StroMAX: Partitioning-based Scheduler for Real-time Stream Processing System

Jiawei Jiang, Zhipeng Zhang, Bin Cui, Yunhai Tong, Ning Xu

School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
{blue.jwjiang, zhipengzhang, bin.cui, yhtong, ning.xu}@pku.edu.cn

Abstract. With the increasing availability and scale of data from Web 2.0, the ability to efficiently and timely analyze huge amounts of data is important for industry success. A number of real-time stream processing platforms have been developed, such as Storm, S4, and Flume. A fundamental problem of these large scale decentralized stream processing systems is how to deploy the workload to each node so as to fully utilize the available resources and optimize the overall system performance. In this paper, we present StroMAX, a graph-partitioning based approach of workload scheduling for real-time stream processing systems. StroMAX uses two advanced generic schedulers to improve the performance of stream processing systems by reducing the inter-node communication cost while keeping the workload of nodes below a certain computational load threshold. The first scheduler analyzes the workload structure when a job is committed and uses the graph-partitioning result to determine the deployment of tasks. The second scheduler monitors system performance, analyzes the statistical information of physical nodes, and dynamically reassigns the tasks during runtime to improve the overall performance. Besides, StroMAX can be used and deployed to many other state-of-the-art real-time stream processing systems easily. We implemented StroMAX on Storm, a representative real-time stream processing system. Extensive experiments conducted with real-world workloads and datasets demonstrate the superiority of our approaches against the existing solutions.

Keywords: Real-time stream processing, Task allocation, Workload scheduling, Graph partition

1 Introduction

With the unprecedented proliferation of data from web, it is natural to extend the scope to efficient processing mechanisms and methods that can handle real-time workloads [19]. For example, Twitter, the popular online social network, processes over 500 million tweets every day. It is challenging to process and analyze such a big data stream in real-time. Traditional distributed processing frameworks, such as MapReduce, are designed for offline batch processing. They are ill-suited to process real-time workloads. Real-time stream computing is an effective way to process big data with low-latency. It is becoming one of the fastest and most efficient ways to obtain useful knowledge from various kinds of real-time data. Thus, many real-time stream processing frameworks have been proposed, such as Storm [21], S4 [14], and Flume [1].

Compared to the batch processing systems, resource allocation and scheduling in real-time stream processing systems are much more difficult and important due to the dynamic nature of the input data streams. An application or workload in these systems consists of several processing components. A component can produce the input of stream or execute the processing logic to generate results. In this paper, we use **Spout** and **Bolt**, borrowed from Storm, to represent the input component and processing component, respectively. Tuples emitted by a spout constitute a stream that can be transformed by passing through one or more bolts that implement the user-defined logic. Therefore, we can use a directed acyclic graph, called a topology, to denote the stream transformations. When a topology is submitted, the system schedules the tasks of each spout and each bolt to a certain physical node of the cluster. Similar to the batch data processing systems such as Hadoop, the allocation strategy impacts the performance of a real-time stream processing system. However, most of the above systems apply a round-robin method as their default scheduler which evenly distributes the components of a topology to the physical nodes. This basic scheduler is easy to implement, however, it does not take into account the cost of tuple transmission between components. Furthermore, the communication cost of tuple transmission heavily increases the average processing latency and deteriorates the overall performance of the system.

In this paper, we design and implement StroMAX, which provides two novel schedulers for the real-time stream processing systems to improve their performance. Different from the default round-robin scheduler, StroMAX aims at reducing the average processing latency of tuple by minimizing the total inter-node communication cost and keeping computational load balanced on each node. These two schedulers use graph-partitioning based algorithms to partition the topology. The first scheduler, named *Bootstrap Scheduler*, analyzes the topology graph and partitions the topology when it is submitted to the system. This strategy is simple and is executed before the topology is started, so neither the cluster workload nor the network traffic is taken into account. The second scheduler, named *Rebalance Scheduler*, goes one step further by monitoring the runtime statistics of all the topologies and the workload of cluster, then it rebalances the topologies for overall performance optimization when necessary. Besides, Rebalance Scheduler provides a heuristic to dynamically move components from the bottleneck nodes to the idle ones based on the statistical information of the cluster and the topologies.

To evaluate our schedulers, we implemented StroMAX on Storm. The performance of StroMAX is validated with several real-world workloads. The experimental results show that the proposed graph-partitioning based approaches significantly outperform the original scheduler and demonstrate superior scalability on both synthetic benchmarks and real-world scenarios.

Our contributions in this paper can be summarized as follows:

1. We propose Bootstrap Scheduler which analyzes the graph structure of the input topology and partitions the topology when it is committed to the system.
2. We design Rebalance Scheduler that generates a global-topology-graph and partitions all the topologies to the nodes so as to improve the overall performance of the system. In addition, Rebalance Scheduler provides a novel mechanism to dynamically reassign the components when necessary.
3. We implement StroMAX on Storm, a prevailing open-source real-time stream processing system. We conduct extensive experimental studies to exhibit the advantages of our approach.

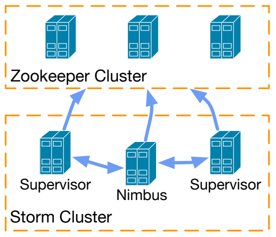
The remaining of this paper is organized as follows. In Section 2, we review the representative real-time systems and relevant performance issues. In Section 3, we present the Bootstrap Scheduler and Rebalance Scheduler, followed by the architecture of StroMAX in Section 4. Section 5 reports the results of extensive experimental studies. Finally, we introduce the related work and conclude this paper in Section 6 and 7.

2 Background

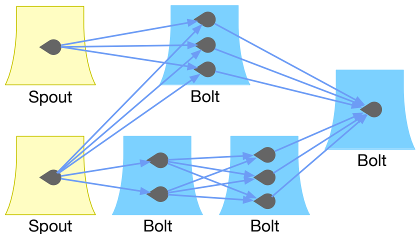
In this section, we first introduce Storm on which our prototype system is built. We next introduce the weakness of the default scheduler and analyze the cost of inter-node and inner-node communication.

Architecture of Storm. Apache Storm [21] is an open-source distributed real-time stream computation system. For parallelism, Storm uses two levels of abstractions: physical and logical.

– *Physical*: Storm consists of a master node (Nimbus), a number of zookeeper nodes that serve as a control unit, and a set of slave physical nodes (Supervisors) which process stream workload as shown in Figure 1a.



a. Physical Abstractions.



b. Topology Example.

Fig. 1: Architecture of Storm.

– *Logical*: As shown in Figure 1b, a Storm workload, called a *topology*, is a *directed acyclic graph* (DAG). Each vertex represents a processing component and each edge represents data transferred between two components. As mentioned in Section 1, there are two types of components: spout and bolt. The spouts provide a general mechanism to emit tuples into a topology. The bolts consume tuples from spouts or other bolts, and process them in the way defined by the user. Each component consists of a group of *tasks* communicating with other groups of *tasks* connected to it. A task can be considered as an instance of a spout or bolt.

When a topology is committed to a Storm cluster, the tasks are assigned to the physical nodes. Consequently, we need a scheduler to determine the assignment.

Scheduler in Real-time Stream Processing Systems. The default scheduler used in state-of-the-art systems is even scheduler. It enforces a round-robin strategy to balance the computation cost of each physical node, however, it lacks the consideration of communication cost. There are two types of communication among tasks. If two connected tasks are assigned to the same physical node, they use inner-node communication mechanism; otherwise, they use inter-node communication. Generally speaking, inter-node communication is much slower than inner-node communication. Therefore, we need to minimize inter-node communication while keeping computation load balanced on each node.

3 Graph-partitioning Based Schedulers

In this section, we first introduce the notations and our graph partitioning models of the scheduling problems. Then we present the graph-partitioning based schedulers. Table 1 lists the symbols used in this paper.

3.1 Problem Definition

Graph Partitioning. Given a graph $G(V, E)$ where V denotes the set of vertices and E denotes the set of edges, we let $P = \{P_1,...,P_k\}$ be k subsets of V. P is defined to be a partition of G if: $P_i \neq \emptyset$, $P_i \cap P_j = \emptyset$, and $\cup P_i = V$ (i, j = 1,...,k; $i \neq j$). The number k is called the cardinality of the partition. Graph partitioning problem is to find an optimal partition P based on an objective function. Here we give a formal definition:

Definition 1 *The graph partitioning problem can be defined by a triplet (S, p, f). S is a discrete set of all the partitions of G. p is a predicate on S which creates a admissible solution set $S_p \in S$. All the partitions in S_p is admissible for p. The aim is to find a partition $\bar{P} \in S_p$ that minimizes the objective function $f(P)$:*

$$\bar{P} = arg \min_{P \in S_p} f(P) \tag{1}$$

Graph Partitioning in Real-time Stream Processing System. For real-time stream processing systems, we can use $G(V, E)$, a directed acyclic graph, to represent the topology T. The vertex $v_i \in V$ is the i-th component in the topology which can be a spout or a bolt. As mentioned above, each processing component

Symbols	Description
$G_t(V_t, E_t)$	task graph
$G_g(V_g, E_g)$	global task graph
n_i	i-th physical node
$\varpi(n_i)$	maximum processing capability of node n_i
$\omega(n_i)$	capacity used in node n_i
v_i, P_i	i-th task and i-th set of tasks
$\|v_i\|, \|P_i\|$	computation cost of task v_i and the set P_i
$N(v)$	neighbors of vertex v
$Edgecut(P_i, P_j)$	number of cross edges between two sets P_i, P_j
$Comm(P_i, P_j)$	communication cost between two sets P_i, P_j
$rc(v_i, v_j)$	bandwidth cost between two tasks v_i, v_j
$\mathit{\Gamma}(v_i)$	total inter-node communication cost of v_i

Table 1: Notations.

consists of several tasks. Thus, we have $v_i = \{t_i^1, t_i^2, ..., t_i^m\}$, where t_i^n is the n-th task for processing component v_i and m is the number of parallelized tasks of the i-th component. The edge $(i, j) \in E$ denotes each task in v_i is connected to each task in v_j. Then, we can use a directed acyclic graph, $G_t(V_t, E_t)$, to represent the graph of tasks. A vertex v_i in V_t represents a task and an edge (i, j) in E_t represents the connection from task v_i to v_j. The data flow of the topology is organized as a graph of tasks. Here we give a formal definition of the scheduling methods based on graph-partitioning in real-time systems.

Definition 2 *Given a task graph $G_t(V_t, E_t)$, where each vertex represents a task and each edge denotes the data flows between them, the goal of a graph-partitioning based scheduling method is to partition G_t into k parts, so that each part has the same number of tasks and the number of edges between different parts is minimized. We assume that each part P_i is allocated to the i-th physical node.*

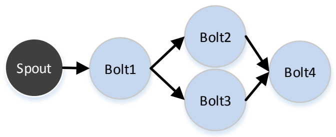
3.2 Bootstrap Scheduler

Motivation. In real-time stream processing systems, the key of the scheduling algorithm is to balance the computation cost and minimize the communication cost. The even scheduler achieves balanced computation, while overlooks the importance of communication cost. Since the processing latency is dominated by inter-node transfer time, reducing the tuples sent through the network can help to improve the performance. In this section, we propose **Bootstrap Scheduler** that considers both computation cost and communication cost.

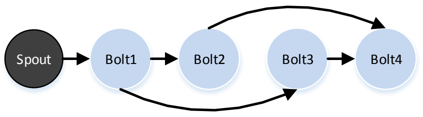
Modeling the Node Capability and Tuple Cost. We first formally model the capability of physical nodes and the cost of tuples. Given a cluster consisting of m physical nodes — $N = \{n_1, ..., n_m\}$, we define that the maximum processing capability of node n_i is $\varpi(n_i)$, and the current computation capacity of n_i is denoted as $\omega(n_i)$. For Bootstrap Scheduler, which is executed before the topology is actually executed, we cannot measure the computation cost of a task to process a tuple and the communication cost to transfer a tuple between two tasks. Therefore, we assume that the communication cost to transfer a tuple is equal to one and the computation cost to process a tuple is equal to one for all the tasks. In other words, $\varpi(n_i)$ denotes the number of tasks each node can handle, while $\omega(n_i)$ denotes that already handled.

Graph Partitioning in Bootstrap Scheduler. The goal for Bootstrap Scheduler is to partition $G_t(V_t, E_t)$ into m parts — $P = \{P_1, ..., P_m\}$, and then assign each part P_i to the physical node n_i, so that the total inter-node communication cost is minimized and the processing cost does not exceed each node's maximum capacity $\varpi(n_i)$. Therefore, we can formalize the objective function for Bootstrap Scheduler:

$$f(P) = \sum_{i,j \in [1,m], i \neq j} (Edgecut(P_i, P_j)) \tag{2}$$



a. Original Topology



b. Linearization Result

Fig. 2: An Example of Topology Linearization.

$$S_p = \{P \in S \ , \ |P_i| \leq \varpi(n_i) \ , i \in [1, m]\} \tag{3}$$

where $Edgecut(P_i, P_j)$ denotes the number of cross edges between P_i and P_j, and $|P_i|$ denotes the computation cost of P_i. Then $f(P)$ measures the total communication cost of graph $G_t(V_t, E_t)$ and S_p is the set of admissible solutions. Based on Equation 2 and 3, the aim is to find the partition $\bar{P} \in S_p$ that minimizes f:

$$\bar{P} = arg \min_{P \in S_p} \sum_{i,j \in [1,m], i \neq j} (Edgecut(P_i, P_j)) \tag{4}$$

This graph partitioning problem is NP-hard by reducing it to Task Allocation Problem [4]. Bootstrap Scheduler leverages a linear streaming method to solve the above graph partitioning problem.

If the vertices of the task graph arrive in some order with the set of their neighbors, and we partition the graph based on the vertex stream, it is called a streaming graph partitioning, which is fast and easy to implement. Streaming graph partitioning decides which part to assign the incoming vertex to. Once the vertex is placed, it will not be removed. This algorithm makes decisions based on incomplete information; therefore, the order of vertex stream will significantly affect the performance [20]. In this paper, we use a novel linearization approach to get the stream order.

Topology Linearization. We linearize the topology based on the property of DAG using topological sorting. Given $G_t(V_t, E_t)$, if a task v_i emits tuples into a stream that is consumed by another task v_j, then we have $v_i < v_j$, where the $<$ denotes the partial order between v_i and v_j. If $v_i < v_j$ and $v_j < v_k$, we have $v_i < v_k$ by transitivity of partial order. Since we deal with acyclic graphs, we can determine a linearization $\mathcal{L}$ of the components according to the partial order: ① If $v_i < v_j$ holds, then v_i appears in $\mathcal{L}$ before v_j. ② If neither $v_i < v_j$ nor $v_i > v_j$ holds, v_i and v_j can appear in $\mathcal{L}$ in any order. ③ The first element of $\mathcal{L}$ is a random spout task v_k of the topology. Figure 2 showcases an example of a linearization of a topology with 5 components.

The linearization approach generates a linear order of tasks for the input stream. Then we study a one-pass method to partition the graph with this order. There is a streaming loader to successively read vertices (tasks), and send them to the partition program. Afterwards, the program determines the assignment of each incoming vertex (task) according to the current partition state and vertex information.

Intuition for Task Assignment. There are two intuitions the task (vertex) assignment should consider.

1. The first intuition is that we need to assign a task to the physical node that has less running tasks, in order to balance the computation cost and prevent too much computational load on one node.
2. The second intuition is that we need to assign a task to the physical node that has more neighbors of the task, in order to minimize the inter-node communication cost.

A Heuristic Solution. Motivated by these two intuitions, we use a novel streaming heuristic to solve the graph partitioning problem, i.e., to decide which part to assign the incoming vertex (task) v to.

$$index = arg \max_{i \in [1,m]} \{|P_i \cap N(v)| \left(1 - \frac{|P_i|}{\varpi(n_i)}\right)\} \tag{5}$$

In the above equation, m is the number of the partitions, $\varpi(n_i)$ is the total capacity of physical node n_i, and $N(v)$ is the set of neighbors of vertex v. For each node n_i, the first part $|P_i \cap N(v)|$ measures the number of neighbors of the incoming vertex, and the second part $(1-|P_i|/\varpi(n_i))$ measures the computation idleness. In other words, we make a combinatorial decision considering both balancing the computation load and minimizing the inter-node communication.

Algorithm 1 Bootstrap Scheduler

```
Require: # of physical node: $m$, DAG: $G_t(V_t, E_t)$.
Ensure: Partition $P = \{P_1, P_2, ..., P_m\}$ for $G_t(V_t, E_t)$.
 1: $\mathcal{L} \leftarrow \varnothing$, $S \leftarrow$all vertices $v$ with $in(v) = 0$, $P_i \leftarrow \varnothing$
 2: for each vertex $v$ in $S$ do
 3:     $S = S - v$
 4:     $\mathcal{L} = \mathcal{L} \cup v$
 5:     for each vertex $u$ that has $edge(v, u) \in E_t$ do
 6:         $E_t = E_t - edge(v, u)$
 7:         if $in(u) = 0$ then
 8:             $S = S \cup u$
 9:         end if
10:     end for
11: end for
12: if $E_t \neq \varnothing$ then
13:     return Error: the graph is not DAG.
14: end if
15: SL = streamingloader($\mathcal{L}$)
16: for each vertex $v$ in $SL$ do
17:     $index = arg \max\{|P_i \cap N(v)| \left(1 - \frac{|P_i|}{\varpi(n_i)}\right)\}$;
18:     Insert vertex $v$ into $P_{index}$;
19: end for
20: return $P$
```

Algorithm 2 Dynamic-Task-Reassignment

```
Input: Partition result: $P = \{P_1, ..., P_m\}$; $\theta$; $G_g = (V_g, E_g)$ and $rc(v_i, v_j)$ $v_i, v_j \in V_g$.
 1: for each $P_i$ in $P$ do
 2:     if $\sum_{v \in P_i} |v| > \theta$ then
 3:         $List \leftarrow$ Sort $\{\Gamma(v), v \in P_i\}$ in non-descending order
 4:         while $\sum_{v \in P_i} |v| > \theta$ do
 5:             $v_t = pop(List)$
 6:             $j = arg \max_{v_j \in P_j, i \neq j}\{\sum rc(v_t, v_j)\}$
 7:             Reassign $v_t$ to the node $j$
 8:             $V_i = V_i$ - $v_t$
 9:         end while
10:     end if
11: end for
```

Let $in(v)$ denote the incoming degree of vertex v, we summarize Bootstrap Scheduler in Algorithm 1. The topology linearization is executed in line 1-14. With the streaming graph-partitioning heuristic, we partition the task graph $G_t(V_t, E_t)$ into m parts (line 15-19). Finally we assign the tasks to m physical nodes according to the partition result P.

3.3 Rebalance Scheduler

In this section, we propose **Rebalance Scheduler** which leverages the runtime statistics to assign all the topologies to improve the overall performance. Rebalance Scheduler uses two techniques for task reallocation, i.e., *global-topology-graph-partitioning* to repartitions all the topologies and *dynamic-task-reassignment* to move tasks from skew nodes to idle ones automatically. Following, we first discuss the motivation, and then describe these two techniques in detail.

Motivation. Bootstrap Scheduler produces an initial assignment of the tasks when it is submitted. The goal of Bootstrap Scheduler is to allocate tasks to physical nodes so as to satisfy the constraint on the number of running tasks on each physical node and minimize the inter-node communication cost. Bootstrap Scheduler is executed before the topology actually runs and only considers the newly committed topology. In practice, however, there are other topologies running on the system before the new topology is committed. Therefore, the scheduler should allocate tasks based on all the running topologies. Besides, for Bootstrap Scheduler, we assume that the communication cost to transfer a tuple equals to one and the computation cost to process a tuple equals to one for all the tasks. However, in practice, the computation cost to process a tuple and the communication cost to transfer a tuple are significantly different for different tasks.

Traditional database systems use collected historical statistics to estimate the running time for query optimization. For real-time stream processing systems, we also study the strategy that collects historical information to estimate the computation cost and communication cost of each task during the execution.

Metrics of Computation Cost and Communication Cost. To measure the runtime statistics in terms of the computation cost and communication cost of each task, we use two metrics as described below.

1. *Computation cost of a task* is measured by the average computation time for processing a tuple for a certain task. To measure this metric, we use the running logs to estimate the computation cost of a certain task during the execution. In order to deal with the heterogeneity of nodes in the clusters (different computational abilities such as different CPU frequencies), we need to consider the CPU speed of the nodes. For example, if a task takes 10 millisecond on a 1GHz CPU, then migrating the task to a node with 2GHz CPU would generate about a time cost of 5 millisecond. For this reason, we measure the computation cost unit as the multiplication of CPU frequency(GHz) and time(millisecond). Specially, in the above example, the computation cost for that task to process one tuple is 10.
2. *Communication cost between two tasks* is measured by the average size of data transferred between them. Similar as the measurement of the computation cost, we log the size of package for the tuple transferred from one task to the other during the execution. Then we use the average size of package for transferring one tuple as the average communication cost. We use 1024 Bytes as the unit. For example, if the average size of a tuple transferred from task i to task j is 5 KB, then the communication cost between i and j is 5.

Global-Topology-Graph Partitioning. In order to better partition the tasks of the running topologies, we model the global-topology-graph which contains all the topologies running on the cluster. The global-topology-graph is represented as a weighted directed acyclic graph $G_g(V_g, E_g)$, where V_g is the set of all the tasks in the cluster and E_g is the set of connections between tasks. The weight of each vertex v_i represents the computation cost of task v_i, denoted as $comp(v_i)$. The weight of each edge (v_i, v_j) represents the communication cost between task v_i and v_j, denoted as $comm(v_i, v_j)$. Let $G_t^i(V_t^i, E_t^i), i \in [1, m]$ be a single topology running on the system, we can generate the global-topology-graph G_g:

$$G_g = (\bigcup_{i\in[1,m]} V_t^i, \bigcup_{i\in[1,m]} E_t^i)$$

The global-topology-graph is the combination of all the topologies run on the system with the weight of the computation cost and communication cost. With the global-topology-graph, we can allocate the topologies based on the global information of all the topologies. Besides, the vertex weight and edge weight of global-topology-graph provide us the information of the heterogeneity of the tasks which further improves the accuracy of the partition result.

Given a global-topology-graph $G_g(V_g, E_g)$, our goal is to partition the graph $G_g(V_g, E_g)$ into m parts — $P = \{P_1, ..., P_m\}$, and assign each part P_i to a physical node n_i, so that the inter-node communication cost is minimized and the processing cost on each node does not exceed the maximum capacity $\varpi(n_i)$. We define our objective function $f(P)$ and the admissible solution set S_p as follows:

$$f(P) = \sum_{i,j\in[1,m],i\neq j} (Comm(P_i, P_j)) \tag{6}$$

$$S_p = \{P \in S \ and \ |P_i| < \varpi(n_i), i \in [1, m]\} \tag{7}$$

where P_i denotes the vertex set of the i-th part, and $Comm(P_i, P_j)$ denotes the sum of communication cost between P_i and P_j which can be computed as:

$$Comm(P_i, P_j) = \sum_{v_i\in P_i} \sum_{v_j\in P_j} comm(v_i, v_j) \tag{8}$$

This graph partitioning problem aims to find the partition $\bar{P} \in S_p$ that minimizes:

$$\bar{P} = arg \min_{P\in S_p} \sum_{i,j\in[1,m],i\neq j} Comm(P_i, P_j) \tag{9}$$

This graph partitioning problem is known as the k-balanced graph partitioning problem and has been proved NP-hard [3]. Similar to Bootstrap Scheduler, we use a streaming graph partitioning heuristic to solve the k-balanced graph partitioning problem. We first use topological sorting to linearize the global-topology-graph into a linearized vertex stream $\mathcal{L}$. For a vertex v from $\mathcal{L}$, we use the following partitioning heuristic to determine which node to assign it to.

$$index = arg \max_{i \in [1,m]} \{ \sum_{x \in P_i \cap N(v)} |x| \left(1 - \frac{|P_i|}{\varpi(n_i)} \right) \} \tag{10}$$

where $\varpi(n_i)$ is the maximum computation ability of physical node i, $N(v)$ is the set of neighbors of vertex v, $|x|$ is the computation cost of task x, and $|P_i|$ is the sum of computation cost of tasks on physical node P_i. Different from Bootstrap Scheduler, we take into consideration the real-time computation cost of all the tasks on each physical node. The basic intuition is quite similar, we choose a physical node most relevant to task v with most capacity remained when assigning task v.

Dynamic-Task-Reassignment. During the execution of the system, we further use the statistics of log to monitor the running status of each node. If some nodes become bottlenecks of the whole system, we dynamically reassign the tasks from these nodes to other nodes with less workload to improve the overall performance. We use a threshold θ to judge whether a node is skew enough and needed to reassign its tasks to other nodes. The threshold is defined as follows:

$$\theta = \vartheta * \frac{\sum_{i \in [1,m]} |P_i|}{m} \tag{11}$$

Here ϑ is the percentage we will discuss in Section 5.3, $|P_i|$ is the total computation cost of node i and m is the number of the physical nodes. The basic intuition is that, if the computation cost $|P_i|$ of physical node i is higher than the average cost among the cluster by θ, then we move some tasks from that node to others.

Here we propose a novel heuristic to move the tasks to an appropriate node. The movement tries to minimize the inter-node communication cost and rebalance the computation cost of each node in the cluster. For a task v_i in the system, StroMAX logs the bandwidth cost of the communication between two tasks, denoted by $rc(v_i, v_j)$. If v_i and v_j are on the same node, we have $rc(v_i, v_j)$ = 0. Let $N(v)$ be the neighbor set of task v_i, the total inter-node bandwidth of v_i is denoted as $\mathit{\Gamma}(v_i)$.

$$\mathit{\Gamma}(v_i) = \sum_{v_j \in N(v)} rc(v_i, v_j)$$

The algorithm of reallocating tasks is illustrated in Algorithm 2. If node i needs a reassignment, we first choose the tasks with more inter-node communication according to $\mathit{\Gamma}(v_i)$. Specially, we sort the remote communication bandwidth of each task in a non-descending order (line 3), and greedily reassign the task with the max $\mathit{\Gamma}(v_i)$ to the node which communicates with v_i most frequently, i.e., maximal communication cost (line 5-7). When a task is removed, the estimated communication cost of the node will be decreased by $\mathit{\Gamma}(v_i)$. The reassignment stops when the total computation cost is below the threshold.

4 The StroMAX Architecture

System Architecture. Figure 3 shows how StroMAX is integrated into Storm. Note that our system can be migrated to other real-time stream processing systems as well. For the Storm cluster, there are three types of nodes — one master node called the nimbus node, zookeeper nodes, and worker nodes. When a new topology is submitted, the nimbus allocates the tasks to the workers and monitors failures. The zookeeper maintains the coordination state of nimbus and worker nodes. The topologies are executed on the worker nodes where a supervisor daemon is run on each worker for communication with zookeeper.

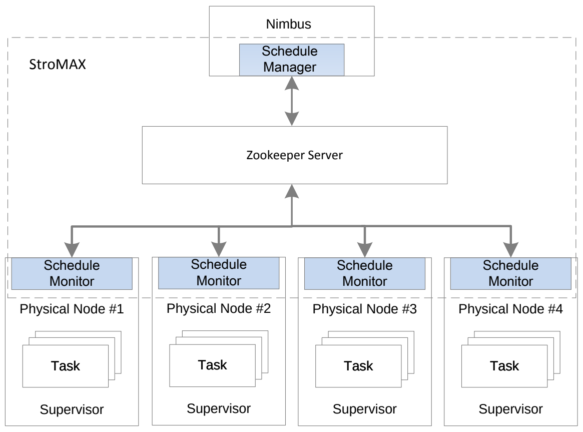


Fig. 3: Architecture of StroMAX.

The components of StroMAX run on the nimbus node and worker nodes. There is a schedule manager running on the nimbus node that provides meta data for partitioning. It stores the meta data of the cluster and log statistics submitted from StroMAX monitors. When a new topology is committed, the schedule manager analyzes and partitions the topology by Bootstrap Scheduler. Then, it triggers the global-topology-graph-partitioning and dynamic-task-reassignment to rebalance the tasks running on the cluster. A schedule monitor runs on each worker node to record, collect, and report log information to the schedule manager. It also calculates the computation and communication measurements. For example, we use the Java API to log the CPU time for 1000 tuples and then calculate the average computation cost.

Implementation of Task Reassignment. To reassign the tasks, we use the Storm infrastructure, which supports suspending and resuming tasks during runtime. Storm blocks the spouts, and thus prevents new stream from being propagated to the bolts and forwarded through the topology. Then all of the in-flight data is propagated through the bolts until all communication queues among these bolts are empty. Our scheduler then reconfigures the cluster by reassigning tasks to proper physical nodes.

5 Evaluation

In this section, we conduct extensive experiments to evaluate StroMAX. We first describe the experimental setup, then present and discuss the performance with different workload settings.

5.1 Experimental Settings

In this section, we briefly introduce the experimental settings for the evaluation, including the cluster, workload, and evaluation metrics. All the evaluation results are measured by average of five executions.

Cluster. All the experiments were conducted on a cluster of 42 nodes. Each node was equipped with two 2.80GHz Intel Xeon E5-2680 CPU, 2GB memory, and 48GB SSD disk. All the nodes were connected by 1Gb bandwidth routers. We used one node for the nimbus, one for the zookeeper, and 40 for the workers.

Workload. Experiments are conducted with six data processing topologies as illustrated in Figure 4 and described below — word count (WC), throughput test (TT), twitter trending topics (TWTT), log processing (LP), twitter stream sentiment analysis (TSSA), and synthetic communication (SC). We compared our proposed schedulers against the default scheduler in Storm. To find a proper tuple input rate for each topology, we first used a low initial rate and increased it gradually, until the average CPU usage of the cluster is above 50%. Then we used this rate in the whole evaluation.

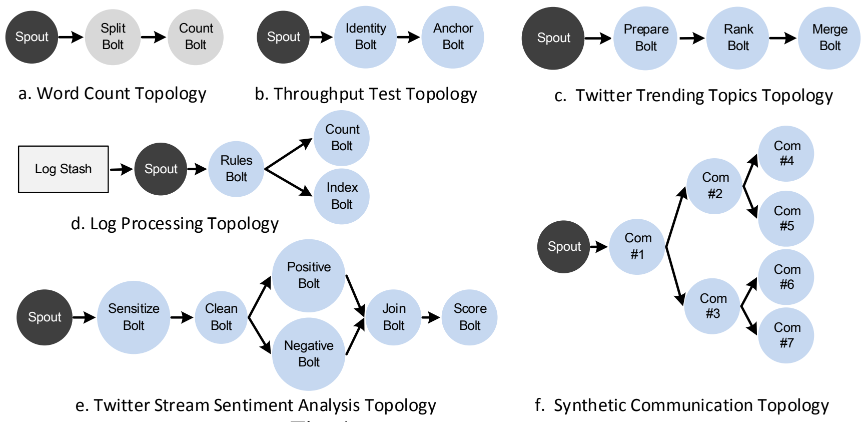
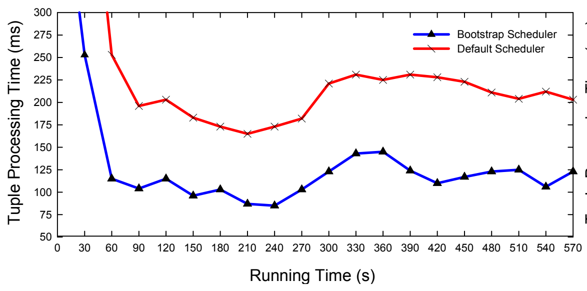


Fig. 4: Evaluated Topologies.

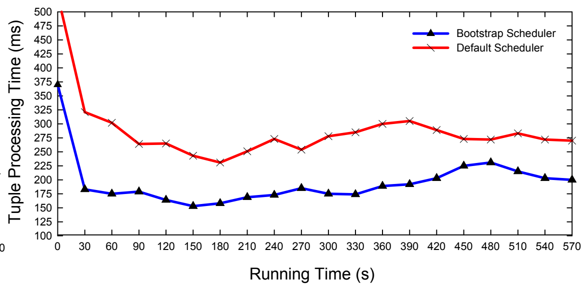
1. *Word Count (WC)*: *WC* is a basic topology shown in Figure 4a. It has one spout and two bolts. The Spout reads English words one line at a time from a local file which is made from 10 thousand random pages crawled from Wikipedia. The Split Bolt splits each line into words and passes them to the Count Bolt. The Count Bolt increases the counters based on distinct input word tuples and produces the results.
2. *Throughput Test (TT)*: *TT* has one spout and two bolts as shown in Figure 4b. The Spout repeatedly generates random 10 KB strings as input tuples. The Identity Bolt emits the received tuples to the Anchor Bolt without any change. The Anchor Bolt increases a counter by one and records the processing time.
3. *Twitter Trending Topics (TWTT)*: This topology computes the top-k trending twitter topics as shown in Figure 4c. The topology is a pipeline of one spout and three bolts. The Spout pushes tweets into the topology. Then the Prepare Bolt updates the counter of each topic, partitions topics alphabetically, and propagates the topic/count pairs. The Rank Bolt receives the topic/count pairs and maintains a list of top-k topics. Finally, the Merge Bolt merges all the lists to produce a single list of the current top-k topics. We used a dataset with one million English tweets from Twitter4j API.
4. *Log Processing (LP)*: *LP* presents a real-world case of log processing which is shown in Figure 4d. The Spout reads tuples from an open-source log agent, Log Stash, as the input for the topology. The Log Stash reads log information from local file which is the kernel logs of Ubuntu server of our lab. The Rules Bolt performs a rule-based analysis on the log and emits log entry tuples to the Index Bolt and Counter Bolt. The Index Bolt and Counter Bolt perform the indexing and counting operations on the log entries respectively.
5. *Twitter Stream Sentiment Analysis (TSSA)*: This topology analyzes sentiment of tweets. The Spout, as shown in Figure 4e, parses the Twitter JSON data and emits tuples into the topology. The Sensitize Bolt performs the first-round data sensitization which removes all non-alpha characters. Following, the Clean Bolt performs the next round of data cleaning by removing stop words to reduce noise for the classifiers. The Positive Bolt and Negative Bolt are two classifiers for the positive and negative classes. Next, Join Bolt joins the scores from the two previous classifiers, and the Score Bolt compares the scores from the classifiers and declares the class accordingly. We used the same dataset for twitter trending topics from Twitter4j API.
6. *Synthetic Communication (SC)*: This is a synthetic topology as shown in Figure 4f. The Spout reads one line at a time from the local file used in the *WC* topology. Each Communication Bolt doubles the received words and passes them to the next bolt. This is a typical communication intensive workload.

Evaluation Metrics. We use two metrics to systematically evaluate the result of our experiments.

– *Tuple Processing Time (TPT)*: It presents the average elapsed time for a tuple emitted from spout till its completion. We leveraged the timing mechanism in Storm to track each tuple's processing time. We calculated the average TPT every 30 seconds for performance evaluation.



a. Tuple Processing Time on LP Topology.



b. Tuple Processing Time on TSSA Topology.

Fig. 5: Tuple Processing Time on Two Workloads.

– *Inter-Node Bandwidth (INB)*: It indicates the inter-node bandwidth of the topology during the execution. INB can well characterize the effect of our graph-partitioning based schedulers because they are designed to reduce the cost of inter-node communication among physical nodes.

5.2 Effect of Bootstrap Scheduler

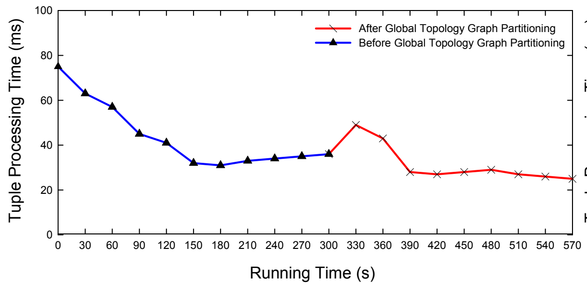
Results and Analysis. We first evaluate the performance of Bootstrap Scheduler. We used 40 worker nodes, with 10 task slots on each node. Here we present the results on LP topology and TSSA topology as representatives. The experimental results on the other workloads yield similar improvements.

1. *Log Processing (LP)*: For this topology, we used 10 tasks for the spout, 50 tasks for the log rule bolt, 30 tasks for the index bolt, and 30 tasks for the counter bolt. Figure 5a shows the TPT of the default scheduler and Bootstrap Scheduler on this topology. The TPT of Bootstrap Scheduler is reduced by 37% on average compared to the default scheduler during the first 10 minutes. This is because Bootstrap Scheduler reduces inter-node communication cost by considering the structure of the committed topology while the default scheduler evenly distributes the task to the physical nodes.
2. *Twitter Stream Sentiment Analysis (TSSA)*: For this topology, we used 8 tasks for tuple input, and 20 tasks for each of the other bolts. As shown in Figure 5b, the TPT of Bootstrap Scheduler is shorter than that of the default schduler by 39% on average after the system reaches a stable state. The TPT of TSSA is higher than that of LP because the graph structure of TSSA is more complicated, which incurs more inter-node communication, as shown in Figure 4d and 4e.

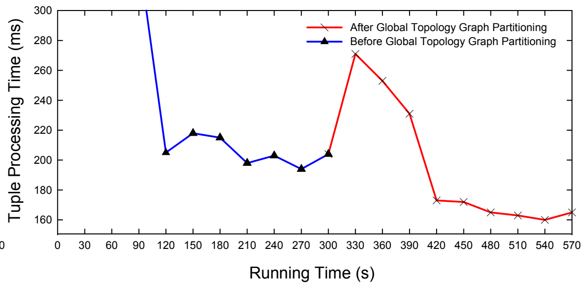
Summary. When a new topology is committed to the system, Bootstrap Scheduler can analyze and assign its tasks to the proper physical nodes. This assignment takes inter-node communication cost into account, thus, Bootstrap Scheduler outperforms the default Storm scheduler. The above experiments prove that, with the help of graph partitioning, Bootstrap Scheduler can significantly reduce the average tuple processing time on various workloads.

5.3 Effect of Rebalance Scheduler

We next present the performance of Rebalance Scheduler. Rebalance Scheduler uses two techniques for task reallocation, i.e., *global-topology-graph-partitioning* which repartitions all the topologies on the cluster and *dynamic-task-reassignment* which moves tasks from overloaded nodes to idle ones automatically.
Effect of Global-Topology-Graph-Partitioning. In this experiment, we initially committed the WC topology to the cluster, then we committed the TT topology after 30 seconds and the TWTT topology after 60

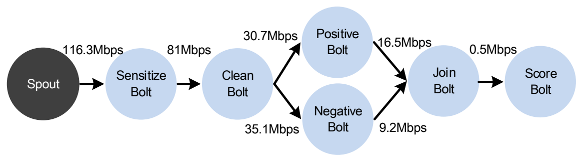


a. Word Count Topology.

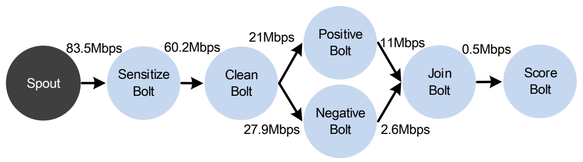


b. Twitter Trending Topics Topology.

Fig. 6: Evaluation of Global-Topology-Graph-Partitioning.



a. INB before Dynamic-Task-Reassignment.



b. INB after Dynamic-Task-Reassignment.

Fig. 7: INB between Components of Twitter Stream Sentiment Analysis Topology.

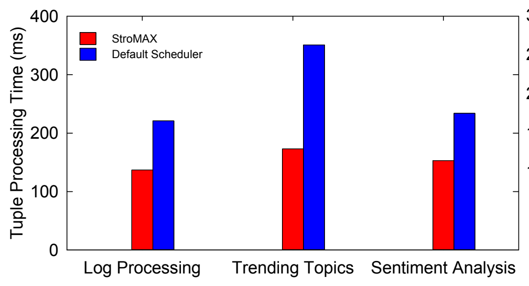
seconds. These topologies were allocated by Bootstrap Scheduler when committed. As same as the setting in Bootstrap Scheduler, we used 40 worker nodes, with 10 task slots on each node. In addition, we used 5 tasks for each spout and 20 tasks for each bolt. We started global-topology-graph-partitioning to reassign the tasks of these topologies at 300 seconds.

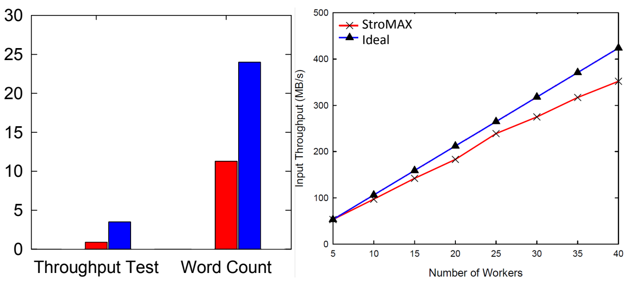
Figure 6 summarizes the TPT of these three topologies in 10 minutes. Due to the space constraint, we present the results of WC and TWTT topologies, and the result of TT topology is similar. As we can see, once global-topology-graph-partitioning was triggered, it calculated a new assignment for all the topologies in the cluster, which briefly increased the tuple processing time of these topologies. Afterwards, the tuple processing time dropped sharply to a normal value and outperformed the previous result, with a 10.9%-26.5% reduction of TPT on these workloads. This is because global-topology-graph-partitioning utilizes the collected runtime statistics to estimate the computation cost and communication cost of each task during the execution, and then optimizes the task assignment for all the topologies. In contrast, Bootstrap Scheduler allocates the newly committed topology via the graph partitioning result of a single topology. Therefore, when we commit three different topologies to the system, global-topology-graph-partitioning improves the tuple processing time based on the runtime result of Bootstrap Scheduler.

Effect of Dynamic-Task-Reassignment. We proceed to evaluate the effect of the dynamic-task-reassignment. We used 40 worker nodes, with 10 task slots on each node. We committed all the 6 topologies to the system with Bootstrap Scheduler. We used 3 tasks for each spout and 10 tasks for each bolt. We reassigned the tasks by global-topology-graph-partitioning after the system reached a stable state. Then we used the dynamic-task-reassignment to monitor and reallocate the tasks when they were skew enough. We recorded the inter-node bandwidth (INB) between components of the TSSA topology before and after executing the dynamic-task-reassignment.

As shown in Figure 7, most of the inter-node bandwidths were reduced after the dynamic-task-reassignment. Specially, the communication cost between the Spout and Sensitive Bolt decreased from 116.3Mbps to 83.5Mbps, and the cost between the Negative Bolt and Join Bolt decreased from 9.2Mbps to 2.6Mbps.

Parameter Tuning of Dynamic-Task-Reassignment. As we mentioned in Section 3.3, we use a parameter ϑ to judge whether a node is skew enough and needed to reassign its tasks to other nodes. We have conducted





a. Overall Performance. b. Scalability.

Fig. 8: Overall Performance and Scalability of StroMAX.

a series of experiments to investigate the selection of ϑ. Due to the space limit, we do not show the details of parameter tunning.

We find that when ϑ is chosen between 10% and 15%, the dynamic-task-reassignment achieves the best performance for most of the workloads. The reason is that, when ϑ is small ($\vartheta < 10\%$), there are too many tasks reallocated during the execution. When ϑ becomes larger ($\vartheta > 15\%$), the reassignment is hard to be triggered. Thus we use $\vartheta = 12\%$ for the dynamic-task-reassignment in our experiments.

Summary. The above experiments indicate that Rebalance Scheduler can efficiently reduce the communication cost among the components of the topologies by reassigning the tasks based on the global-topology-graph. It leverages the statistics of log to monitor the running status of each node and dynamically reallocates bottleneck tasks. Therefore, Rebalance Scheduler can significantly improve the performance of real-time processing systems.

5.4 Overall Performance and Scalability of StroMAX

Overall Performance. In this part, we investigate the overall performance of StroMAX. In this experiment, we executed each of the five topologies on the cluster separately. The process was as follows: we added the topologies onto the cluster one by one. When a topology finished, we restarted the cluster and deployed a new topology onto the cluster with the help of Bootstrap Scheduler and Rebalance Scheduler.

Figure 8a illustrates the TPT of StroMAX and the default Storm scheduler on five topologies. As we can observe, compared to the default scheduler, the TPT of most workloads significantly decreases. For instance, the TPT on TT topology decreases by 87.3%. These results confirm that StroMAX can significantly reduce the tuple processing time and inter-node communication cost with the graph partitioning. Besides, the results also reveal the generality of the proposed approach that it can be applied to various workloads.

Scalability Study. Scalability is an important issue for real-time stream processing systems. We further evaluate the scalability of StroMAX by increasing the number of worker nodes. We increase the number of worker nodes from 5 to 40, and present the input throughput of the TSSA topology in Figure 8b. We use both Bootstrap Scheduler and Rebalance Scheduler to schedule the topology. We use an ideal curve to represent the ideal execution time which assumes that the performance is linear to the number of the workers. As expected, as the number of workers increases, the throughput performance of StroMAX is close to the ideal case. This result confirms that StroMAX has a graceful scalability.

6 Related Work

The graph-partitioning based scheduling problem in real-time stream processing systems discussed in this paper is related to several fields. We briefly review the most relevant works.

Real-time Stream Processing Systems. System S [2] is a stream processing system developed by IBM. A query in System S is modeled as an event processing network which consists of a set of event processing agents. S4 [14] is another stream processing system, developed by Yahoo, where queries are designed as graphs of processing elements. Recently, Storm [21,10,11], an open-source, distributed, reliable, and fault-tolerant processing system, was proposed by Twitter for real-time stream processing. Some works [5] tried to bridge the gap between stream workload and MapReduce abstraction by proposing a stream version of the MapReduce approach. In these systems, events flow among the map and reduce stages without incurring. Wang [23] studied the problem of efficient load distribution in D-DSMS to minimize end-to-end latency. Besides, Xing [24] studied operators moving to dynamically change loads in high-performance computing clusters such as blade computers.

Graph Partitioning. Graph partitioning is a optimization problem which has been studied for decades [25,18]. The widely used k-balanced graph partitioning aims to minimize the number of edge-cut between partitions while balancing the number of vertices. Though the k-balanced graph partitioning problem is an NP-Complete problem [8], several solutions have been proposed to tackle this challenge. Andreev et al. [3] presented an approximation algorithm which guarantees polynomial running time with an approximation ratio of $\mathcal{O}(logn)$. Another solution was proposed by Even et al. [7] who gave an LP method based on spreading metrics which also gets an $\mathcal{O}(logn)$ approximation. Besides approximated solution, Karypis et al. [13] proposed a parallel multi-level graph partitioning algorithm to minimize the bisection on each level. There are some heuristic implementations like METIS [12], parallel version of METIS [16], and Chaco [9] which are widely used in many existing systems. Although these heuristics cannot provide a precise performance guarantee, they are effective.

The aforementioned methods are offline and generally require long processing time. Recently, Stanton and Kliot [20] proposed a series of online streaming partitioning method using heuristics. Fennel [22] extended the Stanton's work by proposing a streaming partitioning framework which combines some other heuristic methods. However, these methods are designed for generally graph partitioning and lack the consideration of the characteristics of DAG.

Beyond these static graph partitioning technologies, Nicosia [15] theoretically studied how to adapt to the graph structure changing without the overhead of reloading or repartitioning the graph. Some of the recent works [26,6] can cope with the changes in graph structure. However, the cost of these approaches to handle the changes is quite high. Shang et al. [17] investigated several graph algorithms and proposed simple, yet effective, policies that can achieve dynamic workload balance, while this approach uses hashing partitioning as the initial input.

7 Conclusion

In this paper, we systematically investigated the performance issues of real-time stream processing systems. We designed a novel system, StroMAX, to allocate the topology based on two graph partitioning based schedulers. The first scheduler, named Bootstrap Scheduler, analyzes the topological graph and partitions the topology when it is committed to the system. The second scheduler, named Rebalance Scheduler, goes one step further by monitoring the effectiveness of all the topologies and the load of cluster during runtime. Rebalance Scheduler then rebalances the topologies for a performance improvement when necessary. The experimental results confirmed the improvements of our proposed approaches.

Acknowledgment

This research is supported by the National Natural Science Foundation of China under Grant No. 61572039, 973 program under No. 2014CB340405, Shenzhen Government Research Project JCYJ20151014093505032, and Tecent Research Grant (PKU).

References

1. Flume, http://flume.apache.org/
2. Amini, L., Andrade, H., et al.: Spc: A distributed, scalable platform for data mining. In: DM-SSP. pp. 27–37 (2006)
3. Andreev, K., Racke, H.: Balanced graph partitioning. Theory of Computing Systems 39(6), 929–939 (2006)
4. Billionnet, A., Costa, M.C., Sutter, A.: An efficient algorithm for a task allocation problem. JACM 39(3), 502–518 (1992)
5. Brito, A., Martin, A., Knauth, T., Creutz, S., Becker, D., Weigert, S., Fetzer, C.: Scalable and low-latency data processing with stream mapreduce. In: CloudCom. pp. 48–58 (2011)
6. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: taking the pulse of a fast-changing and connected world. In: EuroSys. pp. 85–98 (2012)
7. Even, G., Naor, J., Rao, S., Schieber, B.: Fast approximate graph partitioning algorithms. In: SODA. pp. 639–648 (1997)
8. Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some simplified np-complete problems. In: STOC. pp. 47–63 (1974)
9. Hendrickson, B., Leland, R.W.: A multi-level algorithm for partitioning graphs. SC 95, 28 (1995)
10. Huang, Y., Cui, B., Jiang, J., Hong, K., Zhang, W., Xie, Y.: Real-time video recommendation exploration. In: SIGMOD. pp. 35–46 (2016)
11. Huang, Y., Cui, B., Zhang, W., Jiang, J., Xu, Y.: Tencentrec: Real-time stream recommendation in practice. In: SIGMOD. pp. 227–238 (2015)
12. Karypis, G., Kumar, V.: Multilevel graph partitioning schemes. In: ICPP. pp. 113–122 (1995)
13. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. In: SC (1996)
14. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: ICDM. pp. 170–177 (2010)
15. Nicosia, V., Tang, J., Musolesi, M., Russo, G., Mascolo, C., Latora, V.: Components in time-varying graphs. Chaos: An Interdisciplinary Journal of Nonlinear Science 22(2), 023101 (2012)
16. Schloegel, K., Karypis, G., Kumar, V.: Parallel static and dynamic multi-constraint graph partitioning. Concurrency and Computation: Practice and Experience 14(3), 219–240 (2002)
17. Shang, Z., Yu, J.X.: Catch the wind: Graph workload balancing on cloud. In: ICDE. pp. 553–564 (2013)
18. Shao, Y., Cui, B., Ma, L.: Page: a partition aware engine for parallel graph computation. TKDE 27(2), 518–530 (2015)
19. Shi, X., Cui, B., Shao, Y., Tong, Y.: Tornado: A system for real-time iterative analysis over evolving data. In: SIGMOD. pp. 417–430 (2016)
20. Stanton, I., Kliot, G.: Streaming graph partitioning for large distributed graphs. In: KDD. pp. 1222–1230 (2012)
21. Toshniwal, A., Taneja, S., et al.: Storm@ twitter. In: SIGMOD. pp. 147–156 (2014)
22. Tsourakakis, C.E., Gkantsidis, C., Radunović, B., Vojnović, M.: Fennel: Streaming graph partitioning for massive scale graphs. Tech. rep., Microsoft (2012)
23. Wang, W., Sharaf, M.A., Guo, S., Özsu, M.T.: Potential-driven load distribution for distributed data stream processing. In: SSPS. pp. 13–22 (2008)
24. Xing, Y., Zdonik, S., Hwang, J.H.: Dynamic load distribution in the borealis stream processor. In: ICDE. pp. 791–802 (2005)
25. Xu, N., Cui, B., Chen, L., Huang, Z., Shao, Y.: Heterogeneous environment aware streaming graph partitioning. TKDE 27(6), 1560–1572 (2015)
26. Yang, S., Yan, X., Zong, B., Khan, A.: Towards effective partition management for large graphs. In: SIGMOD. pp. 517–528 (2012)