J Intell Inf Syst (2011) 37:245–265
DOI 10.1007/s10844-010-0138-z

Load shedding for multi-way stream joins based on arrival order patterns

Tae-Hyung Kwon · Ki Yong Lee · Myoung Ho Kim

Received: 23 December 2009 / Revised: 4 October 2010 / Accepted: 5 October 2010 /
Published online: 15 October 2010

Abstract We address the problem of load shedding for continuous multi-way join queries over multiple data streams. When the arrival rates of tuples from data streams exceed the system capacity, a load shedding algorithm drops some subset of input tuples to avoid system overloads. To decide which tuples to drop among the input tuples, most existing load shedding algorithms determine the priority of each input tuple based on the frequency or some historical statistics of its join attribute value, and then drop tuples with the lowest priority. However, those *value-based* algorithms cannot determine the priorities of tuples properly in environments where join attribute values are unique and each join attribute value occurs at most once in each data stream. In this paper, we propose a load shedding algorithm specifically designed for such environments. The proposed load shedding algorithm determines the priority of each tuple based on *the order of streams* in which its join attribute value appears, rather than its join attribute value itself. Consequently, the priorities of tuples can be determined effectively in environments where join attribute values are unique and do not repeat. The experimental results show that the proposed algorithm outperforms the existing algorithms in such environments in terms of effectiveness and efficiency.

T.-H. Kwon
Command and Control Directorate, Systems Division, ROK Air Force, P.O. Box, 309, Sinjang-dong, Pyeongtaek-si, Gyeonggi-do, Republic of Korea
e-mail: thkwon923@naver.com

K. Y. Lee (✉)
Department of Computer Science, Sookmyung Women's University, 52 Hyochangwon-gil, Yongsan-gu, Seoul, 140-742, Republic of Korea
e-mail: kiyonglee@sookmyung.ac.kr

M. H. Kim
Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 373-1 Guseong-Dong, Yuseong-Gu, Daejeon 305-701, Republic of Korea
e-mail: mhkim@dbserver.kaist.ac.kr

Keywords Load shedding · Multi-way stream join · Stream processing · Arrival order pattern

1 Introduction

Recently, there has been a growing interest in the processing of continuous queries over multiple data streams. In multiple data stream applications, tuples from each data stream continuously arrive at the system and queries posed to the system are continuously evaluated against the incoming tuples. Example of such applications include network traffic monitoring (Das et al. 2003; Cranor et al. 2003), sensor-based environmental monitoring (Yu et al. 2006; Gehrke and Madden 2004; Golab and Ozsu 2003), and moving object tracking (Hammad et al. 2003).

In reality, data streams are often bursty and the arrival rates of tuples from data streams are quite unpredictable (Das et al. 2003; Dobra et al. 2002; Gedik et al. 2007; Bai et al. 2007; Law and Zaniolo 2007). Since most data stream applications require real-time or near real-time response, it is important for the system to properly handle even very high tuple arrival rates. If the arrival rates of tuples from data streams exceed the system capacity, the system becomes overloaded and will not be able to process all the queries posed to it. To address this problem, various *load shedding* techniques have been proposed in the literature (Gedik et al. 2007; Bai et al. 2007; Law and Zaniolo 2007). The main idea of load shedding is to keep up with high tuple arrival rates by dropping some subset of input tuples, while maximizing the output rate of the queries (Bai et al. 2007; Law and Zaniolo 2007).

In this paper, we address the problem of load shedding for multi-way stream join queries. Multi-way join queries are widely used in multiple data stream applications to identify objects or entities that appear in all the data streams. The following are typical examples of multi-way stream join queries.

Example 1 (*Finding news articles posted on multiple news sites*) Consider five news sites (e.g., CNN.com), each of which feeds a data stream in which each tuple contains information about a news article posted on the site. Assume that a single news article can be posted on multiple news sites. Each article is identified by three attributes: *Author*, *Title*, and *Date*. To find news articles that are posted on all the five sites, we can run a 5-way join query over the five data streams on *Author*, *Title*, and *Date*.

Example 2 (*Monitoring network traffic passing through multiple routers*) Consider four network routers, R_1, R_2, R_3, and R_4, located sequentially along a network path. Assume that each router feeds a data stream in which each tuple contains information about a packet passing through the router. Each packet is identified by its packet identifier (*pid*). To monitor packets passing through all the four routers, we can run a 4-way join query over the four data streams on *pid*.

Example 3 (*Analyzing user access patterns on a Web site*) Consider *n* Web pages in a Web site, which are linked together by hyperlinks. A user can travel from one Web page to another via hyperlinks to find the desired information. To analyze user access patterns, a data stream is generated for each Web page, in which each tuple contains information about an access to the Web page (e.g., session ID, IP address, timestamp, etc.) Each user is identified by a unique session ID (*sessionID*). To count the number of users visiting all the n Web pages, we can run an n-way join query over the n data streams on *sessionID*.

Example 4 (*Assessing traffic flows on a road network*) Consider n points on a road network, at each of which an automatic number plate recognition (ANPR) system is deployed. Assume that each ANPR system feeds a data stream in which each tuple contains information about a vehicle passing the point. Each vehicle is identified by its plate number (*plateNo*). To determine the number of vehicles passing all the n points in a day, we can run an n-way join query over the n data streams on *plateNo*.

Because data streams are unbounded and potentially infinite, a *windowed* join query joins a tuple from one data stream with only the tuples in the *windows* of the other data streams. A window can be *time-based* (e.g., the tuples that have arrived within the last 10 minutes) or *count-based* (e.g., the 100 most recently arrived tuples) (Golab and Ozsu 2003; Viglas et al. 2003).

In this paper, we propose an effective load shedding algorithm for multi-way windowed stream join queries. If the memory allocated to the window of a data stream is full and a new tuple arrives at the window of the data stream, a load shedding algorithm decides which tuple(s) to drop among those in the window. To make such a decision, a load shedding algorithm determines the priority of each tuple in the window based on some measure, and then drops tuple(s) with the lowest priority. For a join query, a load shedding algorithm determines the priority of each tuple based on its *productivity* (i.e., the number of output tuples that will be produced by the tuple) to maximize the output rate of the query.

1.1 Motivation

To determine the priority of a tuple for a join query, most existing load shedding algorithms are based on some statistics of its join attribute value. More specifically, most existing load shedding algorithms can be classified into two categories: (1) *Frequency-based algorithms* (Law and Zaniolo 2007): The priority of a tuple t is determined in proportion to the number of tuples in windows with the same join attribute value as that of t, and (2) *Output-based algorithms* (Srivastava and Widom 2004): The priority of a tuple t is determined in proportion to the number of output tuples that have been produced by tuples with the same join attribute value as that of t.

However, there are many applications where those *value-based* algorithms cannot determine the priorities of tuples properly. Very often, data streams are joined on key attributes that serve as a unique identifier for an object or entity (e.g., *Author*, *Title*, and *Date* in Example 1, *pid* in Example 2, *sessionID* in Example 3, and *plateNo* in Example 4). In many such cases, the values of key attributes are unique and each value occurs at most once in each data stream. For example, the combined values of the attributes *Author*, *Title*, and *Date* are unique and can occur at most once in each data stream in Example 1, and the values of the attribute *pid* do not repeat in Example 2. Also, in Example 3 and 4, the values of the attributes *sessionID* and *plateNo* do not (or rarely) repeat. If join attribute values are unique and each join attribute value occurs at most once in each data stream, the value-based algorithms (i.e., the frequency-based algorithms and output-based algorithms) cannot determine the priorities of tuples properly because they are based on the statistics of each join attribute value. However, as far as we know, there is no load shedding algorithms specially designed for environments where join attribute values are unique and each join attribute value occurs at most once in each data stream.

In this paper, we propose a load shedding algorithm for multi-way stream joins in environments where join attribute values are unique and each join attribute value occurs at most once in each data stream. Unlike the existing value-based algorithms, the proposed algorithm determines the priority of a tuple based on *the order of streams* in which its join attribute value appears, rather than its join attribute value itself. In many applications, join attribute values that appear in all the data streams, i.e., join attribute values that produce output tuples, show some common patterns in the order of streams in which they appear. For instance, in Example 1, news articles that first appear on a more prominent site (e.g., CNN.com) are more likely to appear on all the sites than those that first appear on a less prominent one (e.g., a regional news site). In other words, news articles that appear in all the sites have a tendency to appear in from more prominent sites. Also, in Example 2, packets that first appear in R_1 or R_4 have a higher probability of appearing in all the routers than those that first appear in R_2 or R_3. This is because R_1, R_2, R_3, and R_4 are located sequentially along a network path so that most packets appearing in all the routers appear in those routers in the order of $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4$ or $R_4 \rightarrow R_3 \rightarrow R_2 \rightarrow R_1$. As for Example 3, there are usually frequent sequential patterns in traversing Web pages, as studied in previous research (Chen et al. 1998; Nanopoulos et al. 2003). As a result, users visiting all the n Web pages usually show some common patterns in their visiting order. Similarly, in Example 4, vehicles passing all the n points usually exhibit some common patterns in the order of points they pass, mainly due to the structure of the underlying road network. In all of these cases, the priority of a tuple can be measured more effectively by using the order of streams in which its join attribute value appears, rather than its join attribute value itself.

Based on the above observations, we propose a load shedding algorithm that determines the priority of a tuple based on the order of streams in which its join attribute value appears. Consequently, the proposed load shedding algorithm has the following advantages: (1) In many real world applications where join attribute values are unique and never (or rarely) repeat, the proposed algorithm can determine the priorities of tuples more effectively than the value-based algorithms. (2) Because the proposed algorithm is based on the order of streams rather than actual join attribute values, it can make load shedding decisions very fast compared with the value-based algorithms, whose complexity depends on the size of the domain of join attribute values. A fast decision is very important for load shedding because a load shedding algorithm should handle very high data arrival rates. As will be described in Section 3, the computation for making load shedding decisions in the proposed algorithm involves only a few operations on a small-sized bit vector table.

To demonstrate the effectiveness and efficiency of the proposed algorithm, we have conducted extensive experiments with various parameters. The experimental results show that the proposed algorithm is very efficient in terms of the number of output tuples and processing time.

1.2 Paper outline

The paper is organized as follows: Section 2 describes the multi-way join processing model used in the paper and presents related work. Section 3 describes the proposed load shedding algorithm that is based on the order of streams in which join attribute values appear, and illustrates it with examples. Section 4 analyzes the space and time overhead of the proposed algorithm. Section 5 presents experimental evaluation of the proposed algorithm and Section 6 concludes the paper.

2 Preliminaries

2.1 System model

Figure 1 shows the multi-way join processing model used in the paper. Let $S_1, S_2, \ldots, S_n$ be n input streams. Let Q be an n-way join query over $S_1, S_2, \ldots, S_n$, i.e., $Q = S_1 \bowtie S_2 \bowtie \ldots \bowtie S_n$. For simplicity and without loss of generality, we assume that $S_1, S_2, \ldots, S_n$ are joined only on a single join attribute J. To process Q, a window W_i is defined for each input stream S_i $(1 \le i \le n)$. Each W_i contains tuples from S_i that satisfy the window condition (e.g., the tuples that have arrived within the last 10 minutes), and is allocated a fixed amount of memory. When a new tuple t_i from S_i arrives at W_i, all expired tuples are removed from $W_1, W_2, \ldots, W_n$ and t_i is inserted into W_i. Then, a join operation $W_1 \bowtie \ldots \bowtie W_{i-1} \bowtie t_i \bowtie W_{i+1} \bowtie \ldots \bowtie W_n$ is performed to produce output tuples and the result is fed to the output stream.

When a new tuple t_i from S_i arrives at W_i, if the memory allocated to W_i is full and W_i has no expired tuples, we cannot keep the entire tuples of W_i for providing the exact join result. In this case, a load shedding algorithm makes a decision which tuple(s) to drop and which to retain among tuples in W_i. To make such a decision, the algorithm determines the priority of each tuple in W_i based on some measure, and then drops the tuple(s) with the lowest priority. For a join query, the goal of a load shedding algorithm is to maximize the output rate of the query. Thus, a load

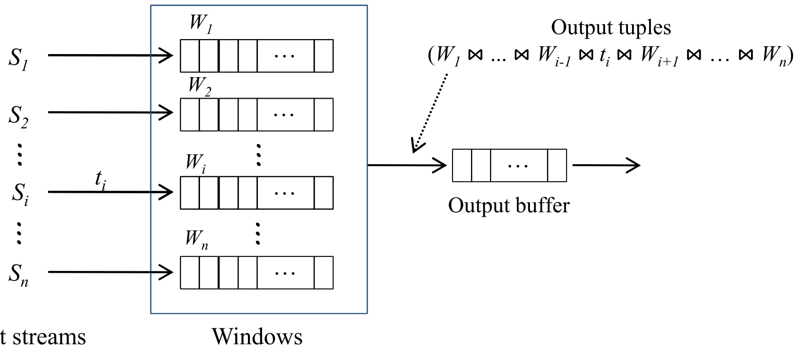

S_1
S_2
S_i
S_n
t_i
W_1
W_2
W_i
W_n
Output tuples
$(W_1 \bowtie \ldots \bowtie W_{i-1} \bowtie t_i \bowtie W_{i+1} \bowtie \ldots \bowtie W_n)$
Output buffer
Input streams
Windows

Fig. 1 n-way windowed join processing model shedding algorithm determines the priority of each tuple based on its productivity, i.e., the number of output tuples that will be produced by the tuple.

2.2 Related work

As described in Section 1, most existing load shedding algorithms can be classified into frequency-based algorithms and output-based algorithms. In Das et al. (2003), a load shedding decision is made in such a way as to keep the tuples with most frequent join attribute values or to keep the tuples the frequency of whose join attribute values multiplied by their remaining lifetime is highest. In Srivastava and Widom (2004), a solution was given to optimize the memory allocation between the windows of input streams based on the cumulative number of output tuples produced by tuples in each window. Both of them, however, focused on load shedding problem in binary stream joins.

For multi-way joins over multiple data streams, there has been relatively little work on finding load shedding policies. Recently, Gedik et al. (2007) have proposed a time correlation-aware CPU load shedding for multi-way stream joins. They divide each window into a number of segments and exploits time correlations among tuples in interrelated streams to prioritize the segments of the windows. Then, only segments with high priority are joined under CPU overloaded condition. Law and Zaniolo (2007) have proposed a frequency-based load shedding algorithm for multi-way joins under memory limited condition. They estimate the productivity of a tuple based on the frequency distribution of join attribute values. Because their algorithm is based on the actual distribution of join attribute values in windows, they also use a sketch technique (Law and Zaniolo 2007; Dobra et al. 2002) to minimize processing costs. Although the goal of Law and Zaniolo (2007) is similar to that of our work in that both consider memory limited condition in multi-way stream join processing, it differs from our approach in that the proposed load shedding algorithm exploits the order of streams in which join attribute values appear, rather than the frequency distribution of join attribute values, to determine the priorities of tuples.

Another closely related issue is the algorithm for processing multi-way stream joins. Recently, there have been many studies on algorithms for processing multi-way stream joins. When a multi-way stream join is evaluated by a tree of binary pipelined join operators, Golab and Ozsu (2003) have proposed join order heuristics to minimize the processing cost per unit time. Viglas et al. (2003) have proposed a single multi-way join operator called *MJoin* to maximize the output rate of multi-way stream joins, which generalizes the binary symmetric hash join operator to work for more than two inputs. In our previous work (Kwon et al. 2009), we have proposed a new multi-way join operator called *AMJoin* to improve the performance of MJoin by detecting *join failtures* efficiently. As described in Section 2.1, whenever a new tuple t_i from S_i arrives at W_i, a join operation $W_1 \bowtie \ldots \bowtie W_{i-1} \bowtie t_i \bowtie W_{i+1} \bowtie \ldots \bowtie W_n$ is performed to produce output tuples. If there is any W_j $(j \neq i)$ that does not contain a tuple with the same join attribute value as that of t_i, the join operation will produce no output tuples, i.e., *join failure*. To avoid unnecessary probes into W_j $(j \neq i)$ in cases of join failures, AMJoin maintains an in-memory table called *BiHT (Bit-vector Hash Table)*, in which each entry has the form (v, w), where v is a hash value and w is an n-bit vector. If a tuple whose join attribute value is hashed to v is currently in W_i $(1 \leq i \leq n)$, the ith bit of w is set to 1, otherwise 0. Then, by simply checking the corresponding entry in BiHT when a new tuple arrives at a window, we can detect a join failure immediately without probing the other windows. As a result, the performance of a multi-way stream join can be significantly improved.

In this paper, we adopt the framework of AMJoin. Let h be a given hash function. When a new tuple t_i from S_i arrives at W_i $(1 \leq i \leq n)$, the join attribute value of t_i, denoted by $t_i.J$, is hashed to a hash value v using the hash function h, i.e., $v = h(t_i.J)$. Let (v, w) be the entry in BiHT that corresponds to the hash value v. After t_i is inserted into W_i, the ith bit of w is set to 1. Then, only when all bits of w is 1, $W_1 \bowtie \ldots \bowtie W_{i-1} \bowtie t_i \bowtie W_{i+1} \bowtie \ldots \bowtie W_n$ is performed. Otherwise, the join operation is not performed.

3 Load shedding algorithm based on arrival order patterns

In this section, we describe the proposed load shedding algorithm based on a new measure of the priorities of tuples.

3.1 Priorities of tuples

Unlike the existing value-based algorithms, which determine the priority of a tuple based on the frequency or some historical statistics of its join attribute value, the proposed load shedding algorithm determines the priority of a tuple based on the order of streams in which its join attribute value appears in a given time window.

When a new tuple t_i arrives at a window W_i $(1 \leq i \leq n)$, we associate t_i with an n-bit vector, called an *existence pattern*, which is defined as follows:

Definition 1 (Existence pattern) Let t_i be a tuple that arrives at W_i at time T. Let h be a given hash function. The existence pattern of t_i, denoted by $ep(t_i)$, is an n-bit vector, where the j-th bit $(1 \leq j \leq n)$ is 1 if there exists a tuple t_j in W_j at time T such that $h(t_i.J) = h(t_j.J)$ and 0 otherwise.

In other words, $ep(t_i)$ indicates windows in which a tuple with the same hashed join attribute value as that of t_i appears when t_i arrives at W_i. Note that, for tuples t_i arriving at W_i, the ith bit of $ep(t_i)$ is always 1. Figure 2 shows examples of existence patterns for a 3-way stream join, where a tuple in a window is represented by its hashed join attribute value. In Fig. 2a, when a new tuple t_1 with hashed join attribute value 5 (i.e., $h(t_1.J) = 5$) arrives at W_1, a tuple with the same hashed join attribute value as that of t_1 does not appear in either W_2 or W_3. Thus, $ep(t_1) =$ '100'. In Fig. 2b, when a new tuple t_3 with hashed join attribute value 4 (i.e., $h(t_3.J) = 4$) arrives at W_3, a tuple with the same hashed join attribute value as that of t_3 appears in W_1. Thus, $ep(t_3) =$ '101'.

For tuples arriving at a window W_i $(1 \leq i \leq n)$, there are 2^{n-1} possible existence patterns. For example, in Fig. 2, for tuples arriving at W_1, there are $2^{3-1} = 4$ possible existence patterns, i.e., '100', '101', '110', and '111'. Similarly, for tuples arriving at W_2, possible existence patters are '010', '011', '110', and '111', and for tuples arriving at W_3, possible existence patters are '001', '011', '101', and '111'.

Whenever a new tuple t_i arrives at a window W_i, we associate t_i with its existence pattern $ep(t_i)$. Then, for each window W_i $(1 \leq i \leq n)$, we maintain an *existence pattern*

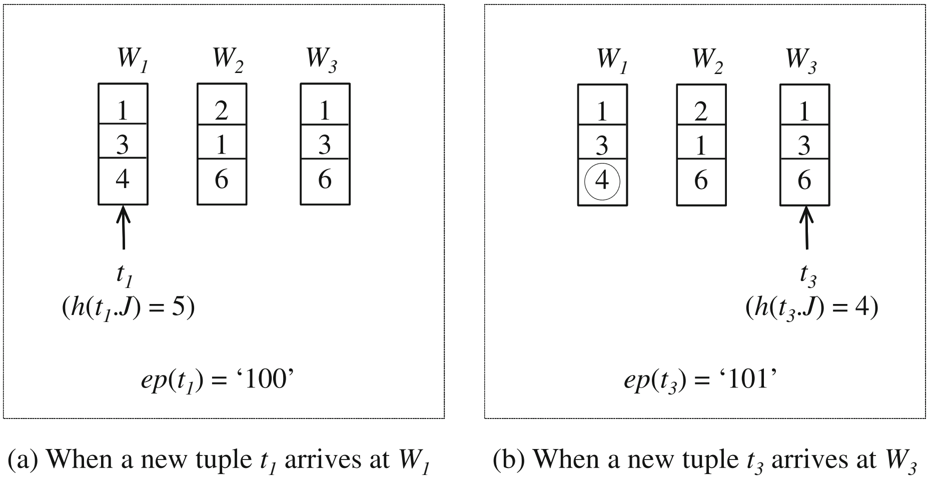


Fig. 2 Examples of existence patterns

table to keep track of tuples with the same existence pattern. An existence pattern table is defined as follows:

Definition 2 (Existence pattern table) An existence pattern table for a window W_i ($1 \leq i \leq n$), denoted by EPT_i, is a table consisting of 2^{n-1} entries, one for each possible existence pattern for tuples arriving at W_i. Each entry in EPT_i has the form $(e, l_i(e), r_i(e), n_i(e))$, where e is an existence pattern, $l_i(e)$ is a list of pointers to tuples in W_i whose existence pattern is e, $r_i(e)$ is the cumulative number of output tuples produced by tuples in W_i whose existence pattern is e, and $n_i(e)$ is the cumulative number of tuples in W_i whose existence pattern is e.

Figure 3 shows examples of existence pattern tables for W_1, W_2, and W_3 in Fig. 2. Note that an existence pattern table EPT_i is maintained for each window W_i. Now we define the priority of a tuple as follows:

Definition 3 (The priority of a tuple) Let t be a tuple in a window W_i ($1 \leq i \leq n$). The priority of t, denoted as $\pi(t)$, is defined as

$$\pi(t) = \begin{cases} 0 & \text{if all bits of } ep(t) \text{ are 1} \\ r_i(ep(t))/n_i(ep(t)) & \text{otherwise.} \end{cases}$$

EPT_1

e	$l_1(e)$	$r_1(e)$	$n_1(e)$
100	$t_{11}, \ldots$	8	10
101	$t_{12}, \ldots$	6	10
110	$t_{13}, \ldots$	8	20
111	$t_{14}, \ldots$	10	10

EPT_2

e	$l_2(e)$	$r_2(e)$	$n_2(e)$
010	$t_{21}, \ldots$	12	30
011	$t_{22}, \ldots$	8	10
110	$t_{23}, \ldots$	2	20
111	$t_{24}, \ldots$	7	7

EPT_3

e	$l_3(e)$	$r_3(e)$	$n_3(e)$
001	$t_{31}, \ldots$	8	10
011	$t_{32}, \ldots$	4	10
101	$t_{33}, \ldots$	1	10
111	$t_{34}, \ldots$	5	5

Fig. 3 Example of existence pattern tables

In other words, for a window W_i, we define the priority of a tuple t as the average number of output tuples produced by a tuple with the same existence pattern as that of t. For example, in Fig. 3, the priority of t_{22} in W_2 whose existence pattern is ‘011’ is computed as $\pi(t_{22}) = r_2(‘011') / n_2(‘011') = 0.8$. Note that we assign the lowest priority (0) to tuples with the existence pattern of which all bits are 1. This is because they have already participated in join operations to produce output tuples and, when join attribute values are unique and do not repeat, those tuples that have produced output tuples will not produce any more output tuples.

Because we determine the priority of a tuple based on its existence pattern, rather than its actual join attribute value, we can make better load shedding decisions in environments where join attribute values are unique and do not repeat. In the next section, we describe our load shedding algorithm based on the proposed measure of the priorities of tuples.

3.2 Load shedding algorithm

Our load shedding algorithm is built upon the framework of AMJoin, as mentioned in Section 2.1. We assume that BiHT and EPT_i $(1 \le i \le n)$ are maintained in memory and a hash function h is given. Whenever a new tuple t_i from S_i arrives at W_i, the following three procedures are consecutively triggered to perform the n-way stream join:

1. *Hash*: The join attribute value of t_i is hashed to a hash value v using the hash function h, i.e., $v = h(t_i.J)$.
2. *Insertion*:
   (a) If the memory allocated to W_i is full and W_i has no expired tuples, $l_i(\epsilon)$ is checked whether it is empty, where ϵ is the existence pattern of which all bits are 1. If $l_i(\epsilon)$ is not empty, the oldest tuple among those in $l_i(\epsilon)$ is dropped from W_i. Otherwise, if $l_i(\epsilon)$ is empty, the oldest tuple among those in $l_i(e)$ is dropped from W_i, where e is the existence pattern with the lowest $r_i(e)/n_i(e)$ and $l_i(e)$ not empty. Then, t_i is inserted into W_i.
   (b) Otherwise, if the memory allocated to W_i is not full, t_i is simply inserted into W_i.
   (c) Let (v, w) be the entry in BiHT that corresponds to the hash value v. The ith bit of w is set to 1 and w becomes the existence pattern of t_i, i.e., $ep(t_i) = w$. Then, a pointer to t_i is inserted into $l_i(ep(t_i))$ and $n_i(ep(t_i))$ is increased by one.
3. *Join*: If all bits of w are 1, a join operation $W_1 \bowtie \ldots \bowtie W_{i-1} \bowtie t_i \bowtie W_{i+1} \bowtie \ldots \bowtie W_n$ is performed and the result is fed to the output stream. Otherwise, the join operation is not performed. If the above join operation produces any output tuple, for each input tuple t_j in W_j $(1 \le j \le n)$ that contributes to the output tuple, $r_j(ep(t_j))$ in EPT_j is increased by one.

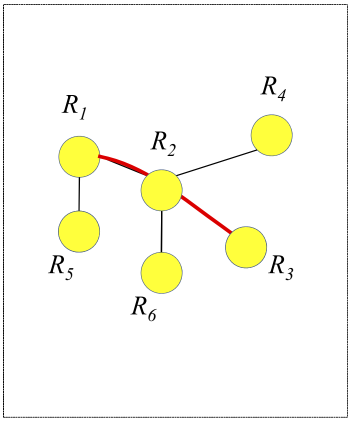
Note that, in the procedure *Insertion* (a), we drop the oldest tuple among those in $l_i(e)$ (or $l_i(\epsilon)$) from W_i. Because all tuples in $l_i(e)$ (or $l_i(\epsilon)$) have the same priority, we drop the oldest tuple among them for load shedding.

3.3 Example

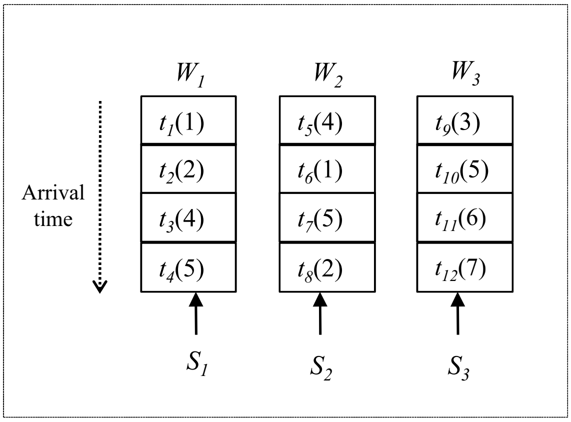
In this section, we present examples of the proposed load shedding algorithm. Consider a network with six routers $R_1, R_2, \ldots, R_6$ as shown in Fig. 4a. Each router R_i $(1 \leq i \leq 6)$ feeds a data stream S_i in which each tuple contains information about a packet passing through the router. Each tuple has the identifier of the packet (*pid*). Suppose that, in order to count packets passing through all of R_1, R_2, and R_3, we are running a 3-way join query over S_1, S_2, and S_3 on *pid*.

Suppose that Fig. 4b shows the current windows of S_1, S_2 and S_3, denoted by W_1, W_2, and W_3, respectively. In Fig. 4b, $t_i(v)$ represents a tuple t_i whose join attribute value is v. In our example, because R_1, R_2, and R_3 are located sequentially in the network, most join attribute values (i.e., *pid* values) that appear in all of W_1, W_2, and W_3 appear in them in the order of $W_1 \rightarrow W_2 \rightarrow W_3$ or $W_3 \rightarrow W_2 \rightarrow W_1$. Assume that the maximum number of tuples in the memory allocated to each window is 4. Assume also that the join attribute values appearing in W_1, W_2, and W_3 in Fig. 4b have never occurred before. For simplicity of presentation, we assume that $h(x) = x$.

Case 1 If a new tuple from S_1 arrives at W_1 and W_1 has no expired tuples, the proposed load shedding algorithm drops t_4 from W_1. Note that $ep(t_1)$ = '100' (which means that the join attribute value of t_1 (i.e., 1) first appeared in W_1 before appearing in other windows), $ep(t_2)$ = '100' (which means that the join attribute value of t_2 (i.e., 2) first appeared in W_1 before appearing in other windows), $ep(t_3)$ = '110' (which means that the join attribute value of t_3 (i.e., 4) appeared in W_1 after first appearing in W_2), and $ep(t_4)$ = '111' (which means that the join attribute value of t_4 (i.e., 5) appeared in all windows when t_4 arrived at W_1). Because $ep(t_4)$ = '111', t_4 has already participated in a join operation to produce an output tuple and will not produce any more output tuples. Thus, the proposed algorithm assigns t_4 the lowest priority and drops t_4 from W_1. On the other hand, the frequency-based and output-based algorithms assign t_4 the highest priority among those in W_1, because 5 is the most frequent join attribute value in windows and 5 is the only join attribute value



(a) Example network



(b) Current windows of S_1, S_2 and S_3

Fig. 4 Example of the proposed load shedding algorithm

that has produced any output tuple among those in W_1, which is obviously wrong. The frequency-based and output-based algorithms treat t_1, t_2, and t_3 in W_1 with the same priority because their join attribute values (i.e., 1, 2, and 4, respectively) have the same frequency in windows and none of them have produced output tuples, and drop one of t_1, t_2, and t_3 from W_1. However, the proposed algorithm distinguishes them by assigning t_3 a lower priority than t_1 and t_2. Because the join attribute value of t_3 (i.e., 4) appeared in windows in the order of $W_2 \rightarrow W_1$ (i.e., $ep(t_3) =$ '110'), it is more unlikely to appear in all of W_1, W_2, and W_3 than those of t_1 and t_2 (i.e., 1 and 2, respectively), which first appeared in W_1 before appearing in other windows (i.e., $ep(t_1) = ep(t_2) =$ '100'). Thus, the proposed algorithm assigns t_3 a lower priority than t_1 and t_2.

Case 2 If a new tuple from S_2 arrives at W_2 and W_2 has no expired tuples, the proposed load shedding algorithm drops t_5 from W_2. Note that $ep(t_5) =$ '010' (which means that the join attribute value of t_5 (i.e., 4) first appeared in W_2 before appearing in other windows), $ep(t_6) =$ '110' (which means that the join attribute value of t_6 (i.e., 1) appeared in W_2 after first appearing in W_1), $ep(t_7) =$ '011' (which means that the join attribute value of t_7 (i.e., 5) appeared in W_2 after first appearing in W_3), and $ep(t_8) =$ '110' (which means that the join attribute value of t_8 (i.e., 2) appeared in W_2 after first appearing in W_1). Because the join attribute value of t_5 (i.e., 4) first appeared in W_2 before appearing in other windows (i.e., $ep(t_5) =$ '010'), it is more unlikely to appear in all of W_1, W_2, and W_3 than those of t_6, t_7, and t_8 (i.e., 1, 5, and 2, respectively), which appeared in windows in the order of $W_1 \rightarrow W_2$ or $W_3 \rightarrow W_2$ (i.e., $ep(t_6) = ep(t_8) =$ '110' and $ep(t_7) =$ '011'). Thus, the proposed algorithm assigns t_5 the lowest priority and drops t_5 from W_2. On the other hand, the frequency-based and output-based algorithms, similarly to the above case, assign t_7 the highest priority among those in W_2 and drop one of t_5, t_6, or t_8 from W_2, treating them with the same priority.

From the above two cases, we can observe that the behavior of the frequency-based and output-based algorithms becomes similar to random selection when join attribute values are unique and do not repeat. On the contrary, the proposed algorithm can determine the priorities of tuples more effectively by using their arrival order patterns.

4 Analysis of the proposed algorithm

In this section, we analyze the space and time overhead of the proposed load shedding algorithm. First, we analyze the space overhead of the proposed algorithm. The proposed algorithm maintains BiHT and EPT_i $(1 \leq i \leq n)$ in memory. Let H be the number of possible hash values of the hash function h. For each entry (v, w) in BiHT, we use 4 bytes for a hash value v and 2 bytes for a bit vector w. Thus, the total size of BiHT is $H \cdot (4 + 2) = 6 \cdot H$ bytes. Recall that the number of entries in each EPT_i is 2^{n-1}. For each entry $(e, l_i(e), r_i(e), n_i(e))$ in EPT_i, we use 2 bytes for a bit vector e, 4 bytes for a value $r_i(e)$, and 4 bytes for a value $n_i(e)$. $l_i(e)$ contains pointers to tuples in W_i whose existence pattern is e. Let M be the maximum number of tuples in the memory allocated to W_i. If we use 2 bytes for a pointer to a tuple in W_i, the total size of all pointers in EPT_i is $2 \cdot M$ bytes at most. Thus, the total size of EPT_i is $(2^{n-1} \cdot (2+4+4) + 2 \cdot M)$ bytes at most. Since there are n such existence pattern tables, the total space overhead of the proposed algorithm is

$$(6 \cdot H + n \cdot (10 \cdot 2^{n-1} + 2 \cdot M))\ bytes. \tag{1}$$

We compare the space overhead of the proposed algorithm with that of the value-based algorithms. In the value-based algorithms, we need to maintain a table T_i for each window W_i $(1 \leq i \leq n)$, in which each entry contains some statistics of a join attribute value that has occurred in W_i. Assume that each entry in T_i $(1 \leq i \leq n)$ has the form (v, l, s), where v is a join attribute value, l is a list of pointers to tuples in W_i whose join attribute value is v, and s is a certain statistics of v (e.g., the number of output tuples produced by tuples with join attribute value v). Let D be the size of the domain of join attribute values. If we use 4 bytes for v, 4 bytes for s, and 2 bytes for a pointer to a tuple in W_i, the total size of T_i is $(D \cdot (4+4) + 2 \cdot M)$ bytes at most. Since there are n such tables $T_1, T_2, \ldots, T_n$, the total space overhead of the value-based algorithms is approximately

$$n \cdot (8 \cdot D + 2 \cdot M)\ bytes. \tag{2}$$

When $n = 5$, $H = 10^6$, and $D = 10^8$, the space overhead of the proposed algorithm is about $(10 \cdot M + 6 \cdot 10^6)$ bytes, while that of the value-based algorithms is about $(10 \cdot M + 4 \cdot 10^9)$ bytes. Although each EPT_i has 2^{n-1} entries, the proposed algorithm does not require much space overhead compared to the value-based algorithms, because n is not very large in practice (i.e., less than 10).

Now we analyze the time complexity of the proposed algorithm. Whenever a new tuple arrives at a window, the proposed algorithm performs the three procedures described in Section 3.2. First, the time required for performing the procedure *Hash* is constant. In the procedure *Insertion* (a), we need to find the existence pattern e with the lowest $r_i(e)/n_i(e)$ in EPT_i. To find such e efficiently, we maintain entries in EPT_i in a priority queue in increasing order of $r_i(e)/n_i(e)$. With a priority queue maintained in increasing order of $r_i(e)/n_i(e)$, the time for finding e with the lowest $r_i(e)/n_i(e)$ in EPT_i is constant. The time for finding the oldest tuple in $l_i(e)$ is also constant if $l_i(e)$ is maintained in a FIFO queue. In the procedure *Insertion* (c), we need to update BiHT and EPT_i. The time for updating BiHT is constant and the time for updating EPT_i is $O(\log_2 2^{n-1}) = O(n)$ because 2^{n-1} entries in EPT_i are maintained in a priority queue in increasing order of $r_i(e)/n_i(e)$. In the procedure *Join*, a join operation is performed and then each EPT_i $(1 \leq i \leq n)$ is updated if there is any output tuple produced by the join operation. We do not consider the time for performing the join operation in this discussion. Because there are n existence pattern tables, the time for updating n existence pattern tables is $O(n \cdot n) = O(n^2)$. Therefore, the overall time complexity of the proposed algorithm for processing a tuple is $O(n^2)$. Note that the time complexity of the proposed algorithm is $O(n)$ when a newly arrived tuple produces no output tuple, which is mostly the case. Because n is not very large in practice, the proposed load shedding algorithm can be very fast compared to the value-based algorithms, whose complexity depends on D, as will be demonstrated in Section 5.

5 Experiments

In this section, we show various experimental results to clarify the performance of the proposed load shedding algorithm. We measure the performance of a load shedding algorithm in terms of the number of output tuples produced by the algorithm. We compare the proposed algorithm, denoted by *EP*, with the following algorithms:

- *Frequency-based*: When the memory allocated to a window is full, a tuple with the join attribute value whose frequency in windows is the lowest is dropped from the window.
- *Output-based*: When the memory allocated to a window is full, a tuple with the join attribute value that has produced the least number of output tuples is dropped from the window.
- *Random*: When the memory allocated to a window is full, a randomly selected tuple is dropped from the window.

Note that the frequency-based and output-based algorithms do not assume that join attribute values are unique and do not repeat, while the proposed algorithm is specially designed for environments where such assumptions hold. However, as mentioned in Section 1, as far as we know, there is no load shedding algorithms specially designed for such environments. For this reason, we have compared the proposed algorithm with the two algorithms because they are the two most popular load shedding algorithms known so far. Hence, the objective of our experiments is to show that (1) the existing known algorithms are not very effective in environments where join attribute values are unique and do not repeat and (2) the proposed algorithm performs better than the existing algorithms in such environments.

In order to verify the performance of the proposed algorithm under various conditions, we have developed a data generator that produces synthetic datasets with varying parameters such as the number of tuples per stream, the number of input streams, the distribution of join attribute values, the arrival rate of tuples, etc. Each dataset consists of a set of tuples, each of which has the form (*stream_id*, *key_value*, *arrival_time*, *misc*), where *stream_id* is the identifier of the input stream from which the tuple arrives, *key_value* is a join attribute value, *arrival_time* is the arrival time of the tuple at the window, and *misc* is miscellaneous information. In each experiment, we read tuples from a dataset and send them to the system according to their *arrival_time*. We then count the number of output tuples produced by the system and measure the time taken for processing input tuples.

Table 1 shows the experimental parameters used in the experiments. Here, *Memory size* represents the size of the memory allocated to each window in terms of the number of tuples. *Arrival rate per stream* represents the arrival rate of tuples from each input stream. When we generate a dataset, we generate join attribute values first. Then, for each generated join attribute value, we randomly assign the order of streams (i.e., windows) in which the join attribute value appears (e.g., $W_1 \rightarrow W_2 \rightarrow W_3$, $W_2 \rightarrow W_1$, $W_4 \rightarrow W_2 \rightarrow W_1 \rightarrow W_3$) using a Zipf distribution. Note that a join attribute value does not necessarily appear in all streams. As the *Zipf skew factor* α increases, more join attribute values are assigned the same order of streams in which they appear. When α is 0.0, the Zipf distribution reduces to a uniform distribution.

We have implemented all the load shedding algorithms in Java and conducted experiments on an Intel Core 2 Duo 2.66 GHz machine running Windows XP with 4

Table 1 Experimental parameters

Parameters	Values
The number of input streams	3, 4, 5, 6, 7
The number of tuples per stream	10,000, 20,000, 40,000, 80,000, 160,000
Memory size	100, 200, 300, 400, 500 (tuples)
Arrival rate per stream	10.0, 12.5, 16.7, 25.0 (tuples/s)
Zipf skew factor (α)	0.0, 0.5, 1.0, 1.5, 2.0
Window condition	The tuples that have arrived within the last 100 s

GB memory. In all the experiments, except the last one, we consider an environment where join attribute values are unique and do not repeat. However, in the last experiment, we also investigate the performance of the proposed load shedding algorithm when the distribution of join attribute values follows a normal distribution.

5.1 Effect of memory size

In this experiment, we compare the performance of the load shedding algorithms by varying the memory size allocated to each window. Figure 5 shows the performance of the four algorithms for different memory sizes. The number of input streams is 5, the number of tuples per stream is 10,000, the arrival rate per stream is 10 tuples/s, and the Zipf skew parameter is 0.0. We vary the memory size allocated to each window from 100 to 500 in terms of the number of tuples. In Fig. 5, the performance of all algorithms increases as the memory size increases, because less tuples are dropped from windows as the memory size increases. However, the proposed algorithm base on arrival order patterns (*EP*) significantly outperforms the other algorithms for all memory sizes, while the other algorithms show almost the same performance. This is because, when each join attribute value occurs at most once in each input stream, only the proposed algorithm can intelligently determine which tuples to drop based on their existence patterns, while the frequency-based

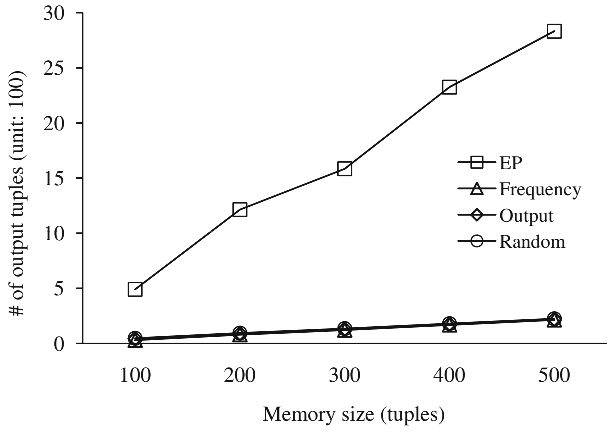


Fig. 5 Performance of the load shedding algorithms for different memory sizes

and output-based algorithms treat most tuples with the same priority. As a result, the performance of the frequency-based and output-based algorithms becomes similar to that of the random algorithm.

5.2 Effect of skewness

In this experiment, we investigate the performance of the load shedding algorithms by varying the Zipf skew factor α. As α increases, the distribution of existence patterns becomes more skewed. Figure 6 shows the result of the experiment with varying α. The number of input streams is 5, the number of tuples per stream is 10,000, the memory size allocated to each window is 500 tuples, and the arrival rate per stream is 10 tuples/s. We vary α from 0.0 to 2.0. As shown in Fig. 6, the performance of the proposed algorithm improves as α increases. This is because, as the distribution of existence patterns becomes more skewed, the proposed algorithm can determine more clearly which tuples are likely to produce output tuples and which are not by using their existence patterns. However, the distribution of existence patterns does not affect the performance of the other algorithms so that their performance remains almost the same in Fig. 6.

5.3 Effect of arrival rate

In this experiment, we evaluate the performance of the load shedding algorithms by varying the arrival rate per stream. We set the arrival rate per stream to 10.0, 12.5, 16.7, and 25.0 tuples/s. The number of input streams is 5, the number of tuples per stream is fixed to 10,000, the memory size allocated to each window is 500 tuples, and the Zipf skew factor is 0.0. Figure 7 show the number of output tuples produced by four algorithms with varying arrival rate per stream. Also in this case, the proposed algorithm outperforms the other algorithms for all arrival rates. Note that the performance of the proposed algorithm improves as the arrival rate increases to 16.7 and degrades slightly when the arrival rate is greater than 16.7. Because only the proposed algorithm can intelligently determine which tuples to drop in an environment where join attribute values are unique and do not repeat, the number of output tuples produced by the proposed algorithm increases as the arrival rate

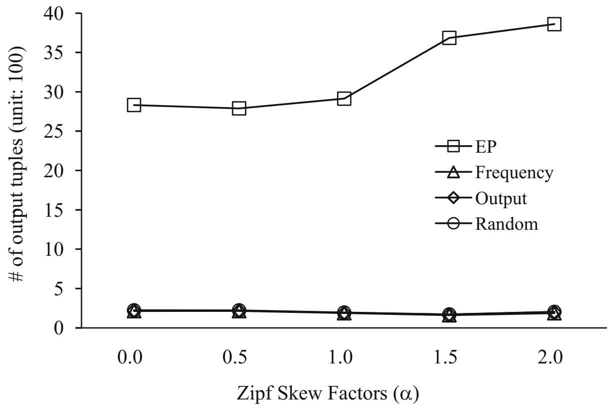


Fig. 6 Performance of the load shedding algorithms by varying the Zipf skew factor

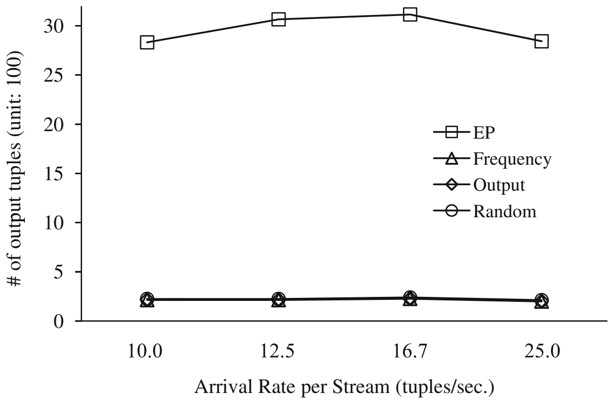


Fig. 7 Performance of the load shedding algorithms by varying the arrival rate per stream

increases to 16.7. However, when the arrival rate becomes greater than 16.7, the arrival rate exceeds the processing rate of the algorithm, resulting in performance degradation. On the other hand, the number of output tuples produced by the other algorithms remains almost the same regardless of the arrival rate, because all of them behave like a random selection for a fixed number of tuples per stream (i.e., 10,000).

5.4 Effect of the number of input streams

In order to study the effect of the number of input streams on the performance of the load shedding algorithms, we vary the number of input streams in this experiment. Figure 8 shows the performance of the load shedding algorithms when we vary the number of input streams from 3 to 7. The number of tuples per stream is 10,000, the memory size allocated to each window is 500 tuples, the arrival rate per stream is 10 tuples/s, and the Zipf skew factor is 0.0. Note that because the number of tuples per stream is fixed to 10,000, the number of output tuples does not necessarily increase as the number of input streams increases. In Fig. 8, we can see that the performance advantage of the proposed algorithm over the other algorithms is not much affected by the number of input streams. Even when the number of input streams increases,

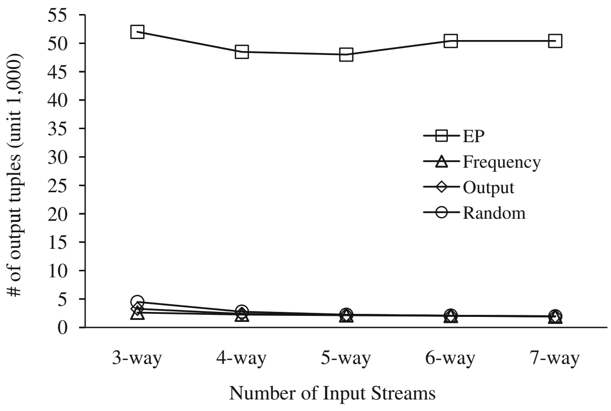


Fig. 8 Effect of the number of input streams

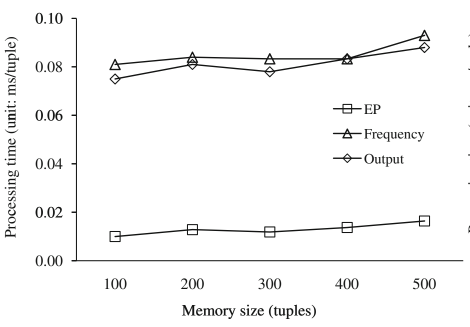
the proposed algorithm can effectively determine which tuples to drop by using their existence patterns.

5.5 Processing speed

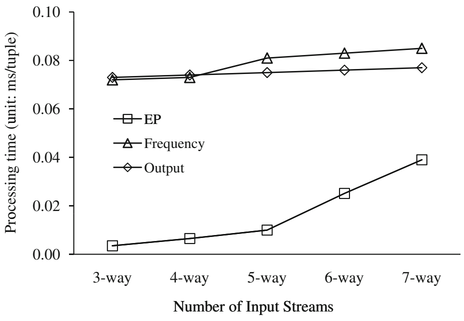
As mentioned in Section 1, the processing speed of load shedding algorithms is very important because load shedding algorithms should handle very high tuple arrival rates. In this experiment, we measure the processing time (for making load shedding decisions and updating the data structures) spent by each load shedding algorithm. We omit the random algorithm from this experiment as it does not perform any significant operations for load shedding.

Figure 9a shows the average processing time per input tuple for the algorithms as the memory size increases. The number of input streams is 5, the number of tuples per stream is 80,000, the arrival rate per stream is 10 tuples/s, and the Zipf skew parameter is 0.0. We vary the memory size allocated to each window from 100 to 500 tuples. As shown in Fig. 9a, the proposed algorithm takes much less time to process each input tuple, while the frequency-based and output-based algorithms show similar processing time. Because the number of entries in EPT_i is very small (only 16 when the number of input streams is 5), the proposed algorithm can be very fast to make load shedding decisions and update EPT_i. On the other hand, the processing time of the frequency-based and output-based algorithms depends on the size of the domain of join attribute values. More specifically, as described in Section 4, the number of entries in T_i used in the frequency-based and output-based algorithms is proportional to the size of the domain of join attribute values D, which affects the table lookup and update cost. Thus, the proposed algorithm does not require much time overhead compared with the value-based algorithms.

Figure 9b shows the average processing time per input tuple for the algorithms as the number of input streams increases. The number of tuples per stream is 80,000, the memory size allocated to each window is 500 tuples, the arrival rate per stream is 10 tuples/s, and the Zipf skew factor is 0.0. We vary the number of input streams from 3 to 7. As analyzed in Section 4, the processing time of the proposed algorithm increases as the number of input streams increases, mainly because the number of entries in EPT_i increases. On the other hand, the processing time of the other



(a) Average processing time per input tuple for the algorithms as the memory size increases



(b) Average processing time per input tuple for the algorithms as the number of input streams increases

Fig. 9 Processing speed of the load shedding algorithms

algorithms is not much affected by the number of input streams. However, even when the number of input streams is 7, the proposed algorithm takes much less processing time than the other algorithms. Note that the number of input streams is not very large in practice (i.e., less than 10).

5.6 Normal distribution

Until now we consider an environment where join attribute values are unique and each join attribute value occurs at most once in each input stream. In this experiment, we examine the performance of the load shedding algorithms when the distribution of join attribute values follows a normal distribution. Figure 10 show the number of output tuples produced by the algorithms when the distribution of join attribute values follows a normal distribution with mean 100. In Fig. 10, we vary the variance of the normal distribution from 15 to 35. The number of input streams is 5, the number of tuples per stream is 10,000, the memory size allocated to each window is 500 tuples, the arrival rate per stream is 10.0 tuples/s, and the Zipf skew factor is 0.0. As expected, the performance of the proposed algorithm is worse than that of the frequency-based and output-based algorithms when the distribution of join attribute values follows a normal distribution. As the variance becomes smaller, the performance of the frequency-based and output-based algorithms gets better than that of the proposed algorithm. This is because, as the variance becomes smaller, more join attribute values are concentrated around the mean so that the frequency-based and output-based algorithms keep in windows tuples with join attribute values that are near the mean, which produce more output tuples. However, as the variance becomes larger, the performance of the frequency-based and output-based algorithms degrades because join attribute values are more spread out from the mean. Note that the performance of the value-based algorithms and the proposed algorithm becomes similar as the variance increases.

Finally, note also that the proposed algorithm assumes an environment where each join attribute value is unique, while the value-based algorithms do not. As expected, the proposed algorithm does not perform better than the value-based algorithms in this experiment because join attribute values are not unique. Because

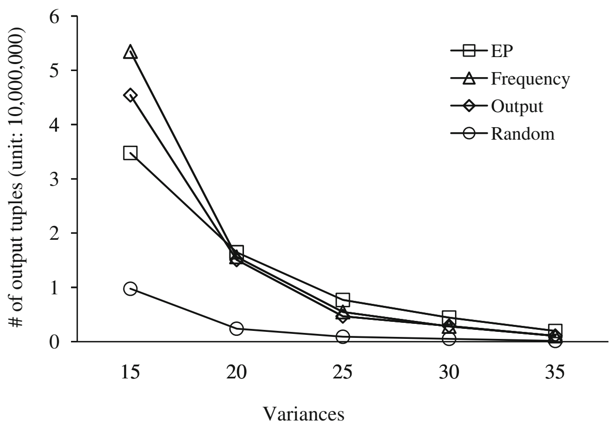


Fig. 10 Performance of the load shedding algorithms when the distribution of join attribute values follows a normal distribution

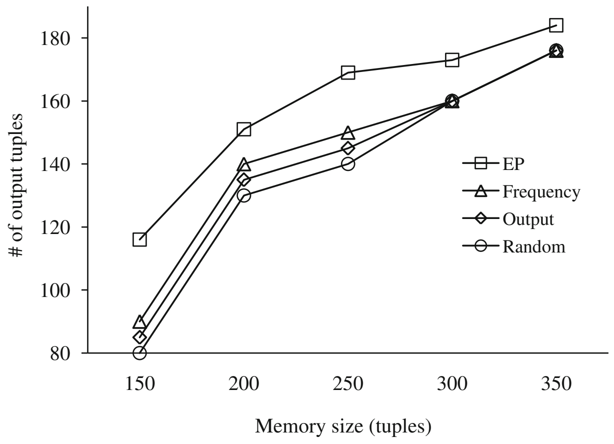


Fig. 11 Performance of the load shedding algorithms on real data

the proposed algorithm and the value-based algorithms are targeting on different environments, we should choose an appropriate load shedding algorithm depending on the environment.

5.7 Real data

Finally, we evaluate the performance of the load shedding algorithms on real data. For real data, we use a real Web server trace, called the *ClarkNet-HTTP*, available from the site http://ita.ee.lbl.gov/html/traces.html. Each line in the trace contains information about an HTTP request to a Web page, which has the form (*host*, *timestamp*, *request*, *HTTP reply code*, *bytes in the reply*), where *host* is the host (i.e., hostname or IP address) making the request. We select four Web pages, namely, *factinfo*, *main*, *search*, and *signup*, and generate a data stream for each Web page, in which each tuple contains information a request to the Web page. We then run a 4-way join query over the four data streams on *host* to count the number of hosts visiting all the four Web pages. In this real data, hosts visiting all the four Web pages are found to have a tendency to visit *signup* first.

Figure 11 shows the performance of the algorithms on the real data by varying the memory size allocated to each window. We vary the memory size allocated to each window from 150 to 350 in terms of the number of tuples. As in the case of the synthetic data, the proposed algorithm outperforms the other algorithms for all memory sizes. Because the proposed algorithm performs load shedding intelligently based on the order of Web pages hosts visit, the proposed algorithm can achieve better performance than all the other algorithms. On the other hand, the other algorithms show similar performance.

6 Conclusions

We have presented a load shedding algorithm for multi-way stream joins based on *arrival order patterns*, when join attribute values are unique and each join attribute value occurs at most once in each data stream. In order to determine which tuples to drop, the proposed algorithm exploits the order of streams in which their join attribute values appear. To do this, we associate each tuple with its *existence pattern*, which indicates windows in which its join attribute value appears when it arrives at a window, The proposed algorithm then keeps track of the productivity of each existence pattern, i.e., the average number of output tuples produced by tuples with that existence pattern. When the memory allocated to a window is full, the proposed algorithm drops from the window a tuple with the least productive existence pattern. As a result, the proposed load shedding algorithm has the following advantages: (1) The proposed algorithm can determine the priorities of tuples more effectively than the value-based algorithms in environments where join attribute values are unique and do not repeat. (2) The proposed algorithm can make load shedding decisions very fast and incurs a small overhead compared with the value-based algorithms. Through experiments with various parameters, we have demonstrated the performance and efficiency of the proposed algorithm.

Acknowledgement This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0018865).

References

Bai, Y., Wang, H., & Zaniolo, C. (2007). Load shedding in classifying multi-source streaming data: A bayes risk approach. In: *Proceedings of the 7th SIAM international conference on data mining* (pp. 425–430).

Chen, M. S., Park, J. S., & Yu, P. S. (1998). Efficient data mining for path traversal patterns. *IEEE Transaction on Knowledge and Data Engineering, 10*(2), 209–221.

Cranor, C. D., Johnson, T., Spatscheck, O., & Shkapenyuk, V. (2003). Gigascope: A stream database for network applications. In: *Proceedings of the 2003 ACM SIGMOD international conference on management of data* (pp. 647–651).

Das, A., Gehrke, J., & Riedewald, M. (2003). Approximate join processing over data streams. In: *Proceedings of the 2003 ACM SIGMOD international conference on management of data* (pp. 40–51).

Dobra, A., Garofalakis, M. N., Gehrke, J., & Rastogi, R. (2002). Processing complex aggregate queries over data streams. In: *Proceedings of the 2002 ACM SIGMOD international conference on management of data* (pp. 61–72).

Gedik, B., Wu, K. L., Yu, P. S., & Liu, L. (2007). A load shedding framework and optimizations for m-way windowed stream joins. In: *Proceedings of the 23rd IEEE international conference on data engineering* (pp. 536–545).

Gehrke, J., & Madden, S. (2004). Query processing in sensor networks. *IEEE Pervasive Computing, 3*(1), 46–55.

Golab, L., & Ozsu, M. T. (2003). Processing sliding window multi-joins in continuous queries over data streams. In: *Proceedings of the 29th international conference on very large data bases* (pp. 500–511).

Hammad, M. A., Aref, W. G., & Elmagarmid, A. K. (2003). Stream window join: Tracking moving objects in sensor-network databases. In: *Proceedings of 15th international conference on scientific and statistical database management* (pp. 75–84).

Kwon, T. H., Kim, H. G., Kim, M. H., & Son, J. H. (2009). Amjoin: An advanced join algorithm for multiple data streams using a bit-vector hash table. *IEICE Transaction on Information and Systems, E92-D*(7), 1429–1434.

Law, Y. N., & Zaniolo, C. (2007). Load shedding for window joins on multiple data streams. In: *Proceedings of the 23rd IEEE international conference on data engineering* (pp. 674–683).

Nanopoulos, A., Katsaros, D., & Manolopoulos, Y. (2003). A data mining algorithm for generalized web prefetching. *IEEE Transaction on Knowledge and Data Engineering, 15*(5), 1155–1169.

Srivastava, U., & Widom, J. (2004). Memory-limited execution of windowed stream joins. In: *Proceedings of the 30th international conference on very large data bases* (pp. 324–335).

Viglas, S., Naughton, J. F., & Burger, J. (2003). Maximizing the output rate of multi-way join queries over streaming information sources. In: *Proceedings of the 29th international conference on very large data bases* (pp. 285–296).

Yu, H., Lim, E. P., & Zhang, J. (2006). On in-network synopsis join processing for sensor networks. In: *Proceedings of the 7th international conference on mobile data management* (pp. 32–39).