Cache vs. Session Store

There's a saying, "In Silicon Valley you can throw a rock in any direction and you'll probably hit a software engineer." The other day I was traveling by train from San Jose to San Francisco when I overheard two software engineers comparing the speed of the products they were developing. When I heard one developer say, "Our app runs fast because we cache all the session data," I realized that even a Silicon Valley engineer may be unaware of the subtle differences between a cache and a session store. Redis, as an in-memory database, is used for both caching and session store scenarios. Let's look at how the use cases differ.

Cache

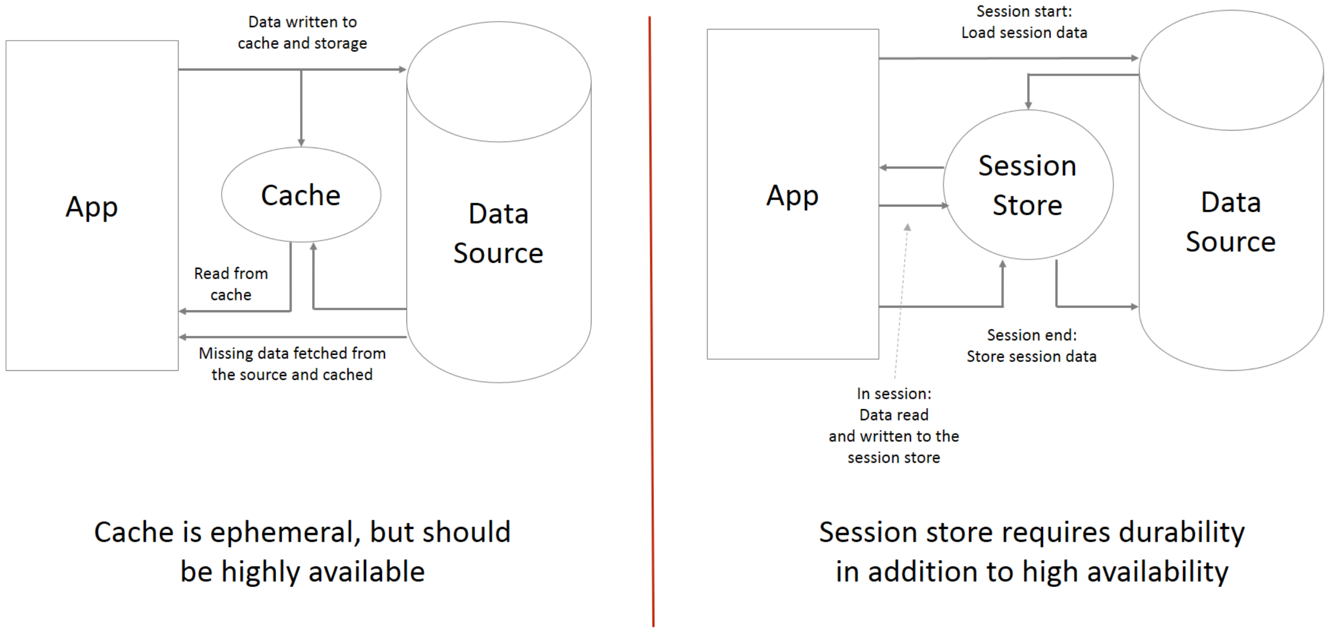
An application stores data in the cache to serve future requests faster. Typically, the cache storage is located in the RAM and has sub millisecond latency.

In the data fetch lifecycle, the application first looks for the data in the cache. If there's a hit (i.e. the data is in the cache), it serves the data instantaneously. Conversely, if there's a miss it fetches the data from a permanent store, stores a copy in the cache, and serves the data to the consumer. For all future requests, the data is already there in the cache and is served faster. When an application updates the data, it updates both the cache and the permanent store.

This lifecycle works well for scenarios where different consumers request the same data over a period of time. One should also note that the data is stored at the application level, and not at the user level. So the data that's stored in the cache is shared among users. Images, videos, static HTML pages, JavaScript libraries and style sheets are examples of data that are often stored in cache.

Session Store

A session-oriented application (a web application, for example) starts a session when a user logs in, and is active until the user logs out or the session times out. During this period, the application stores all session-related data either in the main memory or in a session store—a database that doesn't lose the data when the application goes down. Session data may include user profile information, messages, personalized data and themes, recommendations, targeted promotions and discounts, etc.



Data written to cache and storage
App
Cache
Data Source
Read from cache
Missing data fetched from the source and cached
Cache is ephemeral, but should be highly available
Session start: Load session data
App
Session Store
Data Source
Session end: Store session data
In session: Data read and written to the session store
Session store requires durability in addition to high availability

Figure 1. Cache vs Session Store

The following points contrast session store from a cache:

1. In a session store, the data is not shared between the sessions of different users. The solutions architecture must ensure the data remains isolated between users.
2. As described in the figure above, the life cycle of the data differs between the cache and the session store. The figure shows a write-through cache that writes to both the cache and the backend data store. This lowers the throughput for write operations. On the other hand, session stores rely on reading and writing data to the in-memory database. They cannot afford to compromise on the performance for write operations.
3. Session store data isn't ephemeral; it's the only source of truth when the session is live. Therefore, it needs to satisfy the data durability requirements of a true database. If the data in a cache gets lost, there's always a copy in the permanent store.
4. A session store requires replication, high availability, and data durability to ensure transactional data is not lost. Whereas, the high-availability requirements for caching are driven by the operational requirements and need to prevent cache stampeding.

Designing Cache and Session Store with Redis Enterprise

Redis Enterprise is a popular database ideal for both cache and session store use cases, delivering both the high-availability required for caching and session store scenarios as well as the durability needed for session store with in-memory replication. It's possible to use Redis Enterprise as both a cache and a session store in a single setup, as shown in the picture below.

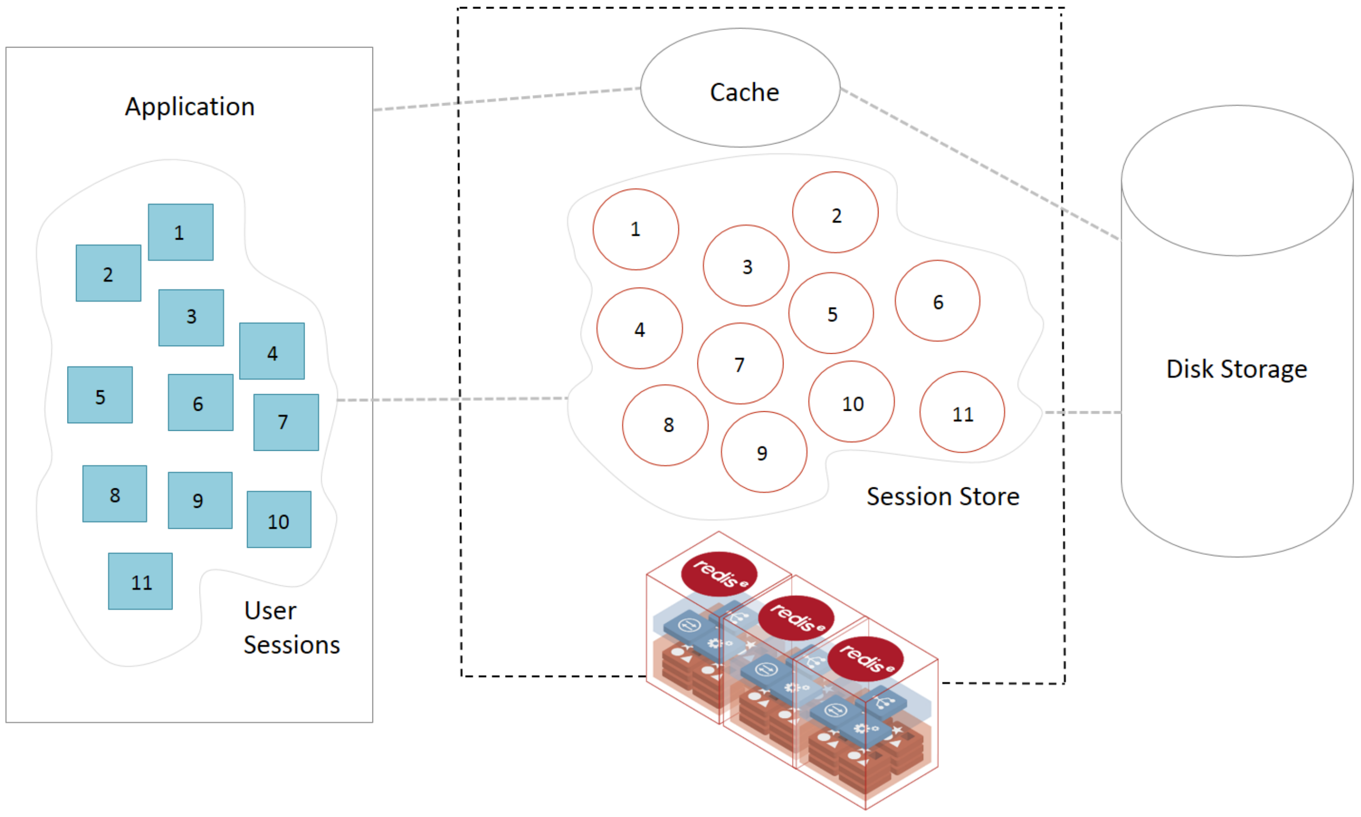


Figure 2. Designing Cache and Session Store with Redis Enterprise

In this design, the application layer accesses and maintains the data in the cache. Therefore, all the sessions running inside the application access the same data stored in the cache.

In addition to the data in cache, each session holds the session-related data inside Redis Enterprise. How do you ensure that the sessions don't access each other's data? Here's a solution: First, each session must acquire a random session id that's not shared with other sessions. Second, the session must append the session id to the keys. In the example below, a session stores the personal information in a Hash data structure, and the user recommendations in a Set data structure. As you may note, the keys have session ID in them. This prevents sessions from accessing the data owned by other sessions.

```
session_data:(session_id):personal_info
name - John Smith
email - john@smith.com
phone - 987-111-1111
```

```
session_data:(session_id):recommendations
{"product a", "product b", "product c",.....“product n"}
```

In conclusion, cache and session store are two different use cases. While high availability is important to a cache to prevent situations such as cache stampede, a session store requires high availability and durability to support transactional data and uninterrupted user engagement. With proper architecture and design, a single database instance of Redis Enterprise can be used for both caching and session data use cases while ensuring data separation between the sessions.