Cluster Comput (2019) 22:S10503–S10517
https://doi.org/10.1007/s10586-017-1093-z

Efficient data retrieval using adaptive clustered indexing for continuous queries over streaming data

M. R. Sumalatha[1] · **M. Ananthi**[2]

Received: 22 May 2017 / Revised: 27 July 2017 / Accepted: 29 July 2017 / Published online: 31 August 2017

Abstract The Modern era has highly dynamic, heterogeneous and massive data volumes, generated from sensor networks, social media and telecommunications, stock market analyses and the Internet, etc. makes constant query processing quite challenging in processing real-time data, which exist as streams and undergo dynamic changes. Large volumes of data can be efficiently handled by partitioning them into clusters followed by Indexing. An efficient clustering and indexing method is required to process continuous queries for retrieving data streams. A new index structure called adaptive clustering and block-based indexing (ACBBI) is proposed, which is a fusion of cluster-based and block-based techniques to process continuous queries. The incoming data are clustered and stored as blocks using the adaptive clustering method and further indexed by the adaptive indexing approach. Livestock market values that are time variant are used for experimentation. The experimental analysis demonstrates that the ACBBI tree structure significantly decreases half of the space cost, scales better with increasing data size and improves the retrieval rate 30% more than an existing CKDB approach.

Keywords Data stream · Indexing · Query processing · Data management · Clustering · Data retrieval · Continuous queries

✉ M. R. Sumalatha
sumalatha@annauniv.edu

M. Ananthi
ananthi.it@sairam.edu.in

[1] Department of Information Technology, Anna University, Chennai, India

[2] Department of Information Technology, Sri Sairam Engineering College, Chennai, India

1 Introduction

The data of any area are observed dynamic in the modern world and that changes their values in no time difference and these sorts of vibrant and time-variant data are obtained using real-time data streams. Data streams are the database format that is used to record streaming data that holds unbounded data that flows continuously henceforth results in difficult to store and process these bulk data. Concerning the storage and processing a difficulty that requires larger memory space, we designed the adaptive query processor to process the data immediately by formulating queries dynamically and executes to deliver the desired output as soon as data arrives that avoids storing of intermediate results. Data streams record large volume of data that flows into the system. Streaming data continuously change over time. The clustering model requires the development of a partial data stream that is updated with new incoming data. Pairs of points tend to look equidistant from one another because of the sparsity of data in high-dimensional space, which is not viable for controlling the data arrival order or storing all data elements. Both indexing and query processing must use append-only file access. Indexes in data stream management systems (DSMSs) are stored and processed in the main memory itself, since it is cache oriented and compact. Streaming window indexes are essential for adapting the incoming data flow rate to handle high rates of insertion, updation and deletion and records the continuous data with a predefined timing constraint (say 10 s) and clubs the data for processing. Online financial applications, for example, stock market applications, continuously produce stock price values that depend on time. There are an infinite number of stock values that vary over time, producing a huge volume of stock quotes from various companies. The queries have to be executed continuously in a real-time environment to produce information on the livestock data.

Continuous monitoring and recording of stock values are required to produce up-to-date results of continuous queries upon request. Various company details with varying stock prices need to be considered for continuous query execution. The most common information that are users search for in any stock market site or Apps are current stock values, analyzing minimum and maximum share value, gainers and losers on the time/day list, companies that hold top position, and comparing various company's stock values. This information is produced for them by querying on the continuous data that are recorded in our tool. These queries need to be executed daily and the results should be dynamic and time variant, since these are the probable searches of any stock market users.

Data partitioning and indexing are one of the main design considerations for huge volumes of data. Clustering and then indexing the data help in fast retrieval. Clustering is one way to categorize incoming data based on the similarity of sectors. Grouping similar data items performs clustering. Incremental maintenance and updating are required in clustering, which is more expensive than grouping. Indexing infinite streaming data with finite storage space is a challenging task. Well-organized storage is required to handle user queries and filter the index streams efficiently. High-speed query processing needs appropriate indexing and dynamic retrieval of streaming data, which leads to efficient query processing. The research issues in processing continuous queries over data streams are as follows:

- Finding an appropriate indexing approach that can accommodate and process huge volumes of data.
- Addressing the new challenges for query processing associated with streaming databases owing to both the vibrant nature of the data which frequently changes over time and the wider range of queries proposed by the user.
- Maintaining scalability to streaming data of increasing density.

In this proposed work, a suitable indexing approach is proposed to process large volumes of real-time vibrant data. A novel clustering and indexing algorithm is introduced for efficient retrieval of data. A new data index structure called ACBBI is proposed, which aims to address the three main challenges in data indexing: (1) adaptive insertion, (2) quick retrieval, and (3) dynamic omission. The rest of this paper is structured as follows: Related work is discussed in Sect. 2. The proposed stream processor is presented in Sect. 3. Section 4 explains the operations of the ACBBI structure. The performance evaluation is discussed in Sect. 5, and the conclusions and future work are described in Sect. 6. Section 7 lists the papers referenced i this work.

2 Related work

The incremental clustering method for data stream of chunks using supervised learning [29] and post-processing phases. First phase is the learning phase to create clusters adaptively and continue the process until no more clusters created. Outliers are removed in the post-processing phase. Predefined data sets were used in their approach. The above incremental clustering technique is enhanced to process timestamp-based real-time streaming data in this research work. Xie et al. [27] used selectivity method to store the particular sliding window samples and recorded for a predefined window size of 200. However, streaming data require dynamic adaptive window because the window size cannot be predetermined. Omran Saleh Stefan [22] described patterns from continuous incoming data retrieved in (near) real-time. Linked stream data (LSD) was introduced to give these data streams a meaningful structure. Relational operators, windows operators, and source and sink operators are used as data flow operators in PipeFlow. Some tests showed that PipeFlow acted upon existing LSD systems in which some requirements went beyond classic data stream processing. An extremely scalable and elastic stream-processing engine [8] was implemented to detect fraud calls within telephone description records in real time. The stream-processing engine was implemented in shared-nothing clusters using the parallelization approach to attenuate the distribution overhead. The limitation in this work is that even though scalability and elasticity were proven, semantic web-based query processing was not used.

Multi-top-k queries optimized for uncertain data streams are discussed in [5]. It tries to balance additional overhead and save computation time using a faster greedy algorithm and the intermediate results are stored in materialized evaluation. Storing intermediate results could also be a materialized approach which may lack in space and performance. Query indexing [14] was proposed to process continuous queries on RFID streaming data. Aggregating segments and transformation into single object were implemented, specifically designed for RFID tags. All types of tags were not supported in this work. Continuous time-series data were clustered based on predicting trend characteristics [12]. Analyzing the trends of incoming streams and split and conquer techniques lead to performance overhead. ClusTree [9] proposed clustering multiple data streams concurrently. Summaries of multiple concurrent streams are maintained. Maintaining summary statistics of each data object leads to a larger workload. Various clustering methodologies are discussed in [28]. Among them, incremental clustering considers instances, one at a time. The entire dataset is not required to be stored in advance in the main memory, which saves time and space. This clustering method is suitable for processing dynamic incoming data in which the whole dataset is not known in advance. This method is utilized in our proposed work. A

survey of various clustering algorithms [1,10,21,25] for time series datasets is discussed. From this survey, it is realized that effective clustering algorithms are required to process high-dimensional data to increase the processing speed. In Trie indexing [4], multiway tree structure was used in which each node was represented as an array of pointers. A word is considered as a key at each level. The string handling used in this work leads to memory utilization problems, since any new keyword that not part of existing array would be added and making the array grow larger and utilizing space of the system.

Judy [13] uses a compact trie indexing method where node structures change dynamically according to the current distribution of keys. One cache block is used to hold multiple compact node structures. Since, dynamic real-time data streams produce huge volumes of data continuously, managing a single cache block for indexing the streaming data is intricate. Evolving fuzzy-rule based systems and metacognitive learning are discussed in existing systems as an autonomous learning machine to process dynamic data streams [2,17–20]. Evolving systems based on Takagi-Sugeno fuzzy models have been discussed [2]. Pratama [20] initiated a parsimonious classifier called pClass using a fuzzy-rule based classifier. The first-order regression based classifier is used for the extraction of fuzzy rules for streaming data. A feature weighting mechanism is implemented to analyze concept drifts by recalling past data distribution. The multivariable type-2 fuzzy model [18] and type-2 neuro fuzzy class using the incremental clustering method [17] are discussed to automate the machine learning processes for streaming data. Scaffolding Type-2 classifier (ST2) [16] discussed handling big data analytics without prior knowledge of the system using machine learning and meta-cognitive learning processes. Incremental clustering method can be adopted by automating the machine learning process without knowing prior knowledge of the past data. This concept was proposed by Pratama [16–20] and is adopted in the proposed work. It uses the incremental clustering method by dropping out past data and hence failed to have a complete dataset.

Wang et al. [26] described subsequent versions that pointed on each index node to minimize the cost of time delay. Many versions are maintained in the index with the version number; live versions are marked with an asterisk (*). One data version is stored as arrays on each page. The limitation is that the usage of version arrays consumes additional spaces in the main memory. The number of updates is reduced due to version splits, system performance, and effectiveness affected in the flash memory. The arrival rate of update transactions is predefined and is updated every fixed period. It concentrates only on version-range queries, not on exact-match queries. Both exact match queries and version-range queries are considered in the proposed work. The CKDB tree-indexing scheme was introduced by Deng et al. [6] to support dynamic continuous queries by combining the cell-based and KDB tree. Query indexing is performed in the CPU and stream data was filtered by GPU in parallel. Only range queries are considered. Top-k queries and KDB trees are maintained in separate arrays. Any query that overlaps in more than one cell takes more memory space and retrieval time. Various index entries were maintained for cross-boundary queries that lead to more space and maintenance costs. The above space inefficiency has been resolved in the proposed work with the adaptive indexing method. The indexing requirements of big data were discussed by Shoshani [24]. There is a need to retrieve billions or trillions of data values within a second. Multi-variable queries are executed. It has been mentioned that an efficient index generation and a parallel processor are required to speed up the retrieval and execution processes in this work. Research directions of big data are given by Valduriez [15]. It is stated that database cracking, which breaks the databases into manageable pieces could be done by building the index dynamically at the time of query processing. This can be achieved through the column-store method. This method is utilized in the proposed work by using the adaptive clustering method.

The close dominance graph (CDG) index structure described by Santoso and Chiu [23] was implemented to address continuous top-k dominating queries where insertion and deletion are taken simultaneously. Searching time was reduced while processing regular updates in the database. A similar approach is used in the proposed work by eliminating the duplicate entries and inserting only valid data, which change in time and stock price value. Mahnoosh kholghi and Keyvanpour [11] stated that the update time will be longer if the database is distributed in more than one site. They analysed the performance of different indexing techniques and focused on multi-resolution indexing techniques to overcome the problem in sliding window indexing over data streams. Indexing resolution is based on feature extraction and accuracy is provided to user queries by error bounds. This method is used in the proposed adaptive indexing technique. The balanced stream tree [7], called BSTree was developed for searching, finding similarities and monitoring real-time data streams. In this work, the least recently visited pruning technique and symbolic discretization method were used for similarity search queries to minimize the response time and lexicographical order was used to store data, which consumes more space and requires more search time for alphabets.

A comparison of existing indexing techniques is listed in Table 1. In index-structured trees, tries and the advanced Judy implementation of compact tries were discussed in previous work. Some tries were found to be relevant indexing structures that permit constant time insert and access rates. Each node in the trie is a pointer array that pursues a multi-way tree structure, and each level in the tree indexes a letter in

Table 1 Comparison of existing indexing methods

Indexing name	Method	Advantages	Limitations
Tries [4]	Multi-way tree structure	Constant insertion and access time if the key length is fixed. Suitable for high insertion rates	Less memory utilization
Burst tries [4]	Same suffix keys in same containers, Binary search	Efficient memory utilization. Less memory consumption	Fixed size containers. Internal nodes may have null pointers—memory wastage. Not optimal solution
Judy compact trie [13]	Dynamic structure based on current distribution keys. Range search	More and more densely populated. Less cache hit misses	Constant tree depth for all single element operations
BSTree [7] (balanced stream tree)	Similarity search	To minimize response time	Consume more space and more searching time

a word. Although it follows a multi-way tree structure, over utilization of memory is a major problem. The key distribution leads to a memory utilization problem with tries. In the worst case, when the keys are uniformly dispersed across the domain, memory is wasted with naïve tries because many null pointers denote the trie nodes in the sparse pointer arrays. To overcome naïve tries, diverse compression methods were hosted. It is very challenging to index dynamic data, and queries need to be executed continuously with a minimum impediment. Therefore, an indexing method is required for faster access of incoming data, and detecting the update rate of streaming data is an open problem.

3 Proposed work

Dynamic, heterogeneous and large volumes of data are continuously generated from various applications, such as sensor networks, social media, telecommunications, stock market analyses, search engines, network monitoring and so on. An efficient clustering and indexing method is proposed to process these streaming data and continuous queries. Clustering is used to partition the incoming data streams based on some key factors that will make the processing efficiencies instead of handling bulk data. Appropriate indexing approaches are essential to handle fast incoming data and to process continuous flow of queries. Owing to the dynamic nature of incoming flow, adaptive processing is required. So, adaptive clustering and indexing structure is proposed in this work to efficiently process continuous queries.

3.1 ACBBI stream processor

A new index structure is proposed to reduce the cost of space and speed up the retrieval from data storage, which mainly focuses on leaf node indexing where the data are stored. The tree-based indexing structure requires lesser space than the linear structure. Henceforth, tree-based indexing proposed in the system in order to handle proper indexing and efficient retrieval for real-time, time-variant data. ACBBI is proposed to index and retrieve speedy streaming data. The ACBBI stream processor architecture is shown in Fig. 1. Incoming data streams are fragmented according to the key value pairs and grouped together by applying incremental clustering based on the timestamp. Here, clustering is performed adaptively only when there is a variation in the incoming value and timestamp. Each cluster is further divided into blocks. Each block is indexed based on an extended B^+ tree. Next, the query processor for processing continuous queries proceeds are used the indexed data. The stream query processor consists of query indexing, a query plan and query executor, which are used to process continuous queries efficiently. User queries are sent through the application and the results are fed back to the users. The notations used in the ACBBI stream processor are represented in Table 2.

3.2 Adaptive clustering algorithm

Adaptive clustering groups live streaming data into different clusters based on the key value and incremental clustering approach. Incremental clustering envisages instances of one at a time and allocate them to prevailing clusters. In incremental clustering, a complete dataset is not required in advance for storing the data. This mechanism is very much suitable for dynamic data streams. Algorithm 1 describes the adaptive clustering technique. Incoming streaming data are identified by mapping with the key value k (step 2). The key value is the segregation parameter of incoming data. The flag is initialized as false initially and becomes true after cluster is updated (step 3–4). The current timestamp value is set as threshold value to control and identify the live data (step 5). The data are hashed with k to find their corresponding cluster and mapping data are filtered (step 6).In the stock market application, stock values are identified based on the company name as a key. In addition, the companies are grouped together as sectors

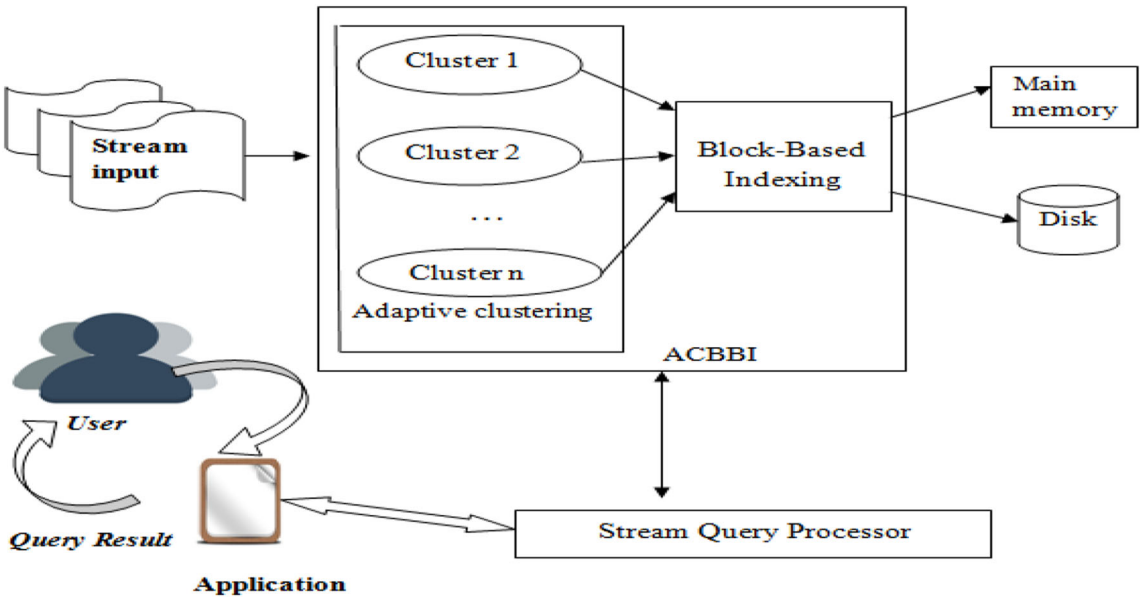


Fig. 1 ACBBI stream processor

Table 2 Notations used

D_s	Data streams
K	Key value
d_k	Filtered mapping of data with key
T_s	Time stamp
Sp	Block starting pointer
B	Block id
#	Live entry
Ni	Number of update index entries
M	Number of data items
Ţ	Threshold timestamp
R_s	Record set
R_l, R_h	Low range and high range value

and the sector name is used as a key value for quick retrieval of company details. Livestock market entries are clustered based on the sector and only the values that vary in terms the stock price are considered. The incoming data are checked for live entries by considering the timestamp as a threshold Ţ (step 8) and a cluster segment is identified through the findStream method. If the data belong to an existing segment, then existing cluster is updated (step 9) and increment the cluster (step 10–12). Otherwise (step 15), a new cluster is formed (step 16) and we set the counter value as 1 (step 17). A new segment is created for a new cluster (step 18) and added to the existing cluster list (step 19). Finally, data with their timestamp are inserted into the corresponding cluster (step 22). Here, data clustering is performed adaptively only when there is a variation in the data and timestamp.

Algorithm 1: AdaptiveClustering(D_s)

Input: Live stream data entries Ds and key k, Threshold Ţ

Output: Clusters grouped in form of segments

```
1   Clusters ← φ
2   Look for live entries in Ds for key K
3   For all Xi ∈ Ds
4               Flag=false
5   Ţ = Ts                          // timestamp
6   dk=findStream(Ds,k)
7   For all Cluster ∈ Clusters do
8     If (Ts(dk) ≥ Ţ and A(dk) ϵ Cluster then
9           Updatecluster(Cluster)
10          Counter(Cluster)++
11          Flag=true
12          Exit loop
13    End if
14  End for
15  If not(flag)  and Ts(dk) ≥ Ţ  then
16          Newcluster=dk
17          Counter(newCluster)=1
18          newSegment←newcluster
19          Clusters←Clusters ∪ newcluster
20   End if
21  End for
22  Clusters.Insertdata(K,Ts)
23  end
```

Clustering is done by grouping the incoming data based on the sector as a key factor. Algorithm 2 describes about sector based clustering for streaming data. findStream algorithm gives an overview about how the incoming stream data are segregated as clusters based on sector. The algorithams two functions: splitting and grouping. The algorithm splits data based on key (sector) and group data that have same keys. (step 4) Sectors are classified by hashing the incoming stream

data with key value and are stored as C_h (clustering with hashing). This classification is done only when there is a change in data (step 1–3). Then, that is split based on C_h (step 5–6). Segregated data are grouped based on similar sectors (step 7) that are identified by checking the similarity among the data entries. Similarity measurement is represented in Algorithm 3. The grouped entries are inserted as blocks (step 9). The live entries are split using keys ($k_1, k_2, \ldots k_n$) through which various sub streams such as ($S_{p1}, S_{p2,\ldots}S_{pn}$) are obtained. Grouping operation is shown in Eqs. 1 thru Eq. 3.

$$G_{spi} = \{D_{i1}, D_{i2}, D_{i3}, \ldots\ldots\ldots, D_{ik}\}. \quad (1)$$

$$G_{spj} = \left\{\underset{n}{D_{j1}}, D_{j2}, D_{j3}, \ldots\ldots\ldots, D_{jk}\right\}. \quad (2)$$

$$D_{spk} = \bigcup_{k=1} G_{spk}. \quad (3)$$

Algorithm 2 findStream (Ds,k)

Input:D_s(Data Stream)
Output:G_{sp} (Group Split)

1 Look for live entries in Ds for key K
2 While (change)
3 C_h←hash (D_s,k)
4 Split $D_s$into sp_1, sp_2,...sp_nbased on C_h
5 D_s←{sp_1, sp_2,...sp_n}
6 G_{sp}=Group each sp_xbased on k
7 end while
8 Copy Grouped entries G_{sp}into datapart

Algorithm 3 Similarity Measurement

Ck-Cluster, xip-attributes.

1. Read values of objects i,j
2. i=(*xi*1,*xi*2, ...,*xip*) and *j* = (*xj*1,*ji*2, ...,*xjp*)
3. Calculate similarity for x and y
4. sim=√(xip-yip)^2
5. if (sim==0)
6. Ck.add(i,j)
7. goto step1
8. else
9. goto step4
10. End if
11. for i=1 to n
12. sim= √(xip-zlp)^2
13. If(sim==0)
14. Ck.add(i,l)
15. read next object
16. else
17. Ck.add(i)
18. End if
19. End for
20. return

Checking for changes in the value and time validity controls incoming streams. If both conditions are satisfied, then data are clustered based on similarity. Grouped entries are stored as blocks in the memory for further indexing. Similarity between streams of data is measured for clustering and is described in Algorithm 3. Two categories are considered here: (i) the same sector with different values and (ii) different sectors with different values. Different values represent changes in the timestamp and data. The same sector with the same value and different sectors with the same value are omitted to reduce repeated data, which leads to memory wastage. Controlling and simultaneously checking incoming data, perform data removal. Readings of *xij* (step 1–2) represents different data streams of various companies of the same sector. Differences in values are denoted as x and y values. The similarity between x and y is measured (step 3–4). Ck signifies clusters. The same sector with different values is added to the same cluster (step 5–6). Different sectors with varying data indicated as x and z are stored in different clusters (step 12–14). A basic similarity checking method is used and calculated as shown in Eqs. 4 and 5.

$$x_i = \{x_{i1}, x_{i2}, x_{i3}, \ldots x_{ip}\} \quad \text{and} \quad y_i = \{y_{i1}, y_{i2}, y_{i3}, \ldots..y_{ip}\} \quad (4)$$

$$sim\left(x_{ip}, y_{ip}\right) = \sqrt{\left(x_{ip} - y_{ip}\right)^2} \quad (5)$$

After clustering, the clustered data are stored as blocks for further indexing using the adaptive indexing method.

4 Adaptive indexing methodology

Adaptive indexing is the method used to index incoming dynamic data, which leads to fast retrieval of data for further processing. Each cluster is divided into equal sized blocks to store the large volume of incoming data. All the blocks are indexed as tuple where each tuple consists of <K,Sp,#, B>,where K is the key value based on clusters. The initial value of each block is represented as Sp and the block number is denoted as B. The # symbol represents the live entries based on the timestamp. Because stream data are embedded with a timestamp, a timestamp number is attached to each entry. Newly arriving data are represented using a # symbol, whereas expired data are considered as meta data. Each block is indexed using B^+ Tree. Each non-leaf node has at most Ni index entries. Each index entry includes a corresponding child node and its pointer. Each leaf node stores a maximum of N data items. The leaf node shares the array list to maintain the data and metadata. The query processor processes, continuous queries by retrieving incoming data using the adaptive indexing method. Insertion becomes efficient and good for accessing useful data by clustering and indexing adaptively. The adaptive indexing approach is shown in Fig. 2.

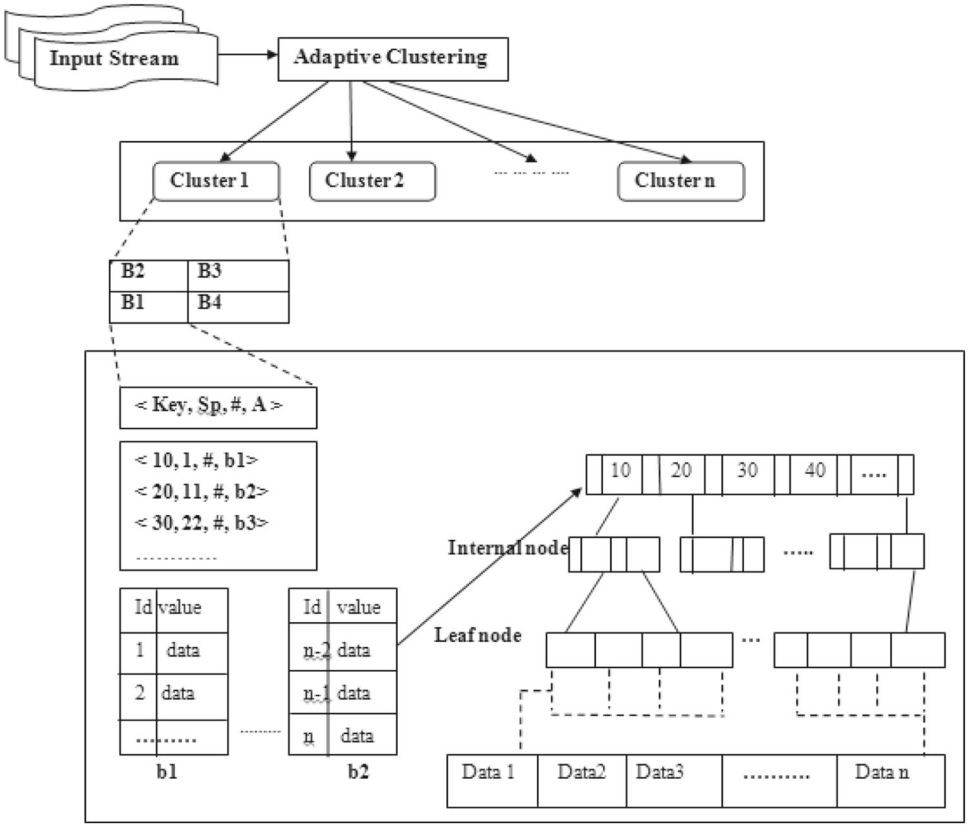


Fig. 2 Adaptive indexing approach

Insertion and Searching are two main issues in handling data items. Inserting data streams and searching data for query execution algorithms are discussed in this section. Dynamic deletion is the removal of duplicate entries and filtering the valid incoming data. The deletion operation is not discussed separately because incoming streams are filtered based on the timestamp and the change in value. Subsequently, repeated data that are not required for further processing are automatically removed and no separate procedure is required for deletion. The insertion phase accepts new data by comparing it with the existing data and key value. Data are inserted only when there is a change in the value and change in the timestamp, as shown in (10).

$$\Delta(T_s) > 0 \quad and \quad \Delta(Price) \neq 0; \quad where \quad T_s > T_{sp} \quad and \quad key \in D_s \tag{6}$$

Differences in the timestamp are represented as $\Delta(T_s)$. and calculated from the present timestampT_s and previous timestamp T_{sp}. Changes in price are calculated from the variation with the previous reading, represented as $\Delta(Price)$. If there is no match found with previous data as in Eq. 6, then the changed data are added into the cluster. Otherwise, new incoming data are omitted. This is to control the duplicate entering of incoming data. Similarly, new updates are accepted by comparing (Ts, Price). If price remains the same and Ts differs, an update is made to the existing cluster, thereby reducing the overall size of the cluster, as shown in (Eq. 7).

$$\Delta(T_s) > 0 \quad and \quad \Delta(Price) = 0; \quad where \quad T_s > T_{sp} \quad and \quad key \in D_s. \tag{7}$$

4.1 Insertion in ACBBI

Insertion of new arrivals is shown in Algorithm 4. Live entries are identified using timestamp Ts. We search an empty block (step 1) and create a leaf node to insert new entries (step 3–4). Next, data are inserted into tree array list AL (step 5). If the timestamp of current data is equal to or within the system timestamp and there is a change in the value (step 7), then we insert current data into array list (step 8); otherwise, we update the timestamp of the existing value as current (Step 10). Previous data that are retrieved until the threshold timestamp Ţ (step 11) are grouped together, as represented in CompNode algorithm (step 11–12).

Algorithm 4 Insert Data

Input: Key K, Timestamp T_s, Value

```
1   Search an empty block with capacity Ce
2   Write value and timestamp in data block
3   Create a leafentry L=<K,Ts,Value>
4   Search leaf node for L and
5   EntryInsert(AL,L)
6   end insert
EntryInsert(AL,L)
Input: Node AL and item.
7   if Ts<=T(cur_data) then
8     Insert cur_data into AL
9   Otherwise
10    Update Ts(AL) = Ts`
11  if AL>Ʈ then
12    CompNode(AL)
13  return
```

The CompNode algorithm provides the simultaneous update of cumulating incoming entries for further searches and is shown in Algorithm 5. Similar data are mapped by hashing with key value (Step 1–2). The timestamp of grouped data is set as T_L, which is the last timestamp value (step 3). Previous entries up to T_L are summed up and stored as N_{sum} (step 4). The average value is calculated for the grouped data and further stored to maintain the metadata (step 5).

Algorithm 5CompNode(A_L)

Input: A_L, Ʈ

Output: compressednode

```
1   Group AL based on K. // K-Key(sector)
2   Sh←Hash (AL, K)
3   Group Sh and set Ts as TL. // Ts and TL are Timestamps.
4   Nsum=Σ(Sh) + TL.
5   compressednode=Avg (Nsum).  // Nsum- Price
6   return compressednode
```

The aggregation is part of a scalable removal operation specified as the CompNode algorithm. This algorithm is used to compute the average of stock prices of a company on a particular day and replace with a single value. This is performed to minimize the overall memory used. After some point of time, some values may not be looked up frequently, so they can be replaced by a single entry that contains the average of stock prices. This is achieved by simultaneously compressing the entries using the CompNode algorithm. The change in value and average of the threshold timestamp can only be stored in the memory, which saves memory and reduces the number of reads/writes. Simultaneously, past data of expired timestamp are aggregated and stored as metadata to retrieve historical based query. Aggregated data are removed from the current index in the main memory, which is not required to process continuous real-time queries.

4.2 Search algorithm on ACBB

Searching of data is explained in Algorithm 6. Data are searched in the ACBT(Adaptive Clustering Based Tree). Three cases of retrieval analyzed. Timestamp based search, key based search and range based search in which timestamp and key based are considered as exact match query search (step 2–10); range based search is considered as range query and shown in (step 13–17). Timestamp of incoming data and its array list are checked for live entries as presented in (step 23–27). Live data are retrieved based on the current timestamp, as represented in steps 3–4. If a timestamp is used as one of the key values and searching data are another key (Step 6), then this represents an exact match query. First, the cluster is identified based on key values (step 8). Then, the corresponding block is identified based on the search data (step 9). Live entries from the block are searched with key values to find the data (Step 10). The query range is specified as R_l(low range value) and R_h(high range value), which are used as key values in the search. Live entries are searched based on these range values in the corresponding block and cluster (step 14–17). The search of live data from the list is shown in steps 22–27.

Algorithm 6 Search on ACBBI Tree

Searchproc(ACBT, V(D_s),R_s)

```
1   for each datarecordr∈V(Ds) do
2   Case 1: if(key == Ts) then                    /* Timestamp based Query */
3   Block= findblock(r,ACBT)
4   searchlivelist(Block.E(K,V),r,Rs,key)
5   End if
6   Case 2: if(key1==Ts&& key2)       /* Exact match query */
7   for each data tuple r∈V(Ds) do
8   sector=findSector(Key2,ABT)
9           Block=findBlock(r,sector)
10                searchlivelist(Block.E(K,V),r,Rs,key2)
11          End for
12          End if
13  Case 3: if(key≥Rl&&key≤Rh)       /* range query */
14  for each data tuple , r∈V(Ds) do
15  sector=findSector(key,ABT)
16  block=findBlock(r,sector)
17  searchlivelist(Block.E(k,v),r,Rs,Rl,Rh)
18          End for
19          End if
20  End for
21  Return Rs.
22  Searchlivelist(E,r,Rs,Key)
23  for each query(Q)∈ E then search live entries block that contains key value k
24   if r.Ts∈ [reg(Q) &&r.T.V∈ Q.I[ai]] where i∈[1,...n]
25  Rs=Rs(Q)
26   End if
27   Return Rs.
28  End
```

5 Implementation

A real-time system, i.e., a stock market application has been used for experimentation. Live readings are recorded from

the online stock web site http://www.money.rediff.com. An application named TrayApp is developed which records the livestock market values and continuously updates the current stock value of each company. This updating is done every time when there is a small change in the live values. National Stock Exchange (NSE) and BSE(Bombay Stock Exchange), which provide livestock readings of various companies, are used in the web application. Stock entries are updated every 30 s continuously. The readings are recorded daily during business hours. Simple, complex, time-based and predictive analysis queries are considered in this work. The user cannot find queries regarding predictive analysis and comparative analysis between companies in the present scenario. This webpage also provides a search area for the users to post their queries and obtain reliable answers in a quick way. The web page also provides comparison charts on future predictions about the stock market and can give suggestions to the user on future investments. For new users who do not know much about the stock market, the web page provides a well-defined view on the stocks and provides suggestions for investing in companies. An API is developed for query processing, retrieving live stock market data from the web and processing the requested query. Data tuples are considered in the range of 100–1000, varying based on the incoming data arrival. The incoming entries are updated for every 30 s. The maximum number of entries in a node is 8 and underflow is considered below 2 entries. A strong version condition underflow is considered as 3 entries to maintain a balanced tree.

5.1 Query formulation

Stock quotes of companies are used as a streaming database that is uploaded daily or even hourly and readings are taken every 30 s to include new quotes. Some sample queries based on exact matches and range queries have been considered for the experiment. We formalize the queries as follows:

Definition 1 (Valid key update) this query takes the readings of stock values continuously every 30 s (t). Given a streaming data Ds, let q be the streaming query object at time position t. The timestamps of the query for the last Ts time period would be r,

$$\begin{cases} \forall D_s \left\{V\left(q\left[t-T_s\right], r\left[t-T_s\right]\right)\right\}, \Delta\left(T_s\right) > 0 \; and \; \Delta\left(P\right) \neq 0 \\ \forall D_{s,} \quad \Delta\left(T_s\right) > 0 \quad and \quad \Delta\left(P\right) = 0 \end{cases} \tag{8}$$

The insertion of new tuples is described in (eq8) if there is a variation in time and price. If there is no variation in the previous value, only timestamp is updated. This denotes the variation in the timestamp Ts and stock value P.

Definition 2 (To obtain the min value and max value for each key maintained in Ds) given a streaming database Ds, let q be the streaming query object at time position t then,

$$Mn = \forall \mathrm{Ds} \sum_{k=1}^{n} \min\left(q\left[t\right], r\left[t\right]\right) \tag{9}$$

$$\mathrm{Mx} = \forall \mathrm{Ds} \sum_{k=1}^{n} \max\left(q\left[t\right], r\left[t\right]\right) \tag{10}$$

where $k = 1, 2, 3 \ldots n, n$ is the amount of incoming data maintained in Ds. The minimum value of the key is denoted in (9) and the maximum value of quotes is represented in (10).

Definition 3 (Displaying list of gainers and losers) given a streaming database, let q be the streaming query object at time position t,

$$Gain = \sum_{\mathrm{Ds}=1}^{n} Max(\%ch\left[q\left[t\right]\right) \tag{11}$$

where %ch is the percentage change in values.

$$Loss = \sum_{\mathrm{Ds}=1}^{n} Min\left(\%ch\left[q\left[t\right]\right]\right). \tag{12}$$

Equations 11 and 12 represents the list of gainers, and losers.

6 Performance evaluation

The performance of the proposed indexing method ACBBI is evaluated for streaming data. CKDB tree (cell-based KDB tree) was a similar approach to the proposed ACBBI system. In their systems, indexing was done by combining adaptive cell and KDB-tree to support dynamic continuous queries over streaming data. Both query indexing and filtering of streaming data were done. Queries were split and stored in cells. Each cell was partitioned into equal sized sub-cells. The drawback of this existing system was cross boundary queries were maintained with multiple index entries which leads to more space overhead and maintenance costs. This drawback is overcome in this proposed system ACBBI. Live stock market values, which continuously produce data that vary in time, are used for experimentation. The storage costs, maintenance costs and query performance of the proposed indexing method are evaluated. Live entries and stock entries are maintained separately to efficiently handle query retrieval. Query indexing approaches must handle dynamic continuous queries, such as exact match queries and range queries. Existing systems concentrate mostly on range queries only.

Table 3 Performance comparison of ACBBI with the web-based approach

Approach/streaming data	Adaptive clustering based approach (ACBBI)	Traditional Web based approach	Variation in readings
No. of entries/min	250	∼>1500	>1200
No. of entries/h	696	41,760	∼> 40,000
Valid input data/day	1624	2,71,440	2,69,816
Response time/entry	0.6	≥15	∼14.4
Available memory (MB)	21.56	3550	3528.44
Memory used (MB)	1.94	280	>270
Remaining memory (MB)	19.62	3270	3250

Exact match queries and range-based queries, which are more frequent, are considered and tested in the proposed system. Frequently used queries are considered as registered queries that execute continuously to evaluate the proposed indexing method. Data entries are tested by varying the number of tuples from 1,00,000 (100 K) to 9,00,000 (900 K), which is considered as the maximum range of incoming tuples.

6.1 Data retrieval evaluation

Indexing should be adaptive, efficient and reliable due to the dynamic nature of incoming data in data stream systems. Cluster-based indexing is analyzed in terms of data indexing and query indexing. Data indexing is considered for efficient storage and fast retrieval. Query indexing is required to handle dynamic continuous queries, which lead to quick response and processing time of queries. ACBBI includes both data and query indexing. This indexing is compared with an existing cell tree indexing method called the CKDB tree. The CKDB [6] tree splits the query into cells. In this existing work, most frequent queries are maintained in the top-k list and the remaining queries are in the KDB tree. The same query is fragmented into two cells, so referring that query requires the comparison of more than one cell. The query retrieval must compare both the top-k list and the KDB tree, which requires additional space and more processing time. In ACBBI, only required live data are updated. All the incoming streams of data are not stored like traditional systems. Because stock values are used for experimentation, only variation in values and changes in the timestamp are considered. Only timestamps are updated for the stock quotes that do not change for a particular time period. A new indexing node does not need to be inserted in the tree. Only updating of the old entries is performed, which leads to space consumption. Each cluster size is determined by the variation in incoming data entries and time, which are allocated based on their block size. The cluster size is calculated as shown in Eq. 13:

$$C_i = \alpha^j + \beta^j . \frac{\left(\bigcup_{i=1}^{k} N_i\right)}{B}. \tag{13}$$

The number of entries varies over time j. α^j. is a constant factor determining the minimum entry that varies over time j. N_i. is the change in β values, which may increase or decrease. β^j (±values) varies over time j. B is the block size, which determines the number of entries occupied in one cluster.

Readings from 232 companies are taken for experimentation as shown in Table 3. An average of more than 40,000 records is retrieved for an hour in traditional existing system. More than 2 lakh entries are received per day. These readings vary based on different input queries. More than 90% variation in performance is found from the results obtained. In the proposed ACBBI work, all incoming data are not required for processing queries. Duplicate records are eliminated by filtering incoming data using adaptive incremental clustering based approach. Though it takes a little bit of processing time, it saves 90% of memory and processing time. Also, accuracy in results is maintained subsequently for exact information is retrieved always.

6.2 Storage cost evaluation

Storage for a given set of queries are evaluated and compared with existing CKDB tree and KDB-tree. A set of query data Q where number of queries at a time |Q| is 10,000 considered for execution. Amount of storage required for using ACBBI is compared with existing system using real-time application which is shown in Fig. 3. The M, N and constant value c are used where M refers to the actual number of records without incremental updation and N is the number of records after adaptive incremental updation. Eq. 14 shows that N is lesser than M. When the queries are executed using ACBBI for a 400 K number of queries 4000 MB is used while the existing system CKDB consumes 8000 MB for the same number of queries.

$$N \leq \sqrt{M}.c \tag{14}$$

The comparison between the proposed storage of ACBBI indexing approach and the CKDB tree is done and shown in Fig. 3. The streaming processor when receiving 1500 number of records, with the CKDB existing system approach of

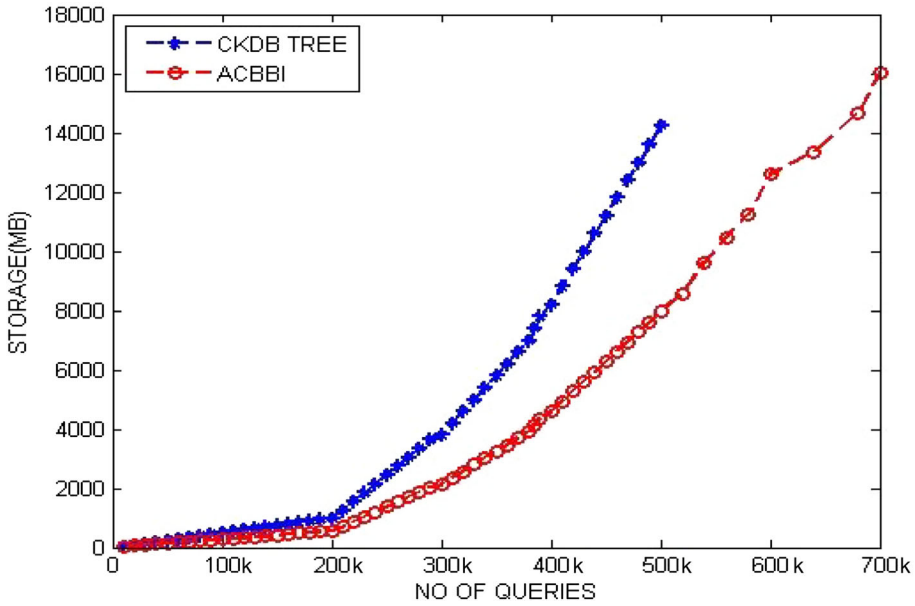
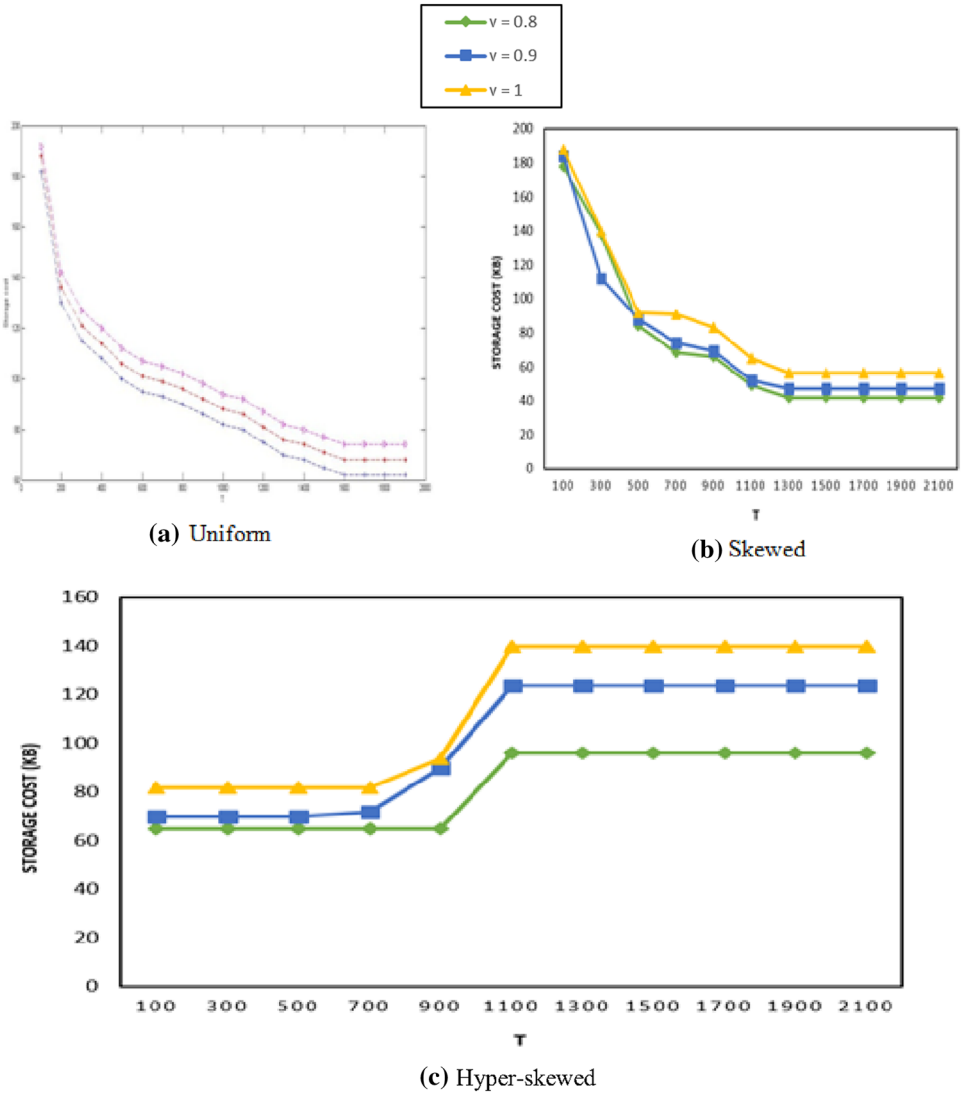


Fig. 3 Storage comparison of ACBBI with CKDB-tree



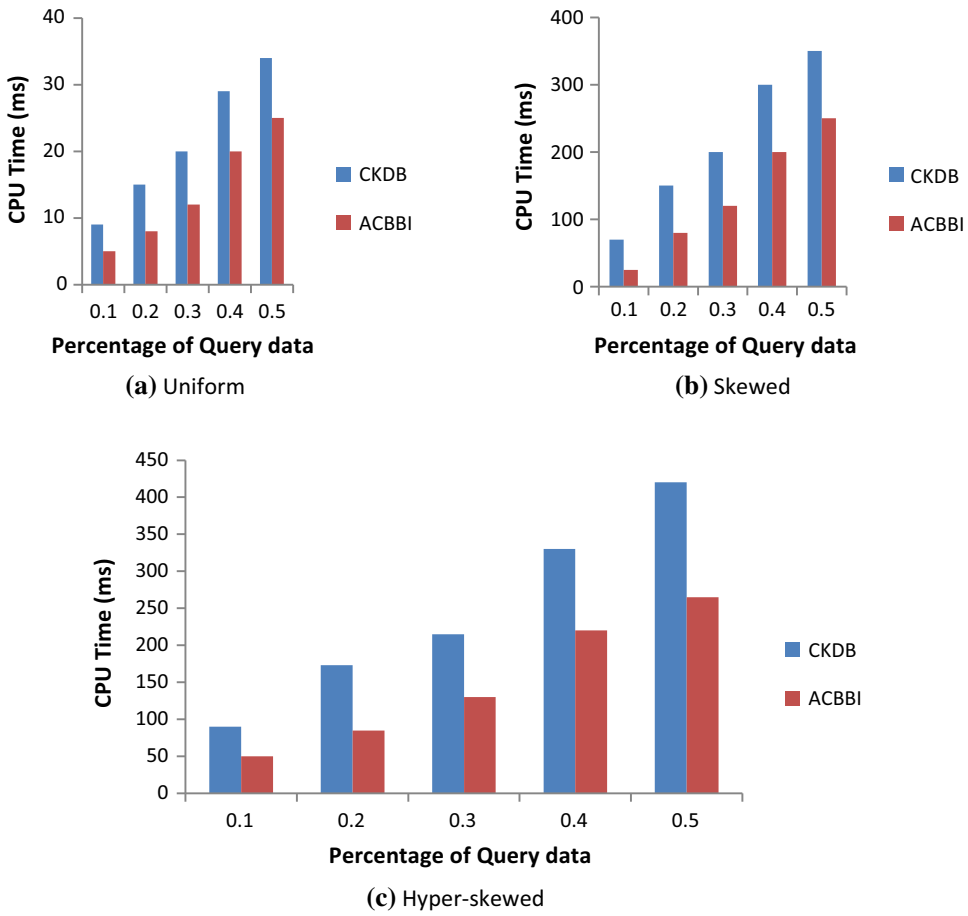
(a) Uniform

(b) Skewed

(c) Hyper-skewed

Fig. 4 Space cost with parameters of cluster size T by varying v

Fig. 5 Index maintenance cost



indexing occupies 280 MB space while executing a 100 K number of queries. However, the proposed ACBBI indexing has occupied 1.94 MB only since the arrival of 1500 number of records is reduced to 250 numbers. Hence, the incoming streaming data are analyzed and duplicate values removed. The storage is highly reduced by the proposed indexing method. The storage size is reduced by 44% when compared with CKDB-tree. Moreover, ACBBI scales up for the huge data values too.

6.3 Cluster capacity evaluation

Each cluster capacity is viewed from varying the coverage ratio of cluster represented as v and cluster capacity denoted by T. Amount of storage is calculated by varying the data distribution with the given set of queries. The overflow condition of cluster capacity is considered for 75% of incoming data. Therefore, the space cost is monitored by changing the block size with 80, 90 and 100% and are shown in Fig. 4. Query data apply with three distributions of uniform, skewed and hyper-skewed cases are evaluated for estimating the storage cost of ACBBI. The constant arrival of streams is measured and represented as uniform distribution that is calculated by the coverage ratio of blocks and is shown in Fig. 4a Storage cost is almost similar when there is variation in cluster size.

Apart from the uniform distribution, sudden changes in streams which is known as skewed data are measured when the threshold value for block size reaches 100% at $v = 1$, and storage cost increases T, the cluster capacity value is optimal and uniform when the cluster size increases gradually. Skewed distribution is shown in Fig. 4b. The storage cost of the high speed rate of incoming data named as hyper-skewed and is shown in Fig. 4c. Storage capacity increases when the cluster size T is above 1000. However, the results are consistent when there is an increase in cluster size. The performance is high compared to the existing system in skewed and hyper-skewed cases.

6.4 Index maintenance cost evaluation

The index maintenance cost is measured using storage space used by the clustered streaming data by varying percentages of query data from 10 to 80%. ACBBI outperforms than existing KDB and CKDB-tree. The ratio of frequency in updation time of ACBBI is compared with CKDB, ACBBI varies with 34% than CKDB tree. The updation in ACBBI happens only

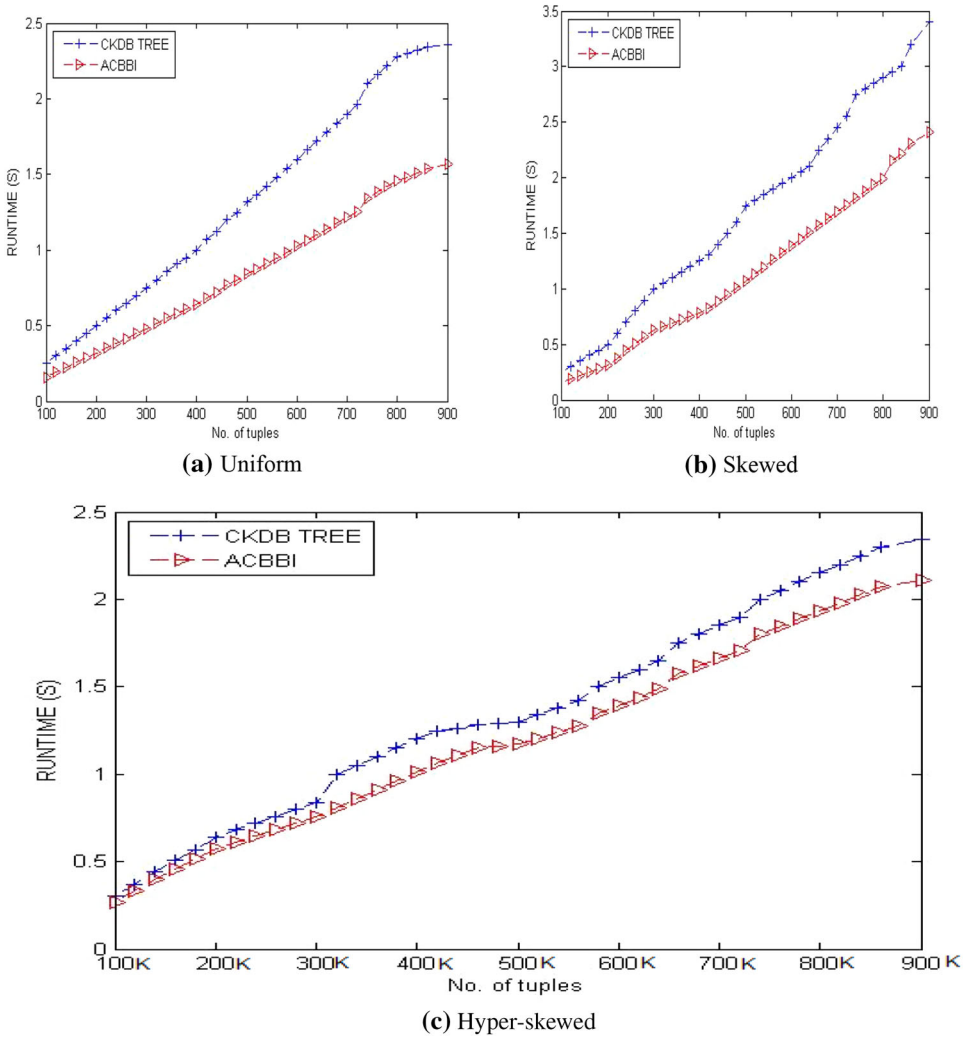


Fig. 6 Query Performance of ACBBI compared with CKDB tree

(a) Uniform **(b)** Skewed

(c) Hyper-skewed

when there is a change in value and timestamp in incoming record which reduces the amount of update operations. Thus, execution time of update operation is less than the existing system and shown in Fig. 5.

Figure 5a shows the uniform readings where x-axis represents the percentage of query data with respect to CPU execution time. ACBBI varies by more than 34% compared with CKDB. ACBBI varies by more than 10–50% compared with CKDB respectively, both in the cases of skewed and hyper skewed and is shown in Fig. 5b and c.

6.4.1 Query performance evaluation

The ACBBI indexing method is compared with an existing CKDB-tree in terms of query performance. A set of queries is executed continuously by varying the number of tuples. CPU time is measured with the number of data tuples using ACBBI and CKDB. Execution time of the uniform, skewed and hyper-skewed distribution of data while varying the number of queries is shown in Fig. 6a, b and c respectively. ACBBI performs much higher than CKDB-tree around 36% for uniform, skewed cases and 10% of hyper skewed cases. The proposed method outperforms for voluminous records. ACBBI approach on an average, improves query performance than CKDB tree. The search through ACBBI index has only the minimum amount of data because the insertion of all incoming streaming data is avoided.

7 Conclusion

Adaptive clustering block-based indexing (ACBBI) is proposed and implemented to capably process data streams. ACBBI consists of both cluster-based and block-based indexing methodologies. The adaptive clustering algorithm clusters incoming streaming data dynamically based on the extension of an incremental clustering method. Block-based indexing reduces the storage space and provides easy retrieval. Different data distributions of incoming data, such

as uniform, skewed and hyper-skewed by varying query data, are investigated to measure the space cost and query performance. The results show that (1) ACBBI has an improved its performance of retrieving data by 41% to that of existing systems and works proficiently in easy retrieval. (2) This system also outperforms more than half times of the existing system in terms of storage cost. An experimental analysis, a real-time stock market application is considered. The incoming streaming data for this application are handled by the proposed ACBBI. The efficiency of the stream query processing is improved and the space cost is reduced. Frequent updates of incoming data are accommodated. Thus, ACBBI shows substantial potential in reducing the storage cost, and the retrieval rate is improved with increasing data size. Hence, an effective data stream management technique is devised and analyzed in this work. Big data is the latest technology where huge amount of data is stored and processed. In the future, this approach may be enhanced for the streaming of big data to efficiently store and retrieve huge volume of data. The velocity of huge data can be retrieved instantly by enhancing dynamic heuristic optimization technique. This can be applied in various other streaming, time-variant applications and semantic approach can be applied. Semantic based streaming data along with adaptive indexing and dynamic query processing will improve the efficiency and scalability of stream processing in future.

References

1. Amini, A., Wah, T.Y., Saboohi. H.: On density-based data streams clustering algorithms: a survey. J. Comput. Sci. Technol. **29**(1), 116–141 (2014). doi:10.1007/s11390-013-1416-3
2. Angelov, P., Filev, D.: An approach to online identification of Takagi-Sugeno fuzzy models. IEEE Trans. Syst. Man Cybern. B **34**, 484–498 (2004)
3. Angelov, P.P., Zhou, X.: Evolving fuzzy-rule-based classifiers from data streams. IEEE Trans. Fuzzy Syst. **16**(6), 1462–1475 (2008)
4. Badiozamany, S., Risch, T.: Scalable ordered indexing of streaming data, VLDB Proceedings (2012)
5. Chen, T., Chen, L., Ozsu, M.T.: NongXiao, optimizing multi-Top-k queries over uncertain data streams. IEEE Trans. Knowl. Data Eng. **25**(8), 1814–1829 (2013)
6. Deng, X.W., Wang, L., Chen, X., Ranjan, R., Zomaya, A., Chen, D.: Parallel processing of dynamic continuous queries over streaming data flows. IEEE Trans. Parallel Distrib. Syst. **26**(3), 834–845 (2015)
7. Ferchichi, A., Gouider, M.S.: BSTree—an incremental indexing structure for similarity search and real time monitoring of data streams. Lecture Notes in Electrical Engineering, Future Information Technology, vol. 276, pp. 185–190. Springer, Heidelberg (2014)
8. Gulisano, V., Jimenez-Peris, R., Patiño-Martínez, M., Soriente, C.: StreamCloud: an elastic and scalable data streaming system. IEEE Trans. Parallel Distrib. Syst. **23**(12), 2351–2365 (2012)
9. Hesabi, Z.R., Sellis, T., Zhang, X.: Anytime Concurrent Clustering of Multiple Streams with an Indexing Tree. JMLR: Workshop and Conference Proceedings, vol. 41, pp. 19–32 (2015)
10. Khalilian, M., Mustapha, N.: Data stream clustering: challenges and issues. In: Proceedings of International Multi Conference of Engineers and Computer Scientist IMECS, vol. 1(1) (2010)
11. Kholghi, M., Keyvanpour, M.R.: Comparative evaluation of data stream indexing models. Int. J. Mach. Learn. Comput. **2**(3), 257–260 (2012)
12. Kontaki, M., Papadopoulos, A., Manolopoulos, Y.: Continuous trend-based clustering in data streams. Data Warehous. Knowl. Discov. 251–262 (2008)
13. Luan, H., Du, X., Wang, S.: Prefetching, J+ tree: a cache-optimized main memory database index structure. J. Comput. Sci. Technol. **24**(4), 687–707 (2009)
14. Park, J., Hong, B., Ban, C.: An efficient query index on RFID streaming data. J. Inf. Sci. Eng. **25**, 921–935 (2009)
15. Patrick Valduriez INRIA, Montpellier, Indexing and Processing Big Data, 2014. http://www.lirmm.fr/mastodons/talks/Valduriez-Bigdata-indexing-2014.pdf
16. Pratama, M., Lu, J., Zhang, G., Anavatti, S.: Evolving type-2 fuzzy classifier. IEEE Trans. Fuzzy Syst. **24**(3), 574–589 (2015)
17. Pratama, M., Lu, J., Zhang, G., Anavatti, S.: Scaffolding type-2 classifier for incremental learning under concept drifts. Neurocomputing **191**, 304–329 (2016)
18. Pratama, M., Lu, J., Zhang, G., Anavatti. S.: An incremental type-2 meta-cognitive extreme learning machine. IEEE Trans. Cybern. (99) 1–15 (2016)
19. Pratama, M., Anavatti, S., Lughofer, E.: pClass: an effective classifier to streaming examples. IEEE Trans. Fuzzy Syst. **23**(2), 369–386 (2014)
20. Pratama, M., Anavatti, S., Lu, J.: Recurrent classifier based on an incremental meta-cognitive scaffolding algorithm. IEEE Trans. Fuzzy Syst. **23**(6), 2048–2066 (2015)
21. Punithavalli, K.V.M.: Clustering time series data stream—a literature survey. Int. J. Comput. Sci. Inf. Secur. (IJCSIS) **8**(1), 289–294 (2010)
22. Saleh, O., Hagedorn, S., Sattler, K.-U.: Processing, complex event, on linked stream data. Datenbank Spektrum **15**, 119–129 (2015). doi:10.1007/s13222-015-0190-5
23. Santoso, B.J., Chiu, G.-M.: Close dominance graph: an efficient framework for answering continuous top-k dominating queries. IEEE Trans. Knowl. Eng. **26**(8) 1853–1865 (2014)
24. Shoshani, On the Role of Indexing in Scientific Domains. Big data and Extreme Computing. Lawrence Berkeley National Lab (2013). http://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/17_BDEC_Shoshani_indexing.pdf
25. Silva, J.A., Faria, E.R., Barros, R.C., Hruschka, E.R., De Carvalho, A.C.P.L.F., Gama, J.A.P.: Data stream clustering: a survey. J. ACM **46**(1) (2013)
26. Wang, J., Lam, K.-Y., Chang, Y.-H., Hsieh, J.-W., Huang, P.-C.: Block-based multi-version B^+tree for flash-based embedded database systems. IEEE Trans. Comput. **64**(4), 925–940 (2015)
27. Xie, Q., Zhang, X., Li, Z., Zhou, X.: Optimizing cost of continuous overlapping queries over data streams by filter adaption. IEEE Trans. Knowl. Data Eng. **28**(5), 1258–1271 (2016)
28. Yogita, D.T.: Clustering techniques for streaming data—a survey. IEEE Conference on Advance Computing Conference (IACC), pp. 951–956 (2013). doi:10.1109/IAdCC.2013.6514355
29. Zheng, L., Huo, H., Guo, Y., Fang, T.: Supervised adaptive incremental clustering for data stream of chunks. J. Neurocomput. 502–517 (2017). http://dx.doi.org/10.1016/j.neucom.2016.09.054

M. R. Sumalatha is an Associate Professor at Department of Information Technology and Deputy Director at Centre for Technology Development and Transfer, Anna University. She has been selected as one of the leading achievers around the globe from New Providence, USA (2010) and for Dr. APJ Abdul Kalam Award for Teaching Excellence in the year 2015. She has many research publications to her credit and her area of research interest includes Distributed Systems, Cloud Computing, Big Data Analytics, Social Platforms, Data Security and Privacy.

M. Ananthi is an Associate Professor at Department of Information Technology, Sri Sairam Engineering College. She received her B.E degree from the Bharathidasan University, Trichy, M.E degree from Sathyabama University, Chennai. She is currently doing her PhD in the area of Data Management issues and online data stream analysis in Anna University. She has published her research work pertaining to data streams in international conferences Her research interests include data management, data streaming, and query processing.