YALM : Yet Another Lifecycle Manger

CSC 568 : Enterprise Storage Architecture

By
Team 3
Chakshu Singla (csingla)
Neeharika Sompalli (nsompal)
Darshak Harisinh Bhatti (dbhatti)
CSC Department, NC State University
May 7, 2018

Contents

Abstract

The purpose of this research is to review and improve the management of lifecycle of the objects on public cloud. Cloud vendors provide multiple storage tiers with differing data access latencies and storage costs. Data which is not frequently used may be stored on a less superior tier to reduce the overall storage cost. Therefore, the multi-tier approach may result in savings for the enterprises and individuals. However, the policies to move around the data should be dynamic, self-evolving and bi-directional. We try different policies to achieve the balance between the storage cost without compromising the access times. In the first phase, we wish to achieve smart policies by applying machine learning algorithms using synthetic workloads in a simulated cloud environment. In phase two, we will apply the promising policies observed from phase one to AWS S3 cloud store.

1. Introduction

Every cloud provider has storage offerings in different tiers. For example AWS offers storage service in three tiers,

- **S3** : Low data retrieval time, High cost
- **Infrequent Access S3** : Medium data retrieval time, Medium cost
- **Glacier** : High data retrieval time, Low cost

Similarly, Microsoft Azure has Hot, Cool and Archive [1] and Google has similar offerings [2] as well.

These tiers are differentiated based on access frequency and retrieval time; vendors like AWS provides data lifecycle configurations rules. These rules allow you to manage the lifecycle of objects in S3 allowing movement of objects after a set time period. The configurations rules are rather naive and have a limited scope.

- These policies are not meant for short time frames. Usually they kick in after 30 days or so [3].
- The policies work in a single direction that is moving data from S3(hot tier) to Glacier (cold tier).
- Majorly, these policies are not based on the access patterns of the object but only the last modified or creation times.
- Due to the above constraints, policies do not adapt to the dynamic workloads and need to be manually altered.

This trade-off between the storage costs and time offers a rather interesting problem. We believe that a better Cost-Time optimization is achievable with smarter policies which move around data between Hot and Cold tiers in an automated fashion. We try to do this by introducing an additional abstraction layer "YALM" which maintains metadata store for the objects as well as executes policies.

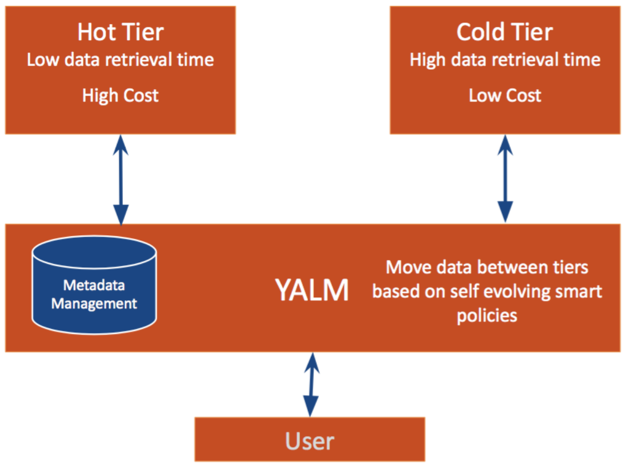


Figure 1: YALM Architecture

In figure 1 we show our system architecture. User interacts with our system which maintains metadata of objects. The metadata consists of access frequency, last-modified time, creation time and time difference between last two accesses. User is not aware of the tier of the object i.e. whether it is stored in Hot tier or Cold tier. Our system is responsible for the data movement between these tiers based on some policies. We decided to maintain our own metadata so that we can have more parameters other than creation or modification time. In future we wish to incorporate parameters such as file type or some tags to make more informed decisions about the type of user workload.

We have done this project in two phases,

- Simulation using Synthetic Workloads
- Integrating YALM with AWS

2. Simulation using Synthetic Workloads

To establish the above proof of concept we started with processing synthetic workloads on a simulated environment. This gave us a fair idea of the flow of the process and the optimizations that need to be done. We decided to mimic the near real world workloads; 1000 objects and total of 20000 requests were generated for each experiment. Technology stack we committed to is,

- Python
- MongoDB
- R and MS Excel to analyze the data

For the simulations we are using python as the tool to generate random requests for variable sized objects and get metrics such as total size of tiers, time required to get an object, number of objects in a particular tier etc. As mentioned earlier, we are maintaining our own metadata for which we are using MongoDB in this phase. We have done two types of simulations namely Psuedo-Random Requests and Power Law [4].

2.1 Pseudo-Random Requests

In the starting phase of the simulation we generated requests for the objects using pseudo-random number generator of python in a range of [0,999]. As shown in 2 we could see that it followed normal distribution for 20000 requests.

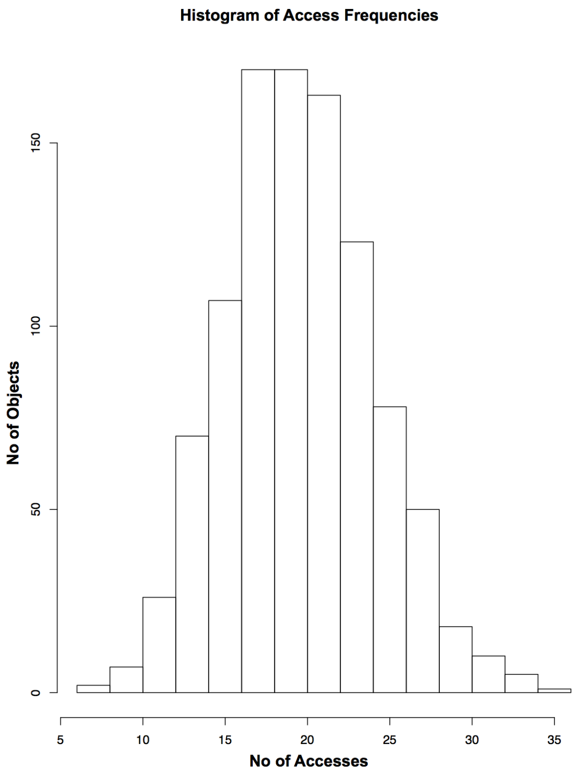


Figure 2: Histogram of access frequencies

The storage cost ratio for hot tier to cold tier was taken as 1:100 and the access time ratio for hot tier to cold tier was assumed to be 1:5. Our first and one of the most basic policies moved around the data in the object tiers using a threshold 'min-freq'. This threshold was computed using multiples of standard deviations obtained from the above graph. The objects were divided between hot and cold tiers based on whether their respective frequency was above 'min-freq'. The objects which had frequency higher than min-freq were considered hot and hence were kept in the hot tier and anything below min-freq was moved to the cold tier. From 2 we decided to divide data based on access frequencies 20,25,27,30 i.e. we further conducted four experiments,

- Objects having $access\ Frequency > 20$ in Hot tier which is $\approx 50\%$
- Objects having $access\ Frequency > 25$ in Hot tier which is $\approx 10\%$
- Objects having $access\ Frequency > 27$ in Hot tier which is $\approx 5\%$
- Objects having $access\ Frequency > 30$ in Hot tier which is $\approx 1\%$

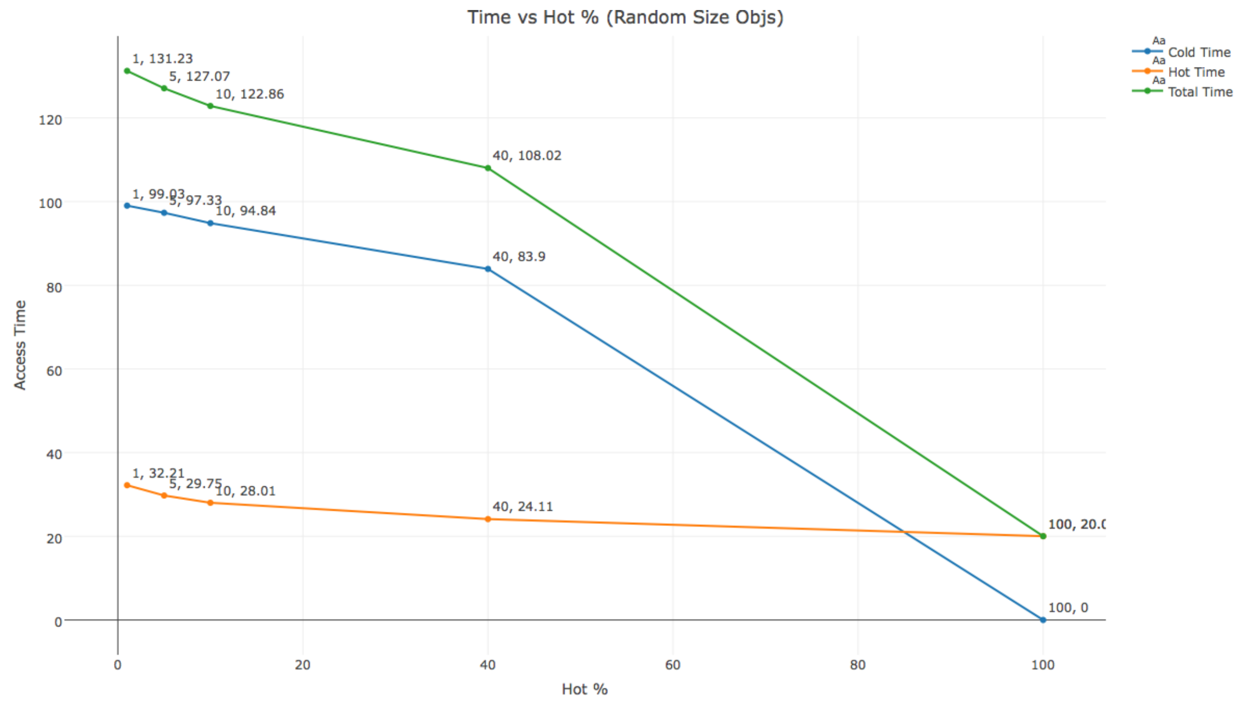


Figure 3: Pseudo Random Results

Figure 3 shows the results for the above four experiments. The three scatter lines captures the average access time for objects in Hot Tier, Cold Tier and the total access time plotted against the percentage of objects in the hot tier. As expected, on increasing the number of objects in the hot tier (which is directly proportional to the total cost of storage) the access time decreases. This observation was used as a baseline for the classic cost versus time trade-off problem.

2.2 Power Law Requests

The motivation behind carrying out the second simulation experiment was to mimic the workloads and access patterns as close to the real world as possible. Hence, the requests for the objects were generated using power law distribution using scipy [5] Python library. As it can be seen from 4 the frequency of individual objects follow a power Law distribution. Each series correspond to the frequencies observed after an interval of four thousand requests. One key observation that can be made from the figure is that as we capture the results for larger number of requests there is only a selected number of objects which display a significant change in frequency over time.

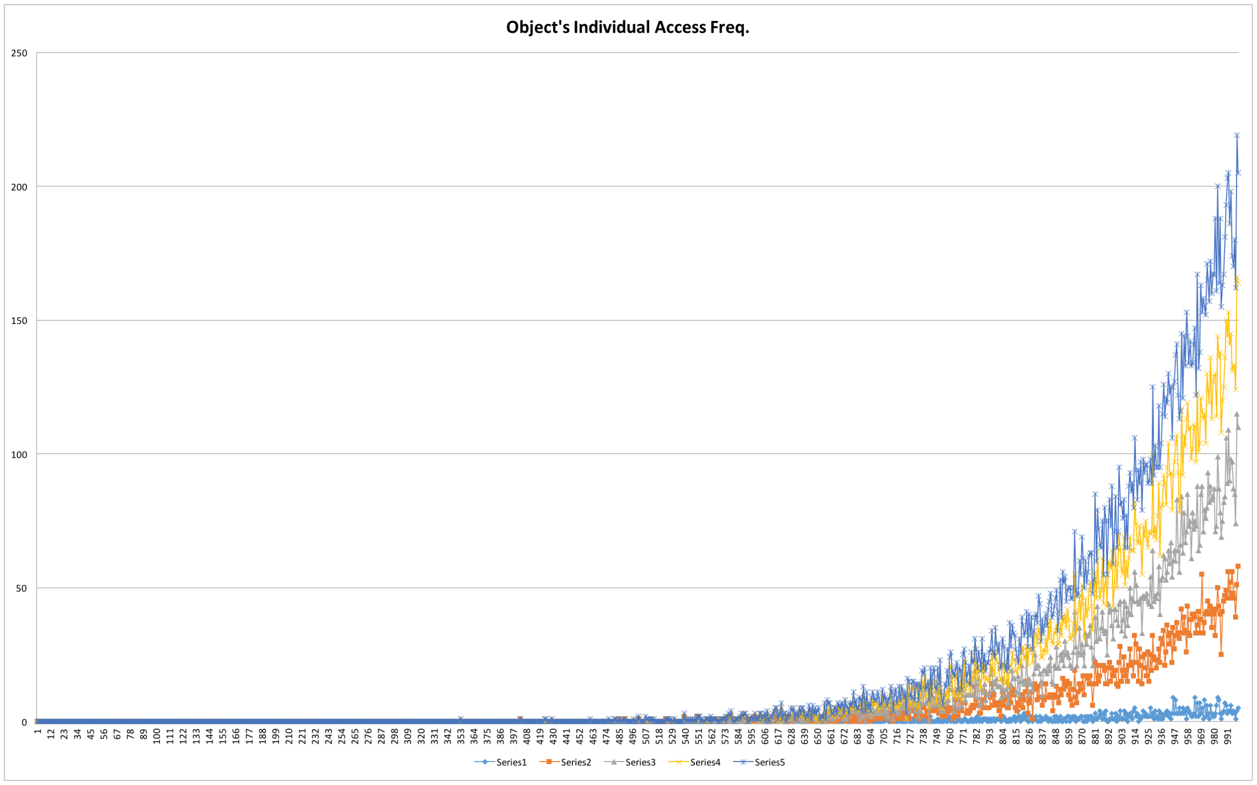


Figure 4: Scatter Line plot for individual object frequency

2.2.1 Threshold using standard deviation

The first policy we used to move objects between the tiers used the same four points as described in 2.1. As seen in 5 the policy did not yield the expected results. The access time did not drop uniformly despite increasing the percentage of objects in hot tier. One possible explanation for this can be that standard deviation points corresponding to the normal distribution were used for the requests generated in power law fashion. This policy was rendered inapt and was not examined further.

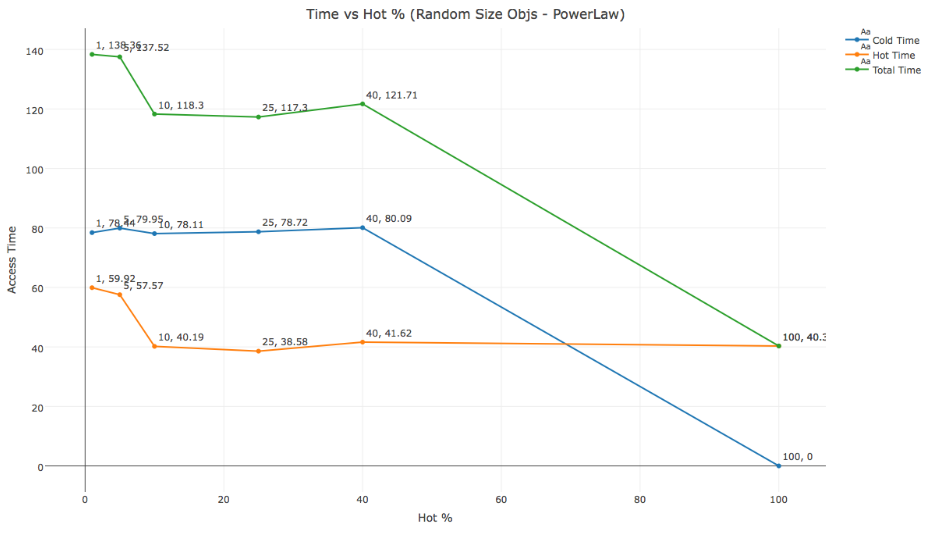


Figure 5: Powerlaw Freq Policy

2.2.2 Threshold using mean frequency

We needed to come up with a better policy for the power law distribution. Hence, mean which is a measure of central tendency was used to calculate min-freq threshold. Figure 6 shows the total access time versus the percentage objects in the hot tier. The following observations can be made from the given graph:

- The access time decreases as the hot percentage is increased. This is the expected result which was not demonstrated by the policy which calculated min-freq using standard deviation.
- Also, this policy can be seen as a self evolving policy, as the mean was recalculated for every interval of five hundred requests. Hence, we can conclude that this policy can adapt itself to the dynamic workloads.
- Another important observation made from the plot was that the access time did not show any drastic improvement on increasing the hot percentage from 40 % to 100 %. Therefore, some objects can be redistributed to cold tiers for this workload without heavily impacting the performance of the system. This can result in cost reductions.

Thus, this policy seems to be more promising candidate for the real-world like request distributions.

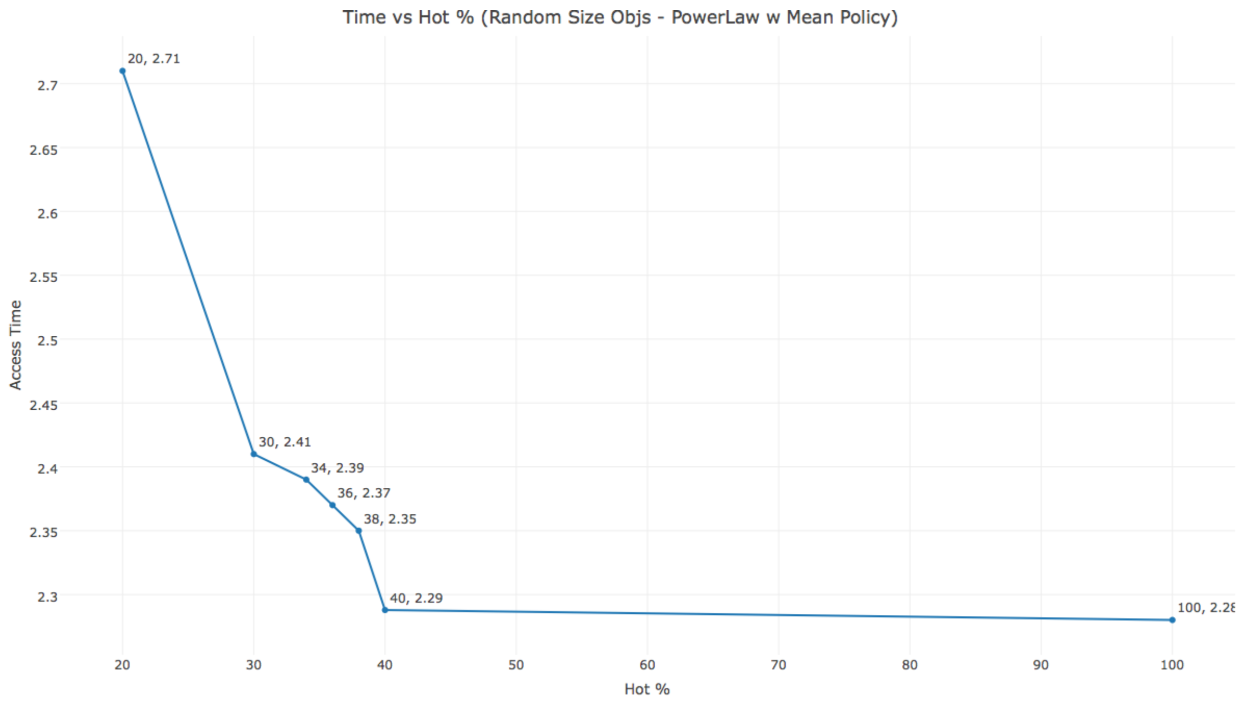
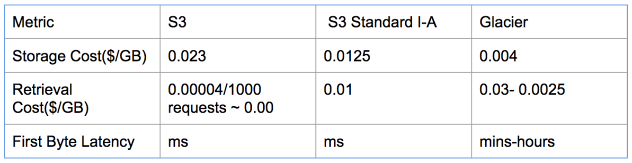


Figure 6: Powerlaw with Mean Policy

3. Integrating YALM with AWS

In the second phase of our project, we decided to apply the knowledge gained from the simulations to a public cloud. AWS S3 object store was the choice of public cloud we went for, since it offers a python client to directly access the objects in the cloud and has a large community and support base. S3 offers multiple classes of cloud storage as can be seen in Figure 7. We decided to build a double tiered system using S3 as the hot tier and Glacier as the cold tier. Standard I-A was not chosen as cold tier due to inability of the python client to restore objects from I-A to S3.



Metric	S3	S3 Standard I-A	Glacier
Storage Cost($/GB)	0.023	0.0125	0.004
Retrieval Cost($/GB)	0.00004/1000 requests ~ 0.00	0.01	0.03- 0.0025
First Byte Latency	ms	ms	mins-hours

Figure 7: Storage classes offered by AWS S3

3.1 Test Bench

We used Amazon RDS to store the metadata of the objects in the cloud. Python Boto [6] was used to interact with the cloud and move around the objects. Further, R was used to study the policies and their results. Also, one major problem encountered in moving objects from Glacier to S3 was the lack of visual interface for Glacier and high retrieval time. Therefore, to facilitate this experiment we created two buckets in S3; one was considered as a hot tier and other was considered as the cold tier. Whenever objects were retrieved from the cold bucket , they were first moved to the hot bucket and also penalized with a latency of 300 seconds (similar to expedited retrieval latency) and retrieval cost corresponding to Glacier. This was done to imitate the Glacier behaviour. We created 1000 variable sized binary files with random content and generated object requests using Power Law Distribution as described earlier.

3.2 Policies

3.2.1 Basic Policy

As mentioned earlier, we do not need to have all the objects in Hot Tier. We can reduce the cost by keeping some objects in the Cold Tier. So, for our first policy we decided to kick some of the data from Hot Tier to Cold Tier. We developed a policy called "Basic Policy" which would run after every 500 requests to move data from Hot Tier to Cold Tier.

We define a **movement index** mi to determine candidates to evict from Hot Tier.

$$mi = \frac{size\, of\, Object}{access\, Frequency\, of\, Object\, in\, last\, 500\, requests}$$

Then, we calculate mi_{mean} which is mean of mi of all objects accessed in last 500 requests. We then move objects having $mi > mi_{mean}$ to Cold Tier.

size	accessFrequency	**mi**
high	low	**high (Move to cold in order to reduce cost)**
high	high	neutral
low	low	neutral
low	high	**low (Good to keep in Hot)**

Table 1: Basic Policy Movement Index

Pre-Fetching

Overall storage cost could be reduced by moving the less accessed objects to Glacier but this should not be done by deteriorating the overall access time of the system. We had to optimize for the access time of the objects in the cold tier. Since, each object access in Glacier encounters an additional first byte latency which is quite high, we decided to make our policies learn from the previous access patterns and predict the objects in the cold tier which are likely to be used in the near future. We bundled all these predicted requests into one big request (Bulk Request)and prefetched the objects from the cold tier to the hot tier even before they were requested. This approach is similar to disk prefecthing and caching and provided us with multiple advantages. If the prefetched object was actually accessed by the user, he/she does not have to experience a delay in the retrieval. Also, for multiple objects we experience a first byte latency of 300 ms only once.

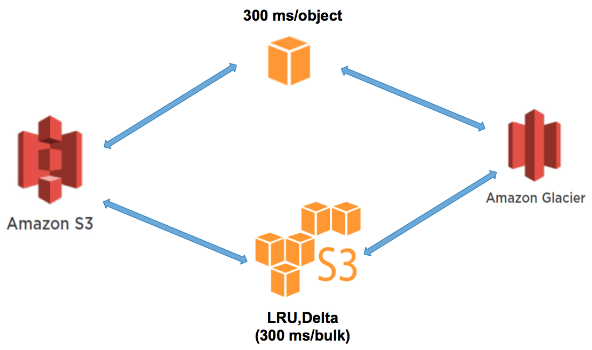


Figure 8: Prefetching from Glacier to S3

3.2.2 LRU Policy

The second policy we implemented was Least Recently Used. At the end of every 500 requests, the system chose 10% of the least recently used objects and these objects qualified as the prefetching candidate. These were fetched from Glacier into S3 as a part of a Bulk Move Request. After the end of every interval, the bulk request was re-evaluated and executed.

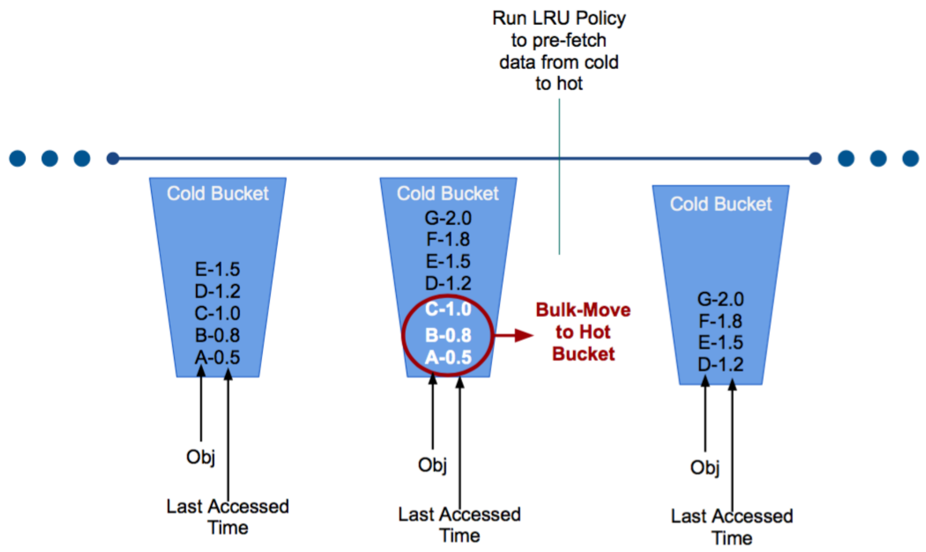


Figure 9: Prefetching from Glacier to S3 using LRU Policy

3.2.3 Delta Policy

After the LRU policy yielded better results than the Basic Policy, we decided to go with another variation of prefetching policy. Here, the objects which qualified to be a part of Bulk Move Request were predicted using the difference in the time of the last two accesses known as Δ. At the beginning of each interval a threshold of Δ value was calculated by averaging the Δ values of all the cold requests in the preceding interval. Here, the cold requests are the requests which were serviced from the cold tier. The 10% of the objects in the cold tier which had $\Delta > \Delta_{mean}$ were made a part of the Bulk Move Request.

$$\Delta_{mean} = \frac{\Delta_P + \Delta_Q + \cdots + \Delta_R + \Delta_S}{n}$$

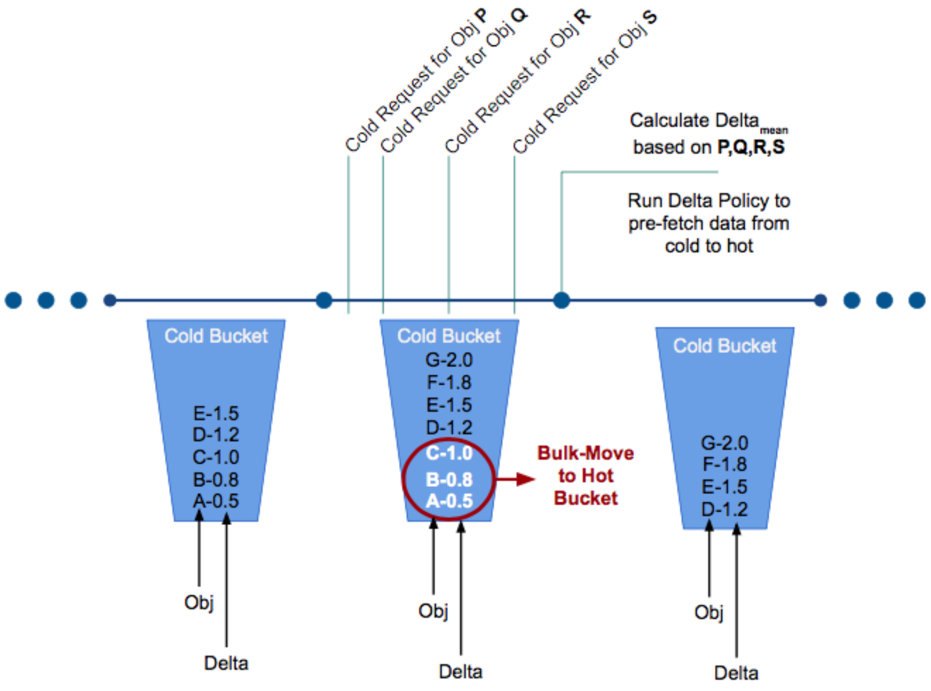


Figure 10: Prefetching from Glacier to S3 using Delta Policy

4. Results

4.1 Basic Policy

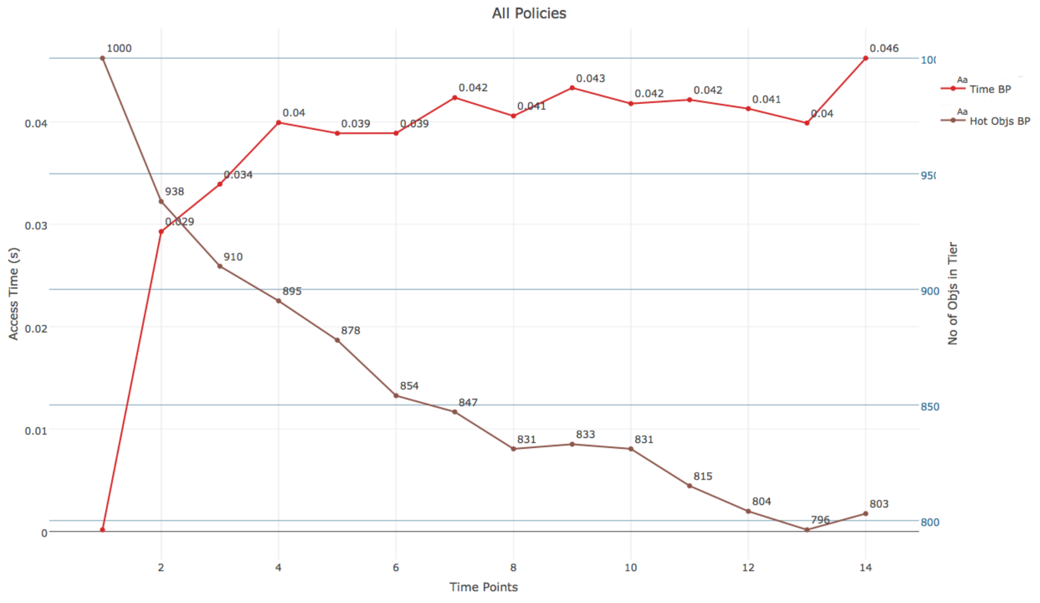


Figure 11: Basic Policy : Time vs No. of Objs

From figure 11 we can see that even if we reduce number of objects in Hot tier, time does not get affected much.

4.2 Basic Policy vs. Pre-fetching Policies (LRU Policy & Delta Policy)

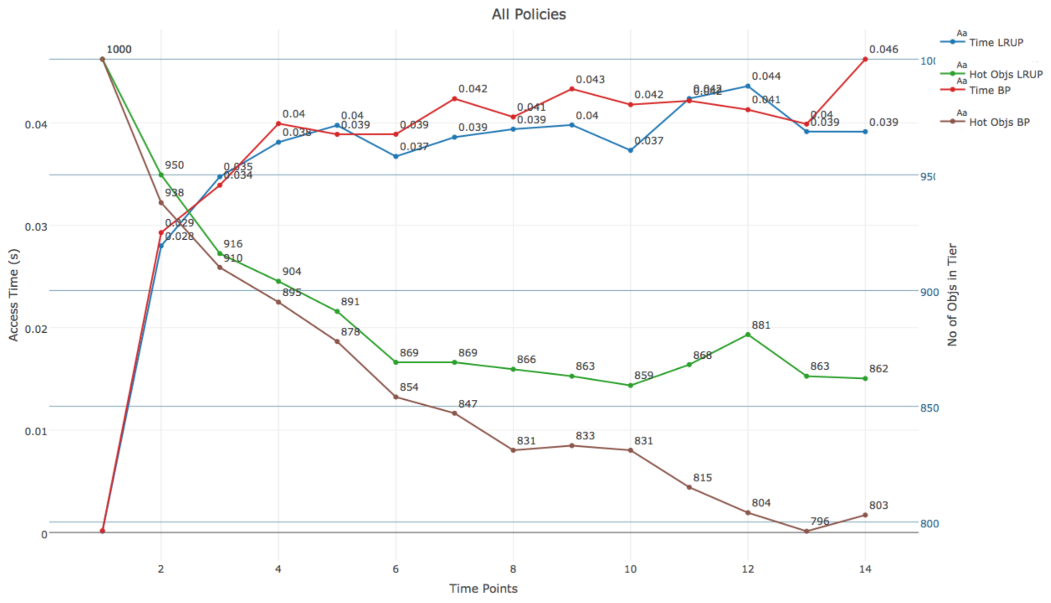


Figure 12: Time vs No. of Objs for Basic Policy (BP) and LRU Policy (LRUP)

As shown in figure 12, pre-fetching objects does reduce access time. Though, it keeps more objects in Hot tier which incurs more cost.

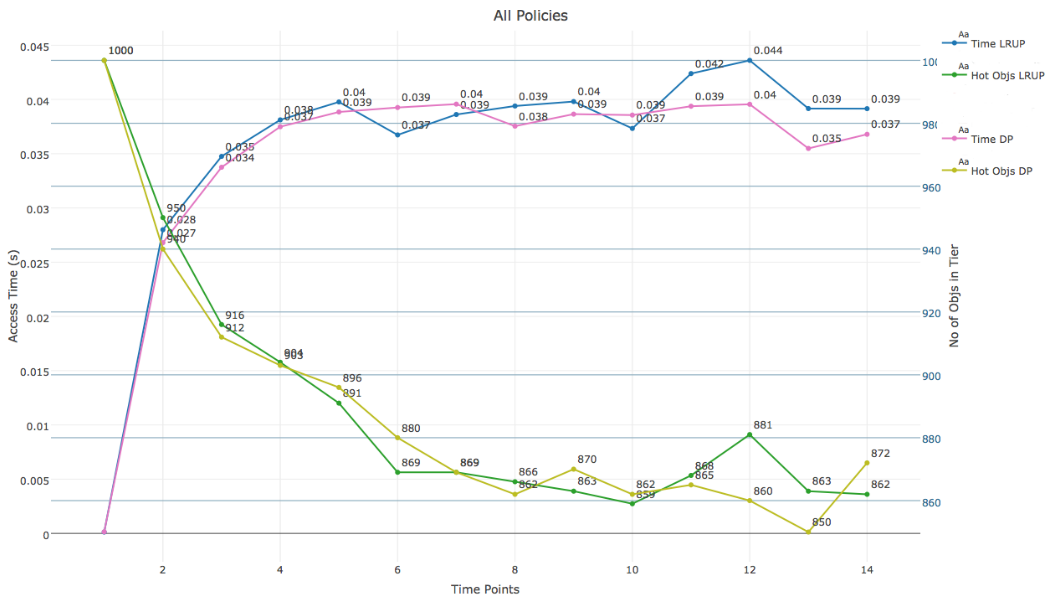


Figure 13: Time vs No. of Objs for LRU Policy (LRUP) and Delta Policy (DP)

As mentioned earlier, we believe that Delta policy is smarter than LRU Policy. We can see from figure 13 that even though LRU Policy and Delta Policy almost has same number of objects in Hot tier but Delta Policy results in much better access time.

Comparison of Individual Cold Requests

Below we show number of individual cold request (as defined in 3.2.3) during the period of 500 requests.

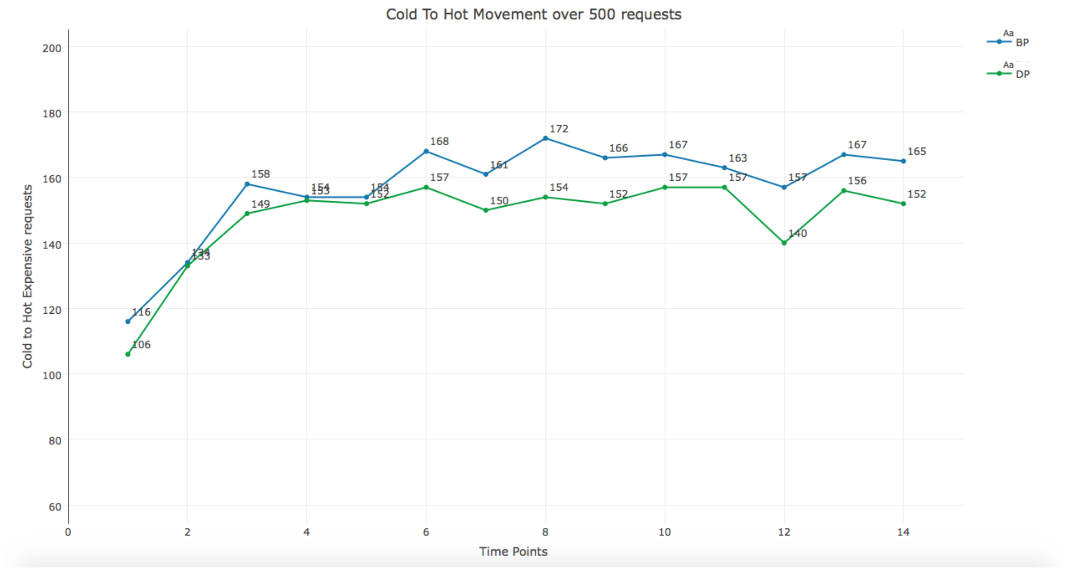


Figure 14: Individual Cold Requests: BP vs. DP

From figure 14 we can say that Delta Policy incurs much less individual cold requests than Basic Policy which does not do pre-fetching from Cold to Hot tier.

Figure 15 shows that Delta Policy is better than LRU Policy. Figure 16 shows comparision of all three policies.

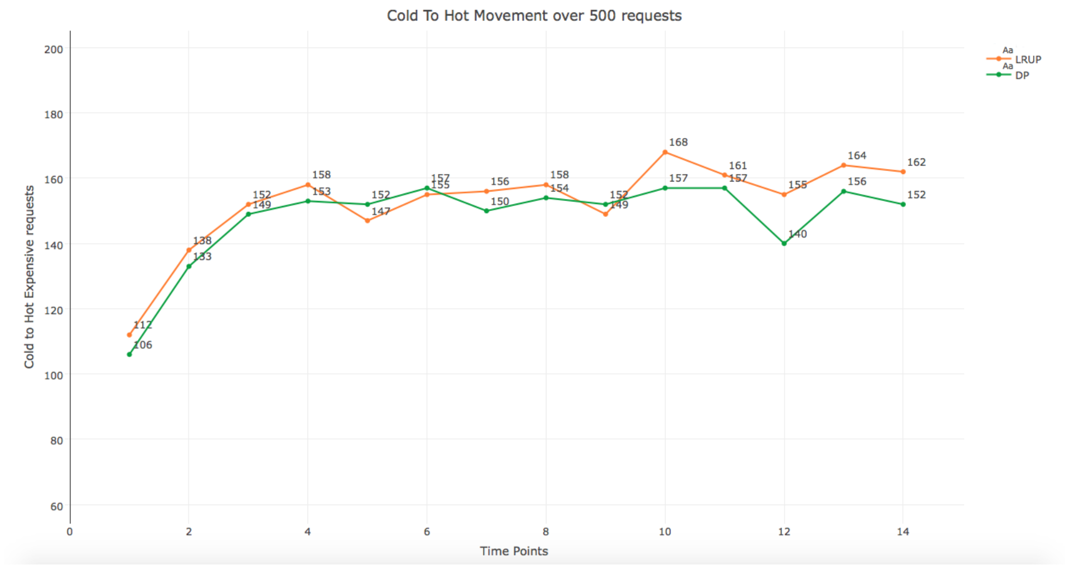


Figure 15: Individual Cold Requests: LRUP vs. DP

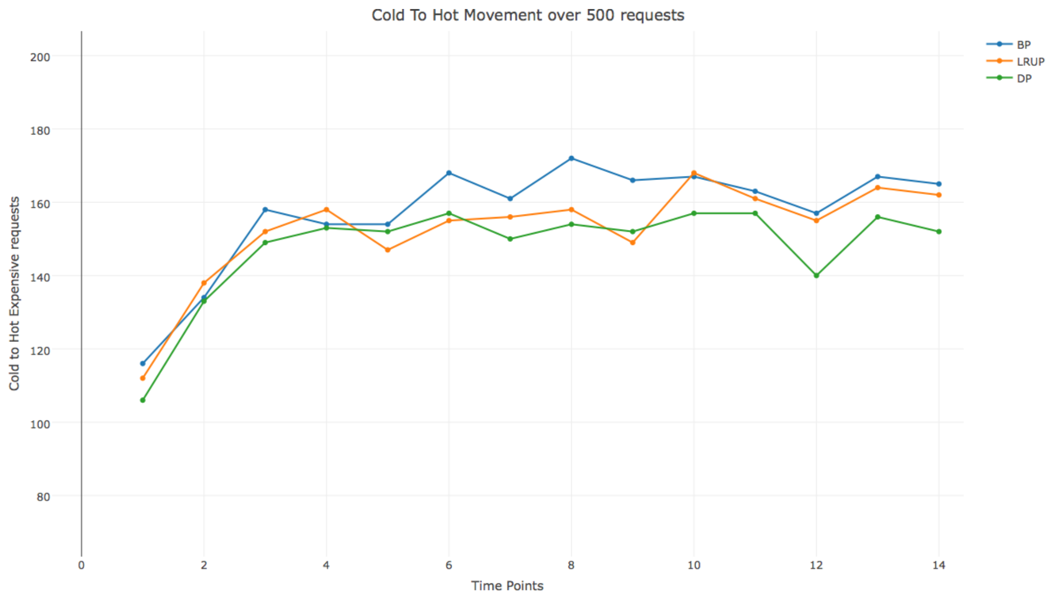


Figure 16: Individual Cold Requests:Delta Policy, LRU Policy and Basic Policy

5. Future and Related Work

In future we wish to record more parameters corresponding to the workloads so that our policies can assess the workload and its type better. Also, since the retrieval cost of Glacier is on the higher side [7], we can extend our system to support multiple cloud vendors. For example the cold tiers provided by Microsoft Azure has lesser retrieval cost [8] than Glacier. There is a vast scope of improvement in the Machine Learning Engine of YALM, we can deploy neural net for the system to learn overtime and come up with better predictions. Also, a multiverse of policies can be tested to achieve better results.

There have been many studies stressing on the importance of the information lifecycle management given the trends of data growth. As of now our system is not capable of evicting/deleting the objects from both the tiers. These studies lay a great deal of importance on this process and our system can be further extended to include this feature.

6. Conclusion

A good balancing point between storage cost and access time can be achieved by studying the workloads. The simulation phase of the project help us understand the importance of coming up with good policies to achieve expected results. All this could be done without bearing the costs of commercial cloud. The refined knowledge could further be applied to the public cloud in phase 2. Prefetching improved the results by a considerable amount and Delta Policy proved to be a promising candidate. If smart and self- evolving policies can be applied, we are sure to achieve cost deductions without compromising on the performance of the system.

7. Acknowledgement

This project is a result great efforts and teamwork. However, this would not have been possible without the kind support of Dr. Vincent Freeh. Brainstorming sessions with Prof. Freeh helped us conceptualize the idea better and go out of our comfort zone to deliver more. We would also like to thank Mr. Brian Beach, his interactive guest lecture inspired us to work on this idea.

Links

- GitHub Repository
- YALM Smart Policies
- Individual Fetches (Cold Requests)

References

[1] Azure storage. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/.

[2] Google storage. https://cloud.google.com/storage/pricing-summary/.

[3] Aws object lifecycle management. https://docs.aws.amazon.com/AmazonS3/latest/dev/object-lifecycle-mgmt.html.

[4] Power law. https://en.wikipedia.org/wiki/Power_law.

[5] scipy stats documentation. https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.stats.powerlaw.html.

[6] Boto 3 documentation. http://boto3.readthedocs.io/en/latest/.

[7] Amazon glacier pricing. https://aws.amazon.com/glacier/pricing/.

[8] Azure pricing. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/.