**Experiment 2: Word Count using map reduce**

**Aim:** To write a program that counts the occurrences of each word in a large text file using the MapReduce framework.

**Theory: What is MapReduce?**

MapReduce is a programming model for processing huge datasets in parallel across a cluster of computers. It breaks a big problem down into two main phases: the **Map** phase and the **Reduce** phase.

Think of it like taking a massive census.

1. **Input Splits:** You can't give the whole country's census forms to one person. You split them into "Input Splits" (e.g., one box of forms for each city).
2. **Map Phase (The "Mappers"):** You hire many "Mappers" (one for each city) to do the initial tally. Each Mapper reads their box of forms, ignores most of the information, and just outputs a simple note: (person's age, 1) for every form. This creates millions of intermediate (age, 1) pairs .
   * **For Word Count:** The Mapper reads a line of text, splits it into words, and for each word, it outputs a key-value pair: (word, 1).
3. **Shuffle & Sort Phase:** Before you can reduce, you must group all the notes. This phase automatically collects all the intermediate pairs and **groups them by their key**. All the (age, 1) notes for "age 30" are put into one big pile: (age\_30, [1, 1, 1, ...]).
   * **For Word Count:** All the (Hello, 1) pairs are grouped: (Hello, [1, 1, 1]) .
4. **Reduce Phase (The "Reducers"):** You hire a few "Reducers" to do the final sum. The "age 30" Reducer gets the pile (age\_30, [1, 1, 1, ...]), sums the 1s, and produces the final, single answer: (age\_30, 4520).
   * **For Word Count:** The "Hello" Reducer gets (Hello, [1, 1, 1]), sums the list, and outputs the final result: (Hello, 3)

**Code2:**

text = input("Enter sentence:")

words = text.lower().split()

word\_freq = {}

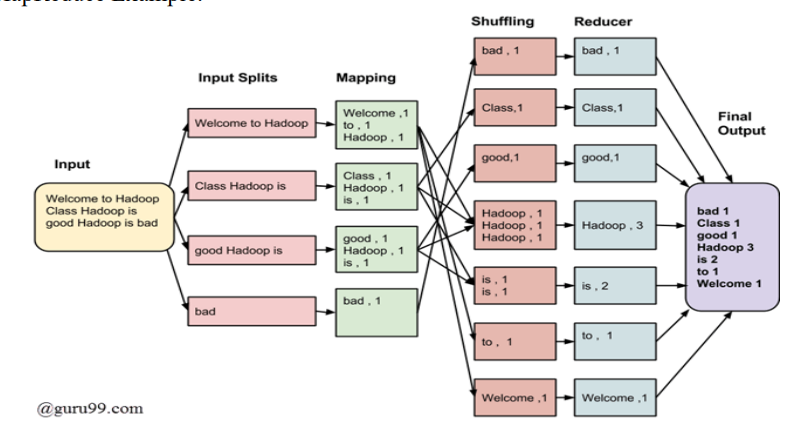
for word in words:

    word\_freq[word] = word\_freq.get(word,0)+1

print("Word frequency:")

for word, freq in word\_freq.items():

    print(f"{word}: {freq}")

****

**Experiment 3: Data Visualization in R**

**Aim:** To transform raw data into graphical or visual formats (like charts, graphs, and plots) to help understand, interpret, and gain insights from the data.

**Theory**

Data visualization is the process of representing data visually to identify patterns, trends, and relationships that might not be obvious from raw numbers.

* **Bar Plot:** A bar plot is used to compare **categorical data**. It shows the frequency (count) of items in each category as a rectangular bar, where the height of the bar is proportional to the count.
* **Box Plot (Box-and-Whisker Plot):** A box plot displays the statistical distribution of a **numeric variable**. It's excellent for comparing distributions between different categories . The "box" shows the interquartile range (IQR, from 25th to 75th percentile), the line inside is the median (50th percentile), and the "whiskers" show the minimum and maximum values (excluding outliers).
* **Scatter Plot:** A scatter plot visualizes the relationship between two **numeric (continuous) variables**. Each point on the plot represents one observation (or row). It's used to identify correlations (e.g., if one variable goes up, does the other go up too?) .
* **Pie Chart:** A pie chart shows **proportions**. It displays a single categorical variable as slices of a pie, where the size of each slice is proportional to the count or percentage of that category.

**Using Your Own Small Dataset**

Instead of loading a CSV, you can create your own small table (a data.frame) directly in R.

**Data<-read.csv(file.choose())**

**Code3:**

**Step 1: Create Your Custom Data**

student\_data <- data.frame(

Name = c("Anna", "Ben", "Clara", "David", "Emily", "Frank", "Grace", "Henry"),

Hours\_Studied = c(4, 6, 7, 3, 8, 5, 6, 2),

Grade = c(75, 85, 90, 65, 95, 80, 82, 55),

Gender = c("Female", "Male", "Female", "Male", "Female", "Male", "Female", "Male"))

head(student\_data)

str(student\_data)

**Step 2: Plotting Examples with student\_data**

**1. Bar Plot**

gender\_counts <- table(student\_data$Gender)

barplot(gender\_counts,

main = "Number of Students by Gender",

xlab = "Gender",

ylab = "Count",

col = c("lightpink", "lightblue"))

**2. Box Plot**

boxplot(Grade ~ Gender,

data = student\_data,

main = "Grade Distribution by Gender",

xlab = "Gender",

ylab = "Grade",

col = c("lightpink", "lightblue"))

**3. Scatter Plot**

plot(student\_data$Hours\_Studied, student\_data$Grade,

main = "Hours Studied vs. Grade",

xlab = "Hours Studied",

ylab = "Grade",

col = "purple")

**4. Pie Chart**

# 1. Count the number of students by Gender

gender\_counts <- table(student\_data$Gender)

# 2. Create the pie chart

pie(gender\_counts,

main = "Proportion of Students by Gender",

col = c("lightpink", "lightblue"))

# Load the data first

data(iris)

# --- 1. Bar Plot ---

# (Counts how many of each Species)

species\_counts <- table(iris$Species)

barplot(species\_counts,

main = "Count of Iris Species",

xlab = "Species",

ylab = "Count",

col = "steelblue")

# --- 2. Box Plot ---

# (Compares Sepal.Length for each Species)

boxplot(iris$Sepal.Length ~ iris$Species,

main = "Sepal Length by Iris Species",

xlab = "Species",

ylab = "Sepal Length",

col = c("pink", "lightblue", "lightgreen"))

# --- 3. Scatter Plot ---

# (Checks relationship between Petal.Length and Sepal.Length)

plot(iris$Petal.Length, iris$Sepal.Length,

main = "Sepal Length vs. Petal Length",

xlab = "Petal Length",

ylab = "Sepal Length",

col = "blue")

# --- 4. Pie Chart ---

# (Shows proportion of each Species)

pie(species\_counts,

main = "Proportion of Iris Species",

col = c("pink", "lightblue", "lightgreen"))

**Experiment 4: Flajolet-Martin Algorithm (Python)**

**Aim:** To study and implement the Flajolet-Martin (FM) algorithm, which is used to approximate the number of **unique (distinct) elements** in a data stream in a single pass.

**Theory**

The Flajolet-Martin algorithm is a **probabilistic** algorithm, meaning it gives an approximation, not an exact count. Its main advantage is **memory efficiency**—it can estimate the number of unique items in a massive stream using a very small, fixed amount of memory.

The algorithm works as follows :

1. **Hash:** For each element in the data stream, a hash function is applied to map it to an integer.
2. **Binary:** This integer hash value is converted to its binary (base-2) string representation.
3. **Find Trailing Zeros (R):** The algorithm counts the number of zeros at the *end* of the binary string. This count is called R. For example, the binary string "101000" has r=3.
4. **Track Max (max\_R):** The algorithm maintains a single variable, max\_R, which stores the *maximum R value* it has seen so far across all elements in the stream.
5. **Estimate:** The final estimate for the number of distinct elements is simply 2^maxCount.

**Code4:**

data\_input=input("Enter Elements: ")

data=list(map(int, data\_input.split(" ")))

def hash\_function(x):

    return (6\*x+1)%5

maxCount=0

print("---Processing Streams---")

for i in data:

    print("Element: ",i,end="")

    x\_hash=hash\_function(i)

    print(", Hash: ",x\_hash,end="")

    x\_binary=bin(x\_hash)[2:]

    print(", Binary: ",x\_binary,end="")

    trailingZeroes=0

    j=len(x\_binary)-1

    while j>=0:

        if x\_binary[j]=='0':

            trailingZeroes+=1

        else:

            break

        j-=1

    maxCount=max(maxCount,trailingZeroes)

    print(", Trailing Zeroes: ",trailingZeroes)

print("--------------------------")

print("r: ", maxCount)

print("No of Distinct Elements: ", 2\*\*maxCount)

**Experiment 5: MongoDB (NoSQL) Queries**

**Aim:** To install MongoDB (a NoSQL database) and perform basic queries for creating, reading, updating, and deleting data10.

**Theory**

MongoDB is the most popular type of **NoSQL** ("non-relational") database11. Instead of storing data in tables and rows like a traditional relational database (RDBMS), it uses a **document-oriented model**12.

* Data is stored in **BSON** (Binary JSON), which is a flexible, JSON-like format13.
* The terminology is different 14:
  + RDBMS **Database** $\rightarrow$ MongoDB **Database**
  + RDBMS **Table** $\rightarrow$ MongoDB **Collection**
  + RDBMS **Row** $\rightarrow$ MongoDB **Document**
  + RDBMS **Column** $\rightarrow$ MongoDB **Field**

The basic queries follow the **CRUD** model:

* **Create:** insertOne() or insertMany() to add new documents.
* **Read:** find() or findOne() to retrieve documents. You can pass a filter to find specific documents.
* **Update:** updateOne() or updateMany() to modify existing documents. These use operators like $set (to change a value) or $inc (to increment a number).
* **Delete:** deleteOne() or deleteMany() to remove documents that match a filter.

**Code5:**

**// --- 0. Setup ---**

**// Switch to (or create) your database**

use myPracticeDB

**// Create your "collection" (table)**

db.createCollection("users")

**// --- 1. CREATE (Insert Data) ---**

**// Insert one document**

db.users.insertOne({

empId: 101,

name: "MegaKnight",

department: "HR",

salary: 45000

})

**// Insert many documents**

db.users.insertMany([

{ empId: 102, name: "Beserker", department: "IT", salary: 60000 },

{ empId: 103, name: "HogRider", department: "Finance", salary: 55000 }

])

**// --- 2. READ (Find Data) ---**

**// Find all documents in the collection**

db.users.find()

**// Find all users in the 'IT' department**

db.users.find({ department: "IT" })

**// --- 3. UPDATE (Change Data) ---**

**// Update one document**

**// Finds empId 102 and $sets their department to "Support"**

db.users.updateOne(

{ empId: 102 },

{ $set: { department: "Support" } }

)

**// Update many documents**

**// Finds ALL in 'Finance' and $incs (increments) their salary by 5000**

db.users.updateMany(

{ department: "Finance" },

{ $inc: { salary: 5000 } }

)

**// --- 4. DELETE (Remove Data) ---**

// Delete one document

db.users.deleteOne({ empId: 103 })

// Delete all documents matching a filter

db.users.deleteMany({ department: "HR" })

**// --- 5. CLEANUP ---**

// Delete the entire database you are currently using

db.dropDatabase()

**Experiment 6: DGIM Algorithm (Python)**

**Aim:** To study and implement the DGIM algorithm for estimating the count of **ones** in a data stream, specifically within a recent **sliding window**1.

**Theory**

The DGIM algorithm, like Flajolet-Martin, is a memory-efficient method for summarizing a data stream. Its goal is to answer queries like, "How many 1s have we seen in the last K bits?" .

It works by storing summaries of 1s in **"buckets"**:

1. **Buckets:** Each bucket has a **timestamp** (the position of its most recent 1) and a **size** (the number of 1s it represents, which must be a power of 2, like 1, 2, 4, 8, ...).
2. **The Rule:** To save space, the algorithm maintains a key invariant: for any given size (e.g., size 4), there can be **at most two buckets** of that size.
3. **Adding a 1:** When a new 1 arrives at the current timestamp:
   * A new bucket of size 1 is created with that timestamp.
   * The checkAndMergeBucket function is called.
4. **Merging:** If adding the new bucket breaks "The Rule" (e.g., we now have *three* buckets of size 1), the two *oldest* buckets of that size are merged into a *single* new bucket of the next size up (e.g., two size-1 buckets become one size-2 bucket). This merging process can cascade up to higher levels.
5. **Sliding Window:** As new bits arrive, old buckets are "aged out." Any bucket whose timestamp is older than the window size (e.g., older than $K$ bits) is removed 19191919.
6. **Estimating:** To answer a query, you sum the sizes of all buckets within the time window. The algorithm provides a bounded error by adding all bucket sizes, but **subtracting half the size of the single oldest bucket** that is (at least partially) in the window.

**Code6:**

import math

def checkAndMergeBucket(bucketList, t):

  bucketListLength = len(bucketList)

  for i in range(bucketListLength):

    while len(bucketList[i]) > 2:

      bucketList[i].pop(0)

      if i + 1 >= bucketListLength:

        bucketList[i].pop(0)

      else:

        bucketList[i + 1].append(bucketList[i].pop(0))

# --- Get Stream ---

bitstream = input("Enter a binary stream (e.g., 010110...): ").strip()

# --- BETTER LOGIC ---

# Set the window K and stream length N to the ACTUAL length of the input

# This makes the print statement logical.

K = len(bitstream)

N = K  # Set N to K

if N == 0: N = 1 # Avoid math.log(0)

# ---

k = int(math.floor(math.log(N, 2)))

t = 0

onesCount = 0

bucketList = [[] for i in range(k + 1)]

# Process the stream bit by bit

for c in bitstream:

  if c not in ('0', '1'):

    continue

  # Advance timestamp

  t = t + 1 # We'll just count up, no wraparound needed for this simple version

  # (No sliding window logic is needed since K = N)

  if c == '1':

    bucketList[0].append(t)

    checkAndMergeBucket(bucketList, t)

  elif c == '0':

    continue

# --- Estimation Phase (same as before) ---

print("\nFinal Buckets:")

earliestTimestamp = None

total\_size\_sum = 0

for i in range(k + 1):

  bucket\_size = 2\*\*i

  for ts in bucketList[i]:

    print(f"Size of bucket: {bucket\_size}, timestamp: {ts}")

    if earliestTimestamp is None or ts < earliestTimestamp:

      earliestTimestamp = ts

    total\_size\_sum += bucket\_size

oldest\_bucket\_size = 0

if earliestTimestamp is not None:

  for i in range(k + 1):

    if earliestTimestamp in bucketList[i]:

      oldest\_bucket\_size = 2\*\*i

      break

onesCount = total\_size\_sum - (0.5 \* oldest\_bucket\_size)

# --- BETTER PRINT STATEMENT ---

# Now K matches your stream length, so the line makes sense.

print(f"\nApproximate number of 1s in the last {K} bits: {int(onesCount)}")

**Experiment 7: Basic Queries in R**

**Aim:** To perform basic queries in R Programming.

**Theory:**

Querying is a way of extracting and manipulating data by "asking" specific questions. R provides functions to filter, group, and summarize data .

* **One-Time Query:** A query that is evaluated once over a point-in-time snapshot of the data. summary() is a good example, as it runs once and gives you a single statistical summary.
* **Ad-hoc Query:** A query issued "on the fly" to answer a specific question about the current data . subset() is an example, as it filters rows based on a condition you provide.
* **Continuous Query:** A query that is evaluated continuously as new data arrives. The experiment **simulates** this by running a query inside a for loop with a Sys.sleep() command to pretend it's "monitoring" the data over time .

**Code7:**

**# Load the 'dplyr' library for the select() function**

library(dplyr)

**# Load dataset**

student\_data <- read.csv("student\_performance.csv")

**# Remove an unnecessary column**

student\_data <- student\_data %>% select(-class\_participation)

**# Print the first 10 rows**

print("Student Performance Data (first 10 rows):")

print(head(student\_data, 10))

**# --- 1. One-Time Query ---**

# Gets a single statistical snapshot of the 'total\_score' column

cat("\n--- One-Time Query: Total Score Summary ---\n")

print(summary(student\_data$total\_score))

**# --- 2. Ad-hoc Query ---**

# A specific, one-time question: "Which students have < 85% attendance?"

cat("\n--- Adhoc Query: Students with Attendance < 85% ---\n")

adhoc\_result <- subset(student\_data, attendance\_percentage < 85)

print(head(adhoc\_result, 10))

**# --- 3. Continuous Query (Simulation) ---**

# Simulates "monitoring" the data for 3 iterations

cat("\n--- Continuous Query: Monitoring High Self-Study Hours (>20) ---\n")

for (i in 1:3) {

# The query is run

monitor\_result <- subset(student\_data, weekly\_self\_study\_hours > 20)

# The result is printed

cat(paste("\nIteration:", i, "\n"))

print(head(monitor\_result, 10))

# The system pauses for 5 seconds to simulate a live stream

Sys.sleep(5)

}

cat("\n--- Script Completed ---\n")

**Experiment 8: Regression in R**

**(Based on BDA exp8 scanned.pdf)**

**Aim:** To study and implement Linear, Multiple, and Logistic regression in R.

**Theory:**

Regression is a statistical technique used to model the relationship between a **dependent variable** (the outcome, Y) and one or more **independent variables** (the predictors, X).

1. **Simple Linear Regression:** Models the relationship between a *single* predictor (X) and a *continuous* outcome (Y) using a straight line46.
   * **Equation:** Y = beta\_{0} + beta\_{1}X + epsilon.
   * **R Function:** lm()
2. **Multiple Linear Regression:** Extends this to *multiple predictors* (X1, X2, ...), estimating their combined effect on a continuous outcome (Y)48.
   * **Equation:** $Y = beta\_{0} + beta\_{1}X\_{1} + beta\_{2}X\_{2} + ... + beta\_{n}X\_{n} + epsilon.
   * **R Function:** lm() (e.g., Y ~ X1 + X2)
3. **Logistic Regression:** Used when the outcome (Y) is **categorical** (e.g., "Yes/No", "setosa/versicolor"). It models the *probability* of an event using the logistic (sigmoid) function, which creates an S-shaped curve51.
   * **R Function:** glm(..., family = binomial)

**Code8:**

**1. Linear Regression**

data(iris)

model\_linear<-lm(Sepal.Length~Petal.Length, data=iris)

plot(iris$Petal.Length,iris$Sepal.Length,

main="Linear regression: Finding Sepal Length from Petal Length",

xlab="Petal Length",

ylab="Sepal Length",

)

abline(model\_linear,col="red")

**2. Multiple Regression (Plotting)**

library(ggplot2)

data(iris)

iris$Petal.Length.category<-cut(iris$Petal.Length,

breaks=3,

labels= c("Short","Medium","Long"))

my\_plot<-ggplot(iris,aes(x=Sepal.Width,y=Sepal.Length,color=Petal.Length.category))+

geom\_point()+

geom\_smooth(method="lm",se=FALSE)+

labs(title="Multiple Linear Regression (Grouped)",

x= "Sepal Width",

y="Sepal Length",

color="Petal Length Group")

print(my\_plot)

**3. Logistic Regression**

data(iris)

iris\_subset <- subset(iris, Species == "setosa" | Species == "versicolor")

iris\_subset$Species <- factor(iris\_subset$Species)

model\_logistic <- glm(Species ~ Sepal.Length,

data = iris\_subset,

family = "binomial")

x\_axis\_data <- data.frame(

Sepal.Length = seq(min(iris\_subset$Sepal.Length),

max(iris\_subset$Sepal.Length),

length.out = 100)

)

predicted\_probs <- predict(model\_logistic, x\_axis\_data, type = "response")

plot(x\_axis\_data$Sepal.Length, predicted\_probs,

type = "l", # 'l' stands for "line"

col = "blue",

lwd = 2,

main = "Logistic Regression: Species vs. Sepal Length",

xlab = "Sepal Length",

ylab = "Probability (1 = versicolor)")

actual\_points <- as.numeric(iris\_subset$Species) - 1

points(iris\_subset$Sepal.Length, actual\_points, pch = 19, col = "red")