International Journal of Grid and Distributed Computing
Vol. 9, No. 2 (2016), pp.103-120
http://dx.doi.org/10.14257/ijgdc.2016.9.2.10

Multi-Way Windowed Streams θ -Joins Using Cluster

[1]Xinchun Liu, [2]Jing Li*, [3]Xiaopeng Fan and [4]Jun Chen

[1,2]*School of Computer Science and Technology,*
University of Science and Technology of China, Hefei, 230026, China
[3]*Shenzhen Institutes of Advanced Technology,*
Chinese Academy of Sciences, Shenzhen, 518052, China
[4]*Xinhua News Agency, Beijing, 100803*
[1]*liuxch@mail.ustc.edu.cn,* [2]*lj@ustc.edu.cn,*
[3]*xp.fan@siat.ac.cn,* [4]*chenjun2008@xinhua.org*

Abstract

Recent years have witnessed an increasing interesting in data stream processing, such as network monitoring, the e-business, advertising system and etc. Join is applied to explore the correlation among the tuples from multiple streams. In this paper, we present a general method named Distributed Streams Join (DSJ) to process multi-way windowed streams Θ-joins using a shared-nothing cluster. DSJ contains a distribution method named Time-Slice Distribution Method (TDM) and a join method named Transfer Join Method (TJM). Different from previous work, DSJ can (1) process multi-way Θ-joins under arbitrary predicates; (2) preserve the integrity of results and load balance while distributing tuples to different nodes for parallel joining; (3) carry out the join operation in a local optimum order according to the histograms maintained in a real-time way. We have built DSJ on our own stream processing cluster to deal with multi-way streams joins and the experiments demonstrate that our DSJ can not only guarantee the load balance among all the computing nodes but also improve the throughput effectively.

Keywords*: multi-way streams, join, distribution, cluster*

1. Introduction

Recently many applications produce data in the form of stream[1]. A data stream is a sequence of data elements made available over time. Normally, each single stream can only provide parts of information and queries on streams often require combining several of them in order to acquire comprehensive answers. As one of basic operations, join plays an important role in the process of streams.

Data streams are classified into three categories by S. Muthukrishnan[1], including the series model, the register model and the turnstile model. In this paper, our work is based on the time model, in which each element (tuple) s_i in the stream comes with an increasing timestamp, for example, the trade data of every minute in the Internet Action, the GPS trajectory data of a car every 30 seconds, and so on.

With the coming of the ear of big data, the arrival speed of data is increasing and many new distributed stream processing platforms have been developed, such as S4 [23] and Storm[4,5,22]. These platforms provide general programming frameworks and all kinds of communication API, the streams flow among the process units in the form of directed acyclic graph (DAG). Users need not to care about the underlying details. Unfortunately, they have no toolkits for some common operations, such as windows join[6,10,15,16,20], map, clustering[24] and so on.

Jing Li is the corresponding author.

ISSN: 2005-4262 IJGDC

Google has built its own real-time streams joining system named Photon, which can relate a primary user even (*e.g.* a search query) with subsequent events (*e.g.* a click on an ad)[15]. But it provides no details on multi-way joins. In many previous works, the join predicates are focus on equi-join[18][20], in which the join predicate is $(s_1.b_1 = s_2.a_2) \wedge (s_2.b_2 = s_3.a_3) \ldots \wedge (s_{n-1}.b_{n-1} = s_n.a_n)$. Tuples from different streams can be distributed to different nodes by hashing. But when the join predicate is $(s_1.b_1 =_\theta s_2.a_2) \wedge (s_2.b_2 = s_3.a_3) \ldots \wedge (s_{n-1}.b_{n-1} = s_n.a_n)$, hash is not suitable any more, here $=_\theta$ can be great (>), equal (=), little (<) or others. In this paper, we allocate tasks according to timestamp, divide the sliding windows into many sub-windows by time slices and keep them in different nodes. Different from previous CTR method, DSJ is suitable for very large windowed streams θ-joins. There are some new challenges in our algorithm:

- Integrity. Since tuples in the same sliding window may be allocated to different nodes, the algorithm should make sure that all the tuples can be scanned in the process of joining. Otherwise, the results may be incomplete.
- Load balance. All the nodes in the cluster work collaboratively. Our algorithm should guarantee that the workload is balanced among all the computing nodes.
- Efficiency. Stream processing always requires the execution in an on-the-fly style. An effective join algorithm is necessary.

In this paper, we focus on multi-way streams θ-joins in a shared-nothing cluster. Because of the characteristics of the cluster architecture, the above problems have great influence on the efficiency of the distribution and join policies. Our contributions are listed as follows:

- We present a multi-way streams joins method named Distributed Streams Join (DSJ), which consists of a distribution method named Time-Slice Distribution Method (TDM) and a join method named Transfer Join Method (TJM).
- In Time-Slice Distribution Method (TDM), we divide the sliding windows of all the input streams into many sub-windows according to timestamps and allocate them to different nodes for parallel execution.
- The histograms of all the sub-windows are maintained in each computing node. Once a new tuple or intermediate result comes, Transfer Join method (TJM) will generate a join order dynamically according to the data distributions supplied by the histograms. TJM can avoid producing some unnecessary intermediate results and improve the join efficiency.

The rest of the paper is organized as follows. In section 2, we present our join model and some related definitions. In section 3, we discuss the details of Distributed Streams Join in a shared-nothing master-worker cluster. Experiments are carried out in section 4. In section 5, we report some related work. Finally, we conclude this paper.

2. Model and Definitions

A data stream $S = \{s_1, s_2 \ldots, s_n \ldots\}$ is a large volume of data coming in an unbounded, rapid, time-varying and unpredictable way. There are three significant features of a stream: (1) unbounded size. The tuples of a stream come in a certain arrival rate continuously, it means the volume of data is increasing as time goes on; (2) uncontrollable arrival rate. The arrival rate of a data stream is related with the data source. Passive receives, users or processors could not control the arrival rate but adapt to the data stream; and (3) unpredictable order. The tuples of a stream may come from thousands of data sources, users could not identify the data source of the next tuple. To facilitate explanation, some variables used in this paper are defined in Table 1.

Table 1. Means of Some Variables

S_i	Stream i
W_i	Sliding window of stream S_i
B_i	Buffer of stream S_i
T_c	The maximum calculation time of a tuple's whole join process
T_t	The maximum transmission time of a tuple's whole join process
$\bowtie_\theta$	θ-join
$=_\theta$	Join predicate
r_i	The arrival rate of stream i
WL	Workload
t_{app}	Tuple's generation time
t_{arr}	Tuple's arrival time
T_i	The size of B_i

Due to the above features, unlike the original definition in traditional databases, the join operation for data streams only considers parts of data streams. Furthermore, it is impossible for a real-time processing system to search all the stored data to find the query answers in a few of seconds. We set a buffer for each stream to keep the latest tuples, when a tuple comes, each tuple is with a timestamp, in an explicit or implicit way. When a tuple is generated with a timestamp, it has an explicit timestamp t_{app}. Otherwise, its arrival time is considered as the timestamp t_{arr}.

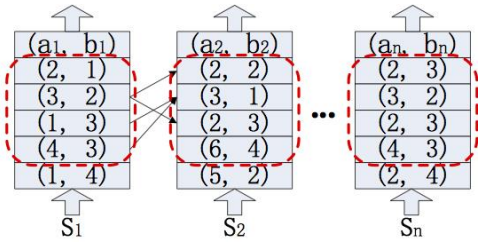


Figure 1. Multi-Way Windowed Streams Joins

For the purpose of orderly management, many studies introduced sliding windows into streams. They can be classified into two categories, including count-based windows and time-based windows. If the window size is measured by the number of tuples, such as 10000 tuples, it is considered as a count-based window. If the window size is described by time, such as 15 minutes, this kind of window is named as time-based window. Here, we assume all of the data sources have the same clock and each tuple s_i has a timestamp t_{app}, and its sliding window related with S_j is $[s_i.t_{app} - |W_j|, s_i.t_{app})$, where $|W_j|$ is the window size of S_j.

Shown in Figure 1, when a tuple s_i from stream S_i comes, the multi-way streams θ-joins is donated as $S_1[W_1] \bowtie_\theta S_2[W_2] \ldots S_{i-1}[W_{i-1}] \bowtie_\theta s_i \bowtie_\theta S_{i+1}[W_{i+1}] \ldots \bowtie_\theta S_n[W_n]$, where $W_1, W_2 \ldots, W_n$ are the sliding windows of data streams $S_1, S_2 \ldots, S_n$. The join result $(s_1, s_2 \ldots, s_n)$ consists of all the tuples $s_1 \in W_1$, $s_2 \in W_2 \ldots$, $s_{i-1} \in W_{i-1}$, s_i, $s_{i+1} \in W_{i+1} \ldots$, $s_n \in W_n$, the join predicate is $(s_1.b_1 =_\theta s_2.a_2) \wedge (s_2.b_2 =_\theta s_3.a_3) \ldots \wedge (s_{n-1}.b_{n-1} =_\theta s_n.a_n)$, where $a_2, a_3 \ldots, a_n, b_1, b_2 \ldots, b_{n-1}$ can be same or different,

$=_\theta$ can be great (>), little (<), equal (=) or others. Let $s_i.t_{app}$ and $s_i.t_{arr}$donate the tuple s_i's generation and arrival time, the join operator must scan the whole buffer to find out all the tuples generated during$[s_i.t_{app} - |W_j|, s_i.t_{app})$, then executes the join process. Because of limited memory, when $s_i.t_{arr} - s_i.t_{app} > max\{|W_1|, |W_2| \dots, |W_n|\}$, we consider the tuple is invalid, and drop it. In order to guarantee the integrity of results, we should make sure the whole sliding window W_i of each stream S_i can be kept in its own buffer B_i. The size of buffer B_i is

$$T_c + T_t + 2|W| \times r_i \leq B_i \quad (1)$$

In (1), T_c and T_t represent the maximum calculation and transmission time of a tuple's whole join process with other streams in the cluster, r_i is stream S_i's arrival rate, and $|W| = \max\{|W_1|, |W_2| \dots, |W_n|\}$.

The detailed procedure of multi-way streams θ-joins is described as follows. After receiving a tuple s_i from S_i, the join operator joins s_i with tuples in other streams following a certain order, *i.e.*, $S_1[W_1] \bowtie_\theta S_2[W_2] \dots \bowtie_\theta S_{i-1}[W_{i-1}] \bowtie_\theta s_i \bowtie_\theta S_{i+1}[W_{i+1}] \dots \bowtie_\theta S_n[W_n]$. Before $S_j[W_j]$ is joined, the join operator begins with an operation to purge all the tuples that out of date in buffer B_j first of all, and then finds out all the tuples in W_j from B_j and execute the join when satisfying the join predicate. Finally, s_i is inserted into B_i.

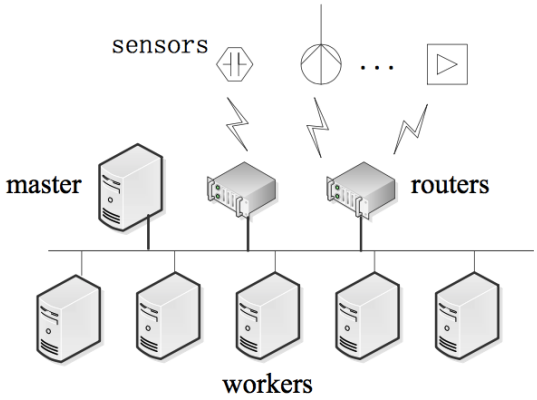


Figure 2. Cluster Architecture

In this paper, our multi-way streams joins method is executed on a master-worker architecture cluster. Shown as Figure 2, there are three kinds of nodes in the cluster:

- Master node. It is responsible for managing the whole cluster, monitoring the workload of all the computing nodes, generating scheduling policy and so on. There are no tasks running on master node, and its workload is quit light.
- Distribution node. There can be multiple distribution nodes in our cluster, each stream can choose any one of them to flow into the cluster. The distribution nodes allocate streams according to the scheduling policy made by master node.
- Computing nodes. There are two functions of computing nodes, maintaining the data distributions of datasets keeping on themselves and executing the join processing.

We named our method as Distributed Stream Join (DSJ), which is composed of a distribution method and a join method. The former is executed on the master node while the latter runs on computing nodes.

The whole join process is executed as follows. The master monitors the conditions of the whole cluster, and makes scheduling policy according to the workload of all the computing nodes periodically. When a stream comes, it chooses a distribution node and

flows into the cluster through the node. Distribution nodes ask the master node for the scheduling policy periodically and record all the scheduling information. When receiving tuples, the distribution nodes decide the streams' destination nodes according to the information. Each computing node keeps parts of tuples in the streams and executes the join according to the join predicates.

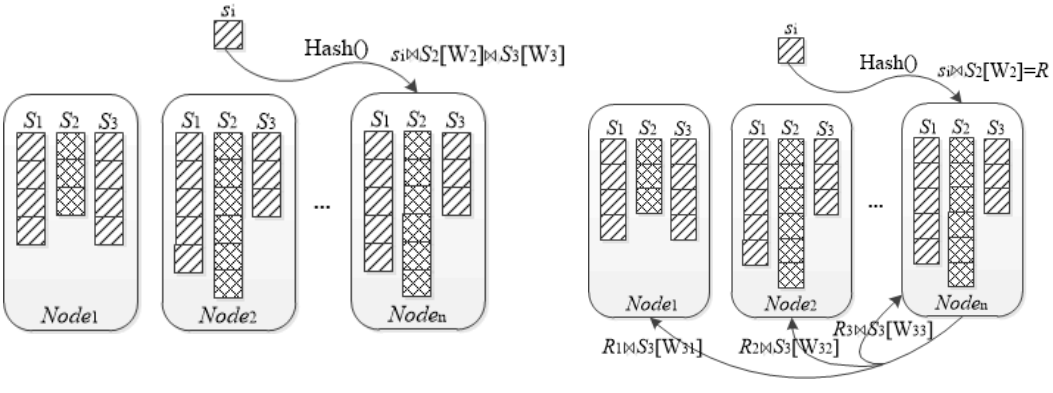
3. The DSJ Execution Model

In this section, we describe the whole process of multi-way windowed streams θ-joins in a shared-nothing cluster. In the following first subsection we discuss the distribution method of data streams, then, in the second subsection, we investigate the join process executed on a certain computing node.

3.1 Streams Distribution in Cluster

Apart from the load balance among all the computing nodes, we should also consider the relevance among tuples from different streams when executing the join processing in a distributed cluster. All the tuples satisfying the join predicates should meet with each other in a certain computing node for result integrity. For the purpose of above, it is necessary to replicate parts of tuples at different nodes or transfer the intermediate results among nodes.

In multi-way streams equi-joins $S_1[W_1] \bowtie_= S_2[W_2] \ldots \bowtie_= S_n[W_n]$, the join predicate is $(s_1.b_1 = s_2.a_2) \wedge (s_2.b_2 = s_3.a_3) \ldots \wedge (s_{n-1}.b_{n-1} = s_n.a_n)$, and the attributes a_2, $a_3 \ldots, a_n, b_1, b_2 \ldots, b_{n-1}$ can same or different. It is usual to allocate tuples according to attribute values. Equi-joins can be classified into two categories, including shared attribute joins and non-shared attribute joins. The former's attributes $a_2, a_3 \ldots, a_n, b_1, b_2 \ldots, b_{n-1}$ are same while the latter's are different. For shared attribute joins, we distribute the tuples to different nodes by one-hop hashing the shared attribute values. For non-shared attribute joins, we allocate the tuples by multi-hop hashing. In one-hop hashing, there are no intermediate results are transferred through network, all the tuples with the same attribute are allocated to the same node, and the whole join process is executed in one node when a new tuple comes. In multi-hop hashing, the tuples from different streams are hashed according to different attributes, the tuples satisfying the join predicate may be allocated to different nodes, and we need to transfer intermediates through network for purpose of integrity. The details of these two kinds of joins are illustrated in Figure 3.



(a) One-Hop Hash (b) Multi-Hop Hash

Figure 3. Distribution by Hash

Algorithm 1: Time-Slice Distribution Method Algorithm for Master

Input: all the Streams' buffers B_1, B_2…, B_n
The number of time slices *m*
Output: time slice information *TS*
1 $B \Leftarrow \max\{B_1, B_2 \ldots, B_n\}$;
2 $T \Leftarrow B.\text{size}$;
3 $t \Leftarrow$getCurrentTime();
4 **while** *true* **do**
5 create a new time slice *TS*;
6 *TS*.begin⇐*t*;
7 *TS*.end⇐$t+\frac{1}{m}T$;
8 find the lowest workload node's *ID*;
9 *TS.ID*⇐*ID*;
10 output *TS*;
11 sleep($\frac{1}{m}T$);
12 **end while**

Unfortunately, hash is not suitable for θ-joins, we could not divide streams only based on attribute values. As mentioned earlier, streams are processed in "on-the-fly" model, and memory is scarce resource. What's more, the whole sliding window could not be kept in a single node as the arrival rate increasing with the coming of the ear of big data. Here we employ transferring intermediate results method, and split sliding windows of streams into many sub-windows by time slices, then store the sub-windows in different computing nodes for joining in a parallel way, we call this distribution method as Time-Slice Distribution Method (TDM).

In TDM, the master node asks for each computing node's workload periodically, and selects the computing node with the minimum workload as the target node, we adopt

$$WL = \varphi_1 WL_{cpu} + \varphi_2 WL_{mem} + \varphi_3 WL_{net} \quad (2)$$

as the measurement of workload. WL_{cpu}, WL_{mem}, WL_{net} are the utilization ratios of CPU, memory and network, φ_1, φ_2 and φ_3 are the weight factors. The details are described in Algorithm 1 and the method of calculating the value of input parameter *m* will be given latter.

We set a buffer B_i for stream S_i to keep the latest tuples. Here we consider if a tuple s_i can joins with all the tuples generated during $[s_i.t_{app} - |W_j|, s_i.t_{app})$ from other stream S_j, the results are integrity. What's more, when $s_i.t_{arr} - s_i.t_{app} \geq |W|$, we drop it and it is not influence the integrity of results.

Theorem 1 The size of stream S_i's buffer $B_i \geq (T_c + T_t + 2|W|) \times r_i$.

Proof: If a tuple s_j from S_j is produced at t_{app}, and arrives at the cluster at t_{arr}. Assume it joins with other streams in the other of $S_1[W_1] \bowtie_\theta S_2[W_2] \ldots \bowtie_\theta S_{j-1}[W_{j-1}] \bowtie_\theta s_j \bowtie_\theta S_{j+1}[W_{j+1}] \ldots \bowtie_\theta S_n[W_n]$. When joining with tuples from stream S_i produced during $[t_{app} - |W_i|, t_{app})$, the calculation and transmission time have consumed are t_c and t_t, therefore the size of buffer B_i is

$$\begin{aligned}&(t_c + t_t + s_j.t_{arr} - s_j.t_{app} + |W_i|) \times r_i \\ &\leq (t_c + t_t + |W| + |W_i|) \times r_i \quad (3) \\ &\leq (T_c + T_t + 2|W|) \times r_i \\ &\leq B_i\end{aligned}$$

From (3) we can get that when s_j joins with $S_i[W_i]$, all the tuples in sliding window W_i are kept in B_i . □

Assume $T_i = B_i.size$, we divide T_i into m equal disjoint time slices: $[t, t+\frac{1}{m}T_i)$, $[t+\frac{1}{m}T_i, t+\frac{2}{m}T_i)$, $[t+\frac{2}{m}T_i, t+\frac{3}{m}T_i)$ ..., $[t+\frac{m-1}{m}T_i, t+T_i)$ and B_i is divided into m segments by time slices. We keep the segments in different computing nodes. Because of being kept in buffer, the sliding window W_i is divided into m segments (we call sub-windows) too. For example, shown as (4), in $S_1[W_1] \bowtie_\theta S_2[W_2] \ldots \bowtie_\theta S_n[W_n]$, each sliding window is divided into m sub-windows and we can get that when a new tuple arrives, it needs to join with $(n\text{-}1)\times m$ sub-windows for the purpose of integrity. The sub-windows may be kept in different nodes, so the intermediate results have to be transferred among nodes through network. This increases the time delay and network load.

$$\begin{aligned} & S_1[W_1] \bowtie_\theta S_2[W_2] \ldots \bowtie_\theta S_n[W_n] \\ =& S_1\left[\sum_{i_1=1}^{m} W_{1i_1}\right] \bowtie_\theta S_2\left[\sum_{i_2=1}^{m} W_{2i_2}\right] \ldots \bowtie_\theta S_n\left[\sum_{i_n=1}^{m} W_{ni_n}\right] \\ =& \sum_{i_1=1}^{m} S_1[W_{1i_1}] \bowtie_\theta \sum_{i_2=1}^{m} S_2[W_{2i_2}] \ldots \bowtie_\theta \sum_{i_n=1}^{m} S_n[W_{ni_n}] \\ =& \sum_{i_1=i_2 \ldots i_n=1}^{m} S_1[W_{1i_1}] \bowtie_\theta S_2\left[W_{2i_2}\right] \ldots \bowtie_\theta S_n[W_{ni_n}] \end{aligned} \quad (4)$$

To deal with the problem, we set

$$T = \max\{T_1, T_2 \ldots, T_n\} \quad (5)$$

and all the streams share the same time slices. No matter which stream come from during $[t+\frac{i-1}{m}T, t+\frac{i}{m}T)$, all the tuples are kept in the same node. Shown as Algorithm 2, we build up a distribution table to keep the location information of sub-windows on each distribution node. When a tuple comes, our TDM finds out all the related sub-windows' locations and generates a route to ensure the tuple can join with all the related tuples from other streams. Then the tuple is distributed to all the nodes in the route.

Algorithm 2: Time-Slice Distribution Method Algorithm for Distribution Nodes

Input: s_i from Stream S_i distribution table DT
the sizes of all the sliding windows $|W_1|$, $|W_2|\ldots,|W_n|$
Output: s_i with route and mask information $(s_i, route, mask)$

```
route ⇐ ∅;
pid ⇐ fork();
if pid = 0 then
    while true do
        TS⇐getTSFromMaster();
        insert TS into DT;
        sleep(TS.end-TS.begin);
    end while
else
    while true do
        s_i ⇐recv();
        s_i.ID ⇐ TS.ID;
        route ⇐getAllTargetNodes(DT, W_i);
        mask ⇐ (0 ... 010 ... 0);
        send(s_i, route, mask) to all the nodes in route;
    end while
end if
```

Assume, the arrival routes of $S_1, S_2 \ldots, S_n$ are $r_1, r_2 \ldots, r_n$. A new coming tuple s_i consumes Δt to finish the join process $S_1[W_1] \bowtie_\theta S_2[W_2] \ldots \bowtie_\theta S_{i-1}[W_{i-1}] \bowtie_\theta s_i \bowtie_\theta S_{i+1}[W_{i+1}] \ldots \bowtie_\theta S_n[W_n]$ on a single node. The size of memory in each node is *M*, so

$$\begin{cases} \frac{T_c}{\Delta t} m \geq (T_t + T_c) \sum_{i=1}^{n} r_i \\ Mm \geq Bn \end{cases} \tag{6}$$

$$m \geq \max\left\{\left(1 + \frac{T_t}{T_c}\right) \Delta t \sum_{i=1}^{n} r_i, \frac{Bn}{M}\right\} \tag{7}$$

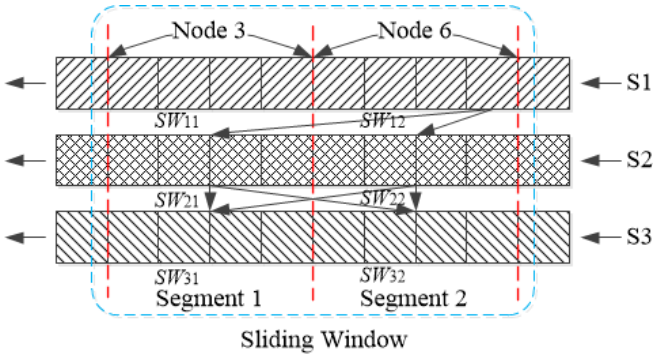


Figure 4. Distribution Method

In Figure 4, for example, we divide $S_2[W_2]$ and $S_3[W_3]$ into two sub-windows respectively by the same time slices, including SW_{21}, SW_{22}, SW_{31} and SW_{32}. SW_{21} and SW_{31} are distributed to $Node_3$ while SW_{22} and SW_{32} are allocated to $Node_6$. When a new tuple s_{1i} from S_1 comes, the join $s_{1i} \bowtie_\theta S_2[W_2] \bowtie_\theta S_3[W_3]$ is translated to

$$\sum_{j=k=1}^{2} s_{1i} \bowtie_\theta S_2[W_{2j}] \bowtie_\theta S_3[W_{3k}] \tag{8}$$

and s_{1i} is routed to $Node_3$ and $Node_6$, the join process $s_{1i} \bowtie_\theta S_2[W_{21}] \bowtie_\theta S_3[W_{32}]$ and $s_{1i} \bowtie_\theta S_2[W_{22}] \bowtie_\theta S_3[W_{31}]$ have to transfer intermediate results between $Node_3$ and $Node_6$ during the process while $s_{1i} \bowtie_\theta S_2[W_{21}] \bowtie_\theta S_3[W_{31}]$ and $s_{1i} \bowtie_\theta S_2[W_{22}] \bowtie_\theta S_3[W_{32}]$ need not. The method that all the tuples from different streams in the same time slice are distributed to the same node can avoid transferring parts of the intermediate results among nodes. What' more, it can also reduce network load and time delay.

Theorem 2 There are at least $\frac{1}{m}$ intermediate results need not to transfer through network, where *m* is the number of time slices in sliding windows.

Proof: We assume there are *n* streams and divide each sliding window into *m* sub-windows (time slices). The volume of each sub-window is defined as $\bar{V}$, and join selectivity between data stream S_j and S_{j+1} is $R_{(j,j+1)}$. When a new sub-window is created, it needs to be routed to all the computing nodes that holding the sub-windows of current sliding windows. During each stage in the process of joining, the intermediate results are routed to *m* nodes and the volume of the intermediate results need to be transferred is

$$m\bar{V} + m^2\bar{V}^2 R_{(1,2)} \ldots + m^n \bar{V}^n \prod_{i=1}^{n-1} R_{(i,i+1)}$$
$$= m\bar{V}\left(1 + \sum_{i=1}^{n-1} \left(m^i \bar{V}^i \prod_{j=1}^{i} R_{(j,j+1)}\right)\right) \tag{9}$$

and

$$\bar{V} + m\bar{V}^2 R_{(1,2)} \ldots + m^{n-1} \bar{V}^n \prod_{i=1}^{n-1} R_{(i,i+1)}$$

$$=\bar{V}(1+\sum_{i=1}^{n-1}(m^i\bar{V}^i\prod_{j=1}^{i}R_{(j,j+1)})) \quad (10)$$

intermediate results need not to transfer. That means

$$\frac{\bar{V}(1+\sum_{i=1}^{n-1}(m^i\bar{V}^i\prod_{j=1}^{i}R_{(j,j+1)}))}{m\bar{V}(1+\sum_{i=1}^{n-1}(m^i\bar{V}^i\prod_{j=1}^{i}R_{(j,j+1)}))}=\frac{1}{m} \quad (11)$$

of all the intermediate results have no need to be transferred through Ethernet. When tuples in two or more time slices of the sliding window are kept in the same node, the proportion is greater than $\frac{1}{m}$.

□

3.1 Streams Distribution in Cluster

After exported from distribution node, two kinds of information are added to each tuple s_i for the purpose of result integrity.

- Route. It records all the nodes that hold the sub-windows of sliding windows. After a intermediate result is produced, it will be transferred to all the nodes in the Route.
- Mask. All the streams that have not been joined with the tuple or intermediate result are recorded here.

Route information makes sure all the sub-windows can join with s_i and Mask information guarantees all the other streams can be joined with. The whole join process related with s_i is running on several computing nodes and we divide the process into several steps, in each step two streams join with each other or the intermediate results join with one stream, shown as Figure 5.

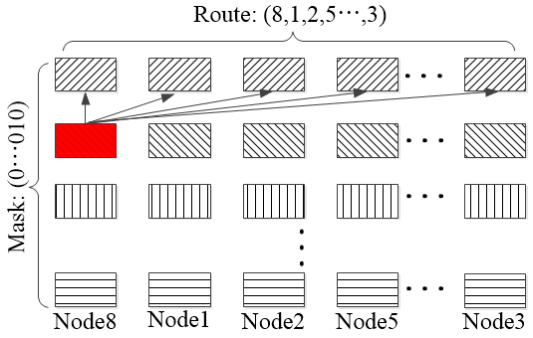


Figure 5. Route and Mask

In the above subsection, we divide each stream's sliding window into *m* sub-windows by the public time slices and all the tuples in the same time slice are kept in the same computing node. For the purpose of integrity, when a tuple comes, it needs to join with all the sub-windows of the sliding windows, the Route information is marked as

$$\underbrace{(id_a, id_c, id_g \ldots, id_f)}_{k}$$

id_i is the ID of a certain computing node keeping the sub-windows of current sliding windows. What's more, $1 \le k \le m$, because sub-windows of a stream's current sliding

window produced in different time slices may be kept in the same computing nodes. The intermediate results containing s_i are transferred to all the computing nodes recorded in the Route in the next step.

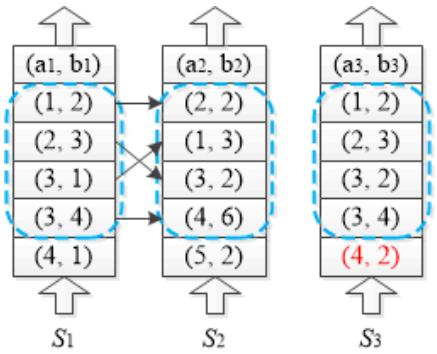


Figure 6. Three-Way Streams Joins

Algorithm 3: Transfer Join Method Algorithm

Input: a tuple or intermediate result s_{cur}
all the tuples in sliding windows on current nodes
histograms $H = \{H_1, H_2 \ldots, H_n\}$
Output: join result *res*

```
1  s_cur ⇐recv();
2  mask ⇐ s_cur.getMask();
3  route ⇐ s_cur.getRoute();
4  {l, r} ⇐getTwoCandidates(s_cur, L, C, H);
5  purge all the tuples that out of date in S_l and S_r;
6  for s_i in S_l do
7     for s_j in S_r do
8        if s_i =_θ s_j then
9           res ⇐createTuple();
10          res.setContent(s_cur, s_l, s_j);
11          res.setRoute(route);
13          res.setMask(l, r, mask);
14          if res.getMask()=null then
15             Output(res);
16          else
17             Send(route, res);
18          end if
19       end if
20    end for
21 end for
22 if s_cur is a tuple then
23    ID ⇐ getHostID();
24    if ID = s_cur.ID then
25       Insert s_cur into sliding window of S_cur in host ID;
26    end if
27 end if
```

As is known to all, different join orders produce different volumes of intermediate results and consume different time. For example, in Figure 6, if the join process is executed in the order of $(S_1[W_1] \bowtie_\theta S_2[W_2]) \bowtie_\theta S_3[W_3]$. When a new tuple (4, 2) from S_3 arrives, $S_1[W_1] \bowtie_\theta S_2[W_2]$ is executed first and produces the intermediate results set *temp*={(1, 2, 2), (2, 3, 2), (3, 1, 3), (3, 4, 6)}. Then $temp \bowtie_\theta$ (4, 2) is executed and produces nothing, $S_1[W_1] \bowtie_\theta S_2[W_2]$ is unnecessary. But when executing the process in the order of $S_1[W_1] \bowtie_\theta (S_2[W_2] \bowtie_\theta S_3[W_3])$, $S_2[W_2] \bowtie_\theta S_3[W_3]$ is executed and

produces nothing, the whole process is finished. $S_1[W_1] \bowtie_\theta (S_2[W_2] \bowtie_\theta S_3[W_3])$ is a better choice compared to $(S_1[W_1] \bowtie_\theta S_2[W_2]) \bowtie_\theta S_3[W_3]$.

Normally, there are two phases for query optimization, including logical optimization and physical optimization. The former is focus on SQL equivalent transformation and the latter concerns about the join order. Here we only pay attention to physical optimization. For the purpose of generating an effective join order, one of useful method in traditional database systems is to grasp the data distributions of all the tables and make an optional join order according to these distributions. Similar to the traditional database, we build a statistics unit in each computing node and maintain equi-width histograms for the distribution of tuples in sliding windows. We optimize the join order based on the information provided by the histograms[16][17].

There are several strategies for generating the optimized join order, for example, PostgreSQL adopts dynamic programming while MySQL chooses greedy method. As is known to all, dynamic programming can get an optimal policy but it is costly and not suitable for a real-time environment. Greedy method can only get a local optimal policy but the time complexity is low and it is available for stream model.

As mentioned earlier, after exported from a distribution node, the mask information is added to tuple s_i, it is a bit sequence, just like 00…0100…0:

$$\overbrace{(00\ldots0}^{i-1}\underbrace{100\ldots0)}_{n-i}$$

There are *i*-1 and *n*-*i* bits valued 0 on the left and right of 1. If the bit on the *j*th location is 0, it means the tuple has not joined with stream S_j*;* if the bit is 1, it means the tuple has joined with S_j. Shown as Algorithm 3, after the tuple s_{cur} is transferred to a computing node, the join operation gets the tuple's Route and Mask. Mask records the streams' IDs that have not joined with S_{cur}, the join operation chooses two streams S_l and S_r from Mask. The volume of $S_l \bowtie_\theta S_r$ is minimum in all the possible combinations of two streams in Mask. In line 4 of Algorithm 3, when a tuple or intermediate result s_{cur} comes, we get two candidates, which may be two streams or one is a stream and the other one is s_{cur}. Then, the join operation joins S_l and S_r, the results are output or sent to the next step according to Route. The content of *res* is set to a combination of s_{cur}.content, s_i.content and s_j.content. And its mask is assigned to

$$s.\text{Mask OR } (00\ldots0\ \overbrace{1}^{the\ lth\ bit}\ 0\ldots0\ \underbrace{1}_{the\ rth\ bit}\ 0\ldots00)$$

It illustrates that stream S_l and S_r have been joined with s_{cur}. What's more, if s_{cur} is a tuple (not an intermediate result), it will be stored in the node keeping the current sub-windows.

Actually, whole join process is a variety of greedy method. The join order is not fixed, in each step of join process, the system always chooses the candidate streams producing the minimum intermediate results to execute joining. The join order may change as the distributions of tuples in sliding windows varying. We named this method Transfer Join Method (TJM).

4. Experiments

In the section, we run DSJ in a cluster with shared-nothing architecture and present some experimental results. The cluster consists of one master node, one distribution node and ten computing nodes. Each node is with an Intel (R) Xeon (R) 2.67GHZ CPU, 4GB memory and connects with each other by 100MB Ethernet. The datasets used in the paper are synthetic and real. The synthetic dataset is generated by Zipf distribution generator Coded by Kenneth J. Christenen[11] and the keys of tuples range from 1 to 3000. The real

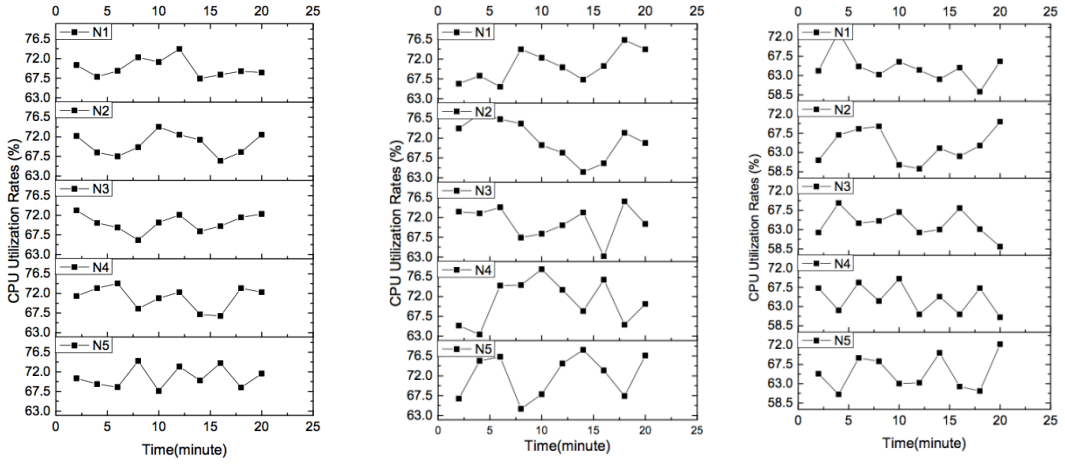
dataset is the cars' GPS information from a certain large city and they consist of real-time streams.

In the first two experiments, we execute four-way streams joins $S_1[W_1] \bowtie_\theta S_2[W_2] \bowtie_\theta S_3[W_3] \bowtie_\theta S_4[W_4]$ to test out DSJ, and the join predicate is $(s_1.b_1 =_\theta s_2.a_2) \wedge (s_2.b_2 =_\theta s_3.a_3) \wedge (s_3.b_3 =_\theta s_4.a_4)$. We compare our DSJ method with Gu's Coordinated Tuple Routing (CTR) strategy. The mainly difference between DSJ and CTR is that all the streams shared the same time slices in DSJ and all the tuples from different streams coming during the same time slice are kept in the same node while each stream has its own time slices in CTR and tuples from different streams are distributed according to their own time slices. We focus on the load balance and throughput of the cluster in the first two experiments. In the third experiment, we change the number of computing nodes and the number of streams to test the factors influencing on performance.

4.1 Load Balance

In this subsection, we use the synthetic Zipf distribution dataset and the real GPS dataset to test the load balance among all the computing nodes. In the experiment, there are five computing nodes in the cluster. We use Alpha represent the skew degree of the distribution in Zipf dataset, Alpha $\in (0,1)$. With the decreasing of Alpha, the Zipf distribution tends to be a uniform distribution. In this paper, we set the value of Alpha to 0.8, it means there are a lot of keys with small values. The GPS dataset contains the records of over 15 thousand taxis, these cars report their locations every 15-30 second. We consider these records follow uniform distribution.

Figure 7(a) and (b) show that both DSJ and CTR can maintain the load balance among the computing nodes. When a new time slice begins, the master node always selects the computing node with the minimum workload as the target node and the target node keeps all the tuples arriving during this time slice. The experiments show that when using DSJ, the fluctuation of all the computing nodes' work load is a little bigger than that of CTR. The reason may be that each stream has its own time slices and the slices from different streams are allocated independently in CTR strategy. For example, the tuple s_{1i} from the stream S_1 and the tuple s_{2j} from the stream S_2 come at the same time, but they may be kept in different computing nodes, it is different from DSJ. All the streams share the same time slices in DSJ, and all the tuples from different streams are kept in the same node during each time slice. The data is distributed more uniform among different computing nodes in CTR than DSJ.



a) Zipf with CTR (b) Zipf with DSJ (c) GPS information with DSJ

Figure 7. CPU Utilization Rates of Computing Nodes

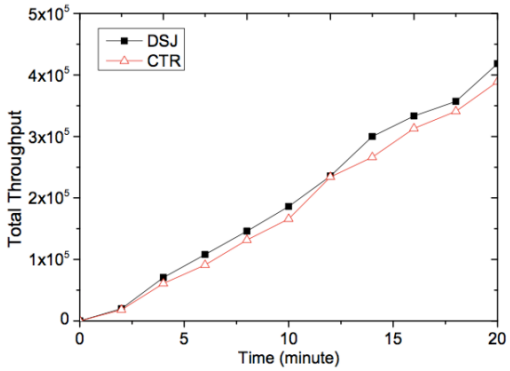
When running DSJ on the GPS dataset, the utilization rates of CPU are much lower than that of the Zipf dataset. It is because the data in Zipf distribution dataset are skew and the data in the GPS dataset are uniform. In Zipf distribution, when a new tuple s_i comes, it is very likely that s_i is one of the frequency items and may produce a lot of intermediate and final results, this occupy many computing resources.

As Figure 7 (b) and (c) shown, DSJ method can guarantee load balance on both non-uniform and uniform distributions. In hash join, tuples are distributed according to attribute values and there are strong correlations among the tuples. Tuples with the same attribute value are allocated to the same node. If a data distribution is skew, the tuples with high frequency are always allocated to fixed nodes and it cannot guarantee the load balance among different computing nodes. DSJ method divides the streams according to time slices and always selects the lowest workload node as the target when a new time slice begins, this breaks the correlations among tuples. Tuples with the same attribute value may be allocated to different nodes. This can guarantee the load balance.

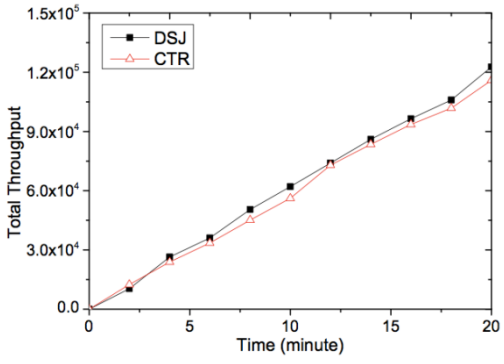
4.2 Throughput

In this subsection, we run DSJ and CTR methods on both the Zipf and GPS dataset. In the experiment, there are eight computing nodes in the cluster. When a new time slice is created, all the tuples coming during the time slice are kept in the node with the lowest workload in DSJ and CTR, and some time slices in the same sliding window may be kept in the same nodes. If the number of the computing nodes N is much bigger than the number of time slices m, it is not likely that multiple time slices in current sliding window are kept in the same computing nodes. Otherwise, it is possible that some time slices may be kept in the same nodes.

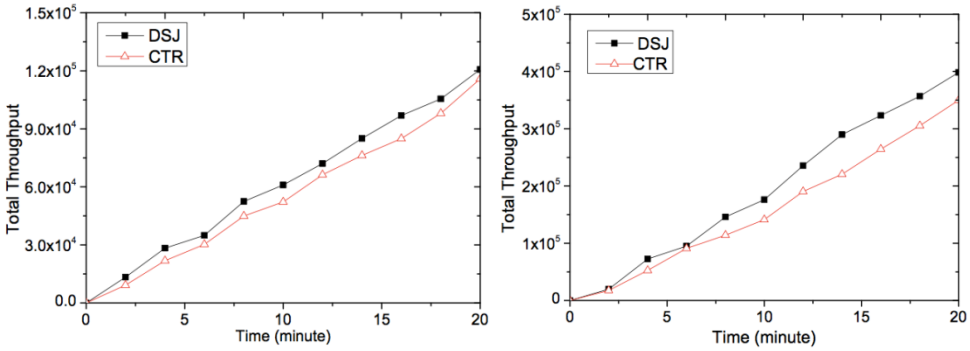
In the experiment, we test the throughput in two stages. We only consider the workload as the measure and the new time slice is always allocated to the lowest workload node in the first stage. In the second stage, besides considering the workload, we make sure that all the time slices in current sliding window are kept in different nodes. That means when a time slice is allocated, the system checks whether there is any other time slices in current sliding window has already been kept in the lowest workload node. If yes, find another node; if not, choose the node as the target.



(a) Zipf Distribution in the First Stage



(b) GPS Information in the First Stage



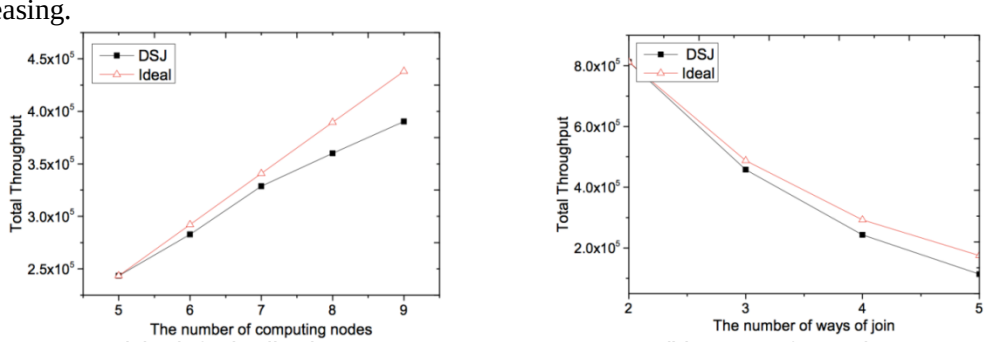
(c) Zipf Distribution in the Second Stage (d) GPS Information in the Second stage

Figure 8. Throughput Comparison

As shown in Figure 8, (a) and (b) are the results of the first stage, the performance of DSJ is better than CTR. DSJ employs a derived greedy algorithm to decide the join order and it always chooses the two candidates which produce the minimum intermediate results in each step. This reduces some unnecessary calculation and transmission. (c) and (d) show the results of the second stage, the throughput of DSJ is much bigger than that of CTR. The throughput of DSJ is improved about 10% when running on Zipf and GPS datasets than CTR. DSJ guarantees that at least $\frac{1}{m}$ intermediate results do not need to be transferred among node while CTR has no such guarantee. When running in the first stage, many time slices may be allocated to the same nodes and the performances of DSJ and CTR are similar with each other. In the second stage, we enforce all the time slices are kept in different nodes to simulate the scene that N is much bigger than m. Furthermore, when running on the GPS dataset, the throughput is little than Zipf dataset, because the former is a uniform and sparse dataset and the join selectivity is low.

4.2 Influence Factors on Performance

In this subsection, we discuss the impact of some factors on the performance of DSJ. Two factors are considered, including (1) the number of computing nodes and (2) the number of streams. We run four-way joins on the clusters containing different number of computing nodes. What's more, the number of time slices in the sliding window is equal to the number of computing nodes. We run DSJ 10 minutes for each case. The result is illustrated in Figure 9 (a). As the number of computing nodes increasing, the throughput of cluster grows. The line with triangle symbol shows the ideal case and the growth rate of throughput is fixed as the number of computing nodes increasing. The growth rate of DSJ is lower than the ideal case. Because the percentage of intermediate results that do not need to be transferred through network decreases as the number of computing nodes increasing.



(a) Zipf Distribution (b) GPS Information

Figure 9. Influence Factors on Performance

ISSN: 2005-4262 IJGDC

In the test of the influence of the number of streams on performance, we set the join selectivity of two streams is 0.6 and the number of computing nodes is 5. As shown in Figure 8 (b), as the number of streams increasing, the throughput decreases. The throughput is lower than the ideal case, because many intermediate results are transferred through network and this increases the delay of the whole join process.

5. Related Work

With the development of sensors, RFIDs and other real-time applications, multi-way streams joins using cluster draws more and more attention.

In our previous work[19], we discuss the problem of joining two-way streams in a cluster. Since there are only two streams, we propose a novel architecture named Dual-Assembly-Pipeline (DAP) strategy, in which the cluster architecture is organized as a pipeline. DAP makes full use of the advantage of a pipeline. Two streams flow into the cluster in the opposite directions and this can make sure all the tuples in the two sliding windows can meet with each other only once. We split the sliding windows into each other in the pipeline evenly and it is unnecessary to consider the load balance problem and the integrity of join results. However, DAP is only suitable for two-way streams joins and it could not process n-way streams joins, where n>2.

In [3], Babcock reviews the models and issues in a stream system, including the definitions, the timestamps, the sliding window technology, histograms[7-9][12-14], and so on. In [2], Golab also presents some examples of stream applications and lists the requirements of stream processing, such selections, frequent item queries, joins and windows queries, *etc*. The two papers only consider the streams joins on single node, but give no details in distributed shared-nothing architecture.

Gu proposes two methods for multi-way windows streams joining in [17] named Aligned Tuple Routing (ATR) and Coordinated Tuple Routing (CTR). ATR selects a stream as the main stream periodically. Some parts of the main stream, such as window $[t, t+T)$ are routed to one certain node according to the workloads of all the computing nodes and other streams' tuples related to the sliding window will be routed to the same node too. ATR is only suitable for streams with small sliding windows, but DSJ can deal with streams with large sliding windows. The difference between CTR and DSJ is that DSJ assumes the sliding window is so big that tuples in the sliding windows are distributed to multiple computing nodes and all the tuples from different streams in the same time slice are allocated to the same node, this can avoid transferring parts of intermediate results.

Ananthanaraynan and Basker *et al* build up a special data streams joining system named Photon, which joins user click stream and query stream in the Google engine and provides fault-tolerance guarantee in the level of data center in [15]. What's more, Photon only provides two-way streams joins and gives no details on multi-way streams joins. We focus on multi-way streams θ-joins in this paper. In [18], Gedik proposes a method for parallel streams joins using multiple cell processors. In [16], Kang presents the basic steps of a sliding window join between streams A and B. He also evaluates the performance of the windowed join over unbounded streams. DSJ can run on a distributed environment and expends Kang's algorithm to multi-way streams θ-joins.

In [13], Chakraborty parallelizes the windowed joins over a shared-nothing cluster by hash-partitioning the input streams, distributing a subset of partitions to the available nodes and adjusting the data flow towards the nodes based on the availability of resources within the nodes. Hash-partition is only suitable for equi-joins, when the join predicate is arbitrary, the method is not effective. In DSJ, the join predicate is $(s_1.b_1 =_\theta s_2.a_2) \wedge (s_2.b_2 =_\theta s_3.a_3) \ldots \wedge (s_{n-1}.b_{n-1} =_\theta s_n.a_n)$, $=_\theta$ can be >, <, ≥, *like*, and so on, it is more common than hash-partition.

6. Conclusion and Future work

In this paper, we present a solution for multi-way streams θ-joins in a cluster named Distribution Streams join (DSJ). DSJ contains a distributions method named Time-Slice Distribution Method and a join method named Transfer Join Method (TJM). DSJ can process multi-way windowed streams θ-joins under the join predicate $(s_1.b_1 =_\theta s_2.a_2) \wedge (s_2.b_2 =_\theta s_3.a_3) \ \ldots \ \wedge (s_{n-1}.b_{n-1} =_\theta s_n.a_n)$. The experiments show that DSJ can effectively keep load balance on the uniform and non-uniform datasets. What's more, DSJ can decrease the intermediate results transmission and improve throughput compared with the previous work. However, we do not consider the fault-tolerance of the cluster, the streams are processed in a "on-the-fly" style and the sliding windows are kept in the memory. Once a certain node is unavailable, all the tuples kept in it will be lost. This leads to incomplete results. In the future work, we will focus on fault-tolerance problem in stream joining system.

Acknowledgements

We thank the Supercomputing Center at University of Science and Technology of China (USTC) for their platform and technical support. This work is supported by the National Key Technology R&D Program under Grant No. 2012BAH17B03. the open project of State Key Laboratory of high performance servers and storage techniques, investigation on feasible techniques of virtual machine migration across virtual pools under Grant No. 2014HSSA09 and the key science research project of Anhui province, the design and research of large-scale data stream processing and storage system in cloud environment under Grant No. KJ2014A150.

References

[1] S. Muthukrishnan, Data Streams: Algorithms and Applications, Foundations and Trends in Theoretical Computer Science, Vol. 1, No 2(**2005**) 117-236.
[2] L. Golab and M. Tamer Özsu, Issues in Data Stream Management, SIGMOD Record, Vol. 32, No. 2, **(2003**).
[3] B. Bacock, S. Babu, Models and Issues in Data Stream Systems, ACM PODS, Madison, Wisconsin, USA, (**2002**).
[4] L. Golab and M. Tamer Özsu, Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams, Proceedings of the 29th VLDB Conference, Berlin, Germany, **(2003**).
[5] A. Arasu, S. Babu, J. Widom, The CQL continuous query Language: Semantic Foundations and Query Execution, The International Journal on Very Large Data Bases, Volume 15 Issue 2, USA, (**2006**).
[6] P K. Agarwal, J Xie, J Yang, H Yu, "Input-Sensitive Scalable Continuous Join Query Processing", ACM Transactions on Database Systems (TODS).Volume 34 Issue 3, NY, USA, (**2009**).
[7] C Jin, K Yi, L Chen, Sliding-Window Top-k Queries on Uncertain streams, Proceedings of the VLDB Endowment, Volume 1 Issue 1, Pages 301-312, (**2008**).
[8] L. K. Lee, H. F. Ting, Maintaining Significant Stream Statics over Sliding Windows, SODA '06 Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, Pages 724-732, PA, USA, (**2006**).
[9] S Tirthapura, B Xu, C Busch, Sketching Asynchronous Streams over A Sliding Window, PODC '06 Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing, Pages 82 - 91, NY, USA, (**2006**).
[10] [M. Fang, N. Shivakumar, and H. Garcia-Molina, Computing Iceberg Queries Efficiently, In Proc. of the 1998 Intl. Conf. on VLDB, pages 299-310, (**1998**).
[11] K J. Christensen, http: //www.csee.usf.edu/ christen/tools/genzipf.c.
[12] M. Greenwald and S. Khanna. Space-Efficient Online computation of Quantile Summaries. In Proc. Of the 2001 ACM SIGMOD Intl. Conf. on Management of Data, Pages 58-66, (2001).
[13] A. Gilbert, S. Guha, p. Indyk, Fast, Small-space Algorithms for Approximate Histogram Maintenance, In Proc. of the 2002 annual ACM Symposium on Theory of Computing, (**2002**).
[14] H. Jagadish, N. Koudas, *etc*., Optimal Histograms with Quality Guarantees, In Proc. of the 1998 Intl. Conf. on VLDB, pages 275-286, (**1998**).
[15] R Ananthanaryanan, V Basker, *etc*., Photon: Fault-Tolerant and Scalable Joining of Continuous Data Streams, SIGMOD '13 Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, Pages 577-588, NY, USA, (**2013**).

[16] J Kang, F Jeffrey, Naughton and Stratis D. Viglas, Evaluating Window Joins over Streaming Data, Annual ACM-SIAM Symposium on Discrete Algorithm(**2002**).
[17] X Gu, P S. Yu and H Wang, Adaptive Load Diffusion for Multi-Way Windowed Stream Joins, Data Engineering, 2007, ICDE 2007, IEEE 23rd International Conference on, (**2007**).
[18] B Gedik, R Rajesh . Bordawekar and P S. Yu, "CellJoin: A Parallel Stream Join Operator for the Cell Processor", The VLDB journal, Volume 18, Issue 2, pp 501-519, (**2009**).
[19] X Liu, X Fan, J Li , "A Novel Parallel Architecture with Fault-Tolerance for Joining Bi-directional Data Streams in Cloud[C]". //International Conference on Cloud Computing & Big Data. IEEE Computer Society, (**2013**).
[20] Q Chen, M Hsu, "Stream-Join Revisited In the Context of Epoch-Based SQL Continuous Query", IDEAS '12 Proceedings of the 16th International Database Engineering & Applications Symposium, Pages 130-138, NY, USA, **(2012**).
[21] B Catnia, G Guerrini, "Towards Relaxed Selection and Join Queries over Data Streams, ADBIS'12 Proceedings of the 16th East European conference on Advances in Databases and Information Systems", pp. 125-138, Berlin, (**2012**).
[22] A. Toshniwal and S. Taneja, Storm@twitter, SIGMOD'14 Proceedings of the 2014 ACM SIGMOD international conference on management of data, pages 147-156, NY, USA, (**2014**).
[23] L Neumeyer B, Robbins, A Nair, "S4: Distributed Stream Computing Platform[C]". 2010 IEEE International Conference on Data Mining Workshops. IEEE Computer Society, (**2010**).
[24] M Khalilian, N Mustaph, "Data Stream Clustering: Challenges and Issues[J]", Lecture Notes in Engineering & Computer Science,**(2010).**

Authors

Xinchun Liu, He received the B.E. degree in Computer Science from QingDao University in 2010. He is currently working toward the Ph.D. degree at the School of Computer Science and Technology in University of Science and Technology of China (USTC). His research interests include parallel algorithms, Cloud Computing and High Performance Computing.

Jing Li, He receive his B.E. degree in Computer Science from University of Science and Technology of China (USTC) in 1987, and Ph.D. in Computer Science from USTC in 1993. Now he is a Professor in the School of Computer Science and Technology at USTC. His research interests include Distributed Systems, Could Computing, Big Data Processing and Mobile Computing.

Xiaopeng Fan, He received the Ph.D. degree in Computer Science from Hong Kong Polytechnic University in 2010. He also received the B.E. and M.E. degrees in Computer Science from Xidian University, Xi'an, China, in 2001 and 2004 respectively. He is currently an Assistant Professor in Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China. His research interests include Big Data Analytics, Mobile Computing, Cloud Computing. He has served as a reviewer for several international journals and conference proceedings.

Jun Chen, She received the M.E. degree in Beiyou University in 2010. Now she is a senior engineer in Xinhua News Agency, Beijing.