Whale: Efficient One-to-Many Data Partitioning in RDMA-Assisted Distributed Stream Processing Systems

Jie Tan, Hanhua Chen, Yonghui Wang, Hai Jin
National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computing Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
{jmaster,chen,yhw,hjin}@hust.edu.cn

ABSTRACT

To process large-scale real-time data streams, existing *distributed stream processing systems* (DSPSs) leverage different stream partitioning strategies. The one-to-many data partitioning strategy plays an important role in various applications. With one-to-many data partitioning, an upstream processing instance sends a generated tuple to a potentially large number of downstream processing instances. Existing DSPSs leverage an instance-oriented communication mechanism, where an upstream instance transmits a tuple to different downstream instances separately. However, in one-to-many data partitioning, multiple downstream instances typically run on the same machine to exploit multi-core resources. Therefore, a DSPS actually sends a data item to a machine multiple times, raising significant unnecessary costs for serialization and communication. We show that such a mechanism can lead to serious performance bottleneck due to CPU overload.

To address the problem, we design and implement Whale, an efficient RDMA (*Remote Direct Memory Access*) assisted distributed stream processing system. Two factors contribute to the efficiency of this design. First, we propose a novel RDMA-assisted stream multicast scheme with a self-adjusting non-blocking tree structure to alleviate the CPU workloads of an upstream instance during one-to-many data partitioning. Second, we re-design the communication mechanism in existing DSPSs by replacing the instance-oriented communication with a new worker-oriented communication scheme, which saves significant costs for redundant serialization and communication. We implement Whale on top of Apache Storm and conduct comprehensive experiments to evaluate its performance with large-scale real world datasets. The results show that Whale achieves 56.6× improvement of system throughput and 97% reduction of processing latency compared to existing designs.

CCS CONCEPTS

• **Computer systems organization**;

KEYWORDS

Distributed stream processing system, one-to-many data partition, remote direct memory access (RDMA)

ACM Reference Format:
Jie Tan, Hanhua Chen, Yonghui Wang, Hai Jin. 2021. Whale: Efficient One-to-Many Data Partitioning in RDMA-Assisted Distributed Stream Processing Systems. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476192

1 INTRODUCTION

With the ability to cope with the emerging big streams, *Distributed Stream Processing Systems* (DSPSs) have attracted much research efforts. To achieve high throughput and low latency, DSPSs exploit different parallel processing techniques. First, they exploit pipeline parallelism by abstracting a stream application as a *Directed Acyclic Graph* (DAG), where a vertex represents an operator and a directed edge represents a data transmission channel from an upstream operator to a downstream one. A DSPS deploys such an application topology on a cluster of servers and processes continuously incoming tuples in an efficient pipeline style. Second, a DSPS leverages data parallelism by generating multiple instances for an operator. Thus, the workloads of an operator are partitioned among multiple instances, which can take full advantage of multi-core architecture to improve system throughput. Figure 1 illustrates an example of a DSPS deployed on a cluster with four quad-core machines. Each machine contains one worker process which hosts four task threads as executing instances of an operator (we use the terms instance and task interchangeably in the rest of the paper).

Existing stream partitioning strategies mainly include shuffle grouping, key grouping, and all grouping [20]. The all grouping strategy is a typical one-to-many data partitioning strategy. Specifically, it lets an upstream instance send a tuple in the coming stream to all the downstream instances. Such a strategy plays vital roles in many big data applications, such as on-demand ride-hailing [19] and stream join applications [9].

Figure 2 examines the performance of the all grouping strategy which leverages one-to-many data partitioning. We emulate an on-demand ride-hailing application system atop Apache Storm [20], a popular distributed stream processing system. We use the dataset collected from Didi [1], the most popular on-demand ride-hailing platform in China. The dataset contains 13 billion trajectory records associated with six million drivers and 74 million passenger requests. Figure 4 shows the order dispatch on the on-demand ride-hailing platform. The source instance partitions the stream of drivers' locations into downstream order matching instances using a

*The corresponding author is Hanhua Chen (chen@hust.edu.cn).

SC '21, November 14–19, 2021, St. Louis, MO, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476192

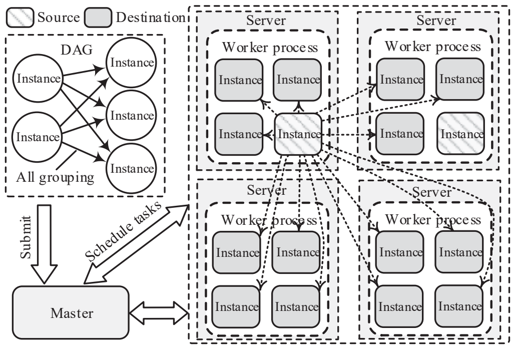


Figure 1: One-to-many data partitioning in a DSPS

hash-based strategy. Meanwhile, a tuple of the passenger request stream is sent to all the order matching instances, which perform a joining operation and return the best driver. We deploy the emulation system on a cluster with 30 machines, each with a 16-core 2.6 GHz Intel (R) Xeon (R) CPU and a 1 Gbps Ethernet NIC. In the experiment, we adjust the parallelism level (i.e., the number of matching instances) and examine the system performance. Figure 2a shows the system throughput decreases as the parallelism level increases. When the parallelism level reaches 480, the throughput becomes one-tenth of that when parallelism level = 30. Figure 2b shows the rapid rise of the processing latency. The results show a striking performance degradation of one-to-many data partitioning.

We further examine the CPU utilization of both the upstream and downstream instances. Figure 2c shows that the CPU utilization of the upstream instance rises sharply along with the increase of the parallelism level while the CPU utilization of a downstream instance changes slightly. When the parallelism level = 300, the CPU of the upstream instance becomes overloaded while the CPU of a downstream instance remains under-utilized. Figure 2d breaks down the CPU time of the upstream instance. It shows that two kinds of workloads dominate the CPU time, tuple serialization and packet processing with multi-layer network protocol during tuple transmission. Such CPU overload inspires the opportunity of exploiting the zero-copy *Remote Direct Memory Access* (RDMA) network [8, 23], whose kernel-bypass feature has the potentials to save the cost for kernel data copying and context switching in OS.

Using RDMA is not difficult. However, how to support efficient one-to-many data partition in a stream processing system with the assist of RDMA is not trivial. Yang et al. [24] design an RDMA-based Storm which replaces the underlying TCP/IP network of Apache Storm with the RDMA network. However, their system sequentially sends a tuple to multiple downstream instances for one-to-many data partitioning and still suffers from heavy serialization overhead and high processing latency. Behrens et al. [3] recently proposed an elaborate RDMA multicast protocol called RDMC [3]. The RDMC design leverages a static binomial tree structure to cope with large object transmission from one source node to multiple destination nodes [21]. However, their design cannot support efficient one-to-many partition for stream data due to stream dynamics [14].

We examine the performance of RDMC on our 30-node emulation system by setting up the Mellanox InfiniBand FDR (*Fourteen Data Rate*) 56 Gbps RDMA network.

In the examination, we create 480 matching instances and dynamically adjust the stream input rate according to the results by Mai et al. [14]. Figure 3a shows that the system throughput first increases with the input rate, then stops increasing when *input rate*≥12,000 tuples/s, and starts to decrease when *input rate*≥14,000 tuples/s. Figure 3b shows that the processing latency increases when *input rate*≥10,000 tuples/s. We can clearly see the blocking of the transfer queue when *input rate* = 25,000 tuples/s in the mini figure in Fig. 3a. However, we find sufficient computing resources for downstream instances. This demonstrates RDMC cannot support efficient one-to-many data partitioning for processing dynamic streams.

The challenges to design an efficient RDMA-assisted one-to-many data partitioning mechanism in a DSPS are in two aspects. First, the arriving rate of a stream in a DSPS is commonly dynamic [14], making the buffer queue difficult to support satisfying service availability. Tuple accumulation can cause buffer blocking in the presence of a sharp rising of the input rate. Second, large-scale data parallelisms in one-to-many data partitioning can lead to system bottleneck due to heavy CPU overloads.

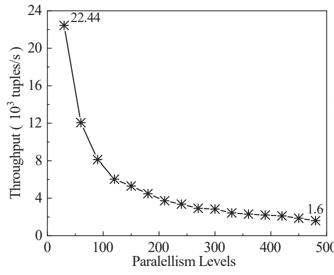
In this work, we propose Whale, a novel efficient RDMA-assisted one-to-many stream data partitioning design. Whale designs a novel self-adjusting non-blocking tree structure, which prevents the transfer queue of an instance from blocking due to stream dynamics. In addition, Whale designs and implements a new *worker-oriented* communication mechanism which replaces the instance-oriented design in previous DSPSs and significantly reduces the CPU overhead for serialization and communications. We implement Whale on top of Apache Storm and evaluate its performance using large-scale datasets from real-world systems. The results show that Whale greatly improves throughput and latency of existing designs.

The rest of the paper is structured as follows. Section 2 reviews the related work. Section 3 presents the Whale design. Section 4 describes the implementation details. Section 5 evaluates this design. Section 6 concludes the work.

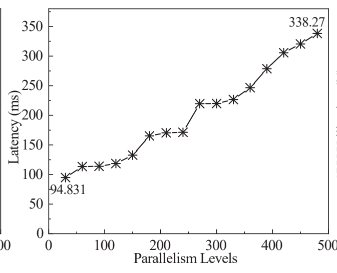
2 RELATED WORK

A DSPS models a stream application as a DAG and deploys an application topology on a cluster to achieve pipeline parallelism. It further generates multiple instances for a processing element to exploit data parallelism, and thus an upstream instance can partition workloads among different downstream instances. Popular DSPSs [4, 12, 13, 20] employ an unified instance-oriented communication mechanism for inter-instance tuple transmission and thus an application programmer can focus on implementing the specific logical function of an operation. Based on the instance-oriented communication mechanism, DSPSs leverage different stream partitioning strategies for different application requirements, including key grouping, shuffle grouping, and all grouping [20].

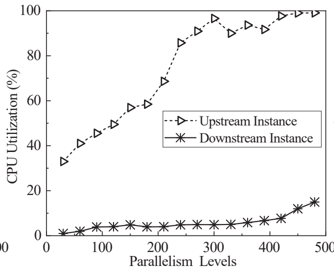
Previous efforts on improving the efficiency of stream data parallelism mainly focus on one-to-one style data partitioning strategies including key grouping and shuffle grouping [6, 7, 16]. To cope with skewed stream workloads, Gedik [7] proposes a frequency-based key grouping function to achieve load balance. Initially, the system uses a hash scheme to partition the keys from an upstream to downstream instances. During processing, the system uses a



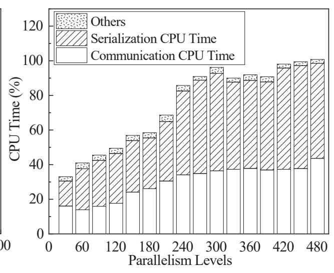
(a) Throughput with different parallelism levels



(b) Processing latency with different parallelism levels

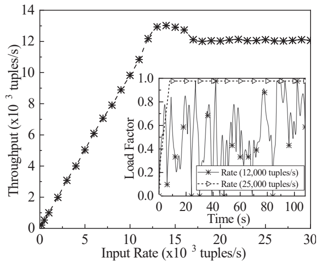


(c) CPU utilization of upstream and downstream instances

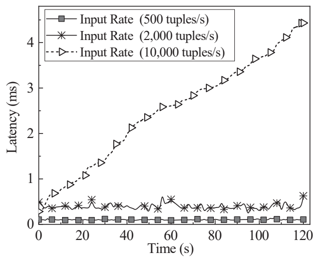


(d) Breakdown CPU time of upstream instance

Figure 2: Performance bottleneck in one-to-many data partitioning for stream processing



(a) Throughput and load factor with different input rates



(b) Processing latency with different input rates

Figure 3: Performance bottleneck in RDMC

lossy counting algorithm to identify high frequency keys. Based on this, it dynamically adjusts the mapping structure among keys and downstream instances to achieve load balance. However, as the frequency of keys fluctuates frequently, their design needs a costly switching operation. Nasir et al. [16] leverage the principle of power of two choices to achieve load balance during data partitioning. Specially, the system associates each key with a pair of two candidate downstream instances, while it sends an associated tuple to the one with lighter workload. However, the principle works well based on the assumption that the upstream instance selects two candidate downstream instances randomly and uniformly for any tuple. It is difficult for their design to meet such a strong requirement and hence it suffers from unsatisfied performance. Chen et al. [6] design an efficient probabilistic counter to identify hot keys and propose a differentiated partitioning scheme. It uses shuffle grouping for hot

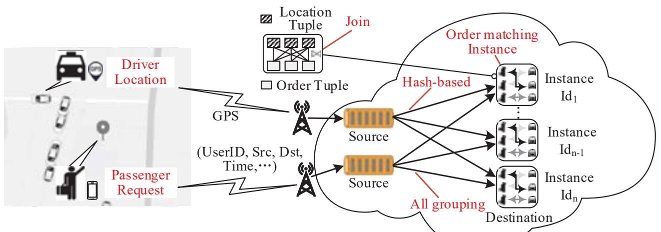


Figure 4: An on-demand ride-hailing system

keys to achieve load balance and uses key grouping for rare ones to avoid memory bottleneck.

Little is known about how to address at improving one-to-many data partitioning for stream processing. With the instance-oriented communication mechanism, existing systems use sequential transmission for one-to-many data partitioning, i.e., the upstream instance sends a tuple independently to all the downstream instances. This indeed suffers from system performance bottleneck in large-scale parallelisms due to the CPU overload for redundant serialization and communication (Figs. 2a and 2b).

RDMA [8, 23, 25] is an efficient remote memory access technology, whose zero-copy and kernel bypass characteristics can effectively save CPU resource during remote memory access. It has shown great potentials in designing large-scale distributed systems. Islam et al. [8] optimize the network communication in HDFS with RDMA. Their optimization focuses on HDFS write, which is more network intensive than other operations. They further design a DataStreamer to determine whether to use RDMA communication library or socket. Wei et al. [22] leverage RDMA in in-memory distributed transaction processing. They exploit the mutex feature of RDMA when accessing the same remote memory to ensure the serializability among concurrent transactions. Behrens et al. [3] propose RDMC for one-to-many large object transmission with RDMA network, where the sender splits a large object into multiple data blocks. The system organizes all the receivers as a static binomial tree structure and uses a relaying pattern to reduce the transmission latency. However, such a design is not suitable for one-to-many data partitioning for stream processing. It can suffer from blocking in the transfer queue in the presence of dynamic streams (Fig. 3a and Fig. 3b). RDMA also has potentials in lightweight data transmission, because TCP/IP network cannot reach saturation due to heavy CPU overheads [27]. Yang et al. [24] propose RDMA-based Storm which replaces traditional TCP/IP network of Apache Storm with RDMA. Their design offloads the communication overhead of CPU to the network card based on the zero-copy character of RDMA. However, the system still suffers from performance bottleneck in one-to-many data partitioning. Steffen et al. [26] use the high throughput RDMA network to boost the ingestion rate and improve the overall system throughput. However, the RDMA network does not solve the overload problem of the upstream instance in one-to-many data partitioning. Little is known about how to improve one-to-many data partitioning in a DSPS. To the best of our

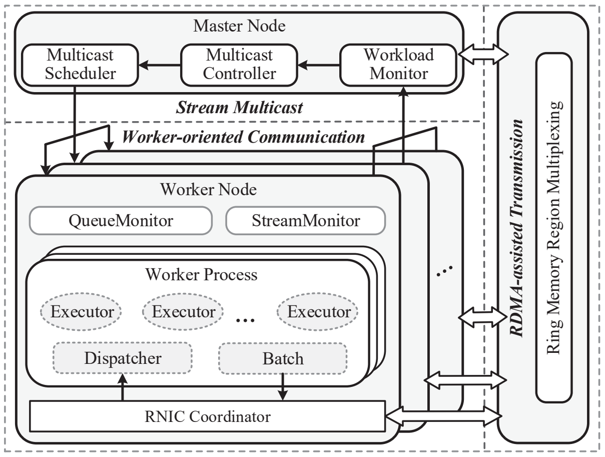


Figure 5: Architecture of Whale

knowledge, we are the first to address the problem of one-to-many data partitioning in a DSPS by using an RDMA-assisted design.

3 SYSTEM DESIGN

In this section, we first briefly introduce the system overview. Then, we describe the RDMA-assisted stream multicast and the worker-oriented communication of Whale in detail.

3.1 System Overview

As we analyzed above, the root cause of the performance bottleneck in one-to-many data partitioning in a DSPS lies in the CPU overload of the upstream instance because of the redundant serialization and communication. In order to reduce the overhead of the upstream instance, our Whale system designs and implements an RDMA-assisted one-to-many stream data partitioning strategy.

First, Whale proposes a novel self-adjusting non-blocking multicast tree structure and designs an RDMA-assisted stream multicast scheme based on the structure. In such a design, the RDMA network can effectively alleviate the CPU cost for communication while the self-adjusting non-blocking tree structure prevents the transfer queue of the upstream instance from blocking in the presence of stream dynamics during one-to-many data partitioning. Second, Whale designs a new worker-oriented communication mechanism and replaces the existing instance-oriented communication mechanism in popular DSPSs to further save the overhead for redundant serialization and communication.

Figure 5 shows the architecture of Whale, which includes two main components, stream multicast and worker-oriented communication. The stream multicast component fully exploits the zero-copy RDMA network to efficiently process highly dynamic streams.

In the stream multicast component, Whale constructs the proposed efficient non-blocking multicast tree structure to support efficient one-to-many data partitioning. The system can dynamically adjust the structure to copy with stream dynamics. Specially, the stream multicast scheduler elaborately chooses the maximum out-degree of the non-blocking multicast tree structure for the incoming stream and minimizes the average multicast latency. The system workload monitor periodically monitors the current workloads of the source instance while the multicast controller adjusts the multicast structure accordingly. The worker-oriented communication component addresses at reducing the redundant serialization and communication. The *batch* component packages the IDs with the data item into a *BatchTuple* and serializes the *BatchTuple* for multicasting. The *dispatcher* deserializes the *BatchTuple* and dispatches the receiving tuple to the locally hosted destination instances according to the obtained instance IDs.

3.2 RDMA-assisted One-to-many Stream Data Partitioning

An efficient RDMA-assisted one-to-many data partitioning design has two main requirements: 1) the structure should be adjustable for stream dynamics, so that it can prevent the transfer queue from blocking; 2) the transmission latency should be minimized when organizing the instances to relay tuples. Whale organizes all the instances into a relay-based structure for tuple multicasting. The source establishes an RDMA channel to each directly cascading instance and sends a tuple to every cascading instance sequentially. A cascading instance also builds up RDMA channels to other instances and acts as a relay node. Each instance T_i ($0 \leq i \leq n$) relays the tuple to a certain number of instances. Upon an instance finishes sending a tuple to all directly cascading instances, it continues with a following-up tuple.

The multicast latency for a tuple is the time from when it enters the source to the time when the last destination instance receives it. The latency in a single hop includes four parts: 1) the time for serialization, 2) the time for the tuple to wait for scheduling in the RDMA transfer queue, 3) the time for encapsulating the tuple into an RDMA *work request* for sending, and 4) the time for transmitting the tuple through RDMA channel. The multicast latency is the sum of the processing latency through all the hops.

In stream multicast, a source S needs to send every tuple to n destination instances $T = \{T_1, T_2, ..., T_n\}$. Given a transfer queue with capacity Q, as the input rate λ changes dynamically, the queue should not block and the average multicast latency should be minimized. In Whale, we propose a self-adjusting non-blocking multicast structure with elaborate number of cascading instances based on the current workloads.

3.2.1 *Determine the Maximum Out-degree.*

Intuitively, the out-degree of an instance determines the performance of the relay-based stream multicast structure, because an instance cannot process the following-up tuple until it sends the current tuple to all its directly cascading instances. With a larger out-degree, a source instance actually processes the incoming stream tuples at a lower rate. If the processing rate of the source instance cannot keep up with the input rate, the length of the transfer queue increases. Eventually, the full transfer queue becomes the performance bottleneck of the system. Therefore, it is important to determine the maximum out-degree d^* for the source instance.

As the stream tuples arrive randomly with an unpredictable rate, we analyze the dynamic system with the queuing theory and formally model the process as an M/D/1 system [11]. Let λ denote the stream input rate at the source instance S. With an out-degree of d_0, S needs to generate d_0 replicas and send them to the corresponding RDMA channels. Each replica has a processing time of t_e. Thus, the processing rate μ of the source instance can be computed by Eq. (1),

$$\mu = 1/(d_0 \cdot t_e) \tag{1}$$

The average queue length of S can be computed by Eq. (2),

$$E(L) = \frac{\lambda^2}{2\mu(\mu - \lambda)} + \frac{\lambda}{\mu} \tag{2}$$

When $\mu > \lambda$, the workload of the transfer queue can be alleviated and the out-degree of S can be increased. When $\mu < \lambda$, the length of the transfer queue continuously increases. To prevent the transfer queue from blocking, Whale decreases the out-degree of S to increase the processing rate. Formally, we need to ensure that the average queue length of the source instance is not greater than its maximum capacity Q. Deducing from $E(L) \leq Q$, we have

$$d_0 \leq \frac{2Q}{\lambda \cdot t_e \cdot (Q + 1 - \sqrt{Q^2 + 1})} \tag{3}$$

We obtain the maximum d^* when the equality holds. In practice, it is important to collect the system statistical information to achieve d^*. In Whale, we implement a *statistics monitoring* module to bridge the gap between the queue model and a real-world system. The monitor collects the system statistics to obtain λ and t_e following the technique widely used in previous designs [15, 17]. Section 4 will present more details of the module.

3.2.2 Non-blocking Multicast Tree. Whale designs a non-blocking multicast tree which restricts a binomial multicast tree with d^*. Whale dynamically adjusts d^* to prevent the system from blocking. The non-blocking multicast tree has multiple logic layers. Algorithm 1 describes the construction of the non-blocking multicast tree. Given a set T ($|T| = n$) of destination instances. Initially, the multicast tree only contains S. Whale iteratively constructs the instances of each layer (lines 5 - 15), with the out-degree of each instance limited within d^*. In each round, an instance with out-degree less than d^* can connect to a new instance in T (lines 9 - 13).

Table 1: Notations

Notations	Description
S	the source instance
T	a set of destination instances
T_i	the ith destination instance
$T_i \longrightarrow T_j$	the RDMA channel between T_i and T_j
T_{i-j}	the jth instance on the ith layer
d_i	the number of cascading instances of T_i ($T_0 = S$)
d^*	the maximum affordable out-degree
t_e	the tuple processing time for a hop
Q	the capacity of the transfer queue
λ	the stream input rate in the M/D/1 queue system
μ	the processing rate in the M/D/1 queue system
$E(L)$	the average queue length in the M/D/1 queue system
$v_{in}(t)$	the input rate of the source instance at time t
$v_{out}(t)$	the processing rate of the source instance at time t
$v(t)$	the queue length changing rate at time t
$q(t)$	the current queue length at time t
M	the maximum affordable input rate
T_{down}	the threshold for the negative scale-down
T_{up}	the threshold for the active scale-up

Algorithm 1: Build Non-blocking Multicast Tree

```
Input: the source instance $S$, the number of destination
       instances $n$, $d^*$
Output: topology for multicasting
1  $topology \leftarrow$ an initial empty DAG;
2  $list \leftarrow$ an initial empty set;
3  $list.add(S)$;
4  $topology.addVertex(S)$;
5  while $list.size \leq n$ do
6      $size \leftarrow list.size$;
7      for $i = 1$ to $size$ do
8          $t = list.get(i)$;
9          if $t.outdegree < d^*$ then
10             $instance \leftarrow tClass.newInstance$;
11             $topology.addVertex(instance)$;
12             Let $instance$ be the cascading instance of $t$;
13             $list.add(instance)$;
14         if $list.size > n$ then
15             return $topology$.
```

If the out-degree of an instance reaches d^*, it stops connecting to any instances. The procedure ends until all the instances in T are connected (lines 14 - 15).

Figure 6 shows an example for constructing a non-blocking multicast tree with $|T| = 7$ and $d^* = 2$. For simplicity, we use T_{i-j} to denote the jth instance on the ith logical layer. Initially, the structure only contains S. The first logical layer contains only T_{1-1}, which is the directly cascading instance of S. On the second logical layer, S connects to one instance, T_{2-1}, while T_{1-1} connects to T_{2-2}. If the out-degree of an instance on the multicast tree reaches $d^* = 2$, it cannot connect to any new instances remaining in T. For example, when constructing the third logical layer, if S connects to one more instance, the out-degree of S will exceed $d^* = 2$. Thus, Whale lets T_{1-1} connect to a new instance, denoted as T_{3-1}. In the same way, T_{2-1} connects to T_{3-2}; T_{2-2} connects to T_{3-3}. On the fourth logical layer, T_{2-1} connects to T_{4-1}.

Figure 6 also illustrates the stream multicast procedure based on the constructed multicast tree. In the first time unit, S sends the tuple t_1 to the instance T_{1-1}. In the second time unit, S and T_{1-1} send t_1 to T_{2-1} and T_{2-2}, respectively. In the third time unit, t_2 arrives and S starts to send t_2 to T_{1-1}. Meanwhile, T_{1-1} sends t_1 to T_{3-1}; T_{2-1} sends t_1 to T_{3-2}; and T_{2-2} sends t_1 to T_{3-3}. In the fourth time unit, S sends t_2 to T_{2-1}; T_{1-1} sends t_2 to T_{2-2}; and T_{2-1} sends t_1 to T_{4-1}. Thus, Whale completes the multicast of t_1. We analyze the non-blocking multicast tree as follow.

Definition 1 (Maximum affordable input rate) *The maximum affordable input rate M is defined as the maximum input rate that the system can afford to ensure the queue length of the source $E(L)$ does not exceed its maximum capacity Q.*

Definition 2 (Multicast capability) *The multicast capability $L(t)$ is defined as the number of destination instances that the source instance multicasts to in the t^{th} time unit.*

Theorem 1. *The maximum affordable input rate M of a non-blocking multicast tree is inversely proportional to the out-degree of the source instance* d_0, i.e., $M \propto 1/d_0$.

Proof. Deducing from $E(L) \leq Q$ and $\mu = 1/(d_0 \cdot t_e)$, we have,

$$0 < \lambda \leq \frac{1}{d_0 \cdot t_e}(Q + 1 - \sqrt{Q^2 + 1}) \tag{4}$$

Thus, with M for a source instance, we obtain,

$$M = \frac{1}{d_0 \cdot t_e}(Q + 1 - \sqrt{Q^2 + 1}) \tag{5}$$

Since $Q + 1 > \sqrt{Q^2 + 1}$, M is inversely proportional to d_0, i.e., $M \propto 1/d_0$. Theorem 1 is thus proved. ■

In the following, we further compare the maximum affordable input rate of our proposed non-blocking multicast tree $M_{nonblock}$ with that of a traditional binomial multicast tree $M_{binomial}$ [3]. The out-degree of the source instance with a traditional binomial multicast tree is $\lceil \log_2(n+1) \rceil$ given a number of n destination nodes. However, the out-degree of the source instance d_0 in our proposed non-blocking multicast tree is $min\{d^*, \lceil \log_2(n+1) \rceil\}$. If $d^* > \lceil \log_2(n+1) \rceil$, all the destination instances will be connected to the non-blocking multicast tree when the out-degree of the source instance increases to $\lceil \log_2(n+1) \rceil$. Based on Theorem 1, the maximum affordable input rate of the non-blocking multicast tree is not less than that of the binomial multicast tree, i.e., $M_{nonblock} \geq M_{binomial}$. It is not difficult to obtain $\frac{M_{nonblock}}{M_{binomial}} = \lceil \log_2(n+1) \rceil / d_0$.

Theorem 2. *The multicast capability of a non-blocking multicast tree $L(t)$ and the out-degree of the source instance d_0* $(d_0 \in \{1, 2, ..., \lceil \log_2(n+1) \rceil\})$ *are positively correlated.*

Proof. Given a total number of n destination instances in T. When $d^* \geq \lceil \log_2(n+1) \rceil$, the non-blocking multicast tree is modeled as a binomial tree multicast structure. The out-degree of source instance d_0 is $\lceil \log_2(n+1) \rceil$. In the binomial tree multicast structure, each completed instances can transmit to one new destination instance in T at each time unit. Thus, if $d^* \geq \lceil \log_2(n+1) \rceil$, we have,

$$L(t) = 2L(t-1) \tag{6}$$

When $d^* < \lceil \log_2(n+1) \rceil$, d^* limits the maximum degree of the non-blocking multicast tree. If the out-degree of S is not larger than d^*, S transmits a tuple to one new destination instance following the binomial tree multicast structure. After the out-degree of S reaches d^*, each completed instance on the current multicast tree with the out-degree less than d^* can transmit the tuple to one new destination instance in T at each time unit. There are $L(t-1) - L(t-d^*-1)$ completed instances on the current multicast tree with the out-degree less than d^*. Thus, if $d^* < \lceil \log_2(n+1) \rceil$, we have,

$$L(t) = \begin{cases} 2L(t-1), & t \leq d^* \\ 2L(t-1) - L(t-d^*-1), & t > d^* \end{cases} \tag{7}$$

Given two non-blocking multicast trees with their maximum out-degrees d_1^* and d_2^* satisfying $d_1^* < d_2^* < \lceil \log_2(n+1) \rceil$, the corresponding multicast capability $L_1(t)$, $L_2(t)$ have,

$$L_1(t) - L_2(t) = L_1(t - d_2^* - 1) - L_2(t - d_1^* - 1) \tag{8}$$

For the multicast performance of the non-blocking multicast tree, there is a positive correlation between $L(t)$ and the time t. Since $t - d_1^* - 1 > t - d_2^* - 1$, we have,

$$L_1(t - d_1^* - 1) > L_2(t - d_2^* - 1) \tag{9}$$

Combining Eq. (8) and Eq. (9), we obtain,

$$L_1(t) < L_2(t) \tag{10}$$

Thus, there is a positive correlation between $L(t)$ and d_0 $(d_0 \in \{1, 2, 3, ..., \lceil \log_2(n+1) \rceil\})$. Thus proved. ■

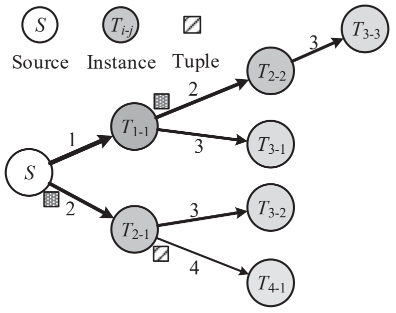


Figure 6: Non-blocking multicast tree with $|T| = 7$

3.3 Queue-based Self-Adjusting Mechanism

To cope with stream dynamics, we model the transfer queue of S as a pool with a floor drain at the bottom. The stream continuously pours data into the pool, while the floor drain releases data at the bottom. If the input rate is higher than the output rate, the waterline rises. On the contrary, it declines. For dynamic input rate, we can prevent the pool from becoming full by adjusting the output rate.

Similarly, Whale prevents the amount of waiting tuples from exceeding the maximum capacity of the transfer queue Q. Whale sets a warning waterline l_w and periodically monitors the current workloads of the transfer queue at each time interval Δt. When the monitor detects that the waterline rises and exceeds l_w, Whale adjusts the current multicast structure by decreasing the out-degree of S to increase the processing rate. We call such a strategy *negative scale-down*. On the contrary, if the monitor detects that the waterline drops rapidly under l_w, Whale adopts an *active scale-up* strategy to increase the out-degree of S. Besides, Whale can determine d^* according to the stream input rate λ. We use the notation $l' \rightarrow l$ to denote the length change of the transfer queue from l' to l. Whale determines whether to adjust the multicast structure or not based on the following rules.

Negative scale-down. Supposing the current queue length increases by ΔL ($\Delta L = l - l'$), Whale decreases the out-degree of the source instance appropriately if the ratio of ΔL to $l_w - l$ is greater than or equal to a certain threshold T_{down}, i.e., $\Delta L/(l_w - l) \geq T_{down}$.

Active scale-up. Supposing the current queue length decreases by ΔL ($\Delta L = l' - l$), Whale increases the out-degree of the source instance appropriately if the ratio of ΔL to l' is greater than or equal to a certain threshold T_{up}, or the transfer queue is empty, i.e., $\Delta L / l' \geq T_{up}$ or $l = l' = 0$.

We analyze the self-adjusting mechanism of Whale as belows. We use the notations $v_{in}(t)$ and $v_{out}(t)$ to denote the stream input and output rates at time t, $q(t)$ to denote the queue length at t. The queue length changing rate $v(t)$ is computed by: $v(t) = v_{in}(t) - v_{out}(t)$.

The queue length $q(t)$ increases when $v(t) > 0$, decreases when $v(t) < 0$, and remains unchanged when $v(t) = 0$. The change of the queue length from t to $t + \Delta t$ is computed by $\int_t^{t+\Delta t} v(t)\, dt$. For simplicity, we give the following definition.

Definition 3 (Baseline dynamic switch) *The baseline dynamic switch is defined as adjusting the multicast structure when the current queue length increases to* l_w.

Theorem 3. *The maximum queue length of the negative scale-down strategy is less than that of the baseline dynamic switch.*

Proof. Given an initial time t_0 with an empty queue. For the *baseline dynamic switch* strategy, the queue length increases to l_w at $t_0 + \Delta t$, and the system begins to adjust the multicast structure. The current queue length $q(t_0 + \Delta t)$ is l_w. We have,

$$l_w = \int_{t_0}^{t_0+\Delta t} v_{in}(t)\, dt - \int_{t_0}^{t_0+\Delta t} v_{out}(t)\, dt. \tag{11}$$

Due to the delay time $t^{'}$ of the dynamic switching, the stream output rate $v_{out}(t)$ of S will not increase immediately. The length of the queue keeps increasing in a short period of time. The maximum queue length L_{base} can be computed by

$$\begin{aligned} L_{base} &= l_w + \int_{t_0+\Delta t}^{t_0+\Delta t+t^{'}} v_{in}(t)\, dt - \int_{t_0+\Delta t}^{t_0+\Delta t+t^{'}} v_{out}(t)\, dt \\ &= \int_{t_0}^{t_0+\Delta t+t^{'}} v_{in}(t)\, dt - \int_{t_0}^{t_0+\Delta t+t^{'}} v_{out}(t)\, dt. \end{aligned} \tag{12}$$

Suppose that the *negative scale-down* strategy starts to run at t^*. According to the *negative scale-down* strategy, we have,

$$\frac{v(t^*)}{l_w - q(t^*)} \geq T_{down} \tag{13}$$

Combining Eq. (11) and Eq. (13), we obtain,

$$q(t^*) \leq q(t_0 + \Delta t) \tag{14}$$

For the *negative scale-down* strategy, $q(t)$ is proportional to time t, i.e., $q(t) \propto t$. Thus, we have,

$$t_0 \leq t^* \leq t_0 + \Delta t \tag{15}$$

The current queue length $q(t^*)$ can be computed by

$$q(t^*) = \int_{t_0}^{t^*} v_{in}(t)\, dt - \int_{t_0}^{t^*} v_{out}(t)\, dt \tag{16}$$

Considering the switching delay, the maximum queue length $L_{negative}$ for the *negative scale-down* switch is

$$\begin{aligned} L_{negative} &= q(t^*) + \int_{t^*}^{t^*+t^{'}} v_{in}(t)\, dt - \int_{t^*}^{t^*+t^{'}} v_{out}(t)\, dt \\ &= \int_{t_0}^{t^*+t^{'}} v_{in}(t)\, dt - \int_{t_0}^{t^*+t^{'}} v_{out}(t)\, dt \end{aligned} \tag{17}$$

From Eq. (15) and Eq. (17), we have,

$$\begin{aligned} L_{negative} &\leq \int_{t_0}^{t_0+\Delta t+t^{'}} v_{in}(t)\, dt - \int_{t_0}^{t_0+\Delta t+t^{'}} v_{out}(t)\, dt \\ &= L_{base} \end{aligned} \tag{18}$$

Theorem 3 is thus proved. ■

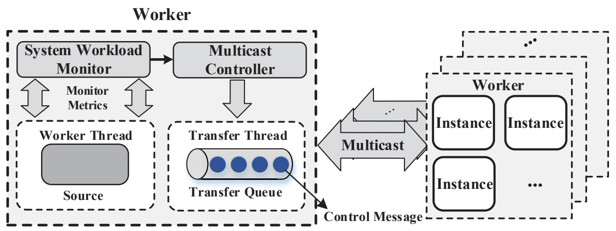


Figure 7: Dynamic switching mechanism

3.4 Dynamic Switching Mechanism

Whale dynamically adjusts the multicast structure according to the current d^*. It can rapidly reorganize the positions of the destination instances to adapt to a new maximum out-degree d^* based on the current multicast structure. Specifically, Whale dynamically reorganizes the partial destination instances by disconnecting certain connections and establishing new connections based on the current multicast structure without significant change.

Figure 7 shows the dynamic switching of Whale's non-blocking multicast structure. The system workload monitor monitors the workloads of the transfer queue while the multicast controller determines the structure adjustment. When determining to adjust, a *ControlMessage* is generated and multicast to all the destination instances. The *ControlMessage* contains the current status (*negative scale-down* or *active scale-up*) and the control configuration, which is used to inform destination instances to disconnect or establish a new connection for reorganizing the structure. A relay instance stores the structure of the multicast tree with *ControlMessage* and obtains its direct cascading instances from the structure. We describe the dynamic switching as follows.

Negative scale-down. The switching algorithm traverses the multicast tree from S to the maximum layer and marks destination instances which need to be adjusted. If the out-degree of an instance is larger than d^*, the algorithm marks the sub-tree which leads the instance to exceed d^*. Then, it searches from S to the maximum layer for a suitable position to insert the marked instance. If an instance has an out-degree less than d^*, the algorithm moves the marked instance to this new position. The procedure ends until all the marked instances reconnect to the multicast tree.

Figure 8 shows two examples of the dynamic switching strategies. In the example of the dynamic switching mechanism for negative scale-down (Fig. 8a), d^* changes from three to two. The algorithm traverses the multicast tree from S to the third layer and marks the

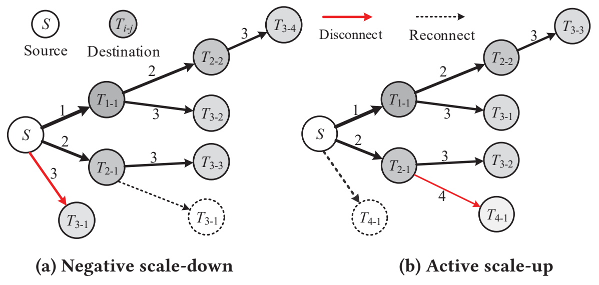


Figure 8: Example of dynamic switching

instances which need to be adjusted. In this example, d^* is larger than two. Thus, Whale collects T_{3-1} and re-assigns it. Finally, T_{3-1} connects to T_{2-1} whose out-degree is less than d^*. Specifically, T_{3-1} disconnects from S and re-connects to T_{2-1}.

In the following, we analyze the negative scale-down mechanism.

Definition 4 (Stream input loss) *During the dynamic switching phase, the length of the transfer queue continues to increase with the continually coming tuples. Eventually, it may cause the transfer queue overflow and tuple loss.*

Theorem 4. *Dynamic switching for negative scale-down will not lead to the stream tuple loss if the dynamic switching delay, T_{switch}, satisfies the following condition,*

$$T_{switch} < \frac{Q - q(t^*)}{v_{in}(t^*)} \tag{19}$$

where *dynamic switching for negative scale-down* is triggered at t^* when the current queue length rises to $q(t^*)$.

Proof. When the dynamic switching mechanism is triggered, the remaining capacity of the transfer queue is $Q - q(t^*)$. Since the stream output rate has decreased to zero, the queue becomes fully occupied after $\frac{Q-q(t^*)}{v_{in}(t^*)}$. Thus, the necessary and sufficient condition without stream input loss in *dynamic switching for negative scale-down* is $T_{switch} < \frac{Q-q(t^*)}{v_{in}(t^*)}$. Theorem 4 is thus proved. ■

Active scale-up. The switching algorithm traverses the multicast tree from the last destination instance to S. It searches for a suitable position to insert the rescheduled instance from S to the maximum layer. If an instance has an out-degree less than d^*, it can connect a direct cascading instance. The switching process moves the rescheduled instance to this new position. The procedure ends until the original position and the new position of the rescheduled instance are on the same logical layer.

Figure 8b shows an example of *dynamic switching for active scale-up*, where d^* changes from two to three. The algorithm traverses the multicast tree from the fourth layer to S. Thus, Whale collects T_{4-1} and re-assigns it. Finally, T_{4-1} disconnects from T_{2-1} and connects to S because the out-degree of S is less than d^* ($d^* = 3$).

In the following, we theoretically analyze the active scale-up mechanism. We first give the following definition.

Definition 5 (Multicast rate) *The multicast rate γ is the number of destination instances that S multicasts to in a unit time.*

Theorem 5. *Dynamic switching for active scale-up will improve system multicast performance if the number of multicast tuples X satisfies the following condition,*

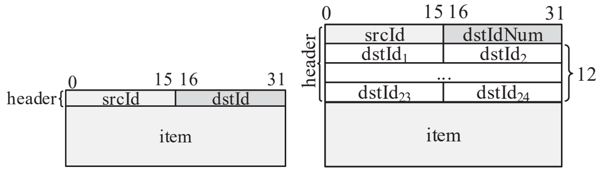


Figure 9: Tuple formats in Apache Storm and Whale

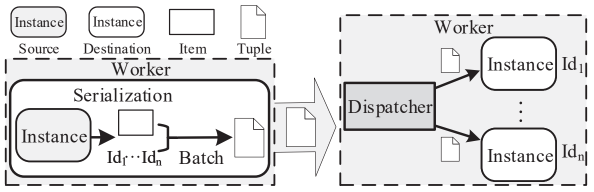


Figure 10: Worker-oriented communication mechanism

$$X > \frac{\gamma\gamma' T_{switch}}{\gamma - \gamma'} \tag{20}$$

where the multicast rates before and after dynamic switching are γ' and γ, respectively; T_{switch} is the delay of dynamic switching; X is the number of multicast tuples.

Proof. When the dynamic switching is triggered, the system multicast rate increases from γ' to γ. Due to the delay of dynamic switching T_{switch}, the system multicast performance will not immediately catch up with the multicast performance before dynamic switching. Thus, the necessary and sufficient condition to make the system multicast performance exceed the multicast performance before dynamic switching is $\frac{X}{\gamma'} > \frac{X}{\gamma + T_{switch}}$. Thus, we have $X > \frac{\gamma\gamma' T_{switch}}{\gamma - \gamma'}$. Thus proved. ■

3.5 Worker-oriented Communication

Whale proposes a new worker-oriented communication mechanism. For one-to-many data partitioning, the source instance sends a tuple to the worker process hosting multiple instances instead of separately sending to the instances. The receiving worker locally dispatches the tuple to the hosted designated instances. Figure 9b shows Whale's tuple format, where *BatchTuple* contains a data item and the IDs of destination instances. The sources instance packages the IDs of the destination instances hosted on the same worker with the data item into a *BatchTuple*, serializes *BatchTuple* only once, and sends it to the destination worker. Thus, Whale saves significant costs for redundant serialization and communication. Figure 10 illustrates the worker-oriented communication mechanism. The *batch* component generates *BatchTuple*. Then, the *batch* component serializes the data item and packages the serialized data item with the IDs into a *WorkerMessage*. In the destination worker, the *dispatcher* component receives the *WorkerMessage* from the source instance, deserializes the data item and obtains the ID of the destination instances, and dispatches the data item locally to the destination instances according to the obtained IDs.

4 IMPLEMENTATION

We implement Whale atop Apache Storm [20] and make the source code publicly available[1]. Our design is also applicable to other DSPSs, e.g., Flink [4], Samza [12], and Heron [13], which support one-to-many data partitioning and use instance-oriented communication mechanism. Besides, we also implement a general channel-oriented communication framework[2].

[1]https://github.com/CGCL-codes/Whale
[2]https://github.com/CGCL-codes/WhaleRDMAChannel

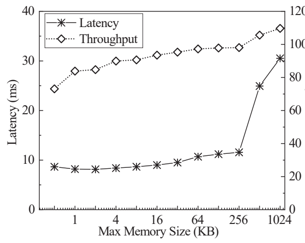


Figure 11: System performance with different MMS

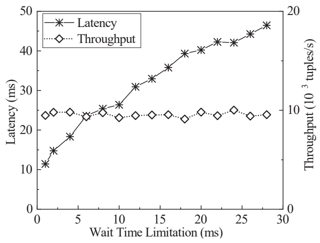


Figure 12: System performance with different WTL

Whale uses two types of RDMA operations, two-sided send/recv and one-sided read/write [18], and chooses suitable operations for different scenes. For multicast, a destination node uses RDMA read to fetch data from a source. To avoid redundant RDMA operations, Whale implements a ring memory region structure to maintain sequential access and obtain the data address for destination nodes. The control messages disconnect or establish connections to switch the non-blocking multicast tree. Besides, the multicast controller sends the control messages only when needed. It cannot obtain the message address with the help of the ring memory region structure. Hence, Whale uses send/recv operations to transfer these messages.

Whale further implements two mechanisms, *Stream Slicing* and *Ring Memory Region Multiplexing*. The sender maintains a transfer buffer. When the data size in the transfer buffer reaches *Max Memory Size* (MMS), the sender assembles it into an RDMA *work request* and sends it out. The sender also has a timer with a *Wait Time Limit* (WTL) to count the waiting time of the earliest waiting tuple. When the timer expires, the data will also be sent out. The timer will be reset when an RDMA *work request* is consumed by *RDMA Network Interface Card* (RNIC).

In order to achieve optimized settings of MMS and WTL, we examine the system performance with different values of them on our emulation system. We first examine the performance of Whale with MMS varying from 512B to 1,024KB. The result in Fig. 11 shows that the processing latency increases slightly with MMS when MMS < 256KB and increases significantly when MMS > 256KB. While the system throughput increases gradually as MMS increases. This is because with a larger MMS, Whale sends data faster but the the waiting time increases for the buffer to be full. In order to achieve the system throughput as high as possible and a lower latency, we can set MMS to 256KB in Whale. We then examine the performance of Whale with WTL varying from 1ms to 30ms. Figure 12 shows the throughput decreases slightly but the processing latency increases significantly as WTL increases. In order to reduce latency as much as possible, we set WTL to 1ms in Whale.

To avoid frequent registering and recycling *memory region* of RNIC, the sender and receiver register a continuous address space from RNIC. They both contain a head and a tail pointers to jointly specify the address space for data production/consumption. Whale models the continuous address space as a ring, where each memory region can be reused after consumed by the RNIC coordinator.

Whale constructs an RDMA-assisted non-blocking multicast tree. The multicast controller periodically monitors the length of the transfer queue, the tuple emit time, and the number of tuples arriving in a unit time. At first, the multicast controller initializes the transfer queue monitor *QueueMonitor* and the stream tuple monitor *StreamMonitor*. Then, the *QueueMonitor* measures each tuple's emit time in the transfer queue and obtains the tuple processing time t_e. It also monitors the length change of the transfer queue. The *StreamMonitor* measures the number of tuples arriving in a unit time and obtains the stream input rate λ. Based on the statistics, the multicast controller can determine whether to adjust the stream multicast structure or not (Section 3.3).

If the multicast controller determines to adjust the structure, S multicasts *StatusMessage* to all the destination instances to notify them to perform dynamic switching with either *negative scale-down* or *active scale-up*. Next, the multicast controller generates a *ControlMessage* by the dynamic switching algorithm (Section 3.4). To improve efficiency, the multicast controller first sends the *ControlMessage* to the destination instances which need to disconnect or establish new connections. Once receiving the notification, they disconnect from the corresponding instance or establish a new connection. If the destination instance completes dynamic switching, it sends an *ACK* signal to S. After S receives *ACKs* from all these destination instances, the dynamic switching completes. The multicast controller sends the *ControlMessage* to other destination instances as the streaming tuples being processed.

Whale implements the worker-oriented communication as follows. We redesign the format of tuples in Apache Storm as shown in Figs. 9a and 9b. Whale only serializes the data item once and inserts the multiple IDs of the destination instances residing on the same worker into the header. We recompute the processing rate by $\mu = 1/(d \cdot t_d + t_s)$, where t_s denotes the tuple serialization time and t_d denotes the scheduling time. In the following, we describe the procedures of tuple sending and tuple receiving in greater detail.

The source instance first encapsulates a tuple into a *BatchTuple* with dstIds. Then, it puts the *BatchTuple* into the send queue of the executor. Next, the sending thread of the executor obtains the *BatchTuple* from the send queue and serializes the data item of the *BatchTuple*. A *WorkerMessage* is generated, pushed into the transfer queue, and sent to the corresponding destination worker according to the destination task Ids. Each worker has a specialized receiving thread listening on the communication port. Once the receiving thread receives a *WorkerMessage*, the dispatcher deserializes the message and generates multiple *AddressedTuple*s according to dstIds. An *AddressedTuple* consists of the corresponding destination task Ids and the data item. The dispatcher sends *AddressedTuples* to the corresponding executors. Next, the working thread of an executor obtains the tuple from the executor incoming-queue and performs the processing logic.

Whale creates a single thread on the same node with the source instance to record the number of tuples arriving per unit time. With the variation of the input rate, to avoid the effects of noise, the measurement module needs to eliminate the noise, message loss, and outliers. The pre-processing includes statistics collecting and result smoothing. We adopt an α-weighted averaging to pre-process

Table 2: Statistics of the datasets

Dataset	# of tuples	# of keys
Didi Orders	13 B	6 M
Nasdaq Stock	274 M	6.7 K

the statistical data. Then, we have $\lambda(t) = \alpha * \lambda(t-1) + (1-\alpha) * N(t)$, where $N(t)$ denotes the number of tuples in time unit t. To avoid frequent communication with the controller, the monitor thread sends the stream input rate only under certain conditions (Section 3.3). The tuple processing time for transmission depends on hardware. The monitor thread records the tuple processing time of multiple tuples and computes the average value of t_e.

5 PERFORMANCE EVALUATION

5.1 Experiment Setups

We deploy Whale on a 30-node cluster, each machine is equipped with a 16-core 2.6 GHz Intel Xeon CPU, 64GB RAM, a Mellanox InfiniBand FDR 56 Gbps NIC, and a 1 Gbps Ethernet NIC. One server acts as *Nimbus*, and the others as *Supervisors*.

We examine the system performance of two real-world applications, on-demand ride-hailing and stock exchange. The on-demand ride-hailing application matches the passengers' requests to the drivers' locations. We use the Didi dataset [1] including 13 billion trajectory records associated with six million drivers and 74 million order records. The stock exchange application executes stock transaction matching and computes real-time trading volume. We have collected a one-month trace from NASDAQ, which contains 274 million exchange records related to 6,649 stock symbols. Each record consists of a stock symbol, the trading type (buy or sell), price, and a timestamp. Table 2 shows the statistics of the datasets.

In the experiment, we use Apache Kafka [2] as the source of the stream for both application topologies. In the on-demand ride-hailing application topology, we create two source operators to fetch stream data from Kafka. One partitions the drivers' location records with key grouping, and the other broadcasts the passengers' requests to all the matching instances. The matching operator receives drivers' location and stores them locally. At the same time, it receives passengers' order requests and joins them with drivers' locations to find the drivers within a certain distance from passengers. The matching operator outputs all the qualified matching results. An aggregation operator receives the matching results and returns the most suitable driver. In the stock exchange application topology, we use one source operator to read stock exchange records from Kafka. A split operator filters out the records without complying with trading rules and divides the stream into two streams, including a buying stream and a selling stream, based on the trading type. Then the split operator partitions the two streams to a matching operator, which joins the two streams and generates successful orders. An aggregation operator computes the real-time trading volume of successful orders.

We use Apache Storm [20], RDMA-based Storm [24], and RDMC [3] as baselines. We carefully examine and breakdown the gains of different techniques we proposed in Whale. For simplicity, we use the following notations. Whale-WOC denotes Whale with worker-oriented communication; Whale-WOC-RDMA stands for Whale with worker-oriented communication and the optimized RDMA primitives; Whale-WOC-RDMA-Nonblock denotes Whale with worker-oriented communication and RDMA-based non-blocking multicast.

In the experiments, we input the maximum stream rate following the Poisson process that the system can sustain.

We examine several important metrics of DSPSs [10]. The system throughput is defined as the number of tuples processed per unit time. The processing latency is the time for fully processing a tuple from the source instance to the sink. The multicast latency is the time from the producing of the tuple until all the destination instances receive it. The serialization time is the time spent to serialize a tuple. The communication time is the time that the source instance transmits a tuple [5]. The communication traffic is the number of bytes that the source instance transmits in network when it generates 10,000 tuples. Each machine of our 30-node cluster is equipped with a 16-core CPU and we create 120∼480 instances to evaluate the system performance. We run each experiment ten times and report the average. We examine the number of bytes in network transmission when the source instance generates 10,000 tuples.

5.2 Results

We first evaluate the system in on-demand ride-hailing application. Figure 13 shows the system throughput with different parallelism levels of the matching operator. The result shows that the throughput of Storm and RDMA-based Storm decline as the parallelism level increases. In contrast, the system throughput of Whale increases along with the parallelism level. When parallelism level = 480, Whale achieves 56.6× and 15× throughput improvements compared to Storm and RDMA-based Storm. The worker-oriented communication, the optimized RDMA primitives, and the non-blocking multicast contribute to 54%, 17%, and 29% of the improvement compared to RDMA-based Storm, respectively. The reason of the throughput decline of Storm and RDMA-based Storm is that a larger number of matching instances raises heavier transmission workloads of the upstream instance. This demonstrates the efficiency of our worker-oriented communication in preventing the increase of communication overhead as the parallelism level increases. Meanwhile, the non-blocking multicast strategy relieves the pressure on the upstream instance.

Figure 14 shows the processing latency with different parallelism levels of the matching operator in on-demand ride-hailing application. When the parallelism level increases, the latency of Storm and RDMA-based Storm increases, while that of Whale decreases. When parallelism level = 480, Whale achieves overall 96.6% and 95.9% reductions of processing latency compared to Storm and RDMA-based Storm, respectively. Besides, Whale-WOC and Whale-WOC-RDMA reduce the latency of RDMA-based Storm by 85.4% and 93.3%, respectively. The latency reduction increases as the parallelism level. The latency of Storm and RDMA-based Storm increases linearly with parallelism levels due to the increasing costs of sequentially tuple transmission. Benefiting from the worker-oriented communication, the latency of Whale decreases as the parallelism level increases. With more matching instances sharing the workloads, each instance performs less computation, leading to the decreasing of the processing latency.

Figures 15 and 16 show the system throughput and processing latency in stock exchange application. Figures 15 shows when the parallelism level = 480, Whale achieves an overall 51.2× and 16× improvement of system throughput compared to Storm and RDMA-based Storm, respectively. The worker-oriented communication,

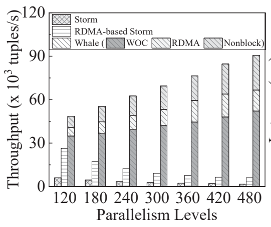


Figure 13: Overall system throughput on Didi dataset

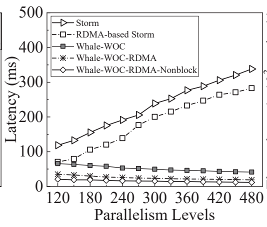


Figure 14: Overall system latency on Didi dataset

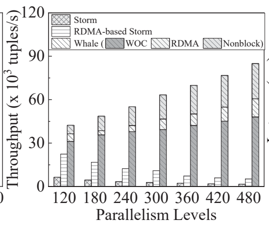


Figure 15: Overall system throughput on Nasdaq dataset

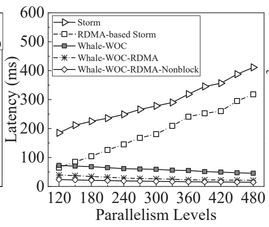


Figure 16: Overall system latency on Nasdaq dataset

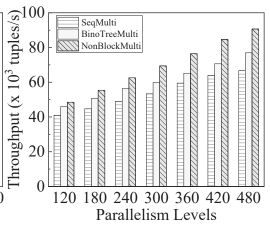


Figure 17: Throughput comparison using Didi dataset

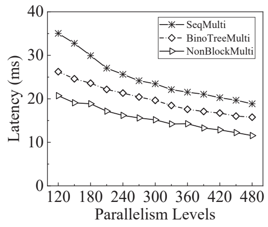


Figure 18: Latency comparison using Didi dataset

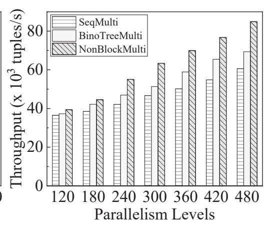


Figure 19: Throughput comparison using Nasdaq dataset

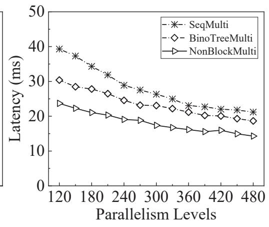


Figure 20: Latency comparison using Nasdaq dataset

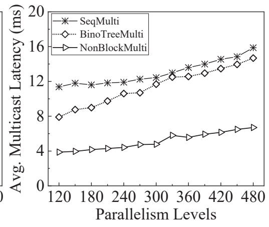


Figure 21: Average multicast latency using Didi dataset

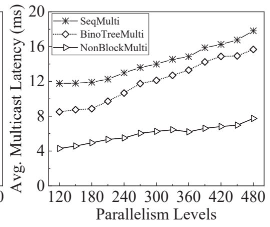


Figure 22: Average multicast latency using Nasdaq dataset

the optimized RDMA primitives, and the non-blocking multicast contribute to 53%, 16%, and 31% of the throughput improvement compared to RDMA-based Storm, respectively. The worker-oriented communication and the non-blocking multicast contribute most to the improvement, demonstrating our observation that the sequential one-to-many stream partitioning is the root cause of the system bottleneck. Figure 16 shows that when the parallelism level = 480, Whale overall reduces the processing latency of Storm and RDMA-based Storm by 96.5% and 95.5%, respectively. While Whale-WOC-RDMA and Whale-WOC reduce the latency of RDMA-based Storm by 93.3% and 85.7%, respectively. The non-blocking multicast strategy effectively reduces the transmission latency in the one-to-many partitioning.

We further compare the non-blocking multicast of Whale with the sequential multicast of Storm [20] and the binomial multicast of RDMC [3]. For a fair comparison, we implement all the three designs on top of Whale-WOC-RDMA. Figure 17 shows that when the parallelism level = 480, the non-blocking multicast tree of Whale achieves 1.2× and 1.4× improvements of the system throughput compared to RDMC’s binomial multicast tree and Storm’s sequential multicast, respectively. Figure 18 shows our non-blocking multicast reduces the latency of the binomial multicast and the sequential multicast by 26.9% and 38.8%, respectively. This is because Whale’s non-blocking multicast tree avoids the bottleneck in the sequential multicast and the previous binomial multicast tree.

Figures 19 and 20 show the system throughput and the processing latency in stock exchange application. The non-blocking multicast achieves 1.22× and 1.4× throughput improvements compared to the binomial multicast and the sequential multicast. Meanwhile, Whale reduces the latency of the binomial multicast and the sequential multicast by 23.4% and 32.6%, respectively. This shows Whale achieves better data parallelism.

We further evaluate the performance of the three stream multicast structures by examining the average multicast time. In the experiment, the maximum out-degree in non-blocking stream multicast tree is set to three. Figures 21 and 22 present the average multicast latency using the Didi and NASDAQ datasets. Figures 21 shows that the non-blocking multicast tree has a lower average multicast latency compared to the other two multicast models with the Didi dataset. When parallelism level = 480, the average multicast latency using Whale’s non-blocking multicast is 54.4% and 57.8% less than that using RDMC’s binomial multicast and Storm’s sequential multicast, respectively. Figure 22 shows that Whale achieves similar significant improvements of 50.6% and 56.6% for stock exchange. The non-blocking multicast tree achieves lower multicast latency because it limits the forwarding time for each tuple. Once a matching instance finishes forwarding a tuple, it forwards the next tuple.

We further evaluate Whale with highly dynamic data streams with the on-demand ride-hailing application. In the experiment, the input rate changes from 30k tuples/s to 60k tuples/s, 80k tuples/s, 100k tuples/s, and 80k tuples/s at the 40^{th}, 80^{th}, 120^{th}, 160^{th} second. Figures 23 and 24 show the system throughput and processing latency, where the numbers marked in the figures indicate the input rate in different time periods. Figure 23 shows that at the 40^{th} second, the system throughput of the non-blocking multicast structure drops within 126 milliseconds and catches up with the stream input rate quickly after switching. The non-blocking multicast structure improves throughput by 33% compared to the sequential multicast when the input rate is 100k tuples/s. The results show that the switching time is not affected by the delta of the input rate. Figure 24 shows the processing latency of the sequential multicast rises with the stream input rate, while that of the non-blocking multicast structure recovers within only 30ms. The switching delay of the non-blocking multicast structure is slight and acceptable with the dynamic switching strategy.

Figures 25 and 26 plot the communication time and the ratio of the serialization time in communication time in on-demand

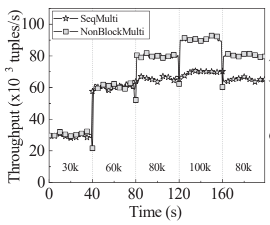


Figure 23: Throughput variation as stream rate changes

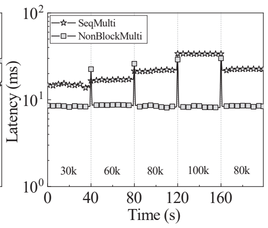


Figure 24: Latency variation as stream rate changes

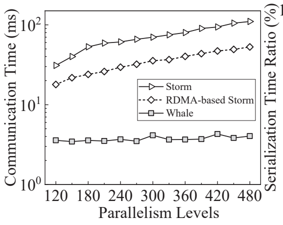


Figure 25: Communication time on different parallelism

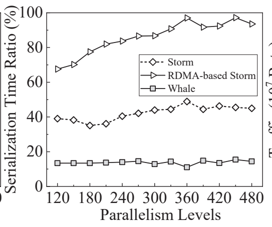


Figure 26: The ratio of serialization time

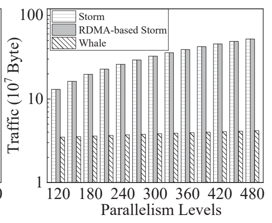


Figure 27: Communication traffic using Didi dataset

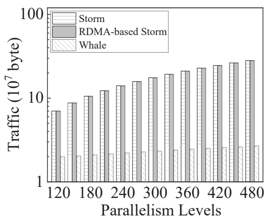


Figure 28: Communication traffic using Nasdaq dataset

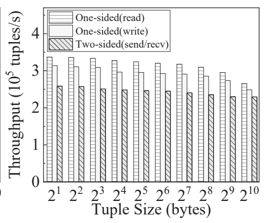


Figure 29: Throughput of different RDMA primitives

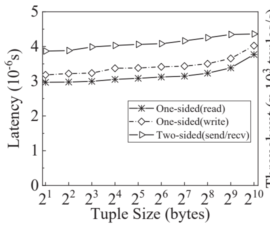


Figure 30: Average latency of different RDMA primitives

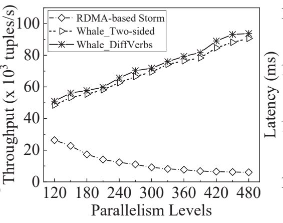


Figure 31: Throughput with different RDMA mechanisms

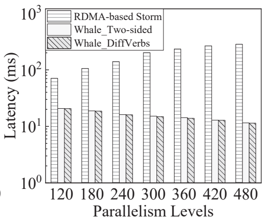


Figure 32: Latency with different RDMA mechanisms

ride-hailing application. Figure 25 shows the communication time of Whale changes slightly as the parallelism changes. When the parallelism = 480, Whale reduces the communication time of Storm and RDMA-based Storm by 96% and 92%, respectively. As Fig. 26 shows though RDMA-based Storm reduces the data transfer time, the serialization is still time-consuming. Whale takes only 15% communication time for serialization, while Storm and RDMA-based Storm need fractions of 45% and 94%, respectively. When the parallelism level = 480, the serialization time is 49.5 ms in Storm while it is only less than 1 ms in Whale.

Figures 27 and 28 show the communication traffic in on-demand ride-hailing and stock exchange. When the parallelism level reaches 480, Whale reduces the traffic by 91.9% for on-demand ride-hailing and by 90% for stock exchange. As the additional data in Whale is the destination IDs when the parallelism of destination instances increases, the traffic of Whale changes slightly, while those of Storm and RDMA-based Storm increase rapidly. RDMA-based Storm does not change the instance-oriented communication pattern in Storm, they have the same traffic.

Figures 29 and 30 compare the throughput and average latency of one-sided and two-sided RDMA operations. The results show

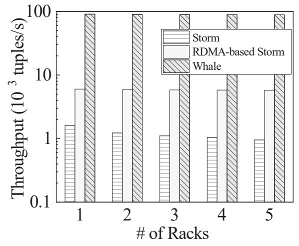


Figure 33: Throughput with different number of racks

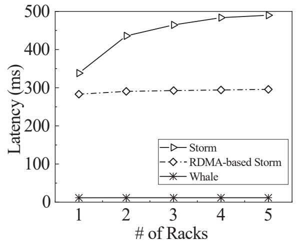


Figure 34: Latency with different number of racks

one-sided operations perform better than two-sided operations. In one-sided operations, RDMA read has higher throughput and lower average latency than RDMA write. We also evaluate the system throughput and processing latency of our proposed RDMA optimization strategy in Figs. 31 and 32 in the on-demand ride-hailing application. The results show with suitable different RDMA operations (Whale_DiffVerbs), Whale achieves 15.6x improvement of throughput and 96% reduction of processing latency compared to RDMA-based Storm.

We also evaluate the performance of Whale with different physical cluster topology, by partitioning the machines into one to fives racks. We compare system throughput and processing latency of Whale with Storm and RDMA-based Storm in the on-demand ride-hailing application. Figure 33 shows the throughput of Whale keeps stable when the number of racks varies from one to five. Figure 34 shows the latency of Whale changes very slightly.

6 CONCLUSIONS

In this paper, we proposes Whale, a novel RDMA-assisted distributed stream processing system with efficient one-to-many data partitioning. Whale designs a novel non-blocking multicast tree structure for RDMA-assisted one-to-many data partitioning and a new worker-oriented communication mechanism in distributed stream processing systems. Comprehensive experimental results show that Whale greatly outperforms existing designs. It achieves 56.6× improvement of system throughput and 97% reduction of processing latency compared to existing designs.

7 ACKNOWLEDGEMENTS

This research is supported in part by the National Key Research and Development Program of China under grant No.2018YFB1004602, NSFC under grants Nos. 61972446, 61422202.

REFERENCES

[1] Gaia Initiative. https://outreach.didichuxing.com/research/opendata/en, 2020.

[2] Kafka. http://kafka.apache.org, 2020.

[3] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. RDMC: A reliable RDMA multicast for large objects. In *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Luxembourg City, Luxembourg, June 25-28, 2018.

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28-38, 2015.

[5] Hanhua Chen, Hai Jin, and Shaoliang Wu. Minimizing inter-server communications by exploiting self-similarity in online social networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1116–1130, 2016.

[6] Hanhua Chen, Fan Zhang, and Hai Jin. Popularity-aware differentiated distributed stream processing on skewed streams. In *Proceedings of the 25th IEEE International Conference on Network Protocols (ICNP)*, Toronto, ON, Canada, October 10-13, 2017.

[7] Bugra Gedik. Partitioning functions for stateful data parallelism in stream processing. *Proceedings of the VLDB Endowment*, vol. 23, no. 4, pp. 517-539, 2014.

[8] Nusrat Sharmin Islam, Mohammad Wahidur Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K Panda. High performance RDMA-based design of HDFS over infiniband. In *Proceedings of the 2012 International Conference on High Performance Computing Networking, Storage and Analysis (SC)*, Salt Lake City, UT, USA, November 11-15, 2012.

[9] Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching, Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz, Anirban Banerjee, Benjamin Heintz, Shridar Iyer, and Anshul Jaiswal. Providing streaming joins as a service at facebook. *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1809-1821, 2018.

[10] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, Paris, France, April 16-19, 2018.

[11] David G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, vol. 24, no. 3, pp. 338-354, 1953.

[12] Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 4-14, 2015.

[13] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Melbourne, Victoria, Australia, May 31 - June 4, 2015.

[14] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanam Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1303-1316, 2018.

[15] Michael G. Moore and Mark A. Davenport. Estimation of poisson arrival processes under linear models. *IEEE Transactions on Information Theory*, vol. 65, no. 6, pp. 3555-3564, 2019.

[16] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE)*, Seoul, South Korea, April 13-17, 2015.

[17] Gengbiao Shen, Qing Li, Shuo Ai, Yong Jiang, Mingwei Xu, and Xuya Jia. How powerful switches should be deployed: A precise estimation based on queuing theory. In *Proceedings of 2019 IEEE Conference on Computer Communications (INFOCOM)*, Paris, France, April 29 - May 2, 2019.

[18] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: when RPC is faster than server-bypass with RDMA. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, 2017.

[19] Yongxin Tong, Yuqiang Chen, Zimu Zhou, Lei Chen, Jie Wang, Qiang Yang, Jieping Ye, and Weifeng Lv. The simpler the better: a unified approach to predicting original taxi demands based on large-scale online platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, Halifax, NS, Canada, August 13-17, 2017.

[20] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Snowbird, UT, USA, June 22-27, 2014.

[21] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, vol. 21, no. 4, pp. 309-315, 1978.

[22] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast inmemory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, USA, October 4-7, 2015.

[23] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Filemr: Rethinking RDMA networking for scalable persistent memory. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, USA, February 25-27, 2020.

[24] Seokwoo Yang, Siwoon Son, Mi-Jung Choi, and Yang-Sae Moon. Performance improvement of Apache Storm using infiniband RDMA. *The Journal of Supercomputing*, vol. 75, no. 10, pp. 6804-6830, 2019.

[25] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. Towards zero copy dataflows using RDMA. In *Proceedings of the 2017 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, pages 28–30, Los Angeles, CA, USA, August 21-25, 2017.

[26] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 516-530, 2019.

[27] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, London, United Kingdom, August 17-21, 2015.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We evaluate the performance of Whale on a 30-node cluster, each machine in the cluster is equipped with a 16-core 2.6GHz Intel Xeon E5-2670 CPU, 64GB RAM, 256GB hard disk, a Mellanox InfiniBand FDR 56Gbps NIC, and a 1Gbps Ethernet NIC. We deploy the Whale system on machines which run Red Hat Enterprise Linux Server release 6.2, JDK 1.8.0, Scala 2.11.12, Maven 3.5.4. What's more, we deploy Zookeeper 3.4.6 to coordinate the cluster and Kafka 0.10.1 to serve as the data source. The testing code we used for evaluation can be publicly accessed at https://github.com/Whale2021/Whale/tree/master/benchmark/.

Author-Created or Modified Artifacts:

```
Persistent ID: DOI: 10.5281/zenodo.4897500, Github
↪  URL: https://github.com/Whale2021/Whale
Artifact name: Whale
Citation of artifact: RDMAChannel

Persistent ID: DOI: 10.5281/zenodo.4675656, Github
↪  URL: https://github.com/Whale2021/RDMAChannel
Artifact name: RDMAChannel
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: a 16-core 2.6GHz Intel Xeon E5-2670 CPU, 64GB RAM, 256GB hard disk, a Mellanox InfiniBand FDR 56Gbps NIC, and a 1Gbps Ethernet NIC

Operating systems and versions: Red Hat Enterprise Linux 6.2 running Linux kernel 2.6.32

Compilers and versions: Maven v3.5.4

Applications and versions: Testing code at Whale

Libraries and versions: Storm v2.0.0-SNAPSHOT, disni v1.6

Input datasets and versions: Dataset, https://github.com/Whale2021/Dataset. The traces used for evaluation are obtained from Gaia Initiative of Didi Chuxing: https://outreach.didichuxing.com/research/opendata/en and NASDAQ stock exchange records: https://www.nasdaq.com.

URL to output from scripts that gathers execution environment information.

```
https://github.com/Whale2021/Whale/blob/master/sc21\⌋
↪  _execution\_env.txt
```