Investigating I/O performance for Amazon EC2 and S3 - Team OR1

Daniel Fernandez
Computer Science Department (student)
University of Houston
Houston, Texas
Email:dferwave@gmail.com

Glenn Turner
Computer Science Department (student)
University of Houston
Houston, Texas
Email: gaturne@gmail.com

***Abstract*—This project was initiated to explore the practical performance of finding and locating records in a large file on Amazon Web Services Elastic Block Store (EBS) and Simple Storage Service (S3). We created two programs to store a moderately large number of records, 2.48 million, in random order. We then selected 215 thousand of these records to retrieve from EBS and S3, and timed the results. Performance differences were seen between instance types and and disk types. Additionally, we installed MongoDB on the fastest instance we tested and found that the response times greatly overshadowed the disk access times and was not suitable for the type of performance evaluations we attempted here.**

I. Introduction

In the past (2016), there were observations that Oracle on Amazon RDS maxed out the I/O per second, or bandwidth, in performing some table queries [1]. This prompted further investigations by that author for Oracle on Amazon Elastic Block Store (EBS) for Elastic Compute Cloud (EC2). Amazon EBS allows for a variety of different kinds of storage devices: General purpose SSDs (gp2) and a higher performance SSD (io1). In addition there are instances with throughput optimized devices, namely the "D" and "I" family of instances. All EBS devices are SSD block storage devices.

The Linux utility, *iostat*, was used in that study to determine throughput (bytes per second) and I/O operations per second. For the "provisioned" I/O devices (io1) the author determined that the performance for large device block sizes is throughput bound (bytes / second) and for smaller blocks, it is bound by I/O operations per second [2]. This is no surprise. For the standard (gp2) devices used in that study, the throughput was about the same for all block sizes, but I/O per second was seen to have definite peak values.

Storage technology has evolved, and in fact Amazon EC2 now offers NVMe devices, but it was not clear to us how to create these volumes and use them. We would like to see these optimal performance points for the General-purpose SSDs, the Provisioned SSDs and optimal throughput devices for Amazon EBS.

In addition, the Simple Storage Service (S3) offers the possibility of storing each individual record as an object and retrieving that, or, storing blocks of objects that could possibly be retrieved faster than "seeking" then reading a block as with a conventional file system. However, it should be noted that S3 primarily boasts about being scalable, highly available and durable, and able to provide virtually limitless storage for any type of data. And that relatively little is mentioned about its time performance when advertised.

The purpose of this work is to explore the practical performance of some particular instances and disk types that are currently available for EC2. In addition, we want to see if object storage can provide the transaction speed to facilitate data retrieval. For this we developed experimental software platforms, where we have control over the physical record size. In the case of EC2 block devices we evaluate the performance of the devices noting the block size of the device and it's relationship to the physical record size.

II. Methodology

We hold a list of registered voters for Harris County Texas valid in 2021. Further, we can make use of publicly available election rosters (the list of voters for each day of voting, as well as cumulative ballot by mail) for the election in 2021. Each voter record has a unique voter ID, and the election rosters contain the voter ID for each person casting a ballot. We store registered voter data, 2.48 million records in blocks and then, retrieve in random order, from the election rosters, voter records based on the voter ID in the rosters. All of this data was provided in .csv files.

This design was based on our thought that a good metric of functional IO would be to aggregate these records in random order, then to select some sparse number of records at random. The retrieval time is a practical measure of the utility of the storage mechanism.

A. Experiments with EBS

The data store is constructed with the voter data from a .csv file being padded to give a uniform length of 128 bytes. (This wasn't necessary but we thought it would provide for future expansion.) The records are stored in binary array buffers (blocks) of a chosen size. This size is a global program parameter. The buffer number and the offset into the buffer of the start of the record is calculated. A python dictionary entry holds the voter ID as key, and the location of the record specified by buffer number and the offset within that buffer.

Once the dictionary is built it is then written out to the file. The dictionary entries are packed into blocks (physical records) of the chosen size (same size for dictionary entries and for the data), and written out. The .csv file of voter records is read again and the data is packed into blocks and appended to the file. We call this the database file.

This process of building the dictionary and even writing the dictionary out to the file is surprisingly compute intensive. This is noted by the fact that the process times for these phases of building the database are not affected by disk speed but are greatly affected by the power of the instance chosen. See Table 1 for the instances used. This is demonstrated by figure 1 and in the detail of the T3 instance described below. This is is partly due, no doubt, to having to create hash codes for all the dictionary entries. In addition python does not allow of direct manipulation of binary data, but the raw data is converted to byte array objects to be stored in the block.

TABLE I: Instance types and characteristics tested.

Instance Type	Characteristics	Cost
t2.micro	1 core, 1, thread 1 GB, 2.5 GHz,	Free
t3 large	1 core 2 threads, 8 GB, 2.5 GHz	$0.0832
d3 xlarge	2 cores, 4 threads, 32 GB, 2.5 GHz	$0.499
I4 xlarge	2 cores, 4 threads, 32 GB, 2.9 GHz	$0.343

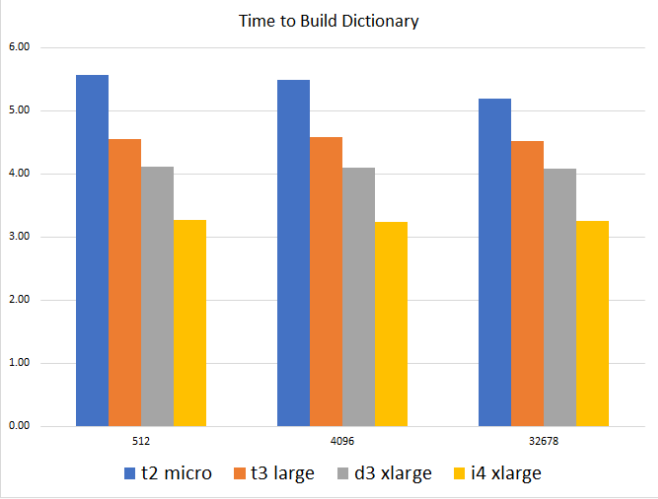

Fig. 1: Sensitivity to compute power for building dictionary, Times are in seconds and are an average of five runs.

To use the database, first the python dictionary must be re-created. This consists of reading the entries from the database file, and adding each entry into a python dictionary. Then a file of IDs is read in, and for each ID, a record is retrieved. The retrieval mechinism is to add the the number of directory blocks to the dictionary specified block, and seek to that block then perform the read.

To verify the retrieval was working, we initially printed out the first few records, then when we performed the runs for timed tests, every 2000 records the ID on the record retrieved is compared to the ID of the key to be found in order to verify the correct records are being retrieved. As noted in the figure, and will be noted again here, all times reported are an average of five runs.

1) General Purpose Instances for which we used mounted disks:

When any instance is created a system disk is provided. The user can specify either of the types General Purpose (gp2, gp3), Provisioned (io1 io2), Optimized Throughput, or Magnetic Standard. For the free tier the only allowed types are gp2 or gp3 volumes. Most instances have some limitation on the volume created to hold the instance storage. We created t2 and t3 instances using gp2 as instance store then created an additional volume to mount. This additional volume was used to hold the database. Table II shows the volumes we evaluated in this study.

TABLE II: Instance types and characteristics tested.

Designation	Measured Speed (buffered read)	Size
GP2	81.3 MB per sec	8 GB
GP2	170.1 MB per sec	8 GB
GP3	260 MB per sec	10 GB
IO2	330 MB per sec	10 GB
d3 xlarge	181 MB per sec	1.8TB
I4 xlarge(Nitro)	1050 MB per sec	800 GB

The disks listed for d3 and I4 are the disks automatically added. Another disk holds the Operating system and home directory - this can be specified at instance creation

It should be noted that when an instance is created the Region may be selected but within that region there are different Availability Zones. The instance may be created in any Availability Zone within the chosen region, but this is not within user control. When the volume is created, the availability zone may be chosen. The volume must be in the same availability zone to be attached. If a new instance is created to use that same volume it is a matter of luck if it will be in the same availability zone. This limited our experimentation some.

The code was developed on a desktop computer using Windows Subsystem for Linux. This was then first tried on EC2 instance of type ”t2 micro”. This is a free tier, and was created with a disk of type designated by ”gp2”. This particular instance is a minimal compute capacity. The process times for the various phases and block (physical record) sizes are shown in figure 2. These times were roughly twice the times seen using WSL. In addition, the process times don’t reflect the actual ”wall clock” times. The six seconds or so of process time, in some cases was actually about 25 seconds of ”waiting time”. There was limited experimentation with this instance.

We then went on to create a t3 large instance and mounted an additional gp2 volume. This was moderately faster than the t2 micro. The ”wall clock” time was noticeably faster and made experimentation more practical. We mounted a gp3 class disk disk and then an io2 class disk. For these attached volumes AWS lists a specification of IOPs, that is how many reads of 1k can the device perform in a second. In addition, we measured the burst read spend in MB / sec with the *hdparm* command. The t3 large instance was used with these devices:

- gp2 – 100 IOPs – 170.4 MB /sec

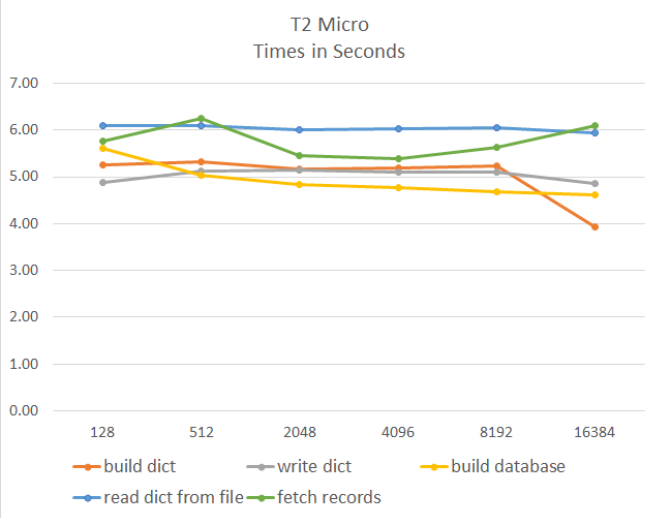

Fig. 2: The times (in seconds) for the various stages of database building and use, for various record sizes.

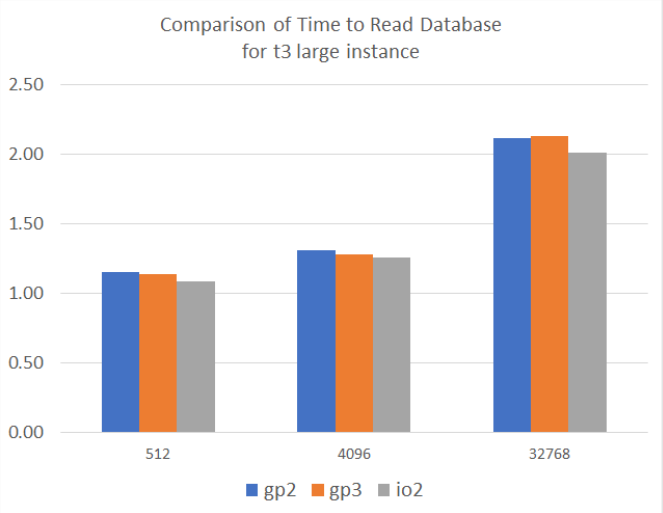

Fig. 3: The times (in seconds) for the various stages of database building and use, for various record sizes.

- gp3 – 4000 IOPs – 256 MB / sec
- io2 – 4000 IOPS – 330 MB / sec

Note that the gp2 we mounted with this instance was created specifying a higher throughput than the default gp2 we used earlier. Tests were run with each of these devices. Figure 3 shows the time in seconds to retrieve the 215 thousand records for each disk type. Comparing that to the "fetch records" line in figure 2 we see that it is about five times faster.

The longer times for the larger block sizes are due to the fact that a new block is read for almost every new record. We set a command line option to track the physical record number currently in the buffer, and avoided the seek, then read, if that record number is needed next. With the largest physical record size tested, 32768 bytes, the read was only avoided 17 times out of the 215000 records. In the case of the largest block size the entire 332768 byte buffer is read to retrieve a single 128 byte record.

2) Optimal Throughput Instances:

We sought to test some instances advertised as being "storage optimized[3]. Specifically we looked at the "D" family, and "I" family of instances. We chose the d3 xlarge as it seemed to have greater compute power than the t3 instance, and was, in fact, the smallest, and cheapest, of the d3 family. Creation of this instance gave a system disk that we let default to gp2, and it also created a 1.8 TB disk. We don't know the AWS designated type for this disk. Measurement of the throughput was 181 MB / sec, slower than the gp3 or io2 disks. The overall performance of this was greater than seen with the t3 however, this is probably due to the greater compute power: 4 threads, 32 GB of memory.

We also tested an instance of type i4 xlarge. The "i4" family is said to have the "highest local storage performance", we chose a type that had comparable CPU power to the d3 instance, 4 threads. In addition, this instance has a 2.9 GHz processor, vs the 2.5 GHz of all the others tested. The disk is part of the "Nitro" system. This system is not only high performance NVMe disks but a lightweight hypervisor. This instance also automatically provided a second disk, 800 GB, besides the default system disk. The measured throughput of this disk is much greater than any of the others we tested, 1050 MB / sec. An additional bonus for this instance type, it is only 68% the price of the d3 instance.

Figure 4 shows the performance of the "provisioned" io2 disk, the best of those not included in the storage optimized instances, with the d3 and i4 instances. We can easily see the i4 outperforms the others by a large margin at all physical record sizes tested.

One interesting note when looking at figure 4, the d3 disk outperforms the io2 disk when the record size is 4096 but not on the others. The io2 disk actually has a larger burst read speed than the d3 disk. The *fdisk -l* command prints out that the optimal block size for that disk is 4096, this seems to bear that out. For all of the disk tested the sector size is 512 bytes.

3) Tools used:

All instances were loaded with Ubuntu version 22, and code was written in Python 3. To measure the process times of the program we used the python

```
time.process_time()
```

before the function call to carry out the phase of the operation. To measure disk speed we used the Linux command

```
hdparm -Tt
```

Logical and physical sector size was determined with the command

```
fdisk -l
```

which must be run as super user. We did use the command

```
iostat 1
```

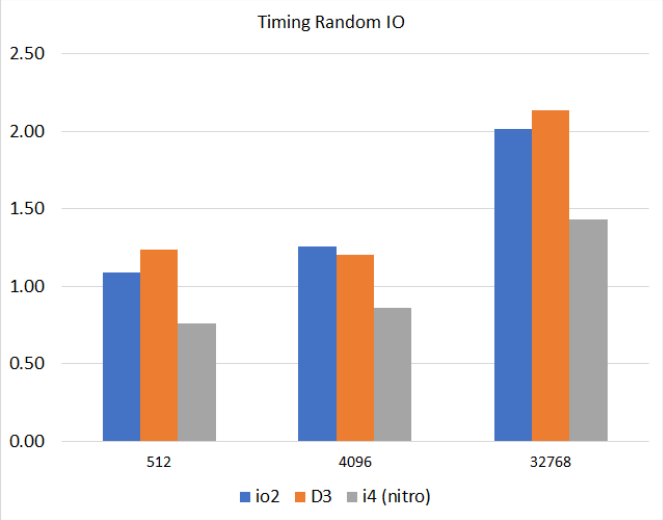

Fig. 4: The times (in seconds) to retrieve 215k records in random order for the highest performance disks tested.

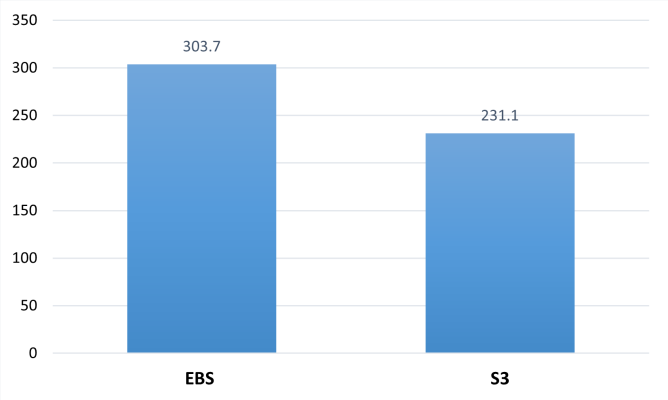

Fig. 5: The space (in megabytes) required for each storage service.

which prints out the cpu utilization (percent) and various disk parameter (total blocks read /written) every second. Our intention was that this would be a workhorse utility for this study. In actuality, the one second report interval, tended to capture slices of activity and not peak activity. It was useful in the tests of t2 instance, it showed idle CPU at times, and idle disk at times.

B. Experiments with S3

All S3 operations in the source code use the AWS CLI (Command Line Interface). To accomplish this, a virtual environment was used to install the python modules *awscli* and *boto3*. Then, an IAM user was created with the appropriate permissions for AWS S3 and the corresponding access keys were inserted after running the command

```
aws configure
```

so that the access keys would not appear in the source code. The resulting program was then run on a local laptop for initial testing and on an EC2 instance (t2 micro) for experiment results.

The data store is constructed with the voter data from a .csv file. Voter records are stored in the S3 bucket as binary/octet-streams and are separated by newline characters if there are multiple records in each object. The reason for separating records like this in each object was to avoid padding the records to fixed length to maintain one of the key differences between object storage and block storage. That is, object storage enables S3 to save the unused space in block storage since the records are padded to 128 bytes and the average record size is only about 97.4 bytes. As such, as shown in figure 5, S3 only needs to use about 76% the amount of space that EBS requires for this data set. However, it should be noted that this method comes with the drawback of not knowing exactly where a voter record starts within an object in advance. Thus, to improve random access time of records, which are represented as lines in a file, the python module *linecache* was used.

In each experiment, all objects have approximately the same number of records. Specifically, the range of the number of records an object contains is no more than one. Additionally, a python dictionary is always built so the program knows the location of each record within the S3 objects, with the exception of one experiment, where each object corresponds to exactly one record.

The first experiment attempted to have each object correspond to exactly one voter record. Thus, 2.48 million objects has to be put into the S3 bucket. By doing so, no location dictionary would be needed by the program. However, the resulting time performance of this experiment was atrocious. It took about **15 hours** to put all these objects in the S3 bucket and about **10 hours** to read 215 thousand records from it.

The next experiment attempted the opposite by having all records in one object. And while the resulting time performance is hundreds of times better, this approach requires requesting and saving all data into memory for any voter record(s) request.

Thus, the next S3 experiments attempted to balance the resulting time performance and the number of objects that the records are partitioned into. But, unfortunately, as seen in figure 6, it becomes clear that the time performance only increases as the number of objects the records are partitioned into increases. That is, past a thousand objects, the time performance trends towards that of the first experiment's.

Nonetheless, the best time performances that S3 displayed in figure 7 have process times comparable to the results shown previously in figure 2, which used the same type of EC2 instance. In fact, it has a faster cumulative build and read time. Though, of course, this is no longer true when one looks at the actual time that each took. Specifically, the EBS results shown in figure 2 consistently only took less than 30 seconds. Whereas the S3 results shown in figure 7 take at least seven times longer to build and longer to read despite loading the entirety of the data into memory from S3.

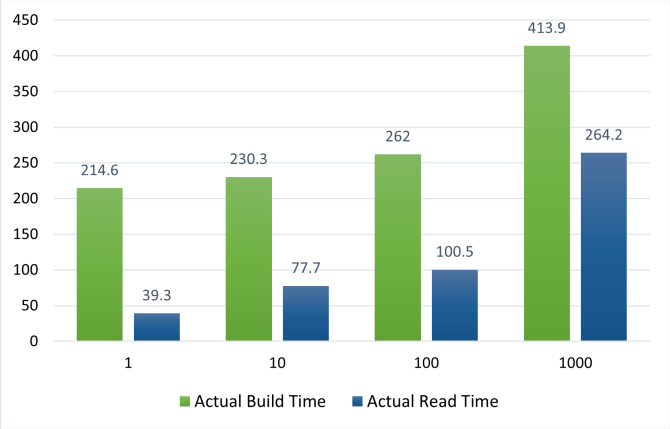

Fig. 6: The times (in seconds) to retrieve 215k records in random order from four S3 buckets with all 2.48 million records partitioned into objects.

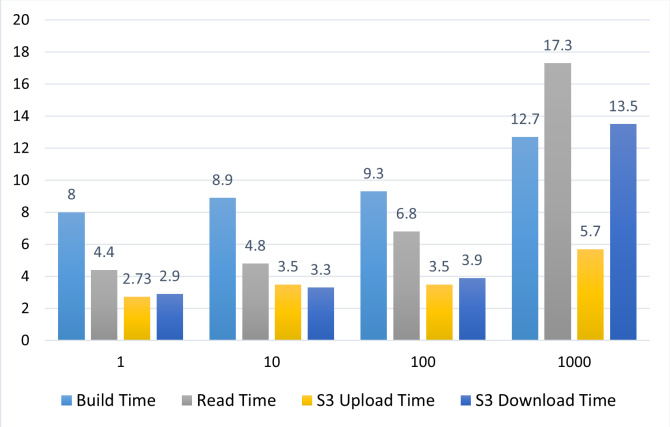

Fig. 7: The process times (in seconds) to retrieve 215k records in random order from four S3 buckets with all 2.48 million records partitioned into objects.

C. Experiments with MongoDB

We were able to install MongoDB on the i4 instance with some minor difficulty. The instance we created was using version 22 of Ubuntu. It was not apparent if another version was available. Currently there is no MongoDB version compatible with that Ubuntu version. A workaround was found by pre-installing some libraries and MongoDB was successfully run.

We used a python interface to MongoDB, importing the *pymongo* module. The database was built as before reading the lists of registered voters. As before, the records were padded to 128 bytes. Then the list of keys was read, and each record was retrieved. We specified that the database be loaded on the fast disks, not the system disk by use of the command:

```
mongod --dbpath /data
```

specifying the mount point for that disk. This database was surprisingly slow. So slow that we only took the average of times for three runs. The average time to build the database was **272.69 seconds**, and the average time to retrieve the 215000 records was **27.11 seconds**. Compare this to the roughly 9 seconds to build and write the database and less than 1 second to retrieve records for our software on this same instance. This database may not be suited for a large volume of rapid transactions and doesn't give a measure of disk performance.

We installed MongoDB on a standard desktop computer running Ubuntu version 20. The installation was done without any special intervention. On that system the build database took about 4 minutes. Our code took about 11 seconds for the total build time (build dictionary, write dictionary and write data). For this reason we think the poor performance is the MongoDB software itself, and not due to the installation problems.

III. Conclusion

While there are many things that affect IO performance, such as workload demand, IO request rate, even CPU clock speed. In a virtual environment the work load of the physical hardware comes into play. We nevertheless saw easily distinguishable differences in the instances and in the disks tested. We note that the cost is not negligible but prohibitive either. For a short project with large IO needs using an instance such as i4 makes more economic sense than buying a large fast disk drive. Or even a moderately long project using an appropriate instance is more economical than buying a desktop computer.

As for S3, while it does boast great availability, durability, and flexibility, has comparatively very poor time performance when it comes to creating a large database and supporting many random access requests for it by one client. However, when it comes to their process times, the difference between EBS and S3 are not nearly as drastic.

A. Possible Future Work

Since we have seen that it is awkward to manipulate binary data in python, and we think that there is a significant performance penalty for using python, rewriting the entire EBS portion of this software in C++ may be informative on the performance of python. An attempt to read the file, as we are creating it now, with a program in C++ was not successful. The standard integer on these Intel based machines is in Little Endian order; when python translates this to a byte array the order is changed, therefore the byte array needs to be ”unscrambled” for use.

Code Repository

The source code, as well as the spreadsheets containing timing data and presentation slides, can be found here.

References

[1] Elsins, Maris. “Investigating IO Performance on Amazon RDS for Oracle.” *Pythian*, 13 Dec. 2016, https://blog.pythian.com/investigating-io-performance-on-amazon-rds-for-oracle/.

[2] Elsins, Maris. “Investigating IO Performance on Amazon EC2.” *Pythian*, 30 Dec. 2016, https://blog.pythian.com/investigating-io-performance-on-amazon-ec2/.

[3] Amazon Web Services, ”Amazon EC2 Instance Types”, 13 Apr. 2023, https://aws.amazon.com/ec2/instance-types/.