Foster Mosden
Big Data - Final Exam

Question 1

1. Hadoop is both scalable and cost effective. The distributed structure of the Hadoop architecture and file system allows horizontal scaling using commodity hardware. A secondary advantage of the distributed architecture is that data management and computing can be performed in a parallel manner, improving performance and maximizing hardware usage. As well, Hadoop is open source, which eliminates the need for any licensing fees to be paid.
2. For one, Hive is great for batch processing, but this optimization makes it bad for complex join commands and transactional workloads. An RDS is better suited for these tasks. For two, Hive is not great for low latency scenarios, so in cases where quick simple querying is required, Hive may add unnecessary load time for users. Finally, Hive has a more flexible approach to schemas than a traditional relational database. So if schema rigidity is an important component, or benefits the workload, then Hive may not be optimal.
3. In a NoSQL system, the combinations of CAP properties that are most important are Availability and Partition Tolerance (AP) (the most important), and Consistency and Partition Tolerance (CP). AP ensures that your data is available, regardless of whether there are failures in or in between your various nodes. This is obviously a great choice for when constant, uninterrupted availability is paramount. On the other hand, CP is a great choice for when you want the system to certify the consistency of data across various nodes, and have the data and the system be tolerant of failure in or between nodes. Overall, the system and use case will dictate the most important parts of CAP theorem, but these are the two most important combinations when it comes to NoSQL.
4. Horizontal scaling is a method of expanding a database's capacity by adding more hardware and resources by means of adding more machines, or nodes. This is different from "vertical scaling", which involves increasing the power of a singular machine by increasing its capability via hardware upgrades. Horizontal scaling is great for databases because it not only increases storage capacity and distributed computing power, but also enables features like fault tolerance and increased demand from various users at once.

5. As Hadoop is a distributed computing and file system, data locality refers to processing data on the same node where it is stored, which helps to minimize the unnecessary movement of data across the network during computing. This is especially important for the processing of big data, where the datasets are massive -- in these cases, moving data across nodes for processing would be an immense waste of computing power, network resources, and memory use.
6. The two key components of Hadoop are the Hadoop Distributed File System (or HDFS), and MapReduce (MR). HDFS is the storage system that Hadoop uses to store and manage extremely large files across a cluster of commodity hardware. Files are broken into blocks that are distributed across various nodes in the cluster, which allows for parallel processing and fault tolerance. MapReduce is the engine that Hadoop uses for distributed processing of big data. Developers can use MR in their programs too. The MR model “maps” the data first, processing and transforming it in parallel across nodes. Then, it “reduces” it by aggregating the data (and applying one more layer of processing if needed). MapReduce can be run in multiple layers in order to continuously collect and process the data until the desired result is achieved.
7. Apache Sqoop is the tool designed to import relational databases into HDFS. The name Sqoop comes from the conjunction of SQL and Hadoop. There are two good reasons to use Sqoop for the task specified here. For one, Sqoop has an incremental import feature, which would allow it to identify and transfer only data that has newly appeared since the last import occurred. Secondly, Sqoop uses parallel processing, like much of the Hadoop ecosystem, to support parallel data transfers. By distributing the workload, the data transfer process can be accelerated. This also allows for more scalable importing.
8. A left semi join in Hive behaves more like a filter than a join. In a left semi join, only rows from the left table that have matching keys in the right table are maintained. It doesn’t modify the structure of either table like a filter would, but it selects rows based on whether they have matching keys. On the other hand, an inner join includes the rows that have matching keys in both the left and the right table, combining the structure of both tables. A left semi join doesn’t

actually include any columns from the right table. An outer join on the other hand joins the non-matching rows from one or both of the tables, and NULL values are used to fill in gaps where data is missing from the other table. The left semi join doesn't include any non-matched rows.

9. Hive supports three main data types; array, map, and struct. An array is an ordered collection of data of the same type. Maps are collections of key-value pairs. Each key and value can be of any of the data types. Finally, struct groups together fields of differing data types into a single entity. It's similar to a tuple in Python.
10. One reason to use complex data types in Hive is for hierarchical data. This allows users to represent and store and represent data in a way that reflects the natural organization, which can make it easier to work with. The other reason is the flexibility of the data types. When schemas are constantly evolving, using one of these Hive data types allows you to work with varying structures. This is especially useful in big data applications. Finally, there is efficient storage and querying for these data types. Because there are various complex structures to serve specific needs, each one is highly optimized for its purpose. In a distributed computing environment, like Hadoop, this is crucial to maintaining performance. Overall, these data types in Hive enable the platform's ability to deal with diverse, and even nested data structures. It also allows schemas to change, and keeps systems running efficiently. In a big data environment, this is all crucial to success.
11. The sentences function in Hive splits a string into arrays of sentences, where each sentence itself is an array of words. In this case, for each sentence, it would strip the quotes from any unnecessary punctuation, such as quotation marks, commas, and periods, and then it would store each work individual as a string in an array. Each array that represents all of the words from a sentence would be stored in an array of sentences.
12. For one, Spark uses in-memory processing, which means intermediate data can be cached and therefore accessed quickly and easily. On the other hand, Hive uses disk based processing --

reading data from disks, and then writing data to disks in between phases. This takes much longer. For two, Spark uses immutable RDDs (Resilient Distributed Datasets), which can be used for iterative processing. On the other hand, Hive uses MapReduce, which again, due to disk based processing, can be slower for iterative tasks. Finally, Spark uses DAG, or Directed Acyclic Graph, as the execution engine. This minimizes data movement across the cluster, which again, minimizes processing time. Since Hive uses MR in HDFS, disk IO and data movement has a great impact on processing times.

13. Although SparkSQL has faster query processing, both it and HiveQL have their own optimal use cases. For one, HiveQL would be a better choice for small scale or simple processes, as the overhead involved with creating and maintaining a Spark cluster may outweigh the benefits. For two, if the data is stored in HDFS, using HiveQL may make more sense due to its greater integration with the Hadoop ecosystem. Finally, if there is already a solid Hive infrastructure in place, it might make more sense to stick with Hive than transition, especially if you don't expect massive performance benefits from Spark.

14. When stream processing in PySpark, a tumbling window divides the stream into time intervals that are fixed and non-overlapping. On the other hand, a sliding window does allow for windows to overlap, which enables data to be considered across consecutive intervals.

15. In Spark, a watermark is used to manage event time in data streams, which is paricularly helpful in handling data which arrives late. The watermark sets a threshold for the amount of time beyond which the data is considered as arriving too late, and therefore no longer relevant to input into the processing stream. Let's say we're tracking the temperature of a 18 wheeler truck that has refrigerated food inside. A watermark can be used to ignore temperature readings that arrive after a preset threshold, which prevents outdated data on the temperature from affecting how the AC unit runs.

Question 2:

2.1 MySQL query for loading the dataset into RDS:

```
USE Final;
DROP TABLE IF EXISTS ad_data;
CREATE TABLE ad_data (
      id LONGTEXT,
      click INT,
      `hour` TEXT,
      C1 INT,
      banner_pos INT,
      site_id VARCHAR(255),
      site_domain VARCHAR(255),
      site_category VARCHAR(255),
      app_id VARCHAR(255),
      app_domain VARCHAR(255),
      app_category VARCHAR(255),
      device_id VARCHAR(255),
      device_ip VARCHAR(255),
      device_model VARCHAR(255),
      device_type INT,
      device_conn_type INT,
      C14 INT,
      C15 INT,
      C16 INT,
      C17 INT,
      C18 INT,
      C19 INT,
      C20 INT,
      C21 INT
);
LOAD DATA LOCAL INFILE 'C:/Users/foste/Downloads/ads.csv'
INTO TABLE ad_data
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;

UPDATE ad_data
SET hour = STR_TO_DATE(hour, '%y%m%d%H');
```

2.1 (cont): Sqoop command for importing the data from RDS into HDFS:

```
sqoop import --connect
jdbc:mysql://final-gh.cqjkbf4mswjw.us-east-1.rds.amazonaws.com:3306/F
inal \
--username admin \
--password poopy2000 \
--table ad_data \
--target-dir /user/Hadoop/ads \
--fields-terminated-by ',' \
--lines-terminated-by '\n' \
-m 1
```

Note: Since the table doesn't have a primary key (id repeats), I had to use "-m 1" to only use one mapper and thus allow the table to be imported without a primary key.

2.2 + 2.3: Hive queries to create new "ads" table, then load the data from HDFS into the table:

```
USE default;
CREATE TABLE ads (
      id STRING,
      click INT,
      hour STRING,
      C1 INT,
      banner_pos INT,
      site_id STRING,
      site_domain STRING,
      site_category STRING,
      app_id STRING,
      app_domain STRING,
      app_category STRING,
      device_id STRING,
      device_ip STRING,
      device_model STRING,
      device_type INT,
      device_conn_type INT,
      C14 INT,
      C15 INT,
      C16 INT,
      C17 INT,
```

```
      C18 INT,
      C19 INT,
      C20 INT,
      C21 INT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
LOAD DATA INPATH '/user/Hadoop/ads/' INTO TABLE ads;
```

2.4: HiveQL query to show the to 10 app_ids by CTR:

```
USE default;

CREATE TABLE temp_ads AS
SELECT
      app_id,
      COUNT(*) AS impressions,
      SUM(click) AS clicks
FROM
      ads
GROUP BY
      app_id;

SELECT
      app_id,
      impressions,
      clicks,
      clicks / impressions AS ctr
FROM
      temp_ads
WHERE
      impressions >= 10
ORDER BY
      ctr DESC
LIMIT 10;

DROP TABLE IF EXISTS temp_ads;
```

```
95827a92        12      12      1.0
89bdfe24        23      12      0.5217391304347826
74afe595        10      5       0.5
af163a84        12      6       0.5
9c13b419        297     130     0.4377104377104377
27550a3c        16      7       0.4375
0639409a        12      5       0.4166666666666667
4e02fbd3        36      14      0.3888888888888889
a37bf1e4        41      14      0.34146341463414637
d7a5d468        12      4       0.3333333333333333
Time taken: 2.448 seconds, Fetched: 10 row(s)
```

2.5: Pig script used to complete the same query as I did in Hive

```
ads = LOAD 'hdfs:///user/Hadoop/ads' USING PigStorage(',')
      AS (id:chararray, click:int, hour:chararray, C1:int,
banner_pos:int, site_id:chararray, site_domain:chararray,
site_category:chararray, app_id:chararray, app_domain:chararray,
app_category:chararray, device_id:chararray, device_ip:chararray,
device_model:chararray, device_type:int, device_conn_type:int,
C14:int, C15:int, C16:int, C17:int, C18:int, C19:int, C20:int,
C21:int);
filtered_ads = FILTER ads BY click is not null AND app_id is not
null;
grouped_ads = GROUP filtered_ads BY app_id;
agg_data = FOREACH grouped_ads GENERATE
                group AS app_id,
                COUNT(filtered_ads) AS impressions,
                SUM(filtered_ads.click) AS clicks;
ctr_data = FOREACH agg_data GENERATE
                app_id,
                impressions,
                clicks,
                (float)clicks / (float)impressions AS ctr;
filtered_ctr_data = FILTER ctr_data BY impressions >= 10;
ordered_ctr_data = ORDER filtered_ctr_data BY ctr DESC;
top_10_apps = LIMIT ordered_ctr_data 10;
DUMP top_10_apps;
```

```
(95827a92,12,12,1.0)
(89bdfe24,23,12,0.5217391)
(af163a84,12,6,0.5)
(74afe595,10,5,0.5)
(9c13b419,297,130,0.43771043)
(27550a3c,16,7,0.4375)
(0639409a,12,5,0.41666666)
(4e02fbd3,36,14,0.3888889)
(a37bf1e4,41,14,0.34146342)
(d7a5d468,12,4,0.33333334)
```

2.6: SparkSQL code to repeat the same query from 2.4:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("CTRAnalysis").getOrCreate()

from pyspark.sql.types import StructType, StructField, StringType,
IntegerType

schema = StructType([
      StructField("id", StringType()),
      StructField("click", IntegerType()),
      StructField("hour", StringType()),
      StructField("C1", IntegerType()),
      StructField("banner_pos", IntegerType()),
      StructField("site_id", StringType()),
      StructField("site_domain", StringType()),
      StructField("site_category", StringType()),
      StructField("app_id", StringType()),
      StructField("app_domain", StringType()),
      StructField("app_category", StringType()),
      StructField("device_id", StringType()),
      StructField("device_ip", StringType()),
      StructField("device_model", StringType()),
      StructField("device_type", IntegerType()),
      StructField("device_conn_type", IntegerType()),
      StructField("C14", IntegerType()),
      StructField("C15", IntegerType()),
      StructField("C16", IntegerType()),
      StructField("C17", IntegerType()),
```

```
      StructField("C18", IntegerType()),
      StructField("C19", IntegerType()),
      StructField("C20", IntegerType()),
      StructField("C21", IntegerType())
])

ads_df = spark.read.csv("/user/Hadoop/ads", header=False,
schema=schema, sep=",")

ads_df.createOrReplaceTempView("ads")

query_result = spark.sql("""
      SELECT
      app_id,
      COUNT(*) AS impressions,
      SUM(click) AS clicks,
      SUM(click) / COUNT(*) AS ctr
      FROM
      ads
      GROUP BY
      app_id
      HAVING
      impressions >= 10
      ORDER BY
      ctr DESC
      LIMIT 10
""")

query_result.show()
```

app_id	impressions	clicks	ctr
95827a92	12	12	1.0
89bdfe24	23	12	0.5217391304347826
af163a84	12	6	0.5
74afe595	10	5	0.5
9c13b419	297	130	0.4377104377104377
27550a3c	16	7	0.4375
0639409a	12	5	0.4166666666666667
4e02fbd3	36	14	0.3888888888888889
a37bf1e4	41	14	0.34146341463414637
d7a5d468	12	4	0.3333333333333333

2.7: Discussion:

PySpark demonstrated the fastest execution at 12 seconds, followed by Hive at 16 seconds, while Pig using MapReduce took considerably longer at 1 minute and 56 seconds. The notable differences in performance are likely attributed to the underlying processing frameworks. PySpark leverages Spark's in-memory computing, allowing for faster data manipulations, and much faster read times from the table. Hive, also utilizing Spark, benefits from optimized query execution plans. In contrast, Pig relies on MapReduce, which involves more disk I/O and intermediate data storage steps, contributing to its longer processing time. The insights underscore the advantages of in-memory processing and optimized execution plans in PySpark and Hive, highlighting their superior performance compared to the traditional MapReduce-based approach employed by Pig. (note that I had to run my Pig script with “pig -x mapreduce -f ctr.pig”)

Question 3:

3.1: S3 URI: s3://fm-final/json_award.json

3.2: Using Spark Session to load the dataset and register it as a temp table:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("LoadJSON").getOrCreate()

aws_access_key_id = "CENSORED"
aws_secret_access_key = "CENSORED"
s3_path = "s3://fm-final/json_award.json"

spark._jsc.hadoopConfiguration().set("fs.s3a.access.key",
aws_access_key_id)
spark._jsc.hadoopConfiguration().set("fs.s3a.secret.key",
aws_secret_access_key)

df = spark.read.json(s3_path)

# Register the DataFrame as a temporary table
df.createOrReplaceTempView("award_table")
```

3.3: Using SparkSQL to identify multi-time Nobel winners:

```
# flatten the laureates array as a new dataframe
flattened_df = df.selectExpr("explode(laureates) as laureate")

# extract what we need from JSON
flattened_df = flattened_df.select(
      "laureate.knownName.en",  # Assuming English names are stored
in "knownName"
      "laureate.id"
)

# group by name, count # awards
result = flattened_df.groupBy("en", "id").count().filter("count >=
2")
```

```
result.show()
```

```
+----------------+---+-----+
|              en| id|count|
+----------------+---+-----+
|Frederick Sanger|222|    2|
|   Linus Pauling|217|    2|
|    John Bardeen| 66|    2|
|            null|482|    3|
|            null|515|    2|
|     Marie Curie|  6|    2|
+----------------+---+-----+
```

3.4: Using SparkSQL to count the frequency of words in motivation text:

```
from pyspark.sql.functions import explode, split, lower
from pyspark.sql.types import StringType

flattened_df = df.selectExpr("explode(laureates) as laureate")

motivation_df = flattened_df.select("laureate.motivation.en")

tokenized_df = motivation_df.select(explode(split(lower("en"),
"\\W+")).alias("word"))

filtered_df = tokenized_df.filter("word != ''")

word_count_df = filtered_df.groupBy("word").count()

result = word_count_df.orderBy("count", ascending=False).limit(5)

result.show()
```

```
+----+-----+
|word|count|
+----+-----+
|  of| 1206|
| the| 1169|
| for| 1041|
| and|  606|
|  in|  429|
+----+-----+
```

Question 4:

4.1: In predicting Google Store sales using the train_v2.csv dataset, we'll need a comprehensive strategy. Initial steps involve utilizing Apache Spark for efficient distributed processing and machine learning frameworks like TensorFlow or PyTorch for predictive modeling, emphasizing scalability. Given the data's complexity, including JSON blobs and varied depth, meticulous data preprocessing using Pandas and SQL is crucial to extract the relevant information. To address challenges highlighted in articles 2 and 3, a thoughtful feature engineering approach will focus on eliminating deficiencies in raw process data, handling missing values, and capturing both deterministic and statistical aspects of user behavior. This strategy underscores the need for smart big data utilization, encompassing technology, data engineering, and model development practices.

To understand the Google Store customer base and forecast sales, a multifaceted analysis of user behavior, transaction patterns, and key factors like device specifications and geographic information is necessary. By leveraging big data analytics and machine learning models, my approach seeks actionable insights contributing to sales growth. Acknowledging the potential insights and challenges associated with big data, as highlighted in provided articles, underscores the importance of smart data utilization.

Predicting Google Store sales involves leveraging variables like user demographics and historical transaction data. While considering factors such as channel grouping and device specifications, caution is exercised against sole reliance on big data, aligning with Nielsen's recommendation to incorporate panel data for stability. Employing regression-based or machine learning techniques like XGBoost allows flexibility to capture non-linear relationships. The critical focus remains on meticulous data preprocessing, encompassing handling missing values, feature engineering, and addressing biases, ensuring model accuracy and robustness.

The outlined approach for predicting Google Store sales and making customer targeting decisions faces challenges in handling the complexities of big data, ensuring interpretability and addressing

potential biases in machine learning models, and navigating uncertainties associated with evolving customer preferences. Additionally, integrating varied data sources, including JSON blobs, requires meticulous data preprocessing to mitigate inconsistencies, and the cautious incorporation of panel data highlights challenges in ensuring data quality and representativeness for advertising decisions.

4.2: To design a Hive database schema for the Google Store business incorporating insights from Adobe and Talend, I would create three main tables: "Customer," "Transactions," and "Marketing." The "Customer" table would include customer details for personalized experiences, leveraging Adobe's real-time customer experience platform. The "Transactions" table would capture sales data, aligning with Q4.1's focus on predicting product sales. Lastly, the "Marketing" table would incorporate datasets from multiple clouds, reflecting Talend's emphasis on merging regional marketing efforts. Key columns would include customer identifiers in the "Customer" table, transaction-related details in the "Transactions" table, and marketing-related information in the "Marketing" table. This schema would facilitate a holistic approach to customer experiences, transactions, and marketing insights within the Google Store.

To enhance customer experience and revenue, three OLAP star schemas can be designed based on Adobe's suggested ways to improve customer experience. The first schema, "Customer Loyalty," would have dimensions like customer segments, loyalty programs, and interaction channels, connected to a central fact table containing loyalty metrics. The second schema, "Real-time Experiences," would feature dimensions such as user interactions, platform capabilities, and touchpoints, linked to a fact table capturing real-time customer experiences. The third schema, "Marketing Success," would include dimensions like marketing channels, global messaging, and success metrics, connected to a fact table aggregating marketing performance data. These star schemas provide a foundation for comprehensive analytics, facilitating data-driven strategies to enhance customer satisfaction and drive revenue growth.