Optimizing Queue-based Semi-Stream Joins with Indexed Master Data

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

[1]School of Computer and Mathematical Sciences, Auckland University of Technology
[2]Department of Computer Science, The University of Auckland
[1]mnaeem@aut.ac.nz
[2]{gerald,christof,gill}@cs.auckland.ac.nz

Abstract. In Data Stream Management Systems (DSMS) semi-stream processing has become a popular area of research due to the high demand of applications for up-to-date information (e.g. in real-time data warehousing). A common operation in stream processing is joining an incoming stream with disk-based master data, also known as semi-stream join. This join typically works under the constraint of limited main memory, which is generally not large enough to hold the whole disk-based master data. Many semi-stream joins use a queue of stream tuples to amortize the disk access to the master data, and use an index to allow directed access to master data, avoiding the loading of unnecessary master data. In such a situation the question arises which master data partitions should be accessed, as any stream tuple from the queue could serve as a lookup element for accessing the master data index. Existing algorithms use simple safe and correct strategies, but are not optimal in the sense that they maximize the join service rate. In this paper we analyze strategies for selecting an appropriate lookup element, particularly for skewed stream data. We show that a good selection strategy can improve the performance of a semi-stream join significantly, both for synthetic and real data sets with known skewed distributions.

Keywords: Real-time Data Warehousing, Stream processing; Join; Performance measurement

1 Introduction

Real-time data warehousing plays a prominent role in supporting overall business strategy. By extending data warehouses from static data repositories to active data repositories, business organizations can better inform their users and make effective timely decisions. In real-time data warehousing the changes occurring at source level are reflected in data warehouses without any delay. Extraction, Transformation, and Loading (ETL) tools are used to access and manipulate transactional data and then load them into the data warehouse. An important phase in the ETL process is a transformation where the source level changes are mapped into the data warehouse format. Common examples of transformations

are unit conversion, removal of duplicate tuples, information enrichment, filtering of unnecessary data, sorting of tuples, and translation of source data keys.

A particular type of stream-based joins called semi-stream joins are required to implement the above transformation examples. In this particular type of stream-based join, a join is performed between a single stream and a slowly changing table. In the application of real-time data warehousing [4, 9, 10], the slowly changing table is typically a master data table while incoming real-time sales data may form the stream.

Most stream-based join algorithms [9, 10, 7, 3, 2, 6, 5] use the concept of staging in order to amortize the expensive disk access cost over fast stream data. The concept of staging means the algorithm loads stream data into memory in chunks, while these chunks are differentiated by their loading timestamps. To implement the concept of staging these algorithms normally use a data structure called a queue. The main role of the queue is to keep track of these stages with respect to their loading timestamps. Some of these algorithms [3, 2, 6, 5] use elements of the queue to look up and load relevant disk-based master data through an index. In this paper we call these queue elements *lookup elements*.

Most of the above algorithms choose the oldest value of the queue as lookup element. The process of choosing the oldest tuple in the queue is at first glance intuitive: this tuple must be joined eventually to avoid starvation, and since it was not processed before it is now due. Hence choosing the oldest tuple in the queue ensures correctness. However, as we will show in this paper, this strategy is not optimal with regard to join service rate. The optimal lookup element is the element for which the most useful partition of master data is loaded into memory, i.e. the partition which will join the most stream tuples currently in the queue.

This paper addresses the challenge of finding near-optimal strategies for selecting the lookup element. We have identified two *eviction aims* that influence the choice of the stream tuple for master data lookup: 1) to maximize the expected number of stream tuples that are matched and hence removed from the queue, and 2) to limit the cost (or lost opportunity) that a tuple incurs while sitting in the queue. An important property of stream tuple behavior in our context is the average frequency of matches to a whole master data partition, i.e. the average number of tuples that is matched when the partition is loaded from disk.

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 [9, 10] 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) [7] 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 [3] 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) [2] 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.

One of our algorithms, HYBRIDJOIN [5], addresses the issue 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 it prevents starvation. However, as will be explained in this paper, the choice of the last queue element as lookup element is suboptimal.

CACHEJOIN [6] 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.

Recently, we presented an improved version of CACHEJOIN called SSCJ [8], 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. To the best of our knowledge, the CACHEJOIN/SSCJ class of semi-stream join algorithms is currently the fastest when considering skewed data. SSCJ and CACHEJOIN use the same suboptimal strategy to access the queue, and this can be improved with the approach presented here. Due to space limitations we test this approach using HYBRIDJOIN and CACHEJOIN only.

3 Problem Definition

This section defines the problem we are addressing, using the existing HYBRIDJOIN algorithm as an example for clarification. HYBRIDJOIN has a simple architecture, using a queue to load disk-based master data into memory, and is therefore particularly suitable for illustration.

Fig. 1 presents an overview of HYBRIDJOIN. The master data on disk contains three partitions, $p_1, \ldots, p_3$. Partitions are loaded into memory through the *disk buffer*, which can hold one partition. In the original algorithm the stream tuples are stored in a hash table and the queue stores pointers to these stream tuples. To make our discussion clearer, we will refer to the queue as if the stream tuples are directly contained therein, as our focus is to highlight the behavior of the queue. The queue is implemented as a doubly-linked list, allowing the random deletion of stream tuples.

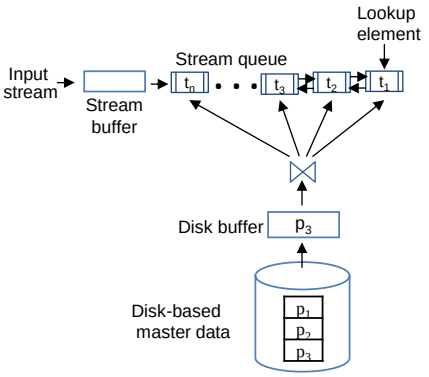


Fig. 1. Overview of HYBRIDJOIN

In each iteration the algorithm loads a chunk of stream tuples into the queue and a partition of master data into the disk buffer. To decide which partition will be loaded into the buffer, HYBRIDJOIN retrieves the oldest (i.e. last) element of the queue and uses it as a lookup element in the index for the master data table. Once the relevant partition is loaded into the disk buffer, the algorithm performs a join between the master data tuples and stream tuples. During the join operation the algorithm removes the matched stream tuples, which lie at random positions, from the queue. With parameter skew some tuples occur more frequently in the stream data. Consequently, there can be many matches of stream tuples in the queue against one master data tuple.

For a master data partition, we define the *stream probability* as the probability that a random stream tuple matches a master data record on that partition. If the stream probability of a partition is at least $1/h_S$, with h_S the size of the queue in number of tuples, we call this a *common partition*. Among the common partitions there are typically some *high-probability partitions*.

The purpose of queueing stream tuples is to amortize a master data partition access by processing several stream tuples with it. After a master data access, all the h_S stream tuples in the queue are processed and potentially joined. We define the *load probability* of a master data partition as the probability that a random stream tuple is *used* to load that partition, i.e. that a lookup element matches the master data partition. The load probability of a partition is smaller than its stream probability because after a partition access all the stream tuples matching it are removed from the queue, reducing the occurrence of lookup elements that match the partition. Due to the amortization process over a queue of h_S elements, the load probability is in fact always smaller than $1/h_S$. After one lookup for a partition p, the earliest stream tuple that can lead to a lookup of p has to be a fresh stream tuple from the same page that enters the queue after the last lookup. 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. These stream tuples must therefore be more than h_S stream tuples after the last lookup.

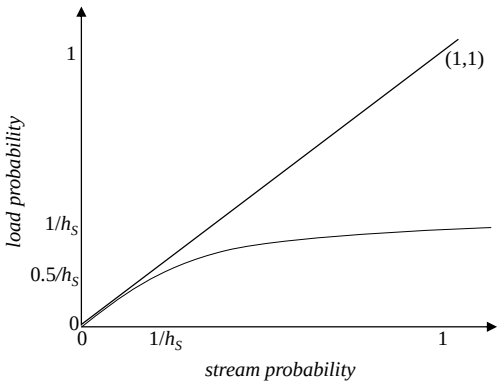


Fig. 2. The load probability, shown as a function of the stream probability.

Hence there is a *saturation effect*: If we look at partitions in order of increasing stream probability, the load probability first is equal to the average frequency, but can never go beyond $1/h_S$. We furthermore observe that for all common partitions the load frequency will be close to, or larger than $1/(2h_S)$, since after one loookup of a partition, we expect the partition to be again among the next h_S stream tuples entering the queue. Hence we see that, interestingly, the high-probability partitions are loaded not much more often than the common partitions. A qualitative illustration of how the load probability depends on the stream probability is shown in Fig. 2.

It is important to realize that this behavior is not optimal. Although at first glance it seems to optimally amortize disk accesses, this is not the case. For high-probability partitions, a large number of matching stream tuples accumulate in the queue before the partition is loaded again. This takes up space in the queue,

therefore these tuples should be evicted earlier. Hence we have to reduce the saturation effect for high-probability partitions, i.e. we have to ensure that high-probability partitions are loaded more frequently.

4 Proposed Solution

We propose an improved strategy that reduces the saturation effect for high-probability partitions by loading them more often, but not too often (which would reduce the number of joins). We will now argue that we can ensure this by adjusting the position of the lookup element, i.e. the position in the queue of the stream tuple that is used to look up the master data partition to load next. It is clear, however, that we have to retain the oldest queue tuple as a lookup element for correctness reasons: we have to ensure that tuples that have come to this position are eventually evicted. We therefore propose a strategy that alternates between the last element of the queue for lookup and an earlier element. We now discuss why for an earlier lookup element position, high-probability pages will be loaded more often. We assume each stream tuple in the queue is annotated with the frequency of matching of the master data partition.

In Fig. 3, we illustrate our expectation for the HYBRIDJOIN strategy at an arbitrary point in time; this is purely an illustration and is not intended to be quantitatively correct. The crucial point is that for each of the high-frequency master data partitions, the time of last lookup (expressed in stream tuples) was on average half a queue length ago. Therefore, the tuples of each of these partitions will only have advanced a fraction of the queue, on average half of the queue. This is a random process, so at the start of the queue, many frequent pages will have started to reappear, and their prevalence diminishes towards the end of the queue. The figure shows a made-up example where the queue contains join attribute values of 50 stream tuples and the corresponding matching frequencies lie between 1 and 5. We also have drawn a moving average for the corresponding matching frequencies. Again, the shape of the moving average is only indicating the tendency to drop towards the end of the queue; we do not consider its exact shape here. The figure also illustrates that some of the tuples with corresponding high matching frequencies will come through to the end of the queue. However, HYBRIDJOIN accumulates the least frequent stream tuples towards the end of the queue, where the lookup element is situated. The join attributes towards the

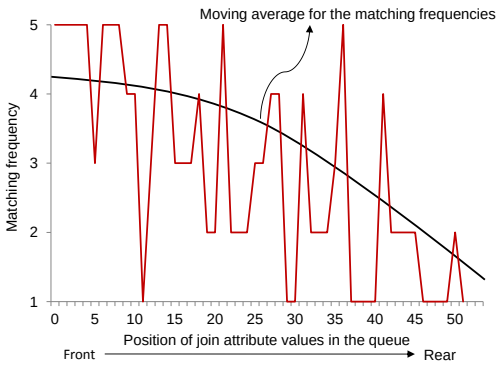


Fig. 3. Queue analysis

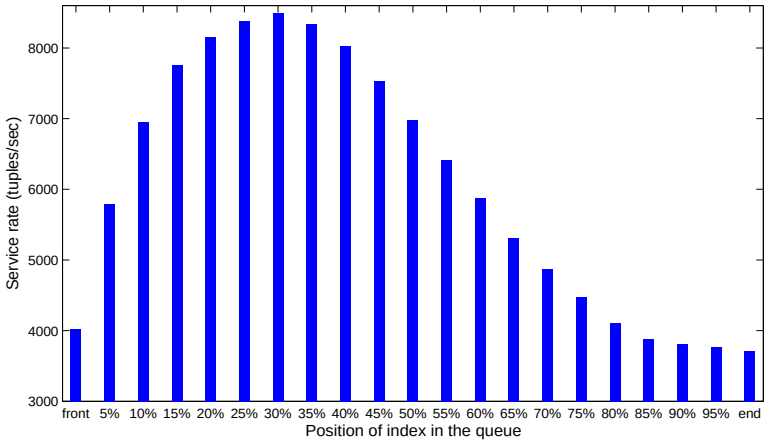


Fig. 4. Performance analysis of HYBRIDJOIN by accessing the queue at different positions

front of the queue have higher matching frequencies as compared to the older stream tuples in the queue. Hence if any algorithm, such as HYBRIDJOIN or CACHEJOIN, chooses only the oldest stream position at the end of the queue as a lookup position, it will under-represent the high-probability pages, as we have predicted in our considerations about the saturation effect.

With the introduction of a second lookup position earlier in the queue we can therefore expect to hit high-probability pages more frequently. The second lookup position will lead to a behavior different from the one in Fig. 3. The front of the queue, i.e. the newest tuples, is not be the best position for the new lookup element since it would over-represent high-probability pages; there would be a high probability that the most frequent page gets loaded before new tuples for that page had an opportunity to accumulate in the queue. As explained previously, the algorithm will alternate between the two lookup positions, the new position and the oldest element in the queue, as some tuples will remain in the queue for a long time otherwise.

We used an empirical approach to find an optimal position for the new lookup element and present the effect of the new lookup position on the performance of HYBRIDJOIN. Again, HYBRIDJOIN is chosen because it is the simplest algorithm this new approach can be applied to. We conducted an experiment in which we measured the performance of the algorithm while varying the new lookup position. We started with a lookup position at the front of the queue and then moved it in 5% steps towards the end of the queue.

The results of our experiment are shown in Fig. 4. From the figure it can be observed that a lookup element at the front of the queue is not optimal, as we predicted, and as we move the lookup position away from the front the performance starts improving. This behavior continues up to a certain fraction of the queue length (about 30% of the queue size in our experiment, i.e. the

index of the entry in the queue is approx. $0.3h_s$) and after that the throughput starts to decrease again. We have validated this fraction of 30% with different parameters and used it in in all the performance experiments described later on.

Given a certain index $c \cdot h_s$, $0 < c < 1$ for the lookup position, one could assume that a partition is loaded immediately if a corresponding tuple reaches the new lookup position. Then the partition would be loaded $1/c$ times more frequently as compared to a lookup position at the end of the queue. However, tuples can "sneak through" as the lookup element at the end of the queue is used alternatingly, effectively decreasing the load probability. The probability that a tuple sneaks through is independent of the page. The expected distance that the tuple can then move (this is $c' - c$) is proportional to the stream probability.

5 Experiments

5.1 Setup

We performed our experiments on a *Pentium-i5* with 8GB main memory and 500GB hard drive as secondary storage running Java. We compared the original HYBRIDJOIN [5] and CACHEJOIN [6] algorithms with optimized versions using the proposed approach, using the cost models presented with the original algorithms to distribute the memory among their components. The master data R was stored on disk using a MySQL database with an index.

The two algorithms retrieve master data using a lookup element from a stream tuple queue. Choosing these two algorithms allows us to compare the effects of near optimal lookup elements for algorithms with and without a cache component. HYBRIDJOIN does not have a cache component so all stream tuples are processed through the queue, while CACHEJOIN has an additional cache module that processes the most frequent stream tuples. For this reason, CACHEJOIN is considered an efficient algorithm in comparison to other semi-stream-join algorithms [3, 2, 9, 10, 7]. We analyzed the service rates of all algorithms using synthetic, TPC-H and real-life datasets.

Synthetic datasets: The stream datasets we used is based on a Zipfian distribution, which can be found in a wide range of applications [1]. We tested the service rate of the algorithms by varying the skew value from 0 (fully uniform) to 1 (highly skewed). Details of the synthetic datasets are specified in Table 1.

TPC-H datasets: We also analyzed the service rates using the TPC-H datasets, which is a well-known decision support benchmark. We created the datasets using a scale factor of 100. More precisely, we used the table `Customer` as master data and the table `Order` as the 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.

Table 1. Data specification of the synthetic datasets (similar to those used for the original HYBRIDJOIN and CACHEJOIN algorithms)

Parameter	**Value**
Total allocated memory M	50MB to 250MB
Size of disk-based relation R	0.5 *million* to 8 *million tuples*
Size of each disk tuple	120 *bytes*
Size of each stream tuple	20 *bytes*
Size of each node in the queue	12 *bytes*
Data set	Based on Zipf's law (exponent varies from 0 to 1)

Real-life datasets: We also compared the service rates of the algorithms using a real-life datasets[1]. These datasets contains cloud information stored in a summarized weather report format. The master data table is constructed by combining meteorological data corresponding to the months April and August, and the stream data by combining data files from December. The master data table contains 20 million tuples and the stream 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].

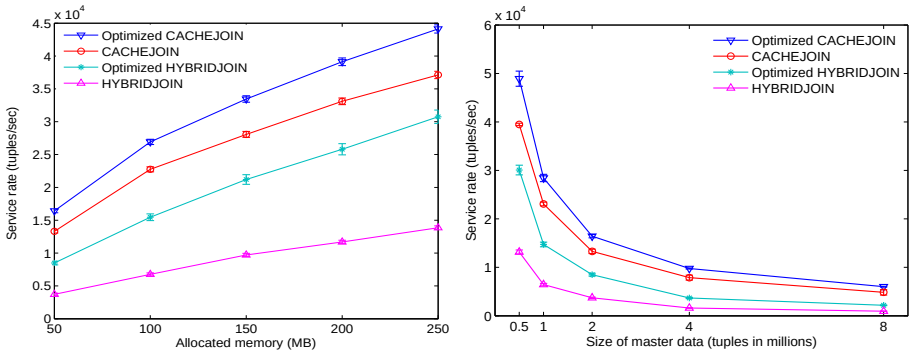
Measurement strategy: The performance or service rate of a join is measured by calculating the number of tuples processed in a unit second. In each experiment, the algorithms first completed a warmup phase before starting the actual measurements. These kinds of algorithms normally need a warmup phase to tune their components with respect to the available memory resources, so that each component can deliver a maximum service rate. The calculation of the confidence intervals is based on 2000 to 3000 measurements for one setting. During the execution of the algorithm, no other application was running in parallel. The stream arrival rate throughout a run was constant.

5.2 Performance evaluation

We identified three parameters for which we want to understand the behavior of the algorithms. The three parameters are: the total memory available M, the size of the master data table R, and the value of the parameter skew e in the stream data. For the sake of brevity, we restrict the discussion for each parameter to a one-dimensional variation, i.e. we vary one parameter at a time.

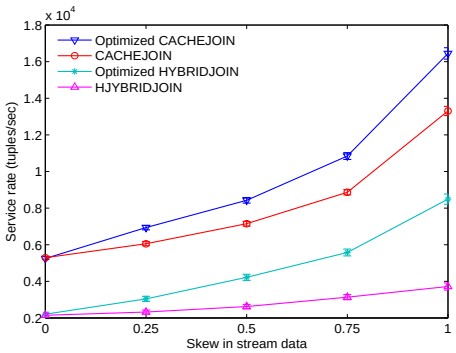
Performance comparisons for different memory budgets: In our first experiment we tested the performance of all algorithms using different memory budgets while keeping the size of R fixed (*2 million tuples*). We increased the available memory linearly from 50MB to 250MB. Fig. 5(a) presents the comparisons of both approaches with and without implementing the optimal lookup element strategy. From the figure the performance improvement in the case of the both algorithms is clear. More concretely, in the case of Optimized

[1] These datasets are available at: `http://cdiac.ornl.gov/ftp/ndp026b/`

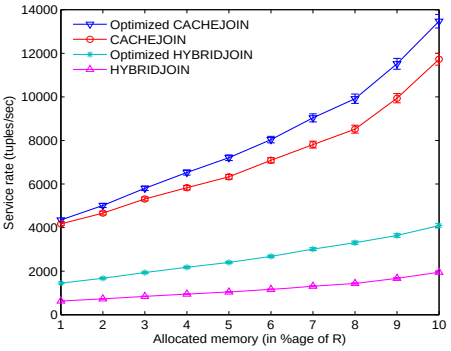


(a) Size of allocated memory varies

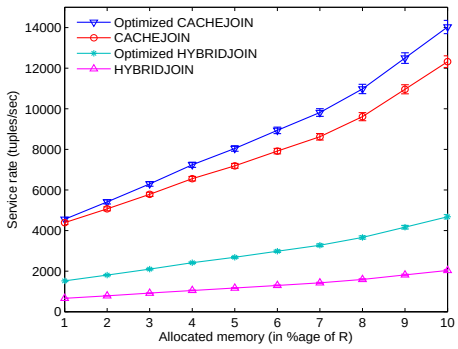
(b) Size of master data on disk varies



(c) Skew in stream data varies



(d) TPC-H datasets



(e) Real datasets

Fig. 5. Performance comparisons

HYBRIDJOIN the algorithm performed 3 times better than the original HYBRIDJOIN. Although the improvement is comparatively smaller in the case of Optimal CACHEJOIN, it is still considerable. The reason the improvement is

smaller is the cache component that processes the most frequent part of the stream data.

Performance comparisons for varying the size of *R*: In our second experiment we tested the performance by varying the size of the disk-based relation R. We chose values for R from a simple geometric progression. The performance results are shown in Fig. 5(b). From the figure it can be seen that Optimized HYBRIDJOIN performed more than twice as well as the original HYBRIDJOIN. Also in the case of Optimized CACHEJOIN the performance improved considerably.

Performance comparisons for different values of skew: In this experiment we compared the service rates of both algorithms with and without the optimal lookup element strategy while varying the skew in the stream data. To vary the skew, we varied the Zipfian exponent from 0 to 1. At 0 the input stream S has no skew, while at 1 the stream was strongly skewed. The size of R was fixed at 2 million tuples and the available memory was set to 50MB. The results presented in Fig. 5(c) show that both optimized algorithms (especially Optimized HYBRIDJOIN) performed significantly better than the existing ones, even for only moderately skewed data. This improvement became more pronounced for increasing skew values. At a skew of 1, Optimized HYBRIDJOIN performs approximately 3 times better than the original HYBRIDJOIN. In the case of Optimized CACHEJOIN the improvement was comparatively smaller but is still noticeable. Our strategy does not add any overhead to the processing cost, therefore in the case of fully uniform data, when the skew is equal to 0, the performance is not worse than that of the original algorithms. We do not present data for skew values larger than 1, which models short tails.

TPC-H datasets: In this experiment we measured the service rates produced by the algorithms at different memory settings. We allocated the size of primary memory as a percentage of the size of R. The results are shown in Fig. 5(d). From the figure it can be noted that the optimized versions of the algorithms performed better than the original algorithms. Especially in the case of Optimized HYBRIDJOIN this improvement is remarkable.

Real-life datasets: We also tested our approaches using real data. The details of the datasets were presented in the beginning of this section. In this experiment we also measured the service rate produced by the algorithms at different memory settings, similar to the one using the TPC-H datasets. The results are shown in Fig. 5(e). From the figure, the performance of the optimized algorithms is again significantly better than that of the original algorithms, supporting our argument.

6 Conclusions

Most semi-stream join algorithms amortize disk accesses to master data over a queue of stream tuples in memory. Several of those algorithms use an index to look up master data partitions for particular elements in that queue. We identified the choice of the lookup element, i.e. the queue stream tuple used as a key in

such an index, as an important and underutilized issue for such algorithms. For example, HYBIRDJOIN and CACHEJOIN always choose lookup elements from the end of their queues. Because of that they under-represent high-probability partitions of disk-based master data and do not fully exploit the characteristics of skew in stream data, resulting in a suboptimal performance.

As a solution, we have proposed a new approach in this paper for choosing an element for index-based master data lookup from a stream tuple queue, based on the position of the lookup element in the queue. The approach alternates between the last queue element to avoid starvation, and an intermediate queue element, balancing the rate in which high-probability partitions are loaded. We provided a theory for the improved behavior and validated it with experiments using HYBIRDJOIN and CACHEJOIN, showing that the optimized algorithms perform significantly better than the original ones.

References

1. Chris Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More.* Hyperion, 2006.
2. M.A. Bornea, A. Deligiannakis, Y. Kotidis, and V. Vassalos. Semi-streamed index join for near-real time execution of ETL transformations. In *ICDE '11: IEEE 27th International Conference on Data Engineering*, pages 159 –170. IEEE Computer Society, 2011.
3. Abhirup Chakraborty and Ajit Singh. A partition-based approach to support streaming updates over persistent data in an active datawarehouse. In *IPDPS '09: IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11. IEEE Computer Society, 2009.
4. Alexandros Karakasidis, Panos Vassiliadis, and Evaggelia Pitoura. ETL queues for active data warehousing. In *IQIS '05: 2nd International Workshop on Information Quality in Information Systems*, pages 28–39. ACM, 2005.
5. M. Asif Naeem, Gillian Dobbie, and Gerald Weber. HYBRIDJOIN for near-real-time data warehousing. *International Journal of Data Warehousing and Mining (IJDWM)*, 7(4), 2011.
6. M Asif Naeem, Gillian Dobbie, and Gerald Weber. A lightweight stream-based join with limited resource consumption. In *DaWaK '12: Data Warehousing and Knowledge Discovery*, pages 431–442. Springer, 2012.
7. M. Asif Naeem, Gillian Dobbie, Gerald Weber, and Shafiq Alam. R-MESHJOIN for near-real-time data warehousing. In *DOLAP '10: ACM 13th International Workshop on Data Warehousing and OLAP*. ACM, 2010.
8. M. Asif Naeem, Gerald Weber, Gillian Dobbie, and Christof 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.
9. N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.E. Frantzell. Supporting streaming updates in an active data warehouse. In *ICDE '07: 23rd International Conference on Data Engineering*, pages 476–485, 2007.
10. Neoklis Polyzotis, Spiros Skiadopoulos, Panos Vassiliadis, Alkis Simitsis, and Nils Frantzell. Meshing streaming updates with persistent data in an active data warehouse. *IEEE Trans. on Knowl. and Data Eng.*, 20(7):976–991, 2008.