Peter Meleney
DATA 516
December 10, 2019

Performance of Snowflake for LSH Search

Abstract

Fractal is a company dedicated to protecting photographers' intellectual property online. To that end Fractal requires that their customers only upload photos that are their own original work. Fractal has developed several technical solutions to ensure that photos that are uploaded are unique, and not derivatives or copies of work already on the site, one of these features involves using Locality-Sensitive Hashing (LSH) to identify photos that have a high likelihood of being copies of work already on the site. Fractal maintains a database of 1000 hashes of every photo on the site, this database must be searched every time a new photo is uploaded. The speed of the search is paramount because this feature will determine the length of time a customer is waiting before an uploaded image is accepted onto the platform. This paper investigates the speed and scalability of Snowflake, a leading cloud-based Relational Database Managements Systems (RDBMS) in searching for duplicate LSH hashes in databases of variable size.

1 Introduction

Fractal is a company dedicated to protecting photographers' intellectual property online. Fractal runs a website which guarantees that once an image is uploaded, no other copy or derivative of that image may be uploaded for as long as the image is maintained. Fractal achieves this in two ways: (1) images are digitally watermarked and if the watermark, or any part of it, is detected in an uploaded image, that image is rejected by the website's back-end, and (2) Fractal stores 1000 LSH hashes of each uploaded image, if an image with an identical LSH is uploaded to our servers, we subject that image to further screening to determine if it is copy or a derivative of a previously uploaded image.

LSH is used for similarity searching at scale by large technology companies like Uber and Pinterest[3]. LSH is particularly useful because by tuning the parameters b and r, where b is the number of bands, and r is the number of rows per band, we can finely control the proportion of false positives. These

parameters are important to data curation because they control the number of hashed values that must be stored in the database.

Snowflake is a data warehouse built for the cloud. In this paper we assess the speed with which an Compute_WH XL node handles LSH comparisons on databases of up to 1TB of data. This is equivalent to a database of approximately 80M photos.

2 Data

For our experiment we used photos from the 2017 Common Objects in Context (COCO) image data set. Created by Microsoft, this repository is free to access online at [2]. Since we are not creating a computer vision model in this application, I combined the Train, Test, Val, and Unlabeled repositories, for a total of 287,360 unique images of common objects. The images are in color, are of variable size, and together are over 46.8GB on disk.

3 Methods

To perform LSH I downloaded a package from github called LSHash version 0.0.4dev,[1] which needed updating to run in python 3.7. I chose to run LSH with 1000 unique hashes for each image of 32 bits each. The package uses the random projection method to generate the binary hashes. The images were first run through ResNet50 with 1k ImageNet optimal weights through to the final average-pool layer from which I extracted embeddings which were vectors of shape 1x1000. I performed the hashing on my local machine, and hashing 100 images took approximately one hour of machine time. Because of time constraints I was not able to hash all 287k images in the data set. Instead I hashed the first 1220 images and replicated the data on the snowflake server before running each successive search to determine how the database responded to scale-up in this way.

3.1 SQL Query

To maintain consistency between runs the exact same query was run every time. In this query I took the hashes from the first picture in the data set (see figure 1) and searched for that hash exactly.

```
SELECT filename FROM table1 WHERE
(HASH_0 = '00100001101011111111101111101011'
AND HASH_1 = '11111011100110101011100001111001') OR
(HASH_2 = '00110110110100000001111011000010'
AND HASH_3 = '01111100010011000111100101110001') OR
(HASH_4 = '00100101000100010101011001011111'
AND HASH_5 = '10001111111110111111100011001101') OR ...
```

Figure 1: The first image in the COCO Dataset. This is the image searched for in the database repeatedly.

4 Results

I found that the cold-cache run times remained fairly constant at approximately 20 seconds until the database reached 31.7GB, or approximately 2.5M images when run times began to deteriorate in an exponential fashion (see figure 2). This indicates that until the database reaches 32GB the cold-cache run time is dominated by overhead and not the query itself.

There were two outliers that occurred when running the query against the 1TB data set with a warm-cache (see Appendix 3). These were 84 and 94 seconds, whereas the average time was closer to 10 seconds excluding these two outliers (see figures 3, 4, and 5).

Furthermore I found that the average warm-cache run time remained fairly constant at approximately 2 seconds below 63.4GB, or approximately 5M images (see figure 3. However the run times of queries on a database below 64GB had a bimodal distribution (see figure 5.

5 Discussion

For our applications at Fractal we are most interested in the cold-cache run time because we do not expect the same query to be run repeatedly. Therefore in all cases 20 seconds is too long for our customers to be waiting to determine whether a single image is accepted or rejected by the system. We will need to either run these queries in the background without forcing the user to wait or find a faster method of querying a database that snowflake, with less overhead.

6 Appendix 1: Figures

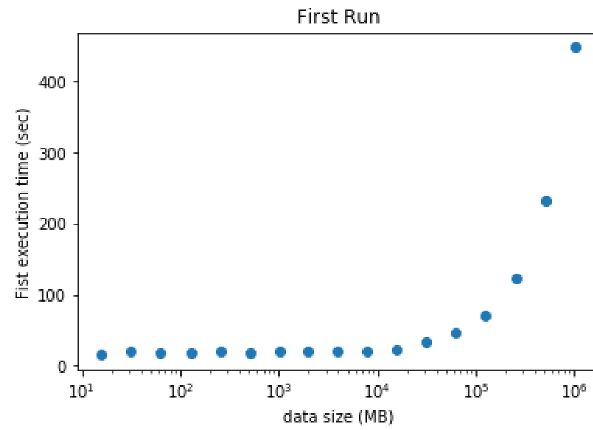

Figure 2: Cold-cache run times, note relative stability below 32GB.

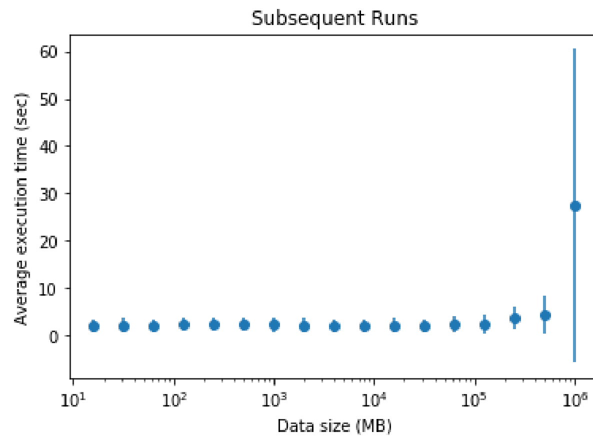

Figure 3: Warm-cache runtimes with outliers included. Points are average time and bars represent 1sd.

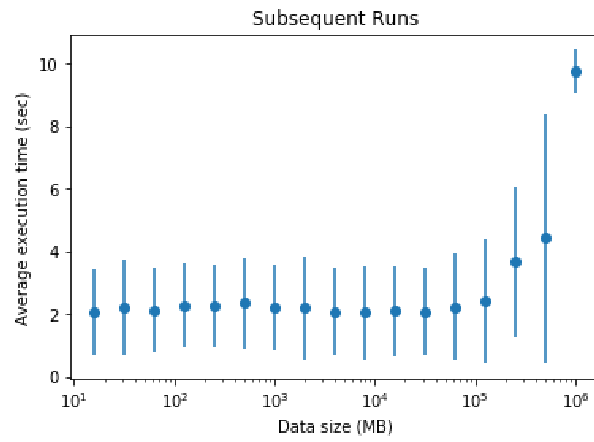

Figure 4: Warm-cache runtimes with outliers excluded. Points are average time and bars represent 1sd.

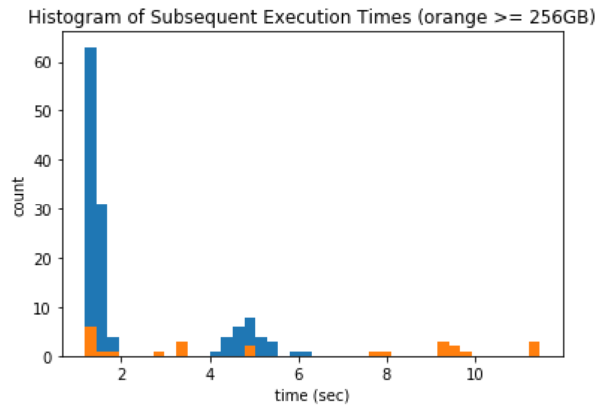

Figure 5: Warm-cache runtimes with outliers excluded. Note that runs below 256GB have a bimodal distribution with a peak near 1.5 seconds and another near 5 seconds.

Appendix 2: Cold-Cache Run Times

Run #	# Pics	Database (MB)	cold-cache execution time (s)
1	1220	15.8	15.55
2	2440	31.7	20.49
3	4880	63.4	18.32
4	9760	126.8	17.85
5	19520	253.6	20.88
6	39040	507.2	18.83
7	78080	1014.4	19.2
8	156168	2000	19.67
9	312320	4000	19.75
10	624640	7900	19.89
11	1249280	15900	23.29
12	2498560	31700	33.61
13	4997120	63400	45.76
14	9994240	126800	71
15	19988480	253600	123
16	39976960	507200	233
17	79953920	1014400	448

Table 1: Caption

Appendix 3: Cold-Cache Run Times

Run #	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9
1	4.21	1.24	4.96	1.24	1.21	1.61	1.52	1.23	1.24
2	4.44	1.47	5.52	1.23	1.28	1.47	1.34	1.56	1.40
3	1.37	1.37	4.75	1.39	4.54	1.31	1.54	1.46	1.36
4	4.69	1.48	4.9	1.63	1.65	1.44	1.39	1.50	1.78
5	1.76	1.42	1.45	1.51	4.54	1.85	1.53	4.89	1.52
6	1.63	4.80	5.20	1.43	1.53	1.51	1.80	1.59	1.53
7	4.92	1.40	1.47	4.65	1.49	1.52	1.53	1.39	1.53
8	5.34	5.22	1.26	1.43	1.22	1.26	1.24	1.35	1.39
9	4.96	4.29	1.20	1.53	1.22	1.20	1.31	1.48	1.46
10	5.08	4.51	1.23	1.2	1.22	1.22	1.40	1.25	1.21
11	4.63	5.00	1.34	1.31	1.18	1.18	1.34	1.51	1.25
12	4.52	1.23	1.23	1.24	4.85	1.35	1.42	1.44	1.39
13	5.26	1.38	1.23	1.42	5.50	1.34	1.18	1.37	1.33
14	1.38	5.93	1.43	1.35	6.27	1.24	1.42	1.29	1.33
15	7.84	3.36	1.40	3.42	1.37	7.93	3.37	2.93	1.38
16	11.46	11.38	1.31	4.78	1.29	1.86	1.51	1.42	4.78
17	9.27	84	94	9.48	9.36	9.39	9.69	11.44	9.53

Table 2: Warm-cache run times, all times in seconds.

References

[1] Lshash, 2013.

[2] Coco: Common objects in context, 2019.

[3] Andrey Gusev. Image similarity detection using lsh and tensorflow, 2018.