Counting Millions of Ad Bids, Quickly

How Wallaroo enables real-time analytics on bidding data for Pubmatic

Sean T. Allen
VP of Engineering
sean@wallaroolabs.com

Simon Zelazny
Software Engineer

© 2019 Wallaroo Labs

Counting Millions of Things, Quickly

How Wallaroo enables real-time analytics on bidding data for Pubmatic

About Wallaroo

Wallaroo is an efficient, low-footprint, event-by-event data processing engine with on-demand scaling. Wallaroo is ideal for use cases involving high-frequency data, analytics on large amounts of event or time-series data, and AI algorithms.

We designed Wallaroo to handle demanding high-throughput, low-latency tasks where the accuracy of results is critical. Our goals with Wallaroo are to provide better per-worker throughput, dramatically lower latencies, simpler state management, and more natural operational experience than existing tools or home-grown solutions.

Many people are familiar with the Java-based tools in the “Big Data” landscape like Storm, Flink, and Spark. Wallaroo fits into this same category of distributed computing applications while providing a solution for shops that are looking for a lower overhead option with high-performance characteristics or are looking to deploy algorithms in languages other than Java. For many use cases, Wallaroo can deliver dramatically faster time-to-market and a significantly lower infrastructure footprint.

Wallaroo supports resilient, in-memory distributed state, easy cluster formation and tear-down, and the ability to grow and shrink cluster size with a running application. It comes with connectors that can ingest data from and output results to many different sources such as Kafka, Kinesis, PostgreSQL, S3, and more.

This whitepaper goes into detail on how we used Wallaroo to help PubMatic, an Adtech client, build a very low cost, high-performance system to aggregate and analyze data from their real-time bidding systems.

The Problem

PubMatic came to us with a problem: in order to facilitate the next evolution of their business, they needed to get better real-time insight into their bidding systems and they needed to do it on-budget. They'd evaluated competing solutions and decided they would not meet the requirements based on latency, scale, and infrastructure cost. That's where Wallaroo and Wallaroo Labs entered the picture.

Their problem was roughly this:

They have a stream of real-time impression and bid request data that is produced by their real-time bidding servers. They needed to take this data and

1) parse it
2) count impressions and associated bid requests
3) store the aggregate numbers in 5-minute buckets for downstream systems to consume.

The Challenges

At peak, the PubMatic receives millions of queries per second in aggregate from publishers, which translates into tens of millions of bids per second. Thus, the primary question is one of cost and performance. While it's tempting to assume that this issue can be addressed with high-performance hardware, the margins on real-time bidding activity are such that cost-effectiveness is always a concern.. This system must pay for itself by running reliably on relatively **limited hardware**.

In addition to the cost/performance ratio, the system must meet the strict response-time SLAs imposed by the rest of the bidding infrastructure. In this case, the **99.999th** percentile response time for the HTTP POST requests coming in from the bidding servers must remain within **150ms**, with average response times **under 10ms**.

Finally, to ensure that downstream analytics have the freshest data possible, the **maximum allowed processing time** between ingestion at the HTTP server and writing out to the persistence backend is **10 minutes**.

The High-level Solution

Spreading the load across multiple overlapping shards, fanning-in at the last stage

The current Wallaroo solution consists of:

1) A layer of special-purpose HTTP servers, which handle only the domain-specific payload required by the business;
2) A layer of “Tier1” Wallaroo application, whose purpose is to accumulate data in parallel from the firehose of incoming data
3) A “Tier2” Wallaroo application, which serves as a fan-in target for the previous layer, and which provides a global aggregate of the aggregates that Tier 1 pulls out of the real-time bidding firehose
4) A persistence connector application, which can be configured to write CSV files to a cloud storage backend of choice.

The Wallaroo application is split into two tiers to leverage data locality to the highest degree possible. By employing several independent Tier1 nodes, we’re able to aggregate data for a given key regardless of which node happens to receive the HTTP request affecting that key. Then, after a set time elapses, we ship all the information to a second Wallaroo cluster, which merges (or reduces, in MapReduce parlance) aggregates for the same key from different Tier1 nodes into one state object.

The diagram below illustrates the application, as deployed in a cluster of 3 r5.24xlarge AWS instances, receiving data from a load-balancer, and writing CSVs to S3.

© 2019 Wallaroo Labs

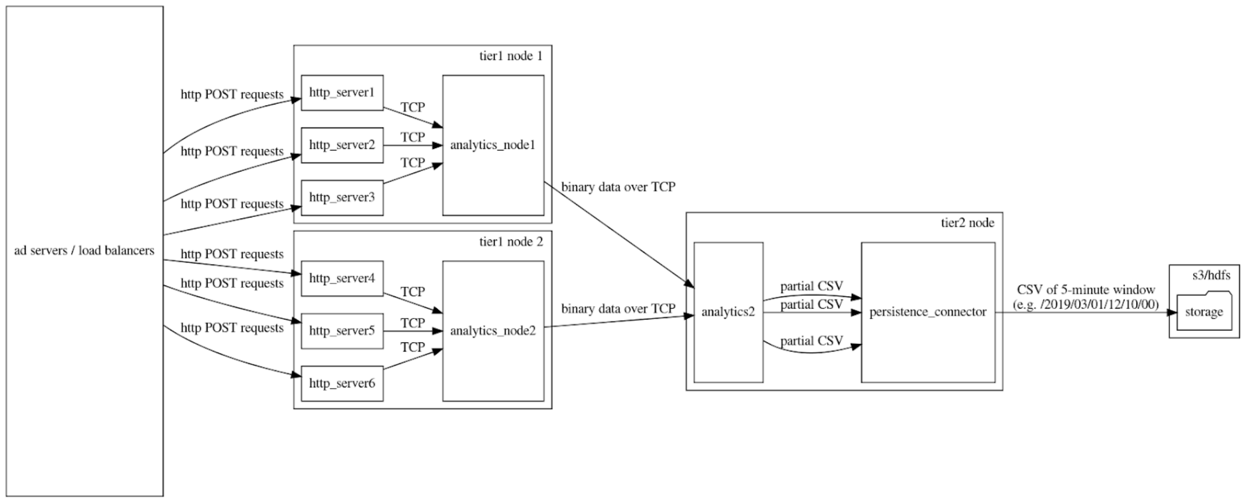

Fig. 1: High-level system architecture

The layout of the application as detailed above is as follows:

Machine type/role	Count	Running applications
R5.24xlarge Tier 1 node	2	**46** HTTP server processes **1** instance of the Wallaroo Tier 1 application
R5.24xlarge Tier 2 node	1	**1** instance of the Wallaroo Tier 2 application **1** instance of the persistence connector

The rest of this paper will cover in more detail:

- The data inputs that drive system
- Our solution for doing high-volume HTTP request acceptance and JSON payload parsing
- The Tier 1 and Tier 2 aggregation architecture
- The final hand-off of data to downstream systems via CSV

The Shape and Volume of Input Data

The data arrives in the form of HTTP POST requests, where each request contains 10 newline-separated JSON objects. Each object represents an impression (containing, among other data, information about the publisher, site, campaign, etc) and its associated ad bid requests, which vary in number from 0 to 40.

The entire POST payload is compressed using an industry standard method. For our purposes here, we will refer to the various identifier fields as ImpField1, etc.

```
{“ImpField1”:1234567,
 “ImpField2”:1234567,
 “ImpField3”:1234567,
 “Bids”:[
     {“BidField1”:123,“BidField2”:1},
     {“BidField1”:234,“BidField2”:1},
     … ]}
{“ImpField1”:9876543,
 “ImpField2”:9876543,
 “ImpField3”:9876543,
 “Bids”:[
   {“BidField1”:987,“BidField2”:2},
   {“BidField1”:876,“BidField2”:2},
   … ]}
…
```

Fig. 2: POST body, after zlib-inflation (fragment)
)

PubMatic’s current peak data volume is **400K** requests per second. Since each request contains 10 Impressions, and each impression contains on average 20 Bids, the system which implements a solution to this problem must be able to process (400,000 × 10) 4,000,000 Impressions and (4,000,000 × 20) 80,000,000 Bid Requests, for a sum of roughly 84,000,000 (**eighty-four million**) distinct logical **events per second.**

© 2019 Wallaroo Labs

Business Logic Rules

Incoming Impressions and Bids are aggregated according to the following rules:

1. Each inbound Impression increments a counter corresponding to a compound of keys that we will call ImpField1, ImpField2, and ImpField3.
2. Each Bid request increments a counter corresponding to the ImpField1, ImpField2 and ImpField3 of the parent Impression, combined with the two additional features of the Bid that we will call BidField1 and BidField2.

When processed, the following impression:

```
{“ImpField1”:9876543,
 “ImpField2”:9876543,
 “ImpField3”:9876543,
 “Bids”:[
   {“BidField1”:987,“BidField2”:5},
   {“BidField1”:987,“BidField2”:5},
   {“BidField1”:876,“BidField2”:5}]}
```

Fig. 3: A single Impression

Will update these counters:

Key	Increment
(ImpField1=9876543, ImpField2=9876543, ImpField3=9876543)	+1
(ImpField1=9876543, ImpField2=9876543, ImpField3=9876543, BidField1=**987**, BidField2=5)	+**2**
(ImpField1=9876543, ImpField2=9876543, ImpField3=9876543, BidField1=**876**, BidField2=5)	+1

Fig. 4: Corresponding aggregate increments

Note that two Bid requests that fall under the same key increase the aggregate count by 2.

© 2019 Wallaroo Labs

All counters are maintained per five-minute tumbling (non-overlapping) window. Counters that were accumulated between, say, **12:00 and 12:05**, will not be incremented by data arriving at **12:06**. This new data will increment counters in another logical window, starting at a count of zero.

12:00 -- 12:05		**12:05 -- 12:10**		12:10 -- 12:15	
Key	Count	Key	Count	Key	Count
(9876543, 9876543, 9876543)	100	(9876543, 9876543, 9876543)	23	*<no keys yet seen>*	*<will start at 0 for every new key>*
(9876543, 9876543, 9876543, 987,2)	225	(9876543, 9876543, 9876543, 987,2)	56		
(9876543, 9876543, 9876543, 876,2)	231				
After **12:05** elapses, aggregates in the yellow column will stop receiving updates and flow down to the next stage in the computation. **From 12:05 to 12:10**, the blue aggregates will accumulate counts, starting from 0 for each encountered key. **After 12:10**, the counts will again start at zero for every key.					

Fig. 5: Logical tumbling windows and their aggregations

The HTTP Server

Accepting and parsing POST requests within a strict SLA

The HTTP Server layer comprises the interface between the Wallaroo analytics system and the real-time bidding infrastructure which provides input data. While reading and parsing HTTP request payloads is an embarrassingly parallel task well-suited to a stateless cluster of front-end servers, the customer's specific set of constraints meant that we aimed for processing all 400K requests per second on as few machines as possible.

Having stress-tested and eliminated several candidate solutions (such as Nginx with Lua scripting, H2O with mruby, and Erlang/cowboy backed by RapidJSON NIFs), we finally settled on using multiple concurrent instances of a C++ server, written on top of libh2o and RapidJSON, fronted by a hardware-based F5 load-balancer.

Each C++ HTTP server opens a TCP connection to the downstream Wallaroo cluster, and for every well-formed incoming HTTP POST request:

1) inflates the zlib payload;
2) parses the JSON objects within;
3) serializes the parsed data to an efficient binary representation; and
4) sends the binary data on the TCP socket.

The C++ servers are capable of handling the peak production load of **400K requests per second** on **3 AWS r5.24xlarge** instances.

To address high tail latencies that can be introduced by network speed variability (including occasional TCP backpressure generated by the Tier 1 Wallaroo system) we split the HTTP server into two threads. The first thread is responsible for accepting, parsing and serializing POST requests, while the second sends the resulting binary data to the downstream Wallaroo system. The two threads communicate via a circular buffer, which dampens the impact of network delays and helps maintain 99.999th-percentile HTTP response times well below 150ms.

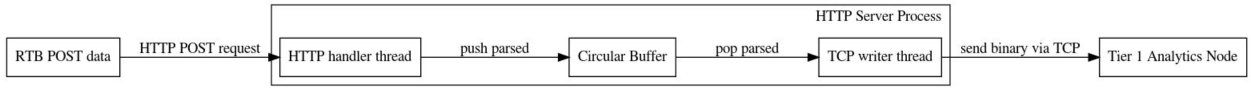

Fig. 6: HTTP front-end server architecture

Tier 1 Aggregations with Wallaroo

Reducing stream cardinality

The Tier 1 Wallaroo application accepts incoming binary data and does the processing necessary to maintain the two distinct aggregate counters as described in the Business Logic section above.

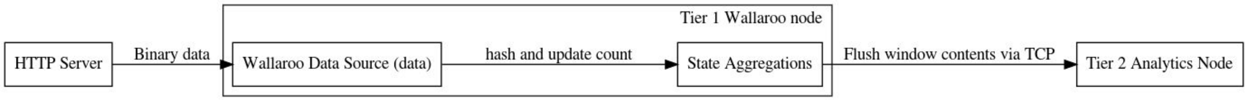

Fig. 7: Tier 1 analytics architecture

When windowing is triggered every 5 minutes, all existing aggregates are serialized and sent downstream to a single Tier 2 Wallaroo node. The Tier 2 Wallaroo node is responsible for a final global aggregation.

Tier 2 Aggregations with Wallaroo

Finalizing counts per window

Each Tier 1 application aggregates data independently, therefore the Tier 2 Wallaroo analytics application exists as a fan-in point for each 5-minute window dump. Effectively, the Tier 2 application maintains a similar structure of (key, count) pairs for every distinct Impression and Bid request. It collects data from multiple Tier 1 nodes, but benefits from a greatly reduced cardinality.

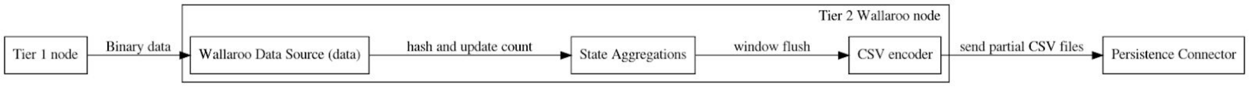

Fig. 8: Tier 2 analytics architecture

Tier 2 data windows are serialized to CSV-formatted ASCII text and sent via TCP to the persistence connector.

© 2019 Wallaroo Labs

Persistence Connector

The persistence connector receives multiple concurrent CSV fragments from Wallaroo and formats them into the final form required by PubMatic.

Summary

With the help of Wallaroo, PubMatic was able to meet their pressing launch deadlines, as well as stringent performance and cost requirements. Running on **four r5.24xlarge** AWS instances, the system is able to process and aggregate ~**84 million distinct data points per second**, while maintaining average HTTP response times **below 10 milliseconds** and 99.999th percentile response times **below 150 milliseconds.**. PubMatic is very satisfied with the current Wallaroo solution and its ability to grow with them as their volume of data continues to increase.

Since this application went into production, we added native support for the kind of data locality modeled by our two tiers. This means that with the current version of Wallaroo, the application described here can be implemented with only 1 tier, and our performance tests have shown equal or better performance.

We hope you've learned a bit about Wallaroo and are ready to learn more. Wallaroo is useful for a wide variety of use cases beyond the application explored in this whitepaper. We've included links to a number of them in the Additional Resources section below.

Additional Resources

- Wallaroo Labs Website
- Wallaroo Labs Blog
- Wallaroo GitHub repository

© 2019 Wallaroo Labs