**SHEEP Enhancements in Memcached:**

(*Note: SHEEP is a paper published by a research team from Yahoo! Labs, Barcelona*)

SHEEP is a distributed in-memory key-value store just like Memcached and Redis, optimized for multi-get queries, which are widely used in social networking systems. In multi-get queries, data associated with all the friends of a user is read and processed on the client. So a single “multi-get” query is issued to read data of all friends instead of sending a “get” query for each friend. This way if multiple friends data resides on a data storage server, then all this data (of multiple friends) can be retrieved in a single response. This saves sending multiple requests to a server and receiving/processing multiple responses from a server. If client is single threaded this also avoids RTT (round trip time) associated with multiple requests.

SHEEP improves performance of multi-get queries by employing two methods:

1. Doing “data aggregation on server”. Normally in multi-get queries, huge amount of data is carried over from storage server to client and then this data is processed on client. Often a very small subset of the total-read data is returned as output (filtered data) to the user. In SHEEP, in addition to “data filtering on client” we also perform “data filtering on server”. Due to this very small amount of filtered data (only) is carried over from storage server to client. This drastically reduces network load.
2. Data colocation. Sheep tries to store data associated with a user and its friends on the same storage server, so that only one server (or very few storage servers) is contacted for serving the request. In memcached/redis data is distributed randomly across multiple storage servers and in the worst case all storage servers are contacted for serving the request. Data colocation reduces number of network packets sent out and received. It also complements “Data aggregation on server” as lot of data (almost all) get filtered on server and very few data flows over network.

SHEEP advantages:

1. Client consolidation/Cost Reduction (with same amount of hardware we can get higher performance or with less amount of hardware we can get same performance).
2. Sustained high performance at peak loads. SHEEP is virtually unaffected by peak loads as it ensures that very less data flow over network from storage server to client thereby avoiding network saturation.

We have made enhancements proposed in SHEEP in memcached. This is done because of following reason:

* Memcached is stable, popular, tried & tested and used in many mission critical applications. It supports almost all required features like persistence, replication, data re-placement policy, concurrency control (CAS), efficient use of memory etc.
* Putting all these features of memcached in SHEEP would take lot of effort and time. So it was decided instead to put SHEEP enhancements in memcached.

How is this different or better than Redis + Lua:

“Data aggregation on server” that we have implemented in memcached is similar to executing Lua script on server in Redis. Our approach has following advantages over Redis:

* Redis being single-threaded does not scale up well with number of cores. We often need to run multiple instances of Redis on each server core. So each server core ends up doing “data aggregation” independently for the same query and each sends filtered data separately to client though all run on the same server.
* As opposed to this Memcached being multi-threaded scales up well with number of cores. Each server performs “data aggregation” just once for a given query and sends very small set of filtered data to client.
* Besides in Memcached there is one more option that is to keep data in de-serialized form, which saves lot of CPU cycles that are generally spent in de-serializing data in query processing critical path. (Though here memcached uses quite a lot of additional memory)

Changes in memcached server:

1. Added two new options to memcached server (-x and –y):

# memcached –h

…

-x <num> -y <filter library path>

           Enable data filtering at server - helps in multi-get operations

           <num> = 1 - Data filtering at server enable (no deserialized data)

                       Data deserialized at the time of query processing

           <num> = 2 - Data filtering at server enable (with deserialized data)

                      Uses more memory but gives better performance. Avoids

Data deserialization at the time of query processing and

                       saves CPU cycles

           <filter library path> - path of filter library 'libfilter.so'

                          This library implements filtering functions and data

                          serialization/deserialization functions

2. Added a new fget command to memcached server:

# fget <filter-library-path/name> <Number-of-Parameters> <List of Parameters – space separated> <key-1>…<key-N>

**Overall Design:**

**Memcached Storage Server**

process\_filterget\_command(

L1.so, 1, {2}, {u1, u2})

{

data <- {u1, u2}

filtered-data <- (L1.so)**readfilter**(

data, 1, {2})

}

**Filter lib: L1.so**

- deserialize()

- free\_msg()

- **readfilter**(data, 1, {2})

{

return filtered-data

}

**Filter lib: Ln.so**

- deserialize()

- free\_msg()

- **readfilter**(data, 1, {2})

{

return filtered-data

}

**Client**

***fget L1.so 1***

***{2} {u1, u2}***

**flitered data**

**flitered data**

…

…

…

Fig: 1 ***dlopen()*** is used to open filter libraries and ***dlsym()*** is used to get pointers to filter libraries entry-point functions (currently 3 of them, *deserialize()*, *free\_msg()* and *readfilter()*) from memcached server. All filter libraries (and their associated function pointers) are ***registered*** with memcached server.

Fig 1 shows the overall design. Application developers have to implement ***filter library***, which performs desired filtering as per the objective function (generally same kind of filtering that is performed at memcached client – like finding 10 latest updates across all friends). Multiple filter libraries can be specified each processing data in a different way as per the desired objective function. There is a provision to pass input parameters to filtering functions. Filter libraries can be added or deleted dynamically without stopping/restarting memcached server.

All filter libraries need to implement three functions:

* deserialize() – function that de-serializes data. Takes input as an array of raw bytes and outputs a structured de-serialized data.
* free\_msg() – function that frees “structured de-serialized data” returned by “deserialize()”
* readfilter – function that takes an array of “structured de-serialized data” (data of all friends) as input, filters the data, serializes filtered data and returns serialized filtered data as output.

These functions are called from memcached server at specific points. For example: readfilter() is called on receiving “fget” command. “fget” retrieves data of all friends from memcached server and passes this data to user defined filtering function readfilter() implemented in filter library. Output of readfilter() is returned as final output by “fget”. Filter library can be developed independently and has no dependency on memcached server code.

Function declarations are given towards the end of the document in section “Filter library function prototypes”. More filtering functions for example pre-processing on write/set could be added in future.

dlopen() and dlsym() framework used by memcached server to interact with user defined filter libraries:

Memcached server opens user defined filter libraries using “dlopen()”. Filter libraries function pointers (for deserialize, free\_msg, readfilter) are obtained using “dlsym()”. Filter library name and its function pointers are stored in a list on memcached server. So, in short all user defined filter libraries (and function pointers) are ***registered*** with memcached server.

In “fget” client supplies *filter-library-name*, parameters to be passed to readfilter() and list of friends. Memcached server looks-up *filter-library-name* in the list and calls corresponding filter libraries readfilter() function. Memcached server passes list of received parameters to the readfilter() function.

Internal changes in memcached:

Introduced a new function *process\_filterget\_command()* in memcached.c to process fget command. All logic/new-changes are contained in this function.

Introduced 2 new files, filter.h and filter.c. These act as an abstraction layer between memcached server and user defined filter libraries. All logic of calling and interacting with filtering functions implemented in user defined filter libraries is contained in filter.h and filter.c

Made following changes in memcached.h:

Introduced a field *msg* (type void \*) in structure *item.* This holds a pointer to “structured de-serialized data” when “memcached with de-serialized data” option is enabled i.e. when memcached server is started with option “-x 2” i.e. “memcached –x 2 –y …”

typedef struct \_stritem {

…

void \*msg; /\* deserialized message \*/

…

} item;

Introduced a field *filteredmsg* (type char \*) in structure *conn.* This holds a pointer to “serialized filtered data” that is returned as output of fget command. This is freed when the connection is terminated.

typedef struct conn {

…

char \*filteredmsg;

…

} conn;

In all following files were modified/added:

configure

Makefile.in

memcached.c

memcached.h

filter.c (added)

filter.h (added)

**Performance:**

Currently changes are implemented for linux platform (tested on linux version RHEL 5.6). Changes made on memcached version "memcached-1.4.13". Changes made for ascii protocol (not for binary protocol), no impact on "gets" (get with CAS) functionality.

Tests performed:

- Setup details:

* 1 memcached server, RHEL 6.1, 64 bit, 16 core, 24 GB RAM, 1 Gb ethernet card
* 1 memcached client, RHEL 6.1, 64 bit, 16 core, 24 GB RAM, 1 Gb ethernet card

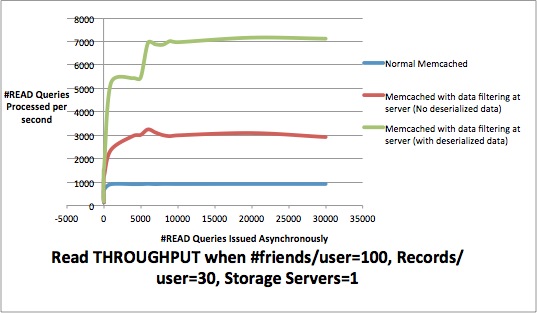
- Test details:

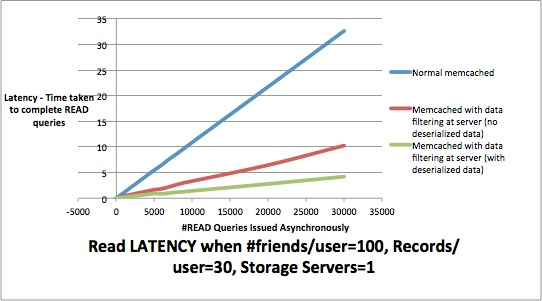
There are one million users (each user represented by a unique key). Each user has 100 friends. Each user has 30 records of type (userId, articleId, timestamp) stored as value. On READ query for a user, all records associated with all friends of that user are READ, sorted in increasing order of timestamp, and top/latest 10 records across all friends are returned as output. So basically on READ query 100 keys (100\*30=3000 records) are read, 3000 records are sorted and top 10 records are returned as output.

- For normal memcached all these operations of READING 100 keys, sorting 3000 records, and finding top 10 records are done on client. For this 3000 records are sent from server to client on each READ query.

- With our changes (where filtering (sorting) happens on server), on server 100 keys are read, 3000 records are sorted locally by filtering function (implemented in user provided library – similar processing is done on server as it is done on client), and only 10 records are sent to client on each READ query.

Created a multithreaded CLIENT application, which issues READ, queries asynchronously (multiple threads are used for issuing and processing READ queries). READ queries are issued for varying number of users starting from 1 user to 30000 users. Time taken to complete these queries is used to compute throughput and latency.





Observations:

Some of the observations from above are (for multi-get queries that need filtering):

- Modified memcached gives better throughput and latency under normal query-load. Improvements are 3X to 7X higher than normal memcached. 3X for “memcached without de-serialized data” and 7X for “memcached with de-serialized data” => can result in client consolidation

- Since most data is filtered on server, very less data flows over network (from server to client). This avoids network saturation (and latencies/delays caused by this), which might happen under high query-load with normal memcached.

We also plan to open source memcached client that was developed for above test.

Data placement algorithms:

In SHEEP we try to place a user and all its friends on same storage server, so that only one storage server is contacted while processing query. This minimizes number of network packets sent and received and also complements data aggregation on server. For this we need to partition a social graph into clusters. Each cluster is a densely connected graph. Property of these clusters is all users in a cluster are densely connected to users in the same cluster and sparsely connected to users in other cluster. Each cluster or partition (and users contained in that) is placed on one storage server, so that just one storage server (in almost all cases, about 95% cases) is contacted for processing multi-get query.

We evaluated following “data placement or graph partitioning algorithms”, using a self-developed tool called “gengraph”. This tool generates a synthetic social graph that has “social networking” like structure or social communities. Advantage of this tool is we know what is the expected output (i.e. expected optimal clusters and their properties) and we can compare this with the output generated by various data placement algorithms and find out which algorithm performs well.

We also plan to open source this tool “gengraph”.

Following algorithms were evaluated:

* + Metis
  + fastgreedy.community (from igraph)
  + walktrap.community (from igraph)
  + spinglass.community (from igraph)
  + edge.betweenness.community (from igraph)
  + leading.eigenvector.community (from igraph)
  + label.propagation.community (from igraph)
  + multilevel.community (from igraph)

Based on four paramteres:

* + Balanced partitions

We want almost all partitions to be balanced (or of equal size) so that each storage server manages almost same number of users. This ensures that RAM of all storage servers is used equally and load is also distributed equally among all servers. Here the assumption is all users have same production and consumption rate, same followers and same subscribers, and all are equally active. So here we try to optimize for storage and not for particular workload (where say one user/celebrity is followed by millions of users and server containing this celebrity receives lot of queries/load as compared to other servers). Replicating user/celebrity data to multiple servers and load balancing queries among these servers can solve this problem. Other option is to assign different weights to different edges and balance partitions based on weights and not necessarily size.

* + Intra-cluster connectivity

This property tells how many users in a cluster are completely connected to users in the same cluster. This value needs to be high.

* + Inter-cluster connectivity

This property tells a user in a partition is connected to users in how many different partitions. For ex: it is fine if a user’s friends span across 2 partitions but may not be good if users friends span across 10 partitions. So basically we need to minimize number of partitions that are spanned by users and minimize total number of such users if at all they exist.

* + Time to produce partitions

We would like the algorithm to run fast.

We found that Metis performs really well and scores very high on all the 4 parameters mentioned above as compared to other algorithms. For ex: Metis was able to partition a social graph containing 1 million users in about 11 seconds and all partitions were almost balanced and close to what was expected. We also plan to evaluate these algorithms on real-world social graphs like flickr and Yahoo! IM.

Social networks are dynamic in nature. Network topology changes constantly as new users are added, new connections are made, existing connection are broken etc. So a user, which is initially in one cluster, may get connected to users in other clusters. We may have to do social graph re-partitioning in order to accommodate these changes and ensure that all connected users/friends are in the same partition. Re-partitioning may involve movement of data from one cluster/storage-server to other cluster/storage-server. We need to ensure that there is very little movement of data across storage servers.

Zoltan is a graph-partitioning algorithm which is built on top of Metis and which does graph re-partitioning/re-balancing in an efficient way minimizing “data transfer” across storage servers.

We could use following logic to decide when to re-partition social graph: Re-partition social graph when more than 10% of queries access more than 10% of storage servers. There is a relation between “when to re-partition” and “data moved during re-partition”. The earlier we re-partition less could be the “data movement” (may not be always true though). But there is a computation cost associated with re-partitioning.

As it is obvious “data-placement/graph-partitioning algorithm” need to be implemented on memcached client. “libmemcached” is a popular C memcached client. By default, libmemcached uses “hashing” to place data on servers. "libmemcached “ is flexible enough where its easy to add new data placement policy. Adding new data placement algorithm requires adding just one file with just one function. We have tested libmemcached with a sample algorithm that uses pre-defined partition table.

Graph partitioning need to run as an offline process and is not part of query processing path. Partitions produced by graph-partitioning algorithm are stored in a file and given as input to memcached client.

**Filter library function prototypes:**

/\*

\* Functions to be implemented by filter library and that are called by memcached server.

\*/

/\*

\* deserialize()

\*

\* Input:

\* buf : Buffer containing data to be deserialized

\* len : Length of buf

\*

\* Return Value:

\* - Pointer to deserialized data buffer in case of success

\* - NULL in case of error

\*

\* Note: "Deserialized data buffer" (return value, void \*) should be

\* freed by caller after use by calling function "free\_msg".

\*/

void \*deserialize(uint8\_t \*buf, int len);

/\*

\* free\_msg()

\*

\* Input:

\* msg : Pointer to buffer (containing deserialized data) to be freed.

\* This pointer must have been returned by function "deserialize"

\* i.e. the deserialized data buffer must have been allocated earlier

\* by function "deserialize".

\*/

void free\_msg(void \*msg);

/\*

\* readfilter()

\*

\* Input:

\* msglist : Array of "Deserialized data/buffer" to be filtered.

\* len : Length of array msglist

\* alloc\_func : Function used to allocate "buf" (buffer containing

\* filtered ouptut data in serialized form). This function

\* is supplied by caller and is similar to "malloc".

\*

\* Output:

\* buf : Buffer containing filtered ouptut data in serialized form

\* buflen : Length of buf

\*

\* Note: Caller should supply function "alloc\_func" similar to "malloc" and should

\* free output buffer "buf"

\*/

void readfilter(void \*\*msglist, int len, void \*\*buf, int \*buflen, void \*(\*alloc\_func)(int), int numParam, char \*\*params);