Distributed Event Stream Processing with Non-deterministic Finite Automata*

Lars Brenna
University of Tromsø
larsb@cs.uit.no

Johannes Gehrke
Cornell University
johannes@cs.cornell.edu

Dag Johansen
University of Tromsø
dag@cs.uit.no

Mingsheng Hong
Cornell University
mshong@cs.cornell.edu

ABSTRACT

Efficient matching of incoming events to persistent queries is fundamental to event pattern matching, complex event processing, and publish/subscribe systems. Recent processing engines based on non-deterministic finite automata (NFAs) have demonstrated scalability in the number of queries that can be efficiently executed on a single machine. However, existing NFA based systems are limited to processing events on a single machine. Consequently, their event processing capacity cannot be increased by adding more machines.

In this paper, we present an experimental evaluation of different methods for distributing an event processing system that is based on NFAs across multiple machines in a cluster. Our results show that careful input stream partitioning gives close to linear performance scaleup for CPU bound workloads.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems—*Query Processing*

General Terms

Experimentation, design, performance

Keywords

Publish-subscribe, Continuous queries, NFA, Event streams

1. INTRODUCTION

Large-scale applications, such as e-commerce systems, search engines, and stock exchanges create huge amounts of data which must be analyzed with tight latency bounds. To match these latency bounds, a powerful new system paradigm has emerged: Complex Event Processing (CEP) systems that match incoming events continuously against long-running queries that are registered with the system. One important processing model for CEP are non-deterministic finite state automata that have been used in systems such as Cayuga and SASE+ [2, 11]. These systems have shown that the resulting query processing system can process complex event pattern queries, but at the same time is highly scalable both with the number of events and the number of queries. Both Cayuga and SASE+ have shown single-node implementations that can deliver an event throughput of tens of thousands of events per second.

However, in many applications the rate of incoming events can be too high to be processed by a single node. For instance, the NASDAQ stock exchange matching engine must routinely handle over 125,000 messages per second [21]. Matching thousands of complex queries at this rate is beyond the processing capacity of a single machine with existing hardware technology. If the throughput of the CEP system is lower than the rate of incoming events, latency increases and the system will eventually run out of buffer space. To improve throughput, the CEP must be *distributed* across several machines.

Parallelization and distribution of NFA-based processing engines have received little attention in earlier literature. Tanenbaum and van Steen [27] discuss how NFAs can be decomposed into smaller NFAs that communicate by sending intermediate events between each other. This feature is called resubscription in Cayuga [11]. Resubscription leads to an architecture where components of an NFA can run on separate nodes in a cluster. However, they conclude (without an experimental evaluation) that achieving both expressiveness and scalability is not feasible.

An approach leveraging programmable hardware is used to handle extreme event rates in Gigascope [10, 18]. Query components are installed on the network device for two purposes. One is to reduce the amount of events that reach the higher-level application code. The second is to partition the input stream so it can be processed in parallel using multiple machines. However, in a typical shared, virtualized environment, we can not assume the availability of programmable network devices. Thus, solutions leveraging hardware are not an option.

In this paper, we perform an experimental evaluation of several practical approaches to distributing NFAs. We perform our experiments in the distributed event pattern matching system *Johka*,[1] which is based on two open-source projects: the Cayuga system for NFA-based event processing, and the multicast substrate Spread for communication [5].

The rest of the paper is organized as follows. Section 2 gives some background of NFA-based event processing systems, using Cayuga as an example for our discussion. Section 3 discusses pos-

*This work is supported in part by the Research Council of Norway through the National Center for Research-based Innovation.

DEBS'09, July 6-9, Nashville, TN, USA.

[1] Johka means *big river* in the Northern Sámi language.

sible strategies for distributed stream processing and demonstrates how NFA workloads can be parallelized and distributed. Section 4 describes the implementation of Johka, our prototype system. Section 5 contains an experimental analysis of our prototype. Section 6 discusses related work, and we conclude in Section 7.

2. BACKGROUND: THE CAYUGA SYSTEM

This section provides a brief overview of the Cayuga CEP system. Most of this discussion follows previous work in [12].

2.1 The Cayuga Data and Query Models

Cayuga allows users to express event patterns in an SQL-like query language called the Cayuga Event Language (CEL) [12]. It is based on the Cayuga data model and query algebra, which are explained in detail in [11]. Several unique features of the data model lead to important design decisions in Cayuga. Like a traditional relational database system, Cayuga treats data as relational tuples, referred to as events. However, Cayuga is designed to monitor streams of events, not static relations. Thus, rather than sets of tuples, the Cayuga data model consists of temporally ordered *sequences* of tuples, referred to as event streams. Each event stream has a fixed relational schema. Each event in the stream has two timestamps, the start timestamp, denoted as t_0, and the end timestamp, denoted as t_1. Together they represent a duration interval, defined by $t_1 - t_0$. Events are serialized in order of t_1; for this reason, t_1 is also referred to as the "detection time" of an event. Because of the semantic issues discussed in White et al. [28], Cayuga considers events with the same detection time to be simultaneous, and guarantees to produce the same result regardless of the order of processing these simultaneous events. This guarantee is realized through *epoch-based processing* in Cayuga.

The Cayuga algebra includes the unary operators of relational algebra, such as selection, projection, and renaming, as these can all be processed in a single epoch. It also includes union. However, for two reasons it does not include Cartesian product or window join, supported by full-fledged data stream processing systems. First, they are difficult to implement efficiently in a stream setting with a large number of queries. Second, they are less useful in their natural form than more restricted forms of join. Cayuga algebra instead introduces two restricted forms of join operators to permit temporal correlation of events.

The first operator is the sequencing operator $;_\theta$. This operator is a forward-looking join that combines a tuple with the "next" tuple in the data stream that satisfies the *filter* predicate θ. For example, if S_1 and S_2 are streams of stock quotes including a "name" attribute, then $S_1;_{S_1.\texttt{name}=S_2.\texttt{name}}\ S_2$ produces a stream of pairs, each pair comprising a stock quote from S_1 and the next quote for the same stock from S_2. Here "next" is determined according to the first epoch that contains a satisfying tuple. Tuples from S_2 that overlap the S_1 tuple are not considered; however, there can be multiple "next" tuples if they have simultaneous detection times.

To get more complex joins, Cayuga has the iteration operator $\mu_{\mathfrak{F},\theta}$, where $\mathfrak{F}$ is some composition of unary operators like selection or renaming. This operator is an iterated version of sequencing, where $\mathfrak{F}$ is used to define the duration of the iteration. This operator allows users to express a large class of forward-looking joins.

2.2 Automaton Model

Demers et al. [11] showed that any left-associated Cayuga algebra expression can be implemented by a variant of a nondeterministic finite state automaton, referred to as a *Cayuga automaton*. Non-left-associated expressions can be broken up into a set of left-associated ones, and will therefore be implemented by a set of corresponding Cayuga automata. Since CEL is based on Cayuga algebra, these results are applicable to CEL queries as well. We now describe how to process CEL queries with Cayuga automata.

Cayuga automata generalize on traditional NFAs in two ways: (1) instead of a finite input alphabet they read arbitrary relational streams, with state transitions controlled using predicates; and (2) they can store data from the input stream, allowing selection predicates to compare incoming events to previously encountered events.

Each automaton state is assigned a fixed relational schema, as well as an input stream. All the out-going edges of a state read that input stream. Each edge, say between states P and Q, is labeled by a pair $\langle \theta, f \rangle$, where θ is a predicate over $\mathrm{schema}(P) \times \mathrm{schema}(S)$; and f, the *schema map*, is a partial function taking $\mathrm{schema}(P) \times \mathrm{schema}(S)$ into $\mathrm{schema}(Q)$. The Cayuga automata operate as follows. Suppose an automaton instance is in state P with stored data x (note x conforms to $\mathrm{schema}(P)$). Let an event e arrive on stream S such that $\theta(x, e)$ is satisfied. Then the machine non-deterministically transitions to state Q, and the stored data becomes $f(x, e)$.

Predicates in CEL formulations are translated into automaton edge predicates in an obvious way. In particular, attribute decorators in CEL are translated into prefixes $e.$ or $Q.$ for a given automaton edge predicate, depending on whether the attribute comes from the schema of the current event read by that edge, denoted as e, or from the schema of the automaton state, denoted as Q, from which the edge originates.

Predicates in Cayuga automata are associated with edges. However, since there is always one filter edge for each state (except for start and end states), we can associate predicates on filter edges with automaton states without ambiguity. Similarly, since there is at most one rebind edge for each state, associating rebind edge predicates with automaton states is also not ambiguous.

Note that the predicate on a filter edge is the *negation* of the corresponding filter predicate in CEL formulation. In the following text, to avoid ambiguity, we avoid using the term filter edge predicate. To be consistent with the notion of a filter predicate in CEL formulations, in the context of automaton edge predicates, we use the term *filter predicate associated with state* Q to refer to the negation of the predicate of the filter edge associated with Q. For example, in Figure 1, the filter predicate associated with state Q_1 is $Q_1.\mathrm{N} = e.\mathrm{Name}$. Recall that we refer to the predicate on a rebind (resp. forward) edge as its rebind (resp. forward) predicate.

Any Cayuga automaton maintains the following invariants on its edge predicates:

- For any automaton instance I under state P, if the current event together with I satisfy the predicate of a forward edge from state P to Q, then they must together satisfy the filter predicate associated with state P.
- For any automaton instance I under state P, if the current event together with I satisfy the predicate of a forward edge from state P to Q, then they must together satisfy the rebind predicate associated with state P, if there is one.
- For any automaton instance I under state P, if the current event together with I satisfy the rebind predicate associated with state P, then they must together satisfy the filter predicate associated with state P.

A consequence of these invariants is that for any automaton instance I under state P, if the current event together with I do not satisfy the filter predicate associated with state P, then none of the

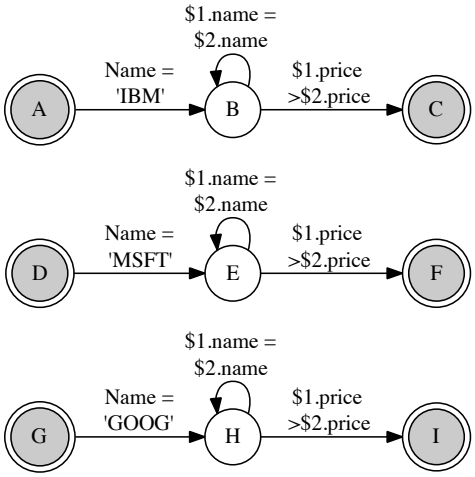

Figure 1: Three query automata looking for stock events on IBM, MSFT, and GOOG.

predicates on the rebind or forward edges associated with state P will be satisfied. Therefore, the instance I must traverse the filter edge of P and is unmodified (due to the identity schema map of the filter edge). In this case, we say instance I is *not affected* by the current event. Otherwise, if the current event together with I satisfy the filter predicate of P, we say I is *affected* by the current event.

These invariants can be realized in the implementation by predicate conjunctions. For example, the first invariant could be realized by attaching the filter predicate of state P as a conjunct to the predicate of each forward edge leaving P, and to the rebind predicate associated with P, if there is one. With an understanding of these invariants, to simplify the presentation, in the automaton figures we usually do not duplicate filter predicates on forward or rebind predicates.

2.3 Query Compilation

CEL queries are compiled to an intermediate XML format that represents queries as state machines. The XML-formatted queries are then loaded directly into the query engine data structures. Each Cayuga query is represented as an automaton extending the classical non-deterministic finite automaton [17]. Every predicate is mapped to an edge, and each new event can affect the state of the automaton. An edge is traversed if an incoming event satisfies the corresponding predicate. If no edges are traversed, the event is dropped. This mechanism implements selection. When an event traverses an edge, its values are evaluated by an identity function corresponding to the schema of the destination state. The output of the identity function is stored at the destination state as an instance. This mechanism allows Cayuga to generate witness events that contain data from all the events that caused the query automaton to reach its end state. Figure 1 illustrates three NFA representations of queries for two consecutively increasing stock updates.

By exploiting the relationship of the query algebra to the automata-based query execution, and commonality among queries, Cayuga can efficiently evaluate a large number of concurrent event queries. First, Cayuga achieves sharing of both computation and storage by merging all queries into a single NFA. Secondly, automaton edge predicates that are given a static parameter such as {name = "IBM"} can be managed efficiently by indices in a way similar to the techniques for processing multiple selection operators [14]. In this case, Cayuga creates an index where the $name$ value is the key identifying all predicates that must be evaluated for an event containing that value. This can be done when the query is inserted. Predicates such as {$1.name = $2.name}, do not have any static parameters to index, and must dynamically add a new index entry when an event arrives containing a new value for $name$. The separation gives two classes of queries; those with dynamic and those with static predicate indices. Dynamic predicate parameters are useful when the value space of the parameter is large or unknown. If it is known, a dynamic query can be instantiated into one static query per parameter in the value space.

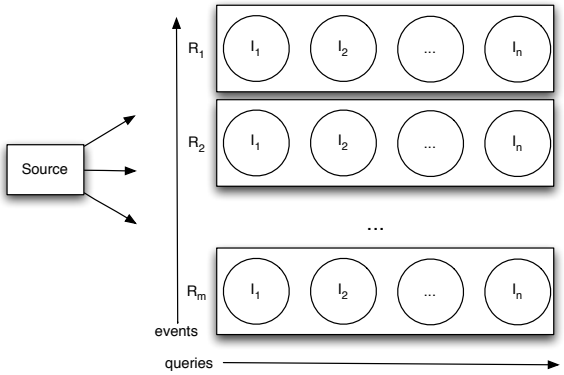

Figure 2: The row/column approach to query processing.

2.4 Discussion

We conjecture that there are at least two categories of techniques that can be applied to scale NFA-based event processing in a distributed system. Perhaps the most intuitive approach is to divide the queries in Figure 1 so that all queries regarding IBM stocks run on one machine, and all MSFT queries on another. Second, as described in [27], complex NFA's designed to run on a single computer can be decomposed into separate processes running in a distributed system. The next section will propose different techniques within these two categories, and discuss the challenges for each of them.

3. DISTRIBUTING CAYUGA

Cayuga implements several techniques to reduce the load of event processing on a single machine. We shall now discuss techniques from distributed computing that can be adapted to scale up the processing capacity of Cayuga on a cluster of machines. We divide our techniques into two categories; row/column and pipelining. Finally, we discuss combinations of these techniques.

3.1 Row/Column Scaling

A technique to scale query processing in stateless publish/subscribe engines is to organize the queries in a $n \times m$ matrix, as shown in Figure 2. The queries are divided equally among the machines in a row $R_i = \{1, 2, \ldots, n\}$. This row is then replicated m times, forming a row/column matrix. An incoming event is dispatched to a row i chosen, for instance, in a round-robin fashion, and replicated to each machine in that row. This guarantees that every event is matched against every query. A stateless query model requires no communication between queries running on separate machines, and event dispatching does not need to consider which machine runs a given query. The advantage of this technique is that it allows event processing to scale nicely by the number of available machines, and

more machines can easily be added to increase processing capacity.

Unfortunately, basic row/column scaling cannot be directly applied to stateful event pattern matching. If the automata in a given row detect an event that marks the beginning of a pattern, then every event that can finish that pattern must be delivered to and processed by that particular row.[2] If the event dispatcher selects rows in a random or round-robin fashion, the entire cluster might need to synchronize state between each incoming event to ensure that no query patterns go undetected. This is not a scalable solution.

We conjecture that a row/column matrix can be used to scale stateful event pattern matching system if related events are always processed by the same queries. There are at least two ways to achieve this. One is to partition the original input stream into substreams of related events within the original stream. One or several rows are then assigned to receive all events in a specific substream. It is the selectivity of the query workload that define such substreams. Queries that select different events from the input stream thus define one substream each. In stock applications, the stock ticker name partitions the input stream if all queries depend on individual stock tickers. Query-aware stream partitioning is also applied in Gigascope [18].

Once a partitioning into substreams is determined, it can be expressed as Cayuga queries that select events in the original stream to publish on separate sub-streams. In their most basic form, the partitioning queries consist of a selection predicate and a renaming operator that changes the stream name of the event to a name identifying its substream. We can then install these queries on a designated machine, which will function as an event dispatcher for the matrix. Note that the most basic partitioning queries can also be expressed as hash functions. Splitting the stream and processing it in parallel require that output from the partial streams are merged. Adding split and merge functionality to partition the stream is equivalent to Box Splitting in Aurora* [9].

Our second technique to scale using a row/column matrix is partitioning of the query workload, similar to query plan partitioning in Borealis [7]. In this technique, a full replica of the input stream is delivered to each row. To make each row process different stream partitions, we partition the query workload across the rows. The effect is that the machines in each row will receive but disregard most events. If the query workload consists of a set of queries with static predicate parameters, then the queries can be distributed across rows as they are. However, queries with dynamic predicate parameters will need an extra selection predicate to limit which events it will process. In a stock application, we can add a selection predicate that selects events based on the stock ticker name.

Both techniques have drawbacks. Query set partitioning does not reduce the rate of events that each machine receives, so each machine must evaluate many more events than are actually relevant for its queries. Stream partitioning leaves many redundant queries at each machine, since their host machine will never receive events that are related to them. However, stream and query set partitioning can be used together so that each row only runs queries for events that will actually arrive. A drawback we share with other systems is that stream partitioning becomes a bottleneck, as the total throughput of the system cannot be higher than that of the dispatcher.

One approach to alleviate the bottleneck is to use multiple dispatchers. In this case, all dispatchers receive the same input stream in order to share the load of event dispatching. Another is to remove the use of dispatchers altogether. In the latter case, the full event stream will be delivered to all rows and the only optimization we have discussed so far for this situation is query set partitioning.

[2]We define *related events* as a sequence of events that may be part of the same query pattern.

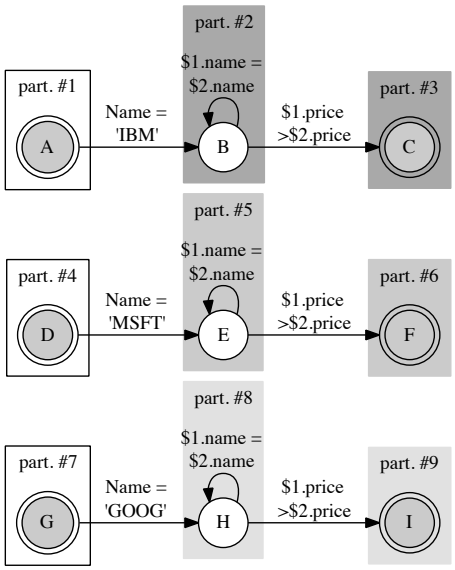

Figure 3: An NFA partitioned into nine parts that can be configured to run as pipeline steps on separate machines.

Our conjecture is that row/column scaling can improve throughput. The two contributing factors is that both the event rate per NFA, and the number of predicates that must be evaluated per event per machine are lowered linearly as more machines are added.

3.2 Pipelining

Row/column partitioning increases the number of queries that can be processed at a given throughput rate. It equally increases throughput for a given number of queries by dividing queries and events across a cluster. However, it is still possible that the processing cost of even a single query can be too high for a single machine. In this case, the query itself must be split up and distributed to scale up throughput. This can be the case for query automata with many states, and consequently many predicates, or with costly predicates. Boxes and arrows-based systems such as Aurora* solve this scenario by distributing boxes across multiple machines, and streaming the output from one operator into the input of another [9].

In contrast, Cayuga does not have any logically separate operators or components that can be easily distributed. The computational units in Cayuga are NFA edge predicates, belonging to a given NFA state. Those states are in turn connected by forward edges. The Cayuga query language allows to split any nested query into smaller queries, and forward output from one query to another via a feature called resubscription. On an automaton level, this means every automaton with at least one forward edge can be split into smaller automata that run on separate machines, connected and synchronized to run as a *pipeline*. Figure 3 shows how the automata from Figure 1 can be decomposed into nine pipeline steps.

It is important to note that the graph in Figure 3 is neither a dataflow graph nor a regular pipeline, but an instance flow graph. All states except the end states can have predicates, which are evaluated for every incoming event. Since an incoming event can cause state transitions in every state of an automaton, each state must receive every event. When an NFA is split at arbitrary forward edges without other changes to the queries, the input stream must be replicated and equally delivered to all NFA components. Additionally,

the output stream from the components #1, #4, and #7 must be delivered to components #2, #5, and #8, respectively. The throughput improvement from decomposing the NFA might then be reduced by the disadvantage that the rate of incoming events actually increases for some components.

Some queries can be decomposed and additionally rewritten to avoid this extra overhead. For example, if a query looks for ten consecutive stock events, it can be divided into two sub-queries where the first looks for five consecutive events and the second looks for two consecutive events coming from the first sub-query.

Our conjecture is that partitioning the query NFA and pipelining its partitions can improve overall throughput to reach that of the most demanding partition. For maximum throughput, an NFA should be split so that the most demanding state runs on a separate machine. If the load of a single state exceeds the load of the rest of an automaton, it hardly makes sense to partition the query into more than two parts even if the automaton has several more states.

3.3 Combined Techniques

It is possible to combine row/column scaling with pipelining. In this case, the rows in the row/column model are replaced by replicated pipelines. The dispatcher partitions the stream so each pipeline replica process one partition each.

Stream partitioning requires a dispatcher, which may become a bottleneck. A separate dispatcher can be avoided by moving the stream partitioning predicates to the processing machines. The technique replicates the query set to all rows, then modifies each replica by adding a predicate on the first forward edge. The added predicates correspond to those that would have been used in the dispatcher to partition the input stream. The effect is equivalent to query set partitioning, since each row must still receive the full input stream. For ease of reference, we name this technique *vertical NFA partitioning*.

4. IMPLEMENTATION

Cayuga consists of approximately 23,000 lines of C++ code, including its own copying garbage collector and query compiler. It runs on Windows, Linux, and OS X. The source code for the centralized version is freely available for download [8]. It was a design requirement for the distributed Johka system that it should be fully compatible with the centralized Cayuga system. Our goal was not to build a new query engine, but rather to evaluate if and how an existing centralized engine could be used in a distributed setting. Our approach has been to wrap the Cayuga processing core with a new communication layer. This layer has added an ability to receive events in batches to limit lock contention and to create larger messages for the network. We have also moved serialization costs from the core thread to the I/O threads. We observed that this was beneficial on multi-core systems when the process is CPU bound by the core thread. The original source code we downloaded has been thoroughly profiled and optimized for our use.

4.1 Internal Architecture

The Cayuga system is multi-threaded and event driven. It has an object-oriented design with clearly defined borders between event processing and I/O, allowing administrators to configure at startup which input and output components to use. The typical configuration includes one thread for the query engine, and separate event receiver and sender threads for I/O. Figure 4 shows this architecture, along with the necessary queues and the Cayuga heap where the events and instances are stored. The system has event receivers and senders for the file system and raw sockets. The event receiver will receive, deserialize and allocate events on the Cayuga heap (1),

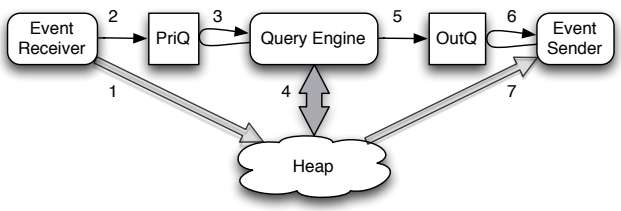

Figure 4: Cayuga System Architecture.

then insert a reference to them on a priority queue (2). The priority queue sorts the events by epochs, so when the query engine polls it (3), a reference to the event with the lowest epoch is returned. The query engine will then evaluate the event and update NFA state instances (4). References to output events are put on the output queue (5), allowing the event sender to fetch batches of event references (6). The event sender is responsible for serializing and deallocating the output events from the Cayuga heap (7).

Our first challenge was to optimize the queue semantics for high throughput under varying event rates. Originally, both queues had a non-blocking push-one, pop-one functionality. Before popping an event from a queue, the consumer had to check that the queue was non-empty by calling $peek$. Instead of blocking on pop when there were no events to consume, the consumer would sleep for a set period if $peek$ returned $NULL$. There are two drawbacks to this scheme. One is that push-one, pop-one can cause the producer and consumer of a queue to compete for the lock, which is expensive since pthread calls go to the kernel if the lock is contested. Typically, the threads would be processing events at different speeds so the queues would either grow or stay empty much of the time. Secondly, sleeping on empty queues works well for workloads where events arrive steadily so the queue rarely, if ever, empties. However, it can cause variable latency during bursty load in a cluster where different machines receive and process events at different rates. Variable latency between bursts makes it difficult to set the sleep period a-priori. One solution is to adjust sleep times dynamically during runtime [15].

To alleviate the push-one, pop-one issue, we chose to extend the queue APIs. Producers and consumers now push and pop batches of events, and they can block until new events are available. The priority queue function $peek()$ is given an additional parameter that allows the query engine to block on a conditional wait until data is available. If $peek(block = false)$ is called when the priority queue is empty, it returns $NULL$. The query engine thread can then perform garbage collection if necessary. Additionally, all output buffers are flushed before the engine thread blocks. Using a conditional wait to block for a batch of events removes the need to dynamically adjust sleep times.

Batched queue operations requires that each thread has a private queue of the same type as the shared queue. Events are retrieved by calling $Queue{\rightarrow}popQueue(privateQptr)$, which blocks on a condition variable. $Queue{\rightarrow}pushQueue(privateQptr)$ is called by the producer when it has put a set number of events on its private output queue. This function switches the two queue pointers, signals on the condition the consumer is blocking on, and returns a pointer to the empty queue that came from the consumer. Then, the consumer is unblocked and $popQueue$ returns a pointer to a full queue. A buffer is flushed with a single operation when it has reached a preset threshold. The gains of batching must be weighed against the added latency it takes to build a batch [15]. In Johka, buffer thresholds must be tuned to keep outgoing events from wait-

ing too long when the throughput is low, and maximize batch gains during high load.

4.2 Connecting Cayuga Engines Using Spread

We decided to use the Spread Toolkit for communication [5, 25]. Spread has an API that fits to the existing Cayuga architecture. Using a group communication system gives us an abstraction where we can map streams and substreams directly to group names. Our second challenge was to integrate Cayuga with Spread for maximum throughput under varying circumstances. We have added new I/O components for sending and receiving events using Spread. These components send batches of events in messages up to 100 KB for better performance under Spread.

An important task of the communication layer is to perform query-aware event partitioning. Our solution is to map event stream identifiers to Spread multicast groups. A designated Cayuga machine, named the *event dispatcher*, groups and dispatches incoming events to the processing engines. The dispatching workload partitions the event stream into substreams, and renames the event stream identifier of events according to their designated substream. Partitioning and stream identifier renaming correspond to the query partitions on the cluster. Events are then multicasted on the groups where the corresponding processing engines are listening.

We use a set of scripts to create configuration files that contain the mapping between Cayuga stream identifiers on the query level to multicast groups on the Spread level. When a Cayuga engine starts, it reads a configuration file pre-written especially for that machine. It subscribes to one or more input groups as specified there. Event stream replication is done by having several receivers subscribe to the same Spread group. During event stream partitioning, each Cayuga engine is given a set of output groups where its subscribers are listening. The event sender thread then uses an event multiplexer to map outgoing events to groups in the same way the configuration files direct which queries should be loaded where, effectively routing events to the correct engines.

4.3 Synchronization

The temporal semantics of event sequences in Cayuga requires that all simultaneous events detected at epoch t_i are processed before the engine can proceed to process events at t_{i+1}. During event partitioning, sequential stream partitions can be processed in parallel. Consequently, a condition defining correct behavior is broken when strictly ordered events can arrive out of order where those parallel stream partitions are merged. However, this broken condition can be masked as long as we merge the output from those parallel stream partitions correctly. We do this by making sure every engine has received all output for epoch t_i from its predecessors before it can proceed to epoch t_{i+1}. The challenge lies in making an efficient implementation that does not void the advantage given by the parallelization optimization.

To a certain extent, the problem is already solved by the buffer functionality present in Cayuga. If an event belonging to epoch t_{i+1} arrives before an event belonging to epoch t_i, while Cayuga is still processing at epoch t_{i-1}, correct ordering is guaranteed by the semantics of the priority queue. Similar problems are handled in window-based systems by using *heartbeats* to signal the end of epochs [19, 26]. The advantage of processing fixed time windows one by one is that even though events arrive out of order and with unpredictable timestamp gaps due to filtering, a heartbeat can flush blocked operators and input buffers such as our priority queue. However, if for some reason an event belonging to epoch t_{i-1} arrives while Cayuga is processing epoch t_i, the event cannot be processed and must be dropped.

Cayuga originally assumed a strictly ordered input stream. Therefore, it always consumed the next event as decided by the priority queue semantics, and updated its epoch counter to the epoch that event belonged to. That has been changed so that it can never increment its epoch counter by more than one step. This preserves the condition of causal ordering. However, in the case where events are filtered by up-stream engines, there might not be events for every epoch. This can cause Cayuga to block and wait forever as there is no way to know whether or not an epoch update will arrive. In Johka, we solve these synchronization issues by introducing a special punctuation event to signal the *end of an epoch*(EOE). These EOE events must be present in the stream when it arrives at the cluster, can not be filtered by queries, and must be sent to all output groups. The event receiver thread in each Cayuga engine must know how many publishers there are in its input groups, (i.e., how many EOE messages to expect per epoch). When all EOE events are received, Cayuga knows that all events for that epoch have been sent from their publishers, received and processed. It is now safe to move on to the next epoch, and the engine sends EOE events to its subscribers for the epoch it just completed. The $peek$ function of the priority queue enables the query engine to inspect which event is next in the queue without actually removing it from the queue. For efficiency, EOE events are not sent through the NFA engine.

Our scheme meets the critique by Li et al. [20], and solves an important distributed synchronization issue in Johka. Although the solution enables fault detection, there is currently no way to recover and proceed if an upstream publisher fails to forward EOE events.

4.4 I/O Optimizations

To evaluate our new design, and to investigate any potential for improvement, we performed several incremental performance tests. Table 1 compares the throughput in events/sec in five different setups. Experiments that include NFA processing, are measured using an event dispatcher workload for stock data.

Table 1: Throughput of different versions.

Spread	Orig.	Queues	Bypass	Parse	Final
1,554,122	25,514	38,023	996,086	96,279	94,075

For comparison, the first column is the maximum event rate we could achieve with Spread between two machines. We then add a third machine running Cayuga in the middle, forwarding from the sender to the receiver. In the second column, the original Cayuga distribution with support for Spread achieved a throughput of only 1.7% of Spread's maximum. We then profiled and optimized memory handling, and introduced new queues. The *Queues* column shows that throughput increased to 38,023 events/sec. New profiling revealed bottlenecks in event parsing and output event production in the query engine thread. On a single-core CPU, threading can not alleviate such bottlenecks. However, our cluster consists of multi-core machines. We subsequently adapted Cayuga to multi-core CPUs by offloading this load to the two I/O threads. For reference, the *Bypass* number shows throughput when we measure the overhead of receiving and forwarding incoming events from Spread. In this experiment, no events are parsed or processed. The input thread queues entire batches, which the engine thread passes directly to the output queue. The sender thread then immediately passes the batches to Spread. The throughput was now 70% of Spread's throughput between two machines. Adding a second sending member to the Spread segment may account for large portions of this 30% drop. For the *Parse* throughput, we add event parsing and pass all events through the queue system, only

bypassing the NFA itself. Throughput now drops to one tenth. NFA processing is re-introduced in the *Final* column without significant impact, giving a throughput of 94,075 events/sec.

The results show that I/O is the dominating cost of event processing on a single machine. Parsing and copying events is CPU intensive. The current implementation is an attempt to lower the impact of I/O on throughput. The remaining bottlenecks consist mainly of parsing events from the ASCII-encoded input stream into memory, and copying NFA instances to produce output events. In the rest of the paper, we will refer to the final, optimized version of Cayuga unless stated otherwise.

5. EXPERIMENTS

To evaluate the effects of our NFA distribution schemes, we have conducted several experiments. Our experimental platform consists of 13 machines, each with two Intel Xeon E5335 2 GHz Quad Core CPUs, i.e. eight CPU cores per machine, and 8 GB of main memory. The machines communicate over a 1Gbit fully switched network, and use NTP to synchronize clocks. All measurements are repeated at least ten times and all reported numbers are mean averages with a measured deviation of less than 2.5% without outlier elimination. Unless otherwise specified, our experiments are based on streams of ten million events with 1,000 stock symbols each.

Throughput on Spread improves when events can be batched into larger messages to reduce the overhead per event. We evaluated message sizes from 25 bytes to Spread's maximum size 100 kB. The highest event rates, corresponding to 400 Mb/s, was achieved with the maximum size. This batching will sometimes give a bursty network traffic pattern. To better tolerate such traffic, we have recompiled Spread with message buffers ten times larger than default. Because Johka does its own application level synchronization, messages are sent using Spread's FIFO by sender ordering semantic. All experiments assume a static group membership, and we do not consider failures. However, we do request reliable message delivery from Spread in case of packet loss due to network congestion.

To emulate an external event source, a special process running on a designated cluster machine reads an input file from disk, logs its start timestamp and sends events at a set rate to the event receivers. If a dispatcher is used, it is the only receiver. It will then perform the query-aware event forwarding to the event processing engines where the actual experiment workload runs. The event dispatching queries used in our experiments group stock events by their stock names, which ensures an equal distribution of stock symbols across the processing engines. A designated cluster machine merges the output streams and logs the time when all events have been received. Throughput is measured from when the first event is sent to Johka until the last output event has been received by the merging machine.

5.1 Single Machine Baseline

We will now evaluate the effect of workload characteristics on the throughput of Cayuga running on a single machine. We conjecture that varying NFA size and selectivity will affect throughput. The NFA size is varied by first experimenting with an increasing number of queries. Some of these queries are for events that are not present in the stream, thereby reducing the overall selectivity of the NFA. Second, the NFA size is varied by experimenting with queries for event sequences of increasing length.

In the first experiment, we query patterns within streams of stock exchange updates. The queries are for ten consecutively increasing stock price updates, and have static indexed predicates. As input to the experiment, we generated two event streams, S1 and S2, each containing 500,000 stock exchange updates. Stream S1 contains 1000 different stock symbols, while S2 contains 3000 different stock symbols. To ensure an even and predictable output flow, both S1 and S2 are generated so that stock prices are monotonically increasing. We feed these streams to Cayuga while varying the number of installed queries from one to 3,000 for S1, and from one to 6,000 for S2. The first 1000 queries on stream S1 are for stock symbols that exist within the stream, while the remaining 2000 are not in the stream. On stream S2, only the first 3000 queries are for symbols in the stream. Increasing the number of queries until every event triggers a query, allows us to observe how Cayuga performs when the stream contains events that are not sought for. This will show us how efficient the engine is with redundant queries.

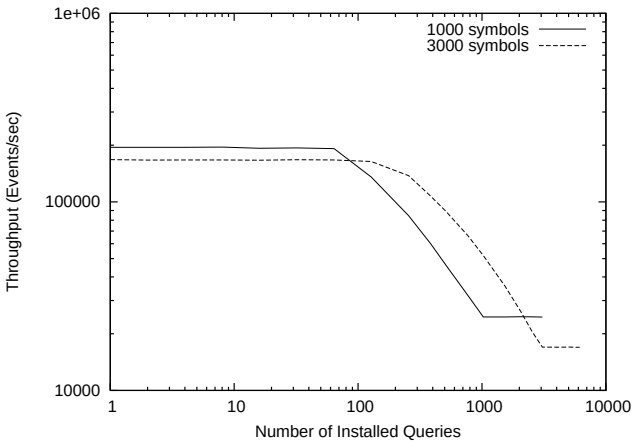

Figure 5: Event throughput vs number of predicates.

Adding more queries implies that the number of states in the NFA grows, and that more predicates must be evaluated for each event. Figure 5 shows how throughput decreases as we increase the number of queries. The graphs start in the upper left corner showing that Cayuga manages a steady throughput with one to 16 queries of approximately 167,000 events/sec on the large stream, and 195,000 events/sec on the smaller stream. Throughput starts declining after 16 queries, indicating that the bottleneck shifts from I/O to CPU when we add more queries. We increase the number of queries until each stock symbol is covered by a query, at 1,000 and 3,000, respectively. We observe that throughput decreases linearly to approximately 17,000 and 24,000 events/sec.

The declining curves indicate a correlation between event throughput and the number of installed queries. The curves both reach a bottom and flatten to a plateau, respectively at 1,000 and 3,000 queries. Adding more queries increases the number of states in the NFA, but not the number of predicates that must be evaluated for each event. Cayuga does not evaluate queries for symbols that are not in the stream. Thus we can conclude that NFA size and selectivity affect throughput. More precisely; throughput is directly affected by the number of predicates that must be evaluated for every event.

The skew between the graphs indicates the batch benefits of a stream with three times more events per epoch. To preserve concurrency properties, all state transitions within one epoch are not visible until the next epoch. Cayuga will then install all new instances as a batch. Thus the 3,000 stock stream triggers fewer NFA update operations per second, which allows a higher throughput per predicate than a stream with fewer events per epoch.

Each run in the second experiment uses a 1,000 symbol stream and a single query with dynamic predicates. The first run uses a

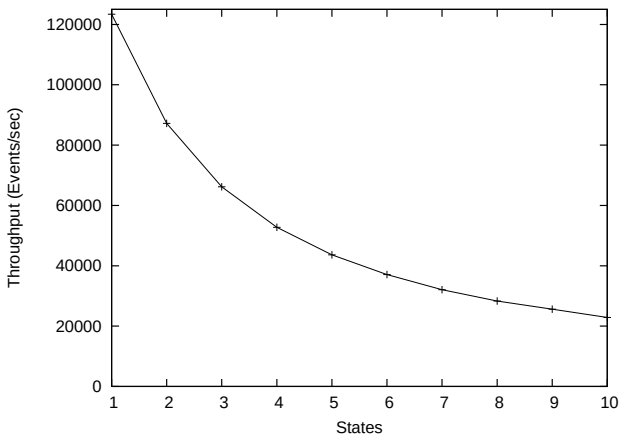

Figure 6: Event throughput vs sequence length.

query with one middle state that looks for two consecutively increasing stock price updates. This query is similar to the queries in Figure 1, except it does not have a static predicate on the initial forward edge. For each run, we add one more such middle state. This increases the NFA size by adding more states in a different way than by adding queries. Each event will lead to a number of state transitions that is equal to the sequence length, since every price update is an increment. Thus, doubling the sequence length of the installed query should double the number of state transitions per event, and also double the storage requirements for the NFA.

Figure 6 shows how throughput declines as the number of states increase. Cayuga manages a throughput of approximately 123,000 events/sec for a query with one middle state. The graph shows that throughput declines approximately 40% each time the NFA size doubles. Doubling the NFA size also means doubling the number of predicates that must be evaluated. These findings are consistent with the previous experiment, showing that throughput is directly affected by the number of predicates that must be evaluated for every event.

5.2 Event Dispatcher Capacity

We have measured the capacity of Spread between two machines at 400 Mbit/s., which is near the Gigabit capacity of our network. However, using an event dispatcher adds overhead and may become a bottleneck. To alleviate the potential bottleneck of a single dispatcher, we evaluate the effect of using up to four dispatchers in parallel. We employ query set partitioning to divide the partitioning workload among the dispatchers.

Surprisingly, this dispatcher does not scale. Throughput increases little, from 94,000 to 96,300 events/sec as we increase from one to four dispatchers. Adding more dispatchers means each dispatcher produces less output, but does not change the rate of incoming events to each dispatcher. Offloading production of output events to the sender thread means that the cost of filtering out an event is not much lower than the cost of producing an output event. Thus, a dispatcher does not gain from a more selective workload, and the workload per dispatcher is not reduced as we add more machines.

The observed throughput of the dispatcher is still just one-tenth of the throughput of the batch forwarder shown as *Bypass* in Table 1. Since Spread is based on a token-ring protocol, network bandwidth is divided equally between all participating senders in a Spread segment. Thus, the end-to-end throughput will decrease with the number of Spread clients that are introduced between the end-points. This effect is illustrated in Table 1, where the *Bypass* using three machines gets only 2/3 of the throughput achieved by two machines in the *Spread* experiment. Thus, adding a machine to run as a dispatcher could reduce the potential end-to-end throughput by as much as 1/3. Furthermore, the dispatcher is a Cayuga engine that splits the incoming stream into substreams based on arbitrarily complex rules expressed in CEL. However, if the stream partitioning rules are simple enough to be expressed as hash functions then using Cayuga as a dispatcher can become an unnecessary bottleneck. For comparison, we developed a stateless dispatcher that treats all events as raw byte arrays. A hash function inspects the first field in an event, in our case a stock ticker name, hashing it to a Spread group name. The event can then be sent without extra copies or serialization.

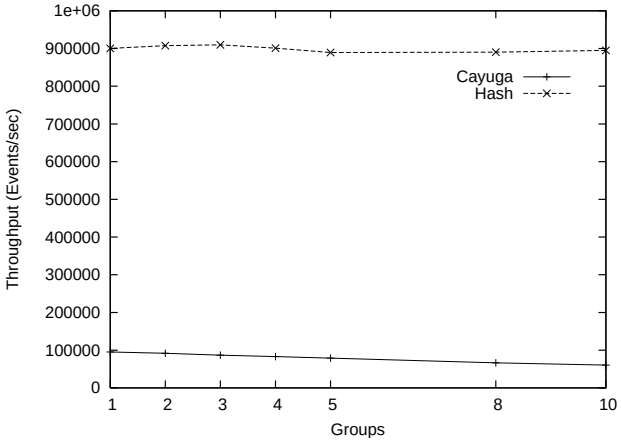

Figure 7: Throughput vs. the number of groups.

The dispatcher implements stream partitioning by publishing events belonging to separate substreams on separate Spread groups. Figure 7 shows an experiment where we measure the effect of increasing the number of Spread groups. In this experiment, there is only one receiving machine, receiving all events from all groups. We introduce the stateless, hash-function based dispatcher here. We observe that its throughput is approximately 900,000 events/sec, almost one magnitude higher than that of the Cayuga-based dispatcher. Throughput declines slowly when we add more groups in both cases.

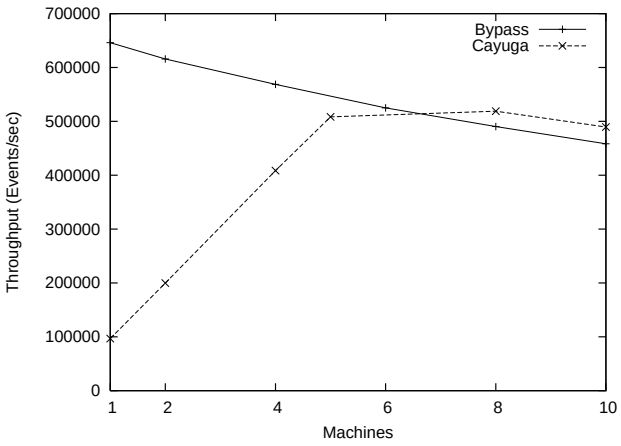

Figure 8: Throughput without Cayuga.

We then measure the impact Spread has on throughput as we add more machines. Figure 8 shows the results from the stateless dispatcher sending to up to ten machines. First, each machine runs Cayuga with the dispatcher workload sending to a single group. Second, each machine runs the *Bypass* version. We observe that throughput for Cayuga scales linearly up to five machines, which process 500,000 events/sec together. However, throughput does not increase above that. Throughput for the *Bypass* version declines when more machines are added. The graph shows that Johka will benefit from input stream partitioning, but that throughput may at some point top out when Spread can no longer deliver enough bandwidth. We can already conclude that Spread will become a bottleneck in scaling Johka if the distributed workloads become I/O bound.

5.3 Row/Column Scaling

Guided by the performance analysis of a single Cayuga engine in Section 5.1, we conjecture that we can improve throughput by reducing the amount of predicates that must be evaluated per machine. We will do this as outlined in Section 3 by partitioning the input stream and query set across a number of machines.

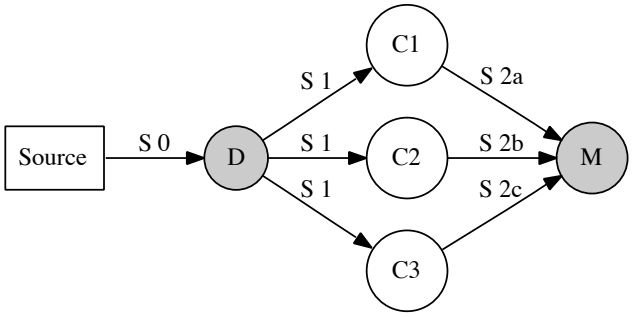

Figure 9: Example cluster layout for row/column scaling.

Figure 9 shows a simple cluster layout for row/column scaling. The machines C1, C2 and C3 will process the stream S1 in parallel. The query set is partitioned equally across the three machines, and in this example each machine will process every event. We implement row scaling by partitioning the input stream and replicating the query set. This reduces the number of predicates that must be evaluated by reducing the number of symbols in the stream partition each machine receives. We implement column scaling by replicating the input stream to all processing engines, but partitioning the query set. This reduces the size of the NFA on each processing engine, and thus the number of predicates that must be evaluated for each incoming event. Note that query set partitioning is only possible when the workload contains multiple queries. We achieve full row/column scaling by combining the two such that each machine has a partition of the query set, and only receives events that will be matched by its predicates. To show the generality of our approach, we will evaluate it with two different workloads.

The first query set consists of 1,000 queries ($N = 1,000$), one for each stock ticker, where each looks for ten *sequentially increasing* stock prices ($L = 10$). We generate an input stream for this workload where the stock prices always increase. Thus, there will be one output event for every event except the nine last on each stock. The query set consists of multiple queries, so we can apply our techniques to it one by one and then together.

Figure 10 shows the results of the different configurations. We observe that query set partitioning increases sublinearly from 11,000 events/sec to 46,500 events/sec. Input stream partitioning has close

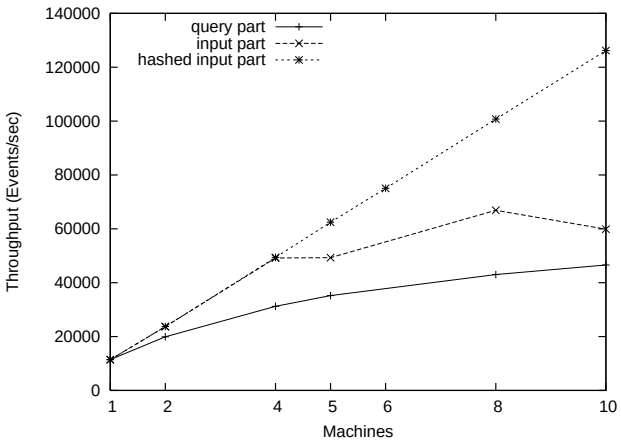

Figure 10: Row/column scaling of 1,000 static queries.

to linear performance gains until it catches up with the dispatch rate using four machines. Query set partitioning has lower effect than stream partitioning. This is consistent with the observation in Section 5.1. While they both reduce the number of predicates that must be evaluated, combining them cannot reduce the number further. Thus, we omit combined results from the graph.

Referring to Figure 8, we can assume that Johka is now bound by the dispatchers capacity. The final plot in Figure 10 shows that the stateless input stream partitioner scales this workload further, since its throughput is larger than the processing capacity of ten machines.

The second query set is a single complex query used in technical analysis of stock trade data, where event stream pattern matching can be applied to recognize trading opportunities. The query looks for patterns where stock movements appear as an M-shaped curve on the chart. For this experiment, we only evaluate input stream partitioning. It has dynamic filter predicates, i.e. $event.name = node.name$. In contrast to the first query set, where each query looked for a pattern on a specific stock, this query looks for any stock with this pattern. However, since the previous query set covered the entire set of stock symbols, the effect is the same. To ensure that this query has any output, we carefully generate an input stream that contains M-shape patterns.

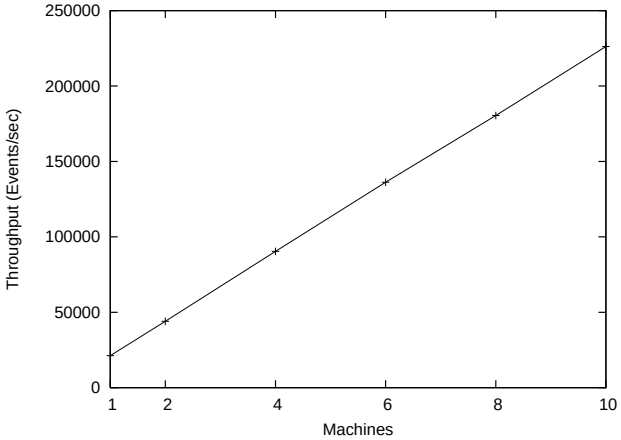

Figure 11: Row/colum scaling of the M-Shape query.

Figure 11 shows that this workload has a lower throughput since it is more demanding per machine than the previous, but it appears to scale linearly on our cluster. We did not have enough machines to observe this workload meet the capacity of the dispatcher, but its scaling properties are the same as the previous workload.

5.4 Pipelining

The second approach to reduce the number of predicates that must be evaluated is to split the automata into several parts. Each of these parts can then run on separate machines, organized as a pipeline. All pipeline steps must still receive the original input stream in addition to the output stream from their predecessor. However, we expect that processing sub-automata in a pipeline can increase throughput.

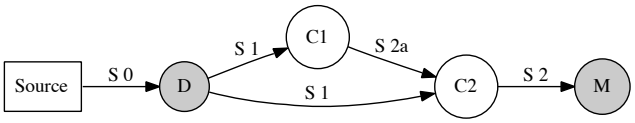

Figure 12: Two step pipeline.

Our first query is looking for three consecutively increasing stock prices. The query automaton is similar to the automata shown in Figure 1, but has one more middle state with a filter edge, in total four states. It has dynamic predicates, and thus looks for any stock with this pattern. We will divide this query on the forward edge between the two middle states and run them as two steps in a pipeline. The resulting cluster layout is shown in Figure 12. Notice that machine $C2$ receives all events from D as well as the output stream $S2a$ from $C1$.

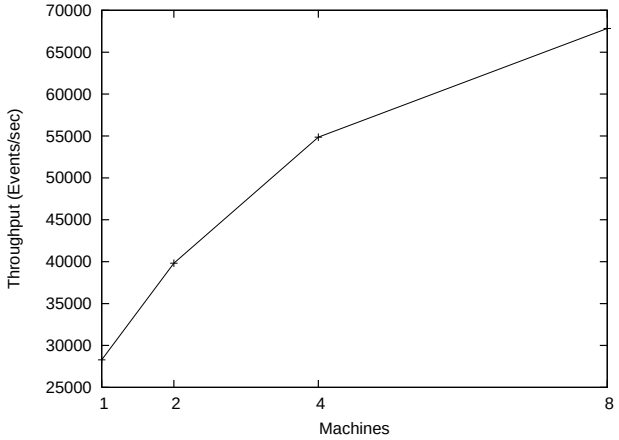

Figure 13: Pipelining eight states.

As two states cannot be pipelined across more than two machines, Figure 13 shows the effect of pipelining a query with eight states looking for nine consecutively increasing events. Throughput for the entire query on one machine is 28,300 events/sec, which increases 40% when it is pipelined across two machines. From two to four throughput increases approximately 35%, and approximately 25% from four to eight machines. Pipelining an NFA requires that a pipeline step receives the output stream from its predecessor as well as the original input stream. Thus, later steps in a pipeline will receive more events than the first step, explaining the decreasing speedup factor.

5.5 Combined Techniques

Row/column scaling can be combined with pipelining by replicating the query pipeline and partitioning the input stream between the pipelines. The combination lowers the amount of predicates that must be evaluated per pipeline step by partitioning the input stream.

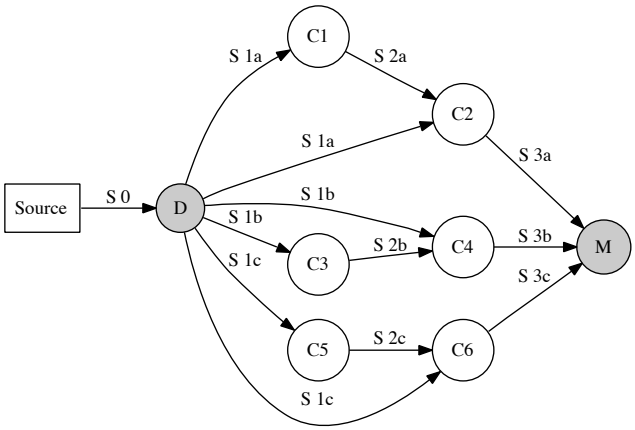

Figure 14: Combining Row/Column with Pipelining.

We will replicate the query pipeline of length three shown in Figure 12 to run on four and eight machines. Figure 14 shows an example cluster layout for three pipeline replicas a, b, and c for the six machines $C1$-$C6$. The event dispatcher D partitions the input stream $S0$ into the streams $S1a$, $S1b$ and $S1c$. These are replicated to both machines of each pipeline. The results are shown in Figure 15. The datapoints for one machine represent both pipeline steps running on a single machine, and the datapoints for two machines are for a single, un-replicated pipeline. The graph shows that four machines met the capacity of the Cayuga dispatcher at approximately 90,000 events/sec.

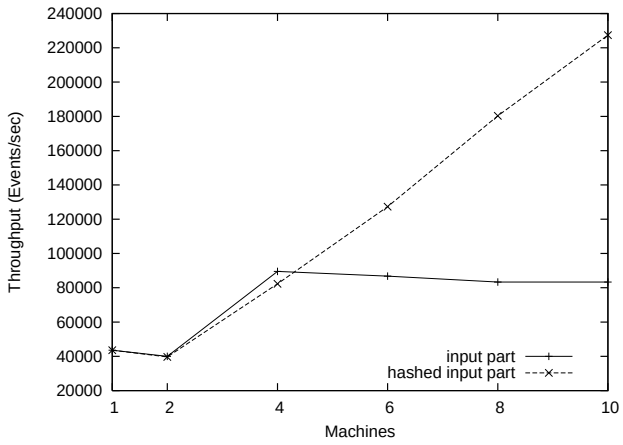

Figure 15: Throughput for Row/Column with Pipelines.

Next we use the dispatcher with hash-based input partitioning. We observe that throughput is no longer limited by the dispatcher, and scales close to linearly. Throughput for ten machines is more than twice than with the Cayuga dispatcher.

To avoid the bottleneck imposed by the dispatchers, we introduced vertical NFA partitioning as an alternative. In the next experiment, we replace the dispatcher with corresponding forward edge

predicates on each of the pipeline partitions. This causes every partition on each pipeline to receive the full stream, but only process their designated events.

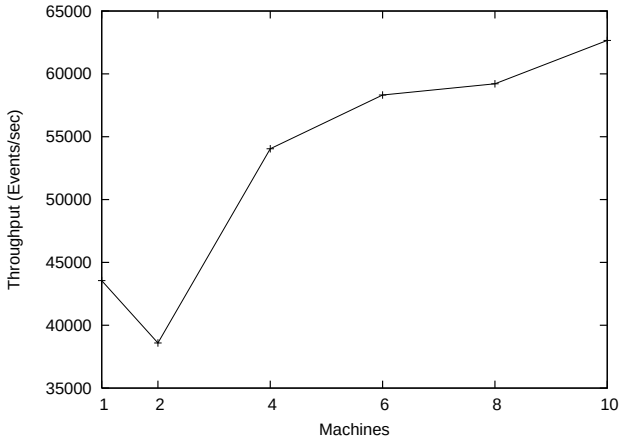

Figure 16: Throughput for vertical NFA partitioning of a length 2 pipeline.

Figure 16 shows the results of running the pipeline in Figure 15 on one machine, pipelining it on two, and replicating it to four, six, eight and ten machines. Similar to the previous experiment, results dip at two machines. This is due to the second pipeline step having to process many more events than the first. Compared to the results in Figure 13, which does not dip at two machines, a query with two middle states requires less CPU than eight middle states. This means that throughput depends to a larger degree on I/O overhead, which is lower on a single machine than on two.

There is no gain from not using a dispatcher for one and two machines. However, from four machines and up the throughput for the same number of machines is considerably lower than with a dispatcher.

5.6 Discussion

We observe from our experiments that input stream partitioning lets our NFA-based CEP system scale close to linearly. Using pipelining to partition NFAs appears to not scale well, since each new pipeline step must receive more events than the previous. This hurts throughput, which is affected by the rate of incoming events. It is possible that other workloads where the first step filters out most events would behave differently. However, in this case the first step is likely to be the bottleneck of the pipeline, thus limiting the effect of pipeline scaling.

Although using a Cayuga engine to partition the input stream is flexible and allows for complex partitioning rules, it quickly becomes a bottleneck. We introduced an alternative, a stateless dispatcher based on a hash-function that treats events as raw byte arrays. It improved the scalability of Johka an order of magnitude above that of using Cayuga to dispatch events. However, usability reasons makes a hardcoded approach less attractive.

6. RELATED WORK

Some classes of queries can be very expensive to evaluate even when the event rate is reduced through partitioning. The approach then is to reduce such queries into components that can be evaluated individually. Operator-based event processing systems such as Borealis and System S have the advantage that their query operators are strictly separated components [1, 4]. Such components can be distributed across a cluster of machines, with query plans partitioned and executed in a distributed fashion. The machines in the system then cooperate to evaluate every event. Since there is no strong notion of operators as components in NFA-based systems, this technique cannot be directly adapted to work for Johka. Our solution exploits low-level parallelism of the Cayuga automaton to divide it into sub-automata that run on separate machines connected by the network and coordinated by epoch synchronization. Pietzuch et al. implement and evaluate a distributed event processing system using finite state machines [23]. Their basic distribution principles are shared by Johka, but their implementation and evaluation is not targeted at high-performance cluster installations. Data and query partitioning for NFAs are techniques that resemble pipelined and partitioned parallelism in parallel databases [13].

Dividing queries into pipelined subcomponents enables a system to filter out non-relevant events (noise) early. Gigascope leverages programmable hardware by installing query components on the network device to reduce the amount of events that reach the higher-level application code [10, 18]. The principle is to let non-complex query components perform coarse initial filtering so the more expensive query components receive less events. Johka is geared towards a virtualized environment, where programmable network devices cannot be assumed to be available. Early event filtering can be taken even further by installing query components close to their (remote) source [22]. These approaches all assume that events are pushed towards the processing system. With a pull-based system, Akdere et al. discuss selection and timing of event pulls from remote sources based on a transmission cost metric [3]. Johka does not assume bandwidth constraints, but could use similar cost-based planning for NFA partitioning.

Fault tolerance has not been a focus in our work. However, Johka can be configured to run as process-pairs to achieve fault tolerance [24]. This would require that output stream merging also includes duplicate detection.

Few benchmarks have been proposed for complex event processing systems. The Aurora and STREAM systems have published results on the Linear Road [6] benchmark. The workload of Linear Road is a mix of continuous and historic queries, of which the latter is currently not supported by Cayuga.

7. CONCLUSIONS

The focus of this paper has been to design, implement, and evaluate a distributed event stream processing system based on nondeterministic automata (NFA). Existing NFA-based systems are centralized, run on a single machine, and consequently suffer from an upper bound on throughput. Our goal has been to improve throughput by distributing the computation across a cluster of machines. Approaches used in distributed operator-based systems could not be used, since an NFA has no clear concept of operators as separate components. Techniques such as round-robin load sharing between NFA replicas do not work. To detect patterns stretching over multiple events, all related events must be processed by the same machines.

The contributions of this paper are the following novel approaches to NFA-based distributed event processing: 1) Row/colum scaling using query-aware event forwarding, 2) Partitioning automata to pipeline the resulting sub-automata, and 3) Combining row/column with pipelining. Our results show that throughput for realistic workloads can scale when the input stream is partitioned across machines in a cluster. The lessons learned can be used to implement automatic partitioning and distributed deployment of event processing NFAs. Our analysis may also be used as a basis for dynamic load balancing during runtime. Our results show that di-

viding CPU-bound event processing load across several machines is scalable. We leave for future work to consider running multiple Cayuga engines per machine to further scale up I/O bound processing on multi-core CPU architectures.

So far we have only considered transformations of tree-shaped NFAs which enable perfect partitioning. This appears to be the most prevalent class of queries [9, 16, 18]. We leave for future work to consider NFAs with more complex dependencies. One unresolved issue is to define partitioning in mixed query workloads.

Our results have been achieved with a system that is fully compatible with the centralized version of Cayuga. Although our implementation is based on one specific system, we conjecture that our findings are generally valid for NFA-based event processing systems. For some workloads, the scalability of Johka is limited by the capacity of Spread; we conjecture that this may be alleviated by using a different network communication system.

8. ACKNOWLEDGMENTS

The authors would like to thank Alan Demers, Ken Birman, Krzysztof Ostrowski, Håvard Johansen, and Åge Kvalnes for their valuable comments and criticisms. We would also like to thank the anonymous reviewers for their insightful feedback.

9. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD*, pages 147–160, New York, NY, USA, 2008. ACM.

[3] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, 2008.

[4] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. Spc: a distributed, scalable platform for data mining. In *Proc. of DMSSP '06*, pages 27–37, New York, NY, USA, 2006. ACM.

[5] Y. Amir, C. Danilov, and J. R. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of DSN '00*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society.

[6] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proc. of VLDB '04*, pages 480–491. VLDB Endowment, 2004.

[7] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of NSDI'04*, pages 15–15, Berkeley, CA, USA, 2004. USENIX Association.

[8] Cayuga System (Accessed 11/2008). http://www.cs.cornell.edu/bigreddata/cayuga/.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *CIDR'03*, Asilomar, California, 2003.

[10] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proc. of SIGMOD*, pages 647–651, New York, NY, USA, 2003. ACM.

[11] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *Proc. of EDBT*, pages 627–644, 2006.

[12] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[13] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[14] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proc. SIGMOD*, pages 115–126, 2001.

[15] R. Friedman and E. Hadad. Adaptive batching for replicated servers. In *Proc. of SRDS '06*, pages 311–320, Washington, DC, USA, 2006. IEEE Computer Society.

[16] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation, Second Edition*. Addison Wesley, 2000. 2nd edition.

[18] T. Johnson, M. S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. Query-aware partitioning for monitoring massive network data streams. In *Proc. of the 2008 ACM SIGMOD*, pages 1135–1146, New York, NY, USA, 2008.

[19] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in Gigascope. In *Proc. of VLDB '05*, pages 1079–1088, 2005.

[20] M. Li, M. Liu, L. Ding, E. A. Rundensteiner, and M. Mani. Event stream processing with out-of-order data arrival. In *In Proc. of ICDCS '07 Workshops*, page 67, Washington, DC, USA, 2007. IEEE Computer Society.

[21] NASDAQ Performance Statistics (Accessed 11/2008). http://www.nasdaqtrader.com/trader.aspx?id=marketshare.

[22] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the 2003 ACM SIGMOD*, pages 563–574, New York, NY, USA, 2003. ACM.

[23] P. R. Pietzuch, B. Shand, and J. Bacon. A framework for event composition in distributed systems. In *Proc. of the 2003 Intl. Conf. on Middleware*, pages 62–82, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[24] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proc. of the ACM SIGMOD*, pages 827–838, New York, NY, USA, 2004.

[25] Spread Concepts LLC (Accessed 11/2008). http://www.spread.org.

[26] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of PODS '04*, pages 263–274, New York, NY, USA, 2004. ACM.

[27] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*, chapter 13, pages 603–607. Prentice-Hall, Inc. NJ, USA, 2006.

[28] W. White, M. Riedewald, J. Gehrke, and A. Demers. What is "next" in event processing? In *Proc. of PODS '07*, pages 263–272, New York, NY, USA, 2007. ACM.