TinyLFU-based semi-stream cache join for near-real-time data warehousing

M. Asif Naeem · Wasiullah Waqar · Farhaan Mirza · Ali Tahir

Accepted: 29 July 2022 / Published online: 11 September 2022
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2022

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 (S) and a slow disk-based relation (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 R in the cache that have a high hit rate in S. This mechanism of TinyLFU-CACHEJOIN allows it to deal with the sudden and abrupt trend changes in 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 (Lee 2017). Continuous streams of big data provide various challenges (Baig et al. 2019) for extracting useful information from these streams to make successful business decisions. Continuous real-time data are 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 (Jain and Sharma 2018). In DWH, we have three main stages namely Extraction, Transformation and Load (ETL) (Vyas and Vaishnav 2017). Figure 1 shows the ETL process that is necessary for DWH.

Traditional data warehouses are updated in a batch manner during offline periods (Martínez, Galvis-Lista and Florez 2012; Wijaya and Pudjoatmodjo 2015) (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 (Sabtu et al. 2017) has become an emerging area of research which plays an influential part in processing continuous real-time data to support strategic decision making (Arora and Gupta 2017) and business strategies (Garani et al. 2019).

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. Figure 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 (Polyzotis et al. 2008), HybridJoin (Dobbie et al. 2011; Naeem 2013) 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) (Naeem et al. 2012) was proposed to address the limitations 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 categorized into low or high frequent tuples. In the above example, a tuple with frequency five or six can be categorized as a low frequent tuple, while a tuple with frequency ten can be categorized 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 enables 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 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 organized 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 Sect. 4. Section 5 presents the service rate evaluation of TinyLFU-CACHEJOIN and finally Sect. 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 (Polyzotis et al. 2008) 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 Zhang et al (2019), 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 (Dobbie et al. 2011) was proposed to join semi-stream data by merging MESHJOIN (Polyzotis et al. 2008) and INLJ (Ramakrishnan et al. 2003). 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 does not account for the skewed nature of the input stream.

In Aziz et al. (2021), 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) (Naeem et al. 2019) 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 (Kim and Lee 2020) 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. Recently, a research work has been published on semi-stream join; however, the focus was eval-
uating its performance in building a real-time data warehouse using MongoDB (Mehmood and Anees 2019). Similarly, another work has been published on evaluating joins performance in distributed environment (Kim and Lee 2020) which is not our primary focus in this paper.

Another adaptive semi-stream join algorithm was the CACHEJOIN algorithm (Naeem et al. 2012) 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 main components of CACHEJOIN are two hash tables, a stream-buffer (Sb), a disk-buffer (Db) 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 Hs, whereas the other hash table saves most frequent records of R is denoted by Hr. 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 Hs. Those stream records that are matched successfully are sent to output, whereas unsuccessfully matched stream records fill Hs by storing their key values in queue. Then, the DP phase initiates its execution by loading a partition of R into Db by making use of the oldest record value of Q like indexing. One by one, all the records are probed by the algorithm from Db into Hs. If there is a match, then that stream record is probed against records in Hr. Those stream records that are matched successfully are sent to output, while concurrently it is deleted from Q and Hs. As there is a possibility of more than one match in Hs against one record of Db, 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 Db is frequent or not. A record is seen as frequent if the frequency is more than the predefined value; hence, it is shifted to Hr. Once all the records from Db have been probed into Hs, 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 shown in Figure 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 (Agrahari and Singh 2017). The process of cache replacement is a primary focus of this paper.

The pioneer algorithm in this area was least recently used (LRU) (Singh and Agrahari 2018). 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 does not keep track of the frequency of data items.

The other algorithm presented in this area was First In First Out (FIFO) (Singh and Agrahari 2018). 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) (Kudagi and Jayakumar 2019). 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 (Einziger et al. 2017). 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 (Patgiri et al. 2018). 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 are 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 (Naeem et al. 2012) 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 are rapidly changing, and those products which were popular yesterday may not be pop-
ular 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 frequent. 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 HR of the CACHEJOIN is suboptimal and this can be further improved. HR 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 categorize 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 (Einziger et al. 2017) in the cache module of the CACHEJOIN algorithm (Naeem et al. 2012).

4.1 Execution architecture

Figure 4 illustrates the detailed architecture of TinyLFU-CACHEJOIN. Major components of TinyLFU-CACHEJOIN are a stream buffer (S_B), a disk buffer (B_D), two hash tables namely (H_R) and (H_S) and a queue Q. H_R stores most frequent tuples of R, whereas 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 HR 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.
Figure 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 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 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 (Einziger et al. 2017) 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 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 not 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 \( H_R \). If \( t \) is matched from \( H_R \), the algorithm generates as an output else if \( t \) does not match in \( H_R \), then the algorithm places \( t \) in \( H_S \) 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 \( B_d \). The algorithm then reads \( B_d \), and for every tuple, \( r \) in \( B_d \) finds its matching tuple in \( H_S \). If \( r \) matches in \( H_S \), 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 \( H_R \)). If the frequency of the candidate is greater than the frequency of victim, we insert candidate in \( H_R \) and vice versa. Lastly, the algorithm deletes those tuples which matched against \( r \) from \( H_S \) 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 (Naeem et al. 2012) which includes both the memory and processing 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 disk buffer \( (D_B) \). The memory for each component can be calculated using the following equations below:
Algorithm 1 TinyLFU-CACHEJOIN algorithm

Input: Disk-based R with indexed join attribute and Stream S
Output: R ⊆ S
Parameters: W tuples of S and BD tuples of R

1: while true do
2: READ W tuples from the stream buffer
3: for each tuple t in W do
4: if t ∈ HS then
5: OUTPUT t
6: else
7: LOAD stream t in HS and its join attribute value to Q
8: end if
9: end for
10: LOAD BD tuples of R into the disk buffer
11: for each tuple r in BD from the disk buffer do
12: if r ∈ HS then
13: OUTPUT r
14: fi ← total number of tuples matched in HS against r
15: can ← candidate node(incoming node)
16: vic ← victim node(find the lowest that is not the candidate)
17: canFreq ← Frequency of candidate node
18: vicFreq ← Frequency of victim node
19: if f ≥ ThresholdValue then
20: LOAD that tuple r into HR
21: if canFreq > vicFreq then
22: INSERT can in HR
23: else
24: INSERT vic in HR
25: end if
26: end if
27: REMOVE r from HS with its joins attribute value from Q
28: end if
29: end for
30: end while

Memory(in bytes) for the disk buffer = BD · VR
Memory(in bytes) for HR = hR · VR
Memory(in bytes) for TinyLFU = TF
Memory(in bytes) for HS = α[ M − {(BD + hR)VR + TF}]
Memory(in bytes) for the Queue = (1 − α)[ M − {(BD + hR)VR + TF}]

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

\[ M = BD \cdot VR + hR \cdot VR + TF + \alpha[M − {(BD + hR)VR + TF}] + (1 − \alpha)[M − {(BD + hR)VR + TF}] \]  

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 BD tuples from R to disk buffer = C_{I/O}(BD)
Cost (in nanoseconds) to match WN tuples in HR = WN · CH
Cost (in nanoseconds) to match BD tuples in HS = BD · CH
Cost (in nanoseconds) of TinyLFU component for BD tuples = BD · C_{TF}
Cost (in nanoseconds) to construct the result for WN tuples = WN · CO
Cost (in nanoseconds) to construct the result for WS tuples = WS · CS
Cost (in nanoseconds) to read WN tuples from the stream buffer = WN · CS
Cost (in nanoseconds) to read WS tuples from the stream buffer = WS · CE
Cost (in nanoseconds) to append WS tuples into HS and Q = WS · CA
Cost (in nanoseconds) to remove WS tuples from HS and Q = WS · 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 in Eq. 2.

\[ C_{loop}(secs) = 10^{-9}[C_{I/O}(BD) + BD \cdot (CH + C_{TF}) + WS \cdot (CA + CE + CO + CS) + WN \cdot (CO + CH + CS)] \]  

As the algorithm takes C_{loop}(secs) to process WS and WN stream tuples, the service rate \( \mu \) can be computed using Eq. 3 which is shown below.

\[ \mu = \frac{WN + WS}{C_{loop}} \]  

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

5 Experimentation

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...
Table 1: Notations used for cost model of tinyLFU-CACHEJOIN

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

The built-in Java API called “nanoTime().” We used an external java library called “sizeofaj.jar” to calculate 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 (Sarna and Bhatia 2018). In order to produce stream dataset, a real-time data-generating script implementing Zipf’s Law (Ferrer-i Cancho and Vitevitch 2018) 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 H_R and 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 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 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. 7a. Figure 7a 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. Figure 7b illustrates the results of this experiment. Figure 7b shows that our algorithm TinyLFU-CACHEJOIN again outperforms the CACHEJOIN algorithm on all the four various memory settings, i.e., 50 MB, 100 MB, 150 MB and 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 were hardly skewed (uniform) and 1 meaning that the stream data were 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. Figure 7c 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. Figure 7c 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 Cost validation

In this experiment, we validate our cost model in which we compared the calculated cost with empirical or measured cost. The results of the experiment are shown in Fig. 7d. From the figure, it can be observed that for each memory setting the calculated cost closely resembled the measured cost. This validated our correct implementation of the TinyLFU-CACHEJOIN.

5.2.5 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. 8a. 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.

### 5.2.6 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 shown in Fig. 8b. 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.

### 5.2.7 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 both the algorithms. For this experiment, the frequency threshold was set for both the algorithms to two which means that only those items will be admitted to the cache which have a frequency of two or more. Figure 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. Figure 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.

Author Contributions MAN has a leading role in this research. He presented the idea and prepared the architecture for the proposed approach. WW as a master student implemented the algorithm and produced the initial performance results. FM contributed in performance tuning and proofreading the paper. AT helped in write up of the paper.

Funding This is not a funded research.

Data availability Enquiries about data availability should be directed to the authors.

Declarations

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.

References

Agrahari K, Singh D (2017) Realisation of cache optimisation using new technique. Int J Adv Res Comput Sci 8:750–752
Arora R, Gupta M (2017) E-governance using data warehousing and data mining. Int J Comput Appl 169:28–31
Aziz O, Anees T, Mehmood E (2021) An efficient data access approach with queue and stack in optimized hybrid join. IEEE Access 9:41261–41274
Baig M, Shuib L, Yadegaridehkordi E (2019) Big data adoption: state of the art and research challenges. Inf Process Manag 56:102095 Dobbie G, Naeem MA, Weber G (2011) Hybridjoin for near-real-time data warehousing. Int J Data Wareh Min (IJDWM) 7(4):21–42
Einziger G, Friedman R, Manes B (2017) Tinylfu: a highly efficient cache admission policy. ACM Trans Storage 13:1–31
Ferrer-i Cancho R, Vitevitch M (2018) The origins of zipf’s meaning-frequency law. J Am Soc Inf Sci 69:1369–1379
Garani G, Chernov A, Savvas I, Butakova M (2019) A data warehouse approach for business intelligence. p 70–75
Gupta D, Batra S (2017) A short survey on bloom filter and its variants. p 1086–1092
Jain S, Sharma S (2018) Application of data warehouse in decision support and business intelligence system, pp 231–234
Kim H, Lee K (2020) Semi-stream similarity join processing in a distributed environment. IEEE Access 8:130194–130204
Kim H-J, Lee K-H (2020) Semi-stream similarity join processing in a distributed environment. IEEE Access 8:130194–130204
Kim K, Jeong Y, Lee Y, Lee S (2019) Analysis of counting bloom filters used for count thresholding. Electronics 8:779
Kudag J, Jayakumar N (2019) Survey on different cache replacement algorithms 7:10–13
Lee I (2017) Big data: dimensions, evolution, impacts, and challenges. Bus Horiz 60:293–303
Martinez AB, Galvis-Lista EA, Florez LCG (2012) Modeling techniques for extraction transformation and load processes: a critical review, pp 41–47
Mehmood E, Anees T (2019) Performance analysis of not only sql semi-stream join using mongodb for real-time data warehousing. IEEE Access 7:134215–134225
Naeem MA (2013) Efficient processing of semi-stream data. In: Eighth international conference on digital information management (ICDIM 2013), p 7–10
Naeem MA, Dobbie G, Weber G (2012) A lightweight stream-based join with limited resource consumption 7448:431–442
Naeem MA, Weber G, Lutteroth C (2019) A memory optimal many-to-many semi-stream join. Distrib Parallel Databases 37:623–649
Patgiri R, Nayak S, Borgohain S (2018) Role of bloom filter in big data research: a survey. Int J Adv Comput Sci Appl 9:655–661
Polyzotis N, Skiadopoulos S, Vassiliadis P, Simitsis A, Frantzell N (2008) Meshing streaming updates with persistent data in an active data warehouse. IEEE Trans Knowl Data Eng 20(7):976–991
Ramakrishnan R, Gehrke J, Gehrke J (2003) Database management systems. McGraw-Hill, New York
Sabtu A et al. (2017) The challenges of extract, transform and loading (etl) system implementation for near real-time environment. p 1–5
Sarna G, Bhatia M (2018) Identification of suspicious patterns in social network using zipf’s law. p 957–962
Singh N, Agrahari K (2018) Enhanced performance of cache memory. Int J Adv Res Comput Sci 9:34–36
Vyas S, Vaishnav P (2017) A comparative study of various etl process and their testing techniques in data warehouse. J Stat Manag Syst 20(4):753–763
Wijaya R, Pudjoatmodjo B (2015) An overview and implementation of extraction-transformation-loading (etl) process in data warehouse (case study: department of agriculture). p 70–74
Zhang F, Chen H, Jin H (2019) Simois: a scalable distributed stream join system with skewed workloads. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), p 176–185

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.