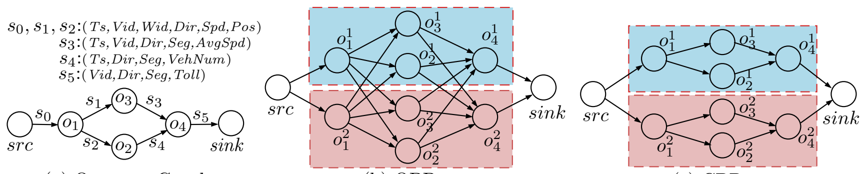
CBP: A New Parallelization Paradigm for Massively Distributed Stream Processing

Abstract. Resource efficiency is essential for distributed stream processing engines (DSPEs), in which a streaming application is modeled as an operator graph where each operator is parallelized into a number of instances to meet the low-latency and high-throughput requirements. The major objectives of optimizing resource efficiency in DSPEs include minimizing the communication cost by collocating the tasks that transfer a lot of data between each other, and by dynamically configuring the systems according to the load variations at runtime. In the current literature, most proposals handle these two optimizations separately, and a shallow integration of these techniques, such as performing the two optimizations one after another, would result in a suboptimal solution. In this paper, we present component-based parallelization (CBP), a new paradigm for optimizing the resource efficiency of DSPEs, which provides a framework for a deeper integration of the two optimizations. In the CBP paradigm, the operators are encapsulated into a set of non-overlapping components, in which operators are parallelized consistently, i.e., using the same partitioning key, and hence the intra-component communication is eliminated. According to the changes of workload, each component can be adaptively partitioned into multiple instances, each of which is deployed on a computing node. To optimize a CBP plan, We build a cost model to capture both the communication cost and adaptation cost of a CBP plan, and then propose several optimization algorithms. We implement the CBP scheme and the optimization algorithms on top of Apache Storm, and verify its efficiency by an extensive experiment study.

1 Introduction

Real-time big data analysis requires processing of *continuous queries* (CQ) over fast streaming data with low latency. Usually, distributed stream processing engines (DSPEs) [22,18,1] organize CQs as an operator graph as shown in Fig. 1(a). To handle the deluge of data, one can resort to massive parallelization that each operator is cloned with a number of instances and its inputs is partitioned into disjoint substreams. For the sake of resource efficiency, there are in general two critical optimizations to be considered:

1. **Runtime resource reconfiguration.** Load variations caused by the changes of data distribution and input rate are ubiquitous in the streaming context [22,24,25]. It is essential to provide adaptive data partitioning to achieve load balancing and to scale the number of parallel instances of each operator to avoid over-provisioning or under-provisioning.
2. **Communication cost minimization.** A large amount of data has to be continuously transmitted among the neighboring operators. Data transfer not only consumes bandwidth but also incurs significant computation overhead, including serializing and de-serializing the transmitted data. Optimizing the allocation of operator instances can to minimize cross-node communication can significantly reduce the resource consumption in a DSPE.



(a) Operator Graph (b) OBP (c) CBP

Fig. 1: **Paradigms for parallelizing operator graph**. This is a query that calculates the tolls of vehicles based on the source stream s_0 containing data of vehicles' speeds and positions. It consists of four operators: (1) a stateless operator o_1 that filters and partitions input data to the next operators; (2) two operators o_2 and o_3 calculate the average speed *AvgSpd* and the traffic volume respectively; and (3) o_4 calculates the toll of each vehicle, which is a function of *AvgSpd* and *SegNum*. The format of streams are specified in figure (a) and each operator o_i is designated with key k_i for partitioning the streams, where k_1=$\{$*Ts, Vid, Spd, Dir, Seg, Pos*$\}$, k_2=$\{$*Vid, Dir, Seg*$\}$, k_3=$\{$*Dir, Seg*$\}$, and k_4=$\{$*Dir, Seg*$\}$. We use two ways to parallelize the query: (1) in figure (b), the input streams of the operators are partitioned with different keys; and (2) in figure (c), the input streams of the four operators are partitioned consistently with the same key $\{$*Dir, Seg*$\}$.

In existing solutions, the two problems have been addressed separately. For example, M.A. Shah et al. [22] studied how to dynamically partition the input data at runtime to balance the workload across the parallel instances of an operator, while Y. Ahmad et al. [4] and P. Pietzuch et al. [19] investigated the operator placement to minimize the bandwidth usage by implicitly assumed assumption that operators do not need to be parallelized.

One can simply combine these methods to provide a complete solution. For example, we can first determine the parallelism for each operator [1], and transform the operator graph into a graph of operator instances. Thereafter, we can optimize the deployment by applying an operator placement algorithm, such as [4,19]. Suppose we have two nodes in the cluster, Fig. 1(b) shows a possible parallelization and task allocation plan for the operator graph in Fig. 1(a). Dynamic reconfigurations, such as re-scaling and load balancing, can be performed on each operator independently. However, such a shallow integration would provide suboptimal performance. As shown in Fig. 1(b), if the 4 operators are not parallelized consistently, e.g., partitioning the input on the same key, then each operator instance may have to transfer data to all its downstream instances. This limits the opportunity to minimize communication cost by collocating the instances that communicate with each other.

On the contrary, if we can parallelize the operators consistently using a common partitioning key, then we could have a plan as shown in Fig. 1(c), which minimizes cross-node communication. Although this idea may sound simple, it is nevertheless challenging to implement in a DSPE supporting runtime reconfiguration. First of all, dynamic data repartitioning makes it difficult or even impossible to achieve consistent parallelization of multiple operators given that the operators could be reconfigured at runtime independently. Secondly, dynamic scaling and data repartitioning involve a lot of state movements [22,23]. The overhead of moving the states around operator instances in order to maintain the consistency of data partitioning may offset the benefits of collocating their communicating instances. Therefore we need a new parallelization framework that can optimize the parallelization of operators such that the total cost is minimized, including the communication and reconfiguration cost.

To address the challenges, we present **component-based parallelization (CBP)**—a new operator parallelization paradigm that considers both dynamic reconfiguration and resource optimization. In CBP, an operator graph is first decomposed into non-overlapping *components*, each being a connected subgraph. The operators in a component should have partitioning keys "compatible" with each other, i.e., sharing common attributes, and thus they can be parallelized using the same key. Each component acts as a singleton that is parallelized into a set of instances and the parallelism can be adapted at runtime in accordance with the load variations. This strategy simplifies the optimization of parallel stream processing and localizes the side-effect of reconfiguration within each component. In general, in the CBP paradigm, the more operators are grouped into a component, the less communication cost there would be, with a probable increase of the component's reconfiguration cost. This is because every time we have to re-scale or re-balance one operator within a component, we have to trigger repartitioning of all the operators within the component. Therefore, a good trade-off should be found to minimize the total cost of a CBP plan.

We propose a cost-based optimizer to compute an optimized CBP plan for a given query graph. We develop a novel cost model that integrates the reconfiguration overhead into the optimization. We formally define the optimization problem as a MINIMUM-COST-COMPONENT-BASED-PARALLELIZATION problem (MCCBP). We prove that MCCBP is NP-hard, and then present two heuristic algorithms to solve it. All the techniques have been implemented on top of Apache Storm [1]. We compare our solutions with the operator placement algorithm by using both synthetic workload and an extension of the Linear Road Benchmark [6]. The experiments show that our methods can save the network communication by up to 40%. Furthermore, our solutions can reduce the average end-to-end data latency by about 10% to 30%.

2 Background

2.1 Parallel Stream Processing

Continuous queries(CQs) [17] over streaming data are usually organized as an operator graph [13,18,1] in a *distributed stream processing engine* (DSPE). DSPEs like Flux [22] and StreamCloud [14] exploit data parallelism [11] to cope with the deluge of data, in which an operator is cloned into a set of independent instances each working on a partition of the input data. The number of partitioning can be determined according the input rates to achieve high throughput.

Operators can be categorized as stateless and stateful. For a stateless operator, the input tuples can be processed independently by any instance of it. While the stateful operators, such as **join** and **group-by aggregate**, are "context-sensitive", so tuples with the same keys should be processed by the same instance to guarantee correctness. *Stream grouping* specifies the way how a stream of tuples is grouped and dispatched to the consumer operator instances. We consider two primitives: (1) *shuffle grouping*, where the input tuples are randomly routed to the operator instances; (2) *key grouping*, in which tuples are partitioned into a number of substreams based on a specified set of keys. Shuffle grouping is often the optimal choice for stateless operators since the load can be easily balanced, while key grouping is necessary for stateful operators.

Challenges of load variation. Usually, one can easily observe two kinds of variations over streaming data: (1) the fluctuation of input rates [22,24], and (2) change of data value distributions [22,24,25]. If an SPE does not react to the variations, applications can run into problems:

- **Unmatched provision:** the over-provisioning or under-provisioning caused by the fluctuation of the input rates can result in low system utilization, high operational cost (e.g., using pay-as-you-go cloud services), and system failures.
- **Load imbalance:** the load distribution is skewed due to the change of data value distribution. For example some stream grouping keys become more popular than the others so that some operator instances are over-loaded while the others are under-loaded. Load imbalance can harm the processing latency and system throughput if the skewness is not resolved soon.

To handle the above problems, we resort to adaptation techniques including *dynamic scaling* [18] and *load balancing* [25]. CQs use the concept of *sliding windows* of tuples over a stream to specify the operational context of an operator. For instance, to perform a windowed join, we need to buffer the tuples within the current window(s) as the context of the join operation on the newly incoming tuples. This kind of context is called as *processing state* [8]. While processing an adaptation, the substreams should be reassigned around operator instances, and the processing states needs to be reallocated accordingly. This process is called *state movement*. Note that both scaling and load balancing involve state movements, which consume both significant CPU and network bandwidth and thus cannot be ignored [22,23].

2.2 System Model

Data model. A data stream s is an unbounded and append-only sequence of tuples $(\dots, t_{i-1}, t_i, t_{i+1}, \dots)$. Each tuple $t = (\tau, \alpha)$ has a timestamp $\tau \in \mathbb{T}$ and a set of attributes $\alpha = (a_1, \dots, a_k)$. We assume that the attribute set α of every stream conforms to a relational schema. For simplicity, τ is assumed to be unique. In practice, if τ is not unique, existing systems usually use a unique sequence number to identify each tuple.

Operator model. A CQ is composed of a number of *operators*, each implementing a certain computation logic, such as *join*, *aggregate*, *filter*, or *user-defined functions*. An operator o is a 6-tuple, $(\mathtt{IN}_o, \mathtt{OUT}_o, \mathtt{K}_o, \mathtt{F}_o, \mathtt{W}_o, \mathtt{PS}_o)$, where $\mathtt{IN}_o$ and $\mathtt{OUT}_o$ are the input and output streams respectively. $\mathtt{K}_o$ is the *key*, a subset of attributes of the input streams $\mathtt{IN}_o$, which used for partitioning $\mathtt{IN}_o$. $\mathtt{F}_o$ defines the processing logic, where its operating context, i.e. the *processing state* $\mathtt{PS}_o$, is specified by the sliding window $\mathtt{W}_o$. For stateless operators like *map* and *filter*, $\mathtt{PS} = \emptyset$.

We organize CQs as an operator graph $\mathtt{G} = (\mathtt{O}, \mathtt{S})$, which is a directed acyclic graph of the operator set $\mathtt{O}$ and the stream set $\mathtt{S}$. A stream $s \in \mathtt{S}$ is represented as a directed arc (u_s, d_s), $u_s, d_s \in O$, where u_s and d_s are its producer and consumer respectively. Two special operators, *Src* and *Sink*, are responsible for spouting source streams and collecting the final results respectively. An operator graph is also referred to as a *topology* and these two terms are interchangeable throughout this paper.

Physical execution. The operator graph is executed on a cluster of identical nodes. The *execution graph* is a physical realization of the query in which each operator o is parallelized into multiple instances $\mathcal{I} = \{o^1, \ldots, o^\pi\}$, where $\pi \in \mathbb{N}^+$ is the *parallelism*. For an input stream s of o, each tuple is a key-value pair <k,v>, where v is the tuple and $\text{k} = t.\text{K}_o$. A partitioning function split the domain of $\texttt{K}_o$ into p groups, where $p \gg \pi$. Then, the tuples of s, according their key values, form a number of substreams $\mathcal{S} = \{s^1, \ldots, s^p\}$. An assignment $\mathcal{F} : \mathcal{S} \to \mathcal{I}$ allocate the processing of each substream to a unique operator instance. The degree of parallelism π and the assignment $\mathcal{F}$ are adapted at runtime to handle load variations.

3 Component-Based Parallelization

3.1 CBP Abstraction

In essence, CBP decomposes an operator graph into a set of non-overlapping *components*, which act as the parallelization unit. In particular, CBP relies on two essential properties: *compatibility* and *connectivity*.

Compatibility concerns if some operators can be parallelized consistently. A set of operators $\{o_1, \ldots, o_k\}$ is compatible iff the intersection of their keys is not empty, i.e., $\texttt{K}_{o_1} \cap \cdots \cap \texttt{K}_{o_k} \neq \emptyset$. Note that the compatibility property is not *transitive*. For example, suppose we have three operators o_1, o_2, and o_3 with keys $\texttt{K}_1 = \{a_1, a_2\}$, $\texttt{K}_2 = \{a_2, a_3\}$, and $\texttt{K}_3 = \{a_1, a_3\}$ respectively. Even though any pair of them are compatible, they as a whole are incompatible because $\texttt{K}_1 \cap \texttt{K}_2 \cap \texttt{K}_3 = \emptyset$.

The rationale of assembling the topology into components is to reduce the communication cost. One can benefit from placing compatible operators into a node only if they are connected by streams.

Formally, we can define a component as follow.

Definition 1 (Component). *A* ***component*** $\texttt{C} = (\texttt{O}_\texttt{C}, \texttt{S}_\texttt{C})$ *is an induced subgraph of the operator graph* $\texttt{G} = (\texttt{O}, \texttt{S})$*, where* $\texttt{C}$ *is connected and the operators in* $\texttt{O}_\texttt{C}$ *are compatible.*

Let $\texttt{IN}(\texttt{C})$ be the set of all input streams of the operators in component $\texttt{C}$, then $\texttt{IN}(\texttt{C}) = \cup_{o \in \texttt{O}_\texttt{C}} \texttt{IN}_o$. Assuming $\texttt{O}_\texttt{C} = \{o_1, \ldots, o_{|\texttt{C}|}\}$. The streams of $\texttt{IN}(\texttt{C})$ can be grouped by a partition function over the key $\texttt{K} = \texttt{K}_{o_1} \cap \cdots \cap \texttt{K}_{o_{|\texttt{C}|}}$, which is the intersection of the keys of all the operators in $\texttt{C}$. Since $\texttt{K} \neq \emptyset$, all the streams of $\texttt{IN}(\texttt{C})$ can be partitioned uniformly into p substreams with $\mathcal{H}(\texttt{K})$. For the convenience of discussion, we regard the streams in $\texttt{IN}(\texttt{C})$ as a *composite stream cs*, which is partitioned into a set of substreams $\mathcal{CS} = \{cs^1, \ldots, cs^p\}$.

Each component $\texttt{C}$ is parallelized into a set of instances $\mathcal{CI} = \{ci^1, \ldots, ci^\pi\}$, where π is the *parallelism* of $\texttt{C}$ and each instance has a clone of the computation logic of each operator in $\texttt{C}$. The parallel processing of the composite stream $\mathcal{CS}$ is specified by an assignment $\mathcal{F}_\texttt{C} : \mathcal{CS} \to \mathcal{CI}$, which is adapted at runtime to handle load variations.

4 MCCBP

4.1 Metrics

The cost of a CBP plan can be put into three parts: (1) *Processing cost* $\mathcal{PC}$, which is the CPU usage of the computation, (2) *Communication cost* $\mathcal{CC}$, which is the CPU and network usages of data transmission, and (3) *Adaptation cost* $\mathcal{AC}$, which is the CPU and network usages of carrying out adaptations.

In particular, we assume that $\mathcal{PC}$ keeps the same regardless of the physical execution, and thus it can be disregarded in our cost model. In addition, we categorize data communication into *inter-component communication* and *intra-component communication*. The first one involves three sequential steps: (1) data serialization, (2) network propagation, and (3) de-serialization. Steps (1) and (3) consume CPU cycles and step (2) occupies network bandwidth. In contrast, the intra-component communication is realized via local memory access, whose overhead is negligible. Therefore, we only take the overhead of inter-component communication into account.

Statistics measurements. The cost calculation relies on the statistics of execution of the operator graph. In our implementation, the statistics are measured periodically over a sequence of time intervals of length $\mathit{\Delta}$, which are called as *statistics windows*. Suppose the historical data spans m statistics windows that start at the time instance $\tau = 0$, then the timespan of historical data is $[0, m\mathit{\Delta}]$. The following discussions are confined within the timespan $[0, m\mathit{\Delta}]$.

For the input stream $s \in \mathtt{S}$ of a component that is split into p partitions, the statistics are represented as a sequence of histograms $\boldsymbol{Y}(s) = (Y_1, \dots, Y_m)$, where the histogram $Y_r = (y_{1,r}, \dots, y_{p,r})^T$, $r = 1 \dots m$, is a vector recording the data rate of the p partitions over the r-th statistics window. In other words, the data distribution of s at the r-th window can be approximated with Y_r. With $\boldsymbol{Y}$, we can derive other statistics as needed. For instance, denote $s = (o_i, o_j)$, then the load l_{ij} of s during $[0, m\mathit{\Delta}]$ is $l_{ij} = \sum_{r=1}^{m} \sum_{k=1}^{p} y_{kr}$.

The adaptation cost is closely related to the adaptation frequency f. For simplicity, we suppose that SPE performs an adaptation for each window with length $\mathit{\Delta} = {}^1/_f$. Let ψ_i^r be the number of state movements in the r-th adaptation of component $\mathtt{C}_i$, then $\mathcal{AC} = \sum_{i=1}^{|\mathcal{C}|} \psi_i$, where $\psi_i = \sum_{r=1}^{m} \psi_i^r$ is the adaptation cost of $\mathtt{C}_i$.

4.2 Problem Formulation

Consider an operator graph that is grouped into a set of disjoint components $\mathcal{C} = \{\mathtt{C}_1, \mathtt{C}_2, \dots\}$, it is called a CBP plan if $\cup_{i=1}^{|\mathcal{C}|} \mathtt{O}_{\mathtt{C}_i} = \mathtt{O}$ and $\mathtt{O}_{\mathtt{C}_i} \cap \mathtt{O}_{\mathtt{C}_j} = \emptyset$ for any two components of $\mathcal{C}$. Let $\mathtt{X}$ be the streams interconnecting components in $\mathcal{C}$. Let $w(\mathtt{C}_i)$ be the adaptation cost of $\mathtt{C}_i$ and $c(s)$ be the communication cost incurred by stream s. Since $\mathcal{PC}$ is independent on the CBP plan, the cost of a CBP plan $\mathcal{C}$, denoted as $cost(\mathcal{C})$, is measured by the sum of the communication cost $\mathcal{CC}$ and adaptation cost $\mathcal{AC}$. That is,

$$cost(\mathcal{C}) = \mathcal{CC} + \mathcal{AC} = \sum_{s \in \mathtt{X}} c(s) + \sum_{\mathtt{C}_i \in \mathcal{C}} w(\mathtt{C}_i) \qquad (1)$$

In addition, we introduce a constraint on the adaptation cost, i.e, $w(\mathtt{C}_i) \le \beta$, to prevent any component from being the bottleneck. Consequently, the objective of optimizing a CBP plan is to minimize $cost(\mathcal{C})$. We call this the MINIMUM COST COMPONENT-BASED PARALLELIZATION (MCCBP) problem. MCCBP is a variant of graph partitioning problem with additional constraints on connectivity and compatibility. Formally, it is stated as follow.

Definition 2 (MCCBP). *Given an operator graph* $\mathtt{G} = (\mathtt{O}, \mathtt{S})$ *and a positive constant* β*, the MCCBP problem is to find a CBP plan, which is a partition of* $\mathtt{G}$ *into a set of disjoint components* $\mathcal{C} = \{\mathtt{C}_1, \mathtt{C}_2 \dots\}$*, to achieve the following objective:*

$$\begin{aligned} \textit{minimize} \quad & cost(\mathcal{C}) \\ \textit{subject to} \quad & \cup_{i=1}^{|\mathcal{C}|} \mathtt{O}_{\mathtt{C}_i} = \mathtt{O} \\ & w(\mathtt{C}_i) \leq \beta \end{aligned}$$

MCCBP can be proved to be NP-hard by simplifying it to a *Minimum-Capacity-Graph-Partitioning* (MCGP) problem, which has been shown to be NP-hard.

5 Computing CBP Plans

In this section, we present two approximate algorithms to tackle MCCBP.

5.1 Greedy Algorithm

A straightforward idea is to obtain an initial CBP plan $\mathcal{C}_0$ in advance, and then make improvement incrementally. The algorithm, as shown in Alg. 1, begins with the initial plan $\mathcal{C}_0$ (Line 2) and makes improvement step by step (Line 9–21). The initial plan $\mathcal{C}_0$ is generated by a depth-first search (DFS) of the operator graph. The traversal is tracked by an operator stack OS. In each iteration, we peek an operator from OS. Let o be the current operator being visited and $\mathtt{C}(o)$ be the component containing o. Then o will be popped out from OS if it has no unvisited child or is a leaf node. Otherwise we choose an unvisited child v of o and then check the compatibility between v and $\mathtt{C}(o)$. If they are compatible, v will be added into component $\mathtt{C}(o)$; Otherwise, a new component $\mathtt{C}_i$ containing operator v is created.

The essence of Alg. 1 is to reduce the cost by moving operators around components. Let *move(*$\mathtt{C}_i$,$\mathtt{C}_j$,o_k*)* be the *potential movement* that attempts to move o_k from $\mathtt{C}_i$ to $\mathtt{C}_j$. It is *admissible* if $o_k \in \mathtt{C}_i$ and the new operator set $\mathtt{C}_j \cup \{o_k\}$ can form a component. Given a CBP plan $\mathcal{C}$, the execution of the potential movement *move(*$\mathtt{C}_1$,$\mathtt{C}_2$,o_k*)* gives rise to a new plan $\mathcal{C}'$ if it is admissible. The admissibility of it is checked in Line 8.

The movement results in the following change of costs: (1) the change of communication cost between $\mathtt{C}_1$ and $\mathtt{C}_2$, and (2) the change of adaptation costs of $\mathtt{C}_1$ and $\mathtt{C}_2$. Hence the profit $\delta_{12}(o_k)$ of *move(*$\mathtt{C}_1$,$\mathtt{C}_2$,o_k*)* consists of two parts: the changes of the communication cost and adaptation cost, denoted as $\delta^1_{12}(o_k)$ and $\delta^2_{12}(o_k)$ respectively. Let $\varphi_1(o_k)$ be the data rate between o_k and $\mathtt{C}_1$. Then, $\varphi_1(o_k) = \sum_{\substack{(o_k,o_t)\in\mathtt{S}\vee(o_t,o_k)\in\mathtt{S} \\ o_t\in\mathtt{C}_1}} l_{kt}$. $\varphi_1(o_k)$ does not contribute to $\mathcal{CC}$ if $o_k \in \mathtt{C}_1$, otherwise it does. After the movement, $\varphi_i(o_k)$ contributes to $\mathcal{CC}$, but $\varphi_j(o_k)$ does not contribute to $\mathcal{CC}$. Therefore, the gain on communication cost is $\delta^1_{12}(o_k) = \varphi_2(o_k) - \varphi_1(o_k)$. Let $\psi(o_k)$ be the new adaptation cost of a component after the movement. Then, $\delta^2_{12}(o_k) = (\psi_1 + \psi_2) - (\psi_1(o_k) + \psi_2(o_k))$.

Summing all together, we get the overall profit of the movement, $\delta_{12}(o_k) = \delta^1_{12}(o_k) + \delta^2_{12}(o_k)$. In each run, we choose an admissible movement with the maximum positive profit to execute. Suppose that $\delta_{12}(o_k)$ is the best movement in the current run,

Algorithm 1: Greedy Algorithm

```
   Input: Operator graph $\mathtt{G} = (\mathtt{O}, \mathtt{S})$, load statistics $\{\mathbf{Y}(s_1), \mathbf{Y}(s_2), \dots\}$, state statistics
          $\{\mathbf{Z}(o_1), \mathbf{Z}(o_2), \dots\}$
   Output: CBP plan $\mathcal{C}$
1  $\mathcal{C} \leftarrow$ InitialPartition(G);
2  compute load statistics $\mathcal{Y}(\mathtt{C}_i)$, state statistics $\mathcal{Z}(\mathtt{C}_i)$, and adaptation cost $\psi_i$ for each
     component $\mathtt{C}_i \in \mathcal{C}$ ;
3  $\delta \leftarrow 1.0$ ;
4  while $\delta > 0$ do
5      foreach $o_k \in \mathtt{O}$ do
6          $\mathtt{C}_i \leftarrow$ get the component containing $o_k$ ;
7          foreach $\mathtt{C}_j \in |\mathcal{C}|$ and $j \neq i$ do
8              if $a_{jk} \neq -1$ and $\mathtt{C}_j \cup \{o_k\}$ is compatible then
9                  $\delta^1_{ij}(o_k) \leftarrow \ell_{jk} - \ell_{ik}$ ;
10                 $\delta^2_{ij}(o_j) \leftarrow (\psi_i + \psi_j) - (\psi_i(o_k) + \psi_j(o_k))$ ;
11                 $\delta_{ij}(o_k) \leftarrow \delta^1_{ij}(o_k) + \delta^2_{ij}(o_k)$ ;
12     $\delta \leftarrow \max\{\delta_{ij}(o_k)\}$ ;
13     move $o_k$ from $\mathtt{C}_i$ to $\mathtt{C}_j$;
14     update the load and state statistics of $\mathtt{C}_1$ and $\mathtt{C}_2$;
15     recompute the profits for any admissible movement involves $\mathtt{C}_i$ or $\mathtt{C}_j$ ;
16 return $\mathcal{C}$;
```

then the load and state statistics of $\mathtt{C}_1$ and $\mathtt{C}_2$ should be changed after the execution of *move(*$\mathtt{C}_1$,$\mathtt{C}_2$,o_k*)* (Line 14). The movement also causes changes of the profits of any admissible movement involving $\mathtt{C}_1$ or $\mathtt{C}_2$. To prepare the next iteration, we should recompute the profits of these admissible movements (Line 15).

5.2 MWSC

We proceed to consider an alternative solution that transforms MCCBP into the *minimum weighted set cover problem* (MWSC). Let $\Omega = \{\mathtt{C}_1, \mathtt{C}_2, \dots\}$ be a set containing all the possible components of $\mathtt{O}$. Let N be the cardinality of Ω, i.e., $N = |\Omega|$. A CBP plan $\mathcal{C} = \{\mathtt{C}_i | \mathtt{C}_i \in \Omega\}$ is a subset of Ω. It is apparent that the plan $\mathcal{C}$ is a set cover of $\mathtt{O}$, since $\bigcup_{i=1}^{|\mathcal{C}|} \mathtt{C}_i = \mathtt{O}$ and $\mathtt{C}_i \cap \mathtt{C}_j = \emptyset$ for $\forall \mathtt{C}_i, \mathtt{C}_j \in \mathcal{C}$. Therefore, MCCBP is equivalent to find a subset $\mathcal{C}$ of Ω such that $\mathcal{C}$ is a partition of $\mathtt{O}$. We attempt to optimize this problem by enumerating all the feasible components and finding the optimal CBP plan from them.

Each component associates with adaptation cost ψ_i and intra-component communication cost ϕ_i, where $\phi_i = \sum_{o_i, o_j \in \mathtt{C}} l_{ij}$. For each component $\mathtt{C}_i \in \Omega$, we assign a weight w_i to it such that $w_i = \psi_i + l - \phi_i$, where l is the overall load, $l = \sum_{i=1}^{n} \sum_{j=1}^{n} l_{ij}$. It is obvious that $\psi_i > 0$ and $l - \phi_i \geq 0$.

Let x_i be a decision variable that indicates whether component $\mathtt{C}_i$ is chosen in the set cover $\mathcal{S}$, where $x_i = 1$ if $\mathtt{C}_i$ is picked, or $x_i = 0$ otherwise. Then the MCCBP is transformed to the weighted set cover problem. A set cover $\mathcal{S}$ of $\mathtt{O}$ has some redundant

Algorithm 2: MWSC

```
   Input: Operator graph $\mathtt{G} = (\mathtt{O}, \mathtt{S})$ , load statistics $\{\mathbf{Y}(s_1), \mathbf{Y}(s_2), \dots\}$, state statistics
          $\{\mathbf{Z}(o_1), \mathbf{Z}(o_2), \dots\}$
   Output: CBP plan $\mathcal{C}$
1  $l \leftarrow \sum_{i=1}^{n}\sum_{j=1}^{n} l_{ij}$ ;                           /* overall loads */
2  $\Omega \leftarrow \mathtt{Enumerate}(\mathtt{G}, k)$ ;
3  foreach component $\mathtt{C}_i$ in $\Omega$ do
4      compute the adaptation cost $\psi_i$ ;
5      $\phi_i \leftarrow \sum_{o_i, o_j \in \mathtt{C}} l_{ij}$ ;
6      $w_i \leftarrow \psi_i + l - \phi_i$ ;                             /* weight of $\mathtt{C}_i$ */
7  $\mathcal{S} \leftarrow$ compute the MWSC of $\mathtt{O}$ over $\Omega$ ;
8  $\mathcal{C} \leftarrow \mathcal{S}$ ;
9  return $\mathcal{C}$;
```

operators, for example $\mathtt{C}_i \cap \mathtt{C}_j = o_k$. Denote $\mathcal{S}'$ as the new set cover by discarding o_k. Since $\psi_i > 0$ and $l - \phi_i \geq 0$, the cost of $\mathcal{S}'$ is definitely smaller than that of the former one, i.e., $w(\mathcal{S}') < w(\mathcal{S})$. Finally, we can get the minimum set cover of $\mathtt{O}$ by removing all the redundant operators.

Definition 3 (MWSC). *Given a universe $\mathtt{O}$ and a family Ω of subsets of $\mathtt{O}$, the minimum weighted set cover of $\mathtt{O}$ can be expressed as an integer linear programming:*

$$\begin{aligned} &\textit{minimize} \quad w(\mathcal{S}) = \mathbf{w}^T\mathbf{x} \\ &\textit{subject to} \quad \sum_{\mathtt{C}_i : o \in \mathtt{C}_i}^{N} x_i \geq 1 \quad \textit{for each operator } o \in \mathtt{O}, \\ &\qquad\qquad\quad x_i \in \{0, 1\} \end{aligned} \tag{2}$$

where $\mathbf{w} = (w_1, \dots, w_N)$ is the weight vector and $\mathbf{x} = (x_1, \dots, x_N)$ is the decision vector for Ω respectively.

Apparently, a MWSC is a partition of $\mathtt{O}$. Thus,

$$w(\mathcal{S}) = \sum_{i=1}^{N} x_i\psi_i + |\mathtt{S}|l - \sum_{i=1}^{N} x_i\phi_i = \underbrace{\sum_{i=1}^{N} x_i\psi_i} + \underbrace{\left[l - \sum_{i=1}^{N} x_i\phi_i\right]} + \underbrace{(|\mathtt{S}| - 1)l} \tag{3}$$

where $|\mathtt{S}|$ is the number of edges of $\mathtt{G} = (\mathtt{O}, \mathtt{S})$.

Comparing to the cost model Eq. (1), we have the first component $\sum_{i=1}^{N} x_i\psi_i$ and the second $l - \sum_{i=1}^{N} x_i\phi_i$ of Eq. (3) equal to the adaptation cost $\mathcal{AC}$ and communication cost $\mathcal{CC}$ respectively. As the third component $(|\mathtt{S}| - 1)l$ is a constant, the best solution of MWSC is equivalent to the optimal CBP plan.

The idea is depicted in Alg. 2. We first enumerate all the possible components of $\mathtt{G}$ (Line 2). Then we compute the adaptation cost ψ_i and the load ϕ_i of each component $\mathtt{C}_i$, and assign a weight to each component (Line 3–6). Finally, we compute a solution

$\mathcal{S}$ of MWSC and take it as a CBP plan by discarding all the redundant operators (Line 7–8). MWSC can be solved exactly with a MIP solver like Gurobi [2] when N is not too large. But we also implement a greedy routine to solve MWSC (Line 7) according to the description in [9, chap. 35]. The greedy routine is a useful option when N is large. Since the set cover $\mathcal{S}$ obtained through the greedy routine might not be a CBP plan, we have to remove the redundant operators to get the final solution $\mathcal{C}$.

6 Evaluation

6.1 Experimental Setup

Evaluation metrics—We use the following metrics in the evaluation:

– **Communication cost** counts the number of tuples transmitted through inter-component communication.
– **Adaptation cost** counts the number of state movements in an adaptation process.
– **End-to-end latency** indicates the time completing the processing of a source tuple. It includes the time spent on processing, adaptation, and communication, and thus it is a overall metric to reflect the effectiveness of CBP.

Tested solutions— We implement the sparse-cut algorithm, a graph partition algorithm used in COLA [15], to compare with our solutions. Note that the objective of baseline is merely to minimize communication cost. In general, we evaluated the following three algorithms: (1) *greedy algorithm*, (2) the *MWSC algorithm*, and (3) the *baseline algorithm* which implements an OBP-based operator placement algorithm in [15].

We implement our algorithms in Java and integrate them with Apache Storm [1] by extending it with runtime adaptation. Part of the experiments are conducted via simulation, while the rest are conducted on Amazon's EC2 with medium VM instances (*m1.medium*), where each has 1.7 GB of RAM, moderate IO performance and one EC2 compute unit (approximately equivalent to a 1.2 GHz 2007 Xeon CPU). While these VMs have low processing capabilities, they are representatives of public cloud VMs.

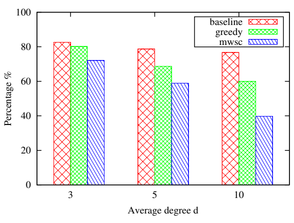
6.2 Simulation Result

In the test, we used a randomized topology $\mathtt{G} = (\mathtt{O}, \mathtt{S})$. In the topology $\mathtt{G}$, each operator o, except src, maintains computing states and randomly forwards the received data to downstream operators according to the selectivity $\delta(o)$. The specific setting of $\mathtt{G}$ is summarized in Fig. 2. Operator src generates two synthetic streams s_1 and s_2 to simulate two types of variations, where the key values of s_1 and s_2 follow the uniform distribution and Zipf respectively. Therefore, s_1 only results in scaling. In contrast, data distribution of s_2 is skewed and thus the adaptations involve both scaling and load balancing. Each operator of $\mathtt{G}$ randomly chooses two attribute of sch as the partition key. The data arrivals of s_1 and s_2 follow a Poisson process $X(t) : P[N(t+\tau) - N(t) = k] = (k!)^{-1}e^{-\lambda\tau}(\lambda\tau)^k$, where τ is set to 1 second and $\lambda = 10,000$. Both s_1 and s_2 conform to the schema: SynStream(ts:Unix timestamp, a_1:int, a_2:int, a_3:int, a_4:int), in which each attribute has 4 Bytes.

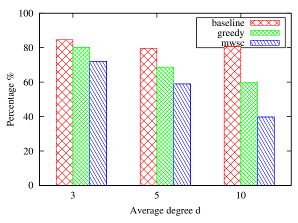
We measured the communication cost and state movements by varying the average degree d and the adaptation frequency f. Let N_1 be the number of tuples processed by

Parameters	Settings
Random graph	$\mathtt{G} = (\mathtt{O}, \mathtt{S}, d)$
Number of operators	$\lvert\mathtt{O}\rvert = 100$
Average degree d	$d = \{3, 5, 10\}$
Selectivity $\delta(o)$	$\delta(o) \sim N(0.5, 1.0)$
Size of states $\lvert\mathtt{PS}_o\rvert$	$\lvert\mathtt{PS}_o\rvert = X(t)$

Fig. 2: Setting of parameters



(a) stream s_1



(b) stream s_2

Fig. 3: Comparison of communication costs

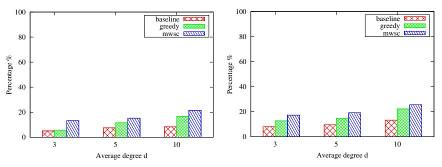
all the operators and N_2 be the number of tuples in the states of all the operators at every adaptation. To have a better understanding of experimental results, we calculate the percentages, $\frac{100n_c}{N_1}$ and $\frac{100n_a}{N_2}$, of tuples involved in the communication and state movement, where n_c and n_a are the communication cost and adaptation cost respectively.

Comparison of communication costs—Fig. 3a and Fig. 3b show the percentages achieved by three algorithms. We can observe that the baseline algorithm can save the cost by at most 20%, but the CBP solutions can reduce the cost by at least 20%. In particular, the greedy algorithm saves about 20% when $d = 3$, and it increases to 40% when $d = 10$. MWSC outperforms the greedy algorithm. It significantly reduces the communication cost by about 27.8% when $d = 3$ and by nearly 60% when $d = 10$. The baseline algorithm deploys the operator graph based on a placement plan, which is generated in advance with a graph partitioning algorithm. Since the operators are incompatible, the physical topology of the query changes as adaptation process. The parallelization plan is no longer optimal when the physical topology has been changed. Therefore, we cannot optimize the communication cost efficiently by only leveraging operator placement.

The intra-component communications of a CBP plan are eliminated completely regardless of the change of physical topology. This is confirmed by Fig. 3. By comparing Fig. 3a and Fig. 3b, the communication costs of CBP solutions keep the same regardless of the difference of s_1 and s_2. However, the costs of baseline algorithm is slightly different over s_1 and s_2, where the cost is about 3% higher over s_2 than that over s_1. The frequency f shows similar impact to the algorithms.

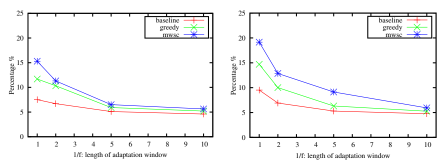
Comparison of adaptation costs—Fig.4 and Fig. 5 shows the impact of load variation and the adaptation frequency. In this experiment, the frequency f is varied by changing the length of adaptation window from 1 minute to 10 minutes, i.e., $1/f = \{1, 2, 5, 10\}$. Fig.4 plots the adaptation cost of each algorithm when $1/f = 1$. It is clear that CBP has larger adaptation costs than the baseline algorithm. Moreover, the adaptation cost over a skewed stream, s_2 in Fig. 4b, is higher than the uniformly distributed stream, s_1 in Fig. 4a. We observe similar results when $f = \{2, 5, 10\}$. The results also justify an implicit assumption in this paper that the adaptation cost is normally higher when we assemble operators into components.

Fig. 5 shows the impact of adaptation frequency f. As we can see from the figure, the cost drops greatly at the beginning when we increase $1/f$. The number of state movements is determined by two factors: (1) the adaptation frequency f, and (2) the skewness of the data. The skewness usually goes serious if we increase $1/f$, i.e., it al-



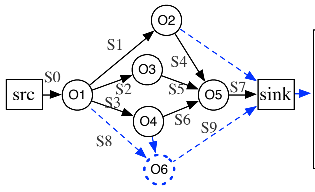
(a) stream s_1 (b) stream s_2

Fig. 4: Comparison of adaptation costs



(a) stream s_1 (b) stream s_2

Fig. 5: Adaptation costs with respect to $1/f$



Schema	ID
(*Ts*, *Vid*, *XWay*, *Dir*, *Seg*, *Spd*, *Pos*)	S1, S2, S3
(*Ts*, *Vid*, *XWay*, *Dir*, *Seg*, *Acdt*)	S4
(*Ts*, *Vid*, *XWay*, *Dir*, *Seg*, *AvgSpd*)	S5
(*Ts*, *XWay*, *Dir*, *Seg*, *VehNum*)	S6
(*Ts*, *Vid*, *XWay*, *Dir*, *Seg*, *Acdt*)	S7
(*Ts*, *Vid*, *XWay*, *Dir*, *Seg*, *Toll*)	S8

Operator	Partition Key
O1:Forwarder	{*Ts*, *Vid*, *XWay*, *Dir*, *Seg*, *Spd*, *Pos*, Type}
O2:AcdDetector	{*Ts*, *Vid*}
O3:AvgSpeed	{*Vid*, *XWay*, *Dir*, *Seg*}
O4:SegVolume	{*Xway*, *Dir*, *Seg*}
O5:TollCalculator	{*Xway*, *Dir*, *Seg*}

Fig. 6: Operator graph for LRB

ways involves more state movements in a single adaptation. As we expected, the decline of adaptation cost is much gentle when $1/f$ is larger.

6.3 End-to-end Latency

We proceed to evaluate the end-to-end latency of the tested solutions. In this experiment, we use the *Linear Road Benchmark* (LRB) [6]. LRB models a road toll network, in which tolls depend on the level of congestion. The primitive LRB gadget only has 7 operators, which is too small to represent a large-scale computation. So we extend it by connecting a number of LRB gadgets together with a road network. The road network $\mathtt{G} = (\mathtt{V}, \mathtt{E})$ is a graph where an edge $e \in \mathtt{E}$ stands for an expressway of LRB and a vertex $v \in \mathtt{V}$ represents the joint of expressways. This extension has a wide range of applications. If we want to measure the traffic between two locations or track the route of a vehicle, then an LRB gadget must dispatch result to its downstream LRB gadgets. Consequently, we introduce a new operator o_6 to calculate the traffic between every pairs of vertices every 1 minute. Fig. 6 shows the topology of the extended LRB, where some new streams (blue dashed arcs), have been added into a LRB gadget to fulfill the requirement.

$\mathtt{G} = (\mathtt{V}, \mathtt{E})$ is generated with the random graph presented in Section 6.2. In particular, $|\mathtt{V}| = 10$, $|\mathtt{E}| = 30$, and the average degree $d = 3$. Therefore, we have 30 LRB gadgets and 180 operators in total excluding srcs and $sink$s. For each LRB gadget, the data rate of the source stream is controlled with the Poisson process used in the previous section. The experiments are conducted on EC2 with 30 VMs and accomplished in two phases: (1) We first deploy it over EC2, and keep it running for two hours to collect statistics. The length of statistic window is set to 1 minute. (2) With the statistics, we partition the topology into components or subgraphs with the tested algorithms. Thereafter, we deploy the partitioned topology on EC2 and run the experiments.

We measure the end-to-end latency at 4 scales of the adaptation frequency f, i.e., $1/f = \{1, 2, 5, 10\}$. The latency values are given in Tab. 1, where "95%" is the 95th percentile of latency. In general, the results follow what we expected. By comparing

Table 1: Statistics about end-to-end latency(ms)

	Mean				Median				95%				Maximum			
1/f	1	2	5	10	1	2	5	10	1	2	5	10	1	2	5	10
Greedy	677	610	566	617	141	121	109	116	1501	1236	1095	1130	2825	2223	1736	2117
MWSC	583	517	534	602	131	120	97	118	1532	1333	1054	1171	3103	2703	1853	1853
Baseline	775	710	673	681	153	137	114	127	1017	928	856	889	2109	1809	1673	1681

the mean, we observe that our algorithms reduce the latencies by about 10%–25%. It shows that the CBP algorithm can indeed improve the performance and thus confirms the effectiveness of CBP. We can further identify the impacts of adaptation process and load imbalance in these values. For example, the CBP algorithms are more sensitive to adaptation process and load imbalance comparing to the greedy algorithm. The maximum latency is 3103 ms for MWSC when $1/f = 1$, which is higher than the maximum latency of Baseline.

Tuples with a latency smaller than the median are less affected by the adaptation process and load imbalance. In contrast, tuples with latencies larger than 95-th percentiles are greatly affected by the adaptation process and load imbalance. We take the latency when $1/f = 1$ as an example, the medians of MWSC and Greedy are about 75% and 87% of that of Baseline. So the results confirm that CBP can save communication cost efficiently. In contrast, the 95-th percentiles for MWSC and Greedy are about 29% and 26% greater than the baseline algorithm.

During an adaptation, input tuples are buffered by the upstream operators. The tuples will be replayed to downstream after the completion of adaptation. Therefore, adaptation process increases the end-to-end latencies for a portion of tuples. As we can see from the table, the maximum latency peaks up to about 3 seconds.

For each algorithm, each numeral characteristic drops with the increase of $1/f$ at first and then grow with the increase of $1/f$ on the contrary. This behavior is obvious for the 95-th percentile. In terms of the 95-th percentile, it is obvious MWSC is higher than Greedy and Baseline. This phenomena confirms the impact of adaptation process and load imbalance. The adaptation cost drops with the increase of $1/f$, but load imbalance get worse on the contrary. Thus we observe that all lines are concave. It means that the adaptation frequency is very important as it can trade off between impact of adaptation cost and load imbalance. In this experiment, $f = 1/2$ is the best choice for MWSC and $f = 1/5$ is the best choice for Greedy and Baseline.

7 Related Work

The related work falls into two categories: (1) scalable parallel stream processing, and (2) operator placement.

Parallel stream processing. Much work has been focused on exploiting parallelism in stream processing. The early SPEs aim at providing transparent parallelization for distributed stream processing in a shared-nothing environment. Aurora [7] and Borealis [3] supports intra-query parallelism by organizing a topology into a set of boxes and conducting parallelization via *box-splitting*.

Many SPE proposals, e.g., System S [5] and Flux [22], leverage partitioned parallelism [11] to improve scalability. They propose new new ”Exchange” operators between stream producers and consumers to encapsulate the adaptive state partitioning and stream routing. In recent years, many efforts have been made to improve the scalability of parallelization [12,21,20]. The MapReduce model [10] enables programmer to think in a *data-centric* fashion and hence provides a practical implementation for partitioned parallelism. Distributed SPEs like Apache Storm [1], Yahoo! S4 [18], and StreamCloud [14] are inspired by such a model.

Operator placement. If an application is geographically distributed, the transmission latency is sensitive to the communication channels. The SAND project [4] exploits the knowledge of the underlying network characteristics such as topology and link bandwidths to make intelligent in-network placement of query graph. In contrast, [19] develops a *stream-based overlay network* (SBON) over Borealis, which is a network-aware optimization framework that manages operator placement within a pool of wide-area overlay nodes in order to make efficient use of network bandwidth. The placement decisions are made based on the cost space that encodes multidimensional metrics such as latency and load.

COLA [15] employs graph-partitioning algorithms to compute an optimal allocation of operators with regard to a cost model that captures the communication and CPU cost. The operator graph is partitioned into subgraphs at compile-time, which acts as a processing element (PE) and a deployable unit. COLA attempts to balance load across the processing nodes and minimize the communication cost of the PEs. It only measures the CPU usages incurred by processing and communicating, but ignores the network bandwidth usage. However COLA does not consider how to parallel the operators. Moreover a partition plan obtained at compile-time is incapable to handle the load variations at runtime.

The essence of operator placement is to optimize an assignment of operators to the computing nodes based on an objective function. Unfortunately, the existing solutions are static and the cost of the state migration cannot be ignored in the presence of load variations. For more detailed comparisons of the placement strategies, please refer to a survey paper [16].

8 Conclusion

We present CBP, a succinct parallelization paradigm for DSPEs that leverages both the connectivity and compatibility of operators. CBP seamlessly integrates operator placement with parallelization and thereby provides a framework to integrate the optimizations of runtime resource reconfiguration and communication cost minimization. Furthermore, we introduce a cost model that captures the cost of communication and adaptation. Two algorithms are proposed to optimize the CBP plans for a given computation. The extensive experiments confirm that an optimized CBP plan can improve the resource efficiency of DSPEs significantly.

References

1. Apache Storm. `http://storm.apache.org/`.
2. Gurobi Parallel MIP solver. `http://www.gurobi.com/resources/getting-started/mip-basics`.

3. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR '05*, Asilomar, CA, January 2005.
4. Y. Ahmad and U. Çetintemel. Network-aware Query Processing for Stream-based Applications. volume 30 of *VLDB '04*, pages 456–467, 2004.
5. H. Andrade, B. Gedik, K. Wu, and P. S. Yu. Scale-up strategies for processing high-rate data streams in system S. ICDE '09.
6. A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *VLDB '04*.
7. D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. VLDB '02, pages 215–226, 2002.
8. R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.
9. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
10. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. volume 6 of *OSDI'04*, Berkeley, CA, USA, 2004. USENIX Association.
11. D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
12. B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25:1447–63, 2010.
13. G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. SIGMOD '90, pages 102–111. ACM, 1990.
14. V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. StreamCloud: A Large Scale Data Streaming System. ICDCS '10, pages 126–137, 2010.
15. R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. COLA: Optimizing Stream Processing Applications via Graph Partitioning. Middleware '09, pages 16:1–16:20. Springer-Verlag New York, Inc., 2009.
16. G. T. Lakshmanan, Y. Li, and R. Strom. Placement strategies for internet-scale data stream systems. *Internet Computing, IEEE*, 12(6):50–60, Nov 2008.
17. R. Motwani, J. Widom, and et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR '03*, pages 245–256, January 2003.
18. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.
19. P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. ICDE '06. IEEE, 2006.
20. S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. Elastic scaling of data parallel operators in stream processing. *IPDPS*, pages 1–12, 2009.
21. S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. PACT '12, pages 53–64, New York, NY, USA, 2012. ACM.
22. M. A. Shah, S. Chandrasekaran, J. M. Hellerstein, S. Ch, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, pages 25–36, 2002.
23. S. Wu, V. Kumar, K.-L. Wu, and B. C. Ooi. Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? DEBS '12, pages 278–289.
24. Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing Resiliency to Load Variations in Distributed Stream Processing. In *VLDB '06*, pages 775–786. VLDB Endowment, 2006.
25. Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *ICDE '05*, pages 791–802. IEEE Computer Society, 2005.