DCF: An Efficient Data Stream Clustering Framework for Streaming Applications

Kyungmin Cho, Sungjae Jo, Hyukjae Jang, Su Myeon Kim, and Junehwa Song

Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
{kmcho, sjjo, hjjang, smkim, junesong}@nclab.kaist.ac.kr

Abstract. Streaming applications, such as environment monitoring and vehicle location tracking require handling high volumes of continuously arriving data and sudden fluctuations in these volumes while efficiently supporting multi-dimensional historical queries. The use of the traditional database management systems is inappropriate because they require excessive number of disk I/O in continuously updating massive data streams. In this paper, we propose DCF (Data Stream Clustering Framework), a novel framework that supports efficient data stream archiving for streaming applications. DCF can reduce a great amount of disk I/O in the storage system by grouping incoming data into clusters and storing them instead of raw data elements. In addition, even when there is a temporary fluctuation in the amount of incoming data, it can stably support storing all incoming raw data by controlling the cluster size. Our experimental results show that our approach significantly reduces the number of disk accesses in terms of both inserting and retrieving data.

Keywords: Data Archiving, OLAP, Clustering, R-tree, Fast Insertion, Query Performance.

1 Introduction

Rapid and continued advances in sensor and wireless communication technologies have fueled a new type of application called streaming applications [16] such as habitat and environment monitoring, RFID-enabled supply chain networks, vehicle location tracking, and transaction log analysis. Such applications have different workload characteristics from traditional applications. Extremely high volumes of data are continuously generated from a lot of data sources. These data need to be stored in permanent storage systems in order to apply analysis tools such as online analytical processing (OLAP) and data mining. These analytical operations are complex and require quite high processing costs. Additionally, in some special situations such as a forest fire, the application monitoring those events must face a sudden rise in data updates.

In such streaming applications, the high costs of disk accesses overload the storage system. Processing a high volume of continuous data requires numerous disk accesses in the storage system in order to write new incoming data into disk and update the index structure. Processing retrieval queries also causes many disk accesses to look up the index structure and retrieve the corresponding data from disk. In the case of

S. Bressan, J. Küng, and R. Wagner (Eds.): DEXA 2006, LNCS 4080, pp. 114 – 122, 2006.

temporary load peaks, the situation would become much more severe. Considering these challenges, for streaming applications, exploiting traditional data management systems, which are designed to support applications over static data sets, is inappropriate.

In this paper, we propose DCF (Data Stream Clustering Framework), a novel framework that supports efficient data stream archiving for streaming applications. It can handle high rates of data insertion and adapt to sudden spikes in the input rate while not degrading retrieval performance. DCF can reduce a great amount of disk I/O for both index updates and look-ups. The key idea of DCF is adopting the scheme of *cluster indexing*, in which the storage system stores data in units of clusters and leaf nodes in the index structure point to clusters instead of individual data. In the case of an insertion, a set of data is grouped into a cluster based on a clustering policy and inserted the cluster via a single insertion operation. Hence, the total number of insertion operations, each of which causes multiple disk accesses, can be considerably reduced. In addition, DCF constructs the resulting index structure with fewer indexing nodes, thereby reducing the index lookup time. DCF also provides the ability to adapt to load fluctuations by monitoring the rate of incoming data and controlling cluster size.

The cluster indexing scheme could cause two additional overheads in retrieving data from disk, since the retrieval operation returns data in units of clusters, not as individual data items. Depending on the cluster size, a cluster could occupy more than one disk block and thus may require multiple disk block accesses to get a cluster from disk. Thus, although the larger cluster size yields the better insertion performance, the retrieval performance is decreased as the cluster size increases. However, typically retrieving a data item from disk requires at least one disk block access. A good compromise is confining the cluster size less than one block, thereby avoiding this overhead. There is another overhead in filtering out unwanted data from a retrieved cluster, but this computation cost of post-processing can be ignored. Typically, the dominant cost of processing a query is the time that it takes to bring a block from disk into main memory. Once we have fetched the block, the time to scan the entire block is negligible.

The rest of the paper is organized as follows. Section 2 describes the proposed DCF. In Section 3, we discuss the scheme of cluster indexing. Section 4 presents experimental results. Section 5 reviews related work. Finally, Section 6 concludes our work.

2 DCF Framework

This section details how our DCF framework is constructed and how it handles a large number of data streams and queries. As shown in Figure 1, DCF is composed of three components: the Clusterer, Load Monitor, and Query Handler. The back-end storage system is responsible for indexing, storing and retrieving stream clusters.

Our framework processes two types of queries: insertion and retrieval queries. For insertion requests, which are stream data elements, the Clusterer first receives and clusters them according to the clustering policy. The clustering policy is a system parameter set by the system administrator. Then, data elements are stored in the internal buffer and periodically sent to the back-end storage system.

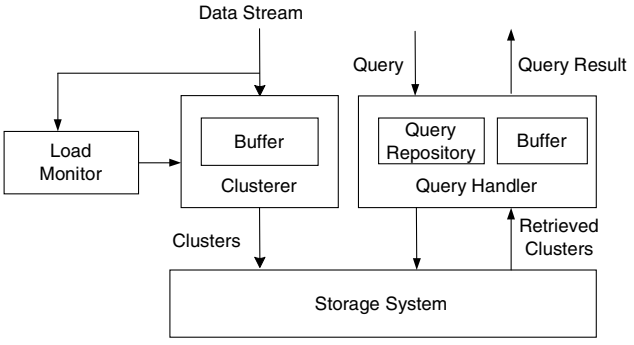


Fig. 1. Overall Architecture of DCF

In the case of a retrieval queries, the Query Handler (QH) temporarily stores the query information for post-processing, then forwards the query to the storage system. The QH has to delay the forwarding of the query until all the corresponding data are processed and stored in the storage system. Since the result is in the form of clusters, the QH needs to do post-processing on the result. Next, we will describe each component in detail.

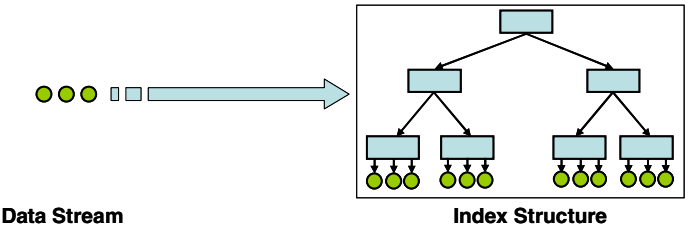
Clusterer
The Clusterer receives data streams from data stream sources. Its primary role is making incoming data into a set of clusters. When receiving a data element, the Clusterer assigns it to the proper cluster, and then loads the data into the buffer. In order to prevent the storage system from being overloaded, the Clusterer should bound the total number of generated clusters to the maximum available update rate that the storage system can hold. The Clusterer receives the data arrival rate from the Load Monitor and uses this information for load adaptation. Depending on the data input rate, more data elements are included in one cluster. Also, note that the Clusterer could use multiple threads to manage multiple requests efficiently.

Load Monitor (LM)
The Load Monitor is in charge of monitoring the data input rate and notifying the Clusterer. Once the input rate of stream data reaches a threshold, the LM warns the Clusterer.

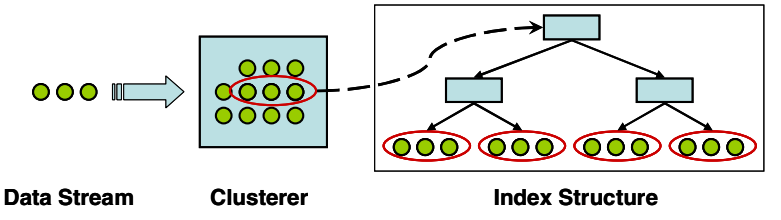
Query Handler (QH)
The Query Handler is mainly responsible for processing retrieval queries. Since the query result is a series of clusters, it could contain a set of data not matching the range of user's query. Thus, the QH unpacks the clusters and filters out unnecessary data elements in order to return the exact query results. For this filtering process, the QH stores and maintains a list of user queries, which are registered in the Query Repository when receiving them. In the context of streaming applications, it is common for a number of users to show similar interests. Thus, with the aid of more intelligent refinement or buffering scheme, the query processing could be further optimized by reusing the pre-fetched clusters.

3 Cluster Indexing

This section describes the difference between *cluster indexing* and *data indexing*, and several advantages of cluster indexing.



(a) Data Indexing



(b) Cluster Indexing

Fig. 2. Brief scheme of data indexing and cluster indexing

Figure 2-(a) shows data indexing, the existing indexing process where leaf nodes in the index structure point to individual data. On the other hand, Figure 2-(b) shows cluster indexing where the storage system stores data in units of clusters grouped by the Clusterer. Leaf nodes in the index structure point to clusters instead of individual data. Hence, cluster indexing leads to an index structure with fewer nodes as compared to data indexing and thus reduce the tree height of the index structure. Given N number of data and fanout f, the height of index structure can be described as $h = \left|\log_f N\right| - 1$. Since cluster indexing reduces the total number of indexed objects N to $N'=N/C$, where C is average cluster size, the height of resulting index structure is likely to be reduced.

Cluster indexing, which has the reduced number of the indexed objects, provides several advantages in both insertion and retrieval operations. It can reduce the number of disk I/O occurred in insertion operations by reducing the frequency of updating index structure. Due to the shortened height of the resulting index structure, the number of nodes visited by a retrieval query is decreased. Cluster indexing is also beneficial in supporting complex queries such as region query or k-nearest-neighbor query. As noted, cluster indexing groups individual data based on its proximity and stores them in one cluster. Thus, for collecting a number of data belonging to the queried area, cluster indexing only need to search a small number of clusters.

We use R-tree (an R-tree [1] or one of its variants) as an index structure for cluster indexing. We assume that the main purpose of data stream archiving is online analytical processing and data mining operations, so the most prevalent query type is the multi-attribute range query. R-tree is the most common index structure for such query type. Also, R-tree doesn't need to be modified for cluster indexing, since R-tree and our cluster indexing scheme use the same data representation type. Every object is abstracted as Minimum Bounding Rectangle (MBR) in R-tree and the Clusterer represent clusters as MBR.

4 Experiment

In this section, we demonstrate the performance benefit of DCF compared with the existing approach. For evaluation, we made DCF prototype in GNU C/C++ and we ran performance comparisons on the Linux platform. For rapid DCF prototyping, existing library package including the R-tree and the storage manager [15] was used for the storage system. The fill factor of the R-tree is set to 40% and the maximum number of entries each node can hold in the R-tree index structure is set to 100.

As an experimental scenario, we suppose that DCF is used for a taxi location tracking application. It receives and manages all the position data of taxis in a downtown area. Our experimental scenario is modeled with the following parameters: (1) the size of the area is 30Km x 30Km, (2) the total number of taxis varies from 1,000 to 8,000, (3) every taxi reports its new position – longitude, altitude, and time - every three seconds, (4) all the taxis move at an average speed of 60Km/h, (5) initially all vehicles are uniformly distributed over the whole area and continue to move to the top-right direction. According to the above scenario, we synthetically generate position data of vehicles using the *"Generate_Spatio_Temporal_Data"* (GSTD) [14], which is a well-known spatiotemporal data generator.

Clustering policies affect the retrieval performance of the index structure. To show this effect, we present three different clustering algorithms, which are a well-known K-means clustering algorithm, a 3D R-tree clustering algorithm which is used for spatiotemporal databases, and a Hash-based clustering algorithm.

The Hash-based clustering is the simplest way to group data streams into clusters. Each data consists of several attributes. Entire attributes' value ranges are divided into equal sized grids in the Hash algorithm. Incoming data stream's attribute values are hashed into a specific grid. Data in the same grid are periodically grouped into a cluster. In this scheme, there is no overlap between clusters.

The K-means clustering [17] is the most popular clustering algorithm. We give parameter values; the number of clusters k, the coarsening value C, and the refining value R, and the flush values *fmin* and *fmax*. Initially, the first k data become a cluster of size one, and the next data elements are assigned to the closest cluster. After finishing the first assignment, the K-means algorithm repeats the calculation of the centroid for each cluster and reassigns all data to the closest cluster.

The 3D R-tree clustering periodically builds a small in-memory 3D R-tree [18] with incoming data stream. Data elements pointed to by the same leaf-node of the in-memory R-tree are grouped as one cluster. Because the tree algorithm is a one-pass algorithm, overlaps between clusters are much larger than other two clustering algorithms.

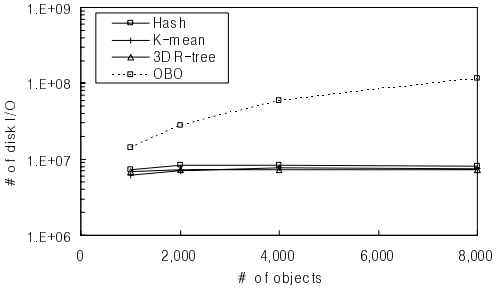


Fig. 3. Insertion performance with varying number of objects

In order to examine the insertion performance of DCF, we measure the total number of disk I/O for updating the index structure. The two hours of position data generated from 1,000, 2,000, 4,000, and 8,000 cars are inserted into the index structure. As shown in Figure 3, in the case of OBO insertion, the total number of disk I/O increases with the number of objects. However, in the case of the clustering schemes, the total number of disk I/O remains stable since our cluster indexing scheme generates equal or fewer clusters than the maximum number of data which can be handled by the storage system, which is assumed to be 200 insertions per second in our experiment.

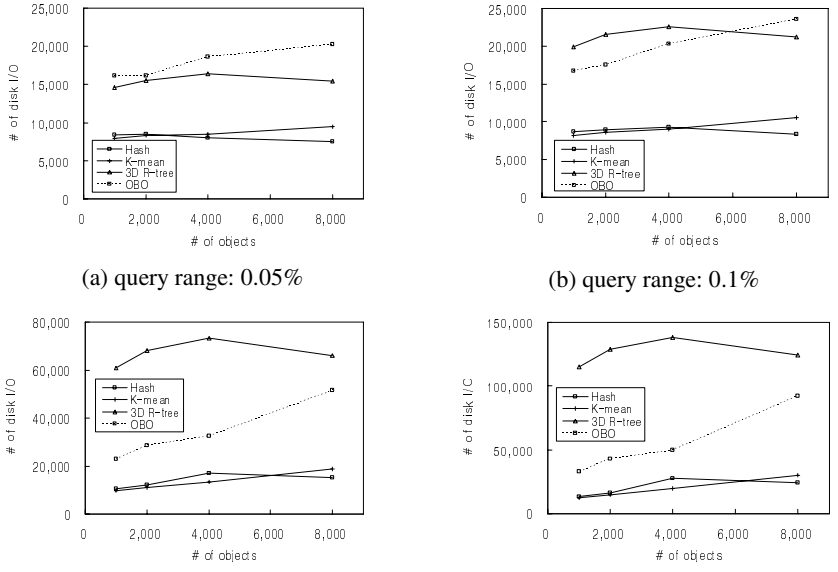
The total number of nodes at each level after finishing the whole insertion process is shown in Table 1. We can see that DCF has a smaller space requirement than OBO insertion.

Table 1. Total number of nodes and height of the resulting R-tree index structure

# of objects	Hash-based		K-mean		3D R-tree		OBO	
	# of nodes	Height	# of nodes	Height	# of nodes	Height	# of nodes	Height
1,000	19,370	4	16,370	4	21,246	4	63,779	4
2,000	22,612	4	19,591	4	22,562	4	120,846	4
4,000	23,168	4	21,553	4	22,774	4	233,901	5
8,000	24,429	4	22,755	4	22,800	4	394,652	5

We also investigate the retrieval performance of our approach. The query type is the window query, which is "find all objects that exists in a certain area during a certain time range." The ranges of queries are 0.05%, 0.1%, 0.5%, and 1% of the total range with respect to each dimension. Each query set includes 1,000 queries. We measure the total number of disk I/O caused by index lookup operations to process each query set when the R-tree is populated with data generated from 1,000, 2,000, 4,000, and 8,000 objects.

As seen in Figure 4, DCF using the Hash-based and the K-mean clustering algorithms outperforms the existing OBO approach. However, DCF using the 3D R-tree clustering algorithm shows that its retrieval performance decreases more severely as the query range increases. From 0.1% of the query range, the 3D R-tree clustering algorithm generates a resulting R-tree having worse retrieval performance than OBO insertion. This indicates that DCF does not always provide better retrieval performance than OBO insertion. We measure the MBR overlap between clusters



(a) query range: 0.05%

(b) query range: 0.1%

(c) query range: 0.5%

(d) query range: 1%

Fig. 4. Retrieval Performance with varying number of objects at different query ranges

Table 2. MBR overlap of clusters generated over the same period

# of objects	Hash-based	K-mean	3D R-tree
1,000	0.00E+00	3.88E-11	1.94E-10
2,000	0.00E+00	5.54E-10	1.23E-08
4,000	0.00E+00	5.56E-09	3.69E-07
8,000	0.00E+00	3.12E-08	1.27E-06

generated over the same period as shown in Table 2. The larger overlap increases the number of nodes to be traversed and results in an inefficient index structure. The clusters generated by the 3D R-tree clustering algorithm show the largest overlap between clusters such that its overlap is almost two orders of magnitude larger. Thus, although cluster indexing generates fewer objects to be indexed and allows the resulting R-tree to be more compact, a poor clustering algorithm like the 3D R-tree clustering algorithm results in an index structure with poor retrieval performance.

5 Related Work

Many researchers have studied indexing overhead reduction, since indexing is the most serious bottleneck in handling large amounts of data. [2][3] try to mitigate indexing overhead by dropping those updates which do not affect the current structure. They are very effective in handling position data of moving objects. However, they are not suitable for data archiving where every update should be

recorded without any loss of data. a bottom-up R-tree [4] speeds up index updates by utilizing locality among incoming data.

Bulk loading approaches [5][6][7][8][9][10] have been proposed to efficiently build multidimensional index structures such as R-trees on massive amounts of data. In these approaches, input data are first sorted according to a certain criteria such as proximity. Then, a number of sorted data are grouped together and the index tree is built upon the data groups. The indexing overhead is reduced by about a factor of the average size of the groups. However, since all the data should be known before using bulk loading, this approach can not be used directly for streaming applications, where data are continuously coming from data sources.

Bulk updating, a.k.a. bulk insertion, [11][12][13] is an approach to efficiently load a bulk of data into an already existing index tree. [11] achieves this goal by creating new index trees on partitions of incoming data, then the index trees are merged with a pre-existing big tree. [12] further improves the performance of index merging by exploiting the characteristics of an existing index tree while constructing a new small index tree. However, there still remains the problem of how to efficiently build the small tree when updates occur very frequently. [13] reduces the frequency of index updates by delaying the propagation of insertions to other tree nodes until a threshold number of data are collected. But this approach requires too much main memory for buffering. Especially, this problem becomes very serious in historical data archiving since the indexing tree grows endlessly. In addition, bulk updating techniques do not address the problem of adapting to load fluctuations, which happens frequently when dealing with stream-based applications.

6 Conclusion

In this paper, we describe DCF, a novel framework that supports data stream archiving for streaming applications. High volumes of continuously arriving data cause a lot of disk I/O, overloading the storage system. The proposed framework, DCF can reduce a great amount of disk I/O in both updating and looking up index structure. The basic idea of DCF is indexing a group of data, namely a cluster, instead of individual data. Our experimental studies show that DCF is very effective in reducing disk I/O resulting from updating index structures, and it is beneficial in reducing the number of disk accesses required to process queries due to the compactness of the index structure.

Possible future works include thorough cost analysis in order to reveal influential factors to retrieval performance, exploration about standard criterion for developing a clustering algorithm, and an adaptive load management mechanism for DCF.

References

1. Antonin Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching, In Proceedings of ACM SIGMOD, pages 47-57, 1984
2. Ouri Wolfson, A. Prasad Sistla, Sam Chamberlain, and Yelena Yesha, Updating and Querying Databases that Track Mobile Units, Special issue on mobile data management and applications of distributed and parallel databases, Vol. 7, Issue 3, July, 1999, pp 257-387

3. Dongseop Kwon, Sangjun Lee, and Sukho Lee, Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree, Proceeding of the Third International Conference on Mobile Data Management, Singapore, January, 2002
4. Mong Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, and Keng Lik Teo, Supporting Frequent Updates in R-Trees: A Bottom-Up Approach, In Proceedings of the 29th VLDB Conferences, Berlin, Germany, pages 608-619, 2003
5. Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos, Fast Subsequence Matching in Time-Series Databases, Proceeding of ACM SIGMOD Conference, Mineapolis, MN, 1994.
6. Ibrahim Kamel, and Christos Faloutsos, On Packing R–trees, Proceedings of the second international conference on Information and Knowledge Management, Washington D.C., US., pp 490-499, 1993
7. D. J. Dewitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server Paradise, In Proceedings of the 20th International Conference on Very Large Data Base (VLDB '94), pages 558-569, Morgan Kaufmann, 1994
8. Kamel, M. Khalil, and V. Kouramajian, Bulk insertion in dynamic R-trees. In Proceedings of the 4th International Symposium on Spatial Data Handling (SDH '96), pages 3B.31-3B.42, 1996
9. S. T. Leutenegger, M.A. Lopez, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. Proceedings of the Thirteenth International Conference on Data Engineering, pages 497-506, 1997
10. N. Roussopoulos and D. Leifker, Direct spatial search on pictorial databases using packed R-trees, In Proceedings ACM-SIGMOD International Conference on Management of Data, SIGMOD Record, Vol 14.4, pages 17-31
11. Li Chen, Rupesh Choubey, and Elke A. Rundensteiner, Bulk-insertions into R-trees using the samll-tree-large-tree approach. In Proceedings of the sixth ACM international symposium on Advances in geographic information systems, pages 161-162, 1998.
12. Taewon Lee, Bongki Moon, and Sukho Lee, Bulk Insertion for R-tree by Seeded Clustering, Proceeding of the DEXA 2003, pp. 129-138
13. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter, Efficient Bulk Operations on Dynamic R-trees. Algorithmica, 33 (1), pages 104-128, 2002
14. Yannis Theodoridis, and Mario A. Nascimento, Generating Spatiotemporal Datasets on the WWW, SIGMOD Record, Vol 29., No 3., pp 39-43, 2000
15. http://www.cs.ucr.edu/~marioh/spatialindex/index.html
16. Lukasz Golab, M. Tamer Ozsu, Data Stream Management Issues – A Survey, Technical Report CS 2003-08, University of Waterloo, April 2003
17. M. R. Anderberg, Probability and Mathematical Statistics, Academic Press, New York, San Francisco, London, 1973
18. M. Vazirgiannis, Y. Theodoridis, and T. Sellis, Spatio-temporal composition and indexing for large multimedia applications, Multimedia Systems, 6(4):284-298, 1998