Contents

Part 1

Corrected and Simplified Summary

This section explains how to complete a homework assignment using Spark and HDFS to analyze two CSV files containing real restaurant inspection data. Here's a clearer, step-by-step breakdown:

1. Uploading Data to HDFS

- You start by creating a directory in HDFS using the hdfs dfs -mkdir -p command.
- Both CSV files are uploaded into this directory using the hdfs dfs -put command, which requires two arguments: the local file path and the HDFS target directory. If you don't specify the target, the file goes to your default HDFS user directory1.

2. Preparing to Use Spark

- Import necessary Spark libraries after uploading your files.
- You'll use Spark DataSets, which are typed DataFrames. This means you need to define *case classes* in Scala to represent the structure of your data—one for restaurants and one for inspections1.

3. Defining Schemas

- Since the CSV files have no headers, Spark would otherwise assign generic column names like _c0, _c1, etc.
- To avoid this, you explicitly define the schema for each file using Spark's StructType, listing each column name, its data type, and whether it can be null1.

4. Loading Data into Spark DataSets

- Use the Spark read function to load the CSVs from HDFS:
  - Set header to false because there's no header row.
  - Set inferSchema to false since you're providing the schema.
  - Specify the schema and file path.
- Map each row to your case class using .as[CaseClassName]. Spark automatically matches CSV columns to your case class fields1.
- Optionally, use .cache() to keep the data in memory for faster access during analysis.

5. Exploring the Data

- Use .count() to get the number of records in each DataSet.
- Use .show(n, false) to display the first n rows without truncating the output1.

6. Running Basic Queries

- To find the top 10 cuisine types or violation codes, use group-by and count operations, then order the results. For example:
  - Group by cuisine_type, count, and order by count descending.
- When ordering by a computed column (like count), you must use the column object, not just the string name, to apply descending order1.

7. Using Zeppelin for Visualization

- Results can be shown as tables or charts in Zeppelin.
- Use Zeppelin's show command to display Spark DataFrame results as tables or graphs.

8. Using Spark SQL API

- Create temporary views from your DataSets with .createOrReplaceTempView("view_name"). These views exist only in memory for the session and are read-only.
- Run SQL queries directly on these views using Spark SQL, which is useful for more complex analysis or for those more comfortable with SQL syntax1.

Key Points to Remember

- Always define schemas explicitly when headers are missing to avoid confusion.
- Use case classes for typed DataSets in Spark.
- Temporary views make it easy to use SQL for data exploration.
- Zeppelin can visualize both text and table outputs from Spark queries.

This approach ensures your data is structured, loaded, and analyzed efficiently using Spark and HDFS, with clear steps for both code and visualization1.

Part 2

Choosing the Best File Format for Big Data: A Simple, Accurate Summary

This section discusses how to select the best file format for storing and processing large datasets in distributed systems like Hadoop and Spark. Here's a clear breakdown of the main points:

1. Why File Format Matters

- In earlier lessons, you learned how to upload data to HDFS, load it with Spark, partition it, define schemas with Hive, and query it using Trino or Superset.
- Up to now, you've mainly worked with CSV or text files. These are easy to use but not ideal for large-scale, distributed data processing because they are slow and not optimized for performance1.

2. Introduction to Advanced File Formats

- For production environments and better performance, you need more efficient file formats.
- The three main formats discussed are Avro, Parquet, and ORC. These are widely supported in the Hadoop and Spark ecosystems and are designed to be much more efficient than CSV or plain text1.

3. What Makes a Good File Format?

A good file format for big data should have these features:

For Humans (Ease of Use):

- **Well-defined:** The format should avoid errors when reading data, even if some values are missing or incorrectly coded. For example, CSV files can cause errors if a comma appears in the data without proper handling1.
- **Expressive:** It should be easy to extract the needed information in as few steps as possible. For example, JSON arrays are easier to read than splitting a string in CSV1.
- **Simple:** The format should not require complex parsing or extra metadata (like XML does).

For Machines (Performance):

- **Optimized Binary Encoding:** The format should store data in a compact, efficient way to save space and speed up processing. For example, binary formats like Avro and Parquet use special encoding to reduce file size1.
- **Native Compression:** The format should support built-in compression, so you don't need to manually zip or unzip files, which slows down processing1.
- **Splittable:** The format should allow large files to be split into chunks for parallel processing, which is essential in distributed systems like Hadoop1.

4. Why Not Just Compress CSV Files?

- Simply zipping a CSV file reduces its size but makes it unreadable for distributed processing. You'd have to unzip it before processing, which wastes time and resources. Efficient file formats keep data compressed and readable at the same time.

5. Examples of File Formats

- **CSV/TSV:** Simple, human-readable, but not efficient for big data.
- **JSON/XML:** More expressive but can be large and complex.
- **Avro, Parquet, ORC:** Binary formats optimized for big data. They are well-defined, expressive, simple for machines, use efficient encoding, support native compression, and are splittable1.

6. Summary Table

Feature	**CSV/JSON/XML**	**Avro/Parquet/ORC**
Human-readable	Yes	No
Well-defined	Sometimes	Yes
Expressive	Sometimes	Yes
Simple	CSV: Yes, XML: No	Yes

Feature	**CSV/JSON/XML**	**Avro/Parquet/ORC**
Optimized for machines	No	Yes
Native compression	No	Yes
Splittable	No	Yes

In short:
For big data processing, use Avro, Parquet, or ORC instead of CSV or JSON. These formats are faster, save space, and are designed for distributed systems, making your queries and data handling much more efficient.

Part 3

Clear and Accurate Summary of Avro, Parquet, and ORC File Formats
Avro

- **What it is:**
  Avro is a row-based data serialization system, which means it stores data one row after another, similar to how a table works in a traditional database.
- **Schema requirement:**
  You must provide a schema (in JSON format) that defines the column names, data types, and whether each field can be null. This schema is mandatory for both writing and reading Avro files1.
- **Use cases:**
  Avro is ideal for collecting and transferring data in real time (data in motion), such as streaming data from one system to another1.
- **Advantages:**
  - Highly compatible with Hadoop and distributed systems
  - Splittable and supports native compression (efficient for big data)
  - Supports complex and nested data structures
  - Works with many programming languages
  - Data is stored on disk, not in memory, making it reliable and safe in case of failures (“write and forget”)1.
- **Summary:**
  Use Avro when you need to serialize and transfer data reliably between systems, especially for streaming or real-time scenarios.

Parquet

- **What it is:**
  Parquet is a column-based storage format. Instead of saving one row at a time, it stores all values of each column together. This makes it very efficient for analytical queries that read only a few columns from large datasets1.
- **Schema requirement:**
  Like Avro, Parquet also requires a schema, which can often be reused between formats1.
- **Use cases:**
  Parquet is best for storing and querying large datasets in distributed environments, especially in the cloud, where you pay for the amount of data processed1.
- **Advantages:**
  - Extremely fast for reading and querying data
  - Supports efficient compression and encoding, reducing storage size (e.g., a 10GB CSV might become a 1.2GB Parquet file)
  - Allows reading only selected columns, saving time and resources
  - Widely supported in Spark, Hive, and cloud platforms
- **Considerations:**

    - Encoding (writing) Parquet files uses a lot of memory, as it needs to process all rows for each column before writing to disk
    - In case of failure during writing, data may be lost since it's kept in memory until flushed to disk1.
- **Summary:**
  Use Parquet for efficient storage and fast querying of large, analytical datasets, especially when you often read only a subset of columns.

ORC (Optimized Row Columnar)

- **What it is:**
  ORC is a columnar storage format, mainly used with Hive, that combines the best features of Avro and Parquet1.
- **Schema requirement:**
  ORC also requires a schema, which can be created manually or inferred from Hive tables1.
- **Use cases:**
  ORC is designed for heavy, long-running queries and large-scale data processing, especially in Hive environments1.
- **Advantages:**
    - Fast for reading and aggregating data, thanks to built-in metadata and statistics (like bloom filters) stored in the file header
    - Supports selective column reading and native compression (default is Deflate, which is highly efficient)
    - Compatible with Hive, Spark, and Trino
- **Summary:**
  Use ORC for high-performance, large-scale analytics in Hive or Hadoop environments, especially when you need fast aggregation and filtering.

General Workflow for Using These Formats

1. **Define a schema:**
   Always start by creating a schema (preferably in lowercase, as it's case-sensitive).
2. **Create an empty container:**
   Use the schema to create an empty Avro, Parquet, or ORC file (or table).
3. **Populate the data:**
   Load or insert your data into the container. In Hive, simply creating a table does not convert your data; you must explicitly insert or select data into the new format.
4. **Read and use the data:**
   Once populated, these files can be read by Spark, Hive, Trino, and many other tools1.

Key Differences Table

Feature	**Avro (Row)**	**Parquet (Column)**	**ORC (Column)**
Orientation	Row	Column	Column
Schema required	Yes	Yes	Yes
Best for	Data in motion, transfer	Analytics, fast queries	Analytics, Hive workloads
Compression	Native	Native	Native (Deflate)
Memory usage (write)	Low	High	Moderate
Failure safety (write)	High	Lower	Moderate
Selective column read	No	Yes	Yes
Main ecosystem	Hadoop, Streaming	Spark, Cloud, Hadoop	Hive, Hadoop, Spark

---

In summary:

- Use **Avro** for reliable, schema-based data transfer and streaming.
- Use **Parquet** for fast, efficient analytics and querying of large datasets.
- Use **ORC** for high-performance analytics in Hive and Hadoop environments, especially when you need advanced metadata and compression1.

Part 4

1. Introduction to Hive

- **Hive** is a data warehouse system built on **Hadoop**. It lets you query big data stored in **HDFS** using **HiveQL** (a SQL-like language).
- It translates HiveQL into **MapReduce jobs** and runs them on the Hadoop cluster.

2. Hive's Meta Store (HMS)

- **Hive Meta Store (HMS)** stores metadata about:
  - **Databases**
  - **Tables**
  - **Schemas**
  - **Partitions**
- Other tools like **Spark** or **Trino** can access the Hive Meta Store to understand data structure.

3. File Formats in Hive

Hive supports these file formats:

- **Text (CSV, JSON):**
  - Human-readable but inefficient (large file size).
- **Avro:**
  - Good for row-based storage.
  - Schema is stored with the data.
- **Parquet & ORC:**
  - Columnar formats → faster for read-heavy operations.
  - Better compression and performance.

Why use Avro, Parquet, or ORC?

- Faster reads/writes
- Less storage space
- Supports schema evolution (especially Avro)

4. Partitioning

- **Partitioning** splits large datasets by column values (e.g., date, region).
- Queries can read only relevant partitions → much faster.

🟡 **Note:** Partitioning is like creating subfolders for your data in HDFS.

5. Compression

- Saves storage space and speeds up data transfer.
- Types:
  - **Snappy**: Fast, commonly used with Parquet/ORC.
  - **Gzip**: Good compression, slower than Snappy.
- Columnar formats (Parquet, ORC) work very well with compression.

6. Schema Management

- Hive uses **schemas** (like tables with column names and types).
- Stored in the **Meta Store**.
- Avro format supports **schema evolution**: you can change column names/types over time without breaking older data.

7. ACID in Hive

- Hive supports **ACID (Atomicity, Consistency, Isolation, Durability)** for:
  - INSERT
  - UPDATE
  - DELETE
- Works with **ORC format** and **transactional tables**.
- ACID is useful for changing data in place, which is rare in big data but sometimes needed (e.g., GDPR deletions).

8. Hive vs Spark vs Trino

Tool	Use Case	Engine	Notes
Hive	Batch queries on HDFS	MapReduce/Tez	Older but stable
Spark	Fast, in-memory analytics	Spark engine	Good for complex jobs
Trino	Fast, interactive SQL queries	No data movement	Reads from many sources (S3, Hive, etc.)

---

9. Data at Rest vs Data in Motion

- **Data at Rest**: Stored in databases, HDFS, S3, etc.
- **Data in Motion**: Flowing in real-time (e.g., via Kafka).

💡 **Compression & schema handling** are more common for data at rest.

10. Key Benefits of Hive

- Works well with Hadoop and HDFS.
- Scalable via partitioning and file formats.
- Widely supported by tools like Spark, Trino, and Presto.
- Meta Store makes it easy to manage schema across tools.

Part 5

Here is a **corrected, accurate, and simplified summary** of the original technical walkthrough. It has been rewritten in clear, professional language and organized in logical steps to make it easier to understand:

🔧 **Summary: Comparing File Formats and Compression in Hive Using Avro and Parquet**

1. Creating and Populating Avro Tables Without Compression

- A Hive table was created using the Avro format (STORED AS AVRO) and populated using CREATE TABLE AS SELECT from a base table (originally in CSV/text format).
- By default, Hive does **not compress** Avro output unless explicitly specified.
- Compression was **disabled manually** (set avro.output.codec=uncompressed) to observe the raw size of the data.

Results:

- **Time Taken:** ~20 seconds to run the query.
- **Number of Files:** 7 files were generated (due to parallel processing).
- **Total Size:** Increased to **~366 MB** from the original **~324 MB**.
- **Why Bigger?** Avro uses **16-byte sync markers** between data blocks, which increase file size when compression is off.

2. Creating Avro Tables with Snappy Compression

- Repeated the same process, but enabled **Snappy compression**.
- Snappy aims to balance **speed and file size**, not maximum compression.

Results:

- **Time Taken:** 19 seconds.
- **Compressed Size:** Reduced to **~147 MB** (a ~55% reduction from the raw CSV).
- **Number of Files:** Still 7, due to the same degree of parallelism.

3. Creating Avro Tables with Deflate Compression

- Used **Deflate compression** (default in Hive for Avro).
- Again ran CREATE TABLE AS SELECT and computed statistics.

Results:

- **Compressed Size:** Further reduced to **~94 MB**, down from 324 MB (~70% space savings).

4. Understanding File Count and Parallelism

- The number of output files (e.g., 7) depends on **how many parallel tasks** ran during processing.
- In production, with more resources (e.g., more CPU cores), **you may see a different number of files**.

5. Inspecting Avro and Parquet Files in HDFS

- Avro files (both compressed and uncompressed) were saved to HDFS.
- Tools such as hdfs dfs -ls, du -s, and others were used to verify file size and structure.

6. Parquet Format: Raw vs. Compressed

- Parquet tables were also created (from CSV sources) to compare formats.

Observations:

- **Uncompressed Parquet** is **smaller than uncompressed Avro** because it doesn't use 16-byte sync markers.
- **Parquet + Snappy**: Balanced compression, size ~129 MB.
- **Parquet + Gzip/Deflate**: Better compression, size as low as **82 MB**.
- **Best compression** was achieved with **Gzip**, reducing file size from 340 MB to **~78 MB** (~75% reduction).

7. Why Use Snappy or Gzip?

Compression	**Purpose**	**Size Reduction**	**Speed**
Snappy	Balanced (speed + reasonable size)	Moderate (~50-60%)	Fast (preferred in Spark)
Gzip	Maximum size reduction	High (~70-75%)	Slower

- On **cloud environments**, Gzip is often used to **minimize storage costs**, especially for large datasets.

8. Reading Metadata and Schema

- Using **Avro and Parquet tools**, the internal schema, compression codec, and stats (e.g., min/max values, null counts) were inspected.
- **Parquet files** store more metadata, including statistics like:
  - Min/Max values
  - Number of nulls
  - Bloom filter support (partial)
  - Compression type (e.g., Snappy, Gzip)

9. Key Differences: Avro vs. Parquet

Feature	**Avro**	**Parquet**
Format Type	Row-based	Columnar
Compression Support	Yes (Snappy, Deflate, etc.)	Yes (Snappy, Gzip, etc.)
File Size (Raw)	Larger (due to sync markers)	Smaller
Schema Evolution	Excellent	Good
Best Use Case	Streaming, write-heavy	Analytics, read-heavy

10. Final Remarks and Errors

- There was an issue reading Avro due to a **Java version mismatch** after updating tooling.
- A fix will be provided for compatibility.
- For **already populated Avro files**, you can **create external Hive tables** directly pointing to the file path without reloading data.

References:
Part 1
1. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/51956437/96b0a270-c8e7-4ef3-a50a-530d11c80b0e/paste.txt

Part 2
1. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/51956437/d65ea0ba-ee57-4665-a67d-d10b970195c8/paste.txt
2. https://www.upsolver.com/blog/the-file-format-fundamentals-of-big-data
3. https://en.wikipedia.org/wiki/Apache_Parquet
4. https://orc.apache.org/docs/
5. https://www.hashstudioz.com/blog/optimizing-storage-formats-in-data-lakes-parquet-vs-orc-vs-avro/
6. https://en.wikipedia.org/wiki/Apache_Avro
7. https://www.ibm.com/think/topics/avro
8. https://airbyte.com/data-engineering-resources/what-is-avro
9. https://avro.apache.org
10. https://techdocs.broadcom.com/us/en/vmware-tanzu/data-solutions/tanzu-greenplum-platform-extension-framework/6-7/gp-pxf/hdfs_avro.html
11. https://towardsdatascience.com/comparing-performance-of-big-data-file-formats-a-practical-guide-ef366561b7d2/

Part 3:
1. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/51956437/15e0bb93-13e8-4775-8eab-d7492ae91e9d/paste.txt