www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 6 Issue 10 October 2017, Page No. 22806-22812
Index Copernicus value (2015): 58.10 DOI: 10.18535/ijecs/v6i10.17

Optimizing Processing of Multiple Queries

Shilpa M. Hanchinal[1], Rashmi R. Rachh[2]

[1]Visvesvaraya Technological University, Department of Computer Science and Engineering, Jnana Sangam, Belagavi, India
[2] Visvesvaraya Technological University, Department of Computer Science and Engineering, Jnana Sangam, Belagavi, India

Abstract:
Many modern applications generate data streams which are time-ordered, continuous, and unbounded in nature and are generated at a rapid rate. Processing these streams requires more memory, the results need to be generated faster and the incoming data streams can only be viewed once. Traditional algorithms cannot be used for mining streaming data. Therefore, in this project optimizing processing of multiple queries (OPMQ) framework is being implemented for simultaneously executing multiple queries according to the inherent commonalities within. The input data set is converted into different clusters so that the accessing time of the data is reduced thereby improving the performance of the system. The final executable clusters are generated using the k-means algorithm which uses Euclidean distance as the heuristic to find the nearest cluster centroid.

Keywords: Data streams processing, Optimization, Multiple queries

1. Introduction

The awaited arrival of social networking sites, “Internet of Things” (IoT) and many other recent technological advancements have led to data inundate. The incomprehensibly bigger data volumes and new resources are known as Big Data. Gathering and storing huge information for analysis is ages old. Big data has been in use since 1990s and it is a collection of large datasets which are so complex that traditional computing techniques cannot be used to analyze the data. Nowadays, the quantity of data being created and stored is almost inconceivable and still it keeps growing. Data sets are growing apace because they are progressively collected by inexpensive and many information detecting IoT devices [1]. It includes information from various gadgets and applications. Enormous information has gigantic volume, high speed and comprises of extensible assortment of information. But only a small amount of this huge data is useful for analyzing. Big data may be structured, unstructured or semi-structured [2].

Big data has benefits in the field of marketing, medical services, production, education, government, banking and retail. It faces many challenges such as capturing the huge volume of data, organizing and storing it, searching through and sharing it, transferring, updating, information privacy, analysis and presentation of the huge volume [3].

Many modern applications generate data streams. A data stream is time-ordered, continuous, unbounded data generated at a rapid rate. Therefore, processing these streams poses the following challenges: a data stream can be observed only once, limited memory space and the processing results should be generated faster [4]. Hence traditional algorithms cannot be used for mining streaming data. Data streaming and processing of the continuous data streams is playing a vital role in business and scientific applications. These kinds of applications require the data to be transmitted to/from the distributed sources efficiently.

The data sources for the streaming applications may be large simulators or observatories which generate megabytes of data per second and the data aggregated per day amounts to terabytes. This high volume of data is then transferred to the resource constrained remote processing nodes. Due to the storage constraint of these nodes and the huge amount of data to be processed, it is not a good practice to store all the data in the respective nodes before processing it. Therefore, the data streams need to be processed as and when they arrive i.e., in a real time manner. The data which is not required for further analysis is deleted in order to make space for the new incoming data streams which induces a strong bind between the networking, storage, and computing resources [5].

Data stream processing systems are used to handle data bursts, with high speed, by scaling in and out dynamically. The challenges put forth by these systems for auto-scaling techniques are unexpected workload, fast changing workload, and decision making for individual hosts [6]. The purpose of query optimization is to find a way to process a given query in minimal time [7].

With enhanced database capabilities, a single query may include more than one actual query to be executed on the database. These multiple queries share commonalities. The purpose of multi query optimization is to generate an optimal combined evaluation plan for the multiple queries by exploiting their commonalities and also to reduce the system resources required to execute a query. Query optimization enables the user to get faster result set. It also enables the server to run more efficiently i.e., consume less power, use less memory etc., by reducing the wear on the hardware [8].

The rest of the report is organized as follows; the related work of the multiple queries optimization is discussed in chapter 2. Chapter 3 presents the design aspects and requirements of the intended optimizing processing of multiple queries framework. Chapter 4 provides the implementation details of the Optimizing Processing of Multiple Queries (OPMQ) model. Chapter 5 contains the snapshots representing the results.

2. Literature review

This chapter provides a survey on the existing systems for multiple queries optimization.

T. Heinze et al. [6] have used a set of four requirements, namely, workload independence, adaptivity, configurability, computational feasibility to select the available auto-scaling methods to be used in elastic data streaming system. The look-up table is created when a host is allocated and it is deleted when the host is released. The reinforcement learning is modified to take feedback as a negative penalty to reduce the reward of the action taken. The prior decisions are learnt using the utilization after the grace period.

J. Cao et al. [9] have implemented a virtualized environment for data streaming applications in order to avoid the limitations of redundancy. A dynamic resource control method consisting of fuzzy logic controller and iterative bandwidth allocation is used for co-scheduling and co-allocating the networking and computing resources.

Fatma Mohamed et al. [10] have proposed an optimized query mesh for data streams processing. In this mesh framework, data streams are processed over multiple query plans. Each query plan is suitable for a subset of data with the identical characteristics.

L. Ding et al. [11] have implemented a semantic query optimization (SQO) method for dynamically exploiting the metadata of the incoming substream in order to determine the best query plan. A load distribution strategy robust (RLD) to data stream variations has been proposed by C. Lei et al. [14]. RLD exhibits optimal performance under load fluctuations.

A. Dou et al. [12] have proposed a suite of algorithms and index structures that support different historical online queries on flash-equipped sensor devices: pattern matching queries, temporally constrained aggregate queries-aggregate queries with time restrictions and historical online sampling queries.

H. Gyu Kim [13] has implemented a different hash table organization. A hash table is allocated for a set of incoming data stream tuples arriving for a window slide interval, instead of the stream source. Tao Chen et al. [14] have discussed different cases of sharing amongst the multiple top-k queries. This sharing is based on the maximum frequency of each top-k query.

2.1 Issues in existing systems

The issues identified in the existing data stream processing systems are that these systems depend on a single plan for executing continuous, unbounded data which is not reliable with the changing data streams. Multiple continuous queries are executed separately without exploiting the commonalities shared by them ultimately resulting in poor performance in terms of access time. Non-optimized environments are used for data streams processing.

2.2 Problem statement

This project aims at providing an integrated solution for the problems identified in the existing data streams processing systems through the Optimizing Processing of Multiple Queries (OPMQ) model. This model generates multiple query plans for each query in the multiple queries. It executes multiple continuous queries simultaneously by exploiting the commonalities between them. Hadoop mapreduce programming model is used for efficient utilization of the available resources.

3. System Configuration and Design

3.1 System architecture

Fig. 3.1 shows the architecture diagram of the Optimizing Processing of Multiple Queries (OPMQ) framework. It consists of two phases.

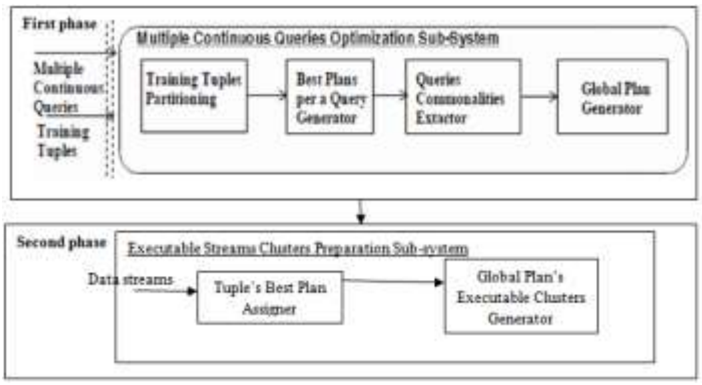

First phase
Multiple Continuous Queries Optimization Sub-System
Multiple Continuous Queries
Training Tuples
Training Tuples Partitioning
Best Plans per a Query Generator
Queries Commonalities Extactor
Global Plan Generator
Second phase
Executable Streams Clusters Preparation Sub-system
Data streams
Tuple's Best Plan Assigner
Global Plan's Executable Clusters Generator

Fig. 3.1 Optimizing Processing of Multiple Queries (OPMQ) framework

The multiple continuous queries optimization represents the first phase. In this phase, a set of training data and many continuous queries are used as inputs to generate the best segments of the training data and a global query plan for the input queries. This phase includes four blocks. They are:

Training tuples partitioning:
It first produces the initial clusters of the training data. It generates query plan for each training tuple and groups them according to their query plans resulting in the initial clusters. Each initial cluster consists of the tuples with the same query plan. It then generates the best clusters using the initial clusters where each best cluster includes tuples with the nearest properties.

Best plans per a query generator:
It yields best multiple plans for each query in the continuous queries. One plan among the multiple plans is the best strategy for one cluster in the best clusters and also the most desirable one for all the tuples in that cluster.

Queries commonalities extractor:
It extracts commonalities or common sub-queries among all the plans generated for the multiple continuous queries.

Global plan generator:
It uses the observed commonalities of all the plans to produce the global plan. And this global plan can be used to execute multiple queries simultaneously.

Executable streams clusters preparation sub-system represents the second phase. In this phase, a sub-global plan is assigned for each incoming tuple and then these tuples are grouped together according to their assigned sub-global plan. This phase consists of two blocks. They are:

Tuple's best plans assigner:
A nearest best cluster from the best clusters is assigned for each of the incoming tuple where the center of the assigned cluster is the nearest center to this tuple. Each plan in the nearest cluster is the most suitable plan for one of the multiple queries. Therefore, each tuple is assigned its most suited plan using the plans in the nearest cluster.

Global plan's executable clusters generation:
It generates the final clusters of the incoming tuples according to the sub-global plans assigned, in order to execute the global plan of all the multiple continuous queries.

3.2 System requirements

The minimum system requirements for the OPMQ framework are: a Core I3 processor with minimum 2.2GHZ, a minimum of 4GB RAM and 10GB memory, Hadoop file system, Java JDK 1.6 or above, Linux (Ubuntu, Redhat, Fedora) operating system.

3.3 Counters used in Hadoop MapReduce

A counter in MapReduce is utilized for gathering statistical data about each of the MapReduce jobs. These counters are characterized in a program (Map or Reduce). Some of the Hadoop counters are:

MAP_INPUT_RECORDS: It represents the total input records used by all the map tasks in the jobs.

MAP_INPUT_BYTES: It stores the total bytes of decompressed inputs consumed by all map tasks in the job. It is increased every time a record is read.

MAP_OUTPUT_MATERIALIZED_BYTES: It represents the total map output bytes written to disk (when compression is enabled).

SPILLED_RECORDS: The total records spilled to disk by the map and reduce tasks in a job.

MAP_OUTPUT_BYTES**:** Total number of records produced by all the maps in a job.

CPU_MILLISECONDS: Represents cumulative CPU time for all tasks (ms).

SPLIT_RAW_BYTES: Amount of data consumed for metadata representation during splits.

REDUCE_INPUT_RECORDS: It represents the total input records of all the reducers in the job.

REDUCE_INPUT_GROUPS: Total number of unique keys. Represents the discrete keys processed by all reducers.

COMBINE_OUTPUT_RECORDS: It stores the total number of records generated by combiners.

PHYSICAL_MEMORY_BYTES: Total physical memory used by all tasks including spilled data (bytes).

REDUCE_OUTPUT_RECORDS**:** Total number of records returned by all reducers.

VIRTUAL_MEMORY_BYTES: Total virtual memory used by all tasks.

MAP_OUTPUT_RECORDS: Total number of outputs generated by all mappers and is updated when record is passed to output collector.

FILE_BYTES_READ: Amount of data read from local storage.

HDFS_BYTES_READ: Amount of data read from HDFS.

FILE_BYTES_WRITTEN: Amount of data written to local storage.

HDFS_BYTES_WRITTEN: Amount of data written to HDFS.

SLOTS_MILLIS_MAPS: It gives total time consumed by all map tasks in occupied slots (ms).

SLOTS_MILLIS_REDUCES: It stores the time spent by all reduce tasks in occupied slots (ms).:

BYTES_READ: Amount of data read by every tasks for every file system.

BYTES_WRITTEN: Amount of data written by every tasks for every file system.

4. Implementation

The optimization of processing of multiple queries is implemented using Hadoop, MapReduce, and K-means algorithm for the sample input data. Hadoop distributes the large data set across many different servers which are cost effective and can be executed in parallel.

K-means clustering is a data-partitioning algorithm that iteratively assigns the input observations to exactly one of the k clusters defined initially. The algorithm outputs k clusters of the input data points and the centroids of these k clusters. The centroids can then be used for labeling the new data. Each centroid is a collection of feature values which define the resulting groups.

K-means clustering takes as input the data set and k-the number of clusters. The data set is a collection of data points. The algorithm starts with the initial values for the k centroids and then iteratively refines these values to obtain the final centroid values. The initial centroids may be generated randomly or selected from the data set randomly. K-means iterates between the two steps:

1) Assignment

Each cluster is defined by its centroid. Based on the Euclidean distance, each data point is allocated to its nearest centroid. A centroid with minimum Euclidean distance is the nearest centroid.

2) Update

This step includes refinement of the centroid values. It is done by taking the mean of all the data points allocated to the centroids.

The algorithm recomputes the centroid values by iterating between the above two steps till a terminating condition is reached. The stopping measure is the minimized sum of the distances. It reduces the entire input space into disjoint sub-spaces.

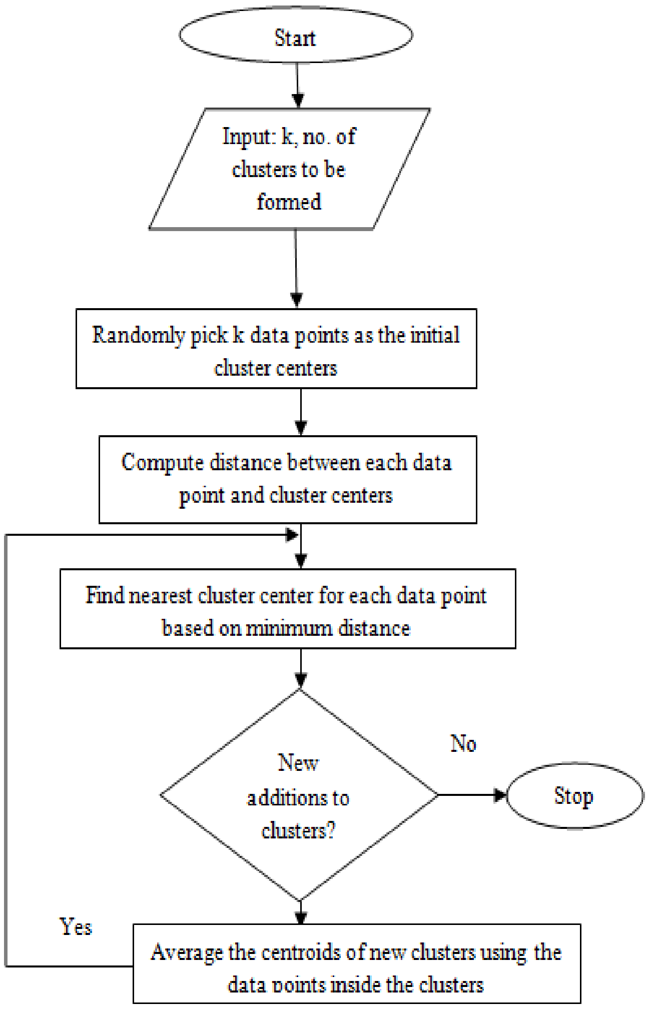

Start
Input: k, no. of clusters to be formed
Randomly pick k data points as the initial cluster centers
Compute distance between each data point and cluster centers
Find nearest cluster center for each data point based on minimum distance
New additions to clusters?
No
Stop
Yes
Average the centroids of new clusters using the data points inside the clusters

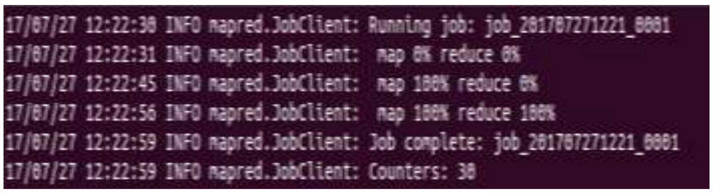
Fig. 4.1 Flow chart of K-means algorithm

The algorithm starts by randomly selecting k. The sample input data which is in the text format is first copied from the local storage to the distributed file system using Hadoop. An output directory is created to store the results. The map function computes the Euclidean distance between the data points and the cluster centroids. The combine function groups the data points with identical cluster id. Reduce function groups all the results together.

The k-means algorithm divides the input data set into different categories and then these categories are passed on to the Hadoop. And then Hadoop mapreduce is used to convert these categories into clusters where each category is responsible for maintaining the respective category that the k-means algorithm optimized. The final clusters can then be used to segregate the incoming data into different categories so that accessing particular data from the big data becomes easy by only referring to the respective category instead of searching the entire input space. Data access time can be reduced by clustering which leads to enhanced performance of the system. An execution plan for a job consists of the following: the map output key for each job, partitioning of the jobs into distinct groups and a technique for processing the jobs in each group.

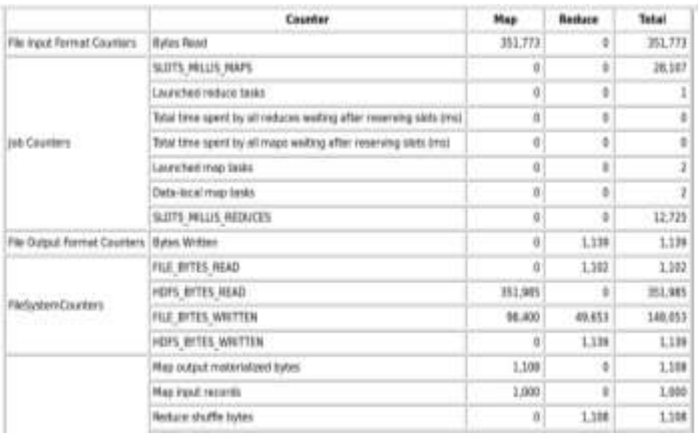
5. Results

This chapter contains the results of the presented optimizing processing of multiple queries model in the form of snapshots.



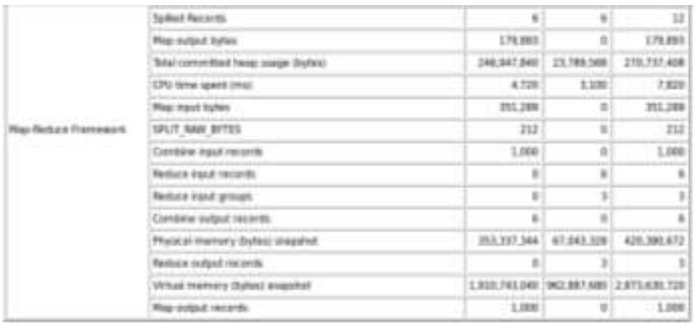
```
17/07/27 12:22:30 INFO mapred.JobClient: Running job: job_201707271221_0001
17/07/27 12:22:31 INFO mapred.JobClient:  map 0% reduce 0%
17/07/27 12:22:45 INFO mapred.JobClient:  map 100% reduce 0%
17/07/27 12:22:56 INFO mapred.JobClient:  map 100% reduce 100%
17/07/27 12:22:59 INFO mapred.JobClient: Job complete: job_201707271221_0001
17/07/27 12:22:59 INFO mapred.JobClient: Counters: 30
```

Fig. 5.1 Status of job one tasks



	Counter	Map	Reduce	Total
File Input Format Counters	Bytes Read	351,773	0	351,773
Job Counters	SLOTS_MILLIS_MAPS	0	0	28,107
	Launched reduce tasks	0	0	1
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Launched map tasks	0	0	2
	Data-local map tasks	0	0	2
	SLOTS_MILLIS_REDUCES	0	0	12,725
File Output Format Counters	Bytes Written	0	1,139	1,139
FileSystemCounters	FILE_BYTES_READ	0	1,102	1,102
	HDFS_BYTES_READ	351,985	0	351,985
	FILE_BYTES_WRITTEN	98,400	49,653	148,053
	HDFS_BYTES_WRITTEN	0	1,139	1,139
	Map output materialized bytes	1,108	0	1,108
	Map input records	1,000	0	1,000
	Reduce shuffle bytes	0	1,108	1,108

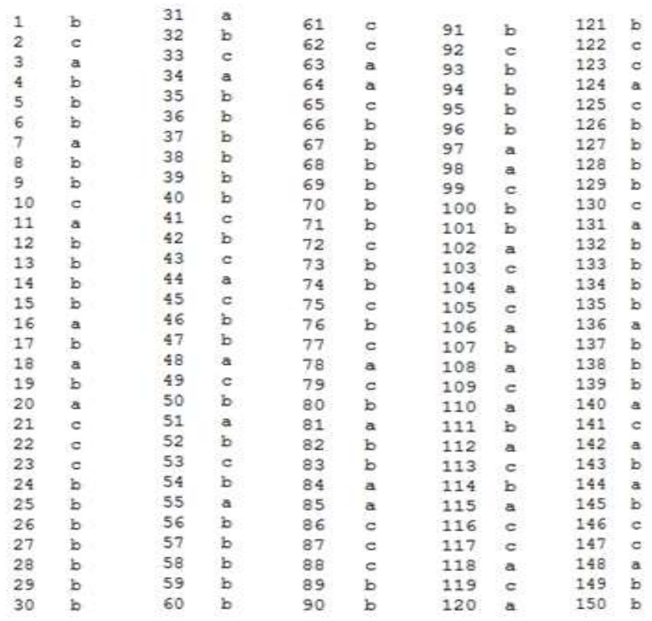
Fig. 5.2(a) Counters used by the first job



Map-Reduce Framework	Spilled Records	6	6	12
	Map output bytes	179,893	0	179,893
	Total committed heap usage (bytes)	246,947,840	23,789,568	270,737,408
	CPU time spent (ms)	4,720	3,100	7,820
	Map input bytes	351,288	0	351,288
	SPLIT_RAW_BYTES	212	0	212
	Combine input records	1,000	0	1,000
	Reduce input records	0	6	6
	Reduce input groups	0	3	3
	Combine output records	6	0	6
	Physical memory (bytes) snapshot	353,337,344	67,043,328	420,380,672
	Reduce output records	0	3	3
	Virtual memory (bytes) snapshot	1,910,743,040	962,887,680	2,873,630,720
	Map output records	1,000	0	1,000

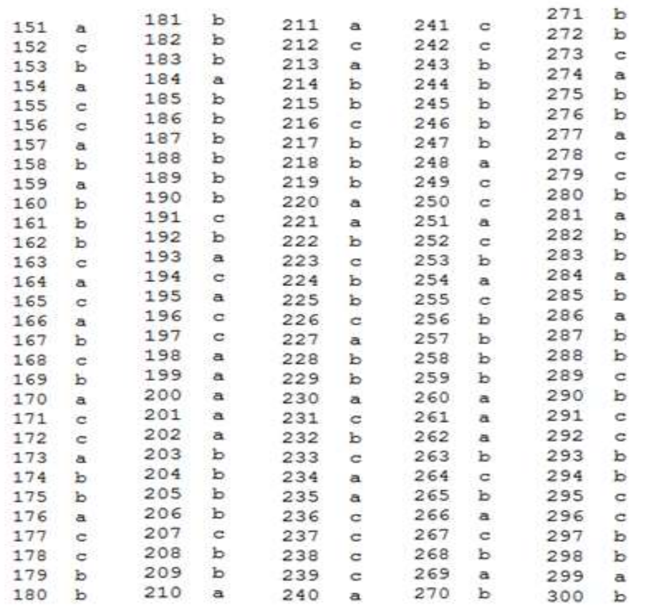
Fig. 5.2(b) Counters used by the first job

DOI: 10.18535/ijecs/v6i10.17



```
1    b      31   a      61   c      91   b      121  b
2    c      32   b      62   c      92   c      122  c
3    a      33   c      63   a      93   b      123  c
4    b      34   a      64   a      94   b      124  a
5    b      35   b      65   c      95   b      125  c
6    b      36   b      66   b      96   b      126  b
7    a      37   b      67   b      97   a      127  b
8    b      38   b      68   b      98   a      128  b
9    b      39   b      69   b      99   c      129  b
10   c      40   b      70   b      100  b      130  c
11   a      41   c      71   b      101  b      131  a
12   b      42   b      72   c      102  a      132  b
13   b      43   c      73   b      103  c      133  b
14   b      44   a      74   b      104  a      134  b
15   b      45   c      75   c      105  c      135  b
16   a      46   b      76   b      106  a      136  a
17   b      47   b      77   c      107  b      137  b
18   a      48   a      78   a      108  a      138  b
19   b      49   c      79   c      109  c      139  b
20   a      50   b      80   b      110  a      140  a
21   c      51   a      81   a      111  b      141  c
22   c      52   b      82   b      112  a      142  a
23   c      53   c      83   b      113  c      143  b
24   b      54   b      84   a      114  b      144  a
25   b      55   a      85   a      115  a      145  b
26   b      56   b      86   c      116  c      146  c
27   b      57   b      87   c      117  c      147  c
28   b      58   b      88   c      118  a      148  a
29   b      59   b      89   b      119  c      149  b
30   b      60   b      90   b      120  a      150  b
```

Fig. 5.3(a) Output showing the examples and associated clusters



```
151  a      181  b      211  a      241  c      271  b
152  c      182  b      212  c      242  c      272  b
153  b      183  b      213  a      243  b      273  c
154  a      184  a      214  b      244  b      274  a
155  c      185  b      215  b      245  b      275  b
156  c      186  b      216  c      246  b      276  b
157  a      187  b      217  b      247  b      277  a
158  b      188  b      218  b      248  a      278  c
159  a      189  b      219  b      249  c      279  c
160  b      190  b      220  a      250  c      280  b
161  b      191  c      221  a      251  a      281  a
162  b      192  b      222  b      252  c      282  b
163  c      193  a      223  c      253  b      283  b
164  a      194  c      224  b      254  a      284  a
165  c      195  a      225  b      255  c      285  b
166  a      196  c      226  c      256  b      286  a
167  b      197  c      227  a      257  b      287  b
168  c      198  a      228  b      258  b      288  b
169  b      199  a      229  b      259  b      289  c
170  a      200  a      230  a      260  a      290  b
171  c      201  a      231  c      261  a      291  c
172  c      202  a      232  b      262  a      292  c
173  a      203  b      233  c      263  b      293  b
174  b      204  b      234  a      264  c      294  b
175  b      205  b      235  a      265  b      295  c
176  a      206  b      236  c      266  a      296  c
177  c      207  c      237  c      267  c      297  b
178  c      208  b      238  c      268  b      298  b
179  b      209  b      239  c      269  a      299  a
180  b      210  a      240  a      270  b      300  b
```

Fig. 5.3(b) Output showing the examples and associated clusters

Conclusion

In this project, optimizing processing of multiple queries (OPMQ) framework has been implemented for simultaneously executing multiple queries according to the inherent commonalities within. The input data set is converted into different clusters so that the accessing time of the data is reduced thereby improving the performance of the system. The final executable clusters are generated using the k-means algorithm which uses Euclidean distance as the heuristic to find the nearest cluster centroid. In future, this framework can be integrated with website applications to provide real-time services for multiple queries optimization.

References

1. *Dedic, N.; Stanier, C. (2017). "Towards Differentiating Business Intelligence, Big Data, Data Analytics and Knowledge Discovery". 285. Berlin; Heidelberg: Springer International Publishing*. ISSN 1865-1356. OCLC 909580101.
2. Snijders, C.; Matzat, U.; Reips, U.-D. "'Big Data': Big gaps of knowledge in the field of Internet". *International Journal of Internet Science*, 2012.
3. Fatma M. Najib, Rasha M. Ismail, et. al, “Multiple Queries Optimization for Data Streams on Cloud Computing”, *in proceedings of IEEE conference*, pp. 28-33, 2015.
4. J. Cao, W. Zhang and W. Tan, “Dynamic control of data streaming and processing in a virtualized environment”, *IEEE Transactions on Automation Science and Engineering*, vol.9, pp. 365 – 376, 2012.
5. T. Heinze, V. Pappalardo, Z. Jerzak and C.Fetzer, “Auto-scaling techniques for elastic data stream processing”, *in proceedings of ACM International Conference on Distributed Event-Based Systems*, pp.318-321, 2014.
6. https://en.wikipedia.org/wiki/Query_optimization
7. http://www.geeksforgeeks.org/query-optimization
8. J. Cao, W. Zhang and W. Tan, “Dynamic control of data streaming and processing in a virtualized environment”, *IEEE Transactions on Automation Science and Engineering*, vol.9, pp. 365 – 376, 2012.
9. Fatma Mohamed, Rasha Ismail, Nagwa Badr, Mohamed Fahmy Tolba, “Efficient optimized query mesh for data streams”, *in Proceedings of the 9th IEEE International Conference on Computer Engineering & Systems (ICCES)*, pp. 157 – 163, 2014.
10. L. Ding, K. Works and E. A. Rundensteine, “Semantic stream query optimization exploiting dynamic metadata”, *in Proceedings of IEEE Conference on Data Engineering (ICDE)*, pp. 111 – 122, 2011.
11. A. Dou, S. Lin, V. Kalogerak and D. Gunopulos, “Supporting historic queries in sensor networks with flash storage”, *Journal of Information Systems*, vol. 39, pp. 217-232, 2014. https://en.wikipedia.org/wiki/Big_data

12. H. Gyu Kim, “A Structure for sliding window equijoins in data stream processing”, *in Proceedings of IEEE International Conference on Computational Science and Engineering (CSE)*, pp. 100 – 103, 2013.
13. Tao Chen, Lei Chen , M.Tamer O¨zsu, Fellow, Nong Xiao, “Optimizing multi-top-k Queries over uncertain data streams”, *IEEE Transactions on Knowledge and Data Engineering (TKDE)* , vol. 25, no. 8, 20