**Code**

**Scenario:**Sensor Data Monitoring for IoT Devices Overview: Your testing scenario simulates an IoT-based environmental monitoring system. The system collects temperature and humidity readings from multiple sensors deployed in various locations at regular intervals. This dataset is suitable

## Create table:

|  |  |
| --- | --- |
| **Engines** | **Code** |
| PostgreSQL | CREATE TABLE sensor\_data (  time TIMESTAMP NOT NULL,  sensor\_id INT NOT NULL,  temperature DOUBLE PRECISION,  humidity DOUBLE PRECISION  ); |
| MongoDB | db.createCollection("sensor\_data", {  validator: {  $jsonSchema: {  bsonType: "object",  required: ["time", "sensor\_id", "temperature", "humidity"],  properties: {  time: { bsonType: "date" },  sensor\_id: { bsonType: "int" }  }  }  }  }); |
| InfluxDB | To create a table called sensor\_data in influxDB, only need to create a new bucket called sensor\_data by using InfluxDB Ul. |
| Redis | EVAL "  local key = KEYS[1]  local time = ARGV[1]  local sensor\_id = tonumber(ARGV[2])  local temperature = tonumber(ARGV[3])  local humidity = tonumber(ARGV[4])  if not time or not sensor\_id then  return 'Error: time and sensor\_id are required fields.'  end  redis.call('HMSET', key, 'time', time, 'sensor\_id', sensor\_id, 'temperature', temperature or 'null', 'humidity', humidity or 'null')  return 'Success: Data inserted into ' .. key  " 1 sensor:3 "2024-12-11T12:00:00Z" 103 24.0 50 |
| Apache Cassandra | CREATE KEYSPACE IF NOT EXISTS sensor\_data\_keyspace WITH replication = {'class': 'SimpleStrategy', 'replication\_factor': 1};  USE sensor\_data\_keyspace;  CREATE TABLE IF NOT EXISTS sensor\_data (  time TIMESTAMP,  sensor\_id INT,  temperature DOUBLE,  humidity DOUBLE,  PRIMARY KEY (time, sensor\_id)  ); |
| Apache IoTDB | SET STORAGE GROUP TO root.sensor\_data;  CREATE TIMESERIES root.sensor\_data.sensor\_id WITH DATATYPE=INT32, ENCODING=PLAIN;  CREATE TIMESERIES root.sensor\_data.temperature WITH DATATYPE=DOUBLE, ENCODING=RLE;  CREATE TIMESERIES root.sensor\_data.humidity WITH DATATYPE=DOUBLE, ENCODING=RLE; |

**\*When performing data creation, data manipulation and data retrival will be inserted 10,000 rows into the table before running the test**

|  |  |
| --- | --- |
| **Engines** | **Code** |
| **PostgreSQL** | SELECT COUNT(\*) FROM sensor\_data;  DELETE FROM sensor\_data;  INSERT INTO sensor\_data (time, sensor\_id, temperature, humidity)  SELECT  NOW(), -- Current timestamp  generate\_series(1, 10000), -- sensor\_id from 1 to 10,000  (RANDOM() \* 100), -- Random temperature between 0 and 100  (RANDOM() \* 100); -- Random humidity between 0 and 100 |
| **MongoDB** | db.sensor\_data.countDocuments();  db.sensor\_data.deleteMany({});  use test; // Make sure you are using the correct database  // Insert 10,000 rows into MongoDB  for (let i = 1; i <= 10000; i++) {  db.sensor\_data.insert({  time: new Date(), // Current timestamp  sensor\_id: i, // sensor\_id from 1 to 10,000  temperature: Math.random() \* 100, // Random temperature between 0 and 100  humidity: Math.random() \* 100 // Random humidity between 0 and 100  });  } |
| **InfluxDB** | from influxdb\_client import InfluxDBClient, Point  import random  from datetime import datetime, timedelta  # Connection details  url = "http://localhost:8086"  token = " N2gaqMKeBm5xVwqF8ve3Ov3ggzilX0FLRZETcpgS9OElaNiJGvWXvK8\_Td3ZrEoXgY1iqmQTpYeefTaZHP2WpA=="  org = "my-org"  bucket = "sensor\_data"  # Initialize the client  client = InfluxDBClient(url=url, token=token, org=org)  write\_api = client.write\_api()  # Generate and write 10,000 rows  start\_time = datetime.utcnow()  points = []  for i in range(1, 10001): # Generate 10,000 rows  timestamp = start\_time + timedelta(seconds=i) # Increment timestamp for each row  temperature = random.uniform(0, 100) # Random temperature between 0 and 100  humidity = random.uniform(0, 100) # Random humidity between 0 and 100    point = Point("sensor\_data") \  .tag("sensor\_id", i) \  .field("temperature", temperature) \  .field("humidity", humidity) \  .time(timestamp)  points.append(point)  # Write data in batch  write\_api.write(bucket=bucket, org=org, record=points)  print("10,000 rows of data written successfully!")  client.close() |
| **Redis** | EVAL "for \_, key in ipairs(redis.call('KEYS', 'sensor:\*')) do redis.call('DEL', key) end" 0  EVAL "  for i = 1, 10000 do  local key = 'sensor:' .. i  local time = redis.call('TIME')[1] -- Get current Unix timestamp  local temperature = math.random() \* 100 -- Random temperature between 0 and 100  local humidity = math.random() \* 100 -- Random humidity between 0 and 100  redis.call('HMSET', key, 'time', time, 'sensor\_id', i, 'temperature', temperature, 'humidity', humidity)  end  return 'Success: 10,000 sensor records inserted'  " 0 |
| **Apache Cassandra** | cqlsh:  TRUNCATE TABLE sensor\_data\_keyspace.sensor\_data;  External script:  const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace', // Replace with your actual keyspace name  pooling: {  maxRequestsPerConnection: 5000, // You can adjust this if needed  connectionsPerHost: {  'local': 10,  'remote': 5  }  }  });  // Function to generate random data for insertion  function generateRandomData(batchSize, startSensorId) {  const now = new Date();  const batch = [];  for (let i = 0; i < batchSize; i++) {  const timeValue = new Date(now.getTime() - ((i + 1) \* 1000)); // Time progression by 1 second  const sensorId = startSensorId + i; // Unique sensor\_id for each record (incremental)  const temperature = Math.random() \* 100; // Random temperature between 0 and 100  const humidity = Math.random() \* 100; // Random humidity between 0 and 100  batch.push([timeValue, sensorId, temperature, humidity]);  }  return batch;  }  // Function to insert data in batches  async function insertBatchData(batchSize, startSensorId) {  const query = 'INSERT INTO sensor\_data (time, sensor\_id, temperature, humidity) VALUES (?, ?, ?, ?)';  const batch = generateRandomData(batchSize, startSensorId);  // Prepare batch queries  const requests = batch.map(data =>  client.execute(query, data, { prepare: true })  );  // Execute batch insert  await Promise.all(requests);  }  // Function to insert data with batching and timing  async function executeInsert() {  const totalRecords = 10000; // Total number of records to insert  const batchSize = 500; // Batch size for each insertion  let startSensorId = 1; // Start with a unique sensor\_id for the first record  try {  console.log(`Inserting ${totalRecords} records in batches of ${batchSize}...`);  const start = Date.now(); // Record start time  for (let i = 0; i < totalRecords; i += batchSize) {  const currentBatchSize = Math.min(batchSize, totalRecords - i); // Handle remaining records  await insertBatchData(currentBatchSize, startSensorId); // Insert the current batch  startSensorId += currentBatchSize; // Update the starting sensor\_id for the next batch  }  const end = Date.now(); // Record end time  const elapsedTime = end - start; // Calculate the time taken in ms  console.log(`Inserted ${totalRecords} records in ${elapsedTime} ms.`);  } catch (error) {  console.error('Error inserting data:', error);  } finally {  client.shutdown(); // Gracefully shut down the client connection  }  }  // Run the insertion function  executeInsert(); |
| **Apache IoTDB** | IOTDB:  DELETE FROM root.sensor\_data.sensor\_id;  DELETE FROM root.sensor\_data.temperature;  DELETE FROM root.sensor\_data.humidity;  External script:  from iotdb.Session import Session  import random  from datetime import datetime  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def insert\_sensor\_data():  # Create a session and open a connection  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)  try:  # Number of records to insert  total\_records = 10000  batch\_size = 500 # Insert in batches of 500  for batch\_start in range(1, total\_records + 1, batch\_size):  # Insert data in batches  for sensor\_id in range(batch\_start, min(batch\_start + batch\_size, total\_records + 1)):  # Generate unique timestamp  timestamp = int((datetime.now().timestamp() + sensor\_id \* 0.001) \* 1000) # Adding a small offset to timestamp  temperature = random.uniform(0, 100) # Random temperature between 0 and 100  humidity = random.uniform(0, 100) # Random humidity between 0 and 100  # Construct the SQL insert statement  insert\_sql = (  f"INSERT INTO root.sensor\_data (time, sensor\_id, temperature, humidity) "  f"VALUES ({timestamp}, {sensor\_id}, {temperature:.2f}, {humidity:.2f})"  )  # Execute the SQL command  session.execute\_statement(insert\_sql)  # Commit after each batch  print(f"Inserted batch from {batch\_start} to {min(batch\_start + batch\_size - 1, total\_records)}")  print(f"Successfully inserted {total\_records} records.")  except Exception as e:  print(f"Error while inserting data: {e}")  finally:  # Close the session  session.close()  if \_\_name\_\_ == "\_\_main\_\_":  insert\_sensor\_data() |

## Data Creation:

|  |  |  |
| --- | --- | --- |
| **Engines** |  | **Code** |
| **PostgreSQL** | **Throughput (Ops/sec)** | INSERT INTO sensor\_data (time, sensor\_id, temperature, humidity)  SELECT NOW() - INTERVAL '1 second' \* generate\_series(1, 100000),  FLOOR(random() \* 10 + 1)::int,  random() \* 10 + 20,  random() \* 30 + 40; |
|  | **Insertion Latency** | DO $$  DECLARE  start\_time TIMESTAMP;  end\_time TIMESTAMP;  elapsed\_time DOUBLE PRECISION;  total\_time DOUBLE PRECISION := 0;  BEGIN  FOR i IN 1..5 LOOP  start\_time := clock\_timestamp();  FOR j IN 1..1000 LOOP  INSERT INTO sensor\_data (time, sensor\_id, temperature, humidity)  VALUES (NOW(), FLOOR(random() \* 10 + 1), random() \* 10 + 20, random() \* 30 + 40);  END LOOP;  end\_time := clock\_timestamp();  elapsed\_time := EXTRACT(EPOCH FROM (end\_time - start\_time)) \* 1000; -- in ms  total\_time := total\_time + elapsed\_time;  END LOOP;  RAISE NOTICE 'Average Insertion Time per Record (ms): %', total\_time / (5 \* 1000);  END $$; |
|  | **Batch Processing Capability** | DO $$  DECLARE  start\_time TIMESTAMP;  end\_time TIMESTAMP;  elapsed\_time DOUBLE PRECISION;  total\_time DOUBLE PRECISION := 0;  batch\_sizes INTEGER[] := ARRAY[1000, 10000, 50000]; -- Define the batch sizes array  batch\_size INTEGER;  iteration INTEGER; -- Variable for the outer loop  BEGIN  -- Loop through each batch size  FOREACH batch\_size IN ARRAY batch\_sizes LOOP  total\_time := 0;  FOR iteration IN 1..5 LOOP  start\_time := clock\_timestamp();  INSERT INTO sensor\_data (time, sensor\_id, temperature, humidity)  SELECT NOW() - INTERVAL '1 second' \* iteration, -- Use the loop variable here  FLOOR(random() \* 10 + 1),  random() \* 10 + 20,  random() \* 30 + 40  FROM generate\_series(1, batch\_size) AS gen\_series(id); -- Rename alias to avoid conflict  end\_time := clock\_timestamp();  elapsed\_time := EXTRACT(EPOCH FROM (end\_time - start\_time)) \* 1000; -- in ms  total\_time := total\_time + elapsed\_time;  END LOOP;  RAISE NOTICE 'Batch Size: %, Average Time (ms): %', batch\_size, total\_time / 5;  END LOOP;  END $$; |
| **MongoDB** | **Throughput (Ops/sec)** | let bulkData = [];  for (let i = 0; i < 100000; i++) {  bulkData.push({  time: new Date(new Date() - i \* 1000), // 1-second intervals  sensor\_id: Math.floor(Math.random() \* 10) + 1,  temperature: Math.random() \* 10 + 20,  humidity: Math.random() \* 30 + 40  });  }  let start = new Date();  db.sensor\_data.insertMany(bulkData);  let end = new Date();  print("Insert Time (ms):", end - start); |
|  | **Insertion Latency** | let total\_time = 0;  for (let run = 0; run < 5; run++) {  let start = new Date();  for (let i = 0; i < 1000; i++) {  db.sensor\_data.insertOne({  time: new Date(),  sensor\_id: Math.floor(Math.random() \* 10) + 1,  temperature: Math.random() \* 10 + 20,  humidity: Math.random() \* 30 + 40,  });  }  let end = new Date();  total\_time += (end - start);  }  print("Average Insertion Latency per Record (ms):", total\_time / (5 \* 1000)); |
|  | **Batch Processing Capability** | for (let batch\_size of [1000, 10000, 50000]) {  let total\_time = 0;  for (let run = 0; run < 5; run++) {  let bulkData = [];  for (let i = 0; i < batch\_size; i++) {  bulkData.push({  time: new Date(),  sensor\_id: Math.floor(Math.random() \* 10) + 1,  temperature: Math.random() \* 10 + 20,  humidity: Math.random() \* 30 + 40,  });  }  let start = new Date();  db.sensor\_data.insertMany(bulkData);  let end = new Date();  total\_time += (end - start);  }  print(`Batch Size: ${batch\_size}, Average Time (ms):`, total\_time / 5);  } |
| **InfluxDB** | **Throughput (Ops/sec)** | from influxdb\_client import InfluxDBClient, Point, WriteOptions  from datetime import datetime  import random  import time  # Connection details  url = "http://localhost:8086"  token = "N2gaqMKeBm5xVwqF8ve3Ov3ggzilX0FLRZETcpgS9OElaNiJGvWXvK8\_Td3ZrEoXgY1iqmQTpYeefTaZHP2WpA=="  org = "my-org"  bucket = "sensor\_data"  # Connect to InfluxDB  client = InfluxDBClient(url=url, token=token, org=org)  # Use WriteOptions for batch writing  write\_api = client.write\_api(write\_options=WriteOptions(batch\_size=500))  # Parameters for throughput measurement  total\_points = 0  write\_throughput\_runs = []  start\_time = time.time()  # Write data and measure throughput  for i in range(5000):  # Reduced total points to allow for 5 throughput logs      point = Point("sensor\_data") \          .tag("sensor\_id", str(random.randint(1, 10))) \          .field("temperature", random.uniform(20.0, 30.0)) \          .field("humidity", random.uniform(40.0, 60.0)) \          .time(datetime.utcnow())      write\_api.write(bucket=bucket, record=point)      total\_points += 1      # Log throughput every 1000 points (5 runs in total)      if total\_points % 1000 == 0:          elapsed\_time = time.time() - start\_time  # Time since start          throughput = total\_points / elapsed\_time  # Points per second          print(f"Write Throughput for Run {len(write\_throughput\_runs) + 1}: {throughput:.2f} points/sec")          write\_throughput\_runs.append(throughput)          start\_time = time.time()  # Reset start time for the next run          total\_points = 0  # Reset total points for the next run  # Close client  write\_api.close()  client.close()  # Calculate and print average write throughput  average\_throughput = sum(write\_throughput\_runs) / len(write\_throughput\_runs)  average\_throughput\_ms = (1 / average\_throughput) \* 1000  # Convert to milliseconds  print(f"Average Write Throughput: {average\_throughput:.2f} points/sec")  print(f"Average Write Throughput: {average\_throughput\_ms:.2f} ms") |
|  | **Insertion Latency** | from influxdb\_client import InfluxDBClient, Point, WriteOptions  from datetime import datetime  import time  # Connection details  url = "http://localhost:8086"  token = "N2gaqMKeBm5xVwqF8ve3Ov3ggzilX0FLRZETcpgS9OElaNiJGvWXvK8\_Td3ZrEoXgY1iqmQTpYeefTaZHP2WpA=="  org = "my-org"  bucket = "sensor\_data"  # Connect to InfluxDB  client = InfluxDBClient(url=url, token=token, org=org)  # Use WriteOptions for individual writes  write\_api = client.write\_api(write\_options=WriteOptions(batch\_size=1))  # Warm-up writes to initialize the database and avoid first-write latency effects  print("Warming up the database...")  for \_ in range(100):  # Perform 100 warm-up writes      point = Point("sensor\_data").tag("sensor\_id", "1").field("temperature", 25.0).time(datetime.utcnow())      write\_api.write(bucket=bucket, record=point)  print("Warm-up complete. Starting latency tests...\n")  # Parameters for latency measurement  runs = 5  # Number of test runs  num\_inserts = 1000  # Number of inserts per run  latencies = []  # Perform the tests  for run in range(1, runs + 1):      print(f"Run {run}:")      start\_time = time.time()      for \_ in range(num\_inserts):          point = Point("sensor\_data").tag("sensor\_id", "1").field("temperature", 25.0).time(datetime.utcnow())          write\_api.write(bucket=bucket, record=point)      end\_time = time.time()      latency = (end\_time - start\_time) \* 1000 / num\_inserts  # Average latency per record in milliseconds      latencies.append(latency)      print(f"Insertion Latency for Run {run}: {latency:.6f} ms\n")  # Calculate and display the average latency across all runs  average\_latency = sum(latencies) / len(latencies)  print(f"Average Insertion Latency (ms): {average\_latency:.6f}")  # Close the client  write\_api.close()  client.close() |
|  | **Batch Processing Capability** | from influxdb\_client import InfluxDBClient, Point, WriteOptions  from datetime import datetime  import time  # Connection details  url = "http://localhost:8086"  token = "N2gaqMKeBm5xVwqF8ve3Ov3ggzilX0FLRZETcpgS9OElaNiJGvWXvK8\_Td3ZrEoXgY1iqmQTpYeefTaZHP2WpA=="  org = "my-org"  bucket = "sensor\_data"  # Connect to InfluxDB  client = InfluxDBClient(url=url, token=token, org=org)  # Use WriteOptions for batch writing  write\_api = client.write\_api(write\_options=WriteOptions(batch\_size=500))  # Test configuration  batch\_sizes = [1000, 10000, 50000]  # Batch sizes to test  num\_runs = 5  # Number of iterations for each batch size  # Testing batch processing capability  print("Starting batch processing tests...\n")  for batch\_size in batch\_sizes:      total\_time = 0  # To accumulate total time for averaging      print(f"Testing Batch Size: {batch\_size}")      for run in range(1, num\_runs + 1):          # Prepare data          data = [Point("sensor\_data").tag("sensor\_id", "1").field("temperature", 25.0).time(datetime.utcnow())                  for \_ in range(batch\_size)]            # Measure time for writing the batch          start\_time = time.time()          write\_api.write(bucket=bucket, record=data)          end\_time = time.time()          batch\_time = (end\_time - start\_time) \* 1000  # Convert time to milliseconds            total\_time += batch\_time          print(f"Run {run}: Time (ms) = {batch\_time:.2f}")      # Calculate average time and time per record      average\_time = total\_time / num\_runs      time\_per\_record = average\_time / batch\_size      print(f"\nBatch Size: {batch\_size}")      print(f"Average Time (ms): {average\_time:.2f}")      print(f"Time per Record (ms): {time\_per\_record:.5f}\n")  # Close the client  write\_api.close()  client.close() |
| **Redis** | **Throughput (Ops/sec)** | EVAL "  for i = 1, 100000 do  local key = 'sensor:' .. i  local time = redis.call('TIME')[1] - i -- Generate timestamp decrementing by 1 second for each record  local sensor\_id = math.floor(math.random() \* 10 + 1) -- Random sensor ID between 1 and 10  local temperature = math.random() \* 10 + 20 -- Random temperature between 20 and 30  local humidity = math.random() \* 30 + 40 -- Random humidity between 40 and 70  redis.call('HMSET', key, 'time', time, 'sensor\_id', sensor\_id, 'temperature', temperature, 'humidity', humidity)  end  return 'Success: 100,000 sensor records inserted'  " 0 |
|  | **Insertion Latency** | EVAL "  local total\_time = 0  for i = 1, 5 do  local start\_time = os.clock() -- Start timing  for j = 1, 1000 do  local key = 'sensor:' .. j  local time = redis.call('TIME')[1] -- Current Unix timestamp  local sensor\_id = math.floor(math.random() \* 10 + 1) -- Random sensor ID between 1 and 10  local temperature = math.random() \* 10 + 20 -- Random temperature between 20 and 30  local humidity = math.random() \* 30 + 40 -- Random humidity between 40 and 70  redis.call('HMSET', key, 'time', time, 'sensor\_id', sensor\_id, 'temperature', temperature, 'humidity', humidity)  end  local end\_time = os.clock() -- End timing  local elapsed\_time = (end\_time - start\_time) \* 1000 -- Convert to ms  total\_time = total\_time + elapsed\_time  end  local average\_time\_per\_record = total\_time / (5 \* 1000)  return 'Average Insertion Time per Record (ms): ' .. average\_time\_per\_record  " 0 |
|  | **Batch Processing Capability** | EVAL "  local batch\_sizes = {1000, 10000, 50000}  local results = {}  local total\_time, iteration, batch\_size = 0, 0, 0  for \_, batch\_size in ipairs(batch\_sizes) do  total\_time = 0  for iteration = 1, 5 do  local start\_time = os.clock()  for j = 1, batch\_size do  local key = 'sensor:' .. j  local time = redis.call('TIME')[1] - iteration -- Simulate time with decrement  local sensor\_id = math.floor(math.random() \* 10 + 1)  local temperature = math.random() \* 10 + 20  local humidity = math.random() \* 30 + 40  redis.call('HMSET', key, 'time', time, 'sensor\_id', sensor\_id, 'temperature', temperature, 'humidity', humidity)  end  local end\_time = os.clock()  local elapsed\_time = (end\_time - start\_time) \* 1000  total\_time = total\_time + elapsed\_time  end  local average\_time\_per\_record = total\_time / (5 \* batch\_size)  table.insert(results, 'Batch Size: ' .. batch\_size .. ', Average Time (ms): ' .. average\_time\_per\_record)  end  return results  " 0 |
| **Apache Cassandra** | **Throughput (Ops/sec)** | const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace' // Replace with your actual keyspace name  });  async function insertData() {  const query = 'INSERT INTO sensor\_data (time, sensor\_id, temperature, humidity) VALUES (?, ?, ?, ?)';  const now = new Date();    for (let i = 0; i < 100000; i++) {  const timeValue = new Date(now.getTime() - (i \* 1000)); // Time progression by 1 second  const sensorId = Math.floor(Math.random() \* 10) + 1; // Random sensor\_id between 1 and 10  const temperature = Math.random() \* 10 + 20; // Random temperature between 20 and 30  const humidity = Math.random() \* 30 + 40; // Random humidity between 40 and 70    try {  await client.execute(query, [timeValue, sensorId, temperature, humidity], { prepare: true });  } catch (err) {  console.error('Error inserting data:', err);  }  }  console.log('Data insertion complete');  }  async function measureTime() {  const start = Date.now(); // Record start time in ms  await insertData();  const end = Date.now(); // Record end time in ms  console.log(`Time taken to insert 100000 rows: ${end - start} ms`); // Calculate time difference in ms  }  measureTime()  .then(() => client.shutdown())  .catch((err) => console.error('Error:', err)); |
|  | **Insertion Latency** | const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace' // Replace with your actual keyspace name  });  async function insertData() {  const query = 'INSERT INTO sensor\_data (time, sensor\_id, temperature, humidity) VALUES (?, ?, ?, ?)';  const now = new Date();    for (let i = 0; i < 1000; i++) {  const timeValue = new Date(now.getTime() - (i \* 1000)); // Time progression by 1 second  const sensorId = Math.floor(Math.random() \* 10) + 1; // Random sensor\_id between 1 and 10  const temperature = Math.random() \* 10 + 20; // Random temperature between 20 and 30  const humidity = Math.random() \* 30 + 40; // Random humidity between 40 and 70    await client.execute(query, [timeValue, sensorId, temperature, humidity], { prepare: true });  }  }  async function measureTime() {  let total\_time = 0;  const totalLoops = 5; // Number of iterations (like your outer loop)  const rowsPerLoop = 1000; // Number of rows per iteration (inner loop)  for (let i = 0; i < totalLoops; i++) {  const start = Date.now(); // Record start time in ms  await insertData();  const end = Date.now(); // Record end time in ms  const elapsed\_time = end - start; // Calculate the time taken for this iteration in ms  total\_time += elapsed\_time;  console.log(`Time taken for loop ${i + 1}: ${elapsed\_time} ms`);  }  const average\_time\_per\_record = total\_time / (totalLoops \* rowsPerLoop);  console.log(`Average Insertion Time per Record: ${average\_time\_per\_record.toFixed(4)} ms`);  }  measureTime()  .then(() => client.shutdown()) // Gracefully shutdown the client connection  .catch((err) => console.error('Error:', err)); |
|  | **Batch Processing Capability** | const cassandra = require('cassandra-driver');  // Configure the client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'],  localDataCenter: 'datacenter1',  keyspace: 'sensor\_data\_keyspace',  pooling: {  coreConnectionsPerHost: {  [cassandra.types.distance.local]: 4,  },  maxRequestsPerConnection: 4096,  },  socketOptions: {  readTimeout: 60000, // 60 seconds timeout  },  });  // Function to generate and insert batch data in chunks  async function insertBatchData(batchSize) {  const maxBatchSize = 500; // Maximum sub-batch size allowed  const totalSubBatches = Math.ceil(batchSize / maxBatchSize);  for (let subBatch = 0; subBatch < totalSubBatches; subBatch++) {  const queries = [];  const currentBatchSize = Math.min(maxBatchSize, batchSize - subBatch \* maxBatchSize);  for (let i = 0; i < currentBatchSize; i++) {  const time = new Date(Date.now() - (subBatch \* maxBatchSize + i) \* 1000).toISOString();  const sensorId = Math.floor(Math.random() \* 10 + 1);  const temperature = (Math.random() \* 10 + 20).toFixed(2);  const humidity = (Math.random() \* 30 + 40).toFixed(2);  queries.push({  query: 'INSERT INTO sensor\_data (time, sensor\_id, temperature, humidity) VALUES (?, ?, ?, ?)',  params: [time, sensorId, temperature, humidity],  });  }  // Execute the sub-batch  await client.batch(queries, { prepare: true });  }  }  // Main function to test batch processing  async function testBatchProcessing() {  const batchSizes = [1000, 10000, 50000];  for (const batchSize of batchSizes) {  let totalTime = 0;  console.log(`Testing with Batch Size: ${batchSize}`);  for (let iteration = 1; iteration <= 5; iteration++) {  const startTime = Date.now();  await insertBatchData(batchSize);  const endTime = Date.now();  const elapsedTime = endTime - startTime; // in ms  totalTime += elapsedTime;  console.log(`Iteration ${iteration} with Batch Size ${batchSize}: ${elapsedTime} ms`);  }  console.log(`Batch Size: ${batchSize}, Average Time: ${(totalTime / 5).toFixed(2)} ms`);  }  }  // Execute the test  testBatchProcessing()  .then(() => {  console.log('Batch processing completed successfully.');  return client.shutdown();  })  .catch(err => {  console.error('Error during batch processing:', err);  return client.shutdown();  }); |
| **Apache IoTDB** | **Throughput (Ops/sec)** | from iotdb.Session import Session  import random  from datetime import datetime, timedelta  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def insert\_sensor\_data():  # Create a session and open a connection  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)  try:  # Number of records to insert  total\_records = 100000  # Start timer  start\_time = datetime.now()  for i in range(1, total\_records + 1):  # Generate timestamp decreasing by 1 second for each record  timestamp = int((datetime.now() - timedelta(seconds=i)).timestamp() \* 1000)  sensor\_id = random.randint(1, 10) # Random sensor\_id between 1 and 10  temperature = random.uniform(20, 30) # Random temperature between 20 and 30  humidity = random.uniform(40, 70) # Random humidity between 40 and 70  # Construct the SQL insert statement  insert\_sql = (  f"INSERT INTO root.sensor\_data (time, sensor\_id, temperature, humidity) "  f"VALUES ({timestamp}, {sensor\_id}, {temperature:.2f}, {humidity:.2f})"  )  # Execute the SQL command  session.execute\_statement(insert\_sql)  # End timer  end\_time = datetime.now()  time\_taken = (end\_time - start\_time).total\_seconds() \* 1000 # Convert to milliseconds  print(f"Successfully inserted {total\_records} records in {time\_taken:.2f} ms.")  except Exception as e:  print(f"Error while inserting data: {e}")  finally:  # Close the session  session.close()  if \_\_name\_\_ == "\_\_main\_\_":  insert\_sensor\_data() |
|  | **Insertion Latency** | from iotdb.Session import Session  import random  from datetime import datetime  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def insert\_sensor\_data():  # Create a session and open a connection  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)  try:  total\_batches = 5 # Number of batches  records\_per\_batch = 1000 # Records per batch  total\_records = total\_batches \* records\_per\_batch  total\_time = 0 # Total time for all batches  for batch in range(total\_batches):  start\_time = datetime.now()  for \_ in range(records\_per\_batch):  timestamp = int(datetime.now().timestamp() \* 1000) # Current time in milliseconds  sensor\_id = random.randint(1, 10) # Random sensor\_id between 1 and 10  temperature = random.uniform(20, 30) # Random temperature between 20 and 30  humidity = random.uniform(40, 70) # Random humidity between 40 and 70  # Construct the SQL insert statement  insert\_sql = (  f"INSERT INTO root.sensor\_data (time, sensor\_id, temperature, humidity) "  f"VALUES ({timestamp}, {sensor\_id}, {temperature:.2f}, {humidity:.2f})"  )  # Execute the SQL command  session.execute\_statement(insert\_sql)  end\_time = datetime.now()  elapsed\_time = (end\_time - start\_time).total\_seconds() \* 1000 # Time in milliseconds  total\_time += elapsed\_time  print(f"Batch {batch + 1} completed in {elapsed\_time:.2f} ms.")  average\_time\_per\_record = total\_time / total\_records  print(f"Average Insertion Time per Record (ms): {average\_time\_per\_record:.2f}")  except Exception as e:  print(f"Error while inserting data: {e}")  finally:  # Close the session  session.close()  if \_\_name\_\_ == "\_\_main\_\_":  insert\_sensor\_data() |
|  | **Batch Processing Capability** | from iotdb.Session import Session  import random  from datetime import datetime, timedelta  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def insert\_sensor\_data():  # Create a session and open a connection  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)  try:  batch\_sizes = [1000, 10000, 50000] # Define the batch sizes  iterations = 5 # Number of iterations per batch size  for batch\_size in batch\_sizes:  total\_time = 0 # Reset total time for each batch size  for iteration in range(1, iterations + 1):  start\_time = datetime.now()  # Insert data for the current batch size  for i in range(batch\_size):  timestamp = int((datetime.now().timestamp() + (iteration \* 0.001) + (i \* 0.0001)) \* 1000) # Ensuring unique timestamps  sensor\_id = random.randint(1, 10) # Random sensor\_id between 1 and 10  temperature = random.uniform(20, 30) # Random temperature between 20 and 30  humidity = random.uniform(40, 70) # Random humidity between 40 and 70  # Construct the SQL insert statement  insert\_sql = (  f"INSERT INTO root.sensor\_data (time, sensor\_id, temperature, humidity) "  f"VALUES ({timestamp}, {sensor\_id}, {temperature:.2f}, {humidity:.2f})"  )  # Execute the SQL command  session.execute\_statement(insert\_sql)  end\_time = datetime.now()  elapsed\_time = (end\_time - start\_time).total\_seconds() \* 1000 # Time in milliseconds  total\_time += elapsed\_time  print(f"Batch Size: {batch\_size}, Iteration {iteration}, Time Taken: {elapsed\_time:.2f} ms")  average\_time = total\_time / iterations  print(f"Batch Size: {batch\_size}, Average Time (ms): {average\_time:.2f}")  except Exception as e:  print(f"Error while inserting data: {e}")  finally:  # Close the session  session.close()  if \_\_name\_\_ == "\_\_main\_\_":  insert\_sensor\_data() |

## Data Manipulation:

|  |  |  |  |
| --- | --- | --- | --- |
| **Engine** |  | **Operation Type** | **Code** |
| **PostgreSQL** | **Update Performance** | **Single Update** | UPDATE sensor\_data SET temperature = temperature + 1 WHERE sensor\_id = 1; |
|  |  | **Bulk Update** | UPDATE sensor\_data SET temperature = temperature + 1; |
|  | **Delete Operation Efficiency** | **Single Deletion** | DELETE FROM sensor\_data WHERE sensor\_id = 1; |
|  |  | **Bulk Deletion** | DELETE FROM sensor\_data WHERE sensor\_id <= 10; |
|  | **Data Modification Overhead** | **Single Update**  **Bulk Update**  **Single Deletion**  **Bulk Deletion** | Execution time  (PSQL terminal)  \timing -- Enable query timing  -- Single Update  UPDATE sensor\_data SET temperature = temperature + 1 WHERE sensor\_id = 1;  -- Bulk Update  UPDATE sensor\_data SET temperature = temperature + 1;  -- Single Delete  DELETE FROM sensor\_data WHERE sensor\_id = 1;  -- Bulk Delete  DELETE FROM sensor\_data WHERE sensor\_id <= 10; |
| **MongoDB** | **Update Performance** | **Single Update** | let start = new Date();  db.sensor\_data.updateOne({ sensor\_id: 1 }, { $inc: { temperature: 1 } });  let end = new Date();  print("Single Update Execution Time (ms):", end - start); |
|  |  | **Bulk Update** | let start = new Date();  db.sensor\_data.updateMany({}, { $inc: { temperature: 1 } });  let end = new Date();  print("Bulk Update Execution Time (ms):", end - start); |
|  | **Delete Operation Efficiency** | **Single Deletion** | let start = new Date();  db.sensor\_data.deleteOne({ sensor\_id: 1 });  let end = new Date();  print("Single Delete Execution Time (ms):", end - start); |
|  |  | **Bulk Deletion** | let start = new Date();  db.sensor\_data.deleteMany({ sensor\_id: { $lte: 10 } });  let end = new Date();  print("Bulk Delete Execution Time (ms):", end - start); |
|  | **Data Modification Overhead** | **Single Update,**  **Bulk Update,**  **Single Deletion,**  **Bulk Deletion** | Execution time  let start, end;  // Single Update  start = new Date();  db.sensor\_data.updateOne({ sensor\_id: 1 }, { $inc: { temperature: 1 } });  end = new Date();  print("Single Update Execution Time (ms):", end - start);  // Bulk Update  start = new Date();  db.sensor\_data.updateMany({}, { $inc: { temperature: 1 } });  end = new Date();  print("Bulk Update Execution Time (ms):", end - start);  // Single Delete  start = new Date();  db.sensor\_data.deleteOne({ sensor\_id: 1 });  end = new Date();  print("Single Delete Execution Time (ms):", end - start);  // Bulk Delete  start = new Date();  db.sensor\_data.deleteMany({ sensor\_id: { $lte: 10 } });  end = new Date();  print("Bulk Delete Execution Time (ms):", end - start); |
| **InfluxDB** | **Update Performance** | **Single Update,**  **Bulk Update** | from influxdb\_client import InfluxDBClient, Point, WriteOptions  from datetime import datetime  import random  import time  # Connection details  url = "http://localhost:8086"  token = "N2gaqMKeBm5xVwqF8ve3Ov3ggzilX0FLRZETcpgS9OElaNiJGvWXvK8\_Td3ZrEoXgY1iqmQTpYeefTaZHP2WpA=="  org = "my-org"  # Replace with your actual organization name  bucket = "sensor\_data"  # Replace with your actual bucket name  # Connect to InfluxDB  client = InfluxDBClient(url=url, token=token, org=org)  write\_api = client.write\_api(write\_options=WriteOptions(batch\_size=500))  # Function for Single Update Performance  def single\_update():      start\_time = time.time()      for i in range(1000):  # Write 1000 updates (one at a time)          write\_api.write(              bucket=bucket,              record=Point("sensor\_data").tag("sensor\_id", "1").field("temperature", 26.0).time(datetime.utcnow())          )      end\_time = time.time()      total\_time = end\_time - start\_time      return 1000 / total\_time  # Updates per second  # Function for Bulk Update Performance  def bulk\_update():      start\_time = time.time()      points = []      for i in range(1000):  # Create 1000 points          point = Point("sensor\_data").tag("sensor\_id", "1").field("temperature", 26.0).time(datetime.utcnow())          points.append(point)        # Write all 1000 points in one batch      write\_api.write(bucket=bucket, record=points)        end\_time = time.time()      total\_time = end\_time - start\_time      return 1000 / total\_time  # Updates per second  # Run single and bulk update tests  single\_update\_throughput = single\_update()  bulk\_update\_throughput = bulk\_update()  # Print the results  print(f"Single Update Throughput: {single\_update\_throughput:.2f} updates/sec")  print(f"Bulk Update Throughput: {bulk\_update\_throughput:.2f} updates/sec")  # Close client  write\_api.close()  client.close() |
|  | **Delete Operation Efficiency** | **Single Deletion,**  **Bulk Deletion** | from influxdb\_client import InfluxDBClient, Point, WriteOptions  from datetime import datetime  import random  import time  # Connection details  url = "http://localhost:8086"  token = "N2gaqMKeBm5xVwqF8ve3Ov3ggzilX0FLRZETcpgS9OElaNiJGvWXvK8\_Td3ZrEoXgY1iqmQTpYeefTaZHP2WpA=="  org = "my-org"  # Replace with your actual organization name  bucket = "sensor\_data"  # Replace with your actual bucket name  # Connect to InfluxDB  client = InfluxDBClient(url=url, token=token, org=org)  write\_api = client.write\_api(write\_options=WriteOptions(batch\_size=500))  # Function for Single Delete Performance  def single\_delete():      start\_time = time.time()      for i in range(1000):  # Write 1000 points (simulating deletes)          write\_api.write(              bucket=bucket,              record=Point("sensor\_data").tag("sensor\_id", "1").field("temperature", 26.0).time(datetime.utcnow())          )      end\_time = time.time()      total\_time = end\_time - start\_time      return 1000 / total\_time  # Deletes per second  # Function for Bulk Delete Performance  def bulk\_delete():      start\_time = time.time()      points = []      for i in range(1000):  # Create 1000 points to simulate deletes          point = Point("sensor\_data").tag("sensor\_id", "1").field("temperature", 26.0).time(datetime.utcnow())          points.append(point)        # Write all 1000 points in one batch      write\_api.write(bucket=bucket, record=points)        end\_time = time.time()      total\_time = end\_time - start\_time      return 1000 / total\_time  # Updates per second  # Run single and bulk delete tests  single\_delete\_throughput = single\_delete()  bulk\_delete\_throughput = bulk\_delete()  # Print the results  print(f"Single Delete Throughput: {single\_delete\_throughput:.2f} deletes/sec")  print(f"Bulk Delete Throughput: {bulk\_delete\_throughput:.2f} deletes/sec")  # Close client  write\_api.close()  client.close() |
|  | **Data Modification Overhead** |  | from influxdb\_client import InfluxDBClient, Point, WriteOptions  from datetime import datetime  import random  import time  # Connection details  url = "http://localhost:8086"  token = "N2gaqMKeBm5xVwqF8ve3Ov3ggzilX0FLRZETcpgS9OElaNiJGvWXvK8\_Td3ZrEoXgY1iqmQTpYeefTaZHP2WpA=="  # Replace with your actual token  org = "my-org"  # Replace with your organization name  bucket = "sensor\_data"  # Replace with your bucket name  # Connect to InfluxDB  client = InfluxDBClient(url=url, token=token, org=org)  write\_api = client.write\_api(write\_options=WriteOptions(batch\_size=500))  # Function to simulate multiple updates and deletes in quick succession and measure time taken (overhead)  def data\_modification\_overhead():      start\_time = time.time()      # Simulating multiple updates and deletes in quick succession (e.g., 1000 operations)      for i in range(1000):          # Simulating an update (writing new data to represent update)          write\_api.write(              bucket=bucket,              record=Point("sensor\_data")  # Using "sensor\_data" measurement              .tag("sensor\_id", str(random.randint(1, 10)))  # Random sensor\_id for diversity              .field("temperature", random.uniform(20.0, 30.0))  # Random temperature value              .field("humidity", random.uniform(40.0, 60.0))  # Random humidity value              .time(datetime.utcnow())  # Timestamp for the data          )            # Simulating a delete (writing new data with modified values to represent a delete)          write\_api.write(              bucket=bucket,              record=Point("sensor\_data")              .tag("sensor\_id", str(random.randint(1, 10)))  # Random sensor\_id              .field("temperature", random.uniform(20.0, 30.0))  # Random temperature value              .field("humidity", random.uniform(40.0, 60.0))  # Random humidity value              .time(datetime.utcnow())          )      end\_time = time.time()      total\_time = end\_time - start\_time      total\_time\_ms = total\_time \* 1000  # Convert time to milliseconds      return total\_time\_ms  # Return total time in milliseconds  # Run the data modification overhead test  modification\_overhead\_time\_ms = data\_modification\_overhead()  # Print the result in milliseconds  print(f"Data Modification Overhead Time for 1000 operations (simulating updates and deletes): {modification\_overhead\_time\_ms:.2f} ms")  # Close client  write\_api.close()  client.close() |
| **Redis** | **Update Performance** | **Single Update** | EVAL "  local key = 'sensor:1'  local temperature = redis.call('HGET', key, 'temperature')  if temperature then  temperature = tonumber(temperature) -- Convert to a number  if temperature then  redis.call('HSET', key, 'temperature', temperature + 1) -- Add 1 and set the new value  return 'Temperature updated successfully'  else  return 'Temperature value is not a valid number'  end  else  return 'Temperature field does not exist'  end  " 0 |
|  |  | **Bulk Update** | EVAL "  local keys = redis.call('KEYS', 'sensor:\*') -- Get all keys matching the pattern 'sensor:\*'  for i, key in ipairs(keys) do  local temperature = redis.call('HGET', key, 'temperature')  if temperature then  temperature = tonumber(temperature) -- Convert to a number  if temperature then  redis.call('HSET', key, 'temperature', temperature + 1) -- Increment the temperature by 1  end  end  end  return 'Bulk update completed'  " 0 |
|  | **Delete Operation Efficiency** | **Single Deletion** | EVAL "  local keys = redis.call('KEYS', 'sensor:\*') -- Get all keys matching the pattern 'sensor:\*'  for \_, key in ipairs(keys) do  local sensor\_id = redis.call('HGET', key, 'sensor\_id')  if sensor\_id and tonumber(sensor\_id) == 1 then -- Check if sensor\_id equals 1  redis.call('DEL', key) -- Delete the key  end  end  return 'Deletion completed for sensor\_id = 1'  " 0 |
|  |  | **Bulk Deletion** | EVAL "  local keys = redis.call('KEYS', 'sensor:\*') -- Get all keys matching the pattern 'sensor:\*'  for \_, key in ipairs(keys) do  local sensor\_id = redis.call('HGET', key, 'sensor\_id')  if sensor\_id and tonumber(sensor\_id) <= 10 then -- Check if sensor\_id is <= 10  redis.call('DEL', key) -- Delete the key  end  end  return 'Deletion completed for sensor\_id <= 10'  " 0 |
|  | **Data Modification Overhead** | **Single Update**  **Bulk Update**  **Single Deletion**  **Bulk Deletion** | EVAL "  local start, keys, single\_update\_time, bulk\_update\_time, single\_delete\_time, bulk\_delete\_time, log\_output;  log\_output = {};  -- Measure single update time  start = os.clock();  local key = 'sensor:1';  local temperature = redis.call('HGET', key, 'temperature');  if temperature then  redis.call('HSET', key, 'temperature', tonumber(temperature) + 1);  end  single\_update\_time = (os.clock() - start) \* 1000;  table.insert(log\_output, string.format('Single Update Execution Time (ms): %.6f', single\_update\_time));  -- Measure bulk update time  start = os.clock();  keys = redis.call('KEYS', 'sensor:\*');  for \_, key in ipairs(keys) do  local temperature = redis.call('HGET', key, 'temperature');  if temperature then  redis.call('HSET', key, 'temperature', tonumber(temperature) + 1);  end  end  bulk\_update\_time = (os.clock() - start) \* 1000;  table.insert(log\_output, string.format('Bulk Update Execution Time (ms): %.6f', bulk\_update\_time));  -- Measure single delete time  start = os.clock();  redis.call('DEL', 'sensor:1');  single\_delete\_time = (os.clock() - start) \* 1000;  table.insert(log\_output, string.format('Single Delete Execution Time (ms): %.6f', single\_delete\_time));  -- Measure bulk delete time  start = os.clock();  keys = redis.call('KEYS', 'sensor:\*');  for \_, key in ipairs(keys) do  local sensor\_id = redis.call('HGET', key, 'sensor\_id');  if sensor\_id and tonumber(sensor\_id) <= 10 then  redis.call('DEL', key);  end  end  bulk\_delete\_time = (os.clock() - start) \* 1000;  table.insert(log\_output, string.format('Bulk Delete Execution Time (ms): %.6f', bulk\_delete\_time));  return log\_output;  " 0 |
| **Apache Cassandra** | **Update Performance** | **Single Update** | const cassandra = require('cassandra-driver');  // Configure the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'],  localDataCenter: 'datacenter1',  keyspace: 'sensor\_data\_keyspace',  });  async function updateTemperature(sensorId, time, newTemperature) {  try {  const updateQuery = 'UPDATE sensor\_data SET temperature = ? WHERE sensor\_id = ? AND time = ?';  const startTime = Date.now();  await client.execute(updateQuery, [newTemperature, sensorId, time], { prepare: true });  const endTime = Date.now();  console.log(`Temperature updated to ${newTemperature} for sensor\_id: ${sensorId} at time: ${time}`);  console.log(`Time taken for the update: ${endTime - startTime} ms`);  } catch (err) {  console.error('Error updating temperature:', err);  } finally {  await client.shutdown();  }  }  (async () => {  const sensorId = 1;  const time = '2024-12-12T12:00:00Z'; // Replace with your actual timestamp  const newTemperature = 25.5;  await updateTemperature(sensorId, time, newTemperature);  })(); |
|  |  | **Bulk Update** | const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace' // Replace with your keyspace name  });  async function getAllRecords() {  const query = 'SELECT time, sensor\_id, temperature FROM sensor\_data';  try {  const result = await client.execute(query);  return result.rows;  } catch (err) {  console.error('Error fetching records:', err);  return [];  }  }  async function updateTemperature(sensorData) {  const query = 'UPDATE sensor\_data SET temperature = ? WHERE time = ? AND sensor\_id = ?';  try {  for (let row of sensorData) {  const { time, sensor\_id, temperature } = row;  const updatedTemperature = temperature + 1; // Add 1 to the current temperature  await client.execute(query, [updatedTemperature, time, sensor\_id], { prepare: true });  }  } catch (err) {  console.error('Error updating records:', err);  }  }  async function updateAllTemperatures() {  const sensorData = await getAllRecords();    if (sensorData.length === 0) {  console.log('No data found to update.');  return;  }  console.log(`Updating temperature for ${sensorData.length} sensor records...`);    const startTime = Date.now();  await updateTemperature(sensorData);  const endTime = Date.now();  const elapsedTime = endTime - startTime;  console.log(`Updated ${sensorData.length} records in ${elapsedTime} ms.`);    client.shutdown();  }  // Run the update process  updateAllTemperatures(); |
|  | **Delete Operation Efficiency** | **Single Deletion** | const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace' // Replace with your keyspace name  });  async function deleteRecord(sensorId, time) {  const query = 'DELETE FROM sensor\_data WHERE sensor\_id = ? AND time = ?';  try {  const startTime = Date.now(); // Record start time  await client.execute(query, [sensorId, time], { prepare: true }); // Execute the delete query  const endTime = Date.now(); // Record end time  const elapsedTime = endTime - startTime; // Calculate time taken in ms  console.log(`Deleted record with sensor\_id = ${sensorId} and time = ${time} in ${elapsedTime} ms.`);  } catch (error) {  console.error('Error deleting record:', error);  } finally {  client.shutdown(); // Gracefully shut down the client connection  }  }  // Example: Deleting a record for sensor\_id = 1 and a specific time  const sensorId = 1;  const time = '2024-12-12 10:00:00'; // Replace with an actual timestamp from your data  deleteRecord(sensorId, time); |
|  |  | **Bulk Deletion** | const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace' // Replace with your keyspace name  });  // Function to delete records where sensor\_id <= 10  async function deleteSensorRecords(sensorIdLimit) {  try {  const startTime = Date.now(); // Record start time  // Iterate through each sensor\_id from 1 to sensorIdLimit (10 in this case)  for (let sensorId = 1; sensorId <= sensorIdLimit; sensorId++) {  // Query for all records for the current sensor\_id with ALLOW FILTERING  const selectQuery = 'SELECT time FROM sensor\_data WHERE sensor\_id = ? ALLOW FILTERING';  const result = await client.execute(selectQuery, [sensorId], { prepare: true });  // If there are records for the current sensor\_id  if (result.rows.length > 0) {  // Delete each record by time for the given sensor\_id  for (const row of result.rows) {  const { time } = row;  const deleteQuery = 'DELETE FROM sensor\_data WHERE sensor\_id = ? AND time = ?';  await client.execute(deleteQuery, [sensorId, time], { prepare: true });  console.log(`Deleted record for sensor\_id = ${sensorId} at time = ${time}`);  }  } else {  console.log(`No records found for sensor\_id = ${sensorId}`);  }  }  const endTime = Date.now(); // Record end time  const elapsedTime = endTime - startTime; // Calculate time taken in ms  console.log(`Completed deletion of records for sensor\_id <= ${sensorIdLimit} in ${elapsedTime} ms.`);  } catch (error) {  console.error('Error deleting records:', error);  } finally {  client.shutdown(); // Gracefully shut down the client connection  }  }  // Example: Delete records for sensor\_id <= 10  deleteSensorRecords(10); |
|  | **Data Modification Overhead** | **Single Update**  **Bulk Update**  **Single Deletion**  **Bulk Deletion** | const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace' // Replace with your keyspace name  });  // Update the temperature for a given sensor\_id and time  async function updateTemperature(sensorId, time, newTemperature) {  try {  const updateQuery = 'UPDATE sensor\_data SET temperature = ? WHERE sensor\_id = ? AND time = ?';  const startTime = Date.now();  await client.execute(updateQuery, [newTemperature, sensorId, time], { prepare: true });  const endTime = Date.now();  console.log(`Temperature updated to ${newTemperature} for sensor\_id: ${sensorId} at time: ${time}`);  console.log(`Time taken for the update: ${endTime - startTime} ms`);  } catch (err) {  console.error('Error updating temperature:', err);  }  }  // Fetch all records to be updated  async function getAllRecords() {  const query = 'SELECT time, sensor\_id, temperature FROM sensor\_data';  try {  const result = await client.execute(query);  return result.rows;  } catch (err) {  console.error('Error fetching records:', err);  return [];  }  }  // Update all temperatures by adding 1 to the current value  async function updateAllTemperatures() {  const sensorData = await getAllRecords();    if (sensorData.length === 0) {  console.log('No data found to update.');  return;  }  console.log(`Updating temperature for ${sensorData.length} sensor records...`);    const startTime = Date.now();  for (let row of sensorData) {  const { time, sensor\_id, temperature } = row;  const updatedTemperature = temperature + 1; // Add 1 to the current temperature  await client.execute('UPDATE sensor\_data SET temperature = ? WHERE time = ? AND sensor\_id = ?',  [updatedTemperature, time, sensor\_id], { prepare: true });  }  const endTime = Date.now();  const elapsedTime = endTime - startTime;  console.log(`Updated ${sensorData.length} records in ${elapsedTime} ms.`);  }  // Delete a record by sensor\_id and time  async function deleteRecord(sensorId, time) {  const query = 'DELETE FROM sensor\_data WHERE sensor\_id = ? AND time = ?';  try {  const startTime = Date.now();  await client.execute(query, [sensorId, time], { prepare: true });  const endTime = Date.now();  console.log(`Deleted record with sensor\_id = ${sensorId} and time = ${time} in ${endTime - startTime} ms.`);  } catch (err) {  console.error('Error deleting record:', err);  }  }  // Delete records for sensor\_id <= sensorIdLimit  async function deleteSensorRecords(sensorIdLimit) {  try {  const startTime = Date.now();    for (let sensorId = 1; sensorId <= sensorIdLimit; sensorId++) {  const selectQuery = 'SELECT time FROM sensor\_data WHERE sensor\_id = ? ALLOW FILTERING';  const result = await client.execute(selectQuery, [sensorId], { prepare: true });  if (result.rows.length > 0) {  for (const row of result.rows) {  const { time } = row;  const deleteQuery = 'DELETE FROM sensor\_data WHERE sensor\_id = ? AND time = ?';  await client.execute(deleteQuery, [sensorId, time], { prepare: true });  console.log(`Deleted record for sensor\_id = ${sensorId} at time = ${time}`);  }  } else {  console.log(`No records found for sensor\_id = ${sensorId}`);  }  }  const endTime = Date.now();  console.log(`Completed deletion of records for sensor\_id <= ${sensorIdLimit} in ${endTime - startTime} ms.`);  } catch (error) {  console.error('Error deleting records:', error);  }  }  // Execute the test: Update and delete operations  async function runTest() {  // Update data (change temperature for a specific record)  const sensorIdToUpdate = 1;  const timeToUpdate = '2024-12-12T12:00:00Z'; // Replace with an actual timestamp from your data  const newTemperature = 25.5;  await updateTemperature(sensorIdToUpdate, timeToUpdate, newTemperature);  // Update all temperatures  await updateAllTemperatures();  // Delete a specific record  const sensorIdToDelete = 1;  const timeToDelete = '2024-12-12T10:00:00Z'; // Replace with an actual timestamp from your data  await deleteRecord(sensorIdToDelete, timeToDelete);  // Delete sensor records for sensor\_id <= 10  await deleteSensorRecords(10);  // Shutdown the client after testing  client.shutdown();  }  // Run the test  runTest(); |
| **Apache IoTDB** | **Update Performance** | **Single Update** | IoTDB (Apache IoT Database) **does not support the traditional SQL UPDATE statement** for modifying data in place. This is because IoTDB is designed primarily for time-series data, which is optimized for append-only writes (i.e., adding new data). The typical pattern for time-series data in IoTDB is to insert new data rather than modify existing records. |
|  |  | **Bulk Update** | IoTDB (Apache IoT Database) **does not support the traditional SQL UPDATE statement** for modifying data in place. This is because IoTDB is designed primarily for time-series data, which is optimized for append-only writes (i.e., adding new data). The typical pattern for time-series data in IoTDB is to insert new data rather than modify existing records. |
|  | **Delete Operation Efficiency** | **Single Deletion** | from iotdb.Session import Session  from datetime import datetime  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def delete\_single\_record(time\_to\_delete):  # Create a session and open a connection  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)  try:  # Start timing the delete operation  start\_time = datetime.now()  # Construct the DELETE SQL query  delete\_sql = f"DELETE FROM root.sensor\_data WHERE time = {time\_to\_delete}"  # Execute the DELETE statement  session.execute\_statement(delete\_sql)  # End timing the delete operation  end\_time = datetime.now()  elapsed\_time = (end\_time - start\_time).total\_seconds() \* 1000 # Time in milliseconds  print(f"Record at time {time\_to\_delete} deleted. Time taken: {elapsed\_time:.2f} ms")  except Exception as e:  print(f"Error while deleting data: {e}")  finally:  session.close()  if \_\_name\_\_ == "\_\_main\_\_":  time\_to\_delete = 1734190737524 # Replace with the time of the record you want to delete (in milliseconds)  delete\_single\_record(time\_to\_delete) |
|  |  | **Bulk Deletion** | from iotdb.Session import Session  from datetime import datetime  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def delete\_multiple\_records(time\_values):  # Create a session and open a connection  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)  try:  # Start timing the delete operation  start\_time = datetime.now()  # Loop through each time and execute the delete statement  for time\_to\_delete in time\_values:  delete\_sql = f"DELETE FROM root.sensor\_data WHERE time = {time\_to\_delete}"  session.execute\_statement(delete\_sql)  # End timing the delete operation  end\_time = datetime.now()  elapsed\_time = (end\_time - start\_time).total\_seconds() \* 1000 # Time in milliseconds  print(f"Deleted {len(time\_values)} records. Time taken: {elapsed\_time:.2f} ms")  except Exception as e:  print(f"Error while deleting data: {e}")  finally:  session.close()  if \_\_name\_\_ == "\_\_main\_\_":  # Example: List of times (in milliseconds) of records to delete  # Replace these with actual time values (timestamps in ms)  time\_values\_to\_delete = [  1734190737513, 1734190737514, 1734190737515,  1734190737516, 1734190737517, 1734190737518,  1734190737519, 1734190737520, 1734190737521,  1734190737522  ]    delete\_multiple\_records(time\_values\_to\_delete) |
|  | **Data Modification Overhead** | **Single Update**  **Bulk Update**  **Single Deletion**  **Bulk Deletion** | from iotdb.Session import Session  from datetime import datetime  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def delete\_single\_record(time\_to\_delete):  """Deletes a single record based on a specific time value."""  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)    try:  # Start timing the single delete operation  start\_time = datetime.now()  delete\_sql = f"DELETE FROM root.sensor\_data WHERE time = {time\_to\_delete}"  session.execute\_statement(delete\_sql)    # End timing the delete operation  end\_time = datetime.now()  elapsed\_time = (end\_time - start\_time).total\_seconds() \* 1000 # Time in milliseconds  print(f"Single Deletion Time: {elapsed\_time:.2f} ms")  return elapsed\_time  except Exception as e:  print(f"Error while deleting single record: {e}")  finally:  session.close()  def delete\_bulk\_records(time\_values):  """Deletes multiple records based on a list of time values."""  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)    try:  # Start timing the bulk delete operation  start\_time = datetime.now()    for time\_to\_delete in time\_values:  delete\_sql = f"DELETE FROM root.sensor\_data WHERE time = {time\_to\_delete}"  session.execute\_statement(delete\_sql)  # End timing the delete operation  end\_time = datetime.now()  elapsed\_time = (end\_time - start\_time).total\_seconds() \* 1000 # Time in milliseconds  print(f"Bulk Deletion Time: {elapsed\_time:.2f} ms")  return elapsed\_time  except Exception as e:  print(f"Error while deleting bulk records: {e}")  finally:  session.close()  def main():  # Example time values for deletion (replace with actual timestamps)  single\_time\_value = 1734190737485 # Example timestamp for single deletion (in ms)  bulk\_time\_values = [  1734190737486, 1734190737487, 1734190737488,  1734190737489, 1734190737490, 1734190737491,  1734190737492, 1734190737493, 1734190737494, 1734190737495  ] # Example timestamps for bulk deletion (in ms)  # Perform single and bulk deletions, and output the time taken for each  single\_deletion\_time = delete\_single\_record(single\_time\_value)  bulk\_deletion\_time = delete\_bulk\_records(bulk\_time\_values)  # Output total times taken  print(f"Total Time for Single Deletion: {single\_deletion\_time:.2f} ms")  print(f"Total Time for Bulk Deletion: {bulk\_deletion\_time:.2f} ms")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

## Data Retrieval:

|  |  |
| --- | --- |
| **PostgreSQL** | **Code** |
| **Query Response Time** | SELECT \* FROM sensor\_data; |
| **Complex Query Performance** | SELECT sensor\_id, AVG(temperature) AS avg\_temp FROM sensor\_data GROUP BY sensor\_id; |
| **Time-Range Query Efficiency** | SELECT \* FROM sensor\_data  WHERE time BETWEEN '2024-12-01 00:00:00' AND '2024-12-02 00:00:00'; |

|  |  |
| --- | --- |
| **MongoDB** | **Code** |
| Query Response Time | let start = new Date();  db.sensor\_data.find({}).toArray();  let end = new Date();  print("Query Time (ms):", end - start); |
| Complex Query Performance | let start = new Date();  db.sensor\_data.aggregate([  { $group: { \_id: "$sensor\_id", avg\_temp: { $avg: "$temperature" } } }  ]);  let end = new Date();  print("Aggregation Query Time (ms):", end - start); |
| Time-Range Query Efficiency | let start = new Date();  db.sensor\_data.find({  time: { $gte: ISODate("2024-12-01T00:00:00Z"), $lt: ISODate("2024-12-02T00:00:00Z") }  }).toArray();  let end = new Date();  print("Time-Range Query Time (ms):", end - start); |

|  |  |
| --- | --- |
| **InfluxDB** | **Code** |
| Query Response Time  Complex Query Performance  Time-Range Query Efficiency | from influxdb\_client import InfluxDBClient  import time  # Connection details  url = "http://localhost:8086"  token = "N2gaqMKeBm5xVwqF8ve3Ov3ggzilX0FLRZETcpgS9OElaNiJGvWXvK8\_Td3ZrEoXgY1iqmQTpYeefTaZHP2WpA=="  # Replace with your token  org = "my-org"  # Replace with your organization  bucket = "sensor\_data"  # Replace with your bucket name  # Connect to InfluxDB  client = InfluxDBClient(url=url, token=token, org=org)  query\_api = client.query\_api()  # Function to measure query execution time with error handling  def measure\_query\_time(query, query\_type, runs=5):      print(f"Running {query\_type}...")      times = []      for i in range(1, runs + 1):          try:              start\_time = time.time()              query\_api.query(org=org, query=query)              end\_time = time.time()              exec\_time = (end\_time - start\_time) \* 1000  # Convert to ms              times.append(exec\_time)              print(f"Run {i}: {exec\_time:.2f} ms")          except Exception as e:              print(f"Error during {query\_type} Run {i}: {e}")              times.append(float('inf'))  # Assign a large value for failed runs      average\_time = sum(times) / runs      print(f"Average Time for {query\_type}: {average\_time:.2f} ms\n")      return average\_time  # Queries  fetch\_all\_query = f'''  from(bucket:"{bucket}")    |> range(start:-1d)  '''  complex\_query = f'''  from(bucket:"{bucket}")    |> range(start:-1d)    |> filter(fn: (r) => r["\_field"] == "temperature")    |> group(columns: ["sensor\_id"])    |> mean(column: "\_value")  '''  time\_range\_query = f'''  from(bucket:"{bucket}")    |> range(start:2024-12-01T00:00:00Z, stop:2024-12-02T00:00:00Z)  '''  # Measure execution times for each query type  fetch\_all\_avg\_time = measure\_query\_time(fetch\_all\_query, "Query Response Time (Fetch All)")  complex\_query\_avg\_time = measure\_query\_time(complex\_query, "Complex Query Performance")  time\_range\_query\_avg\_time = measure\_query\_time(time\_range\_query, "Time-Range Query Efficiency")  # Final Results  print("Final Results:")  print(f"Query Response Time (Fetch All): {fetch\_all\_avg\_time:.2f} ms")  print(f"Complex Query Performance: {complex\_query\_avg\_time:.2f} ms")  print(f"Time-Range Query Efficiency: {time\_range\_query\_avg\_time:.2f} ms")  # Close the client  client.close() |

|  |  |
| --- | --- |
| **Redis** | **Code** |
| **Query Response Time** | EVAL "  local start = os.clock(); -- Record the start time  local keys = redis.call('KEYS', 'sensor:\*'); -- Retrieve all keys matching the pattern 'sensor:\*'  local end\_time = os.clock(); -- Record the end time  -- Calculate the query time in milliseconds  local query\_time = (end\_time - start) \* 1000;  -- Log the query time with millisecond precision  redis.log(1, string.format('Query Time (ms): %.6f', query\_time));  return keys; -- Return the keys  " 0 |
| **Complex Query Performance** | EVAL "  local start = redis.call('TIME')[1] + redis.call('TIME')[2] / 1000000; -- Start time measurement  local keys = redis.call('KEYS', 'sensor:\*'); -- Get all keys matching the pattern 'sensor:\*'  local sensor\_aggregates = {}; -- Table to store aggregated sensor data  -- Iterate over each key to collect temperature data and count  for i, key in ipairs(keys) do  local sensor\_id = redis.call('HGET', key, 'sensor\_id');  local temperature = redis.call('HGET', key, 'temperature');  if sensor\_id and temperature then  sensor\_id = tonumber(sensor\_id);  temperature = tonumber(temperature);  if sensor\_aggregates[sensor\_id] == nil then  sensor\_aggregates[sensor\_id] = {sum = 0, count = 0};  end  sensor\_aggregates[sensor\_id].sum = sensor\_aggregates[sensor\_id].sum + temperature;  sensor\_aggregates[sensor\_id].count = sensor\_aggregates[sensor\_id].count + 1;  end  end  -- Calculate the average temperature for each sensor\_id  local averages = {};  for sensor\_id, values in pairs(sensor\_aggregates) do  if values.count > 0 then  local avg\_temp = values.sum / values.count;  table.insert(averages, {sensor\_id = sensor\_id, avg\_temp = avg\_temp});  end  end  -- End time measurement and log the execution time  local end\_time = redis.call('TIME')[1] + redis.call('TIME')[2] / 1000000;  local query\_time = (end\_time - start) \* 1000;  redis.log(1, string.format('Aggregation Query Time (ms): %.6f', query\_time));  -- Return the list of average temperatures  return averages;  " 0 |
| **Time-Range Query Efficiency** | -- Start time measurement  local start = redis.call('TIME')[1] + redis.call('TIME')[2] / 1000000 -- Current time in seconds with microsecond precision  -- Get all keys matching the pattern 'sensor:\*'  local keys = redis.call('KEYS', 'sensor:\*')  -- Define the time range for filtering  local start\_time = 1669852800 -- Example: Unix timestamp for "2024-12-12T00:00:00Z"  local end\_time = 1669939200 -- Example: Unix timestamp for "2024-12-13T00:00:00Z"  -- Table to store results that match the time range  local results = {}  -- Iterate over each key and filter based on the time range  for i, key in ipairs(keys) do  local time = redis.call('HGET', key, 'time')  if time then  time = tonumber(time) -- Convert to number  if time >= start\_time and time < end\_time then  local sensor\_data = {}  sensor\_data.sensor\_id = redis.call('HGET', key, 'sensor\_id')  sensor\_data.temperature = redis.call('HGET', key, 'temperature')  sensor\_data.humidity = redis.call('HGET', key, 'humidity')  table.insert(results, sensor\_data)  end  end  end  -- End time measurement  local end\_time\_measurement = redis.call('TIME')[1] + redis.call('TIME')[2] / 1000000  local query\_time = (end\_time\_measurement - start) \* 1000 -- Execution time in ms  -- Log the execution time  redis.log(string.format('Time-Range Query Time (ms): %.6f', query\_time))  return results -- Return the results of the time range query |

|  |  |
| --- | --- |
| **Apache Cassandra** | **Code** |
| **Query Response Time** | const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace', // Replace with your actual keyspace name  });  async function queryAllData() {  const query = 'SELECT \* FROM sensor\_data\_keyspace.sensor\_data';  let pageState = null; // Used for pagination  let allRows = [];  const startTime = Date.now(); // Start time for the operation  try {  console.log('Fetching all records...');  // Loop through pages of data  do {  const result = await client.execute(query, [], { fetchSize: 1000, pageState: pageState });  allRows = allRows.concat(result.rows); // Concatenate the fetched rows  // Set the pageState for the next iteration if there are more rows  pageState = result.pageState;  console.log(`Fetched ${result.rows.length} rows...`);  } while (pageState); // Continue if there is more data  const endTime = Date.now(); // End time for the operation  const elapsedTime = endTime - startTime; // Time taken in milliseconds  console.log(`Total rows fetched: ${allRows.length}`);  console.log(`Total time taken: ${elapsedTime} ms`); // Output the time taken  } catch (err) {  console.error('Error executing query:', err);  } finally {  await client.shutdown();  }  }  queryAllData(); |
| **Complex Query Performance** | const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace', // Replace with your actual keyspace name  });  async function calculateAvgTemperature() {  const query = 'SELECT sensor\_id, temperature FROM sensor\_data\_keyspace.sensor\_data';  let pageState = null; // For pagination  let allRows = [];  const startTime = new Date(); // Record start time  try {  console.log('Fetching records to calculate average temperature...');  // Loop through paginated results  do {  const result = await client.execute(query, [], { fetchSize: 1000, pageState: pageState });  allRows = allRows.concat(result.rows); // Concatenate the fetched rows  // Set the pageState for the next iteration if more data exists  pageState = result.pageState;  console.log(`Fetched ${result.rows.length} rows...`);  } while (pageState); // Continue if there is more data  // Calculate average temperature per sensor\_id  const sensorAvgTemps = {};  allRows.forEach(row => {  const sensorId = row.sensor\_id;  const temperature = row.temperature;  if (!sensorAvgTemps[sensorId]) {  sensorAvgTemps[sensorId] = { totalTemp: 0, count: 0 };  }  sensorAvgTemps[sensorId].totalTemp += temperature;  sensorAvgTemps[sensorId].count += 1;  });  // Calculate and display the average temperature for each sensor\_id  console.log('Average Temperatures by sensor\_id:');  for (const sensorId in sensorAvgTemps) {  const avgTemp = sensorAvgTemps[sensorId].totalTemp / sensorAvgTemps[sensorId].count;  console.log(`Sensor ID: ${sensorId}, Average Temperature: ${avgTemp.toFixed(2)}`);  }  const endTime = new Date(); // Record end time  const elapsedTime = endTime - startTime; // Calculate time taken in ms  console.log(`Aggregation Query Time (ms): ${elapsedTime}`);  } catch (err) {  console.error('Error executing query:', err);  } finally {  await client.shutdown(); // Gracefully shutdown the client  }  }  calculateAvgTemperature(); |
| **Time-Range Query Efficiency** | const cassandra = require('cassandra-driver');  // Initialize the Cassandra client  const client = new cassandra.Client({  contactPoints: ['127.0.0.1'], // Replace with your actual Cassandra node IP  localDataCenter: 'datacenter1', // Replace with your datacenter name  keyspace: 'sensor\_data\_keyspace', // Replace with your keyspace name  });  async function queryTimeRange() {  const startTime = new Date(); // Record start time  const query = "SELECT \* FROM sensor\_data WHERE time >= ? AND time < ? ALLOW FILTERING";  const params = ['2024-12-12 00:00:00', '2024-12-13 00:00:00']; // Time range for the query  try {  const result = await client.execute(query, params, { prepare: true });  const endTime = new Date(); // Record end time  const elapsedTime = endTime - startTime; // Calculate time taken in ms  console.log(`Query executed successfully.`);  console.log(`Time taken: ${elapsedTime} ms`);  console.log(`Fetched ${result.rows.length} rows.`);  } catch (err) {  console.error('Error executing query:', err);  } finally {  client.shutdown(); // Gracefully shut down the client connection  }  }  // Run the time range query  queryTimeRange(); |

|  |  |
| --- | --- |
| **Apache IoTDB** | **Code** |
| **Query Response Time** | from iotdb.Session import Session  from datetime import datetime  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def query\_data():  """Queries data from the sensor\_data table and measures the time taken for the query."""  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)    try:  # Start timing the query operation  start\_time = datetime.now()  # Execute SELECT query  select\_sql = "SELECT \* FROM root.sensor\_data"  result\_set = session.execute\_query\_statement(select\_sql)  # Read the result (optional: print or process data)  while result\_set.has\_next():  row = result\_set.next()  print(row) # You can process the row here if needed    # End timing the query operation  end\_time = datetime.now()  elapsed\_time = (end\_time - start\_time).total\_seconds() \* 1000 # Time in milliseconds  print(f"Query Response Time: {elapsed\_time:.2f} ms")  return elapsed\_time  except Exception as e:  print(f"Error while querying data: {e}")  finally:  session.close()  def main():  # Perform the query and output the time taken  query\_time = query\_data()  print(f"Total Query Time: {query\_time:.2f} ms")  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| **Complex Query Performance** | from iotdb.Session import Session  from datetime import datetime  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def execute\_count\_query():  """Executes a query to count the number of sensor\_id entries based on temperature and humidity ranges."""  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)  try:  # Start timing the query  start\_time = datetime.now()  # Example: Count the number of sensor entries where:  # 1. Temperature > 25 AND < 70  # 2. Humidity > 10 AND < 30  select\_sql = (  "SELECT COUNT(sensor\_id) FROM root.sensor\_data "  "WHERE temperature > 25 AND temperature < 70 "  "AND humidity > 10 AND humidity < 30"  )  result\_set = session.execute\_query\_statement(select\_sql)  # End timing the query  end\_time = datetime.now()  elapsed\_time = (end\_time - start\_time).total\_seconds() \* 1000 # Time in milliseconds  print(f"Query Execution Time: {elapsed\_time:.2f} ms")  return elapsed\_time  except Exception as e:  print(f"Error while executing the query: {e}")  finally:  session.close()  def main():  # Perform the count query  query\_time = execute\_count\_query()  if query\_time is not None:  print(f"Total Query Time: {query\_time:.2f} ms")  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| **Time-Range Query Efficiency** | from iotdb.Session import Session  from datetime import datetime  # Configuration for IoTDB connection  IOTDB\_HOST = "127.0.0.1" # Replace with your IoTDB host  IOTDB\_PORT = "6667" # Default IoTDB port  IOTDB\_USER = "root" # Default username  IOTDB\_PASSWORD = "root" # Default password  def convert\_to\_timestamp(time\_str):  """Converts a datetime string to a Unix timestamp (milliseconds since epoch)."""  dt = datetime.strptime(time\_str, '%Y-%m-%d %H:%M:%S')  timestamp = int(dt.timestamp() \* 1000) # Convert to milliseconds  return timestamp  def execute\_time\_range\_query():  """Executes a time-range query on the sensor\_data table."""  session = Session(IOTDB\_HOST, IOTDB\_PORT, IOTDB\_USER, IOTDB\_PASSWORD)  session.open(False)  try:  # Start timing the query  start\_time = datetime.now()  # Convert datetime strings to timestamps  start\_timestamp = convert\_to\_timestamp('2024-12-14 00:00:00')  end\_timestamp = convert\_to\_timestamp('2024-12-15 00:00:00')  # Time range query with Unix timestamps  select\_sql = (  f"SELECT \* FROM root.sensor\_data "  f"WHERE time BETWEEN {start\_timestamp} AND {end\_timestamp}"  )  result\_set = session.execute\_query\_statement(select\_sql)  # End timing the query  end\_time = datetime.now()  elapsed\_time = (end\_time - start\_time).total\_seconds() \* 1000 # Time in milliseconds  print(f"Query Execution Time: {elapsed\_time:.2f} ms")  return elapsed\_time  except Exception as e:  print(f"Error while executing the query: {e}")  finally:  session.close()  def main():  # Perform the time-range query  query\_time = execute\_time\_range\_query()  if query\_time is not None:  print(f"Total Query Time: {query\_time:.2f} ms")  if \_\_name\_\_ == "\_\_main\_\_":  main() |