Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management

Raul Castro Fernandez
Imperial College London
rc3011@doc.ic.ac.uk

Matteo Migliavacca
University of Kent
mm53@kent.ac.uk

Evangelia Kalyvianaki
Imperial College London
ekalyv@doc.ic.ac.uk

Peter Pietzuch
Imperial College London
prp@doc.ic.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.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

Keywords

Stateful stream processing, scalability, fault tolerance

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 [25]; online marketplace providers such as eBay and BetFair run sophisticated fraud detection algorithms on real-time trading activity [24]; and scientific experiments require on-the-fly processing of data.

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 [23] and Twitter Storm [29] 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.

A cloud computing model offers SPSs access to a virtually unlimited number of virtual machines (VMs). To gain widespread adoption, however, cloud-hosted systems must hide the complexity of data parallelism and failure recovery from users, as evidenced by the popularity of parallel batch processing systems such as MapReduce. Therefore cloud-hosted SPSs face the same two fundamental challenges:

1. On-demand parallelism. To reduce financial costs under "pay-as-you-go" pricing models in public cloud environments such as Amazon EC2 and Rackspace, an SPS should acquire resources *on demand*. It should request additional VMs at runtime, reacting to changes in the processing workload and repartitioning query operators accordingly.

2. Resource-efficient failure recovery. A cloud-deployed SPS with hundreds of VMs is likely to suffer from failure. It must therefore be fault-tolerant, i.e. be able to recover from failures without affecting processing results. Due to the size of deployments, the per-machine resource overhead of any fault tolerance mechanism should be low. Since failures are common, recovery must not affect performance adversely.

While mechanisms for scale out [27, 26] and fault tolerance [30, 28, 33] 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 [23, 29] 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 [5]. 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.

When scaling out large stateful operators at runtime, operator state must be partitioned correctly across a larger set of VMs. Existing techniques either ignore this problem by assuming stateless operators [23, 29] or restrict support to a few relational operators [15]. Similarly, to recover stateful operators, an SPS must restore operator state after failure to maintain correct results. *Active* operator replication incurs a prohibitively high resource cost by doubling the number of required VMs, whereas *passive* replication has been shown to lead to high recovery times, and upstream backup techniques cannot efficiently recover operators whose state depends on the complete history of processed tuples [8].

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**. The SPS (i) obtains access to operator state through a well-defined interface; (ii) maintains information about the precise set of tuples that have been processed by an operator and are thus reflected in its state; and (iii) assumes a stream data model with a key attribute in order to be able to partition state with correct semantics. 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. In our approach, the SPS checkpoints state periodically and backs it up to VMs hosting upstream operators. Tuples that are not yet reflected in a checkpoint are buffered by upstream operators until included in a checkpoint.

(i) To scale out stateful operators, the SPS monitors queries for bottleneck operators and, according to a scale out policy, parallelises individual operators at runtime. The upstream VM with the checkpointed state of the bottleneck operator requests the allocation of extra VMs and deploys new partitioned operators, alleviating the bottleneck. To reduce the impact of delays when provisioning new VMs, the SPS maintains a VM pool of pre-allocated VMs for fast scale out.

(ii) To recover a failed stateful operator, the upstream VM with the most recent checkpointed state of the failed operator requests a new VM and restores the failed operator from that checkpoint. Since unprocessed tuples were buffered by the upstream operator, they can be replayed to bring the restored state up-to-date. To speed up operator recovery, the failed operator may be scaled out before recovery.

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 [5] on the Amazon EC2 cloud platform. Our results show that the system can scale to a load factor of L=350 using 50 VMs—the second highest result reported in the literature. It automatically partitions individual operators at a fine granularity, with little impact on processing latency, while recovering large stateful operators within seconds.

In summary, the paper makes the following contributions:

1. a description of operator state and associated 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 discusses integration with an SPS; §6 provides experimental results; and we finish with related work (§7) and conclusions (§8).

2. BACKGROUND

2.1 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. Stream processing operators can be *stateless* (e.g. filter or map) or *stateful* (e.g. join or aggregate). Relational stream models [22] use the concept of a finite *window* of tuples to define the current state of an operator. Sliding windows in such models encourage the use of operators with small amounts of state, which only depends on the last few processed tuples. More recently, streaming data flow graph models [29, 23] have gained in popularity, especially as a way for incrementally processing large datasets. They permit users to implement black-box operators (e.g. a frequency counter) that maintain arbitrary state, potentially depending on the entire history of the data stream.

Such stateful operators with state, which cannot easily be recreated by re-processing a small section of the input stream, are not well supported by current SPSs. Existing systems typically assume that operators are either stateless [23] or that state can be ignored when e.g. recovering operators [29]. While this simplifies the architecture of the SPS, it puts a considerable burden on developers when they need scalable and fault-tolerant stateful operators.

Intra-query parallelism. A cloud-hosted SPS can improve the processing throughput of computationally expensive queries by parallelising operators, which would otherwise become a bottleneck. Existing systems have identified this requirement and include mechanisms to indicate the potential for operator-level parallelism [23, 27, 29].

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 [32]. 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.

To scale out dynamically, the SPS must identify the bottleneck operator, limiting processing throughput of a query. This is challenging for complex query graphs, in which operator performance may be compute or network-bound. The SPS must have a *scale out policy* that reacts to bottleneck operators based on observable system and query metrics. In addition, it requires the reconfiguration of the query exe-

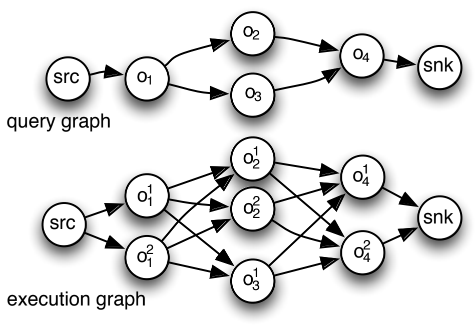

Figure 1: Example of query and execution graphs

cution graph at runtime without affecting the correctness of results. This is simple for stateless operators: the SPS can create new operator instances and split the streams. For stateful operators, however, operator state must also be split across partitioned instances based on operator semantics, e.g. by join key and table tag when using an *improved repartition join* [9].

Fault tolerance. Previous studies have shown that a substantial fraction of machines in large data centres develop faults during operation [12]. 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.

After the loss of a VM with operators, stream processing must resume from the point at which the operators failed. Recovering a stateful operator is challenging because it only produces correct output with the correct state. Operator state that depends on a finite subset of a stream (e.g. for a windowed stream join) can be reconstructed using upstream backup [7] by replaying past tuples. However, this increases recovery times and becomes infeasible for operators whose state depends on all past tuples.

Due to the large number of involved machines in a cloud-hosted SPS, the per-node resource overhead of any fault tolerance mechanism must be low. Active replication strategies are therefore impractical because they typically double resource requirements, thus substantially increasing the cost of a cloud deployment. Passive replication has a lower resource overhead, making it more applicable.

In addition, to reduce the impact of failure, fast recovery is important. Failure recovery incurs an additional resource demand on top of regular processing by the SPS. For example, processing performance may be reduced when reprocessing past tuples during recovery [29].

VM deployment. A cloud-hosted SPS must adapt to the specifics of an IaaS cloud model. In particular, IaaS platforms are known to exhibit delays on the order of minutes when provisioning new VMs. When a cloud-hosted SPS scales out at runtime, this may lead to unpredictable performance, e.g. incurring a period of degraded throughput due to overload until a VM becomes available.

2.2 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 (see §3.1). 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*, denoted by the set $I_o = \{s_1, \dots, s_n\}$, processes their tuples and produces one or more *output streams*, O_o. For ease-of-presentation, we assume that an operator emits only a single output stream (unless the downstream operator is partitioned). The notation $I_o[\overline{\tau}]$ specifies all tuples in the input streams with timestamps less than $\overline{\tau}$ where $\overline{\tau} = (\tau_1, \dots, \tau_n)$ are the timestamps of the individual input streams.

An *operator function* f_o defines the processing of operator o on input tuples: $f_o : (I_o, \overline{\tau_o}, \theta_o, \overline{\sigma_o}) \rightarrow (O_o, \overline{\tau_o}, \theta_o, \overline{\sigma_o})$. At each invocation of f_o, the operator accepts a finite set of tuples $I_o[\overline{\tau_o}]$ where $\overline{\tau_o}$ are the timestamps of the most recent tuples already processed. After processing, the operator advances to new positions in the input streams and updates the value of $\overline{\tau_o}$. 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 $\overline{\sigma_o}_i \leq \overline{\tau}_i \leq \overline{\tau_o}_i$ for each input stream s_i. A *stateless* operator has $\theta_o = \emptyset$.

Query model. As shown at the top of Fig. 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. A stream $s \in \mathcal{S}$ is a directed edge between two operators, $s = (o, o')$ where $\{o, o'\} \subseteq \mathcal{O}$. A query has two special operators, *src* and *snk*, which act as the sources and sinks for data streams, respectively. We assume that sources and sinks cannot fail. An operator u is *upstream* to o, denoted by $u \in up(o)$, when $\exists (u, o) \in \mathcal{S}$. Similarly, an operator d is *downstream* to o, $d \in down(o)$, when $\exists (o, d) \in \mathcal{S}$.

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 Fig. 1. In the physical *execution graph* $\bar{q}$, an operator o may be parallelised into a set of *partitioned* operators $o^1 \dots o^\pi$. The value $\pi \in \mathbb{N}^+$ is the *parallelisation level* of o. Each o^i implements the semantics of o. It takes as input a partitioned stream $s^1 \dots s^\pi$, which is obtained from the original stream s. The current execution graph is maintained by a logically centralised *query manager.*

3. STATE MANAGEMENT

Next we describe our approach for externalising and managing operator state. State in stream processing is typically considered an implementation detail [1] and only managed for specific purposes, such as persistence [30] or overload handling [19]. In contrast, we (i) make operator state externally visible to the SPS and (ii) define primitives for the SPS to manage state in a generic fashion. Based on these primitives, the SPS supports more complex operations such as scale out and failure recovery, which affect operator state.

3.1 Query State

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 Fig. 2. The figure shows a query with three operators for collecting word frequencies in a text stream every minute: a stateless *word split* operator o, which tokenises a stream of strings into words, and two partitioned stateful *word count* operators c^1 and c^2, which maintain a windowed frequency count of words. We use the query 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, as described in §3.2; and (ii) it allows the SPS to redistribute processing state across a set of new partitioned operators to support scale out.

Based on our data model, we define the processing state of an operator o as a set of key/value pairs, $\theta_o = \{(k_1, v_1), \ldots\}$. Each key k is unique and refers to the corresponding tuple keys from the input streams (see §2.2). Its associated value v stores the portion of processing state that the operator requires when processing tuples with that key. In addition, the processing state is associated with a vector of timestamps $\overline{\tau_o}$, as returned by the operator function f_o. It specifies the most recent timestamps of input tuples that are reflected in θ_o.

An operator can maintain state using efficient data structures internally and only translate it to key/value pairs when requested by the SPS. To expose its processing state, the developer of an operator o implements a function get-processing-state$(o) \rightarrow (\theta_o, \overline{\tau_o})$. It is invoked by the SPS and takes a copy of the state. It locks all internal operator data structures to obtain a consistent copy and records the timestamp $\overline{\tau_o}$ of the most recent tuples that affected the state.

In Fig. 2, we give an example of processing state for the word frequency operators. To simplify presentation, keys are assumed to be the first letter of a word. The upstream word split operator sends the word "first" to the word count operator c^1 at $\tau = 1$, resulting in the processing state $\theta_{c^1} = \{(\text{'f'}, \text{"first:1"})\}$ 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 $\theta_{c^2} = \{(\text{'s'}, \text{"second:1, set:2"})\}$.

Buffer state. An SPS typically interposes *output buffers* between operators, which buffer tuples before sending them to downstream operators (see Fig. 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.

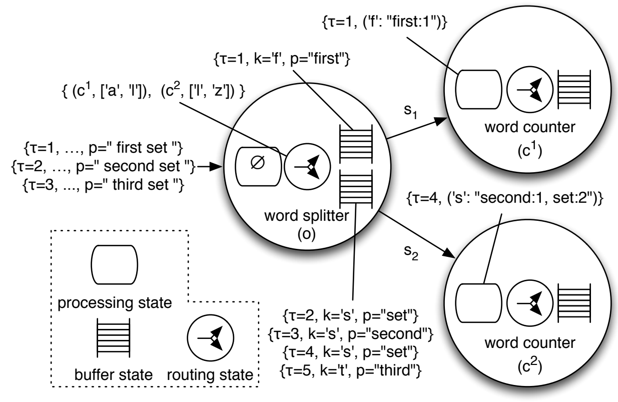

Figure 2: Different types of state in a stateful query for counting word frequencies

We model the output buffer of operator o to a partitioned downstream operator d as having *buffer state* $\beta_o = \{(d^1, \{t_1, \ldots\}), \ldots\}$ with $t_1 \in (o, d^1)$ (see §2.2). It stores a finite number of past output tuples. The notation $\beta_o(d^i)$ refers to the output tuples for a partitioned downstream operator d^i.

Tuples in an output buffer are discarded after they are no longer needed for recovery (see §4.2). An operator trims tuples from an output buffer by removing tuples with timestamps older than τ when it executes the function trim(o, τ).

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.

For an operator o, we define the routing state as $\rho_o = \{(d^1, [k_1, k_2]), \ldots, (d^\pi, [k_{\pi-1}, k_\pi])\}$, which maps the keys $k \in [k_i, k_j)$ to a partitioned downstream operator d^i. For example, the word split operator in Fig. 2 has $\rho_o = \{(c^1, [\text{'a'}, \text{'l'}]), (c^2, [\text{'l'}, \text{'z'}])\}$. It sends words starting with letters up to 'l' to operator c^1 and with letters from 'l' to c^2.

3.2 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. A short checkpointing interval results in a smaller number of tuples that must be re-processed to bring the process-

Algorithm 1: Operator state backup and restore

```
   operator identifier function: $id : \mathcal{O} \rightarrow \mathcal{N}$,
   upstream op. of $o$: $up(o) = \{o_1, \ldots, o_i, \ldots, o_m\}$
   previous backup operator: $backup(o) = o_j$ or $\perp$ if undef.
   function backup-state($o$)
 1     $(\theta_o, \overline{\tau_o}, \beta_o) \leftarrow$ checkpoint-state(o)
 2     $i = hash(id(o)) \bmod |up(o)|$
 3     store-backup($o_i, o, \theta_o, \overline{\tau_o}, \beta_o$)
 4     for $u$ in $up(o)$ do trim($u, \overline{\tau_{oj}}$) : $s_j = (u, o)$
 5     if $backup(o) \neq \perp \wedge backup(o) \neq o_i$ then
 6         delete-backup($backup(o), o$)
 7     $backup(o) \leftarrow o_i$
   function restore-state($o, \theta, \overline{\tau}, \beta, \rho$)
 8     set-processing-state($o, \theta, \overline{\tau}$)
 9     $\beta_o \leftarrow \beta, \rho_o \leftarrow \rho$
   function replay-buffer-state($u, o$)
10     for $t$ in $\beta_u(o)$ do send o: t
```

ing state up-to-date, but it incurs a higher overhead. Note that the checkpointing interval should be shorter than the window size of the operator. If checkpoints are taken more rarely, they contain processing state that is superseded by tuples that must be re-processed. To reduce the size of checkpoints, it is also possible to use incremental checkpointing techniques [17].

Note that routing state is not included in the state checkpoint because it only changes in case of scale out or recovery and not during regular tuple processing. Instead, routing state is maintained by the query manager (see §2.2).

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.

When an upstream operator has many downstream operators that use it for state backups, the backup operation incurs significant overhead, which may result in a scale out of the upstream operator. In that case, operators should balance the backup load across all of their partitioned upstream operators.

Algorithm 1 defines function backup-state(o) for backing up the state of operator o. First, the SPS creates a checkpoint (line 1). It then selects a backup operator $backup(o)$ among the upstream operators of o. The backup load is spread among all upstream operators by using a hash function (line 2). The operator state is backed up in line 3. After the state backup, the output buffers of the upstream operators are trimmed (line 4). If the upstream operators are repartitioned, the choice of $backup(o)$ may change. If the new backup operator is different from the previous one (line 5), the old backup operator is released (line 6).

Restore state. Backed up operator state is restored to another operator to recover a failed operator or to redistribute state across partitioned operators. The function restore-state($o, \theta, \overline{\tau}, \beta, \rho$), described in Algorithm 1, takes the state to restore to operator o. It then initialises the processing state using the function set-processing-state (line 8) and also assigns the buffer and routing states (line 9).

After the state was restored from a checkpoint, the function replay-buffer-state(u, o) is used to replay unprocessed

Algorithm 2: Operator state partitioning

```
 1 key interval : $(k_l, k_h)$ : $(o, [k_l, k_h]) \in \rho_u \wedge u \in up(o)$
 2 key split : $(k_1, \ldots, k_\pi)$ : $k_l = k_1 < \ldots < k_{\pi+1} = k_h$
   function partition-processing-state($o, \pi$)
 3     $(\theta, \overline{\tau}, \beta) \leftarrow$ retrieve-backup($backup(o), o$)
 4     for $i \leftarrow 1$ to $\pi$ do
 5         $\theta_i \leftarrow \{(k, v) \in \theta_i : k_i \leq k < k_{i+1}\}$
 6         $\overline{\tau_i} \leftarrow \overline{\tau}$
 7         if $i \neq 1$ then $\beta_i \leftarrow \emptyset$ else $\beta_1 \leftarrow \beta$
 8         store-backup($backup(o^i), o^i, \theta_i, \overline{\tau_i}, \beta_i$)
   function partition-routing-state($u, o, \pi$)
 9     $\rho_u \leftarrow \rho_u \setminus \{(o, [k_l, l_h])\}$
10     for $i \leftarrow 1$ to $\pi$ do
11         $\rho_u \leftarrow \rho_u \cup \{(o^i, [k_i, k_{i+1}])\}$
12     store-routing-state($u, \rho_u$)
   function partition-buffer-state($u$)
13     for $(o, T)$ in $\beta_u$ do
14         for $t = (\tau, k, p)$ in $T$ do
15             for $(o', [k_1, k_2])$ in $\rho_u$ do
16                 if $k_1 \leq k < k_2$ then $\beta(o') \leftarrow \beta(o') \cup \{t\}$
17     $\beta_u \leftarrow \beta$
```

tuples in the output buffer from an upstream operator u 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. 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.

In Algorithm 2, we define the function partition-processing-state(o, π), which partitions the state of operator o for π new partitioned operators $o^1, \ldots, o^\pi$. The partitioning is performed from the state saved by $backup(o)$ to allow partitioned operators to recover in case of failure. First, the key range processed by o, as specified by the routing state of the upstream operator u, is split into π intervals (lines 1–2). The key space can be distributed evenly using *hash partitioning*, or the key distribution can be used to guide the split. The operator state is retrieved from $backup(o)$ (line 3) and is split by partitioning the processing state (line 5). The timestamps associated with the processing state are copied for each partition, and the buffer state is assigned to the first partition (lines 6–7). Finally, the operator state for each partition is stored in $backup(o^i)$ in order to provide an initial backup for each partition; afterwards $backup(o)$ is removed safely from the system (line 8).

The function partition-routing-state(u, o, π) is used to update the routing state of each upstream operator u. The entry for the old key interval is removed and key intervals for the new partitioned operators are added (lines 9–11). The routing state is then stored at the query manager to be recovered in case of failure (line 12).

An upstream operator u can repartition its buffer state β_u according to the updated routing state ρ_u using the function partition-buffer-state(u). It iterates over the tuples in

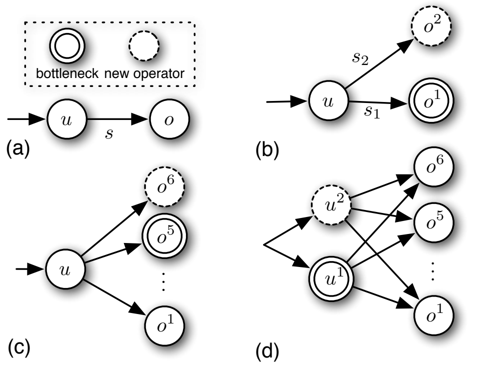

Figure 3: Example of scale out of stateful operators

β_u (lines 13–14), assigning each tuple to a partition according to the ρ_u key intervals (lines 15–16).

3.3 Discussion

As shown in the next section, the above state management primitives are a minimum set, which lets an SPS perform scale out and failure recovery in an integrated manner. Other state management primitives, however, can be added to cover a range of previously proposed functionality:

For example, to scale in operators when resources are under-utilised, the state of two operators can be *merged* [15]. For operators with large state sizes, a *spill* operation can temporarily store state on disk, freeing memory resources [19]. More generally, part of the operator state can be supported by external storage through a *persist* operation, e.g. when combining real-time data processing with historical data [3].

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.

4.1 Scale Out

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. We discuss heuristics for identifying the bottleneck in a query in §5.1. 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 Fig. 3, which shows four versions of an execution graph during scale out. When first deployed (Fig. 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. Fig. 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 (Fig. 3c). When the upstream operator u becomes the new bottleneck (Fig. 3d), it is also partitioned and its output streams are replicated.

Algorithm 3: Integrated fault-tolerant scale out

```
   function scale-out-operator($o, \pi$)
1      partition-processing-state($o, \pi$)
2      $\rho \leftarrow$ retrieve-routing-state($o$)
3      for $i \leftarrow 1$ to $\pi$ do
4          $o^i \leftarrow$ get-new-VM-with-operator()
5          $(\theta_i, \overline{\tau_i}, \beta_i) \leftarrow$ retrieve-backup($backup(o), o^i$)
6          restore-state($o^i, \theta_i, \overline{\tau_i}, \beta_i, \rho$)
7          for $d$ in $down(o)$ do replay-buffer-state($o^i, d$)
8      stop-operator-and-release-VM($o$)
9      for $u$ in $up(o)$ do
10         stop-operator($u$)
11         partition-routing-state($u, o, \pi$)
12         partition-buffer-state($u$)
13         for $i \leftarrow 1$ to $\pi$ do replay-buffer-state($u, o^i$)
14         start-operator($u$)
```

4.2 Fault Tolerance

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 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.

Operator recovery puts a strain on processing throughput because the SPS must restore operator state and replay tuples missing from the recovered state. In our approach, we can reduce recovery times by restoring a failed operator using several new partitioned operators, i.e. performing *parallel recovery*. Parallel recovery improves the recovery speed because it effectively adds extra resources to the SPS for failure recovery.

4.3 Fault-Tolerant Scale Out Algorithm

The SPS maintains two versions of operator o's state, which could be partitioned for scale out: the current state, maintained by o, and a recent state checkpoint stored by operator $backup(o)$. In our approach, the SPS partitions the most recent state checkpoint, which has multiple benefits: (i) it means that the scale out mechanism can be used to recover operator o after failure; (ii) it avoids adding further load to operator o, which is already overloaded, by requesting it to checkpoint or partition its own state; and (iii) it makes the scale out process itself fault-tolerant: if it fails or is aborted, operator o can continue processing unaffected.

Algorithm 3 shows the steps for scaling out operator o to a parallelisation level π. First, the SPS executes the function partition-processing-state to partitions o's processing state located on $backup(o)$, backing it up again to survive failure (line 1). It also retrieves o's routing state from the query manager (line 2). It then creates π new partitioned operators o^i to replace o (line 4). After o's processing and buffer state were retrieved from $backup(o)$ (line 5), they are restored to the new partitioned operators o^i (line 6). Tuples yet unprocessed by the downstream operators are replayed from o^i's buffer state (line 7). After that, o is stopped and its VM is released (line 8).

The next step is to update the execution graph. The SPS first stops o's upstream operators (line 10), updates their routing state by partitioning it (line 11) and then reparti-

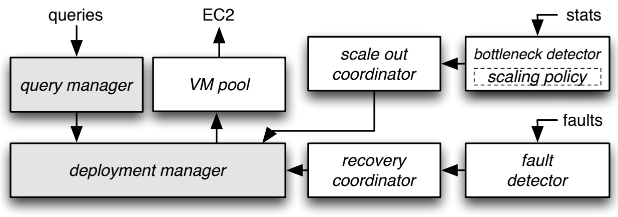

Figure 4: Integration with stream processing system

tions their buffer states (line 12). After that, the SPS replays tuples that are not reflected in the state checkpoint from the output buffers (line 13). The final step is to restart the upstream operator (line 14).

The same algorithm is used to recover from the failure of a stateful operator o by executing scale-out-operator$(o, 1)$ because it restarts processing from o's last checkpoint, without o being operational. To reduce recovery time using parallel recovery, the SPS can also partition the failed operator by executing scale-out-operator$(o, 2)$. In this case, each restored operator only has to process half of the replayed tuples in replay-buffer-state, thus reducing recovery time (see §6.2).

Discussion. The above algorithm assumes that a state checkpoint is available at operator $backup(o)$. As a consequence, the SPS cannot scale out if $backup(o)$ fails before the state checkpoint is partitioned and restored across the new partitioned operators successfully (i.e. lines 1-6). In that case, the scale out is aborted and retried after $backup(o)$ has recovered and o has again backed up its operator state. Therefore the VM hosting o is only released after restore-state has completed on all partitioned operators (line 8).

If any upstream operator u—other than $backup(o)$—fails, aborting the scale out process is unnecessary. After operator u has recovered, its output buffer is replayed to the partitioned operators according to the updated routing state retrieved from the query manager (line 2), as stored by the non-failed $backup(o)$ (line 12 in Algorithm 2). If one of the partitioned operators o^i fails, it can be recovered from the checkpoint stored at $backup(o^i)$ (line 8 in Algorithm 2).

5. STREAM PROCESSING IN A CLOUD

In this section, we describe how our integrated approach for stateful operator scale out and recovery can be realised in a cloud-hosted SPS. This entails several challenges: the SPS must (i) have heuristics for identifying bottleneck operators; (ii) have a policy as to when to scale out operators; and (iii) handle the delay when new VMs are provisioned.

We add our fault-tolerant scale out approach to an experimental distributed SPS developed in our group. Fig. 4 shows the architecture of the system after integration. A query graph is submitted to a *query manager*, which performs a mapping of query operators to VMs, to obtain an execution graph. The execution graph is used by a *deployment manager* to initialise VMs, deploy operators, set up stream communication and start processing.

To support dynamic scale out, we add a *bottleneck detector* that, based on system statistics, identifies the bottleneck operators in the query, as explained in the next section. According to a *scaling policy*, it invokes the *scale out coordinator*, which implements scale out decisions. For fault recovery, a *failure detector* notifies the *recovery coordinator* to recover a failed operator. The *deployment manager* is extended so that it can request new VM instances from a *VM pool*. The VM pool masks the delay of cloud platforms when provisioning new VM instances, as described in §5.2.

5.1 Bottleneck Detection and Scaling Policy

When detecting operator bottlenecks, we focus on compute bottlenecks because they are the most common type observed in practice. In particular, they limit the scalability of the Linear Road Benchmark used in §6.1.

We adopt a simple yet effective *scaling policy* based on measured CPU utilisation of operators. Every r seconds, VMs hosting operators submit CPU utilisation reports to the bottleneck detector, which record the user and system CPU time that each operator executed. This accounts for "stolen" CPU time when other VMs on the same physical node were scheduled. When k consecutive reports from an operator are above a user-defined threshold δ, the bottleneck detector notifies the scale out coordinator to parallelise the operator. Empirically, we determined that collecting CPU utilisation reports every r=5 s and scaling out after k=2 consecutive measurements are above δ=70% utilisation of the CPU time slice leads to appropriate scaling (see §6.1).

5.2 Virtual Machine Pool

The time taken to deploy a new operator is a critical issue. When scaling out, VMs must be allocated quickly in order to reduce the duration of an overload condition. When recovering a failed VM, fast recovery minimises the disruption caused by the failure. Current IaaS cloud platforms, however, require on the order of minutes to provision new VM instances, as confirmed by our own empirical experience. This makes it impractical to request new VM instances on-demand when they are required by an SPS.

Our solution is to decouple the request for a new VM from the provisioning of the VM. We pre-allocate a small pool of VM instances of size p ahead of time. New VM instances for operators are requested from this VM pool, which can happen in seconds. Asynchronously the pool is refilled to size p by requesting VM instances from the cloud provider.

A challenge is to decide on the optimal VM pool size. A VM pool that is too small may get exhausted when multiple VMs are requested in short succession. A large VM pool incurs an unnecessarily high financial cost because preallocated VMs are billed by the cloud provider.

We make two observations regarding the VM pool size p: (i) with real-world failures, preallocating 1–2 VMs is sufficient, which means that p is primarily determined by the scale out requirement; (ii) p can be adjusted to the scale out behaviour over time. For example, p may be kept larger while the SPS scales out aggressively to adapt to an open loop workload. After the rate of new VM requests decreases, the VM pool can shrink to support steady-state operation. For simplicity, we use a constant VM pool size in §6.1.

6. EVALUATION

Next we evaluate our integrated approach for dynamic scale out and fault tolerance in an SPS. The goals of our experimental evaluation are to investigate:

(i) the **effectiveness** of our **stateful operator scale out** approach for a *closed loop* workload, i.e. when the SPS has to sustain the input rate without tuple loss, using the Linear Road Benchmark (§6.1); and, for an *open loop* workload,

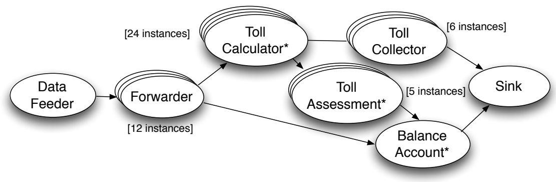

Figure 5: Query for the Linear Road Benchmark

i.e. when tuples can be discarded during overload, using a streaming map/reduce-style query over Wikipedia data. We show that our approach scales effectively with the input rate while maintaining low processing latency;

(ii) the **recovery time** of the **stateful recovery** mechanism for a windowed word frequency query (§6.2). We show that our approach recovers operator state faster than *source replay* [29] and *upstream backup* [8] of tuples. We also demonstrate how recovery times can be reduced through parallel recovery; and

(iii) the **impact** of our **state management** approach on tuple processing latency (§6.3). We study how input rate, state size and the checkpointing interval affect the overhead.

The experiments are conducted using an experimental stream processing system implemented in Java. We deploy it on Amazon EC2 using *small* VM instances for the query operators, which have 1.7 Gb of RAM, moderate IO performance and 1 EC2 compute unit (equivalent to a 1.0–1.2 GHz 2007 Xeon CPU). While these VMs have low processing capabilities, they are representative of public cloud VMs. We use *high-memory double extra large* VM instances (34 Gb of RAM; high IO performance; and 4 virtual cores with 3.25 compute units) to emit the source data streams and to gather output results.

6.1 Dynamic Scale Out

Closed loop workload. We first evaluate the effectiveness of our scale out approach when adapting to an increasing closed loop workload, i.e. when the SPS has to scale out to match an increasing input stream rate without tuple loss.

Linear Road Benchmark (LRB). Similar to previous work [32, 18], we use the LRB [5]. It models a road toll network, in which tolls depend on the time of day and level of congestion. It specifies the following queries: (i) provide toll notifications to vehicles within 5 s; (ii) detect accidents within 5 s; and (iii) answer balance account queries about paid toll amounts. The goal is to support the highest number of express-ways L while satisfying the above latency constraint. Over the course the benchmark, the input rate for a single express-way (L=1) begins at 15 tuples/s and increases to 1700 tuples/s. An SPS without dynamic scale out support would have to be provisioned to sustain the peak rate. To generate a sufficiently high input stream rate, we precompute the input stream for L=1 in memory and replicate it for multiple express-ways [10].

Our LRB query implementation consists of 7 operators, as shown in Fig. 5. A *data feeder* acts as the source and generates the input data stream. A *forwarder* operator routes tuples downstream according to their type. The stateful *toll calculator* maintains tolls and detects accidents. The stateful *toll assessment* operator computes account balances and responds to balance queries in tuples. The stateless *collector* operator gathers notifications. The stateful *balance account* operator receives the balance account notifications and aggregates the results. The *sink* operator collects all results.

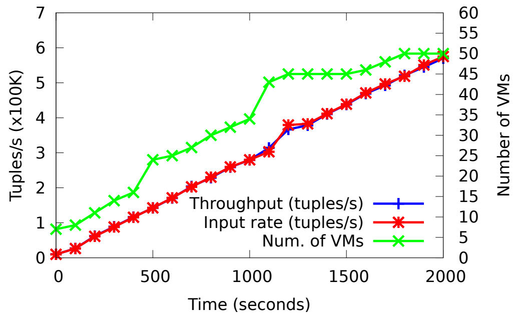

Figure 6: Dynamic scale out for the LRB workload with L=350 (closed loop workload)

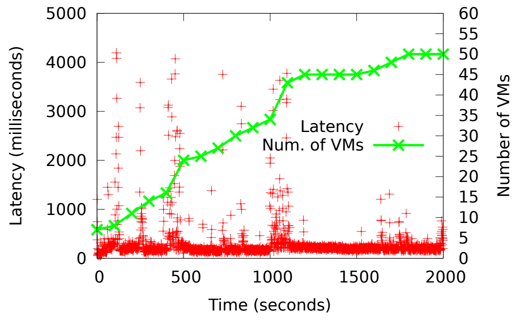

Figure 7: Processing latency for LRB workload

We deploy the LRB query on Amazon EC2 and set the scale out threshold to 70% utilisation, as discovered below. 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 Fig. 5. The main computational bottleneck in the query, the *toll calculator*, is partitioned the most by the system, followed by the *forwarder*.

This result is about 70% of L=512, which is currently the highest reported L-rating in the literature by Zeitler and Risch [32]. Their result was obtained on a private cluster with 560 dedicated cores with 2.27 Ghz—substantially more resources than what we used. Since our approach only scales out bottleneck operators at a fine granularity, it can be more resource efficient than the replication of the whole query graph used by Zeitler and Risch.

Fig. 6 shows the number of allocated VMs along with the input rate and achieved result throughput over time. For L=350, the input rate is initially approx. 12,000 tuples/s and increases to 600,000 tuples/s. 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.

In Fig. 7, we show the processing latencies of output tuples, as a representative metric for the performance experienced by the query. The 99th and 95th 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. Dynamic scale out, however, affects tuple latency—there are latency peaks of up to 4 s after scale out events due to stream buffering and replay.

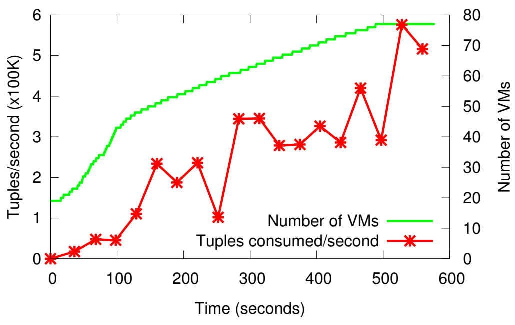

Figure 8: Dynamic scale out for a map/reduce-style workload (open loop workload)

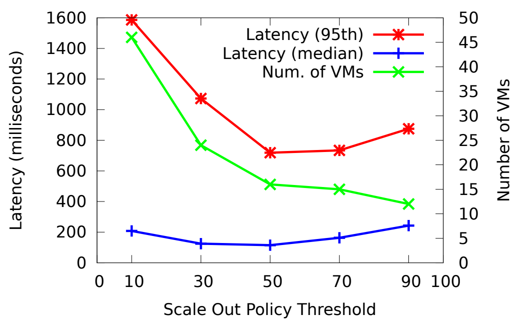

Figure 9: Impact of the scale out threshold δ on processing latency

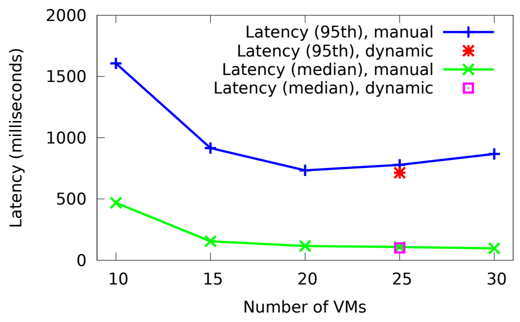

Figure 10: Comparison between dynamic and manual scale out

Towards the end of the experiment, the median latency increases when the system becomes overloaded.

Open loop workload. In addition, we explore an open loop workload, in which the SPS is initially under-provisioned.

Map/reduce-style top-k query. We implement a *map/reduce-style top-k query* that outputs every 30 seconds the ranking of the most visited Wikipedia language versions based on Wikipedia data traces. Initially, we set the input stream rate to be well over the performance capacity of the SPS, incurring tuple loss. The goal is to let the SPS scale out the query in order to sustain the incoming rate.

We use 18 data sources to inject tuples to a stateless *map* operator, which removes unnecessary fields from tuples, and a stateful *reduce* operator, which maintains a top-K dictionary of the frequency of visited Wikipedia language versions. When the reducer scales out, we use the sink to aggregate the partial results and output the final answer.

We present the dynamic scale out behaviour in Fig. 8. As expected, the SPS scales out until it can sustain the incoming rate of 550,000 tuples/s. The scale out process leads to peaks in the tuple throughput. After scaling out an operator, the input buffers of the new partitioned operators consume tuple faster than they can be processed. Only after the input queues have filled, performance stabilises again.

Another observation is that the rate of scale out is higher in the first part of the experiment (until t=100). The reason is that initially more map operators are scaled out than reduce operators: the stateless map operators scale out faster than the stateful reduce operators.

Impact of scale out threshold. Next we evaluate the impact of the scale out policy. We study how different scale out thresholds δ affect the number of allocated VMs and the tuple processing latency (see §5.1). The goal is to find the best trade-off between resource allocation efficiency and processing performance. To explore the efficiency of VM allocation, we compare our dynamic scale out approach to manual scale out by a human expert.

We investigate different thresholds δ using the LRB with L=64. We initially deploy the query with one VM per operator and observe the number of VMs at the end of the experiment and the processing latency. Fig. 9 shows that, as δ increases from 10% to 90%, fewer VMs are allocated. The median latency curve is concave, increasing not only for high thresholds, when VMs are close to overload, but also for low ones. This behaviour can be understood better by considering the 95th percentile of tuple latencies. When δ is small, the system performs many scale out operations, which impacts processing latency, especially at higher percentiles.

Based on these results, a threshold δ of 50%–70% provides the best trade-off. It follows the best practice in data centre management to maintain a headroom of unused resources in anticipation of workload bursts and transient peaks [13].

Efficiency of resource allocation. We evaluate the efficiency of the dynamic scale out policy against a human expert who manually parallelises operators. In this experiment, we use the LRB query with L=115. The human expert is given a fixed number of VMs and uses them as effectively as possible to support this workload. The human expert, based on their understanding of the relative costs of operators, tracks the bottleneck across multiple scaled out versions of the LRB query. The dynamic scale out policy allocates 25 VMs at the end of the experiment.

Fig. 10 shows the processing latency as a function of the number of VMs for the manual scale out decisions. In addition, the median (101 ms) and 95th percentile (714 ms) of latencies for the dynamic scale out policy are indicated in the figure. The results show that the most efficient manual allocation for this workload is 20 VMs—with fewer VMs, the 95th latency percentile starts to increase due to the high VM utilisation. In comparison, automatic scale out achieves low latency with only 25% more resources than the optimum.

6.2 Failure Recovery

To evaluate failure recovery, we first compare recovery time against other fault tolerance approaches that can handle stateful operators. We also measure the impact of the checkpointing interval on recovery time. We finish with an exploration of the benefit of parallel recovery.

We compare our approach for *recovery using state management* (R+SM) with *upstream backup* (UB) and *source re-*

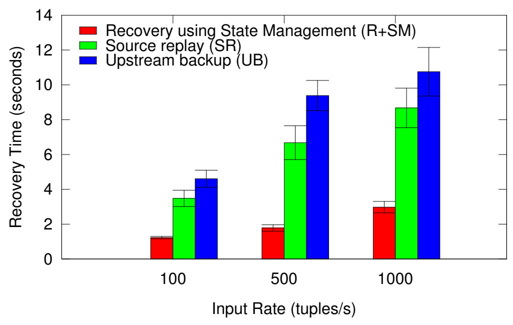

Figure 11: Recovery time for different fault tolerance mechanisms

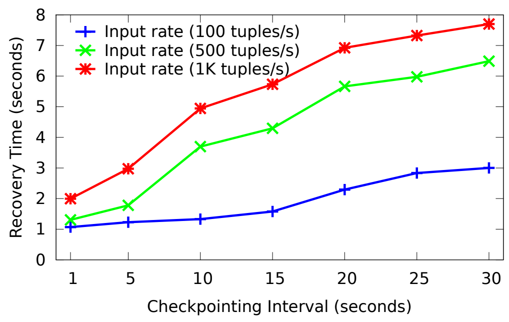

Figure 12: Recovery time for different R+SM checkpointing intervals

play (SR). UB buffers tuples in each operator 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 [29].

SR and UB are only suitable for stateful operators whose state can be restored after re-processing few tuples. As a result, we cannot use the LRB query for comparison because its operators require the whole history of tuples. Instead, we use a simple windowed word count query.

Windowed word frequency query. This query counts word frequencies over a 30 s window. It executes over a stream of sentence fragments, each 140 bytes in size. It has two operators: a *word splitter* tokenises the input stream into words; and a *word counter* maintains frequency counters for each word. The state of the word counter is a dictionary of words and their counters.

Recovery time. 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 hosting the *word counter* operator and measure the time to recover (i.e. until the complete operator state was restored).

Fig. 11 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 the worst case, it must replay 5 s worth of tuples instead of the entire window of 30 s. Especially at higher input stream rates, the overhead of re-processing tuples dominates recovery time.

Impact of checkpointing interval. In Fig. 12, we show the change in recovery time as a function of the checkpointing interval for different input rates. Recovery time increases

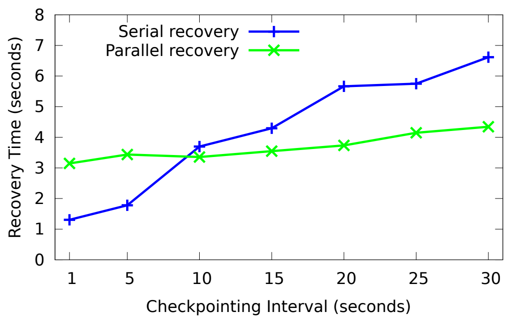

Figure 13: Recovery time for serial and parallel recovery using state management

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.

Parallel recovery. To prevent the SPS from falling behind during recovery, *parallel recovery* combines scale out with recovery (see §4). In this experiment, we compare serial to parallel R+SM with two partitioned operators. Fig. 13 shows recovery times for different checkpointing intervals with an input rate of 500 tuples/s. For short intervals, parallel recovery does not bring a benefit due to its higher overhead with two partitioned operators. As the interval increases, however, more tuples have to be replayed when restoring operator state, and parallel recovery can process at a higher rate with two partitioned operators.

6.3 State Management Overhead

Our state management approach based on periodic checkpointing incurs an overhead. Since the overhead on processing throughput is negligible and could not be observed, we measure its effect on tuple processing latency for different input rates, state sizes and checkpointing intervals.

Processing latency. We explore the processing latency of the windowed word frequency query with different operator state sizes and input rates. We synthetically vary the dictionary size between *small* (10^2 entries; ≈2 Kb), *medium* (10^4 entries; ≈200 Kb) and *large* (10^5 entries; ≈2 Mb). We use a checkpointing interval of 5 s and a window of 30 s. We compare to a baseline without checkpointing, which gives raw processing latency independent of state size.

Fig. 14 shows that the 95th 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. Since the system becomes overloaded at this rate, it does not have sufficient resources for checkpointing, which could be addressed by scaling out the operator. Note that measuring overhead as a high latency percentile results in a worst case—the effect on the medium latency is less pronounced and on the throughput is not observable.

Impact of checkpointing interval. We investigate the impact of different checkpointing intervals by measuring the

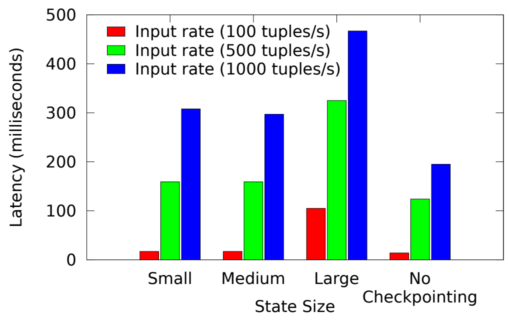

Figure 14: Overhead of state checkpointing for different input rates and state sizes

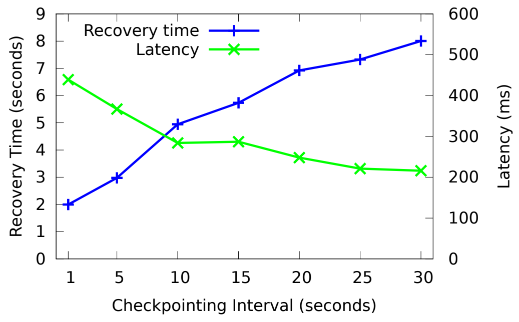

Figure 15: Trade-off between processing latency and recovery time for different checkpointing intervals

processing latency for the windowed word frequency query with 1000 tuples/s. Fig. 15 shows how the 95th percentile of latency decreases with an increased checkpointing interval. For comparison, the plot includes the expected recovery time for that interval. There is a trade-off: the larger the interval, the lower the impact on tuple processing, at the expense of a longer recovery time after failure. The checkpointing interval should therefore be chosen based on the anticipated failure rate and the performance requirements of a query.

7. RELATED WORK

State management in stream processing. There are proposals for managing state in SPS with particular goals. Gulisano et al. [15] describe how to partition the state of specific operators such as join and aggregate for scale out. *State spilling* was proposed to handle overload conditions by temporarily storing parts of operator state on disk or a remote node [19]. In addition, operator state is made explicit when replicating operators across nodes. Feng et al. [11] use counting bloom filters to improve the performance of state transfers for replicated operators. In contrast, we expose state through a set of generic set of management primitives to integrate scale out and failure recovery.

Parallelism in stream processing. Much work has focused on exploiting parallelism in stream processing and, more recently, performing scale out. Apache S4 [23] and Twitter Storm [29] express queries as directed acyclic graphs with parallel operators. S4 schedules parallel instances of operators but does not manage their parallelism or state. Storm allows users to specify a parallelisation level and supports stream partitioning based on key intervals but it ignores operator state and cannot scale out at runtime.

IBM System S [4] supports intra-query parallelism through a fine-grained subscription model, which specifies stream connections, but management is manual. Hirzel [16] provides a MatchRegex operator for System S, which detects tuple patterns in parallel. This approach does not consider dynamic repartitioning, and the state is specific to automata-based pattern detection. Schneider et al. [27] add elastic operators to the SPADE language, which gradually find the optimal number of threads for stateless processing with maximum throughput. Our work is orthogonal in that it focuses on parallelising stateful operators.

StreamCloud [15] parallelises stateful queries at runtime but does not support failure recovery. It uses a query compiler to synthesise high-level queries into a graph of relational algebra operators. It uses hash-based parallelisation, which is geared towards the semantics of joins and aggregates. Instead, we develop lower-level state management primitives, applicable to a wider class of stateful operators. As shown in §6.1, our approach can support custom optimised operators for the LRB query, which can achieve higher per-node performance than relational operators.

Some proposals for scalable SPSs [21] are inspired by the map/reduce paradigm, adapting it for low-latency stream processing, but are limited by its expressiveness. Martin et al. [21] add stateful reducers to map/reduce and rely on external fault tolerance mechanisms. In contrast, our state management approach is integrated as part of the SPS.

Zeitler and Risch [32] propose the *parasplit* operator for partitioning streams statically according to a cost model. Instead, we make decisions about the parallelisation level at runtime in response to performance metrics. Backman et al. [6] partition operators across nodes in an SPS to minimise processing latency by balancing load according to simulated latency estimates. In contrast, we partition operators on-demand to remove processing bottlenecks and show experimentally that this maintains low latency.

Fault tolerance in stream processing. *Active replication* in SPSs suffers from a high resource overhead, doubling the number of required processing nodes, which is not cost-effective in large cloud-based deployments.

Martin et al. [20] propose to reduce the resource footprint of active replication in cloud-hosted SPSs by using spare resources due to over-provisioning. However, this requires an average utilisation of less than 50%—active replication is suspended when peak workloads demand all resources. While we assume long-lived failures, Zhang et al. [33] focus on transient failures. They combine active replication in the face of failures for fast recovery with passive replication during normal operation.

Passive replication is more resource efficient but incurs a periodic overhead due to state synchronisation, and is prone to higher recovery times, especially with large state sizes.

IBM System S [30] provides a reliability mechanism, in which operator state is persisted using an object/relational mapping to a DBMS. It ensures the consistency of checkpoints with asynchronous operations pending at the time of failure by saving them along with the state, similarly to our buffer state. Our approach, however, does not depend on an external DBMS, which may become a bottleneck.

To reduce recovery times in passive replication, Sebepou et al. [28] partition the state into smaller chunks, updating operator state incrementally. Their approach is only evaluated for aggregate operators, and it remains unclear how it can be applied to other types of stateful partitioned operators.

D-Stream [31] is a streaming version of the Spark parallel processing framework. It processes datasets across different nodes and periodically checkpoints results. Similar to our parallel recovery, missing results are recovered in parallel from multiple nodes. However, the use of Spark means that processing latencies are high.

Upstream backup [8] requires nodes to buffer tuples until they have been processed by downstream operators. On failure, lost tuples are replayed by upstream nodes. In general, this suffers from long recovery times when a large number of tuples have to be re-processed to restore stateful operators and cannot support state that depends on the complete stream history. Sweeping checkpointing [14] reduces recovery times by checkpointing state when all downstream operators have completed processing and buffers are smallest. This is orthogonal to our approach and could be used to choose the best checkpointing time.

8. 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 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.

9. REFERENCES

[1] D. 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] Y. Ahmad, O. Kennedy, et al. DBToaster: Higher-Order Delta Processing for Dynamic, Frequently Fresh Views. In *VLDB*, 2012.

[4] L. Amini, H. Andrade, et al. SPC: A Distributed, Scalable Platform for Data Mining. In *DMSSP*, 2006.

[5] A. Arasu, M. Cherniack, et al. Linear Road: A Stream Data Management Benchmark. In *VLDB*, 2004.

[6] N. Backman, R. Fonseca, et al. Managing Parallelism for Stream Processing in the Cloud. In *HotCDP*, 2012.

[7] M. Balazinska, J. Hwang, et al. Fault Tolerance and High Availability in Data Stream Management Systems. In *Encyclopedia of Database Systems*, 2009.

[8] M. Balazinska, A. Rasin, et al. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, 2005.

[9] S. Blanas, J. M. Patel, et al. A comparison of join algorithms for log processing in mapreduce. *SIGMOD*, 2010.

[10] M. Duller, J. S. Rellermeyer, et al. Virtualizing Stream Processing. In *Middleware*, 2011.

[11] Y.-H. Feng, N.-F. Huang, et al. Efficient and Adaptive Stateful Replication for Stream Processing Engines in High-Availability Cluster. *TPDS*, 22(11), 2011.

[12] P. Gill, N. Jain, et al. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *SIGCOMM*, 2011.

[13] A. Greenberg, J. Hamilton, et al. The Cost of a Cloud: Research Problems in Data Center Networks. In *SIGCOMM*, 2008.

[14] Y. Gu, Z. Zhang, et al. An Empirical Study of High Availability in Stream Processing Systems. In *Middleware*, 2009.

[15] V. Gulisano, R. Jimenez-Peris, et al. StreamCloud: An Elastic and Scalable Data Streaming System. *TPDS*, 99(PP), 2012.

[16] M. Hirzel. Partition and Compose: Parallel Complex Event Processing. In *DEBS*, 2012.

[17] J. H. Hwang, Y. Xing, et al. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *ICDE*, 2007.

[18] N. Jain, L. Amini, et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD*, 2006.

[19] B. Liu, Y. Zhu, et al. Run-time Operator State Spilling for Memory Intensive Long-running Queries. In *SIGMOD*, 2006.

[20] A. Martin, C. Fetzer, et al. Active Replication at (Almost) No Cost. In *SRDS*, 2011.

[21] A. Martin, T. Knauth, et al. Scalable and Low-Latency Data Processing with Stream MapReduce. In *CLOUDCOM*, 2011.

[22] R. Motwani, J. Widom, et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR*, 2003.

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

[24] N. Parikh and N. Sundaresan. Scalable and Near Real-Time Burst Detection from eCommerce Queries. In *SIGKDD*, 2008.

[25] M. Russell. *Mining the Social Web*. O'Reilly, 2011.

[26] B. Satzger, W. Hummer, et al. Esc: Towards an Elastic Stream Computing Platform for the Cloud. In *IEEE CLOUD*, 2011.

[27] S. Schneider, H. Andrade, et al. Elastic Scaling of Data Parallel Operators in Stream Processing. In *IPDPS*, 2009.

[28] Z. Sebepou and K. Magoutis. CEC: Continuous Eventual Checkpointing for Data Stream Processing Operators. In *DNS*, 2011.

[29] Twitter Storm. `github.com/nathanmarz/storm/wiki`.

[30] R. Wagle, H. Andrade, et al. Distributed Middleware Reliability and Fault Tolerance Support in System S. In *DEBS*, 2011.

[31] M. Zaharia, T. Das, et al. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. In *HotCloud*, 2012.

[32] E. Zeitler and T. Risch. Massive Scale-out of Expensive Continuous Queries. *VLDB Endowment*, 4(11), 2011.

[33] Z. Zhang, Y. Gu, et al. A Hybrid Approach to HA in Stream Processing Systems. In *ICDCS*, 2010.