Regression Type	Formula	Description
Linear Regression	$y = \beta_0 + \beta_1 x_1 + \epsilon$	where: y is the dependent variable. x is the independent variable. β_0 is the y-intercept. β_1 is the slope of the line. ϵ is the error term (residual)
Multiple Linear Regression	$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + .. \beta_n x_n + \epsilon$	where: y is the dependent variable. $x_1, x_2, x_3, ... x_n$ are the independent variables. $\beta_1, \beta_2, \beta_3, ... \beta_n$ are the coefficients. ϵ is the error term (residual)
Logistic Regression	$P(y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ...\beta_n x_n)}}$	where: y is the dependent variable. $x1, x2, x3, ... xn$ are the independent variables. $\beta 1, \beta 2, \beta 3, ... \beta n$ are the coefficients. e Is Euler's Coeffiecient,

Aspect	Description
Algorithm Name	DGIM (Datar-Gionis-Indyk-Motwani) Algorithm
Purpose	Approximate the number of 1s in the last N bits of a binary stream
Application	Processing large data streams with limited memory
Key Idea	Maintain a summary of the stream using buckets, each representing a group of 1s
Bucket Properties	• Stores timestamp of most recent 1 • Sizes are powers of 2 (1, 2, 4, 8, etc.) • At most two buckets of each size
Process	1. Create a bucket of size 1 for new 1s 2. Merge oldest two buckets if more than two of a size exist 3. Remove buckets outside the window
Querying	1. Sum sizes of buckets fully within the window 2. Add fraction of oldest bucket partially in window
Accuracy	Guarantees approximation within 50% of true count
Space Complexity	O(log N) buckets, where N is the window size

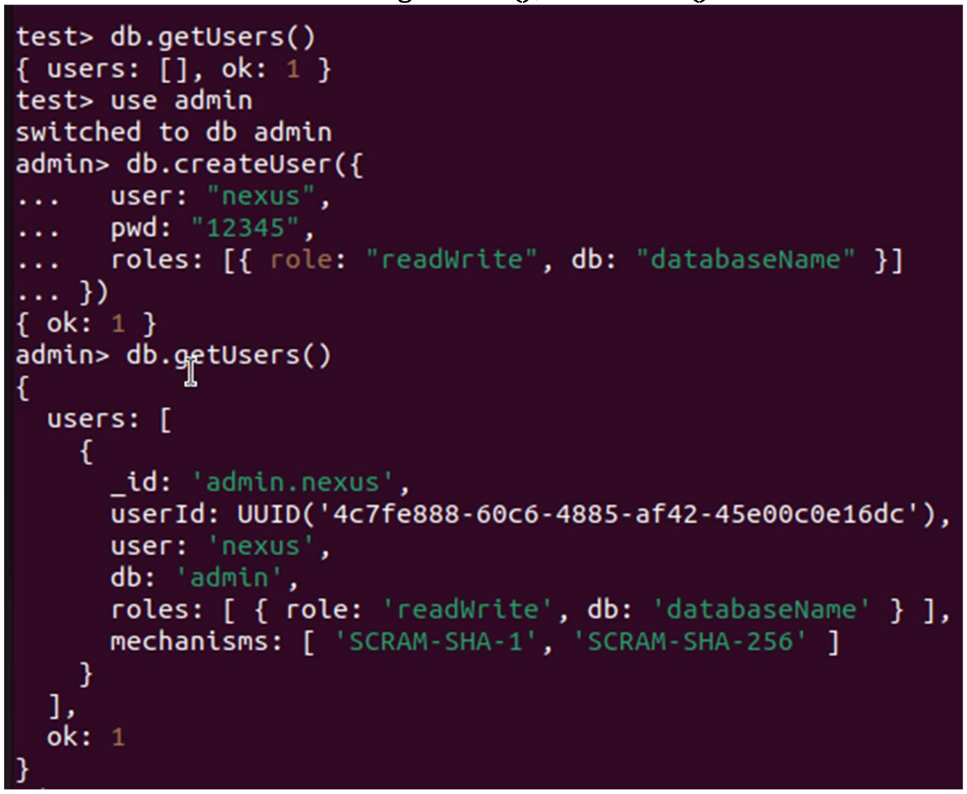
MongoDB Commands:

Command	Description	Syntax Example
Database Operations		
show dbs	Lists all databases.	show dbs
use <database>	Switches to the specified database. If it doesn't exist, it will be created upon inserting data.	use myDatabase
db	Displays the current database.	db
db.dropDatabase()	Deletes the current database.	db.dropDatabase()
Collection Operations		
show collections	Lists all collections in the current database.	show collections
db.createCollection()	Creates a new collection.	db.createCollection("myCollection")
db.<collection>.drop()	Drops the specified collection.	db.myCollection.drop()
Insert Operations		
db.<collection>.insertOne()	Inserts a single document into the collection.	db.myCollection.insertOne({ name: "John", age: 30 })
db.<collection>.insertMany()	Inserts multiple documents into the collection.	db.myCollection.insertMany([{ name: "Jane" }, { name: "Doe" }])
Update Operations		
db.<collection>.updateOne()	Updates a single document that matches the filter criteria.	db.myCollection.updateOne({ name: "John" }, { $set: { age: 31 } })
db.<collection>.updateMany()	Updates all documents that match the filter criteria.	db.myCollection.updateMany({ name: "John" }, { $set: { age: 32 } })
db.<collection>.replaceOne()	Replaces a single document that matches the filter criteria.	db.myCollection.replaceOne({ name: "John" }, { name: "John Doe", age: 33 })
Delete Operations		
db.<collection>.deleteOne()	Deletes a single document that matches the filter criteria.	db.myCollection.deleteOne({ name: "John" })
db.<collection>.deleteMany()	Deletes all documents that match the filter criteria.	db.myCollection.deleteMany({ name: "John" })
Query Operations		
db.<collection>.find()	Finds all documents in a collection that match the query criteria.	db.myCollection.find({ age: { $gt: 25 } })
db.<collection>.findOne()	Finds the first document in a collection that matches the query criteria.	db.myCollection.findOne({ name: "Jane" })
Indexing		
db.<collection>.createIndex()	Creates an index on a field or fields.	db.myCollection.createIndex({ age: 1 })
db.<collection>.getIndexes()	Returns a list of all indexes for a collection.	db.myCollection.getIndexes()
db.<collection>.dropIndex()	Drops a specified index from a collection.	db.myCollection.dropIndex("age_1")

Aggregation Operations		
db.<collection>.aggregate()	Performs aggregation operations like grouping, sorting, and averaging.	db.myCollection.aggregate([{ $match: { age: { $gt: 25 } } }, { $group: { _id: "$age", total: { $sum: 1 } } }])
Miscellaneous Commands		
db.stats()	Provides statistics about the current database.	db.stats()
db.<collection>.stats()	Provides statistics about a specific collection.	db.myCollection.stats()

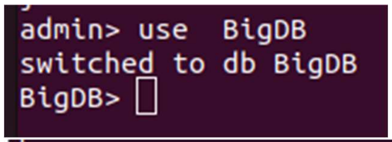
Part B CRUD Commands Implementation

- User based commands: getUsers(), createUser()

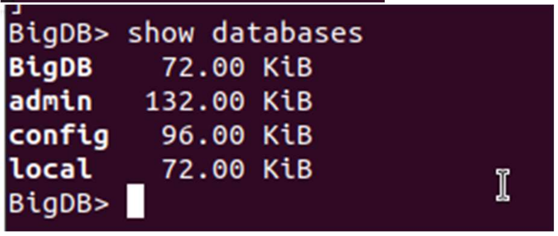


```
test> db.getUsers()
{ users: [], ok: 1 }
test> use admin
switched to db admin
admin> db.createUser({
...    user: "nexus",
...    pwd: "12345",
...    roles: [{ role: "readWrite", db: "databaseName" }]
... })
{ ok: 1 }
admin> db.getUsers()
{
  users: [
    {
      _id: 'admin.nexus',
      userId: UUID('4c7fe888-60c6-4885-af42-45e00c0e16dc'),
      user: 'nexus',
      db: 'admin',
      roles: [ { role: 'readWrite', db: 'databaseName' } ],
      mechanisms: [ 'SCRAM-SHA-1', 'SCRAM-SHA-256' ]
    }
  ],
  ok: 1
}
```

- Database Commands: Create database, show databases

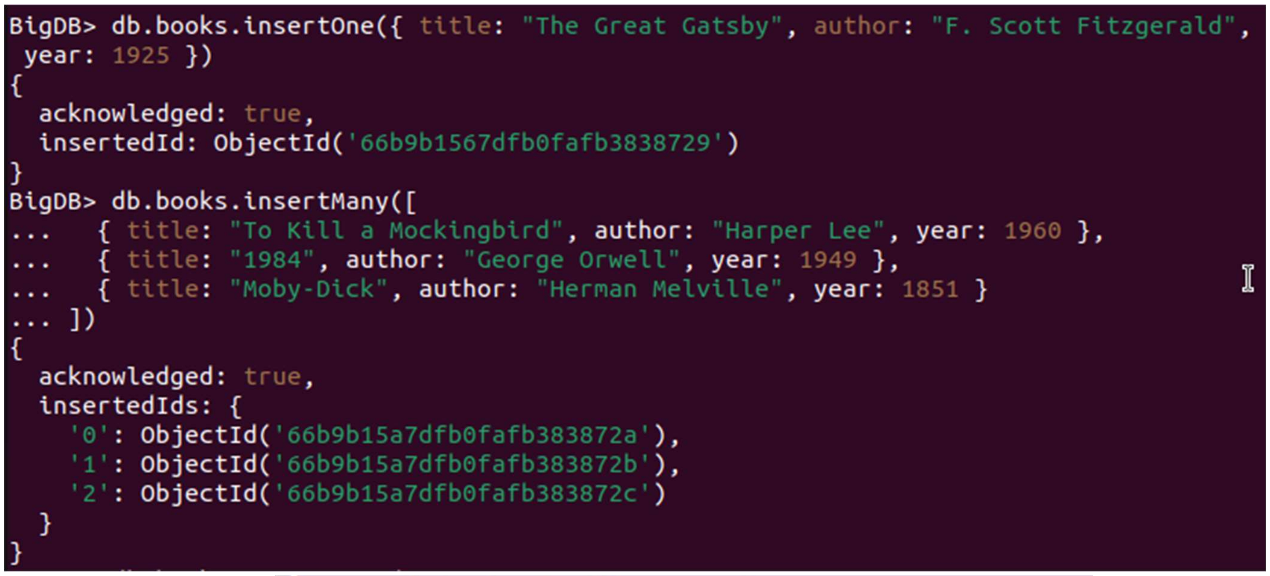


```
admin> use  BigDB
switched to db BigDB
BigDB>
```

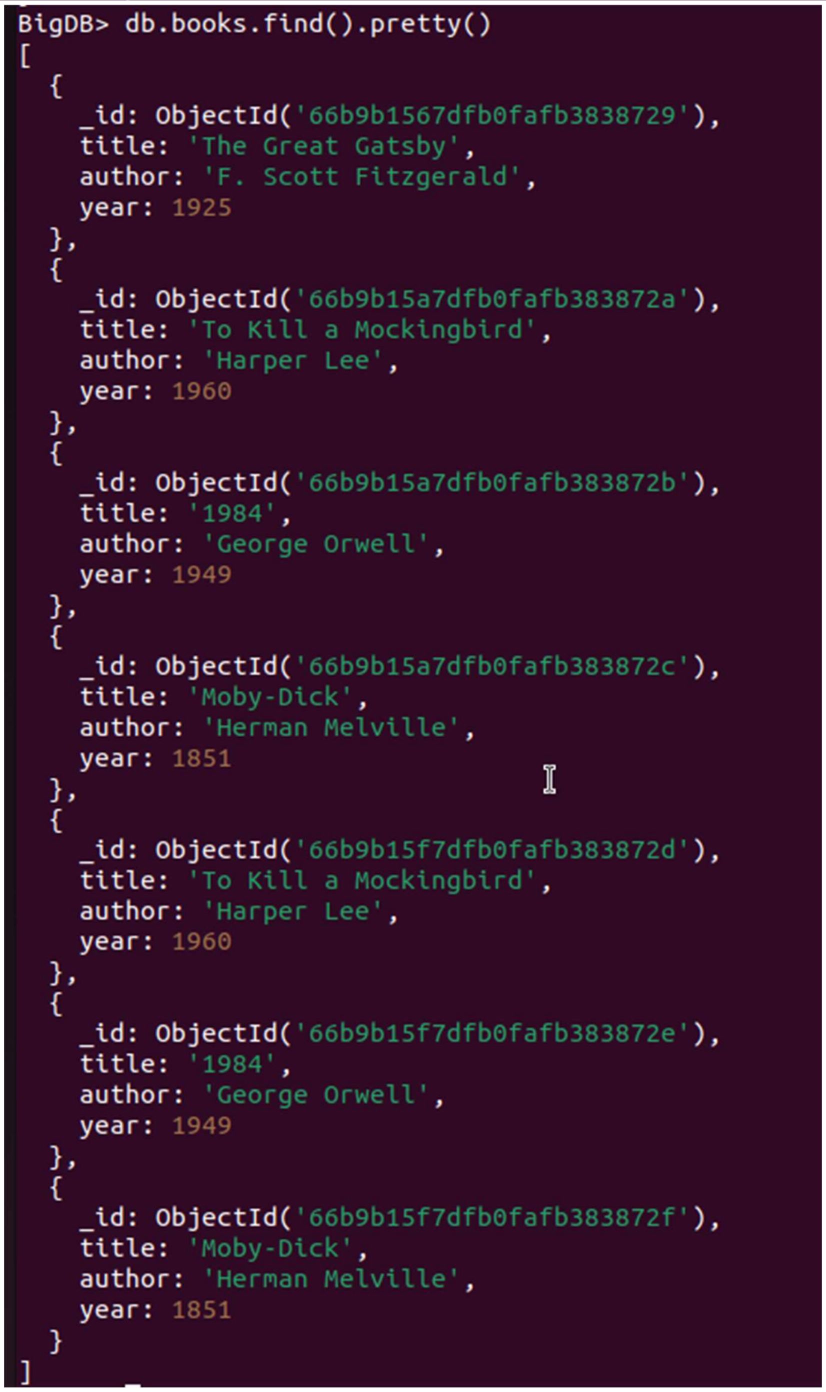


```
BigDB> show databases
BigDB     72.00 KiB
admin    132.00 KiB
config    96.00 KiB
local     72.00 KiB
BigDB>
```

- Insert Commands: insertOne(), Insert Many



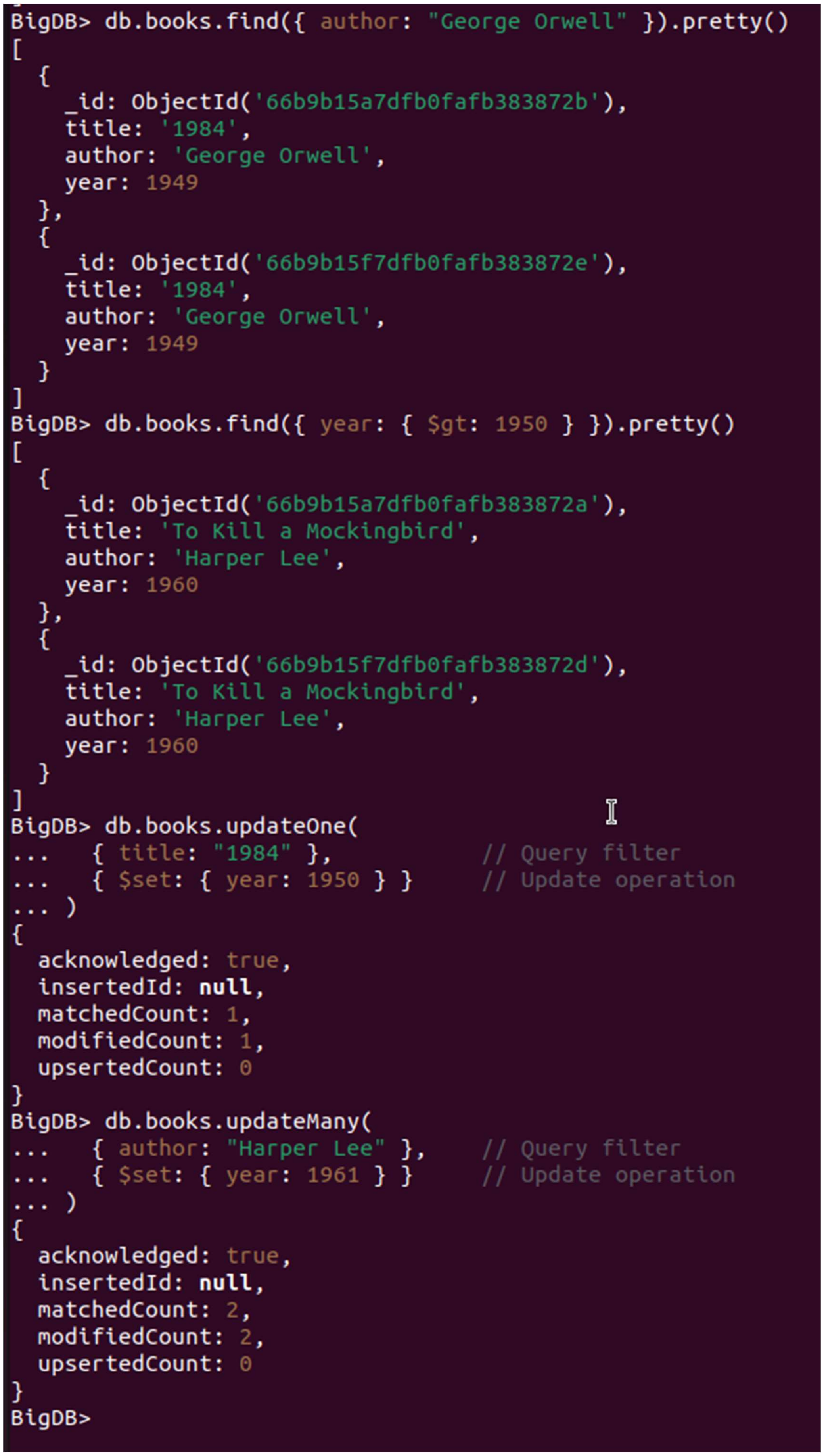
```
BigDB> db.books.insertOne({ title: "The Great Gatsby", author: "F. Scott Fitzgerald",
 year: 1925 })
{
  acknowledged: true,
  insertedId: ObjectId('66b9b1567dfb0fafb3838729')
}
BigDB> db.books.insertMany([
...   { title: "To Kill a Mockingbird", author: "Harper Lee", year: 1960 },
...   { title: "1984", author: "George Orwell", year: 1949 },
...   { title: "Moby-Dick", author: "Herman Melville", year: 1851 }
... ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('66b9b15a7dfb0fafb383872a'),
    '1': ObjectId('66b9b15a7dfb0fafb383872b'),
    '2': ObjectId('66b9b15a7dfb0fafb383872c')
  }
}
```



```
BigDB> db.books.find().pretty()
[
  {
    _id: ObjectId('66b9b1567dfb0fafb3838729'),
    title: 'The Great Gatsby',
    author: 'F. Scott Fitzgerald',
    year: 1925
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872a'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1960
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872b'),
    title: '1984',
    author: 'George Orwell',
    year: 1949
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872c'),
    title: 'Moby-Dick',
    author: 'Herman Melville',
    year: 1851
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872d'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1960
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872e'),
    title: '1984',
    author: 'George Orwell',
    year: 1949
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872f'),
    title: 'Moby-Dick',
    author: 'Herman Melville',
    year: 1851
  }
]
```

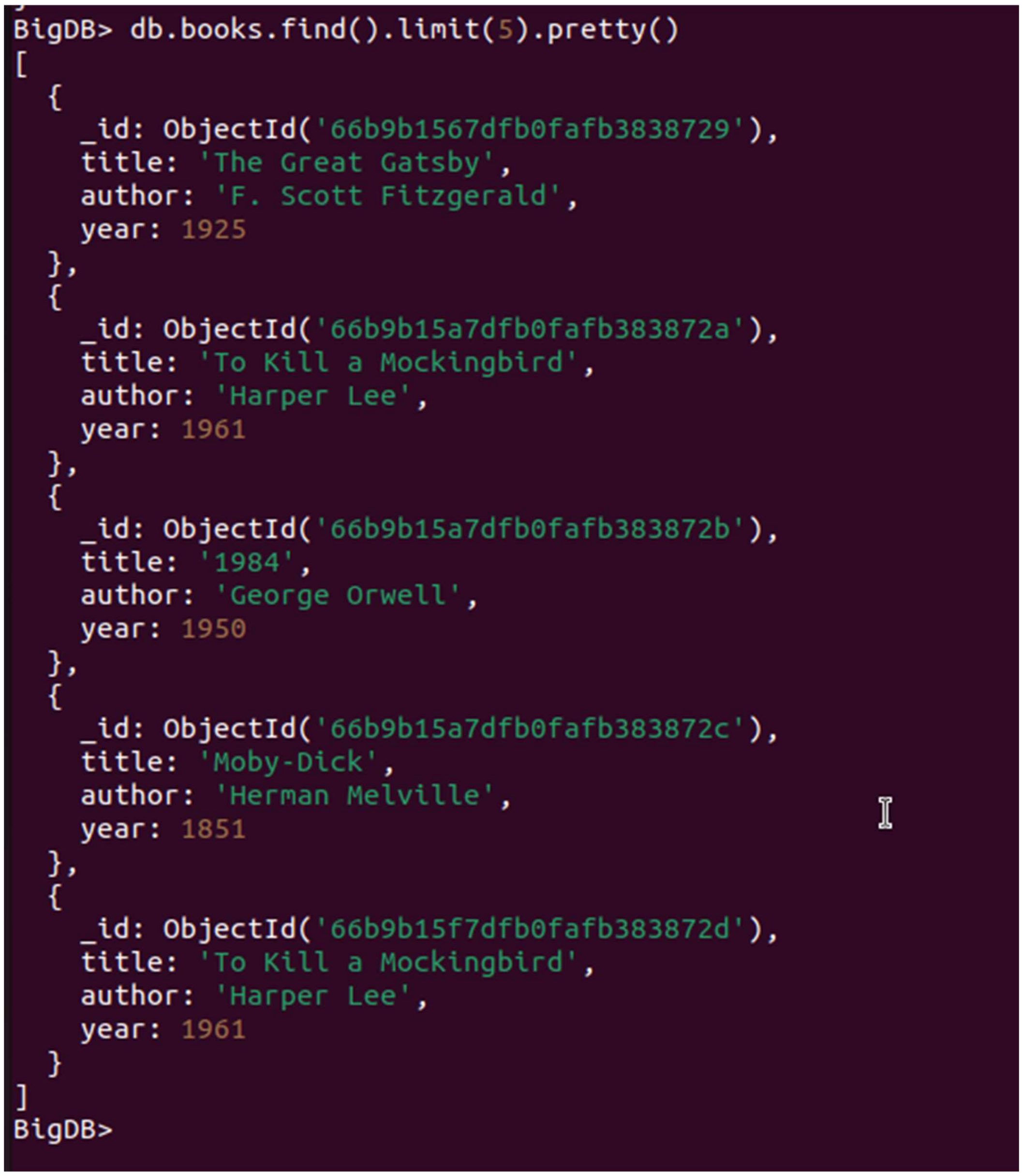
- Find

- Find with Condition:

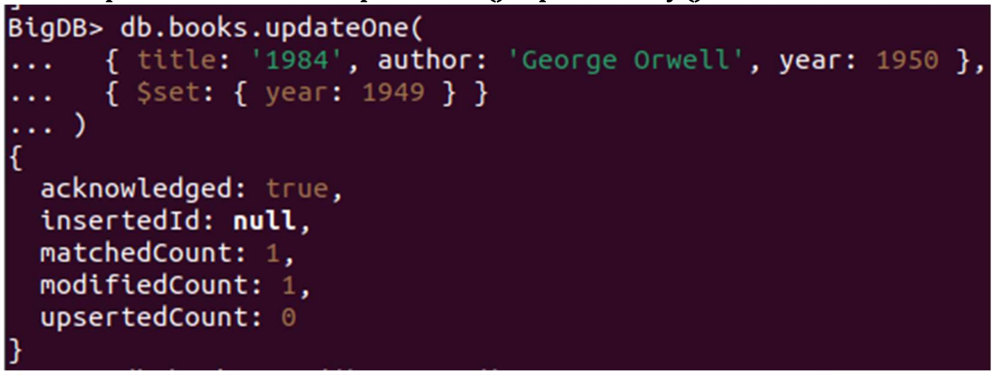


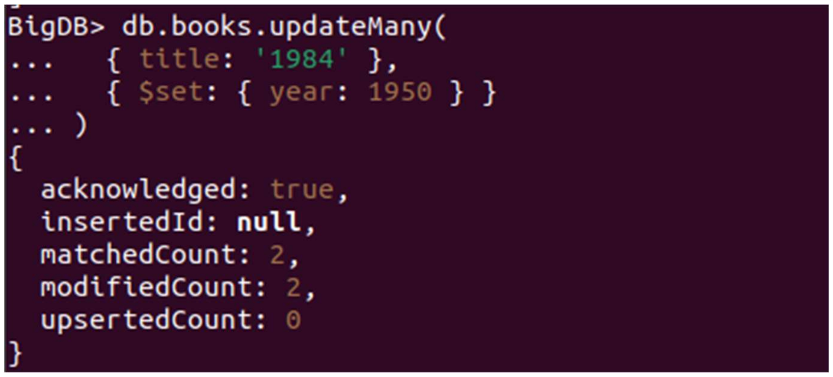
```
BigDB> db.books.find({ author: "George Orwell" }).pretty()
[
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872b'),
    title: '1984',
    author: 'George Orwell',
    year: 1949
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872e'),
    title: '1984',
    author: 'George Orwell',
    year: 1949
  }
]
BigDB> db.books.find({ year: { $gt: 1950 } }).pretty()
[
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872a'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1960
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872d'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1960
  }
]
BigDB> db.books.updateOne(
...    { title: "1984" },             // Query filter
...    { $set: { year: 1950 } }       // Update operation
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
BigDB> db.books.updateMany(
...    { author: "Harper Lee" },      // Query filter
...    { $set: { year: 1961 } }       // Update operation
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 2,
  upsertedCount: 0
}
BigDB>
```

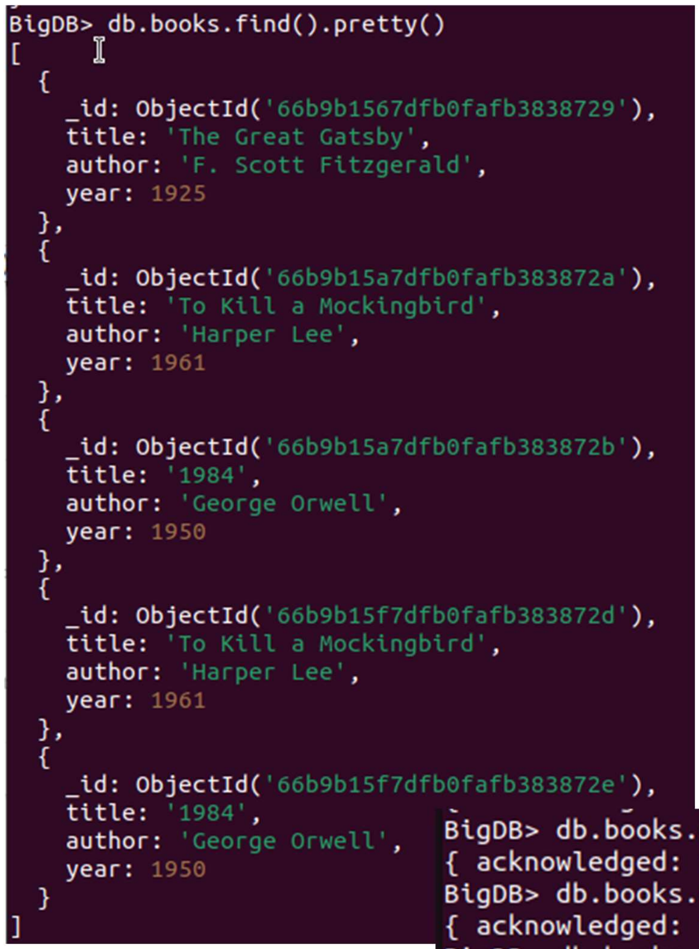
- Find with limiting output


```
BigDB> db.books.find().limit(5).pretty()
[
  {
    _id: ObjectId('66b9b1567dfb0fafb3838729'),
    title: 'The Great Gatsby',
    author: 'F. Scott Fitzgerald',
    year: 1925
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872a'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1961
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872b'),
    title: '1984',
    author: 'George Orwell',
    year: 1950
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872c'),
    title: 'Moby-Dick',
    author: 'Herman Melville',
    year: 1851
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872d'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1961
  }
]
BigDB>
```

- Update Commands: updateOne(), updateMany()

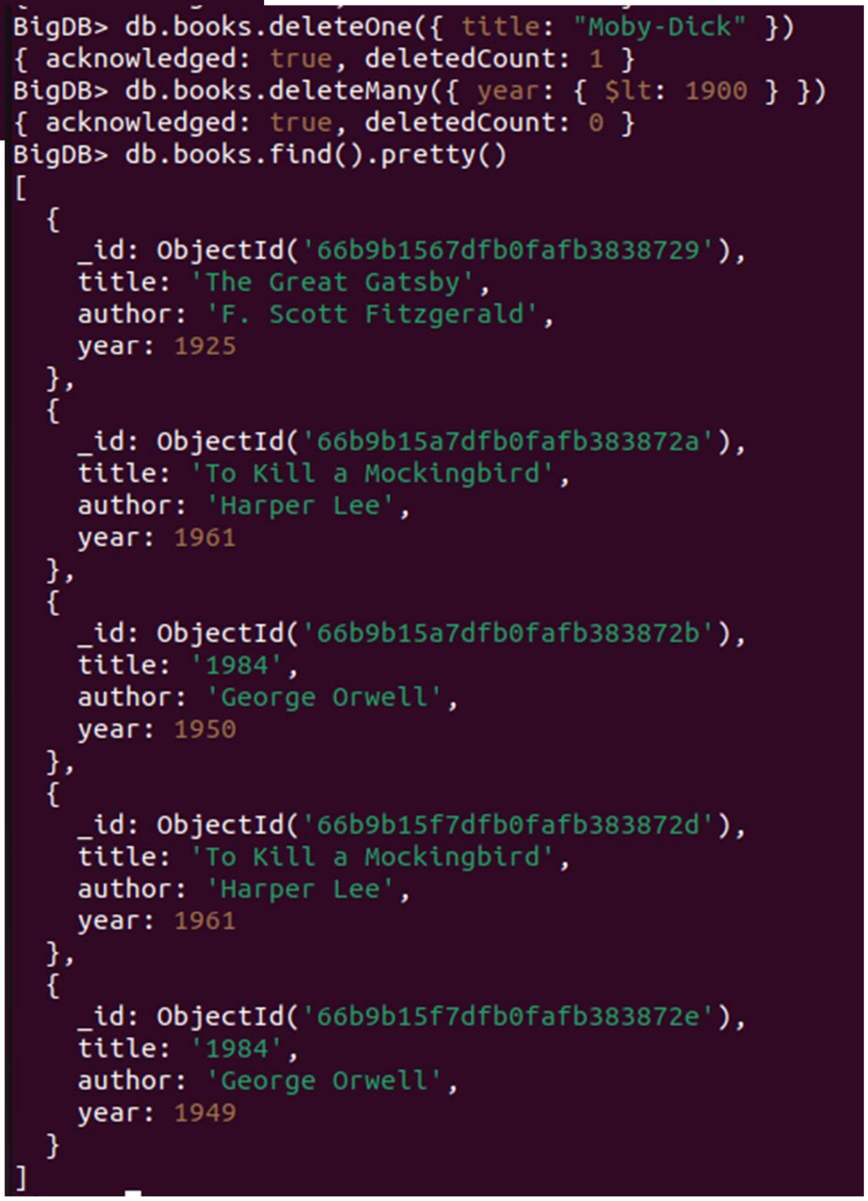

```
BigDB> db.books.updateOne(
...    { title: '1984', author: 'George Orwell', year: 1950 },
...    { $set: { year: 1949 } }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```


```
BigDB> db.books.updateMany(
...    { title: '1984' },
...    { $set: { year: 1950 } }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 2,
  upsertedCount: 0
}
```



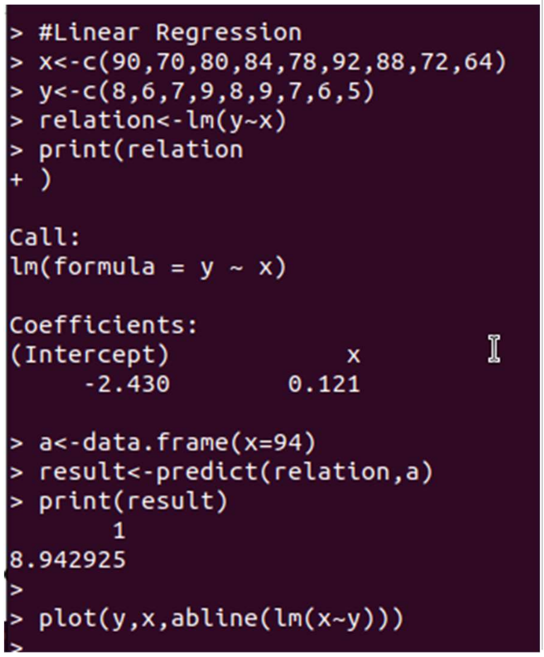
```
BigDB> db.books.find().pretty()
[
  {
    _id: ObjectId('66b9b1567dfb0fafb3838729'),
    title: 'The Great Gatsby',
    author: 'F. Scott Fitzgerald',
    year: 1925
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872a'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1961
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872b'),
    title: '1984',
    author: 'George Orwell',
    year: 1950
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872d'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1961
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872e'),
    title: '1984',
    author: 'George Orwell',
    year: 1950
  }
]
```

- Delete Commands: deleteOne(), deleteMany()



```
BigDB> db.books.deleteOne({ title: "Moby-Dick" })
{ acknowledged: true, deletedCount: 1 }
BigDB> db.books.deleteMany({ year: { $lt: 1900 } })
{ acknowledged: true, deletedCount: 0 }
BigDB> db.books.find().pretty()
[
  {
    _id: ObjectId('66b9b1567dfb0fafb3838729'),
    title: 'The Great Gatsby',
    author: 'F. Scott Fitzgerald',
    year: 1925
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872a'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1961
  },
  {
    _id: ObjectId('66b9b15a7dfb0fafb383872b'),
    title: '1984',
    author: 'George Orwell',
    year: 1950
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872d'),
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1961
  },
  {
    _id: ObjectId('66b9b15f7dfb0fafb383872e'),
    title: '1984',
    author: 'George Orwell',
    year: 1949
  }
]
```

LINEAR REGRESSION →

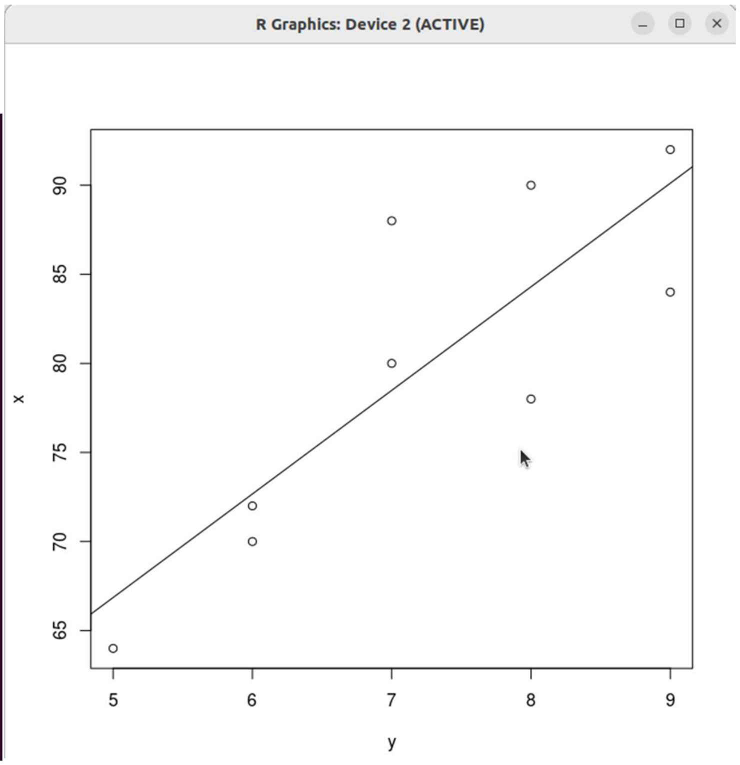


```
> #Linear Regression
> x<-c(90,70,80,84,78,92,88,72,64)
> y<-c(8,6,7,9,8,9,7,6,5)
> relation<-lm(y~x)
> print(relation
+ )

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     -2.430        0.121

> a<-data.frame(x=94)
> result<-predict(relation,a)
> print(result)
       1
8.942925
>
> plot(y,x,abline(lm(x~y)))
>
```



STEP 01 Open R compiler and Load
#Get working Directory:
getwd()

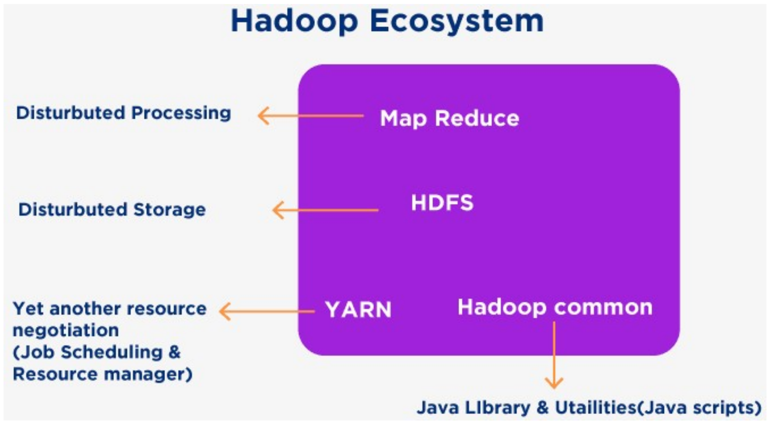
#Download and load the Dataset:
download.file("https://cmu-lib.github.io/os-workshops/reproducible-research/data/interviews_plotting.csv","interviews_plotting.csv", mode = "wb")

#List all Files in Working Directory:
list.files()
# Load the dataset
data <- read.csv("interviews_plotting.csv")

# View the first few rows of the dataset
View(data)

Big Data → It refers to datasets that are large, complex, and rapidly growing, exceeding the capabilities of traditional data processing applications. Key characteristics include volume (large amounts of data), velocity (high speed of data generation), and variety (different types of data).
Hadoop provides a robust platform for organizations to store, process, and analyze big data efficiently, enabling them to derive valuable insights and make data-driven decisions at scale.

Hadoop Ecosystem →
The Hadoop ecosystem refers to a collection of open-source software utilities, frameworks, and tools that work together with Hadoop to solve big data problems. It extends the core capabilities of Hadoop (which includes components like HDFS, YARN, and MapReduce) by providing additional functionalities for data storage, processing, management, and analysis.


Hadoop Ecosystem
Disturbuted Processing
Map Reduce
Disturbuted Storage
HDFS
Yet another resource negotiation (Job Scheduling & Resource manager)
YARN
Hadoop common
Java LIbrary & Utailities(Java scripts)

4 Primary Basic Components
a. **Hadoop Common:** Utilities and libraries used by other Hadoop modules
b. HDFS: Hadoop Distributed File System (HDFS) stores large datasets across nodes and maintans metadata. It consists of two main components: NameNode (stores metadata) and DataNodes (store actual data). This setup on commodity hardware makes Hadoop cost-effective, managing cluster coordination centrally.
c. YARN: Yet Another Resource Negotiator (YARN) manages resources in Hadoop clusters. It includes three components: Resource Manager (allocates resources), Node Manager (manages resources like CPU and memory per machine), and Application Manager (mediates between Resource Manager and Node Manager).
d. MapReduce: MapReduce enables distributed processing of large datasets with parallel algorithms. It uses Map() for data sorting and filtering into key-value pairs, and Reduce() for aggregating and summarizing mapped data, producing manageable outputs from big datasets.

Hadoop Ecosystem Tools Overview →

Tool	Designed/Developed By	Functions
Hadoop Distributed File System (HDFS)	Apache Software Foundation	Distributed storage system for large datasets across nodes, fault-tolerant and scalable.
YARN (Yet Another Resource Negotiator)	Apache Software Foundation	Resource management framework that schedules tasks and manages resources in Hadoop clusters.
MapReduce	Java	Programming model for processing and generating large data sets with a parallel, distributed algorithm on a cluster.
HBase	Apache Software Foundation	NoSQL database for real-time read/write access to large datasets.
Apache Pig	Yahoo! Inc.	Platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs.
Apache Hive	SQL, JDBC	Data warehouse infrastructure built on Hadoop for providing data summarization, query, and analysis.
Apache Spark	Apache Software Foundation	In-memory data processing engine for fast iterative algorithms and interactive data querying.

Apache Kafka	Apache Software Foundation	Distributed streaming platform for handling real-time data feeds.
Apache ZooKeeper	Apache Software Foundation	Centralized service for maintaining configuration information, naming, and providing distributed synchronization.
Sqoop	Apache Software Foundation	Tool for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.
Flume	Apache Software Foundation	Distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data from various sources to Hadoop storage.
Apache Mahout	Apache Community	Scalable machine learning and data mining library.
MLlib (in Apache Spark)	Apache Software Foundation	Distributed machine learning framework on top of Apache Spark for scalable and efficient machine learning computations.
Apache Drill	Apache Software Foundation	Schema-free SQL Query Engine for Hadoop, NoSQL and Cloud Storage.
Oozie	Yahoo! Inc.	Workflow scheduler system to manage Hadoop jobs.
Presto	Facebook	Distributed SQL query engine optimized for ad-hoc analysis at interactive speed.

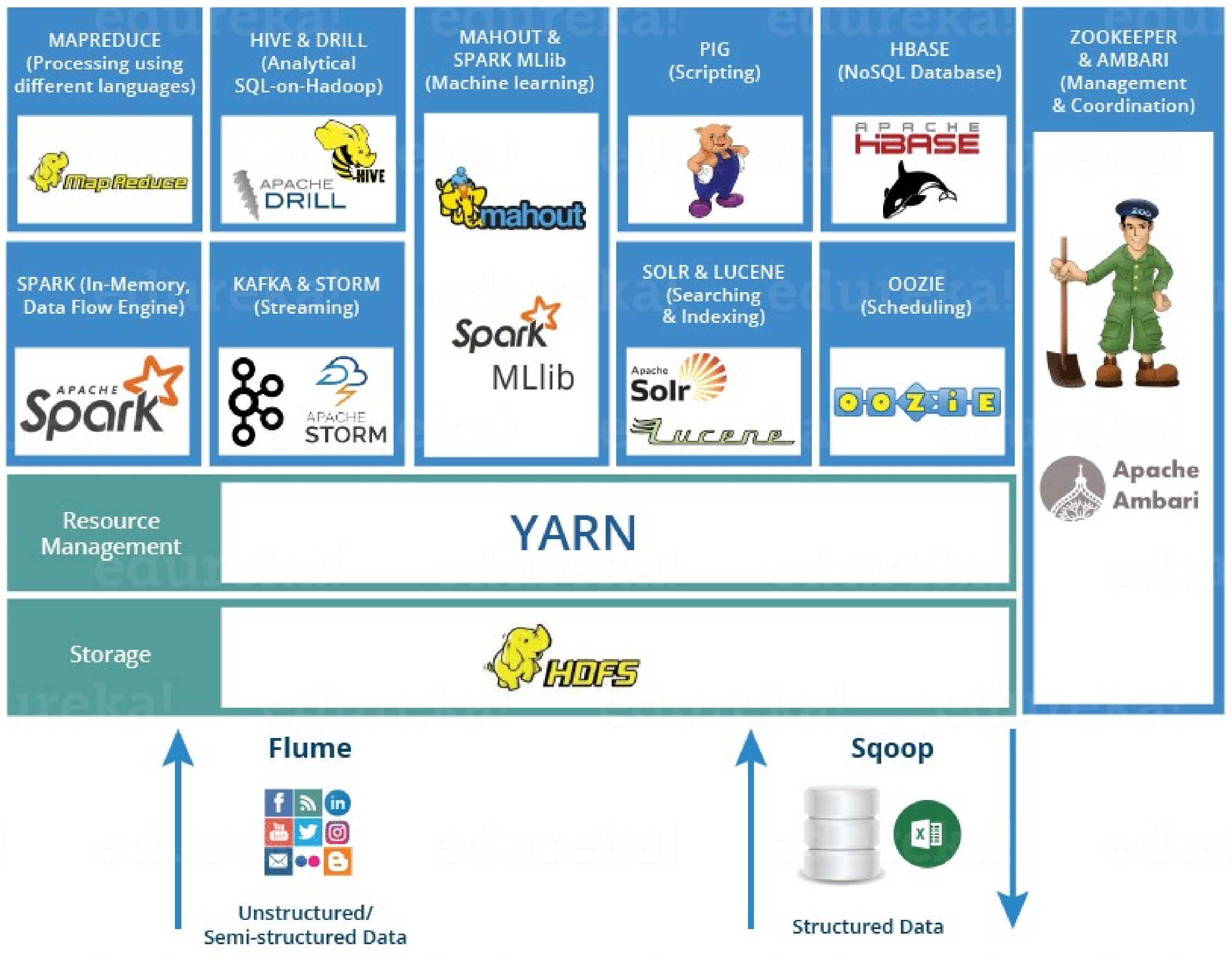

MAPREDUCE (Processing using different languages)
MapReduce
HIVE & DRILL (Analytical SQL-on-Hadoop)
HIVE
APACHE DRILL
MAHOUT & SPARK MLlib (Machine learning)
mahout
Spark MLlib
PIG (Scripting)
HBASE (NoSQL Database)
APACHE HBASE
ZOOKEEPER & AMBARI (Management & Coordination)
Apache Ambari
SPARK (In-Memory, Data Flow Engine)
APACHE Spark
KAFKA & STORM (Streaming)
APACHE STORM
SOLR & LUCENE (Searching & Indexing)
Apache Solr
Lucene
OOZIE (Scheduling)
OOZIE
Resource Management
YARN
Storage
HDFS
Flume
Unstructured/ Semi-structured Data
Sqoop
Structured Data
edureka!

Table summarizing some basic Hadoop commands, their syntax, and their functions:

Function	Command	Syntax	Description
Copying File to Hadoop	hdfs dfs -put	hdfs dfs -put <local-source> <hdfs-destination>	Copies a file or directory from the local file system to HDFS.
Copying File from Hadoop File System	hdfs dfs -get	hdfs dfs -get <hdfs-source> <local-destination>	Copies a file or directory from HDFS to the local file system.
Copy from Hadoop File System and Deleting File	hdfs dfs -getmerge	hdfs dfs -getmerge <hdfs-source> <local-destination>	Merges files from HDFS and copies them to the local file system.
Moving and Displaying Files in HDFS	hdfs dfs -mv	hdfs dfs -mv <source> <destination>	Moves files or directories within HDFS.
Removing Files in HDFS	hdfs dfs -rm	hdfs dfs -rm <hdfs-file>	Deletes a file or directory in HDFS.
Removing a Directory and its Contents	hdfs dfs -rm -r	hdfs dfs -rm -r <hdfs-directory>	Recursively deletes a directory and its contents in HDFS.
Listing Files in HDFS	hdfs dfs -ls	hdfs dfs -ls <hdfs-directory>	Lists files and directories in the specified HDFS directory.
Displaying the Content of a File in HDFS	hdfs dfs -cat	hdfs dfs -cat <hdfs-file>	Displays the content of a file in HDFS.

Additional Notes for Windows

1. **Environment Setup**: Ensure Hadoop is properly installed and configured on your Windows system. This might involve setting environment variables such as HADOOP_HOME.
2. **Command Line Tools**: Use Command Prompt or PowerShell to run these commands. Ensure you have the correct permissions to execute Hadoop commands.
3. **Path Syntax:** In Windows, use backslashes (\) for local file paths, but when interacting with HDFS paths, use forward slashes (/).

MAP REDUCE

What is MapReduce?

MapReduce is a parallel, distributed programming model in the Hadoop framework that can be used to access the extensive data stored in the Hadoop Distributed File System (HDFS). The Hadoop is capable of running the MapReduce program written in various languages such as Java, Ruby, and Python. One of the beneficial factors that MapReduce aids is that MapReduce programs are inherently parallel, making the very large scale easier for data analysis.

MapReduce is a programming model and processing framework used for handling large-scale data processing across distributed computing environments. Developed by Google and popularized through its open-source implementation in Hadoop, MapReduce allows for the processing of massive amounts of data by breaking it down into smaller, manageable pieces.

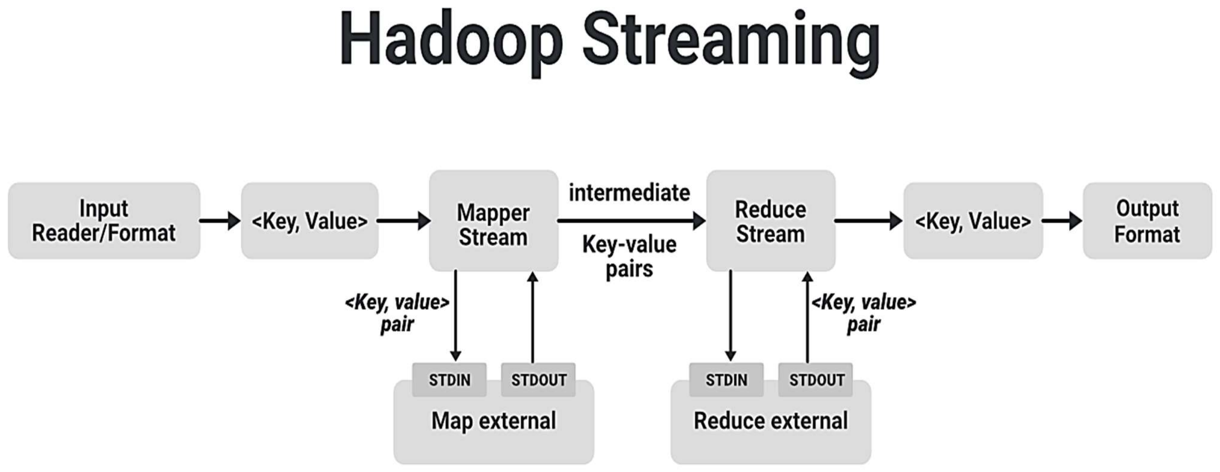
When the MapReduce programs run in parallel, it speeds up the process. The process of running MapReduce programs is explained below.

- **Dividing the input into fixed-size chunks:** Initially, it divides the work into equal-sized pieces. When the file size varies, dividing the work into equal-sized pieces isn't the straightforward method to follow, because some processes will finish much earlier than others while some may take a very long run to complete their work. So one of the better approaches is that one that requires more work is said to split the input into fixed-size chunks and assign each chunk to a process.

- **Combining the results:** Combining results from independent processes is a crucial task in MapReduce programming because it may often need additional processing such as aggregating and finalizing the results.

Key components of MapReduce
There are two key components in the MapReduce. The MapReduce consists of two primary phases such as the map phase and the reduces phase. Each phase contains the key-value pairs as its input and output and it also has the map function and reducer function within it.

- Mapper: Mapper is the first phase of the MapReduce. The Mapper is responsible for processing each input record and the key-value pairs are generated by the InputSplit and RecordReader. Where these key-value pairs can be completely different from the input pair. The MapReduce output holds the collection of all these key-value pairs.
- Reducer: The reducer phase is the second phase of the MapReduce. It is responsible for processing the output of the mapper. Once it completes processing the output of the mapper, the reducer now generates a new set of output that can be stored in HDFS as the final output data.



Key Concepts of MapReduce

1. Map Phase:

Function: The map function processes input data in parallel. It takes a set of data (often key-value pairs) and applies a transformation to it.
Output: It emits intermediate key-value pairs.
Example: In a word count application, the map function reads a text file and outputs each word with a count of 1.

2. Shuffle and Sort Phase:

Function: After the map phase, the intermediate key-value pairs are shuffled and sorted. This phase groups all values with the same key together.
Purpose: To prepare the data for the reduce function by ensuring that all values associated with a particular key are brought together.

3. Reduce Phase:

Function: The reduce function processes the grouped key-value pairs. It takes the intermediate key-value pairs produced by the map phase and combines them to produce the final result.
Output: It emits the final key-value pairs.
Example: In the word count example, the reduce function sums up all counts associated with each word to produce the final count for each word.

Example Workflow: Word Count

1. **Map Phase:**

Input: A line of text: "hello world"

Output: Key-value pairs: ('hello', 1) and ('world', 1)

2. **Shuffle and Sort Phase:**

Intermediate data is grouped by key: { 'hello': [1, 1], 'world': [1] }

3. **Reduce Phase:**

Input: Grouped key-value pairs

Output: Aggregated results: ('hello', 2) and ('world', 1)

Advantages of MapReduce

1. **Scalability**: It can handle vast amounts of data by distributing the workload across many machines.
2. **Fault Tolerance**: It is designed to handle failures gracefully, redistributing tasks if necessary.
3. **Simplicity**: The programming model abstracts the complexity of distributed computing, making it easier to develop parallel algorithms.

Implementation

- **Hadoop**: The most well-known open-source implementation of MapReduce, developed by the Apache Software Foundation. Hadoop provides a distributed file system (HDFS) and a processing framework for executing MapReduce jobs.
- **Google Cloud Dataflow**: A fully managed service for processing data with a similar model to MapReduce, but with additional features and optimizations.

DGIM algorithm (*Datar-Gionis-Indyk-Motwani Algorithm)*

Designed to find the number 1's in a data set. This algorithm uses O($\log^2$N) bits to represent a window of N bit, allowing to estimate the number of 1's in the window with an error of no more than 50%. So this algorithm gives a 50% precise answer.
n the DGIM algorithm, each bit that arrives has a timestamp, for the position at which it arrives. if the first bit has a timestamp 1, the second bit has a timestamp 2 and so on.
The positions are recognized with the window size N (the window sizes are usually taken as a multiple of 2). The windows are divided into buckets consisting of 1's and 0's.

Rules for forming Buckets:

1. The right side of the bucket should always start with 1. (if it starts with a 0,it is to be neglected) E.g. · 1001011 → a bucket of size 4 ,having four 1's and starting with 1 on its right end.
2. Every bucket should have at least one 1, else no bucket can be formed.
3. All buckets should be in powers of 2.
4. The buckets cannot decrease in size as we move to the left. (move in increasing order towards left)

DGIM Algorithm:

- The right end of a bucket always starts with a position with a 1.
- Number of 1s must be a power of 2.
- Either one or two buckets with the same power of 2 number of 1s exists.
- Buckets do not overlap in timestamps
- Buckets are stored by size.
- Buckets disappear when their end-time is N time units in the past.

Overview of the algorithm:

1. Bucket Creation and Update:

- When a 1 is observed, create or update a bucket. Each bucket has a timestamp and represents a period during which 1s were observed.
- Buckets are merged if they have the same size or if the time difference between them is small.

2. Bucket Merging:
   - Merging reduces the number of buckets by combining smaller buckets into larger ones, ensuring that only buckets of sizes in powers of 2 are maintained.
   - buckets are merged to keep memory usage within bounds.
3. Removing Old Buckets:
   - Buckets older than the current window size N are removed from the system.
4. Estimating Number of 1s:
   - The estimated number of 1s is calculated based on the remaining buckets and their sizes.
   - The formula used is: Estimate = $2^{\text{max\_register\_value}}$, where `max_register_value` is the maximum size of the buckets.

Advantages

1. Space Efficiency: Uses a fixed number of buckets with $O(\log^2 N)$ bits, making it highly memory-efficient.
2. Scalability: Effectively handles high-speed data streams and large window sizes with minimal increases in memory usage.

Disadvantages

1. Estimation Error: The result is an estimate with an error margin of up to 50%.
2. Complexity: Merging and managing buckets can be complex to implement.

Application:

- Trending new articles on Twitter
- Trending sales on Amazon

Flajolet-Martin algorithm

The Flajolet-Martin algorithm, also known as the FM algorithm, is a probabilistic method developed by Philippe Flajolet and G. Nigel Martin in 1983. It is primarily used for estimating the number of unique elements in a large dataset or data stream with a single pass, consuming space logarithmic in the maximal number of possible distinct elements. This makes it particularly useful for handling big data scenarios where traditional methods might not be feasible due to memory constraints or computational inefficiency. It approximates the number of unique objects in a stream or a database in one pass. If the stream contains n elements with m of them unique, this algorithm runs in $O(n)$ time and needs $O(log(m))$ memory.

Advantages

- **Scalability:** Handles large datasets efficiently without requiring full dataset storage.
- **Memory Efficiency:** Uses minimal memory by employing bit manipulations and hash functions.
- **Speed:** Ideal for real-time applications due to its fast computation.

Disadvantages

- **Accuracy:** Provides estimates, not exact counts; accuracy depends on factors like hash function and bit vector size.
- **Dataset Sensitivity:** Accuracy varies with dataset distribution; less effective on skewed datasets.
- **Hash Function Selection:** Performance relies on the choice of hash functions for balance between accuracy and efficiency.
- **Limited Applicability:** Primarily for estimating unique counts; does not provide details about specific elements or frequencies.

Use Cases of the Flajolet-Martin Algorithm

1. **Big Data Analytics:**

Problem: Traditional counting methods are too slow for real-time data streams.
Solution: Flajolet-Martin estimates distinct elements quickly, even with massive, continuous data flows.

2. **Network Traffic Monitoring:**

Problem: Monitoring unique IP addresses requires significant memory.
Solution: Flajolet-Martin estimates the number of unique IPs efficiently without storing each address.

3. **Database Deduplication:**

Problem: Identifying unique entries in large databases can be resource-intensive.
Solution: Flajolet-Martin provides an efficient estimate of unique records, speeding up deduplication.

4. **Online Advertising:**

Problem: Tracking unique visitor or impressions is crucial for accurate campaign measurement.
Solution: Flajolet-Martin estimates unique counts to help advertisers gauge reach and effectiveness.

5. **Biodiversity Studies:**

Problem: Estimating distinct species in large datasets is challenging.
Solution: Flajolet-Martin approximates species counts without manual identification.

6. **Machine Learning:**

Problem: Preprocessing large datasets involves managing unique features efficiently.
Solution: Flajolet-Martin estimates the number of unique features, optimizing feature hashing.

Algorithm Purpose: Estimate cardinality (number of unique elements) in large dataset or data stream.
Steps

1. **Hash Function**: Use a hash function to map each input element to a binary string.
2. **Binary Representation:** For each element, compute its hash value and convert it to a binary string.
3. **Count Trailing Zeros:**
   - For each binary string, count the number of trailing zeros.
   - Track the maximum count of trailing zeros (R) encountered across all elements.
4. **Estimate Unique Count:**
   - The estimate of the number of unique elements is given by the formula: Estimate=2R\text{Estimate} = 2^REstimate=2R
   - A correction factor (often φ ≈ 0.77351) can be applied to improve accuracy: Adjusted Estimate=ϕ×2R\text{Adjusted Estimate} = \phi \times 2^RAdjusted Estimate=ϕ×2R

NoSQL MongoDB

NoSQL : It's a non-relational database designed to handle unstructured or semi-structured data with flexible schema requirements. Unlike traditional relational databases, NoSQL databases use various data models, such as key-value pairs, document stores, column-family stores, or graph databases, to optimize scalability and performance for specific use cases. They are often used in applications requiring high performance, scalability, and flexibility, such as big data analytics and real-time web applications.

What is MongoDB?

MongoDB is a popular, open-source NoSQL database management system designed to store and manage large volumes of unstructured or semi-structured data. Unlike traditional relational databases, MongoDB uses a flexible, schema-less data model that stores data in JSON-like documents (BSON format) rather than tables. This document-oriented approach allows for easier scalability, as MongoDB can handle a wide variety of data types and structures without requiring a predefined schema. It supports dynamic queries, indexing, and replication, making it well-suited for applications requiring high performance and horizontal scalability. MongoDB also offers features like built-in sharding for distributing data across multiple servers, automatic failover, and rich query capabilities, which make it a popular choice for modern web applications, real-time analytics, and big data projects.

Feature	SQL Databases	NoSQL Databases
Data Model	Relational (table-based)	Non-relational (document, key-value, graph, wide-column)
Schema	Fixed schema, predefined structure	Dynamic schema, flexible structure
Scalability	Vertically scalable (adding more resources to a single server)	Horizontally scalable (adding more servers to the database)
Normalization	Data normalization to reduce redundancy	Denormalization for performance, allowing redundancy
Query Language	SQL (Structured Query Language)	Various query languages (e.g., NoSQL-specific, JavaScript, etc.)
ACID Compliance	Generally ACID-compliant (Atomicity, Consistency, Isolation, Durability)	May or may not be ACID-compliant; often follow BASE model (Basically Available, Soft state, Eventually consistent)
Data Integrity	Enforced through constraints and relationships	Responsibility of the application layer
Use Cases	Suitable for complex queries, transactions, and structured data	Ideal for hierarchical data storage, large-scale data, and unstructured data
Examples	MySQL, PostgreSQL, Oracle, SQL Server	MongoDB, Cassandra, Redis, Couchbase
Performance	Optimized for complex queries and transactions	Optimized for large datasets, scalability, and real-time processing
Flexibility	Less flexible due to fixed schema	More flexible, supports evolving data models
Joins	Supports JOIN operations to link tables	No native JOINs; relationships handled at the application level
Storage Type	Row-based storage	Varies by type: document, column, key-value, or graph-based
Backup and Restore	Generally supports comprehensive backup and restore functionalities	Backup and restore processes vary by database type and implementation

Features of MongoDB

- Schema-less Database: MongoDB does not require a predefined schema before you add data to the database. Documents within a single collection can have different fields and structures, offering great flexibility.
- Document-Oriented: Data is stored in BSON format (a binary representation of JSON-like documents), allowing for rich, nested data structures.
- Indexing: Any field in a MongoDB document can be indexed, improving the performance of search operations.
- Scalability: MongoDB supports horizontal scalability through sharding, distributing data across multiple servers.
- Relication: High availability and redundancy are achieved through replication, keeping copies of data on different servers.
- Aggregation: MongoDB provides powerful aggregation tools that allow for complex transformations and analyses of data.

- **High Performance:** Designed for scalability and performance, MongoDB works efficiently with large datasets.

Uses of MongoDB: MongoDB is widely used across various industries due to its flexibility, scalability, and performance:

- **Content Management Systems (CMS):** Ideal for handling dynamic content types and structures.
- **E-commerce Platforms:** Efficiently manages large amounts of product data and user profiles.
- **Real-Time Analytics:** Suitable for applications requiring high-performance data ingestion and querying.
- **Internet of Things (IoT):** Can efficiently store and process vast amounts of sensor data.
- **Gaming Applications**: Supports complex data structures for player profiles and game states.
- **Log Managem**ent and Analysis: Handles large volumes of unstructured log data.
- **Customer Relationship Management (CRM):** Manages customer interactions and sales pipelines effectively.
- **Social Networks:** Supports complex relationships and real-time interactions.
- **Big Data Applications:** Integrates well with big data technologies for advanced analytics.
- **Healthcare Systems:** Manages patient records and clinical data efficiently.

Queries in Big Data Analytics

The Data Stream Model → In it, some or all of the input data that are to be operated on are not available for random access from disk or memory, but rather arrive as one or more continuous data streams. Data streams differ from the conventional stored relation model in several ways:

- The data elements in the stream arrive online.
- The system has no control over the order in which data elements arrive to be processed, either within a data stream or across data streams.
- Data streams are potentially unbounded in size.
- Once an element from a data stream has been processed it is discarded or archived — it cannot be retrieved easily unless it is explicitly stored in memory, which typically is small relative to the size of the data streams.

Operating in the data stream model does not preclude the presence of some data in conventional stored relations. Often, data stream queries may perform joins between data streams and stored relational data. For the purposes of this paper, we will assume that if stored relations are used, their contents remain static. Thus, we preclude any potential transaction-processing issues that might arise from the presence of updates to stored relations that occur concurrently with data stream processing.

Queries in Big Data Analytics

It is a structured request for specific data or insights from large, complex datasets. It can involve retrieving, filtering, aggregating, or analyzing data to answer questions or generate reports. Queries can be executed in batch mode on historical data, in real-time on streaming data, or interactively for exploratory analysis.

A. The first distinction is between one-time queries and continuous queries.

One-time queries (a class that includes traditional DBMS queries) are queries that are evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user. Continuous queries, on the other hand, are evaluated continuously as data streams continue to arrive. Continuous queries are the more interesting class of data stream queries, and it is to them that we will devote most of our attention. The answer to a continuous query is produced over time, always reflecting the stream data seen so far. Continuous query answers may be stored and updated as new data arrives, or they may be produced as data streams themselves. Sometimes one or the other mode is preferred. For example, aggregation queries may involve frequent changes to answer tuples, dictating the stored approach, while join queries are monotonic and may produce rapid, unbounded answers, dictating the stream approach.

B. **The second distinction is between predefined queries and ad hoc queries.**

A predefined query is one that is supplied to the data stream management system before any relevant data has arrived. Predefined queries are generally continuous queries, although scheduled one-time queries can also be predefined. Ad hoc queries, on the other hand, are issued online after the data streams have already begun. Ad hoc queries can be either one-time queries or continuous queries. Ad hoc queries complicate the design of a data stream management system, both because they are not known in advance for the purposes of query optimization, identification of common subexpressions across queries, etc., and more importantly because the correct answer to an ad hoc query may require referencing data elements that have already arrived on the data streams (and potentially have already been discarded). **Thus in this model, queries can be categorized as follows:**

- **One-Time Queries**: Evaluated once over a snapshot of data, returning the result immediately.
- **Continuous Queries**: Continuously evaluated as data arrives, providing ongoing results that reflect the data seen so far. Results can be stored or produced as new data arrives.

Additionally, queries can be:

- **Predefined Queries**: Supplied before any relevant data arrives, often continuous but can also be scheduled one-time queries.
- **Ad Hoc Queries**: Issued online after data streams have started, potentially complicating query processing due to the need to access already processed or discarded data.

Attribute	One-Time Query	Continuous Query	Ad-Hoc Query
Description	Executes a specific query or analysis once.	Simulates periodic data updates and querying.	Performs dynamic queries based on changing conditions.
Purpose	To retrieve or analyze data for a single instance.	To monitor and query data over time or in near real-time.	To answer specific questions with variable conditions.
Execution Frequency	Single execution	Periodic or continuous	As needed
Parameters	Query conditions and data columns.	Time intervals, data update logic.	Dynamic query conditions and parameters.

Library package used in R → dplyr

The dplyr package in R is a powerful and popular tool for data manipulation and transformation.
dplyr is an R package that simplifies data manipulation and transformation tasks. It provides a set of intuitive and readable functions for filtering, selecting, arranging, summarizing, and mutating data. It is particularly useful for working with data frames and tibbles. It's part of the "tidyverse," a collection of R packages designed for data science.

Key Functions in dplyr

1. select(): Choose specific columns from a data frame.
2. filter(): Subset rows based on logical conditions.
3. arrange(): Sort rows by one or more columns.
4. mutate(): Add or transform columns.
5. summarize() (or summarise()): Aggregate data and produce summary statistics.
6. group_by(): Group data by one or more variables for aggregation and summary operations.
7. rename(): Rename columns in a data frame.

Usage in Query Processing → dplyr is often used to streamline data manipulation and query processing.

- **Data Manipulation:** dplyr functions can be applied to data frames and tibbles to perform various manipulations. The syntax is designed to be readable and expressive.
- **Query Processing with Databases:** dplyr integrates with databases through the dbplyr package. This allows you to use dplyr syntax to build queries that are translated into SQL and executed directly on the database server.

Linear Regression

Concept: Linear regression is a statistical method for modeling the relationship between a dependent variable Y and one independent variable X. The model assumes that Y is linearly dependent on X, which can be expressed with the equation:

$Y=\beta_0+\beta_1X+\epsilon$

where:

- β_0 is the y-intercept (the value of Y when X=0).
- β_1 is the slope of the line (the change in Y for a one-unit change in X).
- ϵ epsilon is the error term, representing the variability in Y not explained by X.

Assumptions:

1. **Linearity**: The relationship between X and Y is linear.
2. **Independence**: Observations are independent of each other.
3. **Homoscedasticity**: The variance of the residuals (errors) is constant across all levels of X.
4. **Normality**: The residuals of the model are normally distributed.

Goal: The goal is to estimate β0and β1 such that the sum of squared differences between observed values and predicted values (residual sum of squares) is minimized. This is typically done using the method of least squares.

Multiple Regression

Concept: Multiple regression extends the concept of linear regression to include multiple independent variables. The model is used to understand how multiple predictors influence the dependent variable Y. The equation for multiple regression is:

$Y=\beta_0+\beta_1X_1+\beta_2X_2+\cdots+\beta_pX_p+\epsilon$

where:

- $X_1, X_2, \ldots, X_p$ are the independent variables.
- $\beta_1, \beta_2, \ldots, \beta_p$ are the coefficients corresponding to these independent variables.

Assumptions:

1. **Linearity**: The relationship between the dependent variable and each of the independent variables is linear.
2. **Independence**: Observations are independent of each other.
3. **Homoscedasticity**: The variance of the residuals is constant for all levels of the independent variables.
4. **Normality**: The residuals are normally distributed.
5. **No multicollinearity**: Independent variables are not too highly correlated with each other.

Goal: The goal is to find the best-fitting line in a multidimensional space, such that the sum of squared residuals is minimized. This allows for understanding the influence of each predictor on the dependent variable while holding other predictors constant.

Logistic Regression

Concept: Logistic regression is used for binary classification problems, where the dependent variable is categorical with two possible outcomes (e.g., success/failure, yes/no). It models the probability that a given observation falls into one of the categories. The model is:

$\text{logit}(P)=\ln(P/(1-P)) = \beta_0+\beta_1X_1+\beta_2X_2+\cdots+\beta_pX_p$

where:

- P is the probability of the dependent variable being 1 (e.g., success).
- The logit function, $\ln(P/(1-P))$, transforms the probability into a continuous scale.

Assumptions:

1. **Linearity of the logit**: The logit (log-odds) of the outcome is a linear combination of the independent variables.
2. **Independence**: Observations are independent of each other.
3. **No multicollinearity**: Independent variables are not too highly correlated with each other.
4. **Binary outcome**: The dependent variable is binary (two possible outcomes).

Goal: The goal is to estimate the coefficients $\beta_0,\beta_1,\ldots,\beta_p$ such that the likelihood of the observed data is maximized. This is done using methods like maximum likelihood estimation.

Summary

- **Linear Regression**: Predicts a continuous dependent variable using one independent variable. Assumes linear relationships.
- **Multiple Regression**: Extends linear regression to multiple predictors. Assumes linear relationships and checks for multicollinearity.

Logistic Regression: Used for binary classification. Models the probability of an outcome using a logit function and estimates the coefficients using maximum likelihood.

PROJECT OVERVIEW →

It's is a data analytics project that automates the detection and clustering of test-pads on Printed Circuit Boards using big data tools and image processing techniques. The project aims to help engineers in PCB maintenance and testing, particularly when dealing with PCBs that lack schematics or CAD data.

Core Problem:

- PCBs often need maintenance due to damage from heat, vibrations, and external forces
- Traditional manual inspection is time-consuming and error-prone
- Many PCBs lack proper documentation or CAD data
- Engineers need accurate test-pad locations for using Flying Probe Testers

Solution Approach: The project implements a two-stage clustering approach:

1. Stage 1: Identifies grey pixels that represent test-pads
2. Stage 2: Clusters these grey pixels to identify individual test-pad locations

Role of Big Data Analytics:

1. Large-scale Image Processing:
   - Processing high-resolution PCB images generates massive pixel-level data
   - Each pixel has X, Y coordinates and RGB values
   - Example: A single PCB image contained 71,040 pixels
2. Distributed Computing:
   - Uses Hadoop ecosystem for distributed storage and processing
   - Leverages PySpark for parallel processing of image data
   - Enables processing of multiple PCB images simultaneously
3. Advanced Analytics:
   - Implements K-means clustering for pixel classification
   - Uses machine learning for pattern recognition
   - Performs real-time data processing and visualization

Hadoop Ecosystem Tools and Functions

Tool/Component	Function	Role in Project
HDFS (Hadoop Distributed File System)	Distributed storage	• Stores PCB images and processed data • Splits large image datasets across nodes. • Provides fault tolerance and data replication
PySpark	Distributed processing framework	• Handles image processing tasks. • Implements K-means clustering. • Provides DataFrame operations for data manipulation. • Enables parallel processing of pixel data
YARN (Yet Another Resource Negotiator)	Resource management	• Manages cluster resources. Schedules processing jobs. Handles resource allocation across nodes
Spark MLlib	Machine learning library	• Provides distributed K-means clustering implementation. • Enables scalable machine learning operations. • Handles feature extraction and transformation
Spark SQL	Structured data processing	• Processes structured pixel data. • Enables SQL-like queries on image data. • Provides DataFrame API for data manipulation
Apache Hadoop Core	Base framework	• Provides basic MapReduce functionality. • Handles distributed processing coordination. • Manages node communication

Project Architecture Flow:

1. Input Layer:
   - PCB images are loaded into HDFS
   - Images are converted to RGB format
2. Processing Layer:
   - PySpark extracts pixel data (X, Y, R, G, B)
   - Data is transformed into Spark DataFrames
   - K-means clustering identifies grey pixels
   - Second clustering identifies test-pad locations
3. Output Layer:
   - Results are visualized using Matplotlib
   - Test-pad coordinates are exported for Flying Probe Testers
   - Data is stored back in HDFS for future use

Key Achievements:

- 93.75% accuracy in test-pad detection
- Successfully identified 120 test-pads on sample PCB
- Automated process reducing engineer man-hours
- Scalable solution for industrial applications

The project demonstrates effective use of big data technologies for solving a real-world industrial problem, combining image processing, machine learning, and distributed computing to create a practical solution for PCB maintenance and testing.

Project Workflow:

1. Image Processing Stage:

- Input: PCB image
- Process: The Image-Processing.py script:
    - Converts image to RGB format
    - Extracts pixel data (X, Y coordinates and RGB values)
    - Creates a distributed dataframe using PySpark

2. Two-Stage Clustering:

Stage 1 - Grey Pixel Detection:

- Uses K-means clustering (k=2) to separate grey pixels from non-grey pixels
- Grey pixels represent potential test-pads
- Features used: RGB values

Stage 2 - Test-Pad Clustering:

- Uses K-means clustering (k=170) on identified grey pixels
- Features used: X-Y coordinates
- Filters out small clusters (< 10 points) to remove noise
- Result: Identifies individual test-pad locations

3. Visualization:

- Creates scatter plots of detected test-pads
- Uses color coding to show cluster predictions

The MapReduce implementation works in these key steps:

1. Map Phase:

- Splits large PCB image into manageable chunks
- Each mapper processes a chunk of the image
- Extracts pixel data (position and RGB values)
- Emits key-value pairs: (pixel_position, RGB_values)

2. Reduce Phase:

- Groups pixel data by position
- Applies grey pixel detection logic
- Combines results from all mappers
- Outputs coordinates of identified grey pixels

3. Clustering Phase:

- Takes reduced data (grey pixels)
- Applies distributed K-means clustering using PySpark
- Identifies test-pad locations

Benefits of this MapReduce approach:

1. Scalability: Can handle large PCB images by processing chunks in parallel
2. Fault Tolerance: If a node fails, work can be redistributed
3. Load Distribution: Evenly distributes image processing across cluster
4. Memory Efficiency: Processes image in chunks rather than loading entire image

The implementation leverages Hadoop's distributed storage (HDFS) and PySpark's distributed computing capabilities to process large PCB images efficiently. This is particularly useful when dealing with high-resolution PCB images or processing multiple PCBs simultaneously.