An Introduction to Wallaroo Performance

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

John Mumm
Head of Architecture
john@wallaroolabs.com

© 2019 Wallaroo Labs

Wallaroo: An Introduction

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 faster time-to-market and a 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.

This whitepaper goes into detail on how we achieve the performance goals of Wallaroo. In this white paper, you will:

- Learn about Wallaroo by exploring an example use case and application.
- Develop an understanding of the technical underpinnings of Wallaroo’s first-in-class performance.
- Find additional resources that you can explore to learn more about Wallaroo and the problems it can help you solve.

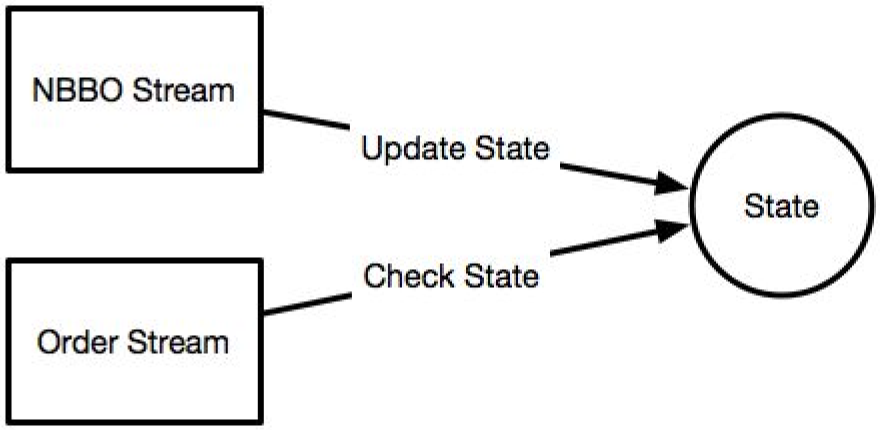
© 2019 Wallaroo Labs

Wallaroo by example: Trading Risk Checks

Let's start talking about what Wallaroo is by way of an example. One of our demo applications--called "Market Spread"--is based on a proof of concept we did for a large bank that was looking at ways to modernize their infrastructure. Market Spread is an application designed to run alongside a trading system. Its goal is to monitor market data for irregularities around different symbols and potentially withdraw some trades that have been sent to market should certain anomalies occur.

When we break the application down into its key components we get:

- A stream of market data, aka "NBBO Stream"
- A stream of trades, aka "Order Stream"
- State in the form of the latest market conditions for various symbols
- A calculation to possibly withdraw the trade based on our state for that symbol



For the proof of concept, our client was looking to handle hundreds of thousands of messages a second with median latencies measured in microseconds. Further, they needed flat, predictable tail latencies that were measured in single-digit milliseconds. While the proof of concept was only run on a single machine, Wallaroo is designed to support applications that respond to ever-increasing rates of data and seamlessly scale to run on multiple machines while handling millions of messages a second.

© 2019 Wallaroo Labs

Writing distributed data processing applications is hard. We know: over the course of our careers, we've been involved in building a lot of them. Here's a quick checklist of requirements we'd need to worry about for an application like Market Spread if we were to start from scratch:

- Create a robust communication layer between communicating processes.
- Ensure correctness in the face of failure.
- Efficiently shard data across multiple machines.
- Optimize computations for performance.
- Minimize administrative messages.
- Minimize data movement.
- Handle message source back-pressure.
- Provide metrics, monitoring, and telemetry.
- Implement our domain logic.

That's a lot to tackle and a lot of opportunities to introduce bugs. Wallaroo is, in part, an answer to that problem. It provides a robust platform that handles sharding and partitioning data across many machines. By leveraging Wallaroo, you can build demanding high-performance applications like Market Spread without having to worry about replaying messages when failures occur or deduplicating data on message replay. We focus on the hard infrastructure problems so you don't have to. If you build Market Spread using Wallaroo, the checklist of requirements becomes:

- Implement our domain logic.

Core Wallaroo abstractions

How does one go about building a Wallaroo application? Via our developer framework and its APIs. The core abstractions from our API that I want to focus on here are:

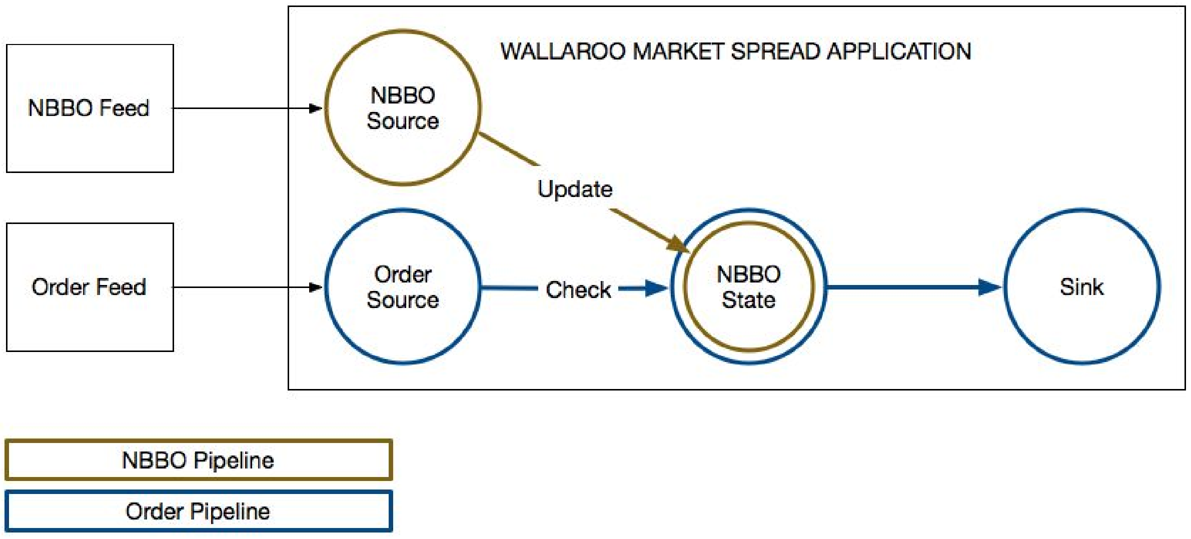
- Computation
- Source
- Sink

The most important of these is a *Computation*. Computations come in two varieties: stateless and stateful. A stateless computation takes some data as an input and creates some new data as an output. For example, a "double computation" might take in an integer such as 2 and output 4. A stateful computation is similar to a stateless computation except it takes an additional input: the state it will operate on. A simple example of a stateful computation would be a counter that keeps track of the running total of all numbers it has processed.

You can combine computations together into a data pipeline. A pipeline allows you to say, for example, that the output from computation A will be processed by computation B. A pipeline begins with one or more *Source* stages, which are responsible for receiving and decoding incoming external messages. Likewise, the pipeline may end at one or more *Sink* stages, which encode data and send it to external receivers. In this way, you can take individual computations and start turning them into applications that take in data from various external sources and ultimately produce outputs that are sent to external systems via sinks.

Market Spread in Wallaroo

Given these abstractions, what does our Market Spread application look like?



Market Spread is an application with two sources:

- one for our *NBBO Feed*
- one for the *Order Feed*

© 2019 Wallaroo Labs

We have a single type of state that we are storing (*NBBO State* by symbol), two stateful computations--one to *Update* state, another to *Check* it--and finally a *Sink* that we send any output to.

Market Spread Performance

Wallaroo has been developed to provide correct results quickly. What does quickly mean? We are able to efficiently process high volumes of data by supporting intelligent distribution across multiple machines. And on each machine, Wallaroo is designed to process millions of events with median latencies that are measured in microseconds and tail latencies that are measured in single digit milliseconds. Low latencies are all well and good, but what does that mean in practice?

Let's talk about some recent performance numbers we've gotten with Market Spread. First, though, let's highlight some key points about the application itself:

- The logic is reasonably straightforward.
- The state it stores in memory remains constant in size.
- It ingests two streams of data, only one of which occasionally produces output that results in network traffic (1 out of 70,000).
- Messages are 50 to 60 bytes in size.

During a recent performance testing run using 16 cores on an AWS c4.8xlarge class machine, we were able to run each stream of data at around 1.5 million messages per second for a total of 3 million messages per second across the two streams. Our processing latencies were:

	50%	95%	99%	99.9%	99.99%
<=	130µs	.5ms	.5ms	1ms	1ms

You can achieve even lower latencies when your throughput is less. Testing in the same environment with a total of 175 thousand messages per second across the streams, we had processing latencies of:

	50%	95%	99%	99.9%	99.99%
<=	2µs	32µs	66µs	130µs	1ms

© 2019 Wallaroo Labs

Even with the simple application caveats that we laid out, we think that is some pretty impressive performance.

What's the secret sauce?

People often ask us, "what's the secret sauce?" What they mean is, "how does Wallaroo get those great performance numbers?"

At a high level, Wallaroo's performance comes from a combination of design choices and constant vigilance. Wallaroo uses an actor-model approach that encapsulates data, minimizes coordination, and brings state close to the computation. We test every feature for performance. We reimplement functionality when we find performance lacking.

High-performance runtime

Let's start with Wallaroo's language of implementation. While it is possible to write programs that perform poorly in any programming language, some make it more difficult to write efficient programs. We wrote Wallaroo in a high-performance, actor-based language called Pony. Pony programs compile down to highly efficient native code via LLVM that can achieve performance equivalent to C.

Pony comes with a runtime that provides many desirable features. It provides an actor-based programming model that makes it easy to write lockless, high-performance code. The runtime itself comes with an efficient work-stealing scheduler that is used to schedule actors in a CPU friendly fashion. By default, a program will start one scheduler thread per CPU. When combined with CPU pinning on operating systems such as Linux, this can allow for CPU cache friendly programs. Rather than a large number of threads stepping on one another to get access to a given CPU, each CPU is dedicated to a single thread.

One of our goals with Wallaroo has been consistent, stable performance. Pony's actor model implementation helps us achieve this by eliminating any "stop-the-world" garbage collection phase, unlike the Java Virtual Machine. The

© 2019 Wallaroo Labs

JVM has a single large heap that requires all threads to be stopped to collect garbage. In the Pony runtime, each actor has its own heap that can be garbage collected by a single scheduler thread without impacting the rest of the threads running in the process. There's never a point in time when a Wallaroo application is doing nothing but collecting garbage. The result of this ongoing concurrent garbage collection is predictable performance. Many Wallaroo applications see a difference in latencies between the median and the 99.99% of less than 1 millisecond whereas JVM applications often measure that gap in terms of hundreds of milliseconds.

In a clustered environment like Wallaroo, this consistency can become a huge source of performance gains. Imagine, if you will, two processes that feature stop-the-world garbage collection that are working together. Process A feeds data into Process B. Any time Process A experiences garbage collection, no other processing will be done and Process B ends up starved for work. The stop-the-world pause on process A ends up acting as a stop-the-world for our cluster of machines. The same interaction can happen when process B experiences a stop-the-world event. When B is no longer able to process work, A will start to become backlogged and will either have to 1) exert backpressure to slow all producers down, or 2) queue large amounts of work that it needs to send to B, thereby increasing the likelihood that it will soon experience a garbage collection event of its own. Wallaroo never suffers from such cross worker pauses because there is never a stop-the-world garbage collection event to completely halt processing.

The problems that stop-the-world garbage collection can cause in a clustered environment are covered in depth in "Trash Day: Coordinating Garbage Collection in Distributed Systems". Wallaroo builds on top of Pony's efficient runtime with highly optimized code.

Avoid coordination

Coordination is a performance killer. Any time we introduce coordination into our designs, we introduce a potential bottleneck. Coordination is when two or more components need to agree on something before we can make further progress.

Coordination can take many forms. Locks are an example of coordination. Multiple threads need to update some shared data so that it remains consistent. To do this, they coordinate by introducing a lock. Consensus is another form of coordination. We see “consensus as coordination” in our daily office lives. We want to make a major decision. We need to get three people together to discuss a topic. To do this, we have to find a time everyone is available and then wait.

We have designed Wallaroo to avoid coordination. We avoid locks. We design so that individual components can proceed using local knowledge. How large of an impact can coordination have on performance? Let's take a look at one of our early design mistakes. In an early version of Wallaroo, we had "global" routing actors. Every message processed had to pass through one of these routers. Changing message routing was very easy. High performance was difficult. We have since removed the global router and replaced it with many local routers. This one change in design resulted in an order of magnitude improvement in performance.

The performance and scalability impact of coordination can be huge. In "Silence is Golden", Peter Bailis discusses the topic at length. If you are interested in learning more about how your systems can benefit from a coordination-avoiding design, we suggest you check it out.

In-process coordination-free state

Want to make a streaming data application go slow? Add a bunch of calls to a remote data store. Those calls are going to end up being a bottleneck. To maximize performance, you need to keep your data and computation close to each other.

Imagine an application that tracks the price activity of stocks. We want to be able to update the state of each stock as fast as possible. We have at our disposal three 16-core computers. We want to put all 48 cores to work updating price information. To achieve this, we need to be able to update each stock independently.

Wallaroo's state object API provides independent, parallelizable individual state “entities.” In our application, each state object would be the data for a given stock symbol. In "Life Beyond Distributed Transactions", Pat Helland presents a means of avoiding data coordination. Wallaroo's "state objects" closely resemble the independent "entities" that Pat discusses as being key to scaling distributed data systems. The state object API makes it easy to partition state to avoid coordination among partitions.

© 2019 Wallaroo Labs

Measure the “cost” of every feature

In the end, Wallaroo performance comes down to careful measurement. The performance of computer applications is often surprising. Who among us hasn't made an innocent looking change and suffered a massive performance degradation? Recognizing this reality, we've adopted a simple solution. As we add features or otherwise make changes to Wallaroo, we test the impact those features have on performance.

We seek to keep the latency overhead of any feature as small as possible. Why keep the latency as low as possible? Lowering per-feature latency is a key to increasing throughput. Increased throughput means we can do the same amount of work with fewer resources. Not clear? Don’t worry, let’s take a look.

Imagine for a moment a simple Wallaroo application. It takes some input, does a computation or two, and creates some output. We're going to measure performance in "units of work." Each unit of work takes the same amount of time. If two different computations take 1 unit of work each, they take the same amount of time. Our application has a certain amount of units of work that it takes to transform an input into an output. Let’s say each input takes a total of 7 units of work to complete. Of those, 4 units of work are for user computations and 3 for Wallaroo overhead. Let's further imagine that our computer can do 30 units of work at any one time. Given that processing 1 message requires 7 units of work, this means that at most, we can process 4 messages at a time.

$30 / 7 \approx 4$

Now, let's say that we can lower our Wallaroo overhead from 3 units to 1. If we do that, it will only take 5 units of work to process a message. And with that change, we can handle 6 messages at a time. That's a 50% improvement over what we were doing before!

$30 / 5 = 6$

That's a simple, contrived example but the basic logic holds in the real world. The less time it takes to complete a given task, the more times we can complete that task. We take this approach every day when building Wallaroo. Watch the overhead; take fewer resources to do the work; save money.

Additional Resources

We hope you've learned a bit about Wallaroo and are ready to learn more.

Referenced Materials

- Performance testing a low-latency stream processing system
- Pony programming language
- Trash Day: Coordinating Garbage Collection in Distributed Systems
- Silence is Golden: Coordination-Avoiding Systems Design
- Life Beyond Distributed Transactions

Further Reading

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

© 2019 Wallaroo Labs