Efficient Migration of Very Large Distributed State for Scalable Stream Processing

Bonaventura Del Monte
supervised by Prof. Volker Markl
DFKI GmbH
bonaventura.delmonte@dfki.de

ABSTRACT

Any scalable stream data processing engine must handle the dynamic nature of data streams and it must quickly react to every fluctuation in the data rate. Many systems successfully address data rate spikes through resource elasticity and dynamic load balancing. The main challenge is the presence of stateful operators because their internal, mutable state must be scaled out while assuring fault-tolerance and continuous stream processing. Both rescaling, load balancing, and recovering demand state movement among work units. Therefore, how to guarantee those features in the presence of large distributed state with minimal impact on the performance is still an open issue. We propose an incremental migration mechanism for fine-grained state shards through periodic *incremental checkpoints* and *replica groups*. This enables moving large state with minimal impact on stream processing. Finally, we present a low-latency hand-over protocol that smoothly migrates tuples processing among work units.

1. INTRODUCTION

Existing scalable *Stream Data Processing Engines* (SPEs) offer fast stateful processing of data streams with low latency and high throughput despite fluctuations in the data rate. To this end, stateful processing benefits from on-demand resource elasticity, load balancing, and fault tolerance. Currently, both research [5, 6, 17, 13] and industry [1, 4, 18] address scaling up stateful operators while assuring fault tolerance *in case of partitioned or partially distributed large state*. Here, large state means hundreds of gigabytes.

A motivating example. Many streaming applications require stateful processing. Examples of such applications are the data analytics stacks behind popular multimedia services, online marketplaces, and mobile games. These stacks perform complex event processing on live streams. Multimedia services and online marketplaces recommend new contents or items to their users through collaborative filtering [16]. Producers of mobile games track in-game behaviour of players to promote the best in-app purchase and to detect frauds. The size of the state in these applications scales with the number of users and their interactions with the application (e.g., rated items, purchases, actions of a player) and can grow to terabyte sizes. State in the size of terabytes introduces a multifaceted challenge. The SPE must optimally manage cluster resources respecting the size of the state. This does not only apply to intra-cluster instances but also inter-cluster ones, e.g., migrating the SPE among operational environments or to cheaper "pay-as-you-go" instances. Besides, parallel analytic algorithms need global state. Parallel instances of an operator work on their state partition and then update global state, e.g., machine learning models. Therefore, these analytics result in *very large distributed state*.

Research goal. Motivated by industrial needs, our goal is to achieve stream processing with low latency and high throughput when operators handle very large state. To this end, we focus on management techniques that enable fault-tolerance, on-demand resource scaling, and load balancing in the presence of very large distributed state.

Problem statement. Handling operators with very large distributed state is cumbersome. Guaranteeing fault-tolerance, resource elasticity, and dynamic load balancing for these operators (i) require state transfer, (ii) must not undermine the consistency of distributed state shards, and (iii) demand robust query processing performance. State transfer introduces latency proportional to its size. Exactly-once stream processing requires consistent state, i.e., results must be as accurate as if no failure happened or the SPE did not perform any rescaling or rebalancing operation on the state. Besides, a SPE must continuously process stream tuples despite any of those operations.

Current approaches. To the best of our knowledge, there is no system that fully features efficient state management when distributed very large state is involved. Many authors investigated this problem by constraining their scope to partitioned or partially distributed state [5, 4, 18] and to smaller size [7, 6, 17].

Proposed solution. Our solution is a low-latency incremental migration mechanism that moves fine-grained state shards by using periodic *incremental checkpoints* and *replica groups*. An incremental checkpoint is a periodic snapshot of a state shards that involves only modified values. A replica group is a set of computing instances holding a copy of a portion of the state. Our migration mechanism moves large operator states with low impact on the system performance and without stopping the streaming topology. Although incremental migration reduces the transfer overhead, we also provide a placement scheme for primary state shards and replica groups that minimizes transfer cost. Our solutions are as follows:

1. a communication-efficient replication protocol that keeps a replica group consistent with the changes in the state of the primary operator
2. an optimal primary state shards and replica groups placement for decreasing migration cost
3. a hand-over protocol that migrates the processing between two work units with minimal latency.

We point out that this thesis is at an early stage, hence, we do not have any experimental validation yet.

Proceedings of the VLDB 2017 PhD Workshop, August 28, 2017. Munich, Germany.

2. RELATED WORK

Castro et al. address the problem of scaling up and recovering stateful operators in a cloud environment through a set of primitives for state management that enables scaling up and recovery of stateful operators [5]. Their experiments include operators with small states and they confirmed that larger state has a higher recovery time. In a second work, the same authors propose a new abstraction over large mutable state, called stateful dataflow graph, which manages partitioned or partial distributed state [6]. Our aim is to fill the gap in this area by providing a mechanism that both scales out and recovers a long-running system with very large distributed state. ChronoStream is a system that seamlessly migrates and executes tasks [17], whose authors believe to have achieved costless migration thank to a locality-sensitive data placement scheme, delta checkpointing, and a lightweight transactional migration protocol. Although their experiments look promising, we argue transactional migration may be avoided by using two different protocols (one for state migration and one for the hand-over) and delta checkpoints adds synchronization issues.

Ding et al. deal with finding the optimal task assignment that minimizes the costs for state migration and satisfies load balancing constraints [7]. To this end, they introduce a live and progressive migration mechanism with negligible and controllable delay. They come to a different conclusion w.r.t. ChronoStream, because they also argue that synchronization issues may affect results correctness while performing a migration. The solution of Ding et al. performs multiple mini-migrations progressively: each mini-migration migrates a number of tasks smaller than a given threshold [7]. On the other hand, their experiments do not cover large state migration and it is unclear how the system could perform in such task. Furthermore, both ChronoStream and Ding et al. consider partitioned state.

Nasir et al. present *partial key grouping* as a solution to handle load imbalance caused by skewness in the keys distribution of input streams [14, 15]. The main idea is to keep track of the number of items in each parallel instance of an operator and route a new item to the instance with smaller load. Items with the same key are routed to different parallel instances of the same operator. An improvement to the solution is to determine the "hottest" keys in the stream and assign more workers to those keys. However, they assumed the operator state has the associative property, thus merging intermediate partitioned substates is possible with an extra aggregate operation. Splitting the state of a given key, indeed, mitigates its growth on one working unit, yet aggregating large state will require some potentially expensive network transfers. Our aim is to propose a load balancing approach that avoids such partial aggregations.

Gedik et al. propose transparent auto-parallelization for stream processing through a migration mechanism [8]. However, we argue that their approach does not consider distributed large state and it is totally decoupled from fault-tolerance.

Many SPEs have effectively implemented state management techniques (e.g., Apache Flink [1, 4], Apache Spark [18], SEEP [6], Naiad [13]). In particular, Apache Flink features a technique that asynchronously checkpoints the global states to minimize the latency of a snapshot [3].

3. RESEARCH ISSUES

Our goal is to move large operator states with minimal impact on the performance of query processing. Migrating large states between operator instances in one shot is expensive due to network transfer, especially if the system is already overloaded during its regular operation. Our key idea is to incrementally maintain a replica group for each fine-grained state unit over different work units. Each replica is updated through *incremental checkpoints* generated on the primary operator. In addition, intrinsic issues of migration pose new challenges, e.g., data consistency, tuples rerouting, physical shards handling, and network transfer cost. To better explain our key idea, we first define our data and system models. Then we provide an analysis of our research goals.

3.1 System Model

Data Model. Let S be a stream of tuples, for each tuple $q \in S$, we define k_q as the value of the partitioning key and t_q as its monotonically generated time-stamp.

Stream processing. Our system is made of p work units running on z physical nodes (each of them can run a variable number of work units). Our system executes jobs/queries expressed as a dataflow graph. Each operator of the graph runs on maximum p parallel instances. An operator takes n streams and outputs m streams. Every parallel instance receives tuples (sent from upstream operators) w.r.t. a distribution function that computes the assignments through k_q.

State model. The global state of all the operators in the streaming topology is a distributed logically partitionable data store (e.g., a distributed K-V store). Partitions of this data store contain a single state entry, e.g., window content of an operator, user-defined counters. Each logical partition is made of physical shards. Every parallel instance of an operator holds its own shard. Besides, each shard is made of fine grained data items. Each key of the input stream owns few data item in every logical partition of the state and each shard holds a range of keys. Each range of keys can be further partitioned and optionally split. Distributed state demands some consistency guarantee in case (i) a key needs to be stored in multiple shards, and (ii) tuple processing might trigger changes in more than one shard. The distribution function determines the content of the shards kept by stateful instances. Thus, each parallel instance of an operator does not only process tuples with specific keys but it also holds the data items of state for those keys.

3.2 Incremental Checkpoints

A prerequisite for our set of protocols is an incremental checkpoint protocol based on the approach of Carbone et al. [3]. Instead of taking a snapshot of the whole state, we asynchronously checkpoint the modified state values between the previous checkpoint and the current one. An asynchronous checkpoint executed at time time t will not contain updates happened later than t.

3.3 Replication Protocol

We design a replication protocol to keep the global state of a streaming topology replicated and consistent. This protocol replicates every primary state of each operator instance on a given number of work unit, i.e., each sub-range of keys has its own replica group. The purpose of a replica group is to keep a copy of different sub-ranges of keys for each operator. A primary operators sends incremental checkpoints for a given range of keys to its replica through the network.

3.4 Hand-Over Protocol

The hand-over protocol moves the processing of a given keys range (k_s,k_e) between two parallel instance of a target stateful operator. The system triggers this protocol when it detects the need of either rescaling an operator, balancing the load over parallel instances of an operator, or recovering an operator. Main ideas behind this protocol are the usage of replica groups, incremental checkpoints and the embedding of the protocol itself in the dataflow paradigm. Moving the processing of any key involves tuple rerouting and migration of the state for that key. This operation is lightweight if the destination instance is in the replica group of the moved key. Indeed, the replica group misses at most the last incremental checkpoint. Let *upstream* be all the operators that send some input tuples to a target downstream operator, the steps of the protocol are:

1. The system decides to migrate that tuples marked with keys in range (k_s,k_e), from downstream instance o_s to o_t, which is in the replica group of (k_s,k_e)
2. Upstream injects a *key move* event in the data flow for keys k_s,...,k_e involving operators o_s and o_t
3. Upstream sends its outgoing tuples marked with keys k_s,...,k_e to o_t, which processes them creating new states s'_e,...,s'_t
4. o_s generates an incremental checkpoint that contains its current states s_e,...,s_t for keys k_s,...,k_e and sends it to o_t
5. As soon as o_t gets the incremental checkpoint, it updates its current states s_e,...,s_t with the received checkpoint
6. Then o_t asynchronously merges them with s'_e,...,s'_t. If new tuples arrive in o_t, it generates new states and subsequently merges them.

As a result, the handover protocol guarantees eventual consistency on every migrated primary state after merging. Moreover, we assume that user-defined state has *update* and *merge* policies; the former updates state by processing a stream tuple, whereas the latter merges two partial states for the same key. If merging of partial state is not semantically possible, then the target instance buffers incoming tuples and updates the state upon its full receiving.

3.5 Optimal Placement of Replica Groups

Each keys replica group is composed of q physical nodes, as we aim to minimize continuous migration cost, the replica group has to be optimally placed over the streaming topology. Indeed, transferring an incremental checkpoint from a node a to b could potentially have a different cost than shipping the same checkpoint to node c. This problem can be mapped as a bipartite graph matching problem whose classic solution is well-know as the Hungarian or Kuhn-Munkres algorithm [11, 12]. Nevertheless, our scenario is not static as we need to deal with resource scaling and failing nodes. Therefore, a dynamic approach to the assignment problem [10] is the best fit for our needs, since we look for an optimal assignment of the state items to an elastic set of physical nodes. Our optimization problem is formulated as follows: given l sub-ranges of keys and z physical nodes, find a placement for each sub-range of keys over q out of z nodes that minimizes the migration cost. We evaluate the cost of shipping an incremental checkpoint between two nodes by considering their workloads and the number of network switches involved.

4. RESEARCH PLAN

In this thesis, we intend to investigate above research issues w.r.t. our goal: transparently providing fault-tolerance, resource elasticity, and load balancing in the presence of very large distributed state. Our focus is to investigate the trade-offs behind our proposed solution. First, the hand-over protocol presents several challenges, e.g., the granularity of the keys ranges, the concurrent execution of the protocol, and the triggering policy of the protocol (through either consensus, common knowledge or centralized entity). Secondly, we plan to investigate the usage of log-less replication (similarly to Bizur [9]) by using shared registers [2]. Besides, log-less replication implies no log compaction overhead. As network is the main bottleneck, we plan to research orthogonal optimizations to reduce network overhead, e.g, remote direct memory access, data compression, and approximation. Lastly, the placement scheme of replica groups may require further investigation as our initial definition of migration might neglect significant hidden cost.

4.1 Achievement Plan

We have a clear idea about the achievement of our goal, which we define in the following sections.
Fault-tolerance. Our replication protocol guarantees that each replica group holds a copy of some ranges of keys for different operators. Since each key is replicated in $q+1$ physical units, the system can sustain up to q failing instances of an operator by resuming the computation on one unit in its replica group. The system may need to replay some tuple unless the group has the latest state checkpoint and the failing unit did not process any newer tuple.
Load balancing. Relying on a load balancing policy (e.g., shard size or ingested tuples count above a given threshold), the system triggers the hand-over protocol. Then, the hand-over protocol seamlessly moves the processing of some keys ranges from a primary work unit to another in their replica groups. Determining the placement of ranges of keys for the primary state is another orthogonal challenge that we plan to overcome.
Resource elasticity. Regardless of the chosen elasticity policy, we need to efficiently rescale the state of every range of keys along with its replicas minimizing the transfer cost. Rescaling possibly involves deleting some replicas, whereas state transfer can be still done incrementally by using above protocols. As we consider primary state and replica as one entity, we reassign them to parallel instances as described in Section 3.5. This procedure could benefit from current IaaS platforms where multiple VMs or containers share physical hardware. Indeed, we may provision new resources on either an already used physical node or a new node. The last scenario is more challenging as the system must migrate entire shards of the state to the new node.

4.2 The system in action

In Figure 1, we show a toy example of our system while it seamlessly performs resource scaling, state recovery, and load balancing. The figure shows a simple dataflow graph made of one source (parallelism=2) and one operator (parallelism=4). For the sake of simplicity, we marked tuples and state for the same keys range with the same colour. Each primary state for every keys range has only one replica group. In Figure 1.A, the first instance is failing while the third one is overloaded. In Figure 1.B, the hand-over protocol seamlessly moves the processing and the state of both yellow and violet keys ranges. As the state of the yellow key range was on a failing node, our system must reply lost tuples. Meanwhile, the fourth instance processes violet tuples and creates a new partial state. The hand-over protocol merges this partial state with the current replica and the last incremental checkpoint. Simultaneously, our system provisions a new instance and migrates the red state as it detects an overloaded second instance. In Figure 1.C, the system is finally stable. The violet state is migrated and replicated on the fourth and second instance, respectively. The yellow state is restored on the second instance and replicated on the new instance. The red state is replicated on the new instance.

4.3 Evaluation Plan

We assess the capabilities of our system through the following set of *Key Performance Indicators* (KPIs):

1. the execution of our protocols must have negligible effect on query processing performance
2. the system must guarantee exactly-once stream processing and state consistency
3. performing a load balancing or a resources scaling operation must improve resource utilization of the physical infrastructure and prevent bottlenecks (e.g., operator back-pressure)

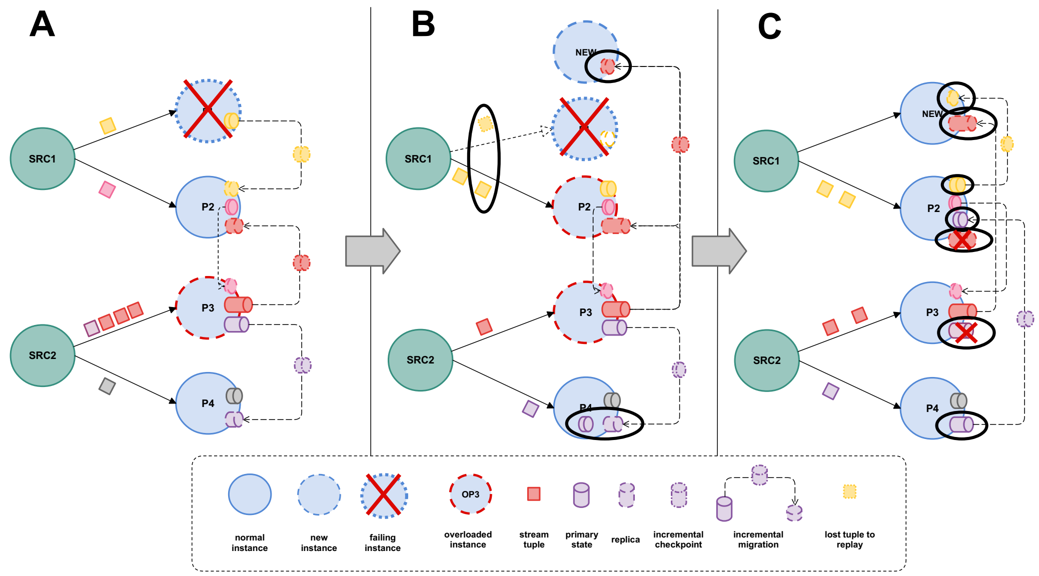

Figure 1: Our protocols in action: tuples and state for the same keys range are marked with the same colour. Primary state for every keys range is incrementally replicated only once. Sensible steps of the hand-over protocol are circled.

To meet above KPIs, we intend to design a suite of benchmarks that thoroughly stresses our proposed system. We plan to define a set of metrics (e.g., tuple processing throughput and latency, migrated state items, checkpoint size) and measure them in our system on different real-world workloads, with distinct scaling and balancing policies, and different replica factors. Finally, we expect to compare our results with baseline systems.

4.4 Future directions

We envision a system able to continuously process stream tuples despite data rate spikes and failures. This system can also seamlessly migrate itself among cluster, e.g., from one IaaS-provider to a cheaper vendor, between two operational environments. Incremental state migration will be a building block of such operations. Other orthogonal research areas may be: (i) investigating the usage of new storage hardware, e.g, NVRAM and SSD, (ii) considering non-keyed state and queryable state, (iii) providing elastic job maintenance, (iv) exploring data compression techniques to reduce state size, and (v) investigating incremental state migration (and resource elasticity) in case of Hybrid Transactional-Analytical Processing (HTAP) workloads.

Acknowledgments: We would like to thank our advisor Prof. Volker Markl, as well as Prof. Tilmann Rabl for his valuable guidance, as well as Dr. Asterios Katsifodimos, Dr. Sebastian Breß, and Dr. Alireza Rezaei Mahdiraji for their support. This work has been partially supported by the European Commission through PROTEUS (ref. 687691).

5. REFERENCES

[1] A. Alexandrov, R. Bergmann, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 2014.

[2] H. Attiya, A. Bar-Noy, et al. Sharing memory robustly in message-passing systems. In *ACM PODC*. 1990.

[3] P. Carbone, G. Fóra, et al. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.

[4] P. Carbone, A. Katsifodimos, et al. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 30(40), 2015.

[5] R. Castro Fernandez, M. Migliavacca, et al. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM SIGMOD*. 2013.

[6] —. Making state explicit for imperative big data processing. In *USENIX ATC*. 2014.

[7] J. Ding, T. Fu, et al. Optimal operator state migration for elastic data stream processing. *CoRR*, abs/1501.03619, 2015.

[8] B. Gedik, S. Schneider, et al. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.*, 2014.

[9] E. Hoch, Y. Ben-Yehuda, et al. Bizur: A key-value consensus algorithm for scalable file-systems. *CoRR*, abs/1702.04242, 2017.

[10] G. A. Korsah, A. T. Stentz , et al. The dynamic hungarian algorithm for the assignment problem with changing costs. Tech. Rep. CMU-RI-TR-07-27, 2007.

[11] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 1955.

[12] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society of Industrial and Applied Mathematics*, 1957.

[13] D. Murray, F. McSherry, et al. Naiad: A timely dataflow system. In *ACM SOSP*. 2013.

[14] M. Nasir, G. Morales, et al. The power of both choices: Practical load balancing for distributed stream processing engines. *CoRR*, abs/1504.00788, 2015.

[15] —. When two choices are not enough: Balancing at scale in distributed stream processing. *CoRR*, abs/1510.05714, 2015.

[16] R. Sumbaly, J. Kreps, et al. The big data ecosystem at linkedin. In *ACM SIGMOD*. 2013.

[17] Y. Wu and K. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *IEEE ICDE*. 2015.

[18] M. Zaharia, T. Das, et al. Discretized streams: Fault-tolerant streaming computation at scale. In *ACM SOSP*. 2013.