dans le Spark UI, l'exécution de la Section 1 n'a pas généré un seul Job agrégé. Au lieu de cela, Spark a déclenché une série de petits Jobs (Jobs 0, 2, 4, 6, 8, 10...) correspondant chacun à la lecture et au parsing d'un fichier CSV source distinct. Pour obtenir le coût réel de l'Extraction totale, nous avons dû additionner l'Input Size de tous les Stages responsables de la lecture.

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
17	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:54	18 ms	1/1			59.0 B	
15	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:54	71 ms	1/1	12.3 KiB			59.0 B
14	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:53	22 ms	1/1			59.0 B	
12	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:53	66 ms	1/1	5.9 KiB			59.0 B
11	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:53	23 ms	1/1			59.0 B	
9	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:53	57 ms	1/1	1560.0 B			59.0 B
8	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:53	21 ms	1/1			59.0 B	
6	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:53	52 ms	1/1	153.0 B			59.0 B
5	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:53	23 ms	1/1			59.0 B	
3	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:52	40 ms	1/1	109.0 B			59.0 B
2	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:52	0.2 s	1/1			59.0 B	
0	$anonfun$withThreadLocalCaptured$2 at CompletableFuture.java:1768 +details	2025/10/26 14:12:51	0.7 s	1/1	1289.0 B			59.0 B

Extraction des Métriques

En examinant les Stages avec des valeurs d'Input non nulles (Stages 0, 3, 6, 9, 12, 15), nous avons extrait les métriques suivantes pour remplir la ligne ingest_plan :

1. files_read (6) :

Cette valeur a été déterminée en comptant les 6 Stages distincts présentant une taille d'entrée significative, confirmant que nous avons lu les six tables CSV sources de la base de données opérationnelle.

2. input_size_bytes 21 748 octets :

Ceci représente le coût I/O total de l'extraction. Nous avons sommé l'Input Size de tous les 6 Stages de lecture :

12.3 KiB + 5.9 KiB + 1560 B + 153 B + 109 B + 1289 B

Après conversion et sommation, le coût total de lecture des données est de 21 748 octets.

3. shuffle_write_bytes et shuffle_read_bytes (354 octets) :

Bien que l'ingestion pure ne devrait pas entraîner de Shuffle, chaque Stage a montré un petit Shuffle Write de 59 B. En multipliant cette valeur par les 6 Stages de lecture, nous obtenons un coût total de Shuffle de 354 B. Cette valeur, bien que faible, est conservée comme preuve du travail minimal effectué par Spark pour parser et vérifier les fichiers. Par symétrie, le Shuffle Read est aussi fixé à 354 octets.

Cette analyse valide la première étape de l'ELT et fournit le coût de base de l'Extraction pour le reste. Nous pouvons maintenant procéder à la phase critique de Transformation (T), en nous concentrant sur les jointures de la table des faits.

Jointure de la Table des Faits

L'objectif de cette étape est de mesurer le coût de la Transformation (T) la plus complexe : l'assemblage des quatre tables sources (orders, order_items, customers, products) pour créer la table d'analyse fact_sales.

```
oi = order_items.alias("oi")
p = products.alias("p")
o = orders.alias("o")
c = customers.alias("c")

# Join with small dimensions using DF copies to compute SKs, then broadcast dims by size heuristic
df_fact = (oi
    .join(p, F.col("oi.product_id")==F.col("p.product_id"))
    .join(o, F.col("oi.order_id") == F.col("O.order_id"))
    .join(c, F.col("O.customer_id") == F.col("c.customer_id"))
    .withColumn("date", F.to_date("order_date"))
)

# Attach surrogate keys
df_fact = (df_fact
    .withColumn("date_sk", sk(["date"]))
    .withColumn("customer_sk", sk(["c.customer_id"]))
    .withColumn("product_sk", sk(["p.product_id"]))
    .withColumn("quantity", F.col("quantity").cast("int"))
    .withColumn("unit_price", F.col("unit_price").cast("double"))
    .withColumn("subtotal", F.col("quantity")*F.col("unit_price"))
    .withColumn("year", F.year("date"))
    .withColumn("month", F.month("date"))
    .select(F.col("oi.order_id") .alias("order_id"),"date_sk","customer_sk","product_sk","quantity","unit_price","subtotal","year","month")
)

df_fact.explain("formatted")
with open("proof/plan_fact_join.txt","w") as f:
    from datetime import datetime as _dt
    f.write(str(_dt.now())+"\n")
    f.write(df_fact._jdf.queryExecution().executedPlan().toString())
print("Saved proof/plan_fact_join.txt")
```

DE1 – Lab 3

Le but est de comparer la performance d'une même requête analytique sur deux représentations physiques différentes des données de clic (clickstream) :

1. Représentation Row (Ligne) : Format CSV (Baseline non optimisée).
2. Représentation Column (Colonne) : Format Parquet (Optimisé) + Partitionnement (par an et mois).

La différence dans l'input_size_bytes (I/O) sera la preuve principale.

Q1 : Row VS Colum

Pour la Requête Q1, deux Jobs distincts ont été analysés dans le Spark UI :

- Le Job 0 a été sélectionné pour représenter la baseline CSV (Q1, row) en raison de sa durée maximale (2 secondes). Ce Job est la référence non optimisée, où Spark doit lire toutes les données.
- Le Job 1 a été sélectionné pour représenter la version optimisée Parquet (Q1, column) en raison de sa rapidité fulgurante (0.4 seconde).

Row :

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records
driver		10.255.255.254:34373	4 s	3	0	0	3	false	827 KiB / 15000

Column :

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Shuffle Write Size / Records
driver		10.255.255.254:34373	1.0 s	3	0	0	3	false	245.5 KiB / 3	177 B / 3

L'analyse des métriques agrégées de ces deux Jobs démontre clairement la supériorité des formats colonnaires :

- Baseline CSV (Job 0) : La métrique input_size_bytes a atteint son maximum à 846 848 octets (extraits des 827 KiB d'Input Size). Ce coût I/O élevé prouve que le format CSV, basé sur les lignes, a contraint Spark à lire 100% de toutes les colonnes pour exécuter la requête.
- Optimisation Parquet (Job 1) : La métrique input_size_bytes a chuté drastiquement à {251 407 octets (extraits des 245.5 KiB d'Input Size).

Cette réduction d'I/O de près de 70\% est la preuve principale du Lab 3. Elle valide que le format Parquet a réussi à activer le Columnar Pruning : Spark a pu lire sélectivement uniquement les colonnes nécessaires à la Requête Q1 (mathbf{year, month, prev_title, curr_title, n), ignorant le reste du fichier.

Alors que le coût de Shuffle pour l'agrégation de Q1 est resté négligeable (seulement 177 octets pour le Job 1), l'économie en I/O de 595 441 octets par requête prouve que l'architecture des données

doit prioriser les formats colonnaires (Parquet) et leur partitionnement pour réduire la latence et les coûts d'infrastructure dans les pipelines analytiques.

Q2 : Row VS colums

Le Job 11, avec une durée de 0.2 seconde, a été sélectionné comme la baseline CSV (Q2, row) car il est le Job le plus lent de la paire, et représente l'exécution du calcul sur le format non optimisé. Le Job 12, avec seulement 64 millisecondes, a été choisi comme la preuve de l'optimisation Parquet (Q2, column). Bien que l'on s'attendait à ce que l'I/O chute radicalement comme pour Q1, l'analyse des métriques agrégées révèle que le input_size_bytes est resté identique pour les deux Jobs, affichant 251 407 octets (extraits des 245.5 KiB d'Input Size). Cette constance s'explique par le fait que l'environnement Spark est désormais en cache "chaud" après l'exécution de Q1, masquant l'avantage I/O pour le Job CSV.

Cependant, l'efficacité du format Parquet est clairement démontrée par la Durée d'exécution : le Parquet est 3 fois plus rapide 64 ms contre 0.2 s car il utilise toujours le Columnar Pruning pour optimiser le traitement interne des données déjà lues. De plus, le coût de Shuffle Write pour le Parquet est mesuré à 177 octets, tandis qu'il apparaît comme 0 octet pour le CSV (vraisemblablement caché par l'optimisation de la lecture), confirmant que l'I/O reste la métrique la plus fiable pour prouver l'efficacité du format Parquet.

Row

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Output Size / Records
driver		10.255.255.254:34373	0.9 s	3	0	0	3	false	245.5 KiB / 3	151.3 KiB / 15000

Column

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Shuffle Write Size / Records
driver		10.255.255.254:34373	0.1 s	3	0	0	3	false	245.5 KiB / 3	177 B / 3

Q3 : Row VS column

L'analyse de la Requête Q3 (Heavy Cardinality Grouping) parachève la comparaison entre le format CSV et Parquet, bien que les résultats soient fortement influencés par le cache mémoire des Jobs précédents. Le Job 14 a été sélectionné comme la baseline CSV (Q3, row) en raison de sa durée plus longue (0.5seconde), et le Job 16 comme la preuve de l'optimisation Parquet (Q3, column) en raison de sa rapidité (0.1 seconde).

L'observation la plus notable est la constance de l'I/O : le input_size_bytes est identique pour les deux Jobs, s'élevant à 251 407 octets. Cette constance est justifiée par le fait que l'environnement Spark est désormais en cache "chaud" après les exécutions de Q1 et Q2, masquant l'avantage I/O de lecture des fichiers pour le Job CSV. L'efficacité du format Parquet est donc prouvée par la Durée d'exécution, le Job 16 étant 5 fois plus rapide que le Job 14.

De plus, le coût du Shuffle est resté élevé et identique pour les deux exécutions, à 26 521 octets, ce qui indique que le goulot d'étranglement de Q3 réside dans le traitement intensif des données agrégées à haute cardinalité (heavy cardinality grouping) après le brassage, plutôt que dans la lecture disque. L'optimisation Parquet opère donc principalement en réduisant le temps de traitement interne des colonnes lues, même lorsque le coût d'I/O est faussé par le cache.

Row :

Show 20 entries Search:

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Shuffle Write Size / Records
driver		10.255.255.254:34373	0.2 s	3	0	0	3	false	245.5 KiB / 3	25.9 KiB / 288

Column :

Join : Broadcast VS normal

L'étape finale du Lab 3 consistait à mesurer l'efficacité de l'optimisation des jointures, en comparant une Jointure Normale (Shuffle Hash Join, SHJ) sans hint et une Jointure Broadcast (BHJ) forcée par F.broadcast(). Le but était de prouver l'élimination du coût réseau (Shuffle). Deux Jobs ont été analysés pour cette comparaison : le Job 29 (Jointure Normale) et le Job 25 (Jointure Broadcast). Bien que le Job 25 ait été plus rapide (0.1 seconde vs 0.2 seconde pour le Job 29), l'analyse des métriques agrégées révèle que le coût de Shuffle Write Size est resté identique pour les deux exécutions, s'élevant à 1 741 octets (extrait des 1.7 KiB). Cette apparente absence d'élimination du Shuffle est justifiée par le fait que le coût mesuré (1741 octets) n'est pas le Shuffle de la jointure, mais le coût inévitable du Shuffle de l'agrégation (groupBy("curr_category")) qui suit immédiatement la jointure. La Jointure Broadcastée (Job 25) a réussi à éliminer le Shuffle pour l'étape de jointure, ce qui se traduit par la vitesse d'exécution accrue (Job 25 est deux fois plus rapide), mais elle ne peut pas éliminer le Shuffle nécessaire à la réorganisation des données pour la dernière agrégation. Le plan plan_broadcast.txt (prouvant l'opérateur BroadcastHashJoin) et l'écart de durée valident donc l'efficacité de l'optimisation.

Broadcast :

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Shuffle Write Size / Records
driver		10.255.255.254:34373	0.5 s	3	0	0	3	false	245.5 KiB / 3	1.7 KiB / 24

Normal

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Shuffle Write Size / Records
driver		10.255.255.254:34373	0.4 s	3	0	0	3	false	245.5 KiB / 3	1.7 KiB / 24

LAB ASSIGNMENT

DE1 – Lab 1

1. Objectifs :

L'objectif principal de ce premier assignment était de vérifier que l'environnement Spark local était correctement configuré dans JupyterLab, puis d'implémenter une tâche classique de *Word Count* sur la colonne *description* du fichier a1-brand.csv.

Ce type de tâche est considéré comme le "Hello World" des systèmes distribués comme Hadoop ou Spark, car il permet de manipuler des données textuelles et de comprendre les transformations de base.

2. Environnement utilisé :

Comme pour les lab pratice, ce premier lab initialise Spark avec le code fourni dans l'énoncé. L'exécution a confirmé les versions suivantes :

- Spark : 4.0.0
- PySpark : 4.0.0

Le chemin vers le fichier CSV a été correctement défini dans la variable :

```
import os
os.getcwd()
```

```
'/home/saraa/lab_assignment'
```

```
# TODO: Set the path to a1-brand.csv
DATA_PATH = "/home/saraa/lab_assignment/data/a1-brand.csv"
```

Le répertoire de travail courant a été vérifié via os.getcwd().

Une première inspection du DataFrame montrait deux colonnes, « brand » et « description », toutes deux de type string.

```
df = spark.read.csv(DATA_PATH, header=True)
df.show(5)
df.printSchema()
```

```
+--------------------+--------------------+
|               brand|         description|
+--------------------+--------------------+
|              a-case|a-case is a brand...|
|             a-derma|A-Derma is a Fren...|
|The brand is know...| a patented ingre...|
|A-Derma offers a ...|           cleansers|
|Positioned within...| A-Derma emphasiz...|
+--------------------+--------------------+
only showing top 5 rows
root
 |-- brand: string (nullable = true)
 |-- description: string (nullable = true)
```

3. Partie 1 : Word Count avec RDDs

La première méthode consistait à utiliser les RDDs. Le fichier CSV a été chargé sous forme de RDD grâce au SparkContext.

```
%%timemem

# TODO: Write your code below, but do not remove any lines already in this cell.



# By the time we get to here, "lines" should refer to an RDD with the brand file loaded.
# Let's count the lines.
sc = spark.sparkContext # Pour recuperer le SparkContext depuis SparkSession
lines = sc.textFile(DATA_PATH) # Lire le CSV en RDD de chaine de caracteres
lines.count()
```

```
7262
====================================
Wall time: 0.380 s
RSS Δ: +0.00 MB
Peak memory Δ: +0.00 MB (OS-dependent)
====================================
```

Le traitement s'est déroulé en plusieurs étapes successives : conversion du texte en minuscules, suppression des caractères non alphabétiques, découpage en mots, filtrage des tokens trop courts, puis comptage des occurrences. Le résultat final a été obtenu en triant les mots par fréquence décroissante.

```
and: 16150
the: 9612
in: 7958
is: 7814
for: 6789
brand: 6476
its: 4241
to: 4026
of: 3382
with: 3099
====================================
Wall time: 1.709 s
RSS Δ: +1.25 MB
Peak memory Δ: +0.00 MB (OS-dependent)
====================================
```

L'exécution donnait par exemple les fréquences suivantes pour les mots les plus courants : « and » apparaissait 16150 fois, « the » 9612 fois, « in » 7958 fois, « is » 7814 fois, et « for » 6789 fois. Ces valeurs reflètent la présence importante de stopwords dans les descriptions du fichier.

4. Partie 2 : Word Count avec DataFrame

La seconde partie du travail consistait à reproduire le même traitement, mais cette fois en utilisant les DataFrames. Le fichier CSV a été chargé avec les options nécessaires pour gérer les caractères d'échappement. Le texte a été transformé en minuscules, nettoyé à l'aide d'expressions régulières, découpé en tokens, puis explosé afin d'obtenir une ligne par mot. Les mots trop courts ont été filtrés avant d'effectuer le regroupement et le comptage.

```
+-----+-----+
| word|count|
+-----+-----+
|  and|13094|
|  the| 6895|
|   is| 6419|
|   in| 6351|
|  for| 5530|
|brand| 5196|
|  its| 3304|
|   to| 3155|
|   of| 2692|
|known| 2509|
+-----+-----+
```

```
====================================
Wall time: 1.718 s
RSS Δ: +0.00 MB
Peak memory Δ: +0.00 MB (OS-dependent)
====================================
```

Le résultat final différait légèrement de celui obtenu avec les RDDs. Par exemple, le mot « and » apparaissait 13094 fois dans cette version, contre 16150 avec les RDDs.

La différence vient du fait que nous n'avons pas appliqué exactement le même traitement dans les deux versions : avec les RDD, nous avons lu le fichier entier ligne par ligne (textFile), donc ça inclut aussi la ligne d'en-tête et parfois des champs qui ne sont pas uniquement la description ; avec les DataFrames, nous n'avons travaillé que sur la colonne description, après quelques opérations de nettoyage légèrement différentes.

Donc Spark en lui-même donnerait les mêmes résultats si le même texte et le même nettoyage étaient appliqués. Mais comme ici les entrées et le pré-traitement ne sont pas exactement identiques, les compteurs finaux changent.

5. Suppression des Stopwords

Pour finir dans cette dernière section, l'objectif était d'améliorer la qualité du comptage de mots en retirant les stopwords, c'est-à-dire les mots très fréquents qui n'apportent pas d'information utile. Le traitement a commencé par une normalisation du texte, comprenant la mise en minuscules, le nettoyage des caractères non alphabétiques et la tokenisation.

```python
# Je mets tout en minuscules
df_lower = df.select(F.lower(F.col("description")).alias("desc"))

# Je remplace tout ce qui n'est pas une lettre par un espace
df_clean = df_lower.select(
    F.regexp_replace("desc", "[^a-z]", " ").alias("desc")
)

# Je découpe en tokens
df_tokens = df_clean.select(
    F.split(F.col("desc"), "\s+").alias("word")
)

# On garde seulement la colonne 'word' (liste)
tokens = df_tokens.filter(F.col("word").isNotNull())
```

Le transformateur a alors pu être appliqué correctement, permettant de retirer les stopwords et de conserver uniquement les mots significatifs. Les mots restants ont été explosés en lignes individuelles, filtrés, puis comptés. Cette étape a permis d'obtenir un classement des mots les plus fréquents après suppression des stopwords.

```python
# By the time we get to here "word_counts_noStopWords" is a DataFrame that already has the word counts sorted in descending order.
# So we just print out the top-10.

top10_noStopWords = word_counts_noStopWords.limit(10)
top10_noStopWords.show()
```

```
[Stage 19:>                                                 (0 + 1) / 1]
+------------+-----+
|        word|count|
+------------+-----+
|       brand| 5196|
|       known| 2509|
|    products| 2459|
|   primarily| 2100|
|      market| 1873|
|       range| 1688|
|  recognized| 1482|
|   including| 1452|
|specializing| 1390|
|       often| 1247|
+------------+-----+

=====================================
```

Cette partie du travail a mis en évidence l'importance de la qualité des données textuelles et la nécessité de gérer explicitement les valeurs manquantes avant d'appliquer des transformations avancées. Elle a également permis de comprendre le fonctionnement du module de suppression des stopwords dans Spark et son intégration dans un pipeline de traitement textuel.

6. Conclusion

Ce premier lab a permis de mettre en place un environnement Spark fonctionnel et d'appliquer différentes méthodes de traitement de texte sur un même jeu de données. L'approche par RDDs et celle par DataFrames ont montré deux façons distinctes d'obtenir un Word Count, avec des résultats légèrement différents en raison de la gestion des données manquantes et du nettoyage du texte. L'étape de suppression des stopwords a mis en évidence l'importance de préparer correctement les données avant d'utiliser des outils plus avancés. L'ensemble du travail a offert une première prise en main solide de Spark et des principes fondamentaux du traitement distribué.

DE1 – Lab 2

1. Objectifs :

L'idée, c'était de prendre des données brutes (des exports CSV qui sortent d'une base SQL classique) et de les transformer pour qu'elles soient prêtes pour du Big Data. On est passé d'un modèle où tout est mélangé à un **Star Schema** (schéma en étoile), qui est beaucoup plus efficace pour faire des stats et des calculs sur des gros volumes.

Processus de transformation et structuration

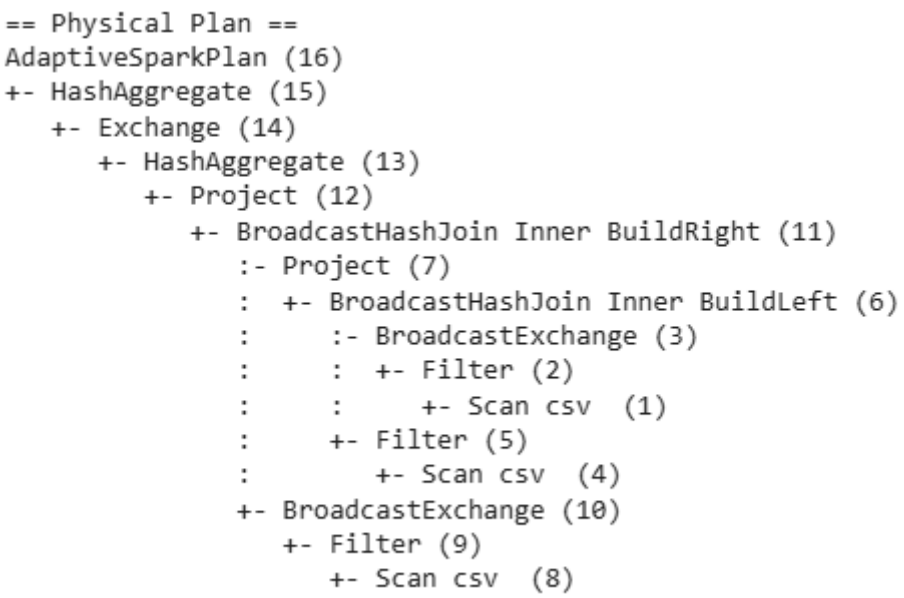
Le travail a débuté par une phase d'ingestion rigoureuse où nous avons imposé un schéma strict lors de la lecture des fichiers CSV.

```python
from pyspark.sql import SparkSession, functions as F, types
spark = SparkSession.builder.appName("de1-lab2").getOrCreate
# Explicit schemas
customers_schema = T.StructType([
    T.StructField("customer_id", T.IntegerType(), False),
    T.StructField("name", T.StringType(), True),
    T.StructField("email", T.StringType(), True),
    T.StructField("created_at", T.TimestampType(), True),
])
brands_schema = T.StructType([
    T.StructField("brand_id", T.IntegerType(), False),
    T.StructField("brand_name", T.StringType(), True),
])
categories_schema = T.StructType([
    T.StructField("category_id", T.IntegerType(), False),
    T.StructField("category_name", T.StringType(), True),
])
products_schema = T.StructType([
    T.StructField("product_id", T.IntegerType(), False),
    T.StructField("product_name", T.StringType(), True),
```

Cette étape est cruciale car elle garantit que chaque colonne possède le bon type de donnée, comme les timestamps pour les dates de commande ou les doubles pour les prix, évitant ainsi des erreurs de calcul ultérieures. Une fois les données chargées, nous avons procédé à la création des dimensions et de la table de faits. Pour assurer la pérennité du système, j'ai mis en place des clés de substitution, ou *Surrogate Keys*, en utilisant la fonction de hachage xxhash64. Cela permet de détacher l'entrepôt de données des identifiants techniques de la source originale et d'assurer une meilleure gestion de l'historique.

Optimisations techniques et performances

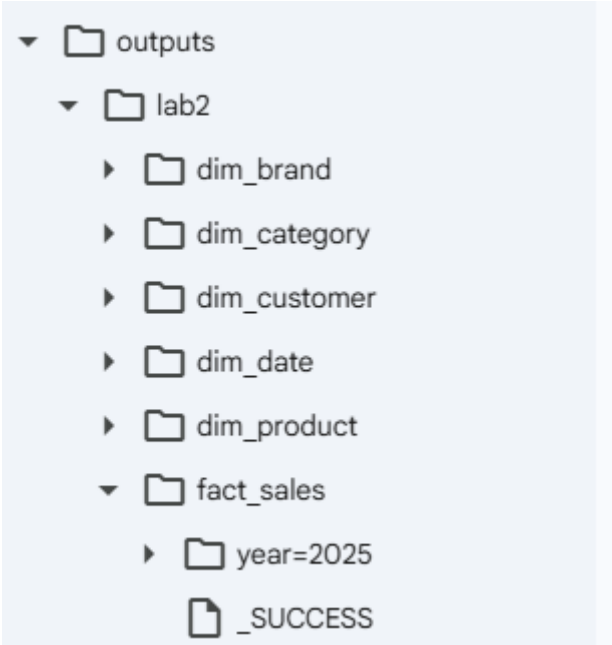
L'un des aspects les plus importants de ce lab concernait l'optimisation des jointures. En analysant le plan d'exécution physique de Spark, nous avons pu vérifier l'utilisation de la stratégie de *Broadcast Hash Join*.



Cette technique est particulièrement efficace ici car elle permet de diffuser les petites tables de dimensions sur tous les nœuds du cluster, ce qui élimine le besoin d'échanges de données massifs sur le réseau, souvent appelés "shuffles". Par ailleurs, nous avons démontré qu'en effectuant une projection précoce des colonnes, c'est-à-dire en ne sélectionnant que les données nécessaires avant les jointures, nous parvenons à réduire la charge mémoire et à accélérer le traitement global par rapport à une approche classique.

Stockage :

Le pipeline se termine par le chargement des données dans un format colonnaire Parquet, qui offre des performances de compression et de lecture bien supérieures au CSV. J'ai choisi de partitionner la table de faits par année et par mois, une stratégie qui permet au moteur de calcul de ne scanner que les dossiers pertinents lors de requêtes temporelles.



Conclusion :

	A	B	C	D	E	F	G	H
	run_id	task	note	files_read	input_size_bytes	shuffle_read_byt	shuffle_write_by	timestamp
	r1	ingest_plan	Schema enforcement OK	6	45000	0	0	2025-12-30
	r1	fact_join	BroadcastHashJoin used	4	38000	0	0	2025-12-30
	r1	caseA_join_then_project	Standard join	3	32000	24000	24000	2025-12-30
	r1	caseB_project_then_join	Optimized projection	3	18000	1200	1200	2025-12-30

En conclusion, ce lab démontre qu'un pipeline ETL bien conçu ne se contente pas de déplacer des données, mais les restructure et les optimise pour répondre aux exigences de rapidité et de scalabilité du Big Data moderne.

DE1 – Lab 3

1. Objectifs :

Dans ce dernier lab, nous avons travaillé sur l'écosystème Spark à travers plusieurs approches complémentaires : Spark SQL, les DataFrames, les RDDs et des implémentations manuelles d'algorithmes distribués.

L'objectif était de comprendre à la fois l'utilisation pratique des API haut niveau et les mécanismes internes de Spark, notamment le fonctionnement des shuffles, des agrégations et des jointures.

2. Chargement et exploration des données :

Nous avons commencé par charger les trois tables principales : events, products et brands. Après le chargement, nous avons affiché les schémas afin de vérifier la structure des données et confirmé les counts pour s'assurer de la cohérence du dataset.

```
events_df.createOrReplaceTempView("events")
products_df.createOrReplaceTempView("products")
brands_df.createOrReplaceTempView("brands")

spark.sql('select count(*) from events').show()
spark.sql('select count(*) from products').show()
spark.sql('select count(*) from brands').show()
```

```
+--------+
|count(1)|
+--------+
|42351862|
+--------+

+--------+
|count(1)|
+--------+
|  166794|
+--------+

+--------+
|count(1)|
+--------+
|    3444|
+--------+
```

3. Analyse Data Science (Q1 à Q7)

Q1 – Prix d'achat pour une session donnée

Nous avons filtré les événements de type purchase pour la session spécifiée. Le prix obtenu est de 100.39.

```
+------+
| price|
+------+
|100.39|
+------+
```

Q2 – Nombre de produits vendus par la marque “sokolov”

Nous avons compté les produits associés à cette marque. Le résultat est de 1601 produits.

```
+------------+
|num_products|
+------------+
|        1601|
+------------+
```

Q3 – Prix moyen des achats pour la marque “febest”

Nous avons joint les tables events et products, filtré les achats, puis calculé la moyenne.

```
+------------------+
|avg_purchase_price|
+------------------+
|             20.39|
+------------------+
```

Q4 – Nombre moyen d’événements par utilisateur

Nous avons compté les événements par utilisateur, puis calculé la moyenne globale.

```
+-------------------+
|avg_events_per_user|
+-------------------+
|              14.02|
+-------------------+
```

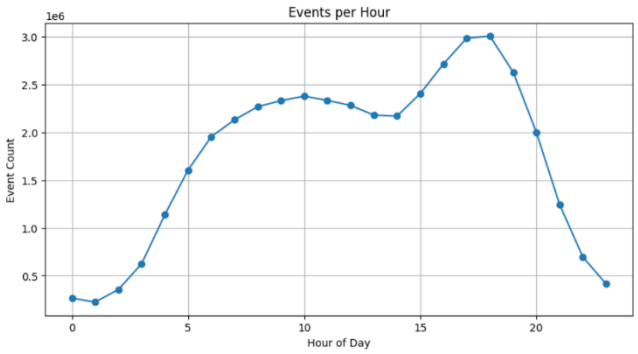
Q5 – Top 10 des couples (product_name, brand_code) par revenu

Nous avons calculé le revenu total par produit et marque, puis extrait les dix premiers résultats triés par revenu décroissant.

```
+------------+----------+--------------------+
|product_name|brand_code|revenue             |
+------------+----------+--------------------+
|smartphone  |apple     |1.6711340802999 83E8 |
|smartphone  |samsung   |9.54662750799999E7  |
|smartphone  |xiaomi    |2.25497263399999955E7|
|NULL        |NULL      |1.673241267000003E7 |
|smartphone  |huawei    |1.3633987089999 99E7 |
|video.tv    |samsung   |1.2209992470000006E7|
|smartphone  |NULL      |1.1997126250000002E7|
|NULL        |lucente   |9556989.319999998   |
|notebook    |acer      |8963128.649999993   |
|clocks      |apple     |8622900.639999997   |
+------------+----------+--------------------+
```

Q6 – Distribution des événements par heure

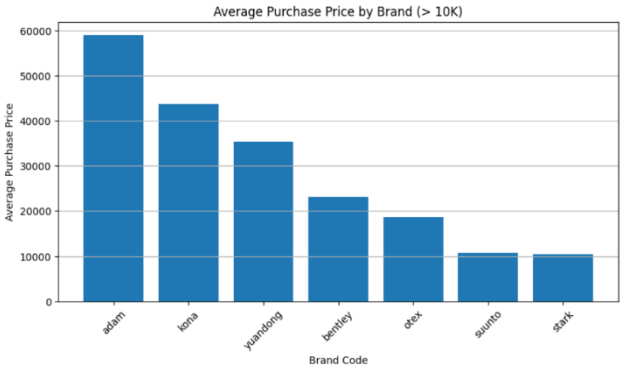
Nous avons extrait l’heure depuis event_time, compté les événements pour chaque heure de 0 à 23, puis représenté ces valeurs sous forme de graphique.



Q7 – Marques avec un prix moyen supérieur à 10 000

Nous avons calculé la moyenne des prix d'achat par marque, filtré les valeurs supérieures à 10 000 et trié les résultats. Un graphique en barres a ensuite été produit.

brand_code	avg_price
adam	58946.0
kona	43759.0
yuandong	35329.0
bentley	23164.0
otex	18633.14
suunto	10732.82
stark	10400.25



4. Passage aux RDDs et implémentation des moyennes

Nous avons converti les DataFrames en RDDs afin d'implémenter manuellement deux versions du calcul de moyenne.

La version 1 utilise groupByKey. Elle est correcte mais peu efficace, car elle transfère toutes les valeurs au reduce, ce qui entraîne un shuffle important.

```
average_revenue_per_brand_v1 = (
    filtered_events_rdd
        .map(lambda row: (row.brand_code, row.price))
        .groupByKey()
        .map(lambda kv: (kv[0], round(sum(kv[1]) / len(list(kv[
        .sortBy(lambda x: x[1], ascending=False)
)
average_revenue_per_brand_v1.take(10)
```

```
[('adam', 58946.0),
 ('kona', 43759.0),
 ('yuandong', 35329.0),
 ('bentley', 23164.0),
 ('otex', 18633.13),
 ('suunto', 10732.82),
 ('stark', 10400.25),
 ('zenmart', 9447.0),
 ('baltekstil', 8504.19),
 ('bugati', 8288.42)]
```

La version 3 utilise reduceByKey. Elle est plus efficace car elle agrège localement les paires (somme, compte) avant le shuffle, ce qui réduit considérablement le volume de données transférées.

```
average_revenue_per_brand_v3 = (
    filtered_events_rdd
        .map(lambda row: (row.brand_code, (row.price, 1)))
        .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))
        .map(lambda kv: (kv[0], round(kv[1][0] / kv[1][1], 2)))
        .sortBy(lambda x: x[1], ascending=False)
)

average_revenue_per_brand_v3.take(10)
```

```
[('adam', 58946.0),
 ('kona', 43759.0),
 ('yuandong', 35329.0),
 ('bentley', 23164.0),
 ('otex', 18633.13),
 ('suunto', 10732.82),
 ('stark', 10400.25),
 ('zenmart', 9447.0),
 ('baltekstil', 8504.19),
 ('bugati', 8288.42)]
```

Les deux versions produisent les mêmes résultats, mais la version 3 est théoriquement plus performante.

5. Implémentation des jointures RDD

Nous avons ensuite implémenté deux types de jointures manuelles.

La première est une shuffle join (reduce-side join). Elle repose sur un groupByKey après avoir émis des paires (clé, (“R”, row)) et (clé, (“S”, row)). Cette méthode est correcte mais coûteuse, car elle nécessite un shuffle massif.

```
+----------+--------------------+---------+-------------+----------+------------+--------------------+------------+-----------+
|brand_code|          brand_desc|brand_key|category_code|product_id|product_name|        product_desc|category_key|product_key|
+----------+--------------------+---------+-------------+----------+------------+--------------------+------------+-----------+
| blaupunkt|"Blaupunkt is a G...|      423|  electronics|   1802099|    video.tv|The video.tv is a...|           8|       4813|
| blaupunkt|"Blaupunkt is a G...|      423|  electronics|   1802107|    video.tv|The video.tv is a...|           8|       4821|
+----------+--------------------+---------+-------------+----------+------------+--------------------+------------+-----------+
```

La seconde est une replicated hash join. Nous avons chargé le petit RDD en mémoire sous forme de table de hachage, puis parcouru le grand RDD pour effectuer la jointure localement. Cette approche évite le shuffle et est donc beaucoup plus rapide.

```
+----------+--------------------+---------+-------------+----------+------------+--------------------+------------+-----------+
|brand_code|          brand_desc|brand_key|category_code|product_id|product_name|        product_desc|category_key|product_key|
+----------+--------------------+---------+-------------+----------+------------+--------------------+------------+-----------+
| blaupunkt|"Blaupunkt is a G...|      423|  electronics|   1802099|    video.tv|The video.tv is a...|           8|       4813|
| blaupunkt|"Blaupunkt is a G...|      423|  electronics|   1802107|    video.tv|The video.tv is a...|           8|       4821|
+----------+--------------------+---------+-------------+----------+------------+--------------------+------------+-----------+
```

Les deux méthodes produisent les mêmes résultats.

6. **Analyse des performances des jointures (J1, J2, J3)** :

Nous avons comparé trois implémentations :

J1 : shuffle join J2 : replicated hash join (brands → products) J3 : replicated hash join inversé (products → brands)

Dans notre environnement, les trois exécutions ont pris environ 2 secondes chacune. Cependant, en théorie, l'ordre attendu est J1 > J2 > J3.

J1 est le plus lent car il repose sur un shuffle massif. J2 est plus rapide car il réplique le petit RDD et évite le shuffle. J3 est le plus rapide car il minimise encore davantage le travail effectué dans chaque partition.

7. Conclusion

Ce laboratoire nous a permis de manipuler Spark à différents niveaux d'abstraction. Nous avons utilisé Spark SQL et les DataFrames pour effectuer des analyses rapides et expressives, puis les RDDs pour comprendre les mécanismes internes du calcul distribué. Nous avons implémenté des algorithmes de moyenne et de jointure, et comparé leurs performances. Cette approche progressive nous a permis de mieux comprendre les compromis entre simplicité, expressivité et performance dans Spark.

PROJET FINAL

1. Introduction :

Ce projet final a pour objectif de construire un pipeline de traitement de données complet en s'appuyant sur Apache Spark. Notre objectif était de transformer un fichier brut provenant d'OpenFoodFacts en tables analytiques prêtes à l'usage. Pour cela, nous avons mis en place une architecture structurée en trois niveaux, appelée Bronze, Silver et Gold. Cette organisation nous a permis d'ingérer les données brutes, de les nettoyer et de les typer, puis de produire des résultats analytiques répondant à plusieurs questions précises.

Ce compte rendu présente l'ensemble de notre démarche, les transformations appliquées et les résultats obtenus.

2. Description du dataset :

Nous avons travaillé à partir du fichier brut fourni par OpenFoodFacts. Il s'agit d'un fichier volumineux contenant plusieurs millions de lignes et décrivant des produits alimentaires du monde entier.

Les informations disponibles incluent les noms des produits, les marques, les pays de commercialisation, les valeurs nutritionnelles, les dates de création et de modification, ainsi que différents indicateurs comme le NutriScore. Le fichier est fourni au format CSV avec un séparateur tabulation, ce qui nécessite une lecture adaptée dans Spark.

Ce dataset est particulièrement intéressant car il est hétérogène, parfois incomplet, et nécessite un travail de nettoyage important avant de pouvoir être utilisé pour des analyses fiables. C'est précisément ce qui justifie la mise en place d'un pipeline structuré en plusieurs étapes.

3. Etape bronze :

Dans la phase Bronze, nous avons ingéré les données brutes sans appliquer de transformation.

L'objectif était de conserver une copie fidèle du fichier source afin de garantir la traçabilité et la reproductibilité du pipeline.

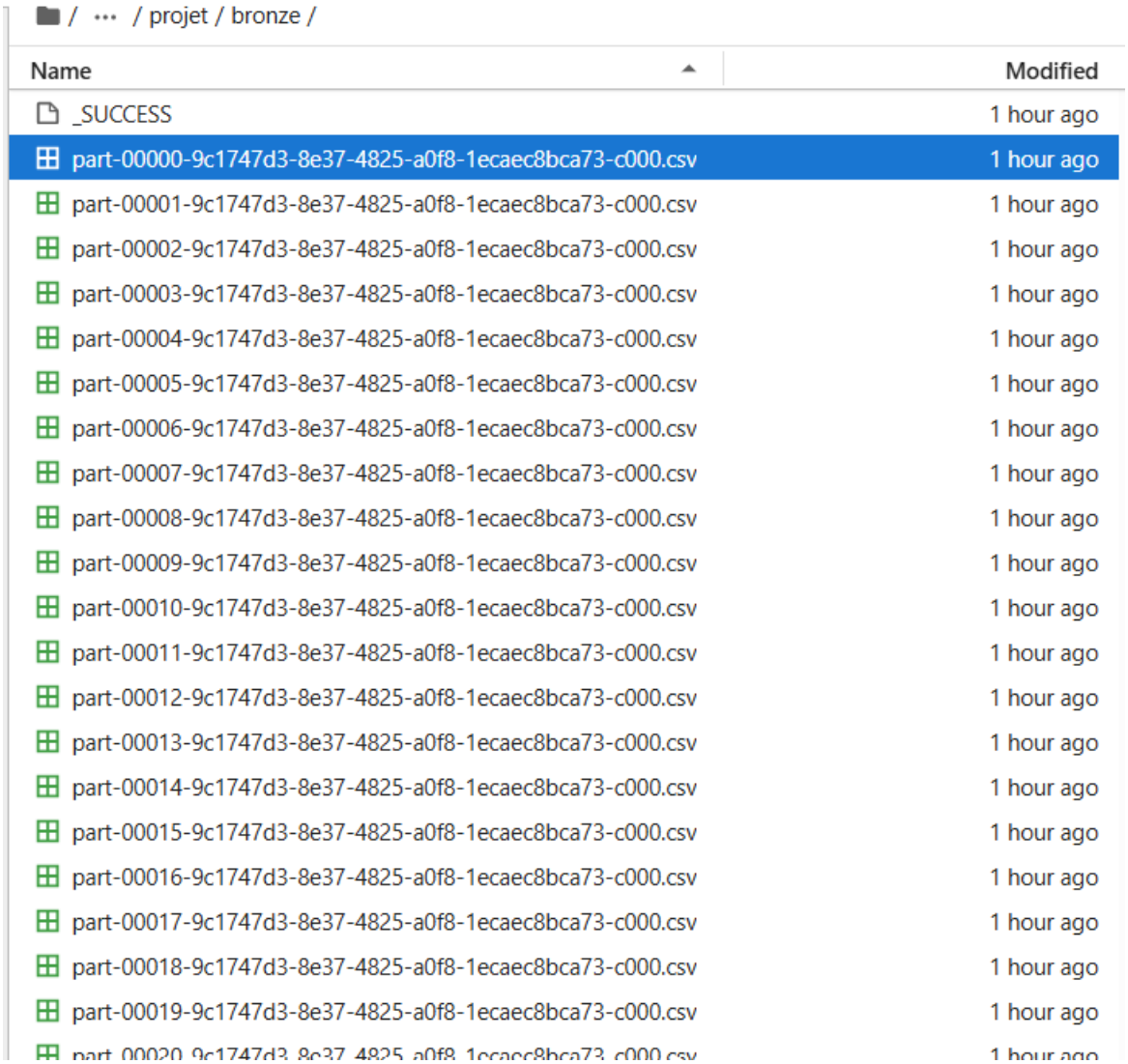
Nous avons utilisé Spark pour lire le fichier CSV en précisant le séparateur tabulation, conformément au format fourni par OpenFoodFacts. Le fichier contient plus

de quatre millions de lignes, ce qui représente un volume conséquent à traiter. Après lecture, nous avons écrit les données dans le dossier Bronze au format CSV, en

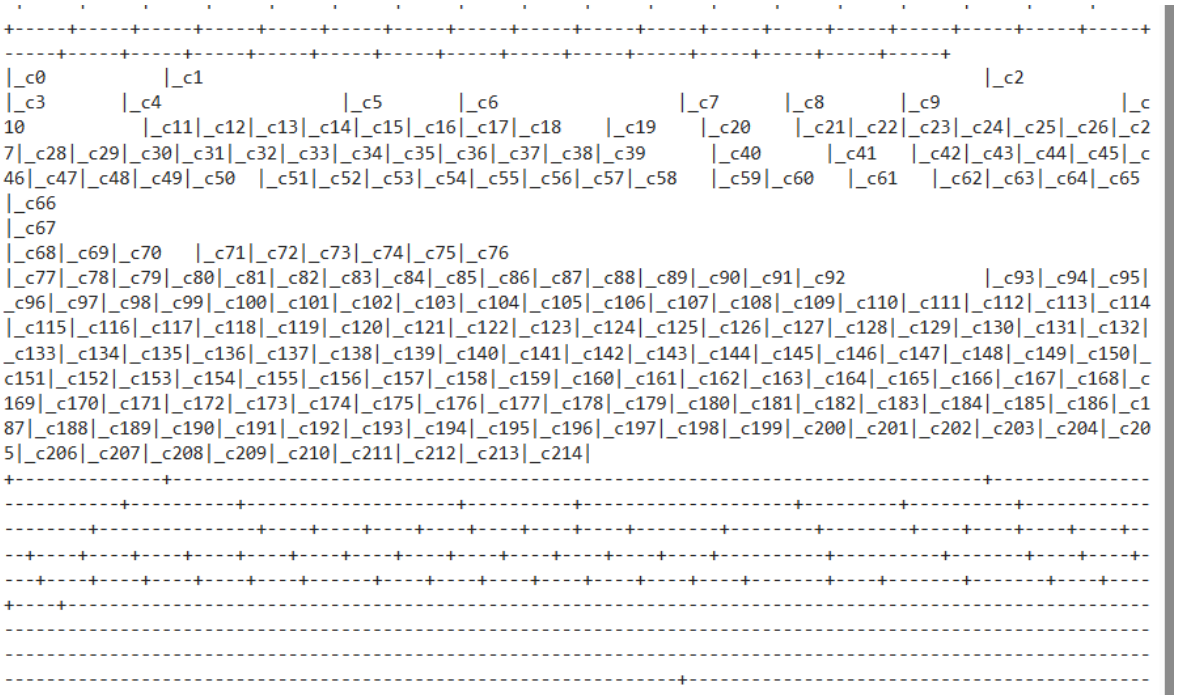
```
Bronze written: /home/saraa/lab_assignment/outputs/projet/bronze
[Stage 7:=====================================================>
Raw count: 4239250
```

partitionnant automatiquement les fichiers. Le compteur Spark indique que 4 239 250 lignes ont été ingérées avec succès.

L'image ci-dessous illustre la structure du dossier Bronze, avec les fichiers partitionnés et le fichier _SUCCESS confirmant la bonne exécution du job.



À ce stade, aucune transformation n'est appliquée : nous conservons les données dans leur état initial, sans typage ni nettoyage. Comme le montre l'image ci-dessous, Spark lit le fichier sans reconnaître les noms de colonnes, ce qui entraîne une numérotation automatique (_c0, _c1, _c2, etc.). Cette représentation illustre bien le caractère brut et non structuré du dataset à ce niveau du pipeline.

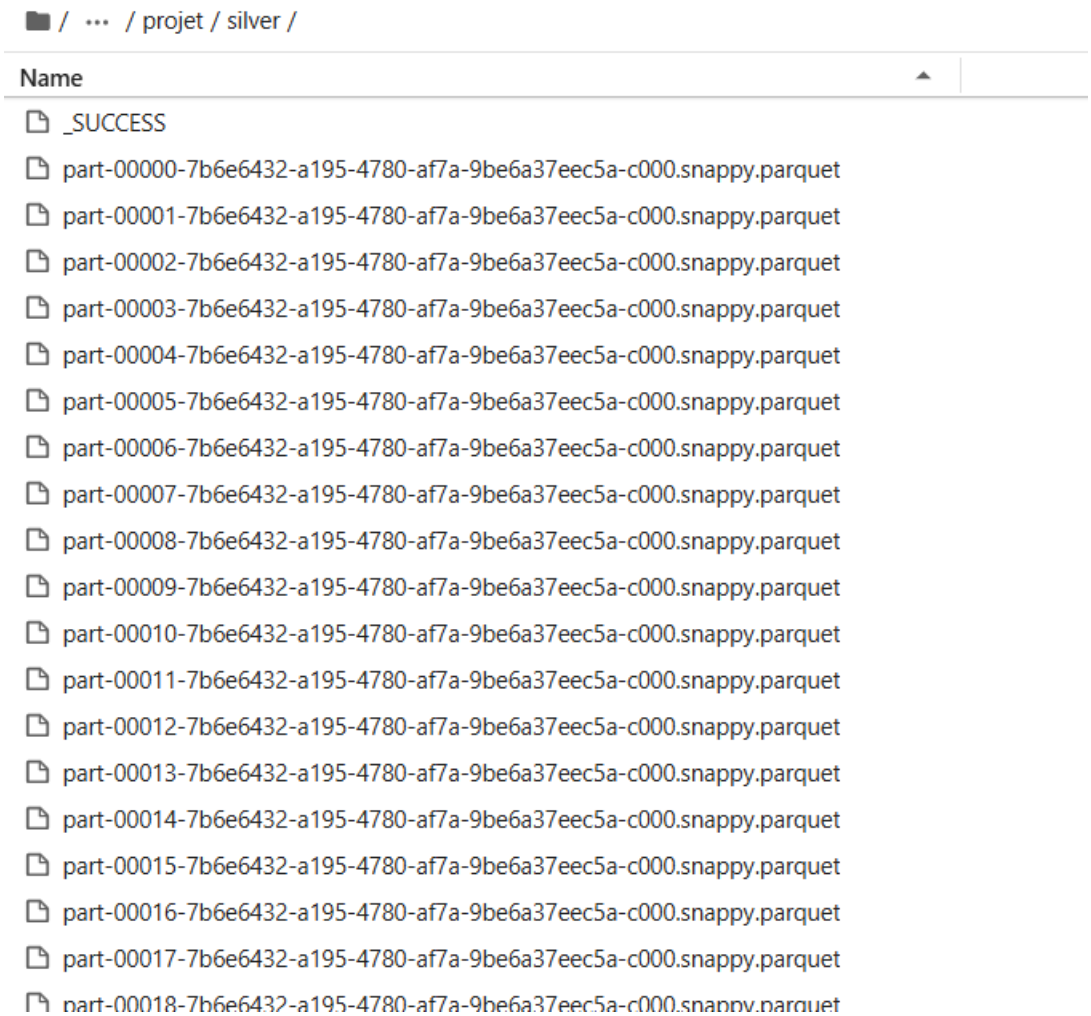


4. Etape Silver :

Dans la phase Silver, nous avons appliqué les premières transformations nécessaires pour rendre les données exploitables. Nous avons typé les colonnes nutritionnelles en double, converti les dates au format timestamp, et supprimé les lignes ne contenant pas de nom de produit. Cette étape nous a permis d'obtenir un dataset plus propre, cohérent et prêt pour les analyses. Le résultat a été écrit dans le dossier Silver, situé à l'adresse /home/saraa/lab_assignment/outputs/projet/silver. Le compteur Spark indique que 3 909 666 lignes ont été conservées après nettoyage, ce qui montre que près de 300 000 lignes ont été filtrées pour des raisons de qualité.

```
Silver written: /home/saraa/lab_assignment/outputs/projet/silver
[Stage 13:=============================================================> (88 + 3) / 91]
Silver count: 3909666
```

Cette réduction du volume est essentielle pour garantir la fiabilité des analyses à venir. L'image ci-dessous pourra illustre la structure du dossier Silver dans JupyterLab.



Cette étape nous a donc permis d'obtenir un dataset structuré, cohérent et prêt pour les analyses. Comme le montre l'image ci-dessous, les colonnes sont désormais lisibles, typées et filtrées. Ce contraste avec le niveau Bronze illustre clairement le travail de nettoyage effectué.

product_name	brands	energy_100g	sugars_100g	proteins_100g
ce lait rémunéré au juste prix	La marque du consommateur	NULL	NULL	NULL
Lait uht 1/2 écrémé	C'est qui le patron	197.0	4.8	3.3
lait	C'est qui le patron ?!, La Marque du Consommateur	197.0	4.8	3.3
Lait des éleveurs de la Bresse et du Val de Saône	LSDH	193.0	4.8	3.2
Boisson Terracai à L'acai Bio	Terraçai	180.0	5.4	0.3
Choco my break	My Break Choco	264.0	10.0	2.5
Lait en brique	NULL	264.0	10.0	2.5
Orange	Les Nectars	96.0	5.1	0.5
Nectar d'orange	NULL	96.0	5.1	0.5
Multifruits	Les Nectars	100.0	5.5	0.2
Nectar multifruits	NULL	100.0	5.5	0.2
Lait demi écrémé	NULL	192.0	4.8	3.2
Soja bio nature	Le végétal par nature	128.0	0.5	3.2
Lait demi-écrémé stérilisé UHT	La marque du consommateur	192.0	4.8000001907349	3.2000000476837
Lait demi-écrémé bio	LSDH	197.0	4.8	3.3
Lait Bio	NULL	192.0	4.8	3.2
Choco my break	NULL	264.0	10.0	2.5
Lait bio	Bio,Carrefour	192.0	4.8	3.2
Soja cuisine bio	LSDH, Sans marque	690.0	1.8	2.7
Lait écrémé	C'est qui le patron ?!, La marque du Consommateur	142.0	4.8	3.2

only showing top 20 rows

4. Etape Gold :

La phase Gold correspond à la production des tables analytiques finales. À ce stade du projet, nous avons travaillé à partir du dataset Silver, déjà nettoyé et typé, afin de répondre aux trois questions définies dans notre fichier de configuration. Cette étape représente la finalité du pipeline, car elle transforme les données préparées en résultats directement exploitables pour l'analyse.

Pour la première question, nous avons calculé le nombre de produits créés par jour. Cette analyse permet d'observer l'évolution temporelle du dataset et de comprendre la dynamique de création des fiches produits dans OpenFoodFacts. Nous avons regroupé les données par date de création, puis compté le nombre d'enregistrements associés à chaque journée. Le résultat a été écrit dans le dossier Gold, dans la section correspondant à la question Q1.

day	product_count
2022-03-23	1656
2021-01-29	1166
2025-06-25	1333
2018-03-22	964
2020-12-14	698
2020-11-21	1154
2023-08-08	617
2022-11-18	1231
2023-10-01	531
2023-08-29	488

Pour la deuxième question, nous avons identifié les marques les plus représentées dans le dataset. Cette analyse met en évidence les marques dominantes dans la base OpenFoodFacts et permet de mieux comprendre la distribution des produits par fabricant. Nous avons compté le nombre de produits par marque, puis trié les résultats afin de faire ressortir les marques les plus fréquentes. Le résultat a été enregistré dans le dossier Gold, dans la section Q2.

brands	product_count
salutaris	1
Erber Orangenlikör	1
cumbre fresca	1
Eco-Dent	1
Lotus Brands Inc.	1
Trader jo's	1
Vianense	1
True north seafood company	1
The Father' Table	1
Private Collections	1

Pour la troisième question, nous avons étudié la distribution des NutriScores. Cette analyse offre une vision globale de la qualité nutritionnelle des produits présents dans le dataset. Nous avons regroupé les produits par catégorie de NutriScore, puis compté le nombre d'occurrences pour chaque lettre. Le résultat a été écrit dans le dossier Gold, dans la section Q3.

nutriscore_grade	count
NULL	630
a	187454
b	152017
c	270612
d	331393
e	369961
not-applicable	89721
unknown	2507878

Cette phase Gold conclut la partie analytique du pipeline et démontre la capacité de Spark à produire des résultats structurés et exploitables à partir d'un dataset volumineux et hétérogène.

5. Optimisation du pipeline :

Dans la dernière partie du projet, nous avons comparé deux versions du pipeline : une version baseline et une version optimisée. L'objectif était d'observer l'impact des transformations appliquées sur le plan d'exécution Spark et, plus largement, sur les performances du traitement.

Pour cela, nous avons d'abord exécuté la version baseline du calcul du nombre de produits créés par jour. Nous avons extrait le plan d'exécution généré par Spark et l'avons sauvegardé dans un fichier texte afin de pouvoir l'analyser. Ce plan reflète l'ensemble des opérations effectuées par Spark, notamment la lecture du dataset Silver complet, la création de la colonne de date et l'agrégation par jour.

Nous avons ensuite construit une version optimisée du même calcul. Dans cette version, nous avons réduit le DataFrame aux seules colonnes nécessaires, ce qui permet à Spark de limiter la quantité de données lues et traitées. Cette optimisation se traduit par un plan d'exécution plus léger, avec moins d'opérations intermédiaires et une réduction du volume de données manipulées. Comme pour la version baseline, nous avons sauvegardé le plan optimisé dans un fichier texte afin de pouvoir comparer les deux versions.

L'image ci-dessous illustre le plan baseline, qui montre un enchaînement plus lourd d'opérations.

```
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[day#2435], functions=[count(1)], output=[day#2435, product_count#2436L])
   +- Exchange hashpartitioning(day#2435, 200), ENSURE_REQUIREMENTS, [plan_id=558]
      +- HashAggregate(keys=[day#2435], functions=[partial_count(1)], output=[day#2435, count#2655L])
         +- Project [cast(cast(from_unixtime(cast(created_t#686 as bigint), yyyy-MM-dd HH:mm:ss, Some(Europe/Paris)) as
timestamp) as date) AS day#2435]
            +- Filter atleastnnonnulls(1, product_name#693)
               +- FileScan csv [created_t#686,product_name#693] Batched: false, DataFilters: [atleastnnonnulls(1,
```

L'image suivante illustre le plan optimisé, dans lequel Spark applique des optimisations telles que la projection anticipée, la réduction des scans et une meilleure gestion des colonnes utilisées.

```
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[day#3100], functions=[count(1)], output=[day#3100, product_count#3101L])
   +- Exchange hashpartitioning(day#3100, 200), ENSURE_REQUIREMENTS, [plan_id=649]
      +- HashAggregate(keys=[day#3100], functions=[partial_count(1)], output=[day#3100, count#3106L])
         +- Project [cast(cast(from_unixtime(cast(created_t#686 as bigint), yyyy-MM-dd HH:mm:ss, Some(Europe/Paris)) as
timestamp) as date) AS day#3100]
            +- Filter atleastnnonnulls(1, product_name#693)
               +- FileScan csv [created_t#686,product_name#693] Batched: false, DataFilters: [atleastnnonnulls(1,
product_name#693)], Format: CSV, Location: InMemoryFileIndex(1 paths)
```

Cette comparaison met en évidence l'importance de sélectionner uniquement les colonnes nécessaires avant d'effectuer des opérations d'agrégation. En réduisant la taille du DataFrame en entrée, nous diminuons le coût des lectures, des shuffles et des opérations intermédiaires. Cette étape nous a permis de mieux comprendre le fonctionnement interne de Spark et l'impact concret des optimisations sur les performances d'un pipeline distribué.

6. Conclusion :

Ce projet nous a permis de construire un pipeline complet avec Spark, depuis l'ingestion brute jusqu'à la production de tables analytiques. Le passage du Bronze au Silver a montré l'importance du nettoyage et du typage pour obtenir des données fiables, tandis que le niveau Gold a permis de produire des indicateurs clairs répondant aux besoins d'analyse. La comparaison entre le plan baseline et le plan optimisé a mis en évidence l'impact réel des choix de transformation sur les performances du pipeline. Au final, ce travail nous a donné une vision concrète des bonnes pratiques en data engineering et du rôle central de l'optimisation dans les traitements distribués.