POLITECNICO
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

Key-value Databases – Redis

Marco Brambilla

marco.brambilla@polimi.it

@marcobrambi

Overview – NoSQL Family

Data stored in 4 types:

- Document
- Graph
- Key-value
- Wide-column

Document Database	Graph Databases
 Couchbase  	 
Key-value Databases	Wide Column Stores
   	     

Why is performance important?

- Amazon - Every 1/10 second delay resulted in 1% loss of sales.
- Google - Half a second delay caused a 20% drop in traffic.
- Industrial Group - 1-second delay in page-load time
- 11% fewer page views
- 16% decrease in customer satisfaction
- 7% loss in conversions

Key-Value databases

Why Key-value Store?

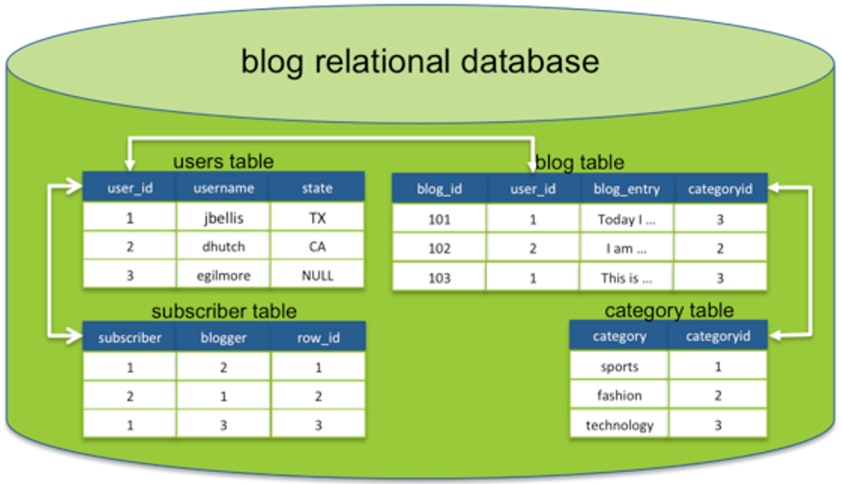
- (Business) Key -> Value

- (twitter.com) tweet id -> information about tweet
- (kayak.com) Flight number -> information about flight
- (yourbank.com) Account number -> information about it
- (amazon.com) item number -> information about it

- Search by ID is usually built on top of a key-value store

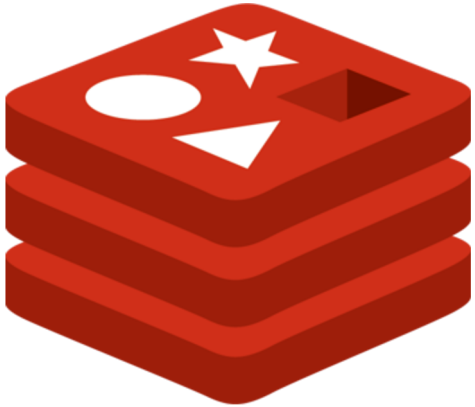
Isn’t that just a database?

- Queried using SQL
- Key-based
- Foreign keys
- Indexes
- Joins



SQL queries: SELECT user_id from users WHERE username = “jbellis”

Example’s Source



redis

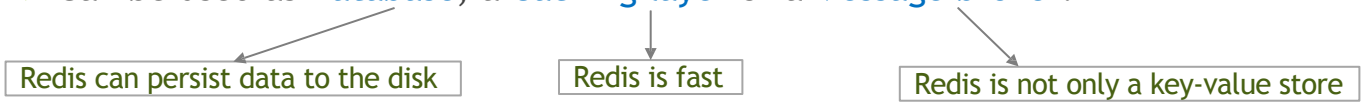
REmote **DI**ctionary **S**erver

What is REDIS ?

- Redis is an advanced **key-value store**, where keys can contain data structures such as strings, hashes, lists, sets, and sorted sets. Supporting a set of **atomic operations** on these data types.

- Redis is a different evolution path in the key-value databases where values are complex data types that are closely related to fundamental data structures and are exposed to the programmer as such, without additional abstraction layers.

- Can be used as Database, a Caching layer or a Message broker.



What is **not** REDIS

- Redis is not a replacement for Relational Databases nor Document Stores.
- It might be used complementary to a SQL relational store, and/or NoSQL document store.
- Even when Redis offers configurable mechanisms for persistency, increased persistency will tend to increase latency and decrease throughput.
- ***Best used*** *for rapidly changing data with a foreseeable database size (should fit mostly in memory).*

NoSQL comparisons:
http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis
http://www.infoivy.com/2013/07/nosql-database-comparison-chart-only.html

Redis Use Cases

- Caching
- Counting things
- Blocking queues
- Pub/Sub (service bus)
- MVC Output Cache provider
- Backplane for SignalR
- ASP.NET Session State provider*
- Online user data (shopping cart, …
- … Any real-time, cross-platform, cross-application communication

* ASP.NET session state providers comparison: http://www.slideshare.net/devopsguys/best-performing-aspnet-session-state-

When to consider Redis

- Speed is critical
- More than just key-value pairs
- Dataset can fit in memory
- Dataset is not critical

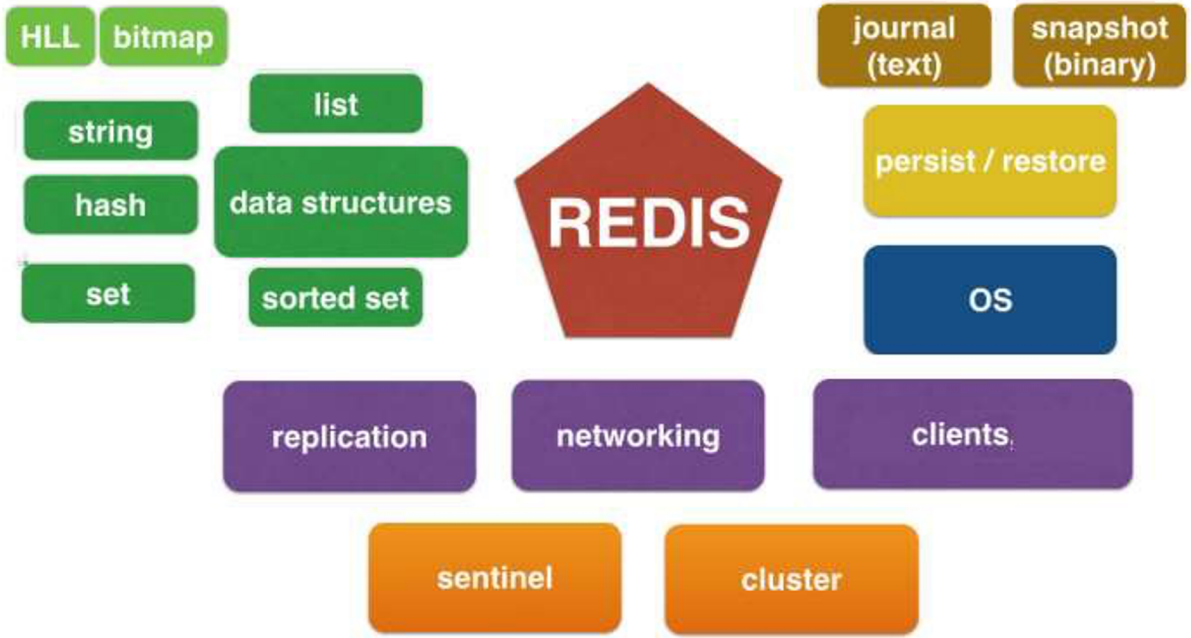
From: http://www.slideshare.net/autonomous/redis-overview-for-software-architecture-

Redis Architecture

- Written in ANSI C by Salvatore Sanfilippo (@antirez).
- Works in most POSIX systems like Linux, BSD and OS X.
- Linux is the recommended
- No official support for Windows, but Microsoft develops and maintains an open source Win-64 port of Redis*
- Redis is a single-threaded server, not designed to benefit from multiple CPU cores.
  Several Redis instances can be launched to scale out on several cores.
- All operations are atomic (no two commands can run at the same time).
- It executes most commands in O(1) complexity and with minimal lines of code.

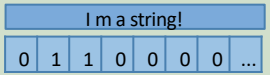
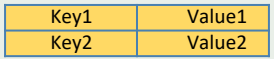
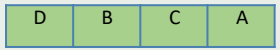
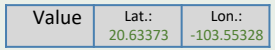
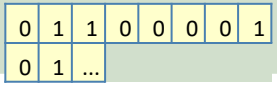
*Redis on Windows:
https://github.com/MSOpenTech/redis

Redis Architecture



*from https://matt.sh/redis-architecture-diagram

Redis data types

Redis Data Type		Contains	Read/write ability
String		Binary-safe strings (up to 512 MB), Integers or Floating point values, Bitmaps.	Operate on the whole string, parts, increment/decrement the integers and floats, get/set bits by position.
Hash		Unordered hash table of keys to string values	Add, fetch, or remove individual ítems by key, fetch the whole hash.
List		Doubly linked list of strings	Push or pop items from both ends, trim based on offsets, read individual or multiple items, find or remove items by value.
Set		Unordered collection of unique strings	Add, fetch, or remove individual items, check membership, intersect, union, difference, fetch random items.
Sorted Set		Ordered mapping of string members to floating-point scores, ordered by score	Add, fetch, or remove individual items, fetch items based on score ranges or member value.
Geospatial index		Sorted set implementation using geospatial information as the score	Add, fetch or remove individual items, search by coordinates and radius, calculate distance.
HyperLogLog		Probabilistic data structure to count unique things using 12Kb of memory	Add individual or multiple items, get the cardinality.

*Redis data types internals: https://cs.brown.edu/courses/cs227/archives/2011/slides/mar07-redis.pdf

Redis Commands - Basic

Strings

Get/Set strings

redis> SET foo "hello!"

OK

redis> GET foo

"hello!"

SET [key value] / GET [key] **O(1)**

Increment numbers

redis> SET bar 223

OK

redis> INCRBY bar 1000 (integer) 1223

INCRBY [key increment] **O(1)**

Get multiple keys at once

redis> MGET foo bar

1. "hello!" 2. "1223"

MGET [key key ...] **O(N) : N=# of keys.**

Set multiple keys at once

> MSET foo "hello!" bar 1223

OK

MSET [key value key value ...]
O(N) : N=# of keys.

Get the length of a string

redis> STRLEN foo (integer) 6

STRLEN [key] **O(1)**

Update a value retrieving the old one

redis> GETSET foo "bye!"
redis> GET foo "bye!"

GETSET [key value] **O(1)**

POLITECNICO
MILANO 1863

Redis Commands - Basic

Keys	
Key removal *redis> DEL foo (integer) 1*	DEL [key ...] O(1)
Test for existence *redis> EXISTS foo (integer) 1*	EXISTS [key ...] O(1)
Get the type of a key *redis> TYPE foo string*	TYPE [key] O(1)
Rename a key *redis> RENAME bar new_bar* *OK*	RENAME [key newkey] O(1)
redis> EXPIRE foo 10 (integer) 1	O(1)
Get key time-to-live *redis> TTL foo (integer)* *10*	TTL [key] O(1)

Redis Commands - Lists & Hashes

Lists

Push on either end
RPUSH/LPUSH [key value] O(1)

redis> RPUSH jobs "foo"
(integer) 1
redis> LPUSH jobs "bar"
(integer) 1

Pop from either end
RPOP/LPOP [key] O(1)

redis> RPOP jobs
"foo"
redis> LPOP jobs
"bar"

Blocking Pop
BRPOP/BLPOP [key] O(1)

redis> BLPOP jobs
redis> BRPOP jobs

Pop and Push to another list
RPOPLPUSH [src dst] O(1)

Get an element by index
LINDEX [key index] O(N)

redis> LINDEX jobs 1
"foo"

Get a range of elements
LRANGE [key start stop] O(N)

redis> LRANGE jobs 0 -1
1. *"bar"*
2. *"foo"*

Hashes

Set a hashed value
HSET [key field value] O(1)

redis> HSET user:1 name John (integer) 1

Set multiple fields
HMSET [key field value ...] O(1)

redis> HMSET user:1 lastname Smith visits 1 OK

Get a hashed value
HGET [key field] O(1)

redis> HGET user:1 name "John"

Get all the values in a hash
HGETALL [key] O(N) : N=size of hash.

redis> HGETALL user:1
1)"name"
2) "John"
3) "lastname"
4) "Smith"
5)"visits" 6) "1"

Increment a hashed value
HINCRBY [key field incr] O(1)

redis> HINCRBY user:1 visits 1
(integer) 2

Redis Commands - Sets & Sorted sets

Sets

Add member to a set — SADD [key member ...] O(1)

```
redis> SADD admins "Peter"
(integer) 1
redis> SADD users "John" "Peter"
(integer) 2
```

Pop a random element — SPOP [key] O(1)

```
redis>
SPOP
users
"John"
```

Get all elements — SMEMBERS [key] O(N) : N=size of set.

```
redis> SMEMBERS users
1) "Peter"
2) "John"
```

Union multiple sets — SUNION [key key ...] O(N)

```
redis> SUNION users admins
1) "Peter"
2) "John"
```

Diff. multiple sets — DIFF [key key ...] O(N)

```
redis> SDIFF users admins
1) "John"
```

Sorted sets

Add member to a sorted set — ZADD [key score member] O(log(N))

```
redis> ZADD scores 100 "John"
(integer) 1
redis> ZADD scores 50 "Peter" 200 "Charles" 1000 "Mary"
(integer) 3
```

Get the rank of a member — ZRANK [key member] O(log(N))

```
redis> ZRANK scores "Mary" (integer) 3
```

Get elements by score range — ZRANGEBYSCORE [key min max] O(log(N))

```
redis> ZRANGEBYSCORE scores 200 +inf WITHSCORES
1) "Charles" 2) 200
3) "Mary" 4) 1000
```

Increment score of member — ZINCRBY [key incr member] O(log(N))

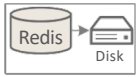
```
redis> ZINCRBY scores 10 "Mary" "1010"
```

Remove range by score — ZREMRANGEBYSCORE [key min max] O(log(N))

```
redis> ZREMRANGEBYSCORE scores 0 100
(integer) 2
```

Scaling Redis

Persistence



Redis provides two mechanisms to deal with persistence: Redis database snapshots (RDB) and append-only files (AOF).

Replication

A Redis instance known as the *master*, ensures that one or more instances kwown as the *slaves* become exact copies of the master. Clients can connect to the master or to the slaves. Slaves are read only by default.

Partitioning

Breaking up data and distributing it across different hosts in a cluster.

Can be implemented in different layers:

- **Client:** Partitioning on client-side code.
- **Proxy:** An extra layer that proxies all redis queries and performs partitioning (i.e. Twemproxy).
- **Query Router:** instances will make sure to forward the query to the right node. (i.e Redis Cluster).

Failover

- Manual
- Automatic with Redis Sentinel (for master-slave topology)
- Automatic with Redis Cluster (for cluster topology)

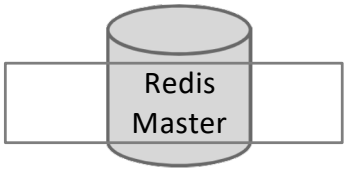
Redis topologies

I. Standalone
II. Sentinel (automatic failover)
III. Twemproxy (distribute data)
IV. Cluster (automatic failover and distribute data)

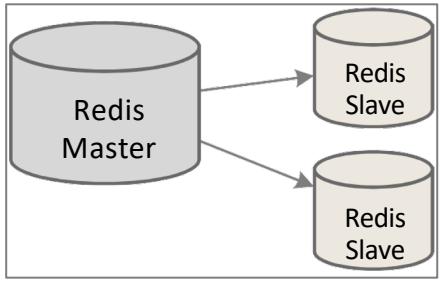
Redis topologies I - Standalone

- The master data is optionally replicated to slaves.
- The slaves provides data redundancy, reads offloading and save-to-disk offloading.
- Clients can connect to the Master for read/write operations or to the Slaves for read operations.
- Slaves can also replicate to its own slaves.
- There is no automatic failover.

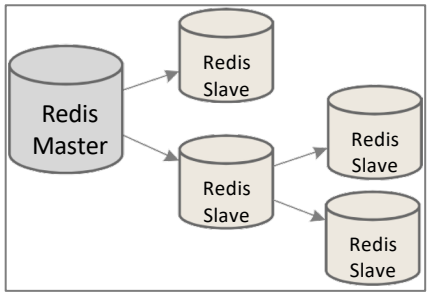
Master only



Master-slave



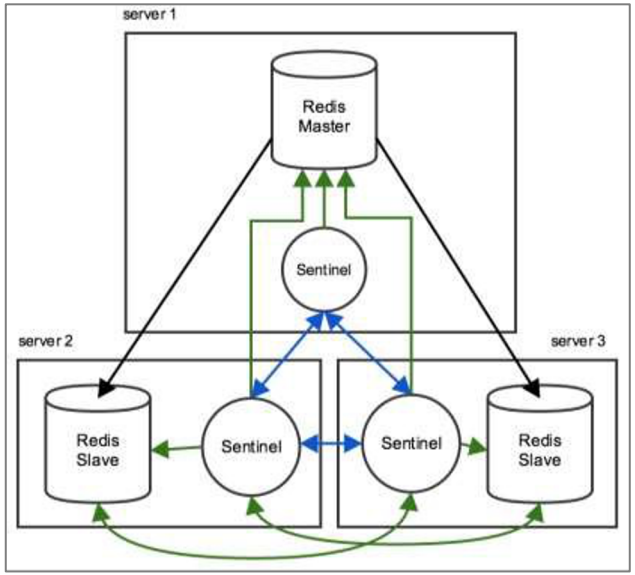
Master-slave multi-level



Redis topologies II - Sentinel

- Redis Sentinel provides a reliable **automatic failover** in a master/slave topology, automatically promoting a slave to master if the existing master fails.
- Sentinel does not distribute data across nodes.

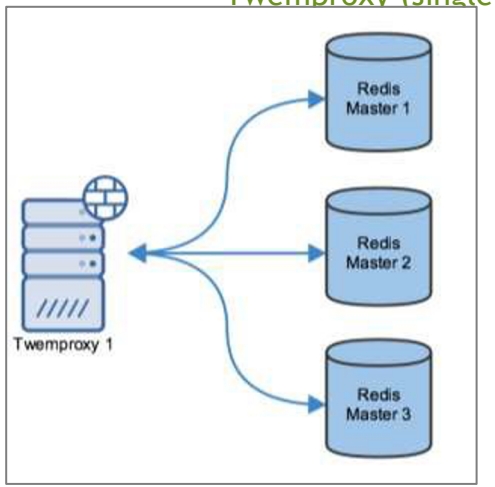
Master-slave with Sentinel



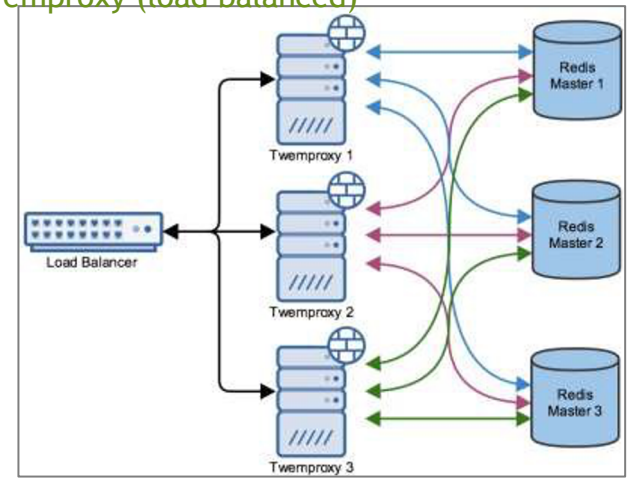
Redis topologies III - Twemproxy

- Twemproxy* works as a proxy between the clients and many Redis instances.
- Is able to **automatically distribute data** among different standalone Redis instances.
- Supports consistent hashing with different strategies and hashing functions.
- Multi-key commands and transactions are not supported.

Twemproxy (single)



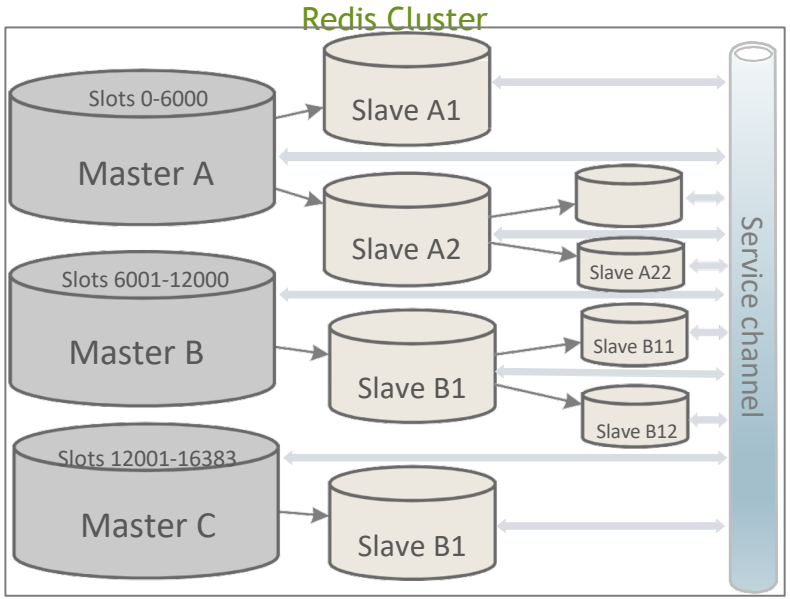
Twemproxy (load balanced)



*Twemproxy is a project from Twitter and is not part of redis:
https://github.com/twitter/twemproxy

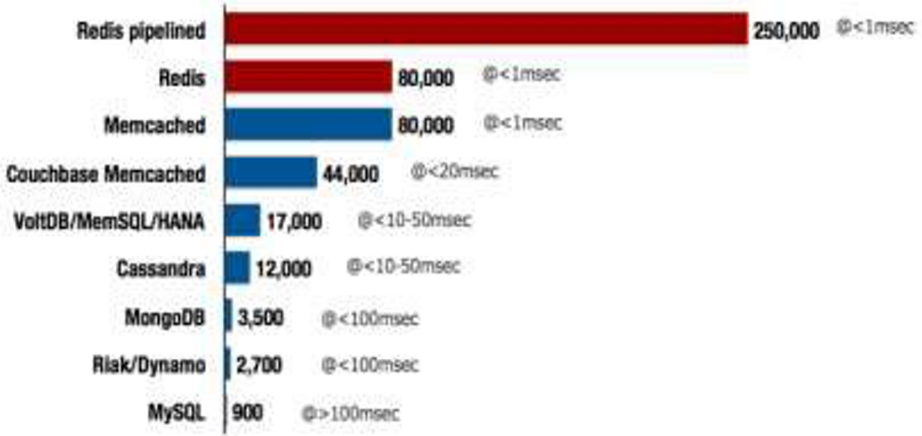
Redis topologies IV - Cluster

- Redis Cluster **distributes data** across different Redis instances and **perform automatic failover** if any problem happens to any master instance.
- All nodes are directly connected with a service channel.
- The keyspace is divided into hash slots. Different nodes will hold a subset of hash slots.
- Multi-key commands are only allowed for keys in the same hash slot.



Redis advantages

- Performance
- Availability
- Fault-Tolerance
- Scalability (adaptability)
- Portability



NoSQL & SQL response performance comparison
(from https://redislabs.com/blog/the-proven-redis-performance)

- Support for complex data types and structures
- Atomic operations and Transactions
- Pipelining (send multiple commands at once)
- LUA scripting support
- LRU eviction of keys
- Keys with limited time-to-live
- Simple to install, setup and manage
- Highly configurable
- Supported on many languages
- Straightforward and well documented API
- Open Source
- Available on Azure

Key-value and Caching

What is caching?

- From Wikipedia: "A cache is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (owing to longer access time) or to compute, compared to the cost of reading the cache."
- Term introducted by IBM in the 60's

The anatomy

- simple key/value storage
- simple operations
  - save
  - get
  - delete

Terminology

- storage cost
- retrieval cost (network load / algorithm load)
- invalidation (keeping data up to date / removing irrelevant data)
- replacement policy (FIFO/LFU/LRU/MRU/RANDOM vs. Belady's algorithm)
- cold cache / warm cache

Terminology

- cache hit and cache miss
- typical stats:
  - hit ratio (hits / hits + misses)
  - miss ratio (1 – hit ratio)

  - 45 cache hits and 10 cache misses
    - 45/(45+10) = 82% hit ratio
    - 18% miss ratio

When to cache?

- caches are only efficient when the benefits of faster access outweigh the overhead of checking and keeping your cache up to date
  - more cache hits then cache misses

Where are caches used?

- at hardware level (cpu, hdd)
- operating systems (ram)
- web stack
- applications
- your own short term vs long term memory

Caches in the web stack

- Browser cache
- DNS cache
- Content Delivery Networks (CDN)
- Proxy servers
- Application level
  - full output caching (eg. Wordpress WP-Cache plugin
)
  - …

Efficiency of caching?

- the earlier in the process, the closer to the original request(er), the faster
  - browser cache will be faster than cache on a proxy
- but probably also the harder to **get it right**
  - the closer to the requester the more parameters the cache depends on

Memcached

About memcached

- Free & open source, high-performance, distributed memory object caching system
- Generic in nature, intended for use in speeding up dynamic web applications by alleviating database load.
- key/value dictionary

About memcached

- Developed by Brad Fitzpatrick for LiveJournal in 2003
- Now used by Netlog, Facebook, Flickr, Wikipedia, Twitter, YouTube ...

Technically

- It's a server
- Client access over TCP or UDP
- Servers can run in pools
  - eg. 3 servers with 64GB mem each give you a single pool of 192GB storage for caching
  - Servers are independent, clients manage the pool

What to store in memcache?

- **high demand** (used often)
- **expensive** (hard to compute)
- **common** (shared accross users)
- **Best: All three**

What to store in memcache? (cont'd)

- Typical:
  - user sessions (often)
  - user data (often, shared)
  - homepage data (eg. often, shared, expensive)

Memcached principles

- **Fast network access** (memcached servers close to other application servers)
- **No persistency** (if your server goes down, data in memcached is gone)
- **No redundancy / fail-over**
- **No replication** (single item in cache lives on one server only)
- **No authentication** (not in shared environments)

Memcached principles (cont'd)

- 1 key is maximum 1MB
- keys are strings of 250 characters (in application typically MD5 of user readable string)
- No enumeration of keys (thus no list of valid keys in cache at certain moment)
- No active clean-up (only clean up when more space needed, LRU: Least Recently Used)

PHP Client functions

Memcached::add — Add an item under a new key
Memcached::addServer — Add a server to the server pool

Memcached::decrement — Decrement numeric item's value
Memcached::delete — Delete an item
Memcached::flush — Invalidate all items in the cache
Memcached::get — Retrieve an item
Memcached::getMulti — Retrieve multiple items
Memcached::getStats — Get server pool statistics

Memcached::increment — Increment numeric item's value
Memcached::set — Store an item

```php
<?php

$html    = $cache->get('mypage');
if (!$html)
{
  ob_start();
  echo "<html>";
    //              all the          stuff      goes   here
    fancy
    echo "</html>";
    $html =     ob_get_contents();
    ob_end_clean();
    $cache->set('mypage',          $html);
}
echo   $html;

?>
```

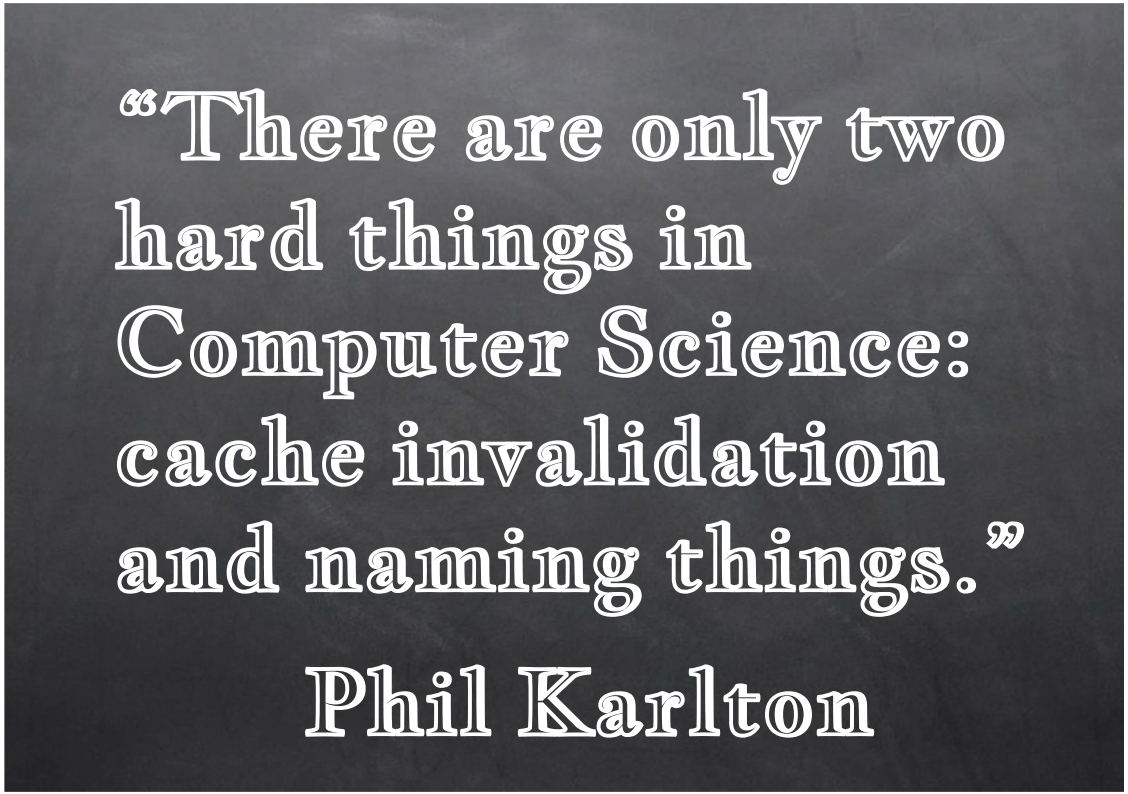
Data caching

- on a lower level
- easier to find all dependencies
- ideal solution for offloading database queries
  - the database is almost always the biggest bottleneck in backend performance problems

```php
<?php

function getUserData($UID)
{
    $key = 'user_' . $UID;
    $userData = $cache->get($key);
    if (!$userData)
    {
        $queryResult = Database::query("SELECT * FROM USERS
WHERE uid = " . (int) $UID);
        $userData = $queryResult->getRow();
        $cache->set($userData);
    }
    return $userData;
}

?>
```


“There are only two hard things in Computer Science: cache invalidation and naming things.”
Phil Karlton

Invalidation

- Caching for a certain amount of time
  - eg. 10 minutes
  - don’t delete caches
  - thus: You can’t trust that data coming from cache is correct

Invalidation (cont'd)

- Use: Great for summaries
  - Overview
  - Pages where it's not that big a problem if data is a little bit out of date (eg. search results)
- Good for quick and dirty optimizations

Invalidation (cont'd)

- Store forever, and expire on certain events
  - the userdata example
    - store userdata for ever
    - when user changes any of his preferences, throw cache away

```php
<?php
function getUserData($UID, $invalidateCache = false)
{
    $db = DB::getInstance();
    $db->prepare("SELECT *
                  FROM
                  USERS
                  WHERE uid = {UID}");
    $db->assignInt("UID", $UID);
    $db->setCacheTTL(0); // if cache forever
    ($invalidateCache)
    {
        return $db->invalidateCache();
    }
    $db->execute(); return $db-
    >getRow();
}
?>
```

```php
<?php
function updateUserData($UID, $data)
{
    $db = DB::getInstance();
    $db->prepare("UPDATE USERS
                  SET ...
                  WHERE uid = {UID}");

    ...

    getUserData($UID, true); // cache invalid
                             ate
    return $result;
}
?>
```

Multi-Get Optimisations

- We reduced database access
- Memcached is faster, but access to memcache still has it's price
- Solution: multiget
  - fetch multiple keys from memcached in one single call
  - result is array of items

More tips ...

- **Be carefull when security matters.** (Remember ‘no authentication’?)
  - Working on authentication for memcached via SASL Auth Protocol
- **Caching is not an excuse not to do database tuning.** (Remember cold cache?)
- **Make sure to write unit tests for your caching classes and places where you use it.** (Debugging problems related to out-of-date cache data is hard and boring. Very boring.)

POLITECNICO
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

Key-value Databases – Redis

Marco Brambilla

marco.brambilla@polimi.it

@marcobrambi