SSCJ: A Semi-Stream Cache Join using a Front-Stage Cache Module

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

[1]School of Computing and Mathematical Sciences, Auckland University of Technology
[2]Department of Computer Science, The University of Auckland
Private Bag 92006, Auckland, New Zealand
[1]mnaeem@aut.ac.nz [2]{gerald,gill,lutteroth}@cs.auckland.ac.nz

Abstract. Semi-stream processing has become an emerging area of research in the field of data stream management. One common operation in semi-stream processing is joining a stream with disk-based master data using a join operator. This join operator typically works under limited main memory and this memory is generally not large enough to hold the whole disk-based master data. Recently, a number of semi-stream join algorithms have been proposed in the literature to achieve an optimal performance but still there is room to improve the performance. In this paper we propose a novel Semi-Stream Cache Join (SSCJ) using a front-stage cache module. The algorithm takes advantage of skewed distributions, and we present results for Zipfian distributions of the type that appear in many applications. We analyze the performance of SSCJ with a well known related join algorithm, HYBRIDJOIN (Hybrid Join). We also provide the cost model for our approach and validate it with experiments.

Keywords: Semi-stream processing; Stream-based join; Data warehousing; Performance measurement

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 join, a semi-stream join that joins a single stream with a slowly changing table. Such a join can be applied in real-time data warehousing [6, 4]. In this application, the slowly changing table is typically a master data table. Incoming real-time sales data may comprise the stream. The stream-based join can be used for example to enrich the stream data with master data. In this work we only consider one-to-many equijoins, as they appear between foreign keys and the referenced primary key in another table.

For executing stream-based operations, the large capacity of current main memories as well as the availability of powerful cloud computing platforms means, that considerable computing resources can be utilized. For master data of the right size for example, main-memory algorithms can be used.

However, there are several scenarios, where stream joins that use a minimum of resources are needed. One particular scenario is an organization trying to reduce the carbon footprint of the IT infrastructure. A main memory approach as well as cloud-computing approaches can be power-hungry. Also in the area of mobile computing and embedded devices a low-resource consumption approach can be advantageous. Therefore, approaches that can work with limited main memory are of interest.

In the past, the algorithm HYBRIDJOIN (Hybrid Join) [7] was proposed for joining a stream with a slowly changing table with limited main memory requirements. This algorithm is an interesting candidate for a resource aware system setup. The key objective of this algorithm is to amortize the fast input stream with the slow disk access within limited memory budget and to deal with the bursty nature of the input data stream. Further details about HYBRIDJOIN are presented in Section 3.

Although the HYBRIDJOIN algorithm amortizes the fast input stream using an index-based approach to access the disk-based relation and can deal with bursty streams, the performance can still be improved if some characteristics of stream data are taken into consideration. We are looking for characteristics of data that are considered ubiquitous in real world scenarios. A Zipfian distribution of the foreign keys in the stream data matches distributions that are observed in a wide range of applications [1]. We therefore created a data generator that can produce such a Zipfian distribution. A Zipfian distribution is parameterized by the exponent of the underlying power law. In different scenarios, different exponents are observed, and determine whether the distribution is considered to have a short tail or a long tail. Distributions with a short tail would be more favourable for SSCJ from the outset, therefore we decided not to use a distribution with a short tail in order to not bias our experiment towards SSCJ. Instead we settled on a natural exponent that is observed in a variety of areas, including the original Zipf's Law in linguistics [5] that gave rise to the popular name of these distributions. The main result of our analysis is that SSCJ performs better on a skewed dataset that is synthetic, but following a Zipfian distribution as is found frequently in practice. For our analysis we do not consider joins on categorical attributes in master data, e.g. we do not consider equijoins solely on attributes such as gender.

The rest of the paper is structured as follows. Section 2 presents related work. In Section 3 we describe our observations about HYBRIDJOIN. Section 4 describes the proposed SSCJ with its execution architecture and cost model. Section 5 describes an experimental analysis of SSCJ. 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 this area, focusing on those which are closely related to our problem domain.

A seminal algorithm Mesh Join (MESHJOIN) [9, 10] has been designed especially for joining a continuous stream with a disk-based relation, like the scenario in 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. The algorithm makes no assumptions about data distribution and the organization of the master data. The MESHJOIN authors report that the algorithm performs worse with skewed data.

R-MESHJOIN (reduced Mesh Join) [8] clarifies the dependencies among the components of MESHJOIN. As a result the performance has been improved slightly. However, R-MESHJOIN implements the same strategy as in the MESHJOIN algorithm for accessing the disk-based relation.

One approach to improve MESHJOIN has been a partition-based join algorithm [3], which can also deal with stream intermittence. It uses a two-level hash table in order to attempt to join stream tuples as soon as they arrive, and uses a partition-based waiting area for other stream tuples. For the algorithm in [3], however, the time that a tuple is waiting for execution is not bounded. We are interested in a join approach where there is a time guarantee for when a stream tuple will be joined.

Another recent approach, Semi-Streaming Index Join (SSIJ) [2] joins stream data with disk-based data. SSIJ uses page level cache i.e. stores the entire disk pages in cache. It is possible that not all the tuples in these pages are frequent in the stream and as a result the algorithm can perform suboptimally. Also the authors do not include the mathematical cost model for the algorithm.

3 Problem Definition

To clarify our observations, we present the HYBRIDJOIN algorithm in detail and at the end of this section we formulate an argument that we focus on in this paper.

A semi-stream join algorithm, HYBRIDJOIN, was based on two objectives. The first objective was to amortize the disk I/O cost over the fast input data stream more effectively by introducing an index-based approach to access the disk-based relation R. The second objective was to deal with the bursty nature of a data stream effectively. An abstract level working overview of HYBRIDJOIN is presented in Figure 1 where we consider m partitions in the queue to store stream tuples and n pages in disk-based relation R. In order to keep it simple, currently, we assume that the stream tuples are stored in a queue rather than in a hash table and the join is directly performed with the queue. The disk buffer is used to load one disk page into memory.

The key to HYBRIDJOIN is that, for each iteration the algorithm reads the oldest tuple from the queue and using that tuple as an index it loads the relevant disk page into the disk buffer. After loading the disk page into memory, the algorithm matches each tuple on the disk page with the stream tuples in the

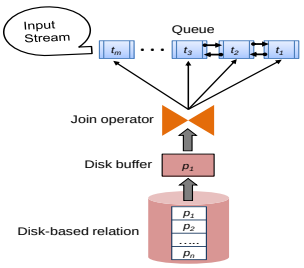


Fig. 1. HYBRIDJOIN working overview

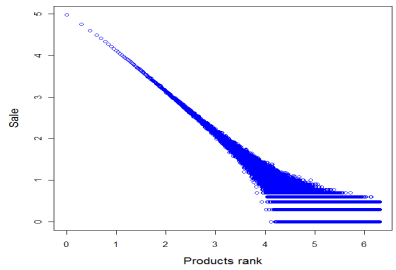


Fig. 2. Current market sales (on log-log scale)

queue. When a match is found, the algorithm generates a tuple as an output after deleting it from the queue. In the next iteration, the algorithm again reads stream input, extracts the oldest element from the queue, loads the relevant disk page into the disk buffer and repeats the entire procedure.

Although HYBRIDJOIN accesses R using an index reducing the I/O cost as compared to the other approaches, described in Section 2, if we analyse the current market sales then we observe that I/O cost can also be minimized further, ultimately improving the performance. To elaborate, we consider a benchmark which is based on current market sales based on the 80/20 rule [1]. According to this rule, 20 percent of products account for 80 percent of revenues and even in that 20 percent only a small number of products are sold very frequently. This rule can be implemented using Zipf's law with an exponent value equal to 1. The graphical representation of the benchmark is shown in Figure 2. From the figure it can be observed that the frequency of selling a small number of products is significantly higher compared to the rest of the products. Therefore, in the stream that propagates toward the warehouse, most of the tuples need to join with a small number of records on disk again and again. Currently the HYBRIDJOIN algorithm does not consider this feature and loads the pages from the disk frequently. Consider the reduction in I/O costs, if these pages can be held permanently in memory.

4 Semi-Stream Cache Join (SSCJ)

In this paper, we propose a new algorithm, SSCJ, which overcomes the issues stated above. This section presents a detailed description of SSCJ and its cost model.

4.1 Execution Architecture

The SSCJ algorithm possesses two complementary hash join phases, somewhat similar to Symmetric Hash Join. One phase uses R as the probe input; the largest part of R will be stored in tertiary memory. We call it the disk-probing

phase. The other join phase uses the stream as the probe input, but will deal only with a small part of relation R. We call it the stream-probing phase. For each incoming stream tuple, SSCJ first uses the stream-probing phase to find a match for frequent requests quickly, and if no match is found, the stream tuple is forwarded to the disk-probing phase.

The execution architecture for SSCJ is shown in Figure 3. The largest components of SSCJ with respect to memory size are two hash tables, one storing stream tuples denoted by H_S and the other storing tuples from the disk-based relation denoted by H_R. The other main components of SSCJ are a disk buffer, a queue, a frequency recorder, and a stream buffer. Relation R and stream S are the external input sources. Hash table H_R contains the most frequently accessed tuples of R and is stored permanently in memory. SSCJ alternates between stream-probing and disk-probing phases. According to the procedure described above, the hash table H_S is used to store only that part of the stream which does not match tuples in H_R. A stream-probing phase ends if H_S is completely filled or if the stream buffer is empty. Then the disk-probing phase becomes active. The length of the disk-probing phase is determined by the fact that a few disk partitions (or disk blocks) of R have to be loaded at a time in order to amortize the costly disk access. In the disk-probing phase of SSCJ, the oldest tuple in the queue is used to determine the partition of R that is loaded for a single probe step. In this way, in SSCJ it is guaranteed that every stream tuple loaded in memory will be processed in a certain time period. This is the step where SSCJ needs an index on table R in order to find the partition in R that matches the oldest stream tuple. However, a non-clustered index is sufficient, if we consider equijoins on a foreign key element that is stored in the stream. After one disk-probing phase, a number of stream tuples are deleted from H_S, so the algorithm switches back to the stream-probing phase. One phase of stream-probing with a subsequent phase of disk-probing constitutes one outer iteration of SSCJ.

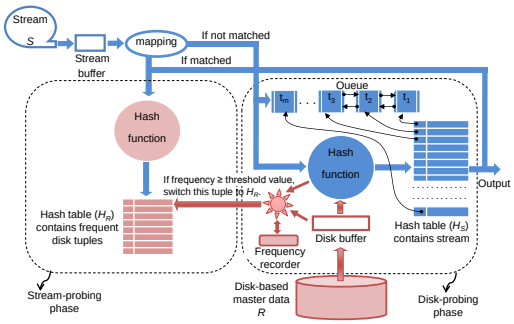


Fig. 3. Execution architecture of SSCJ

The stream-probing phase is used to boost the performance of the algorithm by quickly matching the most frequent master data. For determining very frequent tuples in R and loading them into H_R, the frequency detection process is required. This process tests whether the matching frequency of the current tuple is larger than a pre-set threshold. If it is, then this tuple is entered into H_R. If there are no empty slots in H_R the algorithm overwrites an existing least frequent tuple in H_R. This least frequent tuple is determined by the component frequency recorder. The question of where to set the threshold arises, i.e. how frequently must a stream tuple be used in order to get into this phase, so that the memory sac-

Table 1. Notations used in cost estimation of SSCJ

Parameter name	Symbol
Number of stream tuples processed in each iteration through H_R	w_N
Number of stream tuples processed in each iteration through H_S	w_S
Disk tuple size *(bytes)*	v_R
Disk buffer size *(tuples)*	d
Size of H_R *(tuples)*	h_R
Size of H_S *(tuples)*	h_S
Memory weight for the hash table	α
Memory weight for the queue	$1 - \alpha$
Cost to look-up one tuple in the hash table *(nano secs)*	c_H
Cost to generate the output for one tuple *(nano secs)*	c_O
Cost to remove one tuple from the hash table and the queue *(nano secs)*	c_E
Cost to read one stream tuple into the stream buffer *(nano secs)*	c_S
Cost to append one tuple in the hash table and the queue *(nano secs)*	c_A
Cost to compare the frequency of one disk tuple with the specified threshold value *(nano secs)*	c_F
Total cost for one loop iteration *(secs)*	c_{loop}

rificed for this phase really delivers a performance advantage. The threshold is a flexible barrier. Initially, an appropriate value is assigned to it while later on this value can be varied up and down depending on available size of H_R and the rate of matching the disk tuples in the disk buffer. The disk buffer stores the swappable part of R and for each iteration it loads a particular partition of R into memory. The other component queue is used to store the values for the join attribute. The main purpose of the queue is to keep the record of each stream tuple in memory with respect to time. The stream buffer is included in the diagram for completeness, but is in reality always a tiny component and it will not be considered in the cost model.

4.2 Cost Model

In this section we develop the cost model for our proposed SSCJ. The cost model presented here follows the style used for HYBRIDJOIN and MESHJOIN. Equation 1 represents the total memory used by the algorithm (except the stream buffer), and Equation 2 describes the processing cost for each iteration of the algorithm. The notations we used in our cost model are given in Table 1.

Memory cost: The major portion of the total memory is assigned to the hash table H_S together with the queue while a comparatively much smaller portion is assigned to H_R, the frequency detector, and the disk buffer. The memory for each component can be calculated as follows:
Memory for the disk buffer (bytes)= $d \cdot v_R$
Memory for H_R (bytes)=$h_R \cdot v_R$
Memory for frequency recorder (bytes)=$8h_R$
Memory for H_S (bytes)=$\alpha(M - d \cdot v_R - h_R \cdot v_R - 8h_R)$
Memory for the queue (bytes) = $(1 - \alpha)(M - d \cdot v_R - h_R \cdot v_R - 8h_R)$

By aggregating the above, the total memory M for SSCJ can be calculated as shown in Equation 1.

$$M = d{\cdot}v_R{+}h_R{\cdot}v_R{+}8h_R{+}\alpha(M{-}d{\cdot}v_R{-}h_R{\cdot}v_R{-}8h_R){+}(1{-}\alpha)(M{-}d{\cdot}v_R{-}h_R{\cdot}v_R{-}8h_R) \quad (1)$$

Currently, the memory for the stream buffer in not included because it is small (0.05 MB is sufficient in our experiments).

Processing cost: In this section we calculate the processing cost for the algorithm. To make it simple we first calculate the processing cost for individual components and then sum these costs to calculate the total processing cost for one iteration.
Cost to load d tuples from disk to the disk buffer (nanosecs)=$c_{I/O}(d)$
Cost to look-up w_N tuples in H_R (nanosecs)=$w_N \cdot c_H$
Cost to look-up disk buffer tuples in H_S (nanosecs)=$d \cdot c_H$
Cost to compare the frequency of all the tuples in disk buffer with the threshold value (nanosecs)=$d \cdot c_F$
Cost to generate the output for w_N tuples (nanosecs)=$w_N \cdot c_O$
Cost to generate the output for w_S tuples (nanosecs)=$w_S \cdot c_O$
Cost to read the w_N tuples from the stream buffer (nanosecs)=$w_N \cdot c_S$
Cost to read the w_S tuples from the stream buffer (nanosecs)=$w_S \cdot c_S$
Cost to append w_S tuples into H_S and the queue (nanosecs)=$w_S \cdot c_A$
Cost to delete w_S tuples from H_S and the queue (nanosecs)=$w_S \cdot c_E$
By aggregating the above costs the total cost of the algorithm for one iteration can be calculated using Equation 2.

$$c_{loop}(secs) = 10^{-9}[c_{I/O}(d){+}d(c_H{+}c_F){+}w_S(c_O{+}c_E{+}c_S{+}c_A){+}w_N(c_H{+}c_O{+}c_S)] \quad (2)$$

Since in c_{loop} seconds the algorithm processes w_N and w_S tuples of the stream S, the service rate μ can be calculated using Equation 3.

$$\mu = \frac{w_N + w_S}{c_{loop}} \quad (3)$$

In fact, based on the cost model we tuned SSCJ to a provably optimal distribution of memory between the two phases, and the components within the phases[1].

5 Performance Experiments

5.1 Experimental Setup

Hardware specification: We performed our experiments on a *Pentium-core-i5* with 8GB main memory and 500GB hard drive as secondary storage. We implemented our experiments in `Java` using the `Eclipse IDE`. The relation R is stored on disk using a `MySQL` database.

[1] Due to the page limit we are unable to include the tuning of SSCJ in the paper.

Table 2. Data specification

Parameter	**value**
Size of disk-based relation R	100 million tuples (≈11.18GB)
Total allocated memory M	1% of R (≈0.11GB) to 10% of R (≈1.12GB)
Size of each disk tuple	120 *bytes* (similar to HYBRIDJOIN)
Size of each stream tuple	20 *bytes* (similar to HYBRIDJOIN)
Size of each node in the queue	12 *bytes* (similar to HYBRIDJOIN)

Measurement strategy: The performance or service rate of the join is measured by calculating the number of tuples processed in a unit second. In each experiment both algorithms first complete their warm-up phase before starting the actual measurements. These kinds of algorithms normally need a warm-up phase to tune their components with respect to the available memory resources so that each component can deliver maximum performance. In our experiments, for each measurement we calculate the confidence interval by considering 95% accuracy, but sometimes the variation is very small.

Synthetic data: The stream dataset we used is based on the Zipfian distribution. We test the performance of all the algorithms by varying the skew value from 0 (fully uniform) to 1 (highly skewed). The detailed specifications of our synthetic dataset are shown in Table 2.

TPC-H: We also analyze the performance of all the algorithms using the TPC-H dataset which is a well-known decision support benchmark. We create the datasets using a scale factor of 100. More precisely, we use table `Customer` as our master data table and table `Order` as our stream data table. In table `Order` there is one foreign key attribute `custkey` which is a primary key in `Customer` table. So the two tables are joined using attribute `custkey`. Our `Customer` table contains 20 million tuples while the size of each tuple is 223 bytes. On the other hand `Order` table also contains the same number of tuples with each tuple of 138 bytes.

Real-life data: Finally, we also compare the performance of all the algorithms using a real-life dataset[2]. This dataset basically contains cloud information stored in summarized weather reports format. The same dataset was also used with the original MESHJOIN. 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 streaming data table is 128 bytes. Both the tables are joined using a common attribute, `longitude (LON)`, and the domain for the join attribute is the interval [0,36000].

5.2 Performance Evaluation

In this section we present a series of experimental comparisons between SSCJ and HYBRIDJOIN using synthetic, TPC-H, and real-life data. In order to understand the difference between the algorithms better, we include two other

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

algorithms. First we include MESHJOIN, which is a seminal algorithm in the field that serves as a benchmark for semi-stream joins. Then we include R-MESHJOIN, which is a slight modification of MESHJOIN. It introduces an additional degree of freedom for the optimization of MESHJOIN.

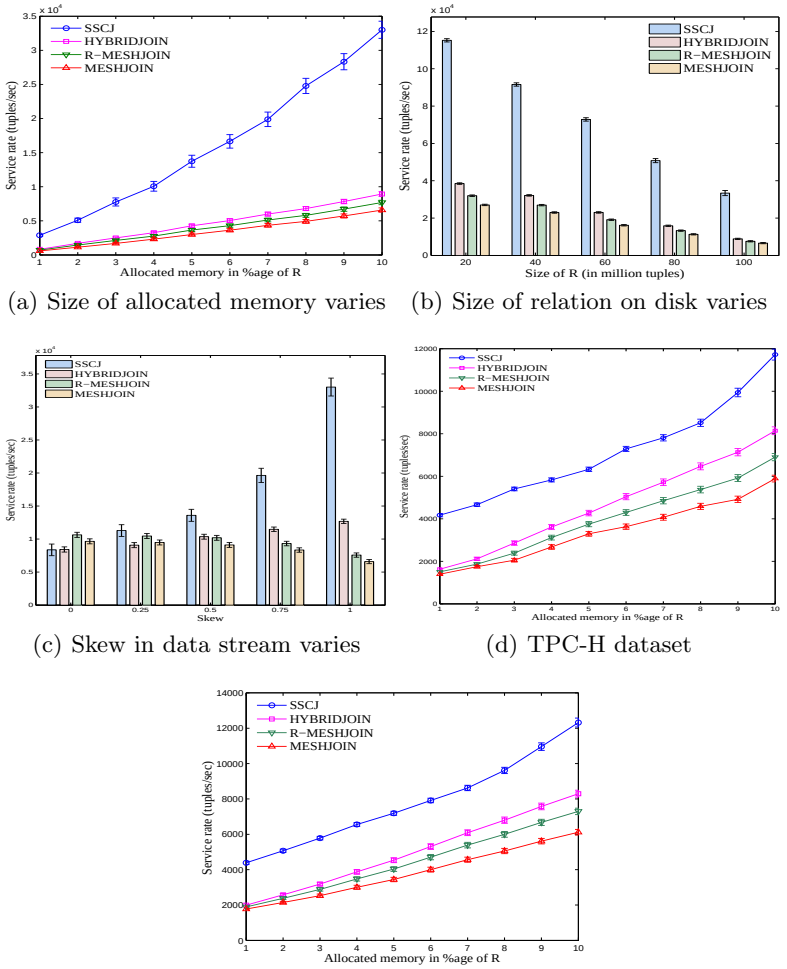
In our experiments we perform three different analyses. In the first analysis, we compare service rate, produced by each algorithm, with respect to the externally given parameters. In the second analysis, we present time comparisons, both processing and waiting time, for all four approaches. Finally, in our last analysis we validate our cost models for each of the algorithm.

External parameters: We identify 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 skew 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.

Analysis by varying size of memory M: In our first experiment we compare the service rate produced by all four algorithms by varying the memory size M from 1% to 10% of R while the size of R is 100 million tuples (≈11.18GB). We also fix the skew value equal to 1 for all settings of M. The results of our experiment are presented in Figure 4(a). From the figure it can be noted that SSCJ performs up to 4.5 times faster than HYBRIDJOIN in the case of a 10% memory setting. While in the case of a limited memory environment (1% of R) SSCJ still performs up to 3 times better than HYBRIDJOIN making it an adaptive solution for memory constraint applications. SSCJ also performs significantly better than both R-MESHJOIN and MESHJOIN.

Analysis by varying size of R: In this experiment we compare the service rate of SSCJ with the other three algorithms at different sizes of R under fixed memory size, ≈1.12GB. We also fix the skew value equal to 1 for all settings of R. The results of our experiment are shown in Figure 4(b). From the figure it can be seen that SSCJ performs up to 3 times better than HYBRIDJOIN under all settings of R. On the other hand if we compare the performance of SSCJ with MESHJOIN and R-MESHJOIN, it also performs significantly better than both of the algorithms under all settings of R.

Analysis by varying skew value: In this experiment we compare the service rate of all the algorithms by varying the skew value in the streaming data. To vary the skew, we vary the value of the Zipfian exponent. In our experiments we allow it to range from 0 to 1. At 0 the input stream S is completely uniform while at 1 the stream has a larger skew. We consider the sizes of two other parameters, memory and R, to be fixed. The size of R is 100 million tuples (≈11.18GB) while the available memory is set to 10% of R (≈1.12GB). The results presented in Figure 4(c) show that SSCJ again performs significantly better among all approaches even for only moderately skewed data. Also this improvement becomes more pronounced for increasing skew values in the streaming data. At skew value equal to 1, SSCJ performs about 3 times better than HYBRIDJOIN. Contrarily, as MESHJOIN and R-MESHJOIN do not exploit the data skew in their algorithms, their service rates actually decrease slightly for more skewed



(a) Size of allocated memory varies (b) Size of relation on disk varies

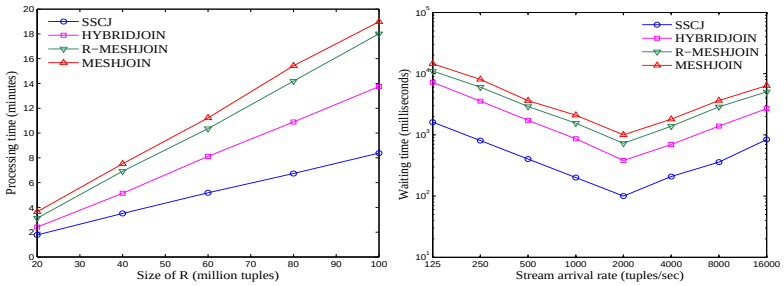
(c) Skew in data stream varies (d) TPC-H dataset

(e) Real-life dataset

Fig. 4. Performance analysis

data, which is consistent to the original algorithms findings. We do not present data for skew value larger than 1, which would imply short tails. However, we predict that for such short tails the trend continues. SSCJ performs slightly worse than MESHJOIN and R-MESHJOIN only in a case when the stream data is completely uniform. In this particular case the stream-probing phase does not contribute considerably while on the other hand random access of R influences the seek time.

TPC-H and real-life datasets: We also compare the service rate of all the algorithms using TPC-H and real-life datasets. The details of both datasets have already been described in Section 5.1. In both experiments we measure



(a) Processing time (b) Waiting time

Fig. 5. Time analysis

the service rate produced by all four algorithms at different memory settings. The results of our experiments using TPC-H and real-life datasets are shown in Figures 4(d) and 4(e) respectively. From both figures it can be noted that the service rate in case of SSCJ is remarkably better among all three approaches.

Time analysis: A second kind of performance parameter besides service rate refers to the time an algorithm takes to process a tuple. In this section, we analyze both waiting time and processing time. *Processing time* is an average time that every stream tuple spends in the join window from loading to matching without including any delay due to a low arrival rate of the stream. *Waiting time* is the time that every stream tuple spends in the stream buffer before entering into the join module. The waiting times were measured at different stream arrival rates. The experiment, shown in Figure 5(a), presents the comparisons with respect to the processing time. From the figure it is clear that the processing time in case of SSCJ is significantly smaller than the other three algorithms. This difference becomes even more pronounced as we increase the size of R. The plausible reason for this is that in SSCJ a big part of stream data is directly processed through the stream-probing phase without joining it with the whole relation R in memory.

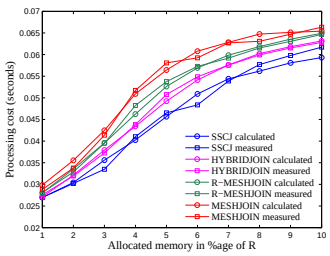


Fig. 6. Cost validation

In the experiment shown in Figure 5(b) we compare the waiting time for each of the algorithms. It is obvious from the figure that the waiting time in the case of SSCJ is significantly smaller than the other three algorithms. The reason behind this is that in SSCJ since there is no constraint to match each stream tuple with the whole of R, each disk invocation is not synchronized with the stream input.

Cost analysis: The cost models for all the algorithms have been validated by comparing the calculated cost with the measured cost. Figure 6 presents the comparisons of both costs for each algorithm. The results presented in the figure show that for each algorithm the calculated cost closely resembles the measured cost, which proves the correctness of our cost models.

6 Conclusions

In this paper we propose a new semi-stream-based join called SSCJ and we compare it with HYBRIDJOIN and other earlier well-known semi-stream join algorithms. SSCJ is designed to make use of skewed, non-uniformly distributed data as found in real-world applications. In particular we consider a Zipfian distribution of foreign keys in the stream data. Contrary to HYBRIDJOIN, SSCJ stores these most frequently accessed tuples of R permanently in memory saving a significant disk I/O cost and accelerating the performance of the algorithm. We have derived a cost model for the new algorithm and validated it with experiments. We have provided an extensive experimental study showing an improvement of SSCJ over the earlier HYBRIDJOIN and other related algorithms.

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 *IEEE 27th International Conference on Data Engineering (ICDE'11)*, pages 159 –170, April 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: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.
4. Alexandros Karakasidis, Panos Vassiliadis, and Evaggelia 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, New York, NY, USA, 2005. ACM.
5. Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
6. M. Asif Naeem, Gillian Dobbie, and Gerald 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.
7. 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):21–42, 2011.
8. M. Asif Naeem, Gillian Dobbie, Gerald Weber, and Shafiq 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.
9. N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.E. 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.
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.