CHAPTER 6

Caching

Caching is an essential ingredient of a scalable system. Caching makes the results of expensive queries and computations available for reuse by subsequent requests at low cost. By not having to reconstruct the cached results for every single request, the capacity of the system is increased, and it is hence able to scale to handle greater workloads.

Caches exist in many places in an application. The CPUs that run applications have multi-level, fast hardware caches to reduce relatively slow main memory accesses. Database engines can make use of main memory to cache the contents of the data store in memory so that in many cases queries do not have to touch relatively slow disks.

This chapter covers:

- application based caching, in which service business logic incorporates the caching and access of precomputed results
- Web based caching, which exploits mechanisms built into the HTTP protocol to enable caching of results within the infrastructure provided by the Internet.

Application Caching

Application caching is designed to improve request responsiveness by storing the results of queries and computations in memory so they can be subsequently served by later requests. For example, think of an online newspaper site. Once posted, articles change infrequently and hence can be cached and on first access and reused on all subsequent requests until the article is updated.

In general, caching relieves databases of heavy read traffic, as many queries can be served directly from the cache. It also reduces computation costs for objects that are expensive to construct, for example those needing queries that span several different databases. The net effect is to reduce the computational load on our services and databases and create head room for more requests.

Caching requires additional resources, and hence cost, to store cached results. However, well designed caching schemes are low cost compared to upgrading database and service nodes to cope with higher request loads. As an indication of the value of caches, approximately 3% of infrastructure at Twitter is dedicated to application level caches.[1] At Twitter scale, that is a lot of infrastructure!

Application level caching exploits dedicated distributed cache engines. The two predominant technologies in this area are memcached[2] and Redis[3]. Both are essentially distributed in-memory key-value stores designed for arbitrary data (strings, objects) from the results of database queries or downstream service API calls. The cache appears to services as a single store, and objects are allocated to individual cache servers using hashing on the object key.

The basic scheme is shown in Figure 1. The service first checks the cache to see if the data it requires is available. If so, it returns the cached contents as the results – we have what is known as a *cache hit*. If the data is not in the cache – a *cache miss* - the service retrieves the requested data from the database and writes the query results to the cache so it is available for subsequent client requests without querying the database.

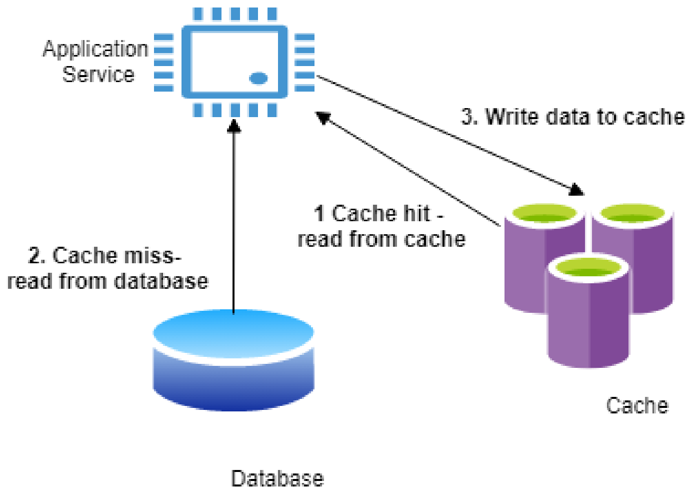
Application Service
3. Write data to cache
1 Cache hit - read from cache
2. Cache miss- read from database
Cache
Database

Figure 1 Application Level Caching

Let's return to our mythical winter resort business for a use case. At a busy resort, skiers and boarders can use their mobile app to get an estimate of the lift wait times across the resort. This enables them to plan and avoid congested areas where they will have to wait to ride a lift for say 15 minutes (or sometimes more!).

Every time a skier loads a lift, a message is sent to the company's service that collects data about skier traffic patterns. Using this data, the system can estimate lift wait times from the number of skiers who ride a lift and the rate they are arriving. This is an expensive calculation as it requires aggregating potentially 10's of thousands of lift ride records and performing the wait time calculation. For this reason, once the results are calculated, they are deemed valid for five minutes. Only after this time has elapsed is a new calculation performed and results produced.

Figure 2 shows an example of how a stateless `LiftWaitService` might work. When a request arrives, the service first checks the cache to see if the latest wait times are available. If they are, the results are immediately returned to the client. If the results are not in the cache, the service

[1] https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html

[2] https://memcached.org/

[3] https://redis.io/

calls a downstream service which can perform the lift wait calculations and returns them as a `List`. These results are then stored in the cache and then returned to the client.

Cache access requires a key with which to associate the results with. In this example we construct the key with the string “`liftwaittimes:`” concatenated with the resort identifier that is passed by the client to the service. This key is hashed by the cache to identify the server where the cached value resides. When we write a new value to the cache (line 8), we pass a value of 300 seconds as a parameter to the `put` operation. This is known as a *time to live*, or *TTL* value. It tells the cache that after 300 seconds this key-value pair should be evicted from the cache as the value is no longer relevant.

When the cache value is evicted, the next request will calculate the new values and store them in the cache. But while the cache value is valid, all requests will utilize it, meaning there is no need to perform the expensive lift wait time calculation for every call. Hence if we get N requests in a 5 minute period, N-1 are served from the cache. Imagine if N is 20,000. This is a lot of expensive calculations saved.

```
1.  public class LiftWaitService {
2.
3.    public List getLiftWaits(String resort) {
4.      List liftWaitTimes = cache.get(“liftwaittimes:” + resort);
5.        if (liftWaitTimes == null) {
6.           liftWaitTimes = skiCo.getLiftWaitTimes(resort);
7.           // add result to cache, expire in 300 seconds
8.           cache.put("liftwaittimes:" + resort, liftWaitTimes, 300);
9.        }
10.     return liftWaitTimes;
11.   }
12. }
```

Figure 2 Caching Example

Using an expiry time like the *TTL* is a common way to invalidate cache contents. It ensures a service doesn’t deliver stale, out of date results to a client. It also enables the system to have some control over cache contents, which are typically limited. If cached items are not flushed periodically, the cache may fill up. In this case, a cache will adopt a policy such as *least recently used* or *least accessed* to choose cached values to evict and create space for more current, timely results.

Application caching can provide significant throughput boosts, reduced latencies, and increased client application responsiveness. The key to achieving these desirable qualities is to satisfy as many requests as possible from the cache. This is known as the cache hit rate. The general design principle is to maximize the cache hit rate and minimize the cache miss rate – when a request cannot be satisfied by cached items. When a cache miss occurs, the request must be satisfied through querying databases or downstream services. The results of the request can then be written to the cache and hence be available for further accesses.

There’s no hard and fast rule on what the cache hit rate should be, as it depends on the cost of constructing the cache contents and the update rate of cached items. Ideal cache designs have many more reads than updates. This is because when an item must be updated, the application needs to invalidate cache entries that are now stale because of the update. This means the next request will result in a cache miss.

When items are updated regularly, the cost of cache misses can negate the benefits of the cache. Service designers therefore need to carefully consider query and update patterns an application experiences, and construct caching mechanisms that yield the most benefit. It is also crucial to monitor the cache usage once a service is in production to ensure the hit and miss rates are in line with design expectations. Caches will provide both management utilities and APIs to enable monitoring of the cache usage characteristics. For example, memcached is accessible through a `telnet` session. A large number of statistics are available, including the hit and miss counts as shown in the snippet in Figure 3.

```
STAT get_hits 98567
STAT get_misses 11001
STAT evictions 0
```

Figure 3 Example of memcached monitoring output

Application level caching is also known as the *cache-aside* pattern[4]. The name references the fact that the application code effectively bypasses the data storage systems if the required request results are available. This contrasts with other caching patterns, in which the application always reads from and writes to the cache. These are known as *read-through*, *write-through* and *write-behind* caches as explained below:

- **Read-through:** The application satisfies all requests by accessing the cache. If the data required is not available in the cache, a loader in invoked to access the data systems and store the results in the cache for the application to utilize.
- **Write-through:** The application always writes updates to the cache. When the cache is updated, a writer is invoked to write the new cache values to the database. When the database is updated, the application can complete the request.
- **Write-behind:** Like write-through, except the application does not wait for the value to written to the database from the cache. This increases request responsiveness at the expense of possible lost updates if the cache server crashes before a database update is completed. This is also known as a write-back cache, and internally is the strategy used by most database engines.

The beauty of these caching approaches is that they simplify application logic. Applications always utilize the cache for reads and writes, and the cache provides the 'magic' to ensure the cache interacts appropriately with the backend storage systems. This contrasts with the cache-side pattern, in which application logic must be cognizant of cache misses.

The application still needs to make this magic happen of course. These strategies require a cache technology which can be augmented with an application-specific handler that performs database reads and writes when the application accesses the cache. For example, NCache[5] supports provider interfaces that the application implements. These are invoked automatically on cache misses for read-through caches and on writes for write-through caches. Other such caches are essentially dedicated database caches, and hence require cache access to be identical to the underlying database model. An example of this is Amazon's DynamoDB Accelerator (DAX).[6] DAX sits between the application code and DynamoDB, and transparently acts as a high-speed in memory cache to reduce database access times.

One significant advantage of the cache-aside strategy is that it is resilient to cache failure. In such circumstances, as the cache is unavailable, all requests are essentially handled as a cache miss.

[4] https://www.ehcache.org/documentation/3.3/caching-patterns.html#cache-aside

[5] https://www.alachisoft.com/resources/docs/ncache/prog-guide/server-side-api-programming.html

[66] https://aws.amazon.com/dynamodb/dax/

Performance will suffer, but services will still be able to satisfy requests. In addition, scaling cache-aside platforms such as Redis and Memcached is straightforward due their simple, distributed key-value store model. For these reasons, the cache-aside pattern is the primary approach seen in massively scalable systems.

Web Caching

One of the reasons that Web sites are so highly responsive is that the Internet is littered with Web caches. Web caches stores a copy of a given resource for a defined time period. The caches intercept client requests and if they have a requested resource cached locally, it returns the copy rather than forwarding the request to the target service. Hence many requests can be satisfied without placing a burden on the service. Also, as the caches are closer to the client, the requests will have lower latencies.

Figure 4 gives an overview of the Web caching architecture. Multiple levels of caches exist, starting from the client's local Web browser cache, local *proxy* caches within organizations and Internet Service Providers, and reverse proxy caches that exist within the services execution domain. Web browser caches are also known as private caches (for a single user) and proxy caches are shared caches that support requests from multiple users.

Caches typically store the results of GET requests only, and the cache key is the URI of the associated GET. When a client sends a GET request, it may be intercepted by one or more caches along the request path. Any cache with a fresh copy of the requested resource may respond to the request. If no cached content is found, the request is served by the service endpoint, which is also called the origin server.

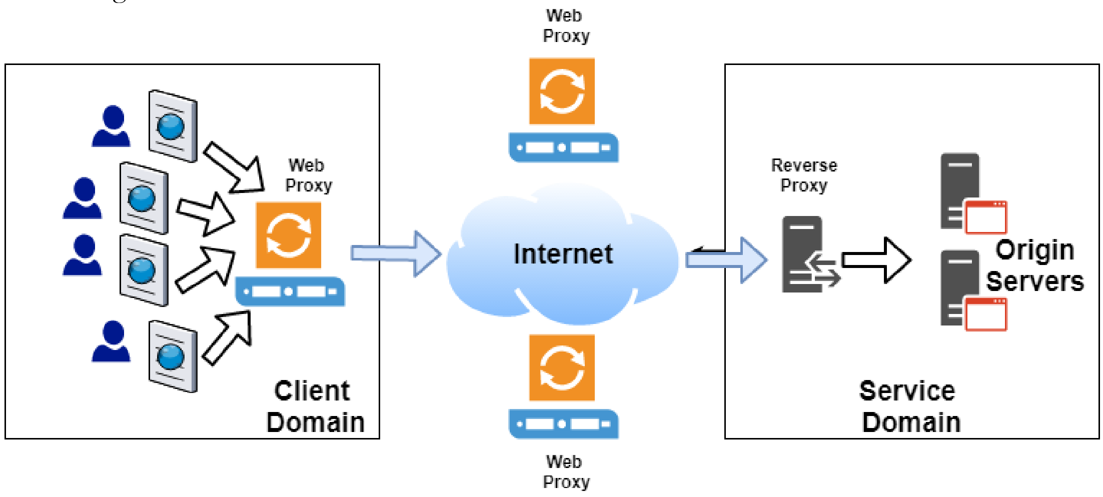

Figure 4 Web Caches in the Internet

Services can control what results are cached and for how long they are stored by using HTTP caching directives. Services set these directives in various HTTP response headers, as shown in the simple example in Figure 5. We will describe these directives in the following subsections.

```
Response:
HTTP/1.1 200 OK Content-Length: 9842
Content-Type: application/json
Cache-Control: public
```

```
5. Date: Fri, 26 Mar 2019 09:33:49 GMT
6. Expires: Fri, 26 Mar 2019 09:38:49 GMT
```

Figure 5 Example HTTP Response with caching directives

Cache-Control

The `Cache-Control` HTTP header can be used by client requests and service responses to specify how the caching should be utilized for the resources of interest.

- `no-store`: Specifies that a resource from a request response should not be cached. This is typically used for sensitive data that needs to be retrieved from the origin servers each request.
- `no-cache`: Specifies that a cached resource must be revalidated with an origin server before use. We discuss revalidation in the `Etag` subsection below.
- `private`: Specifies a resource can be cached only by a user-specific device such as a Web browser
- `public`: Specifies a resource can be cached by any proxy server
- `max-age`: defines the length of time in seconds a cached copy of a resource should be retained. After expiration, a cache must refresh the resource by sending a request to the origin server.

Expires and Last-Modified

The `Expires` and `Last-Modified` HTTP headers interact with the `max-age` directive to control how long cached data is retained.

Caches have limited storage resources and hence must periodically evict items from memory to create space. To influence cache eviction, services can specify how long resources in the cache should remain valid, or *fresh*. Once this time period for a cached resource expires, it becomes stale and may become a candidate for eviction. When a request arrives for a fresh resource, the cache serves the locally stored results without contacting the origin server.

Freshness is calculated using a combination of header values. The "Cache-Control: max-age=N" header is the primary directive, and this value the specifies the freshness period in seconds.

If max-age is not specified, the `Expires` header is checked next. If this header exists, then it is used to calculate the freshness period. As a last resort, the `Last-Modified` header can specify the freshness lifetime based on a heuristic calculation that the cache can support. This is usually calculated as (`Date` header value – `Last-Modified` header value)*0.1.

Etag

HTTP provides another directive that controls cache item freshness. This is known as an Etag. An Etag is an opaque value that van be used by a cache to check if a cached resource is still valid. Let's explain this using another winter sports example.

A ski resort posts a weather report at 6am every day during the ski season. If the weather changes during the day, the resort updates the report. Sometimes this happens two or three times each day, and sometimes not at all if the weather is stable. When a request arrives for the weather report, the service responds with a maximum age to define cache freshness, and also an ETag that represents the version of the weather report that was last issued. This is shown in Figure 6, which tells a cache to treat the weather report resource as fresh for at least 3600 seconds, or 60 minutes.

```
Request:
GET /skico.com/weather/Blackstone

Response:
HTTP/1.1 200 OK Content-Length: ...
Content-Type: application/json
Date: Fri, 26 Mar 2019 09:33:49 GMT
Cache-Control: public, max-age=3600
ETag: "09:33:49"
<!-- Content omitted -->
```

Figure 6 HTTP Etag Example

For the next hour, the cache simply serves this cached weather report to all clients who issue a GET request. This means the origin servers are freed from processing these requests – the outcome that we want from effective caching. After an hour though, the resource become stale. Now, when a request arrives for a stale resource, the cache forwards it to the origin server with a `If-None-Match` directive along with the `Etag` to enquire if the resource, in our case the weather report, is still valid. This is known as revalidation.

There are two possible responses to this request.

1. If the Etag in the request matches the value associated with the resource in the services, the cached value is still valid. The origin server can therefore return a 304 (Not Modified) response, as shown in Figure 7. No response body is needed as the cached value is still current, thus saving bandwidth, especially for large resources. The response may also include new cache directives to update the freshness of the cached resource.
2. The origin server may ignore the revalidation request and respond with a 200 response code and a response body and Etag representing the latest version of the weather report.

```
Request:
GET /upic.com/weather/Blackstone
If-None-Match: "09:33:49"
Response:
HTTP/1.1 304 Not Modified
```

Figure 7 Validating an Etag

In the service implementation, a mechanism is needed to support revalidation. In our weather report example, one strategy is as follows:

1. **Generate new daily report:** The weather report is constructed and stored in a database. The service also creates a new cache entry that identifies the weather report resource and associates an Etag with this version of the resource, for example {#resortname-weather, Etag value}.
2. **GET requests:** When a GET request arrives, return the weather report and the Etag. This will also populate Web caches along the network response path,
3. **Conditional GET requests:** Lookup the Etag value in cache at {#resortname-weather} and return 304 if the value has not changed. If the cached `Etag` has changed, return 200

along with the latest weather report and a new `Etag` value.

4. **Update weather report:** A new version of the weather report is stored in the database and the cached Etag value is modified to represent this new version of the response.

When used effectively, Web caching can significantly reduce latencies and save network bandwidth. This is especially true for large items such as images and documents. Further, as Web caches handle requests rather than application services, this reduces the request load on origin servers, creating additional capacity.

Proxy caches such as Squid[7] and Varnish[8] are extensively deployed in the Internet. Scalable applications therefore exploit the powerful facilities provided by HTTP caching to exploit his caching infrastructure.

Summary and Further Reading

Caching is an essential component of any scalable distribution. Caching stores information that is requested by many clients in memory and serves this information as the results to client requests. While the information is still valid, it can be served potentially millions of times without the cost of recreation.

Application caching using a distributed cache is the most common approach to caching in scalable systems. This approach requires the application logic to check for cached values when a client request arrives and return these if available. If the cache hit rate is high, with most requests being satisfied with cached results, load on backend services and database can be considerably reduced.

The Internet also has a built in, multilevel caching infrastructure. Applications can exploit this through the use of cache directives that are part of HTTP headers. These directives enable a service to specify what information can be cached, for how long it should be cached, and protocol for checking to see if a stale cache entry is still valid.

[7] http://www.squid-cache.org/

[8] https://varnish-cache.org/