Send Orders for Reprints to reprints@benthamscience.ae

Open Access

Research on Sliding Window Join Semantics and Join Algorithm in Heterogeneous Data Streams

Du Wei[1,2,4] and Zou Xianxia[*,3,4]

[1]*Department of Computer Science, Guangdong Police College, Guangzhou 510232, P.R. China;* [2] *Guangzhou Key Research Center of Public Safety, Guangzhou 510232, P.R. China;* [3]*Department of Computer Science, Jinan University, Guangzhou 510632, P.R. China;* [4]*State Key Laboratory of Software Engineering, Wuhan University, 430072, P.R. China*

Abstract: Sliding windows of data stream have rich semantics, which results all kinds of window semantics of different data stream, so join semantics between the different types of windows becomes very complicated. The basic join semantic of data streams, the join semantic of tuple-based sliding window and the join semantic of time-based sliding window have partly solved the semantics of stream joins, but the heterogeneity of sliding windows is difficult to be solved. In this paper we present the join semantic model based on matching window identifies for joining of multi-data stream. We make use of window identifies to shield the difference of window attribute, window size, and window slide. In this paper, a sliding window is divided into a number of sub-windows when the newest sub-window fills up it and it is appended to the sliding window while the oldest sub-window in the sliding window is removed. We use the equivalence relation of overlapping sub-window belonging to the adjacent sliding window to reduce the number of join computing. We propose the corresponding algorithm of window join to maintain the window. The theoretical and experimental analysis show that the joining model of window identifies can synchronize multiple data stream.

Keywords: Data stream, heterogeneous sliding window, join semantics, join algorithm.

1. INTRODUCTION

As the merger of multiple data stream (MDS) has drawn attention in related fields, various join algorithms and semantic models have been advocated. For example, Xjoin [1] algorithm, the first algorithm used to deal with the MDS join in unstable network environment, employs non-block join algorithm. It includes three phases. First, uses symmetric hash join (SHJ) [2] to handle the join computing in memory; second, supplements the unfinished join of the first phase; third, does the further checkup for the first two stages and produces the final result. By extending the traditional hash join algorithm, non-block SHJ algorithm is able to support the flow process and maintain hash buckets of both source A and source B. It accepts new tuple t from source A, directly probes the hash bucket of Source B, and meanwhile puts tuple t into the hash bucket of source A through hash function. However, SHJ is suitable only when memory was large enough for the both the two tables. HMJ [3] (hash-merge join) algorithm improved the XJoin by using the traditional merge join algorithm in the second phase, and algorithms like RPJ [4] (rate-based progressive join) and DPHJ [5] (double pipelined hash join) have made further refinements. XJoin, HMJ (hash merge join) and PMJ [6] algorithms, which focus on the approximate join computing of real-time data stream and has no explicit join semantics.

To deliver the semantics more clearly, the window mechanism was introduced into the query computation of the data stream. Document [7] considered the join between logic windows, while document [8] took that between the physical windows into account, and document [9] provided join query algorithms according to different arrival rates of the data stream. However, the difference between data streams is much more complex. Different window types, slides and initial points will make the semantics in window synchronization more complicated. When dealing with window synchronization over MDS, some data stream management systems (DSMS) may adopt simple approaches like using common time as the upper bound of the windows or advancing the windows at the same time [10]. CQL/STREAM [11]'s MDS merger uses common time points as the upper bounds of the windows, whereas TelegraphCQ [12] employs common iterative amount to achieve this goal.

In system implementation, SyncSQL/Nile [13] formally proposed the merge problem of MDS and its calculation model. In that system, logical time is the common time domain, and the synchronization point is determined according to the slide of the window. If a time point is the upper bound of two data streams, it's called full synchronized time point and other upper bounds are partial synchronized points. The merger of MDS only happens at the full synchronization points and partial synchronization points. The merger thought of MDS in SyncSQL/Nile is shown in Fig. (**1**). To merge data streams, the SyncSQL/Nile system uses logical time as the time domain and has the same start time points. Let the slides of two data streams are two and three time units respectively, then the time points which are only multi-

1874-110X/15 2015 Bentham Open

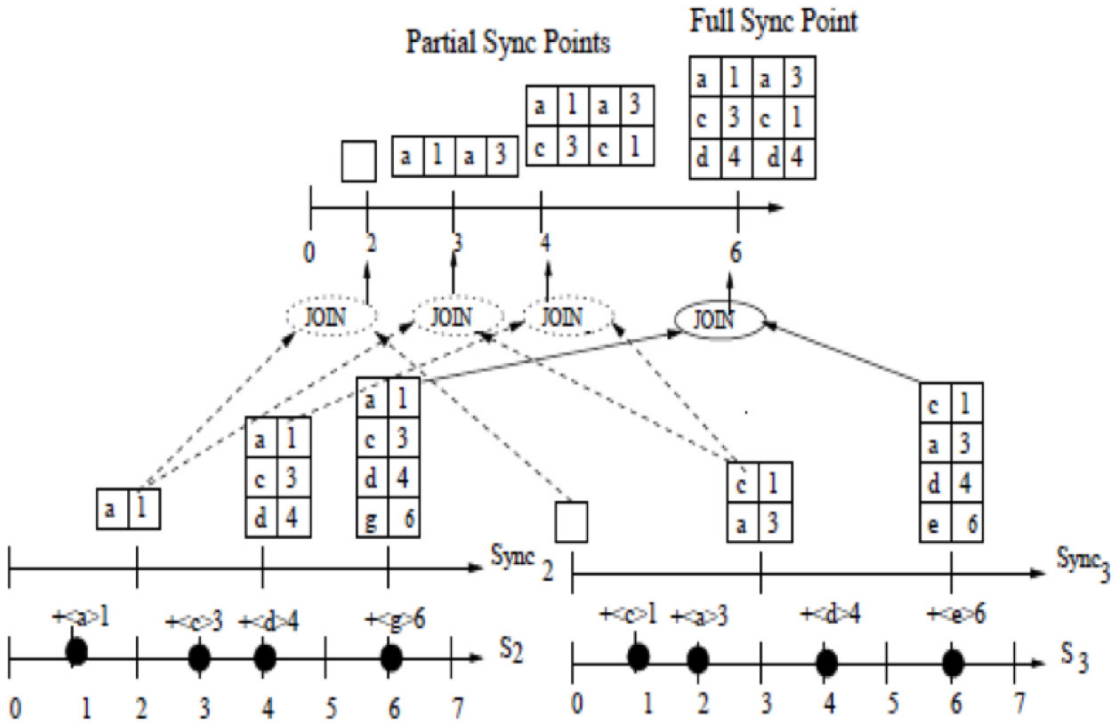

Fig. (1). The example of SyncSQL/Nile window synchronization.

ple of 2 or only multiple of 3 are partial synchronization points, while those are multiple of both 2 and 3 are full synchronization points, which can only be computed at synchronization points.

While window synchronization in SyncSQL/Nile only applies to the merger of MDS with same start points, it cannot synchronize data streams produced by transactions. Data streams produced by transactions generally use the submit time of transaction instead of logical time as the time domain. If two data streams S1 and S2 adopt transaction submit time as their common time domain, and meanwhile S2 always occurs later than S1, then there will always be a time delay between window queries defined on S1 and S2. And the time delay can be eliminated neither by window size nor by its slide while merging MDS.

This paper proposed a join semantic model and corresponding join algorithms for slide window based on matching window-id. The Differences between data streams in properties including time domain, start point, slide and window sizes will be concealed by the window-id. This paper is organized as follows: section 2 describes the semantic model of window join, and advocates a join semantic model for matching window-ids; section 3 introduces the data structure and window maintaining methods used by this semantic model; section 4 introduces the join algorithm and improved data structure of slide window and section 5 is about their theoretical and experimental analysis.

2. THE SEMANTIC MODELS OF THE JOIN OF SLIDE WINDOW

2.1. The Basic Model of the Join of Data Stream.

The join result of the data stream that do not use windows can be seen as the join view [14] of the append-only package. And the join process of data stream S1 and S2 can be seen as the join view of append-only package S1 and S2 which updates and maintains the package view when the package S1 or S2 is updating. Package S1 or S2 only supports insert operation, so its join relationship is monotonic and the one which time is larger of two tuples is used as the time of join result and the package join view is append-only. As a result the join of data stream defined as append-only package is monotonic, in accordance with the nature of data stream.

The semantic model requires that the merger status always be saved in memory, namely the memory saves all the input tuples of the data stream, and when a new tuple arrives, it joins the merger status of its data stream and meanwhile probes other data streams and produces the output based on the exploration result.

2.2. The Join Model of Slide Window

The basic model of data-stream join is unrealizable for unboundedly growing data and limited memory, because memory overflow will eventually result in error in data stream processing system. Therefore, the merger of data

stream is converted to the merger of recently arrived data, namely the sliding-window join semantics. According to the definition of the window, the sliding-window join can be classified as the time-based model and the tuple-based model [15]. Let the window size of data stream S is w units, and the time-based slide window join requires that the data stream it joins must join the tuple which arrives within w time; tuple-based slide window join requires that the data stream joins it must join the latest k tuples. If data stream S1 and S2 defines the time-based equi-join $S_1[w_1] \rtimes S_2[w_2]$ of the slide window, it means: for any $s_1 \in S_1$, $s_2 \in S_2$, there's $s_1 \cdot A = s_2 \cdot A$ on attribute A And when $s_1 \in S_1[w]$, at time point $s_2 \cdot t$, there's $s_1 \cdot t \in [s_2 \cdot t - w_1, s_2 \cdot t]$ Or when $s_2 \in S_2[w]$, at time point $s_1 \cdot t$, there's $s_2 \cdot t \in [s_1 \cdot t - w_2, s_1 \cdot t]$, in which w_1 and w_2 are the window sizes of S1 and S2. The merger processing of slide window will clear the data that isn't in the window in real time so as to avoid memory overflow, and to guarantee that future data will not merge with them. This will not affect the join semantics.

Since the slide window join model cannot be backtracked, i.e. new tuples will not compute the expired windows, it resolved the feasibility of MDS calculation, but these semantics don't concern the heterogeneity of the data streams. Document [16] considered the merger of heterogeneous data streams, but it mainly concentrates on the heterogeneity of the structure and the model of data streams not that of the slide window. In this paper, the heterogeneity of data-stream slide window is summarized as follows:

(1) Different time domain. Some data streams use the transaction time, some use data reception time and some use logical time;

(2) Different start points of the data streams;

(3) Different window types. Some use logic window, and some use physical window;

(4) Different units and size of the slide;

In systems like CQL/STREAM and TelegraphCQ, the join of sliding-window over MDS requires that the merging data streams employ exactly the same window mechanism and time system. As for SyncSQL/Nile system, it realizes synchronization according to different slides, or directly adopt approximate calculation [17, 18] to ignore the existence of heterogeneity in the slide window.

In this paper, the windows of each data stream are defined and identified independently, transforming the join conditions of multiple-data slide window into the respective definitions of their window-ids. For example, equi-join $S_1[w_1] \rtimes S_2[w_2]$ on data stream S1 and S2 is defined as: for any $s_1 \in S_1, s_2 \in S_2$, there's $s_1 \cdot A = s_2 \cdot A$ and $S_1 \cdot WID = S_2 \cdot WID$ Or $S_1 \cdot WID = S_2 \cdot WID \bmod k$ on property A. Window-id separates the definition of data-stream window from the join conditions to solve join semantic problems caused by the heterogeneity of slide window.

3. THE STORAGE AND MAINTENANCE OF SLIDE WINDOW

Let the size and slide of the slide window are expressed by the same unit; the slide window consists of several sub-windows; take the greatest common divisor of window-size and window-slide as the size of every sub-window, namely the size of the sub-window is $m = GCD(size, slide)$, then:

Window size $size = n \times size(sub\_window)$; the window size is n times that of the sub-window;

Window-slide $slide = k \times size(sub\_window)$; the slide is k times that of the sub-window.

The sub-window sequence is stored in a one-dimensional array of n bits, and in order to facilitate the management of time-type windows, the tuple of the sub-windows employs list structure. When each sub-window reaches its window-id wid, if the window-id is new, it joins the FIFO queue, and sets the pointer between sub-window array on the one side and sub-window elements and window id wid on the other side.

3.1. The Creation of Slide Windows

If window-id wid starts from 0, then the CN th sub-window to arrive belongs to the data stream window $w = \lceil \frac{CN-n}{k} \rceil$. If $CN > w \cdot k + n$, it also belongs to other windows, i.e. the wid range of the CN th sub-window is:

$$w \le wid \le w + \lceil \frac{CN - w \cdot k - 1}{k} \rceil$$

Example: let the slide window on data stream have three sub-windows and its slide is one sub-window, then its data structure is shown in Fig. (**2**).

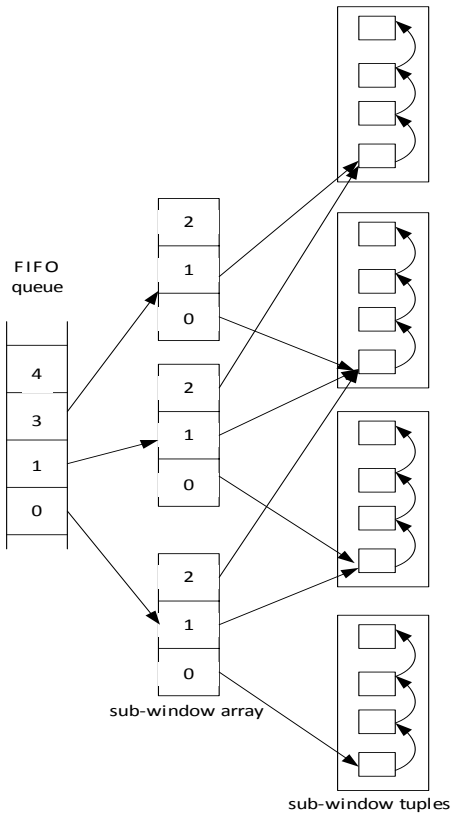

Fig. (2). The data structure of the data stream window.

The following is the algorithm of creating the data structure of the slide window:

Algorithm1: Create data structure of slide window (CDS-SW)

INPUT: The newly arrived sub-window sub_w; CN ,the count value of the sub-window(starts from 1); the window queue $list$; WCN ,the count value of sub-windows; The size of the window $size = n$ sub-windows; the slide of the window $slide = k$ sub-windows;

OUTPUT: The storage structure of slide window, the window queue $list$ and WCN ,the count value of sub-windows.

METHOD:

BEGIN

Step 1: Compute the least value of the sub-window $w = \lceil \frac{CN-n}{k} \rceil$ and the number of slide windows it belongs to $l = \lceil \frac{CN-wk-1}{k} \rceil$ according to its count value CN of sub-window.

Step 2: Set the window-id $wid = w$;

Step 3: For i = 0 to l do

BEGIN

Step 3.1: Scan FIFO queue $list$, IF wid is already in the queue

THEN

Compute the array index at the wid th sub-window $j = \text{mod}((CN - wid \cdot k - 1), n)$;

Add 1 to the number of sub-windows, i.e., $list[wid].WCN = list[wid].WCN + 1$;

ELSE

wid join the slide window queue $list$;

Initialize the pointer array of sub-windows $array[0..n-1] \rightarrow null$, $list[wid] \rightarrow array[0]$;

Compute the array index at the wid th sub-window $j = \text{mod}((CN - wid \cdot k - 1), n)$.

Initialize the number of sub-windows $list[wid].WCN = 1$;

ENDIF;

Step 3.2: The array pointer points to the CN th sub-window $list[wid].array[i] \rightarrow sub\_w$;

Step3.3: Add 1 to the window-id, i.e. $wid = wid + 1$;

ENDFOR

END

Algorithm 1: Create the data structure algorithm of the slide window

3.2. The Maintenance of Slide Windows

Whenever a sub-window arrives, the sub-window will be added into the data structure of the data stream, and meanwhile join the corresponding window of its join object and output the result into the buffer. When the output window expires, it will output the join result of the entire window. The join process of sub-windows uses SHJ algorithm, as shown in Fig. (**3**). Both the input and the output of sliding-window join are windows.

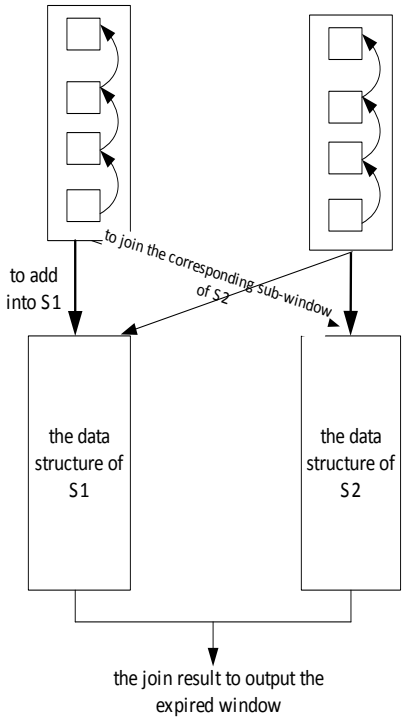

Fig. (3). The arrival of the new sub-window.

There are two conditions to determine whether the window expires: if the count value of the current window reaches n and the matched window of the join object reaches the count value of the sub-window, it expires, the data pointer and queue pointer of the sub-window will be deleted, as shown in Fig. (**4**).

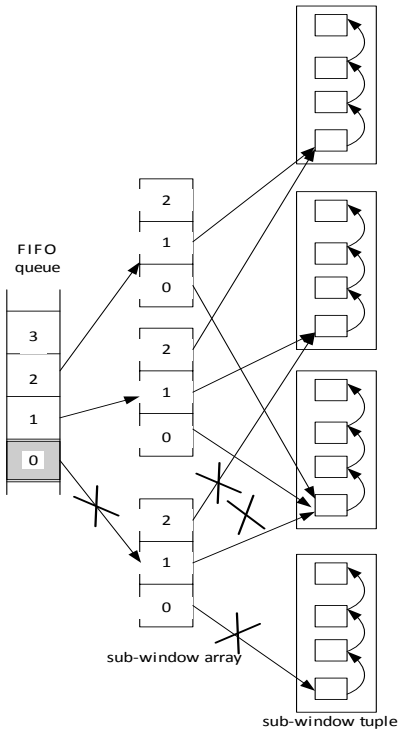

Fig. (4). To delete the expired window.

Here's the algorithm of maintaining slide window:

Algorithm 2: Delete expired window of data stream (DEW)

INPUT: The data structure of the sliding-window of data stream S1 and S2

OUTPUT: The data structure of the sliding-window of data stream S1

METHOD:

Step 1: Initialize $old1 = \min(wid\ of\ list)\ of$ S1, the window-id that matches $old1$ and S2 $old2 = wid\ of$ S2 $join$ S1

Step 2: IF $list[old1] \cdot WCN = n1$ and $list[old2] \cdot WCN = n2$ // $n1$ and $n2$ are the total number of sub-windows of stream S1 and S2

THEN FOR l=1 TO n1 DO

Disconnect all pointers of $array[l]$;

Disconnect the pointer of $list[old1]$ that points to $array[l]$;

Delete $old1$ from queue $list$;

ENDIF

END

Algorithm 2: The algorithm of the maintenance of slide window

4. THE JOIN ALGORITHM OF SLIDE WINDOWS THAT BASED ON WINDOW-ID

The join process of the slide window on data stream is shown in Fig. (**3**). In this SHJ, the newly arrived sub-window will be added into the queue of data stream windows through CDS-SW algorithm and then set a pointer. After that it will search for the window in the join object based on matching conditions and do join computing with all its sub-windows. The join computing process uses nested loop join algorithm.

However, a sub-window may belong to more than one data stream slide windows. And there will be a large number of repeated nesting process, if the new sub-window joins all those sub-windows of its matched window, resulting in a waste of computing resources. If the sub-window belongs to more than one slide windows, then these windows must be next to each other. In this case, it can be expressed by the equivalence relation and the forward pointer of the sub-window array. Fig. (**5**) shows us the data structure of slide windows which added equivalence relation based on Fig. (**2**).

4.1. Improve the Data Structure of Slide Window

The data structure created in Fig. (**5**) is based on improved CDS-SW.

Algorithm 3: Improved CDS-SW (I-CDS-SW)

INPUT: The newly arrived sub-window sub_w; CN, the count value of the sub-window (starts from 1); window queue $list$; WCN, the count value of sub-windows of slide windows; The size of the window $size = n$ sub-windows; The slide of the window $slide = k$ sub-windows;

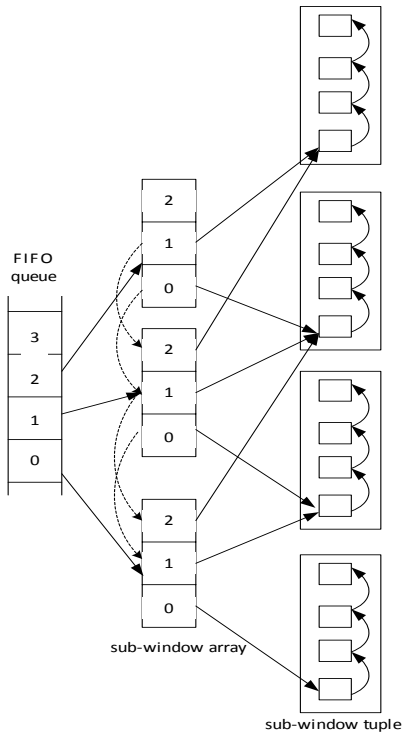

Fig. (5). The equivalence relation between the adjacent sub-windows.

OUTPUT: The storage structure of the slide window, the window queue $list$ and WCN, the count value of sub-windows of the slide window

METHOD:

Step 1: Compute the minimum $w = \lceil \frac{CN-n}{k} \rceil$ of the slide windows which the sub-window belongs to and the number $l = \lceil \frac{CN-wk-1}{k} \rceil$ of all its belonging sliding-windows in accordance with the count value CN.

Step 2: Set the window-id $wid = w$;

Step 3: Initialize the minimal slide window and the pointer of the equivalent sub-window.

Step 3.1: Initialize the head pointer of the equivalent sub-window $hq \rightarrow null$;

Step 3.2: Scan the window queue $list$, IF w in $list$

THEN

Compute the array index of the w th sub-window $i = \mathrm{mod}((CN - wid \cdot k - 1), n)$;

Modify the count value of the sub-window $list[w] \cdot WCN = list[w] \cdot WCN + 1$;

ELSE

Press w into the slide window queue $list$;

Initialize the pointer array of sub-window $array[0..n] \rightarrow null$, $list[wid] \rightarrow array[0]$;

Compute the array index at wid th sub-window $i = \mathrm{mod}((CN - wid \cdot k - 1), n)$;

Initialize the count value of the sub-windows $list[wid] \cdot WCN = 1$;

END IF;

Step 3.3: The pointer of the sub-window of the current slide window points to CN th sub-window $list[wid] \cdot array[i] \rightarrow sub\_w$;

Step 3.4: The current array element points to the head pointer of the equivalent sub-window $list[wid] \cdot array[i] \cdot q \rightarrow hq$.

Step 3.5: The head pointer points to the current array element $hq \rightarrow list[wid] \cdot array[i] \cdot q$;

Step 4: FOR i = 1 TO l DO / / add the current sub-window to other slide windows

Step 4.1: Modify the count value of the current slide window $wid = wid + 1$;

Step 4.2: Scan FIFO queue $list$, IF wid in $list$,

THEN

Find the array index of the wid th sub-windows $i = \mathrm{mod}((CN - wid \cdot k - 1), n)$;

Modify the count value of the sub-windows $list[wid] \cdot WCN = list[wid] \cdot WCN + 1$;

Else

Press wid into the slide window queue $list[wid] \cdot WCN = 1$;

Initialize the pointer array of the sub-window $array[0..n] \cdot sw \rightarrow null$, $list[wid] \rightarrow array[0]$;

Compute the array index at the wid th sub-window $i = \mathrm{mod}((CN - wid \cdot k - 1), n)$;

Initialize the count value of sub-windows $list[wid] \cdot WCN = 1$;

END IF

Step 4.3: The array pointer points to the CN th sub-window $list[wid] \cdot array[i] \rightarrow sub\_w$

Step 4.4: The current array element points to the head pointer of the equivalent sub-window $list[wid] \cdot array[i] \cdot q \rightarrow hp$

Step 4.5: The head pointer points to the current array element $hp \rightarrow list[wid] \cdot array[i] \cdot q$

ENDFOR

END

Algorithm 3: Improved algorithm of setting the data structure of slide window

4.2. Join Algorithm of Slide Window

Let the sizes of the windows of data stream S1 and S2 are $n1$ sub-windows and $n2$ sub-windows respectively, $list\_R$ is the matched window pair, then the resultant array is at most composed of $n1 \times n2$ sub-windows and its array elements are the array indexes of each data stream.

Example: let the window size of data stream S1 is $n1 = 3$ sub-windows and the slide is $k1 = 1$ sub-windows; The window size of data stream S2 is $n2 = 2$ sub-windows with a slide of $k2 = 1$ sub-windows, just as shown in Fig. (**6**); every pair of marched slide window consists of $n1 \times n2 = 6$ sub-windows, and the number of the tuples of each sub-windows are determined in the light of the join and matching conditions. The data structure of the join result is shown in Fig. (**7**).

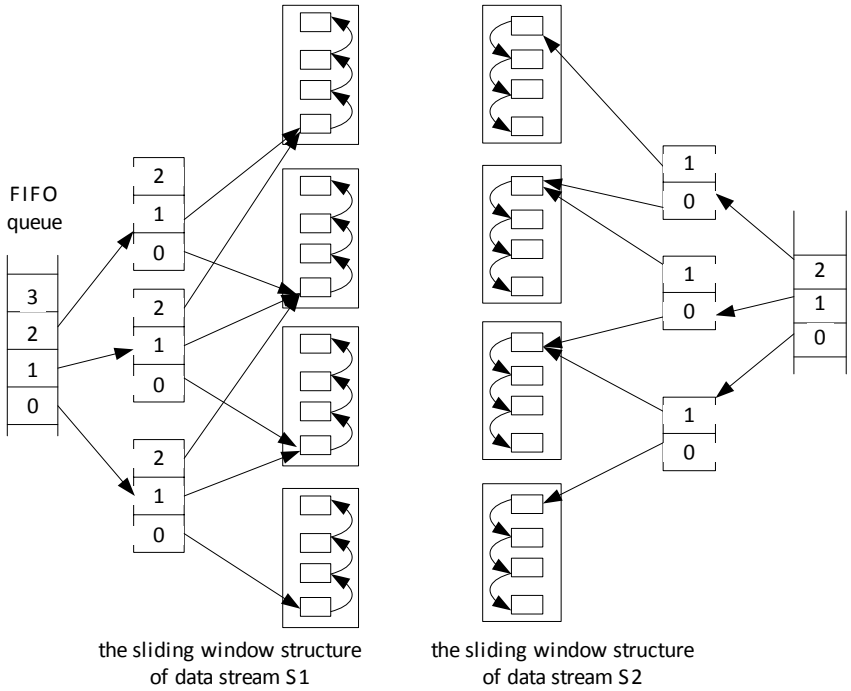

Fig. (6). The data structure of those sliding window which have 3 sub-windows and 2 sub-windows.

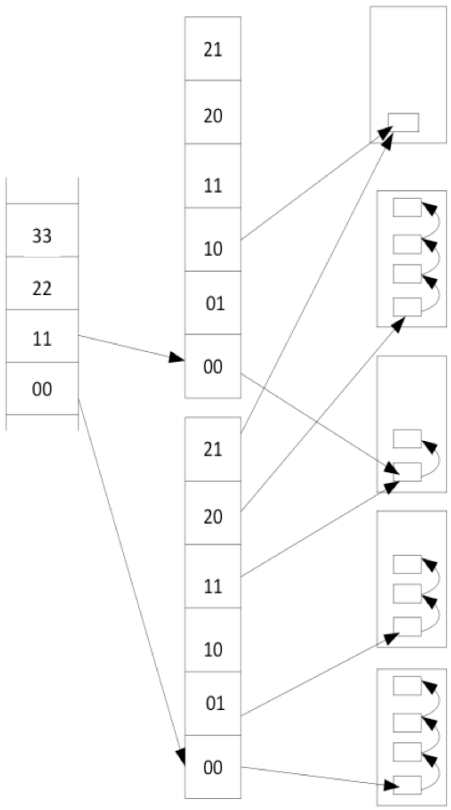

Fig. (7). The data structure of the join result.

Here is the join algorithm of slide window:

Algorithm 4: Join of slide window (JSW)

INPUT: The storage structure of slide window in data stream S1 and S2; the window match condition is that the *wid* is equal

OUTPUT: The window queue of the join result $list\_R$.

METHOD: While true do

Step 1: Let s_win ,the sub-window of S1 arrives, call algorithm 3 to compute all the slide window that the sub-window belongs to($wid_l..wid_m$), and the array index of the sub-window ai , in which i take a value from （$0..n1-1$）.

Step 2: For each wid_k of ($wid_l..wid_m$) in which $k = l..m$ Step 2.1: IF wid_k is not in $list_2$ RETURN; / / determine whether the matched window is already in the list.

ELSE IF (wid_k, wid_k) is not in $list\_R$ / / The matched window is in the queue; determine whether it has already finished joining

THEN (wid_k, wid_k) join the result queue $list\_R$; Initialize the array $array[0..(n1\times n2-1)]$

Step 2.2: Call equiv-window (wid_k, i) / / Find all the equivalent sub-windows ew_list1 of (wid_k, ai)

Step 2.3: For each i of $list_2[wid_k]\cdot array[0..n-1]$ is not null call equiv-window (wid_k, i) / / Find all the equivalent sub-windows of the join object ew_list2

Set flag Flag = 0 / / used to flag whether the join result of the sub-windows has been computed.

WHILE (ew_list1 is not null) and (ew_list1 is not null) DO / / corresponds to the join of sub-window

BEGIN

IF $ew\_list1\cdot wid_k > ew\_list2\cdot wid_k$ THEN dequeue the current element of ew_list1

ELSE IF $ew\_list1\cdot wid_k < ew\_list2\cdot wid_k$ THEN dequeue the current element of ew_list2

ELSE IF (flag) THEN

$list[wid_k\ wid_k]\cdot array[ew\_list1\cdot i\ ew\_list2\cdot i]\rightarrow R\_S\_W$

ELSE BEGIN

Find R_S_W , the join result of $ew\_list1[wid_k]\cdot array[i]$ and $ew\_list2[wid_k]\cdot array[i]$

$list[wid_k\ wid_k]\cdot array[ew\_list1\cdot i\ ew\_list2\cdot i]\rightarrow R\_S\_W$

/ / Set the result pointer

END IF

Flag = 1; / / Modify the flag

END WHILE

ENDFOR

END;

PROCEDURE equiv-window (wid_k, i) / / Find all the equivalent sub-window ew_list1 of(wid_k, i)

BEGIN

Step 1: Add $(wid_k\ i)$ to list ew_list / / ew_list use a the FIFO structure

Step 2: $p=(list[wid_k]\cdot array[i]\rightarrow q)$; / / p is the pointer of the sub-window(wid_k, i)'s equivalent chain

Step 3: WHILE $p\neq null$ DO

BEGIN

$p=(p\rightarrow next)$; / / p point to the adjacent node

$wid'=wid_k-1$; / / The window-id of the slide window which pointer p belongs to

$i'=i+1$; / / The pointer index of the sub-window that pointer p points to at the corresponding slide window

Add $(wid'\ i')$ to ew_list;

END WHILE;

Step 4: Return(ew_list);

END;

Algorithm 4: The join algorithm of slide window based on matching window-ids.

The maintaining process for join results is similar to that for the slide window. If the two matched windows both expires, then the join window expires too, the algorithm will delete all the pointers in window queue and join window, as shown in Fig. (**8**).

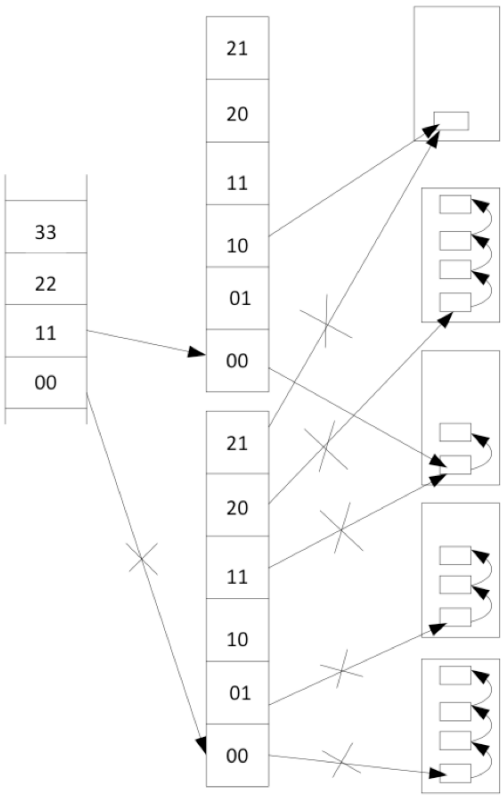

Fig. (8). The process of the expired windows of the join result.

5. PERFORMANCE ANALYSIS

5.1. Proving the Correctness of the Algorithm

We can see from algorithm 3 that equivalent sub-window is essentially the same sub-window that divided into successive slide windows, thus if a sub-window exists, all its equivalent sub-windows exists. Let $\exists(s1\_w, s2\_w)$: $s1\_w \in S1$, $s2\_w \in S2$, in which $s1\_w$, $s2\_w$ are the sub-windows of transaction data stream S1 and S2 respectively, and $(s1\_w, s2\_w)$ satisfy the window-join conditions, then $s1\_w$ and $s2\_w$ must have the same window-id wid. According to algorithm 4, these two sub-windows either directly carry on the join calculation or point to the join results of their equivalent sub-windows; i.e. the algorithm can compute the join of all matched sub-windows.

Let the result of $s1\_w$ and $s2\_w$ have been computed several times, then $s1\_w$ and $s2\_w$ must belong to several different equivalence classes of the sub-window, i.e., it's impossible that a sub-window can appear many times in the data structure of slide windows, it is impossible in the data structure of the slide windows(one hash function only have one hash value), thus the algorithm will compute the matched sub-window only once. Through the above analysis, we can see that this algorithm can correctly compute the join of slide window.

5.2. Experimental Analysis

In this section, two basic experiments are used to illustrate that the algorithm can be applied to join those different types windows and compare the computing time of those different data structure. The experimental platforms are pentium dual-core CPU@2.6GHz and C language. First, carry on join calculation on the streams of two windows with different types, in which data stream S1 is defined as a physical window with a window size of 5000 tuples and a slide of 1000 tuples, and S2 as a logic window with a window size of 5 minutes and a slide of 1 minute. To simplify the experiment, the data-stream tuples are produced by loop structure, random function and time-delay function, and all the tuples are integer, ignoring factors like network instability.

According to I-CDS-SW, the maintenance algorithm of data stream, every window of S1 and S2 consists of 5 windows, and has a slide of 1 sub-window; S2 produces 120 tuples per second averagely; the maintenance of the windows of every stream is independent; there's a 10-minute time delay between the two data streams. Since the join algorithm JSW only judges on the join conditions and the window-id, the time cost of this algorithm can be expressed as:

The total computing time = time of maintaining data structure + query time+ computing time + delay time

The time of reading the disk file when there's not enough memory is neglected here. The output result of different-type windows is shown in Fig. (**9**). This figure shows that the join output is irrelevant to window type; the output is affected by the time delay of the two data streams and the output result only relates to the number of sub-windows and the data in sub-windows.

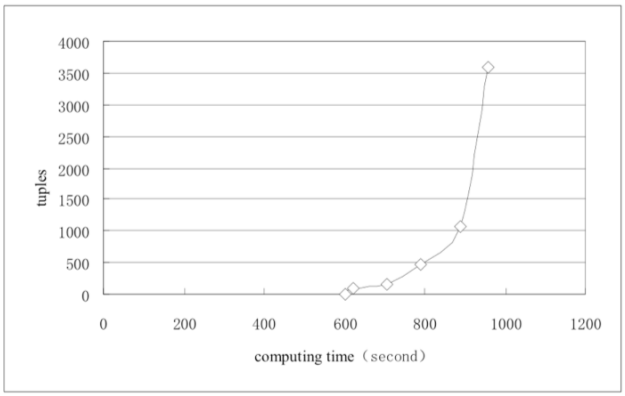

Fig. (9). The join output of those different types windows.

Besides, the join of the data streams of two physical windows contrasted the influence of the data structure of CDS-SW on join calculation with that of I-CDS-SW.Through the comparison result shown in Fig. (**10**), it uses the equivalence relation when calculating, reduced the time for query and computing so that the total computing time is shortened.

The join semantics based on window-id is an extension to the current join semantics of slide window. This experiment proved the practicability of the join algorithm and relevant data structure, illustrating the fact that using window-id match of slide window is a viable way to synchronize MDS.

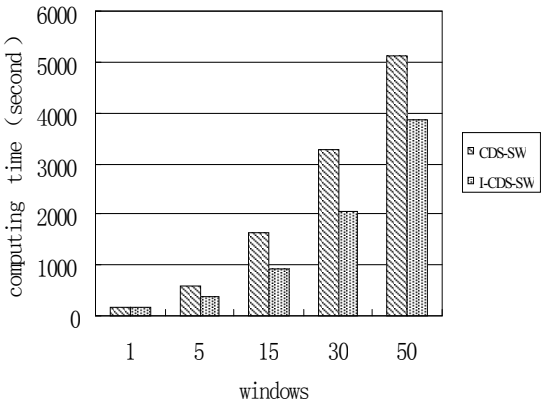

Fig. (10). The comparison of computing time of those two kinds of data structures.

CONCLUSION

The join semantic model of heterogenous data stream is an important and a complex problem to the join calculation of MDS, which is significant to real time processing and data integration. In this paper we advocated a join semantic model based on the match of window-id by focusing on the differences in time system, window-size and slide of the data stream. And according to this model, the data structure and respective join algorithms were provided. To reduce the repeated calculation of slide window, the equivalence relation of sub-windows was used to improve the data structure of slide window. The semantic model also solved the heterogenetity problem in the join calculation of MDS, but it still cannot deal with the situation where the window-size and window-slide are expressed by different units; the join algorithm didn't cover the possibility that the data is too big to be stored in memory, all of which need further research.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflict of interest.

ACKNOWLEDGEMENTS

This work was supported in part by Technology Research Project of the Ministry of Public Security Grant No. 2014JSYJB048, State Key Laboratory of Software Engineering Grant No.SKLSE2012-09-37 and National Natural Science Foundation of China Grant No.61272413. Zou Xianxia is the corresponding author.

REFERENCES

[1] T. Urhan and M. J. Franklin. "*XJoin: Getting Fast Answers From Slow and Burst Networks*," Technical Report CS-TR-3994, UMIACS-TR-99-13, Computer Science Department, University of Maryland, 1999.

[2] A. N. Wilschut, and P. M. G. Apers, "Dataflow query execution in a parallel main-memory environment," In: *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, PDIS, 1991, pp. 68-77.

[3] M. F. Mokbel, M. Lu, W. G. Aref, and H.M. Join, "A non-blocking join algorithm for producing fast and early join results," In: *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, 2004, pp. 251-263.

[4] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis, "RPJ: producing fast join results on streams through rate-based optimization", In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, 2005, pp. 371-382.

[5] M.A. Bornea, V. Vassalos, Y. Kotidis, and A. Deligiannakis, "Double index nested-loop reactive join for result rate optimization,". In: *Proceedings of the International Conference on Data Engineering (ICDE)*, 2009, pp. 481-492.

[6] J.-P. Dittrich, B. Seeger, D. S. Taylor and P. Widmayer, "Progressive merge join: a generic and non-blocking sort-based join algorithm," In: *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2002, pp. 299-310.

[7] S. Chandrasekaran, and M. J. Franklin, "Streaming queries over streaming data," In: *Proceedings of the VLDB Conference*, 2002, pp. 203-214.

[8] S. Madden, M. Shah, J. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," In: *Proceedings of the SIGMOD Conference*, 2002, pp. 49-60.

[9] J. Kang, J.F. Naughton, and S.D. Viglas, "Evaluating window joins over unbounded streams," In: *Proceedings of the ICDE Conference*, 2003, pp. 341-352.

[10] L. Golab, S. Garg, and M.T. Ozsu, "On indexing sliding windows over on_line data streams," In: *Proceeding of the 9th International Conference on Extending Database Technology (EDBT)*, 2004, pp. 712-729.

[11] A. Arasu, S. Babu and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 2006, no. 15, pp. 121-142, 2006.

[12] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A, Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S.R. Madden, F. Reiss, and M.A. Shah, "TelegraphCQ: an architectural status report," *IEEE Data Engineering Bulletin*, vol. 26, no. 1, pp. 11-18 March 2003.

[13] T. M. Ghanem, and A. K. Elmagarmid, "Per-AKE LARSON, Walid G. Aref. supporting views in data stream management systems," *ACM Transactins on Database Systems*, vol. 35, no. 1, pp. 1-47, 2010.

[14] A. Chakraborty, and A. Singh, "*A Disk-based, Adaptive Approach to Memory-Limited Computation of Exact Results for Windowed Stream Joins*," Department of Electrical & Computer Engineering, Technical Report UW-ECE #2009-09, 2009.

[15] J. Xie and J. Yang. "A survey of join processing in data streams," *Data Streams*, vol. 31, pp. 209-236, 2007.

[16] L. P. Danh, J. X. Parreira, and H. Manfre, "Linked stream data processing," In: *Proceedings of the 3rd International Conference on Web Intelligence, Mining and Semantics*, vol. 31, 2012, pp. 245-289.

[17] A. Das, J. Gehrke, and M. Reidewald, "Approximate join processing over data streams," In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 40-51.

[18] T. Jens, and R. Mueller, "How soccer players would do stream joins," In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011, pp. 625-636.

Received: September 16, 2014 Revised: December 23, 2014 Accepted: December 31, 2014