Optimising Queue-based Semi-Stream Joins by Introducing a Queue of Frequent Pages

[1]M. Asif Naeem, [2]Gerald Weber, and [2]Christof Lutteroth

[1]School of Engineering, Computer and Mathematical Sciences,
Auckland University of Technology,
Private Bag 92006, Auckland, New Zealand.
[2]Department of Computer Science,
The University of Auckland,
Auckland, New Zealand.
[1]`mnaeem@aut.ac.nz`
[2]`gerald@cs.auckland.ac.nz`, `christof@cs.auckland.ac.nz`

Abstract. Semi-stream joins perform a join between a stream and a disk-based table. These joins can easily deal with typical workloads in online real-time data warehousing in many scenarios and with relatively modest system requirements. The disk access is page-based. In the past, several proposals have been made to exploit skew in the distribution of the join attribute. Such skew is a common result of natural short- or long-tailed distributions in master data. Several semi-stream joins use caching strategies in order to improve performance. This works up to a point, but these algorithms still require relatively slow processing of stream data that matches with the infrequent tuples in the master data. In this work we explore the possibility of an additional strategy to exploit data skew: disk pages that are frequently accessed as a whole are accessed with priority. We show that considerable gain in service rate can be achieved with this strategy, while keeping memory consumption low. In essence we gain a three-stage approach to deal with skewed, unsorted data: caching plus our new strategy plus processing of the long tail of the distribution. Our experiments show that the new approach improves service rate significantly.

Keywords: Semi-stream join, Performance optimisation, Indexing

1 Introduction

Stream-based joins are important operations in modern system architectures, where just-in-time delivery of data is expected. We consider a particular class of stream-based joins, a semi-stream join that joins a single stream with a disk-based table. Such a join can be applied, for example, in real-time data warehousing [7, 6, 5]. In this work we only consider one-to-many equijoins, as they appear between foreign keys in the stream and a referenced primary key in the master data table. This is obviously a very important class of joins, and they are a natural case of a join between a stream of updates and master data in a data warehousing context [5], online auction systems [2] and supply-chain management [15]. If a join is one-to-many and if high throughput is the aim, it is important to exploit this and to choose a one-to-many join operator, since it allows to expire stream tuples as soon as they have matched a master data record. Consequently, we do not consider here many-to-many joins, e.g. joins on categorical attributes in master data, such as gender.

Most semi-stream join algorithms keep recent stream tuples in main memory in order to amortize the expensive disk access cost over a large section of the stream[13, 14, 10, 8, 9, 11]. As a consequence, many algorithms have a queue data structure that represents a section of the stream currently in processing. The queue keeps track of the stream tuples based on their loading timestamps. The queue typically stores join attribute values, and some of these algorithms [8, 9, 11] use these join attribute values in the queue (queue elements) as look up and load relevant disk-based master data through an index. In this paper we denote these queue elements as *lookup elements*.

Several semi-stream algorithms have contributed caching strategies in order to exploit skew in the distribution of join attributes in the stream. Skewed data is the norm in many application scenarios [1], such as data warehousing for purchases. A skewed distribution commonly found in applications is the 80/20 rule, for example it is a rule of thumb in many markets that 80% of sales are related to 20% products [1]. However, caching is only effective down to a certain frequency of the join attribute, even if one allows for a tuple-level cache [9, 11]. In our recent work [12] we have investigated whether one can further improve the disk-based part of the join operations. We have made some inroads by showing that by optimizing the choice of a lookup element from the queue, the service rate can be improved for a given skew. However, up to now it was not clear that this is an optimal strategy. This strategy was evaluated in the existing HYBRIDJOIN and CACHEJOIN algorithms [8, 9]; in this paper we denote these algorithms once they are improved by the lookup strategy (and therefore modified) by HYBRIDJOIN-L and CACHJOIN-L respectively. In this paper we present a new analysis that shows that this lookup strategy suffers still from two drawbacks: (a) this position varies with the degree of skew in stream data and (b) at the lookup position we do not find a good lookup element each time we look, so the algorihm still performs suboptimal. Further details about these two issues are presented in Section 3.2.

In this paper we address these issues by presenting a new approach that replaces the early lookup element: it is a priority queue PQ of frequent pages. The PQ contains page IDs of the master data pages with a high stream probability. We added this new strategy again to the basic HYBRIDJOIN and CACHEJOIN, resulting in two new algorithms HYBRIDJOIN-PQ and CACHEJOIN-PQ. In this paper we compare them with HYBRIDJOIN-L and CACHEJOIN-L and can demonstrate a further improvement. This improvement can be explained because the new algorithms now access the frequent pages in an optimal time interval; in contrast the strategy of HYBRIDJOIN-L and CACHEJOIN-L achieved a preferential treatment of the high probability pages, but with the side effect of loading other pages before time. Further details about this are presented in Section 4.

Our main findings in this research can be summarized as follows:

Higher service rate: By implementing the new strategy, both HYBRIDJOIN-PQ and CACHEJOIN-PQ outperform existing HYBRIDJOIN-L and CACHEJOIN-L, respectively, for skewed data.

Adaptability: By using the new strategy both HYBRIDJOIN-PQ and CACHEJOIN-PQ to adapt to changes in the stream data, e.g. the value of skew in stream data. By contrast, due to the fixed position for the optimal lookup element in the queue, HYBRIDJOIN-L and CACHEJOIN-L cannot optimally adapt themselves to skew variation in the stream data.

Three-stage approach to skewed data: The new algorithm CACHEJOIN-PQ can be said to employ three different stages to deal with a skewed distribution. For very frequent tuples, an in-memory cache is employed, for tuples that are not in themselved frequent but cluster on pages that then get frequent as a page (e.g. due to locality effects), we employ the in-memory priority queue of frequent pages. Finally, for all other tuples we use direct lookup.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 describes background and based on that formulate the problem statement. Section 4 presents the solution. Section 5 describes our experiments and finally Section 6 concludes the paper.

2 Related work

In this section, we present an overview of the previous work that has been done in the area of semi-stream joins, focusing on those that are closely related to our problem domain.

A seminal algorithm MESHJOIN [13, 14] has been designed especially for joining a continuous stream with disk-based master data, like in the scenario of active data warehouses. The MESHJOIN algorithm is a hash join, where the stream serves as the build input and the disk-based relation serves as the probe input. A characteristic of MESHJOIN is that it performs a staggered execution of the hash table build in order to load in stream tuples more steadily. To implement this staggered execution the algorithm uses a queue. The algorithm makes no assumptions about data distribution or the organization of the master data, hence there is no master data index. The algorithm always removes stream tuples from the end of the queue, as they have been matched with all master data partitions.

R-MESHJOIN (reduced Mesh Join) [10] clarifies the dependencies among the components of MESHJOIN. As a result the performance is improved slightly. However, R-MESHJOIN implements the same strategy as the MESHJOIN algorithm for accessing the disk-based master data, using no index.

Partitioned Join [4] improved MESHJOIN by using a two-level hash table, attempting to join stream tuples as soon as they arrive, and using a partition-based wait buffer for other stream tuples. The number of partitions in the wait buffer is equal to the number of partitions in the disk-based master data. The algorithm uses these partitions as an index, for looking up the master data. If a partition in a wait buffer grows larger than a preset threshold, the algorithm loads the relevant partition from the master data into memory. The algorithm allows starvation of stream tuples as tuples can stay in a wait buffer indefinitely if the buffer's size threshold is not reached.

Semi-Streaming Index Join (SSIJ) [3] was developed recently to join stream data with disk-based data. In general, the algorithm is divided into three phases: the pending phase, the online phase and the join phase. In the pending phase, the stream tuples are collected in an input buffer until either the buffer is larger than a predefined threshold or the stream ends. In the online phase, stream tuples from the input buffer are looked up in cached disk blocks. If the required disk tuple exists in the cache, the join is executed. Otherwise, the algorithm flushes the stream tuple into a stream buffer. When the stream buffer is full, the join phase starts where master data partitions are loaded from disk using an index and joined until the stream buffer is empty. This means that as partitions are loaded and joined, the join becomes more and more inefficient: partitions that are joined later can potentially join only with fewer tuples because the stream buffer is not refilled between partition loads. By keeping the stream buffer full and selecting lookup elements carefully the performance could be improved.

CACHEJOIN [9] is an extension of HYBRIDJOIN, which adds an additional cache module to cope with Zipfian stream distributions. This is similar to Partitioned Join and SSIJ, but a tuple-level cache is used instead of a page-level cache to use the cache memory more efficiently. CACHEJOIN is able to adapt its cache to changing stream characteristics, but similar to HYBRIDJOIN, it uses the last queue element as a lookup element for tuples that were not joined with the cache. SSCJ [11] is an improved version of CACHEJOIN, which optimizes the manipulation of master data tuples in the cache module. While CACHEJOIN uses a random approach to overwrite tuples in the cache when it is full, SSCJ overwrites the least frequent tuples. However, both SSCJ and CACHEJOIN use the same suboptimal strategy to access the queue.

3 Background

Semi-stream joins which implement staggered execution of stream data mostly use a queue data structure [13, 14, 10, 8, 9, 11]. The key role of this queue component is to keep track of every stream tuple in memory with respect to loading time. The other purpose of the queue is to ensure that a stream tuple which enters into memory will certainly be processed. Moreover, some of these semi-stream joins [8, 9, 11] also use these queue values as lookup values to load master data into memory via an index.

3.1 HYBRIDJOIN-L and CACHEJOIN-L

We recently presented HYBRIDJOIN [8], an index-based semi-stream join with a simple architecture as shown in Fig. 1. HYBRIDJOIN [8], addresses the issue

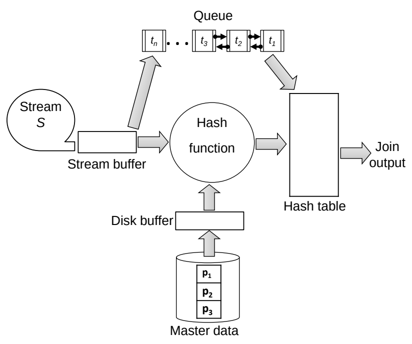


Fig. 1. Data structures and architecture of HYBRIDJOIN

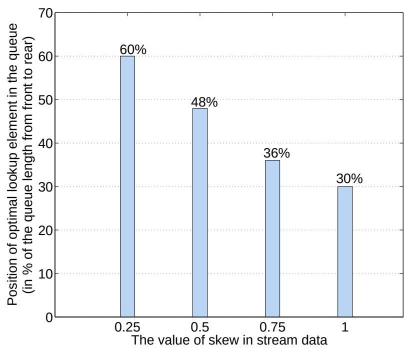


Fig. 2. Analysis of position of the optimal lookup element in the HYBRIDJOIN-L queue using different values of the skew

of accessing disk-based master data efficiently. Similar to SSIJ, an index based strategy to access the disk-based master data is used, but every master data partition load is amortized by joining over a full stream tuple queue. HYBRIDJOIN uses the last queue element as lookup element, which means that unlike Partitioned Join [4] it prevents starvation. However, the choice of the last queue element as lookup element is suboptimal.

Since HYBRIDJOIN uses the oldest tuple of the queue as lookup element before a partition is loaded again after a join, new tuples matching that partition need to move all the way from the beginning to the end of the queue. For frequent partitions (partitions that have more matches) in master data are loaded not much more often than the common partitions. Therefore, the choice of the oldest tuple of the queue as lookup element is not optimal. Recently, we presented our work to determine a position for optimal lookup element from the queue [12]. This lookup element is then used as an index to load the master data partition in memory.

We provided an explanation of the achievement by introducing the concepts of stream probability and load probability. Based on these concepts it was empirically determined that the position at about 30% of the queue size contains a lookup element that results in a particularly high service rate for a skew of 1. This improved the performance of HYBRIDJOIN, and we refer to this new improved algorithm as HYBRIDJOIN-L.

We also demonstrated this concept by applying to our existing CACHEJOIN [9] yielding CACHEJOIN-L. We have shown in the past that this still delivers an improvement over both CACHEJOIN and HYBRIDJOIN-L [12]. This means, that even with a cache, the careful optimization of master data access of frequent pages brings advantages.

3.2 Limitations and Problem Definition

HYBRIDJOIN-L and CACHEJOIN-L perform significantly better than their counterparts those use only the oldest element in the queue for lookup. However, there are limtations. In [12] we were concerned with the optimal lookup element position solely for the skew value of 1. However, in further experiments we have confirmed that the optimal position of the lookup element is quite variable depending on the data, and specifically depending on the skew in the data. We present here the results of our experiment, and we use HYBRIDJOIN-L to illustrate the effect. We ran HYBRIDJOIN-L algorithm with synthetic stream data with different values for the skew and measured the position of the optimal lookup element in the queue, maximising throughput. The results of the experiment are presented in Figure 2. From the figure we can observe that the position of the optimal lookup element in the queue moves forward to the young end of the queue (where new stream tuples are inserted) as the skew increases, reaching a position of 30 percent for a skew exponent of 1. For less pronounced skew with an exponent of 0.25 the optimal value is nearer to the other end of the queue.

Therefore both algorithms HYBRIDJOIN-L and CACHEJOIN-L would need further adaptations to perform optimally. Furthermore, even in the optimal position, the early lookup does not make a good choice every time. In a substantial fractions of the lookup, the element found is not on a frequent page and hence the early lookup is inefficient. The advantage only materializes in the average behavior. Therefore the question arises, whether we can avoid the bad cases and load a frequent page, at the right time, every time. This is what we will propose now.

4 Priority Queue of Frequent Pages

The improvement first offered in HYBRIDJOIN-L is based on the observation that unmatched tuples in the queue take up space. Therefore it is best for achieving a high service rate to load frequent pages as soon as a certain number of tuples in the queue is expected to match with this page. In HYBRIDJOIN-L this is achieved with the early lookup element: this element has been shown to be preferentially from a frequent page. However if we know which pages are frequent, we can use a more direct approach. A page is frequent if it has a high stream probability: This is the probability that if we pick a random stream tuple, that stream tuple belongs to that page [12]. We denote this value for the page here as *page.frequency*. This value can be easily observed at the time the page is first looked up: the observed number of matches divided by the queue length gives a good estimate of the frequency due to the law of large numbers: cases where an infrequent page is just accidentally matched by many stream tuples (and therefore this infrequent page enters the queue) are rare. For a frequent page, if we wait longer with loading that page we get more matches with a single lookup. However the many tuples that accumulate use up space in the hashtable, and they do that for a longer time. Therefore there is a tradeoff value at which it makes sense to load the page, we call this *pageThreshold*. Since new incoming stream tuples belong to random pages (but we cannot tell which tuple belongs to which page without accessing the disk) we know that after a certain number of new stream tuples have been entered into the queue, there is an expected number of tuples matching a given page. In fact for every page (not only the frequent ones), the expected number of stream tuples matching that page increases linearly with the number of stream tuples entered into the queue. Therefore if this expected number is larger as $pageThreshold$, then we should load the page. This is the case after $pageThreshold * \frac{h_S}{page.frequency}$ new loaded stream tuples. Hence the best strategy is to load each frequent page after a period of so many newly loaded stream tuples. The straightforward way to implement this is to keep a running counter of how many stream tuples have been processed and to keep a priority queue. The pages are inserted into the priority queue at the next counter value when they should be loaded. As soon as the counter value for the top page in the priority queue has been reached, this page is popped from the priority queue, the page is loaded and processed according to the algorithm for the disk phase, and finally the page is inserted with the new counter value $ptc + pageThreshold * \frac{h_S}{page.frequency}$. We do not discuss here issues of resetting the counter in order to prevent overflow since these are trivial exercises. We also do not discuss here removal of pages from the priority queue. This is trivial and can be based on the observed number of matches. As soon as the number of matches is below a certain threshold the page can be removed from the queue. Likewise the frequency of a page can steadily be adapted, by using a gliding average. A gliding average with an exponential decay model simple and does not require any further datastructure: Each time the page is loaded, its current number of matches c is observed, and we update according to $page.frequency = (page.frequency * d) + c * (1 - d)$ with a parameter d that controls the decay, e.g. $d = 0.9$ would be a typical value that smoothens out accidental changes in page frequency. This gliding average was however not installed in the algorithms used in our apparatus, since it would introduce an element of arbitrariness in the measurements (which decay parameter should we choose?) and the test data that we use here do not employ a drift of the skew. Nevertheless the algorithm is adaptive as it is now: if it is run with data with different skew values, it will adapt to whatever skew is present, without any configuration necessary.

5 Experiments

In this section we present an extensive experimental study of our algorithms. We compare the performance of all the algorithms using synthetic, TPC-H, and real-life datasets.

5.1 Experimental Setup

Hardware and software specifications: We performed our experiments on a *Intel-i5* with 8GB main memory and 250GB Solid Sate Drive (SSD) as secondary

Table 1. Data specifications for synthetic dataset

Parameter	**value**
Total allocated memory M	50MB to 250MB
Size of disk-based master data R	0.5 million to 8 million tuples
Size of each disk tuple v_R	120 *bytes*
Size of each disk page p_{size}	$8 * 2^{10}$ *bytes*
Size of each stream tuple v_S	20 *bytes*
Size of each node in the queue	12 *bytes*
Size of each node in the priority queue PQ	16 *bytes*
Stream data	based on Zipf's law (skew value from 0.25 to 1)

storage. We implemented our experiments in Java using the Eclipse IDE. As join attribute values can be duplicated in stream data due to the attribute being a foreign key, a hash table is needed that can store multiple values against one value of the master data. The hash table provided by the Java library does not support this feature, therefore `org.apache.MultiHashMap` was used.

Measurement strategy: The service rate of the join is measured by calculating the number of tuples processed in a unit second. In each experiment, the algorithm first completed their warmup phase before starting the actual measurements. These kinds of algorithms normally need a warmup phase to tune their components with respect to the available memory resources, so that each component can deliver a maximum service rate. For each service rate measurement, we calculated the 95% confidence interval. The calculation of the confidence intervals is based on 1000 to 4000 measurements for one setting. During the execution of the algorithm, no other application was running in parallel. We used a constant stream arrival rate throughout a run in order to measure the service rate for all algorithms.

Data specifications: We analyzed the service rate of the algorithms using synthetic, TPC-H, and real-life datasets. The master data R was stored on disk using a MySQL database. Both the algorithms read master data from the database. To measure the I/O cost more accurately, we set the fetch size for `ResultSet` equal to the disk buffer size.

Synthetic data. The stream dataset we used is based on a Zipfian distribution. We tested the service rate of all algorithms by varying the skew value from 0.25 (lightly skewed) to 1 (highly skewed). The master data we used was unsorted and had an index. Moreover, the size of the master data could be changed online. We used memory, master data, disk tuple, stream tuple and queue *pointer* sizes similar to original CACHEJOIN-L and HYBRIDJOIN-L. The detailed specifications of our synthetic dataset are shown in Table 1.

TPC-H. We also analyzed the service rate of all algorithms using the TPC-H dataset, which is a well-known decision support benchmark. We created the dataset using a scale factor of 100. More precisely, we used the table `Customer` as master data and the table `Order` as stream data. In table `Order` there is one

foreign key attribute `custkey`, which is a primary key in the `Customer` table, so the two tables can be joined. Our `Customer` table contained 20 million tuples, with each tuple having a size of 223 bytes. The `Order` table contained the same number of tuples, with each tuple having a size of 138 bytes. The plausible scenario for such a join is to add customer details corresponding to an order before loading the order into the warehouse.
Real-life data. We also compared the service rate of all algorithms using a real-life dataset[1]. This dataset basically contains cloud information stored in a summarized weather report format. It was also used to evaluate the original MESHJOIN, CACHEJOIN-L and HYBRIDJOIN-L. We consider our master data table by combining meteorological data corresponding to months April and August, while consider the stream data by combining data files from December. The master data table contains 20 million tuples, while the streaming data table contains 6 million tuples. The size of each tuple in both the master data table and the stream data table is 128 bytes. Both tables are joined using a common attribute, longitude (`LON`). The domain of the join attribute is the interval [0,36000].

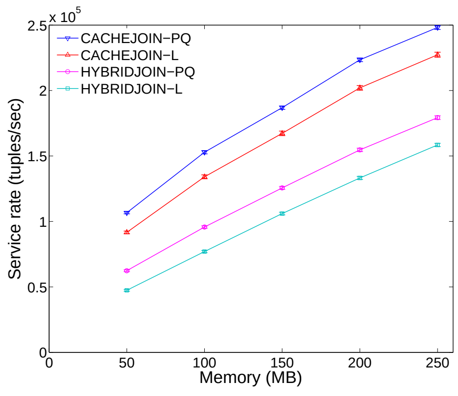
5.2 Service Rate Analysis

We conducted the service rate analysis with respect to three key parameters: the size of the master data table R, the total memory available, and the value of skew in the Zipfian distribution. For the sake of brevity, we restricted the discussion for each parameter to a one-dimensional variation, i.e. we vary one parameter at a time.

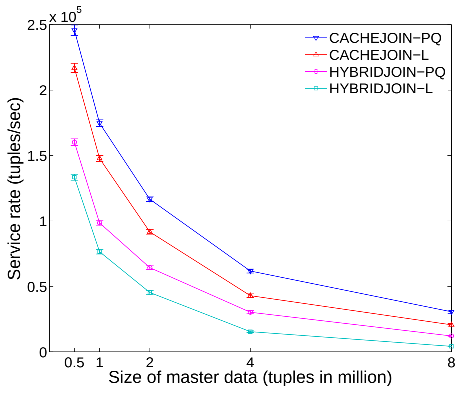
Analysis by varying size of memory: In this experiment we compared the service rate of all algorithms while varying the memory size from 50MB to 250MB, with the fixed size of R equal to 2 million tuples and skew value in the stream data is equal to 1. Figure 3(a) presents the results of the experiment. From the figure it can be noted that for even a small memory size (50MB) both our algorithms perform noticeably better than the existing algorithms and this improvement increases with the increase in the memory size. Furthermore in case of HYBRIDJOIN-PQ the scale of improvement is even better as because of no cache component, PQ takes the full advantage of the skew in stream data.

Analysis by varying size of R: In this experiment we compared the service rate of all the algorithms with different sizes of R. We keep fixed memory size (50MB) and skew value is equal to 1. The results of the experiment are shown in Figure 3(b). From the figure it can be seen that again both the new algorithms perform considerably better than the existing algorithms. Also in case of increasing the size of R the service rate of both CACHEJOIN-PQ and HYBRIDJOIN-PQ does not decrease with that rate as it decreases in CACHEJOIN-L and HYBRIDJOIN-L respectively. The plausible reason behind this behaviour is

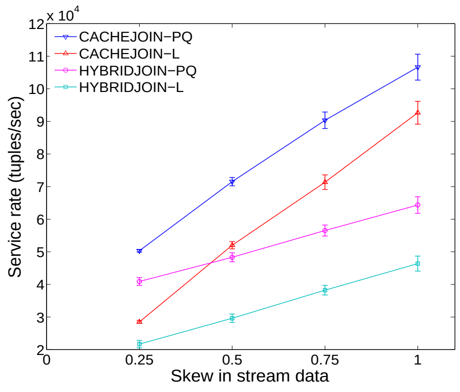
[1] This dataset is available at: `http://cdiac.ornl.gov/ftp/ndp026b/`



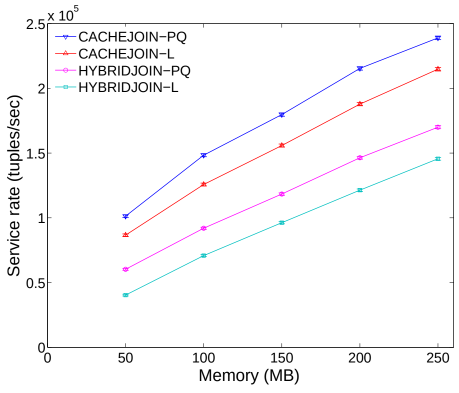
(a) Service rate vs memory



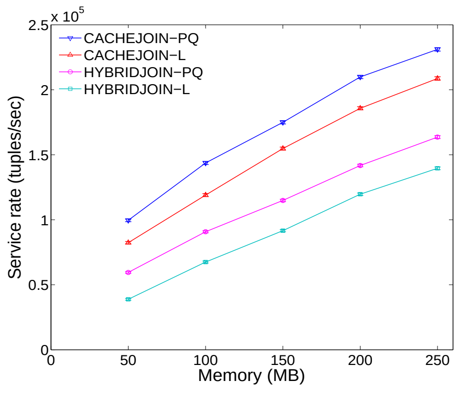
(b) Service rate vs size of R



(c) Service rate vs skew



(d) TPC-H datasets



(e) Real-life dataset

Fig. 3. Service rate analysis

that, in the existing algorithms the position of optimal lookup element changes by increasing the size of R.

Analysis by varying skew value: In this experiment we compared the service rate of all the algorithms while varying the skew value in streaming data. To vary the skew, we varied the Zipfian exponent from 0.25 to 1. At 0.25 the input stream S has a light skew, while at 1 the stream has a strong skew. The size of R was fixed at 2 million tuples and the available memory was set to 50MB. The results presented in Figure 3(c) show that both CACHEJOIN-PQ and HYBRIDJOIN-PQ perform significantly better than CACHEJOIN-L and HYBRIDJOIN-L respectively. Particularly for small values of skew this difference of improvement is more prominent. This is an evidence for our argument that by changing the skew value in stream data the position of optimal lookup element in the queue also changes. We do not present data for skew values larger than 1, which would imply short tails. Also we do not consider fully uniform stream data (i.e. Zipfian exponent is equal to 0) as this is very unlikely in supermarket transactional data.

TPC-H and real-life datasets: In these experiments we measured the service rate produced by all the algorithms at different memory settings. The results of using both TPC-H and real-life datasets are shown in Figure 3(d) and Figure 3(e) respectively. From the both figures it can be noted that under all memory settings both CACHEJOIN-PQ and HYBRIDJOIN-PQ outperform existing CACHEJOIN-L and HYBRIDJOIN-L respectively .

6 Conclusions

Recently we published the strategy of finding a position of optimal lookup element from the queue. However, the approach suffers with two drawbacks (a) it is not necessary that the lookup element always gives a frequent page from the master data, a page that has frequent number of matches with stream tuples in memory (b) the position of the lookup element can vary by varying the nature of stream input, e.g. the value of skew in stream data. In this paper we addressed these issues by introducing a new component called priority queue. The priority queue keeps the record of frequent disk pages by storing their page IDs. While the criteria of deciding that a page is frequent is based on the total number of matches in whole queue size stream data against that page. In this way unlike to the existing approach, every element in the priority queue gives a frequent page of master data and this does not affect by changing the skew in stream data. To validate our argument we implemented our new strategy to the existing CACHEJOIN-L and HYBRIDJOIN-L algorithms and named them CACHEJOIN-PQ and HYBRIDJOIN-PQ. We provided experimental results that show that the both new algorithms perform significantly better than the existing ones for all synthetic, TPC-H and real-life datsets.

In the future, we will extend our work to consider many-to-many equijoins and certain classes of non-equijoins.

References

1. C. Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More.* Hyperion, 2006.
2. A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical Report 2002-57, Stanford InfoLab, 2002.
3. M. Bornea, A. Deligiannakis, Y. Kotidis, and V. Vassalos. Semi-streamed index join for near-real time execution of ETL transformations. In *IEEE 27th International Conference on Data Engineering (ICDE'11)*, pages 159 –170, april 2011.
4. A. Chakraborty and A. Singh. A partition-based approach to support streaming updates over persistent data in an active datawarehouse. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.
5. L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with datadepot. In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 847–854, New York, NY, USA, 2009. ACM.
6. A. Karakasidis, P. Vassiliadis, and E. Pitoura. ETL queues for active data warehousing. In *IQIS '05: Proceedings of the 2nd International Workshop on Information Quality in Information Systems*, pages 28–39. ACM, 2005.
7. M. A. Naeem, G. Dobbie, and G. Weber. An event-based near real-time data integration architecture. In *EDOCW '08: Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops*, pages 401–404, Washington, DC, USA, 2008. IEEE Computer Society.
8. M. A. Naeem, G. Dobbie, and G. Weber. HYBRIDJOIN for near-real-time data warehousing. *International Journal of Data Warehousing and Mining (IJDWM)*, 7(4), 2011.
9. M. A. Naeem, G. Dobbie, and G. Weber. A lightweight stream-based join with limited resource consumption. In *DaWaK '12: Data Warehousing and Knowledge Discovery*, pages 431–442. Springer, 2012.
10. M. A. Naeem, G. Dobbie, G. Weber, and S. Alam. R-MESHJOIN for near-real-time data warehousing. In *DOLAP'10: Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP*, Toronto, Canada, 2010. ACM.
11. M. A. Naeem, G. Weber, G. Dobbie, and C. Lutteroth. SSCJ: A semi-stream cache join using a front-stage cache module. In *DaWaK '13: 15th International Conference on Data Warehousing and Knowledge Discovery*, pages 236–247. Springer, 2013.
12. M. A. Naeem, G. Weber, C. Lutteroth, and G. Dobbie. Optimizing queue-based semi-stream joins with indexed master data. In *DaWaK '14: Data Warehousing and Knowledge Discovery*, pages 171–182. Springer, 2014.
13. N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N. Frantzell. Supporting streaming updates in an active data warehouse. In *ICDE 2007: Proceedings of the 23rd International Conference on Data Engineering*, pages 476–485, Istanbul, Turkey, 2007.
14. N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N. Frantzell. Meshing streaming updates with persistent data in an active data warehouse. *IEEE Trans. on Knowl. and Data Eng.*, 20(7):976–991, 2008.
15. E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.