hazelcast JET

In-Memory Computing Summit

Greg Luck
@gregrluck

DISTRIBUTED COMPUTING.SIMPLIFIED.

What is Jet?

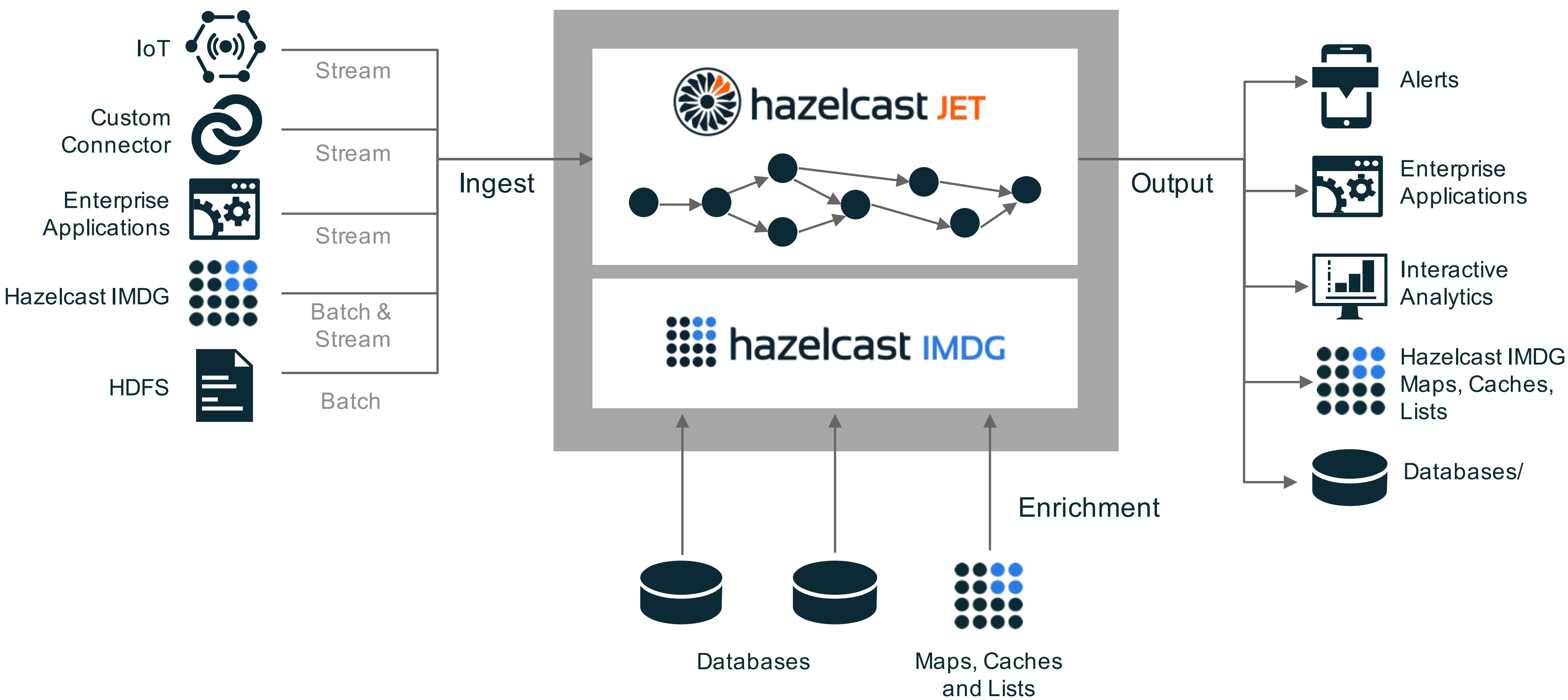
A general purpose **distributed data processing engine** built on **Hazelcast** and based on **directed acyclic graphs (DAGs)** to model data flow.

Why?

- Build a much faster engine than the industry has seen
- Better address IoT and streaming use cases than by using the computational capabilities within Hazelcast (`IExecutorService`, `EntryProcessor`, `JobTracker, …`)
- Provide a distributed `java.util.stream` and `java.util.function` implementation to get Java programmers started
- Relatively easy for us to do by OEMing Hazelcast IMDG
- Unifying IMDG and advanced data processing capabilities

Hazelcast Jet Overview

Stream and Batch Processing



Jet is a DAG Processing Engine

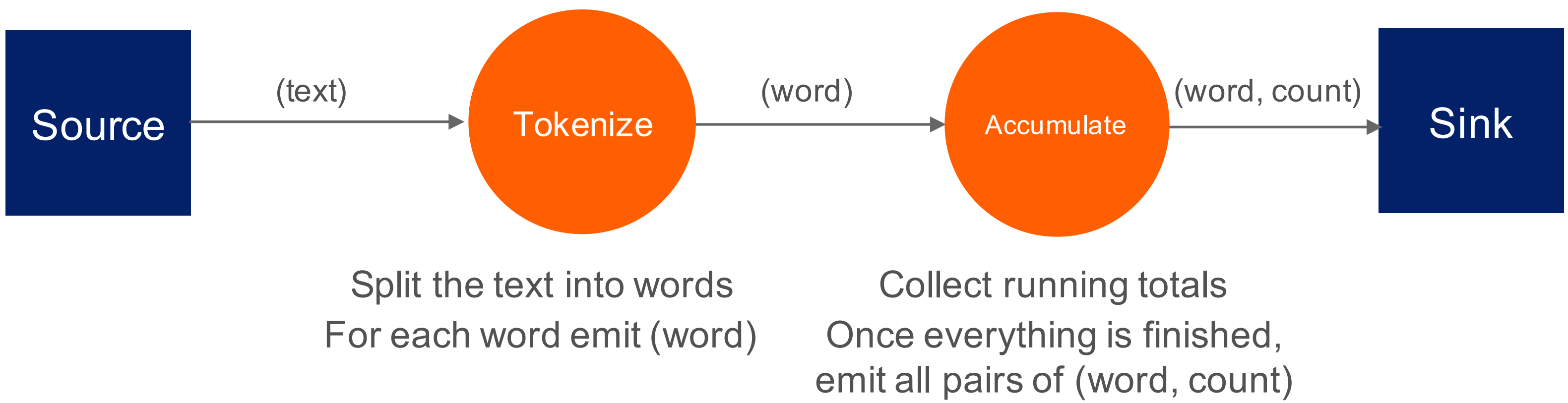
Example: Word Count

If we lived in a single-threaded world, before java.util.stream:

1. Iterate through all the lines
2. Split the line into words
3. Update running total of counts with each word

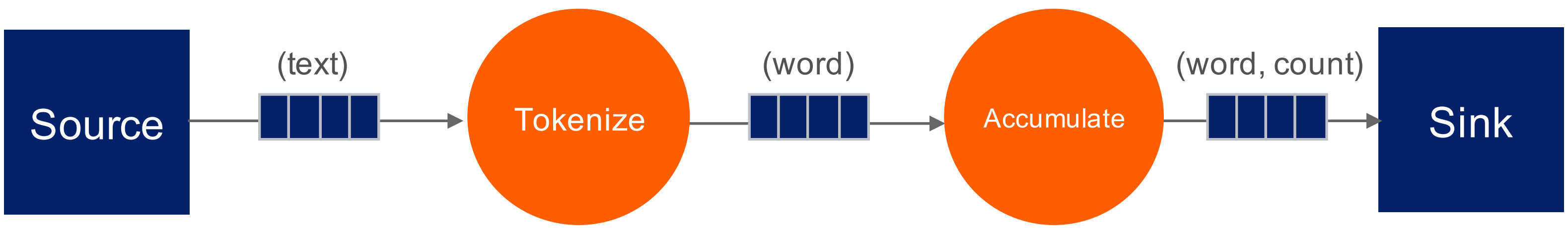
```
final String text = "...";
final Pattern pattern = Pattern.compile("\\s+");
final Map<String, Long> counts = new HashMap<>();

for (String word : pattern.split(text)) {
    counts.compute(word, (w, c) -> c == null ? 1L : c + 1);
}
```

We can represent the computation as a **DAG**



Still single-threaded execution:
each Vertex is executed in turn sequentially,
wasting the CPU cores

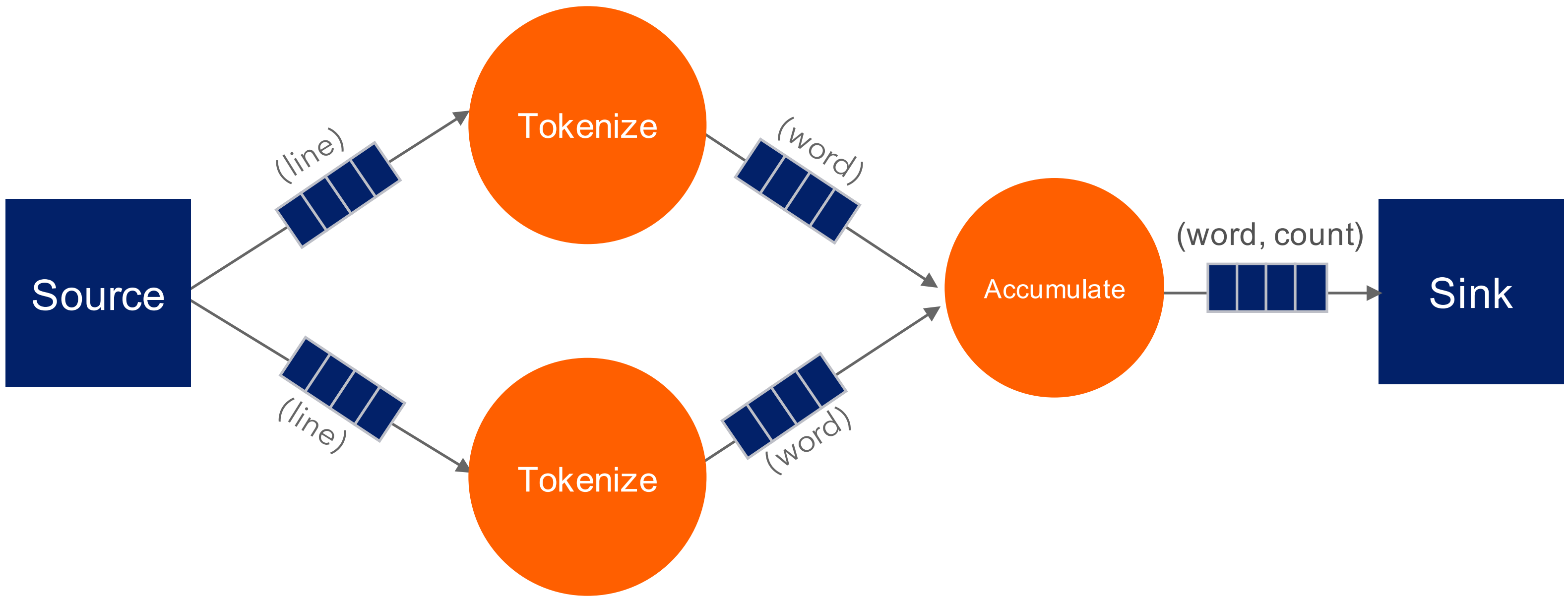
By introducing **concurrent queues** between the vertices we enable each vertex to run concurrently



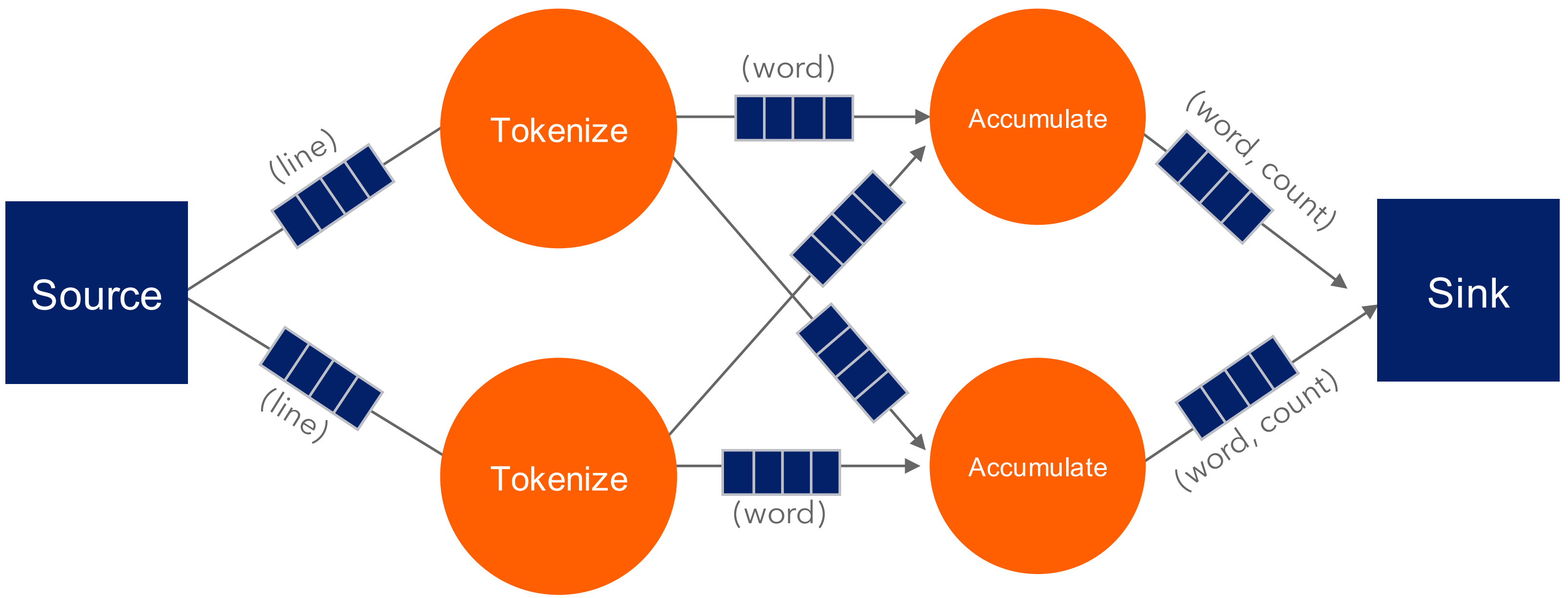
Split the text into words
For each word emit (word)

Collect running totals.
Once everything is finished,
emit all pairs of (word, count)

We can parallelize the Tokenize step by dividing the text into lines, and using multiple threads, thus using even more CPU cores concurrently.

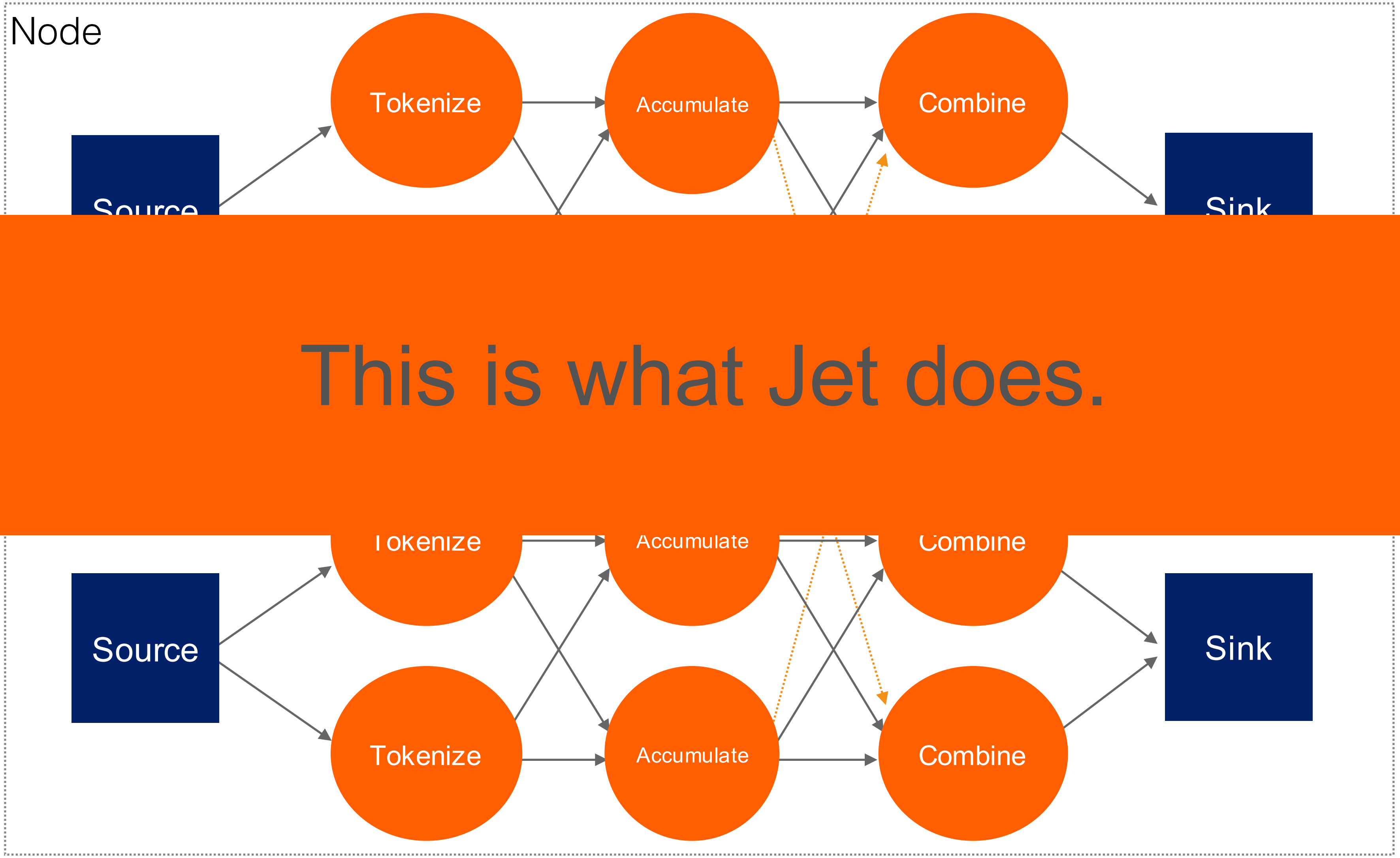


The Accumulator vertex can also be executed in parallel by **partitioning** the accumulation step by the individual words.



We only need to ensure the **same** words go to the **same** Accumulator.

The steps can also be distributed across multiple nodes. To do this you need a distributed **partitioning** scheme.

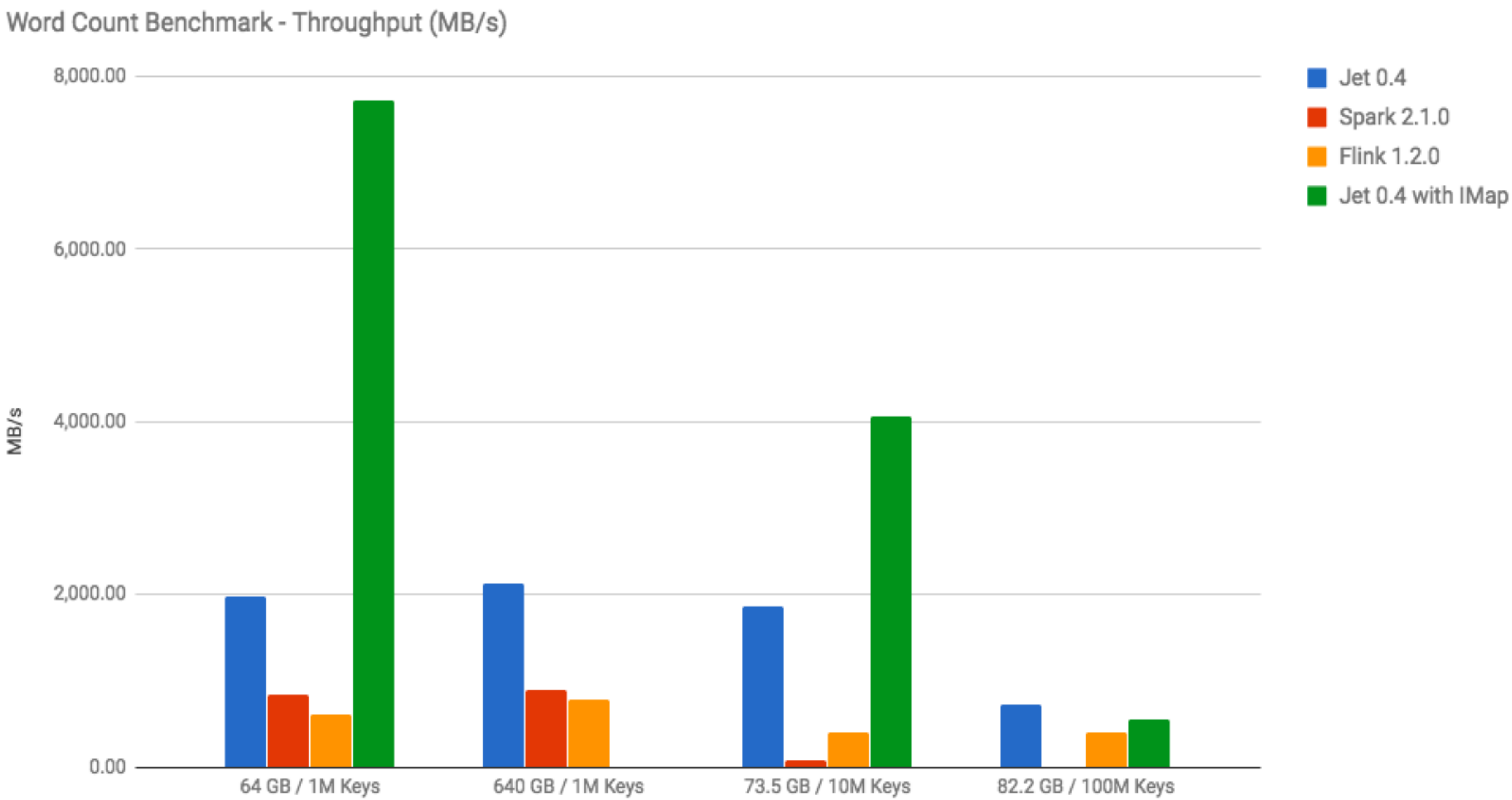


This is what Jet does.

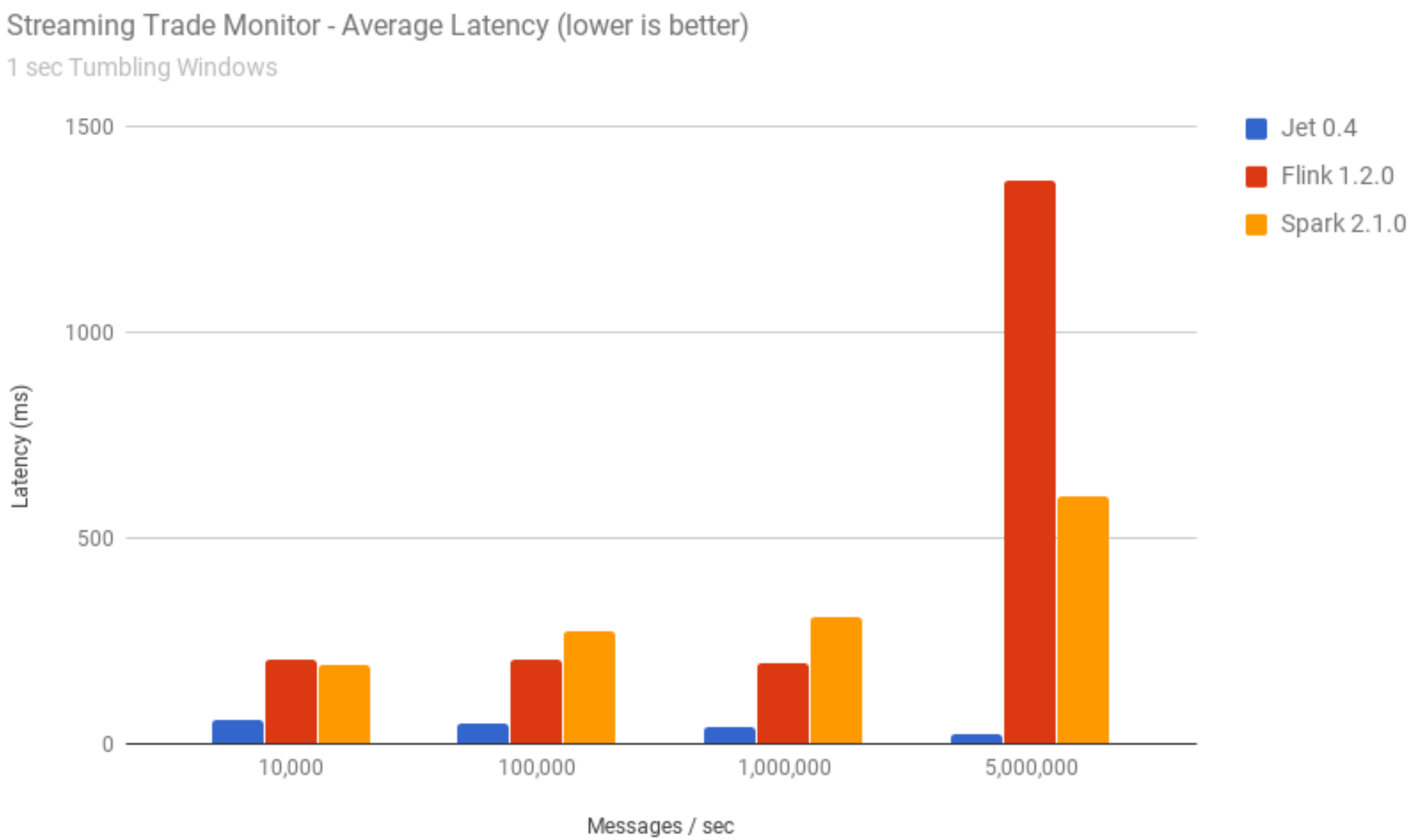
Hazelcast Jet Key Competitive Differentiators

- High Performance | *Industry Leading Performance*
- Works great with Hazelcast IMDG | *Source, Sink, Enrichment*
- Very simple to program | *Leverages existing standards*
- Very simple to deploy | *embed 10MB jar or Client Server*
- Works in every Cloud | *Same as Hazelcast IMDG*
- For Developers by Developers | *Code it*

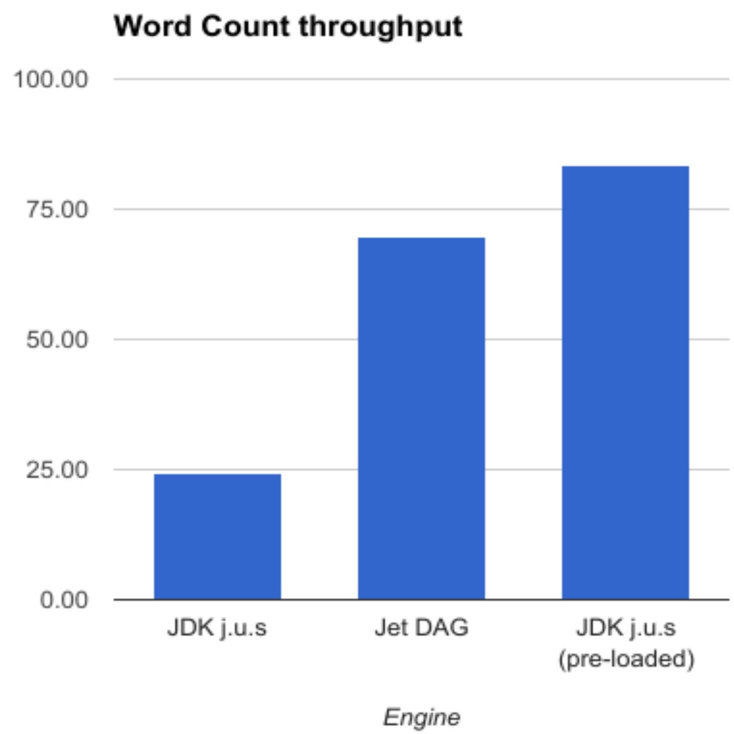
Performance - Throughput



Latency


Streaming Trade Monitor - Average Latency (lower is better)
1 sec Tumbling Windows
1500
1000
500
0
Latency (ms)
10,000
100,000
1,000,000
5,000,000
Messages / sec
Jet 0.4
Flink 1.2.0
Spark 2.1.0

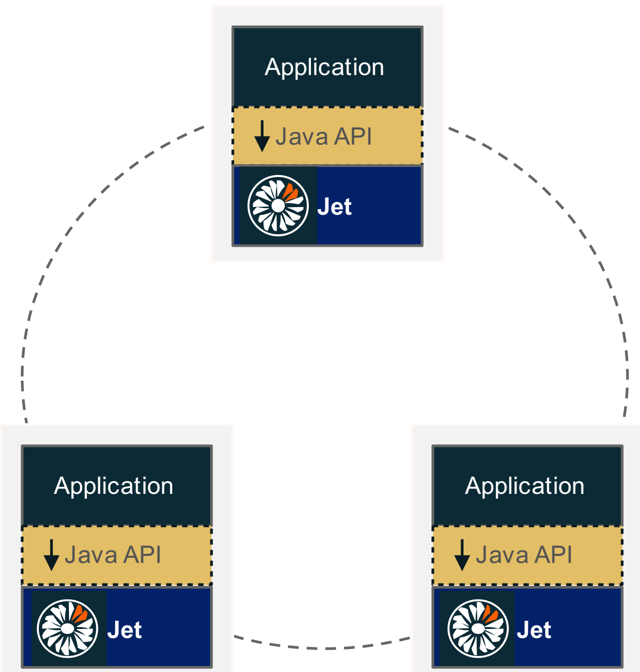
Performance - ForkJoin



- Jet is faster than JDK's j.u.s implementation. The issue is in the JDK the character stream reports "unknown size" and thus it cannot be parallelised.
- If you first load the data into a List in RAM then run the JDK's j.u.s it comes out a little faster

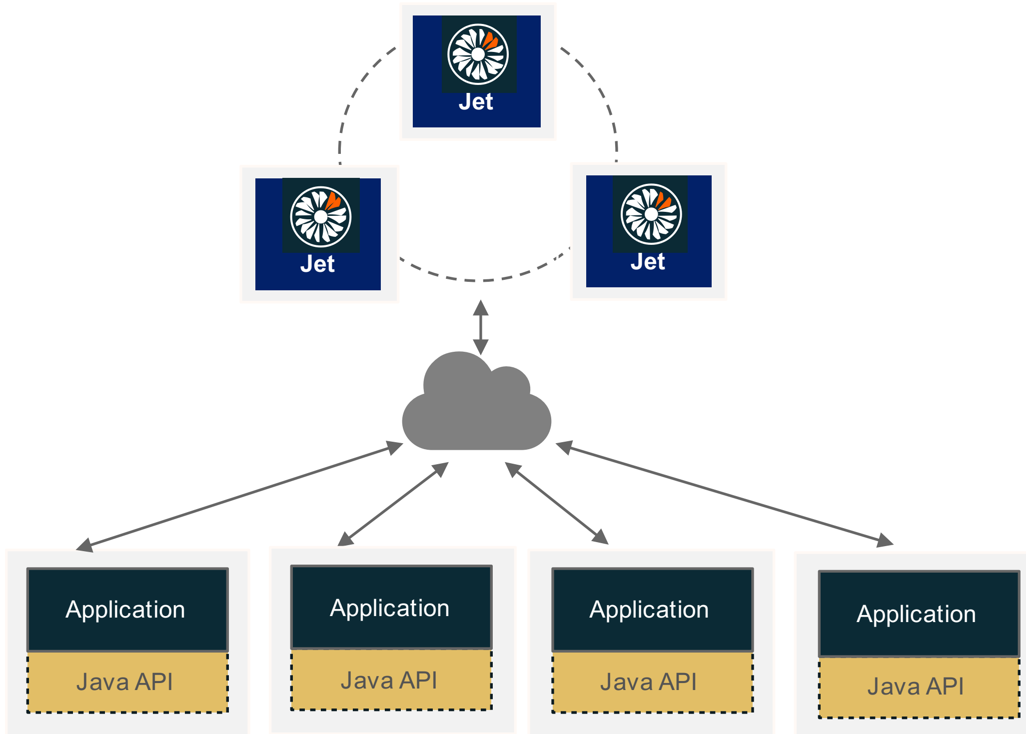
Jet Application Deployment Options

Embedded



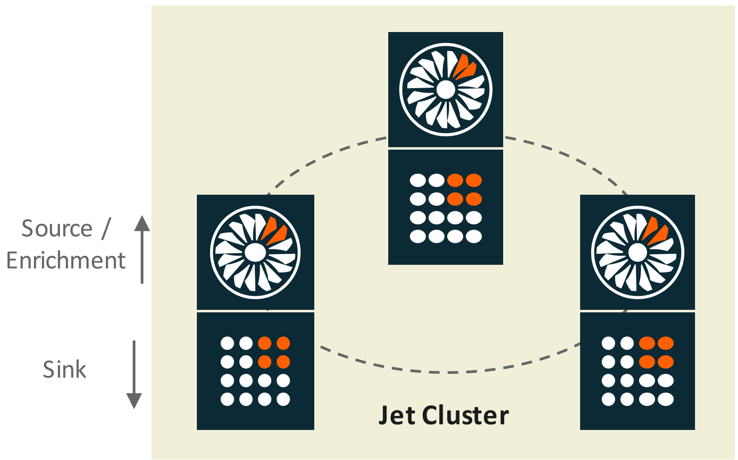
- No separate process to manage
- Great for microservices
- Great for OEM
- Simplest for Ops – nothing extra

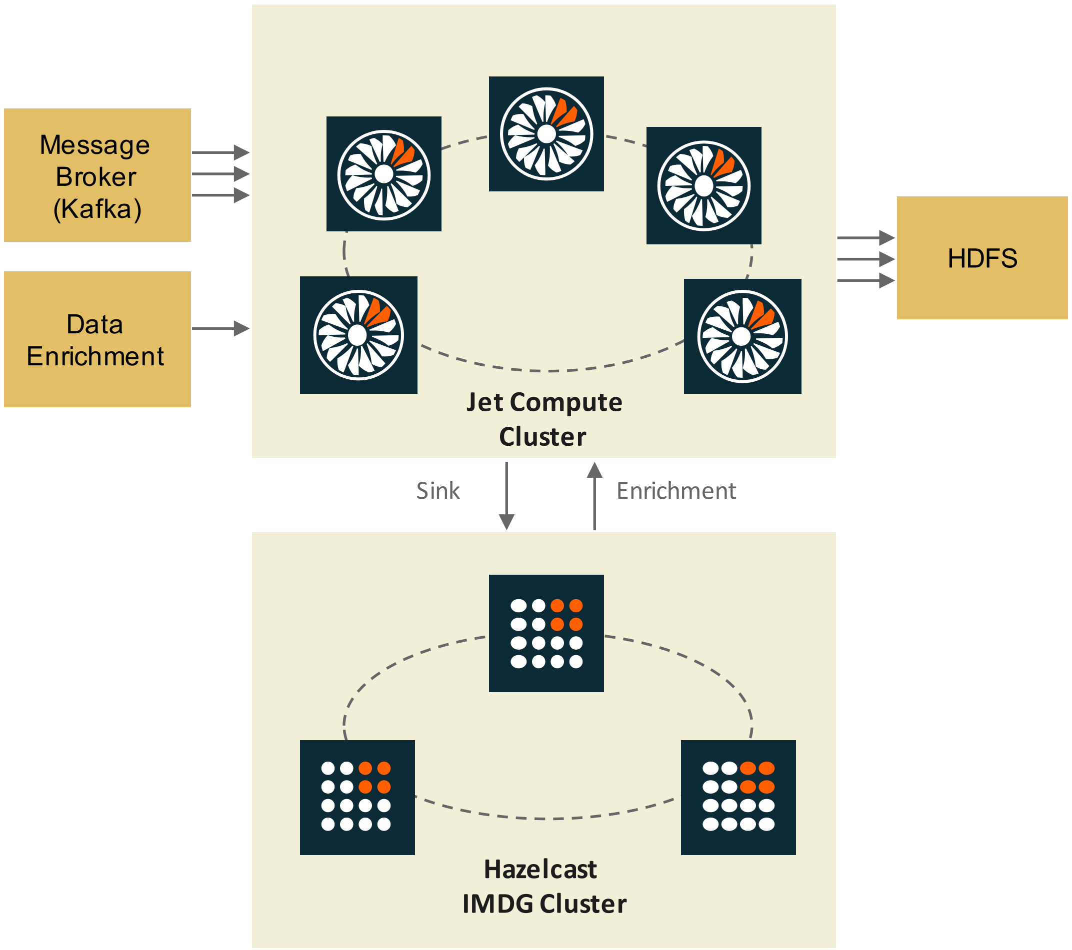
Client-Server



- Separate Jet Cluster
- Scale Jet independent of Applications
- Isolate Jet from Application Server Lifecycle
- Managed by Ops

Jet with Hazelcast Deployment Choices





Good when:
- Where source and sink are primarily Hazelcast
- Jet and Hazelcast have equivalent sizing needs

Good when:
- Where source and sink are primarily Hazelcast
- Where you want isolation of the Jet cluster

APIs

APIs

	Pipeline API (High-Level API)	java.util.stream	Core API (DAG API)
Use for	▪ General purpose High-Level API for processing both bounded and unbounded data.	▪ Filter-map-reduce operations on top of a bounded data set. ▪ Fast adoption, as j.u.stream is a well-known Java 8 API.	▪ Computations modelled as DAGs. ▪ If the use case is too specific to match the Pipeline API. ▪ For fine-tuned performance. ▪ For building DSLs.
Declarative (what) x Imperative (how)	Declarative	Declarative	Imperative
Map/FlatMap/Filter	✅	✅	✅
Aggregations	✅	✅	✅
Joins	✅	❌	✅
Processing bounded data (batch)	✅	✅	✅
Processing unbounded data (streaming)	*✅	❌	✅

Pipeline API

- General purpose, declarative API which is both powerful and simple to use.
- Recommended as the best place to start using Jet
- Supports fork, join, cogroup, map, filter, flatmap, reduce, groupby
- Works with all sinks and sources
- Is a DSL which is put through a planner and converted to DAG plan for execution.
- Batch and Streaming (window support in 0.6)
- See https://github.com/hazelcast/hazelcast-jet-code-samples

Pipeline API Code Sample

```
JetInstance jet = Jet.newJetInstance();
Pattern delimiter = Pattern.compile("\\W+");
Pipeline p = Pipeline.create();
p.drawFrom(Sources.<Long, String>readMap(BOOK_LINES))
  .flatMap(e -> traverseArray(delimiter.split(e.getValue().toLowerCase())))
  .filter(word -> !word.isEmpty())
  .groupBy(wholeItem(), counting())
  .drainTo(Sinks.writeMap(COUNTS));
Job job = jet.newJob(p);
job.join();
```

Distributed java.util.stream API

- Jet adds distributed support for the java.util.stream API for Hazelcast Map, List and Cache.
- Supports all j.u.s. operations such as:
  - map(), flatMap(), filter(), reduce(), collect(), sorted(), distinct()
- Lambda serialization is solved by creating **`Serializable`** versions of the interfaces
- j.u.s streams are converted to Processor API (DAG) for execution
- Strictly a batch processing API
- Easiest place to start, but we recommend the Pipeline API to exploit all features of Jet
- See https://github.com/hazelcast/hazelcast-jet-code-samples

j.u.s API

```
JetInstance jet = Jet.newJetInstance();
Jet.newJetInstance();
IStreamMap<Long, String> lines = jet.getMap("lines");

Map<String, Long> counts = lines
        .stream()
        .flatMap(m ->
Stream.of(PATTERN.split(m.getValue().toLowerCase())))
        .filter(w -> !w.isEmpty())
        .collect(DistributedCollectors.toIMap(”counts", w -> w,
 w -> 1L, (left, right) -> left + right));
```

DAG API – Powerful, Low Level API

DAG describes how vertices are connected to each other:

```
DAG dag = new DAG();
// nil -> (docId, docName)
Vertex source = dag.newVertex("source", readMap(DOCID_NAME));
// (docId, docName) -> lines
Vertex docLines = dag.newVertex("doc-lines",
        nonCooperative(flatMap((Entry<?, String> e) ->
                traverseStream(docLines(e.getValue()))))
    );
// line -> words
Vertex tokenize = dag.newVertex("tokenize",
        flatMap((String line) -> traverseArray(delimiter.split(line.toLowerCase()))
                                    .filter(word -> !word.isEmpty()))
);
// word -> (word, count)
Vertex accumulate = dag.newVertex("accumulate", accumulateByKey(wholeItem(),
    AggregateOperations.counting()));
// (word, count) -> (word, count)
Vertex combine = dag.newVertex("combine", combineByKey(AggregateOperations.counting()));
// (word, count) -> nil
Vertex sink = dag.newVertex("sink", writeMap("counts"));

return dag.edge(between(source.localParallelism(1), docLines))
          .edge(between(docLines.localParallelism(1), tokenize))
          .edge(between(tokenize, accumulate).partitioned(wholeItem(), HASH_CODE))
          .edge(between(accumulate, combine).distributed().partitioned(entryKey()))
          .edge(between(combine, sink));
```

Building Custom Processors

- Unified API for sinks, sources and intermediate steps
- Not required to be thread safe
- Each Processor has an **`Inbox`** and **`Outbox`** per inbound and outbound edge.
- Two main methods to implement:

```
boolean tryProcess(int ordinal, Object item)
```

  - Process incoming item and emit new items by populating the outbox

```
boolean complete()
```

  - Called after all upstream processors are also completed. Typically used for sources and batch operations such as **group by** and **distinct**.

- Non-cooperative processors may block indefinitely
- Cooperative processors must respect `Outbox` when emitting and yield if `Outbox` is already full.

Hazelcast Jet Architecture

Hazelcast Jet Architecture

	Java		Client
High-Level APIs	**Pipeline API** (Stream and Batch Processing)	**java.util.stream** (Batch Processing)	hazelcast JET
Connectors	**Streaming Readers and Writers** (Kafka, FileWatcher, Socket, IMap & ICache streamer)	**Batch Readers and Writers** (Hazelcast IMDG IMap, ICache & IList, HDFS, File)	
Processing	**Stream Processing** (Tumbling, Sliding and Session Windows)	**Batch Processing** (Aggregations, Join)	
Core	**Core API**		
	Fault-Tolerance (System state snapshots, Recovery after failure, At-least once or Exactly once)		
	Execution Engine (Distributed DAG Execution, Processors, Back Pressure Handling, Data Distribution)		
	Job Management (Jet Job Lifecycle Management, Resource Distribution and Deployment)		

Data Structures (Distributed Map, Cache, List)	hazelcast IMDG
Partition Management (Members, Lite Members, Master Partition, Replicas, Migrations, Partition Groups, Partition Aware)	
Cluster Management with Cloud Discovery SPI (Apache jclouds, AWS, Azure, Consul, etcd, Heroku, IP List, Kubernetes, Multicast, Zookeeper)	
Networking (IPv4, IPv6)	
Deployment (On Premise, Embedded, AWS, Azure, Docker, Pivotal Cloud Foundry)	

jet-core

- Provides low-latency and high-throughput distributed DAG execution
- Hazelcast provides clustering, partitioning, discovery, networking and serialization
- Each vertex in the graph is represented by **Processors**
- Vertices are connected by **Edges**.
- **Processors** are executed by **Tasklets**, which are allocated to threads.

Processors

- Main work horse of a Jet application – each vertex must have corresponding **Processors**
- Just Java code
- It typically takes some input, and emits some output
- Also can act as a **Source** or a **Sink**
- Convenience processors for: **map**, **filter**, **flatMap**, **groupByKey**, **coGroup, hashJoin** and several others

DAG Execution

- Each node in the cluster runs the whole graph
- Each vertex is executed by a number of **Tasklets** which correspond to the processors.
- Bounded number of execution threads (typically system processor count)
- **Back Pressure** is applied between vertices

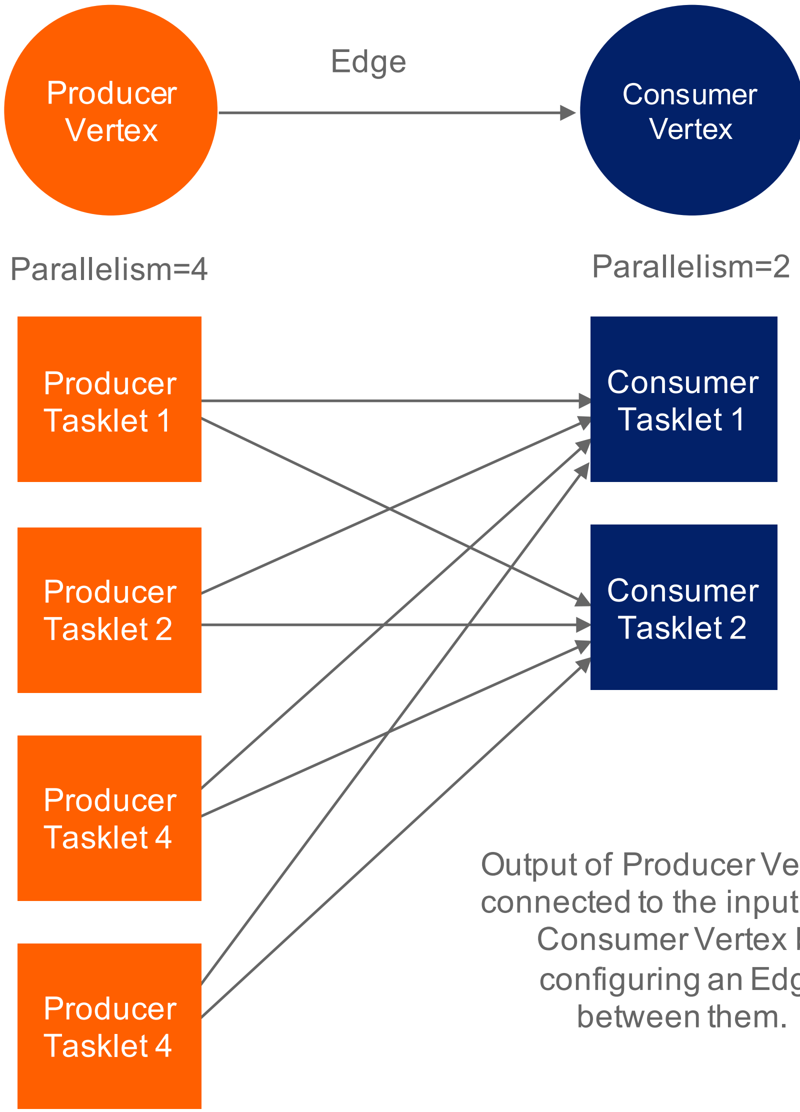
Cooperative Multithreading

- Similar to **green threads**
- Tasklets run in a loop serviced by the same native thread.
  - No context switching.
  - Almost guaranteed core affinity
- Each tasklet does a **small** amount of work at a time (<1ms)
- Cooperative tasklets must be **non-blocking**.
- Each native thread can handle thousands of cooperative tasklets
- If there isn't any work for a thread, it eventually backs off to a ceiling of 1ms to save CPU

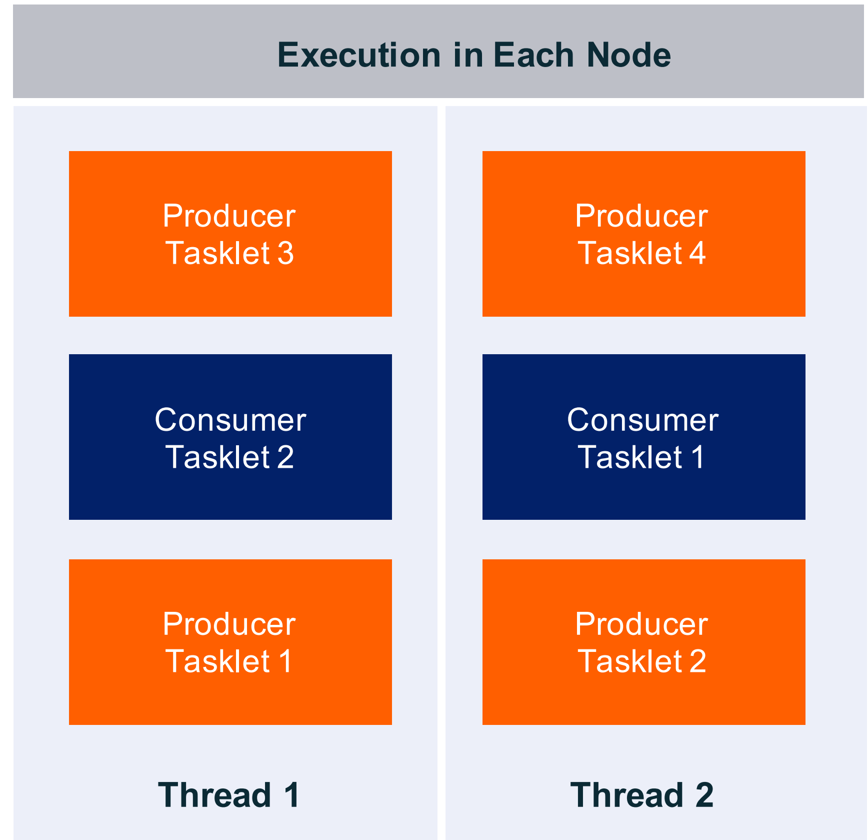
Cooperative Multithreading

- Edges are implemented by lock-free single producer, single consumer queues
  - It employs wait-free algorithms on both sides and avoids volatile writes by using lazySet.
- Load balancing via back pressure
- Tasklets can also be non-cooperative, in which case they have a dedicated thread and may perform blocking operations.

Tasklets – Unit of Execution



Output of Producer Vertex is connected to the input of the Consumer Vertex by configuring an Edge between them.



Parallelism setting controls how many tasklets are created on each Vertex. Default is number of cores

Edges

- Different types of edges:
  - **Unicast – pick any downstream processor**
  - **Broadcast – emit to all downstream processors**
  - **Partitioned – pick based on a key**
- Vertices can have more than one input: allows joins and co-group
- Vertices can have more than one output: splits and branching
- Edges can be **local** or **distributed**

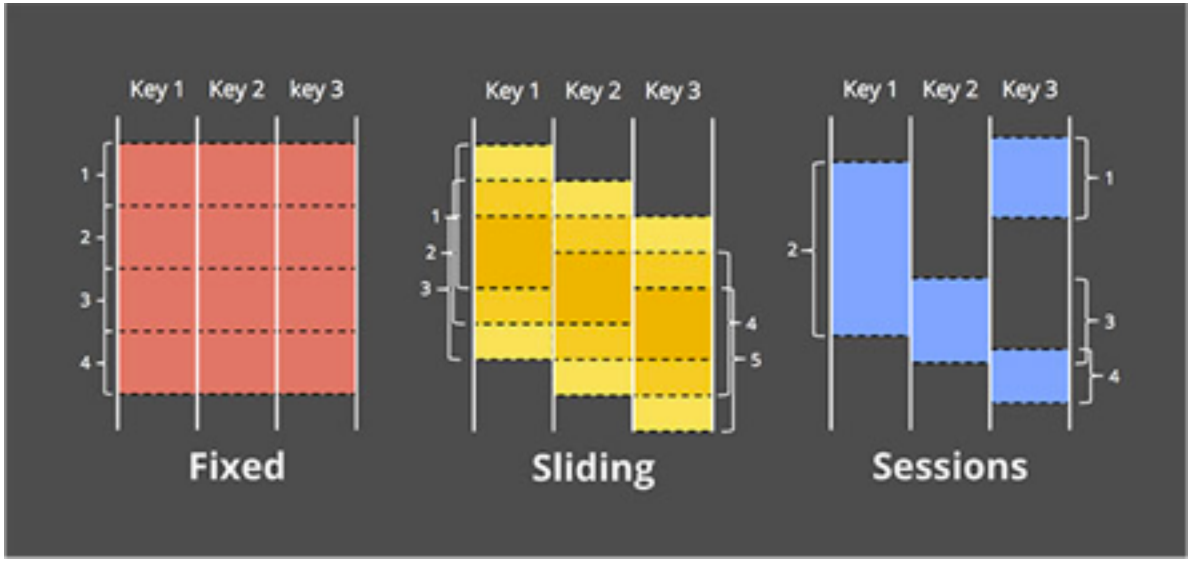
Data Input and Output

Sources and Sinks for:

- Hazelcast **`Icache (Jcache)`**, (batch and streaming of changes)
- Hazelcast **`Imap`** (batch and streaming of changes)
- Hazelcast **`Ilist`** (batch)
- HDFS (batch)
- Kafka (streaming)
- Socket (text encoding) (streaming)
- File (batch)
- FileWatcher (streaming – as new files appear)
- Custom, as sources and sinks are blocking **Processors**.

Stream Processing

- Support for events arriving out of order via Watermarks
- Sliding, Tumbling and Session window support



Job Management & Fault Tolerance

- Job state and lifecycle saved to IMDG IMaps and benefit from their performance, resilience, scale and persistence
- Automatic re-execution of part of the job in the event of a failed worker
- Tolerant of loss of nodes, missing work will be recovered from last snapshot and re-executed
- Cluster can be scaled without interrupting jobs – new jobs benefit from the increased capacity
- State and snapshots can be persisted to resume after cluster restart

Processing Guarantees

Guarantee	Snapshots	Performance
None	No	Fastest
At-Least Once	Yes	Slower
Exactly-Once	Yes	Slower

Questions?

Version 0.5 this week

http://jet.hazelcast.org

Minimum **JDK 8**