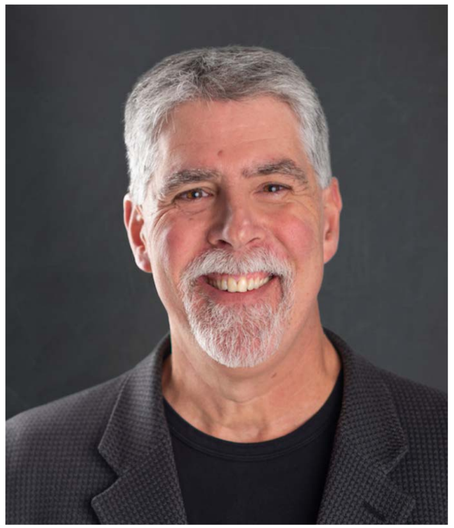
O'REILLY®

LIVE ONLINE TRAINING

Microservices Caching Strategies



Mark Richards

Independent Consultant

Hands-on Software Architect / Published Author

Founder, DeveloperToArchitect.com

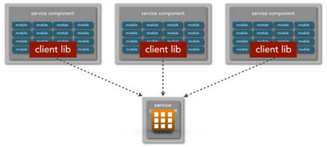
http://www.wmrichards.com

https://www.linkedin.com/in/markrichards3

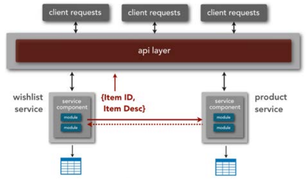
@markrichardssa

v5a

agenda

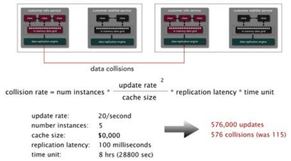


caching
topologies



data
sharing

space-based
microservices

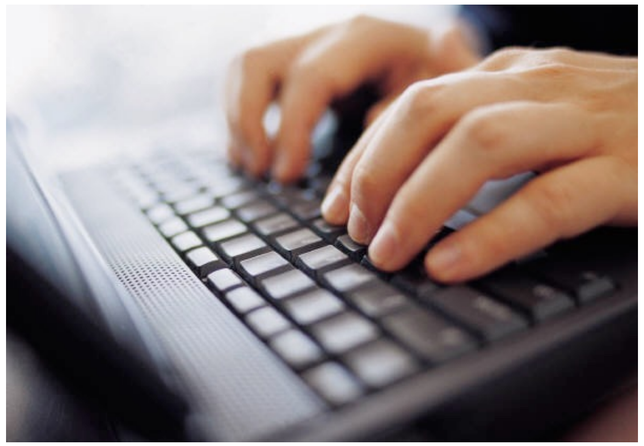


data
collisions

cache eviction
policies

Caching Topologies

caching topologies

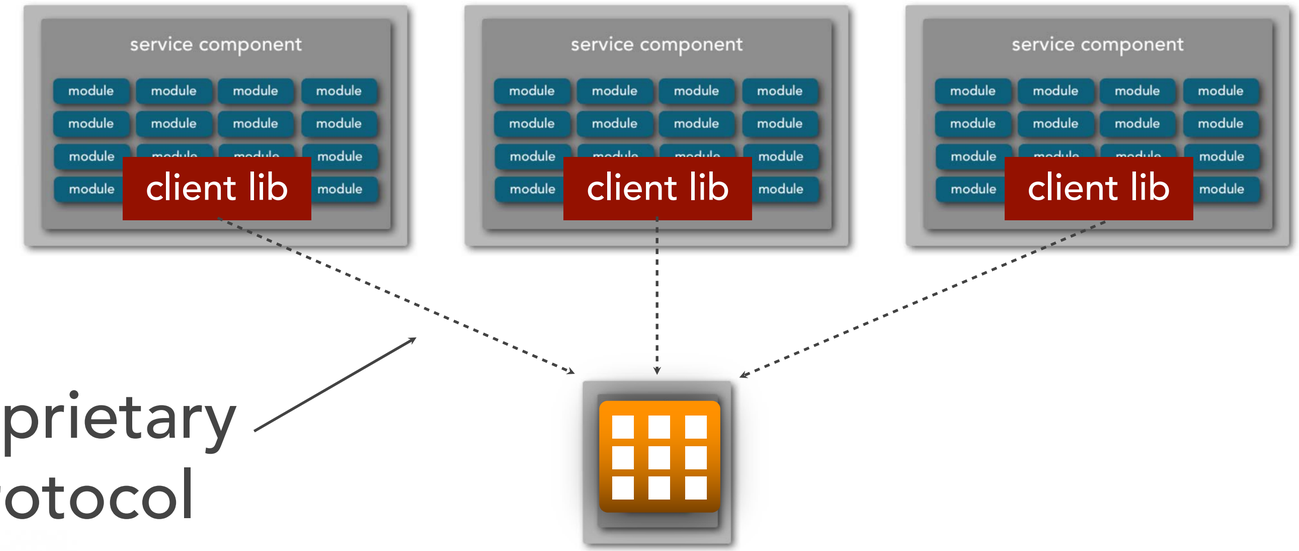


https://github.com/wmr513/caching/tree/master/src/main/java/ignite

GitHub

caching topologies

distributed (client-server) cache



apache Ignite

Couchbase

EHCACHE

NCache™
Distributed Cache for .NET

hazelcast

MEMCACHED

redis

Pivotal
GemFire®

ORACLE®
Coherence

caching topologies

distributed (client-server) cache

```
$ ./ignite.sh
[16:34:38]    __________  ________________
[16:34:38]   /  _/ ___/ |/ /  _/_  __/ __/
[16:34:38]  _/ // (7 7    // /  / / / _/
[16:34:38] /___/\___/_/|_/___/ /_/ /___/
[16:34:38]
[16:34:38] ver. 2.7.0#20181130-sha1:256ae401
[16:34:38] 2018 Copyright(C) Apache Software Foundation

[16:34:41] Ignite node started OK (id=5387041b)
[16:34:41] Topology snapshot [ver=1, locNode=5387041b, servers=1, clients=0, state=ACTIVE,
CPUs=4, offheap=3.2GB, heap=1.0GB]
```

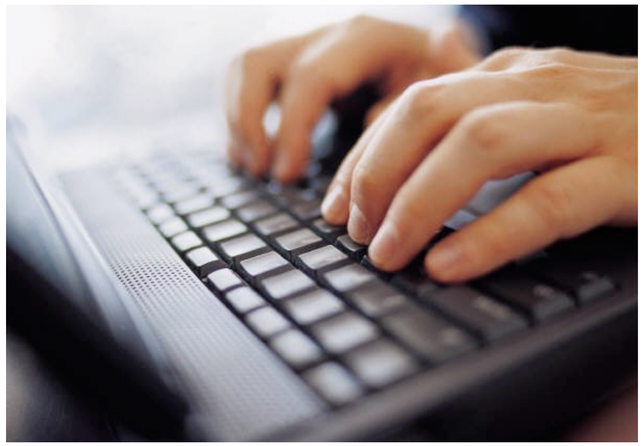
caching topologies

distributed (client-server) cache

```
ClientConfiguration cfg = new ClientConfiguration().setAddresses("127.0.0.1:10800");

IgniteClient ignite = Ignition.startClient(cfg);

ClientCache<String, String> cache = ignite.getOrCreateCache("names");

String customerId = getCustomerIdFromRequest();
String newName = getNameFromRequest();
String currentName = cache.get(customerId);
if (currentName == null) { currentName = getNameFromDatabase(customerId); }
updateDatabase(customerId, newName);
cache.put(customerId, newName);
```

cache is on server, not stored locally in memory

caching topologies

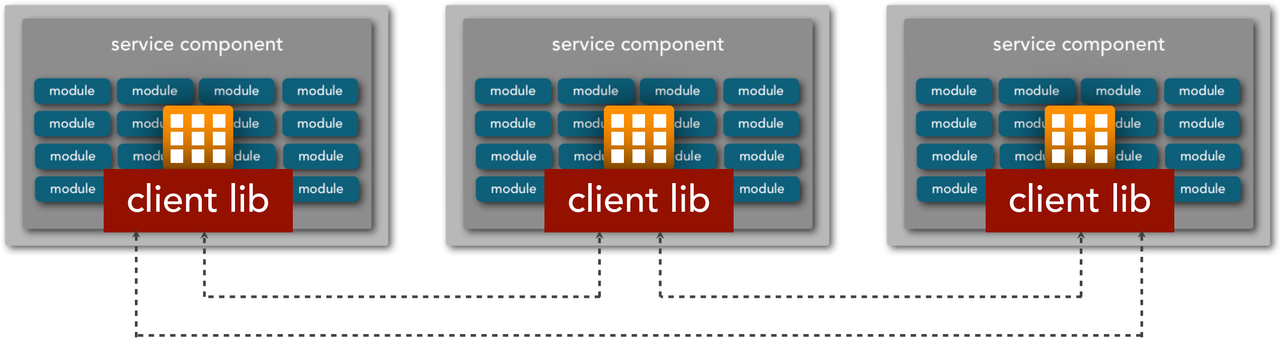


let’s see this in action!

https://github.com/wmr513/caching
(tree/master/src/main/java/ignite/DistributedTest.java)

caching topologies

replicated (in-process) cache



proprietary protocol

apache Ignite

hazelcast

Pivotal GemFire®

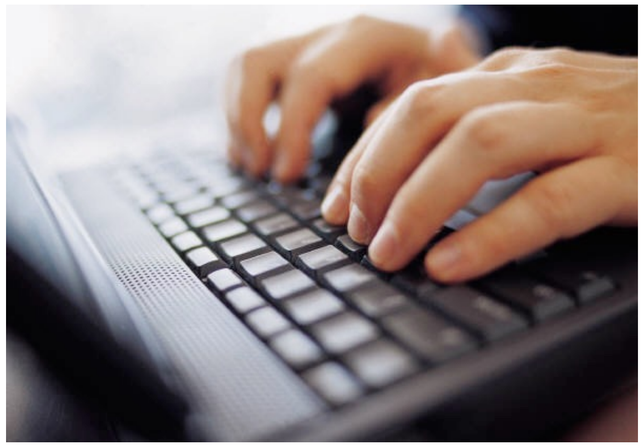
ORACLE® Coherence

caching topologies

replicated (in-process) cache

```
CacheConfiguration cfg = new CacheConfiguration("names");
cfg.setCacheMode(CacheMode.REPLICATED);

IgniteConfiguration ic = new IgniteConfiguration();
ic.setClientMode(false);

Ignite ignite = Ignition.start(ic);
IgniteCache<String, String> cache = ignite.getOrCreateCache(cfg);

String customerId = getCustomerIdFromRequest();
String newName = getNameFromRequest();
String currentName = cache.get(customerId);
if (currentName == null) { currentName = getNameFromDatabase(customerId); }
updateDatabase(customerId, newName);
cache.put(customerId, newName); <-------- all other replicas are synced automatically here...
```

caching topologies

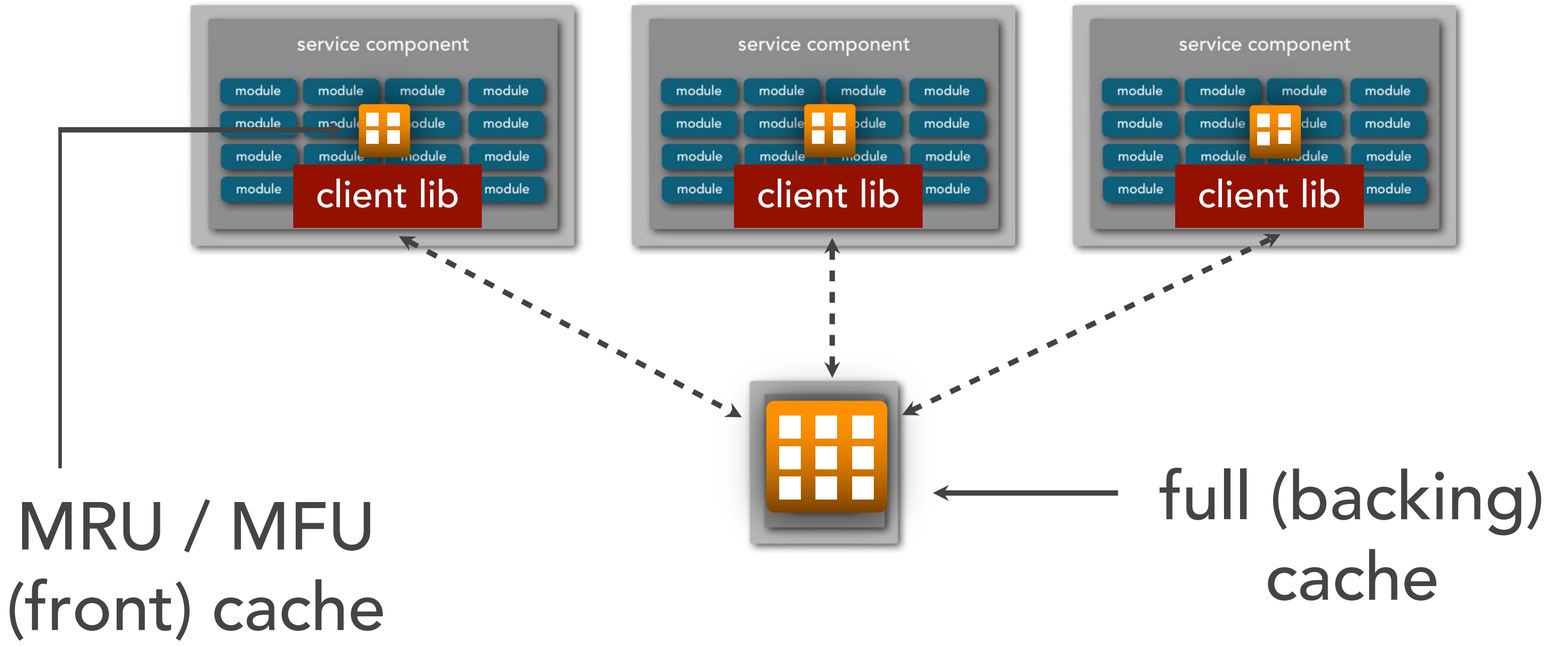


let’s see this in action!

https://github.com/wmr513/caching
(tree/master/src/main/java/ignite/ReplicatedTest1.java)
(tree/master/src/main/java/ignite/ReplicatedTest2.java)

caching topologies

near-cache hybrid



apache Ignite

hazelcast

Pivotal GemFire®

ORACLE® Coherence

caching topologies

near-cache hybrid

```
$ ./ignite.sh
[16:34:38]    __________  ________________
[16:34:38]   /  _/ ___/ |/ /  _/_  __/ __/
[16:34:38]  _/ // (7 7    // /  / / / _/
[16:34:38] /___/\___/_/|_/___/ /_/ /___/
[16:34:38]
[16:34:38] ver. 2.7.0#20181130-sha1:256ae401
[16:34:38] 2018 Copyright(C) Apache Software Foundation

[16:34:41] Ignite node started OK (id=5387041b)
[16:34:41] Topology snapshot [ver=1, locNode=5387041b, servers=1, clients=0, state=ACTIVE,
CPUs=4, offheap=3.2GB, heap=1.0GB]
```

caching topologies

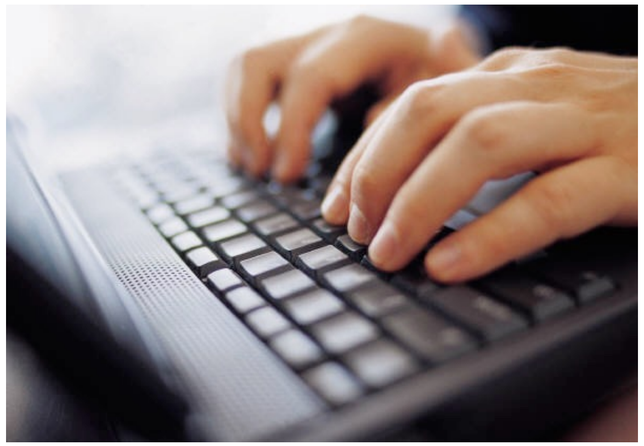
near-cache hybrid

```
CacheConfiguration serverCfg = new CacheConfiguration("names");

IgniteConfiguration ic = new IgniteConfiguration();
ic.setClientMode(true);

NearCacheConfiguration<String, String> nearCfg = new NearCacheConfiguration<>();
nearCfg.setNearEvictionPolicyFactory(new LruEvictionPolicyFactory<>(20000));

Ignite ignite = Ignition.start(ic);
IgniteCache<String, String> cache = ignite.getOrCreateCache(serverCfg, nearCfg);

String customerId = getCustomerIdFromRequest();
String newName = getNameFromRequest();
String currentName = cache.get(customerId);
if (currentName == null) { currentName = getNameFromDatabase(customerId); }
updateDatabase(customerId, newName);
cache.put(customerId, newName);
```

near cache will be searched first, then server cache

added to near cache, eviction if no room, then added to server cache

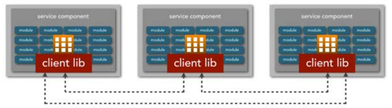
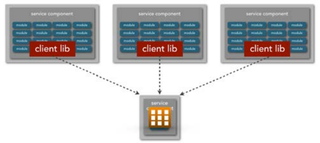
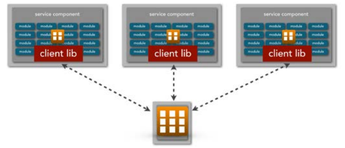
caching topologies



let’s see this in action!

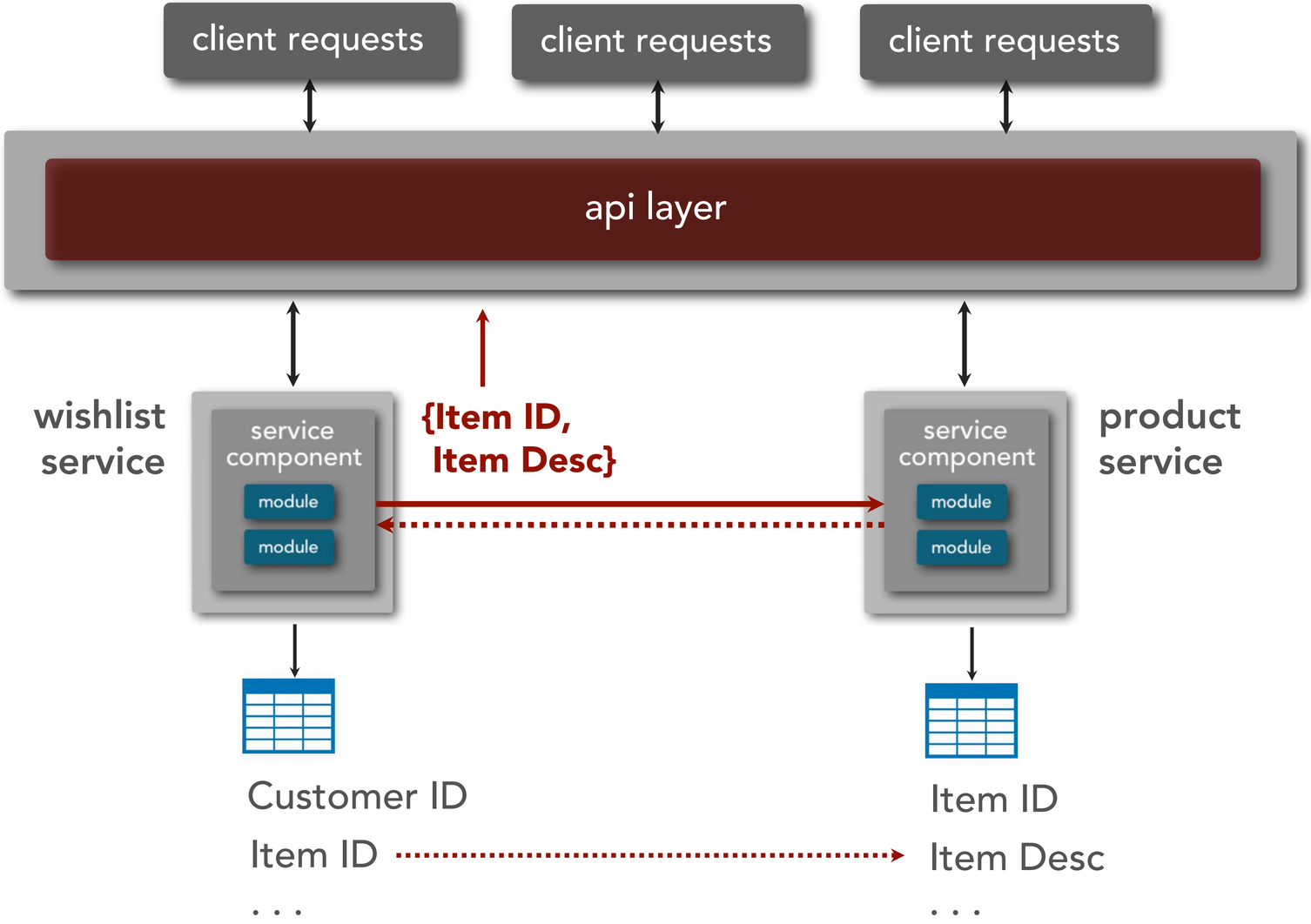
https://github.com/wmr513/caching
(tree/master/src/main/java/ignite/NearTest.java)

caching topologies

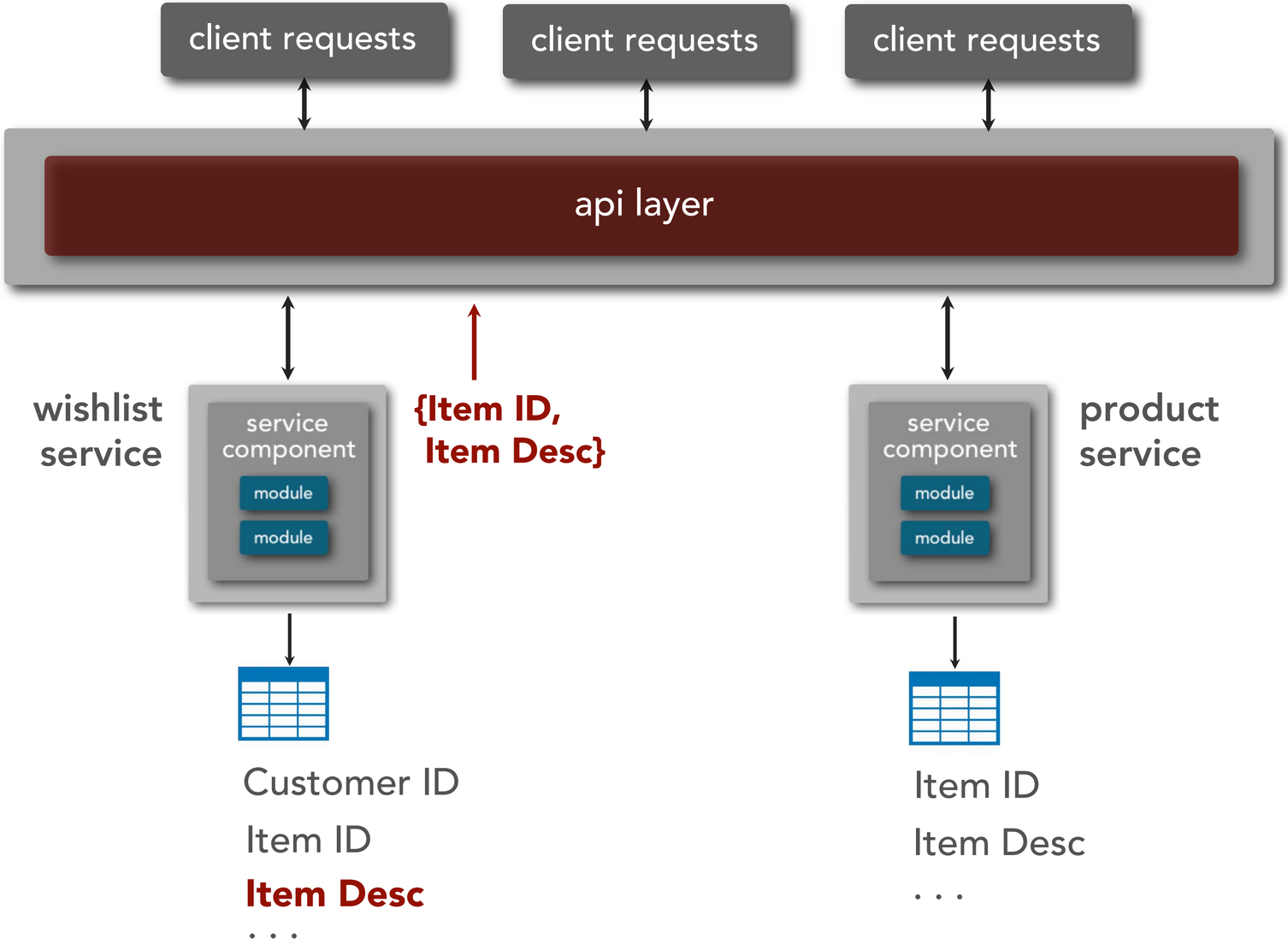
	replicated cache 	distributed cache 	near cache 
optimization	performance	consistency	balanced
cache size	small cache	large cache	large cache
type of data	relatively static	transactional	relatively static
update frequency	relatively low	high update rate	relatively low
fault tolerance	high	low	low
responsiveness	high	medium	variable

Data Sharing

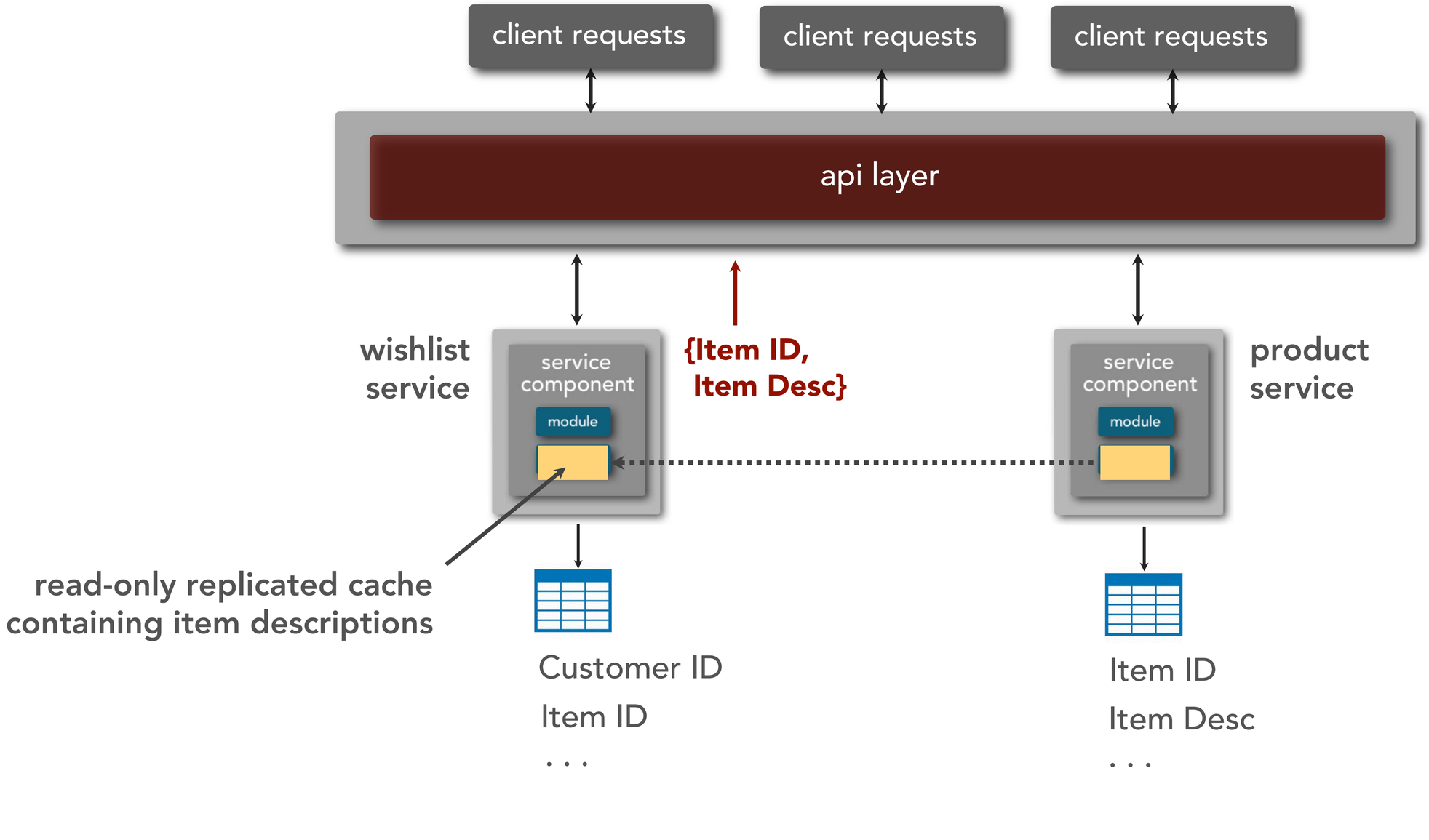
data sharing



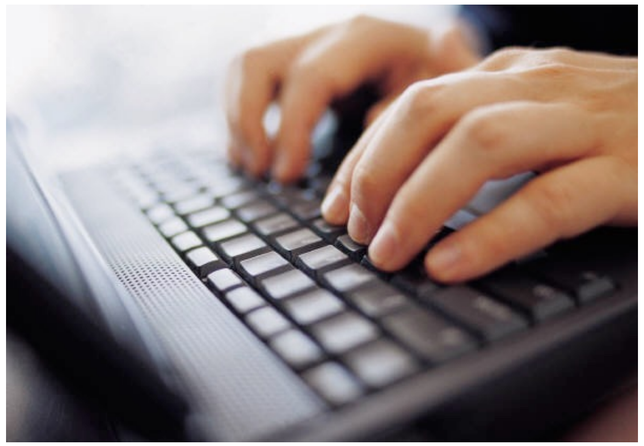
data sharing



data sharing



data sharing



let’s see this in action!

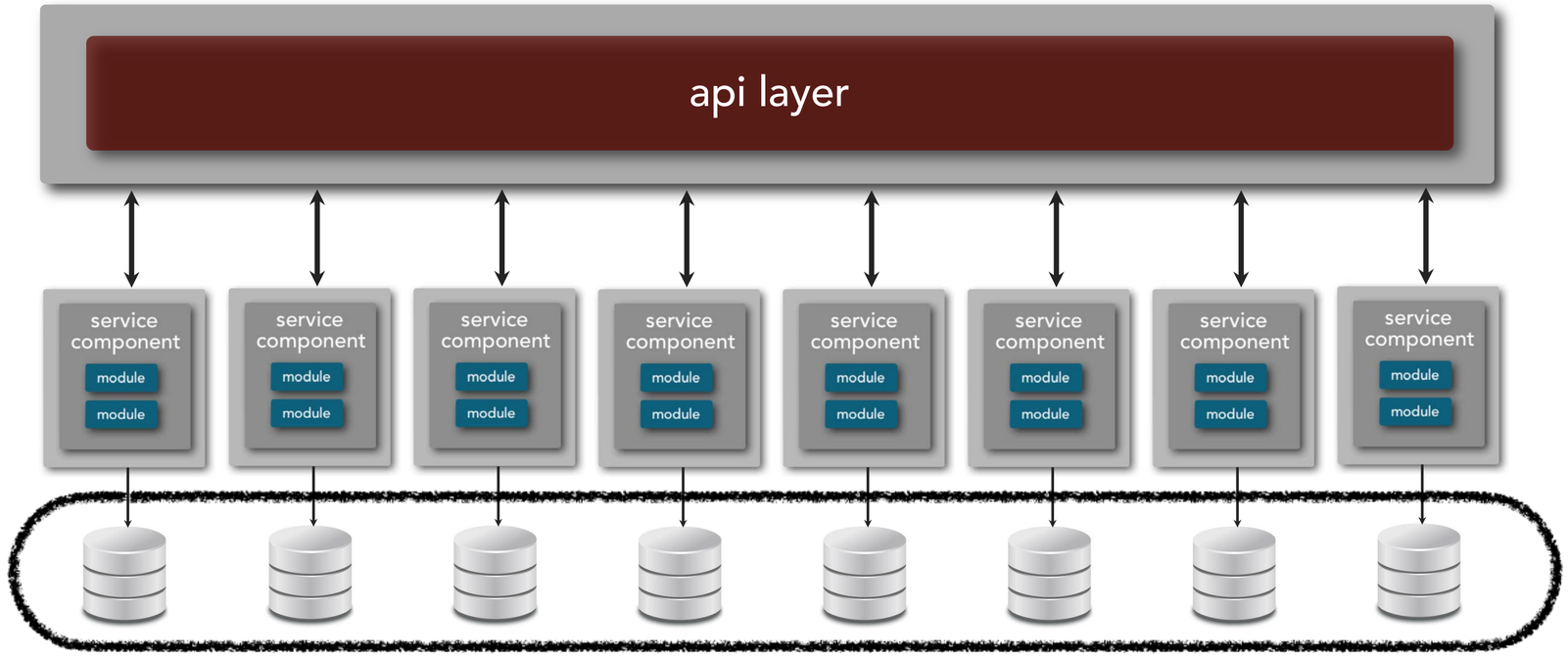
https://github.com/wmr513/caching

https://github.com/wmr513/caching/tree/master/src/main/java/hazelcast

Taking It One Step Further…

space-based microservices

what makes microservices so hard?



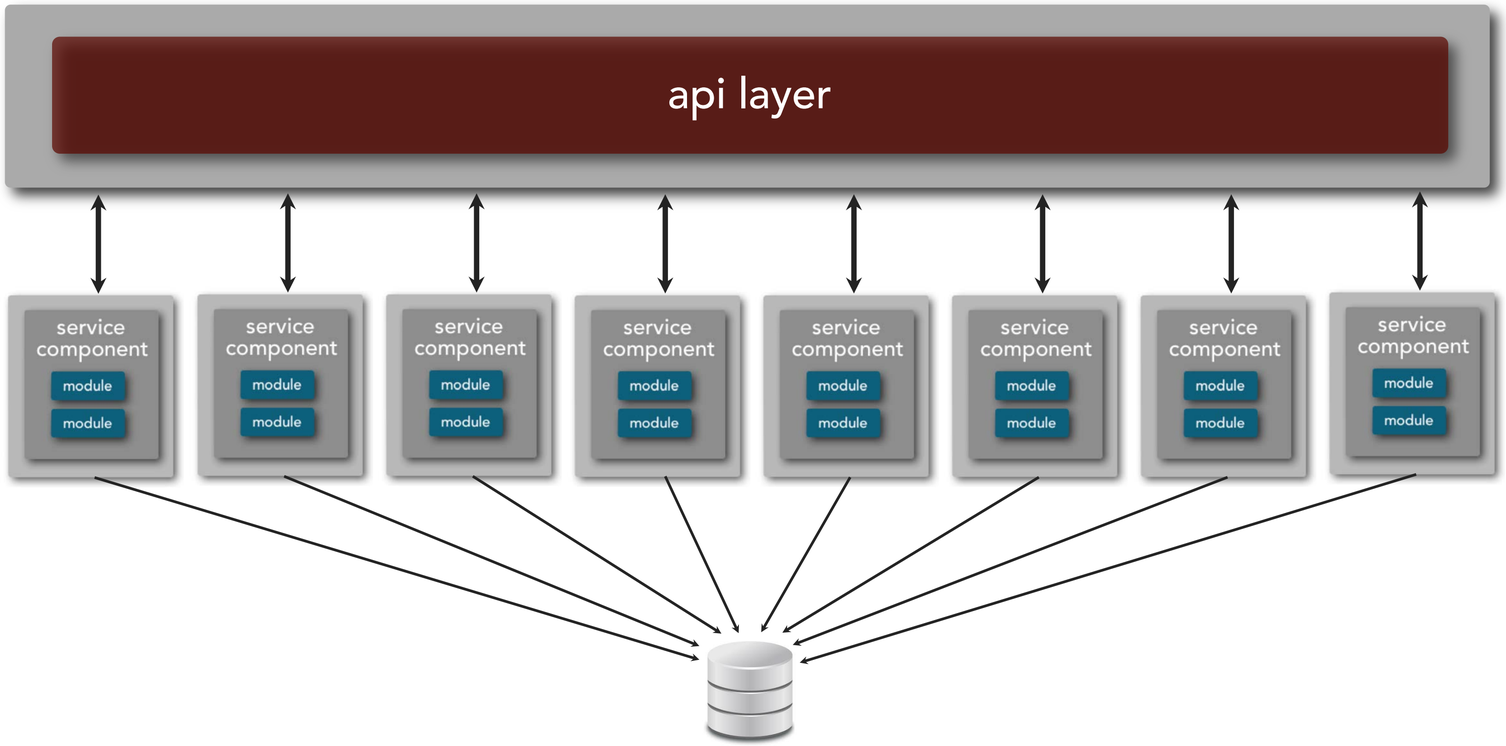
feasibility of breaking apart monolithic databases

data associations and relational artifacts (FK, SP, Views)

reporting and data analytics

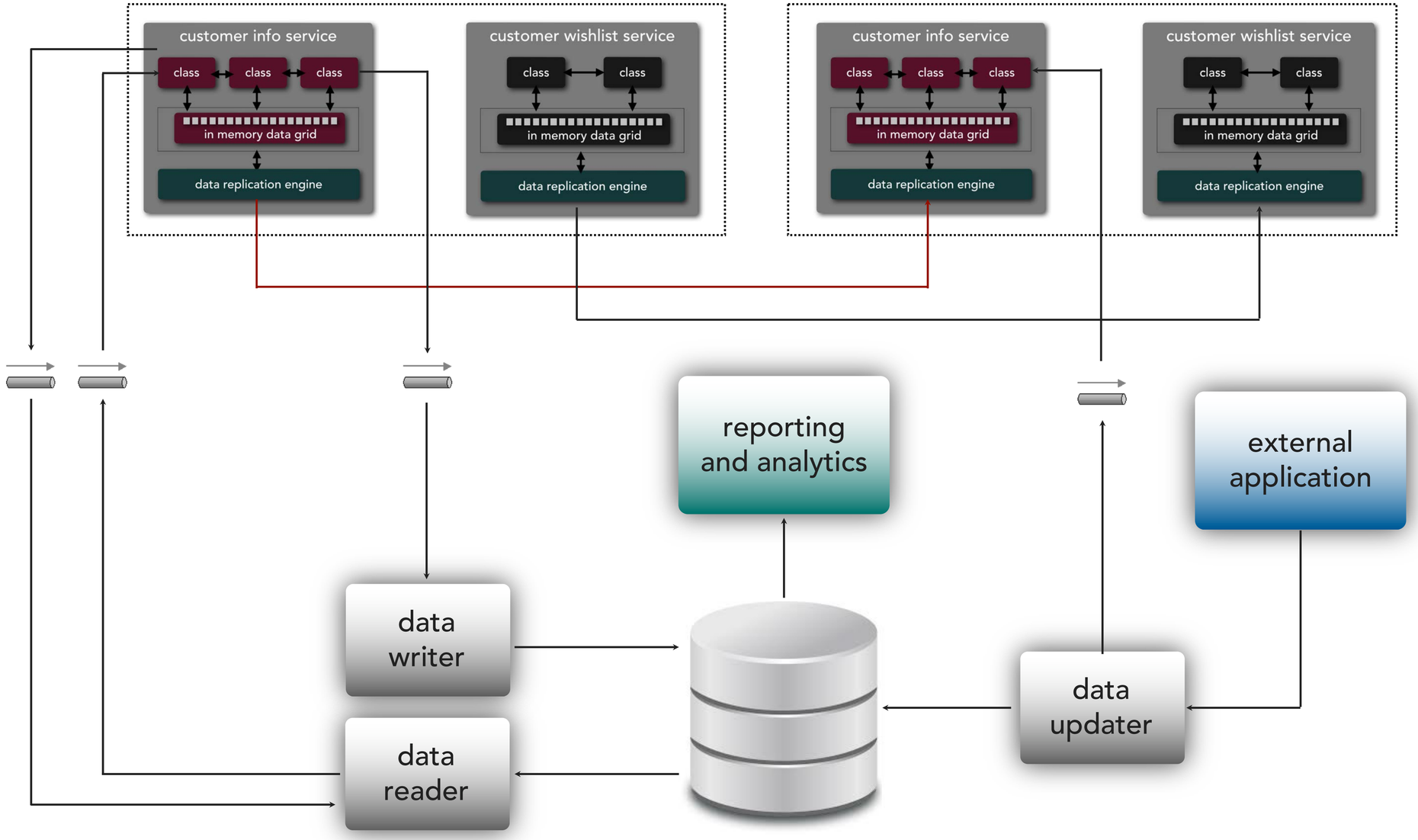
space-based microservices

what about sharing a single monolithic database and schema?



- database becomes a coupling point and single point of failure
- database changes impact too many services
- database connection pool management, data ownership issues

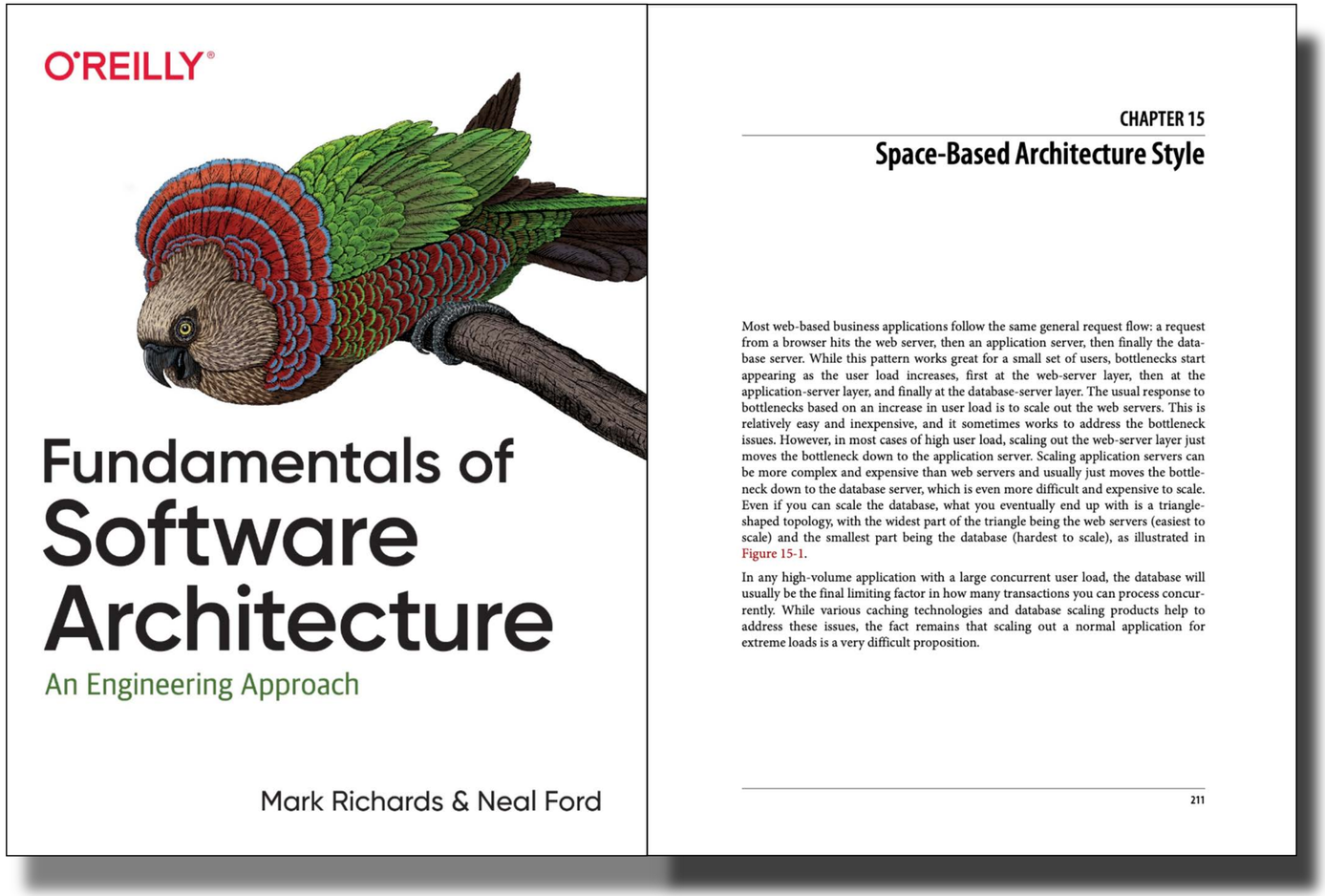
space-based microservices



Fundamentals of Software Architecture

by Mark Richards and Neal Ford

https://www.amazon.com/gp/product/1492043451



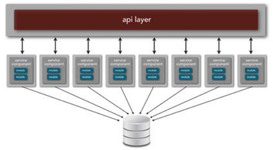
CHAPTER 15

Space-Based Architecture Style

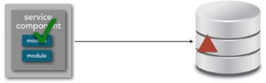
Most web-based business applications follow the same general request flow: a request from a browser hits the web server, then an application server, then finally the database server. While this pattern works great for a small set of users, bottlenecks start appearing as the user load increases, first at the web-server layer, then at the application-server layer, and finally at the database-server layer. The usual response to bottlenecks based on an increase in user load is to scale out the web servers. This is relatively easy and inexpensive, and it sometimes works to address the bottleneck issues. However, in most cases of high user load, scaling out the web-server layer just moves the bottleneck down to the application server. Scaling application servers can be more complex and expensive than web servers and usually just moves the bottleneck down to the database server, which is even more difficult and expensive to scale. Even if you can scale the database, what you eventually end up with is a triangle-shaped topology, with the widest part of the triangle being the web servers (easiest to scale) and the smallest part being the database (hardest to scale), as illustrated in Figure 15-1.

In any high-volume application with a large concurrent user load, the database will usually be the final limiting factor in how many transactions you can process concurrently. While various caching technologies and database scaling products help to address these issues, the fact remains that scaling out a normal application for extreme loads is a very difficult proposition.

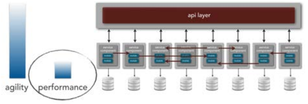
space-based microservices



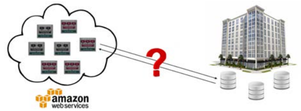
share a single database



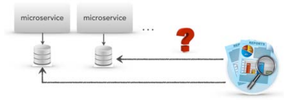
version database changes



increased performance

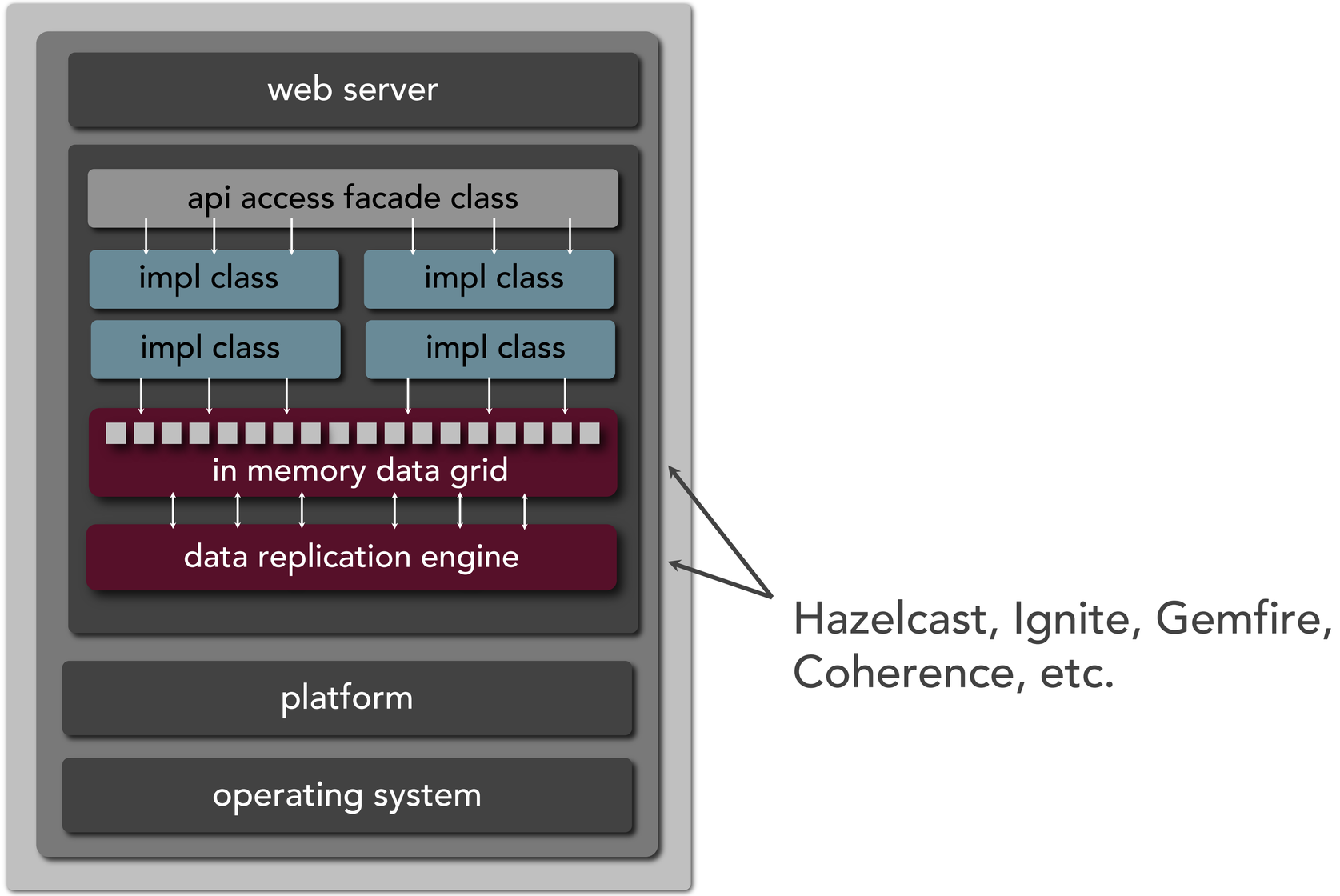


cloud-based data synchronization

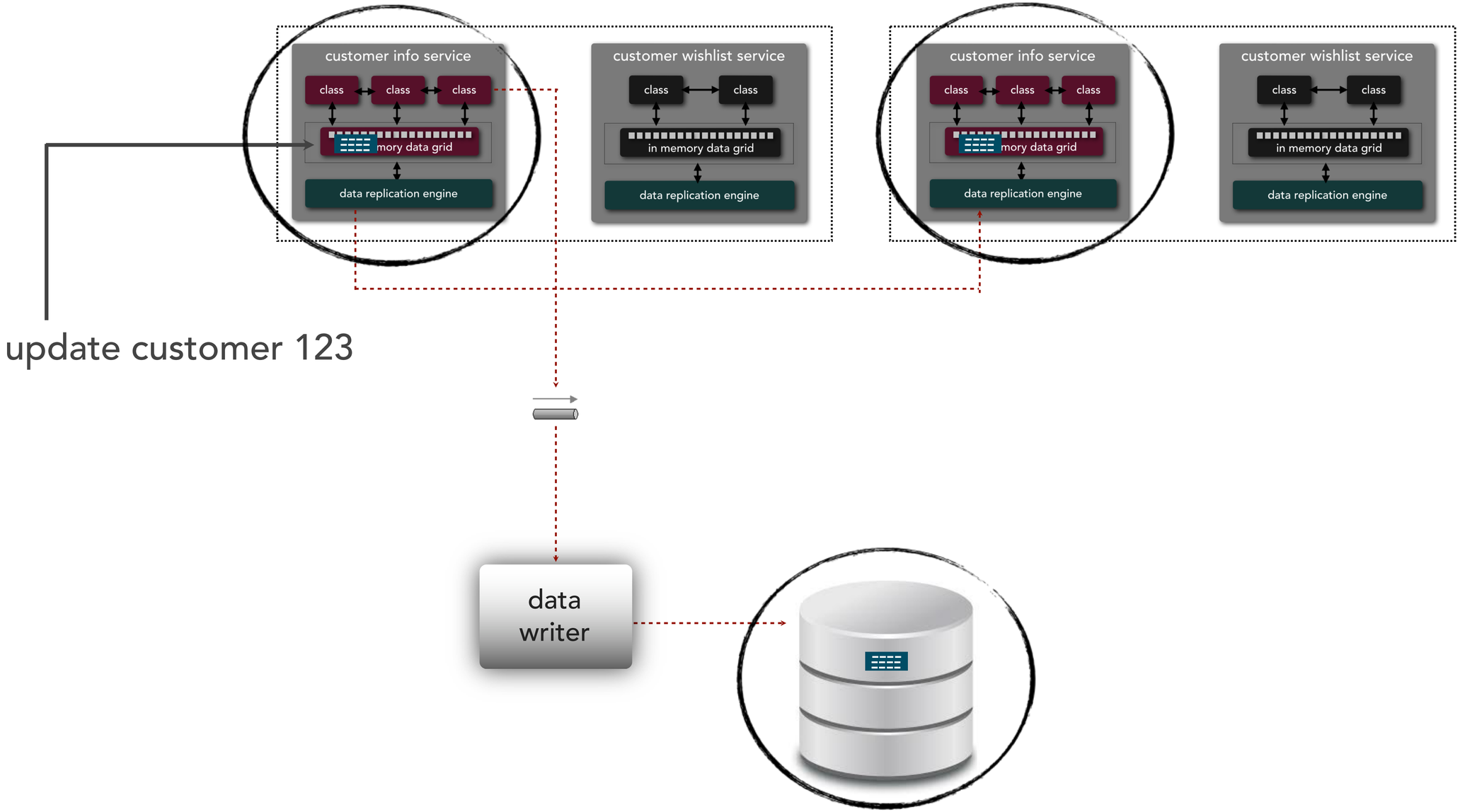


reporting and data analytics

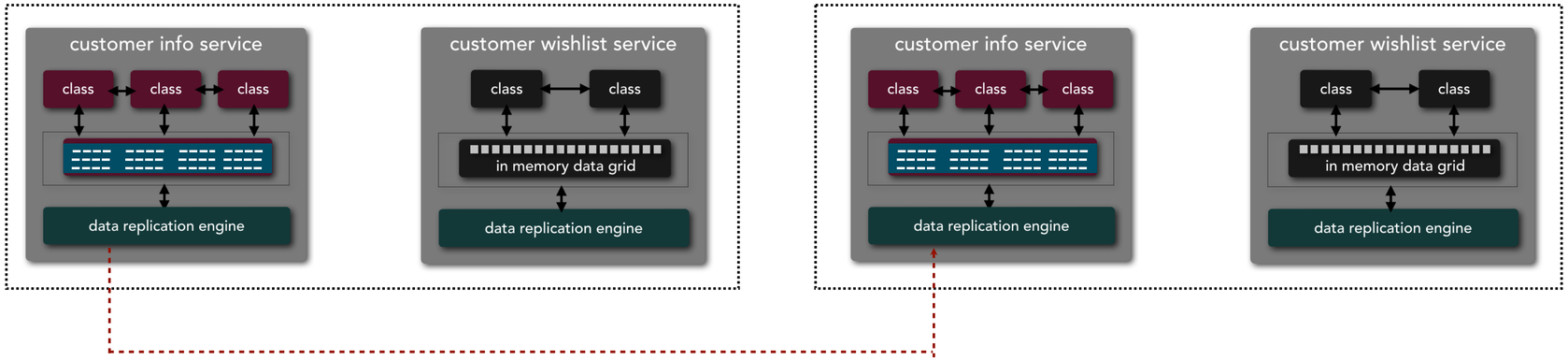
space-based microservices

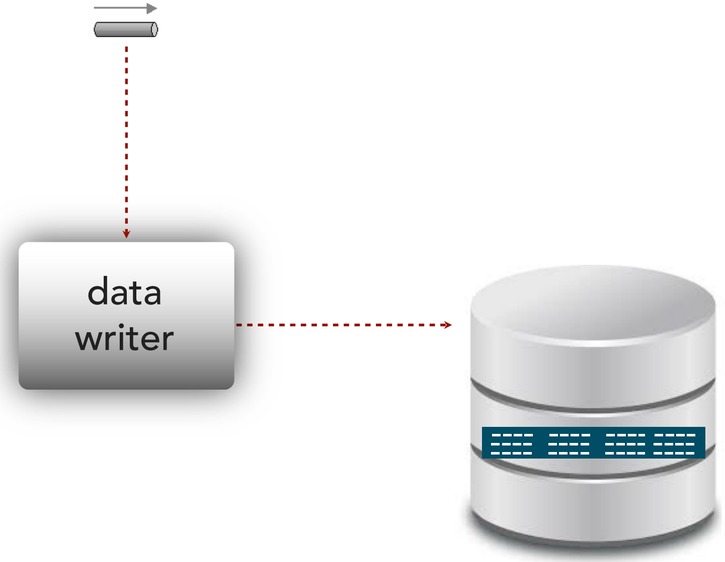


space-based microservices

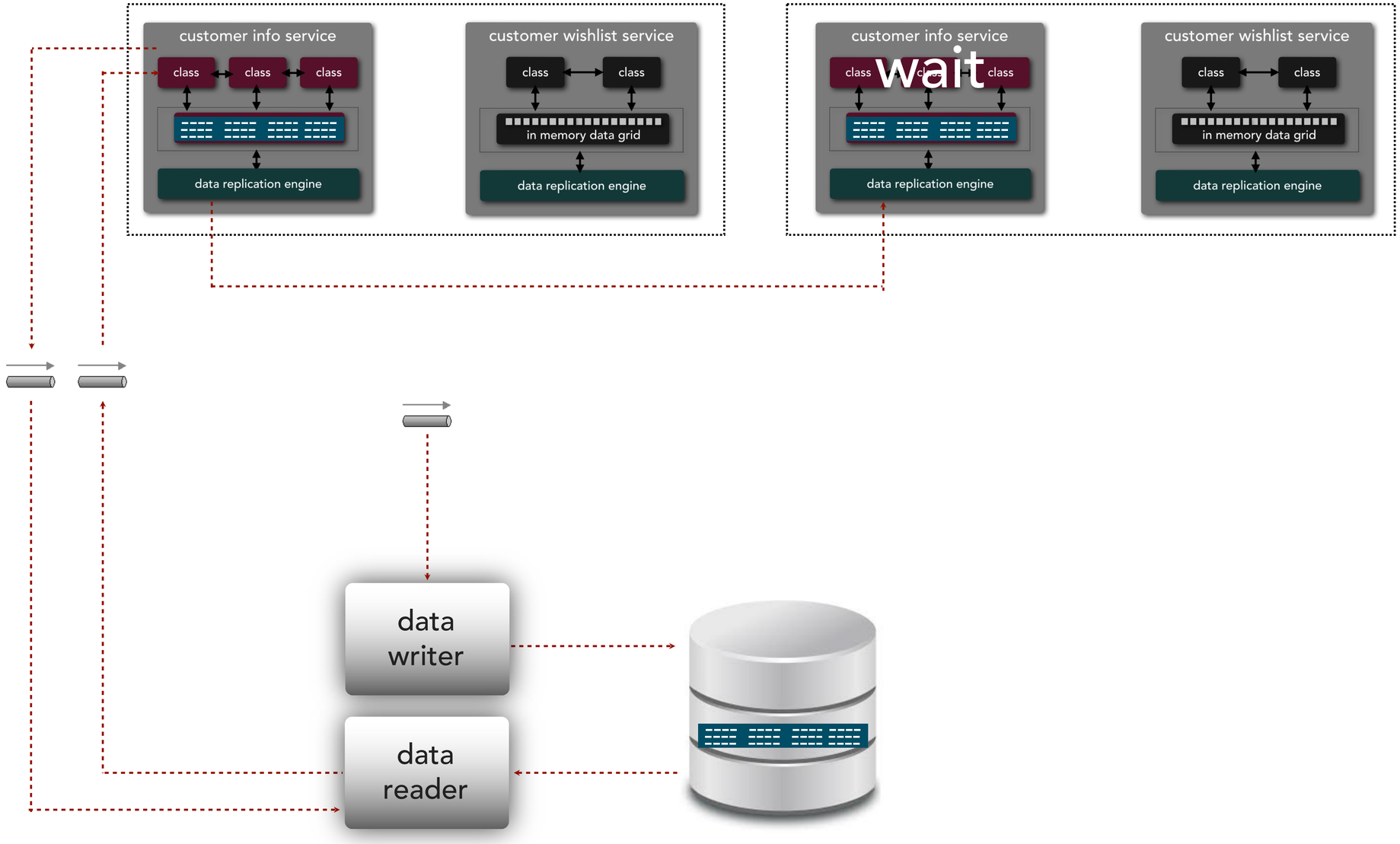


space-based microservices

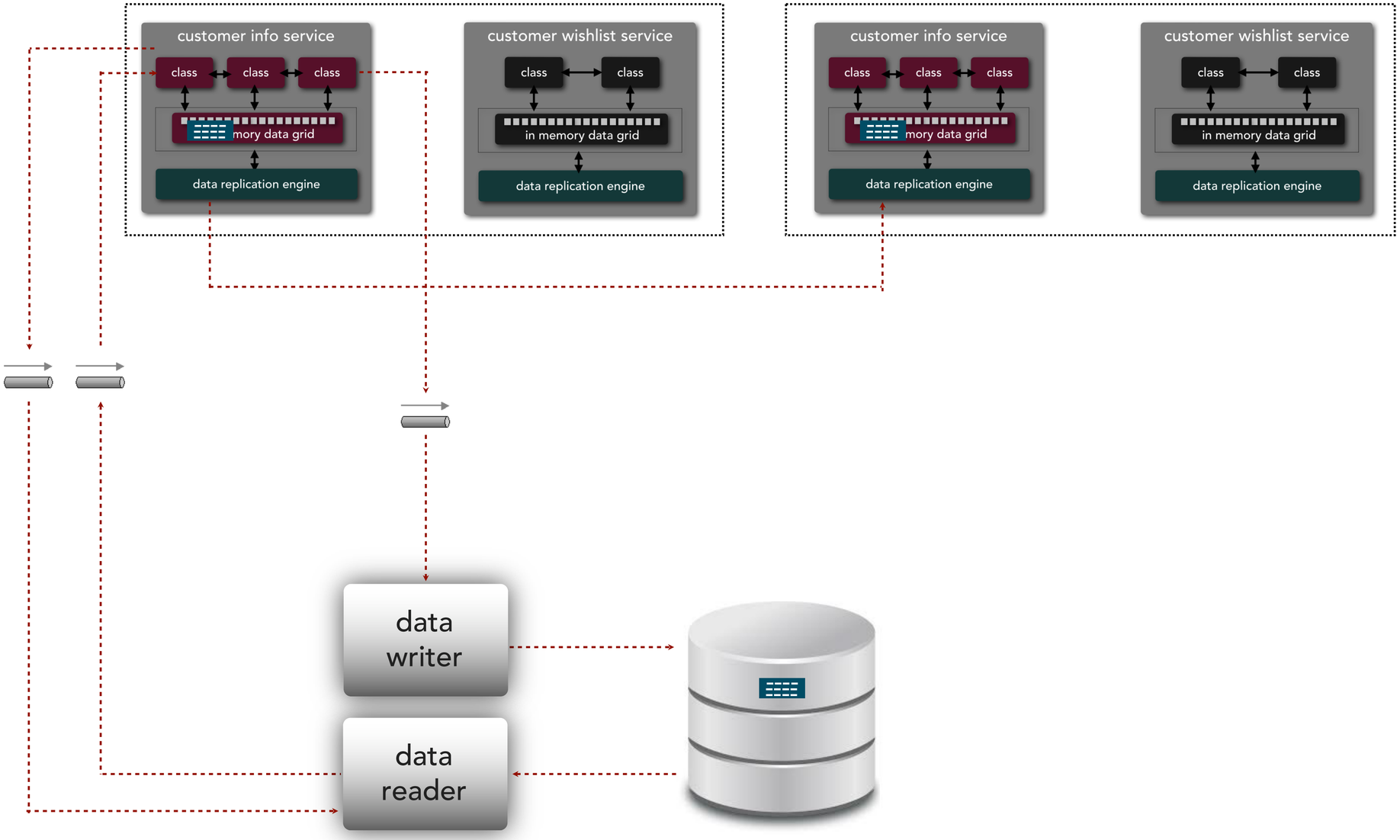




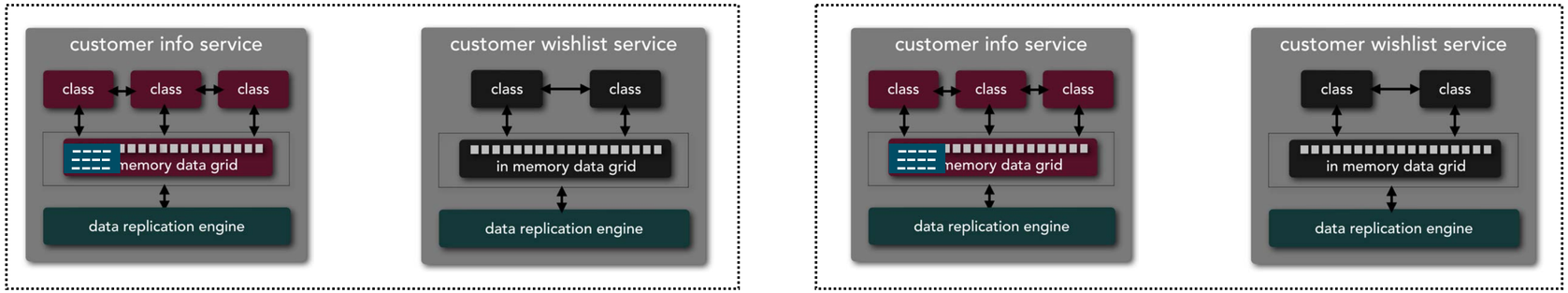
space-based microservices

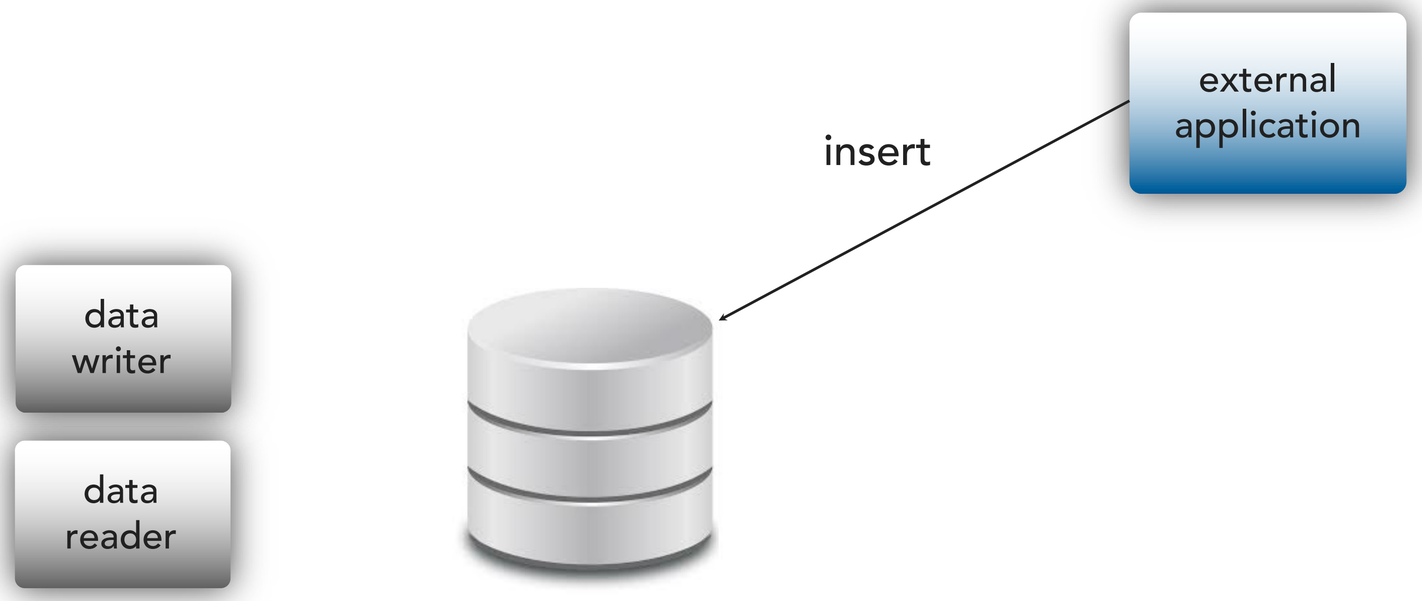


space-based microservices

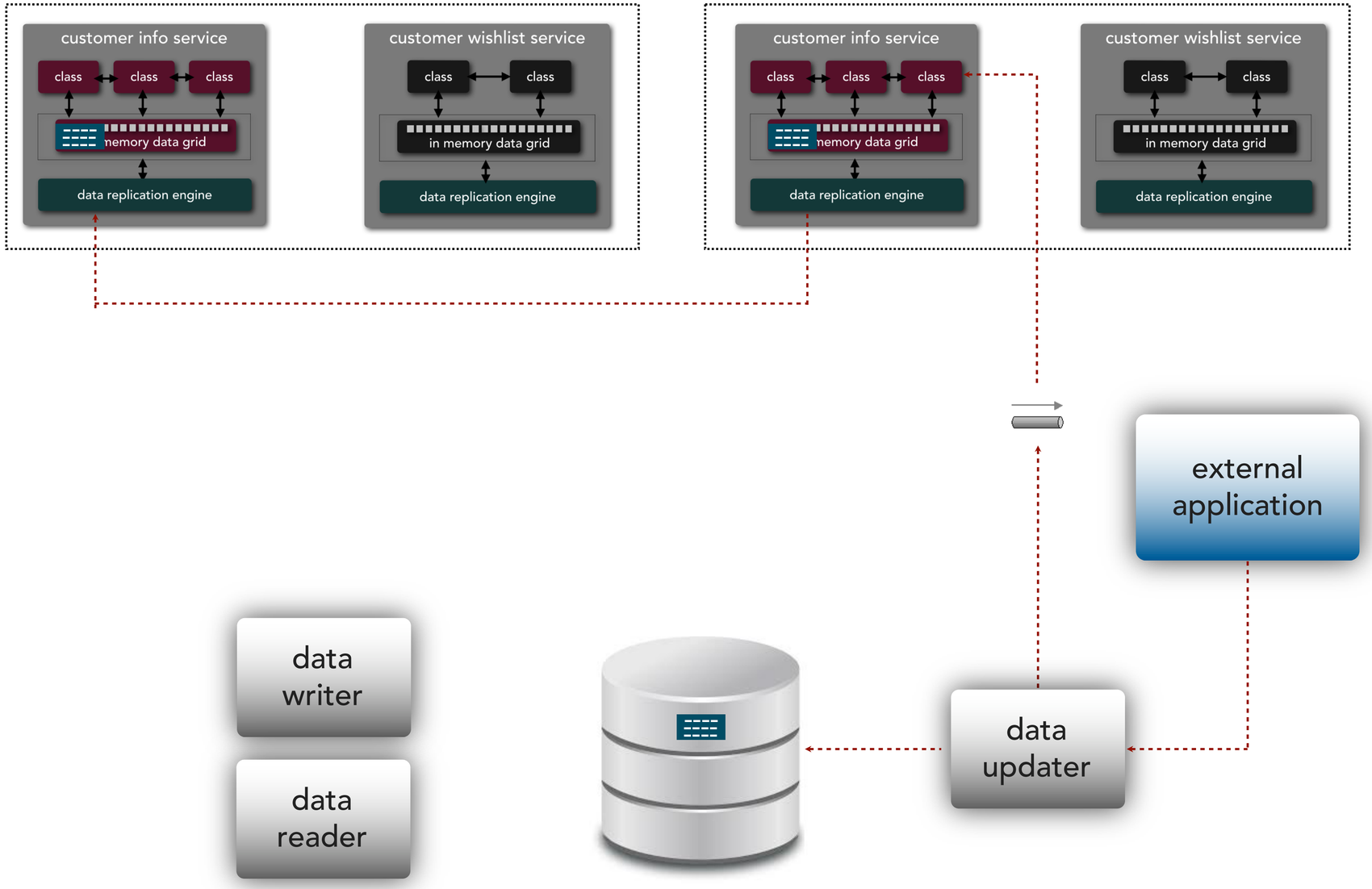


space-based microservices

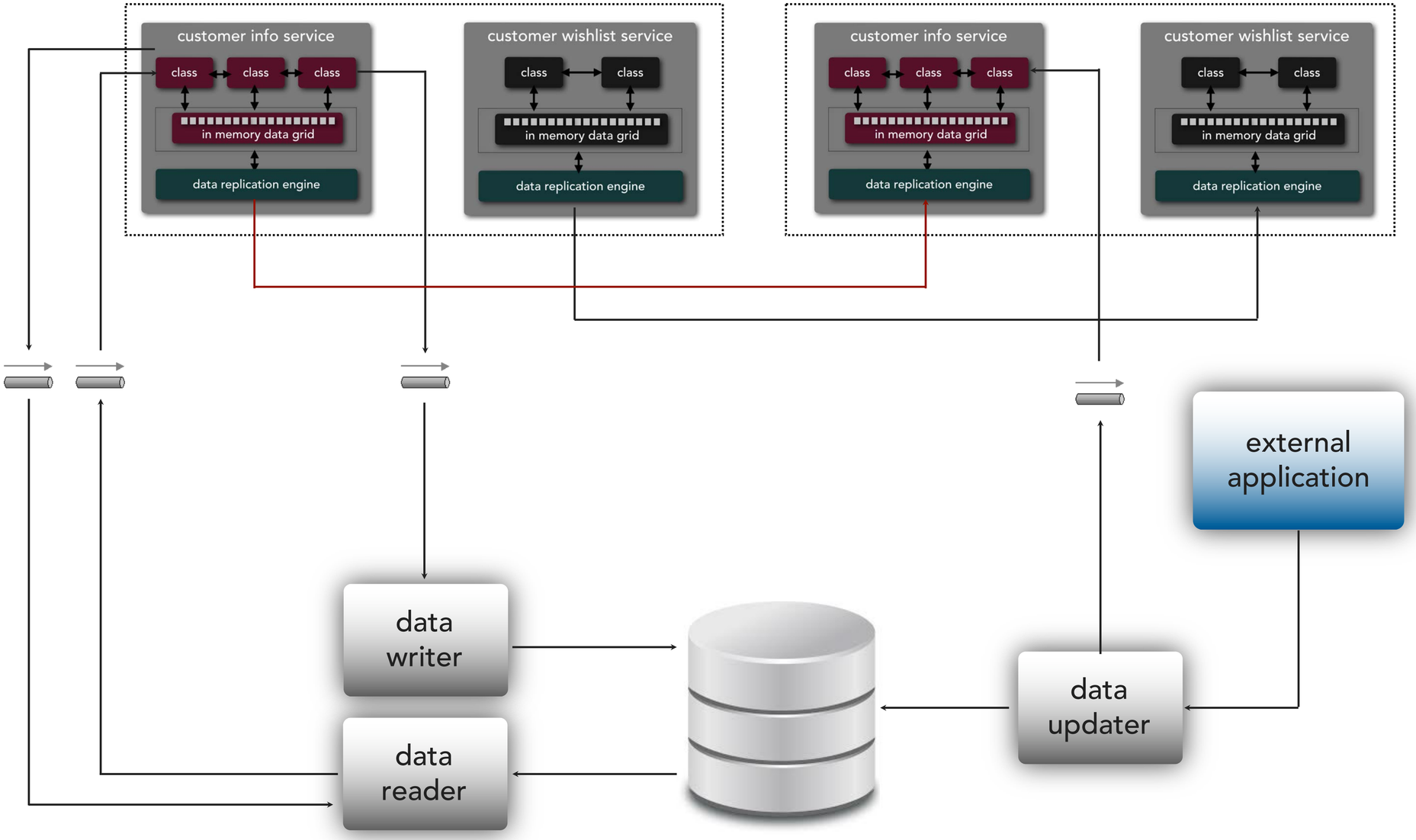




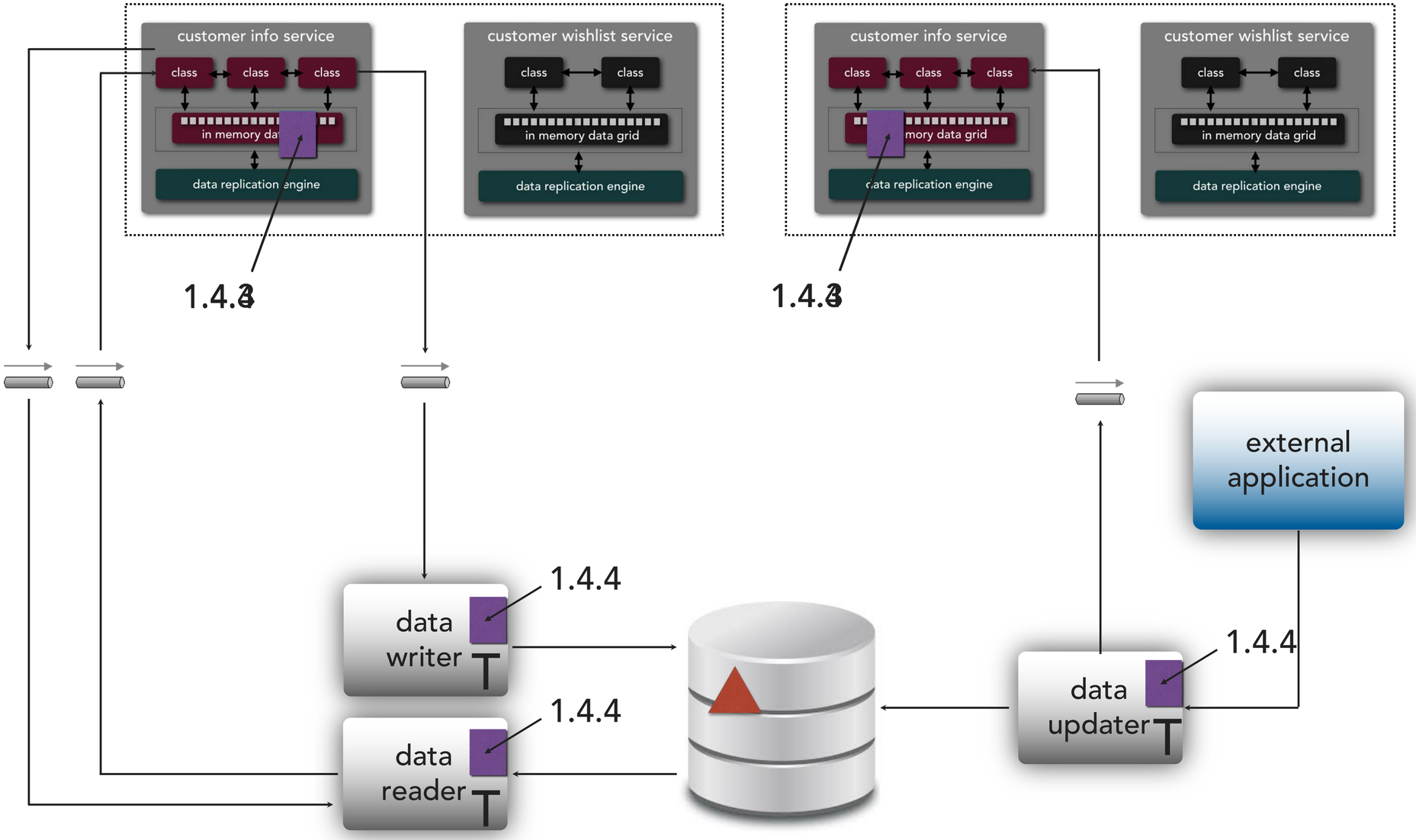
space-based microservices



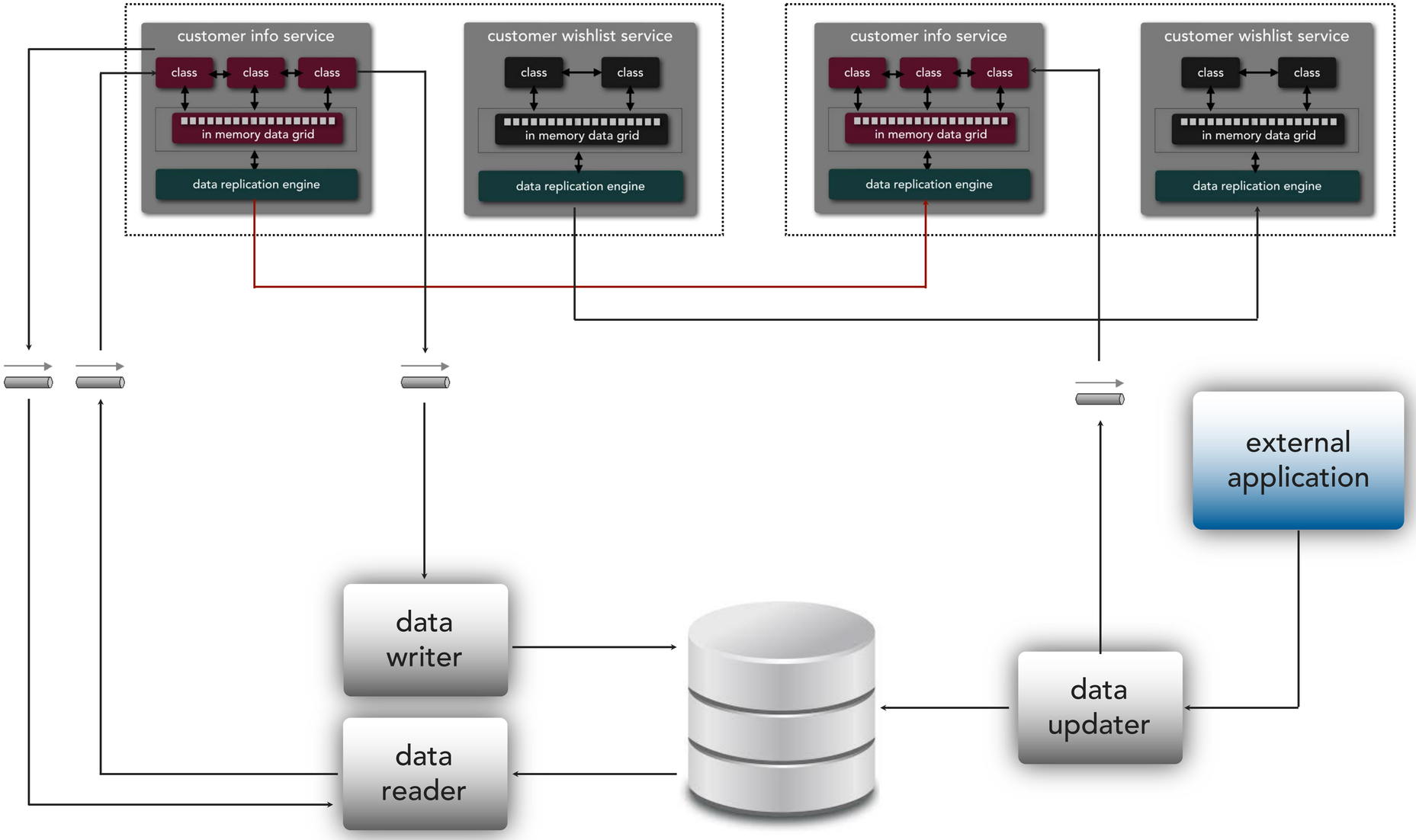
space-based microservices



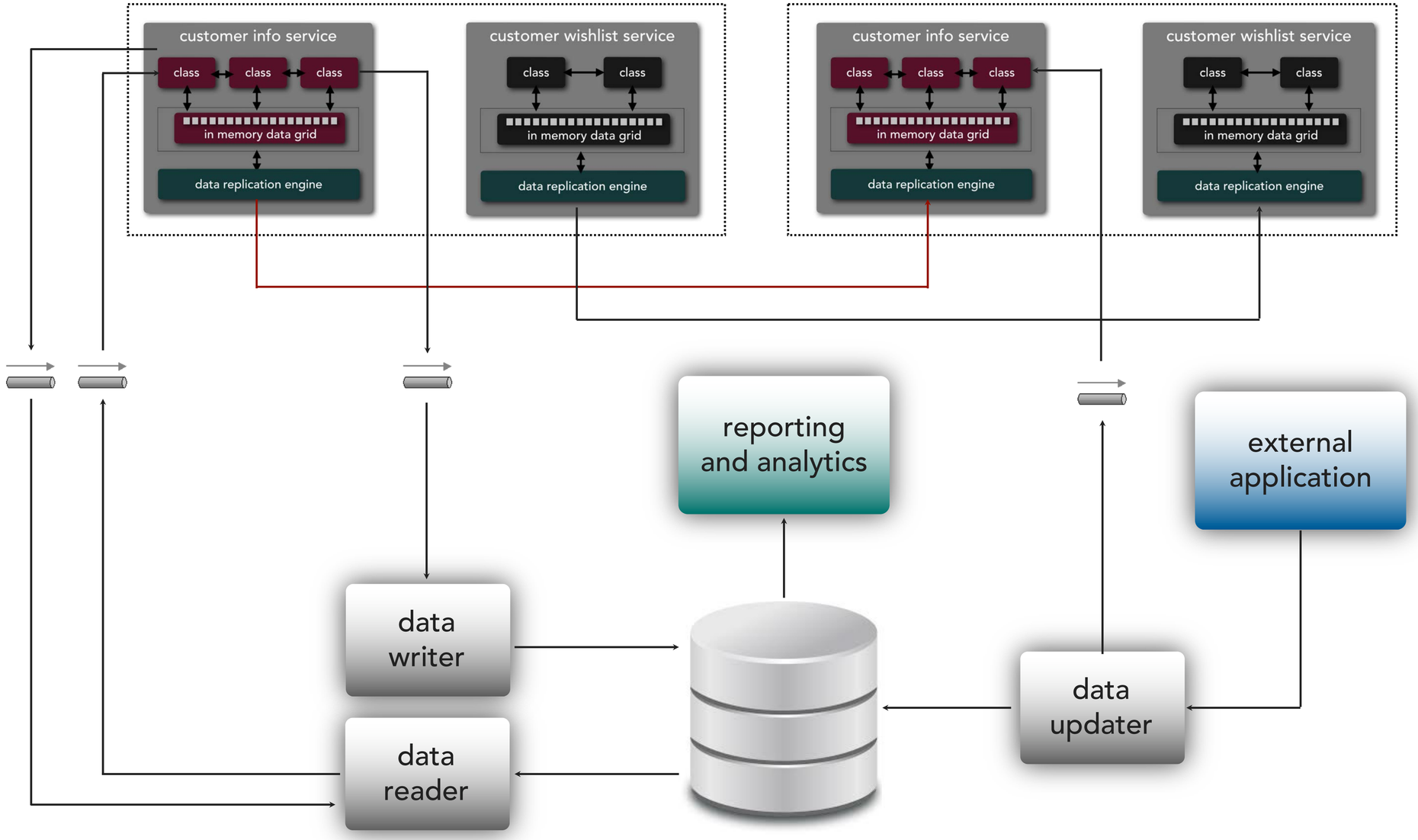
space-based microservices



space-based microservices



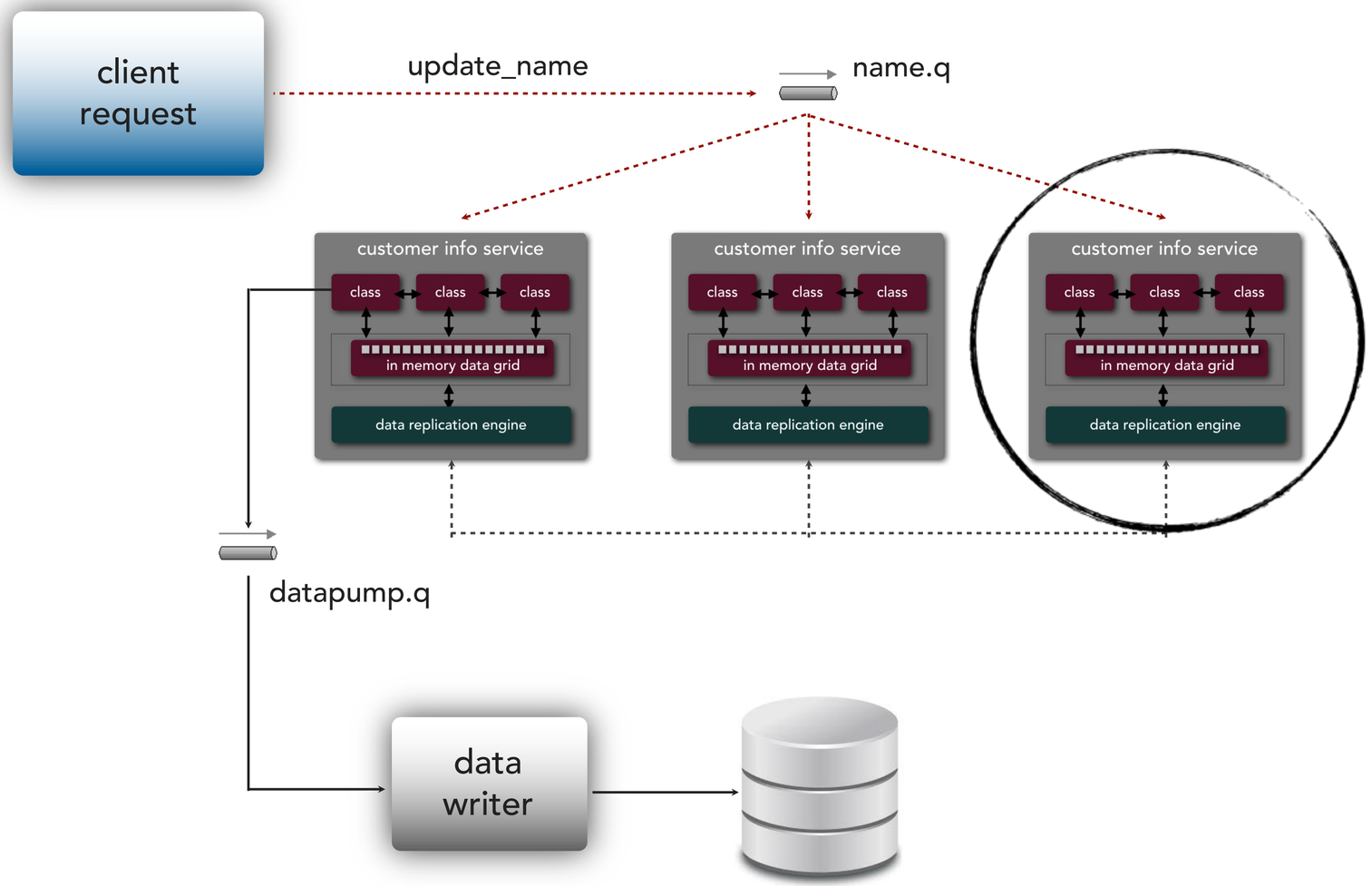
space-based microservices



space-based microservices

let’s see this in action!

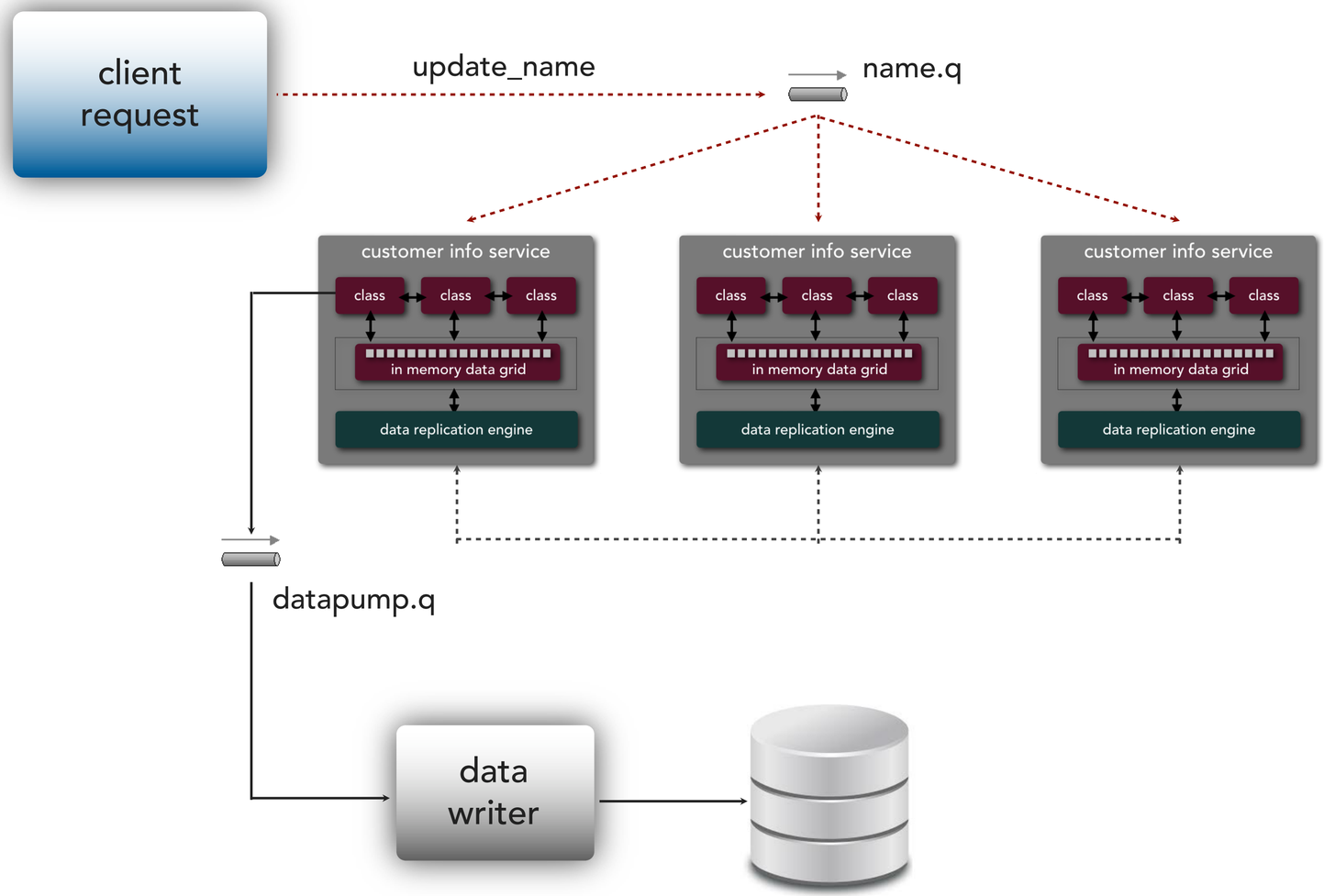
space-based microservices



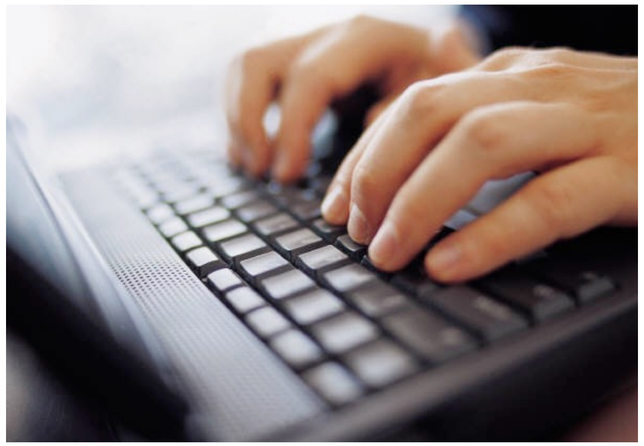
customer info service

```
Channel channel = AMQPCommon.connect();
QueueingConsumer consumer = new QueueingConsumer(channel);
channel.basicConsume("name.q", true, consumer);


HazelcastInstance hz = Hazelcast.newHazelcastInstance();
Map<String, String> cache = hz.getReplicatedMap("names");


 while (true) {
    QueueingConsumer.Delivery msg = consumer.nextDelivery(1000);
    if (msg != null) {
        String name = new String(msg.getBody());
        System.out.println("--> UPDATING NAME: " + name);
        cache.put("123", name);

        byte[] message = msg.getBody();
        channel.basicPublish("", "datapump.q", null, message);
     }
 }
```

space-based microservices



space-based microservices



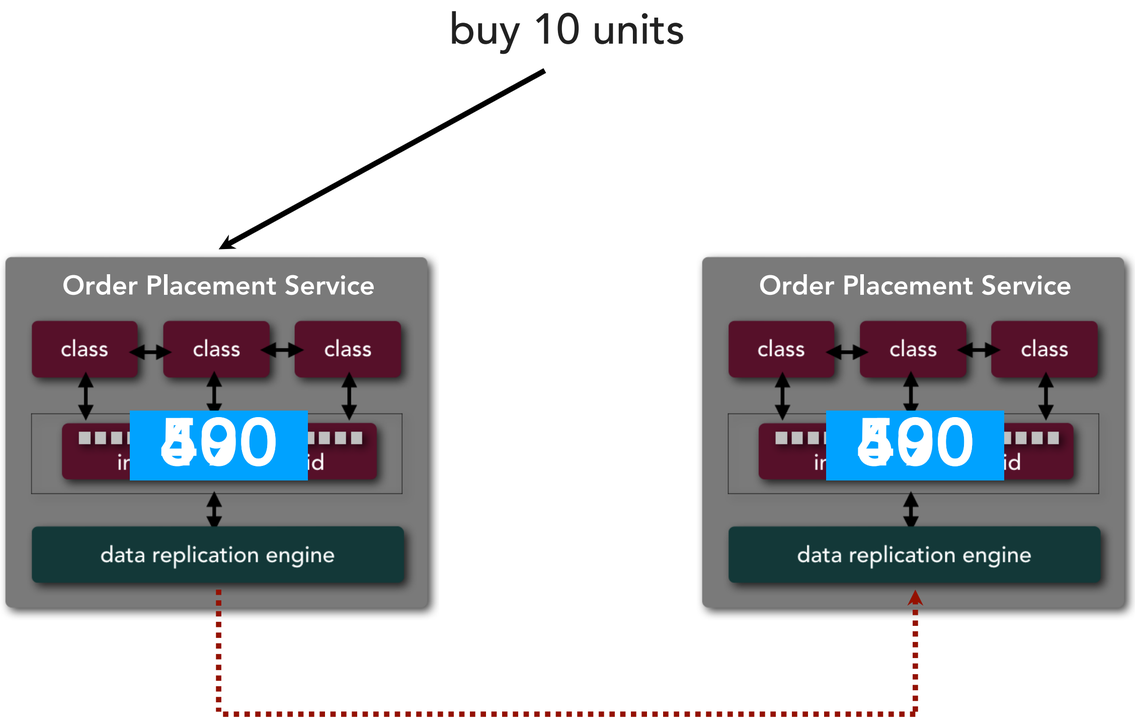
let’s see this in action!

https://github.com/wmr513/caching

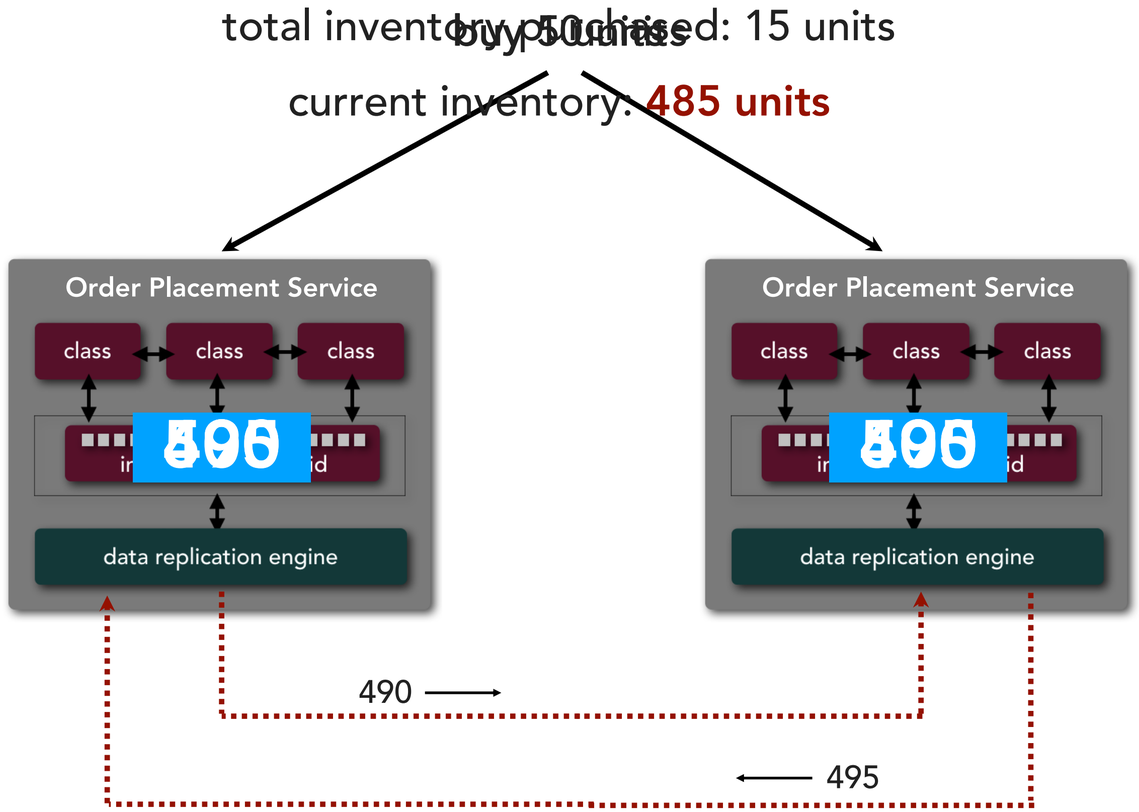
https://github.com/wmr513/caching/tree/master/src/main/java/hazelcast

Replication and Data Collisions

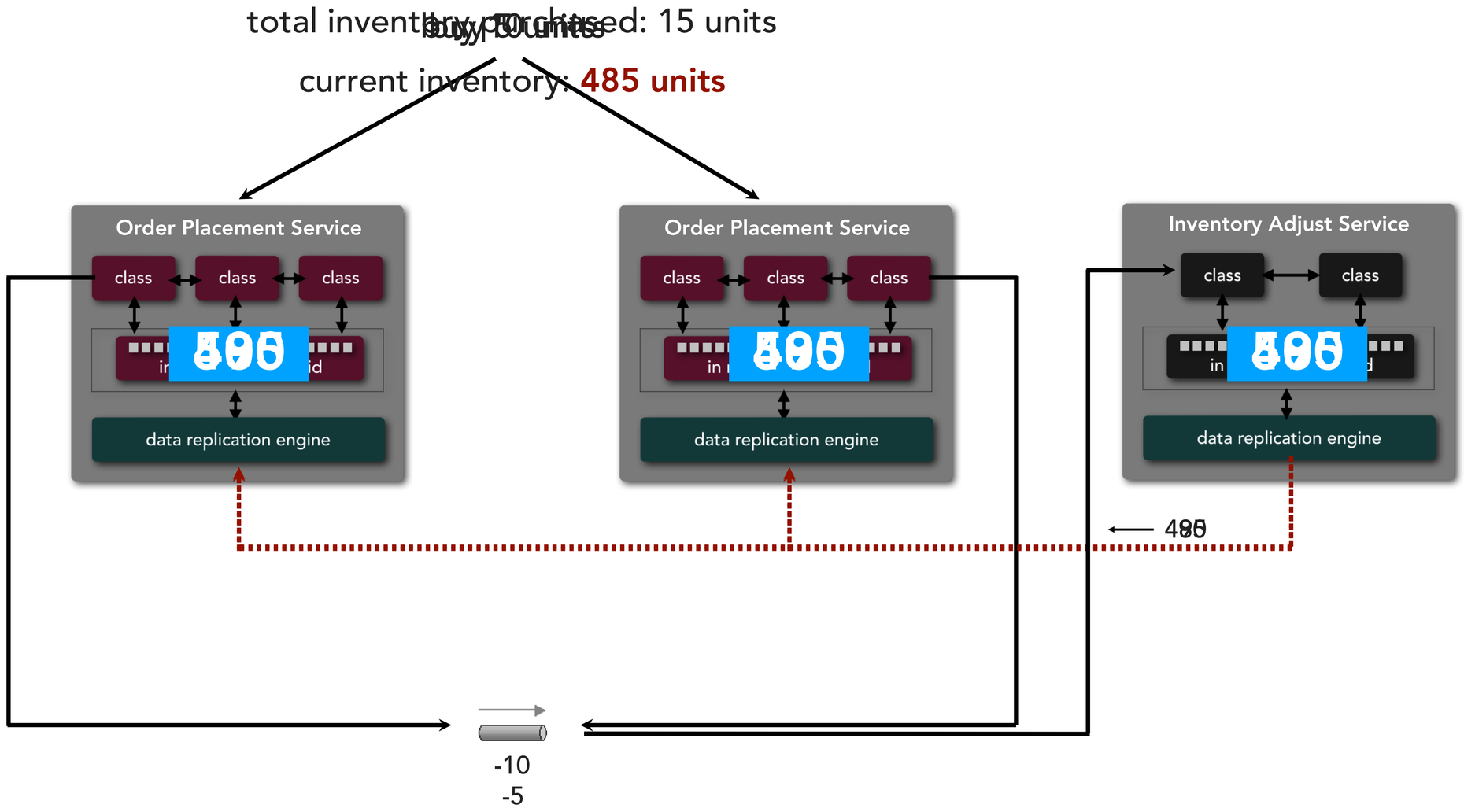
replicated cache data collisions



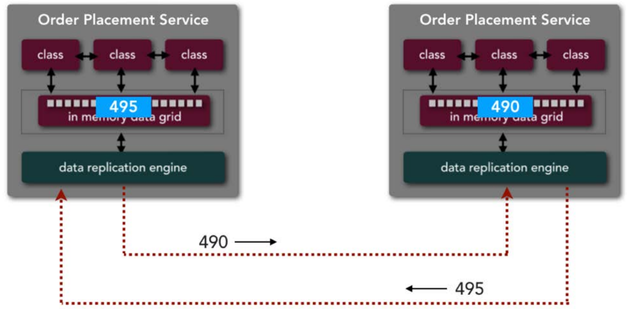
replicated cache data collisions



replicated cache data collisions

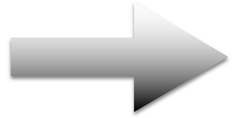


replicated cache data collisions



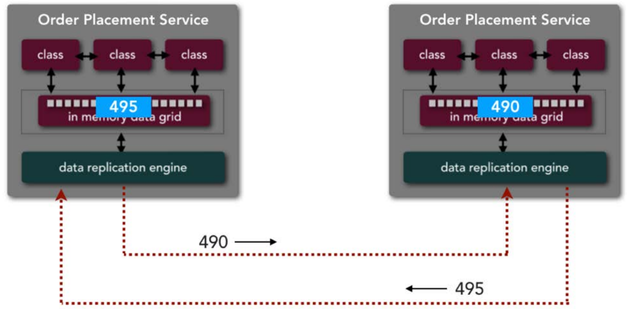
$$\text{collision rate} = \text{num instances} * \frac{\text{update rate}^2}{\text{cache size}} * \text{replication latency}$$

number instances: 5
update rate: 200/second
cache size: 50,000 rows
replication latency: 100 milliseconds



720,000 updates/hr
1440 collisions/hr
0.2% collision rate

replicated cache data collisions



$$\text{collision rate} = \text{num instances} * \frac{\text{update rate}^2}{\text{cache size}} * \text{replication latency}$$

number instances: 5
update rate: 200/second
cache size: 50,000 rows
replication latency: 100 milliseconds

720,000 updates/hr
1440 collisions/hr
0.2% collision rate

https://github.com/wmr513/caching/tree/master/src/main/java/datacollisions

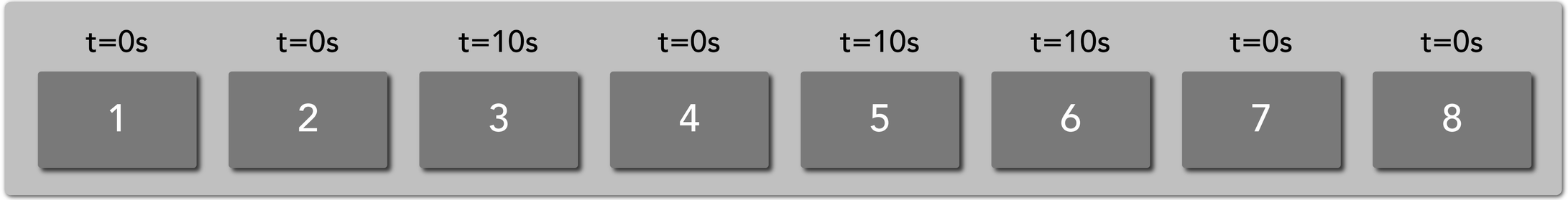
Cache Eviction Policies

Time-To-Live Policy (TTL)

cache eviction strategies

time to live policy (TTL)

cache items are evicted based on a timeout parameter



nameCache.put(1, "Mark", 10, TimeUnit.SECONDS) (Hazelcast)

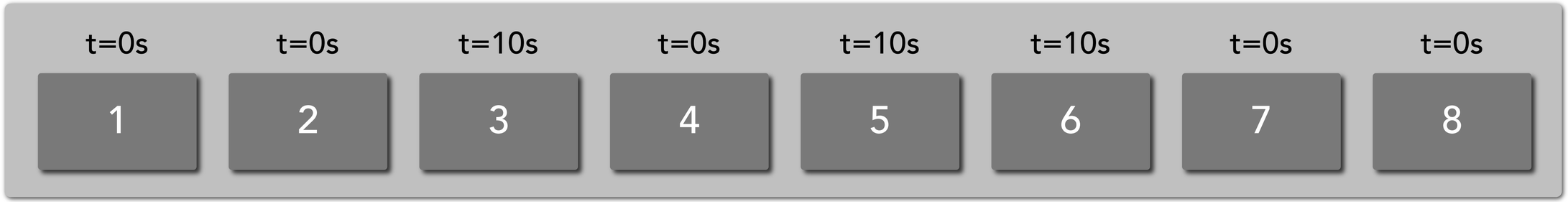
```
</caching-schemes>
  <scheme-name>NAME_CACHE</scheme-name>
    <expiry-delay>10s</expiry-delay>
    ...
</caching-schemes>
```

(Coherence)

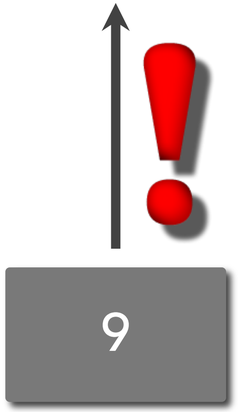
cache eviction strategies

time to live policy (TTL)

cache items are evicted based on a timeout parameter



eviction policy doesn't directly address a full cache scenario
(other policies can be invoked with TTL however)



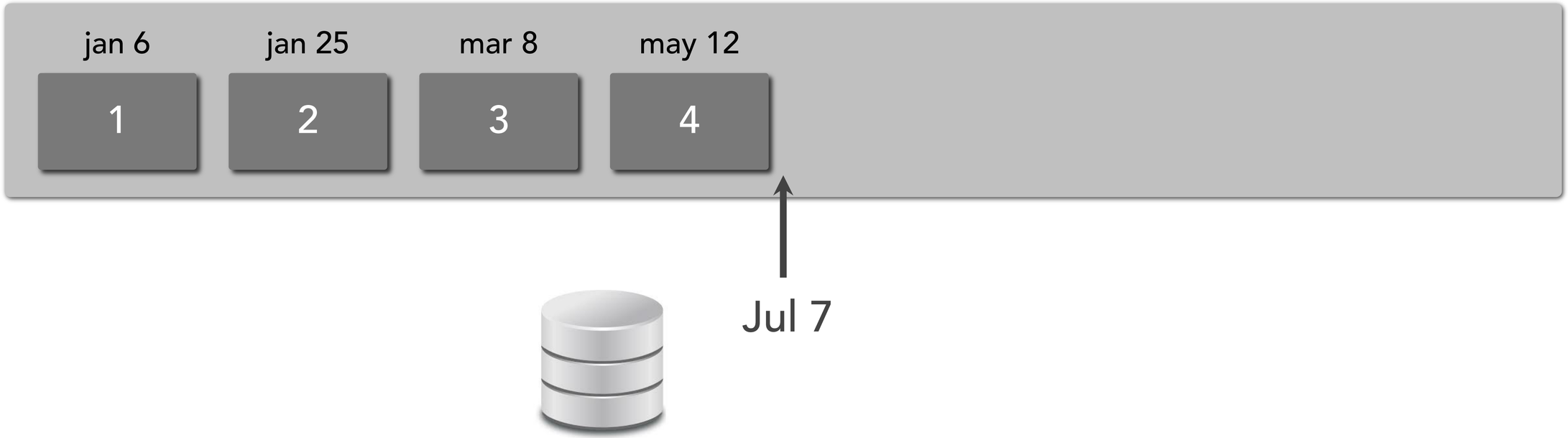
Archive Policy (ARC)

cache eviction strategies

archive policy (ARC)

cache items are evicted based on a date from when they were created (not when they were added to the cache)

example: evict all orders older than 6 months old

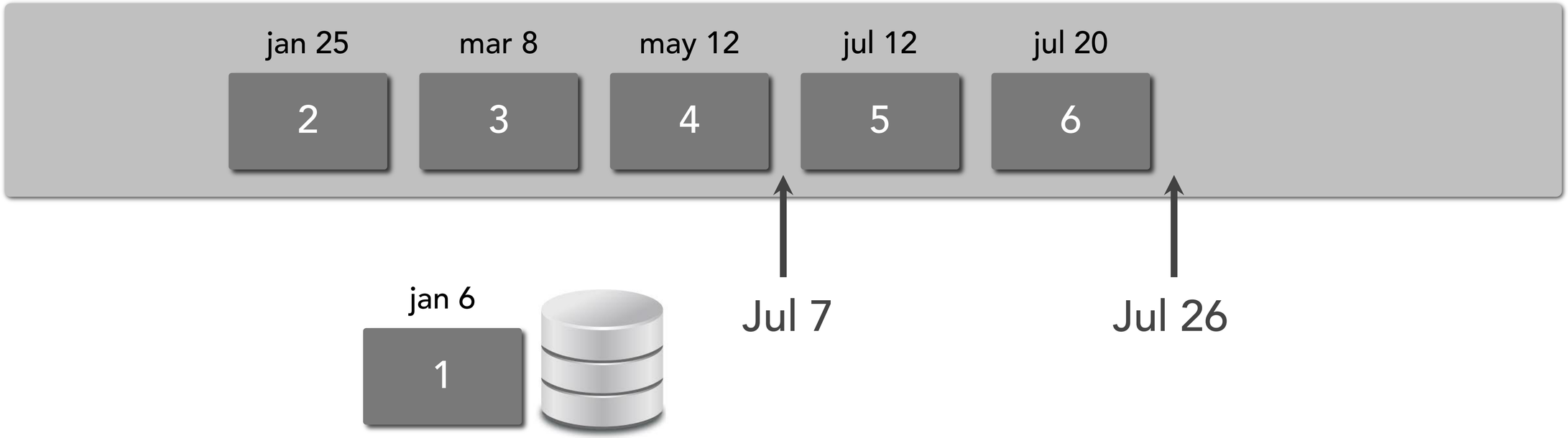


cache eviction strategies

archive policy (ARC)

cache items are evicted based on a date from when they were created (not when they were added to the cache)

example: evict all orders older than 6 months old

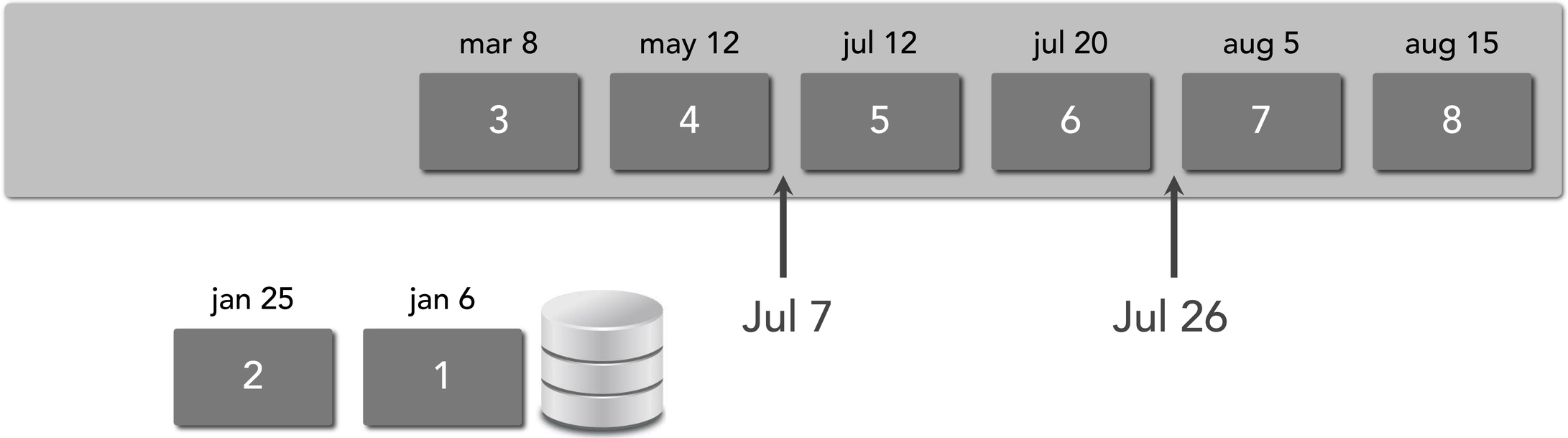


cache eviction strategies

archive policy (ARC)

cache items are evicted based on a date from when they were created (not when they were added to the cache)

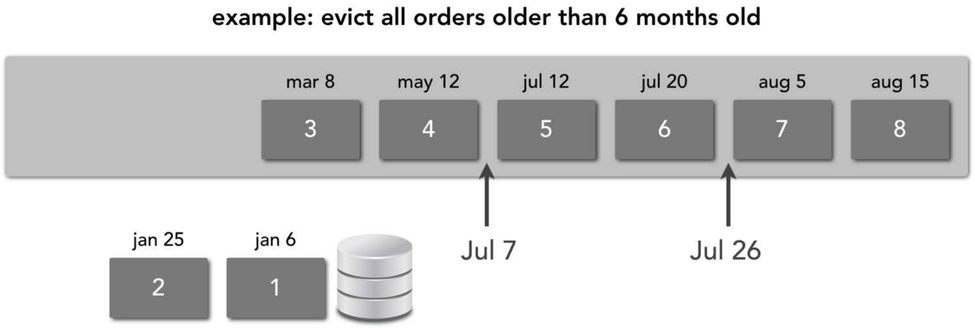
example: evict all orders older than 6 months old



cache eviction strategies

archive policy (ARC)

cache items are evicted based on a date from when they were created (not when they were added to the cache)



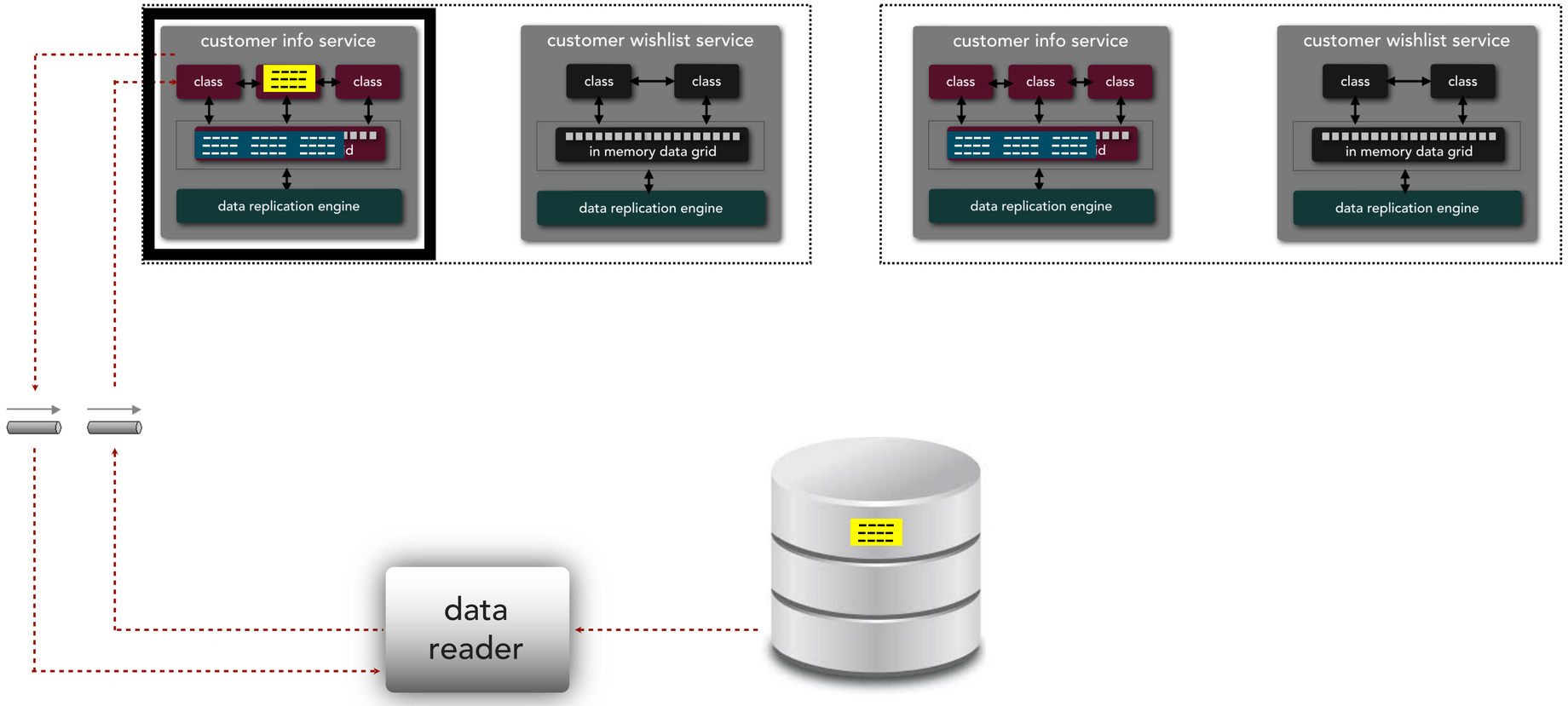
❗ eviction policy doesn't directly address a full cache scenario

✅ excellent strategy for space-based microservices (full replica)

cache eviction strategies

archive policy (ARC)

“retrieve my order history from over 6 months ago”

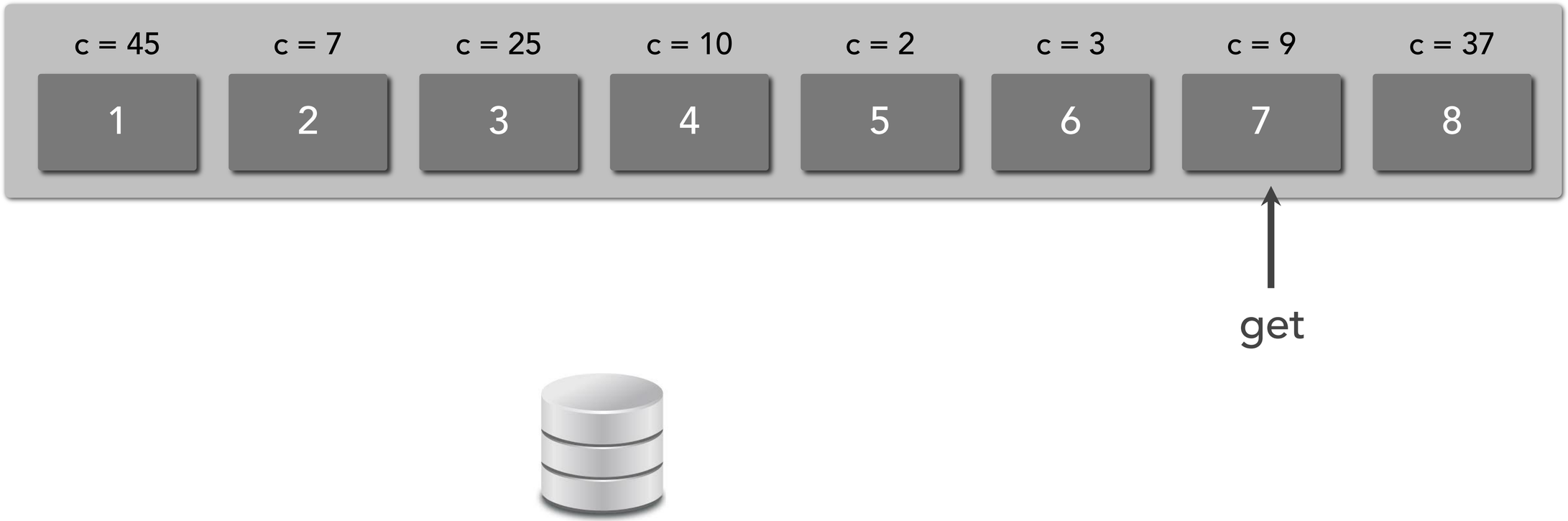


Least Frequently Used Policy (LFU)

cache eviction strategies

least frequently used policy (LFU)

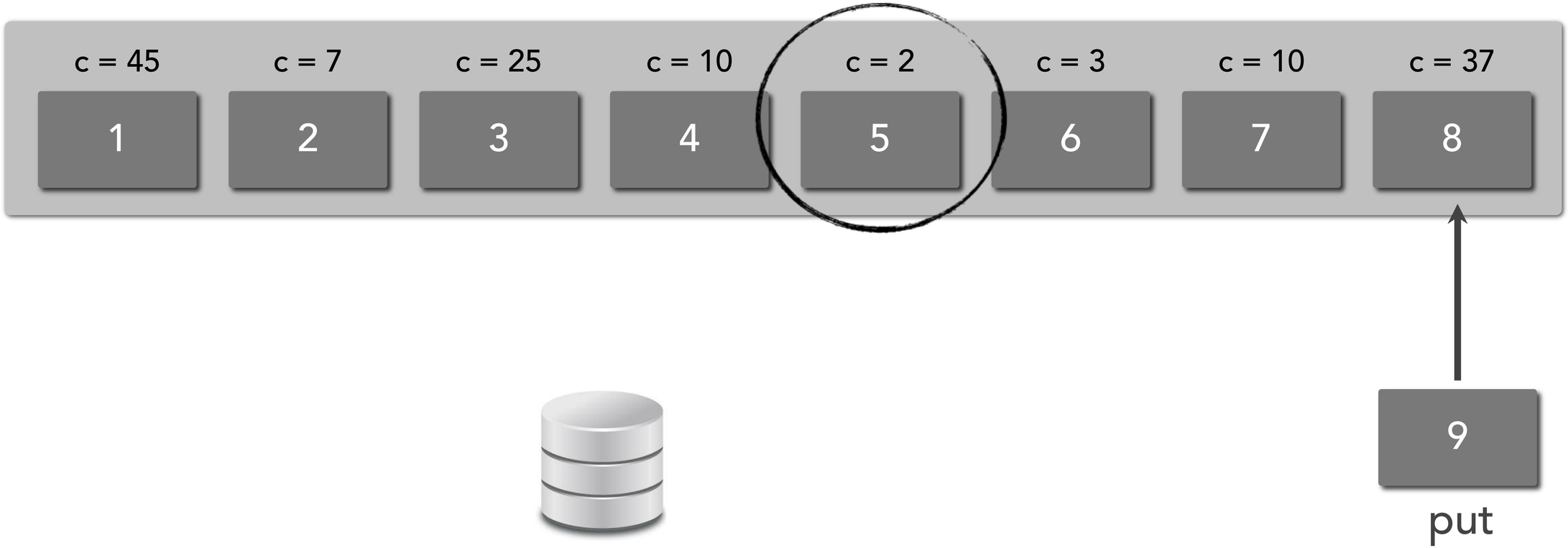
cache items are evicted that are accessed the least amount of times



cache eviction strategies

least frequently used policy (LFU)

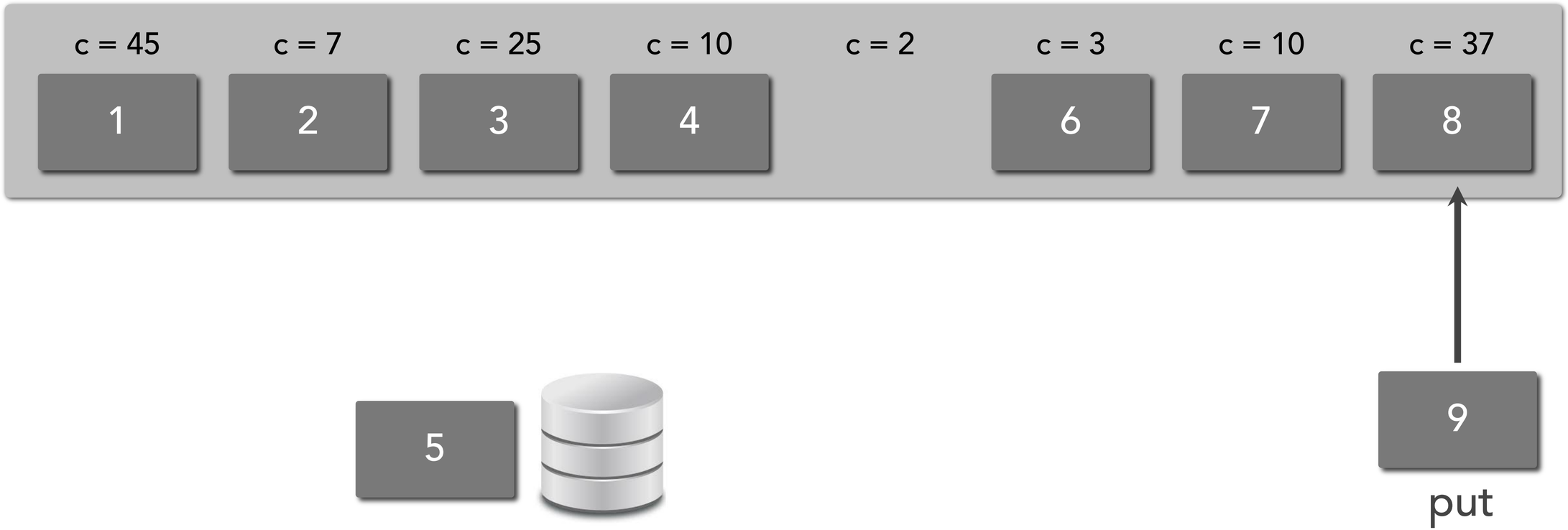
cache items are evicted that are accessed the least amount of times



cache eviction strategies

least frequently used policy (LFU)

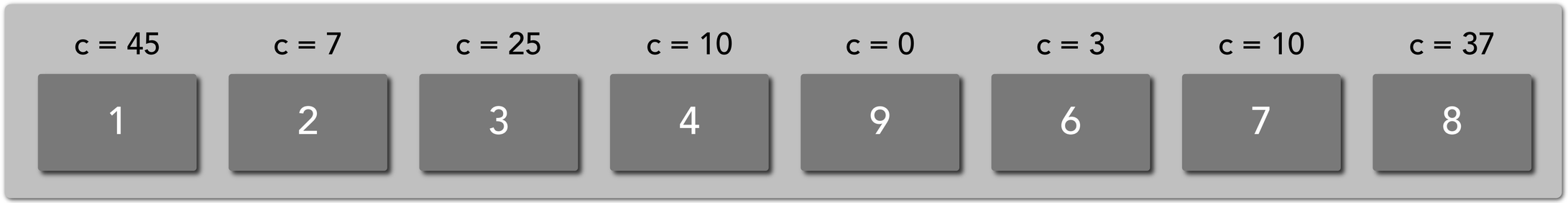
cache items are evicted that are accessed the least amount of times



cache eviction strategies

least frequently used policy (LFU)

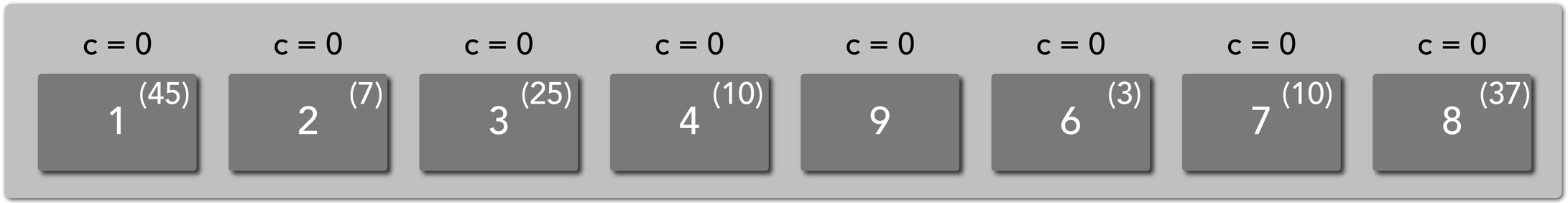
cache items are evicted that are accessed the least amount of times



cache eviction strategies

least frequently used policy (LFU)

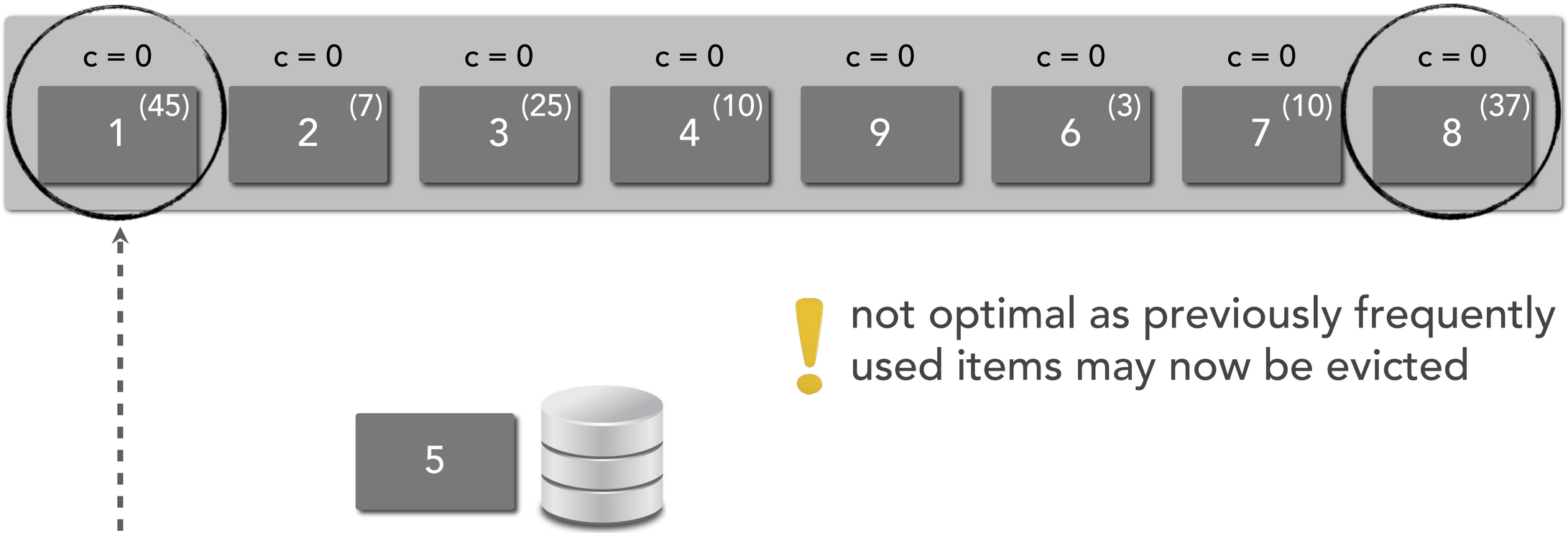
cache items are evicted that are accessed the least amount of times



cache eviction strategies

least frequently used policy (LFU)

cache items are evicted that are accessed the least amount of times



! not optimal as previously frequently used items may now be evicted

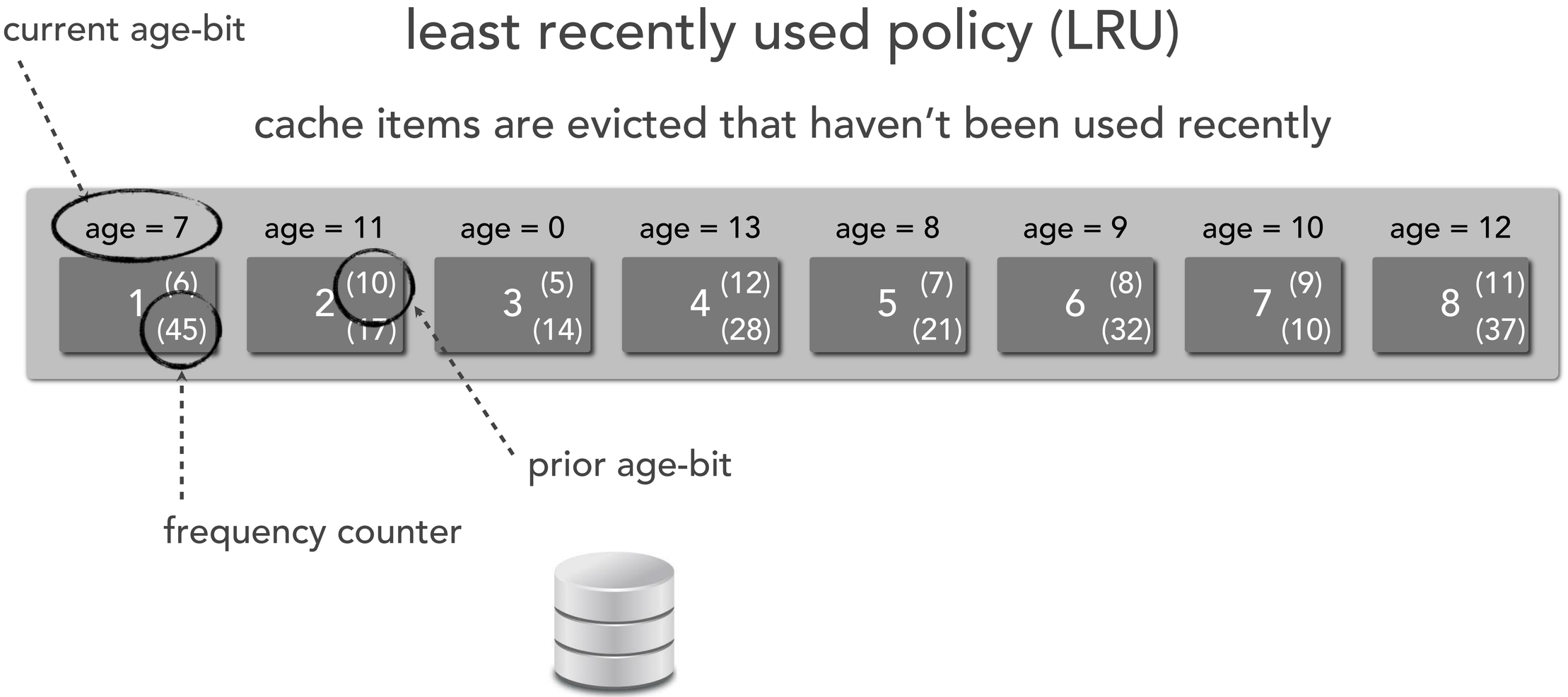
! low frequency ties are removed in FIFO order

Least Recently Used Policy (LRU)

cache eviction strategies

least recently used policy (LRU)

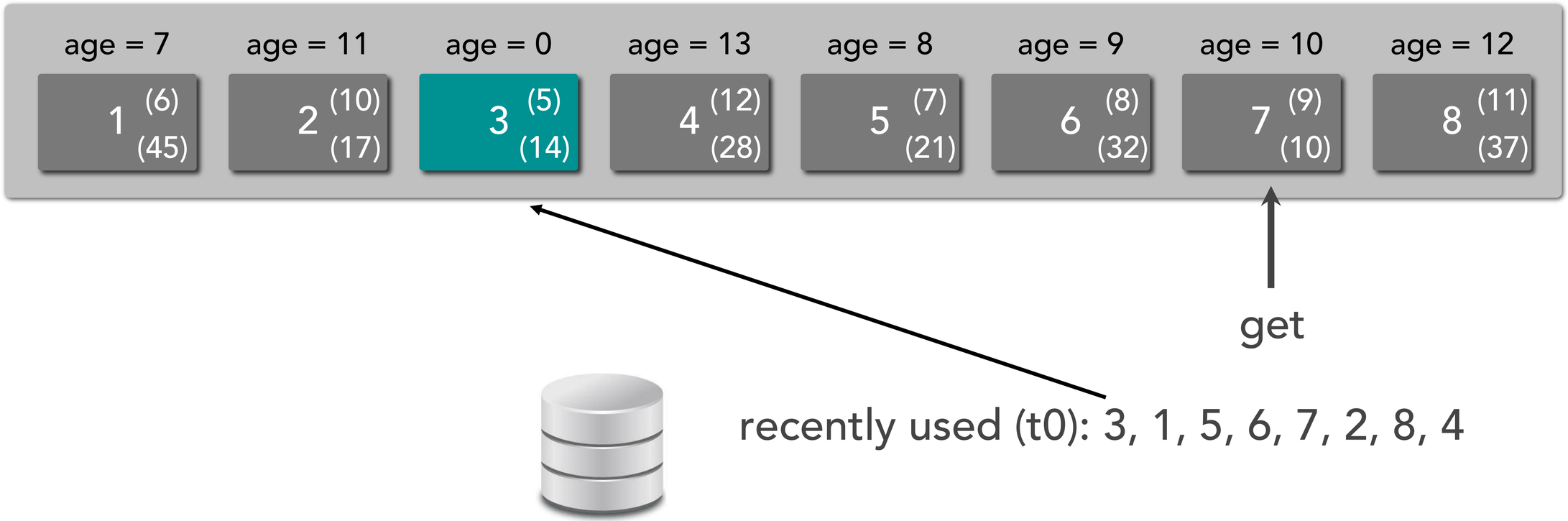
cache items are evicted that haven't been used recently



cache eviction strategies

least recently used policy (LRU)

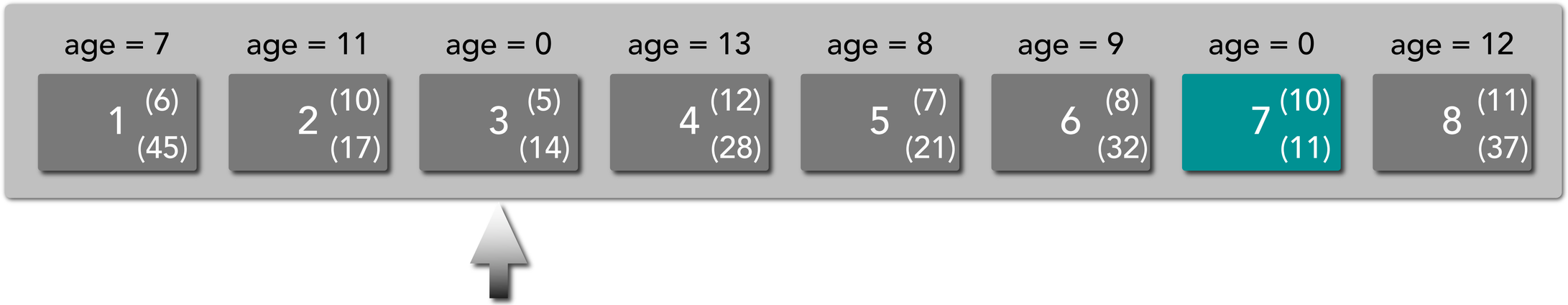
cache items are evicted that haven't been used recently



cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently

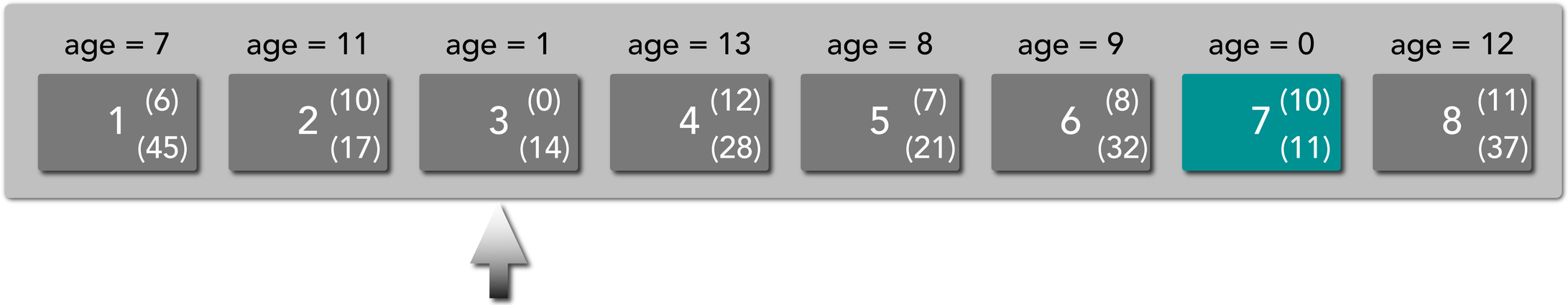


recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently

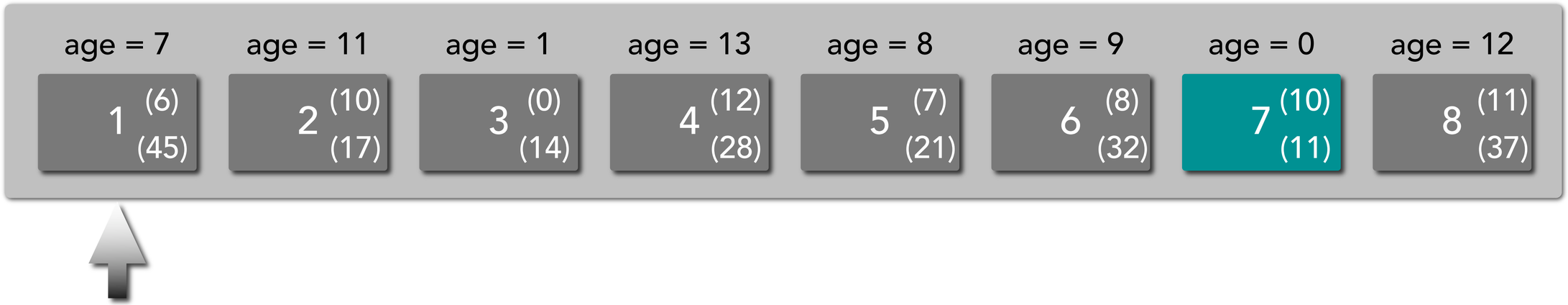


recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently

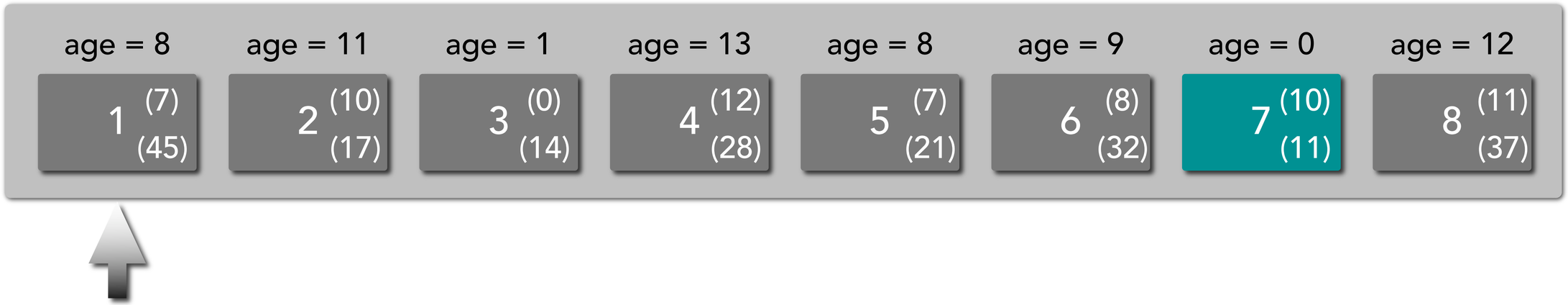


recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently



recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

cache eviction strategies

least recently used policy (LRU)

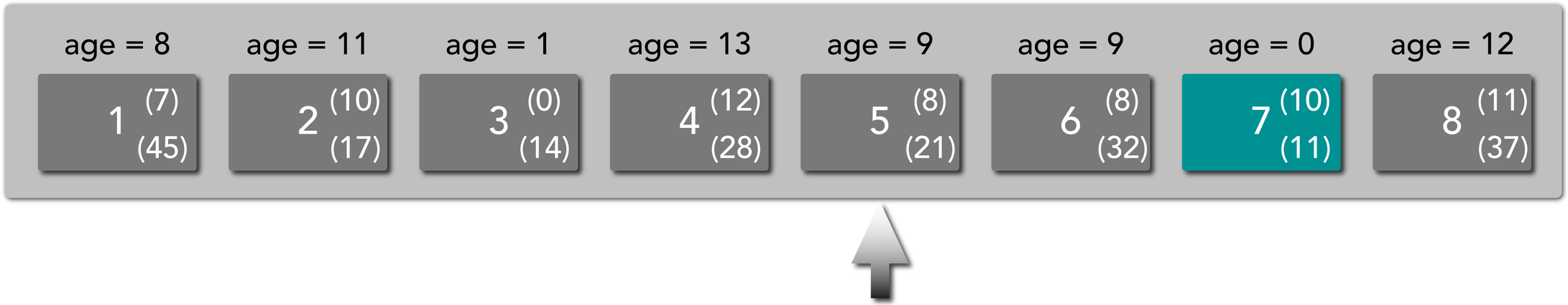
cache items are evicted that haven't been used recently

recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently

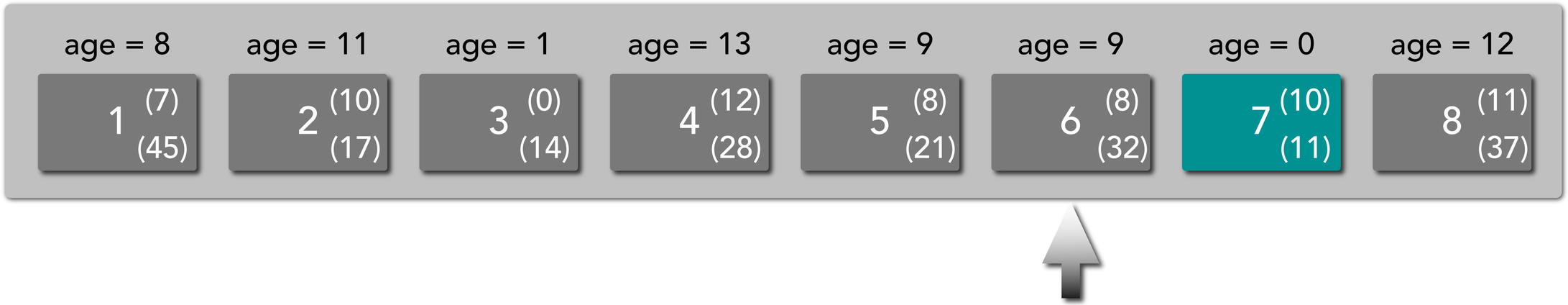


recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently

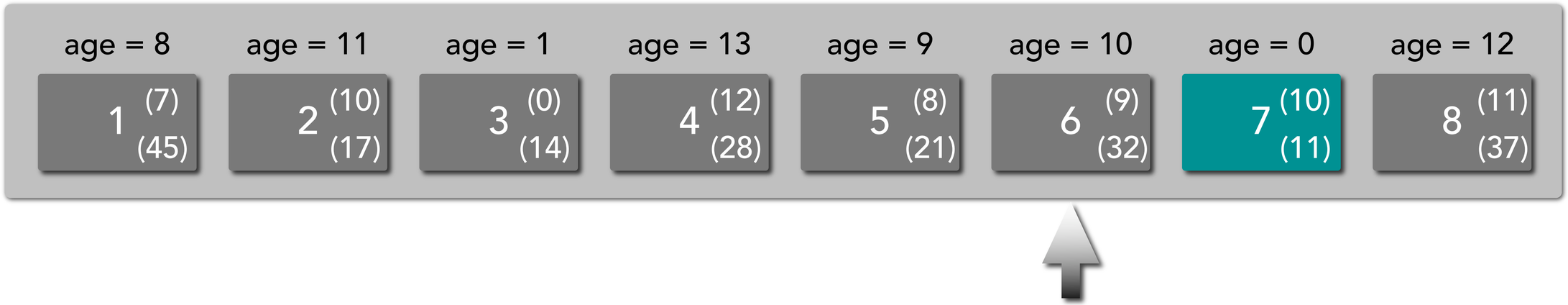


recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently

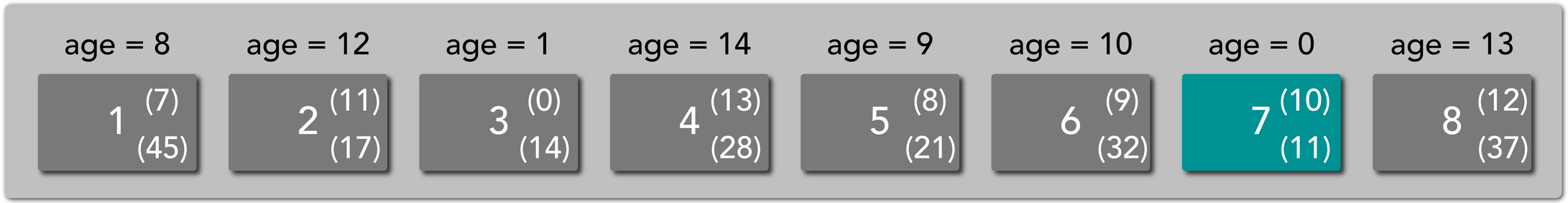


recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently



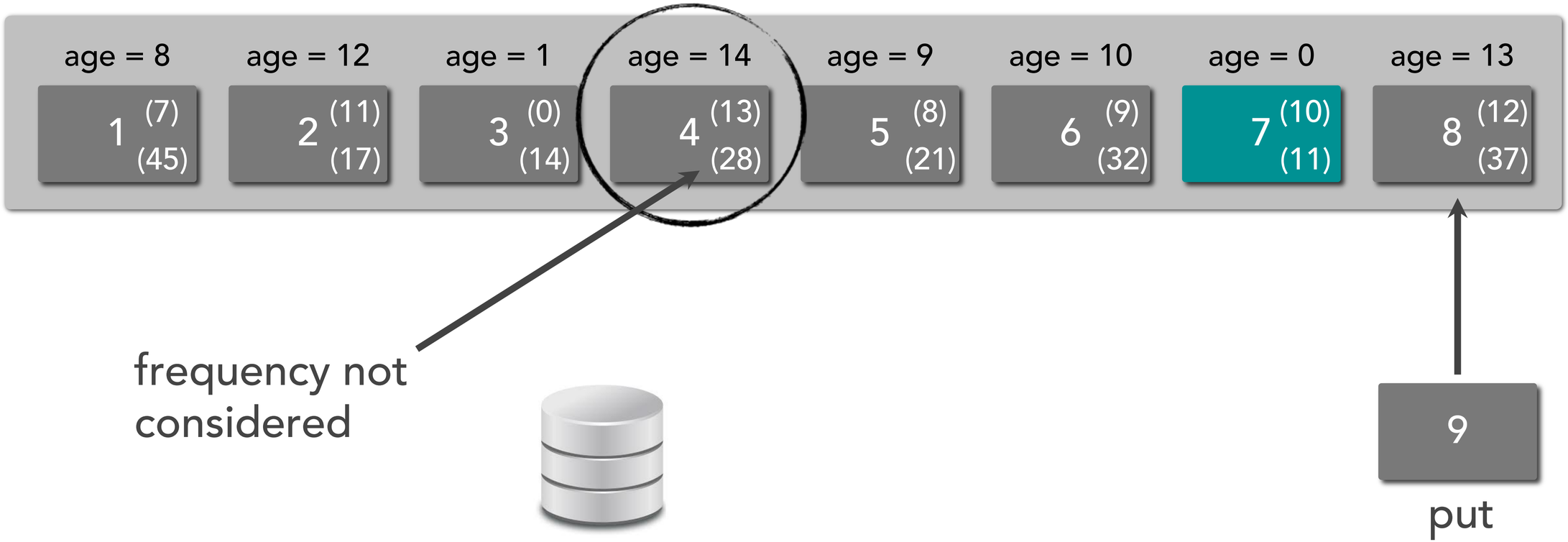
recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

recently used (t1): 7, 3, 1, 5, 6, 2, 8, 4

cache eviction strategies

least recently used policy (LRU)

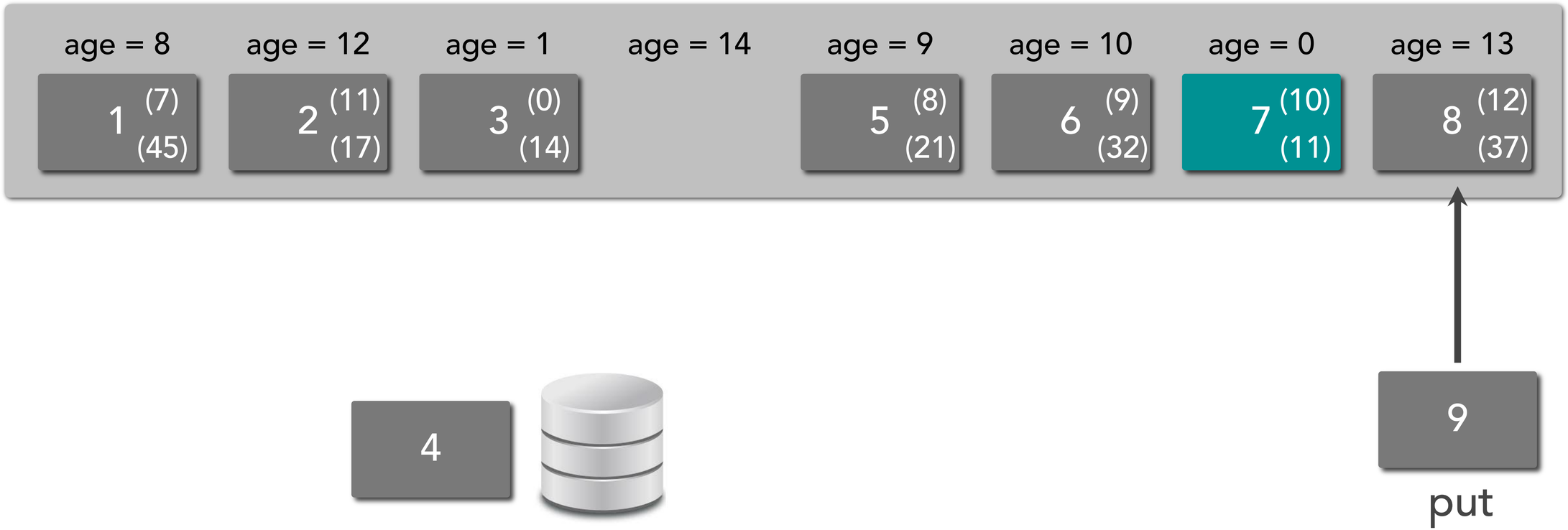
cache items are evicted that haven't been used recently



cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently



cache eviction strategies

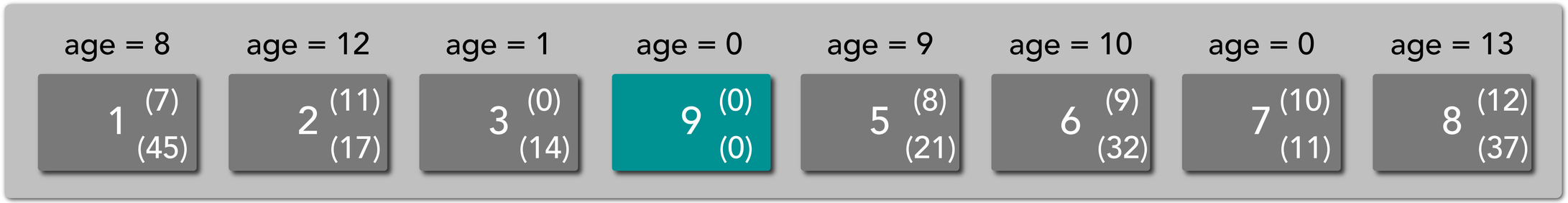
least recently used policy (LRU)

cache items are evicted that haven't been used recently

cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently



cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently

age = 9	age = 13	age = 2	age = 0	age = 10	age = 11	age = 1	age = 14
1 (8) (45)	2 (12) (17)	3 (1) (14)	9 (0) (0)	5 (9) (21)	6 (10) (32)	7 (0) (11)	8 (13) (37)

recently used (t0): 3, 1, 5, 6, 7, 2, 8, 4

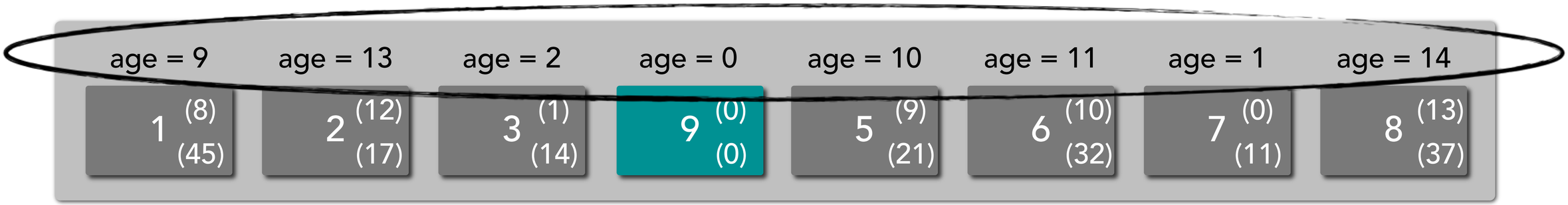
recently used (t1): 7, 3, 1, 5, 6, 2, 8, 4

recently used (t2): 9, 7, 3, 1, 5, 6, 2, 8

cache eviction strategies

least recently used policy (LRU)

cache items are evicted that haven't been used recently



expensive policy due to re-aging of each cache item (depending on algorithm used)

MRU (most recently used) is exactly the opposite - cache items are evicted that have been used recently

Random Replacement Policy (RR)

cache eviction strategies

random replacement policy (RR)

cache items are evicted randomly

cache eviction strategies

random replacement policy (RR)

cache items are evicted randomly

cache eviction strategies

random replacement policy (RR)

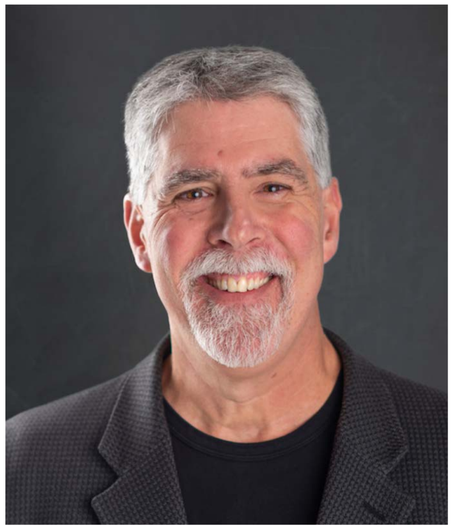
cache items are evicted randomly

- ✓ simple and fast strategy (no meta information is stored and maintained for each cache item)
- ✓ good strategy when access is consistent or non-deterministic across cache items

O'REILLY®

LIVE ONLINE TRAINING

Microservices Caching Strategies



Mark Richards

Independent Consultant

Hands-on Software Architect / Published Author

Founder, DeveloperToArchitect.com

http://www.wmrichards.com

https://www.linkedin.com/in/markrichards3

@markrichardssa