The Case for Better Integrating Scalable Data Stores and Stream-Processing Systems

Antonis Papaioannou*†, Chrysostomos Zeginis* and Kostas Magoutis*†
*Institute of Computer Science, Foundation for Research and Technology – Hellas, Heraklion 70013, Greece
†Computer Science Department, University of Crete, Heraklion 70013, Greece

***Abstract*—Scalable stream processing systems require external storage systems for long-term storage of non-emphemeral state. Recent research have pointed to scalable in-memory key-value stores, such as Redis, as an efficient solution to external management of state [1], [2]. While such data stores have been interconnected with scalable streaming systems, they are currently managed independently, missing opportunities for optimizations, such as exploiting locality between stream partitions and table shards, as well as coordinating elasticity actions.**

I. Introduction

Scalable stream-processing systems (SPSs) and analytics are key to a number of high data-volume Internet services [2], [3], [4], [5]. A stream application can be modeled as a directed dataflow graph comprising processing operators, as shown in Fig. 1. Each operator transforms data flowing from its inputs to its outputs. Stateless operators produce their output based on the current inputs, whereas stateful operators preserve a sequence of recent data as its internal state and apply a transformation on a subset of its incoming data (e.g. aggregating values over a time window).

Early SPSs stored ephemeral operator state either in memory or on an external data store (e.g., a SQL server). Modern systems store operator data using a combination of an embedded local key-value store (KVS) and a remote scalable KVS for fault-tolerance [6], [7]. *External state* (streams produced by other applications or even by other systems [3], [4], [5] or ground-truth table data, can be accessed during the execution of a streaming application (see 'External State' in Fig. 1). External state represents data whose lifecycle is disconnected from that of the streaming application, and stored on (typically scalable) KVSs [3], [4], [8]. In contrast, *internal state* of a streaming application refers to intermediate (ephemeral) data used by operators of the application. While it is important to store internal state in a reliable, recoverable manner for fault-tolerance, persistence of external state is a separate concern. Current state-of-the-art solutions, such as IBM InfoSphere Streams, use scalable KVSs such as Redis, for scalable storage of external state. However, lack of coordination in the placement/partitioning policies of SPSs and external-state KVSs, results to often unnecessary network cross-talk, and mis-alignment of adaptation policies such as elasticity.

Our research focus is to highlight the benefits of coordinating placement/partition policies of scalable SPSs and external-state KVSs. In this poster we report preliminary experimental results from a representative application (derived from the Yahoo streaming benchmark [8]) on the Flink [7] SPS using Redis as an external-state KVS and discuss ongoing and future work on carrying over these results to larger scale experiments, and to compare to previous work using commercial state-of-the-art systems [1], [2].

978-1-7281-6677-3/20/$31.00 © 2020 IEEE

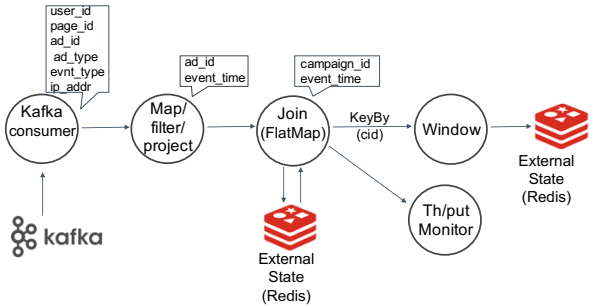

Fig. 1: Streaming application used in this work

II. Related work

Existing scalable SPSs provide ways to interact with scalable external stores, viewing them as source/sink operators that exercise an API to a scalable KVS such as Cluster Redis [6], [7]. IBM InfoSphere Streams offers *distributed process store*, a remote interface to a scalable data store [1], demonstrated to support scalable setups in commercially relevant use cases [2]. Accessing external table state in the common path of stream processing is known to be a cause of overhead (higher latency, as well as protocol overhead). With out-of-core datasets stored as external state (e.g., user profiles in large Internet services [3], [5], [6]), external state is a pain point that is not well addressed so far. In this poster we provide quantitative evidence that locality is indeed worthwhile in modern testbeds, and describe future work to extend the results as well as address ways to achieve them (by aligning stream partitions and table shards to achieve locality). In particular, there is recent research on incremental elastictity of SPSs [9] and KVSs [10], using similar techniques, that however need to be coordinated to preserve locality between the two systems.

III. Preliminary results

In this section we discuss preliminary results of our ongoing research work. Our results quantify the performance benefits of aligning the data placement of an external store (Redis) with

stream processing tasks. Our use case is an application derived from the Yahoo Streaming Benchmark (YSB) [8], programmed and deployed on the Flink [7] SPS. We ingest load using a synthetic data generator similar to that in YSB. Figure 1 shows the dataflow of our application. The application reads events from Kafka, deserializes the JSON string, filters the event types and keeps (projects) the necessary fields. It then performs a join operation on each event to map the `ad_id` of each input tuple to its associated `campaign_id`. This information is maintained in a Redis database storing external state (tables). The join operation is implemented with a FlatMap operator which takes one input element and produces zero, one, or more output elements. In our case, it performs a get request to the Redis store in order to map the `ad_id` of every incoming tuple to the corresponding `campaign_id` (similar to YSB). After the join, data are partitioned by the `campaign_id` field before the window operator splits the stream into buckets of 10 seconds and write the output to a Redis database.

We aim to study the performance impact of collocated vs. remote placement of the Redis store relative to the join operator task. As the join operator interacts with Redis on each incoming tuple, it is on the critical path of stream processing, and thus we expect that a collocated Redis will provide an advantage for streaming application performance. We measure the throughput of the join operator using a custom operator (Throughput monitor) that we developed for this purpose.

Our experimental testbed consists of four servers, each equipped with a Intel Xeon Bronze 3106 8-core 1.70GHz CPU, 16GB DRAM, 256GB Intel D3-S4610 SSD, Ubuntu Linux 16.04.6 LTS, interconnected with a 10Gb/s Dell N4032 switch. We dedicate a server for the data generator and Kafka service. In our first scenario, the application logic and Redis instance are deployed on a single server (*local*). A third node is used only in runs with a remote Redis instance (*remote*). We run experiments with one or three parallel instances of each processing operator (*single-1, single-3*) all deployed on a single node. The system exhibits up to 1.77x higher throughput when the external state is colocated with the join operator due to eliminating network overhead in accessing Redis (Figure 2).

Next we study a multi-node scaling scenario using Redis in clustered mode. We use three servers as processing nodes, each hosting an instance of Flink operators and a node of the Redis cluster. In the first case (*remote*), data are randomly spread across the three Redis nodes by Redis' native partitioning scheme. Each operator accesses data on its local and remote Redis instances. In a second scenario (*local*) we replicate all data in each of the three Redis nodes, ensuring that each processing operator can access all external state locally. The system exhibits 1.66x higher throughput in *local* compared to *remote* (Figure 2, *multi-3*). Speedup is slightly higher in *single-3* compared to *multi-3* as the latter exhibits slighly higher throughput in the *remote* Redis configuration. This is because *multi-3 remote* satisfies a fraction of get requests locally whereas in *single-3 remote* all requests are satisfied remotely. Our results validate and quantify the benefits of locality in accessing external state over high-performance hardware, motivating further research in larger scale benchmarks.

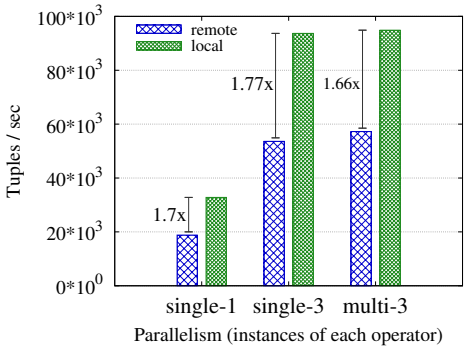

Fig. 2: Throughput of join operator

IV. Future Work

Preliminary results validate that exploiting data locality between processing tasks (SPS) and external state (KVS) can lead to significant (1.7x) performance improvements in small-scale setups by reducing cross-talk between processing tasks and the corresponding external state over the network. To the best of our knowledge, such collocation has not been explored in previous work [2], [4]; we expect that coordination between SPS and KVS will lead to similar measurable benefits when extended in deployments over larger clusters. Control over data placement however requires the coordination of actions on both stream processing and distributed data stores, including during adaptation actions such as elasticity. Although there are efficient elasticity mechanisms for KVSs [10] and SPSs [9], we are pursuing the integration and coordination of such actions in a SPS-KVS system.

The authors thankfully acknowledge funding by the Hellenic Foundation for Research and Innovation through the STREAMSTORE faculty grant (Grant ID HFRI-FM17-1998).

References

[1] N. Senthil and G. Bugra, "Using InfoSphere Streams with memcached and Redis," in *IBM developerWorks*, 2013. [Online]. Available: https://www.ibm.com/developerworks/library/bd-streamsmemcached/

[2] R. Rea, "Walmart & IBM Revisit the Linear Road Benchmark," May 10-11, 2016. [Online]. Available: https://www.slideshare.net/RedisLabs/walmart-ibm-revisit-the-linear-road-benchmark

[3] R. Sumbaly, J. Kreps, and S. Shah, "The big data ecosystem at linkedin," in *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, 2013.

[4] N. Jain *et al.*, "Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core," in *Proc. of 25th ACM SIGMOD Int. Conf.on Management of Data (SIGMOD '06)*, June 2006.

[5] G. J. Chen *et al.*, "Realtime data processing at facebook," in *Proc. of the 2016 Int. Conf. on Management of Data (SIGMOD '16)*, 2016.

[6] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, no. 12, Aug. 2017.

[7] P. Carbone *et al.*, "State management in apache flink: Consistent stateful distributed stream processing," *Proc. VLDB*, vol. 10, no. 12, Aug. 2017.

[8] S. Chintapalli *et al.*, "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming," in *Proc. of IPDRM'16*, 2016.

[9] M. Hoffmann *et al.*, "Megaphone: latency-conscious state migration for distributed streaming dataflows," in *Proc. VLDB*, vol. 12, no. 9, 2019.

[10] A. Papaioannou and K. Magoutis, "Incremental Elasticity for NoSQL Data Stores," in *Proc. of 36th IEEE SRDS*, 2017.