Contents

[Part 1 - ChatGPT 1](#_Toc199777750)

[Part 2 – ChatGPT 2](#_Toc199777751)

[Part 3 – ChatGPT 4](#_Toc199777752)

[Part 4 – ChatGPT 5](#_Toc199777753)

[Part 5 - ChatGPT 6](#_Toc199777754)

[Part 6-ChatGPT 7](#_Toc199777755)

[Part 7-ChatGPT 8](#_Toc199777756)

[**Part 8-ChatGPT** 9](#_Toc199777757)

## Part 1 - ChatGPT

**Homework Deadlines and Submission Instructions**  
Students were informed that the first homework will be available after class and must be submitted by Wednesday midnight, before the next class. The assignment will be based on Apache Spark and should be submitted as a .zip file. Although some confusion arose due to conflicting dates in the course portal, the confirmed deadline is June 4. Extensions are not allowed because of a tight schedule, and late submissions will be penalized. The instructor will provide detailed instructions after the class break.

**Last Homework and Final Course Schedule**  
The final team homework will be given on July 17, which is one week before the last class on July 24. Students will have ten days to complete and submit this last homework, covering two weekends. This extended deadline is provided due to the complexity of the final task. No extensions are planned for any assignments because the course timeline is already packed.

**Tools and Technologies for Homework**  
The first homework is based on Spark and Scala, requiring students to write 6–7 lines of code. Although not difficult, it's essential for students to become comfortable with Spark early on, as future homeworks will build on this. The Spark tutorial and workshop are expected to help students gain the needed hands-on experience. All practical work for now will continue to be done using Zeppelin.

**Recap of Course Part 1 – Distributed Storage & Processing**  
The first part of the course focused on distributed data storage and processing. Students learned how to store data in HDFS and process it using tools like MapReduce (on disk) and Spark (in memory). They practiced managing files in HDFS, which forms the foundation for future assignments. Comfort with uploading, downloading, and exploring data in HDFS is essential as it will be used throughout the course.

**Start of Part 2 – Data Analysis with Spark and Hive**  
The second part of the course begins with data analysis using Spark and Hive. Students will learn to clean, filter, and prepare data stored in HDFS. New topics introduced include partitioning and bucketing to organize and optimize large datasets. The upcoming classes will also cover SQL and semi-SQL operations, including working with different file formats like CSV and Parquet.

**Introduction to Hive and Partitioning Concepts**  
This week’s focus is Hive, a tool that allows SQL-like querying of large datasets. Students will learn Hive architecture, its data model, and HiveQL, which includes familiar SQL syntax along with some new keywords. Partitioning and organizing data into structured segments are key skills covered. Next week, the topic will continue with Trino, another SQL engine.

**Hands-On with Structured and Semi-Structured Data**  
The practical component includes working with structured data (like CSV files) and semi-structured data (like JSON). Students will learn how to handle headers, clean files, and prepare them for analysis. JSON files require slightly advanced handling in Hive due to their format. The focus is currently on structured data, with tools like Spark SQL, Hive, and later Trino being central to the process.

**Real-World Relevance of Tools (Hive, Trino, Spark)**  
Tools being taught—like Hive, Trino, and Spark SQL—are widely used in the industry. For example, services like Google BigQuery and AWS Athena use Trino or its earlier version, Presto, in the background. Learning these tools will prepare students for similar environments in real-world data platforms. The course emphasizes transferable skills across cloud and on-premises systems.

**Understanding the Big Data Tool Landscape**  
Students were introduced to a high-level overview of the big data ecosystem. Tools like Hadoop, Spark, MapReduce, and YARN manage computation and storage, while others like Flink handle real-time data processing. Students will work mainly with text and binary file formats, especially Parquet and ORC. Familiarity with different data formats is critical for efficient storage and querying.

## Part 2 – ChatGPT

**1. Optimized File Formats for Big Data**  
You can use many data formats like CSV, JSON, or XML, but these are not efficient for large-scale data processing. More optimized formats like Avro, Parquet, and ORC are better suited for big data because they store data efficiently and support faster processing. Formats like Delta Lake and Hudi are also used in data lakes to store changes or versions of data efficiently. Though we focus on Avro, ORC, and Parquet, the same ideas apply to formats like Iceberg and Hudi.

**2. Indexing and Limitations in HDFS**  
HDFS (Hadoop Distributed File System) does not have an indexing feature like databases do. It mainly handles file operations such as creating, renaming, or moving files and directories. If you want to read a file, you must read all its blocks in order—you cannot skip or jump to a specific block. This limitation makes data partitioning very important for efficient querying.

**3. What is Partitioning in Big Data**  
Partitioning means splitting data based on a specific column (called the partition key). For example, if you partition by country, all rows with "USA" will be grouped in one partition, and "France" in another. Each partition becomes a separate folder on HDFS containing the related data. You can use multiple columns to create sub-partitions like year, month, and day, depending on how users query the data.

**4. Good vs. Bad Partitioning**  
Partitioning works best when the chosen column has few unique values. For example, countries work well because there are less than 200 of them. But if you partition by something like product ID with thousands of unique values, it becomes inefficient. In that case, bucketing is a better choice.

**5. What is Bucketing**  
Bucketing helps divide data within partitions into smaller groups based on a column with many unique values, like product ID. It improves balance and performance by evenly spreading data across buckets. You can use partitioning without bucketing, or vice versa, but often they are used together. Bucketing makes queries faster when dealing with large datasets and high-cardinality columns.

**6. Hive Overview**  
Hive is a data warehouse system built on top of Hadoop. It organizes data on HDFS and allows you to query it using a SQL-like language. Hive supports many file formats and can be extended using SerDe (serializer/deserializer) components, which tell Hive how to read and write data in each format, like CSV or JSON.

**7. Hive Schema and Metastore**  
When using Hive, you must define your table’s schema (the list of columns and their types). Hive cannot automatically detect this. The schema is stored in the Hive Metastore, which keeps track of table structures, partitions, and configurations. This setup allows Hive to manage and query large datasets effectively.

**8. Hive Extensions and UDFs**  
Hive supports user-defined functions (UDFs) to extend its capabilities. You can create your own functions or use prebuilt ones for tasks like machine learning. These are written in Java and enhance how you process and analyze data inside Hive. Though UDFs are advanced, it's important to know they exist and can greatly expand Hive's functionality.

**9. HiveQL: Hive Query Language**  
Hive uses HiveQL, a simplified version of SQL with added keywords for dealing with file formats and SerDes. For example, it lets you handle headers, quote characters, and custom delimiters in CSV files. HiveQL is not full SQL, but it is close and easy to learn for those familiar with SQL. Spark can also run HiveQL queries, making it flexible.

**10. Advanced Hive Data Types**  
Hive supports complex types like STRUCT, LIST, and MAP. STRUCT allows nesting columns (useful for objects like "User" with fields like name and date of birth). LIST stores arrays (e.g., a product package with multiple items). MAP holds key-value pairs (often used for processing MapReduce outputs). These types allow Hive to handle semi-structured data efficiently.

**11. Hive’s Use Case and Limitations**  
Hive is designed to handle large-scale batch processing, not real-time or interactive queries. It runs jobs in the background and is efficient for analyzing petabytes of data. If you need fast, interactive querying, tools like Trino (covered later) are more suitable. Hive is excellent for scheduled or long-running data analytics tasks.

## Part 3 – ChatGPT

**1. Hive Storage and Partitioning Structure**  
Hive organizes data in a directory structure, but users should not modify or delete files in the default Hive warehouse directory (/user/hive/warehouse) as it can break Hive’s ability to access data. Hive tables can be partitioned to improve performance. A partition is like a folder labeled by a specific column value, such as date=2023-05-13. Developers decide how to partition data, and further subdirectories like hour=18 can also exist. Inside these folders, data files are often divided into "buckets" for better distribution.

**2. Partitions and Buckets Overview**  
Partitions help organize data by column values (e.g., dates), and each partition can have multiple buckets—numbered files that store segments of the data. This structure helps with faster data access and is useful in large-scale systems. While partitioning is optional, it is highly recommended for tables used in production environments or shared across tools. In smaller or temporary setups, partitioning may not be necessary.

**3. Internal vs External Tables in Hive**  
Hive supports two types of tables: **internal (managed)** and **external (unmanaged)**. Internal tables are fully controlled by Hive—it manages both the schema and the data. External tables are only schema-managed by Hive; the actual data is controlled by the user. When an internal table is dropped, both the schema and data are deleted. In contrast, dropping an external table deletes only the schema, keeping the data safe in HDFS.

**4. Use Cases for Table Types**  
Use **internal tables** when data is temporary, used only by Hive, or not needed after deletion—like for staging or transformation steps. Use **external tables** when data is shared with other tools (e.g., Spark), already exists in HDFS, or is important and must not be deleted accidentally. External tables are safer for production because the data stays intact even if the Hive table is dropped.

**5. Creating Internal Tables and Loading Data**  
To create an internal table, use the CREATE TABLE command and define the schema (column names and types). After that, you must load the data explicitly using commands like LOAD DATA INPATH. This tells Hive where the data file is in HDFS and loads it into the table. Dropping an internal table deletes both the table and the file loaded into it.

**6. Creating External Tables and Pointing to Data**  
To create an external table, use CREATE EXTERNAL TABLE, define the schema, and specify the LOCATION of the existing directory in HDFS. Hive uses this location to read the data but doesn't control it. This is useful when data was created by other tools or exists outside Hive. Hive won’t delete these files even if the table is dropped, which protects valuable production data.

**7. Why Location Must Be a Directory**  
When creating external tables, the LOCATION must point to a directory, not a single file. This is because Hadoop and Spark often generate multiple output files during processing. These systems are designed to handle reading multiple files automatically, so merging them isn’t needed. Hive will read all valid files in the directory as part of the table.

## Part 4 – ChatGPT

**1. Static vs. Dynamic Partitioning in Hive**  
Hive supports two types of partitioning: static and dynamic. Static partitioning (also called strict mode) means the user manually provides the partitioning column and values (e.g., partitioning sales data by country). It's efficient, fast, can be scripted, and is preferred in production. Dynamic partitioning (non-strict mode) lets Hive decide the partitions by loading all data into memory and grouping it—which consumes more memory and is very slow. For this reason, dynamic partitioning is only used for testing or in development environments, not in production.

**2. How Dynamic Partitioning Works and Its Limitations**  
In dynamic partitioning, Hive reads the entire dataset into memory, groups it by a column, and then writes each group as a separate partition. This process is memory-intensive and slow, so it's disabled or limited by default in most environments. It’s mainly useful for developers to quickly test partitioning scripts on small datasets before switching to static partitioning for production use.

**3. Using Spark to Handle Partitions More Efficiently**  
Since dynamic partitioning is inefficient, it’s better to handle partitioning with Spark. Spark allows you to load data, clean and prepare it, and then output the data in a partitioned format. These partitions can then be stored and later used by Hive. This method is faster and more scalable.

**4. Bucketing in Hive**  
Bucketing is a method to further organize data inside partitions by splitting it into manageable "buckets" using a CLUSTERED BY clause. Bucketing can be done with or without partitioning. A table can have any combination: partitioned, bucketed, both, or neither. Bucketing helps group data more efficiently and improves query performance, especially when combined with sorting.

**5. Sorting Within Buckets and Performance Benefits**  
You can sort data inside a bucket, though sorting is not enabled by default because it uses more resources. For example, in real estate data, you can partition by city, bucket by street, and sort prices in ascending order. This way, when you query data for a specific city, Hive can quickly access and return sorted results without scanning the entire dataset.

**6. Directory Structure for Partitioned and Bucketed Tables**  
Partitioning and bucketing create a physical directory structure in Hive. For example, a sales table might be partitioned by year and month, and each partition may contain 6 buckets. Queries targeting a specific year and month can directly access the correct folder and bucket, avoiding full table scans and improving performance.

**7. File Formats Hive Can Read**  
Hive supports several file formats: ORC, CSV/TSV (text), Avro, Parquet, JSON, SequenceFile, and more. Most common formats like ORC, Parquet, and JSON are supported in standard environments. Choosing the right format depends on your data and performance needs.

**8. Loading Data into Hive**  
Data can be loaded into Hive tables using the LOAD command from HDFS, specifying the correct file format (e.g., text, JSON). You can also populate a table using INSERT INTO, or create a new table using CREATE TABLE AS SELECT. Later in the course, tools like NiFi and Spark will be used to push data into Hive more efficiently.

**9. Hive Query Language: SQL-like but Not Relational**  
Hive uses a SQL-like language to query data using SELECT, WHERE, and JOIN, but it’s not relational like traditional RDBMS. You should avoid designing Hive tables as if they were relational tables. Instead, tables should be independent, and joins should be used sparingly to avoid performance issues.

**10. Hive DDL, DML, and Data Types**  
Hive supports standard SQL-like commands: DDL (create databases, tables, views) and DML (select, insert, update, delete). It also supports various primitive data types like int, float, double, string, and boolean. Working with dates and timestamps can be tricky because Hive requires millisecond precision—if missing, it may not parse timestamps correctly.

**11. Complex Data Types in Hive**  
Hive supports complex data types: STRUCT (nested columns), MAP (key-value pairs), and ARRAY (lists). These allow you to store structured or hierarchical data within a single column, but they require specific functions to access and manipulate.

## Part 5 - ChatGPT

**1. Organizing Data by Schema**  
It's important to store different types of data (with different structures) in separate folders. For example, data about bike stations and bike rides have different columns, so they should be placed in separate directories. This is necessary because Hive expects all files in a folder to have the same structure. Mixing schemas in the same directory can cause errors during data loading.

**2. Understanding Hive Databases and Tables**  
When you create a table in Hive without specifying a database, it is stored in the default database. The SHOW TABLES command lists all tables in the current database. Most users create their own custom databases to keep things organized and avoid clutter in the default one.

**3. Creating an Internal Table in Hive**  
If you use CREATE TABLE without mentioning EXTERNAL, Hive creates an internal table. Hive manages both the table schema and the data for internal tables. It automatically adds metadata like row format, field delimiter (default is tab), and storage type (e.g., text file). Although these settings are added by default, it's better to specify them yourself for clarity and easier maintenance.

**4. Loading Data into a Table**  
To load data into a table, use LOAD DATA INPATH. If you don’t provide a full HDFS path, Hive looks for the file in the current user's HDFS directory. The path can be a file or a folder, and the exact name (including file extension if it has one) must match the actual stored file name.

**5. Writing Queries with Limits**  
To avoid fetching huge amounts of data, always limit the number of rows in your queries. Use LIMIT to return a small number of rows, especially when dealing with large datasets. For example: SELECT \* FROM table WHERE frequency > 100 ORDER BY frequency LIMIT 10.

**6. Creating Tables for CSV Files**  
When creating a table to read a CSV file, use FIELDS TERMINATED BY ',' to set the comma as the column separator. If the CSV file has a header row, you need to manually remove or handle it, as Hive doesn't ignore headers by default. You can also use IF NOT EXISTS to avoid errors if the table already exists.

**7. External Tables and Reserved Keywords**  
External tables are defined with the EXTERNAL keyword, and their data is not managed by Hive (just referenced). The LOCATION keyword specifies where the data is stored (always a directory). If a column name uses a reserved word (like year), use backticks (`year`) to avoid errors.

**8. Loading Partitioned Data**  
To load data into a partitioned table, use LOAD DATA INPATH INTO TABLE table\_name PARTITION (key=value). For example, you can partition by date and country, and Hive will treat these as folder levels in the directory. Partitioning helps organize large datasets efficiently.

**9. Exporting Table Data**  
You can export data using INSERT OVERWRITE DIRECTORY 'path' SELECT .... Make sure Hive has write permissions to the target directory, or you'll get a permission error. This method stores query results as files, but in practice, many pipelines use Spark or other tools for exporting and transformation.

**10. SerDe and File Formats**  
SerDe (Serializer/Deserializer) handles reading and writing data between memory and disk. Hive has built-in SerDes for formats like CSV, Parquet, ORC, and more. For JSON, you might need to specify an external SerDe, which is available in your tutorial materials.

**11. Hive Interfaces and Tools**  
You can access Hive through tools like Zeppelin (a notebook interface) or Beeline (a basic command-line tool). Administrators use Beeline to check configurations and connections. For more advanced usage, tools like DBVisualizer let you connect to Hive and browse data visually.

**12. Running Hive Scripts and Queries**  
You can also run HiveQL scripts using the command line with hive -f script.hql. This helps automate repetitive tasks. Hive supports common SQL-like commands, but it's not a full relational database, so performance improves when data is denormalized.

## Part 6-ChatGPT

**1. Sampling and Table Commands in Hive**  
You can randomly select data from large tables using sampling. The command INSERT OVERWRITE TABLE stores the result of a query in a table. For example, using TABLESAMPLE BUCKET 3 OUT OF 32 means picking random parts (or “buckets”) of data. To view tables, you can use SHOW TABLES. The DESCRIBE commands help you understand the table's columns and types. DESCRIBE EXTENDED and DESCRIBE FORMATTED give more detailed information like partitioning and file formats.

**2. Hive Interfaces and Logs**  
Hive provides web interfaces that show system logs and queries. If the Hive UI appears, the server is working correctly. Admins can use this to see how queries are executed and find slow parts to improve performance. Another interface, the Tez Web UI, displays detailed execution plans for heavier queries (like aggregations), helping to analyze where processing time is spent. These tools are mainly for administrators but help check system health and query behavior.

**3. Using DBVisualizer and ODBC with Hive**  
DBVisualizer is a free tool that lets you connect and work with Hive databases easily. You can download it and use it to explore Hive data from outside tools. By installing an ODBC driver (like Microsoft’s), you can also connect Hive to other tools such as Excel. This setup helps integrate Hive into more familiar software environments.

**4. Working with Hive Databases and Tables in Practice**  
In the hands-on workshop, you learn how to create Hive databases and tables. You start with CREATE DATABASE IF NOT EXISTS, then list existing databases. When you create tables without the EXTERNAL keyword, they are internal tables. You can then upload data to HDFS and load it into a table using LOAD DATA INPATH. Use full notation like database\_name.table\_name to clearly refer to tables.

**5. Describing Tables and Viewing Data**  
Once data is loaded into Hive, you can view it with SELECT \* FROM table. Use DESCRIBE to check column names and types. DESCRIBE EXTENDED gives more metadata, though harder to read. DESCRIBE FORMATTED presents this information in a clearer, tabular format. These commands help you understand what data the table contains and how it is stored.

**6. Creating and Dropping External Tables**  
External tables in Hive point to data stored outside Hive. You use the EXTERNAL keyword when creating them and specify a location in HDFS. This means that even if you delete the table using DROP TABLE, the data in HDFS remains. Naming tables with “ext” helps identify them as external, making maintenance easier.

## Part 7-ChatGPT

**1. Internal vs External Tables in Hive**  
Internal tables in Hive store both the table structure and data. If you drop an internal table, the data is deleted. External tables only store the structure, while the actual data stays in HDFS. If you delete an external table, the data remains. However, you can force deletion of external table data using a special property (external.table.purge=true), but this is only recommended in development environments, not production.

**2. Partitioning in Hive (Static)**  
Partitioning helps organize and filter data efficiently. First, data is loaded into a staging (internal) table. Then, a new partitioned table is created where certain columns (e.g., "country") are removed from the table schema and added as partition columns. Data is inserted into the new table using static partitioning, where you specify the partition value manually (e.g., country='Canada'). The data is then physically stored in separate folders based on these partition values.

**3. Adding More Partitions in Hive**  
More partitions can be added by running similar insert queries with different partition values (e.g., country='United States'). You can view existing partitions using the SHOW PARTITIONS command. It's important to remember that Hive is case-sensitive, so partition values must match exactly in case. Dynamic partitioning is also possible but must be explicitly enabled and is typically used in development only.

**4. Partitioning in Spark**  
Spark also supports data partitioning and has more flexibility. Data can be loaded with options like inferring the schema from headers or customizing column names. To partition in Spark, you use the partitionBy() function when writing the data. This creates separate folders for each partition value, just like Hive. Optionally, repartition() can be used to control in-memory data distribution, but it's not required for basic partitioning.

**5. Spark Output and Parquet Format**  
Spark typically writes data in Parquet format, which is a compressed binary format that’s efficient for big data processing. When using write.partitionBy(), Spark saves the data in folders by partition key (e.g., marital\_status=Single). These Parquet files are smaller and faster to read compared to text or CSV files. Snappy compression is the default, and it's important to tell Hive to expect this format when reading Spark-generated Parquet files.

**6. Hive Reading Spark Output (Parquet Files)**  
When reading Parquet files written by Spark in Hive, you must create an external Hive table that matches the schema and specify the storage format as Parquet. Also, set the compression type to Snappy using the correct case-sensitive property. After creating the table, it appears empty until you run MSCK REPAIR TABLE, which loads the partitions from HDFS into Hive’s metadata.

**7. Optimizing Hive Query Performance**  
Before running queries in Hive on the newly loaded Parquet data, use the ANALYZE TABLE ... COMPUTE STATISTICS command. This collects metadata like row count, file size, and partition details. Without this step, queries may still work but will be slower. Once statistics are computed, Hive can optimize how queries are executed, improving performance significantly.

**8. Case Study Overview**  
A real case study involves processing zipped data using Spark, saving the output to HDFS in Parquet format, and reading it back in Hive. This demonstrates the full cycle of loading, partitioning, and querying big data. You also learn how to cache data in Spark to improve speed, normalize data, and prepare it for querying in Hive. Next week, the same case study will be explored using Trino, a much faster, interactive query engine.

## **Part 8-ChatGPT**

**1. Homework Instructions and Expectations**  
Your first homework is available on MyCourses. Before **looking at the case study or solution**, try solving the exercise on your own. Use the tutorials to guide you if needed. Only refer to the **case study** if you are truly stuck. The homework involves reading data from HDFS using Hive—nothing beyond what's already covered in tutorials.

**2. Overview of the Assignment Tasks**  
The homework is simple: you will use Spark’s Dataset API with Scala (not Python or Java). The task is based on analyzing restaurant inspections using two CSV files. One file has restaurant information (ID, name, address, cuisine), and the other has inspection data (ID, date, code, description, grade, score). First, load these files into HDFS, **then load them into Spark and cache the data for better performance.**

**3. Dataset API Queries**  
In the first part, use Scala and Spark Dataset API to complete 4 queries:

1. Show the first 10 rows of each dataset.
2. Count total rows.
3. Find top 10 most common cuisine types.
4. Find top 10 most frequent violation codes.  
   Make sure to define proper case classes and import the necessary libraries for handling dates and SQL.

**4. SQL API Queries**  
In the second part, use SQL queries in Spark to complete 3 tasks:

1. Find the top 5 restaurants with the most inspections.
2. **Identify top 5 cuisine types with the most violation descriptions.**
3. Calculate the average inspection score per borough.  
   Again, these are simple and based on what you’ve learned.

**5. Submission Guidelines**  
You must submit **two things**:

1. A Word document named Summer2025\_Homework1 with your full name, code snippets (limit to 5–10 rows), labeled tasks, and explanations (in English or French).
2. A copy of your Zeppelin notebook.  
   Do not zip the files, and do not email your submission—it must go through MyCourses only.

**6. Late Policy and Plagiarism Warning**  
Homework is due **Wednesday, June 4, at midnight**. Submissions after that will lose 15% **per day**. No submissions = grade 0. If AI tools like ChatGPT, Copilot, or DeepSeek are used, your grade will be cut by **50%**. Don’t copy from or share with classmates—this will be considered plagiarism, and both students will be penalized.

**7. Technical Notes on Accessing Data**  
You don’t need to download any data. The two required CSV files (restaurant.csv and restaurant\_inspection.csv) are already on the sandbox in the /home/training/data/restaurant folder. The first file is 2MB and the second is 78MB—Spark can handle them easily. Just point to the correct path when loading them in Spark.

**8. Access and Tools Support**  
If you're using a Windows computer, use **MobaXterm** to connect to the sandbox. If you're on a Mac, you can use **Tabby** or any SSH client. You don’t need to go into the Ubuntu interface—just use the provided directory to load your files directly in Zeppelin.

**9. Final Notes on Document and Notebook Submission**  
Your Word document is for explanations and must include your name. Zeppelin is for your full code and results. The Word file lets instructors see your thinking, while the Zeppelin notebook helps verify the code. Both are required—don’t just submit one. If your Word doc and notebook don’t match, the notebook will be used for grading.