Hazelcast

(explained using Java client)

Agenda

- Why software caching?
- Introduction to hazelcast
- Getting started with hazelcast
- Data Partitioning
- Cluster setup
- Client Setup
- Distributed data structure
- Distributed query
- Distributed computing
- Loading and Storing Persistent Data
- Implementaion in #Fame

Why software caching?

Why software caching?

- Application performance:

i. Many concurrent users

i. Time and costs overhead to access application's data stored in RDBMS or file system

i. Database-access bottlenecks caused by too many simultaneous requests

Caching implementation challenges?

- **Memory size:**

i. is limited
ii. Can become acceptably huge

- **Synchronization complexity:**

i. consistency between the cached data state and data source's original data

- **Durability:**

i. Eviction policy, e.g :- LRU, LFU, FIFO
ii. Eviction percentage
iii. Expiration, e.g:- TTL, absolute/relative time-based expiration

- **Scalability**

- **Cache types**:

i. Local cache

i. Replicated cache

i. Distributed cache

i. Remote cache

i. Near cache

a distribued caching and IMDG solution

Hazelcast overview

- **IMDG** : Hazelcast is an open source In-Memory Data Grid (IMDG). It provides elastically scalable distributed In-Memory Computing.

- **Native implementation** : Hazelcast makes distributed computing simple by offering distributed implementations of many developer friendly interfaces from Java such as Map, Queue, ExecutorService, Lock, and Jcache.

- **Lightweighted** : It delivered as a compact library (JAR) without any external depenecy

- **Peer-to-peer** : There is no master and slave. there is no single point of failure. All nodes store equal amounts of data and do equal amounts of processing.

- **Scalable** : Hazelcast is designed to scale up to hundreds and thousands of nodes. Simply add new nodes and they will automatically discover the cluster and will linearly increase both memory and processing capacity.

- **Fast** : Since everything stores in-memory.

- **Redundant** : Hazelcast keeps the backup of each data entry on multiple nodes. On a node failure, the data is restored from the backup and the cluster will continue to operate without downtime.

Hazelcast use case

- Scale your application
- Share data across cluster
- Partition your data
- Send / receive messages
- Balance the load
- Process in parallal on many JVM

Who uses hazelcast ?

- and many more ...
- Every **sec** one Hazelcast instance starts around the globe

Getting started with hazelcast

Installation

Step 1 : Create a gradle project

Step 2 : Add following dependency in build.gradle

```
Dependencies {
----------
    compile 'com.hazelcast:hazelcast-all:3.6.3'
----------
}
```

Step 3:

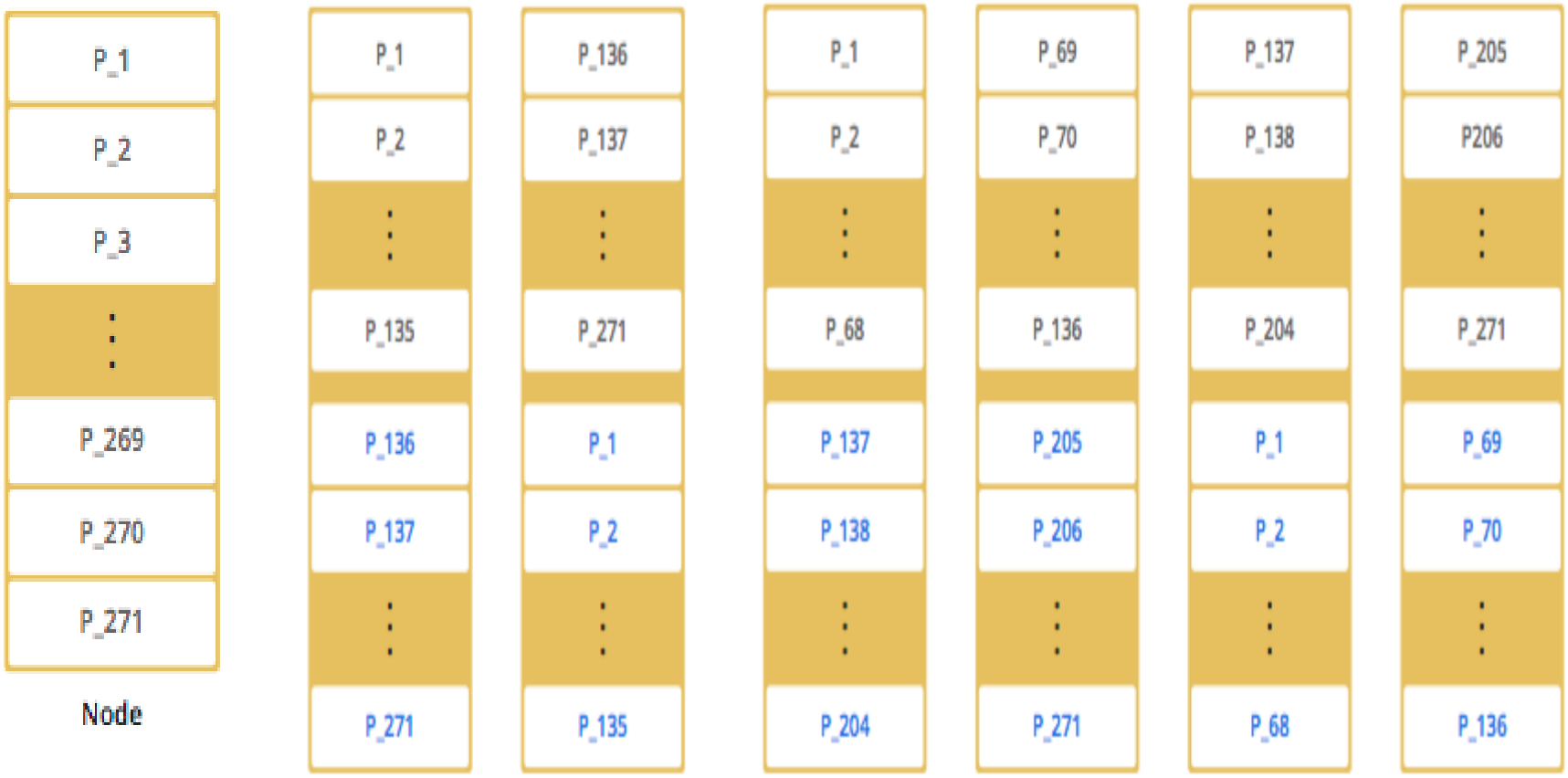
```
 // member instance
 Hazelcast.newHazelcastInstance()

//client instance
HazelcastClient.newHazelcastClient()
```

Data Partitioning

Hazelcast partition / sharding

- Partitioning



Partitin table

- hazelcast.partition.**count** = 271

- When a cluster member starts, a **partition table** is created within it. This table stores the partition IDs and the cluster members they belong, so can each member in a cluster knows where the data is.

- **The oldest member takes the responsibility** to sync all cluster members by sending partition table information in a regular interval, by default 15 sec

- We can configure this interval value by changing system property < **hazelcast.partition.table.send.interval** >

- **Pertiotionid**= MOD(hash result, partition count).

Hazelcast distributes data entries into the partitions using a hashing algorithm. Given an object key (for example, for a map) or an object name (for example, a topic or list):

i. the key or name is **serialized** (converted into a byte array),
ii. this byte array is **hashed**, and
iii. the result of the hash is **mod by the number of partitions**.

NOTE :- For all members in the cluster, the **partition ID for a given key** will always be the **same**.

- **Repartitioning** : Hazelcast performs the repartitioning in the following cases:

i. When a member joins to the cluster.
ii. When a member leaves the cluster.

Cluster setup

Configuration

- Network configuration :

A Hazelcast cluster is a network of cluster members that run Hazelcast. Cluster members (also called nodes) automatically join together to form a cluster. Hazelcast uses the following discovery mechanisms that the cluster members use to find each other:

i. Multicast (default mechanism)
ii. TCP
iii. EC2 Cloud
iv. jclouds R

TCP Discovery mechanism :

```
joinConfig.getMulticastConfig()
            .setEnabled(false);     // disabling multicasting

joinConfig.getTcpIpConfig()
            .setEnabled(true)  // enable TCP based discovery
            .addMember(<comma separated list of hosts>);
// add all members IP

Hazelcast.newHazelcastInstance(config);
```

- **Cluster group configuration** :

```
config.getGroupConfig()
                    .setName("intellimeet")
                    .setPassword("june");
```

- **Enabling lite members** :

Lite members are the Hazelcast cluster members that do not store data. These members are used mainly to execute tasks and register listeners, and they do not have partitions.

e.g :- config.setLiteMember(true);

Client setup

Configuration

```
ClientConfig clientConfig = new ClientConfig();

clientConfig.getGroupConfig()
        .setName("intellimeet")
        .setPassword("june");

clientConfig.getNetworkConfig()
        .addAddress(<member address>); // one or more

HazelcastClient.newHazelcastClient(clientConfig);
```

Operation mode

The client has two operation modes because of the distributed nature of the data and cluster,--

Smart client: In smart mode, clients connect to each cluster node. Since each data partition uses the well known and consistent hashing algorithm, **each client can send an operation to the relevant cluster node**, which increases the overall throughput and efficiency.
Smart mode is the default mode.

Dummy Client:
In dummy client mode, the client will only connect to one of the configured addresses. This **single node will behave as a gateway** to the other nodes.

Handling Failures

```
clientConfig.getNetworkConfig()
        .setConnectionAttemptLimit(10) // by default 2

        .setConnectionAttemptPeriod(1000) // by default 3000 ms

        .setConnectionTimeout(3000) // by default 5000 ms

        .setSmartRouting(true) // by default true


HazelcastClient.newHazelcastClient(clientConfig);
```

LifecycleEvent : fired when HazelcastInstance's state changes.

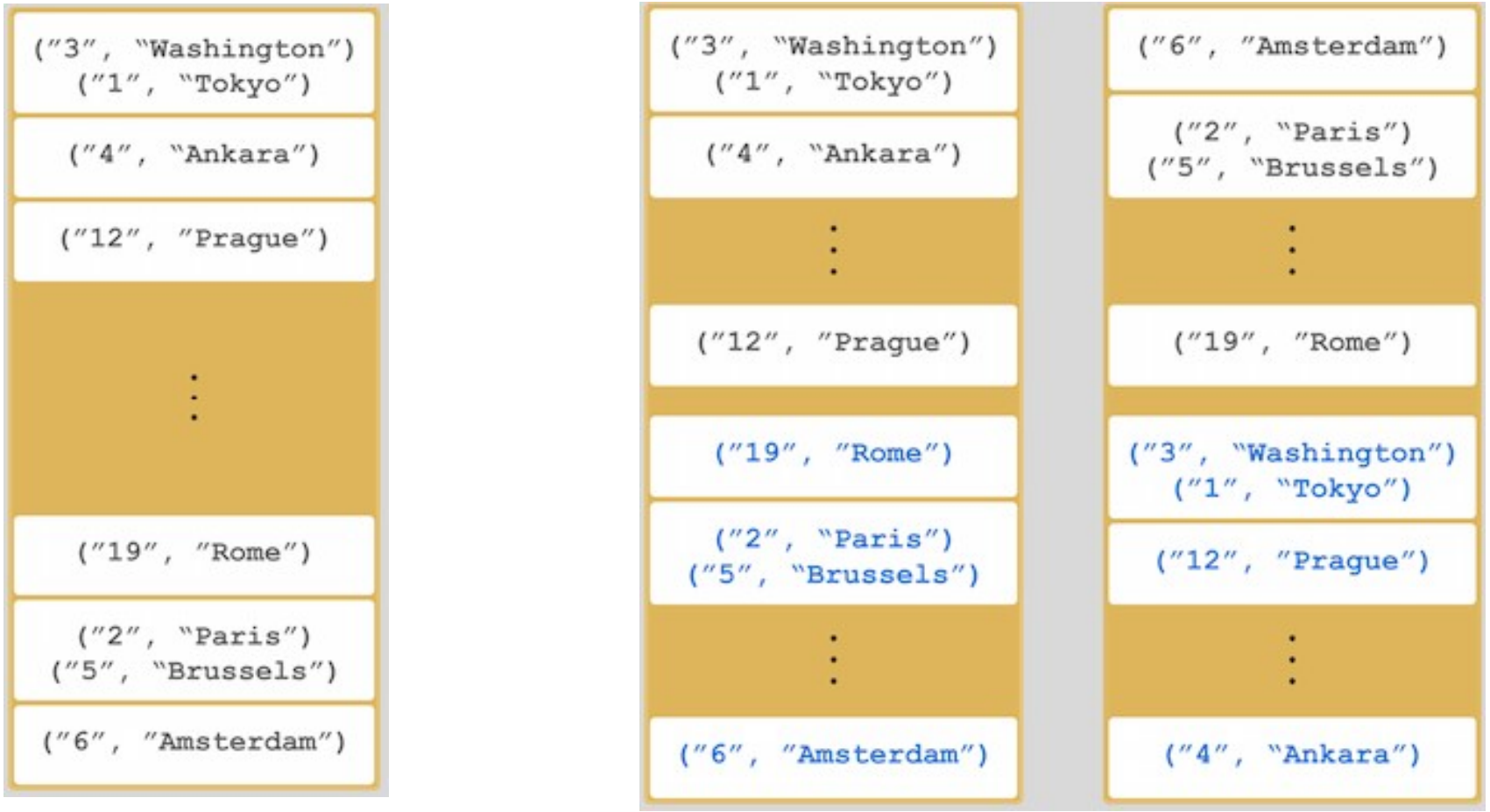
STARTING,
STARTED,
SHUTTING_DOWN,
SHUTDOWN,
MERGING,
MERGED,
CLIENT_CONNECTED,
CLIENT_DISCONNECTED

LifecycleService : allows you to shutdown, terminate, and listen to lifecycle events

Distributed data structure

Map

hzInstance.getMap("capitals").put("1", "Tokyo");



Pub /sub

Queue : is an implementation of BlockingQueue.
Ringbuffer : is a distributed data structure that stores its data in a **ring-like structure** with a given capacity.

- Each Ringbuffer has a **tail and a head**. The tail is where the items are added and the head is where the items are overwritten or expired.

- Unlike IQueue, **Ringbuffer does not remove the items**, it only reads items using a certain position.

- **Reads are cheap** since there is no change in the Ringbuffer

- Reads and writes can be batched to speed up performance.

Lock : ILock is the distributed implementation of java.util.concurrent.locks.Lock.

Others ...

- MultiMap, Replicated map
- Lock, Isemaphore, IcountDownLatch
- Set, List
- IAtomicLong, IAtomicRefernce, IdGenerator
- Topic, Reliable Topic

Distributed query

How Distributed Query Works

i. The requested predicate is sent to each member in the cluster.

i. Each member looks at its own local entries and filters them according to the predicate.

i. At this stage, key/value pairs of the entries are deserialized and then passed to the predicate.

i. The predicate requester merges all the results coming from each member into a single set.

Distributed query is scalable

- With every new member the partition count for each member is reduced and hence the time spent by each member on iterating its entries is reduced.

- The pool of partition threads evaluates the partition entries concurrently in each member

- The network traffic is reduced since only filtered data is sent to the requester.

- Hazelcast offers the following APIs for distributed query :

1) Criteria API, 2) Distributed SQL Query

- Criteria API methods :-

Equal, notEqual , instanceOf, like, ilike, greaterThan, greaterEqual, lessThan, lessEqual, between, in, isNot, regex

- Supporrtred SQL syntax :-

- AND/OR
- Equality: =, !=, <, <=, >, >=
- BETWEEN
- NOT
- IN / NOT IN
- LIKE / ILIKE

- Can be applied to map.values(), map.keySet(), map.entrySet()

- We can querying **inner objects** by 'innerObjRefName.attribute'

- Hazelcast also allows querying in **collections and arrays** by collectionObjName[index/any].attribute

- We can **sort** by passing a comparator to the predicate

- With **PagingPredicate** class, we can get a collection of keys, values, or entries page by page by filtering them with predicates and giving the size of the pages.

Index

- **Index** : When a query runs on a member, Hazelcast will iterate through the entire owned entries. This can be made faster by indexing the mostly queried fields.

- Indexing will add overhead for each write operation but queries will be a lot faster.
- Always configure index, before loading data
- If you have ranged queries such as age>30, age BETWEEN 40 AND 60, then you should set the ordered parameter to true.

```
mapCfg.addMapIndexConfig(new MapIndexConfig("age",true));
```

Distributed compute

Entry Processor

The distributed executor service is a distributed implementation of java.util.concurrent.ExecutorService.

It allows you to execute serializable tasks(Runnable / Callable) in the cluster.

Execution mode :-

- Execute on any member
- Execute on a specific member
- Execute on a key owner
- Execute on all or subset of cluster members

Executor Service

The distributed executor service is a distributed implementation of java.util.concurrent.ExecutorService.

It allows you to execute serializable tasks(Runnable / Callable) in the cluster.

Execution mode :-

- Execute on any member
- Execute on a specific member
- Execute on a key owner
- Execute on all or subset of cluster members

Loading and Storing Persistent Data

- **Cache access pattern**:

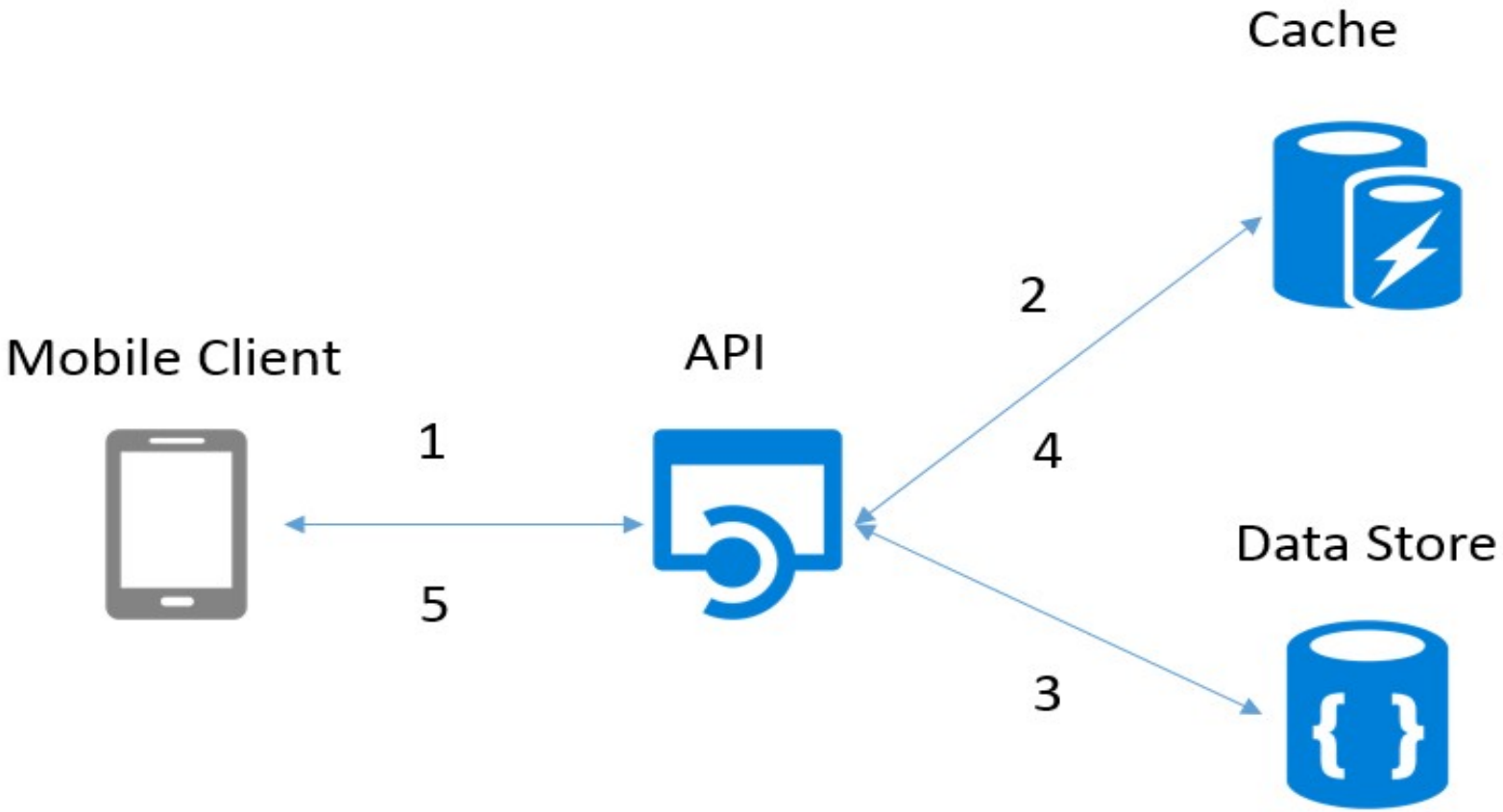
i. Cache Aside

i. Read-Through / Write-Through

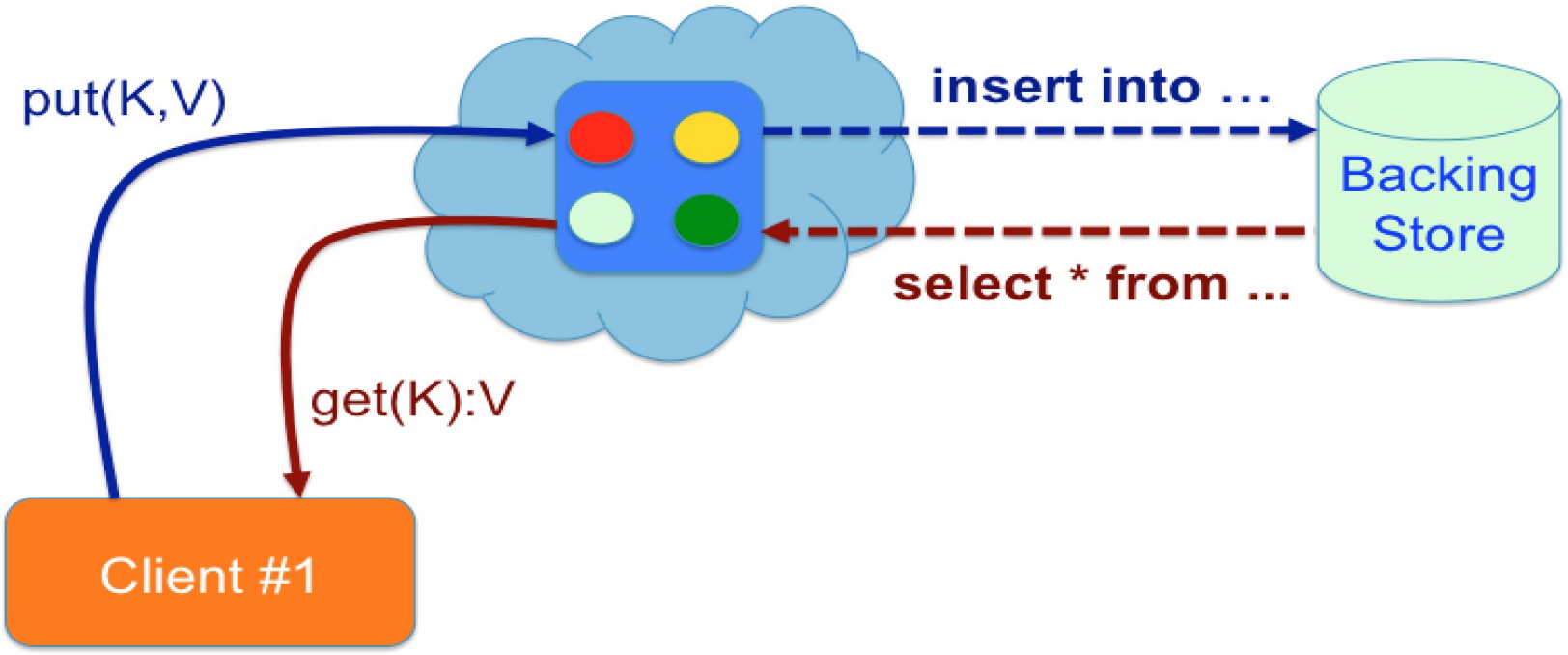
i. Write Behind

i. Refresh Ahead

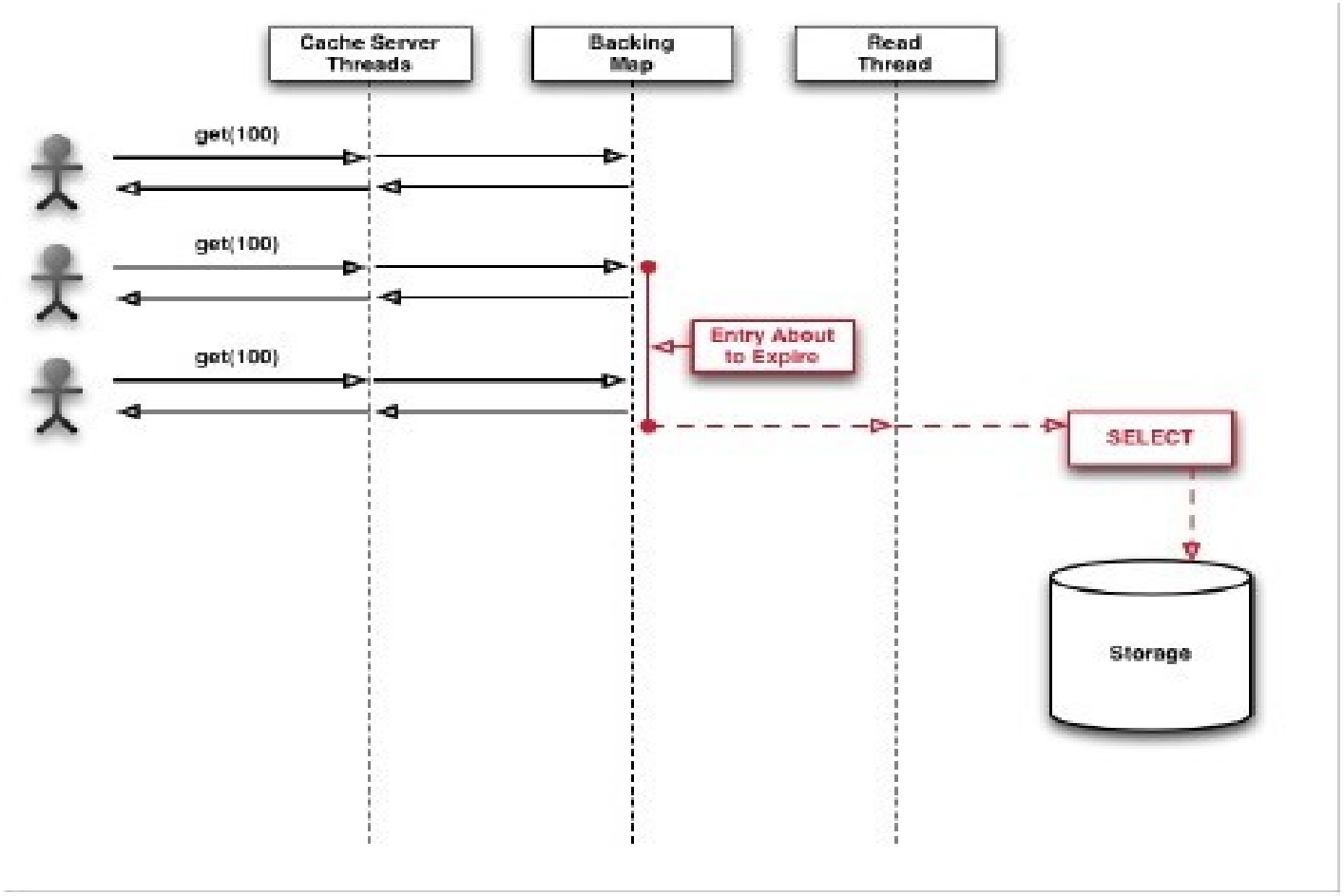
- Cache Aside Pattern



Read-Through/ Write-Through / Write-Behind



Refresh ahead



- Hazelcast allows us to **load and store the distributed map entries from/to a persistent data** store such as a relational database with the implementation of MapStore and MapLoader interfaces.

- Hazelcast supports **read-through, write-through, and write-behind** persistence modes

Read Through

- If requested an **entry (IMap.get()) that does not exist in memory, MapLoader's load or loadAll** methods will load that entry from the data store.

- It is **not allowed to make operations** like (Map/ Set / List / Semaphore etc) from MapStore interface methods in case of write through. Because writethrough map store operations run on partition thread, and using another partition based operation(like Containskey) can cause deadlock

Write Through

- In this mode, when the map.put(key,value) call returns:

1. **MapStore.store(key,value)** is successfully called so the entry is persisted.
2. **In-Memory entry is updated**.
3. **In-Memory backup** copies are successfully created on other cluster members (if backup-count is greater than 0).

- The same behavior goes for a **map.remove(key)** call.
- map.putTransient / map.evict / map.evictAll : **will not be called** to store/persist the entry
- entryView.getLastStoredTime : Returns the last time the value was flushed to mapstore.
- PostProcessingMapStore : only applicable to write through operation

Write Behind

- In this mode, when the map.put(key,value) call returns:

1. **In-Memory entry is updated**.
2. **In-Memory backup** copies are successfully created on other cluster members (if backup-count is greater than 0).
3. **The entry is marked as dirty** so that after write-delay-seconds, it can be persisted with MapStore.store(key,value) call.
4. For fault tolerance, **dirty entries are stored in a queue** on the primary member and also on a back-up member.

- **write-coalescing**: by default Hazelcast coalesces updates on a specific key, i.e. applies only the last update on it. By setting false this element to store all updates performed on a key to the data store

Thank you!

Questions??