In Depth Stream Processing With Hazelcast

By Randy May

Course Summary

The In-Depth Stream Processing Course is a 3 day hands-on experience intended to get Java developers ready to use the Hazelcast platform to build solutions that process and react to streaming data in real time. It consists of the following 3 elements:

Introduction to stream processing with Hazelcast Jet (5 hours)

> This session introduces stream processing concepts and the Jet Pipeline API for building stream processing applications and provides a hands-on introduction to the functional style of the Jet API.

Processing data in memory with IMDG (4 hours)

> This portion of the class provides an introduction to Hazelcast IMDG and more in-depth treatment of those elements of the product that are useful in a stream processing context.

Stream Processing Comprehensive Lab(12 hours)

> Students are introduced to a real life scenario involving processing streams of diagnostic and location data produced by a vehicle fleet. The student will use what they have learned about Jet and IMDG to build, deploy and update a stream processing system. The final solution is built over a day and a half in 10 guided labs and consists of multiple streams running in a multi-node Jet cluster.

hazelcast

Facility Prerequisites

Enough seating and table space to allow each participant and the instructor to sit and work on a laptop or personal computer.

Power for all participants.

Internet access for all participants.

> Please let us know if this is not possible. The labs require participants to download a significant amount of open source libraries, docker images, etc.. Using the internet is convenient and allows us to be responsive and distribute changes during the class. However, if internet access is out of the question then please discuss your preferred method of data transfer (e.g. USB) with the instructor ahead of time.

A projector of large display

A whiteboard or flip chart and markers

hazelcast

Attendee Prerequisites

Each attendee should have a laptop or personal computer (Windows or Mac)

Solid Java Programming Skills

An account in GitHub (if this is out of the question please discuss with the instructor us ahead of time).

Knowledge of Java 8 Streams is a plus but not required

Knowledge of Hazelcast IMDG or familiarity with other in memory grid products is a plus but not required.

Attendee Laptop Requirements

Windows or Mac

8G RAM

A Java 1.8 JDK (or higher)

Maven

Git

A Java IDE (IntelliJ, Eclipse, etc.)

> For Jet training, IntelliJ is *strongly preferred* due to the fact that its assistive features handle complex generic types better than those of Eclipse. The community edition is free and works very well.

Docker Desktop

> **Please install this before the first day.** Docker Desktop does not support some old versions of Windows 10.

hazelcast

Stream Processing Essentials

hazelcast

Lab Setup

Requirements

- Java 8 JDK or higher
- An IDE like IntelliJ IDEA, Eclipse, Netbeans
- Maven

Steps

1/ Download the lab source
github.com/hazelcast/hazelcast-jet-training

2/ Import it to your IDE

3/ Build it to download dependencies!

hazelcast

Course Objectives

- Be able to answer the questions …
  - What is stream processing ?
  - When is it useful ?
  - How does it differ from ETL ?

- Become familiar with the JET Pipeline API for defining streams.

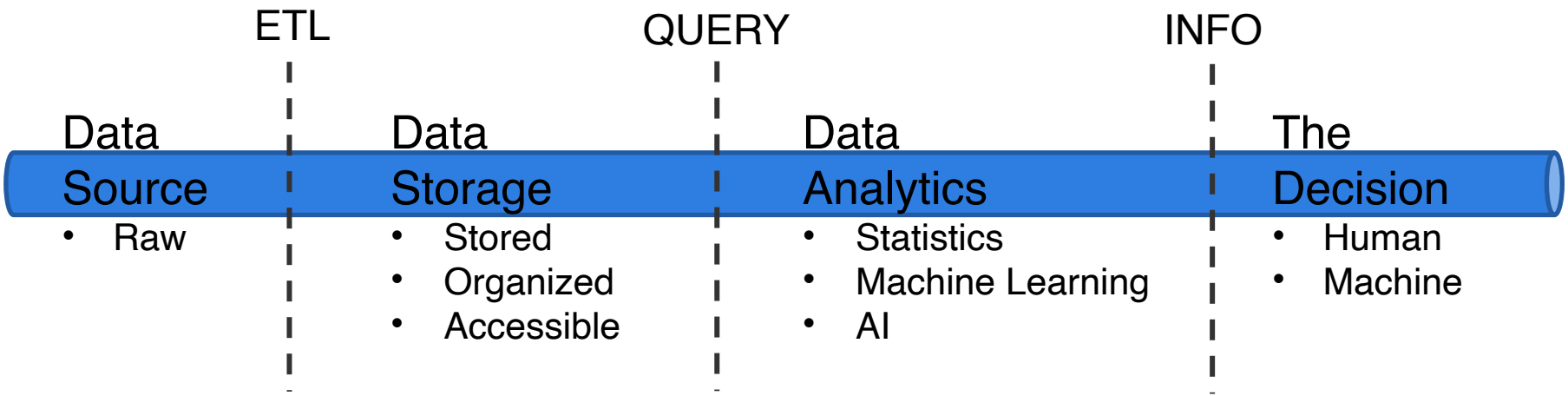
hazelcast

Stream Processing Defined

hazelcast

Data Processing Pipelines

- How to deal with scale?
- How to get the information as soon as possible?



hazelcast

Batch Processing

Use-cases

Post-mortem analysis

ML training/data science

Offline transaction processing

Extract Transform Load (ETL)

Tools

Hadoop

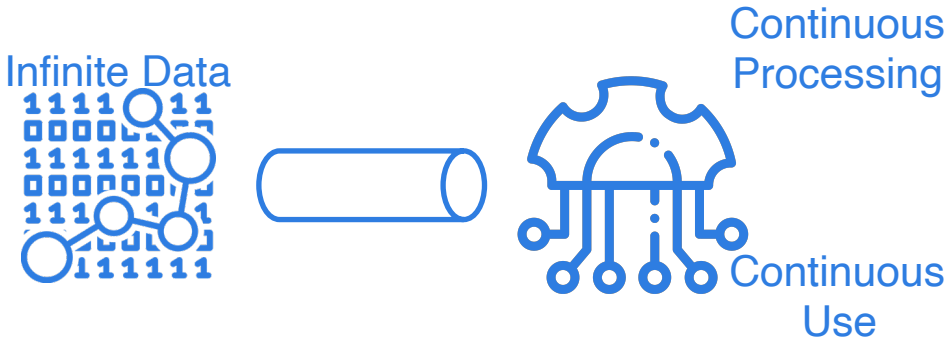
Hive for SQL people

Often custom

hazelcast

Stream Processing

Real world data doesn't come in batches!



Characteristics

Querying made pro-active

Pre-processing for database

Benefits

Low latency

Continuous programming model

Event time, not ingestion time

Batch is just a finite stream

hazelcast

A Simple Scenario

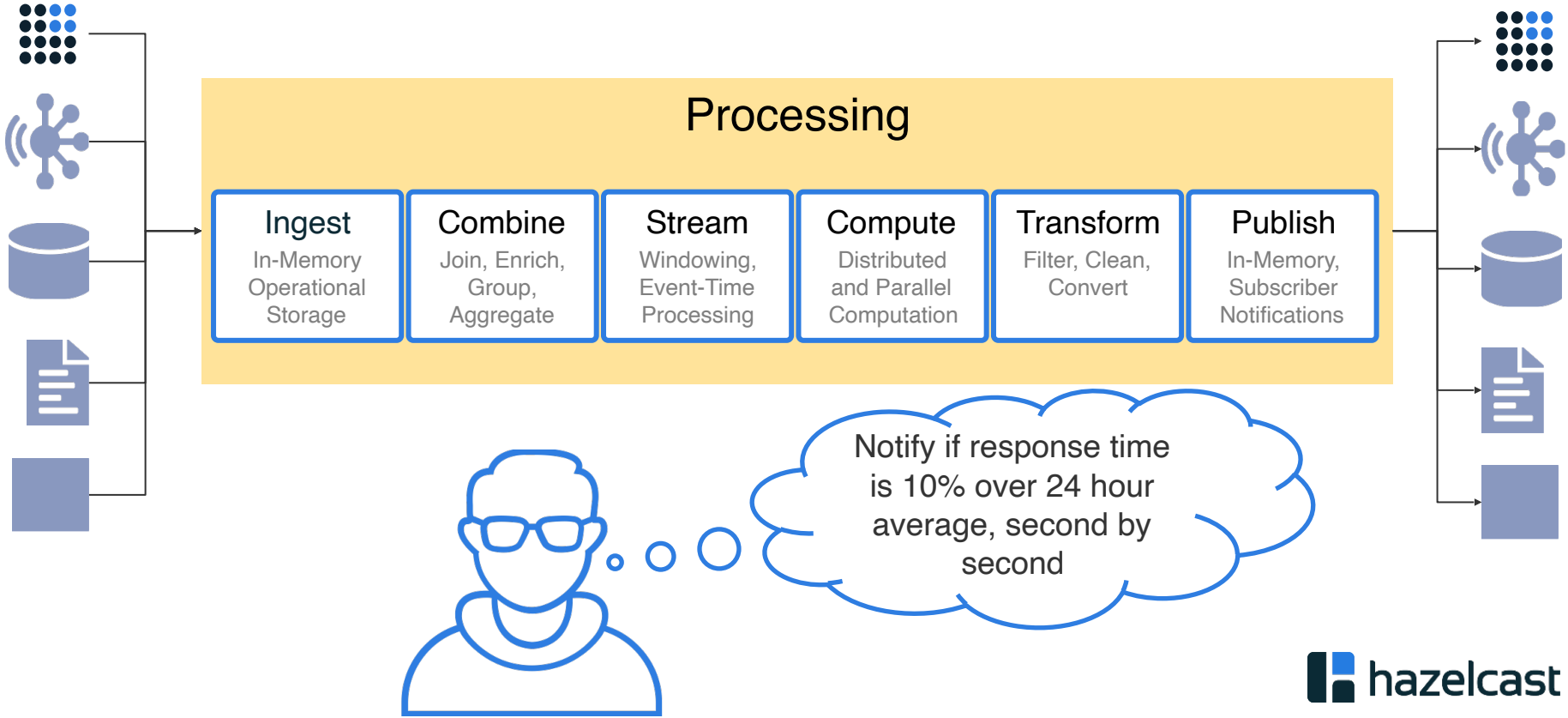
System health, app logs, etc.

Notify if response time is 10% over 24 hour average, second by second

Data Source → Data Storage → Data Analytics → The Decision

hazelcast

Streaming Pipelines



hazelcast

Use Case: Continuous ETL

- ETL - Data Integration
- ETL in the 21st Century
  - Maintains the derived data (keeps it in sync)
  - Performance - adapt data to various workloads
  - Modularization - microservices own the data
- Why continuous ETL?
  - Latency
  - Global operations (no after hours)
  - Continuous resource consumption

Use Case: Analytics

- Real-time dashboards
- Stats (gaming, infrastructure monitoring)
- Prediction - often based on algorithmic prediction (push stream through ML model)

hazelcast

Use Case: Real Time Decision Making

- Streams are ideal for *operationalizing* an ML model
- Real time fraud check.
- Churn prediction
- Fleet monitoring

hazelcast

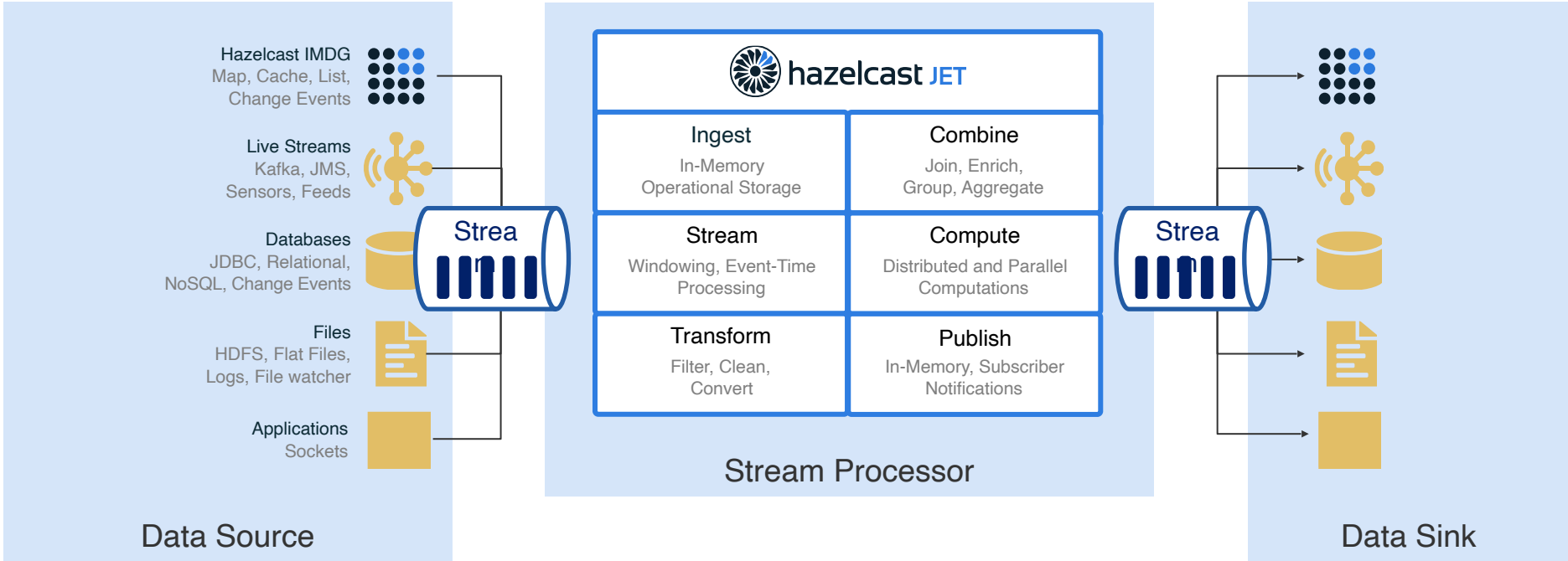
Key Points

- Stream processing is an evolution of the traditional data processing pipeline
- Continuous programming model for infinite data sets
- Pre-process data before storing / using it -> reduces access times when you need the results
- Processed results kept in sync with latest updates

How Hazelcast Does Stream Processing

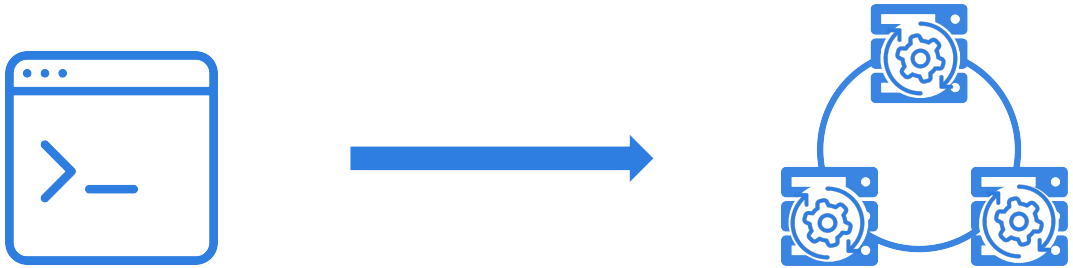
hazelcast

The Big Picture



hazelcast

Pipeline and Job



Pipeline

- Declaration (code) that defines and links sources, transforms, and sinks
- Platform-specific SDK (Pipeline API in Jet)
- Client submits pipeline to the Stream Processing Engine (SPE)

Job

- Running instance of pipeline in SPE
- SPE executes the pipeline
  - Code execution
  - Data routing
  - Flow control
- Parallel and distributed execution

hazelcast

Declarative Programming Model

- Compare counting words in Java 8
  - Imperative - Iterators (user controls the flow)

```
final String text = "...";
final Map<String, Long> counts = new HashMap<>();

for (String word : text.split("\\W+")) {
    Long count = counts.get(word);
    counts.put(count == null ? 1L : count + 1);
}
```

  - Declarative – Java Streams (code defines logic, not flow)

```
Map<String, Long> counts = lines.stream()
    .map(String::toLowerCase)
    .flatMap(line -> Arrays.stream(line.split("\\W+")))
    .filter(word -> !word.isEmpty())
    .collect(Collectors.groupingBy(word -> word, Collectors.counting()));
```

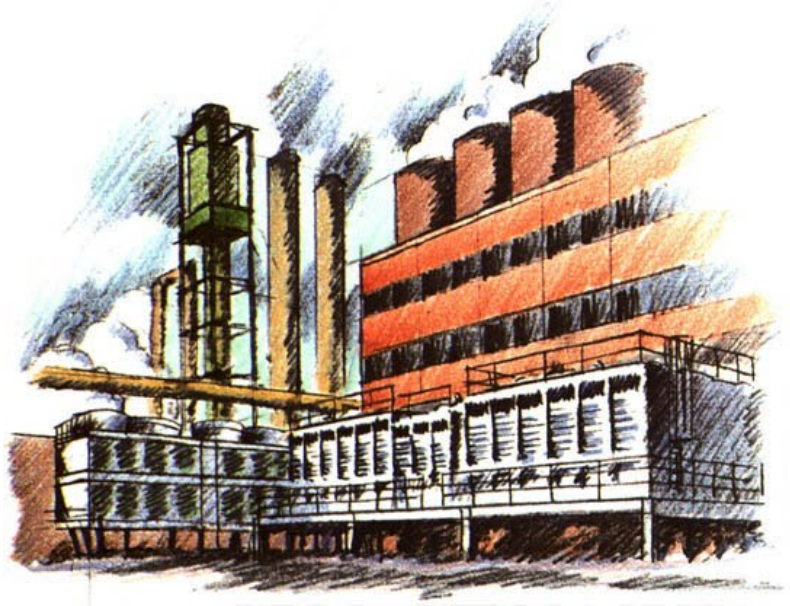
hazelcast

Why Pipelines Use Declarative

- Developer defines “what”
- SPE handles the “how”
  - Data routing
  - Partitioning
  - Invoking pipeline stages
  - Running your pipeline in parallel

hazelcast

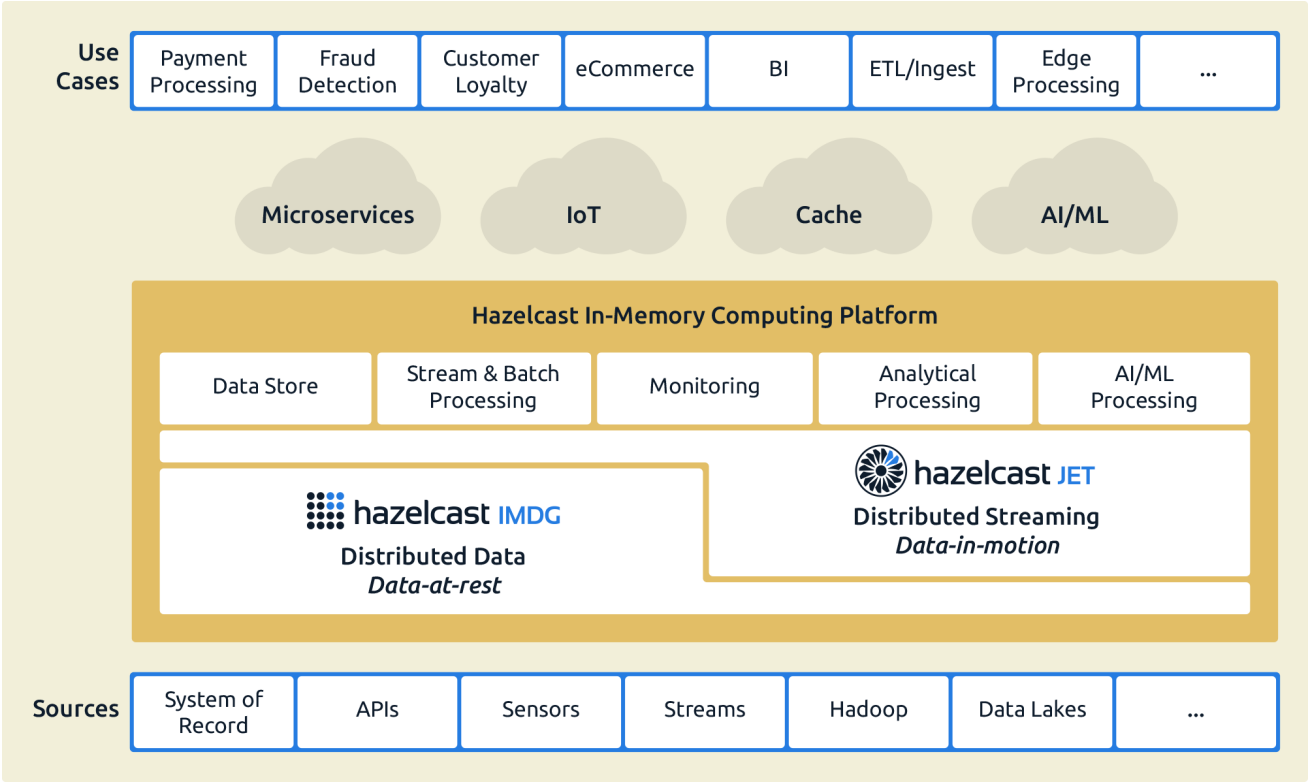
An Analogy… Code as Factory



- Stream processing pipelines are data “factories”
  - Raw data in
  - Usable data out
  - One *or more* intermediate stages
- Design first, design second, code third

hazelcast

Hazelcast In-Memory Computing Platform

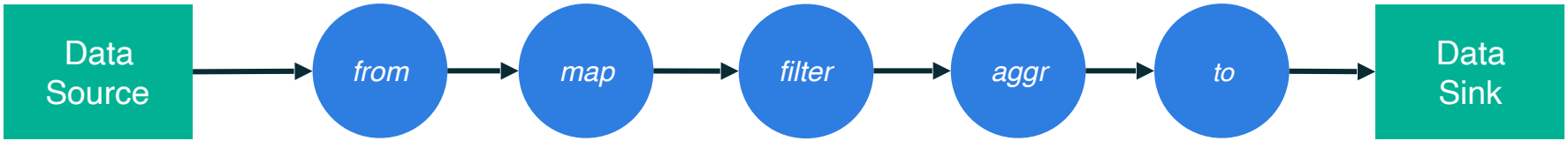


hazelcast

Jet Does Distributed Parallel Processing

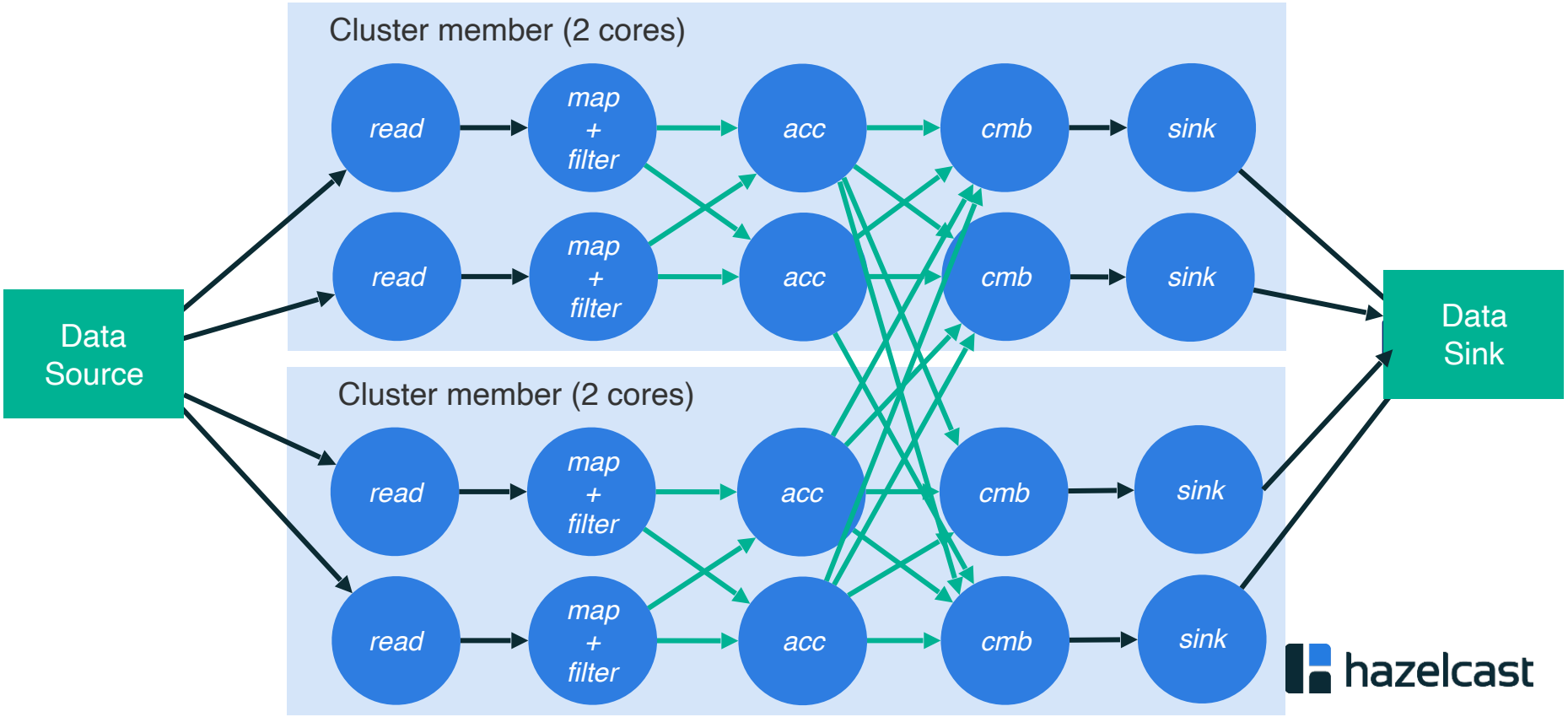
- Jet translates declarative code to a DAG

```
Pipeline p = Pipeline.create();
p.drawFrom(Sources.<Long, String>map(BOOK_LINES))
    .flatMap(line -> traverseArray(line.getValue().split("\\W+")))
    .filter(word -> !word.isEmpty())
    .groupingKey(wholeItem())
    .aggregate(counting())
    .drainTo(Sinks.map(COUNTS));
```

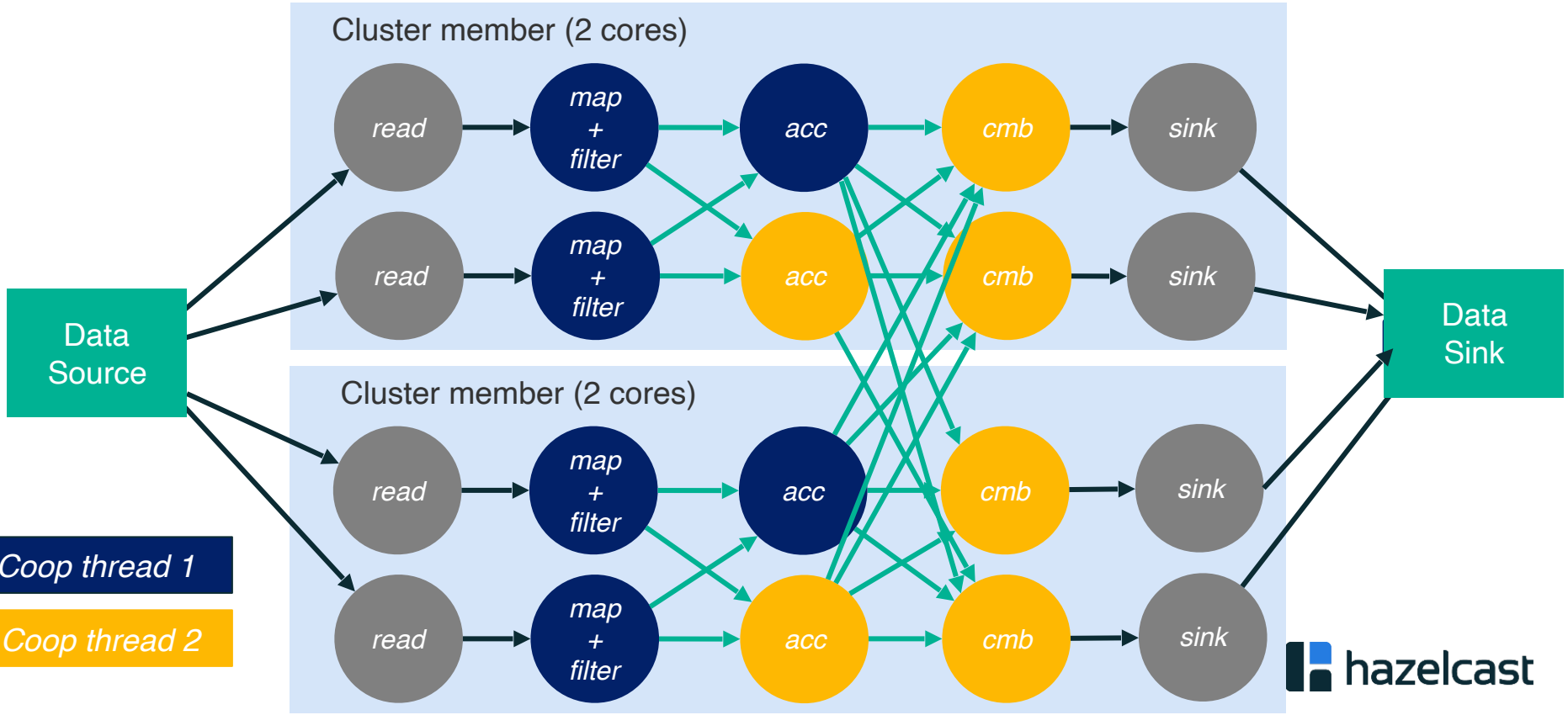


hazelcast

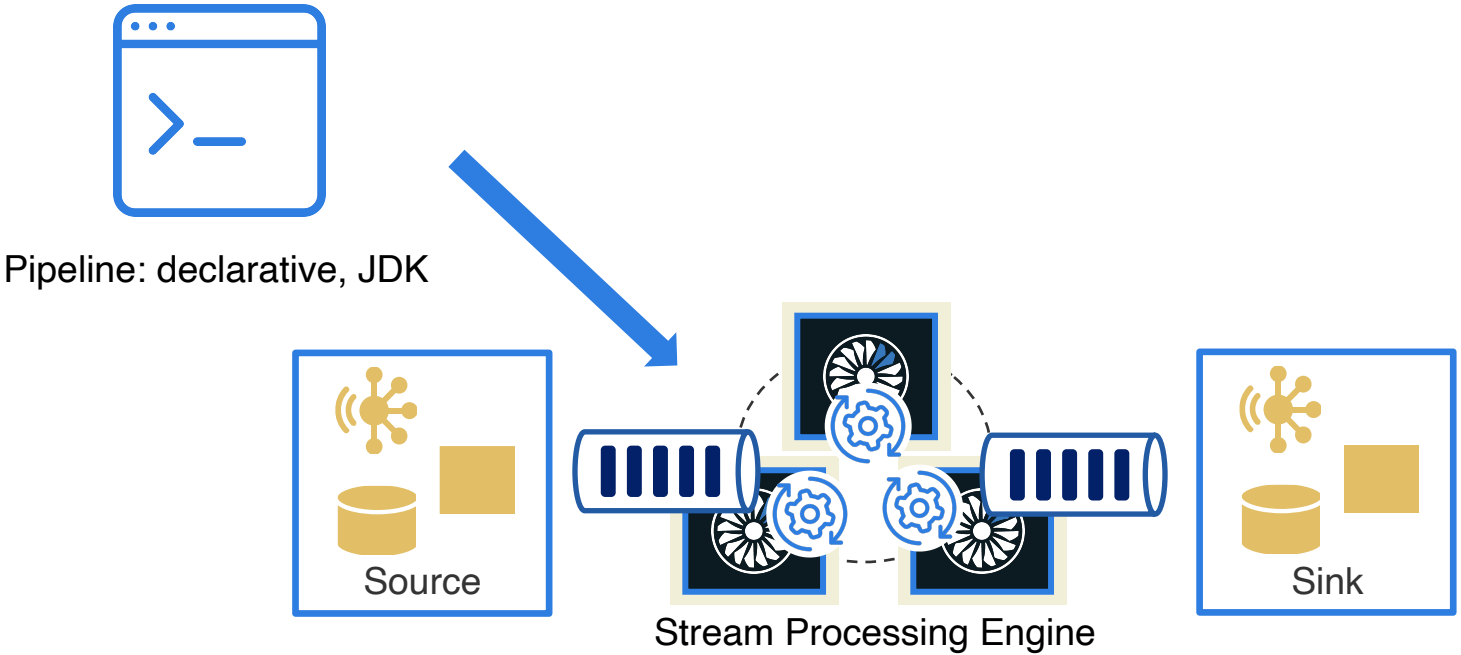
Jet Does Distributed Parallel Processing



Cooperative Execution



All the Building Blocks



hazelcast

Introducing the Jet Pipeline API

hazelcast

Pipeline API

- Remember our building blocks?

- API follows same pattern

```
Pipeline p = Pipeline.create();

//specify source(s) for stream
p.drawFrom(Sources.<String>list("input"))

//specify transform operation(s)
        .map(String::toUpperCase)

//specify sink(s) for transform results
        .drainTo(Sinks.list("result"));
```

hazelcast

Sources and Sinks

- drawFrom and drainTo require a source as a parameter

```
p.drawFrom( Source definition )
```

- Libraries with sources and sinks available out-of-the-box

```
com.hazelcast.jet.pipeline.Sources

com.hazelcast.jet.pipeline.Sinks
```

- SourceBuilder and SinkBuilder classes for custom sources/sinks

hazelcast

We'll Start Simple

- Turn off event-time processing for now

```
p.drawFrom( Source definition )
        .withoutTimestamps()
```

  - We'll explain this when talking about windowing and event-time processing

hazelcast

Basic Transformation Operations

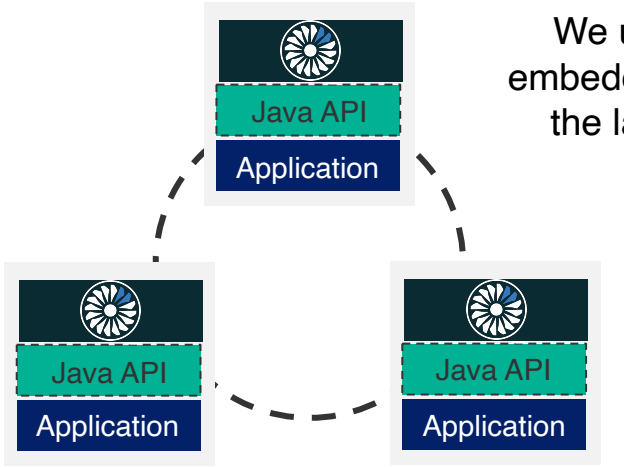
- Filter: discards items that don’t match the predicate
  - Data cleaning, reformatting, etc.
- Map: transforms each item to another item
  - Trim records to only required data
- flatMap: transforms each item into 0 or more output items
  - Example: separate a line of text into individual words

Transformation Examples

- NewPowerCo has installed smart meters at all homes in a service area
  - Meters stream constant usage data: address, region code, kw/hour consumption, etc.
- Filter example
  - Keep all records exceeding a given kw/hour rate
- Map example
  - Strip addresses – keep only region code and kw/hour data
- flatMap example
  - Separate record with multiple measurements to multiple records

Interacting with the Cluster

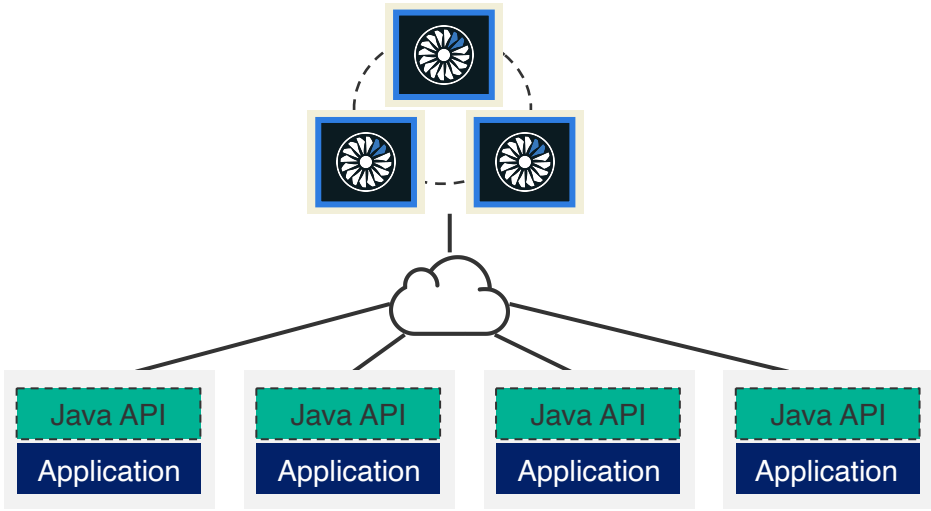
Embedded



We use embedded in the labs

```
// Create new cluster member

JetInstance jet = Jet.newJetInstance();
```

Client/Server




```
// Connect to  running cluster

JetInstance jet = Jet.newJetClient();
```

hazelcast

Submitting the Pipeline

```
public static void main (String[] args) {
    JetInstance jet = Jet.newJetInstance();
    Pipeline p = buildPipeline();
    try {
        jet.newJob(p).join();
    }
    finally {
        jet.shutdown();
    }
}
```

- Use the cluster handle
- Submit the job
  - Submit and return

    ```
    jet.newJob(pipeline);
    ```

  - Submit and block

    ```
    jet.newJob(pipeline).join();
    ```

- Stop cluster when processing is done

hazelcast

Lab 1: Filter Records from Stream

- Step 1: log new lines
  - file watcher (infinite), it's like `tail -f`
  - log sink
  - manually adding data to the directory to see the results
- Step 2: Filter out "word" records

hazelcast

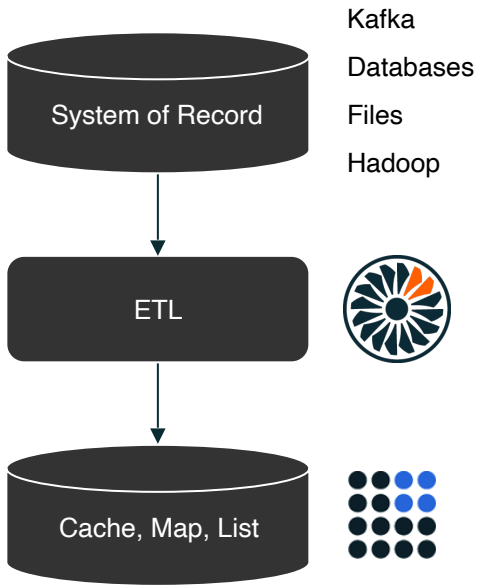
What We Learned…

- The Pipeline
- Basic connectors: fileWatcher, logging sink
- Basic operators: filter, map, flatMap
- Lambdas (serializable)
- Embedded (in-process) JetInstance
- Obtaining a cluster handle and submitting the job

hazelcast

Great for Moving Data to IMDG

- Frequent solutions:
  - Custom data pumps
  - ETL tools with GUI
- Jet brings:
  - Speed (works in parallel)
  - Fault Tolerance
  - Focus on business logic, not on infrastructure / integration
  - Java API



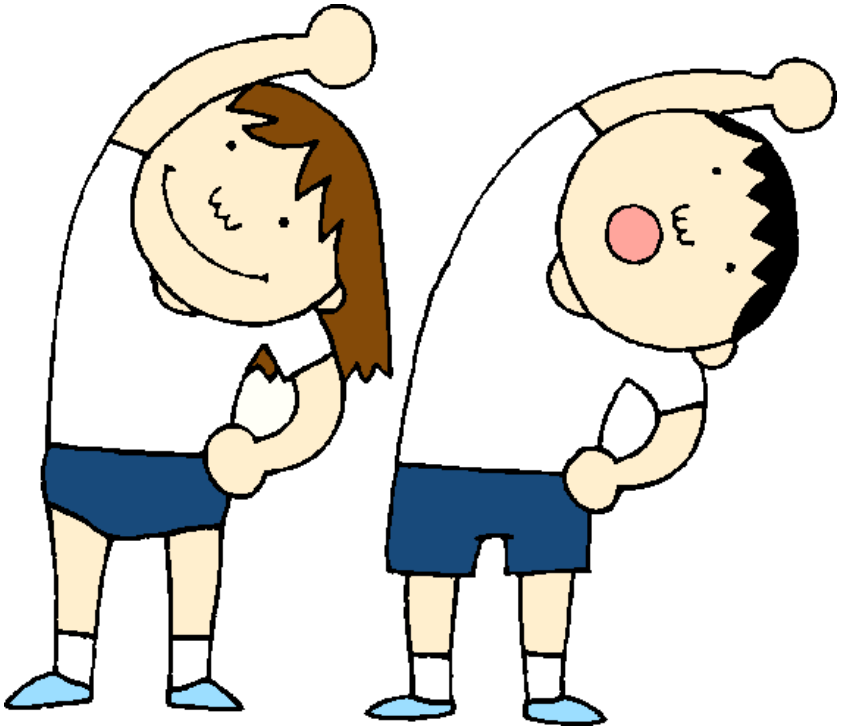
hazelcast

Honorable Mentions

- Connectors in Jet Library
  - Hazelcast, Journal, Kafka, HDFS, JMS, JDBC, Elasticsearch, MongoDB, InfluxDB, Redis, Socket, File
  - The list is growing fast!
- Custom connectors
  - Builders: see the code samples
  - Examples: Twitter Source, REST JSON Service Source, Web Camera Source
- Pipeline can have multiple sources, sinks and branches

hazelcast

Stretch/Bio Break



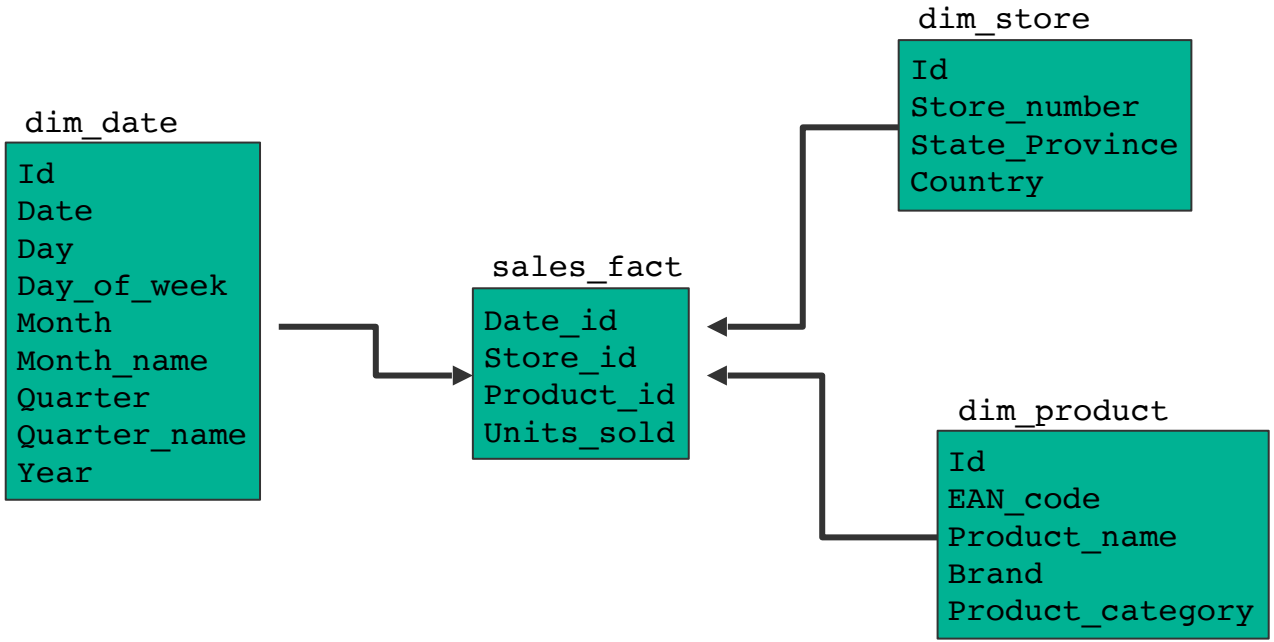
15 Minute Break

hazelcast

Enrichment

Enriching the Stream

- Do a lookup to enrich the stream
- Similar to relations in RDBMS



hazelcast

Enrichment Options

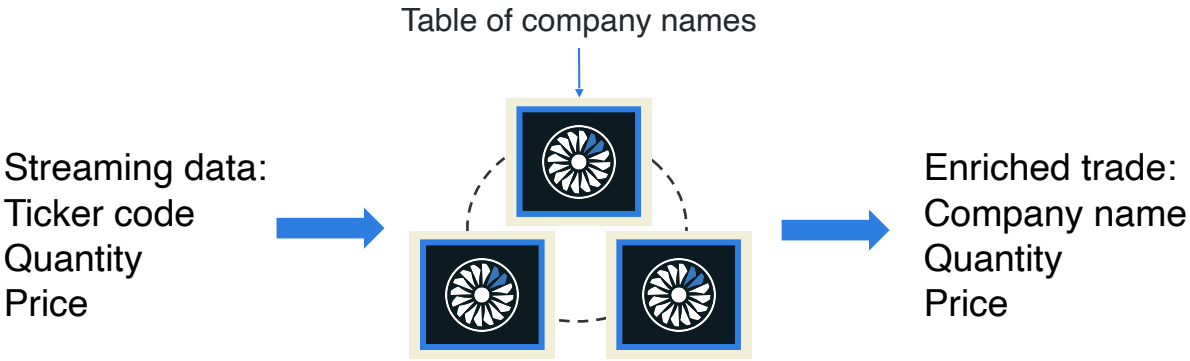
- Local map
  - Advantage: fast, hard-coded
  - Disadvantage: change = redeploy
- Remote service lookup (database, RPC call)
  - Advantage: always up to date
  - Disadvantage: slow retrieval
- IMap in the cluster
  - Hazelcast Jet contains rich distributed cache (thanks to embedded IMDG)
  - Distributed and elastic, scales with cluster size
  - Read/Write through for databases

Enrichment API in Jet

```
items.mapUsingIMap(
      // the name of the lookup cache in Hazelcast
      "enriching-map",
      // how to obtain foreign key from the stream?
      item -> item.getDetailId(),
      // how to merge the stream with looked up data
      (Item item, ItemDetail detail)
            -> item.setDetail(detail)
 )
```

hazelcast

Lab 2: Enrich the Stream



- Enrich a trade stream using the cache
  - Randomly-generated “trade“ stream
  - Replace ticker code with company name
    - IMap (distributed map) caches name table

hazelcast

Lab 2: Enrich the Stream

- Enrich a trade stream using the cache
  - Use the Trade generator as a source
    - sources.TradeSource.tradeSource()
  - Trade contains the symbol - a foreign key, referring to a company
  - Lookup company name
  - Convert Trades to EnrichedTrades by enriching it with company name
  - IMap (distributed map) used for caching

What We Learned…

- Trading off flexibility and performance
  - Enrich from local memory
  - Remote lookup
  - Enrichment from a cache
- https://blog.hazelcast.com/ways-to-enrich-stream-with-jet/
- Operators: hashJoin, mapUsingImap, mapWithContext
- Jet Cluster provides powerful caching services

hazelcast

Honorable Mentions

- Hazelcast is a powerful distributed in-memory framework
  - Caching: read/write-through, JCache support
  - Messaging: buffer to connect multiple Jet jobs (journal)
  - Storage: in-mem NoSQL for low-latency use-cases (source, sink)
  - Coordination: unique
  - Polyglot (Java, Scala, C++, C#/.NET, Python, Node.js, Go)
    - These are IMDG clients – Jet pipelines are Java-only

hazelcast

Aggregations and Stateful Streaming

hazelcast

Aggregations (Stateful Processing)

- Combine multiple records to produce a single value
  - Examples: sum, average, min, max, count
- Stateful processing
- Concerns
  - Bounding the scope: batch, rolling and windowed aggregations
  - “Group by”: Global and keyed aggregations

hazelcast

Bounding the Data

- Scope - which data to aggregate?
  - ”Since the app started” – potentially infinite with a stream!
  - “Last 30 seconds” defines a window
- Frequency - When is the result provided?
  - With each item (frequent)
  - After all data in the scope were processed
  - In specified intervals - early results

hazelcast

Bounding the Data

Scope \ Frequency	Each input record	End of scope	Intermediate results*
Whole dataset	Rolling aggregation	N/A (Dataset generally infinite)	N/A
Window	Windowed aggregation with early results (*special case)	Windowed aggregation	Windowed aggregation with early results

*Frequency can be defined using time in Jet 3.0.
More general behaviour might be used as well, e.g. data-driven.

hazelcast

Aggregation API in Jet

- Rolling aggregation

```
items.rollingAggregate(aggregateOperation)
```

- Windowed aggregation

```
items.window(def).aggregate(aggregateOperation)
```

  - Must include window definition (more on that after the lab)

- Library with aggregate operations

```
com.hazelcast.jet.aggregate.AggregateOperations
```

  - count, sum, average, min, max, toList …

hazelcast

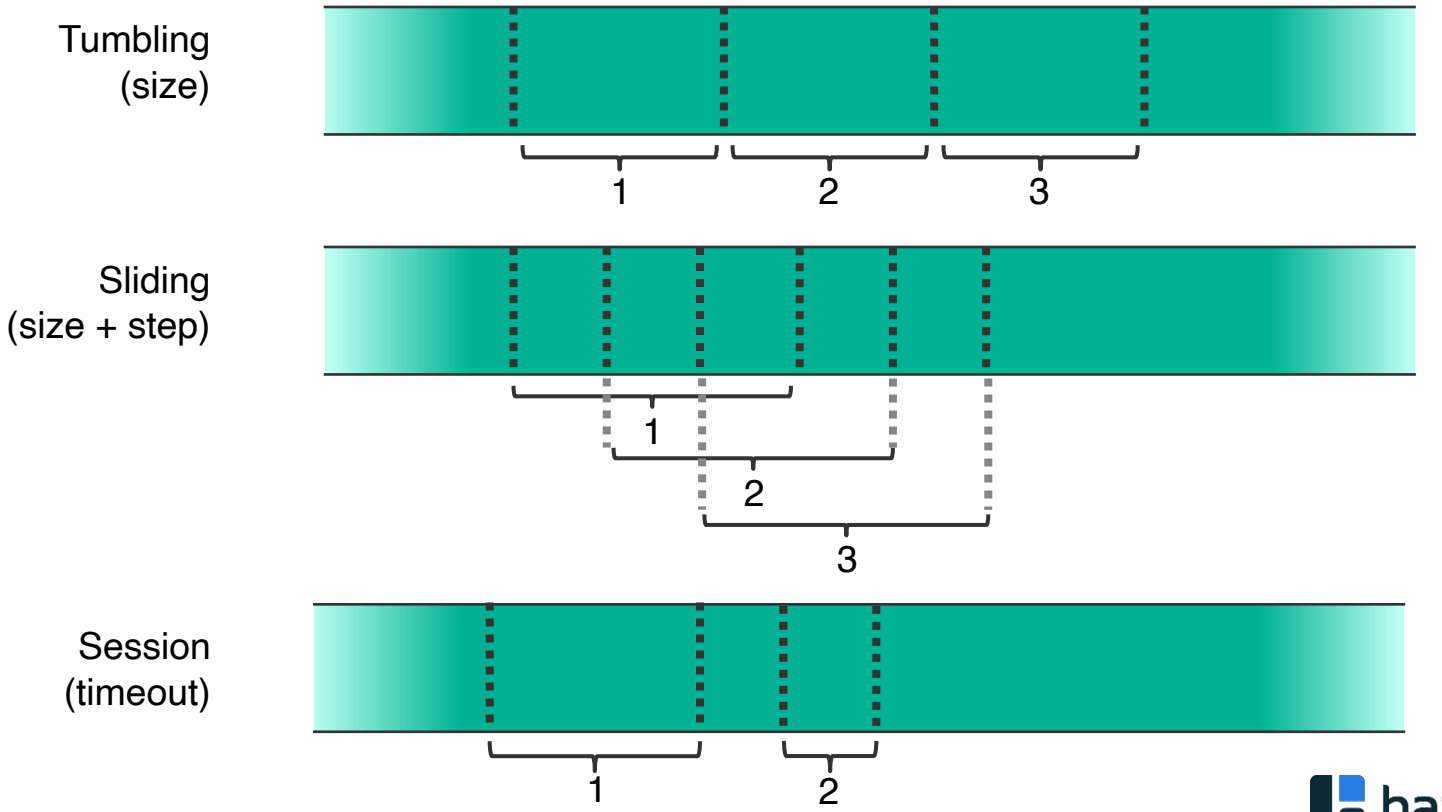
Lab 3: Rolling Aggregation

- Compute maximum trade price observed
  - Scope: whole dataset
  - Output frequency: with each input item
  - -> Rolling aggregation

Make sure you stopped Lab 2!

hazelcast

Types of Windows

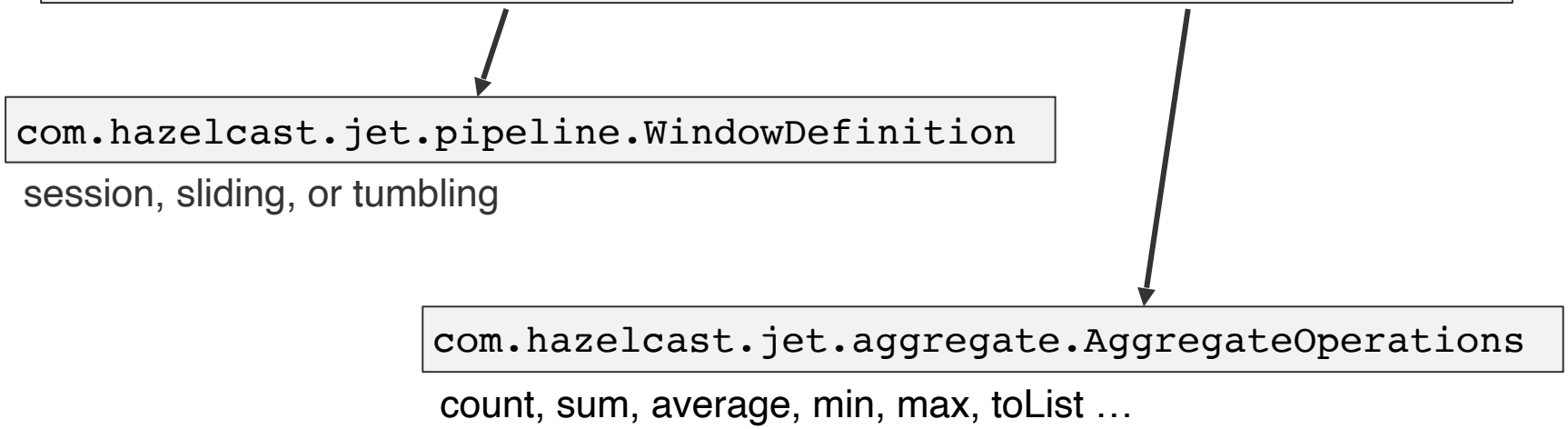


hazelcast

Windowing API in Jet

- Declare the aggregation as windowed

```
items.window(windowDef).aggregate(aggregateOperation)
```



hazelcast

Timestamps

- Timestamp in record metadata (e.g. Kafka)

```
.withNativeTimestamps()
```

- From embedded in record

```
.withTimestamps(r -> r.getTimestamp())
```

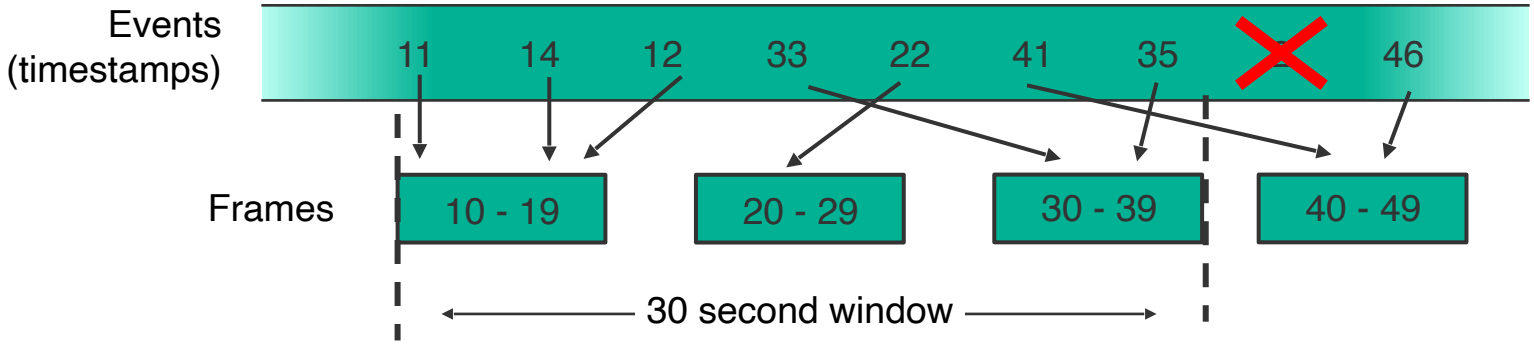
  - Need to specify how to extract the timestamp

- Jet uses local clock to timestamp the record

```
.withIngestionTimestamps()
```

hazelcast

Event Time Processing



- No ordering
- How long will we wait for stragglers?

hazelcast

Lab 4: Windowing

- Compute sum of trades for 3-second intervals
  - Tumbling window
  - Output each 3 seconds, should be roughly constant
- Compute sum of trades for 3-second intervals, result so far each second
  - Tumbling windows with early results
  - Output each second, grows for 3 seconds and then drops
- (Optional) Compute sum of trades in last 3 seconds, update each second
  - Sliding windows
  - Output each second, should be roughly constant

hazelcast

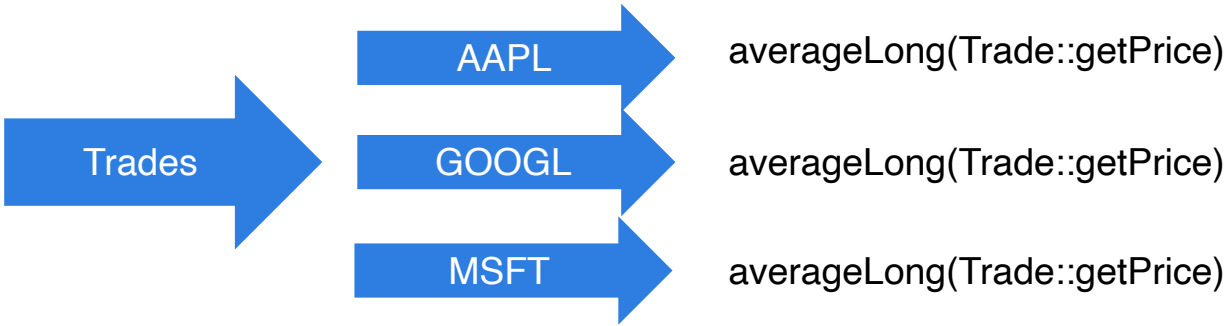
Grouping

- Global aggregation
  - One aggregator sees the whole dataset
  - Complex Event Processing use-cases (if A is observed after B then do C)
- Keyed aggregation
  - GROUP BY from the SQL
  - Splits the stream to sub-streams using the key extracted from each record
  - Sub-streams processed in parallel
- Grouping API in Jet
  - `users.groupingKey(User::getId)`

hazelcast

Grouping

- Global aggregation
  - One aggregator sees the whole dataset
  - Complex Event Processing use-cases
    - if A is observed after B then do C
- Keyed aggregation (SQL GROUP BY)



hazelcast

Grouping API

```
users.groupingKey(User::getID)
```

- Groups event objects by specified key
- Groups processed in parallel

hazelcast

Lab 5: Grouping

- Compute sum of trades for 3-second intervals per ticker
  - Per-ticker results for the previous lab

hazelcast

Aggregations - Takeaways

- Aggregations combine multiple input records to a single value
- Aggregate the whole stream or split it using a key and aggregate the sub-streams (GROUP BY)
- Define the scope of the aggregation: whole dataset or a window
- How often does aggregate operation provide the result?
  - With each record
  - When the scope has been processed
  - Something in between
    - In specified intervals (early results)
    - Data-driven (Not supported by Jet now)

hazelcast

Honorable Mentions

- Cascade aggregate operations
  - Aggregate aggregations (e.g. maximal average value)
  - Flight Telemetry Demo
- Use grouping for join and correlation
  - Windowing to be added for streaming join
  - https://docs.hazelcast.org/docs/jet/latest-dev/manual/#cogroup

- Custom aggregation API

hazelcast

Lunch Break

60 Minute Break

hazelcast

Scaling and Operations

hazelcast

Scaling and Fault Tolerance

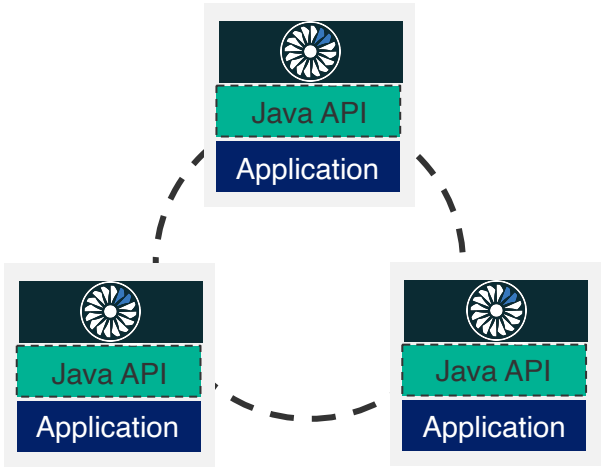
- Jet clusters can grow and shrink elastically
  - Add members to accommodate workload spikes
  - Workload is dynamically distributed across all cluster members
- Resilience through redundancy
  - Fault tolerance (network, server)
  - Cluster redistributes workload to available members

How Jet Fault Tolerance Works

- Regular backups (snapshots)
  - Restart computation from last snapshot if topology changes
  - Single node failures/additions - snapshot restored from replicated memory storage
- Preconditions:
  - Replayable source (e.g Kafka, Hazelcast Event Journal, not JMS)
  - Deterministic (no mutable lookup tables, no randomness)
  - Idempotent sink

Jet Application Deployment Options

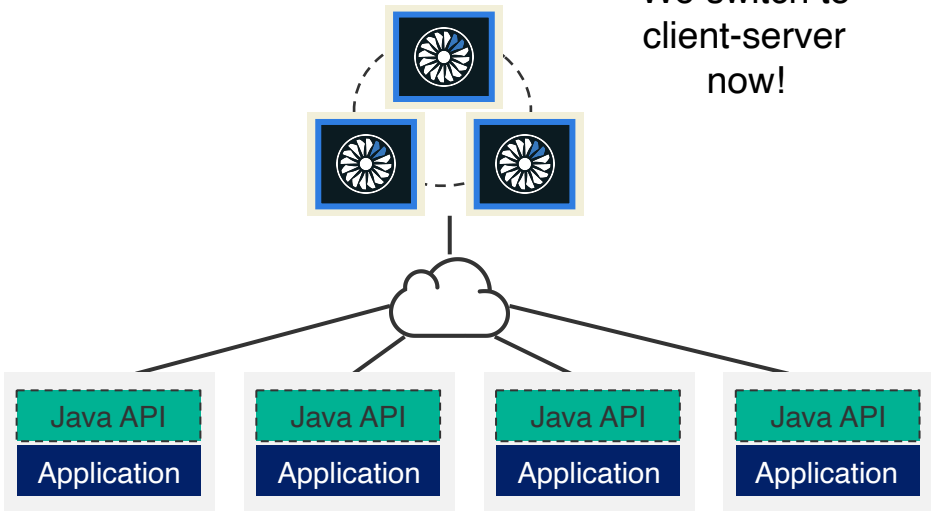
Embedded



- No separate process to manage
- Prototyping, Microservices. OEM.
- Java API for management
- Simplest for Ops – nothing extra

Client/Server

We switch to client-server now!



- Separate Jet cluster
- Isolate Jet from the application lifecycle
- Jet CLI for management
- Managed by Ops

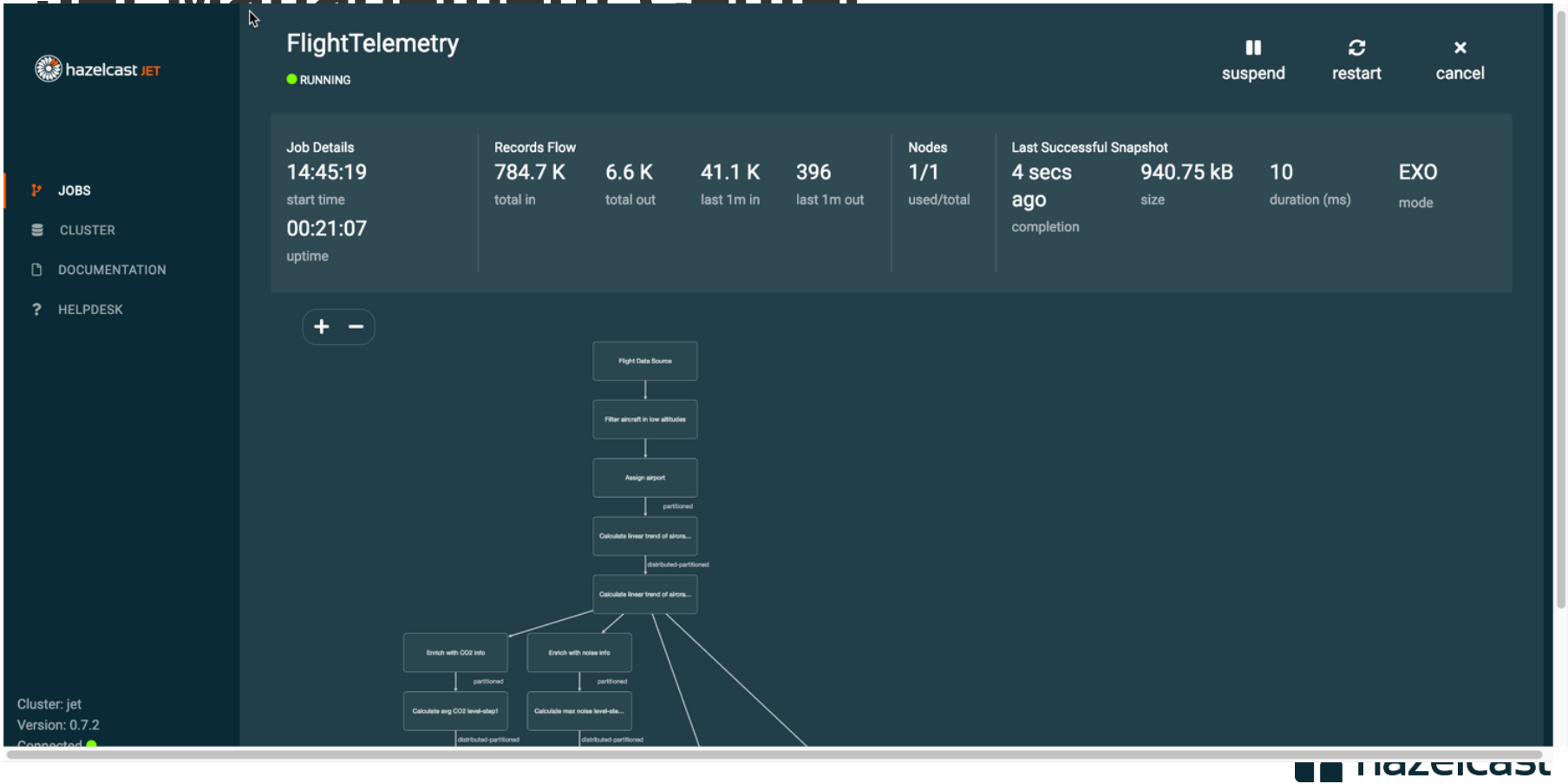
hazelcast

Command Line Tools

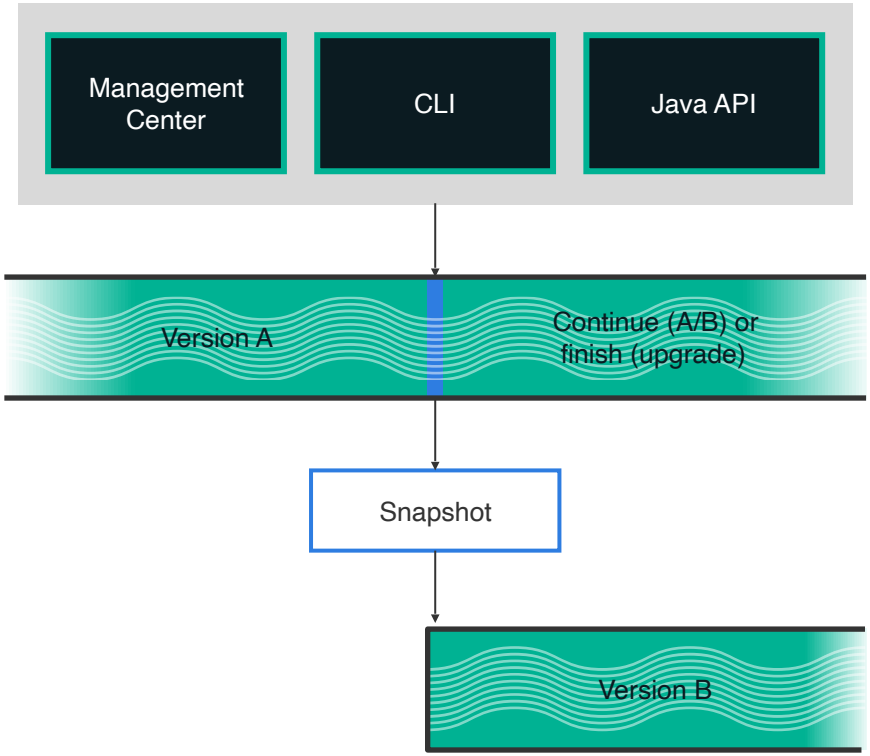
- `/bin/`
  - Command line tooling for Jet
- `jet-start.sh, jet-stop.sh`
  - Control cluster lifecycle
- `jet.sh`
  - Control job lifecycle
  - Command-line alternative to the Java API

hazelcast

Jet Management Center



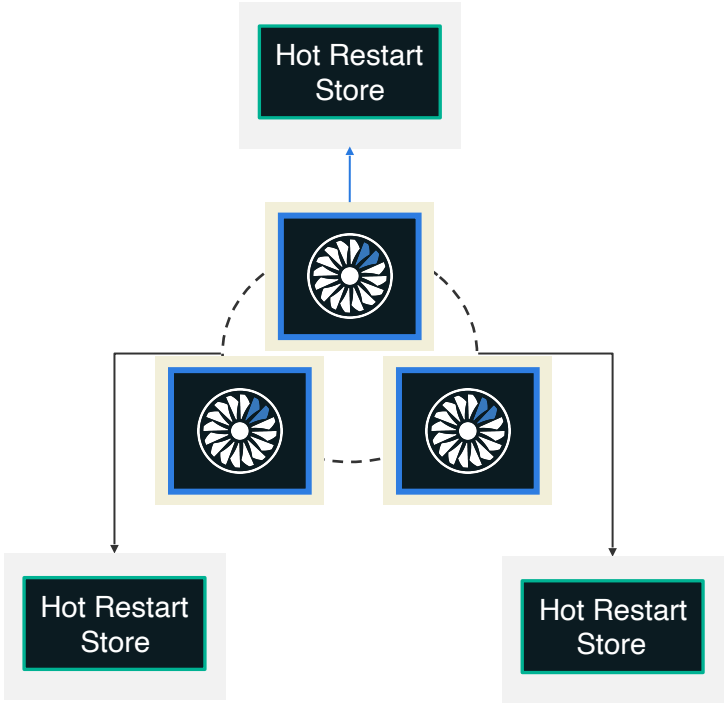
Rolling Job Upgrades



- Allow jobs to be upgraded without data loss or interruption
- Processing Steps
  1. Jet stops the current Job execution
  2. It then takes the state snapshot of the current Job and saves it
  3. The new classes/jars are distributed to the Jet nodes
  4. The job then restarts
  5. Data is read from the saved snapshots
- All of this in a few milliseconds!

hazelcast

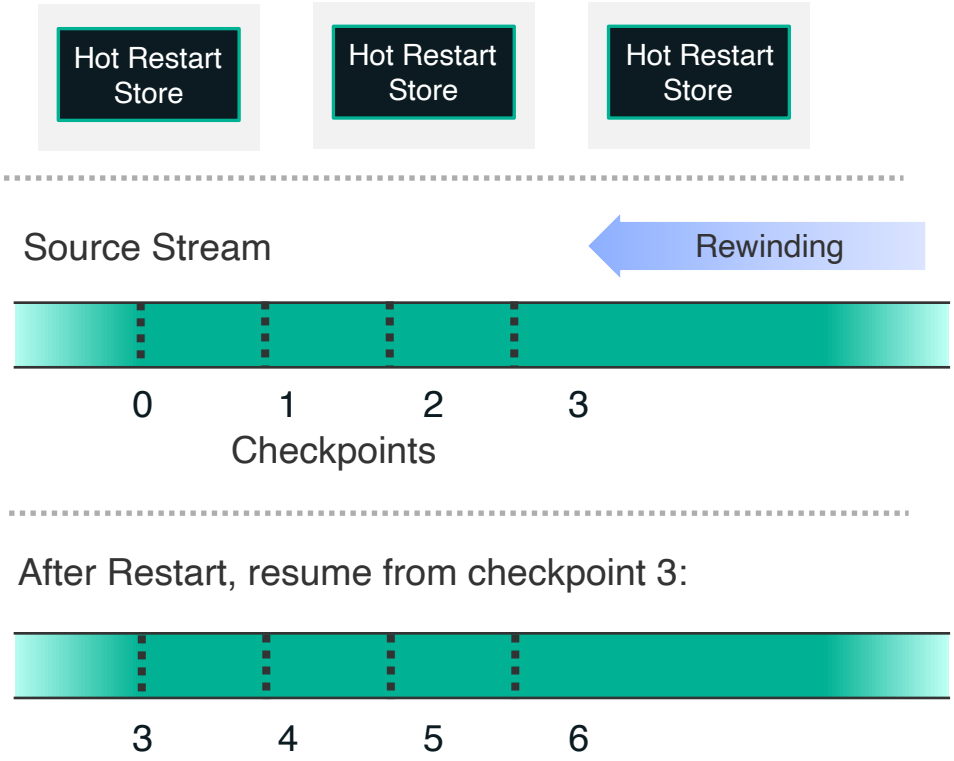
Lossless Recovery: Before Lights Out



- Jobs, Job State, Job Configuration configured to be persistent with Hazelcast's Hot Restart Store capability
- Checkpoints are similarly configured to be Hot Restartable
- Jet is configured to resume on restart

hazelcast

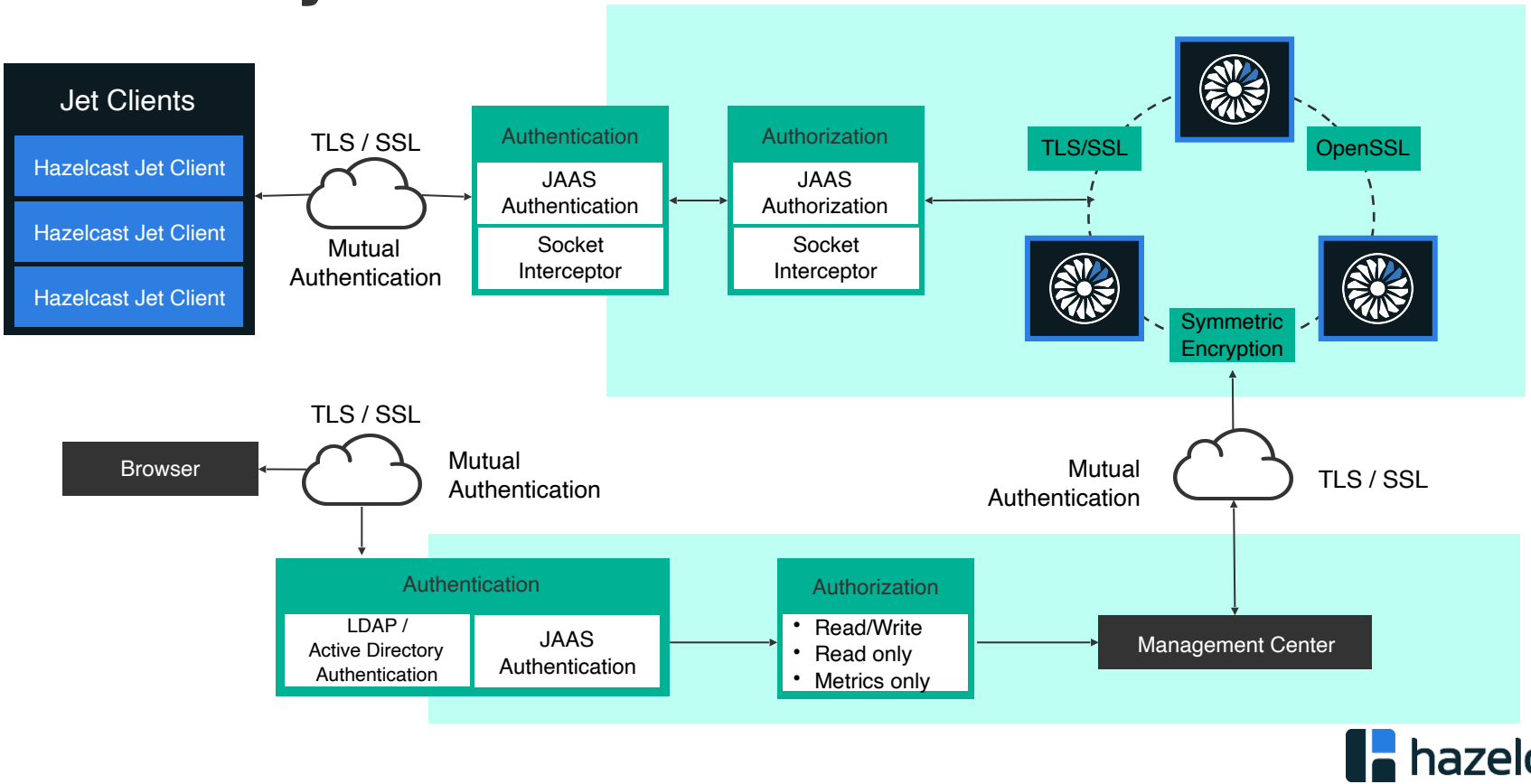
Lossless Recovery: Automatic Job Resumption



- When cluster is restarted, Jet discovers it was shut down with running jobs
- Jet restarts the jobs
- Checkpoints are recovered
- For streaming, rewindable sources are rewound using saved offsets (Kafka, Hazelcast IMap, Hazelcast ICache events).
- If the source cannot be fully rewound, the job is terminated with error, or continued, depending on configuration
- Batch sources are resumed from last pointer, otherwise from the beginning

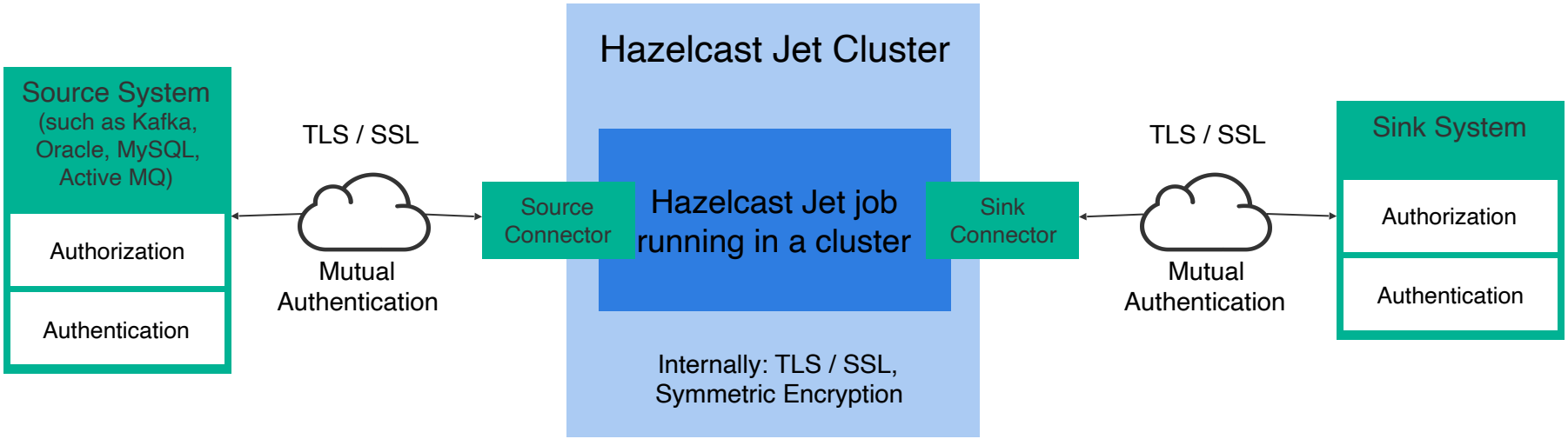
hazelcast

Security Suite: Job Submission



hazelcast

Security Suite: Data Pipeline



hazelcast

Scaling and Operations - Takeaways

- Jet can run embedded (in-process) or in a Client-Server mode
- Elastic clustering for scaling and increasing resilience
  - Preconditions: replayable source, deterministic computations without side effects, idempotent sink
- Tools for monitoring and managing the cluster and the jobs
  - Command line tools in `/bin`
  - Management Center

hazelcast

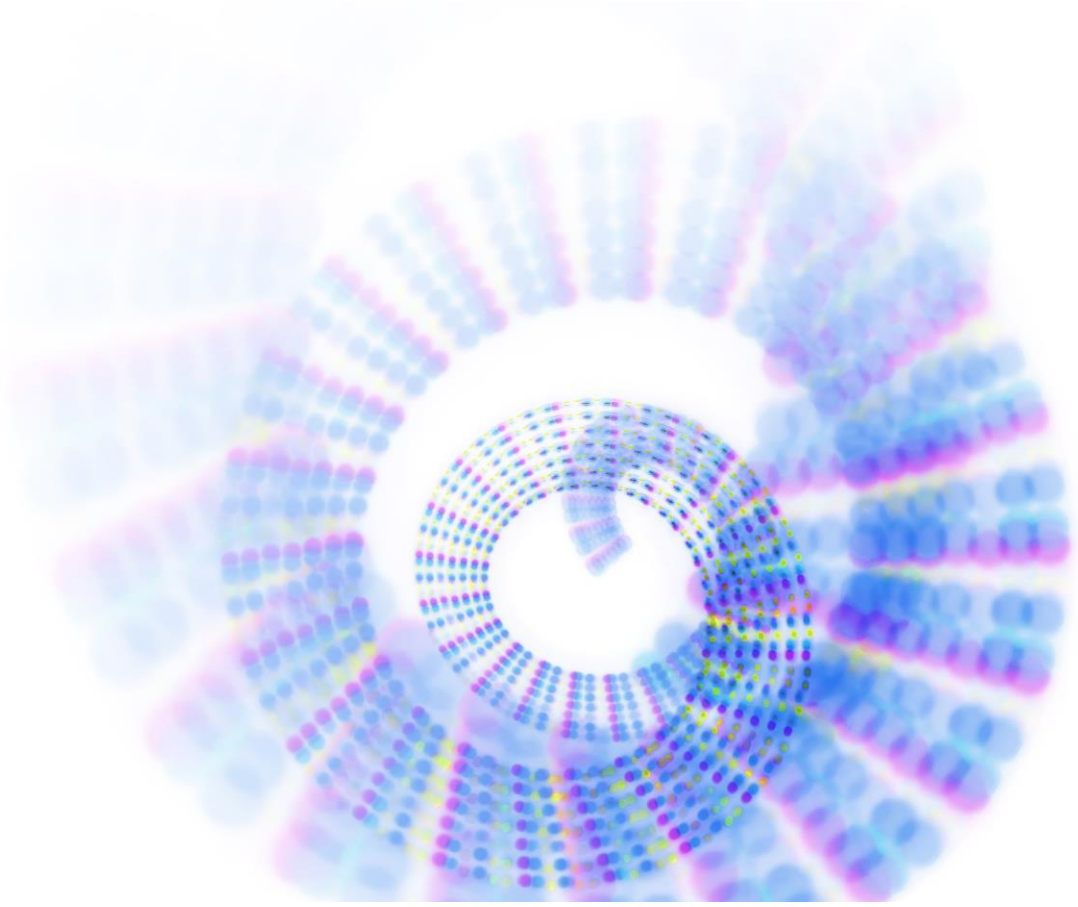
Honorable Mentions

- Other deployment options
  - Docker
  - Kubernetes
  - Hazelcast Cloud
- Job Upgrades
- Lossless Recovery
- Security

hazelcast

Q and A

Next Up … IMDG

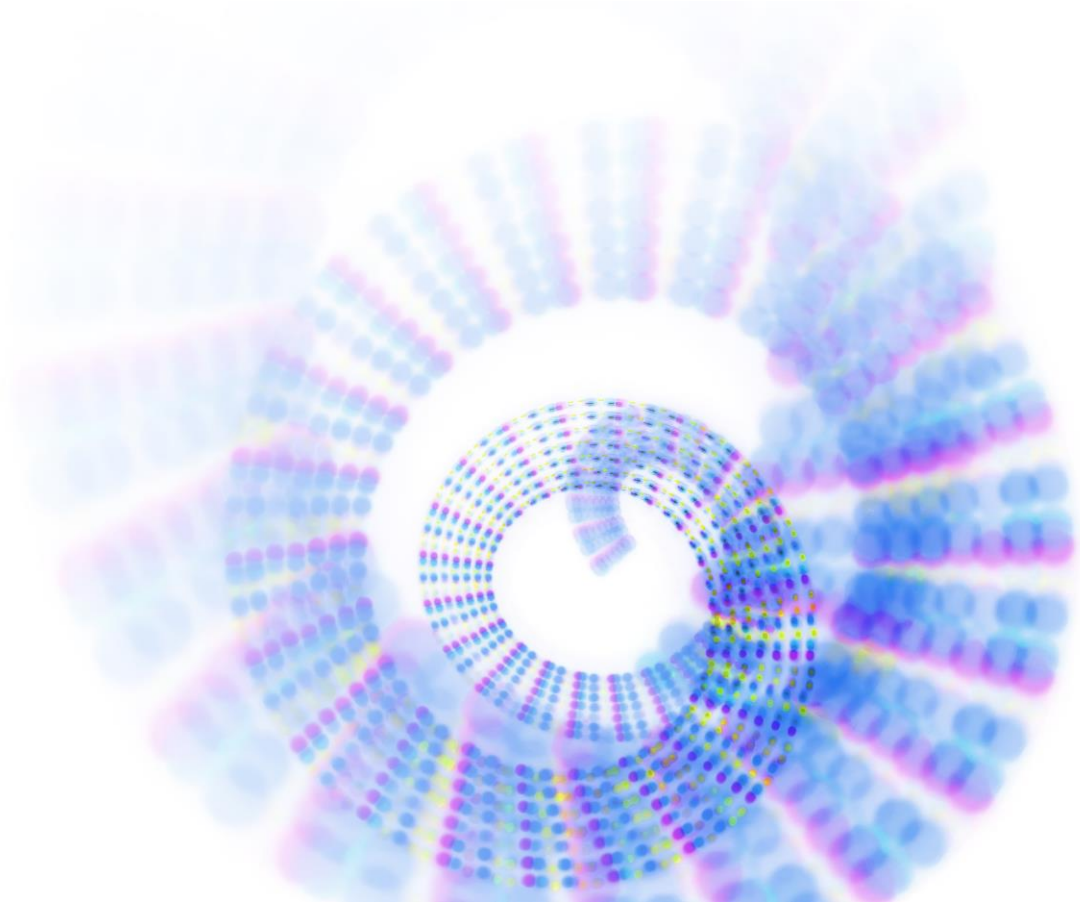


hazelcast

Hazelcast IMDG

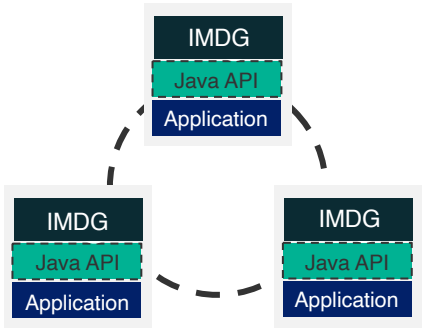
Version 3.12 May, 2019

Topologies



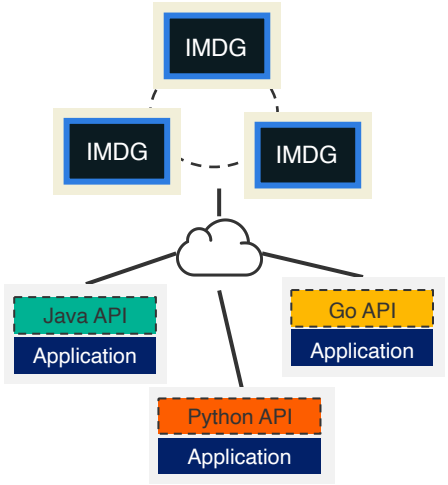
Review: Two Main Topologies

Embedded



- Rapid Prototyping
- Microservices Architectures
- OEM integration

Client/Server



- Scale-Up and Scale-Out Deployments
- Decouple scaling of Storage and Application
- Ease GarbageCollection Optimization

hazelcast

Embedded Pros and Cons

- Easy to get started
- Easy deployment
- In a cluster of size x, 1/x of the data is local to the reader

- Service and storage is on JVM
- Harder to optimize garbage collection
- Harder to scale
- Hard to minimize data movement
- Can’t just “restart the app”

hazelcast

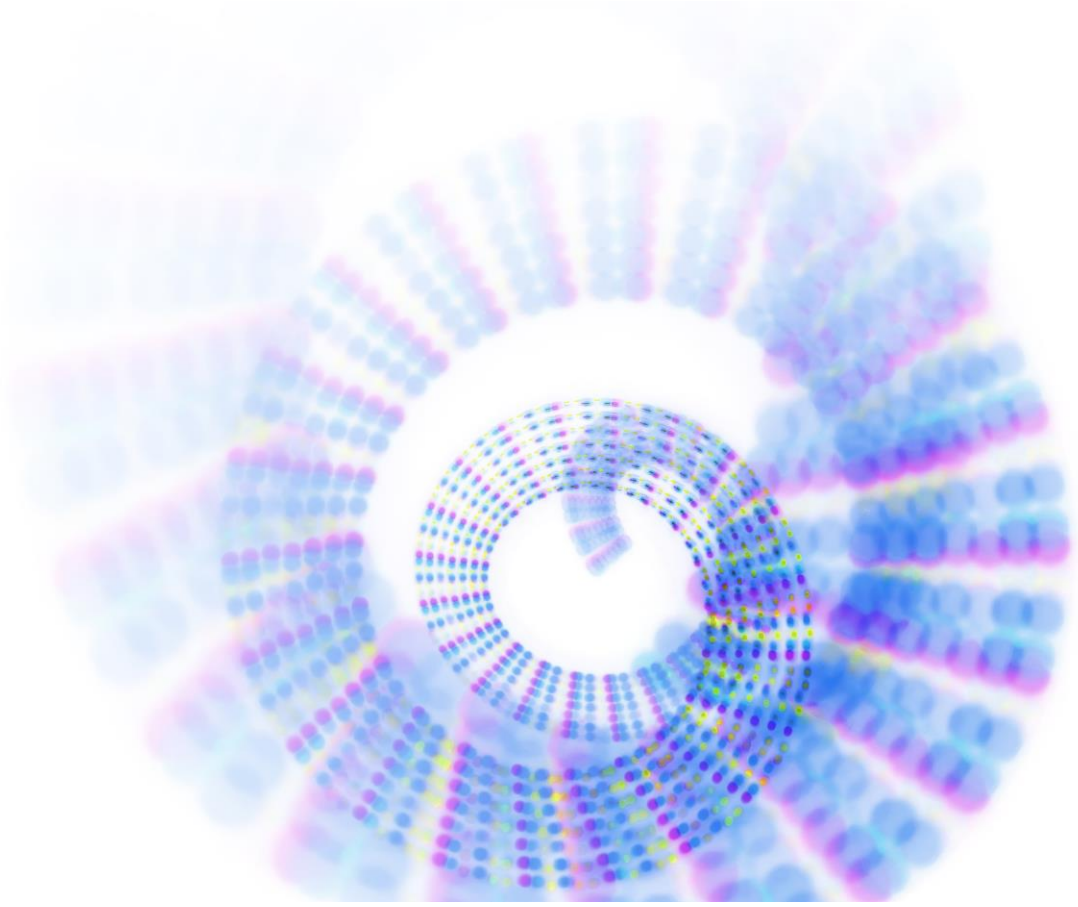
Client/Server Pros and Cons

- Most commonly used
- Easily scalable
- Additional security options (Enterprise)
- Segregates storage and application (GC optimization)
- Operational separation between client application and server cluster

- Separate setup of standalone cluster
- No data locality

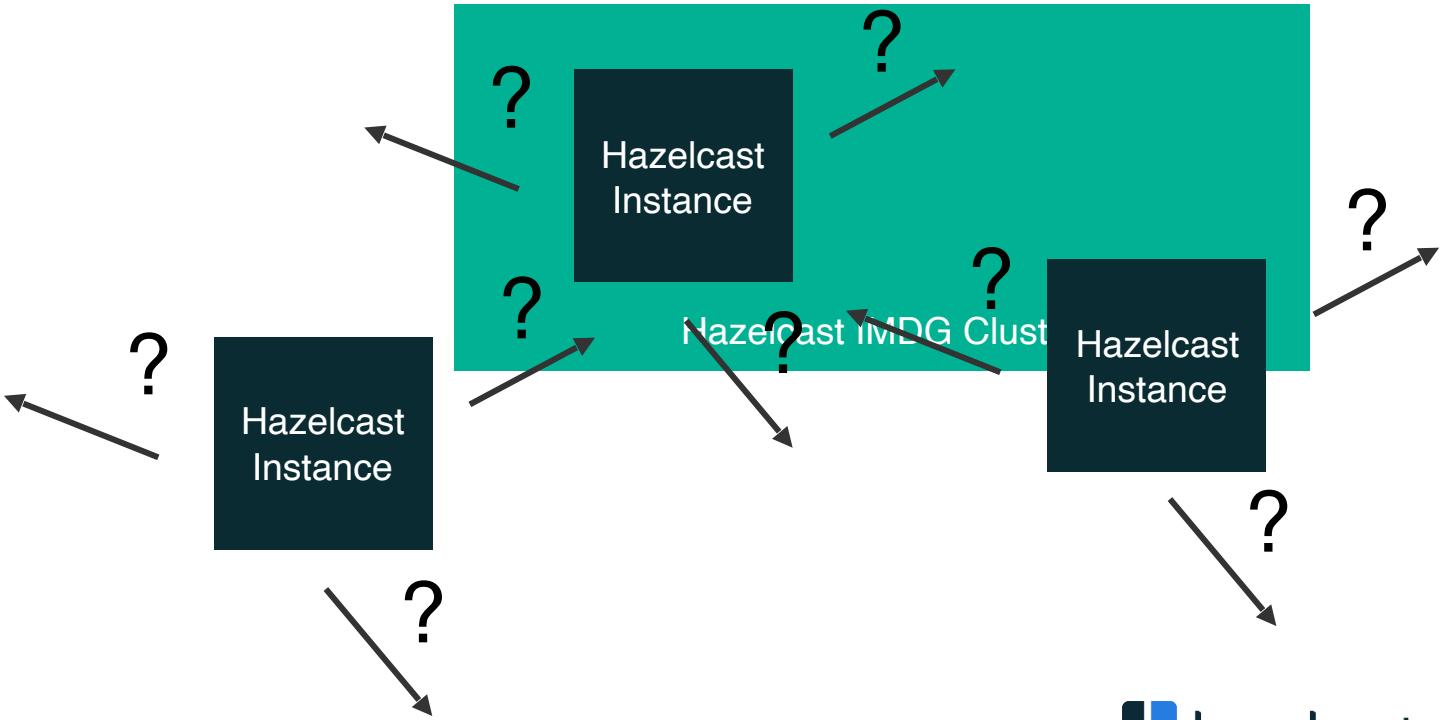
hazelcast

Discovery



What is Discovery?

- Form
- Find
- Join

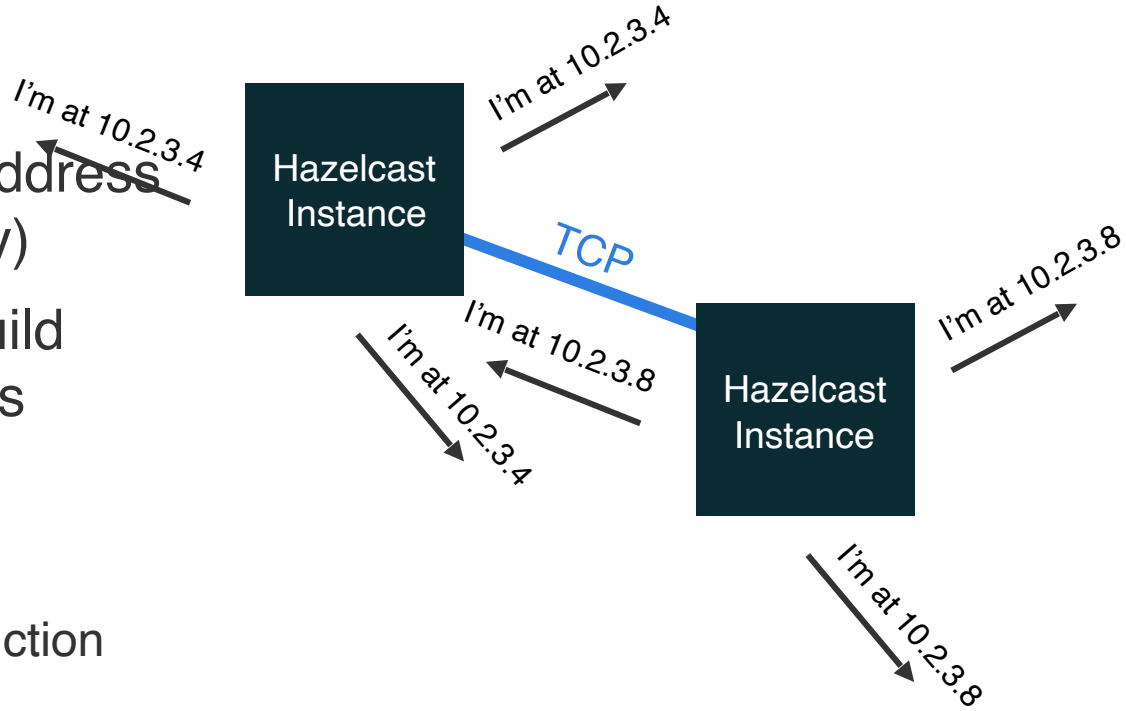


Hazelcast Discovery Options

- TCP/IP - multicast and unicast
- Discovery plug-ins for cloud
- Discovery Service Provider Interface (SPI)

TCP/IP Multicast

- Send and listen on configured multicast address and port (224.x.x.x:yyy)
- Members learn and build direct TCP connections
- Useful for test/POC environments
  - Not suitable for production



TCP/IP Unicast Discovery

- New members connect to specified list of IP addresses
  - First two form the cluster
  - Later members learn complete list when connecting
- Best practice #1: Specify at least two peer addresses in config
- Best practice #2: Specify local IP interface used for cluster access

hazelcast

TCP/IP Unicast Discovery Configuration

```
<hazelcast>
  ...
 <network>
   ...
   <join>
     <multicast enabled="false">
     </multicast>
     <tcp-ip enabled="true">
       <member>machine1</member>
       <member>machine2:5799</member>
       <member>192.168.1.0-7</member>
       <member>192.168.1.21</member>
       <interface>192.168.1.10</interface>
     </tcp-ip>
     ...
   </join>
   ...
 </network>
 ...
</hazelcast>
```

- Specify cluster members
  - Host name or address
  - Range of addresses
  - Port number optional
- Specify local interface address

hazelcast

Discovery Plug-Ins

Hazelcast-Supported	Community-Supported
Apache jclouds AWS Azure Eureka Google Cloud Platform (GCP) Kubernetes OpenShift Pivotal Cloud Foundry (PCF) Zookeeper	Consul .etcd Heroku

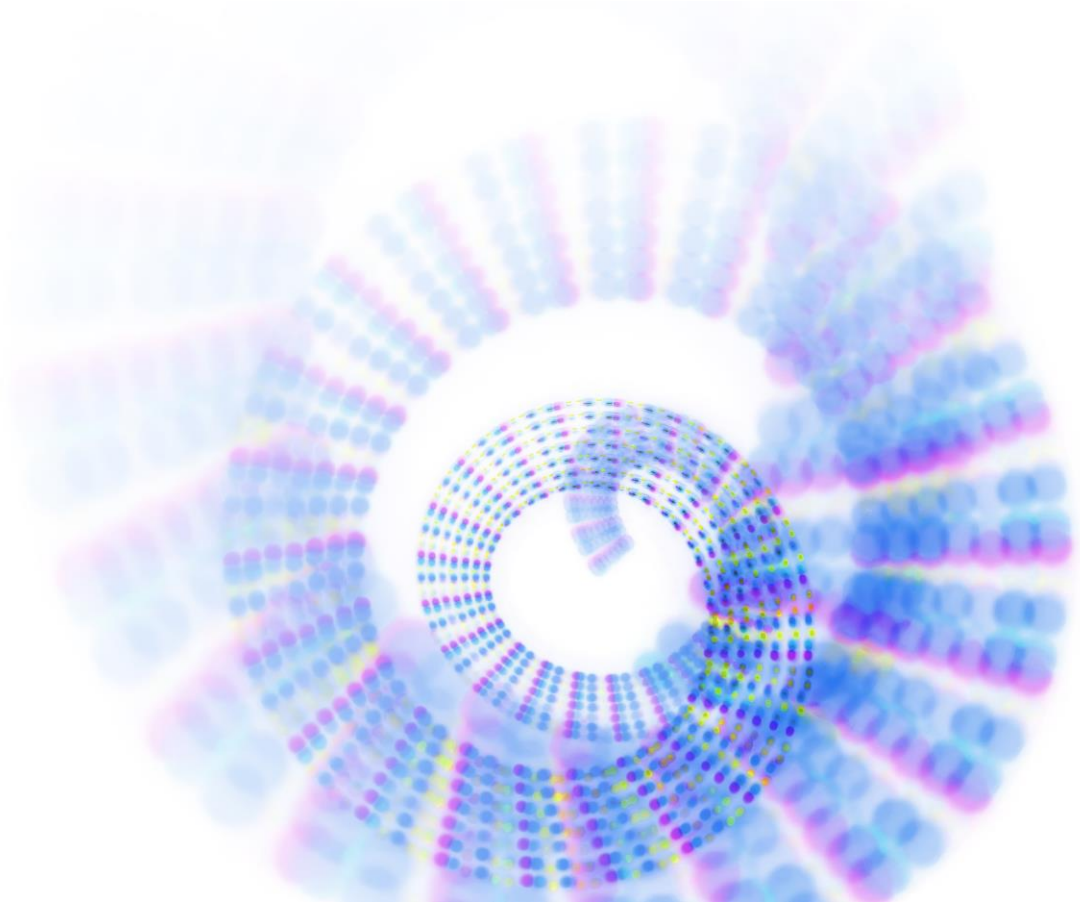
hazelcast

Discovery Service Provider Interface (SPI)

- Develop your own discovery plug-in
  - Fully documented in IMDG Reference Manual
  - Example using lookup in /etc/hosts on Github under Hazelcast Code Samples

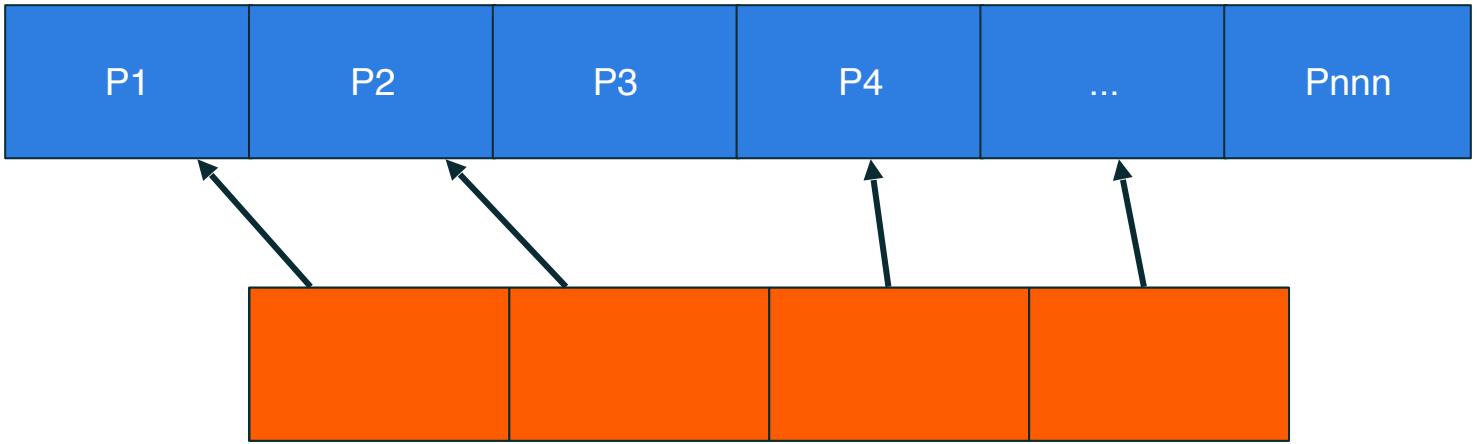
hazelcast

Partitioning



What is a Partition?

A “chunk” of memory that holds data




```
Partition number = (hash of key/name) MOD (partition count)
```

hazelcast

Partitioning Across a Cluster

I'm the oldest member - I'll keep you all updated.

Primary

Backup

Member1

P1
P2
P3
P4
. . .
P271

Partition	Primary	Backup
P1	1	2
P2	2	1
P*n*	1	2

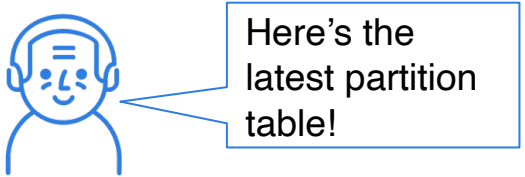
Member2

P1
P2
P3
P4
. . .
P271

Partition	Primary	Backup
P1	1	2
P2	2	1
P*n*	1	2

hazelcast

Adding a Member (Data Rebalancing)



Primary Backup

Member1

P1
P2
P4

Partition	Primary	Backup
P1	1	3
P2	2	1
P*n*	2	3

Member2

P2
P3
P271

Partition	Primary	Backup
P1	1	3
P2	2	1
P*n*	2	3

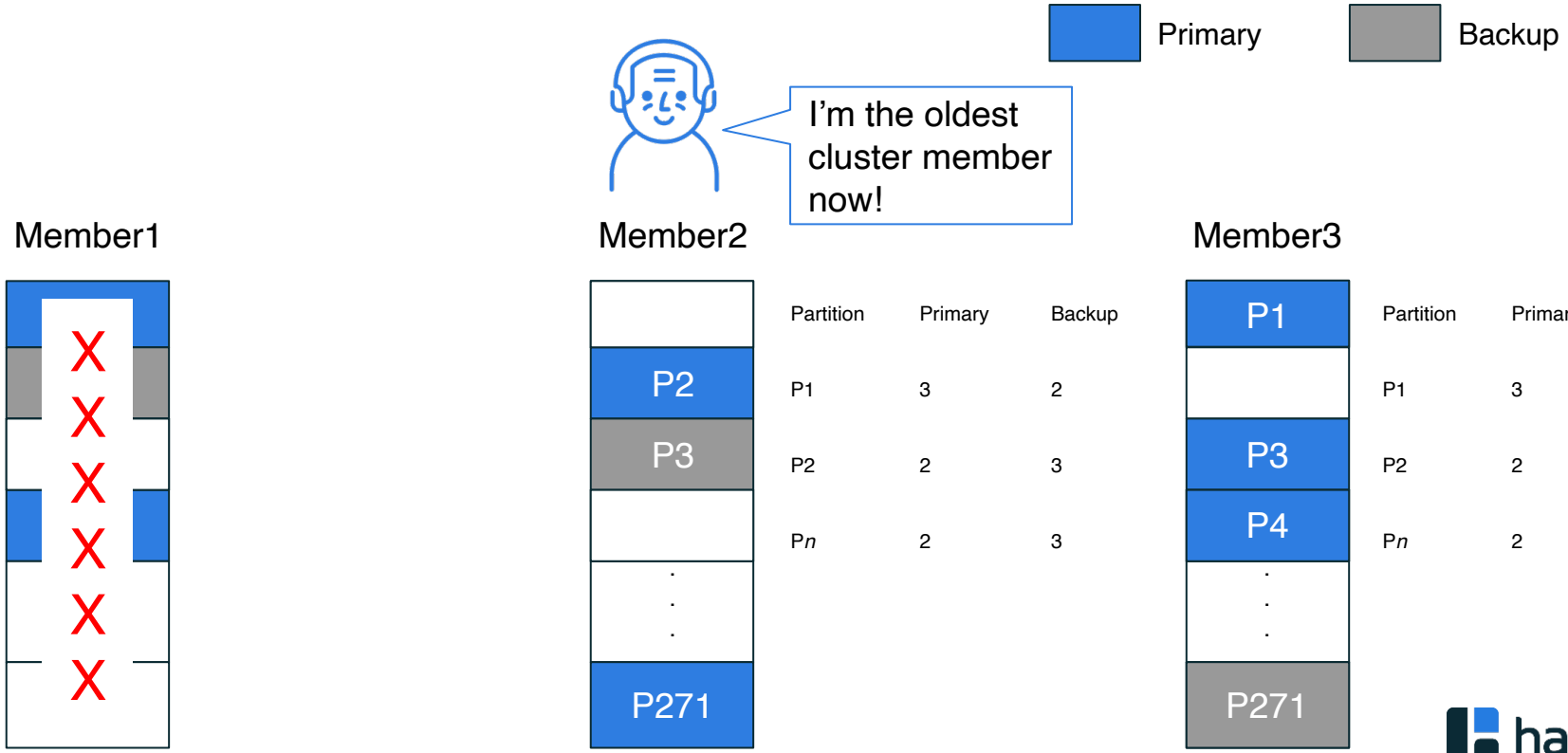
Member3

P1
P3
P4
P271

Partition	Primary	Backup
P1	1	3
P2	2	1
P*n*	2	3

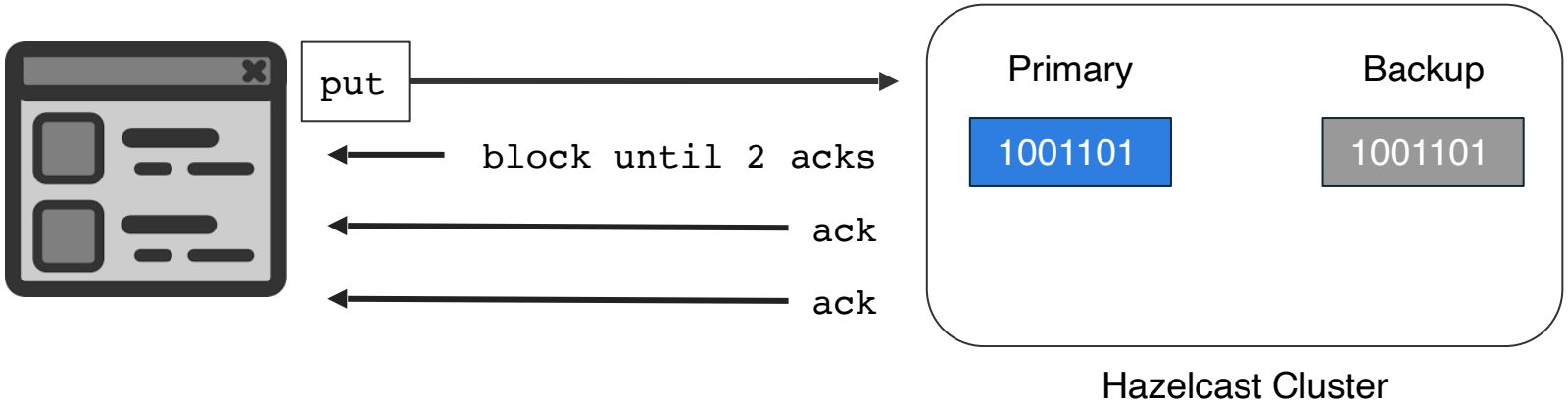
hazelcast

Removing a Member



hazelcast

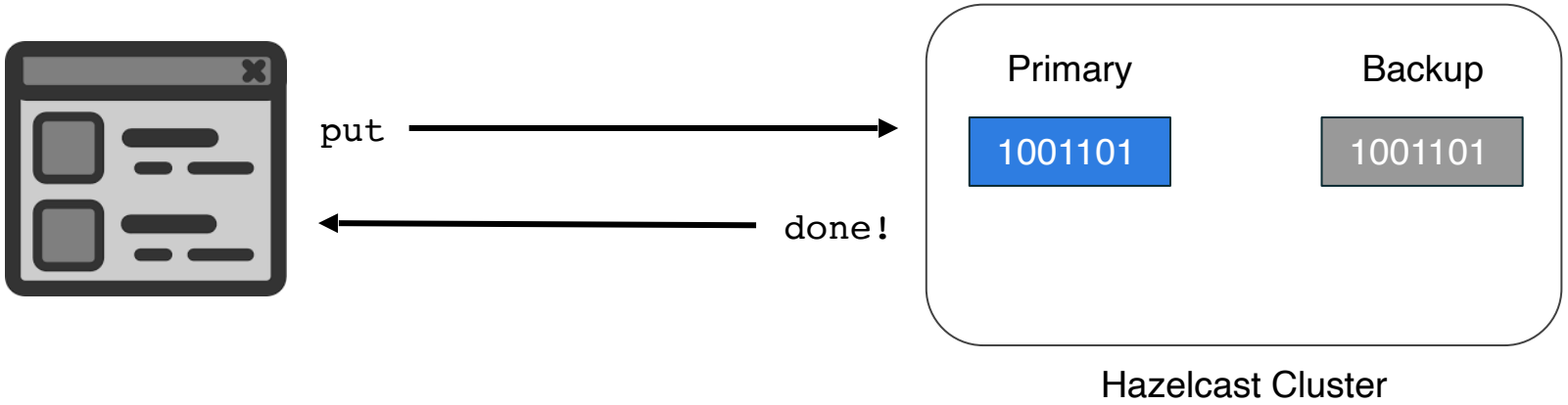
Synchronous Backups



- Assures backup is complete and available if needed
- Blocking operation - can lead to latency issues

hazelcast

Asynchronous Backups



- “Fire and forget”
  - Relies on Hazelcast internal synchronization to complete backup
- No blocking, but less assurance of backup availability

hazelcast

Configuration

```
<hazelcast>
  <map name="default">
    <backup-count>1</backup-count>
    <async-backup-count>1</async-backup-count>
  </map>
</hazelcast>
```

- Can have both sync and async for same map
  - 6 total backups shared by sync and async
  - Make sure you have enough distributed memory!
- Default setting: backup count of 1, no async backup

hazelcast

Data Safety

- How many backups do you really need?
  - One backup is most common
  - Consider memory consumption
  - Consider overall resource availability if a node fails
    - If multiple nodes are failing regularly, backups won't solve the problem

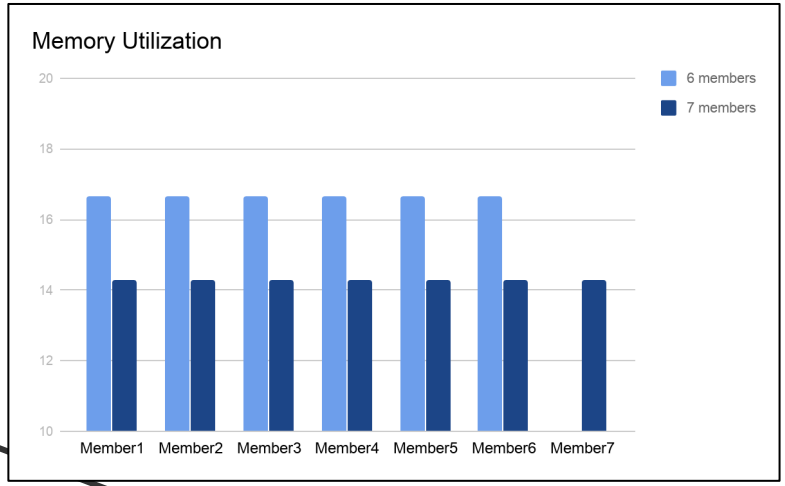
hazelcast

You Can’t Partition Everything!

Partitioned	Non-Partitioned
Map	Queue
MultiMap	Set
Cache (Hazelcast JCache implementation)	List
PN Counter	Ringbuffer
Event Journal	Lock
	ISemaphore
	IAtomicLong
	IAtomicReference
	FlakeIdGenerator
	ICountdownLatch
	Cardinality Estimator

hazelcast

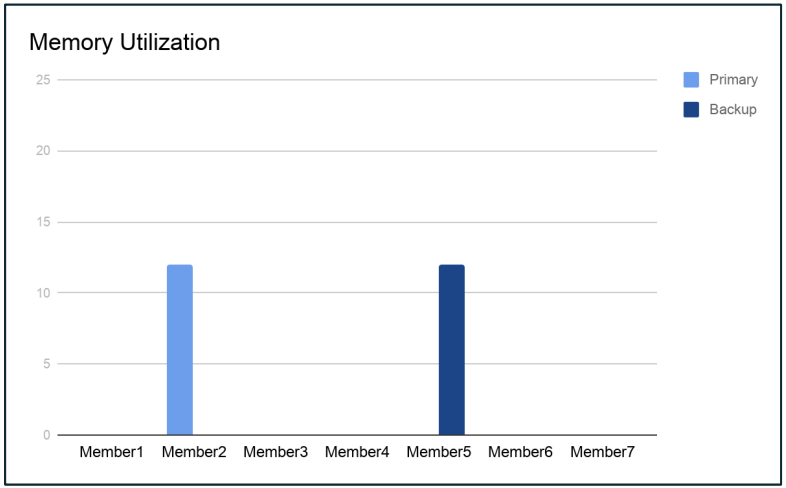
Memory Utilization - Partitioned



Member1 Member2 Member3 Member4 Member5 Member6 Member7

hazelcast

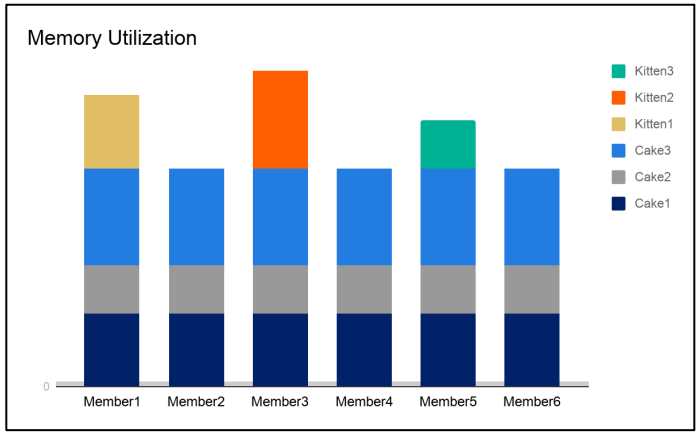
Memory Utilization - Non-Partitioned

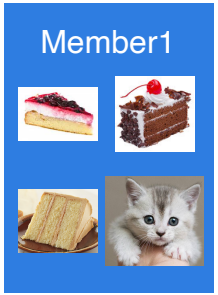


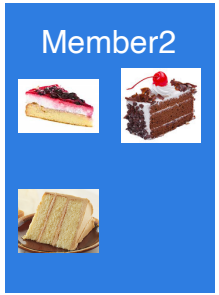
Member1 | Member2 | Member3 | Member4 | Member5 | Member6 | Member7

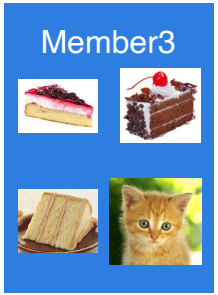
hazelcast

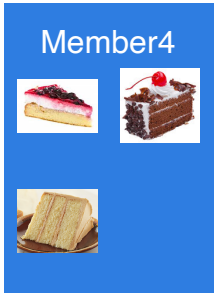
Memory Balancing

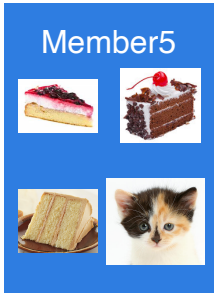


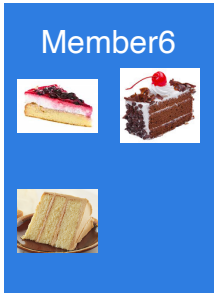






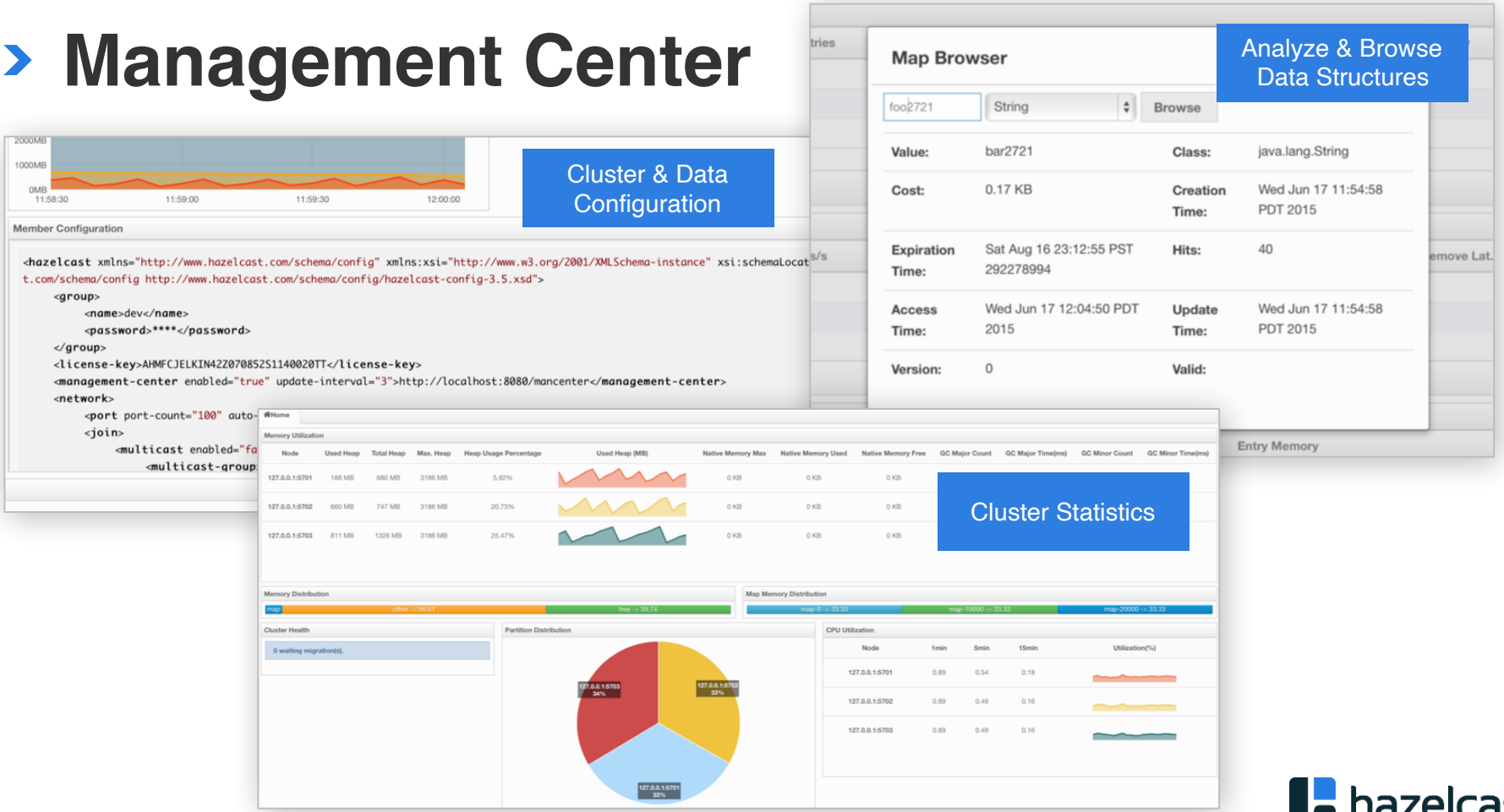




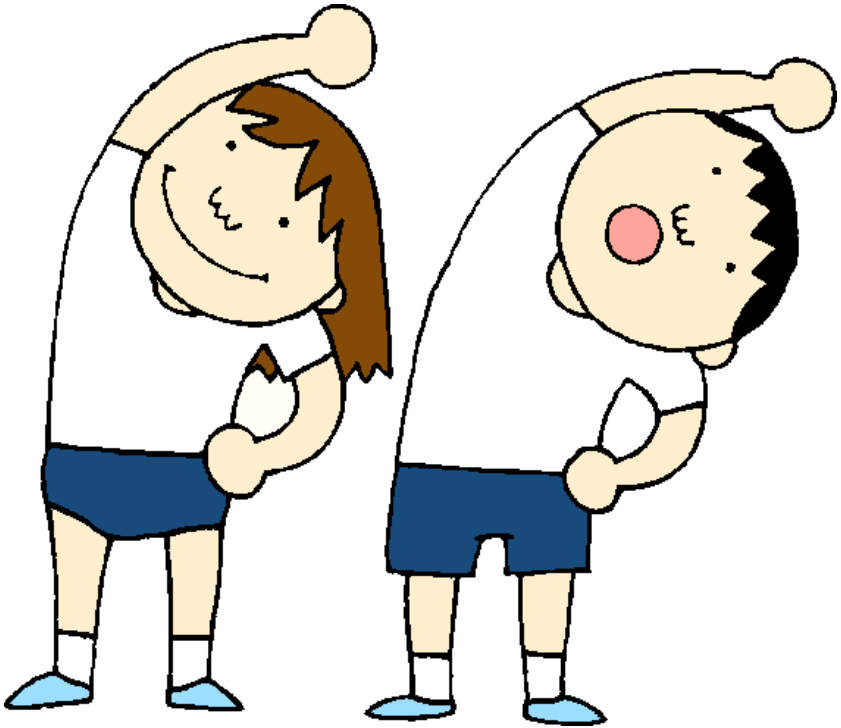


hazelcast

Management Center



Stretch/Bio Break



15 Minute Break

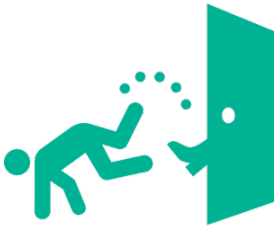
hazelcast

IMap: Expiration and Eviction

Expiration and Eviction

Purpose: to use memory efficiently and to prevent memory overruns

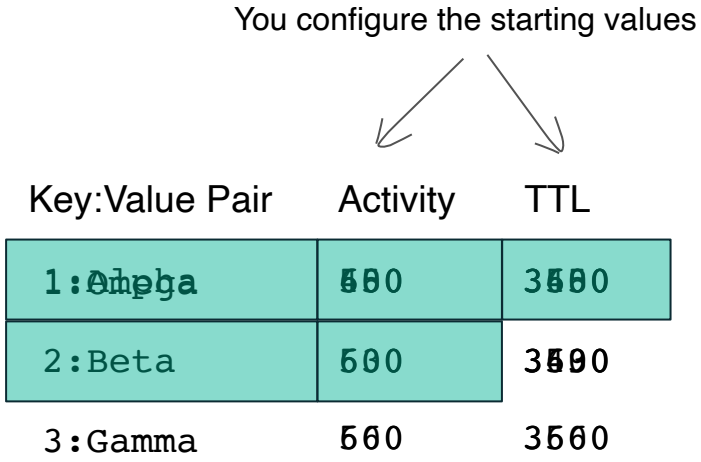
Expiration removes unused entries after a configured amount of time



Eviction removes entries when memory use reaches a configured limit

hazelcast

How Expiration Works



- Two types of timer
  - Activity resets when entry is read
  - Both reset when entry is updated
- Entry is removed from map if timer reaches 0
- You can configure just one or both types of timer

hazelcast

Configuring Expiration Policies

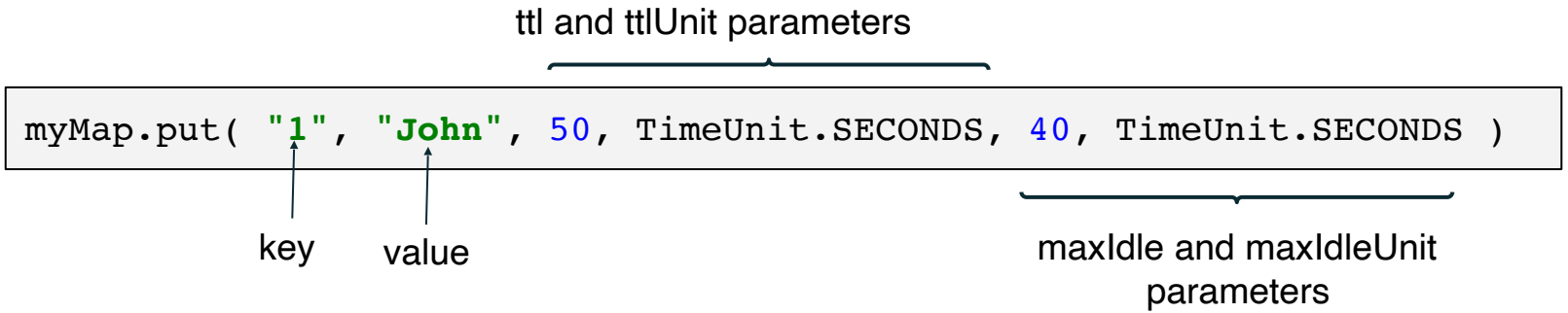
- Configured on per-map basis
- Default is set to 0 (no expiration)

```
<map name="myMap">
    <time-to-live-seconds>300</time-to-live-seconds>
     <max-idle-seconds>1800</max-idle-seconds>
</map>
```

hazelcast

Setting Timers for Specific Entries

- Set timers when adding data to table



- Modify TTL for existing entry

```
myMap.setTTL( "1", 50, TimeUnit.SECONDS )
```

hazelcast

Eviction

Max-size policy - many options!

```
IF (memory use = configured threshold) {
    remove entries based on configured policy;
    }
```

LFU: Least frequently used

LRU: Least recently used

Custom

hazelcast

Max Size Policy Options

- Total number of entries - limit individual map size

`PER_NODE`	Maximum number of map entries per cluster member
`PER_PARTITION`	Maximum number of map entries per partition

- Memory in use (cannot use in-memory format of `OBJECT`)

`USED_HEAP_SIZE`	Maximum amount of used memory in megabytes for each JVM
`USED_HEAP_PERCENTAGE`	Maximum percentage of used memory for each JVM

- Memory available

`FREE_HEAP_SIZE`	Minimum amount of free memory in megabytes for each JVM
`FREE_HEAP_PERCENTAGE`	Minimum percentage of free memory for each JVM

hazelcast

Enterprise Max Size Policy Options

- Requires in-memory format to be set to `NATIVE`

`USED_NATIVE_MEMORY_SIZE`	Maximum used native memory size in megabytes for each Hazelcast instance
`USED_NATIVE_MEMORY_PERCENTAGE`	Maximum used native memory percentage for each Hazelcast instance
`FREE_NATIVE_MEMORY_SIZE`	Minimum free native memory size in megabytes for each Hazelcast instance
`FREE_NATIVE_MEMORY_PERCENTAGE`	Minimum free native memory percentage for each Hazelcast instance

hazelcast

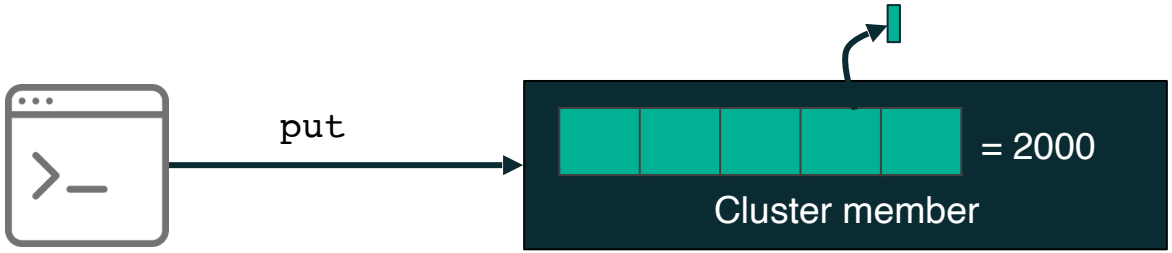
Limiting Map Size Using PER_NODE

- How big do you want your map to get?

```
max_size = total map size / members
```


```
max_size = 10000 / 5
max_size = 2000
```

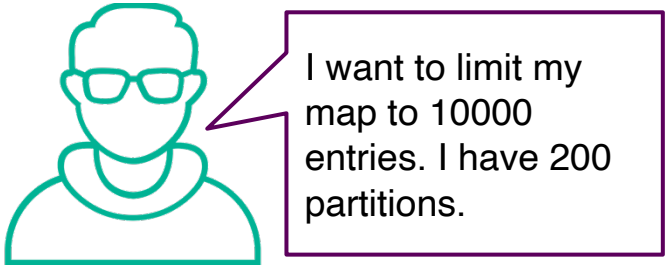
```
<max-size policy="PER NODE">2000</max-size>
<eviction-policy>LFU</eviction-policy>
```



hazelcast

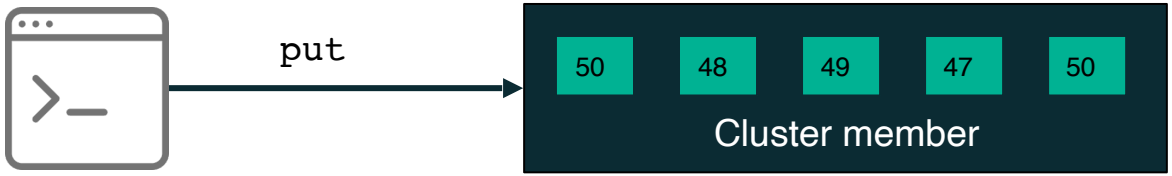
Limiting Map Size Using PER_PARTITION

```
max_size = total map size / number of partitions
```




```
max_size = 10000 / 200

max_size = 50
```

- Not a good option for smaller clusters



hazelcast

Controlling Map Size by Memory Use


```
<max-size policy="USED_HEAP_SIZE">100</max-size>
<eviction-policy>LFU</eviction-policy>
```

- Used memory is per map, not per cluster member
  - Consider number of cluster members, total map size, and other maps/ memory use

```
                    Total map size: 500 MB

Cluster members: 5      Cluster members: 2      Cluster members: 1
USED_HEAP_SIZE: 100 MB  USED_HEAP_SIZE: 250 MB      USED_HEAP_SIZE:
500 MB
```

hazelcast

Controlling Memory Utilization

I want to evict entries if unused memory is less than 5% of total memory.

```
<max-size policy="FREE_HEAP_PERCENTAGE">5</max-size>
<eviction-policy>LFU</eviction-policy>
```

- Free memory options are good for wildcard or default policy
  - Individual map size is not a consideration
  - If “something else” consumes memory, maps get evicted

hazelcast

Manual Eviction

```
map.evictAll()
```

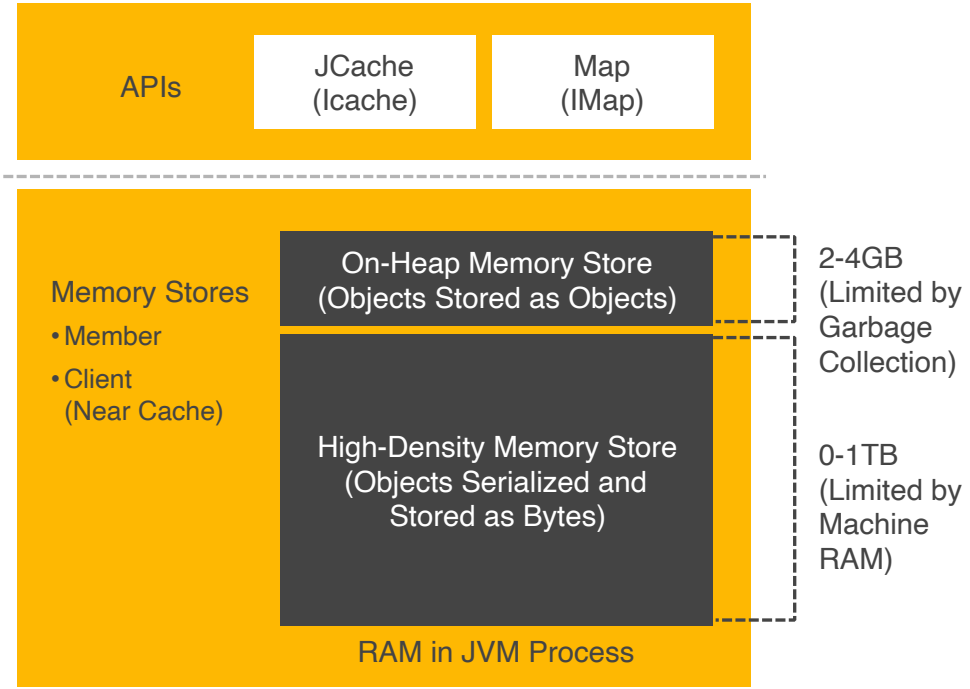
- Removes all unlocked entries from map
  - Any eviction method will not remove locked entries

hazelcast

High-Density Memory Store

High-Density Memory Store (HDMS)

- Enterprise feature
- Alternative in-memory storage implementation
- Data stored outside of garbage-collected heap
  - Managed by IMDG itself
- Maximizes usable memory per machine
- Scales into hundreds of gigabytes.



Optimizing Garbage Collection

- Minimizes heap size handled by garbage collector
- Delivers more predictable latency

	On-Heap	HDMS
Native Storage	0 MB	3.3 GB
Heap Storage	3.9 GB	0.6 GB
Major GC	9 (4900 ms)	0 (0 ms)
Minor GC	31 (4200 ms)	356 (349 ms)

hazelcast

Configuring HDMS

1. Set up HDMS and define size
   - Note: memory assigned to HDMS is not part of heap or available to rest of JVM

```
<hazelcast>
  <native-memory enabled="true">
    <size value="1" unit="GIGABYTES"/>
  </native-memory>
</hazelcast>
```

2. Define data structure(s) that will use HDMS

```
<hazelcast>
 <map name="training">
    <in-memory-format>NATIVE</in-memory-format>
  </map>
</hazelcast>
```

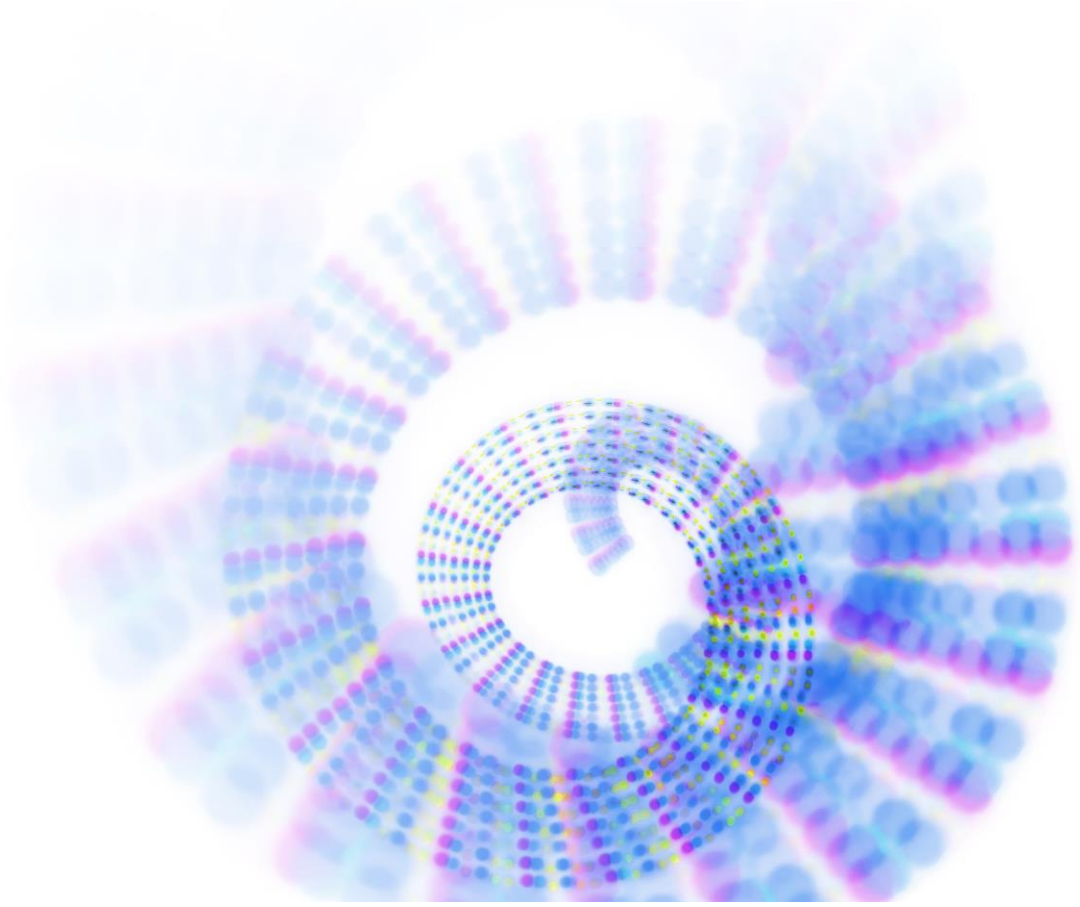
hazelcast

Sidebar: In Memory Format Setting

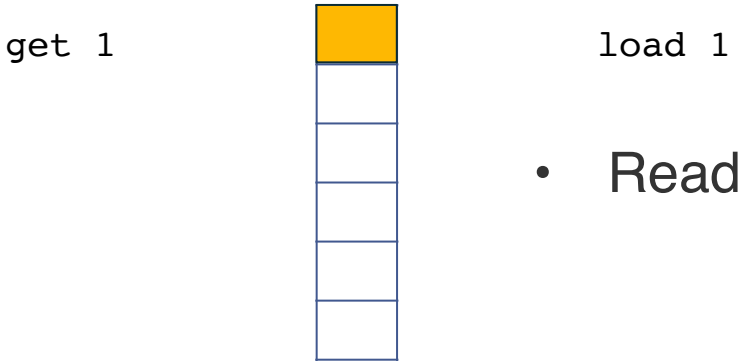
- Default is `BINARY`
  - Both key and value stored in serialized binary format
  - Best if most map operations are gets and puts
- `NATIVE` is same as binary but uses HDMS
- `OBJECT` stores value in non-serialized format
  - Key is still binary
  - Best if map operations are mostly queries and and entry processing
    - Removes deserialization cost

hazelcast

IMap: Persistent Data (External Connectivity)



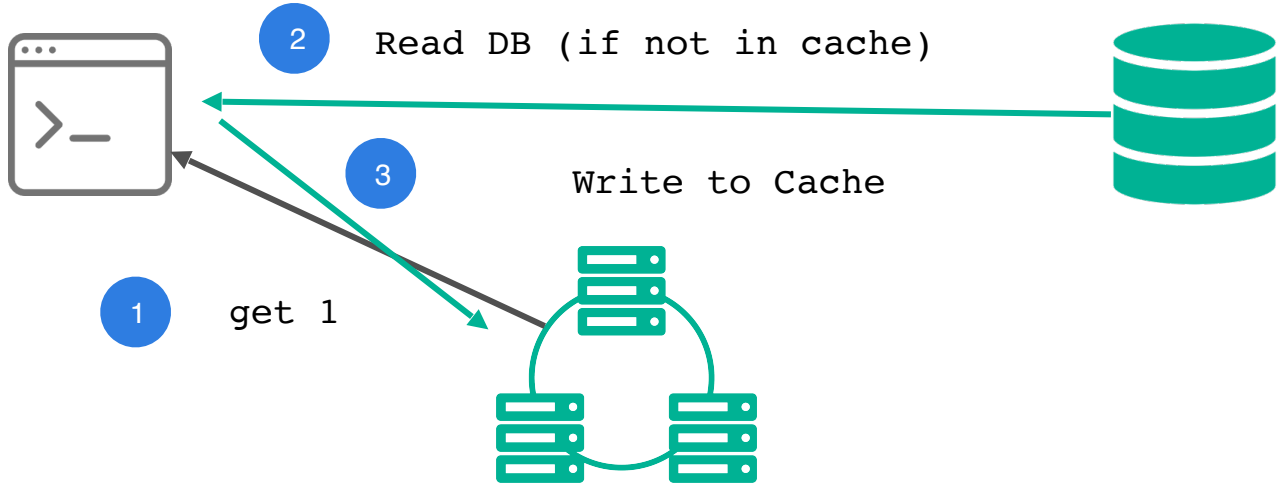
Persistent Data Overview



- Read-through persistence

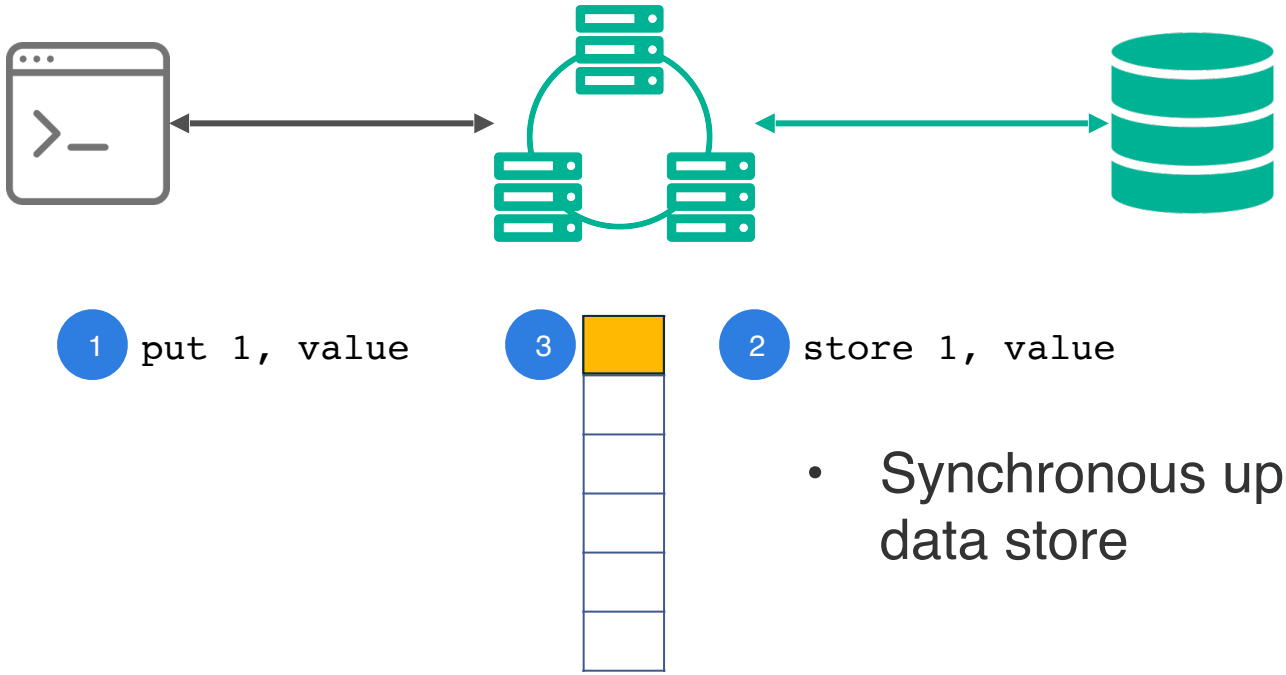
hazelcast

Look Aside Caching



hazelcast

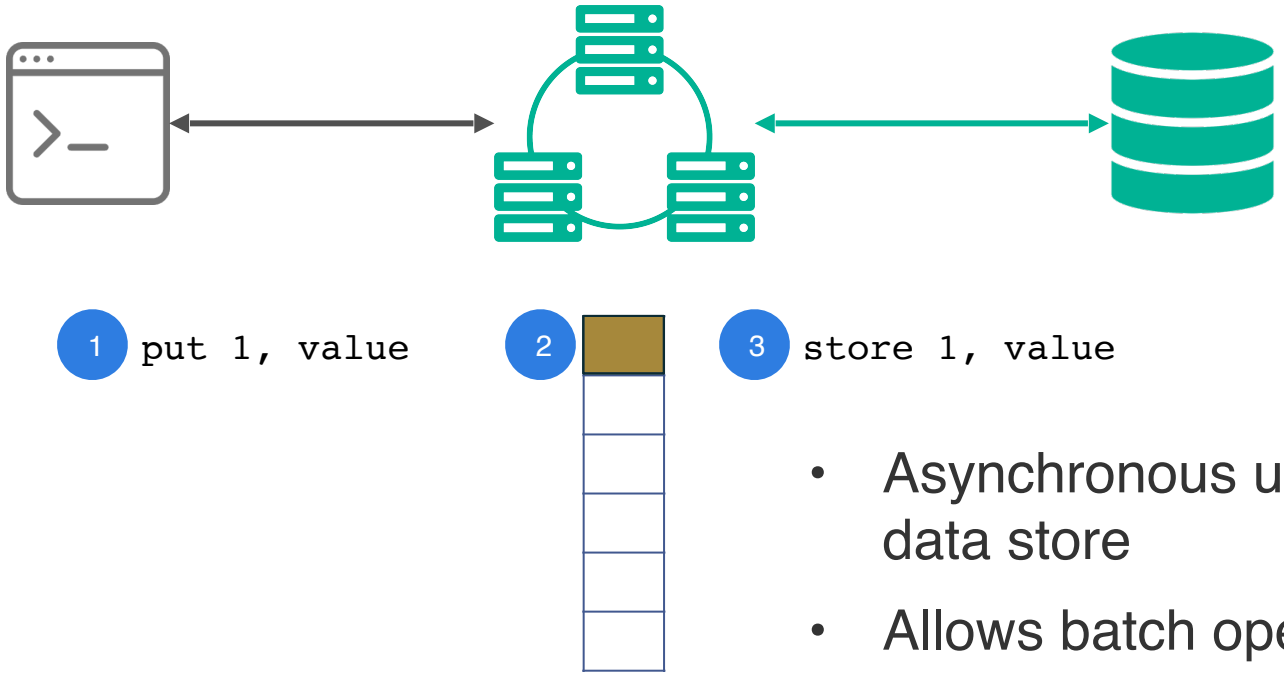
Write-Through Persistence



- Synchronous updates to data store

hazelcast

Write-Behind Persistence



- Asynchronous updates to data store
- Allows batch operations

hazelcast

Implementation

- Create class for MapLoader/MapStore
  - MapStore is subinterface of MapLoader
  - Class contains details for each type of transaction
    - `load`
    - `loadAll`
    - `delete`
    - `deleteAll`
    - `store`
    - `storeAll`
    - `loadAllKeys`
  - Class contains location of data store
- Reference class in map configuration

hazelcast

MapLoader

```
public class MyExternalConnectivity implements MapLoader<K, V> {
        public V load(K key) {
                return …;
        }
```

- When call is made for `IMap.get(K)`
  - If entry exists, it is returned; loader code is not called
  - If entry does not exist, loader code is called
    - If code returns null, caller gets null
    - If code returns non-null, entry is added to IMap and returned to caller

hazelcast

Pre-Loading

1. `getMap()` triggers `MapLoader.loadAllKeys`
    a. `loadAllKeys` returns `Iterable` that can be populated by queries
2. Member is selected as “key-loader” and retrieves all keys
3. Member distributes keys in batches to all other members
4. Members retrieve values using `MapLoader.LoadAll(Keys)`
    a. Initial Load Mode = LAZY, `getMap` call returns when partition is loaded
    b. Initial Lode Mode = EAGER, `getMap` call returns when entire map is loaded

hazelcast

MapStore

- When IMap contents change, `MapStore` code is called to update the external store

```
public class MyExternalConnectivity implements MapStore<K, V> {
          public void store(K key, V value) {
                              …;
          }
          public void delete(K key) {
                              …;
          }
//bulk methods
          public void storeAll(Map<K, V> map) {
                              …;
          }
          public void deleteAll(Collection<K> keys) {
                              …;
          }
```

hazelcast

Evict, Delete, Remove

Method	Erase in-memory	Erase from store	Return value
`map.evict`	Y	N	N
`map.delete`	Y	Y	N
`map.remove`	Y	Y	Y

- Use the right method for your desired outcome

hazelcast

Configuration

```
<map name="dataStoreMap">
   …
   <map-store enabled="true" initial-mode="LAZY">
     <class-name>com.hazelcast.examples.DummyStore</class-name>
     <write-delay-seconds>60</write-delay-seconds>
     <write-batch-size>1000</write-batch-size>
     <write-coalescing>true</write-coalescing>
   </map-store>
 </map>
```

hazelcast

Synchronization - Writing

- Can you accommodate temporary deviation?
  - Use write-behind persistence
  - Changes in IMap happen at memory speed
- Can’t accommodate temporary deviation?
  - Use write-through persistence
  - Changes in IMap wait on external store

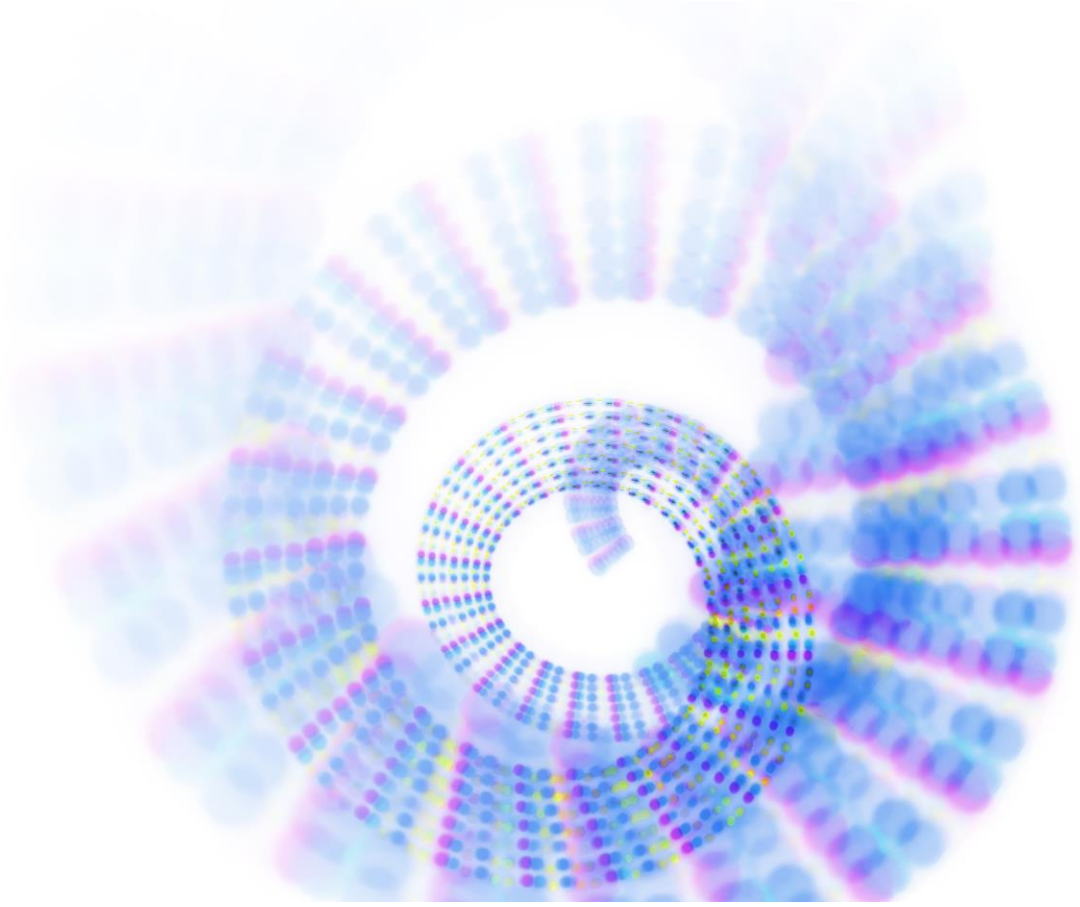
hazelcast

Synchronization - Reading

- What if data changes in external source?
  - IMap will continue to use out-of-date local data
- Periodically expire IMap data
  - MapLoader will refresh entry when needed
- Change data capture
  - Use http://www.striim.com/ to push changes from supported external stores into Hazelcast
  - Write your own version

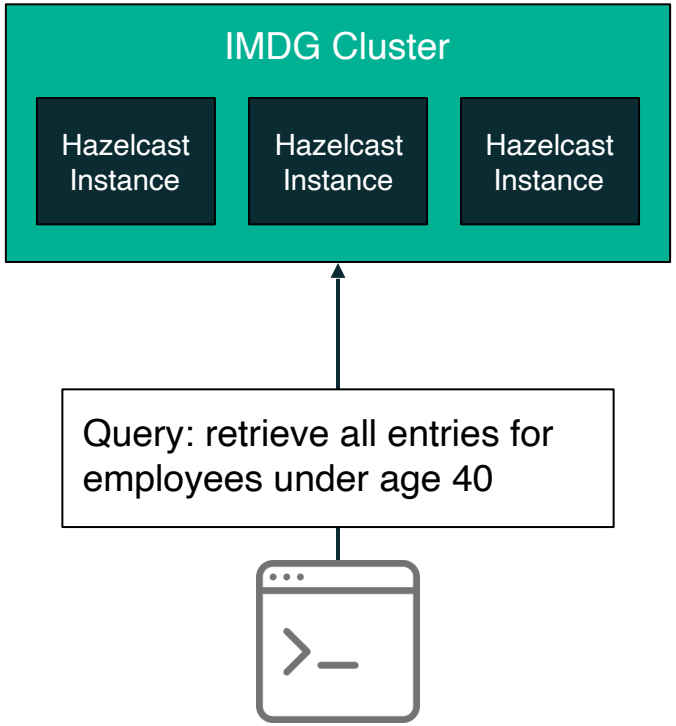
hazelcast

IMap: Queries



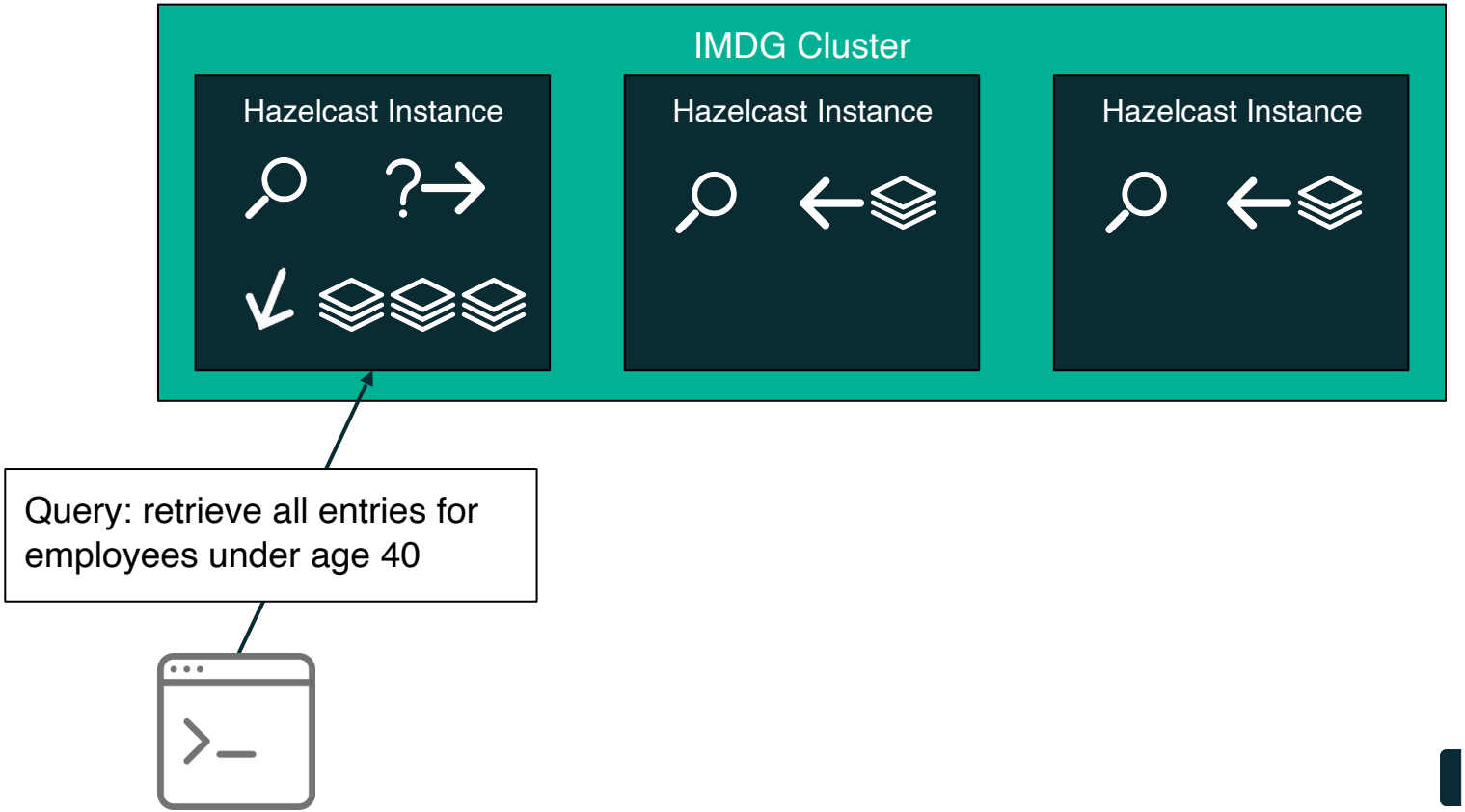
Hazelcast Distributed Query

- FIND data that matches a given criteria
- FILTER remotely to return only matching data to application
  - FIND/FILTER performed in parallel
  - Faster return - no unnecessary data, no wasted bandwidth



hazelcast

How Queries are Executed

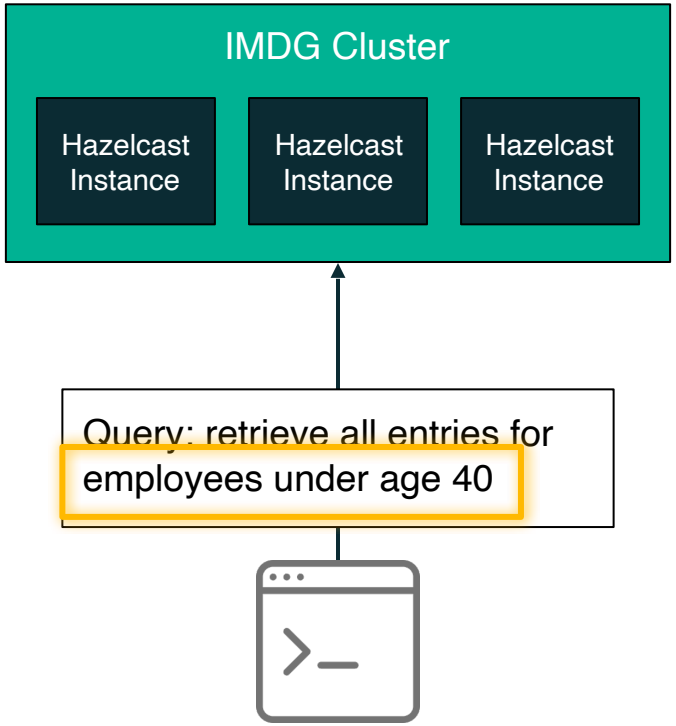


hazelcast

Predicates

```
interface Predicate<K, V> extends Serializable {
    boolean apply(Map.Entry<K, V> mapEntry);
}
```

- Used to construct the “search string”
- Only entries that match will be returned to the application
  - Uses “apply method”:
    if predicate = true, return entry



hazelcast

Predicates Two Ways...

Data Set:
Employee Objects:
name (String)
age (int)
salary (double)
active (boolean)

Query: return entries for active employees under the age of 30

Criteria API

```
IMap<String, Employee> map = hazelcastInstance.getMap( "employee" );

EntryObject e = new PredicateBuilder().getEntryObject();

Predicate ActiveUnder30 =
e.is( "active" ).and( e.get( "age" ).lessThan( 30 ) );

Collection<Employee> employees = map.values( ActiveUnder30 );
```

SQL Query

```
IMap<String, Employee> map = hazelcastInstance.getMap( "employee" );

Collection<Employee> employees = map.values( new SqlPredicate( "active
AND age < 30" ) );
```

hazelcast

Using the Criteria API

- Similar to Java Persistence Query Language
- Includes common operators
  - Comparison: equal, notEqual, greaterThan, greaterEqual, lessThan, lessEqual
  - Pattern matching: LIKE 'Alic%', ILIKE (case-insensitive), REGEX (for standard regular expressions)
  - Member test: name IN ('Bob', 'Alice')
  - Range: age BETWEEN 20 and 33
- Complete list of operators in Hazelcast Javadocs

hazelcast

Using AND, OR, and NOT

- Use to combine predicates
- Example: return all entries with a salary greater than $50,000 and are under age 35

```
IMap<String, Employee> employeeMap = hazelcastInstance.getMap("employee");

Predicate salaryPredicate = Predicates.greaterThan("salary", 50000);
Predicate agePredicate = Predicates.lessThan("age", 35);
Predicate salaryAndAge = Predicates.and(salaryPredicate, agePredicate);

Collection<Employee> result = employeeMap.values(salaryAndAge);
```

hazelcast

Using the PredicateBuilder class

- Same query: return all entries with a salary greater than $50,000 and are under age 35

```
IMap<String, Employee> employeeMap = hazelcastInstance.getMap("employee");

EntryObject entry = new PredicateBuilder().getEntryObject();

Predicate salaryAndAge =(entry.get("salary").greaterThan(50000)).and(entry.get("age").lessThan(35));

Collection<Employee> result = employeeMap.values(salaryAndAge);
```

hazelcast

Using Distributed SQL Query

- Allows you to write queries in SQL format
- Operators
  - Comparison: =, !=, >, >=, <, <=
  - Pattern matching: name LIKE 'Alic%', ILIKE (case-insensitive), REGEX (for standard regular expressions)
  - Member test: name IN ('Bob', 'Alice')
  - Range test: age BETWEEN 20 AND 33
- AND, OR, and NOT

hazelcast

Distributed SQL Query Example

- Same query: return all entries with a salary greater than $50,000 and are under age 35

```
IMap<String, Employee> employeeMap = hazelcastInstance.getMap( "employee" );

Collection<Employee> SalaryAndAge = employeeMap.values( new SqlPredicate( "salary > 50000 AND age < 35" ) );
```

- Behind the scenes: SqlPredicate converts SQL format into predicate tree (as in Criteria API)
  - Can use both types of queries interchangeably

hazelcast

IMap with JSON

- IMap supports JSON objects
- Query as you would any other object

```
String person1 = "{ \"name\": \"John\", \"age\": 35 }";
String person2 = "{ \"name\": \"Jane\", \"age\": 24 }";
String person3 = "{ \"name\": \"Trey\", \"age\": 17 }";

IMap<Integer, HazelcastJsonValue> idPersonMap = instance.getMap("jsonValues");

idPersonMap.put(1, new HazelcastJsonValue(person1));
idPersonMap.put(2, new HazelcastJsonValue(person2));
idPersonMap.put(3, new HazelcastJsonValue(person3));

Collection<HazelcastJsonValue> peopleUnder21 =
idPersonMap.values(Predicates.lessThan("age", 21));
```

hazelcast

JSON to Java “Translation”

- Hazelcast IMDG treats JSON values as Java types for comparison to query

JSON Primitive	Java Primitive
`number`	`long` if possible `double` if not
`string`	`String`
`true`	`boolean`
`false`	`boolean`
`null`	`null`

JSON Metadata

- Hazelcast stores a metadata object per `HazelcastJsonValue`
  - Speeds querying throughput
  - Increases put latency
- Stored on-heap even if data itself is in `NATIVE` off-heap memory
- Disable if put performance is more important than throughput

```
<map name="map-a">
    <!--
    valid values for metadata-policy are:
      - OFF
      - CREATE_ON_UPDATE (default)
    -->
    <metadata-policy>OFF</metadata-policy>
</map>
```

```
MapConfig mapConfig = new MapConfig();
 mapConfig.setMetadataPolicy(MetadataPolicy.OFF);
```

hazelcast

Custom Predicates

- If standard predicates do not meet your needs, use PREDICATE interface
- Example: return entries with name length = 10

```
class NameLengthPredicate implements Predicate<int, Employee> {
    @Override
    public boolean apply(Map.Entry<int, Employee> entry) {
        Employee person = entry.getValue();
        return person.getName().length() == 10;
    }
}
```

Pros:

Flexible

Possibly more performant than SQL-like queries for full scans

Cons:

Lots of boilerplate code

Does not support indexing

hazelcast

Paging Predicate

- Allows retrieval of data page by page
  - Specify predicate
  - Specify page length

```
IMap<String, Employee> employeeMap = hazelcastInstance.getMap( "employee" );

EntryObject entry = new PredicateBuilder().getEntryObject();
Predicate salaryAndAge = ( entry.get( "salary" ).greaterThan( 50000 ) ).and( entry.get
("age").lessThan( 35 ) );

PagingPredicate PageLength10 = new PagingPredicate( salaryAndAge, 10);
Collection<Employee> getWithSalaryandAge = employeeMap.values( PageLength10 );

//do something with the entries ... this is omitted here
PageLength10.nextPage();
getWithSalaryandAge = employeeMap.values( PageLength10 );
```

hazelcast

Paging Predicate and Comparators

- Entries forming a sequence may be ordered using a Comparator

```
Comparator<Map.Entry> ageOrderComparator = new Comparator<Map.Entry>() {
   public int compare(Employee e1, Employee e2) {
        Employee Em1 = (Employee) e1.getValue();
        Employee Em2 = (Employee) e2.getValue();
        return e2.getAge() - e1.getAge();
   }
PagingPredicate PageLength10 = new PagingPredicate( salaryAndAge,
ageOrderComparator, 10);
```

- If no explicit comparator is used, entries will be ordered naturally
  - Entries must implement java.lang.Comparable

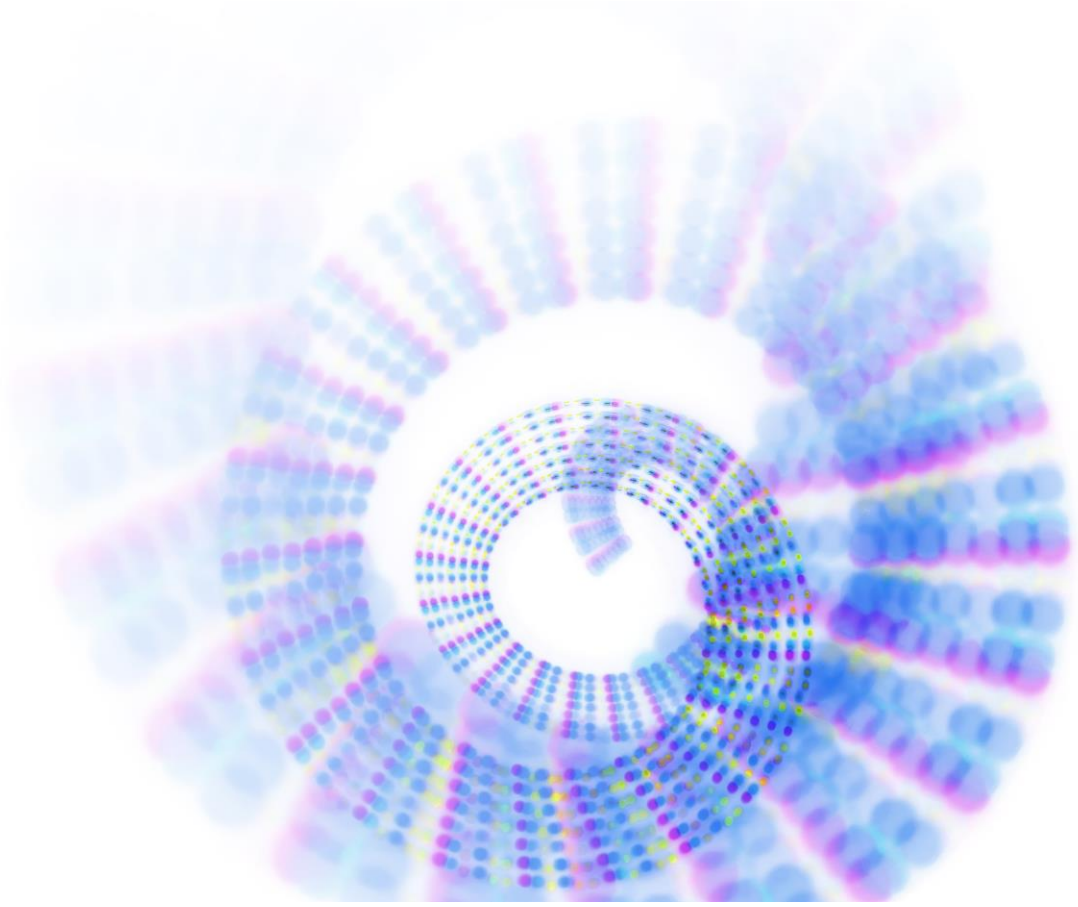
hazelcast

Paging Predicate Caveats

- Each node might send a full page of results to the caller
  - the caller wants a full page of results
  - each node doesn't know how many results the others will send so it has to set a full page just in case
  - the caller discards the excess
- For 10 nodes and a page size of 5, you might get 10 * 5 = 50 results
- For 100 nodes and a page size of 50, you might get 100 * 50 = 5,000 results
- Potential to overflow the caller's memory
  - Not usually a problem, but just be aware

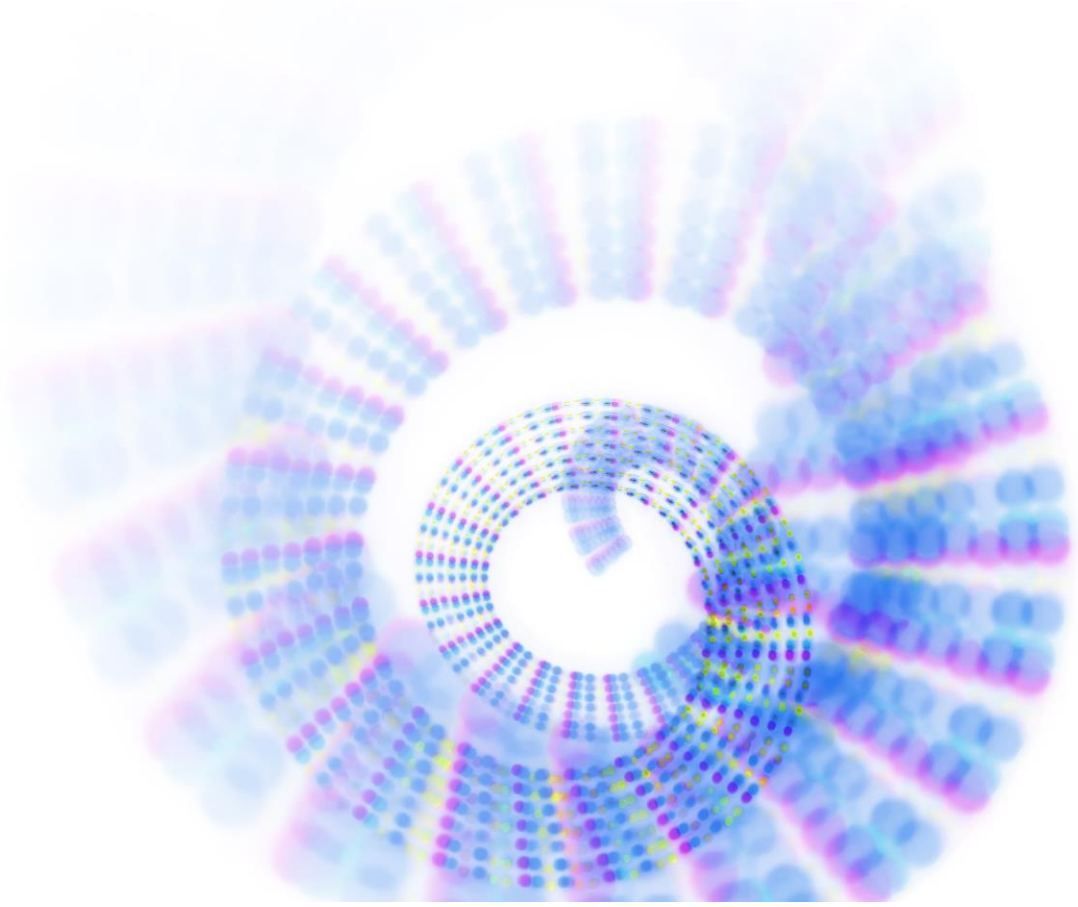
hazelcast

End of Day 1

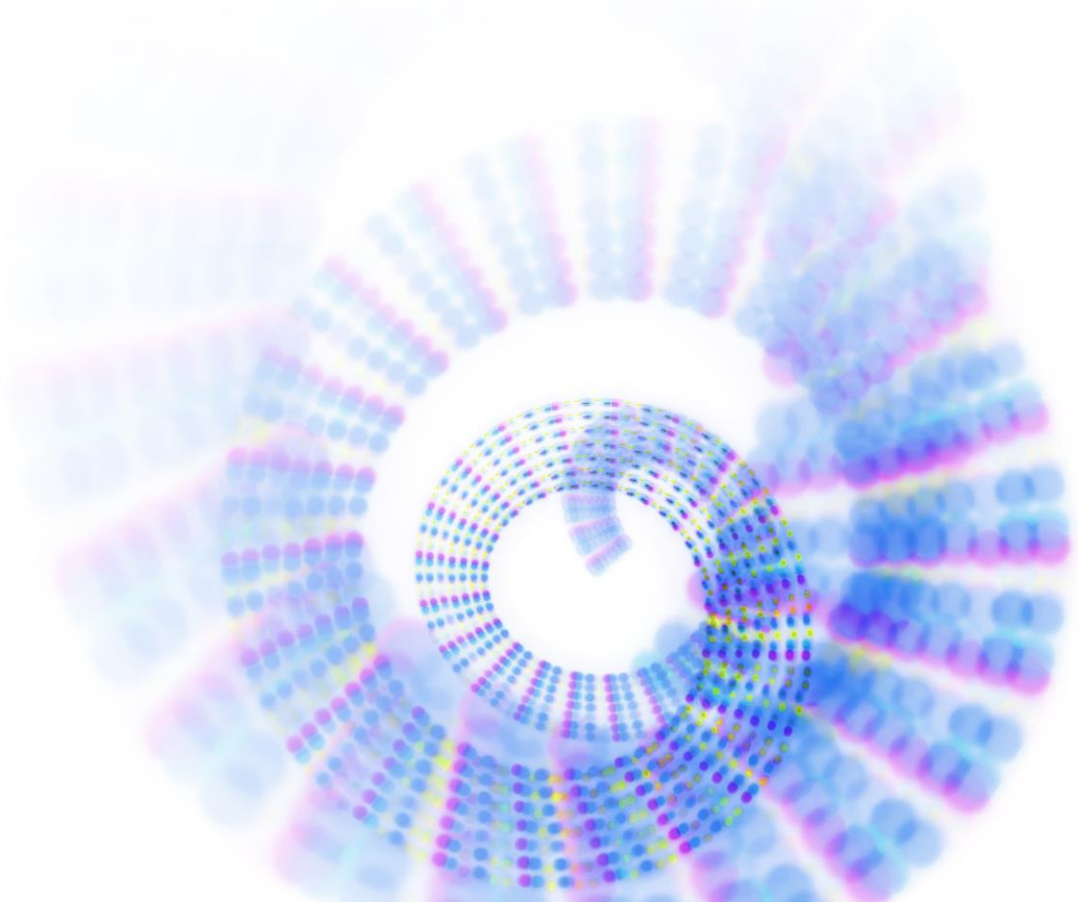


Q and A

Next Up … IMDG



Indexing



Indexes

- Speed up queries by avoiding full scans
  - Without indexing, objects have to be de-serialized
- Two types
  - Ordered - for order comparison queries (range queries)

```
new SqlPredicate("age > 33")
new SqlPredicate("age > 33 and age <= 66")
new SqlPredicate("age between 33 and 66")
```

  - Unordered – for equality queries (point queries)

```
new SqlPredicate("name = 'Bob'")
new SqlPredicate("name in ('Alice', 'Bob')")
```

hazelcast

Using Indexing

- Recommendation to use with primitive types for best speed
  - byte, short, int, long, float, double, String
- Only use ordered indexing with orderable values (numbers)
  - Downside: index needs reordering when data is updated
- Unordered indexing does not prevent full scan
  - Performance gain comes from no de-serialization

hazelcast

Composite Indexing

- Use when frequent query is an AND
- Unordered composite index
  - All predicate values are point queries

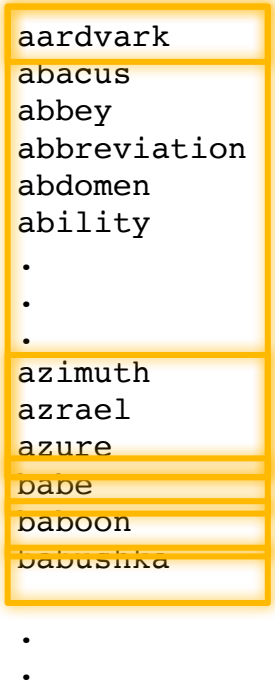
```
new SqlPredicate("name = 'Bob' and age = 33")
```

- Ordered composite index
  - Multiple point queries plus one ordered predicate

```
new SqlPredicate("name = 'Bob' and salary > 50000")
```

hazelcast

How Composite Indexing Works



- Point query = “prefix”
- Searches that are fully answered by index

```
new SqlPredicate("letter1 = a")
new SqlPredicate("letter1 > a")
new SqlPredicate("letter1 = a and letter2 <= m")
new SqlPredicate("letter1 = a and letter2 =z and letter3 > a")
```

- Searches that are partially answered by index

```
new SqlPredicate("letter1 = a and letter 2 > b and letter 3 = a")
new SqlPredicate("letter1 = a and letter 2 > b and letter 3 > a")
```

hazelcast

Setting Up Indexing

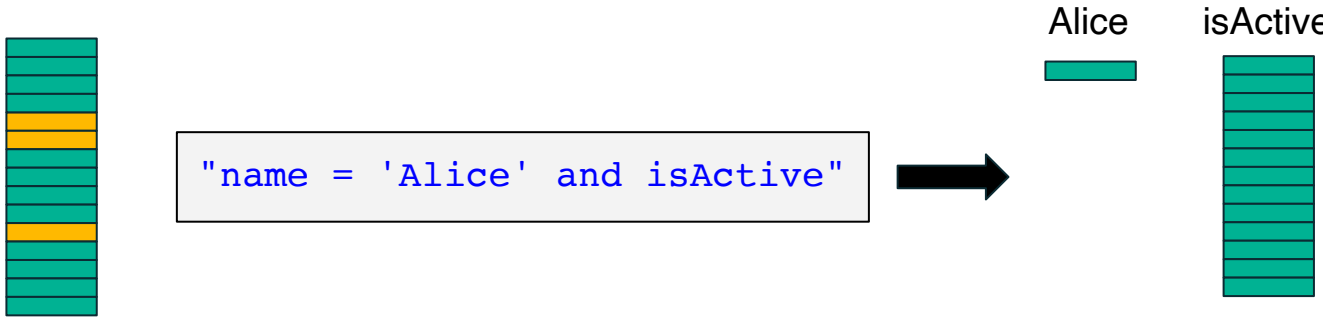
- Set up in configuration
  - STRONGLY recommend using declarative

```
<map name="employeeMap">
    <indexes>
        <index ordered="false">name</index>
        <index ordered="true">age</index>
            <index ordered="false">name,age</index>
            <index ordered="true">name,salary</index>
</indexes>
</map>
```

hazelcast

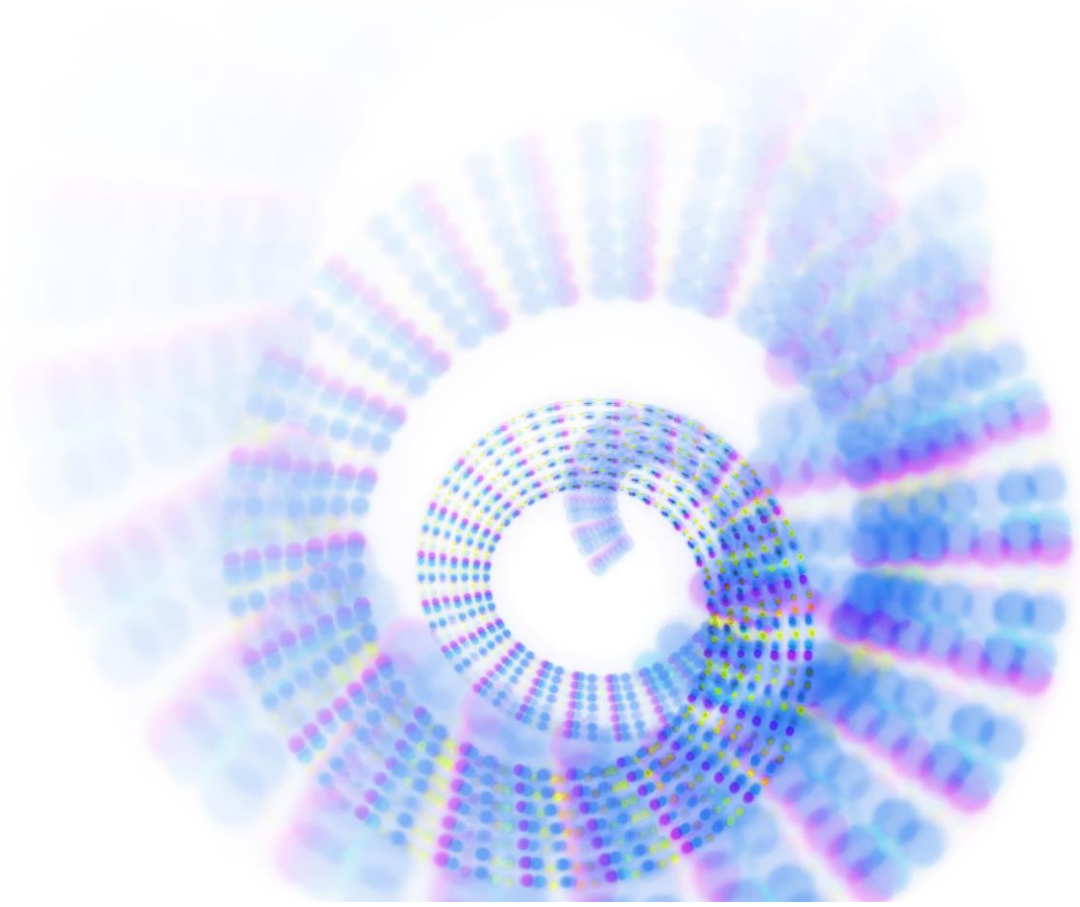
Indexing Cautions

- Unordered indexes are very slow at performing range queries
- Each index consumes memory
- Custom predicates need to implement com.hazelcast.query.IndexAwarePredicate
- Bad selectivity = bad performance
  - Example - boolean attribute `isActive`

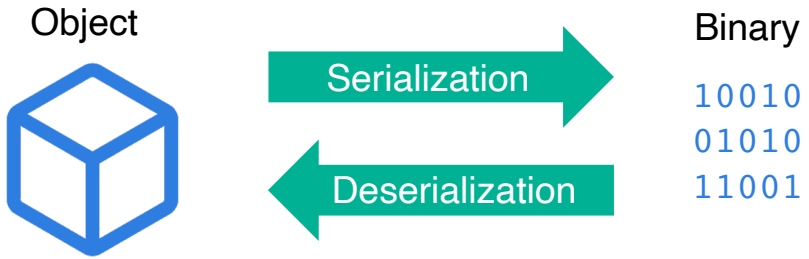


hazelcast

Serialization



Serialization



- Objects can't be sent via network
  - Most Hazelcast operations are remote, so serialization is all over the place and often happens multiple times
- The faster the serialization algorithm, the faster is Hazelcast IMDG!

Serialization Examples

- IMap put

```
String oldValue = map.put(key, value);
```

deserialization — serialization

- IMap set

```
map.set(key, value);
```

serialization

- IMap get

```
String value = map.get(key);
```

deserialization — serialization

hazelcast

Types Optimized “Out of the Box”

Byte / byte

Boolean / boolean

Character / char

Short / short

Integer / int

Long / long

Float / float

Double / double

Byte[ ] / byte[ ]

Boolean[ ] / boolean[ ]

Character[ ] / char[ ]

Short[ ] / short[ ]

Integer[ ] / int[ ]

Long[ ] / long[ ]

Float[ ] / float[ ]

Double[ ] / double[ ]

String

Date

BigInteger

BigDecimal

Class

Enum

ArrayList

LinkedList

HashMap

TreeMap

hazelcast

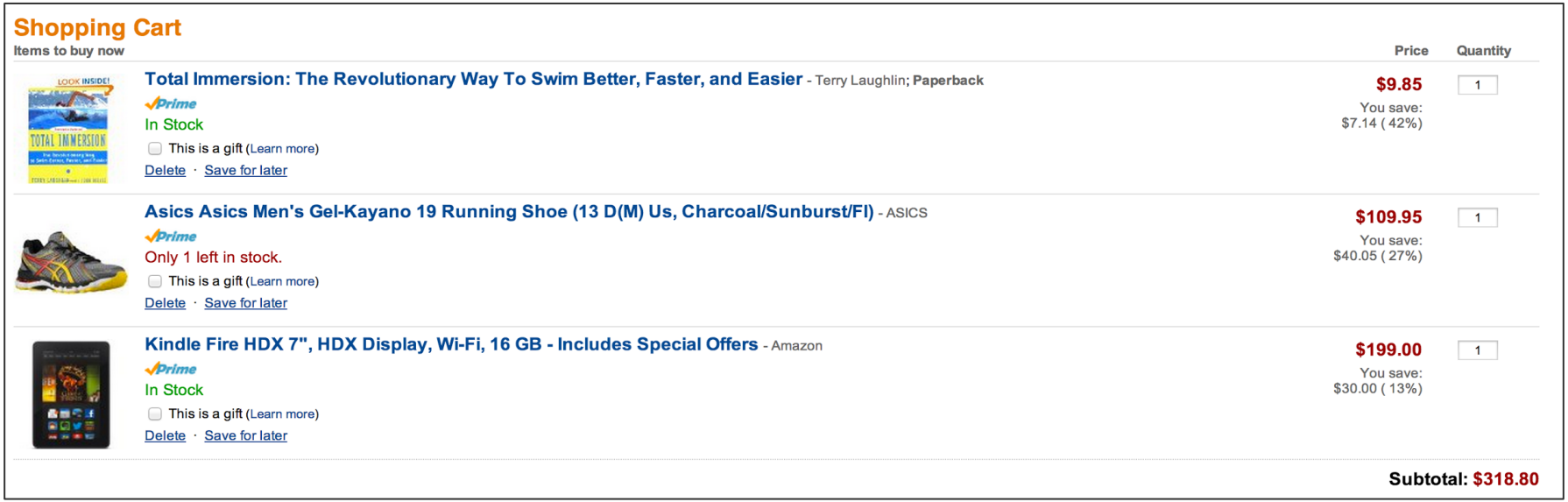
Serialization Options

- Java standard serialization
  - java.util.Serializable (only recommended for prototyping)
  - java.util.Externalizable
- Hazelcast optimized serialization
  - com.hazelcast.nio.serialization.DataSerializable
  - com.hazelcast.nio.serialization.IdentifiedDataSerializable
  - com.hazelcast.nio.serialization.Portable
- External or custom serialization
  - com.hazelcast.nio.serialization.ByteArraySerializer
  - com.hazelcast.nio.serialization.StreamSerializer

hazelcast

Which to Use?

- Consider this shopping cart



hazelcast

java.util.Serializable

```
public class ShoppingCartItem implements Serializable {
    public long cost;
    public int quantity;
    public String itemName;
    public boolean inStock;
    public String url;
}
```

- Pros
  - Java standard
  - Works out of the box – no custom code
- Cons
  - Least efficient in terms of CPU load
  - Biggest resulting payload
  - Uses reflection during deserialization
  - Does not support attribute changes !

hazelcast

java.util.Externalizable

```
public void writeExternal(ObjectOutput out) throws ...
{
  out.writeLong(cost);
  out.writeInt(quantity);
  out.writeUTF(itemName);
  out.writeBoolean(inStock);
  out.writeUTF(url);
}
```

```
public void readExternal(ObjectInput in)
throws ... {
  cost = in.readLong();
  quantity = in.readInt();
  itemName = in.readUTF();
  inStock = in.readBoolean();
  url = in.readUTF();
}
```

- Pros
  - Java Standard
  - More efficient than Serializable in terms of CPU and memory
- Cons
  - Requires the developer to implement the serialization code
  - Uses reflection during deserialization
  - Does not support attribute changes !

hazelcast

com.hazelcast.nio.serialization.DataSerializable

```
public void writeData(ObjectDataOutput out) throws ...
{
  out.writeLong(cost);
  out.writeInt(quantity);
  out.writeUTF(itemName);
  out.writeBoolean(inStock);
  out.writeUTF(url);
}
```

```
public void readData(ObjectDataInput in)
throws ... {
  cost = in.readLong();
  quantity = in.readInt();
  itemName = in.readUTF();
  inStock = in.readBoolean();
  url = in.readUTF();
}
```

- Pros
  - More efficient than Serializable in terms of CPU and memory
- Cons
  - Requires the developer to implement the serialization code
  - Uses reflection during deserialization
  - Does not support attribute changes !

hazelcast

com.hazelcast.nio.serialization.IdentifiedDataSerializable

```
public void writeData(ObjectDataOutput out) throws ...
{
  // like DataSerializable
}

public void readData(ObjectDataInput in) throws ... {
  // like DataSerializable
}
```

```
public int getFactoryId() { return 1; }

public int getId() { return 1; }
```

```
public class MyFactory implements
DataSerializableFactory {
  public IdentifiedDataSerializable create(int typeId)
{
    switch (typeId) {
      case 1:
        return new ShoppingCartItem();
    }
    throw new RuntimeException("Unknown type");
  }
}
```


hazelcast

com.hazelcast.nio.serialization.IdentifiedDataSerializable

- Pros
  - More efficient than Serializable in terms of CPU and memory
  - Doesn’t need reflection
  - Cross-Language support
- Cons
  - Requires the developer to implement the serialization code
  - Requires the developer to implement a factory
  - Does not support attribute changes !

hazelcast

com.hazelcast.nio.serialization.Portable

```
public void writePortable(PortableWriter out) throws ... {
    out.writeLong("cost", cost);
    out.writeInt("quantity", quantity);
    out.writeUTF("name", itemName);
    out.writeBoolean("stock", inStock);
    out.writeUTF("url", url);
}
```

```
public void readPortable(PortableReader in) throws ... {
    url = in.readUTF("url");
    quantity = in.readInt("quantity");
    cost = in.readLong("cost");
    inStock = in.readBoolean("stock");
    itemName = in.readUTF("name");
}
```

```
public void writePortable(PortableWriter out) throws ... {
  // …
}

public void readPortable(PortableReader in) throws ... {
  // …
}
```

```
public int getFactoryId()
{
  return 2;
}

public int getClassId() {
  return 1;
}
```

```
public class MyFactory implements PortableFactory {
  public Portable create(int typeId) {
    switch (typeId) {
      case 1:
        return new ShoppingCartItem();
    }
    throw new RuntimeException("Unknown type");
  }
}
```


hazelcast

com.hazelcast.nio.serialization.Portable

- Pros
  - More efficient than Serializable in terms of CPU and memory
  - Doesn’t need reflection
  - Cross-Language support
  - Partial deserialization for queries
  - Supports attribute changes
- Cons
  - Requires the developer to implement the serialization code
  - Requires the developer to implement a factory
  - Additional metadata to be sent

hazelcast

com.hazelcast.nio.serialization.ByteArray/StreamSerializer

```
public interface ByteArraySerializer<T> extends Serializer {
    byte[] write(T object) throws IOException;
    T read(byte[] buffer) throws IOException;
}
```

```
public interface StreamSerializer<T> extends Serializer {
    void write(ObjectDataOutput out, T object) throws
IOException;
    T read(ObjectDataInput in) throws IOException;
}
```

- Great for separation of concerns, does not implementation of an interface
- Convenient and flexible
- Often used for external serialization frameworks (like protobuffers, Apache Avro, Apache thrift, kryo, etc)

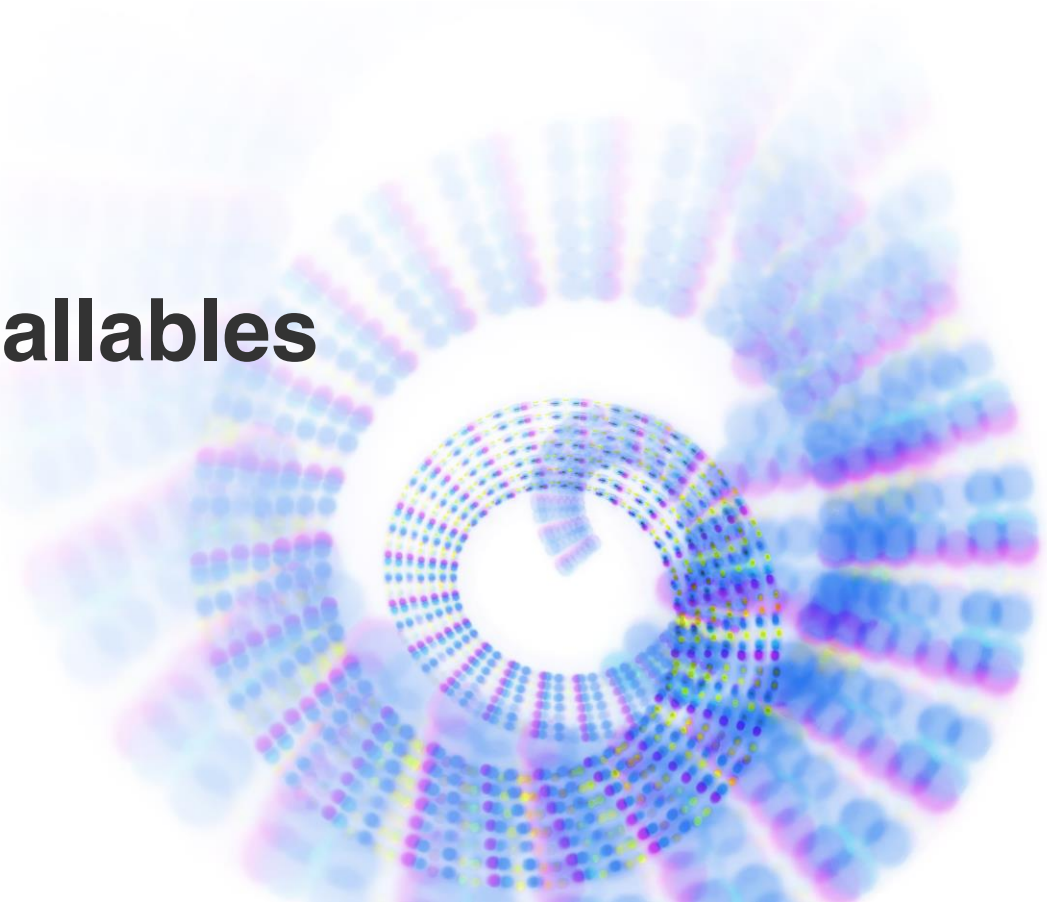
For kryo see: https://github.com/jerrinot/subzero

hazelcast

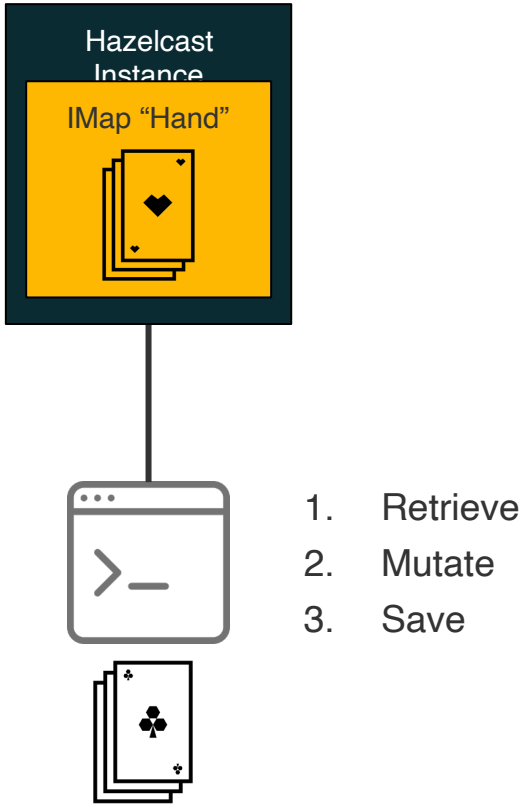
Simple Performance Test

	Read Ops/ms	Write Ops/ms	Payload
Serializable	56	46	514
Externalizable	67	70	228
DataSerializable	80	78	261
IdentifiedDataSerializable (supported by all Native Clients)	80	78	192
Portable (supported by all Native Clients)	73	68	417
Kryo (implemented with StreamSerializer)	70	57	198

Runnables and Callables



What's Wrong With This Application?



1. Retrieve
2. Mutate
3. Save

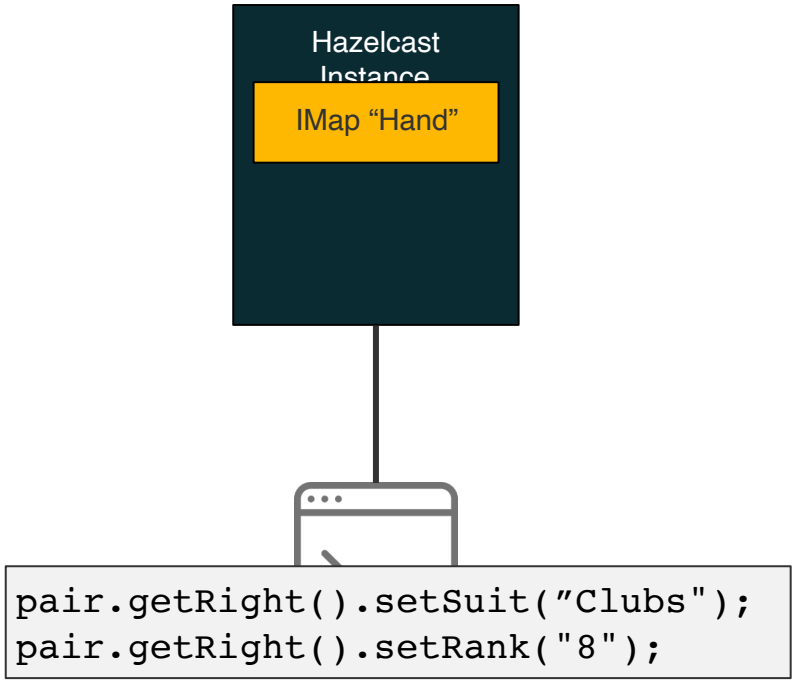
A. Nothing
   - It is easy to understand and for junior developers to maintain

B. Data moves across the network
   - Not good for big data

C. Another client could update the cluster while this client is processing
   - Not good for concurrent access

hazelcast

A Better Approach – Push the Code



- Small network packet
  - Faster!
- Working on the master copy
  - No concurrent modification worries
- Needs processing code on IMDG server classpath
  - Extra deployment complication

Push Code Options

`com.hazelcast.map.EntryProcessor`

- Process one entry at a time
- Process code should be fast
- Is specialized and optimized
  - More on this later

`java.lang.Runnable`

`java.util.concurrent.Callable`

- Can read/mutate multiple entries
- Suitable for longer running processing

hazelcast

Runnable and Callable

- Java standard tasks
- Standard Java: run in another thread on same JVM
- Hazelcast IMDG: thread can be in different JVM (cluster member)
  - An IMDG client or IMDG server invokes the runnable/callable
    - It runs on one of the servers in the IMDG
    - It runs on all of the servers in the IMDG in parallel
    - It runs on a subset of the servers in the IMDG in parallel

hazelcast

The Callable

You need a way to transmit the callable (the constructor arguments) to where it will run

```
public class MySleepyCallable implements Callable<String>, Serializable {

    private long duration;

    public MySleepyCallable(long arg0) {
          this.duration = Math.abs(arg0);
    }

    public String call() throws Exception {
          log.info("Start run of {} for duration {}ms", this, this.duration);

          while (this.duration > STEP) {
                this.duration -= STEP;
                Thread.sleep(STEP);
          }

          Thread.sleep(this.duration);

          log.info("End run of {}", this);
          return "Ended at " + new Date();
    }
```


hazelcast

The Caller's Side

```
DurableExecutorService durableExecutorService =
MyClient.hazelcastInstance.getDurableExecutorService("lab2");

MySleepyCallable myCallable = new MySleepyCallable(duration);

//submit to grid
DurableExecutorServiceFuture<String> future =
durableExecutorService.submit(myCallable);

long taskId = future.getTaskId();

Future<String> future = durableExecutorService.retrieveResult(taskId);

//callable result
String result = future.get(1000, TimeUnit.MILLISECONDS);
```

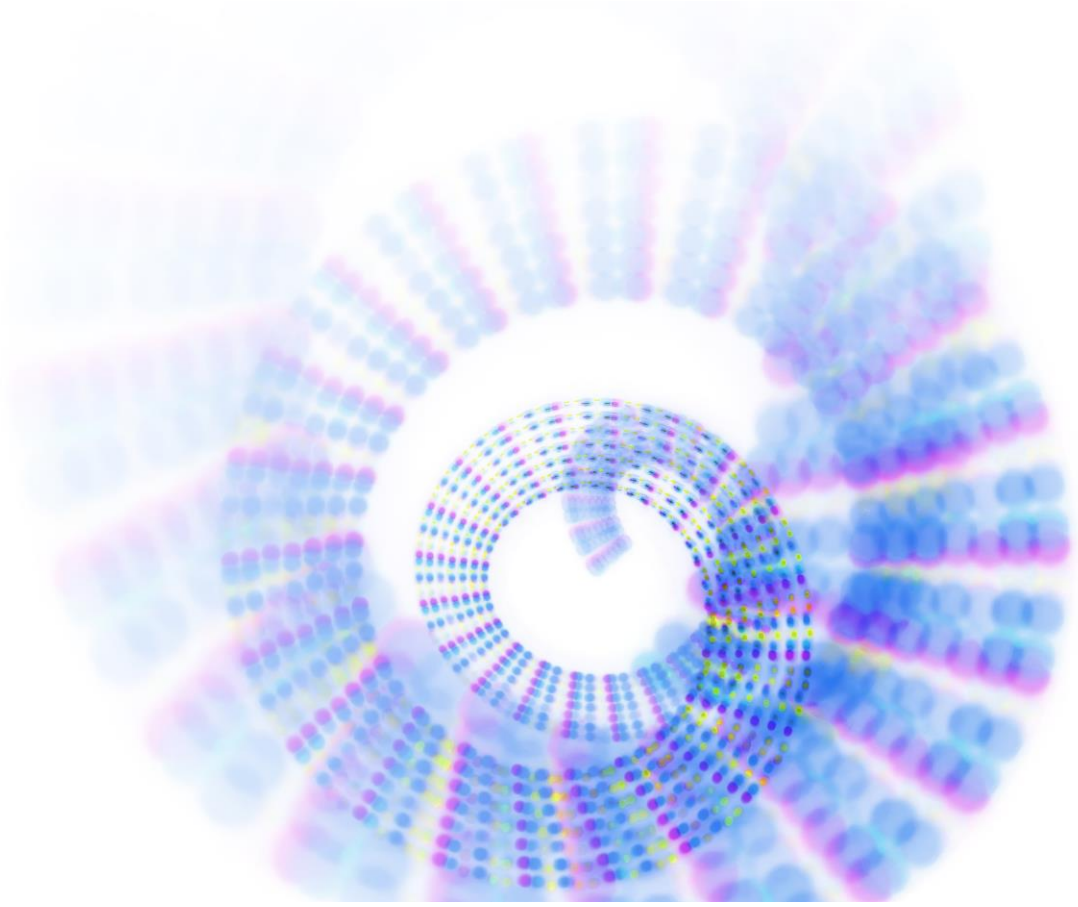
The Callee's Side

- Configure a named execution pool on each server

```
<executor-service name="default">
        <pool-size>16</pool-size>
        <queue-capacity>0</queue-capacity>
</executor-service>
```


hazelcast

Entry Processor

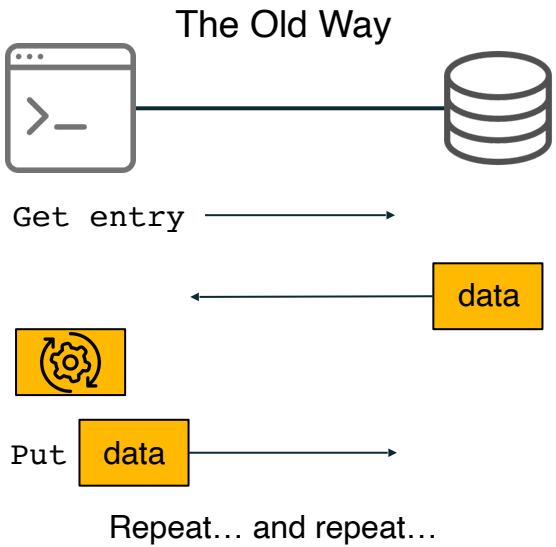


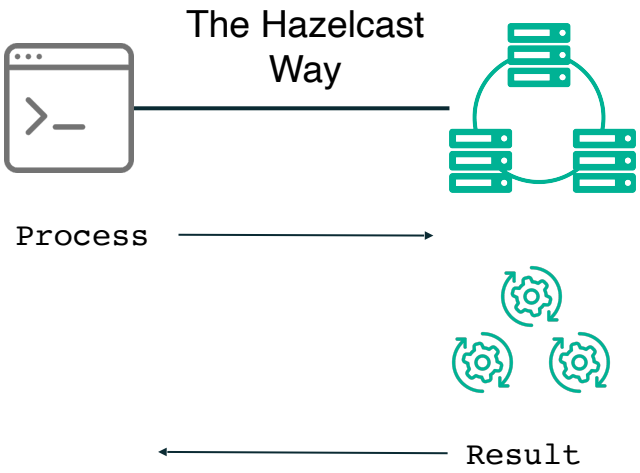
Entry Processor

- Performs atomic computation on entries stored in distributed maps
- Used for bulk processing of IMap data
  - Distributed in-memory processing on cluster
  - Eliminates client-server network hops

hazelcast

Entry Processor





hazelcast

Entry Processor Characteristics

- No issues with locks or concurrency
- Supports predicates to limit processing to a subset of the map data
- Supports indexing to increase processing performance
- Operate best when in-memory format is OBJECT
- Read-only, mutating (read-write), and offloadable processors

Entry Processor Advantages

- No concurrency worries
  - The entry is yours til you're done, but be quick!
- Great for delta processing
  - Changing a small proportion of a big object
    - Pass the changing fields as arguments e.g. new delivery date for a big order
    - The changing fields can be implied rather than sent e.g. `OrderCancelledEntryProcessor` changes order status
- Great if the processing is fast, just like a new value replacement
- For processing a single entry, not for looking at other entries

hazelcast

Entry Processor Disadvantages

- Must be on the server's classpath
- Not good if processing is slow
  - Reads and writes to the same key queue up
  - Offload processing can address this issue

hazelcast

Entry Processor Interface

```
interface EntryProcessor<K, V> extends Serializable {
    Object process(Map.Entry<K, V> entry);
    EntryBackupProcessor<K, V> getBackupProcessor();
}
```

- The process method is a core of an entry processor
  - Accepts map entry, performs process, returns result
  - If there is no result, may return null
- Entry backup processor executes on backup partitions to update backups if primary was updated
  - Must be explicitly defined if using entry processor interface

hazelcast

Abstract Entry Processor Class

```
public abstract class AbstractEntryProcessor<K, V>
extends Object
implements EntryProcessor<K, V>
```

- Convenient base for implementing entry processors
- Backup processor already implemented
  - If no backup processing is required (e.g. read-only), return null

hazelcast

Entry Processor Example

```
private static class IncrementingEntryProcessor extends AbstractEntryProcessor<Integer,
Integer> {
    public Object process(Map.Entry<Integer, Integer> entry) {
        Integer value = entry.getValue()

        //Process and modify the value. We are just incrementing it by 1.
        value=(value + 1);

        //Return result of processing to map.
        entry.setValue(value);

        // here we're not returning any results from this class.
        return null;
    }

}
```

hazelcast

Executing Entry Processors

mapname.executeOnEntries	Process all entries in a map
mapname.executeOnEntries *Predicate*	Process all entries that match the specified predicate
mapname.executeOnKey	Process an entry mapped by a key
mapname.executeOnKeys *Collection*	Process an entry mapped by a collection of keys
mapname.submitToKey	Asynchronously process an entry mapped by a key

```
IMap<Integer, Integer> sampleMap = hazelcastInstance.getMap("myMap");
sampleMap.executeOnEntries( new IncrementingEntryProcessor() );
```

hazelcast

Mutating Entry Processor

- Modify map entries
- Optionally return results from the process method
  - Collecting results takes cluster memory
- Recommended to use Abstract EntryProcessor
- `executeOnKey` and `submitToKey` will not run on locked keys

Read-Only Entry Processor

- Does not modify map entries
- Marked with `ReadOnly` interface
  - Skips entry locking logic
- `UnsupportedOperationException` is thrown if a read-only EP attempts to modify an entry
- Optionally return results from the process method

Entry Processor Performance

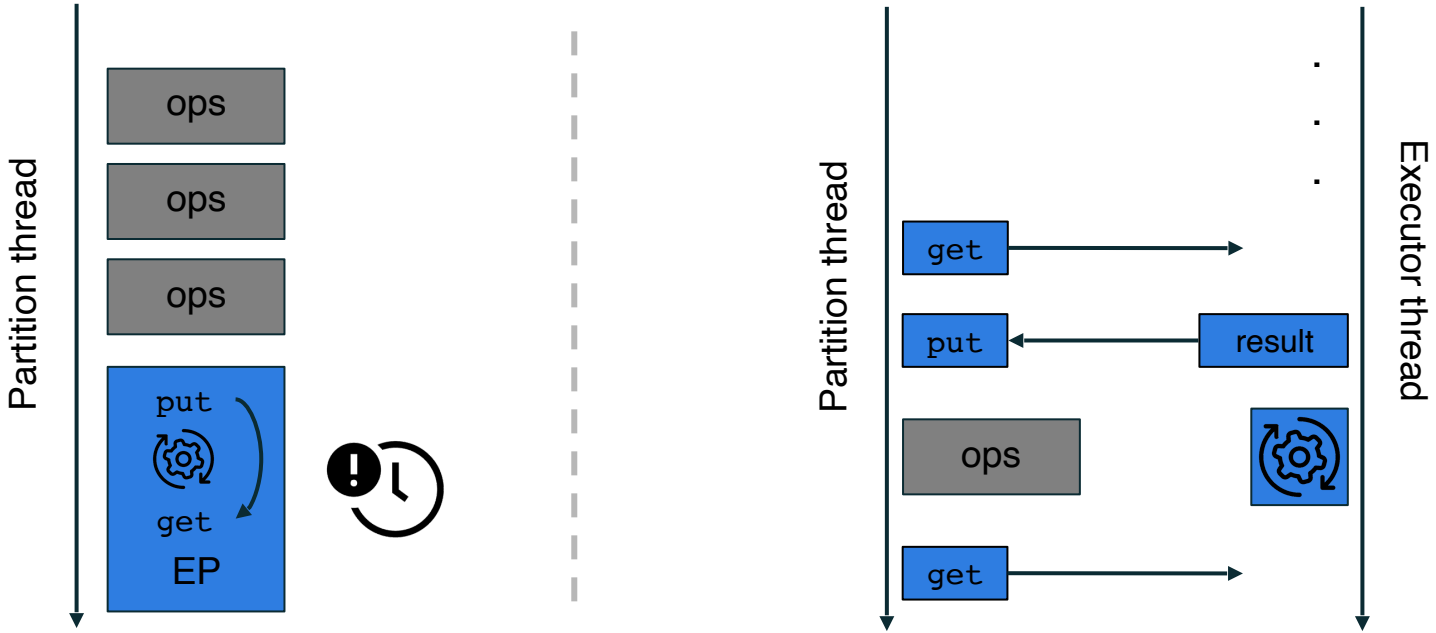
- Design of entry processor assumes fast execution of the process method
- What can slow things down?
  - Extremely large maps - use predicates, indexing, or process by key
  - BINARY entry format - incurs serialization/deserialization cost
  - Heavy code
- Check during development with slow code detector system properties

```
hazelcast.slow.operation.detector.enabled - set to true (default)
hazelcast.slow.operation.detector.threshold.millis - set to 1 ms (default is 10000)
```

hazelcast

Offloadable Entry Processor

- Offloads execution processing to separate executor thread



hazelcast

Implementing Read Only/Offloadable

- Add to EP

```
private static class MyEntryProcessor extends AbstractEntryProcessor<Integer, Integer>
     implements ReadOnly {
```

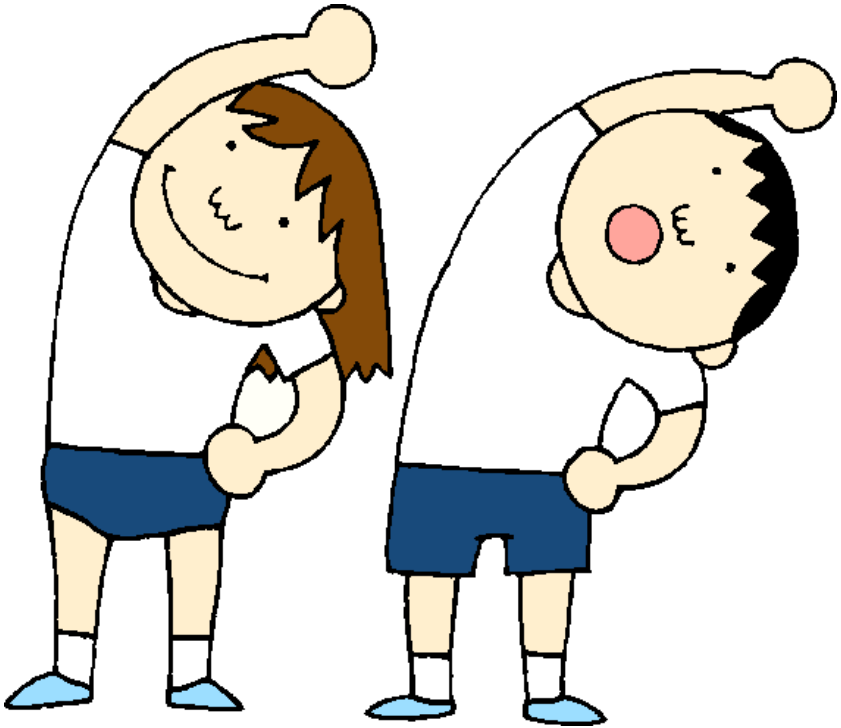
```
private static class MyEntryProcessor extends AbstractEntryProcessor<Integer, Integer>
     implements Offloadable {
```

```
private static class MyEntryProcessor extends AbstractEntryProcessor<Integer, Integer>
     implements Offloadable, ReadOnly {
```

- Offloadable only applies to `executeOnKey` or `submitToKey`

hazelcast

Stretch/Bio Break

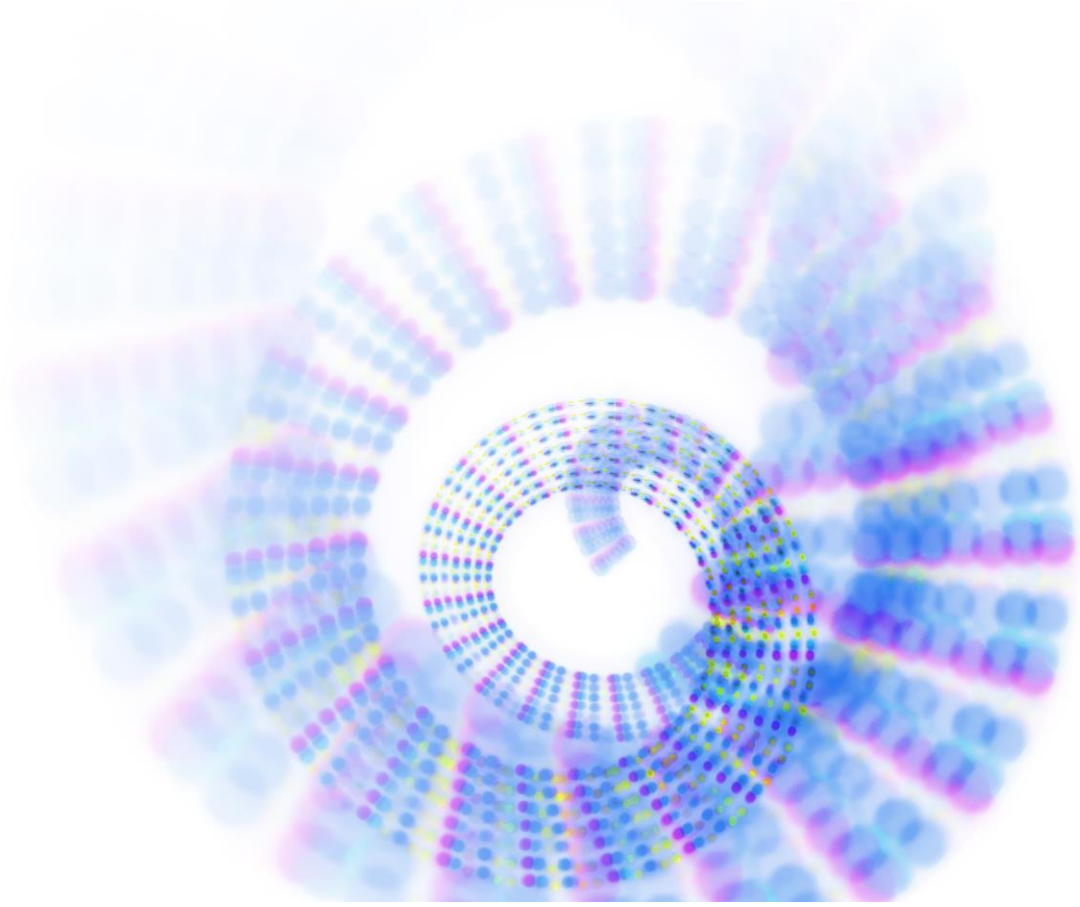


15 Minute Break

hazelcast

Entry Listener

15 minute break



Thinking Reactive

- E-Commerce application with multiple IMDG servers
  - We want to count the value of today's sales
- Good
  - We could do a "cursor"
    - query for matching records, bring them to the caller, total the order amounts
  - simple to understand and easy to maintain
- Better
  - We could submit a Callable to the grid to total each server's orders
    - Parallelizable and scalable, each server tallies only the data it holds
    - only the subtotal from each Callable instance passes across the network
  - slightly harder to understand, but faster
- Best
  - Update the totals as orders as place
    - What facilitates this reactive style ? Event listeners

hazelcast

Hazelcast Eventing Framework

- Events generate notifications
  - Data changing in IMap
  - Data added to IQueue
  - Data migration
  - Cluster member add/drop
  - Etc.
- Events only published if something is listening

hazelcast

Map Listener

- Listen for map-wide or entry-based events

```
MapClearedListener
MapEvictedListener
EntryAddedListener
EntryEvictedListener
EntryRemovedListener
EntryMergedListener
EntryUpdatedListener
EntryLoadedListener
```

- Can include code to execute after event occurs
- Generates `event`
- Extension of event listener interface

hazelcast

Map Listener Example

```
public class ListeningMember {

   public static void main(String[] args) {
       HazelcastInstance hz = Hazelcast.newHazelcastInstance();
       IMap<String, String> listenToMe = hz.getMap("someMap");
       listenToMe.addEntryListener(new MyEntryListener(), true);
       System.out.println("EntryListener registered");
   }

   private static class MyEntryListener implements EntryAddedListener<String, String>,
           EntryRemovedListener<String, String>, EntryUpdatedListener<String, String> {
       @Override
       public void entryAdded(EntryEvent<String, String> event) {
           System.out.println("entryAdded: " + event);
       }
            // create corresponding objects for removed and updated listeners
   }
}
```

hazelcast

Listener with Predicate

- Listens to modifications performed on entries matching predicate

```
listenToMe.addEntryListener(new MyEntryListener(), new SqlPredicate("name = Smith")
true);
```

hazelcast

Local Listener

- Only listens for events on local partitions
- Can add predicate to entry listener

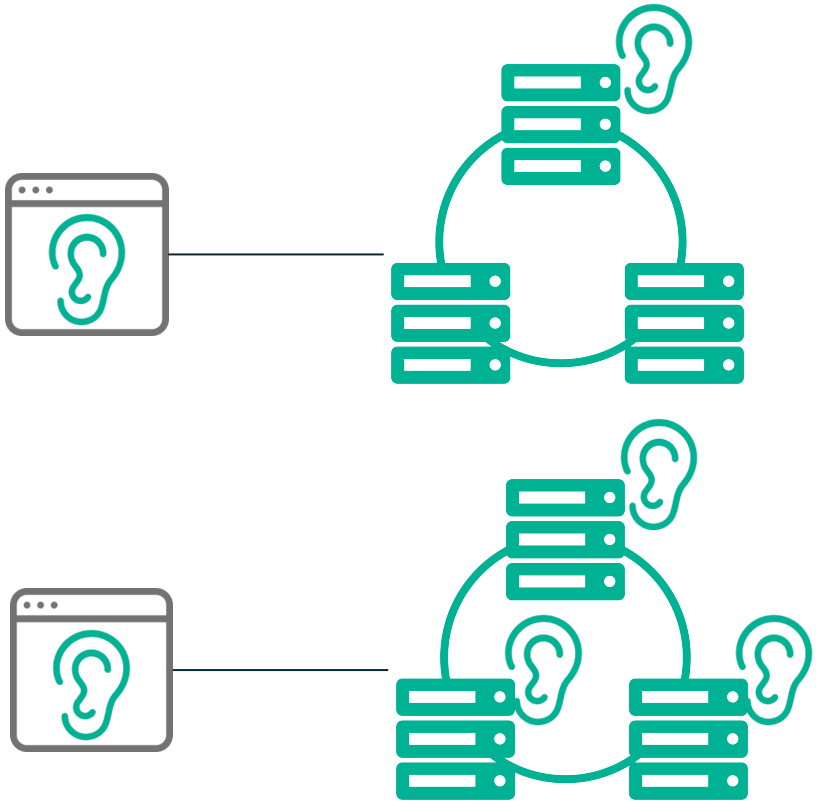
```
String addLocalEntryListener(MapListener listener);

String addLocalEntryListener(EntryListener listener);
```

- Partitions can migrate for load balancing

hazelcast

Deploying Listeners

- Run as code
  - Single member instance
  - Client application
- In map configuration
  - On all members
    - Use local listener to prevent duplicate actions
  - Client application



hazelcast

Adding Listener to Configuration

```
<hazelcast>
   ...
   <map name="somemap">
      ...
      <entry-listeners>
         <entry-listener include-value="false" local="false">
            com.yourpackage.MyEntryListener
         </entry-listener>
      </entry-listeners>
   </map>
   ...
</hazelcast>
```

hazelcast

Listener Result

- When data changes, you get sent a `com.hazelcast.core.EntryEvent` that includes
  - the entry's key
  - the old value (null for create)
  - the new value (null for remove)
  - the event type
  - the map it occurred on
  - the Hazelcast member it occurred on

hazelcast

The Architect's View

- Significant design pattern change – you can now create reactive processing
- Use case: online shop using IMap to store orders
  - Actions on order processing are shipping, confirmation email and restocking the shelf
  - You could scan the IMap for orders needing handling, a daily batch
  - You could react to each order being placed by doing the consequential actions
    - customer gets the confirmation email almost immediately
    - shipping occurs sooner, keep up with your competitors
    - stock is re-ordered as soon as the low stock threshold is reached

hazelcast

Three Additional Points

- The invocation happens in Hazelcast's thread so needs to execute quickly, or spawn another thread
- The event has already happened (perhaps on another JVM) so you can only react
  - Interceptor (on next slide) can be used to stop things from occurring
- Events are delivered via a queue
  - Queue has finite capacity of 1,000,000 events

hazelcast

Interceptors

- Synchronous operation triggered by map event
  - Listeners are asynchronous - can trigger a subsequent action
  - Interceptors literally **intercept** the event
- Interceptors can
  - Alter the behavior of an operation
  - Change the operation value
  - Cancel an operation

Interceptor Options

`interceptGet`	Replace returned map value with new value
`afterGet`	Action to take after get is done
`interceptPut`	Replace value in put with new value
`afterPut`	Action to take after put is done
`interceptRemove`	Collects removed map value
`afterRemove`	Action to take after remove is done

hazelcast

Interceptor Example

```
private static class MyMapInterceptor implements MapInterceptor {

   @Override
   public Object interceptGet(Object value) {
       return value + "-foo"
;
   }
}

...

IMap<String, String> myMap = hz.getMap("myMap");
myMap.addInterceptor(new MyMapInterceptor());
```

hazelcast

Data Change Review

- Before: To stop or adjust an entry change immediately prior

```
com.hazelcast.map.MapInterceptor
```

- During: To change the entry in-situ

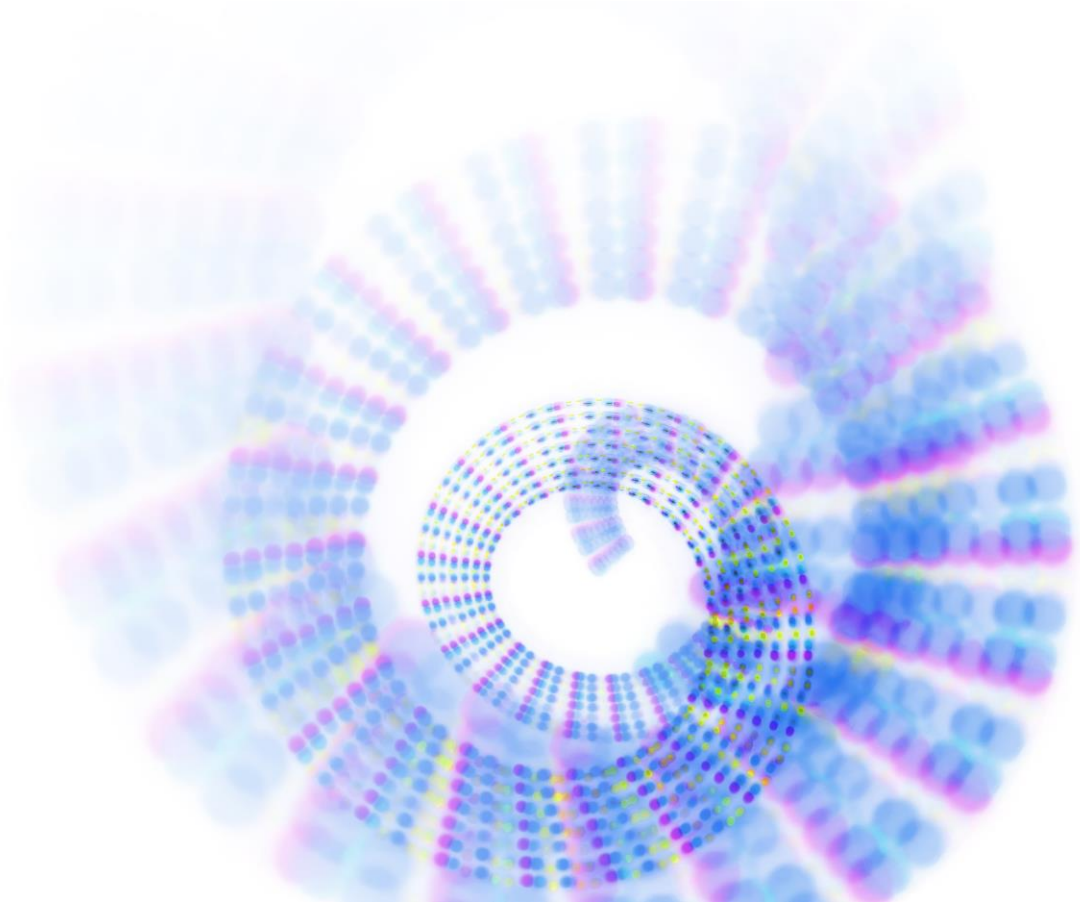
```
com.hazelcast.map.EntryProcessor
```

- After: To react to entry change almost immediately after

```
com.hazelcast.map.listener.EntryAddedListener
com.hazelcast.map.listener.EntryEvictedListener
com.hazelcast.map.listener.EntryExpiredListener
com.hazelcast.map.listener.EntryMergedListener
com.hazelcast.map.listener.EntryRemovedListener
com.hazelcast.map.listener.EntryUpdatedListener
```

hazelcast

Messaging



Hazelcast Messaging

- Hazelcast implementations of basic messaging structures
  - Queue
  - Topic
- Easy to use
- Hopefully familiar concepts
- Distributed implementations (of course)
- Configurable

hazelcast

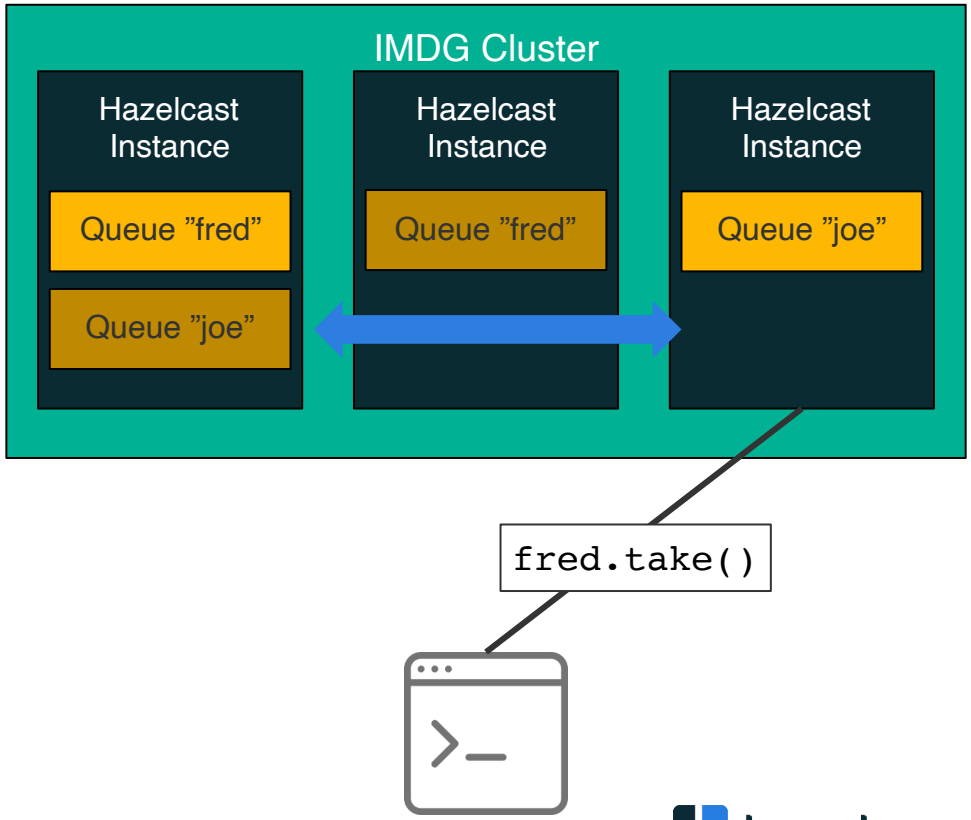
IQueue

- Hazelcast distributed implementation of java.util.concurrent.BlockingQueue
  - Runs on any cluster member
  - FIFO operation
  - Can be bounded to enforce a max size
  - Works with underlying persistent storage for durability
  - Works with listener processes for event based processing
- All the usual queuing functionality
  - Client can add to end, pull from front

hazelcast

IQueue Operations

- Queue cannot be partitioned
  - Entire queue must fit in instance
- Hazelcast maintains primary and backup (if configured)
- Proxy process relays offers/ takes if queue is not local



IQueue is Easy to Use

```
IQueue<Date> theQueue = this.hazelcastInstance.getQueue(”when");
     Date now = new Date();
theQueue.offer(now);
TimeUnit.SECONDS.sleep(5L);
Date then = theQueue.take();
System.out.println("Same ? " + then.equals(now));
```


hazelcast

Bounded IQueue

- Entire queue must fit on one cluster member
  - Default: no limit on queue size
- Configure max-size property to limit queue
  - Size = number of items in queue
  - When limit reached, put operations are blocked
    - IQueue.offer() returns false, item not added, when queue is full
    - IQueue.put() blocks when queue is full

hazelcast

Configuring IQueue

- Queue size
- Number of backups
- Entry max time to live
- Items can expire from the queue

```
<queue name="when">
    <max-size>30</max-size>
    <backup-count>0</backup-count>
    <async-backup-count>1</async-backup-count>
     <empty-queue-ttl>86400</empty-queue-ttl>
</queue>
```

hazelcast

IQueue with Listener

- Similar to MapListener
- Code listener methods using EventListener interface

```
public void itemAdded(ItemEvent itemEvent) {
}

public void itemRemoved(ItemEvent itemEvent) {
}
```

- Apply to queue via configuration

```
<queue name=”when">
  <item-listeners>
    <item-listener include value="true">the.package.MyItemListener</item-listener>
  </item-listeners>
</queue>
```

hazelcast

IQueue with Persistent Store

- Similar to MapStore
- Create class to access external store
  - Implement these methods to ship queue data in/out of store
  - Key (position in queue) is `Long`

```
public void delete(Long arg0)
public void deleteAll(Collection<Long> arg0)
public Object load(Long arg0) {
public Map<Long, Object> loadAll(Collection<Long> arg0)
public Set<Long> loadAllKeys()
public void store(Long arg0, Object arg1)
public void storeAll(Map<Long, Object> arg0)
```

hazelcast

IQueue with Persistent Store (part 2)

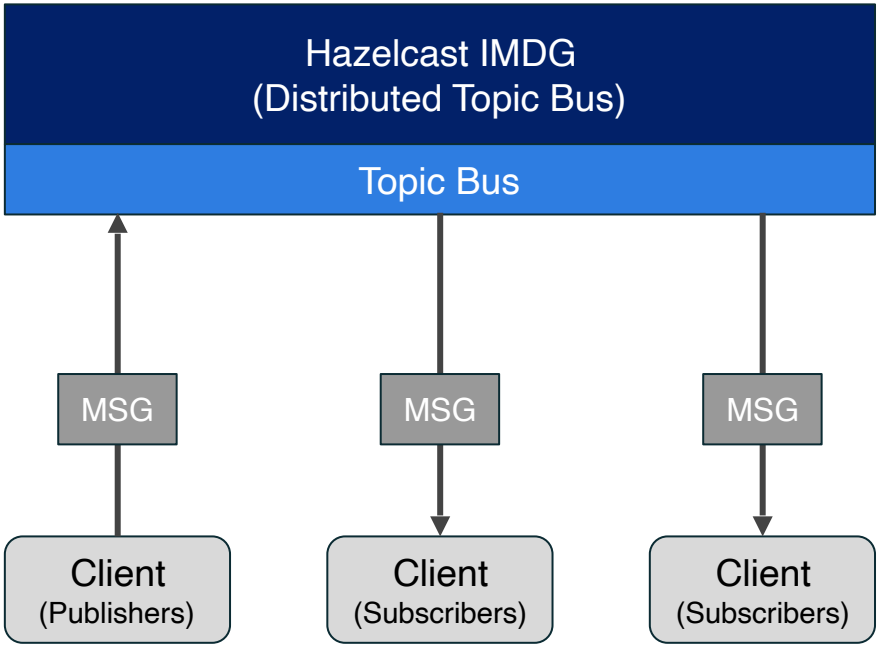
- Apply store to queue via configuration

```
<queue name=”when">
  <queue-store>
    <class-name>the.package.MyQueueStore</class-name>
    <properties>
      <property name="binary">false</property>
      <property name="memory-limit">3</property>
      <property name="bulk-load">5</property>
    </properties>
  </queue-store>
</queue>
```

hazelcast

Topic/Reliable Topic

- Pub/Sub messaging model
  - Publisher submits message
  - Cluster relays to subscribers
- Cluster-wide – any member can publish
- Reliable Topic backed with Ringbuffer
  - Provides backup
  - Each topic has own buffer
  - No competition for event handler



hazelcast

ITopic

```
ITopic<String> englishTopic = hazelcastInstance.getTopic("english");
ITopic<String> frenchTopic = hazelcastInstance.getReliableTopic("french");
englishTopic.publish("hello");
frenchTopic.publish("bonjour");
```

- Simple to use
- Payload does not have to be a string
  - Can be any serializable class
- Two implementations – Topic and ReliableTopic

hazelcast

Subscribing to a Topic

- Add a callback

```
englishTopic.addMessageListener(new MyMessageListener());
frenchTopic.addMessageListener(new MyMessageListener());

class MyMessageListener implements MessageListener<String> {

        @Override
        public void onMessage(Message<String> message) {
                System.out.println(message.getSource() + "," +
                        message.getMessageObject());
        }
}
```

- Output:

```
english,hello
french,bonjour
```


hazelcast

Using ITopic

- Any process can send
- Any process can receive if it registers a listener
  - Servers
  - Clients
  - Some, none or all processes – even the sending process if wanted
- The listener is invoked for messages published *after* the listener is registered

hazelcast

Reliable and “Unreliable”

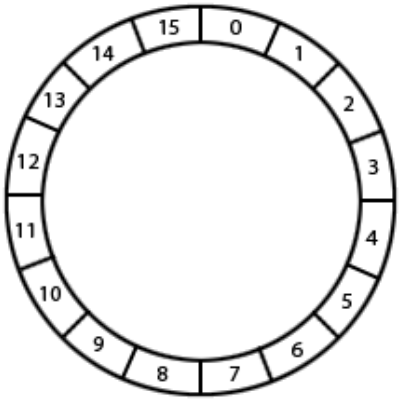
```
ITopic<String> englishTopic = hazelcastInstance.getTopic("english");
ITopic<String> frenchTopic = hazelcastInstance.getReliableTopic("french");
```

- The `getTopic` style has no backups
  - Faster to update
  - If the hosting member crashes, messages in flight may be lost
- The `getReliableTopic` style has backups
  - Backed by a Ringbuffer

hazelcast

Ringbuffer

- Distributed (not partitioned) data structure
- Stores the items in a single, fixed-size buffer (array)
- Items are identified by a sequence
  - Tail sequence - newest items
  - Head sequence - oldest items
  - StaleSequenceException
- Has a read position and a write position



hazelcast

Ringbuffer Configuration

```
<ringbuffer name="wagner">
     <capacity>5</capacity>
     <backup-count>1</backup-count>
     <async-backup-count>0</async-backup-count>
     <time-to-live-seconds>0</time-to-live-seconds>
     <in-memory-format>BINARY</in-memory-format>
</ringbuffer>
```

- Backup count is what makes ITopic reliable

hazelcast

Writing to the Ringbuffer

```
Ringbuffer<String> germanRingbuffer =hazelcastInstance.getRingbuffer("wagner");
          germanRingbuffer.add("montag");
          germanRingbuffer.add("diestag");
          germanRingbuffer.add("mittwoch");
          germanRingbuffer.add("donnerstag");
          germanRingbuffer.add("freitag");
          germanRingbuffer.add("samstag");
          germanRingbuffer.add("sonntag");
```

hazelcast

Reading from the Ringbuffer

```
long head = germanRingbuffer.headSequence();
          System.out.println(germanRingbuffer.readOne(head));

          long tail = germanRingbuffer.tailSequence();
          System.out.println(germanRingbuffer.readOne(tail));
```

- Do not have to read from end
- Read does not remove an item
  - Can re-read

hazelcast

Capacity…

- Capacity set to 5

```
<capacity>5</capacity
```

- So, at write…

- When we read head and tail

```
Output:
mittwoch
sonntag
```


hazelcast

Hazelcast Threading

I/O Threading

- 3 types of NIO:
  - Accept Requests - 1 thread
  - Read data from other members/clients - # configured
  - Write data to other members/clients - # configured
- hazelcast.io.thread.count
  - Default is 3
  - If set to 3 then 7 I/O threads in total - 1 Accept, 3 Read and 3 Write
- Each I/O thread has its own Selector instance and waits on the `Selector.select` for next I/O operation

hazelcast

I/O Threading

- Automatic rebalancing of IO thread
  - Periodic scan of thread utilization, if found over-used then connection is shifted to another thread
  - `hazelcast.io.balancer.interval.seconds` - scanning interval. Default is 20

hazelcast

Event Threading

- Shared Event system for events driven components: Topic, Listeners, Near Cache
- Every member has array of event threads `EventThread[]`
- Every Thread in the array has its own work queue
  - `ET[n] = Qn`
- On an event:
  - An event thread from the array is selected, based on message ordering and thread load
  - Event is placed on the work queue of this thread

hazelcast

Event Threading

- `hazelcast.event.thread.count`: Number of event-threads in this array. Its default value is 5
- `hazelcast.event.queue.capacity`: Capacity of the work queue. Its default value is 1000000
- `hazelcast.event.queue.timeout.millis`: Timeout for placing an item on the work queue. Its default value is 250

Event Threading

- Increase `hazelcast.event.thread.count` if you have many cores
- Multiple components share event threads and queues: recommended to offload processing of events to a dedicated thread pool
  - Topics A and B publish messages
  - Some messages share the queue and hence the same event threads
  - If event thread takes lot of time in processing A's messages, B will suffer because of thread starvation

hazelcast

Event Threading

- Problems: Event system is a “best effort” system i.e. no guarantee that you will get an event
- If the events are produced at a higher rate than they are consumed, the queue grows in size
- When Queue’s maximum capacity is reached, the items are dropped
  - Topic A has lot of pending messages
  - Queue has no capacity and messages for B are dropped
  - B does not receive messages

hazelcast

Operations Threads

- 2 types of operations
  - Partition Aware operation e.g. `IMap.get(key)`
  - Non Partition Aware operation e.g. `IExecutorService`

hazelcast

Partition-Aware (Partition) Threads

- Operation Thread <> Queues
  - Default thread count = 2 x number of cores, min 2
- `hazelcast.operation.thread.count`
- Operation thread consumes messages from its own work queue. To find the thread to process operation:

```
threadIndex = partitionId % partition thread-count
```

- After `threadIndex` is determined, operation is put in the work queue of that operation thread

hazelcast

Partition-Aware Threads

- A single operation thread executes operations for multiple partitions

  271 partitions / 10 partition threads = 1 thread per 27 partitions

- Each partition belongs to only 1 operation thread
  - All operations for a partition are always handled by the same operation thread
  - Concurrency control not needed

hazelcast

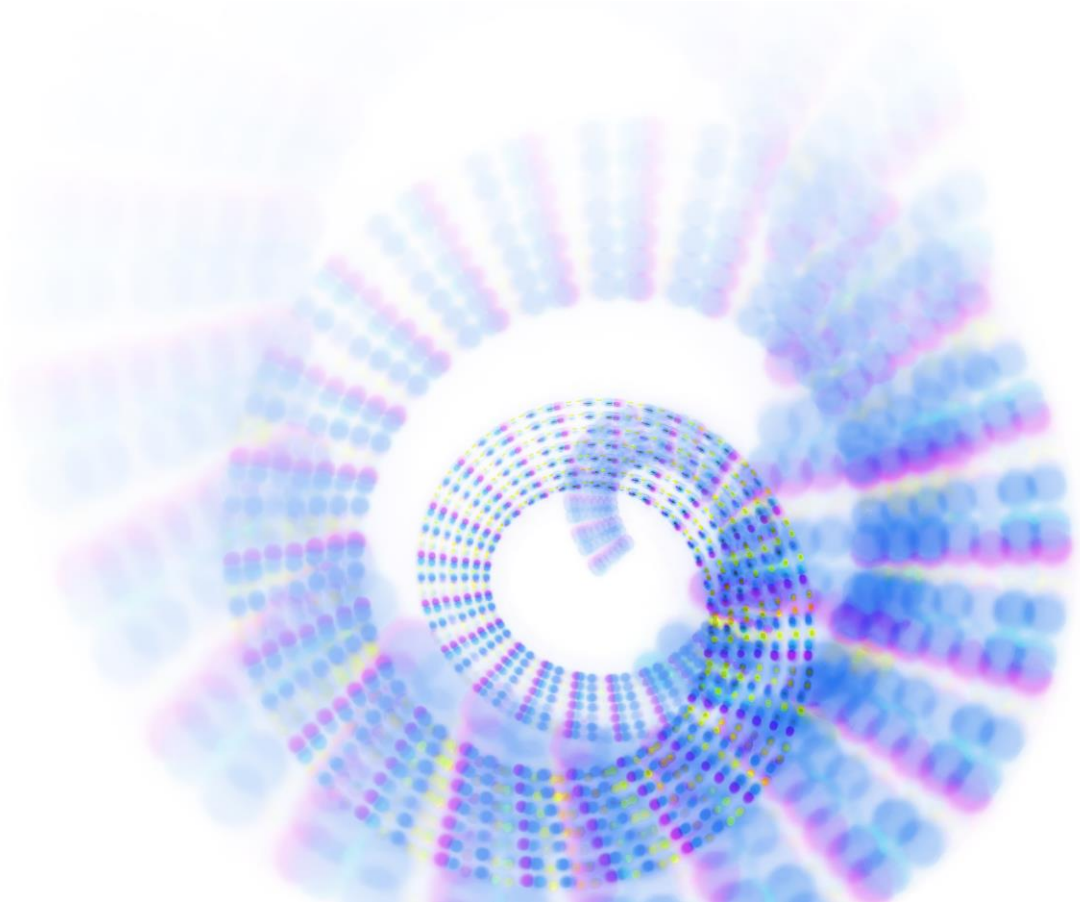
Non-Partition-Aware (Generic) Threads

- Default thread count = (number of cores) / 2, min 2
- `hazelcast.operation.generic.thread.count`
  - All generic operation threads share a single work queue: `genericWorkQueue`
  - Automatic work balancing: Operation in work queue can be handled by any generic operation thread
  - Shared queue can be a point of contention

hazelcast

Q and A

Next Up … Comprehensive Lab



Lunch Break

60 Minute Break

hazelcast

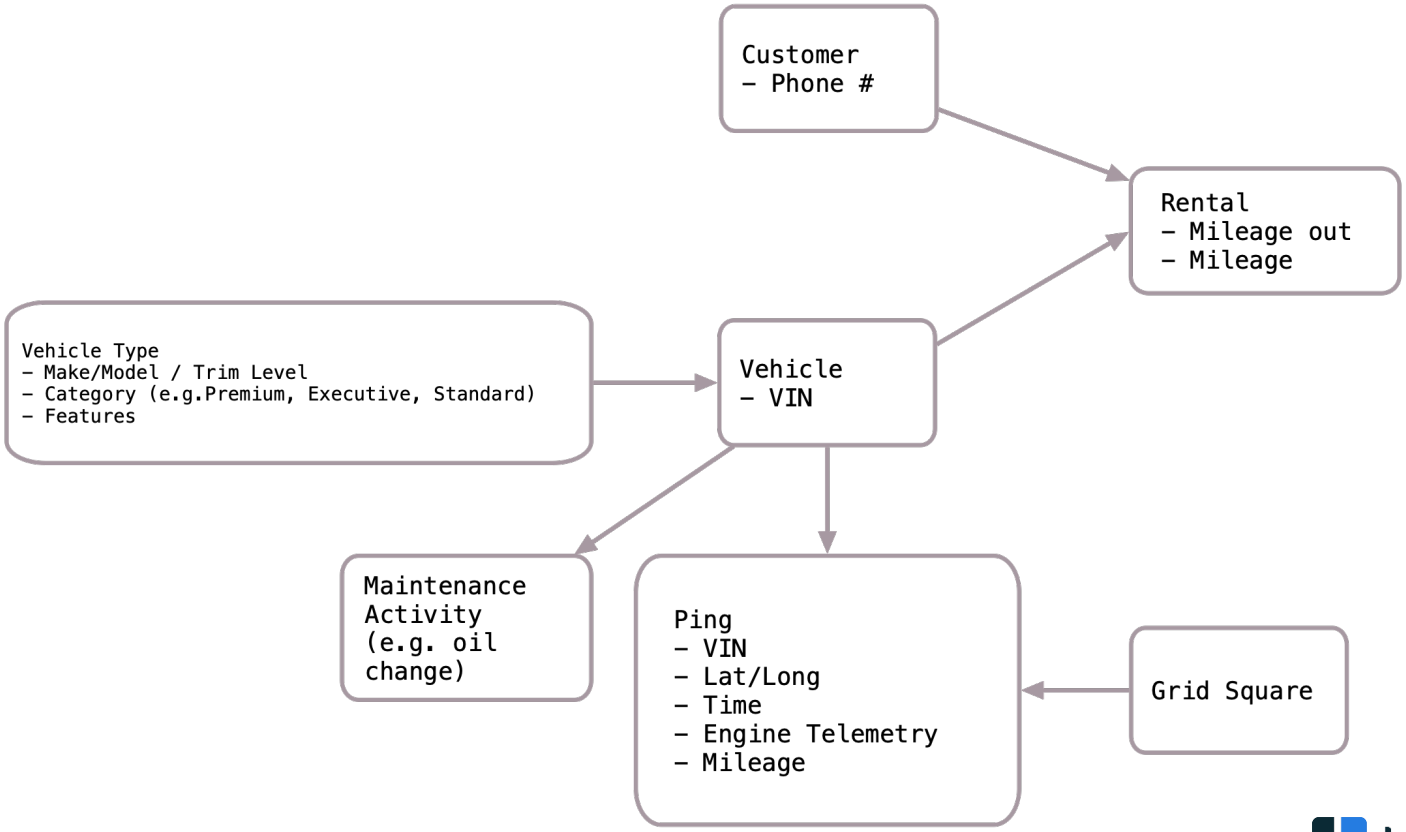

hazelcast
Comprehensive Streaming Lab

The Scenario

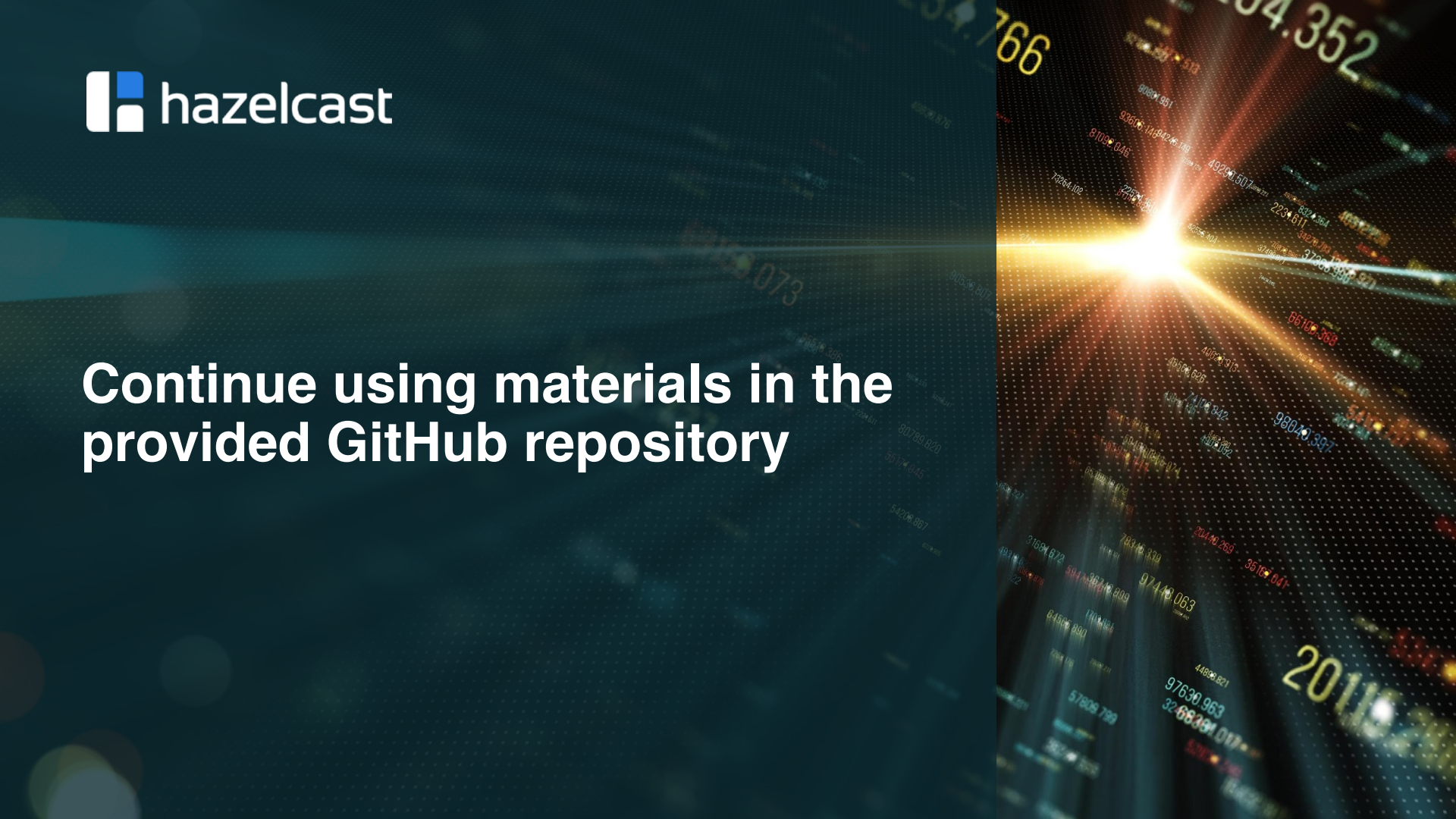
In this lab, we will be developing solutions for a truck rental company, UMove. All trucks in the UMove fleet are equipped with GPS and engine monitoring devices which periodically transmit location and other information over cell networks. We will be designing solutions that help UMove turn the continuous stream of data produced by their fleet into meaningful insights, and to act on those insights in near real time.

hazelcast

Data Model



hazelcast


hazelcast
Continue using materials in the provided GitHub repository