CLOUD COMPUTING SYSTEMS

Lecture 4

Nuno Preguiça

(nuno.preguica_at_fct.unl.pt)

OUTLINE

Application cache at the data-center.

Content-distribution network.

OUTLINE

Application cache at the data-center.

Content-distribution network.

PROBLEM

Application servers fetch data from the database.

Potential problems?

- Slow… databases store data on disk.
- Cost… cloud platforms pay-per-use model: each access is charged.

Standard solution?

- Introduce a caching layer.

How to cache database results?

Cache data in the application server machine. Problems?

- The size of the cache is limited to the memory of each machine.
- Distributing the load means that the same data will be cached in multiple machines.

Can we share the cache among multiple machines?

Useful Latency Numbers (from Jeff Dean, Google)

Read 1 MB sequentially from memory	100,000 ns
Round trip within same datacenter	500,000 ns
Read 1 MB sequentially from SSD*	1,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns

SHARE CACHE AMONG MULTIPLE MACHINES

Advantages:

- Much larger cache size than caching only in the local memory.
- Pricing: cache access is cheaper than the database access at scale.

Disadvantages:

- Slower when compared to access to the local memory.

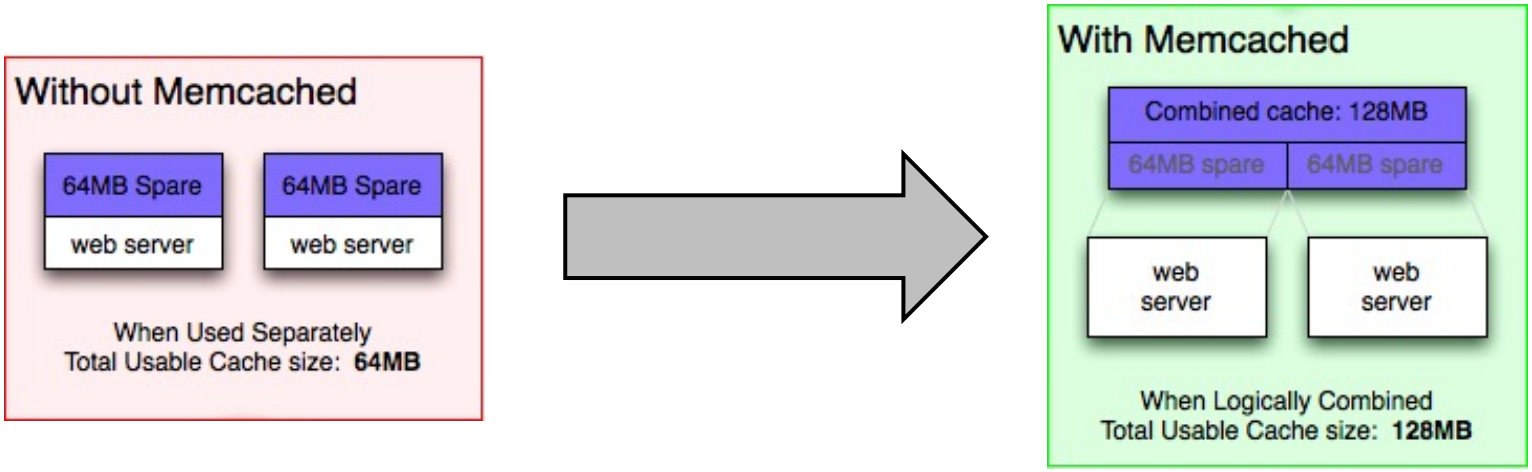


Image from: https://memcached.org/about

Common uses of (application server) caches

- Content Caching

  Store data that changes infrequently – e.g. page templates, data modified periodically (e.g. main page in a newspaper), etc.

  Reduces the processing time and server load.

- Cache-Aside

  Cache part of the database for faster access. Modify the cache when modifying the backend data.

  Reduces the server load.

- User session caching

  Store information associated with a user session in cache instead of (or in addition to) on the database. E.g. store info such as history, shopping carts, etc.

  Allows faster interaction; Client requests can be processed by any server.

TWO MAIN SOLUTIONS (CURRENTLY)

Memcached

- Basic distributed cache, where objects are treated (mostly) as BLOBs.

Redis

- Advanced data model, with support for types such as List, Set, etc.

TWO MAIN SOLUTIONS (CURRENTLY)

Memcached

- **Basic distributed cache, where objects are treated (mostly) as BLOBs.**

Redis

- Advanced data model, with support for types such as List, Set, etc.

MEMCACHED OVERVIEW

Distributed key-value store (hashtable)

Limited size – key,value pairs are discarded when cache is full

- Cache eviction policy: LRU (least-recently used).
- More details: cache divided in HOT, WARM and COLD; new items enter the HOT; cache eviction moves object to the lower level.

Designed for:

- High-throughput servers – accessing memory is much faster than disk.
- High-latency queries – avoid repeating costly queries.

MEMCACHED ARCHITECTURE

One-hop DHT:

- Clients know about all servers.
- Clients know the hash function to assign a key to a server. Can use consistent hashing.
- Clients send operations to the server that will hold a given key directly.
- Servers maintain a “key-value” store.

CHANGES IN THE APPLICATION

Why is the key userrow:user_id ? Because we will use the cache to stora all types of objecys and we cannot risk having the same id for different types of objects.

```
function get_foo(int userid)
    data = db_select(
          "SELECT * FROM users WHERE userid = ?",userid)
    return data
```

... should be modified to...

```
function get_foo(int userid)    /* first try the cache */
    data = memcached_fetch("userrow:" + userid)
    if not data          /* not found : request database */
        data = db_select(
            "SELECT * FROM users WHERE userid = ?", userid)
                /* then store in cache until next get */
        memcached_add("userrow:" + userid, data)
    return data
```

Example from Wikipedia.

CHANGES IN THE APPLICATION CODE (2)

On updates should update the cache.

```
function update_foo(int userid, string dbUpdateString)
                              /* first update database */
    result = db_execute(dbUpdateString)
    if result           /* database update successful : fetch
data to be stored in cache (id needed) */
        data = db_select(
            "SELECT * FROM users WHERE userid = ?", userid)
                /* then store in cache until next get */
        memcached_set("userrow:" + userid, data)
```

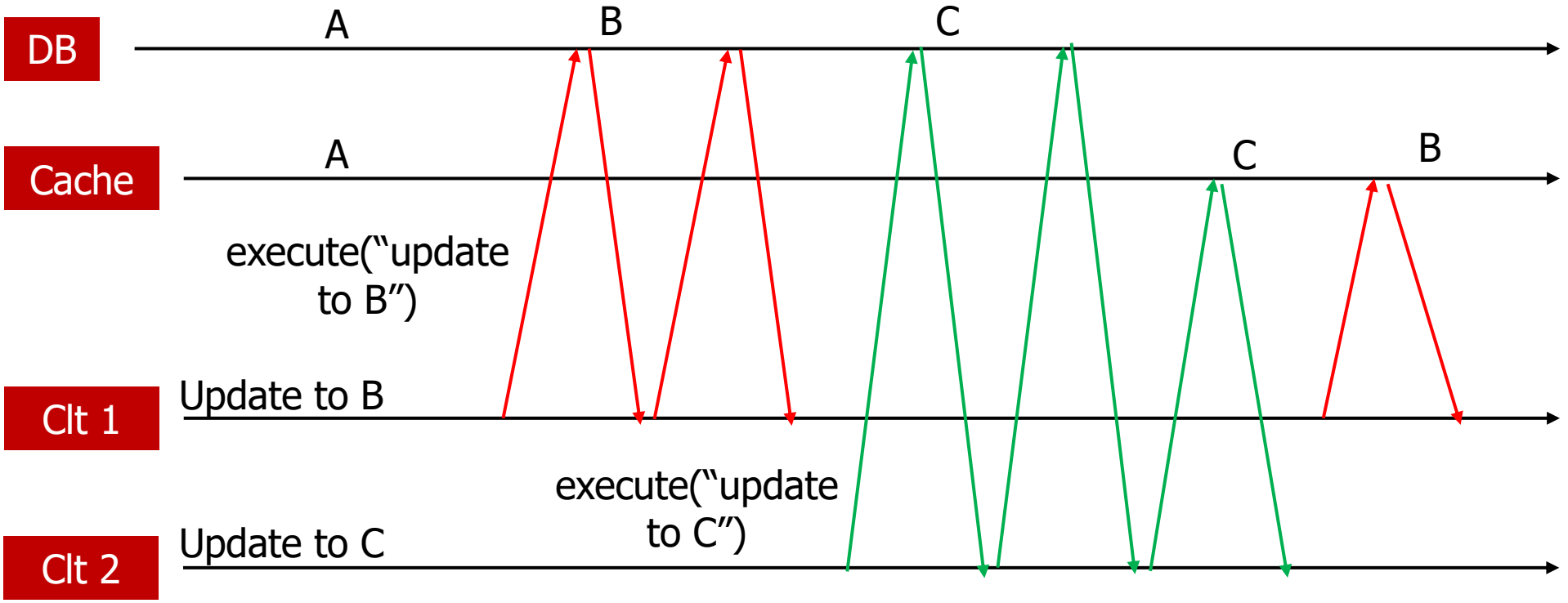
Why is it necessary to read from DB before setting the value in the database? Because on concurrent updates, the value stored in the database is undetermined.

Is this correct?
Not completely – see next run.

Example from Wikipedia.

CHANGES IN THE APPLICATION

How to solve?
No simple way. See next.




```
function update_foo(int userid, string dbUpdateString)
result = db_execute(dbUpdateString)
    if result
        data = db_select("SELECT * FROM users WHERE userid = ?", userid)
        memcached_set("userrow:" + userid, data)
```

Example from Wikipedia.

MEMCACHED LIMITATIONS: THUNDERING HERDS

On a cache miss, multiple clients may end up trying to set the value of a cache entry.

For a value updated very frequently, writes may be reordered and the old value get written in the cache.

Solution [from Facebook]:

- Clients get leases for writing the value of a key. This guarantees that a single client writes at a time.

More info at: R. Nishtala, et. al. Scaling Memcache at Facebook. NSDI'13.

ADDITIONAL FEATURES

Support for counter:

- Increment operation available to atomically increment an integer.

```
Memcache::increment ( string $key [, int $value = 1 ] ) : int
```

What can this be used for?

MEMCACHED LIMITATIONS: NO REPLICATION

Designed for volatile data

- Failure: Clients just go to disk.
- Recovery: Cache gets populated as consequence of the normal execution of clients.

Need redundancy?

- Need to create redundancy above Memcached.

TWO MAIN SOLUTIONS (CURRENTLY)

Memcached

- Basic distributed cache, where objects are treated (mostly) as BLOBs.

Redis

- **Advanced data model, with support for types such as List, Set, etc.**

REDIS OVERVIEW

Distributed key-value store (hashtable)

Values are types:

- List of strings (with insertion on head or tail)
- Set of strings
- Sorted set of strings
- Hashes (similar to a struct/map)
- Bit array
- HyperLogLogs (probabilistic data structure for estimating the number of elements in a set)

Each data type has a set of operations it supports.

REDIS OVERVIEW (2)

Redis can be used as a database, a cache or a message broker.

When used as a cache:

- it uses a limited amount of memory;
- it uses a LRU algorithm for cache eviction. Possible to control cache eviction by:
  - Set a TTL for eviction;
  - Define that only keys with a TTL can be evicted (making some entries persistent).

Redis supports transactions including a sequence of operations.

REDIS ARCHITECTURE

Each server maintains key-value pairs.

Each server executes operations in the values.

Possible to create a cluster of Redis servers, with data partitioned. Available partitioning strategies:

- Range partitioning;
- Hashing (some clients implement consistent hashing).
  - Tag hashing to control location of data – a tag is a prefix to the key; only the tag is hashed.
  - E.g. {CSS}key1 and {CSS}key2 are hashed to the same server, as only CSS is hashed.

Redis architecture (2)

Redis supports replication, but replication typically not used for caching.

Primary-backup (remember Dist. Sys. course), with asynchronous replication.

Multi-master replication with automatic conflict-resolution (based on CRDTs).

REDIS PERSISTENCE

Several alternatives:

- No persistence.
- RDB persistence: performs point-in-time snapshots of the database state at specified intervals.
- Append-only fashion: logs every write operation received by the server, that will be played again at server startup. Log compressed in background.
  - Durability depends on the parameters used for *fsync*. Remember the OS course.

REDIS OPERATIONS: EXAMPLES

Set

```
redis> SADD myset "Hello”
(integer) 1
redis> SADD myset "World”
(integer) 1
redis> SADD myset "World”
(integer) 0
redis> SMEMBERS myset
1) "Hello"
2) "World”
redis>
```

List

```
redis> LPUSH mylist "world”
(integer) 1
redis> LPUSH mylist "hello”
(integer) 2
redis> LRANGE mylist 0 –1
1) "hello"
2) "world”
redis>
```

EXAMPLES OF USE

Simple caching (as with Memcached).

Advanced caching functionalities, using data types support.

E.g.: maintain a leaderboard, maintain a list of recent topics, etc.

CACHING AND CONSISTENCY

Using a distributed caching poses consistency problems for data in the cache and in the database.

Need to control it in the application.

- When updating the database, it is necessary to update the cache also.
- What happens if multiple clients are modifying the database concurrently? How to guarantee that the cache is up-to-date with the database.

With CosmosDB and Redis, a possible solution is to use the timestamp associated with the document to guarantee that the latest version is kept in the cache. The value set should include both the key and the timestamp. When setting the value of the cache, the old value can be returned, allowing the client to check if it has overwritten a more recent value.

CLOUD PLATFORMS SUPPORT

Azure

- Redis.

AWS

- ElastiCache with support for Redis and Memcached.

Google

- Memcached.

OUTLINE

Application cache at the data-center.

Content-distribution network / content-delivery network.

CACHING CLOSER TO END-CLIENTS?

Caching in the servers:

1. does not reduce the latency incurred by client-server communication;
2. does not reduce the number of request arriving at the servers.

Can we cache data closer to the client?

Content Delivery Network (CDN)

A content delivery network (CDN) is a distributed network of edge servers that cache contents in point-of-presence (POP) locations that are close to end users, to minimize latency.

Google Cloud Cache locations @ google.

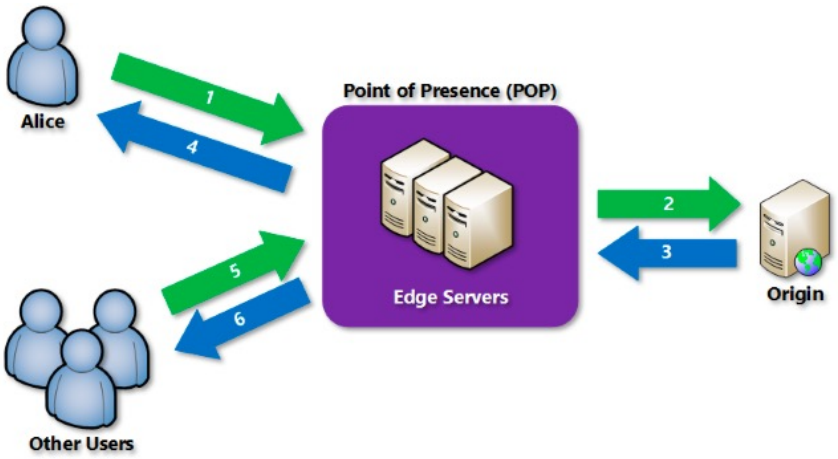
CDN BENEFITS

Lower latency for clients, especially for applications in which multiple round-trips are required to load content.

Large scaling to better handle instantaneous high loads, such as the start of a product launch event.

Reduce the traffic sent to the origin server, as requests are handled by the edge servers.

CDN: HOW IT WORKS



1: Request forwarded to the edge server (DNS returns server based on the client location).

2/3: If data is not in the cache, it is requested to the origin server.

4: The result is returned to the client. The data is cached according to a TTL.

Image and example from: https://docs.microsoft.com/en-us/azure/cdn/cdn-overview

CACHING RULES

Possible to define caching rules that affect:

- All requests;
- Request for given paths, extensions;
- Requests with given query string.

Possible caching rules include defining whether a page should be cached or not, and the TTL for caching.

ADVANCED CACHING FEATURES

Route Optimization

Route optimization chooses the most optimal path to the origin to guarantee that content is delivered to end users via the fastest and most reliable route possible.

TCP Optimizations

Several optimizations to TCP to speedup communication.

Eliminating TCP slow start.

Leveraging persistent connections.

Tuning TCP packet parameters.

ADVANCED CACHING FEATURES (2)

Object prefetch

Prefetch consists in retrieving images and scripts embedded in the HTML page while the HTML is served to the browser, and before the browser even makes these object requests.

When the client makes the requests for the linked assets, the CDN edge server already has the requested objects and can serve them immediately without a round trip to the origin

Adaptive image compression

This feature automatically monitors network quality, and employs standard JPEG compression methods when network speeds are slower to improve delivery time (e.g. for mobile users).

ADVANCED CACHING FEATURES (3)

Zone-based restriction

Allows to restrict access to content by country/region.
With *geo-filtering*, it is possible to create rules on specific paths on the CDN endpoint to allow or block content in selected countries/regions.

DDoS protection

A content delivery network provides DDoS protection by design, by being able to absorb volumetric attacks. CDN also include always-on traffic monitoring, and real-time mitigation of common network-level attacks.

To know more:

https://github.com/memcached/memcached/wiki/Overview

https://github.com/memcached/memcached/wiki/Programming

https://redis.io/topics/data-types-intro (too detailed)

https://docs.microsoft.com/en-us/azure/cdn/cdn-overview

https://docs.microsoft.com/en-us/azure/cdn/cdn-how-caching-works

https://docs.microsoft.com/pt-pt/azure/cdn/cdn-dynamic-site-acceleration

ACKNOWLEDGMENTS

Some text and images from Microsoft Azure online documentation.

PROJECT - HOW ALL THIS FITS TOGETHER FOR THE PROJECT?