A-DSP: An Adaptive Join Algorithm for Dynamic Data Stream on Cloud System

Junhua Fang Rong Zhang Yan Zhao Kai Zheng Xiaofang Zhou *IEEE fellow* Aoying Zhou

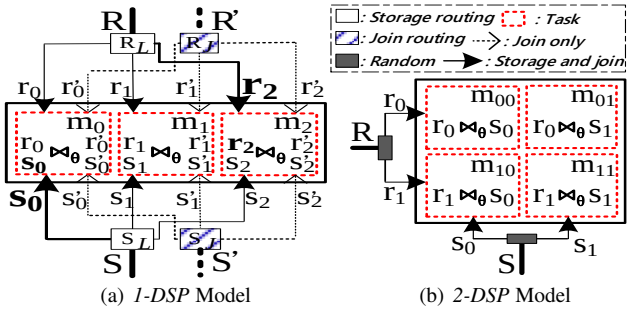
Abstract—The join operations, including both equi and non-equi joins, are essential to the complex data analytics in the big data era. However, they are not inherently supported by existing *DSPEs* (Distributed Stream Processing Engines). The state-of-the-art join solutions on DSPEs rely on either complicated routing strategies or resource-inefficient processing structures, which are susceptible to dynamic workload, especially when the *DSPEs* face various join predicate operations and skewed data distribution. In this paper, we propose a new cost-effective stream join framework, named *A-DSP* (Adaptive Dimensional Space Processing), which enhances the adaptability of real-time join model and minimizes the resource used over the dynamic workloads. Our proposal includes 1) a join model generation algorithm devised to adaptively switch between different join schemes so as to minimize the number of processing task required; 2) a load-balancing mechanism which maximizes the processing throughput; and 3) a lightweight algorithm designed for cutting down unnecessary migration cost. Extensive experiments are conducted to compare our proposal against state-of-the-art solutions on both benchmark and real-world workloads. The experimental results verify the effectiveness of our method, especially on reducing the operational cost under pay-as-you-go pricing scheme.

Index Terms—Distributed stream join; Theta-join; Cost effective

1 Introduction

The big data era has urged the techniques for real-time data processing in various domains, such as stock trading analysis, traffic monitoring and network measurement. Introduction of distributed and parallel processing frameworks enables the efficient processing of massive streams. However, the load balancing issue [16], [21], [23], [34], [35] is emerging as a common problem to those parallel processing frameworks, which deteriorates the performance of the distributed platform on big data processing. Extensive efforts are devoted to tackle the load balancing issue for different operators, including summarization [6], aggregation [25], join [31], [35], [38], [39], etc. Among these operations, θ-join [9], [22], [26] is recognized as the most challenging one, due to its large variety in join predicate options $\{>, \geq, <, \leq, =, \neq\}$.

In practice, θ-join should be supported as a basic operation. For instance, band-joins are common in spatial applications [37], and similarity joins between data sets are also necessary in correlation analysis [26]. There are a handful of approaches designed for join operator over fast and continuous streams, which falls into two categories:*1-DSP* (1-dimensional space processing) model [4], [5], [14], [18], [25] and *2-DSP* (2-dimensional space processing) model [9], [22], [26], [31]. We use m to denote the task instance in *DSPE*. For *1-DSP*, we map the key space K in any stream to *m* tasks by a dividing function *F*(e.g., range function, consistent hashing, etc.), which is defined as $F(k_i) \rightarrowtail m_k$, with $k_i \in K$ and $0 \leq m_k \leq m-1$. As shown in Fig.1(a), it has three tasks numbered by $m_{0\sim2}$. Any key from either stream R or S is supposed to be assigned to one of the tasks. On the other hand, *2-DSP* arranges tasks into two dimensional space, where each dimension represents the division of one stream. As in Fig.1(b), it organizes the tasks as a matrix, represented as m_{ij}, with $0 \leq i \leq 1$ and $0 \leq j \leq 1$. And stream S and R are divided along the horizontal and vertical dimension respectively. More specifically, the processing mode of these two models are explained as follows:



(a) *1-DSP* Model (b) *2-DSP* Model

Fig. 1. Two Typical Paradigms of Existing Stream Join Models.

1-DSP. In Fig.1(a), stream R (S) is split into three sub-streams r_i (s_i) for join processing and stored locally in m_i, with $0 \leq i \leq 2$. However, for non-equi join, a join tuple may get involved in multiple local join tasks according to join predicates (θ). In addition to the locally stored sub-stream r_i in task m_i, additional sub-stream r_i' is sent to m_i for join purpose with locally stored tuples s_i, which contain tuples to be joined but not included in r_i. Therefore, *1-DSP* has two routing methods, one for local storage (*Storage routing*, i.e., R_L) and the other one for join routing (*Join routing*, i.e., R_J) defined by the join predicate as shown in

- *Junhua Fang and Yan Zhao are with the Institute of Artificial Intelligence, School of Computer Science and Technology, Soochow University, China. E-mail: {jhfang, zhaoy}@suda.edu.cn*
- *Kai Zheng is with University of Electronic Science and Technology of China, China. Email: zhengkai@uestc.edu.cn. (Corresponding author)*
- *Rong Zhang and Aoying Zhou are with School of Data Science and Engineering, East China Normal University, China. Email: rzhang@sei.ecnu.edu.cn, ayzhou@dase.ecnu.edu.cn*
- *Xiaofang Zhou is with the School of Information Technology and Electrical Engineering, The University of Queensland, Australia. E-mail: zxf@itee.uq.edu.au.*

Fig.1(a). During the join process, the destination of the incoming tuple, e.g., r_i, is determined by two different routing strategies: *storage routing* R_L decides its storage task, while *join routing* R_J determines to which task(s) r_i shall be assigned to produce join results. In other words, the $R_J(S_J)$ acts only when $R_L(S_L)$ can not make the complete results for any join predicate; otherwise, *join routing* is useless where $R_J = \emptyset$ ($S_J = \emptyset$)

2-DSP. Join-matrix model is a typical represent for *2-DSP*, in which a partitioning scheme is employed on each dimension to split the incoming stream data randomly into a set of mutually exclusive sub-streams to promise the balance requirement. As shown in Fig.1(b), stream R is randomly split into two sub-streams r_0 and r_1 along the vertical side, with $r_0 \cap r_1 = \emptyset$ (s_0 and s_1 for S respectively). Therefore, $R \bowtie_\theta S$ is decomposed into four sub-tasks, each of which m_{ij} ($0 \leq i, j \leq 1$) takes a pair of sub-streams from R and S. The join-matrix model can always return complete results for any join predicate as it can promise each pair of tuples from two data streams meet once.

TABLE 1
Summary of the advantages and disadvantages of different models

	Resource saving	Arbitrary join	Workload balance
1-DSP	✓	×	×
2-DSP	×	✓	✓
A-DSP(Our)	✓	✓	✓

In fact, different processing models have their respective performance characteristics. Tab.1 exhibits the advantages and disadvantages of *1-DSP* and *2-DSP* from perspectives of resource consumption, application scope, and operational status. In accordance to the routing strategies in Fig.1, it is not difficult to find that *1-DSP* is a content-sensitive partitioning scheme, while *2-DSP* uses the random partition strategy determining it is a content-insensitive one. This also explains the fundamental differences exhibited in Tab.1. Specifically, *1-DSP* is a resource-saving join model because it is unnecessary to maintain a specific processing architecture. Instead, it consumes resources on demand. However, having a content-sensitive distribution strategy decides that *1-DSP* is only better for the small-scale join (e.g., equal-join) and clumsy in a dynamic distribution and skewed environment. For *2-DSP*, although its content-insensitive partition strategy enables it to support for the arbitrary join predicate and handle data skewness perfectly, this is at the high cost of task consumption because this model must maintain a two-dimensional processing architecture.

In this paper, we propose a flexible and adaptive model, named *A-DSP* (Adaptive Dimensional Space Processing), for parallel stream join. By integrating the processing strategies of *1-DSP* and *2-DSP*. In other words, we break the limit of processing scheme in *1-DSP* and *2-DSP* with a new solution for parallel θ-join to explore a much larger optimization space: maximizing the resource utilization, supporting various join predicates and guaranteeing workload balance among parallel tasks. In contrast to up-to-date studies, *A-DSP* has several advantages as follows. First, this scheme achieves high flexibility by easily redirecting the keys to new worker threads with simple editing on the routing function. Second, it is highly efficient when the system faces θ-join and dynamic incoming workload, by providing a general strategy to generate the partition function for data balance under dynamic stream. Moreover, workload redistribution with the scheme is scalable and effective, by allowing the system to respond promptly to the short-term workload fluctuation even when there are heavy traffic of tuples present in the incoming data stream.

TABLE 2
Table of Notations

Notations	Description
R, S	R, S stream
C / $\widetilde{C}$	tuple matrix without/with ordered join key
CA /$\widetilde{CA}$	*complete information area* of C/$\widetilde{C}$
ca / $\widetilde{ca}$	coverage area without/with ordered join key
TaskA	the area that is feasible to a single task
$c_{(i,j)}$	the cell located in the i^{th} row and j^{th} column
$\lceil c_{(i,j)}, c_{(i',j')} \rfloor$	the rectangle from $c_{(i,j)}$ to $c_{(i',j')}$
$\widetilde{ca}_x$	the x^{th} $\widetilde{ca}$
$TaskA_{ij}$($TaskA_{ij}^{ca_x}$)	the *TaskA* in the i^{th} row, j^{th} column of ca(ca_x)
$\alpha/\beta(\alpha^{ca_x}/\beta^{ca_x})$	the *TaskA* number of rows/columns in ca(ca_x)
$\widetilde{ca}^r/\widetilde{ca}^s/\overrightarrow{\widetilde{ca}}$	the tuple set of rows/columns/output set in area $\widetilde{ca}$
$V(V_h)$	memory size of task (half size $V_h = \frac{V}{2}$)

Although our preliminary work in [12] has already optimized the join performance by designing a varietal matrix join model based on *2-DSP*, it is still fails to get rid of the inherent and insurmountable disadvantages of regular matrix model. To fully unleash the power of our *A-DSP* model for minimizing the operational cost while maximizing the processing throughput over various join predicate requirement, a suite of optimization techniques are introduced for our syncretic processing scheme to fully exploit the benefits of *1-DSP* and *2-DSP*. In summary, the major contributions of this work are summarized as:

- We propose the *A-DSP* join model to reduce the operational cost with full support of arbitrary join predicates while guaranteeing the correctness of join results.
- We devise the scheme generation algorithms for our *A-DSP* model. Moreover, we present a detailed theoretical analysis for proposed algorithms, and prove their usability and correctness.
- We design a lightweight computation model to support rapid migration plan generation, which incurs minimal data transmission overhead and processing latency.
- We empirically evaluate our model against the state-of-the-art solutions on both standard benchmarks and real-world stream data workloads with detailed explanations on experimental results.

The remainder of this paper is organized as follows. Section 2 introduces the preliminaries. Section 3 describes how to generate the processing scheme. Section 4 explains the detailed implementation of our proposed method. Section 5 shows the experimental results. Section 6 reviews the existing studies on stream join processing in distributed systems. Section 7 finally concludes the paper and discusses future research directions.

2 PRELIMINARIES

2.1 Definitions

Join matrix model organizes the parallel processing tasks as a matrix to handle the join operation between two incoming streams, where each side of which corresponds to one stream. In order to describe our work more intuitively, we define the universal set of tuples arranged by join matrix model as tuple matrix, denoted by C. The tuple matrix enumerates all tuple pairs of the different streams, that is, assigning the Cartesian product of R and S ($R \times$

S) as a tuple matrix. We use cell $c_{(i,j)}$ to denote the tuple pair of the ith row and jth column of tuple matrix. Tab.2 lists the notations commonly used in the rest of the paper.

Definition 1. *Given the streams R and S, if their tuple matrix orders its two side tuples both by their join key, then the matrix is defined as* ***Ordered tuple matrix****, denoted by $\widetilde{C}$.*

Definition 2. *In tuple matrix C/$\widetilde{C}$, we refer to the set of cells that covered by a rectangle as* ***Coverage area****, denoted by ca/$\widetilde{ca}$. The rectangle is represented by $\lceil c_{(i,j)}, c_{(i',j')} \rfloor$, meaning its upper left corner is $c_{(i,j)}$ and lower right is $c_{(i',j')}$.*

Definition 3. *In $\widetilde{C}$, we define the set of cells comprised by a series of 1) no-interval, 2) non-overlapping, and 3) results completion $\widetilde{ca}$s as* ***Complete information area****, denoted by $\widetilde{CA}$. The constraints of $\widetilde{ca}$s are defined as:*

1) *No-interval. This property requires that the complete set of $\widetilde{ca}$s is able to cover all incoming tuples in both the horizontal and vertical directions, which can be expressed as $\bigcup_{\widetilde{ca} \in \widetilde{C}} \widetilde{ca}^r = R$ and $\bigcup_{\widetilde{ca} \in \widetilde{C}} \widetilde{ca}^s = S$.*
2) *Non-overlapping. This property requires that each cell in $\widetilde{CA}$ should be covered by the complete set of $\widetilde{ca}$s at most once, which can be expressed as $\forall \widetilde{ca}_i, \widetilde{ca}_j \in \widetilde{CA}, i \neq j$, then $\widetilde{ca}_i \cap \widetilde{ca}_j = \emptyset$.*
3) *Results completion. This property requires that all join results in $\widetilde{C}$ should be covered by $\widetilde{CA}$, which can be expressed as $\bigcup_{\widetilde{ca} \in \widetilde{C}} \underrightarrow{\widetilde{ca}} = \underrightarrow{\widetilde{C}}$.*

Definition 4. *In tuple matrix, we call the coverage area that assigned to single one task instance as* ***Task area****, denoted by TaskA.*

Fig.2 shows a toy example of join operation using tuple matrix. Each stream (*R*/*S*) has 12 tuples and their join key is denoted by *R.k*/*S.k*. Its join condition is $|R.k - S.k| \leq 2$, and cells which produce join results are marked in grey. Then, the tuple matrix in Fig.2(a) can be seen as a $\widetilde{C}$ according to Def.1, and in Fig.2(c), the union of ca_1 and ca_2 can be regarded as a $\widetilde{CA}$ according to Def.3. Furthermore, based on Def.3, C will be treated as a *complete information area*, since its out-of-order tuples make each cell in C become a potential result ($CA = C$), which can be reflected by Fig.2(b).

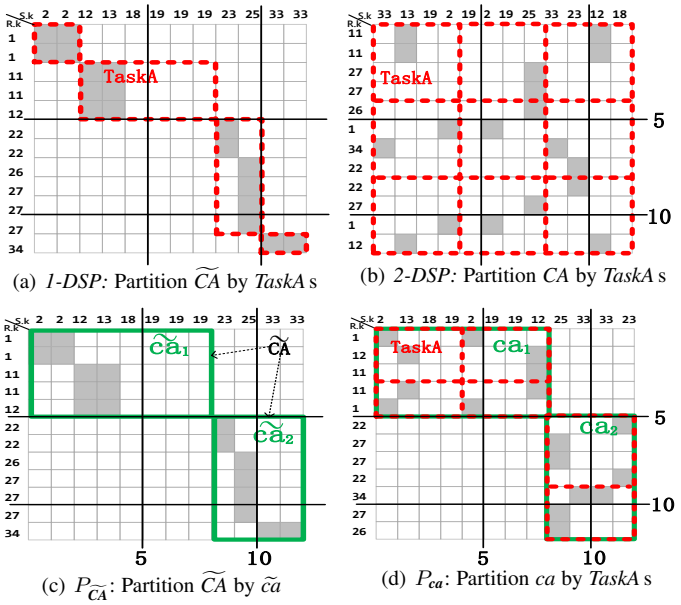
2.2 Models

Based on the above definitions, the essence of parallel processing of those different methods is partitioning the whole tuple matrix into multi-*TaskA* s. As shown in Tab.3, *1-DSP* and *2-DSP* can be expressed as splitting $\widetilde{CA}$ and CA into multi-*TaskA* s, respectively. Fig.2(a) and Fig.2(b) show their data partition results. They also reflect the characteristics of the two methods shown in Tab.1, that is, *1-DSP* uses less resources, but exists workload imbalance, while *2-DSP* is the opposite. Specifically, the workload of each task instance includes input workload and output workload, which can be measured by the number of tuples flowing in and out [31]. For example in Fig.3, we assume the maximum input workload is 8 and the maximum output workload is 5. Then, 4 tasks in Fig.2(a) set an imbalance status while 9 tasks in Fig.2(b) provide a prefect balance both in input and output workload. In fact, only the balance of input workload needs to be taken into account while using *2-DSP*, for its tuple-based random routing strategy can ensure the output results being evenly distributed in tasks. Our subsequent approach inherits this advantage.

TABLE 3
Summary of methods.

Methods	Partition way	Example
1-DSP	$\widetilde{CA} \rightarrow$ *TaskA* s	Fig.2(a)
2-DSP	$CA \rightarrow$ *TaskA* s	Fig.2(b)
A-DSP(Our)	$\widetilde{CA} \xrightarrow{P_{\widetilde{CA}}} ca$ s $\xrightarrow{P_{ca}}$ *TaskA* s	Fig.2(c) & Fig.2(d)

The partitioning process of *A-DSP* model mainly consists of two phases: partition $\widetilde{CA}$ into $\widetilde{ca}$s and partition $\widetilde{ca}$ into *TaskA* s, denoted by $P_{\widetilde{CA}}$ and P_{ca}, respectively. For each $\widetilde{ca}$ generated by $P_{\widetilde{CA}}$, P_{ca} takes it as an unordered ca, and then, partitions it into multi-*TaskA* s using *2-DSP*. Intuitively, using *1-DSP* to generate $\widetilde{ca}$s in $P_{\widetilde{CA}}$ may cause a workload imbalance status. However, those $\widetilde{ca}$s in our *A-DSP* model are usually carried by multiple tasks, which can smooth the workload imbalance. In other words, $\widetilde{ca}$ in *A-DSP* can act as a cushioning layer for workload imbalance. This characteristic is similar to that of task grouping in [22]. Fig.2(c) shows that $P_{\widetilde{CA}}$ takes $\widetilde{ca_1}$ and $\widetilde{ca_2}$ as $\widetilde{CA}$. Then, P_{ca} in Fig.2(d) takes $\widetilde{ca_1}$ as ca_1 and splits it into four *TaskA* s, takes $\widetilde{ca_2}$ as ca_2 and splits it into two *TaskA* s. Compared with *2-DSP*, this scheme consumes less resources (6 task instances) while ensuring each task workload no excess the ideal workload. However, there may be another partition manner which consumes 4 tasks, i.e., Fig.5, and that is what we are after. Next, we are ready to define the optimization goal of our *A-DSP*.



(a) *1-DSP:* Partition $\widetilde{CA}$ by *TaskA* s
(b) *2-DSP:* Partition *CA* by *TaskA* s
(c) $P_{\widetilde{CA}}$: Partition $\widetilde{CA}$ by $\widetilde{ca}$
(d) P_{ca}: Partition ca by *TaskA* s

Fig. 2. A Toy Example of Different Join Models.

2.3 Optimization Goal

For ca, its resource cost includes *Memory*, *Network* and *CPU*, which can be modeled as follows:

Memory. Since tuples are duplicated along rows or columns in P_{ca}, values of α^{ca} and β^{ca} in ca determine the memory consumption C_m considering the join operation, defined as

$$C_m = |ca^r| \cdot \beta^{ca} \cdot R_{size}^{state} + |ca^s| \cdot \alpha^{ca} \cdot S_{size}^{state}, \quad (1)$$

where R_{size}^{state} and S_{size}^{state} are the state sizes in stream R and S, respectively.

Network. When we duplicate the tuples along rows or columns, it consumes the network bandwidth C_n, which is the total communication cost of data transmission among tasks. C_n is defined as

$$C_n = |ca^r| \cdot \beta^{ca} \cdot R_{size}^{tuple} + |ca^s| \cdot \alpha^{ca} \cdot S_{size}^{tuple}, \quad (2)$$

where R_{size}^{tuple} and S_{size}^{tuple} are the tuple sizes in stream R and stream S, respectively.

CPU. For theta-join, we define its computation complexity as

$$C_c = |ca^r| \cdot |ca^s|. \quad (3)$$

In general, it satisfies $R_{size}^{tuple} \propto R_{size}^{state}$ and $S_{size}^{tuple} \propto S_{size}^{state}$ [14]. Then we get $C_n \propto C_m$ based the above discussion and analysis. In other words, minimizing memory usage is equivalent to optimizing network cost. Furthermore, C_c is a constant value which is $|ca^r| \cdot |ca^s|$ and is independent of α^{ca} and β^{ca}. Based on this, we can conclude that optimizing memory is a representative choice.

Suppose the maximum memory size for each task is V and the whole tuple matrix is split into ϵ *ca*s. We formulate our objective as an optimization problem defined as follows:

$$\min \quad \sum_{x \in [1,\epsilon]} \alpha^{ca_x} \cdot \beta^{ca_x}, \quad (4)$$

$$\text{s.t.} \quad Constrains\ in\ Def.3; \quad (5)$$

$$\forall x \in [1,\epsilon], \forall i \in [1, \alpha^{ca_x}], \forall j \in [1, \beta^{ca_x}],$$

$$|(TaskA_{ij}^{ca_x})^r| + |(TaskA_{ij}^{ca_x})^s| \leq V. \quad (6)$$

As the shown in above formulations, our optimization objective is (Exp.4) minimizing the resource cost, while (Exp.5) guaranteeing the correctness of processing and (Exp.6) load balance among all task instances. In fact, under the premise of workload constraint, how to determine $\widetilde{CA}$ to ensure the minimum task consumption is a combinatorial NP-hard problem, as it can be reduced to the Subset-sum problem [28].

3 Scheme Generation

3.1 Partition *ca* into *TaskA* s

Before discussing how to partition ca into *TaskA* s, we first introduce a theorem for guiding the formulation of partition plan.

Theorem 1. *Assuming we use matrix model to handle stream join operation, if there exist α and β in ca with $\frac{|ca^r|}{\alpha} = \frac{|ca^s|}{\beta} = V_h$, then the processing cost for $ca^r \bowtie_\theta ca^s$ is minimum.*

Proof. Assuming the length and width of *TaskA* are a $(a > 0)$ and b $(b > 0)$, respectively, we have the following expression:

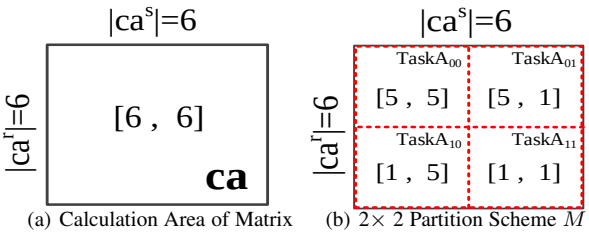
$$a \cdot b \leq \frac{(a+b)^2}{4}. \quad (7)$$

Equ.7 can be modified as the following expression:

$$a + b \geq 2 \cdot \sqrt{a \cdot b}. \quad (8)$$

In the above Equ.7∼8, a+b can be seen as the memory volume of *TaskA* and $a \cdot b$ represents the calculation amount. Each task has the predefined CPU and memory resources. According to Equ.8, for the given CPU resource C_c (represented by the calculation area $a{\cdot}b$), square ($a{=}b$) has the least memory usage C_m. For the given memory C_m, square ($a{=}b$) will produce the least cells and maximize the computation ability according to Equ.7. The network cost C_n can be represented by memory usage C_m as declared in Sec.2.3. Based on the above, we can conclude that if the length ($\frac{|ca^r|}{\alpha}$) and the width ($\frac{|ca^s|}{\beta}$) of the subrange derived from ca partition are equal, that is, equal to half of the memory size (V_h), then the system consumes the least resources including CPU, memory and network.

□



(a) Calculation Area of Matrix (b) 2× 2 Partition Scheme M

Fig. 3. A Toy Example of Scheme Partition at $|ca^r|$=6 and $|ca^s|$=6.

According to Theo.1, the numbers of row and column should be $\lceil \frac{|ca^r|}{V_h} \rceil$ and $\lceil \frac{|ca^s|}{V_h} \rceil$, respectively. Then the total number of tasks used in ca can be expressed as

$$N = \lceil \frac{|ca^r|}{V_h} \rceil \cdot \lceil \frac{|ca^s|}{V_h} \rceil. \quad (9)$$

Fig.3 shows a ca partition example with the stream volumes for R and S are $|ca^r| = 6GB$ and $|ca^s| = 6GB$. Given task memory $V = 10GB$ and $V_h = 5GB$, ca can be divided as shown in Fig.3(b). The generated scheme in Fig.3(b) takes up 4 tasks with two rows and two columns (4 divisions: s_0, s_1, r_0 and r_1). Among those *TaskA* s, we load data to rows and columns by V_h in a top-down manner. In the last row and column of ca, the *TaskA* s may have less data divisions compared to previous *TaskA* s and are called fragment tasks filled with fragment data. In this example, $TaskA_{00}$ is first fed by V_h for both ca^r and ca^s streams and then $TaskA_{00} = (5GB, 5GB)$. However the last column and row can not be filled up. According to Equ.9, though we can calculate the number of tasks needed for joining processing, workload are not evenly distributed among tasks because of the fragment data. Then tasks for fragment data may have free resources compared to other tasks, and it will be more obvious for the large matrix.

In order to facilitate the subsequent description, we name the two join streams as a primary stream P and a secondary stream D. Then we take the primary stream P as the basis volume to calculate how much memory should be assigned for it in each task and accordingly, the secondary stream D gets the remaining portion of memory in each cell. Specifically, we use P_γ to represent the number of divisions generated by primary stream, and then for the secondary stream, its division number D_γ^c can be calculated as

$$D_\gamma^c = \lceil \frac{|D|}{V - \frac{|P|}{P_\gamma}} \rceil. \quad (10)$$

P may be either stream R or S. We use N_c to represent the number of tasks for processing join between P and D in ca which is calculated as:

$$N_c = P_\gamma \cdot D_\gamma^c = P_\gamma \cdot \lceil \frac{|D|}{V - \frac{|P|}{P_\gamma}} \rceil. \quad (11)$$

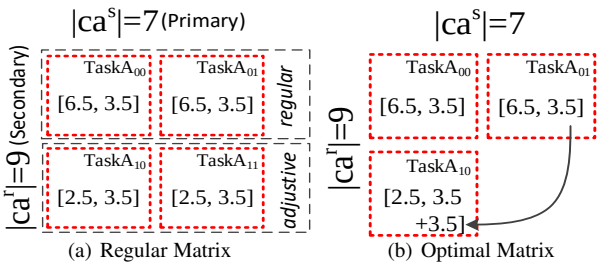


Fig. 4. Example with the Optimized Scheme

In Fig.3, if we take R as the primary stream P and the number of divisions generated by primary stream is $P_\gamma = \lceil \frac{|ca^r|}{V_h} \rceil$, then the matrix scheme of example in Fig.3(a) should have only one column with two cells. However, D^c_γ calculated in Equ.10 will have fragment tasks if its rounding up value is not equal to the rounding down value. For example in Fig.4(a), given $V = 10GB, |ca^r| = 9GB,\ and\ |ca^s| = 7GB$, a ca with $\alpha = 2$ and $\beta = 2$ will be generated. Sine S is the primary stream, each task first gets data assignment from S by $\frac{|ca^s|}{\beta} = 3.5GB$ and the remaining space $6.5GB$ can be used for divisions from ca^r. The memory utilization percentage of the two tasks in the last row is 60%.

An optimized partition scheme of Fig.4(a) is shown as Fig.4(b) , which takes up only 3 tasks for $ca^r \bowtie_\theta ca^s$. In Fig.4(b), the replica $|TaskA^s_{01}| = 3.5GB$ of ca^s in $TaskA_{01}$ are sent to $TaskA_{10}$ to join with subset $|TaskA^r_{10}| = 2.5GB$ of ca^r to promise the correctness and completeness of join results. It is feasible to move tuples in $TaskA^s_{01}$ to $TaskA_{10}$ to complete the join work for the memory threshold $V = 10GB$ remains satisfied.

Compared to the regular matrix scheme in Fig.4(a), the optimal workload assignment in Fig.4(b) can make better use of resource among tasks. Let's take Fig.4(b) to explain the generation of the irregular line, which is the last row. Given the divisions for primary stream S, which is $\beta = P_\gamma = 2$ in Fig.4(a), we divide those α rows for secondary stream R into two types: regular ones (the first row) and adjustive ones (the second row). For regular ones, we have $\alpha^f = D^f_\gamma$ lines of segments (labeled as *regular* in Fig.4(a)), where $D^f_\gamma = \lfloor \frac{|D|}{V - \frac{|P|}{P_\gamma}} \rfloor$. By using these regular division, we may not manage all the workload for secondary stream R, since $D^f_\gamma \cdot (V - \frac{|P|}{P_\gamma}) \leq |D|$. For the remaining loads D^L from D calculated as Equ.12, it is expected to add additional P_δ (as in Equ.12) number of columns along the last row with $P_\delta < P_\gamma$.

$$\begin{aligned} D^L &= |D| - D^f_\gamma \cdot (V - \frac{|P|}{P_\gamma}) \\ P_\delta &= \lceil \frac{|P|}{V - D^L} \rceil \end{aligned} \tag{12}$$

The difference from previous regular matrix is that we will not add one whole row (column) of tasks for the join work with the purpose of making full use of system resources. Accordingly, the optimal number of tasks N_o for $R \bowtie_\theta S$ can be calculated as Equ.13:

$$N_o = \begin{cases} P_\gamma \cdot D^f_\gamma, & |D| = D^f_\gamma \cdot (V - \frac{|P|}{P_\gamma}), \\ P_\gamma \cdot D^f_\gamma + P_\delta, & \text{otherwise.} \end{cases} \tag{13}$$

As described in Theo.1, we use V_h as the divisor to find P_γ for generating an optimal scheme by probing all four cases. Specifically, we make $P_\gamma \in \{\lceil \frac{|ca^r|}{V_h} \rceil, \lfloor \frac{|ca^r|}{V_h} \rfloor, \lceil \frac{|ca^s|}{V_h} \rceil, \lfloor \frac{|ca^s|}{V_h} \rfloor\}$ in Equ.13.

The algorithm of finding an optimal partition scheme is described in Alg.1. Firstly, the minimal number of tasks is determined in line 1 according to Equ.13, by enumerating all the case of round up and down; then in line $2 \sim 5$, the number of rows α and columns β can be calculated according to values of P and P_γ. If $P_\gamma \in \{\lceil \frac{|ca^r|}{V_h} \rceil, \lfloor \frac{|ca^r|}{V_h} \rfloor\}$, R is the primary stream, or else S is the primary one. The scheme of varietal matrix generated from Alg.1 can be expressed as $< \alpha, \beta, P_\delta, + >$ which means that the varietal matrix has α rows, β columns and has P_δ additional cell in $+$ ($+ \in row, column$).

algorithm 1 Partition *ca* to *TaskA* s

input: Stream ca^r, Stream ca^s, Memory size V
output: Row number α, Column number β, additional cells P_δ, additional position $+$

1: $N_o \leftarrow$ Get the minimum task consumption according to Equ.13 where $P_\gamma \in \{\lceil \frac{|ca^r|}{V_h} \rceil, \lfloor \frac{|ca^r|}{V_h} \rfloor, \lceil \frac{|ca^s|}{V_h} \rceil, \lfloor \frac{|ca^s|}{V_h} \rfloor\}$.
2: **if** $P_\gamma \in \{\lceil \frac{|ca^r|}{V_h} \rceil, \lfloor \frac{|ca^r|}{V_h} \rfloor\}$ **then**
3: $\quad \alpha \leftarrow P_\gamma, \beta \leftarrow D^f_\gamma, P_\delta \leftarrow N_o - P_\gamma \cdot D^f_\gamma, + \leftarrow column$
4: **else**
5: $\quad \alpha \leftarrow D^f_\gamma, \beta \leftarrow P_\gamma, P_\delta \leftarrow N_o - P_\gamma \cdot D^f_\gamma, + \leftarrow row$
6: **return** $\alpha, \beta, P_\delta, +$

We still use Fig.4(b) to explain Alg.1. According to Equ.12 , Equ.13 and line 1 in Alg.1, S is the primary stream and R is the secondary one, in which $\beta = 2$, $D^f_\gamma = \lfloor \frac{9}{10 - 7/2} \rfloor = 1$. $TaskA_{00}$ and $TaskA_{01}$ are supposed to have regular workload assignments with $3.5GB$ from ca^s, and the remaining memory is used for ca^r, which will be $10 - 3.5 = 6.5GB$. Since ca^r has load more than $6.5GB$, we add one more row which is the adjustive division for the remaining data of ca^r that $D^L = 9 - 6.5 = 2.5GB$. In such a case, the *TaskA* s in regular division lines have full usage to memory $6.5 + 3.5 = 10GB$ as in $TaskA_{00}$ and $TaskA_{01}$. For adjustive purpose, we add $P_\delta = 1$ *TaskA* to join the remaining data D^L in R with S, which belongs to *TaskA* $TaskA_{10}$ and $TaskA_{11}$ in Fig.4(a). In this adjustive row, D^L will join with the subset of ca^s for $TaskA_{10}$ and $TaskA_{11}$ (3.5GB) and then the workload in $TaskA_{10}$ comes to be $9.5GB$. The varietal matrix scheme shown in Fig.4(b) can be expressed as $< 1, 2, 1, row >$.

Theorem 2. *Alg.1 will consume less tasks than using Theo.1 directly and ensure the correctness of operation when using matrix model for $ca^r \bowtie_\theta ca^s$ with the memory size of each task V.*

Proof. Assume that there exists another varietal matrix M' with scheme $< \alpha', \beta', P'_\delta, +' >$, and the number of tasks N' used in M' is smaller than N_o. To find the smaller N_o, Alg.1 tries all possible values that make $\frac{|P|}{P_\gamma}$ nearest to V_h according to Equ.12, Equ.13 and line 1. In other words, it is impossible for $\frac{|ca^r|}{\alpha'}$ and $\frac{|ca^s|}{\beta'}$ to get closer to V_h than $\frac{|ca^r|}{\alpha}$ and $\frac{|ca^s|}{\beta}$. According to Theo.1 that squared cell shape consumes the minimal resources, the assumption of existing regular matrix M' does not hold. □

3.2 Partition $\widetilde{C}$ into $\widetilde{ca}$s (*ca*s)

In this subsection, we introduce a heuristics algorithm to accomplish the optimization problem with theoretical perspectives. Specifically, here we present how *A-DSP* provides a theoretical

guarantee upper-bound for task consumption to partition $\widetilde{C}$ into $\widetilde{ca}$s (cas) . Using tuple matrix model to handle θ-join, a key observation is that the distribution of $\overrightarrow{\widetilde{CA}}$ has the feature of *Monotonicity*. The *monotonicity* can be expressed as if $c_{(i,j)}$ is not a result cell, then either all cells (k,l) with $k \leq i$ and $l \geq j$, or all cells (k,l) with $k \geq i$ and $l \leq j$ are also not result cells [26], [31]. It should be pointed out that the "$\neq$" operation is not monotonic, for its operation semantic making each cell in $\widetilde{C}$ become a potential result. In such cases, our *A-DSP* model will adaptively adopt the *2-DSP* architecture, which can be reflected from Alg.2 later. Next, we define a partition way for $\widetilde{CA}$ as follows.

Definition 5. *Splitting the area $\widetilde{ca}$ into two sub-areas: $\widetilde{ca}^d$ and $\widetilde{ca}_d$, if $\widetilde{ca}^d \cup \widetilde{ca}_d = \widetilde{ca}$ and $\widetilde{ca}^d \cap \widetilde{ca}_d = \emptyset$, we call this area partition action as $\widetilde{ca}$'s **area dichotomy**, denoted as $AD(\widetilde{ca})$.*

Based on the above Def.5, we call d as the split point and the minimum task consumption of $\widetilde{CA}$ based on area dichotomy can be expressed as:

$$N^{AD(\widetilde{CA})} = Min_{d \in \widetilde{CA}_D}\{N^{AD(\widetilde{CA}^d)} + N^{AD(\widetilde{CA}_d)}\}, \quad (14)$$

where $\widetilde{CA}_D$ is the universal set of split point in $\widetilde{CA}$. Equ.14 is an iteration expression to enumerate all split scheme to find the minimal task consumption. Thus, we have the following theorem.

Theorem 3. *Among all schemes generated by $\widetilde{CA}$'s area dichotomy, Equ.14 can denote the one which with minimum task consumption.*

Proof. Assume there is a partition scheme with the set of split point $\{d_i\}$, where d_i is the split point of the i^{th} iteration partition, and N_{min} is the task consumption by this partition scheme. If $N_{min} < N^{AD(\widetilde{CA})}$, then we have $\{d_i\} \nsubseteq \widetilde{CA}_D$, which contradicts to the assumption that $\widetilde{CA}_D$ is the universal set of split point in $\widetilde{CA}$ in Equ.14. □

Essentially, Equ.14 is a combinatorial mathematics problem and its iteration times is a Catalan number [17]. Directly adopting a exhaustive enumeration method to obtain the partition scheme will undoubtedly incurs the high computational complexity, reaching up to $\mathcal{O}(\frac{4^{|\widetilde{CA}_D|}}{|\widetilde{CA}_D|^{\frac{3}{2}} \cdot \sqrt{\pi}})$. Therefore, we define an iteration termination condition as follows.

Definition 6. *Using area dichotomy to split the coverage area $\widetilde{ca}$ into two sub-areas $\widetilde{ca}^d$ and $\widetilde{ca}_d$, the iteration termination condition is $\forall d \in \widetilde{ca}_D$, $N^{\widetilde{ca}} \geq N^{\widetilde{ca}^d} + N^{\widetilde{ca}_d}$.*

Alg.2 lists the steps of $\widetilde{CA}$ partition based on the above discussions. At the initialization stage of Alg.2, the matrix is generated by the Cartesian produce of the two join streams. First, it finds the split point according to Def.5 in line 4. Second, Alg.2 judges whether to stop the iterative process, and gathers the corresponding split results in lines 5 $\sim$ 10. Finally, it picks out the partition result which with the minimal task consumption as its output in line 11. We continue using the example in Fig.2 to illustrate Alg.2, and the partition result is shown in Fig.5. Alg.2 takes the whole tuple matrix $\widetilde{C}$ as iterative entry and splits it into two areas: $ca_1(\lceil c_{(1,1)}, c_{(5,5)} \rfloor)$ and $ca_2(\lceil c_{(6,6)}, c_{(12,12)} \rfloor)$, using *area dichotomy*. For ca_1, Alg.2 stops splitting it because the continuing split action can not reduce task consumption. Due to $N^{ca_2} = 4$ and $N^{AD(ca_2)} = 2$, Alg.2 picks the fewer tasks consumption scheme by splitting $\lceil c_{(6,6)}, c_{(12,12)} \rfloor$ into two sub-areas. Finally, Alg.2 generates a scheme that consumes 4 tasks.

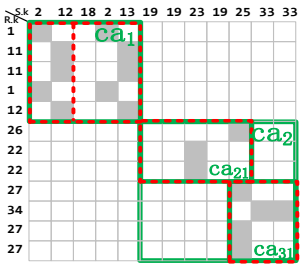


Fig. 5. Partition the Example in Fig.2 using Alg.2

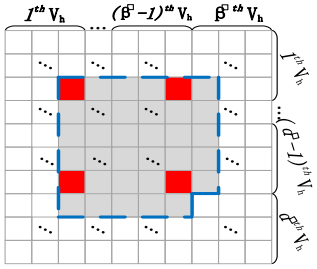


Fig. 6. Illustration of the Worst Case of using Area Dichotomy.

algorithm 2 Partition $\widetilde{C}$ to cas

input: $\widetilde{C}$
output: cas
1: $\widetilde{ca} \leftarrow \widetilde{CA}$
2: **function** PARTITIONAREA($\widetilde{ca}$)
/*$c_{(b,b)}/c_{(e,e)}$: the beginning/ending cell of $\widetilde{ca}$*/
3: **foreach** $c_{(i,j)}$, $c_{(i',j')}$ in $\widetilde{ca}$ **do**
4: **if** $\lceil c_{(b,b)}, c_{(i,j)} \rfloor \cup \lceil c_{(i',j')}, c_{(e,e)} \rfloor = \widetilde{ca}$ *and* $\lceil c_{(b,b)}, c_{(i,j)} \rfloor \cap \lceil c_{(i',j')}, c_{(e,e)} \rfloor = \emptyset$ **then**
/*According to Def.5*/
5: **if** $N^{\lceil c_{(b,b)}, c_{(i,j)} \rfloor} + N^{\lceil c_{(i',j')}, c_{(e,e)} \rfloor} < N^{\widetilde{ca}}$ **then**
/* According to Def.6*/
6: PARTITIONAREA($\lceil c_{(b,b)}, c_{(i,j)} \rfloor$)
7: PARTITIONAREA($\lceil c_{(i',j')}, c_{(e,e)} \rfloor$)
8: **else**
9: $cas_{(i,j)} \leftarrow \lceil c_{(b,b)}, c_{(e,e)} \rfloor$
10: CAs $\leftarrow cas_{(i,j)}$
/* $cas_{(i,j)}$: the set of coverage area be derived from the first iteration, CAs: the set of $cas_{(i,j)}$*/
11: get cas ($cas \in CAs$) with the minimal $\sum_{ca \in cas} N^{ca}$
/* According to Theo.3*/
12: **return** cas

Next, we discuss the task consumption upper-bound of Alg.2.

Definition 7. *Given area $\widetilde{ca}$, suppose there is a minimum task consumption $N_1^{\widetilde{ca}}$, and the task consumption by Alg.2 is $N_2^{\widetilde{ca}}$, we refer to the ratio of $\frac{N_1^{\widetilde{ca}}}{N_2^{\widetilde{ca}}}$ as consumption coefficient, denoted by ξ.*

Lemma 1. *For the area $\widetilde{ca}$ with $\alpha^{\widetilde{ca}} \geq 2$ and $\beta^{\widetilde{ca}} \geq 2$, if Alg.2 stops splitting it, then the cell set RCc in $\widetilde{ca}$, where*

$$RCc = \{c_{(k,l)} | (\alpha^{\widetilde{ca}} - 1) \cdot V_h + 1 \geq k \geq V_h, (\beta^{\widetilde{ca}} - 1) \cdot V_h + 1 \geq l \geq V_h\} - c_{(\alpha^{\widetilde{ca}}-1) \cdot V_h + 1, \beta^{\widetilde{ca}}-1) \cdot V_h + 1)},$$

should be all the result.

Proof. To verify in the worst case, we use the naive method to measure the task consumption according to Equ.9, where $N^{\widetilde{ca}} = \alpha^{\widetilde{ca}} \cdot \beta^{\widetilde{ca}} = \lceil \frac{|\widetilde{ca}^r|}{V_h} \rceil \cdot \lceil \frac{|\widetilde{ca}^s|}{V_h} \rceil$. The formation of RCc can be accumulated in three steps:
Step-1 (According to Def.6): The precondition of Lem.1 that Alg.2 stops splitting area $\widetilde{ca}$ means $\forall d \in \widetilde{ca}_D$, $N^{\widetilde{ca}} < N^{\widetilde{ca}^d} + N^{\widetilde{ca}_d}$. Then, the cells $\{c_{(k,l)} | k\%V_h = 0, l\%V_h = 0, k \neq$

$|\widetilde{ca}^r|, l \neq |\widetilde{ca}^s|\}$ (e.g., the red cells in Fig.6) are definitely not split points, otherwise, the scheme cannot be the worst case among all partition schemes. They should be contained by RCc.

Step-2 (According to Def.5): Due to the fact that *area dichotomy* splits the area $\widetilde{ca}$ into two sub-areas and $c_{(i,j)}$ in $\widetilde{ca}$ is not a split point, then, existing cells $c_{(i,j')\in\widetilde{ca}}(i < i')$ and $c_{(i',j)} \in \widetilde{ca}(j < j')$ are both result. Based on this, the cells in RCc are expanded to

$$\{c_{(k,l)}, c_{(k,l+1)}, c_{(k+1,l)} | k\%V_h = 0, l\%V_h = 0, k \neq |\widetilde{ca}^r|, l \neq |\widetilde{ca}^s|\}.$$

Step-3 (According to *monotonicity*): The *monotonicity* defines that the cell(s) between result cells are also results. Combining this, the result cells RCc can be expressed as

$$\{c_{(k,l)} | (\alpha^{\widetilde{ca}} - 1) \cdot V_h + 1 \geq k \geq V_h, (\beta^{\widetilde{ca}} - 1) \cdot V_h + 1 \geq l \geq V_h\} - c_{(\alpha^{\widetilde{ca}}-1)\cdot V_h+1, \beta^{\widetilde{ca}}-1)\cdot V_h+1)},$$

which equivalents to the description of Lem.1 (surrounded by blue dotted line in Fig.6). □

Then, the upper-bound of ξ can be expressed as follows:

Theorem 4. *Using Alg.2 to partition C can ensure an upper-bound of its task consumption: $\xi \leq \frac{\alpha^C \cdot \beta^C}{\alpha^C \cdot \beta^C - \alpha^C - \beta^C + 3}$, where $\alpha^C \geq 2$ and $\beta^C \geq 2$.*

Proof. According to Lem.1, the minimal task consumption is consisted of three parts: $\lceil c_{(1,1)}, c_{(V_h,V_h)} \rfloor$ consumes 1 task, $\lceil c_{(V_h+1,V_h+1)}, c_{((\alpha^{\widetilde{ca}}-1)\cdot V_h, (\beta^{\widetilde{ca}}-1)\cdot V_h)} \rfloor$ consumes $(\alpha^{\widetilde{ca}} - 1) \cdot (\beta^{\widetilde{ca}} - 1)$ task(s) and $\lceil c_{((\alpha^{\widetilde{ca}}-1)\cdot V_h+1, (\beta^{\widetilde{ca}}-1)\cdot V_h+1)}, c_{(|R|,|S|)} \rfloor$ consumes 1 task. Then, we have $N_2^{\widetilde{ca}} \geq 1 + (\alpha^{\widetilde{ca}} - 1) \cdot (\beta^{\widetilde{ca}} - 1) + 1$. Since $\xi = \frac{N_1^{\widetilde{ca}}}{N_2^{\widetilde{ca}}}$ and $N_1^{\widetilde{ca}} = \alpha^{\widetilde{ca}} \cdot \beta^{\widetilde{ca}}$, we have $\xi \leq \frac{\alpha^{\widetilde{ca}} \cdot \beta^{\widetilde{ca}}}{1+(\alpha^{\widetilde{ca}}-1)\cdot(\beta^{\widetilde{ca}}-1)+1}$. Making C as $\widetilde{ca}$, Theo.4 is verified. □

The statement of Theo.4 gives an estimation or hint on the worst case of task consumption. However, just as the unique distribution shown in Fig.6, the probability of a worst case scenario is negligible, which can be validated by the experimental results in Sec.5. Algo.2 uses *area dichotomy* to split the coverage area $\widetilde{ca}$ into two sub-areas $\widetilde{ca}^d$ and $\widetilde{ca}_d$, and its iteration termination condition is $\forall d \in \widetilde{ca}_D$, $N^{\widetilde{ca}} \geq N^{\widetilde{ca}^d} + N^{\widetilde{ca}_d}$, its computational complexity is not more than T($|\widetilde{ca}^r| \cdot |\widetilde{ca}^s|$), where T($|\widetilde{ca}^r| \cdot |\widetilde{ca}^s|$)= $\kappa \cdot$ T($|\widetilde{ca}^r| \cdot |\widetilde{ca}^s|$/2)+$\mathcal{O}(|\widetilde{ca}^r| \cdot |\widetilde{ca}^s|)$. As illustrated in Fig.6 and Theo.4, the cells $\{c_{(k,l)} | k\%V_h = 0, l\%V_h = 0, k \neq |\widetilde{ca}^r|, l \neq |\widetilde{ca}^s|\}$ (the four red cells in Fig.6) are definitely not split points, then $\kappa = 4$ and T($|\widetilde{ca}^r| \cdot |\widetilde{ca}^s|$) = $\mathcal{O}((|\widetilde{ca}^r| \cdot |\widetilde{ca}^s|)^2)$.

4 IMPLEMENTATION

The adaptive processing architecture is shown in Fig.7 and can be decomposed into the following five steps: **Step-1: Monitoring workload.** It collects information about the tuple matrix and the space occupied by the two join streams, and the task consumption of the existing processing scheme. On receiving the reporting information, the controller first evaluates the number of task consummation according to Algo.1 and Algo.2. **Step-2: Generating processing scheme.** It produces a new scheme according to workload and resource information, as presented in Sec.3; **Step-3: Making task-load mapping.** It expects to find the task-load mapping function so as to maximize non-moving data volume

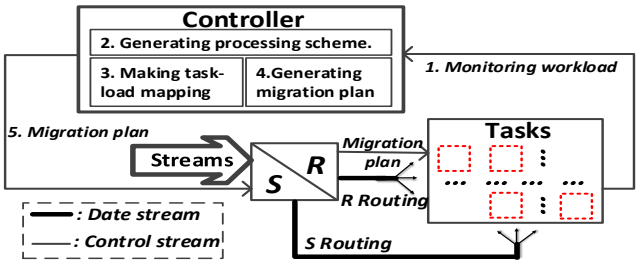


Fig. 7. Architecture of Adaptive Processing.

when transforming from the old scheme to the new scheme. This step will be described in Sec.4.2. **Step-4: Generating migration plan.** It schedules the data migration among tasks according to task-load mapping scheme. This step will be described in detail in Sec.4.3. **Step-5: Defining consistent protocol according to migration plan.** This step defines consistent protocols to ensure the correctness of system.

We omit the detailed description of step-5 in this paper since it is same as [9]. Moreover, the implementation process of our model should include two phases: $P_{\widetilde{CA}}$ (partition $\widetilde{CA}$ into $\widetilde{ca}$ s) and P_{ca} (partition $\widetilde{ca}(ca)$ into *TaskA* s). After the final scheme generated by Alg.1 and Alg.2, the workload adjustment strategy among ca s is a key-based issue which has been explored in our previous work [11], so that will not be reiterated in this paper. Due to the fact that the routing strategy is the essential step for the incoming stream, we first introduce it in the following subsection.

4.1 Routing Strategy

In view of our *A-DSP* model is a composite structure, its routing strategy includes two layers as listed in Alg.3.

algorithm 3 Routing Strategy Algorithm

input: Input tuple: τ, Mapping function: F
output: Routing destination RD: $\{(ca, \ddagger, th)\}$

```
1: k ← E(τ)  /* extract the key of tuple τ*/
2: /**Step-1: Routing tuples in C̃ ***/.
3: Ca ← RoutingInC̃(C̃, k, F )  /* Ca is the set of ca*/
4: /**Step-2: Routing tuples in ca ***/.
5: RD ← RoutingInCa( Ca,τ )
6: return RD
7: function RoutingInC̃(C̃, k, F )
8:     K* ← F(k)
9:     foreach k* ∈ K* do
10:        x ← BasicPartition(k*), Ca ← ca^x
11:    return Ca
12: function RoutingInCa(Ca,τ)
13:    foreach ca* ∈ Ca do
14:        if τ ∈ R then
15:            ‡ ← row, th ← RandomInt[0 ∼ (α^{ca*} − 1)]
16:        else
17:            ‡ ← column, th ← RandomInt[0 ∼ (β^{ca*} − 1)]
18:        RD ← (ca*, ‡, th)
19:    return RD
```

Step-1: Routing tuples in $\widetilde{C}$. This step is responsible for routing tuples into ca(s). To ensure that one tuple could be sent to

various destinations, here using a content-sensitive manner, which according to a basic mapping function $F(k)$. To this end, Alg.3 first extracts the join key from each incoming tuple in line 1. Then it figures out which key(s) the tuple should pair in line 8. According to the pair key, Alg.3 generates the destination(s) for the incoming tuple, based on a basic routing function (e.g., consistent hash) in lines 9∼11.

Step-2: Routing tuples in *ca*. This step is responsible for determining which task(s) in ca is(are) destination(s). Alg.3 first identifies which stream the inputting tuple comes from in lines 14 $\sim$ 17. Accordingly, all task(s) in the row or column are selected as destination(s) in line 18. In Alg.3, the destination is denoted by $(ca, \ddagger, th)$, means the input tuple should be send to all tasks located at $\ddagger^{th}(\ddagger \in (row, column), th \in N)$ in ca.

In example of Fig.2(d), for the tuple from stream S and with join key 18, Alg.3 first determines the first *ca*(ca_1) is its direction, which according to the join predicate expression($|R.k - S.k| \leq 2$) and *storage strategy*(i.e.,$\{k_1 \sim k_{20}\} \rightarrowtail ca_1$, $\{k_{21} \sim k_{40}\} \rightarrowtail ca_2$). Since ca_1 has two columns, Alg.3 randomly selects the 0^{th} column as its destination line (i.e., destination is $(ca_1, column, 0)$ in Alg.3).

4.2 Scheme Changing

There are two criteria which should be guaranteed during the process of ca's scheme changing. The first one is to ensure the correctness of process and the second one is to lower migration cost during the process of scheme changing as far as possible. To simplify our description, some additional notations used in the following sections are summarized in Tab.4.

TABLE 4
Table of Additional Notations

Notations	Description
m_{ij}	task instance corresponding to *TaskA*$_{ij}$
h_{ij}^{xR}/h_{ij}^{xS}	the sub-stream R/S that has been stored in m_{ij}^{x}, represented as a range $[b, e]$
M_o/M_n	old task matrix and new task matrix for ca
k/l	k-th row and l-th column in new scheme
h_{ij}^{R}/h_{ij}^{S}	data has been stored in m_{ij} in ca
s_{kl}^{R}/s_{kl}^{S}	data for m_{kl} in new scheme M_n in ca, represented as a range $[b, e]$ on stream ca^r/ca^s
tpi	the mapping of tasks between old and now scheme
mp	the migration plan
TP/MP	the set of tpi/mp

We now use m and M to denote task instance and task matrix respectively, and m_{ij} corresponding to calculation area *TaskA*$_{ij}$ in ca. Furthermore, we use m_{ij} and m_{kl} to represent the task instance in M_o and M_n respectively, where i(k) and j(l) are the row number and column number of M_o (M_n). We define the correlation coefficient λ_{kl}^{ij} to reflect the volume of data overlap between m_{ij} and m_{kl}, which can be calculated as $\lambda_{kl}^{ij} = |h_{ij}^{R} \cap s_{kl}^{R}| + |h_{ij}^{S} \cap s_{kl}^{S}|$.

Alg.4 shows the task mapping method, which aims to minimize the states migration when using the new scheme M_n. Specifically, Alg.4 first enumerates all the possible mappings in form of $tpi =< m_{ij}, m_{kl}, \lambda_{kl}^{ij} >$ as shown in lines $(1 \sim 4)$; then it selects the optimal mappings which have larger λ_{kl}^{ij} among all mapping sets in lines $(6 \sim 10)$. Task mapping TP generated by Alg.4 gathers the largest cumulative values of λ_{kl}^{ij}. Because each m_{ij} or m_{kl} appears in TP only once at most, then the correlation

algorithm 4 Task Mapping TP Generation

input: Old task matrix scheme M_o, New task matrix scheme M_n
output: Task mapping set TP

1: **foreach** m_{ij} in Old scheme M_o **do**
2: **foreach** m_{kl} in New scheme M_n **do**
3: $\lambda_{kl}^{ij} \leftarrow (h_{ij}^{R} \cap s_{kl}^{R}) \cdot |ca^r| + (h_{ij}^{S} \cap s_{kl}^{S}) \cdot |ca^s|$
4: $TPI \leftarrow < m_{ij}, m_{kl}, \lambda_{kl}^{ij} > / * TPI$ is a temp set $*/$
5: Initialize TP = $Null$
6: **foreach** tpi in TPI in descending order based on λ_{kl}^{ij} **do**
7: **if** m_{ij} or m_{kl} in tpi has been appeared in TP **then**
8: Continue
9: $< m_{ij}, m_{kl} > \rightarrow TP$
10: **return** TP

coefficient λ_{kl}^{ij} is independent of others. Our algorithm always selects the mapping pair with biggest λ_{kl}^{ij}, and then it will generate the maximum cumulative value of λ_{kl}^{ij}. In other words, Alg.4 finally produces the minimal migration cost during the process of scheme changing.

Considering scheme generation in Sec.3 and task mapping generation discussed above, we design an advanced scheme mapping for reducing migration cost. In order to make it easy for explanation, we take a one dimension scheme division as an example and treat its total volume as unit "1", shown in Fig.8, where $h_i(s_i)$ represents the range processed by task i.

A naive method is to divide the stream into even ranges and assign each range to one task as shown in Fig.8(a). In old scheme, there are 4 tasks (No.0∼3) and each task manages 25% percentage of the whole stream, represented as h_i ($i \leq 3$). When it is scaled out to 5 tasks, 5 ranges are generated evenly and each range s_j ($j \leq 4$) maps to one task. Based on Alg.4, we can figure out the best mapping from $\{h_i\}$ to $\{s_j\}$ shown in Fig.8(a), represented by the dotted line. The shadowed range in old scheme will be migrated among tasks, and the newly added task will be assigned the data in range $[2/5, 3/5]$ labeled as s_4. The total volume of data migration is $\frac{1}{20} + \frac{1}{5} + \frac{1}{20} = \frac{3}{10}$.

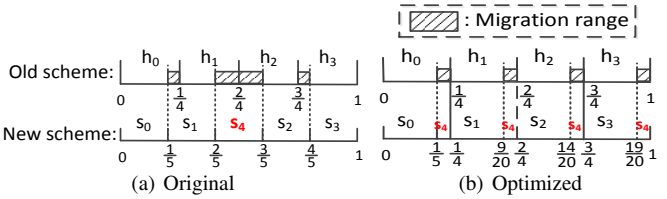


Fig. 8. Advanced Scheme Generation and Task Mapping.

Given that join correctness is independent of the order of tuples as long as we can guarantee that the same row (column) has the same states as described in Sec.2. In this context, we can conduct an optimization based on start-point-alignment method for task mapping as follows. We first align the same range start points for $\{s_j\}$ in new scheme to those from $\{h_i\}$ in old scheme, if any. In Fig.8(b), the old scheme is scaled out to 5 tasks. We align the same start range points between 4 tasks such as $s_0 \sim s_3$ from the new scheme and $h_0 \sim h_3$ from the old scheme. Each h_i cuts down the shadowed $1/20$(=$1/4 - 1/5$) of the range and moves them to the newly inserted task that is s_4. In such a case, the total migration volume is $\frac{1}{20} + \frac{1}{20} + \frac{1}{20} + \frac{1}{20} = \frac{1}{5}$, which is less than the migration cost generated by Alg.4. For scaling down, it is almost

the same. We align the start points of new tasks with some of the old ones, and split ranges managed by the removing nodes (to be deleted) to the tasks kept in the new scheme.

4.3 Migration Plan Generation

Figuring out the part of data which should be migrated among tasks is a necessary information in the process of scheme changing. We use $\diamondsuit$ to represent the join stream that $\diamondsuit \in \{ca^r, ca^s\}$ and $n_{kl}^{\diamondsuit}$ means the data that are lacked in m_{kl}, and then the range of data $n_{kl}^{\diamondsuit}$ is:

$$n_{kl}^{\diamondsuit} = s_{kl}^{\diamondsuit} - (s_{kl}^{\diamondsuit} \cap h_{kl}^{\diamondsuit}) \tag{15}$$

Then we define the migration plan as $mp =< m_{ij}, m_{kl}, \nu^{\diamondsuit} >$ that can be read as: for stream $\diamondsuit$, the data $\nu^{\diamondsuit}$ should be copied for m_{ij} in old scheme M_o to m_{kl} in new scheme M_n, where $\nu^{\diamondsuit}$ is:

$$\nu^{\diamondsuit} = h_{ij}^{\diamondsuit} \cap n_{kl}^{\diamondsuit} \tag{16}$$

For each m_{kl}, we use $d_{kl}^{\diamondsuit}$ to represent the data which are no longer needed and should be moved out, and $d_{kl}^{\diamondsuit}$ can be calculated as:

$$d_{kl}^{\diamondsuit} = h_{kl}^{\diamondsuit} - (s_{kl}^{\diamondsuit} \cap h_{kl}^{\diamondsuit}) \tag{17}$$

In this case, the migration plan can be represented as $mp =< \odot, m_{kl}, d_{kl}^{\diamondsuit} >$. The migration plan with mark $\odot$ can be read as: for stream $\diamondsuit$, delete data $d_{kl}^{\diamondsuit}$ from m_{kl}.

The generation of migration plan is described in Alg.5 and can be divided into four steps as follows: **Step-1: Discarding states that are no longer needed in M_n.** As shown in line 1 of Alg.5, this step is used to clean useless states generated by old scheme M_o. Specifically, some states will be discarded when scheme change happens. **Step-2: Filling stream data for task instances in the new scheme.** We can get the whole data set in stream R or S by combining the data from the first row or the first column in M_o. As in line ($2 \sim 11$), we can fill each task instance with data in M_n using tuples which are stored in the first row or column in M_o. **Step-3: Handling the last row or column with irregular scheme.** This step plays the role that making scheme full use of resources. It redistributes the subset of data of primary stream located in task instances $P_\delta \sim P_\gamma$ in lines ($12 \sim 18$). **Step-4: Deleting useless tuples.** It deletes data with $\odot$ mark in mp under the new scheme M_n in lines ($19 \sim 21$).

In above steps, **step-1** will be triggered when the old scheme M_o is an irregular matrix and **step-3** will be triggered when the new scheme M_n is irregular. To make it comprehensible, we first walk through an example with regular scheme transformation (just use **step-2** and **step-4** in Alg.5) and then discuss the details of the irregular scheme generation.

A scheme changes from 2×2 to 2×3 as depicted in Fig.9 and we also treat the total volume of ca^r/ca^s as unit "1". In old scheme M_o, each task manages half size of stream data from ca^r and ca^s shown in Fig.9(a): each row manages half size of ca^r with $h_{00}^R = h_{01}^R = [0, \frac{1}{2}]$ and $h_{10}^R = h_{11}^R = [\frac{1}{2}, 1]$; each column manages half size of ca^s with $h_{00}^S = h_{10}^S = [0, \frac{1}{2}]$ and $h_{10}^S = h_{11}^S = [\frac{1}{2}, 1]$. When the workload of stream ca^s increases, system may scale out by adding one more column with two tasks using a 2×3 scheme as shown in Fig.9(b). In this case, data partitions to ca^r are unchanged where tasks in the first row still manage half size of volume ($s_{0j}^R = [0, \frac{1}{2}], j \in \{0, 1, 2\}$) and tasks in the second row manage the other half ($s_{1j}^R = [\frac{1}{2}, 1], j \in \{0, 1, 2\}$). Stream ca^s should be split into three partitions for three columns, each of which manages $\frac{1}{3}$ range of data, that is $s_{i0}^S = [0, \frac{1}{3}]$, $s_{i1}^S = [\frac{1}{3}, \frac{2}{3}]$ and $s_{i2}^S = [\frac{2}{3}, 1]$, with $i \in \{0, 1\}$.

algorithm 5 Migration Plan Generation

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

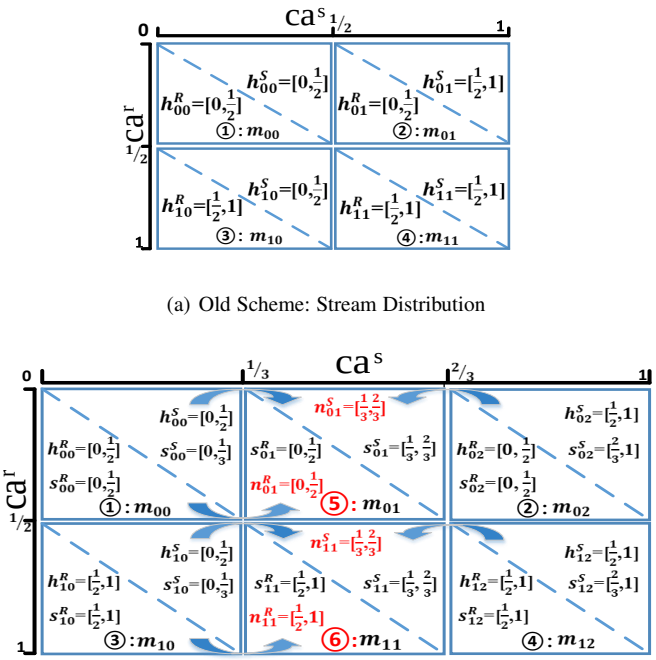
1: Discard states with mark $\boxdot$ state for each task in new scheme
2: **foreach** row i with column 0 in old scheme M_o **do**
3: **foreach** task m_{kl} in new scheme M_n **do**
4: **if** $h_{i0}^R \cap n_{kl}^R \neq \varnothing$ **then** $/*According\ to\ Equ.15\&16*/$
5: $< m_{i0}, m_{kl}, h_{i0}^R \cap n_{kl}^R > \rightarrow MP$
6: Update n_{kl}^R
7: **foreach** column j with row 0 in old scheme M_o **do**
8: **foreach** task m_{kl} in new scheme M_n **do**
9: **if** $h_{0j}^S \cap n_{kl}^S \neq \varnothing$ **then**
10: $< m_{0j}, m_{kl}, h_{0j}^S \cap n_{kl}^S > \rightarrow MP$
11: Update n_{kl}^S
12: **foreach** task x located between P_δ and P_γ **do**
13: **if** $P = R$ **then** $/*\ y \in [0, P_\delta - 1]\ */$
14: $< \boxdot, m_{xD_\gamma^f}, m_{yD_\gamma^f}, s_{xD_\gamma^f}^R > \rightarrow MP$
15: Set task in $m_{xD_\gamma^f}$ in M_n as inactive
16: **else if** $P = S$ **then**
17: $< \boxdot, m_{D_\gamma^f x}, m_{D_\gamma^f y}, s_{D_\gamma^f x}^S > \rightarrow MP$
18: Set task in $m_{D_\gamma^f x}$ in M_n as inactive
19: **foreach** task m_{kl} in new scheme M_n **do**
20: $< \odot, m_{kl}, d_{kl}^R > \rightarrow MP$ $/*\ According\ to\ Equ.17\ */$
21: $< \odot, m_{kl}, d_{kl}^S > \rightarrow MP$
22: **return** MP

According to the discussion in Sec.4.2, if we have the optimal partitioning scheme with minimal migration cost, TP is $\{< m_{00}^o, m_{00}^n >, < m_{01}^o, m_{02}^n >, < m_{10}^o, m_{10}^n >, < m_{11}^o, m_{12}^n >\}$. In Fig.9(b), we pair the relevant tasks between M_o and M_n by assigning the same numbers for tasks. The tasks tagged with red numbers ⑤ and ⑥ in m_{01} and m_{11} are not paired. m_{01} is supposed to manage data $n_{01}^R = [0, \frac{1}{2}]$ and $n_{01}^S = [\frac{1}{3}, \frac{2}{3}]$; m_{11} is supposed to manage data $n_{11}^R = [\frac{1}{2}, 1]$ and $n_{11}^S = [\frac{1}{3}, \frac{2}{3}]$. According to Alg.5, in m_{01}, s_{01}^R is generated by duplicating ca^r data from m_{00}. Since ca^s is reallocated by splitting into 3 parts for the insertion of a new column, it first generates the complete data set by combining h_{00}^S and h_{01}^S in scheme M_o, and then s_{01}^S is generated by replicating ca^s data from m_{00}^S with h_{00}^S by range $[\frac{1}{3}, \frac{1}{2}]$ and from m_{01}^S with h_{01}^S by range $[\frac{1}{2}, \frac{2}{3}]$. Then the data for s_{01}^S is deleted from these task instances. m_{11} in M_n will be assigned data in the same way as m_{01} does.

5 Evaluation

5.1 Experimental Setup

Environment: We implement the approaches and conduct all the experiments on top of *Apache Storm* [1]. The *Storm* system is deployed on a cluster of 21 HP blade instances, each of which runs CentOS 6.5 operating system and is equipped with two Intel Xeon processors (E5335 at 2.00GHz) with four cores and 32GB RAM. Overall, there are 300 virtual machines available exclusively for our experiments, each with dedicated memory resources of 2GB.
Data sets: We test the proposed algorithms using two types of data sets. The first TPC-H data sets is generated by the $dbgen$ tool



(a) Old Scheme: Stream Distribution

(b) New Scheme: Stream Distribution

Fig. 9. Stream Distribution Example

shipped with TPC-H benchmark [2]. Before feeding the data to the stream system, we pre-generate and pre-process all the input data sets. We adjust the data sets with different degrees of skew on the join attributes under $Zipf$ distribution by choosing a value for skew parameter z. By default, we set $z = 1$. The second data set is $10GB$ social data[1] (coming from Weibo which is the biggest Chinese social media data) which consists of 20,000,000 real feed tuples. We run self-join on social data to find the correlation degree among tuples.

Queries: We experiment on four join queries, namely one equi-join from the TPC-H benchmark and three synthetic band-joins. The equi-join, E_{Q_5}, represents the most expensive operation in query Q_5 from the benchmark. B_{NCI}, B_{MR} and $Social\ data\ query$ are all band-joins, which are different in the range of join. Specifically, B_{NCI} is the θ-join query that corresponds to a small range, while B_{MR} is the theta-join query that represents a big range. B_{MR} and $Social\ data\ query$ are full band-joins which require each tuple from one stream meets all the tuples from the other stream. The E_{Q_5} and B_{NCI} are used in [9], [22].

Baseline Approaches: In A-DSP, we compress the tuple matrix in our experiments using a matrix compression method proposed in [31], which generates the sample matrix having a much smaller size than the original matrix and guarantees the sample tuple matrix being high fidelity. For the purpose of comparison, we evaluate four different distributed stream join algorithms as follows.

$Square$ [12] figures out the number of tasks through a simple and easy way defined Equ.9.

$Dynamic$ [9] takes join matrix as its processing model and assumes that the number of tasks in a matrix must be a power of 2. Since the model needs to maintain its matrix structure, if the workload of one stream increases twofold, $Dynamic$ will double cells along the side corresponding to this stream. Meanwhile, cells along the other side will get reduced by half. In addition, during system scaling out, $Dynamic$ splits the states of every node into 4 nodes if storage of any task exceeds half of specified memory capacity.

Readj [14] is designed to minimize the load of restoring the keys based on the hash function, implemented by key rerouting over the keys with maximal workload. The migration plan of keys for load balance is generated by pairing tasks and keys. For each task-key pair, their algorithm considers all possible swaps to find the best move alleviating the workload imbalance. In *Readj*, σ is a configurable parameter, deciding which keys should take part in action of swap and move. Given a smaller σ, *Readj* tends to track more candidate keys and thus finds better migration plans. In order to make fair comparison, in each of the experiment, we run *Readj* with different σ's and only report the best result from all attempts.

Bi_6 and Bi [22] handle θ-join using a complete bipartite graph. On one side of the bipartite graph, a data stream R is decomposed into substreams by a key-based hash function, each partition of which is stored and maintained by a computation node. In the following part of this paper, we use Bi to represent there is only one subgroup for each side of join-biclique and Bi_6 to denote there are six subgroups for each side. For routing the incoming tuple, they use a two-tier architecture of the router, consisting of shuffler and dispatcher. The shuffler, implemented as a spout, ingests the input streams and forwards every incoming tuple to the dispatcher. The dispatcher, implemented as a bolt, forwards every received tuple from the shuffler to the corresponding units in the joiner to store and join.

Performance Metrics: We evaluate resource utilization and system performance through the following metrics:

$Tasknumber$ is the total number of tasks consumed by system and each task is assigned with a specified quota of memory space V.

$ExecutionTime$ is consumption time taken to deal with a certain amount of data.

$Thoughtput$ is the average number of tuples that processed by system per second (time unit).

Average join request account is the average number of tuples which should be joined within each task.

$Migration\ volume$ is the total number of tuples which should be moved to other tasks during scheme changing.

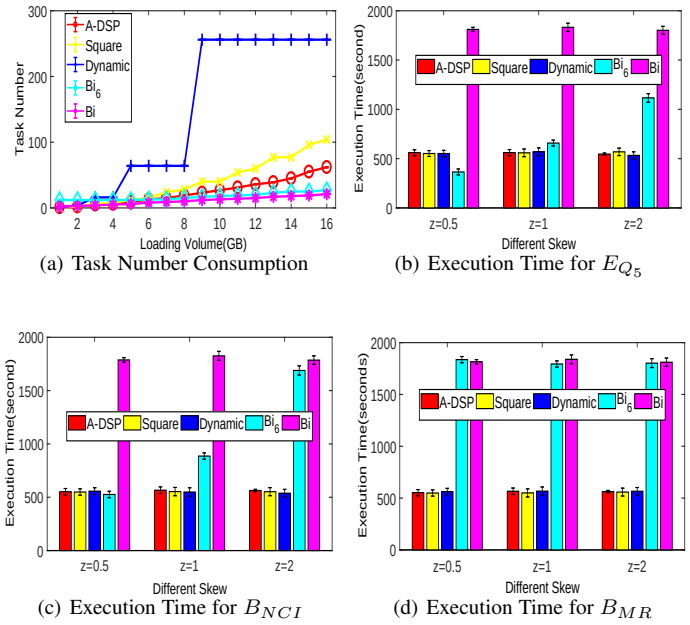
$ResurceCost\ (RC)$ is the cumulative usage volume of a certain resource which can be expressed as $RC = \sum_{r\in\mathbb{R}} r \cdot t_r$, where r is the measurement of resource consumption, which may be memory, network or CPU, $\mathbb{R}$ is the set of r and t_r is the duration time for those resource r.

5.2 Full History Stream Joins

Fig.10 demonstrates the trend on task consumption and execution time during loading all $16GB$ data into Storm system. The maximum input rate can be set to consume all the computing power of each task, using enough parallelism for spout in Storm.

During data loading as in Fig.10(a), our algorithm *A-DSP* has stable performance while $Dynamic$ meets sharp increase in task number. This is because $Dynamic$ has a strict requirement that the number of tasks must be a power of two, and then, $Dynamic$ must quadruple its tasks and may waste resources when system scales out. Contrarily, our algorithm *A-DSP* generates the processing scheme based on current workload. $Square$ limits the

1. $http://open.weibo.com/wiki/2/statuses/user\_timeline$



(a) Task Number Consumption (b) Execution Time for E_{Q_5}

(c) Execution Time for B_{NCI} (d) Execution Time for B_{MR}

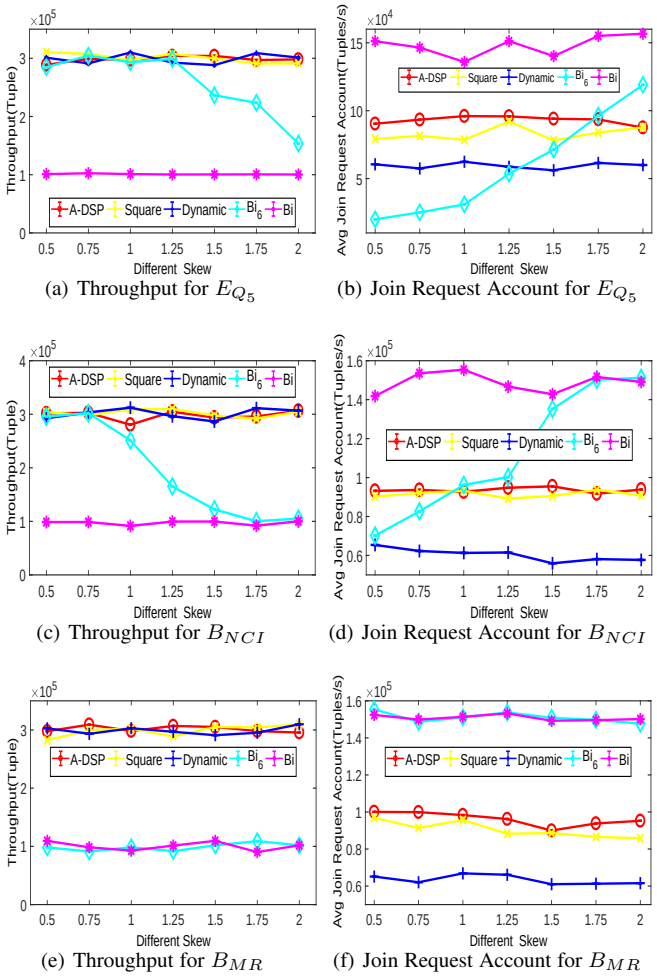
Fig. 10. Task Consumption and Execution Time of Full History Join

TaskA must be square shape and the divides *CA* directly makes it use more computing resources. Since Bi is designed for memory optimization, it is obvious that Bi uses the minimal number of tasks. Specifically, Bi_6 sets up 6 subgroups for each stream and every subgroup contains one task initially. As data flows in, tasks inside subgroups will scale out dynamically. Figs.10(b), 10(c) and 10(d) depict processing efficiency under different data skewness executing E_{Q_5}, B_{NCI} and B_{MR}. *A-DSP* and matrix-join methods including $Square$ and $Dynamic$, can process join stably since data are distributed to tasks randomly. Although Bi takes up fewer tasks, its efficiency is almost three times lower than others due to its lack of computing resources. Specifically, when $z = 2$, severer skewness may cause tuple broadcast among groups. B_{MR} is a wide range of band-join, the advantages of subgrouping in Bi_6 completely disappear as shown in Fig.10(d). Due to the random tuples distribution on join-matrix models, the execution time of $Square$ and $Dynamic$ is immune to data skewness. Furthermore, our *A-DSP* uses a hybrid routing strategy which also can be benefited from the random distribution manner.

In Fig.10, the execution time of $Dynamic$ is similar with our methods, but it is at the expense of more tasks. Since Bi and Bi_6 take minimizing memory usage as the optimization goal, it may decrease processing efficiency when it is lack of CPU resources as shown in Fig.10(d). From Fig.10 we can conclude that our algorithms are more scalable and efficient.

5.3 Window-based Stream Joins

For this group of experiments, we have 64GB data with window size 180 seconds and slide size 60 seconds. We set stream flow-in rate at about $3 \cdot 10^5$ tuples per second to make full use of CPU resources. To facilitate the description, we define the tuple that is distributed to the storage side for join as join request. To further validate the effects of different band-joins and data skewness on system performance, we run window-based joins to testify throughput under different models.



(a) Throughput for E_{Q_5} (b) Join Request Account for E_{Q_5}

(c) Throughput for B_{NCI} (d) Join Request Account for B_{NCI}

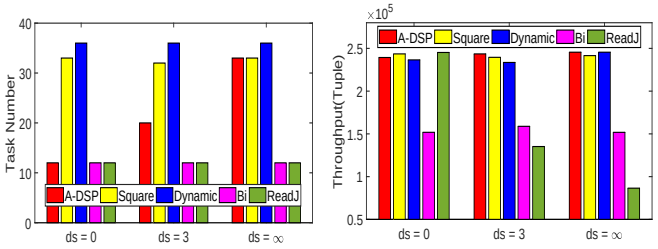
(e) Throughput for B_{MR} (f) Join Request Account for B_{MR}

Fig. 11. Throughput and Tuple Account of Window-based Join for Different Queries

Fig.11 shows system throughput and average number of join requests received by task in different processing schemes for queries E_{Q_5}, B_{NCI} and B_{MR} under different data skewness. Fig.11(a), 11(c) and 11(e) compare throughput of each algorithm. Throughput of matrix-based algorithms is obviously higher than those of bipartite-graph-based ones which result from data broadcasting among groups and are lacking of CPU resources. While executing E_{Q_5} under low-skew data, Bi_6 can route join requests more selectively to subgroups, guaranteeing the execution efficiency within each task. With severer skewness, however, the amount of data broadcasting among subgroups lows down system performance greatly. While executing B_{NCI} and B_{MR}, the band-join operations incur more broadcasts among subgroups and aggravate the CPU loads compared to E_{Q_5}. It is obvious in Fig.11(f), where the large range band-join requests make Bi_6 lose advantages of selection operation on grouping data. Because Bi has only one subgroup for each stream and it broadcasts the same amount of data for these three kinds of queries as shown in Fig.11(b), 11(d) and 11(f). Furthermore, the reason that Bi_6 is unaffected by B_{MR} is that B_{MR} is a big range band-join. Matrix-based algorithms share the CPU load equally among tasks, but suffer from consuming more tasks. In this group of experiments, the task usages of *A-DSP*,$Square$,$Dynamic$,Bi_6 and Bi are 13, 16, 64, 12 and 6, respectively.

5.4 Adaptivity

In order to validate the adaptivity of our processing scheme, we simulate two scenarios as follows: **1)** we do selectivity transmission for join operation by altering the calculation range of join predicate; **2)** we keep on varying stream volume ratio by $\varrho = \frac{|R|}{|S|}$ under the specified total stream volume of 40 GB. Both those experiments are run based on window and all settings are similar to Sec.5.3. For the threshold controlling scales out or down, $Dynamic$ uses its own setting in [9] and other algorithms change scheme when memory load is higher than eighty or lower than fifty percent of configured for each task.



(a) Task Consumption (b) Throughput Performance

Fig. 12. Performance under Various Selectivity

5.4.1 Adaptivity on Selectivity Transmission

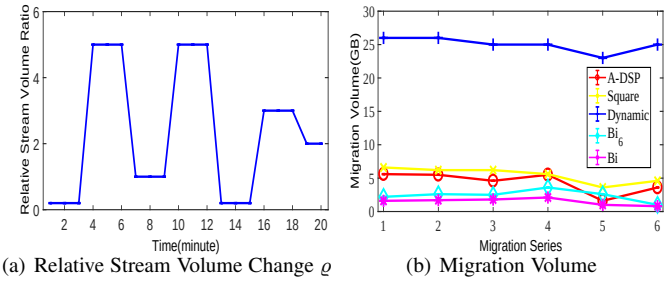
To better understand our *A-DSP* can adaptively change the dimensions of its processing architecture to satisfy various operation requirements, we disrupt the time series of $Social$ data and run the *Social data query* with various selectivity. The different selectivity scenarios are generated by changing the correlative days, i.e., ds, where $ds = |S1.date - S2.date|$. Specifically, we use $ds = 0$, $ds = 3$ and $ds = \infty$ to denote the equivalent, range and full join operation, respectively. Fig.12(a) shows the task consumption of different approaches under above settings. Regardless of the join types, $Square$ and $Dynamic$ always consume the most tasks. This is because they are both based on the fixed matrix architecture. Conversely, both Bi and $ReadJ$ consume the least tasks, for they do not need to store copies of the incoming stream. However, their complicated routing strategies and broadcast action inevitably be an obstacle to the throughput performance, as shown in Fig.12(b). Fig.12(a) shows that our *A-DSP* model can consume resource on-demand, which based on the specific join operation. Furthermore, Fig.12(b) reflects *A-DSP* can enable the system to provide stable high throughput although it consumes less tasks.

5.4.2 Adaptivity on Relative Stream Volume Change

In Fig.13(a), we adjust ϱ every three minutes. Accordingly, the migration volumes of each approach are displayed in Fig.13(b). Both Bi and Bi_6 incur less migration cost because they contain the lesser redundant data. However, this is at the cost of a higher CPU consumption, which inevitably decreases the throughput. $Dynamic$ suffers from high migration cost because this model requires its task number must be a power of two. Our *A-DSP* algorithm explores the most appropriate processing scheme which takes relatively small migration cost.

5.5 Throughput and Latency

Our synthetic workload generator creates snapshots of tuples for discrete time intervals from an integer key domain K. The tuples



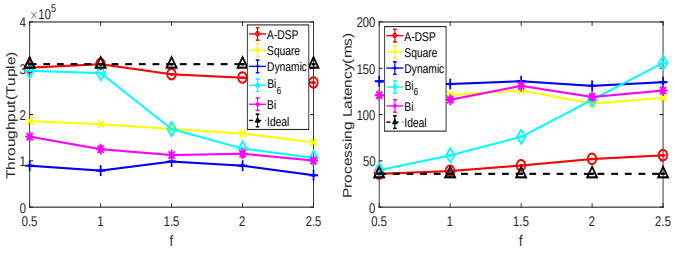
(a) Relative Stream Volume Change ϱ (b) Migration Volume

Fig. 13. Performance with Relative Stream Volume Change

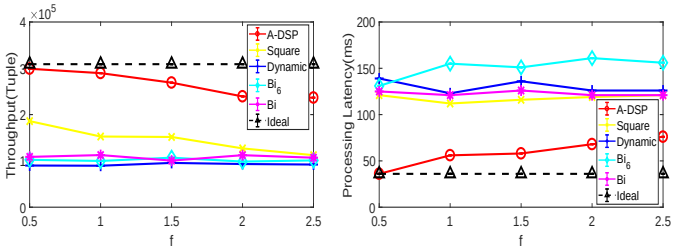
follow Zipf distributions controlled by skewness parameter z, by using the popular generation tool available in Apache project. We use parameter f to control the rate of distribution fluctuation across time intervals. At the beginning of a new interval, our generator keeps swapping frequencies between keys from different task instances until the change on workload is significant enough. i.e., $\frac{L_i(d) - L_{i-1}(d)}{\bar{L}} \geq f$. In the above expression, $L_i(d)$ and $L_{i-1}(d)$ mean the total workload of task instance d in the ith and (i-1)th time interval (window), respectively. $\bar{L}_i$ means the average load of all tasks in the ith time interval. Then, f can be used to denote the workload rangeability of task d between now and the last time interval.

Fig. 14 shows the throughput and latency with varying distribution change frequency running on E_{Q_5} (Fig. 14(a) and Fig. 14(b)) and B_{MR} (Fig. 14(c) and Fig. 14(d)), respectively. In Fig. 14, we draw the theoretical limit of the performance with the line labeled as *Ideal*, which processing corresponding query using *Square* in the condition of adequate resources are available. Obviously, *Ideal* always generates a better throughput and lower processing latency than any others, but cannot be used in real-world application, for its resource waste. When varying the distribution change frequency f, both the throughput and latency of Bi_6 change dramatically in Fig. 14(a) and Fig. 14(b). In particular, Bi_6 works well only in the case with less distribution variance (smaller f) in Fig. 14(a) and Fig. 14(b). In the meantime, both Bi and $Dynamic$ alway have a low throughput and high processing latency in Fig. 14, this is because the broadcast action in Bi and the limit number of task using in $Dynamic$. On the other hand, our *A-DSP* algorithm always performs well, with performance very close to the optimal bound set by *Ideal*.

To expose the cost of our approach in finding the right processing scheme, Fig.15 shows the adjustment times and migration latency of each approach under varying distribution change frequency. We run this group experiments on B_{MR} and the relative stream volume change parameter is configured as Fig.13(a). Fig.15(a) shows *A-DSP* incurs more scheme adjustment during the data distribution change. However, its migration latency is still acceptable to the system as shown in Fig.15(b). This is because we have designed a lightweight computation model (Alg.4 and Alg.5) to support rapid migration plan generation, which incurs minimal data transmission overhead and processing latency. *Square* uses the same migration plan generation strategies, and it has a similar performance. *Dynamic* incurs a smaller adjustment time since it adjusts its processing scheme only after the relative stream volume changed. Due to the face that *Dynamic* needs to maintain a large amount of copy data, its migration latency is the one biggest as shown in Fig.15(b). Bi and Bi_6 using the fewest processing tasks

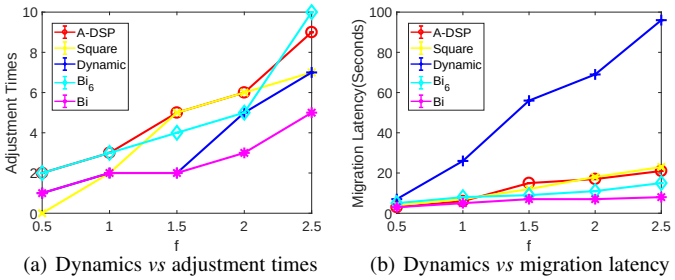


(a) Stream dynamics *vs* throughput (b) Stream dynamics *vs* latency



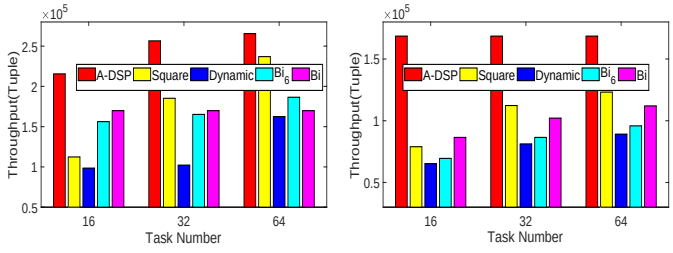
(c) Stream dynamics *vs* throughput (d) Stream dynamics *vs* latency

Fig. 14. Throughput and Latency with Varying Distribution Change Frequency, where (a) and (b) Run on a E_{Q_5}, and (c) and (d) on B_{MR}.



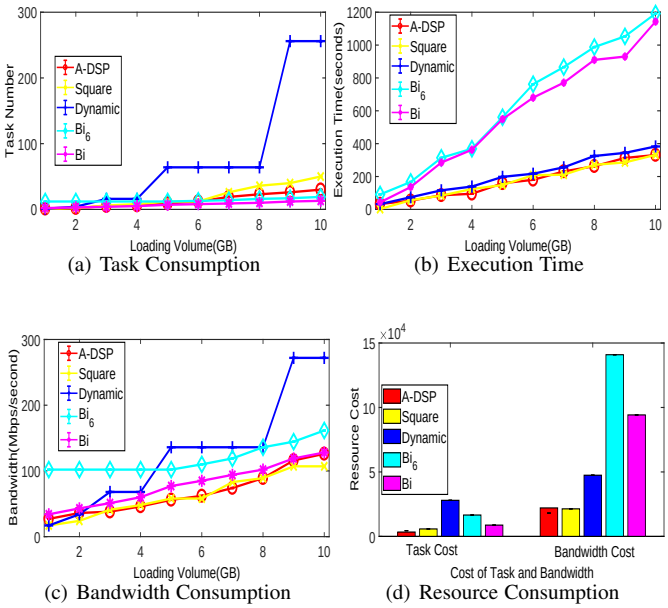
(a) Dynamics *vs* adjustment times (b) Dynamics *vs* migration latency

Fig. 15. Adjustment Times and Migration Latency with Varying Distribution Change Frequency.



(a) Social data (b) Stock data

Fig. 16. Performance under Limit Tasks.



(a) Task Consumption (b) Execution Time

(c) Bandwidth Consumption (d) Resource Consumption

Fig. 17. Performance on Real Data

make them generate a smaller migration volume and latency.

5.6 Performance on Real Data

5.6.1 Throughput under limited resources

To better understand the performance of the approaches in action, we present the dynamics of the throughput on two real workloads, especially when the system is limited the using number of tasks. On Social data, we implement a band-join topology on Storm, with distributing tuples to tasks for store and join on keywords. On Stock data, a self-join on the data over sliding window is implemented, which maintains the recent tuples based on the size of the window over intervals. The results are available in Fig. 16, showing that our method *A-DSP* always produce a higher throughput. Though *Dynamic* is content-insensitive, it produces a lower throughput than *A-DSP*, for the available resources always cannot meet its demand. On both *Social data* and *Stock data* with different available tasks, Bi and Bi_6 have little change in the performance of throughput, this is because the biggest obstacle to their performance is the huge broadcast tuples for this queries. The throughput of *Dynamic* only be one-third of *A-DSP* when the limitation task number is 32, which leads to huge resource waste and is definitely undesirable to cloud-based streaming processing applications.

5.6.2 Resource utilization efficiency

To prove the usability of our algorithm, we do band-join $Social\ data\ query$ on $10GB$ Weibo datasets with the settings introduced in Sec.5.2. We load the $10GB$ dataset continuously, and measure resource consumption by different algorithms. In Fig.17(a), bipartite-graph-based models Bi and Bi_6 use less tasks for it is mainly designed for memory optimization as explained in Fig.10. Our method *A-DSP* provides a flexible matrix scheme which applies for new tasks according to its real load while $Dynamic$ scales out in a generous way. Our algorithm is efficient in completing all the work while the bipartite-graph-based models are the slowest for its lack of CPU resources, shown in Fig.17(b).

Fig.17(c) shows the bandwidth usage for each algorithm when loading data into Storm system. Though Bi uses less tasks than *A-DSP* as shown in Fig.17(a), its bandwidth usage is more than ours for it broadcasts the same amount of incoming tuples to all tasks in the other side of bipartite-graph. $Dynamic$ costs the most bandwidth. Fig.17(d) provides the overall resource cost (Task and Bandwidth) by time duration, as computed by $RC = \sum_{r \in \mathbb{R}} r \cdot t_r$ and we consider cost by continuing occupation to resources. It confirms that *A-DSP* can deal with the band-join more economically.

6 Related Work

In the early 21st century, considerable researches have been unfolded on designing efficient stream join operators in a distributed environment with a cluster of machines. As mentioned in Sec.1, there are two categories model for join operation in distributed stream processing systems, namely *1-DSP* and *2-DSP*.

1-DSP. Photon [3] is designed by Google to join data streams such as web search queries and user clicks on advertisements. It utilizes a central coordinator to implement fault-tolerant and scalable joining through key-value matching, but cannot handle θ-join. D-Streams [36] adopts mini-batch on continuous data streams in a blocking way on Spark. Though it upholds θ-join well, under the constraint of window size, some tuples may miss each other and such batch-mode computing can only give approximate results. TimeStream [27] equipped with resilient substitution and dependency tracking mechanisms provides both MapReduce-like batch processing and non-blocking stream processing. However, it suffers from the excessive communication cost due to maintenance of distributed join state. Join-Biclique [22] based on bipartite-graph model supports full-history and window-based stream joins. Compare to the join-matrix model used in DYNAMIC [9], it reduces data backup redundancy and improves resource utilization. In the meantime, considerable efforts have been put on the elasticity feature of the distributed stream processing systems [7], [13], [15], [20], [32], dealing with when and how to efficiently scale in or out computing resources.

2-DSP. In recent years, join-matrix model for parallel and distributed joins has been restudied. Intuitively, it models a join between two data sets *R* and *S* as a matrix, where each side of which corresponds to one relation. Stamos et al. [29] proposed a symmetric fragment and replicate method to support parallel θ-joins. This method relies on replicating input tuples to ensure result completeness and extending the fragment and replicate algorithm [10], which suffers from high communication and computation cost. Okcan [26] proposed techniques in MapReduce that adopts join-matrix model and supports parallel θ-joins. He designed two partitioning scheme, that is 1-Bucket and M-Bucket. 1-Bucket scheme is content-insensitive but incurs high data replication; M-Bucket scheme is content-sensitive in that it maps a tuple to a region based on the join key. Because of the inherent nature of MapReduce, these two algorithms are offline and require that all the data statistics must be available beforehand, which are not suitable for stream join processing. In data stream processing, Elseidy et al. [9] presented a *(n,m)- mapping scheme* partitioning the matrix into $J(J = n \times m)$ equal regions and propose algorithms which adjust the scheme adaptively to data characteristics in real time. However, [9] assumes that the number of partitions *J* must be powers of two. What's more, the the matrix structure suffers from bad flexibility because when the matrix needs to scale out(down), it must add(remove) the entire row or column cells.

Due to the characteristics of data streams, e.g., infiniteness and instantaneity, conventional join algorithms using blocking operations, e.g., sorting, cannot work any more. For stream join processing, much effort has been put into designing non-blocking algorithms. Wilschut presents the symmetric hash join SHJ [33]. It is a special type of hash join and assumes that the entire relations can be kept in main memory. For each incoming tuple, SHJ progressively creates two hash tables, one for each relation, and probes them mutually to identify joining tuples. XJoin [30] and DPHJ [19] both extend SHJ by allowing parts of the hash tables to be spilled out to the disk for later processing, greatly enhancing the applicability of the algorithm. Dittrich [8] designed a non-blocking algorithm called PMJ that inherits the advantages of sorting-based algorithms and also supports inequality predicates. The organic combination of XJoin and PMJ is conductive to the realization of HMJ [24], presented by Mokbel. However, all the previous algorithms belong to centralized algorithms and rely on a central entity doing join computation, which cannot be applied in distributed computing directly.

7 Conclusion

This paper presents a new parallel stream join mechanism for θ-join in distributed stream processing engines. Inspired by *1-DPS* and *2-DPS* model, we propose a *A-DSP* model to promise processing correctness together with efficient resource usages. *A-DSP* can handle parallel stream joins in a more flexible and adaptive manner, especially to address the demands on operational cost minimization over the cloud platform. Additionally, we design algorithms with moderate complexity to generate task mapping schemes and migration plans to fulfill the objective on operational cost reduction. Empirical studies show that our proposal incurs the smallest cost when the stream join operator is run under the pay-as-you-go scheme. In the future, we will explore the possibility on more flexible and scalable mapping and migration strategies with sampled matrix in pursuit of a smaller migration cost. We will also try to design a new mechanism, to ensure the correctness of processing results when there exist(s) extremely dynamic key(s).

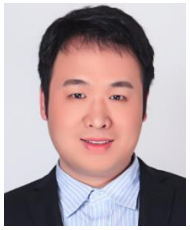
Acknowledgments

Rong Zhang and Aoying Zhou are supported by the Key Program of National Natural Science Foundation of China (No. 2018YFB1003402) and NSFC (No. 61672233). Kai Zheng is supported by NSFC (No. 61972069, 61836007, 61832017, 61532018). Junhua Fang is supported by NSFC (No.61802273), the Postdoctoral Science Foundation of China under Grant (No. 2017M621813), the Postdoctoral Science Foundation of Jiangsu Province of China under Grant (No. 2018K029C), and the Natural Science Foundation for Colleges and Universities in Jiangsu Province of China under Grant (No. 18KJB520044). Xiaofang Zhou is supported by NSFC (No.61772356). This work is also partially supported by the Major Program of Natural Science Foundation, Educational Commission of Jiangsu Province (No. 19KJA610002).

References

[1] Apache Storm. http://storm.apache.org/.
[2] The TPC-H Benchmark. http://www.tpc.org/tpch.
[3] R. Ananthanarayanan, V. Basker, S. Das, and et al. Photon: fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, pages 577–588, 2013.
[4] M. Anis Uddin Nasir, G. De Francisci Morales, and et al. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE*, pages 137–148, 2015.
[5] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Robust and skew-resistant parallel joins in shared-nothing systems. In *CIKM*, pages 1399–1408. ACM, 2014.
[6] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
[7] J. Ding, T. Z. J. Fu, R. T. B. Ma, M. Winslett, Y. Yang, Z. Zhang, and H. Chao. Optimal operator state migration for elastic data stream processing. *CoRR*, abs/1501.03619, 2015.

[8] J.-P. Dittrich, B. Seeger, and et al. Progressive merge join: A genetic and non-blocking sort-based join algorithm. In *VLDB*, pages 299–310, 2002.
[9] M. Elseidy, A. Elguindy, A.and Vitorovic, and C. Koch. Scalable and adaptive online joins. In *VLDB*, 2014.
[10] R. S. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD*, pages 169–180, 1978.
[11] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and X. Zhou. Distributed stream rebalance for stateful operator under workload variance. *TPDS*, 2018.
[12] J. Fang, R. Zhang, X. Wang, T. Z. Fu, Z. Zhang, and A. Zhou. Cost-effective stream join algorithm on cloud system. In *CIKM*, pages 1773–1782, 2016.
[13] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. Drs: Dynamic resource scheduling for real-time analytics over fast streams. In *ICDCS*, 2015.
[14] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *VLDBJ*, 23(4):517–539, 2014.
[15] L. Gu, M. Zhou, Z. Zhang, M.-C. Shan, A. Zhou, and M. Winslett. Chronos: An elastic parallel framework for stream benchmark generation and simulation. In *ICDE*, pages 101–112, 2015.
[16] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *ICDE*, pages 522–533, 2012.
[17] P. Hilton and J. Pedersen. Catalan numbers, their generalization, and their uses. *The Mathematical Intelligencer*, 13(2):64–75, 1991.
[18] R. Huebsch, M. Garofalakis, J. Hellerstein, and I. Stoica. Advanced join strategies for large-scale distributed computation. In *VLDB*, pages 1484–1495, 2014.
[19] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *SIGMOD*, pages 299–310, 1999.
[20] C. Jin, A. Zhou, J. X. Yu, J. Z. Huang, and F. Cao. Adaptive scheduling for shared window joins over data streams. *Frontiers of Computer Science in China*, 1(4):468–477, 2007.
[21] Y. Kwon, M. Balazinska, and et al. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD*, 2012.
[22] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD*, pages 811–825, 2015.
[23] B. Liu, Y. Zhu, M. Jbantova, and et al. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, pages 1338–1341, 2005.
[24] M. Mokbel, M. Lu, and W. Aref. Hash-merge join: Non-blocking join algorithm for producing fast and early join results. In *ICDE*, pages 251–263, 2004.
[25] M. A. U. Nasir, M. Serafini, and et al. When two choices are not enough: Balancing at scale in distributed stream processing. In *ICDE*, 2016.
[26] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960, 2011.
[27] Z. Qian, Y. He, C. Su, and et al. Timestream: reliable stream computation in the cloud. In *Eurosys*, pages 1–14, 2013.
[28] C.-P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1-3):181–199, 1994.
[29] J. W. Stamos and H. C. Young. A symmetric and replicate algorithm for distributed joins. *TPDS*, 4(12):1345–1354, 1993.
[30] T. Urhan and M. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, 2001.
[31] A. Vitorovic, M. ElSeidy, and C. Koch. Load balancing and skew resilience for parallel joins. In *ICDE*, 2016.
[32] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD*, pages 1279–1294, 2016.
[33] A. Wilschut and P. Apers. Dataflow query execution in a aarallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.
[34] Y. Xing, J. Hwang, U. Cetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, pages 775–786, 2006.
[35] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*, pages 1043–1052, 2008.
[36] M. Zaharia, T. Das, and et al. Discretized streams: fault-tolerant streaming computation at scale. In *SIGOPS*, pages 423–438, 2013.
[37] K. Zheng, Y. Zheng, N. J. Yuan, S. Shang, and X. Zhou. Online discovery of gathering patterns over trajectories. *TKDE*, 26(8):1974–1988, 2014.
[38] Y. Zhou, Y. Yan, B. C. Ooi, K.-L. Tan, and A. Zhou. Optimizing continuous multijoin queries over distributed streams. In *CIKM*, pages 221–222. ACM, 2005.
[39] Y. Zhou, Y. Yan, F. Yu, and A. Zhou. Pmjoin: Optimizing distributed multi-way stream joins by stream partitioning. In *DASFAA*, pages 325–341. Springer, 2006.



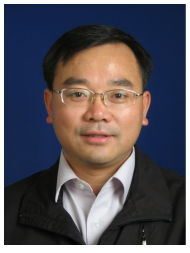
Junhua Fang is a lecturer in Advanced Data Analytics Group at School of Computer Science and Technology, Soochow University, Suzhou. Before joining Soochow University, he earned his Ph.D degree in computer science from East China Normal University, Shanghai, in 2017. He is a member of CCF. His research focuses on distributed database and parallel streaming analytics.

Rong Zhang is a member of China Computer Federation. She received her Ph.d. degree in computer science from Fudan University in 2007. She joined East China Normal University since 2011 and is currently a professor in the university. From 2007 to 2010, she worked as an expert researcher in NICT, Japan. Her current research interests include knowledge management, distributed data management and database benchmarking.

Yan Zhao received the Master degree in Geographic Information System from University of Chinese Academy of Sciences, in 2015. She is currently a PHD student in Soochow University. Her research interests include spatial database and trajectory computing.

Kai Zheng is a Professor of Computer Science with University of Electronic Science and Technology of China. He received his PhD degree in Computer Science from The University of Queensland in 2012. He has been working in the area of spatial-temporal databases, uncertain databases, social-media analysis, in memory computing and block chain technologies. He has published over 100 papers in prestigious journals and conferences in data management field such as SIGMOD, ICDE, VLDB Journal, ACM Transactions and IEEE Transactions. He is a member of IEEE.

Xiaofang Zhou received the BSc and MSc degrees in computer science from Nanjing University, China, in 1984 and 1987, respectively, and the PhD degree in computer science from the University of Queensland, Australia, in 1994. He is a professor of computer science at the University of Queensland and adjunct professor in the School of Computer Science and Technology, Soochow University, China. His research interests include spatial and multimedia databases, high performance query processing, web information systems, data mining, bioinformatics, and e-research. He is a fellow of the IEEE.



Aoying Zhou is a professor on Computer Science at East China Normal University (ECNU), where he is heading the School of Data Science & Engineering. Before joining ECNU in 2008, Aoying worked for Fudan University at the Computer Science Department for 15 years. He is the winner of the National Science Fund for Distinguished Young Scholars supported by NSFC and the professorship appointment under Changjiang Scholars Program of Ministry of Education. He is now acting as a vice-director of ACM SIGMOD China and Database Technology Committee of China Computer Federation. He is serving as a member of the editorial boards VLDB Journal, WWW Journal, and etc. His research interests include data management, in-memory cluster computing, big data benchmarking and performance optimization.