# Traitement des données massives avec Hive

ISSA BABAN CHAWAÏ, ABDOULAYE

# Table des matières

# 1 Présentation du projet

Nous avons voulu dans le cadre de ce projet, utiliser Apache HIVE qui est un Framework pour l'entreposage de données sur Hadoop. Hive permet d'explorer, traiter, tirer des informations des nos données. Hive a été créé pour permettre aux analystes de données possédant de solides compétences en SQL (avec peu de compétences en programmation Java) d'exécuter des requêtes sur d'énormes volumes de données que Facebook stockait dans HDFS. Hive est donc en quelque sorte un moteur SQL sur Hadoop. Pour ce projet, On s'est donné pour objectif l'application des connaissances acquises lors de notre lecture du livre "Hadoop The Definitive Guide, 4th Edition" sur le chapitre 17 Hive. Les données utilisées pour mener à bien ce projet ont été téléchargé du site : (https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.pagesousformatparquet).

Pour analyser ces données on a utilisé deux canevas ou approches :

— L'utilisation du Framework Apache Spark d'une part car nous permettant une manipulation plus facile des données sous format parquet.

— L'utilisation de HDFS sur Cloudera d'autre part car nous permet d'effectuer nos commande en console ainsi on était en contact direct avec notre interface HDFS.

Avant de débuter la mise en œuvre de notre projet, nous avons pris le temps de définir et d'expliquer plusieurs concepts clés, par la suite nous avons structuré notre projet autour d'un ensemble de tâches bien définies :

— Le shell Hive (Coquille Hive)

— Fonctionnement de Hive

— Comparaison de Hive avec les bases de données traditionnelles

— Optimisation et partionnement.

— Création d'une table,

— Formats de stockage,

— Requête sur les données, filtrage et aggrégation sur les données,

— Partitions et Buckets,

Il faut souligner que l'utilisation de Hive présente plusieurs avantages. Étant donné que Hive utilise l'approche de schéma à la lecture, il est flexible et permet de modifier la définition d'une table après sa création. Hive permet à cet effet de modifier la définition des colonnes, d'ajouter de nouvelles colonnes ou même de remplacer toutes les colonnes existantes d'une table par un nouvel ensemble. Avec Hive, on a la possibilité de charger en mémoire une table si cette dernière est suffisamment petite, ceci nous permettra d'avoir un gain considérable de performance lors des requêtes.

# 2 Dataset Information

L'ensemble de données comprend des informations sur les assurés, telles que : le nom, l'âge, le zip code, le numéro de police, etc.

## 2.1 Exemple de données

Un exemple de nos données se présente sous la forme suivante :

## 2.2 Les colonnes de nos données :

Nos données sont représentées un ensemble de 17 colonnes.

— **VendorID** : Identifiant unique représentant l'opérateur du taxi.

— **tpep_pickup_datetime** : Horodatage du moment où le passager monte dans le taxi.

— **tpep_dropoff_datetime** : Horodatage du moment où le passager descend du taxi.

— **passenger_count** : Nombre de passagers dans le taxi.

— **trip_distance** : Distance totale parcourue en miles

— **RatecodeID** : Code de tarif appliqué au trajet (par exemple, tarif standard, tarif aéroport, etc.).

— **store_and_fwd_flag** : Indicateur indiquant si le trajet a été mis en attente avant d'être envoyé (en cas de connexion défaillante).

| Vendor ID | Pickup DateTime | Dropoff DateTime | Pass. Count | Distance (mi) | Rate ID | Fwd Flag | PU Loc | DO Loc | Payment Type | Fare ($) | Extra ($) | Tax ($) | Tip ($) | Tolls ($) | Improv. ($) | Total ($) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 2023-01-01 00 :32 :10 | 2023-01-01 00 :40 :36 | 1.0 | 0.97 | 1.0 | N | 161 | 141 | 2 | 9.3 | 1.0 | 0.5 | 0.0 | 0.0 | 1.0 | 14.3 |
| 2 | 2023-01-01 00 :55 :08 | 2023-01-01 01 :01 :27 | 1.0 | 1.1 | 1.0 | N | 43 | 237 | 1 | 7.9 | 1.0 | 0.5 | 4.0 | 0.0 | 1.0 | 16.9 |
| 2 | 2023-01-01 00 :25 :04 | 2023-01-01 00 :37 :49 | 1.0 | 2.51 | 1.0 | N | 48 | 238 | 1 | 14.9 | 1.0 | 0.5 | 15.0 | 0.0 | 1.0 | 34.9 |
| 1 | 2023-01-01 00 :03 :48 | 2023-01-01 00 :13 :25 | 0.0 | 1.9 | 1.0 | N | 138 | 7 | 1 | 12.1 | 7.25 | 0.5 | 0.0 | 0.0 | 1.0 | 20.85 |
| 2 | 2023-01-01 00 :10 :29 | 2023-01-01 00 :21 :19 | 1.0 | 1.43 | 1.0 | N | 107 | 79 | 1 | 11.4 | 1.0 | 0.5 | 3.28 | 0.0 | 1.0 | 19.68 |
| 2 | 2023-01-01 00 :50 :34 | 2023-01-01 01 :02 :52 | 1.0 | 1.84 | 1.0 | N | 161 | 137 | 1 | 12.8 | 1.0 | 0.5 | 10.0 | 0.0 | 1.0 | 27.8 |
| 2 | 2023-01-01 00 :09 :22 | 2023-01-01 00 :19 :49 | 1.0 | 1.66 | 1.0 | N | 239 | 143 | 1 | 12.1 | 1.0 | 0.5 | 3.42 | 0.0 | 1.0 | 20.52 |
| 2 | 2023-01-01 00 :27 :12 | 2023-01-01 00 :49 :56 | 1.0 | 11.7 | 1.0 | N | 142 | 200 | 1 | 45.7 | 1.0 | 0.5 | 10.74 | 3.0 | 1.0 | 64.44 |
| 2 | 2023-01-01 00 :21 :44 | 2023-01-01 00 :36 :40 | 1.0 | 2.95 | 1.0 | N | 164 | 236 | 1 | 17.7 | 1.0 | 0.5 | 5.68 | 0.0 | 1.0 | 28.38 |
| 2 | 2023-01-01 00 :39 :42 | 2023-01-01 00 :50 :36 | 1.0 | 3.01 | 1.0 | N | 141 | 107 | 2 | 14.9 | 1.0 | 0.5 | 0.0 | 0.0 | 1.0 | 19.9 |

TABLE 1 – Données de trajets de taxi

— **PULocationID** : Identifiant de la station de prise en charge
— **DOLocationID** : Identifiant de la station de dépôt
— **payment_type** : Type de paiement effectué.
— **fare_amount** : Montant du tarif de base pour le trajet (avant les frais supplémentaires et les taxes).
— **extra** : Frais supplémentaires pour le trajet, comme les frais de bagages ou de surcharge.
— **mta_tax** : Taxe appliquée par la Metropolitan Transportation Authority (MTA) dans la région de New York.
— **tip_amount** : Montant du pourboire donné.
— **tolls_amount** : Montant total des péages payés durant le trajet.
— **improvement_surcharge** : Frais supplémentaires pour l'amélioration des infrastructures de transport
— **total_amount** : Montant total à payer pour le trajet, y compris le tarif, les taxes, les frais et le pourboire.

# 3 Bibliothèques utilisées :

Pour réaliser ce projet, nous utiliserons le langage de programmation python essentiellement associé aux languages SQL et SparkSQL.

## 3.1 Les Librairies installées

Dans le cadre de ce projet plusieurs librairies ont été installées entre autre :
• **pyspark** :Cette librairie est une interface Python pour Apache Spark. Elle permet un traitement distribué de grandes quantités de données, une opération sur les DataFrames à grande échelle, d'effectuer des requêtes SQL distribuées, machine learning (MLlib), streaming

• **findspark** :Cette librairie est utilisée pour localiser Spark dans notre environnement Python .Elle permet d'initialiser PySpark dans des notebooks Jupyter ou des scripts Python. Elle permet par la même occasion de résoudre les problèmes de configuration du SPARK_HOME

• **delta-spark** : Elle permet une implémentation de Delta Lake pour PySpark. Elle utile pour la création des tables ACID sur data lake, accès aux versions historiques de la table, avoir une idée sur l'évolution du schéma de notre table et enfin elle permet d'effectuer les opérations telles que : MERGE, UPDATE, DELETE sur des fichiers Parquet

• **numpy** : C'est une librairie fondamentale pour le calcul numérique en Python. Elle permet de définir les arrays multidimensionnels performants, les fonctions mathématiques et statistiques tilisé dans les UDF, conversions de données ...

## 3.2 Librairies Python standard

• **os** : Cette librairie nous permet d'interargir le système d'exploitation. Elle nous donne la possibilité de Vérifier si un chemin existe, créer des répertoires ou de supprimer des fichiers.

• **matplotlib** : Cette librairie nous permet de créer des visualisations graphiques afin de mieux comprendre nos données tels que : les graphiques, les histogrammes, les nuages de points...

## 3.3 PySpark Core

• **pyspark.sql.SparkSession** : Cette librairie représente le point d'entrée principal pour interagir avec Spark. Elle permet de entre autre de lire/écrire des données, exécuter des requêtes SQL, créer des DataFrames.

## 3.4 PySpark Types (pyspark.sql.types)

• **StructType & StructField** : Elle permet de définir le schéma d'un DataFrame.

• **StringType** : Avec cette librairie, nous définissons les chaînes de caractères

- **IntegerType** : Elle nous permet de définir les nombres entiers
- **DateType** : Elle nous permet de définir le format des dates.
- **DoubleType** : Elle nous permet de définir les nombres à virgule flottante.
- **LongType** : Elle nous permet de définir les valeurs très grandes nécessitant une large plage de valeurs.
- **TimestampType** : Utilisé pour stocker des valeurs de date et heure à une précision milliseconde ou microseconde.

### 3.5 PySpark Functions (pyspark.sql.functions)

pyspark.sql.functions est un module dans PySpark qui regroupe une collection de fonctions utiles permettant de manipuler des données dans un DataFrame Spark.

- **year()** : Elle nous donne la possibilité d'extrait l'année d'une colonne de type TimestampType ou DateType.
- **month()** : Elle nous donne la possibilité de d'extraire le mois d'une colonne de type TimestampType ou DateType.
- **dayofmonth()** : Elle nous donne la possibilité de d'extraire le jour du mois (1 à 31) d'une colonne de type TimestampType ou DateType.
- **dayofmonth()** : Elle nous donne la possibilité de d'extraire le jour du mois (1 à 12) d'une colonne de type TimestampType ou DateType.
- **hour()** : Elle nous donne la possibilité de d'extraire le l'heure (1 à 23) d'une colonne de type TimestampType ou DateType.

### 3.6 Pandas

- **pandas (pd)** : Utilisée pour manipuler des données en mémoire (DataFrames Pandas). Elle permet de convertir des données entre Spark DataFrames et Pandas DataFrames. Elle permet ainsi une analyses exploratoires rapides sur des petits datasets.

## 4 Création et configuration de la session Spark :

Nous définissons successivement : un nom à notre application, le répertoire où Spark stockera nos tables Hive, on précise que Spark doit utiliser Hive comme moteur de gestion de métadonnées pour les tables, On désactive le lecteur vectorisé pour les fichiers Parquet pour éviter tout problème qui peut en découler, Les options qui permettent de gérer la gestion des dates dans les fichiers Parquet car des problèmes peuvent survenir si les dates sont réajustées de manière incorrecte dans le format parquet, on active le support pour Hive, qui permettra à Spark de lire et d'écrire des tables Hive directement.

```
1     spark = SparkSession.builder \
2     .appName("Analyses des données de NYC Taxi") \
3     .config("spark.sql.warehouse.dir", "/opt/workspace/hive-warehouse") \
4     .config("spark.sql.catalogImplementation", "hive") \
5     .config("spark.sql.parquet.enableVectorizedReader", "false") \
6     .config("spark.sql.parquet.int96RebaseModeInRead", "CORRECTED") \
7     .config("spark.sql.parquet.int96RebaseModeInWrite", "CORRECTED") \
8     .enableHiveSupport() \
9     .getOrCreate()
10
```

# 5 INGESTION DES DONNÉES ET VALIDATION DU SCHEMA :

## 5.1 Ingestion des données

Pour charger les données, nous devons spécifier la source de nos données, le schéma à utiliser et le format dans lequel se trouvent nos données.

```
df = spark.read \
    .schema(schema) \
    .parquet("/opt/workspace/data/nycdriver/yellowtrip/*.parquet")
```

## 5.2 Validation du schema

Nous avons spécifié un schéma à appliquer sur les données CSV. Ceci nous donne une options pour gérer les données corrompues. Certaines colonnes ont été 'castées' (cast) pour suivre un format précis (exemple les dates). Malgré le fait que nous ayons défini les dates en string sur notre schéma, on a décidé de les caster en datatype plus tard afin de mieux les manipuler.

```
schema = StructType([
    StructField("VendorID", LongType(), True),
    StructField("tpep_pickup_datetime", TimestampType(), True),
    StructField("tpep_dropoff_datetime", TimestampType(), True),
    StructField("passenger_count", DoubleType(), True),
    StructField("trip_distance", DoubleType(), True),
    StructField("RatecodeID", DoubleType(), True),
    StructField("store_and_fwd_flag", StringType(), True),
    StructField("PULocationID", LongType(), True),
    StructField("DOLocationID", LongType(), True),
    StructField("payment_type", LongType(), True),
    StructField("fare_amount", DoubleType(), True),
    StructField("extra", DoubleType(), True),
    StructField("mta_tax", DoubleType(), True),
    StructField("tip_amount", DoubleType(), True),
    StructField("tolls_amount", DoubleType(), True),
    StructField("improvement_surcharge", DoubleType(), True),
    StructField("total_amount", DoubleType(), True),
    StructField("congestion_surcharge", DoubleType(), True),
    StructField("airport_fee", DoubleType(), True)
])

```

## 5.3 schéma de données

Nous pouvons avoir une description des colonnes du schéma de données associé à notre dataFrame :

```
Schéma:
root
 |-- VendorID: long (nullable = true)
 |-- tpep_pickup_datetime: timestamp (nullable = true)
 |-- tpep_dropoff_datetime: timestamp (nullable = true)
 |-- passenger_count: double (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- RatecodeID: double (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- PULocationID: long (nullable = true)
 |-- DOLocationID: long (nullable = true)
 |-- payment_type: long (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- extra: double (nullable = true)
 |-- mta_tax: double (nullable = true)
 |-- tip_amount: double (nullable = true)
 |-- tolls_amount: double (nullable = true)
 |-- improvement_surcharge: double (nullable = true)
 |-- total_amount: double (nullable = true)
```

```
 |-- congestion_surcharge: double (nullable = true)
 |-- airport_fee: double (nullable = true)

```

# 6 CRÉER UNE TABLE

## 6.1 Table normale

```
    CREATE TABLE locations (
    LocationID BIGINT,
    Borough STRING, --arrondissement
    Zone STRING,
    service_zone STRING
);
```

## 6.2 Table Externe

Une table externe se comporte différemment d'une table normale. Dans une table externe, vous contrôlez la création et la suppression des données. L'emplacement des données externes est spécifié lors de la création de la table. Grâce au mot-clé EXTERNAL, Hive sait qu'il ne gère pas les données et ne les déplace donc pas vers son répertoire d'entrepôt. En effet, il ne vérifie même pas l'existence de l'emplacement externe au moment de sa définition. Cette fonctionnalité est utile car elle permet de créer les données en différé après la création de la table. Lorsque vous supprimez une table externe, Hive conserve les données intactes et ne supprime que les métadonnées.

```
CREATE EXTERNAL TABLE nyc_yellow_taxi (
    VendorID BIGINT,
    tpep_pickup_datetime BIGINT,      -- C'est un timestamp en microsecondes
    tpep_dropoff_datetime BIGINT,
    passenger_count BIGINT,
    trip_distance DOUBLE,
    RatecodeID BIGINT,
    store_and_fwd_flag STRING,
    PULocationID BIGINT,
    DOLocationID BIGINT,
    payment_type BIGINT,
    fare_amount DOUBLE,
    extra DOUBLE,
    mta_tax DOUBLE,
    tip_amount DOUBLE,
    tolls_amount DOUBLE,
    improvement_surcharge DOUBLE,
    total_amount DOUBLE,
    congestion_surcharge DOUBLE,
    airport_fee DOUBLE
)
STORED AS PARQUET
LOCATION '/user/hive/taxi_data/';
```

La différence des tables réside dans leurs utilisation : si nous voulons effectuer tout notre traitement avec Hive, utiliser des tables normales est plus approprié, mais si nous souhaitons utiliser Hive et d'autres outils sur le même ensemble de données, utiliser les tables externes semble être le meilleur choix.

## 6.3 Supprimer la table

```
DROP TABLE IF EXISTS nyc_yellow_taxi;
);
```

### 6.4 Tester ou lire la table

```
SELECT COUNT(*) FROM nyc_yellow_taxi;
);
```

| _c0 |
|---|
| 3066766 |

Table 2 – Données de notre Table

### 6.5 Structure hive warehouse

Après la création de nos tables, Hive crée un système de stockage qu'on appelle le Hive Warehouse qui est le répertoire principal de stockage des données pour Apache Hive, généralement situé à /user/hive/warehouse dans HDFS. Cependant, certains des répertoires et sous-répertoires sont crées automatiquement selon les colonnes de partition, les fichiers de données des tables, des tables partitionnées, ce qui permet d'optimiser les performances des requêtes en ne lisant que les partitions nécessaires. Lorsqu'une table gérée est supprimée avec la commande DROP TABLE, Hive supprime également les données physiques du warehouse, contrairement aux tables externes où seules les métadonnées sont supprimées et les fichiers de données restent intacts.La structure de notre hive warehouse se présente comme suit :

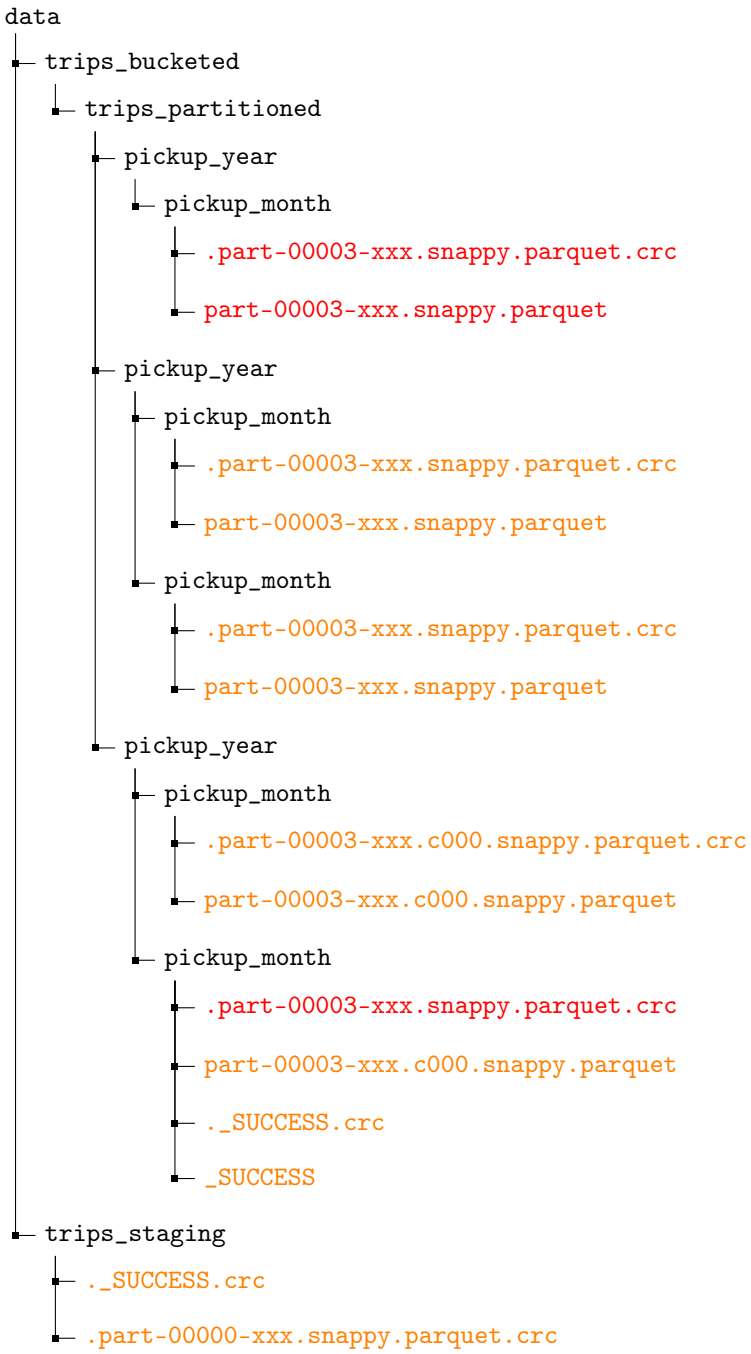


FIGURE 1 – Structure du répertoire de données

# 7 FORMATAGE, NETTOYAGE, PARTITIONNEMENT, INDEXATION, BUCKETISATION DE NOS DONNÉES ET OPTIMISATION :

## 7.1 Formatages des colonnes des données

Il se peut qu'il existe des colonnes qui ne soient pas optimales pour l'exécution sur nos bases de données et qui de ce fait doivent être restructurées. C'est ainsi le cas des colonnes tpep_pickup_datetime et tpep_dropoff_datetime :

```
1    FROM_UNIXTIME(CAST(tpep_pickup_datetime/1000000 AS BIGINT)) as pickup_time,
2    FROM_UNIXTIME(CAST(tpep_dropoff_datetime/1000000 AS BIGINT)) as dropoff_time,
```

Ainsi, nous pouvons, si nécessaire ou requis, extraire les valeurs correspondantes aux années, mois, heures, minutes, etc. Nous avons à la fin de cette manipulation un tableau comme celui-ci-dessous plus lisible.

| vendorid | pickup_time | dropoff_time | passenger_count | trip_distance | fare_amount | total_amount |
|---|---|---|---|---|---|---|
| 2 | 2022-12-31 16 :32 :10 | 2022-12-31 16 :40 :36 | 1.0 | 0.97 | 9.3 | 14.3 |
| 2 | 2022-12-31 16 :55 :08 | 2022-12-31 17 :01 :27 | 1.0 | 1.1 | 7.9 | 16.9 |
| 2 | 2022-12-31 16 :25 :04 | 2022-12-31 16 :37 :49 | 1.0 | 2.51 | 14.9 | 34.9 |
| 1 | 2022-12-31 16 :03 :48 | 2022-12-31 16 :13 :25 | 0.0 | 1.9 | 12.1 | 20.85 |
| 2 | 2022-12-31 16 :10 :29 | 2022-12-31 16 :21 :19 | 1.0 | 1.43 | 11.4 | 19.68 |

TABLE 3 – Données de trajets de taxi

## 7.2 Partitionnement

Hive organise les tables en partitions, une méthode permettant de diviser une table en parties grossières selon la valeur d'une colonne de partition, comme une date. L'utilisation de partitions permet d'accélérer les requêtes sur des tranches de données.

Prenons un exemple d'utilisation courante des partitions : imaginez des fichiers journaux dont chaque enregistrement est horodaté. Si nous partitionnons par date, les enregistrements correspondant à la même date seront stockés dans la même partition. L'avantage de ce schéma est que les requêtes restreintes à une date ou à un ensemble de dates spécifiques sont beaucoup plus efficaces, car elles n'analysent que les fichiers des partitions concernées.

Prenons nos données et partitionons par année et mois. les enregistrements correspondant à la même année et mois seront stockés dans la même partition. L'avantage de ce schéma est que les requêtes restreintes à une année/mois ou à un ensemble d'années/mois spécifiques sont beaucoup plus efficaces, car elles n'analysent que les fichiers des partitions concernées.

```
# Sauvegarder avec partitionnement par année et mois
df_partitioned.write.mode("overwrite") \
    .partitionBy("pickup_year", "pickup_month") \
    .format("parquet") \
    .saveAsTable("nyc_taxi.trips_partitioned")
```

Cependant, il est à noter qu'une table peut être partitionnée en plusieurs dimensions. Par exemple, outre le partitionnement des journaux (logs) par date, nous pouvons également sous-partitionner chaque partition de date par heure afin de permettre des requêtes encore plus efficaces.

Les tables ou partitions peuvent être subdivisées en compartiments afin de structurer les données et d'optimiser les requêtes. Par exemple, le découpage par identifiant utilisateur permet d'évaluer rapidement une requête basée sur un utilisateur en l'exécutant sur un échantillon aléatoire de l'ensemble des utilisateurs.

### 7.2.1 Création d'une table partitionnée

Une table partitionnée peut être crée comme suit :

```
DROP TABLE IF EXISTS nyc_taxi_partitioned;

CREATE TABLE nyc_taxi_partitioned (
    VendorID BIGINT,
    pickup_datetime TIMESTAMP,          -- Converti en TIMESTAMP
    dropoff_datetime TIMESTAMP,
    passenger_count DOUBLE,
    trip_distance DOUBLE,
    PULocationID BIGINT,
    DOLocationID BIGINT,
    payment_type BIGINT,
    fare_amount DOUBLE,
    tip_amount DOUBLE,
    total_amount DOUBLE
)
PARTITIONED BY (year INT, month INT)
STORED AS ORC;
```

```
-- Voir les partitions
SHOW PARTITIONS nyc_taxi_partitioned;
```

| partition |
|---|
| year=2008/month=12 |
| year=2022/month=10 |
| year=2022/month=12 |
| year=2023/month=1 |

Table 4 – Partitions

#### 7.2.2 Exemple de requête sur une table partitionnée

```
SELECT year, month, COUNT(*) as num_trips
FROM nyc_taxi_partitioned
GROUP BY year, month
ORDER BY year, month;
```

On obtient le résultat suivant :

| year | month | num_trips |
|---|---|---|
| 2008 | 12 | 2 |
| 2022 | 10 | 11 |
| 2022 | 12 | 25689 |
| 2023 | 1 | 3041064 |

Table 5 – Nombre de trajets par année et mois

### 7.3 Buckets

Il existe deux raisons d'organiser vos tables (ou partitions) en buckets. La première est de permettre des requêtes plus efficaces. La seconde est d'optimiser l'échantillonnage.

```
spark.sql("""
    CREATE TABLE IF NOT EXISTS nyc_taxi.trips_bucketed (
        VendorID INT,
        tpep_pickup_datetime TIMESTAMP,
        tpep_dropoff_datetime TIMESTAMP,
        passenger_count INT,
        trip_distance DOUBLE,
        fare_amount DOUBLE,
        tip_amount DOUBLE,
        total_amount DOUBLE,
        PULocationID INT,
        DOLocationID INT
    )
    CLUSTERED BY (PULocationID) INTO 32 BUCKETS
    STORED AS PARQUET
""")
```

### 7.4 Indexes

Pour améliorer l'efficacité et optimiser certaines requêtes afin d'obtenir un gain de temps, nous avons utilisé des index. Nous avons utilisé l'index Bitmap qui est un type spécifique d'index qui est utile pour les colonnes ayant un nombre limité de valeurs distinctes. On l'utilise particulièrement pour des colonnes avec des valeurs catégorielles comme "État" ou "Genre". Cependant, dans notre cas, à titre d'exemple, nous l'avons utilisé sur la colonne payment_type.

```
CREATE TABLE nyc_taxi_extended (
    VendorID BIGINT,
    pickup_datetime TIMESTAMP,
    dropoff_datetime TIMESTAMP,
    passenger_count DOUBLE,
```

```
    trip_distance DOUBLE,
    PULocationID BIGINT,
    DOLocationID BIGINT,
    payment_type BIGINT,
    fare_amount DOUBLE,
    tip_amount DOUBLE,
    total_amount DOUBLE
)
STORED AS PARQUET
LOCATION '/user/hive/taxi_data/';

LOAD LOCAL DATA INPATH '/home/cloudera/Desktop/yellow_tripdata_2023-01.parquet'
OVERWRITE INTO TABLE nyc_taxi_extended;

DROP INDEX IF EXISTS nyc_taxi_index ON nyc_taxi_extended;
CREATE INDEX nyc_taxi_index
ON TABLE nyc_taxi_extended (payment_type)
AS 'BITMAP' WITH DEFERRED REBUILD;
ALTER INDEX nyc_taxi_index ON nyc_taxi_extended REBUILD;
```

## 7.5 Formats de stockage

Deux dimensions régissent le stockage des tables dans Hive : le format de ligne et le format de fichier. Le format de ligne détermine la manière dont les lignes et les champs d'une ligne donnée sont stockés. Les formats orientés colonnes fonctionnent bien lorsque les requêtes n'accèdent qu'à un petit nombre de colonnes de la table, tandis que les formats orientés lignes sont appropriés lorsqu'un grand nombre de colonnes d'une seule ligne sont nécessaires au traitement en même temps. Nos tables peuvent donc être stockées en deux catégories :

— • Lorsqu'une table est créée sans clauses ROW FORMAT ou STORED AS, le format par défaut est un texte délimité avec une ligne par ligne.

— • La catégorie peut être spécifiée en format binaire tels que les fichiers de séquence, fichiers de données Avro, fichiers Parquet, RCFile et fichiers ORC.

En exemple, on peut sauvegarder notre table en Avro :

```
SET hive.exec.compress.output=true;
SET avro.output.codec=snappy;
CREATE TABLE ... STORED AS AVRO;
```

on peut sauvegarder notre table en Parquet :

```
CREATE TABLE nyc_taxi_parquet STORED AS PARQUET
```

on peut également sauvegarder notre table en Sequencefile :

```
CREATE TABLE nyc_taxi_seqfile STORED AS SEQUENCEFILE
```

on peut également sauvegarder notre table en ORC :

```
CREATE TABLE nyc_taxi_seqfile STORED AS ORC
```

Il est à noter que nous pouvons également sauvegarder nos tables de manière compressée grâce à :

```
SET hive.exec.compress.output=true;
SET mapreduce.output.fileoutputformat.compress.codec=org.apache.hadoop.io.compress.DeflateCodec;
SET mapreduce.output.fileoutputformat.compress.type=BLOCK;
```

### 7.6 Optimisations

Pour une meilleure optimisation, nous avons configuré et contrôlé la partition dynamique lors de l'exécution de requêtes d'insertion dans des tables partitionnées.

- Hive ne crée pas de partitions automatiquement par défaut lors des insertions nous avons voulu en créer automatiquement des partitions dynamiques.
- Nous avons choisi un mode de partitionnement non stricte. Ceci nous permet une insertion de données plus flexible. On peut ainsi insérer des données dans plusieurs partitions en une seule requête.
- Nous avons autorisé Hive à créer jusqu'à 1000 partitions dynamiques lors d'une seule requête.
- Nous avons voulu qu'un nœud de Worker dans un cluster Hadoop puisse créer jusqu'à 1000 partitions dynamiques lors d'une opération d'insertion.

```
SET hive.exec.dynamic.partition=true;
SET hive.exec.dynamic.partition.mode=nonstrict;
SET hive.exec.max.dynamic.partitions=1000;
SET hive.exec.max.dynamic.partitions.pernode=1000;
```

## 8 TRANSFORMATION, FILTRAGE ET AGGRÉGATION SUR NOS DONNÉES :

### 8.1 Transformation :

Dans cette section, nous avons décidé d'ajouter un certain nombre de colonnes dérivées des clonnes existantes pour mieux appréhender nos données.

Pour ajouter une colonne nous avons utilisé la fonction suivante :

```
df = df \
    .withColumn("pickup_year", year(col("tpep_pickup_datetime"))) \
    .withColumn("pickup_month", month(col("tpep_pickup_datetime"))) \
    .withColumn("pickup_day", dayofmonth(col("tpep_pickup_datetime"))) \
    .withColumn("pickup_hour", hour(col("tpep_pickup_datetime"))) \
    .withColumn("trip_duration_minutes",
                (unix_timestamp("tpep_dropoff_datetime") -
                 unix_timestamp("tpep_pickup_datetime")) / 60) \
    .withColumn("tip_percentage",
                when(col("fare_amount") > 0,
                     (col("tip_amount") / col("fare_amount")) * 100)
                .otherwise(0))
```

ceci nous permet :

- d'extraire l'année à partir de la colonne tpep_pickup_datetime
- d'extrait le mois (de 1 à 12) à partir de la colonne tpep_pickup_datetime.
- d'extraire le jour du mois (de 1 à 31) à partir de la colonne tpep_pickup_datetime.
- d'extraire l'heure (de 0 à 23) à partir de la colonne tpep_pickup_datetime.
- de calculer la durée du trajet en minutes.
- de calculer le pourcentage du pourboire par rapport au montant du tarif pour chaque trajet

Au total, nous avons ajouté 7 nouvelles colonnes. Il s'agit entre autre des colonnes :

### 8.2 Filtrage des données :

Le Filtrage dans notre cas de figure se fait essentiellement dans la clause where ou encore en présisant une condition tels que :

```
WHERE passenger_count > 0 AND passenger_count <= 6
ORDER BY trip_count DESC
```

# 9 MONITORING ET OBSERVABILITÉ :

Ici, nous allons essayer de produire ou de faire des requêtes sur nos données.

## 9.1 Revenues totaux selon l'année et le mois

Sur nos données partitionnées en pickup_year et pickup_month, nous voulons ressortir quelques informations tels que :

- •avg_distance_miles : la distance moyenne en miles
- •avg_fare : le prix moyen d'une course
- •avg_tip : le pourboire moyen
- •total_revenue : revenues totaux.
- •pickup_year : Année de ramassage du client
- •pickup_month : Mois de ramassage du client

Ces informations seront groupées en pickup_year et pickup_month par ordre croissant.

```
SELECT
    pickup_year,
    pickup_month,
    COUNT(*) as total_trips,
    ROUND(AVG(trip_distance), 2) as avg_distance_miles,
    ROUND(AVG(fare_amount), 2) as avg_fare,
    ROUND(AVG(tip_amount), 2) as avg_tip,
    ROUND(SUM(total_amount), 2) as total_revenue
FROM nyc_taxi.trips_partitioned
GROUP BY pickup_year, pickup_month
ORDER BY pickup_year, pickup_month
```

| pickup_year | pickup_month | total_trips | avg_distance_miles | avg_fare | avg_tip | total_revenue |
|---|---|---|---|---|---|---|
| 2008 | 12 | 2 | 8.88 | 35.0 | 0.0 | 80.55 |
| 2022 | 10 | 11 | 0.98 | 59.91 | 8.2 | 760.49 |
| 2022 | 12 | 25 | 3.31 | 16.63 | 4.28 | 658.23 |
| 2023 | 1 | 3066718 | 3.85 | 18.37 | 3.37 | 8.286337914E7 |
| 2023 | 2 | 10 | 4.57 | 22.02 | 3.88 | 313.81 |

TABLE 6 – Statistiques des trajets par mois

## 9.2 Information sur les trajets parcourus par heure de la journée

Nous voulons ressortir quelques informations tels que :

- •pickup_hour : l'heure où le passager monte dans le taxi.
- •avg_fare : le prix moyen de la course
- •avg_distance : la distance moyenne de la course
- •trip_count : le nombre de trajet.

```
SELECT
    pickup_hour,
    COUNT(*) as trip_count,
    ROUND(AVG(fare_amount), 2) as avg_fare,
    ROUND(AVG(trip_distance), 2) as avg_distance
FROM nyc_taxi.trips_partitioned
GROUP BY pickup_hour
ORDER BY pickup_hour
```

| pickup_hour | trip_count | avg_fare | avg_distance |
|---|---|---|---|
| 0 | 84969 | 19.59 | 4.02 |
| 1 | 59799 | 17.75 | 3.49 |
| 2 | 42040 | 16.73 | 3.2 |
| 3 | 27438 | 17.67 | 3.74 |
| 4 | 17835 | 22.0 | 4.77 |
| 5 | 18011 | 26.16 | 15.3 |
| 6 | 43860 | 21.95 | 5.42 |
| 7 | 86877 | 18.88 | 5.25 |
| 8 | 116865 | 17.43 | 5.52 |
| 9 | 131111 | 17.49 | 3.12 |
| 10 | 143666 | 17.54 | 3.09 |
| 11 | 154157 | 17.24 | 3.21 |
| 12 | 169858 | 17.6 | 3.13 |
| 13 | 178739 | 18.27 | 3.23 |
| 14 | 191604 | 19.49 | 3.56 |
| 15 | 196424 | 19.08 | 3.67 |
| 16 | 195977 | 19.3 | 3.47 |
| 17 | 209493 | 18.46 | 4.48 |
| 18 | 215889 | 16.87 | 3.96 |
| 19 | 192801 | 17.43 | 3.8 |
| 20 | 165862 | 17.8 | 3.46 |
| 21 | 161548 | 18.27 | 3.88 |
| 22 | 147415 | 19.15 | 3.83 |
| 23 | 114528 | 20.31 | 4.23 |

TABLE 7 – Statistiques des trajets par heure de la journée

## 9.3 Analyse des pourboires par type de paiement

Nous voulons analyser certaines informations selon le type de payement :

— •trip_count : le nombre de trajet.

— •avg_tip : le pourboire moyen.

— •avg_tip_pct : le pourcentage moyen de pourboire.

— •total_tips : le montant total de pourboire.

```
SELECT
    payment_type,
    COUNT(*) as trip_count,
    ROUND(AVG(tip_amount), 2) as avg_tip,
    ROUND(AVG(tip_percentage), 2) as avg_tip_pct,
    ROUND(SUM(tip_amount), 2) as total_tips
FROM nyc_taxi.trips_partitioned
```

| payment_type | trip_count | avg_tip | avg_tip_pct | total_tips |
|---|---|---|---|---|
| 1 | 2411185 | 4.17 | 26.25 | 1.005718249E7 |
| 2 | 526058 | 0.0 | 0.0 | 177.94 |
| 0 | 71677 | 3.73 | 18.65 | 267611.72 |
| 4 | 18366 | 0.02 | 0.23 | 409.12 |
| 3 | 13321 | 0.01 | 0.3 | 140.89 |

TABLE 8 – Statistiques des pourboires par type de paiement

## 9.4 Les informations selon le type de trajets

Certaines informations seront analysées parmi lesquelles :

— •distance_category : classification des trajets selon la distance parcourue.

— •trip_count : le nombre de trajets parcourus.

— •avg_tip_pct : le pourcentage moyen de pourboire.

— •avg_fare : prix moyen des trajets

— •avg_duration_min : la durée moyenne de trajet par minute

— •avg_tip_pct : le montant moyen de pourboire.

```
SELECT
    CASE
        WHEN trip_distance < 2 THEN 'Court (< 2 miles)'
        WHEN trip_distance BETWEEN 2 AND 5 THEN 'Moyen (2-5 miles)'
        WHEN trip_distance BETWEEN 5 AND 10 THEN 'Long (5-10 miles)'
        ELSE 'Très Long (> 10 miles)'
    END as distance_category,
    COUNT(*) as trip_count,
    ROUND(AVG(fare_amount), 2) as avg_fare,
    ROUND(AVG(trip_duration_minutes), 2) as avg_duration_min,
    ROUND(AVG(tip_percentage), 2) as avg_tip_pct
FROM nyc_taxi.trips_partitioned
WHERE trip_distance > 0 AND trip_duration_minutes > 0
GROUP BY
    CASE
        WHEN trip_distance < 2 THEN 'Court (< 2 miles)'
        WHEN trip_distance BETWEEN 2 AND 5 THEN 'Moyen (2-5 miles)'
        WHEN trip_distance BETWEEN 5 AND 10 THEN 'Long (5-10 miles)'
        ELSE 'Très Long (> 10 miles)'
    END
ORDER BY trip_count DESC
```

| distance_category | trip_count | avg_fare | avg_duration_min | avg_tip_pct |
|---|---|---|---|---|
| Court (< 2 miles) | 1645424 | 9.48 | 9.3 | 22.76 |
| Moyen (2-5 miles) | 869999 | 17.86 | 18.02 | 18.74 |
| Long (5-10 miles) | 261121 | 32.81 | 25.41 | 16.68 |
| Très Long (> 10 miles) | 244287 | 63.59 | 40.64 | 21.58 |

Table 9 – Statistiques des trajets par catégorie de distance

## 9.5 Analyse des Zones de Pickup/Dropoff les Plus Populaires

Nous avons voulu connaitre quelles sont les zones ou la demande de taxi sont élevées :

— •pickup_location : lieu de ramassage

— •pickup_count : nombre de trajets ou de ramassage

— •avg_fare : prix moyen.

```
SELECT
    PULocationID as pickup_location,
    COUNT(*) as pickup_count,
    ROUND(AVG(fare_amount), 2) as avg_fare
FROM nyc_taxi.trips_partitioned
GROUP BY PULocationID
ORDER BY pickup_count DESC
LIMIT 10
```

## 9.6 Les zones d'arrivée les plus fréquentées

Nous avons voulu connaitre les zones ou les clients y vont le plus selon :

— •dropoff_location : Lieu de dépôt

| pickup_location | pickup_count | avg_fare |
|---|---|---|
| 132 | 160030 | 58.64 |
| 237 | 148074 | 12.24 |
| 236 | 138391 | 13.06 |
| 161 | 135417 | 15.08 |
| 186 | 109227 | 15.38 |
| 162 | 105334 | 14.75 |
| 142 | 100228 | 13.54 |
| 230 | 98991 | 17.22 |
| 138 | 89188 | 40.89 |
| 170 | 88346 | 14.75 |

Table 10 – Top 10 des lieux de prise en charge

— •dropoff_count : notre de dépôt

— •avg_fare : prix moyen

```
SELECT
    DOLocationID as dropoff_location,
    COUNT(*) as dropoff_count,
    ROUND(AVG(fare_amount), 2) as avg_fare
FROM nyc_taxi.trips_partitioned
GROUP BY DOLocationID
ORDER BY dropoff_count DESC
LIMIT 10
```

| dropoff_location | dropoff_count | avg_fare |
|---|---|---|
| 236 | 146348 | 13.05 |
| 237 | 132364 | 12.12 |
| 161 | 116149 | 14.51 |
| 230 | 89878 | 19.18 |
| 170 | 88783 | 14.43 |
| 239 | 87969 | 14.69 |
| 142 | 87969 | 13.96 |
| 141 | 87655 | 13.55 |
| 162 | 82739 | 14.9 |
| 48 | 77383 | 16.96 |

Table 11 – Top 10 des lieux de dépose

## 9.7 Revenus selon les jours de la Semaine

Nous avons essayé de tirer les informations des revenus et des trajets par jour de la semaine :

— •day_of_week : jours de la semaine

— •day_name : Nom du jour de la semaine

— •trip_count :Nombre de trajet parcouru

— •total_revenue : revenue total

— •avg_revenue_per_trip : revenue moyen par trajet

```
SELECT
    dayofweek(tpep_pickup_datetime) as day_of_week,
    CASE dayofweek(tpep_pickup_datetime)
        WHEN 1 THEN 'Dimanche'
        WHEN 2 THEN 'Lundi'
        WHEN 3 THEN 'Mardi'
```

```
        WHEN 4 THEN 'Mercredi'
        WHEN 5 THEN 'Jeudi'
        WHEN 6 THEN 'Vendredi'
        WHEN 7 THEN 'Samedi'
    END as day_name,
    COUNT(*) as trip_count,
    ROUND(SUM(total_amount), 2) as total_revenue,
    ROUND(AVG(total_amount), 2) as avg_revenue_per_trip
FROM nyc_taxi.trips_partitioned
GROUP BY dayofweek(tpep_pickup_datetime)
ORDER BY day_of_week
```

| day_of_week | day_name | trip_count | total_revenue | avg_revenue_per_trip |
|---|---|---|---|---|
| 1 | Dimanche | 436434 | 1.222993475E7 | 28.02 |
| 2 | Lundi | 404809 | 1.144494019E7 | 28.27 |
| 3 | Mardi | 490988 | 1.332750629E7 | 27.14 |
| 4 | Mercredi | 416570 | 1.108097892E7 | 26.6 |
| 5 | Jeudi | 441929 | 1.196376344E7 | 27.07 |
| 6 | Vendredi | 434381 | 1.169806544E7 | 26.93 |
| 7 | Samedi | 441655 | 1.112000319E7 | 25.18 |

TABLE 12 – Statistiques des trajets par jour de la semaine

## 9.8 Détection des trajets suspects

Nous savons tous que certains taxi peuvent augmenter les prix en mal régulant leur taximètre ou prolonger les distances pour les rendre plus longue. Nous avons voulu répertorier les informations :

— • trip_distance : les distances qui sont longues plus 150 miles ou inférieures à zéro.

— • fare_amount : les voyages dont les montants sont supérieurs a 500, ou inférieures à zéro,

```
SELECT
    tpep_pickup_datetime,
    trip_distance,
    fare_amount,
    total_amount,
    ROUND(total_amount / NULLIF(trip_distance, 0), 2) as cost_per_mile
FROM nyc_taxi.trips_partitioned
WHERE
    trip_distance > 100 OR  -- Distance anormalement longue
    fare_amount > 500 OR    -- Tarif très élevé
    fare_amount < 0 OR      -- Tarif négatif
    trip_distance < 0       -- Distance négative
ORDER BY fare_amount DESC
LIMIT 10
```

## 9.9 Analyse de Rentabilité par Heure

Quelles sont les moments les plus efficients pour conduire (ratio revenus/trajet)

```
SELECT
    pickup_hour,
    COUNT(*) as trip_count,
    ROUND(SUM(total_amount), 2) as hourly_revenue,
    ROUND(AVG(total_amount), 2) as avg_revenue_per_trip,
    ROUND(AVG(trip_duration_minutes), 2) as avg_duration,
    ROUND(SUM(total_amount) / COUNT(*), 2) as revenue_efficiency
FROM nyc_taxi.trips_partitioned
WHERE trip_duration_minutes > 0
```

| tpep_pickup_datetime | trip_distance | fare_amount | total_amount | cost_per_mile |
|---|---|---|---|---|
| 2023-01-24 12 :43 :44 | 177.88 | 1160.1 | 1169.4 | 6.57 |
| 2023-01-09 16 :17 :32 | 0.0 | 999.0 | 1000.0 | — |
| 2023-01-30 13 :17 :33 | 0.0 | 900.0 | 901.0 | — |
| 2023-01-30 13 :23 :56 | 0.0 | 750.0 | 751.0 | — |
| 2023-01-30 16 :17 :35 | 8.81 | 701.6 | 705.6 | 80.09 |
| 2023-01-04 20 :11 :27 | 103.8 | 656.8 | 667.1 | 6.43 |
| 2023-01-08 08 :15 :40 | 0.0 | 655.35 | 656.85 | — |
| 2023-01-22 23 :24 :55 | 0.0 | 650.0 | 651.0 | — |
| 2023-01-10 00 :55 :47 | 0.0 | 625.0 | 626.0 | — |
| 2023-01-26 10 :28 :15 | 0.0 | 600.0 | 602.25 | — |
| 2023-01-17 10 :04 :34 | 0.06 | 600.0 | 603.5 | 10058.33 |
| 2023-01-19 00 :20 :12 | 204.1 | 600.0 | 607.55 | 2.98 |
| 2023-01-29 14 :46 :13 | 85.94 | 598.7 | 614.45 | 7.15 |
| 2023-01-12 02 :25 :26 | 10.28 | 580.0 | 583.5 | 56.76 |
| 2023-01-06 16 :53 :01 | 88.83 | 557.4 | 581.7 | 6.55 |
| 2023-01-06 08 :13 :47 | 0.0 | 550.35 | 551.35 | — |
| 2023-01-23 23 :01 :23 | 100.46 | 550.0 | 560.05 | 5.57 |
| 2023-01-24 02 :08 :50 | 0.0 | 550.0 | 557.55 | — |
| 2023-01-06 06 :18 :19 | 80.38 | 547.6 | 550.35 | 6.85 |
| 2023-01-07 04 :29 :56 | 93.05 | 542.0 | 564.05 | 6.06 |

Table 13 – Top 20 des trajets les plus coûteux

```
GROUP BY pickup_hour
ORDER BY revenue_efficiency DESC
```

| pickup_hour | trip_count | hourly_revenue | avg_revenue_per_trip | avg_duration | revenue_efficiency |
|---|---|---|---|---|---|
| 5 | 18005 | 649398.9 | 36.07 | 15.41 | 36.07 |
| 4 | 17823 | 551315.77 | 30.93 | 15.4 | 30.93 |
| 6 | 43832 | 1325112.75 | 30.23 | 15.09 | 30.23 |
| 23 | 114489 | 3384431.24 | 29.56 | 15.29 | 29.56 |
| 16 | 195911 | 5726167.34 | 29.23 | 17.56 | 29.23 |
| 0 | 84948 | 2416259.32 | 28.44 | 15.5 | 28.44 |
| 17 | 209422 | 5930259.05 | 28.32 | 16.9 | 28.32 |
| 22 | 147365 | 4155679.38 | 28.2 | 14.98 | 28.2 |
| 14 | 191529 | 5309220.67 | 27.72 | 17.5 | 27.72 |
| 21 | 161475 | 4401002.12 | 27.26 | 14.24 | 27.26 |
| 15 | 196346 | 5329948.65 | 27.15 | 17.54 | 27.15 |
| 19 | 192743 | 5224867.35 | 27.11 | 14.52 | 27.11 |
| 7 | 86841 | 2316552.71 | 26.68 | 15.11 | 26.68 |
| 20 | 165805 | 4420565.74 | 26.66 | 14.23 | 26.66 |
| 18 | 215817 | 5705225.95 | 26.44 | 15.23 | 26.44 |
| 13 | 178656 | 4674425.77 | 26.16 | 16.12 | 26.16 |
| 1 | 59786 | 1558296.08 | 26.06 | 14.82 | 26.06 |
| 3 | 27429 | 705825.86 | 25.73 | 14.4 | 25.73 |
| 12 | 169791 | 4306825.21 | 25.37 | 15.37 | 25.37 |
| 9 | 131070 | 3314764.43 | 25.29 | 15.25 | 25.29 |

Table 14 – Revenus et efficacité par heure de prise en charge

On peut se rendre compte que les meilleurs moments pour conduire c'est à 5h, 6h qui représentent surement les heures de sortie des boites de nuits. Ensuite nous avons des heures comme 23h, 0h, 16h, ou 17h qui peuvent être associées aux heure de sortie de bureau ou aux heures pour se rendre en boite de nuit.

## 9.10 Quelles sont les heures les plus rentables pour conduire ?

```
SELECT
    pickup_hour,
    COUNT(*) as trips,
    ROUND(SUM(total_amount), 2) as total_revenue,
```

```
    ROUND(AVG(total_amount), 2) as avg_revenue
FROM nyc_taxi.trips_partitioned
GROUP BY pickup_hour
ORDER BY total_revenue DESC
LIMIT 5
```

| pickup_hour | trips | total_revenue | avg_revenue |
|---|---|---|---|
| 17 | 209493 | 5931916.23 | 28.32 |
| 16 | 195977 | 5728111.57 | 29.23 |
| 18 | 215889 | 5706585.63 | 26.43 |
| 15 | 196424 | 5332055.9 | 27.15 |
| 14 | 191604 | 5311244.71 | 27.72 |

TABLE 15 – Top 5 des heures les plus rentables

Nous pouvons voir que les heures tels que 17h, 16h, 18h sont les plus rentables.

## 9.11 Le nombre de passagers a t-il un impact sur le pourboire ?

```
SELECT
    passenger_count,
    COUNT(*) as trips,
    ROUND(AVG(tip_amount), 2) as avg_tip,
    ROUND(AVG(tip_percentage), 2) as avg_tip_pct
FROM nyc_taxi.trips_partitioned
WHERE passenger_count > 0 AND passenger_count <= 6
GROUP BY passenger_count
ORDER BY passenger_count
```

| passenger_count | trips | avg_tip | avg_tip_pct |
|---|---|---|---|
| 1.0 | 2261400 | 3.32 | 21.57 |
| 2.0 | 451536 | 3.59 | 19.92 |
| 3.0 | 106353 | 3.37 | 19.39 |
| 4.0 | 53745 | 3.28 | 17.92 |
| 5.0 | 42681 | 3.38 | 20.98 |
| 6.0 | 28124 | 3.36 | 20.76 |

TABLE 16 – Statistiques des pourboires par nombre de passagers

Nous pouvons dire que oui plus on a de passager, plus le montant du pourboire augmente légèrement.

## 9.12 Quelle est la durée moyenne de trajet par distance ?

```
SELECT
    CASE
        WHEN trip_distance < 2 THEN '0-2 miles'
        WHEN trip_distance < 5 THEN '2-5 miles'
        WHEN trip_distance < 10 THEN '5-10 miles'
        ELSE '10+ miles'
    END as distance_range,
    ROUND(AVG(trip_duration_minutes), 2) as avg_duration_min,
    ROUND(AVG(trip_distance / NULLIF(trip_duration_minutes, 0) * 60), 2) as avg_speed_mph
FROM nyc_taxi.trips_partitioned
WHERE trip_duration_minutes > 0 AND trip_distance > 0
GROUP BY
    CASE
        WHEN trip_distance < 2 THEN '0-2 miles'
        WHEN trip_distance < 5 THEN '2-5 miles'
```

```
            WHEN trip_distance < 10 THEN '5-10 miles'
            ELSE '10+ miles'
        END
```

| distance_range | avg_duration_min | avg_speed_mph |
|---|---|---|
| 5-10 miles | 25.36 | 21.85 |
| 2-5 miles | 18.01 | 12.16 |
| 10+ miles | 40.58 | 66.72 |
| 0-2 miles | 9.3 | 9.47 |

TABLE 17 – Durée et vitesse moyennes par plage de distance

# Références

— • Hadoop The Definitive Guide, 4th Edition https://www.oreilly.com/library/view/hadoop-the-definitive/9781491901687/).