CS553_HW3_Report

Fan, Zexiang (A20313247)
Liu, Xin (A20314819)
Wang, Dingwen (A20304858)

(To use our application, please visit http://qualified-cacao-745.appspot.com/)

Design

Overall program design:

The program we developed is a cached distributed storage system based on Google App Engine.

Programming tools:

- The language we use is Java, JavaScript, and HTML.
- The libraries we include are Google App Engine SDK (including BlobStore API, DataStore API, and Memcache API) and Google Cloud Storage Client Library.
- The IDEs we use are Maven 3.1.1 and Eclipse Luna (to edit the imported maven project).

Program architecture: the program is a MVC architecture, where:

- Model is GCS objects (files in GCS), Memcache objects (small files in Memcache), and DataStore objects (metadata of the files in GCS stored in DataStore)
- Controller is Java Servlet classes.
- View is HTML pages integrated in JavaScript files.

Functionalities and implementations:

1. **Boolean = Insert (key, value)**: A Client uploads a list of files(HTTP Post request) to the server.

   The servlet decodes the request's header(key) and contents(value), generates a GCS object with the name = key and the contents = value, then store it into GCS.

   If the size of the contents is less than 100KB, the servlets generate another Memcache object (same key and value) and stores into Memcache.

   Then, the servlet stores the file information (file name and file size) into a DataStore object and stores it into DataStore.

   At last, the servlet send a HTTP response to client, indicating whether the store operation succeed.
2. **Boolean = Check (key)**: A client provides a filename(HTTP Post request) to the server.

   The servlet get the filename, make a query in DataStore with object name = filename, if there is a object, respond to client a "true" message; otherwise respond a "false" message.

   If "true" in the above step, make another query in the memcache, and respond to client accordingly.

   (The algorithm above is based on the fact that there CANNOT be an object that exists in memcache but don't have a counterpart in GCS)

   (This function also implement the Boolean = checkCache(key), which is in EXTRA CREDIT)
3. **Value = Find (key)** A client provides a filename(HTTP Get request) to the server.

   The servlet get the filename, make a query in DataStore with object name = filename, if there isn't such an object, send an "error" message to client. Otherwise, make another query in Memcache,

   If there is such an object, read this object and generate it as a HTTP response to serve to client.

   If there isn't such an object in Memcache, read from GCS use the object key stored in DataStore, and use BlobStore API to serve this GCS object.

4. **Boolean = Remove(key)** A client provides a filename(HTTP Post request) to the server.

   The servlet get the filename, make a query in DataStore with object name = filename, if there isn't such an object, send an "error" message to client.

   Otherwise, the servlet deletes the GCS object with this filename, and deletes corresponding DataStore object, which contains the metadata of the GCS object.

   Then, the servlet make another query in Memcache, if there is such a file, delete it.

   Then, the servlet send an "succeed" message to the client.

5. **Key[] = Listing()** A client asks the server to list all files(HTTP Post request).

   The servlet lists all the filenames stored in the DataStore objects(they are the metadata of the files in GCS), and shows the lists to the client.

EXTRA CREDITS

Note: The algorithms of the extra functionalities are similar to the basic ones, some of them are based on the basic ones.

1. **Boolean = Check(key)** and **Boolean = CheckCache(key)** are already implemented in (2).
2. **Boolean = RemoveAllCache()** is implemented by simply calling a "MemcacheService.clearAll()" method defined in Memcache API.
3. **Boolean = RemoveAll()** is implemented by listing all the filenames, and remove each one in GCS, also remove the metadata in DataStore, and finally clear the Memcache.
4. **Double = CacheSizeMB()** and **Double = CacheSizeElem()** are implement by simply calling a "MemcacheService.getStatistics()" and using the Stat object to get the #elements and #bytes information.
5. **Double = StorageSizeMB()** and **Double = StorageSizeElem()** are implemented by listing all the DataStore object, counting the number of objects, and sum up all the "size" properties of the objects.
6. **Boolean = FindInFile(key, string)** and **Key[] = Listing(string)** are implemented using the Java standard package "regex" to match the pattern to the filenames.

Design tradeoffs and Difficulties:

- Uploading big files

To implement function (1), i.e. upload files, there is an "Incoming Bandwidth" limitation of GAE, that each HTTP request can be no larger than 32 MB.

To solve this problem, the client side's JavaScript needs to partition the large file(the 100MB one) into smaller HTTP Post request, and let one final GET request command the GcsService.compose() method. Given that GAE only supports Servlet version 2.5, which don't provide direct method to carve out multipart/form-data content from an http request, hand writing this function is error prone. But we managed to store exactly the same content uploaded, and we made sure that the result of a GCS compose operation results in the exact file.

- Downloading big files

To implement function (3), there is also a 32MB HTTP respond size limitation. To solve this, a BlobstoreService.serve() method is used to help client downloading large files. This mechanism requires to create a BlobKey with a GCS object name, using the "BlobstoreService.createGSkey()" method.

- Downloading benchmark

Problem is that JavaScript can receive the data as response from the server, but it cannot save them as files to local file system because of its security permission. In our benchmark, we use JavaScript to get the response data, and show size of these data in bytes, and discard it.

- Multithreading

  Since the benchmark needs to measure the improvement of multithreading, the JavaScript on the client side has a functionality that partition a big number of HTTP request into four groups, and create four threads to send each group of requests. The Java Servlet Container will automatically handle this multithreading itself.

  We first implement the 1 thread version. To make request sequentially actually as not a natural way to express in JavaScript. But by embedding recursive function call in within req.onreadystatechange, we manage to control that a request can only be made after a previous successful request.

  Following the 1 thread version, implementing the 4 thread version was actually much simpler. Our JavaScript program first divide all works into 4 lists, and pass each of the lists to on 1 thread to execute.

- Memcache switch

  Since the benchmark needs to measure the improvement of caching technique, the client HTTP page has a option to choose whether to enable memcache. If it’s checked, the uploading, downloading and removing processes are different from the ones without caching.

Possible improvements and extensions

The largest limitation of the program is that it can only handle text file uploading right now, due to the limitation of our Servlet decoding HTTP request technique. To fix this, we need to implement other techniques to handle Post request that contains images or videos.

Manual

How to compile the program

Having our source code (Servlet class file, index.html, web.xml and pom.xml), the compilation is followed by the official google website of GAE tutorial.

1. Downloading Java(version 7) and Maven(version 3.1.0 or above)
2. Create a maven project with architype = google app engine skeleton
3. In directory /src/main/java create a directory /com/google/test/
4. Copy all the Servlet class files into src/main/java/com/google/test/
5. Copy the web.xml into src/main/webapp/WEB-INF/
6. Copy the index.html into src/main/webapp/
7. Copy the pom.xml into the project root directory
8. Execute the following command in the project root directory

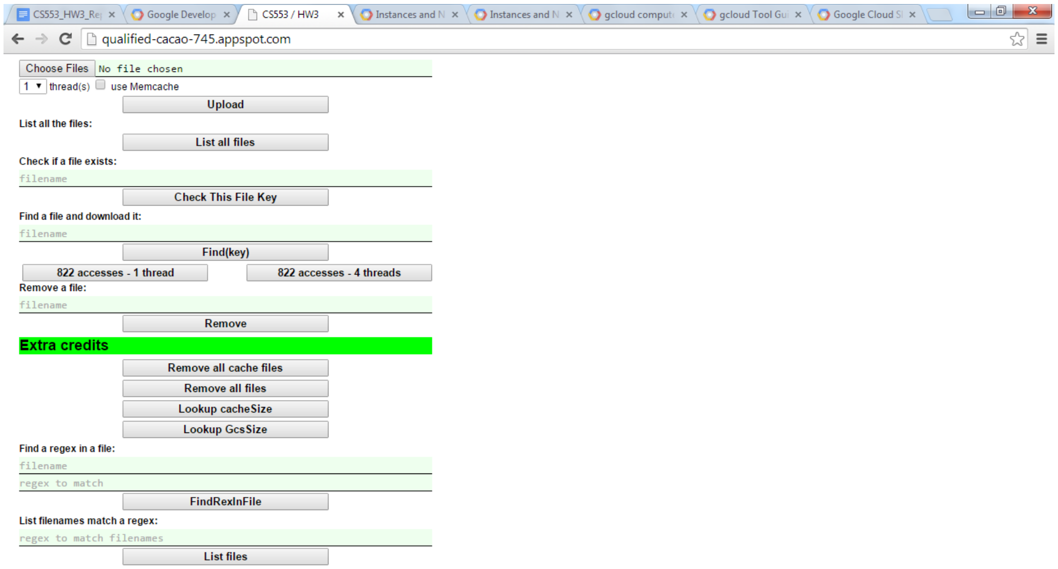
```
mvn clean install
mvn appengine:update
```

Now, the program should be available on http://qualified-cacao-745.appspot.com/

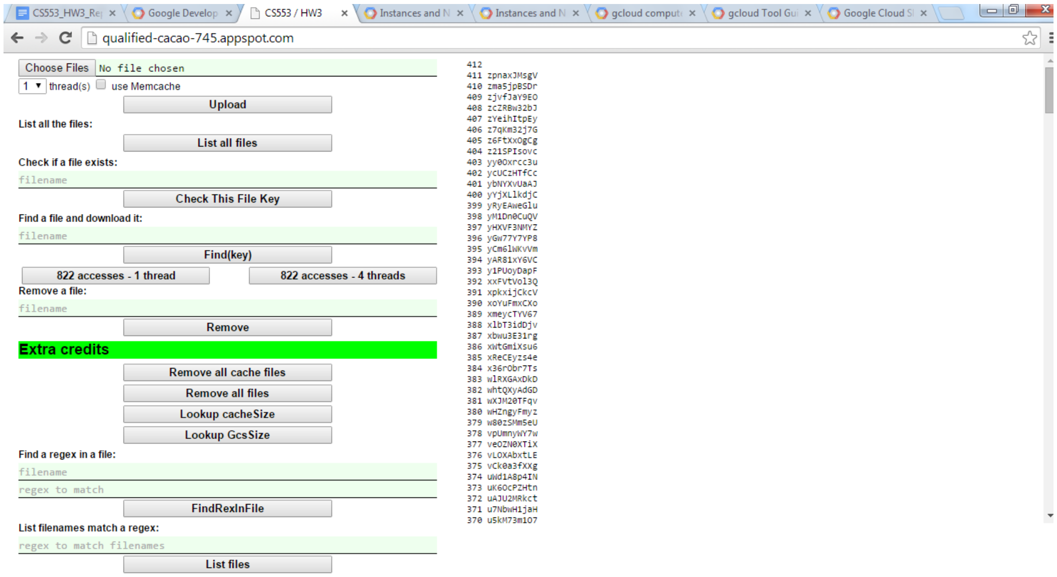
How to run the program

When the user access the program website(http://qualified-cacao-745.appspot.com/), the list of operations appears on the left hand side, the right hand side will show the result of each operation invoked by the user.

- HomePage (Control panel on the left, output log panel on the right)



- Example: list all files operation



```
412
411 zpnaxJMsgV
410 zma5jpBSDr
409 zjvfJaY9EO
408 zcZRBw32bJ
407 zYeihItpEy
406 z7qKm32j7G
405 z6FtXxOgCg
404 z21SPIsovc
403 yy0Oxrcc3u
402 ycUCzHTfCc
401 ybNYXvUaAJ
400 yYjXLlkdjC
399 yRyEAweGlu
398 yM1Dn0CuQV
397 yHXVF3NMYZ
396 yGw77Y7YP8
395 yCm6lWKvVm
394 yAR81xY6VC
393 y1PUoyDapF
392 xxFVtVol3Q
391 xpkxijCkcV
390 xoYuFmxCXo
389 xmeycTYV67
388 xlbT3idDjv
387 xbwu3E31rg
386 xWtGmiXsu6
385 xReCEyzs4e
384 x36rObr7Ts
383 wlRXGAxDkD
382 whtQXyAdGD
381 wXJM20TFqv
380 wHZngyFmyz
379 w80zSMm5eU
378 vpUmnyWY7w
377 veOZN0XTiX
376 vLOXAbxtLE
375 vCk0a3fXXg
374 uWd1A8p4IN
373 uK6OcPZHtn
372 uAJU2MRkct
371 u7NbwH1jaH
370 u5kM73m1O7
```

- Upload files

Click the “Choose File” button, choose files to upload, and click “Upload” button. (For now, we only support text file upload.)

- List files

Click the “List all files” button.

- Check a file

Type the filename in the textbox above, then click “Check This File Key” button.

- Download a file

Type the filename in the textbox above, then click “Find(key)”

- Remove a file

Type the filename in the textbox above, then click “Remove” button.

The **extra functionalities** are listed below the “Extra credits” line, user can use these functions by click the corresponding buttons, especially:

- To see if a filename contains a certain *regular expression*, a user can type the filename and an regular expression (e.g. ^\d, ^[A-Z], etc), then click “FindRexInFIle” button.
- To see the list of filenames that contain a certain regular expression, a user can type the regular expression in the textbox, then click “List files” button.

Performance benchmarks

Experiment dataset, and how we generated it

The dataset includes 411 files spanning 311MB of data, as described by the PA3 sheet.

We implemented a Java program to generate the files. The program has a sample string, containing the character eligible in the files. A random number generator is declared using the `java.util.random` package. Then, for each file to generate, the number of lines is calculated by size / 100 byte. For some sizes that are not multiple of 100, the module is calculated as the number of characters in the last line.

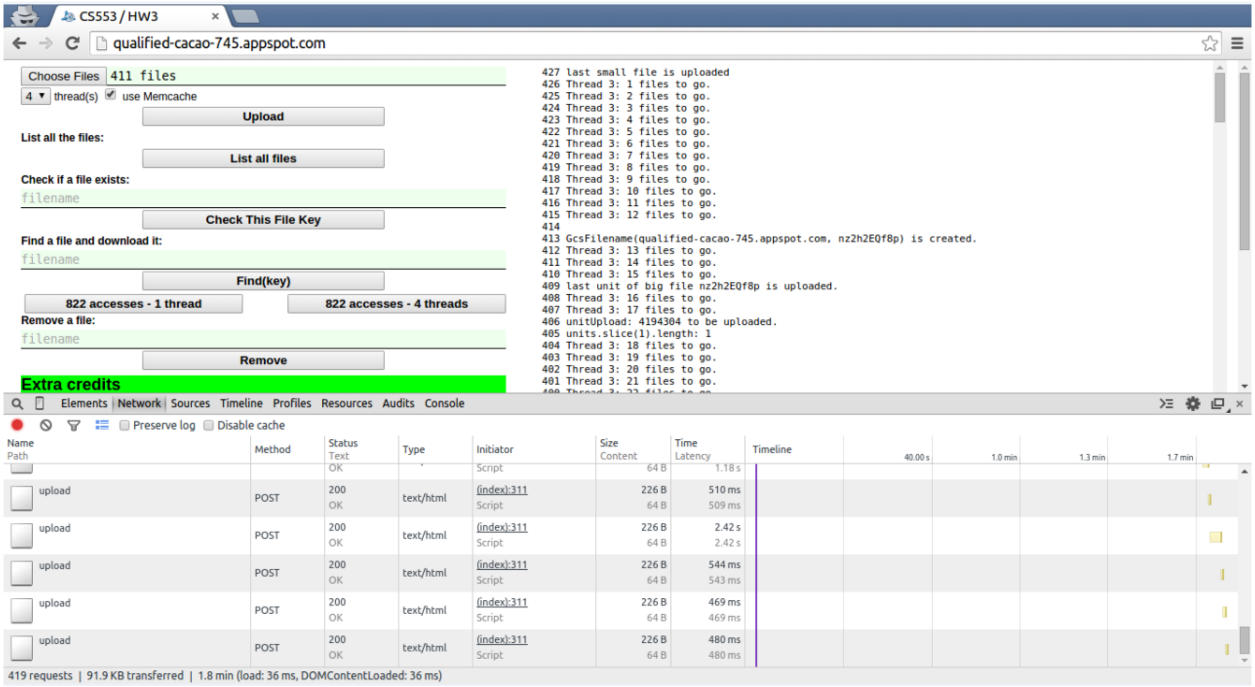
The filenames for these files are generated in a similar way.

How we run the experiments

1. Upload All 411 files

Since the HTML form has a multiple upload function, no further implement is needed. Simply invoke insert() operation, and choose all the files will complete this experiment.

Experiment: 4 threads with MemCache, upload



Choose Files: 411 files

4 thread(s) ☑ use Memcache

```
427 last small file is uploaded
426 Thread 3: 1 files to go.
425 Thread 3: 2 files to go.
424 Thread 3: 3 files to go.
423 Thread 3: 4 files to go.
422 Thread 3: 5 files to go.
421 Thread 3: 6 files to go.
420 Thread 3: 7 files to go.
419 Thread 3: 8 files to go.
418 Thread 3: 9 files to go.
417 Thread 3: 10 files to go.
416 Thread 3: 11 files to go.
415 Thread 3: 12 files to go.
414
413 GcsFilename(qualified-cacao-745.appspot.com, nz2h2EQf8p) is created.
412 Thread 3: 13 files to go.
411 Thread 3: 14 files to go.
410 Thread 3: 15 files to go.
409 last unit of big file nz2h2EQf8p is uploaded.
408 Thread 3: 16 files to go.
407 Thread 3: 17 files to go.
406 unitUpload: 4194304 to be uploaded.
405 units.slice(1).length: 1
404 Thread 3: 18 files to go.
403 Thread 3: 19 files to go.
402 Thread 3: 20 files to go.
401 Thread 3: 21 files to go.
```

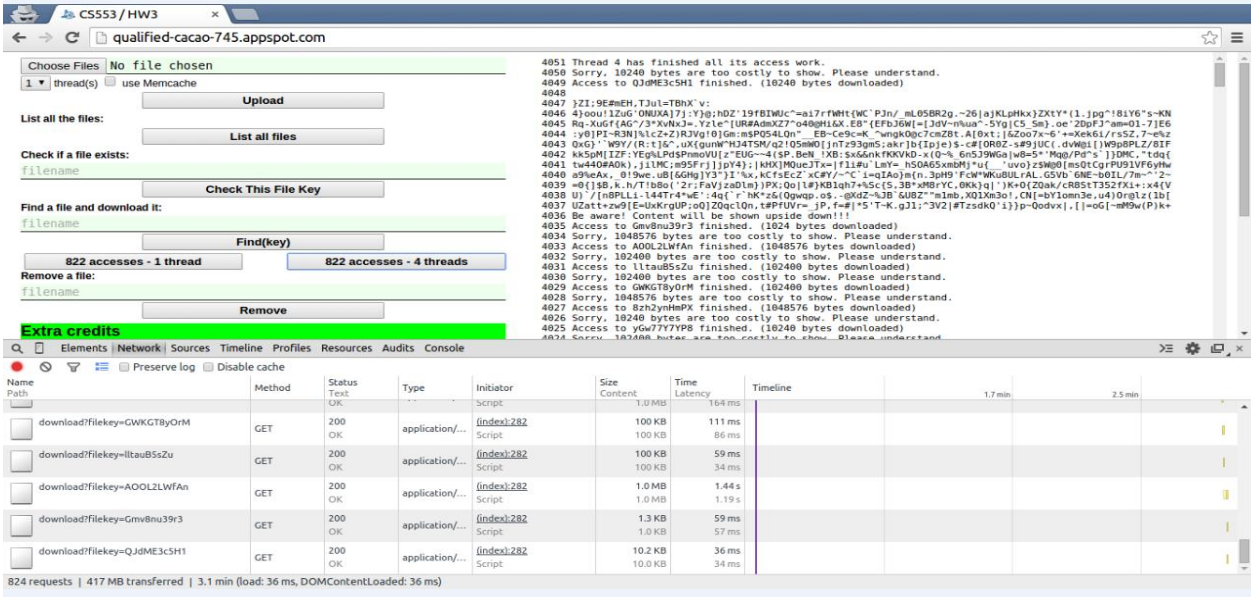
Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
		OK		Script	64 B	1.18 s	
upload	POST	200 OK	text/html	(index):311 Script	226 B 64 B	510 ms 509 ms	
upload	POST	200 OK	text/html	(index):311 Script	226 B 64 B	2.42 s 2.42 s	
upload	POST	200 OK	text/html	(index):311 Script	226 B 64 B	544 ms 543 ms	
upload	POST	200 OK	text/html	(index):311 Script	226 B 64 B	469 ms 469 ms	
upload	POST	200 OK	text/html	(index):311 Script	226 B 64 B	480 ms 480 ms	

419 requests | 91.9 KB transferred | 1.8 min (load: 36 ms, DOMContentLoaded: 36 ms)

2. Download 822 times with random filename generating

To generate random filename, first we list all the filenames stored in GCS, then make random choice on this sample. Generate 822 times will give us an array of 822 filenames(with some duplication). Then invoke the find() operation use this array.

Experiment: 4 threads with no MemCache, download



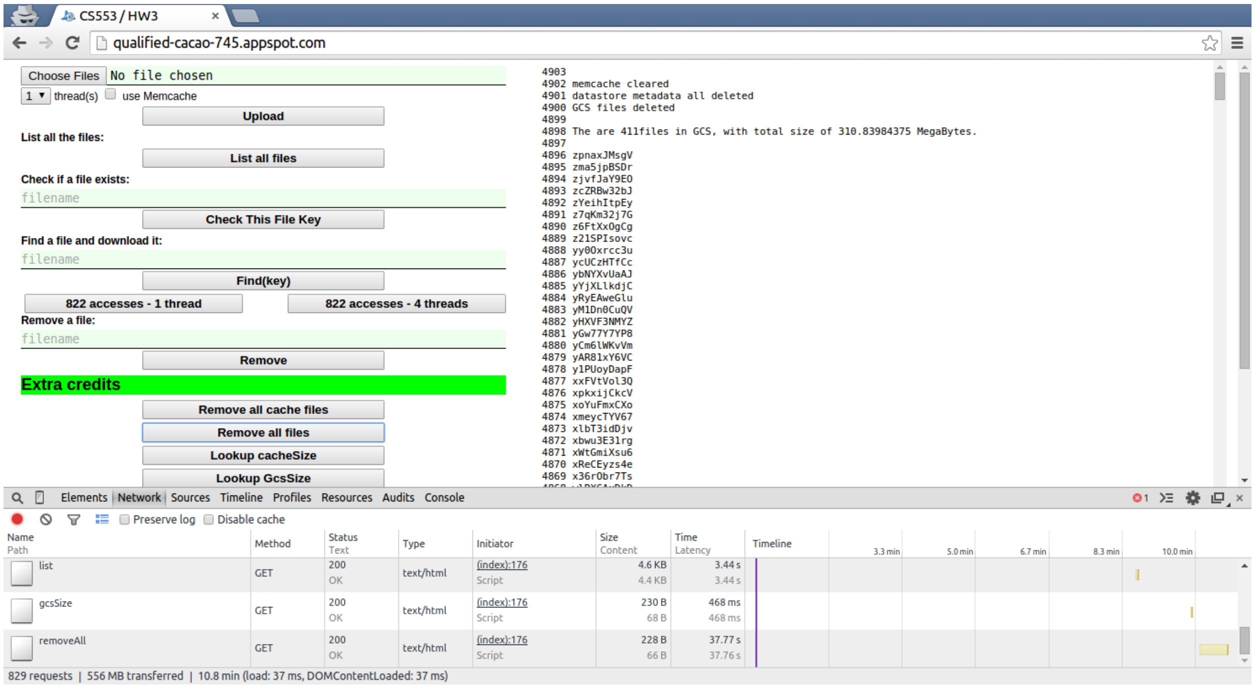
```
4051 Thread 4 has finished all its access work.
4050 Sorry, 10240 bytes are too costly to show. Please understand.
4049 Access to QJdME3c5H1 finished. (10240 bytes downloaded)
4048
4047 }ZI;9E#mEH,TJul=TBhX`v:
4046 4}oou!1ZuG'ONUXA]7j:Y}@;hDZ'19fBIWUc^=ai7rfWHt{WC`PJn/_mL05BR2g.~26|ajKLpHkx}ZXtY*(1.jpg^!8iY6"s~KN
4045 Rq-XuGf{AG^/3*XvNxJ=.Yzle^[UR#AdmXZ7^o40@Hi&X.E8"{EFbJ6W[=[JdV~n%ua^-5Yg|C5_Sm}.oe'2DpFJ^am=O1-7]E6
4044 :y0]PI~R3N]%lcZ+Z)RJVg!0]Gm:m$PQ54LQn"__EB~Ce9c=K_^wngkO@c7cmZ8t.A[0xt;|&Zoo7x~6'+=Xek6i/rsSZ,7~e%z
4043 QxG}'`W9Y/(R:t]&^,uX{gunW^HJ4TSM/q2!Q5mWO[jnTz93gmS;akr]b{Ipje)$-c#[OR0Z-s#9jUC(.dvW@i[)W9p8PLZ/8IF
4042 kk5pM[IZF:YEg%LPd$PnmoVU[z"EUG~~4($P.BeN_!XB:$x&&nkfKKVkD-x(Q~%_6n5J9WGa|w8=5*'Mq@/Pd^s`]}DMC,"tdq{
4041 tw44O#AOk),jilMC;m95Frj]jpY4};|kHX]MQueJTx=|fli#u`LmY=_hSOA65xmbMj*u{__'uvo}z$W@0[msQtCgrPU91VF6yHw
4040 a9%eAx,_0!9we.uB[&GHg]Y3"}I'%x,kCfsEcZ`xC#Y/~^C`i=qIAo}m{n.3pH9'FcW*WKu8ULrAL.G5Vb`6NE~b0IL/7m~^'2~
4039 =0{]$B,k.h/T!b8o('2r;FaVjzaDlm})PX;Qo|l#}KB1qh7+%Sc{S,3B*xM8rYC,0Kk}q|')K+O{ZQak/cR8StT352fXi+:x4{V
4038 U)`/[n8PLLi-l44Tr4*wE':4q{`r`hK*z&(Qgwqp.o$.-@XdZ~%JB`&U8Z""m1mb,XQ1Xm3o!,CN[=bY1omn3e,u4)Or@lz(1b[
4037 UZatt+zw9[E=UxKrgUP;oQ]ZQqclQn,t#PfUVr=_jP,f=#|*5'T~K.gJ1;^3V2|#TzsdkQ'i}}p~Qodvx|,[|=oG[~mM9w(P)k+
4036 Be aware! Content will be shown upside down!!!
4035 Access to Gmv8nu39r3 finished. (1024 bytes downloaded)
4034 Sorry, 1048576 bytes are too costly to show. Please understand.
4033 Access to AOOL2LWfAn finished. (1048576 bytes downloaded)
4032 Sorry, 102400 bytes are too costly to show. Please understand.
4031 Access to lltauB5sZu finished. (102400 bytes downloaded)
4030 Sorry, 102400 bytes are too costly to show. Please understand.
4029 Access to GWKGT8yOrM finished. (102400 bytes downloaded)
4028 Sorry, 1048576 bytes are too costly to show. Please understand.
4027 Access to 8zh2ynHmPX finished. (1048576 bytes downloaded)
4026 Sorry, 10240 bytes are too costly to show. Please understand.
4025 Access to yGw77Y7YP8 finished. (10240 bytes downloaded)
4024 Sorry, 102400 bytes are too costly to show. Please understand.
```

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
		OK		Script	1.0 MB	164 ms	
download?filekey=GWKGT8yOrM	GET	200 OK	application/...	(index):282 Script	100 KB 100 KB	111 ms 86 ms	
download?filekey=lltauB5sZu	GET	200 OK	application/...	(index):282 Script	100 KB 100 KB	59 ms 34 ms	
download?filekey=AOOL2LWfAn	GET	200 OK	application/...	(index):282 Script	1.0 MB 1.0 MB	1.44 s 1.19 s	
download?filekey=Gmv8nu39r3	GET	200 OK	application/...	(index):282 Script	1.3 KB 1.0 KB	59 ms 57 ms	
download?filekey=QJdME3c5H1	GET	200 OK	application/...	(index):282 Script	10.2 KB 10.0 KB	36 ms 34 ms	

824 requests | 417 MB transferred | 3.1 min (load: 36 ms, DOMContentLoaded: 36 ms)

3. Remove All 411 files

Since we already implement the RemoveAll() operation, simply invoke this operation one time will complete this experiment.



```
4903
4902 memcache cleared
4901 datastore metadata all deleted
4900 GCS files deleted
4899
4898 The are 411files in GCS, with total size of 310.83984375 MegaBytes.
4897
4896 zpnaxJMsgV
4895 zma5jpBSDr
4894 zjvfJaY9EO
4893 zcZRBw32bJ
4892 zYeihItpEy
4891 z7qKm32j7G
4890 z6FtXxOgCg
4889 z21SPIsovc
4888 yy0Oxrcc3u
4887 ycUCzHTfCc
4886 ybNYXvUaAJ
4885 yYjXLlkdjC
4884 yRyEAweGlu
4883 yM1Dn0CuQV
4882 yHXVF3NMYZ
4881 yGw77Y7YP8
4880 yCm6lWKvVm
4879 yAR81xY6VC
4878 y1PUoyDapF
4877 xxFVtVol3Q
4876 xpkxijCkcV
4875 xoYuFmxCXo
4874 xmeycTYV67
4873 xlbT3idDjv
4872 xbwu3E31rg
4871 xWtGmiXsu6
4870 xReCEyzs4e
4869 x36rObr7Ts
```

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
list	GET	200 OK	text/html	(index):176 Script	4.6 KB 4.4 KB	3.44 s 3.44 s	
gcsSize	GET	200 OK	text/html	(index):176 Script	230 B 68 B	468 ms 468 ms	
removeAll	GET	200 OK	text/html	(index):176 Script	228 B 66 B	37.77 s 37.76 s	

829 requests | 556 MB transferred | 10.8 min (load: 37 ms, DOMContentLoaded: 37 ms)

Result and analysis

(*Note*: given that the file access benchmark is generated randomly, each run results in different size of of data being transferred. Our metrics are not affected however, since

(File access) ***Latency*** *= Duration of Transfer ÷ 822 (File accesses)*

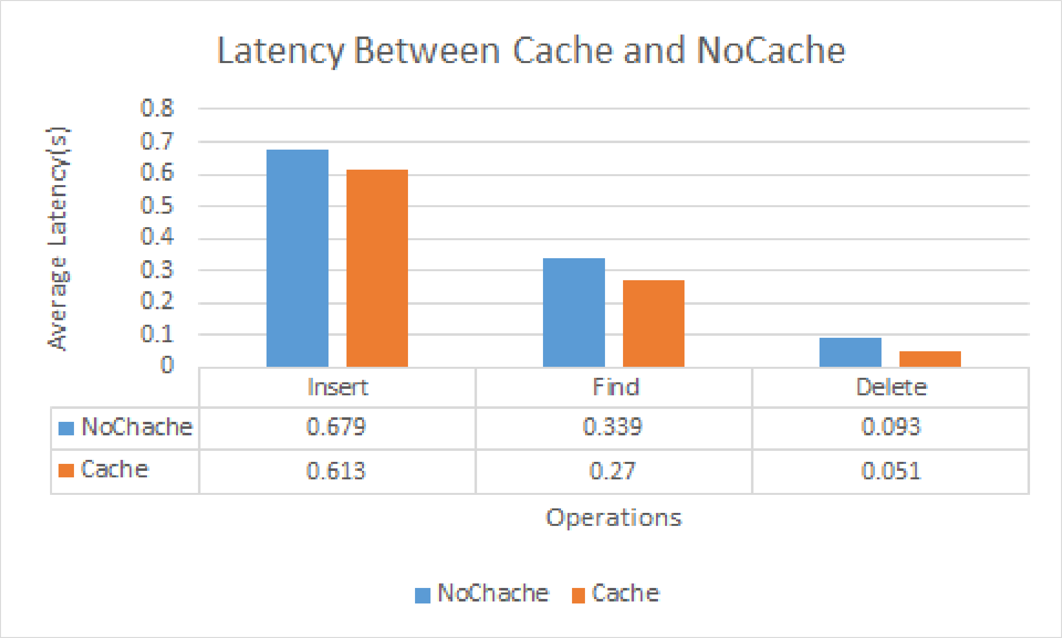
Throughput *= Amount of Data Transferred ÷ Duration of Transfer*

.)

Latency results:

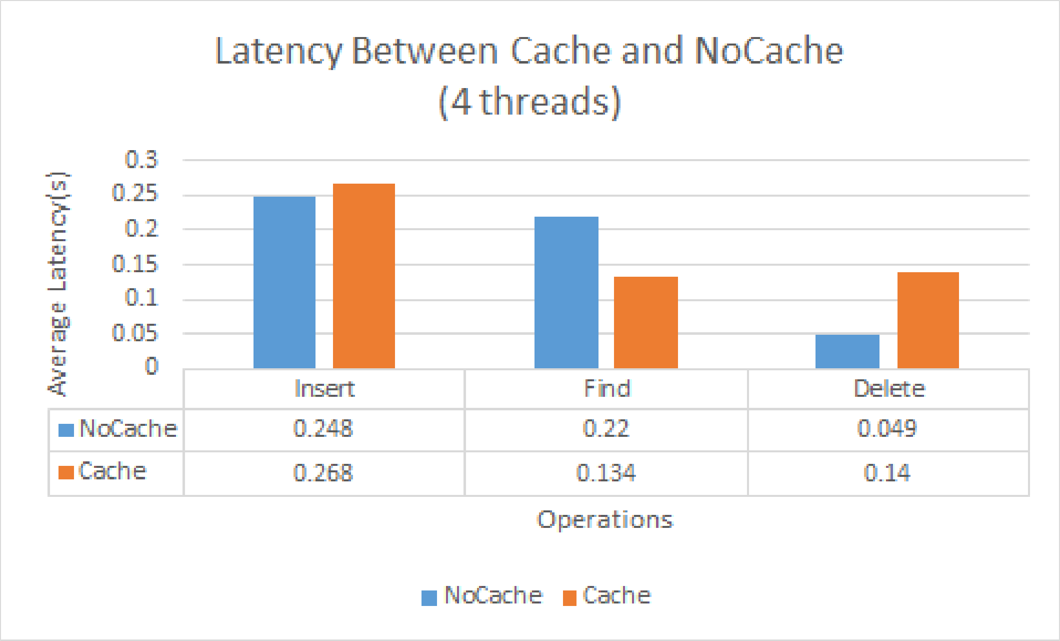
There are 12 experiments in total, as described by the PA3 sheet. The results are shown below:

1. **1 thread, with or without Memcache**



As is shown above, use of Memcache service provides quite some improvements on all of the three operations in terms of *latency*. Although in the case of running in single thread, inefficiency at the client side is high, Memcache still helps. However, the result in deletion is surprising considering that deletion operation with Memcache requires more work. (*Note*: throughout the experiments, we did found out that the deletion operation shew very inconsistent performance.)

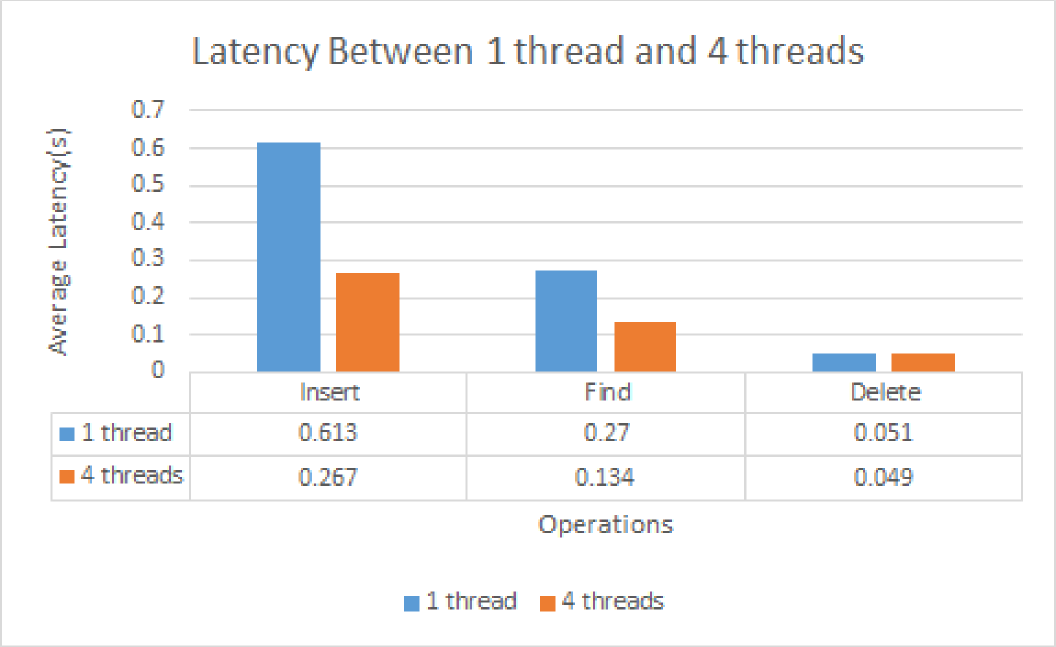
2. **4 threads, with or without Memcache**



	Insert	Find	Delete
🟦 NoCache	0.248	0.22	0.049
🟧 Cache	0.268	0.134	0.14

Another thing to notice here is that the deletion in the case of using Memcache uses less time is not reasonable, since by using Memcache, the deletion will require clearing of Memcache in addition to deletion of files in GCS and entities in Datastore, which should lead to worse performance when compared with the without-cache case. In our experiment, deletion without Memcache failed in its first try at 59.7 second. Then we pressed the list function which shows that all files are there (list according to Datastore records). When we try the second time, immediately after, the deletion without Memcache finished in 20 seconds.

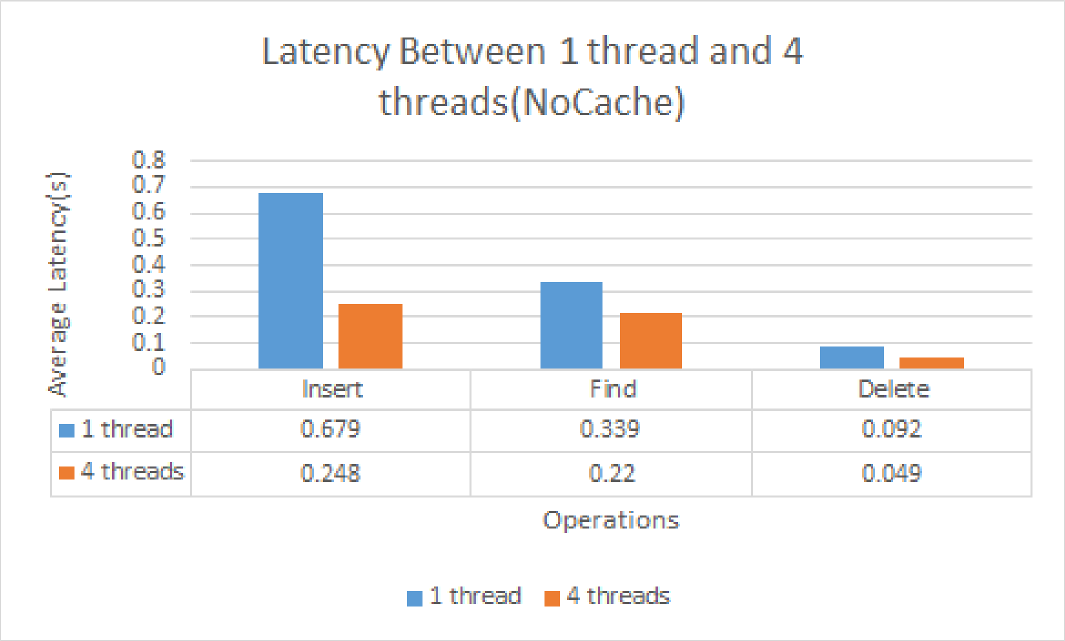
3. **With Memcache, 1 or 4 thread(s)**



	Insert	Find	Delete
🟦 1 thread	0.613	0.27	0.051
🟧 4 threads	0.267	0.134	0.049

As is shown above, multi-thread at the client side provides significant improvement. The major reason is that HTTP requests are asynchronous in nature, and making multiple requests simultaneously better utilizes available bandwidth, thus hides latency.

4. **Without Memcache, 1 or 4 thread(s)**

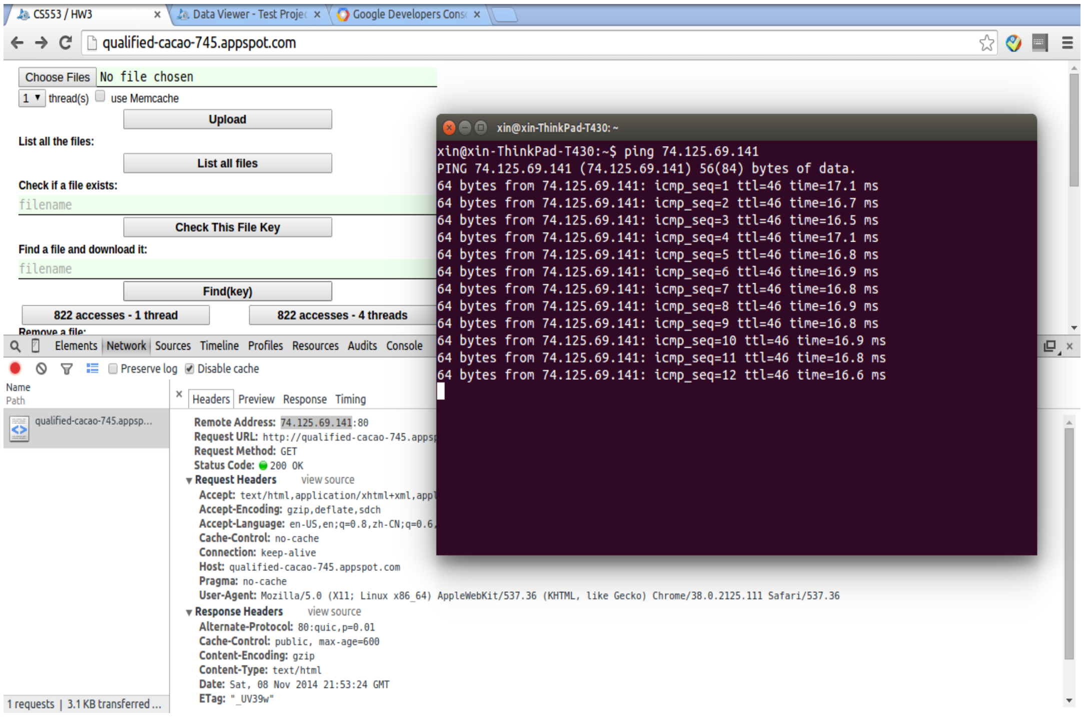


	Insert	Find	Delete
🟦 1 thread	0.679	0.339	0.092
🟧 4 threads	0.248	0.22	0.049

In this graph showing 1 thread v.s. 4 thread, we see the significant improvement as in with Memcache experiments. It again tells us that the client side, if running in 1 thread, is very inefficient, in terms of bandwidth utilization. Nonetheless, by not using Memcache, this experiment gives lower performance than the one with help from Memcache.

5. **Understand the minimum latency**

   We measure the minimum latency from our local machine(which did the experiment) to the remote server of our program using `ping` utility. `ping` the server's remote address



Choose Files No file chosen
1 ▾ thread(s) □ use Memcache

List all the files:

Check if a file exists:
filename

Find a file and download it:
filename

Remove a file:

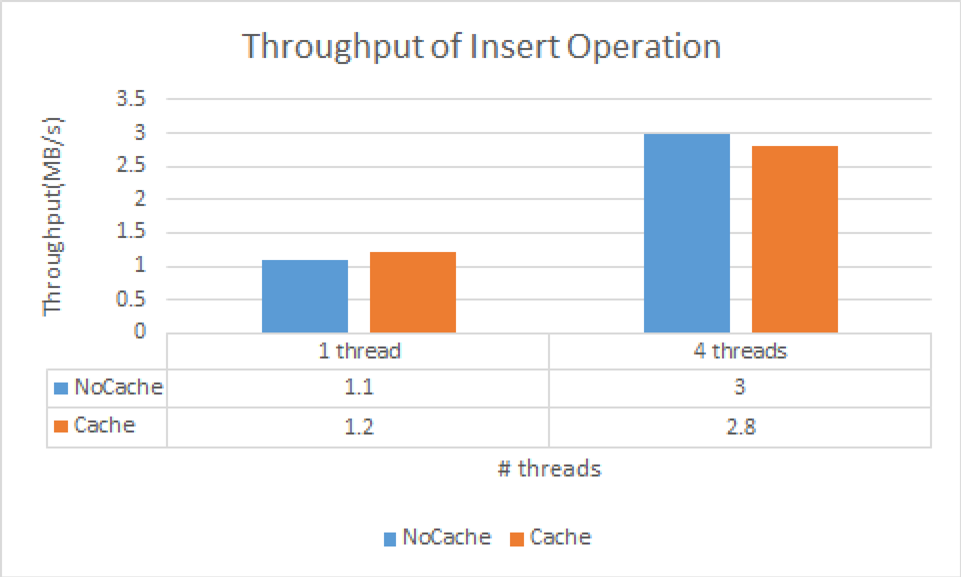
□ Preserve log ☑ Disable cache

```
Remote Address: 74.125.69.141:80
Request URL: http://qualified-cacao-745.appsp
Request Method: GET
Status Code: ● 200 OK
▾ Request Headers
   Accept: text/html,application/xhtml+xml,appl
   Accept-Encoding: gzip,deflate,sdch
   Accept-Language: en-US,en;q=0.8,zh-CN;q=0.6,
   Cache-Control: no-cache
   Connection: keep-alive
   Host: qualified-cacao-745.appspot.com
   Pragma: no-cache
   User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.111 Safari/537.36
▾ Response Headers
   Alternate-Protocol: 80:quic,p=0.01
   Cache-Control: public, max-age=600
   Content-Encoding: gzip
   Content-Type: text/html
   Date: Sat, 08 Nov 2014 21:53:24 GMT
   ETag: "_UV39w"
```

```
xin@xin-ThinkPad-T430:~$ ping 74.125.69.141
PING 74.125.69.141 (74.125.69.141) 56(84) bytes of data.
64 bytes from 74.125.69.141: icmp_seq=1 ttl=46 time=17.1 ms
64 bytes from 74.125.69.141: icmp_seq=2 ttl=46 time=16.7 ms
64 bytes from 74.125.69.141: icmp_seq=3 ttl=46 time=16.5 ms
64 bytes from 74.125.69.141: icmp_seq=4 ttl=46 time=17.1 ms
64 bytes from 74.125.69.141: icmp_seq=5 ttl=46 time=16.8 ms
64 bytes from 74.125.69.141: icmp_seq=6 ttl=46 time=16.9 ms
64 bytes from 74.125.69.141: icmp_seq=7 ttl=46 time=16.8 ms
64 bytes from 74.125.69.141: icmp_seq=8 ttl=46 time=16.9 ms
64 bytes from 74.125.69.141: icmp_seq=9 ttl=46 time=16.8 ms
64 bytes from 74.125.69.141: icmp_seq=10 ttl=46 time=16.9 ms
64 bytes from 74.125.69.141: icmp_seq=11 ttl=46 time=16.8 ms
64 bytes from 74.125.69.141: icmp_seq=12 ttl=46 time=16.6 ms
```

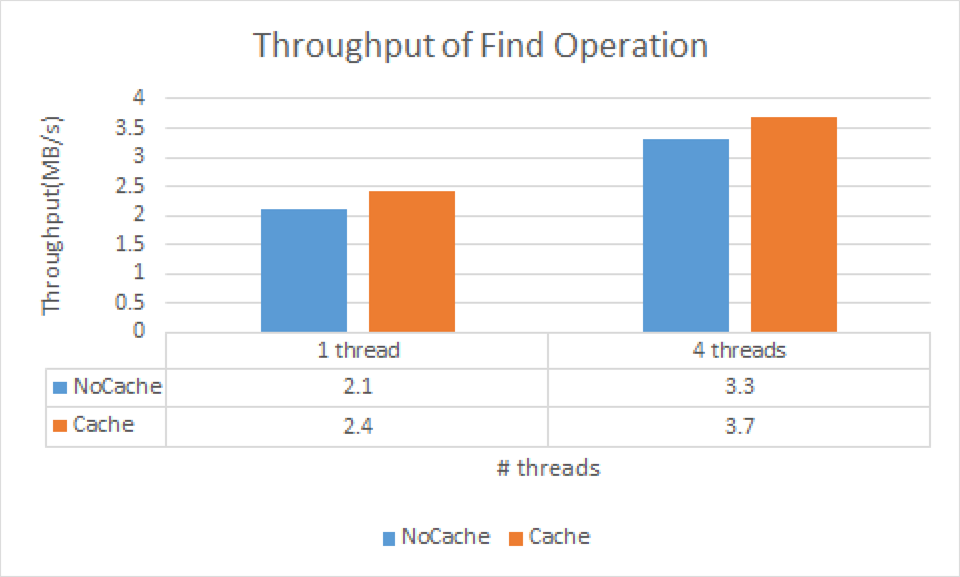
   From the graph above, the minimum latency is around **16.8 ms**.

Throughput results

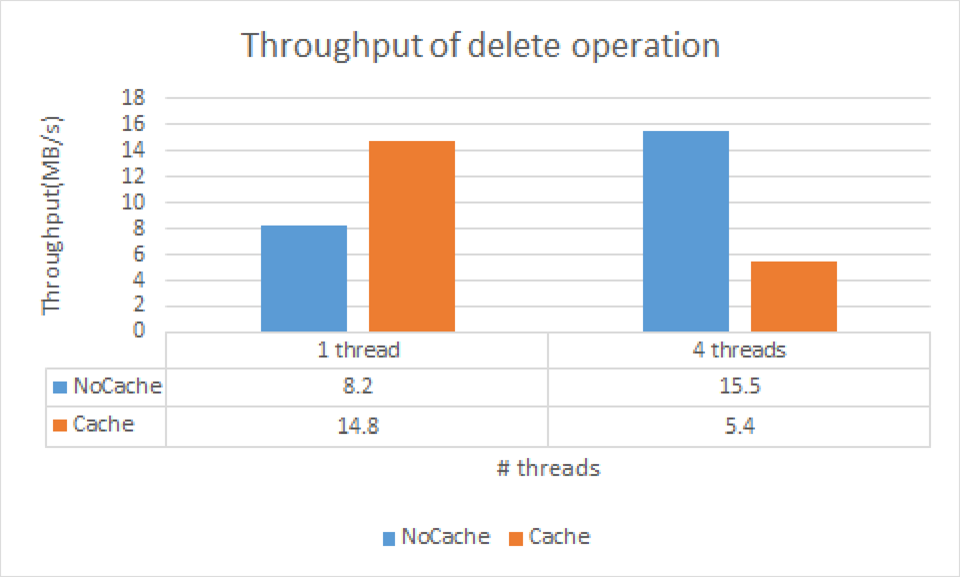


	1 thread	4 threads
NoCache	1.1	3
Cache	1.2	2.8

As is shown in previous graphs, improvements brought by multithreading is obvious. On the other hand, in 4-thread case, throughput of insertion is lower when using Memcache, which is reasonable, since the insertion with Memcache enabled requires more work.



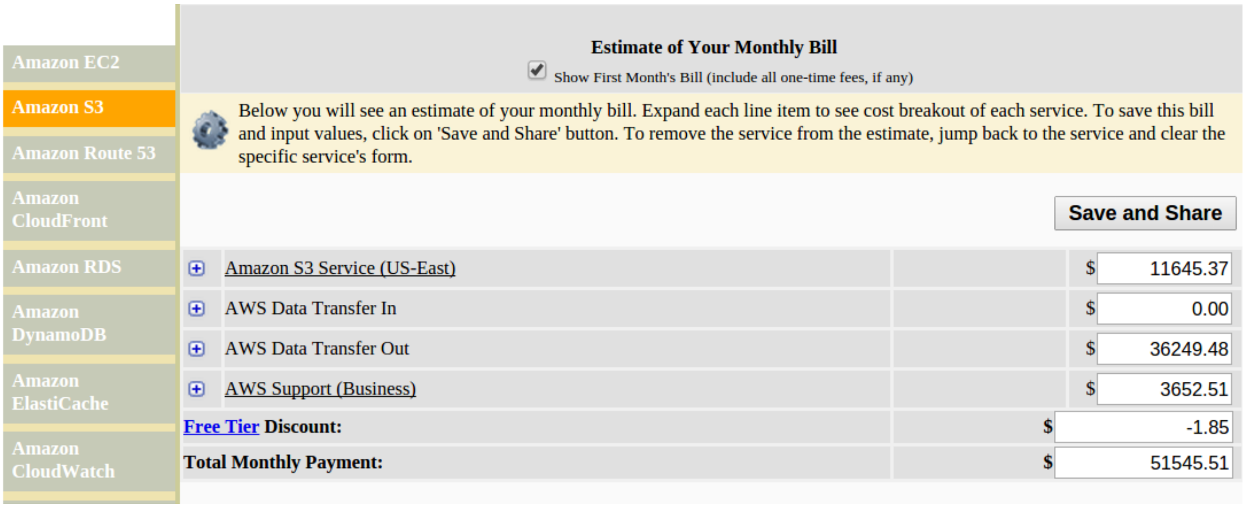
Here we see consistent improvement brought by both multithreading and the use of Memcache service. The result concerning the use of Memcache is not surprising since for every find access, our application checks the target's availability in Memcache first, and if it is in Memcache, the target is served to user directly.



As we illustrated before, deletion performance is not consistent on GCS. But in the 1 thread case, the result is reasonable, since deletion with Memcache requires more work.

Comparison to Amazon S3

The Object is to compare the cost efficiency between Amazon S3 and Google Cloud Storage. Using AWS Simple Monthly Calculator for S3, the result shows below:



Estimate of Your Monthly Bill

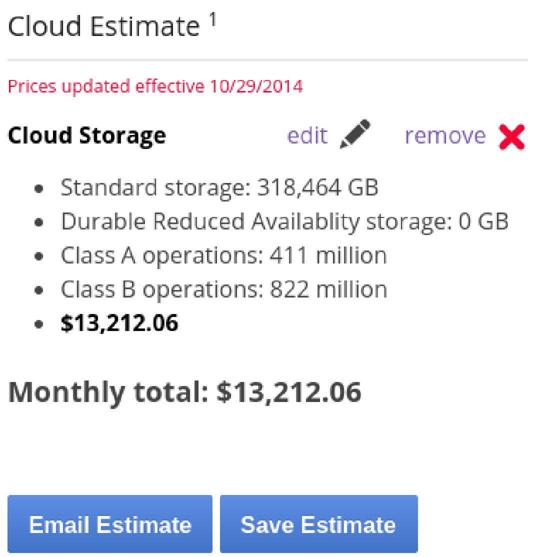
☑ Show First Month's Bill (include all one-time fees, if any)

Below you will see an estimate of your monthly bill. Expand each line item to see cost breakout of each service. To save this bill and input values, click on 'Save and Share' button. To remove the service from the estimate, jump back to the service and clear the specific service's form.

⊞ Amazon S3 Service (US-East)	$	11645.37
⊞ AWS Data Transfer In	$	0.00
⊞ AWS Data Transfer Out	$	36249.48
⊞ AWS Support (Business)	$	3652.51
Free Tier Discount:	**$**	-1.85
Total Monthly Payment:	**$**	51545.51

It show the total cost of Amazon S3 for one month is 51545.51 dollars.

Using Google Cloud Platform Pricing Calculator, the result shows below:



Cloud Estimate [1]

Prices updated effective 10/29/2014

Cloud Storage

- Standard storage: 318,464 GB
- Durable Reduced Availablity storage: 0 GB
- Class A operations: 411 million
- Class B operations: 822 million
- **$13,212.06**

Monthly total: $13,212.06

It shows the total cost of GAE storage for one month is 13212.06 dollars

To conclude, for the given assumption, the better choice is GAE storage.