medium.datadriveninvestor.com

All things caching- use cases, benefits, strategies, choosing a caching technology, exploring some popular products

Kousik Nath

38-48 minutes

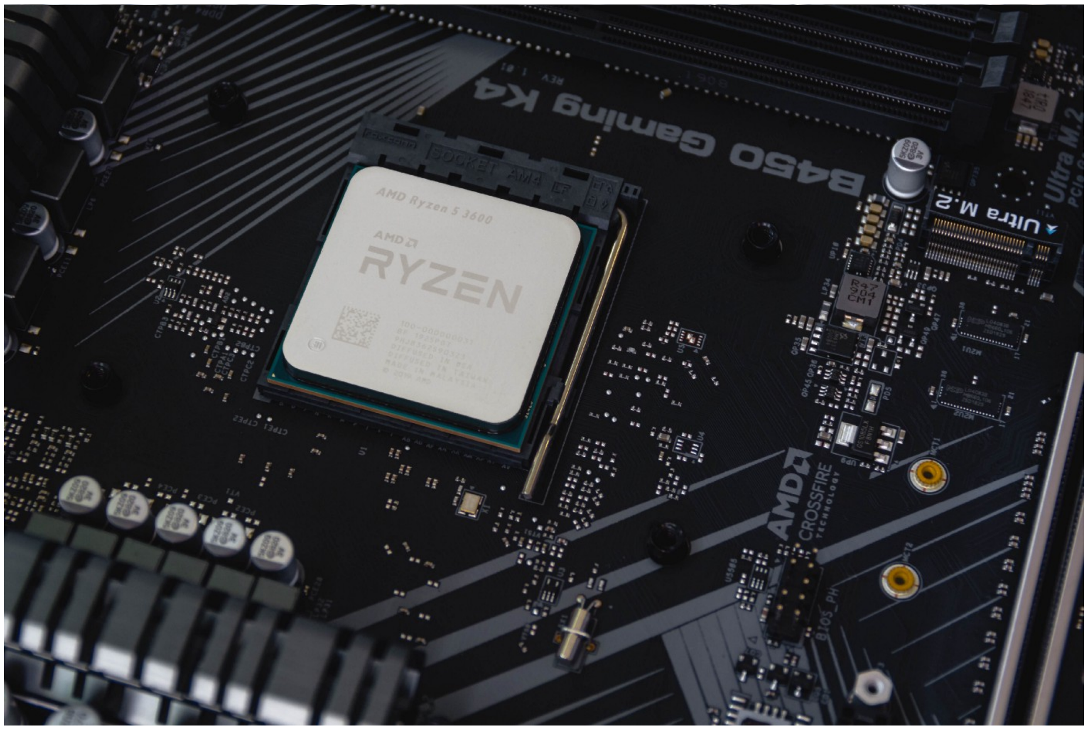


Image by [Christian Wiediger](#) on [unsplash.com](#)

Almost everyone is familiar with caching after all this technology is so ubiquitous nowadays, starting from CPU to browser to any web app - all software rely on caching to a certain extent to provide blazing fast response. A latency of few milliseconds can cause billion dollar loss so sub-millisecond response is becoming an everyday need. There are a vast number of caching solutions available in the market. But that does not mean using any technology solves your problem. So in this post, I try to explain different parameters that help you decide a technology, also we discuss features & some real-world use cases of already available solutions in the market so that before using them you know their common use cases.

Caching Prerequisites

Before even deciding on the caching layer, you need to ask yourself the following questions:

- Which business use-cases in your system require high throughput, fast response or low latency?
- Are you fine with data inconsistency if you use the cache?
- What kind of data do you want to store? Objects, static data, simple key-value pairs or in-memory data structures?
- Do you need to maintain the cache for transactional / master data?
- Do you need in-process cache or shared cache in a single node or distributed cache for n number of nodes?
- Do you need an open-source, commercial, or framework-provided cache solution?
- If you use distributed cache, what about performance, reliability, scalability, and availability?

Caching Benefits

> Effective caching aids both content consumers and content providers. Some of the benefits that caching brings to content

> delivery are:
>
> **Decreased network costs:** Content can be cached at various points in the network path between the content consumer and content origin. When the content is cached closer to the consumer, requests will not cause much additional network activity beyond the cache.
>
> **Improved responsiveness:** Caching enables content to be retrieved faster because an entire network round trip is not necessary. Caches maintained close to the user, like the browser cache, can make this retrieval nearly instantaneous.
>
> **Increased performance on the same hardware:** For the server where the content originated, more performance can be squeezed from the same hardware by allowing aggressive caching. The content owner can leverage the powerful servers along the delivery path to take the brunt of certain content loads.
>
> **Availability of content during network interruptions:** With certain policies, caching can be used to serve content to end users even when it may be unavailable for short periods of time from the origin servers.

General Cache Use Cases

Let’s see some popular use case for caching probably your use case might overlap with one of them:

In-memory data lookup: If you have a mobile / web app front end you might want to cache some information like user profile, some historical / static data, or some api response according to your use cases. Caching will help in storing such data. Also when you create a dynamic programming solution for some problem, 2-dimensional array or hash maps work as a caching.

RDBMS Speedup: Relational databases are slow when it comes to working with millions of rows. Unnecessary data or old data, high volume data can make their index slower, irrespective of whether you do sharding or partitioning, a single node can always experience delay & latency in query response when the database is full in its capacity. In such scenarios, probably many ‘SELECT’ queries or read queries can be cached externally at least for some small time window. Relational databases use their own caching as well, but for better performance external caching can have much more capacity than internal caching. It’s one of the most popular use cases of caching.

Manage Spike in web/mobile apps: Often popular web/mobile apps experience heavy spike when it attracts a lot of user traction. Many of such calls may end up being read queries in database, external web service calls, some can be computed data like on the fly computation of previous payments, some can be non-critical dynamic data like no of followers for a user, no of re-tweets, viewers counts etc. Caching can be used to serve such data.

Session Store: Active web sessions are very frequently accessed data — whether you want to do api authentication or you want to store recent cart information in an e-commerce app, the cache can serve session well.

Token Caching: API Tokens can be cached in memory to deliver high-performance user authentication and validation.

Gaming: Player profile & leader board are 2 very frequent screens viewed by gamers specially in online multiplayer games. So with millions of gamers, it becomes extremely important to update & fetch such data very fast. Caching fits for this use case as well.

Web Page Caching: In order make mobile/web app lightweight & flexible UI, you can create dynamic web pages in the server & serve it through api along with appropriate data. So if you have millions of users, you can serve such on the fly created full/fragmented web pages from the cache for a certain time period.

Global Id or Counter generation: When you have variable number of say relational database instances across nodes & you want to generate an auto incrementing primary key for them, or when you want to assign unique id to your users, you can use caching to fetch & update such data at scale.

Fast Access To Any Suitable Data: Many times we think cache is only used to store frequently accessed data for read purpose. Although this is mostly correct, this behaviour can vary according to use cases. Cache can be used to store less frequent data also if you really need fast access to that data. We use cache to access the data very fast, so storing most frequent / least frequent data is just a matter of use case.

Caching Data Access Strategies

Application/system design depends a lot on the data access strategies used. They dictate the relationship between data source & the caching system. So it's very important to choose proper data access strategy. Before choosing any strategy, analyze the access pattern of the data & try to fit your application's suitability with any of the following.

Read Through / Lazy Loading: Load data into the cache only when necessary. If application needs data for some key **x**, search in the cache first. If data is present, return the data, otherwise, retrieve the data from data source, put it into the cache & then return.

Advantages:

1. It does not load or hold all the data together, it's on demand. Suitable for cases when you know that your application might not need to cache all data from data source in a particular category.

2. If there are multiple such cache nodes & a node fails, it does not harm the application although in such situation, the application faces increased latency. As new cache node comes up online, more & more request flows through it & it keeps populating required data with every cache miss.

Disadvantages:

1. For cache miss, there are 3 network round trips. Check in the cache, retrieve from database, pour the data into the cache. So cache causes noticeable delay in the response.
2. Stale data might become an issue. If data changes in the database & the cache key is not expired yet, it will throw stale data to the application.

Write Through: While inserting or updating data in the database, upsert the data in the cache as well. So both of these operations should occur in a single transaction otherwise data staleness will be there.

Advantages:

1. No stale data. It addresses the staleness issue of Read Through cache.
2. Suitable for read heavy systems which can't much tolerate staleness.

Disadvantages:

1. It's a write penalty system. Every write operation does 2 network operations — write data to data source, then write to cache.
2. *Cache churn:* If most of the data is never read, cache will unnecessarily host useless data. This can be controlled by using TTL or expiry.
3. In order to maintain the consistency between cache & data source, while writing a data, if any of your cache node goes missing, the

write operation fails altogether.

Write Behind Caching: In this strategy, the application writes data directly to the caching system. Then after a certain configured interval, the written data is asynchronously synced to the underlying data source. So here the caching service has to maintain a queue of ‘write’ operations so that they can be synced in order of insertion.

Advantages:

1. Since the application writes only to the caching service, it does not need to wait till data is written to the underlying data source. Read and write both happens at the caching side. Thus it improves performance.
2. The application is insulated from database failure. If database fails, queued items can be retried or re-queued.
3. Suitable for high read & write throughput system.

Disadvantages:

1. Eventual consistency between database & caching system. So any direct operation on database or joining operation may use stale data.

Application design constraints with write-behind strategy:

1. Since in this strategy cache is written first & then database — they are not written in a transaction, if cached items can not be written to the database, some rollback process must be in-place to maintain consistency over a time window.
2. Write-behind caching may allow out of order database updates, so database have to be able to relax foreign key constraints. Also if the database is a shared database, other apps may also use it, hence no way to know whether write-behind cache updates will conflict with other external updates. This has to be handled manually or heuristically.

Refresh Ahead Caching: It’s a technique in which the cached data is refreshed before it gets expired. Oracle coherence uses this technique.

> The refresh-ahead time is expressed as a percentage of the entry’s expiration time. For example, assume that the expiration time for entries in the cache is set to 60 seconds and the refresh-ahead factor is set to 0.5. If the cached object is accessed after 60 seconds, Coherence will perform a *synchronous* read from the cache store to refresh its value. However, if a request is performed for an entry that is more than 30 but less than 60 seconds old, the current value in the cache is returned and Coherence schedules an *asynchronous* reload from the cache store.

So what refresh ahead caching does is it essentially refreshes the cache at a configured interval just before the next possible cache access although it might take some time due to network latency to refresh the data & meanwhile few thousand read operation already might have happened in a very highly read heavy system in just a duration of few milliseconds.

Advantages:

1. It’s useful when large number of users are using the same cache keys. Since the data is refreshed periodically & frequently, staleness of data is not a permanent problem.
2. Reduced latency than other technique like Read Through cache.

Disadvantages:

1. Probably a little hard to implement since cache service takes extra pressure to refresh all the keys as and when they are accessed. But in a read heavy environment, it’s worth it.

The point of discussing the strategies is that — while designing the system, you know what kind of system you are designing — whether it’s a read heavy, write heavy or mix of both at a very high scale. Different systems will have different requirements, so it's very hard to come up with some solid use cases & full-proof strategy. You can choose one or a combination of more strategies as described above according to your use case.

Eviction Policy

> An eviction policy enables a cache to ensure that the size of the cache doesn't exceed the maximum limit. To achieve this, existing elements are removed from a cache depending on the eviction policy, but it can be customized as per application requirements.

A caching solution may provide different eviction policies, but before choosing a caching technology, it's good to know what kind of eviction policy your application might need. It might happen that your application needs different eviction policies for different use cases, that's perfectly fine, but knowing it will help you better deciding on caching technology.

- **Least Recently Used (LRU):** One of the most used strategies is LRU. In most caching use cases, applications access the same data again & again. Say in any Google search engine, when you search for something, you will get the same results again & again at least for some time window. When you search flights or bus or train, you get the same routes unless & until some route gets deactivated or fully reserved. In such use cases, the cached data that is not used very recently or sort of cold data can be safely evicted.

> **Advantages:**
> 1. Is nearest to the most optimal algorithm
> 2. Selects and removes elements that are not used recently
> **Disadvantages:**
> 1. Only mild success rate since often it is more important how often an element was accessed than when it was last accessed

- **Least Frequently Used (LFU):**

> Your mobile keyboard uses `LFU`. When you type some letters you can see few suggested words at the top of the keyboard matching with the letters you have typed. At the begining when the keyboard app's `cache` is empty, it may show you these 4 words (Lets assume, you typed letters "STA". Suggested words may pop like ex. start, stand, statue, staff). The idea here is that, based on the words you use, it will ignore the LRU word in the suggestions after a certain time. You may not see the word "staff" in the suggesions later on if you haven't used it.
>
> If you have a case where you know that the data is pretty repetative, surely go for `LFU` to avoid `cache` miss. It seems that these both are independent quite clearly and have isolative significance. It depends on the use case of where you want to use any of these.
>
> **Advantages:**
> 1. Takes age of the element into account
> 2. Takes reference frequency of the element into account
> 3. Works better under high load when quickly a lot of elements is requested (less false eviction)
>
> **Disadvantages:**
> 1. A frequently accessed element will only be evicted after lots of misses
> 2. More important to have invalidation on elements that can change

- **Most Recently Used (MRU):** Let’s consider Tinder. Tinder personalises matching profiles for you and say it buffers those result in a cache or a high performance cache. So you can assume that some space for every user is allocated to queue corresponding personalised results. When you see Tinder’s recommendation page, the moment you right or left swipe, you don’t need that recommendation view any more. So in this use case, Tinder can remove the recommendation from that user’s queue & free up space in memory. This strategy removes most recently used items as they are not required at least in the near future.

- **First In, First Out (FIFO):** It's more of like MRU but it follows strict ordering of inserted data items. MRU does not honour insertion order.

  In some use cases, you might need to apply a combination of eviction policies such as LRU + LFU to decide on which data to evict. That's your use case dependent, so try to choose such technologies which are inline with the eviction policies you thought of.

Data Type Wise Cache

Different caching technology serves well to different data types due to their internal design. So choosing the correct technology also depends on what type of data they can efficiently store.

Object Store: Suitable to store unmodifiable objects like HTTP Response Object, database result set, rendered html page etc. Example: Memcached.

Simple Key Value Store: Storing simple string key to string value is almost supported by any cache. Example: Redis, Memcached, Hazelcast, Couchbase etc.

Native Data Structure Cache: If your use case supports storing in & retrieving data from natively supported data structures, then Redis & Aerospike are good choice.

In-Memory Caching: Suitable to store any key value or objects directly accessible through run time memory in the same node. Example: HashMap, Guava Cache etc, Hibernate & MySQL query caching.

Static File Cache: Suitable to cache image, files, static files like — css or javascript files. Example: Akamai CDN, Memcached to a certain extent.

Caching Strategy

Single Node (In-Process) Caching:

It’s a caching strategy for non distributed systems. Applications instantiate & manage their own or 3rd party cache objects. Both application & cache are in the same memory space.

This type of cache is used for caching database entities but can also be used as some kind of an object pool, for instance pooling most recently used network connections to be reused at a later point.

Advantages: Locally available data, so highest speed, easy to maintain.

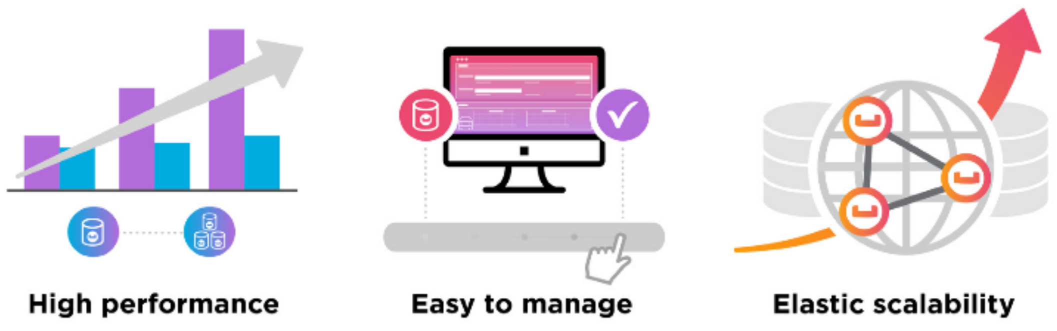
Disadvantages: High memory consumption in a single node, cache shares memory with the application. If multiple application relies on same set of data, then there might be a problem of data duplication.

Usecase: Choose this strategy when you are making standalone applications like mobile apps, or web front end apps where you want to temporarily cache website data that you got from back end api or other stuffs like images, css, java script contents. This strategy is also useful when you want to share objects that probably you created from an api response across different methods in different classes in your backend application.

Distributed Caching:

When we talk about internet scale web applications, we actually talk about millions of requests per minute, petabytes or terabytes of data in the context. So a single dedicated powerful machine or two will not be able to handle such humongous scale. We need several machines to handle such scenarios. Multiple machines from tens to hundreds make a cluster & at a very high scale we need multiple such clusters.

Key requirements for an effective distributed caching solution: Following are most important requirements for distributed caching.



Courtesy: Couchbase

1. **Performance:** Cache should be able to constantly sustain the throughput requirements in terms of read or write queries from application. So the more it is able to take advantage of resources like — RAM, SSD or Flash, CPU etc, the better it is at producing output.
2. **Scalability:** Caching system has to be able to maintain steady performance even if number of operations, requests, users & amount of data flow increases. It must be able to scale linearly without any adverse impact. So elastically growing up or down is an important characteristic.
3. **Availability:** High availability is the utmost requirement in today's systems. It's fine to get stale data (depending on use case) but unavailable systems are not desired. Whether there is a planned or unplanned outage, or a portion of system is crashed or due to natural calamity some data centre is non-operational, cache has to be available all the time.
4. **Manageability:** Easy deployment, monitoring, useful dashboard, real-time matrices make every developer & SRE's life simple.

5. **Simplicity:** All other things equal, simplicity is always better. Adding a cache to your deployment should not introduce unnecessary complexity or make more work for developers.
6. **Affordability:** Cost is always a consideration with any IT decision, both upfront implementation as well as ongoing costs. Your evaluation should consider total cost of ownership, including license fees as well as hardware, services, maintenance, and support.

So depending on your use cases, you can decide what kind of caching to use. In most of the cases enterprise systems or highly established startups which cater to millions of customers go for distributed caching using multiple clusters. Small & middle scale companies also use multiple nodes for caching but they don't always need enterprise grade solutions. As mentioned, in-process cache is usable only in standalone apps or local caching in users' devices.

Let's explore some popular real life caching technologies which might solve our daily caching needs.

Real Life Caching Solutions

Memcached

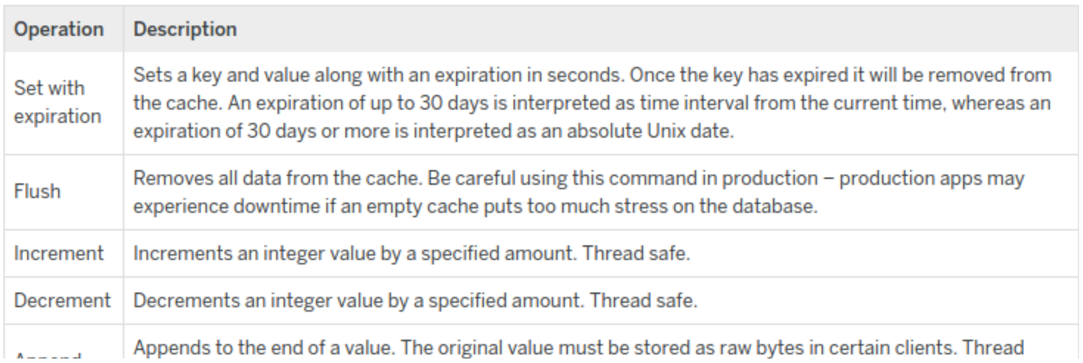
> *Free & open source, high-performance, distributed memory object caching system*, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load. Memcached is an in-memory key-value store for small arbitrary data (strings, objects).

Features:

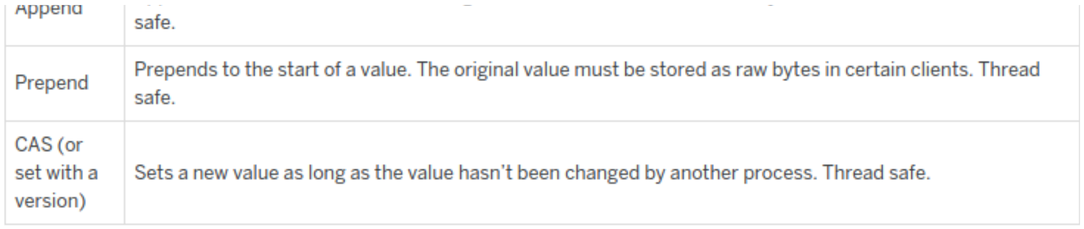
1. Memcached is engineered as a high performance caching solution that can satisfy throughput of very large, internet scale apps.
2. It's very easy to install & deploy as by design, Memcached is a bare bone cache.

3. Very low cost solution, licensed under the revised BSD license.
4. Simple key value store. Memcached does not understand what an application is saving — it can store String & Object values, keys are always of type String. It allows storing object as value in serialized form. So before saving any object, you have to serialize it, after retrieval you have to deserialize it accordingly.
5. In a distributed settings, Memcached nodes don't talk to each other, there is no synchronization, no replication. So essentially it embraces simple design where client has to choose the node where it has to read / write a particular data.
6. It's multi-threaded. So it can take advantage of multiple CPU cores.
7. All memcahed commands are fast & lock free as possible. So query speed is near deterministic for all cases.
8. Client keeps the logic to map cache keys to nodes in case multiple nodes are available.
9. Apart from Get, Set & Delete operation, memcached offers other features as well like — key expiration (TTL), completely flushing the database, light weight counters which support increment & decrement operation, a native list data structure which supports append & prepend operations of items, thread safe CAS (Compare & Swap) backed set operation.
10. Cache invalidation is easy as client keeps track of which key is going to which node, it can just delete that key from that node.

    Following is a summary of all operation:



Operation	Description
Set with expiration	Sets a key and value along with an expiration in seconds. Once the key has expired it will be removed from the cache. An expiration of up to 30 days is interpreted as time interval from the current time, whereas an expiration of 30 days or more is interpreted as an absolute Unix date.
Flush	Removes all data from the cache. Be careful using this command in production – production apps may experience downtime if an empty cache puts too much stress on the database.
Increment	Increments an integer value by a specified amount. Thread safe.
Decrement	Decrements an integer value by a specified amount. Thread safe.
[illegible]	Appends to the end of a value. The original value must be stored as raw bytes in certain clients. Thread



Append	safe.
Prepend	Prepends to the start of a value. The original value must be stored as raw bytes in certain clients. Thread safe.
CAS (or set with a version)	Sets a new value as long as the value hasn't been changed by another process. Thread safe.

Courtesy: Heroku

Memory Management Technique: Memcached only supports LRU memory management. So analyze your use case carefully before choosing this solution.

Suitable Usecases: Store simple string key / value pairs. Store database resultset object, HTTP api response, or serializable in-memory objects, JSON / XML document as value with a string key, results of page rendering etc.

Limitations:

1. Since there is no persistence, with every crash or restart, you need to warm up or refill data every time.
2. If you want to store large objects or data sets, serialized representation of data may take more space & can make the memory fragmented.
3. Memcached limits size of data upto 1 MB per key.
4. Avoid read-modify-write operation use cases. Since you need to serialize / deserialize objects while inserting / retrieving data, update operations seem to be very costly. Try to store immutable objects with expiration time as much as possible.
5. Memcached is not good for enterprise use cases. It does not offer many features like automatic elastic cluster management, sophisticated high availability, auto failover, load re-balancing, cross data centre replication etc.
6. It lacks advance monitoring & tooling ecosystem for administrative use cases. If you face any issue, either you have to depend on your

resource or the Memcached community, there is no commercial entity supporting it.

LinkedIn started using Memcached in 2010, but by 2012, they faced many issues & forced to get rid of it. Following snippets are taken from LinkedIn's engineering blog, it will give a good idea why Memcached is not a great enterprise fit:

> **The Memcached days**
>
> Memcached was first introduced to LinkedIn in the early 2010s as a fast, in-memory, distributed caching solution when we needed to scale our source-of-truth data stores to handle increased traffic. It worked well for what it provided:
>
> Single-digit millisecond GETs and SETs for applications that needed caching in front of their source-of-truth data stores.
>
> Provisioning a cluster was simple, and the process of getting started was very fast.
>
> However, as the number of Memcached clusters grew with the number of applications that started using it, we quickly ran into operability issues, some of the main ones being:
>
> **Lack of persistence.** Memcached was an in-memory store. Although fast response times was Memcached's greatest strength, it was also brutal when we needed to restart the Memcached process for maintenance reasons, as we would also lose the entire contents of the cache.
>
> **Breaking the hash ring.** Resizing clusters (i.e. expanding the cluster with more nodes) was impossible without breaking the hash ring and invalidating portions of the cache. Consistent hashing algorithms helped, but didn't solve the issue completely.
>
> **Replacing hosts.** Our hashing algorithm was based on the hostname of the nodes in the cluster, so any time we would replace a host with a different hostname, it would not only invalidate the

> cache on the replaced node, but also would disrupt the hash ring.
>
> **Lack of cache-copying functionality.** Say we wanted to build out a new datacenter and we wanted to copy the contents of the cache. This was not simple. We ended up building some tooling around brute-forcing keys to populate the new cache, but this wasn't ideal nor elegant.
>
> You can see why Memcached was so easily adopted at LinkedIn (provided quick and easy wins), but at the same time why it was so annoying to work with at scale.

Notable Memcached Users: YouTube, Reddit, Craigslist, Zynga, Facebook, Twitter, Tumblr, Wikipedia.

Redis

One of the most popular cache, in memory key value store ever known in the market.

> Redis is often referred as a *data structures* server. What this means is that Redis provides access to mutable data structures via a set of commands, which are sent using a *server-client* model with TCP sockets and a simple protocol. So different processes can query and modify the same data structures in a shared way.

Features:

1. Redis supports native mutable data structures namely — list, set, sorted set, string, hash. It also supports range queries, bitmap, hyperloglogs, geo-spatial indexes with radius queries.
2. Redis stores all data in memory, essentially redis is a big in-memory dictionary. So it's very fast. It can run commands in pipeline.
3. Data can be asynchronously saved on disk after a configured interval or a specific number of operations.
4. Redis is typically known as single threaded. It means the

application logic that directly serves the clients is a single thread only. While syncing data on disk, redis spawns background thread which does not directly deal with clients.

5. Redis supports out of the box master slave replication. It's just a configuration settings & replication is up & running.
6. Redis supports transaction. All commands in a transaction are serialized & they run sequentially. As usual, redis transactions also guarantee either all commands will pass or none are processed.
7. Redis keys are TTL or expiration time supported.
8. Redis has out of the box support for pub-sub mechanism. It has commands that enable pub-sub.
9. Automatic failover is supported by Redis Sentinel.
10. Redis supports server side Lua scripting. So a batch of commands can run without much hassle of communication between server & client.
11. Redis is portable, works on almost all varities of Linux, Windows, Mac etc.
12. Support for size of value upto 512 MB per key.
13. Also Redis enterprise edition supports a lot more features.

Memory Management Technique: Redis supports following techniques:

- **allkeys-lru**: Evicts the least recently used keys out of all keys.
- **allkeys-random**: Randomly evicts keys out of all keys.
- **volatile-lru**: Evicts the least recently used keys out of all keys with an "expire" field set.
- **volatile-ttl**: Evicts the shortest time to live keys (out of all keys with an "expire" field set).

- **volatile-random**: Evicts keys with an “expire” field set.
- **no-eviction**: Redis will not evict any keys and no writes will be possible until more memory is freed.

Suitable Usecases: Redis has many many lucrative use cases:

1. Redis hash can be used in place of relational tables if you can model your data accordingly & your use cases don’t require any transactional guarantee.
2. Redis pub-sub can be used to broad cast messages to multiple subscribers.
3. Redis list can be used as queue of messages. Celery — a distributed task processing system leverages redis data structures to manage tasks.
4. Session store is a very popular use case for redis. Persistent ability of redis makes it suitable for such case.
5. Redis sorted sets can be used to manage leader boards in online gaming.
6. Redis can store, increment, decrement integers. It can be used to generate global id for any use cases.

More use cases can be found here & here.

Limitations:

1. Redis does not support secondary index.
2. Redis offers querying / scanning of data based on regular expression search on key names. So before choosing to use Redis data structures like hash, sorted sets etc, try to think in terms of how your applications fits into Redis & what is your access pattern of data in case you use these data structures. For simple key value use cases, it’s chill, you don’t need to think a lot.

Notable Redis Users: Twitter, GitHub, Weibo, Pinterest, Snapchat, Craigslist, Digg, StackOverflow, Flickr

Aerospike

An extremely fast, open source, NoSQL key value store. One of the important selling points of Aerospike is it has a hybrid memory model — it means if your server is maxed out on RAM, SSD (if the server uses) can be used as alternative. Built-in clustering capability, less operational & maintenance overhead, high availability make Aerospike a great choice for insanely scalable systems.

Features:

1. Aerospike is optimized for flash drives like — SSD, PCIs, NVMe. Aerospike uses flash drives to scale vertically. SSD enables enormous vertical scaleup at a 5x lower total cost of ownership than pure RAM. Generally the IOPS (Input Output Per Second) keeps on increasing for devices. SSDs can store an order of magnitude more data per node than DRAM. NVMe drives can now perform 100K IOPS per drive. Aerospike exploits these capabilities, so it can run upto millions of operations per second with a sub millisecond latency all the time.
2. It supports key value storage, batch queries, scans, secondary index queries & aggregations etc.
3. It supports running in a cluster environment. It has the capability to automatically manage the cluster, node addition & departure is identified automatically & data distribution happens accordingly.
4. It is schemaless.
5. It supports TTL expiry for all records.
6. It supports strings, integers, blobs, lists, maps, and serialized objects.
7. Aerospike does cross data centre replication (XDCR) of data, it

continuously copies data among cluster nodes, thus it helps to create very highly available crash-proof system.

8. Aerospike performs large block write & small block read for higher performance.
9. It’s multi threaded.
10. Server side Lua scripting is supported. It makes server side batch operations faster.
11. AQL — Aerospike Query Language is supported to help developers query the key values in the data store.

Suitable Usecases:

1. *Ad-tech industry:* Various use cases like storing user profiles, users’ history, session data, counters for ad impressions fits well with Aerospike due to its key value store capability.

> When building any form of advertising or marketing application, you’ll need to store user profiles. Those profiles will often have recent user behaviour, segments loaded from an analytics system, partner cookies, and a variety of other data. Smaller sizes — like 1 KB to 10 KB — per profile is common. Besides pure profiles, you’ll need cookie matching, campaign budgets and status, and other front-end data.

2. Aerospike offers a data structure called LDT — Large Data Type. It can be used to store millions of items (say integer or string) per key. So if you want to store follower list of a celebrity mapped to the celebrity id, you can do that with LDT. It’s just a very simple & granular use case.

3. Aerospike can be used to store nested data i.e; a data structure under another data structure. So a native list can be contained in a map & so on.

4. If you want to cache rapidly changing dynamic data, Aerospike is

a good choice.

5. *Data Analytics:* Storing consumer behaviour like in financial industry which transactions a consumer is executing & identifying if it's fraudulent transaction, the system needs to write & retrieve consumers' data very fast so that the analysis can happen quickly by analyzing rules or concerned service. Aerospike fits well to accommodate such large number of high read & write operations.

6. *Recommendation Engine:*

> A recommendation engine uses innovative math, combined with domain-specific knowledge, to increase online engagement. If you're designing one, you'll need a data layer that's fast — to support multiple requests per recommendation — and flexible, since you'll need either more throughput or more data as your system evolves. You'll want one that supports high write throughputs when ETLing data from your data scientists, or if you are recording recent behaviors that your algorithms will use.
>
> Aerospike is an excellent database for recommendation engines. Key features are large lists (for efficiently recording behavior), optimized Flash support to handle datasets from terabytes to petabytes, queries and aggregations for real-time reporting, and strong support for languages such as Python and Go.

7. Can be used to very large data sets like geo-location hash. If you want to store distance & time duration data between 2 locations & you want to cache millions of such data points, Aerospike can help you out here.

Memory Management Technique: Can be found here.

Notable Aerospike Users: InMobi, Flipkart, Adobe, Snapdeal, AppsFlyer, PubMatric, Swiggy.

Hazelcast

Hazlecast is an in-memory data grid which is a clustered system, highly available & scalable. It’s very fast because it automatically distributes & replicates data across nodes. It’s a Java based caching system & fits quite well in the Java ecosystem although Hazlecast clients are available in other major languages as well.

> Hazelcast is a schema-less in-memory data store and is approximately 1,000 times faster than a RDBMS achieving query and update times measured in microseconds for high volumes of data.

Features:

1. Hazlecast does not have a master node. All nodes are like master. They maintain a metadata information called ‘partition table’. Partition table keeps the information of members detail, cluster health, backup information, re-partitioning, etc. Hazlecast clients also have access to this ‘partition table’, so they can query the associated node directly where the data lies.
2. Hazlecast distributes copies of data across all nodes in the cluster. So if a node goes down, the data is not lost.
3. With more & more data appearing, Hazlecast can grow & scale horizontally.
4. It’s multi-threaded. Hence all CPU cores can be potentially used.

More features here.

Suitable Usecases:

1. Hazlecast re-implements List, Map, Set, AtomicLong, AtomicReference, Count down lacth so that they can be used safely in a clustered / distributed environment when accessed from applications running across multiple nodes.
2. Hazlecast can be used to generate unique id for partitioned databases or other use cases.
3. It also implements several distributed computation like Scheduled

executor, executor service etc to be used in a distributed environment.

4.

> Hazelcast IMDG provides web session clustering. User sessions are maintained in the Hazelcast cluster, using multiple copies for redundancy. In the event of an application server failure, the load balancer redirects to a new application server which has access to the user session. The hand off to the new application server is seamless for the user. This provides the highest level of customer experience. Web session clustering use case is available for Tomcat and Jetty using native integration, and any application server using filters.

5. Hazlecast is preferred in Financial industry also. More here.

More usecases here as well.

Memory Management Technique: Hazlecast supports LRU, LFU or None of these.

Notable Hazelcast Users: Ola cabs (India), American Express, Credit Suisse, Hyperwallet Systems, PayPal, Atlassian, Apache Camel, Twilio, Vert.x etc.

Couchbase

Couchbase is an open source, enterprise grade NoSQL document store (can be used as either key value or document store) which is more than just a caching solution & offers features that make it less complex & adaptable by a lot of large number of enterprise players.

> Couchbase was architected to solve the core problems that are often related to caching but are really just standard performance and scalability issues. Additionally, Couchbase addresses challenges relating to high availability for applications as well as comprehensive durability for data with failover options.

Features:

1. Couchbase has a memory first architecture, the whole database can run in-memory which makes it suitable for caching scenarios. Couchbase is built for high performace, it provides sub-millisecond latency.

   It can also service data beyond the size of memory, using efficient data access methods that keep memory loaded with the latest updates.

2. Couchbase offers persistence also. So if a node crashes, data is not lost.

3. Elastic scalability is something where if you have a lot of data & you want to spread it across nodes or clusters, couchbase can do it without making you taking a lot of load.

4. Couchbase offers cross cluster & cross data centre replication (XCDR) to offer high availability.

5. Couchbase supports querying JSON data with its own query language called N1QL.

6. It also offers real time analytics querying, full text search, server side event processing.

7. It supports automatic application failover between clusters.

8. Couchbase offers auto sharding & it uses cluster aware clients. So when a new node is introduced or a node leaves, you don't need to do sharding, everything will happen behind the scene.

9. Enterprise support is available.

Suitable Usecases: Large enterprises which require caching + NoSQL use cases with ease of maintenance, high availability & resiliency, high throughput with auto fail over, auto sharding, easy monitoring & administrative support etc.

Limitations: Can be found here.

Some notable Users: PayPal, Intuit, Viber, Tesco, AT&T, Verizon, Ebay etc.

The Last Thing: Cache-As-A-Service

In large enterprise, distributed shared cache is used. So you can directly add dependency for the specific cache in your application in order to start using it. But that's not a very suitable method since it might happen that in near future, another fantastic caching solution comes into existence & your organization decides to use that. So to abstract the cahcing layer from future troubles, cache can be hidden behind a service. All different applications can communicate with the cache through api calls. It encapsulates the internal details of the cache and the caching team will be able to make any necessary changes required quickly without changing the exposed interface. But once you decide to take this path, you have to make sure that the caching service is up 24x7. So all challenges related to microservice architecture will be valid for this case, but nevertheless, the kind of abstraction & flexibility the service is supposed to offer is priceless for a large organization.

In this post we have seen different parameters while choosing & designing our own caching service. It's not only a big / popular technology name or your familiarity with a technology that decides what you choose, but it's more of our application use cases, data access pattern, type of objects we want to cache, desired duration of the data, eviction policy, infrastructure requirement, desirable persistence of data, scalability & clustering requirements, availability, volume of data etc that decides what the best technology to go for. It's upto the engineers how much consideration they want to have before they choose something — it's very normal behaviour that people choose a technology which they are already used to, but knowing all of these parameters for sure helps you to take a better design decision.

Please let me know if you can add any value to this post or I can improve anything by adding a comment below.

Reference:

[1] https://dzone.com/articles/introducing-amp-assimilating-caching-quick-read-fo

[2] https://www.itpro.co.uk/virtualisation/30271/our-5-minute-guide-to-distributed-caching

[3] https://docs.oracle.com/cd/E15357_01/coh.360/e15723/cache_rtwtwbra.htm#COHDG5181

[4] https://blogs.dropbox.com/tech/2012/10/caching-in-theory-and-practice/

[5] https://stackoverflow.com/questions/17759560/what-is-the-difference-between-lru-and-lfu

[6] https://stackoverflow.com/questions/44343510/in-which-case-lfu-is-better-than-lru

[7] https://devcenter.heroku.com/articles/advanced-memcache

[8] https://www.quora.com/What-use-cases-is-Redis-suitable-not-suitable-for-in-a-high-traffic-web-application-as-of-early-2016

[9] http://www.tothenew.com/blog/caching-what-why-and-how-with-hazelcast/

[10] https://blog.hazelcast.com/hazelcast-use-cases/

[11] https://hazelcast.com/why-hazelcast/imdg/

[12] https://www.aerospike.com/products/features/