2011 Third IEEE International Conference on Coud Computing Technology and Science

Scalable and Low-Latency Data Processing with StreamMapReduce

Andrey Brito
Universidade Federal de Campina Grande
Campina Grande, Brazil
E-mail: andrey@dsc.ufcg.edu.br

André Martin, Thomas Knauth, Stephan Creutz,
Diogo Becker, Stefan Weigert, Christof Fetzer
Technische Universität Dresden
Dresden, Germany
E-mail: {firstname.lastname}@inf.tu-dresden.de

***Abstract*—We present StreamMapReduce, a data processing approach that combines ideas from the popular MapReduce paradigm and recent developments in Event Stream Processing. We adopted the simple and scalable programming model of MapReduce and added continuous, low-latency data processing capabilities previously found only in Event Stream Processing systems. This combination leads to a system that is efficient and scalable, but at the same time, simple from the user's point of view. For latency-critical applications, our system allows a hundred-fold improvement in response time. Notwithstanding, when throughput is considered, our system offers a ten-fold per-node throughput increase in comparison to Hadoop. As a result, we show that our approach addresses classes of applications that are not supported by any other existing system and that the MapReduce paradigm is indeed suitable for scalable processing of real-time data streams.**

I. Introduction

The problem of processing huge amounts of data has been around for decades. However, relatively recent technologies like the Internet of things, mobile computing, social networks, and smart societies (e.g., omnipresent sensor networks) create the more challenging problem of processing data continuously as it is being produced or updated.

Event Stream Processing (ESP) is a set of techniques to address such classes of problems [9], [13]. ESP targets applications that both require the processing of large amounts of data and low processing latencies. One application scenario for such systems is when too much data is generated, but only a fraction of the data can be stored. This is the case, for example, for the computational systems in the Large Hadron Collider at CERN [21]: data is produced at a higher rate than can be stored and, thus, has to be filtered or summarized in real-time. Another common scenario is system monitoring. The observed data must be analyzed on-the-fly in order to trigger adequate reactions. High frequency (algorithmic) trading and real-time fraud detection are two such examples.

ESP applications are expressed as a directed graph of operators. The operators are then traversed by the data as it arrives and the individual pieces of data are referred to as events. Some examples of operations are the following: filtering (discarding unimportant events from the data stream); transformation (applying a function over the individual events as for format conversion); aggregation (summarizing a set of events in a single, more abstract event as in an average or a detected pattern); join (combining events from different streams). Note that some of these operations depend only on the current input, while others depend on a state accumulated during the processing of previous events.

ESP can be considered a spin-off from databases. Traditional databases are, however, not suitable for the types of applications ESP addresses. First, conventional databases need to store and index the data before queries are executed, which introduces considerable latency and makes it inefficient for processing data that is used only once. Second, databases offer a much richer set of features (such as transactions), which hinders scalability. Finally, databases typically consider all the data when processing a query and produce a precise result. In the stream case, the amount of data is unbounded and operations need to be adapted, for example, to consider only recent inputs or to produce approximations as outputs.

If, on the one hand, conventional databases have scalability and latency problems, on the other hand, some batch processing systems (like MapReduce), scale very well but also do not provide low latency. Consider, for example, the recent popular approach from MapReduce [7] and its major open-source implementation, Hadoop [1]. Hadoop considers a staged approach where one stage (equivalent to one operation in the graph of operators) can start only after the preceding stage has finished (see also Figure 1). This approach not only inserts considerable latency, but also can limit scalability: if Hadoop can scale to N_i hosts in stage i and N_{i+1} in stage $i+1$, a system could scale to $N_i + N_{i+1}$ hosts if stages could be active in parallel (as is the case with StreamMapReduce).

Unfortunately, current ESP systems are also not able to handle next-generation applications. They provide low latency, fault tolerance, and are simple to use (applications are normally expressed in an SQL-like language or directly as a graph of operators) but provide limited scalability. For example, in GSDM [11] and StreamFlex [18] operators are assumed to be stateless, i.e., the processing depends only on the current input (being this input a single event in StreamFlex and a set of events in GSDM). Scaling stateless operators is easier because operators can be simply replicated. For stateful operators, Borealis [20] addresses scalability using load-shedding techniques and StreamMine [3], applies a speculation-based

Most of this work has been carried out while Andrey Brito was at the Technische Universität Dresden.

978-0-7695-4622-3/11 $26.00 © 2011 IEEE
DOI 10.1109/CloudCom.2011.17

parallelization approach that scales only among processing cores in a single node.

In this paper, we address two very important issues for the implementation of high scale ESP applications: the lack of *responsiveness in batch processing* systems and the lack of *scalability in ESP systems*. On one side, the lack of responsiveness in batch processing systems is mainly consequence of design decisions regarding fault-tolerance and data movement. In traditional MapReduce implementations, fault tolerance and data movement is handled by storing intermediate results in a filesystem. On the other side, lack of scalability in ESP systems is due to the lack of simple ways to express the availability of parallelism in applications.

In summary, our contribution is twofold. First, we bring the MapReduce programming style, and its consequent scalability, to ESP applications. Second, we bring the ESP-style responsiveness to MapReduce jobs without requiring modifications to their familiar programming model. We show that this combined approach supports classes of applications that were not previously well served by any existing system.

The rest of the paper is organized as follows. In Section II, we briefly overview ESP and MapReduce and give an application example that will be used throughout the paper. Section III discusses the programming model and the main design decisions in StreamMapReduce. Scalability and responsiveness of our system is evaluated in Section IV, where we extrapolate real workloads from telephony and banking domains to illustrate workloads expected in the near future. Finally, related work is discussed in Section V and Section VI concludes the paper.

II. Background

In this section, we review the basic concepts of both MapReduce and ESP. We also classify applications according to their responsiveness requirements and detail a simple example of operation commonly found in all of these classes.

A. MapReduce

The MapReduce programming model consists in a sequence of stages that transforms a set of input data into a set of output data. A stage is composed by a set of nodes and all the nodes in a stage execute either a map or a reduce operation.

A node in a map stage, i.e., a *mapper*, receives a set of inputs and produce a set of outputs in the form $(key, value)$. The computation in the mapper is stateless, an output depends only on the current input. A node in a reduce stage, i.e., a *reducer*, receives a set of $(key, value)$ pairs and uses a reduce function to aggregate all values regarding a specific key into a single $(key, value)$ output. To conform with the event processing terminology, we will refer to a $(key, value)$ tuple as an *event*. For events composed of several attributes, the key can be a composition of the primary attribute values, the value then contains the complete set of attribute names and values.

A MapReduce implementation needs to guarantee that all $(key, value)$ tuples produced for a specific key k_1 will be delivered to a single reducer. It also needs to guarantee that the reducer is only executed when all $(key, value)$ pairs for key k_1 are available to the respective reducer when it starts to execute. Therefore, conventional MapReduce is a staged approach that requires a stage to be completed before the following stage can be started.

The parallelization and scalability power of MapReduce is a consequence of the fact that mappers are stateless and that different reducers are independent (they process different keys). The communication pattern is illustrated in Figure 1: any mapper can produce an event that will be sent to a reducer (based on the event's key) and a reducer receives all the events related to a specific key.

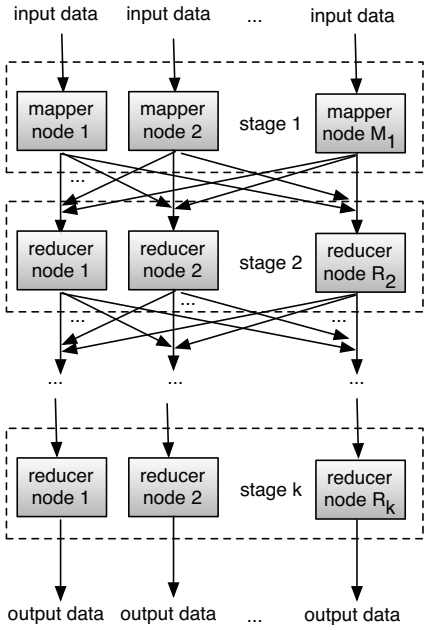

Fig. 1. MapReduce processes data in stages. The number of nodes per stage can be tuned to maximize throughput and to minimize latency.

The traditional example for the MapReduce programming model is shown in Listing 1, counting the number of occurrences of a word in a large corpus of text. For each word in document doc the mapper will emit an event in the form $(word, 1)$. Eventually, a reducer will receive a list of all such events for an specific word and will add them up, emitting a final counter value for this word.

```
1  map(Key k, Document doc) {
2    for each word in doc:
3     emit(word, 1);
4  }
5
6  reduce(Key word, List counts) {
7    int total = 0, count;
8
9    for each count in counts:
10    total = total + count;
11
12   emit(word, total);
13 }
```

Listing 1. The word frequency count algorithm for MapReduce.

B. Event Stream Processing

As mentioned in Section I, an ESP application is a graph of operators. Some operators are stateless, but other operators are stateful. The stateful operators produce outputs that are derived from both the current event and the current operator state, which was accumulated during the processing of previous events. However, it is not possible to consider all events. For blocking operators, such as computing the average of an attribute value for the events, if all the events in the (potentially infinite) stream need to be considered, the operator will never finish. In addition, there are other cases when only the recent events are of interest (e.g., the average temperature of a piece of equipment in the last 10 seconds). To solve these problems, event windows are normally used. The two most common ways to define event windows are time (e.g., the events in the last 10 seconds) or a counter (e.g., the last 10 events).

Another distinguishing feature of ESP applications is the responsiveness requirements. The main motivation for using ESP techniques instead of other technologies (like databases) is the need for low processing latency. Nevertheless, not all applications require the same latency requirements. Consider the following examples:

- In the LHC example mentioned earlier, huge amounts of data are continuously produced. The software system [21] has *40 milliseconds* to decide if a set of measurements from a specific area in the sensor chamber should be forwarded for further analysis or discarded. This decision is made based on how "interesting" the points appear to be and also on how much data has been sent already (the system should neither be overloaded nor underloaded).
- In a real-time fraud detection systems (e.g., for credit cards or telephony), a transaction can be approved or rejected based on previous transactions and on a set of business-specific rules. This real-time analysis should not introduce considerable delays to the existing transaction processing system and, therefore, should not take more than *100 milliseconds*.
- In an algorithmic trading system, stock market events may trigger buy or sell actions. Different competitors will have different strategies and processing systems, but for a specific situation the system with the lowest response time will have the biggest profit. Therefore, responsiveness requirements for high frequency trading systems are in the order of *10 milliseconds*.

A simple classification for responsiveness requirements and a representative example for each class is shown in Table I. In this work, we do not consider applications in class VI, which can be adequately handled by existing batch systems. Applications in classes III to V could be handled by conventional databases, but only for moderate scalability and event input rates. We consider workloads of current and forthcoming applications that cannot be supported by the scalability levels provided by conventional databases. Classes I and II require ESP techniques. Applications with responsiveness in the sub-millisecond range exist, but normally depend on custom, highly-tuned solutions (e.g., hardware implementations [22]) and, thus, are not considered here.

Class	Response time	Example
I	~ 5 ms	Algorithmic trading
II	~ 50 ms	Credit card fraud detection
III	~ 500 ms	Mobile phone fraud detection
IV	~ 5 s	Equipment/service monitoring
V	~ 50 s	Continuous data analysis
VI	> 500 s	Traditional data analysis (batch)

TABLE I
APPLICATION CLASSES AND RESPONSIVENESS REQUIREMENTS.

Finally, we will use a moving average operator as a running example. The moving average is a good example because it is commonly found in different applications classes, like computing price averages in algorithmic trading or average temperatures in an automation system. In addition, it is common both with jumping windows, which considers non-overlapping windows, and with sliding windows, in which only a portion of the window is updated at a time. Lastly, it maps well to the MapReduce programming model (specially if we assume that different averages are kept for different stock names or for different pieces of equipment, as in a *group-by* clause in an SQL query). These characteristics of the moving average allow us to consider a wide range of implementation options, and, thus, to provide an intuition of how different operators can be implemented and are expected to behave.

III. STREAMING MAPREDUCE

Our main goal is to provide low-latency processing for data that is arriving in multiple streams. To achieve this goal, we propose a MapReduce-based programming model that is extended for ESP. We want the programming model to be familiar to the MapReduce user, while still providing flexibility to the advanced user who wants a highly-tuned solution. In addition, we also discuss some key aspects of our system, such as state management and fault tolerance.

A. Programming model

In our programming model, we keep the concept of mappers and reducers, but reinterpret their meanings to match the ESP terminology. A mapper is therefore any stateless component and can implement operations like filtering, where events are discarded or forwarded based on their attribute values, and transformation, applying a function over event attributes and generating one or more output events.

Stateful operators are implemented as reducers. Reducers can be further divided according the type of their state. *Windowed reducers* consider a state composed of a window of events. *Stateful reducers* consider an arbitrary state that is customizable by the user. In the following, we illustrate the programming model through examples.

In Listing 2, we detail the implementation of a simple moving window using the MapReduce approach. The listing includes a configuration comment. In fact, the configuration would be specified in a separate file, where the user also

specifies the processing stages and their connections. In the example, the configuration means that function $map1$ is a mapper (i.e., stateless) and that $reduce1$ is a reducer that uses a jumping window with the events that arrive within a 1-minute interval. Specially important in this example is that the interface for the reduce function is exactly the same as in the original MapReduce. The same code that is used to compute the average of a set of data files can now be used to compute the average of data arriving in a live stream.

```
// map1 := MAPPER
// reduce1 := REDUCER + J_WINDOW<1 minute>

map1(Key k, Value v) {
   // extract new key (k1) and value (v1) from (k,v)
   emit(k1, v1);
}

reduce1(Key k, List values) {
   int sum = 0, count = 0, v;
   for each v in values:
      sum += v;
      count++;
   emit(k, sum/count);
}
```

Listing 2. Simplest moving average with a jumping window.

Another option would be to have the operator use a sliding window instead of a jumping window. Note that the only difference would be the configuration (the window type is specified as $S\_WINDOW<1\,minute, 6\,seconds>$). There is, again, no difference between the original MapReduce code and the streaming version. The windows are now updated each 6 seconds and consecutive windows overlap in 90%. The internal handling of the events is then aware that events are to be reused, but, nevertheless, the computation needs to be completely redone for each window.

The next two examples consider the extended interfaces, which enable more efficient solutions while still being intuitive to both the MapReduce and the ESP programmer. The pseudocode in Listing 3 illustrates a reducer for a sliding moving average with incremental computation. The reducer function has now access to which events should be removed and which should be added to the incrementally-computed average.

```
// reduce3:= REDUCER + S_WINDOW_INCREMENTAL<1 min, 6
      seconds>

reduce3_init(k1) { // first time key k1 is received
  S = new State(); // custom user class
  S.sum = 0; S.count = 0;
  return S; // object S is now associated with key k1
}

reduce3(Key k, <List newValues, List expiredValues, List
       commonValues, UserState S>) {
   for v in expiredValues do:
     S.sum -= e; // Remove contribution of expired events
     S.count--;
   for in newValues do: // Add contribution of new events
     S.sum += v
     S.count++;
   emit(k1, S.sum/S.count);
}
```

Listing 3. Enhanced moving average with a sliding window.

In our last example, shown in Listing 4, we illustrate an operator where the state is fully controlled by the user. In this case, the $reduce4$ functions is called for each new event and there is another function, $ttreduce4$, to be executed periodically with a timer. Both the regular reducer and the *time-triggered* reducer functions have the same interface so that the same function can be used in both roles if desired. In addition, because both reducers use the same state, if both are ready to be executed (i.e., an event arrived at the same time that the timer expired), their execution will be serialized. Note that once a key has being initialized and until it is explicitly destroyed (by a call to a special API function), the time-triggered reducer function is periodically invoked. This periodic function is necessary to implement operations where the absence of events also needs to be considered.

```
// reduce4:= REDUCER + STATEFUL
// ttreduce4:= TT_REDUCER<1 minute>

reduce4_init(Key k1) { // first time key k1 is received
  S = new State(); // custom user class
  S.sum = 0; S.count = 0;
  return S; // object S is now associated to key k1
}

reduce4(Key k, <Value v, UserState S>) {
   S.sum += v; S.count++;
}

ttreduce4(k1, <Value unused, UserState S>) {
   emit(k1, S.sum/S.count);
   S.sum = 0; S.count = 0;
}
```

Listing 4. Enhanced moving average with a jumping window.

The first three programming alternatives above are explicitly window based. This information is important for the underlying system because it enables some optimizations regarding fault tolerance and load balancing to be transparently made. For example, migrating an window-based operator can be done by simply redirecting events belonging to the next window to the new operator location. Similarly, for a sliding window, some events will be duplicated in the two nodes until the destination node has the complete window, when it can then transparently take over the following computations. For the forth alternative, transferring (or recovering) an operator includes not only redirecting the events, but also transferring the complete operator state.

StreamMapReduce also supports a combiner function as in traditional MapReduce. The combiner function works as a (local) reducer stage that computes an additional aggregation over all keys of a node and reduces the downstream traffic by combining multiple local keys and states before emitting events to the next stage.

Finally, the role of the mapper can be more significant than in the examples above. Besides filtering and converting events, mappers are also used to duplicate events whose keys map to two or more partitions of the global state. For example, our evaluation includes a mobile phone fraud-detection system in which a community of interests (COI) is kept for each user. A user's COIs is updated when a new call detail record is received that include the user either as an origin or as a destination. The mapper then needs to duplicate the input call detail record events so that the two reducers associated with the two users receive the update.

B. Event processing and state handling

In order to be able to exploit the growing number of cores in a single node, a StreamMapReduce node has a thread pool. Exploiting parallelism when considering stateful components is particularly difficult. Here, the MapReduce approach helps to make the code more parallelizable. Unfortunately, not all algorithms map well to this approach and other techniques may have to be considered (e.g., explicit parallel coding). Nevertheless, even for MapReduce, ordering events and managing the state should not be neglected (specially if fault tolerance needs to be considered).

The major steps in processing an event in a reducer (i.e., stateful) operator are shown in Figure 2. First, threads are responsible for dequeuing events from the input queues. When an event arrives, a thread from the pool deserializes the event and calls the Sequencer component. As the name suggests, the *Sequencer* serializes the execution of events (potentially coming from different connections) that refer to the same key and, thus, access the same portion of the state.

After a thread is permitted to proceed by the Sequencer, the thread retrieves the corresponding piece of the state from the key table. As discussed in the previous section, this state can be a window of events (Listings 2), a user-provided state (Listing 4), or a combination of both (Listing 3). If necessary, the lists with new, evicted, and common events, are updated. The next step is to call the user provided reducer function. Finally, if the user provided reducer generate output events, these are handled by the finalizer, which manages buffering and sending of events (again, buffering is needed for fault tolerance, as discussed in the next section).

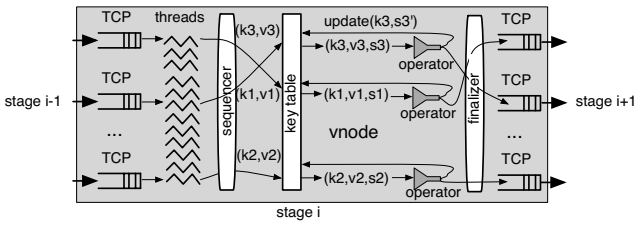

Fig. 2. Internal structure of a node: threads receive the events; a sequencer serializes the threads with events for the same key; the key table keeps the per-key state; and the finalizer handles buffering and sending of the events.

Still regarding the reducer nodes, one aspect is not depicted in the figure above: the usage of the time-triggered reducers. These reducers are triggered by timers and consider no event from the input queues. Nevertheless, timer events are also handled by the thread pool as the mechanism is similar. When a timer for a time-triggered reducer expires, it carries the corresponding key. A thread from the pool handles this expiration and goes through the sequencer. The execution of the time-triggered reducer needs to be serialized with regular reduce operations on the same key because both use the same portion of the state. The thread then retrieves the state and executes the user-provided time-triggered reducer function.

Finally, processing in a mapper operator is similar. However, because mappers are stateless the sequencer and the key table components of Figure 2 are not present. As before, the finalizer handles the buffering and sending of events.

C. Fault tolerance

Fault tolerance implementation requires several trade-off decisions regarding the performance overhead for the failure-free runs, the recovery time in case of a failure, and the additional resource cost:

- For applications in classes I and II approaches based on rollback-recovery (i.e., keeping logs and checkpoints [8]) are infeasible. These approaches add considerable latency for failure free executions and, more importantly, impose a prohibitive recovery time. For this case, active replication (e.g., [17]) is the best choice.
- For large clusters, the resource cost of active replication is typically prohibitive.
- For applications in class IV and above, rollback-recovery solutions are ideal. Because checkpoints do not need to be frequent, the overhead is minimized. This is also the case when small to medium clusters are used (i.e., up to a few hundred nodes) and failures are rare enough to make an unavailability in the order of seconds to be acceptable.

In a summary, no simple solution will be satisfactory for all classes of applications and a complex solution is out of the scope of this paper. Nevertheless, as many design decisions in MapReduce systems were considerably influenced by the need for fault tolerance, we present an approach that handles at least all application classes that these systems can handle.

For this fault-tolerance approach we assume a cluster in a local area network (connected by a Gigabit Ethernet) running an NTP clock synchronization service. In such a scenario, clocks in different nodes will have synchronization errors of only a few tens of microseconds. Because of the assumption of a medium cluster, we optimize for the case that nodes of two neighboring stages (i.e., the upstream and the downstream node) do not fail at the same time. The connection between one upstream node and one downstream node is assumed to be reliable and ordered (FIFO).

A reasonable approach for fault tolerance in such a scenario seems to be the following. First, nodes timestamp output events. Downstream nodes then enforce this timestamp order in the processing. Second, nodes execute periodic checkpoints. Third, upstream nodes keep copies of their messages (writing to disk, if available memory is not enough) in order to replay these messages in case of a failure in a downstream node. Finally, when an upstream node fails, the receiving node logs that it will stop waiting for the absent node. When a node finishes checkpoint it acknowledges the upstream node, which can then truncate its output log, discarding the messages included in that checkpoint.

Unfortunately, the reasonable approach above does not perform too well in our measurements. This happens because of small fluctuations in the timeliness of upstream nodes cause the receiving node to be blocked for short periods too frequently.

The key point in our approach is inspired in virtual synchrony [2]. This approach is detailed and evaluated elsewhere [15], but it is summarized below for self containment. We enable nodes to process events that are in the same epoch without having to block to enforce timestamp ordering or to log processing order. Instead of a single queue for input connection, a node has two input queues, named $current\_epoch$ and $next\_epoch$. In addition, each node has a variable ($next\_checkpt$) with the time when the next checkpoint will be executed (checkpoint intervals are defined by $chkpt\_interval$). Upon the reception of an event, the thread that is listening to the connection has three options: (i) if the timestamp of the event is below the time of the next checkpoint, it inserts the event in the $current\_epoch$ queue; otherwise, (ii) if the timestamp of the event is after the time for the next checkpoint, but still belongs to the following epoch (i.e., between $next\_epoch$ and $next\_epoch + chkpt\_interval$), the event is inserted in the $next\_epoch$ queue; last, (iii) if the timestamp of the event is too far in the future (i.e., beyond $next\_epoch + chkpt\_interval$), it triggers a back pressure that slows down the fast node. As soon as an event from each upstream node is inserted in the $next\_epoch$ queue, the current epoch is finished and the state can be checkpointed.

Using the above approach, instead of having the processing in a node blocked whenever an upstream node is slightly late, the processing is blocked only when the lateness occurs during the change of an epoch.

Finally, for application classes requiring better recovery times or when failures are frequent, in [14] we detail an approach based on active replication.

IV. Evaluation

In this section, we evaluate our StreamMapReduce framework. Our goal was to design a system that provides the scalability of the MapReduce programming paradigm and the responsive of an ESP system.

Our experiments were executed on 384 processing cores (distributed among 48 machines). The machines are connected via Gigabit Ethernet and have 2 quad-core processors each (Intel Xeon E5405). In addition, machines are also equipped with 8GB RAM and 7200 RPM disks attached via SATA2. The operating system is Debian Linux 5.0.

A. Programming model microbenchmarks

We start by evaluating the performance of a simple ESP benchmarks using the extended MapReduce programming model. As we discussed in Section II, the moving average operator can be considered a canonical operator: (i) it is used in different application classes; (ii) it is used both with sliding and jumping windows; (iii) it is partitionable; and (iv) it the base for many different aggregation operations over windows.

Regarding the key attribute in the events, we use 10000 different keys, which could represent different sensor or user ids. We use time-based windows and the length of the window is the same as the required responsiveness. In other words, in an application with 5 s responsiveness, we assume windows are likely to have a length in the same order of magnitude as 5 s. In the case of sliding windows, 90% percent of a window is propagated to the next windows (e.g., for a window length of 50 seconds, a new window is created after 5 seconds).

The first set of experiments considers the three alternatives the developer has when writing an stateful operator in StreamMapReduce (see Section III):

1) The *traditional MapReduce interface* (Listings 2) forces the computations to be executed when the window is closed. In the case of sliding windows, it also requires that the windows are completely recomputed although only part of the window has changed.
2) The *extended interface for sliding windows* (Listing 3) requires the adaptation of the original MapReduce operator. However, because incremental computations are allowed, efficiency can be considerably improved.
3) The *generic interface for stateful operators* (Listing 4) gives the developer flexibility to write the stateful computations: considering incremental computations or not.

Figure 3 illustrates the performance of a moving average operator implemented using different approaches and systems. We initially compare the performance of the traditional and the extended versions of the MapReduce programming interface when used to process streams of events. The first interface is used for implementing both a jumping window and a sliding window. This implementation is referred to as *basic*. Note that the achievable throughput for the sliding window is considerably lower. This is consequence of the fact that, for this implementation of the sliding window, although 90% of the values are the same between two consecutive windows, the whole window is reprocessed each time a window closes.

The second implementation of a sliding window, whose performance is referred to as *enhanced* in Figure 3 (right-hand side), considers the extended window interface. For large windows, this configuration offers a higher throughput compared to the basic interface as it allows the reuse of partial computations. Finally, the second implementation of a jumping window (shown as *enhanced* on the left-hand side of the figure) considers the generic stateful interface. In this case, incremental computations can be done as the events are received, resulting in a higher throughput as the computation is not deferred to when the window is closed.

We next consider a jumping window and compare its implementation in three different systems. Note, however, that building an ESP application with Hadoop requires it to first store the incoming events for the current window into files in the Hadoop filesystem (HDFS). After the window is complete, the Hadoop job can then be started to process that window. We simplify this scenario by not considering the storage step as it would require making assumptions on how many streams will be processed (different streams could be written in parallel).

Also included in Figure 3 is the average throughput of Hadoop and Hadoop Online (HOP [5]) when computing a jumping moving average with the window lenght and responsiveness constraints specified on the horizontal axis. HOP modifies the regular Hadoop to enable communication and

processing to be performed in a more streamed fashion. As mentioned above, the basic StreamMapReduce versions use the same programming interface as Hadoop.

The results show that Hadoop is not able to support an ESP application requiring responsiveness of 5 seconds or below, not even for windows with a small quantity of data. This is consequence of the job-starting overhead. It takes up to 30 seconds from the triggering of a job until the actual processing starts. HOP can keep up with smaller windows (and latencies). First, HOP has a *pipeline* feature that sends results from the mapper nodes directly to the reducer nodes. Second, the startup overhead can be overcome by programming reducer nodes to be run periodically (referred to as *snapshots*) and to truncate its input buffer after each execution. Unfortunately, HOP implementation sustains only very low throughput values.

In a summary, on the one hand, the basic MapReduce interface in StreamMapReduce is simple as it does not require changes to MapReduce operators. On the other hand, more sophisticated interfaces can provide more efficient solutions for the cases where the computation is considerably expensive and performance needs to be improved. Furthermore, neither Hadoop nor HOP (equipped with the pipeline and snapshot features) perform well enough for highly responsive (i.e., low latency) applications.

B. MapReduce benchmarks

In the previous section, we evaluated the MapReduce programming approach in ESP benchmarks. In this section, we evaluate StreamMapReduce with the benchmarks used in the related work for MapReduce: word frequency count jobs.

The word frequency problem was described in Section II and consists in counting how many times each word appears in a large corpus of text. As other works, we use the Wikipedia[1] data repository as text source.

In Figure 4, we show the job completion time for different input lengths for Hadoop, HOP, and StreamMapReduce. In contrast to the experiments in the previous section, using the pipelining feature in HOP *without* the snapshort feature improves its efficiency, resulting in a lower completion time.

As shown in the figure, again, neither Hadoop nor HOP can provide good responsiveness – even for small-sized input data. The graph shows that the StreamMapReduce is considerably faster. Detailed experiments with Hadoop and HOP have shown us that large improvements would be possible if the framework were implemented in C++. In this way, more functionalities can be tuned for the classes of loads expected in ESP applications. As an example, hash maps are heavily used and having efficiently allocated hash maps considerably decreases manipulation costs for these structures. For example, good memory layout decreases the overhead of writing checkpoints, updating states, and transferring them across the network. Therefore, the good performance of the word frequency count benchmark in StreamMapReduce is partly because StreamMapReduce is implemented in C++ and

[1] http://www.wikipedia.org

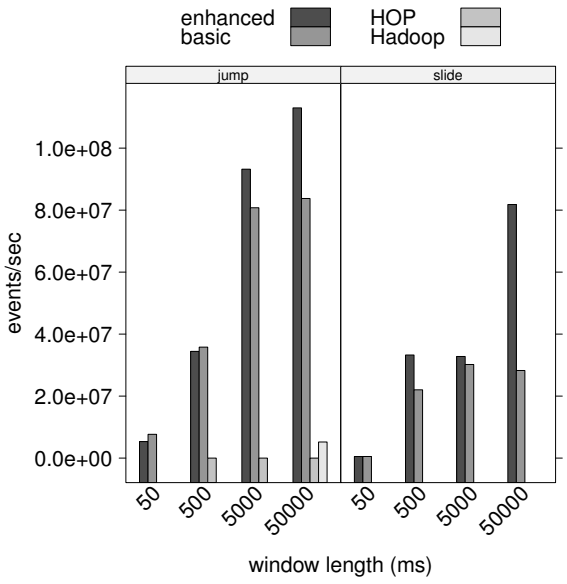

Fig. 3. Comparison between different implementations of a moving average.

also because StreamMapReduce (1) has effective multicore support, (2) we do not write intermediate results to disk, but (3) write only the state of the combiner and reducer to disk (see Section IV-D).

Note that the measurements in HOP's original paper ([5]) show a per-node throughput for word frequency count of about $1.77\ MB/s$ for Hadoop and $2.37\ MB/s$ for HOP. In our experiments (Figure 4) we measured $3.6\ MB/s$ for Hadoop and $6.2\ MB/s$ for HOP. In contrast, StreamMapReduce has a per-node throughput of between $75\ MB/s$ to $80\ MB/s$ (see Section IV-D).

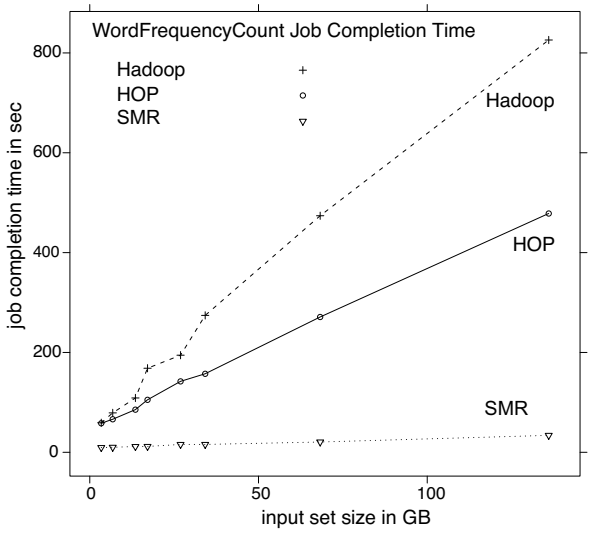

Fig. 4. Total job times for different input sizes of the word frequency count benchmark (batch).

C. Application scalability benchmarks

In this section, we consider three benchmarks. In all cases, we do not have sufficient data and hence, the data has been extrapolated (by concatenation) in order to provide enough input to feed the system throughout a complete experiment.

1) Service level agreement (SLA) conformance monitoring for mobile phone recharge systems: the experiment has been performed using real as well as synthetic data with similar characteristics. A recharge operation triggers several events in the underlying systems. The goal is to monitor these patterns (of 3 to 4 events each, happening within a time window of 10 seconds) and to trigger adequate reactions and monitor success rate and time for completion. The response time requirement is around 5 seconds (class V application).
2) Fraud detection in mobile telephony: we have $2.9\ GB$ of data, corresponding to over 40 million call detail records. The system continuously keeps the community of interests (COI) of a user (most frequent contacts and their typical call durations; see [6]). The goal is to detect frauds early by (1) issuing alerts when a user abruptly deviates from their normal behavior, and (2) detecting the COIs of known fraudsters. There is a trade-off between accuracy and responsiveness and, therefore, the exact deadlines are not yet defined.

SLA conformance monitoring application consists of two stages: (1) the "source" stage (S) parses the input data, and (2) the "worker" stage (W) consists of the SLA operator. In our measurements, the source stage reads the input data from local disk, parses the data, and sends the events across the network to the worker node that processes this data: each SLA event has a transaction id that we use to partition the processing of the events.

Figure 5 illustrates the per-node throughput (for different number of total nodes), the cumulative throughput over all workers, and the total number of SLA events per second for all nodes for four different configurations. The number of nodes (x-axis) is the number of source plus worker nodes. Configuration "1S:1W disjoint" means that the source and worker stage use a disjoint set of hosts and there are the same number of source and worker nodes. The other configurations use co-locations of the source and worker stage, e.g., (2S:1W) means that each node runs 2 source operators and 1 SLA operator. Note that in the case of co-location on N nodes, about $1/N$ of the traffic between the two stages is local and $(N-1)/N$ of the traffic has to go across the network.

In the case of the SLA conformance monitoring, the best configuration is to co-locate 3 sources and 1 worker on the same node. We achieve almost $45\ MB/s$ per-worker throughput which results in almost $2\ GB/s$ total throughput in our 48 node cluster. Note that this throughput translates to the processing of more than 80 million SLA events per second.

The COI application also consists of a source and a worker stage. We evaluated additional co-located and disjoint configurations (see Figure 6). For the COI application, we achieve the best per-worker node throughput by having three source nodes per worker host. For the co-located configurations with multiple sources per node, there is more resource contention which leads the worker stage to enforce stronger back pressure. In turn, this leads to throughput per worker node of less than $40\ MB/s$. The total throughput is still best for the configuration in which we co-locate one source and one worker per host: we achieve more than $1500\ MB/s$ in our 48 node cluster. This translates into more than 20 million calls per second.

Finally, in Figure 7 we evaluate the latency of our operators. The curve on the left-hand side of the figure depicts the end-to-end latency (from the source, through the SLA operator, arriving at a sink operator that consumes the data) for a per-node generated event rate. As the generation rate reaches the processing limit of the system the latency suddenly increases. Similarly, the right-hand side of Figure 7 depicts the latency for the COI system. In this case, the computation is more complex and latency is also influenced by event-batching levels in the communication subsystem.

D. Fault tolerance

In our last set of experiments, we demonstrate the impact of checkpointing and the recovery time for the word frequency benchmark.

Figure 8 depicts the impact of the checkpointing on the throughput and over time of an application running on the 48 nodes with and without periodic checkpoints. Note that after second 30 a node failure is forced and after 3 seconds another node (two nodes are currently kept as spare) restores the latest checkpoint of the failed node and continues with the computations.

V. Related Work

The MapReduce approach [7] has become very popular. The main reason is that once the user writes his program using the map and reduce functions the infrastructure can take care of scaling the processing to a large number of nodes. The MapReduce success motivated several implementations, Hadoop [1] being the main open-source implementation. Several variations of the model have also been proposed. Hadoop Online [5] improves Hadoop's efficiency by enabling a direct communication between mappers and reducers. This modification improves Hadoop's performance as it increases the overlap between the map and reduce phases. In their original publication, the authors consider the processing of continuous data streams, but provide only a minimal example and no performance evaluation. Our studies show that HOP does not yet provide adequate support for continuous data stream applications.

Among other MapReduce variants, the Continuous Bulk Processing (CBP) system [12] incorporates state into MapReduce, as we do. However, their goal is to provide better support for incremental batch jobs and, thus, do not support applications requiring low latency. Better incremental support for batch systems is also the focus of the new Google system,

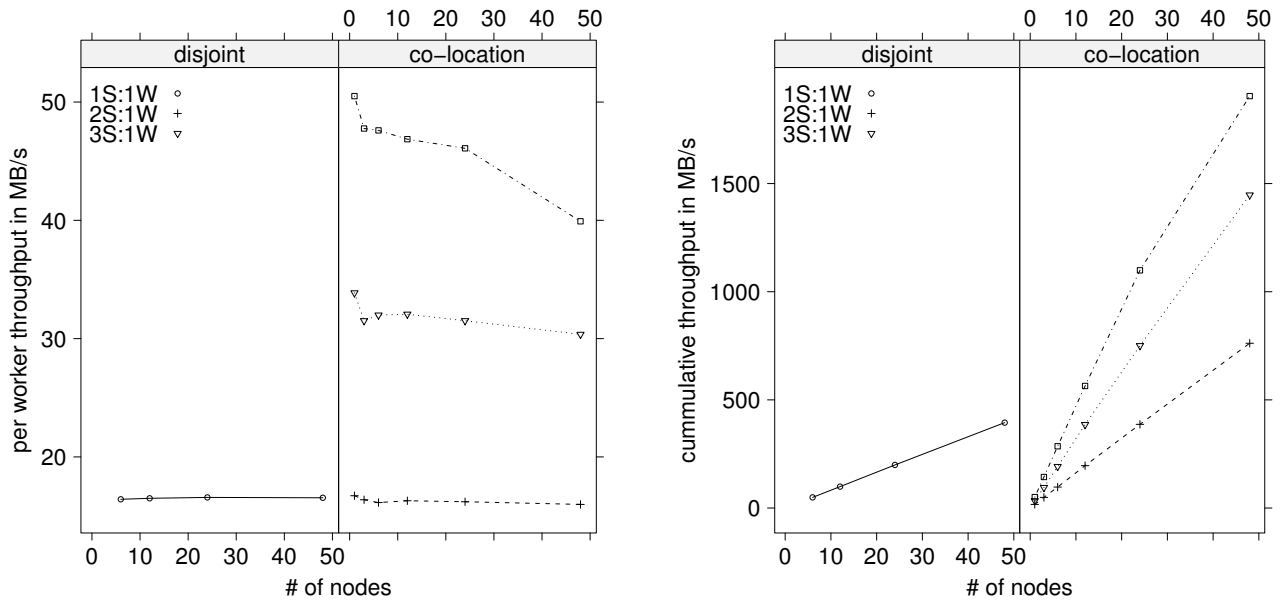

Fig. 5. Scalability of the SLA conformance application: (a) per-worker node throughput depending on the total number of nodes, (b) the cumulative throughput depending on the total number of nodes and different configurations.

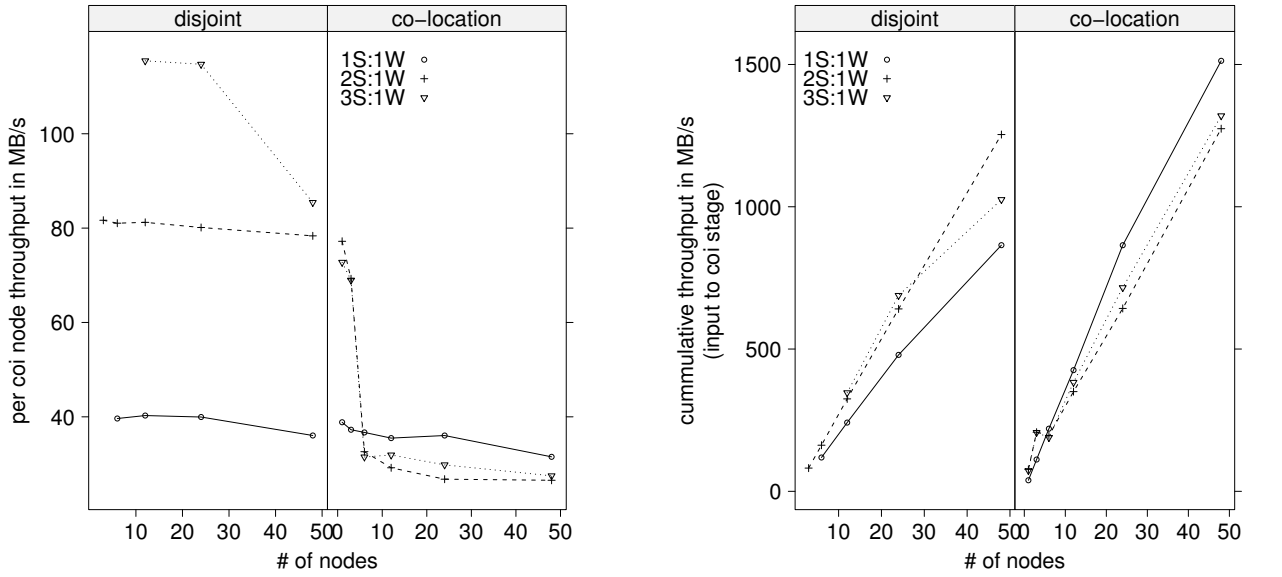

Fig. 6. Community of Interest computation: (a) per-worker node throughput, (b) cumulative throughput in MB/s. Each for different number of sources (S) and COI operators (W). Left column: sources and operators are on disjoint nodes, right column: sources and workers are co-located on the same hosts.

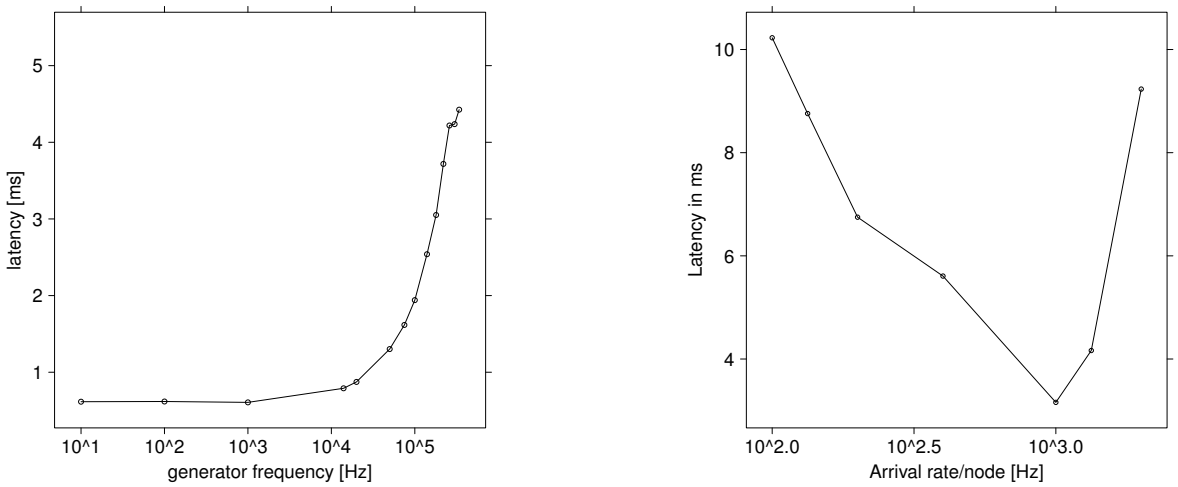

Fig. 7. Latency for the SLA and the COI monitoring benchmarks.

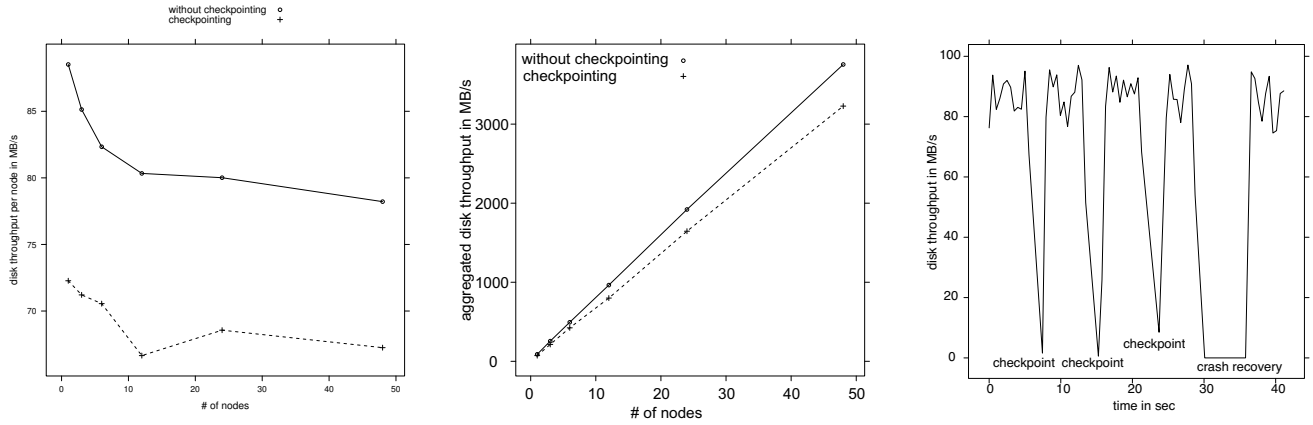

Fig. 8. Influence of checkpointing the state in the word frequency count: (a) per-node throughput, (b) total throughput, and (c) node throughput behavior in a run with a failure arount time 30.

named Percolator [16]. In addition to not focusing on low-latency applications, Percolator also considers a different programming model (based on observers and callback functions to update the results) and, therefore, is not MapReduce-user friendly. Among batch systems, Dryad [10] is also related to our system. Jobs in Dryad are expressed as an acyclic graph of operators. Dryad tries to optimize communication between operators (e.g., using shared memory or direct network connections).

When comparing MapReduce and databases, Stonebraker et al. [19] point out MapReduce to be suitable as a tool for the transformation of huge amounts of data, storing results in a database. They show that if a large volume of data is to be analyzed only once, the cost of inserting this data into a database is too high, and MapReduce (using the Hadoop implementation) can perform considerably better. They also point out that in comparison to parallel databases, MapReduce is easy to install and considerably cheaper, as there are no open-source, well-supported parallel database systems (although there are good open-source options for a single-node database). Further, MapReduce is good for semi-structured data (which requires too many rows in a conventional database) and for complex analytics, where multiple processing stages are necessary. However, once the data is in the database, MapReduce performs considerably worse than parallel database systems. To illustrate such cases, they present benchmarks were MapReduce performs up to $36\times$ slower than a commercial parallel database solution.

Some of the arguments against MapReduce can be used in favor of ESP systems. For example, ESP systems do not store intermediate results in a filesystem and apply more sophisticated fault tolerance strategies that improve performance. At the same time, some arguments in favor of MapReduce also apply for ESP. For example, previously loading the data is not needed. In addition, many ESP systems do not require pre-programmed data schemas, making ESP systems also suitable

to process semi-structured data for Extract-Transform-Load (ETF) tasks. Notwithstanding, neither parallel databases nor MapReduce can support the responsiveness required by some modern data-processing applications (such as some of the application discussed in Section II).

Finally, ESP is currently a very active research area and many systems have been developed, Borealis ([20]) being the best known among them. Although Borealis addresses scalability by optimizing the placement of operators, balancing load, and applying sophisticated techniques for load shedding, it does not address the problem of parallelizing a single operation among a large number of nodes, which is one of our major goals. GSDM ([11]) addresses operations that are partitionable, like in the case of MapReduce, but they do not consider operators that keep state (their operator processes whole windows at a time, similar to our jumping windows) and no support is provided for fault tolerance. Finally, StreamMine ([3], [4]) addresses low latency, fault tolerance, and parallelization of stateful operators, but the speculative approach does not scale horizontally (i.e., among nodes within a cluster).

VI. Conclusion

In this work, we have presented a system that combines responsiveness of ESP systems with the scalability of the MapReduce programming model. We presented a programming model that enables the MapReduce programmer who needs his application to support quick reaction to incoming data to quickly migrate to an ESP system. At the same time, our programming model offers all the functionality ESP programmers need while enabling a MapReduce-like scalability. Many common ESP operators can be mapped to be MapReduce model (e.g., joins, pattern discovery and match, frequency estimation).

One key aspect of StreamMapReduce is the introduction of stateful reducers. This enables us to overcome the strict phasing of MapReduce: reducers can use this state to store intermediate results when reducing partial value lists. The reducer state introduces however a new challenge: we need to recover this state in case of a node or process crash. We use a simple combination of in-memory logging in the upstream node and checkpoints to enable fault tolerance. Combined with a virtual-synchrony-like approach to determine the processing order of events, our implementation both provides precise recovery (i.e., states after recovery are always consistent with previous ones) and has a low performance impact.

Our evaluation shows that ESP operators using the MapReduce approach can scale well and that some batch jobs (like Word Frequency Count, used to benchmark MapReduce implementations) achieve even higher throughputs in our system.

References

[1] Hadoop. http://hadoop.apache.org/, January 2010.

[2] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, 1987.

[3] A. Brito, C. Fetzer, and P. Felber. Minimizing latency in fault-tolerant distributed stream processing systems. In *ICDCS '09: Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, Washington, DC, USA, 2009. IEEE Computer Society.

[4] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative out-of-order event processing with software transaction memory. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 265–275, New York, NY, USA, 2008. ACM.

[5] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[6] C. Cortes, D. Pregibon, and C. Volinsky. Communities of interest. *Intell. Data Anal.*, 6(3):211–219, 2002.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[9] L. Golab and M. T. Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5—-14, 2003.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, New York, NY, USA, 2007. ACM.

[11] M. Ivanova and T. Risch. Customizable parallel execution of scientific stream queries. In *Very Large Data Bases*, pages 157—-168. VLDB Endowment, 2005.

[12] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 51–62, New York, NY, USA, 2010. ACM.

[13] D. C. Luckham. *The Power of Events:An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[14] A. Martin, A. Brito, and C. Fetzer. Active replication at (almost) no cost. In *SRDS '11: Proceedings of the 2011 30th IEEE International Symposium on Reliable Distributed Systems*, Washington, DC, USA, 2011. IEEE Computer Society.

[15] A. Martin, T. Knauth, S. Creutz, D. B. de Brum, S. Weigert, A. Brito, and C. Fetzer. Low-overhead fault tolerance for high-throughput data processing systems. In *Proceedings of the 2011 31st IEEE International Conference on Distributed Computing Systems*, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[16] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2010. USENIX Association.

[17] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22:299–319, 1990.

[18] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throughput stream programming in java. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 211–228. ACM Press, New York, NY, October 2007.

[19] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[20] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: efficient load shedding techniques for distributed stream processing. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[21] F. Winklmeier. The ATLAS High Level Trigger infrastructure, performance and future developments. In *Real Time Conference, 2009. RT '09. 16th IEEE-NPSS*, pages 183–188, 2009.

[22] L. Woods, J. Teubner, and G. Alonso. Real-Time Pattern Matching with FPGAs. In *International Symposium on Data Engineering*, Hannover, 2011.