Memcached:

What it is:

Free & open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load. Memcached is a developer tool, not a "code accelerator", nor is it database middleware.

Memcached is an in-memory key-value store for small arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.

Memcached is simple yet powerful. Its simple design promotes quick deployment, ease of development, and solves many problems facing large data caches. Its API is available for most popular languages. In the ML world, it is compatible with python.

What is it Made Up Of?

- Client software, which is given a list of available memcached servers.
- A client-based hashing algorithm, which chooses a server based on the "key".
- Server software, which stores values with their keys into an internal hash table.
- LRU, which determines when to throw out old data (if out of memory), or reuse memory.

Strength and limitations:

Some strengths of Memcached include its high performance, enabling it to handle numerous requests per second and deliver low latency. Additionally, Memcached is scalable through horizontal scaling, allowing it to handle more traffic by adding more nodes to the cluster. Memcached is also simple to use and integrate with applications, and it supports various protocols, including text and binary protocols. Furthermore, Memcached provides flexible data storage by allowing data to be stored in different formats such as strings, hashes, and lists.

However, Memcached has several limitations. For instance, it does not offer built-in support for data consistency, which may result in inconsistent data across multiple nodes in the cluster. Furthermore, Memcached only stores data in memory, which means that there is no disk-based persistence, and data is lost in case of a restart. Additionally, Memcached has limited query capabilities and does not support complex queries or data aggregation, which may require applications to perform these tasks themselves. Memcached also lacks built-in security features, such as authentication or encryption, which may be a concern in some environments. Lastly, Memcached has a limit on the size of data that can be stored in memory, which may limit its use in certain applications.

Demonstration - Movie Streaming Scenario:

- Why it is useful in production in Movie Streaming Scenario:

With all the information above, now, we want to talk about why we use memcache in our system.

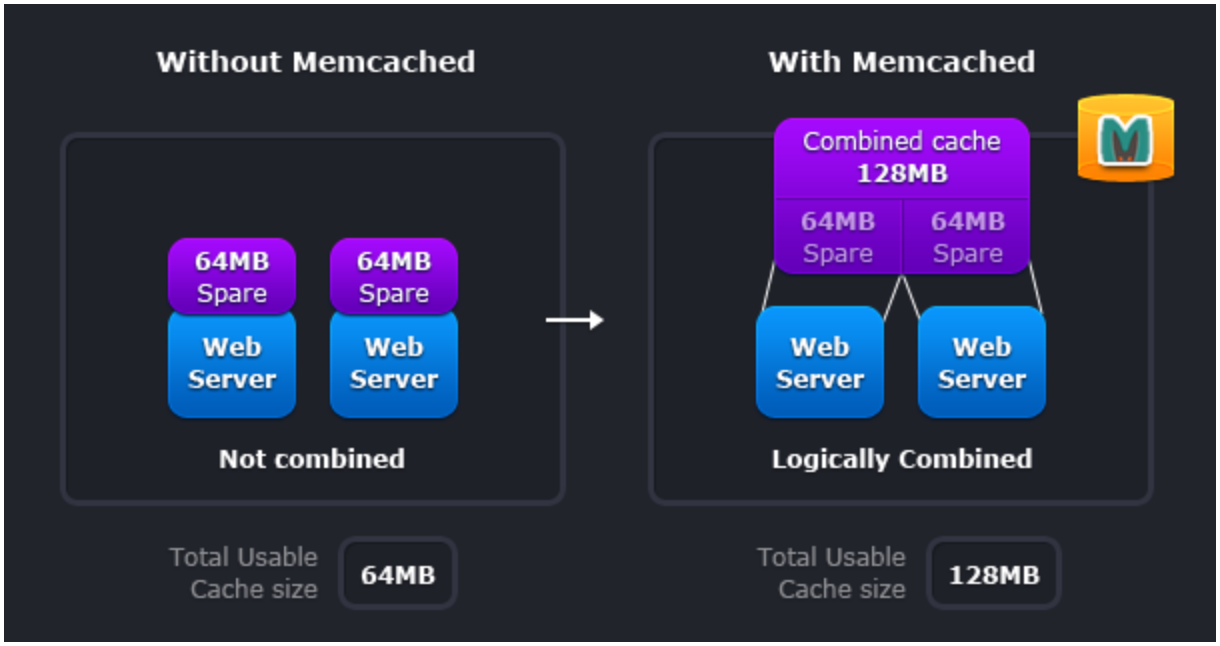
Currently:

Our online evaluation is currently based on using dictionaries built in the python library. In Python, dictionaries are typically stored in memory (RAM) rather than on disk. When you create a dictionary in Python, it is allocated in the heap memory of the process that creates it. This means that the dictionary is stored in the RAM of the server while the program is running. It's worth noting that if we want to persist the contents of a Python dictionary to disk, we can use techniques such as serialization to write the dictionary to a file or a database. But by default, Python dictionaries are stored in memory and are only available while the program is running. When the program exits or the server is rebooted, any data stored in memory is lost unless it has been written to disk or a database.

With Memcache:

How we can leverage memcache, there are two benefits we can bring our system:

1. We are using a larger potential pool of key-value pairs. In our online evaluation, we are using userId + score, where score is the number of times they listened to our recommendation for the movie. These key-value pairs could be large sometimes, memcache all of the servers are looking into the same virtual pool of memory. This means that a given item is always stored and always retrieved from the same location in the web cluster. In other words, we have a larger pool of RAM to spare if you want to get more servers. See example graph below:



2. We have failed safe backup. Since we are storing these dict or action process map into RAM right now, if the program stops for any reason, we will not have a safe backup. For memcache, we can easily allocate a back up cache where it will still have migrated data from the previous cache, so this prevents us from losing all data in one batch during online evaluation.
3. If our data ever grows large, to the point, we have to start writing them into disk, then memcache can store most frequent use userId and their score, so this could largely save our operation duration and we don't have to visit db as frequently as before. This is good for overall performance of a large data system.

Example implementation:

Here is github PR to demonstrate how we can use memcache in our team repo. To begin, you need to download memcache following these three command:

```
Python
- brew install memcached
- pip install pymemcache
```

Then we can use cache after we imported as follows : https://github.com/danyatingshen/Group3-Movie-Recommandation-System/pull/29/files and we can also set up the FALLback client with `ignore_exc=True.` In this example, we store key-value pairs as userId and result, where the result is the number of times the user watched the recommended movie. The Implementation very like how we would use dictionary but with more efficient and safe infrastructure.