Empowering Stream Processing through Edge Clouds

Sergio Esteves[1]*, Nico Janssens[2], Bart Theeten[2], and Luis Veiga[1]
[1]INESC-ID, Instituto Superior Tecnico, Universidade de Lisboa
[2]Bell Labs, Nokia, Antwerp

ABSTRACT

CHive is a new streaming analytics platform to run distributed SQL-style queries on edge clouds. However, CHive is currently tightly coupled to a specific stream processing system (SPS), Apache Storm. In this paper we address the decoupling of the CHive query planner and optimizer from the runtime environment, and also extend the latter to support pluggable runtimes through a common API. As runtimes, we currently support Apache Spark and Flink streaming. The fundamental contribution of this paper is to assess the cost of employing inter-stream parallelism in SPS. Experimental evaluation indicates that we can enable popular SPS to be distributed on edge clouds with stable overhead in terms of throughput.

1. INTRODUCTION

Stream Processing Systems (SPS) are vastly used by companies and organizations to extract insights and value from continuous streams of user data in near real-time. Storm [5], Spark [4], and Flink [2] are popular examples of such systems. At present date, there has been an accentuated demand for these systems to 1) fully support geo-distributed scenarios and 2) support SQL-like queries at interactive speeds. The former implies that partial computation graphs can be computed at possibly distant geographic locations and connected by a same cluster. As for the latter, there have been some recent efforts like Catalyst [7] for Spark. However, this native support for structured queries is yet limited to data sets of fixed size, and precludes continuous streams of data.

In our previous work, CHive [15] is a streaming analytics platform tailored for distributed edge clouds.[1]It enables SQL-like queries, that are executed over continuous streams of data, to be partitioned and distributed over constelations of micro-datacenters (i.e., following an edge computing model). CHive's fundamental contribution is that it optimizes query plans in such a way that the overall bandwidth consumption is minimal.

CHive targets a new scenario of edge clouds that is yet uncommon and not supported by major SPS. Typical widely-deployed SPS, such as Spark, are designed to operate on large clusters within a single datacenter, and assume nodes to be interconnected through high-throughput, low latency, Local Area Networks with full bandwidth availability. Further, CHive is tightly coupled with its runtime environment, Storm, which hinders the adoption of CHive by users of different SPS.

In this paper, we report our experience while addressing this problem of decoupling the CHive query planner and optimizer from its underlying runtime environment. We also propose a middleware layer, named CHive Deployer, that supports the plugging of different runtimes into CHive through a common API. Currently, we provide support for two major and recent SPS, Spark and Flink streaming.

Since commonly used SPS (e.g., Spark) are not designed to allow a cluster to span multiple datacenters (in different geographic areas), CHive relies on orchestrating multiple SPS clusters: each of the clusters typically corresponds to a datacenter, and CHive is responsible for connecting them according to a query plan. This, not commonly explored scenario, follows a edge computing model in which multiple datacenters are combined to execute different parts of a single distributed query. For example, a cluster might handle the first part of a query,

*This work was carried out while the author was an intern at Bell Labs

[1]A typical telecommunications network is built up into multiple layers: the core network, the edge network and the access network. The core network is located at the central office of the operator, while the access network is where the end-user communication lines are terminated. In between sits the edge layer, which is a geographically distributed network of smaller datacenters serving only a limited number of end users. Over the recent years, general purpose compute resources have been added to these distributed datacenters, effectively building out a distributed cloud, also called an edge cloud.

which relies on performing a project and a filter over data coming from a nearby source (in order to reduce the data stream volume), and another cluster might compute the rest of the query which relies on counting the tuples, within a temporal window, grouped by some key.

Supporting multiple runtimes is challenging because it involves using different programming APIs and models that have their own specificities. In addition, the distributed deployment of client applications, that interact directly with the SPS, varies immensely across different systems (e.g., complete Scala application without restrictions, or just a specification of the job computation graph). We aim at making CHive Deployer neutral, with respect to overhead introduced on the underlying SPS, and transparent for applications.

The fundamental contribution of this paper is to generalize CHive so it can be used with widely-deployed SPS. By doing so, we empower commonly used SPS to be distributed on edge clouds and enjoy major bandwidth reductions.

In the next section we survey related work. Section 3 describes the architecture design of the CHive Deployer, and Section 4 its evaluation in a distributed scenario. Finally, Section 5 concludes the paper and points out future work directions.

2. RELATED WORK

Performing streaming data analytics on the edge of networks follows a computing model that has been gaining significant traction lately, specially after the advent of IoT. This model permits to a great extent reducing the amount of data that needs to be transmitted and stored in a central system to perform analytics.

In addition, supporting SQL-style queries over continuous streams of data is a significantly trending topic, especially in an industrial setting. The advantages of using SQL-style are many, including short development cycles and lower maintaince costs when compared to low-level general purpose languages, such as Java and C++, to express analytic-based computations [14]. As aforementioned, the work in this paper builds upon and extends CHive [15], which enables such support for structured queries with windowing-based operators over a edge computing model.

In the research literature, SPSs like Aurora and Medusa [9] have corroborated our vision that stream-based systems can be inherently geographically distributed. They propose a distributed federation of participating nodes (e.g., datacenters) in different administrative domains, that can be scattered in different locations around the globe. Global applications include market data analytics, network monitoring, global surveillance, and e-fraud detection. Despite the author's described intentions regarding declarative query support, it is not clear whether Aurora supports in practice such SQL-style queries.

Apache Edgent [1] is a new project tailored to IoT that allows analyzing data on distributed edge devices. It consists of a programming model and runtime for edge devices. Analytics can be performed locally or in a back-end system according to their complexity. Unlike CHive, devices do not coordinate and share data among themselves, thereby always requiring a centralized system to answer queries involving more than 1 device. Edgent highlights however the necessity of running analytics on the edge of network (premise shared with CHive).

To the best of our knowledge, there is a considerable gap between theory and practice in what concerns to SQL support over streams of data (examples include [10, 8]). Other projects, although claiming to have SQL-like queries support implemented [11, 16, 12], have remained as research projects mainly used by academics, and not available for the general public nor companies. Following, we focus on open-source available solutions.

In the open-source domain, SparkSQL [7] is a new module that enables relational processing, and SQL queries, in Apache Spark. It introduces a highly extensible optimizer, Catalyst, that makes it easy to add new optimization techniques. Although it is part of the authors' future plans, SparkSQL with Catalyst currently do not support the streaming component of Spark. To overcome this, StreamingSQL [6] attempts to extend SparkSQL with windowing based capabilities. However, StreamingSQL functionality is still limited and inefficient, since queries can only be executed over data frames (like a table in a traditional DBMS) that are obtained by converting the results of stream transformations.

MRQL [3] is the closest project to ours: it shares our goals of providing a query processing and optimization system that can be plugged to different underlying data processing systems. Nevertheless, the streaming support is still a work in progress. Despite that, the query optimization techniques are unaware of any network topology, unlike CHive which follows an edge computing model that distributes operators across different datacenters. To the best of our knowledge, none of the available and popular open-source project allows a query to be partitioned and distributed across more than one cluster/datacenter.

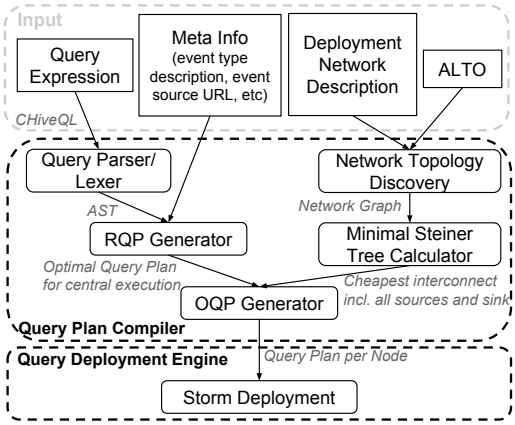

Figure 1: CHive architecture and work flow

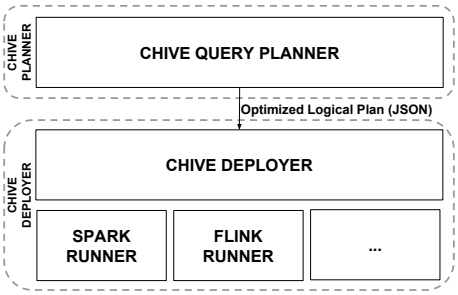

Figure 2: CHive Deployer architecture

3. DESIGN AND IMPLEMENTATION

Figure 1 depicts the general architecture and work flow of the original CHive. Aside from the input layer on top, we can see that there are two main layers in this architecture, represented by the Query Plan Compiler and the Query Deployment Engine components.

Briefly, the query plan compiler generates a reference query plan by using a CHive query expression along with meta information describing event types and event source URLs (among other parameters). Soon after, this reference query plan is combined with a Network topology description and an Optimized Query Plan (OQP) is generated. This OQP that is outputted from the query compiler specifies which chain of operators (or query primitives) should run on which datacenters. This mapping between operators and datacenters is made in a way such that the overall bandwidth consumption is minimal. Afterwards, the deployment engine takes the OQP, which in fact contains a local query plan per datacenter, and deploys them to run on top of Storm, the SPS used with CHive. For more details we refer to [15].

In this work we focus on decoupling the Query Plan Compiler from the Query Deployment Engine, which is specific to Storm in the original CHive. We also change and extend the deployment engine so that different and widely-deployed SPS can be plugged into CHive. Figure 2 clearly depicts the separation that we want to achieve and the architecture overview of the CHive Deployer (Query Deployment Engine in the original CHive).

3.1 Work flow

Figure 3 illustrates the distributed architecture that we get with Spark for a simple pipeline job with 3 stages. In darker grey, we have the components addressed in this paper. The general work flow works as follows. First, the CHive Deployer receives an OQP from the the CHive Query Planner and translates it into a common specification language (more details are given in Section 3.2). Then, this specification is sent to the selected runner and SPS computation graphs are generated.

Soon after, the runner launches the client applications (Spark Driver) to run onto (geo-distributed) remote clusters. These applications, in their turn, submit jobs to the spark executors and collect the corresponding results. Also, these applications, except the one on the last cluster, execute connectors, which serve their corresponding output data to the next downstream cluster.

Finally, a source injects data into the first cluster, which performs some initial computation on it, and sends the results to the next cluster in the middle. This cycle is repeated for the other 2 clusters, using as sources the output of the previous cluster, until the final computation results are sunk to the client.

3.2 CHive Deployer

The CHive Deployer, which takes an OQP from the CHive Planner, is responsible for preparing the local plans therein contained to be deployed on an SPS. This preparation involves translating and adapting the local plans to a common abstract SPS specification. For example, CHive query plans refer to schema attributes, such as to perform a project over *name* and *age* of a stream of data containing people's information. Since most SPS are schemaless, CHive Deployer replaces the attribute's names (e.g., *age*) by the position by which they appear in the stream of data against a string separator; if our stream of data is composed of text lines containing $name|address|age|telephone$, then *name* and *age* would be replaced by *0* and *2* respectively (against the separator |). Hence, this abstract specification is an attempt to find the greatest common denom-

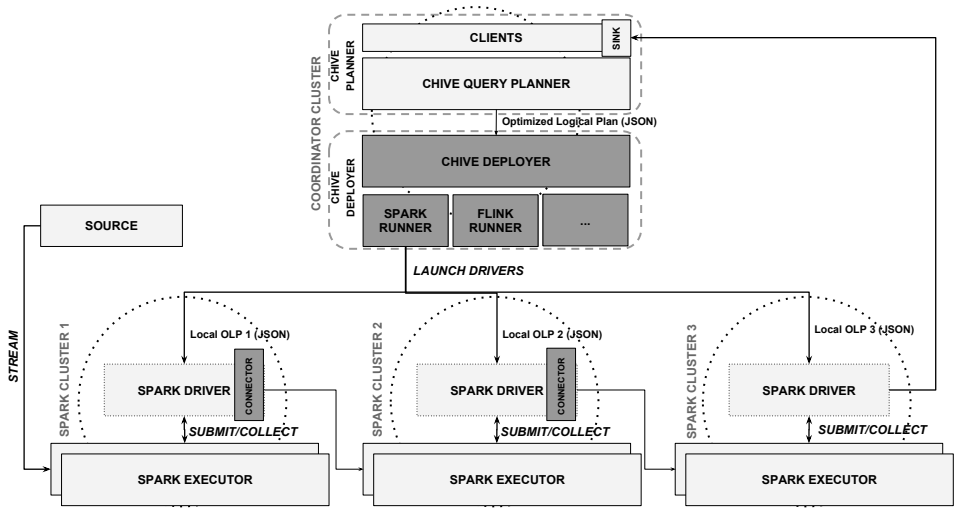

Figure 3: Distributed architecture for simplified pipeline job spanning 3 clusters

inator between the majority of SPS.

In its turn, a runner is responsible for translating this common SPS specification to a target runtime, which includes using the programming model and APIs that are specific to a given SPS. Different runners can be plugged into the CHive Deployer through a common API, which mainly communicates the common SPS specification. Further, this specification must be as fine grained as possible, since some operators of some SPS can perform all-in-one actions; e.g., *reduceByKeyAndWindow* in Spark versus *keyBy().timeWindow().reduce()* in Flink.

By design, it is not currently possible to execute a query plan over heterogeneous runners (e.g., executing a plan using clusters of distinct SPSs). In theory, however, this is possible, since client SPS applications can implement custom code to handle all different types of data and data sources.

3.3 Runner

A runner is responsible for building the computation graph for each local plan (which has been translated into the common specification) that will run on each cluster or datacenter, thereby making use of the underlying programming model that is specific to a given SPS. A runner is also responsible for launching the SPS client applications onto remote clusters. These client applications (one per cluster) interact directly with the SPS, namely submitting jobs, corresponding to the local plans, and collecting results to be shipped out to the next downstream clusters.

Finally, a runner should also take care of connecting the stream of data across clusters. It basically needs to add sources and sinks to all local execution graphs that do not have them, so that all clusters get connected into a single distributed execution graph.

Developing a runner takes a considerable effort: since each SPS has a unique programming model and API, it is necessary to implement all functions and operators that concretize the common specification in a target SPS runtime. Currently, the common abstract specification generated by the CHive Deployer is fully compatible with Spark and Flink, but as we add new features to CHive, it might be possible that not all runners support them (e.g., session windows are available in Flink but not in Spark).

3.4 Connectors

A connector is a component that is responsible for connecting the stream of data across intermediate clusters. For example, having a pipeline job comprising 3 stages spanned across clusters, we would have one cluster to get the input from a given external source; one cluster to sink the final results of the entire computation; and one intermediate cluster that would use connectors to: i) receive its input from the output of the first cluster; and ii) send its output to the input of the last cluster.

A connector can be embedded in SPS client applications or launched as an external process on the same cluster nodes as of the client applications. Whenever possible, connectors should be embedded in SPS client applications, since it is slightly more efficient to send records directly from the application than piping them to an external process. How-

ever, for some SPS this is not possible: Flink, for instance, only allows job graphs to be launched onto remote clusters, and not general full client applications (like it is allowed by Spark).

At its core, a connector maintains an open server connection (e.g., socket based) so that records can be shipped out to the next downstream clusters. In practice, the next downstream clusters fetch input data from the servers opened by connectors.

In case of embedded connectors, they are dependent of the SPS, and thus they need to be developed for each runner. Otherwise, external connectors can be used by different runners, and thus the development effort is reduced at the expense of a slightly slower inter-cluster connection (as aforementioned).

4. EXPERIMENTAL EVALUATION

All benefits of CHive, especially in terms of bandwidth, were already demonstrated in our previous work [15]. In this paper we evaluate the neutrality of the CHive Deployer; i.e., we assess the impact that the CHive Deployer has in terms of overhead on the underlying SPS. Specifically, we show that the CHive Deployer runners are coherent with their corresponding single cluster versions (i.e., without being distributed in a edge clouds model). The objective of this evaluation is to understand whether the overhead of the CHive Deployer is stable and not highly influenced by the specific underlying SPS.

All tests were conducted using 6 machines with an Intel Core i7-2600K CPU at 3.40GHz, 11926MB of RAM memory, and HDD 7200RPM SATA 6Gb/s 32MB cache, connected by 1 Gbps LAN (which ensures a fair reference comparison, given that the network latency between all machines was the same). Also, we used Spark Streaming 1.5.2 and Flink 0.10.2 in our two provided runners.

To evaluate the CHive Deployer we relied on a scenario to calculate the top 20 websites generating the highest download volumes in the last 10 seconds, from a stream of real-world traces of a large mobile operator. Figure 4 depicts the Optimized Query Plan that we get from the CHive Planner for this considered scenario. We can see that the respective query is distributed across 6 different datacenters, corresponding to 6 different machines in our experiment. The normal triangles, ellipses, and inverted triangle represent data sources, operators, and the sink respectively.

In a first experiment, we measured the throughput obtained for the considered scenario in terms of total number of records processed, per window of 10 seconds, for the entire computation. We measured this throughput for our two implemented runners, Spark and Flink, and compared CHive with the baseline system. In CHive mode (i.e., using a edge computing model), there are 6 Spark/Flink separate clusters where each executes part of the computation graph (like depicted in Figure 4). As for the baseline mode, which represents a classic situation, it comprises a single Spark/Flink cluster spanned across 6 machines where each one runs the entire computation graph on different data partitions. Note that these systems are not designed to run across different geographic locations that are distant from one another; i.e., there are strong limitations in spanning a cluster across multiple datacenters [13] (unlike the CHive approach).

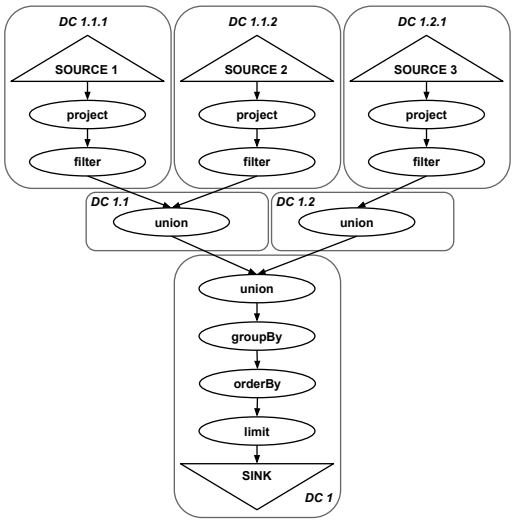

Figure 4: OQP for the top 20 websites with the highest download volumes

We observed that the baseline mode yields higher throughput than the CHive mode for both considered SPS. That is to be expected, since input data in the baseline system is given to all of the 6 machines, whereas in CHive only 3 machines (corresponding to 3 clusters) are fed with data from the sources. (Refer to our previous work to see the advantages of CHive against other systems, especially in terms of bandwidth.) The important point to note here is the neutrality of the CHive Deployer: on each considered SPS, the throughput difference remains in (almost) the same proportion between baseline and CHive modes.

Table 1 shows the differences of throughput in proportion (i.e., we divide the throughput obtained with baseline by the one of CHive for Spark and Flink). In both SPS, the ratio between baseline and CHive is the same within a deviation of less than 5%, which indicates that CHive Deployer is neutral

Table 1: Deviation (as percentage change) of throughput ratios between baseline and CHive with Spark and Flink runners

Spark	Flink	Deviation
1.32	**1.37**	**4%**
(2832/2150)	(1184/863)	(1.37/1.32-1)

Table 2: Deviation (as percentage change) of total number of bytes ratios between baseline and CHive with Spark and Flink runners

Spark	Flink	Deviation
1.50	**1.38**	**9%**
(148631/99159)	(62372/45175)	(1.50/1.38-1)

and not intrusive in relation to the baseline system.

In a second experiment, and using the same setup and query as of the first experiment, we assessed the deviation in the results, originated by different throughputs (within a window), with the objective of seeing how far results are between baseline and CHive. Specifically, we have counted the total number of bytes that we obtain from the output, which corresponds to the sum of the bytes of all top 20 websites, in a 10 second window for Spark and Flink, while comparing the CHive mode against the baseline system. This comparison consisted of the division between the total number of bytes in baseline and CHive, for Spark and Flink, as shown in Table 2. The result accuracy difference that we obtain for both SPS is the same within a deviation of less that 10% (cf. table below), which indicates CHive Deployer is significantly coherent across runners.

As a side effect, we have also shown a comparison between Spark and Flink themselves. For our specific workload, Spark outperformed Flink. This was mainly due to some operators/tasks that have higher parallelism levels and are more optimized in Spark (such as sorting tuples within a window).

5. CONCLUSION

In our previous work, CHive [15] enables structured interactive queries to run distributed on continuous streams of data, over edge clouds, in a bandwidth efficient manner. This computing model, that has been gaining significant traction lately (specially with the advent of IoT), is made available to popular SPS through the generalization of CHive (that is addressed in this paper).

In particular, this paper addressed the decoupling of the CHive query planner from its underlying runtime environment. We have built CHive Deployer, a middleware layer that makes possible to use different and widely-deployed SPS with CHive through a common API. We have also demonstrated the feasibility of plugging runners and SPS to CHive Deployer by developing two for popular SPS: Spark and Flink. Experimental evaluation, with real-world data, indicates that CHive Deployer is neutral (not affected by the underlying technology) and does not introduce any additional overhead.

Acknowledgements: This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference UID/CEC/50021/2013.

6. REFERENCES

[1] Apache Edgent. `http://edgent.incubator.apache.org/`.
[2] Apache Flink. `http://flink.apache.org/`.
[3] Apache MRQL. `https://mrql.incubator.apache.org/`.
[4] Apache Spark. `http://spark.apache.org/`.
[5] Apache Storm. `http://storm.apache.org/`.
[6] Streaming SQL for Apache Spark. `https://github.com/Intel-bigdata/spark-streamingsql`.
[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.
[8] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, Sept. 2001.
[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.
[10] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: The system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.
[11] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong. Nile: a query processing engine for data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 851–, March 2004.
[12] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 245–256, Asilomar, California, Jan. 2003.
[13] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 421–434, New York, NY, USA, 2015. ACM.
[14] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, Dec. 2005.
[15] B. Theeten and N. Janssens. Chive: Bandwidth optimized continuous querying in distributed clouds. *Cloud Computing, IEEE Transactions on*, 3(2):219–232, April 2015.
[16] Y. Wei, S. H. Son, and J. A. Stankovic. Rtstream: real-time query processing for data streams. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 10 pp.–, April 2006.