Distributed Hashing Strategies for Tweets Lookup

290S Winter 2016 - UCSC

Sanjana Maiya, Greeshma Swaminathan

***Abstract*—The aim of the project is to build a scalable distributed Tweets lookup system and explore various hashing strategies to distribute Tweets across machines. We explore static hashing, consistent hashing and one hop distributed hash table (DHT). This report presents the different strategies that we tried and their results.**

I. Introduction

The aim of the project is to build a scalable distributed Tweet lookup system. Our lookup system provides search on Tweets for a user. To explore the best possible design of the system, we study mainly two different hashing options, static hashing and consistent hashing. Static hashing helped us distribute and lookup the data in a deterministic manner, but scalability proved to be a problem. Consistent hashing is known to solve the scalability issue by having to reconfigure only a subset of the Tweets while adding more servers. However, when exploring this hashing strategy, we encountered 'hot spot' issues in which popular user searches were overloading some machines while the others remained underutilized. To tackle this, we needed the ability to move 'hot' Tweets to the lightly loaded systems. Our solution was to use a one hop distributed hash table which uses consistent hashing and static hashing. This report presents the results of our experiments.
The report is organized in sections. The first section explains briefly about the different hashing strategies, the second section outlines the design of our system, the third section presents our experiments, followed by related work, future work, conclusion, acknowledgments and references.

II. Distributed hashing strategies

Hashing is a useful tool to distribute data in a deterministic way across multiple machines so that the same function can be used for distribution and lookup. We explored the following hashing techniques for distributing data:

A. Static Hashing

Static hashing does a good job of distributing data uniformly across a fixed number of machines. A good static hashing strategy is to use a good hash function to hash the lookup key, modulo with the number of machines and map it on to the resulting machine numbers. This strategy works well when the number of machines are fixed, but as machines get added or removed, a complete redistribution of the entire data will be needed to get the system working again.

B. Consistent Hashing

Consistent hashing is a technique introduced by Karger et al.[1]. Consistent hashing assigns keys to buckets such that each bin roughly receives the same number of items. Addition or removal of bins need only a remap of k/n items where k is the number of keys and n is the number of bins. While consistent hashing overcomes the limitation of static hashing when it comes to redistribution of keys, the system can still have issues caused by hot spots. In our case, when one server is heavily loaded with search requests for Tweets, consistent hashing does not provide a strategy to redistribute this load.

C. One hop distributed hash table with consistent hashing

In order to solve the hot spot problem seen with consistent hashing, we used an approach described in Orleans[3]. A One hop mechanism is used - the first step involves look up of the location of the data using a distributed hash table. This hash table uses consistent hashing at its core, and contains a mapping of keys to locations of the data (in our case, Tweets). The second step involves the actual retrieval of data using this location. The data itself is distributed , and this could have been done in several ways. But the simplest method is to use the static strategy itself. Therefore, we use a combination of the previous strategies and an additional hop in order to overcome the shortcomings of redistribution and hot spots.

III. System setup and design

A. Data

We harvested Tweets for our data using Twitter4J[5]. The fields of interest in each tweet are TwitterUserId and UserId. The TwitterUserId is used as an identifier for Tweets and UserId uniquely identifies a Twitter user. UserId is used in our hashing function for each of the hashing strategies, since a lookup for a user's tweets is done through the UserId. There are upto 3200 Tweets per user.

B. System

Amazon Web Services[4] Relational Database Servers (RDS) instances serve as the repositories for data in all the hashing strategy experiments. There are a total of 10 RDS servers that we are using for our experimental setup.

Figure 1 shows the high level design of our system. The tweets are stored in MySQL database instances running on Relational Database Services on AWS. Data replication is

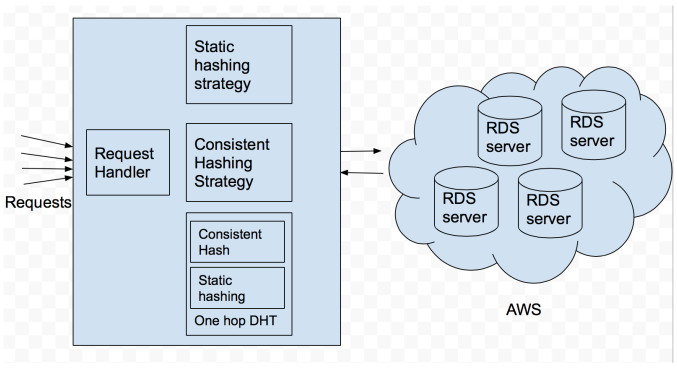


Fig. 1: Tweet Lookup System Setup

not handled in the current scope of the project. We maintain one RDS instance as central repository which contains all the Tweets. To set up our system, we have a module which splits Tweets across multiple instances of servers based on the Hashing Strategy specified.A client interface is provided through which we can lookup the user by supplying the UserId and the hashing strategy.

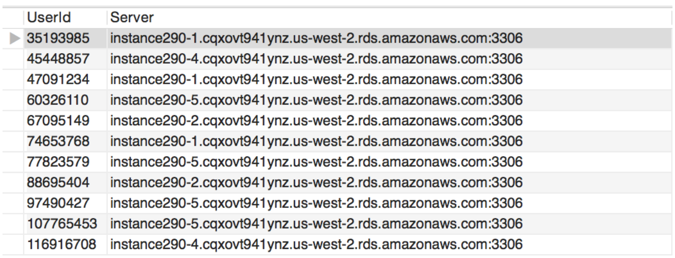
1) Static hashing:
   In static hashing, an MD5 Hash function is used to hash the UserId of each Tweet. We then calculate the hash modulo number of servers to find the server where the data should be located. To perform lookup, the same operations are used which provides the server to read the data from. MD5 distributes users evenly across servers and this works well when the number of servers do not change.
   We started with 5 servers and tested adding a server and then removing it to simulate scaling and crash recovery. In this scenario Static hashing requires all the data to be rehashed again. When the system detects an addition or removal of server, it blocks all incoming requests until the system completes the reconfiguration. This causes a spike in latency for requests.

2) Consistent Hashing:
   For Consistent hashing, we use an implementation of Consistent hashing provided by Cloudera, Inc[6] with small modifications. This implementation also uses MD5 as its hashing function and was configured with 5 virtual nodes per server. Unlike Static hashing, Consistent hashing needs to maintain state which stores the number of servers and maps them onto a circle. For each key, the nearest server mapped in the circle ahead of the key's hash is returned.
   Initially, the data is split across 5 servers and Consistent hashing performs well when the servers do not change. The next stage added and removed a server. The advantage of Consistent hashing is that it needs to redistribute only a part of the data (k/n) when nodes are added or removed. When the system detects addition of a server, it blocks all incoming requests and does the following:
   - The new server and its virtual nodes are mapped on to the circle.
   - The existing virtual nodes from where the data has to be stolen is retrieved. The UserIds of Tweets in each of these nodes are rehashed and checked if they now map to the new virtual nodes. If so, the Tweet is removed from the old node and added to the new node.

   When the system detects removal of a server it does the following.
   - The server and its virtual nodes are removed from the circle.
   - The UserIds of Tweets which were present in the removed nodes are rehashed to the remaining servers.

   For both the cases of adding and removing servers, the latency spike is expected to be much lower than static hashing. However, both consistent hashing and static hashing have the drawback that the hashing of a UserId to a server is deterministic. When a server becomes a hot spot some data has to be moved to relieve the server's load, especially when there are other underprovisioned servers. The next hashing strategy explores the solution to this problem.

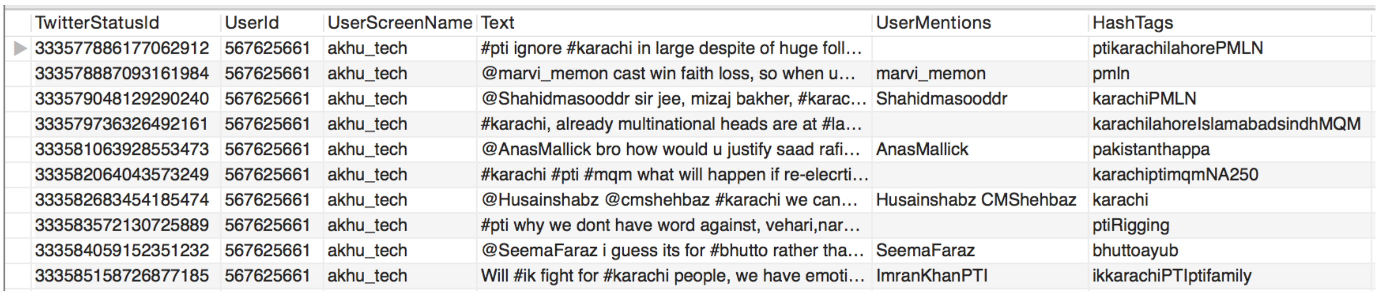
3) One hop Distributed Hash Table:
   The Distributed Hash Table has a mapping of UserId to the server where the Tweet is located. The first lookup locates the server on which this user's Tweets are stored. The second lookup actually retrieves the Tweets. This gives us the flexibility to change the location of Tweets, which was not possible with the earlier strategies. So in case a hot spot is detected on a server, some of the users having popular Tweets are moved out to a less loaded server.
   To create the one hop Distributed Hash Table, consistent hashing is used to split the UserId-Server mapping. In our setup, we use 3 RDS servers to maintain this table (Refer Figure 2). For the actual Tweets to be split, any strategy may be used. We use static hashing to split the Tweets across 5 servers (Refer Figure 3). The entries in the UserId-Server table will point to one of these 5 servers . When the system detects a hot spot the following steps are taken:



UserId	Server
35193985	instance290-1.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
45448857	instance290-4.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
47091234	instance290-1.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
60326110	instance290-5.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
67095149	instance290-2.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
74653768	instance290-1.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
77823579	instance290-5.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
88695404	instance290-2.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
97490427	instance290-5.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
107765453	instance290-5.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306
116916708	instance290-4.cqxovt941ynz.us-west-2.rds.amazonaws.com:3306

Fig. 2: Distributed Hash Table : Table for First Lookup

   - The set of UserIds causing the hot spot are identified. These Ids correspond to popular requests from the client.



TwitterStatusId	UserId	UserScreenName	Text	UserMentions	HashTags
333577886177062912	567625661	akhu_tech	#pti ignore #karachi in large despite of huge foll…		ptikarachilahorePMLN
333578887093161984	567625661	akhu_tech	@marvi_memon cast win faith loss, so when u…	marvi_memon	pmln
333579048129290240	567625661	akhu_tech	@Shahidmasooddr sir jee, mizaj bakher, #karac…	Shahidmasooddr	karachiPMLN
333579736326492161	567625661	akhu_tech	#karachi, already multinational heads are at #la…		karachilahoreIslamabadsindhMQM
333581063928553473	567625661	akhu_tech	@AnasMallick bro how would u justify saad rafi…	AnasMallick	pakistanthappa
333582064043573249	567625661	akhu_tech	#karachi #pti #mqm what will happen if re-elecrti…		karachiptimqmNA250
333582683454185474	567625661	akhu_tech	@Husainshabz @cmshehbaz #karachi we can…	Husainshabz CMShehbaz	karachi
333583572130725889	567625661	akhu_tech	#pti why we dont have word against, vehari,nar…		ptiRigging
333584059152351232	567625661	akhu_tech	@SeemaFaraz i guess its for #bhutto rather tha…	SeemaFaraz	bhuttoayub
333585158726877185	567625661	akhu_tech	Will #ik fight for #karachi people, we have emoti…	ImranKhanPTI	ikkarachiPTIptifamily

Fig. 3: Distributed Hash Table : Table for Second Lookup

- All the Tweets of some of these users are redistributed to randomly selected, less loaded servers. Further requests for these UserIds are blocked till redistribution is complete.
- The entries for the UserId-Server mappings are updated in the Distributed hash table.

This resolves the hot spot as new request will be diverted to multiple servers. Some of the advantages of this approach are

- Hot spot resolution
- Addition and removal of a server which serves as a DHT need only K/n UserId-Server mappings to be redistributed across the DHTs.
- Easy addition of servers which serve as repositories for Tweets. Initially, the new server added will not have any Tweets and will not receive any requests. As load increases on other servers, Tweets will get redistributed to this new server. This happens automatically as a part of hot spot resolution.
- Removal of a Tweet repository server involves only redistributing that server's Tweets to other servers and updating their entries in the DHT mapping table.

Therefore, this approach has all the advantages that consistent hashing has and additionally resolves the hot spot issue. However, since this approach requires an additional hop for each request, it may perform slightly worse than the previous approaches. We explore this in some detail in our experiments.

IV. Experiments

A. *Static Vs Consistent Hashing*

Our first experiment involved evaluating static versus consistent hashing strategies. We set up the system with static and consistent hashing and tested Tweet requests for users. We also added tests for addition and deletion of servers. Once setup, static and consistent hashing performed similarly when all the servers were up and running. When a server was added or deleted, the latency of Tweet requests for static hashing was much higher than for consistent hashing. Figure 4 shows the difference in request latency between the two. The spikes in latency (of the order of several minutes) happens when a server is being added or removed. Once the reconfiguration is completed, latencies drop to the order hundreds of milli seconds which appears close to zero in the graph. As is clear from the graph, consistent hashing causes comparatively smaller spikes and affects the latencies for a much shorter time. Latency was computed as a rolling average over a period of one hour.

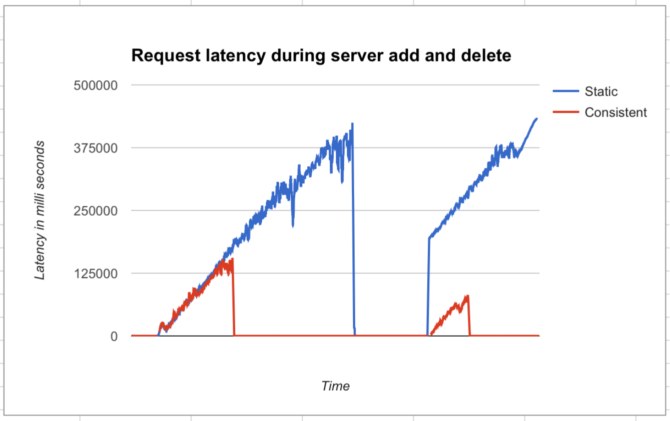


Fig. 4: Request latency spikes during adding and removing a server

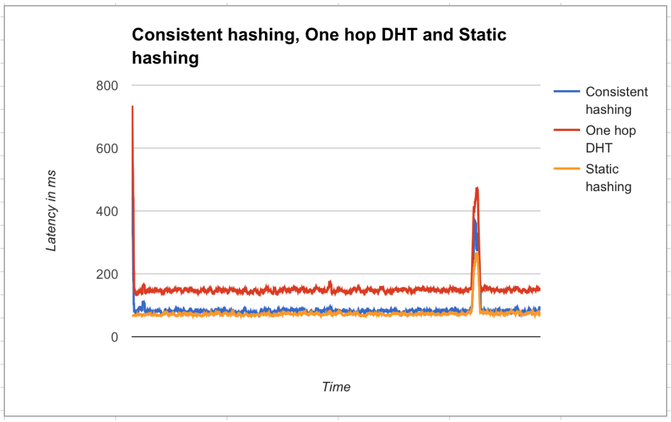


Fig. 5: Latency comparison for the three strategies

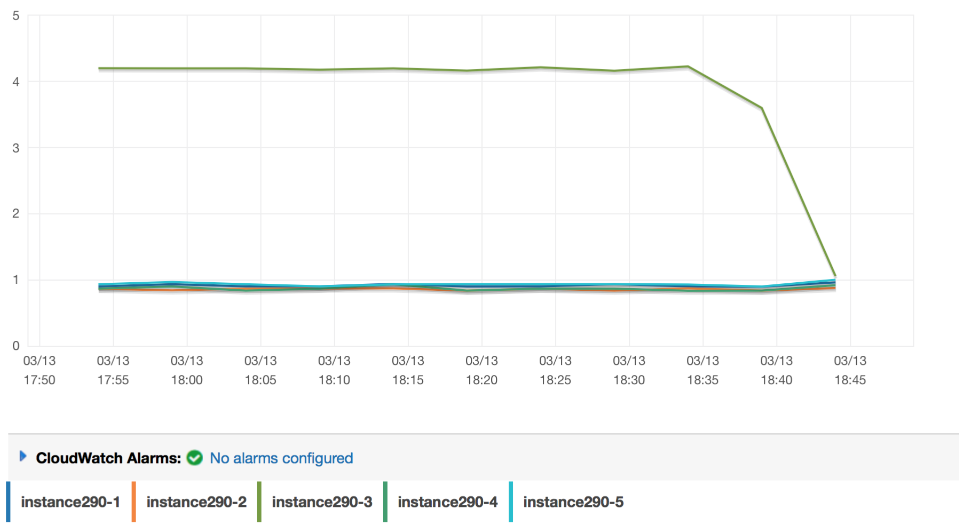


Fig. 6: Amazon AWS CPU usage graph depicting hot spot issue with consistent hashing

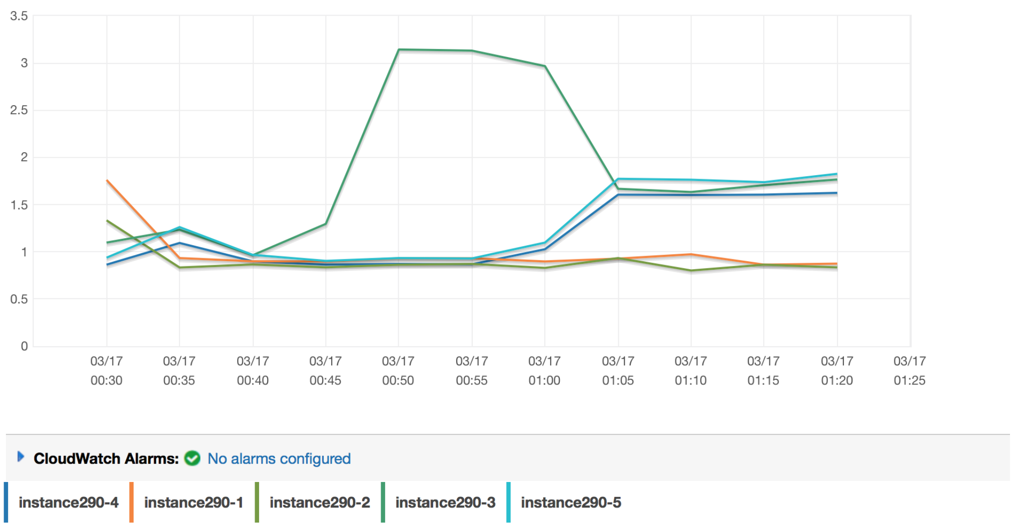


Fig. 7: Amazon AWS CPU usage graph depicting hot spot resolution using One hop DHT

B. *Hot spots with Consistent Hashing*

The hot spot problem can be readily replicated in our system. Since consistent hashing is based on UserIds of tweets, a particular user always gets mapped to the same server bin. Therefore simultaneous multiple requests for users' tweets from a particular server is all is needed in order to observe the hot spot. The consistent hashing strategy cannot overcome this hot spot since it cannot remap the UserIds to other servers which may have lesser load. Figure 6 shows the Amazon Web Services CPU usage graph for our system. The graph plots CPU utilization of each server instance over time. Tweets of server instance290-3's users are being requested heavily, while the other servers are being hit with much fewer requests. Of the five server instances running, instance290-3's CPU utilization remains high over a prolonged period of time compared to all other servers.

C. *Hot spot resolution with one hop DHT*

In order to evaluate the one hop DHT stategy, we first simulated a hot spot, similar to the previous experiment. Multiple simultaneous requests were sent to a particular server for users' tweets. The initial upward trend in the CPU utilization can be seen in Figure 7. Our claim was that the one hop strategy would rectify the hot spot by redistributing the tweets from a heavily loaded server to a server which was relatively ”free”, and thus even out the load across the servers. This can be seen in the Figure 7, where once the threshold for the CPU utilization for instance290-3 was reached, its load dropped, and the utilization of instance290-4 and instance290-5 picked up. The change in the utilization is seen once Tweets from instance290-3 are redistributed to servers instance290-4 and instance290-5 and the DHT was remapped to point requests for these tweets to the new servers.

We next wanted to evaluate how One hop DHT would perform compared to the other two hashing strategies. There was a 60% increase in latency compared to the other two strategies when serving requests. However we have not performed any optimization to reduce this - maintaining a cache for the mappings on the local device should solve this problem in normal scenarios. The result of this experiment is shown in the Figure 5

V. Related work

Orleans [3] has a one hop distributed hash table from which we took inspiration for our implementation. Orleans uses distributed virtual actors for programming and scalability. The DHT in Orleans holds actor placement mappings.

The followup paper on consistent hashing ”Web caching with consistent hashing” [2] discusses a hot spot issue for the web. They use the approach of replicating hot pages in multiple servers and use the DNS mappings to point to a list of IP addresses and randomly picking one from it. This differs from our approach since we do not consider using replication in order to resolve hot spots.

VI. Future work

Our experiments with One hop Distributed Hash Table included resolution of hot spots. However, due to time constraints, we were unable to put to test features such as adding and removing of servers for One hop DHT. It will be interesting to observe the effect of adding a Tweets server in this implementation, since doing so should automatically redistribute the load among all servers. We also plan to cache the mapping present in the DHT on the local device and check if the latency reduces for serving requests.

Another area of future work is exploring other hashing strategies, especially schemes which can overcome the hot spot issue seen in consistent hashing.

VII. Conclusion

Our goal for this project was to understand how different distributed hashing methods could be used for Tweets lookup. Static and consistent strategies performed well in the ideal case, but each had its own disadvantages. Consistent hashing suffered from hot spots, and static hashing additionally could not scale easily without rehashing all the Tweets. We attempted to solve these problems with a One hop strategy which uses consistent hashing for the first lookup, and static hashing for the second lookup. While the additional hop introduces a small latency increase, it resolves the problems seen in the other strategies. Our work, including the code, and experiments can be found here: https://github.com/greeshmaswaminathan/290S

VIII. Acknowledgments

This project has been a great learning experience and we express our sincere thanks to Professor Peter Alvaro for supporting and guiding us. We also thank Caitie McCaffrey whose talk on Distributed Hashing Strategies inspired us to implement a One hop system to solve the hot spot problem.

References

[1] Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D. (1997). Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. . *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*,ACM Press New York, NY, USA. pp. 654663.

[2] Karger, David, et al. Web caching with consistent hashing. *Computer Networks 31.11 (1999): 1203-1213.*

[3] Bernstein P, Bykov S, Geller A, Kliot G, Thelin J. Orleans: Distributed virtual actors for programmability and scalability. MSR Technical Report (MSR-TR-2014-41, 24). http://aka. ms/Ykyqft; 2014 Mar.

[4] Amazon Web Services : aws.amazon.com

[5] http://twitter4j.org/en/index.html

[6] Cloudera Consistent Hash Implementation under Apache 2.0 license: https://github.com/cloudera/flume/tree/1d7535638556998e895d55599a2f4a024390edd core/src/main/java/com/cloudera/util/consistenthash