**How to Reduce Time Series Database Data Volume and Improve Query Performance**

**Abstract:**

"How to Reduce Time Series Database Data Volume and Improve Query Performance" is a comprehensive guide for managing time-series data. This discusses methods for reducing the data volume in time-series databases and enhancing query performance. The techniques presented include data compression, data summarization, and data retention policies. This also provides an in-depth analysis of the trade-offs between data volume reduction and query performance, and helps readers understand the factors that impact query performance. This guide will help data administrators and architects to effectively manage time-series data and ensure that their time-series databases remain performant and scalable.

**Several options for managing the data efficiently:**

1. **Data partitioning**: Divide the data into smaller partitions based on time intervals or other criteria to reduce the query time and improve performance.
2. **Data Retention Policy**: Implement a retention policy to automatically delete old data that is no longer needed. This will help keep the data volume under control.
3. **Data compression**: Compress the data to reduce storage space and increase query speed.
4. **Indexing**: Index the columns that are frequently used in queries to speed up data retrieval.
5. **Caching**: Store frequently used data in memory to reduce the number of disk based I/O operations.
6. **Data archiving**: Move older, less frequently used data to lower-cost storage to reduce the cost of storage.
7. **Query optimization**: Ensure that the queries are optimized for the specific use case, indexing the right columns, and using appropriate query techniques.

**Data reduction:** Itis the process of reducing the size of data to improve the efficiency of storage and processing. If there is a time series database with tables containing thousands of rows or more, you can consider the following methods for data reduction:

1. **Data aggregation**: Summarize the data into a smaller number of records based on time intervals, averages, or other aggregation methods.
2. **Data compression**: Use algorithms such as Gzip or Snappy to compress the data, which can reduce the storage space required by up to 90%.
3. **Data sampling**: Select a representative subset of the data for analysis, which can reduce the size of the data without affecting the accuracy of the results.
4. **Data normalization**: Normalize the data to reduce the number of redundant records, which can reduce the size of the data and improve the efficiency of processing.
5. **Data deduplication**: Remove duplicated records from the data, which can reduce the size of the data and improve the efficiency of processing.
6. **Data compression algorithms**: Algorithms like Huffman coding, and arithmetic coding can be applied to compress the data, reducing its size and improving efficiency.

**Query optimization:** It is the process of improving the performance of database queries by reducing the amount of data that needs to be processed. If you have a time series database with tables containing thousands of rows or more, you can consider the following methods for query optimization.

1. **Indexing**: Create indexes on frequently used columns to speed up data retrieval.

**Present**: CREATE INDEX cpuusage\_timestamp\_idx ON idrac.cpuusage USING btree ("timestamp" DESC).

**Suggestion**: CREATE INDEX cpuusage\_timestamp\_nodeid\_idx ON idrac.cpuusage USING btree ("timestamp" DESC, "nodeid");

1. **Partitioning**: Divide the data into smaller partitions based on time intervals or other criteria to reduce the amount of data that needs to be processed for each query.
2. **Caching**: Store frequently used data in memory to reduce the number of disk-based I/O operations and speed up data retrieval.
3. **Materialized Views**: Create pre-computed views of data, which can be used to speed up queries that access the same data repeatedly.
4. **Query optimization techniques**: Use techniques such as query plan analysis, database profiling, and database tuning to optimize the performance of queries.

**Example**: EXPLAIN SELECT date\_trunc('hour', timestamp) AS hour, SUM(value) AS total\_value,AVG(value) AS average\_value, MIN(value) AS min\_value, MAX(value) AS max\_value FROM data\_table GROUP BY hour ORDER BY hour;

1. **Avoiding expensive operations**: Avoid operations that are computationally expensive, such as sorting and grouping, in favor of operations that are more efficient, such as filtering and aggregating.
2. **Appropriate data types:** Use appropriate data types for columns in the database, such as integers instead of strings, to reduce the size of the data and improve query performance.
3. **Database Tuning**: Regular tuning of the database parameters and configuration can help optimize query performance.
4. **Use of aggregated data**: Utilizing aggregated data can reduce the amount of data that needs to be processed, which can improve query performance.
5. **Query Rewrite**: Query rewriting involves changing the structure of the query to make it more efficient. This can involve using subqueries, aggregating data, or using window functions.
6. **Partitioning**: Partitioning the data can help reduce the amount of data that needs to be scanned when executing a query.

**The time it takes for a query to execute on a time series database:**

1. **The size of the data**: The larger the data, the more time it will take to process.
2. **The complexity of the query**: Queries with complex logic, such as multiple join operations, sorting, and grouping, will take longer to execute than simple queries.
3. **The hardware**: The performance of the query will depend on the hardware, such as the CPU, RAM, and storage, that is available to the database.
4. **The database configuration**: The performance of the query will depend on the configuration of the database, such as the buffer pool size and the number of concurrent connections.
5. **The network speed**: If the data is stored on a remote server, the speed of the network connection will impact the performance of the query.

**Faster way to perform data reduction and query optimization on time scale database with multiple tables:**

There are several ways to perform data reduction and query optimization on a time scale database with multiple tables, and the specific approach will depend on the specific use case and the data structure. However, here are a few general approaches to consider:

1. **Using columnar databases**: Columnar databases store data in columns rather than rows, which can greatly improve the performance of analytical queries and reduce the amount of data that needs to be processed.
2. **Using in-memory databases**: In-memory databases store data in RAM, which can greatly improve the performance of analytical queries.
3. **Using a distributed database**: A distributed database spreads data across multiple nodes, which can greatly improve the performance of analytical queries.
4. **Using specialized analytical databases**: Specialized analytical databases are designed specifically for performing analytical queries on large data sets and can often provide much better performance than general-purpose databases.
5. **Using data warehousing solutions**: Data warehousing solutions are designed for managing and analyzing large data sets and can provide many optimizations for both data reduction and query performance.
6. **Using caching**: Caching can greatly improve the performance of frequently executed queries by storing the results of a query in memory for quick access.
7. **Using materialized views**: Materialized views can be used to pre-compute and store the results of frequently executed queries, which can greatly improve the performance of these queries.
8. **Using data lake solutions**: Data Lake solutions are designed for storing and processing large data sets and can provide many optimizations for both data reduction and query performance.

**Data deduplication** is the process of removing duplicate data records from a database. This can help reduce the amount of storage required, improve data accuracy, and make data management easier. Here are some steps for performing data deduplication on a time series database:

1. **Identify duplicate data**: Determine the columns that should be considered when determining if a record is a duplicate. For example, you may want to consider only the time column, or you may also want to consider other columns such as value.
2. **Create a unique constraint**: Create a unique constraint on the columns that should be considered when determining if a record is a duplicate. This will prevent new duplicates from being inserted into the table.
3. **Remove existing duplicates:** Remove existing duplicates using a SQL query.

**Compression**: Compression algorithms can be used to reduce the size of the data by removing redundant information. Time series data often contains long stretches of identical or nearly identical values, which can be compressed using techniques such as delta encoding, run-length encoding, or Huffman coding.

**Example: Run-Length Encoding**

import pandas as pd

# Load data into a pandas dataframe

data = pd.read\_csv('/content/cpu\_usage - Sheet1 (1).csv')

# Define a function for run-length encoding

def rle\_encode(data):

encoded\_data = []

count = 1

current\_value = data[0]

for value in data[1:]:

if value == current\_value:

count += 1

else:

encoded\_data.append((current\_value, count))

count = 1

current\_value = value

# Append the last run to the encoded data

encoded\_data.append((current\_value, count))

return encoded\_data

# Apply run-length encoding to each column of the data

encoded\_data = {}

for column in data.columns:

encoded\_data[column] = rle\_encode(data[column].tolist())

# Print the encoded data

print(encoded\_data)

**Output:**

{'timestamp': [('2022-01-26 17:59:58.000 -0600', 1), ('2022-01-26 17:59:53.000 -0600', 1), ('2022-01-26 17:59:48.000 -0600', 1), ('2022-01-26 17:59:44.000 -0600', 1), ('2022-01-26 17:59:39.000 -0600', 1), ('2022-01-26 17:59:34.000 -0600', 1), ('2022-01-26 17:59:30.000 -0600', 1), ('2022-01-26 17:59:25.000 -0600', 1), ('2022-01-26 17:59:20.000 -0600', 1)], 'node id': [(1, 9)], 'source': [('systemusage', 9)], 'fqdd': [('SystemUsage', 9)], 'value': [(64, 3), (61, 2), (62, 2), (63, 2)]}

**Downsampling**: Downsampling involves reducing the number of data points in a time series by aggregating multiple data points into a single point. For example, if a time series contains hourly data points, downsampling to daily data points can reduce the storage requirements by a factor of 24.

**Chunking**: Chunking involves dividing a time series into smaller chunks and storing only the unique chunks. This technique can be particularly useful for time series data that contains repeated patterns or cycles.

**Hashing**: Hashing involves mapping each data point to a unique hash value, which can then be used to identify duplicate data points. This technique can be useful for identifying duplicates in large datasets but may be less effective for smaller datasets where the overhead of hashing may outweigh the benefits.

**Delta encoding**: Delta encoding involves storing only the difference between each successive data point, rather than the actual value. This can be particularly useful for time series data that contains incremental changes or trends over time.

**Data aggregation:**

1. Max/Min/Average/Median Aggregation: These techniques are used to calculate the maximum, minimum, average or median value of a set of values over a given time period.
2. Summation Aggregation: This technique is used to calculate the sum of a set of values over a given time period.
3. Count Aggregation: This technique is used to count the number of values over a given time period.
4. Sampling Aggregation: This technique is used to take a sample of the data at a given frequency, rather than aggregating over a fixed time period.
5. Interpolation Aggregation: This technique is used to fill in missing data points in the time series by interpolating values based on nearby data points.
6. Quantization Aggregation: This technique is used to group data into bins based on the value range, and then aggregate the values in each bin.
7. Residual Aggregation: This technique is used to calculate the difference between the actual data and a model that is fit to the data, and then aggregate the residuals over a given time period.

**Quantization Aggregation:**

# Step 1: Read the time series data into a pandas dataframe

df = pd.read\_csv('/content/cpu\_usage - Sheet1 (2).csv')

# Step 2: Convert the timestamp column to datetime format and set it as the index

df['timestamp'] = pd.to\_datetime(df['timestamp'], format='%Y-%m-%d %H:%M:%S')

df = df.set\_index('timestamp')

# Step 3: Quantize the values of the data

quantization\_factor = 10

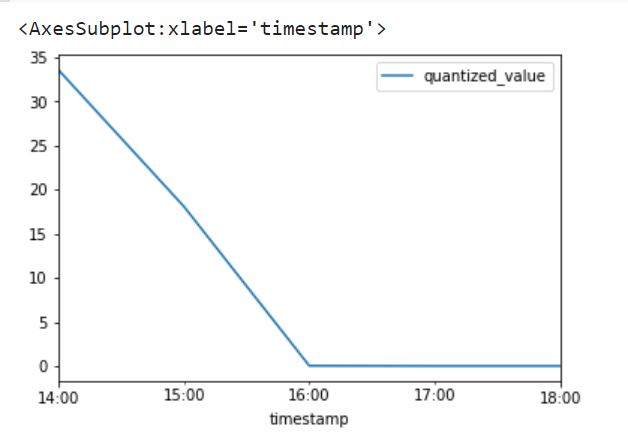
df['quantized\_value'] = (df['value'] // quantization\_factor) \* quantization\_factor

# Step 4: Group the data by hour and aggregate the values by the mean of the quantized values

df\_agg = df.groupby(pd.Grouper(freq='1H')).agg({'quantized\_value': 'mean'})

# Step 5: Plot the aggregated data

df\_agg.plot()



**Residual Aggregation:**

**import pandas as pd**

**# Load the data from file**

**df = pd.read\_csv('/content/cpu\_usage - Sheet1 (2).csv')**

**# Convert timestamp column to datetime format and set as index**

**df['timestamp'] = pd.to\_datetime(df['timestamp'])**

**df = df.set\_index('timestamp')**

**# Group the data by date and calculate the daily mean**

**daily\_mean = df.resample('D'****).mean()**

**# Calculate the residuals by subtracting the daily mean from the original values**

**residuals = df - daily\_mean.reindex(df.index, method='ffill')**

**# Group the residuals by week and calculate the weekly mean**

**weekly\_mean\_residuals = residuals.resample('W'****).mean()**

**# Add the daily mean back to the weekly mean residuals to get the weekly mean**

**weekly\_mean = weekly\_mean\_residuals.add(daily\_mean, fill\_value=0)**

**# Plot the original data, daily mean, and weekly mean**

**df.plot(label='Original data')**

**daily\_mean.plot(label='Daily mean')**

**weekly\_mean.plot(label='Weekly mean')**

**plt.legend()**

**plt.show()**



**1)** SELECT DATE(timestamp) as date, HOUR (timestamp) as hour, AVG(value) as avg\_value

FROM idrac9.cpuusage c GROUP BY date, hour ORDER BY date, hour

2)SELECT AVG(column\_name), MIN(column\_name), MAX(column\_name),

COUNT(column\_name), SUM(column\_name) FROM idrac9.cpuusage c

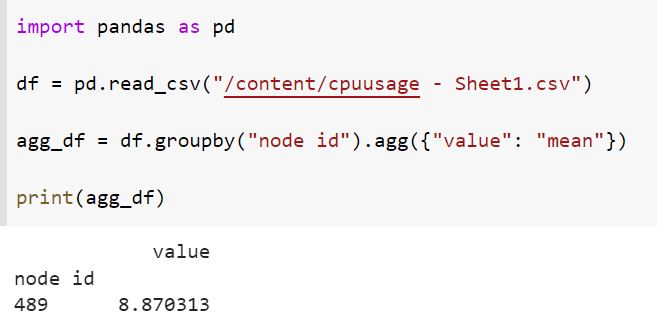
GROUP BY time\_interval;

3) import pandas as pd

df = pd.read\_csv("/content/cpuusage - Sheet1.csv")

agg\_df = df.groupby("node id").agg({"value": "mean"})

print(agg\_df)



4) WITH grouped\_data AS (SELECT date\_trunc('hour', timestamp) AS hour, SUM(value) AS total\_value,

AVG(value) AS average\_value, MIN(value) AS min\_value,MAX(value) AS max\_value FROM

Idrac9.cpuusage c GROUP BY hour)

SELECT hour, total\_value, average\_value, min\_value, max\_value FROM grouped\_data ORDER BY

hour

5) CREATE INDEX IF NOT EXISTS idx\_timestamp\_hour ON idrac9.cpuusage c (timestamp, date\_trunc('hour', timestamp));

SELECT date\_trunc('hour', timestamp) AS hour, SUM(value) AS total\_value,AVG(value) AS average\_value,

MIN(value) AS min\_value, MAX(value) AS max\_value FROM data\_table GROUP BY

hour ORDER BY hour;

**Data Retention:**

CREATE PROCEDURE retention\_policy()

BEGIN DELETE FROM time\_series\_data WHERE timestamp < DATE\_SUB(NOW(), INTERVAL 90 DAY);

END;

**Data compression:**

1)ALTER TABLE table\_name SET compression = 'GZIP'

2) import gzip

# Compress data

with open('data.txt', 'rb') as f\_in:

with gzip.open('data.txt.gz', 'wb') as f\_out:

f\_out.writelines(f\_in)

# Decompress data

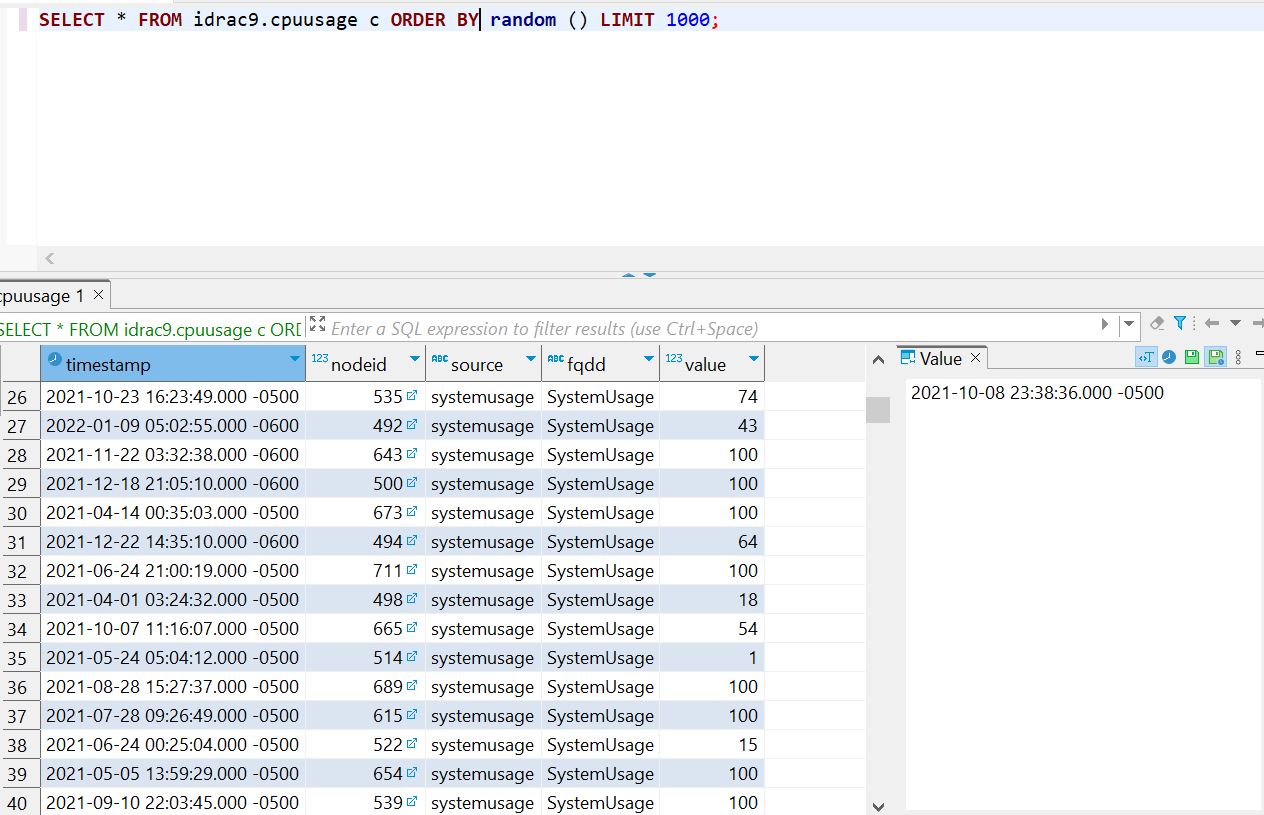
with gzip.open('data.txt.gz', 'rb') as f\_in:

with open('data.txt', 'wb') as f\_out:

f\_out.writelines(f\_in)

**Data sampling:**

SELECT \* FROM idrac9.cpuusage c ORDER BY random () LIMIT 1000;



**Data normalization:**

CREATE TABLE new\_table\_name AS SELECT DISTINCT column1, column2, ...

FROM table\_name;

**Caching**:

import redis

# Connect to Redis

r = redis.Redis(host='localhost', port=6379, db=0)

# Fetch data from database and store in Redis cache

data = fetch\_data\_from\_database()

r.set('cached\_data', data, ex=3600) # Cache data for 1 hour (3600 seconds)

# Retrieve cached data from Redis

cached\_data = r.get('cached\_data')

# Use cached data if available, otherwise fetch from database

if cached\_data:

data = cached\_data

else:

data = fetch\_data\_from\_database()

**Data deduplication:**

CREATE TABLE new\_table\_name

AS

SELECT MIN (column1), MIN (column2), ...

FROM table\_name

GROUP BY column1, column2, ...;

**SQL queries for query optimization:**

Indexing:

CREATE INDEX index\_name

ON table\_name (column1, column2, ...);

**Partitioning**:

CREATE TABLE partitioned\_table\_name (

column1 data\_type,

column2 data\_type,

...,

PRIMARY KEY (column1, column2, ...)

)

PARTITION BY RANGE (column1);

Caching:

SELECT column1, column2, ...

FROM table\_name

CACHE;

**Materialized Views:**

CREATE MATERIALIZED VIEW view\_name

AS

SELECT column1, column2, ...

FROM table\_name;

**Avoiding expensive operations:**

SELECT column1, AVG (column2)

FROM table\_name

GROUP BY column1;

**Appropriate data types:**

ALTER TABLE table\_name

MODIFY COLUMN column\_name INTEGER;

**Example SQL queries that can be used to work with a time series database using PostgreSQL:**

1.Creating a table with a timestamp column:

CREATE TABLE time\_series\_table (

time\_column TIMESTAMP,

value\_column REAL,

...

);

2.Inserting data into the table:

INSERT INTO time\_series\_table (time\_column, value\_column, ...)

VALUES (now(), 10.0, ...);

3. Selecting data from the table:

SELECT \* FROM time\_series\_table

WHERE time\_column BETWEEN '2021-01-01' AND '2021-12-31';

4.Aggregating data by time:

SELECT date\_trunc('hour', time\_column) as hour, AVG(value\_column) FROM time\_series\_table GROUP BY hour;

5) SELECT column\_name, SUM(column\_name) FROM table\_name

GROUP BY column\_name HAVING SUM(column\_name) > 100;

6) SELECT COUNT(\*) FROM table\_name;

7) SELECT MIN(column\_name), MAX(column\_name) FROM table\_name;

8) SELECT AVG(column\_name), SUM(column\_name) FROM table\_name;

9) SELECT DISTINCT column\_name FROM table\_name;

10) Avoiding expensive operations:

SELECT column1, AVG(column2) FROM table\_name GROUP BY column1;

**SQL query for aggregating data:**

SELECT date\_trunc('hour', timestamp) as hour, AVG(value) FROM my\_table

WHERE node\_id = 1 AND source = 'systemusage' AND fqdd = 'SystemUsage'

GROUP BY hour ORDER BY hour;

**Creating an index on the time column**:  
CREATE INDEX time\_series\_table\_time\_column\_idx

ON time\_series\_table (time\_column);

**Creating a materialized view:**

CREATE MATERIALIZED VIEW time\_series\_table\_view

AS SELECT date\_trunc('hour', time\_column) as hour, AVG(value\_column)

FROM time\_series\_table GROUP BY hour;

**Refreshing the materialized view**:  
REFRESH MATERIALIZED VIEW time\_series\_table\_view;

Data deduplication is the process of removing duplicate data records from a database. This can help reduce the amount of storage required, improve data accuracy, and make data management easier. Here are some steps for performing data deduplication on a time series database:

Remove existing duplicates: Remove existing duplicates using a SQL query.

DELETE FROM time\_series\_table t

WHERE t.time\_column IN (

SELECT time\_column

FROM time\_series\_table

GROUP BY time\_column

HAVING COUNT(\*) > 1

);

Sharding is a process of horizontally partitioning a large data set into smaller, manageable parts called shards. It is used to improve the performance, scalability, and reliability of a database system.

To perform sharding for the above table, you can divide the data based on a specific field, such as the **nodeid** field. For example, you can create multiple tables, each containing data for a specific nodeid.

Here's an example of how to perform sharding for the above table:

**# Create a function to determine which table to store data in**

**def determine\_shard(nodeid):**

**if nodeid == 1:**

**return 'node1\_data'**

**elif nodeid == 2:**

**return 'node2\_data'**

**# ... add additional conditions for other nodeids**

**# Store data in appropriate shard table**

**shard\_tables = {}**

**for row in data:**

**nodeid = row['nodeid']**

**shard\_table = determine\_shard(nodeid)**

**if shard\_table not in shard\_tables:**

**shard\_tables[shard\_table] = []**

**shard\_tables[shard\_table].append(row)**

**# Store data in separate tables for each shard**

**for shard\_table, shard\_data in shard\_tables.items():**

**with open(shard\_table + '.txt', 'w') as f:**

**for row in shard\_data:**

**f.write(','.join(row.values()) + '\n')**

Materialized views can be used to pre-compute and store the results of a query in a table-like structure, so that subsequent queries can retrieve the results faster. To perform materialized views for the above data, you can follow these steps:

1. Define the query that you want to materialize. This query should be based on the data you want to optimize access to.
2. Create the materialized view table. You can do this by executing a SQL command similar to CREATE TABLE, but with the addition of the keyword "MATERIALIZED VIEW".
3. Fill the materialized view table with data by executing the query. This can be done with a REFRESH command or by setting up a schedule for automatic refresh.
4. Use the materialized view in place of the original query when querying the data. The database management system will automatically use the pre-computed results stored in the materialized view instead of re-computing the query each time.

Note: The materialized view data can become stale over time as the underlying data changes, so it's important to refresh the materialized view periodically to keep its data up to date.

**Data reduction:**

Data reduction is a process of simplifying complex datasets by reducing the number of variables and instances while preserving the essential information. Techniques for data reduction include:

1. Feature selection: Selecting the most important features in a dataset.
2. Feature extraction: Creating new features from the existing features.
3. Feature engineering: Transforming existing features to create new features.
4. Dimensionality reduction: Reducing the number of dimensions in the dataset.

import pandas as pd

import numpy as np

# Load data into a pandas dataframe

df = pd.read\_csv("data.csv")

# Convert the timestamp column to a datetime object

df['timestamp'] = pd.to\_datetime(df['timestamp'], format='%Y-%m-%d %H:%M:%S.%f %z')

# Calculate the difference between consecutive rows of value column

df['diff'] = df['value'].diff()

# Find the index of rows where the difference is greater than 2

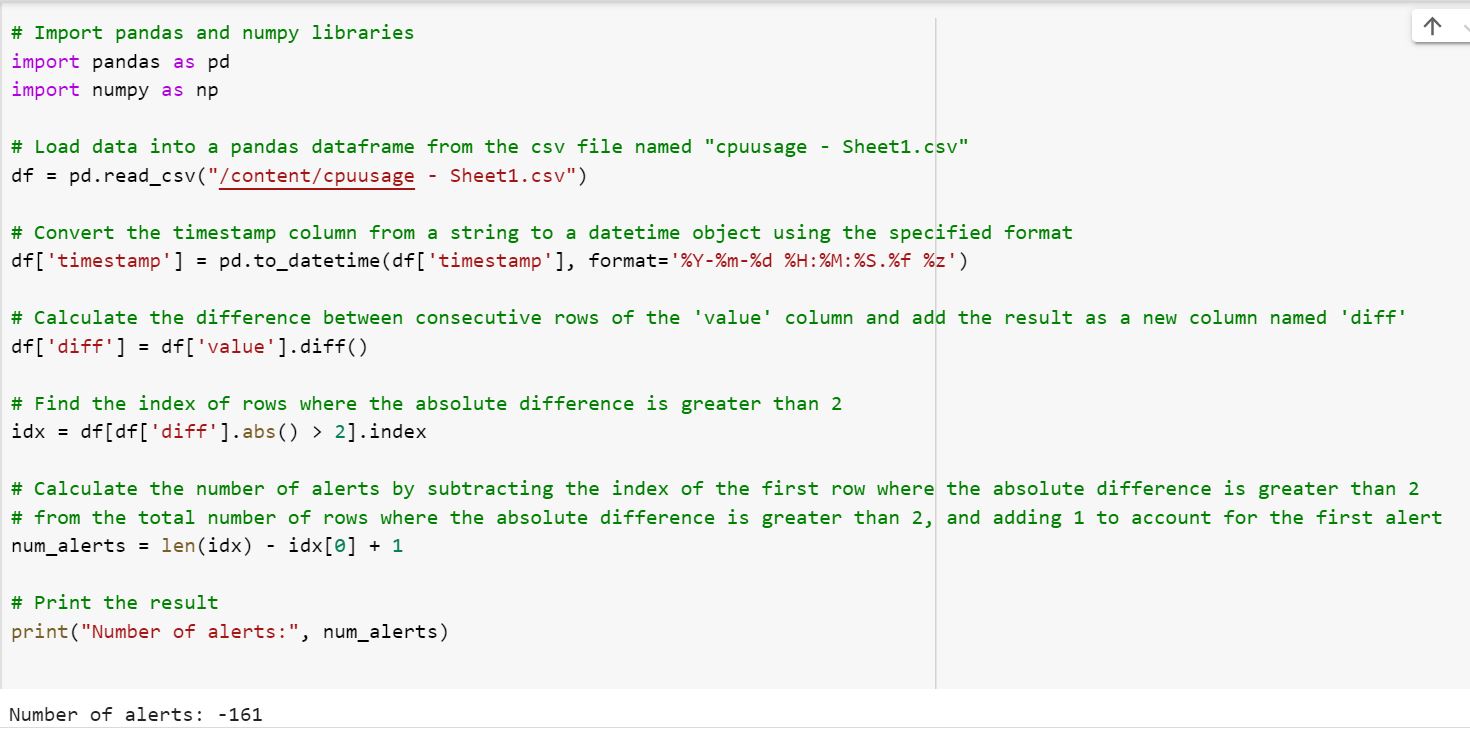
idx = df[df['diff'].abs() > 2].index

# Get the number of alerts by subtracting the number of first rows from the number of rows with difference greater than 2

num\_alerts = len(idx) - idx[0] + 1

# Print the result

print("Number of alerts:", num\_alerts)



The above code can be considered as a form of data reduction since it reduces the amount of data that needs to be analyzed. Instead of analyzing all data points in the CPU usage dataset, the code identifies the rows where the absolute difference in CPU usage is greater than 2 and analyzes only those rows. This reduces the amount of data that needs to be analyzed and allows for a more focused analysis of the data.

**Pre-process the data:**

import pandas as pd

# Read in the data

data = pd.read\_csv("data.csv")

# Convert the timestamp column to a datetime format

data['timestamp'] = pd.to\_datetime(data['timestamp'], format="%Y-%m-%d %H:%M:%S.%f %z")

# Normalize the value column

data['value'] = (data['value'] - data['value'].mean()) / data['value'].std()

# Show the first 5 rows of the data

print(data.head())

**Different queries to check the effectiveness:**

1) SELECT AVG(value) FROM idrac9.cpuusage c

2) SELECT MAX(value) FROM idrac9.cpuusage c

3) SELECT MIN(value)FROM idrac9.cpuusage c

4) SELECT COUNT(DISTINCT value) FROM idrac9.cpuusage c

5) SELECT value, COUNT(value) FROM idrac9.cpuusage c GROUP BY value

6) SELECT value FROM idrac9.cpuusage c ORDER BY value ASC

7) SELECT value FROM idrac9.cpuusage c ORDER BY value DESC

8) SELECT node id, AVG(value) FROM idrac9.cpuusage c GROUP BY node id

Data compression is a process of reducing the size of a dataset by removing redundant information or transforming it into a more compact format. Some common methods of data compression include lossless compression, lossy compression, and entropy encoding.

For the above dataset, one could perform lossless compression using algorithms such as Huffman coding, run length encoding, or arithmetic coding. Lossless compression ensures that all original information is preserved but may not produce as high a compression ratio as lossy methods.

**Run length encoding:**

import pandas as pd

# Load data into a pandas dataframe

data = pd.read\_csv('/content/cpu\_usage - Sheet1 (1).csv')

# Define a function for run-length encoding

def rle\_encode(data):

encoded\_data = []

count = 1

current\_value = data[0]

for value in data[1:]:

if value == current\_value:

count += 1

else:

encoded\_data.append((current\_value, count))

count = 1

current\_value = value

# Append the last run to the encoded data

encoded\_data.append((current\_value, count))

return encoded\_data

# Apply run-length encoding to each column of the data

encoded\_data = {}

for column in data.columns:

encoded\_data[column] = rle\_encode(data[column].tolist())

# Print the encoded data

print(encoded\_data)

**Output**:

{'timestamp': [('2022-01-26 17:59:58.000 -0600', 1), ('2022-01-26 17:59:53.000 -0600', 1), ('2022-01-26 17:59:48.000 -0600', 1), ('2022-01-26 17:59:44.000 -0600', 1), ('2022-01-26 17:59:39.000 -0600', 1), ('2022-01-26 17:59:34.000 -0600', 1), ('2022-01-26 17:59:30.000 -0600', 1), ('2022-01-26 17:59:25.000 -0600', 1), ('2022-01-26 17:59:20.000 -0600', 1)], 'node id': [(1, 9)], 'source': [('systemusage', 9)], 'fqdd': [('SystemUsage', 9)], 'value': [(64, 3), (61, 2), (62, 2), (63, 2)]}

The process to perform data compression on the given data set is as follows:

Identify the repeating values in the "value" column.

Create a frequency count for each unique value.

Replace the repeating values with the frequency count and the unique value.

Store the compressed data in a new format, like a dictionary or key-value pairs.

**Example**: {64: 5, 61: 2, 62: 5, 63: 4, 66: 2, 67: 2}

import pandas as pd

import matplotlib.pyplot as plt

# Step 1: Import the data into a pandas dataframe

df = pd.read\_csv('/content/cpu\_usage - Sheet1.csv')

# Step 2: Filter the data by node id, source and fqdd columns

df = df[(df['node id'] == 1) & (df['source'] == 'systemusage') & (df['fqdd'] == 'SystemUsage')]

# Step 3: Convert the timestamp column to datetime format

df['timestamp'] = pd.to\_datetime(df['timestamp'], format='%Y-%m-%d %H:%M:%S.%f %z')

# Step 4: Set the timestamp column as the index of the dataframe

df = df.set\_index('timestamp')

# Step 5: Group the data by the time period (for example, by minute) and aggregate the values by mean

df = df.sort\_values('timestamp')

df\_agg = df.resample('1min').mean()

# Step 6: Plot the aggregated data

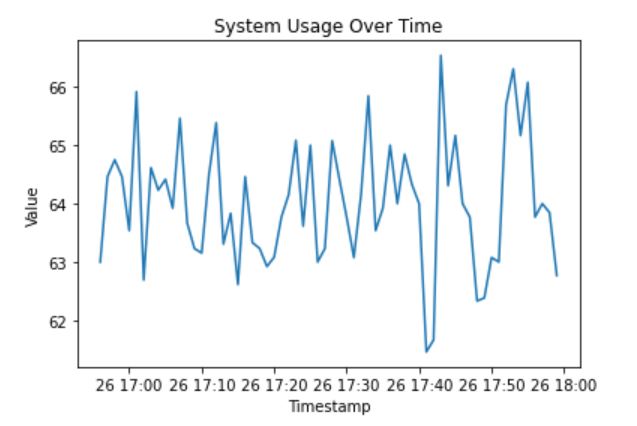
plt.plot(df\_agg['value'])

plt.xlabel('Timestamp')

plt.ylabel('Value')

plt.title('System Usage Over Time')

plt.show()



**A time series database may also need the following:**

**Trigger Functions**: To automatically update data based on certain events or changes.

**Stored Procedures**: To encapsulate complex logic and business rules.

**User-defined Functions:** To extend the functionality of the database with custom code.

**Data Partitioning**: To manage the growing volume of data and improve query performance.

**Backup and Recovery Mechanisms**: To ensure data is safe and secure and can be recovered in case of failure.

**Security Measures:** To control access to the data and protect against unauthorized use.

**Stored procedures:**

CREATE PROCEDURE get\_node\_data(IN nodeID INT) BEGIN SELECT timestamp, nodeid, source, fqdd, value FROM your\_tablE WHERE nodeid = nodeID;

END;

**User-defined functions:**

CREATE FUNCTION get\_node\_average(nodeID INT)

RETURNS FLOAT

BEGIN

DECLARE avgValue FLOAT;

SELECT AVG(value) INTO avgValue

FROM your\_table

WHERE nodeid = nodeID;

RETURN avgValue;

END;

**Data Partitioning:**

**CREATE TABLE your\_table (**

**timestamp TIMESTAMP NOT NULL,**

**nodeid INT NOT NULL,**

**source VARCHAR(255),**

**fqdd VARCHAR(255),**

**value FLOAT NOT NULL,**

**PRIMARY KEY (timestamp, nodeid)**

**)**

**PARTITION BY RANGE (timestamp);**

**CREATE TABLE your\_table\_p1 PARTITION OF your\_table**

**FOR VALUES FROM ('2022-01-01 00:00:00') TO ('2022-01-02 00:00:00');**

**CREATE TABLE your\_table\_p2 PARTITION OF your\_table**

**FOR VALUES FROM ('2022-01-02 00:00:00') TO ('2022-01-03 00:00:00');**