A Multi-Way Semi-Stream Join for a Near-Real-Time Data Warehouse

M. Asif Naeem[1], Kim Tung Nguyen[1], and Gerald Weber[2]

[1] School of Engineering, Computing and Mathematical Sciences, Auckland University of Technology
mnaeem@aut.ac.nz, jvc0109@autuni.ac.nz
[2] Department of Computer Science, The University of Auckland
gerald@cs.auckland.ac.nz

Abstract. Semi-stream processing, the operation of joining a stream of data with non-stream disk-based master data, is a crucial component of near real-time data warehousing. The requirements for semi-stream joins are fast, accurate processing and the ability to function well with limited memory. Currently, semi-stream algorithms presented in the literature such as MeshJoin, Semi-Stream Index Join and CacheJoin can join only one foreign key in the stream data with one table in the master data. However, it is quite likely that stream data have multiple foreign keys that need to join with multiple tables in the master data. We extend CacheJoin to form three new possibilities for multi-way semi-stream joins, namely Sequential, Semi-concurrent, and Concurrent joins. Initially, the new algorithms can join two foreign keys in the stream data with two tables in the master data. However, these algorithms can be easily generalized to join with any number of tables in the master data. We evaluated the performance of all three algorithms, and our results show that the semi-concurrent architecture performs best under the same scenario.

Keywords: Multi-Way Stream Processing, Join Operator, Near-Real-Time Data Warehouse

1 Introduction

Near-real-time data warehousing (RDW), with its ability to process and analyze data nearly instantly, is increasingly adopted by the business world. Among several approaches to RDW, Data Stream Processing is a crucial component, handling the continuous incoming information - a stream of data, from multiple sources [1]. One method of stream processing is a join operation which combines the streaming data with the slowly changing disk-based master data (denoted as R) [1, 2]. As the join deals with two sources, one being a stream, and the other being fairly stable data stored in a disk, such as master data, the join is considered "semi-stream".

With the rapid development of new technologies, the large capacity of current main memories as well as the availability of powerful cloud computing platforms can be utilized to execute stream-based operations [3]. However, to enable

the efficient use of ICT infrastructure, semi-stream joins that can process the streaming data in a near-real-time manner while requiring minimum resource consumption, are still of interest. Several semi-stream joining methods have been proposed so far. The authors of the MeshJoin algorithm [8] argued for the need to support streaming updates in RDW [4]. Since then, many other join operators have been developed by improving or adding more features to MeshJoin, such as R-MeshJoin [5], Partition-based Join [6], HybridJoin [7], Semi-Stream Index Join (SSIJ) [2] and CacheJoin [3], to name a few. The authors of the MeshJoin operator suggest that one of the most important research topics in the field that need to be examined next is multi-way semi-stream joins between a stream (whose tuples have two or more foreign keys) and many relations [8]. Indeed, it would be quite practical to process stream data with multiple foreign keys to join with multiple tables in R.

In this paper, we address the problem by developing a multi-way semi-stream join. We propose three different approaches to the joins namely Sequential, Semi-concurrent and Concurrent. The joins are developed by extending CacheJoin (CJ), one of the most advanced semi-stream joins proposed in the field [3]. The advantage of CJ is that it requires very little in the way of computing resources while its service rate is higher than other joins such as MeshJoin, R-MeshJoin and HybridJoin [3]. As extended versions of CJ, the new multi-way joins inherit the main characteristics of their precursor. For example, as CJ performs well with skewed, non-uniformly distributed data, such as the Zipfian distribution of foreign keys in the stream data [3], the newly developed multi-way joins are expected to have the same characteristics.

In this paper, we first develop new multi-way joins which can match a stream data having two foreign keys with two tables in R. The joins then can be generalized to join more tables. To test the new algorithms, we apply them to a scenario where a stream tuple includes customer and product foreign keys which need to join with customer and product tables in R. In the Sequential approach, there are two CJs running concurrently where the first CJ joins customer keys and produces output as the input for the second CJ. After this, the second CJ processes the product foreign key and produces output for the whole multi-way join. In Semi-concurrent, only part of the stream tuples are processed in sequence, and the rest are processed concurrently by two separate CJs. In Concurrent, there are also two CJs running concurrently, but they match the two foreign keys of a tuple at the same time, and the tuple will be sent to output only when both keys are matched. After testing the new joins with different datasets, results show that Semi-concurrent performs best under the same memory setting.

The rest of this paper is organized as follows. Section 2 presents a review of the available semi-stream joins in the academic literature, which focuses on the architecture of the CJ algorithm. This is expected to provide the background theory required to comprehend the new multi-way join algorithms. Section 3 describes the architectures of the Sequential, Semi-Concurrent and Concurrent joins in detail. In Section 4 we present a cost model to measure the performance of the new joins. Section 5 presents the performance evaluation and, from the

experimental data, it is concluded that the Semi-concurrent performs best while Concurrent performs worst among the three. In Section 6 we explain our explanation for this order. Finally Section 7 concludes the paper.

2 Related Work

This section presents an overview of some of the semi-stream joins available and then examines in detail the architecture and characteristics of CJ, which is the antecedent of our multi-way semi-stream joins.

In the past, the algorithm MeshJoin was proposed for joining a data stream with a slowly changing table under limited main memory conditions [8, 9]. The two fundamental features of MeshJoin are: (1) accessing the disk-based R with fast sequential scans, and (2) armotizing the cost of I/O operations over a large number of stream tuples.The features, therefore, can help MeshJoin reduce costly disk access. Other advantages of MeshJoin are: (1) it can work well with limited main memory and, (2) the organization of R has hardly any effect on its performance. However, the join operation has some limitations. The first limitation is caused by the fact that MeshJoin does not consider the distribution of the incoming stream data as well as the organization of R. Therefore its performance on skewed data is inferior [10]. Also, the performance of MeshJoin is inversely proportional to the size of R. Thus this algorithm does not perform well with large Rs[3].

To improve the MeshJoin algorithm, R-MeshJoin (Reduced MeshJoin) was developed in 2010 [5].R-MeshJoin improves the MeshJoin operator by clearly defining the dependent relationships between its antecedent's components. Therefore, R-MeshJoin is simpler and obtains slightly better performance than MeshJoin.

We presented another improved version named HybridJoin in the past [7].The main goals of HybridJoin are: (1) to amortize the fast-coming data stream with slow disk access using limited computer memory, and (2), to deal with an input data stream sent in small and sporadic groups [10]. The main technique used by HybridJoin to amortize the fast-coming data stream is an index-based approach to access R, which is quite efficient. However, like MeshJoin, HybridJoin does not take data distribution of the streaming data into consideration.

CJ is an improved HybridJoin operator that inherits the advantages and solves the limitation of its former algorithms [3]. The architecture of CJ is presented in Figure 1. The main improvement of CJ is an additional hash table stored in computer memory, which stores the most frequent tuples coming from the stream (denoted as H_R). When tuples from the stream arrive, they enter the cache phase first where they are matched with H_R. In this way more frequent tuples can be processed faster as memory access is faster than disk access. If a tuple is not matched in the cache phase, it will be sent to disk phase which is basically a HybridJoin. In disk phase, stream tuples are stored in a hash table named H_S and their foreign keys are also added to a queue. To minimize expensive disk access, a few disk pages of R are loaded to a Disk-Buffer (DB) whenever the join conducts a database query. The oldest tuple in the queue is

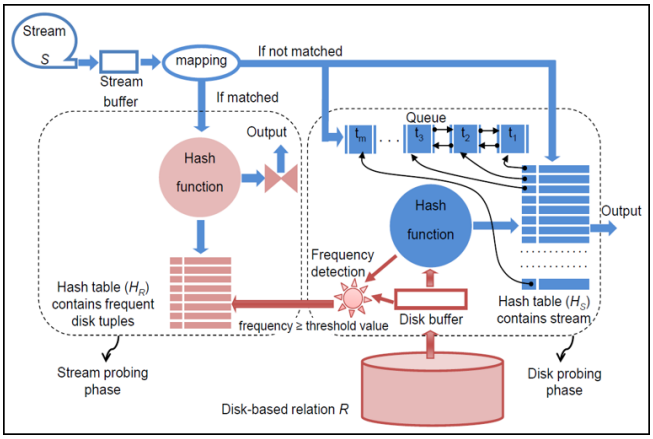

Fig. 1. CACHEJOIN Architecture

used to determine the partition of R which will be loaded in each probing iteration. More specifically, after probing the foreign key of a tuple into R, a few disk pages starting from the matching page in R will be put into DB. All these tuples will be matched with H_S, in order to amortize the seek and disk access time. Thus, the higher the number of tuples in H_S, the higher the probability that some tuples in H_S can be matched with DB, which leads to the faster service rate of CJ's Disk Phase. Another important component of CJ is the frequency detector whose algorithm is as follows: In each matching iteration between DB and H_S, rows that have the number of matches above a certain threshold will be considered as frequent tuples and added to H_R.

A comparison between CJ and MeshJoin shows that CJ outperforms MeshJoin in many cases, such as with different settings of R, under different memory conditions and when stream data is skewed [3]. The only situation where CJ processes slower than MeshJoin is when the distribution of stream data is completely uniform, which hardly ever happens in practice. Of the algorithms described above, CJ is the only one which considers the distribution of the stream data, while still including the positive features of the others.

All of the above algorithms can join only one-foreign-key stream data with a single table in the master data, but in business there is a need of joining multiple foreign keys with multiple tables in R. The review of current literature shows that not much research has been carried out in this direction.

3 Multi-way Semi-Stream Joins

In this paper, we developed three different multi-way semi-stream joins extended from CJ and named them Sequential, Semi-concurrent and Concurrent. As mentioned above, in our experiment presented here, the joins were applied to match two foreign keys of stream tuples with two tables in R. As there are two keys

that need to be joined with two tables, our approach is to process each key using a CJ. Thus, we need to organize the process of the two CJs in a suitable order to optimize the multi-way joins' performance in regard to both service rate and resource consumption. The first decision was whether we should create two CJ threads executing the two keys concurrently, or only one thread which processes one key at a time. The two-thread approach was our preference for the following reasons:

- Both approaches require the same level of memory: as both of them contain two CJs, they have similar objects.
- The two-thread approach is feasible. Although running two threads concurrently means doubling the CPU calculation, this approach is still feasible as CJ consumes few resources [3].
- Utilizing multiple threads may improve the applications' performance [11].

Another advantage of the two-thread approach is that it reduces the idle time of the join operator. In CJ, after sending a SQL query to a Database Management System (DBMS) such as MySQL, the join is idle as it waits for the DBMS to execute the query and return the results. Similarly, the DBMS sleeps when the CJ is processing the data returned from the queries. By running two CJs concurrently, the idle time of the both systems (CJs and DBMS) will be reduced as one thread may be working while the other is idling. Therefore, we expect that the time required to process two keys will be less than double the time required to match only one key of the stream tuples.

3.1 Sequential Multi-Way Semi-Stream Join

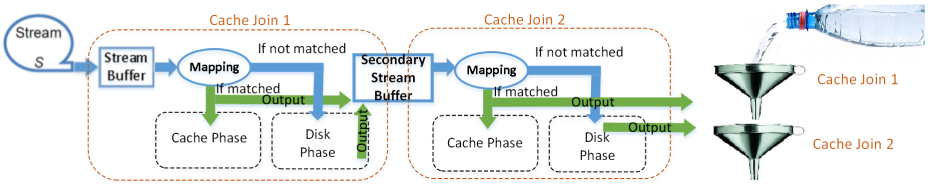

Fig. 2. Simplified Architecture Architecture of the Sequential Approach

Figure 2 presents the simplified architecture of the sequential join, which abstracts CJ to the cache and disk phase level (The cache phase and disk phase boxes are referred to in more detail in Figure 1). Basically, the sequential join contains two CJs running in sequence, i.e. a tuple is firstly joined with the Customer table by the Customer CJ. Then the matched tuple taken from the Customer table is attached to the stream tuple to form the input for the second CJ, which is the secondary stream buffer (SB). The second CJ takes tuples from SB and processes the other key of the tuples (Product key) and adds the

probed product tuple to the stream tuple to form the final join's output. It is worth noting that, although the two keys of a stream tuple are processed in sequence, the two CJs are running concurrently. In Figure 2, we use a visual metaphor where the water is the stream of data, and the two funnels depict the two CJs running concurrently while the tuples are processed in sequence. With this architecture, we may expect that, although Sequential matches two keys of a tuple, its service rate is equal to the service rate of the slower of the two CJs.

3.2 The Semi-concurrent Approach

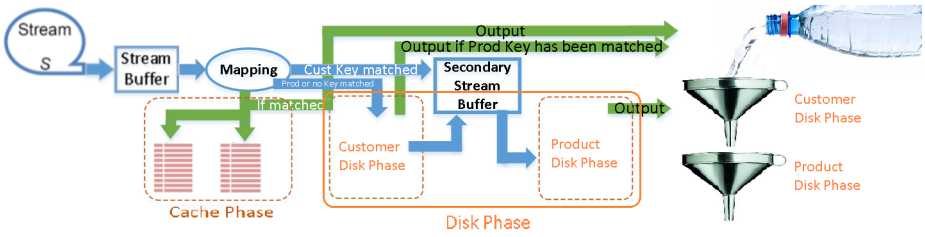

Fig. 3. Simplified Architecture Architecture of the Semi-concurrent Approach

Figure 3 presents the simplified architecture of the semi-concurrent algorithm. Similar to CJ, the semi-concurrent join has a cache phase and a disk phase. When a tuple first enters Semi-concurrent, both of its keys are matched with two hash tables H_{R_C} and H_{R_P}, which retain the most frequent tuples of the customer and product tables in cache respectively. If both keys of a stream tuple are matched, the tuple will be ready for output. In all other cases, the tuple will be sent to the disk phase. Semi-concurrent's disk phase has two CJ disk phases running concurrently, processing tuples in sequence, which is quite similar to the sequential process. In Figure 3, we use the same visual metaphor as the sequential join, but the two funnels are only disk phases instead of complete CJs. If only one key of a tuple is matched in the cache phase, the tuple will be sent to the relevant CJ disk phase to be joined with the other key, e.g. if the product key of a tuple is matched in cache, the tuple will be sent to the customer disk phase. After the second key is processed, the disk phases will produce the final output for the join. With this architecture, only tuples having both keys unmatched within cache go through both customer and product disk phases.

3.3 The Concurrent Approach

Figure 4 presents the simplified architecture of the concurrent join. The cache phase of the concurrent system is very similar to that of the semi-concurrent system, while its disk phase has a new processing method. The concurrent system stores stream, customer and product tuples in its queue, which makes the

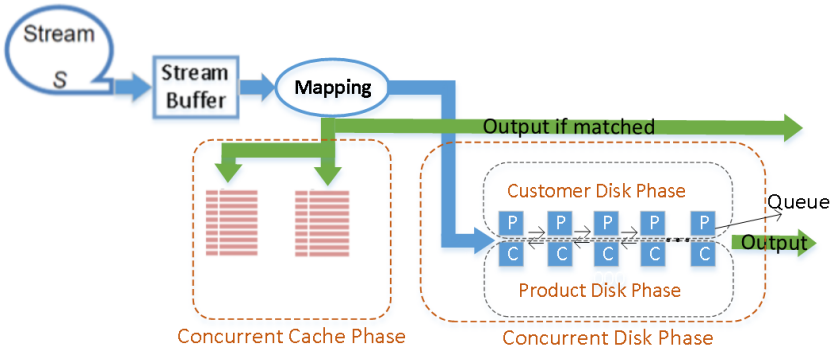

Fig. 4. Abstract Level Architecture of the Concurrent Approach

queue the largest component of the join with regard to memory consumption. At each queue node, its customer/product tuple will be set to null if its customer/product key has not been matched, otherwise the customer/product tuple will store the matched item. During the disk phase, there are two CJ disk phase threads simultaneously executing the unmatched customer and product keys in the queue, which are called customer and product disk phases. The disk phases are supported by two hash tables H_{S_C} and H_{S_P}, whose key/value pairs are an unmatched key and its associated queue node. In this architecture, a queue node will be sent to output only when both customer and product tuples are not null.

For example, if the customer key of a stream tuple is matched in the cache phase, this partially matched stream tuple will be added to a queue where the product key will be matched. At the same time, as only its product key has not been found in cache, the product key is put to H_{S_P}. In another instance, if neither key of a tuple is matched with the cache, the tuple will be added to a queue node where both customer and product items are null, and the customer and product keys are put to H_{S_C} and H_{S_P} respectively.

As opposed to its predecessor CJ, the number of queue nodes in the concurrent system is not equal to the numbers of tuples in H_{S_C} and H_{S_P}. Rather, the numbers of unmatched customer and product keys in the queue are equal to the sizes of H_{S_C} and H_{S_P}, respectively.

4 Cost Model

To evaluate the new multi-way joins, we developed a cost model to measure critical factors of their performance. In the case of CJ, the factors are classified into two main groups being memory cost and processing cost. As these multi-way joins are developed from CJ, we have adopted the notations used in the cost estimation of CJ to the new joins. Unfortunately, the processing cost in the cost model is not meaningful when applied to multi-way joins. For example, for

each CJ run, processing costs such as; costs to conduct a database query and read disk pages to the DB, cost to look up one tuple in the hash table H_R can be recorded and added together to get the total processing cost. However, we cannot simply sum the processing costs of the two CJs to calculate the cost of the whole multi-way join as the CJ threads run concurrently and the costs overlap. Furthermore, as the two CJ threads execute independently, the multi-way joins do not have a common iteration. Thus multi-way joins do not have a total cost for one loop iteration as in CJ. To this end we have chosen one processing cost factor for evaluating multi-way joins, which is service rate. The service rates of the new joins are calculated as follows:

$$SR = \frac{total\_processing\_time}{total\_number\_of\_tuples\_processed} \tag{1}$$

In regard to memory cost, we used the total runtime memory required by the Java programs to operate the multi-way joins in order to compare their performance. The runtime memory of a Java program includes both used and free memory, which are the memory allocated for currently used objects and possible new objects respectively [12]. In this way runtime memory may best reflect the memory cost of each semi-stream join. In our research, we use the memory cost objects adopted from CJ to calculate the memory required by all objects of the joins, but it is only an estimation because sizes of some objects change overtime. For example, the size of Concurrent's queue depends on the number of matched tuples in its nodes, but the number changes overtime. Another example is the secondary SB of Sequential and Concurrent, whose memory size is also not stable. By having the estimations, we adjust the setting of each multi-way join, so that the three joins have the same memory setting.

5 Evaluation

5.1 Experimental Setup

Testing Environment We ran our experiments on an Core i3-2310 CPU@ 2.10GHz with Solid State Drive (SSD). We implemented our experiments in Java, using *Eclipse Java Neon 4.6.3*. Measurements were taken with Apache plug ins and *nanoTime()* from *Java API*. The R is stored on a disk using a $MySQL$ database, the fetch size for the result set was set to be equal to the disk buffer size. Synthetic data, the stream data, was generated with a Zipfian distribution of the foreign key. The detailed specifications of the data set used for analysis are shown in Table 1.

Memory Setting In the concurrent join, the largest component in terms of memory use is the queue. Indeed, each node of the queue stores the stream, product and customer objects, where customer and product objects are null if the objects have not been matched. To avoid memory consumption of the join becoming too high, there is a fixed maximum number of queue nodes. The

Table 1. Data Specifications

Object	**Value**
Stream tuple size	*20 bytes*
Size of customer disk tuple	*120 bytes*
Size of product disk tuple	*120 bytes*
Data set	based on Zipf's law (exponent is set to 1) Case 1: Both customer and product tables have 1 million tuples Case 2: Customer table: 1 million tuples, product table: 300,000 tuples

memory size of the queue, therefore, will reach its maximum when all nodes are half-matched (either the customer or the product object is matched). We used N_{CQ} to denote the number of nodes in the concurrent queue.

In the Sequential and Semi-concurrent joins, the largest components in term of memory use are their two hash tables H_{S_C} and H_{S_P} and these hash tables' sizes also need to be fixed. In both joins, we set the same size for both Customer and Product Hash tables, and used N_{SQ} and N_{SCQ} to denote the size of the sequential and semi-sequential hash tables respectively.

To test the performance of each join, we attempted to allocate the same amount of memory for each multi-way join. For our test dataset, the size of each customer and product object are the same (120 bytes), and the size of a stream object is 20 bytes. With this setting, to allocate the same amount of memory to all the joins, N_{SQ} and N_{SCQ} are set to equal to around 2/3 of N_{CQ}.

5.2 Comparison of the three Multi-Way Joins

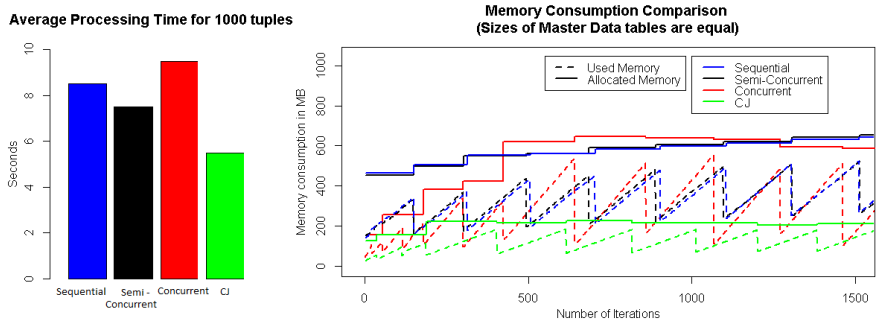

Fig. 5. Comparison of the three multi-way joins with the original CJ (Customer table: 1 million tuples, Product table: 1 million tuples)

Figures 5 and 6 show comparisons between the three approaches and a single CJ in two different cases as stated in Table 1. It must be remembered that, while the multi-way joins match two keys of a stream tuple with two tables in R, CJ joins only one. It can be observed that the time required to join two keys in the newly developed multi-way joins is less than double the time of a single CJ to process one key. In regard to the memory cost, these three new joins consume a similar level of memory, around 600MB and is three times more than CJ.

In both cases, the semi-concurrent join is the best performer, and Concurrent is the slowest multi-way join. The average time Semi-concurrent requires to process 1000 tuples in Case 1 is 7.5 seconds, while the single CJ requires 5.5 seconds, and, in Case 2, the difference is only one second.

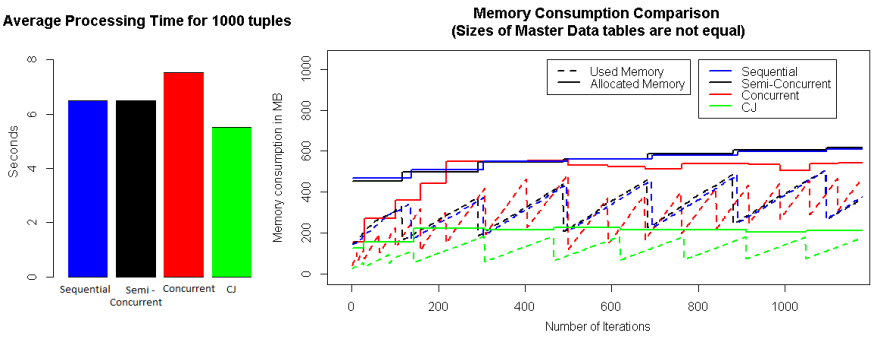

Fig. 6. Comparison of the three multi-way joins and the original CJ (Customer table: 1 million tuples, Product table: 300,000 tuples)

6 Discussion

The reason why Concurrent is slower than Semi-concurrent is as follows. Since both joins have the same cache phase, their disk phases cause the difference. As mentioned above, a node in concurrent process's queue will be moved to output only when both its customer and product keys are matched. While the concurrent process progresses, the number of half-matched nodes increases, which leads to the numbers of unmatched customer and product keys decreasing (because the total number of queue nodes is fixed to N_{CQ}). However, a characteristic of the CJ algorithm mentioned above is that the fewer unmatched items there are, the slower the join performs. In our experiment, after the join runs for a while, the number of unmatched customer and product keys is around 60% of N_{CQ}, which is smaller than in N_{SCQ} (which is equal to 2/3 of N_{CQ}). As a result, the concurrent system becomes slower than the semi-concurrent system because the number of unmatched keys in the semi-concurrent join is always fixed at N_{SCQ}. Figure 7 simulates the concurrent join's queue status while the join is in operation.

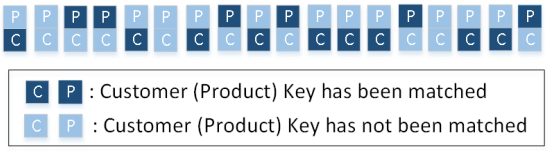

Fig. 7. Simulation of a concurrent queue at a given time

There are several reasons to explain why the sequential join's service rate is lower than that of the semi-concurrent join. First, in a case where both keys of a tuple are recognized as frequent keys, the semi-concurrent join will send the tuple direct to output after matching it with the cache phase. However, the sequential join requires more steps in processing the tuple as, after matching the first key, the sequential join puts the tuple into a secondary stream buffer, and the tuple must wait for the second CJ to be executed. Second, if the product key of a tuple is matched with the semi-concurrent join's cache, the tuple will go to the customer disk phase, and this phase may directly send the tuple to output. However, after processing this tuple's customer key, sequential join also needs to put it in the secondary stream buffer, and again the tuple must wait for the second CJ to be executed.

Although Sequential has some weaknesses when compared with Semi-concurrent, the two joins have quite similar architecture. Basically, the two joins have two CJ disk phase threads running concurrently and processing tuples in sequence, and this architecture has been proved to be more effective than the concurrent architecture. This provides an answer as to why the concurrent join performs the least well of the three.

Another advantage of the semi-concurrent architecture is that the join is quite flexible. Depending on the case we can adjust its components to achieve better performance. For example, if the size of the product disk tuple is smaller than the customer disk tuple, we can put the product disk phase first in the architecture to save memory. In the semi-concurrent join, after we match a stream tuple with the second CJ, the tuple will be sent to output. Therefore, we do not keep the disk tuple of the second CJ in memory. However, after matching a stream tuple with the first CJ, we need to put keep the matched tuples for the other key to be matched. Hence, by putting the disk tuple which has a smaller memory first in the processing order, the memory required to store the tuples will decrease.

The semi-concurrent join can also be generalized to match more keys by adding more H_R tables to its cache phase and more CJ disk phase threads to its disk phase. The main problem with generalization is that the more keys the join needs to match, the more memory the join requires. Even so, the multi-way join is still expected to be more efficient than other approaches.

7 Conclusion

In this paper, we proposed three different multi-way join architectures called Sequential, Semi-concurrent and Concurrent. Initially, we developed new joins to match two-foreign keys in stream data with two tables in the master data. We also developed a cost model to measure the joins' performance. We compared the performance of the all three newly developed joins with the original CJ. Our results show that Semi-concurrent performed best among the three approaches. In future we aim to generalize our multi-way semi-concurrent approach to join with n number of tables in the master data. Also we will optimize Semi-concurrent by making some adjustments on the algorithm such as the frequency detector and allocating different memories to different CJs in accordance with the distributions of each streaming tuple's foreign key.

References

1. Naeem, M.A, Jamil, N.: An Efficient Stream-based Join to Process End User Transactions in Real-Time Data Warehousing. Journal of Digital Information Management 12(3), 201-215, (2014)
2. Naeem, M.A., Dobbie, G., Lutteroth, C.: SSCJ: A Semi-Stream Cache Join using a Front-Stage Cache Module. In Bellatreche, L., Mohania, M., Data Warehousing and Knowledge Discovery. Berlin, Heidelberg: Springer 8057,236-247 (2013)
3. Naeem, M.A., Dobbie, G., Weber, G.: A lightweight stream-based join with limited resource consumption. In DaWaK '12: Data Warehousing and Knowledge Discovery, Berlin Heidelberg: Springer 431-442 (2012)
4. Bornea, M., Deligiannakis, A., Kotidis, Y., Vassalos, V.: Semi-Streamed Index Join for Near-Real Time Execution of ETL Transformations. IEEE 27th International Conference on. IEEE, 159-170 (2011)
5. Naeem, M.A., Dobbie, G., Weber, G., Alam, S.: R-MESHJOIN for near-real-time data warehousing, in: DOLAP'10: Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP, ACM, Toronto, Canada (2010)
6. Chakraborty, A., Singh, A.: 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, IEEE Computer Society, Washington, DC, USA, 1-11 (2009)
7. Naeem, M.A., Dobbie, G., Weber, G.: HybridJoin for near-real-time data warehousing, Int. J. Data Warehous. Min. (IJDWM) 7(4) (2011)
8. Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell, N.: Meshing Streaming Updates with Persistent Data in an Active Data Warehouse. IEEE Trans. on Knowl. and Data Eng., 20(7) 976-991 (2008).
9. Polyzotis, N., Skiadopoulos, S., Vassiliadis, P.: Supporting Streaming Updates in an Active Data Warehouse. ICDE 2007: Proceedings of the 23rd International Conference on Data Engineering,Istanbul 476-485 (2007)
10. Naeem, M., Dobbie, G., Lutteroth, C., Weber, G.: Skewed distributions in semi-stream joins: How much can caching help? Information Systems, 64, 63-74 (2017)
11. Oracle. Advantages and Disadvantages of a Multithreaded/Multicontexted Application. Retrieved from https://docs.oracle.com/cd/E13203_01/tuxedo/tux71/html/pgthr5.htm
12. Shirazi, J.: Java performance tuning. O'Reilly Media, Inc. (2003)