Integrating Fault-Tolerance and Elasticity in a Distributed Data Stream Processing System

Kasper Grud Skat Madsen Philip Thyssen Yongluan Zhou
{kaspergsm, zhou}@imada.sdu.dk phthy87@gmail.com
University of Southern Denmark

ABSTRACT
Recently there has been an increasing interest in building distributed platforms for processing of fast data streams. In this demonstration, we highlight the need for elasticity in distributed data stream processing systems and present Enorm, a data stream processing platform with focus on elasticity, i.e. the ability to dynamically scale resource usage according to the runtime workload fluctuations. In order to achieve dynamic scaling with minimal overhead and latency, we use an integrated approach for both fault-tolerance and elasticity. The idea is that both fault-tolerance and elasticity essentially require replicating or migrating computation states among different nodes. Integrating and sharing the state management operations between the two modules can not only provide abundant opportunities to reduce the system’s runtime overhead but also simplify the system’s architecture.

Categories and Subject Descriptors
H.2.4 [Database Management]: Systems

Keywords
Fault-Tolerance; Elasticity; MapReduce; Resource Management

1. INTRODUCTION
Conventional batch-based systems allow defining the set of computing units at job submission. In most cases that is acceptable as the job describes a finite computation and the resources required to process the job can be estimated beforehand.

State-of-the-art data stream systems are designed to execute long-standing jobs, whose input properties are usually hard to predict at job submission. This poses great challenges to the users of the system: the input rate might be fluctuating heavily over runtime and the user has to predict the needed resources for the job, which are then fixed.

Recently emerging systems are trying to solve the problem by allowing different kinds of elasticity between batches, e.g. Storm1 allows the end-user to pause execution after a batch and then change the set of processing units. Techniques like this incur excessive latency as all processing must be paused during reconfiguration. Furthermore, most existing work often do not explicitly support window operations and the maintenance of computation state at the system level. Instead the users are responsible for implementing them at the application level. In such a design, it is very difficult for the system to support autonomic scaling, as scaling often require the migration of computation state.

Enorm [7, 8], is a data stream processing platform with focus on elasticity, i.e. the ability to dynamically scale resource usage according to the runtime workload fluctuations. Enorm uses an integrated approach combining passive fault-tolerance and elasticity, to obtain faster state migration in certain cases. For the demonstration, we present
1. A live instance of enorm undergoing periodical elastic adaptations, scaling in/out on Amazon EC2.
2. A poster showing details of our system, which will be used to explain the behaviour exhibited by the live instance.

2. RELATED WORK
Elasticity in a multi-core system has previously been explored [10]. In their work, they modify the number of threads (spawn or kill) used for processing. Often this is achieved by using an abstraction to decouple the number of threads and the number of processing units (i.e. the executor abstraction of Storm). This technique constitutes internal scaling, as it focuses on elasticity in a multi-core system. The techniques discussed in this paper, works for both internal and external scaling, specifically by changing the number of computing units at runtime.

There are multiple issues with modifying the number of computing units directly. One of the issues when scaling a stateful partitioned operator, is that the partitioning depends on the number of instances. Changing the number of instances, thus requires repartitioning some state. A demonstration paper [1], discusses how to avoid this, by adding another operator to perform total aggregation of the partial results of the scaled operators.

Improving the operators to support intraoperator data partitioning at runtime, has been investigated by Shah et. al. [11]. Their work focuses strictly on load-balancing and not on adding/removing computation resources. Dynamic

1http://www.storm-project.net
ordering and placement of operators have also been studied in previous work, such as [12, 13]. All these works can be considered as part of the solution towards dynamic resource management.

There is recent work, on integrating passive fault-tolerance and elasticity [2]. A strategy for maintaining fault-tolerance while elastically scaling in/out is discussed. We are in many ways extending on this work, e.g. by employing a system model that allow us to place checkpoints anywhere and supports efficient load-balancing.

There is a recent research report [3], which focuses on how to design an adaptation algorithm to achieve elasticity in a distributed low-latency streaming system. In their work, they rely on two simple metrics (congestion and throughput), to determine how/when to scale. In enorm, we choose another strategy and employ an adaptation algorithm which works (simply put) by considering the average load of nodes (cpu, memory and bandwidth) over a period, then comparing the load to a set of thresholds defined by the user.

3. SYSTEM MODEL

3.1 Data Model

In enorm, the input data is modelled as a number of continuous streams of events and each event is a tuple $⟨id, key, value, ts_1, ts_2⟩$, where

- $id$ is a unique identifier of the event composed by the unique id of the stream that the event comes from and the per-stream unique sequence number of the event.
- $key$ is the key of the event, as a blob in arbitrary form, that is opaque to the system. It is the value of the key which decides how the tuple is routed, similar to other systems, e.g. storm or hadoop.
- $value$ is the value of the event as a blob in an arbitrary form that is opaque to the system.
- $ts_1$ and $ts_2$ are two timestamps associated with the event. We use two timestamps to identify the sliding windows that the data belongs to. Typically, if the event is a computation result over a sliding window, then $ts_1$ and $ts_2$ indicate the start and end points of the window, $ts_1$ and $ts_2$ can be identical which indicates that the tuple has only one timestamp.

For this work, only a subset of the enorm programming model is required. It consists of one function:

\[
\text{compute}(key, tuple) \rightarrow \text{list}(key, tuple)
\]

which takes a key-tuple pair and return a list of tuples. In addition, compute is associated with a local storage to keep track of the current state of the computation.

By considering only a subset of the programming model, it becomes logically similar to the MapUpdate framework [5]. For a description of the full framework, refer to [7, 8]

3.2 Query Model

A job can be considered a set of continuous queries, which can be represented as an operator network. An operator network is a directed acyclic graph $(O, E)$, where each vertex is an operator $O_i$ and each edge $E_j$ is a stream where the direction represents the direction of data flow. Note that here an operator does not necessarily correspond to a relational operator, as in most literature on data stream processing. Every query graph contains at least two special operators a source (src) and a sink (snk). Both these operators are assumed to be fault-tolerant.

3.3 Operator Model

An operator has a set of input streams $In_i$, and a set of output streams $Out_i$. On each incoming tuple, a function $F_i$ is invoked, which updates processing-state $σ_i$ and a map $m_{key→seq}$ which for each key processed by the operator contains the id of the latest tuple contributing to $σ_i$. The operator adds tuples to the set of output streams $Out_i$ as specified by the function. An operator $O_i$ is defined by function $F_i : (η(\text{In}_i), σ_i, m_{key→seq}) \rightarrow (ψ(\text{Out}_i), σ'_i, m'_{key→seq})$, where $η(\text{In}_i)$ returns the next incoming tuple and $ψ(\text{Out}_i)$ returns the set of resulting tuples.

3.4 Execution Model

An operator $O_i$ in the operator network, can be parallelized to several instances $O^j_i$, where $j$ denotes the operator instance id. Formally, an operator $O_i$, is modelled as

\[
O_i = \{O^j_i, o^1_j, o^2_j, ..., o^n_j\}, \text{ where } n \in N^+
\]

where $n$ describes the degree of parallelization of the operator $O_i$. $O_i$ is responsible for processing a set of independent keys $K_i$. The keys $K_i$ are partitioned into a set of groups, such that each group contains a non-overlapping set. The set of groups on $O_i$, is modelled as

\[
G_i = \{g^1_i, g^2_i, ..., g^p_i\}, \text{ where } p \geq n
\]

Every operator instance $O^j_i$ will handle a non-overlapping subset of $G_i$.

Based on the model above, the main assumption is that execution can be parallelized on a per key basis, which means the processing of groups are independent of one another. The relation between keys and groups, can vary from one-to-one to all-to-one. Each group $g^j_i \in G_i$, where $j$ denotes the group number, has an independent processing state $σ'_j$, and a corresponding checkpoint $ψ^j_i$.

A cluster has a set of nodes $N = \{n_1, ..., n_n\}$. Every node $n$ processes a non-overlapping subset of groups $G_n$, from all groups defined in the query network. Furthermore, each node stores a set of checkpoints $κ_n$.

Given that an operator has several input sources, obtaining a reproduceable ordering of input tuples is costly. We assume out-of-order processing, which means that operators produce the same result, given the same data as long as the unorderenedness is within some bounds [6].

4. FAULT-TOLERANCE

To provide fault-tolerance in a distributed streaming system, there are multiple well-known techniques, such as active backup, passive backup and upstream backup. Enorm uses a combination of upstream backup and passive backup.

Every tuple included in the processing state of an operator $O_i$ is kept in an upstream buffer until the processing state of $O_i$ no longer depends on it. If an operator fails, all tuples from the upstream buffer are replayed. As out-of-order processing is assumed, the same results are produced when the buffer is replayed. An operator must therefore, be aware of all received tuples that is part of its current state.
processing state such that it can dismiss tuples that have already been received from the crashed operator upstream.

The above mentioned technique is an example of pure upstream backup, which has a significant problem when used alone, as the size of the upstream buffer is not bounded. In the worst case, a single calculation might depend on all the data ever processed, which means, the buffer must also keep all the data, which can lead to performance issues. To handle this problem, upstream backup is often used in conjunction with passive backup.

Passive backup relies on checkpoints, which are simply copies of the current processing state. The system periodically checkpoints state, then stores the checkpoint on a separate node and lastly trims the upstream buffer. In case of a fault, the checkpoint can be fetched and then the upstream buffer can be replayed. For details of these techniques, we refer to [4].

5. ALLOCATING CHECKPOINTS

Given that checkpoints are replicas of state, they can be used for state migration in some cases. If a checkpoint used by passive backup, could also be used as basis for state movement, the non-cost of fault-tolerance could be used to speed up migration. For instance, let the checkpoints of an operator instance be spread out among other instances of the same operator. We argue the following benefits can be achieved:

1. When the state $\sigma_i^j$ is large, the time required to transfer the state over the network might be significant. It is possible avoid this cost, if moving to an operator instance $o_i^j$, which has the newest checkpoint of $\sigma_i^j$. The cost is to replay the upstream buffers to $o_i^j$.

2. When an operator instance $o_i^j$ fails, the operator instance that contains the checkpoint $cp_i^j$, can transform the checkpoint to a processing state efficiently, by replaying the upstream buffers.

The location of checkpoints is thus important, because the optimization described only can be employed when there is a suitable checkpoint available.

Runtime fluctuations can change the load of groups. A node $n$, has a set of groups $\varsigma_n$ and checkpoints $\kappa_n$. Fluctuations in groups $g \in \varsigma_n$ can heavily change the workload of node $n$. The intuition of our approach is to place a checkpoint of $\kappa_n$, on a node $m$, such that the correlation between $\varsigma_n$ and $\varsigma_m$ is negatively correlated. In this way, the probability that both nodes are overloaded at the same time is low. The checkpoint will thus be available for state migration, e.g. for load balancing an overloaded node.

The system periodically monitors the workload of each group and calculates the correlations among all the groups. In this work, we employ the Pearson product-moment correlation coefficient, which is a measure of the linear correlation.

6. ELASTICITY

The adaptive resource management component in enorm handles workload rebalancing and elastic scaling of operators. In previous work for data stream processing, these two problems are often handled separately [10, 11]. We argue that both problems are actually resource allocation problems and treating them independently may produce suboptimal or inconsistent solutions. For instance, if the load distribution among an operators parallel instances is unbalanced or it’s parallelization degree is not enough, the operator could become incapable of handling it’s input data. Therefore, a consistent decision has to be made by considering both problems in an integrative manner.

Furthermore, the handling of both problems often requires the same set of system operations. For example, rebalancing the load distribution among the parallel instances of an operator requires the system to move parts of the intermediate states and computations of the operator to another node. Such an operation is also required by the scaling of a operator. Therefore, we employ an integrative resource management scheme in enorm.

6.1 Adaptation Algorithm

In this section, the adaptation algorithm is briefly discussed, as large parts of the algorithm are straight-forward and other papers have already taken an indepth look on this kind of algorithm [3].

The adaptation algorithm is run relatively infrequently, because

1. Reacting to short-lived fluctuations can be more costly than beneficial. Any system must be able to cope with short-lived fluctuations in input, without scaling. We rely on a backup pressure system.

2. Gathering statistics over a longer period, makes the statistics more credible.

The adaptation algorithm uses three thresholds, a user-defined node-level threshold $NLT$, a upper global threshold $UGT$ and a lower global threshold $LGT$. The node-level threshold is a number from $[0 - 100]$, deciding the maximal load in percent on a node, before it is considered overloaded by the adaptation algorithm. The upper global threshold is defined by $UGT = |N| \times NLT$ and the lower global threshold is defined by $LGT = (|N| - 1) \times NLT$.

Each operator periodically sends statistics to a designated control-unit $c$, which is responsible for executing the adaptation algorithm. The statistics include the following information:

1. Tuples received per group
2. Size in bytes of state stored per group
3. Average load per group and local node

Load Balancing is executed if the sum of all loads on the set of nodes in the cluster, is in the range $[LGT, UGT]$ and one or more nodes are greater than $NLT$. The algorithm will move a subset of groups from overloaded to underloaded nodes, such that all nodes are loaded, less than the user-provided threshold.

Scaling In is executed if the sum of all loads on the set of nodes in the cluster is less than $LGT$. First it is determined, which set of nodes should be removed. Secondly all groups are reallocated and thirdly all checkpoints are reallocated, such that nothing is allocated to the set of nodes to remove. Lastly, when the new allocation have been effectuated, the nodes will be killed.

Scaling Out is executed if the sum of all loads on the set of nodes in the cluster is greater than $UGT$. A new set
of nodes will be started, and an instance of each operator in the job, will be placed on these new nodes. When the new nodes have been started, load balancing will be done. Lastly checkpoints are redistributed, using an algorithm that reflects the discussion in section 5.

7. STORY LINE

The demonstration consists of two parts: an introduction and a system demonstration. The introduction underlines the need for elasticity and shows the problems of the conventional techniques used to obtain elasticity. The system demonstration shows how enorm dynamically adapts itself to continuously fluctuating input. We provide a real-time visualization of the input and adaptation, in the form of graphs using Ganglia [9] with custom metrics.

Part 1: Introduction. A poster is used to highlight the importance of elasticity, and show the benefit of using an integrated approach with passive fault-tolerance. Then we present three case-studies, each a graph showing a fluctuating input-rate and how the system handles it. This provides the attendees with an understanding of how the adaptation algorithm can handle different types of fluctuations.

Part 2: System demonstration. The system demonstration is designed to show how enorm elastically adapts at runtime. We will present a simple job calculating multi-level aggregation (while continuously outputting partial results), and we will use synthetic data. This is chosen, as it allows us to show many different scenarios and allows the attendees to change attributes of the data as it is generated on the fly.

There is a graphical user-interface, allowing the attendees to change the input rate and skewness of data, on the fly.

There is a graphical user-interface, showing real-time information about the running job, statistics about performance and system wide information, such as processing latency. The interface is built using Ganglia, extended with custom metrics. Using this interface it is easy to get an overview of the system, but also to explore the performance in details as needed.

8. “TAKE-AWAY” MESSAGE

This demonstration highlights the need for elasticity in large-scale data stream processing systems and focuses on issues raised by realizing it, such as obtaining low latency migration and maintaining state. The demonstration will provide the attendees with an in-depth understanding of why it is beneficial to use an integrated solution combining passive fault-tolerance and elasticity to achieve these goals.

9. REFERENCES


