In-Memory Computing SUMMIT NORTH AMERICA 2018

Fast and Easy Stream Processing with Hazelcast Jet

Gokhan Oner
Hazelcast

Stream Processing
Why should I bother?

What is stream processing?

Data Processing: Massage the data when moving from place to place.

On-Line systems – request/response, small volumes, low-latency

Batch Processing – data in / data out, big volumes, huge latency

Stream processing – data in / data out, big volumes, low-latency

When to Use Stream Processing

- Real-time analytics
- Monitoring, Fraud, Anomalies, Pattern detection, Prediction
- Event-Driven Architectures
- Real-Time ETL
- Moving batch tasks to near real-time
- Continuous data
- Consistent Resource Consumption (1GB/sec -> 86TB/day)

What modern SPE can do for you?

- Offers high-level API to implement the processing pipeline
  - `map, filter, groupBy, aggregate, join …`
- Offers connectors to read and write the data
  - Kafka, HDFS, JDBC, JMS …
- *You implement the data pipeline and submit it to the SPE*
- Executes the pipeline in a parallel and distributed environment
- Moves the data through the pipeline (partitioning, shuffling, backpressure)
- Is fault-tolerant (survives failures) and elastic
- Monitoring, diagnostics etc.

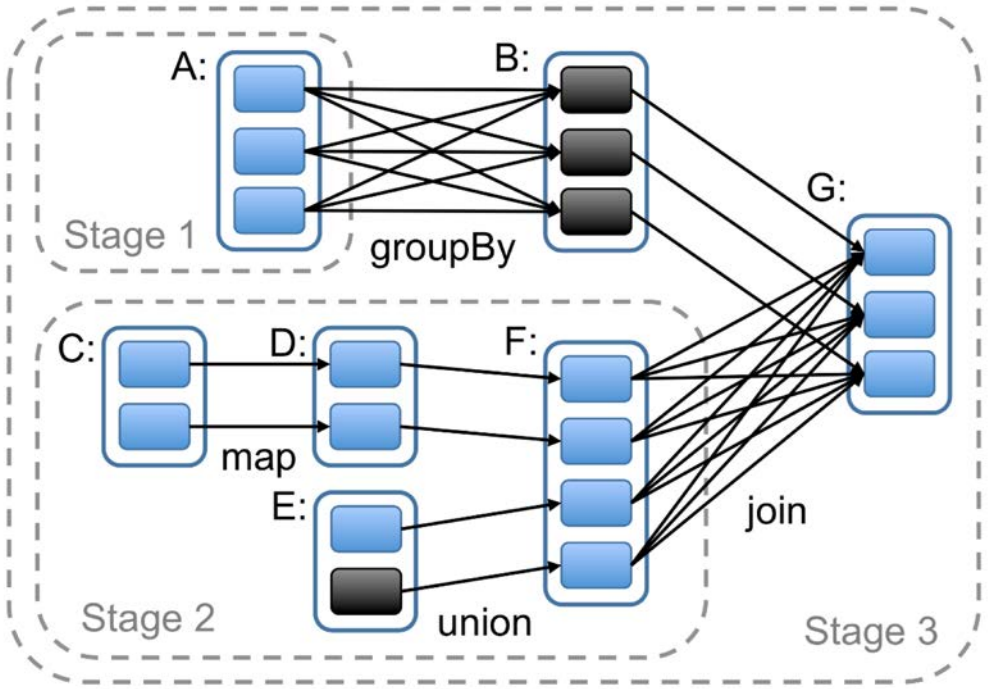
Streaming and Hadoop Major Evolutionary Steps

1st Gen: MapReduce

- Google File System Paper, 2003 https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf
- MapReduce paper, 2004 https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf
- Apache Hadoop founded by Doug Cutting and Mike Cafarella at Yahoo 2006
- Commercial Open Source Distros: Cloudera, Hortonworks, MapR
- Lots of additions to the ecosystem

2nd Gen: Spark

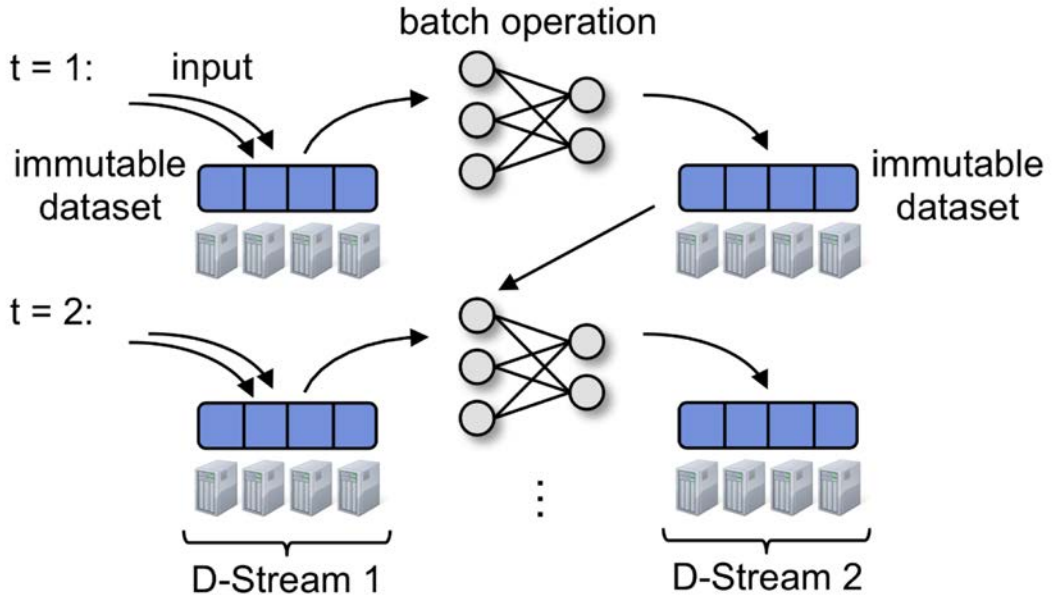
- Apache Spark started as a research project at UC Berkeley in the AMPLab, which focuses on big data analytics, in 2010.
- Goal was to design a programming model that supports a much wider class of applications than MapReduce. Introduced DAG



http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf

2nd Gen: Spark Streaming

- Fault-Tolerance of Spark designed with for batch
- Spark Streaming Paper, 2012. Stream as a sequence of micro-batches



- Spark has now moved on to DataFrames, Tungsten and Spark Streaming and its *architecture continues to evolve so it is also 3rd Gen (Continuous)*.

`http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf`

3rd Gen: Continuous streaming

- DAG based
- Streaming based. Not micro-batch
- Batch is a simply streaming with bounds
- Learns from previous systems
- Informed by academic papers in the last decade, such as the Google[1][2] but many more
- Plethora of Choice: Apache Storm, Twitter Heron, Apache Flink, Kafka Streams, Google DataFlow, Hazelcast Jet, Spark Continuous Streaming

[1] MillWheel: Fault-Tolerant Stream Processing at Internet Scale
[2] FlumeJava: Easy, Efficient Data-Parallel Pipelines

DISTRIBUTED COMPUTING. SIMPLIFIED.

What is Hazelcast Jet?

Distributed computation engine built on **Hazelcast IMDG**
using **directed acyclic graph (DAGs)** to model data flow

Hazelcast IMDG

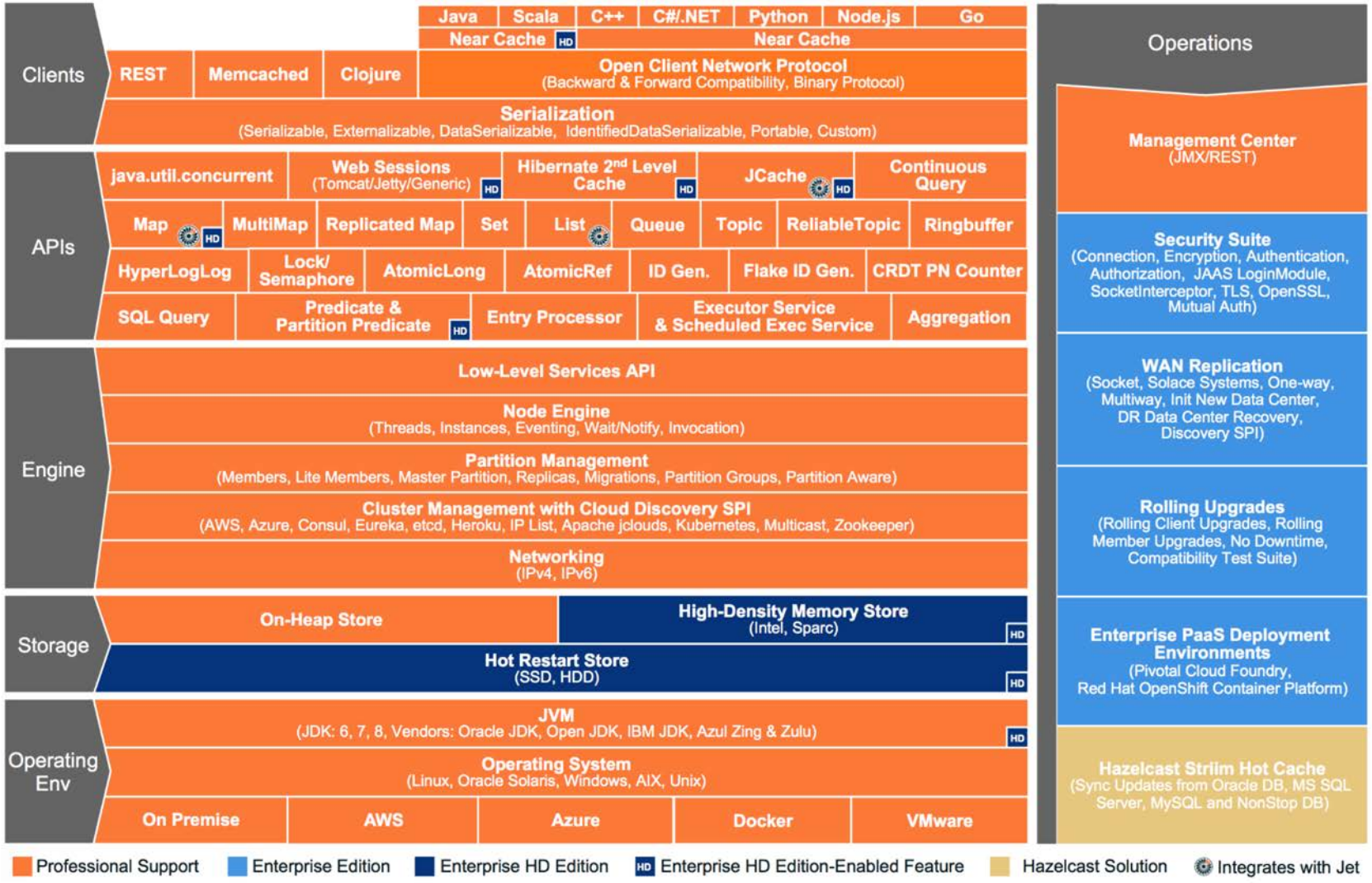
- IMDG = “In-Memory Data Grid”, distributed cache with computational capabilities and additional data structures
- Hazelcast values *simplicity:*

```
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import java.util.concurrent.ConcurrentMap;

public class UseHazelcastMap {
    public static void main(String[] args) {

        HazelcastInstance hz = Hazelcast.newHazelcastInstance();

        ConcurrentMap<String, String> map = hz.getMap("my-map");
        map.put("key", "value");
        String value = map.get("key");
        map.putIfAbsent("somekey", "somevalue");
        map.replace("key", "value", "newvalue");
    }
}
```

Hazelcast IMDG

- Distributed implementations of **ConcurrentMap**, **List**, **Queue**, **MultiMap, JCache**
- Distributed querying and data processing
- Distributed implementations of **java.util.concurrent** primitives (**AtomicLong**, **AtomicReference**, **CountdownLatch**,..)
- More features: **Ringbuffer**, **HyperLogLog** and Distributed **ExecutorService**.
- Embeddable single JAR

Hazelcast IMDG



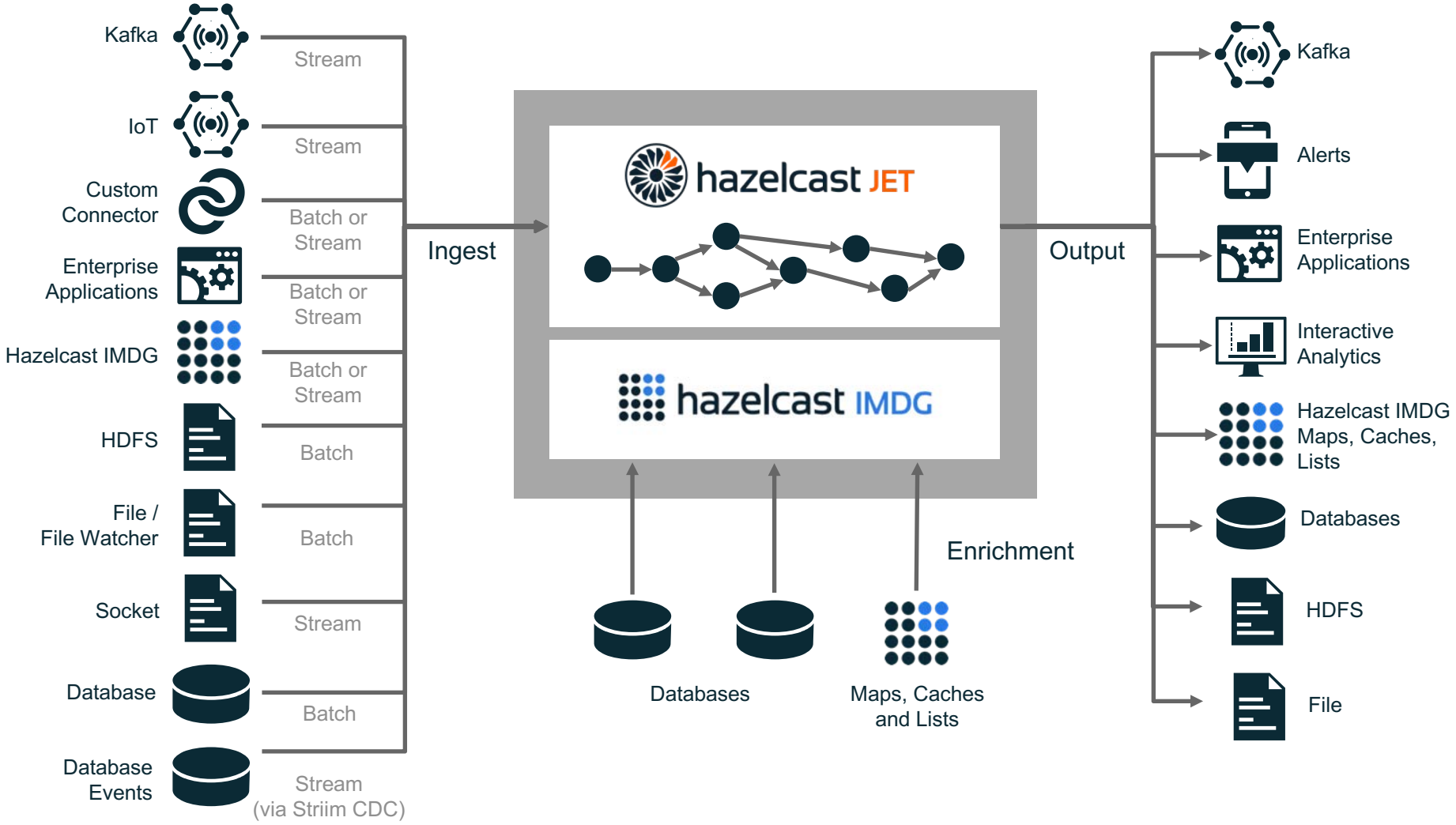
Hazelcast Jet

- Distributed data processing engine with in memory storage (through IMDG)
- Supports bounded (batch) and unbounded (stream) data sources
- Project started in 2015, first public release Feb 2017 with quarterly releases
- Single embeddable 10MB JAR
- JDK 8 minimum

What distributed means

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

Stream and Batch Processing



Jet goes far beyond IMDG

- Jet supports building general purpose data pipelines
- Several built in, distributed and scalable sources and sinks: IMDG, files, sockets, HDFS, Kafka, JMS, JDBC, ..
- Windowed aggregation of infinite streams
- At-least-once and exactly-once processing through distributed in-memory snapshotting
- Elasticity and fault tolerance to scale jobs up or tolerate outages.
- Extensive support for Java 8 lambdas with expressive API

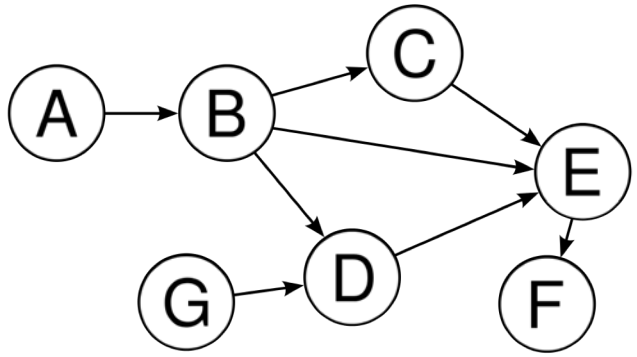
Twitter Cryptocurrency Sentiment Analysis

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

DAG Based Processing

DAG Based Processing

- Directed Acyclic Graphs are used to model computations



- Each vertex is a step in the computation
- It is a generalisation of the MapReduce paradigm
- Supports both batch and stream processing
- Other systems that use DAGs: Apache Tez, Flink, Spark, Storm…

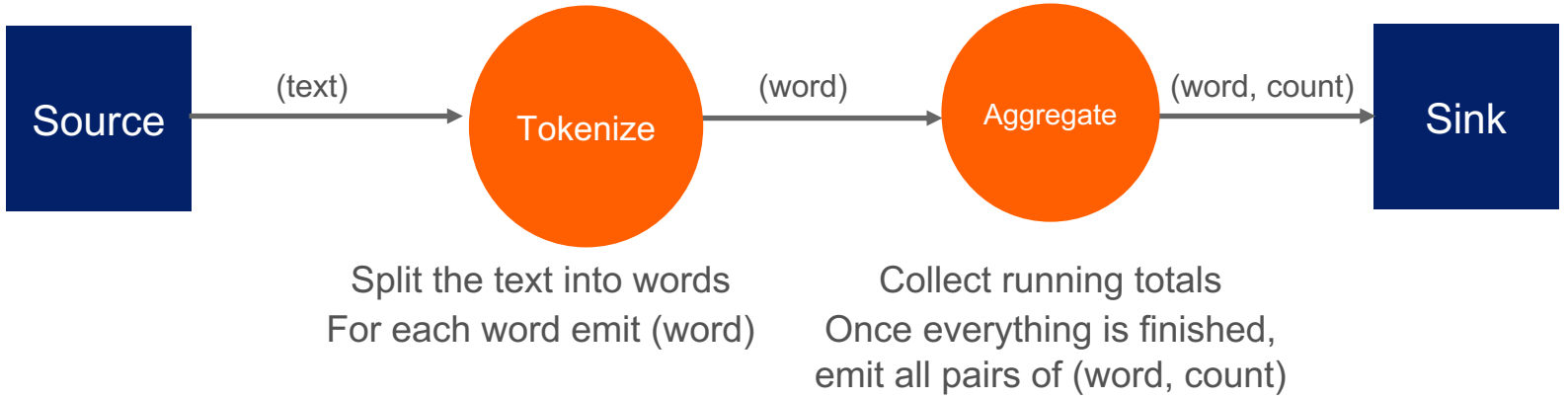
Example: Word Count

If we lived in a single-threaded world:

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

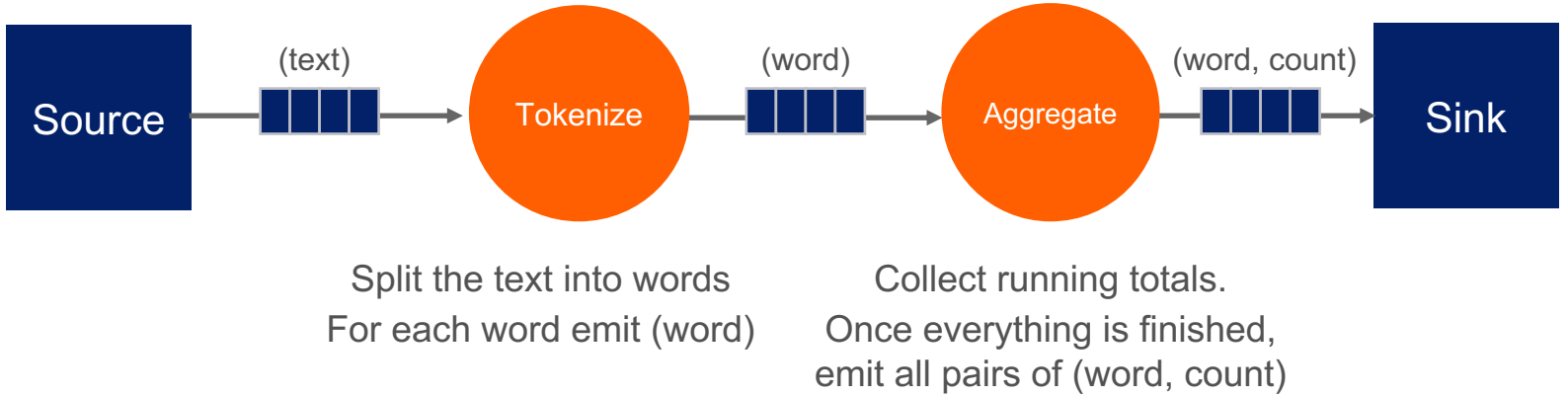
```
final String text = "...";
final Pattern pattern = Pattern.compile("\\s+");
final Map<String, Long> counts = new HashMap<>();

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

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

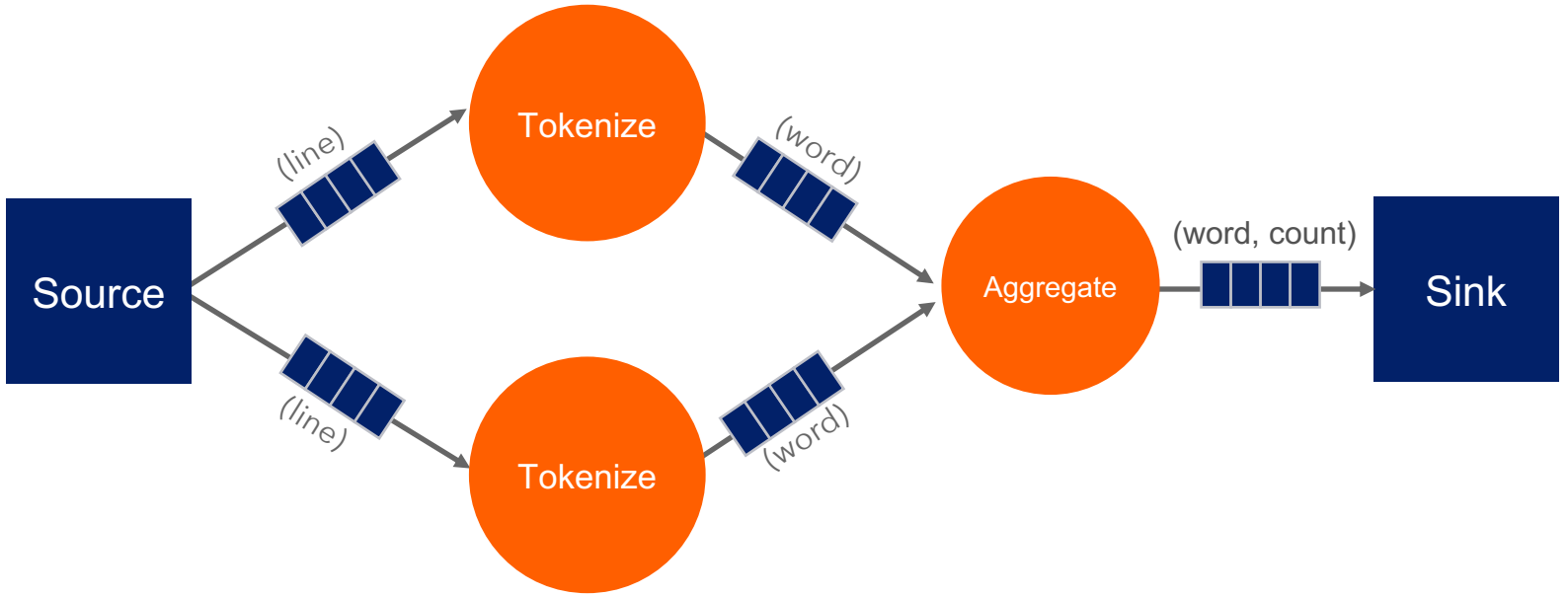


Still single-threaded execution:
each Vertex is executed in turn

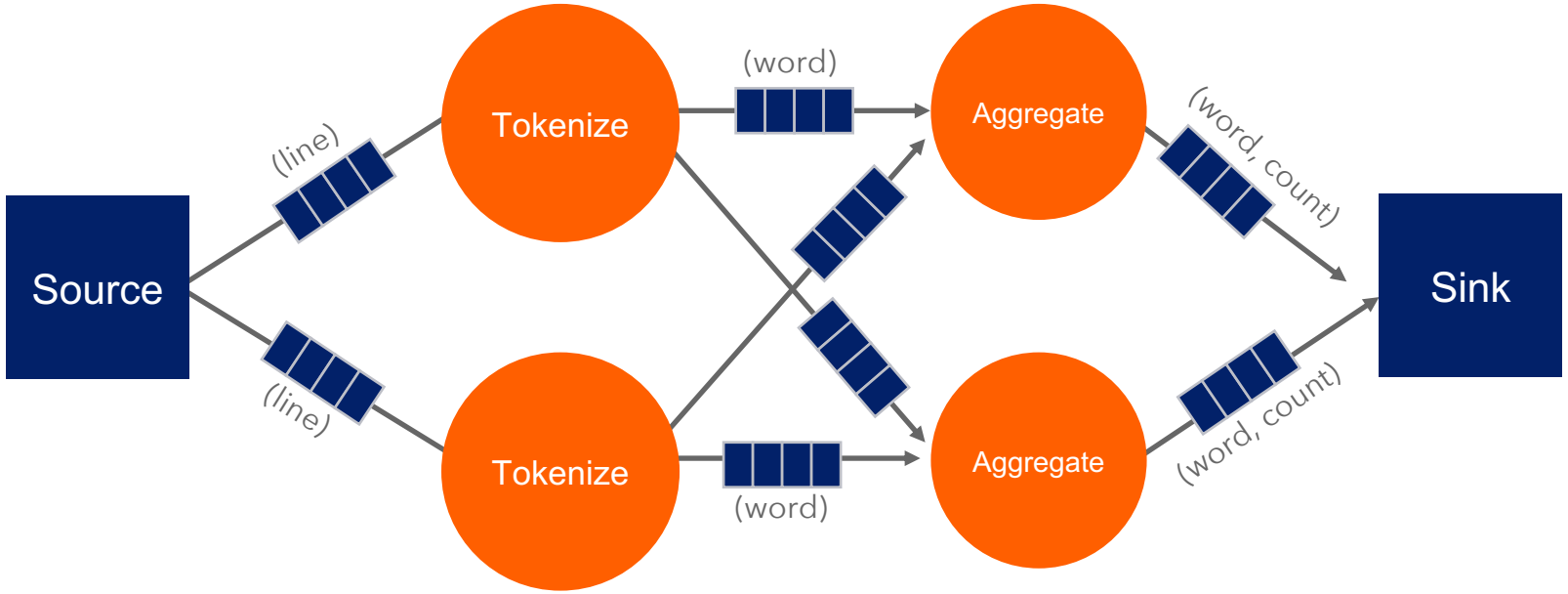
We can parallelise the execution of the vertices by introducing **concurrent queues** between the vertices.



By dividing the text into lines, and having multiple threads, each tokenizing vertex can process a separate line thus parallelizing the work.

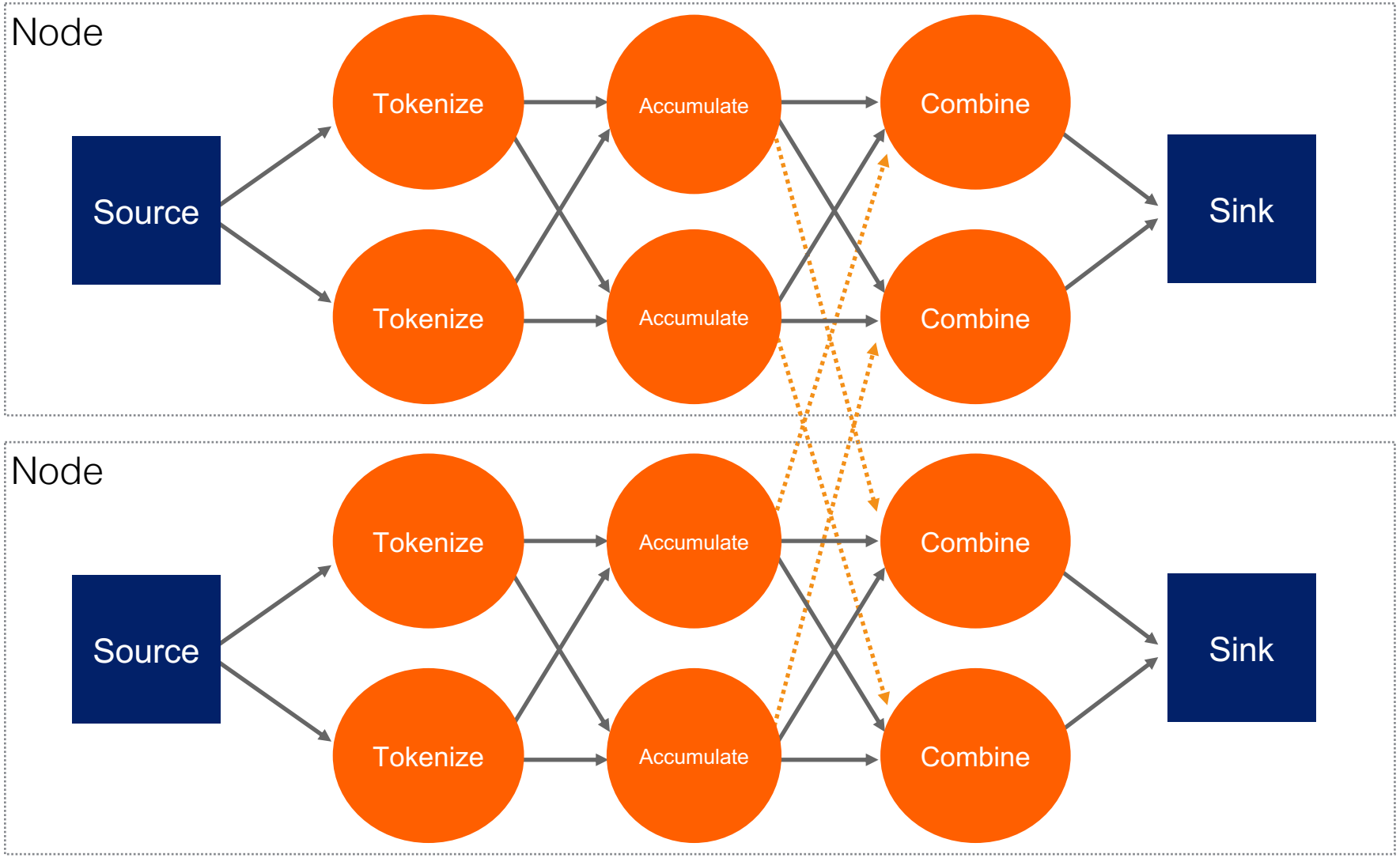


Aggregator can also be executed in parallel by **partitioning** by the individual words.



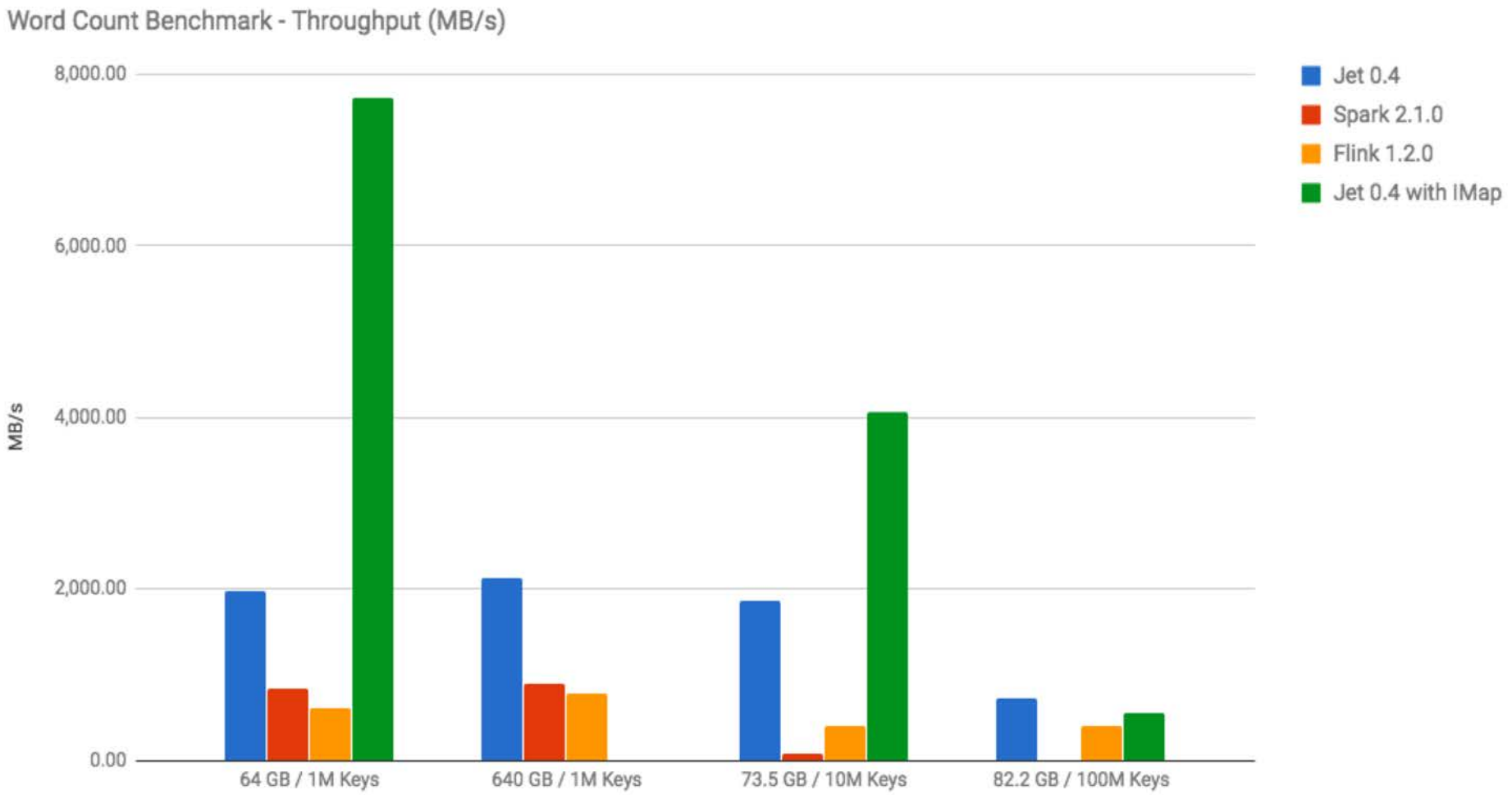
We only need to ensure the **same** words go to the **same** Aggregator.

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

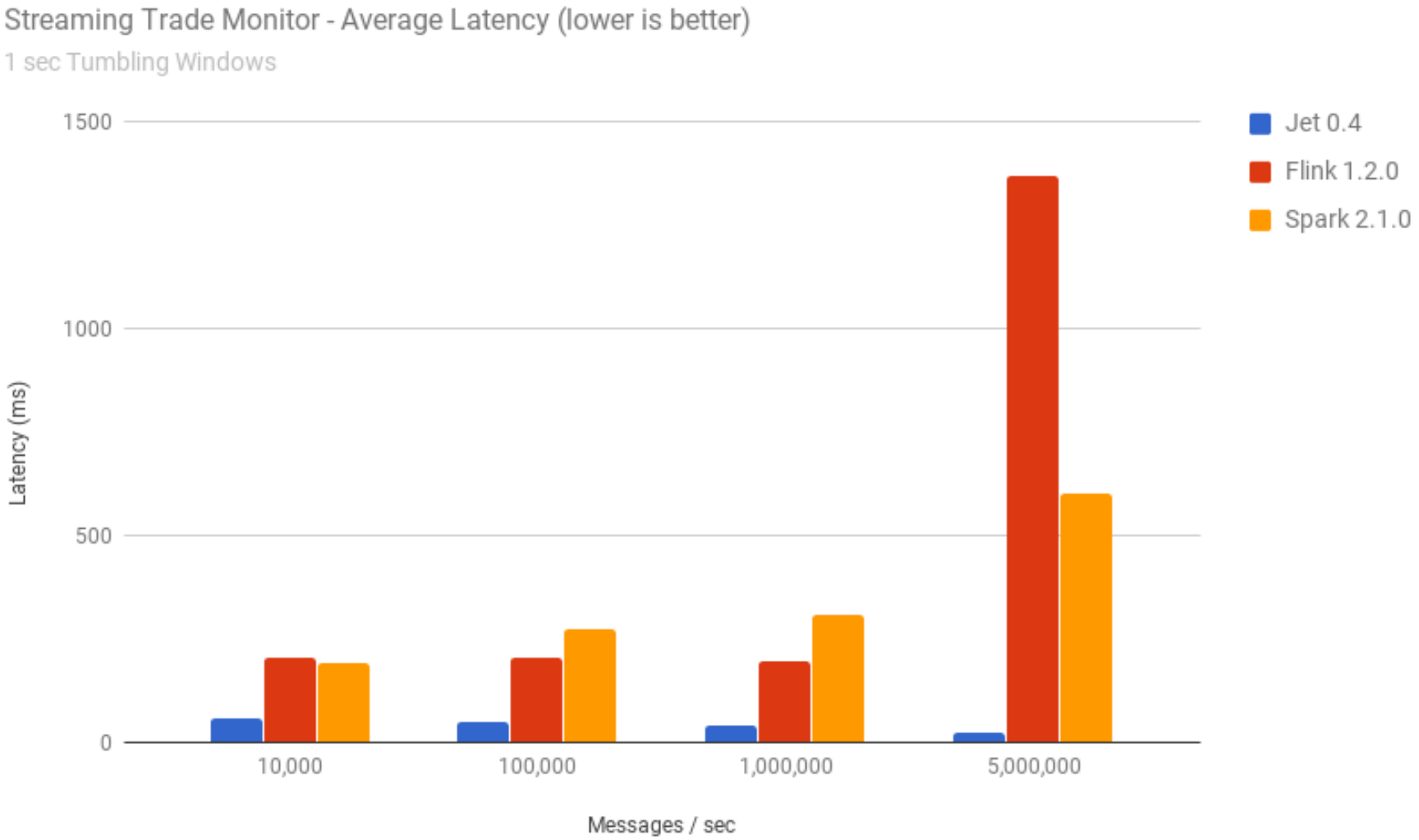


Performance

Throughput



Latency



Hazelcast Jet Architecture

Job Execution

- Pipeline is converted to a DAG with **vertices** and **edges**.
- The graph is distributed to the whole cluster. Each node will run the whole graph
- Each vertex corresponds to one or more **Processor** instances which parallelize the work.
- Starting from source vertices, data flows along the edges, backed by concurrent queues and eventually ends up in one or more sinks.

Processors

- Executes the main business logic for each vertex, typically taking some input and emitting some output
- Can be stateful (aggregation) or stateless (mapping)
- Sources and sinks are also processors, with the difference that they act as initial or terminal vertices.
- Jet provides processors for most common operations which cover most of the use cases: grouping, mapping, filtering

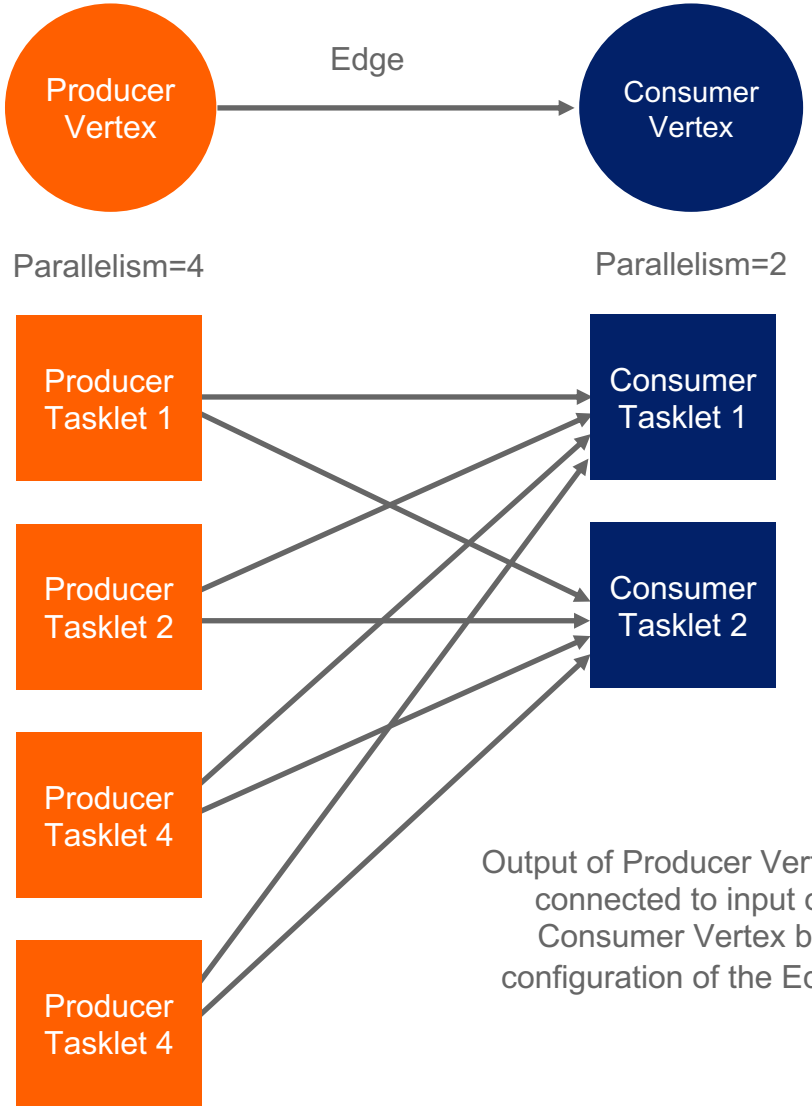
Cooperative Multithreading

- All execution is done through **tasklets**, such as network IO, processors and snapshotting.
- Similar concept to **green threads**
- Tasklets run in a loop serviced by the same native thread.
  - No context switching.
  - Almost guaranteed core affinity
- Each tasklet does **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 backs off

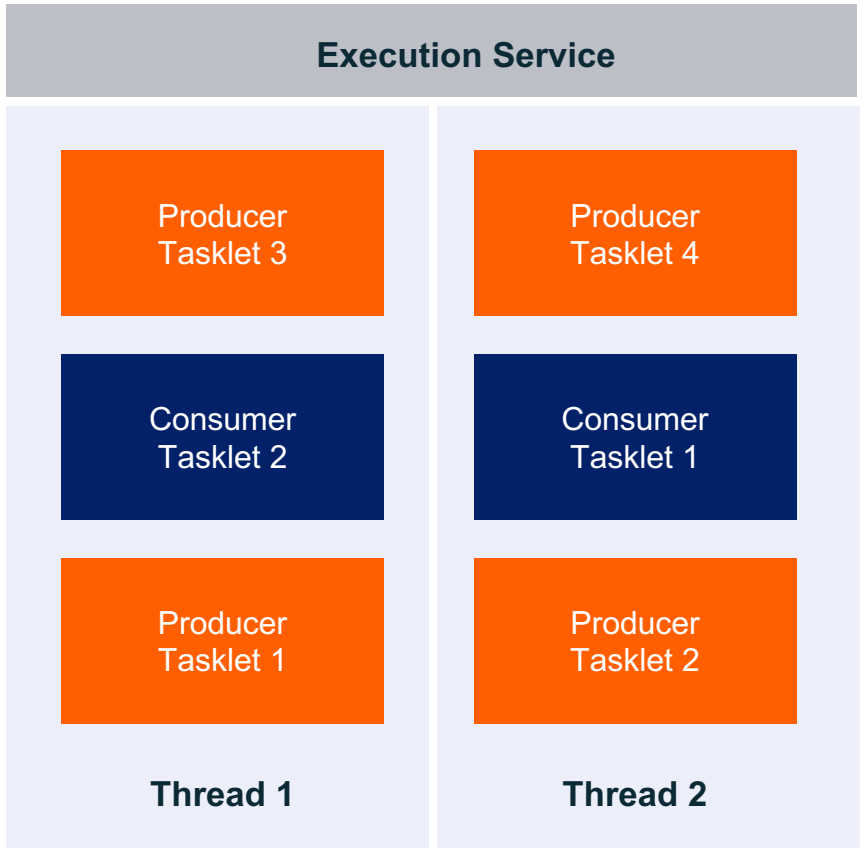
Cooperative Multithreading

- 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 input of Consumer Vertex by configuration of the Edge.



APIs

Pipeline API – High level, Expressive, Type-safe

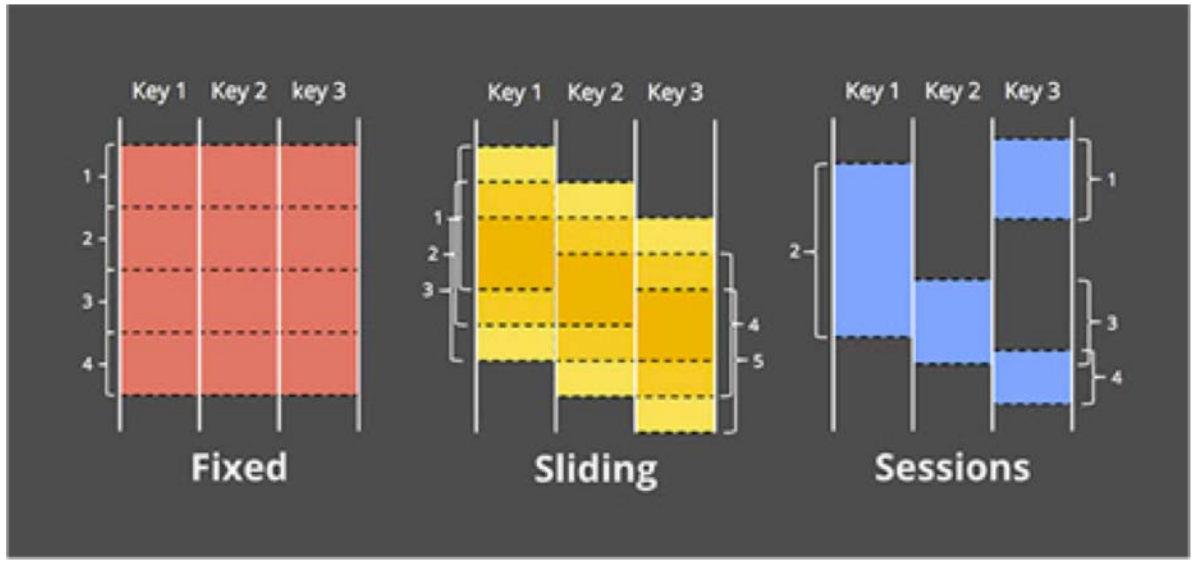
```
Pipeline p = Pipeline.create();
p.drawFrom(Sources.<Long, String>map(“lines”))
 .flatMap(e -> traverseArray(delimiter.split(e.getValue().toLowerCase())))
 .filter(word -> !word.isEmpty())
 .groupingKey(wholeItem())
 .aggregate(counting())
 .drainTo(Sinks.map(“counts”));
```

DAG API – Low level, verbose

```
DAG dag = new DAG();
Vertex source = dag.newVertex("source", readMapP(DOCID_NAME));
Vertex docLines = dag.newVertex("doc-lines",
        nonCooperativeP(flatMapP((Entry<?, String> e) -> traverseStream(docLines(e.getValue()))))
);
Vertex tokenize = dag.newVertex("tokenize",
        flatMapP((String line) -> traverseArray(delimiter.split(line.toLowerCase()))
                .filter(word -> !word.isEmpty()))
);
 dag.edge(between(source, docLines))
          .edge(between(docLines, tokenize))
…
```

Windowing

- Makes it possible to do aggregations on a stream of events
- Count based – every 100 events
- Tumbling – discrete windows – e.g. every minute
- Sliding – overlapping – e.g. every minute, sliding by 1 sec
- Session – dynamically sized – i.e. activity of a single user



Watermarks

- Event time is not the same as wall clock time
- How do you order a disordered stream?
- Time advances only with events, not with wall clock
- You are only as ahead as the stream that is most behind.
- *Allowed lag* controls allowed out of orderness
- What happens to “late events” ?

Fault Tolerance

- Streaming jobs can be running for a long time! How to ensure correctness?
- Hazelcast IMDG provides an in-memory storage which is partitioned and replicated
- Jet uses this storage to keep snapshots of processor states
- If one of the nodes fails, the job can be restarted and the processor can resume where it left off.
- It’s not that simple: you still need to coordinate the states of different steps in the computation: Distributed snapshots

Distributed Snapshots

- Need to coordinate snapshots happening between different vertices in the graph.
- This is achieved by injecting an item into the stream called a *snapshot barrier.*
- Exactly-once vs at-least-once behaviour is determined by how the barrier is treated.

Flight Telemetry

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

Demo Applications

Real-time Image Recognition

Recognizes images present in the webcam video input with a model trained with CIFAR-10 dataset.

Twitter Cryptocurrency Sentiment Analysis

Twitter content is analyzed in real time to calculate cryptocurrency trend list with popularity index.

Real-Time Road Traffic Analysis And Prediction

Continuously computes linear regression models from current traffic. Uses the trend from week ago to predict traffic now

Real-time Sports Betting Engine

This is a simple example of a sports book and is a good introduction to the Pipeline API.
It also uses Hazelcast IMDG as an in-memory data store.

Flight Telemetry

Reads a stream of telemetry data from ADB-S on all commercial aircraft flying anywhere in the world. There is typically 5,000 - 6,000 aircraft at any point in time. This is then filtered, aggregated and certain features are enriched and displayed in Grafana.

Market Data Ingest

Uploads a stream of stock market data (prices) from a Kafka topic into an IMDG map. Data is analyzed as part of the upload process, calculating the moving averages to detect buy/sell indicators. Input data here is manufactured to ensure such indicators exist, but this is easy to reconnect to real input.

Markov Chain Generator

Generates a Markov Chain with probabilities based on supplied classical books.

Real-Time Trade Processing

Processes immutable events from an event bus (Kafka) to update storage optimized for querying and reading (IMDG).

Hazelcast Jet

- 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*

Roadmap

2018 Jet Roadmap Highlights

Features	Description
Management Center	Management and monitoring features for Jet.
Job Elasticity	Scaling up running jobs automatically
Rolling Job Upgrades	Make changes to running jobs and restart them
Kotlin API	Coroutines support

Questions?

Version 0.6.1 is the current release with 0.7 coming in October
aiming for 1.0 this year

http://jet.hazelcast.org

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