CHAPTER 11

Key Technologies for Big Data Stream Computing

Dawei Sun, Guangyan Zhang, Weimin Zheng, and Keqin Li

CONTENTS

ABSTRACT

As a new trend for data-intensive computing, real-time stream computing is gaining significant attention in the Big Data era. In theory, stream computing is an effective way to support Big Data by providing extremely low-latency processing tools and massively parallel processing architectures in real-time data analysis. However, in most existing stream computing environments, how to efficiently deal with Big Data stream computing and how to build efficient Big Data stream computing systems are posing great challenges to Big Data computing research. First, the data stream graphs and the system architecture for Big Data stream computing, and some related key technologies, such as system structure, data transmission, application interfaces, and high availability, are systemically researched. Then, we give a classification of the latest research and depict the development status of some popular Big Data stream computing systems, including Twitter Storm, Yahoo! S4, Microsoft TimeStream, and Microsoft Naiad. Finally, the potential challenges and future directions of Big Data stream computing are discussed.

11.1 INTRODUCTION

Big Data computing is a new trend for future computing, with the quantity of data growing and the speed of data increasing. In general, there are two main mechanisms for Big Data computing, that is, Big Data stream computing (BDSC) and Big Data batch computing. BDSC is a model of straight-through computing, such as Storm [1] and S4 [2], which does for stream computing what Hadoop does for batch computing, while Big Data batch computing is a model of storing then computing, such as the MapReduce framework [3] open-sourced by the Hadoop implementation [4].

Essentially, Big Data batch computing is not sufficient for many real-time application scenarios, where a data stream changes frequently over time and the latest data are the most important and most valuable. For example, when analyzing data from real-time transactions (e.g., financial trades, e-mail messages, user search requests, sensor data tracking), a data stream grows monotonically over time as more transactions take place. Ideally, a real-time application environment can be supported by BDSC. Generally, Big Data streaming computing has the following defining characteristics [5,6]. (1) The input data stream is a real-time data stream and needs real-time computing, and the results must be updated

every time the data changes. (2) Incoming data arrive continuously at volumes that far exceed the capabilities of individual machines. (3) Input streams incur multistaged computing at low latency to produce output streams, where any incoming data entry is ideally reflected in the newly generated results in output streams within seconds.

11.1.1 Stream Computing

Stream computing, the long-held dream of "high real-time computing" and "high-throughput computing," with programs that compute continuous data streams, has opened up the new era of future computing due to Big Data, which is a data set that is large, fast, dispersed, unstructured, and beyond the ability of available hardware and software facilities to undertake its acquisition, access, analytics, and application in a reasonable amount of time and space [7,8]. Stream computing is a computing paradigm that reads data from collections of software or hardware sensors in stream form and computes continuous data streams, where feedback results should be in a real-time data stream as well. A data stream is a sequence of data sets, a continuous stream is an infinite sequence of data sets, and parallel streams have more than one stream to be processed at the same time.

Stream computing is one effective way to support Big Data by providing extremely low-latency velocities with massively parallel processing architectures and is becoming the fastest and most efficient way to obtain useful knowledge from Big Data, allowing organizations to react quickly when problems appear or to predict new trends in the near future [9,10].

A Big Data input stream has the characteristics of high speed, real time, and large volume for applications such as sensor networks, network monitoring, microblogging, web exploring, social networking, and so on. These data sources often take the form of continuous data streams, and timely analysis of such a data stream is very important as the life cycle of most data is very short [8,11,12]. Furthermore, the volume of data is so high that there is not enough space for storage, and not all data need to be stored. Thus, the storing-then-computing batch computing model does not fit at all. Nearly all data in Big Data environments have the feature of streams, and stream computing has appeared to solve the dilemma of Big Data computing by computing data online within real-time constraints [13]. Consequently, the stream computing model will be a new trend for high-throughput computing in the Big Data era.

11.1.2 Application Background

BDSC is able to analyze and process data in real time to gain immediate insight, and it is typically applied to the analysis of a vast amount of data in real time and to processing them at a high speed. Many application scenarios require BDSC. For example, in financial industries, Big Data stream computing technologies can be used in risk management, marketing management, business intelligence, and so on. In the Internet, BDSC technologies can be used in search engines, social networking, and so on. In the Internet of things, BDSC technologies can be used in intelligent transportation, environmental monitoring, and so on.

Usually, a BDSC environment is deployed in a highly distributed clustered environment, as the amount of data is infinite, the rate of the data stream is high, and the results should be real-time feedback.

11.1.3 Chapter Organization

The remainder of this chapter is organized as follows. In Section 11.2, we introduce data stream graphs and the system architecture for BDSC and key technologies for BDSC systems. In Section 11.3, we present the system architecture and key technologies of four popular example BDSC systems, which are Twitter Storm, Yahoo! S4, Microsoft TimeStream, and Microsoft Naiad. Finally, we discuss grand challenges and future directions in Section 11.4.

11.2 OVERVIEW OF A BDSC SYSTEM

In this section, we first present some related concepts and definitions of directed acyclic graphs and stream computing. Then, we introduce the system architecture for stream computing and the key technologies for BDSC systems in BDSC environments.

11.2.1 Directed Acyclic Graph and Stream Computing

In stream computing, the multiple continuous parallel data streams can be represented by a task topology, also named a data stream graph, which is usually described by a directed acyclic graph (DAG) [5,14–16]. A measurable data stream graph view can be defined by Definition 1.

Definition 1

A *data stream graph* G is a directed acyclic graph, which is composed of set of a vertices and a set of directed edges; has a logical structure and a special function; and is denoted as $G = (V(G), E(G))$, where $V(G) = \{v_1, v_2, \ldots, v_n\}$ is a finite set of n vertices, which represent tasks, and $E(G) = \{e_{1,2}, e_{1,3}, \ldots, e_{n-1,n}\}$ is a finite set of directed edges, which represent a data stream between vertices. If $\exists e_{i,j} \in E(G)$, then $v_i, v_j \in V(G)$, $v_i \neq v_j$, and $\langle v_i, v_j \rangle$ is an ordered pair, where a data stream comes from v_i and goes to v_j.

The in-degree of vertex v_i is the number of incoming edges, and the out-degree of vertex v_i is the number of outgoing edges. A source vertex is a vertex whose in-degree is 0, and an end vertex is a vertex whose out-degree is 0. A data stream graph G has at least one source vertex and one end vertex.

For the example data stream graph with 11 vertices shown in Figure 11.1, the set of vertices is $V = \{v_a, v_b, \ldots, v_k\}$, the set of directed edges is $E = \{e_{a,c}, e_{b,c}, \ldots, e_{j,k}\}$, the source vertices

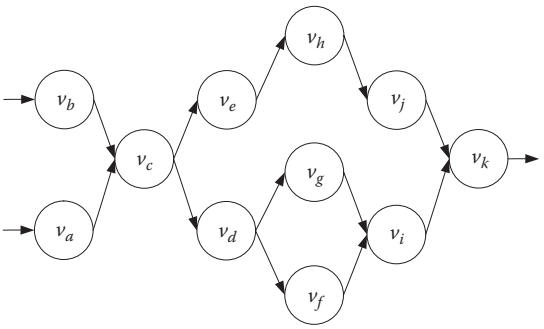

FIGURE 11.1 A data stream graph.

are v_a and v_b, and the end vertex is v_k. The in-degree of vertex v_d is 1, and the out-degree of vertex v_d is 2.

Definition 2

A *subgraph sub-G* of the data stream graph *G* is a subgraph consisting of a subset of the vertices with the edges in between. For vertices v_i and v_j in the subgraph *sub-G* and any vertex *v* in the data stream graph *G*, *v* must also be in the *sub-G* if *v* is on a directed path from v_i to v_j, that is, $\forall v_i, v_j \in V(sub\text{-}G)$, $\forall v \in V(G)$, and if $v \in V\ (p(v_i, v_j))$, then $v \in V\ (p(sub\text{-}G))$.

A subgraph *sub-G* is logically equivalent and can be substituted by a vertex. But reducing that subgraph to a single logical vertex would create a graph with cycle, not a DAG.

Definition 3

A *path* $p(v_u, v_v)$ from vertex v_u to vertex v_v is a subset of $E(p(v_u, v_v))$, which should meet the conditions $\exists e_{i,k} \in p(v_u, v_v)$ and $e_{k,j} \in p(v_u, v_v)$ for any directed edge $e_{k,l}$ in path $p(v_u, v_v)$ that displays the following properties: If $k \neq i$, then $\exists m$, and $e_{m,k} \in p(v_u, v_v)$; if $i \neq j$, then $\exists m$, and $e_{l,m} \in p(v_u, v_v)$.

The latency $l_p(v_u, v_v)$ of a path from vertex v_u to vertex v_v is the sum of latencies of both vertices and edges on the path, as given by Equation 11.1:

$$l_p(v_u, v_v) = \sum_{v_i \in V(p(v_u, v_v))} c_{v_i} + \sum_{e_{i,j} \in E(p(v_u, v_v))} c_{e_{i,j}}, c_{v_i}, c_{e_{i,j}} \geq 0. \tag{11.1}$$

A critical path, also called the longest path, is a path with the longest latency from a source vertex v_s to an end vertex v_e in a data stream graph *G*, which is also the latency of data stream graph *G*.

If there are *m* paths from source vertex v_s to end vertex v_e in data stream graph *G*, then the latency $l(G)$ of data stream graph *G* is given by Equation 11.2:

$$l(G) = \max\left\{l_{p_i}(v_s, v_e), l_{p_2}(v_s, v_e), \ldots, l_{p_m}(v_s, v_e)\right\}, \tag{11.2}$$

where $l_{p_i}(v_s, v_e)$ is the latency of the *i*th path from vertex v_s to vertex v_e.

Definition 4

In data stream graph *G*, if $\exists e_{i,j}$ from vertex v_i to vertex v_j, then vertex v_i is a *parent* of vertex v_j, and vertex v_j is a *child* of vertex v_i.

Definition 5

The *throughput* $t(v_i)$ of vertex v_i is the average rate of successful data stream computing in a Big Data environment and is usually measured in bits per second (bps).

We identify the source vertex v_s as in the first level, the children of source vertex v_s as in the second level, and so on, and the end vertex v_e as in the last level.

The throughput $t(level_i)$ of the ith level can be calculated by Equation 11.3:

$$t(level_i) = \sum_{k=1}^{n_i} t(v_k), \tag{11.3}$$

where n_i is the number of vertices in the ith level.

If data stream graph G has m levels, then the throughput $t(G)$ of the data stream graph G is the minimum throughput of all the throughput in the m levels, as described by Equation 11.4:

$$t(G) = \min\{t(level_1), t(level_2),\ldots,t(level_m)\}, \tag{11.4}$$

where $t(level_i)$ is the throughput of the ith level in data stream G.

Definition 6

A *topological sort* $TS(G) = \left(v_{x_1}, v_{x_2},\ldots,v_{x_n}\right)$ of the vertices $V(G)$ in data stream graph G is a linear ordering of its vertices, such that for every directed edge $e_{x_i,x_j}\left(e_{x_i,x_j} \in E(G)\right)$ from vertex v_{x_i} to vertex v_{x_j}, v_{x_i} comes before v_{x_j} in the topological ordering.

A topological sort is possible if and only if the graph has no directed cycle, that is, it needs to be a directed acyclic graph. Any directed acyclic graph has at least one topological sort.

Definition 7

A *graph partitioning* $GP(G) = \{GP_1, GP_2,\ldots,GP_m\}$ of the data stream graph G is a topological sort–based split of the vertex set $V(G)$ and the corresponding directed edges. A graph partitioning should meet the nonoverlapping and covering properties, that is, if $\forall i \neq j$, $i, j \in [1, m]$, then $GP_i \bigcap GP_j = \varnothing$, and $\bigcup_{i=1}^{m} GP_i = V(G)$.

11.2.2 System Architecture for Stream Computing

In BDSC environments, stream computing is the model of straight-through computing. As shown in Figure 11.2, the input data stream is in a real-time data stream form, all

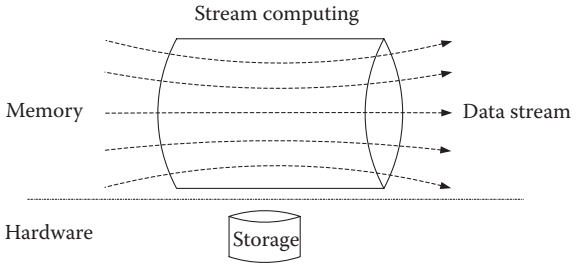

FIGURE 11.2 A Big Data stream computing environment.

continuous data streams are computed in real time, and the results must be updated also in real time. The volume of data is so high that there is not enough space for storage, and not all data need to be stored. Most data will be discarded, and only a small portion of the data will be permanently stored in hard disks.

11.2.3 Key Technologies for BDSC Systems

Due to data streams' distinct features of real time, volatility, burstiness, irregularity, and infinity in a Big Data environment, a well-designed BDSC system always optimizes in system structure, data transmission, application interfaces, high availability, and so on [17–19].

11.2.3.1 System Structure

Symmetric structure and master–slave structure are two main system structures for BDSC systems, as shown in Figures 11.3 and 11.4, respectively.

In the symmetric structure system, as shown in Figure 11.3, the functions of all nodes are the same. So it is easy to add a new node or to remove an unused node, and to improve the scalability of a system. However, some global functions such as resource allocation, fault tolerance, and load balancing are hard to achieve without a global node. In the S4 system, the global functions are achieved by borrowing a distributed protocol zookeeper.

In the master–slave structure system, as shown in Figure 11.4, one node is the master node, and other nodes are slave nodes. The master node is responsible for global control of the system, such as resource allocation, fault tolerance, and load balancing. Each slave node

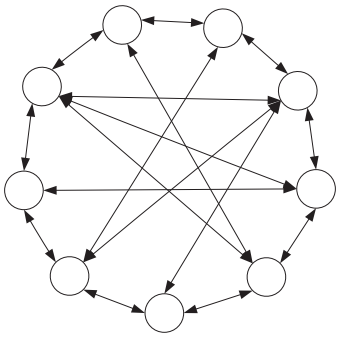

FIGURE 11.3 Symmetric structure.

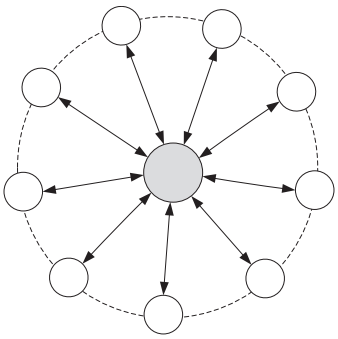

FIGURE 11.4 Master–slave structure.

has a special function, and it always receives a data stream from the master node, processes the data stream, and sends the results to the master node. Usually, the master node is the bottleneck in the master–slave structure system. If it fails, the whole system will not work.

11.2.3.2 Data Stream Transmission

Push and pull are two main data stream transmissions in a BDSC system.

In a push system, once an upstream node gets a result, it will immediately push the result data to downstream nodes. When this is done, the upstream data will be immediately sent to downstream nodes. However, if some downstream nodes are busy or fail, some data will be discarded.

In a pull system, a downstream node requests data from an upstream node. If some data need to be further processed, the upstream node will send the data to the requesting downstream node. When this is done, the upstream data will be stored in upstream nodes until corresponding downstream nodes make a request. Some data will wait a long time for further processing and may lose their timeliness.

11.2.3.3 Application Interfaces

An application interface is used to design a data stream graph, a bridge between a user and a BDSC system. Usually, a good application interface is flexible and efficient for users. Currently, most BDSC systems provide MapReduce-like interfaces; for example, the Storm system provides spouts and bolts as application interfaces, and a user can design a data stream graph using spouts and bolts. Some other BDSC systems provide structured query language (SQL)-like interfaces and graphical user interfaces.

11.2.3.4 High Availability

State backup and recovery is the main method to achieve high availability in a BDSC system. There are three main high-availability strategies, that is, passive standby strategy, active standby strategy, and upstream backup strategy.

In the passive standby strategy (see Figure 11.5), each primary node periodically sends checkpoint data to a backup node. If the primary node fails, the backup node takes over from the last checkpoint. Usually, this strategy will achieve precise recovery.

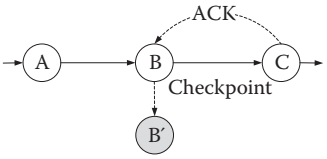

FIGURE 11.5 Passive standby. ACK, Acknowledgment.

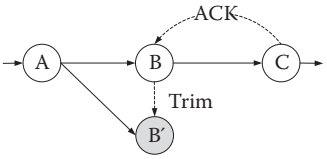

FIGURE 11.6 Active standby. ACK, Acknowledgment.

In the active standby strategy (see Figure 11.6), the secondary nodes compute all data streams in parallel with their primaries. Usually, the recovery time of this strategy is the shortest.

In the upstream backup strategy (see Figure 11.7), upstream nodes act as backups for their downstream neighbors by preserving data streams in their output queues while their downstream neighbors compute them. If a node fails, its upstream nodes replay the logged data stream on a recovery node. Usually, the runtime overhead of this strategy is the lowest.

A comparison of the three main high-availability strategies, that is, passive standby strategy, active standby strategy, and upstream backup strategy, in runtime overhead and recovery time is shown in Figure 11.8. The recovery time of the upstream backup strategy is the longest, while the runtime overhead of the passive standby strategy is the greatest.

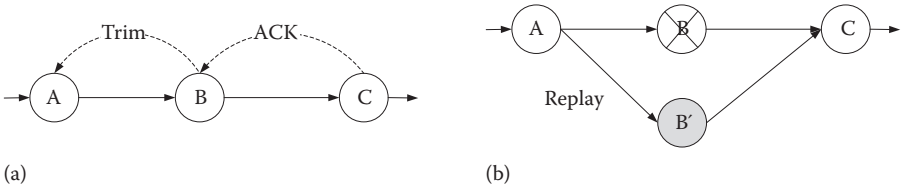

FIGURE 11.7 Upstream backup. ACK, Acknowledgment.

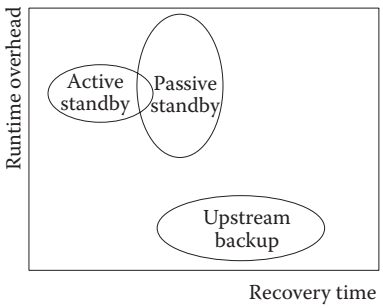

FIGURE 11.8 Comparison of high-availability strategies in runtime overhead and recovery time.

11.3 EXAMPLE BDSC SYSTEMS

In this section, the system architecture and key technologies of four popular BDSC system instances are presented. These systems are Twitter Storm, Yahoo! S4, Microsoft TimeStream, and Microsoft Naiad, which are specially designed for BDSC.

11.3.1 Twitter Storm

Storm is an open-source and distributed BDSC system licensed under the Eclipse Public License. Similar to how Hadoop provides a set of general primitives for doing batch processing, Storm provides a set of general primitives for doing real-time Big Data computing. The Storm platform has the features of simplicity, scalability, fault tolerance, and so on. It can be used with any programming language and is easy to set up and operate [1,20,21].

11.3.1.1 Task Topology

In BDSC environments, the logic for an application is packaged in the form of a task topology. Once a task topology is designed and submitted to a system, it will run forever until the user kills it.

A task topology can be described as a directed acyclic graph and comprises spouts and bolts, as shown in Figure 11.9. A spout is a source of streams in a task topology and will read data streams (in tuples) from an external source and emit them into bolts. Spouts can emit more than one data stream. The processing of a data stream in a task topology is done in bolts. Anything can be done by bolts, such as filtering, aggregations, joins, and so on. Some simple functions can be achieved by a bolt, while complex functions will be achieved by many bolts. The logic should be designed by a user. For example, transforming a stream of tweets into a stream of trending images requires at least two steps: a bolt to do a rolling count of retweets for each image and one or more bolts to stream out the top n images. Bolts can also emit more than one stream. Each edge in the directed acyclic graph represents a bolt subscribing to the output stream of some other spout or bolt.

A data stream is an unbounded sequence of tuples that is processed and created in parallel in a distributed BDSC environment. A task topology processes data streams in many complex ways. Repartitioning the streams between each stage of the computation is needed. Task topologies are inherently parallel and run across a cluster of machines. Any

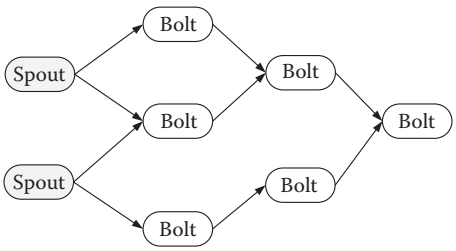

FIGURE 11.9 Task topology of Storm.

vertex in a task topology can be created in many instances. All those vertices will simultaneously process a data stream, and different parts of the topology can be allocated in different machines. A good allocating strategy will greatly improve system performance.

A data stream grouping defines how that stream should be partitioned among the bolt's tasks; spouts and bolts execute in parallel as many tasks across the cluster. There are seven built-in stream groupings in Storm, such as shuffle grouping, fields grouping, all grouping, global grouping, none grouping, direct grouping, and local or shuffle grouping; a custom stream grouping to meet special needs can also be implemented by the CustomStreamGrouping interface.

11.3.1.2 Fault Tolerance

Fault tolerance is an important feature of Storm. If a worker dies, Storm will automatically restart it. If a node dies, the worker will be restarted on another node. In Storm, Nimbus and the Supervisors are designed to be stateless and fail-fast whenever any unexpected situation is encountered, and all state information is stored in a Zookeeper server. If Nimbus or the Supervisors die, they will restart like nothing happened. This means you can kill the Nimbus and the Supervisors without affecting the health of the cluster or task topologies.

When a worker dies, the Supervisor will restart it. If it continuously fails on startup and is unable to heartbeat to Nimbus, Nimbus will reassign the worker to another machine.

When a machine dies, the tasks assigned to that machine will time out, and Nimbus will reassign those tasks to other machines.

When Nimbus or Supervisors die, they will restart like nothing happened. No worker processes are affected by the death of Nimbus or the Supervisors.

11.3.1.3 Reliability

In Storm, the reliability mechanisms guarantee that every spout tuple will be fully processed by corresponding topology. They do this by tracking the tree of tuples triggered by every spout tuple and determining when that tree of tuples has been successfully completed. Every topology has a "message timeout" associated with it. If Storm fails to detect that a spout tuple has been completed within that timeout, then it fails the tuple and replays it later.

The reliability mechanisms of Storm are completely distributed, scalable, and fault tolerant. Storm uses mod hashing to map a spout tuple ID to an acker task. Since every tuple carries with it the spout tuple IDs of all the trees they exist within, they know which acker tasks to communicate with. When a spout task emits a new tuple, it simply sends a message to the appropriate acker telling it that its task ID is responsible for that spout tuple. Then, when an acker sees that a tree has been completed, it knows to which task ID to send the completion message.

An acker task stores a map from a spout tuple ID to a pair of values. The first value is the task ID that created the spout tuple that is used later on to send completion messages. The second value is a 64-bit number called the "ack val." The ack val is a representation of the state of the entire tuple tree, no matter how big or how small. It is simply the exclusive

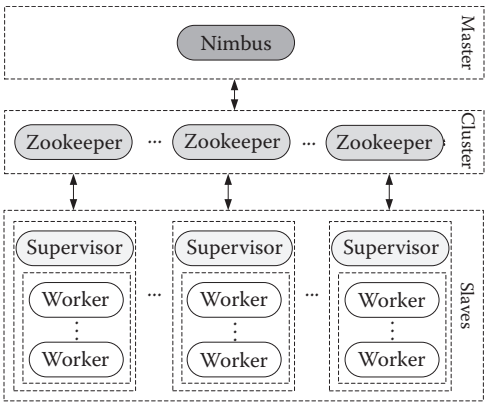

FIGURE 11.10 Storm cluster.

OR (XOR) of all tuple IDs that have been created and/or acked in the tree. When an acker task sees that an "ack val" has become 0, then it knows that the tuple tree is completed.

11.3.1.4 Storm Cluster

A Storm cluster is superficially similar to a Hadoop cluster. Whereas on Hadoop, you run "MapReduce jobs," on Storm, you run "topologies." As shown in Figure 11.10, there are two kinds of nodes on a Storm cluster, that is, the master node and the worker nodes.

The master node runs Nimbus node, which is similar to Hadoop's "JobTracker." In Storm, Nimbus node is responsible for distributing code around the cluster, assigning tasks to machines, monitoring for failures, and so on.

Each worker node runs a Supervisor node. The Supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it. Each worker process executes a subset of a topology. Usually, a running topology consists of many worker processes spread across many machines.

The coordination between Nimbus and the Supervisors is done through a Zookeeper cluster. Additionally, the Nimbus daemon and Supervisor daemons are fail-fast and stateless; all states are kept in a Zookeeper server. This means that if you kill the Nimbus or the Supervisors, they will start back up like nothing has happened.

11.3.2 Yahoo! S4

S4 is a general-purpose, distributed, scalable, fault-tolerant, pluggable platform that allows programmers to easily develop applications for computing continuous unbounded streams of Big Data. The core part of S4 is written in Java. The implementation is modular and pluggable, and S4 applications can be easily and dynamically combined for creating more sophisticated stream processing systems. S4 was initially released by Yahoo! Inc. in October 2010 and has been an Apache Incubator project since September 2011. It is licensed under the Apache 2.0 license [2,22–25].

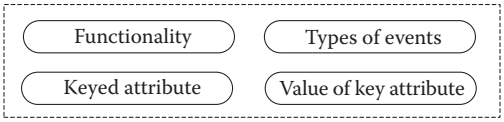

FIGURE 11.11 Processing element.

11.3.2.1 Processing Element

The computing units of S4 are the *processing elements* (PEs). As shown in Figure 11.11, each instance of a PE can be identified by four components, that is, functionality, types of events, keyed attribute, and value of the keyed attribute. Each PE processes exactly those events that correspond to the value on which it is keyed.

A special class of PEs is the set of keyless PEs, with no keyed attribute or value. This type of PE will process all events of the type with which they are associated. Usually, the keyless PEs are typically used at the input layer of an S4 cluster, where events are assigned a key.

11.3.2.2 Processing Nodes

Processing nodes (PNs) are the logical hosts to PEs. Many PEs work in a PE container, as shown in Figure 11.12. A PN is responsible for event listeners, dispatcher events, and emitter output events. In addition, the routing model, load balancing model, fail-over management model, transport protocols, and zookeeper are deployed in a communication layer.

All events will be routed to PNs by S4 according to a hash function. Every keyed PE can be mapped to exactly one PN based on the value of the hash function applied to the value of the keyed attribute of that PE. However, keyless PEs may be instantiated on every PN. The event listener model of a PN will always listen to an event from S4. If an event is allocated to a PN, it will be routed to an appropriate PE within that PN.

11.3.2.3 Fail-Over, Checkpointing, and Recovery Mechanism

In S4, a fail-over mechanism will provide a high-availability environment for S4. When a node is dead, a corresponding standby node will be used. In order to minimize state loss when a node is dead, a checkpointing and recovery mechanism is employed by S4.

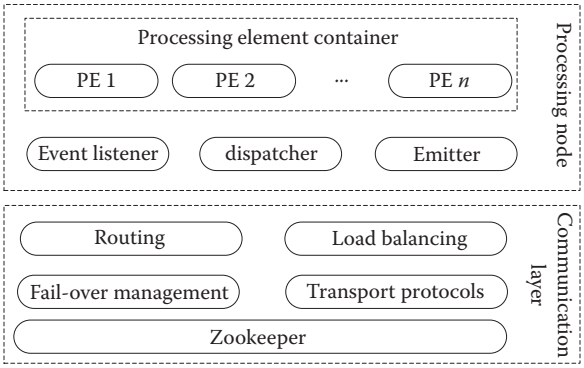

FIGURE 11.12 Processing node.

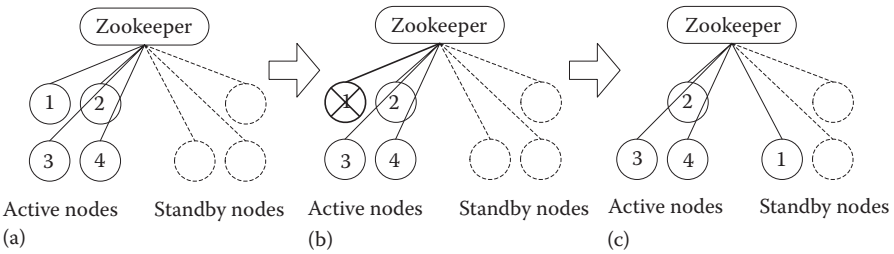

FIGURE 11.13 Fail-over mechanism. (a) In working state, (b) under failed state, and (c) after recovery state.

In order to improve the availability of the S4 system, S4 system should provide a fail-over mechanism to automatically detect failed nodes and redirect the data stream to a standby node. If you have n partitions and start m nodes, with $m > n$, you get $m - n$ standby nodes. For instance, if there are seven live nodes and four partitions available, four of the nodes pick the available partitions in Zookeeper. The remaining three nodes will be available standby nodes. Each active node consistently receives messages for the partition that it picked, as shown in Figure 11.13a. When Zookeeper detects that one of active nodes fails, it will notify a standby node to replace the failed node. As shown in Figure 11.13b, the node assigned with partition 1 fails. Unassigned nodes compete for a partition assignment, and only one of them picks it. Other nodes are notified of the new assignment and can reroute the data stream for partition 1, as shown in Figure 11.13c.

If a node is unreachable after a session timeout, Zookeeper will identify this node as dead. The session timeout is specified by the client upon connection and is, at minimum, twice the heartbeat specified in the Zookeeper ensemble configuration.

In order to minimize state loss when a node is dead, a checkpointing and recovery mechanism is employed by S4. The states of PEs are periodically checkpointed and stored. Whenever a node fails, the checkpoint information will be used by the recovery mechanism to recover the state of the failed node to the corresponding standby node. Most of the previous state of a failed node can be seen in the corresponding standby node.

11.3.2.4 System Architecture

In S4, a decentralized and symmetric architecture is used; all nodes share the same functionality and responsibilities (see Figure 11.14). There is no central node with specialized responsibilities. This greatly simplifies deployment and maintenance.

A pluggable architecture is used to keep the design as generic and customizable as possible.

11.3.3 Microsoft TimeStream and Naiad

TimeStream and Naiad are two BDSC systems of Microsoft.

11.3.3.1 TimeStream

TimeStream is a distributed system designed specifically for low-latency continuous processing of big streaming data on a large cluster of commodity machines and is based on

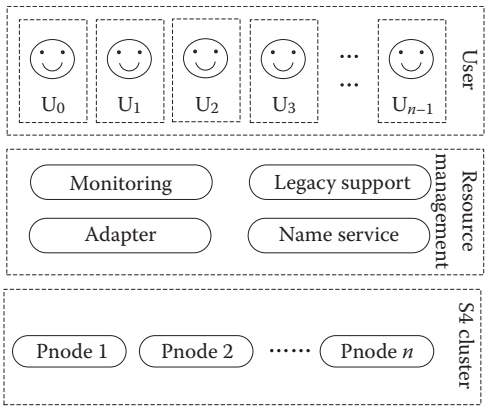

FIGURE 11.14 System architecture.

StreamInsight. TimeStream handles an online advertising aggregation pipeline at a rate of 700,000 URLs per second with a 2 s delay [5,26–29].

1. Streaming DAG

   Streaming DAG is a type of task topology, which can be dynamically reconfigured according to the loading of a data stream. All data streams in the TimeStream system will be processed in streaming DAG. Each vertex in streaming DAG will be allocated to physical machines for execution. As shown in Figure 11.15, streaming function f_v of vertex v is designed by the user. When input data stream i is coming, streaming function f_v will process data stream i, update v's state from τ to τ', and produce a sequence o of output entries as part of the output streams for downstream vertices.

   A sub-DAG is logically equivalent and can be reduced to one vertex or another sub-DAG. As shown in Figure 11.16, the sub-DAG comprised of vertices v_2, v_3, v_4, and v_5 (as well as all their edges) is a valid sub-DAG and can be reduced to a "vertex" with i as its input stream and o as its output stream.

2. Resilient Substitution

   Resilient substitution is an important feature of TimeStream. It is used to dynamically adjust and reconfigure streaming DAG according to the loading change of a

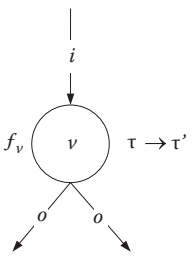

FIGURE 11.15 Streaming DAG.

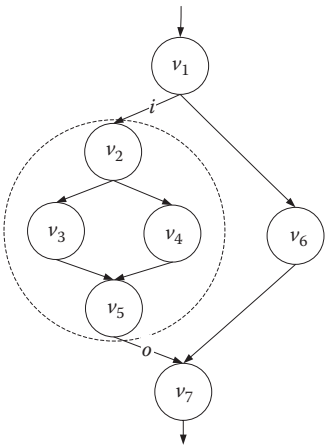

FIGURE 11.16 Streaming DAG and sub-DAG.

data stream. There are three types of resilient substitution in TimeStream. (a) A vertex is substituted by another vertex. When a vertex fails, a new corresponding standby vertex is initiated to replace the failed one and continues execution, possibly on a different machine. (b) A sub-DAG is substituted by another sub-DAG. When the number of instances of a vertex in a sub-DAG needs to be adjusted, a new sub-DAG will replace the old one. For example, as shown in Figure 11.17, a sub-DAG comprised of vertices v_2, v_3, v_4, and v_5 implements three stages: hash partitioning, computation, and union. When the load increases, TimeStream can create a new sub-DAG (shown on the left), which uses four partitions instead of two, to replace the original sub-DAG. (c) A sub-DAG is substituted by a vertex. When the load decreases, there is no need for so many steps to finish a special function, and the corresponding sub-DAG can be substituted by a vertex, as shown in Figure 11.16.

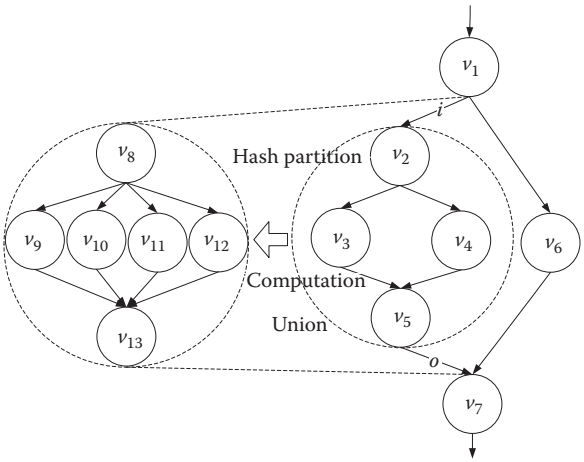

FIGURE 11.17 Resilient substitution.

11.3.3.2 Naiad

Naiad is a distributed system for executing data-parallel, cyclic dataflow programs. The core part is written in C#. It offers high throughput of batch processors and low latency of stream processors and is able to perform iterative and incremental computations. Naiad is a prototype implementation of a new computational model, timely dataflow [30].

1. Timely Dataflow

   Timely dataflow is a computational model based on directed graphs. The dataflow graph can be a directed acyclic graph, like in other BDSC environments. It can also be a directed cyclic graph; the situation of cycles in a dataflow graph is under consideration. In timely dataflow, the time stamps reflect cycle structure in order to distinguish data that arise in different input epochs and loop iterations. The external producer labels each message with an integer epoch and notifies the input vertex when it will not receive any more messages with a given epoch label.

   Timely dataflow graphs are directed graphs with the constraint that the vertices are organized into possibly nested loop contexts, with three associated system-provided vertices. Edges entering a loop context must pass through an ingress vertex, and edges leaving a loop context must pass through an egress vertex. Additionally, every cycle in the graph must be contained entirely within some loop context and include at least one feedback vertex that is not nested within any inner loop contexts. Figure 11.18 shows a single-loop context with ingress (I), egress (E), and feedback (F) vertices labeled.

2. System Architecture

   The system architecture of a Naiad cluster is shown in Figure 11.19, with a group of processes hosting workers that manage a partition of the timely dataflow vertices. Workers exchange messages locally using shared memory and remotely using TCP connections between each pair of processes.

   A program specifies its timely dataflow graph as a logical graph of stages linked by typed connectors. Each connector optionally has a partitioning function to control the exchange of data between stages. At execution time, Naiad expands the logical graph into a physical graph where each stage is replaced by a set of vertices and each connector by a set of edges. Figure 11.19 shows a logical graph and a corresponding

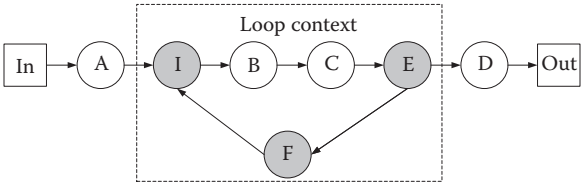

FIGURE 11.18 Timely dataflow graph.

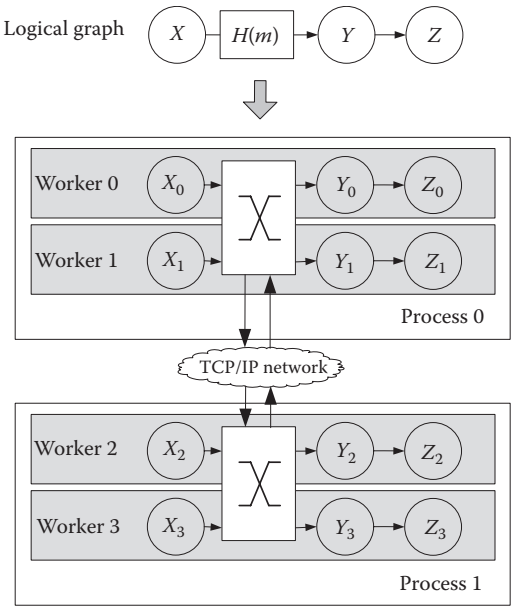

FIGURE 11.19 System architecture of a Naiad cluster.

physical graph, where the connector from X to Y has partitioning function $H(m)$ on typed messages m.

Each Naiad worker is responsible for delivering messages and notifications to vertices in its partition of the timely dataflow graph. When faced with multiple runnable actions, workers break ties by delivering messages before notifications, in order to reduce the amount of queued data.

11.4 FUTURE PERSPECTIVE

In this section, we focus our attention on grand challenges of BDSC and the main work we will perform in the near future.

11.4.1 Grand Challenges

BDSC is becoming the fastest and most efficient way to obtain useful knowledge from what is happening now, allowing organizations to react quickly when problems appear or to detect new trends helping to improve their performance. BDSC is needed to manage the data currently generated at an ever-increasing rate from such applications as log records or click-streams in web exploring, blogging, and twitter posts. In fact, all data generated can be considered as streaming data or as a snapshot of streaming data.

There are some challenges that researchers and practitioners have to deal with in the next few years, such as high scalability, high fault tolerance, high consistency, high load balancing, high throughput, and so on [6,9,31,32]. Those challenges arise from the nature of stream data, that is, data arrive at high speed and must be processed under very strict constraints of space and time.

11.4.1.1 High Scalability

High scalability of stream computing can expand to support increasing data streams and meet the quality of service (QoS) of users, or it can shrink to support decreasing data streams and improve resource utilization. In BDSC environments, it is difficult to achieve high scalability, as the change of data stream is unexpected. The key is that the software changes along with the data stream change, grows along with increased usage, or shrinks along with decreased usage. This means that scalable programs take up limited space and resources for smaller data needs but can grow efficiently as more demands are placed on the data stream.

To achieve high scalability in BDSC environments, a good scalable system architecture, a good effective resource allocation strategy, and a good data stream computing mode are required.

11.4.1.2 High Fault Tolerance

Highly fault-tolerant stream computing can enable a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components. Fault tolerance is particularly sought after in high-availability or life-critical systems. In BDSC environments, it is difficult to achieve high fault tolerance, as the data stream is infinite and real time, and more importantly, most of the data are useless.

To achieve high fault tolerance in BDSC environments, a good scalable high-fault-tolerance strategy is needed, as fault tolerance provides additional resources that allow an application to continue working after a component failure without interruption.

11.4.1.3 High Consistency

Highly consistent stream computing can improve system stability and enhance system efficiency. In BDSC environments, it is difficult to achieve high consistency, as it is hard to decide which nodes should be consistent and which data are needed.

To achieve high consistency in BDSC environments, a good system structure is required. Usually, the master–slave structure is a good choice, as all data are in the master node, and it is easy to achieve highly consistent states.

11.4.1.4 High Load Balancing

Highly load-balanced stream computing can make a stream computing system self-adaptive to the changes of data streams and avoid load shedding. In BDSC environments, it is difficult to achieve high load balancing, as it is impossible to dedicate resources that cover peak loads 24 h a day, 7 days a week. Traditionally, stream computing systems use load shedding when the workload exceeds their processing. This employs a trade-off between delivering a low-latency response and ensuring that all incoming data streams are processed. However, load shedding is not feasible when the variance between peak and average workload is high, and the response should always be kept in real time for users.

To achieve high load balancing in BDSC environments, a good distributed computing environment is needed. It should provide scalable stream computing that automatically

streams a partial data stream to a global computing center when local resources become insufficient.

11.4.1.5 High Throughput

High-throughput stream computing will improve data stream computing ability by running multiple independent instances of a task topology graph on multiple processors at the same time. In BDSC environments, it is difficult to achieve high throughput, as it is hard to decide how to identify the need for a replication subgraph in a task topology graph, to decide the number of replicas, and to decide the fraction of the data stream to assign to each replica.

To achieve high throughput in BDSC environments, a good multiple-instance replication strategy is needed. Usually, the data stream loading of all instances of all nodes in a task topology graph being equal is a good choice, as the computing ability of all computing nodes are efficient, and it is easy to achieve high-throughput states.

11.4.2 On-the-Fly Work

Future investigation will focus on the following aspects:

1. Research on new strategies to optimize a task topology graph, such as subgraph partitioning strategy, subgraph replication strategy, and subgraph allocating strategy, and to provide a high-throughput BDSC environment
2. Research on dynamic extensible data stream strategies, such that a data stream can be adjusted according to available resources and the QoS of users, and provide a highly load-balancing BDSC environment
3. Research on the impact of a task topology graph with a cycle, and a corresponding task topology graph optimize strategy and resource allocating strategy, and provide a highly adaptive BDSC environment
4. Research on the architectures for large-scale real-time stream computing environments, such as symmetric architecture and master–slave architecture, and provide a highly consistent BDSC environment
5. Develop a BDSC system with the features of high throughput, high fault tolerance, high consistency, and high scalability, and deploy such a system in a real BDSC environment

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61170008 and Grant No. 61272055, in part by the National Grand Fundamental Research 973 Program of China under Grant No. 2014CB340402 in part by the National High Technology Research and Development Program of China under Grant No. 2013AA01A210, and in part by the China Postdoctoral Science Foundation under Grant No. 2014M560976.

REFERENCES

1. Storm. Available at http://storm-project.net/ (accessed July 16, 2013).
2. Neumeyer L, Robbins B, Nair A et al. S4: Distributed stream computing platform, *Proc. 10th IEEE International Conference on Data Mining Workshops, ICDMW 2010*, Sydney, NSW, Australia, IEEE Press, December 2010, pp. 170–177.
3. Zhao Y and Wu J. Dache: A data aware caching for big-data applications using the MapReduce framework, *Proc. 32nd IEEE Conference on Computer Communications, INFOCOM 2013*, IEEE Press, April 2013, pp. 35–39.
4. Shang W, Jiang Z M, Hemmati H et al. Assisting developers of Big Data analytics applications when deploying on Hadoop clouds, *Proc. 35th International Conference on Software Engineering, ICSE 2013*, IEEE Press, May 2013, pp. 402–411.
5. Qian Z P, He Y, Su C Z et al. TimeStream: Reliable stream computation in the cloud, *Proc. 8th ACM European Conference on Computer Systems, EuroSys 2013*, Prague, Czech Republic, ACM Press, April 2013, pp. 1–14.
6. Umut A A and Yan C. Streaming Big Data with self-adjusting computation, *Proc. 2013 ACM SIGPLAN Workshop on Data Driven Functional Programming, Co-located with POPL 2013, DDFP 2013*, Rome, Italy, ACM Press, January 2013, pp. 15–18.
7. Demirkan H and Delen D. Leveraging the capabilities of service-oriented decision support systems: Putting analytics and Big Data in cloud, *Decision Support Systems*, vol. 55(1), 2013, pp. 412–421.
8. Albert B. Mining Big Data in real time, *Informatica (Slovenia)*, vol. 37(1), 2013, pp. 15–20.
9. Lu J and Li D. Bias correction in a small sample from Big Data, *IEEE Transactions on Knowledge and Data Engineering*, vol. 25(11), 2013, pp. 2658–2663.
10. Tien J M. Big Data: Unleashing information, *Journal of Systems Science and Systems Engineering*, vol. 22(2), 2013, pp. 127–151.
11. Zhang R, Koudas N, Ooi B C et al. Streaming multiple aggregations using phantoms, *VLDB Journal*, vol. 19(4), 2010, pp. 557–583.
12. Hirzel M, Andrade H, Gedik B et al. IBM streams processing language: Analyzing Big Data in motion, *IBM Journal of Research and Development*, vol. 57(3/4), 2013, pp. 7:1–7:11.
13. Dayarathna M and Toyotaro S. Automatic optimization of stream programs via source program operator graph transformations, *Distributed and Parallel Databases*, vol. 31(4), 2013, pp. 543–599.
14. Farhad S M, Ko Y, Burgstaller B et al. Orchestration by approximation mapping stream programs onto multicore architectures, *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011*, ACM Press, March 2011, pp. 357–367.
15. Schneider S, Hirzel M and Gedik B. Tutorial: Stream processing optimizations, *Proc. 7th ACM International Conference on Distributed Event-Based Systems, DEBS 2013*, ACM Press, June 2013, pp. 249–258.
16. Khandekar R, Hildrum K, Parekh S et al. COLA: Optimizing stream processing applications via graph partitioning, *Proc. 10th ACM/IFIP/USENIX International Conference on Middleware, Middleware 2009*, ACM Press, November 2009, pp. 1–20.
17. Scalosub G, Marbach P and Liebeherr J. Buffer management for aggregated streaming data with packet dependencies, *IEEE Transactions on Parallel and Distributed Systems*, vol. 24(3), 2013, pp. 439–449.
18. Malensek M, Pallickara S L and Pallickara S. Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals, *Future Generation Computer Systems*, vol. 29(4), 2013, pp. 1049–1061.
19. Cugola G and Margara A. Processing flows of information: From data stream to complex event processing, *ACM Computing Surveys*, vol. 44(3), 2012, pp. 15:1–15:62.

20. Storm wiki. Available at http://en.wikipedia.org/wiki/Storm (accessed July 16, 2013).
21. Storm Tutorial. Available at https://github.com/nathanmarz/storm/wiki (accessed July 16, 2013).
22. Chauhan J, Chowdhury S A and Makaroff D. Performance evaluation of Yahoo! S4: A first look, *Proc. 7th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2012*, Victoria, BC, Canada, IEEE Press, November 2012, pp. 58–65.
23. Simoncelli D, Dusi M, Gringoli F et al. Scaling out the performance of service monitoring applications with BlockMon, *Proc. 14th International Conference on Passive and Active Measurement, PAM 2013*, Hong Kong, China, IEEE Press, March 2013, pp. 253–255.
24. S4, distributed stream computing platform. Available at http://incubator.apache.org/s4/ (accessed July 16, 2013).
25. Stream computing StreamBase Yahoo S4 borealis comparis. Available at http://oracle-abc.wikidot.com/zh:stream-computing-streambase-yahoo-s4-borealis-comparison (accessed July 16, 2013).
26. Guo Z Y, Sean M D, Yang M et al. Failure recovery: When the cure is worse than the disease, *Proc. 14th USENIX conference on Hot Topics in Operating Systems, USENIX 2013*, Santa Ana Pueblo, NM, ACM Press, May 2013, pp. 1–6.
27. Ali M, Chandramouli B, Goldstein J et al. The extensibility framework in Microsoft StreamInsight, *Proc. IEEE 27th International Conference on Data Engineering, ICDE 2011*, Hannover, Germany, IEEE Press, April 2011, pp. 1242–1253.
28. Chandramouli B, Goldstein J, Barga R et al. Accurate latency estimation in a distributed event processing system, *Proc. IEEE 27th International Conference on Data Engineering, ICDE 2011*, Hannover, Germany, IEEE Press, April 2011, pp. 255–266.
29. Ali M, Chandramouli B, Fay J et al. Online visualization of geospatial stream data using the WorldWide telescope, *VLDB Endowment*, vol. 4(12), 2011, pp. 1379–1382.
30. Derek G M, Frank M S, Rebecca I et al. Naiad: A timely dataflow system, *Proc. the 24th ACM Symposium on Operating Systems Principles, SOSP 2013*, Pennsylvania, ACM Press, November 2013, pp. 439–455.
31. Garzo A, Benczur A A, Sidlo C I et al. Real-time streaming mobility analytics, *Proc. 2013 IEEE International Conference on Big Data, Big Data 2013*, Santa Clara, CA, IEEE Press, October 2013, pp. 697–702.
32. Zaharia M, Das T, Li H et al. R Discretized streams: Fault-tolerant streaming computation at scale, *Proc. the 24th ACM Symposium on Operating Systems Principles, SOSP 2013*, Farmington, PA, ACM Press, November 2013, pp. 423–438.