hazelcast

In-Memory Data Grid (IMDG)

Ludovít Labaj, Mojmír Odehnal

Hazelcast IMDG

- It is system for keeping data in distributed in-memory data structures like **key-value map, queue, set**, etc. for quick access, parallel processing, scalability, resilience, …
- It also provides distribution mechanism for publishing messages that are delivered to multiple subscribers.
- The system is written Java as one JAR file with no dependencies.
    - You can embed the member JAR (both server & client) into your Java application making it another member of the cluster.
    - Or you can run the member JAR separately and in your application, you would embed only a client, which is available for multiple platforms and languages:

Microsoft® .NET · Java · C++ · python™ · node JS® · Scala

        - (it is smart client - e.g. it knows on which node the data is)
    - You can also communicate with the cluster through any of its members via HTTP RESTful API.

Some base characteristics Hazelcast IMDG:

- The data is always stored in-memory (RAM) of the servers
- Multiple copies are stored in multiple machines for automatic data recovery in case of single or multiple server failures
- Servers can be dynamically added or removed to increase the amount of CPU and RAM
- The data can be persisted from Hazelcast to a relational or NoSQL database

Distributed structures and synchronization tools

- Distributed data structures: *(many of them are based on structures in java.util.* package of Java)*
  - **Map** - distributed key-value storage
    - Map Listener - any member or client can register itself to get notified about changes on certain map
  - **MultiMap** - a map, that allows adding multiple values for one key
  - **Queues** - allows some nodes to supply items and other nodes to consume them
  - **Set, List, AtomicLong, AtomicReference, Ringbuffer, ReplicatedMap**
  - **Topic and ReliableTopic** - allows nodes to subscribe for receive certain messages. Any node can then publish message of that topic, which will be asynchronously delivered to subscribers.
- Tools:
  - **Lock, Semaphores**
  - **CountDownLatch**
  - **IdGenerator**
  - ...

Use cases

- Caching
  - An elastic/scalable “alternative” to **Memcached** or as a plug-in enhancement.
  - **JCache** caching layer API integration
  - **Hibernate Second Level Cache** and **Spring Cache** compatibility
  - **Cache-as-a-Service** layer for multiple applications in your company
- Messaging
  - **Distributed Queue (IQueue)** - it is built on top of java.util.concurrent.BlockingQueue
  - Hazelcast **Topics** - application can subscribe in a cluster for messages of some topic
- Distributed Computing
  - **Entry Processor, Executor Service, Fast Aggregations, MapReduce, Query, …**
- Backbone of a Microservices architecture
- Web Session Clustering - e.g. **No lost sessions** - if you have couple of load balanced web servers and one fails, sessions would backed in IMDG embedded in the web servers, …

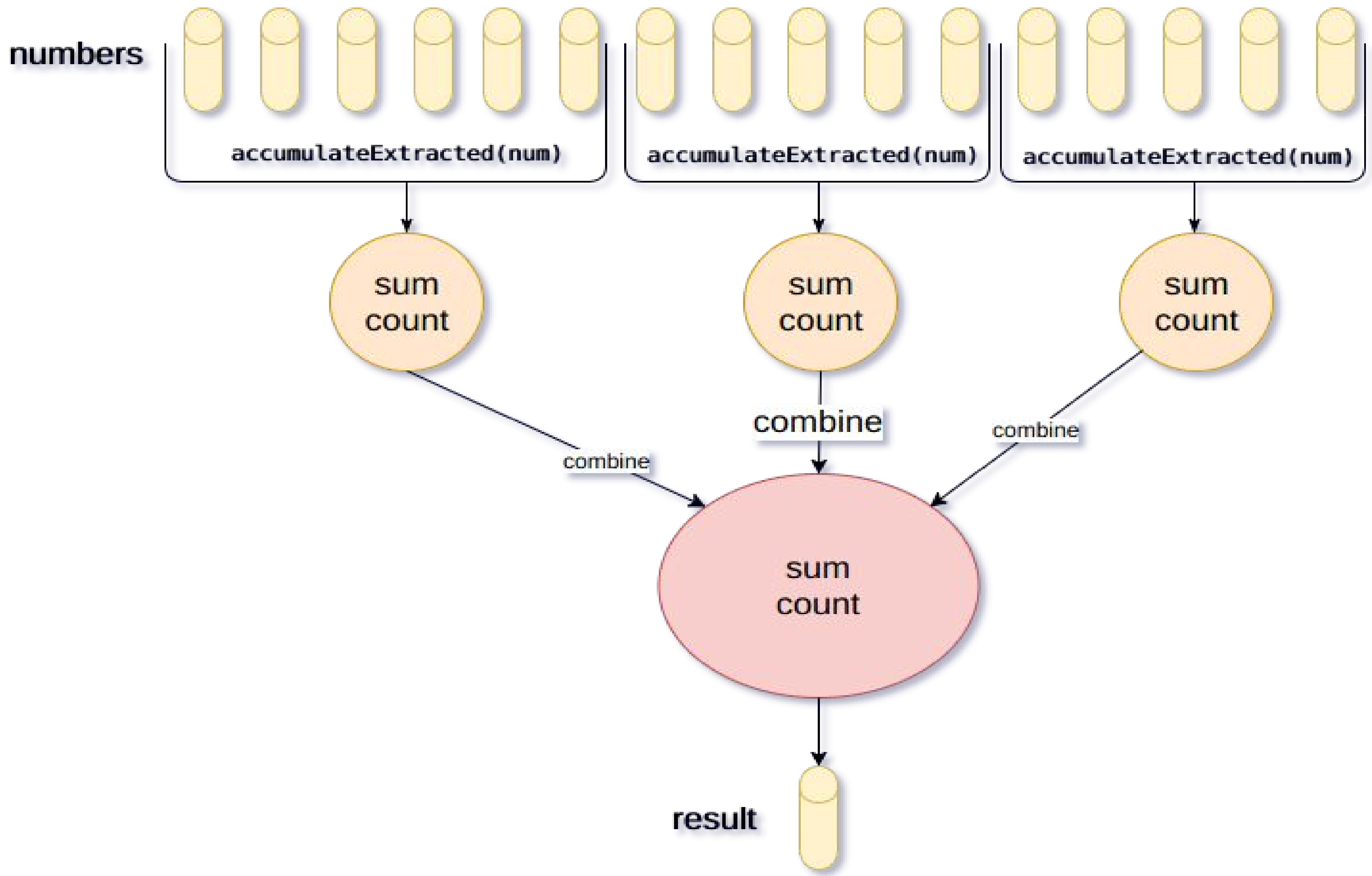
Distributed processing and computing tools

- **Fast Aggregations**
  - Aggregates some entries in three phases: Accumulation, Combination and Aggregation
- **MapReduce**
- **Query**
  - You can search for entries with some predicate. The predicate can be defined with API similar to JPQL Criteria API or it can be string using syntax of SQL Where clause.
  - Predicate is sent to all members, they find matching entries, send them to requester, who merges them.
  - You can add indexes to the map to speed up queries based on some attributes.
- **Listener with Predicate**
  - Allows node to subscribe for notification of changes of specific type of entries

Distributed processing and computing tools

- **Entry Processor**
  - Enables fast in-memory operations on a distributed map without having to worry about locks or concurrency issues. It supports choosing target entries using predicates. You do not need any explicit lock on entry: it locks the entry, runs the EntryProcessor, then unlocks the entry.
  - It is sent to each member and they it to map entries. More members => Faster execution
- **Executor Service**
  - With it, you can execute tasks asynchronously. If your task execution takes longer than expected, you can cancel the task execution. Tasks are implemented as java.util.Runnable and java.util.concurrent.Callable. If you need to return a value, use Callable. Otherwise, use Runnable. Tasks should be Serializable since they will be distributed.
- **Scheduled Executor**
  - Allows to schedule tasks at a given moment in time, or repetitive scheduling at fixed intervals in your cluster.

Demo


numbers
accumulateExtracted(num)
accumulateExtracted(num)
accumulateExtracted(num)
sum
count
sum
count
sum
count
combine
combine
combine
sum
count
result

Thank you

for your attention