Efficient Application of Cardinality-Aware Partitioning Algorithm on Scale-Out Architectures

Ahmad Diab, Panos K. Chrysanthis
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
ahd23@pitt.edu, panos@cs.pitt.edu

Abstract

The world of database and query execution depended for a long time on sequential processing, due to the limited available hardware. The outcome was satisfying based on available applications, until the amount of data exploded and started to overwhelm our conventional methods of handling data, satisfaction was no longer the general norm.

This was the motivation behind seeking alternative approaches to tackle the massive amount of data, and due to the evolution of hardware architecture, it was possible to develop several methods that can produce results in a satisfying way. Stream processing has made its way as one of the most promising models for the task.

Stream processing in its core is a sequence of data and a series of operations to be performed on each of element of this data, usually in a pipelined manner, trying to take advantage of the optimal available hardware, in order to maximize execution of queries. The nature of stream processing can handle large volumes of data compared to other processing models, allowing continuous and timely results in a decentralized and decoupled infrastructure.

In this work, we concentrate on cardinality-aware partitioning model proposed by [12], we expand this model to efficiently perform on multi-node (scale-out) architectures by introducing latency between nodes in load balancing metric. The target is to deploy this algorithm, that beats state-of-the-art partitioning algorithms, in different multi-node environments with best performance.

Introduction

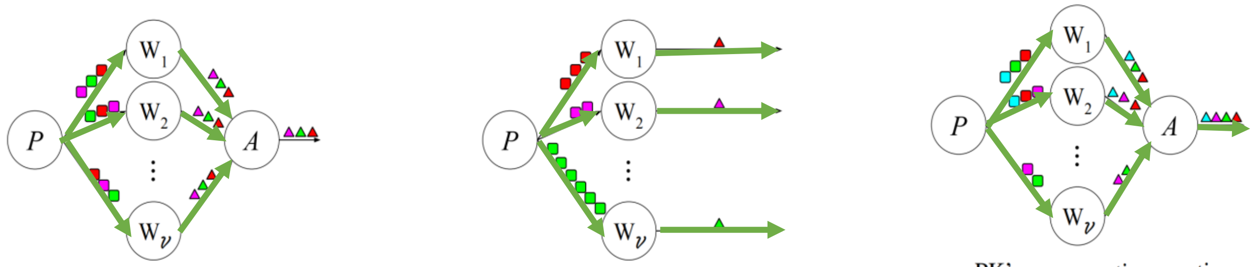
In any of the data processing models, latency is an important factor in load balancing tasks over the available working units. Designing an efficient partitioning algorithm under the consideration of data type and hardware architecture has been an important aspect for many researchers. There has been plenty of research considering Parallel processing of large data [2, 6, 7], their work focused on exploring different techniques to perform queries on a parallel fashion, taking advantage of the new available technologies, their research varies from proposing new methods to process data, to enhancing a conventional way to boost performance. This came after a long wave of single-thread stream processing research [4, 5, 3, 13] which was limited by the type of hardware available at that time. Their models were relatively simple compared to nowadays environments (for example, they didn't have to consider inter-process communications for correct/fast results). One particular concept was Stream Processing, which drove much of scholars' attention in the past couple of years due to its promising performance and dynamic structure. This idea was studied heavily in literature [8, 14, 16], and was explored on both single-node multi-threaded (scale-up) [1], and multi-node (scale-out) [15, 10, 11] architectures.

The work proposed in [12] particularly stands out, it introduces a novel cost model for stream partitioning that considers both imbalance and aggregation to find the best load balancing division among workers. It also proposes a new partitioning algorithm based on the introduced cost model, which, based on their experimental results, beats the state-of-the-art partitioning algorithms performance in various benchmarks.

The work is interesting in its novelty, it creatively proposes a new technique to partition streams based on suggested cost model, it also comprehensively studies the benefit of using aggregation cost in load balancing tasks over workers. However, it does not explore the influence of external factors (communications, latency, etc) between nodes in scale-out environments. Instead, it assumes that any parallel Stream Processing Engines (pSPE) method includes such interventions. This assumption is valid for a newly proposed algorithm that is designed to fit different architectures. In this paper, we extend this work for the most effective application on scale-out architecture, we focus on stretching the novel cost model to include latency between nodes, and extend the proposed novel cardinality-aware partitioning method to produce the best performance in any scale-out environment. For the sake of simplicity, and for the rest of this paper, we will call the existed method CAPA (short for Cardinality Aware Partitioning Algorithm), and our proposed extension as E-CAPA (where E stands for Extended).

Let's first shed a light on some of the most common partitioning algorithms. We start with Shuffle Partitioning – SH (round robin), this technique divides the stream into chunks and sends them to all available worker. As simple as it may sound, this algorithm saves time on dividing initial input only to spend more on aggregation, since the key separation is randomly distributed. The second algorithm is Hash Partitioning – FLD (field), it divides input based on key, sending all portions that has the same key to the same working unit. In ideal cases, this algorithm has a fast performance with low cost in both partitioning and aggregation (since output is ordered at all times), however, it extremely suffers from skewness, which is inevitable in stream processing applications, meaning that some workers will be assigned more work than others based on the key distribution of input, which is usually not uniform. The last algorithm is Partial Key Partitioning – PK, this is considered the state-of-the-art algorithm, it is a tweak on FLD method that attempts to solve the skewness problem, allowing more uniform distribution of work over available units.

CAPA uses cost model to balance the load distribution over workers, where E-CAPA attempts to extend the approach by adding latency between nodes to its calculation, allowing more robust and scalable performance in any environment. Figure 1 demonstrates the difference between the two algorithms.



a: SH's *aggregation* runtime is proportional to $\mathcal{V}$ times the number of distinct groups.

b: FLD doesn't require *aggregation*, but fails to balance load under skewed input.

c: PK's *aggregation* runtime is proportional to M times the number of distinct groups (M is the number of choices).

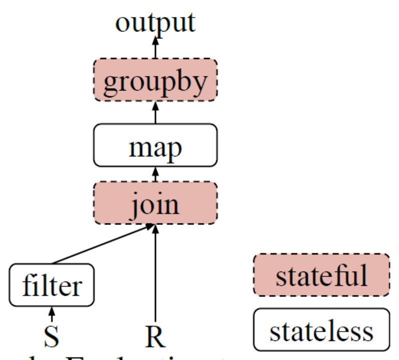
Figure 1: Green arrows show the additional element being considered in cost model, it represents the latency between nodes in scale-out architecture. (figure is taken – and modified – from [12]

Proposed Model

Since this work is an extension to CAPA, we will focus on pSPEs executing stateful operations. This type of operations stores data in some sort of storage, allowing the application to survive service restarts. Scale-out architecture focuses on horizontal growth, allowing the addition of new and different processing units to increase the environment capacity, this is done in a dynamic manner where resources can be added or removed based on availability and the target performance desired for the application.
For the sake of simplicity, and to make our equation and model easier to compare against the original work, we are going to use the same query and setup for the rest of this section. The query we will analyze, and its evaluation tree is shown in Figure 2. It is important to understand the tree structure, the root represents the output, where input streams are in the leaves. Although the model can accommodate multiple streams at the same time, we will demonstrate our work on a single input stream.

```
SELECT R.a, COUNT(*)
FROM R JOIN S
     on R.a = S.b
[ Range 5 minutes
  slide 30 seconds
]
WHERE S.c < 100
GROUP BY R.a
```

a: Input Query $\mathcal{Q}$.



b: Evaluation tree.

Figure 2: The query to be evaluated by the model (and algorithm). This figure was taken from [12], and included here for further demonstration.

a. Node Latency in cost model

Before we move forward in our model, let's talk about the need to include architectural intervention and overhead caused by scale-out environment.

In a horizontal elasticity environment, also known as scale-out architecture, a system is expected to dynamically utilize resources and balance workloads among available nodes in runtime. Different nodes of various natures can be added, which means that the processing power, resources (processor, memory, etc), distance, and connection medium also varies, we will call this "node-latency" for the rest of the paper. These specifications heavily influence the time taken for the same workload to execute. We believe that this overhead should not be neglected, in fact, it can crucial for a well-balanced partitioning algorithm with best performance.

To demonstrate our concept, let us first assume a horizontal environment where all nodes are identical (homogenous network), practically this might be impossible, but we will include the case where node-latency difference is very small between all workers. This setup would be similar to scale-up (vertical) system. However, this is not usually the situation for horizontal scaling, many applications favors this type of parallel paradigm for its property of adding/removing workers on demand, which creates an irregular node-latency throughout the network.

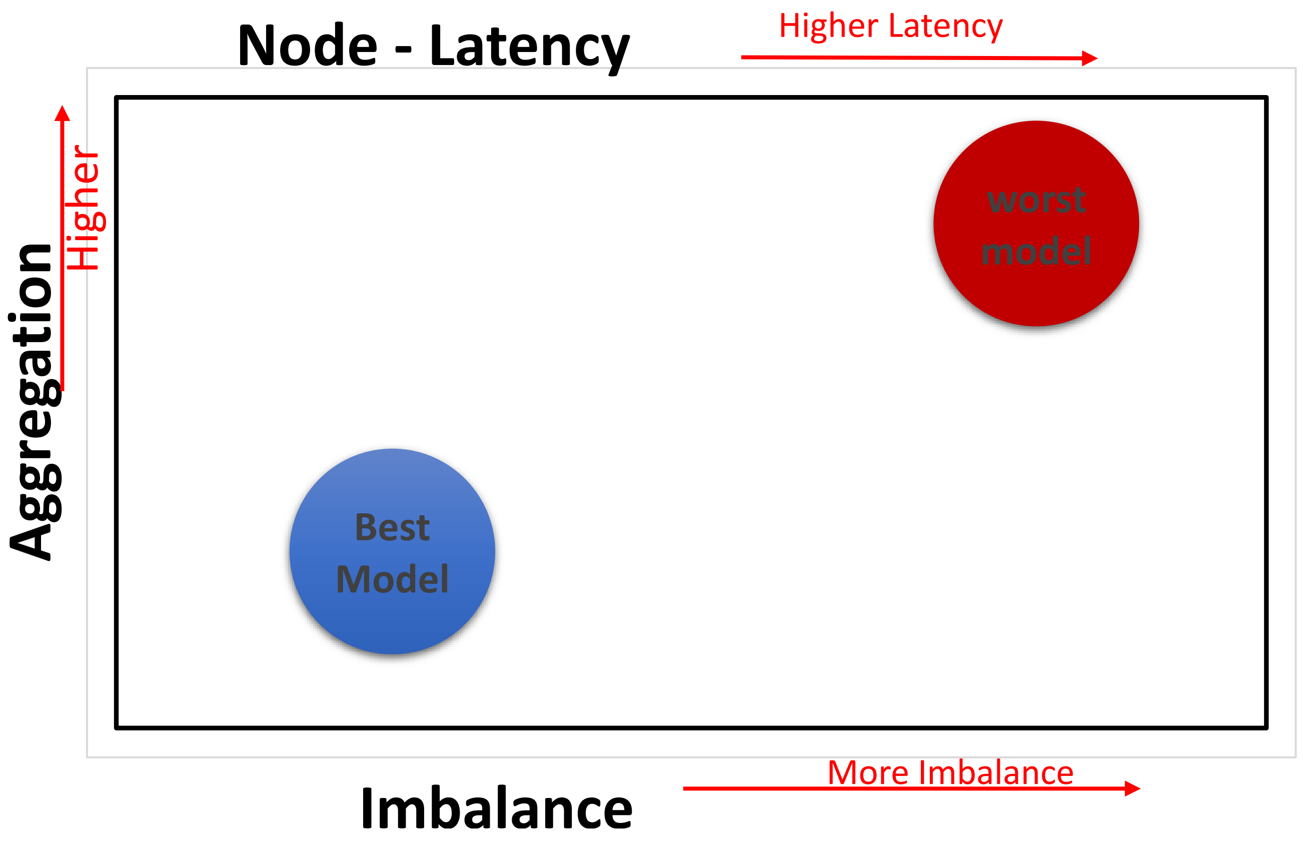


Figure 3: The relationship between Imbalance, Aggregation cost, and Node-Latency cost. The best model (in blue) will attempt to minimize these values where a bad model (in red) will have high values for all these metrics.

Based on the argument mentioned above, a cost model should count not only for Aggregation cost and Imbalance, but also for node-specific properties, which we highlighted as node-latency. A good model would minimize the values for the three metrics, where a bad model can not restrain them from increasing. Figure 3 demonstrates this relationship.

We believe that the study of node-latency is promising to enhance the overall performance, especially after studying the following scenario:

In a simple infrastructure, where we have only two nodes, the original CAPA suggest that we study the input stream to achieve the cardinality-aware concept, this will build a cost estimation based on aggregation cost and imbalance, based on that, the task will be divided on the two workers relatively equally and we will consider the total cost to be 10 time-units. Our model extends the decision making to include a node-aware concept, and to demonstrate the difference, we will say that node A has node-latency of 10 time-units where node B has 2 time-units. Based on these numbers, if the load was distributed equally, node A will take 20 time-units to finish its task where node B will be done after 12 time-units and stay idle until the work is done. But since out extended considers node-latency, the best balance of the work might be where node A is doing only 40% of the load to accommodate the additional latency, where node B will do the rest (60%), and the system will finish the task in about 16 time-units (compared to 20 for the original algorithm).

We understand that the numbers mentioned in the above example are not realistic, but the aim was to show the influence of node-latency in a partitioning algorithm and how it can effect the overall performance.

b. Node-Latency Calculation

As we have seen so far, the scale-out architecture is not always homogeneous, and despite the fact that we can ignore this, as it will effect any partitioning algorithm performed on the network, we try to understand and include this latency in cost model calculations.

Node-specific latency is determined by different factors (hardware resources of node, network type and structure, etc). Calculating each factor with accurate or estimate values was the first initial thought of this work, however, this need extra calculations that will translate to more overhead on already time-hungry task. Instead, we propose to hide this latency complication and represent it by a single variable, $\mathcal{L}_{n\_i}$, where $\mathcal{L}$ stands for Latency, and n is for node. In any environment with changing number of nodes, i can range from $1 < i < N$, which is the max number of nodes at any given time. To calculate this value for all nodes, the system initially sends a small, identical, and independent task to each one of the available workers individually. A separate timer keeps track of the sending and receiving time and register results. Since these tasks are identical in nature, any time difference between nodes is due to its properties. We might need to do this step more than once at the beginning, for example, after adding/removing a new worker in the pool, or after executing an input stream that took more than the anticipated total finishing time. Regardless, the state of node-latency is saved at any point to be included in the cost model and partitioning algorithm.

c. Extended Cost Model

In general, partitioning algorithms aim to divide the work as evenly as possible among available workers, while maintaining low cost of aggregation. This is the case for CAPA (the original work introduced in [12]), they expressed the tuple imbalance introduced by [9] to cover the window aspect of a stream rather than the entire stream. The model turns to a minimizing problem, where the optimal partition is the one that reduces the imbalance of a work portion sent to a worker ($I(P (S_i^w))$), while keeping the aggregation cost of that part less than a maximum threshold.

Our proposed idea stretches this novel model to create a node-aware algorithm. After obtaining node-latency for each worker in the network, $\mathcal{L}_{n\_i}$, and knowing that this value cannot be changed in the system (unless a physical intervention was made by replacing a node or updating its hardware), the new cost model tries to achieve the same minimization goal but with extra parameter to consider. The following equation summarizes the concept:

$$Minimize \{ \mathbf{I}(P (S_i^w)) + \Gamma(S_i^w) + \mathcal{L}_{n\_i} \}$$

This equation allows the algorithm to accommodate external factors that may affect the performance of any applied partitioning method in scale-out architecture. Figure 4 shows a sample query of group-by, it is applied on windowed stream, and the cost model considers the three aspects. Note that the node-latency occurs before and after the partial evaluation. However, in the previous equation we only counted it once, we assumed that $\mathcal{L}_{n\_i}$ represents the total node-latency.

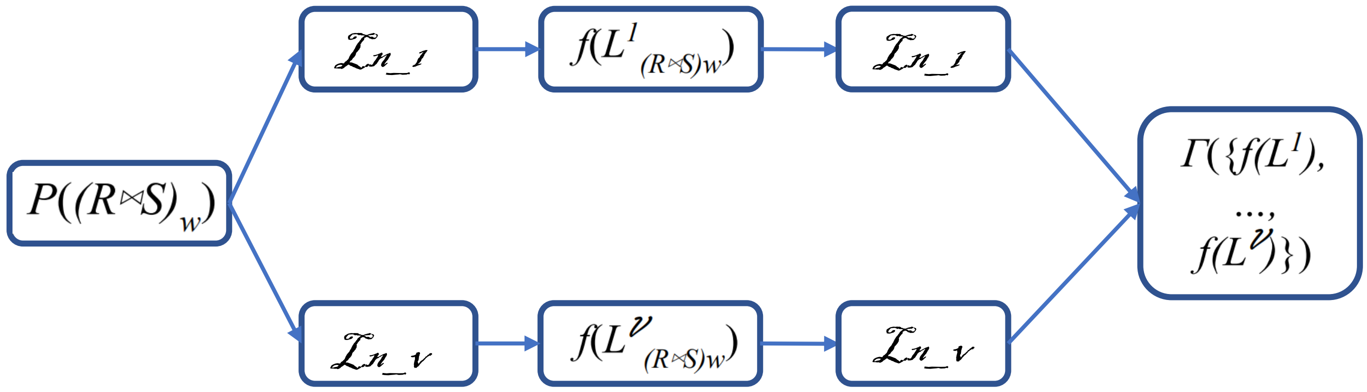


Figure 4: Sample query to demonstrate of including node-latency in cost model

d. Cardinality-Aware, Node-Aware Partitioning Algorithm

The original algorithm is divided into two parts. First is the Partition step, it maintains Cardinality estimations and Tuple Counters in two separated arrays, it also uses hash functions related to the number of available candidates, these values are passed to '*decide*' function, which will pop out the worker to which the work will be sent to.

The second part is the implementation of '*decide*' function. The paper provided four different variation to implement this method. The variations are CM, AM, cAM, and LM. Each provides a different flavor of how to pick the worker that will be assigned the job. AM and cAM will send the task to whichever already contains the key. CM and LM skip this part.
Our proposed model works the same way, which means we can still take advantage of all these decision implementations and figure out which gives the best performance. However, we need to adjust each decision metric for each of the algorithms by the value of node-latency.
For example, CM builds the decision on the estimated cardinality of each candidate *C[c$_i$].estimate()*. When we extend this to a node-aware algorithm, the value should consider $\mathcal{L}_{n\_i}$ as well. Building the final decision based on:

$$l_i = C[c_i].estimate() + \mathcal{L}_{n\_i}$$

The same thing applies for each variation, this way we can take advantage of the fast performance of these algorithms without adding any overhead (a very small computation processing is added, to retrieve and add node-latency for each worker). This assumes that we already obtained node-latency values for all workers at the beginning, which we demonstrated earlier is a small task sent to all nodes. It can be repeated several times based on the nature of the network itself, for example, if we expect a change in transfer rate from time to time, we might need to run the code that calculates node-latency more regularly.

Suggested Experimental Setup

Since the main concentration is to compare the performance of the original method with respect to the newly extension proposed in this work, we suggest performing two types of experiments. The first would be similar to the one mentioned in the original paper, a multi-threaded AWS environment, with the same operating system (Ubuntu 14.04), programming language (C++ 11), and compiler (GCC 4.8.2). The experiment can serve as a base line to measure how much overhead would the proposed extension take over the original algorithm. We already have seen that E-CAPA requires some extra steps to measure the latency in the given environment, there is a possibility that these steps might take time enough to hurt the performance and make it useless. The second experiment, our aim is to establish an environment with different workers connected by different mediums. To achieve this, we propose having 4 workers with varying properties, and attempt to generate (1:2:5:11) ratio of latency among them. This is to ensure that the latency values do not average out, and to create an unusual setting for the original algorithm to deal with. The ratio means that if the first worker W_1 has a latency value of $\mathcal{L}$, then W_2 will have double that value, and W_3 will have (5x $\mathcal{L}$), leaving W_4 with (11x $\mathcal{L}$). This gives some flexibility in how to generate this ratio, the actual properties of workers do not particularly matter in this case.
In both experiments, we can compare the outcome with that of shuffle (SH), hash key (FLD), and Partial Key (PK). This will give us insights of where this newly proposed algorithm sets in performance comparison against some of the most famous partitioning algorithms.

Expected Results

The outcome of the first experiment is some how expected between CAPA and E-CAPA, the additional steps the latter needs to set up the environment will add more overhead. It would be interesting to see how much time the extension takes in comparison to the original method, as well as how it will perform against other methods.

The second experiment result is the more decisive one. It will demonstrate which algorithm performs better on a scale-out environment. Although we expect the suggested extension will enhance the overall running time of the partitioning; we need to study the results closely to confirm such assumption. We also expect that our extension will outperform other traditional partitioning algorithms, since it is the only node-aware approach that handles different nature of workers in the same environment.

Without a proper bench test to judge the performance of the two, our statement cannot be considered correct in the academic world.

Conclusion

Inspired by the work of Cardinality-Aware Partitioning Algorithm (CAPA), we presented and extended version which utilizes any multi-node architecture capacity to guarantee the best possible performance. We extended the proposed cost model to consider latency between node in load distribution over workers, we also expanded the algorithm for scale-out designs to count for interventions between nodes.

Since CAPA already beats state-of-the-art partitioning algorithms, we expect out proposed algorithm to also perform similarly. Although a practical comparison between CAPA and E-CAPA is needed to decide which one outperforms the other; we believe that E-CAPA suits multi-node environments more, since it actually considers the nature of the structure without missing with the core idea of the original approach, and therefore it has a good chance of producing better results.

References

[1] B. Chandramouli, J. Goldstein, M. Barnett, et al. Trill: A high-performance incre- mental query processor for diverse analytics. In PVLDB, pages 401–412, 2015.

[2] B. Lohrmann, D. Warneke, and O. Kao. Massively-Parallel Stream Processing under QoS Constraints with Nephele. In HPDC, 2012.

[3] D. J. Abadi, D. Carney, U. C¸ etintemel, M. Cherniack, et al. Aurora: A new model and architecture for data stream management. VLDBJ, 12(2):120–139, 2003.

[4] H. Zhong, S. A. Lieberman and S. A. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications," 2007 IEEE 13th International Symposium on High Performance Computer Architecture, Scottsdale, AZ, 2007, pp. 25-36.

[5] Huiyang Zhou, "Dual-core execution: building a highly scalable single-thread instruction window," 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05), St. Louis, MO, USA, 2005, pp. 231-242.

[6] J. Fang, R. Zhang, T. Z. Fu et al., "Parallel stream processing against workload skewness and variance" in Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, ACM, pp. 15-26, 2017.

[7] K. G. S. Madsen, Y. Zhou and J. Cao, "Integrative Dynamic Reconfiguration in a Parallel Stream Processing Engine," 2017 IEEE 33rd International Conference on Data Engineering (ICDE), San Diego, CA, 2017, pp. 227-230.

[8] Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S., 2015. Twitter Heron: Stream processing at scale. In: ACM SIGMOD International Conference on Management of Data, SIGMOD '15, ACM, New York, USA, pp. 239–250.

[9] M. A. Nasir, G. Morales, D. Garcia-Sorano, et al. The power of both choices: Practical load balancing for distributed stream processing engines. In ICDE, pages 137–148, 2015.

[10] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In SIGMOD, pages 511–526, 2016.

[11] M. Zaharia, T. Das, H. Li, et al. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In HotCloud, 2012.

[12] Nikos R. Katsipoulakis, Alexandros Labrinidis, Panos K. Chrysanthis, A Holistic View of Stream Partitioning Costs, Proceedings of the VLDB Endowment (PVLDB), 10(11):1286-1297, August 2017.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In CIDR, 2003.

[14] Shukla A, Chaturvedi S, Simmhan Y. RIoTBench: an IoT benchmark for distributed stream processing systems. Concurrence Compute: Practical Exp. 2017;29(21). https://doi.org/10.1002/cpe.4257.

[15] Y. Wu and K. L. Tan. Chronostream: Elastic stateful stream computation in the cloud. In ICDE, pages 723–734, 2015.

[16] Zacheilas N., Zygouras N., Panagiotou N., Kalogeraki V., Gunopulos D. (2016) Dynamic Load Balancing Techniques for Distributed Complex Event Processing Systems. In: Jelasity M., Kalyvianaki E. (eds) Distributed Applications and Interoperable Systems. Lecture Notes in Computer Science, vol 9687. Springer, Cham.