Question1:

Hive is high-level abstraction on top of MapReduce and HDFS, using a SQL-alike language called HiveQL. It directly queries on HDFS, which is faster than pig.

Impala see MapReduce as the bottleneck because reading speeds from mapper to reducer. So it directly run SQL query on data nodes, which makes it faster than Hive.

Spark shares a similar concept as Impala, but it not only manipulate or query but also perform computation directly on data nodes utilizing excessive RAM/CPU powers.

1a.

Hive data load in:

```
CREATE EXTERNAL TABLE trip
(VendorID INT,
pickup_datetime STRING,
dropoff_datetime STRING,
store_and_fwd_flag STRING,
RatecodeID INT,
PULocationID INT,
DOLocationID INT,
passenger_count INT,
trip_distance FLOAT,
fare_amount FLOAT,
extra INT,
mta_tax FLOAT,
tip_amount FLOAT,
tolls_amount FLOAT,
ehail_fee FLOAT,
improvement_surcharge FLOAT,
total_amount FLOAT,
payment_type INT,
trip_type INT
```

```
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
tblproperties ("skip.header.line.count"="1");

LOAD DATA INPATH 's3://smokeeveryday/data420/tripdata.csv' INTO TABLE trip;
```

Databases > default > trip

Query | Import | Drop

Overview | Sample (100) | Details

	trip.vendorid	trip.pickup_datetime	trip.dropoff_datetime	trip.store_and_fwd_flag	trip.ratecodeid	trip.pulocationid	trip.dolocationid	trip.passenger_
1	2	1/1/17 0:01	1/1/17 0:11	N	1	42	166	1
2	2	1/1/17 0:03	1/1/17 0:09	N	1	75	74	1
3	2	1/1/17 0:04	1/1/17 0:12	N	1	82	70	5
4	2	1/1/17 0:01	1/1/17 0:14	N	1	255	232	1
5	2	1/1/17 0:00	1/1/17 0:18	N	1	166	239	1
6	2	1/1/17 0:00	1/1/17 0:13	N	1	179	226	1
7	2	1/1/17 0:02	1/1/17 0:26	N	1	74	167	1
8	2	1/1/17 0:15	1/1/17 0:28	N	1	112	37	1
9	2	1/1/17 0:06	1/1/17 0:11	N	1	36	37	1
10	2	1/1/17 0:14	1/1/17 0:28	N	1	127	174	5

Spark data load in:

```
In [1]: trip = spark.read.load("s3://smokeeveryday/data420/tripdata.csv", format="csv", sep=",", inferSchema="true",header="true")
```

Starting Spark application

ID	YARN Application ID	Kind	State	Spark UI	Driver log	Current session?
3	application_1605159892169_0005	pyspark	idle	Link	Link	✓

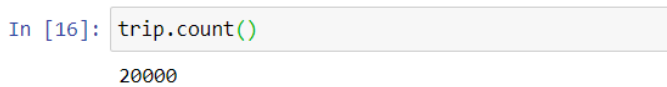
SparkSession available as 'spark'.

```
In [2]: trip
```

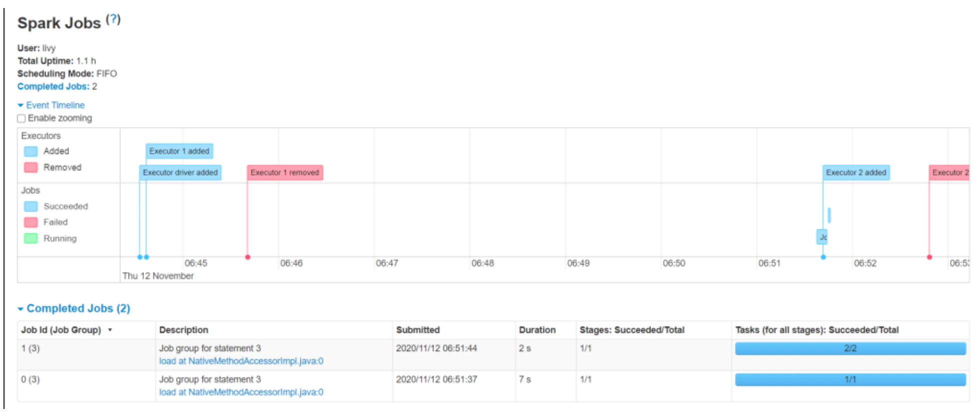
```
DataFrame[VendorID: int, lpep_pickup_datetime: string, lpep_dropoff_datetime: string, store_and_fwd_flag: string, RatecodeID: i
nt, PULocationID: int, DOLocationID: int, passenger_count: int, trip_distance: double, fare_amount: double, extra: double, mta_
tax: double, tip_amount: double, tolls_amount: double, ehail_fee: string, improvement_surcharge: double, total_amount: double,
payment_type: int, trip_type: int]
```

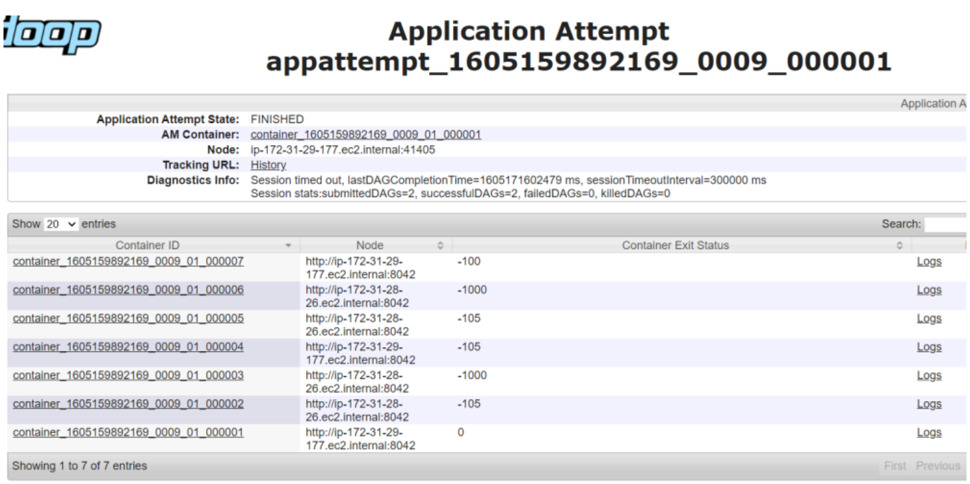
1b.


3/3
29 SELECT count(*)
30 FROM df
INFO : Map 1: 1/1 Reducer 2: 1/1
application_1605159892169_0010
INFO : Completed executing command(queryId=hive_20201112090835_fe0ec487-053a-4
4.486 seconds
INFO : OK
Query History
Saved Queries
Query Builder
Results (1)
_c0
1 20000


In [16]: trip.count()
20000

1c.


Spark Jobs (?)
User: livy
Total Uptime: 1.1 h
Scheduling Mode: FIFO
Completed Jobs: 2
Event Timeline
Enable zooming
Executors
Added
Removed
Jobs
Succeeded
Failed
Running
Executor 1 added
Executor driver added
Executor 1 removed
Executor 2 added
06:45
06:46
06:47
06:48
06:49
06:50
06:51
06:52
Thu 12 November
Completed Jobs (2)
Job Id (Job Group)
Description
Submitted
Duration
Stages: Succeeded/Total
Tasks (for all stages): Succeeded/Total
1 (3)
Job group for statement 3
load at NativeMethodAccessorImpl.java:0
2020/11/12 06:51:44
2 s
1/1
2/2
0 (3)
Job group for statement 3
load at NativeMethodAccessorImpl.java:0
2020/11/12 06:51:37
7 s
1/1
1/1


Application Attempt appattempt_1605159892169_0009_000001
Application Attempt State: FINISHED
AM Container: container_1605159892169_0009_01_000001
Node: ip-172-31-29-177.ec2.internal:41405
Tracking URL: History
Diagnostics Info: Session timed out, lastDAGCompletionTime=1605171602479 ms, sessionTimeoutInterval=300000 ms
Session stats:submittedDAGs=2, successfulDAGs=2, failedDAGs=0, killedDAGs=0
Show 20 entries
Search:
Container ID
Node
Container Exit Status
container_1605159892169_0009_01_000007 http://ip-172-31-29-177.ec2.internal:8042 -100 Logs
container_1605159892169_0009_01_000006 http://ip-172-31-28-26.ec2.internal:8042 -1000 Logs
container_1605159892169_0009_01_000005 http://ip-172-31-28-26.ec2.internal:8042 -105 Logs
container_1605159892169_0009_01_000004 http://ip-172-31-29-177.ec2.internal:8042 -105 Logs
container_1605159892169_0009_01_000003 http://ip-172-31-28-26.ec2.internal:8042 -1000 Logs
container_1605159892169_0009_01_000002 http://ip-172-31-28-26.ec2.internal:8042 -105 Logs
container_1605159892169_0009_01_000001 http://ip-172-31-29-177.ec2.internal:8042 0 Logs
Showing 1 to 7 of 7 entries
First Previous

Question2:

Marketing:

In marketing, some problems Coke might face are ad campaign analysis, digital marketing, market researching and segmentation. The data collection processes are relatively easy as well though daily transactional logs and internet cookies, where data come in as tabular form. Unlike retail stores, Coke don't have to update the data once after load in. So, the whole data discovery and ingestion process are relatively easy, stable/fixed and structured. Also, most of these data are open source or second-hand, which doesn't require extra assurance of data security. Meanwhile, most of those problems could be solved through simple queries and statistical methods, and won't require heavy modeling, etc. So, Impala could be a very good choice. Because, we have a well-defined and relatively fixed schema, and the data job could be mostly solved by simple queries and OLAP/BI, where Impala is extremely efficient and easy to communicate among teams as using standard SQL queries.

Operation:

In operation, some problems Coke might face are logistics, costs and productions, etc. Those problems involve a lot of real time analysis and optimizations, which are heavy on computations. The data sources can be both internal data and outsource second-hand, and schema/columns won't scale frequently or significantly, but there can be streaming data coming in. As doing all the analytical jobs, we don't expect to loading the whole dataset or processing data centers across the world. The machine learning and real time dashboard can totally be calculated on the data nodes and reports only outputs to clients to eliminate bottleneck, where Spark comes in handy and efficient. So the computations can be much faster, and easily form real-time outputs and diagrams that can be effectively communicated among the teams. The SQL/Python alike language are easy to communicate as well.

Finance:

In finance, some problems Coke might face are accounting and risk managements. Where the financial data come in with a relatively fixed format/structure, and the data usually come from internal sources, so ingestion would be relatively easy. For those problems, some degree of processing is expected. The type of work could be either modeling for risk factors or accounting report generations and optimization. Spark can handle both type of jobs and could produce visualizations along with the report. It becomes handy when doing some potential text processing in columns too, where the transaction details might be expected to be analyzed and all the system/user-defined functions are helpful. The SQL/Python alike language are easy to communicate as well.

Question3:

3a.

```
In [3]: plays = sc.textFile("s3://smokeeveryday/Plays/comedies/*.txt")

In [7]: plays.take(1)

        ["< Shakespeare -- A MIDSUMMER-NIGHT'S DREAM >"]

In [4]: counts = plays.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda x, y: x + y)

In [6]: counts.collect()

        [('by', 1280), ('Mike', 17), ('DIR>', 5157), ('', 15353), ('\tNow,', 112), ('to', 6628), ("man's", 72), ('then', 392), ('nigh
        t', 94), ('\tWith', 360), ('\tHappy', 9), ('\tAgainst', 35), ('thou,', 70), ('moonlight', 5), ('sung,', 3), ('prevailment',
        1), ('\tConsent', 3), ('may', 575), ('either', 52), ('\tOr', 201), ('\tOne', 68), ('power', 64), ('himself', 102), ("fathe
        r's", 83), ('eyes.', 20), ('am', 1000), ('worst', 19), ('shady', 1), ('lives,', 7), ('consents', 2), ('\tUpon', 98), ('<DEMET
        RIUS>\t<5%>', 1), ('\tLet', 191), ('am,', 24), ("deriv'd", 2), ('more', 751), ('fortunes', 28), ('prosecute', 1), ('\tMade',
        32), ('daughter,', 58), ('confess', 40), ('heard', 130), ('Demetrius', 12), ('we', 964), ('business', 54), ('concerns', 10),
        ('duty', 32), ('roses', 7), ('there', 470), ('rain,', 6), ('\tCould', 28), ('\tO', 239), ('shadow,', 5), ('collied', 1), ('u
        p:', 8), ('confusion.', 3), ('stands', 45), ('observance', 5), ('thee.', 146), ('strongest', 3), ('doves,', 2), ('Troyan',
        3), ('<HERMIA>\t<9%>', 4), ('catch', 24), ('bated,', 1), ('motion', 22), ('skill.', 4), ('lie,', 21), ('<HELENA>\t<11%>', 1),
        ('that?', 51), ('transpose', 1), ('taste;', 1), ("hail'd", 1), ('oaths', 23), ('\tPursue', 5), ('dear', 119), ('Snout,', 5),
        ('\tHere', 136), ('wedding-day', 1), ('actors,', 4), ('Pyramus', 19), ('merry.', 8), ('\tAnswer', 2), ('lover,', 10), ('mov
        e', 34), ('To', 50), ('rest:', 4), ('split.', 1), ('\tShall', 112), ('gates:', 1), ('\tHere,', 40), ('Thisby', 11), ('comin
        g.', 7), ("\tThat's", 65), ('\tRobin', 2), ('all.', 58), ('</ALL>', 11), ('day;', 7), ('<BOTTOM>\t<16%>', 3), ('bill', 3),
        ('fail', 20), ('<BOTTOM>\t<17%>', 1), ('<A', 83), ('Fairy', 4), ('Puck', 3), ('brier,', 3), ('pensioners', 1), ('Oberon', 5),
        ('had', 582), ('grove,', 2), ('sheen,', 1), ('<FAIRY>\t<18%>', 1), ('harm?', 2), ('\tFairy,', 1), ('jest', 29), ('Would', 2
        3), ('\tIll', 2), ('</OBERON>', 29), ('\tCome', 89), ('furthest', 6), ('bouncing', 1), ('\tGlance', 1), ('\tDidst', 15), ('Pe
        rigouna,', 1), ('mead,', 1), ('fountain,', 2), ('flock;', 1), ('mortals', 3), ('washes', 2), ('progeny', 2), ('womb', 6), ('r
        ound,', 6), ('longer', 37), ('injury.', 7), ('promontory,', 1), ('once:', 3), ("\tI'll", 306), ('conference.', 5), ('chase;',
        2), ('mischief.', 4), ('\tFare', 13), ('bank', 4), ('nodding', 1), ('violet', 2), ('wide', 19), ('\tFear', 24), ('coats,',
```

3b.

```
In [10]: from pyspark.sql.functions import col
         from pyspark.sql import Row
```

```
In [108]: title = ["A Midsummer-Night's Dream","All's Well That Ends Well","As You Like It",
                   "Cymbeline","Love's Labour's Lost","Measure for Measure","Much Ado about Nothing",
                   "Pericles Prince of Tyre","The Comedy of Errors","The Merchant of Venice",
                   "The Merry Wives of Windsor","The Taming of the Shrew","The Tempest",
                   "The Two Gentlemen of Verona","The Winter's Tale","Troilus and Cressida",
                   "Twelfth Night or What You Will"]
```

```
In [109]: dict = {}
```

```
In [110]: for i in title:
              path = "".join(["s3://smokeeveryday/Plays/comedies/",i,".txt"])
              play = sc.textFile(path)
              df = play.map(lambda r: Row(r)).toDF(["line"])
              love = df.filter(col("line").like("%love%"))
              count = love.count()
              dict[i] = count
```

```
In [111]: import operator
          dict = sorted(dict.items(), key=operator.itemgetter(1), reverse=True)
```

```
In [112]: dict
```

```
[('The Two Gentlemen of Verona', 163), ("A Midsummer-Night's Dream", 141), ('As You Like It', 122), ("Love's Labour's Lost", 9
9), ('Much Ado about Nothing', 94), ('Twelfth Night or What You Will', 88), ('Troilus and Cressida', 78), ('The Taming of the S
hrew', 69), ("All's Well That Ends Well", 66), ('The Merchant of Venice', 63), ('The Merry Wives of Windsor', 46), ('Cymbelin
e', 36), ("The Winter's Tale", 36), ('Pericles Prince of Tyre', 30), ('Measure for Measure', 29), ('The Comedy of Errors', 17),
('The Tempest', 17)]
```