E-Book:

Maximizing the REST API Performance on Akeneo PIM Community Edition

Introduction

Are you now, or considering, using Akeneo PIM as your Product Information Management (PIM) system for a very large product data set? Do you have concerns about the ability to get a large volume of data in or out of the Akeneo PIM using its web-based REST API in a timely manner? Do not be concerned. When configured for performance you can update over 500 products/second and retrieve over 3,000 products/second. So Akeneo PIM can easily handle millions of products.

This article outlines the process I used to generate a test data set, insert it into Akeneo PIM, and then select it all from the PIM, over-and-over again, collecting the experimental data to determine how to configure Akeneo PIM to maximize the performance of its REST API.

Along with the results. It took me nearly a month of data processing to acquire this data. I hope it helps you get the most out of your Akeneo PIM.

Hardware Setup

Akeneo PIM is a: Linux, Apache, MySQL, and PHP (LAMP) application. In an Akeneo hosted platform as a service (PAAS) offering like Flexibility, the host is typically an 8 CPU, 16 GB of memory, 256 GB+ of disk space machine. I'm going to assume here, that Akeneo's software as a service (SAAS) offering, Serenity uses the same setup.

To fully explore the possibilities of performance I decided to use three machines. All with the same motherboard, type of CPU, type of memory, network interface, and disk.

Here are the three machine configurations:

- an 8 CPU, 16 GB of memory
- a 16 CPU, 32 GB of memory
- a 32 CPU, 64 GB of memory

I ran three sets of data on each machine, up to their capabilities, that is.

Next, let's talk software.

Software Setup

The software running on the machines consisted of:

- Ubuntu Linux 22
- Apache, the web server
- Elasticsearch, the search engine
- MySQL, the relational database engine
- PHP FastCGI Process Manager (FPM), the executable engine(s) for the application
- Akeneo PIM Community Edition, the application

These were all installed with their default settings.

Ubuntu installation instructions can be found here: https://ubuntu.com/server/docs

The remaining installation instruction can be found here: https://docs.akeneo.com/latest/install_pim/manual/system_requirements/system_install_ubuntu_2204.html

The PIM's PHP FPM settings are initially set to those used by the Flexibility platform. As you will read later, I had to start changing the infrastructure configurations, especially when I got to the larger machines.

Now let's talk data.

Generating a Random Data Set

I wanted to use a typical large product data set. But what is that? Here's what I decided. I wrote a new node application called node-akeneo-perf.

It's first task was to create a test dataset that I could use repeatedly: upload, then download, i.e., in and out.

One identifier.

I decided to randomly create 1 to 10 attributes of each type, sans File and Image:

- Yes/No
- Date
- Decimal (Number)
- Metric
- Multi Select
- Integer (Number)
- Price
- Simple Select
- Text
- Text Area, up to 2,024 bytes

For the Simple and Multi Selects, 1 to 10 randomly generated attribute options.

One family.

Next, I randomly generated one million products.

This produced 4 data sets:

- attributes.vac, 70151 bytes
- attributeOptions.vac, 255280 bytes
- families.vac, 7928 bytes
- products.vac, 9712813516 bytes

During the experiment, I used these datasets every time. Let's talk about the experiment.

The Experimental Process

How do you increase the performance of a LAMP application like Akeneo PIM?

The first step is by increasing the number of fast CGI engines, in this case PHP-FPM processes. I started with 12 + 8. That is, 12 PHP-FPM processes for the web app and 8 for the REST API. This is typically the setup in Akeneo Flexibility.

Once I reached the 8 process limit used by 8 nearly simultaneous REST API calls which consumed all 8 processes, I changed the configuration strategy to 16 + the number of nearly simultaneous calls to the REST API. For example, for 32 nearly simultaneous REST API calls, 16 + 32, so a total of 48 PHP-FPM processes. I made this adjustment for every run of the experiment.

After the random data sets were created, the experiment consisted of the following steps:

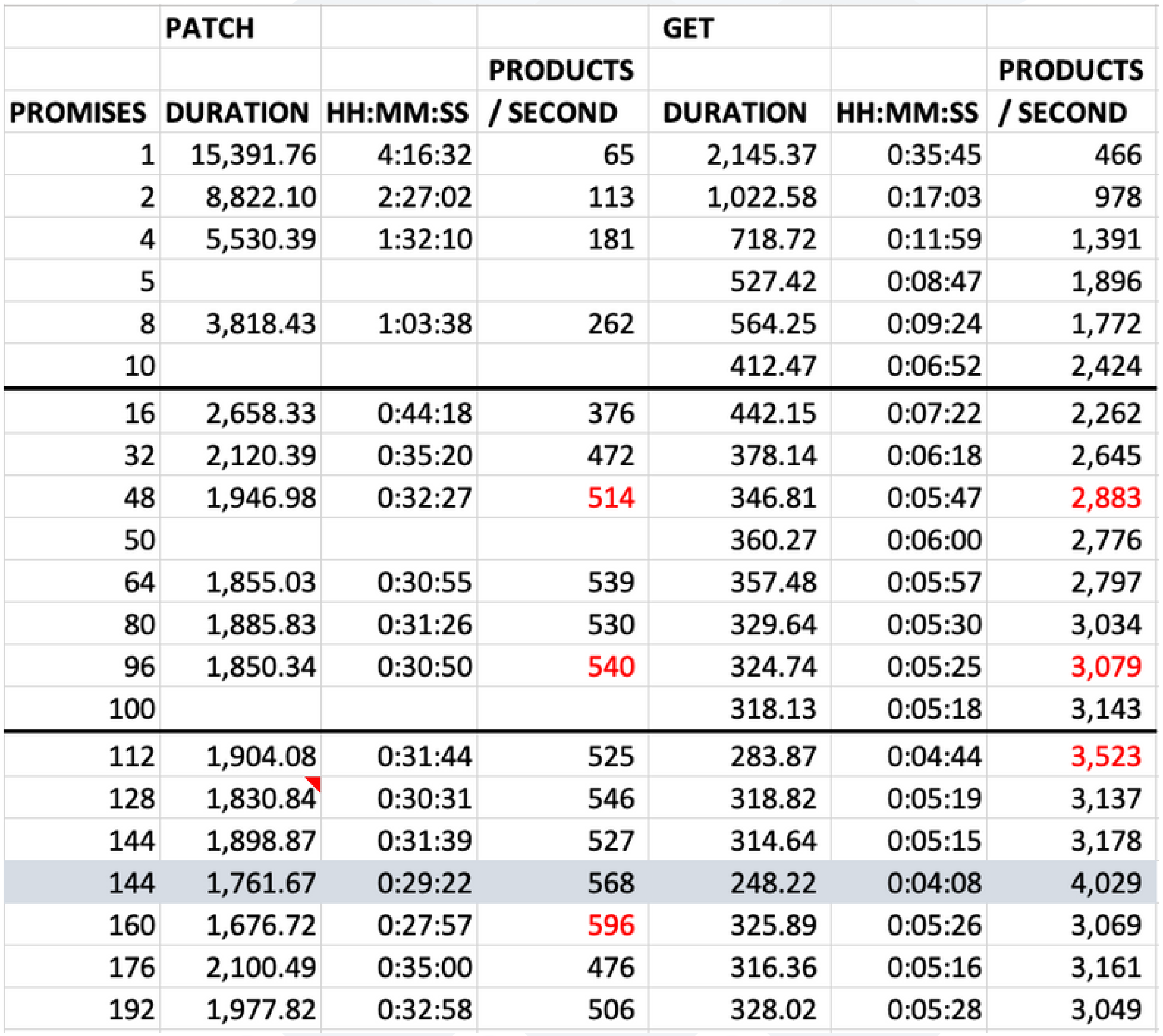
1. Re-create the Akeneo PIM database: php -d memory_limit=4G bin/console pim:installer:db --catalog vendor/akeneo/pim-enterprise-dev/src/Akeneo/Platform/Bundle/InstallerBundle/Resources/fixtures/minimal
2. Create a PIM admin user: php -d memory_limit=4G bin/console pim:user:create admin admin don@donaldbales.com Admin Admin en_US --admin -n
3. Cycle the MySQL database process in order to clear its buffers: sudo service mysql restart
4. Edit the PHP-FPM pool configuration file so the maximum number of children is 16 more than the number of promises used in the run: sudo vi /etc/php/8.0/fpm/pool.d/www.conf, pm.max_children = 20
5. Restart the PHP-FPM processes: sudo service php8.0-fpm restart
6. Restart Apache: sudo service apache2 restart
7. Log into Akeneo PIM and create a new connection. Make sure to set Role to Administrator and Group to IT Support so the REST API will have permissions to write and read products
8. Copy the connection settings to file setenv
9. Set the AKENEO_EXPORT_PATH to the location of our test data set in file setenv
10. Set the AKENEO_PROMISE_LIMIT to the number of promises to be used for the test in file setenv
11. Set the AKENEO_GET_LIMIT to the maximum number of products to GET in the file setenv
12. Import Attributes using node-akeneo_perf's: runlocal -t importAttributes
13. Import Attribute Options using node-akeneo_perf's: runlocal -t importAttributeOptions
14. Import Families using node-akeneo_perf's: runlocal -t importFamilies
15. Import Products using node-akeneo_perf's: runlocal -t importProducts, logging the output for later data collection
16. Set the AKENEO_EXPORT_PATH to another location other than our test data set in file setenv
17. Export Products using node-akeneo_perf's: runlocal -t exportProducts, logging the output for later data collection

Repeat the steps for each desired number of promises.

The steps outline a process of starting with a new empty PIM, with a single user: admin. Then we flush the database cache by restarting the database server. We create a new connection to be used by program node-akeneo-perf to exercise the REST API by creating a randomly generated catalog with randomly generated product data for 1,000,000 products. Then we turn around, and get the data back out, all the time collecting how long it takes, to calculate a products/second rate for REST API PATCHes and GETs.

The Experiment's Results

Here's the data:



	PATCH			GET		
			PRODUCTS			PRODUCTS
PROMISES	DURATION	HH:MM:SS	/ SECOND	DURATION	HH:MM:SS	/ SECOND
1	15,391.76	4:16:32	65	2,145.37	0:35:45	466
2	8,822.10	2:27:02	113	1,022.58	0:17:03	978
4	5,530.39	1:32:10	181	718.72	0:11:59	1,391
5				527.42	0:08:47	1,896
8	3,818.43	1:03:38	262	564.25	0:09:24	1,772
10				412.47	0:06:52	2,424
16	2,658.33	0:44:18	376	442.15	0:07:22	2,262
32	2,120.39	0:35:20	472	378.14	0:06:18	2,645
48	1,946.98	0:32:27	514	346.81	0:05:47	2,883
50				360.27	0:06:00	2,776
64	1,855.03	0:30:55	539	357.48	0:05:57	2,797
80	1,885.83	0:31:26	530	329.64	0:05:30	3,034
96	1,850.34	0:30:50	540	324.74	0:05:25	3,079
100				318.13	0:05:18	3,143
112	1,904.08	0:31:44	525	283.87	0:04:44	3,523
128	1,830.84	0:30:31	546	318.82	0:05:19	3,137
144	1,898.87	0:31:39	527	314.64	0:05:15	3,178
144	1,761.67	0:29:22	568	248.22	0:04:08	4,029
160	1,676.72	0:27:57	596	325.89	0:05:26	3,069
176	2,100.49	0:35:00	476	316.36	0:05:16	3,161
192	1,977.82	0:32:58	506	328.02	0:05:28	3,049

I combined the results for all three machines in one table for calling the REST API to PATCH products and calling the REST API to GET products. I combined them because up to the point of each machine's processing capacity, the results were the same.

The two heavy horizontal lines note when I had to break up the possible starting prefix for SKUs from 10 to 100 queries. The shaded line is a run on the host machine instead of over a 1GB network connection.

What does the data mean?

I interpret the results in the rate of products/second (p/s).

An 8 CPU/16 GB machine was optimally configured with 64 PHP-FPM processes with 48 configured statically for the REST API: it could patch 514 p/s, get 2,883 p/s.

A 16 CPU/32 GB machine was optimally configured with 112 PHP-FPM processes with 96 configured statically for the REST API: it could patch 540 p/s, get 3,079 p/s.

A 32 CPU/64 GB machine was optimally configured with 176 PHP-FPM processes with 160 configured statically for the REST API: it could patch 596 p/s, get 3,523 p/s.
Why not 192? because it becomes i/o bound.

If the node-akeneo-perf ran directly on the host, using localhost as the connection, PATCHes got 7% faster, while GETs 27% faster.

The Promise of Higher Performance

This experimental environment was an ideal setting. A 1GB dedicated network for the connection between the client machine running node-akeneo-perf, and the Akeneo PIM host. Accordingly, the biggest culprit of poor REST API performance, the network, was minimized.

In our ideal setting, Akeneo PIM's rest API was able to PATCH 65 products/second with one connection. And, in turn, GET 466 products/second. In the 40+ years I've worked in this industry, those are better than average numbers.

But are they good enough when you have a large data set? Is it within your business' constraints to wait for 4 1/2 hours to update data in your PIM if you have one million products?

How about getting data out? Is it acceptable to take 30 – 40 minutes the get the data out, every time you send it down stream?

I often here, "it's OK, because we only send delta updates to the PIM." Over time, the data source that is sending deltas will become out-of-sync with the PIM due to network and other errors. It that OK? Probably not. To fix that, you'll need to perform a reconciliation or full update. To do that you'll need to get all the data out, and/or all the data back in. So, speed matters.

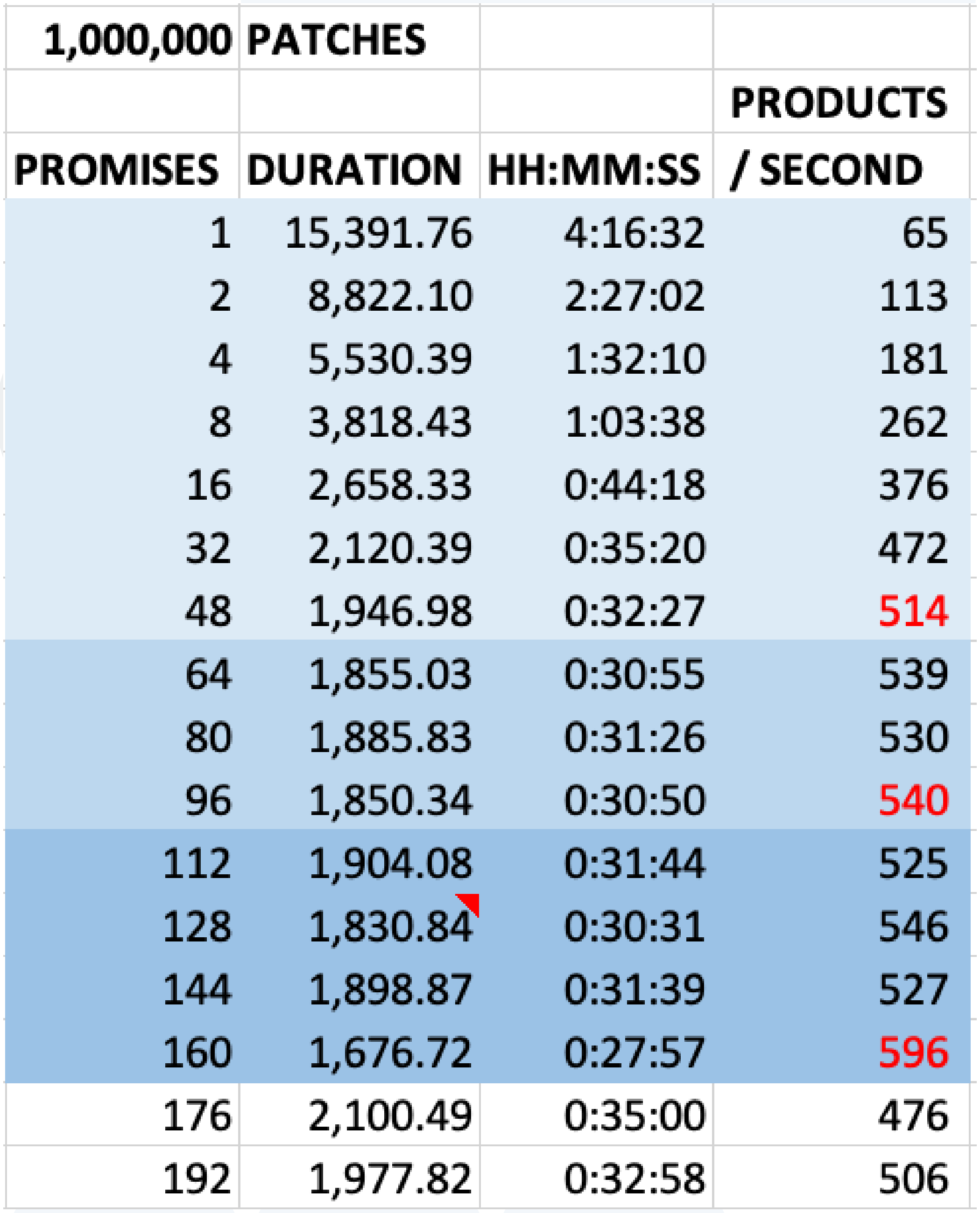
How do we get more data in or out in the same time frame? By PATCHing or GETting more calls against the same endpoints on the REST API at the same time.

The easiest way to do that is to use promises in JavaScript Node. So that is what I've done here.

Next, let's look at inserting the data into the PIM.

PATCHing

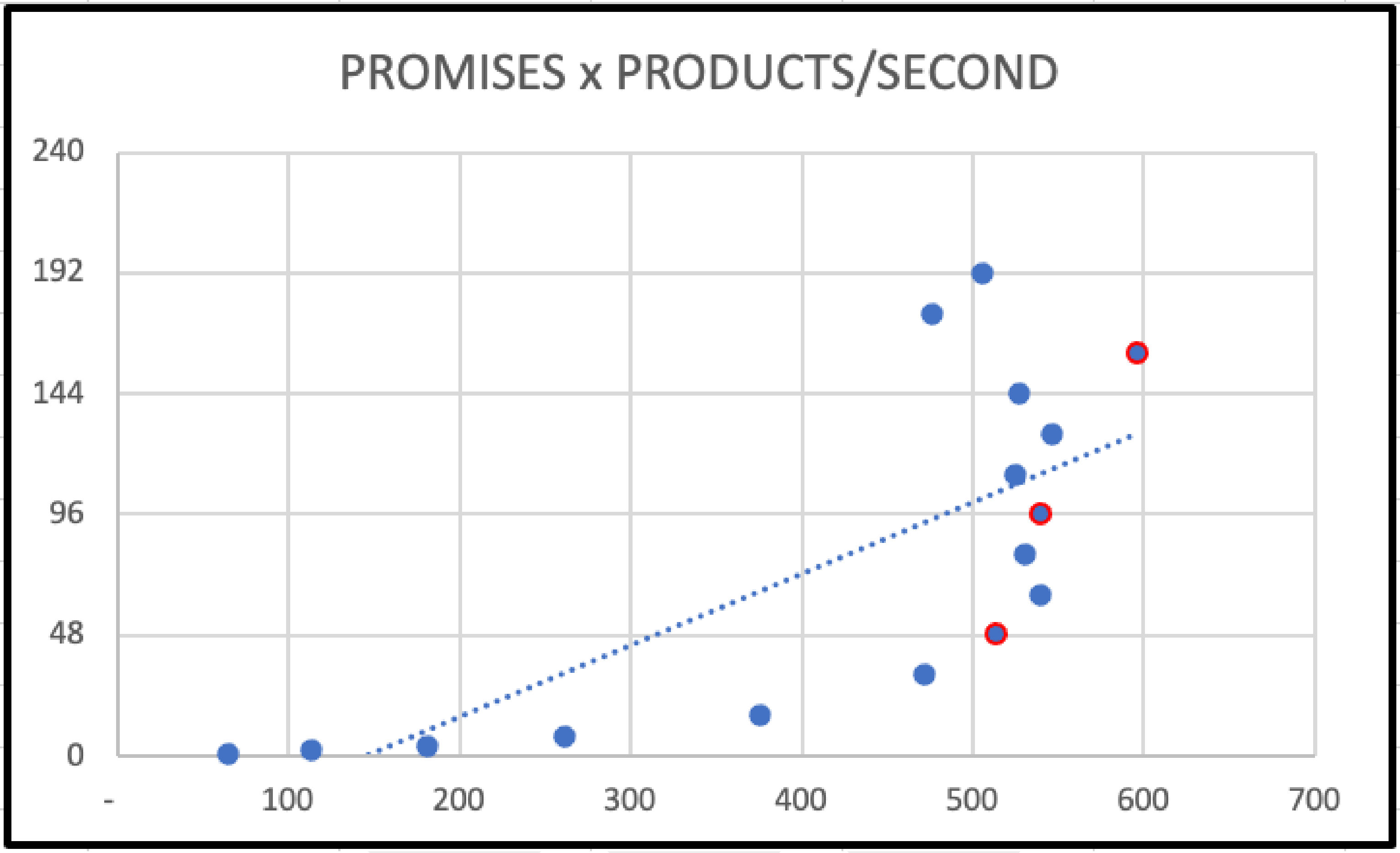
I used the HTTP PATCH endpoint: /api/rest/v1/products, to create the products in Akeneo PIM. As you can see in the following PATCH data, the three machines each had a configuration that was optimal for the number of products that could be created per second; they are highlighted in red.



1,000,000	PATCHES		
			PRODUCTS
PROMISES	DURATION	HH:MM:SS	/ SECOND
1	15,391.76	4:16:32	65
2	8,822.10	2:27:02	113
4	5,530.39	1:32:10	181
8	3,818.43	1:03:38	262
16	2,658.33	0:44:18	376
32	2,120.39	0:35:20	472
48	1,946.98	0:32:27	514
64	1,855.03	0:30:55	539
80	1,885.83	0:31:26	530
96	1,850.34	0:30:50	540
112	1,904.08	0:31:44	525
128	1,830.84	0:30:31	546
144	1,898.87	0:31:39	527
160	1,676.72	0:27:57	596
176	2,100.49	0:35:00	476
192	1,977.82	0:32:58	506

- 8 CPU/16 GB machine: 48 promises (64 PHP-FPM processes)
- 16 CPU/32 GB machine: 96 promises (112 PHP-FPM processes)
- 32 CPU/64 GB machine: 160 promises (176 PHP-FPM processes)

Let’s look at the data graphically:



Here the number of promises (nearly simultaneous) REST API calls is the vertical axis, and the throughput rate in products per second, is the horizontal axis. It’s easy to see that 48 PHP-FPM processes dedicated to the REST API is an optimal setting on an 8 CPU/16 GB memory machine.

Now let’s look at getting the data out.

GETting

I used the HTTP GET endpoint: /api/rest/v1/products, to retrieve all the products out of Akeneo PIM. As you can see in the following GET data, the three machines each had a configuration that was optimal for the number of products that could be created per second; they are highlighted in red.

Breaking the GET into multiple GETs is no trivial process. In my case, I generated SKUs from 10100000 to 10999999. Accordingly, to perform multiple GETs I had to use 10 prefixes, starting with 100, 101, 102, 103 … 109, combined with a STARTS WITH query string:

api/rest/v1/products?pagination_type=search_after&search={\"sku\":[{\"operator\":\"STARTS WITH\",\"value\":\"100\"}]}

That allowed me to use 2 – 10 promises.

Next, I had to use 100 prefixes, start with 1000, 1001, 1002 … 1099, combined with a STARTS WITH query string:

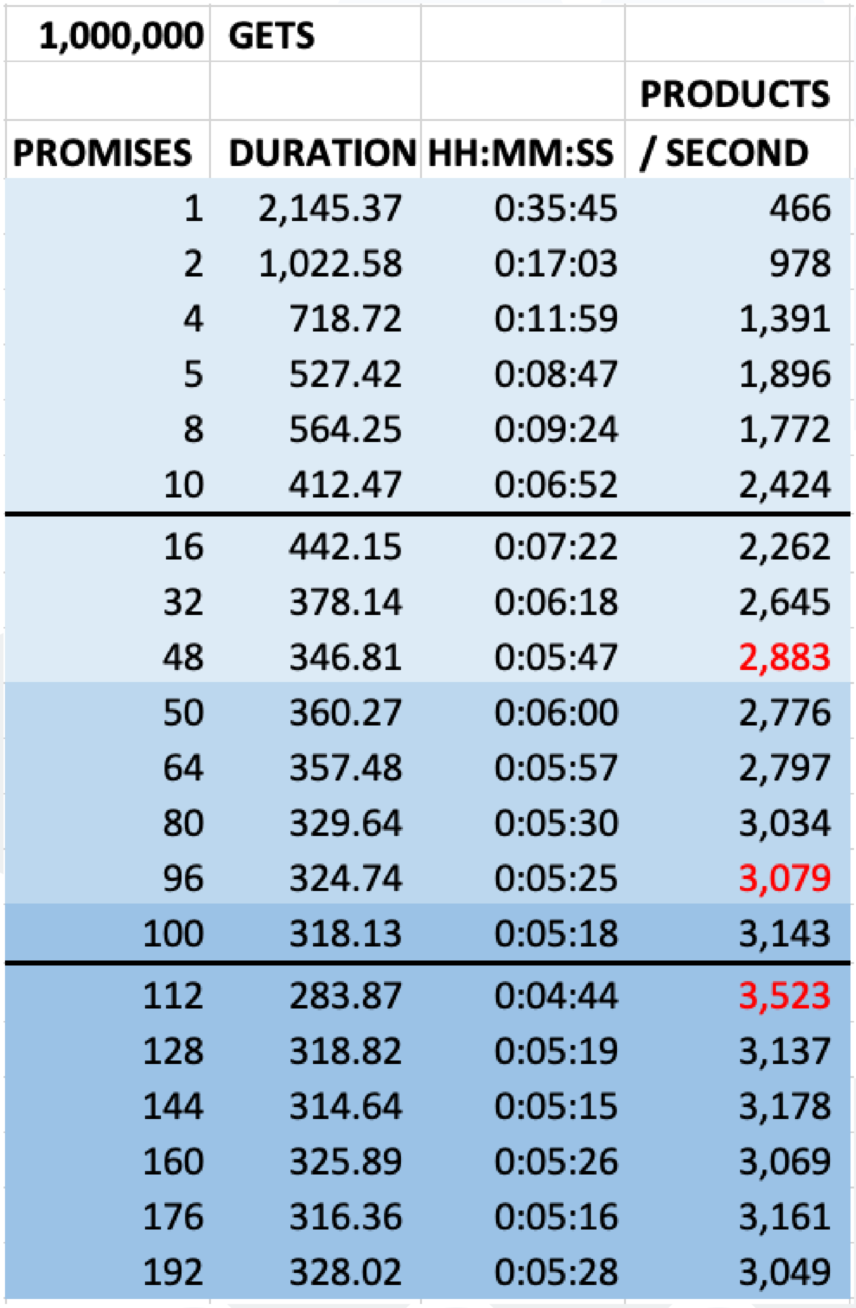
/api/rest/v1/products?pagination_type=search_after&search={\"sku\":[{\"operator\":\"STARTS WITH\",\"value\":\"1000\"}]}

That allowed me to use 16 – 100 promises. The next breakdown would be one thousand, so I did not do that. Instead, the data for 112 to 192 promises is just 100 REST API calls.

It has been my experience that product SKUs typically contain the digits 0 – 9 and characters A – Z.I even encourage my clients to set a regular expression validation defined as:

/[0-9A-Z]+$/

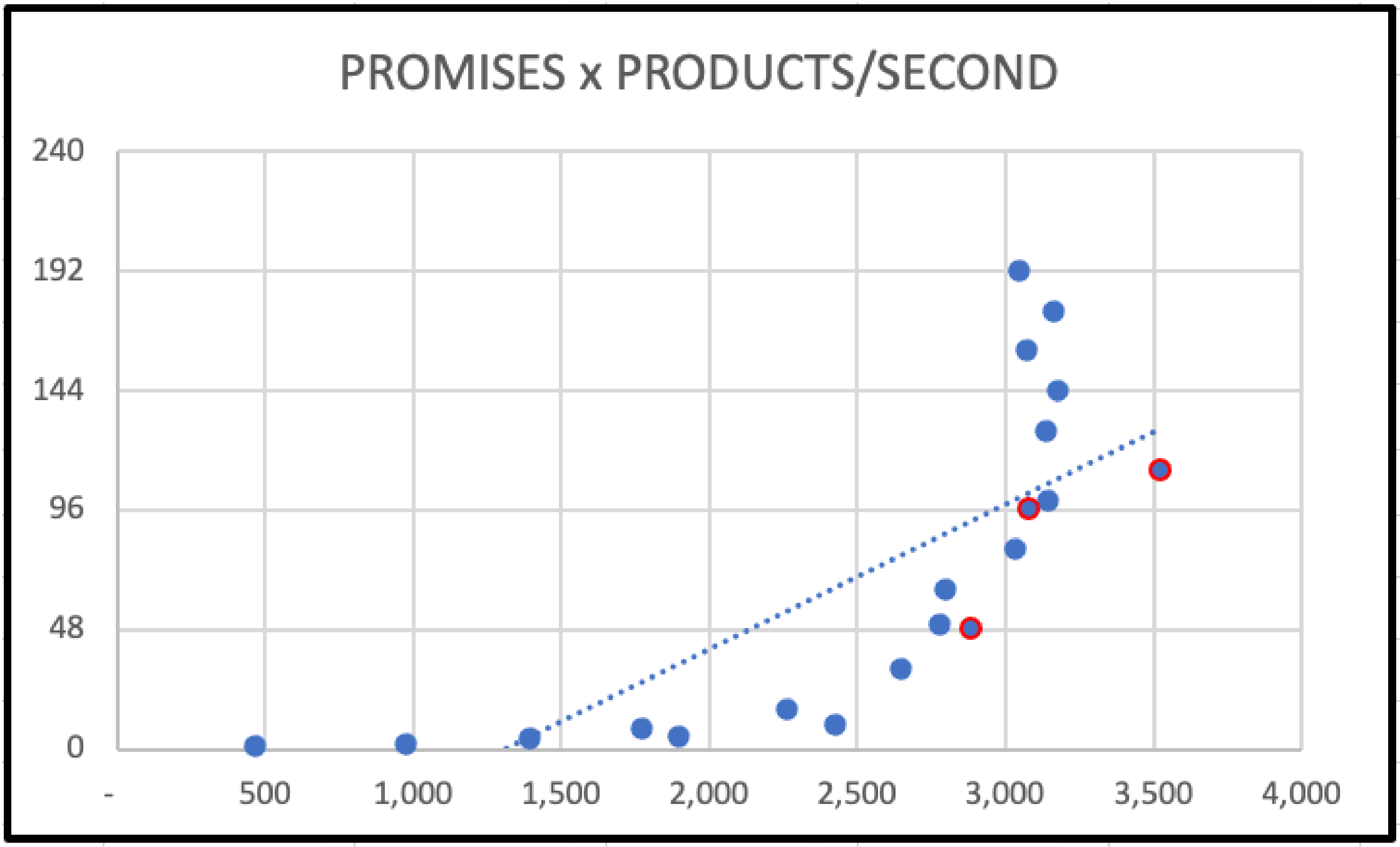
To ensure that is the case. Then I break the GETs into 36 API calls.



1,000,000	GETS		
			PRODUCTS
PROMISES	DURATION	HH:MM:SS	/ SECOND
1	2,145.37	0:35:45	466
2	1,022.58	0:17:03	978
4	718.72	0:11:59	1,391
5	527.42	0:08:47	1,896
8	564.25	0:09:24	1,772
10	412.47	0:06:52	2,424
16	442.15	0:07:22	2,262
32	378.14	0:06:18	2,645
48	346.81	0:05:47	2,883
50	360.27	0:06:00	2,776
64	357.48	0:05:57	2,797
80	329.64	0:05:30	3,034
96	324.74	0:05:25	3,079
100	318.13	0:05:18	3,143
112	283.87	0:04:44	3,523
128	318.82	0:05:19	3,137
144	314.64	0:05:15	3,178
160	325.89	0:05:26	3,069
176	316.36	0:05:16	3,161
192	328.02	0:05:28	3,049

- 8 CPU/16 GB machine: 48 promises (64 PHP-FPM processes)
- 16 CPU/32 GB machine: 96 promises (112 PHP-FPM processes)
- 32 CPU/64 GB machine: 112 promises (128 PHP-FPM processes)

Let's look at the data graphically:



Here the number of promises (nearly simultaneous) REST API calls is the vertical axis, and the throughput rate in products per second, is the horizontal axis. It's easy to see that 48 PHP-FPM processes dedicated to the REST API is an optimal setting on an 8 CPU/16 GB memory machine.

It also shows that there are limits to how much performance you can gain through vertical scaling, that is, adding more CPUs and Memory to a single machine. The 32 CPU/64 GB machines became I/O bound, due to the fact that MySQL and Elasticsearch reside on the same machine. Through other testing, I've found you can get higher throughput by exploding the architecture into two or three machines.

With three machines, the web portion Apache + PHP-FPM engine reside on one host, Elastic search on another, and MySQL on the third. These three machines configured as 8 CPU/16 GB machines provide higher throughput that one 32 CPU/64 GB machine. Accordingly, if you need more throughput than you can get out of one 8 CPU/16 GB machine, I suggest you scale horizontally.

Vertical Scaling Configuration Changes

Earlier, I alluded to the fact that I had to start changing the default configurations to work with more CPUs and Memory, which meant more connections.

Apache

```
sudo vi /etc/apache2/apache2.conf
```

```
Timeout 600
```

```
sudo vi /etc/apache2/mods-available/mpm_event.conf
```

```
<IfModule mpm_event_module>
    StartServers    4
    MinSpareThreads   25
    MaxSpareThreads   75
    ThreadLimit    64
    ThreadsPerChild   25
    MaxRequestWorkers    600
    MaxConnectionsPerChild    0
</IfModule>
```

MySQL

```
sudo vi /etc/mysql/mysql.conf.d/mysqld.cnf
```

```
innodb_buffer_pool_size = 4G
innodb_buffer_pool_instances = 8
key_buffer_size = 128M
max_connections = 256
```

PHP-FPM

```
sudo vi /etc/php/8.0/fpm/pool.d/www.conf

pm = static
pm.max_children = <various>
```

This example program is open-source and available at:

https://github.com/donaldbales/node-akeneo-perf

Conclusion

Yes, you can get a lot more throughput from Akeneo PIM outside its common configuration.

Again, an 8 CPU/16 GB machine was optimally configured with 64 PHP-FPM processes with 48 configured statically for the REST API: it could patch 514 p/s, get 2,883 p/s.

Akeneo PIM is a great Product Information Management (PIM) application and can easily handle well over 1,000,000 products. And its web-based REST API can be configured to handle your most demanding needs.

You can create or update 1,000,000 products in roughly 30 minutes.

You can retrieve 1,000,000 products in roughly 6 minutes.

That's fast.

Do these settings apply to the Enterprise Edition too? Yes.

Now go forth, and multiply, Akeneo PIMs, that is.

Don Bales
Solution Architect, Sitation

An Information/Systems Architect, Business/Systems Analyst, Software Designer/Developer, and Author, Don Bales is fluent in both business and technology speak, performing the analysis, design, and programming of business solutions.

With over thirty five years of experience solving business problems using technology and ample experience in Akeneo PIM installation, implementation, and integration, Bales shares his acquired knowledge through thought leadership publications and consulting. Additionally, Bales is the author of several books on Oracle and Java.

Recent E:Books by Don Bales:

- Using Akeneo PIM Enterprise Edition as a Vendor Portal
- Syncing Data Between Two Akeneo Community Edition PIMs

Other Works by Don Bales:

- Beginning Oracle PL/SQL (second edition)
- Beginning PL/SQL from Novice to Professional
- Java Programming With Oracle JDBC
- JDBC Pocket Reference
- Oracle Application Server 10g