Scalable Distributed Stream Join Processing

Qian Lin[†] Beng Chin Ooi[†] Zhengkui Wang[†] Cui Yu[§]

[†]School of Computing, National University of Singapore
[§]Department of Computer Science and Software Engineering, Monmouth University

[†]{linqian, ooibc, wangzhengkui}@comp.nus.edu.sg, [§]cyu@monmouth.edu

ABSTRACT

Efficient and scalable stream joins play an important role in performing real-time analytics for many cloud applications. However, like in conventional database processing, online theta-joins over data streams are computationally expensive and moreover, being memory-based processing, they impose high memory requirement on the system. In this paper, we propose a novel stream join model, called *join-biclique*, which organizes a large cluster as a complete bipartite graph. Join-biclique has several strengths over state-of-the-art techniques, including *memory-efficiency*, *elasticity* and *scalability*. These features are essential for building efficient and scalable streaming systems. Based on join-biclique, we develop a scalable distributed stream join system, *BiStream*, over a large-scale commodity cluster. Specifically, BiStream is designed to support efficient full-history joins, window-based joins and online data aggregation. BiStream also supports adaptive resource management to dynamically scale out and down the system according to its application workloads. We provide both theoretical cost analysis and extensive experimental evaluations to evaluate the efficiency, elasticity and scalability of BiStream.

1. INTRODUCTION

In recent years, data have been rapidly generated in many applications as data streams, such as stock trading, mobile and networked information management systems. It is increasingly challenging and critical to provide efficient real-time analytics for such applications. Data aggregation is one of the most important and widely used operations in analytics, e.g., online analytical processing in relational databases, as it provides a high level view by summarizing the data using group-by and aggregate functions such as `count`, `sum` and `avg`. Most data aggregations are applied over the join results of two data streams with respect to a given join predicate; hence, efficient progressive stream join processing [15, 6] is essential and important to data stream systems.

There has been a lot of research on data stream join processing. Existing centralized algorithms [39, 18, 17] that were originally designed for a single server are not capable of handling the massive data stream workload. On the other hand, existing distributed and parallel stream join processing algorithms are mainly tailored for equi-join, which would not be efficient for high-selectivity joins such as the theta-join. Further, these methods mostly adopt various hash techniques for workload partitioning, which is sensitive to load distribution and inflexible to scaling out the system due to maintenance complexity.

In order to design an efficient distributed stream theta-join processing system, the following two requirements must be considered. First, in-memory processing is essential to provide efficient stream join and real-time analytics. Second, scalable stream processing is critical to support large-scale data stream applications. That is, a distributed stream join system has to be both memory-efficient and scalable. Among many of the related work, Join-matrix model [34] which was studied a decade ago has recently been revisited for supporting distributed join processing such as in MapReduce-like systems [28] and also stream applications [12]. Intuitively, it models a join operation between two input relations as a matrix, where each side of which corresponds to a relation as shown in Figure 1.a. Each processing unit (i.e., matrix cell) represents a potential join output result. However, the join-matrix model has three inherent weaknesses. First, it has high memory usage as the data need to be replicated and stored in an entire row or column. Second, it is not flexible to scale out/down since it has to add/release one entire row or column each time. Third, it does not scale well, because when a new processing unit to one row/column is added, the data in other units within the same column/row have to be replicated to the new unit. These weaknesses limit the memory-efficiency, elasticity and scalability of the systems that employ the join-matrix model.

In this paper, we propose *join-biclique*, a novel stream join model that is scalable and elastic with respect to the network size, and efficient and effective in terms of memory requirement and resource utilization. The core abstraction of join-biclique is to organize all the processing units in a cluster as a complete bipartite graph (a.k.a. biclique), where each side corresponds to a relation. Suppose R and S are the two input stream relations. Given $m + n$ processing units, m units on one side of the bipartite graph are for R and the other n units are for S. Figure 1.b shows one join-biclique example where m and n are 2 and 3, respectively. The result of $R \bowtie_\theta S$ can be obtained by evaluating each $R_i \bowtie_\theta S_j$ where R_i is the i^{th} partition of R with $i \in \{1, ..., m\}$, and S_j is the j^{th} partition of S with $j \in \{1, ..., n\}$. The

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2746485.

join-biclique model has two important features. First, each relation is stored in its corresponding units without data replicas, which enables the model to be memory-efficient. Second, the processing units are independent of each other, which facilitates scalability and elasticity.

We next propose a distributed stream join system, *BiStream*, based on the join-biclique model. BiStream adopts a two-level architecture, including *router* and *joiner*. The router involves multiple shufflers and dispatchers that are in charge of ingesting the input streams and routing them to the corresponding processing units. The joiner consists of all the processing units which are responsible for data storing and join processing. BiStream adopts a simplified topological design of join-biclique where each processing unit only communicates with the router instead of among themselves to reduce the degree of inter-unit connectivity. Based on this design principle, the basic stream join processing flow works as follows: when one input tuple $r \in R$ (resp. $s \in S$) arrives at the router, it is routed to one of the processing units in R (resp. S) to store, and meanwhile it is sent to join the tuples stored in all the processing units in S (resp. R). As a consequence, r (resp. s) can be directly discarded from the units of S (resp. R) after join processing. This greatly lowers the memory requirement and hence improves the system capacity to efficiently support in-memory real-time analytics.

In summary, we make the following contributions:

- We propose a novel memory-efficient stream join model, *join-biclique*, for joining big data streams.
- Based on join-biclique, we design and develop a scalable stream join system, *BiStream*, for a large-scale commodity cluster. We also discuss how to efficiently support full-history and window-based stream joins, and online data aggregation.
- We propose an adaptive resource management for BiStream to dynamically scale out or down the system based on workload. The adaptive resource management allows the system to automatically increase and release the computation resources to minimize the system running cost.
- We theoretically analyze the proposed techniques and also conduct extensive experimental evaluation over a cluster of 40 nodes. The evaluation results confirm that BiStream is memory-efficient, elastic and scalable.

The remainder of this paper is organized as follows. We summarize the related work of stream join processing in Section 2. We define the join-biclique model and introduce BiStream in Section 3, following which we analyze BiStream in Section 4. Section 5 presents how BiStream supports adaptive stream join processing in the cloud. Evaluation of BiStream is demonstrated in Section 6 and finally, we conclude the paper in Section 7.

2. RELATED WORK

A great deal of research on distributed join processing has been conducted. However, the works [8, 35] have been proposed mainly for non-streaming scenarios. They cannot be directly applied to the streaming systems. For stream join processing, much research has also been devoted mainly for a stand-alone environment [39, 18, 17]. To name a few, symmetric hash join [39] is the earliest proposal for joining finite data streams. While reading in tuples of each relation in turn, it progressively constructs two hash tables (one for each relation) in memory and produces join results by probing them mutually. Online nested-loops join [18] is the first non-blocking join algorithm to progressively report the estimates of the join-based aggregate query results in real-time. Ripple join [17] and its variants [27, 20, 10] further generalize such processing towards online aggregation. However, all these algorithms rely on a centralized maintenance of the entire join states and hence, they are not easy to scale.

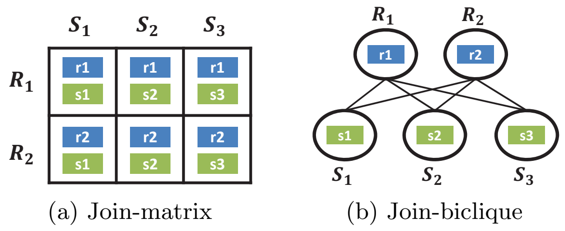


Figure 1: Stream join models.

Further, several stream join algorithms have been proposed for the multi-core and main-memory environment [36, 14, 21, 29]. For instance, CellJoin [14] parallelizes the execution of stream joins on multi-core Cell processors, but its efficiency highly relies on the hardware parallelism of Cell processors. Handshake join [36, 31] organizes the processing units as a doubly linked list. The two joining streams are fed into the system in the opposite directions, and the join condition for each pair of joining tuples is evaluated once they meet in some unit. However, this model is susceptible to node failure and message loss if it is extended to the distributed environment.

Recently, much research has been devoted to stream join processing in a distributed environment with a cluster of commodity machines. Photon [4] was specifically designed for joining data streams of web search queries and user clicks on advertisements in Google. It relies on a centralized coordinator to search for the join tuples through key-value matching, but it is not designed to support general theta-join predicates. D-Streams [41] breaks continuous streams into discrete units and processes them as ordered batch jobs on Spark [40]. Such batch processing for stream joins can only provide approximate results, as a few target tuple pairs in separated batches may miss each other for join operation. TimeStream [30] exploits the dependencies of tuples to perform the joins. PSP [38] transforms a macro join operator into a series of smaller sub-operators by time-slicing of the join states, and the join processing is distributed to these sub-operators in a ring architecture. However, both TimeStream and PSP incur high communication overhead to maintain the dependencies or synchronize the distributed join states. For design and optimization of distributed and in-memory data processing, please refer to [26, 42].

The join-matrix model was adopted in a recently proposed distributed join processing [12]. It organizes the processing units as a matrix, each side of which corresponds to the partitions of one relation. Each matrix cell represents a potential join output result. Figure 1.a shows an example of join-matrix model with 2 and 3 partitions for relations R and S respectively. For instance, when $r1 \in R$ arrives, it is shuffled to one partition of R (e.g., R_1) and replicated to all units in the row R_1; meanwhile, it joins with the tuples of S stored in these units. This model is able to handle theta-join in a parallel and decentralized manner. However, it suffers from high memory consumption as it has to replicate each

Table 1: List of notations.

Notations	Description
R, S	Stream relation
r, s	Stream tuple
R_i, S_j	i-th and j-th partition of R and S
G_R, G_S	Universal set of partitions of R (S)
m, n	Number of partitions of R (S)
$G_{R,k}$, $G_{S,l}$	Subgroup of partitions of R (S)
d, e	Number of subgroups of R (S)
ϖ	Size of the sliding window
$\mathcal{P}$	Archive period of the chained in-memory index
$\mathcal{D}$	Maximum network latency

input tuple to multiple units. Moreover, it is not easy to scale out or down as it has to maintain the matrix structure. To address these issues, our join-biclique model is designed to be memory-efficient and amenable to scaling.

In addition, prior work on stream partitioning [13, 7] was focused on designing approaches to partition a large amount of skewed input streams into different units in order to achieve a balanced load in the unit or a minimized stream redirection cost. They are mainly designed for unary operations such as selection, projection and aggregate functions, not trivially applicable to binary stateful operators like join. Differently, our work is primarily focused on designing a scalable stream join processing model that is effective in both stream partitioning and join operation.

3. THE JOIN-BICLIQUE MODEL

3.1 Definition

Table 1 lists the notations used throughout the rest of the paper. Join-biclique model is developed with the main objectives of reducing memory consumption and facilitating scalability. The formal definition of join-biclique is presented as follows:

Definition 1 (Join-Biclique Model). Given a cluster with $m+n$ processing units (or units when there is no ambiguity), join-biclique organizes all of these units as a complete bipartite graph (a.k.a. biclique). Suppose there are two relations R and S. Each side of the bipartite graph corresponds to one relation for storage. Specifically, data of relation R are partitioned and stored into one side of the bipartite graph with m units without replicas (i.e., $G_R = \{R_1, ..., R_m\}$). Similarly, data of relation S is partitioned into the other side with n units (i.e., $G_S = \{S_1, ..., S_n\}$). In the bipartite graph, there exists an edge (R_i, S_j) between units R_i and S_j where $i \in \{1, ..., m\}$ and $j \in \{1, ..., n\}$. Each edge (R_i, S_j) represents a potential join result produced by $R_i \bowtie_\theta S_j$.

Each edge in the join-biclique represents the join operation between two units of the opposite relations and the join-biclique is able to generate the Cartesian product of the join participants. As a result, the join-biclique model can support any join predicate. In a typical real deployment, one physical machine may host multiple processing units according to the machine configuration. That is, the units of R and S may only be logically separated.

3.2 BiStream System Design

Next, we present a distributed stream join system, *BiStream*, based on the join-biclique model. For the ease of explanation, we focus on the full-history joins (i.e., joining all tuples entered so far) to illustrate how the system works. Discussion on supporting the window-based joins is deferred to Section 3.3.

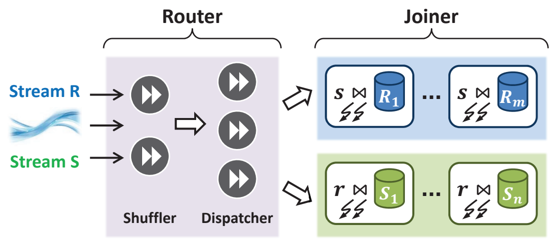


Figure 2: Overall architecture of BiStream.

3.2.1 System Architecture

BiStream is built on top of Storm [1], a distributed real-time computation platform. In order to appreciate BiStream, we shall first briefly introduce Storm terminologies. A Storm topology is a graph of *spouts* and *bolts*. Each spout acts as a stream source adapter, and each bolt consumes the input streams and possibly emits new streams. Spouts and bolts are connected with various *stream groupings* which define how streams' tuples are distributed among bolt tasks. Towards data stream system development on Storm, the system logic is decomposed into functionalities which are further implemented as spouts and bolts, and the dataflows are defined through stream grouping.

Figure 2 depicts the architecture of BiStream. BiStream consists of two functional components: *router* and *joiner*. The router is designed to ingest and route the input streams to the corresponding units for further processing, and the joiner is in charge of data storing and join processing for the data received from the router.

The BiStream topology follows the join-biclique model but reduces the degree of inter-unit connectivity. Conceptually, in a join-biclique, each unit needs to connect with all units of the other relation. Fortunately, this is not necessary, and BiStream avoids the communication among the units by separating the router and the joiner since the data routing is independent from the data store and join processing. That is, each unit only communicates with the router to receive the data to store or join without communicating among each other. Specifically, an incoming tuple (e.g., $r \in R$) is sent to one of the units R_i to store, and at the same time it is sent to all the units of S for join processing. After the join processing, r can be discarded from the units in S.

Router. Figure 2 illustrates 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. Meanwhile, the dispatcher is in charge of maintaining the statistics about how the data is stored among the units.

The main aim of separating shuffler and dispatcher in the router topology is to isolate the ingestion of incoming tuples and the maintenance of storage statistics. To trace the order and time of each incoming event, BiStream assigns one timestamp to each stream input event. This enables the system to always process and join the input events in the right order, especially for the time-based sliding-window joins (i.e.,

Algorithm 1: Control of ContRand routing.

```
    Input: tuple $t$
1:  Procedure Route($t$):
2:      $group \leftarrow$ GetSubgroup(key($t$), $t.rel$)
3:      $p \leftarrow$ a unit randomly chosen from $group$
4:      forward($t$, $p$)                              /* to store */
5:      $\overline{group} \leftarrow$ GetSubgroup(key($t$), opposite($t.rel$))
6:      foreach unit $q \in \overline{group}$ do
7:          forward($t$, $q$)                          /* to join */

    Input: value $v$, relation $rel$
8:  Function GetSubgroup($v$, $rel$):               /* for equi-join */
9:      $rel$ match
10:         $R$: return $G_{R,i}$ s.t. $v \in$ ValueRangeOf($G_{R,i}$)
11:         $S$: return $G_{S,j}$ s.t. $v \in$ ValueRangeOf($G_{S,j}$)
```

joining the tuples with a time constraint). A timestamp is assigned to each tuple based on the time when the tuple arrives at the shuffler in the system. It is important that the tuples arriving at the same time get the same timestamp. To guarantee that, we need to synchronize the timers in the router which is a costly process since the number of timers may be large. There is however a clear demarcation of roles and tasks; the shuffler is in charge of timestamp assignment while the dispatcher handling statistics collection and other routing decision tasks. With this design, the number of shuffler tasks can be greatly reduced, and it will effectively ease the synchronization among the timers.

Joiner. The joiner component consists of all the processing units. It is designed to process the streams from the router. Each unit runs a stateful bolt task that consumes the tuples from the router. Each task has two execution branches – data storing and join processing, whose runtime depends on whether the tuple belongs to the corresponding relation of the unit or from the other joining relation. To be specific, the data storing operation is triggered when one input tuple $r \in R$ arrives at its corresponding unit $R_i \in G_R$, where r will be stored. If one unit $R_i \in G_R$ receives a tuple $s \in S$ from the other relation, the join processing is conducted by joining s with all the tuples stored in R_i.

To support efficient join processing, BiStream adopts a variety of in-memory indexes to index the data, such as hash maps for equi-joins and balanced binary search trees for non equi-joins. More details of the indexing and join processing are described in Section 3.3.

3.2.2 Dataflow Control

The main objective of dataflow control is to balance the load in the system to improve the degree of processing parallelism. To facilitate load balancing, we design BiStream dataflow, which is illustrated in Figure 2. It is a two-stage process: (1) from the shuffler to the dispatcher, and (2) from the dispatcher to the joiner. For dataflow control in the first stage (1), BiStream adopts a random routing strategy, where tuples emitted by the shuffler are randomly distributed across dispatcher tasks such that each dispatcher task receives an equal number of tuples. The stream for random routing is implemented using the *shuffle stream grouping* in Storm. Being content-insensitive, the random routing strategy should be effective in automatically balancing the workload among the dispatcher tasks.

The second stage (2) involves two streams: the store stream and the join stream. The store stream is for routing each

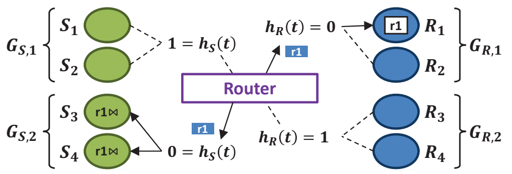


Figure 3: An example of ContRand routing.

tuple to a unit for storing, and the join stream is for sending the tuples to the proper units for join processing. Different routing strategies in these two streams can be implemented for different joins based on join selectivity.

For *high-selectivity* joins such as inequality joins, the random routing strategy should be adopted. High-selectivity joins may generate a large number of join results in most, if not all, of the units, while the random routing strategy helps to automatically balance the load among all the units of a relation and makes the system resilient to data skew. With the random routing strategy, $r \in R$ (resp. $s \in S$) is randomly shuffled to one unit R_i (resp. S_j) to store in a content-insensitive manner (i.e., regardless of the content of tuples) via the store stream. Meanwhile, it is sent to all the units belonging to S (resp. R) for join operation via the join stream. We note that r (resp. s) will be only stored in one R (resp. S) unit whereas it can be discarded after the join operation in each S (resp. R) unit.

For *low-selectivity* joins such as equi-join, a content-sensitive routing strategy may be preferred, e.g. hash-partitioning. Hash-partitioning is able to partition the tuples with the same hash values into the same unit. It guarantees the data locality which facilities efficient join processing. However, hash-partitioning may incur load imbalance when the data is skew. The load imbalance of hashing can be alleviated by pre-computing the data distribution (e.g., histogram) as in traditional DBMS. However, this approach does not work well in the stream system, since it is difficult to obtain such information in advance.

To tackle the problem of load imbalance in content-sensitive routing, we adopt a hybrid routing strategy, *ContRand*, to make use of both the content-sensitive and random routing strategies. Algorithm 1 shows the control flow of ContRand routing, whose basic idea is as follows:

1) Suppose there are m units for relation R (i.e., $G_R = \{R_1, ..., R_m\}$) and n units for relation S (i.e., $G_S = \{S_1, ..., S_n\}$). G_R is logically divided into d disjoint subgroups (i.e., $\{G_{R,1}, ..., G_{R,d}\}$) where each $G_{R,i}$ includes one or more units and $\bigcup_{k=1}^{d} G_{R,k} = G_R$. Similarly, G_S is divided into e subgroups (i.e., $\{G_{S,1}, ..., G_{S,e}\}$). For instance, Figure 3 indicates four units of relation R which are logically divided into two subgroups $G_{R,1} = \{R_1, R_2\}$ and $G_{R,2} = \{R_3, R_4\}$.

2) Based on the partition above, ContRand decides which unit to store a tuple r by hashing r to a subgroup (content-sensitive manner) and randomly choosing a unit in that subgroup (content-insensitive manner). A composite function over r is used, such as $(rand \circ h_R)(r)$, where $rand$ and h_R are random and hashing functions for relation R, respectively. For the example given in Figure 3, according to the hashing value of $r1$, $r1$ is assigned to a unit in the first subgroup $G_{R,1}$. Similarly, ContRand uses $(rand \circ h_S)(s)$ to store tuple s in G_S. This approach is expected to work better than the pure content-sensitive routing strategy, especially when the data is skewed.

3) For join processing, ContRand routes r to all the units

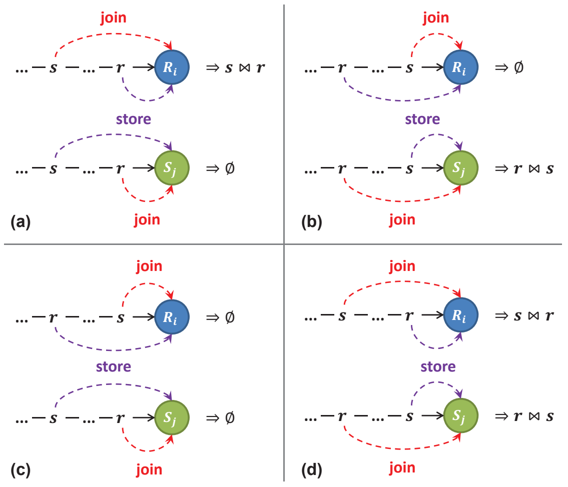


Figure 4: All cases of store-and-join processing.

in a subgroup calculated using $h_S(r)$. This guarantees that r is sent to all the units in one subgroup storing s which has the same hash value as r.

In the implementation of ContRand, we have two extreme cases to handle. First, when each group has only one unit (i.e., $d = m$ and $e = n$), ContRand becomes pure content-sensitive. Second, when there is only one group in each relation (i.e., $d = e = 1$), ContRand falls back to the content-insensitive one. The corresponding quantitative analysis will be presented in Section 4. In the implementation of ContRand, we extend the *field stream grouping* in Storm to support local random routing inside a hash-defined subgroup for the store stream and adopt the *direct stream grouping* for the join stream.

3.2.3 Protocol Design

To ensure the completeness of join results, the join protocol must be carefully designed to handle the situation that the tuples may arrive at the processing unit in different orders. Recall that BiStream does not store the tuples that are sent to the units of the other joining relation for join processing. However, the timing of discarding these tuples needs to be carefully considered. Naively discarding each tuple immediately after the join operation would lead to anomalies due to the out-of-order issue of tuples in the streams.

Suppose there are two input tuples $r \in R$ and $s \in S$ that satisfy the join condition of $R \bowtie S$. Figure 4 illustrates all the possible orders in which they may reach the two units R_i and S_j, where the closer a tuple is drawn to a unit, the earlier it arrives at the unit. Among all the possible cases, Figure 4.a and Figure 4.b work well to generate exactly one join result. For instance, in Figure 4.a, for the R_i side, r arrives earlier than s and is stored into R_i; the later arriving s joins with the stored r and produces the join result. For the S_j side, r which arrives earlier does not join with the later arriving s and is discarded before the arrival of s. As a consequence, only one join result is produced between r and s in Figure 4.a. The symmetric situation is applied to Figure 4.b as well. These two cases share the property that r and s arrive at R_i and S_j in consistent orders. Unfortunately, anomalies arise when the order of arriving r and s conflicts with the order that R_i and S_j are supposed to maintain during the join operation. In Figure 4.c, no join result is produced in the R_i side because s which arrives earlier is discarded before the arrival of r. Similarly, no join result is produced in the S_j side because r is discarded before the arrival of s. As a result, no join result is produced between r and s though they satisfy the join condition. We term this as a *missed-join* problem. On the other hand, in Figure 4.d, the result of $r \bowtie s$ is produced twice because the later arriving s joins with the stored r in R_i and the later arriving r joins with the stored s in S_j. We call this a *duplicated-join* problem.

To avoid the aforementioned anomalies, one solution is to perform a temporal-relational join processing based on the application timestamps [11]. For instance, by assigning each tuple one timestamp based on the application time, we can define constraints on the timestamp ordering of the joining tuples. Let T_r and T_s be the timestamps for r and s respectively. Each unit R_i (resp. S_j) does not evaluate the relational join condition on two joining tuples r and s unless they satisfy the temporal constraint $T_r \leq T_s$ (resp. $T_s < T_r$). By doing so, the expected join behavior can be enforced regardless of the physical timing of tuple arrival. The straightforward implementation of this approach, however, requires each unit to keep the tuples from both relations in memory, which incurs high memory consumption. Hence, we further design a protocol to discard the tuples in a proper way to reduce the memory usage, yet guarantee the correctness of the join results.

As mentioned earlier, if two tuples are processed at the units with a consistent order, the join result is produced correctly. To guarantee such a consistent order, we introduce a protocol to ensure that the relative order for any two joining tuples r and s only depends on one global order, and is consistent over all the units.

Definition 2 (**Order-Consistent Protocol**)**.** Given a set of dispatcher tasks Y and a set of processing units U, each task $y_i \in Y$ sends a set of tuples $X_i = \{x_{i_1}, ..., x_{i_k}, ...\}$ as a stream. Each tuple is broadcast to a set of processing units. A network protocol is called *order-consistent* if and only if: There exists a global tuple sequence $Z = (x_{z_1}, x_{z_2}, ...)$, where Z contains each tuple exactly once. For each unit $u_j \in U$, it receives all the tuples assigned to it (i.e., no loss in the network), and the sequence of tuples it processes is a subsequence of the global tuple sequence Z.

First, we need the message passing and processing between every pair of dispatcher task and processing unit is first-in-first-out (FIFO). Otherwise, the unit cannot determine whether a received tuple is delayed in the network. For example, suppose r and s are sent by the same dispatcher task. R_i receives r and then s, however, r is delayed in S_j. It is impossible to determine how long S_j should wait for r before processing the early arriving s, unless there exists some kind of communication between R_i and S_j. To this end, we design a *pairwise FIFO protocol* to guarantee that for every pair of dispatcher task and processing unit, the in-between message passing and processing is FIFO.

Definition 3 (**Pairwise FIFO Protocol**)**.** Given a dispatcher task y and a processing unit u, y sends a set of tuples $X = \{x_1, ..., x_k, ...\}$ to u as a stream. A network protocol is called *pairwise FIFO* if and only if: For any two tuples x_a and x_b, if x_a is sent by y before x_b, then x_a is processed at u before x_b.

This protocol is widely used in the design of communication networks [24]. The basic idea is to give each tuple an id which is incremented by one after every sent tuple (dispatcher side) or received tuple (processing unit side). The receiver u only processes those tuples with the correct tuple id. Thus, when a tuple is delayed or lost, the tuple id to be received will be held and other tuples cannot be received. However, they can be buffered to make the overhead of this protocol lightweight, and this depends on the implementation of this protocol.

Now we can design the order-consistent protocol based on the above pairwise FIFO protocol. The key component of the protocol is the logical clock [25]. Each dispatcher task maintains a logical clock (i.e., a monotonically increasing counter) which is incremented by one per tuple from the shuffler. Every time before emitting a tuple, the dispatcher task appends a logical timestamp to the tuple. Each processing unit sorts the received tuples based on the timestamp. In this case, the global sequence is just based on the order of timestamp. However, in the context of stream processing, we cannot wait till the last tuple before sorting tuples and output. For this reason, the protocol must know the current status of each dispatcher task, and which part of tuples can be to sort and process (i.e., no more incoming tuples containing such timestamp). Here, we use an idea similar to stream punctuation [37] to solve it. For each dispatcher task, it broadcasts a signal tuple with timestamp to all the units periodically (e.g., every 10ms). Since message passing and processing within every pair of dispatcher task and processing unit are FIFO, such a signal indicates that all tuples (from this dispatcher task) before this timestamp have been received by the units. Each processing unit maintains a table of the latest signal timestamp from each dispatcher task. If all the dispatcher tasks' logical clocks reach a certain timestamp, the processing unit must have received all the tuples before this timestamp. Thus, we use a priority queue in each processing unit to buffer every receive tuple, and process (i.e., store or join) those tuples that have a smaller timestamp than the smallest current latest signal timestamp.

For this protocol, each dispatcher task needs to generate timestamp for its tuple based on its own knowledge. An ideal case is that all the dispatchers have a same global time, which is usually not easy to obtain. However, if we can relax the constraint of timer synchronization among different dispatcher tasks (e.g., within 100ms difference), it would become easier to achieve. Such time difference will not affect the correctness of the protocol. Instead, we only need to buffer more since the difference of the latest signals timestamp from dispatchers may become larger.

By using the order-consistent protocol to send tuples from multiple dispatcher tasks to multiple processing units, the order of tuple processing will be consistent in all processing units. Therefore, the missed-join and duplicated-join problems due to the inconsistent processing order can be definitely avoided.

3.3 Window-Based Stream Joins

Unlike a full-history stream join, a window-based stream join evaluates the join condition only for tuples within the designated window [6]. The most widely used window-based stream join is to apply the time-based sliding window [16, 33]. Typically, given a time window size ϖ, for any input stream tuple $r \in R$ (resp. $s \in S$), it is joined only with the tuple $s \in S$ (resp. $r \in R$) such that $0 \leq r.time - s.time \leq \varpi$ (resp. $0 \leq s.time - r.time \leq \varpi$).

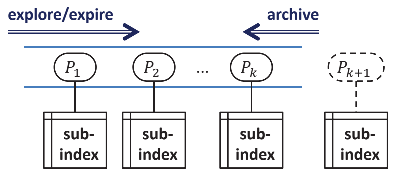


Figure 5: Chained in-memory index.

Two operations are essential for the maintenance of a sliding window. One operation is to verify the window constraint, which determines the tuples that should participate in the join operation. To do this, it requires the system to assign a synchronized timestamp to each incoming tuple. In BiStream, the timestamp is assigned at the shuffler. The other essential operation is to invalidate and discard the expired tuples from memory. This operation is important for releasing the memory and performing efficient join. Theorem 1 provides the criterion when a tuple can be safely removed from memory.

Theorem 1. *Given the maximum network delay $\mathcal{D}$, the stored tuples $r \in R_i$ can be safely discarded from memory, when R_i receives an incoming tuple $s \in S$ iff $s.time - r.time > \varpi + \mathcal{D}$. Likewise, $s \in S$ can be discarded from S_j once S_j receives one tuple such that $r.time - s.time > \varpi + \mathcal{D}$.*

Proof. Given a tuple $r \in R_i$, it needs to join $s \in S$ whose timestamp is within $[r.time, r.time + \varpi]$. With $\mathcal{D}$, the last tuple s' within r's window arrives at R_i at the moment of $r.time + \varpi + \mathcal{D}$ such that $s'.time = r.time + \varpi$. Thus, if $s \in S$ arrives later than s', then $s.time > r.time + \varpi$ which is outside $[r.time, r.time + \varpi]$. In other words, r will not be in any window of s after s' and thus can be safely discarded from memory. □

Indexes are required to support efficient stream join processing. However, the existing approaches of organizing the entire stream data with one single index incur high overhead for re-adjusting the index when expired tuples are discarded. To facilitate low-overhead data discarding and index exploration, we propose the *chained in-memory index* to overcome the high overhead incurred in using a single index.

Figure 5 illustrates the structure of the chained in-memory index. The basic idea is to partition the streaming tuples with respect to the discrete time intervals and construct a subindex per interval. The span of the time interval is referred as the *archive period* $\mathcal{P}$. Each subindex is associated with the minimum and maximum timestamps of the tuples it contains. All subindexes are chained as a linked list by the order of their construction time. According to the window constraint, expired data are discarded from memory in the granularity of subindex-level rather than tuple-level. This reduces the overhead of data discarding since the valid subindexes are not affected when the obsolete subindexes are discarded. We summarize the main procedures of indexing data, discarding data, and join processing in the chained in-memory index as follows.

Data Indexing: When each input tuple (say $r \in R_i$) is inserted into an active subindex, it first updates the minimum and maximum timestamps. It then calculates whether the difference between the minimum and maximum timestamps has exceeded $\mathcal{P}$. If this is true, the active subindex becomes inactive and is archived into the chain, and a new empty active subindex is created. Otherwise, the current active subindex remains active. Only one active subindex is used to index an input tuple.

Data Discarding: An expired subindex can be safely removed from memory. Checking for the expired subindexes is triggered when a tuple reaches the unit belonging to the other relation (e.g., s reaches R_i where $s \in S$). The criterion to discard one subindex from the chain is the difference between the timestamp of s and the maximum timestamp of the subindex being large than $\varpi + \mathcal{D}$ according to Theorem 1. Theorem 1 ensures that the removal of the subindex from memory does not affect the integrity of join results. This coarse-grained level data discarding improves the discarding efficiency by avoiding pairwise comparisons inside each subindex.

Join Processing: After data discarding, s needs to join with all the tuples in the remaining in-memory subindexes. Recall that, for sliding-window joins, the window verification is necessary to identify whether each tuple falls in the window. One optimization we adopted to speed up this window verification is to identify which subindex is partially or completely expired in the window, by comparing the difference between $s.time$ and the minimum timestamp of the subindex. If the minimum timestamp is in the window that s may interact with, then the entire subindex does not need to conduct the window verification and can directly perform join processing. Otherwise, the pairwise window verification is needed during join processing in the partially expired subindex.

3.4 Supporting Other Operators

Besides the join operator, a typical stream query may also involve other operations such as selection, projection and aggregation. Such a query is typically referred to as a stream select-project-join-aggregation (SPJA) query [3]. An example stream SPJA query is shown below.

```
SELECT ONLINE R.a2, sum(S.b2)
FROM R, S
WHERE R.a1 > S.b1 WITHIN 10 minutes
GROUP BY R.a2;
```

where the `WITHIN` clause indicates that it is a sliding-window query. If the `WITHIN` clause is omitted, the aggregation of the query would be based on the full-history join results.

A complex stream SPJA query may need to join more than two streams. For such a complex query, a binary tree-like (e.g., right-deep tree [9]) execution plan is constructed accordingly, which comprises a sequence of binary stream join operators. All the join operators are executed in a pipeline manner: each operator joins two streams, each of which is either an input stream or from another join operation, and produces the join result that is potentially fed to another operator implemented using *shuffle stream grouping* in storm. Focusing on the join processing, we further describe how selection, projection and aggregation are supported for stream SPJA queries in BiStream.

3.4.1 Selection and Projection

The selection and projection operations could be either duplicate-preserving or duplicate-eliminating. Duplicate-preserving selection and projection allow the existence of duplicated results. They are stateless and thus can be easily conducted by filtering the streams in a tuple-at-a-time manner without using any extra memory for storing the intermediate state [5]. BiStream can naturally support these operations at the router side over the input streams and at the joiner side over the stream join results according to the query plan.

However, duplicate-eliminating selection and projection need to remove the duplicated results, which typically requires the stream system to add an additional bolt (referred as the *deduplicator*) to filter out the duplicates. Many existing duplicate-eliminating techniques (such as hash probing [22]) can be used in the implementation of the deduplicator. Due to the potentially unbound memory usage for keeping track of the distinct values over the full-history data [5], in practice, these operators are always applied with windowed constraints. In BiStream, they can be supported by integrating the deduplicator into either the router or the joiner to enable duplicate elimination.

3.4.2 Online Data Aggregation

In order to facilitate real-time decision making, BiStream also supports online data aggregation over the stream join outputs. Performing online data aggregation comprises two tasks: join and aggregation. The join processing can be performed by directly exploiting the proposed join approach (e.g., using the join predicate $R \bowtie_{R.a1>S.b1} S$). The aggregation computation is to aggregate the join results based on group-by and aggregate functions (e.g., in the example query, group by $R.a2$ and perform `sum` over $S.b2$). To support efficient distributed aggregation processing, we adopt a two-phase approach [32]. The first phase is to pre-aggregate a partial local view in each unit in the joiner. As each unit contains part of the join outputs independently, the local view computation can be directly performed. The second phase is to shuffle the local views to a coordinating component (referred as the *merger*) and merge them into a global view from time to time. The merger is implemented as a bolt receiving the data stream from the joiner. The merger may involve multiple processing units to parallelize the processing of constructing the global view.

In a stream environment, frequent view updates may incur high communication and computation overhead between the joiner and the merger. One improvement is to adopt the lazy-update or batch-update strategy. Instead of pushing the local aggregation results to the merger immediately when every tuple arrives, we compute the updated views by batching the updated tuples based on a time period. In this way, multiple local aggregate updates can be batched together so that the overhead can be reduced.

3.5 Fault Tolerance

Two types of failures that typically occur in a stream system are *data loss* and *message loss*. Data loss happens when there is power failure or node corruption, which results in the loss of data stored in memory, and hence, loss of the join states. Message loss is often due to network partition or hiccup, which results in the loss of on-the-fly data during message passing, and hence, incomplete dataflow processing.

To handle data loss, BiStream adopts data replication to prevent data loss for the unit failure in the joiner. BiStream maintains one in-memory data as the primary copy and several other copies as backups for failure recovery purposes. As we have discussed above, during the join processing, every tuple r of R (resp. s of S) is stored into one unit of R (resp. S) that is considered as the primary copy to support the join processing. The other copies are stored in the units of S (resp. R) where r (resp. s) is sent for join processing. Intuitively, when r is sent to the units of S for join, BiStream enforces some units to store r into their persistent storage (e.g., hard disk) as replicas after the join operation. When the system detects a unit failure, the failed partition can be restored from other archived replicas in the system.

To handle message loss, since the topology of the router and joiner is a directed acyclic graph (DAG), BiStream adopts the upstream backup technique [19]. Specifically, the upstream nodes act as backups for their downstream neighbors by preserving tuples in their output queues while their downstream neighbors process them. If the message in the downstream fails, its upstream nodes replay the logged tuples on the recovery node.

4. COMPARATIVE ANALYSIS

In this section we provide the comparative analysis between join-biclique and join-matrix models in terms of memory consumption, scalability and communication cost. Table 2 lists the notations used throughout this section. In addition, we use the subscripts *bi* and *mat* to distinguish join-biclique and join-matrix.

Assume that there are m and n partitions for relations R and S respectively, and the join keys over $R \bowtie_\theta S$ are of uniform distribution. With this setup, join-biclique requires $m + n$ units while join-matrix requires $m \times n$ units. In addition, as the random routing strategy is an extreme case of the ContRand routing strategy, our discussion will be based on ContRand routing for join-biclique where m and n partitions are divided into d ($\leq m$) and e ($\leq n$) subgroups, respectively.

4.1 Memory Consumption

We quantify the memory consumption for the full-history join. As the full-history join can be viewed as an extreme case of the window-based join where the size of window is infinite, the following analysis also applies to the window-based join in the context of the worst case analysis.

The data stored in join-biclique consist of the stored tuples of both joining relations and the buffered tuples within a time window (say $\mathcal{D}$) required by the join protocol. Thus, the total amount of memory consumption is as follows:

$$W_{bi} = \chi_R \cdot (|R| + n \cdot \frac{\varphi_R}{e} \cdot \mathcal{D}) + \chi_S \cdot (|S| + m \cdot \frac{\varphi_S}{d} \cdot \mathcal{D})$$

where $|R|$ and $|S|$ quantify the stored tuples, and $n \cdot \frac{\varphi_R}{e} \cdot \mathcal{D}$ and $m \cdot \frac{\varphi_S}{d} \cdot \mathcal{D}$ quantify the buffered tuples. Typically, the number of stored tuples is much larger than the temporarily buffered ones, inferring $n \cdot \frac{\varphi_R}{e} \cdot \mathcal{D} \ll |R|$ and $m \cdot \frac{\varphi_S}{d} \cdot \mathcal{D} \ll |S|$. By ignoring the memory usage of buffering tuples, the memory consumption of join-biclique can be approximated as

$$W_{bi} \approx \chi_R \cdot |R| + \chi_S \cdot |S| \quad (1)$$

Different from join-biclique, join-matrix replicates each input tuple over an entire row or column. As a consequence, the memory consumption of join-matrix is as follows:

$$W_{mat} = \chi_R \cdot n \cdot |R| + \chi_S \cdot m \cdot |S| \quad (2)$$

Equation 2 shows the memory consumption for join-matrix is sensitive to the matrix configuration. This infers that join-matrix suffers from high memory consumption when m and

Table 2: Terms used in the analytical model.

Terms	Description
$R_{k,i}$	The i-th partition of the k-th subgroup of R
$S_{l,j}$	The j-th partition of the l-th subgroup of S
m_k	Number of partitions in the k-th subgroup of R
n_l	Number of partitions in the l-th subgroup of S
$\|R\|$, $\|S\|$	Number of arrived tuples of relation R (S)
χ_R, χ_S	Size of a tuple of relation R (S)
W	Cluster memory consumption
φ_R, φ_S	Average incoming rate of stream R (S)
$In(\cdot)$	Input bandwidth of a node
B	Intermediate stream bandwidth

n are large. Differently, Equation 1 shows the memory consumption of join-biclique is independent on the partitioning scheme and only proportional to the amount of inputs. This indicates that join-biclique is much more memory-efficient.

4.2 Scalability

We consider two important properties with respect to system scalability. The first is on the *flexibility* of adding or removing the units into or from the system. This can be measured by the system scaling granularity in terms of the number of units being added or removed. The second is on the *effectiveness* of utilizing the memory of newly added units. This can be measured by the scaling gain ratio (SGR).

Definition 4 (**Scaling Gain Ratio**). In system scaling, the scaling gain ratio (SGR) is the percentage of newly added memory that can be used for processing new inputs. Higher SGR indicates better scalability.

In the join-matrix model, since the system needs to maintain the matrix structure, it can only add or remove one entire row or column during system scaling out or down. It does not support single unit increment per scaling operation, as this violates the matrix structure. Especially, when there is a big matrix, adding one row or column may increase many more machines which may not be actually needed. This therefore limits the flexibility of the system. On the contrary, join-biclique does not need to maintain a rigid partitioning scheme, enabling it to be more flexible in scaling. In join-biclique, each unit is independent to each other, and any number of units (even single unit) can be easily added or removed from the system for either relation. In other words, join-biclique supports a more fine-grained units change which provides the flexibility in scaling.

For the effectiveness of resource utilization, in join-matrix, when a new unit is added (corresponding either a row or column increases), part of its memory is used to store the replicas of the in-memory data in other units and only the remainders can be used to increase the system storage capacity. As shown in Figure 1.a, when a row is added for R, tuples of relation S have to be replicated to the new unit in the same column to guarantee the correctness of join processing. The space required for storing the replicas is therefore $\chi_S \cdot (|S|/n)$, where n is the number of columns. Thus, the SGR of adding a new partition for relation R is

$$SGR_{mat,R} = \frac{\mathcal{C} - \chi_S \cdot (|S|/n)}{\mathcal{C}} = 1 - \frac{\chi_S}{\mathcal{C}} \cdot \frac{|S|}{n} \quad (3)$$

where $\mathcal{C}$ is the memory capacity of a unit. Similarly, when adding a partition for relation S, we can calculate SGR as

$$SGR_{mat,S} = \frac{\mathcal{C} - \chi_R \cdot (|R|/m)}{\mathcal{C}} = 1 - \frac{\chi_R}{\mathcal{C}} \cdot \frac{|R|}{m} \quad (4)$$

As can be seen from Equation 3 and Equation 4, for join-matrix, the SGR of expanding relation R (resp. S) is affected by the number of tuples stored in S (resp. R) and the number of partitions of S (resp. R). Consequently, the SGR will be very small when the number of stored tuples becomes large. This indicates the effectiveness of system scaling out tends to be marginal with the growth of system storage capacity.

For join-biclique, as no data replication is needed, the whole memory of the new unit can be utilized to increase the capacity of system storage. For instance, when one unit is added to relation R, no data from other units are replicated to the new unit. In this case, the SGR remains 1 whenever the system scales out, as 100% of the memory can be utilized to share the workload.

4.3 Communication Cost

We further provide the comparative analysis on the communication cost of intermediate streams in different models.

In join-biclique, the bandwidth usage for each unit $R_{k,i}$ or $S_{l,j}$ is provided as follows:

$$In(R_{k,i}) = \chi_R \cdot \frac{\varphi_R}{d} \cdot \frac{1}{m_k} + \chi_S \cdot \frac{\varphi_S}{d}$$
$$In(S_{l,j}) = \chi_S \cdot \frac{\varphi_S}{e} \cdot \frac{1}{n_l} + \chi_R \cdot \frac{\varphi_R}{e}$$

Let $\Sigma_R = \chi_R \cdot \varphi_R$ and $\Sigma_S = \chi_S \cdot \varphi_S$. Then the bandwidth of intermediate streams required by join-biclique is:

$$\begin{aligned} B_{bi} &= \sum_{k=1}^{d}\sum_{i=1}^{m_k} In(R_{k,i}) + \sum_{l=1}^{e}\sum_{j=1}^{n_l} In(S_{l,j}) \\ &= (\frac{n}{e}+1)\cdot\Sigma_R + (\frac{m}{d}+1)\cdot\Sigma_S \end{aligned} \tag{5}$$

In join-matrix, the bandwidth of intermediate streams is:

$$B_{mat} = n \cdot \Sigma_R + m \cdot \Sigma_S \tag{6}$$

As can be seen from Equation 5 and Equation 6, join-biclique is superior to join-matrix in bandwidth usage to support the same number of partitions in most cases. Only when a random routing is adopted (i.e., $d = e = 1$), it is inferior to join-matrix with an extra $\Sigma_R + \Sigma_S$ for the entire cluster.

5. ADAPTIVE RESOURCE MANAGEMENT

The goal of adaptive resource management is to achieve dynamic recalibration, reacting to frequent changes in data and statistics. Adaptive solutions can supplement regular execution with an adaptive resource manager that monitors the unit load and triggers changes. Considering the ContRand routing strategy, each subgroup is independent to each other. In BiStream, the resource manager monitors the load situation of the units in each subgroup and performs scaling out/down operations within each subgroup. We use the *memory load volume* (MLV) as a metric to calculate the memory consumption in each unit.

BiStream adopts a two-phase scaling (2PS) protocol – a *requesting* phase and a *scaling* phase – to adaptively adjust the resource management. For the ease of illustration, we assume all the units are homogeneous with the same memory capacity $\mathcal{C}$. However, the protocol can be easily extended to the heterogeneous scenarios. Next we introduce the basic idea of 2PS protocol. The detailed algorithm is omitted here due to space constraint.

1.) In the requesting phase, the resource manager broadcasts a request to all the units to collect the load status. Each unit acknowledges the request by reporting its MLV to the resource manager. After receiving all the responses, the resource manager summarizes the load status and determines whether a scaling operation is necessary. Once a scaling decision of either scaling out or down is made, the system goes to the scaling phase.

2.) In the scaling phase, the resource manager conducts the scaling operation based on the decision. For scaling out operation, the resource manager allocates new units and performs necessary data migration to rebalance the load among the units in the current subgroup. For scaling down operation, the resource manager first distributes the data in the units that will be removed to other units in the same current subgroup, and then releases these units.

Since BiStream conducts the adjustment in each subgroup independently, we illustrate the control flow based on one subgroup (e.g., subgroup G with k units). The resource manager periodically sends the request messages $q_1, ..., q_k$ to all the k units in G. Every i-th unit acknowledges the request q_i through a response message p_i with its current MLV, denoted as $MLV(p_i)$, where $i \in \{1, 2, ..., k\}$.

After the resource manager has received all the responses $\{p_1, ..., p_k\}$, it calculates the percentage τ of the memory usage in the subgroup as

$$\tau = \frac{\Sigma_{i=1}^{k} MLV(p_i)}{k \cdot \mathcal{C}}$$

τ is used as an indicator to make the decision. Intuitively, if τ exceeds a stated threshold $\mathcal{T}_h$, the system is considered to be overloaded. In contrast, if τ is below a stated threshold $\mathcal{T}_l$, the system is considered to be underloaded. There are two important issues we need to address during the system adjustment. The first one is how to avoid false positive scaling due to the fluctuation of inputs; the second one is how to determine the number of units to adjust.

To address the first issue, the resource manager periodically collects the load status of units and justifies the demand of system adjustment. The adjustment is confirmed only when the resource manager makes the same adjustment decisions for consecutive iterations. This is implemented by maintaining a counter. The counter is incremented by one if the decision is the same as in the precious iteration. Otherwise, the counter is reset to zero. Once the counter reaches the stated threshold, the adjustment is confirmed. In this way, the protocol can tolerate the fluctuation of instant load and avoid making the false positive scaling decision.

The number of units required for adjustment of resources can be quantified as follows:

$$\Delta = \frac{\Sigma_{i=1}^{k} MLV(p_i)}{\Psi \cdot \mathcal{C}} - k = (\frac{\tau}{\Psi} - 1) \cdot k \tag{7}$$

where Ψ ($\Psi \in (\mathcal{T}_l, \mathcal{T}_h)$) is the expected value that indicates the overall load status after adjustment. Equation 7 is applicable to both scaling out and down operations. When Δ is positive, it indicates the system should add units to the subgroup. When Δ is negative, $|\Delta|$ units should be removed from the subgroup.

6. EVALUATION

BiStream is built on Storm [1] and uses Kafka [23] as the input stream adapter. The ripple join is used as the local join algorithm at each joiner task. We compare BiStream with the stream join system developed over join-matrix [1].

6.1 Experimental Setup

Environment. We conduct all the experiments on a cluster of 40 servers, each of which is equipped with an Intel Xeon X3430 @ 2.4GHz CPU and runs CentOS 5.11. Each server is logically partitioned into two processing units, each of which has 2GB RAM. Overall, there are 80 processing units available exclusively for our experiments.

Data sets. We focus on our study using the existing benchmark TPC-H [2]. We generate the TPC-H data sets using the `dbgen` tool shipped with TPC-H benchmark. All the input data sets are pre-generated before feeding to the stream system. In addition, the benchmark tables are projected before feeding into the join system with the attributes only needed in the queries. Different numbers of data are generated to support different data volume requirement during the evaluation. We run all the experiments with asymmetric input rates of two join streams in which case the data volume per second in each stream remains almost the same.

Queries. We consider three join queries including two equi-joins from the TPC-H benchmark and one synthetic band-join which are also used in [12]. The two equi-joins, namely $Q5$ and $Q7$, represent the most expensive join operations in the queries Q5 and Q7 from the benchmark. The synthetic band-join (*Band*) is as follows:

```
SELECT * FROM LINEITEM L1, LINEITEM L2
WHERE ABS(L1.orderkey-L2.orderkey) <=1
AND (L1.shipmode='TRUCK' AND L2.shipinstruct='NONE')
AND L1.Quantity > 48
```

We have also evaluated the performance with online aggregation queries. Due to space constraint, please refer to Appendix B for the results.

Settings. For both join-biclique (JB) and join-matrix (JM), we use the static partitioning schemes that are optimal to the input streams. Specially, to offer the best performance for JM, we consider the square matrix data partitioning scheme for joining equivalent volume streams [12]. For JB, we also assign the same number of units to each relation. Throughout the discussion, we use three different types of join processing settings under JM and JB: `JMx`, `JBx` and `JBx-CRy`. The details of these settings are listed in Table 3.

In addition, we define the *processing volume* as the number of input tuples processed by the system. The *maximum processing volume* is the point when the unit reaches its memory saturation threshold (i.e., the capacity of in-memory processing). Meanwhile, we use the memory load volume (MLV) to measure the memory consumption in each unit.

6.2 Memory Consumption

We first show the memory consumption comparison between JB and JM. Figure 6.a provides the average MLV changes of these two models when the processing volume varies from 1M (million) to 19M. It is conducted over the full-history joins for queries $Q5$, $Q7$ and *Band*. In this experiment, both JB and JM use 16 units, i.e., JB16 and JM16. The memory saturation threshold for each unit is set to 1.5GB. As expected, we can see that JM always consumes memory faster than JB for all the three queries. For example, for query $Q7$, JM reaches the maximum processing volume at 5.8M tuples, whereas JB is at 19M tuples. Similarly, JM reaches the maximum processing volume much earlier than JB for $Q5$ and *Band*. This is because JM needs to store many replications in the joining model, while JB does not. The experimental result confirms that JB is much more memory-efficient and can benefit more in-memory processing than JM over a fixed size of cluster.

[1]`https://github.com/epfldata/squall`

Table 3: List of model settings.

Name	Description
JMx	The join-matrix over $\sqrt{x} \times \sqrt{x}$ matrix, where each partition corresponds to $\sqrt{x}$ units.
JBx	The join-biclique over $x/2 + x/2$ units with random routing, where each relation corresponds to $x/2$ units.
JBx-CRy	The join-biclique over $x/2 + x/2$ units with ContRand routing, where each relation has $x/2$ units partitioned into y subgroups.

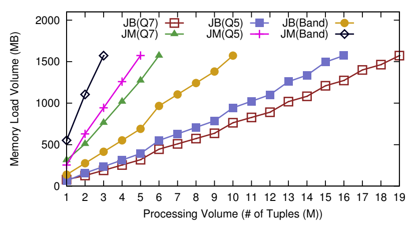
Unlike full-history joins, sliding-window joins can bound the memory usage to the size of the workload within the time window, since expired tuples can be discarded from memory. To illustrate the effectiveness of the chained in-memory index, Figure 6.b shows the memory consumption of JB16 for query $Q5$ with sliding windows of 3, 5 and 8 minutes. In this experiment, the average input rate is set to 15K tuples per second and the archive period $\mathcal{P}$ of the chained in-memory index is set to one-tenth of the window size. The experiment results show that after one window time, the memory usage is bounded via data discarding. The ripples in the MLV curve indicates the memory consuming decreases quickly when the data is discarded. This benefits from the data discarding strategy in JB because the discarding is based on the granularity of subindex other than individual tuple.

Furthermore, Figure 6.b also indicates that a longer window (or a bigger processing volume) causes a higher MLV. For example, the 8-minute window uses more memory than the 3-minute and 5-minute windows. In other words, the maximum size of processing volume that a system can support reflects the size of the time window it can achieve for in-memory join processing. As shown in Figure 6.a, JB can support a much bigger processing volume than JM. This provides us another insight: given a cluster, JB is able to support a much bigger window size in supporting in-memory sliding-window joins than JM. More experimental evaluations about the derived maximum sizes of sliding windows with respect to different settings can be found in Appendix C.

6.3 Throughput and Latency

Next, we study the throughput comparison among the different models. To provide a comprehensive study, we test five different settings, where JB8 has the same number of partitions (4 partitions for each relation) with JM16, and the remaining three settings with JB16 use the same number of units (16 units in total) as JM16. Figure 8 presents the throughput comparisons for queries $Q5$, $Q7$ and *Band* when the system is fed in 10GB, 80GB and 320GB data sets.

As shown in Figure 8.a, for $Q5$, all different settings provide comparative throughput when the system is fed with 10GB data. This is because the memory is sufficient to hold all the 10GB data in memory in all settings. We observe that the throughput of JB16 (with random routing) is smaller than JM16 in the 10GB case. This is because, under random routing, all the units of one relation are involved in the join processing of every input tuple of the other


(a) Full-history joins.

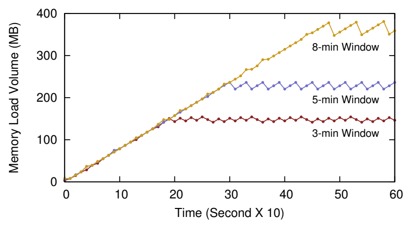

(b) Sliding-window joins.

Figure 6: Memory usage.

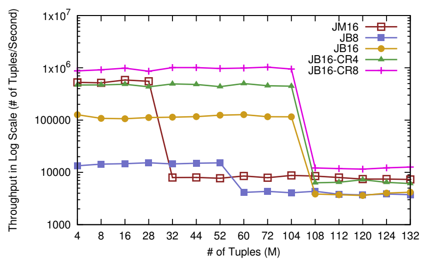
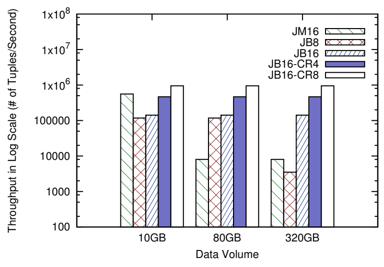
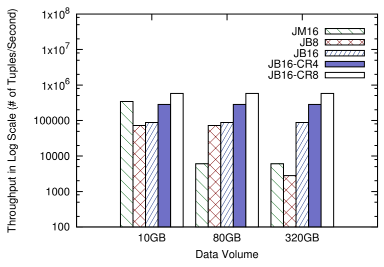


Figure 7: Real-time throughput.


(a) Throughput for query $Q5$.


(b) Throughput for query $Q7$.

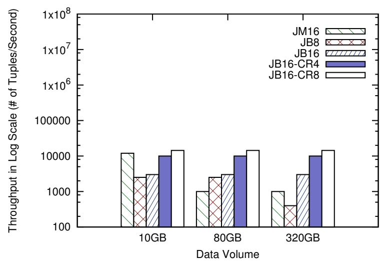

(c) Throughput for query *Band*.

Figure 8: Average throughput with different processing volume.

relation. This causes each unit to process more tuples in JB16 than in JM16, resulting in the throughput decrease of JB16. As shown in the 10GB case, once JB adopts the ContRand routing strategy, its throughput becomes similar or even higher than JM. When the system is fed with 80GB data, the throughput of JM16 decreases sharply and performs far behind JB. For example, JB16-CR8 performs better than JM16 for almost two orders of magnitude. Interestingly, JM16 with 16 units performs even worse than JB8 with 8 units for one order of magnitude. This is because 80GB has exceeded the maximum processing volume of JM16, resulting in frequent disk I/Os, thereby reducing the throughput. With 80GB data, the processing in other JB settings remains in-memory. As shown in Figure 8.a, the throughput of JB8 decreases as well after JB8 reaching its maximum processing volume for 320GB. This further confirms that in-memory processing is the key to efficient stream join processing.

For queries $Q7$ and *Band*, we observe similar trends of throughput change with different settings as query $Q5$. Note that the throughput of $Q5$ is around 1.5X bigger than $Q7$ on average where the difference is unclear after using the log scale in the results. Figure 8 shows that the throughput of processing $Q5$ and $Q7$ is much higher than that of processing *Band*. This is within expectation, as query *Band* involves inequality join and incurs higher computation overhead than equi-join queries $Q5$ and $Q7$.

Figure 7 illustrates the throughput change along the increase of tuples ingested for query $Q5$. Consistently, JB16-CR8 provides the highest throughput than all the other settings. In addition, JM is the first one to reach its system memory capacity when 28M tuples are fed. Interestingly, the maximum processing volume of JB8 is around 52M tuples, which is much larger than that of JM16. Furthermore, the JB settings with 16 units yield high throughput and provide much larger memory capacity than JM for efficient in-memory join processing. Another insight we obtain is that the content-sensitive routing strategy can increase the throughput of JB. Figure 8 and Figure 7 confirm that JB16-CR8 (resp. JB16-CR4) outperforms JB16-CR4 (resp. JB16) in terms of the throughput. This is because the content-sensitive routing strategy renders each input tuple sent to some specific units instead of all units for join processing.

We also examine the latency of different models. The latency is measured by capturing the difference between the time producing a join result and the time of a more recent tuple participated in the join entering the join system. Figure 9 provides the latency of JM and JB under different settings for queries $Q5$, $Q7$ and *Band*. The box in each series encloses the 25/75 percentile of the distribution with median shown as a horizontal line within each box. The top and bottom horizontal lines outside the box show the worst and best cases respectively. The statistical results are collected after the system is running at maximum throughput with in-memory processing. The results indicate that JB and JM achieve a similar average latency (within 10ms difference) within each query. We observe that the latency is dominated by the network transfer (due to the routing architecture) and the queuing of massive tuple processing in each unit. It is not surprising that JB16 setting generates a slightly larger latency than the others due to more data shuffling and join processing. It also shows that the latency of JB decreases when the ContRand routing strategy is adopted. For instance, the JB16-CR8 achieves a lower latency than JM for all the queries. This further confirms that the content-sensitive routing is able to prune the unrelated join tuples and further reduce latency.

6.4 Scalability and Elasticity

We further study the scalability by observing how the maximum processing volume changes when the system scales out. Figure 10 shows the results of processing query $Q7$ on the number of units varying from 4 to 64. As studied in [12], a square matrix is able to achieve a good performance for JM-based processing. So, we only show the results on the perfect squares of units for JB and JM. Interested readers

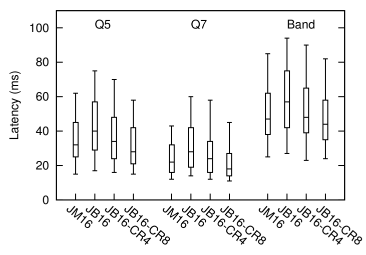


Figure 9: Latency.

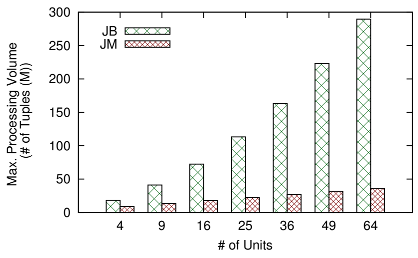


Figure 10: Scalability.

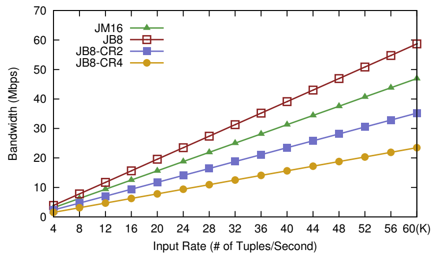


Figure 11: Bandwidth consumption.

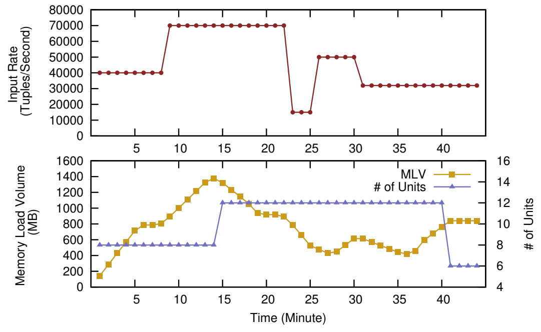


Figure 12: Dynamic resource management.

may refer to Appendix D for more results on other combinations of units. Experimental results indicate that JM does not scale well as the number of units increases. For example, it supports 19M and 37M tuples over 16 and 64 units, respectively. This is reasonable as JM has to store more data replicas with the increased size of matrix. This thereby limits the scalability of JM. As expected, we observe that JB achieves almost linear scalability. The maximum processing volume of JB increases in double when the number of units doubles. For example, JB supports 76M tuples of maximum processing volume with 16 units, while the tuples number increases to 290M with 64 units. This benefits from the unit independence such that the newly added units do not replicate any data from other units.

We next evaluate the capability of BiStream to dynamically adjust the number of resources according to the stream input rate changes. The experiment is conducted using $Q7$ for a 5-minute window join with JB8 as the initial setting. The top of Figure 12 presents the varying stream input rates during the 50 minutes of evaluation. As the input stream rate changes, the MLV changes as well. The overload and underload thresholds are set as $\mathcal{T}_h = 0.8$ and $\mathcal{T}_l = 0.3$ respectively, and the target load status after adjustment is set as $\Psi = 0.6$. In this experiment, we allow the resource manager to check the MLV in each processing unit with a 1-minute period. The scaling out/down operation is triggered once it receives the same scaling requests for 3 consecutive times. The bottom of Figure 12 shows how the processing units are dynamically added/released from the system while the MLV changes. From the result, we can see that the system is able to dynamically add/release the processing units once the memory is over/under loaded. For instance, 4 and 6 processing units are effectively added and removed from the system when the MLV reaches above 1.2GB and falls below 450MB respectively. During the system scaling, for sliding-window join, data migration is avoided since the system discards the expired tuples and controls the storage distribution of the new incoming tuples to equivalently achieve load balancing among the units. This favors no system downtime and no impact on the average latency. Another observation is that BiStream is able to handle the fluctuation of the workload to avoid the false adjustment. BiStream does not trigger any scaling out/down operation when the input stream is shortly increased and deceased between the 22^{nd} and 31^{st} minutes.

6.5 Communication Cost

Finally, we evaluate the network bandwidth usage within the join processing. Figure 11 provides the bandwidth consumption for processing query $Q5$, when the input rate varies from 4K to 60K tuples per second. For the ease of comparison, we measure the bandwidth usage for JM16, JB8, JB8-CR2 and JB8-CR4, where each relation has 4 partitions. Note that under this setting, JB can support double storage capacity than JM. The results of JB8 and JM16 in Figure 11 indicate that JB consumes higher bandwidth than JM, which is due to random routing. Once ContRand routing is adopted, JB8-CR2 and JB8-CR4 consume less bandwidth than JB8. From the experimental results, we can draw two insights: First, ContRand routing strategy can greatly reduce the bandwidth consumption in JB; Second, JB has competitive bandwidth requirement as JM but offers a larger maximum processing volume. Appendix E provides more experimental results on the bandwidth consumption over different settings.

In summary, our experiments show that JB is superior to JM in terms of memory-efficiency, elasticity and scalability, and is comparable in terms of latency. Integration of the content-sensitive routing strategy enables the system to further achieve a lower network consumption and latency.

7. CONCLUSION

In this paper, we proposed a novel stream join model, called join-biclique, for online stream joins with general predicates. Join-biclique logically models all the processing units as a bipartite graph for stream joins with no data replication, flexible partition scheme and processing units independence designs. On the basis of join-biclique, we designed and developed a distributed stream join processing system, *BiStream* with a large-scale commodity cluster. Specifically, it is designed to support efficient full-history joins, window-based joins and other operations including selection, projection and online data aggregation. BiStream also supports an adaptive resource management where the system can dynamically scale out or down. We theoretically analyzed the proposed techniques and also conducted extensive experimental evaluation. The evaluation results demonstrate that BiStream is memory-efficient, elastic and scalable.

Acknowledgments

This research is funded by the National Research Foundation (NRF), Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme ("SP-2: Database Support for Research on Megacities" R-252-000-496-281), and the A*STAR project 1321202073. We would like to thank the anonymous reviewers for their insightful feedback.

APPENDIX

A. REFERENCES

[1] Apache storm. `http://storm.apache.org`.

[2] The tpc-h benchmark. `http://www.tpc.org/tpch`.

[3] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica. Blink and it's done: Interactive queries on very large data. *Proceedings of the VLDB Endowment*, 5(12):1902–1905, 2012.

[4] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: fault-tolerant and scalable joining of continuous data streams. In *Proc. of SIGMOD*, pages 577–588, 2013.

[5] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 29(1):162–194, 2004.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of PODS*, pages 1–16, 2002.

[7] C. Balkesen, N. Tatbul, and M. T. Özsu. Adaptive input admission and management for parallel stream processing. In *Proc. of DEBS*, pages 15–26, 2013.

[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proc. of SIGMOD*, pages 37–48, 2011.

[9] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proc. of VLDB*, pages 15–26, 1992.

[10] S. Chen, P. B. Gibbons, and S. Nath. Pr-join: a non-blocking join achieving higher early result rate with statistical guarantees. In *Proc. of SIGMOD*, pages 147–158, 2010.

[11] D. Dey, T. M. Barron, and V. C. Storey. A complete temporal relational algebra. *The VLDB Journal*, 5(3):167–180, 1996.

[12] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. In *Proc. of VLDB*, pages 441–452, 2014.

[13] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539, 2014.

[14] B. Gedik, P. S. Yu, and R. R. Bordawekar. Executing stream joins on the cell processor. In *Proc. of VLDB*, pages 363–374, 2007.

[15] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of SIGMOD*, pages 13–24, 2001.

[16] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. of VLDB*, pages 500–511, 2003.

[17] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proc. of SIGMOD*, pages 287–298, 1999.

[18] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. of SIGMOD*, pages 171–182, 1997.

[19] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of ICDE*, pages 779–790, 2005.

[20] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. The sort-merge-shrink join. *ACM Transactions on Database Systems*, 31(4):1382–1416, 2006.

[21] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. The hells-join: A heterogeneous stream join for extremely large windows. In *Proc. of DaMoN*, 2013.

[22] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[23] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proc. of NetDB*, 2011.

[24] J. F. Kurose and K. W. Ross. *Computer networking: a top-down approach (6th edition)*. Pearson Education, 2012.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[26] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Computing Surveys*, 46(3):31:1–31:42, 2014.

[27] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A scalable hash ripple join algorithm. In *Proc. of SIGMOD*, pages 252–262, 2002.

[28] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *Proc. of SIGMOD*, pages 949–960, 2011.

[29] J. Qian, Y. Li, Y. Wang, H. Chen, and Y. Dong. An embedded co-processor for accelerating window joins over uncertain data streams. *Microprocessors and Microsystems*, 36(6):489–504, 2012.

[30] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: reliable stream computation in the cloud. In *Proc. of EuroSys*, pages 1–14, 2013.

[31] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. In *Proc. of VLDB*, pages 709–720, 2014.

[32] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proc. of SIGMOD*, pages 104–114, 1995.

[33] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. of VLDB*, pages 324–335, 2004.

[34] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins.

IEEE Transactions on Parallel and Distributed Systems, 4(12):1345–1354, 1993.

[35] J. Teubner, G. Alonso, C. Balkesen, and M. T. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Proc. of ICDE*, pages 362–373, 2013.

[36] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proc. of SIGMOD*, pages 625–636, 2011.

[37] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[38] S. Wang and E. Rundensteiner. Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. In *Proc. of EDBT*, pages 299–310, 2009.

[39] A. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of PDIS*, pages 68–77, 1991.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of NSDI*, pages 2–2, 2012.

[41] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *Proc. of SOSP*, pages 423–438, 2013.

[42] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2015.

B. EVALUATION FOR ONLINE DATA AGGREGATION

We modify the three queries used in Section 6 to add aggregation functions for the online aggregation evaluation. For queries $Q5$ and $Q7$, we apply the `sum` aggregate function to the join results with the corresponding group-by operations as defined in the TPC-H specification [2]. For query *Band*, we add three different aggregate functions `count`, `min` and `max` simultaneously based on the group-by attribute of nation name *n_name*. BiStream adopts the two-phase aggregation approach as described in Section 3.4.2 to support online data aggregation. Figure 13 shows the throughput comparison between the three queries with and without the aggregation functions over the JB16 and JB16-CR4 settings. It can be seen that the aggregation operation exerts negligible impact on the throughput. This is reasonable, as each joiner task directly aggregates its join results in a pipeline fashion and thus is efficient.

Figure 14 shows the network bandwidth usage, when different aggregation strategies are adopted. Recall that BiStream adopts one batch optimization to pre-aggregate the join results in each joiner before generating the global view. This experiment is conducted over a JB16-CR4 setting using queries $Q5$, $Q7$ and *Band* with the input rates of 450K, 270K and 9K tuples per second respectively. From the results, we can see that the batch local view update approach greatly reduces the bandwidth usage for aggregation operation.

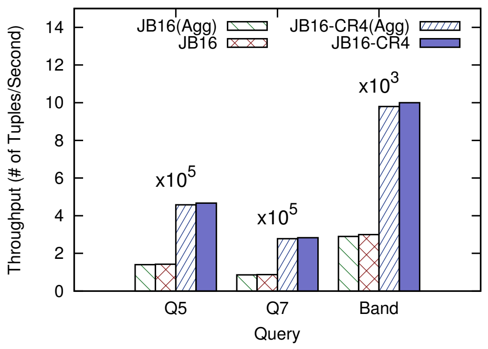


Figure 13: Throughput with and without aggregation.

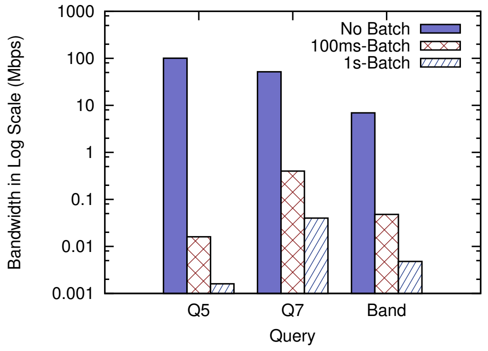


Figure 14: Bandwidth usage with aggregation.

C. MAXIMUM SIZE OF SLIDING WINDOW

Table 4 demonstrates the maximum window size that join-biclique and join-matrix can support in-memory sliding-window join processing. It includes the results for both queries $Q5$ and $Q7$ running on five different settings JB8, JB16, JB36, JM16 and JM36. For query $Q5$, all settings are evaluated under two different input rates of 50K and 80K tuples per second. For query $Q7$, all settings are evaluated with the input rates of 30K and 50K tuples per second. The results confirm that JB can support a larger window for in-memory processing than JM.

Table 4: Maximum size of sliding window.

Query	Rate	JB8	JB16	JB36	JM16	JM36
$Q5$	50K/s	278	549	1262	140	207
	80K/s	171	351	785	88	136
$Q7$	30K/s	572	1167	2636	292	439
	50K/s	341	697	1543	173	262

(Unit: Second)

D. RESULTS ON SCALABILITY

Figure 15 and Figure 16 provide maximum processing volume comparison between join-biclique and join-matrix when the number of units varies from 4 to 64 for queries $Q5$ and *Band* respectively. Similarly to the result for query $Q7$ provided in Section 6, JB shows better scalability than JM for $Q5$ and *Band*.

Figure 17 illustrates the maximum processing volume changes when the number of units varies from 4 to 64 with more partitioning schemes. As shown in the results, the maximum processing volume exhibits linear increment with the increasing number of units in JB.

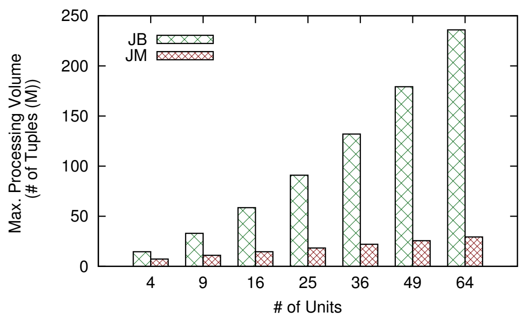


Figure 15: Scalability comparison on query $Q5$.

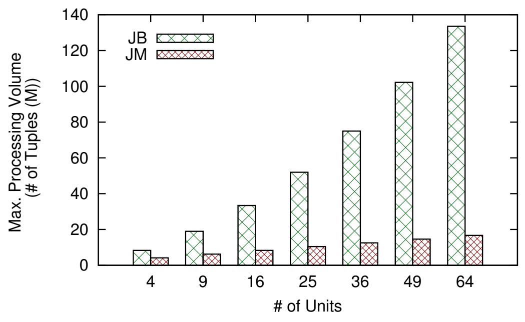


Figure 16: Scalability comparison on query $Band$.

E. RESULTS ON COMMUNICATION COST

Figure 18 and Figure 19 provide more experimental results of bandwidth usage for queries $Q7$ and $Band$ respectively. For query $Q7$, the bandwidth usage is evaluated by changing the input rates from 10K to 50K tuples per second. For query $Band$, the bandwidth usage is evaluated when the input rates vary from 200 to 1600 tuples per second. The results confirm that the ContRand routing strategy is able to reduce the bandwidth usage in JB, and JB offers a larger maximum processing volume while using competitive bandwidth as JM.

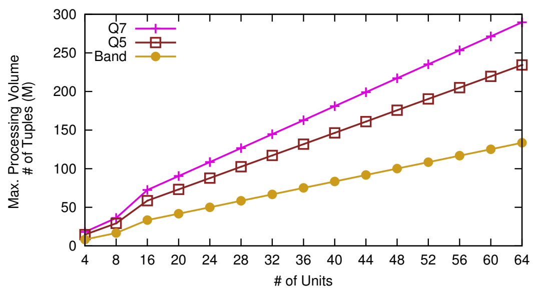


Figure 17: Scalability of join-biclique.

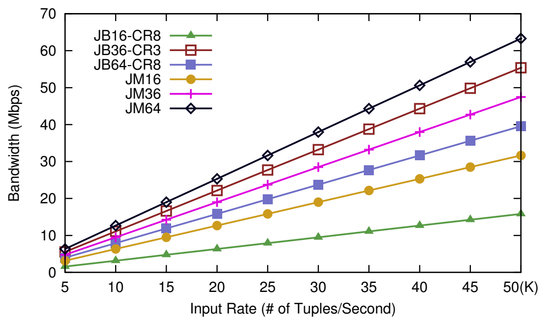


Figure 18: Bandwidth on query $Q7$.

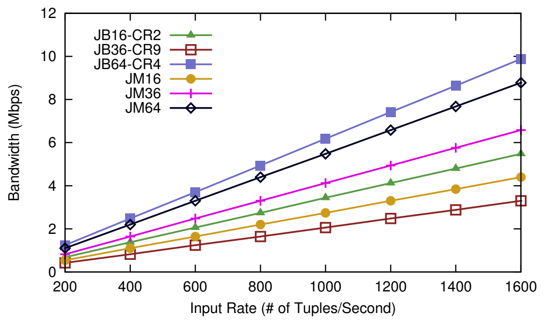


Figure 19: Bandwidth on query $Band$.