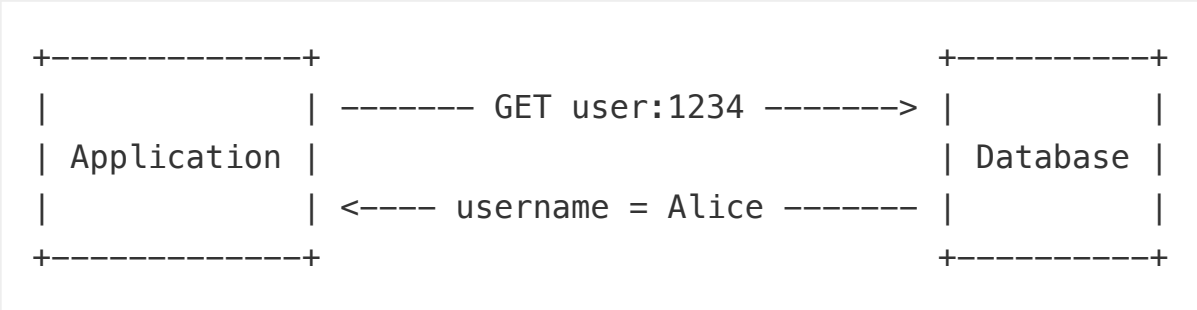
Redis server-assisted client side caching

Client side caching is a technique used in order to create high performance services. It exploits the available memory in the application servers, that usually are distinct computers compared to the database nodes, in order to store some subset of the database information directly in the application side.

Normally when some data is required, the application servers will ask the database about such information, like in the following diagram:



When client side caching is used, the application will store the reply of popular queries directly inside the application memory, so that it can reuse such replies later, without contacting the database again.

While the application memory used for the local cache may not be very big, the time needed in order to access the local computer memory is orders of magnitude smaller compared to asking a networked service like a database. Since often the same small

percentage of data are accessed very frequently this pattern can greatly reduce the latency for the application to get data and, at the same time, the load in the database side.

Moreover there are many datasets where items change very infrequently. For instance most user posts in a social network are either immutable or rarely edited by the user. Adding this to the fact that usually a small percentage of the posts are very popular, either because a small set of users have a lot of followers and/or because recent posts have a lot more visibility, it is clear why such pattern can be very useful.

Usually the two key advantages of client side caching are:

1. Data is available with a very small latency.
2. The database system receives less queries, allowing to serve the same dataset with a smaller number of nodes.

There are only two big problems in computer science...

A problem with the above pattern is how to invalidate the information that the application is holding, in order to avoid presenting stale data to the user. For example after the application above locally cached the user:1234 information, Alice may update her username to Flora. Yet the application may continue to serve the old username for user 1234.

Sometimes, depending on the exact application we are modeling, this problem is not a big deal, so the client will just use a fixed maximum "time to live" for the cached information. Once a given amount of time has elapsed, the information will no longer be considered valid. More complex patterns, when using Redis, leverage the Pub/Sub system in order to send invalidation messages to clients listening. This can be made to work but is tricky and costly from the point of view of the bandwidth used, because often such patterns involve sending the invalidation messages to every client in the application, even if certain clients may not have any copy of the invalidated data. Moreover every application query altering the data requires to use the PUBLISH command, costing the database more CPU time to process this command.

Regardless of what schema is used, there is a simple fact: many very large applications implement some form of client side caching, because it is the next logical step to having a fast store or a fast cache server. For this reason Redis 6 implements direct support for client side caching, in order to make this pattern much simpler to implement, more accessible, reliable and efficient.

The Redis implementation of client side caching

The Redis client side caching support is called *Tracking*, and has two modes:

- In the default mode, the server remembers what keys a given client accessed, and send invalidation messages when the same keys are modified. This costs memory in the

server side, but sends invalidation messages only for the set of keys that the client could have in memory.
- In the *broadcasting* mode instead the server does not attempt to remember what keys a given client accessed, so this mode does not cost any memory at all in the server side. Instead clients subscribe to key prefixes such as `object:` or `user:`, and will receive a notification message every time a key matching such prefix is touched.

To recap, for now let's forget for a moment about the broadcasting mode, to focus on the first mode. We'll describe broadcasting later more in details.

1. Clients can enable tracking if they want. Connections start without tracking enabled.
2. When tracking is enabled, the server remembers what keys each client requested during the connection lifetime (by sending read commands about such keys).
3. When a key is modified by some client, or is evicted because it has an associated expire time, or evicted because of a *maxmemory* policy, all the clients with tracking enabled that may have the key cached, are notified with an *invalidation message*.
4. When clients receive invalidation messages, they are required to remove the corresponding keys, in order to avoid serving stale data.

This is an example of the protocol:

- Client 1 –> Server: CLIENT TRACKING ON
- Client 1 –> Server: GET foo
- (The server remembers that Client 1 may have the key "foo" cached)
- (Client 1 may remember the value of "foo" inside its local memory)
- Client 2 –> Server: SET foo SomeOtherValue
- Server –> Client 1: INVALIDATE "foo"

This looks great superficially, but if you think at 10k connected clients all asking for millions of keys in the story of each long living connection, the server would end up storing too much information. For this reason Redis uses two key ideas in order to limit the amount of memory used server side, and the CPU cost of handling the data structures implementing the feature:

- The server remembers the list of clients that may have cached a given key in a single global table. This table is called the **Invalidation Table**. Such invalidation table can contain a maximum number of entries, if a new key is inserted, the server may evict an older entry by pretending that such key was modified (even if it was not), and sending an invalidation message to the clients. Doing so, it can reclaim the memory used for this key, even if this will force the clients having a local copy of the key to evict it.
- Inside the invalidation table we don't really need to store pointers to clients structures, that would force a garbage collection procedure when the client disconnects: instead what we do is just storing client IDs (each Redis client has an unique numerical ID). If a client disconnects, the information will be incrementally garbage collected as caching slots are invalidated.

- There is a single keys namespace, not divided by database numbers. So if a client is caching the key `foo` in database 2, and some other client changes the value of the key `foo` in database 3, an invalidation message will still be sent. This way we can ignore database numbers reducing both the memory usage and the implementation complexity.

Two connections mode

Using the new version of the Redis protocol, RESP3, supported by Redis 6, it is possible to run the data queries and receive the invalidation messages in the same connection. However many client implementations may prefer to implement client side caching using two separated connections: one for data, and one for invalidation messages. For this reason when a client enables tracking, it can specify to redirect the invalidation messages to another connection by specifying the "client ID" of different connection. Many data connections can redirect invalidation messages to the same connection, this is useful for clients implementing connection pooling. The two connections model is the only one that is also supported for RESP2 (that lacks the ability to multiplex different kind of information in the same connection).

We'll show an example, this time by using the actual Redis protocol in the old RRESP2 mode, how a complete session, involving the following steps: enabling tracking redirecting to another connection, asking for a key, and getting an invalidation message once such key gets modified.

To start, the client opens a first connection that will be used for invalidations, requests the connection ID, and subscribes via Pub/Sub to the special channel that is used to get invalidation messages when in RESP2 modes (remember that RESP2 is the usual Redis protocol, and not the more advanced protocol that you can use, optionally, with Redis 6 using the HELLO command):

```
(Connection 1 -- used for invalidations)
CLIENT ID
:4
SUBSCRIBE __redis__:invalidate
*3
$9
subscribe
$20
__redis__:invalidate
:1
```

Now we can enable tracking from the data connection:

```
(Connection 2 -- data connection)
CLIENT TRACKING on REDIRECT 4
+OK

GET foo
$3
bar
```

The client may decide to cache `"foo" => "bar"` in the local memory.

A different client will now modify the value of the "foo" key:

```
(Some other unrelated connection)
SET foo bar
+OK
```

As a result, the invalidations connection will receive a message that invalidates the specified key.

```
(Connection 1 -- used for invalidations)
*3
$7
message
$20
__redis__:invalidate
*1
$3
foo
```

The client will check if there are cached keys in such caching slot, and will evict the information that is no longer valid.

Note that the third element of the Pub/Sub message is not a single key but is a Redis array with just a single element. Since we send an array, if there are groups of keys to invalidate, we can do that in a single message.

A very important thing to understand about client side caching used with RESP2, and a Pub/Sub connection in order to read the invalidation messages, is that using Pub/Sub is entirely a trick **in order to reuse old client implementations**, but actually the message is

not really sent to a channel and received by all the clients subscribed to it. Only the connection we specified in the `REDIRECT` argument of the `CLIENT` command will actually receive the Pub/Sub message, making the feature a lot more scalable.

When RESP3 is used instead, invalidation messages are sent (either in the same connection, or in the secondary connection when redirection is used) as `push` messages (read the RESP3 specification for more information).

What tracking tracks

As you can see clients do not need, by default, to tell the server what keys they are caching. Every key that is mentioned in the context of a read only command is tracked by the server, because it *could be cached*.

This has the obvious advantage of not requiring the client to tell the server what it is caching. Moreover in many clients implementations, this is what you want, because a good solution could be to just cache everything that is not already cached, using a first-in first-out approach: we may want to cache a fixed number of objects, every new data we retrieve, we could cache it, discarding the oldest cached object. More advanced implementations may instead drop the least used object or alike.

Note that anyway if there is write traffic on the server, caching slots will get invalidated during the course of the time. In general when the server assumes that what we get we also cache, we are making a tradeoff:

1. It is more efficient when the client tends to cache many things with a policy that welcomes new objects.
2. The server will be forced to retain more data about the client keys.
3. The client will receive useless invalidation messages about objects it did not cache.

So there is an alternative described in the next section.

Opt-in caching

Clients implementations may want to cache only selected keys, and communicate explicitly to the server what they'll cache and what not: this will require more bandwidth when caching new objects, but at the same time will reduce the amount of data that the server has to remember, and the amount of invalidation messages received by the client.

In order to do so, tracking must be enabled using the OPTIN option:

```
CLIENT TRACKING on REDIRECT 1234 OPTIN
```

In this mode, by default keys mentioned in read queries *are not supposed to be cached*, instead when a client wants to cache something, it must send a special command

immediately before the actual command to retrieve the data:

```
CLIENT CACHING YES
+OK
GET foo
"bar"
```

The `CACHING` command affects the command executed immediately after it, however in case the next command is MULTI, all the commands in the transaction will be tracked. Similarly in case of Lua scripts, all the commands executed by the script will be tracked.

Broadcasting mode

So far we described the first client side caching model that Redis implements. There is another one, called broadcasting, that sees the problem from the point of view of a different tradeoff, does not consume any memory on the server side, but instead sends more invalidation messages to clients. In this mode we have the following main behaviors:

- Clients enable client side caching using the `BCAST` option, specifying one or more prefixes using the `PREFIX` option. For instance: `CLIENT TRACKING on REDIRECT 10 BCAST PREFIX object: PREFIX user:`. If no prefix is specified at all, the prefix is assumed to be the empty string, so the client will receive invalidation messages for every key that gets modified. Instead if one or more prefixes are used, only keys matching the one of the specified prefixes will be sent in the invalidation messages.
- The server does not store anything in the invalidation table. Instead it only uses a different **Prefixes Table**, where each prefix is associated to a list of clients.
- No two prefixes can track overlapping parts of the keyspace. For instance, having the prefix foo and foob would not be allowed, since they would both trigger an invalidation for the key foobar. However, just using the prefix foo is sufficient.
- Every time a key matching any of the prefixes is modified, all the clients subscribed to such prefix, will receive the invalidation message.
- The server will consume a CPU proportional to the number of registered prefixes. If you have just a few, it is hard to see any difference. With a big number of prefixes the CPU cost can become quite large.
- In this mode the server can perform the optimization of creating a single reply for all the clients subscribed to a given prefix, and send the same reply to all. This helps to lower the CPU usage.

The NOLOOP option

By default client side tracking will send invalidation messages even to client that modified the key. Sometimes clients want this, since they implement a very basic logic that does not

involve automatically caching writes locally. However more advanced clients may want to cache even the writes they are doing in the local in-memory table. In such case receiving an invalidation message immediately after the write is a problem, since it will force the client to evict the value it just cached.

In this case it is possible to use the `NOLOOP` option: it works both in normal and broadcasting mode. Using such option, clients are able to tell the server they don't want to receive invalidation messages for keys that are modified by themselves.

Avoiding race conditions

When implementing client side caching redirecting the invalidation messages to a different connection, you should be aware that there is a possible race condition. See the following example interaction, where we'll call the data connection "D" and the invalidation connection "I":

```
[D] client -> server: GET foo
[I] server -> client: Invalidate foo (somebody else touched it)
[D] server -> client: "bar" (the reply of "GET foo")
```

As you can see, because the reply to the GET was slower to reach the client, we received the invalidation message before the actual data that is already no longer valid. So we'll keep serving a stale version of the foo key. To avoid this problem, it is a good idea to populate the cache when we send the command with a placeholder:

```
Client cache: set the local copy of "foo" to "caching-in-progress"
[D] client-> server: GET foo.
[I] server -> client: Invalidate foo (somebody else touched it)
Client cache: delete "foo" from the local cache.
[D] server -> client: "bar" (the reply of "GET foo")
Client cache: don't set "bar" since the entry for "foo" is missing.
```

Such race condition is not possible when using a single connection for both data and invalidation messages, since the order of the messages is always known in that case.

What to do when losing connection with the server

Similarly, if we lost the connection with the socket we use in order to get the invalidation messages, we may end with stale data. In order to avoid this problem, we need to do the following things:

1. Make sure that if the connection is lost, the local cache is flushed.
2. Both when using RESP2 with Pub/Sub, or RESP3, ping the invalidation channel periodically (you can send PING commands even when the connection is in Pub/Sub mode!). If the connection looks broken and we are not able to receive ping backs, after a maximum amount of time, close the connection and flush the cache.

What to cache

Clients may want to run an internal statistics about the amount of times a given cached key was actually served in a request, to understand in the future what is good to cache. In general:

- We don't want to cache many keys that change continuously.
- We don't want to cache many keys that are requested very rarely.
- We want to cache keys that are requested often and change at a reasonable rate. For an example of key not changing at a reasonable rate, think at a global counter that is continuously `INC`Remented.

However simpler clients may just evict data using some random sampling just remembering the last time a given cached value was served, trying to evict keys that were not served recently.

Other hints about client libraries implementation

- Handling TTLs: make sure you request also the key TTL and set the TTL in the local cache if you want to support caching keys with a TTL.
- Putting a max TTL in every key is a good idea, even if it had no TTL. This is a good protection against bugs or connection issues that would make the client having old data in the local copy.
- Limiting the amount of memory used by clients is absolutely needed. There must be a way to evict old keys when new ones are added.

Limiting the amount of memory used by Redis

Just make sure to configure a suitable value for the maximum number of keys remembered by Redis, or alternatively use the BCAST mode that consumes no memory at all in the Redis side. Note that the memory consumed by Redis when BCAST is not used, is proportional both to the number of keys tracked, and the number of clients requesting such keys.
