Exploring Compression Techniques on a Time Series Database to reduce Query Latency and Data Volume in HPCC

1st Ayesha Anjum Shaik
Computer Science
Texas Tech University
Lubbock, Texas, USA
ayshaik@ttu.edu

2nd Jie Li
Computer Science
Texas Tech University
Lubbock, Texas, USA
Jie.Li@ttu.edu

3rd Dr. Yong Chen
Computer Science
Texas Tech University
Lubbock, Texas, USA
Yong.Chen@ttu.edu

***Abstract*—Understanding the more integrated view of current and historical data in High Performance Computing center is important to gain more insights, enable significant cost benefits. But in recent years, storage capacity and the query performance has become a growing concern as storage infrastructure is being stretched to its scalability limitations in terms of cost, throughput and capacity due to heavy scientific simulations and training overhead in High Performance Computing Center. This paper aims to provide an overview of leveraging mechanisms to reduce data volume. In this paper, we have conducted a detailed comprehensive study on state-of-the-art lossless compression techniques using large HPCC datasets.**

***Index Terms*—TimescaleDB, Compression, Data Volume, Data Retention, Chunking, Metric Collector,**

I. Introduction

Connecting multiple systems as a unit to work as a single unit which allows large scale parallel computing is the High Performance Computing Center. This is important to process large volumes of data and perform efficient data analysis. At Texas Tech University, HPCC is introduced in early 2017 and maintained in three different data centers. We have partitions in HPCC with one partition consisting of 467 nodes and the other consisting of 240 nodes generating billions of records in database per second. Surplus of data to process can result in the risk of inconsistency and error, followed by the negative effect on strategic decision making. Data Volume Reduction is a method of representing the original data in smaller space where smaller flow is easier to control, clean and process. The current operations are performed in 'Hugo' server equipped with 'integrated Dell Remote Access Controller (iDRAC)' which collects nodes metrics from monitoring sensors.

II. Background and Motivation

Relational Database Management Systems(RDBMS) designed to manage heavy volume of data efficiently is TimescaleDB. Also, there is immense need of applying data reduction techniques for storage, cost, performance efficiency and improved analysis of data. Compression techniques have widely been used in the regular systems to reduce the redundancy of data. Some lossless compression techniques like data de-duplication identifies and eliminates the duplicate contents resulting in reducing data redundancy. This paper aims to provide experimental calculations and evaluations of the state-of-the-art lossless compression techniques using the large TTU HPCC datasets.

Motivated by the lossy compressor techniques like ZFP, SZ, and ISABELA [7] on High Performance Computing Center data

A. Timescaledb inbuilt data volume reduction techniques

1) Chunking: TimeScaleDB divides time-series data in a hypertable into chunks, small logical units of data, each chunk assigned to manage and query large amounts of data in a range of time in an efficient way using less memory and storage for processing of data. 'chunk_time_interval' parameter sets the chunk range of time and chunk_detailed_size' parameter is used to keep a check on size and index of chunk in database. The following command shows how to manually create chunks. The following commands show how to manually show and drop chunks.

```
SELECT show_chunks('idrac8.rpmreading',
   older_than => INTERVAL '24 hours');

SELECT drop_chunks('idrac8.rpmreading',
   INTERVAL '1 month');
```

The figure 1 shows the ER diagram of voltagereading table in idrac8 schema in redraider database in Hugo server.

2) Data Retention Policy: TimeScaleDB allows us to apply retention policies which determines how long the data should be withheld in the database.This helps to reduce the amount of data to be stored and processed. Operations are performed on rpmreading table in idrac8 schema in redraider database in the Hugo server.

```
SELECT add_retention_policy('idrac8.
   rpmreading', INTERVAL '24 hours');

SELECT add_retention_policy('idrac8.
   rpmreading', INTERVAL '24 hours');
```

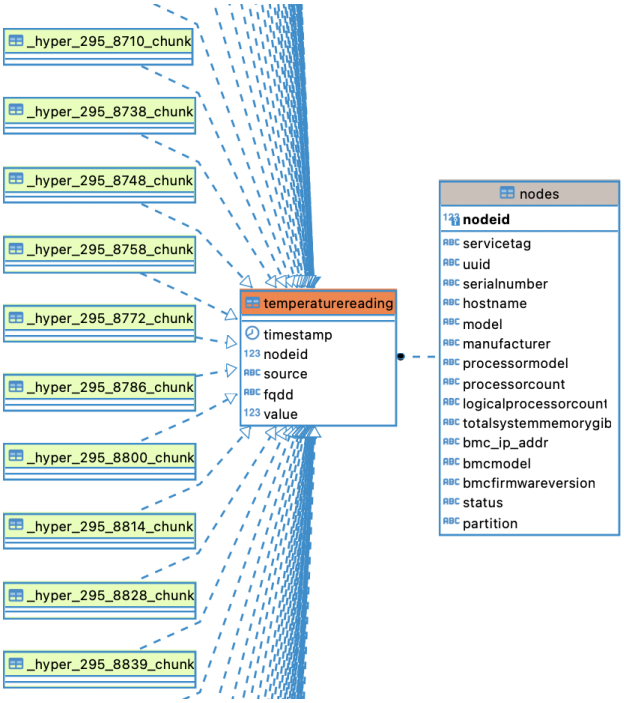

Fig. 1. Representation of voltagereading table in database

3) Compression: TimescaleDB allows us to add an automatic compression policy to reduce the chunk size more than 90%. This is done by using the custom indexes constituting the parameters 'orderby' and 'segmentby' during compression. If we want to add a compression policy manually to compress chunks older than a week, we use the following command.

```
SELECT add_compression_policy('idrac8.
    rpmreading', INTERVAL '7 days');
```

To view the compression policy we have created, we use the following command.

```
SELECT * FROM timescaledb_information.
    jobs WHERE proc_name='idrac8.
    rpmreading';
```

To remove the compression policy we have created, we use the following command.

```
SELECT remove_compression_policy('idrac8.
    rpmreading');
```

The following command shows the details of compression statistics on the voltagereading hypertable of idrac8 schema.

```
SELECT * FROM
    hypertable_compression_stats('idrac8.
    temperaturereading');
```

III. Architecture

In this section, we will see an overview of each functional module in detail.

	Row #1
total_chunks	80
number_compressed_chunks	80
before_compression_table_byte	670,197,334,016
before_compression_index_byte	167,489,970,176
before_compression_toast_byte	655,360
before_compression_total_bytes	837,687,959,552
after_compression_table_bytes	1,054,433,280
after_compression_index_bytes	175,808,512
after_compression_toast_bytes	23,972,610,048
after_compression_total_bytes	25,202,851,840
node_name	[NULL]

Fig. 2. Compression statistics of voltagereading table in database

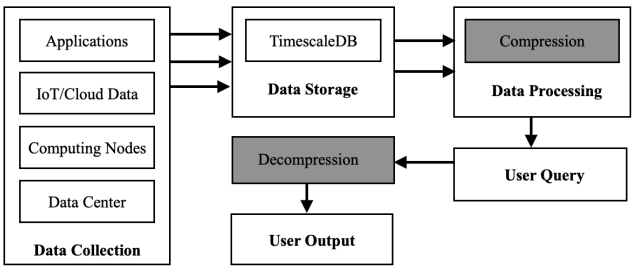

Fig. 3. Architecture

A. Data Collection

High Performance Computing Center at Texas Tech University has a wide range of data sources like the modern computing nodes equipped with BMCs providing Redfish API [3], Resource Managers, Operating Systems. We are using a Metrics Collector, a centralized collecting agent which preprocesses the collected data, builds timestamps and stores data points.

B. Data Storage

Traditional databases are limited which can't store and process this huge amount of time series data. InfluxDB is flexible with many data types being supported, data storage and processing speed, organizing complex indexes but the querying grows tougher. Given TimescaleDB, it is easier to manage the data storage and data management with its inbuilt tools and policies for handling metrics that are timestamped. The Table 1 symbolizes the format in which the time series data of monitoring metrics collected from the data center is stored. The Table II symbolizes some of the metrics and their values. The Table III symbolizes the sample time series data from the database used for the experiments related to the rpmreading table.

TABLE I
DATABASE STORAGE FIELDS

Column Name	*Datatype*
timestamp	timestamptz
nodeid	int4
source	text
fqdd	text
value	int4

TABLE II
DATABASE FIELDS AND VALUES

metric id	*fqdd*
systempowerconsumption	System Power Control
temperaturereading	CPU1 Temp CPU2 Temp Inlet Temp
rpmreading	FAN_1 FAN_2 FAN_3 FAN_4

C. Data Processing

High Performance Computing Center at Texas Tech University has a Metrics Builder. Querying directly from the database includes a lot of historical data ingested and takes a significant amount of waiting and execution time. Metrics Builder acts as the middleware between consumers and producers

D. Data Analysis and Visualization

HiperJobViz [5], visual analytical tool for High Performance Computing Center at Texas Tech University which provides an overview of resource allocations, tracks resource usages by users & jobs, provides a detailed representation of job scheduling and resource usage.

IV. EXPERIMENTS

Data Compression has become extremely crucial in high performance domains these days. There are two types of data compression : Lossy and Lossless.

A. Set Up

1) PostgreSQL: Due to its extravagantly good features of postgreSQL and it also being free and open source, for the efficient query performance and its support, it is chosen Timescale DB supports standard SQL inserts using third party monitoring, collecting and processing tools to build ingest data pipelines.

2) TimescaleDB: TimescaleDB extends PostgreSQL for data representing time-series, assigns PostgreSQL with the high-performance, complex computational, scalable, and analytical capabilities required by modern data-intensive applications.

3) PuTTY (for Windows): We installed a free version of SSH client gateway using my login information so that we could communicate with the Texas Tech University high performance computing center's clusters and access its data tables.

4) GlobalProtect: We used the VPN (Virtual Private Network) to access the data in the HPCC TTU through an SSH client when accessing from outside the campus.

5) DBeaver: We utilized DBeaver to access the tables and run the queries in HPCC due to its capabilities like a Database editor, Data transfer, Database driver, SSH Client, and Multiplatform database.

B. Lossy Data Compression Techniques

Lossy Compression algorithms are used at critical data management times where reconstructed dataset result in the loss of some data compared to that of original dataset though it reduces the size significantly. We can characterize the lossy compressors based on the trial and error approach or the compression ratio.

1) Downsampling: Here, Some of the original data is lost during the process of compression. Though this is used to reduce the storage requirements, simplify the analysis process, increase processing speed, this affects the quality of the data as there is a chance to lose important data.

Algorithm 1: Downsampling of Time series Data

Input: CSV file of time series data
Output: Downsampled CSV file
Record memory_before, start_time;
Start monitoring cpu_usage;
real_data ← stores rpmreading table;
downsampling factor ← 2;
group_data ← real_data.groupby(nodeid,fqdd);
downsampled_data ← mean of every n rows in csv file;
Record memory_after, end_time;
Calculate CPU Usage, Memory Usage, Processing Time during Downsampling;

C. Lossless Data Compression Techniques

1) Run Length Encoding: This method works best in cases of repetitive data and identical sequences which can be stored as a single value and it reduces significant amount of data. The consecutive occurrences of identical sequences or characters is stored along with its count.

For example, if we have a string 'AAAAABBBCCCCCCCD' then the Run Length Encoding stores the string as '5A3B7C1D'. This means 'five A's, three B's, seven C's, one D'. As seen in the above example, requires few characters to store the original string.

From the results, we can understand that run length encoding gives the best output based on criteria, let's see the run length decoding algorithm to retrieve the original output on a user requirement.

2) Huffman Encoding: Huffman Encoding is a loseless compression invented by David A. Huffman in 1952. Here, binary code, which is frequency based is assigned to each symbol, character or sequence of characters, shorter binary code is assigned for longer repetitive strings which results in

Algorithm 2: RunLengthEncoding of Time Series

Input: CSV file of time series data
Output: Run Length Encoded CSV file

```
Record memory_before, start_time
Start monitoring cpu_usage
real_data ← stores rpmreading table
for nodeid in real_data[node_id].unique():
        for source in real_data[source].unique():
                values_encoded ← append(each combination of similar values)
        end
end
data_encoded → append(values_encoded)
Record memory_after, end_time
Calculate CPU Usage, Memory Usage, Processing Time during Run Length Encoding
```

Algorithm 3: RunLengthDecoding of Time Series

Input: CSV file of run length encoded time series data
Output: Run Length Decoded CSV file

```
Open the run length encoded file to read input
Open the run length decoded file to write input
inp ← csv.reader(input_file)
out ← csv.writer(outut_file)
for row in reader :
        Initiate runlengthdecoded_row
        for values in row:
                if ':' in value:
                        split_values ← values.split[:]
                        split_values_count ← int(split_values[0])
                        identical_value ← split_values[1]
                        runlengthdecoded_row ← [identical_value]*split_values_count
                end if
                else:
                        runlengthdecoded_row.append(values)
                end else
        end for
        out.writerow(runlengthdecoded_row)
end for
```

good amount of data volume reduction. This algorithm works by constructing a Huffman tree built based on the frequency of symbols occurred. The frequency of symbols is sorted and assigned to root in ascending order and eventually, tree is built in a way where most frequently occurred character is near to the root of the tree and less frequently occurred character is far away to the root of the tree.

For example, if we have a string 'ABCCADCABD-DDD'. Then the huffman encoding stores the string as ''.

Algorithm 4: Huffman Encoding of Time series

Input: CSV file of time series data
Output: Huffman Encoded CSV file

```
Record memory_before, start_time
Start monitoring cpu_usage
real_data ← stores rpmreading table
value_counts ← real_data[value].value_counts()
for value, count in value_counts.items():
        heaptree ← [count, [value, ""]]
end
while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
                pair[1] ← '0' + pair[1]
        end
        for pair in hi[1:]:
                pair[1] ← '1' + pair[1]
        end
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
end
huffman_codes ← dict(heapq.heappop(heap)[1:])
huffman_encoded_values ← real_data[value].apply(lambda x: huffman_codes[x])
Convert out file to CSV file
Record memory_after, end_time
Calculate CPU Usage, Memory Usage, Processing Time during Huffman Encoding
```

V. RESULTS

To prove the utilization of compression techniques in the architecture is efficient, we compare the compression algorithms with some metrics to be considered along with data volume

TABLE III
RPMREADING TABLE SAMPLE

timestamp	*nodeid*	*source*	*fqdd*	*value*
2023-04-12 17:21:44.000 -0500	1	Thermal.v1_4_0.Fan	FAN_4	9660.0
2023-04-12 17:21:44.000 -0500	1	Thermal.v1_4_0.Fan	FAN_3	9450.0
2023-04-12 17:21:44.000 -0500	1	Thermal.v1_4_0.Fan	FAN_2	9450.0
2023-04-12 17:21:44.000 -0500	1	Thermal.v1_4_0.Fan	FAN_1	9590.0
2023-04-12 17:21:44.000 -0500	2	Thermal.v1_4_0.Fan	FAN_3	9450.0
2023-04-12 17:21:44.000 -0500	2	Thermal.v1_4_0.Fan	FAN_1	9590.0

as follows.The current experiments are performed on a part of the rpmreading table in idrac8 schema in redraider database.

A. Row Count

Comparing the row counts determine the effectiveness of each technique and to what extent the data volume has been reduced to. From the table IV and Fig 4, we can understand that Run Length Encoding then the Huffman Encoding are far better than leaving the data uncompressed.

B. Data Volume

If the goal is to reduce the amount of data being stored or transmitted maintaining the quality of data, we must compare the data volumes to prove which technique is efficiently reducing the data volume. From the table III and Fig 4, we can understand that Run Length Encoding then the Huffman Encoding are far better than leaving the data uncompressed.

C. Compression Ratio

Compression ratio is used to rate the compression algorithm. The compression ratio will always be greater than or equal to one. The more the compression ratio, the better the related algorithm is. From the table III and Fig 4, we can understand that Huffman Encoding and then Run Length Encoding are better than leaving the data uncompressed.
Compression Ratio = $\frac{UncompressedData}{CompressedData}$

D. CPU Usage

If too many processes are lined up, CPU Usage hits 100% and throughput flattens resulting in CPU bottleneck and hence no longer any process can work. It is important to consider CPU Usage in the experiments. From the table III, we can understand that Huffman Encoding and Run Length Encoding are utilizing the CPU efficiently.

E. Memory Usage

If insufficient RAM for executing calculations, it results in a memory bottleneck where data cannot be processed quickly enough and this cuts down on the speed of operations. From the table III, we can understand that Huffman Encoding and Run Length Encoding are utilizing the Memory efficiently.

F. Performance Time

To make sure the technique we are about to use is resource efficient, accurate, speed enough, we must consider the performance time to optimize the resource utilization. From the table III, we can understand that Huffman Encoding and Run Length Encoding are utilizing the Memory efficiently.

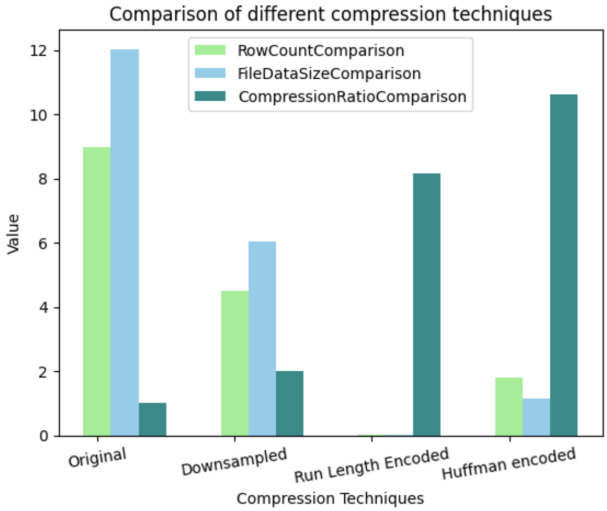

Fig. 4. Comparison of Compression Algorithms on rpmreading table

VI. Conclusion

From the experimental results, we can see that performing run length encoding on the data in database and decoding on a user requirement is an efficient way of approaching the goal of data volume reduction and reducing query latency. However we can perform other techniques on top of compression like de-duplication on the raw data before the data entering into the database.

Acknowledgment

We are very grateful for the high performance computing center at Texas Tech University [1] for providing its resources which were crucial for the completion of this project.

References

[1] HPCC. (2020) High Performance Computing Center. [Online]. Available: http:www.depts.ttu.edu/hpcc/

[2] D. Technologies. (2020) Integrated Dell Remote Access Controller (iDRAC). [Online]. Available: https://www.delltechnologies.com/en-us/solutions/openmanage/idrac.htm

[3] DMTF. (2020) DMTF's Redfish®. [Online]. Available: https://www.dmtf.org/standards/redfish

[4] J. Li et al., "MonSTer: An Out-of-the-Box Monitoring Tool for High Performance Computing Systems," 2020 IEEE International Conference on Cluster Computing (CLUSTER), Kobe, Japan, 2020, pp. 119-129, doi: 10.1109/CLUSTER49012.2020.00022.

[5] N. Nguyen, T. Dang, J. Hass and Y. Chen, "HiperJobViz: Visualizing Resource Allocations in High-Performance Computing Center via Multivariate Health-Status Data," 2019 IEEE/ACM Industry/University Joint International Workshop on Data-center Automation, Analytics, and Control (DAAC), Denver, CO, USA, 2019, pp. 19-24, doi: 10.1109/DAAC49578.2019.00009.

TABLE IV
COMPRESSION ALGORITHMS ON RPMREADING TABLE

Compression Technique	*Row Count*	*File Size*	*Compression Ratio*	*CPU Usage(%)*	*Memory Usage(MB)*	*Processing Time(sec)*
Original Data	897480	12036050	1.00	-	-	-
Downsampled Data	451020	6049435	1.99	4.5	0.0859375	0.0000508
Run Length Encoded Data	1371	1476881	8.15	1.50	3.48828125	0.0000515
Huffman Encoded Data	179496	1133137	10.62	5.00	8.328125	0.0000448

[6] J. Li. (2020) Metrics Builder API. [Online]. Available: https: //influx.ttu.edu:8080/ui/
[7] T. Lu et al., "Understanding and Modeling Lossy Compression Schemes on HPC Scientific Data," 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada, 2018, pp. 348-357, doi: 10.1109/IPDPS.2018.00044.
[8] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn and J. Kunkel, "A study on data deduplication in HPC storage systems," SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 2012, pp. 1-11, doi: 10.1109/SC.2012.14.
[9] "TimescaleDB vs. InfluxDB: Purpose-built for time-series data," Timescale Blog, Aug. 03, 2020. https://www.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/
[10] "Timescale Docs," docs.timescale.com. https://docs.timescale.com/