UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MAYCON VIANA BORDIN

# Operator Placement Algorithms for Distributed Stream Processing Systems

Trabalho Individual I
TI-1

Prof. Dr. Cláudio Fernando Resin Geyer
Advisor

Porto Alegre, october 2013

# CONTENTS

# LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| DBMS | DataBase Management System |
| DSMS | DataStream Management System |
| SPE | Stream Processing Engine |
| SPS | Stream Processing System |
| DSPS | Distributed Stream Processing Engine |

# LIST OF FIGURES

# LIST OF TABLES

# ABSTRACT

Processing continuous streams of data in real-time is not an easy task, and as such stream processing systems (SPS) were devised to help the development of such applications. As the volume of data to be processed increased, these systems moved to a distributed architecture. One of the challenges of distribution lies in the way the load is going to be distributed among the nodes.

A stream processing application is composed of many operators, and the decision of where to place them is made by the scheduler. It has to find a placement plan that evenly distributes the load while meeting the QoS requirements of the application. Furthermore, as data streams have a bursty nature, the scheduler must monitor the system status in order to adapt to load changes without violating the QoS requirements.

In this report we are going to study the main algorithms for operator placement in a distributed stream processing system, highlighting the techniques employed by each solution and comparing them at the end.

**Keywords:** Stream processing systems, distributed systems, real-time processing.

# 1 STREAM PROCESSING

## 1.1 Introduction

A relatively new class of applications has emerged in the last few years, such as sensor data processing, traffic monitoring, stock trading, homeland security and network monitoring. This type of applications generate data in the form of *data streams* at a high input-rate and at a continuos way.

These applications require that data be processed in real-time, because they need to react in a timely manner to environmental changes or new trends, such as a DDoS attack to a web service.

Those requirements make traditional data processing technologies, such as relational DBMSs, unsuitable for stream processing, since data arrives continuously, and so queries must also be performed continuously. Therefore, applications have been developed specially for processing data streams and they are called Data Stream Management Systems (DSMSs), Stream Processing Engines (SPEs) or Stream Processing Systems (SPS). For now on we are going do address to such systems as SPSs.

According to Chakravarthy and Jiang [10], the main shift from traditional DBMSs to SPSs is that now, due to low-latency requirements, data is first processed in memory, therefore decreasing the response time of such applications. The authors also point out that there are some variants of DBMSs that are relevant for stream processing applications, such as in-memory databases, embedded databases and real-time transaction processing systems.

Besides the characteristics above described, Chaudhry, Shaw and Abdelguerfi [12] also points out that in stream processing systems there is a good notion of time, mainly because a data stream is a sequence of values that represent the same entity over time, which also means that this data is not finite.

## 1.2 Requirements

The characteristics of a SPS are, in the same way as those of the traditional DBMS, a generalization of the requirements of a broad spectrum of applications. Chakravarthy and Jiang [10] lists the following requirements common to all of those applications:

- Data arrives continuously, it means that queries need to be evaluated repeatedly.
- Some applications can tolerate results that are not 100% accurate. It may happen due to data loss or as way to the system comply with other requirements (droping tuples to alleviate system load), such as response time or resource usage.

- Besides the system response time and resource usage, these applications have other QoS requirements, such as precision, tuple latency, throughput and scalability.
- In order to meet those QoS requirements, resource utilization has to be carefully managed.
- At last, stream processing applications usually work together with Complex Event Processing (CEP), rule processing and notification systems.

Going further on the specification of the requirements for stream processing systems, Stonebraker, Çetintemel and Zdonik [30] define 8 requirements of such systems. According to the authors, those systems should:

1. Reduce as much as possible tuple latency, which means keep data in memory for processing and leaving persistent storage aside (unless it can be done without impacting performance);
2. enable querying data with something similar to SQL;
3. handle stream imperfections (delayed, missing ad out-of-order data);
4. generate deterministic and repeatable results;
5. allow the system to manage state information, preferably stored in memory;
6. guarantee the availability and integrity of the system at all times (HA);
7. scale through multiple processors and machines, as automatically and transparently as possible; and
8. be able to cope with high loads of data with minimal overhead and latency penalties.

One interesting requirement defined by Abadi *et al.* [1] is that SPSs should be able to modify already consolidated results, a feature that would be used for correcting errors, revising results due to tuples that arrived too late, or in the case of stock market applications, to include revision records to the results. This requirement is directly related to the third requirement listed above, as it proposes a solution to some of the imperfections that appear in data streams.

The authors also define as a requirement the dynamic modification of queries in a fast and automatic manner. This is somewhat similar to the second requirement in the list above about the use of a query language similar to SQL, because such language would be much more flexible than applications developed in languages such as C++ or Java.

In addition to the capability of dynamically modificating queries, a SPS should support the addition or removal of queries on-the-fly [11].

## 1.3 Concepts

**Data streams**. A data stream is a unbounded sequence of data items, tuples or events that arrives continuously, usually ordered by a timestamp or by one or more values of its data elements. Usually, the input rate of a data stream can't be controlled, and it can range from a few bytes per second to a gigabytes or even more. The input rate can also be irregular, having what is called a *burst*, which is a spike in the input rate. Twitter

experiences such bursts in the face of big events such as the World Cup or New Year's Eve.

Even though tuples in a data stream arrive sequentially, it does not mean they can't arrive out-of-order. In fact, as data streams are external sources of data, they are suceptible to corruption or even data loss [12].

**Continuous queries**. Since data streams are unbounded by nature, queries can't be executed only once, instead they run continuously. Continuous queries (CQs) can be usually organized as a *directed acyclic graph* (DAG) or *dataflow graph* where the vertices are the operators and the edges are streams of data [17]. A CQ (or application) is thus represented by a *processing graph* (Figure 1.1) composed of *stream sources* that feed the system with data, operators that transform the data and *sinks* that receive the final product of the CQ.

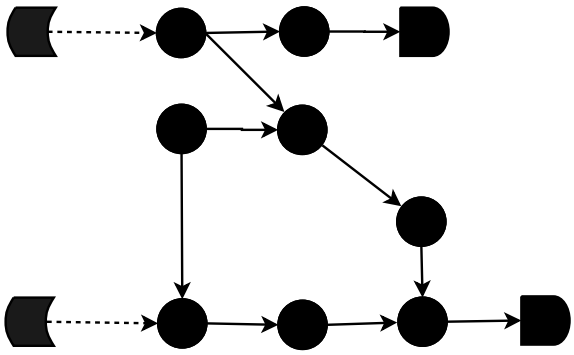

Figure 1.1: Example of a Processing Graph

**Operators**. Operators receive one or more data streams as input and produce one or more data streams as output. Operators can be classified as *stateful* or *stateless*. Operators such as a *filter* or a *map* are stateless because they don't store any information regarding the tuples processed, whereas a *count* or *sum* are classified as statefull.

The operators described above are non-blocking, because they are able to produce an output each time an input arrives. A *join* operator, for example, is blocking, because it can't produce an output until all input has arrived. But since data streams are unbounded, a *join* would wait forever. To solve this problem most systems apply a window-based data model.

**Window**. A window represents a finite portion of a stream with which operators will do thei computation and generate an output. Windows can be defined over a period of time (time-based or physical windows) or over the number of tuples (tuple-based or logical windows).

A window is defined by a window start $W_S$, a window end $W_E$, its $Size$ and an $Advance$. The configuration of these parameters is what will determine the type of the window. Assuming that $t_{in}$ is the most recent tuple to arrive, if $W_S$ is fixed and a $W_E = t_{in}.timestamp$, then the $Size$ will always grow and the window is a *landmark*.

If the $Size$ is fixed, such that $W_E - W_S = Size$, then the window is called *sliding* and the $Advance$ parameter is what will determine its behavior. With $Advance = 0$ the arrival of one tuple will remove the oldest one from a tuple-based window, while in a time-based one, given that $t_{in}.timestamp > W_E$, the new window boundaries will be $[t_{in}.timestamp - Size + 1, t_{in}.timestamp + 1[$.

With $Advance < Size$ the boundaries of the window will be updated to $[W_S + Advance, W_E + Advance[$. And if $Advance = Size$ the new boundaries will be $[W_S +$

$Size, W_E + Size[$, which is called a *tumbling* window because $W_i \cap W_{i+1} = \emptyset$.

**QoS**. The real-time or *near* real-time nature of stream processing imposes some constraints regarding the Quality of Service. These constraints are application-specific and the DSPS should deliver them at all costs. The main QoS metrics for stream processing are:

- *tuple latency* or *end-to-end delay*: time (or average time) for a tuple to traverse the processing graph.
- *memory usage*: the maximum amount of memory the system can use.
- *throughput*: number of tuples processed per unit of time.
- *nature of output streams*: whether the tuples are output regularly or in a bursty manner.
- *accuracy*: whether the results are exact or approximate.

As these metrics are not independent of each other, trade-offs will have to be made. For example, to reduce the memory usage it may be necessary to discard some tuples, which in turn will reduce the accuracy of the results. Or, in order to increase throughput buffers hava is size increased, also increasing the average latency of tuples.

## 1.4 Operator Placement

Meeting QoS requirements is not an easy task as it usually involves balancing aspects that most of the time work against each other. In a environment where data can present bursts of volume, a system has to be able to cope with it, while maintaining the stablished QoS.

In a DSPS, the operators of a query will be spread across multiple computing nodes, thus meeting the QoS requirements means finding a good operator placement and since stream loads can greatly vary along time, these systems should also employ good load balancing techniques.

The processing graph is considered the logical representation of the query which will be used by the scheduler in the placement of operators. The mapping of operators to computing nodes will create the physical representation of the query, where the vertices now represent the nodes [24, 29].

One of the main differences in the way the scheduler works depends on the type of **load partitioning** supported by the system [21]. **Data stream partitioning** allows an operator to be placed in several nodes, distributing the load among them. Whereas in **query plan partitioning** one operator can live only in one node, with the disavantage that if an operator consumes more resources that a machine can offer, the only solution is to upgrade the machine or apply some technique of admission control.

Following the taxonomy defined by Casavant and Kuhl [8] another important aspect of schedulers for DSPSs is the time at which the operator placement is made (**statically** or **dynamically**). Giving the variable nature of data streams most schedulers proposed in the literature approach the problem with a initial operator placement and an online load balancing, with special attention to the management of bursts.

# 2 ALGORITHMS

## 2.1 Medusa Scheduler

Probably one of the first schedulers for DSPSs, the one built for Medusa [13, 6, 7] is based on economic fundamentals in order to regulate the collaboration between the selfish participants in a distributed way.

Each participant has a maximum load level incurred from all the tasks running on it. This load has a processing cost, which is computed by a function that may differ from participant to participant. The proposed solution uses contracts as a mechanism to distribute tasks without overloading any participant.

A contract $C$ is negotiated offline between two participants and it defines the price ranges for moving load in each direction. Different pairs of participants may have different contracts. At runtime, when a participant $i$ is going to transfer a set of tasks $moveset$ to another participant $j$, a unit price $price(C_{i,j})$, within the price range, will be agreed upon for the $moveset$, so that the price of the transfer is $price(C_{i,j}) \times load(moveset)$.

The first solution presented in the paper is the *fixed-price* mechanism. It fixes the lower and upper bounds in the price to the same value. By doing so it is possible to check if the marginal cost per unit of load is higher than the fixed price. In such case, it's cheaper to process the task at the partner than locally. On the other hand, if the marginal cost is lower than the fixed price, then it is best to accept the task. It's important to note that a participant should never accept a fixed price above the marginal cost per unit of load.

An overloaded participant will select tasks that are cheaper to outsource to a partner. It will attempt to transfer the load excess until it succeeds. The participant can choose the contract that offers the lower unit price, in the same way that the partner will try to accept the offers with higher profits.

An extension of the fixed-price contract is proposed, with a small range of prices. This small range enables a participant to accept tasks at higher prices and offer them at lower prices. With this extension a participant can, for example, forward the load of an overloaded partner to a more lightly loaded one.

The mechanism is *descentralized* and *algorithmic* (DAM – distributed algorithmic mechanism). *Indirect*, because the participants reveal their costs and tasks by means of offers and tasks acceptations. *Idividual rational* as a participant can't decrease its utility by participating, he will, however, increase it by accepting tasks at a higher price than the marginal cost or the opposite for task offerings. And *dominant*, because two participants with a signed contract may optimize its utility independently of each other.

The mechanism was evaluated both in a simulated environment and with a real application. The simulation observed the convergence to acceptable allocations in an heterogenous environment, the speed of convergence with different network topologies and how

the mechanism behaves in the face of load variations. Results showed that the mechanism was able to converge to an acceptable allocation both in an underloaded and overloaded scenarios with a minimum of 10 contracts[1]. Switching from fixed-price to price-range contracts also improved the final allocation.

Due to the way the mechanism works, the convergence in all cases evaluated happens within less than 15 seconds. With fixed-price contracts the convergence is faster than price-range because it can move more operators in a single iteration, altough the allocation quality is worse than the price-range. Because fixed-price contracts lead to faster convergences, they are able to handle load variations with fewer re-allocations than with price-range contracts.

The experiment with a network intrusion network only shows the load at each node along the time, enabling the visualization of nodes getting overloaded and shedding part of its load to another node. The experiment does not show any results regarding QoS metrics, or does any comparison with alternative scheduling algorithms.

## 2.2 Network-Aware Query Processing for Stream-Based Applications

The work of Ahmad and Çetintemel [2] deals with the problem of operator placement in very large networks (*i.e.* the Internet) and introduces a set of algorithms that leverage information about the network characteristics to improve the query processing.

In their system an application is represented by a processing tree. To improve the scalability and parallelism of the system the processing tree is partitioned into subtrees (called zones), and each one of this zones is assigned to a coordinator node from the control tree. The coordinator will be responsible for the placement of operators (and dynamic re-placement) and the correct and highly-available execution in his zone.

The first algorithm presented is called **Edge** placement, it's a greedy algorithm that traverses the processing tree operators in post-order, progressively optimizing it. With each operator the algorithm tries to find the location that yields the least cost, taking into account that its cost also depends on the placement of its children. The placement of operators can be

1. at a location that maximizes the total tree cost between the operator and all of its children;

2. at common location where all the children are going to be placed, thus eliminating the communication costs; or

3. placing all children at the proxy (*i.e.* the application that receives the results from the processing tree), which can be beneficial when the cost grows higher near the root of the tree.

The second algorithm, called **Edge+**, is the network-aware version of the Edge placement. One of the changes in this algorithm is in the cost function, that now includes a symmetry distance function. In addition to the three cases listed above, the Edge+ placement also has a fourth case that selects the best (shortest total distance) among a collection of permutations of a given child's descendant locations.

A third algorithm differs from the previous ones by considering the placement of operators at arbitrary network locations. The **In-Network** placement uses a heuristic to

[1] Increasing the number of contracts reduces the diameter of the contract network

prune locations whose distance to all current child placements is greater than that of all pairwise locations.

The last algorithm presented by the authors is the **Latency-Constrained** placement. It extends the previous algorithms by adding a response time constraint, thus bounding the leaf-to-root total distance and consequently reducing the size of the search space for the placement of operators.

A comparison between these algorithms was conducted, evaluating several aspects of them with diffent configurations of proxy location. Regarding bandwidth consumption, both Edge+ and In-network showed similar results, and in all placement schemes they were superior to the Edge algorithm.

Another evaluation measured the latency overhead incurred by the system under loosely and tightly constrained scenarios. In the second scenario the Edge algorithm showed results comparable to Edge+ and In-network, however, it never outperformed the others. The In-network did better in the loosely constrained scenario due to the placement of operators in the direction of the proxy, while the Edge+ does not consider such direction, creating paths with higher end-to-end delay. In the tightly constrained scenario, however, the Edge+ algorithm outperformed the In-network as it had more room for optimizations.

## 2.3 Resource-Aware Distributed Stream Management Using Dynamic Overlays

Another scheduler with network and resource-awareness was proposed by Kumar *et al.* [25]. It is meant for very large networks where the data has to traverse a number of intermediate nodes until arrive in its destination. The basic idea is to make each one of these intermediate nodes to do some of the computation, resulting in a better distribution of workload and reduction of communication overhead.

In order to provide a solution that is scalable, the physical nodes that compose the network are organized in a hierarchy in such a way that nodes that are closer (*i.e.* low latency) belong to the same cluster. Each node knows the path cost between each node within the cluster, whereas the amount of non-local information is limited by setting an upper limit on the number of nodes that can compose a cluster. This scheme is very similar to the one described by Ahmad and Çetintemel [2].

This network partitioning is used to deploy the data-flow graph that represents the application without full knowledge of the network by handing it over to the coordinators in the top-level of the partition hierarchy. At this level a set of possible placements are created and the total cost calculated, an the one with the lowest cost is chosen. The graph is splitted into sub-graphs and deployed in the same way in the next level until it reaches the level 1, which is where the physical nodes reside.

By monitoring certain events, such as changes in network delays, available bandwidth, operator behavior and processing capacities, the solution can respond to it by reconfiguring the placement of operators. The process has a low-overhead because it happens within a cluster and the number of reconfigurations can be limited by setting thresholds for the events.

Experiments conducted by the authors showed that their solution presented a cost of deployment not much worse that with a centralized approach (optimal solution). The use of reconfiguration also proved to be effective in delivering a lower end-to-end delay in comparison with the static-only deployment, whereas the bandwidth consumption didn't change significantly.

## 2.4 Dynamic Load Distribution in the Borealis Stream Processor

This paper [34] describes the scheduler employed in the Borealis Project, a distributed stream processing engine built upon the Aurora and Medusa projects [9].

The solution presented is composed of a global operator mapping and a pair-wise load redistribution algorithm. The first one is employed mainly in the initial deployment of operators, whereas the second algorithm enables the system to adapt to load changes with a smaller incurrence of load changes.

The global operator distribution algorithm works in two steps: (1) distribution of operators with a greedy algorithm, with the goal of reducing the average load variance and balance the load among the nodes; and (2) the maximization of the average load correlation of the system.

The load of an operator is represented as a fixed length time series, and the correlation between two time series is measured by the correlation coefficient, which is a number between -1 and 1. When the coefficient is positive, it means that if the value of one time series at a certain position is large (if compared to its mean), the value at the same position in the other time series is also large. But if the coefficient value is negative, when one time series has a large value the other time series will have a small one.

It has been observed that when the correlation coefficient of two operators is small, placing them in the same node can minimize the load variance. Another observation is that maximizing the correlation coefficient among the nodes of the system will keep their load levels balanced, even in face of load changes, and thus reducing the number of load migrations, which can temporarily deteriorate the system's peformance.

The global algorithm starts with $n$ nodes that have only operators that can't be moved. At each round the node with the lowest load is selected and the (movable) operator $o$ with the highest score $S(o, N_i)$ with respect to node $N_i$ is deployed. After all operators have been placed, the system checks if the average load correlation of all nodes is above a threshold, if not the node pair with the lowest correlation is selected and the two-way operator redistribution algorithm is applied. The new mapping is only accepted if the correlation coefficient is greater than the old one. This process is repeated until the average load correlation of the system becomes greater than the thresold or the maximum number of iterations is reached.

Apart from the initial operator placement, the coordinator of the system periodically receives load information from all nodes. Each time this information is gathered the coordinator sorts the list of nodes by their average load, pairing the nodes with the largest load with the ones with the smallest load. When the load difference between the node pairs is above a predefined threshold, operators will be moved.

The authors proposed three approaches for load balancing between node pairs. The **one-way correlation-based load balancing** only moves the load from the more loaded node to the less loaded one, thus minimizing the overhead incurred from load movement. In this approach operators are ranked based on

$$S(o) = \frac{\rho(o, N_1) - \rho(o, N_2)}{2}$$

and the ones with the righest scores are selected for migration, until the total selected load is less than $(L_1 - L_2)/2$, where $L_i$ is the load at node $N_i$.

In the **two-way correlation-based redistribution** operators can be moved both ways, achieving the best operator mapping quality at the cost of a larger migration overhead. This approach starts "fresh", which means all operators are going to be redistributed,

regardless of their previous locations. The behavior of this algorithm has been described before, except that now the algorithm takes only two nodes in consideration, instead of $n$.

The third approach, called **two-way correlation-based selective exchange** allows load movement in both directions, but only for operators whose score, calculated by the function described in the first approach, is greater than a threshold.

Beyond the approaches above described, the authors also suggest an improvement to all of them, adding a correlation improvement step after the load balancing. This step is only triggered when a node is about to get overloaded, in such case operators are moved between a node pair if their load correlation coefficient is below a threshold.

Results from experiments conducted in a simulation showed that the global algorithm (called *correlation based algorithm* (COR-BAL)) has a much smaller load variance (close to the optimal) in comparison with the *randomized load balancing algorithm* (RAND-BAL) and the *largest-load-first load balancing algorithm* (LLF-BAL), which also leads to a small end-to-end latency. The algorithm proved to be superior regarding to load changes, managing to keep the load movement at a minimum, while the other two algorithms performed poorly.

For the one-way pair-wise load balancing, the three algorithms (COR-BAL, RAND-BAL and LLF-BAL) are also compared. While all of them move the same amount of load, the COR-BAL algorithm showed the best performance, but in the other hand none of them could outperform the global algorithm.

Regarding the *improved operator redistribution* and the *improved selective operator exchange* algorithms, both of them exhibited a superior performance in comparison to COR-BAL, despite their greater overhead due to the increased load movement.

At last, the correlation based algorithms were able to outperform the other algorithms (RAND and LLF) under bursty workloads, with a much smaller increase in the end-to-end delay.

## 2.5 Load Distribution for Distributed Stream Processing

Written by the main author of the paper that describes the load balancing in the Borealis system [34], this paper [33] describes a load balancer that, as opposite to the work on Borealis, is descentralized and decisions depend only on local information.

At a certain level, this solution is similar to the works of Ahmad and Çetintemel [2] and Kumar *et al.* [25]. All of them share the idea of partitioning the overlay network into sub-domains with a node selected as the coordinator, altough in this work there is only one level of domains and there can be overlap between domains.

The basic idea behind the domains is to reduce the amount of information exchange by making all nodes in a domain report their status periodically to a listener node. Nodes have a fixed limit on the number of listener nodes they can report to, and they can only request or send load to nodes in its domain.

This load balancer has the goal of minimizing the average end-to-end delay by reducing network transfer delays. Nodes can be in one of four states: under loaded, moderately loaded, heavily loaded and overloaded. When a node gets overloaded it will try to transfer independent query graphs (by partitioning a query graph) to as few nodes as possible from its domain. Note that an overloaded node is not going to accept more work and the goal of a node is to reach the moderately loaded state.

As one of the goals of the algorithm is to keep the number of nodes participating in the query processing at a minimum, an under loaded node can try acquire sub-query graphs

from his neighbors, thus reducing the total number of nodes.

The decisions about load distribution are based on load forecastings that use past load information to predict the future. A few algorithms for forecasting are cited, but the decision of which one to use depends on the application.

Experiments were conducted with a distributed stream processing simulator, comparing the load balancing and load sharing algorithms with the hybrid solution proposed in the paper. The load balancing algorithm had the worse end-to-end latency during the whole simulation time, while the load sharing and hybrid algorithms showed good static performance. After a warm up period both of them converge to initial distributions, with the hybrid algorithm presenting the lower end-to-end delays.

## 2.6 ACES: Adaptive Control of Extreme-Scale stream processing systems

ACES [3] is a two-tiered scheduler that aims to maximize the *weighted throughput* (each output stream has a weight representing their value), reduce the *end-to-end latency* and *wasted processing*, and keep a stable operation.

In this context a stable operation means that the input, output and processing rates of all operators are stable, managing the size of the buffers in order to prevent them from filling up, which could increase the end-to-end latency and force upstream operators to slow down their processing rate or even pause. With a stable buffer size operators can take advantage of batching, reducing the overhead of context switching and cache misses.

The first tier of this solution is the global optimization. It assigns operators to nodes and determines the allocation of resources through an algorithm that tries to maximize the weighted throughput of the processing graph, based on expected input stream rates. This algorithm is executed in the beginning of the computation and periodically to adapt to workload changes.

Each node has a resource controller (they compose the second tier) that receives the resource allocation targets, monitors the processing rate, input rate and size of buffers, in order to control the rate of upstream operators and keep the system stabilized. The desired input rate of the upstream operators is determined by the allocation algorithm, while the control of the processing rate is performed by the CPU scheduling algorithm, based on feedback from downstream operators. Both algorithms are distributed.

The performance of the ACES scheduler is evaluated in a simulated environment and compared to two traditional scheduling approaches: UDP, which sends tuples to downstream operators regardless of its buffer size, and in case of the buffer being full, incoming tuples are dropped; and Lock-Step, which sends tuples to the downstream operators only if their buffers are not full, in such case the operator will sleep until the buffer has space available.

Results show that the ACES approach is able to deliver a better trade-off between latency and throughput than the Lock-Step approach because it manages to keep the size of the buffers stable. The ACES approach also presented a smaller and smoother increase in the latency in the face of increasing input rates as compared to the other two approaches.

## 2.7 Control-Based Scheduling in a Distributed Stream Processing System

This paper by Khorlin and Chandy [23] presents two algorithms for scheduling distributed stream processing systems.

The first algorithm is based on the processor sharing scheme, where each stream $i$ receives a share $p_i \in [0, 1]$ of the processor. The actual percentage given to a stream only takes into account those whose queues are not empty ($m$), an is calculate by

$$\omega_i = \frac{p_i}{\sum_{j=0}^{m} p_j}$$

. To determine the values of $p_1, ..., p_n$ in such a way that the cost is minimized, the algorithm is represented by a continuous-time Markov process and the arrival processes for each stream is an independent Poisson process with service times as independent exponential random variables.

The states of the process are determined by the queue size of each stream and are used in the prediction of the impact of parameter values in the scheduling for each server. Based on these predictions the scheduler uses a genetic algorithm to optimize the parameters (*i.e.* arrival and service rates) and constantly monitors them. Once they change by more than a threshold, a re-optimization process is started.

The second algorithm, marginal cost-based scheduling, is distributed and based on economic concepts. The decision of which tuples to process is based on the calculation of the benefit of processing a tuple in the server. The tuple with the highest payoff will be selected. The calculus of the marginal cost takes into account the cost of delaying a tuple in the head of a queue, the size of each queue and the downstream delay, received from a downstream server. It's worth noting that the current feedback used by the algorithm does not take into account accurate queue delays.

Experiments in a single-server environment comparing the proposed algorithms with a FIFO approach showed that the processor sharing algorithm was slightly superior to the marginal cost-based algorithms. In a multi-server environment (with two servers), however, the marginal cost algorithm won. Specifically, the best algorithm was the one with feedback control, because it prevented the first server from delaying tuples for too much time, which would make the second server reach its maximum cost.

## 2.8 Network-Aware Operator Placement for Stream-Processing Systems

The paper by Pietzuch *et al.* [27] describes SBON (*Stream-Based Operator Network*), a layer between a DSPS and the physical network that manages the placement of operators in a descentralized fashion (and without global knowledge of the system), while reducing the network usage. The goal of the solution is to optimize the *cost space*, a metric that measures the cost of routing data between nodes.

The SBON layer is in charge of the instantiation and destruction of operators in the physical nodes, creating and destroying links between operators and migration of operators to other nodes. The decision of when to migrate an operator, however, falls to the DSPS, the SBON layer only monitors the performance of the node, manges the cost space and advises the DSPS when it should migrate an operator.

The proposed operator placement algorithm relies on two mechanisms: *cost space* and

*relaxation placement*. The *cost space* is a $d$-dimensional metric space that can be used to estimate the cost of routing data between two nodes through the Euclidean distance. In practice any other metric, such as load, availability, bandwidth or memory can be used. By using the *cost space* a node can estimate the cost to send data to another node without having probed it directly. To do this estimation, a node has only to probe a small subset of nodes periodically, exchanging information with the probed nodes.

With the *relaxation placement* algorithm each operator is placed in the latency dimensions of the cost space using a descentralized technique called *spring relaxation* and then they are mapped to the closest physical node in the cost space.

The relaxation placement has a model similar to the processing graph, with operators represented by massless bodies and operator links (or edges) as springs. The goal of the algorithm is to minimize the potential energies stored in the springs

$$\arg\min_{\overrightarrow{s}_i} \sum_i E_i = \sum_i \overrightarrow{F}_i \overrightarrow{s}_i = \sum_i \frac{1}{2} k_i \overrightarrow{s}_i^2$$

where $\overrightarrow{F}_i$ is the average force experienced by a spring $i$ and is solved by $\overrightarrow{F}_i = \frac{1}{2} k_i \overrightarrow{s}_i$. To minimize the network usage $k_i$, which is the spring constant, it is set to the data rate tranfered through that link $k_l = DR(l)$, and $\overrightarrow{s}_i$, which is the extension vector, to the latency $s_l = Lat(l)$.

By having *pinned* operators, *i.e.* with a fixed location, with the relaxation placement *unpinned* nodes can be brought closer to the pinned nodes, thus reducing the network usage (see Figure 2.1 for an example).

**Algorithm 1** Relaxation Algorithm

1: **repeat**
2: $\quad \overrightarrow{F} \leftarrow \overrightarrow{0}$
3: $\quad$ **for each** $S_{parent}$ **in** $Parents(S)$ **do**
4: $\quad\quad \overrightarrow{F} \leftarrow \overrightarrow{F} + (\overrightarrow{S} - \overrightarrow{S}_{parent}) \times DR(S, S_{parent})$
5: $\quad$ **end for**
6: $\quad$ **for each** $S_{child}$ **in** $Children(S)$ **do**
7: $\quad\quad \overrightarrow{F} \leftarrow \overrightarrow{F} + (\overrightarrow{S} - \overrightarrow{S}_{child}) \times DR(S_{child}, S)$
8: $\quad\quad \overrightarrow{S} \leftarrow \overrightarrow{S} + \overrightarrow{F} \times \delta$
9: $\quad$ **end for**
10: **until** $\left\| \overrightarrow{F} \right\| < F_t$

The Algorithm 1 describes the first step in the operator placement process, which is the virtual placement of operators in the cost space. $\overrightarrow{S}$ corresponds to the current coordinate in the cost space for an unpinned operator. The unpinned operator receives a provisory location close to the origin coordinate. The force $\overrightarrow{F}$ experienced by the operator is calculated by following all his links to parent and child operators and retrieving their coordinates.

The $\delta$ factor is set to smooth the movement of the operator through the cost space, and $F_t$ is the stop condition, which determines the desired precision for the placement of the operator.

The second step is the mapping of the coordinates of the operators from the cost space to physical nodes. It begins by doing a $k$-nearest neighbor search for nodes near the operator coordinates, contacting those nodes to know their operators and resources,

sorting them by distance from the operator's coordinates, iterating over the list to see if one of them is already running the operator, and if not, returns the nearest node that meets the operator's requirements.

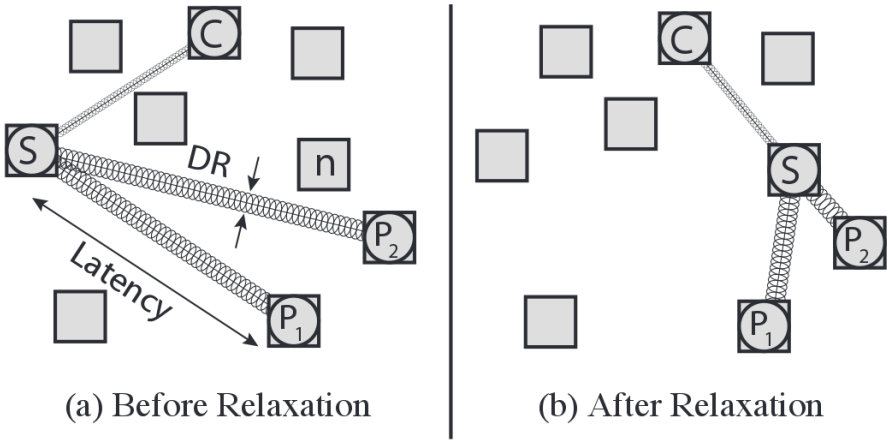


Figure 2.1: Example of Relaxation [27]

Periodically, at each node, a local placement optimizer re-evaluates the placement of operators running locally with the same process described earlier. The difference is that the second step is only carried out if the operator's new coordinate is displaced from the original coordinate by more than a threshold. If it is, the new placement is calculated and the operator is migrated.

The authors evaluated their solution by simulation and experimentation in PlanetLab. Regarding the network usage and delay, the *relaxation* algorithm was compared with other five algorithms in a discrete-event simulator. One of those algorithms compared is the *optimal*, as he does an exhaustive search for the optimal solution, which is used as baseline for comparison. The other algorithms are *IP multicast*, *producer*, *consumer* and *random*. Results showed that the *relaxation* algorithm had the better performance, with an increase of network usage of 15% and delay of 24% in comparison with the optimal solution.

With the experiment in PlanetLab, the goal was to observe the effect of operator migration in the network usage and delay, using a pair of queries for the experiment. With migration of operators enabled a decrease in network usage was observed for 75% of the query pairs, the usage of the total network capacity dropped 16.9% and the aggregate query delay reduced 10.5%.

## 2.9 SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems

SODA [32] is the scheduler employed by the distributed stream processing system called System S, developed by IBM. The scheduler has the goal of maximizing a utility-theoretic function based on the importance of the stream at the end of the processing graph.

It also aims at balancing the allocation of resources to operators, deciding if is better to allocate two operators in the same node or to split an operator among two nodes, for example. And since System S can accept multiple processing graphs, SODA may refuse to accept new applications if it finds out that there's not enough resources for it.

SODA is a dynamic scheduler, and so it has to periodically recalculate the schedulings for the running applications as well as for the new ones, if they are accepted. A period is called an *epoch*. At the beginning of an *epoch*, the scheduler receives as input the current state of the system and during the whole *epoch* it computes the new scheduling. The computation is divided in four optimizations modules, with the first two composing the *macro model*, while the last two compose the *micro model*.

The *macro model* chooses the applications that are going to be executed in the next epoch as well as the candidate nodes that may have operators assigned to it. And the *micro model* chooses the fractional allocations for each operator in the candidate nodes.

Within the *macro model*, the **macroQ** phase does the admission control of applications, chooses the right dataflow for the applications and the required processing power for the operators. This phase is treated as a resource allocation problem (RAP), solved in its discrete version by the *Non-Serial Dynamic Programming* (NSDP) scheme. This scheme is performed for each operator and then over all operators. The result is the optimal allocation values for each operator.

**MacroW** receives the result from **macroQ** as input and has the goal of finding balanced allocations of operators to nodes, respecting constraints such as memory, maximum number of operators per node, maximum number of instances of an operator, and fixed operators. To balance the operators among the nodes the solution in this phase takes into account the average and maximum projected utilization of the processing power, the bandwidth of the network link and the utilization of the network interface. The actual solution is modeled as a mixed-integer optimization program and solved by the proprietary software CPLEX.

**MicroQ** does a revision on the decisions made by **microW** now that the candidate nodes have been selected. And **macroW** computes the fractional allocations for each operator in the candidate nodes with a technique from the network flow literature, with a directed graph composed of nodes as under-allocated operators at the left, processing nodes in the middle and the over-allocated operators in the right.

Experiments were conducted with two different applications in System S: LSD, a data mining application; and DAC, a reference application for analysis and monitoring. The SODA solution is compared with a *random* scheduler that assigns operators uniformly to nodes; a *round-robin* that assigns operators sequentially to the nodes with the least number of assigned operators; and the *expert*, that is the scheduling plan devised by the developers of the LSD and DAC applications.

The results showed that for both applications SODA was able to perform as good or better than the *expert* scheduling plans, using fewer nodes and network capacity.

One aspect that is not covered in the SODA scheduler is the occurrence of bursts, something that can be problematic since new schedulings occurr at fixed periods of time, while bursts can occurr at any time.

## 2.10 Adaptive Component Composition and Load Balancing for Distributed Stream Processing Applications

Repantis, Drougas and Kalogeraki [28] propose a operator placement and load balancing mechanism for large scale systems that satisfies the application's QoS requirements, while adapting to changes. The solution assumes that nodes in the overlay network are organized in groups, in a similar way of Ahmad and Çetintemel [2], Kumar *et al.* [25] and Zing [33]. Each group also has a manager that has information about each member

of the group.

The load distributuion algorithm is descentralized and doesn't require global knowledge of the system. To measure the degree of uniformity of the load distribution, the authors use the *fairness index* [14, 20]

$$\mathcal{F}(\bar{l}) = \frac{(\sum_{i=1}^{n} l_i)^2}{n \cdot \sum_{i=1}^{n} l_i^2}$$

, where $n$ is the number of nodes in the system, $l_i$ the load at the node $i$ and $0 \leq \mathcal{F}(\bar{l}) \leq 1$.

In contrast to the majority of the algorithms presented here, this one has the constraint that nodes can only offer some components of the application, which is a characteristic common to sensor networks. The placement of operators to nodes is done separately for each group of nodes, with all the possible placement plans calculated. Each placement plan is evaluated to determine if it meets the QoS requirements of the application. Among the placement plans that meet the requirements, the one with the highest fairness index is selected and deployed.

Calculating all the possible plans can take up to $O(n^5)$. To reduce the running time, the authors propose three pruning heuristics. The first heuristic removes the placement plans that do not meet the QoS requirements. The second heuristic prunes placement plans that have loops in the instances of operators. And the third heuristic keeps searching for a solution until one with a fairness index equal or greater than a threshold is found.

The mechanism for adaptation to changes in load or application's requirements runs independently in each node. Since the solution doesn't cover load shedding to peer nodes, decrease the load in a node implicates in the reduction of the service quality. This process can be also triggered when the receiver of the application's result (*i.e.* a sink) gives a feedback to the node, demanding a better QoS or providing an acceptable quality interval.

Under a simulated environment, the *fair* algorithm was compared with a *random*, a *greedy*, an *optimal* and an *exhaustive* allocation algorithm. In the first set of experiments the fair algorithm was able to deliver a higher bitrate than the greedy and random algorithms. The observed end-to-end delay showed that the fair algorithm had the closer results to the optimal algorithm. The fair algorithm is also the only one able do tune the quality levels in order do adapt to load changes, while the other algorithms keep a constant quality level at the price of losing data.

Under different loads, the experiments indicated that the fair algorithm was able to deliver end-to-end delays almost as low as the optimal algorithm. The fair algorithm is, however, able to minimize the number of data units that miss their deadlines, presenting better results than all the other algorithms. And the fair algorithm is able to evenly distribute the load among the nodes as well as the optimal solution.

## 2.11 Operator Placement with QoS Constraints for Distributed Stream Processing

Huang *et al.* [19] tackle the operator placement in a distributed stream processing system as a optimization problem constrained by throughput and end-to-end delay, with the network usage as the optimization objective. One of the differences from previous works is that they consider the end-to-end delay to be composed not only of the network delay, but also the processing delay.

Their solution proposes a measure called *Optimization Power* (OP), which determines the fitness of an operator $o$ in a given node $n_k$. The greater the value of OP the more the residual CPU capacity $rr_{cpu}^{n_j}$ available at the node $n_j$, and the smaller the processing delay and network usage.

The scheduling algorithm, called *Optimization Power-based* (OPB) iteratively searches for nodes to place operators with the goal of reducing network usage and satisfying the QoS requirements. To search for nodes it relies on a *Resource Discovery Service* (RDS), using as search parameter the residual capacity of the nodes. The larger the contribution accepted for the processing delay in the total end-to-end delay, the smaller will be the cost of the search. To further reduce the overhead introduced by the search of nodes, it is possible to cache a result and reuse it for the next operator, with a new search being triggered only when the cached results doesn't meet the search parameters.

For each node in the result the OP is calculated for the operators, and the ones with the highest scores are checked to see if they have enough resources for hosting the operator and if the end-to-end delay from the source nodes (those who host the source streams) to the current node are below a certain threshold. The algorithm is iterated until the network usage remains stable for a certain number of times.

Experiments were conducted in a simulation using traces from 240 nodes from PlanetLab with network delays for every pair of them and network coordinates generated with the Vivaldi algorithm. The OPB algorithm was compared with SBON [27]; MIN-DELAY, which does a global search for every operator and selects the node that introduces the least delay (the sum of all processing and network delays from the sources in source until the current node); and the RANDOM which, as the name suggests, randomly assigns operators to nodes.

Regarding network usage, both MIN-DELAY and OPB show almost the same best results with the least input rate. As the input rate is increased the SBON algorithm gets closer to MIN-DELAY and OPB, but OPB is still the one with the least network usage. And the same happens with the end-to-end delay, with OPB showing a delay under 22 in 95% of the placements, followed by MIN-DELAY, SBON and RANDOM.

And the measurement of the success rate of those algorithms (how many placements met the QoS requirements) also showed that OPB was the best algorithm of them.

## 2.12 Managing Parallelism for Stream Processing in the Cloud

Backman, Fonseca and Çetintemel [5] propose a scheduler oriented toward reducing end-to-end latency in a highly parallel distributed stream processing system.

To balance the load among the nodes, this work uses three strategies based on bin-packing algorithms, where workload partitions have a height assigned, proportional to the time required for their processing, and are packed into bins, represented by the nodes. The goal of the algorithm is to move partitions between bins in order to minimize the height of the tallest bin.

The global strategy is centralized and is run only once. It packs together all the operator workloads to all nodes. Whereas the local strategy packs the workload of each operator separately, providing more diversification and encouraging a node to receive partitions from all the operators. The last strategy, tiered, packs together the workload of operators with similar priorities, allowing for better load balancing opportunities.

The partition granularity for each operator and the assignment of partitions to nodes is done via a heuristic seach algorithm that receives as input a set of partitioning configu-

rations generated by a simulator.

Specifically, the search is done via a genetic algorithm that (1) receives a population of randomly generated configurations that are subsequently ranked by their latencies; (2) the ones with the poorest performance are pruned off and (3) the best ones are breed to increase the population; again, (4) the best ones are cloned and mutated to increase the population and (5) new candidates are randomly generated to restore the original population size. The algorithm is repeated until the stop criterion has been reached.

Experiments with the three bin-packing streategies applied to a image processing application and varying the number of partitions per operator showed that the local and tiered strategies presented similar latencies, while the global strategy had the higher latencies for all number of partitions, except one.

Regarding the genetic algorithm, given a simple application composed of ten chained identical operators, it was able to come really close (within 2.5 milliseconds) of the optimal plan, although it took at least 5 seconds for it to be 5% from the optimal plan.

## 2.13 Adaptive Online Scheduling in Storm

The work of Aniello, Baldoni and Querzoni [4] proposes two scheduling algorithms (*offline* and *online*) for the Apache Storm system. The basic idea is to detect hot-spots, which are edges in the processing graph that receive a large amount of tuples, and assign these areas to fast interconnections or even to the same node.

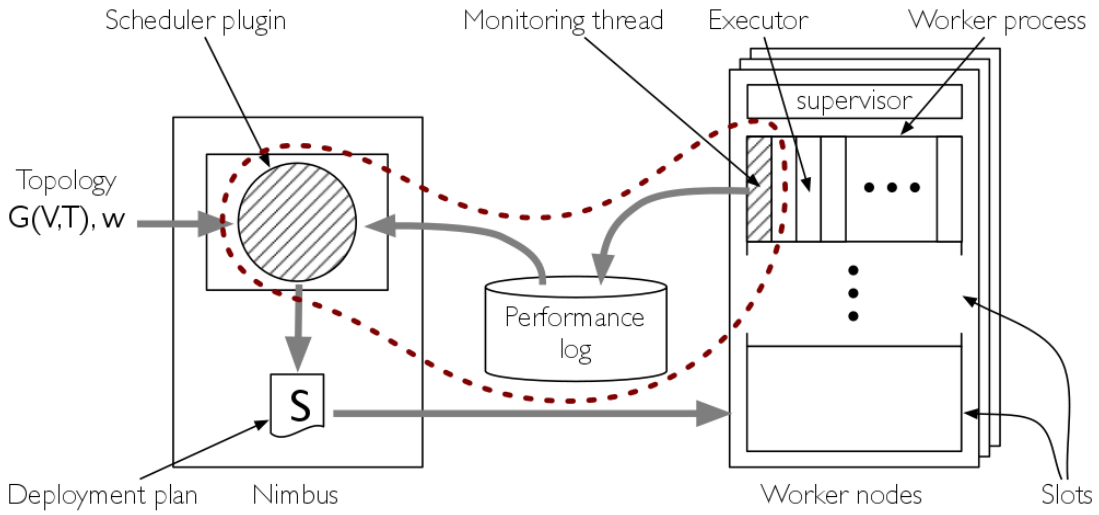


Figure 2.2: Architecture of Storm, with a dashed red line around the components added by the online scheduler [4]

The *offline* scheduler is a heuristic algorithm that only considers the processing graph in the decision making process. It means that neither the load, and consequently the memory and CPU constraints, nor the type of tuple partitioning are considered. The scheduler also only considers acyclic processing graphs (excluding some online machine learning algorithms) and as such it is able to put a partial order in the operators of the graph.

The offline algorithm iterates over the partially ordered operators, placing each one of them in the *slots* that already contain upstream operators that feed the current one. These

slots are then assigned to nodes[2] in a round-robin fashion. To ensure that empty slots receive operators, the scheduler has a threshold variable that can be tuned to force empty slots to be used once it has been exceeded.

The main goal of the *online* scheduler is to reduce the traffic between nodes and slots. To do so the scheduler monitors the output rate of each instance of the operators[3], aggregate the results to have the total traffic of each node in the cluster, computes a new scheduling and the traffic generated with the current traffic, and if the reduction is greater than a certain percentage, the new scheduling is deployed.

The algorithm also monitors the CPU usage of each operator instance to avoid overloads. It is measured in Hz so that the algorithm can calculate how much of the CPU an instance would require if it were moved to a node with a processor with a different speed.

As the operator placement problem is NP-complete, the online scheduler employs a greedy algorithm that works in two phases. In the first phase pairs of instances are interated in descending order of traffic exchange and if none of the instances have been assigned, they go to the least loaded worker. If one of them have already been assigned, the least loaded worker is grouped with the worker that has the instance already assigned. The second phase works in the same way, but now with pairs of workers that get assigned to nodes.

Experiments were conducted with a synthetic application and another one described in the Grand Challenge of DEBS 2013. In both cases the offline and online schedulers showed lower latencies in comparison with the default scheduler. It is noticeable that the online scheduler has a warming period in which his performance is similar to the default scheduler (as the online scheduler works similarly in the beginning), but once it has gathered enough performance measurements to do a re-scheduling the performance is improved. In the second experiment this improvement in the latency is between 20% and 30%.

## 2.14 Network-Aware Workload Scheduling for Scalable Linked Data Stream Processing

The work of Fischer, Scharrenbach and Bernstein [16] proposes the use of well-known graph-partitioning algorithms for workload scheduling in a distributed stream processing system, while minimizing network usage.

Their solution does not cover the placement of operators into nodes, only the way the workload is partitioned among instances of operators. The algorithm used for the partitioning of the graph is METIS [22], configured to create partitions of equal size, optimized for total communication volume.

They performed experiments with two different applications, evaluating their solution with two scenarios: a *oracle* optimizer that has the knowledge about the number of messages that's going to flow along the channels; and a *learning* optimizer that observes the traffic in the channels for a period of time before making any decisions. Their solution was compared with a scheduler that uniformly distributes the load among the nodes.

Results showed that their solution was able to reduce the network usage. In the first application, network usage is kept to a minimum since the parallelized operator, after partitioning of workload, don't need to communicate any further. In the second application, however, there is a join operator that requires communication between nodes, but even so

[2] One node can receive more than one slot

[3] In Storm an instance of an operator is called an executor

the graph-partitioning solution was able to reduce the network usage by more than 40% and in most cases the scenario with the oracle had better results.

## 2.15 Comparison

Table 2.1 presents a comparison between the operator placement algorithms described in the previous sections. Each algorithm is classified by its **type**, with *static* for those that only do the initial placement of operators and *dynamic* for those that somehow adapt to changes in the system; **location** in the sense of where the algorithm is executed, which can be *distributed*, *centralized* or a *hybrid* of both; whether it supports only *query plan partitioning* or *data stream partitioning*.

The classification of the solutions follows the taxonomy of Casavant and Kuhl [8]. As the operator placement problem is NP-complete [2, 19], all algorithms presented here fall in the *suboptimal* category. The *details* column gives a short description of the algorithm, such as the underlying algorithms or theories utilized. The last column gives the complexity of the algorithm, if available in the paper.

Some papers actually present more than one algorithm, they are distinguished in the table by a number between parentheses that corresponds to the order of appearance in the description given here.

| Algorithm | Type | Location | Partitioning | Solution | Details | Complexity |
|---|---|---|---|---|---|---|
| Medusa [13, 6, 7] | Dynamic | Distributed | Query plan/Data stream | Cooperative, suboptimal, heuristic | agoric system | time: $O(N + C)$ (worst w/ notification), $O(NC)$ (worst w/o notification), $O(1)$ (best)<br>messages: $O(NC)$ (worst w/ notification), $O(N^2C)$ (worst w/o notification), $O(N)$ (best) |
| Kumar *et al.* [25] | Dynamic | Hybrid | Data stream | Cooperative, suboptimal, heuristic | network and resource aware | N/A |
| Ahmad and Çetintemel [2] | Dynamic | Distributed | Query plan/Data stream | Cooperative, suboptimal, heuristic | network-aware | computational: $O((d + min(\lfloor d^{h-j} \rfloor, \frac{d^h-k}{d-1})) \times \frac{d^h-1}{d-1})$<br>messages: $O(d^h + d^{h(h-1)/t})$ |
| Borealis [34] | Static and Dynamic | Centralized | Query plan | suboptimal, heuristic | | two-way: $O(m^2k)$<br>global: $O(m^2n^3k$<br>correlation improvement: $O(n^3k + n^4 + m^2n^2k)$ |
| Xing [33] | Dynamic | Distributed | Query plan | | load forecasting | N/A |
| ACES [3] | Static and Dynamic | Hybrid | N/A | Cooperative, suboptimal, heuristic | flow control and CPU scheduling | N/A |
| Khorlin and Chandy [23] | Dynamic | Centralized (1), Distributed (2) | N/A | Cooperative (2), suboptimal (1, 2), approximate (1), heuristic (2) | stream input prediction (1), economic concepts (2) | N/A |
| Aniello, Baldoni and Querzoni [4] | Static (1), Dynamic (2) | Centralized | Query plan | suboptimal, heuristic | hot-spot detection, greedy algorithm (2) | N/A |
| Backman, Fonseca and Çetintemel [5] | Dynamic | Centralized | Data stream | suboptimal, heuristic | bit-packing and genetic algorithms | N/A |
| Huang *et al.* [19] | Dynamic | Centralized | Query plan | suboptimal, heuristic | optimization power algorithm | N/A |
| Fischer, Scharrenbach and Bernstein [16] | Static | Centralized | Data stream | N/A | graph-partitioning | N/A |
| SODA [32] | Dynamic (epoch-based) | Centralized | Data stream | suboptimal, heuristic | admission control | N/A |
| SBON [27] | Dynamic | Distributed | Query plan | Non-Cooperative, suboptimal, approximate | cost space and spring relaxation | N/A |
| Repantis, Drougas and Kalogeraki [28] | Dynamic | Distributed | Query plan | suboptimal, approximate/heuristic | fairness index | w/o pruning: $O(n^5)$ |

Table 2.1: Comparison of operator placement algorithms

# 3 FINAL CONSIDERATIONS

The first schedulers for DSPSs were conceived for very large networks, with high latency costs and an heterogenous environment. In such a scenario they had to be aware of the network topology in order to avoid paths with higher costs.

More recently, as the number of applications for a DSPS increased, as well as the input rates required to be processed, DSPSs moved to more homogeneous environments, such as *clusters*. To take advantage of all this computational power, operators were replicated in multiple nodes in order to enable a higher level of parallelism.

Even with this shift, reducing the network usage continues to be one of the goals in the optimization of an operator placement.

Another optimization that have been used for some time is the operator reutilization, as multiple applications may have some common paths in their processing graphs, they can share these paths instead of doing duplicated work.

Perhaps one of the biggest challenges for DSPS schedulers is the management of bursts. The most straightforward solution is overprovisioning, *i.e.* allocating more resources than the necessary. Unnecessary to say that it is not the most cost effective solution. Some works try to tackle the problem by predicting the input rate, which can be hard since each application may have a different pattern of input rate or no pattern at all.

Vu, Kalogeraki and Drougas [31] handle the issue by delaying the processing of bursty input streams. But delaying the processing of a data stream only increases the end-to-end delay of the application, which can be unacceptable for certain applications. In a previous work, Drougas and Kalogeraki [15] approach the problem by creating rate allocations plans *offline* and in the onset of a burst the best suited plan is deployed, accomodating the burst on time.

With the emergence of *cloud computing* as an environment for running data analysis tools [26, 18], it becomes necessary to devise techniques that are aware of the constraints and advantages (*i.e.* elasticity) of such environment.

# REFERENCES


[1] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., CETINTEMEL, U., CHERNIACK, M., HWANG, J.-H., LINDNER, W., MASKEY, A., RASIN, A., RYVKINA, E., ET AL. The design of the borealis stream processing engine. In *CIDR* (2005), vol. 5, pp. 277–289.

[2] AHMAD, Y., AND ÇETINTEMEL, U. Network-aware query processing for stream-based applications. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30* (2004), VLDB Endowment, pp. 456–467.

[3] AMINI, L., JAIN, N., SEHGAL, A., SILBER, J., AND VERSCHEURE, O. Adaptive control of extreme-scale stream processing systems. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on* (2006), IEEE, pp. 71–71.

[4] ANIELLO, L., BALDONI, R., AND QUERZONI, L. Adaptive online scheduling in storm.

[5] BACKMAN, N., FONSECA, R., AND ÇETINTEMEL, U. Managing parallelism for stream processing in the cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing* (2012), ACM, p. 1.

[6] BALAZINSKA, M., BALAKRISHNAN, H., AND STONEBRAKER, M. Contract-based load management in federated distributed systems. In *NSDI* (2004), vol. 4, pp. 15–15.

[7] BALAZINSKA, M., BALAKRISHNAN, H., AND STONEBRAKER, M. Load management and high availability in the medusa distributed stream processing system. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (2004), ACM, pp. 929–930.

[8] CASAVANT, T. L., AND KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on 14*, 2 (1988), 141–154.

[9] CETINTEMEL, U. The aurora and medusa projects. *Data Engineering 51* (2003), 3.

[10] CHAKRAVARTHY, S., AND JIANG, Q. *Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Advances in database systems. Springer, 2009.

[11] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., AND SHAH, M. A. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR* (2003).

[12] CHAUDHRY, N., SHAW, K., AND ABDELGUERFI, M. *Stream Data Management*. Advances in database systems. Springer, 2006.

[13] CHERNIACK, M., BALAKRISHNAN, H., BALAZINSKA, M., CARNEY, D., CETINTEMEL, U., XING, Y., AND ZDONIK, S. B. Scalable distributed stream processing. In *CIDR* (2003), vol. 3, pp. 257–268.

[14] DROUGAS, Y., AND KALOGERAKI, V. A fair resource allocation algorithm for peer-to-peer overlays. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE* (2005), vol. 4, IEEE, pp. 2853–2858.

[15] DROUGAS, Y., AND KALOGERAKI, V. Accommodating bursts in distributed stream processing systems. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), IEEE, pp. 1–11.

[16] FISCHER, L., SCHARRENBACH, T., AND BERNSTEIN, A. Network-aware workload scheduling for scalable linked data stream processing. *12th International Semantic Web Conference* (2013).

[17] GULISANO, V., JIMENEZ-PERIS, R., PATINO-MARTINEZ, M., SORIENTE, C., AND VALDURIEZ, P. Streamcloud: An elastic and scalable data streaming system. *Parallel and Distributed Systems, IEEE Transactions on 23*, 12 (2012), 2351–2365.

[18] GUNARATHNE, T., WU, T.-L., QIU, J., AND FOX, G. Mapreduce in the clouds for science. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on* (2010), IEEE, pp. 565–572.

[19] HUANG, Y., LUAN, Z., HE, R., AND QIAN, D. Operator placement with qos constraints for distributed stream processing. In *Network and Service Management (CNSM), 2011 7th International Conference on* (2011), IEEE, pp. 1–7.

[20] JAIN, R., CHIU, D.-M., AND HAWE, W. R. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Eastern Research Laboratory, Digital Equipment Corporation, 1984.

[21] JOHNSON, T., MUTHUKRISHNAN, M. S., SHKAPENYUK, V., AND SPATSCHECK, O. Query-aware partitioning for monitoring massive network data streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 1135–1146.

[22] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing 20*, 1 (1998), 359–392.

[23] KHORLIN, A., AND CHANDY, K. M. Control-based scheduling in a distributed stream processing system. In *Services Computing Workshops, 2006. SCW'06. IEEE* (2006), IEEE, pp. 55–64.

[24] KOSSMANN, D. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR) 32*, 4 (2000), 422–469.

[25] KUMAR, V., COOPER, B. F., CAI, Z., EISENHAUER, G., AND SCHWAN, K. Resource-aware distributed stream management using dynamic overlays. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on* (2005), IEEE, pp. 783–792.

[26] MARSTON, S., LI, Z., BANDYOPADHYAY, S., ZHANG, J., AND GHALSASI, A. Cloud computing: The business perspective. *Decision Support Systems 51*, 1 (2011), 176–189.

[27] PIETZUCH, P., LEDLIE, J., SHNEIDMAN, J., ROUSSOPOULOS, M., WELSH, M., AND SELTZER, M. Network-aware operator placement for stream-processing systems. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on* (2006), IEEE, pp. 49–49.

[28] REPANTIS, T., DROUGAS, Y., AND KALOGERAKI, V. Adaptive component composition and load balancing for distributed stream processing applications. *Peer-to-peer networking and applications 2*, 1 (2009), 60–74.

[29] RUNDENSTEINER, E. A., LEI, C., AND GUTTMAN, J. D. Robust distributed stream processing. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)* (Washington, DC, USA, 2013), ICDE '13, IEEE Computer Society, pp. 817–828.

[30] STONEBRAKER, M., ÇETINTEMEL, U., AND ZDONIK, S. The 8 requirements of real-time stream processing. *ACM SIGMOD Record 34*, 4 (2005), 42–47.

[31] VU, D., KALOGERAKI, V., AND DROUGAS, Y. Efficient stream processing in the cloud. In *Quality, Reliability, Security and Robustness in Heterogeneous Networks*. Springer, 2012, pp. 265–281.

[32] WOLF, J., BANSAL, N., HILDRUM, K., PAREKH, S., RAJAN, D., WAGLE, R., WU, K.-L., AND FLEISCHER, L. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware 2008*. Springer, 2008, pp. 306–325.

[33] XING, Y. Load distribution for distributed stream processing. In *Current Trends in Database Technology-EDBT 2004 Workshops* (2005), Springer, pp. 112–120.

[34] XING, Y., ZDONIK, S., AND HWANG, J.-H. Dynamic load distribution in the borealis stream processor. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on* (2005), IEEE, pp. 791–802.