PAPER

AMJoin: An Advanced Join Algorithm for Multiple Data Streams Using a Bit-Vector Hash Table

Tae-Hyung KWON[†], **Hyeon-Gyu KIM**[†], **Myoung-Ho KIM**[†], ***Nonmembers***, ***and*** **Jin-Hyun SON**[††a)], ***Member***

SUMMARY A multiple stream join is one of the most important but high cost operations in ubiquitous streaming services. In this paper, we propose a newly improved and practical algorithm for joining multiple streams called *AMJoin*, which improves the multiple join performance by guaranteeing the detection of join failures in constant time. To achieve this goal, we first design a new data structure called BiHT (Bit-vector Hash Table) and present the overall behavior of AMJoin in detail. In addition, we show various experimental results and their analyses for clarifying its efficiency and practicability.

key words: *multiple stream join, hash table, bit-vector, hashing*

1. Introduction

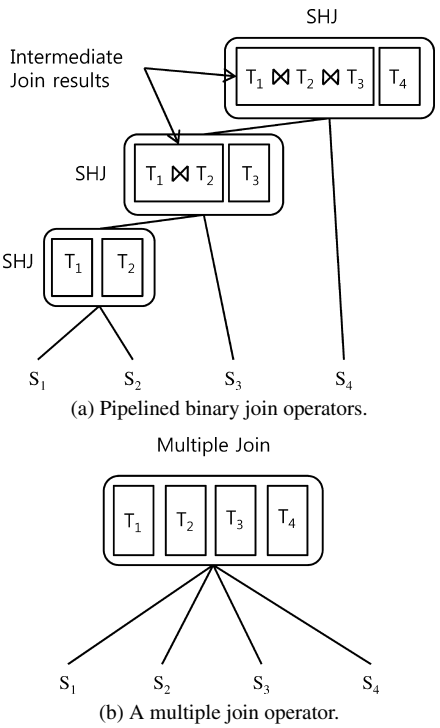
The processing of continuous queries over a set of data streams from multiple data sources is indispensable to ubiquitous streaming applications. For example, in order to monitor which items sell in all target department stores, the following query type that joins multiple data sources using one or more attributes common to each pair of sources should be issued [6].

```
Q1:  SELECT A.ItemName
     FROM Store1 A, Store2 B, Store3 C, Store4 D
     WHERE A.ItemID = B.ItemID AND
           B.ItemID = C.ItemID AND
           C.ItemID = D.ItemID
```

Other useful applications include areas such as object tracking or duplicate detection in sensor networks [4], [5], traffic monitoring in an IP network [2], monitoring streams of stock exchanges [14], and so on.

To support the join of multiple streams continuously, the Symmetric Hash Join (SHJ) has been proposed and improved to allow join results to be produced before completely reading all target data sources [1], [12]. A simple way to implement an n-way join of n streams using SHJ is to combine $n-1$ binary join operators, which are the general method in the conventional DBMSs.

Figure 1 (a) shows a logical query plan to combine 3 binary join operators for query Q1, where T_i is a hash table for input stream S_i and $T_i \bowtie T_j$ is the join results of T_i and T_j. In a pipelined binary join operation like Fig. 1 (a), the intermediate join results are materialized to make a fast binary join such as that between $T_1 \bowtie T_2$ and T_3, and they are incrementally updated whenever new tuples arrive at related input streams. For example, if a new tuple arrives at S_3, it is inserted into T_3 and the only change obtained from joining $T_1 \bowtie T_2$ with S_3 is reflected to the intermediate join result $T_1 \bowtie T_2 \bowtie T_3$. On the other hand, this method has two main limitations: One is that the intermediate join results are quite large and require high maintenance cost, and the other is that the pipelined binary join tree is not adaptable to the changes in the properties of multiple data streams.



(a) Pipelined binary join operators.

(b) A multiple join operator.

Fig. 1 n-ary join operation.

A multiple join method called MJoin [10] is proposed to get over the problems of the pipelined binary join, which joins multiple streams at once as in Fig. 1 (b). Whenever a new tuple arrives at a certain input stream, it is inserted into the corresponding hash table and the new tuple is joined with all the other hash tables at once without delaying the join processing and maintaining any intermediate results. It also does not have any predetermined join order. As you can

Manuscript received November 27, 2008.
Manuscript revised March 7, 2009.

[†]The authors are with the Department of Electrical Engineering and Computer Science, KAIST, 373–1 Kusong-dong Yusong-gu Taejon, 305–701, Republic of Korea.

[††]The author is with the Department of Computer Science and Engineering, Hanyang University, 1271 Sa-3 dong, Ansan, Kyunggi-Do, 426–791, Republic of Korea.

a) E-mail: jhson@hanyang.ac.kr

DOI: 10.1587/transinf.E92.D.1429

imagine, it is important to determine an optimal join order in MJoin. However, finding an optimal n-way join order is known as an NP-hard problem by Toshihide and Tiko [11]. To address this issue, Avnur and Hellerstein [7] have introduced an algorithm for adaptively finding the optimal order of hash table probes. In their algorithm, the join order of each stream is periodically changed to adapt fluctuated selectivity. Assuming selectivity of each stream is independent, they apply a simple heuristic approach where a stream with the lowest selectivity joins first. Babu et al. [9] have discussed the issues when the selectivities of streams are not independent, and have proposed a greedy algorithm to capture correlations among the join selectivities of streams. But, the selectivities may vary among individual tuples (i.e. one tuple may join with many S_1 tuples but few S_2 tuples, while another tuple may behave in opposite way). Moreover, the optimal order can be continuously changed due to the dynamic natures of data streams. At a bottom line, it is difficult for MJoin to find an optimal join order upon a new tuple arrival on any input stream at any time, resulting in unnecessary probing of the other hash tables.

In this paper, we propose a new join algorithm called AMJoin, which extends MJoin to avoid unnecessary probes. For this purpose, we construct a *Bit-vector* Hash Table (*BiHT*) in which each hash entry has a bit-vector consisting of n bits, where i-th bit denotes whether tuples from i-th data stream exist or not. Our AMJoin with the BiHT can achieve the following two important issues: i) It directly determines whether a multiple join can be successfully performed or not upon a new tuple arrival on any input stream. In other words, the unnecessary probes for the other hash tables can be eliminated in case of a multiple join failure. ii) It can efficiently support a selective join query which is executed over partial input streams on demand. The join failure in this kind of a query can also be easily detected using BiHT.

The rest of this paper is organized as follows: Section 2 presents the detail algorithms of AMJoin by comparing with them of previous MJoin. Section 3 shows several experimental results and their analyses. Finally, we summarize our discussion in Sect. 4.

2. Advanced Multiple Join

We first discuss the MJoin algorithm of Stratis et al. [10] and present our AMJoin by improving MJoin to get rid of its unnecessary probes. In Sect. 2.1, we compare the two algorithms in terms of the join processing where all input tuples are fit to main memory. The join processing with memory overflow will be discussed in Sect. 2.2. We discuss the useful query patterns being able to utilize our AMJoin in Sect. 2.3.

2.1 Basic Algorithm

MJoin maintains a hash table per input stream. Let the number of streams be n and a hash table for input stream S_i be T_i ($1 \le i \le n$). The address space of T_i corresponds to the hash values of a join key k. We assume that all hash tables use the same hash function h.

Figure 2 depicts the overall behavior of MJoin which is composed of three consecutive procedures trigged whenever a new tuple r_i arrives in stream S_i.

1. Hashing: Calculate a hash value v_k using hash function $h(r_i.k)$ on join key k for the input tuple r_i, i.e., $v_k = h(r_i.k)$.
2. Moving: Insert r_i to the hash table T_i as an entry for v_k.
3. Probing: Check all T_j ($i \neq j$) to see if each T_j has tuples hashed to bucket v_k. If there exist any T_j ($i \neq j$) with no tuple hashed to the bucket v_k, this join process stops as a failure. Otherwise, r_i joins $T_j(v_k)$ ($i \neq j$) to get the final result.

It is necessary to notice the case of the join failure. In Step 3 of the join process of MJoin, the average $(n-1)/2$ number of probes are required to detect the join failure. This means that there exist unnecessary probes that need to be avoided.

To resolve this problem, we maintain a special hash

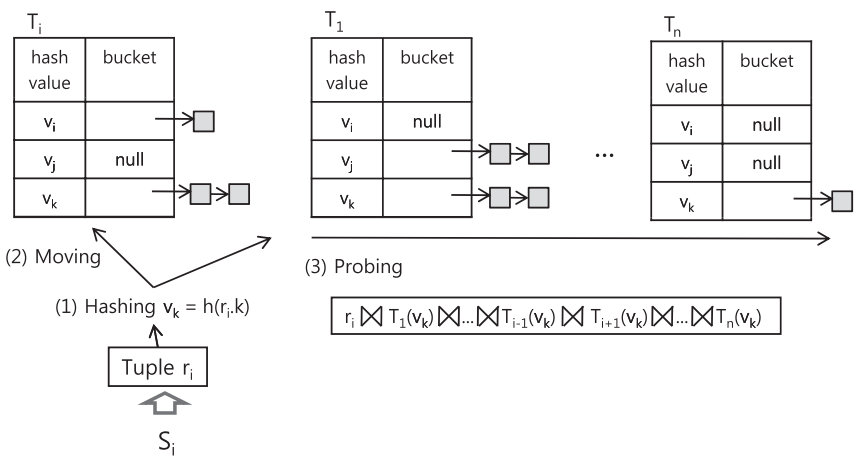


Fig. 2 Overall join process of MJoin.

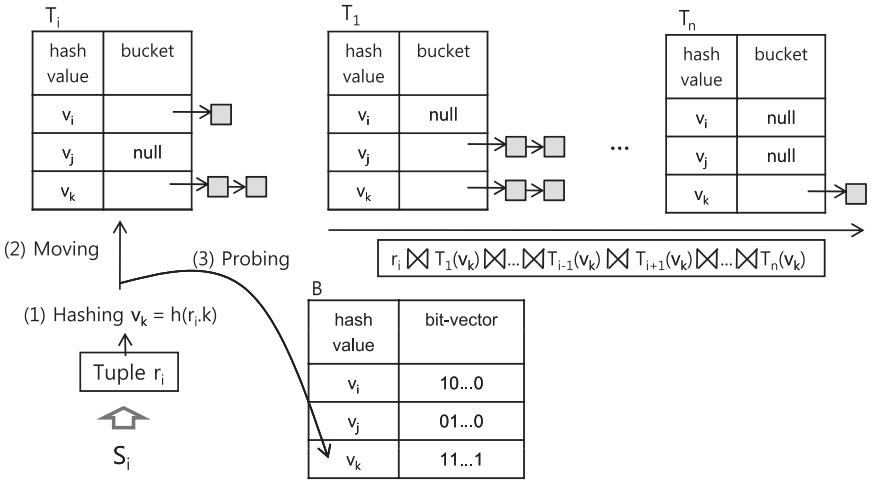


Fig. 3 Overall join process of AMJoin.

table called BiHT (Bit-vector Hash Table) which is denoted as B in Fig. 3. We in this paper assume that the address space of B corresponds to that of hash value v_k which is calculated by hash function h on a join key k, and the same hash function h is used for both B and T_i ($1 \leq i \leq n$). Each hash entry of B has a bit-vector consisting of n bits where i-th bit is set to 1 if tuple r_i from S_i which is hashed to hash value v_k exists. By only checking a proper entry of B, we can easily detect a join failure in real-time. For example, let an input tuple be r_j from stream S_j and its hash value on join key k be v_j, i.e., $v_j = h(r_j.k)$. We can easily find out that r_j cannot be joined with $T_i(v_j)$ if all bits in the bit-vector for hash value v_j of B are not set to 1. Figure 3 depicts the overall behavior of our AMJoin which is composed of three consecutive procedures triggered whenever a new tuple r_i arrives in the stream S_i. The main difference between MJoin and AMJoin is for the probing procedure.

1. Hashing: Calculate a hash value v_k using hash function $h(r_i.k)$ on join key k for the input tuple r_i, i.e., $v_k = h(r_i.k)$.
2. Moving: Insert r_i to the hash table T_i as an entry for v_k.
3. Probing: Set i-th bit of an entry v_k in B to 1, and check whether all bits of the entry v_k are set to 1. If all bits of the entry v_k are not set to be 1, stop the process. Otherwise, r_i joins $T_j(v_k)$ ($i \neq j$) to get the final result.

According to our proposed algorithm described above, the join failure can be easily detected in constant time by simply checking every bit of an entry in B as shown in Step 3. As a result, we can avoid the somewhat large overhead for probing all hash tables to find out whether the join can be processed or not. We will provide various experimental results and their analyses in Sect. 3 to clarify that our AMJoin outperforms previous MJoin.

2.2 Managing Memory Overflow

In this section, we discuss how to deal with memory overflow in both MJoin and AMJoin when input tuples are not fit to main memory. Basically, these two algorithms similarly handle memory overflow except that our AMJoin utilizes BiHT to improve its efficiency.

When an arrival rate of input tuples exceeds the capacity of join processing and then available memory space is exhausted, MJoin temporally prohibits receiving input tuples. And then, it flushes some tuples to make memory available for the join process to be restarted with newly incoming tuples. The flushed tuples remain disks until they are called back when there are no input tuples to be processed. In this way, MJoin tries to process as many as input tuples, while it may not produce the join results in the incoming order. MJoin introduces a method called *coordinate flush* to choose candidate tuples that should go to disks when memory is exhausted. Explaining in detail, it randomly chooses a hash value v and then it flushes all tuples hashed to v from all of the hash tables. After that, newly incoming tuples with a hash value v directly go to disk, because there are no tuples hashed to v in memory. To trace which tuples are flushed to disks, MJoin maintains a flag for each hash entry.

Notice that MJoin randomly chooses a hash value for candidate tuples to be flushed to disks. However, it is necessary to flush as many tuples as possible at once to reserve a sufficient space for restarting the join process. We can achieve this requirement by exploiting BiHT proposed in our AMJoin.

Consider that we want to choose a hash value to flush tuples at the hash tables of Fig. 4. Assume that each hash entry has 3 tuples on the average. If we choose v_{i-1} as the hash value to be flushed, we can reserve a space for 3 tuples because 3 tuples can be flushed to disks. If we, however, choose v_{i+1}, we can reserve a space for 9 tuples. A hash value to flush tuples to disks can be efficiently chosen by probing bit-vectors of BiHT. To meet this purpose of reserving a sufficient space, we expect that the selected hash value should be for a bit-vector whose bit values are almost set to 1.

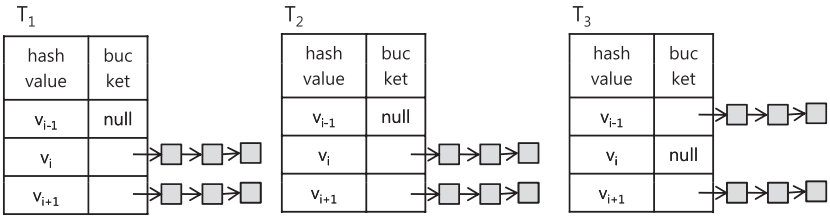


Fig. 4 Coordinate flush in MJoin.

2.3 Useful Query Patterns

As you can imagine, we can identify several useful query patterns which are efficiently processed by AMJoin algorithm. Firstly, consider a query to trace moving objects' trajectory which requires their identifiers, i.e., join key, as query results. For example, consider the following simple query on five sensors as multiple input stream sources.

```
Q2:  SELECT A.id
     FROM Sensor1 A, Sensor2 B, Sensor3 C,
          Sensor4 D, Sensor5 E
     WHERE A.id = B.id and B.id = C.id
     and   C.id = D.id and D.id = E.id
```

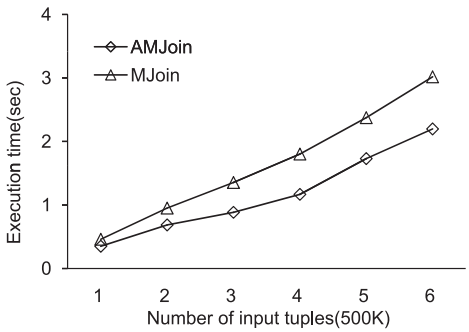
The results of query Q2 can be simply obtained only by maintaining and probing BiHT, not hash tables for input streams.

The next query pattern is to join partial input streams on demand. If there are n input streams and we want a multiple stream join on just m input streams ($m < n$) selectively according to the requirements, the join failure can be easily detected with BiHT of AMJoin. This kind of a query pattern can be considerable for the future streaming applications.

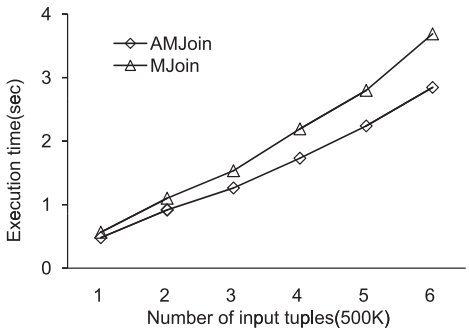
3. Experiments and Analysis

In this section, we present several experimental results to clarify AMJoin's efficiency compared to MJoin in the aspects of join selectivity, application types, and handling memory overflow. First of all, we have implemented a data generator for the experiments based on the given input parameters such as number of input streams, number of tuples per stream, and join selectivity. Each data set is composed of $<key, timestamp, misc>$ which denotes a join key, an arrival timestamp, and a miscellaneous information, respectively. Our experiments have been conducted on Intel Core 2 Duo 2.66 GHz machine running on Window XP with 4 G memory.

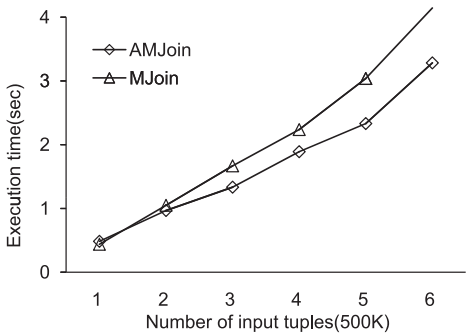
We have observed the execution time required to join input tuples from 5 stream sources for two multiple stream join algorithms discussed in this paper, MJoin and AMJoin. These experiments have been performed by varying the join selectivity from 0.01 to 0.1, which is enough to notice these algorithms' characteristics. Figure 5 shows that our AMJoin gives better performance than MJoin. It is noticeable that AMJoin is more better in case of the lower join selectivity.



(a) Join selectivity 0.01.



(b) Join selectivity 0.05.



(c) Join selectivity 0.1.

Fig. 5 Performance of 5-way join.

This is because AMJoin can efficiently detect join failures, i.e., the cases that not all input streams participate in join, using the BiHT. On the other hand, if the join selectivity is high, it is expected that AMJoin's advantages may be nullified. However, some existing work [13] and [3] shows that the join selectivity in most of the real-world streaming services is known to be relatively low. Actually, [13] and [3] give their experimental results performed with join selectivity less than 0.1. As a result, our AMJoin utilizing BiHT is

said to be practical and efficient.

As we mentioned in Sect. 2.3, there are several query patterns on which AMJoin is always superior to MJoin. To clarify this, we have compared the performance for query Q2 mentioned in Sect. 2.3 under the conditions that the join selectivity is set to 0.1 and there are 5 number of stream sources. Figure 6 shows that our AMJoin provides approximately 3 times better performance than MJoin.

The next experiment is on the flush methods in case of memory overflow mentioned in Sect. 2.2. We have compared the execution time of flush methods in both MJoin and AMJoin as Fig. 7. As we discussed in the above, MJoin conducts the flush by randomly choosing a hash value for candidate tuples flushed to disks. On the other hand, our AMJoin flushed as many tuples as possible at once to reserve a sufficient space for restarting the join process by appropriately selecting a hash value using BiHT proposed in this paper. For this experiment, we configured the memory size to make one third of input tuples to be discarded (i.e., overflow). In addition, the join selectivity is set to 0.1 and we choose one of hash values whose bit-vector has at least 2 bits set to 1. Figure 7 shows that AMJoin copes with memory overflow more efficiently than MJoin.

From now, we need to make various analyses on some performance factors such as memory usage, management overhead, and memory overflow. Our method has somewhat space and management overhead incurred by additional data structure, BiHT. With regard to the space overhead, our method uses 6 bytes, 4 bytes for a hash value and 2 bytes for a bit vector, per hash entry in BiHT. Let us consider a real data streaming application illustrated in [2] as an example. In the AT&T IP backbone, traffic data stream produced by routers are about ten billion fifty-byte records per day (i.e., 115,000 tuples per second) [2]. If we assume to perform a join with the window size of 5 minutes, the total number of independent tuples is 35,000,000 and then the memory size required for hash tables is, after all, about 1.7 Gbytes (i.e., 50 bytes per tuple $\times$ 35,000,000 tuples). For estimating the memory size required for our bit vector, it is important to decide the number of unique tuples within their survival time. Based on the general survival time of network traffic packets [8], if 115,000 tuples generated per second are assumed to survive for 10 seconds, we require about 7 Mbytes for our bit vector table, BiHT (i.e., 115,000 tuples per second $\times$ 10 seconds $\times$ 6 bytes per tuple). As you find out, this amount of additionally required memory is so trivial compared with the size of hash tables for maintaining input tuples.

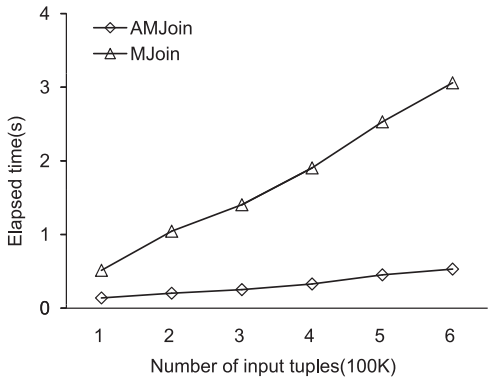


Fig. 6 Performance of Q2. (monitoring moving object trajectory)

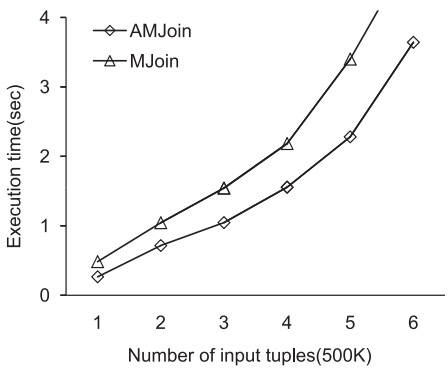


Fig. 7 Performance of 5-way join in memory overflow situation.

With intent of assessing the BiHT management overhead, we have compared the probe costs for MJoin and our AMJoin. The probe cost of MJoin includes the inspection of the hash table, i.e., fetching an entry in a hash table and checking the existence of tuples with a given join key in the entry, while the cost in our AMJoin includes toggling a bit to 1 and checking a bit vector entry within BiHT. Note that we exclude the cost of joining tuples in order to focus on discussing the BiHT management overhead. In our experiments, we find out that it takes 2.88×10^{-4} ms to inspect each hash table in the MJoin probing process. Because it may be necessary to inspect other hash tables for finally deciding whether the join processing should be performed or not, the MJoin probing process may take multiple times of 2.88×10^{-4} ms. On the other hand, our AMJoin probing process takes 1.61×10^{-4} ms which is tightly related with the BiHT management overhead. No additional probes are required for finally deciding the join processing in case of utilizing our BiHT.

Let us consider the memory overflow issue. It is true that our AMJoin is more apt to get into the memory overflow than MJoin because AMJoin additionally keeps its bit vector table, BiHT. As we discussed the memory usage of BiHT above, its space overhead is trivial compared to hash tables. Because of that, the overflow probability occurred by BiHT can be negligible not to affect the overall performance of our AMJoin.

Another performance factor necessary to be considered is the number of input streams. The number of input streams n is more sensitive to the performance of MJoin rather than AMJoin. Its increase may directly make the performance of MJoin degraded due to its average number of probes $(n - 1)/2$ which increases in proportion to n. On the other hand, there is no effect on the performance of our AMJoin except of the slight increase of BiHT memory space. This is caused by the property of the constant time of probing as mentioned in Sect. 2.

4. Conclusions

In this paper, we proposed a new multiple stream join algorithm called AMJoin which extends MJoin to improve its performance in detecting join failures and managing memory overflow, and supports new application requirements such as a query pattern to selectively join partial input streams. To achieve these goals, we first proposed a new data structure called BiHT (Bit-vector Hash Table) in which each hash entry has a bit-vector with n bits corresponding to n input streams. By simply checking a certain bit-vector in BiHT, we can detect the join failure immediately. Based on the BiHT structure, we presented a multiple stream join algorithm called AMJoin. Finally, we showed various experimental results to clarify the efficiency of the algorithm proposed in this paper.

Acknowledgments

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. R0A-2007-000-10046-0 and R01-2007-000-20135-0).

References

[1] A.N. Wilschut and P.M.G. Apers, "Dataflow query execution in a parallel main-memory environment," Distributed and Parallel Databases, vol.1, no.1, pp.103–128, 1993.
[2] 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, San Diego, California, USA, 2003.
[3] H. Yu, E.P. Lim, and J. Zhang, "On in-network synopsis join processing for sensor networks," Proc. 7th International Conference on Mobile Data Management, pp.32–39, Nara, Japan, 2006.
[4] J. Gehrke and S. Madden, "Query processing in sensor networks," IEEE Pervasive Computing, vol.3, no.1, pp.46–55, 2004.
[5] L. Golab and M.T. Ozsu, "Processing sliding window multi-joins in continuous queries over data streams," Proc. 29th International Conference on Very Large Data Bases, vol.29, pp.500–511, Berlin, Germany, 2003.
[6] M.A. Hammad, W.G. Aref, and A.K. Elmagarmid, "Stream window join: Tracking moving objects in sensor-network databases," Proc. 15th International Conference on Scientific and Statistical Database Management, pp.75–84, Cambridge, Massachusetts, USA, 2003.
[7] R. Avnur and J.M. Hellerstein, "Eddies: Continuously adaptive query processing," ACM SIGMOD Record, vol.29, no.2, pp.261–272, 2000.
[8] S.M. Bellovin, "A technique for counting NATted hosts," Proc. 2nd ACM SIGCOMM Workshop on Internet Measurement, pp.267–272, Marseille, France, 2002.
[9] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, "Adaptive ordering of pipelined stream filters," Proc. ACM SIGMOD International Conference on Management of Data, pp.407–418, Paris, France, 2004.
[10] 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, vol.29, pp.285–296, Berlin, Germany, 2003.
[11] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing N-relational joins," ACM Trans. Database Syst. (TODS), vol.9, no.3, pp.482–502, 1984.
[12] T. Urhan and M.J. Franklin, "XJoin: A reactively-scheduled pipelined join operator," IEEE Data Engineering Bulletin, vol.23, no.2, pp.27–33, 2000.
[13] Y. Bai, H. Wang, and C. Zaniolo, "Load shedding in classifying multi-source streaming data: A Bayes risk approach," Proc. Seventh SIAM International Conference on Data Mining, pp.425–430, Minneapolis, Minnesota, USA, 2007.
[14] Y. Zhu, E.A. Rundensteiner, and G.T. Heineman, "Dynamic plan migration for continuous queries over data streams," Proc. ACM SIGMOD International Conference on Management of Data, pp.431–442, Paris, France, 2004.

Tae-Hyung Kwon is a Ph.D. candidate at the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea. He received the B.S. degree in Computer Science from Korean Air Force Academy, Korea in 1995, and M.S. degree in Computer/Information Process from Korean National Defense University, Korea in 2002. His research interests include ubiquitous computing, stream processing, sensor network, and DataBase system.

Hyeon-Gyu Kim is a Ph.D. candidate at the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea. He received the B.S. degree in Computer Science from University of Ulsan, Korea in 1997, and M.S. degree in Computer Science from University of Ulsan, Korea in 2000. His research interests include stream processing, sensor network, and DataBase system.

Myoung-Ho Kim is a full professor of the Department of Computer Science at (KAIST), Taejon, Korea. He received his B.S. and M.S. degrees in Computer Engineering from Seoul National University, Seoul, Korea, in 1982 and 1984, respectively, and his Ph.D. degree in Computer Science from Michigan State University, East Lansing, MI, in 1989. His research interests include database systems, data stream processing, sensor networks, mobile computing, OLAP, XML, information retrieval, workflow and distributed processing. He is a member of the ACM and IEEE computer Society.

Jin-Hyun Son is an associate professor of the Department of Computer Science and Engineering at Hanyang University, Korea, from 2002. He was a postdoctoral researcher in the Division of Computer Science at Korea Advanced Institute of Science and Technology (KAIST), Daejon, Korea. He received his B.S. degree in Computer Science from Sogang University, Seoul, Korea, in 1996, and his M.S. and Ph.D. degrees in Computer Science from (KAIST) in 1998 and 2001, respectively. His research interests include business process management, distributed processing, database systems and ubiquitous embedded software.