A Memory-Optimal Many-To-Many Semi-Stream Join

M. Asif Naeem · Gerald Weber · Christof Lutteroth

Abstract Semi-stream join algorithms join a fast stream input with a disk-based master data relation. A common class of these algorithms is derived from hash joins: they use the stream as build input for a main hash table, and also include a cache for frequent master data. The composition of the cache is very important for performance; however, the decision of which master data to cache has so far been solely based on heuristics. We present the first formal criterion, a cache inequality that leads to a provably optimal composition of the cache in a semi-stream many-to-many equijoin algorithm. We propose a novel algorithm, Semi-Stream Balanced Join (SSBJ), which exploits this cache inequality to achieve a given service rate with a provably minimal amount of memory for all stream distributions. We present a cost model for SSBJ and compare its service rate empirically and analytically with other related approaches.

Keywords Many-to-many semi-stream join · cache optimization · performance evaluation

1 Introduction

Data stream processing deals with continuously arriving information – an important phenomenon in the online world with many applications such as network traffic monitoring [26], sensor data [25], health care data [27], web log analysis [28], and inventory and supply-chain management [29]. In the past, data streams were often processed in an ad-hoc manner, e.g. by transforming chunks of the stream into offline datasets and using traditional query processing [30]. However, over the last decade Data Stream Management Systems (DSMSs), which exploit the inherent online characteristics of streams, have become a significant area of research [31,18,32,36].

Stream-based joins are important operations in DSMSs, where just-in-time delivery of data is expected. We consider an important class of stream-based joins, namely semi-stream many-to-many equijoins, which join a single stream with a slowly changing table. An example application is real-time data warehousing [2, 1,17], where the slowly changing table is typically a master data table and the stream contains incoming real-time sales data. A semi-stream join is used to enrich the stream data with master data. The most common type of join in such a scenario is an equijoin, performed for example on a foreign key in the stream data.

Many-to-many relationships between stream and persistent data are frequent and natural. Consider the following two example scenarios: Online advertising companies have master data about user preferences and ads, and requests by users for web pages containing ads arriving continuously in a stream. Advertising is targeted towards a user by joining the users page request with suitable ads, based on the users preferences and ad characteristics in the master data. Each user can have many matching ads, and each ad can match many users. Another application scenario occurs in retail business where retailers place orders for products available from different suppliers. The orders form a stream while the products offered by the suppliers generate the master data. Suppliers can have many products, and each product may be available from

M. Asif Naeem
School of Engineering, Computer and Mathematical Sciences, Auckland University of Technology,
Private Bag 92006, Auckland, New Zealand
E-mail: mnaeem@aut.ac.nz

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

Christof Lutteroth
Department of Computer Science, University of Bath, BA2 7AY, Bath, United Kingdom
E-mail: c.lutteroth@bath.ac.uk

several suppliers. A many-to-many semi-stream join can be used, for example, to quickly find the supplier that can supply a required product for the best price. Another scenario is that of joining sensor data with historical data [25,33].

All modern semi-stream joins recognize the importance of master data caching for performance [11,19, 22,21]. Most domains deal with skewed stream distributions, and by caching frequent master data these algorithms significantly outperform semi-stream algorithms without cache such as MESHJOIN (Mesh Join) [9, 10]. However, current caching approaches in semi-stream joins have two shortcomings: Firstly, existing caching approaches make further assumptions, namely either a clustered index on the table or a key constraint, i.e. assuming that the join attribute is a unique attribute of the master data so a one-to-many join is performed. In contrast, for many-to-many equijoins, caching has received very little attention. Secondly, existing algorithms typically perform caching and cache eviction based on heuristics as opposed to a proven optimality criterion.

We address these shortcomings and present a formal criterion that leads to a provably optimal composition of the cache in a semi-stream equijoin algorithm – the "cache inequality". It works in many-to-many as well as in one-to-many join scenarios. The inequality guides cache migration, i.e. caching and cache eviction of master data. Following that strategy, minimal memory consumption can be achieved. Furthermore, we propose a novel algorithm, Semi-Stream Balanced Join (SSBJ), which can handle the relatively complex cache migration process for a many-to-many join. The algorithm uses the cache inequality to achieve a given service rate with a minimum of memory resources for all stream distributions. We use the term service rate for the total number of stream tuples that the algorithm processes in a unit second. Our main contributions in this research can be summarized as follows:

Cache for many-to-many relationships: We present SSBJ, a general semi-stream equijoin algorithm that allows caching for join attributes matching several master data records. This is nontrivial since these multiple records for a given join attribute need to be managed in unison with respect to caching. Notably, the algorithm deals efficiently with stream tuples that do not have any matching tuple in the master data. The algorithm uses a tuple-level cache and hence can cache exactly the information that is needed most frequently.

Cache Inequality: As the core contribution of this work, we derive an inequality for the memory distribution between the cache and the subsequent stage. This equation is implemented directly in the algorithm and ensures that the algorithm overall consumes a minimum amount of main memory to achieve the required service rate. The cache inequality makes a local decision in the sense that it only deals with the data pertaining to a single join attribute. By applying the cache inequality locally to every join attribute, we obtain globally optimal behavior. It is by adhering to the cache inequality that the proposed algorithm becomes a "balanced" join.

Adaptability: SSBJ adapts online to changes in the distribution of data and adjusts the major system components autonomously, making manual tuning of parameters unnecessary. This autonomous adaptation is not a separate algorithm component for tuning or intelligence, but is a direct consequence of the local decisions made based on the cache inequality. In that sense the optimal behavior of the algorithm can be called to some extend self-organizing.

Continuous transition into main memory join: A frequent argument against disk-based semi-stream joins such as MESHJOIN is that they are superseded by main memory joins, due to sufficiently low costs of main memory. We will show that SSBJ bridges the gap between both types of joins by choosing always the optimal strategy based on the aforementioned adaptability. There are scenarios where SSBJ adapts itself to become a pure main memory algorithm, and there are other scenarios where SSBJ behaves more like MESHJOIN.

Cost model: We present a cost model for SSBJ and validate it empirically. Our experiments show that SSBJ significantly outperforms MESHJOIN [9,10] for skewed data, as it is often found in many application scenarios [12].

Close relationship to other semi-stream joins: The join algorithm that we present here focuses on scenarios where we cannot rely on a specific index structure in the master data. This is advantageous because of the greater generality that our algorithm achieves. However, the algorithm shares a common architecture with algorithms that can benefit from index access such as SSCJ (Semi-Stream Cache Join) [21,22], which we discuss in the following section.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 describes the execution architecture, pseudo-code and cost model for SSBJ. Section 4 describes an experimental study of SSBJ. Section 5 discusses use of SSBJ in a parallelized/distributed configuration and some possible optimizations. Section 6 concludes the paper.

2 Related work

2.1 Symmetric Hash Joins

The Symmetric Hash Join (SHJ) algorithm [3,4] can be seen as a seminal work in the general field of hash-based stream joins. There is a separate hash table for both input streams. 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 stream entirely; however, it stores both relations in memory.

XJoin [6] is an extended form of SHJ that handles memory overflow by flushing the largest hash partition of tuples to disk. XJoin uses a three-stage strategy: First priority is given to joining memory-resident tuples with each other. While there is no incoming stream data, memory-resident tuples are joined with disk tuples. When the inputs are terminated, the tuples stored on disk are joined with each other. Duplicate tuples are avoided with a timestamp approach.

The Double Pipelined Hash Join (DPHJ) [5] is also an extension of Symmetric Hash Join based on two stages. First the algorithm joins the tuples which are in memory, similar to SHJ and XJoin. Then the algorithm marks the tuples which were not joined in memory and joins them on the disk. 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. The algorithm does not perform well for large data. Hash-Merge Join (HMJ) [7] is a similar approach; it is based on push technology and consists of two phases, hashing and merging.

Early Hash Join (EHJ) [8] is an improved version of XJoin with a different flushing strategy and a simplified technique for detecting duplicate tuples. EHJ uses a biased flushing strategy that supports flushing the partition with large input first, which is similar to the strategy presented in Dynamic Hash Join [20]. EHJ determines duplicate tuples based on cardinality: for one-to-one and one-to-many relationships the algorithm does not use timestamps; for many-to-many relationships it requires an arrival timestamp only.

MJoin [13], a generalized form of XJoin, extends the symmetric binary join operators to handle multiple inputs. MJoin uses a separate hash table for each input. On the arrival of a tuple from an input, it is stored in the corresponding hash table and is probed in the other hash tables. A sequence of probing is determined that performs the most selective probes first, so that it is not necessary to probe all the hash tables for each arrival. 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.

A number of tools have been developed for stream warehousing that can process stream data with archive data [14–17]. However, these tools do not provide optimal solutions for the non-uniform distributions of real world data.

2.2 MESHJOIN

MESHJOIN [9,10] was designed specifically as a semi-stream many-to-many join and is the most important related work. The stream serves as the build input for a hash table and the disk-based relation serves as the probe input. MESHJOIN performs a staggered execution of the hash table build and the join in order to process stream tuples steadily, and amortizes disk reads over many stream tuples.

The large relation R is traversed cyclically in an endless loop, and every stream tuple is compared with every tuple in R. R is stored on disk and is read into memory through a disk-buffer with a size of b pages, splitting R into k equal partitions. One traversal of R, called an R cycle, therefore happens in a loop with k steps, subsequently called the main loop. In each step, a new partition of R is loaded into the disk-buffer and replaces the old partition.

The crux of MESHJOIN is that with every main loop step a new chunk of stream tuples is read into main memory. Each of these chunks will remain in main memory for the time of one full R cycle. The chunks therefore leave main memory in the order that they enter main memory and their time of residence in main memory is overlapping. This leads to the staggered processing pattern of MESHJOIN. In main memory, the incoming stream data is organized in a queue, each chunk being one element of the queue. Figure 1 shows a pictorial representation of the MESHJOIN operation: at each point in time, each chunk Qn in the queue has seen a larger number of partitions than the previous, and started at a later position in R (except for the case where the full traversal of R wraps back to the start of R). The figure shows the moment when partition *R2* of R is read into the disk-buffer but is not yet processed.

After loading the disk partition into the disk buffer, the algorithm probes each tuple of the disk buffer in the hash table. If a matching tuple is found, the algorithm generates the join output. After each iteration the algorithm removes the oldest chunk of stream tuples. This chunk is found at the end of the queue; its tuples were joined with the whole of R and are thus completely processed now.

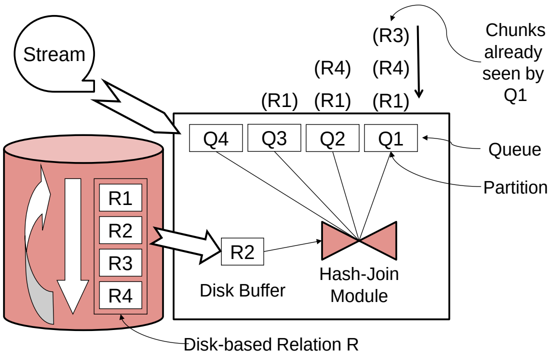


Fig. 1 MESHJOIN before processing R_2

As the algorithm reads R sequentially, no index on R is required. There are also no assumptions about the distribution and the organization of the master data. The MESHJOIN authors reported that the algorithm performs slightly worse with skewed data; at least there is no provision that would make skewed data more efficient to process.

2.3 Other Semi-Stream Joins

A partition-based join [11] was developed to improve MESHJOIN and support also intermittent streams. It requires in the range-partition setup an ordering of the master data according to the range of an attribute, which comes close to requiring a clustered index on the master data. It attempts to join stream tuples with cached master data as soon as they arrive, and puts stream tuples that could not be joined into a partition-based waiting area, to be joined at a later time. However, the time that a tuple is waiting for execution is not bounded, which can be a problem for many applications.

A Semi-Streaming Index Join (SSIJ) [19] was developed to join stream data with disk-based data. The algorithm is divided into three phases. In the pending phase, stream tuples are collected in an input buffer till the size of the buffer is less than a predefined threshold limit or the stream ends. Then the algorithm starts its online phase, where stream tuples from the input buffer are looked up in cached disk blocks. If a required disk tuple exists in the cache, the join is executed and the algorithm produces an output. In the case that a required disk tuple is not available, the algorithm flushes the stream tuple into a stream buffer where it waits for the join phase. If the size of the stream buffer is greater than a threshold, the join phase is activated and a disk invoke is performed. The algorithm joins the tuples from the stream buffer with the tuples in the invoked disk block. A utility counter for each disk block is used to determine which disk blocks should be cached. Although SSIJ is a feasible approach for processing stream data, the algorithm uses a page level cache. The cache will not be fully exploited if some tuples on a page are infrequent. The algorithm does not include a mathematical cost model, so the criteria for tuning the threshold parameters are unclear.

The algorithm for joining stream data with a disk-based relation by Derakhshan et al. [37] uses a cache to store frequent master data tuples and a waiting queue for stream tuples that are not joined through the cache. The algorithm processes this waiting queue in batches. The primary focus of the algorithm is to preserve the arrival order of the stream tuples in the output. While useful for some applications, this order is not important in most applications such as the ones we consider. Preserving this order can cause delays in producing outputs for some stream tuples, e.g. a stream tuple already processed through the cache cannot be output if its predecessor tuple is still in the waiting queue. Derakhshan and others also demonstrated the role of stream-relation joins in a federated stream processing system called MaxStream [38].

Separate cache stages in semi-stream join algorithms have demonstrably increased the performance for data with skewed (e.g. Zipfian) stream distributions. In our previous work we investigated in the past one-to-many joins, where each stream tuple matches exactly one master data record, since this is a very important special case of semi-stream joins. We presented two different approaches: CMESHJOIN (Cached Mesh Join) [24] is based on MESHJOIN and only adds a cache as a front stage, which enabled us to give a first direct analysis of the added value of the cache stage. But since the cache stage did only support one-to-many joins, this limitation applies to the whole algorithm. Accordingly, the algorithm SSCJ (Semi-Stream Cache Join) [21,22] aimed at exploiting this premise (namely that we only consider one-to-many joins) in order to improve performance further and added an index-based approach for the disk-based master data. It can therefore exploit skewed distributions even better. However, it is of course of great interest to have a full generalization of a caching approach to many-to-many joins, without any additional requirement

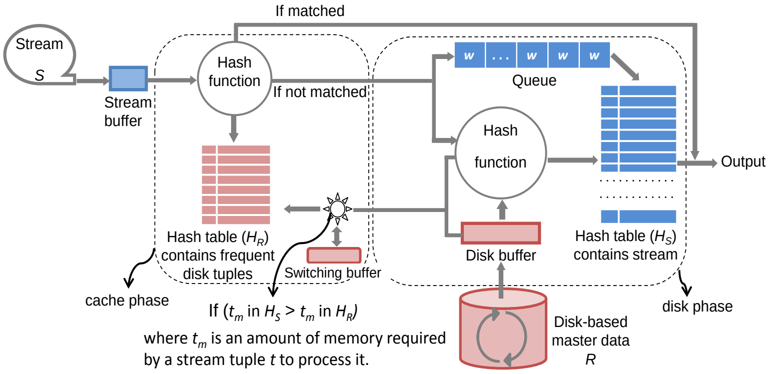

If matched
Stream
S
Stream buffer
Hash function
If not matched
Queue
w ... w w w
Hash function
Output
Hash table (H_R) contains frequent disk tuples
Switching buffer
Disk buffer
Hash table (H_S) contains stream
cache phase
disk phase
If (t_m in H_S > t_m in H_R)
where t_m is an amount of memory required by a stream tuple t to process it.
Disk-based master data
R

Fig. 2 SSBJ architecture

imposed by caching. There are many application scenarios in data warehousing, such as data cleaning [34], where a many-to-many join is required. The extension of a cache-stage to a many-to-many scenario is not straightforward: it is difficult to migrate data consistently to the cache, and the cost benefit in doing so is much more difficult to assess. The new algorithm proposed in this paper addresses these problems.

3 Semi-Stream Balanced Join (SSBJ)

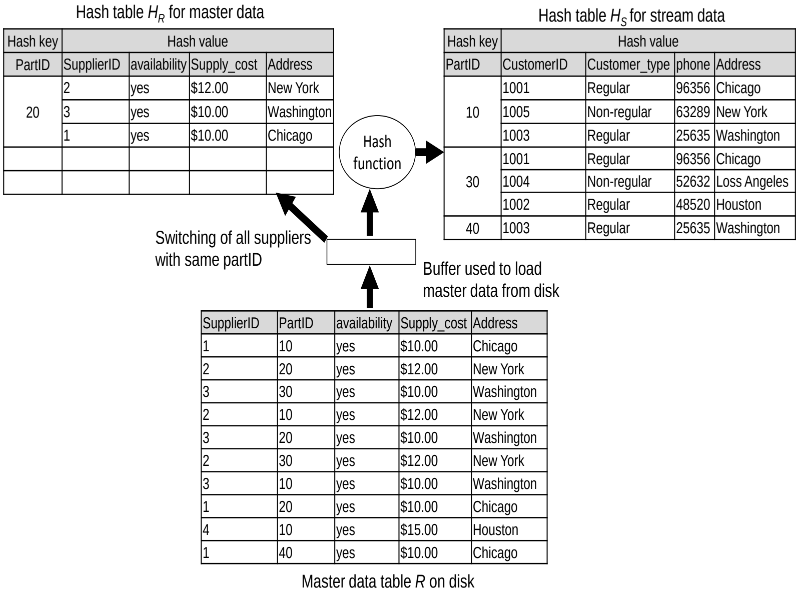
We present here SSBJ, a semi-stream equijoin algorithm suitable for the same scenarios as MESHJOIN. In particular, SSBJ does not require an index on the disk-based table. SSBJ allows for fine-grained, i.e. tuple-level, caching: only frequent tuples are cached. SSBJ uses the cache inequality to make decisions about moving data into the cache and cache eviction. It can deal with many-to-many equijoins. The execution architecture of SSBJ is shown in Figure 2. SSBJ has two complementary hash join modules called *cache phase* and *disk phase*, which directly match phases in the algorithm. Although this architecture is similar to Symmetric Hash Join [3,4], it is used in a different manner.

The stream input first enters the cache phase, which uses selected (cached) master data of relation R as the build input and the stream as the probe input of a hash join. In the cache phase, any stream tuple is either joined completely, or not joined at all and forwarded to the disk phase. The disk phase is based on MESHJOIN and adds instrumentation and cache migration logic: at selected places additional code is executed that controls cache migration but does not change the overall MESHJOIN behavior. SSBJ alternates between the cache and the disk phase, which includes one main loop step of MESHJOIN [9,10]. In further work, the two phases can be parallelized and executed in different threads, but this problem is not the aim of this paper. Note that the stream input for MESHJOIN is filtered by the cache, i.e. certain stream tuples are already processed. Since MESHJOIN makes no assumptions about the stream, this filtering does not affect its correctness.

Relation R and stream S are the external input sources of the join. The dominant components of SSBJ with respect to memory size are two hash tables: the MESHJOIN hash table H_S storing stream tuples, and the cache H_R storing the most frequently accessed tuples from the disk-based relation R. The MESHJOIN queue stores pointers to the stream tuples in H_S enabling the deletion of fully processed tuples, also called *expired tuples*. It is growing with H_S and hence only adds a constant factor to the memory consumption per tuple in the MESHJOIN hash table. The other associated components of SSBJ are a disk buffer, a switching buffer and a stream buffer. The disk buffer is used to load parts of R into memory using equal-size partitions. The switching buffer plays an intermediate role for switching tuples from H_S to H_R. The stream buffer is comparably small, holding part of the stream when necessary.

Algorithm overview: SSBJ runs in an endless loop. One SSBJ loop iteration consists of cache phase, cache eviction, disk phase (MESHJOIN) and cache migration. The cache phase runs until all stream tuples have been processed that have accumulated during the last SSBJ loop iteration. These tuples are referred to as w tuples in the algorithm. The disk phase runs until one partition of the master data has been processed; after each such a step, the oldest stream tuples are deleted from H_S and the queue. Cache eviction and cache migration phases each have two parts: first, deciding whether to switch a join value from one phase to the other, and then executing that decision. While cache eviction is simple, the migration into the cache is complicated and needs a careful choreography based on a state machine approach, as explained later in the detailed algorithm.

Stream model: The caching strategy is based on the same model of the stream that is implicit in most other stream algorithms and that is in accordance with typical application scenarios. From the perspective of the algorithm, the most important aspect is the distribution of the join value in the stream. We have to


Hash function
Switching of all suppliers with same partID
Buffer used to load master data from disk

Hash table H_R for master data

Hash key	Hash value			
PartID	SupplierID	availability	Supply_cost	Address
20	2	yes	$12.00	New York
	3	yes	$10.00	Washington
	1	yes	$10.00	Chicago

Hash table H_S for stream data

Hash key	Hash value			
PartID	CustomerID	Customer_type	phone	Address
10	1001	Regular	96356	Chicago
	1005	Non-regular	63289	New York
	1003	Regular	25635	Washington
30	1001	Regular	96356	Chicago
	1004	Non-regular	52632	Loss Angeles
	1002	Regular	48520	Houston
40	1003	Regular	25635	Washington

SupplierID	PartID	availability	Supply_cost	Address
1	10	yes	$10.00	Chicago
2	20	yes	$12.00	New York
3	30	yes	$10.00	Washington
2	10	yes	$12.00	New York
3	20	yes	$10.00	Washington
2	30	yes	$12.00	New York
3	10	yes	$10.00	Washington
1	20	yes	$10.00	Chicago
4	10	yes	$15.00	Houston
1	40	yes	$10.00	Chicago

Master data table *R* on disk

Fig. 3 An example of a many-to-many join

factor in that the stream can already be an aggregate of several streams. Substreams from many different sources (e.g. purchases from different points of sale) are merged before the stream is received. This usually precludes simple locality between consecutive stream tuples because they stand in no causal connection, and it is natural to assume that the next join value is randomly chosen from an unknown distribution. This also means the algorithm should not rely on locality effects. For every possible join value, it is hence meaningful to speak of the probability that it happens to be the join value of a stream tuple. We call this the *stream probability* of that join value, and because of the above assumption it is independent of the join value of the previous stream tuple. The stream probability of a join value is important for its memory consumption in the disk probing phase. For the algorithm to make caching decisions based on the stream probability, we will have to assume that the stream probability of a join value stays constant over a sufficient time interval. This is necessary in order to estimate it, to make caching decisions based on it and to be able to reap rewards based on a correct caching decision. Over longer periods of time, however, we will expect the probability to change, and we want the algorithm to be able to detect the change if it necessitates a change in caching decision. Changes in probability distribution based on the time of day or the weekday are good examples that match these characteristics.

Cache inequality: SSBJ, like other semi-stream joins, is designed to match with its service rate the arrival rate of the stream. In order to increase the service rate, semi stream joins can increase the memory they use for the internal components. For example, MESHJOIN can be implemented in such a way that it configures itself to use the memory required to deal with a stream arrival rate. The original MESHJOIN does that by determining a set of parameters, especially the chunk size in the queue. In our MESHJOIN implementation we have made this process even simpler by determining the size of each new chunk at the moment it is created. "Dealing with an incoming stream rate" means essentially that the algorithm is able to cope and hence the service rate is equal to that stream rate. If the stream rate gets too high, the service rate would drop and load shedding strategies would have to be employed, but for this work here we consider load shedding out of scope. MESHJOIN, and also our modified version, can therefore deal with a wide range of stream rates. The required service rate is always considered a given parameter. The purpose of a cache is therefore not to increase the service rate, but to reduce the amount of memory needed for a given service rate: SSBJ will use less memory than the original MESHJOIN for a given service rate. This means that given the same memory, the maximum service rate achievable by SSBJ is larger.

Rather than making a global decision about cache size, SSBJ follows a very natural strategy: it decides for every join value individually whether it should be joined in the cache or the disk phase, based on how much memory either option would take. If a join value consumes less memory when joined in the cache, the required amount of memory is added to the cache. The memory used by a join value is the memory that all tuples with that join attribute consume in the respective hash table (H_S or H_R), plus some negligible overheads. We assume that the memory consumption of the hash table is strictly linear in the number of tuples stored.

Example: We consider an equijoin between customers and suppliers on the attribute *partID* as shown in Figure 3. The master data table R contains a list of suppliers. The hash table H_S stores stream tuples which represent customer requests for quotes from all suppliers of a part. The hash table H_R keeps for certain *partID*s all suppliers in cache. In the example we assume that master data tuples and stream tuples all have the same size, i.e. we are considering a special case that will be generalized by the cache inequality in Theorem 2.

We are interested in the number of stream tuples for each *partID* in H_S at any point in time. The current number represents, as explained above, an estimate for the expected value. We see two cases in the example. In the first case, *partID* = 10, H_S contains three stream tuples. However, the total number of suppliers in the master data who produce this *partID* is 4, so switching this *partID* into the cache (H_R) is not a good idea with regard to memory consumption. In other words, H_S is the right place to process this *partID.* In the second case where *partID* is 30, we also have three customer requests against that *partID.* However, there are two suppliers who produce the *partID.* In contrast to the first case, switching *partID* 30 into H_R is a good option because in cache it consumes less main memory.

Note that the memory consumption in the cache for each join attribute value is constant because we consider the master data constant during the runtime of the algorithm, and this means the number of master data tuples matching a join attribute value is constant. The memory consumption in the hash table H_S is stochastically fluctuating for each join attribute value because new stream tuples with this join attribute value arrive in a random fashion. But as we observe it after a full R cycle of master data processing, the observed memory consumption can be considered an estimate for the expected memory consumption according to the law of large numbers. Our experimental results, for example that no hysteresis behavior is needed as shown in Section 4.4, can be seen as corroboration of our estimation strategy: if the estimate were not adequate, a more hesitant approach to evict tuples from the cache would likely have been more appropriate.

The crucial finding is that this strategy is already sufficient to guarantee overall the best memory usage. We capture this aspect in the following Theorem 1. However, we need to generalize the considerations to cases where stream tuples and master data tuples have different sizes, and this will lead to the actual cache inequality in Theorem 2.

Theorem 1 (Minimal memory consumption) If every join value j is placed in the hash table where it uses less memory, then the overall memory consumption is minimal.

Proof The proof is straightforward based on the linear memory consumption of hash tables. Let $m_{cache}(j)$ and $m_{disk}(j)$ be the memory needed for join value j in the cache phase and disk phase, respectively. With J the set of all join values, the memory consumption of both hash tables is:

$$\text{Memory for } H_R \text{ and } H_S = \sum_{j \in J} \min(m_{cache}(j), m_{disk}(j))$$

All join values would use more memory if switched, hence the memory consumption is minimal.

The footprint of other components (such as the queue in MESHJOIN and additional frequency counters in both phases) is negligible, as they are dominated by the tuple sizes. Now we derive what we call *Cache Inequality*, which captures the criterion used in Theorem 1.

Theorem 2 (Cache Inequality) Assume a join value j is used in m master data records and is appearing in an expected number of n stream tuples during one R-cycle. v_S and v_R is the size in bytes of a stream tuple and a master data tuple, respectively. Then j can be processed with less memory consumption in the cache than in the disk phase if:

$$m \times v_R < n \times v_S$$

Proof We prove that the sides of the inequality reflect the memory consumptions of j in either of the two hash tables. To process j in the cache, all master data records matching that join attribute have to be loaded into the cache, i.e. $m \times v_R$. Since all stream tuples remain in H_S for one R-cycle, the expected number of tuples in H_S matching j at any point in time is n. Hence the expected memory consumption of j when processed in the disk phase is $n \times v_S$. The strategy given in the theorem chooses the option with less memory.

Finally we prove that SSBJ automatically turns into a main memory hash join if this is required. The cache phase is a hash join, so if all of R is moved into the cache phase, SSBJ is a pure main memory hash join. Let m_m be the memory consumption if all of R is in the cache. We know that the memory consumption of MESHJOIN increases with service rate [10]. For some high service rate λ_{max}, MESHJOIN consumes more memory than m_m. So we only have to prove the following:

Theorem 3 (SSBJ as main memory join) If the required service rate is stable and larger than λ_{max}, and the joins are configured to match the current service rate, then (a) MESHJOIN consumes more memory than m_m but (b) SSBJ consumes at most m_m. If SSBJ consumes m_m memory, then (c) it moves the whole of R into the cache.

Proof Point (a) is known from MESHJOIN [10]. We prove (b) by contradiction. Let us assume SSBJ runs at a stable service rate greater than λ_{max} and not all of R is yet in the cache phase. If SSBJ uses more memory than m_m, then we know because of the pigeonhole principle that there is at least one join value for which its tuples use more memory in the disk phase than in the cache phase. That join value would have violated the Cache Inequality and would have already been switched to the cache, which contradicts it being in the disk phase and proves (b). Point (c) is the special case of (b) where SSBJ has to move all tuples to the cache.

3.1 Algorithm

Algorithm 1 shows the pseudocode for SSBJ. Without loss of generality, we use *partID* as the join attribute, aligned with the example in Figure 3. The outer loop of the algorithm is an endless loop, which is common in stream processing algorithms (line 2). The body of the outer loop has two main parts: the cache phase (lines 15–23) and the disk phase (lines 24 to 41), and some housekeeping parts (line 3–14 and 42–51). Due to the encircling loop, these two phases alternate.

The algorithm also has a longer-period behavior that is needed for the complex caching protocol of the algorithm. The variable rc is expressing a state of a finite state machine. The finite state machine cycles continuously through three states ('incrementing', 'switching', 'phase-out'), with a fourth state ('cache eviction') occurring in parallel less frequently than the others (lines 5 to 8). Each state has the duration of one full R cycle, i.e. many subsequent executions of the outer loop. Depending on what state the algorithm is in, certain actions in the loop body of the outer loop are activated or deactivated.

In the cache phase the algorithm reads w stream tuples from the stream buffer (line 15). Then the algorithm probes each of those tuples t in the disk-built hash table H_R using an inner loop (line 16). In the case of a match, the algorithm generates the join output(s) without forwarding t to the disk phase and increases the stream tuple frequency *f_count* for join value *t.partID*. Since H_R is a multi-hashmap, there can be more than one match. In the case where t does not match, the algorithm forwards t to the disk phase, i.e. loads t into H_S and enqueues its pointer in the queue Q (line 21).

Lines 24 to 41 comprise the disk phase. Here the algorithm reads a partition of R and loads it into the disk buffer (line 24). In an inner loop (lines 25 to 38) the algorithm looks up all tuples from the disk buffer in the hash table H_S. In the case of a match, the algorithm generates the join output(s); there can be more than one match.

Lines 28 to 37 represent the instrumentation of the disk phase. Leaving the join behavior unchanged, the code maintains important statistics and performs the process of switching join values into the cache. The latter requires three R-cycles, and in each of these cycles a different operation takes place. The cycles are therefore modelled as different states: *incrementing state*, *switching state* and *phaseout state*. These states repeat cyclically, so the current state can be obtained with a modulo-3 counter based on the number of full R-cycles rc (lines 4–7).

Lines 28 and 29 represent the *incrementing state*. The purpose of this state is just to count the number of master data tuples with a given join value *partID* in H_S (line 29), since we made no assumptions that this information can be obtained otherwise. We use variable *s_count* to accumulate these counts. This value will be important later in line 33 to calculate the cache inequality.

Lines 30 to 36 represent the *switching state*. Here first the cache inequality is evaluated in Line 33. In this case the memory consumption in the disk phase is known (left side) while the memory consumption in the cache phase (right side) must be computed from the value obtained in the incremental state. A hysteresis factor s_f is provided that could prevent fluttering if the inequality is just tipping over. However, in our experiments later we will show that the optimal setting of this factor is equal to 1. If the cache inequality recommends switching, then the algorithm marks that *r.partID* for switching and moves the tuple r into a temporary switching buffer named s_B (line 34). The one requirement for switching a join attribute value into the cache is that all master data records for that join attribute must be inserted into the cache at once. We implement this with a switching buffer. An additional reason for the switching buffer is that the algorithm can only reduce memory from H_S in multiples of the size of chunks in the queue, since MESHJOIN assumes equal size of the chunks. It would be easy for us to remove this restriction, but since we wanted to use the precise MESHJOIN algorithm for comparability, we left this restriction in place. At the end of the disk phase the algorithm deletes *t.partID* from Q and all stream tuples related to *t.partID* from H_S (line 39-41).

The *phaseout state* is necessary since after a join value was moved to the cache, it can take one R-cycle until all stream tuples matching that join value are deleted from H_S. In this state an explicit action has to be taken only at the very start of an R-cycle (lines 9 to 13). The join values from the switching buffer are moved to the cache (line 10) and the master data count *s_count* is reset. After that, stream tuples matching those join values are removed from H_S.

Lines 42 to 51 describe the *cache eviction process*. This can happen if a join value gets less frequently used over time. The execution of this feature is not required in every R-cycle. In order to increase the quality of

Table 1 Notations used in cost model of SSBJ

Parameter name	**Symbol**
Total allocated memory *(bytes)*	M
Service rate *(processed tuples/sec)*	μ
Stream arrival rate *(tuples/sec)*	λ
Number of stream tuples processed in each iteration through H_R	w_N
Number of stream tuples processed in each iteration through H_S	w_S
Size of stream tuple *(bytes)*	v_S
Size of disk tuple *(bytes)*	v_R
Size of pointer in the queue *(bytes)*	v_P
Disk buffer size=one partition of R *(tuples)*	b
Size of H_R *(tuples)*	h_R
Size of H_S *(tuples)*	h_S
Size of master data R *(tuples)*	R_t
Fudge factor for both the hash tables	f
Switching factor for moving a tuple into the cache	s_f
Cost to read b number of disk tuples into the disk buffer *(nanosecs)*	$c_{I/O}(b)$
Cost to look-up one tuple either in H_R or H_S *(nanosecs)*	c_H
Cost to generate the output for one tuple *(nanosecs)*	c_O
Cost to switch disk tuples into the cache *(nanosecs)*	c_B
Cost to remove one tuple from H_S and the queue *(nanosecs)*	c_E
Cost to read one stream tuple into the stream buffer *(nanosecs)*	c_S
Cost to append one stream tuple in H_S and the queue *(nanosecs)*	c_A
Total cost for one loop iteration *(secs)*	c_{loop}

the frequency estimate, this process is executed only in every 10th outer loop iteration, i.e. after completing every 10th R-cycle (line 8).

Again, the cache inequality has to be applied (line 46). But this time m_{cache} is known (Line 44) and m_{disk} has to be estimated (line 45) based on the frequency count, which has to be divided by 10 since the count runs for 10 R-cycles. In the case where this comparison is true, the join value j has become infrequent and should be evicted. Again the hysteresis factor s_f is applied. Cache eviction does not need its own state machine approach; the algorithm can simply delete join value j from H_R with all its related tuples (line 46). If now any stream tuple with j appears in the stream data it will be processed in the disk phase. Finally, the algorithm resets *f_count*(s) for all join values in H_R so that the algorithm can start counting again for the next ten iterations (line 49).

Notably, the cache inequality copes with stream tuples that have no matching attribute in the master data. If such stream tuples appear, then the left hand side of the inequality will be smaller, and so the join attribute will be stored in the cache, with no master data tuple. If then later this join attribute does not appear in the stream any more, it will be deleted from the cache, since both sides of the cache inequality are zero.

3.2 Cost Model

In this section we develop a cost model for SSBJ. The main motivation for developing a cost model is to interrelate the key parameters of the algorithm, such as the input size w=w_N+w_S tuples, the processing cost c_{loop} in seconds for these w tuples, the available memory M in bytes, and the service rate μ in tuples/second. The cost model presented here follows the style used for MESHJOIN [9,10]. Equation 1 represents the total memory used by the algorithm (except the stream buffer), and Equation 2 describes the processing cost for each iteration of the algorithm. The notation we use in our cost model is defined in Table 1.

3.2.1 Memory Cost

The largest part of the total memory is assigned to the two hash tables H_S and H_R, while a smaller portion is assigned to the disk buffer and the queue. The memory for the stream buffer and the switching buffer is not considered because it is very small (0.5 MB for each was sufficient in all our experiments). The memory for each component can be calculated as follows:
Memory for the disk buffer (bytes)= $b \cdot v_R$
Memory for H_R (bytes)=$h_R \cdot v_R \cdot f$
Memory for the queue (bytes) = $w_S \frac{R_t}{b} v_P$
Memory for H_S (bytes)=$w_S \cdot f \frac{R_t}{b} v_S$
By aggregating the above, the total memory for the algorithm can be calculated as:

$$M = (b + h_R \cdot f)v_R + w_S \frac{R_t}{b} (v_P + f \cdot v_S) \tag{1}$$

Algorithm 1 Pseudocode code for SSBJ

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:

```
$rc := 0$
while (true) do
    if new $R$-cycle then
        $rc{+}{+}$
        $incrementingState$ :=($rc \mod 3 = 1$)
        $switchingState$ :=($rc \mod 3 = 2$)
        $phaseoutState$ :=($rc \mod 3 = 0$)
        $cacheEvictionProcess$ :=($rc \mod 10 = 0$)
        if $phaseoutState$ then
            MOVE all tuples from $s_B$ to $H_R$
            Adapt MESHJOIN setting $w$
            RESET all $H_S().s\_count$ to 0
        end if
    end if
    READ $w$ stream tuples from the stream buffer
    for each stream tuple $t$ in $w$ do
        if $H_R.lookUp(t.partID)$ then
            OUTPUT all matches with $H_R$
            $H_R(t.partID).f\_count{+}{+}$
        else
            ADD $t$ into $H_S$ and pointer to $t$ into $Q$
        end if
    end for
    READ a partition of $R$ into the disk buffer
    for each tuple $r$ in the disk buffer do
        if $H_S.lookUp(r.partID)$ then
            OUTPUT all matches tuples from $H_S$
            if $incrementingState$ then
                $H_S(r.partID).s\_count{+}{+}$
            else if $switchingState$ then
                $m_{disk} := H_S(r.partID).size() \times v_S$
                $m_{cache} := H_S(r.partID).s\_count \times v_R$
                if $m_{disk} > s_f \times m_{cache}$ then          ▷ This If block implements the cache inequality.
                    LOAD $r$ into $s_B$
                end if
            end if
        end if
    end for
    for each of oldest $w_{old}$ pointers ($t.partID$) in $Q$ do
        DELETE $t.partID$ from $Q$ with tuples from $H_S$
    end for
    if $cacheEvictionProcess$ then
        for each join value $j$ in $H_R$ do
            $m_{cache} := H_R(j).size() \times v_R$
            $m_{disk} := H_R(j).f\_count/10 \times v_S$
            if $m_{cache} \geq s_f \times m_{disk}$ then
                DELETE $H_R(j)$ with all tuples against $k$
            end if
            RESET $H_R(j).f\_count$ to 0
        end for
    end if
end while
```

3.2.2 Processing Cost

In this section we calculate the processing cost for the proposed algorithm. We first calculate the processing cost for individual components:
$c_{I/O}(b)$ =Cost to read b tuples into the disk buffer
$w_N \cdot c_S$ =Cost to read w_N tuples from the stream buffer
$w_S \cdot c_S$ =Cost to read w_S tuples from the stream buffer
$w_N \cdot c_H$ =Cost to look-up w_N tuples in H_R
$w_S \cdot c_A$ =Cost to append w_S tuples into H_S and the queue
$b \cdot c_H$ =Cost to look-up disk buffer tuples in H_S
$w_S \cdot c_E$ =Cost to delete w_S tuples from H_S and the queue
$b \cdot c_B$ =Cost to switch disk tuples into the cache through the switching buffer
$w_N \cdot c_O$ =Cost to generate the output for w_N tuples

$w_S \cdot c_O$ =Cost to generate the output for w_S tuples

By aggregating the above costs the total cost of the algorithm for one iteration can be calculated as:

$$\begin{gathered} c_{loop}(secs) = 10^{-9}[c_{I/O}(b) + b(c_H + c_B) + \\ w_N(c_S + c_H + c_O) + w_S(c_E + c_S + c_A + c_O)] \end{gathered} \tag{2}$$

The term 10^{-9} is a unit conversion from nanoseconds to seconds. Since in c_{loop} seconds the algorithm processes w_N and w_S tuples of the stream S, the service rate μ can be calculated as:

$$\mu = \frac{w_N + w_S}{c_{loop}} \tag{3}$$

The proposed algorithm correctly computes the exact join between a stream and master data provided that $\lambda \leq \mu$. Now by substituting the value of μ from Equation (3):

$$\lambda \leq \frac{w_N + w_S}{c_{loop}} \tag{4}$$

The minimum values of w_N and w_S are specified by Equation 4 as follows:

$$\lambda = \frac{w_N + w_S}{c_{loop}} \tag{5}$$

From Equation (1) the memory required to process w_N and w_S tuples can be calculated as follows:

$$\text{Memory for } w_N = h_R \cdot v_R \cdot f$$

$$\text{Memory for } w_S = b \cdot v_R + w_S \frac{R_t}{b} (v_P + f \cdot v_S)$$

By substituting the amount of memory taken by w_N and w_S in Equation 5, λ can be written as follows:

$$\lambda = \frac{1}{c_{loop}}[h_R \cdot v_R \cdot f + b \cdot v_R + w_S \frac{R_t}{b} (v_P + f \cdot v_S)] \tag{6}$$

This cost model can be compared with the measured behavior of the algorithm.

4 Experimental Evaluation

In this section we present an experimental study comparing SSBJ with MESHJOIN using synthetic and real-life data. We use different parameters as independent variables in order to obtain a range of service rate results that can be consulted in different conditions. Through the experiments we also validate the cost models for SSBJ and MESHJOIN.

4.1 Experimental Setup

Hardware and software specifications: We performed our experiments on a *Pentium-i5* with 8 GB main memory and 500 GB hard drive as secondary storage. We implemented our experiments in Java using the Eclipse IDE. We used libraries, provided by Apache and the Java API, to measure the memory and processing time, respectively. SSBJ needs a hash table that stores several elements against the same key, but the hash table provided by the Java library does not support this feature. Therefore the Apache library class `Multi-Hash-Map` was used. The relation R was stored on disk using a MySQL database. Both of 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.

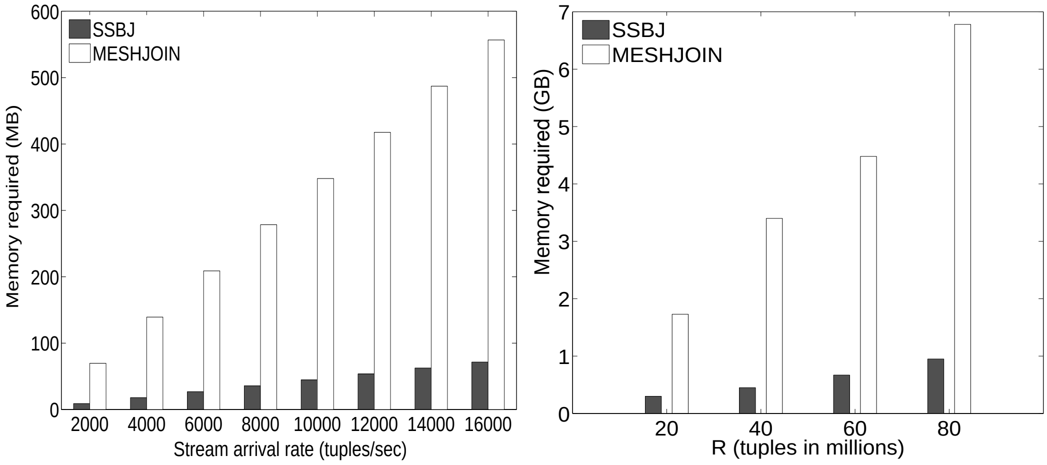
Measurement strategy: In each experiment, both algorithms first completed their warmup phase before starting the actual measurements, so that no transient effects from startup influenced our measurements. For each measurement, we calculated the 95% confidence interval of the mean based on at least 1000 runs for one setting. In the results presented here, the confidence interval is too small to show.

Data specifications: We used three datasets that we refer to in the following as synthetic data, TPC-H, and real-life data. We describe now the characteristics of these three datasets.

Synthetic data. This dataset is providing synthetic data with a configurable Zipfian distribution exponent. This is useful for tests where the Zipfian exponent is an independent variable, since it is not possible to get, e.g., real-life data for a given Zipfian exponent on demand without changing that data. We used this data in experiments varying the exponent from 0 (fully uniform) to 1 (highly skewed). In the following, we

Table 2 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*
Size of each stream tuple	20 *bytes*
Size of each node in the queue	4 *bytes*
Dataset	Based on Zipf's law (skew value from 0 to 1)



(a) Memory for given stream arrival rate while R=2.23 GB, skew=1

(b) Memory for given size of R while service rate=51242 tuples/sec, skew=1

Fig. 4 Memory consumption measurements

refer to this exponent as 'skew' value. The master data in this dataset is unsorted and does not have an index. In the data generator we needed a simple and natural way to ensure that firstly the master data has duplicates and secondly there are join attribute values that do not match a master data record. We achieved these requirements simply by choosing the join attribute values for the master data records randomly with repetition. The data generator is available online, see Section 6. The specifications of our synthetic dataset are shown in Table 2.

TPC-H. As a second dataset we adapted data from the TPC-H benchmark. More precisely, we used the table `Customer` as master data and the table `Order` as stream data. `Order` has a foreign key attribute `custkey`, which is a primary key in `Customer`. The `Customer` table contained 15 million tuples, with each tuple having a size of 223 bytes i.e. (*approx*(3.11GB). The `Order` table contained 150 million tuples, with each tuple having a size of 138 bytes. The motivation of this dataset is to show the performance of the algorithm on a standardized workload that is not designed or chosen by us. A plausible scenario for such a join is to add customer details corresponding to an order, before loading the order into a data warehouse. Although this dataset does not test the many-to-many abilities of SSBJ, it is interesting to measure SSBJ's behavior on data derived from a standard benchmark.

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 [35]. The dataset encodes a sequence of records that evolve over time and is thus well suited for representing the streams of updates from operational sources. As a more concrete example, we can view the weather sensors as the operational data sources that push their updates to a scientific data warehouse, which in turn performs a semi-stream join for the assignment of surrogate keys before the storage of the sensor observations. The same dataset was also used to evaluate the original MESHJOIN [9,10]. More specifically, the master data table was built by combining meteorological data for the months April and August, while the stream data was built by combining data files for December. The master data table contains 20 million tuples and the stream data table contains 6 million tuples. The size of each tuple in both the master data table and the stream data table is 128 bytes. Both tables are joined using a common attribute, longitude (same as in [10] and [19]). The domain of the join attribute is the interval [0,36000]. This join is indeed a many-to-many join.

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

4.2 Memory Consumption

The algorithms SSBJ and MESHJOIN adapt their memory consumption to the stream arrival rate, therefore a first natural experiment is to observe memory as a dependent variable. We measure it with respect to service rate and the size of the master data as independent variables. In this experiment we used the synthetic data, since we wanted to get a clear understanding of the effect of a well-defined Zipfian exponent (set to 1) on the algorithm behavior.

For the presentation in the figures, we naturally present only a slice of the experimental data, i.e. varying one parameter at a time. The results in Figure 4(a) were obtained by varying the stream arrival rate, while keeping a fixed size of R as 20 million tuples (≈2.23 GB). The stream arrival rate was controlled by varying the parameters of our data generator. The results show that MESHJOIN uses about 7 times more memory than SSBJ under all stream arrival rate settings. This shows that the caching strategy removes a large share of the entries in the hash table of the disk phase. The results in Figure 4(b) are obtained by varying the size of R, while keeping a fixed service rate of 51242 tuples/sec. In this case the ratio between the memory consumption of both algorithms varies from 7 for $|R|$=20 million tuples to a ratio of 5 for $|R|$=80 million tuples. This is due to the fact that with the Zipfian distribution, a larger table contains more infrequent tuples which are not cached.

4.3 Service Rate Measurements

For dimensioning a system it is natural to ask what service rate can be obtained with a certain available memory and what service rate impact can be expected when increasing the master data size. In the experiments in this section the service rate is therefore the dependent variable and we measure it for both algorithms while varying available memory, size of R and the skew of the stream distribution. Again, we vary one parameter at a time. For the experiments presented in Figures 5(a)–(c) and (f) we used the synthetic data.

Available memory: We vary the available memory size from 1% to 10% of R, while keeping the size of R at 100 million tuples (≈11.18 GB) and the skew at 1. The results of the experiment are shown in Figure 5(a). SSBJ was 7 times faster than MESHJOIN for very limited memory (1% of R) and 8 times faster for 10% of R. This shows that the impact of the cache increases slightly with larger memory.

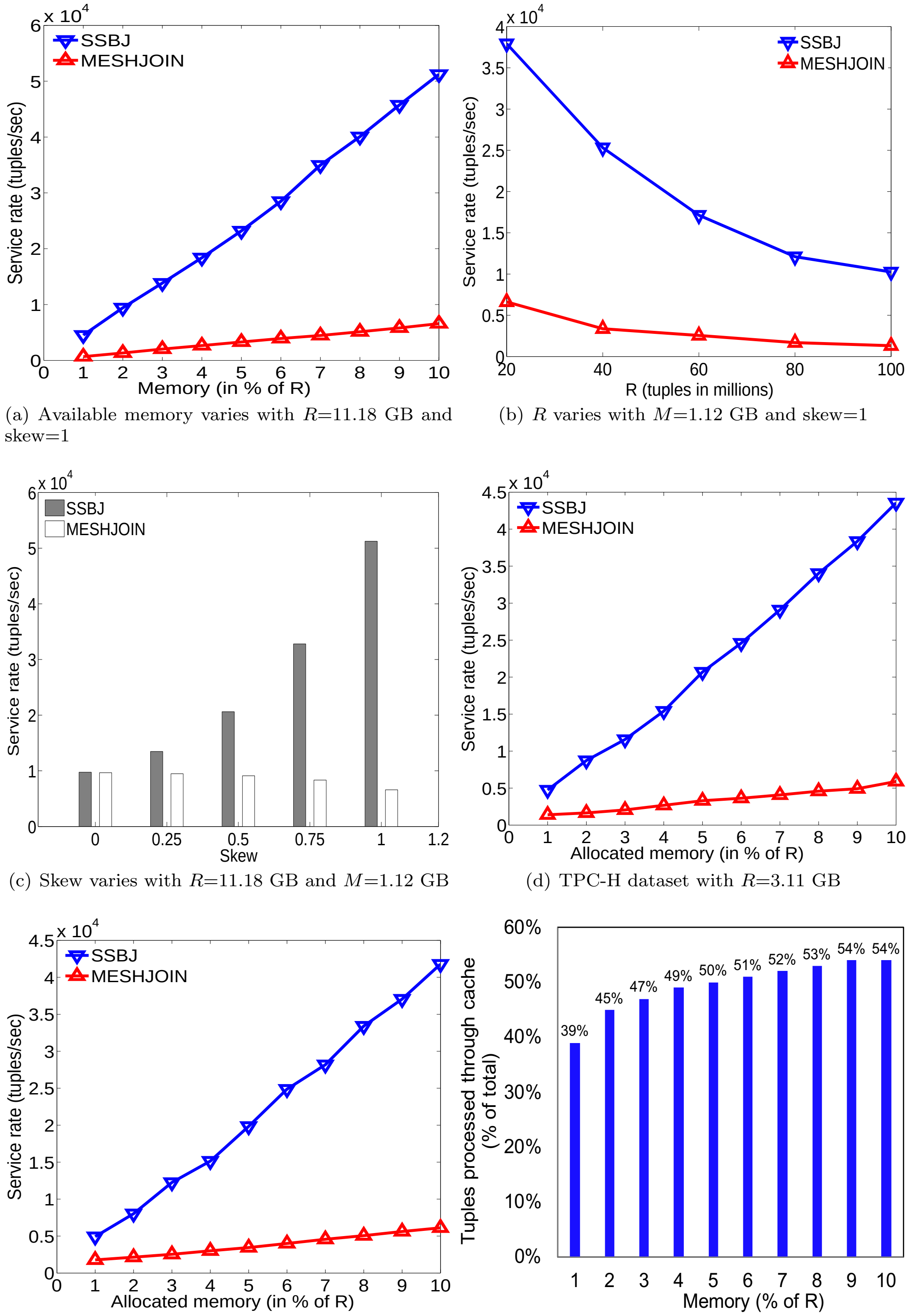
Size of *R*: We varied the size of R while keeping the main memory size fixed (≈1.12 GB). The skew value is 1 for all settings of R. The results in Figure 5(b) show that SSBJ performed 7 times better than MESHJOIN for $|R|$= 20 million tuples and 5.7 times better for $|R|$= 100 million tuples. This general trend, although here expressed as difference in service rate, seems to be consistent with the results in Figure 4(b).

Skew value: Finally, we varied the Zipfian exponent (skew) of the stream data distribution in the range 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 (≈11.18 GB) and the available memory was set to 10% of R (≈1.12 GB). The results presented in Figure 5(c) show that as soon as the data has even a slight skew, SSBJ starts to perform better than MESHJOIN, and this difference becomes more pronounced for increasing skew values. At a skew of 1, SSBJ performed approximately 8 times better than MESHJOIN. MESHJOIN does not exploit skew and 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 relatively short tails. However, we predict that for such short tails the trend would continue.

TPC-H and real-life datasets: We also compared the service rates of both algorithms using the TPC-H and the real-life weather datasets (see Section 4.1). In these experiments we measured the service rate produced by both algorithms at different memory settings. The results of using TPC-H data and real-life data are shown in Figures 5(d) and (e) respectively. From Figure 5(d) it can be noted that, under memory size 10% of R, SSBJ performed about 7.4 times better than MESHJOIN. Also for a smaller memory size, 1% of R, SSBJ still performed about 3.4 times better that MESHJOIN. Similarly for the real-life weather dataset, it is obvious from Figure 5(e) that SSBJ outperforms MESHJOIN under all memory settings.

An interesting aspect of these measurements is that the TPC-H data as well as the real-life data produce very similar behavior to our synthetic dataset. This is giving us confidence that the use of synthetic data is in itself justified and relevant. The behavior for both datasets moreover matches quite closely the behavior for the Zipfian exponent of one in the synthetic data. This is not unexpected, since such a moderately strong skew is, as we said at the very beginning, a good rule of thumb for many real-life situations.

Role of the cache: In order to understand the role of the cache in more detail, we performed an experiment where we counted the stream tuples processed only through the cache phase. We ran the experiment for different memory settings, from 1% of R to 10% of R with R ≈11.18 GB, with a constant arrival rate. This means the independent variable set by the experimenter is memory size, while the service rate is a dependent variable. Since we still want to achieve for the given memory setting the highest possible service rate, the cache inequality still holds. For each memory setting we counted the stream tuples for 3000



(a) Available memory varies with R=11.18 GB and skew=1

(b) R varies with M=1.12 GB and skew=1

(c) Skew varies with R=11.18 GB and M=1.12 GB

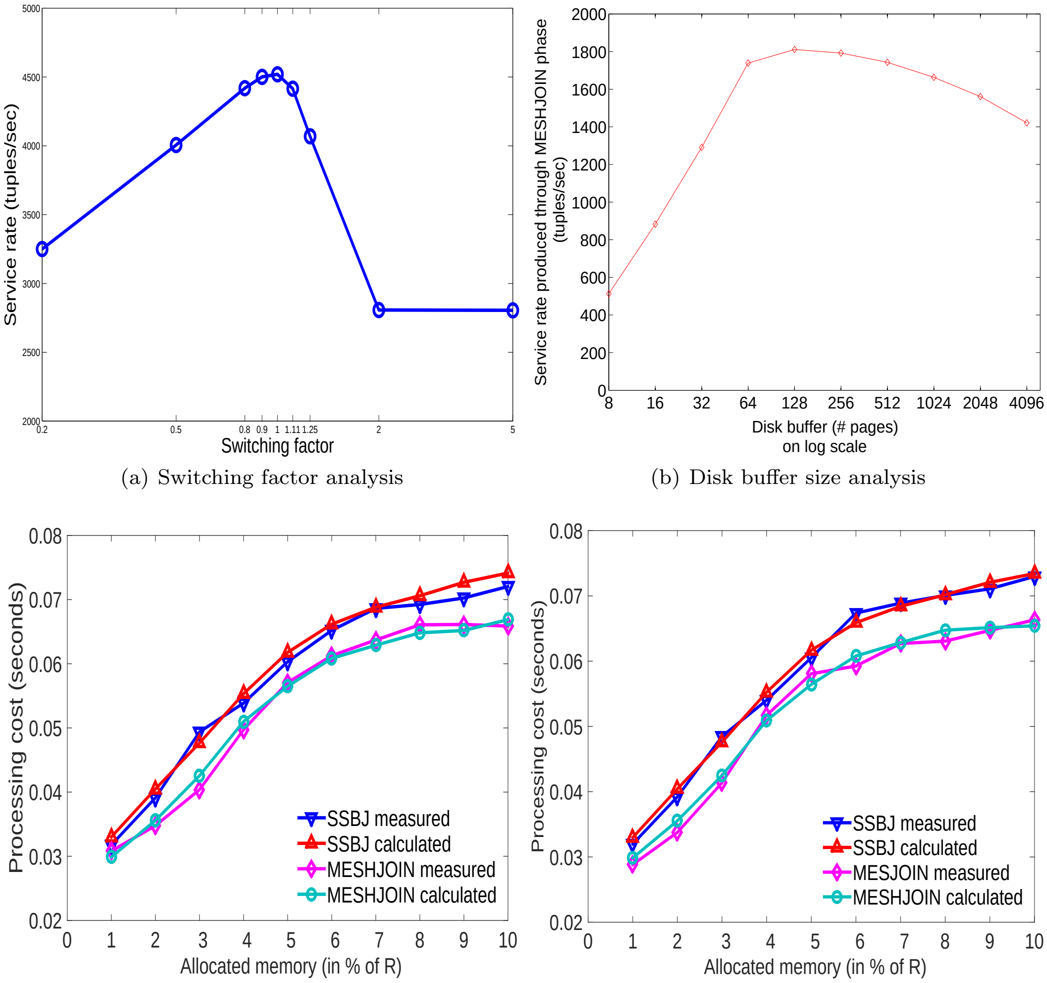
(d) TPC-H dataset with R=3.11 GB

(e) Real dataset with R=2.38 GB

(f) Role of cache with R=11.18 GB and skew=1

Fig. 5 Service rate analysis

iterations and took the average. The results in Figure 5(f) show the percentage of the total stream tuples that were processed through the cache phase. 39% of the stream tuples were processed through the cache for 1% of R, growing to 54% for 9% of R and above. The percentage of tuples processed in the cache grows initially because with the growing queue, stream tuples with the same join value take up more and more space, and the cache inequality optimizes memory by moving the corresponding master data into the cache. The percentage of tuples processed in the cache does not grow further from some point onward because the



(a) Switching factor analysis

(b) Disk buffer size analysis

(c) Cost validation at skew=0

(d) Cost validation at skew=1

Fig. 6 Sensitivity analysis and cost validation

service rate matches the arrival rate and the master data for all frequently occurring join values is already in the cache.

4.4 Further Validation of the Algorithm

Sensitivity of cache migration: A question that is often raised in the context of adaptive behavior is whether it is helpful to implement a hysteresis behavior. In our algorithm a natural point to consider such a behavior is the cache migration: Once a join attribute has been moved from the disk phase to the cache phase or vice versa, a more substantial change in memory consumption would be needed for the tuple to move back. This could avoid oscillations and premature switching of individual join attributes. It would result in a hysteresis. An optional feature allowing such a behavior was added in the form of a switching factor in the cache equation, with a factor of 1 meaning no hysteresis. We investigated empirically which switching factor is optimal, and found that surprisingly the optimal setting was 1, which is equivalent to having no switching factor. The results are shown in Figure 6(a). In fact the measurements show that with a switching factor other than 1 the service rate deteriorates quite markedly. We used fixed sizes of R (20 million tuples, 2.23 GB), and memory (1% of R) and a stream data skew of 1. The results indicate that the disadvantages of a suboptimal cache use far outweigh the stability benefits that a hysteresis could offer in this scenario. In that sense the results are also a corroboration that the estimators for the frequency that we use are good estimators.

Sensitivity to the disk buffer size: The disk buffer is a very sensitive component of the disk phase in SSBJ, 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 (≈11.18 GB), a total allocated memory of 1% of R (≈0.11 GB), and a Zipfian distribution with a skew of 1. To observe the effect clearly, we measured only the service rate produced through the disk phase of SSBJ. The results are shown in Figure 6(b). 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.

Costs model validation: We validated the cost models for both the algorithms with two extreme cases of skew (0 and 1) in the stream data distribution using the synthetic dataset: at 0 the input stream S is uniform, while at 1 the stream has a strong skew. In the validation process we compared the calculated values of a central cost parameter, namely c_{loop}, with measurements of c_{loop}. Figures 6 (c) and (d) present the results of our experiments for skew 0 and 1 respectively. It is clear that for both algorithms the calculated cost closely resembled the measured cost, which supports the correctness of the cost models.

5 Discussion

Distributed configurations: Although the focus of this paper is the performance of SSBJ on a single processing node, SSBJ has characteristics that allow it to be applied also in a distributed configuration with many processing nodes using common parallelization strategies. Data parallelism is possible by running SSBJ as described on different nodes in parallel. The nodes could be fed by different data sources, or stream tuples could be load balanced across them. SSBJ can also be applied in a task parallel manner in scenarios where multiple joins are required. Stream tuples that have been joined can be joined further with the same or other master data tables in other nodes, forming a join pipeline. SSBJ's property of not requiring master data indexes is particularly useful here as it means processing nodes running SSBJ can quickly be reconfigured to perform any equijoin on incoming stream tuples with available master data, no matter if there is an index on the attributes involved.

Indexes on master data: In a flexible scenario where the joins to perform may change there may not be indexes on the desired join attributes available. Furthermore, it may not be desirable to have these indexes in the first place, as a) indexes unnecessary for a particular join take up memory and b) common, page-based indexes do not offer the granularity of the tuple-level cache proposed in SSBJ and would likely be less efficient. If indexes on the required join attributes are available, they could be applied in the disk phase using a semi-stream join such as HYBRIDJOIN [23]. Because of HYBRIDJOIN's similarity to the disk phase of SSBJ, the cache inequality could still be applied in this case.

Projection techniques: It is possible to reduce the amount of memory taken up by the cache by considering the master data columns required by the application. If it is not necessary for the application to join the full master data records with the stream tuples, then it is possible to perform a projection first before loading master data into the cache. In that case, the cache would only contain the columns of interest to the application.

One-to-many joins: When considering the performance of SSBJ in a one-to-many scenario, we can compare it with the performance of a typical one-to-many join using caching for performance gain such as Semi-Streamed Index Join (SSIJ) [19]. When considering the cache inequality in a one-to-many scenario then $m = 1$, so a tuple processed in the cache is consuming less memory if $v_R < n \times v_S$. This means that the likelihood of a tuple being processed in the cache is higher than for the many-to-many case, assuming that master data tuples are not a lot bigger than stream tuples (v_R vs. v_S) and there are frequently occurring join values (n). In that case, both the caches of SSBJ and SSIJ will be able to process stream tuples quickly; however, the cache of SSBJ will be smaller as it is a tuple-level as opposed to a block-level cache and does not require an index to find the cached block required to join a stream tuple. The caches of both SSBJ and SSIJ will contain master data for frequent join values; however, SSIJ will replace infrequent master data blocks in the cache with recently loaded ones even if the latter are even less frequent. Overall, this makes SSBJs cache well suited for the one-to-many scenario and more efficient than that of SSIJ.

Like many semi-stream join algorithms, both SSBJ and SSIJ have a disk phase, which processes tuples after cache misses. In order to amortise disk accesses over many stream tuples, SSIJ buffers stream tuples and creates a disk access plan to join them, using an index to find required blocks. SSBJ also tried to amortise disk reads over several stream tuples, but instead of using an index, SSBJ continuously scans through the whole master data. SSIJs disk access plan can be more efficient than SSBJs table scan approach if loads are skewed towards a comparatively small number of blocks. However, a good cache will already have removed much of the skew in the stream distribution by caching frequent master data. As a result, the disk phase will likely have to deal with a distribution that is much more uniform, having to access many master data blocks at different disk locations. In this case, the table scan approach taken by SSBJ is likely more efficient than a specific access plan. In our experiments we noted that every partition of master data SSBJ read from disk was joined with roughly the same, fairly high number of stream tuples in the queue. This indicates that the cyclic table scans performed by SSBJ are likely more efficient than a targeted disk access plan involving random access.

6 Conclusions

We have presented a many-to-many semi-stream join (SSBJ) with a theoretically founded caching strategy. The first formal criterion for caching in such algorithms – the "cache inequality" – leads to a provably optimal composition of the cache in a semi-stream many-to-many equijoin algorithm. In particular, the semi-stream join presented here provides a smooth transition between disk-based approaches such as MESHJOIN on the one hand and main-memory approaches on the other hand. We have provided a theoretical analysis of the cache inequality and have shown why it reaches optimality within the scope that has been set. We have provided a detailed cost model of the algorithm implementation and extensive experimental results that show a significant service rate improvement of the algorithm over MESHJOIN. The experiments have been performed with synthetic data, with data derived from standard benchmarks and with real-life data. The cache in SSBJ does not interfere with the subsequent disk-based join, therefore it is possible to apply this caching strategy to other semi-stream joins in the future.

Source URL: Our reference implementation of SSBJ can be found at
`https://www.cs.auckland.ac.nz/research/groups/serg/j/bj/`

References

1. A. Karakasidis, P. Vassiliadis and E. Pitoura, "ETL queues for active data warehousing", *Proc. 2nd International Workshop on Information Quality in Information Systems (IQIS '05),* pp. 28-39, ACM, 2005.
2. M. A. Naeem, G. Dobbie, and G. Weber, "An Event-Based Near Real-Time Data Integration Architecture", *Proc, 12th Enterprise Distributed Object Computing Conference Workshops (EDOCW '08),* pp. 401-404, IEEE, 2008.
3. A. N. Wilschut, and P. M. G. Apers, "Dataflow query execution in a parallel main-memory environment", *Proc. first International Conference on Parallel and Distributed Information Systems (PDIS '91),* pp. 68-77, IEEE, 1991.
4. A. N. Wilschut and P. M. G. Apers, "Pipelining in query execution", *Proc. International Conference on Databases, Parallel Architectures and Their Applications (PARBASE '90),* pp. 562-562, IEEE, 1990.
5. Z. G. Ives, D. Florescu, M. Friedman, A. Levy and D. S. Weld, "An adaptive query execution system for data integration", SIGMOD Rec., vol. 28, no. 2, pp. 299–310, ACM, 1999.
6. T. Urhan and M. J. Franklin, "XJoin: A reactively-scheduled pipelined join operator", IEEE Data Engineering Bulletin, vol. 23, pp. 2000.
7. M. F. Mokbel, M. Lu, and W. G. Aref, "Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results", *Proc. 20th International Conference on Data Engineering (ICDE '04)*, pp. 251, 2004.
8. R. Lawrence, "Early Hash Join: A configurable algorithm for the efficient and early production of join results", *Proc. 31st International Conference on Very Large Data Bases (VLDB '05),* pp. 841-852, VLDB Endowment, 2005.
9. N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N. E. Frantzell, "Supporting Streaming Updates in an Active Data Warehouse", *Proc. 23rd International Conference on Data Engineering (ICDE '07),* pp. 476-485, IEEE, 2007.
10. 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., vol. 20, no. 7, pp. 976-991, IEEE Educational Activities Department, 2008.
11. A. Chakraborty and A. Singh, "A partition-based approach to support streaming updates over persistent data in an active datawarehouse", *Proc. IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09),* pp. 1-11, 2009.
12. C. Anderson, *The Long Tail: Why the Future of Business Is Selling Less of More*, Hyperion, 2006.
13. S. D. Viglas, J. F. Naughton and J. Burger, "Maximizing the output rate of multi-way join queries over streaming information sources", *Proc. 29th International Conference on Very large Data Bases (VLDB '2003),* pp. 285-296, VLDB Endowment, 2003.
14. M. H. Bateni, L. Golab, M. T. Hajiaghayi and H. Karloff, "Scheduling to minimize staleness and stretch in real-time data warehouses", *Proc. 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09),* pp. 29-38, 2009.
15. L. Golab, T. Johnson and V. Shkapenyuk, "Scheduling Updates in a Real-Time Stream Warehouse", *Proc. 25th International Conference on Data Engineering (ICDE '09),* pp. 1207-1210, 2009.
16. G. Lukasz and J. Theodore, "Consistency in a Stream Warehouse", *Conference on Innovative Data Systems Research (CIDR '11),* pp. 114-122, 2011.
17. L. Golab, T. Johnson, J. S. Seidel and V. Shkapenyuk, "Stream warehousing with DataDepot", *Proc. 35th ACM SIGMOD International Conference on Management of Data*, pp. 847-854, 2009.
18. 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", *Proc. ACM SIGMOD International Conference on Management of Data,* pp. 668-668, 2003.
19. M.A. Bornea, A. Deligiannakis, Y. Kotidis, and V. Vassalos, "Semi-Streamed Index Join for near-real time execution of ETL transformations", *Proc. IEEE 27th International Conference on Data Engineering (ICDE'11),* pp. 159-170, 2011.
20. D. J. DeWitt and J. F, Naughton, "Dynamic Memory Hybrid Hash Join", *University of Wisconsin,* 1995.
21. M. A. Naeem, G. Weber, G. Dobbie and C. Lutteroth, "SSCJ: A Semi-Stream Cache Join Using a Front-Stage Cache Module", *Proc. 15th International Conference on Data Warehousing and Knowledge Discovery,* pp. 236-247, Springer, 2013.
22. M. A. Naeem, G. Dobbie, G. Weber, "A Lightweight Stream-Based Join with Limited Resource Consumption", *Proc. 14th International Conference on Data Warehousing and Knowledge Discovery,* pp. 431-442, Springer, 2012.

23. M. A. Naeem, G. Dobbie and G. Weber, “HYBRIDJOIN for Near-Real-Time Data Warehousing”, International Journal of Data Warehousing and Mining (IJDWM), vol. 7, issue 4, pp. 21-42, IGI Publishing, 2011.
24. M. A. Naeem, G. Weber, G. Dobbie, and C. Lutteroth, “A Generic Front-stage for Semi-stream Processing”, *Proc. of the 22nd ACM International Conference on Information & Knowledge Management,* pp. 769-774, ACM, 2013.
25. P. Bonnet, J. Gehrke and P. Seshadri, “Towards Sensor Database Systems”, *Proceedings of the Second International Conference on Mobile Data Management (MDM),* pp. 03-14, Springer, 2001.
26. C. Cranor, T. Johnson, O. Spataschek and V. Shkapenyuk, “Gigascope: A Stream Database for Network Applications”, *Proc. ACM SIGMOD International Conference on Management of Data,* pp. 647-651, ACM, 2003.
27. C. Chung-Min, H. Agrawal, M. Cochinwala and D. Rosenbluth, “Stream query processing for healthcare bio-sensor applications”, *Proc. IEEE 20th International Conference on Data Engineering (ICDE),* pp. 791-794, IEEE, 2004.
28. A.C. Gilbert, Y. Kotidis, S. Muthukrishnan and M. Strauss, “Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries”, *Proc. 27th International Conference on Very large Data Bases (VLDB),* pp. 79-88, VLDB Endowment, 2001.
29. E. Wu, Y. Diao and S. Rizvi, “High-performance complex event processing over streams”, *Proc. ACM SIGMOD International Conference on Management of Data,* pp. 407-418, 2006.
30. M.W. Blasgen and K.P. Eswaran, “Storage and Access in Relational Data Bases”, IBM Syst. J., vol. 16, issue 4, pp. 363-377, IBM Corp., 1977.
31. M.W. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul and S. Zdonik, “Aurora: A New Model and Architecture for Data Stream Management”, The VLDB Journal, vol. 12, issue 2, pp. 120-139, Springer, 2003.
32. J. Chen, D. Carney, D.J. DeWitt, F. Tian and Y. Wang, “NiagaraCQ: A Scalable Continuous Query System for Internet Databases”, SIGMOD Rec., vol. 29, issue 2, pp. 379-390, ACM, 2000.
33. S. Madden and M.J. Franklin, “Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data”, *Proc. IEEE 18th International Conference on Data Engineering (ICDE),* IEEE, pp. 555-566, 2002.
34. S.R. Jeffery, *Pay-as-you-go Data Cleaning and Integration*, University of California, Berkeley, 2008.
35. C.J. Hahn, S.G. Warren, and J. London, *Edited synoptic cloud reports from ships and land stations over the globe, 1982–1991*, Oak Ridge National Lab., TN (United States), 1996.
36. S. Chakravarthy, Q. Jiang, “Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing”, *Springer Science & Business Media*, vol. 36, 2009.
37. R. Derakhshan, A. Sattar, B. Stantic, A new operator for efficient stream-relation join processing in data streaming engines. *Proc. of the 22nd ACM international conference on Information & Knowledge Management*, pp. 793-798, ACM, 2013.
38. I. Botan, Y. Cho, R. Derakhshan, N. Dindar, A. Gupta, L. Haas, K. Kim, C. Lee, G. Mundada, MC Shan, N. Tatbul. A demonstration of the MaxStream federated stream processing system. *In IEEE 26th International Conference on Data Engineering (ICDE)*, pp. 1093-1096, IEEE, 2010.