A Generic Front-Stage for Semi-Stream Processing

M. Asif Naeem
School of Computing and Mathematical Sciences, Auckland University of Technology
Private Bag 92006, Auckland, New Zealand
mnaeem@aut.ac.nz

Gerald Weber
Department of Computer Science, The University of Auckland
Private Bag 92019, Auckland, New Zealand
gerald@cs.auckland.ac.nz

Gillian Dobbie
Department of Computer Science, The University of Auckland
Private Bag 92019, Auckland, New Zealand
gill@cs.auckland.ac.nz

Christof Lutteroth
Department of Computer Science, The University of Auckland
Private Bag 92019, Auckland, New Zealand
lutteroth@cs.auckland.ac.nz

ABSTRACT

Recently, a number of semi-stream join algorithms have been published. The typical system setup for these consists of one fast stream input that has to be joined with a disk-based relation R. These semi-stream join approaches typically perform the join with a limited main memory partition assigned to them, which is generally not large enough to hold the whole relation R. We propose a caching approach that can be used as a front-stage for different semi-stream join algorithms, resulting in significant performance gains for common applications. We analyze our approach in the context of a seminal semi-stream join, MESHJOIN (Mesh Join), and provide a cost model for the resulting semi-stream join algorithm, which we call CMESHJOIN (Cached Mesh Join). The algorithm takes advantage of skewed distributions; this article presents results for Zipfian distributions of the type that appears in many applications.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—Systems—Query processing

General Terms

Join Operator, Performance

Keywords

Semi-stream join, Performance optimization

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, for example, in real-time data warehousing [13, 9]. In this application, the slowly changing table is typically a master data table and the stream contains incoming real-time sales data. The stream-based join is used to enrich the stream data with master data. A common type of join in this scenario is an equijoin, e.g. on a foreign key in the stream data.

In this work we consider one-to-many equijoins only, as they appear between foreign keys and the referenced primary key in another table. This is a very important class of joins that occurs naturally in data warehousing [9], online auction systems [2] and supply-chain management [20]. That is, we do not consider joins on categorical attributes, such as gender, in master data.

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. Secondly, an organization may decide to reduce the carbon footprint of the IT infrastructure: main memory as well as cloud-computing approaches can be power-hungry. Thirdly, 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 interesting building blocks for a resource-aware setup.

In this paper we present a novel caching approach that works as a front-stage for existing semi-stream join algorithms. It is different from other cache-based approaches [4, 5] in that it uses a tuple-level rather than a page-level cache. The front-stage significantly improves join performance for data with Zipfian distributions of the foreign keys, which can be found in a wide range of applications [1]. To demonstrate the front-stage, we combine it with MESHJOIN (Mesh Join), which is a seminal algorithm in the field of semi-stream joins [15, 16]. The resulting approach – called CMESHJOIN (Cached Mesh Join) – separates the concerns of optimizing

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2505515.2505734.

for a particular distribution of stream data and performing a join under general assumptions. We provide a cost model and experimental results to clarify the properties and benefits of the approach.

Our main findings can be summarized as follows:

Tuple-level cache: The caching approach in CMESHJOIN has the granularity of tuples. Every tuple in the cache is frequent in the stream data, so memory is utilized well.

Higher service rate: For skewed data, as found in many application scenarios [1] such as data warehousing, CMESHJOIN outperforms MESHJOIN. The advantage increases as the skew becomes more pronounced.

Cost model: We present a cost model for CMESHJOIN and perform a sensitivity analysis with respect to various parameters, validating the cost model in the process.

In certain applications it can be natural to use Solid State Drives (SSDs) for storing the master data in order to gain performance advantages. However, access to the master data is sequential in CMESHJOIN, and SSDs have only a moderately faster sequential access time compared to enterprise strength HDD equipment [7]. Hence we did not consider SDD technology in our discussion. Similarly, we do not consider the effect of processor cache or other main memory hardware caches, as main memory access time in CMESHJOIN is insignificant compared to disk access time.

Section 2 summarizes related work. Section 3 describes CMESHJOIN and its cost model. Section 4 describes an experimental evaluation. Section 5 concludes the paper.

2. RELATED WORK

The Symmetric Hash Join (SHJ) algorithm [19, 18] extends the original hash join algorithm in a pipeline fashion. The Double Pipelined Hash Join [11], XJoin [17] and Early Hash Join (EHJ) [12] are further extensions of SHJ for the pipeline execution of join. All these algorithms take both inputs in the form of streams while our focus is on joins between stream data and non-stream data.

A number of tools have been developed for stream warehousing that can process stream data with archive data [3, 10, 8, 9]. However, these tools do not provide optimal solutions for the non-uniform characteristics of stream data.

MESHJOIN (Mesh Join) [15, 16] has been designed specifically 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) [14] clarifies the dependencies among the components of MESHJOIN. As a result, it improves the performance slightly. However, R-MESHJOIN again does not consider the non-uniform characteristic of stream data.

The partition-based join algorithm described in [5] improves MESHJOIN performance. It uses a two-level hash table for attempting to join stream tuples as soon as they arrive, and uses a partition-based waiting area for other stream tuples. 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. Moreover, [5] uses page-level cache, so the cache memory is not fully exploited if some tuples on a cached page are infrequent in stream data.

Semi-Streaming Index Join (SSIJ) [4] was developed recently to join stream data with disk-based data. Again SSIJ also uses a page-level cache. The published work does not include a mathematical cost model. Consequently, the criteria for choosing optimal parameters for SSIJ are unclear.

Some other approaches [6, 9] have considered the problem of joining stream data with disk-based data, but to the best of our knowledge they did not propose an algorithm for it.

3. CMESHJOIN

We propose a generic cache component that can be used as a front-stage for an arbitrary semi-stream join algorithm. It exploits skewed distributions in the stream foreign keys in order to improve the service rate. We demonstrate this concept by adding the front-stage to MESHJOIN, resulting in a new algorithm called CMESHJOIN. This section gives a high-level description of CMESHJOIN; a detailed walkthrough can be found in Section 3.1. Both the front-stage and MESHJOIN are hash joins, so the CMESHJOIN algorithm can be seen overall as possessing two complementary hash join phases, somewhat similar to Symmetric Hash Join [19, 18]. One phase, the MESHJOIN, uses R as the probe input, with the largest part of R typically being stored in tertiary memory. The other join phase, the front-stage, uses the stream as the probe input and deals only with a small part of R. For each incoming stream tuple, CMESHJOIN first uses the front-stage to find a match for frequent requests quickly, and if no match is found, the stream tuple is forwarded to the MESHJOIN phase.

The execution architecture of CMESHJOIN is shown in Figure 1. Relation R and stream S are the external input sources of the join. The key components of CMESHJOIN 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. H_R is the cache that contains the most frequently accessed part of R. The other main components of CMESHJOIN are a disk buffer, a queue, a frequency recorder, and a stream buffer. The disk buffer is used to load parts of R into memory using equallly sized partitions. The queue stores *pointers* to the stream tuples in H_S, keeping track of their order and enabling the deletion of fully processed tuples. The frequency recorder records the access frequency of each tuple stored in H_R. The stream buffer is only a small buffer for holding part of the stream for a while, if necessary. For reference, we have preserved the original architecture of MESHJOIN in the MESHJOIN phase, although alternative architectures are possible (e.g. by using an order-preserving hash table data structure instead of the queue).

CMESHJOIN alternates between the front-stage and the MESHJOIN 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 front-stage phase ends if H_S is completely filled or if the stream buffer is empty. Then the MESHJOIN phase becomes active. In each iteration of the MESHJOIN 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

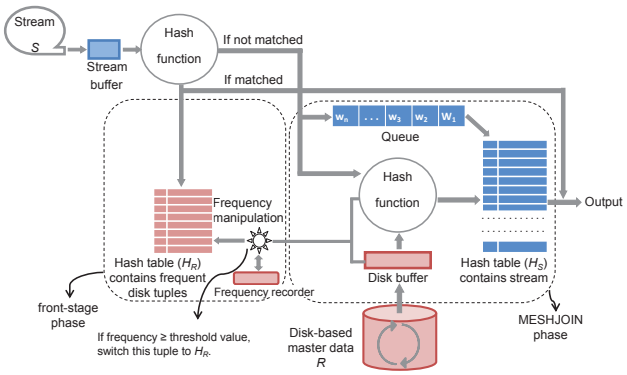


Figure 1: Data structures and architecture of CMESHJOIN

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 have thus been completely processed at this point. Later we call them expired stream tuples. As the algorithm reads R sequentially, no index on R is required. After one probe step, a sufficient number of stream tuples are deleted from H_S, so the algorithm switches back to the front-stage phase. One phase of front-stage with a subsequent phase of MESHJOIN constitutes one outer iteration of CMESHJOIN. The front-stage phase is used to boost the performance of the algorithm by quickly matching the most frequent master data. An important question is how frequently a master data tuple must be used in order to get into this phase, so that the memory sacrificed for this phase really delivers a performance advantage. In Section 3.2 we give a precise and comprehensive analysis that shows that a remarkably small amount of memory assigned to the front-stage phase can deliver a substantial performance gain. In order to corroborate the theoretical model, we present experimental performance measurements in Section 4. For determining very frequent tuples in R and loading them into H_R, a frequency detection process is required, which is described in Section 3.1.

3.1 Algorithm

The execution steps for CMESHJOIN are shown in Algorithm 1. The outer loop of the algorithm is an endless loop, which is common in stream processing algorithms (line 1). The body of the outer loop has two main parts: the front-stage phase and the MESHJOIN phase. Due to the endless loop, these two phases alternate. The MESHJOIN algorithm has two parameters, the disk-buffer size b and the number w of stream tuples processed in each iteration, which are computed once the memory allocation for MESHJOIN and the size of R are fixed, as described in [15, 16].

Lines 2 to 11 specify the front-stage phase. In this phase the algorithm reads stream tuples from the stream buffer (line 4). The algorithm probes each stream tuple t 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 (line 6). In the case where t does not match, the algorithm loads t into H_S, while also enqueuing its pointer in the queue Q (line 8). The front-stage phase stops once w stream tuples have been loaded into H_S.

Algorithm 1 CMESHJOIN

Input: A disk based relation R and a stream of updates S
Output: $R \bowtie S$
Parameters: w (tuples of S) and b (tuples of R).
Method:

```
1:  while (true) do
2:    u ← 0 (the number of unmatched tuples)
3:    while (u < w) do
4:      READ stream tuple t from the stream buffer
5:      if t ∈ H_R then
6:        OUTPUT t
7:      else
8:        ADD stream tuple t into H_S and also place a
          pointer to t into Q
9:        u ← u + 1
10:     end if
11:   end while
12:   READ b number of tuples of R into the disk buffer
13:   for each tuple r in b do
14:     if r ∈ H_S then
15:       OUTPUT r
16:       f ← number of matching tuples found in H_S
17:       if (f ≥thresholdValue) then
18:         SWITCH the tuple r into hash table H_R
19:       end if
20:     end if
21:   end for
22:   DELETE the oldest w tuples from H_S along with
      their corresponding pointers from Q
23: end while
```

Lines 12 to 22 specify the MESHJOIN phase. At the start of this phase, the algorithm reads b tuples from R and loads them into the disk buffer (line 12). 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 (line 15). Since H_S is a multi-hash-map, there can be more than one match; the number of matches is f (line 16).

Lines 17 and 18 are concerned with frequency detection. In line 17 the algorithm tests whether the matching frequency f 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 using the frequency recorder. 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 pointers from the queue (line 22). If the cache is not full, it means the 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 front-stage 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.

3.2 Cost Model for Tuning

We now show how a cost model can be used to tune CMESHJOIN theoretically. The notations we use in our cost model are given in Table 1. We use the assumption

Table 1: Notation for CMESHJOIN cost model

Parameter name	Symbol
Total allocated memory *(bytes)*	M
Service rate *(processed tuples/sec)*	μ
Size of stream tuple *(bytes)*	v_S
Size of disk tuple *(bytes)*	v_R
Disk buffer size *(tuples)*	b
Size of H_R *(tuples)*	h_R
Size of H_S *(tuples)*	h_S
Disk relation size *(tuples)*	R_t

that the stream of updates S has a Zipfian distribution with an exponent of 1. In this Zipfian distribution, the frequency of the second element is half that of the first element. Similarly, the frequency of the third element is $\frac{1}{3}$ that of the first element, and this decreasing pattern continues in the tail of the distribution [1]. In this case, the matching probability for stream S in the front-stage phase can be determined using Equation 1. The denominator is a normalization term to ensure all probabilities sum up to 1.

$$p_N(h_R) = \frac{\sum_{x=1}^{h_R} \frac{1}{x}}{\sum_{x=1}^{R_t} \frac{1}{x}} \quad (1)$$

We consider the derivative of Equation 1:

$$\frac{dp_N}{dh_R} \approx p_N(x+1) - p_N(x) \quad (2)$$

MESHJOIN is used in the second phase of CMESHJOIN (the MESHJOIN phase). The major portion of the total memory is assigned to the MESHJOIN phase. The memory for each component relevant to the front-stage phase can be calculated as follows:

Memory for H_R (bytes)$=h_R \cdot v_R$
Memory for frequency recorder (bytes)$=8h_R$

Therefore the memory consumption for MESHJOIN can be estimated as shown in the enumerator of the following equation. MESHJOIN's performance is roughly proportional to the memory available and inversely proportional to the size of R, therefore the service rate produced in this phase can be calculated using the following equation.

$$\mu_{dp}(M) \approx \frac{M - (8 + v_R)h_R}{R_t v_R} \quad (3)$$

Since we are considering Zipfian distributions with finite skew, there will always be tuples that need to be processed through the MESHJOIN phase and hence $p_N(h_R) < 1$. The service rate of CMESHJOIN is a function of h_R and can be written as

$$\mu_{cm}(h_R) \approx \frac{\mu_{dp}(M)}{1 - p_N(h_R)} \quad (4)$$

Now we take the derivative of the above equation using the chain rule.

$$\frac{d\mu_{cm}(h_R)}{dh_R} \approx \frac{d\mu_{cm}(h_R)}{dp_N} \times \frac{dp_N}{dh_R} \quad (5)$$

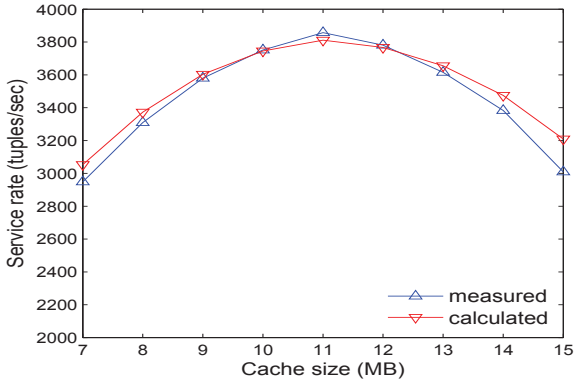


Figure 2: Tuning of the cache module: empirical measurements vs. estimates based on the cost model

By using Equations 2, 3, and 4 in Equation 5, we get:

$$\frac{d\mu_{cm}}{dh_R} \approx \frac{M - (8 + v_R)h_R}{R_t v_R [1 - p_N(h_R)]^2} \times [p_N(x+1) - p_N(x)] \quad (6)$$

To ensure that the root of the first derivative corresponds to a maximum, we consider the second derivative:

$$\frac{d_2\mu_{cm}}{dh_R^2} \approx \frac{2[M - (8 + v_R)h_R]}{R_t v_R [1 - p_N(h_R)]^3} \times [\frac{dp_N}{dh_R}]^2 \quad (7)$$

From here we can determine the value of h_R at which the value of μ_{cm} reaches a maximum. Once the optimal memory size for the front-stage component is determined, the rest of the memory is assigned to the MESHJOIN components using the tuning approach presented in [15, 16].

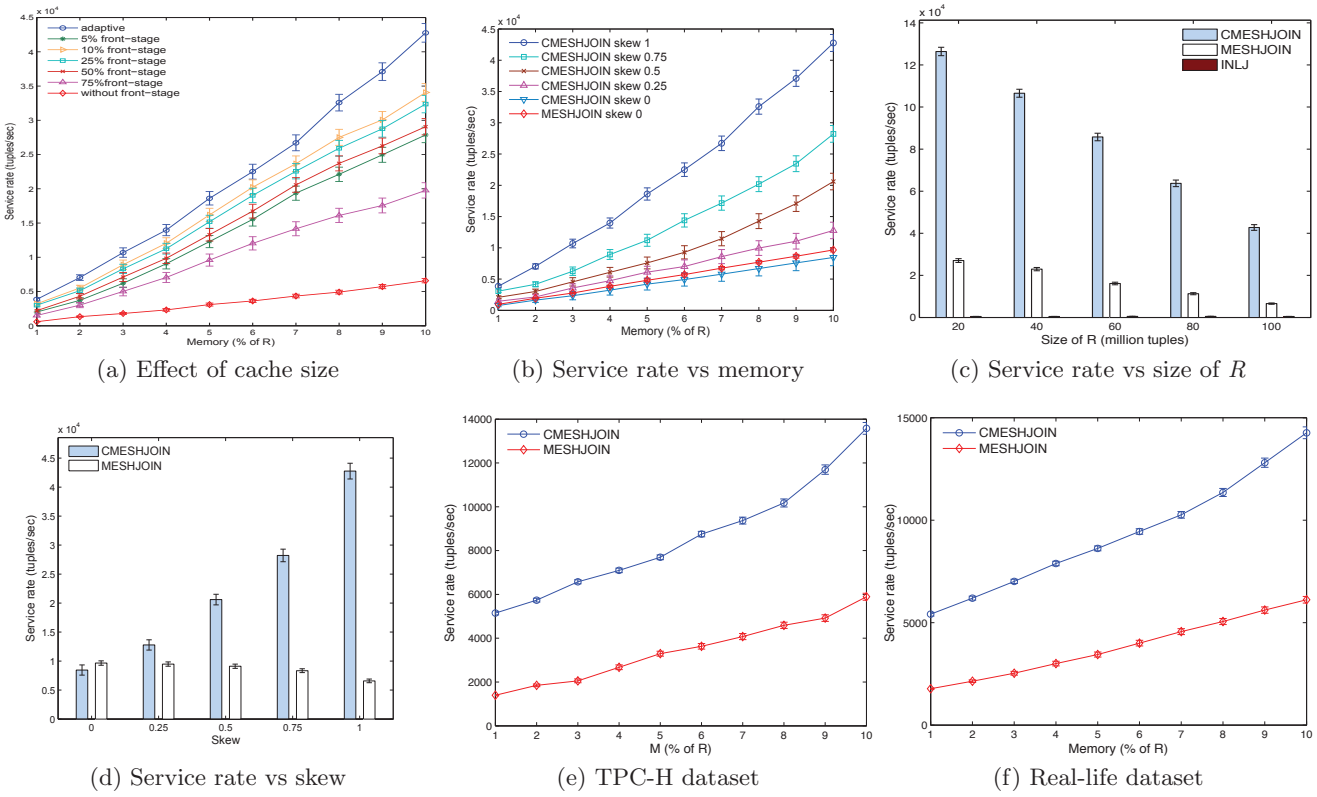
4. EXPERIMENTAL EVALUATION

Hardware and software specifications: We performed our experiments on a *Pentium-i5* with 8GB main memory and 500GB HDD. All experiments were implemented in Java, using the `org.apache.MultiHashMap` as a hash table data structure with support for multiple values per key.

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 completed their warmup phase before starting the actual measurements. The calculation of the confidence intervals is based on 1000 to 4000 measurements for one setting.

Data specifications: We analyzed the service rate of the algorithms using synthetic, TPC-H, and real-life datasets. The relation R was stored on disk using a MySQL database. To measure the I/O cost more accurately, we set the fetch size for `ResultSet` equal to the disk buffer size. The synthetic stream dataset was based on a Zipfian distribution. The synthetic master data was unsorted, did not have an index, and comprised 100 million tuples (≈11.18GB). We used disk tuple, stream tuple and queue pointer sizes similar to the original MESHJOIN (120, 20 and 4 bytes respectively). For the TPC-H dataset we used a scale factor of 100 with the table `Customer` as master data and the table `Order` as stream data. The real-life dataset[1], which was also used to evaluate the original MESHJOIN, contains cloud information stored

[1]Available at: `http://cdiac.ornl.gov/ftp/ndp026b/`



(a) Effect of cache size (b) Service rate vs memory (c) Service rate vs size of R

(d) Service rate vs skew (e) TPC-H dataset (f) Real-life dataset

Figure 3: Service rate analysis (Figures (a) to (d) based on synthetic data)

in a summarized weather report format. Weather data from different months were joined (20 million tuples master data, 6 million tuples stream data, tuple size 128 bytes), using the common attribute longitude (`LON`).

Cache size: The cost model predicts that the cache size h_R influences the service rate of CMESHJOIN. In order to obtain the optimal cache size, we performed experiments and compared the results with estimates obtained from the cost model. The results are shown in Figure 2 for a fixed total memory of 0.11GB and a size of 100 million tuples (11.18GB) for R. As can be seen there is a close match between estimates and measurements. To get optimal results, the memory allocations for the stream-probing phase (cache module) and the disk-probing phase (MESHJOIN) have to be balanced. In our experiments, we adjusted the cache size automatically based on our cost model. Figure 3(a) shows the effect of the cache size for various memory budgets.

Analysis by varying size of memory: We compared the service rate of MESHJOIN and CMESHJOIN while varying the memory size from 1% to 10% of R, with the size of R being 100 million tuples. For each memory setting we repeated the experiment, considering different skew values in the streaming data. In the case of MESHJOIN, we only considered a skew value of 0 since MESHJOIN performs best with uniform data (as indicated later in Figure 3(d)). The results are shown in Figure 3(b). From the figure it can be noted that due to the front-stage phase CMESHJOIN performed up to 7 times faster than MESHJOIN with the 10% memory setting. In a limited memory environment (1% of R), CMESHJOIN still performed up to 5 times better than MESHJOIN, which makes it an adaptive solution suitable for memory-constrained applications.

Analysis by varying size of R: We measured the service rate for CMESHJOIN, MESHJOIN and INLJ (index nested loop join) at different sizes of R, with a fixed memory size (≈1.12GB) and a skew value of 1. Since the size of R is the only parameter that affects INLJ's service rate, we did not include it in the analysis of memory and skew. From Figure 3(c) we see that CMESHJOIN performed up to 4.5 times better than MESHJOIN when the size of R was 20 million tuples. This improvement increased to 6.5 times when the size of R was 100 million tuples. One important fact that can be noted here is that for CMESHJOIN, due to the front-stage phase, the service rate does not decrease inversely when increasing the size of R, as in MESHJOIN. CMESHJOIN substantially outperforms INLJ because of INLJ's high I/O cost due to random probes to the master data. Also, INLJ processes one stream tuple in each iteration, so does not amortize the expensive I/O cost on fast streaming data.

Analysis by varying skew value: We compared the service rate of both algorithms while varying the skew of the streaming data. To vary the skew, we varied the Zipfian exponent from 0 to 1. At 0 the input stream S is uniform, while at 1 the stream has a strong skew. The size of R was fixed at 100 million tuples (≈11.18GB) and the available memory was set to 10% of R (≈1.12GB). The results presented in Figure 3(d) show that CMESHJOIN performs significantly better than MESHJOIN, even for only moderately skewed data. This improvement became more pronounced for increasing skew values. At a skew of 1, CMESHJOIN performs

approximately 7 times better than MESHJOIN. As MESHJOIN does not exploit skew in its algorithm, its service rate actually decreased slightly for more skewed data, which is consistent with the original MESHJOIN findings. We do not present data for skew values larger than 1, which would imply short tails. However, we predict that for such short tails the trend continues. From Figures 3(b) and (d) we can see that CMESHJOIN only performs worse than MESHJOIN when the stream data is completely uniform (exponent of 0). However, this difference is a constant factor.

TPC-H and real-life datasets: In these experiments we measured the service rate produced by both algorithms at different memory settings. From Figure 3(e) it can be noted that CMESHJOIN performed about 4 times better than MESHJOIN for TPC-H data, which is significant especially for a smaller memory size, 1% of R. Similarly, it is obvious from Figure 3(f) that CMESHJOIN outperforms MESHJOIN for the real-life data under all memory settings.

5. CONCLUSIONS

In this paper we have discussed a new semi-stream join called CMESHJOIN. This algorithm is based on MESHJOIN, and extends it with a cache front-stage in order to exploit skewed distributions. We have provided a theoretical cost model as well as experimental results that show that this approach yields substantial speed-ups for Zipfian distributions in the data. This type of skewed, non-uniformly distributed data is frequently found in real-world applications.

Our findings suggest that the front-stage in CMESHJOIN is able to exploit skewed distributions in the data in very general circumstances and provide the join output for a large portion of the stream, thus reducing the overall load for the subsequent stages. We expect that the front-stage is generic and can be used with any semi-stream join operator to enhance its service rate, because it has little overhead and delivers substantial speed-up under very general assumptions. In the future, we will consider many-to-many equijoins and certain classes of non-equijoins.

Source URL: We have provided open-source implementations of CMESHJOIN that can be used for further analysis. `https://www.cs.auckland.ac.nz/research/groups/serg/cmj/`.

6. REFERENCES

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

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

[3] M. H. Bateni, L. Golab, M. T. Hajiaghayi, and H. Karloff. Scheduling to minimize staleness and stretch in real-time data warehouses. In *Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'09)*, pages 29–38. ACM, 2009.

[4] M. Bornea, A. Deligiannakis, Y. Kotidis, and V. Vassalos. Semi-streamed index join for near-real time execution of ETL transformations. In *ICDE'11*, pages 159 –170, April 2011.

[5] A. Chakraborty and A. Singh. A partition-based approach to support streaming updates over persistent data in an active datawarehouse. In *International Symposium on Parallel & Distributed Processing (IPDPS'09)*, pages 1–11. IEEE, 2009.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD'03*, pages 668–668. ACM, 2003.

[7] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09*, pages 181–192. ACM, 2009.

[8] L. Golab and T. Johnson. Consistency in a stream warehouse. In *Conference on Innovative Data Systems Research (CIDR'11)*, pages 114–122, 2011.

[9] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with datadepot. In *SIGMOD'09*, pages 847–854. ACM, 2009.

[10] L. Golab, T. Johnson, and V. Shkapenyuk. Scheduling updates in a real-time stream warehouse. In *ICDE'09*, pages 1207 –1210, 2009.

[11] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. *SIGMOD Rec.*, 28(2):299–310, 1999.

[12] R. Lawrence. Early Hash Join: A configurable algorithm for the efficient and early production of join results. In *VLDB'05*, pages 841–852. VLDB Endowment, 2005.

[13] M. A. Naeem, G. Dobbie, and G. Weber. An event-based near real-time data integration architecture. In *EDOC'08 Workshops*, pages 401–404. IEEE, 2008.

[14] M. A. Naeem, G. Dobbie, G. Weber, and S. Alam. R-MESHJOIN for near-real-time data warehousing. In *DOLAP'10*. ACM, 2010.

[15] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N. Frantzell. Supporting streaming updates in an active data warehouse. In *ICDE'07*, pages 476–485, 2007.

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

[17] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23:2000, 2000.

[18] A. N. Wilschut and P. M. G. Apers. Pipelining in query execution. In *International Conference on Databases, Parallel Architectures and Their Applications (PARBASE'90)*, pages 562–562. IEEE, March 1990.

[19] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *International Conference on Parallel and Distributed Information Systems (PDIS'91)*, pages 68–77. IEEE, 1991.

[20] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD'06*, pages 407–418. ACM, 2006.