Performance of 3Forge AMI
Realtime Visualization Solution

A customized benchmark test

SUT ID: 3FG190902

Rev 1.0 13 October 2019

Key Results

Response times*:

- Average response time for 40 table rows, during ingest of 25K msg/sec: 317 milliseconds
- Average response time for 40 table rows, not during ingest: 273 milliseconds
- Average response time for a chart with 2 million data points, not during ingest: 3.67 seconds

Data freshness*:

- Average age of most recent data returned in request/response queries during ingest of 25K msg/sec: 149 milliseconds
- Average age of most recent data displayed in an auto-updating table during ingest of 25K msg/sec: 247 milliseconds

** Error is within +/- 50 milliseconds. 25K msg/sec equated to ~700K database fields/sec.*

NOTE: The tests in this STAC Report were developed specifically for this project. They are vendor-independent but are not official STAC Benchmark specifications.

STAC
SECURITIES TECHNOLOGY ANALYSIS CENTER

Disclaimer

This document was prepared by the Securities Technology Analysis Center (STAC®) at the request of 3Forge. This document is provided for your internal use only and may not be redistributed, retransmitted, or published in any form without the prior written consent of STAC. "STAC" and all STAC names are registered trademarks or trademarks of the STAC. All other trademarks in this document belong to their respective owners.

The test results contained in this report are made available for informational purposes only. Neither STAC nor the vendor(s) supplying the information in this report guarantee similar performance results. All information contained herein is provided on an AS IS BASIS WITHOUT WARRANTY OF ANY KIND. STAC explicitly disclaims any liability whatsoever for any errors or otherwise.

Contents

References

[1] STAC Configuration Disclosure for SUT ID 3FG190902 (this SUT): www.STACresearch.com/3FG190902. Available to firms with premium subscriptions

Summary

Supporting human analysis of realtime, streaming data throughout an enterprise is an ongoing engineering challenge in finance. In that context, 3Forge recently asked STAC to perform some basic performance tests of its AMI realtime data visualization platform. This report explains the test setup, workloads, and measurement methodologies along with the results they yielded.

The "stack under test" (SUT) consisted of AMI and supporting software running on two machines directly connected over 25 GbE: 1) a Linux server hosting the AMI back end and STAC test harness software, and 2) a Windows server supporting a browser that hosted the AMI display (see [1] for details). The Linux server had only one CPU with 8 cores, which is fewer cores than most firms would probably run in production.

STAC software streamed a fixed rate of 25,000 simulated trade messages per second into the AMI back end, resulting in roughly 700,000 database fields per second. The tester initiated certain queries from the front end during and after message ingest.

Our purpose was to measure two things during and after ingest: 1) response times (the difference between query submission and completion) and 2) data freshness (roughly how long it took once the STAC software supplied a record to AMI for the record to be included in query results). Measuring the ingest capacity of the system was not an objective, given that the server used for the AMI back end had a low core count and also had to support the STAC software supplying the streaming data.

We measured response times with the aid of video capture and an on-screen millisecond timer. Data freshness measurements combined video screen times with STAC timestamps from message supply. An analysis of error concluded that the averages in this report are accurate to well below +/- 50 milliseconds, our objective for these human-oriented use cases.

Through these tests, we answered five questions with respect to this simple test setup:

Response times

1. "Top 40" during ingest. How long, on average, does it take to obtain 40 table rows representing the 20 most recent orders and 20 most recent executions, while the SUT is ingesting 25,000 messages per second (700,000 database fields per second)?
    a. Answer: 317 milliseconds

2. "Top 40" without ingest. Same question as #1 but after ingest has concluded.
    a. Answer: 273 milliseconds.
    b. This implies that ingest at this rate had less than a 50-millisecond impact on the response time for this type query.

3. What is the average response time for a chart of 2 million data points after ingest is over?
    a. Answer: 3.67 seconds

Data freshness

4. During this ingest rate, how much delay is there between an event arriving at the system and it being available in request/response queries?
    a. Answer: 149 milliseconds

5. During this ingest rate with a continuous query in effect, how much delay is there between an event arriving at the system and appearing at the top of the query display when the display updates?
    a. Answer: 247 milliseconds

1. Business context

Trading businesses are event driven. The job of these firms—brokers, exchanges, proprietary trading firms, hedge funds, and the like—is to respond to opportunities and threats presented in fast-flowing streams of information, especially data from the market as well as trade-related messages from counterparties, customers, and groups within the firm. Even in small firms, these event streams can flow in massive volumes.

The major trend of the last two decades has been automation of huge swathes of trading functions.[1] Given this migration of decisions and actions from humans to machines, it is easy to forget that a number of important functions still rely on human analysis. In fact, the increase in automation is actually increasing the need for some kinds of human analysis. For example, the notorious Knight Capital disaster—in which an uncontrolled trading application lost approximately $10 million per minute for about 45 minutes—drove a new level of dedication throughout the industry to ensure that humans had the means to understand how the robots were behaving and to exert proper control over them (including pulling the plug if necessary).

Human analysis requires visual tools: charts, tables, and other displays that summarize information and allow dynamic drill-down. The front office, where trading decisions are made and trading algorithms are designed, has long had sophisticated tools for visualizing streaming data (think user terminals for market data and order management). However, these tools, their underlying infrastructure, and their commercial models do not necessarily cater to needs in the middle and back office. In particular, off-trading-floor use requires browser-based displays, a lightweight network impact, and a low overall cost per user.

At the same time, trading firms report that popular off-the-shelf visualization tools aimed at a broad range of enterprises aren't well suited to streaming data. These tools can be masterful at allowing end-user analysis of static datasets, but when things turn event driven, they simply can't keep up.

So trading firms are left with two alternatives: build a solution themselves (typically involving an off-the-shelf, high-performance columnar database with a custom-built presentation layer) or buy a product specifically designed for this problem.

3Forge offer such a product, called AMI. 3Forge recently asked STAC, as an independent third party, to perform some basic tests of AMI. Whether building or buying a visualization solution for streaming data, customers need to understand a number of performance characteristics. This study explores some of the most basic of those—query response times and data freshness—in a very simple test setup. Of course many more performance and functionality questions apply, but these basic tests are a starting point.

[1] See for example, the cover story of the October 5, 2019 *Economist* or the last 12 years of work at STAC.

2. Product Background

3Forge submitted the following information and claims about its products:

> *The AMI platform is a data virtualization layer with hundreds of adapters, a complex event processor for high speed processing of realtime data, and a web based dashboard with an easy to use drag and drop interface for rapidly building front-end applications. It's deployed on-site or in the customer's cloud, and the data agnostic nature of the platform has enabled its use in countless unique use cases.*
>
> ***Interfaces***
>
> ***Outbound on-demand queries*** *- Execute queries on any number datasources concurrently and then join/prepare the data for display, further analysis or internal storage These can be invoked from user actions, timers, triggers, (or in bound queries) etc.*
>
> ***Inbound queries*** *- External systems can query AMI using our standard SQL client interface, which can then be processed internally and/or delegated to an attached datasource.*
>
> ***Realtime Data subscription*** *- Subscribe to real-time data feeds for a live view of data and real-time analytics.*
>
> ***User solicited callbacks*** *- Users can "take action" through the dashboard which can invoke callbacks to external systems.*
>
> ***Entitlements & Naming Services*** *- AMI can, upon user login, delegate single-sign on to a third party system and consume user-specific entitlements to drive dashboard abilities. Additionally, user preferences can be stored/loaded via adapters and AMI can interact with Proprietary naming services for organizations that support dynamic availability of data/compute resources.*
>
> ***Web Dashboard Interface*** *- Most important of all, Users can interact with AMI through an advanced web interface which includes a comprehensive set of tools for building interactive/dynamic dashboards.*
>
> ***Components***
>
> ***AMI In-memory Database / Event Processor***
>
> *Data is sent into high speed columnar tables, with streaming aggregations, joins and unions. Triggers, Timers and procedures with micro-second run times can be used to analyze and mutate data. External systems have access via our standard JDBC connector. Our dashboards are directly linked in for instantly updating visualizations. Data can be persisted to disk, with guaranteed messaging, and replication for powerful scalability.*
>
> ***AMI Data virtualization Layer***
>
> *Link any number of data sources to allow for seamless access to a variety of systems. Data can be accessed concurrently and then enriched/joined/unioned/reconciled on the fly. Our interface*

supports download and upload capabilities allowing for ETL processing. Decision based queries allow for advanced business/workflows.

Browser Based Dashboard Building and UI

Unprecedented speed and scale of data visualization/interaction with tables and charts handing tens of millions of data through a web page—without downsampling—at scale. Endless layout configurations possible. Native multi-monitor support. Drag & Drop form builder with dozens of widget types allow for easy and dynamic input forms. Inheritance based styling with theme configuration. Support for CORS technology allowing distinct dashboards to be linked. AMIScript can be used to build workflows and dynamic user experiences. Containers can be tuned for reactive scaling across device types (desktop, surface, mobile of choice). Dashboards support team development and integration with popular source control products

Scaling

Vertical

The platform can be split out logically into 3 distinct layers, each running on separate hosts communicating of TCP/IP.

A web-tier where user micro-actions are executed and visualizations are maintained..

A relay-tier for directly accessing external sources, executing command and remote scripts.

A central layer where the in-memory database resides and data is processed,persisted and multicast. This layer connects directly to both the web-tier and connectivity-tier.

Horizontal

Any number of web-servers can be connected to the central layer allowing for regional distribution of data and scaling up as more end-users are added.

Any number or relays can be connected to allow for distribution of localized connectivity of disparate data sources.

Map Reduce

Any number of centers can be linked together in a "tree" like fashion allowing for very complex calculations and exploration across massive datasets.

Guaranteed Messaging & Persistence

Real-time streaming uses a store-and-forward technique and client message acknowledgement (receipt) to guarantee messages are not lost. Relays can be configured to stream data to multiple consumers, enabled hot-hot or hot-warm configuration. The Center's columnar database can be configured on a per-table basis for persistence enabling full recovery in the case of an outage.

3. Test Methodology

Figure 1 provides a simplified view of the test setup, workloads, and measurement.

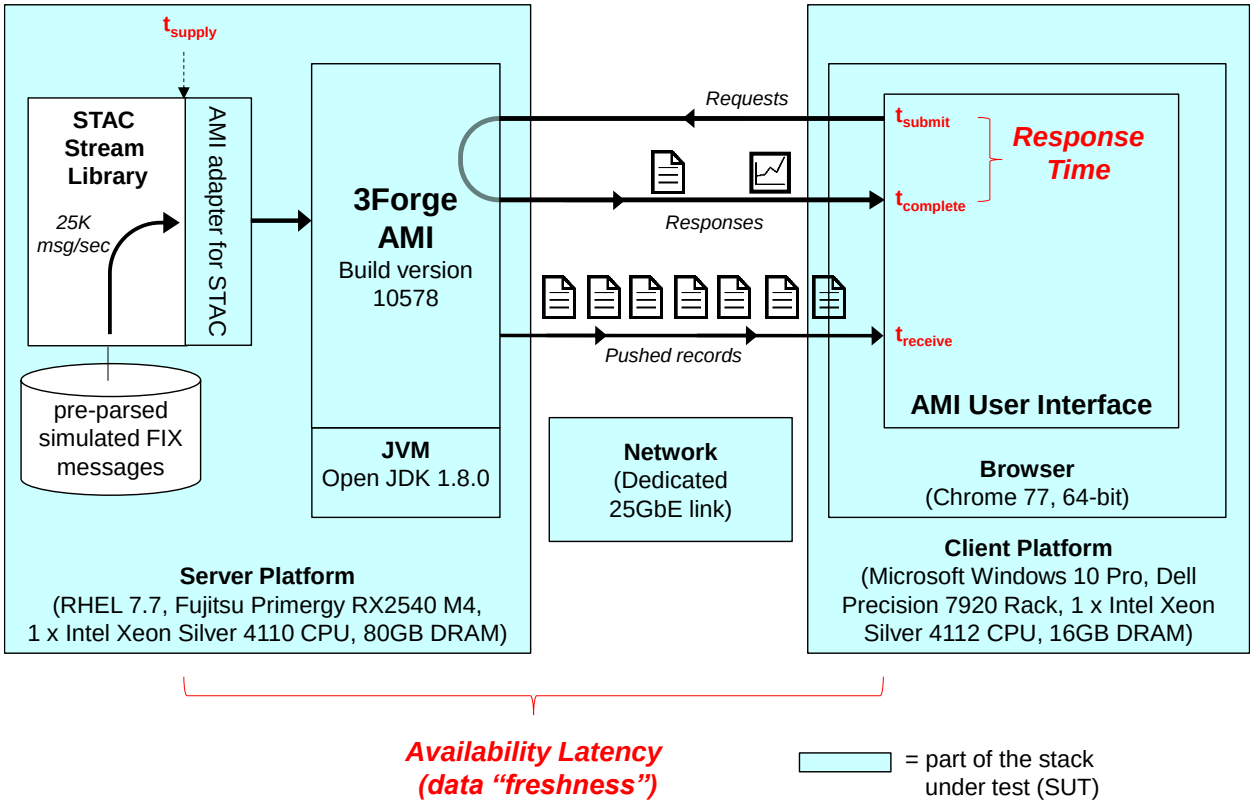

Figure 1

3.1 Setup

The test setup had the following major elements:

- Test Harness Software:
    - **STAC Stream Library.** This C-language library pushes streaming messages to a SUT-specific adapter via a STAC API. The library notes the time that it makes a given message available to the adapter (the Supply Time), irrespective of when the adapter actually picks up the message. That is, queuing delays in the adapter do not pause the clock. The latency clock starts ticking at the Supply Time.
    - **STAC Message File.** The STACstream Library loads a message file into memory prior to tests, which it loops through as many times as necessary to generate the required message rate over the required period, assigning a monotonically increasing sequence number each time it supplies a message. The message file in this project consisted of roughly 11,000 simulated new orders and execution reports. These were based on FIX 4.2 but were pre-parsed into C structs because the intent of this project was to minimize

any parsing overhead imposed on the SUT, putting all focus on the performance of the core AMI product infrastructure. Records were either orders or executions.

- The "Stack Under Test" (SUT):
  - **3Forge AMI.** The product from 3Forge that consumes streaming data and enables users to perform analytics on it via browser-based displays. See Product Background, above, for more information. In this project, we tested build version 10578 of AMI.
  - **A custom-developed adapter.** Code written by 3Forge that consumes from the STACstream Library and sends data to the AMI database. This stands in for a real-world adapter to a source of streaming data. As noted above, a real-world adapter would also typically have parsing duties, but those were excluded from this test setup.
  - **Server environment.** The physical machine on which the AMI server ran, plus the Linux operating system and Java virtual machine.
  - **Client environment.** The physical machine on which the AMI client ran within a browser. Chrome on Windows 10 Pro.
  - **Networking.** In this SUT, the Linux and Windows machines were connected directly via 25 GbE adapters without the use of a switch.

    It's important to understand that all of these components can contribute to the performance of the end-to-end system. They are all part of the solution "stack". In STAC's experience with streaming data distribution systems, the server tends to be the bottleneck. In this test setup, the server was fairly low-range, with just a single CPU with 8 physical cores. The network and client machines were beyond the requirements but were what we could make available within the project timeframes.

- Test conditions:
  - **Streaming ingest.** Several of the tests were conducted while data streamed into the SUT. Measuring the ingest capacity of the system was not an objective of this project (nor would the results have been particularly relevant, since the server CPU configuration was smaller than what we believe most firms would use in production, and the server also dedicated resources to supporting the STAC Stream Library instance that supplied the messages). Instead, we chose a fixed rate of 25,000 messages per second as a respectable message rate for a given analytic context, such as the activity of a major client of a broker.[2] After the AMI adapter added a few timestamps to the messages, each order message had 20 fields, and each execution had 33. Fields had varying types and lengths. Given the mix of orders and executions, 25,000 messages per second equated to about 700,000 fields per second.

[2] According to 3Forge, an AMI installation typically consists of multiple servers, each consuming a subset of the total messages. The system virtualizes the data so that a single query can draw from multiple partitions.

- **Background load.** In addition to AMI, its adapter, and standard Linux services, the Linux server also supported the STAC Library instance that supplied the messages, an X Windows client, Java-based timer program (described below), and a PTP client. The Client server hosted the Chrome browser, X Windows server, Java-based timer program, PTP client, and standard Windows services.

3.2 Measurements

While these tests were not based on standard benchmarks (there are no standards as far as we're aware), we did take the customary STAC approach of making them applicable to any SUT. That is, we treated the SUT as a black box and took measurements only from the outside (not relying on measurements reported by the SUT).

There were two kinds of measurement in this project:

1. **Response times.** The time that passes from the moment the user submits a query to the moment the complete results appear on the screen. We defined the moment of submission as the video frame in which the submit button changed color indicating that the user had released the button click. The moment of complete results was defined as the video frame in which all required rows were filled on the screen. The time assigned to a given video frame was taken from a local display clock that was visible on the client in the video frame.

2. **Availability latency (data freshness).** The time that passes from the moment a message is supplied to the SUT Adapter to the moment it is available for querying (eligible to appear in query results). Another way to think about this is how "fresh" or "stale" the data in the database are. For example, if a solution has trouble keeping up with a burst of traffic, it may queue messages before adding them to the database. Depending on message rates and buffer sizes, this queuing could conceivably result in seconds of delay. The user may even be unaware that the information returned in queries or streaming onto the screen is stale. Measuring availability latency is important, because in STAC's experience, the range of solutions that can sustain high throughput is larger than the range that can sustain high throughput without being subject to significant delays.

   Since we treated the SUT as a black box, this is not something we can measure directly. Therefore, we obtain upper-bound estimates for it in two ways, while messages are streaming into the SUT:

   - Submit a request/response query, find the most recent record in the response, and subtract the supply time of that record (t_{supply}) from the time that the query completed ($t_{complete}$). We obtain t_{supply} for a given record from a log that is output by the STAC library. This difference is a conservative estimate of the availability latency, because $t_{complete}$ must occur after the record is available for querying. The error in this estimate increases with the amount of query overhead, so we want these queries to induce minimal load.

   - Open a continuous query, in which tables automatically update with new records. Take a video frame from the moment the screen refreshes, find the most recent record on the screen, and subtract the supply time of that record (t_{supply}) from the current time on the display ($t_{receive}$).

3.3 Measurement error

Our objective was to obtain the measurements above with an accuracy no worse than +/- 50 milliseconds. This is many orders of magnitude less accurate than other STAC tests with streaming data (which can now have errors has low as fractions of a nanosecond), but we considered it sufficient for a product designed for visual end use.[3]

There were three known sources of error:

1. Video frame rate. We chose a rate of 60 frames per second, which equates to a little under 17 milliseconds per frame.

2. Jitter in the frame rate and client clock display. There was no basis to assume that the frame rate was steady. Nor could we assume that the clock display incremented at a steady rate. We had to go to considerable lengths to arrive at a solution with the required stability and to convince ourselves that we had. For details, see [1].

3. Time sync between the boxes. The availability latency measurements relied on timestamps from two different clocks (one in each server). Time sync between the boxes was therefore important. Synchronizing Windows with Linux and gaining confidence in that synchronization is non-trivial. Again, the details of the solution are in [1]. In the end, we were confident that the Linux and Windows clocks were within 1 millisecond of each other.

4. Timestamp-assignment delay in the Linux server. As STAC has explained elsewhere, having an accurate clock is not a guarantee of accurate timestamps, because in any system there are delays in obtaining timestamps. In non-deterministic systems such as a server with a standard OS, those delays can be quite variable and quite large. But not large in the context of this project. STAC's tests in the past have found delays on the order of 10 milliseconds in untuned systems, which would be egregious for automated trading but acceptable for the measurements of this project.

The considerations above led us to conclude that individual request/response measurements, which involved two timestamps on the Windows box, were probably subject to error in the neighborhood of +/- 34 milliseconds. That implies that our estimates of the mean had an error about half of that (~15 milliseconds), given the sample sizes. Measurements of availability latency, which involved one timestamp from the Linux server and one from the Windows server, had error of just +/- 28 milliseconds at the outside, bringing the standard error of the means down to about 6 milliseconds, thanks to the larger sample size.

Thus, we are comfortable that we met our accuracy goal. Nevertheless, because the error analysis is not an exact science, and it's possible there's error we did not account for, we conservatively cite the error margin as +/- 50 milliseconds.

[3] Literally less than a blink of an eye. See https://bionumbers.hms.harvard.edu/bionumber.aspx?&id=100706&ver=4

4. Project Participants and Responsibilities

The following firms participated in the project:

- 3Forge LLC
- STAC

The Project Participants had the following responsibilities:

- 3Forge sponsored the project; provided feedback on the test specifications; supplied, installed, and configured the AMI software and JVM; wrote the AMI adapter for STACstream; wrote a utility to inject outliers into the message file; and wrote a simple clock program to display the time in milliseconds.

- STAC developed the test specifications with 3Forge; provided the STACstream Library, message file, and associated materials; and conducted the benchmark audit, which included executing the tests and documenting the results.

5. Contacts

- 3Forge: info@3forge.com
- STAC: info@STACresearch.com

6. Results Status

STAC developed these benchmark specifications especially for this project. They were NOT developed by a working group within STAC Benchmark Council and are not to be considered standards. The tests were conducted by STAC.

7. Limitations

As with any benchmark project, this one had limitations:

- Limited sampling. The output of this SUT is visual, which required the data samples to be taken by hand. The number of samples was therefore very limited compared to a typical STAC study that involves hundreds to hundreds of millions of samples. As a consequence, the numbers in this study should be taken as indicative of the test runs performed rather than comprehensive. Most importantly, this study would not be able to detect transient issues within the SUT such as queuing, except by the sheerest luck. In general, averages do not provide insight into system variability over time, which requires greater sampling.

- Realism of workloads and solution scale. Any simulation is just that: a simulation. It is not possible to define a single “standard” data visualization workload given the wide variety of data sources and use cases in the real world. Likewise, any real deployment will have more than one client application and probably more than one server. Section 3 describes the rationale for the workloads chosen for this project.

8. Procedures and results

8.1 Response times

8.1.1 Requesting the "Top 40" records during ingest

In this test, we launched a query that requested the most recent 20 new orders and the most recent 20 executions, as shown in Figure 2. This was a one-off query (as opposed to continuous). Table 2 shows that the mean response time was 317 milliseconds.

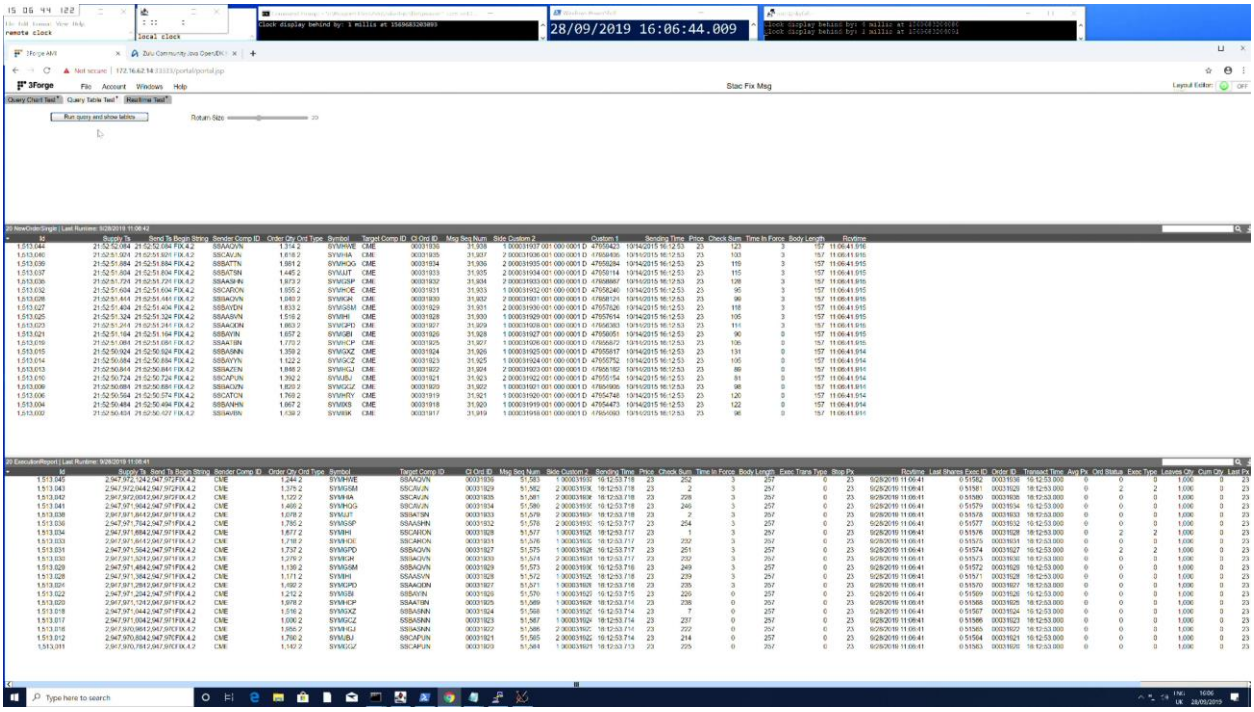

Figure 2

Request/response times Top 40 table (during ingest)		
Submit time	**Complete time**	**Elapsed time (milliseconds)**
15:05:44.783	15:05:45.183	400
15:06:01.452	15:06:01.718	266
15:06:21.087	15:06:21.354	267
15:06:41.922	15:06:42.240	318
15:06:56.290	15:06:56.623	333
	AVERAGE	**317**
(Error less than +/- 50 milliseconds)		

Table 1

8.1.2 Requesting the "Top 40" records with no ingest

This test took place immediately after the previous and was the same except that ingest had concluded. The main point of this test was to get an idea of the degree to which query response times suffered while the back end of the system was consuming at the fixed ingest rate.

Request/response times Top 40 table (not during ingest)		
Submit time	**Complete time**	**Elapsed time (milliseconds)**
15:09:31.402	15:09:31.602	200
15:09:50.671	15:09:50.971	300
15:10:10.638	15:10:10.905	267
15:10:32.208	15:10:32.507	299
15:10:50.977	15:10:51.275	298
	AVERAGE	**273**
(Error less than +/- 50 milliseconds)		

Table 2

Comparing the results from Table 1 (317 milliseconds) to Table 2 (273 milliseconds), we see that this level of ingest increased the average response time by less than 50 milliseconds. In fact, that estimate of the response-time difference is probably conservative. Whereas a Top 40 request during ingest always returns different values, a Top 40 request after ingest will always return the same values. Thus, the latter query could benefit from caching at some levels.

8.1.3 Charting 2 million data points

This test involved playing 2 million records into the system and requesting a chart involving all 2 million records (an XY plot of order quantity versus sequence number) once ingest had stopped (see Figure 3, an XY plot of 2 million records, with cursor hovering over an outlier in the upper left).[4] To provide some confidence that the chart actually contained all the data points, we injected outliers which we then inspected (hovering over an outlier revealed its values, as shown in the screenshot). All outliers were accounted for. We ran this test five times, each time emptying the database and re-ingesting data. The results are summarized in Table 3. The average response time for this chart was 3.675 milliseconds.

[4] It would also be nice to study such a query during ingest, but in manual tests like this, it would be difficult to sufficiently control circumstances like how much additional data has streamed in at the time of request.

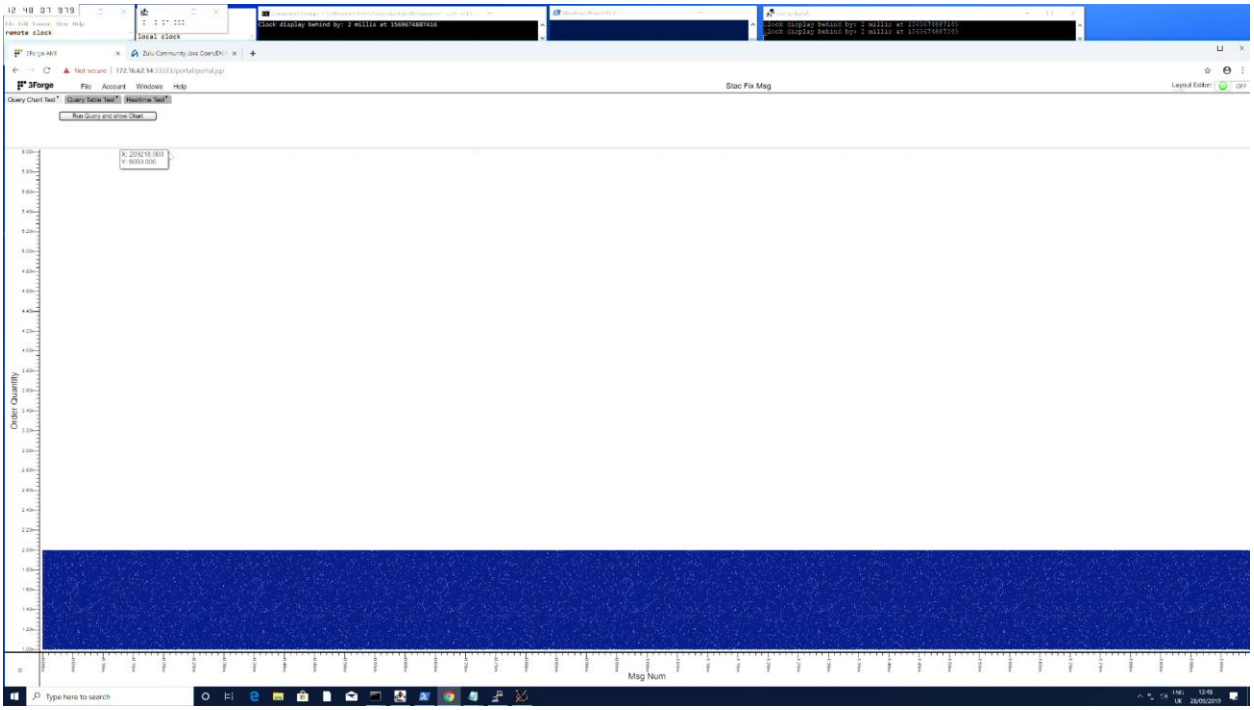

Figure 3

Request/response times 2-million data point chart (not during ingest)		
Submit time	**Complete time**	**Elapsed time (milliseconds)**
12:47:40.877	12:47:44.644	3,767
06:59:07.775	06:59:11.610	3,835
07:03:15.996	07:03:19.498	3,502
07:12:00.709	07:12:04.510	3,801
07:16:34.799	07:16:38.267	3,468
	AVERAGE	**3,675**
(Error less than +/- 50 milliseconds)		

Table 3

8.2 Estimated availability latencies (data freshness)

8.2.2 Tiny query

The purpose of this test is to estimate availability latency through one-off queries. As described in the Measurements section above, we wanted this query to induce as little load as possible on the system. So it simply requested the single most recent order and execution in the database (the fact that it requested two types of records rather than just one was an artifact of the query that 3Forge constructed for us). Figure 4 shows an example of query results.

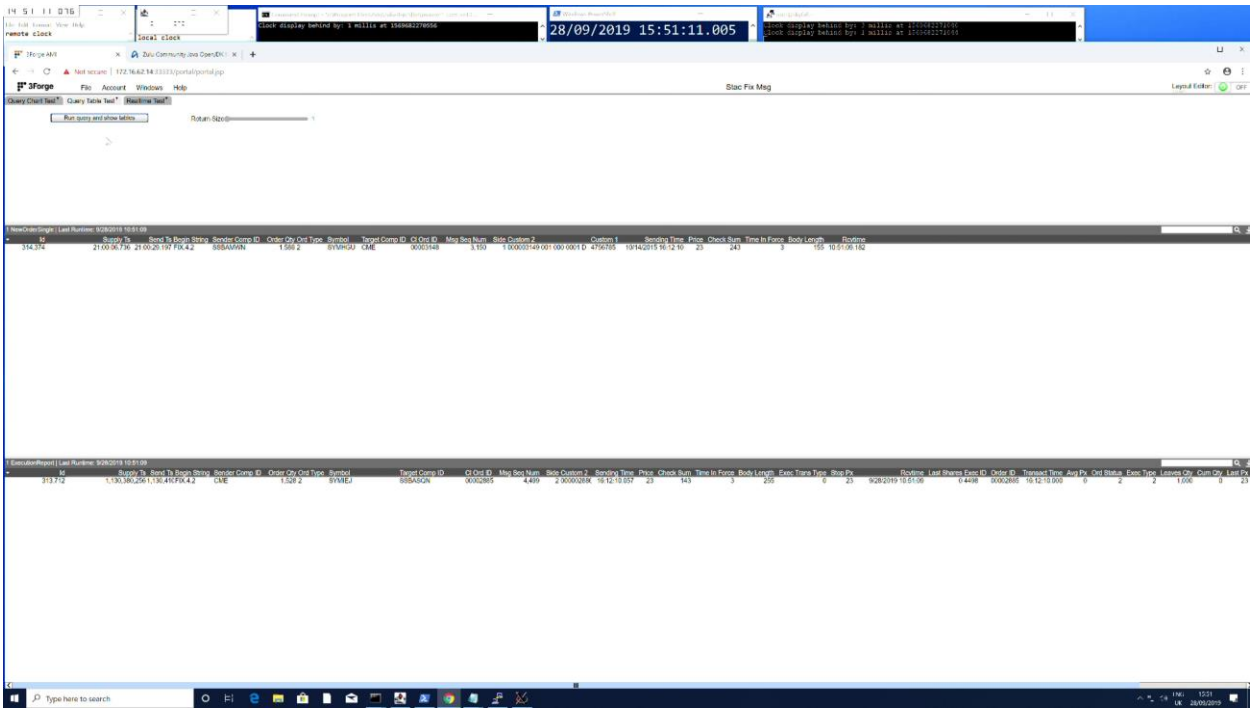

Figure 4

Table 4 contains the latency results. The average difference between query completion time and the time of supply for the most recent record in the query results was 149 milliseconds.

Outer bound of availability latency based on one-off queries during ingest		
Supply Time	**Complete time**	**Difference (ms)**
14:51:00.859	14:51:01.140	**281**
14:51:05.357	14:51:05.475	**118**
14:51:09.159	14:51:09.309	**150**
14:51:13.335	14:51:13.508	**173**
14:51:19.403	14:51:19.609	**206**
14:51:23.784	14:51:23.909	**125**
14:51:28.145	14:51:28.275	**130**
14:51:31.043	14:51:31.177	**134**
14:51:35.018	14:51:35.145	**127**
14:51:39.268	14:51:39.378	**110**
14:51:43.041	14:51:43.145	**104**
14:51:47.501	14:51:47.612	**111**
14:51:51.842	14:51:51.978	**136**
14:51:55.500	14:51:55.646	**146**
14:51:59.471	14:51:59.613	**142**
14:52:03.076	14:52:03.280	**204**
14:52:07.526	14:52:07.647	**121**
14:52:10.001	14:52:10.180	**179**
14:52:12.192	14:52:12.314	**122**
14:52:15.780	14:52:15.947	**167**
	AVERAGE	**149**
(Error less than +/- 50 milliseconds)		

Table 4

8.2.3 Continuous query

The final test involved a query that asked AMI to provide the latest orders as they came into the system. This is a so-called “continuous” query, though the results are typically rendered in frequent batch updates to the screen rather than continuously (just as a video is a sequence of frames). We initiated the query, then started the datastream into the SUT.

In this case, since thousands of orders per second would overwhelm the browser, the 3Forge system appears to grab chunks of records periodically for display. This is similar to how a video divides continuous motion like a propeller into discrete frames. The 3Forge display appears to change the chunk in view about every 150 milliseconds. Figure 5 and Figure 6 demonstrate this (though the table values may not be legible in a printed version of this STAC Report). They are successive frames, showing the moment the screen changed. For each observation taken from this test run, we needed a frame that

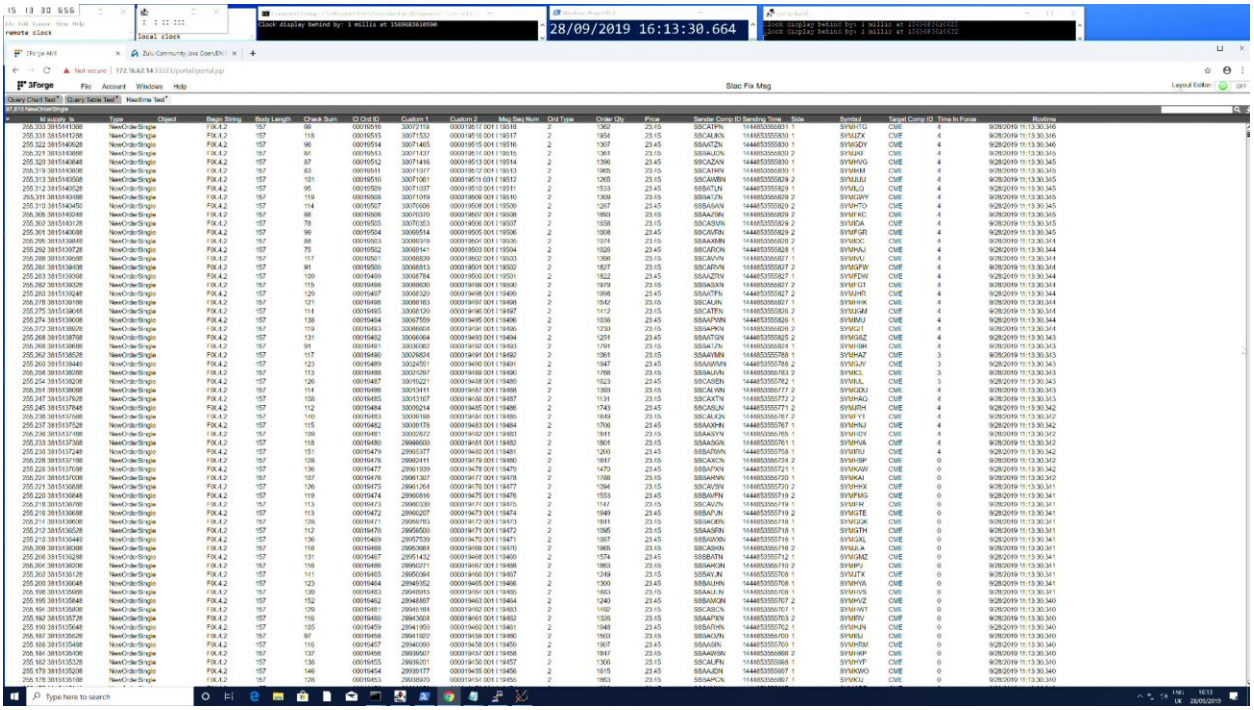

Figure 5

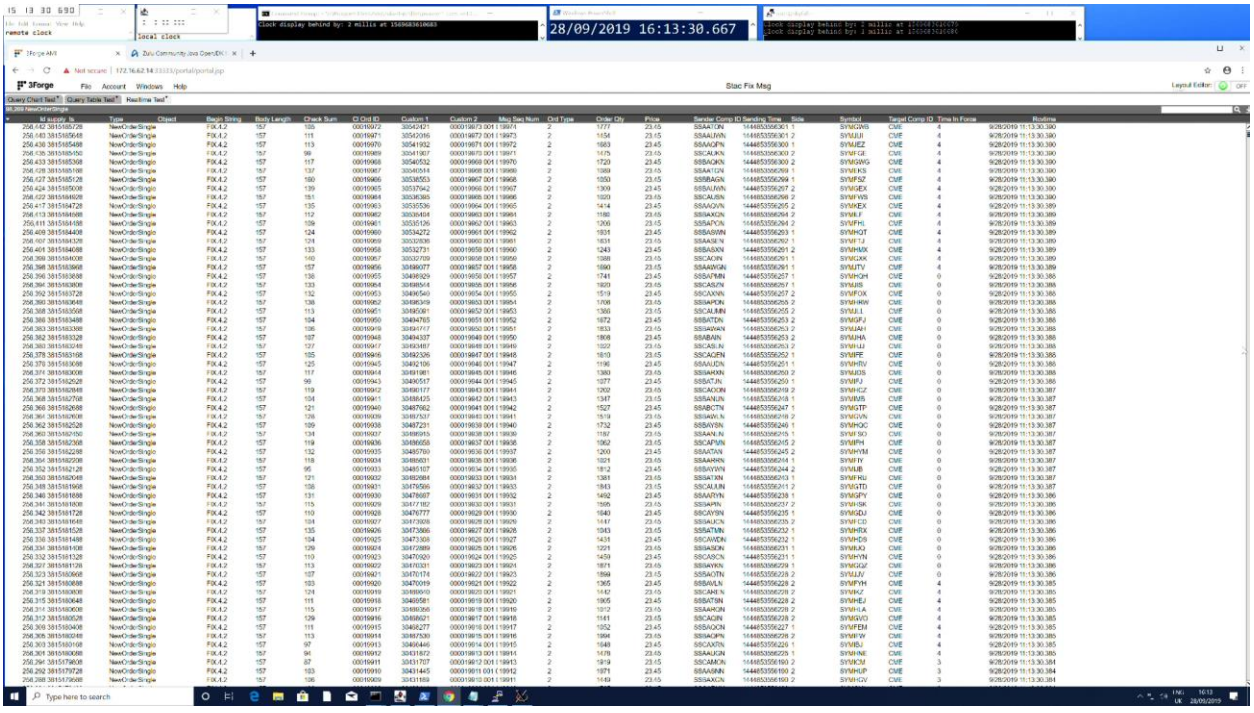

Figure 6

contained new information relative to the previous frame. To take measurements, we arbitrarily chose points throughout the duration of ingest around which we found a frame when the screen refreshed. In that frame we noted the most recent order and looked up its supply time. The results are in Table 5. For the most recent record on display when the screen refreshed, the average delay between supply of the record and its display was 247 milliseconds.

Outer bound of availability latency based on continuous query during 25K msg/sec ingest		
Supply Time	**Receive time**	**Difference (ms)**
15:13:20.144	15:13:20.475	**331**
15:13:20.282	15:13:20.655	**373**
15:13:20.404	15:13:20.822	**418**
15:13:20.507	15:13:20.955	**448**
15:13:33.731	15:13:33.923	**192**
15:13:34.694	15:13:34.890	**196**
15:13:43.822	15:13:44.024	**202**
15:13:45.224	15:13:45.424	**200**
15:13:49.493	15:13:49.690	**197**
15:14:00.727	15:14:00.925	**198**
15:14:03.322	15:14:03.526	**204**
15:14:05.697	15:14:05.892	**195**
15:14:08.903	15:14:09.192	**289**
15:14:12.871	15:14:13.059	**188**
15:14:17.846	15:14:18.026	**180**
15:14:21.066	15:14:21.361	**295**
15:14:24.576	15:14:24.760	**184**
15:14:30.277	15:14:30.462	**185**
15:14:35.804	15:14:36.095	**291**
15:14:39.514	15:14:39.695	**181**
	AVERAGE	**247**
(Error less than +/- 50 milliseconds)		

Table 5

9. Vendor Commentary

3Forge provided the following comments:

> *As outlined in the report, the tests were conducted on a low range, off the shelf server with a single CPU with 8 cores. We would also like to highlight that there is very little load on the desktop while using AMI.*

About STAC

STAC® provides technology research and testing tools that are based upon community-source standards. STAC facilitates the STAC Benchmark™ Council (www.STACresearch.com/council), an organization of leading financial institutions and technology vendors that specifies standard ways to assess technologies used in finance. The Council is active in an expanding range of low-latency, big-compute, and big-data workloads.

STAC helps end-user firms relate the performance of new technologies to that of their existing systems by supplying them with STAC Benchmark reports as well as standards-based STAC Test Harnesses™ for rapid execution of STAC Benchmarks in their own labs. User firms do not disclose their results. Some STAC Benchmark results from vendor-driven projects are made available to the public, while those in the STAC Vault™ are reserved for qualified members of the Council (see www.STACresearch.com/vault).

To be notified when new STAC Reports™ become available, please sign up for free at www.STACresearch.com.