

# Distributed Food Order Management system with Caching

## 1) Introduction

Online food delivery systems have seen a tremendous growth in the past decade, emerging as a strong alternative to the long standing tradition of visiting restaurants for eating food. Now, the users can enjoy the same delicacies from the comfort of their homes without having to endure inconvenient traveling and long queues. In this project, we aim to build an architecture for a distributed food ordering system that utilizes caching for efficient performance. Specifically, we are interested in analysing the impact of caching on the performance of our system.

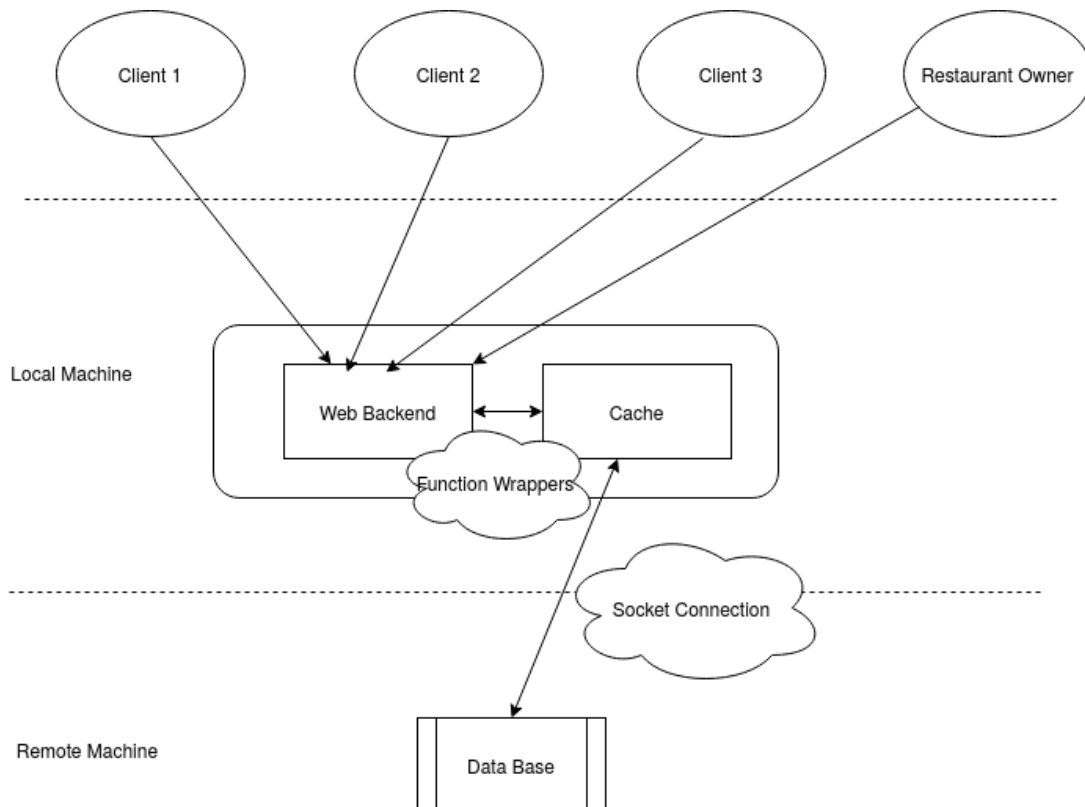
## 2) Motivation

Most distributed service oriented systems use the traditional client server architecture, where the client (customers or restaurant owners) device runs the web backend and a remote server hosts the database of user profiles, order details, inventory of restaurants etc.

Naturally, every request made by the client has to be serviced by the remote server, leading to very large service latencies. The experience would be even worse in slow internet areas and can spoil the company's reputation.

In this project, we tried to experiment by introducing a small cache at the local node running the web-backend and measuring the performance improvement in serving the requests. Using custom designed test cases, we try to gauge the reduction in average latency for different cases and replacement policies.

## 3) System model



As shown in the above diagram, the three principal components of our system architecture are web backend, cache and database. Web backend and cache are hosted on a local machine while the database is hosted on some remote node. As we are food order management services, we have diverse clients ranging from consumers, restaurant owners, food mart, etc. All of them use the web frontend which can be readily accessed as a mobile application or a web browser to connect with our system and place their respective requests. Consumers would order food items while restaurant owners would update the available food inventory. The web frontend connects to the web backend under the hood which in turn is connected to the cache management system using C++ wrapper functions. The cache module communicates with the database through socket programming. The cache module acts as a client while there is a database server running on the remote host which continuously listens for client requests. Client uses session based connection to connect with the database server so that delay due to frequent opening and closing of service ports is lessened. There is no exchange of information whatsoever between the web backend and the database. Every request to the database is routed through the cache module and hence, cache lies on the critical access latency path. The server has been designed to handle multiple connection requests simultaneously through creation of a child process to cater each such connection.

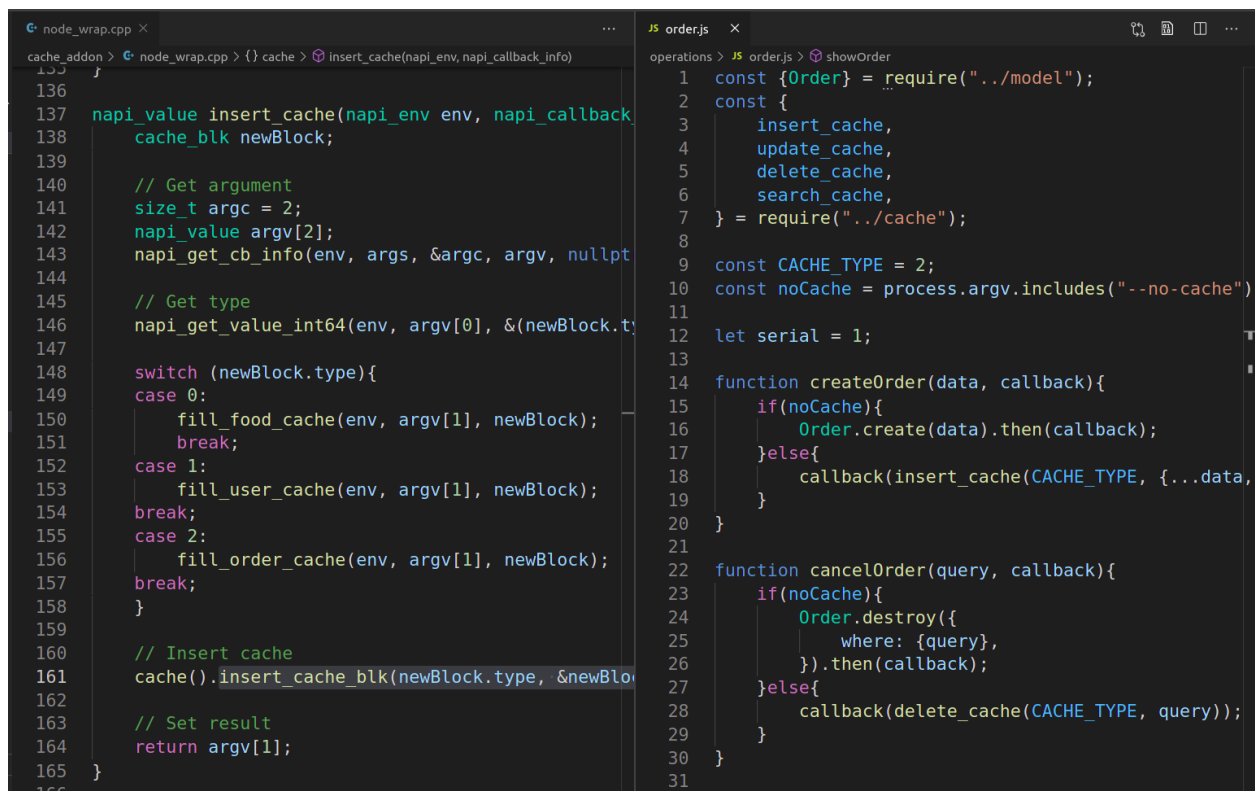
#### **4) Implementation/Approach**

We implemented the entire architecture from scratch using C++ and Node.js. The only external help came in the form of C++ wrapper functions at the web backend and socket functions for placing requests to the database. The source code of the entire architecture is present at this GitHub Link: <https://github.com/radhit91/Food-Order-Management>. The following sections describe the individual modules in depth.

##### **Web Backend**

The Web Backend is the main logic part of this project. It accepts HTTP requests from the client, performs relative computations to the request and sends responses to the user. If the operation needs to access storage, Web Backend will access cache to retrieve/store data.

In our system, Web Backend resides in the same machine with cache, so that we can pass data by function call. We implement the Web Backend with Node.js, and Node.js supports C++ addon that we can wrap our C++ implementations into a library for Node.js to use. Though there may be some overheads while porting from different frameworks, passing data by function call is still very efficient than other communication methods, especially better than making real I/O. In distributed systems, the components are possibly implemented with different frameworks or even different programming languages. Bridging or communicating between these parts is often an issue while integrating a variety of parts. Normally, if there's no such bindings supported in the framework, we have to make them into different processes, and use some Inter-process communication (IPC) methods to communicate, or use a file as the interface. These methods need to involve real I/O operation and more latency. It will be better if there's an opportunity to use function call in distributed systems that leverage the supports from the frameworks.

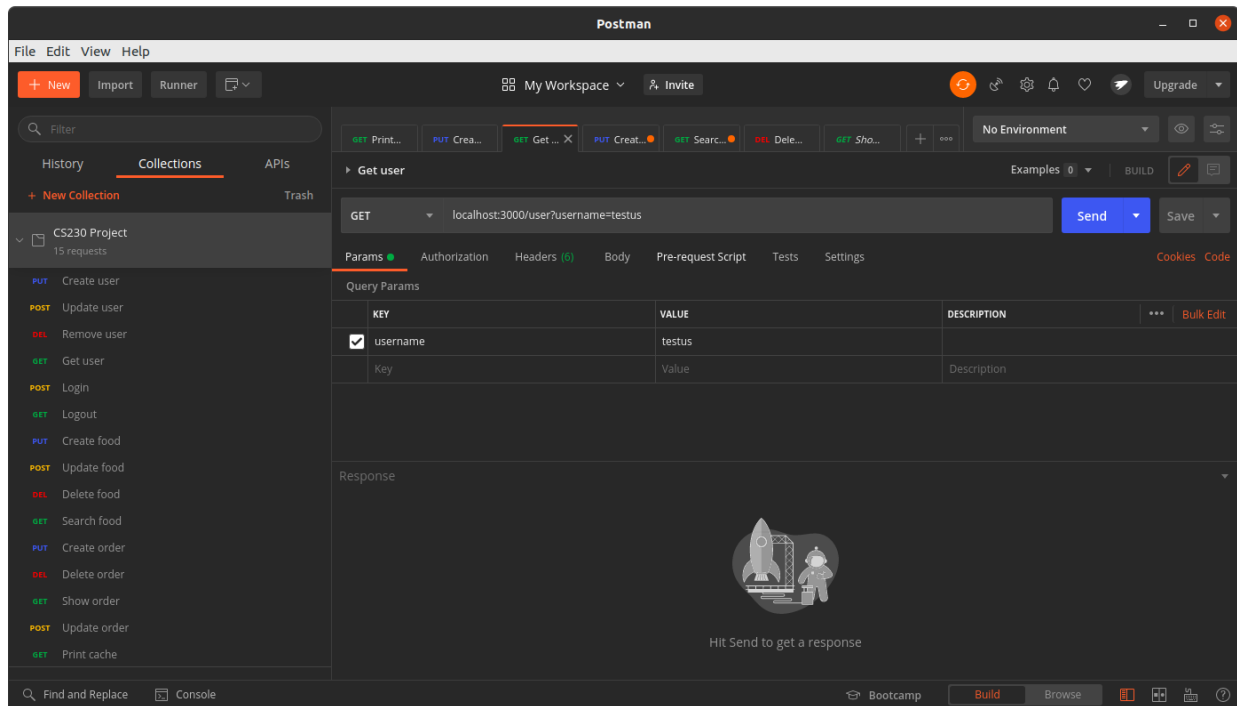
The image shows a code editor with two panels. The left panel, titled 'node\_wrap.cpp', contains C++ code for a Node.js native addon. It defines a function 'insert\_cache' that takes a napi\_env, a napi\_callback\_info, and a napi\_value. It uses 'napi\_get\_cb\_info' to get arguments and 'napi\_get\_value\_int64' to get the cache type. A switch statement handles different cache types (0, 1, 2) by calling 'fill\_food\_cache', 'fill\_user\_cache', or 'fill\_order\_cache'. It then calls 'cache().insert\_cache\_blk' and returns the result. The right panel, titled 'order.js', contains JavaScript code. It imports an 'Order' model and a 'cache' module. It defines a 'showOrder' function that calls 'createOrder' or 'cancelOrder' based on the 'no-cache' flag. 'createOrder' calls 'Order.create' and 'cancelOrder' calls 'Order.destroy'.

```
136
137 napi_value insert_cache(napi_env env, napi_callback_info info) {
138     cache_blk newBlock;
139
140     // Get argument
141     size_t argc = 2;
142     napi_value argv[2];
143     napi_get_cb_info(env, info, &argc, argv, nullptr);
144
145     // Get type
146     napi_get_value_int64(env, argv[0], &(newBlock.type));
147
148     switch (newBlock.type) {
149     case 0:
150         fill_food_cache(env, argv[1], newBlock);
151         break;
152     case 1:
153         fill_user_cache(env, argv[1], newBlock);
154         break;
155     case 2:
156         fill_order_cache(env, argv[1], newBlock);
157         break;
158     }
159
160     // Insert cache
161     cache().insert_cache_blk(newBlock.type, &newBlock);
162
163     // Set result
164     return argv[1];
165 }
166
```

```
1 const {Order} = require("../model");
2 const {
3     insert_cache,
4     update_cache,
5     delete_cache,
6     search_cache,
7 } = require("../cache");
8
9 const CACHE_TYPE = 2;
10 const noCache = process.argv.includes("--no-cache");
11
12 let serial = 1;
13
14 function createOrder(data, callback) {
15     if (noCache) {
16         Order.create(data).then(callback);
17     } else {
18         callback(insert_cache(CACHE_TYPE, {...data,
19             serial: serial++
20         }));
21     }
22 }
23
24 function cancelOrder(query, callback) {
25     if (noCache) {
26         Order.destroy({
27             where: {query},
28         }).then(callback);
29     } else {
30         callback(delete_cache(CACHE_TYPE, query));
31     }
32 }
```

The left side wrap C++ operations into Node.js native addon, and the right side is the usage in JavaScript code

For experiment, we actually do not implement an user interface on the client side. Instead, we use a tool, Postman, to make HTTP requests and pass requests to the backend. In our scenario, there are different types of users, including customers, restaurants ... etc. They may use different kinds of clients to access the backend, so that we shouldn't put too much dependencies on the client. In fact, we can still develop an user interface with care about dependencies, however, using tools can not only simplify the efforts but also force us to concentrate on the communications. Furthermore, if we want to test our systems with batch jobs or other auto-generated methods, we can replace the tool with any other specialized applications that are able to send HTTP requests.



The Postman tool and our collection of Web APIs

## Cache (Write back, Write Allocate)

There are three types of requests that a client can make: food, user and order. Food requests correspond to addition/modification/deletion of attributes of a new food item. The food request includes attributes like food\_id, food\_name, food\_qty, food\_price, discount, popularity, etc. These requests are generally placed by restaurants to change inventory. User requests are related to profile creation at the time of registration. Its attributes include username, passwd, email\_address, home\_address, phone\_num, etc. They also come in handy during web authentication of a user. Order requests are generated by consumers who are verified customers of our service. Each order request has some food items that the consumer wishes to add/modify/delete. The unique identifier for this case is order\_id while username, food\_id, food\_name, food\_qty, etc. are included in such a request. Our cache is responsible to hold these three types of requests as cache blocks and provide them to the user as and when needed.

Cache is implemented in the form of a hash table. It is indexed by username except for the very first set which is reserved for food requests as they are not associated with users. User and order requests for a particular consumer will form a queue at the index referred to by its username. For each cache block, appropriate counters are used to keep track of LRU time, LFU frequency, dirty status. For each set indexed by a username, there are a set of attributes that keep track of no. of valid blocks in that set, minimum time/frequency count for that set (useful during eviction), etc. There are many features that monitor overall status of cache like cache\_filled, num\_users cached, etc. The purpose of keeping all these counts is to help

manage the cache better and to compute a victim for replacement. The implemented cache is of fixed size. Unlike hardware caches where you have only a set of blocks per index and a block having a specific index can only go to its set (results in conflict misses), we have implemented a more flexible version of cache wherein each index (username/the first index) can have as many blocks as they want till all the blocks across different index sets sum to the cache size after which a victim is chosen for every subsequent insertion. This way we can optimally utilize memory by allowing multiple blocks per high traffic user and few blocks for low traffic users.

There are four functions that can operate on the cache. Insert function inserts a new block to both cache and database. If there is no vacancy at the time of insertion, then a victim is evicted according to one of the many replacement policies implemented in this work. Update function modifies the cache block if it is present else brings it from the database and updates it. Whenever an update happens, the dirty bit is set to 1 so that when that block is replaced, we would have to write it to the database as it is ahead of the same copy in the database. Whenever a cache block is deleted, the corresponding entry is also deleted from the database. Apart from all these, there is a search functionality to lookup the cache for a block and if not present then bring that one from the database if it is present there.

## **Database**

It has functionalities similar to that of cache with small modifications like no eviction as it is a permanent storage. This is the last stop of any food management data. It is hosted on a remote server and can handle multiple simultaneous requests from different clients like cache modules. Sockets are created for process to process communication between different processes on different machines.

## **Sockets**

For connection through sockets, the server side creates a socket, assigns appropriate socket parameters, binds its socket with the network and listens for connections through the socket port. Meanwhile, the client side gets the server information for connection by using its IP address. It also creates its socket and then sends a connection request to the listening server who immediately accepts it and transfer of information follow after which sockets are deleted and connection terminates. We use session based connection (close only after transferring the entire data[here, cache clock] at once) instead of object based connection (transfers only a small amount of data per connection).

## **5) Evaluation and results**

At the time of this submission, we have implemented two replacement policies - LRU(Least Recently Used) and LFU (Least Frequently Used). LRU evicts the block that has the least time stamp in the hope that temporal locality holds and the recently accessed ones are accessed again. LFU evicts the block that was least accessed while being in cache. The idea supporting is that an old block which gets frequent accesses should not be removed. We are currently

working on a new policy that keeps LRU and LFU per attribute of the cache block. The admittance score of a cache block can then be calculated as  $\sum LFU_i (LRU_i) / \sum LFU_i$  where “i” refers to the  $i^{th}$  attribute of a cache block. The more the admittance score the lower the chances of eviction on replacement. The rationale behind this policy is that the cache block should be kept only if most of its attributes are getting some attention from the user. If not, then that block is just wasting important cache space by having a few frequently accessed attributes as compared to others in the cache. Also, the attribute weights can be used to coalesce or split a given cache block for high performance.

We manually created many sequences of 10 requests (uniformly as well as randomly sampled from the three request types: food, user and order) to evaluate the miss rates on each cache policy and also compute the access latency. We tried our cache architecture on multiple such inputs but we will report only a selected subset of it here. As we had to manually create data, we had to limit the length of sequence of requests to 10 and our cache size to 5 to see the impact of evictions and writebacks. We present the tabular and graphical results on these input sets below. No caching case is also reported for the sake of baseline comparison.

It can be observed that performance of LRU and LFU is nearly comparable on the dataset that we used and is far better than no caching case where every request used to go to the database. Since the LFU and LRU performances are similar, we can conclude that the data set for experimentation is well balanced with a wide variety of requests which does not let any one policy dominate over another. In a LRU policy setting, the more the MFU(Most Frequently Used) block stays in the cache, the closer it gets to eviction. This is in stark contrast to the case of MRU(Most Recently Used) block in a LFU policy setting where the more it stays the more it consolidates its position in the cache. The plot of access time v/s miss rate is monotonically increasing with respect to increase in miss rates which further vindicates that cache misses result in a higher latency as cache lies on latency critical path. An interesting observation is that some test sets have significantly lower latency as compared to others as they include requests that are not database dependent unlike writebacks, first time insertions, etc. where we have no choice but to visit the database for the purpose of consistency. Test results further prove that inserts generate more database requests while updates generally get fulfilled at cache level.

```

Cache Way Index: 0
Username: fooditems
LRU count: 1, LFU count: 1, Num blocks: 4

```

```

Food Item: item1000 0 1 1 0
Food Item: item1003 99 4 1 0
Food Item: item1006 198 7 1 0
Food Item: item1009 297 10 1 0

```

```

Cache Way Index: 1
Username: user8
LRU count: 9, LFU count: 1, Num blocks: 1

```

```

Order Item: 208 9 1 23 0

```

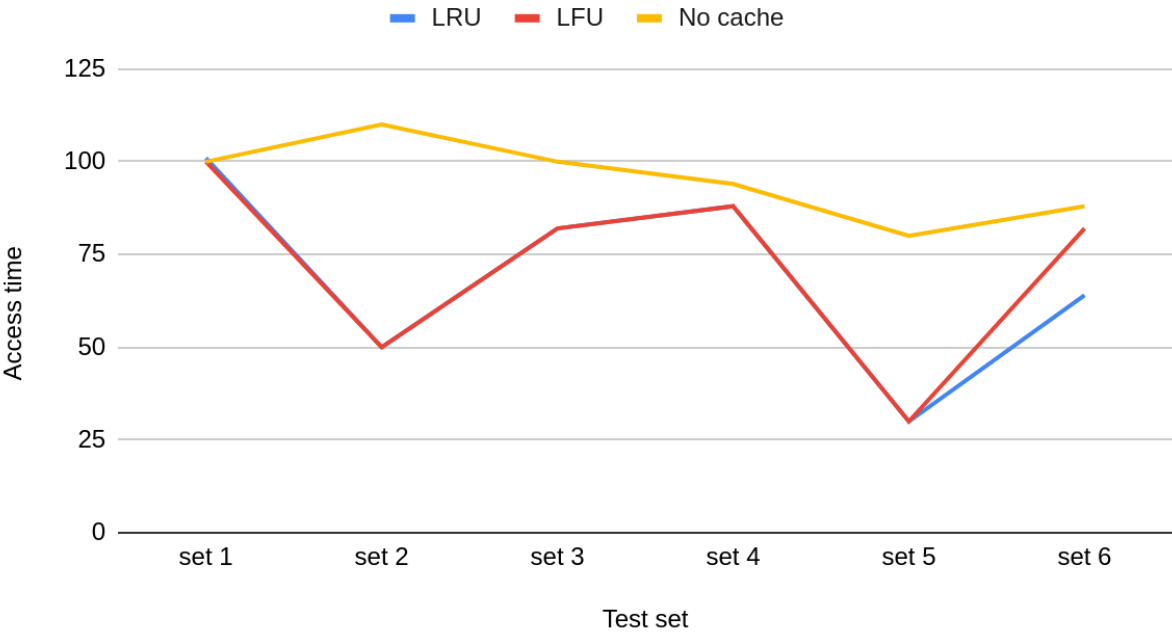
Snapshot of the cache at the end of Test set 1

Test set	Requests Included
Set 1	10 inserts
Set 2	5 inserts + 5 updates
Set 3	6 updates + 2 inserts + 2 deletes
Set 4	5 inserts + 5 deletes
Set 5	3 inserts + 5 updates + 2 searches
Set 6	6 inserts + 2 updates + 1 delete + 1 search

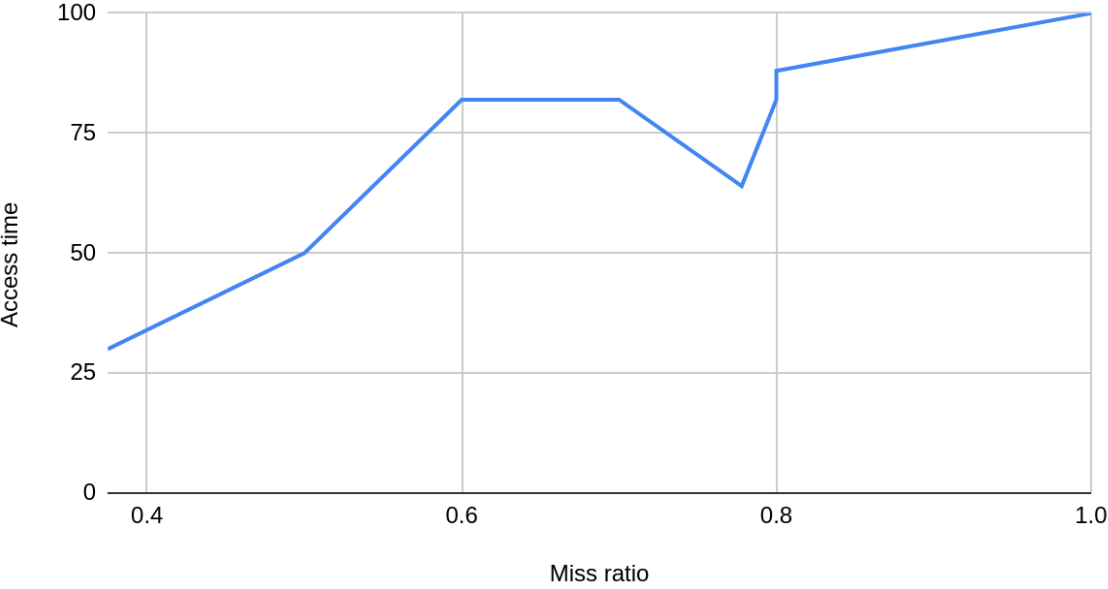
	LRU		LFU		No cache
Test set	Access time(s)	Miss ratio	Access time(s)	Miss ratio	Access time(s)
set 1	101	1	100	1	100
set 2	50	0.5	50	0.5	110
set 3	82	0.7	82	0.6	100
set 4	88	0.8	88	0.8	94
set 5	30	0.375	30	0.375	80
set 6	64	0.778	82	0.8	88

Average	69.16666667		72		95.33333333
---------	-------------	--	----	--	-------------

Access time comparison of different cache policies



Access time vs. Miss ratio





## 6) Conclusion and further improvements

- Even though we used a very small cache size for the sake of simplicity, the effect of cache is evident from the above results. The average access time has reduced significantly. With an even bigger size of the cache, the miss ratio would be even lower and the access time even lower.
  - Initially, we thought of hosting the database on AWS using HDFS. However, we did not receive free AWS credits when we created an account. So, we dropped the idea and implemented the database on openlab.
  - There were no datasets available to test the working of the cache. Therefore, we had to create custom datasets covering all the corner cases manually to make the cache more robust. Because of this manual work, we could not create large test sequences. In future, we would try to automate the dataset creation so that we can experiment with larger sequences and cache sizes.
  - We also faced issues with synchronizing requests through sockets. The problem was that whenever the sender was series of consecutive sends (for ex, to send different elements of the cache entry to the database), the sender sent data at a rate higher than what the receiver was capable of reading from the buffer. This led to the mixing of data from different sends in the buffer and corrupted the system. To resolve this, we forced the sender to wait for some time after sending by using sleep().
- 
- ❖ In addition to limiting the total number of entries in the cache, we can also limit the number of cache entries per user. This way, we can ensure that no single user's blocks occupy all the entries in the cache.
  - ❖ Apart from the LRU and LFU policies implemented, a mixture of both the policies can also be created. One idea we thought was to sample the policy based on the occupancy of the size ie., if the cache capacity is 50, then we would LRU for eviction for occupancies < 25 and LFU for occupancies > 25.