Towards Elastic Stream Processing: Patterns and Infrastructure

Kai-Uwe Sattler
Ilmenau University of Technology
Ilmenau, Germany
kus@tu-ilmenau.de

Felix Beier*
Ilmenau University of Technology
Ilmenau, Germany
felix.beier@tu-ilmenau.de

ABSTRACT

Distributed, highly-parallel processing frameworks as Hadoop are deemed to be state-of-the-art for handling big data today. But they burden application developers with the task to manually implement program logic using low-level batch processing APIs. Thus, a movement can be observed that high-level languages are developed which allow to declaratively model dataflows that are automatically optimized and mapped to the batch-processing backends. However, most of these systems are based on programming models as MapReduce that provide elasticity and fault-tolerance in a natural manner since intermediate results are materialized and, therefore, processes can simply be restarted and scaled with partitioning input datasets. For continuous query processing on data streams, these concepts cannot be applied directly since it must be guaranteed that no data is lost when nodes fail. Usually, these long running queries contain operators that maintain state information which depends on the data that has already been processed and hence they cannot be restarted without information loss. This also is an issue when streaming tasks should be scaled. Therefore, integrating elasticity and fault-tolerance in this context is a challenging task which is subject of this paper. We show how common patterns from parallel and distributed algorithms can be applied to tackle these problems and how they are mapped to the Mesos cluster management system.

1. INTRODUCTION

Processing and analyzing big data is one of todays big challenges. A popular definition from a Gartner report names the *three 'V's – volume*, *velocity*, and *variety* as the main characteristics of big data. Among them, velocity refers to the analytics of dynamic data even in (near) realtime.

Several approaches and techniques have been developed in the past to process dynamic data. *Data stream management systems (DSMS)* like STREAM, Aurora, IBM Infosphere Streams or our own AnduIN engine provide abstractions to process continuous and possibly infinite streams of data instead of disk-resident datasets. Typically, this includes standard (relational) query operators, window-based operators for computing joins and aggregations as well as more advanced data analytics and data mining operators working on portions of the stream, e.g. windows or synopses of data. *Complex Event Processing systems (CEP)* particularly support the identification of event patterns in (temporal) streams of data such as a sequence of specific event types within a given time interval. Typically, systems of both classes provide a declarative interface, either in form of SQL-like query languages like CQL for DSMS, event languages like SASE, or in the form of dataflow specifications like SPL in IBM Infosphere Streams.

Recently, several new distributed stream computing platforms have been developed, aiming at providing scalable and fault-tolerant operation in cluster environments. Examples are Apache S4 or Storm. In contrast to DSMS or CEP engines theses platforms do not (yet) provide declarative interfaces and, therefore, require to program applications instead of writing queries. Developers of these systems argue that they provide the same for stream processing what Hadoop did for batch processing – which raises the hope of a similar movement towards higher-level languages as we can see with Pig, Jaql etc. for MapReduce.

However, there are some challenges in scalable and elastic stream processing which are different from batch processing with Hadoop. Whereas in Hadoop, input data as well as intermediate results are materialized on disk and, therefore,

- both, map and reduce tasks can be restarted arbitrarily in case of failures until the entire job is finished,
- since computation state is saved, the number of nodes assigned to map and reduce tasks can be simply adjusted by partitioning input and intermediate results.

This is more difficult when processing dynamic data – even with platforms as S4 or Storm which to some extent support a reliable and scalable operation. The main differences are:

(1) Partitioning of streams for data-parallel processing is not always easily possible, for example in case of window- or sequence-based operators including CEP operators. Also, elastic operation by adding new nodes at runtime of a query requires at least rerouting of data.

(2) Stream queries are typically long running queries which cannot be simply restarted without losing data. Furthermore, because of this, the deployment and resource allocation (placement of queries on nodes, allocating memory and CPUs) are much more critical.

*This work is partially funded by an IBM PhD Fellowship.

In this paper, we try to answer the question how to bridge the gap between an easy-to-use, high-level declarative interface for data stream analytics and scalable cluster-based stream computing platforms in order to address these challenges. The contribution of this paper is twofold:

- Based on a basic dataflow model for stream queries we describe patterns for fault-tolerant and scalable query processing and discuss constraints of their application.
- We show the implementation and deployment of these patterns using our distributed stream and CEP engine AnduIN and the Mesos cluster infrastructure by describing techniques supporting flexible and elastic deployment.

Though, we use the AnduIN system for describing and implementing the concepts, we think the ideas and patterns are applicable to other platforms, too.

2. RELATED WORK

The relevant work related to this paper can be classified into the main categories: continuous query processing and scalable dataflow platforms.

Continuous query processing is usually implemented in data stream management systems (DSMS). Pioneered by systems like STREAM, Borealis, and Telegraph, several approaches and systems have been developed in the last decade including commercial products such as IBM InfoSphere Streams and StreamBase. Typically, these systems provide a SQL-like query language enhanced by features for dealing with continuous queries such as sliding windows.

Partitioning, distributed processing, and fault tolerance have been studied to some extent, e.g., in Borealis [1] by introducing replicated processing nodes as well as several new tuple types such as punctuation tuples and control tuples like undo tentative (tuples resulting from processing a subset of the input which can be corrected later) and done tuples indicating that state reconciliation finished. State reconciliation is the process of stabilizing the output result, e.g., by replacing previously tentative results. In this way, this approach aims at fault-tolerance but not at partitioning.

Another approach in the form of a programming model has been proposed in [14] as so-called discretized streams (D-Streams). This idea is based on resilient distributed datasets which are storage abstractions used for rebuilding lost data.

An approach addressing load balancing issues by partitioning while providing fault tolerance for pipelined dataflows is Telegraph's FluX [10]. FluX is a dataflow operator extending the idea of the exchange operator form parallel query processing. The operator encapsulates state partitioning and tuple routing and allows to repartition even stateful operators while executing the dataflow pipeline.

Scalable dataflow platforms try to extend the applicability of the MapReduce paradigm for large-scale parallel batch processing to pipeline processing and continuous query support. One example is HOP (Hadoop Online Prototype) [4]. In HOP, map tasks maintain TCP sockets to reducers for pipelining their output. In addition, pipelining is also supported between jobs by sending the output of reducers directly to mappers of a subsequent job. Further, distributed dataflow systems are Twitter's Storm and Apache S4. Storm implements fault detection at task level and guaranteed message passing, whereas in S4 messages can be lost. Storm runs so-called topologies – subsets of these topologies are assigned to worker processes of a cluster. However, these systems only offer a simple programming model and, therefore, operators and topologies have to be implemented in a programming language like Java or Python. Furthermore, state recovery and partitioning have to be implemented manually, too.

Optimus [11] is a framework for dynamic rewriting of execution plans for data-parallel computing, e.g., formulated in DryadLINQ. The framework supports rewriting of MapReduce programs at runtime, addressing issues like repartitioning, fault tolerance, and handling data skew. But the required algorithms have to be implemented by the user.

3. PATTERNS FOR SCALABLE PROCESSING OF DATA STREAM QUERIES

In the following we assume a simple processing model for data stream queries: a query is represented by a dataflow graph which is a common model in literature [9, 11].

In such a directed acyclic graph, nodes represent query operators and edges describe the tuple flow between them. Query nodes can be arbitrary pipeline operators of a stream query algebra [2] like filter, projection etc. as well as window- or synopsis-based operators as sliding window joins and aggregations, but also more complex data analytics operators including CEP and data mining. Communication between query operators is performed either directly by invoking operator functions or via buffers/queues. Obviously, this represents a very generic execution model to which a wide range of declarative query languages like CQL, dataflow specifications such as IBM's SPL, or implementation-oriented approaches as used in S4 or Storm can be mapped. This model can be easily extended to the distributed case by inserting network reader/writer nodes which use appropriate communication protocols and APIs, e.g., TCP/UDP sockets or more advanced solutions like ZeroMQ.

There are two main reasons for distributing query nodes: increasing processing reliability by introducing redundancy, and increasing performance and/or scalability by load distribution. The following patterns support these goals to different degrees. In this discussion, we use the term "query task" as the unit of distribution/scheduling for both, elementary algebra operators, and for dataflow (sub-)graphs with well-defined properties (input/output, stateless vs. statefulness).

Pattern 1: Simple standby: For a critical query node N a standby node S is maintained on a separate compute unit which is activated if N fails. This requires monitoring of N, e.g. by a combination of heartbeats and cluster coordination service such as ZooKeeper as well as rerouting the input tuple stream to S. Since in case of a failure the state of N is lost, this pattern is applicable only for stateless nodes.

Pattern 2: Checkpointing: This pattern is similar to pattern 1 but supports stateful operators. Failover is achieved by periodically checkpointing the state of the critical node N to a shared disk and restarting the standby node S from the checkpoint. Examples of such checkpoints are the content of sliding windows or hash tables for joins and aggregations.

Pattern 3: Hot standby: If the failover time of pattern 2 is not acceptable, a hot standby approach can be chosen where redundant query nodes S are kept actively. To achieve this, the input stream has to be sent to all redundant nodes, either by using multicast strategies at the network or at the application level. This pattern works both with stateless and stateful query operators but requires a special node to eliminate duplicate results, e.g., a stream selector node which forwards the input from only one of multiple streams.

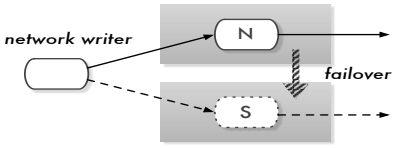

Figure 1: Simple Standby

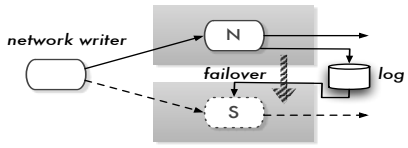

Figure 2: Checkpointing

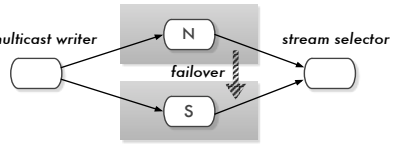

Figure 3: Hot Standby

Pattern 4: Stream partitioning: This pattern exploits data parallelism by partitioning the input stream. It can be implemented by a splitter node redirecting each input tuple to one of the query nodes $N_1 \dots N_k$ or by a multicast writer with an additional partitioning node $P_1 \dots P_k$ for filtering the input stream according to the partitioning scheme. Finally, the results are merged into a single stream.

Pattern 5: Stream pipelining: In contrast to pattern 4 this pattern exploits task parallelism by splitting a complex query node N into a sequence of query nodes $N_1 \dots N_k$ and placing them on separate compute units.

Usually, multiple patterns will be applied in order to achieve certain quality of service (QoS) guarantees as fault tolerance (patterns 1-3) or elasticity for adapting resource consumption (patterns 4-5) according to the needs of the applications. Of course this pattern list does not claim completeness. There are several others that are applicable under certain circumstances, e.g., parallelization through aggregation trees for commutative and associative aggregation operators [9]. Nevertheless, these basic patterns described here are well-known in distributed and parallel algorithms, and – with slight modifications – cover various use cases [1].

In the following we will describe how these patterns can be utilized in a dataflow framework to dynamically restructure the physical representation of the graph in a continuous query context that is executed in a cluster infrastructure. The restructuring is achieved with a simple set of rewriting rules that are automatically applied on the graph without the need to manually code them as in existing approaches [11] while guaranteeing that no state information is lost during the restructuring phase. We present the algorithms based on our AnduINv2 stream processing engine but highlight that they are also applicable on other frameworks as Dryad with slight modifications to achieve streaming semantics.

4. QUERY DEPLOYMENT INFRASTRUCTURE

Fig. 6 illustrates the dataflow model used in AnduINv2 and how it is mapped to the physical layer of executable code. While the first prototype of the system aimed at processing sensor data as well as in-network processing [12] and complex event processing (CEP) [8], our current research focuses on processing techniques for cluster environments. The AnduINv2 system comprises three components: (1) the runtime environment containing the implementation of query operators including a CEP engine as well as operators for controlling the query execution, (2) a query compiler translating a dataflow-based query specification given in an XML file into query tasks, i.e., executable code linked to the runtime environment, (3) a query scheduler and executor integrated with the Mesos cluster framework that deploys query tasks for processing them on physical nodes.

[1]Actually, aggregation trees are just combinations of partitioning and pipelining patterns.

AnduINv2 queries are deployed as separate processes in Mesos which are just-in-time compiled using the system's C++ compiler. This provides an easy mechanism to plug-in user defined operators and exchange operator implementations. During deployment, processes are interlinked by query channels which simply represent an abstraction over network connections (TCP/UDP sockets, ZeroMQ connections).

Query tasks can be shared among multiple queries when they share some common (sub)streams or operators as described in [3]. Further, a query can be implemented as a set of tasks which are distributed across multiple nodes in the cluster. Therefore, the logical query tree is partitioned into smaller subtrees that are translated separately. We will use these mechanisms for implementing the elasticity patterns.

4.1 Dataflow Graph Rewriting

dataflows are specified in an XML format which can be seen as an intermediate representation, allowing to use different frontends such as CQL or graphical tools. A dataflow specification consists of stream type definitions and operator definitions with name, type, type-specific parameters as well as input and output channels. These channels are typed and are used to interconnect operators to form a graph. The following example shows a simple dataflow specification. (We omitted the XML notation for better readability).

```
type name = "aStreamType" {
    column name = "x", type = "int" ...
}
operator name = "source", type = "reader" {
    output name = "aStream", type ="aStreamType"
}
operator name = "myFilter", type = "filter" {
    input name = "aStream"
    param condition = "aStream.x < 42"
    output name = "filteredStream", type = "aStreamType"
}
operator name = "sink", type = "writer" {
    input name = "filteredStream"
}
```

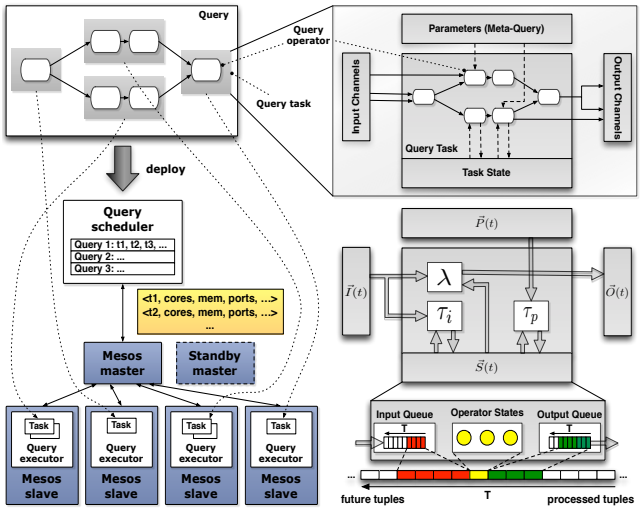

Figure 6: query model

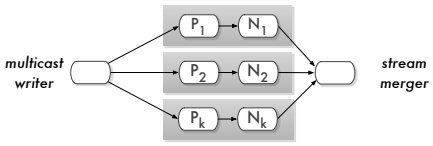

Figure 4: Partitioning

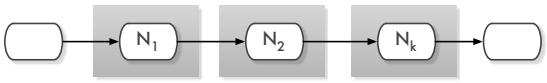

Figure 5: Pipelining

Note, that apart from stream input and possible output no communication operators have to be specified as part of the query. Such operators are added during rewriting if necessary. For formulating rewriting rules we use a simple notation. A dataflow as given above is written as

$$\mathsf{sink} := \mathsf{writer}(\mathsf{f} := \mathsf{filter}(\mathsf{src} := \mathsf{reader}))$$

where writer, filter, and reader are operator types and the optional sink, f, and src names denote operator instances.

During rewriting, graph patterns have to be matched and constraints are checked. For this purpose, the pseudo-type any is used as a placeholder for any possible type, and any* represents a dataflow of arbitrary operator types. The following pattern matches a dataflow subgraph containing a stateless filter operator (which is the case for any filter):

$$\mathsf{a}_2 := \mathsf{any}(\mathsf{f} := \mathsf{filter}(\mathsf{a}_1 := \mathsf{any}^*))[\mathtt{stateless}(\mathsf{f})]$$

To apply the patterns, a rewriting rule can be specified:

$$\Rightarrow \underbrace{\mathsf{a}_2(\mathsf{stream\text{-}selector}(\mathsf{failover}}_{@p_1}(\underbrace{\mathsf{writer}(\mathsf{f}(\mathsf{reader}}_{@p_*}(\underbrace{\mathsf{multicast}(\mathsf{a}_1))}_{@p_2})))))$$

Besides inserting or replacing operators (such as stream-selector, failover, and multicast operators in the previous example), operator nodes are also annotated with placement information where p_i, p_j with $i \neq j$ denote distinct compute nodes and p_* denotes an arbitrary number of nodes.

4.2 Failover Handling

With these rewriting rules, internal data management nodes and different operator implementations can be transparently injected into the query plan without impacting the application. This allows to re-schedule a query task to another node in case of a failure (pattern 1), use an operator implementation that automatically integrates snapshotting (pattern 2), or replicate a task to implement hot standby.

The actual flow of data through query channels during runtime is controlled by special operator parameters – e.g., target IP addresses and ports – that can be adjusted through a concept we call *meta queries* (cf. Sect. 4.5). To detect and react on failures, query tasks are instrumented with monitoring interfaces that inform the query scheduler about the nodes' health and performance measures as tuple processing rates. The scheduler then triggers a graph rewriting.

When a rewritten graph needs to be deployed, it has to be guaranteed that no information of the tuple stream is lost. To analyze the necessary steps, we reduce a query task to a finite state machine model (cf. Fig. 6) which is common for implementing CEP operators [6] but can also be applied for general dataflow transformations. The query task receives a stream of input tuples $\vec{I}(t)$, applies its logical operation(s) λ – e.g., a filter or a join – to generate an output stream $\vec{O}(t)$. (We use vectorial representations here to combine all channels into a single quantity.) The output might depend on the task's state $\vec{S}(t)$, i.e., the state of all internal operators which can basically include anything that is required for implementing the operators (e.g., hash tables, or sliding windows) and are updated with each incoming tuple through a state transition function τ_i [2]. The meta query extension is represented by special input channels $\vec{P}(t)$ that modify the operator state through τ_p.

To guarantee that a node failure does not lead to an information loss it is necessary that all results which have not been consumed by the following target can be reproduced from the possibly infinite input stream. Therefore, the operator state needs to be snapshotted after each input tuple, or – if this is too expensive – the input tuples need to be persisted in order to reproduce this state with just 'replaying' the input. Which tuples are still required for a possible replay can be controlled by special tuple messages that are exchanged between tuple producers and consumers as in the Borealis system [1]. Note when frameworks as ZeroMQ are used to implement query channels, reliable message delivery can be guaranteed without the need to modify operators.

In order to implement fault tolerance with transferring stateful query tasks to other computing nodes, a simple protocol as presented in [10] is sufficient:
(1) quiesce all input streams,
(2) replicate the task state to the target node,
(3) redirect the input streams to the target node,
(4) unquiesce all input streams.

4.3 Elasticity Handling

The same algorithm can be used to replace query tasks with their rewritten versions that compute the same logical transformation but use different operator implementations and/or partitioning schemes of the query graph into tasks for implementing the elasticity patterns 4 and 5.

Rewriting Cost Model: Usually, there are several possibilities for rewriting dataflow graphs. In order to make right decisions which tasks shall be replaced and how many nodes should be allocated, a cost model is required taking possible rewriting benefits into account as well as costs for the restructuring, e.g., for transferring states or costs for additional resources from the cluster infrastructure. Discussing elaborate decision models is not in the focus of this paper and is left for future work. We outline a rate-based model that is suitable to find hot spots in dataflows and is used in related literature [13].

Rewritings should be done when a query task is detected that cannot process its incoming tuples with a rate higher than the arrival rate, e.g., when the computational complexity or the memory consumption is too high and therefore the task accumulates an increasing backlog. Such a task represents the critical path in the dataflow, limiting the overall throughput. For finding these paths, a rate-based optimization approach is suitable that scans the dataflows starting at source nodes and detects such bottlenecks based on monitoring information gathered during the execution [13]. After hot spots have been identified, one or multiple of the following methods can be applied for alleviating these bottlenecks.

[2]Usually, the separation of λ and τ is only conceptual and both functions are combined.

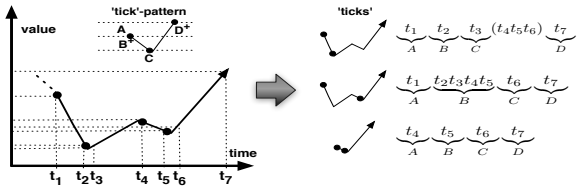

Figure 7: 'tick'-shaped pattern

Task sharing (Pattern 5): When multiple queries share the same (sub)graph to increase data locality [3] and the shared graph is on the critical path, this path can be replicated, sharing groups can be repartitioned, and tuples distributed to all replicas. This is the easiest way to remove burden from the critical path since inputs of sharing groups are independent from each other and no special dataflow transformations need to be performed.

Inter-operator parallelism (Pattern 4): Usually, it is better to keep dataflow operations on few nodes in order to avoid costly transfer operations. Hence, initially compiled queries will comprise few tasks consisting of large flow (sub)graphs. However, when the computational complexity exceeds a certain threshold or memory limits for keeping state information are exceeded, a distributed processing pipeline will yield better performance. Large graphs are partitioned, recompiled, and distributed on additional nodes in the cluster. Moreover, splitting large costly tasks into smaller distributed ones increases fault-tolerance since it will become cheaper to recover from node failures [11].

Intra-operator parallelism (Pattern 4+5): When the previous patterns not applicable, e.g., when operators are not shared or the graph has already been split into base operators, the last possibility to increase parallelism is partitioning input streams and processing each partition independent from each other with multiple operator instances. Unfortunately, this pattern is the most difficult to implement since its applicability depends on the actual operator type. Partitions in input streams need to be found, distributed to the operator instances, and their (partial) results have to be merged afterwards. Further, this pattern is prone to data skews and, therefore, some sort of load balancing has to be implemented, e.g., by monitoring the load of each partition and dynamically re-schedule partitions as proposed in [10]. This concept seamlessly integrates with the graph rewriting patterns described in this paper but again involves additional costs for transferring partitions among cluster nodes.

The most challenging problem in this context is finding suitable partitioning schemes for the operators that shall be deployed in the framework, especially when they are not stateless. In the following, we will present a parallelization scheme in the complex event processing (CEP) context which is prominent in stream processing.

The task of CEP is finding complex patterns in a stream of base patterns. These patterns are defined through certain properties of incoming tuples, usually described through predicates and additional correlations of their arrival time. Mostly, sequences and repetitions are used which can be expressed with regular expressions [15]. We demonstrate the parallelization on the 'tick-shaped' pattern example from [5] which is illustrated in Fig. 7. In the original publication, the task for detecting such patterns is originated in stock exchange trading, but it could also be applied for burst detection. A 'tick-shape' can be expressed by AB^+CD^+ where:

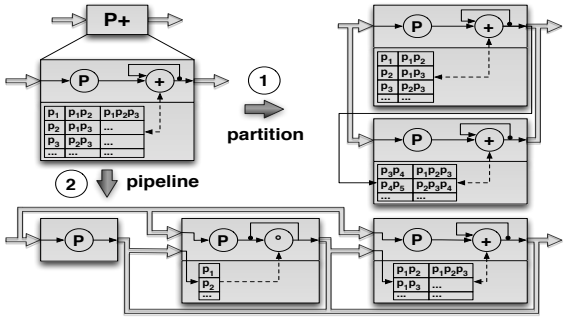

Figure 8: rewriting repetition operator

- A matches any incoming tuple
- B^+ matches all following tuples with decreasing value
- C matches the first tuple with increasing value after B but with a value less than the previous one of A
- D^+ matches following tuples with increasing values $> A$

For parallelization, we focus on the +-operator since it is challenging for three reasons: First, like a join, it can produce multiple output tuples per input tuple. Second, it needs to store state information for extending existing patterns to longer ones. Since each tuple is possibly multiplying the number of results it is likely that – due to memory constraints – such operators will become critical in the dataflow graph. Third, in most cases the behavior of the stream is not predictable, rendering static allocations infeasible.

Fig. 8 illustrates how the operator can be distributed dynamically to multiple nodes with simply applying graph patterns 4 and 5. The P+-operator comprises two parts: a pattern matcher P, and a +-operator which maintains all previously matched patterns as state and concatenates them with subsequent matches. The output of + is the output for the entire operator and serves as input for + again to construct longer matches. On memory overflow, the operator state can be **partition**ed and distributed to multiple instances where all instances receive the original input stream. Two different behaviors of the operator are required to avoid duplicates. The first instance processes the input directly, i.e., all matching tuples serve as new patterns of length 1. Those matches must not be reproduced by other instances that simply serve as targets for overflowing patterns that do not fit into the local state but are independent from each other and hence can be processed on separate nodes. When the complexity of the matching algorithm P is critical, elasticity can also be achieved with implementing a **pipeline**. It exploits the fact that P^+ can be expressed through $P \vee (PP^+)$, i.e., a pipeline of arbitrary length is constructed for matching incoming tuples in parallel, emitting them as output, and forwarding them to the next stage for extension.

Since such parallelization schemes depend on operator semantics, the framework provides them to automatically scale up and down required resources for built-in operators. For all user defined functions which are treated as black boxes by AnduINv2, the parallelization needs to be implemented by the user as in [11] or are provided through libraries that are linked as plugin to the execution environment.

4.4 Mesos Integration

Mesos [7] is a cluster management software for resource isolation and sharing. In Mesos, a master daemon (possibly supported by additional standby masters) manages a set of slaves nodes. An application (called framework) runs

tasks on these slaves which is initiated by so-called executors. Scheduling and resource assignments are managed by an application-specific scheduler. In order to support stream queries we implemented our own framework (cf. Fig. 6), providing an executor for running query executables (query tasks) on slave nodes and a *query scheduler* which gets resource offers from the Mesos master (available cores, memory, and network ports) and requests for executing AnduIN queries. Each query deployment request is described by a unique ID, the executable, and a specification of resource requirements, i.e., CPU cores, memory, and a list of query channels which have to be mapped to network ports. This specification is used by the scheduler to choose a slave node providing the requested resources for execution. Currently, only a simple strategy is implemented selecting the first offer providing the requested resources – more advanced strategies are subject of future work. If the scheduler has chosen an appropriate node, the request is forwarded to the corresponding executor. The scheduler assigns physical network ports to query channels and tracks these assignments to be able to connect subsequent queries referring to the same logical channel. In this way, a query implemented by one or more tasks can be deployed to one or more cluster nodes.

4.5 Meta Queries

Though, Mesos provides mechanisms to deploy processes, it does not support elastic operation for stream queries. In Hadoop, it is the task of the job tracker to partition the work across a set of map and reduce tasks. In case of data streams the situation is a bit different, because we cannot simply stop and continue/restart queries without loosing data. The only way to achieve elasticity is to change query behavior at runtime. Therefore, we introduce the idea of meta queries: in each (adjustable) query task an additional query is running on a control stream consisting of tuples of the form:

$$\langle \mathsf{query\_id}, \mathsf{operator\_id}, \mathsf{parameter}, \mathsf{value} \rangle$$

The control stream is produced by the query scheduler which monitors resource utilization and implements strategies for dynamic reallocation. Meta queries are particular useful for implementing the patterns described in Sect. 3. For instance, for failover without publish-subscribe (pattern 1 and 2), the network writer has to be informed about the network address of the newly activated standby node S. For this purpose, the network writer provides a parameter target-addr for the target address. A control stream tuple like

$$\langle \mathsf{query\#42}, \mathsf{writer\#2}, \mathsf{target\text{-}addr}, \text{"tcp://node2:6666"} \rangle$$

received by the query task triggers sending the tuple stream to the standby node node2. Similarly, for implementing pattern 3, the stream selector node can be informed about switching to the stream produced by query node S.

For partitioning patterns like pattern 4 it is either required to modify the tuple distribution strategy of multicast writers or to adjust partitioning predicates P_i in Fig. 3. Both can be easily implemented by sending appropriate control tuples.

5. CONCLUSION AND FUTURE WORK

We presented basic concepts how fault-tolerance and elasticity can be achieved in the context of continuous query processing by combining techniques that have proven applicability in other scenarios. These approaches are currently being integrated into AnduINv2, but can be applied in other platforms, too. Our main questions we would like to answer with future experiments are:

1) Which cost models are valid for online graph rewriting?
2) How can resource requirements for a query be estimated before actually executing it?
3) How can certain QoS guarantees be given to applications?
4) Can elastic stream processing benefit from heterogeneous clusters nodes?

While the first questions intent to pave the way for a streaming-as-a-service infrastructure, answering the last one is needed to keep up with current hardware development trends. We believe that parallel and specialized processors as many-core CPUs, GPUs, or FPGAs will find their way into future computing centers to provide the most efficient computing platforms for dedicated tasks – an important aspect to tackle the big data challenge.

6. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, et al. The Design of the Borealis Stream Processing Engine. In *CIDR '05*, 2005.

[2] A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *Database Programming Languages*. Springer, 2004.

[3] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. *SIGMOD Rec.*, 29:379–390, 2000.

[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI*, pages 21–21, 2010.

[5] N. Dindar, P. M. Fischer, M. Soner, and N. Tatbul. Efficiently correlating complex events over live and archived data streams. In *DEBS '11*. ACM, 2011.

[6] M. Eckert, F. Bry, S. Brodt, O. Poppe, and S. Hausmann. A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed. In *Reasoning in Event-Based Distributed Systems*. Springer, 2011.

[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, pages 22–22, 2011.

[8] S. Hirte, E. Schubert, A. Seifert, S. Baumann, D. Klan, and K. Sattler. Data3 - A Kinect Interface for OLAP using Complex Event Processing. In *ICDE*, 2012.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS*, 41:59–72, 2007.

[10] M. S. Joseph, J. M. Hellerstein, S. Ch, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, 2002.

[11] Q. Ke, M. Isard, and Y. Yu. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *EuroSys*, pages 15–28, 2013.

[12] D. Klan, M. Karnstedt, K. Hose, L. Ribe-Baumann, and K. Sattler. Stream engines meet wireless sensor networks: cost-based planning and processing of complex queries in AnduIN. *Distrib. and Parallel Databases*, 29:151–183, 2011.

[13] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD '02*. ACM, 2002.

[14] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud '12*. USENIX Association, 2012.

[15] F. Zemke, A. Witkowski, M. Cherniak, and L. Colby. Pattern matching in sequences of rows. Technical report, ANSI Standard Proposal, 2007.