1. Let says, there is Application Server1 & Server 2 connected to Database DB1 & Cache Server CS1  
   There are two user U1 & U2 connected to Server1 & Server2 respectively and modifying a key K1🡪V1 store in CS1  
   Application server first write to database server and acknowledge the user then asynchronously update cache server. Now, follow the events below: -

* U1 update K1🡪V2
* Server1 update it in DB1 and acknowledge back to U1, still has not updated the cache
* U2 update K1🡪V3
* Server2 update it in DB1 and acknowledge back to U2 and immediately update cache

before U1 update goes in

* Server1 update U1 modification in cache, over writing U2 modification
* Now, DB & Cache Server are in inconsistent state, i.e. DB (K1🡪V3) while Cache (K1🡪V2)
* This issue occurred because modification was done by two operations coming from different servers. If both the modification had come from same user from same server, possibly then we could have applied some synchronization logic to avoid this inconsistency. This issue could be solved if update of Cache & DB is done in a single atomic transaction using 2PC which is costly operation
* Another way to solve is using Kafka. All the modification by Application server will send to Kafka Queue, i.e. it will have {(K1🡪V2), (K1🡪V3)}
* Then DB & Cache server will read & apply the updates in the same order from the queue, i.e. first (K1🡪V2) followed by (K1🡪V3). Each client (DB & Cache) will have its own set of modification queue provided by Kafka Server. Each of them will read from their own queue
* This way both DB & Cache will be in inconsistent state

1. When we add a cache layer between service, the major problem is how to keep it updated. E.g. Service A need information from Service B. If queried for the first time to B, give me list of all the users belonging to Group1. Then A will cache the result.  
   Next time, when it has to send messages to all the user of Group1, it will read from its local cache, who all are the users of this group are and send the messages to all of them.  
   Now imagine a new user is added to the group. How will service A will know this.

* Let B, inform all its clients (i.e. A) to update their cache whenever there is a change. It works, but not efficient
* Whenever a new message is about to be send to all the users, A will send a hash of information it has. i.e. A will send Hash(Group1) to B. Then B will match it with its content. If the hashes are equal, then B will respond with ‘True’. In this case A can go ahead with the message sending to list of users it’s known for that group. If response after hash comparison is ‘F’, then A will fetch the latest copy of Group1.  
  Sending hash is good as it will not consume bandwidth

*Update repository first then cache 🡪 Cache Aside (Application has responsibility to keep database & cache up-to-date)*

*Update cache first then repository 🡪 Write Behind (done asynchronously. Cache has the responsibility to update the cache)*

*Update simultaneously both at cache & repository 🡪 Write Through (Cache store has the responsibility to perform write on the database when there is a write or there is cache miss to update itself)*

Redis

* It allows to store data structure such as list, set, hash table
* It offers persistent to disk, hence if a key is updated & still not persisted to the origin data store and Redis server crash, the updated value is not lost. Once server is up, the updated value can be read from and updated to the actual data store
* It maintains log of all operations
* The new release > 3.0 support distributed server

When to cache

* Consider caching data that is read frequently but modified infrequently (for example, data that has a higher proportion of read operations than write operations)
* Caching typically works well with data that is immutable or that changes infrequently. Examples include reference information, such as product and pricing information in an e-commerce application, or shared static resources that are costly to construct
* it can be useful to seed the cache with the static user profile data for customers who use the application regularly (perhaps every day), but not for customers who use the application only once a week.
* Caching can also be used to avoid repeating CPU-intensive computations
* However, we don't recommend that you use the cache as the authoritative store of critical information. Instead, ensure that all changes that your application cannot afford to lose are always saved to a persistent data store
* consider a device that continually reports status or some other measurement. If an application chooses not to cache this data on the basis that the cached information will nearly always be outdated, then the same consideration could be true when storing and retrieving this information from the data store. In the time it takes to save and fetch this data, it might have changed
* In a situation such as this, consider the benefits of storing the dynamic information directly in the cache instead of in the persistent data store. If the data is non-critical and does not require auditing, then it doesn't matter if the occasional change is lost

Caching patterns

* Cache Aside 🡪 Load data on demand into a cache from a data store. If the data isn't in the cache, application retrieved the data from the data store and added to the cache. Application is responsible for reading and writing from the database and to keep cache synchronize with the database
* When to use it
* A cache doesn't provide native read-through and write-through operations
* Resource demand is unpredictable. This pattern enables applications to load data on demand. It makes no assumptions about which data an application will require in advance.
* When not to use it
* When the cached data set is static. If the data will fit into the available cache space, prime the cache with the data on startup and apply a policy that prevents the data from expiring
* For caching session state information in a web application hosted in a web farm. In this environment, you should avoid introducing dependencies based on client-server affinity
* Disadvantage
* Each cache miss results in three trips, which can cause a noticeable delay.
* Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through
* When a node fails, it is replaced by a new, empty node, increasing latency
* Read through
* Advantage
* read-through may reduce database calls by blocking parallel calls for same object
* There are many situations where a cache-item expires, and multiple parallel user threads end up hitting the database. Multiplying this with millions of cached-items and thousands of parallel user requests, the load on the database becomes noticeably higher
* Write through
* when the application updates a piece of data in the cache (that is, calls put(...) to change a cache entry,) the operation will not complete (that is, the put will not return) until it has successfully stored the data to the underlying data source. This does not improve the performance as we are still dealing with the latency of writing to the data source
* Disadvantage
* When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. Cache-aside in conjunction with write through can mitigate this issue.
* Most data written might never read, which can be minimized with a TTL
* Write Behind
* When the application updates X in the cache, X is added to the write-behind queue (if it isn't there already; otherwise, it is replaced), and after the specified write-behind delay it will update the data source with the latest state of X. The data in the data source will never lag behind the cache by more than the write-behind delay
* Advantage
* The application improves in performance, because the user does not have to wait for data to be written to the underlying data source. (The data is written later, and by a different execution thread.)
* The number of write are reduced because multiple changes to the same object within the write-behind interval are "coalesced" and only written once to the underlying data source
* The Write-Behind feature can be configured in such a way that a write failure will result in the object being re-queued for write. If the data that the application is using is in the cache, the application can continue operation without the database being up
* Disadvantage
* database updates occur outside of the cache transaction; that is, the cache transaction will (in most cases) complete before the database transaction(s) begin. This implies that the database transactions must never fail; if this cannot be guaranteed, then rollbacks must be accommodated
* If other applications share the database, there is no way to guarantee that a write-behind transaction will not conflict with an external update. This implies that write-behind conflicts must be handled heuristically or escalated for manual adjustment by a human operator
* As a rule of thumb, mapping each cache entry update to a logical database transaction is ideal, as this guarantees the simplest database transactions
* There could be data loss if the cache goes down prior to its contents hitting the data store.
* It is more complex to implement write-behind than it is to implement cache-aside or write-through

Caching at various layer

* Within one web browser
* DNS caching by ISP
* CDN caching static content (Dynamic Page Caching)

Dynamic Page Caching

* Dynamic Page Caching (DPC) enables the caching of HTML pages based on request path, query strings, cookies, and request headers. Using any combination of attributes from an HTTP Request, Akamai will decide when and how to cache the responses and serve them. Benefits are as follows-
* Reduce the number of requests to the origin infrastructure
* Improve the Time to First Byte (TTFB) and overall performance
* Free up the origin to operate more efficiently for truly dynamic requests
* In the example of an e-Commerce store, users who are not logged-in often see the same content and represent a large portion of hits to the site. These users' empty shopping cart and "Log In / Register" message in the header may be the same for all anonymous users. In this case, understanding if the user is anonymous based on attributes of the HTTP request will allow those users to be served cached content
* Often, a cookie is set, or can be set, for logged-in users but not for anonymous users. Akamai can check that this cookie is not present in the HTTP request, and serve the HTML content from cache
* Within Web platforms through proxy caching & database caching
* Varnish is a caching HTTP reverse proxy or HTTP accelerator that sits in front of one’s application layer, caching rendered responses for re-use later
* Cache-Control HTTP Headers 🡪 max-age, public, private, no-store, no-cache.  
  If a response is cacheable for 10 seconds, then must-revalidate kicks in after 10 seconds, whereas no-cache implies must-revalidate after 0 seconds.  
  no-cache direct agent to re-validate all responses  
  must-revalidate directs agents to revalidate stale responses
* Application caching
  + In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage
  + Since the data is held in RAM, it is much faster than typical databases where data is stored on disk. RAM is more limited than disk, so cache invalidation algorithms such as least recently used (LRU) can help invalidate 'cold' entries and keep 'hot' data in RAM
* Caching database queries result 🡪 Making queries to a relational database such as MySql, Oracle, or PostgreSql is typically the bottleneck in terms of speed and performance within a web platform. It requires data to be read from disk and then queried upon
* Using Redis 🡪 provide persistence & data structure like list, set etc.
* Using Memcache

The Dogpile Effect

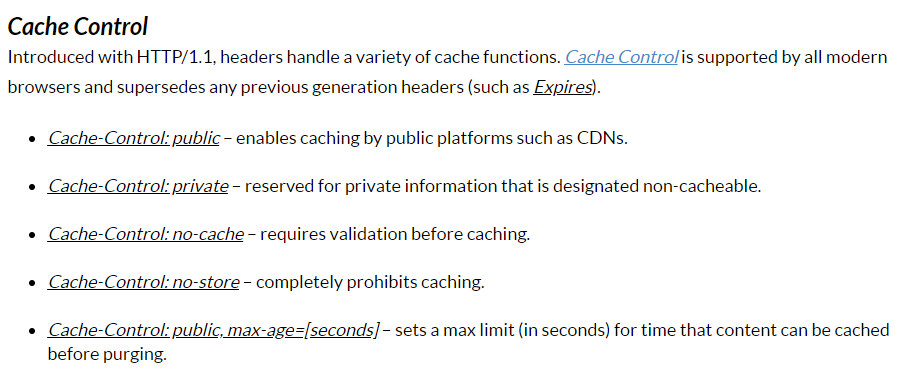
* You check if value is cached. If it is, you fetch cached value from cache and serve it. If it’s not, you generate new value and store in cache for future requests
* what if value expires and then you get hundreds of requests? It cannot be served from cache anymore, so your databases are hit with numerous processes trying to re-generate the value. And the more requests databases receive, the slower and less responsive they get. Load spikes. Until eventually they likely go down
* Preventing dogpile effect boils down to having just one process (first one to come) regenerating new value while other subsequent processes serving stale value from cache until it’s refreshed by the first process.
* It can also be solved use Mem cache lease strategy discussed below

Distributed Caching

* Private Cache
* Advantage
* Cache data is present in the same address space in which application is running, hence data access is fast
* Disadvantage
* If you have multiple instances of an application running concurrently, each application instance has its own independent cache holding its own copy of the data. If this data is not static, it is likely that different application instances hold different versions of the data in their caches. Therefore, the same query performed by these instances can return different results
* Shared Cache
  + Advantage
* Since the cache is centralized, multiple instances will see the same result
* High scalability as cache can be distributed on multiple nodes. An application instance simply sends a request to the cache service. The underlying infrastructure is responsible for determining the location of the cached data in the cluster
  + Disadvantage
* The cache is slower to access because it is no longer held locally to each application instance.
* The requirement to implement a separate cache service might add complexity to the solution

CDN Live Streaming Caching

* When request for live streaming video (or match statistics) is received by CDN server from a user (staying in Mumbai) for the first time, it will get the data from the origin server and cache it.
* When a request from a new list of users staying in Mumbai is received, the CDN server will hold request from both the users, it will fetch the live stream data from the origin server only once and transfer it to all the users
* So, each CDN servers will hold the request of all its nearby user. It will get the content only once from the origin server and distribute it to all the users. This reduces the latency. Data is move close to the users
* This is how Hot star and other live streaming works

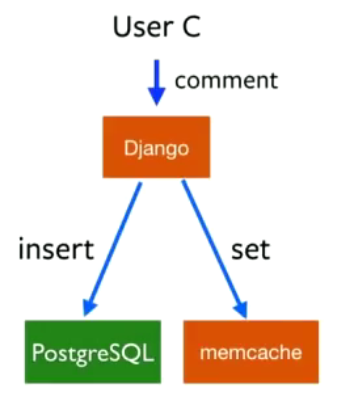


Cache-Control: private 🡪 allow browsers to cache not intermediate sever like CDN

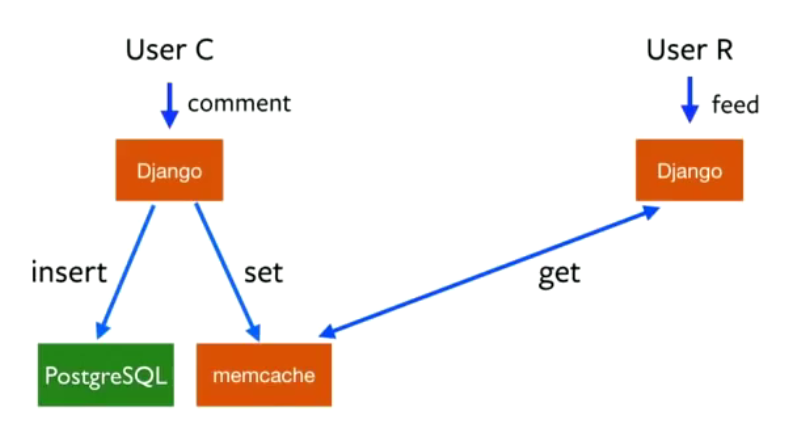
=====================================================================================

**How to keep both Server DB & Cache consistent**

1. Django server update to both DB & Cache present in same data center.   
   Record will be in consistent state if both insert & set complete successfully.   
   Problem 🡪 Since, we are performing dual write if one fails (insert) & other succeed (set) or   
   vice-versa, then it will result in in-consistent view.   
     
   Solution 🡪 To overcome this, we must perform both insert & set within a single transaction   
   i.e. 2 Phase Commit, which will then guarantee the system will always be in consistent state always.  
     
   Impact 🡪 But this will impact throughput & performance

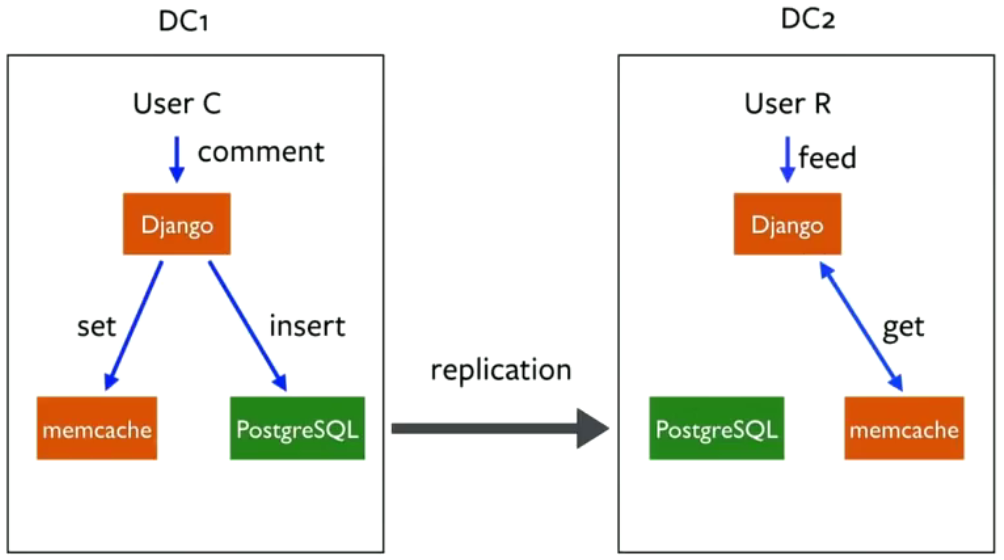


When both users are served (write & read) from same server, there is no issue



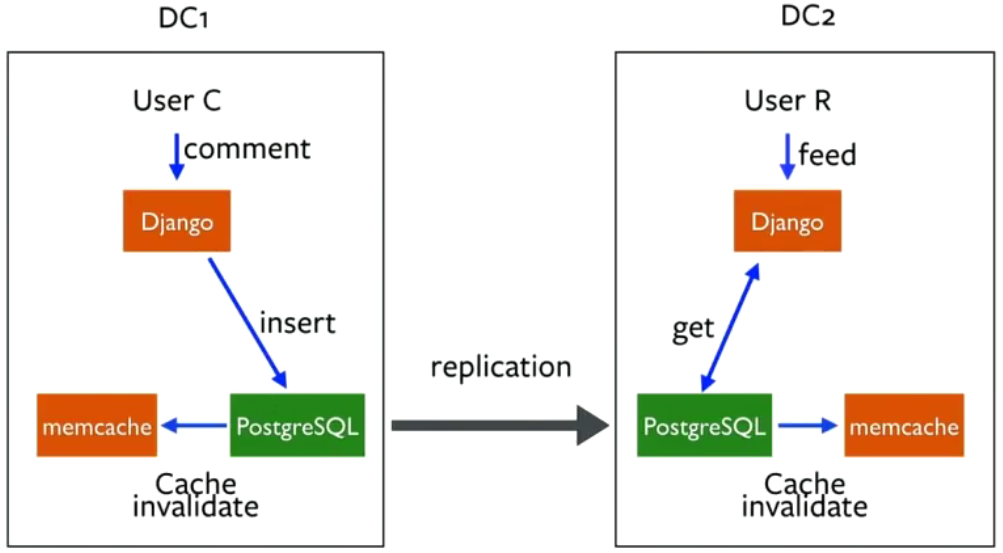
1. If both the users are being served from different data centers, then they will not see consistent view of the data  
   Since, user request is directed to different data center based on sharding key

User C comment was updated to Mem-cache of DC1 but read was done from Mem-cache of DC2. Hence, User R will not see up to date result.



1. To fix the above issue

* Stop dual writing, i.e. perform insert only to Database. Don’t write to mem-cache
* Using database replication mechanism replicate the changes to all the data centers
* Run a daemon process which will tail the changes being performed on its local database (i.e. Change Data Capture using Debezium).   
  Each Daemon process running in the Data Center will capture the changes in its local database and mark it as stale in the local mem-cache
* Now, when User R perform read, the mem-cache will have stale entry and hence he will be redirected to Database to get the updated value



Problem 🡪 While this design solves the in-consistency issue and acceptable for many use cases where data is not updated often.  
But if the requested data is being marked as stale very frequently then user will be redirected to the database every time and hence increasing the load on database & cache is not being used efficiently

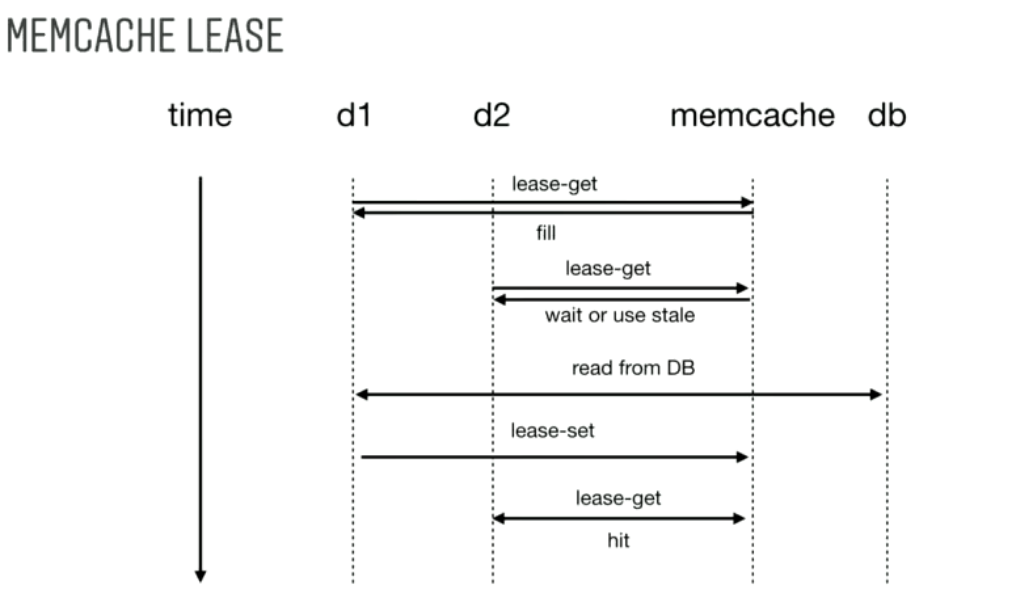
Imagine we need to maintain the number of likes/views received on the user post. Number of likes/views will change quite frequently, and each likes/view will mark its value as stale in cache. Next read will have to be served from database

Another scenario imagines when a data suddenly become popular and request for its coming from various users, since the cache does not have latest copy hence all the request will be

re-directed to the data base server

1. Using lease base approach to solve above problems. This works if the application can live with stale value for some period

* First request d1 goes to cache & cache server respond that it has stale value hence it can go and read from database
* Before d1 has completed it read, imagine d2 comes for the same request, this time cache server instead of sending d2 to database server will ask it to retry after exponential back-off or if its fine they can use the stale information. (e.g. for showing number of likes it’s not a problem if we show 1.1 Million or 1.1 Million + 10)
* d1 read from database & update in the cache. Next request from d2 will now get the most up to date result



1. Asf
2. asg
3. adgadg