A Distributed B+Tree Indexing Method for Processing Range Queries over Streaming Data

Shahab Safaee
Islamic Azad University
Meghdad Mirabi (meghdad.mirabi@cs.tu-darmstadt.de)
Technical University of Darmstadt
Amir Masoud Rahmani
National Yunlin University of Science and Technology
Aliasghar Safaei
Tarbiat Modares University

Research Article

Keywords: B+Tree Index, Distributed Query Processing, Map-Reduce Model, Range Query, Streaming Data

Posted Date: August 18th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1941097/v1

A Distributed B+Tree Indexing Method for Processing Range Queries over Streaming Data

Shahab Safaee[1], Meghdad Mirabi[2], Amir Masoud Rahmani[3], Aliasghar Safaei[4]

[1] Department of Computer Engineering, Faculty of Engineering, Islamic Azad University, South Tehran Branch, Tehran, Iran
[2] Faculty of Computer Science, Technical University of Darmstadt, Germany
[3] Future Technology Research Center, National Yunlin University of Science and Technology, Taiwan
[4] Department of Medical Informatics, Faculty of Medical Sciences, Tarbiat Modares University, Tehran, Iran

safaee.shx@gmail.com, meghdad.mirabi@cs.tu-darmstadt.de, rahmania@yuntech.edu.tw, aa.safaei@modares.ac.ir

Abstract: A data stream exhibits as a massive unbounded sequence of data elements continuously generated at a high rate. Stream databases raise new challenges for query processing due to both the streaming nature of data which constantly changes over time and the wider range of queries submitted by the user when compared with the traditional databases. In this paper, we propose a system architecture which includes components for both distributed indexing of streaming data and distributed processing of range queries over streaming data. By exploiting the proposed system architecture, the process of indexing of streaming data and the process of querying over streaming data can be done in a distributed fashion. We also design a distributed B+Tree indexing method using the map-reduce programming model of the Apache Spark framework which creates small B+Tree indexes on the machines of a Spark cluster instead of using a large and centralized B+Tree index structure. Moreover, we propose a distributed range search algorithm to process range queries in distributed and parallel form using the set of small B+Tree indexes. By performing several experiments, we demonstrate that our proposed distributed B+Tree indexing method is scalable and efficient compared to the existing indexing methods and therefore, it can be used for applications involving data streams with a large volume of data elements and a large number of range queries.

Keywords: B+Tree Index, Distributed Query Processing, Map-Reduce Model, Range Query, Streaming Data

1. Introduction

Today, the use of streaming data in applications such as sensor networks, internet of things, stock market data analysis, etc., has turned significant. In these applications, data is generated in transient, at a high rate, from various sources and must be processed in the shortest possible time and even in some applications within a deadline. The generation rate of streaming data may change dynamically; also, they may be infinite in many applications such as the data of sensors [1-3].

Streaming data can be defined as an order of data elements in a continuous, infinite, transient, unpredictable and time-variant [1, 4, 5]. The paradigm and processing structure that governs this type of data is different from those of common systems. Usually, the processing of this type of data is done in real time [1, 6]. Due to the importance of streaming data in various fields and applications, there is a need for systems that store, retrieve and process this type of data based on the characteristics of streaming data [7-9].

One of the important issues in streaming data is the query processing time. The most common solution to improve the access time during query processing is to use an indexing structure to facilitate the process of data retrieval over streaming data [10-12]. Generally, different structures can be used to implement the index depending on the type of data, the type of query, and the type of application. As common examples of basic indexing structure, the structures of Bit-Map [13], B-Tree [14], and Hash [12] can be mentioned. Due to the use of a data structure to maintain the information of index, the process of indexing is expensive in terms of memory cost and therefore it has memory overhead [11, 15-17].

Unlike batch processing systems, streaming data must be stored in the main memory and processed in real time. Therefore, common indexing methods that are based on batch processing may not be efficient for this type of data. On the other hand, in non-streaming data management systems, such as traditional database management systems, data is stored permanently on storage devices, and the index structures are used to index data permanently. As a result, the rate of data changes in the index is not high and the cost of maintaining the index is low. In contrast, streaming data are transient in nature and it must be processed within the specified deadline. Therefore, the structure of the index for streaming data cannot be permanent. In such systems, the rate of data changes is high. To avoid increasing the size

of index from the data whose time stamp has expired, the index should be updated or pruned; this will impose overhead on the efficiency of data stream indexing [16, 18, 19].

One of the most common queries on streaming data is range queries. A range query retrieves the data between a lower bound and an upper bound from the existing streaming data [20-22]. One of the suitable index structures to facilitate the process of range queries is the use of tree-based structures, especially the B+Tree [16, 20, 23]. Most of the proposed tree-based indexing methods have challenges of single point-of-failure, scalability and efficiency [18-20, 24] due to having a centralized structure and not being designed for a large volume of streaming data. Therefore, there is a strong need to propose a scalable and reliable indexing method with high performance to process range queries over streaming data.

Among the most common distributed processing platforms that work based on the map-reduce programming model, the two most common platforms are the Apache Hadoop [26, 27] and the Apache Spark [28-30]. The Apache Hadoop platform uses the disk to store data, which may not be very efficient for applications involving streaming data, but the Apache Spark platform uses the main memory instead of using the disk for storing data and intermediate results and therefore, it has higher performance [28, 31, 32]. In the Apache Spark platform, a distributed memory structure called Resilient Distributed Datasets (RDD) is used to store data. Such a structure is fault tolerant [28] and when the data is placed in the RDD, this platform will transform the data to another RDD and during this transformation, the desired operations will be done on the data in a distributed manner and then, the result will be stored in another RDD. Another feature of the Apache Spark platform is that it provides a library for processing streaming data called Spark Streaming [33] on the Spark processing core. In this library, it is possible that streaming data can be inserted into an RDD-like structure, called DStream, in a short discrete time interval and then distributed processing can be done using transformation functions such as map(), partitionby(), groupbyKey() and reducebyKey().

In this paper, triggered by its importance, we study the problem of processing range queries over data stream in an efficient and scalable manner. We propose a distributed B+tree index structure to facilitate the process of range queries over data stream. The proposed index structure is designed based on the map-reduce programming model [25] in the Apache Spark framework to benefit from the existence of a distributed and reliable data processing platform. Hence, our main contributions in this paper are summarized as follows:

- We propose a system architecture for both distributed indexing of streaming data and distributed processing of range queries over streaming data. It consists of 2 components called “Distributed Indexing” component and “Distributed Query Processor” component. The “Distributed Indexing” component is responsible to create a set of small B+Tree indexes on the machines of a Spark cluster while the "Distributed Query Processor” component is responsible for processing range queries in a distributed fashion using B+Tree indexes available on the machines of a Spark cluster.
- We design a distributed B+Tree indexing structure using the map-reduce programming model of the Apache Spark to create a set of small B+Tree indexes instead of using a large and centralized B+Tree index. The proposed distributed index structure is scalable and it can be used where the volume of streaming data is large and there are a large number of range queries.
- We propose a distributed range query algorithm called “Distributed Range Search” using the map-reduce programming model of the Apache Spark platform, which can be executed in parallel and distributed manner on the machines on which the B+Tree indexes are placed on.
- We evaluate the performance of our proposed distributed B+Tree indexing method for processing range queries over data stream by performing several experiments.

The rest of the paper is organized as follows: the related works are reviewed in Section 2. Our proposed system architecture and our proposed indexing method are explained in Section 3. In Section 4, our proposed distributed B+Tree indexing method for processing range queries over data stream is evaluated by performing several experiments. Finally, this paper is conducted by a conclusion and discussion of future works in Section 5.

2. Related Works

Several research works have been done to index streaming data, each of which has its own advantages and disadvantages and can be used in a specific application. From the point of view the type of data on which such indexing methods supports, they can be divided into two categories of one-dimensional indexing methods [16, 20, 24, 38, 39], and multi-dimensional indexing methods [18, 34-36]. One-dimensional indexing methods are used for relational data with a tabular structure where streaming data includes a set of tuples of that relation, while multi-dimensional indexing methods are suitable for spatial data that streaming data includes a set of spatial coordinates such as the location of a moving object. Basically, the overhead of maintaining the index in the case of multi-dimensional data is higher than one-dimensional data [15, 18]. In one-dimensional indexing methods, a tree-based structure (e.g., B-Tree and B+Tree) is used to index the data. In [24, 37], a tree-based cache-aware indexing method called CSB+Tree is proposed to index streaming data in main memory. By exploiting the CPU cache, the CSB+Tree indexing method improves the search time by consuming more memory, but the processes of insertion and modification of index are performed a little slower than typical B+Tree indexing method. In ACBBI (Adaptive Clustering and Block-Based Indexing) indexing method proposed by [20], streaming data is first clustered and then, the generated clusters are inserted into B+Tree. Since this indexing method clusters a set of tuples based on similarity and stores them as a block, it has less storage overhead compared to the other tree-based indexing methods [16, 24, 38, 39]. The most important problem of this indexing method is that it only creates a B+Tree index to process queries over streaming data and therefore, it is not scalable and reliable.

A Trie is a multi-way tree structure in which each node is an array of pointers. For example, to index alphabetical words, the size of each array is equal to the number of letters in the alphabet, and each level in a Trie is used to index a letter in a word. The main advantage of Tries based indexing methods is that the access time and insertion time are constant if the key length is constant. Thus, Tries should be very well suited for indexing data stream windows with very high insert rates. The most important drawback of Tries based indexing method is memory wastage, when keys are uniformly scattered due to the many null pointers in the sparse pointer arrays representing Trie nodes. Burst Trie [38], Judy [39, 40] and Extended Judy [16] are the most important research works which use Trie as a tree structure to index the streaming data. In Judy [39] and Extended Judy [16], it has been tried to improve the problem of high memory consumption by utilizing various mechanisms such as compression. In terms of the performance of range searches on these indexes, Burst Trie [38] and Judy [39] do not have good performance, but the Extended Judy [16] has improved the performance of range search operation compared to the other two indexing methods, but this indexing method has become more complex.

In general, indexing methods to process range queries on multi-dimensional spatial data stream can be divided into three categories: tree-based, cell-based, and hybrid indexing methods. The main drawback of tree-based indexing methods such as R*-Tree [34], KDB-Tree [35] is that they have high maintenance cost due to the nature of index tree and may have overlap problems. In cell-based indexing methods like VCR (Virtual Construct Rectangles) [36], a grid structure is generated to partition the indexing space into equal-sized cells. Cell-based indexing methods have better update and query performance than the tree-based indexing methods [41], but the storage cost and the time to construct/reconstruct the index is high. There are also research works that use combination of tree and cell structures together in the index structure like CKDB-Tree and G-CKDB-Tree in [18]. The advantage of this indexing method compared to the tree-based and cell-based indexing methods is that it has less storage cost and better efficiency in search operation. Even in G-CKDB-Tree indexing method, parallel processing is employed using GPU in order to further improve performance. However, all of these proposed indexing methods are centralized and therefore, they have the problem of single point-of-failure and the lack of scalability.

In the literature, a number of distributed indexing methods have been proposed such as those proposed in [19, 23, 42]. In [19], a fault-tolerant and scalable distributed B-tree as an index structure is proposed which provides some important practical features: transactions for atomically executing several operations in one or more B-trees, online migration of B-tree nodes between servers for load-balancing, and dynamic addition and removal of servers for

supporting incremental growth of the system. The proposed index method in [19] is implemented on an underlying distributed data sharing service, called Sinfonia [43], which provides fault tolerance and a light-weight distributed atomic primitive. However, this indexing method is only used for transactional queries and does not support streaming data. In [23], a new distributed R-Tree index structure for trajectory search called DTR-Tree (Distributed Trajectory R-Tree) is proposed using the Spark Apache framework based on the map-reduce programming model. Such an indexing method is scalable and reliable and it is able to optimize the trajectory search operation. In [42], a distributed B-Tree indexing method using map-reduce programming model is proposed to improve the efficiency of random reads. This indexing method is implemented using a chained map-reduce process that reduces intermediate data access time between successive map and reduce functions, and improves efficiency of random reads.

There are several factors to be considered in order to compare the existing indexing methods for streaming data which are listed as follows:

- Index Structure: What structure is used in the proposed indexing method?
- Index Type: Can the proposed indexing method be used only in a centralized form or can it be used in a distributed fashion?
- Data Type: Does the proposed indexing method only support one-dimensional data or is it able to support multi-dimensional data as well?
- Type of Query Supported by the Indexing Method: Is the proposed indexing method able to process range queries over data stream or not?
- Cost of Indexing Method: Is the cost of the proposed indexing method is high in terms of storage cost and maintenance cost?
- Type of Improvement: Does the proposed indexing method improve the query processing performance over data stream? We can consider several metrics in this regard such as efficiency, scalability, and reliability.

Table 1 summarizes the characteristics of existing indexing methods and highlights the differences between the proposed indexing method in this paper and the existing indexing methods. As shown in Table 1, all existing tree-based data stream indexing methods need to update the index tree to prevent the growing of index tree and therefore, the index maintenance cost in these indexing methods is high. In this paper, we propose a distributed B+Tree indexing method which creates a set of small index trees on several machines (or nodes) of a Spark cluster instead of creating a large B+Tree index tree. These B+Tree index trees are created in parallel from in certain time intervals using the map-reduce programming model on the Spark Apache platform. Our proposed distributed B+Tree indexing method completely eliminates the process of maintaining a large B+Tree index in main memory. It is developed to be used in a Spark cluster and therefore, it is reliable. The experimental results in Section 5 demonstrate that it is efficient and scalable for processing range queries over streaming data. However, it has a high storage cost compared to other tree-based indexing methods since it creates a set of small B+Tree index trees in the main memory of machines in a Spark cluster.

Table 1 Comparison of streaming data indexing methods

Index Name	Indexing Structure	Indexing Type	Data Type	Support Range Query	Data Stream Support	Storage Cost	Maintenance Cost	Efficiency	Scalable	Reliable
ACBBI [20]	B+Tree	Centralized	One Dimensional	Yes	Yes	Low	Low	High	No	No
CSB+Tree [24]	B+Tree	Centralized	One Dimensional	Yes	Yes	High	Medium	High	No	No
Extended Judy [16]	Trie	Centralized	One Dimensional	Yes	Yes	Low	Low	Medium	No	No
Judy [39]	Trie	Centralized	One Dimensional	Yes	Yes	Low	Medium	Medium	No	No
Burst Tries [38]	Trie	Centralized	One Dimensional	Yes	Yes	Medium	Medium	Low	No	No
R*-Tree [34]	R-Tree Based	Centralized	Multi-Dimensional	Yes	Yes	Medium	High	Medium	No	No
KDB-Tree [35]	B-Tree & K-D-Tree	Centralized	Multi-Dimensional	Yes	Yes	Medium	Medium	Medium	No	No
VCR [36]	Cell Based	Centralized	Multi-Dimensional	Yes	Yes	Medium	Medium	Medium	No	No
CKDB-Tree [18]	Cell & Tree Based	Centralized	Multi-Dimensional	Yes	Yes	Low	Medium	Medium	No	No
G-CKDB-Tree [18]	Cell & Tree Based	Centralized	Multi-Dimensional	Yes	Yes	Low	Medium	High	No	No
DTR-Tree [23]	R-Tree Based	Distributed	Multi-Dimensional	Yes	No	High	Medium	High	Yes	Yes
Distributed BTree [42]	B-Tree Based	Distributed	One Dimensional	Yes	No	Medium	Medium	Medium	Yes	Yes
Distributed BTree [19]	B-Tree Based	Distributed	One Dimensional	No	No	Medium	Medium	Medium	Yes	Yes
Proposed Indexing	B+Tree	Distributed	One Dimensional	Yes	Yes	High	Low	High	Yes	Yes

3. Our Proposed System Architecture and Indexing Method

In this section, we explain about our proposed system architecture and our proposed indexing method for processing range queries over data stream in more detail.

3.1 Architecture of the proposed method

Our proposed system architecture is shown in Fig. 1. As shown in Fig. 1, our proposed system architecture contains the following components: “Distributed Indexing” component and “Distributed Query Processor” component. “Distributed Indexing” component is responsible for data stream distribution, partitioning, and indexing. It consists of three modules:

- Data Distributing Module: This module gets streaming data from the input and stores it in the form of a distributed memory structure on the machines inside a Spark cluster.
- Data Partitioning Module: This module partitions the distributed data based on the indexing key. Partitioning is in the form of horizontal and we use the range partitioning method in this module.
- Data Indexing Module: This module creates a B+Tree index in each partition of data.

The Distributed Query Processor component is responsible for processing range queries over streaming data and generating query results. It consists of two modules:

- Query Executor Module: This module gets a range query from the input and processes it in a distributed manner using the B+Tree index of each partition.

- Output Generating Module: This module prepares the query results based on the specified structure, such as a tabular structure, and puts them as the output.

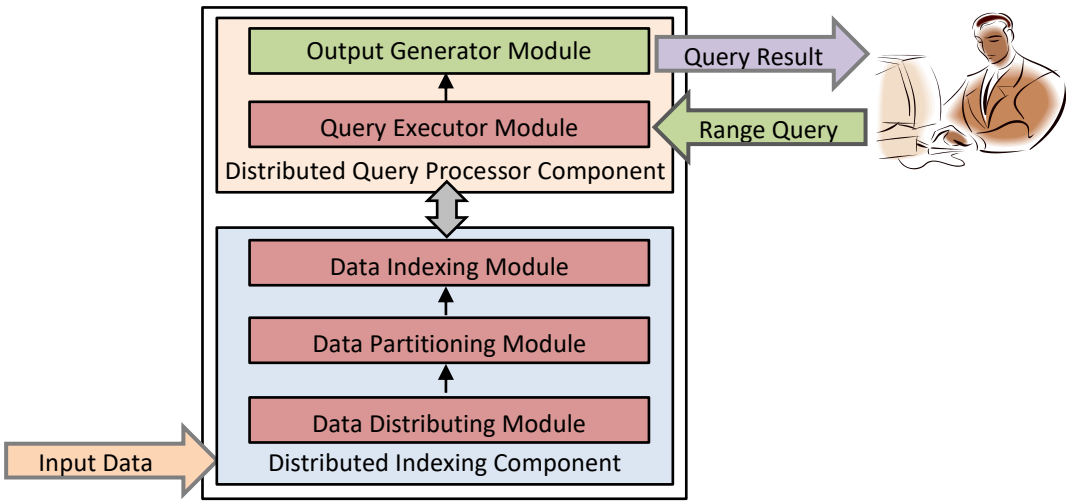

Fig. 1 Our Proposed System Architecture

3.2 Problem formulation

Here, we formulate how to model the streaming data, our proposed indexing method, and range queries.

3.2.1 Data Model

A streaming data is a sequence of data elements that is continuous, unbounded, unpredictable, fast and time-variant, which is represented as $s =< s_1, s_2, \dots >$ [44].

Definition 1 (Streaming Data) [18]: Consider a data stream S consisting of an unbounded data set of (tuple, timestamp) pairs: $S = \{s_i | s_i = (t_i, ts_i), i \in [1, +\infty]\}$. Given a set of attributes $A = \{a_1, a_2, \dots, a_n\}$, let d($a_j$) denote the data domain of an attribute a_j where $1 \leq j \leq n$. Thus, each tuple consists of a set of (attribute, value) pairs: $t_i = \{(a_j, v_j),\ j \in [1, n] \wedge v_j \in d(a_j)\}$.

3.2.2 Indexing Model

Our Proposed indexing method is based on a B+Tree index structure. A B+Tree index structure consists of a root node, intermediate nodes and leaf nodes. The root node may be a leaf node or a node with two or more child nodes.

Definition 2 (DB+Tree): DB+Tree is a distributed B+Tree created by the map and reduce functions in the Apache Spark framework. Let M be the master node, and $W = \{W_1, W_2, \dots, W_n\}$ be the set of worker nodes in the Spark cluster. Node M interconnects with all the worker nodes W_i in W. On the Apache Spark cluster, each worker node can have one or more partitions of streaming data. If the set of partitions are shown as $P = \{P_1, P_2, \dots, P_n\}$, each partition P_i contains a $B + Tree_{P_i}$. Therefore, DB+Tree contains the sum of B +Tree as shown in Equation 1.

$$DB + Tree = \sum_{i=1}^{N_P} \left\{ B + Tree_{P_1}, B + Tree_{P_2}, \dots, B + Tree_{P_{N_P}} \right\} \quad (1)$$

where N_P denotes the number of partitions in the Spark cluster.

There are several properties in our proposed distributed B+Tree indexing method as follows:

Property 1: The total number of partitions in the Spark cluster is equal to

$$N_P = \sum_{i=1}^{m} n \quad (2)$$

where m is the number of worker nodes and n is the maximum number of partitions per worker node in the Spark cluster.

By partitioning the streaming data, each tuple T of streaming data S is assigned to a partition based on the key of each tuple. We use range partitioning to place all tuples which are in the same range based on the tuple key in a partition. A distributed memory space is a set of ranges $\{[S, r_1], [r_1, r_2], \dots, [r_n, E]\}$, where S and E are the start and end of the range, respectively, and $r_1, r_2, \dots, r_n$ are the partition points.

Property 2: Based on Definition 2 and Equation 1, the number of index trees represented by N_{BT} is equal to the number of partitions represented by N_p in the Spark cluster.

$$N_{BT} = N_P \quad (3)$$

Property 3: The total number of processing cores that are called Executors represented by N_E is equal to the total of the processing cores of all worker nodes in the Spark cluster.

$$N_E = \sum_{i=1}^{m} N_C = m \times N_C \quad (4)$$

where m is the number of worker nodes and N_C represents the number of processing cores in each worker node.

Lemma 1. The maximum number of B+Trees is equal to the number of Executors in the Spark cluster.

Proof. Since each partition is assigned at least one processing core or Executor, the total number of partitions will be equal to the number of Executors, which is calculated in Equation 4. Therefore, it can be concluded that $N_P = N_E$ and from Equation 3 we can also conclude that $N_{BT} = N_E$. □

3.2.3 Query Model

In range queries, the search is performed on one of the attributes of data tuples in a range [L, R].

Definition 3 (Range Query) [18]: For a set of attributes $A = \{a_1, a_2, \dots, a_n\}$, the form of a range query is defined as: $q = (I(a_1), I(a_2), \dots, I(a_n))$, where $I(a_i)$ represents the query range for attribute a_i, i.e., $I(a_1).\min \geq d(a_1).\min \wedge I(a_1).\max \leq d(a_1).\max$.

Definition 4 (Result Set) [18]: For a set of continuous range queries $Q = \{q_1, q_2, \dots, q_k\}$, the result set $RS(Q)$ is defined as: $RS(Q) = \bigcup_{i=1}^{m} V_{qi}$, where V_{qi} represents the streaming data that are valid for a range query qi.

3.3 Algorithmic Structure of Our Proposed Indexing Method

Our proposed distributed B+Tree indexing method is designed to be used in the Apache Spark framework. In this subsection, we explain how to create a set of small B+Tree indexes using map-reduce programming model in the Apache Spark framework and how to employ them for distributed processing range queries over data stream.

3.3.1 Distributed Index Structure

Fig. 2 shows how a distributed B+Tree index is created in our proposed indexing method. It can be explained as follows: After Kafka receives the streaming data in short time slots and distributes it to worker nodes in the Spark

cluster, the distributed data is divided into a number of partitions. Then, a B+Tree is created as an index in each data partition, and the data in each partition is inserted into the created B+Tree index of that partition. Then, the streaming data is stored as tuples in the leaves of the B+Tree and the first column in each tuple is the indexing key. Algorithm 1 summarizes the steps of creating a distributed B+Tree index.

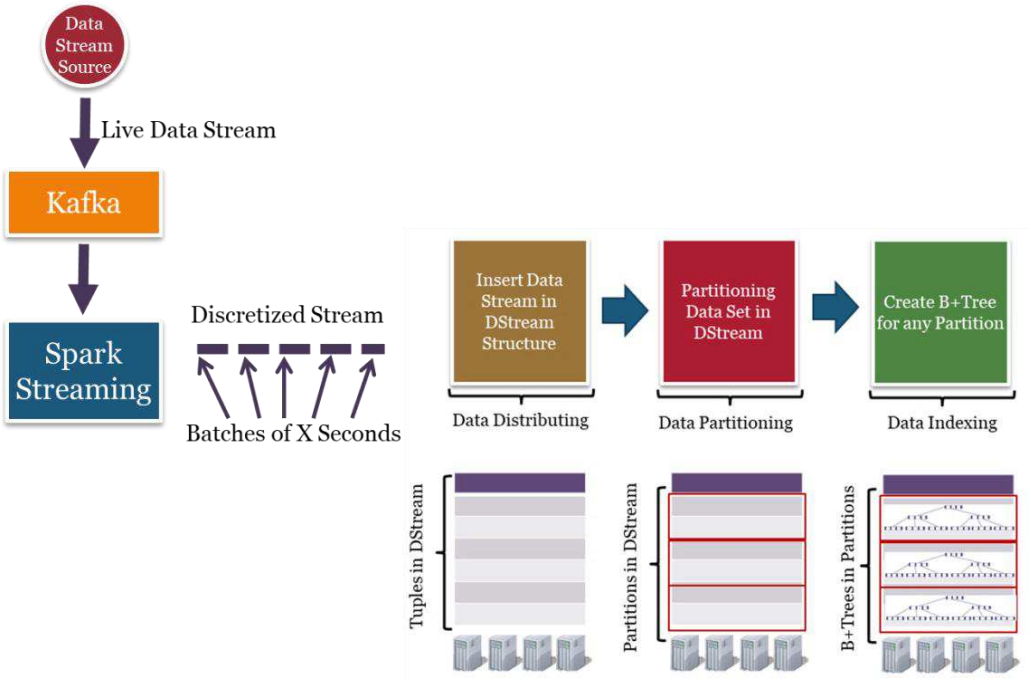

Fig.2 Process of Creating a Distributed B+Tree Index

Algorithm 1: Distributed B+Tree

Input:

Stream Tuple ST<c1,c2,c3,…,cn >

Output:

DB+Tree Index

DistributedBPlusTree(ST){

1. Master node gets stream tuples and distributes them on worker nodes in the form of Spark RDD Structure;
2. Master node partitions RDDs using rangePartitioner() method;
3. In each partition, a B+Tree is created in parallel using mapToPair() method;
4. }

In Algorithm 1, first, the master node receives streaming data tuples and distributes them to the worker nodes in the form of RDD structure (Line 1). Then, the data is partitioned by the master node using rangePartitioner() method (Line 2). Finally, an index is created in each partition (Line 3). In our proposed indexing method, the process of index creation is done based on the distributed programming model in the Apache Spark, which uses a transformation function to transfer processing operations on the distributed data in an RDD and applies the desired operation on the distributed data. Here, the process of index creation can be performed in parallel form using mapToPair() method which results in the mapping of each partition to a B+Tree index.

Algorithm 2 is proposed to insert data into B+Tree index in each partition. The parameters that must be set as input to create a B+Tree index are: Branching Factor bF and Key Index keyIndex. The branching factor bF specifies how many index keys can be placed at each level of the root and internal nodes in the index tree. In Section 4, the effect of this parameter on the index creation time is investigated. Also, the key index keyIndex refers to the column number of the streaming data tuple, which should be set as the indexing key and therefore, data stream's tuples are placed in the leaves of the B+Tree index based on the index key.

Algorithm 2: Insert Data in B+Tree Index

```
Input:
        branchingFactor bF
        keyIndex // A positive decimal number that points to index key;
Output:
        Distributed B+Trees DBT
DistributedBPlusTreeDataInsert(bF,keyIndex){
1.  RDD DDataSet ← kafka.datastream();
2.  RDD Partitions ← DDataSet.partitionby(rangePartitioner);
3.  RDD DBT ← Partitions.mapToPair(bF, keyIndex);
4.  Return DBT;
5.  }

mapToPair(bF, keyIndex){
1.  Create BPlusTree with parameters (bF, keyIndex));
2.  While (End of Partition) {
3.     Insert Partitions.row in the appropriate leaf;
4.     currentNode ← leaf;
5.     While (currentNode overflow) {
6.        split the currentNode into two nodes on the same level,
                  and promote median key up to the parent of currentNode;
7.        currentNode ← parent of currentNode;
8.     }//End While
9.     Partitions.nextRow();
10.  }//End While
11. }//End mapToPair() Method
```

In Algorithm 2, first, streaming data is received by Kafka and distributed to worker nodes in an RDD called DDataSet (Line 1). In the next step, the streaming data is partitioned by calling the partitionby() method on the worker nodes, and the partitions are created in another RDD format called Partitions (Line 2). Each row of the partition represents a streaming data tuple. By utilizing the mapToPair() method, B+Trees are created in parallel in another RDD format called DBT for each partition (Line 3). The mapToPair() method is specified as an independent method. In the first step of this method, an index tree is created based on the branching factor and index key (Line 1). Then, each row of the partition is inserted into the tree corresponding to each partition (Lines 2-10). The process of insertion of data into B+Tree is performed in Lines 3-8.

Lemma 2. The time complexity of the index creation algorithm in our indexing method is $O(nlogn)$.

Proof. Assume n be the number of streaming data tuples arrived in each time slot. Since the time complexity of the insertion in the B+Tree is $O(logn)$ and the data insertion in our proposed indexing method is performed in parallel for each partition, the time complexity of the index creation in our proposed approach is $O(nlogn)$. □

3.3.2 Distributed Range Search Algorithm

The process of finding the results of a range query over streaming data based on our proposed distributed B+Tree indexing method consists of two phases. After receiving the range query, in the first phase, which is the map phase, the search operation is performed in parallel on the B+Tree Index in each partition. In the second phase, which is the reduce phase, the search results are aggregated in each partition and sent to the master node to prepare the query results. Fig. 3 shows how this process can be done.

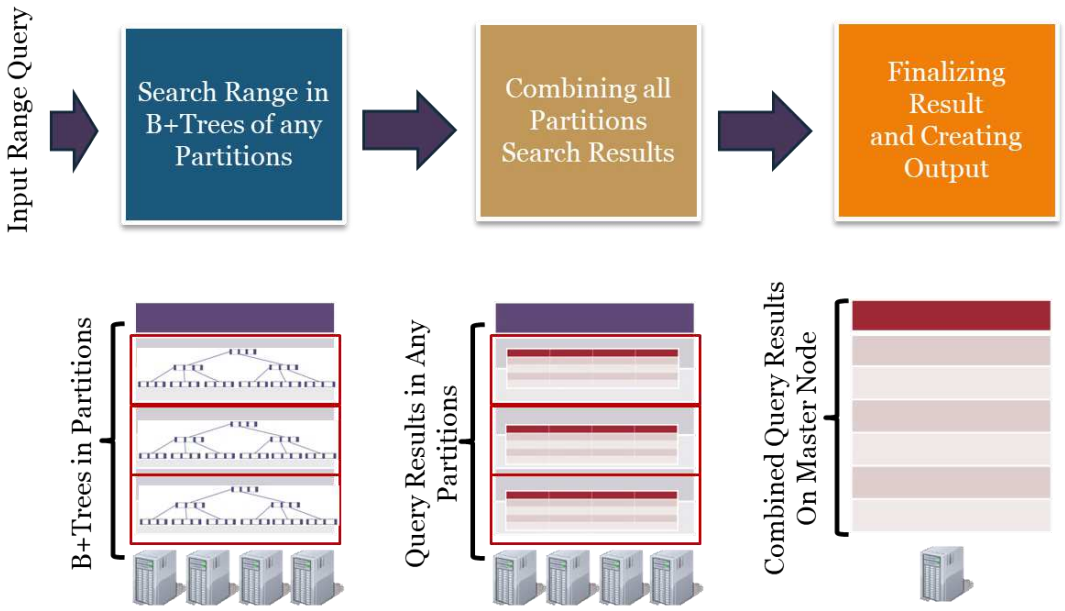

Fig. 3. Process of Finding the Results of a Range Query

Algorithm 3 is designed to process a range query over streaming data using our proposed distributed B+Tree indexing method. As shown in Algorithm 3, the range query Q and distributed B+Tree indexes DBT are as the inputs and the query result QR is the output of the algorithm. In this algorithm, the map() method with the input parameters the lower bound and the upper bound of the search is called in parallel form over all B+Tree indexes (Line 1). After applying the map() method, the search result in each B+Tree is placed in an RDD called DQR (Line 1). Finally, the results obtained in DQR, which are distributed on the worker nodes, must be aggregated and sent to the master node. Therefore, the combineByKey() method as a transformation function is applied to the RDD as the reduce function. Then, the query results are returned to the master node as an integrated list (Line 3). The map method performs the basic search operation in the B+Tree. This method gets the lower bound and upper bound of the search as inputs. First, it selects the root node (Line 1). Then, in a loop, the upper and lower bounds of the search are compared to the root keys (Lines 3-10) and the correct internal node is selected. The same process is repeated for the internal nodes to identify the leaf nodes that contain the searched data. Finally, the data that matches the search range is retired and placed in the leaf nodes of the index tree (Lines 11-14) and the retrieved data values are placed in the Result variable (Line 12).

Algorithm 3: Distributed Range Search

Input:

```
        Query Q
        Distributed B+Trees DBT
Output:
        Query Result QR
DistributedBPlusTreeRangeSerach(){
1.   RDD DQR←DBT.map(Q.lowerbound,Q.upperbound)
2.   QR ← DQR.combinebykey();
3.   Return QR;
4.  }

map(QueryLowerBound,QueryUpperBound){
1.    Node ← Root;
2.    Result ← Null;
3.    While (Node is not a leaf node) {
          //Let i be least number that Lower_Bound ≤ Ki
4.        If (such a number  i does not exist)
5.            Node← last non-null pointer in Node;
6.        else if (Lower_Bound ≤ Node.Ki )
7.            Node ← Pi+1;
8.        else
9.            Node ← Node.Pi;
10.   }//End While;
11.   For (i that Pi is not null) & (Lower_Bound ≤ Ki) & (Upper_Bound ≥ Ki){
12.       Result ← Result + Node.Pi;
13.       i ← i+1;
14.   }//End For
15. }//End map() Method
```

Lemma 3. The time complexity of the Distributed Range Search algorithm is $O(logn)$.

Proof. Considering that the time complexity of the search operation in B+Tree is $O(logn)$ [45-47], and also the search operation is performed in parallel on the B+Tree index of each partition in our proposed indexing method, therefore, the time complexity of Algorithm 3 is O(logn). □

Fig. 4 shows the serial and parallel parts of the Distributed Range Search Algorithm. As shown in Fig. 4, range queries are received serially through the master node. Then, they are distributed in parallel using the B+Tree of each partition. Next, they are executed by the worker nodes. Finally, the query results obtained from the worker nodes are aggregated and returned to the master node.

Fig. 5 shows a more detailed form of query parallelism in this regard. In Fig. 5, Executor is a processing unit for distributed query processing. The number of executors depends on the number of worker nodes as well as the number of processor cores in each worker node as explained in Section 3.2. For example: If the number of worker nodes is 4 and each worker node has 4 processing cores, the number of executors will be 16, which indicates the degree of parallelism. As shown in Fig. 5, an instance of each range query is distributed among Executors and each Executor executes the range search operation using the B+Tree structure called Data Fragment in Fig. 5.

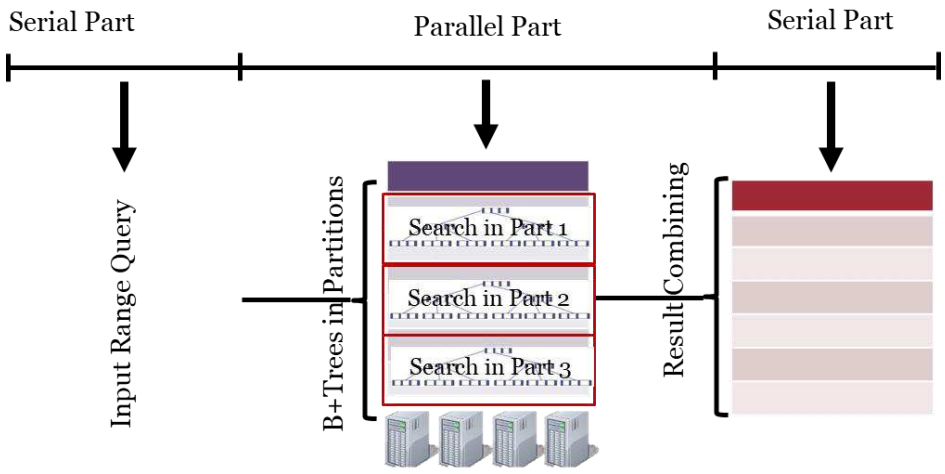

Fig 4 Serial and Parallel Parts in Distributed Range Search Algorithm

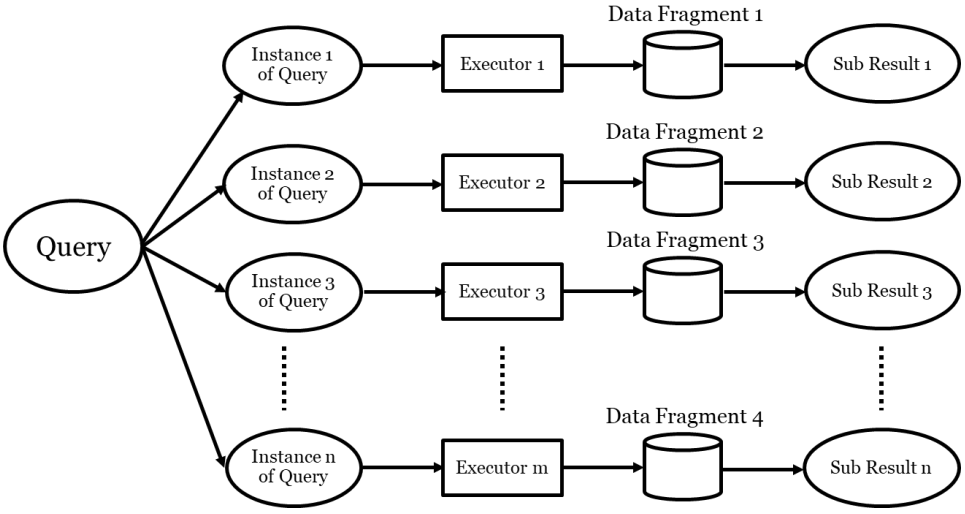

Fig 5 Parallelism Form of Range Search Algorithm

4 Performance Evaluation

In this section, the performance of our proposed distributed B+Tree indexing method to process range queries over streaming data is evaluated by performing several experiments.

4.1 Experimental settings

We used a cluster including 9 nodes for evaluation of our proposed indexing method to process range queries over streaming data. Each node had 4 processing cores with a speed of 2.20 GHz, a main memory with a capacity of 4 GB, a hard disk of SSD type with a capacity of 120 GB and a network interface with a speed of 1 Gbps. The operating system installed on each node was Linux Ubuntu 19.10 x64. The distributed processing platform used in our experiments was Apache Spark 2.4.4 with the Spark Streaming library for streaming data processing. We used Kafka 2.6.13 to manage the streaming data and HDFS file system to store the required files. In our experiments, we used a node from the cluster as the master node and the other 8 nodes as the worker nodes. Fig. 6 shows the structure of the Spark cluster in our experiments.

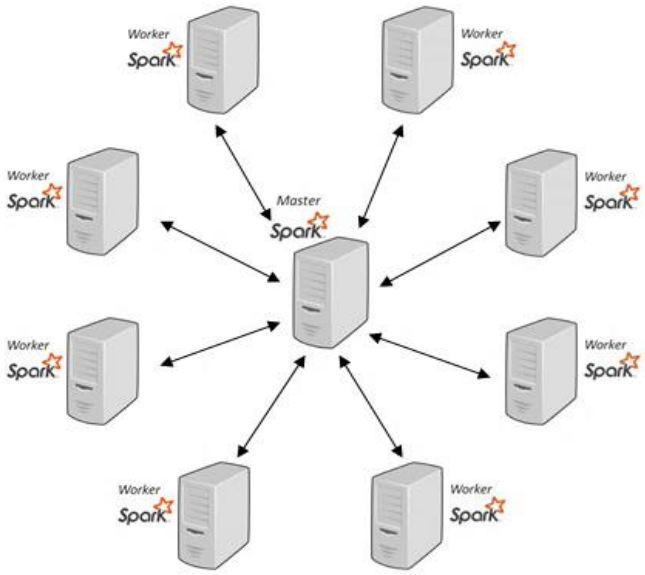

Fig. 6 Structure of the Spark Cluster in Our Experiments

We ran the Driver program on the master node and the Worker program on the worker nodes. The Driver program was responsible for managing the Spark cluster and distributing the tasks to the worker nodes, as well as collecting and aggregating the query results from the worker nodes. The Worker program was responsible for receiving tasks from the Driver program and processing and returning the query results to the Driver program. In our experiments, the maximum number of executors was 32, and the tasks were distributed among these executors by the Driver program run on the master node.

We used the dataset used in [48] in our experiments[1]. This dataset is about famous movies including the following fields: movie name, year of production, and ratings by fans. This data set has more than 10 million records stored as a CSV file.

To measure the performance variation based on the types of data distribution, we used three data distributions Uniform, Skew, and Hyper Skew in our experiments. We produced streaming data for each data distribution using the Spark Streaming library. The time interval of each DStream for generating data was 30 seconds. In the uniform distribution, the records were uniformly placed in the streaming data based on the key field with a probability of $1/n$, where n was the number of records. In the skew distribution, data distribution was normal with zero mean and symmetrical skewness, and in the hyper skew distribution, data distribution was normal, but with non-zero mean and asymmetrical skewness. In other words, the data distribution was either skewed to the right or skewed to the left in this case.

To measure the performance of our proposed indexing method to create index trees in the Spark cluster, we used the index creation time as a metric in our experiments. This metric is dependent on independent parameters: the type of data distribution, the number of records in the dataset, the number of nodes in the Spark cluster, the number of processor cores in each worker node, the number of Executors in the Spark cluster, the number of partitions in the Spark cluster, and the value of branching factor in index tree [18, 20, 23]. Table 2 shows these independent parameters with the values in our experiments. Each experiment was repeated 3 times and the average value was calculated from the obtained results as the final value.

[1] https://grouplens.org/

Table 2 Independent Variable and their Values in Our Experiments

Parameter	Values
Type of Data Distribution	Uniform, Skew, Hyper Skew
Number of Dataset Records	1 M, 2 M, 4 M, 8 M (Records per Minutes)
Number of Partitions	1, 2, 4, 8,16, 32
Branching Factor	64, 128, 256, 512
Number of Nodes	1, 2, 4, 8
Number of Processor Cores per Node	4
Number of Executors	4, 8, 16, 32

To measure the storage cost of our proposed indexing method, we used the total index size as a metric in our experiments. It is defined as follows:

$$Total\ Index\ Size\ =\ Size\ (Total\ Index\ Keys\ per\ B+Tree)\ +\ Size\ (Recodes\ in\ the\ Data\ Stream) \quad (5)$$

where Total Index Keys per B+Tree is calculated by Equation 6 as follows:

$$\begin{aligned} &Total\ Index\ Keys\ per\ B+Tree \\ &=\ Number\ of\ (Root\ Keys)\ +\ Number\ of\ (Internal\ Nodes\ Keys) \\ &+\ Number\ of\ (Leaf\ \ Nodes\ \ Keys) \end{aligned} \quad (6)$$

To measure the storage overhead in our proposed indexing method, we used the memory index overhead as a metric in our experiments which is calculated using Equation 7.

$$Memory\ Index\ Overhead = \frac{Size\ of\ Data}{Total\ Index\ Size} \quad (7)$$

To measure the performance improvement of our proposed indexing method to process range queries over data stream, we used two metrics speed up and scale up in our experiments. They are defined as follows [49]:

$$Speed\ Up = \frac{Elapsed\ Time\ on\ Single\ Node}{Elapsed\ Time\ on\ Multi\ Node} \quad (8)$$

$$Scale\ Up = \frac{Single\ Node\ Elapsed\ Time\ on\ Small\ Data\ Set}{Multi\ Node\ Elapsed\ Time\ on\ Large\ Data\ Set} \quad (9)$$

4.2 Experimental Results

In our experiments, we first considered the impact of different parameters for creating index trees. Then, we investigated the impact of different parameters for processing of range queries over streaming data using our proposed distributed B+Tree indexing method. Finally, we compared our proposed indexing method with the ACBBI indexing method [20].

4.2.1 The Impact of Different Parameters for Creating Index Trees

Here, we explained the impact of different parameters in the creation of index tree.

4.2.1.1 The Effect of the Number of Partitions on the Index Creation Time

Fig. 7 shows the effect of number of partitions on the index creation time for three different data distribution scenarios when the number of records in the data stream was 8M records per minute, the branching factor was 64, and the number of executors in the Spark cluster was 32. As shown in Fig. 7, increasing the number of partitions for the uniform data distribution and the skew data distribution has a positive effect, and with the increasing the number of partitions, the index creation time reduces. However, the results are different for the hyper skew data distribution and the index creation time is significantly increased by increasing the number of partitions. In our proposed indexing method, when the number of partitions is few, the number of index trees is also few, and as a result, the size of the index tree becomes larger, and vice versa. In the hyper skew data distribution, we have a lot of skewness and therefore, some partitions have a few numbers of data records while other partitions may have a large number of data records. Therefore, the index creation time will be more for partitions including the large number of data records. As shown in Fig. 7, in the case that the number of partitions is ¼ of the number of executors in the Spark cluster, the index creation time is almost the same for all three data distributions. Therefore, we can set the number of partitions in the hyper skew data distribution equals to ¼ the number of executors to have better index creation time.

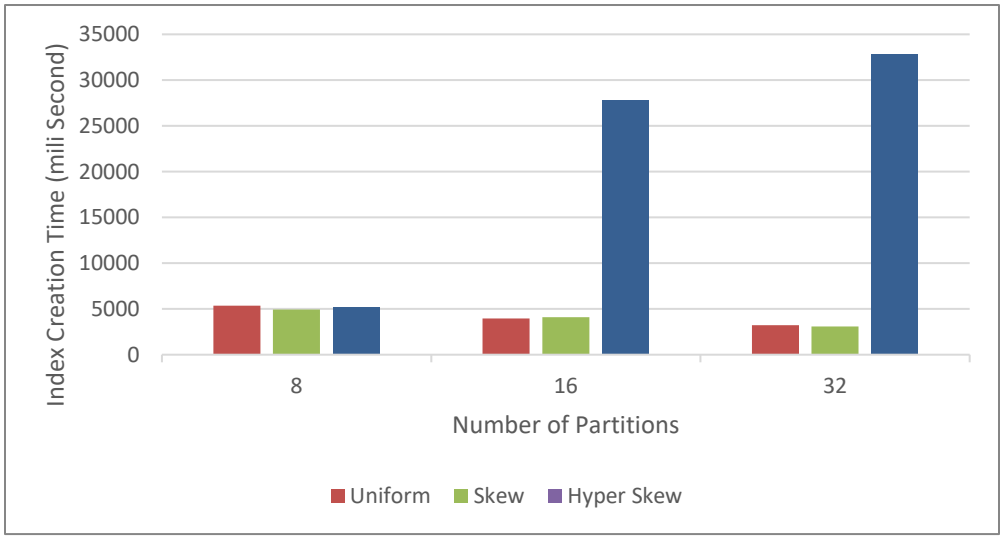

Fig. 7 Effect of the Number of Partitions on Index Creation Time

4.2.1.2 The Effect of Branching Factor on the Index Creation Time

Fig. 8 shows the effect of branching factor on the index creation time for three different data distribution scenarios when the number of records in the data stream was 8M records per minute, the number of partitions was 8, and the number of executors in the Spark cluster was 32. Generally, if the branching factor of the B+Tree is equal to n and all the nodes and keys of the index tree are used in the worst case, in the first level of the index tree, there is a root node with n keys and $(n+1)$ pointers, in the second level of the index tree, there are $(n+1)$ internal nodes containing n keys and $(n+1)$ pointers to nodes in the third level, and in the third level (last level), $(n+1)(n+1)$ leaf nodes containing n key-value pairs. Therefore, when the branching factor is set to 64, the maximum number of records that can be inserted into the index tree is equal to $64 \times 65 \times 65 = 270{,}400$ records, which is greater than $125{,}000$. If the branching factor is changed to 128, 256, and 512, only the number of records greater than 270,400 can be inserted into the index tree and it will not affect the index creation time. However, the index creation time has increased in the case of hyper skew data distribution compared to other data distributions. It is due to the high skewness of the data and the existence of a large number of records in some partitions compared to other partitions.

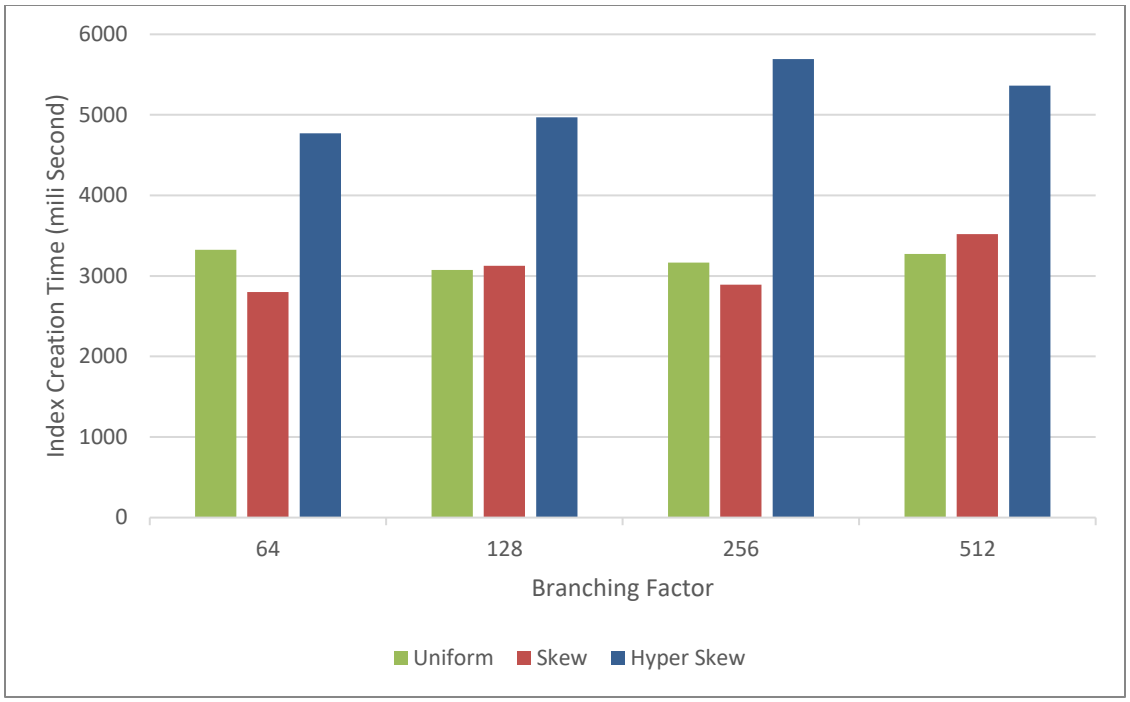

Fig. 8 Effect of Branching Factor on the Index Creation Time

4.2.1.3 Speed Up in the Process of Index Creation

Fig. 9 shows the speed up during the process of index creation in our proposed indexing method for three different data distribution scenarios when the number of records in the data stream was 8M records per minute, the number of partitions was 8, the branching factor was 64, and the number of worker nodes was varied. We set the number of executors to be four times the number of worker nodes in our experiments. As shown in Fig. 9, the pattern of speed up diagram is almost the same for all data distributions since the data records in the same range are placed together in each partition and therefore, the type of data distribution has no effect on the index creation time.

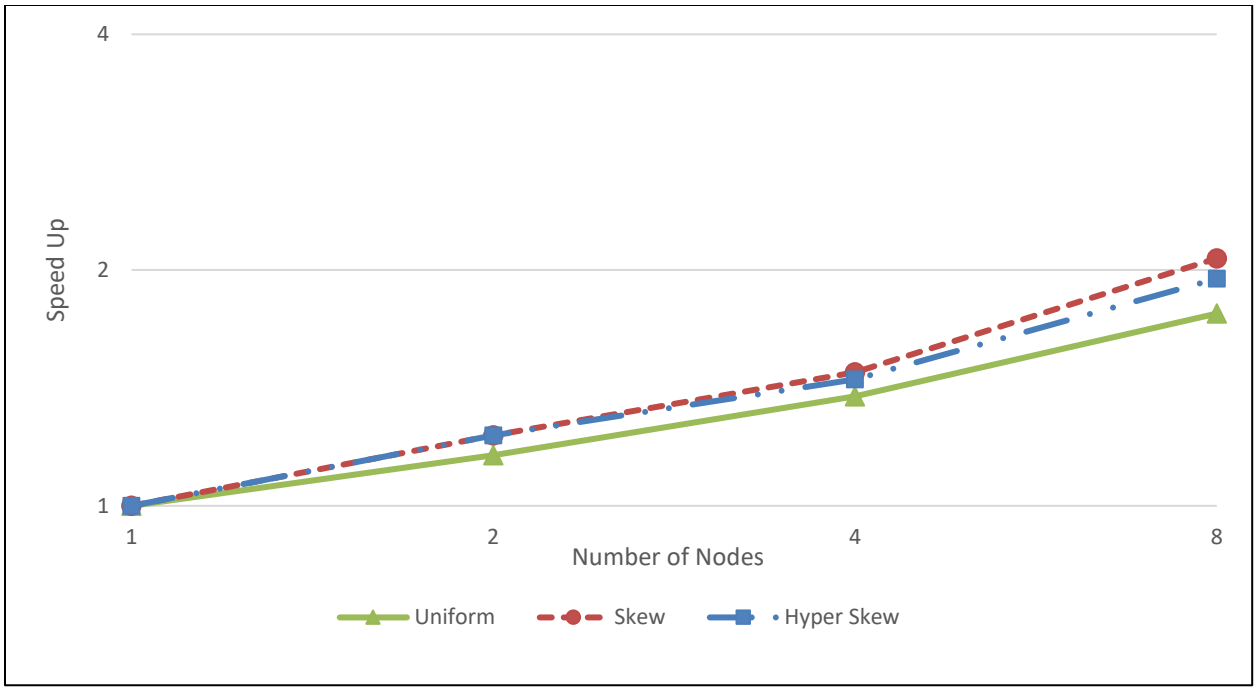

Fig. 9 Speed Up in the Process of Index Creation

4.2.1.4 Scale up in the Process of Index Creation

Fig. 10 shows the scale up during the process of index creation in our proposed indexing method for three different data distribution scenarios when the number of partitions was 8, the branching factor was 64, and the number of records were varied. We changed the number of records from 1M to 2M, 4M, and 8M records per minute in the data stream and the number of executors from 4 to 8, 16 and 32 in our experiments, respectively. As shown in Fig. 10, the pattern of scale up diagram is almost the same for all data distributions since the data records in the same range are placed together in each partition and therefore, the type of data distribution has no effect on the index creation time.

4.2.1.5 The Effect of the Number of Records on the Index Size

Fig. 11 shows the effect of number of records on index size of our proposed indexing method for three different data distribution scenarios when the number of executors in the Spark cluster was 32, the number of partitions was 8, the branching factor was 64, and the number of records per minute was varied. As shown in Fig 11, the growth rate of index size has doubled with 2, 4, 8, and 16 times the data volume per minute since the size of the index tree is dependent on the size of the data records in the data stream. By doubling the data volume, the size of the index has also doubled.

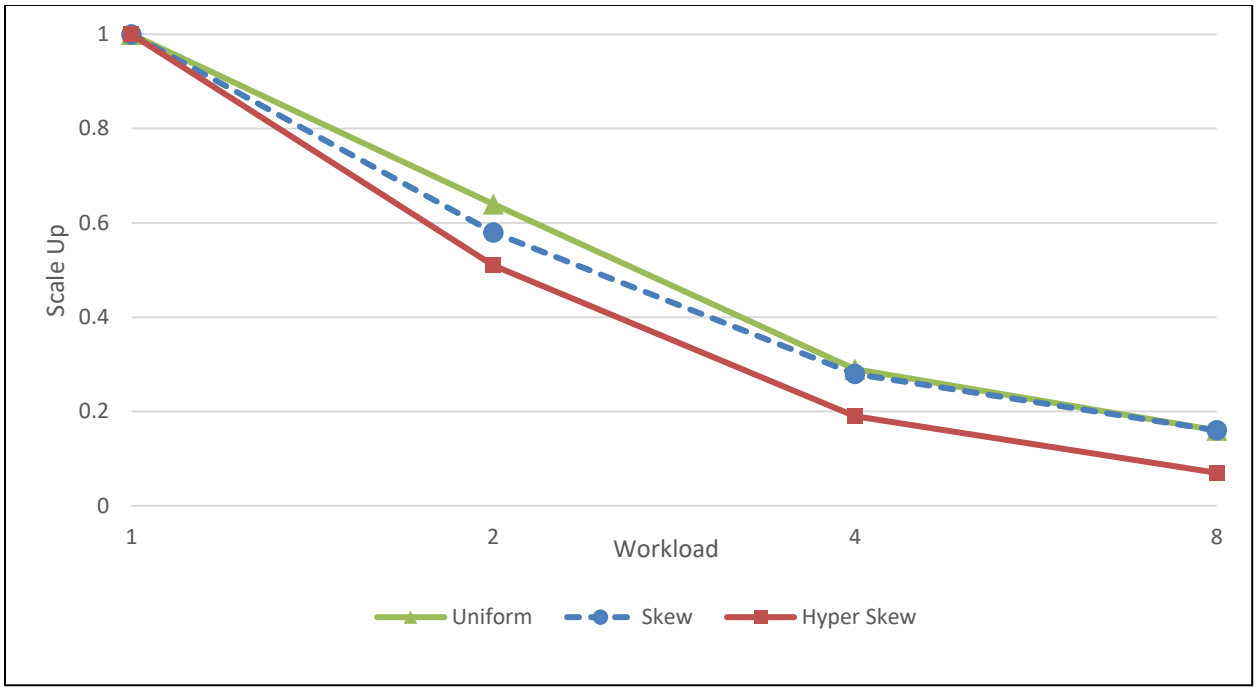

Fig. 10. Scale Up in the Process of Index Creation

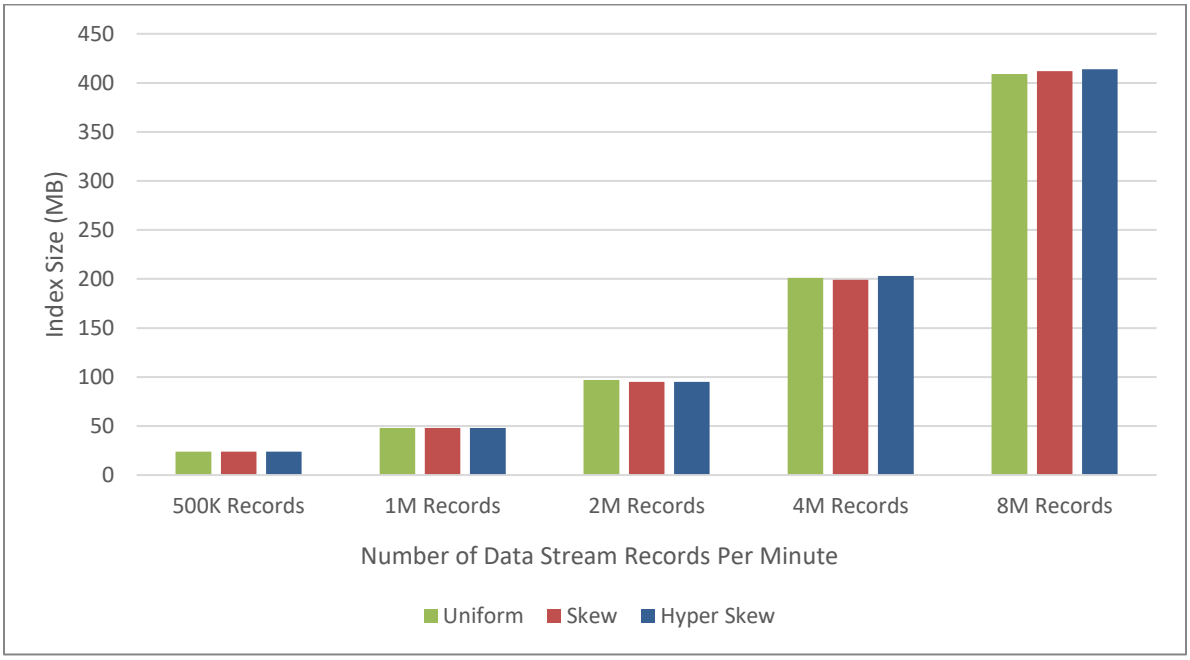

Fig. 11 Effect of Number of Records on Index Size

Fig. 12 shows the memory overhead in our proposed indexing method when the number of executors in the Spark cluster was 32, the number of partitions was 8, the branching factor was 64, and the number of records per minute in the data stream was varied. As shown in Fig 12, it is clear that our proposed indexing method needs 50% of storage space of data records in the stream to store the index data and obviously this amount of storage space does not increase

as the number of records in the data stream increases since the structure of the B+Tree is fixed and the amount of data does not change the structure of the index.

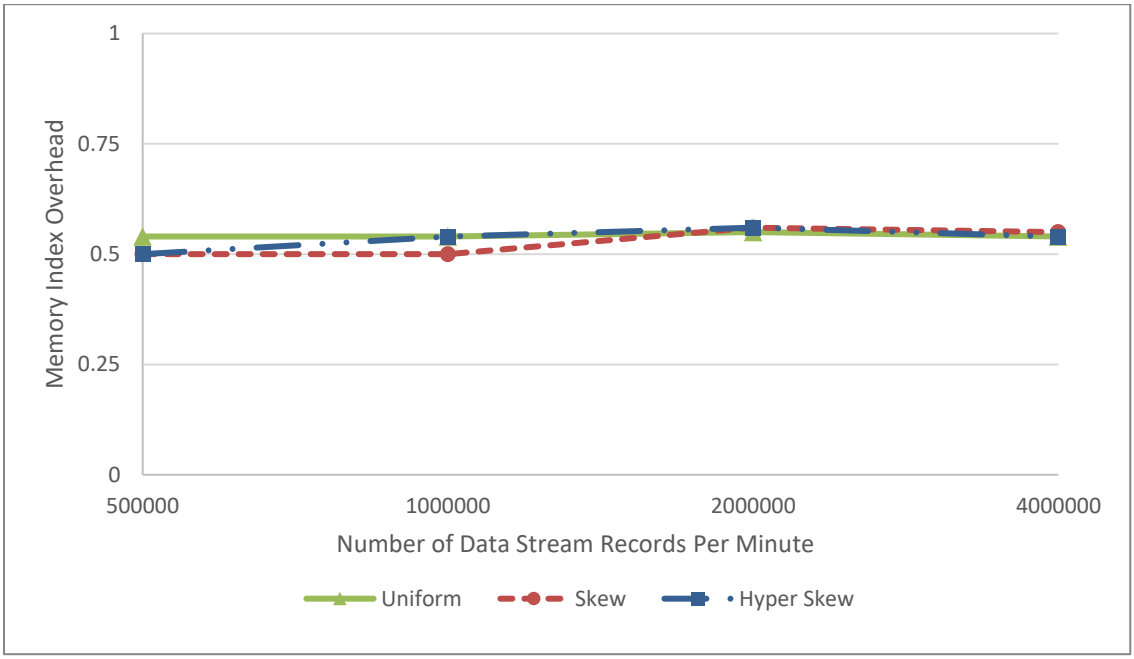

Fig. 12 Memory Overhead of Our Proposed Indexing Method

4.2.2 The Impact of Different Parameters on the Processing of Range Queries

We investigate the impact of different parameters in our proposed indexing method for processing range queries over streaming data.

4.2.2.1 The Effect of the Number of Partitions in the Cluster on Query Processing

Figs. 13, 14, and 15 show the effect of number of partitions in our proposed indexing method to process range queries over streaming data for three different data distribution scenarios to retrieve 10%, 50% and 100% of the data records, respectively. In our experiments, the number of records in the data stream was 8M records per minute, the branching factor was 64, and the number of executors in the Spark cluster was 32. As shown in Figs. 13, 14, and 15, increasing the number of partitions, especially in the uniform data distribution, has a negative effect on the query execution time in most cases. The reason is that the desired records may not exist in some index trees and therefore, many partitions should be checked in this regard. Moreover, it is clear that the type of data distribution does not significantly affect the query processing time. It is an advantage of our indexing methods as it can be used for datasets with different data distributions.

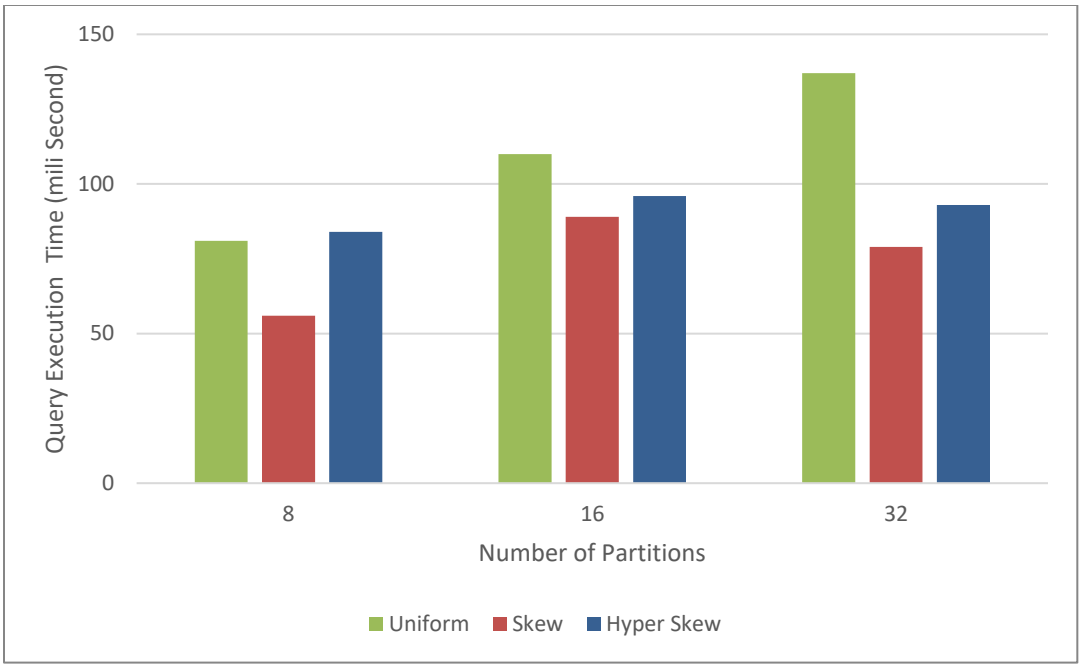

Fig. 13 Effect of Number of Partitions on Query Execution Time to Retrieve 10% of Data

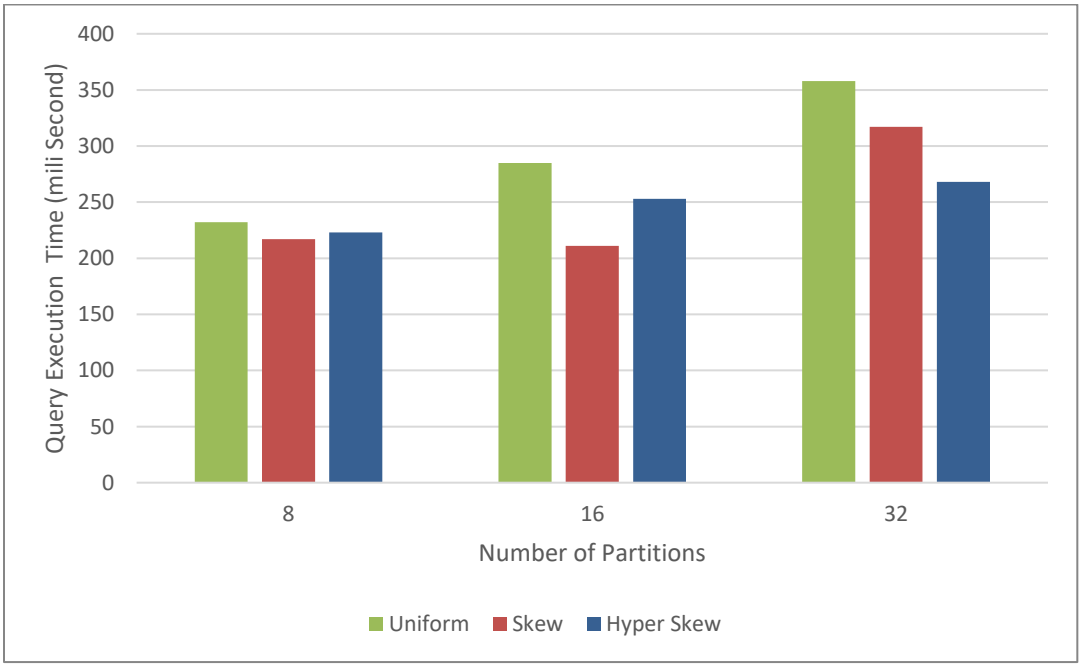

Fig. 14 Effect of Number of Partitions on Query Execution Time to Retrieve 50% of Data

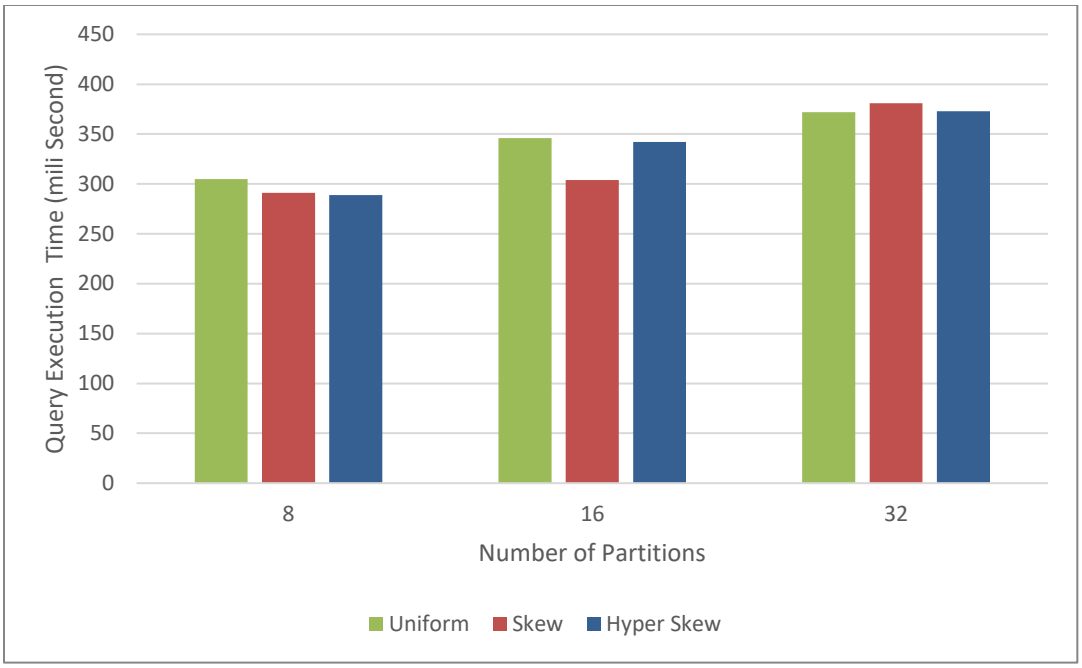

Fig. 15 Effect of Number of Partitions on Query Execution Time to Retrieve 100% of Data

4.2.2.2 Speed Up in Processing the Range Queries

Figs. 16, 17, and 18 show the speed up of our proposed indexing method in processing range queries for three different data distribution scenarios to retrieve 10%, 50%, and 100% of data records in the data stream, respectively. In our experiments, the number of records in the data stream was to 8M records per minute and the number of partitions and the branching factor were to 8 and 64, respectively. We set the number of executors to be four times the number of worker nodes in our experiments. As shown in Figs. 16, 17, and 18, the pattern of speed up diagram is almost the same for all data distributions since the data records in the same range are placed together in each partition and therefore, the type of data distribution has no effect on the index creation time. It is also clear that increasing the number of worker nodes has a positive effect on the query execution time for all experiments since by increasing the number of worker nodes, the number of executors also increases, and therefore, more executors are assigned to execute the range search algorithm on each index tree in the Spark cluster. This feature of our proposed indexing method can increase the degree of parallelism for processing the range queries over streaming data.

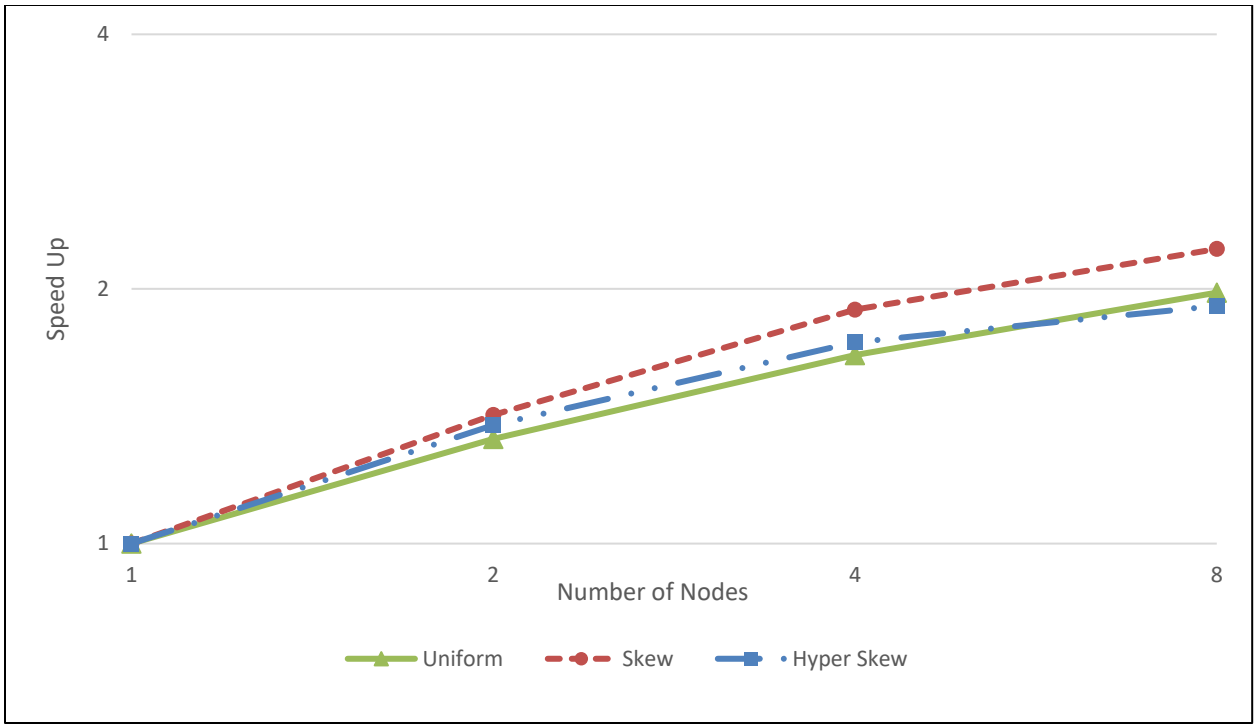

Fig. 16 Speed Up in Processing of Range Queries to Retrieve 10% of Data Records

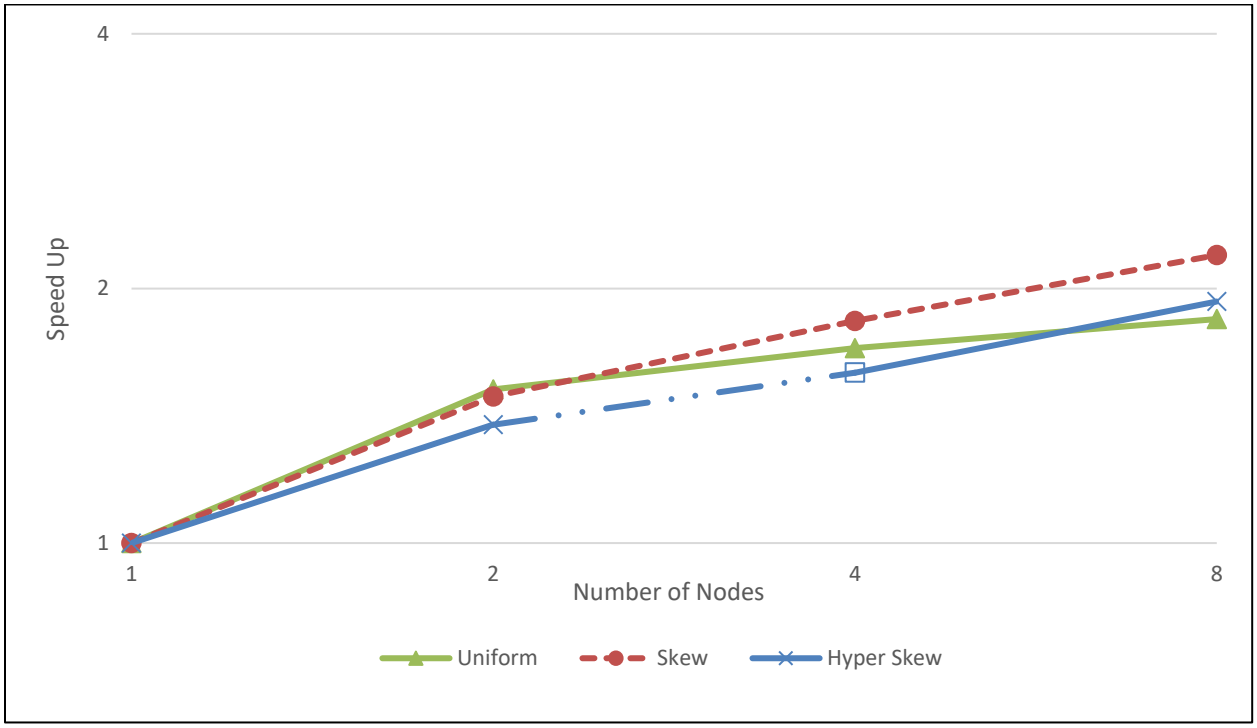

Fig. 17 Speed Up in Processing of Range Queries to Retrieve 50% of Data Records

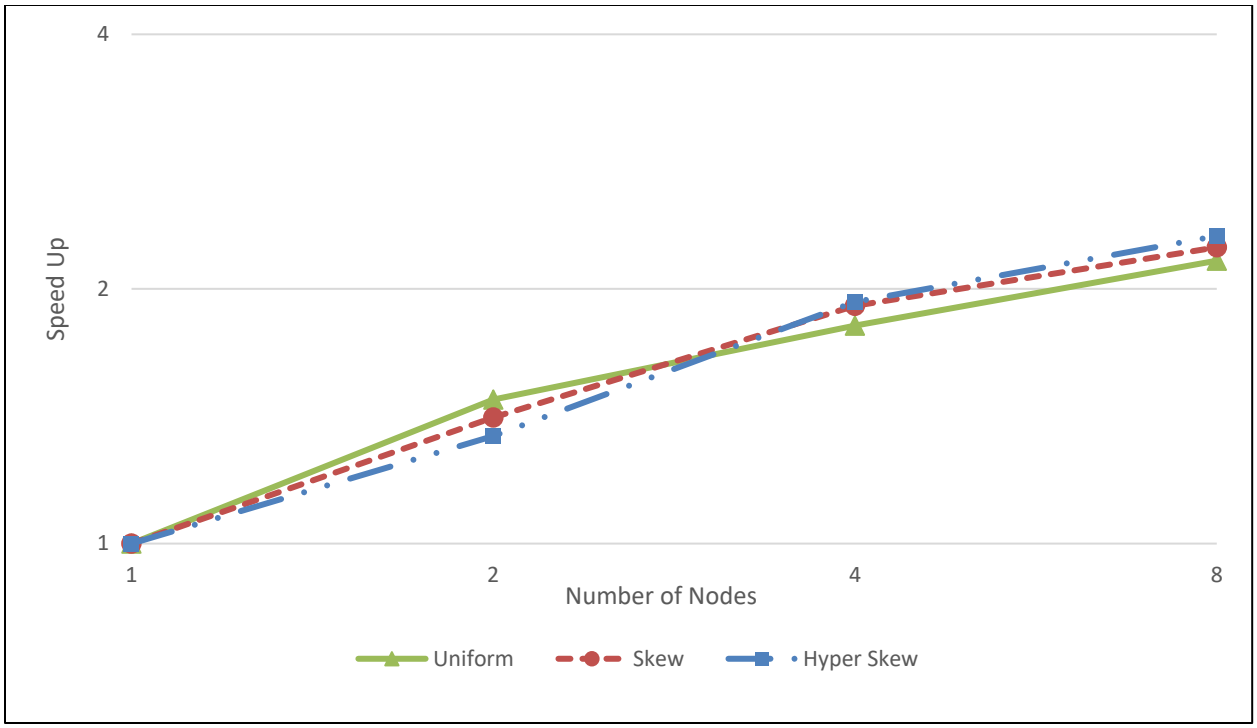

Fig. 18 Speed Up in Processing of Range Queries to Retrieve 100% of Data Records

4.2.2.3 Scale Up in Processing the Range Queries

Figs. 19, 20, and 21 show the scale up of our proposed indexing method in processing range queries for three different data distribution scenarios to retrieve 10%, 50%, and 100% of data records in the data stream, respectively. In our experiments, the number of partitions and the branching factor were to 8 and 64, respectively, and the number of records and the number of executors in the data stream were varied. We changed the number of records from 1M to 2M, 4M, and 8M records per minute in the data stream and the number of executors from 4 to 8, 16 and 32 in our experiments, respectively. We also set the number of executors to be four times the number of worker nodes in our experiments. As shown in Figs. 19, 20, and 21, the pattern of scale up diagram is almost the same for all data distributions in all experiments since the data records in the same range are placed together in each partition and therefore, the type of data distribution has no effect on the index creation time. It is also clear that our proposed indexing method for processing the range queries over the data stream is scalable since by increasing the number of worker nodes, the number of executors increases and therefore, the search algorithm can be executed in parallel form on the index tree of each partition with more executors.

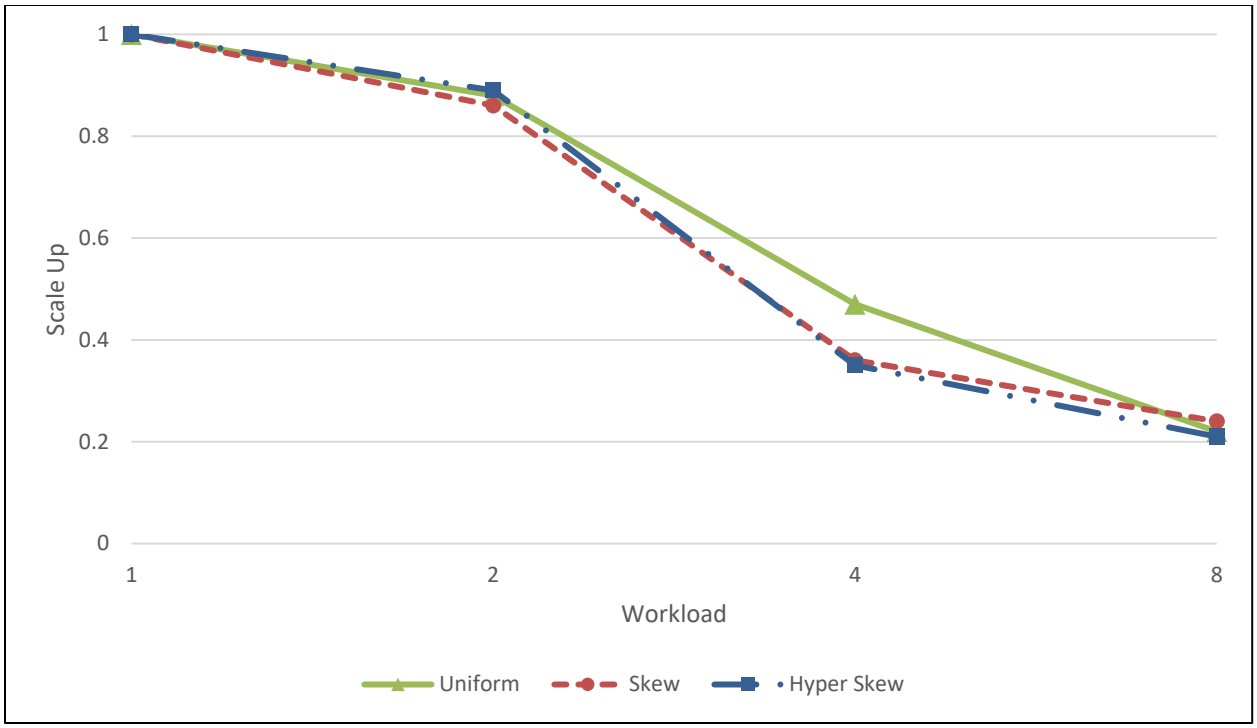

Fig. 19 Scale Up in Processing of Range Queries to Retrieve 10% of Data Records

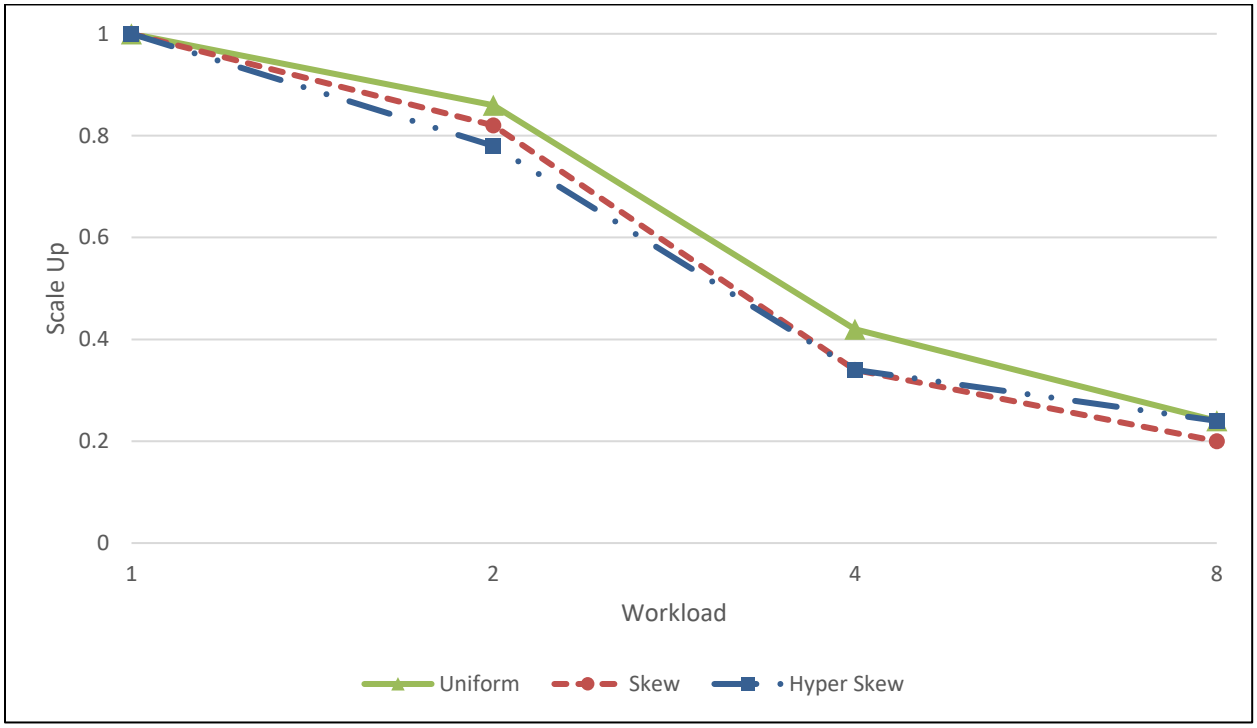

Fig. 20 Scale Up in Processing of Range Queries to Retrieve 50% of Data Records

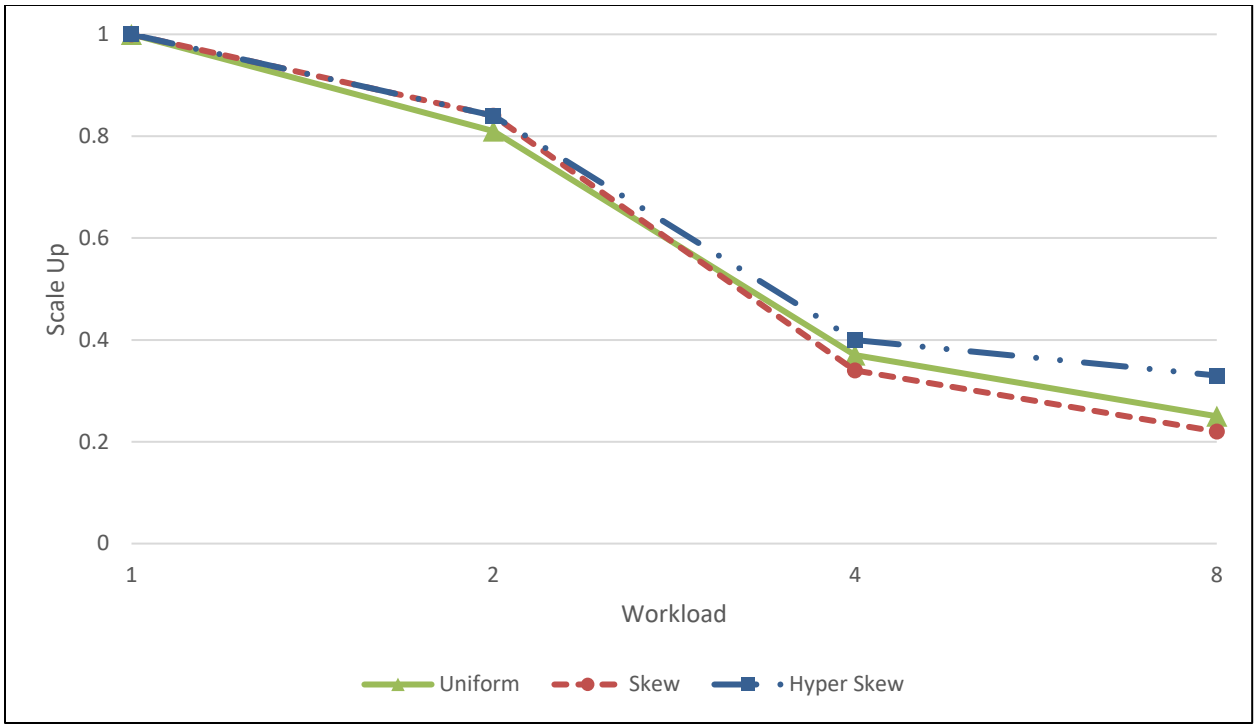

Fig. 21 Scale Up in Processing of Range Queries to Retrieve 100% of Data Records

4.2.3 Compassion with Other Indexing Methods

We compared our proposed indexing method with the ACBBI indexing method proposed in [20] because: 1) This indexing method is a B+Tree indexing method for processing range queries over streaming data. 2) It is more efficient than the indexing methods proposed in [16, 24, 38, 39], in terms of index maintenance cost and the query processing time. In addition, it should be noted that the indexing methods proposed in [18, 19, 23, 34-36, 42] cannot be compared with our proposed indexing method since they do not support streaming data or they are used for multi-dimensional spatial data. We implemented the ACBBI indexing method and used similar experimental settings for this indexing method to compare it with our proposed indexing method fairly.

4.2.3.1 Comparison in terms of Index Creation Time

Fig. 22 shows the index creation time of different indexing methods for three different data distributions. In our experiments, the number of records in the data stream was 2M records per minute, the number of partitions, the branching factor, and the number of executors were 4, 64, and 4, respectively. We only used only 1 node with 4 processing cores in our experiments. As shown in Fig. 22, it is clear our proposed indexing method is more efficient to create index trees compared to the ACBBI indexing method. This is due to the distributed structure of our proposed indexing method which can be created in parallel and distributed form.

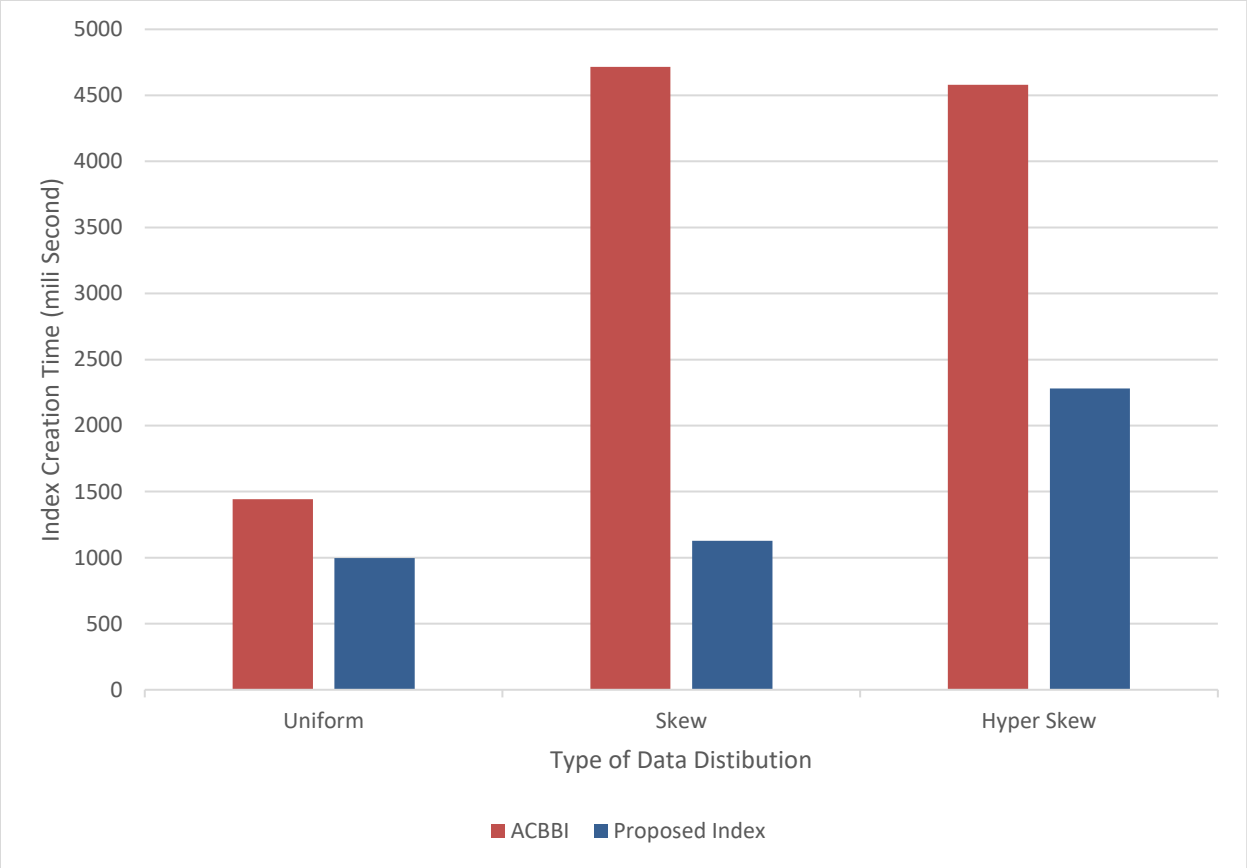

Fig. 22 Index Creation Time in Different Indexing Methods

4.2.3.2 Comparison in terms of Query Processing Time

To compare the query processing time of our proposed indexing method with the ACBBI indexing method to process range queries over streaming data, a set of experiments were performed. In all experiments, the number of records in the data stream was 2M records per minute, the number of partitions, the branching factor, and the number of executors were to 4, 64, 4, respectively. We only used 1 node with 4 processing cores in our experiments to retrieve a fraction of data records from 10% to 100% with an interval of 10%. Figs. 23, 24, and 25 show the time to process range queries over data stream in different indexing methods when the data distribution is uniform, skew, and hyper skew, respectively. As shown in Figs. 23, 24, and 25, it is clear that the time to process range queries for our proposed index method in all data distributions is less than that in the ACBBI indexing method since the process of query processing in our proposed indexing method can be done in parallel form even if there is a single node in the experiment.

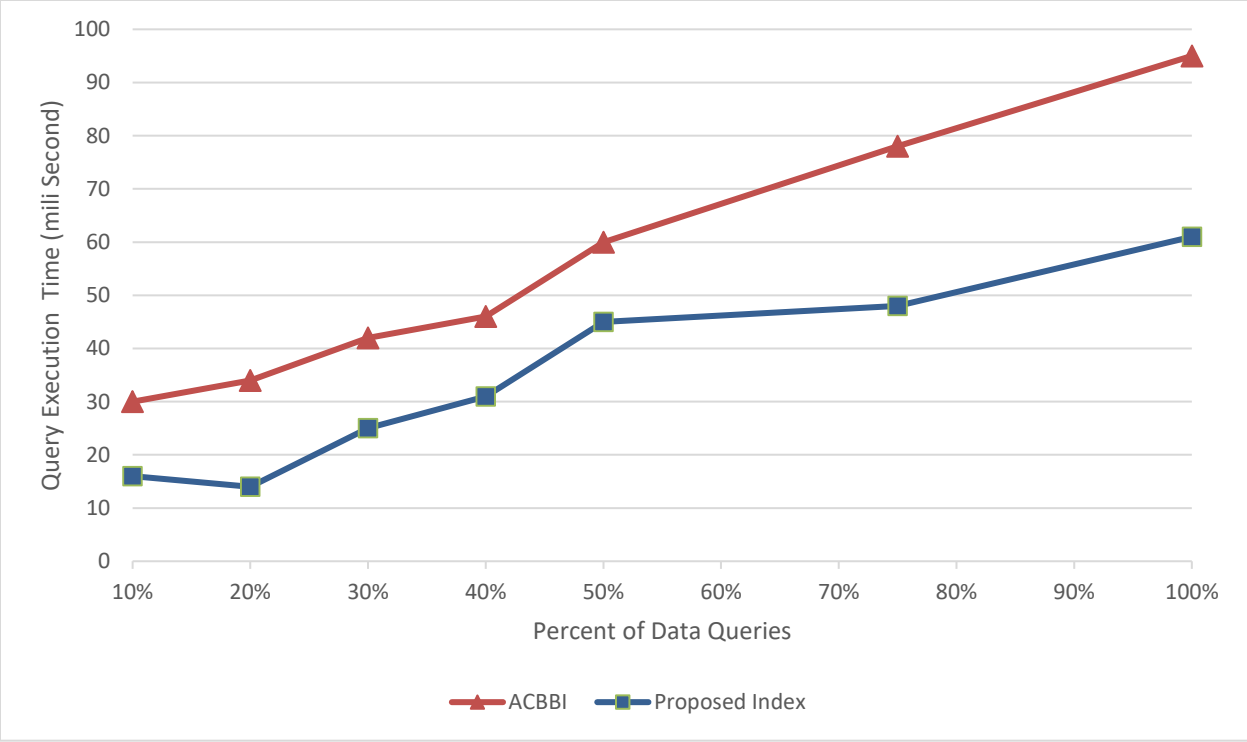

Fig. 23 Query Processing Time of Different Indexing Methods in the Uniform Data Distribution

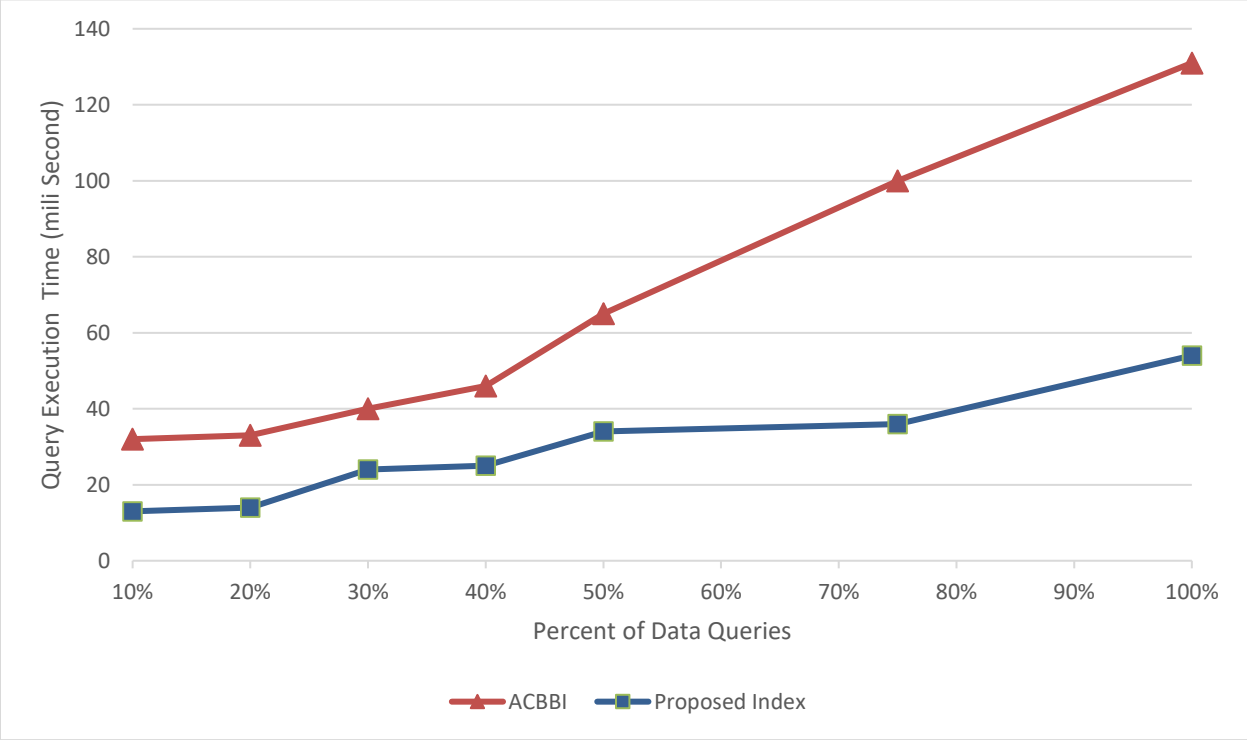

Fig. 24 Query Processing Time of Different Indexing Methods in the Skew Data Distribution

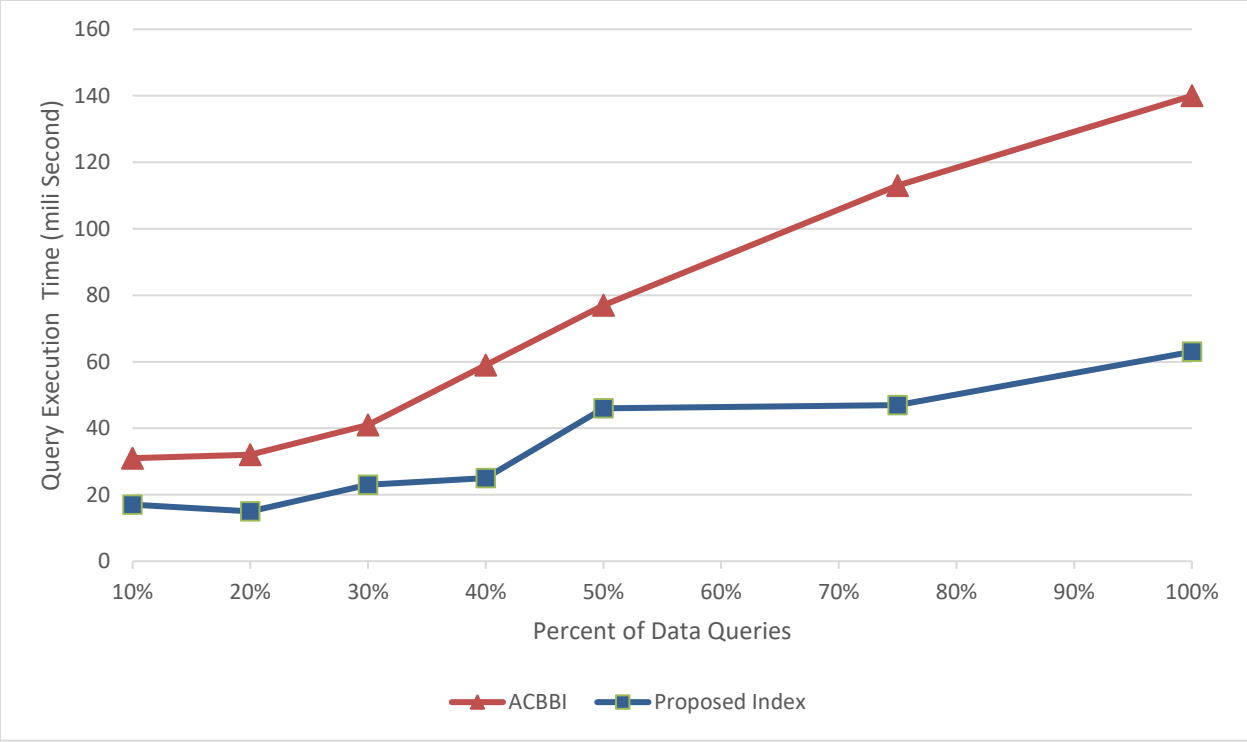

Fig. 25 Query Processing Time of Different Indexing Methods in the Hyper Skew Data Distribution

4.2.3.3 Comparison in terms of Storage Cost

To compare the storage cost of our proposed indexing method with the ACBBI indexing method, a set of experiments were performed. In all experiments, the number of records in the data stream was 2M records per minute, the number of partitions, the branching factor, and the number of executors were 4, 64, and 4, respectively. We only used 1 node with 4 processing cores in our experiments. Fig. 26 shows the index size of different indexing methods for three different data distributions. As shown in Fig. 26, it is clear that the index size in our proposed indexing method is more than that in the ACBBI indexing method in all data distributions since more than one B+Tree index is created in our proposed indexing method but in the ACBBI indexing method, only one B+Tree index tree is created.

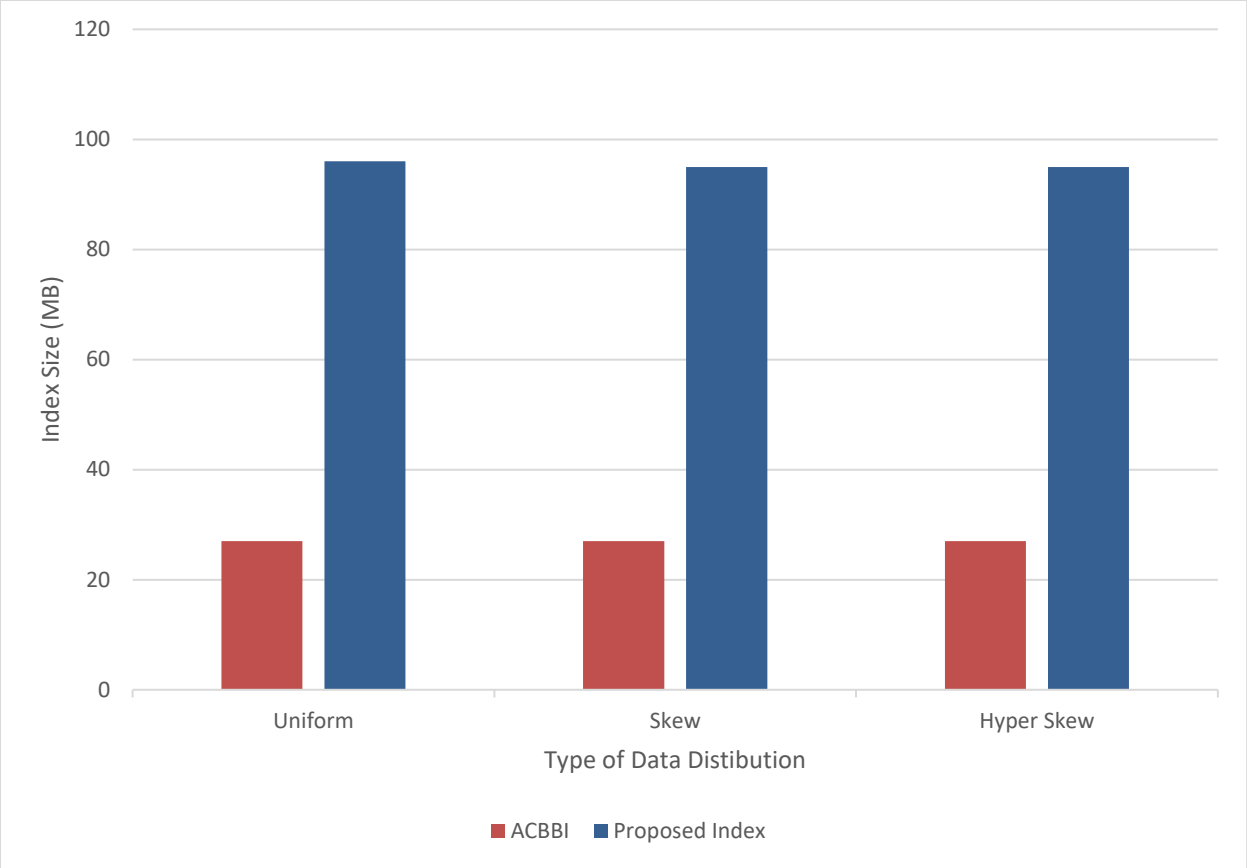

Fig. 26 Index Size of Different Indexing Methods

5. Conclusion and Future Works

In this paper, we proposed a system architecture including components for distributed indexing of streaming data and distributed processing of range queries over streaming data. We also designed a distributed B+Tree indexing method to process range queries over streaming data. The proposed indexing method used the map-reduce programming model in the Apache Spark platform to create a set of small B+Tree indexes over streaming data. In addition, we developed a range search algorithm based on the map-reduce programming model in the Apache Spark platform to process range queries in a distributed fashion. By performing a set of experiments, we showed that our proposed indexing method has acceptable scalability and with increasing data volume, it has a suitable efficiency for processing range queries over streaming data. It was also more efficient than the ACBBI indexing method to process range queries in terms of index creation time and query processing time. Unlike the ACBBI indexing method, it did not require a pruning operation in order to prevent the growth of the index tree size and therefore, it was more efficient than the ACBBI indexing method. In addition, it was developed to be used with the Apache Spark platform and therefore, it was reliable compared to the ACBBI indexing method which has the problem of single pint-of-failure. However, our proposed indexing method needed more storage space to store the index information compared to the ACBBI indexing method due to the creation of a set of small B+Tree indexes in parallel. Although the proposed indexing method required more storage space compared to the ACBBI indexing method, its scalability and high efficiency in processing of range queries could cover this weakness.

In the future, we plan to extend this paper in several directions as follows:

1) The use of some compression techniques to reduce the storage space required to store the index information in our proposed indexing method.
2) To extend our proposed indexing method to support logical operators in processing range queries over streaming data by proposing efficient query processing algorithms.

Declarations

Ethical Approval

This paper does not contain any studies with human participants or animals performed by any of the authors.

Competing Interests

The authors have no relevant financial or non-financial interests to disclose. The authors declare no conflict of interest.

Authors' Contributions

All authors contributed to the study conception and design. The first draft of the manuscript was written by SS and MM, then it is reviewed by AMR and AS. All authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding

The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Availability of data and materials

The source codes and datasets used in the paper are available from the first author on reasonable request.

References

[1] A. Margara and T. Rabl, "Definition of Data Streams," *Encycl. Big Data Technol.*, pp. 648–652, 2019.

[2] A. Bifet and J. Gama, "IoT data stream analytics," *Ann. des Telecommun. Telecommun.*, vol. 75, no. 9–10, pp. 491–492, Oct. 2020.

[3] S. Tiwari and S. Agarwal, "Data Stream Management for CPS-based Healthcare: A Contemporary Review," *IETE Tech. Rev. (Institution Electron. Telecommun. Eng. India)*, pp. 1–24, Jul. 2021.

[4] F. Mohamed, R. M. Ismail, N. L. Badr, and M. F. Tolba, "Data streams processing techniques," *Intell. Syst. Ref. Libr.*, vol. 115, pp. 279–305, 2017.

[5] Y. N. Law, H. Wang, and C. Zaniolo, "Relational languages and data models for continuous queries on sequences and data streams," *ACM Trans. Database Syst.*, vol. 36, no. 2, 2011.

[6] E. Panigati, F. A. Schreiber, and C. Zaniolo, "Data Streams and Data Stream Management Systems and Languages," in *Data Management in Pervasive Systems, Data-Centric Systems and Applications*, 2015, pp. 93–111.

[7] L. Yue-Jie, "Data stream of wireless sensor networks based on deep learning," *Int. J. Online Eng.*, vol. 12, no. 11, pp. 22–27, 2016.

[8] S. Chakravarthy and Q. Jiang, "Dsms Challenges," 2009, pp. 23–31.

[9] A. Behrend, D. Gawlick, and D. Nicklas, "DBMS meets DSMS: Towards a federated solution," *DATA 2012 - Proc. Int. Conf. Data Technol. Appl.*, no. February 2017, pp. 157–162, 2012.

[10] P. L. Lehman and S. B. Yao, "Efficient Locking for Concurrent Operations on B-trees," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, 1981.

[11] A. Gani, A. Siddiqa, S. Shamshirband, and F. Hanum, "A survey on indexing techniques for big data: taxonomy and performance evaluation," *Knowl. Inf. Syst.*, vol. 46, no. 2, pp. 241–284, 2016.

[12] M. Kholghi and M. Keyvanpour, "Comparative Evaluation of Data Stream Indexing Models," *Int. J. Mach. Learn. Comput.*, vol. 2, no. 3, pp. 257–260, 2012.

[13] N. Shivakumar and H. García-Molina, "Wave-Indices: Indexing Evolving Databases," in *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 1997, vol. 26, no. 2, pp. 381–392.

[14] T. Y. C. Leung and R. R. Muntz, "Generalized data stream indexing and temporal query processing," in *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, 1992, pp. 124–131.

[15] F. B. Adamu, A. Habbal, S. Hassan, R. Les Cottrell, B. White, and I. Abdullahi, "A Survey On Big Data Indexing Strategies," in *NETAPPS2015*, 2015.

[16] S. Badiozamany and T. Risch, "Scalable ordered indexing of streaming data," *Int. Work. Accel. Data Manag. Syst. Using Mod. Process. Storage Archit.*, 2012.

[17] Z. Deng *et al.*, "An Efficient Indexing Approach for Continuous Spatial Approximate Keyword Queries over Geo-Textual Streaming Data," *ISPRS Int. J. Geo-Information*, vol. 8, no. 2, p. 57, Jan. 2019.

[18] Z. Deng *et al.*, "Parallel processing of dynamic continuous queries over streaming data flows," in *IEEE Transactions on Parallel and Distributed Systems*, 2015, vol. 26, no. 3, pp. 834–846.

[19] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed B-tree," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 598–609, 2008.

[20] M. R. Sumalatha and M. Ananthi, "Efficient data retrieval using adaptive clustered indexing for continuous queries over streaming data," *Cluster Comput.*, pp. 1–15, 2017.

[21] M. Ananthi, D. K. Sreedhevi, and M. R. Sumalatha, "Dynamic continuous query processing over streaming Data," in *2016 International Conference on Computation of Power, Energy, Information and Communication, ICCPEIC 2016*, 2016, pp. 183–187.

[22] D. Kalashnikov, S. Prabhakar, S. Hambrusch, and W. Aref, "Efficient evaluation of continuous range queries on moving objects," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 2453, pp. 731–740, 2002.

[23] H. Wang and A. Belhassena, "Parallel trajectory search based on distributed index," *Inf. Sci. (Ny).*, vol. 388–389, pp. 62–83, 2017.

[24] J. Rao and K. A. Ross, "Making B + -Trees cache conscious in main memory," in *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 2000, vol. 29, no. 2, pp. 475–486.

[25] R. Li, H. Hu, H. Li, Y. Wu, and J. Yang, "MapReduce Parallel Programming Model: A State-of-the-Art Survey," *Int. J. Parallel Program.*, vol. 44, no. 4, pp. 832–866, 2016.

[26] Ishwarappa and J. Anuradha, "A brief introduction on big data 5Vs characteristics and hadoop technology," *Procedia Comput. Sci.*, vol. 48, no. C, pp. 319–324, 2015.

[27] S. R. M. Zeebaree, H. Shukur, L. Haji, and R. Zebari, "Characteristics and Analysis of Hadoop Distributed Systems," *Technol. Reports Kansai Univ.*, vol. 62, no. 4, pp. 1555–1564, 2020.

[28] "Apache Spark." [Online]. Available: http://spark.apache.org/.

[29] A. Bansal, R. Jain, and K. Modi, *Big Data Streaming with Spark*. Springer Singapore, 2019.

[30] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on Apache Spark," *Int. J. Data Sci. Anal.*, vol. 1, no. 3–4, pp. 145–164, Nov. 2016.

[31] A. V. Hazarika, G. Jagadeesh Sai Raghu Ram, and E. Jain, "Performance comparision of Hadoop and spark engine," in *Proceedings of the International Conference on IoT in Social, Mobile, Analytics and Cloud, I-SMAC 2017*, 2017, pp. 671–674.

[32] Y. Samadi, M. Zbakh, and C. Tadonki, "Comparative study between Hadoop and Spark based on Hibench

benchmarks," in *Proceedings of 2016 International Conference on Cloud Computing Technologies and Applications, CloudTech 2016*, 2017, pp. 267–275.

[33] X. Zhao, S. Garg, C. Queiroz, and R. Buyya, *A Taxonomy and Survey of Stream Processing Systems*, 1st ed. Elsevier Inc., 2017.

[34] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 331–342, Jun. 2000.

[35] J. Park, B. Hong, and C. Ban, "A query index for continuous queries on RFID streaming data," *Sci. China, Ser. F Inf. Sci.*, vol. 51, no. 12, pp. 2047–2061, 2008.

[36] K. L. Wu, S. K. Chen, and P. S. Yu, "Processing continual range queries over moving objects using VCR-based query indexes," in *Proceedings of MOBIQUITOUS 2004 - 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2004, pp. 226–235.

[37] R. A. Hankins and J. M. Patel, "Effect of node size on the performance of cache-conscious B +-trees," in *Performance Evaluation Review*, 2003, vol. 31, no. 1, pp. 283–295.

[38] S. Heinz, J. Zobel, and H. E. Williams, "Burst tries: A fast, efficient data structure for string keys," in *ACM Transactions on Information Systems*, 2002, vol. 20, no. 2, pp. 192–223.

[39] A. Silverstein and D. Baskins, "Judy IV Shop Manual," 2002.

[40] D. Baskins, "Judy home page," 2003. [Online]. Available: http://judy.sourceforge.net.

[41] X. Yu, K. Q. Pu, and N. Koudas, "Monitoring k-nearest neighbor queries over moving objects," in *Proceedings - International Conference on Data Engineering*, 2005, pp. 631–642.

[42] H. Singh and S. Bawa, "A MapReduce-based scalable discovery and indexing of structured big data," *Futur. Gener. Comput. Syst.*, vol. 73, pp. 32–43, 2017.

[43] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," in *SOSP'07 - Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007, pp. 159–174.

[44] A. A. Safaei, "Real-time processing of streaming big data," *Real-Time Syst.*, vol. 53, no. 1, pp. 1–44, 2017.

[45] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts - 7th. ed.*, 7th ed., vol. 4. McGraw-Hill, 2019.

[46] K. Pollari-malmi, "B + -trees." [Online]. Available: https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf.

[47] C. S. By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms, fourth edition*, 4th ed. The MIT Press, 2022.

[48] grouplens, "MovieLens Dataset." [Online]. Available: https://grouplens.org/datasets/movielens/.

[49] D. Taniar, C. H. C. Leung, W. Rahayu, and S. Goel, *High-Performance Parallel Database Processing and Grid Databases*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2008.