TinyLFU-based Semi-Stream Cache Join for Near-Real-Time Data Warehousing

M. Asif Naeem[1*], Wasiullah Waqar[2], Farhaan Mirza[3] and Ali Tahir[1]

[1,2*]School of Computing, National University of Computer and Emerging Sciences (NUCES), H-11/4, Islamabad, 40000, Pakistan.
[3]SECMS, Auckland University of Technology, 6 Paul Street, Auckland, 1010,New Zealand.
[4]Institute of Geographical Information Systems, National University of Science and Technology, Islamabad, 40000, Pakistan.

*Corresponding author(s). E-mail(s): asif.naeem@nu.edu.pk;

Abstract

Semi-stream join is an emerging research problem in the domain of near-real-time data warehousing. A semi-stream join is basically a join between a fast stream ($\boldsymbol{S}$) and a slow disk-based relation ($\boldsymbol{R}$). In the modern era of technology, huge amounts of data are being generated swiftly on a daily basis which needs to be instantly analyzed for making successful business decisions. Keeping this in mind, a famous algorithm called CACHEJOIN (Cache Join) was proposed. The limitation of the CACHEJOIN algorithm is that it does not deal with the frequently changing trends in a stream data efficiently. To overcome this limitation, in this paper we propose a TinyLFU-CACHEJOIN algorithm, a modified version of the original CACHEJOIN algorithm, which is designed to enhance the performance of a CACHEJOIN algorithm. TinyLFU-CACHEJOIN employs an intelligent strategy which keeps only those records of $\boldsymbol{R}$ in the cache that have a high hit rate in $\boldsymbol{S}$. This mechanism of TinyLFU-CACHEJOIN allows it to deal with the sudden and abrupt trend changes in $\boldsymbol{S}$. We developed a cost model for our TinyLFU-CACHEJOIN algorithm and proved it empirically. We also assessed the performance of our proposed TinyLFU-CACHEJOIN algorithm with the existing CACHEJOIN algorithm on a skewed synthetic dataset. The experiments proved that TinyLFU-CACHEJOIN algorithm significantly outperforms the CACHEJOIN algorithm.

Keywords: CACHEJOIN, TinyLFU-CACHEJOIN, Semi-stream joins, Cache optimization, Data Warehousing

1 Introduction

In today's fast-paced era of information technology, massive amounts of data are being generated exponentially on a daily basis [1]. Continuous streams of big data provide various challenges [2] for extracting useful information from these streams to make successful business decisions. Continuous real-time data is needed for making crucial business decisions based on swift evolving trends in the data. Data Warehousing (DWH) is a fundamental part of any decision support system nowadays [3]. In DWH, we have 3 main stages namely Extraction, Transformation and Load (ETL) [4]. Fig 1 shows the ETL process that is necessary for DWH.

Traditional data warehouses are updated in a batch manner during offline periods [5, 6] (i.e. weekly or daily basis) which makes them less feasible for real-time business applications where quick decisions need to be made to get maximum market gain. Hence, near-real-time data warehousing [7] has become an emerging area of research which plays an influential part in processing continuous real-time data to support strategic decision making [8] and business strategies [9].

In the near-real-time data warehousing concept, a stream of transactional data (denoted by S) produced by various data sources must be reflected into the data warehouse with minimal delay. However, S is incomplete and not in the format required by the data warehouse. Therefore, S has to be formatted and enriched with additional information stored in disk-based master data (denoted by R). For this purpose, an efficient join algorithm for joining S with R is required. Usually these types of joins are called semi-stream joins and are performed in the transformation phase of the ETL process. Fig 2 presents a graphical illustration of this semi-stream join operation.

A key challenge in semi-stream join operations is how to join the bursty and high volume S with the slow disk-based R with limited available resources. In other words, the objective is to perform this join operation with a minimal bottleneck in S. To address this challenge, a number of semi-stream join algorithms like MESHJOIN [10], HybridJoin [11, 12] were proposed in the literature. A limitation of the MESHJOIN algorithm is that it does not perform well on skewed data and this is a common characteristic in S as some products in a store get sold more frequently than others.Another improved version called CACHEJOIN (Cache Join) [13] was proposed to address the limitations

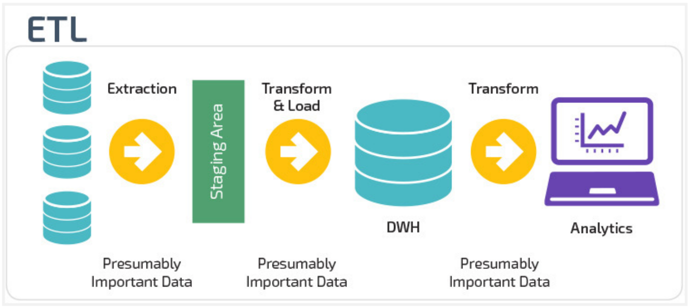

ETL
Extraction
Staging Area
Transform & Load
Transform
DWH
Analytics
Presumably Important Data
Presumably Important Data
Presumably Important Data

Fig. 1: An overview of ETL (Extraction-Transformation-Loading) process

of MESHJOIN algorithm by caching the most frequent records of R. As a result, it performs efficiently for non-uniform and skewed data.

The main objective of implementing cache in semi-stream joins is to improve the service rate. Service rate basically means how many stream tuples an algorithm processes in a unit second. CACHEJOIN moves R tuples into cache which are frequent in S. The criteria of deciding whether a tuple is frequent is to compare its number of occurrences in a join window at a certain time t with a pre-defined frequency threshold. For example if the value of the pre-defined frequency threshold is five then any stream tuple with its number of occurrences greater or equal to five will be considered as a frequent tuple. These frequent tuples can be further categorised into low or high frequent tuples. In the above example, a tuple with frequency five or six can be categorised as a low frequent tuple while a tuple with frequency ten can be categorised as a high frequent tuple. Which means a tuple with frequency five has low hit rate in S while a tuple with frequency ten comparatively has high hit rate in S. The existing CACHEJOIN does not differentiate these low and high frequent tuples and during its cache eviction process it evicts tuples from the cache on random basis. The eviction process is necessary when the size of cache is limited, and no more space is available to accommodate the incoming frequent tuples. Since CACHEJOIN evicts high frequent tuples from the cache with equal probability to the low frequent tuples, this limits the number of hits for the incoming tuples in S and eventually reduces the service rate of the algorithm.

In this paper, we address this limitation of CACHEJOIN by presenting an intelligent version of CACHEJOIN called TinyLFU-CACHEJOIN. TinyLFU-CACHEJOIN stores only those frequent tuples of R in the cache which have a high hit rate in S. Moreover, the algorithm tracks the frequency (or hit rate) of all the tuples in cache and when the eviction process is required it evicts a tuple from the cache which has minimum frequency. This also enable the new algorithm to adapt changing trends in S. In the experiments due to implementing the new feature, a significant gain is observed in the service rate

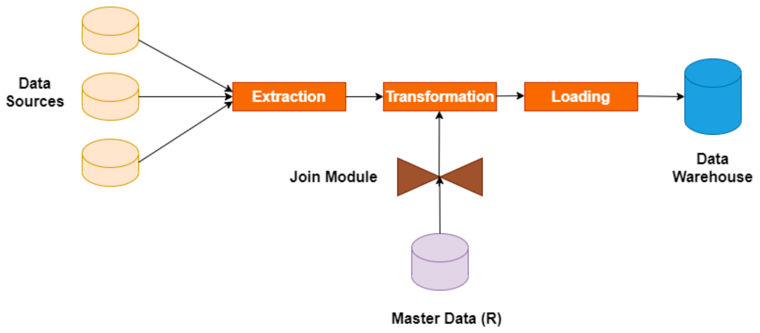


Fig. 2: Semi-stream join during the enrichment phase

of TinyLFU-CACHEJOIN. In summary, the main contributions presented in this paper are

- Developing an intelligent algorithm called TinyLFU-CACHEJOIN that implements intelligent caching technique.
- Enabling TinyLFU-CACHEJOIN algorithm to cope with frequently changing trends in S.
- Tuning the TinyLFU-CACHEJOIN algorithm for optimal performance on skewed stream data.
- Developing a cost model for TinyLFU-CACHEJOIN.
- Evaluating its service rate with existing CACHEJOIN.

The rest of the paper is organised as follows. Section 2 presents existing literature in this area. Section 3 presents the problem in existing CACHEJOIN. The new TinyLFU-CACHEJOIN with its architecture, algorithm, and cost model is presented in Section 4. Section 5 presents the service rate evaluation of TinyLFU-CACHEJOIN and finally Section 6 concludes the paper.

2 Literature Review

This section presents an overview of existing work in the domain of semi-stream join algorithms and cache performance optimization. The literature review is divided into two parts: (a) semi-stream join algorithms like index based, non-index based, and cache based algorithms; (b) cache replacement algorithms.

2.1 Semi-Stream Join Algorithms

MESHJOIN algorithm [10] was the first algorithm which was proposed to join a high-speed continuous stream S with a huge disk-based master data R using restricted memory.The algorithm employs two buffers namely disk-buffer and stream-buffer to manage the two various sources of data, disk and stream. The algorithm is basically a hash join that uses hash table functionality to process stream items more readily. The algorithm scans R sequentially and the service rate of the algorithm affects inversely with the size of R. The algorithm has no assurance that each expensive scan of R process at least a single tuple of S. Therefore, the algorithm has sub-optimal performance for bursty and non-uniform S .

In [14], the authors proposed a new join architecture called Simois which jumbles the possibly highest ranked keys with the most load and hashes the others. In order to pick out those keys which have most of the workload in two streams, the authors presented a novel and efficient counting scheme. The authors validated their claim by testing Simois through rigorous experimentation performed on real world datasets and demonstrated that Simois performed better as compared to existing modern designs.

Another robust join algorithm called HYBRIDJOIN [11] was proposed to join semi-stream data by merging MESHJOIN [10] and INLJ [15]. Unlike MESHJOIN, HYBRIDJOIN employs an index to read R. The advantage of HYBRIDJOIN over MESHJOIN is that HYBRIDJOIN algorithm processes at least one tuple of S against each read of R. HYBRIDJOIN can handle a bursty workload well, but it doesn't account for the skewed nature of the input stream.

In [16], the authors proposed two new optimized algorithms namely Parallel-Hybrid Join (P-HYBRIDJOIN) and Hybrid Join with Queue and Stack (QaS-HYBRIDJOIN). Both of these algorithms were extensions of the existing HYBRIDJOIN algorithm. The main goal of both these algorithms was to reduce processing cost in terms of disk I/O. The experiments presented in their paper reported that both algorithms performed better as compared to existing HYBRIDJOIN.

Semi-Stream Balanced Join (SSBJ) [17] was another join algorithm which introduced cache inequality to deal with many-to-many equijoins in a memory efficient manner. SSBJ was built upon the concept of MESHJOIN algorithm by adding a new module called cache to it. The authors implemented a new theoretical equation for cache known as cache inequality. The cache inequality assured that with limited memory, the algorithm achieved a high service rate. One important thing to observe here was that in SSBJ the cache size was variable based on each join value.

DSim-Join [18] was another algorithm which was proposed to address the problem of semi-stream join. DSim-Join aimed to minimize cache-based database accessed in a distributed stream processing system, equalized the load between parallel join threads, counterbalanced join processing, and decreased the data transmission. The authors showed through their experiments on large real world datasets that DSim-Join performed marginally better than existing techniques.

Another adaptive semi-stream join algorithm was the CACHEJOIN algorithm [13] which is considered as a state of the art work in this paper and compared the performance of our algorithm with it. CACHEJOIN is an efficient algorithm when dealing with non-uniform stream data compared to previous semi-stream join algorithms like MESHJOIN, HYBRIDJOIN. The

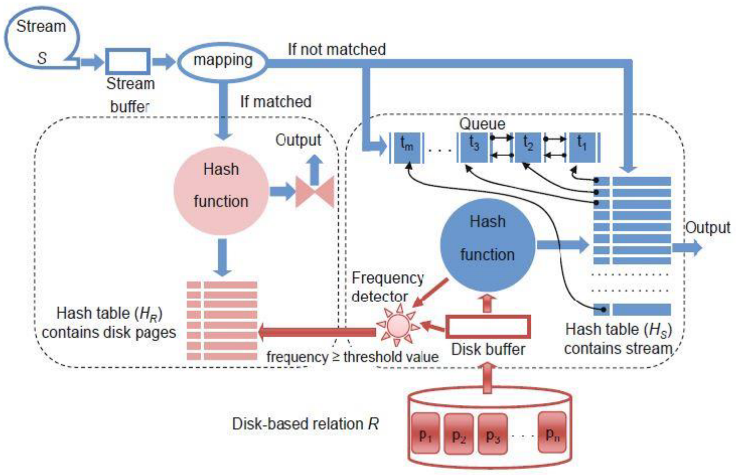


Fig. 3: CACHEJOIN execution architecture [13]

main components of CACHEJOIN are two hash tables, a stream-buffer ($\mathrm{S_b}$), a disk-buffer ($\mathrm{D_b}$) and a queue (Q). The external input sources of the algorithm include incoming S and R. One hash table that saves S records is represented by $\mathrm{H_S}$ whereas the other hash table saves most frequent records of R is denoted by $\mathrm{H_R}$. CACHEJOIN algorithm operates in two sequential phases i.e Stream-Probing (SP) phase and Disk-Probing (DP) phase. First SP phase is executed in which the stream records are matched against records in $\mathrm{H_R}$. Those stream records that are matched successfully are sent to output whereas unsuccessfully matched stream records fill $\mathrm{H_S}$ by storing their key values in queue. Then the DP phase initiates its execution by loading a partition of R into $\mathrm{D_b}$ by making use of the oldest record value of Q like indexing. One by one all the records are probed by the algorithm from $\mathrm{D_b}$ into $\mathrm{H_S}$. If there is a match then that stream record is produced as output while concurrently it is deleted from Q and $\mathrm{H_S}$. As there is a possibility of more than one match in $\mathrm{H_S}$ against one record of $\mathrm{D_b}$, a count of the frequency of matched data items is kept in the DP phase. A preset threshold value is used to compare with the frequency of matched records to conclude whether a record of $\mathrm{D_b}$ is frequent or not. A record is seen as frequent if the frequency is more than the predefined value hence it is shifted to $\mathrm{H_R}$. Once all the records from $\mathrm{D_b}$ have been probed into $\mathrm{H_S}$, the DP phase is finished. This sequential implementation of the two phases (i.e. SP and DP) concludes one full iteration of the CACHEJOIN algorithm. The detailed working of the CACHEJOIN algorithm is displayed in Fig 3.

2.2 Cache Replacement Algorithms

This section focuses on cache replacement algorithms which are employed to optimize the performance of cache by reducing cache miss rates [19]. The process of cache replacement is a primary focus of this paper.

The pioneer algorithm in this area was Least Recently Used (LRU) [20]. This algorithm removes the least recently used items from the cache to make room for new items. It also needs to keep history of all data items to keep a track of least recently used items.The main benefit of this policy is that it is straightforward to implement but its limitation is that it doesn't keep track of the frequency of data items.

The other algorithm presented in this area was First In First Out (FIFO) [20]. It replaces the oldest page in the cache which has not been in use for a long time and it is easy to implement. This algorithm uses a circular buffer to treat pages and discards pages in a round robin manner. Due to this round robin manner, this algorithm is not efficient and can cause early page faults.

A further extension to this was Least Frequently Used (LFU) [21]. It keeps track of the frequency of the data items being used and deletes those items from the cache first that are used less frequently. This algorithm keeps a counter in order to count the frequency of the data items and is also very easy to implement.

The latest algorithm which used in the approach proposed here is TinyLFU [22]. In this work, the authors proposed a new cache policy called TinyLFU that used a system where new items only enter the cache if they improve the hit rate of the cache. TinyLFU is a highly efficient frequency-based cache admission policy based on the concept of LFU policy and it boosts the effectiveness of caches that are subject to skewed or non-uniform data. TinyLFU is very compact and has a low memory footprint because it builds upon the theory of Bloom Filter [23]. TinyLFU keeps an estimated description of the access frequency of a vast sample of recently obtained items. TinyLFU uses approximate counting techniques such as Counting Bloom Filter (CBF) to support a freshness mechanism in order to remove old events and keep the history recent. TinyLFU ensures that the data is updated frequently by using a reset mechanism. A count value is added to approximation sketches whenever a new element is added. When the count value reaches a pre-specified sampling size (W) then all the values obtained by counting Bloom Filter are divided by two. TinyLFU also reduces the memory overhead by reducing the size of every counter which is present in the approximate sketch while also reducing the number of counters allotted by the sketch. Through their experiments, the authors proved that TinyLFU is a memory efficient algorithm which significantly improved the performance of the cache especially for data or distributions that are skewed and non-uniform.

3 Problem Statement

CACHEJOIN algorithm [13] was originally designed to deal with non-uniform data therefore, it does not perform well on stream data which contains frequently changing trends. Data nowadays is rapidly changing and those products which were popular yesterday may not be popular today so there is no use in storing those items in cache which are not popular anymore. A representative example is a video caching service. A video clip which is viral and in high demand on a certain day might not be as popular after some days. The problem with the CACHEJOIN algorithm is that the cache module of the CACHEJOIN algorithm is not optimized. As we stated in the Introduction section, it uses a simple eviction policy by randomly evicting items from the cache once the cache is full. As a result those items are also found in the cache of the CACHEJOIN algorithm which are not popular anymore. The aim is to overcome this limitation of CACHEJOIN algorithm by integrating TinyLFU in the cache module of the CACHEJOIN algorithm. TinyLFU uses an intelligent mechanism where it allows only high-frequency items into the cache that are highly frequency. TinyLFU does this by comparing the frequency of current items in the cache against the frequency of new incoming items. This mechanism of TinyLFU makes sure that the cache is optimized all the time and is able to deal with the frequently changing trends in stream data.

4 TinyLFU CACHEJOIN

As concluded in the Literature Review, the cache module namely $\mathrm{H_R}$ of the CACHEJOIN is suboptimal and this can be further improved. $\mathrm{H_R}$ is basically a hash table which is used to store the frequent disk tuples. To determine if the tuple is frequent, the CACHEJOIN algorithm uses a simple threshold based approach. Any tuple with a frequency greater than the threshold value is considered as a frequent tuple. However, the algorithm does not categorise these frequent tuples into high frequency or low frequency. Due to this, the algorithm evicts a tuple randomly whenever eviction is required. In that case the algorithm evicts the high frequent tuples with equal probability in order to incorporate new disk tuples. This negatively affects the stream hit rate and eventually reduces the service rate of the algorithm. However, this cache module can be made intelligent by tracking the frequency of each tuple in the cache and only evicting low frequent tuples when eviction is required.

Based on this motivation, the paper presents an intelligent version of CACHEJOIN called TinyLFU-CACHEJOIN. The new approach integrates TinyLFU [22] in the cache module of the CACHEJOIN algorithm [13].

4.1 Execution Architecture

Fig 4 illustrates the detailed architecture of TinyLFU-CACHEJOIN. Major components of TinyLFU-CACHEJOIN are a stream buffer ($\mathrm{S_B}$), a disk buffer ($\mathrm{B_d}$), two hash tables namely ($\mathrm{H_R}$) and ($\mathrm{H_S}$) and a queue Q. $\mathrm{H_R}$ stores most

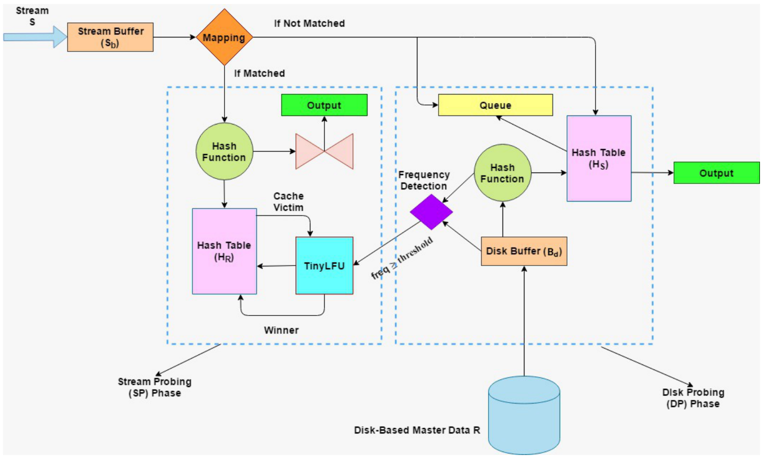


Fig. 4: Execution architecture for TinyLFU-CACHEJOIN

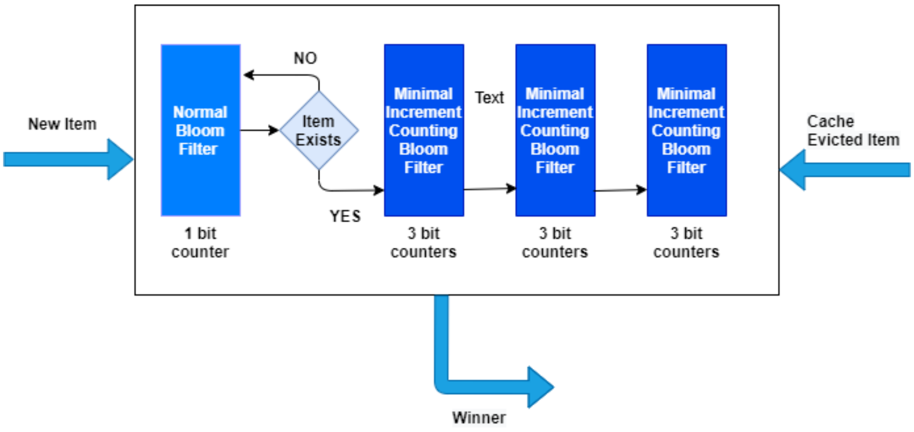


Fig. 5: TinyLFU architecture

frequent tuples of R whereas $\mathrm{H_S}$ stores S. The execution flow of TinyLFU-CACHEJOIN is similar to the existing CACHEJOIN.

Compared to existing methods such as CACHEJOIN, the TinyLFU-CACHEJOIN architecture introduces a new component called TinyLFU. TinyLFU is placed before $\mathrm{H_R}$ in the stream probing phase. The TinyLFU component maintains the cache with highly frequent tuples and evicts a frequent tuple if and only if its frequency is less than that of a new incoming tuple. Thus TinyLFU makes sure that the cache hit rate is high all the times and the cache is ready to cope with any trend changes in S.

Fig 5 illustrates the architecture of TinyLFU. As shown in the figure, TinyLFU takes two inputs, the new data item and the item evicted from the

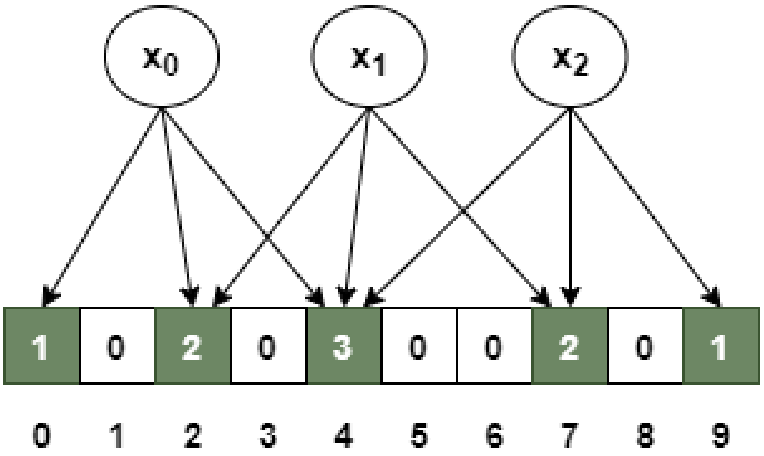


Fig. 6: Illustration of the Counting Bloom Filter (CBF)

cache also known as the cache victim. TinyLFU then decides whether accepting the latest item into the cache at the cost of the cache victim will escalate the cache hit rate or not. If the latest item increases the cache hit rate, then the new item is declared as the winner and moved it to the cache. However, if the new item does not increase the cache hit rate, then the cache victim is declared as the winner and sent it back to the cache.

TinyLFU implements the above rule by keeping a statistics of the frequency of items with a sizeable recent history. Saving these statistics is very important in terms of the experimental implementation of TinyLFU. There are two challenges for such a mechanism. The first challenge is to keep the history of recent items and eliminate old events. The second challenge is to reduce the memory overhead in order for TinyLFU to be considered a practical caching technique.

TinyLFU implements the Bloom Filter [23] data structure whose purpose is to check if an element exists in a set or not. There are many versions of Bloom Filter [24] but we will be focusing on one specific version namely the Counting Bloom Filter [25]. Fig 6 shows an example of a Counting Bloom Filter.

Since TinyLFU is built upon the Bloom Filter Theory [23], it uses minimal increment Counting Bloom Filter. A minimal increment Counting Bloom Filter is basically an Augmented Counting Bloom Filter which supports two methods i.e. Add and Estimate. It only increments the smallest counters by one. We now discuss how TinyLFU overcomes these two major challenges that were mentioned in the previous paragraph.

TinyLFU deals with the first challenge of keeping the recent history of events updated and removing old events by using a reset method. Every time an item is added to the frequency sketch, it augments a counter. When this counter size reaches the size of the sample (W), it divides this and all the other counters in the frequency sketch by two. The authors in their paper [22] have proved analytically and through experiments that errors resulting from this division are minimal. TinyLFU overcomes the second challenge of high memory overhead by reducing the size of every counter in the frequency sketch and also reducing all the counters allocated by the frequency sketch.

TinyLFU employs the doorkeeper mechanism shown in Fig 5 inside the box to reduce the size and number of counters in the frequency sketch. The doorkeeper is a a normal Bloom Filter which is put before the approximate counting scheme which in turns contains multiple minimal increment Counting Bloom Filters. If an item arrives, it first makes sure whether the item is present in the doorkeeper or not. If the item does no t exist in the doorkeeper which is mostly the case with tail items and first timers, it inserts that item into the doorkeeper otherwise it is inserted in the main structure of TinyLFU. In this way it avoids storing large counters for those items which are less frequent or have a single hit within the sample time.

4.2 Algorithm

This section presents the step-by-step execution of TinyLFU-CACHEJOIN as shown in Algorithm 1. Line 1 of the algorithm displays an infinite loop that is quite normal in these types of stream-based algorithms. Lines 2-9 depict the stream-probing phase of the algorithm. In the stream-probing phase, W tuples are read from the stream buffer and each stream tuple t from W is matched in $\mathrm{H_R}$. If t is matched from $\mathrm{H_R}$, the algorithm generates as an output else if t does not match in $\mathrm{H_R}$, then the algorithm places t in $\mathrm{H_S}$ and enqueues its corresponding value of the join attribute to Q. Lines 10-29 present the disk-probing phase of the algorithm. In this phase, those stream tuples which were not matched in the stream-probing phase are handled. The first step in the disk-probing phase is to load a chunk of disk tuples from R to disk buffer $\mathrm{B_d}$. The algorithm then reads $\mathrm{B_d}$ and for every tuple r in $\mathrm{B_d}$ finds its matching tuple in $\mathrm{H_S}$. If r matches in $\mathrm{H_S}$, the algorithm generates output for r. As there is one to many join, there can be more than one matches against r. The count of the total amount of matches are stored in f. Initially (or first time) we use f to determine whether the tuple r is frequent or not based on the predefined threshold value. Once the cache is full we use TinyLFU to optimize the cache. TinyLFU takes two inputs candidate(the incoming new stream tuple) and victim (the evicted disk tuple from $\mathrm{H_R}$). If the frequency of the candidate is greater than the frequency of victim, we insert candidate in $\mathrm{H_R}$ and vice versa. Lastly the algorithm deletes those tuples which matched against r from $\mathrm{H_S}$ along with their join attribute values from Q.

4.3 Cost Model

In this section, we develop a cost model for our TinyLFU-CACHEJOIN algorithm. Our cost model is closely related in nature to the cost model of CACHEJOIN algorithm [13] which includes both the memory and processing

Algorithm 1 TinyLFU-CACHEJOIN algorithm

```
Input: Disk based $R$ with indexed join attribute and Stream $S$
Output: $R \bowtie S$
Parameters: $W$ tuples of $S$ and $B_d$ tuples of $R$
while true
    READ $W$ tuples from the stream buffer
    for each tuple $t$ in $W$
        if $t \in H_R$
            OUTPUT $t$
        else
            LOAD stream $t$ in $H_S$ and its join attribute value to $Q$
    LOAD $B_d$ tuples of $R$ into the disk buffer
    for each tuple $r$ in $B_d$ from the disk buffer
        if $r \in H_S$
            OUTPUT $r$
            $f \leftarrow$ total number of tuples matched in $H_S$ against $r$
            can $\leftarrow$ candidate node(incoming node)
            vic $\leftarrow$ victim node(find the lowest that is not the candidate)
            canFreq $\leftarrow$ Frequency of candidate node
            vicFreq $\leftarrow$ Frequency of victim node
            if f $\geq$ ThresholdValue
                LOAD that tuple $r$ into $H_R$
                if canFreq $>$ vicFreq
                    INSERT can in $H_R$
                else
                    INSERT vic in $H_R$
            REMOVE $r$ from $H_S$ with its joins attribute value from $Q$
```

costs. The memory assigned to the stream buffer (S_B) is very minimal therefore it is not included in our memory cost calculations. A memory size of 0.05 MB for (S_B) was determined to be sufficient in all experiments.

Memory Cost: A large weight of total memory has been allocated to the two hash tables namely H_S and H_R. A relatively small amount of total memory is reserved for Q and the the disk buffer (D_B). The memory for each component can be calculated using the following equations below:

Memory(in bytes) for the disk buffer= $B_d \cdot V_R$
Memory(in bytes) for H_R= $h_R \cdot V_R$
Memory(in bytes) for TinyLFU= T_F
Memory(in bytes) for H_S= $\alpha[M - \{(B_d + h_R)V_R + T_F\}]$
Memory(in bytes) for the Queue= $(1 - \alpha)[M - \{(B_d + h_R)V_R + T_F\}]$

The total memory (M) for the TinyLFU-CACHEJOIN algorithm can be calculated using Eq. 1 which is shown below:

$$M = B_d \cdot V_R + h_R \cdot V_R + T_F + \alpha[M - \{(B_d + h_R)V_R + T_F\}] + (1 - \alpha)[M - \{(B_d + h_R)V_R + T_F\}] \qquad \text{(Eq. 1)}$$

Processing Cost: The processing cost, just like the memory cost, is first calculated separately for each component and then summed up all these costs to compute the total processing cost for one complete iteration:
Cost (in nanoseconds) to read disk tuples B_d tuples from R to disk buffer = $C_{I/O}(B_d)$
Cost (in nanoseconds) to match W_N tuples in H_R = $W_N \cdot C_H$
Cost (in nanoseconds) to match B_d tuples in H_S = $B_d \cdot C_H$
Cost (in nanoseconds) of TinyLFU component for B_d tuples = $B_d \cdot C_{TF}$
Cost (in nanoseconds) to construct the result for W_N tuples = $W_N \cdot C_O$
Cost (in nanoseconds) to construct the result for W_S tuples = $W_S \cdot C_O$
Cost (in nanoseconds) to read W_N tuples from the stream buffer = $W_N \cdot C_S$

Parameter Name	**Notation**
Number of stream records being processed in each iteration through $\mathrm{H_S}$	$\mathrm{W_S}$
Number of stream records being processed in each iteration through $\mathrm{H_R}$	$\mathrm{W_N}$
Disk buffer size (tuples)	$\mathrm{B_d}$
Disk tuple size (bytes)	$\mathrm{V_R}$
Size of $\mathrm{H_R}$ (tuples)	$\mathrm{h_R}$
Memory weight for $\mathrm{H_S}$	α
Memory weight for Q	$(1-\alpha)$
Cost to read $\mathrm{B_d}$ disk tuples from R to disk buffer (nano seconds)	$\mathrm{C_{I/O}(\,B_d)}$
Cost to match one tuple in $\mathrm{H_S}$ or $\mathrm{H_R}$ (nano seconds)	$\mathrm{C_H}$
Cost to construct output for one tuple (nano seconds)	$\mathrm{C_O}$
Cost to remove one tuple from Q and $\mathrm{H_S}$ (nano seconds)	$\mathrm{C_E}$
Cost to read one stream tuple from the stream buffer (nano seconds)	$\mathrm{C_S}$
Cost to append one tuple in Q and $\mathrm{H_S}$ (nano seconds)	$\mathrm{C_A}$
Cost for TinyLFU component for one disk tuple (nano seconds)	$\mathrm{C_{TF}}$
Total cost for one complete loop iteration (seconds)	$\mathrm{C_{loop}}$

Table 1: Notations used for Cost Model of TinyLFU-CACHEJOIN

Cost (in nanoseconds) to read $\mathrm{W_S}$ tuples from the stream buffer $= \mathrm{W_S} \cdot \mathrm{C_S}$
Cost (in nanoseconds) to append $\mathrm{W_S}$ tuples into $\mathrm{H_S}$ and $Q = \mathrm{W_S} \cdot \mathrm{C_A}$
Cost (in nanoseconds) to remove $\mathrm{W_S}$ tuples from $\mathrm{H_S}$ and $Q = \mathrm{W_S} \cdot \mathrm{C_E}$

The total processing cost for the algorithm for one whole loop can be computed by adding up all the individual costs as shown below in Eq. 2.

$$\mathrm{C_{loop}(secs)} = 10^{-9}[\mathrm{C_{I/O}(\,B_d)} + \mathrm{B_d}(\mathrm{C_H} + \mathrm{C_{TF}}) + \mathrm{W_S}(\,\mathrm{C_A} + \mathrm{C_E} + \mathrm{C_O} + \mathrm{C_S}) + \mathrm{W_N}(\,\mathrm{C_O} + \mathrm{C_H} + \mathrm{C_S})] \qquad \text{(Eq. 2)}$$

As the algorithm takes $\mathrm{C_{loop}(secs)}$ to process $\mathrm{W_S}$ and $\mathrm{W_N}$ stream tuples, the service rate μ can be computed using Eq. 3 which is shown below.

$$\mu = \frac{W_{\mathrm{N}} + W_{\mathrm{S}}}{C_{\mathrm{loop}}} \qquad \text{(Eq. 3)}$$

All the notations that were used in our memory cost and processing cost calculations for our cost model are displayed in Table 1.

5 Results and Discussion

5.1 Experimental Setup

This section explains the experimental setup that we used to conduct the experiments. This includes the software, hardware and data specifications for all our experiments.

5.1.1 Software and Hardware Specifications

We implemented both the algorithms namely the existing CACHEJOIN algorithm and our proposed algorithm TinyLFU-CACHEJOIN in the Java language using Eclipse which is an integrated development environment (IDE) for Java. We stored R on the disk through MySQL database(version 5.7.19). The processing costs of both the algorithms were calculated with regard to time measurements by using the built-in Java API called "nanoTime()". We used an external java library called "sizeofaj.jar" to calclulate the memory costs of both the algorithms. All experimental were executed on an Intel quad-core i5 processor along with 8GB RAM and a 500 GB SSD under the Windows 10 64-bit Operating System.

5.1.2 Data Specifications

The both algorithms have been evaluated using a synthetic dataset built on a Zipfian's distribution which is approximately reflect retails applications' sale data [26]. In order to produce stream dataset, a real time data generating script implementing Zipf's Law [27] was designed for the both algorithms. In the experiments, the size of R was varied from 5 million records to 20 million records. The size of each tuple in S and R was 20 bytes and 120 bytes respectively.

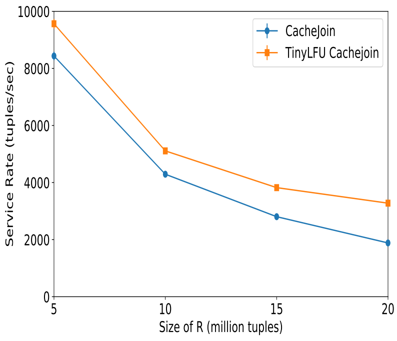
5.2 Results and Evaluation

To evaluate both the algorithms and their performance (service rate) three key parameters namely the size of R, total memory (M) available for the algorithm, and amount of skewness in S were analysed. Two experiments were conducted to tune $\mathrm{H_R}$ and $\mathrm{B_d}$ for their optimal sizes. For the performance experiments, a 95% confidence interval was calculated and plotted.

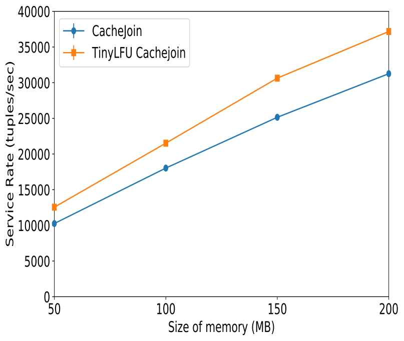
In addition to these experiments, another experiment was conducted to justify that TinyLFU-CACHEJOIN optimizes the Cache and stores more high frequency tuples in $\mathrm{H_R}$ as compared to the existing CACHEJOIN algorithm.

5.2.1 Performance evaluation by varying size of R

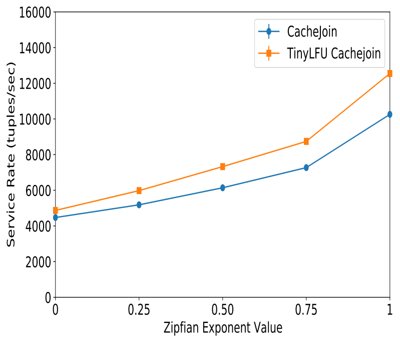
The service rates of the TinyLFU-CACHEJOIN and the CACHEJOIN algorithms were analyzed by changing the size of R from 5 million records to 20 million records in this experiment. The sizes for the other two parameters were



(a) Varying size of R



(b) Varying memory budget



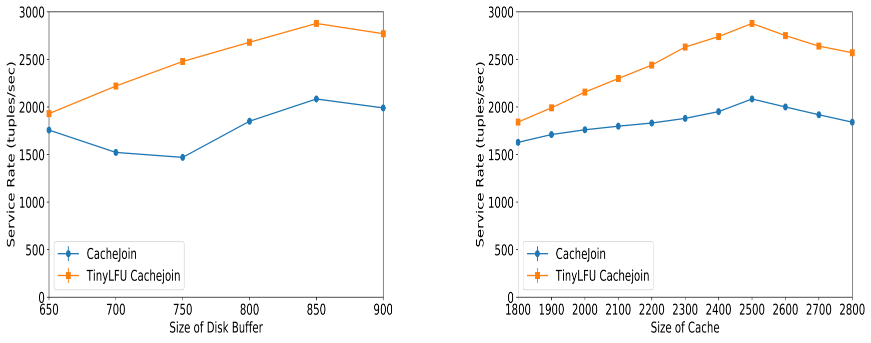
(c) Varying zipfian exponent

Fig. 7: Service rates comparison

fixed where the total available memory was equal to 50 MB and the skewness in the stream data (or the Zipf's exponent) was equal to 1. The results of this experiment are illustrated in Fig 7(a). Fig 7(a) clearly shows that the algorithm TinyLFU-CACHEJOIN significantly outperforms the CACHEJOIN algorithm across all four different settings of R which are 5 million, 10 million, 15 million and 20 million.

5.2.2 Performance evaluation for different memory settings

In this experiment, the service rates of both the algorithms were compared with respect to different memory settings ranging from 50 MB to 200 MB while keeping the other two parameters fixed i.e. the size of R was set to 5 million records and the amount of skewness in the stream data was set to 1. Fig 7(b) illustrates the results of this experiment. It can be seen from Fig 7(b) that our algorithm TinyLFU-CACHEJOIN again outperforms the CACHEJOIN algorithm on all the four various memory settings i.e 50 MB, 100 MB, 150



(a) Varying disk buffer Size

(b) Varying cache size

Fig. 8: Disk buffer and cache tuning

MB, 200 MB. We can see that TinyLFU-CACHEJOIN performs better than CACHEJOIN even when limited memory of 50 MB was assigned.

5.2.3 Performance evaluation for different skew levels in stream data

In this experiment both the algorithms were compared on the basis of their service rates by changing the amount of skewness in the stream data. The skew value is varied from 0 to 1 with 0 meaning that the stream data was hardly skewed (uniform) and 1 meaning that the stream data was highly skewed (non-uniform). While performing this experiment, the other two parameters were kept fixed i.e. size of R was set to 5 million records and the total memory for both the algorithms was set to 50 MB. Fig 7(c) shows the results of this experiment. We can see from the figure that our algorithm TinyLFU-CACHEJOIN again outperforms CACHEJOIN algorithm across all five different values of the Zipfian exponent which were 0, 0.25, 0.50, 0.75 and 1. Fig 7(c) also highlights the point that TinyLFU-CACHEJOIN algorithm starts performing better as soon as the stream data starts getting the moderated skew.

5.2.4 The disk buffer tuning

In this experiment the disk buffer was tuned to find its optimal size. The number of tuples in disk buffer were varied from 650 to 900 and the performance of the both algorithms was compared. The results of this experiment are shown in Fig 8(a). It is clear from the figure that in the both algorithms the optimal service rate is achieved for the setting of 850 tuples in the disk buffer. Therefore this is the reason that 850 tuples were set for the disk buffer in all performance experiments.

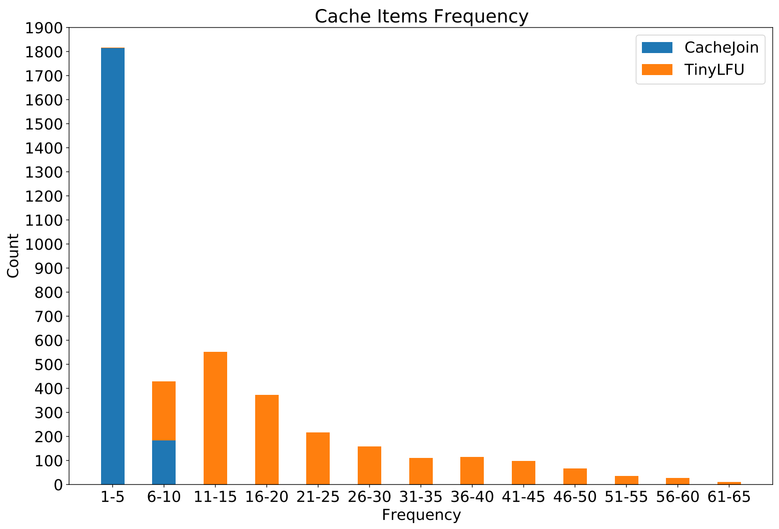


Fig. 9: Cache items frequency experiment

5.2.5 The cache tuning

In this experiment the cache size was varied to investigate how many tuples should be stored in the cache so that the optimal performance can be observed for the both CACHEJOIN and TinyLFU-CACHEJOIN algorithms. The number of tuples in the cache were varied from 1800 to 2800 and the performance of the both algorithms was compared. The results of this experiment are depicted in Fig 8(b). From the figure, the most optimal service rate is achieved in case of the both algorithms for the size of 2500 tuples in the cache. Therefore this is the reason that it was decided to store 2500 tuples in the cache for the all performance experiments.

5.2.6 Frequency comparisons for the tuple in the cache

This experiment compared both the algorithms in terms of frequency of items that they store in their cache. Through this experiment the aim was to improve that TinyLFU-CACHEJOIN stores higher frequency tuples in the cache as compared to existing CACHEJOIN algorithm. For all our previous experiments, 2500 items were stored in the cache but for this experiment 2000 items were stored in the cache for the both algorithms. For this experiment, the frequency threshold was set for the both algorithms to two which means that only those items will be admitted to the cache which have a frequency of two or more. Fig 9 shows the result for CACHEJOIN and TinyLFU-CACHEJOIN respectively. This shows that most of the tuples that are stored in cache under CACHEJOIN have low frequency as almost 1814 tuples out of 2000 are in the bin range of 1-5 frequency whereas TinyLFU-CACHEJOIN does not have any items in 1-5 frequency range. Fig 9 shows that TinyLFU-CACHEJOIN is mostly populated around higher frequencies and it is seen that there are a higher range of frequencies in TinyLFU-CACHEJOIN which is not the case in CACHEJOIN. This supports the argument that TinyLFU-CACHEJOIN performs better than CACHEJOIN because TinyLFU-CACHEJOIN optimizes the cache by storing high frequent tuples in it and that makes the service rate increases.

6 Conclusion and Future Work

The paper improves a well known existing algorithm called CACHEJOIN which is designed for dealing with skewed stream data. The constraint of CACHEJOIN algorithm is that there is no intelligent mechanism to store the most frequent data items in cache due to which the service rate of CACHEJOIN is suboptimal. To overcome this limitation, we proposed an intelligent version of the existing algorithm called TinyLFU-CACHEJOIN. TinyLFU-CACHEJOIN employs an intelligent system called 'TinyLFU' in the cache module of the existing CACHEJOIN algorithm. This optimizes the cache by keeping the highest frequent tuples in it. Through rigorous experimentation it has been demonstrated that TinyLFU-CACHEJOIN significantly outperforms the existing CACHEJOIN algorithm for all three input parameters i.e. size of R, total available memory, and level of skewness present in S. The key components of the algorithm such as the disk buffer and the cache were tuned. A rigorous experimentation has been carried out to show that the TinyLFU-CACHEJOIN algorithm stores higher frequent tuples in the cache as compared to the existing CACHEJOIN algorithm.

For future work, it is planned to implement TinyLFU-CACHEJOIN algorithm by running the two phases i.e. the stream-probing phase and the disk-probing phase in parallel on two separate computers. This parallel execution of the two phases will further accelerate the join process which will further improve the performance of the algorithm.

Declarations

- Funding: This is not a funded research.
- Conflict of interest: The authors have no conflict of interest with any editorial member of the journal.
- Ethics approval: The research presented in the paper has no human involvement and therefore no ethical approval is required.
- Consent for publication: the authors approves the consent for publishing their work in this journal.
- Authors' contributions
  M. Asif Naeem has a leading role in this research. He presented the idea and prepared the architecture for the proposed approach.
  Wasiullah Waqar as a master student implemented the algorithm and

produced the initial performance results.
Farhaan Mirza contributed in performance tuning and proofreading the paper.
Ali Tahir helped in write up of the paper.

References

[1] Lee, I. Big data: Dimensions, evolution, impacts, and challenges. *Business Horizons* **60** (2017) .

[2] Baig, M., Shuib, L. & Yadegaridehkordi, E. Big data adoption: State of the art and research challenges. *Information Processing Management* **56**, 102095 (2019) .

[3] Jain, S. & Sharma, S. *Application of data warehouse in decision support and business intelligence system*, 231–234 (2018).

[4] Vyas, S. & Vaishnav, P. A comparative study of various etl process and their testing techniques in data warehouse. *Journal of Statistics and Management Systems* **20** (4), 753–763 (2017) .

[5] Martínez, A. B., Galvis-Lista, E. A. & Florez, L. C. G. *Modeling techniques for extraction transformation and load processes: A critical review*, 41–47 (2012).

[6] Wijaya, R. & Pudjoatmodjo, B. *An overview and implementation of extraction-transformation-loading (etl) process in data warehouse (case study: Department of agriculture)*, 70–74 (2015).

[7] Sabtu, A. *et al. The challenges of extract, transform and loading (etl) system implementation for near real-time environment*, 1–5 (2017).

[8] Arora, R. & Gupta, M. E-governance using data warehousing and data mining. *International Journal of Computer Applications* **169**, 28–31 (2017) .

[9] Garani, G., Chernov, A., Savvas, I. & Butakova, M. *A data warehouse approach for business intelligence*, 70–75 (2019).

[10] Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A. & Frantzell, N. Meshing streaming updates with persistent data in an active data warehouse. *IEEE Transactions on Knowledge and Data Engineering* **20** (7), 976–991 (2008) .

[11] Dobbie, G., Naeem, M. A. & Weber, G. Hybridjoin for near-real-time data warehousing. *International Journal of Data Warehouse and Mining (IJDWM)* **7** (4), 21–42 (2011) .

[12] Naeem, M. A. Efficient processing of semi-stream data. *Eighth International Conference on Digital Information Management (ICDIM 2013)* 7–10 (2013) .

[13] Naeem, M. A., Dobbie, G. & Weber, G. *A lightweight stream-based join with limited resource consumption*, Vol. 7448, 431–442 (2012).

[14] Zhang, F., Chen, H. & Jin, H. Simois: A scalable distributed stream join system with skewed workloads. *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* 176–185 (2019) .

[15] Ramakrishnan, R., Gehrke, J. & Gehrke, J. *Database management systems* Vol. 3 (McGraw-Hill New York, 2003).

[16] Aziz, O., Anees, T. & Mehmood, E. An efficient data access approach with queue and stack in optimized hybrid join. *IEEE Access* **9**, 41261–41274 (2021) .

[17] Naeem, M. A., Weber, G. & Lutteroth, C. A memory optimal many-to-many semi-stream join. *Distributed and Parallel Databases* **37**, 623–649 (2019) .

[18] Kim, H. & Lee, K. Semi-stream similarity join processing in a distributed environment. *IEEE Access* **8**, 130194–130204 (2020) .

[19] Agrahari, K. & Singh, D. Realisation of cache optimisation using new technique. *International Journal of Advanced Research in Computer Science* **8**, 750–752 (2017) .

[20] Singh, N. & Agrahari, K. Enhanced performance of cache memory. *International Journal of Advanced Research in Computer Science* **9**, 34–36 (2018) .

[21] Kudagi, S. & Jayakumar, N. *Survey on different cache replacement algorithms*, Vol. 7, 10–13 (2019).

[22] Einziger, G., Friedman, R. & Manes, B. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage* **13**, 1–31 (2017) .

[23] Patgiri, R., Nayak, S. & Borgohain, S. Role of bloom filter in big data research: A survey. *International Journal of Advanced Computer Science and Applications* **9**, 655–661 (2018) .

[24] Gupta, D. & Batra, S. *A short survey on bloom filter and its variants*, 1086–1092 (2017).

[25] Kim, K., Jeong, Y., Lee, Y. & Lee, S. Analysis of counting bloom filters used for count thresholding. *Electronics* **8**, 779 (2019) .

[26] Sarna, G. & Bhatia, M. *Identification of suspicious patterns in social network using zipf's law*, 957–962 (2018).

[27] Ferrer-i Cancho, R. & Vitevitch, M. The origins of zipf's meaning-frequency law. *Journal of the Association for Information Science and Technology* **69**, 1369–1379 (2018) .