HealthEngine AWS Proof of Concept

Final Report 2020.12.31

Howard Deiner

Table of Contents

[Introduction 2](#_Toc60662694)

[Project Goals 2](#_Toc60662695)

[Project Methodology 3](#_Toc60662696)

[Metrics Being Collected 4](#_Toc60662697)

[Data Visualization Concept 5](#_Toc60662698)

[Introductory Discussions of the Databases Employed for the Experiments 7](#_Toc60662699)

[Postgres 7](#_Toc60662700)

[MySQL 7](#_Toc60662701)

[Apache Cassandra 8](#_Toc60662702)

[Oracle 8](#_Toc60662703)

[MongoDB 9](#_Toc60662704)

[Apache Ignite 9](#_Toc60662705)

[Results and Observations on the Experiments 10](#_Toc60662706)

[Local Experiments for CECacheEngine 11](#_Toc60662707)

[Postgres 12](#_Toc60662708)

[MySQL 15](#_Toc60662709)

[Apache Cassandra 16](#_Toc60662710)

[Oracle 17](#_Toc60662711)

[MongoDB 18](#_Toc60662712)

[Apache Ignite 19](#_Toc60662713)

[AWS Experiments for CECacheEngine 20](#_Toc60662714)

[Postgres 21](#_Toc60662715)

[MySQL 24](#_Toc60662716)

[Apache Cassandra 26](#_Toc60662717)

[Oracle 27](#_Toc60662718)

[MongoDB 28](#_Toc60662719)

[Apache Ignite 29](#_Toc60662720)

[Local Experiments for Large Data 30](#_Toc60662721)

[Postgres 31](#_Toc60662722)

[MySQL 33](#_Toc60662723)

[Apache Cassandra 34](#_Toc60662724)

[Oracle 37](#_Toc60662725)

[MongoDB 38](#_Toc60662726)

[Apache Ignite 40](#_Toc60662727)

[AWS Experiments for Large Data 41](#_Toc60662728)

[Postgres 42](#_Toc60662729)

[MySQL 45](#_Toc60662730)

[Apache Cassandra 46](#_Toc60662731)

[Oracle 47](#_Toc60662732)

[MongoDB 49](#_Toc60662733)

[Apache Ignite 51](#_Toc60662734)

## 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.

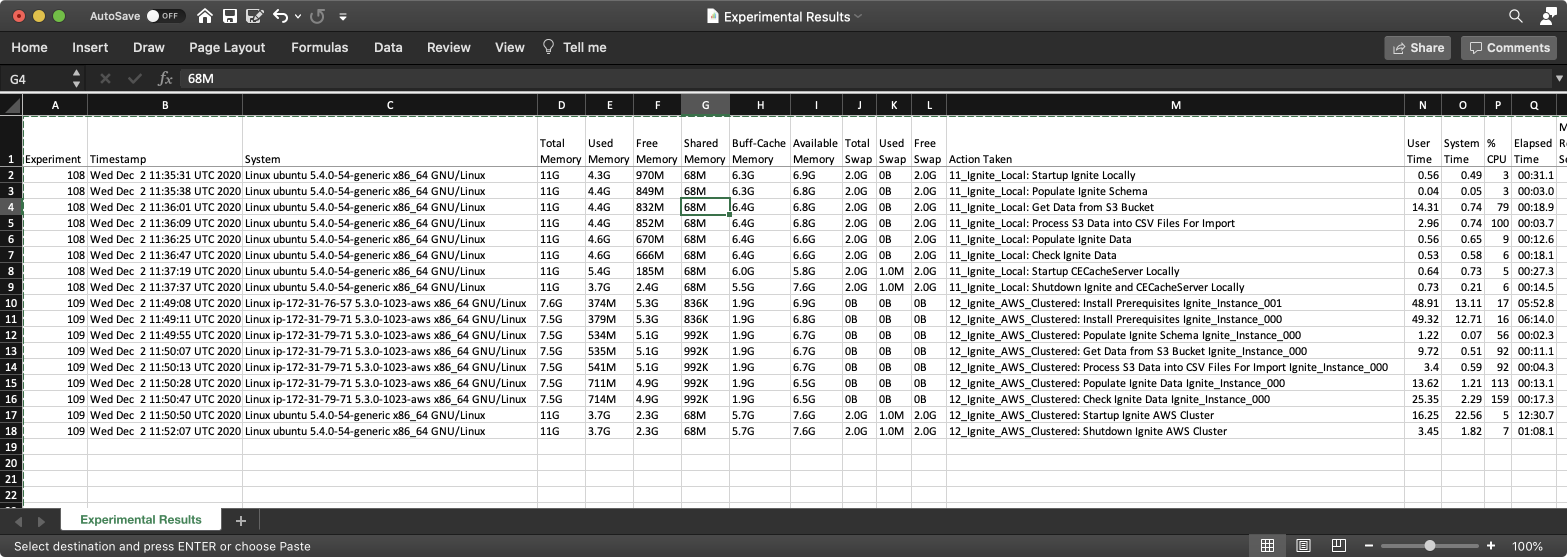
## Metrics Being Collected

* 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 locally 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 locally 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 locally 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 locally 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 locally 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 locally 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

Experiment 139

Text

Description automatically generated with medium confidence

Chart, treemap chart

Description automatically generated

Several things come up right away as we compare against Experiment 171 (Local Postgres CE Data).

1. Startup time for the MySQL database takes 6 times longer than the Postgres database.
2. Startup time for the CECacheServer takes 40% longer than compared to the Postgres database.
3. Startup time for populating the schema in MySQL takes almost 50% longer than compared to the Postgres database.
4. Time to check the data in MySQL is almost 20% faster than compared to Postgres.

Comments.

1. It is still possible that MySQL would be a decent choice. Performance during queries is a very important factor. We do that far more often than starting up the container (except during development). Further evaluation into the numbers for large data performance and AWS performance will tell us some answers here.

#### Apache Cassandra

Experiment 100

Text

Description automatically generated

Compared to Postgres, Apache Cassandra performs poorly on Docker Container creation, schema creation, insertion of data, and basic SQL operations on data. Oddly, it does well on CECacheServer startup, but I suspect that this is because all that is being asked of it is to return all of the initialization tables.

It remains to be shown why Apache Cassandra would be a good choice for our purposes. We’ll see more in further experiments.

#### Oracle

Experiment 51

Text

Description automatically generated

As far as comparing Oracle in a Docker Container against Postgres in a Docker Container for CECacheEngine performance, there is little to recommend Oracle for the application developer.

1. Startup time was almost 2 minutes for Oracle, versus just over 2 seconds for Postgres.
2. Schema creation time for Oracle was twice as slow versus Postgres.
3. Data population time for Oracle was four times slower versus Postgres.
4. The shutdown time was almost twice as slow for Oracle versus Postgres.

The one thing that Oracle was better at than Postgres was the CECacheServer startup time, a one-time event. I also suspect that the time was higher than in Postgres, as it was also measuring and outputting times within the application, which wasn’t carried out for Oracle.

We can do better than staying with Oracle for our application developers when running a database on their desktop.

#### MongoDB

Experiment 30

A picture containing chart

Description automatically generated

MongoDB was run without a schema, as we wanted to get the best results possible from the database engine, and not rely on an IT type crutch to enforce that the documents going into the database conformed to a predefined schema. That sort of activity belongs in application code and used when fear is present, such as in ETL operations. Just like the developers of MongoDB from day one, we expect that applications developers know which attributes are needed for correct schemas and which ones are purely optional.

Running MongoDB in a container versus Postgres in a container bolster that performance is significantly faster in MongoDB than against the database that seems to hold the lead (Postgres).

1. MongoDB was perceivably slower starting up, but Postgres as slower by about the same amount shutting down, so let’s call that a wash.
2. MongoDB has roughly the same populate performance as Postgres. It should be noted that we are running mongoimport to perform the CSV to JSON document translation, which may not perform as well as native programmatic JSON document creation, especially when the JSON document may already be present, as with many GUI frameworks in their browser clients.
3. The preliminary data checks that this experiment uses take about as much time as Postgres.
4. Where MongoDB shines is in the CECacheServer startup time. It takes about a quarter of the time Postgres takes to load the tables in CECacheServer.

This is quite interesting. We still have to see what happens with large data and AWS based solutions, but thus far, we are seeing a rivalry with Postgres. Raw speed for CECacheServer loading should not be the only factor we consider.

#### Apache Ignite

Experiment 22

Table

Description automatically generated with medium confidence

These results aren’t outstanding, especially in the populate data and check data steps. It’s not a knockout blow compared to Postgres, but it’s not particularly appetizing, either. Let’s see how Ignite does in AWS (Experiment 19), local large data (Experiments 404, 406), and AWS large data (Experiments 407, 408, 409).

### 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

Experiment 148

Graphical user interface, text, application, letter

Description automatically generated

Chart, treemap chart

Description automatically generated

Again, you can see that two things dominate this chart and make things difficult to understand:

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).

So, when we take those two variables out of the equation, along with Get Data from S3 Bucket (time it takes to download data is from S3, decrypt it, and make ready for loading into the database, which we know is the same as for Postgres), as well as the Local MySQL time, we are left with this.

Chart, treemap chart

Description automatically generated

Indeed, this is a weird result. We are looking at a result where we see the visualization of the real differences between this experiment and the corresponding one for AWS Postgres.

1. The amount of time it takes to put the schema into MySQL is almost twice the time it takes to do the same schema in Postgres on AWS.
2. The times is takes to insert the data into MySQL as well as query the data in MySQL, compared to the same tasks in Postgres on AWS take almost half the time as they do on Postgres on AWS.

Normally, I would favor the MySQL results at this point, because we will process data far more often than changing schemas. And the end users feel the time for database processing, not time spent on offline schema maintenance. However, all of the same extreme reservations I had for PostgreSQL on AWS RDS carry forward to MySQL on AWS RDS. I’d like to see large volume processing times to make a decision on MySQL and suitability for our purposes.

#### Apache Cassandra

Experiment 128

Text

Description automatically generated

First, a word about how these times are generated. These times are generated when the task they are reporting on is complete. For the AWS RDS Aurora databases (PostgreSQL and MySQL), the “Startup” task has always been completed first, so it reports first. However, for the remainder of the other databases we are testing, they are being built from scratch on EC2 instances. As Terraform is telling AWS which resources to build to our specifications, it is also placing provisioning code on the EC2 instances, and then running that code. So what we see is timings for “Installing Prerequisites” (installing the database and configuring it for our use), “Populate Schema”, “Get Data from S3”, “Process S3 Data into CSV Files”, “Populate Data”, and “Check Data” before exiting the “Startup” task.

Therefore, the “Startup” task includes all the time in those aforementioned provisioning, setup, and testing steps, in addition to the startup time itself. Although, as I compare the time to run everything for Cassandra on an AWS EC2 instance that we build from scratch, it is still about half of the time it took for the AWS RDS Aurora setup time to complete (and is lot cheaper to run, as well!)

In any case, Cassandra performs quite poorly in this experiment against the PostgreSQL instance running on AWS under the same tests when it comes to most important things: adding data to the database and querying data from the database. We’ll look to the large data experiments for Cassandra for more evidence for consideration. But at this point, the outlook is not good, at least for CECacheServer.

#### Oracle

Experiment 418

Text, letter

Description automatically generated

So, we have some explaining to do here, because I believe that these results at first look quite good compared to the AWS Aurora PostgreSQL Clustered results.

1. The almost 14 minutes of Install Prerequisites are not measurable in the PostgreSQL results. That is a consequence of the fact that AWS RDS comes as a service to us, and we perform all of the schema, S3 data, populate data, and check data on the client side for PostgreSQL and on the Oracle server for this experiment.
2. Even with the above true (all client-side things that we are now measuring on the server), we are still close to half of the time that an AWS RDS Aurora setup takes. I don’t believe that’s because Oracle is so much better and faster to install. Rather, I believe that this is a consequence of the fact that we have no knowledge of what’s going on for the AWS RDS setup. You see this same effect on the Shutdown measurement.
3. You definitely see the effect of the client-side operations for the schema installation, data populate, and data check measurements. They are between 50-99% faster.

I feel that this experiment has little to do with the inherent speed measurements between the two databases. We already know, from Experiment 51 (where we measured Postgres versus Oracle operating locally in containers) that Oracle could be up to four times slower than Postgres. What we are seeing here is the difference between client-side bulk operations on a database versus server-side database operations.

#### MongoDB

Experiment 35

A picture containing text

Description automatically generated

As with Experiment 30 (Local Docker MongoDB), these are encouraging numbers when compared to PostgreSQL AWS (experiment 176).

1. We are using an AWS EC2 instance for MongoDB, rather than some database service as AWS RDS Aurora. The “Install Prerequisites” step here should actually be included in the “Startup MongoDB AWS” time, but even doing that, the MongoDB database server instance comes up and shuts down significantly faster than PostgreSQL.
2. The database population and check times are 90% lower than Postgres. But this is likely an effect of the fact that PostgreSQL must be run with a local client, as we don’t have server access to an AWS RDS Aurora instance. Still, consider that we have the option of running things like the load part of ETL directly on the MongoDB server, and that may be of interest as we try to speed up import times.
3. We still need to see how the database performs under large data conditions, although I suspect that we will have a very viable alternative to Postgres.

#### Apache Ignite

Experiment 19

Text, letter

Description automatically generated

Again, at least at first glance, these results are really outstanding compared to AWS RDS Aurora PostgreSQL. Notice that the AWS resource creation and destruction times as quite favorable. And also bear in mind that these experiments are using Ignite running on two instances in a cluster with the ability to have all three qualities of the CAP theorem solved: Consistent, Available, and Partition Tolerant. It will be interesting to see if the performance results can stand up to large data stressing.

### 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

Experiments 414, 415, 416

A picture containing table

Description automatically generated

As you can see in the above analysis, comparing MySQL against Postgres for large data yields consistent results. It consistently

* Takes longer to bring up a MySQL database than a Postgres database. This is a concern for developers, as it adds over 10 seconds for the container to become available for database work.
* Takes less time to apply a schema to a MySQL database than a Postgres database. But developers will feel only about 1.5 seconds of relief.
* Takes consistently about twice as long to populate a MySQL database than a Postgres database. A 1.5 minute operation in Postgres becomes almost 6 minutes in MySQL. Developers will definitely feel that.
* Takes consistently about 30% more time to query a MySQL database than a Postgres database. This is perhaps the biggest concern to developers, as they are writing code that uses the database all the time in their applications.

#### Apache Cassandra

Experiment 271

Text, letter

Description automatically generated

As poorly as Apache Cassandra performs against Postgres, the picture is much worse than this. The Check Cassandra Data step failed outright. Here’s the screenshot from the failure.

Text

Description automatically generated

As you can see, the final three tests failed. Let's describe and discuss.

The first failure is on a rather easy bit of SQL.

select count(\*) from PGYR19\_P063020.PI;

What's going on here? Several things.

First, remember that the central goals of Cassandra revolve around Fault Tolerance, decentralization, scalability to PB scales, and durability. And that brings us to the issue of tombstones.

In Cassandra, deleted data is not immediately purged from the disk. Instead, Cassandra writes a special value, known as a tombstone, to indicate that data has been deleted. Tombstones prevent deleted data from being returned during reads and will eventually allow the data to be dropped via compaction. Tombstones are writes – they go through the normal write path, take up space on disk, and make use of Cassandra’s consistency mechanisms. Tombstones can be propagated across the cluster via hints and repairs. If a cluster is managed properly, this ensures that data will remain deleted even if a node is down when the delete is issued.

Tombstones are also created when null data is imported into the database. If you carefully look at 02\_populate\_large\_data\_load\_data.sh, you will see that I avoid the tombstone issue by converting NULL to the string 'n/a'. I even tried more aggressive options to rid ourselves of tombstones (which are actually poor ideas on production use) of using nodetool to compact the database and reduce the time to live for tombstone data to zero.

Those measures get rid of tombstones for our purposes, but then surface a more deeply disturbing issue. Cassandra does not like table scans. There is an option for cqlsh to "ALLOW FILTERING" if you absolutely need to tell Cassandra to return a result set that requires a table scan, but that brings up an even graver problem: performance for such queries are so slow that the server times out from the client. You can see these on the console output during the test run. It's possible to change the timeout values to make these timeouts have longer values, but doing so on the Cassandra image would require a change to the Dockerfile for Cassandra, which I didn't feel like doing.

Then there's the matter of grouping, aggregation, and ordering.

This fairly simple query fails.

SELECT physician\_first\_name, physician\_last\_name, SUM(total\_amount\_of\_payment\_usdollars), COUNT(total\_amount\_of\_payment\_usdollars)

FROM PGYR19\_P063020.PI

WHERE physician\_first\_name IS NOT NULL AND physician\_last\_name IS NOT NULL

GROUP BY physician\_first\_name, physician\_last\_name

ORDER BY SUM(total\_amount\_of\_payment\_usdollars) DESC

LIMIT 10

These are further limitations of in cqlsh that further make Cassandra unsuiutable for our use. These revolve around grouping and ordering of results. From <https://cassandra.apache.org/doc/latest/cql/dml.html>:

Grouping results The GROUP BY option allows to condense into a single row all selected rows that share the same values for a set of columns.

Using the GROUP BY option, it is only possible to group rows at the partition key level or at a clustering column level. By consequence, the GROUP BY option only accept as arguments primary key column names in the primary key order. If a primary key column is restricted by an equality restriction it is not required to be present in the GROUP BY clause.

Aggregate functions will produce a separate value for each group. If no GROUP BY clause is specified, aggregates functions will produce a single value for all the rows.

If a column is selected without an aggregate function, in a statement with a GROUP BY, the first value encounter in each group will be returned.

and

Ordering results The ORDER BY clause allows to select the order of the returned results. It takes as argument a list of column names along with the order for the column (ASC for ascendant and DESC for descendant, omitting the order being equivalent to ASC). Currently the possible orderings are limited by the clustering order defined on the table:

if the table has been defined without any specific CLUSTERING ORDER, then then allowed orderings are the order induced by the clustering columns and the reverse of that one. otherwise, the orderings allowed are the order of the CLUSTERING ORDER option and the reversed one.

For these reasons, Cassandra is not suitable due to its limitations for our intended uses and results are not recorded. For this reason, the AWS test on large data was not run.

#### Oracle

Experiments 282, 283, 284

Table

Description automatically generated

The result comparisons of Oracle in a Docker Container against Postgres in a Docker Container for large data performance mirror the results from Experiment 51, as there is little to recommend Oracle for the application developer.

1. The startup times are unacceptable.
2. The data population times are unacceptable and grow worse at a faster pace with increasing amounts of data.
3. The shutdown times are unacceptable.

The Get Data from S3 Bucket times are probably better because on this experiment, it is an AWS EC2 instance requesting data from an AWS S3 Bucket (rather than my machine requesting S3 data), and they undoubtably have faster network speeds as well as a closer physical proximity to the data.

Oracle for the desktop is unquestionably a poor choice for application developers.

#### MongoDB

Experiments 324, 330, 331

Table

Description automatically generated

Unhappily, the early success of Experiment 30 do not seem to carry forward into larger datasets for execution speed. The most important metrics, those for populating and then checking query performance are both slower and non-linear increasingly slower the larger the dataset grows. The queries were all translated into mongo native language to make sure that we were using the most efficient form for MongoDB. For example, the query that was probably the hardest looks like this in SQL:

SELECT physician\_first\_name, physician\_last\_name, SUM(total\_amount\_of\_payment\_usdollars), COUNT(total\_amount\_of\_payment\_usdollars)

FROM OP\_DTL\_GNRL\_PGYR2019\_P06302020

WHERE physician\_first\_name IS NOT NULL

AND physician\_last\_name IS NOT NULL

GROUP BY physician\_first\_name, physician\_last\_name

ORDER BY SUM(total\_amount\_of\_payment\_usdollars) DESC

FETCH FIRST 10 ROWS ONLY;

For MongoDB, their aggregation pipeline was used to ensure that the correct optimizations were taken on where things like filtering were performed. So, the same query looks like this:

use PGYR19\_P063020

db.getCollection('PI').aggregate( [

{

$match:

{

"Physician\_First\_Name": {"$exists": true, "$ne": ""},

"Physician\_Last\_Name": {"$exists": true, "$ne": null},

},

},

{

$group:

{

\_id: ["$Physician\_First\_Name", "$Physician\_Last\_Name"],

Total\_Amount\_of\_Payment\_USDollars: {$sum: '$Total\_Amount\_of\_Payment\_USDollars'},

Total\_Number\_of\_Payments: {$sum: 1}

},

},

{

$sort:

{

Total\_Amount\_of\_Payment\_USDollars: -1

}

},

{

$limit: 10

}

],

{

allowDiskUse: true

}

)

exit

Note that MongoDB needed to have disk usage allowed. Without it, things were faster, but the largest datasets could not be processed in 20GB of memory.

The conclusion is that MongoDB has potential if sufficient memory is present and the queries are not difficult (such as aggregation). If we proceed with MongoDB, we should proceed with caution.

#### Apache Ignite

Experiments 404, 406

A picture containing text

Description automatically generated

Well, these experiments didn’t do very good against Postgres in similar conditions. They are miserable across the board.

I’m sure that you noticed that the 9,000,000 row experiment wasn’t run. That’s because I couldn’t get enough memory on my machine to run the experiment! I’m using a 32GB MBP, and running these experiments on Ubuntu in Fusion. Even giving Fusion 24 GB of memory to work with, there was very bad swapping taking place.

Bear in mind two things for this experiment.

1. Having a 9M row database probably isn’t needed for the average developer to use in conjunction with their application development cycles.
2. Apache Ignite is being run in memory only. Switching to persistent storage, and using memory for cache, like normal database managers, would undoubtably speed up the results here and allow us to run with more typical memory settings.

Let’s see the final scenario, where Ignite is run on AWS machines with sufficient memory to run the in memory database engine.

### 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

Experiments 256, 257, 258

Table

Description automatically generated with medium confidence

As you can see in the above analysis, comparing MySQL against Postgres for large data on AWS RDS Aurora Cluster yields consistent results. It consistently

* Takes about the same time to bring up the AWS resources comparing MySQL to PostgreSQL. That should be expected, as both use AWS RDS Aurora resources.
* Takes anywhere as over twice as long to over six times as long to start the local Docker container used for client access. This is consistent with all other MySQL to Postgres Docker Container experiments already reported on.
* Takes statistically significant less time to apply the schema to the MySQL database than the PostgreSQL database, although in real life, this is done rarely and amount to at most around a second of wait time.
* Takes almost the same amount of time to load the data into the MySQL database as it does with the PostgreSQL database.
* Takes significantly more time to process queries against the data in the MySQL database as it does with the PostgreSQL database, especially with larger and larger data.

For the reasons stated above, I would pick the AWS RDS Aurora PostgreSQL service over the Aurora service, although for the reasons stated earlier, I am still violently against recommending AWS RDS Aurora for a database solution.

#### Apache Cassandra

Due to Apache Cassandra being deemed unsuitable, and in fact unable, to run our large data suitability tests during the Local Large Data Tests, the AWS test was not attempted.

#### Oracle

Experiments 312, 313, 316

Table

Description automatically generated

When we compare the results between Oracle running in AWS EC2 instances against PostgreSQL running in AWS RDS Aurora for large data performance, some interesting hypotheses begin to gain evidence.

1. The amount of time Oracle takes to install on EC2 instances is insane, even compared to the absurd amount of time it takes AWS RDS Aurora PostgreSQL to come alive. But the good news is that we now have the proper way to bring Oracle instances alive in a fully automated fashion.
2. It consistently takes about 300% more time to install the Oracle schema than the PostgreSQL schema.
3. Getting the S3 Bucket data is consistently about 70% faster. As discussed earlier, this is probably due to the fact that the PostgreSQL data was done on my own network, which I’m sure is slower than the network inside the AWS datacenters, as well as the latency issues. Bear in mind that AWS S3 CLI works on top of TCP/IP, which must send packets that get acknowledged between client and server. So, the further your client and servers are apart will have a limiting effect on even the fastest connections.
4. The time to do the data loads into Oracle are longer than that for PostgreSQL and grow further and further apart under larger and larger dataset sizes.
5. The data query capability has times that were better for Oracle at more modest sizes and worse for Oracle at the largest dataset size. This may be something that could improve with tuning on the databases.

I would say that there is no compelling evidence that Oracle is better than PostgreSQL for future use. It’s not like Oracle, with its decades of experience, has RDBMS queries capabilities that are heads and shoulders above what even humble Postgres can offer us.

However, it is inevitable that we will have need to transition through Oracle, and it probably would be a good thing to learn how to deploy full CareEngine with Oracle in a complete cloud environment, but not for production. We should get our feet wet with deploying build server tests to environments that last only as long as the integration tests for a build run first. Then, we should start to manually start to deploy less ephemeral environments. This will give us the chance to begin to understand the sizing of AWS EC2 database instances, perhaps begin to support Oracle clustering, and experiment with CareEngine container orchestration, trying out small ELB clusters, Docker Swarm, and Kubernetes (ordered by complexity and ease of maintainability).

So, even if Oracle doesn’t have a future with CareEngine (I do recommend that we “Get The Red Out” and not write application code as if we were an IT department depending on the database to keep us safe from ourselves), the time we spent to develop this Oracle code will serve us well in the months ahead.

#### MongoDB

Experiments 341, 342, 343

Graphical user interface, text, application

Description automatically generated with medium confidence

This performance of MongoDB on an AWS EC2 instance versus PostgreSQL on an AWS RDS Aurora instance isn’t quite as bad as the similar Experiments 324, 330, 331 (local Docker based MongoDB).

In fact, the populate data steps are better than both the MongoDB and the PostgreSQL AWS results. But that may again be the effect of PostgreSQL being a client based upload of data to the RDS server instead of data loaded from the MongoDB server itself.

However, the check data steps are troubling. The performance seems to be much worse for MongoDB and perhaps is exponential with regard to rows.

Chart, line chart

Description automatically generated

Further analysis would be wise before deciding on MongoDB to be fit for function for our use.

#### Apache Ignite

Experiments 407, 408, 409

Table

Description automatically generated

It would great at this point to say “Winner, Winner. Chicken Dinner!” But I won’t. There are a few caveats. Some good points. And perhaps some issues that deserve further examination.

On the good side, we are seeing the best results from all the AWS experiments on large data. And it’s staying much flatter over record count on Ignite than on PostgreSQL.

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Another positive thought: it would be interesting to remove the CECacheServer component altogether from CareEngine and see what sort of penalty, if any, there is in getting rid of the cache component. Startup time would undoubtably improve, and with a properly sized Ignite Server, performance overall should be much improved. Deployments would be quicker and easier as well, because of the simplified application architecture.

Also, the idea that Apache Ignite solves the CAP issues is a blessing of biblical proportions. And the fact that standard SQL can be used instead of something like MongoDB aggregation pipelines makes everything more readable for everyone, and makes moving to another database engine that much easier for the future. Code is written for humans, not for computers.

However, nothing comes for free. Apache Ignite requires significant resources to run. The 5.6GB database required a 64GB machine to run in, although I did not try a 32GB machine if I recall correctly. This meant running an m5.4xlarge EC2 instance (at $0.76/hour) [16 vCPU, 70 ECU, 64 GiB memory, EBS disk, EBS Optimized by default], rather than the m5.large EC2 instances that I ran everything else on (at $0.096/hour) [2 vCPU, 10 ECU, 8 GiB memory, EBS disk, EBS Optimized by default]. Just for reference, something for production might look like the m5d.metal instance type, which costs $5.424/hour, has 96 vCPU, 345 ECU, 384 GiB memory, 4 x 900 NVMe SSD disk, EBS Optimized by default, max bandwidth 19,000 Mbps, max throughput 2,375 MB/s, Max IOPS 80,000. That’s about $48K per year, which is a bargain for such a capable server.

In any case, if there one database that I would choose, it would be Apache Ignite. It’s still a young project, but has a 21st century outlook as far as what hardware is these days, and what problems are significant to solve for databases.