

## Distributed stream join under workload variance

Junhua Fang<sup>1,2</sup> · Rong Zhang<sup>1,2</sup> · Xiaotong Wang<sup>1,2</sup> ·  
Aoying Zhou<sup>1,3</sup>

Received: 4 October 2016 / Revised: 3 December 2016 /  
Accepted: 1 January 2017 / Published online: 23 January 2017  
© Springer Science+Business Media New York 2017

**Abstract** Flexible and self-adaptive stream join processing plays an important role in a parallel shared-nothing environments. Join-Matrix model is a high-performance model which is resilient to data skew and supports arbitrary join predicates for taking random tuple distribution as its routing policy. To maximize system throughputs and minimize network communication cost, a scalable partitioning scheme on matrix is critical. In this paper, we present a novel flexible and adaptive scheme partitioning model for stream join operator, which ensures high throughput but with economical resource usages by allocating resources on demand. Specifically, a lightweight scheme generator, which requires the sample of each stream volume and processing resource quota of each physical machine, generates a join scheme; then a migration plan generator decides how to migrate data among machines under the consideration of minimizing migration cost while ensuring correctness. We do extensive experiments on different kinds of join workloads and the evaluation shows high competence comparing with baseline systems on benchmark data and real data.

**Keywords** Distributed stream join · Workload balance · Matrix model · Theta-join

---

✉ Rong Zhang  
rzhang@sei.ecnu.edu.cn

Junhua Fang  
fangunhua26@gmail.com

<sup>1</sup> Shanghai Key Laboratory of Trustworthy Computing, School of Computer Science and Software Engineering, East China Normal University, Shanghai 200062, China

<sup>2</sup> International Joint Lab of Trustworthy Software, School of Computer Science and Software Engineering, East China Normal University, Shanghai 200062, China

<sup>3</sup> School of Data Science and Engineering, East China Normal University, Shanghai 200062, China

## 1 Introduction

With development of communication technology and hardware equipment, especially the widespread use of small wireless sensor, data is showing the scale of explosive growth in many fields, for example in spatial data management [20–22, 37, 38]. It exposes challenges to such applications to provide efficient online stream processing in real time. Efficient stream join processing with arbitrary predicates is essential and critical to data stream systems. Data skew is a common phenomenon in those scenarios and leads to lingering processing in parallel shared-nothing environment [10]. In this context, load balancing [16, 19, 28, 33, 34] is crucial for improving throughputs by vanishing those lingering tasks which are overloaded. There have been work to solve the imbalance problem among tasks for different operations, such as Summarization [5], Aggregation [4, 23] and Join [14, 30, 34], which may group data by keys for processing. One of the most challenging tasks above is to support  $\theta$ -joins [7, 18, 24] in a flexible, efficient and scalable manner under workload skewness.

There have been great interest in designing stream join algorithms. On one hand, existing centralized algorithms [6, 15, 29, 32] are mainly tailored for a single server, hence they cannot scale out and deal with massive data sets; on the other hand, existing distributed and parallel join algorithms are mostly suitable for equi-join processing and present poor performance when handling  $\theta$ -join operators.

Processing  $\theta$ -joins on streams with skew data distribution, there are two kinds of popular processing models, namely join-biclique [18] and join-matrix [7, 24]. Lin et al. proposed a join-biclique model [18] which organizes all the processing units as a complete bipartite graph where each side corresponds to one stream. Given  $m + n$  units (tasks),  $m$  units from one side of the bipartite graph are used for managing/storing tuples in one stream while flowing tuples from the other stream among those  $m$  units;  $n$  units are for the other one. Join-matrix model supports distributed join processing with arbitrary join predicates perfectly. It was studied a decade ago and has been revisited in both MapReduce-like system [24] and stream applications [7]. Apparently, it models a join operation between two streams as a matrix, where each side is on behalf of one stream. Furthermore, the change of input stream volume will lead to the adjustment of matrix scheme accordingly to handle the new workload distribution. Based on this model, Okcan [24] et al. introduces two partitioning schemes in a MapReduce job, namely 1-Bucket and M-Bucket. It performs well only when input or output dominates processing cost and it requires to get input statistics before optimization execution. *Dynamic*, a join operator designed in [7], adopts a grid-layout partitioning scheme on the matrix. Although it is resilient to data skew as taking a random distribution as routing policy for input tuples and it can perfectly handle any join predicates for it ensures each tuple of one stream to meet any tuple in the other stream. However, it suffers from inflexibility and huge amount of tuple duplication while scaling out or down. For example, *Dynamic* assumes the number of tasks in a matrix must be a power of two and scales out by splitting the states of every task to four tasks if a task storing a number of tuples exceeding the specified storage capacity or processing capacity, and vice versa.

The matrix model seems to be the most suitable one for  $\theta$ -join with correctness guarantee and balance load distribution, but it still has the following inherent disadvantages:

- 1) The number of tasks is strictly decided by the number of cells in the matrix, which is calculated by multiplying the number of rows and columns of the matrix;
- 2) In the case of stream change, that is stream volume increasing or shrinking, adding or removing processing tasks must be consistent with matrix cells.

Since the allocation of tasks is decided by the number of cells of the matrix, it greatly limits the flexibility of processing with the dynamics of stream, and may cause resource waste by generating more tasks than needed.

According to the discussion above, join-matrix model exposes two challenges: **1)** how to generate an appropriate matrix scheme to achieve maximum utilization of resources; **2)** when scheme is changed, how to repartition states among tasks to minimize migration cost.

Our previous work in [9] has shown a flexible and adaptive model for distributed and parallel stream join processing. In this paper, we continue optimize this model and make it inherit the characteristics of traditional matrix model but allow irregular shaping of the matrix to allocate resource in demand. Then it can promise the efficient resource utilization. In summary, we have made the following contributions in this paper:

- 1) We adopt the matrix generation strategy based on the idea of square cells having less resource consumption. In this paper, we give more detailed explanation to scheme generation and tuple migration.
- 2) We show the routing strategy for input tuples in the proposed model. To find the migration plan that leads to minimal data transmission overhead, we present a lightweight computation model and explain it through particular examples.
- 3) We introduce a varietal matrix scheme which aims to build an irregular matrix scheme with minimal task assignment. Besides, we give more detailed theoretical analysis for our proposed model to prove its usability and correctness.
- 4) We have implemented the proposed method based on Apache Storm. In this paper, we do much more extensive experiments using both benchmark datasets and real-work workloads to confirm the advantages of our method.

The remainder of this paper is organized as follows. Section 2 introduces the background and problem formulation of our work. Section 3 presents our scheme generation algorithms to support load change. Section 4 proposes the migration plan generation algorithm. Section 5 will discuss the further optimization for matrix model. Section 6 presents empirical evaluations of our proposal. Section 7 reviews a wide spectrum of related studies on stream join and workload balancing in distributed systems. Section 8 finally concludes the paper and addresses future research directions.

## 2 Background and problem formulation

In this section, we give a detailed introduction to the related matrix model, and then we present our optimization goal on the matrix model.

### 2.1 Matrix model

In order to make it easy for explanation, all notations used in the rest of this paper are summarized in Table 1. A partitioning scheme on the matrix model splits  $R \bowtie S$  into a number of smaller parallel join processing units which are the cells in the matrix decided by rows and columns. Each cell holds partial subset of data from each stream, which is represented as a range  $[b, e]$  to denote the begin  $b$  and end  $e$  points along the stream window. In Figure 1a, a join operation between two data streams  $R$  and  $S$  can be modeled as a matrix, each side of which corresponds to one stream. The calculation area can be represented by a rectangular with width  $|R|$  and length  $|S|$ . A partitioning scheme splits the area into cells

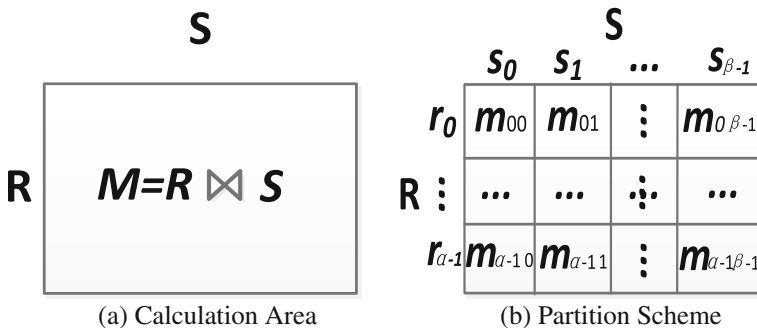
**Table 1** Table of Notations

Notations	Description
$M/M_o/M_n$	matrix / old matrix / new matrix
$R, S$	R, S stream
$\alpha, \beta$	number of rows and columns of $M$
$i/j$	$i^{th}$ row, $j^{th}$ column in old scheme
$k/l$	$k^{th}$ row, $l^{th}$ column in new scheme
$m_{ij}$	element of $i$ -th row and $j$ -th column
$r_i/s_j$	stream in $i$ -th row or $j$ -th column
$mp$	the migration plan
$h_{ij}^R/h_{ij}^S$	the sub-range of stream $R/S$ that has been stored in $m_{ij}$
$s_{kl}^R/s_{kl}^S$	the sub-range of stream $R/S$ that should be stored in $m_{kl}$
$V(V_h)$	memory size of each task(half size $V_h = \frac{V}{2}$ )
$npi$	the mapping of tasks between old and now scheme
$ o $	the volume of set $o$
$NP/MP$	the set of $npi/mp$
$\lambda_{kl}^{ij}$	data overlapping between $m_{ij}$ and $m_{kl}$

$m_{ij} = (r_i, s_j)$  ( $0 \leq i \leq \alpha - 1, 0 \leq j \leq \beta - 1$ ) of equal size representing stream volume as shown in Figure 1b.

Specifically, any process scheme  $M$  has the following characteristics when we use it to perform the operation of  $R \bowtie S$ .

- 1)  $\forall j, j' \in [0, \beta - 1], \forall i \in [0, \alpha - 1]$ , then  $h_{ij}^R = h_{ij'}^R$  and  $\forall i, i' \in [0, \alpha - 1], \forall j \in [0, \beta - 1]$ , then  $h_{ij}^S = h_{i'j}^S$ ;
- 2)  $\forall j \in [0, \beta - 1], \forall i, i' \in [0, \alpha - 1], i \neq i'$ , then  $h_{ij}^R \cap h_{i'j}^R = \emptyset$  and  $\forall i \in [0, \alpha - 1], \forall j, j' \in [0, \beta - 1], j \neq j'$ , then  $h_{ij}^S \cap h_{ij'}^S = \emptyset$ ;
- 3)  $\forall j \in [0, \beta - 1], \bigcup_{i \in [0, \alpha - 1]} h_{ij}^R = R$  and  $\forall i \in [0, \alpha - 1], \bigcup_{j \in [0, \beta - 1]} h_{ij}^S = S$ .



**Figure 1** Example of Partition Scheme

For those characteristics in matrix model, Points **2)** and **3)** enable this model to support arbitrary join calculation by that tuples in one stream can meet all tuples in the other stream. Furthermore, Points **2)** and **3)** also ensure the correctness of  $R \bowtie S$ . Specifically, point **2)** guarantees there will exist none reduplicated results and point **3)** ensures that there will not have missing results. In this context, points **2)** and **3)** act as our principles in designing scheme generation algorithm.

## 2.2 Optimization goal

Our optimization goal is to figure out the proper values for  $\alpha$  and  $\beta$  to achieve the optimal resource usages. Supposing the maximum memory size for each task is  $V$ , we formulate our goal as an optimization problem defined as Eq. 1:

$$\begin{aligned} \min \quad & \alpha \cdot \beta, \\ \text{s.t.} \quad & |R| \cdot \beta + |S| \cdot \alpha \leq \alpha \cdot \beta \cdot V \\ & \alpha \geq 1, \beta \geq 1. \end{aligned} \quad (1)$$

In Eq. 1, we can find the minimal number of task for a regular matrix scheme. In other word, our purpose is to find the proper values for  $\alpha$  and  $\beta$  in Eq. 1. However it is still too strict to generate tasks according to the regular matrix scheme. Then our optimization goal is changed to find an irregular matrix while guaranteeing correctness as Section 3.

## 3 Scheme generation

We first introduce two theorems to explain the guild for generating the optimal matrix scheme. And then, we describe the adaptive process of generating matrix scheme based on these two theorems according to the real workload.

### 3.1 Model design

Since those subsets may be replicated along rows or columns, the values of  $\alpha$  and  $\beta$  decide the memory consumption which is proportional to the subarea's semi-perimeter valued as  $|r_i| + |s_j|$  as in [7]. Given the area or the perimeter, we introduce the following two well known theories:

**Theorem 1** *Given the area with a constant value, the square has the smallest perimeter among all the rectangles.*

**Theorem 2** *Given the perimeter with a constant value, the square has the biggest area among all the rectangles.*

Based on these two theories, we have the following corollary on partitioning scheme:

**Corollary 1** *If there exist  $\alpha$  and  $\beta$  which can make  $\frac{|R|}{\alpha} = \frac{|S|}{\beta} = V_h$ , the consumption of processing resource for  $R \bowtie S$  is minimal.*

*Proof* Supposing CPU resource is a constant value in each task, in order to ensure any tuple meets the others, the computation complexity for stream join is  $|R| \cdot |S|$ . However the

memory usage will be minimized if  $\frac{|R|}{\alpha} = \frac{|S|}{\beta} = V_h$  according to Theorem. 1. Supposing the memory resource of each task is constant, the number of tasks used for the calculation (total area) is smallest when  $\frac{|R|}{\alpha} = \frac{|S|}{\beta} = V_h$  according to Theorem. 2. The network communication cost is decided by memory usages, that is to say the volume of tuples stored in memory equals to the transmission volume. In other words, optimization over memory consumption always lows the bandwidth consumption at the same time. Based on the discussion above, we can draw a conclusion that Corollary. 1 is established.  $\square$

According to Corollary. 1, if the volumes of two streams  $|R|$  and  $|S|$  can both be divisible by  $V_h$ , receiving tuples with quantity of  $V_h$  from both streams is a prefect solution to generate matrix scheme  $M$  with minimal resource usages. However, stream volumes may not always be divisible by  $V_h$ . Given that the number of row (column) in matrix  $M$  must be an integer, we get the row number  $\alpha = \lceil \frac{|R|}{V_h} \rceil$ , and the column number  $\beta = \lceil \frac{|S|}{V_h} \rceil$ . Then the number of tasks  $N$  used in matrix  $M$  can be expressed as

$$N = \lceil \frac{|R|}{V_h} \rceil \cdot \lceil \frac{|S|}{V_h} \rceil \tag{2}$$

In those  $N$  cells, we primarily load the first  $\alpha - 1$  rows or  $\beta - 1$  columns of cells. When the stream volume can not be evenly divided by  $V_h$ , it generates fragment data for the tasks (called fragment tasks) in the last row or the last column in matrix  $M$ .

For example, given task memory  $V = 10GB$ ,  $R$  stream volume  $|R| = 6GB$  and  $S$  stream volume  $|S| = 6GB$ , its calculation area is shown in Figure 2a. Processing  $R \bowtie S$  will take up 4 tasks for its matrix  $M$  with two rows and two columns shown as Figure 2b. In  $M$ ,  $m_{00} = (5GB, 5GB)$ ,  $m_{01} = (5GB, 1GB)$ ,  $m_{10} = (1GB, 5GB)$  and  $m_{11} = (1GB, 1GB)$ . Then  $m_{01}$ ,  $m_{10}$ ,  $m_{11}$  are fragment tasks in that the sum memory consumption of  $|r_{ij}|$  and  $|s_{ij}|$  in these tasks is smaller than  $V$ .

### 3.2 Generation scheme

To find an optimal processing scheme, we differentiate the two streams as a primary stream  $P$  and a secondary stream  $D$ . Supposing we split  $P$  into  $P_\gamma$  subsets assigned to each task, we first ensure the memory usage for those subsets from  $P$ . And the remaining memory  $V - \frac{P}{P_\gamma}$  in each task is used for the subset of data from  $D$ . The number of tasks  $D_\gamma$  required for  $D$  can be calculated as:

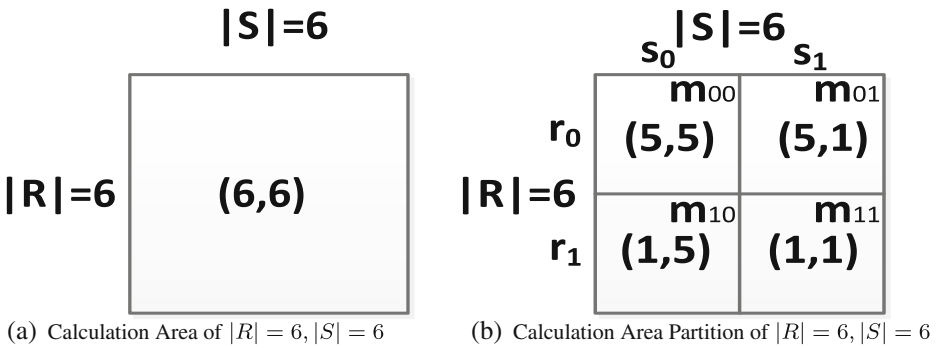
$$D_\gamma = \lceil \frac{D}{V - \frac{P}{P_\gamma}} \rceil \tag{3}$$

We use  $N_c$  to represent the number of tasks and it can be calculated as Eq. 4:

$$N_c = P_\gamma \cdot D_\gamma = P_\gamma \cdot \lceil \frac{D}{V - \frac{P}{P_\gamma}} \rceil \tag{4}$$

As declared in Corollary. 1, the number of tasks is minimized when  $\frac{|R|}{\alpha} = \frac{|S|}{\beta} = V_h$ , but we can not promise to find such  $\alpha$  and  $\beta$ . According to Theorem 4, we can select  $P_\gamma$  from  $\{\lceil \frac{|R|}{V_h} \rceil, \lfloor \frac{|R|}{V_h} \rfloor, \lceil \frac{|S|}{V_h} \rceil, \lfloor \frac{|S|}{V_h} \rfloor\}$ , and there exit one  $P_\gamma$  to generate the minimal number of tasks  $N_c$  calculated as in Eq. 4.

**Theorem 3** *Given stream volumes  $|R|$ ,  $|S|$  and memory size  $V$  of task, using matrix model for  $R \bowtie S$ , the number of tasks generated by  $P_\gamma * D_\gamma$  as Eq. 4 by selecting  $P_\gamma$  from  $\{\lceil \frac{|R|}{V_h} \rceil, \lfloor \frac{|R|}{V_h} \rfloor, \lceil \frac{|S|}{V_h} \rceil, \lfloor \frac{|S|}{V_h} \rfloor\}$  is the minimal.*



**Figure 2** A Toy Example of Calculation Partition

*Proof* We assume that there exists a matrix  $M'$  with row number  $\alpha'$  and column number  $\beta'$  which can be used for  $R \times S$  and the number of tasks  $N'$  is smaller than  $N_c$ . In other words, there is a number  $P_\gamma' : P_\gamma' \notin \{\lceil \frac{|R|}{V_h} \rceil, \lfloor \frac{|R|}{V_h} \rfloor, \lceil \frac{|S|}{V_h} \rceil, \lfloor \frac{|S|}{V_h} \rfloor\}$  and  $N' < N_c$ . According to Corollary. 1, square has the largest area. Then  $\frac{|R|}{\alpha'}$  is closer to  $V_h$  than  $\frac{|R|}{\alpha}$ , and  $\frac{|S|}{\beta'}$  is also closer to  $V_h$  than  $\frac{|S|}{\beta}$ . However, it is impossible for  $\frac{|R|}{\alpha'}$  and  $\frac{|S|}{\beta'}$  to get closer to  $V_h$  simultaneously. Moreover,  $P_\gamma$  occupies all possible values that make  $\frac{P}{P_\gamma}$  nearest to  $V_h$ . Hence, there is not any smaller  $N'$  existing. □

---

**Algorithm 1** Scheme generation

---

**input:** Stream  $R$ , Stream  $S$ , Memory size  $V$   
**output:** Row number  $\alpha$ , Column number  $\beta$

- 1: **foreach**  $P_\gamma \in \{\lceil \frac{|R|}{V_h} \rceil, \lfloor \frac{|R|}{V_h} \rfloor, \lceil \frac{|S|}{V_h} \rceil, \lfloor \frac{|S|}{V_h} \rfloor\}$  **do**
- 2:      $D_\gamma^i \leftarrow \lceil \frac{D}{V - \frac{P}{P_\gamma}} \rceil$
- 3:     **if**  $P_\gamma \cdot D_\gamma > P_\gamma^i \cdot D_\gamma^i$  **then**
- 4:          $P_\gamma \leftarrow P_\gamma^i$  and  $D_\gamma \leftarrow D_\gamma^i$
- 5:     **if**  $P_\gamma \in \{\lceil \frac{|R|}{V_h} \rceil, \lfloor \frac{|R|}{V_h} \rfloor\}$  **then**
- 6:          $\alpha \leftarrow P_\gamma, \beta \leftarrow D_\gamma$
- 7:     **else**
- 8:          $\alpha \leftarrow D_\gamma, \beta \leftarrow P_\gamma$
- 9:     **return**  $\alpha, \beta$

---

The algorithm of finding an optimal partition scheme is described in Algorithm 1. Firstly, the minimal number of tasks is determined in line 1 according to Eq. 4; then in line 5 ~ 8, the number of rows  $\alpha$  and columns  $\beta$  can be calculated according to values of  $P$  and  $P_\gamma$ . If  $P_\gamma \in \{\lceil \frac{|R|}{V_h} \rceil, \lfloor \frac{|R|}{V_h} \rfloor\}$ ,  $R$  is the primary stream, or else  $S$  is the primary one. After we select the  $P$  stream, each task will first be fed up with data from  $P$  with memory  $\frac{|P|}{P_\gamma}$  and leave the remaining memory  $V - \frac{|P|}{P_\gamma}$  for  $D$  stream.

**Theorem 4** Algorithm 1 will consume the minimal number of tasks and ensure the correctness of operation when using matrix model for  $R \bowtie S$  with the memory size of each task  $V$ .

*Proof* Assuming that there exists another matrix  $M'$  with the number of row  $\alpha'$  and column  $\beta'$ . It could be used for  $R \bowtie S$  and the number of tasks  $N'$  used in  $M'$  is smaller than  $N_c$ . To find the smaller  $N_c$ , Algorithm 1 tries all possible values that make  $\frac{|P|}{P_\gamma}$  nearest to  $V_h$  in line (1 ~ 4). In other words, it is impossible for  $\frac{|R|}{\alpha'}$  and  $\frac{|S|}{\beta'}$  to get more closer to  $V_h$  than  $\frac{|R|}{\alpha}$  and  $\frac{|S|}{\beta}$ . According to Theorem. 1 that squared cells consume the minimal resources. Then, the assumption of existing regular matrix  $M'$  does not hold. For ensuring the correctness of join, it is obvious that Algorithm 1 can ensure join correctness when it generates a regular matrix scheme according to the characteristics of matrix model in Section 2.  $\square$

## 4 Implementation

After we generate the new scheme, we should calculate a migration plan from the old scheme to the new scheme. In this section, we will first introduce how to route the input data stream which is designed to promise the correctness of join results, and then we describe how to map tasks between the new and old scheme with the target of lowering the migration cost.

### 4.1 Tuple routing

In this section, we introduce how to route tuples in the basic matrix model with a random tuple distribution manner. As described in Section 2.1, matrix model randomly routes tuples to cells of each stream corresponding to one side of the matrix. Hence, it can handle data skewness perfectly. The whole procedure of basic tuple routing is described in Algorithm 2.

We use  $\Gamma$  ( $\Gamma \in \{row, column\}$ ) to represent which side of matrix the input tuple correspond to and use  $\epsilon$  ( $\epsilon \in [0, (\alpha - 1)]$  or  $\epsilon \in [0 \sim (\beta - 1)]$ ) representing to which line in the side of  $\Gamma$  the tuple should be sent. Then the return pair  $(\Gamma, \epsilon)$  of Algorithm 2 means the input tuple should be sent to the  $\epsilon^{th}$  line in  $\Gamma$  side of matrix. In Algorithm 2, line (2 ~ 3) and line (5 ~ 6) identify which side of the matrix the input tuple belongs to, then, line 4 and line 7 accordingly generate a random position. Finally, for the routing of matrix model, the input tuple will be sent to all the processing tasks which located in the line  $\epsilon$  along the side of  $\Gamma$ .

---

#### Algorithm 2 Basic tuple routing

---

**input:** R stream , S stream  
**output:** Matrix side: $\Gamma$ , Line: $\epsilon$

- 1: **foreach** tuple  $\tau$  **do**
- 2:     **if**  $\tau \in R$  **then**
- 3:          $\Gamma \leftarrow row$
- 4:          $Random[0 \sim (\alpha - 1)] \rightarrow \epsilon$
- 5:     **else if**  $\tau \in S$  **then**
- 6:          $\Gamma \leftarrow column$
- 7:          $Random[0 \sim (\beta - 1)] \rightarrow \epsilon$
- 8:     **return**  $\Gamma, \epsilon$

---



## 4.2 Task-load mapping generation

Supposing  $m_{ij}$  and  $m_{kl}$  corresponds to two cells in the matrix which are  $M_o$  in old schema and  $M_n$  in new schema respectively, and each cell corresponds to one join processing task. In order to lower the data migration among cells during schema change, it is crucial to find the optimal task for each cell in  $M_n$ . Less migration cost means there are more data overlap for the cells between old and new scheme. We then define a overlapping coefficient  $\lambda_{kl}^{ij}$  for each pair of tasks corresponding to  $m_{ij}$  and  $m_{kl}$ , which are in  $M_o$  and  $M_n$  respectively.  $\lambda_{kl}^{ij}$  is a measurement for the cell data overlapping between  $m_{ij}$  and  $m_{kl}$  calculated as Eq. 5.

$$\lambda_{kl}^{ij} = (h_{ij}^R \cap s_{kl}^R) \cdot |R| + (h_{ij}^S \cap s_{kl}^S) \cdot |S| \quad (5)$$

A new indicant  $npi = \langle m_{ij}, m_{kl}, \lambda_{kl}^{ij} \rangle$  (task mapping item) is defined to represent the effort for migration ( $|s_{kl}^R| + |s_{kl}^S| - \lambda_{kl}^{ij}$ ) when using the task in charge of  $m_{ij}$  for the data in  $m_{kl}$ . The whole procedure of task pairing is described in Algorithm 3 and can be divided into two parts:

- 1) *part I* enumerates all the possible  $npi$ s shown in line (2 ~ 5) in Algorithm 3;
- 2) *part II* generates task pairing relationship with the purpose of minimizing migration by selecting  $npi$  with the biggest  $\lambda_{kl}^{ij}$  into  $NP$  set. This  $NP$  set will generate the task-load mapping with the least migration cost according to Theorem. 5.

---

### Algorithm 3 Generation task-load mapping

---

**input:** Old scheme  $M_o$ , New scheme  $M_n$

**output:** Task mapping  $NP$

- 1: Initialize  $NP = Null$
  - 2: **foreach**  $m_{ij}$  in Old scheme  $M_o$  **do**
  - 3:     **foreach**  $m_{kl}$  in New scheme  $M_n$  **do**
  - 4:         Calculate  $\lambda_{kl}^{ij}$  according to Eq. 5
  - 5:         Add the item  $\langle m_{ij}, m_{kl}, \lambda_{kl}^{ij} \rangle$  to a temporary set  $NPI$
  - 6: **foreach** task mapping item with max  $\lambda_{kl}^{ij}$  in  $NPI$  **do**
  - 7:     **if**  $m_{ij}$  or  $m_{kl}$  in  $npi$  not exist in  $NP$  **then**
  - 8:          $\langle m_{ij}, m_{kl} \rangle \rightarrow NP$
  - 9:     Delete  $\langle m_{ij}, m_{kl} \rangle$  from  $NPI$
  - 10: **return**  $NP$
- 

**Theorem 5** Among task pairings between the old and new scheme,  $NP$  set produced by Algorithm 3 leads to the minimal migration cost.

*Proof* For *part I* in Algorithm 3, it enumerates all the possible  $npi$ s with the size of  $|M_o| \cdot |M_n|$ . In other words,  $\alpha_o \cdot \beta_o \cdot \alpha_n \cdot \beta_n$  items are generated where  $\alpha_o$  and  $\beta_o$  are the number of row and column in old scheme  $M_o$ , and  $\alpha_n$  and  $\beta_n$  are the number of row and column in new scheme  $M_n$ . Obviously,  $|NP|$  is a smaller one and each  $m_{ij}$  or  $m_{kl}$  appears in  $NP$  only once at most (guaranteed by line 7). Then we can conclude that the current maximal  $\lambda_{kl}^{ij}$  is independent of others. That is to say *part II* described as line (6 ~ 9) in Algorithm 3

produces the maximal cumulative sum of  $\lambda_{kl}^{ij}$ . It means there is the maximal volume of non-migrating data in  $NP$ , and that the task mapping  $NP$  leads to the minimal migration cost. Based on the discussion above, we can draw that Theorem. 5 is established.  $\square$

### 4.3 Migration plan generation

As described above, a migration plan defines how to migrate data among tasks when scheme changes. In order to make it easy for explanation, we describe data moving among tasks with Stream  $R$ , and it will be the same for Stream  $S$ . We use  $n_{kl}^R$  to denote the range of data should be moved into area  $m_{kl}$  from stream  $R$ , calculated as Eq. 6:

$$n_{kl}^R = s_{kl}^R - (s_{kl}^R \cap h_{kl}^R) \tag{6}$$

Migration plan  $mp = \langle m_{ij}, m_{kl}, N_{ij}^R \rangle$  tells the data moving between two area  $m_{ij}$  in old scheme  $M_o$  and  $m_{kl}$  in new scheme  $M_n$ , with  $N_{ij}^R$  representing the data moving from area  $m_{ij}$  to area  $m_{kl}$  for  $R$ . We define two kinds of actions for moving: duplicating and migrating. Duplicating happens among tasks along the same row/column, otherwise, it is data migrating. Supposing for each cell  $m_{kl}$  in new scheme  $M_n$ ,  $h_{kl}^R$  and  $s_{kl}^R$  are the tuples in it for the last schema and should be kept in current schema. Then cell  $m_{kl}$  deletes the migrated data in set  $h_{kl}^R - s_{kl}^R$  for stream  $R$ , which is represented as  $mp = \langle \ominus, m_{kl}, h_{kl}^R - s_{kl}^R \rangle$ . All the calculations are the same for stream  $S$ .

Migration plan generation is described in Algorithm 4 and is divided into two steps as follows:

**Step-1: Splitting stream data for matrix cells.** According to matrix characteristics described in Section 2.1, it is easy for us to get the whole data set of stream  $R$  or  $S$  by combining the data from the first row or the first column in  $M_o$ . According to the new scheme  $M_n$ , we can divide the streams evenly to fill each cell as in line (1 ~ 8);

**Step-2: Deleting migrated tuples.** It deletes migrated data under the new scheme  $M_n$  in line (9 ~ 11).

Let’s take Figure 3 as an example. A partitioning scheme changes from  $2 \times 1$  to  $2 \times 2$ . In old scheme  $M_o$ , each area manages a half of data volume from  $R$  and the total volume of data from  $S$  shown in Figure 3a:  $h_{00}^R = [0, \frac{1}{2}]$ ,  $h_{10}^R = [\frac{1}{2}, 1]$  and  $h_{00}^S = [0, 1]$ ,  $h_{10}^S = [0, 1]$ . When the workload of streams increases, system may scale out by adding one more column with two tasks forming a  $2 \times 2$  scheme as shown in Figure 3b. In this case, data partitions of  $R$  are unchanged where tasks in the first row still manage a half of data volume ( $s_{0j}^R = [0, \frac{1}{2}]$ ,  $j \in \{0, 1\}$ ) and tasks in the second row manage the other half ( $s_{1j}^R = [\frac{1}{2}, 1]$ ,  $j \in \{0, 1\}$ ). Stream  $S$  should be split into two partitions for two columns, each of which manages  $\frac{1}{2}$  range of data, that is  $s_{i0}^S = [0, \frac{1}{2}]$ ,  $s_{i1}^S = [\frac{1}{2}, 1]$ , with  $i \in \{0, 1\}$ .

According to the discussion in Section 4.2,  $NP$  is  $\{ \langle m_{00}^o, m_{00}^n \rangle, \langle m_{10}^o, m_{10}^n \rangle \}$  as shown in Figure 3b. In Figure 3b, we label the relevant task pairs between  $M_o$  and  $M_n$  by assigning tasks the same numbers. The tasks tagged with red *new* in  $m_{01}$  and  $m_{11}$  are new additive tasks.  $m_{01}$  needs data  $n_{01}^R = [0, \frac{1}{2}]$  and  $n_{01}^S = [\frac{1}{2}, 1]$ ;  $m_{11}$  needs data  $n_{11}^R = [\frac{1}{2}, 1]$  and  $n_{11}^S = [0, \frac{1}{2}]$ . According to Algorithm 4,  $s_{01}^R$  and  $s_{11}^R$  are generated by duplicating  $R$  data from  $m_{00}$  and  $m_{10}$ , respectively.  $m_{01}$  and  $m_{11}$  generate  $S$  by duplicating  $[\frac{1}{2}, 1]$  from  $m_{00}$ . Since  $S$  has been reallocated according to discussion above, then the range of data in  $[\frac{1}{2}, 1]$  from  $S$  are deleting from  $m_{00}$  and  $m_{01}$ .

**Algorithm 4** Migration plan generation

```

input: Old scheme  $M_o$ , New scheme  $M_n$ , Task mapping  $NP$ 
output: Migration plan  $MP$ 

1: foreach row  $i$  with column 0 in old scheme  $M_o$  do
2:   foreach  $m_{kl}$  in new scheme  $M_n$  do
3:     if  $h_{i0}^R \cap n_{kl}^R \neq Null$  then
4:        $\langle m_{i0}, m_{kl}, h_{i0}^R \cap n_{kl}^R \rangle \rightarrow MP$ 
5:   foreach column  $j$  with row 0 in old scheme  $M_o$  do
6:     foreach  $m_{kl}$  in new scheme  $M_n$  do
7:       if  $h_{0j}^S \cap n_{kl}^S \neq Null$  then
8:          $\langle m_{0j}, m_{kl}, h_{0j}^S \cap n_{kl}^S \rangle \rightarrow MP$ 
9:   foreach task  $m_{kl}$  in new scheme  $M_n$  do
10:     $\langle \odot, m_{kl}, h_{kl}^R - s_{kl}^R \rangle \rightarrow MP$ 
11:     $\langle \odot, m_{kl}, h_{kl}^S - s_{kl}^S \rangle \rightarrow MP$ 
12: return  $MP$ 
    
```

**5 Discussion**

In this section, we will discuss the further optimization for matrix model to pursue a more cost-effective model. And then, we describe the tuple routing approach in the variant model which may have better resource usage.

**5.1 Optimized matrix**

In Figure 2 of Section 3, if we take R as the primary stream  $P$  and the number of divisions generated by primary stream is  $P_\gamma = \lceil \frac{|R|}{V_n} \rceil$ , then the matrix scheme of example in Figure 2a should have only one column with two cells. However,  $N_c$  calculated in Eq. 4 will have fragment tasks if its rounding up value is not equal to the rounding down value. For example

<b>1</b>	$m_{00}$	<b>1</b>	$m_{00}$	<b>3</b>	<b>new</b>	$m_{01}$
$h_{00}^R: [0, \frac{1}{2}]$ $h_{00}^S: [0, 1]$		$h_{00}^R: [0, \frac{1}{2}]$ $h_{00}^S: [0, 1]$		$h_{00}^R:$ $h_{00}^S:$		
		$s_{00}^R: [0, \frac{1}{2}]$ $s_{00}^S: [0, \frac{1}{2}]$		$s_{01}^R: [0, \frac{1}{2}]$ $s_{01}^S: [\frac{1}{2}, 1]$		
		$n_{00}^R$ $n_{00}^S$		$n_{01}^R$ $[0, \frac{1}{2}]$ $n_{01}^S$ $[\frac{1}{2}, 1]$		
<b>2</b>	$m_{10}$	<b>2</b>	$m_{10}$	<b>4</b>	<b>new</b>	$m_{11}$
$h_{10}^R: [\frac{1}{2}, 1]$ $h_{10}^S: [0, 1]$		$h_{10}^R: [\frac{1}{2}, 1]$ $h_{10}^S: [0, 1]$		$h_{11}^R:$ $h_{11}^S:$		
		$s_{10}^R: [\frac{1}{2}, 1]$ $s_{10}^S: [0, \frac{1}{2}]$		$s_{11}^R: [\frac{1}{2}, 1]$ $s_{11}^S: [\frac{1}{2}, 1]$		
		$n_{10}^R$ $n_{10}^S$		$n_{11}^R$ $[\frac{1}{2}, 1]$ $n_{11}^S$ $[\frac{1}{2}, 1]$		

(a) Old Scheme

(b) New Scheme

**Figure 3** Example of Scheme Change

in Figure 4, given  $V = 10GB$ ,  $|R| = 9GB$ , and  $|S| = 7GB$ , a matrix  $M$  with  $\alpha = 2$  and  $\beta = 2$  will be generated. Since  $S$  is the primary stream, each task first gets data assignment from  $S$  by  $\frac{|S|}{\beta} = 3.5GB$  and the remaining space  $6.5GB$  can be used for divisions from  $R$ . The memory utilization percentage of the two tasks in the last row is 60 %.

In Figure 4, since data for  $m_{11}$  and  $m_{10}$  both join with  $r_1$ , it is then feasible to move tuples in  $s_1$  to  $m_{10}$  to complete the join work but still satisfies memory threshold  $V = 10GB$  ( $2.5 + 3.5 + 3.5 = 9.5 < 10$ ) and promise the completeness of results. Then, an optimized partition scheme with only 3 tasks for  $R \bowtie S$  can be generated, which is much more resource economic. We will study the optimization strategy of matrix model, the definition of tuple routing, and the specific migration procedure in further study.

### 5.2 Optimized routing

For tuple routing in the matrix model, we load data to rows and columns in a top-down manner. Since we have divided the load along the primary stream randomly and also assign the load from the other one evenly to each cell. In such a case, each cell may have free resources. In this section, we propose an optimized routing method to make full use of our resources.

---

#### Algorithm 5 Tuple routing in the varietal model

---

**input:** R stream , S tream

**output:** Matrix side: $\Gamma$ , Line: $\epsilon$

```

1:  foreach tuple  $\tau$  do
2:    if  $\tau$  belongs to the primary stream then
3:      if R stream the primary stream then
4:         $\Gamma \leftarrow row$ 
5:         $\epsilon \leftarrow Random[0 \sim (\alpha - 1)]$ 
6:      else if S stream the primary stream then
7:         $\Gamma \leftarrow column$ 
8:         $\epsilon \leftarrow Random[0 \sim (\beta - 1)]$ 
9:      else if  $\tau$  belongs to the secondary stream then
10:     if S stream the primary stream then
11:        $\Gamma \leftarrow row$ 
12:        $\epsilon \leftarrow Random[0 \sim (\alpha - 2)] \frac{(V - \frac{P}{P_Y}) \cdot (\alpha - 2)}{D}$ 
13:        $\epsilon \leftarrow (\alpha - 1) 1 - \frac{(V - \frac{P}{P_Y}) \cdot (\alpha - 2)}{D}$ 
14:     else if R stream the primary stream then
15:        $\Gamma \leftarrow column$ 
16:        $\epsilon \leftarrow Random[0 \sim (\beta - 2)] \frac{(V - \frac{P}{P_Y}) \cdot (\beta - 2)}{D}$ 
17:        $\epsilon \leftarrow (\beta - 1) 1 - \frac{(V - \frac{P}{P_Y}) \cdot (\beta - 2)}{D}$ 
18:  return  $\Gamma, \epsilon$ 

```

---

We differentiate the two streams as a primary stream  $P$  and a secondary stream  $D$  to find an optimal processing scheme. Here we propose to differentiate the routing policy for the primary stream and the secondary stream that is the tuples in the primary stream will

**Figure 4** Example of the Further Optimization for Matrix

		<b> S =7</b>	
		<b>s<sub>0</sub></b>	<b>s<sub>1</sub></b>
<b> R =9</b>	<b>r<sub>0</sub></b>	<b>m<sub>00</sub></b> <b>6.5,3.5</b>	<b>m<sub>01</sub></b> <b>6.5,3.5</b>
	<b>r<sub>1</sub></b>	<b>m<sub>10</sub></b> <b>2.5,3.5</b>	<b>m<sub>11</sub></b> <b>2.5,3.5</b>

be sent into the row or column randomly while tuples in the secondary stream will be sent selectively instead of randomly.

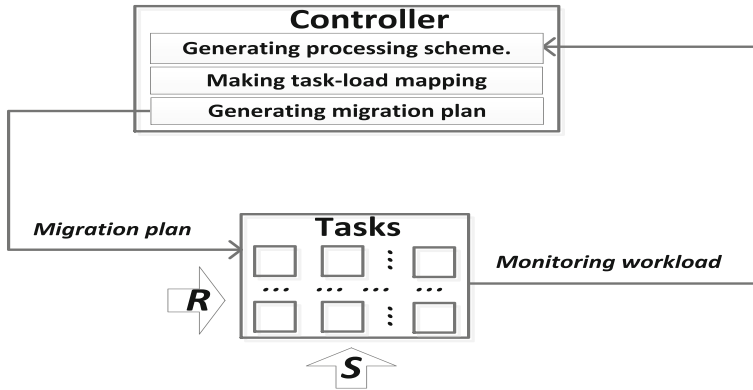
Our algorithm is shown in Algorithm 5 and we take the cells in Figure 2. For the primary stream, line(2~8) randomly split the incoming data into a number of non-overlap substreams. Then, we process the secondary stream. We use  $Random[0 \sim \psi]^\omega \rightarrow \epsilon$  ( $\epsilon \in [0, \psi]$ ) to represent that a tuple randomly selects a line  $\epsilon$  between 0 and  $\psi$  in the probability of  $\omega$ . And then,  $\chi^\omega \rightarrow \epsilon$  means the input tuple will be sent to the  $\chi^{th}$  in the probability of  $\omega$ . In our design, we expect to fill the secondary stream as much as possible to the cells except the last row (S is primary one) or column (S is the secondary one), which is  $V - P/P_\gamma$ . Line (10~13) in Algorithm 5 shows the tuple routing process when the input tuple  $\tau$  belongs to the secondary stream and S stream is the primary stream. Line 12 means the first  $(\alpha - 1)$  rows will have the probability of  $\frac{(V - \frac{P}{P_\gamma}) \cdot (\alpha - 2)}{D}$  to receive the input tuple randomly. And line 13 assigns the input tuple into the last row in the probability of  $1 - \frac{(V - \frac{P}{P_\gamma}) \cdot (\alpha - 2)}{D}$ . Line (14~17) shows the process of tuple routing when the input tuple belongs to the secondary stream and the primary stream is R. This procedure is similar to the process in line (10~13). It may find that the cells in the last row or column managed by the secondary stream will be underloaded. Those cells can be combined to save system resources and then we may get the irregular matrix as discussed in Section 5.1.

### 5.3 Others for optimized scheme

Besides scheme generation and routing tuples as discussed above, there are also others problems needed to be studied for the irregular matrix scheme, such as migration actions and correctness guarantee. Specifically, due to the content that stored in each cell of irregular matrix is different to the regular one, then the migration action will be challenge. Furthermore, the correctness of system during the process of migration also should be re-designed. We will keep this as our future work.

## 6 Evaluation

All of the approaches in our experiment are implemented and run on top of *Apache Storm* [1]. The adaptive processing architecture is shown in Figure 5 and the overall workflow of the adjustment components for distributed stream join is as follows. At the end



**Figure 5** Architecture of Adaptive Processing for Matrix Model

of each time interval (such as 5 seconds), the tasks report the information about current resource usages (such as memory load) to a *controller* module. Then the *controller* decides whether to change the processing scheme; if processing scheme needs change, *controller* first produces a new scheme (Section 3.2); accordingly, it expects to explore the task-load mapping function for mapping tasks in an old scheme to the ones in a new scheme (Section 4.2); Finally, it schedules the data migration among tasks (Section 4.3).

## 6.1 Experimental setup

### 6.1.1 Environment

The *Storm* system (version 0.10.1) is deployed on a 21-instance HP blade cluster with CentOS 6.5 operating system. Each instance in the cluster is equipped with two Intel Xeon processors (E5335 at 2.00GHz) having four cores.

### 6.1.2 Data sets

We evaluate all the approaches using the existing benchmark TPC-H [2] and generate databases using the *dbgen* tool shipped with TPC-H benchmark. Before feeding data to the stream system, we pre-generate and pre-process all the input data sets. Specifically, we adjust the degree of skew on the join attributes by defining skew parameter  $z$  for the Zipf function and we set  $z = 1$  by default. Furthermore, we also use 10GB real social data<sup>1</sup> from Weibo which is the biggest Chinese social media data to test each approach.

### 6.1.3 Queries

We conduct the experiments on three join queries, namely  $E_{Q_5}$ ,  $B_{NCI}$  and  $B_{MR}$ , among which the first two are used in [7, 18].  $E_{Q_5}$  is an equi-join which represents the most expensive operation in query  $Q_5$  from TPC-H benchmark.  $B_{NCI}$  and  $B_{MR}$  are both band-joins, which are different in memory usage by different data selectivity on attribute *Quantity*.

<sup>1</sup>[http://open.weibo.com/wiki/2/statuses/user\\_timeline](http://open.weibo.com/wiki/2/statuses/user_timeline)

```
 $E_{Q_5}$  : SELECT * FROM LINEITEM, REGION, NATION, SUPPLIER WHERE REGION.orderkey =
LINEITEM.orderkey AND LINEITEM.suppkey = SUPPLIER.suppkey AND SUPPLIER.nationkey =
NATION.nationkey
```

```
 $B_{NCI}$  : SELECT * FROM LINEITEM L1, LINEITEM L2 WHERE L1.orderkey - L2.orderkey ≤ 1
AND L1.shipmode = 'TRUK' AND L2.shipinstruck = 'NONE' AND L2.Quantity > 48
```

```
 $B_{MR}$  : SELECT * FROM LINEITEM L1, LINEITEM L2 WHERE L1.orderkey - L2.orderkey ≤ 1
AND L1.shipmode = 'TRUK' AND L2.shipinstruck = 'NONE') AND L2.Quantity > 10
```

We also implement a *Social data query* which is a full band-join and requires each tuple from one stream meets all the tuples from the other stream. We implement both full-history joins and window-based joins, where full-history joins are used to verify system's scalability and window-based joins are used to validate algorithms' flexibility and self-adaptability.

#### 6.1.4 Baseline approaches

For the purpose of comparison, we implement four different distributed stream join algorithms: *MFM*, *Square*, *Dynamic* [7] and *Readj* [11]. *MFM* and *Square* are proposed in this paper. *MFM* denotes our flexible and adaptive algorithm that generates the scheme with less tasks according to Eq. 4. *Square* adopts a naive method to obtain the task number defined in Eq. 2. *Dynamic* [7] assumes the number of tasks in a matrix must be a power of two. If one stream doubles its volume, *Dynamic* adjusts matrix scheme by doubling the cells along the side corresponding to this stream. Meanwhile, it halves cells along the other side of the matrix. Besides, *Dynamic* scales out by splitting the states of every task to four tasks if a task stores a number of tuples exceeding specified memory capacity (Here we do not consider the division of matrix scheme). *Readj* [11] is designed to minimize the load by redistributing tuples based on a hash function on keys. It introduces a similar tuple distribution function, consisting of a basic hash function and an explicit hash table. However, the workload redistribution mechanism used in *Readj* is completely different from ours. The algorithm in *Readj* always tries to move back the keys to their original destination by hash function, followed with migration schedules on keys with relatively larger workload. Their strategy might work well when the granularities of the keys are almost unchanged. When the granularities of keys vary dramatically, their approach either fails to find a reasonable load balancing plan, or incurs huge routing overhead by generating a large routing table.

#### 6.1.5 Evaluation metrics

We measure resource utilization and system performance through the following metrics:

*Task Number* is the total number of tasks used in system and each task is equipped with a constant quota of memory  $V$ ;

*Throughput* is the average number of tuples that processed by system per time unit (second or minute);

*Migration Volume* is the total amount of tuples migrated to other tasks during scheme change;

*MigrationPlanTime* is the average time spent on generating a migration plan.

*LoadRatio* is the ratio of the average load of tasks and the task current load.

## 6.2 Load skewness phenomenon

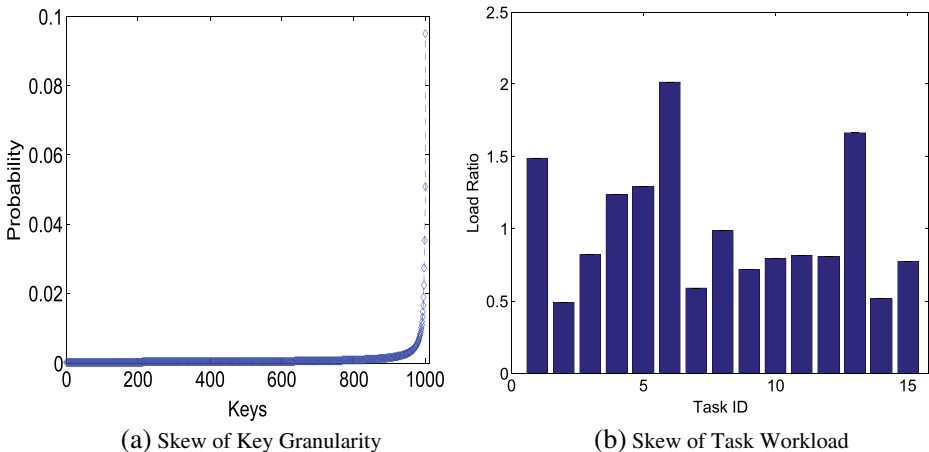
To understand the phenomenon of workload skewness, we report the workload imbalance phenomenon on the task instances by routing keys with the traditional hash-based mechanism. The results of load imbalance (route 1000 keys into 15 tasks) is shown in Figure 6. Figure 6a shows the probability distribution of keys under the skew of  $z = 0.9$ . Figure 6b reflects the load ratio of each task. Among those tasks, the load skewness phenomenon is obvious where the maximal workload is around 4 times larger than the minimal one.

## 6.3 Task consumption of each scheme

The task consumption of each scheme under different loading data volume as shown in Figure 7. As data loading in Figure 7, our algorithms *MFM* and *Square* have stable performance while *Dynamic* meets sharp increase in task number, for *Dynamic* has a strict requirement that the number of tasks must be a power of two. Contrarily, our algorithms *MFM* and *Square* generate the processing scheme based on current workload. Furthermore, *Optimized* produces a smaller scheme than *MFM* and *Square*. This is because it generates the irregular matrix scheme as discussed in Section 5.1. However, how to define the migration action and ensure the correctness of system for irregular matrix scheme are challenges, and we will focus on this work in our future work.

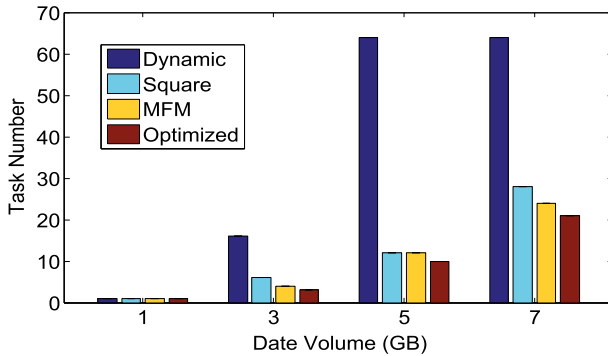
## 6.4 Scalability

To testify the scalability of our join algorithm, we set  $V = 8 \cdot 10^5$  and continue load all  $6 \cdot 10^6$  tuples into our system by executing  $B_{NCJ}$ . Figure 8 shows increasing of task number and migration cost during loading data into the system. With the increase of task number shown in Figure 8a, the memory utilization consumed by *Dynamic* also increases dramatically which is proportional to task number. The naive method *Square* consumes more memory compared to *MFM* since its task number increases a little bit more. Our algorithm *MFM* performs the best among those methods, which can scale out with minimal



**Figure 6** Performance on Workload Skew

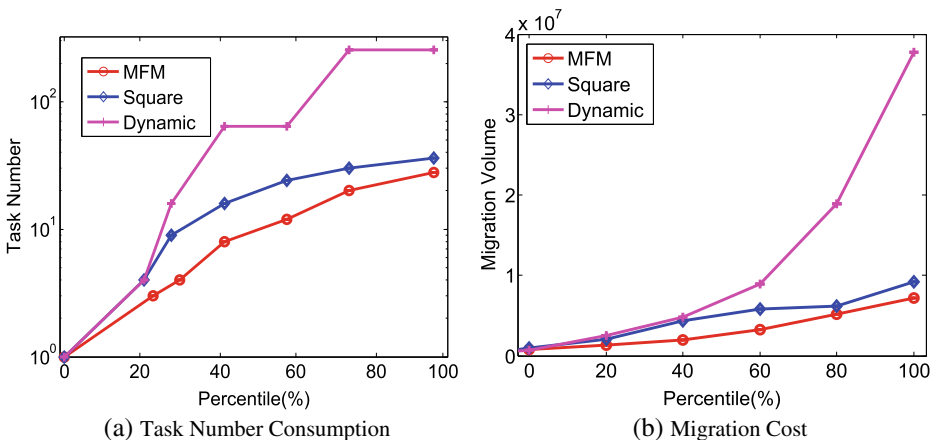




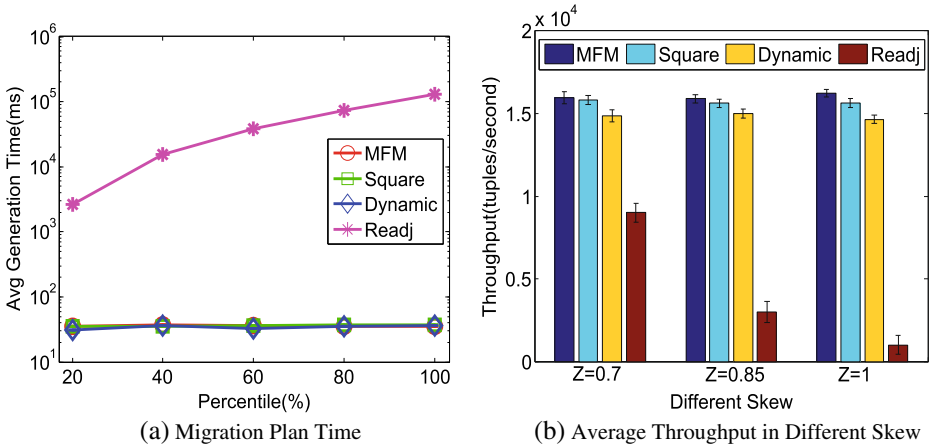
**Figure 7** Task Consumption of Different Scheme while Loading Different Size of Data

number of tasks and apply for resources on its real demand. Figure 8b illustrates the changes on migration cost with query  $B_{NCI}$  when loading the whole dataset into our system. Consistently, *Dynamic* causes the highest migration cost than all other algorithms, because *Dynamic* suffers from massive replications to maintain its matrix structure. Furthermore, *Square* and *MFM* yield low migration volume in that they involve less tasks. From Figure 8 we find that the migration volume increases along with data loading. This is because all matrix schemes progressively get larger.

In addition, we examine the latency for generating migration plan and throughput for equi-join  $E_{Q_5}$  with different algorithms. For the purpose of load balance, we define the balance indicator  $\theta_t$  for task instance  $d$  during time interval  $T_t$  as  $\theta_t = |\frac{L_t(d) - \bar{L}_t}{\bar{L}_t}|$ , where  $\bar{L}_t$  is the average load of all task instances. For this group of experiments, we set  $\theta_t \leq 0.05$ . Figure 9a provides the latency for generating migration plan. Obviously, the latency of *Readj* is much larger than all other algorithms for *Readj* is designed to minimize the load difference among tasks by redistributing data on keys with a hash function and it must recalculate the balance states for each scale-out processing. The other algorithms including



**Figure 8** Performance of Full-history Join with  $B_{NCI}$



**Figure 9** Performance of Full-history with  $E_{Q_5}$

*Dynamic* use random distribution as routing policy, so they need not do calculation for balance scheduling. Figure 9b draws the throughput of each algorithm under different data skewness. Throughput of *Readj* decreases with severer skewness because it spends more time for generating migration plan. Although tasks used by *Dynamic* is much more than our methods, the throughput of ours is more than *Dynamic* due to its massive migration cost.

### 6.5 Dynamics

This group of experiments shows the performance with window-based join, which bounds the memory consumption based on the window size. For this experiment, we set window size as 5 minutes and the average input rate is about  $1.8 \cdot 10^4$  tuples per second. We provide maximum 32 tasks for this group testing. The dynamics is simulated by altering the relative stream volume ratio  $|R|/|S|$  between stream  $R$  and  $S$  [7] with the total volume  $2 \cdot 10^7$  tuples, where the ratio fluctuates between  $f$  and  $\frac{1}{f}$  with  $f$  defined as the fluctuation rate.

Figure 10a and b depict the throughput and number of tasks used for query  $B_{MR}$ . Figure 10a shows that our methods have better throughput compared to *Dynamic*. For our algorithms *MFM* and *Square*, the given 32 tasks are far more than our needs as shown in Figure 10b, while *Dynamic* exhausts all the tasks at any time. This determines the difference of throughputs between *Dynamic* and other ones as shown in Figure 10a.

Figure 10c shows the throughputs of different algorithms under different dynamic ratios  $f$  for Query  $B_{NCI}$ . As described in this paper, the effectiveness of generating migration plan and the network cost of migration determine the efficiency of different algorithms. As shown in Figure 10c, the overall throughput of ours is stable for dynamic ratios  $f$ .

Figure 10d illustrates the throughput of different queries under the workload  $2 \cdot 10^7$  tuples. Because the intermediate results are materialized before being stored in memory, different queries generate different volume of states which are to be stored in memory. Since  $E_{Q_5}$  is lack of filters on predicate, it should store all tuples within a window for join processing and then it requires more memory. In this way, throughput of *Dynamic* decreases dramatically due to its memory requirement for  $E_{Q_5}$ . For the two band-joins,  $B_{NCI}$  will have more throughput for its filters  $Quantity > 48$  can filter out more tuples than  $Quantity > 10$

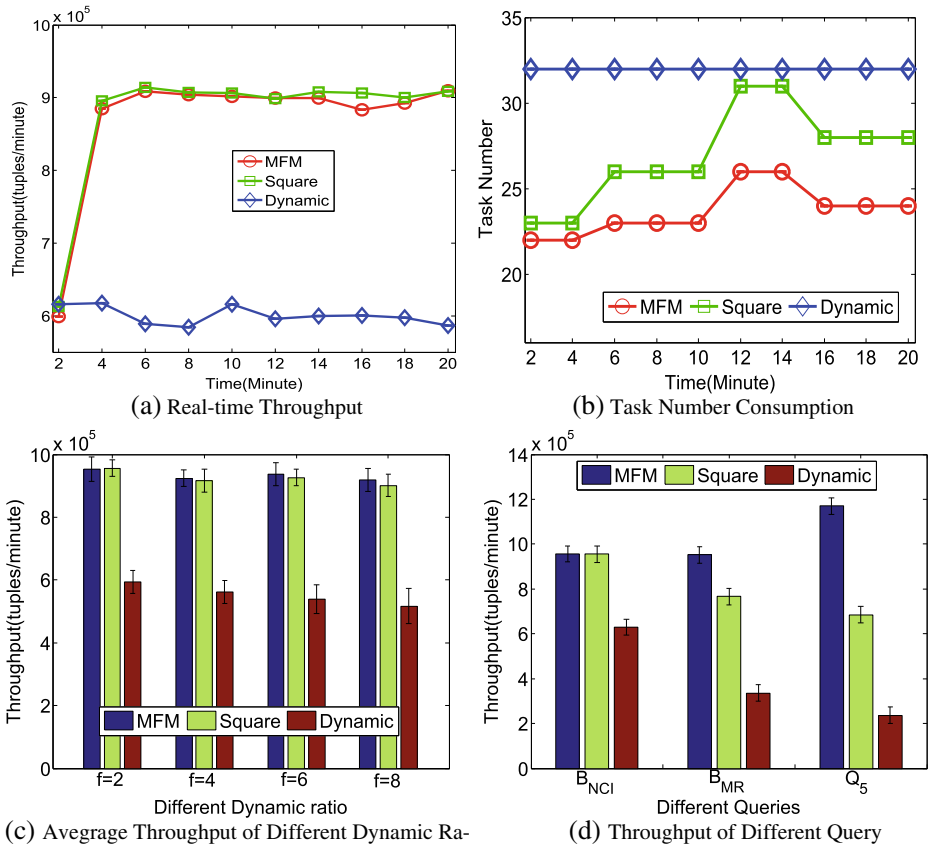


Figure 10 Performance of Window-based Join

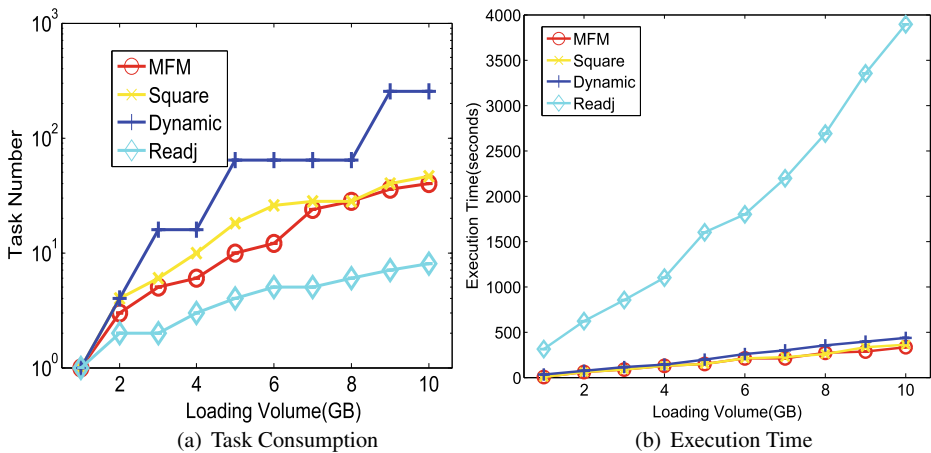
in  $B_{MR}$ . This also indicates that  $B_{MR}$  requires more tasks than  $B_{NCI}$  and it has lower throughput when the total memory size is predefined.

### 6.6 Performance on real data

To prove the usability of our algorithm, we do band-join *Social data query* on 10GB Weibo dataset. We load the 10GB dataset continuously, and measure resource consumption of different algorithms. In Figure 11a, *Readj* uses less tasks, however, it is the slowest for its lack of CPU resources as shown in Figure 11b. Our method *MFM* provides a flexible matrix scheme which applies for new tasks according to its real load while *Dynamic* scales out in a generous way.

## 7 Related work

With the demand of more diverse applications [13, 17, 31, 36, 39], in the past decades, there have been much effort put into designing distributed join algorithms to deal with the rapid growth of data. Blanas et al. Graefe [12] gave an overview of parallel join algorithms.



**Figure 11** Performance on Real Data

However, all these algorithms were mainly proposed for non-streaming scenarios and cannot be directly deployed in streaming processing environments. For non-stream join processing, there also has been much research. To name a few, the symmetric hash join SHJ [32] extends the traditional hash join algorithms and highly supports pipelined processing in parallel database systems. However, it requires that the entire hash tables should be kept in main memory. XJoin [29] is based on SHJ and allow parts of the hash tables to be spilled out to disk for later processing, enhancing the applicability of the algorithm. Similarly, RPJ [27] takes a statistics-based flushing strategy and tries to keep tuples which are more likely to join in memory. Dittrich et al. [6] developed sorted-based but non-blocking progressive merge join algorithm PMJ. However, all these algorithms delegated the processing work to a centralized entity and were not easy to scale when handling the massive data stream workload.

In recent years, there are great interest in designing stream join algorithms in a distributed environment. Photon [3] is a prototype system designed by Google to join data streams such as Web search queries and user clicks on the advertisements. It relies on a central coordinator to support fault-tolerance and scale-out join. It processes incoming tuples through key-value matching in real time in a non-blocking way, but cannot support theta-join well. D-Streams [35] is a data stream operating object defined in Spark Streaming. It adopts mini-batch on data streams in a blocking way. Though it supports theta-join well, some tuples may miss each other due to the constraint of window size. As a result, it can only give approximate join results. TimeStream [25] exploits the resilient substitution and dependency tracking to ensure the dependability of stream computing. It provides MapReduce-like batch processing and non-blocking tuple processing, but encounters high communication cost due to the maintenance of join states. Join-Biclique [18] is based on a bipartite-graph model and supports both full-history and window-based stream joins.

Joining on streams is generally modeled as a matrix, each side of which corresponds to one stream. Stamos et al. [26] adopt the idea of replicating input tuples, extend the fragment and replicate (FR) algorithm [8] and propose a symmetric fragment and replicate algorithm. Okcan [24] employs the join-matrix for processing theta-joins in MapReduce and designs two partitioning schemes, namely 1-Bucket and M-Bucket. The former scheme is content-insensitive and performs load balancing well by assigning equal cells to each region but

suffers from too much replication, while the latter one is content-sensitive because it maps a tuple to a region according to its join key. Due to the nature of MapReduce, the algorithms are offline and require all input statistics must be available beforehand, which incurs blocking behaviors. Consequently, it is more favorable for batch computing rather than stream computing. In data stream scenario, Elseidy et al. [7] present a  $(n,m)$ -mapping scheme dividing the matrix into  $J$  ( $J = n \times m$ ) regions of equal area and introduce the DYNAMIC operator which adjusts the state partitioning scheme adaptively according to data characteristics continuously. However, all the approaches are based on the hypothesis that the number of partitions  $J$  is restricted to powers of two and predefined without intermediate change, and that the ratio of  $|R|$  and  $|S|$  (the number of arrived tuples of two data streams respectively) falls in between  $\frac{1}{J}$  and  $J$ . What's more, the flexibility of the matrix structure is deteriorated when the matrix need to scale out (down).

## 8 Conclusion

In this paper, we propose a novel flexible and adaptive stream join model, called *MFM*, for real-time join processing with arbitrary predicates in distributed and parallel systems. Based on the join-matrix method which can ensure the correctness of join results and be immune to data skewness, the new scheme change algorithm designed in this paper inherits all the advantages of traditional methods but improves them on scalability and effectiveness. We implement our design on Storm and compare it with the other state-of-art work to verify our idea. In the future, we will continue to design a more flexible partitioning scheme algorithm for  $\theta$ -join to break the limits of the matrix shape aiming to take the best usage of system resource.

**Acknowledgments** This work is partially supported by National High Technology Research and Development Program of China (863 Project) No. 2015AA015307, National Science Foundation of China under grant (No. 61232002, No. 61672233 and NO. 61572194).

## References

1. Apache Storm. <http://storm.apache.org/>
2. The TPC-H Benchmark. <http://www.tpc.org/tpch>
3. Ananthanarayanan, R., Basker, V., Das, S., Gupta, A., Jiang, H., Qiu, T., Reznichenko, A., Ryabkov, D., Singh, M., Venkataraman, S.: Photon: fault-tolerant and scalable joining of continuous data streams. In: SIGMOD, pp. 577–588 (2013)
4. Anis Uddin Nasir, M., De Francisci Morales, G., et al.: The power of both choices: Practical load balancing for distributed stream processing engines. In: ICDE, pp. 137–148 (2015)
5. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* **55**(1), 58–75 (2005)
6. Dittrich, J.P., Seeger, B., Taylor, D.S., Widmayer, P.: Progressive merge join: A generic and non-blocking sort-based join algorithm. In: VLDB, pp. 299–310 (2002)
7. Elseidy, M., Elguindy, A., Vitorovic, A., Koch, C.: Scalable and adaptive online joins. In: VLDB, pp. 441–452 (2014)
8. Epstein, R.S., Stonebraker, M., Wong, E.: Distributed query processing in a relational data base system. In: SIGMOD, pp. 169–180 (1978)
9. Fang, J., Wang, X., Zhang, R., Zhou, A.: Flexible and adaptive stream join algorithm. In: APWEB, pp. 3–16 (2016)
10. Fang, J., Zhang, R., Fu, T.Z.J., Zhang, Z., Zhou, A., Zhu, J.: Parallel stream processing against workload skewness and variance. CoRR, abs/1610.05121 (2016)

11. Gedik, B.: Partitioning functions for stateful data parallelism in stream processing. *VLDB J.* **23**(4), 517–539 (2014)
12. Graefe, G.: Query evaluation techniques for large databases. *ACM Comput. Surv. (CSUR)* **25**(2), 73–169 (1993)
13. Huang, X., Cheng, H., Li, R.-H., Qin, L., Yu, J.X.: Top-k structural diversity search in large networks. In: *VLDB*, pp. 1618–1629 (2013)
14. Huebsch, R., Garofalakis, M., Hellerstein, J., Stoica, I.: Advanced join strategies for large-scale distributed computation. In: *VLDB*, pp. 1484–1495 (2014)
15. Ives, Z.G., Florescu, D., Friedman, M., Levy, A., Weld, D.S.: An adaptive query execution system for data integration, vol. 28, pp. 299–310. *ACM* (1999)
16. Kwon, Y., Balazinska, M., et al.: Skewtune: mitigating skew in mapreduce applications. In: *SIGMOD*, pp. 25–36 (2012)
17. Li, J., Liu, C., Liu, B., Mao, R., Wang, Y., Chen, S., Yang, J.-J., Pan, H., Wang, Q.: Diversity-aware retrieval of medical records. *Comput. Ind.* **69**, 81–91 (2015)
18. Lin, Q., Ooi, B.C., Wang, Z., Yu, C.: Scalable distributed stream join processing. In: *SIGMOD*, pp. 811–825 (2015)
19. Liu, B., Zhu, Y., Jbantova, M., et al.: and A dynamically adaptive distributed system for processing complex continuous queries. In: *VLDB*, pp. 1338–1341 (2005)
20. Lu, M., Tang, Y., Sun, R., Wang, T., Chen, S., Mao, R.: A real time displacement estimation algorithm for ultrasound elastography. *Comput. Ind.* **69**, 61–71 (2015)
21. Mao, R., Xu, H., Wu, W., Li, J., Li, Y., Lu, M.: Overcoming the challenge of variety: big data abstraction, the next evolution of data management for aal communication systems. *Communications Magazine, IEEE* **53**(1), 42–47 (2015)
22. Mao, R., Zhang, P., Li, X., Liu, X., Lu, M.: Pivot selection for metric-space indexing. *Int. J. Mach. Learn. Cybern.* **7**(2), 311–323 (2016)
23. Nasir, M.A.U., Serafini, M., et al.: When two choices are not enough: Balancing at scale in distributed stream processing. In: *ICDE* (2016)
24. Okcan, A., Riedewald, M.: Processing theta-joins using mapreduce. In: *SIGMOD*, pp. 949–960 (2011)
25. Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., Zhang, Z.: Timestream: reliable stream computation in the cloud. In: *Eurosys*, pp. 1–14 (2013)
26. Stamos, J.W., Young, H.C.: A symmetric and replicate algorithm for distributed joins. *IEEE Trans. Parallel Distrib. Syst.* **4**(12), 1345–1354 (1993)
27. Tao, Y., Yiu, M., Papadias, D., et al.: Rpj: Producing fasj join results on streams through rate-based optimization. In: *SIGMOD*, pp. 371–382 (2005)
28. ufler, N., Augsten, B., Reiser, A., Kemper, A.: Load balancing in mapreduce based on scalable cardinality estimates. In: *ICDE*, pp. 522–533 (2012)
29. Urhan, T., Franklin, M.J.: Xjoin: A reactively-scheduled pipelined join operator. *Bulletin of the Technical Committee* (2000)
30. Vitorovic, A., ElSeidy, M., Koch, C.: Load balancing and skew resilience for parallel joins. In: *ICDE* (2016)
31. Wang, J., Huang, J.Z., Guo, J., Lan, Y.: Recommending high-utility search engine queries via a query-recommending model. *Neurocomputing* **167**, 195–208 (2015)
32. Wilschut, A., Apers, P.: Dataflow query execution in a aarallel main-memory environment. *Distributed and Parallel Databases* **1**(1), 103–128 (1993)
33. Xing, Y., Hwang, J., Cetintemel, U., Zdonik, S.: Providing resiliency to load variations in distributed stream processing. In: *VLDB*, pp. 775–786 (2006)
34. Xu, Y., Kostamaa, P., Zhou, X., Chen, L.: Handling data skew in parallel joins in shared-nothing systems. In: *SIGMOD*, pp. 1043–1052 (2008)
35. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: *SOSP*, pp. 423–438. *ACM* (2013)
36. Zheng, B., Zheng, K., Xiao, X., Su, H., Yin, H., Zhou, X., Li, G.: Keyword-aware continuous knn query on road networks. In: *ICDE*, pp. 871–882 (2016)
37. Zheng, K., Zheng, Y., Yuan, N.J., Shang, S.: On discovery of gathering patterns from trajectories. In: *ICDE*, pp. 242–253 (2013)
38. Zheng, K., Zheng, Y., Yuan, N.J., Shang, S., Zhou, X.: Online discovery of gathering patterns over trajectories. *IEEE Trans. Knowl. Data Eng.* **26**(8), 1974–1988 (2014)
39. Zhu, Z., Xiao, J., Li, J., Wang, F., Zhang, Q.: Global path planning of wheeled robots using multi-objective memetic algorithms. *Integrated Computer-Aided Engineering* **22**(4), 387–404 (2015)