Benchmarking Pattern Matching Strategies for Scalable Log Analytics

Luka Leko[1,2], Sören Henning[2], Adriano Vogel[2], Otmar Ertl[2], Rick Rabiser[1]
[1]JKU/Dynatrace Co-Innovation Lab, LIT CPS Lab, Johannes Kepler University Linz, Austria
[2]Dynatrace Research, Linz, Austria
{firstname.lastname}@{jku.at, dynatrace.com}

Abstract

High-performance pattern matching on data streams is a critical capability for many large-scale, real-time systems. It can be used to identify patterns in vast volumes of unstructured data such as logs, as well as to quickly find anomalies or potentially harmful patterns. In this paper, we present our prototype of a benchmark for different matching strategies: (1) a dedicated pattern matching engine such as Hyperscan adapted for a streaming context; and (2) custom solutions using OLAP (Online Analytical Processing) databases like Apache Pinot or ClickHouse, where the database ingests from a data stream and stores the data, which can later be queried. We evaluate the approaches across different performance dimensions, including throughput and scalability. We use the Theodolite benchmarking framework to ensure reproducibility and configurability of the benchmarks. The goal of our benchmark is to make a fair comparison between specialized in-stream microservices optimized for pattern matching and a more general-purpose data analytics infrastructure that offers flexibility and extensibility.

1 Introduction

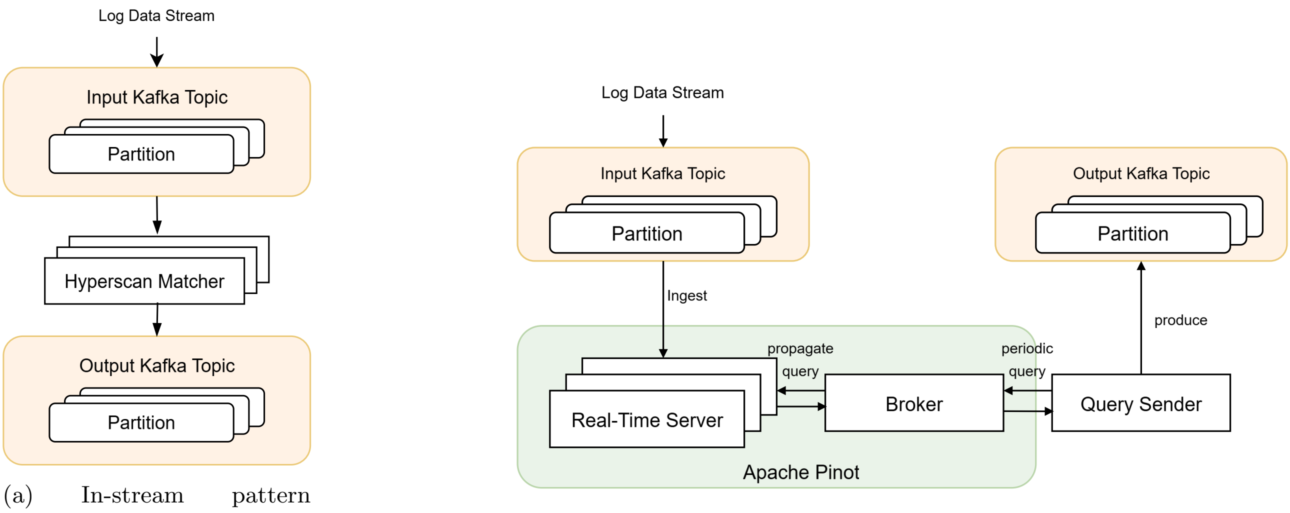
Logs are a ubiquitous component of any modern software system: System logs can be used to track the behavior of a system, capturing information about past activities or system failures. Network logs can be used to track authentication attempts, measure the amount of network traffic, or capture the response time or latency of common requests. Database logs collect information on past user queries or transaction histories, providing insight into user behavior. In short, logs are collected from practically any layer of the software stack and exploited for a variety of use cases, such as understanding user/system behavior, monitoring system reliability, failure diagnosis, anomaly detection in cybersecurity, and more.

One common operation that shows up in many core tasks in log analytics is string pattern matching, i.e., checking if a log contains one or more substrings (patterns). Such operations often need to happen in (near) real-time as part of a stream processing pipeline. Because of the large volume of logs that gets produced in modern software systems, this operation needs to be fast and scalable to ever-increasing workloads. There already exists an extensive amount of literature on the topic of pattern matching, covering both string pattern matching as well as regex pattern matching, for example, Aho-Corasick [1] or Hyperscan [2]. However, while these algorithms can be considered state-of-the-art for pattern matching engines, adapting them to the context of high-performance in-stream matching in a distributed system (where scalability, fault-tolerance, and message delivery guarantees have to be considered) can introduce a lot of additional engineering effort. We contrast this approach with the alternative of integrating stream processing logic into readily available, out-of-the-box infrastructure solutions. Specifically, we use Apache Pinot[1], a distributed OLAP datastore, which ingests the data directly from a messaging system, persists the data, and provides end-points for querying the ingested data. In that case, a lot of the necessary logic for infrastructure maintenance and performance guarantees has been passed along to Apache Pinot and is now just a matter of configuration. For example, Apache Pinot itself can handle upscaling of its individual components, and data persistence. Further, this solution can be much easier extended to support additional functionality, such as high-frequency, low-latency, high-dimensional querying (which is the big selling point of OLAP databases in the first place). In this paper, we aim to make a fair comparison between these two approaches, balancing the trade-off of computational speed to that of ease-of-use and extensibility.

2 The Benchmark Specification

We present two different approaches on how this operation can be integrated. The first approach is through a Stream-Processing application in Kafka Streams which uses Hyperscan, a high-performance multi-regex pattern matching library [2] (see Figure 1a). This architecture does all computation in-stream, consuming logs from a messaging system like Kafka, filtering the stream for strings that match at least one of the patterns, and producing back to Kafka. Both the Kafka brokers as well as the stream matching application can be scaled individually to match the work-

[1]https://pinot.apache.org/



(a) In-stream pattern matcher

(b) Query Pinot OLAP datastore

Figure 1: A comparison of the architectures of the two solutions we present for in-stream pattern matching.

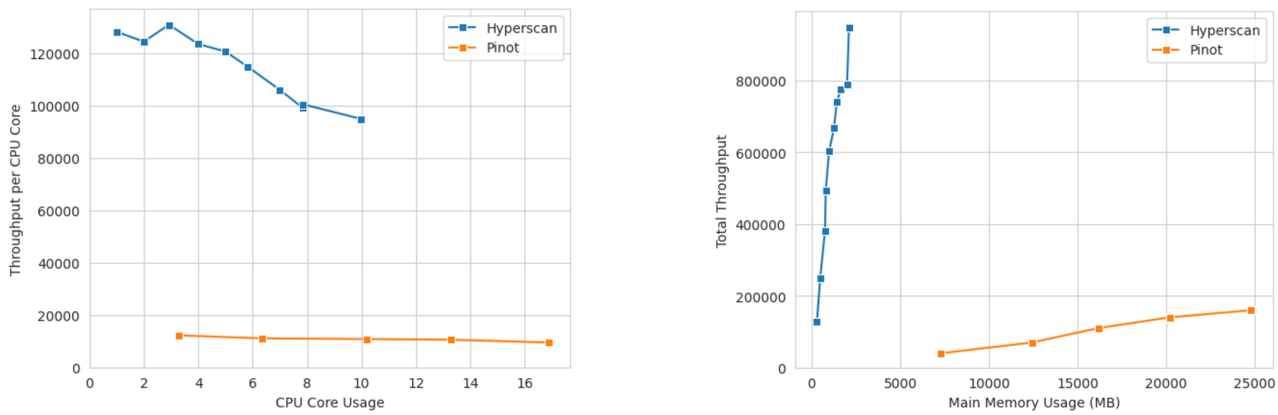
load. Concretely, every partition from the input topic is assigned to exactly one pattern matcher instance such that all topic partitions are evenly spread over the pattern matcher instances. If a new instance gets deployed, some of the topic partitions get reassigned to it, such that the overall partition assignment remains balanced among instances. Because the pattern matcher only applies a stateless transformation on the input data stream, its memory requirements are constant in the size of the workload, and only scale with the number and complexity of the patterns. Further, if both pattern matcher and the messaging system are scaled sufficiently to be able to handle the log ingest, latency will be approximately constant for every log in the stream.

In contrast, in the second approach we present, we do not filter the stream directly (see Figure 1b). Instead, we deploy an Apache Pinot cluster, which first ingests the log data from a Kafka topic. Apache Pinot is a real-time distributed OLAP (online analytical processing) datastore, optimized for executing multi-dimensional queries on large-scale, real-time ingested data. Apache Pinot consists of many different components, two of which we mention here: (1) The (real-time) servers are configured to ingest data directly from a Kafka topic. Similar to Kafka consumers, they subscribe to a subset of topic partitions, for which they are then responsible for. The servers then hold recent data in-memory. (2) Brokers provide query endpoints for users in order to query the ingested data. They hold meta-information about what server holds which segments of data, and can propagate the queries according to what the query requests. The broker collects partial results from the servers, aggregates the partial results to one result set, and returns it to the query sender. Depending on what indexes were set for a particular column in a table, different query functionality will be enabled for users. For example, enabling the “TEXT” index type for a column of a table will allow users to filter for string patterns in blob-like data, like logs. Finally, we have our own application, which periodically sends queries to the database for all logs from within the last time interval and which match at least one of the patterns, and sends the filtered logs to an output Kafka topic.

Notably, this solution introduces a base latency to every data point, ranging from 0 (for data points that were ingested into the database right before the database is queried) up until the duration of the time interval of the query application (for data points that are ingested immediately after the database was queried). Similarly, the memory requirements now scale linearly with the query interval as well as the rate of ingest, as at least all of the logs that haven’t been queried yet need to be stored.

3 Experiments and Results

We conducted an initial experiment and collected the results, measuring and comparing the throughput and scalability of the two presented approaches. The tests were run in an AWS-managed cloud environment, running in the US-EAST region. We conducted multiple experiments, varying both the load generator throughput and the compute resources. In the case of Apache Pinot, we only scale the number of real-time servers, as those handle the main bulk of computation, that is, indexing, saving the ingest to memory, and large parts of the querying logic. We configured our query sender application to query Apache Pinot every 8 seconds for all recent data points that match the set of string patterns (with an additional delay of two seconds, to account for latency). We use the publicly available loghub datasets [5, 6] as a data source during our log generation. The load generator cycles



(a) Relative Throughput to CPU Usage

(b) Total Throughput w.r.t Main Memory Usage

Figure 2: A performance comparison between the two proposed solutions.

through the given dataset for as long as it is kept alive.

We use Theodolite as a framework to benchmark the scalability of the two approaches [3, 4]. Concretely, Theodolite provides the benchmark designer with a declarative way of setting up all the infrastructure. It also handles the clean setup and teardown of every individual experiment run, and the collection of metrics on which we evaluate a particular run.

Figure 2a shows a comparison between our Hyperscan solution vs. our Apache Pinot solution. Every data point in the graph represents one experiment that was run with a fixed number of instances, ranging from 1 to 10 in the case of the Hyperscan solution, and 1 to for the Apache Pinot solution - the Pinot servers themselves employ multi-processing, which is why their total number of used CPU cores goes higher. The Hyperscan solution comes out as a clear favorite, beating the Apache Pinot solution in terms of throughput relative to the average CPU usage by a factor of about 10. Notably, the Hyperscan solution has also lost about 30% of its throughput while up-scaling from 1 instance to 10, which can be explained by the additional coordination that needs to occur on the side of Kafka when you have larger consumer groups.

Figure 2b compares the two solutions in terms of memory usage. As can be seen, again the results strongly favor the Hyperscan solution. Ten Hyperscan instances together require about 2.5 GB, while just the minimal setup of an Apache Pinot cluster with only one instance running of each component already requires 7 GB. The reason for this discrepancy is clear: The Hyperscan application never persists any results. It handles incoming data once it consumes it from Kafka, and then does not hold on to it any longer. On the other hand, a core functionality of Apache Pinot is that it provides failure-tolerant persistence of real-time data, distributed over many components, as well as managing metadata on how this data is stored across the system - a lot of functionality which is not required for this simple use case.

4 Conclusion

This paper presents a scalability benchmark for the problem of in-stream string-pattern matching, motivated by the use case of log analytics. It compares a solution utilizing Hyperscan in-stream vs. using an out-of-the-box solution with Apache Pinot, an OLAP datastore. When only taking performance into account, the Hyperscan solution comes out well on top. Nevertheless, Apache Pinot solves the key problem of managing state in a distributed system. Having such infrastructure already in place, and also being able to simply add to that system when requirements change or need to be extended, can be invaluable, and possibly worth the additional cost in performance.

References

[1] A. V. Aho and M. J. Corasick. "Efficient string matching: an aid to bibliographic search". en. *Communications of the ACM* 18.6 (June 1975), pp. 333–340.

[2] X. Wang et al. "Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs". *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 631–648.

[3] S. Henning and W. Hasselbring. "A configurable method for benchmarking scalability of cloud-native applications". en. *Empirical Software Engineering* 27.6 (Aug. 2022), p. 143.

[4] S. Henning and W. Hasselbring. "Demo Paper: Benchmarking Scalability of Cloud-Native Applications with Theodolite". *2022 IEEE International Conference on Cloud Engineering (IC2E)*. Sept. 2022, pp. 275–276.

[5] J. Zhu et al. *Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics.* arXiv:2008.06448 [cs]. Sept. 2023.

[6] Z. Jiang et al. *A Large-Scale Evaluation for Log Parsing Techniques: How Far Are We?* arXiv:2308.10828 [cs]. Mar. 2024.