Data Streaming for Appliances

Marta Patiño and Ainhoa Azqueta
Laboratorio de Sistemas Distribuidos, E.T.S. Ingenieros Informáticos Universidad Politécnica de Madrid, Spain

Keywords: Data Stream Processing, NUMA Aware, Appliances.

Abstract: Nowadays many applications require to analyse the continuous flow of data produced by different data sources before the data is stored. Data streaming engines emerged as a solution for processing data on the fly. At the same time, computer architectures have evolved to systems with several interconnected CPUs and Non Uniform Memory Access (NUMA), where the cost of accessing memory from a core depends on how CPUs are interconnected. This paper presents UPM-CEP, a data streaming engine designed to take advantage of NUMA architectures. The preliminary evaluation using Intel HiBench benchmark shows that NUMA aware deployment improves performance.

1 INTRODUCTION

Large companies use mainframes for running their businesses. In general, the main frames are characterized by having a large memory and several cores. The mainframe usually runs the operational database which is part of the core business of companies.

The goal of the CloudDBAppliance project[1] is setting up a European Cloud Appliance that integrates three data management technologies: operational database, analytical engine and real-time data streaming on top of a many-core architecture with hundreds of cores and several Terabytes of RAM provided by Bull.

Servers with several CPUs in a single board, many cores, with non-uniform memory access (NUMA) and 128GB or more are common these days.

Nowadays several data streaming engines (DSE) are available and ready to be used like Flink, Spark Streaming, and Storm among others. These data streaming engines were designed to run on a distributed system made of several computers connected through a network in a LAN in order to scale and process large amount of events per second. However, mainframes although they resemble a distributed architecture they expose a centralized architecture with no network communication and large shared memory. In this paper we present UPM-CEP, a NUMA aware DSE for appliances. UPM-CEP provides a scalable architecture to be deployed on NUMA architectures. To the best of our knowledge, this is the first DSE with this feature. The paper describes the UPM-CEP architecture and preliminary performance results using the Intel HiBench benchmark.

The rest of the paper is organized as follows. Section 0 introduces NUMA architectures. Section 0 presents data streaming engines main features. The architecture of UPM-CEP is presented in Section 0. Section 0 presents the performance evaluation and finally, Section 0 presents some conclusions.

2 NUMA ARCHITECTURES

A NUMA system consists of several connected CPUs, also called nodes or sockets. Each CPU has its own memory that can be accessed faster than the memory attached to other CPUs. Main vendors have implemented this model and connect the CPUs using *QuickPathInterconnect* (QPI, Intel) or other means, like *HyperTransport* in AMD processors. The cores of a CPU have its own L1 and L2 caches and share a L3 cache. In the case of Intel each CPU has a number of QPI links used to connect to other CPUs. Depending on the total number of CPUs the memory of other CPUs is reachable in a single hop or more hops are needed.

[1] *The CloudDBAppliance Project*. http://clouddb.eu

Patiño, M. and Azqueta, A.
Data Streaming for Appliances.
DOI: 10.5220/0007905906720678
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 672-678
ISBN: 978-989-758-365-0

For instance, the Bullion S16 consists of 16 Intel NUMA nodes each of them equipped with 18 cores. Figure 1 shows how the CPUs are connected and Figure 2 shows the memory distance between different NUMA nodes.

The distance represents the relative latency for accessing the memory from one CPU to another one. For instance, a program in node 0 accessing data in its own memory has a cost of 10, while accessing the data in the memory attached to node 1 has a higher cost, 15 (50% overhead). If data is the memory of any of the 14 remaining nodes, the cost is 40. That is, it costs 4 times more than accessing local memory of node 0.

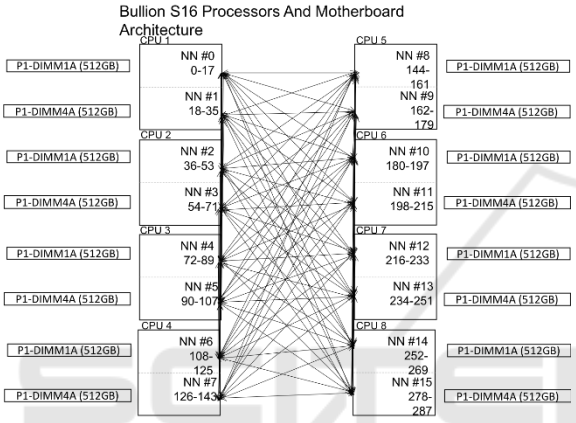

Figure 1: Bullion S16 Architecture.

Bullion S16 Latency Access to Different NUMA nodes

node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0:	10	15	40	40	40	40	40	40	40	40	40	40	40	40	40	40
1:	15	10	40	40	40	40	40	40	40	40	40	40	40	40	40	40
2:	40	40	10	15	40	40	40	40	40	40	40	40	40	40	40	40
3:	40	40	15	10	40	40	40	40	40	40	40	40	40	40	40	40
4:	40	40	40	40	10	15	40	40	40	40	40	40	40	40	40	40
5:	40	40	40	40	15	10	40	40	40	40	40	40	40	40	40	40
6:	40	40	40	40	40	40	10	15	40	40	40	40	40	40	40	40
7:	40	40	40	40	40	40	15	10	40	40	40	40	40	40	40	40
8:	40	40	40	40	40	40	40	40	10	15	40	40	40	40	40	40
9:	40	40	40	40	40	40	40	40	15	10	40	40	40	40	40	40
10:	40	40	40	40	40	40	40	40	40	40	10	15	40	40	40	40
11:	40	40	40	40	40	40	40	40	40	40	15	10	40	40	40	40
12:	40	40	40	40	40	40	40	40	40	40	40	40	10	15	40	40
13:	40	40	40	40	40	40	40	40	40	40	40	40	15	10	40	40
14:	40	40	40	40	40	40	40	40	40	40	40	40	40	40	10	15
15:	40	40	40	40	40	40	40	40	40	40	40	40	40	40	15	10

Figure 2: NUMA distances in the Bullion S16.

Operating systems in general allocate threads on the CPU with lowest usage. Therefore, the threads of a process can be spread across several CPUs and therefore the performance of the application can be affected by *remote access* to the data. Linux systems provide functions to bound threads to a CPU (e.g., *libnuma*) and even tools to bind a process to a CPU (e.g., *numactl*). The *numactl* command also allows to define where the memory is allocated for an application, for instance, on a single CPU, on a set of CPUs or interleaved among a set of CPUs.

3 DATA STREAMING ENGINES

Stream Processing (SP) is a novel paradigm for analysing data in real-time captured from heterogeneous data sources. Instead of storing the data and then process it, the data is processed on the fly, as soon as it is received, or at most a window of data is stored in memory. SP queries are continuous queries run on a (infinite) stream of events. Continuous queries are modeled as graphs where nodes are SP operators and arrows are streams of events. SP operators are computational boxes that process events received over the incoming stream and produce output events on the outgoing streams. SP operators can be either stateless (such as *projection*, *filter*) or stateful, depending on whether they operate on the current event (tuple) or on a set of events (*time window or number of events window*). Several implementations went out to the consumer market from both academy and industry (such as Borealis (Ahmad, 2005), Infosphere (Pu, 2001), Storm (Foundation, 2015), Flink (Fundation, 2014) and StreamCloud (Gulisano, 2012)). Storm and Flink followed a similar approach to the one of StreamCloud in which a continuous query runs in a distributed and parallel way over several machines, which in turn increases the system throughput in terms of number of tuples processed per second. UPM-CEP adds efficiency to this parallel-distributed processing being able to reach higher throughput using less resources. It reduces the inefficiency of the garbage collection by implementing techniques such as object reutilization and takes advantage of the novel Non Uniform Memory Access (NUMA) multicore architectures by minimizing the time spent in context switching of SP threads/processes.

4 UPM-CEP DATA STREAMING ENGINE

UPM-CEP provides a client driver for streaming applications, the JCEPC driver that hides from the applications the complexity of the underlying system. Applications can create and deploy continuous queries using the JCEPC driver as well as register to the source streams and subscribe to output streams of these queries. During the deployment of a streaming

query the JCEPC driver takes care of splitting the query into sub-queries and deploys them in the CEP. Some of those sub-queries can be parallelized. For instance, the query in Figure 3 shows a data streaming query made out 2 input streams and 8 operators. The operators are either stateless (SL) or stateful (SF) operators.

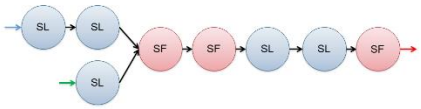

Figure 3: Data streaming query.

UPM-CEP has several stateless and stateful operator implemented and users can create their own customized operators. The stateless operators than can be found are: 1) Map: that allows to select the desired fields from the input tuple and create an output tuple with those fields. 2) Filter: Only the tuples that satisfy a defined condition are sent through the output stream, the rest of tuples are discarded. 3) Demux: sends the input tuples to all the output streams that satisfies the defined conditions and 4) Union: Tuples that arrives to the operator from different input streams are sent to one output stream.

Regarding the stateful operators two window oriented operators are available: 1) Aggregate: group all tuples that are in the time or size window taking into account defined functions executed over the fields of all the tuples. Moreover, tuples can be grouped into different windows if the group by parameter is specified. 2) Join: Correlates tuples from two different input stream. Two time windows are created, one per input stream and when the windows are slide tuples are joined creating one output tuple taking into account a specified predicate.

UPM-CEP partitions queries into subqueries so that, each subquery executes in a different node. Figure 4**Error! Reference source not found.** shows how the previous query is split into four subqueries (SQ1, SQ2, SQ3 and SQ4). The number of subqueries of a given query is defined by the number of stateful operators. All consecutive stateless operators are grouped together in a subquery till a stateful operator is reached. That stateful operator is the first operator of the next subquery. This way of partitioning queries has proven to be efficient in distributed scenarios (Gulisano, 2010). We have applied the same design principles to UPM-CEP although it is not a distributed setup, the same principles apply minimizing the communication across NUMA nodes in this case and keeping the same semantics a centralized system will provide.

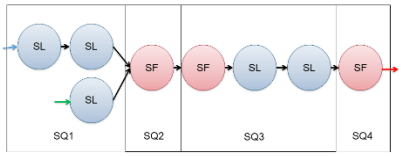

Figure 4: Query partitioning.

Subqueries can be parallelized in order to increase the throughput. Each instance of a subquery can run in a different core in the same node. Figure 5 shows how subqueries of the previous example could be parallelized. There are 3 instances of SQ1, one instance of SQ2, two instances of SQ3 and three instances of SQ4.

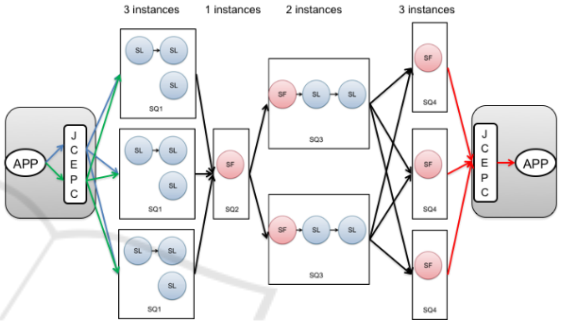

Figure 5: Query parallelization.

The main challenge in query-parallelization is to guarantee that the output of a parallel execution is the same as a centralized one. If we consider a sub-query made by only one operator, this challenge means that the output of a parallel operator must be the same as a centralized operator. On the other hand, window oriented operators require that all tuples that have to be aggregated/correlated together are processed by the same CEP instance. For example, if an *Aggregate* operator computing the total monthly operations of the bank accounts for each client is parallelized over three CEP Instances, it must be ensured that all tuples belonging to the same user account must be processed by the same CEP Instance in order to produce the correct result.

To guarantee the equivalence between centralized and parallel queries, particular attention must be given to the communications among sub-queries. Consider the scenario depicted in Figure 6 where there are two sub-queries, *Sub-query* 1 and *Sub-query* 2, with a parallelization degree of two and three, respectively. If *Sub-query* 2 does not contain any window oriented operator, CEP instances at Sub-query 1 can arbitrary decide to which CEP instance of *Sub-query* 2 send their output tuples. Output tuples of *Sub-query* 1 are assigned to buckets. This assignment

is based on the fields of the tuple. Given *B* distinct buckets, the bucket *b* corresponding to a tuple *t* is computed by hashing one or more fields of the tuple modulus *B*. All tuples belonging to a given bucket are sent to the same CEP instance of the *Sub-query* 2.

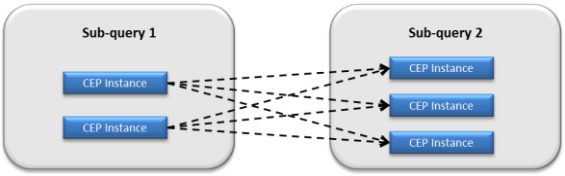

Figure 6: Subquery Connections.

The fields to be used in the hash function depend on the semantics of the window oriented operators defined at *Sub-query* 2 and the mapping among buckets and downstream CEP Instances depends on the load balancing algorithm used in the CEP.

- *Join*: if the join predicate has at least one equality condition, it is an *equi-join* (EJ), otherwise it is a *cartesian product*. For downstream operators of type EJ, the hash function is computed over all the fields used in the equalities plus the optional fields which could appear in the group-by clause.
- *Aggregate*: the fields used in the hash function are all the fields used in the group-by parameter. In this way, it is ensured that all tuples sharing the same values of the attributes specified in the group-by parameter are processed by the same CEP instance.

UPM-CEP comes with this default partition strategy used for splitting a query into sub-queries in the absence of user defined split policies. According to this strategy a new sub-query is created anytime one of the following conditions is satisfied during the query:

- It is a stateful operator.
- It is an operator with more than one input stream.

All the event oriented operators before the first stateful operator are part of the same sub-query.

4.1 UPM-CEP Architecture

The UPM-CEP architecture consists of two main components: the *orchestrator* and *instance managers*. Other components are the reliable registry (Zookeeper) and the *metric server*. Figure 7 depicts how the CEP components can be deployed in a scenario with several NUMA nodes or nodes.

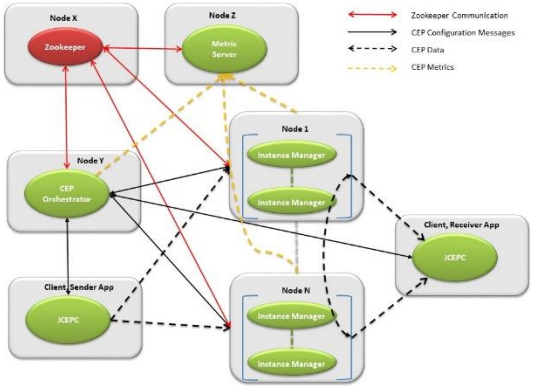

Figure 7: CEP Components.

4.1.1 Orchestrator

The *orchestrator* is in charge of managing the rest of the elements of the CEP. There is only one instance of this component in a deployment. It deploys queries and subqueries in the instance managers and balances the load among different nodes running instance managers.

The state of the orchestrator is kept in Zookeeper (Fundation, 2010) so that, if there is a failure a new orchestrator can be run and take its state from Zookeeper. Active replication could be an alternative design however, although fault-tolerance is easier to implement in this case, the overhead of active replication of the orchestrator will have an impact on regular processing (when there are no failures).

4.1.2 Instance Managers

The instance manager is the component is charge of running queries. Each instance manager runs on a core of NUMA node and can run one or more subqueries. Instance managers are single threaded.

Instance managers receive tuples either from clients (through the JCEPC driver) or from other instance managers. Instance managers must be aware of the nature of the subquery it sends tuples. In a scenario in which there is no parallelism, an instance manger running a subquery will send all the data to the *next subquery*. These means, tuples from the first subquery are sent directly to the next one, this type of tuple sender process is called point to point balancer. Figure 8 shows two subqueries SQ_1 and SQ_2, both are deployed using one instance manager. And tuples t1 to t6 are being sent from SQ_1 to SQ_2 by means of the point to point balancer.

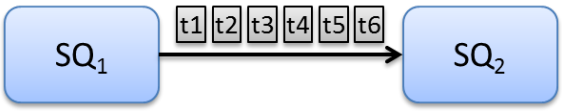

Figure 8: Point to Point Balancer.

However, if SQ_2 is deployed in several instances, SQ_1 has to take into consideration the type of operator is placed at the beginning of SQ_2. If it is a stateless operator, tuples from the previous instance manager are sent in a round-robin fashion. For instance, Figure 9 represents the aforementioned scenario. SQ_2 has been parallelized in three different instances and the first operator is stateless, these means the tuples can be handle by any con the SQ_2 instances. The round robin balancer sends tuples t1 to t6 as presented, tuple t1 is sent to the first instance, t2 is sent to the second instance, t3 is sent to instance number 3 and the process is repeated with tuples t4 to t6 beginning from the first instance.

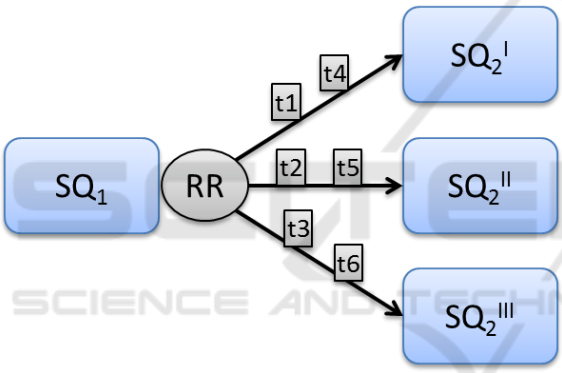

Figure 9: Round Robin Balancer.

Nevertheless, if the first operator is stateful these means that tuples have to be handled by a specified instance. For that cases a route key balancer is required, taking into account the group by clause specified in the operator configuration. Tuples produced by SQ_1 are routed to the required SQ_2 instance, Figure 10 shows how the route key balancer works sending tuples t1 to t6 to the different instances of SQ_2. Tuple t6 is sent to SQ_2^{I}, tuples t1, t2 and t4 are sent to SQ_2^{II} taking into account the route key and finally tuples t3 and t5 are sent to SQ_2^{III}.

To complete the types of balancer presented, the UPM-CEP also defines a broadcast balancer of those operators that requires to send the tuples to all the instances. Figure 11 exposes an example of how tuples are sent from SQ_1 to all SQ_2 instances.

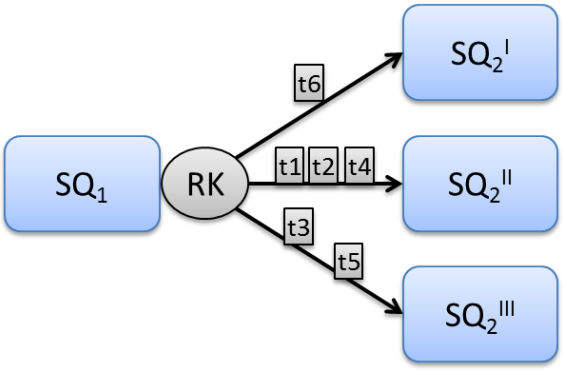

Figure 10: Route key Balancer.

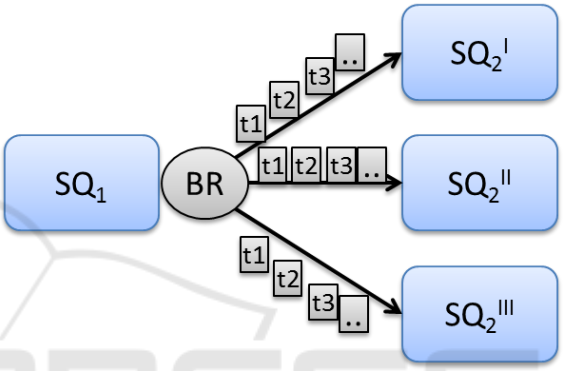

Figure 11: Broadcast Balancer.

5 PERFORMANCE EVALUATION

UPM-CEP performance has been measured using the Intel HiBench benchmark (Intel, 2017). This benchmark allows to evaluate different big data frameworks and contains 19 different workloads that are distributed in: micro, machine learning, sql, graph, websearch and streaming. Specifically, we focus on the screaming workloads: 1) Identity: This workload reads input tuples and produces the same tuples without any modification. A map operator is defined with the same input and output fields. 2) Repartition: Modifies the parallelism level and distributes the load in a round robin fashion. It defines a map operation that copies the input to the output. The query is deployed several times. Tuples are sent to the different instances of the query in a round robin fashion. 3) Stateful wordcount: counts the number word. This workload requires several operators, first of all a map operator picks only the word from the input tuple; an aggregate operator with a number of tuples window and a group by condition based on the word is added. This query tests the route key balancer. 4) Fixed Window: This workload tests the

performance of the time window operator group by a field.

This streaming workload has been implemented to be executed in four different streaming frameworks such as Flink, Storm, Spark and Gearpump. We have implemented same workloads for the UPM-CEP. In this evaluation we use the *Fixed Window* query, which aggregates the connections to a server from each IP address during a period of time. After this time expires, a tuple with the timestamp of the first and last connection from that IP address and the number of connections during that period is produced.

Figure 12: HiBench Fixed Window Topology.

This query, represented in Figure 12, is implemented as a map operator that selects the IP address and the connection time (timestamp) from incoming tuples. Then, an aggregate operator with a time window of 30 seconds per IP is defined. When the window is triggered a tuple is emitted with the timestamp of the first tuple in the window and the number of tuples. To finalize a map operator, add an extra field to the tuple with the timestamp at this moment. The code below corresponds to the *aggregate* function.

```
AggregateOperatorConfig aggregator = new
AggregateOperatorConfig("aggregator",
PROJECTOR_STREAM, AGGR_STREAM);
aggregator.setWindow(OperatorEnums.WindowTyp
e.TIME, wsize, wadv);
aggregator.addGroupByField("ip");
aggregator.addIntegerFunctionMapping("counter",
OperatorEnums.Function.COUNT, "ip");
aggregator.addLongFunctionMapping("startts",
OperatorEnums.Function.LAST_VAL,
ParameterStore.TIMESTAMP_USER_FIELDNAM
E);
        aggregator.addStringFunctionMapping("ip",
OperatorEnums.Function.LAST_VAL, "ip");
 query.addOperator(aggregator );
```

The goal of the evaluation is to demonstrate how NUMA awareness can improve performance. So, we run the same tests twice, one without NUMA awareness (baseline) and then, with NUMA awareness.

The HiBench query is run in a single NUMA node by increasing the load till the maximum throughput is reached. Then, two instances of the query are deployed in two NUMA nodes and repeat the same process with up to 8 NUMA nodes.

The results are shown in Figure 13. The handled load is presented as thousands of tuples per second. With one node, the non-NUMA aware configuration processes up-to 570,000 events per second; the load increases up to 670, 0000 events per second using a NUMA aware configuration. That is, 18% more of load. When 4 nodes are used the handled load without NUMA aware is 2.630 million events per second, while the NUMA aware configuration processes 3,150 million events per second (19% more). The same situation happens with 8 NUMA nodes, where (Fundation, 2010) the load increase reaches up to 26% (6.170 million events and 4.9 million, respectively).

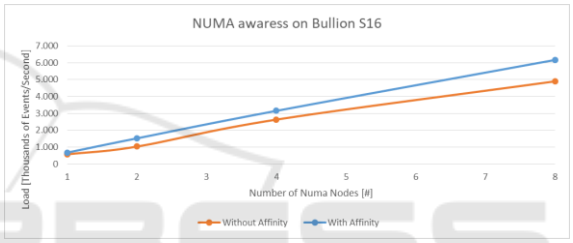

Figure 13: HiBench Throughput with and without NUMA awareness.

6 CONCLUSIONS

In this paper we have presented the design of UPM-CEP and how a NUMA aware configuration can improve performance. The preliminary results with S16 Bullion show large performance gains (up to 25% more load in the largest setup).

As future work we plan to run more benchmarks and use other hardware to confirm the performance results.

ACKNOWLEDGEMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No 732051, 779747, the Madrid Regional Council, FSE and FEDER, projects Cloud4BigData and EDGEDATA (grants S2013TIC2894, S2018/TCS-4499), the ∫œMinistry of Economy and Competitiveness

(MINECO) under project CloudDB (grant TIN2016-80350).

REFERENCES

Ahmad, Y. e. a., 2005. Distributed Operation in the Borealis Stream Processing Engine. *ACM SIGMOD International Conference on Management of Data*, pp. 882-884.

Foundation, A. S., 2015. *Apache Storm*. [En línea] Available at: http://storm.apache.org/

Fundation, T. A. S., 2010. *Apache ZooKeeper*. [En línea] Available at: https://zookeeper.apache.org/

Fundation, T. A. S., 2014. *Apache Flink® - Stateful Computations over Data Streams*. [En línea] Available at: https://flink.apache.org/

Gulisano, V. e. a., 2010. StreamCloud: A Large Scale Data Streaming System. pp. 126-137.

Gulisano, V. e. a., 2012. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel Distributed Systems*, 23 12.pp. 2351-2365.

Intel, 2017. *HiBench*. [En línea] Available at: https://github.com/Intel-bigdata/HiBench

Pu, C. e. a., 2001. Infosphere Project: System Support for Information Flow Applications. *SIGMOD*, Issue 30, pp. 25-34.