Scalable and Fault-tolerant Stateful Stream Processing*

Raul Castro Fernandez[1], Matteo Migliavacca[2], Evangelia Kalyvianaki[3], and Peter Pietzuch[1]

1 **Dept. of Computing, Imperial College London**
rc3011@doc.ic.ac.uk, prp@doc.ic.ac.uk
2 **Dept. of Computing, University of Kent**
mm53@kent.ac.uk
3 **School of Informatics, City University London**
evangelia.kalyvianaki@city.ac.uk

Abstract

As users of "big data" applications expect fresh results, we witness a new breed of stream processing systems (SPS) that are designed to scale to large numbers of cloud-hosted machines. Such systems face new challenges: (i) to benefit from the "pay-as-you-go" model of cloud computing, they must scale out on demand, acquiring additional virtual machines (VMs) and parallelising operators when the workload increases; (ii) failures are common with deployments on hundreds of VMs—systems must be fault-tolerant with fast recovery times, yet low per-machine overheads. An open question is how to achieve these two goals when stream queries include stateful operators, which must be scaled out and recovered without affecting query results.

Our key idea is to expose internal operator state explicitly to the SPS through a set of state management primitives. Based on them, we describe an integrated approach for dynamic scale out and recovery of stateful operators. Externalised operator state is checkpointed periodically by the SPS and backed up to upstream VMs. The SPS identifies individual operator bottlenecks and automatically scales them out by allocating new VMs and partitioning the checkpointed state. At any point, failed operators are recovered by restoring checkpointed state on a new VM and replaying unprocessed tuples. We evaluate this approach with the Linear Road Benchmark on the Amazon EC2 cloud platform and show that it can scale automatically to a load factor of L=350 with 50 VMs, while recovering quickly from failures.

1998 ACM Subject Classification H2.4 Database Systems. Systems

Keywords and phrases Stateful stream processing, scalability, fault tolerance

Digital Object Identifier 10.4230/OASIcs.ICCSW.2013.11

1 Introduction

In many domains, "big data" applications [2], which process large volumes of data, must provide users with fresh, low latency results. For example, web companies such as Facebook and LinkedIn execute daily data mining queries to analyse their latest web logs [8]; online marketplace providers such as eBay and BetFair run sophisticated fraud detection algorithms on real-time trading activity [7]; and scientific experiments require on-the-fly processing of data.

* A longer version of this paper appeared in the proceedings of ACM International Conference on Management of Data (SIGMOD) [4].

2013 Imperial College Computing Student Workshop (ICCSW'13).
Editors: Andrew V. Jones, Nicholas Ng; pp. 11–18

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Therefore *stream processing systems* (SPSs) have evolved from cluster-based systems, deployed on a few dozen machines [1], to extremely scalable architectures for big data processing, spanning hundreds of servers. Scalable SPSs such as Apache S4 [6] and Twitter Storm [12] parallelise the execution of stream queries to exploit *intra-query parallelism*. By scaling out partitioned query operators horizontally, they can support high input stream rates and queries with computationally demanding operators.

While mechanisms for scale out [10, 9] and fault tolerance [13, 11, 15] in stream processing have received considerable attention in the past, it remains an open question *how SPSs can scale out while remaining fault tolerant when queries contain* ***stateful operators***. Especially with recently popular stream processing models [6, 12] that treat operators as black boxes in a data flow graph, users rely on operators that have large amounts of state, which potentially depends on the complete history of previously processed tuples [3]. This is in contrast to, for example, window-based relational stream operators [1], in which state typically only depends on a recent finite set of tuples.

We make the observation that both scale out and failure recovery affect operator state, and therefore can be solved more efficiently using a single integrated approach. Our key idea is to externalise internal operator state so that the SPS can perform explicit operator **state management**. We then define a set of primitives for state management that allow the SPS to *checkpoint*, *backup*, *restore* and *partition* operator state. Based on these primitives, we describe an **integrated approach for scale out and recovery** of stateful operators in an SPS.

We evaluate how our approach scales out queries as part of a prototype SPS using closed and open loop workloads. We report the performance of the Linear Road Benchmark [3] on the Amazon EC2 cloud platform.

In summary, the paper makes the following contributions:

1. a description of operator state and management primitives to be used by an SPS;
2. an integrated approach for automatically scaling out bottleneck operators and recovery of failed operators based on managed operator state;
3. an experimental evaluation on a public cloud, showing that this approach can parallelise complex queries to a large number of VMs, while being resilient to failures.

Next we analyse the problem; §3 presents our state management technique; based on this, we introduce the integrated approach for scale out and recovery (§4); §5 provides experimental results; and we finish with conclusions (§6). For related work and further details on this paper we refer the reader to [4].

2 Problem Statement

We want to enable the deployment of SPSs on *infrastructure-as-a-service* (IaaS) clouds, such as Amazon EC2 and Rackspace, across hundreds of VMs. An SPS in a cloud setting must support the *automated* deployment and management of stateful streaming queries. In particular, this requires (i) the exploitation of *intra-query parallelism* to scale processing across VMs; (ii) the masking of *failures* for continuous processing; and (iii) adaptation to a *VM model*.

Stateful operators. Existing systems typically assume that operators are either stateless [6] or that state can be ignored when e.g. recovering operators [12]. While this simplifies the architecture of the SPS, it puts a considerable burden on developers when they need scalable and fault-tolerant stateful operators.

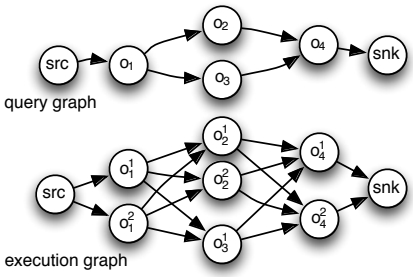

Figure 1 Example of query and execution graphs.

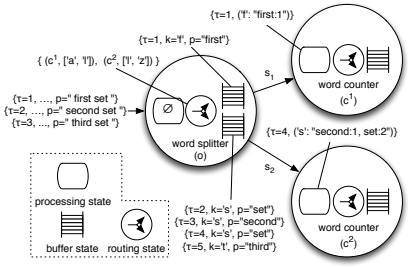

Figure 2 Different types of state in a stateful query for counting word frequencies.

Intra-query parallelism. Decisions about parallelising operators can occur *statically*—at query deployment time—or dynamically—at runtime. Static scale out requires knowledge of resource requirements of operators, which depend on stream rates and data distributions, and are typically estimated by cost models [14]. Therefore dynamic scale out is preferable in a cloud setting because the SPS can adapt to changes in the workload, observing resource consumption and VM performance.

Fault tolerance. Previous studies have shown that a substantial fraction of machines in large data centres develop faults during operation [5]. We assume a typical failure model, in which machine and network failures are modeled as independent, random crash-stop failures. Similar to other cloud-deployed applications, an SPS must be fault tolerant and cope with regular failures.

3 State Management

System Model

Data Model. A *stream* s is an infinite series of tuples $t \in s$. A *tuple* $t = (\tau, k, p)$ has a logical timestamp τ, a key field k and a payload p. The timestamp $\tau \in \mathbb{N}^+$ is assigned by a monotonically increasing *logical clock* of an operator when a tuple is created in a stream. Tuples in a stream are ordered according to their timestamps. Keys are not unique and used to partition tuples. They can be computed as a hash based on the payload.

Operator model. Tuples are processed by operators. An operator o takes n *input streams*, I_o, processes their tuples and produces one or more *output streams*, O_o.

An *operator function* f_o defines the processing of operator o on input tuples: $f_o : (I_o, \overline{\tau_o}, \theta_o, \overline{\sigma_o}) \rightarrow (O_o, \overline{\tau_o}, \theta_o, \overline{\sigma_o})$. A *stateful* operator has access to state θ_o, which is updated after processing. We assume that operators are deterministic and do not have other, externally visible side-effects. The timestamp $\overline{\sigma_o}$ specifies the oldest tuples that affected the state θ_o, i.e. the state depends only on tuples with timestamps $\overline{\sigma_{oi}} \leq \overline{\tau}_i \leq \overline{\tau_{oi}}$ for each input stream s_i.

Query model. As shown at the top of F. 1, a query is specified as a directed acyclic *query graph* $q = (\mathcal{O}, \mathcal{S})$ where $\mathcal{O}$ is the set of operators and $\mathcal{S}$ is the set of streams.

Query execution. A query is deployed on a set of *nodes*. A node can host multiple operators but, without loss of generality, we assume one operator per node. We distinguish between the logical representation of a query, in terms of its query graph, and its physical realisation, as shown at the bottom of F. 1. In the physical *execution graph* $\bar{q}$, an operator o may be parallelised into a set of *partitioned* operators $o^1 \ldots o^\pi$.

State Definition

The state of a query consists of the *operator state* of each query operator. We divide the operator state into *processing state*, *buffer state* and *routing state*, as illustrated in F. 2, which we use as a running example below.

Processing state. Output tuples from stateful operators depend on input tuples and the history of past tuples. Operators typically maintain an internal summary of this history of input tuples, which we term the operator's *processing state*. The current processing state θ_o of an operator o was computed from all past tuples with $\overline{\sigma_{oi}} \leq \overline{\tau}_i \leq \overline{\tau_{oi}} : s_i \in I_o$.

Exposing the processing state to the SPS has several reasons: (i) it enables the SPS to recover stateful operators more efficiently after failure. Instead of re-processing all tuples in the range $\overline{\sigma_{oi}} \leq \overline{\tau}_i \leq \overline{\tau_{oi}}$, recreating the processing state, the SPS can restore the state directly from a state checkpoint, and (ii) it allows the SPS to redistribute processing state across a set of new partitioned operators to support scale out.

In F. 2, we give an example of processing state for the word frequency operators.The upstream word split operator sends the word "first" to the word count operator c^1 at $\tau = 1$, resulting in the processing state $\theta_{c^1} = \{('f', \text{"first:1"})\}$ and timestamp $\overline{\tau_{c^1}} = (1)$. The words "set", "second" and "set" are processed by c^2, instead, which at $\overline{\tau_{c^2}} = (4)$ holds processing state $\theta_{c^2} = \{('s', \text{"second:1, set:2"})\}$.

Buffer state. An SPS typically interposes *output buffers* between operators, which buffer tuples before sending them to downstream operators (see F. 2). Buffers compensate for transient fluctuations of stream rates and network capacity.

Tuples in output buffers contribute to the query state managed by the SPS: (i) output buffers store tuples that have not yet been processed by downstream operators and therefore must be re-processed after failure; (ii) after dynamic operator scale out, tuples in output buffers must be dispatched to the correct partitioned downstream operator.

Routing state. An operator o in the query graph may correspond to multiple partitioned operators $o^1, \ldots, o^\pi$ in the execution graph. An upstream operator u has to decide to which partitioned operator o^i to route a tuple. Since the partitioning can change dynamically, an operator has explicit *routing state*, which must be restored after failure.

Operations

The above operator state can be manipulated by the SPS through a set of state management primitives.

Checkpoint state. The SPS can obtain a representation of the processing state θ_o and the buffer state β_o of an operator o in the form of a *checkpoint*. This is taken by the function checkpoint-state$(o) \rightarrow (\theta_o, \overline{\tau_o}, \beta_o)$. It obtains the processing state θ_o safely by calling the user-implemented function get-processing-state(), which also returns the timestamp $\overline{\tau_o}$ of the most recent tuples in the streams from the upstream operators that affected the state checkpoint. This permits the SPS to discard tuples with older timestamps, which are duplicates, during replay (see below).

The function checkpoint-state is executed asynchronously and triggered every *checkpointing interval* c, or after a user-defined event, e.g. when the state has changed significantly.

Backup state. The operator state, as returned by checkpoint-state, can be backed up to an upstream operator in anticipation of a restore or partition operation. After the operator state was backed up, already processed tuples from output buffers in upstream operators can be discarded because they are no longer required for failure recovery.

Restore state. Backed up operator state is restored to another operator to recover a

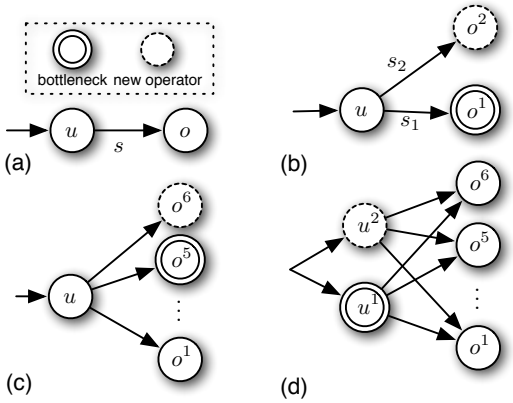

Figure 3 Example of scale out of stateful operators.

failed operator or to redistribute state across partitioned operators. A function takes the state to restore to operator o. It then initialises the processing state using a user-defined function and also assigns the buffer and routing states.

After the state was restored from a checkpoint, unprocessed tuples in the output buffer from an upstream operator are replayed to bring the operator o's processing state up-to-date. Before operator o emits new tuples, it resets its logical clock to the timestamp τ from the restored checkpoint so that downstream operators can detect and discard duplicate tuples.

Partition state. When a stateful operator scales out, its processing state must be split across the new partitioned operators. This is done by repartitioning the key space of the tuples processed by the operator (i.e. by doing consistent hashing). In addition, the routing state of its upstream operators must be updated to account for the new partitioned operators. Finally, the buffer state of the upstream operators is partitioned to ensure that unprocessed tuples are dispatched to the correct partition.

4 Scale Out and Fault Tolerance

Using the above state management primitives, we present our integrated approach for stateful operator scale out and recovery. We discuss our scaling strategy and fault tolerance, before describing our fault-tolerant scale out algorithm.

To scale out queries at runtime, the SPS partitions operators on-demand in response to bottleneck operators. Bottleneck operators prevent the system from increasing processing throughput. After scaling out a bottleneck operator, its processing load is shared among a set of new partitioned operators, thus increasing available resources to the SPS. Our scale out mechanism partitions operator state and streams without violating query semantics.

We give an example of operator scale out in F. 3, which shows four versions of an execution graph during scale out. When first deployed (F. 3a), the execution graph has one operator for each (logical) operator in the query graph. An operator o is connected through stream s to an upstream operator u. We assume that operator o is the bottleneck operator. F. 3b shows how the upstream operator u can partition its output streams into two streams. The two partitioned operators, o^1 and o^2, share the processing load and alleviate the bottleneck condition. In the same way, additional operators can be added to the execution graph for further scale out (F. 3c). When the upstream operator u becomes the new bottleneck (F. 3d), it is also partitioned and its output streams are replicated.

Even in the absence of bottlenecks, if a VM hosting a stateful operator fails, the SPS must replace it with an operator on a new VM. In our approach, overload and failure are

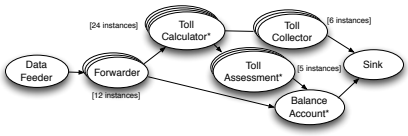

Figure 4 Query for the Linear Road Benchmark.

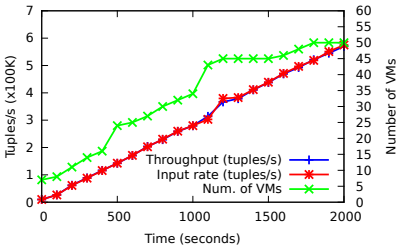

Figure 5 Dynamic scale out for the LRB workload with L=350 (closed loop workload).

handled in the same fashion. Operator recovery becomes a special case of scale out, in which a failed operator is scaled out to a parallelisation level of 1. This means that the SPS does not require a sophisticated failure detector to distinguish between the two cases but instead scales out an operator when it has become unresponsive.

5 Evaluation

The goals of our experimental evaluation are to investigate:
(i) the **effectiveness** of our **stateful operator scale out** approach for a *closed loop* workload.
(ii) the **recovery time** of the **stateful recovery** mechanism for a windowed word frequency query.
(iii) the **impact** of our **state management** approach on tuple processing latency.

The experiments are conducted using an experimental stream processing system implemented in Java. We deploy it on Amazon EC2 across 60 VMs.

Dynamic Scale Out

We first evaluate the effectiveness of our scale out approach when adapting to an increasing workload, i.e. when the SPS has to scale out to match an increasing input stream rate without tuple loss. The workload is the Linear Road Benchmark (LRB) (see [3] for details).

Our LRB query implementation consists of 7 operators, as shown in F. 4. We deploy the LRB query on Amazon EC2. Our deployment achieves a maximum L-rating of L=350 with 50 VMs. After that, the source and sink become the bottleneck, handling a maximum of 600,000 *tuples/s* due to serialisation overheads. The partitioned execution graph of the LRB is as shown in F. 4. We observe that the SPS maintains the required result throughput for the input rate, requesting additional VMs as needed. At times t=475 and t=1016, multiple operators are scaled out in close succession because bottlenecks appear in two operators simultaneously.

F. 6 shows processing latencies of output tuples, as a metric for the performance experienced by the query. The 99^{th} and 95^{th} percentiles of the latency are 1459 *ms* and 700 *ms*, respectively; the median is 153 *ms*, which are all below the LRB target of 5 *s*. This confirms that our maximum L-rating is indeed due to the limited source and sink capacities.

Failure Recovery

To evaluate failure recovery, we first compare recovery time against other fault tolerance approaches, *upstream backup* (UB) and *source replay* (SR). UB buffers tuples in each operator

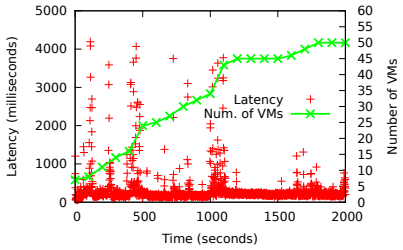

Figure 6 Processing latency for LRB workload.

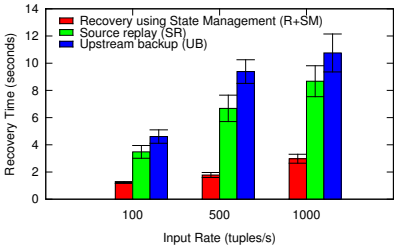

Figure 7 Recovery time for different fault tolerance mechanisms.

and re-processes them to recover operator state. SR is a variant of UB, in which tuples are only buffered and replayed by the source [12]. We use a query that counts word frequencies over a 30 *s* window.

We observe the recovery times for the three approaches. For R+SM, we set the checkpointing interval *c* to 5 *s*. During the experiment, we fail the VM and measure the time to recover (i.e. until the complete operator state was restored).

F. 7 shows results averaged over 10 runs for different input rates. SR achieves slightly faster recovery than UB because of the short length of the operator pipeline and the fact that it stops the generation of new tuples during the recovery phase. R+SM achieves lower recovery times than both UB and SR. Due to the state checkpoints, it re-processes fewer tuples to recover the stateful operator.

In F. 8, we show the change in recovery time as a function of the checkpointing interval for different input rates. Recovery time increases with longer checkpointing intervals because more tuples are replayed. Tuple buffering is the main factor determining recovery time, which is why recovery time increases considerably with higher rates. While frequent checkpointing incurs overhead, it reduces recovery time, even for high rates.

State Management Overhead

The overhead on processing throughput could not be observed, so we measure its effect on tuple processing latency.

We synthetically vary the state size (in this case a dictionary) between *small* (10^2 entries; ≈2 *Kb*), *medium* (10^4 entries; ≈200 *Kb*) and *large* (10^5 entries; ≈2 *Mb*).

F. 9 shows that the 95$^{\text{th}}$ percentile of tuple processing latencies increases with state size. For large state sizes, checkpointing takes longer and occupies more CPU time, which is unavailable for tuple processing. Higher input rates increase the load on the operator, resulting in less headroom for the checkpointing process. For input rates of 100 and 500 *tuples/s*, the latency remains small but grows for 1000 *tuples/s*.

6 Conclusions

We presented an integrated approach for scale out and failure recovery through explicit state management of stateful operators. Our approach treats operator state as an independent entity, which can be checkpointed, backed up, restored and partitioned by the SPS. Based on these operations, the SPS can support dynamic scale out of operators while being fault tolerant.

Our results show that our approach can be used effectively to provision Amazon EC2 resources against increasing input rates in the Linear Road Benchmark and also support

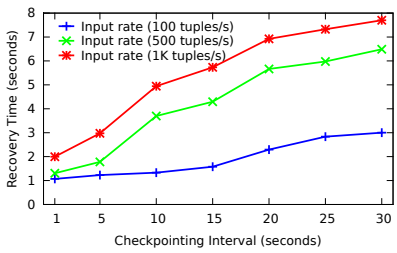

Figure 8 Recovery time for different R+SM checkpointing intervals.

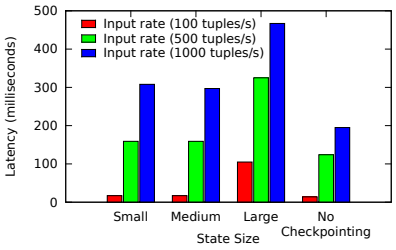

Figure 9 Overhead of state checkpointing for different input rates and state sizes

open loop workloads. Despite the state checkpointing, processing latency remains within desired levels.

As future work, we plan to extend our scale out policy with support for scale in to enable truly elastic deployments of cloud-based SPSs.

Acknowledgements This work was supported by a PhD CASE Award funded by the Engineering and Physical Sciences Research Council (EPSRC) and BAE Systems.

References

1 Daniel J Abadi, Y Ahmand, et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

2 D Agrawal, S Das, et al. Big Data and Cloud Computing: Current State and Future Opportunities. In *EDBT*, 2011.

3 Arvind Arasu, Mitch Cherniack, et al. Linear Road: A Stream Data Management Benchmark. In *VLDB*, 2004.

4 Raul Castro Fernandez, Matteo Migliavacca, et al. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *SIGMOD*, 2013.

5 Phillipa Gill, Navendu Jain, et al. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *SIGCOMM*, 2011.

6 L Neumeyer, B Robbing, et al. S4: Distributed Stream Computing Platform. In *ICDMW*, 2010.

7 Nish Parikh and Neel Sundaresan. Scalable and Near Real-Time Burst Detection from eCommerce Queries. In *SIGKDD*, 2008.

8 M Russell. *Mining the Social Web*. O'Reilly, 2011.

9 Benjamin Satzger, Waldemar Hummer, et al. Esc: Towards an Elastic Stream Computing Platform for the Cloud. In *IEEE CLOUD*, 2011.

10 Scott Schneider, Henrique Andrade, et al. Elastic Scaling of Data Parallel Operators in Stream Processing. In *IPDPS*, 2009.

11 Zoe Sebepou and Kostas Magoutis. CEC: Continuous Eventual Checkpointing for Data Stream Processing Operators. In *DNS*, 2011.

12 Twitter Storm. `github.com/nathanmarz/storm/wiki`.

13 Rohit Wagle, Henrique Andrade, et al. Distributed Middleware Reliability and Fault Tolerance Support in System S. In *DEBS*, 2011.

14 Erik Zeitler and Tore Risch. Massive Scale-out of Expensive Continuous Queries. *VLDB Endowment*, 4(11), 2011.

15 Zhe Zhang, Yu Gu, et al. A Hybrid Approach to HA in Stream Processing Systems. In *ICDCS*, 2010.