HealthEngine AWS Proof of Concept

Final Report 2020.12.31

Howard Deiner

Table of Contents

[Introduction 2](#_Toc60067275)

[Project Goals 2](#_Toc60067276)

[Project Methodology 3](#_Toc60067277)

[Metrics Being Collected 3](#_Toc60067278)

[Data Visualization Concept 4](#_Toc60067279)

[Introductory Discussions of the Databases Employed for the Experiments 6](#_Toc60067280)

[Postgres 6](#_Toc60067281)

[MySQL 6](#_Toc60067282)

[Apache Cassandra 7](#_Toc60067283)

[Oracle 7](#_Toc60067284)

[MongoDB 9](#_Toc60067285)

[Apache Ignite 9](#_Toc60067286)

[Results and Observations on the Experiments 9](#_Toc60067287)

[Local Experiments for CECacheEngine 9](#_Toc60067288)

[Postgres 10](#_Toc60067289)

[MySQL 12](#_Toc60067290)

[Apache Cassandra 12](#_Toc60067291)

[Oracle 12](#_Toc60067292)

[MongoDB 12](#_Toc60067293)

[Apache Ignite 12](#_Toc60067294)

[AWS Experiments for CECacheEngine 12](#_Toc60067295)

[Postgres 12](#_Toc60067296)

[MySQL 12](#_Toc60067297)

[Apache Cassandra 12](#_Toc60067298)

[Oracle 13](#_Toc60067299)

[MongoDB 13](#_Toc60067300)

[Apache Ignite 13](#_Toc60067301)

[Local Experiments for Large Data 13](#_Toc60067302)

[Postgres 13](#_Toc60067303)

[MySQL 13](#_Toc60067304)

[Apache Cassandra 13](#_Toc60067305)

[Oracle 13](#_Toc60067306)

[MongoDB 13](#_Toc60067307)

[Apache Ignite 13](#_Toc60067308)

[AWS Experiments for Large Data 13](#_Toc60067309)

[Postgres 13](#_Toc60067310)

[MySQL 13](#_Toc60067311)

[Apache Cassandra 13](#_Toc60067312)

[Oracle 13](#_Toc60067313)

[MongoDB 13](#_Toc60067314)

[Apache Ignite 13](#_Toc60067315)

## Introduction

The HealthEngineAWSPOC project started in the second half of October 2020. It will conclude by the end of 2020.

## Project Goals

The project has several goals in terms of what we want to accomplish.

* The use of repeatable scientific experiment to provide metrics that measure performance of database related tasks, so as to aid in the selection of appropriate database engines suited for the role they are intended to at an architectural level.
* Demonstrate the use of Docker containerization to run desktop development environments, to allow quick and repeatable development environments that don’t require scheduling time with shared resources, such as databases.
* Since isolated development environments provide the ultimate in test isolation, demonstrate that the environments can be reliably scaled into AWS environments on demand, so that scheduling of databases with specific programmatic states is a thing of the past through the use of Infrastructure as Code (IaC).
* Because of the nature of innovation does not lend itself to estimation at all, we chose to use a very small team of self-organized individuals focused on the work to be done and working to meet the goals to the time to produce outcomes desired.
* Demonstrate the concept of security in the face of lack of trust. Even through the data used for the project is not sensitive, we protect the data used to load the databases with at rest and over the line with “better than NSA” security. As we live in the 21st century, this becomes more and more of a problem, and it doesn’t get solved with merely the use of standardized tools and processes.
* Demonstrate the concept of collaboration through code, so that anyone wishing to monitor the progress of the project has the means to visit the Git Repository and see what’s happening and how to reproduce the results, rather than call for meetings to convey this information.
* But ultimately, the goal is to gain experience in the six databases used (Postgres, MySQL, Cassandra, Oracle, MongoDB, and Ignite) and learn the fitness for use for our applications, considering
  + Speed and capacity
  + Ease and readability for automated setup, data loading, monitoring, and maintenance of the database itself
  + Ease and readability in use for the client application (both programmatic JDBC based and ad hoc SQL query based)
  + Ability to overcome the CAP Theorem (<https://en.wikipedia.org/wiki/CAP_theorem>). In July 2000, EricBrewer gave the keynote speech at the ACM Symposium on the Principles of Distributed Computing, where he stated that it was impossible for a distributed data store to simultaneously satisfy more than two of the three qualities that we want distributed large-scale databases to satisfy for us:
    - Consistency (Every read receives the most recent write or an error)
    - Availability (Every request always receives a non-error response)
    - Partition Tolerance (No set of failures less than total network failure is allowed to cause the system to respond incorrectly)
* Perhaps W. Edwards Deming said it best in “The Deming of America”, where he states “Experience teaches nothing. In fact there is no experience to record without theory…. Without theory there is no learning… And that is their downfall. People copy examples and then they wonder what is the trouble. They look at examples and without theory they learn nothing.” (“The Deming of America” is a 1991 interview of Dr. Deming by Priscilla Petty – see <http://priscillapetty.com/page36/page36.html>). This is the reason that this project is premised on taking a scientific view, generating reproducible metrics, and seeing if the database we think is the best actually has the right stuff. Otherwise, we simply repeat what other organizations do, believe vendors, or just make arbitrary decisions because we have the largest salary in the room.

## Project Methodology

* The Git Repository can be found at <https://github.com/ActiveHealth/HealthEngineAWSPOC>
* The project structure is generally organized by experiment.
* These experiments are the product of six database engines under scrutiny crossed by “local” versus “AWS” environments.
* There are standardized steps for each experiment.
  + A startup step, where the environment is created (either by docker-compose for local experiments or terraform for AWS experiments)
  + A populate step, where the database is given a schema, data is loaded, and data is checked to ensure its arrival.
  + A startup application step, where CECacheEngine is built into an image and that image is used to create a docker container that can mesh with the database already present in that environment.
  + A shutdown step, where the environment is decommissioned.
* There are also two modes that each experiment is run
  + The normal mode, where a small dataset is used to validate CECacheEngine application suitability
  + A large data mode, where the "Complete 2019 Program Year Open Payments Dataset" from the Center for Medicare & Medicaid Services is used. See https://www.cms.gov/OpenPayments/Explore-the-Data/Dataset-Downloads for details. In total, there is over 6GB in this dataset. We run experiments for 1M, 3M, and 9M row samples from that dataset to look at performance trends.

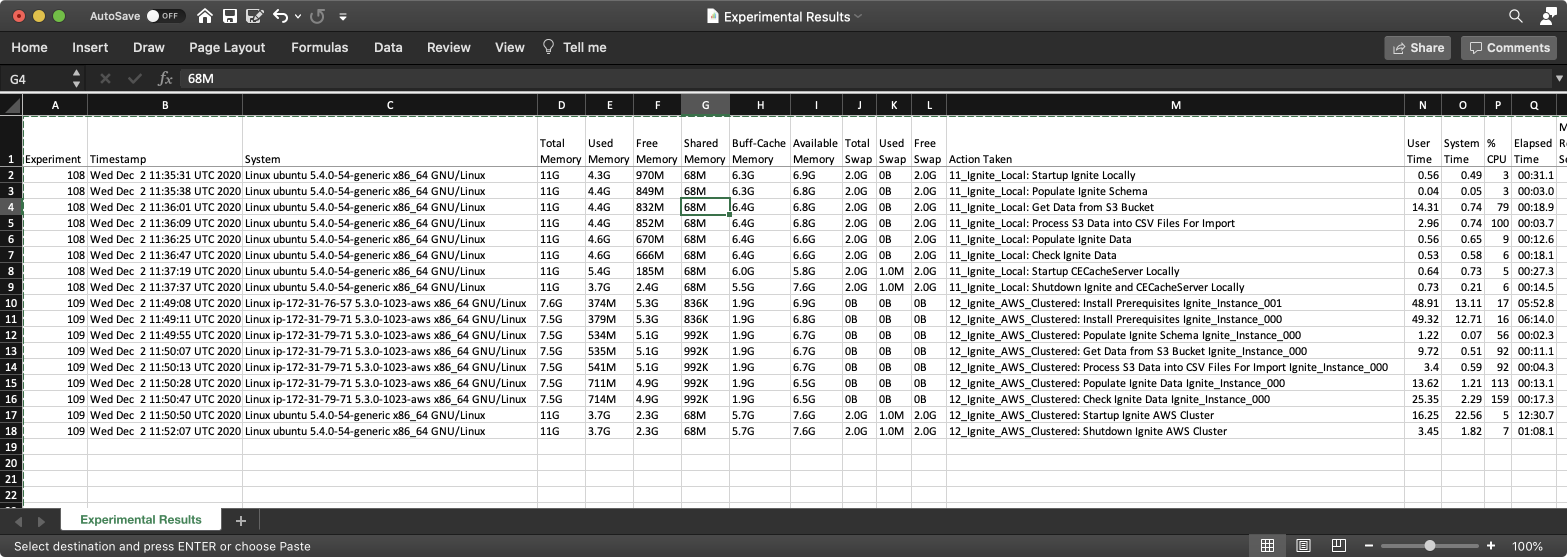
## Metrics Being Collected

* Each step of each experiment collects the following metrics in an S3 CSV:
  + Experiment number (a monotonically increasing identifier that uniquely identifies an experiment being run)
  + Timestamp (UTC of when the metrics were collected)
  + System (uname report of the linux kernel name, the network node hostname, the linux kernel release, the machine hardware name, and the operating system)
  + Total Memory
  + Used Memory
  + Free Memory
  + Shared Memory
  + Buff-Cache Memory
  + Available Memory
  + Total Swap
  + Used Swap
  + Free Swap
  + Action Taken (this is the experiment name followed by a colon, followed by phrases of the major event that is measured, such as “Get Data from S3 bucket”, “Populate Schema”, “Process S3 Data into CSV Files for Import”, etc.)
  + User Time
  + System Time
  + % CPU
  + Elapsed Time
  + Max Resident Set Size
  + Avg Resident Set Size
  + Major Page Faults
  + Minor Page Faults
  + Voluntary Context Switches
  + Involuntary Context Switches
  + Swaps
  + File Input
  + File Output
  + Socket Sent
  + Socket Received
  + Signals Delivered
  + Pagesize

## Data Visualization Concept

The following was run this morning and is a concept of what the full results might look like. We will compare the metrics collected on experiments 11 and 12, which test the Apache Ignite database (which has in-memory, as well as ACID transactions, and works in clusters for both performance and reliability benefits). It is comparing Desktop versus AWS performance of the same database and data, which is not the total deliverables for the project and is only preliminary. Apache Ignite on AWS is being run on two EC2 instances, with each instance run on an m5.large machine ($0.096/hour, 2 vCPU, 10 ECU, 8 GiB memory, 30 GiB of io1 500 iops EBS).

The raw data:



This is then culled into something more manageable for now:

Graphical user interface, table

Description automatically generated

And we can have some initial visualization of the results:

Graphical user interface

Description automatically generated

## Introductory Discussions of the Databases Employed for the Experiments

### Postgres

PostgreSQL, also known as Postgres, is a free and open-source relational database management system (RDBMS) emphasizing extensibility and SQL compliance. It was originally named POSTGRES, referring to its origins as a successor to the Ingres database developed at the University of California, Berkeley. In 1996, the project was renamed to PostgreSQL to reflect its support for SQL. After a review in 2007, the development team decided to keep the name PostgreSQL and the alias Postgres.

We use the official supported postgres:latest image for the Docker based local experiments, and the AWS supported Aurora Clustered environment for the AWS experiments. Postgres does not need to be installed in these experiments, either for server or clients, to keep running the experiments easier.

### MySQL

MySQL is an open-source relational database management system. Its name is a combination of "My", the name of co-founder Michael Widenius's daughter, and "SQL", the abbreviation for Structured Query Language. A relational database organizes data into one or more data tables in which data types may be related to each other; these relations help structure the data. SQL is a language programmers use to create, modify and extract data from the relational database, as well as control user access to the database. In addition to relational databases and SQL, an RDBMS like MySQL works with an operating system to implement a relational database in a computer's storage system, manages users, allows for network access and facilitates testing database integrity and creation of backups.

MySQL is free and open-source software under the terms of the GNU General Public License, and is also available under a variety of proprietary licenses. MySQL was owned and sponsored by the Swedish company MySQL AB, which was bought by Sun Microsystems (now Oracle Corporation). In 2010, when Oracle acquired Sun, Widenius forked the open-source MySQL project to create MariaDB.

MySQL has stand-alone clients that allow users to interact directly with a MySQL database using SQL, but more often MySQL is used with other programs to implement applications that need relational database capability. MySQL is a component of the LAMP web application software stack (and others), which is an acronym for Linux, Apache, MySQL, Perl/PHP/Python. MySQL is used by many database-driven web applications, including Drupal, Joomla, phpBB, and WordPress. MySQL is also used by many popular websites, including Facebook, Flickr, MediaWiki, Twitter, and YouTube..

We use the official supported mysql:latest image for the Docker based local experiments, and the AWS supported Aurora Clustered environment for the AWS experiments. MySQL does not need to be installed in these experiments, either for server or clients, to keep running the experiments easier.

### Apache Cassandra

Apache Cassandra is a free and open source, distributed, wide column store, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra offers robust support for clusters spanning multiple datacenters, with asynchronous masterless replication allowing low latency operations for all clients. Cassandra offers the distribution design of Amazon DynamoDB with the data model of Google's Bigtable.

Avinash Lakshman, one of the authors of Amazon's Dynamo, and Prashant Malik initially developed Cassandra at Facebook to power the Facebook inbox search feature. Facebook released Cassandra as an open-source project on Google code in July 2008. In March 2009 it became an Apache Incubator project. On February 17, 2010 it graduated to a top-level project.

Facebook developers named their database after the Trojan mythological prophet Cassandra, with classical allusions to a curse on an oracle.

We use the official supported cassandra:3.11.8 image for the Docker based local experiments, and the latest apt-get supported Cassandra for AWS experiments run on EC2 instances. Cassandra does not need to be installed in these experiments, either for server or clients, to keep running the experiments easier.

### Oracle

The Oracle Corporation is an American multinational computer technology corporation headquartered in Redwood Shores, California. The company sells database software and technology, cloud engineered systems, and enterprise software products—particularly its own brands of database management systems. In 2019, Oracle was the second-largest software company by revenue and market capitalization. The company also develops and builds tools for database development and systems of middle-tier software, enterprise resource planning (ERP) software, Human Capital Management (HCM) software, customer relationship management (CRM) software, and supply chain management (SCM) software.

Larry Ellison co-founded Oracle Corporation in 1977 with Bob Miner and Ed Oates under the name Software Development Laboratories (SDL). Ellison took inspiration from the 1970 paper written by Edgar F. Codd on relational database management systems (RDBMS) named "A Relational Model of Data for Large Shared Data Banks." He heard about the IBM System R database from an article in the IBM Research Journal provided by Oates. Ellison wanted to make Oracle's product compatible with System R, but failed to do so as IBM kept the error codes for their DBMS a secret. SDL changed its name to Relational Software, Inc (RSI) in 1979, then again to Oracle Systems Corporation in 1983, to align itself more closely with its flagship product Oracle Database. At this stage Bob Miner served as the company's senior programmer. On March 12, 1986, the company had its initial public offering.

In 1995, Oracle Systems Corporation changed its name to Oracle Corporation, officially named Oracle, but sometimes referred to as Oracle Corporation, the name of the holding company. Part of Oracle Corporation's early success arose from using the C programming language to implement its products. This eased porting to different operating systems most of which support C.

It's been a long time since 1977. Today, Oracle revenue has risen to about $10B annually, with revenue of over $2B annually. Larry Ellison has a personal net worth of almost $76B, according to Forbes. Unhappily, the company has not made many changes which keep up with the times and has seduced its users with wonderful features to stay relevant in a manner that is both closed and with a huge barrier to switch. We are reaching the point where our company's future becomes an extension of Larry's largesse. Should Larry decide that he would like to fund his hobby in souped up America's Cup high tech catamarans by tripling license fees, we would have no choice except to pay it.

So, why is this experiment in here, you ask? It should be a slam dunk to get HealthEngine working in the cloud with this. It also provides a baseline for performance against other implementations, and a baseline on the economics of moving away from Oracle at this time. Furthermore, as far as risk mitigation goes, this should be a very safe move to get into the cloud. And if the experiments show that Oracle best satisfies the project goals, we definitely should consider not switching our current Oracle out of the architecture for the next HealthEngine platform.

We use the official supported store/oracle/database-enterprise:12.2.0.1 image for the Docker based local experiments, and the latest Oracle supported version 12.2.0.1 software for AWS experiments run on EC2 instances. Oracle does not need to be installed in these experiments, either for server or clients, to keep running the experiments easier.

### MongoDB

MongoDB is a cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas. MongoDB is developed by MongoDB Inc. and licensed under the Server Side Public License (SSPL).

We use the official supported mongo:latest image for the Docker based local experiments, and the latest apt-get supported mongodb-org versions for AWS experiments run on EC2 instances. MongoDB does not need to be installed in these experiments, either for server or clients, to keep running the experiments easier.

### Apache Ignite

Apache Ignite is an open-source distributed database (without rolling upgrade), caching and processing platform designed to store and compute on large volumes of data across a cluster of nodes.

Ignite was open sourced by GridGain Systems in late 2014 and accepted in the Apache Incubator program that same year. The Ignite project graduated on September 18, 2015.

Apache Ignite's database utilizes RAM as the default storage and processing tier, thus, belonging to the class of in-memory computing platforms. The disk tier is optional but, once enabled, will hold the full data set whereas the memory tier will cache full or partial data set depending on its capacity.

Regardless of the API used, data in Ignite is stored in the form of key-value pairs. The database component scales horizontally, distributing key-value pairs across the cluster in such a way that every node owns a portion of the overall data set. Data is rebalanced automatically whenever a node is added to or removed from the cluster.

On top of its distributed foundation, Apache Ignite supports a variety of APIs including JCache-compliant key-value APIs, ANSI-99 SQL with joins, ACID transactions, as well as MapReduce like computations.

Apache Ignite cluster can be deployed on-premise on a commodity hardware, in the cloud (e.g. Microsoft Azure, AWS, Google Compute Engine) or in a containerized and provisioning environments such as Kubernetes, Docker, Apache Mesos, VMWare.

We use the official supported apacheignite/ignite image for the Docker based local experiments, and the latest Apache Ignite supported version 2.9.0 for AWS experiments run on EC2 instances. Apache Ignite does not need to be installed in these experiments, either for server or clients, to keep running the experiments easier.

## Results and Observations on the Experiments

I have decided that the primary metric to speak to will be elapsed time. After all, it’s the metric we are most likely to be judged on, “How come the system is so slow?”

### Local Experiments for CECacheEngine

#### Postgres

Experiment 171

Text

Description automatically generated

Chart, treemap chart

Description automatically generated

As you can see, the three things that dominate this chart are:

1. Startup CECacheServer Locally (time it takes for CE to start up in Docker and read in the tables from the database).
2. Get Data from S3 Bucket (time it takes to download data is from S3, decrypt it, and make ready for loading into the database).
3. Shutdown Postgres and CECacheServer Locally (time it takes when CE is being shutdown in Docker and local database files are destroyed).

If we take those three variables out of the equation, we are left with this.

Chart, treemap chart

Description automatically generated

Now, the five things that dominate the chart are:

1. Check Postgres Data (time it takes psql to process a SELECT \* LIMIT 2 and SELECT COUNT(\*) on each of ten tables of the CE Data).
2. Process S3 Data into CSV Files for Import (time when time for the sed dominated bash scripts to clean up the input data into a form that psql can bulk copy the csv files into their tables).
3. Populate Postgres Data (time it takes for psql to bulk copy the csv files into their tables).
4. Startup Postgres Locally (time it for starting Postgres in Docker).
5. Populate Postgres Schema (time it takes for Liquibase to populate the schema in the database for CE to use).

If we take those five variables out of the equation, we are left with this.

Shape

Description automatically generated with medium confidence

This is the time is takes each table in CECacheServer to load. It is basically proportional to the size of each table. It is also just the noise in the lower right hand side of the smallest area in the original chart for this experiment, several orders of magnitude smaller than the larger effects on elapsed time.

#### MySQL

#### Apache Cassandra

#### Oracle

#### MongoDB

#### Apache Ignite

### AWS Experiments for CECacheEngine

#### Postgres

Experiment 176

Text

Description automatically generated

Chart, treemap chart

Description automatically generated

As you can see, the three things that dominate this chart are:

1. Startup PostgreSQL AWS Cluster (time it takes for AWS to create the AWS resources for Aurora Clustered Postgres through Terraform and be ready for client connections and work).
2. Shutdown PostgreSQL AWS Cluster (time it takes for AWS to destroy the AWS resources for Aurora Clustered Postgres through Terraform).
3. Get Data from S3 Bucket (time it takes to download data is from S3, decrypt it, and make ready for loading into the database).

If we take those three variables out of the equation, we are left with this.

Chart, treemap chart

Description automatically generated

The largest of these items is over 2 orders of magnitude smaller than the largest items that we removed from this chart, so we can see what’s happening other than AWS resource work. What’s left is

1. Check PostgreSQL Data postgresql-aurora-clustered (time it takes psql to process a SELECT \* LIMIT 2 and SELECT COUNT(\*) on each of ten tables of the CE Data).
2. Populate PostgreSQL Data postgresql-aurora-clustered (time it takes for psql to bulk copy the csv files into their tables).
3. Populate Postgres Schema postgresql-aurora-clustered (time it takes for Liquibase to populate the schema in the database for CE to use).
4. Startup Postgres Locally (time it takes for Docker to bring up local Postgres container, so we can use a local psql client in the container, rather than installing Postgres locally to gain use of a client)
5. Process S3 Data into CSV Files for Import postgresql-aurora-clustered (time when time for the sed dominated bash scripts to clean up the input data into a form that psql can bulk copy the csv files into their tables).
6. Shutdown Postgres Locally (time it for shutting down the local Postgres Docker container).

There are some notable things in this experiment:

1. I am actually violently opposed to using the AWS RDS Aurora Service. I included it because influencers were singing its virtues. But regardless of its performance, use of this AWS service is a bad idea because:
   1. We become dependent on AWS to manage our instance. Any changes need to become a ticket item for managed services at Amazon to fulfill. Need more memory? “Open a ticket.” More disk needed? “Open a ticket.” Don’t like this year’s revised costs? “We have decided to triple your rates for RDS Aurora starting tomorrow, because Jeff has a big alimony payment coming up. You don’t like that? Tough. Pay me or we cut you off.” And we would have no choice.
   2. Because of a), we don’t have access to the server that Postgres is running on. We only have access to the port that gives us access to the database. That might be fine for an IT shop, but for things like bulk loading of data, fixing infrastructure problems while the server is running, and the like, we are divorced from the machine. We become users instead of developers.
   3. If Postgres seems to be a good choice, we should refactor the Terraform code from this project to use EC2 instances to install, configure, and use Postgres ourselves. If we then decide to change cloud providers for whatever reason, it would simply be a matter of small changes in our Terraform code to target another cloud provider’s equivalent of EC2 instances.

#### MySQL

#### Apache Cassandra

#### Oracle

#### MongoDB

#### Apache Ignite

### Local Experiments for Large Data

#### Postgres

Experiments 217, 219, 220

Text, letter

Description automatically generated

Chart, treemap chart

Description automatically generated

As you can see, the Get Data from S3 Bucket (time it takes to download data is from S3, decrypt it, and make ready for loading into the database) dominates the times. Even though we download the exact same data and do the same processing, this has an average of 208 seconds ± 4.45 seconds. The elapsed time value is slightly non-deterministic due to network traffic, etc. Let’s remove that from the chart to get a better sense of what’s going on.

Chart, treemap chart

Description automatically generated

Ah. That’s better. Now, we see that the times look approximately between experiments for their size (grey is 3 times larger than orange, which is about three times bigger than blue).

When we look at the populate times (the largest component), we see confirm that the elapsed time is approximately linear.

Chart, line chart

Description automatically generated

The check data (SQL SELECT) times (the second predominating metric) is also quite linear over the batch sizes.

Chart, line chart

Description automatically generated

This all seems satisfactory and will serve well as a good baseline to work against.

#### MySQL

#### Apache Cassandra

#### Oracle

#### MongoDB

#### Apache Ignite

### AWS Experiments for Large Data

#### Postgres

Experiments 233, 234, 236

Text

Description automatically generated

Chart, treemap chart

Description automatically generated

So, this doesn’t look like what I was expecting. The colored areas should be in approximately 9:3:1 proportions. Let’s remove the Startup AWS Cluster, Shutdown AWS Cluster, Get Data from S3 Bucket, Startup Postgres Locally, and Shutdown Postgres Locally numbers. Those should all be approximately the same across the populations and needed for environment establishment. Now, we get the following.

A picture containing graphical user interface

Description automatically generated

That definitely has the right proportions.

There something notable in this experiment. If we compare the timings for this experiment against the local large data experiment for Postgres (experiments 217, 219, 220), something weird is going on. We see a penalty for using AWS, or perhaps the AWS RDS Aurora Service. I define this penalty as the ratio (expressed as a percentage) of the difference in time between the local versus AWS elapsed time for the operation and the local time for that operation.

Table

Description automatically generated with medium confidence

The results are puzzling. For example, some operations that have nothing to do with AWS (startup and shutdown of local docker container for Postgres for use as a client) show a statistically significant increase in time. Perhaps that is because of increased memory usage on the test machine? Then, we see that the schema creation time takes almost three times longer to do on the AWS setup. Why that is that so slow compared to the check Postgres data operation, which has only about a 40% penalty? Perhaps it’s the JDBC connection that Liquibase uses to access the database remotely? And, for the thing that means the most, the AWS operation for data loading uniformly requires almost 2 orders of magnitude more time than the local operation. That is using the psql native client! We’ll have to keep an eye on these numbers in future databases we compare.

#### MySQL

#### Apache Cassandra

#### Oracle

#### MongoDB

#### Apache Ignite