2010 IEEE International Conference on Data Mining Workshops

S4: Distributed Stream Computing Platform

Leonardo Neumeyer
Yahoo! Labs
Santa Clara, CA
neumeyer@yahoo-inc.com

Bruce Robbins
Yahoo! Labs
Santa Clara, CA
robbins@yahoo-inc.com

Anish Nair
Yahoo! Labs
Santa Clara, CA
anishn@yahoo-inc.com

Anand Kesari
Yahoo! Labs
Santa Clara, CA
anands@yahoo-inc.com

***Abstract*—S4 is a general-purpose, distributed, scalable, partially fault-tolerant, pluggable platform that allows programmers to easily develop applications for processing continuous unbounded streams of data. Keyed data events are routed with affinity to Processing Elements (PEs), which consume the events and do one or both of the following: (1) *emit* one or more events which may be consumed by other PEs, (2) *publish* results. The architecture resembles the Actors model [1], providing semantics of encapsulation and location transparency, thus allowing applications to be massively concurrent while exposing a simple programming interface to application developers. In this paper, we outline the S4 architecture in detail, describe various applications, including real-life deployments. Our design is primarily driven by large scale applications for data mining and machine learning in a production environment. We show that the S4 design is surprisingly flexible and lends itself to run in large clusters built with commodity hardware.**

***Keywords*-actors programming model; complex event processing; concurrent programming; data processing; distributed programming; map-reduce; middleware; parallel programming; real-time search; software design; stream computing**

I. Introduction

S4 (Simple Scalable Streaming System) is a distributed stream processing engine inspired by the MapReduce model. We designed this engine to solve real-world problems in the context of search applications that use data mining and machine learning algorithms. Current commercial search engines, such as Google, Bing, and Yahoo!, typically provide organic web results in response to user queries and then supplement with textual advertisements that provide revenue based on a "cost-per-click" billing model [2]. To render the most relevant ads in an optimal position on the page, scientists develop algorithms that dynamically estimate the probability of a click on the ad given the context. The context may include user preferences, geographic location, prior queries, prior clicks, etc. A major search engine may process thousands of queries per second, which may include several ads per page. To process user feedback, we developed S4, a low latency, scalable stream processing engine.

To facilitate experimentation with online algorithms, we envisioned an architecture that could be suitable for both research and production environments. The main requirement for research is to have a high degree of flexibility to deploy algorithms to the field very quickly. This makes it possible to test online algorithms using live traffic with minimal overhead and support. The main requirements for a production environment are scalability (ability to add more servers to increase throughput with minimal effort) and high availability (ability to achieve continuous operation with no human intervention in the presence of system failures). We considered extending the open source Hadoop platform to support computation of unbound streams but we quickly realized that the Hadoop platform was highly optimized for batch processing. MapReduce systems typically operate on static data by scheduling batch jobs. In stream computing, the paradigm is to have a stream of events that flow into the system at a given data rate over which we have no control. The processing system must keep up with the event rate or degrade gracefully by eliminating events, this is typically called load shedding. The streaming paradigm dictates a very different architecture than the one used in batch processing. Attempting to build a general-purpose platform for both batch and stream computing would result in a highly complex system that may end up not being optimal for either task. An example of a MapReduce online architecture built as an extension of Hadoop can be found in [3].

The MapReduce programming model makes it possible to easily parallelize a number of common batch data processing tasks and operate in large clusters without worrying about system issues like failover management [4]. With the surge of open source projects such as Hadoop [5], adoption of the MapReduce programming model has accelerated and is moving from the research labs into real-world applications as diverse as web search, fraud detection, and online dating. Despite these advances, there is no similar trend for general-purpose distributed stream computing software. There are various projects and commercial engines ([6], [7], [8], [9], [10]), but their use is still restricted to highly specialized applications. Amini et. al. [7] provide a review of the various systems.

The emergence of new applications such as real-time search, high frequency trading, and social networks is pushing the limits of what can be accomplished with traditional data processing systems [11]. There is a clear need for highly scalable stream computing solutions that can operate at high data rates and process massive amounts of data. For example, to personalize search advertising, we need to

978-0-7695-4257-7/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDMW.2010.172

MICROSOFT CORP.
EXHIBIT 1012

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on March 05,2026 at 19:06:47 UTC from IEEE Xplore. Restrictions apply.

process thousands of queries per second from millions of unique users in real-time, which typically involves analyzing recent user activity such as queries and clicks. We found that user session features can increase the accuracy of the models used to predict ad relevance. This performance improvement is used to improve the relevance of the ads shown to each individual user [12]. S4 addresses the need for a general-purpose distributed stream computing platform.

It is worth mentioning that many real world systems implement a streaming strategy of partitioning the input data into fixed-size segments that are processed by a MapReduce platform. The disadvantage of this approach is that the latency is proportional to the length of the segment plus the overhead required to do the segmentation and initiate the processing jobs. Small segments will reduce latency, add overhead, and make it more complex to manage inter-segment dependencies (eg. A segment may need information from prior segments). On the other hand, large segments would increase latency. The optimal segment size will depend on the application. Rather than trying to fit a square peg into a round hole we decided to explore a programming paradigm that is simple and can operate on data streams in real-time. The design goals were as follows:

- Provide a simple Programming Interface for processing data streams.
- Design a cluster with high availability that can scale using commodity hardware.
- Minimize latency by using local memory in each processing node and avoiding disk I/O bottlenecks.
- Use a decentralized and symmetric architecture; all nodes share the same functionality and responsibilities. There is no central node with specialized responsibilities. This greatly simplifies deployment and maintenance.
- Use a pluggable architecture to keep the design as generic and customizable as possible.
- Make the design science friendly, that is, easy to program and flexible.

To simplify the initial S4 design, we made the following assumptions:

- Lossy failover is acceptable. Upon a server failure, processes are automatically moved to a standby server. The state of the processes, which is stored in local memory, is lost during the handoff. The state is regenerated using the input streams. Downstream systems must degrade gracefully.
- Nodes will not be added to or removed from a running cluster.

We found that these requirements are acceptable for most of our applications. In the future, we plan to work on solutions for applications where these limitations aren't acceptable.

The dual goals of allowing distributed operation on commodity hardware, and avoiding the use of shared memory across the cluster led us to adopt the Actors model [1] for S4. This model has a set of simple primitives and has been proven to be effective at an industrial scale through its use in a variety of frameworks [13]. In S4, computation is performed by Processing Elements (PEs) and messages are transmitted between them in the form of data events. The state of each PE is inaccessible to other PEs; event emission and consumption is the only mode of interaction between PEs. The framework provides the capability to route events to appropriate PEs and to create new instances of PEs. These aspects of the design provide the properties of encapsulation and location transparency.

The S4 design shares many attributes with IBM's Stream Processing Core (SPC) middleware [7]. Both systems are designed for big data and are capable of mining information from continuous data streams using user defined operators. The main differences are in the architectural design. While the SPC design is derived from a subscription model, the S4 design is derived from a combination of MapReduce and the Actors model. We believe that the S4 design achieves a much greater level of simplicity due to its symmetry; all nodes in the cluster are identical and there is no centralized control. As we will show, this is accomplished by leveraging ZooKeeper [14], a simple and elegant cluster management service, that can be shared by many systems in the data center.

II. Design

We define a stream as a sequence of elements ("events") of the form (**K**, **A**) where **K**, and **A** are the tuple-valued keys and attributes repectively [9]. Our goal is to design a flexible stream computing platform which consumes such a stream, computes intermediate values, and possibly emits other streams in a distributed computing environment. This section contains an example application, followed by detailed descriptions of the various components of S4.

A. Example

In the example described in Figure 1, input events contain a document with a quotation in English. The task is to continuously produce a sorted list of the top K most frequent words across all documents with minimal latency. `Quote` events are sent to S4 with no key. The `QuoteSplitterPE` object (PE1) listens for `Quote` events that have no key. `QuoteSplitterPE` is a keyless PE object that processes all `Quote` events. For each unique word in a document, the `QuoteSplitterPE` object will assign a count and emit a new event of type `WordEvent`, keyed on `word`. `WordCountPE` objects listen for `WordEvent` events emitted with key `word`. For example, the `WordCountPE` object for key `word`="said" (PE2) receives all events of type `WordEvent` keyed on `word`="said". When a `WordEvent` event for key `word`="said" arrives, S4 looks

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on March 05,2026 at 19:06:47 UTC from IEEE Xplore. Restrictions apply.

up the `WordCountPE` object using the key `word`="said". If the `WordCountPE` object exists, the PE object is called and the counter is incremented, otherwise a new `WordCountPE` object is instantiated. Whenever a `WordCountPE` object increments its counter, it sends the updated count to a `SortPE` object. The key of the `SortPE` object is a random integer in [1, n], where n is the desired number of `SortPE` objects. Once a `WordCountPE` object chooses a sortID, it uses that sortID for the rest of its existence. The purpose of using more than one `SortPE` object is to better distribute the load across several nodes and/or processors. For example, the `WordCountPE` object for key `word`="said" sends an `UpdatedCountEvent` event to a `SortPE` object with key `sortID`=2 (PE5). Each `SortPE` object updates its top K list as `UpdatedCountEvent` events arrive. Periodically, each `SortPE` sends its partial top K lists to a single `MergePE` object (PE8), using an arbitrary agreed upon key, in this example `topK`=1234. The `MergePE` object merges the partial lists and outputs the latest authoritative top K list.

B. Processing Elements

Processing Elements (PEs) are the basic computational units in S4. Each instance of a PE is uniquely identified by four components: (1) its *functionality* as defined by a PE class and associated configuration, (2) the *types of events* that it consumes, (3) the *keyed attribute* in those events, and (4) the *value* of the keyed attribute in events which it consumes. Every PE consumes exactly those events which correspond to the value on which it is keyed. It may produce output events. Note that a PE is instantiated for each value of the key attribute. This instantiation is performed by the platform. For example, in the word counting example, `WordCountPE` is instantiated for each word in the input. When a new word is seen in an event, S4 creates a new instance of the PE corresponding to that word.

A special class of PEs is the set of *keyless PEs*, with no keyed attribute or value. These PEs consume all events of the type with which they are associated. Keyless PEs are typically used at the input layer of an S4 cluster where events are assigned a key.

Several PEs are available for standard tasks such as count, aggregate, join, and so on. Many tasks can be accomplished using standard PEs which require no additional coding. The task is defined using a configuration file. Custom PEs can easily be programmed using the S4 software development tools.

In applications with a large number of unique keys, it may be necessary to remove PE objects over time. Perhaps the simplest solution is to assign a Time-to-Live (TTL) to each PE object. If no events for that PE object arrive within a specified period of time, the PE becomes eligible for removal. When system memory is reclaimed, the PE object is removed and prior state is lost (in our example, we would

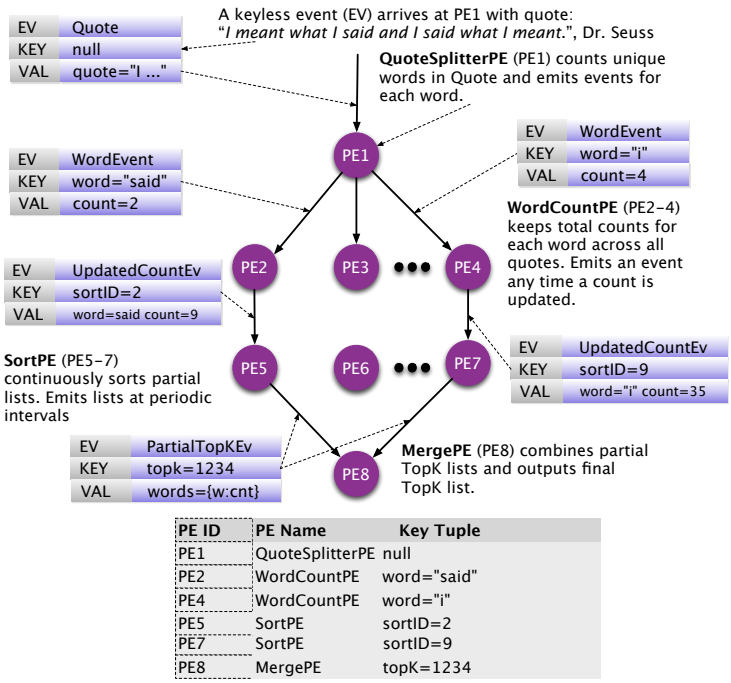

Figure 1. Word Count Example

lose the count for that word). This memory management strategy is simple but not the most efficient. To maximize quality of service (QoS), we should ideally remove PE objects based on the available system memory and the impact the object may have on the overall performance of the system. We envision a solution where PE objects can provide the priority or importance of the object. This value is application specific, hence the logic should be implemented by the application programmer.

C. Processing Node

Processing Nodes (PNs) are the logical hosts to PEs. They are responsible for listening to events, executing operations on the incoming events, dispatching events with the assistance of the communication layer, and emitting output events (Figure 2). S4 routes each event to PNs based on a hash function of the values of all known keyed attributes in that event. A single event may be routed to multiple PNs. The set of all possible keying attributes is known from the configuration of the S4 cluster. An event listener in the PN passes incoming events to the processing element container (PEC) which invokes the appropriate PEs in the appropriate order.

There is a special type of PE object: the PE prototype. It has the first three components of its identity (functionality, event type, keyed attribute); the attribute value is unassigned. This object is configured upon initialization and, for any value V, it is capable of *cloning* itself to create fully qualified PEs of that class with identical configuration and value V

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on March 05,2026 at 19:06:47 UTC from IEEE Xplore. Restrictions apply.

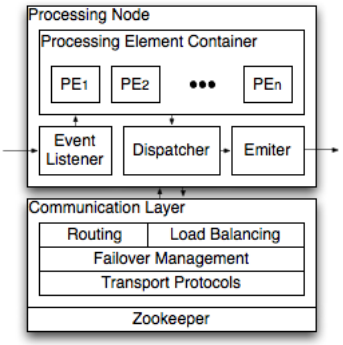

Figure 2. Processing Node

for the keyed attribute. This operation is triggered by the PN once for each unique value of the keyed attribute that it encounters.

As a consequence of the above design, all events with a particular value of a keyed attribute are guaranteed to arrive at a particular corresponding PN, and be routed to the corresponding PE instances withn it. Every keyed PE can be mapped to exactly one PN based on the value of the hash function applied to the value of the keyed attribute of that PE. Keyless PEs may be instantiated on every PN.

D. Communication Layer

The communication layer provides cluster management and automatic failover to standby nodes and maps physical nodes to logical nodes. It automatically detects hardware failures and accordingly updates the mapping [15].

Emitters specify only logical nodes when sending messages. Emitters are unaware of physical nodes or when logical nodes are re-mapped due to failures.

The communication layer API provides bindings in several languages (e.g, Java, C++). Legacy systems can use the communication layer API to send input events in a round-robin fashion to nodes in an S4 cluster. These input events are then processed by keyless PEs.

The communication layer uses a pluggable architecture to select network protocol. Events may be sent with or without a guarantee. Control messages may require guaranteed delivery while data may be sent without a guarantee to maximize throughput.

The communication layer uses ZooKeeper [16] to help coordinate between nodes in an S4 cluster. ZooKeeper is an open source subproject of Hadoop maintained. It is a distributed coordination service for distributed applications.

E. Configuration Management System

We envision a management system where human operators can set up and tear down clusters for S4 tasks, and perform other administrative operations. The assignment of physical nodes to these S4 task clusters is coordinated using ZooKeeper [16]. A subset of *active* nodes are assigned to particular tasks, while the remaining *idle* nodes remain in a pool which can be used as needed (e.g. failover, or dynamic load balancing). In particular, an idle node may be registered as a standby for multiple active nodes which may be assigned to distinct tasks.

III. Programming Model

The high-level programming paradigm is to write generic, reusable and configurable processing elements that can be used across various applications. Developers write PEs in the Java programming language. PEs are assembled into applications using the Spring Framework.

The processing element API is fairly simple and flexible. Developers essentially implement two primary handlers: an input event handler `processEvent()` and an output mechanism `output()`. In addition, developers can define some state variables for the PE. `processEvent()` is invoked for each incoming event of the types the PE has subscribed to. This method implements the logic for input event handling, typically an update of the internal PE state. The `output()` method is an optional method that can be configured to be invoked in a variety of ways. It can be invoked either at regular time intervals `t`, or on receiving `n` input events. This also means it can be invoked on every incoming event, in the case where `n`=1. The `output()` method implements the output mechanism for the PE, typically to publish internal state of the PE to some external system.

We demonstrate this with an example. Consider a PE that subscribes to an event stream of user search queries, counts instances for each query since the beginning and intermittently writes out the counts to an external persister. The event stream consists of events of type `QueryEvent`. The class `QueryCounterPE` implements `processEvent()` and `output()` as described in Figure 3. In this case, `queryCount` is the internal PE state variable that holds the count for the query corresponding to this PE. Finally, the configuration for the PE is described in Figure 4. In this case, the `keys` property tells us that `QueryCounterPE` subscribes to events of type `QueryEvent` and is keyed on the attribute `queryString` from the event. The configuration ties the PE to a data persistence component `externalPersister` (this could be an abstraction for a data serving system) and instructs the `output()` method to be invoked every 10 minutes.

IV. Performance

We introduce a real-world benchmark application, describe how we approached the problem in S4, and present some performance results.

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on March 05,2026 at 19:06:47 UTC from IEEE Xplore. Restrictions apply.

Figure 3. Excerpt from QueryCounterPE.java

```
private queryCount = 0;

public void processEvent(Event event)
{
  queryCount ++;
}

public void output()
{
  String query = (String) this.getKeyValue().get(0);
  persister.set(query, queryCount);
}
```

Figure 4. Excerpt from QueryCounterPE.xml

```
<bean id="queryCounterPE"
      class="com.company.s4.processor.QueryCounterPE">
  <property name="keys">
    <list>
      <value>QueryEvent queryString</value>
    </list>
  </property>
  <property name="persister" ref="externalPersister">
  <property name="outputFrequencyByTimeBoundary"
            value="600"/>
</bean>
```

A. Streaming Click-Through Rate Computation

User clicks are one of the most valuable user behaviors on the web. They provide immediate feedback on the preferences and engagement of the user which can be used to improve user experience by showing the most popular items in more prominent positions. In the case of search advertising that use a pay-per-click revenue model, publishers, agencies, and advertisers determine payments based on click counts. Click-through rate (CTR) is the ratio of the number of clicks divided by the number of ad impressions. When sufficient historical data is available, CTR is a good estimate of the probability that a user will click on an item. Precisely because clicks are an invaluable variable for using in personalization and ranking, it is also subject to click fraud. Click fraud could be used to manipulate ranking in a search engine. Click fraud is typically implemented by using malicious software running on remote computers (bots) or groups of computers (botnets). Another potential threat is impression spam, that is, requests originated by bots. Some of these requests may not be malicious in nature but could affect CTR estimations.

In this example (Figure 5), we show how to use S4 to measure CTR in real-time. In the context of search advertising, a user query is processed by the ad engine, returning a ranked list of ads. In the example, we measure CTR in real-time for each query-ad combination. To eliminate click and impression noise, we use a set of heuristic rules to eliminate suspicious serves and clicks. (In this example, a serve corresponds to a user query and is assigned a unique ID. For each serve, a search results page is returned to the user who may or may not click on hyperlinks. Clicks associated with that page are tagged with the same unique ID.)

Serve events contain data pertaining to the serve, eg. the serve ID, query, user, ads, etc. The click events, on the other hand, only contain information about the click, and additionally the serve ID of the serve associated with the click. To compute CTR at the query-ad level in S4, we need to route click and serve events using a key composed of the query and ad ids. If the click payload doesn't include query and ad information, we need to do a join by serve ID prior to routing the events with `query-ad` as the key. Once joined, the events must pass through a bot filter. Finally, serves and clicks are aggregated to compute CTR. A snapshot of the event flow is shown in Figure 5.

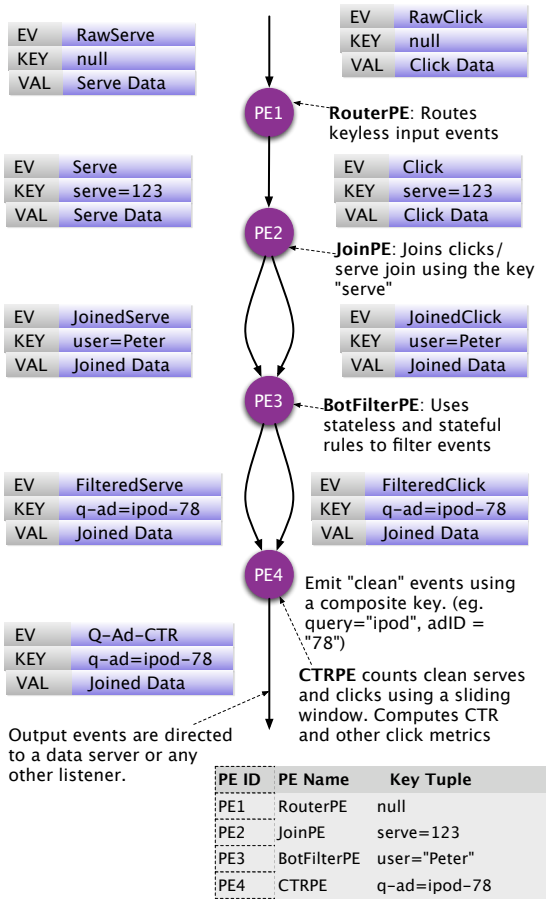

Figure 5. CTR computation

B. Experimental Setup

1) Online Experiment: We ran the streaming click-through rate (CTR) on a random sample of live search traffic. To ensure consistent experience, search engine users where assigned to the experiment randomly but fixed based on a hash of their browser cookies. On average, about one million searches per day were issued by 250,000 users. The experiment ran for two weeks. The peak observed event rate during this experiment was 1600 events per second. The

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on March 05,2026 at 19:06:47 UTC from IEEE Xplore. Restrictions apply.

experimental cluster consisted of 16 servers, each with 4 32-bit processors and 2 GB of memory.

The task was to compute click-through rate (CTR) for a query and advertisement combination with very low latency. The CTR was aggregated over a sliding rectangular window of 24 hours. This was implemented by dividing the window into "slots" of 1 hour each and aggregating clicks and impressions for each slot. Subsequently, on the hour, the aggregations from slots within the window were added up and pushed out to a serving system. This method is quite efficient with respect to memory usage, but the trade-off is in update latency. With more memory, we could maintain finer grained slots, eg., 5 minutes and reduce the update latency. The system, as implemented, provided short term CTR estimates, which were then combined with longer term CTR estimates. In the case of a PN failure, we lose the data in that node, and no short term estimates will be available for the query/advertisement instances that happen to be partitioned to that node. In this case, our failover strategy is to back off to the long term estimates.

2) Offline Experiment: We also ran an offline stress test, in which we setup a test cluster of 8 servers, each with 4 64-bit processors and 16 GB of memory. We run 16 PN's on these machines, with 2 on each. We used real click and serve data from logs of search traffic. We recreated click and serve events and computed the real CTR of search queries, which we use as gold standard for the accuracy tests. The event data consisted of 3 million serves and clicks.

C. Results

The experiment on live traffic showed that we can improve CTR by about `3%` with no loss in revenue, primarily through detecting low quality ads very quickly and filtering them out.

The offline stress test was aimed at evaluating the performance of the system under event rates far beyond the expected operating point. On the test cluster we described above, we streamed offline generated events through the S4 grid, in a sequence of runs at progressively increasing event rates. At the end of each run, we compare the CTR's estimated by the system with the true CTR's computed from search logs. Figure 6 shows the results from this test.

Events per second	Relative Error in CTR	Data Rate
2000	0.0%	2.6 Mbps
3644	0.0%	4.9 Mbps
7268	0.2%	9.7 Mbps
10480	0.4%	14.0 Mbps
12432	0.7%	16.6 Mbps
14900	1.5%	19.9 Mbps
16000	1.7%	21.4 Mbps
20000	4.2%	26.7 Mbps

Figure 6. Relative Error of CTR Estimate

The system showed signs of degrading at about 10 Mbps. The source of degradation was due to the fact that the S4 grid could not process the event stream fast enough at this rate, hence causing event loss.

V. Application: Online parameter optimization

In this section, we introduce a real-life practical application of S4: an online parameter optimization (OPO) system [17], for automating tuning of one or more parameters of a search advertising system using live traffic. The system removes the need for manual tuning and constant human intervention, while searching a larger parameter space in less time than was possible by manual searches. The system ingests events emitted by the target system (in our case, the search advertising system), measures performance, applies an adaptation algorithm to determine new parameters, and injects the new parameters back into the target system. This closed loop approach is similar in principle to traditional control systems.

A. Functional design

We assume the target system (TS) produces output that can be represented as a stream and has measurable performance in the form of a configurable objective function (OF).

We set aside 2 randomly-assigned slices of the output stream of the target system as `slice1` and `slice2`. We require that the target system has the ability to apply different parameter values in each slice and that each slice can be identified as such in the output stream.

The online parameter optimization system (OPO) has 3 high-level functional components: measurement, comparator and optimizer.

1) Measurement: The measurement component ingests the `slice1` and `slice2` streams of the TS and measures the value of OF in each slice. The OF is measured for the duration of a slot. A slot can either be specified in units of time of in terms of event counts of the output stream.

2) Comparator: The comparator component takes as input the measurements produced by the measurement component and determines if and when there is a statistically significant difference in performance between the `slice1` and `slice2` slices. When it determines a significant difference, it sends a message to the optimizer. If no difference is detected after a specified number of measurement slots, the slices are declared as equal.

3) Optimizer: The optimizer implements the adaptation strategy. It takes as input the history of parameter influences, including the latest output of the comparator, and outputs new parameter values for both the `slice1` and `slice2` slices, thus signaling the start of a new "experiment cycle".

B. S4 implementation

Figure 7 describes the implementation of the OPO system in S4. The 3 functional components were implemented

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on March 05,2026 at 19:06:47 UTC from IEEE Xplore. Restrictions apply.

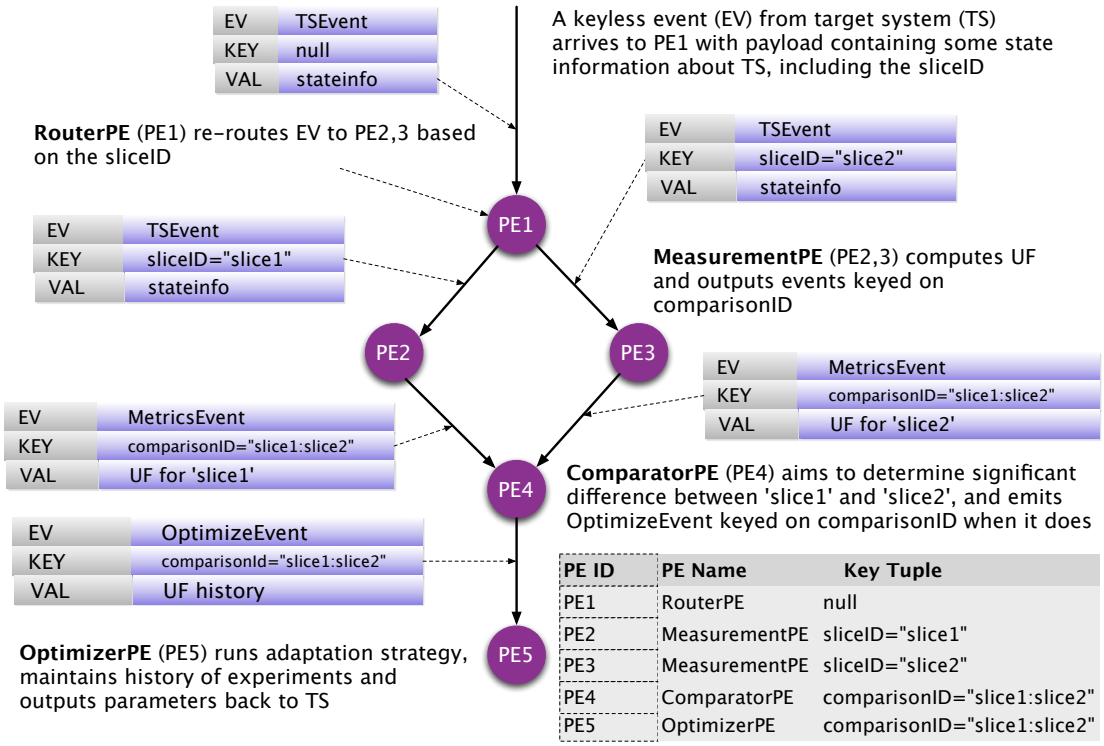

Figure 7. OPO implementation in S4

in the form of 3 PEs. The `MeasurementPE` is keyed on a `sliceID`, and there is one instance for `slice1` and `slice2` each (for more advanced strategies, we can easily extend this to more slices). Measurements of the objective function were on slots of fixed time duration. The `ComparatorPE` is keyed on a `comparisonID`, which maps to a pair of slices, in our case, `slice1:slice2`. The determination of statistically significant difference between slices was based on a dependent t-test for paired measurements. It was configured to require a minimum number of valid measurements. `OptimizerPE` is keyed on `slice1:slice2` as well. For the adaptation strategy, we use a modified version of the Nelder-Mead (aka Amoeba) algorithm [18]: a gradient-free minimization algorithm. The parameters output by OPO are fed back to the search advertising serving system. These parameter values control aspects of the serving system, therefore resulting in different user behavior.

C. Results

We ran the OPO system on real traffic slices of a search advertising system. The goal was to improve system metrics over the current parameter value (tuned using traditional methods) as much as possible. The objective function was a formulaic representation of revenue and user experience on a search engine. The traffic slices were based on partitions of search engine user space: each slice received traffic from about 200,000 users per day. The system ran for two weeks, trying to optimize a parameter known to have a significant effect on search engine performance. The optimal parameters generated by the OPO system demonstrated reasonable improvements in the primary measures of system performance: revenue by 0.25% and click yield by 1.4%

D. Summary

We demonstrated the design and implementation of an online parameter optimization system using S4. We applied this system to tuning a search advertising system with favorable results. This system can be applied to tune any dynamic system (that satisfies a few requirements described earlier) with tunable parameters.

VI. Future Work

The current system uses static routing, automatic failover via ZooKeeper, but lacks dynamic load balancing and robust live PE migration. We plan to incorporate these features.

VII. Acknowledgements

The authors would like to thank all the colleagues at Yahoo! who supported and contributed to this project especially Khaled Elmeleegy, Kishore Gopalakrishna, George Hu, Jon Malkin, Benjamin Reed, Stefan Schroedl, and Pratyush Seth. We are also grateful to Yahoo! for making the S4 source code freely available to the community at large under the Apache License, Version 2.0 [19].

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on March 05,2026 at 19:06:47 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[2] B. Edelman, M. Ostrovsky, and M. Schwarz, "Internet advertising and the generalized second-price auction: Selling billions of dollars worth of keywords," *American Economic Review*, vol. 97, no. 1, pp. 242–259, 2007.

[3] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-136, Oct 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-136.html

[4] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] Apache Hadoop. http://hadoop.apache.org/.

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

[7] 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 *DMSSP '06: Proceedings of the 4th international workshop on Data mining standards, services and platforms*. New York, NY, USA: ACM, 2006, pp. 27–37.

[8] 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*, 2005, pp. 277–289.

[9] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[10] Streambase. http://streambase.com/.

[11] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, 2005.

[12] S. Schroedl, A. Kesari, and L. Neumeyer, "Personalized ad placement in web search," in *ADKDD '10: Proceedings of the 4th Annual International Workshop on Data Mining and Audience Intelligence for Online Advertising*, 2010.

[13] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the JVM platform: a comparative analysis," in *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. New York, NY, USA: ACM, 2009, pp. 11–20.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems," in *USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[15] K. Gopalakrishna, G. Hu, and P. Seth, "Communication layer using ZooKeeper," Yahoo! Inc., Tech. Rep., 2009.

[16] Apache ZooKeeper. http://hadoop.apache.org/zookeeper/.

[17] J. Malkin, S. Schroedl, A. Nair, and L. Neumeyer, "Tuning Hyperparameters on Live Traffic with S4," in *TechPulse 2010: Internal Yahoo! Conference*, 2010.

[18] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Computer Journal*, vol. 7, pp. 308–313, 1965.

[19] The S4 Open Source Project. http://s4.io/.

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on March 05,2026 at 19:06:47 UTC from IEEE Xplore. Restrictions apply.