Project: Memcached Database

INFO-H-415 - Advanced Database

DELBEKE Julien 000425720
HULLEBROECK Nathan 000428458

Université Libre de Bruxelles, Department of computer science

18 December 2018

Contents

1 Introduction

Memcached is a system of memory caching based on the key-value storing principle and the NoSQL paradigm. It is a cross-platform and Open-Source project coded in C that has been released in May 2003. It has been updated and improved continuously as the latest stable version was released in late October 2018 [6]. The main purpose of this system is to speed up other application, mainly web applications, by saving the data in the cache memory which provides fast access. The principle of a key-value storage is to assign a specific key to a specific data which allows the application to retrieve this data by using only its key. [8]

2 Key-value database

Key-value storage is a paradigm developed in order to save, modify or retrieve data. The advantages of this method are its:

- Efficiency, indeed only by knowing the key it is possible to retrieve the data which allow high performance.
- Flexibility in terms of structure which is in tables and where the schema is predefined but also can be re-scaled on demand with little cost.
- Simplicity of use where the commands are only to add, modify, delete or get some data. (See section 5.3 for Memcached commands).

2.1 How it works

It works as a Key-Value Pair (KVP) which consists in a one dimensional table where each keys are linked to a value. To illustrate this, here is a short example:

key	value
Didier	ULB
Arthur	VUB
Eduardo	ICHEC
Charlotte	ESI
Barbara	HELB
Fiona	UCL

The key is the name of the person and the value is the school where this person works, so if you are interested where *Fiona* works, you would have to find *Fiona* in the table and retrieve the linked value to this key. This example shows us that we would need an efficient way of searching for the key.

2.1.1 Linear search

An easy method to find a key in the key-value table would be the linear search which consists in comparing incrementally each key to the wanted key until we find it. This method is inefficient because its complexity is *O(n)*, indeed in the worse case scenario, the key that you are searching for is the last one of the table. In the above example, if we are searching for *Fiona* we would need to compare every keys.

2.1.2 Binary search

A clever way to do it would be to maintain a sorted table on the keys and then we could use the binary search which consists in looking recursively for the key in the middle of the table :

- If the key is the same, we stop.
- If this key is greater, we repeat this method on the superior part of the table.
- If this key is smaller, we repeat this method on the inferior part of the table.

This algorithm is much more efficient than the previous one with a complexity of *O(log n)* and also the most efficient to search a value in a sorted table. This can be proven by using the *Yao's Minmax Principle*.
Here is what our table would look like with the Binary search algorithm:

key	value
Arthur	VUB
Barbara	HELB
Charlotte	ESI
Didier	ULB
Eduardo	ICHEC
Fiona	UCL

If we are still searching for *Fiona*, we would start the algorithm on *Didier* and compare it with the key we are searching and we see that *Fiona* is greater. We then reiterate on the middle key of the superior part of the table which is *Fiona*. We have found the key we were searching for in 2 iterations where the linear search would have found it in 6 iterations.

2.1.3 Hashing

It is a function that consists in converting any type of value to a numerical value of any size depending on the algorithm used. The main idea of this method is that the hashing function must be deterministic which means that for a given input, the result of the hash function must always give the same output.

One of the big advantages of hashing is its efficiency which has a complexity of *O(1)*.

Here is what our table would look like with the Hashing function:

index	value
9	ULB
54	ICHEC
98	UCL
122	VUB
333	HELB
1654	ESI

If we are searching for a specific key like *Fiona*, we should hash this key which would give us the index corresponding to the value we are searching for. Here, the *h("Fiona")* gives us the index *98* which allow us to retrieve the value. One of the downside of a hashing function is collision which occurs when two different inputs give the same output, that is why it is necessary to use a good hashing algorithm which can prevent this from happening. Even with an algorithm that results in some collisions, there are some mechanisms to resolve this issue but with cost to its efficiency. For example, rather that storing the value at a specific index, we store a list of values corresponding to this hash and then use the linear/binary search on this list, but as mentioned earlier, it would obviously increase the cost.

Another downside of using a hashing function is the memory space required which is greater than the required space of storing the values. Indeed, a hashing function uses a table where the value are not subsequent which gives empty spaces between each values. The *Rule of thumb* is used in this case:

- Try to keep utilization of space between 50% and 80%.
- If $< 50\%$, wasting space.
- If $> 80\%$, high risk of collision depending on the hash function used.

Moreover, in order to avoid collisions, we choose to increase the size of the index which use even greater memory space for the table. Fortunately there are some dynamics algorithm that can improve this problem by looking to a smaller portion of the hash in order to decrease the possible indexes, like the Linear hashing or the Extensible hashing.

3 Cache memory

The cache memory is a small hardware component that allows a faster read in memory. It is a memory used by the CPU (Central Processing Unit) to reduce the costs or reading in the hard drive and stores on it frequently accessed data by applications or programs. It is faster for two main reasons:

1. Location: the cache memory is located near the CPU which allows smaller travel time to get information.
2. Type of RAM: the cache memory is composed of SRAM (Static Random-access Memory) and not DRAM (Dynamic Random-access Memory)

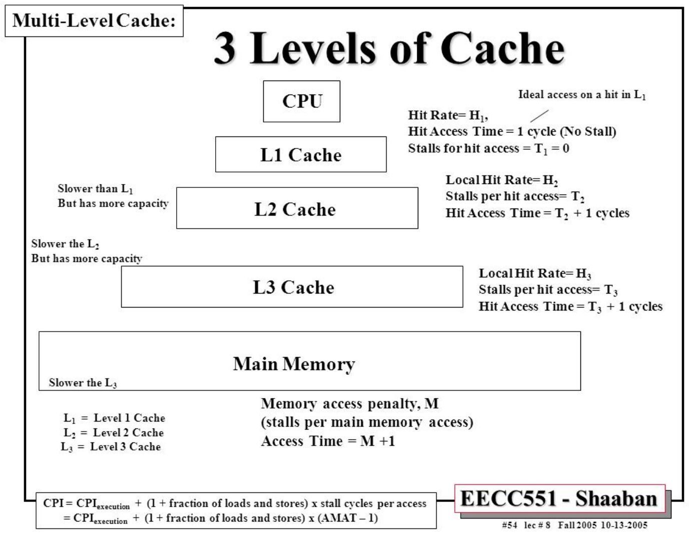
Unfortunately a very efficient product is very expensive, that is why it is usually very small in terms of capacity. When we want to access some data in the cache, we first need to look if it is really stored so there are two possible scenarios:

- Cache hit : means that we find what we are looking for.
- Cache miss : means it is not stored in the cache memory.

When we have a cache miss we need to get the information in the main memory.

3.1 Multi-level caches

To increase the size of the cache a method is to have multi-level of caches. Like we said, the bigger the size of the cache is, the bigger the cost is, but in order to have bigger caches, we decrease the efficiency to keep an affordable price. Here we have different caches sorted by size. So, first we look if what we are searching for is in the first level, if it is we are done, if not we need to look on the next level. We repeat this until we are at the last level. When we are on the last level and we have a cache miss we look on the main memory.


Multi-Level Cache:
3 Levels of Cache
CPU
L1 Cache
L2 Cache
L3 Cache
Main Memory
Ideal access on a hit in L1
Hit Rate= H1,
Hit Access Time = 1 cycle (No Stall)
Stalls for hit access = T1 = 0
Slower than L1
But has more capacity
Local Hit Rate= H2
Stalls per hit access= T2
Hit Access Time = T2 + 1 cycles
Slower the L2
But has more capacity
Local Hit Rate= H3
Stalls per hit access= T3
Hit Access Time = T3 + 1 cycles
Slower the L3
Memory access penalty, M
(stalls per main memory access)
Access Time = M +1
L1 = Level 1 Cache
L2 = Level 2 Cache
L3 = Level 3 Cache
CPI = CPIexecution + (1 + fraction of loads and stores) x stall cycles per access
= CPIexecution + (1 + fraction of loads and stores) x (AMAT – 1)
EECC551 - Shaaban
#54 lec # 8 Fall 2005 10-13-2005

4 NoSQL

NoSQL is a paradigm of database that distant itself from the traditional SQL and the relational database. It means *Not only SQL* because it may also accepts SQL queries. It is a new way of thinking of the database that wishes to be more accurate regarding the use of the user because relational database are not always the best way to store the data.
NoSQL and SQL have made different choices:

1. SQL Store everything in tables so it needs to have a format and we need to know in advance what we want to store for every table. It is then harder to make changes and to be flexible, for example: if we want for a given value different kind of formats like both Integer and String, that would not be possible in a relational database. NoSQL uses Collections that can allow it.

2. Usually in SQL we need to give the information in small size to avoid the duplication of value but the information is splited and thus needs more time to be recomposed. NoSQL stores it as collections so the data is not partitioned but has duplications and then takes more memory space but is faster to access.

3. The scalability in SQL is easier to scale horizontally by adding new columns than to scale vertically and in NoSQL it is the opposite.

There are other differences but we will not explicit all of them here.

In key-value storing it is really interesting to avoid formatting, indeed the key and the value can change, sometime we want to store picture, text or code... we don't want to add tables for every type of key or value that we want to store.

It is also not really interesting to scale horizontally because we want to add new item rather than add information on the item that are already inside the database. The user also don't want to remove duplication and start to uncouple the values, he is more interested by speed than memory use.

So it looks more interesting to use NoSQL rather than SQL in a key-value database.

5 Memcached

Memcached is based on a client server architecture it work with TCP and UDP. It is a distributed server which means that they are many servers and is then not centralized. When the client wants to add or receive information it first needs to hash the key and choose a server regarding the value of a hash.The values will be stored in RAM and removed when more memory is needed.

The fact that we use multi-server help us, first to have more memory available and also to have a better use of the memory. From the client point of view all the servers are in the same virtual pool so it can choose the server that it wants and doesn't have to worry where exactly the data is stored. It also helps when we want to add servers, it is fairly easy and cost effective, so if we need memory we can add some. The keys can be at most 250 bytes and the values 1 megabyte long. It also compiles with SASL to have an authentication support if we need security. [5]

5.1 How it works

To receive a value assigned to a key:

1. The client computes the hash of the key to find the server that stores the value.
2. The client makes a request on the appropriate server.
3. The server computes the hash of the key.
4. The server checks on its table if the key is stored.
5. (a) If the server has the value assigned to the key it will return this value to the client.

   (b) If the sever has not the value it will tell it to the client.

To store a new value:

1. The client computes the hash of the key to find the server that will store the value.
2. The client makes a request on the appropriate server.
3. The server computes the hash of the key.

4. The server saves the value at the index corresponding to the hash.

To remove an old value:

1. The client computes the hash of the key to find the server that stores the value.
2. The client makes a request on the appropriate server.
3. The server computes the hash of the key.
4. The server removes the value at the index corresponding to the hash if it is the right key.

In general, Memcached is uses to speed up a server, so we will only use the insertion and the receive operations. We will attribute an expiration time to automatically remove data if they are not requested for a long period using the *Least Recently Used (LRU)* algorithm.

1. Client makes a request.
2. The Memcached client will look if the result of the request is stored on the Memcached server.
3. (a) If the value is stored we return the result to the client, it is a cache hit. We can also increase the expiration time because the value of the request is relevant.
   (b) If the value is not stored, we will tell the client, it is a cache miss.

   If we have a cache hit we have finished.
   If we have a cache miss we need to add some new steps.
4. We make a request to the server.
5. The server computes the result of the request.
6. The server returns the result to the client.
7. The Memcached client will store the result and the request as the value and the key in the Memcached server with some expiration time.

5.2 Hash algorithm

They are two side in Memcached :

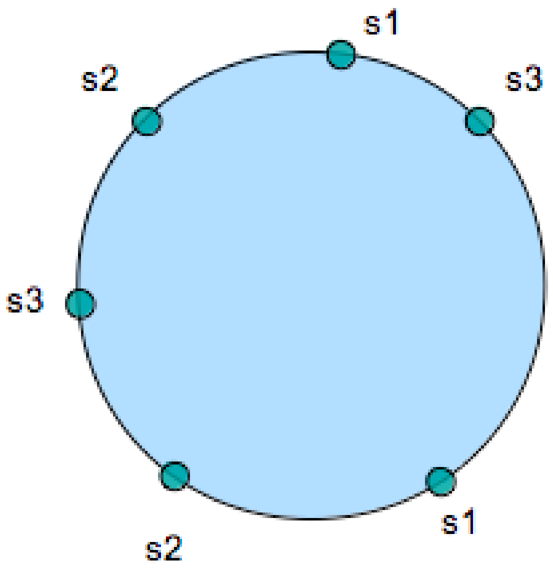
1. Client side
2. Server side

5.2.1 Client

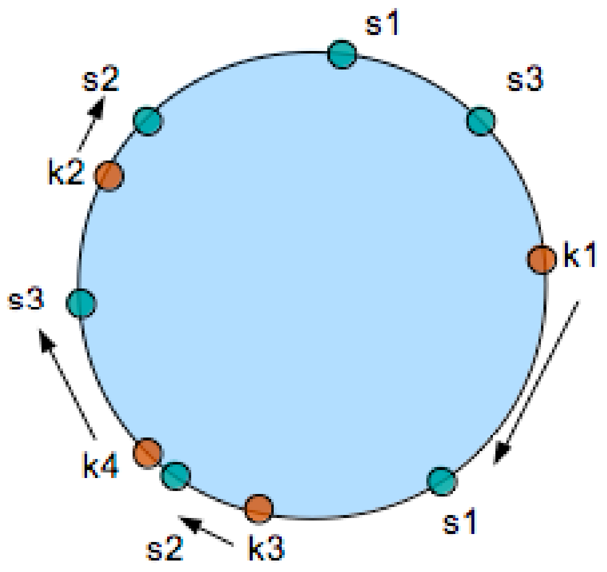
The algorithm available on the client side will depend on the implementation. It can change depending on the language or the version that is used. Every client has at least one hashing function but not necessarily the same. It may cause some problems and the different clients can be incompatible. Indeed, if they don't have a common hashing algorithm they can use the data of the other client: for the same key they will choose different servers. There is also another problem on the client side, the number of the servers may change, so we need to have a consistent hashing. When we add a new server in the pool, it can create a lot of cache misses because we no longer choose the right server. With a consistent hashing we should remap less than 10% of the keys while without it, it is around 40% of the keys than would need to be remapped.

Consistent Hash A consistent hashing is a hashing that is more interesting when the size of the hash table changes, it allows to remap only $\frac{K}{n}$ in average. K is the number of key and n the size of number of slots. A non consistent hashing function needs to remap all the keys in general [4].

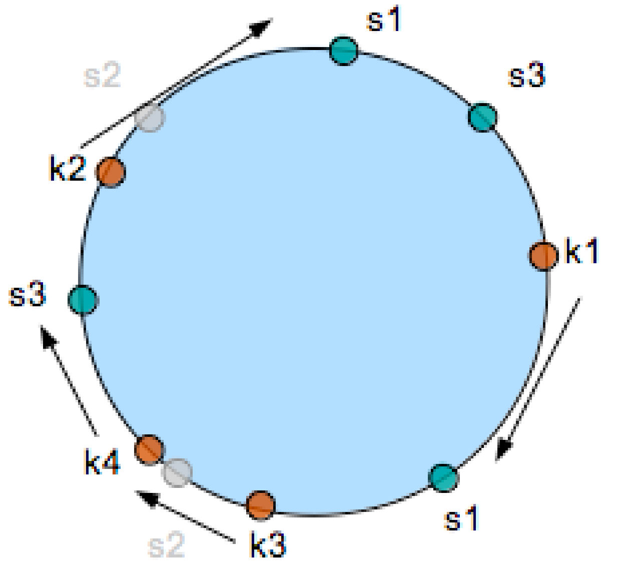
The main idea is to associate each value to points in a clock-circle, called the continuum. Each value is at several point on the circle, we add every value in different random positions. The s_i corresponds to the values that are saves in the hash table.



So when we want to find the value, we hash the key and look for the next value stored that is higher or equal in the continuum. If there is none, we take the first. The k_i corresponds to the key for which we want find the value.



When we want to add or remove a new value we add/remove it on the circle so only the value that are impart whit the new/removed points are impact by the change.



5.2.2 Server

On the server side there is no need to use consistent hashing because every server is independent so we only want to have the best hashing function:

1. Efficient.
2. Few number of collisions.

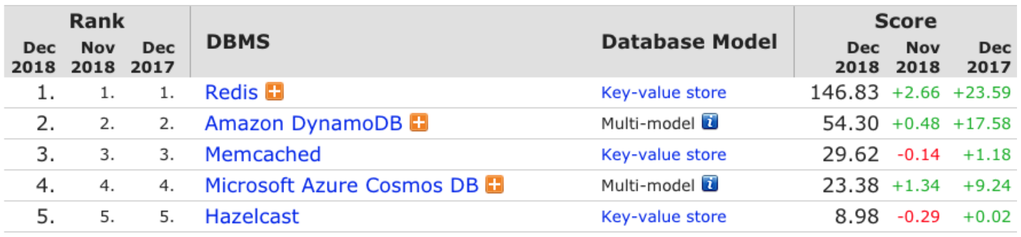
5.3 Commands

Here are some of the main and most useful commands [1]:

Command	Action	Example
get	read a value	get key
set	store a key-value pair	set key <flag> <exptime> <bytes> value
replace	replace a key-value pair	set key <flag> <exptime> <bytes> value
append	add after an existing value	append key <flag> <exptime> <bytes> value
prepend	add before an existing value	prepend key <flag> <exptime> <bytes> value
cas	update only if not updated since last fetch	cas key <flag> <exptime> <bytes> <cas unique> value
delete	delete the key-value pair	delete key
incr	increment the value corresponding to the key	incr key <value to increment>
decr	decrement the value corresponding to the key	decr key <value to increment>
touch	update exptime if exist without fetching	touch key <exptime>
stats	show every statistics	stats
version	return the Memcached server version	version
quit	server closes	quit

5.4 Popularity

Memcached is ranked as the 3rd most popular Key-value store database according to DB-Engines [2].



Rank			DBMS	Database Model	Score		
Dec 2018	Nov 2018	Dec 2017			Dec 2018	Nov 2018	Dec 2017
1.	1.	1.	Redis	Key-value store	146.83	+2.66	+23.59
2.	2.	2.	Amazon DynamoDB	Multi-model	54.30	+0.48	+17.58
3.	3.	3.	Memcached	Key-value store	29.62	-0.14	+1.18
4.	4.	4.	Microsoft Azure Cosmos DB	Multi-model	23.38	+1.34	+9.24
5.	5.	5.	Hazelcast	Key-value store	8.98	-0.29	+0.02

In terms of comparison with the same kind of database model which is key-value store, the main competitor would be Redis which is better than Memcached is some areas like in terms of functionalities but Memcached has better performances in time and multi-threading capacities [3].

6 Application

6.1 Architecture

We are going to show the advantages of Memcached, to do so we are going to make a comparison between queries using Memcached and without using it, with the data stored in a relational database. To build a database with some information, we took a dataset sample from IMDb [7] composed of:

Type of data	Quantity
Movies	4912
People	8513
Actors	6279
Directors	2396
Genres	26
Languages	101
Countries	79
Years	93
Plot keywords	8029

We then parsed the original CSV file and organized the data into multiple tables and structured them in a logical way as shown in the diagram below. In this architecture, there is a table Person from which Actor and Director inherit and who are linked to a movie by a table ActorMovie or DirectorMovie respectively. We have also created few more tables to complete this architecture for languages, genres, countries,... this dataset seemed interesting because of the number of possible relations and large queries, for instance there are 14736 links between Actors and Movies.

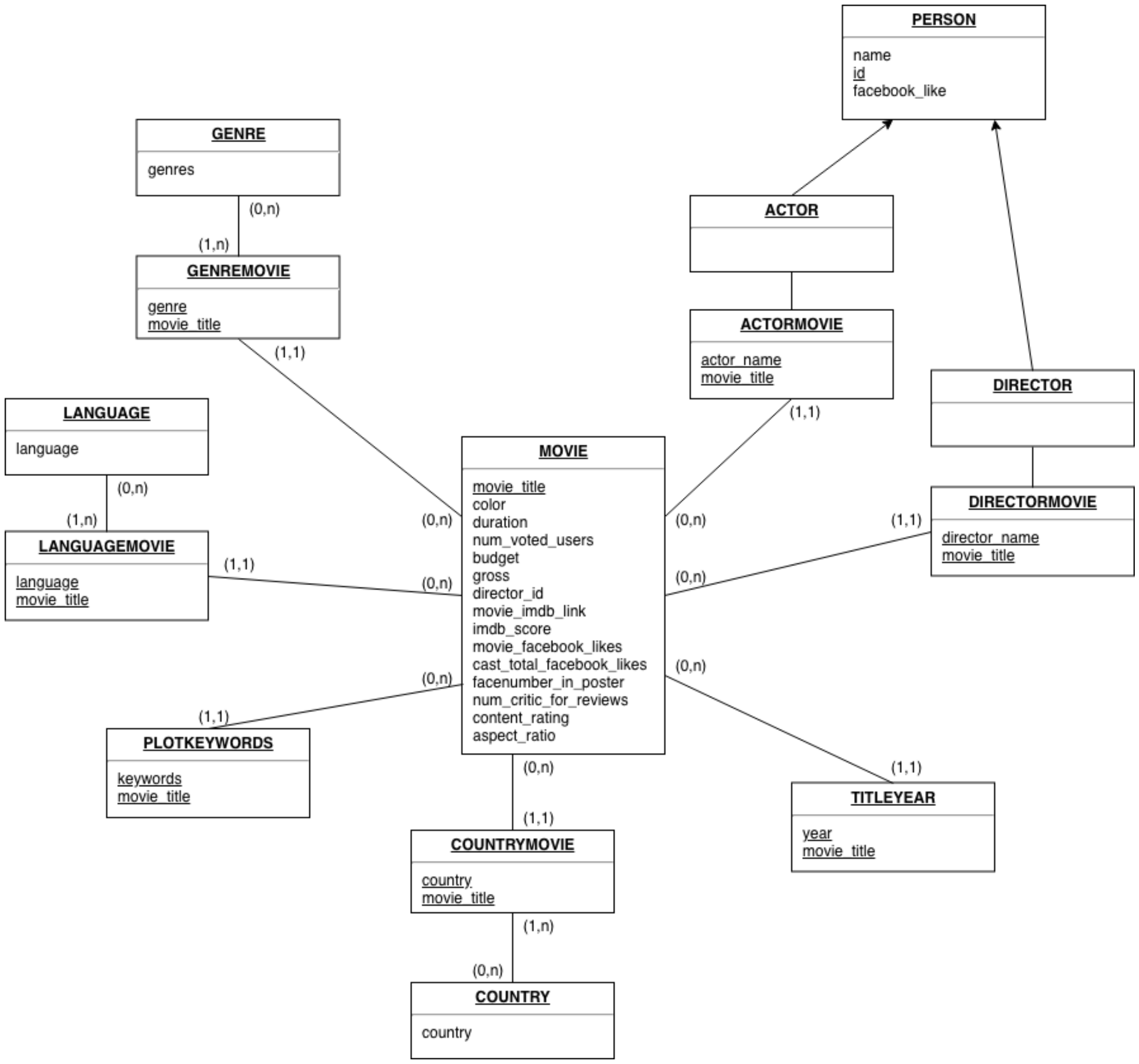


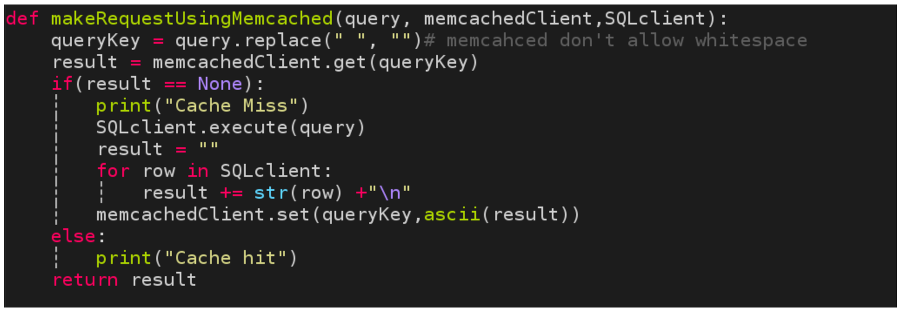
Figure 1: Relational database diagram

6.2 Test plan

In order to point out the differences and the advantages of choosing a key-value store database using the cache memory, we implemented two different functions:

- The first one is used to make a Memcached request to check if the wanted key is stored, if it is stored it is a cache hit, if not it is a cache miss and

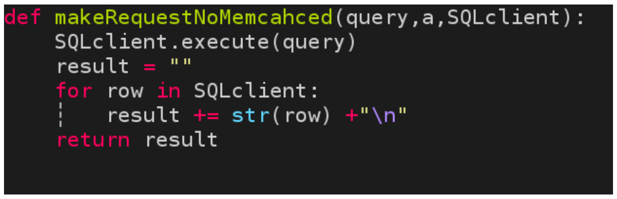
Memcached will then store the data in the cache for a possible future request, it is the same as explained before in section 5.1.



```
def makeRequestUsingMemcached(query, memcachedClient,SQLclient):
    queryKey = query.replace(" ", "")# memcahced don't allow whitespace
    result = memcachedClient.get(queryKey)
    if(result == None):
        print("Cache Miss")
        SQLclient.execute(query)
        result = ""
        for row in SQLclient:
            result += str(row) +"\n"
        memcachedClient.set(queryKey,ascii(result))
    else:
        print("Cache hit")
    return result
```

Figure 2: Function with Memcached

- The second one doesn't use Memcached and makes then same query to the traditional database which is store in the hard drive of the computer.



```
def makeRequestNoMemcahced(query,a,SQLclient):
    SQLclient.execute(query)
    result = ""
    for row in SQLclient:
        result += str(row) +"\n"
    return result
```

Figure 3: Function without Memcached

We are going to execute different SQL queries with and without Memcached as shown below:

The goal of such queries is to show the difference between a cache hit and a cache miss for the key-value store. Indeed, if it is a cache miss, the program will make the request into the traditional relational database.

1. The first query consists on retrieving all the countries in which a movie has been shot and where the languages are either Greek, Kazakh, Russian, Swedish, Arabic or French.
2. The second query consists on retrieving all the actors with all their movies and all the years on which they acted.
3. The third query consists on retrieving the same information as the previous one but it will request all the actors for each letter in the alphabet, so it will make 26 requests in total.
4. The fourth query consists on retrieving all the movie titles where the budget was smaller than the gross.

5. And the last one is to get all the information for the movie 'Avatar'.

6.2.1 Queries

Here we can find the SQL script of those queries:

1.
   ```
   SELECT c.country
   FROM Movie m
       INNER JOIN CountryMovie mc
       ON m.movie_title = mc.movie_title
           INNER JOIN Country c
           ON c.country = mc.country
               INNER JOIN LanguageMovie ml
               ON ml.movie_title = m.movie_title
                   INNER JOIN Language l
                   ON l.language = ml.language
   WHERE l.language in ('Greek', 'Kazakh', 'Russian', 'Swedish', 'Arabic', 'French');
   ```

2.
   ```
   SELECT p.name , m.movie_title, y.year
   FROM Actor a
       INNER JOIN Person p
       ON a.name = p. name
           INNER JOIN ActorMovie am
           ON a.name = am.name
               INNER JOIN Movie m
               ON m.movie_title = am.movie_title
                   INNER JOIN YearMovie ym
                   ON ym.movie_title = m.movie_title
                       INNER JOIN Year y
                       ON y.year = ym.year ;
   ```

3.
   ```
   SELECT p.name , m.movie_title, y.year
   FROM Actor a
       INNER JOIN Person p
       ON a.name = p. name AND a.name LIKE  'A%'
           INNER JOIN ActorMovie am
           ON a.name = am.name
               INNER JOIN Movie m
               ON m.movie_title = am.movie_title
                   INNER JOIN YearMovie ym
                   ON ym.movie_title = m.movie_title
                       INNER JOIN Year y
                       ON y.year = ym.year ;
   ```

Of course here A will change in accordance with the letter that we want to find.

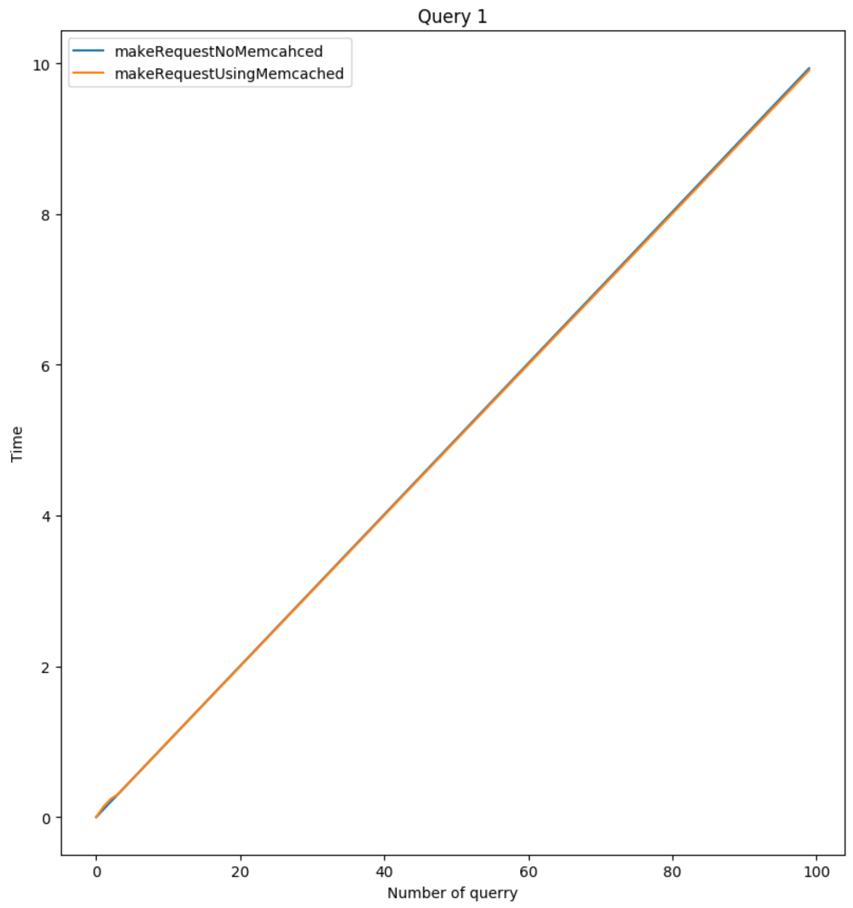
4.
   ```
   SELECT movie_title
   FROM Movie
   WHERE  budget < gross;
   ```

5.
   ```
   SELECT *
   FROM Movie
   WHERE  movie_title = 'Avatar';
   ```

We made an experimentation for all the queries: we ran different number of times the same query on the database and compared the execution times needed. We ran 10 times every experimentation to make an average of the result. We made 1 to 100 executions once using Memcached and another without using it.

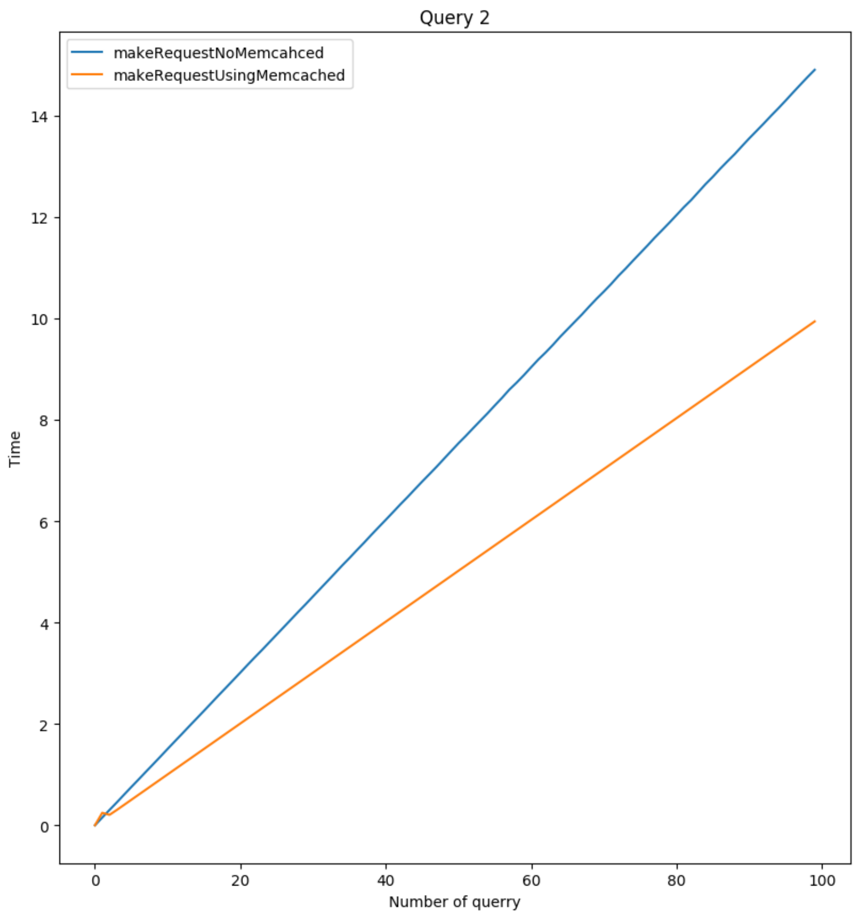
6.3 Test results

1. Query 1:



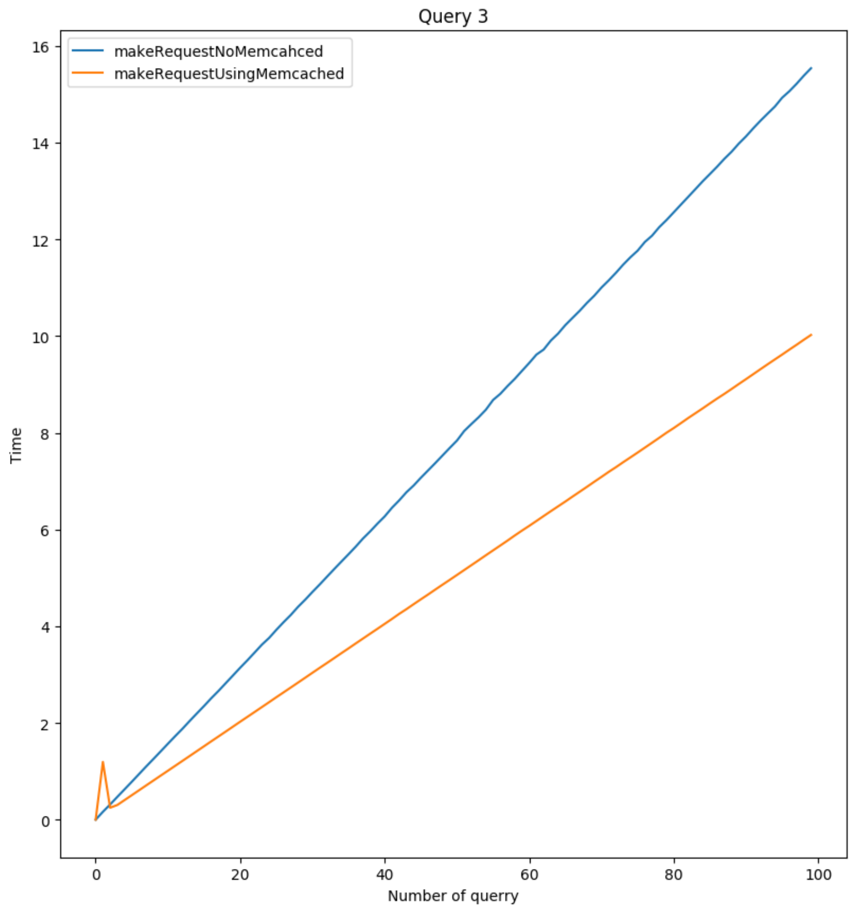
First we can see than both are represented with linear segments. So if we increase the number of equations it will grow linearly. We can see that when there are only few executions, Memcached increases the time needed for the request: it needs to look if the data is already stored, after it needs to store the data and all of that takes more time. If we made many times the same request we can see that both take the same amount of time. Finally we can see that at the end, using Memcached is a little better and we can presume that the gap between the two will still grow if we increase the number of queries.

2. Query 2:



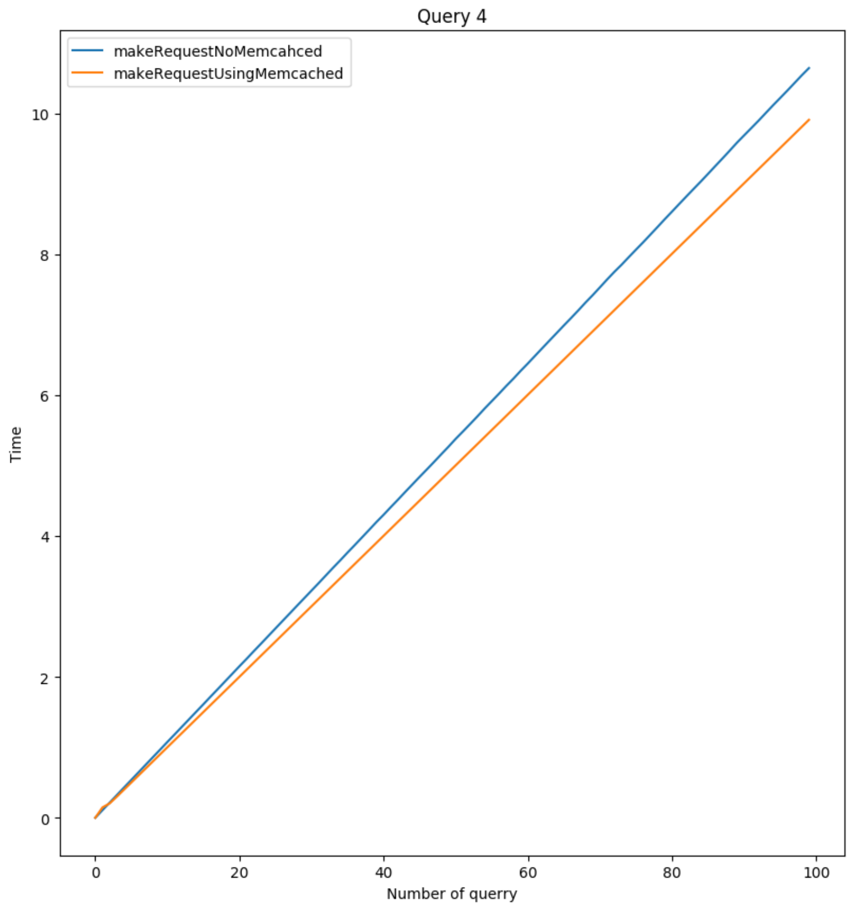
Here we have the same linear progression but the gap between the two is greater so the utility of Memcached is higher. We can still see that at the beginning, without Memcached the time performance is better but it is a really small difference and only for the first few requests.

3. Query 3:



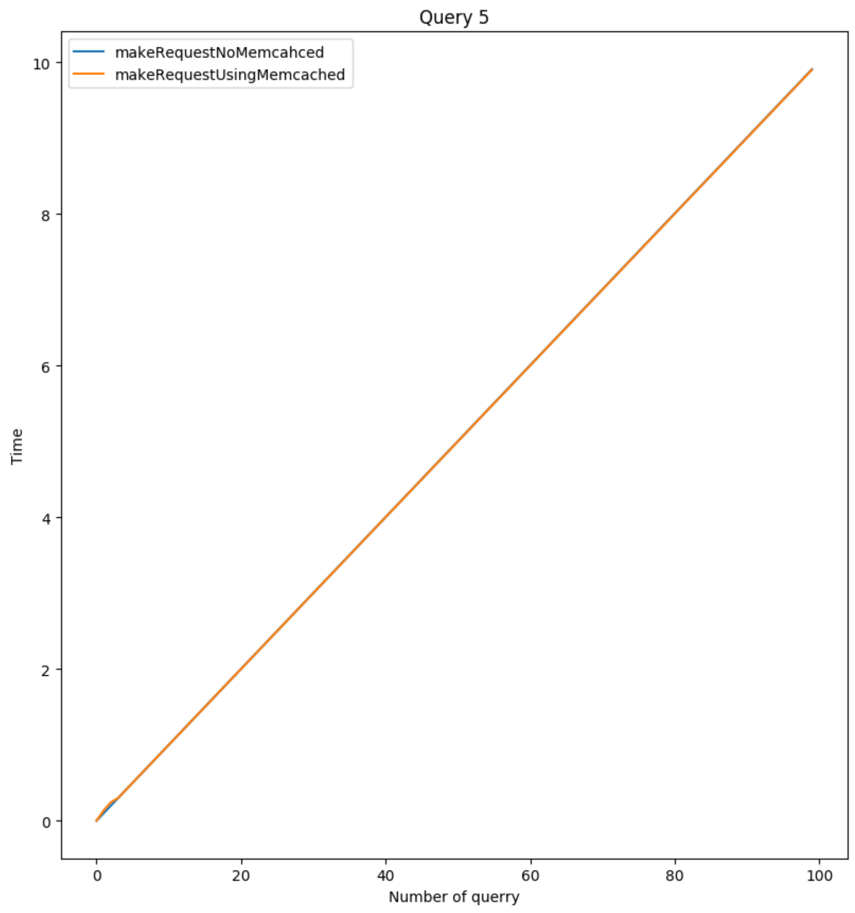
The cost at the beginning is really high here because we made 26 different queries so we need to store all the results and we have 26 cache misses so the cost is indeed higher. But the higher the number of queries are, the higher the gain for the cost is. Here we have also more memory usage because we need to store the results in Memcached. So we can store multiple times the same data because it is the result of multiple different queries because we ask for different letters.

4. Query 4:



The gap between the two is really small and depending of the number of queries it can be negligible so it can be useful not to use Memcached to avoid storing multiple times the same data. Note that for really small numbers of queries, using Memcached is costlier.

5. Query 5:



We ran this query that is really simple and really fast to show that Memcached is not useful every time because it gives the same results and can be worse depending of the number of queries.

During those different tests and experimentation we observed that there are 3 possibilities:

1. Memcached does not change the speed.
2. Memcached helps a lot by improving the speed.
3. Memcached makes the thing worse.

So we can see that the use of Memcached is not necessary every time and can improve a lot but it depends on the scenario and the use of such technology. If we have only few requests and never or rarely make the same request, Memcached is not interesting. The true interest is when we need to make a lot of times the same request that needs a lot of time to be completed. The query 1 and 5 show it clearly but of course if the number of times the request is made is really huge, the interest increases because at the end of the query 1 we can see that a gap between the two lines is developing. If we make a small number of time the same request, it only depends on the memory and time needed to store the result. Note that in every cases, Memecached increases the time needed for really small number of uses.
One of the issue is the choice of key in Memcached, if we use the query as key, it may cause a problem if the number of characters is too big, indeed we can only store 250 bytes for the key. It is still possible to resolve this problem in different ways: we could enumerate the query and store the number of the query and the parameter but it can still be too big depending on the parameter. Another problem occurs when we change the original values, we need time to change it for the user because if we have a cache hit we will have the old values stored on the cache, it is of course possible to remove the value in the cache but it takes time to compute where the value is store and to remove it.

7 Conclusion

We can conclude that Memcached can be really useful but not in every cases. Indeed, to be useful we need to have the same request asked a lot of time or the request needs to be costly in time. The more the request needs time to be completed, to more Memcached will be useful in terms of speed. It is not the case for every scenario, the value also needs not to be modify every time because if it is, Memcached will only store and remove data in the cache without using its properties effectively.
We also need to be aware that Memcached can increase a lot the memory needed because it will store every results and some value multiple times, of course this number will depend on the expiration time. The expiration time needs to be short to avoid storing data not used for long time but it also needs to be big to avoid removing data too early and then create a lot of cache misses. So we can really understand why the websites like : Facebook, Twitter, Reddit ,... need to use this technology because they have a lot of users who make the same requests over and over.

References

[1] Elija Halii Adalier. Memcached telnet command summary. `https://blog.elijaa.org/2010/05/21/memcached-telnet-command-summary/`.

[2] DB-Engines. Db-engines ranking. `https://db-engines.com/en/ranking`.

[3] Disko. Redis vs memcached. `https://www.disko.fr/reflexions/technique/redis-vs-memcached/`.

[4] Joshua and Jekyll Thijssen. Memcache internals. `https://www.adayinthelifeof.nl/2011/02/06/memcache-internal/`.

[5] Memcached. Github repository of memecached. `https://github.com/memcached/memcached/wiki`.

[6] Memcached. Memcached - official website. `https://memcached.org/`.

[7] Chuan Sun. Imdb 5000 movie dataset. `https://data.world/popculture/imdb-5000-movie-dataset`.

[8] Wikipedia. Memcached. `https://en.wikipedia.org/wiki/Memcached`.