Skewed Distributions in Semi-Stream Joins: How much can Caching help?

M. Asif Naeem

School of Engineering, Computer and Mathematical Sciences, Auckland University of Technology, Private Bag 92006, Auckland, New Zealand.

Gillian Dobbie, Christof Lutteroth, and Gerald Weber

Department of Computer Science, The University of Auckland, Private Bag 92019, Auckland, New Zealand.

Abstract

Semi-stream join algorithms join a fast data stream with a disk-based relation. This is important, for example, in real-time data warehousing where a stream of transactions is joined with master data before loading it into a data warehouse. In many important scenarios, the stream input has a skewed distribution, which makes certain performance optimisations possible.

We propose two such optimisation techniques: 1) a caching technique for frequently used master data, and 2) a technique for selective load shedding of stream tuples. The caching technique is fine-grained, operating on a tuple-level. Furthermore, it is generic in the sense that it can be applied to different semi-stream join algorithms to deal with high loads. We analyze it by combining it with various well-known semi-stream joins, and show that it improves the service rate by more than 40% for typical data with skewed distributions. The load shedding technique sheds the fraction of the stream that is most expensive to join. In contrast to existing approaches, the service rate improves under load shedding. We present experimental data showing significant improvements as compared to related approaches and perform a sensitivity analysis for various internal parameters.

Keywords: Semi-stream processing, join, front-stage cache, 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 [1, 2, 3] where the slowly changing table is typically a master data table and the stream contains incoming real-time sales data. The join is used to enrich the stream data with master data. A common type of join in this scenario is an equijoin.

In this work, we consider one-to-many equijoins: we assume the stream data contains a foreign key referring to the master data. This is a very important class as they are used naturally to join a stream of updates and master data in a data warehousing context [3], online auction systems [4] and supply-chain management [5]. We do not consider, for example, joins on categorical attributes in master data, such as gender. In the aforementioned applications, stream data usually has a skewed *stream distribution*: we assume the stream tuples to have random foreign keys, and we assume these keys to follow a Zipfian distribution (which can be a uniform distribution in the extreme).

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 resources are of interest. Firstly, the master data may simply be too large (e.g. dozens of GBs) for the resources allocated for a stream join, so that a scalable algorithm is necessary. Secondly, low-resource consumption approaches may be necessary when mobile and embedded devices are involved. For example, stream joins such as the ones discussed here could be used in sensor networks. As a consequence, semi-stream join algorithms that can function with limited resources are important building blocks for a resource-aware system setup.

In this article we present optimization approaches that allow semi-stream join algorithms to perform well with limited resources. First, we propose a caching approach that works as a front-stage for existing semi-stream join algorithms. The approach was first presented in [6] and we now provide an extensive analysis. It is different from other cache-based approaches [7, 8] 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 [9]. We do not consider the ordering of stream tuples in the join output as it is not important particularly in the scenario of data warehousing. To evaluate the approach, we combine the front-stage with two known semi-stream algorithms: (i) MESHJOIN [10, 11], (ii) HYBRIDJOIN [12]. Our experimental results demonstrate the performance benefit of the approach. Furthermore, in order to obtain a baseline measurement of the influence of the front stage, we combine it with the textbook Index Nested Loop Join (INLJ) [13]. The size of the front stage is kept constant in all three variants in order to make the measurements of the impact of the front stage comparable.

Second, we propose a novel load shedding approach. In our approach tuples are shed only after being *considered*, while other load shedding approaches typically shed tuples before they are processed. In our approach tuples are kept as candidates for joining for a certain amount of time, and if they are not joined, they are shed to the disk/network. In this way the algorithm determines the stream tuples that are most costly to process, without much overhead. As a consequence load shedding increases the service rate, i.e. the rate of join outputs. By contrast, in other load shedding approaches for semi-stream joins, the service rate does not change significantly under shedding; the purpose of the load shedding is simply to store the overhead load. Our experimental data shows that our approach provides a significant improvement.

In summary, we provide the following contributions. **1) A generic front-stage for tuple-level caching:** We present a front-stage caching component, which has the granularity of tuples, with a number of well-known semi-stream join algorithms and study its effect on performance. **2) A novel load shedding technique:** We present a load shedding approach that sheds the tuples that are most expensive to process, thus increasing the service rate. We measure the service rate under load shedding and compare it with other related approaches. **3) Sensitivity analysis:** We perform a sensitivity analysis with respect to various parameters, validating a cost model in the process.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 discusses how the caching strategy can be seen as independent, by studying two well-known algorithms. Section 4 introduces the load shedding approach, including a problem overview and existing approaches. Section 5

describes an experimental study of the proposed approaches. Section 6 concludes the paper.

2. Related work

Early stream-based joins focused on scenarios where both inputs are in the form of streams. Such joins are called stream joins, as opposed to semi-stream joins.

The Symmetric Hash Join (SHJ) algorithm [14, 15] is a stream join that extends the original hash join algorithm. It produces join output as early as possible, i.e. is non-blocking, while increasing the memory requirement. SHJ uses a separate hash table for each of the two input relations. When a tuple t of one input arrives, SHJ probes the hash table of the other input, generates the result (if any) and then stores t into the hash table of its own stream. SHJ can produce results before reading either input relation entirely; however, it stores both relations in memory.

XJoin [16] is an extended form of SHJ that handles memory overflow by flushing the largest partition to disk. XJoin presents a three stage strategy to switch its execution state between disk and memory. The first priority is given to the memory-resident tuples. While there is no incoming stream data, the algorithm executes the second, disk-to-memory phase and lastly deals with the tuples stored on disk (disk-to-disk) in the case when the inputs are terminated. Duplicate tuples are avoided by using a timestamp approach.

The Double Pipelined Hash Join (DPHJ) [17] is also an extension of SHJ based on two stages. In the first stage, which is similar to SHJ and XJoin, the algorithm joins the tuples which are in memory. In the second stage the algorithm marks the tuples which are not joined in memory and joins them on the disk. In DPHJ duplication of tuples is possible in the second phase when all tuples from both inputs have been read and the final clean-up join is executed. This algorithm is suitable for medium-size data but does not perform well for large-size data. A related symmetric join algorithm is the Hash-Merge Join (HMJ) [18], which is based on push technology and consists of two phases, hashing and merging.

Early Hash Join (EHJ) [19] is an improved version of XJoin with a different flushing strategy and a simplified technique to determine the duplicate tuples. EHJ uses a biased flushing strategy flushes the partition with the largest input first, similar to Dynamic Hash Join [20]. The technique used in EHJ to determine duplicate tuples is chosen by cardinality; only for many-to-many relationships arrival timestamps are used.

MJoin [21], a generalized form of XJoin, extends the symmetric binary join operators to handle multiple inputs, using a separate hash table for each input. When a tuple arrives from an input, it is stored in the corresponding hash table and is probed in the other hash tables. It is not necessary to probe all the hash tables for each arrival, as the sequence of probing stops when a probed tuple does not have a match in a hash table. The methodology for choosing the correct sequence of probing is determined by performing the most selective probes first. The algorithm uses a coordinated flushing technique that involves flushing the same partition on the disk for all inputs. To identify duplicate tuples, MJoin uses two timestamps for each tuple, the arrival time and the departure time from memory.

Adaptive, Hash-partitioned Exact Window Join (AH-EWJ) [22] has been introduced to produce accurate results for sliding window joins over data streams. AH-EWJ can also be used to join a stream

with disk-based data. However, the focus of this approach is on giving the join output in the order of the stream input.

MESHJOIN (Mesh Join) [10, 11] has been designed especially for joining a continuous stream with a disk-based relation, like the scenario in active data warehouses. MESHJOIN 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, and the MESHJOIN authors report that the algorithm performs worse with skewed data[10, 11]. We previously presented a variant, R-MESHJOIN (reduced Mesh Join) [23], in order to clarify certain dependencies among the components of MESHJOIN. R-MESHJOIN is simpler and has slightly improved performance compared to MESHJOIN.

The Index Nested Loop Join (INLJ) [13] is a simple textbook algorithm that can also be used as a semi-stream join. It is trivially capable of dealing with intermittent data streams. However, every index has to be considered non-clustered with respect to the stream data. This is because stream data arrive in the order that the updates are performed. For example, if the join is over *product-id* with the stream representing purchases, and the master data is indexed by *product-id*, the index is non-clustered in that regard. INLJ is known to be inefficient for non-clustered index access, as the disk I/O cost cannot be amortized over a fast incoming data stream and eventually produces a low service rate.

The partition-based join algorithm described in [8] improves MESHJOIN's performance and can also deal with intermittent streams. 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. Moreover, it uses a page-level cache, so the cache memory is not fully exploited if some tuples on a cached page are infrequent in the stream.

The Semi-Streamed Index Join (SSIJ) [7] operates in three phases: the pending phase, the online phase and the join phase. In the pending phase, the stream tuples wait in an input buffer until the size of the buffer is less than the predefined threshold limit or the stream ends. Once the size of the input buffer crosses the threshold limit, the algorithm starts its online phase. In the online phase, stream tuples from the input buffer are looked up in cached disk blocks. If the required disk tuple is in the cache, the join is executed and the algorithm produces an output. In the case of a cache miss, the algorithm flushes the stream tuple into a stream buffer where it waits for the join phase. The algorithm specifies another threshold on the stream buffer, and during the join phase a disk block is loaded if the size of the stream buffer is greater than the threshold value. The algorithm joins the tuples from the stream buffer with the tuples in the loaded disk block. It implements a utility counter for each disk block that determines which disk blocks need to be retained in memory. Again, SSIJ uses a page-level cache. The published work does not include a mathematical cost model, so the criteria for choosing optimal parameters for SSIJ are unclear.

Another algorithm HYBRIDJOIN (Hybrid Join) [12] 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. 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 (e.g. skewed in stream data) are taken into consideration.

A number of tools have been developed for stream warehousing that can process stream data with archive data [3, 24, 25, 26]. However, these tools do not provide optimal solutions for the non-uniform characteristics of stream data. Some other approaches [3, 27] 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. Caching

In this section we propose a generic caching technique for semi-stream joins, which is based on previous work on the semi-stream join algorithms CMESHJOIN [6] and CACHEJOIN [28]. We generalise the contributions of the previous works and show that they can be seen as instances of one generic caching technique that can act as a front-stage for any semi-stream join algorithm. We also introduce a baseline for measurements of the effects of caching, by combining the caching module with a simple index nested loop join.

The generic caching technique differs from other approaches such as [7, 8] in two aspects. **1) Granularity**: the technique is a tuple-level cache, i.e. individual master data tuples are cached. Hence every tuple in the cache is frequent in the stream, ensuring optimal memory usage. By contrast, existing approaches use a page-level cache, which can at best match a tuple-level cache in memory utilization, and this only in extreme, artificial cases of locality. **2) Genericity**: the technique can be easily combined with any semi-stream join operator, and then gives some guarantee on better performance with respect to skewed distributions. It guarantees that frequent tuples are joined fast, while generally having a small impact on memory consumption. If the join operator is not in itself already optimized for skewed distributions, this can enhance the performance significantly. Because of its generic nature, we call this cache module a *front-stage* in the following. In the following, we show how to combine this front-stage with three semi-stream join algorithms.

3.1. CMESHJOIN (Cached Mesh Join)

We first illustrate the use of the proposed generic front-stage cache component with the well-known MESHJOIN algorithm. We call the resulting combined algorithm CMESHJOIN (Cached Mesh Join) [**?**]. Both the front-stage as well as MESHJOIN use hash joins, so the CMESHJOIN algorithm can overall be seen as possessing two complementary hash join phases, somewhat similar to Symmetric Hash Join [14, 15]. The MESHJOIN phase uses the master data relation R as the probe input, with the largest part of R typically being stored on disk. The new front-stage phase 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.

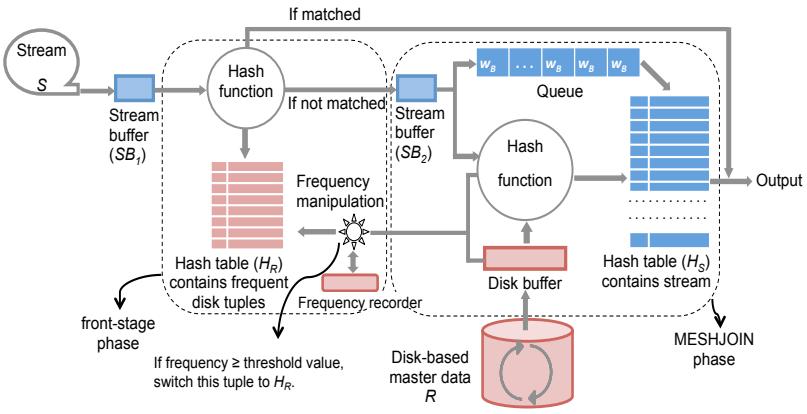


Figure 1: Data structures and architecture of CMESHJOIN

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 (Q), a frequency recorder, and two stream buffers (SB_1 and SB_2). The disk buffer is used to load parts of R into memory using equal-size partitions. Q 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. Both stream buffers are small waiting buffers to hold part of the stream for a while, if necessary. These components are included in the diagram for completeness, but are in reality always tiny e.g. 0.05MB for each was sufficient in all our experiments. For reference, we have preserved the original architecture of MESHJOIN in the MESHJOIN phase, although alternative architectures are thinkable (e.g. using an order-preserving hash table data structure instead of the queue).

CMESHJOIN alternates between the front-stage and the MESHJOIN phases. 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 full or if the stream buffer (SB_1) is empty. Then the MESHJOIN phase becomes active. In each iteration of the MESHJOIN phase, the algorithm loads a set of tuples (a partition) of R into memory to amortize the costly disk access. After loading the disk tuples into the disk buffer, the algorithm probes each tuple of the disk buffer in 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 end of Q; 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 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

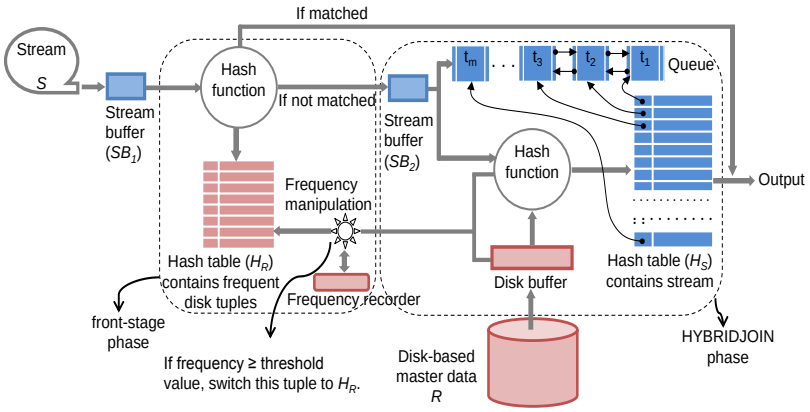


Figure 2: Data structures and architecture of CHYBRIDJOIN

advantage. In Section 5.4 we give a precise and comprehensive analysis that shows that a remarkably small amount of memory (15% of total memory) assigned to the front-stage phase can deliver a substantial performance gain. For determining very frequent tuples in R and loading them into H_R, a frequency detection process is required, which is described in Section 3.2.1.

3.2. CHYBRIDJOIN (Cached Hybrid Join)

We now illustrate how the proposed front-stage can be combined with a different semi-stream join algorithm, HYBRIDJOIN (Hybrid Join) [12]. We call the resulting combined algorithm CHYBRIDJOIN (Cached Hybrid Join) [28]. HYBRIDJOIN differs from MESHJOIN in that it addresses specifically one-to-many joins and accesses the disk-based relation R selectively. For that, an index is needed, however a non-clustered index is sufficient if we consider equijoins on a foreign key element that is stored in the stream. The CHYBRIDJOIN algorithm again possesses two complementary phases: the front-stage phase and the HYBRIDJOIN phase.

The execution architecture for CHYBRIDJOIN is shown in Figure 2. The HYBRIDJOIN phase requires most of the memory. The components for the HYBRIDJOIN phase are similar to those for the MESHJOIN phase in CMESHJOIN. Similar to CMESHJOIN, CHYBRIDJOIN alternates between the front-stage and HYBRIDJOIN phases. The working of the front-stage has already been explained.

In the HYBRIDJOIN phase of CHYBRIDJOIN, the oldest tuple in the queue is used to determine the partition of R that is loaded into the disk buffer, with the help of the non-clustered index. A few disk pages of R have to be loaded at the same time in order to amortize the costly seek time of disk access. After one probe step, a sufficient number of stream tuples in H_S are matched with the disk buffer. These tuples are deleted; for that purpose we choose a queue implementation that supports this process, e.g. a doubly-linked-list. One execution of the front-stage with a subsequent execution of the HYBRIDJOIN phase constitutes one outer iteration of CHYBRIDJOIN.

3.2.1. Algorithm

We now explain CHYBRIDJOIN as shown in Algorithm 1. The algorithm takes stream S and disk-based relation R as inputs, and as an output it produces the join between these two inputs. The main parameters for each iteration of the algorithm are the following: w_F denotes the number of stream tuples

matching in the front-stage phase in one iteration; w_B denotes the number of stream tuples matching in the HYBRIDJOIN phase in one iteration, and therefore the number of slots that are emptied in H_S. At the start of the algorithm all slots in H_S are empty; therefore w_B is initialized with h_S (line 1). The outer loop of the algorithm is an endless loop, which is natural in stream processing algorithms (line 2). The body of the outer loop has two main parts: the front-stage phase and the HYBRIDJOIN phase. Due to the endless loop, these two phases alternate. Before starting the execution of these two phases the algorithm first executes the load shedding module (line 3), which is explained in Section 4.

First, CHYBRIDJOIN executes the front-stage phase (line 4) which is presented in Algorithm 3. The front-stage phase has to know the number of empty slots in H_S, kept in variable w_B. Before starting its actual execution, the front-stage algorithm first resets w_F to zero (line 1). The front stage processes one stream tuple at a time in a loop until either the stream buffer SB_1 is emptied, or the number of unprocessed tuples, which is the *inputSize* for the next phase, has reached w_B (line 2). In each iteration of the loop the algorithm reads a stream tuple t from SB_1 (line 3) and performs a look-up of t in H_R (line 4). In the case of a match, the algorithm generates the join output and increments w_F by one (line 5 and 6). Where t does not match in H_R, the algorithm adds t to SB_2 and increments variable *inputSize* by one (lines 8 and 9).

Lines 5 to 24 of Algorithm 1 describe the HYBRIDJOIN phase. At the start of this phase, the algorithm reads w_B tuples from SB_2 and loads them into H_S while placing their key attribute values (called *pointers*) into the queue (line 5). After reading the input the algorithm resets w_B to zero (line 6). The algorithm then reads the oldest key attribute value g from the end of Q and finds the relative index value in R. If the relative index value does not exist it means this key attribute value does not have a matching tuple in the master data; the algorithm deletes g from the queue with the related stream tuple from H_S and jumps back to line 7 (lines 7 to 10). The algorithm then loads b tuples of R into the disk buffer using g as an index (line 12). In an inner loop, the algorithm looks up one-by-one all tuples r from the disk buffer in H_S. In the case of a match, the algorithm generates the join output (lines 13 to 15). Since H_S is a multi-hash-map, there can be more than one match; the number of matches is f (line 16). The algorithm removes all matching tuples from H_S while also deleting the corresponding pointers from Q (line 17). This creates empty slots in H_S which are accumulated in w_B, denoting the size of the next input (line 18).

Lines 19 to 22 are concerned with frequency detection of stream tuples which means the total number of matching values f in the whole queue against each master data tuple r in the disk buffer. In line 19 the algorithm tests whether this f is larger than a pre-set threshold. If it is, then tuple r needs to be added to the cache H_R. The algorithm overwrites an existing least-frequent tuple in H_R using the frequency recorder (line 20). We assume that empty cache slots are initialized with frequency 0. After inserting r into H_R, the frequency detection process updates the frequency recorder with the new join attribute value from r and its frequency f (line 21). The frequency recorder is a two dimensional list containing join attribute values and their frequencies for all elements in the cache. The threshold is a flexible barrier which can move up and down. If the cache is not full, this means the threshold is too high; in this case, the threshold can be lowered automatically. Similarly, the threshold can be raised if

Algorithm 1 CHYBRIDJOIN algorithm

Input: A disk based relation R with index on join attribute and a stream of updates S.

Output: $R \bowtie S$

Parameters: w_F and w_B tuples of S and k number of pages of R.

Method:

1: $w_B \leftarrow h_S$
2: **while** (true) **do**
3: Load-Shedding(w_B, SB_1, Q, H_S)
4: Front-Stage(w_F, w_B, SB_1, SB_2)
5: LOAD w_B tuples from SB_2 to H_S and also place their pointers (join attribute values) into Q
6: RESET w_B to 0
7: READ the oldest join attribute value g from Q
8: **if** g does not match with index value in R **then**
9: DELETE g from Q
10: Go to line 7
11: **end if**
12: READ b tuples of R into the disk buffer using g as an index look-up
13: **for** each tuple r in disk buffer **do**
14: **if** $r \in H_S$ **then**
15: OUTPUT $r \bowtie H_S$
16: $f \leftarrow$ number of matching tuples found in H_S
17: DELETE all matched tuples from H_S along with the corresponding nodes from Q
18: $w_B \leftarrow w_B + f$
19: **if** $f \geq thresholdValue$ **then**
20: OVERWRITE least frequent tuple in H_R with r using frequency recorder
21: UPDATE frequency recorder with new join attribute value from r and its frequency f
22: **end if**
23: **end if**
24: **end for**
25: **end while**

Algorithm 2 Load-Shedding(w_B, SB_1, Q, H_S)

1: **if** $sizeof(SB_1) > 2w_B$ **then**
2: SHED $(sizeof(SB_1) - 2w_B)$ *pointers* from end of Q along with their corresponding tuples from H_S on disk/network
3: **else**
4: LOAD $2w_B$ tuples from disk/network to SB_1
5: **end if**

Algorithm 3 Front-Stage(w_F, w_B, SB_1, SB_2)

```
1: RESET w_F to 0
2: while inputSize < w_B AND SB_1 is not empty do
3:    READ stream tuple t from SB_1
4:    if t ∈ H_R then
5:       OUTPUT t
6:       w_F ← w_F+1
7:    else
8:       ADD stream tuple t in SB_2
9:       inputSize ← inputSize+1
10:   end if
11: end while
```

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.

3.3. Cached Index Nested Loop Join (CINLJ)

Finally, we consider a combination of the proposed front-stage with the classic Index Nested Loop Join (INLJ) [13]. This setup is not intended as a recommended algorithm for practical purposes, but establishes a baseline for the effects of the front-stage alone. The resulting algorithm is called CINLJ (Cached Index Nested Loop Join). The execution architecture of CINLJ is shown in Figure 3. The front-stage phase is essentially the same component as before. In the INLJ phase, the algorithm processes one stream tuple at a time. For each stream tuple the algorithm queries the disk-based master data using a join attribute value as an index. If a relevant tuple is found, the algorithm performs the join and generates an output.

In order to use our front stage with the INLJ, we need to determine the frequent tuples in the stream. Since the INLJ itself does not accumulate enough information, we need an extra component, a small buffer in the INLJ phase that is labeled "temporary buffer" in the figure. This buffer stores the join attribute values from stream data for a certain time span, and after a certain time span the buffer is refreshed. This buffer is much smaller than the queue plus hash table of the preceding algorithms, and needs to store the join attribute values only long enough so that the most frequent ones can be detected. In general, if the frequency of a join attribute value is greater than the specified threshold value, then the frequency manipulation process puts that tuple into the front-stage cache. If the desire is to make the buffer as small as possible, this frequency can be set to 2, and the life span of the buffer content can be set so that the most frequent tuples are expected to appear with frequency 2.

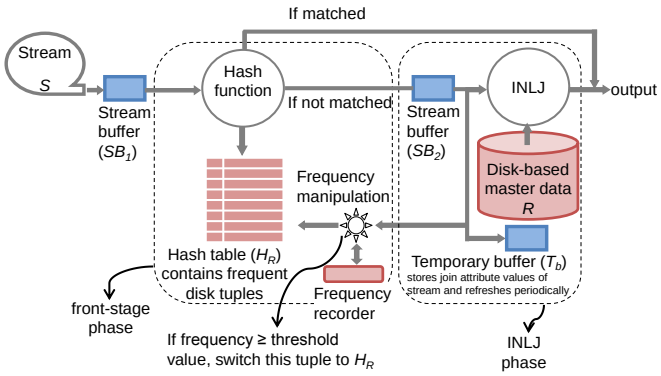


Figure 3: Data structure and architecture of CINLJ

4. Load Shedding

While caching is a useful performance optimisation for semi-stream joins, there are situations where a join is unable to process the incoming stream tuples at the rate they arrive. To deal with such situations, a mechanism for load shedding is necessary, i.e. a mechanism that is able to selectively move arriving tuples to storage for later processing. A load shedding technique should be as efficient as possible on its own, with further improvements possible by adding a caching technique as discussed in the previous sections. In the following, we propose a novel and efficient technique for load shedding in semi-stream joins which is particularly suited for skewed distributions.

Load shedding has been discussed for approximate stream join processing [29, 30, 31], but in these approaches all inputs of the join operator are streams. For semi-stream joins, load shedding has been first discussed in [11]. There the authors discuss two different heuristics (goals) for load shedding in *many-to-one* joins, namely *maximizing the subset* and *random sample*. The random sample goal is that the produced tuples are a random sample of the join output without load shedding. For this goal one can hardly do any better than choosing a random subset, except with very strong assumptions. However, for the other goal, maximizing the subset, we propose a novel improved solution. From the brief description in [11], and due to the fact that the sources are not public, it can be inferred that their "keep" strategy sheds stream tuples *upfront*, i.e. before processing them in a substantial way, which is the standard approach to load shedding. This results in effect to dropping every stream tuple with the same probability. By contrast, we present a load shedding approach which preferably sheds tuples from the stream that are more expensive to process. This is further explained in Section 4.2. As a result, our algorithm yields a significantly higher service rate under shedding than without shedding, as shown in experiments in Section 5.3.

4.1. Problem overview

In algorithms without load shedding, the assumption is that the join operator has been allocated enough memory to cope with the stream arrival rate. However, the converse scenario is of interest, where the stream arrival rate can exceed the service rate of the join operator, for example because the memory is fixed. This is explained in Figure 4. The figure presents both the stream arrival rate and the service rate. From the figure it can be observed that after a certain point (we call it shedding point, at which

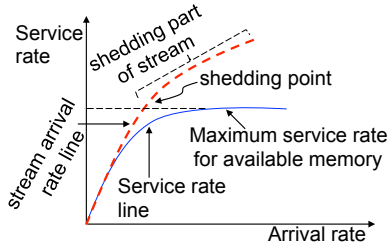


Figure 4: Interpretation of stream load shedding

the service rate is optimal within assigned memory limit) due to memory constraint the service rate cannot withstand the stream arrival rate. So under this memory settings the algorithm is unable to process the stream after the shedding point. A load shedding option should automatically detect high load, and perform load shedding only during said high load. The approach to detect high load is based on the iterative nature of the algorithm. From previous executions of the main join loop, we obtain an expected number of tuples w processed in each iteration. In the simplest case, which is used in the current implementation, this is simply the number of tuples processed in the last iteration. If the current size of the stream buffer is larger than w, then the stream buffer does not get emptied, and tuples might start to accumulate. If the current size of the stream buffer is larger than $2w$, we assume that the buffer is accumulating tuples, and we perform a load shedding operation. In this way, load shedding decisions are made within the timeframe of one loop execution, and the algorithm can react quickly to varying load.

Algorithm 2 describes the load shedding module. The load shedding algorithm first compares the size of SB_1 with w_B (line 1). If the number of tuples in SB_1 is more than twice w_B, load shedding is required. In this case the algorithm sheds $(sizeof(SB_1) - 2w_B)$ pointers from the end of Q along with their corresponding tuples from H_S onto disk/network (line 2). On the other hand if the size of SB_1 is less than twice w_B, this means the stream is slower than the service rate. In this case the algorithm loads $2w_B$ number of tuples from disk/network to SB_1 (line 3 and 4).

4.2. Load shedding after considering tuples

The load shedding approach that we implemented in our CHYBRIDJOIN algorithm sheds the tuples that are most expensive in terms of resource consumption, yielding a higher service rate as the "keep" approach of MESHJOIN. This is possible by dropping tuples only after *considering* them for processing, which is achieved by shedding tuples from the end of the queue. Assume the detection of high load has decided that w tuples have to be shed. Instead of shedding w tuples upfront from the stream buffer, our approach, which we call "considering", sheds w tuples from the end of the queue, thus freeing an identical amount of space.

We show that "considering" is better than the upfront "keep" approach by discussing the processing cost of stream tuples. For this we have to consider for each stream tuple the master data partition that it matches. We can define the *stream probability* [32] of a master data partition, as the probability that a random stream tuple matches a master data record on that partition. We first observe that if stream tuple foreign keys follow a Zipfian random distribution, then there are the following options for the distribution of the stream probability. If the rank of the master data tuples is distributed randomly over the master

data, then the stream probability will be normally distributed according to the central limit theorem of probability theory. This means, there are still more and less frequent master data partitions, but the differences are by far not as extreme as the skew in the foreign keys, but in all data sets considered, it is noticeable.

In reality, we expect locality of master data rank, i.e. frequent items will have other frequent items on the same page (e.g. if all "'bread"' product codes are on the same page). Then we can expect again a Zipfian distribution for the stream probability. However, for the sake of generality we do not make strong assumptions about this distribution. Given a strategy for using queue tuples to look up and load master data partitions (we call this a *lookup strategy*), we define the *load probability* [32] of a master data partition as the probability that this partition is loaded during the HYBRIDJOIN phase. Note that this probability is dependent on the workings of the algorithm; however, the algorithm is for a given stream probability. For any lookup strategy that uses a fixed or random position in the queue to load the corresponding master data partition, the load probability of a partition increases with the stream probability, and this is a desirable property. These concepts can be turned into a property of tuples: for a tuple we can consider the stream probability and load probability of its master data partition. We consider $P_s(n)$, the expected stream probability of the master data partition for the tuple at position n in the queue (counting from the start of the queue).

Lemma 1 *$P_s(n)$ decreases with increasing n.*

Proof. Let t be the tuple currently at position n. The higher n, the higher the expected number of master data lookups that have been performed while t is in the queue. Every master data lookup is more likely to load a partition with high load probability, and will remove all matching tuples from the queue. Therefore tuples such as t that remain after these lookups, have themselves a lower expected load probability and hence also a lower expected stream probability. □

Theorem 1 *Shedding after "considering" yields a higher service rate than shedding upfront.*

Proof. The bottleneck of CHYBRIDJOIN is the master data disk access. If we increase the average number of matches per loaded master data partition then we increase the service rate of the algorithm. Shedding after considering means we shed tuples from the end of the queue. It remains to show that this increases the average number of matches per master data disk access. We can conclude from Lemma 1 that the average stream probability of a group of tuples at the end of the queue is lower than the average stream probability of tuples before entering the queue. This means the strategy indeed picks tuples with lower stream probability than the strategy of shedding upfront. Lower stream probability of a tuple however means that if this tuple needs to be processed and its master data partition has to be loaded, there are fewer other tuples that will match at the same time. This means the tuples picked by shedding after considering are indeed more expensive than average, so shedding them increases the service rate. □

The result is at first glance somewhat paradoxical since one would assume that the process of considering, i.e. having the tuples sit in the queue, incurs some additional cost. But with the alternative

Table 1: Data specifications for synthetic dataset

Parameter	**value**
Total allocated memory M	1% of R (0.11GB) to 10% of R (1.12GB)
Size of disk-based relation R	100 million tuples (≈11.18GB)
Size of each disk tuple	120 *bytes* (same as MESHJOIN)
Size of each stream tuple	20 *bytes* (same as MESHJOIN)
Size of each node in the queue	4 *bytes* in CMESHJOIN while 12 *bytes* in CHYBRIDJOIN
Data set	based on Zipf's law (skew value from 0 to 1)
Stream arrival rate (λ)	varies

strategy of shedding tuples upfront, about the same fraction of tuples sitting in the queue would be low-frequency tuples. The difference to our strategy is that we do not waste the most important resource (loading master data partitions) on these tuples.

5. Experiments

In this section we present an extensive experimental study of our approaches using synthetic, TPC-H, and real-life data. We divide our experiments into three different subsections in the text. In Section 5.2, we present the comparisons of service rate and processing time using front-stage cache module. In Section 5.3, we present the comparisons of service rate under load shedding. Finally, in Section 5.4 we perform sensitivity analysis with respect to various internal design parameters. In this category of experiments we only consider the CMESHJOIN approach. This subsection also validates the calculated costs with measured costs for all algorithms, considered in this article. Before proceeding to the experimental results we first describe the setup in which we carried out our experiments.

5.1. Experimental setup

Hardware and software specifications: We performed our experiments on a *Pentium-i5* with 8GB main memory and 500GB hard drive as secondary storage. We implemented our experiments in Java using the Eclipse IDE. As join attribute values can be duplicated in stream data against one value of the master data due to the attribute being a foreign key, a hash table is needed that can store multiple values against one value of the master data. The hash table provided by the Java library does not support this feature, therefore `org.apache.MultiHashMap` was used.

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. These kinds of algorithms normally need a warmup phase to tune their components with respect to the available memory resources, so that each component can deliver a maximum service rate. For each measurement where necessary, we calculated the 95% confidence interval. During the execution of the algorithm, no other application was running in parallel.

Data specifications: The relation R was stored on disk using a MySQL database. Both the algorithms read master data from the database. To measure the I/O cost more accurately, we set the fetch size for `ResultSet` equal to the disk buffer size.

Synthetic data: The stream dataset we used is based on a Zipfian distribution. The master data we used was unsorted. The detailed specifications of our synthetic dataset are shown in Table 1.

TPC-H: We also analyzed the service rate of both algorithms using the TPC-H dataset, which is a well-known decision support benchmark. We created the datasets using a scale factor of 100. More precisely, we used the table `Customer` as master data and the table `Order` as stream data. In table `Order` there is one foreign key attribute `custkey`, which is a primary key in the `Customer` table, so the two tables can be joined. Our `Customer` table contained 20 million tuples, with each tuple having a size of 223 bytes. The `Order` table contained the same number of tuples, with each tuple having a size of 138 bytes. The plausible scenario for such a join is to add customer details corresponding to an order before loading the order into the warehouse.

Real-life data: We also compared the service rate of both algorithms using a real-life dataset[1]. This dataset contains cloud information stored in a summarized weather report format. It was also used to evaluate the original MESHJOIN. We use one part of the table as master data, and transform another part of the table into a continuous stream of data. The tuple size is therefore the same in the master data table and the stream data, 128 bytes. The master data table contains 20 million tuples. Both tables are joined on the longitude (`LON`) attribute. The domain of the join attribute is the interval [0,36000].

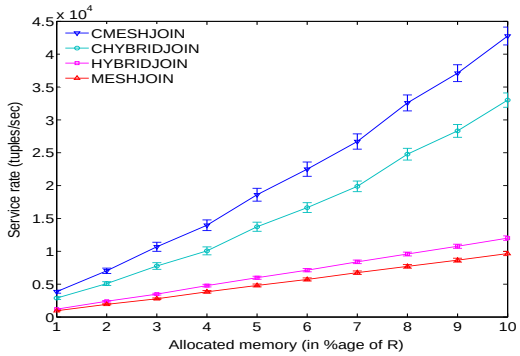
5.2. Experimental evaluation of caching

5.2.1. Service rate analysis

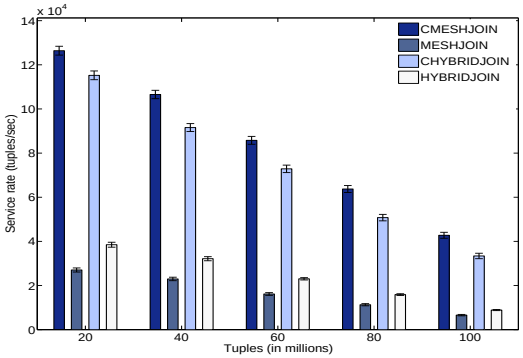
We investigate the effect of three parameters on the behavior of the algorithms: the size of the master data table R, the total memory available, and the value of skew in the foreign keys of stream data. For the sake of brevity, we restrict the discussion for each parameter to a one-dimensional variation, i.e. we vary one parameter at a time. Note that we do not plot the service rate of INLJ and CINLJ in the figures as they were invisible due to their small values. However, where necessary we present these service rate values in the text. First we evaluate the performance of all the algorithms by varying the three parameters using synthetic data. Later we also evaluate the performance of all the algorithms using TPC-H and real-life data. The details about all these datasets have already been presented in Section 5.1.

Analysis by varying size of memory: In this experiment we compared the service rate of the algorithms while varying the memory size from 1% to 10% of R, with the size of R being 100 million tuples (≈11.18GB) and skew value 1. For each memory setting we measure the service rate of each algorithm. The results of our experiment are shown in Figure 5(a). We can see that CMESHJOIN performed up to 6.5 times faster than MESHJOIN and CHYBRIDJOIN performs about 2.8 times better than HYBRIDJOIN with the 10% memory setting. With less memory (1% of R), CMESHJOIN still performed up to 4 times better than MESHJOIN and CHYBRIDJOIN performs up to 2.4 times better than HYBRIDJOIN, which makes the new algorithms good adaptive solutions suitable for memory-constrained applications. We did

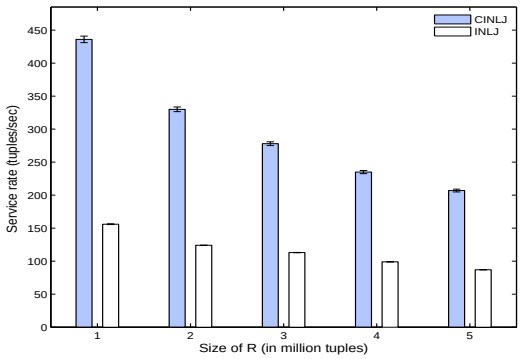
[1]This dataset is available at: `http://cdiac.ornl.gov/ftp/ndp026b/`



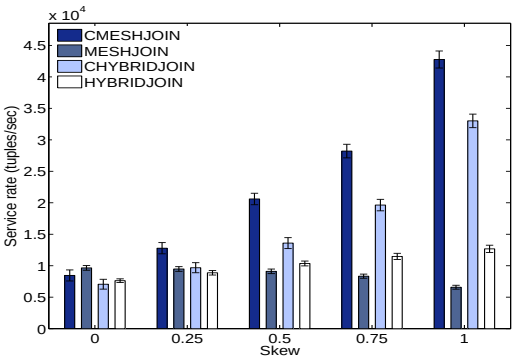
(a) Service rate vs memory



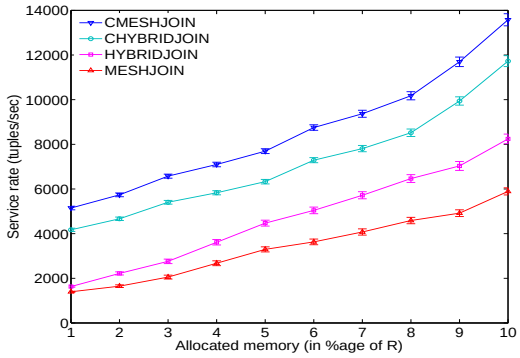
(b) Service rate vs size of R



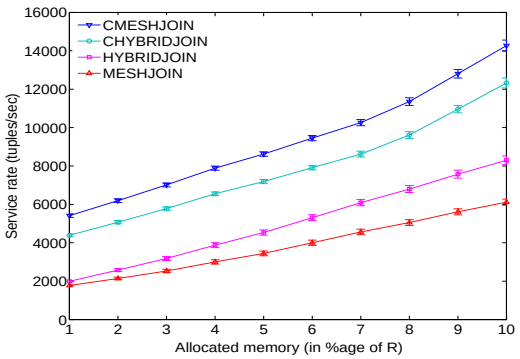
(c) Service rate vs size of R (INLJ vs CINLJ)



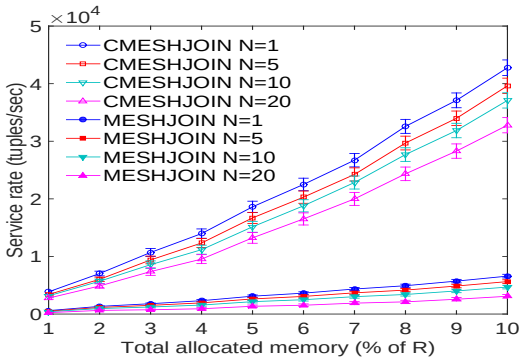
(d) Service rate vs skew



(e) TPC-H dataset



(f) Real-life dataset



(g) Temporal locality

Figure 5: Service rate analysis without load shedding

not include INLJ in this comparison as memory does not matter in its service rate. However, we believe that CINLJ will significantly outperform INLJ for all the memory settings.

Analysis by varying size of *R*: In this experiment we compared the service rate of all the algorithms for different sizes of R, with a fixed memory size (≈1.12GB). We also fixed the skew value to 1 for all settings of R. The results for all the algorithms except INLJ and CINLJ are shown in Figure 5(b). From Figure 5(b) we see that for R with 20 million tuples CMESHJOIN performed up to 4.7 times better than MESHJOIN while CHYBRIDJOIN performs about 3 times better than HYBRIDJOIN. This improvement increased to 6.5 times for CMESHJOIN when the size of R was 100 million tuples. For the same setting CHYBRIDJOIN is still significantly better than HYBRIDJOIN, but the margin of improvement slightly decreases (to 2.8 times) because an index is used to access R. In case of INLJ and CINLJ the service rate is very low as is to be expected for an unclustered index. It is still important to see the impact of the frontstage in this case, therefore we provide a separate graph in Figure 5(c). We see that CINLJ performs up to 2.5 times better than INLJ. However, as expected, these two perform substantially worse than the other four algorithms. The reason for their poor service rate is the high I/O cost of the INLJ phase; it does not amortize the expensive I/O cost.

Analysis by varying skew value: In this experiment we compared the service rate of the algorithms while varying the skew of the streaming data, expressed as the Zipfian exponent, from 0 to 1. At exponent 0 the input stream S is uniform, while exponent 1 expresses a relatively 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 5(d) show that CMESHJOIN and CHYBRIDJOIN perform significantly better than MESHJOIN and HYBRIDJOIN respectively, even for only moderately skewed data, and this becomes more pronounced for higher skew. At a skew of 1, CMESHJOIN performs approximately 6.5 times better than MESHJOIN and CHYBRIDJOIN performs 2.8 times better than HYBRIDJOIN. 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 assume that for such short tails the trend continues. Also from the figure we can see that CMESHJOIN and CHYBRIDJOIN perform worse than MESHJOIN and HYBRIDJOIN respectively when stream data is completely uniform (exponent of 0). The reason is that the front-stage does not add any value in case of fully uniform distribution. However, these differences are relatively small and represent the worst case.

TPC-H and real-life datasets: In these experiments we measured the service rate produced by the above four algorithms at different memory settings. The results of using TPC-H data and real-life data are shown in Figure 5(e) and Figure 5(f) respectively. From Figure 5(e) it can be noted that CMESHJOIN performed about 3.7 times better than MESHJOIN and CHYBRIDJOIN performed about 2.5 times better than HYBRIDJOIN, which is significant especially for a smaller memory size, 1% of R. Similarly, it is obvious from Figure 5(f) that both CMESHJOIN and CHYBRIDJOIN outperform MESHJOIN and HYBRIDJOIN under all memory settings.

Temporal locality: We considered an important characteristic of stream data that is called temporal locality. This is the case if within a certain time period most of the join attribute values in stream data

fall within a subset of the master data domain. In this experiment, we generated data simulating this scenario as follows. We considered 100 million tuples in the master data R, uniformly distributed within the domain of the join attribute, which is $[1, \frac{R}{N}]$. N is an arbitrary number and is used to set the domain value. Now each value in R appears exactly N times. The stream data contained 50 million tuples and was divided into 25 equal-size intervals, with each interval containing 2 million tuples. Each interval I, $I \in [1, 25]$, contained a Zipfian distribution with skew 1 and a domain of $[I \times \frac{R}{25N}, (I+1) \times \frac{R}{25N}]$. Hence, each stream interval required localized reads from a limited number of blocks of R (e.g. in the case of $N = 10$ the domain of each stream interval would be 0.4 million, which is 4% of the domain of R).

The results of this experiment under different memory settings are depicted in Figure 5(g). We consider in this experiment only two algorithms, CMESHJOIN and MESHJOIN. From the figure it can be observed that for N=1 CMESHJOIN performed 6.5 times better than MESHJOIN, while for N=20 it was about 10 times better than MESHJOIN. The service rate in CMESHJOIN also decreased with an increase in the value of N because of the MESHJOIN phase, but it decreased significantly slower than for MESHJOIN. This is because the front-stage stored the frequent tuples of R.

5.2.2. Processing time analysis

A second kind of performance parameter besides service rate refers to the time an algorithm takes to process a tuple. *Processing time* is an average time that every stream tuple spends in the join module from loading to matching without including any delay due to a low arrival rate of the stream. Figure 6 shows a comparison of the processing times. We can see that the processing time in both CMESHJOIN and CHYBRIDJOIN is significantly smaller than those of MESHJOIN and HYBRIDJOIN respectively. This difference became larger particularly in the case of CMESHJOIN as we increased the size of R. The plausible explanation for this is that in both CMESHJOIN and CHYBRIDJOIN a big part of the stream data is directly processed through the front-stage phase without joining it with the whole relation R in memory. We do not present the processing time for CINLJ and INLJ because it does not change significantly, even when the size of R changes. In the cases of CINLJ and INLJ, since the algorithms work at tuple level, a delay appears in the form of a stream backlog that occurs due to faster incoming stream rate than the processing rate.

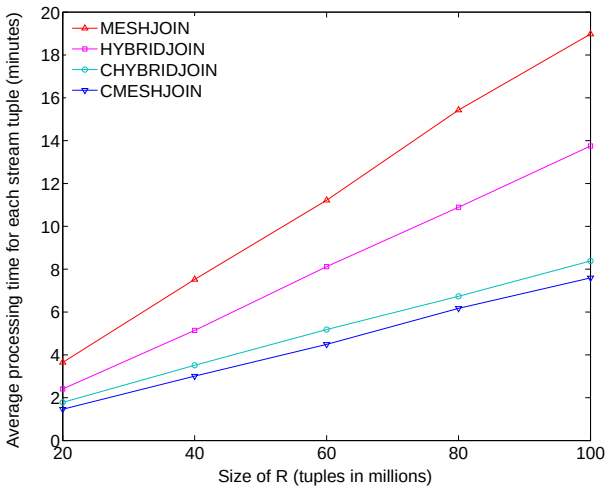


Figure 6: Processing time analysis

5.3. Experimental evaluation of load Shedding

5.3.1. Choosing an optimal position to access the queue at the time of load shedding

Since in our load shedding approach the last queue elements are shedded, it is intuitively clear that one should not use the last element as a lookup element, i.e. the stream tuple that determines which

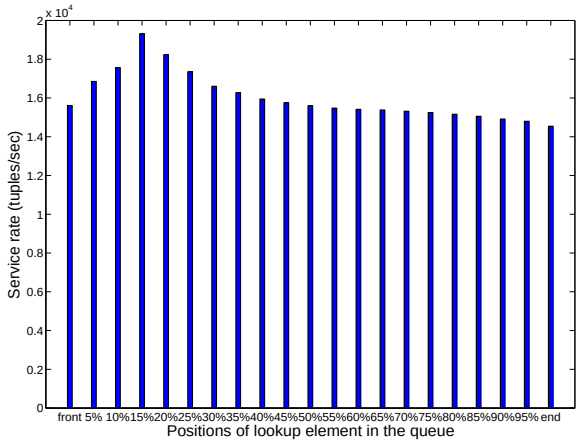


Figure 7: Queue optimization for load shedding

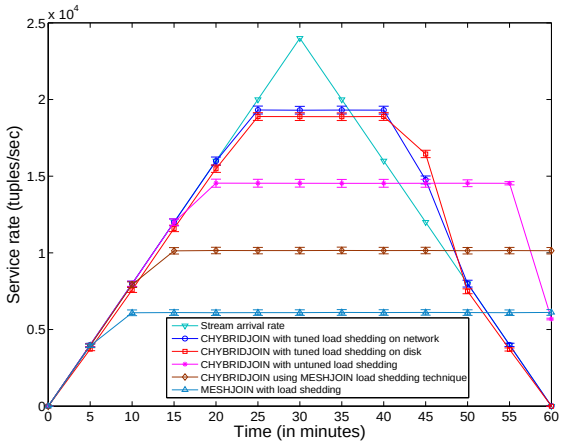


Figure 8: Service rate analysis under load shedding

master data tuple is loaded next. So the question arises which position in the queue the algorithm should choose as the lookup element.

This section explains our quantitative measurements for choosing an optimal position for accessing the queue while load shedding is on. The details about the dataset we used here have been presented in Section 5.1. We conducted an experiment in which we measured the performance of the algorithm by accessing the queue at different positions in steps of 5% of the queue length.

The results of our experiment are shown in Figure 7. As we move the access position away from the front of the queue the performance is improving. This observation matches our qualitative expectation from Theorem 1. This behavior continues up to a certain position (which is at 15% in our experiment) and after that maximum, performance decreases towards the end of the queue. The reason for this behaviour is stated in Theorem 1. As a result of this experiment we can say that position 15% is optimal and we use this position later in all our performance evaluation experiments.

5.3.2. Service rate analysis

We now analyze the service rate produced by CHYBRIDJOIN under load shedding, including a comparison of the service rate using the optimal lookup position with the end lookup position. In the case of optimal lookup the algorithm accesses the queue at a position of 15% while in the case of non-optimal lookup the algorithm accesses the queue from the end. Therefore, we call shedding using the former case *tuned load shedding*, while shedding using the latter case is called *untuned load shedding*. In the case of tuned load shedding, we presented the results for shedding on disk as well as on network.

Figure 8 presents the results of our experiment. To visualize the effect clearly we also plotted the stream arrival rate in the figure. From the figure it is clear that CHYBRIDJOIN with tuned load shedding performs 33% better than that of untuned load shedding, which is a significant improvement. The reason for this better performance has already been described in the above theorem. From the figure we can also observe a slight improvement in service rate in the case when shedding has taken place on the network. The other positive outcome that we can extract from the experiment is that the algorithm does not lose much performance if we shed the stream data onto the disk.

To illustrate the effect of our load shedding approach on service rate, we also implemented the

MESHJOIN load shedding approach in CHYBRIDJOIN and compared the service rate of both approaches. From the figure it can be observed that using our load shedding approach CHYBRIDJOIN performs 3.1 times better than with the MESHJOIN load shedding approach. The reason for this improvement is that in the case of our load shedding the algorithm sheds the stream tuples from the end of the queue, while these tuples are infrequent in the stream. Contrarily, in the case of MESHJOIN load shedding the algorithm sheds the stream tuples from the stream buffer (before processing them) without caring of frequent and infrequent stream tuples. For the sake of completeness we also included MESHJOIN in this experiment. From the figure, MESHJOIN has a very low service rate as the algorithm does not implement tuned load shedding. Apart from that the algorithm also does not have a front-stage.

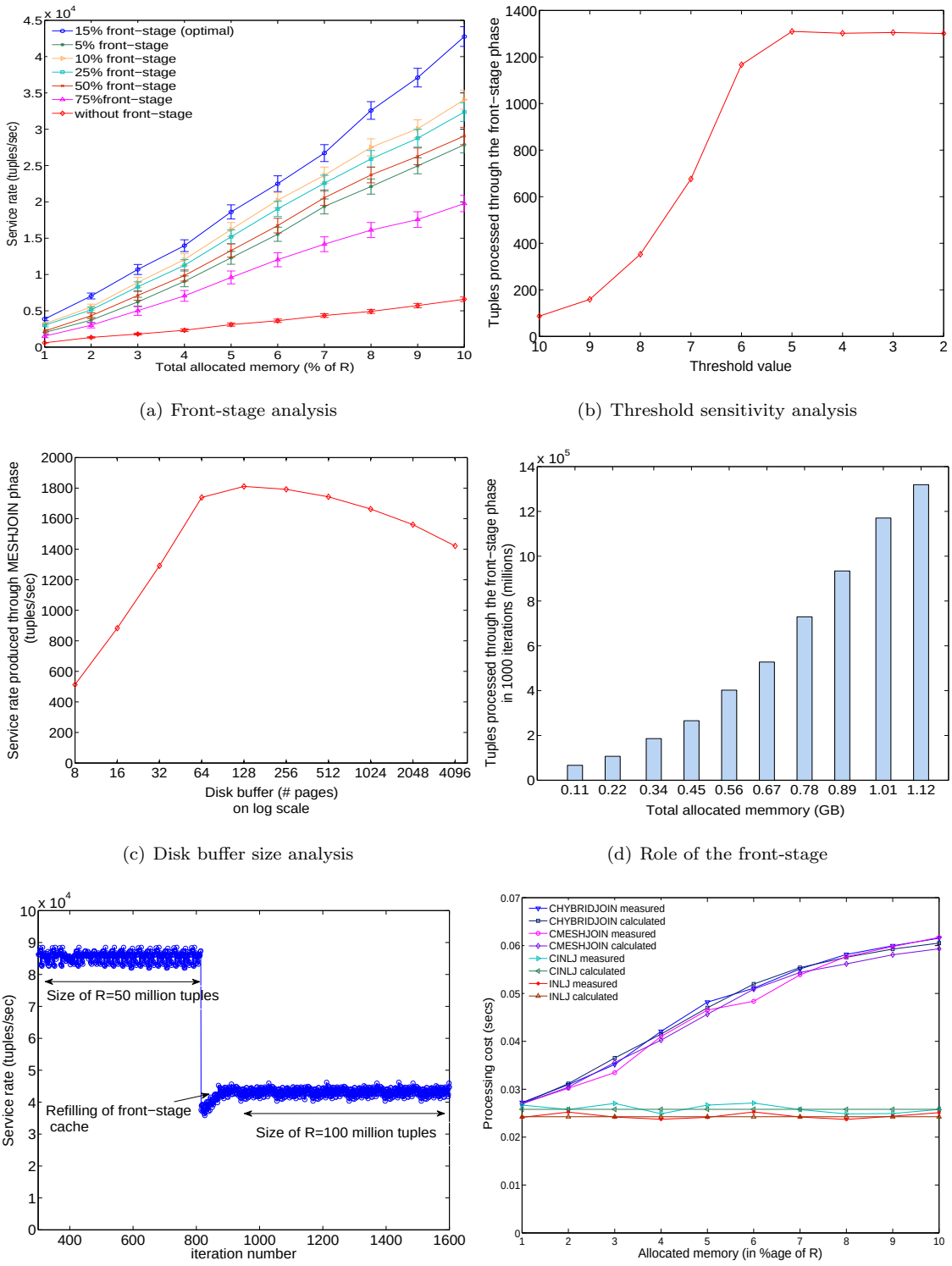
In summary, CHYBRIDJOIN with tuned load shedding reaches a higher service rate than that of the MESHJOIN load shedding approach. This means we have utilized the opportunity of load shedding to focus on those tuples where fast progress is possible.

5.4. Analysis of algorithm design parameters

Design parameters are parameters that can be tuned. In this section we consider only CMESHJOIN and analyze the sensitivity of its design parameters. However, a similar analysis can be performed for the other algorithms' parameters.

Sensitivity to front-stage size: In this experiment we analyzed the effect of the front-stage size on the service rate of CMESHJOIN. We ran the algorithm with different sizes of the front-stage and measured the service rate of the algorithm. We also tested the algorithm without the front-stage; in this case CMESHJOIN essentially operates like MESHJOIN. In this experiment we used an R with size of 100 million tuples (≈11.18GB). The results are shown in Figure 9(a). From the figure it can be observed that if we increase the memory for the front-stage, the service rate continuously improves up to a maximum at ≈15% and after that the service rate again starts decreasing. It is noteworthy that with a 75% cache size the algorithm produces a service rate still greater than that of not using the front-stage at all. This behavior is consistent for a wide range of total memory settings and can be understood as a trade-off between the front-stage phase and the MESHJOIN phase.

Threshold sensitivity: The threshold is a parameter of CMESHJOIN used by the frequency manipulator to decide whether a tuple is frequent or not. In this experiment we observe the sensitivity of the frontstage behavior to this parameter. We therefore measured the service rate produced through the front-stage phase only. The natural expectation is that for lower threshold values this service rate becomes bigger, since that with a high threshold value the cache does not fill up completely, and therefore the algorithm cannot fully exploit the front-stage phase. We used an R with a size of 100 million tuples (≈11.18GB) and allocated memory the size of 10% of R (≈1.12GB). The stream followed a Zipfian distribution with skew value of 1. The results are shown in Figure 9(b). For threshold values decreasing from 10 the service rate increases first rapidly as expected, but only down to a certain threshold. The plausible reason for this is that once the cache fills up completely, the frequency manipulator replaces the least-frequent tuple in the cache with a new one, which does not make a significant difference in the service rate. Normally, CMESHJOIN fills the cache in a warmup phase by adjusting the threshold value



(a) Front-stage analysis

(b) Threshold sensitivity analysis

(c) Disk buffer size analysis

(d) Role of the front-stage

(e) Changing the size of R online

(f) Cost validation

Figure 9: Sensitivity analysis

dynamically. Later, if the memory size or R change, the algorithm has the ability to adapt and refill the cache online.

Sensitivity to the disk buffer size: The disk buffer is a very sensitive component of the MESHJOIN phase, which influences the service rate directly. We analyzed the sensitivity of this parameter in an

experiment with a size of R of 100 million tuples (≈11.18GB), a total allocated memory of 1% of R (≈0.11GB), and a Zipfian distribution with a skew of 1. To observe the effect clearly, we measured only the service rate produced through the MESHJOIN phase. The results are shown in Figure 9(c). When increasing the size of the disk buffer, the cost of loading the disk pages into memory is amortized among a larger number of stream tuples (stored in H_S), and as a result the service rate improves significantly. This trend continues up to a certain value of the disk buffer size. After that, the service rate is affected adversely. The reason for this is that for a large disk buffer this loading cost remains almost the same. However, the increased disk buffer size reduces the available memory for the hash table H_S, so fewer stream tuples are accommodated in memory.

Role of the front-stage: To get a better understanding of the role of the front-stage, we performed an experiment where we counted the stream tuples processed only through the front-stage phase. The results are shown in Figure 9(d). It is clear from the figure that in 1000 iterations, when the size of R is ≈11.18GB and the total allocated memory is ≈1.12GB (10% of R) while skew in input distribution is 1, about 51% of the total stream is processed through the front-stage phase.

Varying R online: CMESHJOIN supports changing the master data online. To verify the argument we conducted an experiment, as shown in Figure 9(e), with a fixed size of allocated memory (≈1.12GB) and the stream following a Zipfian distribution with a skew of 1. Initially, the algorithm ran with a master data size of 50 million tuples. After that, we appended an equal amount of master data while the algorithm was running. The results presented in the figure show that the behavior of the algorithm is completely normal. It does not lose any additional service rate apart from the expected losses: initially due to refilling the cache of the front-stage with respect to new R and due to doubling the size of R.

Costs validation: The cost models for all the algorithms have been validated by comparing the calculated cost with the measured cost. Figure 9(f) presents the comparisons of both costs for each algorithm. The results show that for each algorithm the calculated cost closely resembles the measured cost, which supports the correctness of our implementations.

6. Conclusions

In this paper we discussed two important contributions towards a robust semi-stream join that uses limited memory, is tolerant against high load and utilizes aspects of common distributions in the input data. The first contribution is a generic front stage component for caching. This component has a small memory footprint but provides a large performance gain if the input data has a skewed distribution, as is commonly the case. We quantified the performance gain for the most widely discussed skewed distribution, a Zipfian distribution, and we demonstrated that this component is indeed generic, by combining it with several existing semi-stream joins.

We then introduced a novel approach to load shedding, which again can utilize skew in the input data and does this by exploiting the existing architecture of a successful semi-stream join, our previously proposed HYBRIDJOIN. The load-shedding approach is nonobvious as it also does some processing with the tuples that are shed. However, a theoretical argument and experimental results corroborate its efficiency. We provide open-source implementations of our algorithms that can be used for further

analysis[2].

References

[1] M. A. Naeem, G. Dobbie, G. Weber, An event-based near real-time data integration architecture, in: EDOCW '08: Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops, IEEE Computer Society, Washington, DC, USA, 2008, pp. 401–404. doi:http://dx.doi.org/10.1109/EDOCW.2008.14.

[2] A. Karakasidis, P. Vassiliadis, E. Pitoura, ETL queues for active data warehousing, in: IQIS '05: Proceedings of the 2nd International Workshop on Information Quality in Information Systems, ACM, New York, NY, USA, 2005, pp. 28–39. doi:http://doi.acm.org/10.1145/1077501.1077509.

[3] L. Golab, T. Johnson, J. S. Seidel, V. Shkapenyuk, Stream warehousing with datadepot, in: SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2009, pp. 847–854. doi:http://doi.acm.org/10.1145/1559845.1559934.

[4] A. Arasu, S. Babu, J. Widom, An abstract semantics and concrete language for continuous queries over streams and relations.

[5] E. Wu, Y. Diao, S. Rizvi, High-performance complex event processing over streams, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM, 2006, pp. 407–418.

[6] M. A. Naeem, G. Weber, G. Dobbie, C. Lutteroth, A generic front-stage for semi-stream processing, in: Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management, CIKM '13, ACM, 2013, pp. 769–774. doi:10.1145/2505515.2505734.

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

[8] A. Chakraborty, A. 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, IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–11. doi:http://dx.doi.org/10.1109/IPDPS.2009.5161064.

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

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

[2] URL: www.cs.auckland.ac.nz/research/groups/serg/j/tkde/

[11] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, N. Frantzell, Meshing streaming updates with persistent data in an active data warehouse, IEEE Trans. on Knowl. and Data Eng. 20 (7) (2008) 976–991. doi:http://dx.doi.org/10.1109/TKDE.2008.27.

[12] M. A. Naeem, G. Dobbie, G. Weber, HYBRIDJOIN for near-real-time data warehousing, International Journal of Data Warehousing and Mining (IJDWM) 7 (4). doi:10.4018/jdwm.2011100102.

[13] R. Ramakrishnan, J. Gehrke, Database management systems, Osborne/McGraw-Hill, 2000.

[14] A. N. Wilschut, P. M. G. Apers, Dataflow query execution in a parallel main-memory environment, in: PDIS '91: Proceedings of the first International Conference on Parallel and Distributed Information Systems, IEEE Computer Society Press, Los Alamitos, CA, USA, 1991, pp. 68–77.

[15] A. N. Wilschut, P. M. G. Apers, Pipelining in query execution, in: Proceedings of the International Conference on Databases, Parallel Architectures and Their Applications (PARBASE 1990), IEEE Computer Society Press, Los Alamitos, 1990, pp. 562–562.

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

[17] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, D. S. Weld, An adaptive query execution system for data integration, SIGMOD Rec. 28 (2) (1999) 299–310. doi:http://doi.acm.org/10.1145/304181.304209.

[18] M. F. Mokbel, M. Lu, W. G. Aref, Hash-Merge Join: A non-blocking join algorithm for producing fast and early join results, in: ICDE '04: Proceedings of the 20th International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 2004, p. 251.

[19] R. Lawrence, Early Hash Join: A configurable algorithm for the efficient and early production of join results, in: VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB Endowment, 2005, pp. 841–852.

[20] D. DeWitt, J. Naughton, Dynamic memory hybrid hash join, Tech. rep., University of Wisconsin (1995).

[21] S. D. Viglas, J. F. Naughton, J. Burger, Maximizing the output rate of multi-way join queries over streaming information sources, in: VLDB '2003: Proceedings of the 29th International Conference on Very large Data Bases, VLDB Endowment, 2003, pp. 285–296.

[22] A. Chakraborty, A. Singh, A disk-based, adaptive approach to memory-limited computation of windowed stream joins, in: Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part I, DEXA'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 251–260. URL http://portal.acm.org/citation.cfm?id=1881867.1881892

[23] M. A. Naeem, G. Dobbie, G. Weber, S. Alam, 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. doi:http://dx.doi.org/10.1109/IPDPS.2009.5161064.

[24] M. H. Bateni, L. Golab, M. T. Hajiaghayi, H. Karloff, Scheduling to minimize staleness and stretch in real-time data warehouses, in: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '09, ACM, New York, NY, USA, 2009, pp. 29–38. doi:http://doi.acm.org/10.1145/1583991.1583998.
URL http://doi.acm.org/10.1145/1583991.1583998

[25] L. Golab, T. Johnson, V. Shkapenyuk, Scheduling updates in a real-time stream warehouse, in: ICDE 2009: Proceedings of the 25th International Conference on Data Engineering, 2009, pp. 1207–1210. doi:10.1109/ICDE.2009.202.

[26] L. Golab, T. Johnson, Consistency in a stream warehouse, in: Conference on Innovative Data Systems Research (CIDR11), 2011, pp. 114–122.

[27] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, M. A. Shah, TelegraphCQ: continuous dataflow processing, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03, ACM, New York, NY, USA, 2003, pp. 668–668. doi:http://doi.acm.org/10.1145/872757.872857.
URL http://doi.acm.org/10.1145/872757.872857

[28] M. A. Naeem, G. Dobbie, G. Weber, A lightweight stream-based join with limited resource consumption, in: DaWaK '12: Data Warehousing and Knowledge Discovery, Springer, 2012, pp. 431–442.

[29] A. Das, J. Gehrke, M. Riedewald, Approximate join processing over data streams, in: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, ACM, 2003, pp. 40–51.

[30] U. Srivastava, J. Widom, Memory-limited execution of windowed stream joins, in: Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, VLDB Endowment, 2004, pp. 324–335.

[31] B. Gedik, K.-L. Wu, P. S. Yu, L. Liu, Adaptive load shedding for windowed stream joins, in: Proceedings of the 14th ACM international conference on Information and knowledge management, ACM, 2005, pp. 171–178.

[32] M. A. Naeem, G. Weber, C. Lutteroth, G. Dobbie, Optimizing queue-based semi-stream joins with indexed master data, in: Data Warehousing and Knowledge Discovery, Springer, 2014, pp. 171–182.