PROJECT PROPOSAL

- **Project Topic:** Derive Insight from data using Big Data SQL tools
- **Objective:** Modern systems produce plentiful and varied log data. Big data is the set of technologies created to store, analyze, and manage heterogeneous datasets, a macro-tool created to mine the data and identify patterns in the chaos of this explosion in information to design smart solutions.
- We plan to analyze the Fire Incidents data gathered by the Fire Department, San Francisco using Big Data technologies to ingest, curate, transform and use techniques to explore some interesting data and derive insights on fire incidents from the dataset.
- **Data set Source:** Fire Incidents data collected by the Fire Department, San Francisco
  https://data.sfgov.org/Public-Safety/Fire-Department-Calls-for-Service/nuek-vuh3
  Fire Calls-For-Service includes all fire unit's responses to calls. Each record includes the call number, incident number, address, unit identifier, call type, and disposition.
  This data has the summary of fire call services provided by the fire department. Each incident record has details of the incident date and location, incident number, call number, Alarm date, Arrival date and time at the scene of the call, Primary situation, Action was taken, the damage it causes, counts the loss of life, number of casualties and fatalities and the fire department's response.
- **Proposed Approach: Tools and Technologies**
  (Python notebooks, Spark SQL /Apache Spark 2.0's DataFrame APIs, Apache Hive)
  To implement the project, we planned to use Databricks which is a Cloud-based Data platform powered by Apache Spark. It primarily focuses on Big Data Analytics and Collaboration with Databricks' Machine Learning Runtime, Managed ML Flow, and Collaborative Notebooks.
  We will analyze data and based on that provide observations and case studies of real data sets and demonstrate Spark's power and flexibility for enabling insightful data mining searches.

1. Abstract:

This project will help us analyze Fire Department Calls for Service datasets using technologies such as Apache Hive, Spark, and SparkSQL. We will also be presenting an analysis that could help a fire department can gain a better understanding of the nature of fires in the area they serve and effectively present data and information to help save lives and property. Information such as the type of situation found, action taken, time of alarm, time of arrival, time completed, number of engines responding and number of

personnel responding helps which Fire departments have a legal requirement to document these incidents. The reports can be beneficial to fire departments by providing insight into the nature of fires and casualties in their jurisdiction. Typically, big data helps us in obtaining data such as the number of fires handled last year, the number of fire-related injuries, and the number of fire deaths that can be tracked. You may still question why we should go to all this trouble to analyze data. Many decisions do not require analysis, such as decisions on personnel, grievance proceedings, promotion, and even decisions on how to handle a fire. It is certainly true that fire departments can continue to operate in the same way they always have without doing a lot of analysis. The most compelling reason is that analysis gives insight into fire problems, which in turn can affect operations in the department. The analysis can also be used to identify training needs. Most firefighting training is based on a curriculum that has been in place for many years.[4][5][6].

On the other hand, there are three good reasons for looking closely at the data:

1. Gain insights into fire problems.
2. Improve resource allocation for combating fires.
3. Identify training needs.

In summary, this project will help us deal with the volume of data collected on fire incidents. By using the Big Data tools and techniques presented here, we will be able to improve our skills in collecting and analyzing data, as well as presenting the results.

About the data set:

Fire Calls-For-Service includes all fire unit's responses to calls. Each record includes the call number, incident number, address, unit identifier, call type, and disposition. All relevant time intervals are also included. Because this dataset is based on responses, and since most calls involved multiple units, there are multiple records for each call number. Addresses are associated with a block number, intersection, or call box, not a specific address.[5][6].

The dataset consists following columns:

UnitID	Battalion	ResponseDtTm	NumberofAlarms
IncidentNumber	StationArea	OnSceneDtTm	UnitType
CallType	Box	TransportDtTm	Unitsequenceincalldispatch
CallDate	OriginalPriority	HospitalDtTm	FirePreventionDistrict
WatchDate	Priority	CallFinalDisposition	SupervisorDistrict
ReceivedDtTm	FinalPriority	AvailableDtTm	NeighborhoodDistrict
EntryDtTm	ALSUnit	Address	Location
DispatchDtTm	CallTypeGroup		

Objective:

The objective of the data is to provide answers to some of the interesting questions that pose when looking into the data. One assumption throughout the project is that data on fire incidents and casualties

are available for analysis. Manual analysis is possible, but the tedious calculations quickly overwhelm the ability to perform analysis in any meaningful manner.

2. Literature Review

Big data is data that contains greater variety, arriving in increasing volumes and with more velocity. This is also known as the three Vs (**Volume**-amount of data, **Velocity-** the fast rate at which data is received, **Variety-** types of data that are available). Put simply, big data is larger, fast, and more complex data sets, especially from new data sources which cannot be processed or managed using traditional methods. It is not a single technique or a tool, rather it has become a complete subject, which involves various tools, techniques, and frameworks. But these vast volumes of data can be used to address business problems.

Big Data includes huge volume, high velocity, and an extensible variety of data. The data in it will be of three types.

- Structured data − Relational data.
- Semi-Structured data − XML data.
- Unstructured data − Word, PDF, Text, Media Logs.

Big data technologies are important in providing more accurate analysis, which may lead to more concrete decision-making resulting in greater operational efficiencies, cost reductions, and reduced risks for the business.

To harness the power of big data, you would require an infrastructure that can manage and process huge volumes of structured and unstructured data in real-time and can protect data privacy and security.

In the Traditional Approach user interacts with the application, which in turn handles the part of data storage and analysis. But when it comes to dealing with huge amounts of scalable data, it is a hectic task to process such data through a single database bottleneck.

Google solved this problem using an algorithm called MapReduce. This algorithm divides the task into small parts and assigns them to many computers, and collects the results from them which when integrated, form the result dataset.

Using the solution provided by Google, Doug Cutting and his team developed an Open-Source Project called HADOOP. Hadoop runs applications using the MapReduce algorithm, where the data is processed in parallel with others. In short, Hadoop is used to develop applications that could perform complete statistical analysis on huge amounts of data

Apache Hadoop:

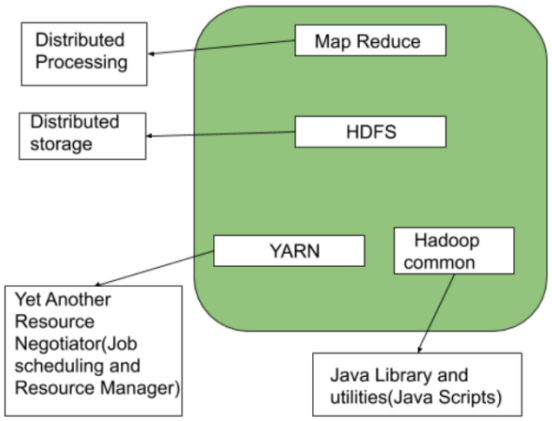
Hadoop is an Apache open-source framework written in java that allows distributed processing of large datasets across clusters of computers using simple programming models. The Hadoop framework application works in an environment that provides distributed storage and computation across clusters of computers. Hadoop is designed to scale up from a single server to thousands of machines, each offering local computation and storage.

Hadoop Architecture: consists of two major layers, the processing/Computation layer (MapReduce), and the Storage layer (Hadoop Distributed File System).

Hadoop framework also includes the following two modules –

Hadoop Common − These are Java libraries and utilities required by other Hadoop modules.

Hadoop YARN − This is a framework for job scheduling and cluster resource management.



MapReduce

MapReduce is a parallel programming model for writing distributed applications devised at Google for efficient processing of large amounts of data (multi-terabyte datasets), on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. The MapReduce program runs on Hadoop which is an Apache open-source framework.

Hadoop Distributed File System (HDFS): - (Data Ingestion)

The function of HDFS is to operate as a distributed file system designed to run on commodity hardware. HDFS is fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets and enables streaming access to file system data in Apache Hadoop.

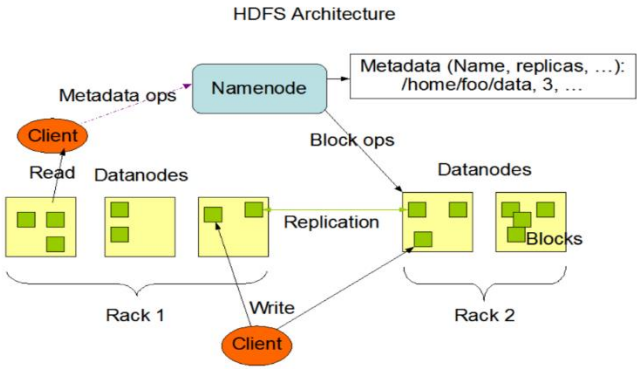
HDFS offers five core benefits when dealing with big data: Fault tolerance, speed, access to all types of data, portability, compatibility, and scalability.

HDFS Architecture:

NameNode and DataNodes

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are several Data Nodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on.

HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of Data Nodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to Data Nodes. The Data Nodes are responsible for serving read and write requests from the file system's clients. The Data Nodes also perform block creation, deletion, and replication upon instruction from the NameNode.



Data Replication:

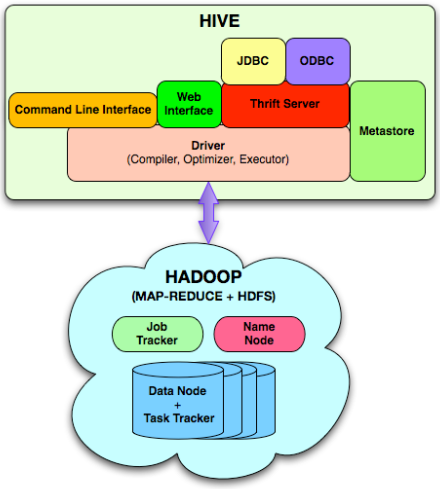
HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode.

The NameNode makes all decisions regarding the replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.

Apache Hive:

Hive is an open-source data warehousing solution built on top of Hadoop. Hive supports queries expressed in a SQL-like declarative language - HiveQL, which are compiled into map-reduce jobs that are executed using Hadoop. In addition, HiveQL enables users to plug in custom map-reduce scripts into queries. The language includes a type system with support for tables containing primitive types, collections like arrays and maps, and nested compositions of the same. The underlying IO libraries can be extended to query data in custom formats. Hive also includes a system catalog - Metastore - that contains schemas and statistics, which are useful in data exploration, query optimization, and query compilation.[2]



Hive two User Interfaces (UI) is Command Line Interface (CLI) and web UI. It also exposes an Application Programming Interface (API) like JDBC/ODBC for Business Intelligence tool/applications via Thrift server. Users or tools can submit HiveQL statements through these interfaces to the Driver. The driver manages the life cycle of the HiveQL statement by compiling, optimizing, and executing the statement. The driver first invokes Compiler to translate the statement into a query plan (or abstract syntax tree) that consists of DAG of MapReduce jobs. Then driver submits individual MapReduce jobs from the query plan to the Execution Engine. The driver needs to contact the Metastore to retrieve needed metadata from a Relational Database Management System (RDBMS)[1]

Apache Spark:

Apache Spark is a lightning-fast real-time processing framework. It does in-memory computations to analyze data in real-time. It came into the picture as Apache Hadoop MapReduce was performing batch processing only and lacked a real-time processing feature. Hence, Apache Spark was introduced as it can perform stream processing in real-time and can also take care of batch processing.

Apart from real-time and batch processing, Apache Spark supports interactive queries and iterative algorithms also. Apache Spark has its cluster manager, where it can host its application. It leverages Apache Hadoop for both storage and processing. It uses HDFS (Hadoop Distributed File system) for storage and it can run Spark applications on YARN as well.

Spark RDD:

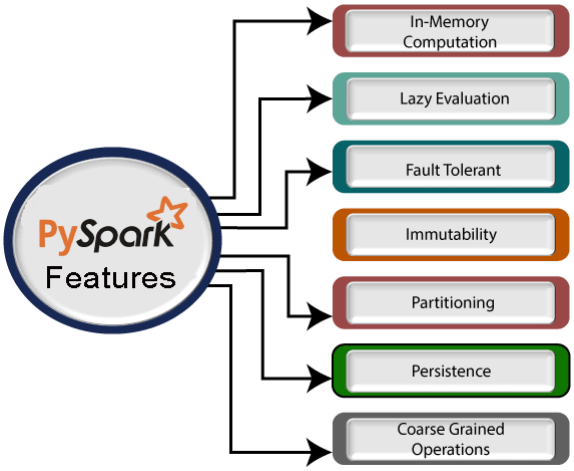
Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs − parallelizing an existing collection in your driver program or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and more efficient MapReduce operations.

PySpark is an interface for Apache Spark in Python. It not only allows you to write Spark applications using Python APIs but also provides the PySpark shell for interactively analyzing your data in a distributed environment.



SparkSQL: -Data Transformation

Apache Spark SQL is a module for structured data processing in Spark. Using the interface provided by Spark SQL we get more information about the structure of the data and the computation performed. With this extra information, one can achieve extra optimization in Apache Spark. We can interact with Spark SQL in various ways like DataFrame and the Dataset API. The Same execution engine is used while computing a result, irrespective of which API/language we use to express the computation. Thus, the user can easily switch back and forth between different APIs, it provides the most natural way to express a given transformation.

Apache Spark SQL we can use structured and semi-structured data in three ways

- To simplify working with structured data it provides DataFrame abstraction in Python, Java, and Scala. **DataFrame** is a distributed collection of data organized into named columns. It provides a good optimization technique.
- The data can be read and written in a variety of structured formats. For example, JSON, Hive Tables, and Parquet.

- Using SQL we can query data, both from inside a Spark program and from external tools. The external tool connects through standard database connectors (JDBC/ODBC) to Spark SQL.

The best way to use Spark SQL is inside a Spark application. This empowers us to load data and query it with SQL

SparkSQL Architecture:

SQL Language API − Spark is compatible with different languages and Spark SQL. It is also, supported by these languages- API (python, scala, java, HiveQL).

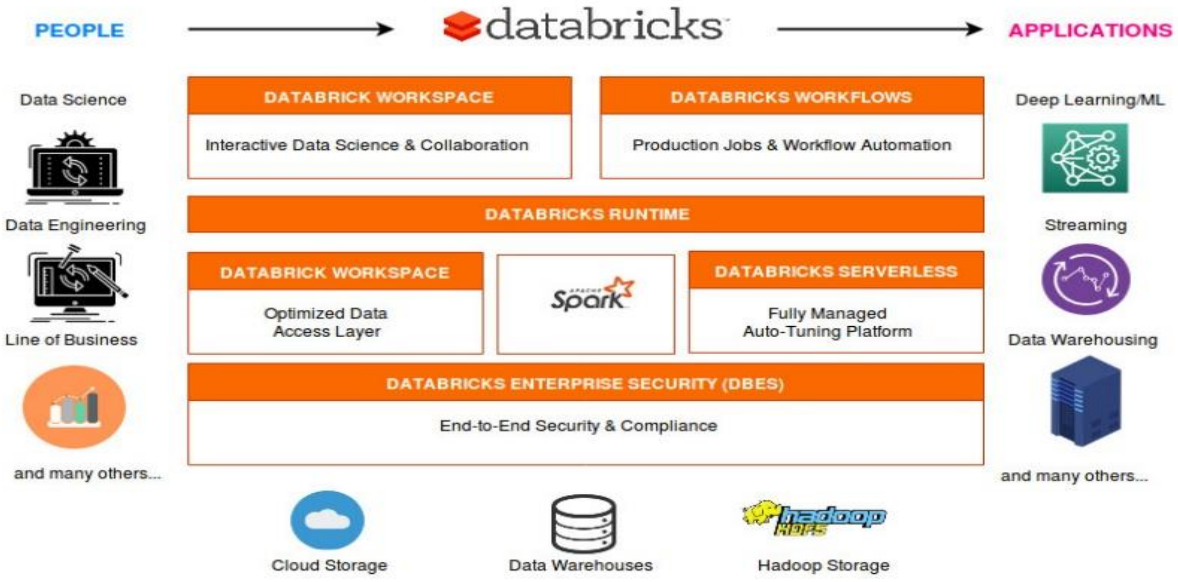
SQL Data Sources − Data Sources for Spark SQL are the Parquet file, JSON document, HIVE tables, and Cassandra database

Schema RDD: Spark Core is designed with a special data structure called RDD. Generally, Spark SQL works on schemas, tables, and records. Therefore, we can use the Schema RDD as a temporary table.

SQL DataFrame is a distributed collection of data organized into named columns. It is equivalent to a relational table in SQL used for storing data in tables. We can create DataFrame using: Structured data files, Tables in Hive, databases, Using existing RDD

Databricks:

Databricks is the first unified analytics engine, that aims to help clients with cloud-based big data processing and machine learning. The team established Databricks who developed Apache Spark, the most active and powerful open-source information handling engine designed for advanced analytics, ease of use, and velocity. Databricks is the largest contributor to the open-source initiative of Apache Spark that delivers ten times as much software as any other company.

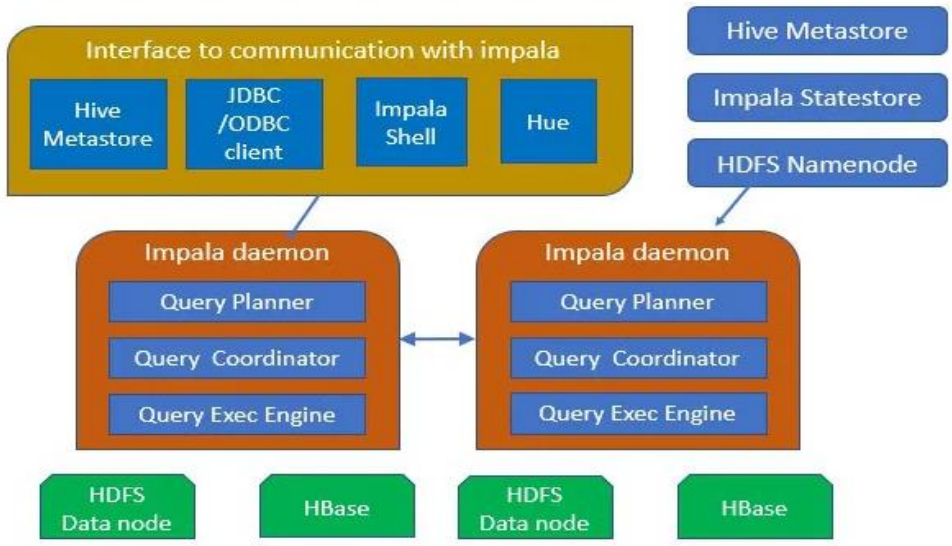


Apache Impala:

IMPALA is an open-source parallel processing query engine designed on top of clustered systems written for processing large volumes of data with SQL interactions. It has interactive SQL-like queries where we can fetch and work on data as needed.

Impala is an MPP (Massive Parallel Processing) query execution engine. It has three main components in its Architecture such as Impala daemon (ImpalaD), Impala statestore, and Impala metadata or metastore.

Architecture:



Hive Meta Store: let impala about the structure for these databases and the databases available.

Daemons: Impala Daemon, is one of the core components of the Hadoop Impala. It runs on every data node. Daemons read data from HDFS / HBASE and accept queries from Hue or any Data Analytics tools via the connection. Once that is done it distributes work across the cluster and parallelizes the query.

The job launching node is called the Coordinator Node. The daemons submit the results to these coordinator nodes only and further, the result is prepared. A query plan is made that parses the Query; a single plan is made first that is further distributed afterward. Then the Coordinator is responsible for executing the entire Query.

State Store Daemons: check the health of all Impala Daemons on all the data nodes in the Hadoop cluster and report the point of failure (if any). It is called a process statestored. The major advantage of this Daemon is it informs all the Impala Daemons if an Impala Daemon goes down. Hence, they can avoid the failed node while distributing future queries.

Catalog Daemons: distributes the metadata information to the impala daemons and checks communicates any changes over Metadata that come over from the queries to the Impala Daemons. So, there are some changes we need to refresh or invalidate the catalog daemons using the "INVALIDATE METADATA "command. It is suggested to have State Store and Catalog Daemons over the host because any change request will be known to both the Daemons.

HBase and HDFS: storage level for the data. Used for data handling after a query planning and optimization is run over impala, they come over here.

Impala Vs Hive:

Hive uses the concept of map-reduce for processing and sometimes takes time for the query to be processed. Impala offers high-performance, low–latency SQL queries. Which is faster compared to Hive. Impala has its own execution engine and stores the results in memory making it very fast for execution It integrates well with hive meta store to share databases and tables between hive and impala.

Impala Features:

- **Faster Access:** It is comparatively much faster for SQL Queries and data processing.
- **Storage System:** Has the capability to access HDFS and HBASE as its storage system.
- **In-Memory Processing:** the data are present in memory that making the query optimization faster and easy.
- **Secured:** It offers Kerberos Authentication making it secure.
- **Multi API Supports** Its supports multiple API that helps it the connection with the data sources.
- **Easily Connected:** It is easy to connect over many data visualization engines such as TABLEAU, etc.
- **Built-in Functions**: Impala comes over with several built-in Functions with which we can go over the results we need.

3. Hive and Spark Environment setup

Spark:

Mounting Dataset

```
ACCESSY_KEY_ID = "AKIAJBRYNXGHORDHZB4A"
SECERET_ACCESS_KEY = "a0BzE1bSegfydr3%2FGE3LSPM6uIV5A4hOUfpH8aFF"

mounts_list = [
{'bucket':'databricks-corp-training/sf_open_data/', 'mount_folder':'/mnt/sf_open_data'}
]
```

```
for mount_point in mounts_list:
  bucket = mount_point['bucket']
  mount_folder = mount_point['mount_folder']
  try:
    dbutils.fs.ls(mount_folder)
    dbutils.fs.unmount(mount_folder)
  except:
    pass
  finally: #If MOUNT_FOLDER does not exist
    dbutils.fs.mount("s3a://"+ ACCESSY_KEY_ID + ":" + SECERET_ACCESS_KEY + "@" + bucket,mount_folder)
```

Create DataFrame

```
fireServiceCallsDF = spark.read.csv('/mnt/sf_open_data/fire_dept_calls_for_service/Fire_Department_Calls_for_Service.csv', header=True,
schema=fireSchema)
```

```
from pyspark.sql.types import StructType, StructField, IntegerType, LongType, StringType, BooleanType

fireSchema = StructType([StructField('CallNumber', IntegerType(), True),
                     StructField('UnitID', StringType(), True),
                     StructField('IncidentNumber', IntegerType(), True),
                     StructField('CallType', StringType(), True),
                     StructField('CallDate', StringType(), True),
                     StructField('WatchDate', StringType(), True),
                     StructField('ReceivedDtTm', StringType(), True),
                     StructField('EntryDtTm', StringType(), True),
                     StructField('DispatchDtTm', StringType(), True),
                     StructField('ResponseDtTm', StringType(), True),
                     StructField('OnSceneDtTm', StringType(), True),
                     StructField('TransportDtTm', StringType(), True),
                     StructField('HospitalDtTm', StringType(), True),
                     StructField('CallFinalDisposition', StringType(), True),
                     StructField('AvailableDtTm', StringType(), True),
                     StructField('Address', StringType(), True),
                     StructField('City', StringType(), True),
                     StructField('ZipcodeofIncident', IntegerType(), True),
                     StructField('Battalion', StringType(), True),
                     StructField('StationArea', StringType(), True),
                     StructField('Box', StringType(), True),
                     StructField('OriginalPriority', StringType(), True),
                     StructField('Priority', StringType(), True),
                     StructField('FinalPriority', IntegerType(), True),
                     StructField('ALSUnit', BooleanType(), True),
```

Visualization of DataSet

```
1  display(fireServiceCallsDF)
```

Python

▸ (1) Spark Jobs

	CallNumber	UnitID	IncidentNumber	CallType	CallDate	WatchDate	ReceivedDtTm	EntryDtTm	Dis
1	142480332	B02	14086309	Alarms	09/05/2014	09/04/2014	09/05/2014 03:15:13 AM	09/05/2014 03:17:26 AM	0
2	153022542	T02	15115908	Structure Fire	10/29/2015	10/29/2015	10/29/2015 03:39:06 PM	10/29/2015 03:39:25 PM	1
3	143451112	AM04	14122741	Medical Incident	12/11/2014	12/11/2014	12/11/2014 09:02:07 AM	12/11/2014 09:03:01 AM	1
4	141660300	E01	14057129	Medical Incident	06/15/2014	06/14/2014	06/15/2014 02:04:57 AM	06/15/2014 02:06:42 AM	0
5	152633454	E36	15100829	Outside Fire	09/20/2015	09/20/2015	09/20/2015 08:15:00 PM	09/20/2015 08:15:53 PM	0
6	160941229	62	16037213	Medical Incident	04/03/2016	04/03/2016	04/03/2016 10:11:05 AM	04/03/2016 10:13:32 AM	0
7									

Truncated results, showing first 1000 rows

Hive:

Get DataSet:

Fire Incidents data collected by the Fire Department, San Francisco

https://data.sfgov.org/Public-Safety/Fire-Department-Calls-for-Service/nuek-vuh3

Locate data Hadoop:

SCP the file over to the home directory (/home/Hadoop) on Hadoop

Copy Data from Local to HDFS

hadoop fs -copyFromLocal fireServices.csv /user/hadoop

Create HIVE Table

```
hive> DROP TABLE IF EXISTS fireDepartment;
OK
Time taken: 0.025 seconds
hive> CREATE EXTERNAL TABLE IF NOT EXISTS fireDepartment (
    > CallNumber integer,
    > UnitID string,
    > IncidentNumber integer,
    > CallType string,
    > CallDate string,
    > WatchDate string,
    > ReceivedDtTm string,
    > EntryDtTm string,
    > DispatchDtTm string,
    > ResponseDtTm string,
    > OnSceneDtTm string,
    > TransportDtTm string,
    > HospitalDtTm string,
    > CallFinalDisposition string,
    > AvailableDtTm string,
    > Address string,
    > City string,
    > ZipcodeofIncident integer,
    > Battalion string,
    > StationArea string,
    > Box string,
    > OriginalPriority string,
    > Priority string,
    > FinalPriority integer,
    > ALSUnit boolean,
    > CallTypeGroup string,
    > NumberofAlarms integer,
    > UnitType string,
    > Unitsequenceincalldispatch integer,
    > FirePreventionDistrict string,
    > SupervisorDistrict string,
    > NeighborhoodDistrict string,
    > Location string,
    > RowID string,
    > Call_Date_TS timestamp,
```

Load data from HDFS to Hive Table

load data in path '/user/hadoop/fireServices.csv' into table fire department.

4. Query Execution and Results

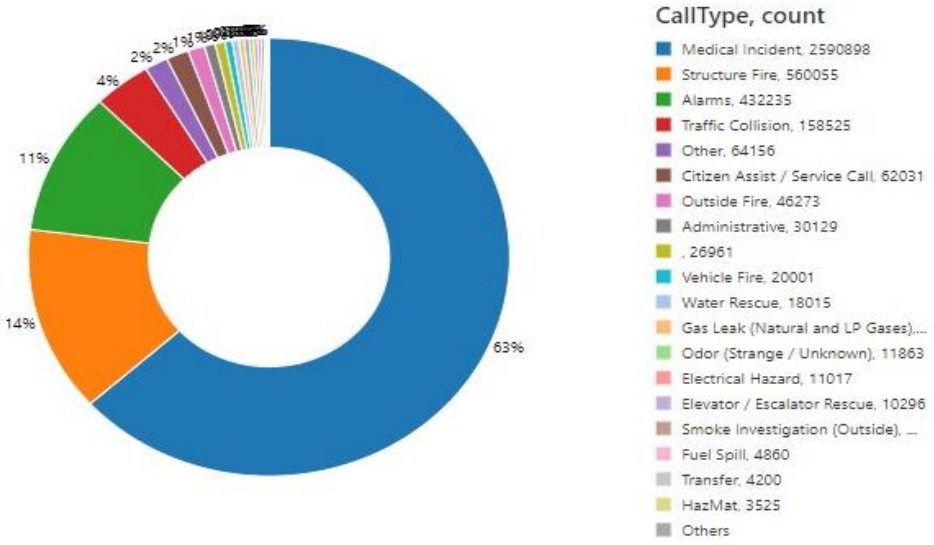
Query 1:

How many incidents of each call type were there?

Spark:

display (fireServiceCallsDF.select('CallType').GroupBy ('CallType'). count (). OrderBy ("count", ascending=False))

Result:



CallType	count
Medical Incident	2590898
Structure Fire	560055
Alarms	432235
Traffic Collision	158525
Other	64156
Citizen Assist / Service Call	62031
Outside Fire	46273
Administrative	30129
null	26961
Vehicle Fire	20001
Water Rescue	18015
Gas Leak (Natural and LP Gases)	14351
Odor (Strange / Unknown)	11863
Electrical Hazard	11017
Elevator / Escalator Rescue	10296
Smoke Investigation (Outside)	8776
Fuel Spill	4860
Transfer	4200

Hive:

select CallType, count (*) as number_of_calls from fireDepartment group by CallType order by number_of_calls desc;

Result:

```
hive> select CallType,count(*) as number_of_calls from fireDepartment group by CallType order by number_of_calls desc;
Query ID = hadoop_20220504153153_a9c09881-a756-4c5c-bc03-dd6ed4cb4308
Total jobs = 1
Launching Job 1 out of 1
Tez session was closed. Reopening...
Session re-established.
Status: Running (Executing on YARN cluster with App id application_1651670031486_0004)

--------------------------------------------------------------------------------
        VERTICES      MODE        STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED  KILLED
--------------------------------------------------------------------------------
Map 1 .......... container     SUCCEEDED      2          2        0        0       0       0
Reducer 2 ...... container     SUCCEEDED      2          2        0        0       0       0
Reducer 3 ...... container     SUCCEEDED      1          1        0        0       0       0
--------------------------------------------------------------------------------
VERTICES: 03/03  [==========================>>] 100%  ELAPSED TIME: 5.65 s
--------------------------------------------------------------------------------
OK
Medical Incident        2256
Alarms  237
Structure Fire  228
Traffic Collision       144
Citizen Assist / Service Call   27
Other   18
Outside Fire    18
Water Rescue    15
Elevator / Escalator Rescue     12
Vehicle Fire    12
Gas Leak (Natural and LP Gases) 9
Odor (Strange / Unknown)        6
Fuel Spill      6
Aircraft Emergency      3
Electrical Hazard       3
Transfer        3
null    3
Time taken: 11.362 seconds, Fetched: 17 row(s)
```

Query 2:

What was the average response time to fires?

Spark:

```
from pyspark.sql.functions import *
spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY")

from_pattern1 = "MM/dd/yyyy"
from_pattern2 = "MM/dd/yyyy hh:mm:ss"

fireServiceCallsTSDF =  \
fireServiceCallsDF.withColumn("Call_Date_TS",  to_timestamp("CallDate", from_pattern1)) \
                  .withColumn("Watch_Date_TS", unix_timestamp(col("WatchDate"), from_pattern1).cast("timestamp")) \
                  .withColumn("Received_DtTm_TS", to_timestamp("ReceivedDtTm", from_pattern2)) \
                  .withColumn("Entry_DtTm_TS", to_timestamp("EntryDtTm", from_pattern2)) \
                  .withColumn("Dispatch_DtTm_TS", to_timestamp("DispatchDtTm", from_pattern2)) \
                  .withColumn("Response_DtTm_TS", to_timestamp("ResponseDtTm", from_pattern2)) \
                  .withColumn("On_Scene_DtTm_TS", unix_timestamp(col("OnSceneDtTm"), from_pattern2).cast("timestamp")) \
                  .withColumn("Transport_DtTm_TS", unix_timestamp(col("TransportDtTm"), from_pattern2).cast("timestamp")) \
                  .withColumn("Hospital_DtTm_TS", unix_timestamp(col("HospitalDtTm"), from_pattern2).cast("timestamp")) \
                  .withColumn("Available_DtTm_TS", unix_timestamp(col("AvailableDtTm"), from_pattern2).cast("timestamp"))

fireServiceCallsTSDF.printSchema()
```

```
root
 |-- CallNumber: integer (nullable = true)
 |-- UnitID: string (nullable = true)
 |-- IncidentNumber: integer (nullable = true)
 |-- CallType: string (nullable = true)
 |-- CallDate: string (nullable = true)
 |-- WatchDate: string (nullable = true)
 |-- ReceivedDtTm: string (nullable = true)
 |-- EntryDtTm: string (nullable = true)
 |-- DispatchDtTm: string (nullable = true)
 |-- ResponseDtTm: string (nullable = true)
 |-- OnSceneDtTm: string (nullable = true)
 |-- TransportDtTm: string (nullable = true)
 |-- HospitalDtTm: string (nullable = true)
 |-- CallFinalDisposition: string (nullable = true)
 |-- AvailableDtTm: string (nullable = true)
 |-- Address: string (nullable = true)
 |-- City: string (nullable = true)
 |-- ZipcodeofIncident: integer (nullable = true)
 |-- Battalion: string (nullable = true)
 |-- StationArea: string (nullable = true)
```

```
fireServices_TSOnly = fireServices_TSOnly.withColumn("Response",col('Response_DtTm_TS').cast('long') - col('Received_DtTm_TS').cast('long'))

display(fireServices_TSOnly.agg({'minutes':'avg'}))
```

Result: -

	avg(minutes)
1	27.235675718747455

Hive:

select avg (unix_timestamp(t.resp)-unix_timestamp(t.rece)) from (select from_unixtime(to_unix_timestamp(ResponseDtTm, 'MM/dd/yyyy HH:mm')) as resp, from_unixtime(to_unix_timestamp(ReceivedDtTm, 'MM/dd/yyyy HH:mm')) as rece from fireDepartment) as t;

Result: -

```
hive> select avg(unix_timestamp(t.resp)-unix_timestamp(t.rece)) from (select from_unixtime(to_unix_timestamp(ResponseDtTm,'MM/dd/yyyy HH:mm')) as resp, from_unixtime(to
_unix_timestamp(ReceivedDtTm,'MM/dd/yyyy HH:mm')) as rece from fireDepartment) as t;
Query ID = hadoop_20220504174733_fd6ba7df-cca6-4f27-b5d2-86f88648aa5a
Total jobs = 1
Launching Job 1 out of 1
Tez session was closed. Reopening...
Session re-established.
Status: Running (Executing on YARN cluster with App id application_1651670031486_0010)

--------------------------------------------------------------------------------
        VERTICES      MODE        STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED  KILLED
--------------------------------------------------------------------------------
Map 1 .......... container     SUCCEEDED      2          2        0        0       0       0
Reducer 2 ...... container     SUCCEEDED      1          1        0        0       0       0
--------------------------------------------------------------------------------
VERTICES: 02/02  [==========================>>] 100%  ELAPSED TIME: 5.92 s
--------------------------------------------------------------------------------
OK
188.46
Time taken: 11.442 seconds, Fetched: 1 row(s)
```

Query 3:

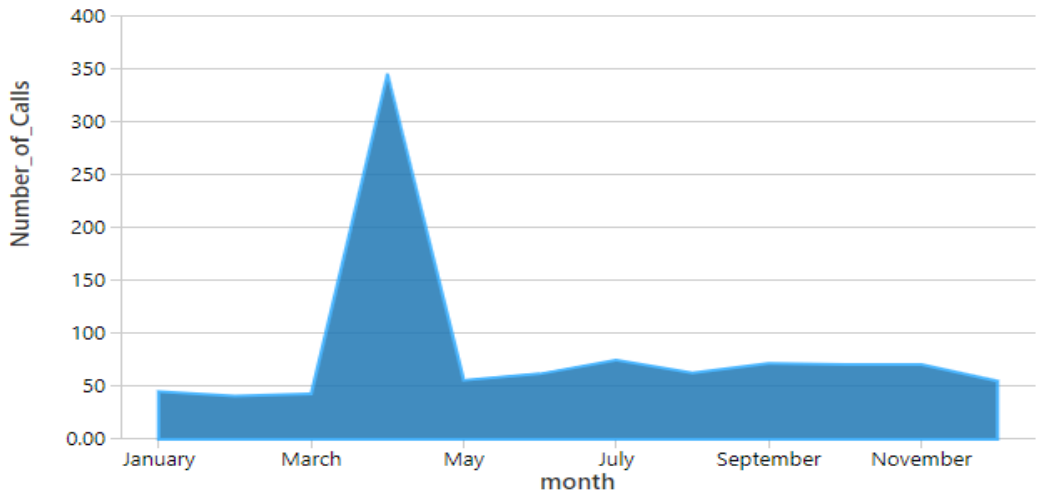
How many fires took place each month of the year?

Spark:

```
display(spark.sql("""select t.month,t.Number_of_Calls from (select distinct date_format(Call_Date_TS,'MMMMM') as month,month(Call_Date_TS) as mn,
count(*) as Number_of_Calls from fireServices_TSOnly_test group by month,mn order by mn asc) t """))
```

Result:

	month	Number_of_Calls
1	January	45
2	February	41
3	March	43
4	April	346
5	May	56
6	June	62
7	July	75
8	August	63
9	September	72
10	October	71
11	November	71
12	December	55



Hive:

select t.month, count(*) as Number_of_Calls from (select date_format(from_unixtime(to_unix_timestamp (ResponseDtTm,'MM/dd/yyyy HH:mm')),'MMMMM') as month,month(from_unixtime(to_unix_timestamp(ResponseDtTm,'MM/dd/yyyy HH:mm'))) as mn from fireDepartment) t group by t. month;

Result:

```
hive> select t.month, count(*) as Number_of_Calls from(select date_format(from_unixtime(to_unix_timestamp(ResponseDtTm,'MM/dd/yyyy HH:mm')),'MMMMM') as month,month(from
_unixtime(to_unix_timestamp(ResponseDtTm,'MM/dd/yyyy HH:mm'))) as mn from fireDepartment) t group by t.month;
Query ID = hadoop_20220504185254_4a8cc22d-4fa8-40c2-bc3d-4746f73da7b7
Total jobs = 1
Launching Job 1 out of 1
Status: Running (Executing on YARN cluster with App id application_1651670031486_0012)

--------------------------------------------------------------------------------
        VERTICES      MODE        STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED  KILLED
--------------------------------------------------------------------------------
Map 1 .......... container     SUCCEEDED      2          2        0        0       0       0
Reducer 2 ...... container     SUCCEEDED      2          2        0        0       0       0
--------------------------------------------------------------------------------
VERTICES: 02/02  [==========================>>] 100%  ELAPSED TIME: 6.29 s
--------------------------------------------------------------------------------
OK
April   3806
August  693
February        451
July    825
June    682
September       792
December        605
January 495
March   473
May     616
November        781
October 781
Time taken: 6.698 seconds, Fetched: 12 row(s)
hive>
```

Query 4:

What was the average time spent at the fire scene?

Spark:

```
fireServices_TSOnly_test = fireServices_TSOnly_test.withColumn("time_spent",col('Transport_DtTm_TS').cast('long') -
col('On_Scene_DtTm_TS').cast('long'))
avg_time_spent = fireServices_TSOnly_test.select(mean('time_spent')*60 % 60*3600 % 60)
newcol = ['timespent']
avg_time_spent = avg_time_spent.toDF(*newcol)
display(avg_time_spent)
```

Result:

	timespent
1	35.999794006347656

Hive:

select avg(unix_timestamp(t. resp)-unix_timestamp(t.rece))*60 % 60*3600 % 60 from (select from_unixtime(to_unix_timestamp(TransportDtTm, 'MM/dd/yyyy HH:mm')) as resp, from_unixtime(to_unix_timestamp(OnSceneDtTm,'MM/dd/yyyy HH:mm')) as rece from fireDepartment) as t;

Result: -

```
hive> select avg(unix_timestamp(t.resp)-unix_timestamp(t.rece))*60 % 60*3600 % 60 from (select from_unixtime(to_unix_timestamp(TransportDtTm,'MM/dd/yyyy HH:mm')) as res
p, from_unixtime(to_unix_timestamp(OnSceneDtTm,'MM/dd/yyyy HH:mm')) as rece from fireDepartment) as t;
Query ID = hadoop_20220504190439_36a2be97-3557-4c93-b137-a1845f9c1dce
Total jobs = 1
Launching Job 1 out of 1
Tez session was closed. Reopening...
Session re-established.
Status: Running (Executing on YARN cluster with App id application_1651670031486_0013)

--------------------------------------------------------------------------------
        VERTICES      MODE        STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED  KILLED
--------------------------------------------------------------------------------
Map 1 .......... container     SUCCEEDED      2          2        0        0       0       0
Reducer 2 ...... container     SUCCEEDED      1          1        0        0       0       0
--------------------------------------------------------------------------------
VERTICES: 02/02  [==========================>>] 100%  ELAPSED TIME: 7.76 s
--------------------------------------------------------------------------------
OK
59.99991416931152
Time taken: 12.694 seconds, Fetched: 1 row(s)
hive>
```

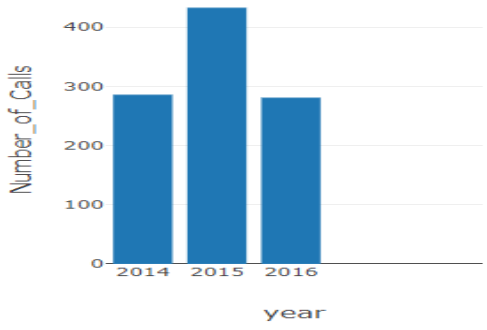
Query 5:

How many calls are made to the Fire Service per year?

Spark:

```
display(spark.sql("""select distinct year(Call_Date_TS) as year, count(*) as Number_of_Calls from fireServices_TSOnly_test group by year(Call_Date_TS)"""))
```

Result: -



	year	Number_of_Calls
1	2014	286
2	2015	433
3	2016	281

Hive:

select t.year,count(*) as Number_of_Calls from (select year(from_unixtime(to_unix_timestamp(ResponseDtTm,'MM/dd/yyyy HH:mm'))) as year from fireDepartment) t group by t.year order by Number_of_Calls desc;

Result: -

```
hive> select t.year,count(*) as Number_of_Calls from (select year(from_unixtime(to_unix_timestamp(ResponseDtTm,'MM/dd/yyyy HH:mm'))) as year from fireDepartment) t grou
p by t.year order by Number_of_Calls desc;
Query ID = hadoop_20220504183831_38d0d5dc-e3e3-45ed-a330-9e4bc474b7f5
Total jobs = 1
Launching Job 1 out of 1
Tez session was closed. Reopening...
Session re-established.
Status: Running (Executing on YARN cluster with App id application_1651670031486_0011)

--------------------------------------------------------------------------------
        VERTICES      MODE        STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED  KILLED
--------------------------------------------------------------------------------
Map 1 .......... container     SUCCEEDED      2          2        0        0       0       0
Reducer 2 ...... container     SUCCEEDED      2          2        0        0       0       0
Reducer 3 ...... container     SUCCEEDED      1          1        0        0       0       0
--------------------------------------------------------------------------------
VERTICES: 03/03  [==========================>>] 100%  ELAPSED TIME: 7.11 s
--------------------------------------------------------------------------------
OK
2015    4763
2014    3146
2016    3091
Time taken: 11.827 seconds, Fetched: 3 row(s)
```

Query 6:

How many calls were made and how much did response times vary by fire station area?

Spark:

```
display(spark.sql("""select UnitID,count(*) as Number_of_Calls,avg(Response) avg_Response from fireServices_TSOnly_test group by UnitID order by
Number_of_Calls desc"""))
```

Result:

	UnitID	Number_of_Calls	avg_Response
1	E03	30	232.4
2	E01	25	220.84
3	88	24	122.83333333333333
4	E36	23	242.91304347826087
5	52	23	150.43478260869566
6	82	17	180.05882352941177
7	E13	17	175.7058823529412
8	77	16	175.875
9	60	15	151.53333333333333
10	74	15	150.73333333333332
11	55	14	190.42857142857142
12	E08	14	187.21428571428572
13	70	13	141.15384615384616
14	AM02	13	167.84615384615384
15	85	13	188.15384615384616
16	89	13	216.30769230769232

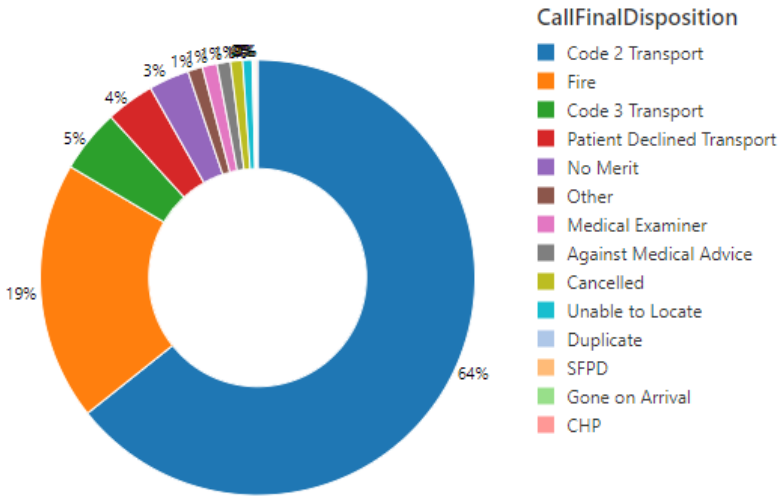
Query 7:

How many calls were made based on Call Final Disposition?

Spark:

```
display(spark.sql("""select CallFinalDisposition,count(*) as Number_of_Calls from fireServices_TSOnly_test group by CallFinalDisposition order by Number_of_Calls desc"""))
```

Result:



	CallFinalDisposition	Number_of_Calls
1	Code 2 Transport	643
2	Fire	192
3	Code 3 Transport	47
4	Patient Declined Transport	36
5	No Merit	30
6	Other	11
7	Medical Examiner	11
8	Against Medical Advice	10
9	Cancelled	9
10	Unable to Locate	7
11	Duplicate	1
12	SFPD	1
13	Gone on Arrival	1
14	CHP	1

4. Conclusion

From the results, most of the queries ran faster in Spark than in Hive.

Spark uses MapReduce more efficiently than Hive, which lends itself to faster processing. Hive is best suited for applications that perform operations on RDBMS databases that need a scale-out database. On the other hand, Spark is best suited for applications performing big data analytics which require faster performance results on the dataset given.

Therefore, it is advisable to use Spark if we need to perform real-time data analytics, but if we aim for querying on large volumes of datasets that are historical, then Hive would be a better option.

5. References:

1. Kukreja, Manish. (2016). Apache Hive: Enterprise SQL on Big Data frameworks. 10.13140/RG.2.1.3387.8008
2. A Thusoo et al. “Hive - a petabyte scale data warehouse using Hadoop”. In: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010). Mar. 2010, pp. 996–1005.
3. Salloum, S., Dautov, R., Chen, X. et al. Big data analytics on Apache Spark. Int J Data Sci Anal 1, 145–164 (2016).
4. J. Flood, " The Fires: How a Computer Formula, Big Ideas, and the Best of Intentions Burned Down New York City--and Determined the Future of Cities” [Paperback]. Available: Riverhead Books (2011)
5. https://doi.org/10.1007/s41060-016-0027-9

6. https://www.kaggle.com/datasets/san-francisco/sf-fire-data-incidents-violations-and-more
7. https://data.sfgov.org/Public-Safety/Fire-Incidents/wr8u-xric
8. https://www.usfa.fema.gov/data/statistics/order_download_data.html#tools
9. https://hadoop.apache.org/
10. https://www.edureka.co/blog/spark-sql-tutorial/
11. https://spark.apache.org/docs/latest/sql-programming-guide.html
12. https://aws.amazon.com/big-data/what-is-hive/
13. https://hive.apache.org/
14. https://www.toptal.com/spark/introduction-to-apache-spark
15. https://spark.apache.org/docs/latest/api/python/getting_started/index.html
16. https://sparkbyexamples.com/pyspark-tutorial/
17. https://datafloq.com/read/why-you-must-choose-databricks-data-science-big-data-workloads/
18. https://databricks.com/learn
19. Course work Assignments and Lecture Slide
20. https://www.educba.com/what-is-impala/
21. https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/impala_components.html