Benchmarking Memcached Neo4j

1 Participants

- Nidhi Praveen Jain | Student Id – 7249597229 | Email Id – nidhiprj@usc.edu
- Sathish Srinivasan | Student Id - 2777060734 | Email Id – sathishs@usc.edu

2 Objective

- Benchmark REST version of Neo4J and observe its performance for different number of users and threads.
- Implement memcache on the REST version of the Neo4J client and benchmark again to observe the performance improvement.

3 Introduction

Neo4j v2.0.2

Neo4j is a NoSQL graph database implemented in java. It is schema less, supports transactions and is fully ACID. It uses graph data model representing data as nodes and relationships. Neo4j database can be accessed in two different versions

1. Embedded Java
2. REST API

Embedded Java is designed for a single user and is not suitable when many users want to access the database. Here the database is local to the user. Querying and writing to the database happens as a normal read and write to the hard disk.

REST API allows multiple users to access the database simultaneously. Neo4j server is started as a service and different users connect to the server. They send query and get reply as HTTP request and response.

Memcached v1.4.2

Memcached is a general purpose distributed in-memory caching system. Memcached API provides a giant hash table distributed across multiple machines. When trying to fetch data, we will first look into the memcached if the result is present, if yes it will be returned from the cache, if not the database will be queried for the same data and the result will also be stored in the cache. In case of an update, the database will be first updated but then cache entry should also be updated and if not it may serve stale data. There are three different ways to update the cache entry which includes

1. Invalidation
2. Refresh
3. Incremental update

With invalidation technique the cache entry is deleted when the corresponding data is updated in the database. Refresh technique involves a read modify write where the data is first checked with the previously cached value and then updated in the cache to avoid inconsistency. Incremental updates involve making changes to the cache entry with database triggers. Each time the database is updated it generates a trigger to make changes in the cache.

4 DESCRIPTION AND IMPLEMENTATION

- In this project we created a REST client that interacts with Neo4j server. Reading and writing to database happens through CYPHER queries using the REST API. The experiments were all done on localhost instead of real network.
- Once the Neo4j REST client is created the query results are memcached and the performance improvement is compared to that of original client. There are different versions of the client that can be used to communicate with memcached server and we made use of spymemcached client.
- With memcached version when the data is to be retrieved, we first check if the data is available in the cache, if it is then the data is returned form the cache directly. If not, we get the data from the datastore and then it is added to the cache.
- When data is to be written it is first updated in the database. Then we try to find a corresponding affected entry in the cache and if it is available we invalidate it by deleting the cache entry.
- Finally we compared the performance (throughput, latency and stale data) by benchmarking REST client with and without memcache using BG benchmark application.
- We benchmarked with various workloads that include View Profile action, List Friends action and Symmetric updates and compared the performance improvement. The maximum execution time for each of these experiments was 180 seconds. The results obtained and steps taken to improve the performance are listed below.

5 SUMMARY AND MAIN FINDINGS

5.1 LOADING NEO4J DATABASE

- Loaded the data into Neo4J using the REST client. The loading was done on Windows 8 operating system with SATA hard disk without batching and the following results were obtained.

Image Availability	**Image Size**	**Threadcount**	**Load Time (seconds)**
False	NA	1	32040
		10	14080
True	12 KB	1	33590
		10	15160
True	500 KB	1	38220
		10	16310

INFERENCES FROM LOADING DATABASES:

- The number of times queries sent over the network affects the load time to huge extent. So reduce the load time the properties of a node were added to a hash map and the map was passed along with the cypher query. This made the load time faster for the creation of nodes.
- However creating relationship between nodes was taking a long time. We successfully overcame this problem by creating the indexes at the beginning and each time a node is created the index gets updated. This made the loading time approximately three times faster.

5.2 COMPARING NEO4J AND MEMCACHED NEO4J RESULTS USING BG BENCHMARK

- After loading the data we benchmarked the Neo4j REST client by emulating various social actions like ViewProfile, ListFriends and Symmetric updates. The maximum execution time for each experiment is 180 seconds and the results are as follows:

1. **ViewProfile Action** – We used 10 threads to load the data store and varied the size of image and threadcount.

Threadcount	**Overall Throughput (Actions/s) – No Images**		**Overall Throughput (Actions/s) – 12KB Images**		**Overall Throughput (Actions/s) – 500 KB Images**	
	No memcache	**Memcached**	**No memcache**	**Memcached**	**No memcache**	**Memcached**
1	604.99	1639.57	170.41	243.79	4.95	5.49
10	2153.02	3255.03	1497.47	1986.47	3.13	4.76

INFERENCES FROM VIEWPROFILE ACTION:

- The number of queries issued to get the desired results has a huge impact on the throughput of the actions. For example in the view profile action to get the count of number of resources owned, number of confirmed friends and number of pending friends if they are issued separately the throughput is three times slower than querying all count in a single query.
- To improve the throughput further we decided to add the counts as a property in the node. This increased the throughput 1.5 times than querying individually.
- However this would make the load time slower. To accommodate this fact updating the counts were also made as part of the same query used to create nodes and relationships. For example when an invite friend action is done the original query

```
MATCH (u:USERS), (u1:USERS) WHERE u.userid=inviterID AND
u1.userid=inviteeID CREATE (u)-[r:INVITES]->(u1)
```

  was rewritten as below

```
MATCH (u:USERS), (u1:USERS) WHERE u.userid=inviterID AND
u1.userid=inviteeID CREATE (u)-[r:INVITES]->(u1)
SET u1.pendincount=u1.pendingcount+1;
```

- Similarly with accept friend, reject friend and thaw friendship and create resources the corresponding counts were updated. This way we were able to achieve better throughput both during loading and benchmarking phase.

2. **ListFriends Action** – We used 10 threads to load the data and varied the number of friends per user and the image sizes and observed the following results.
   i. With no images:

Threadcount	**Overall Throughput (Actions/s) – 10 friends/user**		**Overall Throughput (Actions/s) – 20friends/user**		**Overall Throughput (Actions/s) – 50 friends/user**	
	No memcache	**Memcached**	**No memcache**	**Memcached**	**No memcache**	**Memcached**
1	530.65	898.43	406.18	572.24	289.81	191.24
10	2117.57	3237.51	1610.57	2108.64	1138.66	791.84

ii. With 12KB images:

Threadcount	**Overall Throughput (Actions/s) – 10 friends/user**		**Overall Throughput (Actions/s) – 20friends/user**		**Overall Throughput (Actions/s) – 50 friends/user**	
	No memcache	**Memcached**	**No memcache**	**Memcached**	**No memcache**	**Memcached**
1	357.06	877.118	220.67	429.56	99.53	178.96
10	1275.31	2969.53	1677.40	2176.33	373.60	1256.28

INFERENCES FROM LISTFRIENDS ACTION:

- With increase in the image size the memcache results were lower than the normal Neo4j. This was because with increase in the image size the available memory of 64MB for memcache was completely occupied and resulted in poor throughput.
- Hence we increased the size of the memcache to 2GB and re-ran the experiment and obtained the above results. The throughput now increased to twice the amount of REST client without memcache.

3. **Symmetric Updates** – We used 10 threads to load the data and varied the percentage of updates using 3 different workloads. There experiments were done using the database which has no images.

i. Symmetric Very Low Update Action:

Threadcount	**Overall Throughput (Actions/s)**		**Overall Staleness (staleReads/totalReads)**	
	No memcache	**Memcache**	**No memcache**	**Memcache**
1	198.06	218.25	0	0.
10	1716.15	2373.38	0.08	0.1904

ii. Symmetric Low Update Action:

Threadcount	**Overall Throughput (Actions/s)**		**Overall Staleness (staleReads/totalReads)**	
	No memcache	**Memcache**	**No memcache**	**Memcache**
1	425.74	501.64	0	0
10	1647.98	1723.28	0.15	0.1667

iii. Symmetric High Update Action:

Threadcount	**Overall Throughput (Actions/s)**		**Overall Staleness (staleReads/totalReads)**	
	No memcache	**Memcache**	**No memcache**	**Memcache**
1	125.39	132.49	0	0
10	255.01	326.28	0	0

Inferences from symmetric updates action:

- From the experiment results it is clear that with the increase in updates the memcache performance decreases. This is because with more updates the cache entries gets invalidated and every time a read action is done it has to fallback on datastore to get the results.
- We verified this by comparing the load in Neo4j. In the case of memcache with read only actions the neo4j was taking only 10% of CPU while the java program execution and memcache took 90% of CPU approximately. But with the updates the Neo4j was taking 70% of CPU and java program execution remaining 30%.

6 CONCLUSION

By comparing the benchmarking results on raw Neo4j REST client and memcached Neo4j REST client it is evident that the memcache definitely improves the performance.

7 FUTURE WORK

1. Instead of performing the experiments on localhost would like to perform on real network with the server and clients running on separate machines.
2. With invalidation technique though the stale data being generated is very less it results in poor performance since the whole cache entry is deleted for every minor update. Incremental update will probably result is better usage of memcached and hence would like to implement incremental update of caching.
3. The space available for the memcache can be better utilized if the results are stored properly. For example in the list friends action the whole vector of hash maps are stored for each user and results in redundant storage of same information and available memory of memcache gets full. If the individual users properties are stored as hash maps then the redundancy can be avoided and would result in better memory usage.