A Cached-based Approach to Enrich Stream Data with Master Data

M. Asif Naeem, Noreen Jamil[1, 2], Imran Sarwar Bajwa[3]
[1, 2] School of Computer and Mathematical Sciences
Auckland University of Technology
Private Bag 92006
Auckland, New Zealand
[3]Department of CS & IT
Islamia University of Bahawalpur
Pakistan
mnaeem@aut.ac.nz, imran.sarwar@iub.edu.pk, njam031@aucklanduni.ac.nz

ABSTRACT: *An enriching of stream data with disk-based master data is common in many applications. Real-time data warehousing is one of these applications where stream data is customers transactions producing by operational data source(s). This stream data needs to enrich by disk-based master data before loading this into the data warehouse. To implement this enrichment operation a join operator is required. Normally we call it semi-stream join as it is performed between stream data and disk data. The join operator typically works under limited main memory and this memory is generally not large enough to hold the whole disk-based master data. Therefore, the relation is loaded into memory in partitions. A well-known join algorithm called MESHJOIN (Mesh Join) has been presented in the literature to implement the semistream join operation. However, the algorithm suffers with some limitations. In particular, the algorithm can be improved based on the characteristics of stream data e.g. skew in stream data. In this paper we address this issue by presenting a novel algorithm called Cached-based Stream-Disk Join (CSDJ). The algorithm exploits skew characteristic in stream data more appropriately and over performs existing MESHJOIN. We present results for Zip-fian distributions of the type that appear in many applications. We evaluate our algorithm using synthetic, TPC-H and real datasets. Our experiments show that CSDJ performs significantly better than MESHJOIN.*

Keywords: Data Enrichment, Real-Time Data Warehousing, Semi-Stream Join, Performance Optimization

Received: 11 May 2015, Revised 18 June 2015, Accepted 24 June 2015

1. Introduction

Real-time data warehousing plays a prominent role in supporting overall business strategy. By extending data warehouses from static data re-positories to active data repositories enables business organizations to better inform their users and to take effective timely decisions [6], [10]. 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 units conversion, removal of duplicate tuples, information enrichment, filtering of unnecessary data, sorting of tuples, and translation of source data key. To implement these transformations a stream-based join is required.

In traditional data warehousing the source updates are buffered and join is performed off-line. On the other hand, in real-time data warehousing this operation needs to be performed as the updates are received from the data sources. In implementing the online execution of join, it is observed that due to different arrival rate of both inputs, the transactional or stream input is fast and huge in volume while the master or disk input is slow; the algorithm faces some performance issues due to a bottleneck in the stream of updates.

With the availability of large main memory and powerful cloud computing platforms, considerable computing resources can be utilized when executing stream-based joins. However, there are several scenarios where approaches that can function with limited main memory are of interest. First, the master data may simply be too large for the resources allocated for a stream join, so that a scalable algorithm is necessary. Second, low-resource consumption approaches may be necessary when mobile and embedded devices are involved. For example, stream joins such as the one discussed here could be used in sensor networks. As a consequence, semi-stream join algorithms that can function with limited main memory are important building blocks for a resource-aware system setup.

In the literature, a seminal semi-stream join algorithm MESHJOIN [8], [9] was proposed for joining a continuous stream data with a disk-based master data, such as 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. The algorithm performs a staggered execution of the hash table build in order to load in stream tuples more steadily. Although the MESHJOIN algorithm efficiently amortizes the disk I/O cost over fast input streams, the algorithm makes no assumptions about characteristics of stream data or the organization of the master data. Experiments by the MESHJOIN authors have shown that the algorithm performs worse with skewed data. There-fore, the question remains how much potential for improvement remains untapped due to the algorithm not being consider the characteristics of stream data.

In this paper we focus on one of the most common characteristics, a skewed distribution. Such distributions arise in practice, for example current economic models show that in many markets a select few products are bought with higher frequency [1]. Therefore, in the input stream, the sales transactions related to those products are the most frequent. In MESHJOIN, the algorithm does not consider the frequency of stream tuples.

We propose a robust algorithm called Cached-based Stream-Disk Join (CSDJ). The key feature of CSDJ is that the algorithm introduces a cache module that stores the most used portion of the disk-based relation, which matches the frequent items in the stream, in memory. As a result, this reduces the I/O cost substantially, which improves the performance of the algorithm. CSDJ performs slightly worse than MESHJOIN only in a case when the stream data is completely uniform. This is shown later in our experiments section.

In this work we only consider one-to-many equijoins, as they appear between foreign keys and the referenced primary key in another table. This is obviously a very important class of joins, and they are a natural case of a join between a stream of updates and master data in a data warehousing context [5], online auction systems [2] and supply-chain management [12]. Consequently, we do not consider joins on categorical attributes in master data, such as gender.

The rest of the paper is structured as follows. Section II presents related work. The existing MESHJOIN and problem statement are defined in Section III. Section IV describes the proposed CSDJ with its execution architecture, algorithm, cost model and tuning. Section V describes an experimental study of CSDJ. Finally, Section VI concludes the paper.

2. Related Work

In this section we will outline the well known work that has already been done in this area with a particular focus on those which are closely related to our problem domain. MESHJOIN (Mesh Join) [8], [9] 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) [7] clarifies the dependencies among the components of MESHJOIN. As a result, it improves the performance slightly. However, RMESHJOIN again does not consider the non-uniform characteristic of stream data.

One approach to improve MESHJOIN is a partitionbased join algorithm [4] that can also deal with stream intermittence. It uses a two-level hash table for attempting to join stream tuples as soon as they arrive, and uses a partitionbased waiting area for other stream tuples. For the algorithm in [4], 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) [3] joins stream data with disk-based data. SSIJ uses page level cache i.e. stores the entire disk pages in cache while it is possible that all the tuples in these pages may not be frequent in stream. As a result the algorithm can perform suboptimal. Also the algorithm does not include the mathematical cost model.

3. MESHJOIN and Problem Definition

In this section we summarize the MESHJOIN algorithm and at the end of the section we describe the observations that we focus on in this paper.

MESHJOIN (Mesh Join) [8], [9] was designed specifically for joining a continuous stream with a disk-based relation, i.e. 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.

In MESHJOIN, the whole relation *R* is traversed cyclically in an endless loop and every stream tuple is compared with every tuple in *R*. Therefore every stream tuple stays in memory for the time that is needed to run once through *R*. At a glance MESHJOIN has a staggered processing pattern, where stream tuples that arrive later start the comparison with *R* from a later point in R and wait until this point is reached again in the cyclic reading of *R*.

The usually large relation *R* has to be stored on disk, and is read into memory through a disk-buffer of size *b* pages. The relation R is naturally split into *k* equal parts, where each part is of size *b*. One traversal of *R* therefore happens in *k* steps, in each step a new part of *R* is loaded into the disk-buffer and replaces the old content.

The crux of MESHJOIN is that with every iteration a new chunk of stream tuples is read into main memory. Each of these chunks will remain in main memory for one full cycle in the continuous traversal of *R*. The chunks therefore leave main memory in the order that they enter main memory and their time of residence in main memory is overlapping. This leads to the staggered processing pattern of MESHJOIN. In main memory the incoming stream data is organized in a queue, each chunk defining one partition of the queue. At each point in time, each partition has seen a larger number of iterations than the previous, and started at a later position in *R* (except for the case that the traversal of *R* resets to the start of R). Figure 1 shows a pictorial representation of the MESHJOIN operation at the moment that a part *R*2 of *R* is read into the disk-buffer but is not yet processed.

Although MESHJOIN is a seminal algorithm in the field that serves as a benchmark for semi-stream joins, the algorithm makes no assumptions about the data distribution and the organization of the master data. The MESHJOIN authors reported that the algorithm performs worse with skewed data. In summary, the problems that we consider in this paper is to deal with the skew characteristic in the stream data efficiently.

4. Cache-based Semi-Stream Join

In this paper, we propose a new algorithm, Cache-based Stream-Disk Join (CSDJ), that overcomes the issues stated in previous section. This section gives a detail overview of the CSDJ algorithm and presents its cost model and tuning.

4.1 Execution Architecture

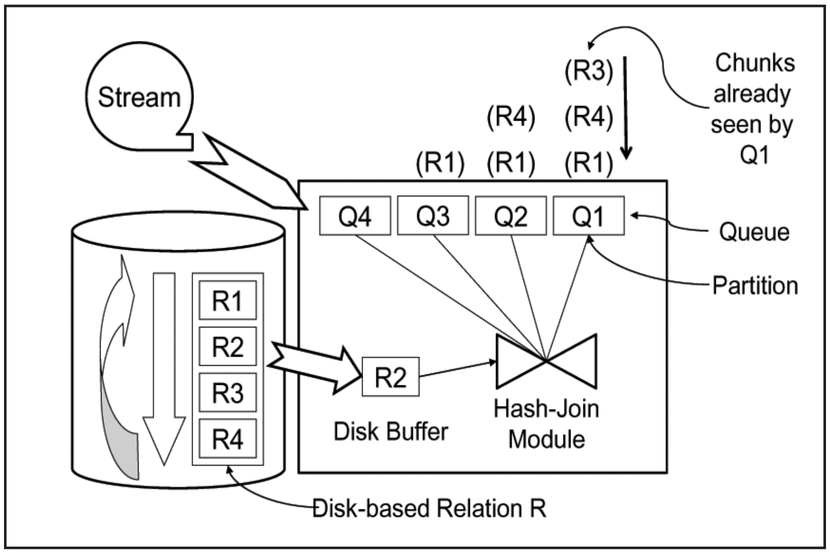


Figure 1. MESHJOIN before processing R2

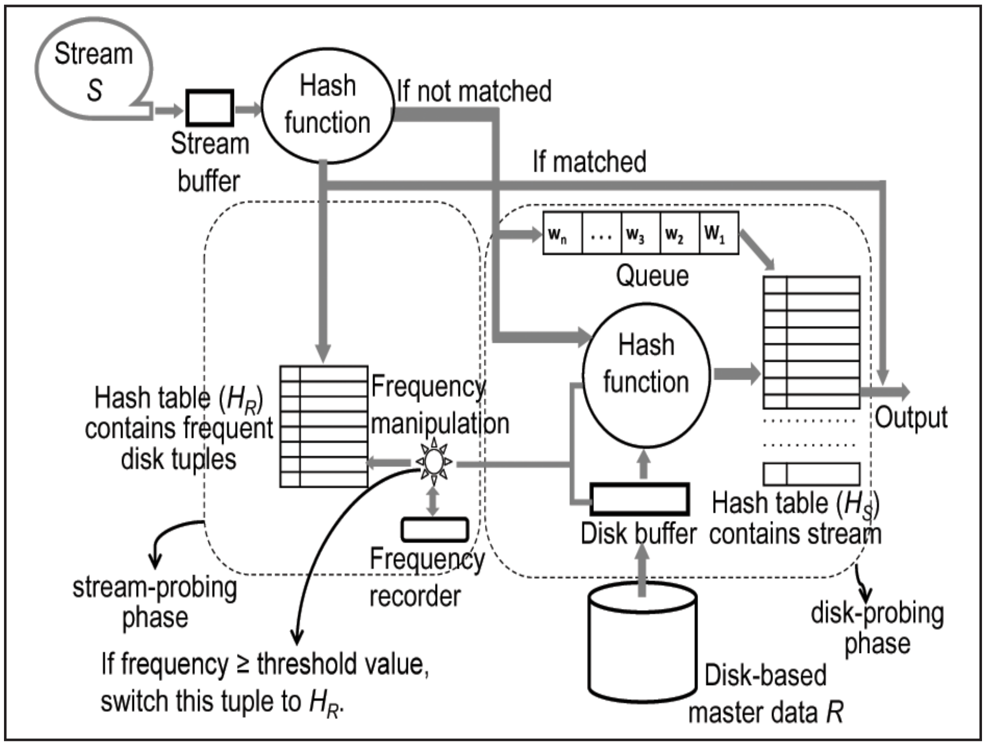


Figure 2. Execution architecture of CSDJ

The CSDJ 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 stream-probing phase. For each incoming stream tuple, CSDJ 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 CSDJ is shown in Figure 2. The largest components of CSDJ with respect to memory size is hash table H_S that stores stream tuples. The other main components of CSDJ are a disk buffer, a queue, a stream buffer, and another hash table H_R. Hash table H_R is cache that contains the most frequently accessed part of *R*. Relation *R* and stream *S* are the external input sources.

CSDJ alternates between the stream-probing and the diskprobing phases. The hash table H_S is used to store only that part of the update stream that 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. In each iteration of the disk-probing phase, the algorithm loads a set of tuples of *R* into memory to amortize the costly disk access. After loading the disk pages into the disk buffer, the algorithm probes each tuple of the disk buffer in the hash table H_S. If the required tuple is found in H_S, the algorithm generates that tuple as an output. After each iteration the algorithm removes the oldest chunk of stream tuples from H_S. This chunk is found at the top of the queue; its tuples were joined with the whole of R and are thus completely processed now. Later we call them expired stream tuples. As the algorithm reads *R* sequentially, no index on *R* is required. After one iteration of disk-probing phase, a sufficient 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 CSDJ.

The stream-probing phase (also called cache module) 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, a frequency detection process is required, which is described in Section IV-B.

Algorithm 1 Pseudo-code for CSDJ

```
Input: A disk based relation R and a stream of updates S.
Output: R ⋈ S
Parameters: w tuples of S and b number of tuples of R.
Method:
1: while (true) do
2: READ w stream tuples from the stream buffer
3: for each tuple t in w do
4: if $t \in H_R$ then
5: OUTPUT t
6: else
7: ADD stream tuple t into HS and also place its
pointer value into Q
8: end if
9: end for
10: READ b number of tuples of R into the disk buffer
11: for each tuple r in b do
12: if $r \in H_S$ then
13: OUTPUT r
14: f ← number of matching tuples found in $H_S$
15: if ($f \geq$ thresholdValue) then
16: SWITCH the tuple r into hash table $H_R$
17: end if
18: end if
19: end for
20: DELETE the oldest w tuples from $H_S$ along with
their corresponding pointers from Q
21: end while
```

4.2 Algorithm

The pseudo-code for CSDJ is shown in Algorithm 1. The outer loop of the algorithm is an endless loop, which is common in

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 MESHJOIN)
Size of each stream tuple	20 bytes (similar to MESHJOIN)
Size of each node in the queue	4 bytes (similar to MESHJOIN)

Table 1. Data specification

stream processing algorithms (line 1). The body of the outer loop has two main parts, the stream-probing phase and the disk-probing phase. Due to the endless loop, these two phases alternate.

Lines 2 to 9 specify the stream-probing phase. In this phase the algorithm reads *w* stream tuples from the stream buffer (line 1). After that the algorithm probes each tuple *t* of *w* in the disk-build hash table H_R, using an inner loop (line 3). In the case of a match, the algorithm generates the join output without storing *t* in H_S. In the case where *t* does not match, the algorithm loads *t* into H_S, while also enqueuing its pointer in the queue *Q* (lines 4-8).

Lines 10 to 20 specify the disk-probing phase. At the start of this phase, the algorithm reads *b* tuples from *R* and loads them into the disk buffer (line 10). In an inner loop, the algorithm looks up all tuples from the disk buffer in hash table H_S. In the case of a match, the algorithm generates that tuple as an output (lines 11 to 13). Since HS is a multihash- map, there can be more than one match; the number of matches is *f* (line 14).

Lines 15 and 16 are concerned with frequency detection. In line 15 the algorithm tests whether the matching frequency f of the current tuple is larger than a preset 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 arbitrary existing tuple in H_R. Finally, the algorithm removes the expired stream tuples (i.e. the ones that have been joined with the whole of R) from H_S, along with their pointer values from the queue (line 20). If the cache is not full, this means the preset threshold is too high; in this case, the threshold can be lowered automatically. Similarly, the threshold can be raised if tuples are evicted from the cache too frequently. This makes the stream-probing phase flexible and able to adapt online to changes in the stream behavior. Necessarily, it will take some time to adapt to changes, similar to the warmup phase. However, this is usually deemed acceptable for a stream-based join that is supposed to run for a long time.

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 a secondary storage. We implemented our experiments in Java using the Eclipse IDE. The relation *R* is stored on disk using a MySQL database.

Measurement Strategy: The performance or service rate of the join is measured by calculating the number of tuples processed in a unit second. In our experiments where it is necessary we calculate the confidence interval by considering 95% accuracy, but sometimes the variation is very small.

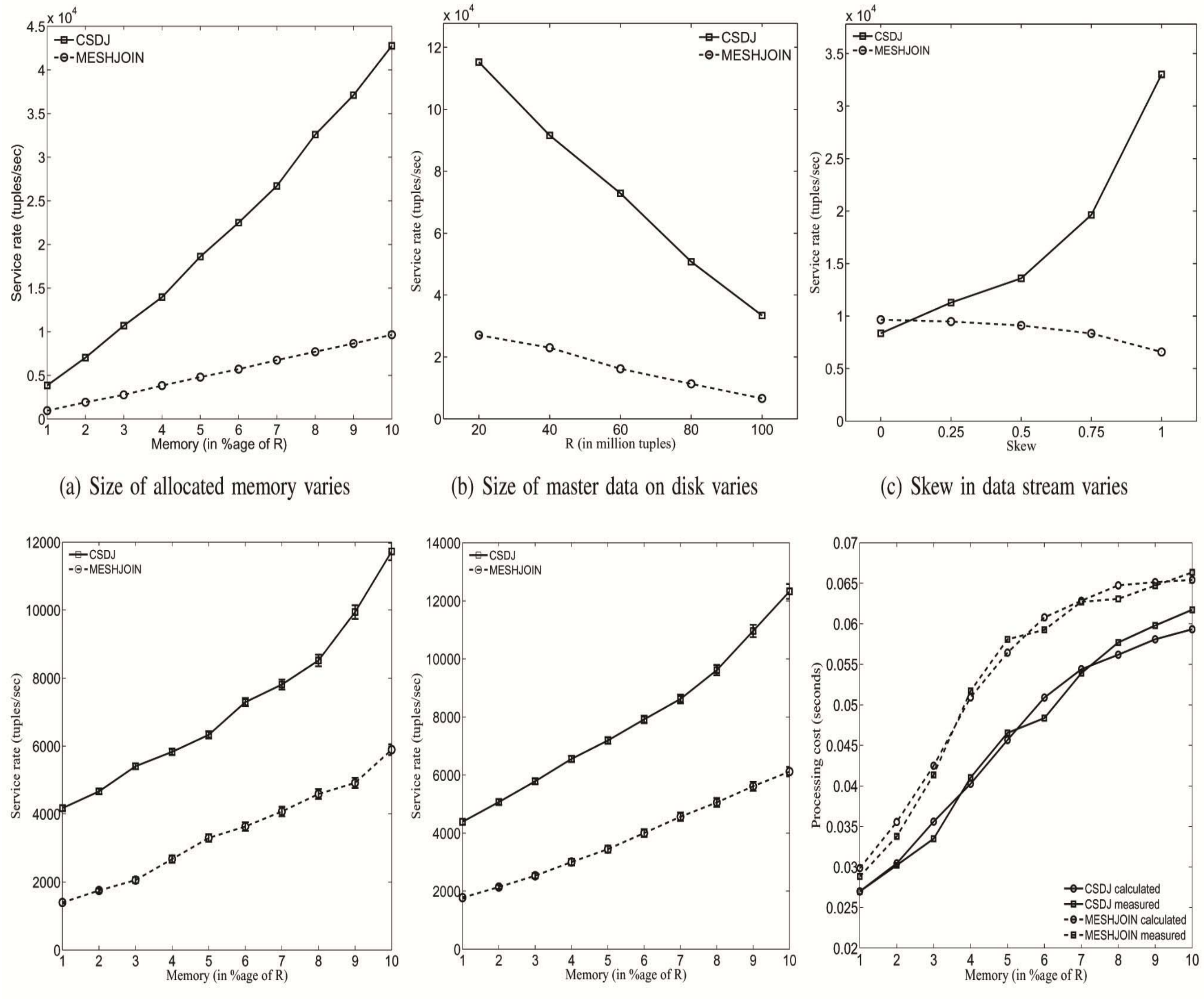
Synthetic Data: The stream dataset we used is based on the Zipfian distribution. We test the performance of both the algorithms by varying the skew value from 0 (fully uniform) to 1 (highly skewed). The detailed specifications of our synthetic dataset are shown in Table I.

TPC-H: We also analyze the performance of both 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 both the algorithms using a real-life dataset[1]. 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 CSDJ and MESHJOIN using synthetic, TPCH, and real-life data. In our experiments 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



(a) Size of allocated memory varies (b) Size of master data on disk varies (c) Skew in data stream varies

(d) TPC-H dataset (e) Real-life dataset (f) Cost validation

Figure 3. Performance analysis and cost validation

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

sake of brevity, we restrict the discussion for each parameter to a one dimensional variation, i.e. we vary one parameter at a time. We also validate our cost models for each algorithm. Due the page limit we skip the details about these cost models.

Analysis by Varying Size of Memory M: In our first experiment we compare the service rate produced by both the 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 fix the skew value equal to 1 for all settings of M. The results of our experiment are presented in Figure 3(a). From the figure it can be noted that CSDJ performs up to 7 times faster than MESHJOIN in case of 10% memory setting. While in the case of a limited memory environment (1% of R) CSDJ still performs up to 3 times better than MESHJOIN that makes it an adaptive solution for memory constraint applications.

Analysis by Varying Size of R: In this experiment we compare the service rate of CSDJ with MESHJOIN 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 3(b). From the figure it can be seen that CSDJ performs up to 3.5 times better than MESHJOIN under all settings of R.

Analysis by Varying Skew Value: In this experiment we compare the service rate of both 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 3(c) show that CSDJ again performs significantly better than MESHJOIN 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, CSDJ performs about 7 times better than MESHJOIN.

Contrarily, as MESHJOIN does not exploit the data skew, its service rates actually decrease slightly for more skewed data, which is consistent to the original MESHJOIN 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. CSDJ performs slightly worse than MESHJOIN only in a case when the stream data is completely uniform. In this particular case the streamprobing phase does not contribute considerably while on the other hand it reduces memory for the disk-probing phase.

TPC-H and Real-Life Datasets: We also compare the service rate of both the algorithms using TPC-H and reallife datasets. The details of both datasets have already been described in Section V-A. In both experiments we measure the service rate produced by both the algorithms at different memory settings. The results of our experiments using TPCH and real-life datasets are shown in Figures 3(d) and 3(e) respectively. From the both figures it can be noted that the service rate in case of CSDJ is remarkably better than MESHJOIN.

In all above experiments the reason of better performance in case of CSDJ is the addition of cache module (streamprobing phase) which processes a big part of stream without an extra I/O cost.

Cost Analysis: The cost models for both the algorithms have been validated by comparing the calculated cost with the measured cost. Figure 3(f) 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.

4. Conclusions

In this paper we discuss a new semi-stream join called CSDJ that can be used to join a stream with a disk-based master data table. We compare it with existing MESHJOIN, a seminal algorithm that can be used in the same context. CSDJ 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 MESHJOIN, CSDJ 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 provided a cost model of the new algorithm and validated it with experiments. We have performed an extensive experimental study showing an improvement of CSDJ over the earlier MESHJOIN algorithm.

References

[1] Anderson, C. (2006). The Long Tail: Why the Future of Business Is Selling Less of More. Hyperion.

[2] Arasu, A., Babu, S., Widom, J. (2002). An abstract semantics and concrete language for continuous queries over streams and relations. Technical Report 2002-57, Stanford InfoLab.

[3] Bornea, M., Deligiannakis, A., Kotidis, Y., Vassalos, V. (2011). Semi-streamed index join for near-real time execution of ETL transformations. *In*: IEEE 27th International Conference on Data Engineering (ICDE'11), p. 159 –170, April.

[4] Chakraborty, A., Singh, A. (2009). 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, p. 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Golab, L., Johnson, T., Seidel, J. S., Shkapenyuk, V. (2009). Stream warehousing with datadepot. In SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data, p. 847–854, New York, NY, USA. ACM.

[6] Golfarelli, M., Rizzi, S. (2009). A survey on temporal data warehousing. *International Journal of Data Warehousing*, 5.

[7] Naeem, M. A., Dobbie, G., Weber, G., Alam, S. (2010) . RMESHJOIN 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.

[8] Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell, N. (2007). Supporting streaming updates in an active data warehouse. *In*: ICDE 2007: Proceedings of the 23rd International Conference on Data Engineering, p. 476– 485, Istanbul, Turkey, 2007.

[9] Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell. N. (2008). Meshing streaming updates with persistent data in an active data warehouse. *IEEE Trans. on Knowl. and Data Eng*., 20 (7) 976–991.

[10] Thomsen, C., Pedersen, T. B. (2005). A survey of open source tools for business intelligence. *In*: Data Warehousing and Knowledge Discovery, p. 74–84. Springer.

[11] Vassiliadis, P. (2009). A survey of extract-transform-load technology. *IJDWM*, 5 (3)1–27.

[12] Wu, E., Diao, Y., Rizvi, S. (2006). High-performance complex event processing over streams. *In*: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06, p. 407–418, New York, NY, USA, 2006. ACM.