hazelcast

Stream Processing Essentials

Presented by: Vladimír Schreiner <vladimir@hazelcast.com>
Nicolas Frankel <Nicolas@hazelcast.com>

Introduction

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 as a Maven project

3/ Build it to download dependencies!

hazelcast

Course Objectives

- When you leave this class, you will be able to
  - Define stream processing and explain when to use it
  - List the building blocks of a streaming application
  - Transform, match, and aggregate streaming data
  - Perform continuous processing of time-series data
  - Scale, deploy, and operate streaming apps

hazelcast

Agenda

Introductions	15 minutes
Stream Processing Overview	15 minutes
The Building Blocks	30 minutes
Break	5 minutes
Transforming a Data Stream (lab)	25 minutes
Enrichment (lab)	30 minutes
Break	15 minutes
Aggregations and Stateful Streaming (3 labs)	50 minutes
Break	5 minutes
Scaling and Operations (lab – no coding)	35 minutes
Q&A and Conclusion	15 minutes

hazelcast

Warning - Lambdas Ahead!

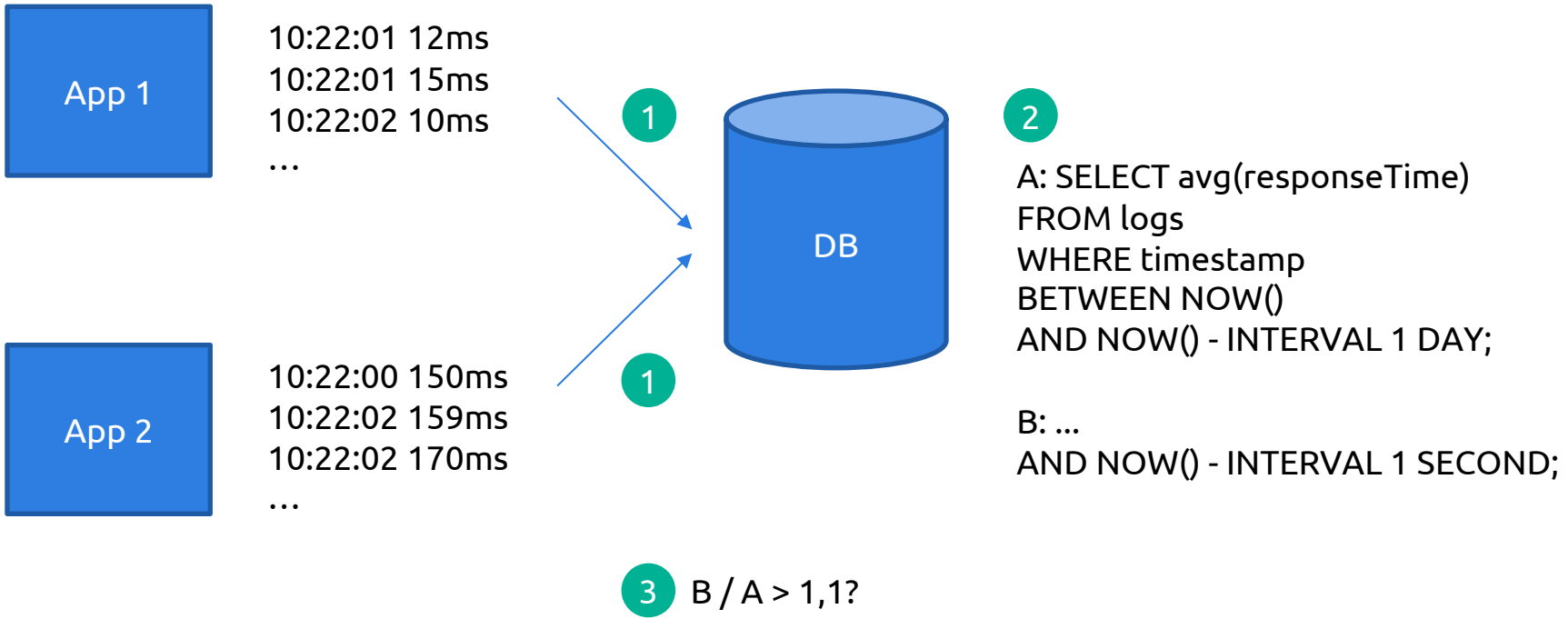
```
int sum = streamCache().collect(
        DistributedCollector.of(
                () -> new Integer[]{0},
                (r, v) -> r[0] += v.getValue(),
                (l, r) -> {
                    l[0] += r[0];
                    return l;
                },
                a -> a[0]
        )
);
```

This is the kind of code we'll be looking at. If you don't understand it, you won't get much out of this class.

hazelcast

Stream Processing Overview

System Health Monitoring



hazelcast

System Health Monitoring - Challenges

- Scale
  - 1k records per sec -> 86 mio per day
  - Increases latency
- Timing
  - How to coordinate data loading and querying?
  - When to submit the query?
    - Network transfer times may vary
    - Timestamps from various sources -> unordered data
  - Balancing correctness and latency

hazelcast

Batch Processing

Linear process
No overlap = huge latency!

Use-cases

Post-mortem analysis

ML training/data science

Offline transaction processing

Extract Transform Load (ETL)

Tools

Hadoop

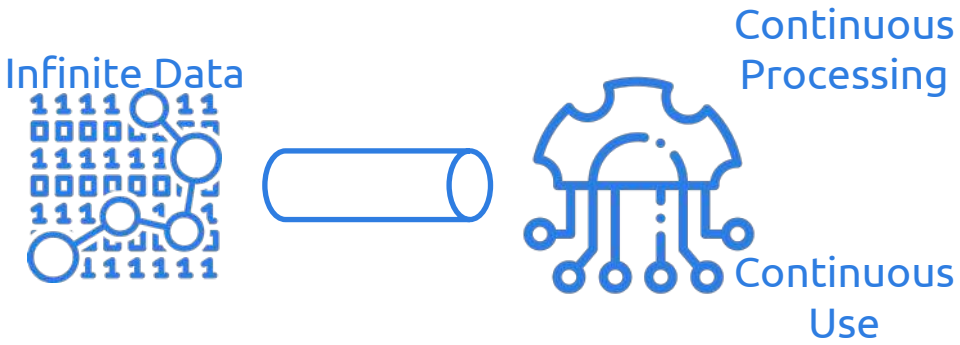
Hive for SQL people

Often custom

hazelcast

Stream Processing

Real world data doesn't come in batches!



Characteristics

Querying made pro-active

Push instead of pull

Pre-processing for database

Benefits

Low latency

Continuous programming model

Deals with time - event-time, watermakrs, late data

hazelcast

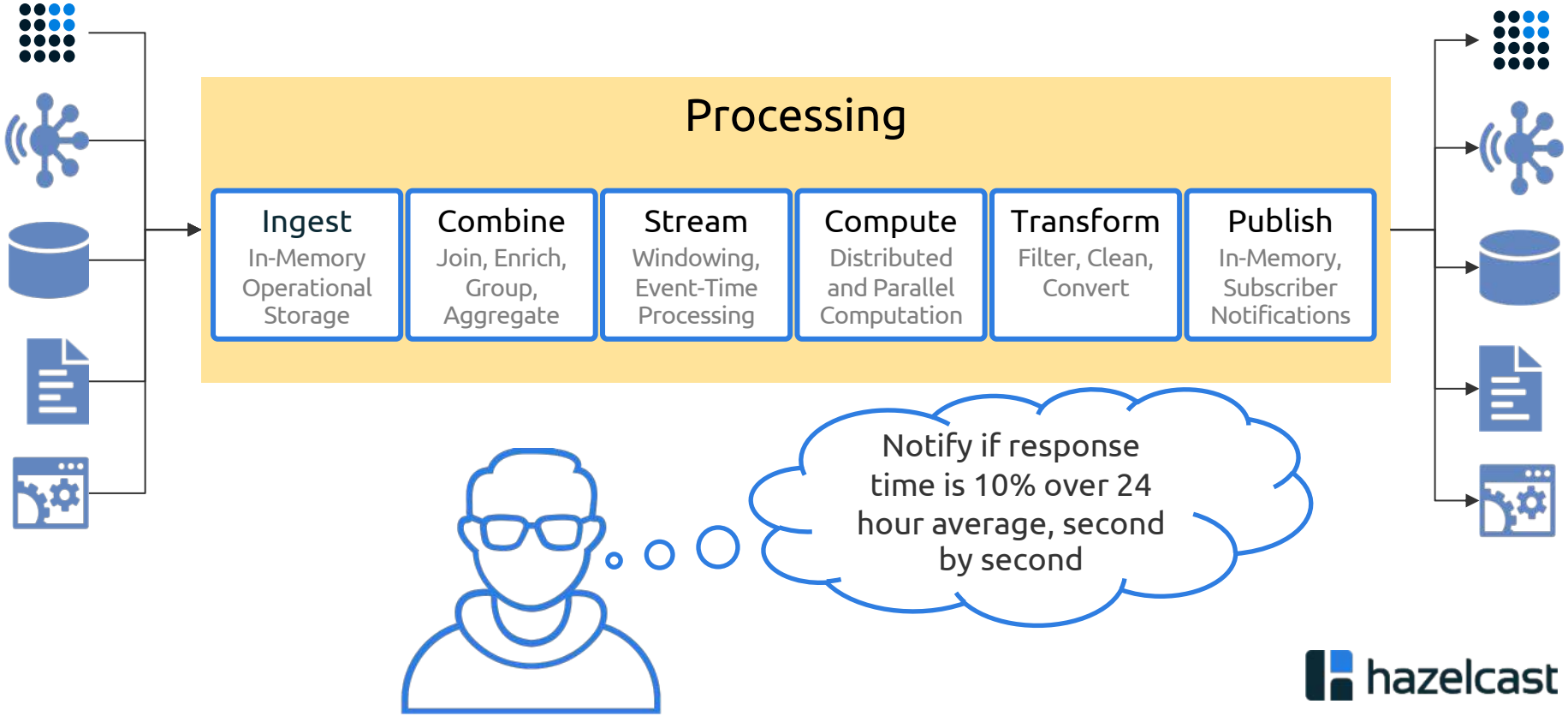
Stream Processing and Databases

- Database allows clients to **pull** data by querying it's state
- Stream processor runs a continuous query and **pushes** updates to consumers

- Continuous programming model maps better to reality
- Things happen continuously in real world, not in batches.

hazelcast

Streaming is “Smart ETL”



hazelcast

What Stream Processing Brings

… to traditional ETL (data pumps)	Scale
… to batch analytics (MapReduce)	Continuous programming, reduces latency
… to JMS topic with Java worker	Fault-tolerance, higher level programming. model
… to CEP	Scale
… to DB triggers	Time as first-class citizen

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

hazelcast

Use Case: Analytics and Decision Making

- Real-time dashboards
- Stats (gaming, infrastructure monitoring)
- Decision making
- Recommendations
- Prediction - often based on algorithmic prediction (push stream through ML model)
- Complex Event Processing

Use Case: Event-Driven Applications

- Event Sourcing
  - Sequence of change events as a shared database.
    - Simpler than replicating every database to every service
  - Apps publish and subscribe to the shared event log.
  - App state is a cache of the event store
- Stream processor is the event handler
- Consumes events from the event store
- Updates the application state

hazelcast

DEMO: How do people feel about crypto?

https://github.com/hazelcast/hazelcast-jet-demos

1. Get Tweets
2. Filter out irrelevant ones
3. Predict the sentiment
4. Compute avg per cryptocurrency in last 30s / 5m

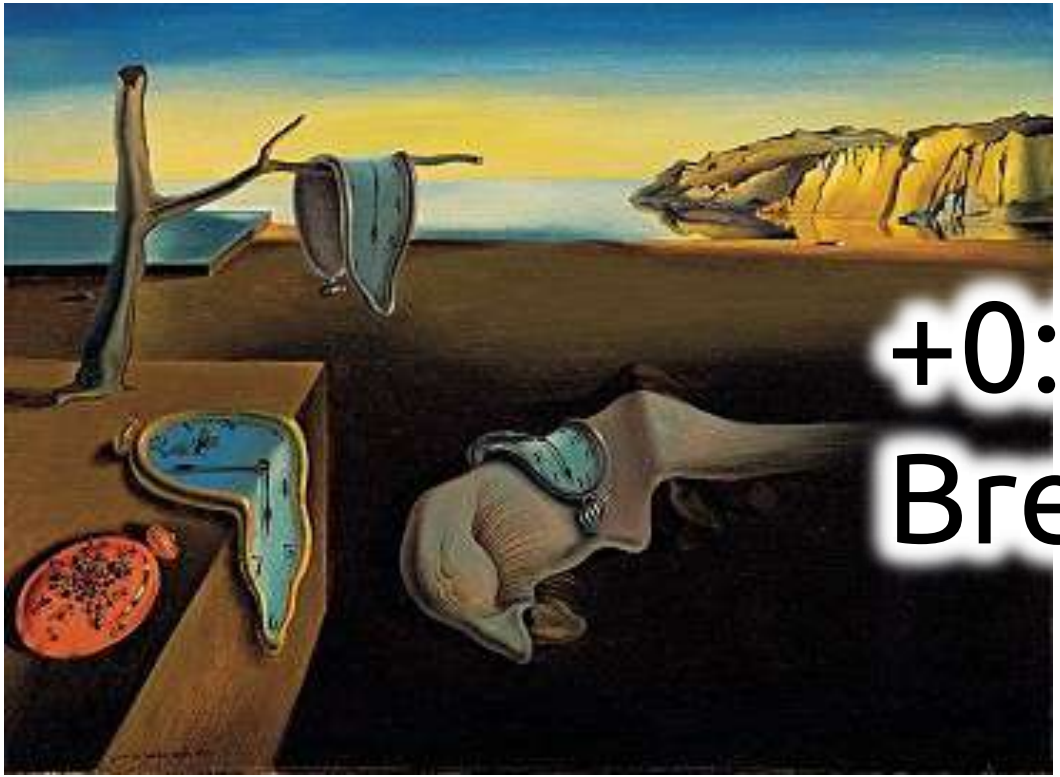
hazelcast

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
  - Driven by data – no external coordination
  - Stays big-data ready (design decision for most streaming tools)

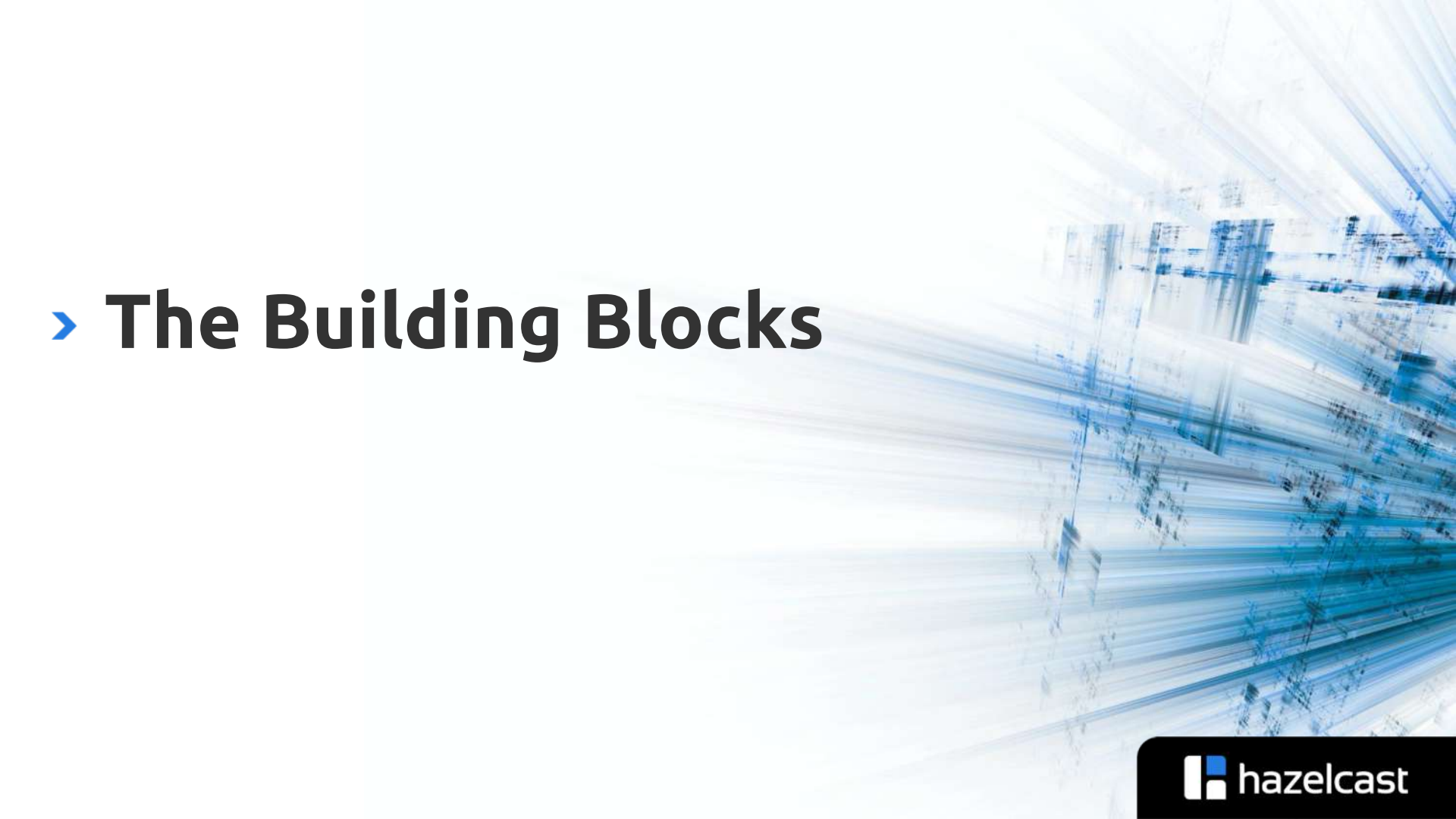
hazelcast

Time Check

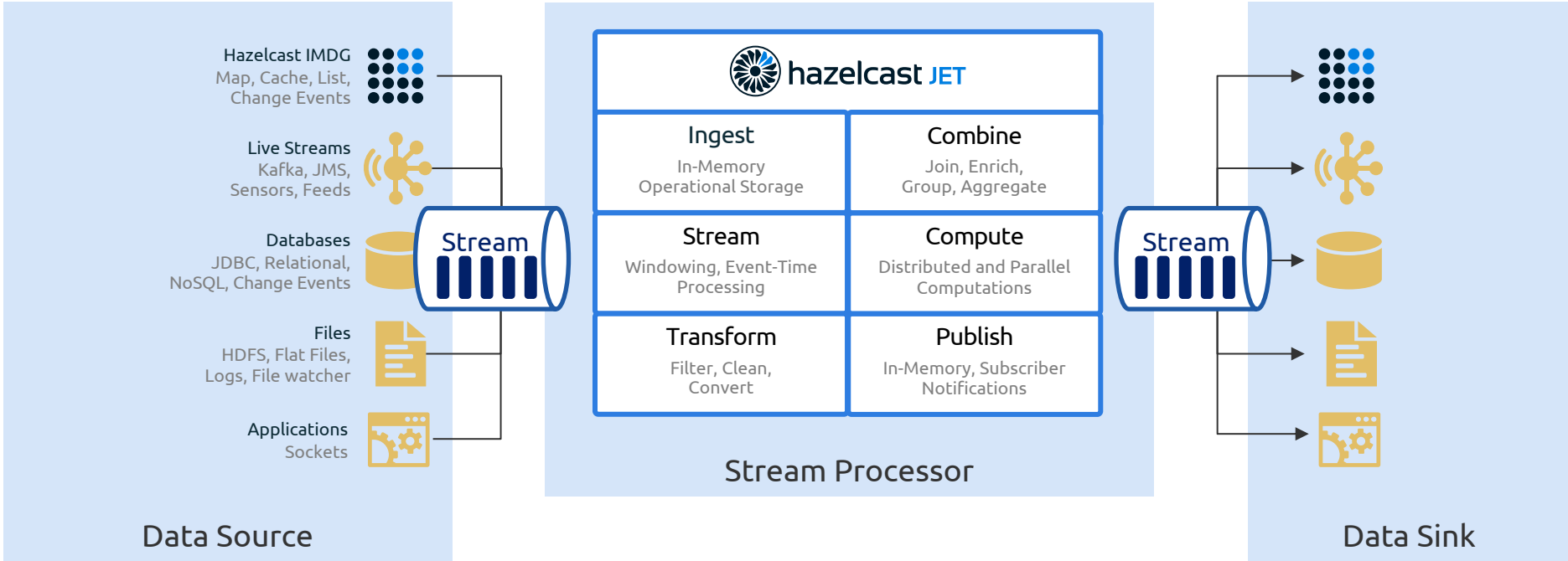


hazelcast

The Building Blocks

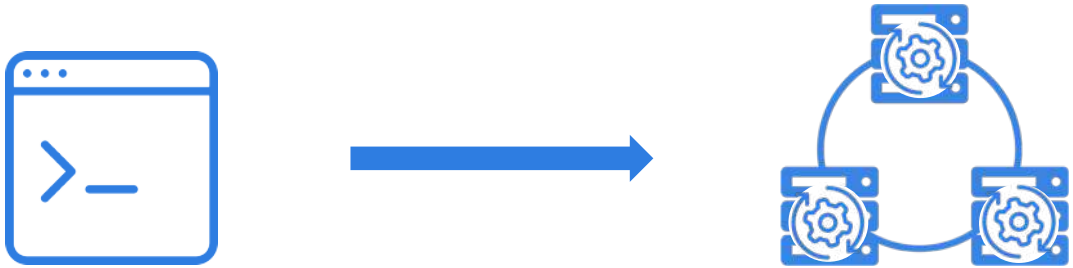


The Big Picture



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)

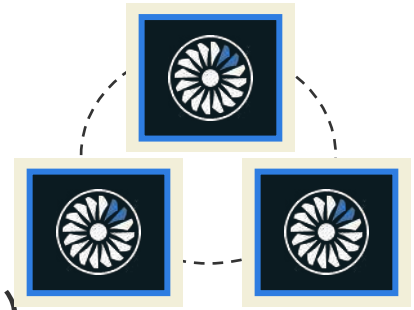
```
Map<String, Long> counts = lines.stream()
    .map(String::toLowerCase)
    .flatMap(line -> Arrays.stream(line.split("\\W+")))
    .filter(word -> !word.isEmpty())
    .collect(Collectors.groupingBy(word -> word, Collectors.counting()));
```

Why Pipelines Use Declarative

- ”What” vs. “How”
- SPE handles the “how”
  - Data routing
  - Partitioning
  - Invoking pipeline stages
  - Running your pipeline in parallel

hazelcast

The SPE: Hazelcast Jet



- Distributed data processing engine
- Supports bounded (batch) and unbounded (stream) data sources
- Java API to define the processing (Pipeline API)
- Built on Hazelcast IMDG
- Single embeddable JAR
- Open-source, Cloud-native (managed srv. for IMDG parts)
- Java SDK
  - JDK 8 minimum

hazelcast

Hazelcast IMDG

- IMDG = “In-Memory Data Grid”, distributed in-memory data structures with computational capabilities
  - Map, List, Queue
  - Querying, Entry Processor
  - Executor Service
  - Lock, Semaphore, AtomicLong, Unique ID generator, HyperLogLog..
- Used as cache, operational database and for coordination
- Clients for Java, Scala, C++, C#/.NET, Python, Node.js, Go
- Good foundation for distributed computing

hazelcast

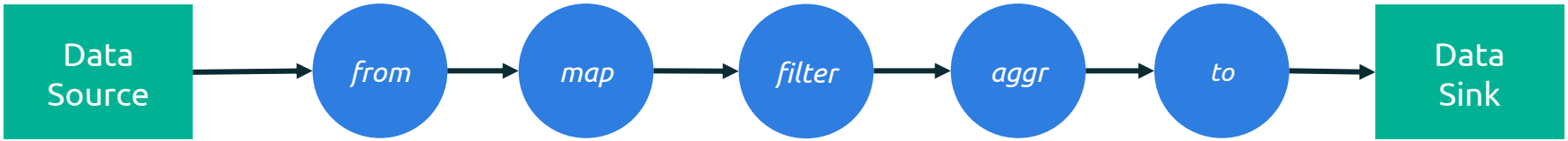
What Distributed Means to Hazelcast

- Multiple nodes (cluster)
- Scalable storage and performance
- Elasticity (can expand during operation)
- Data is stored partitioned and replicated
- No single point of failure

Jet Does Distributed Parallel Processing

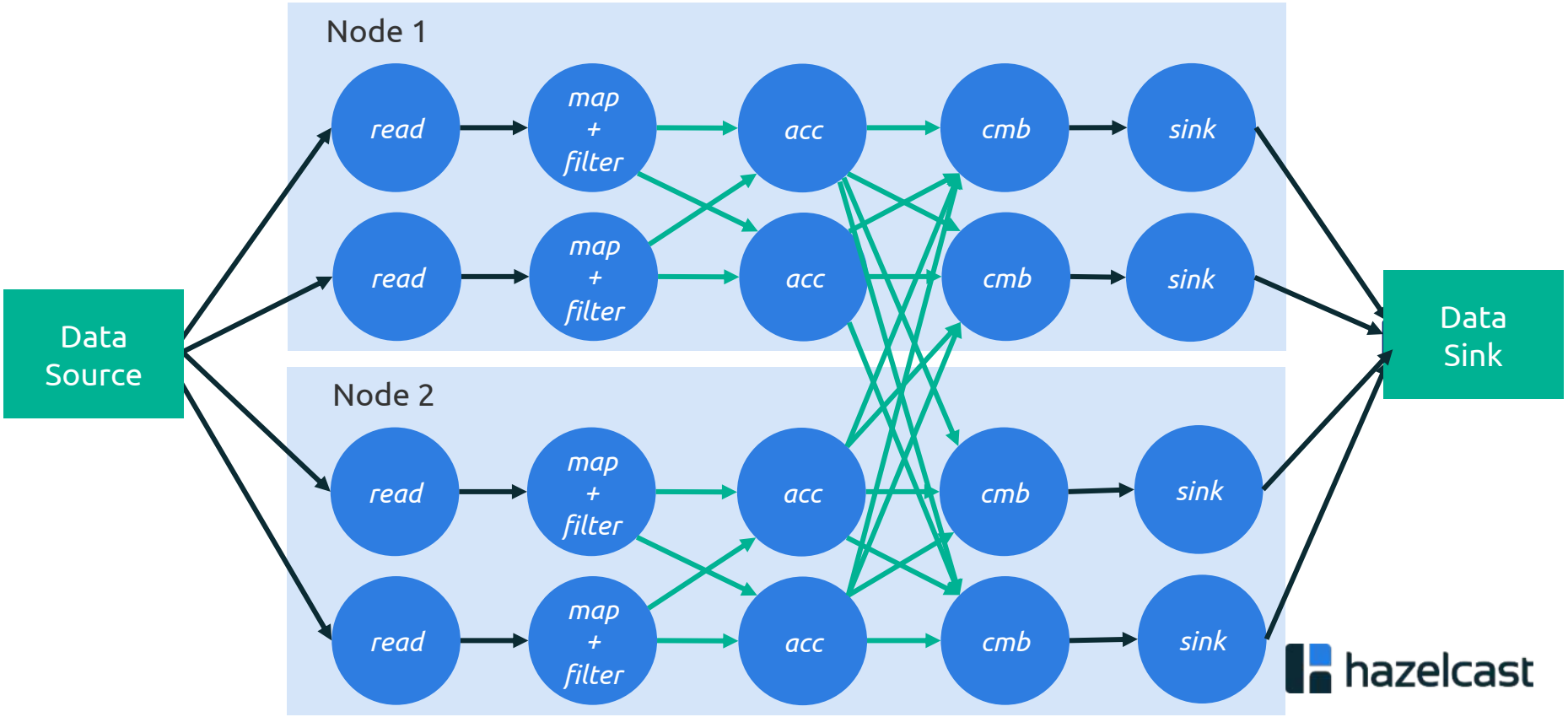
- Jet translates declarative code to a DAG (Task Parallelism)

```
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));
```

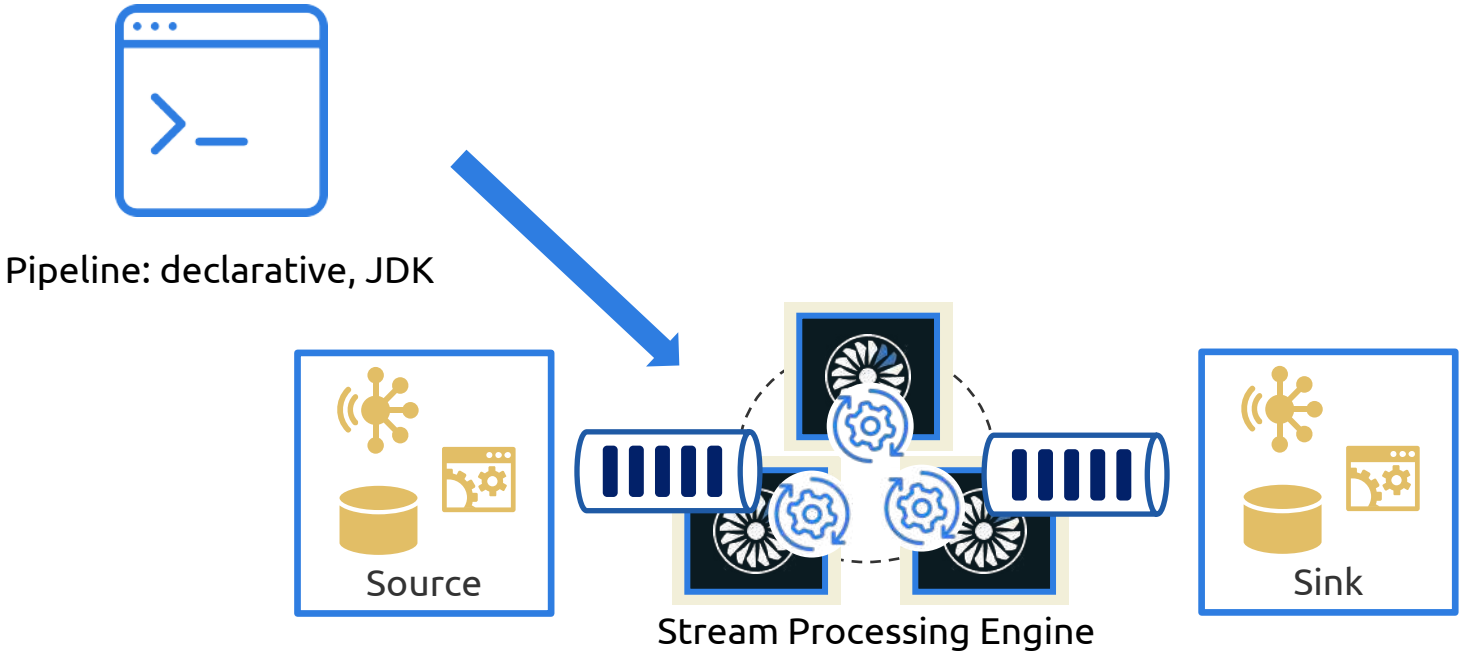


hazelcast

Jet Does Distributed Parallel Processing

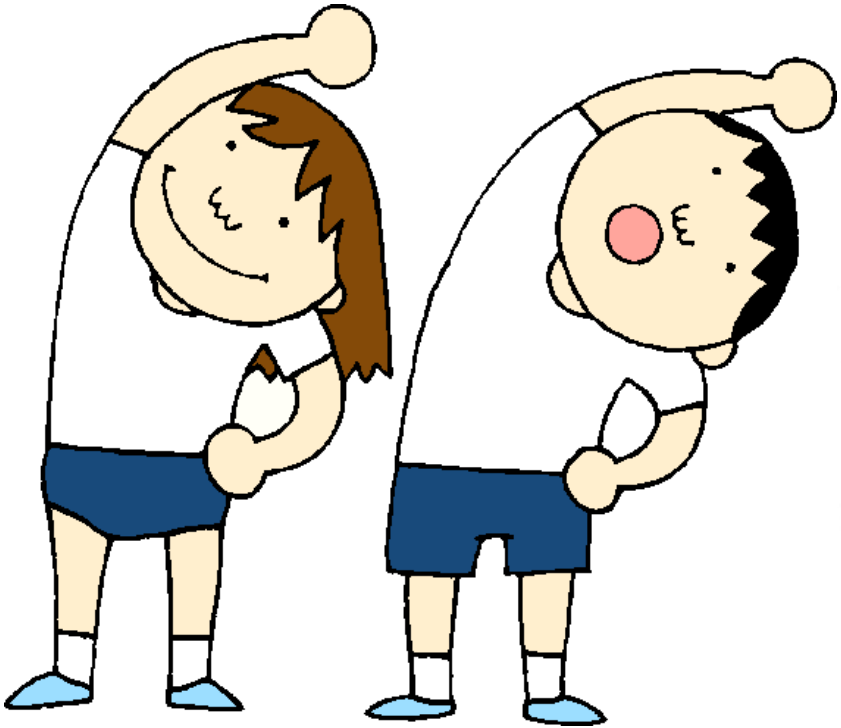


All the Building Blocks



hazelcast

Stretch Break



+1:00

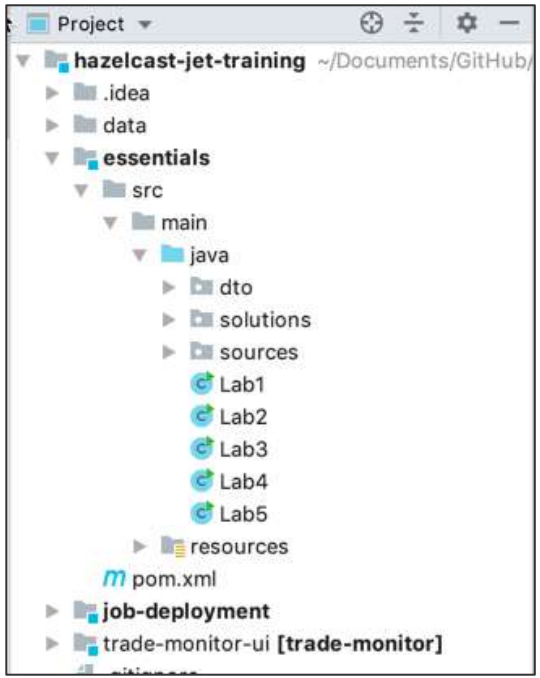
5 Minute Break

Please do not leave the room

hazelcast

Transforming a Data Stream

Checking Your Lab Setup



- Download the lab source github.com/hazelcast/hazelcast-jet-training
- Import it to your IDE as a Maven project
- Build it to download dependencies!
- Open the essentials module

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
```

- Data generators for quick start and testing

```
com.hazelcast.jet.pipeline.test.TestSources
```

hazelcast

We'll Start Simple

- Turn off event-time processing for now

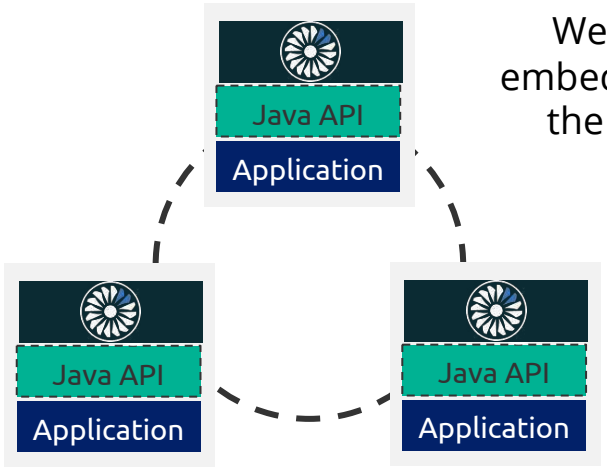
```
p.drawFrom( Source definition )
        .withoutTimestamps()
```

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

hazelcast

Interacting with the Cluster

Embedded

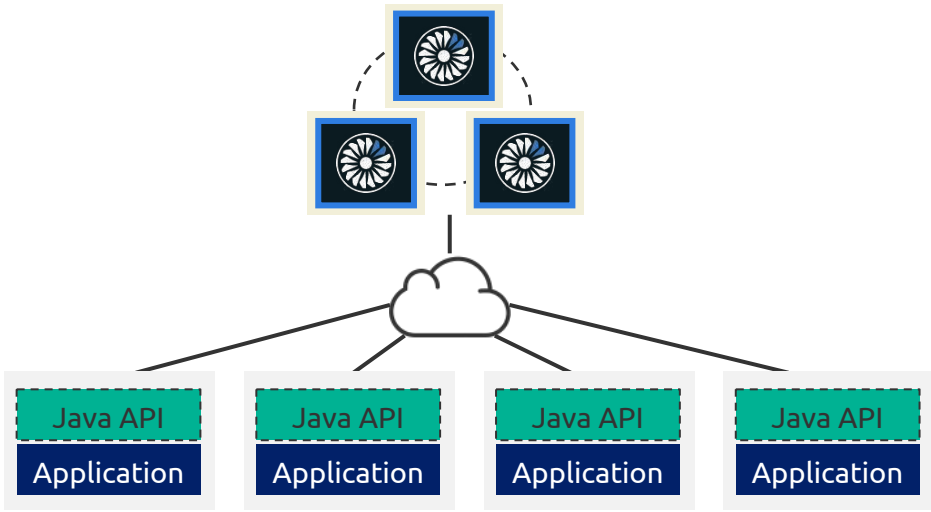


We use embedded in the labs

```
// 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);`
  - `jet.newJob(pipeline).join();`
- Stop cluster when processing is done

hazelcast

Lab 1: Filter Records from Stream

- Step 1: Run your first streaming job
  - Open Lab 1
  - Explore the boilerplate
  - Run the lab and follow results

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

Lab 1: Filter Records from Stream

- Step 2: Filter out odd numbers

- Step 3: Read stream from a file instead of the generator
  - Use `Sources.fileWatcher` source

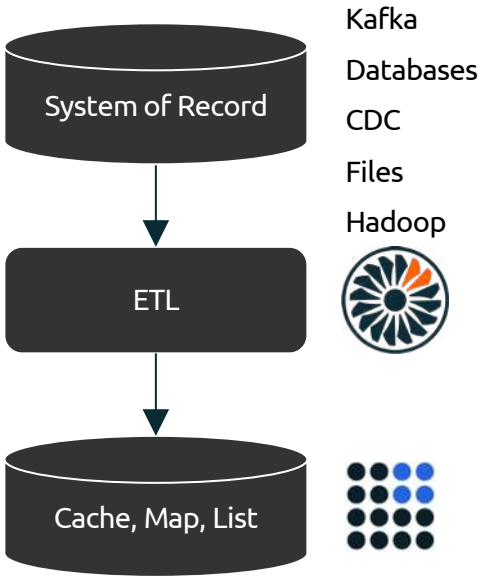
hazelcast

What We Learned…

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

Hot Cache

- Offload data to a distributed cache
  - Enrichment
  - In-memory compute
- Jet brings:
  - Speed (works in parallel)
  - Fault Tolerance
  - Declarative API - Focus on business logic, not on infrastructure / integration



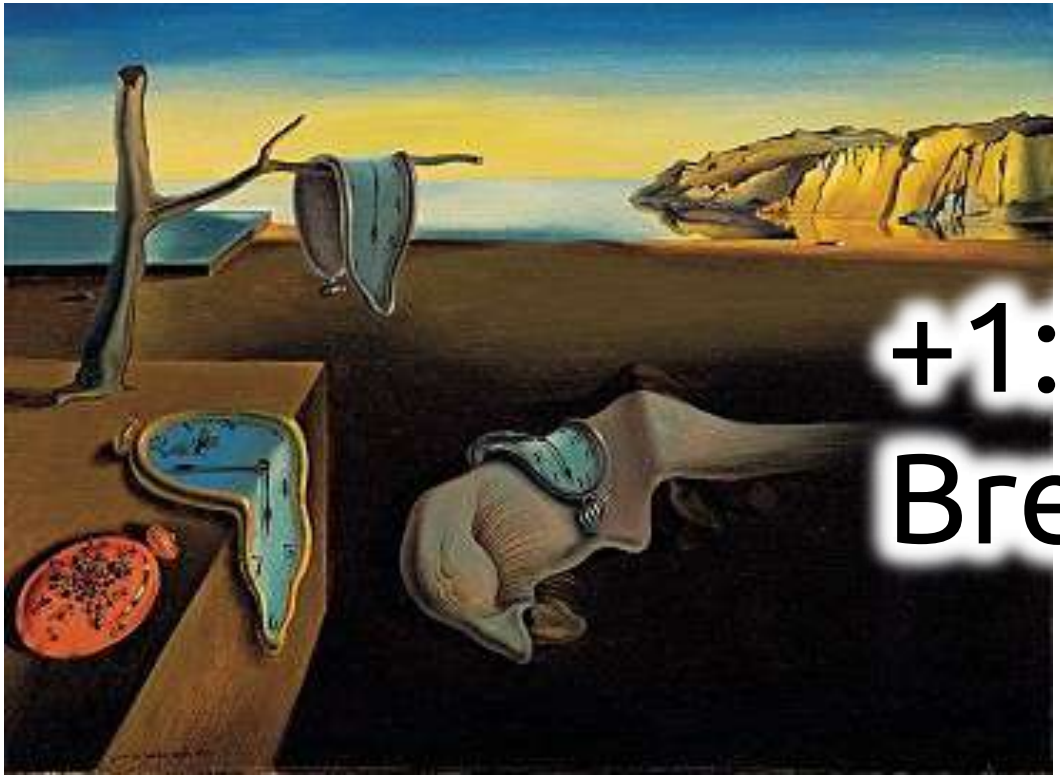
hazelcast

Honorable Mentions

- Connectors in Jet Library
  - Hazelcast, Journal, Kafka, HDFS, JMS, JDBC, Elasticsearch, MongoDB, InfluxDB, Redis, Socket, File
  - CDC connectors using Debezium (4.0 January)
- 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

Time Check

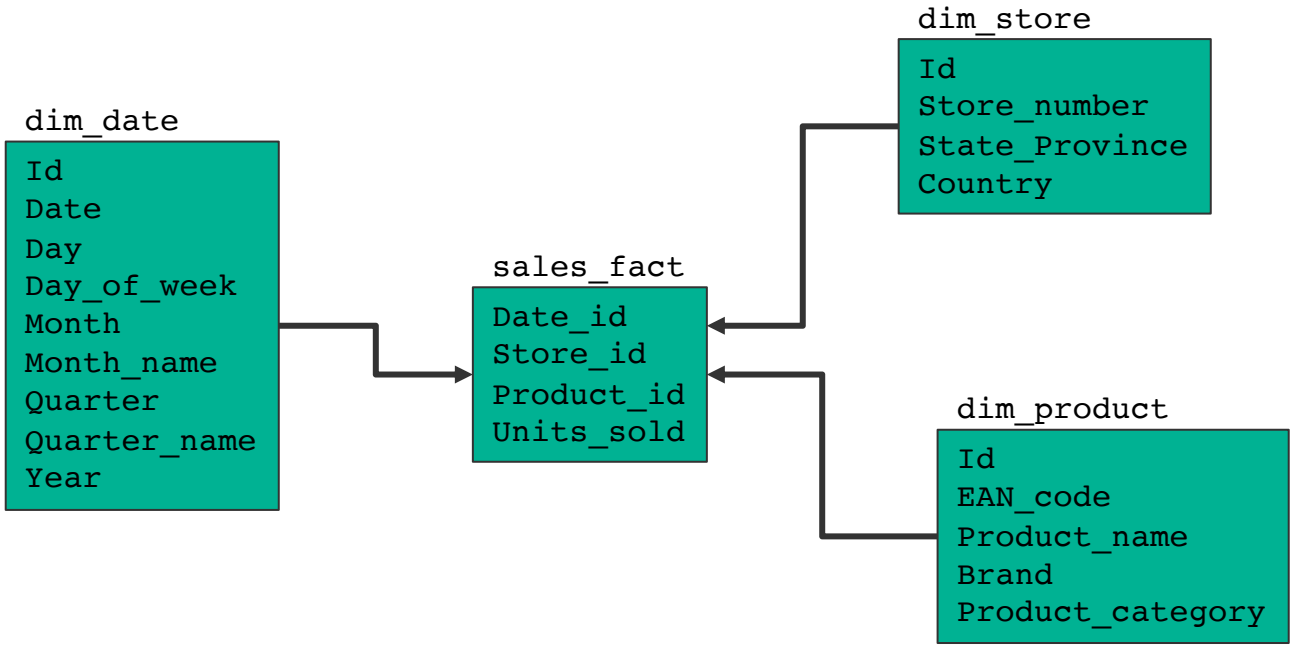


hazelcast

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)
  - Advantge: always up to date
  - Disadvantage: slow retrieval
- Cache local the custer
  - Hazelcast Jet contains rich distributed cache (thanks to embedded IMDG)

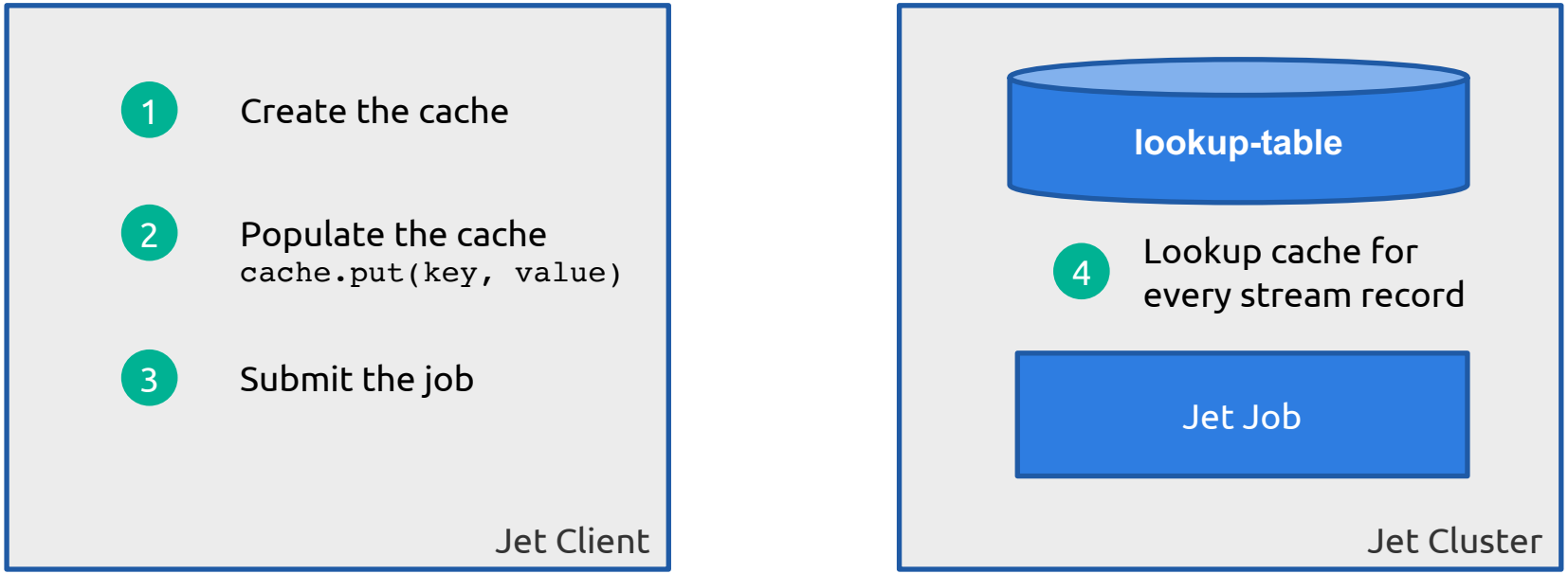
hazelcast

Caching in Hazelcast Jet

- Jet comes with distributed in-memory data structures
  - Implements Java collections and JCache
  - Map, List, Queue
- Data partitioned and distributed across the cluster
- Elastic, scales with cluster size (more nodes = more space)
- Read/Write through for databases

hazelcast

Lab - Architecture



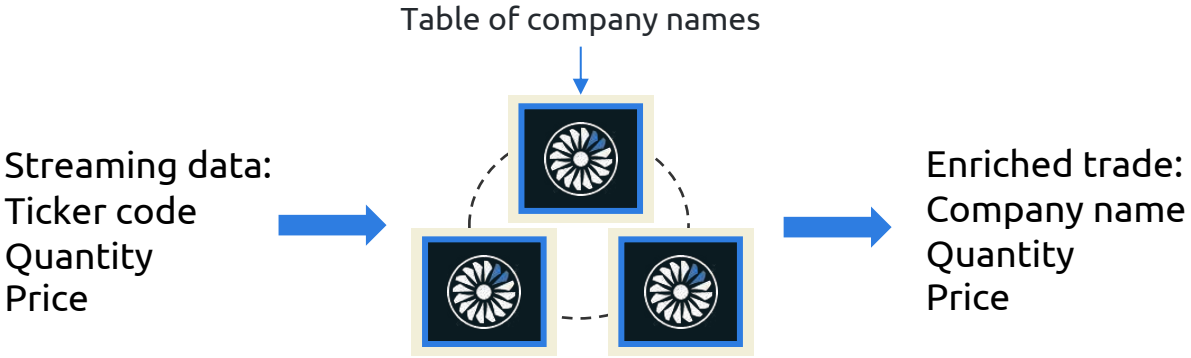
hazelcast

Enrichment API in Jet

```
items.mapUsingIMap(
      // the name of the lookup cache in the cluster
      "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

hazelcast

What We Learned…

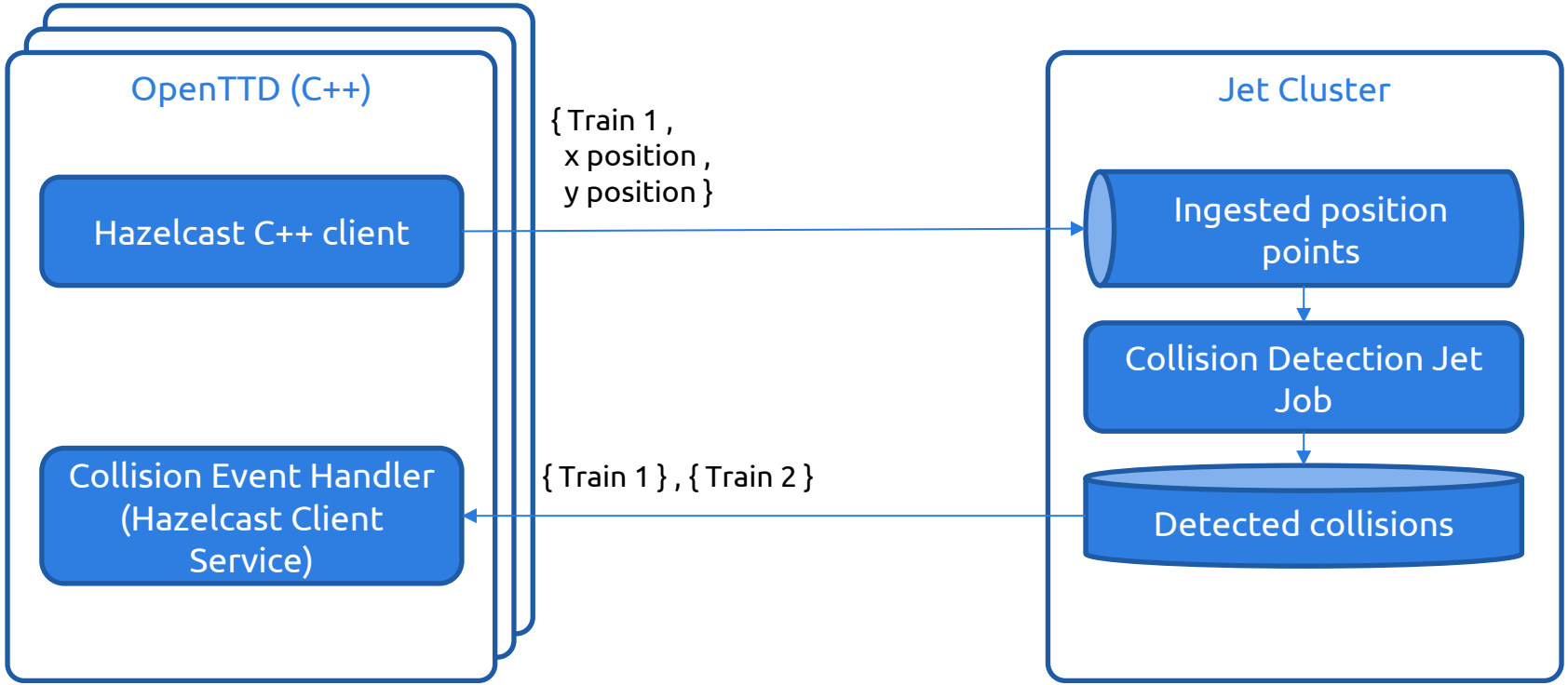
- Trading off flexibility and performance
  - Enrich from local memory
  - Remote lookup
  - Enrichment from a cache
- https://blog.hazelcast.com/ways-to-enrich-stream-with-jet/
- Operators: hashJoin, mapUsingImap, mapWithContext
- Jet Cluster provides powerful caching services

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 as a Platform - Train Demo



https://github.com/vladoschreiner/transport-tycoon-demo

hazelcast

Coffee Break

+2:00

15 Minute Break

hazelcast

Before we start, please

1. Download Hazelcast Enterprise

   https://hazelcast.com/download/customer/#hazelcast-jet

2. Send me an e-mail to vladimir@hazelcast.com
   - To get the slides and the trial license
   - Useful before 4th part of the workshop

hazelcast

Aggregations and Stateful Streaming

Starting at 2 PM

Stateful Processing

- Map with a state
- Remembers events or intermediate results
- Use-Cases:
  - Pattern matching
  - Correlation (of multiple events)
  - Complex Event Processing

hazelcast

Stateful Processing API in Jet

```
items.mapStateful(
        // Object that holds the state. Must be mutable!
        LongAccumulator::new,
        // Mapping function that updates the state
        (sum, currentEvent) -> {
            sum.add(currentEvent); // Update the state object
            return (sum.get() <= THRESHOLD)
                    ? null // Nothing is emitted
                    : sum.get();
        }
 );
```

hazelcast

Lab 3: Stateful Processing

- Detect if price between two consecutive trades drops by more then 200
  - Return the price difference if drop is detected
- Ignore various trade symbols for now

hazelcast

Aggregations

- Combine multiple records to produce a single value

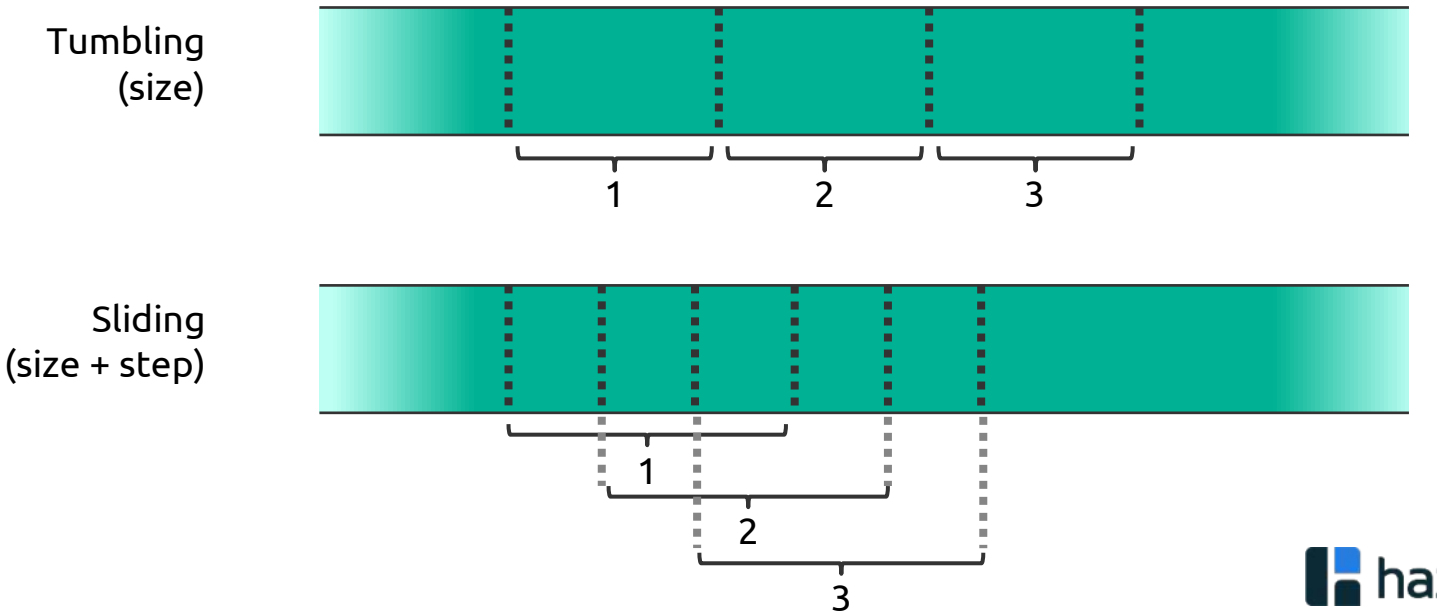
- Aggregate operation - How to aggregate records?
  - Examples: sum, average, min, max, count
- Scope - Which data to aggregate?
- Frequency - When is the result provided?

- Special case of Stateful processing

hazelcast

Scope - which data to aggregate?

- Streaming data is mostly time series
- Windows - time ranges to assign records to



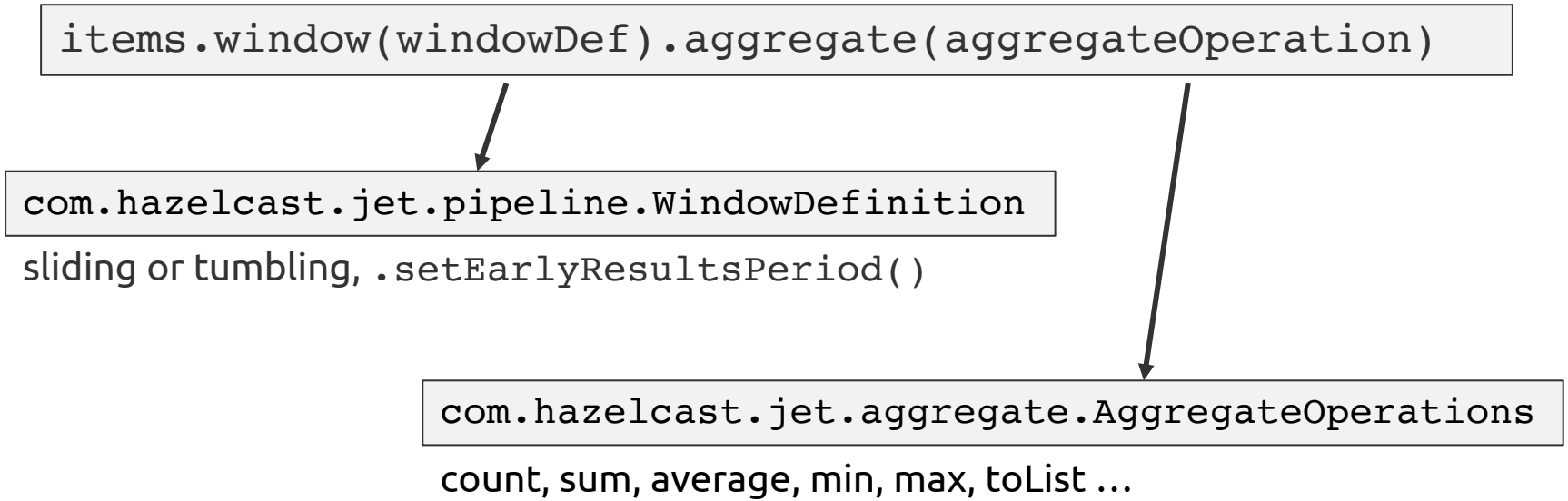
hazelcast

Frequency - When is the result provided?

- When window closes
  - All the data in the window were processed
  - Non-practical for long time ranges (hour and more)
- In specified time intervals - early results
- With each item
  - Use stateful mapping

hazelcast

Windowing API in Jet



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
- Compute sum of trades in last 3 seconds, update each second
  - Sliding windows
  - Output each second, should be roughly constant

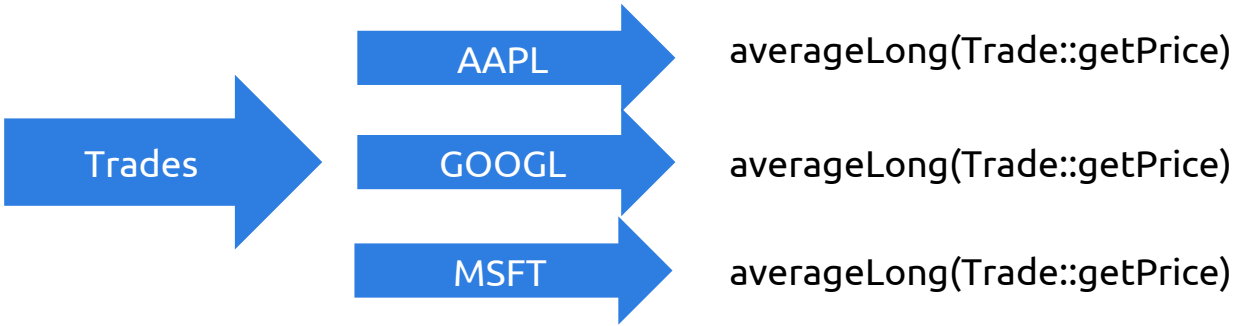
Grouping

- 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 4: Windowing with Grouping

- Compute sum of trades **per ticker** in last 3 seconds, update each second
  - Per-ticker results for the previous lab

hazelcast

Timestamps

- Timestamps necessary to assign records to window
- Uses wall clock - Ingestion-time processing
  - Simple but possibly not correct!

```
.withIngestionTimestamps()
```

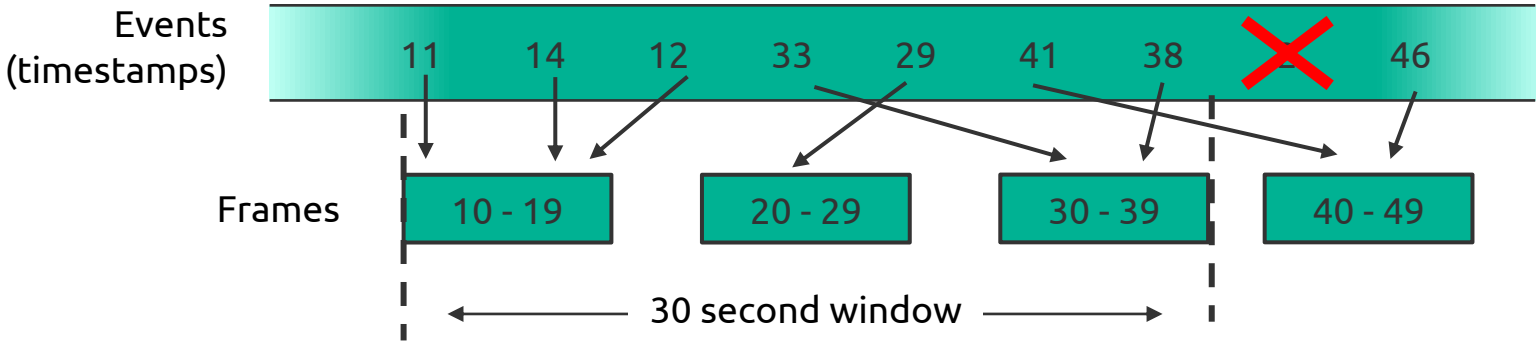
- Timestamp embedded in record - Event-time processing
  - Need to specify how to extract the timestamp

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

```
.withNativeTimestamps()
```

hazelcast

Event Time Processing



- No ordering
  - How long will we wait for stragglers?
- Watermarks
  - “No older items will follow."
  - Trading off latency for correctness

hazelcast

Aggregations and State - Takeaways

- Operator remembers the events or intermediate results
- Aggregations is a special case: combines multiple input records to a single value
  - Define the scope of the aggregation: whole dataset or a window
  - Aggregated result is provided
    - With each record
    - When the scope has been processed
    - Something in between
- Split the stream using a key and aggregate the sub-streams (GROUP BY)

Honorable Mentions

- Custom aggregation API
- Stateful mapping with timeouts and custom evictor

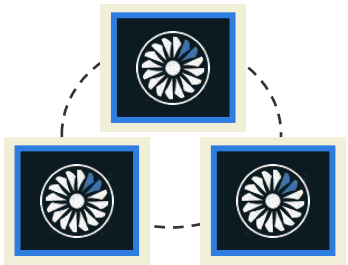
- Cascade aggregate operations
  - Aggregate aggregations (e.g. maximal average value)
  - Flight Telemetry Demo
- Use grouping for join and correlation
  - Windowing to be added for streaming join
  - https://docs.hazelcast.org/docs/jet/latest/manual/#cogroup

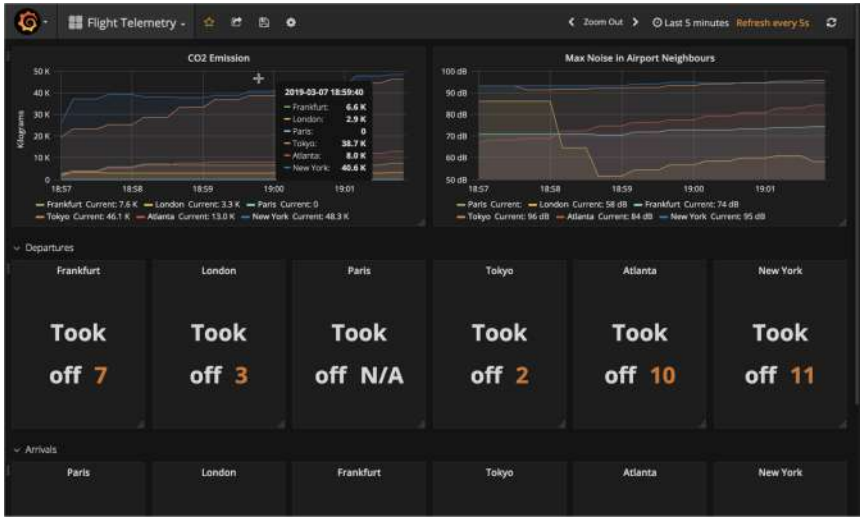
hazelcast

DEMO: Flight Telemetry

"Jets with Jet" - https://github.com/hazelcast/hazelcast-jet-demos

Streaming flight data:
Type of aircraft
Aircraft position

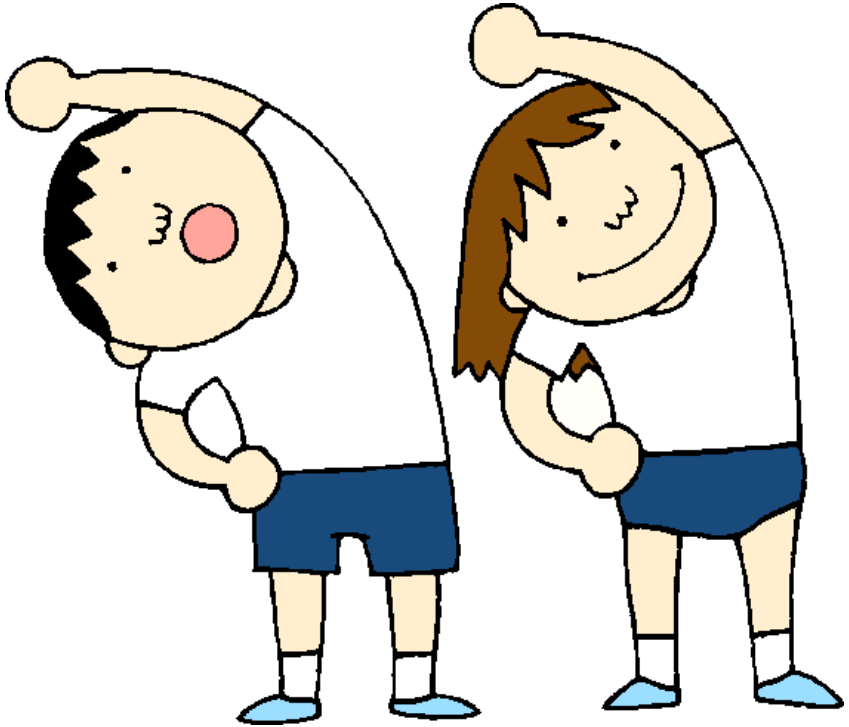




- Filter to defined airports only
- Sliding over last 1 minute, detect altitude changes
- Based on the plane type and phase of flight, calculate noise and CO_2 emissions

hazelcast

Stretch Break



+3:05

5 Minute Break

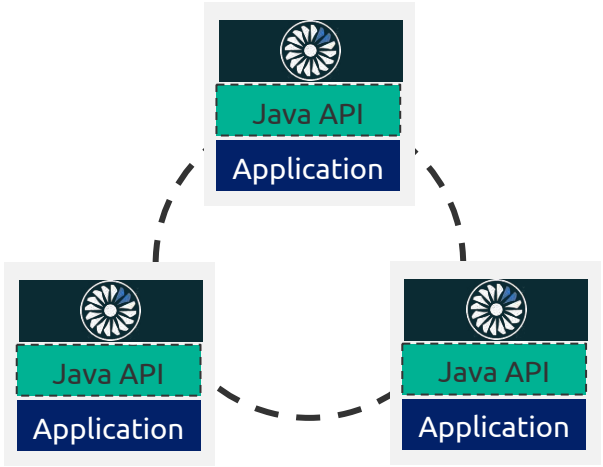
Please do not leave the room

hazelcast

Scaling and Operations

Deployment Options

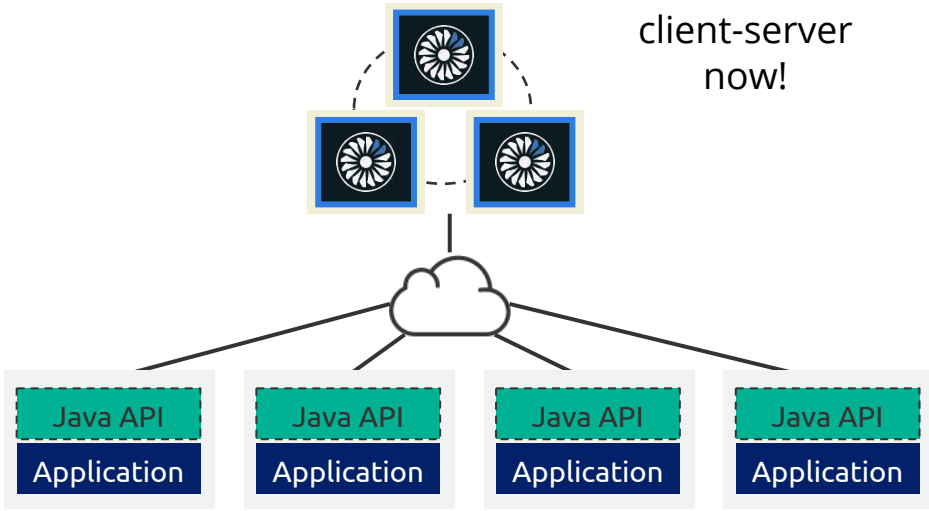
Embedded



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

Client/Server

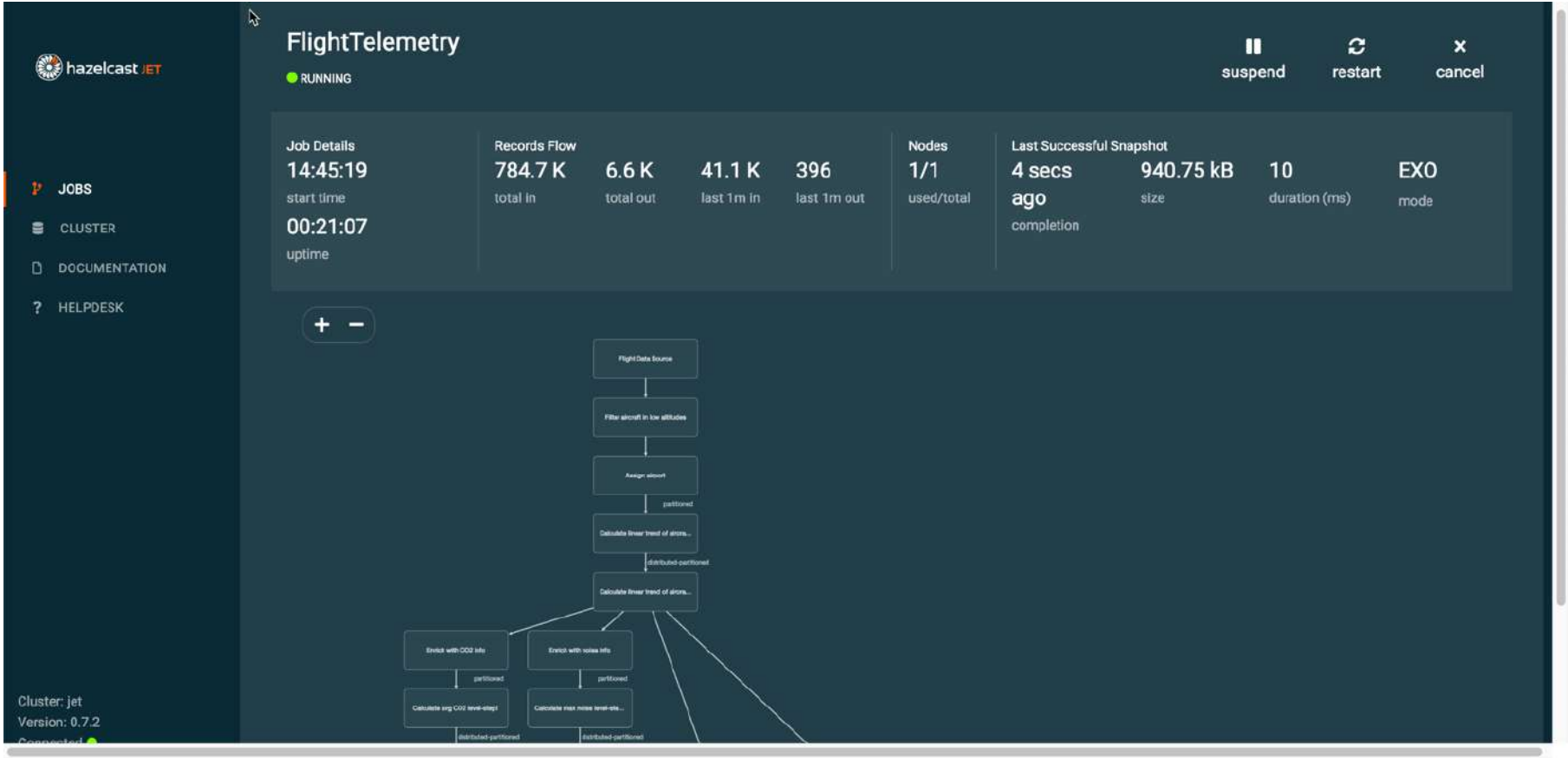
We switch to client-server now!



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

hazelcast

Jet Management Center



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

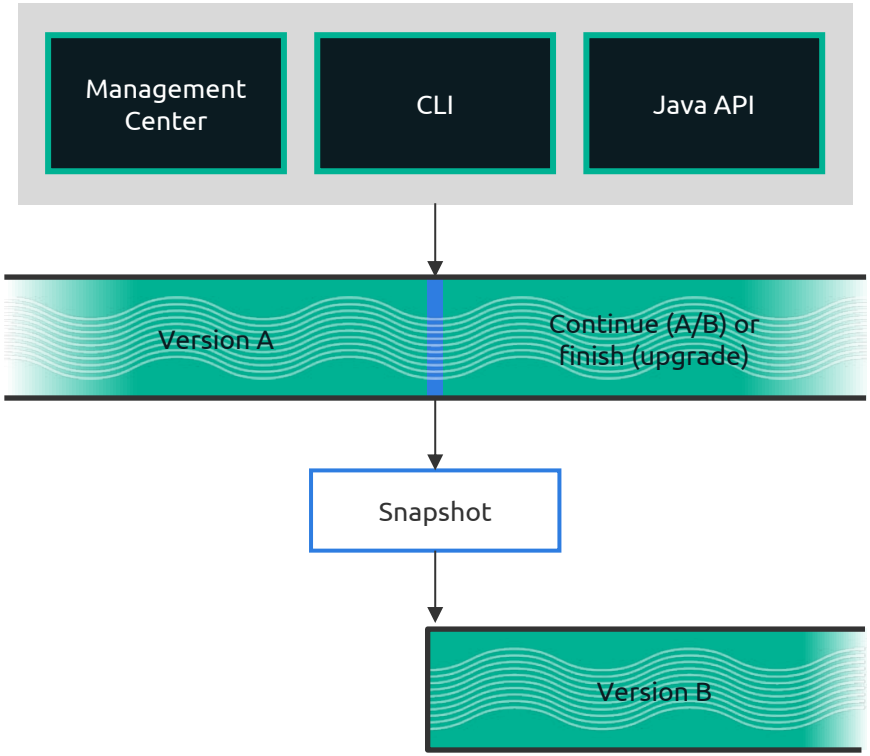
Scaling and Fault Tolerance

- 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

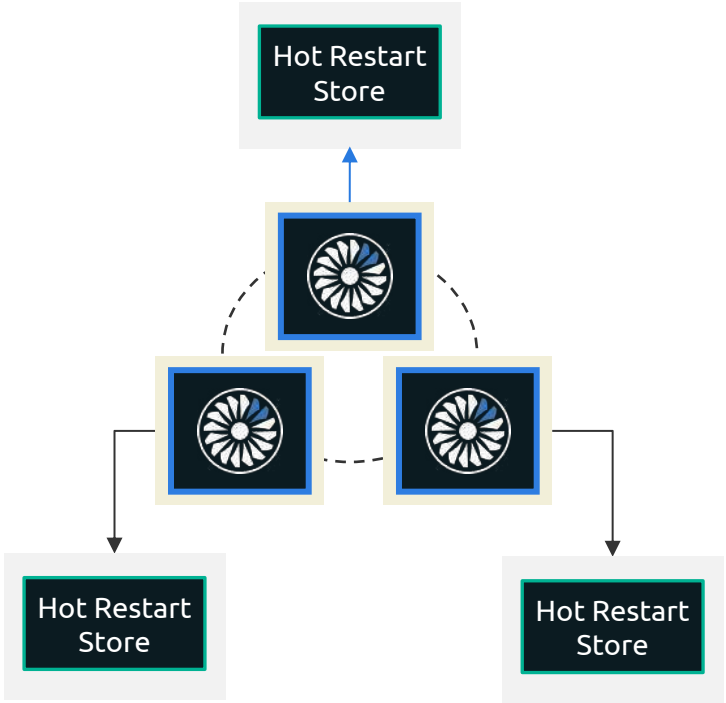
Job Upgrade



- 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

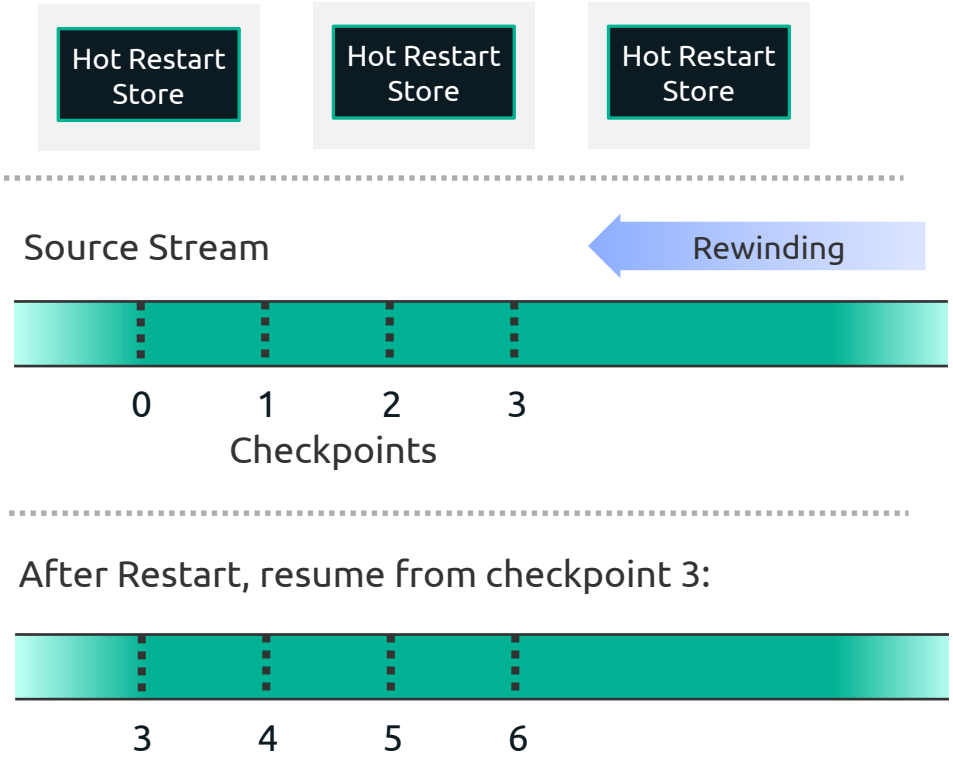
Lossless Recovery: Before Lights Out



- 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

Ops demo

https://github.com/vladoschreiner/hazelcast-jet-job-upgrade/tree/instructions

hazelcast

Lab 5: Start Jet from cmd line

- Get Jet ZIP and unpack it
  https://hazelcast.com/download/customer/#hazelcast-jet
- Jet must be configured not to join clusters within the room
  - `<multicast enabled="false">` `.g/hazelcast.xml`
- Enal `s to <join>`

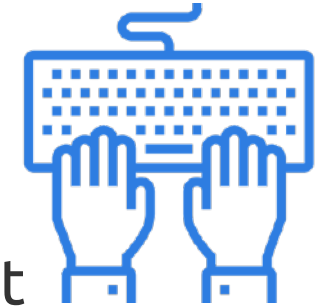
```
<tcp-ip enabled="true">
  <member>localhost</member>
</tcp-ip>
```

- Start a node

```
${JET}/bin/jet-start
```

hazelcast

Lab 5: Submit a Job using CLI



- Get the JAR by building the `job-deployment` module of the training project (mvn package)
- Submit the job

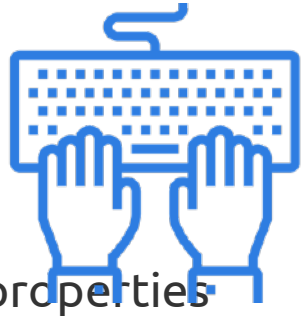
```
${JET}/bin/jet.sh submit
```

- Use the JAR produced

```
${LABS}/job-deployment/target/job-deployment-3.*.jar
```

hazelcast

Lab 5: Management Center



- Plug in the license to the MC to see the cluster
  - ${HZL}/hazelcast-jet-management-center/application.properties
- Start MC
  - ${HZL}/hazelcast-jet-management-center/jet-management-center.sh
  - http://localhost:8081
- Check the cluster and job state in the MC

hazelcast

Lab 5: Upscale the Job

- Add a node and observe the changes in the MC

```
${JET}/bin/jet.sh submit
```

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

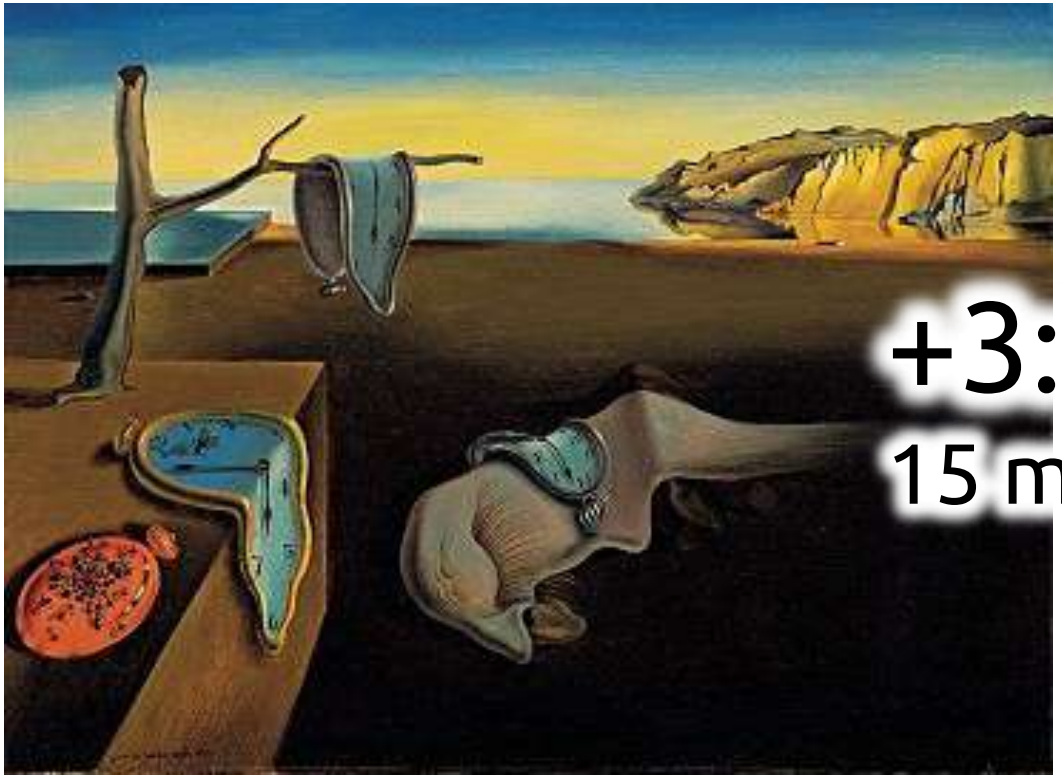
hazelcast

Honorable Mentions

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

hazelcast

Time Check



+3:45

15 minutes left!

hazelcast

Conclusion/Q&A

Helpful Resources

- Online training - https://training.hazelcast.com/stream-processing-essentials
- Jet website - https://jet.hazelcast.org/
  - especially the “Developer resources” section
- Reference card - https://dzone.com/refcardz/understanding-stream-processing
- Communities
  - Stack Overflow - Tag “hazelcast” or “hazelcast-jet”
  - Google Group- https://groups.google.com/forum/#!forum/hazelcast-jet
  - Gitter chat - https://gitter.im/hazelcast/hazelcast-jet
- Commercial
  sales@hazelcast.com
  or
  https://hazelcast.com/

hazelcast


Thank You
hazelcast