Fast and Easy Stream Processing

Scott McMahon | scott@hazelcast.com | St. Louis JUG | 10 Jan 2019

© 2018 Hazelcast Inc.

What is stream processing?

Much of our modern data is received as discreet events.

- Sensor Readings
- Stock Ticker Values
- User Clicks

These events are received as they happen and they are infinite, they never stop.

Stream Processing is the method of continuously receiving these events and acting on them, individually, in as near “real time” as possible

© 2018 Hazelcast Inc.

Why stream processing?

Latency

Processing events as they happen reduces the latency to ‘right now’ – the time from observing an event to acting on it.

The value of much of this data diminishes with time. Ex. GPS

Data Volume

Todays environment produces too much data to store and process later.

Processing as received reduces storage requirements.

© 2018 Hazelcast Inc.

Streaming Evolution

© 2018 Hazelcast Inc.

How Did We Get Here?

Good old daily batch jobs can actually be thought of as a stream, except with a 24 hour latency.

What if we shorten the interval to 4 hours? Better but still a high latency, especially with high event rates (100's k/second).

Micro batches lowered latency to just a few minutes.

However, these systems weren't designed for this.

The overhead associated with the design of these systems prevented much further progress.

Is this really stream processing at all? Are we processing the events, or just loading them?

New technology was needed to do the job that needed to be done.

© 2018 Hazelcast Inc.

Enterprise Service Bus (ESB)

Late 90’s, early 2000’s - EAI

First Message Based Designs

Decoupling sender and receiver

Event Driven Architecture

ESB Handles Messages

Queues and Topics implement Pub/Sub architecture

Store Messages/Events in DB for historical analysis

Tibco, Mule, webMethods, BizTalk

© 2018 Hazelcast Inc.

Complex Event Processing (CEP)

ESB's are now handling much higher volumes

Faster reaction to 'real time' events is an advantage

Pre-Define related event patterns to detect

Emit a new CEP event for subscribed processors

Not much analytics, more reactionary

Apama, Infosphere, Tibco

New technologies begin to emerge:

Apache Hadoop, Machine Learning, Artificial Intelligence

© 2018 Hazelcast Inc.

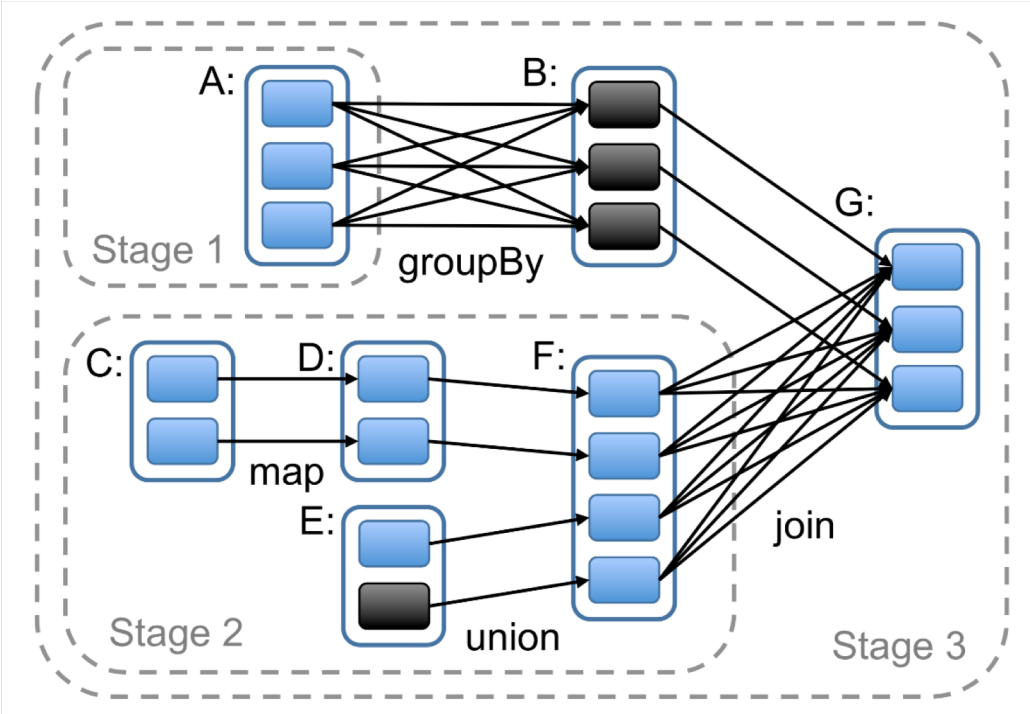
Hadoop with MapReduce – Big Batch Jobs ?

- 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

© 2018 Hazelcast Inc.

Apache Spark – Big Analytics ?

- 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

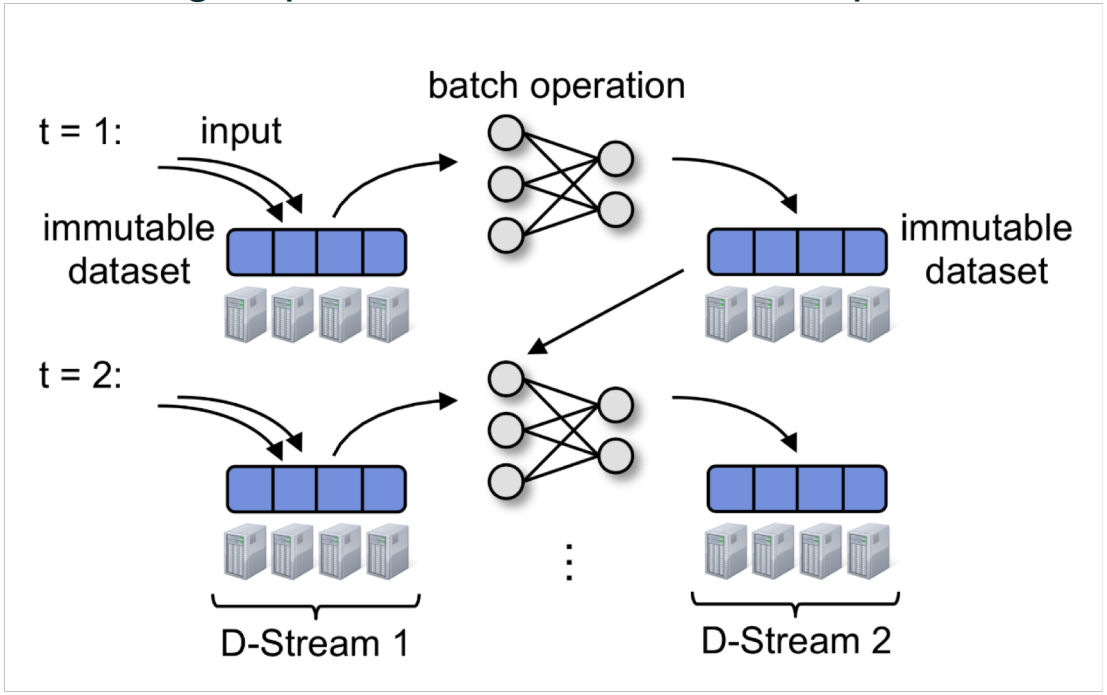


http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf

© 2018 Hazelcast Inc.

Spark Streaming – Big Micro Batch ?

- Fault-Tolerance of Spark designed 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*.

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

© 2018 Hazelcast Inc.

Today's Continuous Stream Processing

- Uses RAM for storage, not Disk
- Usually DAG based procssing
- Individual events, not micro-batch
- Implements batch is as streaming with bounds
- Learns from previous systems (DAG, Windowing, etc.)
- Significant academic research in the last decade - [1][2]
- Hazelcast Jet, Apache Storm, Twitter Heron, Apache Flink, Kafka Streams, Google DataFlow

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

© 2018 Hazelcast Inc.

What does a Stream Processing Engine 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.

© 2018 Hazelcast Inc.

When to Use Stream Processing

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

© 2018 Hazelcast Inc.

DISTRIBUTED COMPUTING. SIMPLIFIED.

What is Hazelcast Jet?

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

© 2018 Hazelcast Inc.

Distributed computing. Simplified.

HAZELCAST IMDG is an **operational, in-memory**, distributed computing platform that manages data using in-memory storage, and performs parallel execution for breakthrough application speed and scale.

HAZELCAST JET is the ultra fast, application embeddable, 3rd generation stream processing engine for low latency batch and stream processing.

© 2018 Hazelcast Inc.

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");
    }
}
```

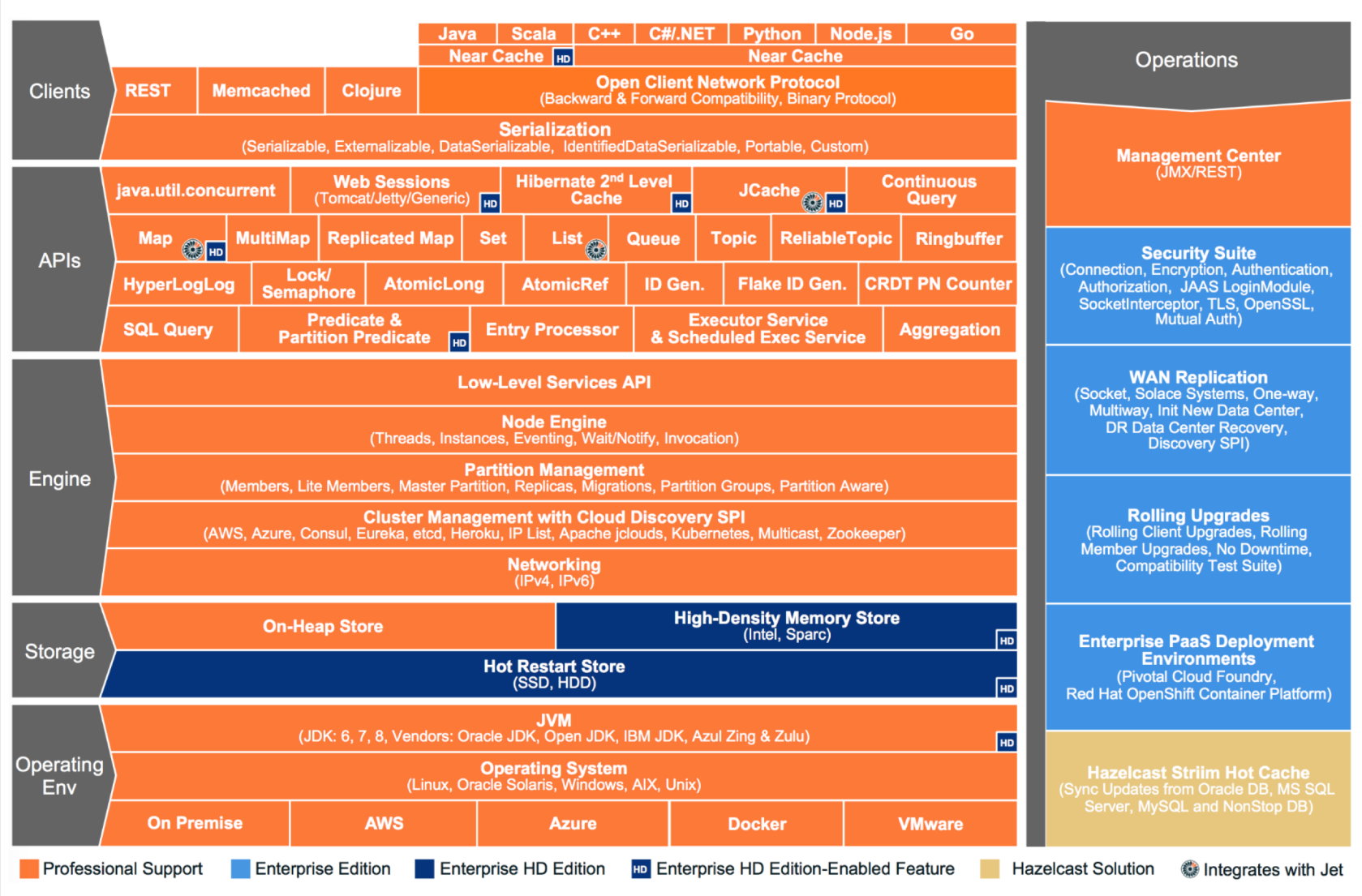

© 2018 Hazelcast Inc.

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

© 2018 Hazelcast Inc.

Hazelcast IMDG



© 2018 Hazelcast Inc.

How are Event Streams and Databases Related?

A database is a snapshot of an event stream at a particular time

The event stream contains all the changes the data went through to get to a particular point in time

Most databases actually implement a single purpose event stream in the form of the write ahead log (WAL)

Events are immutable and the database is mutable

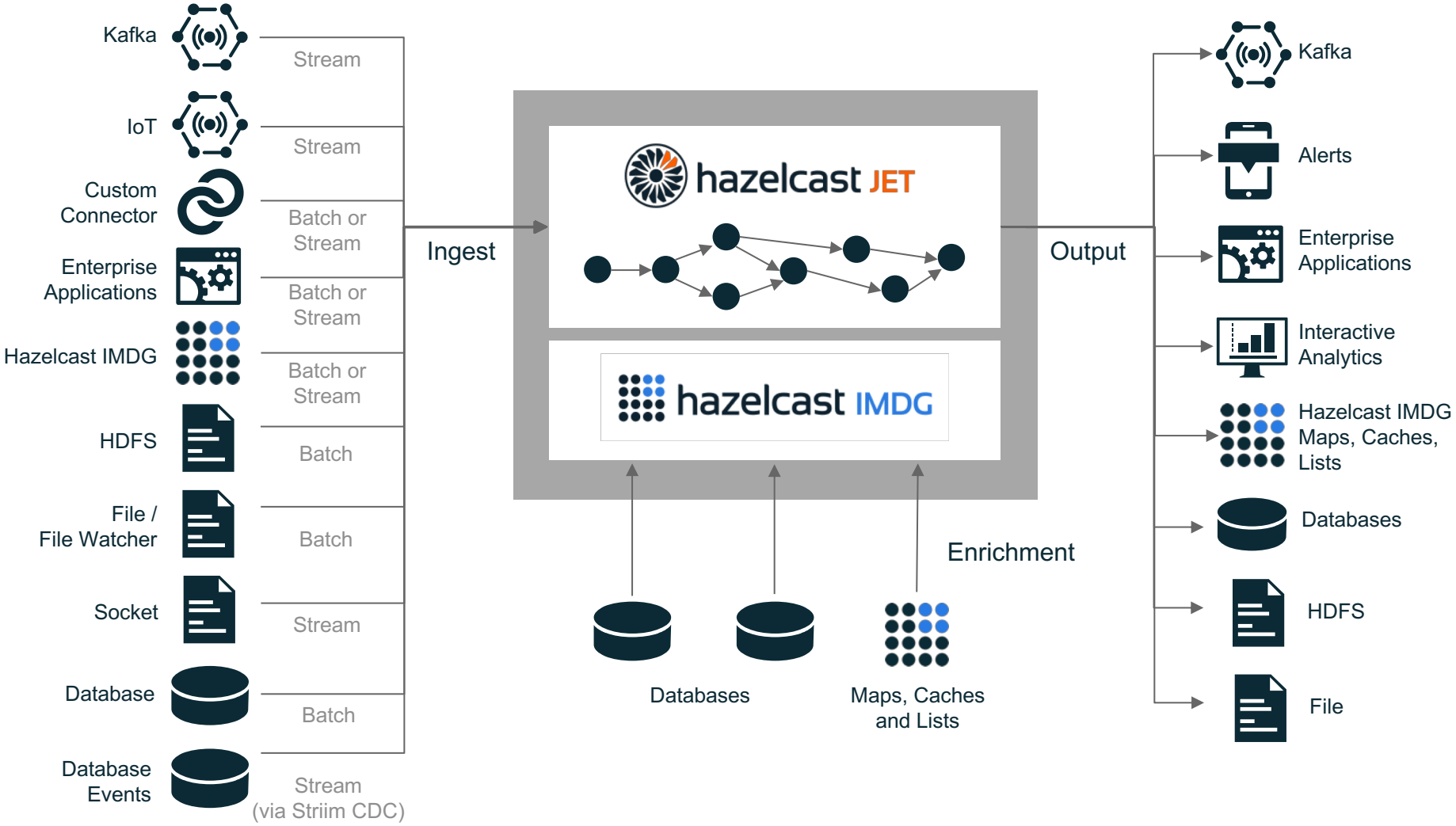
Streams and databases are two sides of a coin

There is additional data in how the data changes over time

We need to actually process each event, not just load it.

© 2018 Hazelcast Inc.

Stream and Batch Processing



© 2018 Hazelcast Inc.

Hazelcast Jet
Stream Processing Platform

Clients
- **Java**
- **Open Client Network Protocol** (Backward & Forward Compatibility, Binary Protocol)

High-Level APIs
- **Pipeline API** (Stream and Batch Processing)
- **java.util.stream** (Batch Processing)

Connectors
- **Streaming Readers and Writers** (IMap & ICache streamer, Kafka, FileWatcher, Socket)
- **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)

hazelcast IMDG
- **Data Structures** (Distributed Map, Cache, List)
- **Node Engine** (Threads, Instances, Eventing, Wait/Notify, Invocation)
- **Partition Management** (Members, Lite Members, Master Partition, Replicas, Migrations, Partition Groups, Partition Aware)
- **Cluster Management with Cloud Discovery SPI** (AWS, Azure, Consul, Eureka, etcd, Heroku, IP List, Apache jclouds, Kubernetes, Multicast, Zookeeper)
- **Networking** (IPv4, IPv6)

Operating Environments
- **JVM** (JDK: 8, 9, 10 Vendors: Oracle JDK, Open JDK, IBM JDK, Azul Zing & Zulu)
- **Operating System** (Linux, Oracle Solaris, Windows, AIX, Unix)
- **On Premise** | **AWS** | **Azure** | **Docker** | **VMware**

Operations

Jet Management Center
(JMX/REST)

Security Suite
(Connection, Encryption, Authentication, Authorization, JAAS LoginModule, SocketInterceptor, TLS, OpenSSL, Mutual Auth)

Lossless Recovery (0.8)
(Resume processing without data loss after lights out event)

Rolling Job Upgrade (0.8)
(Allow jobs to be upgraded without data loss or interruption)

Enterprise PaaS Deployment Environments
(Pivotal Cloud Foundry, Red Hat Openshift Container Platform (0.8))

Professional Support | Enterprise Edition

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

© 2018 Hazelcast Inc.

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

© 2018 Hazelcast Inc.

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

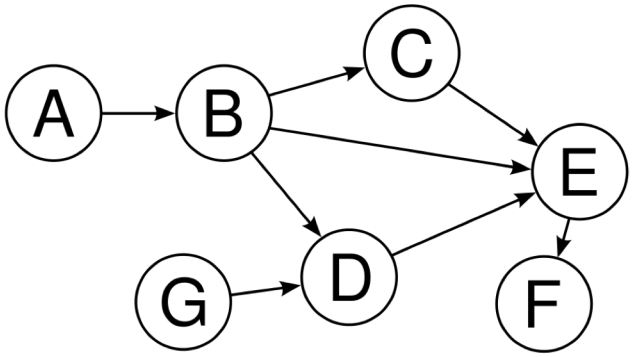
© 2018 Hazelcast Inc.

DAG Based Processing

© 2018 Hazelcast Inc.

DAG Based Processing

- Directed Acyclic Graphs are used to model computations



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

© 2018 Hazelcast Inc.

Example: Word Count

If we lived in a single-threaded world:

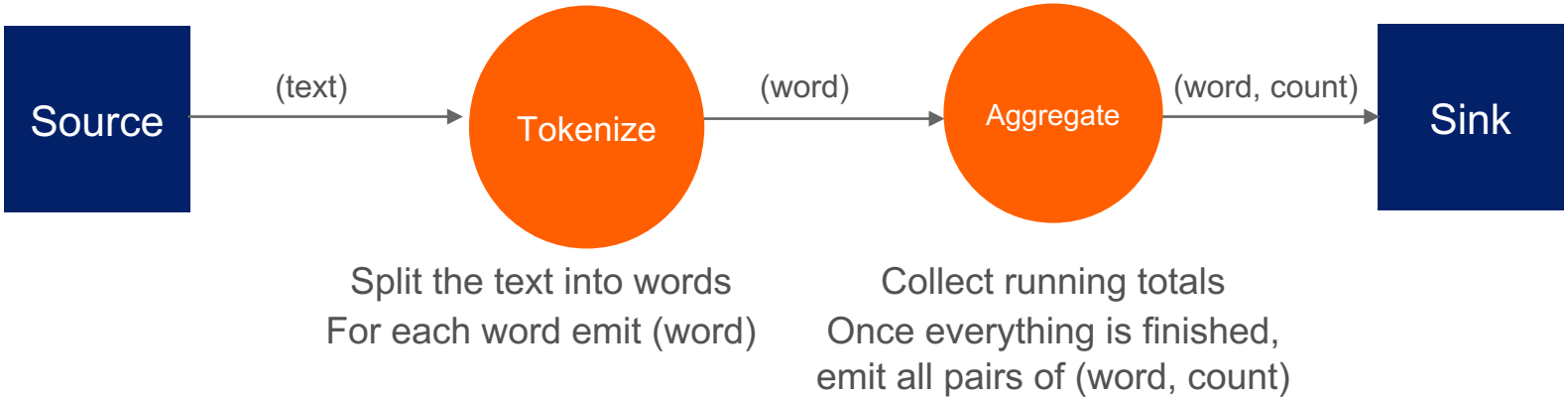
1. Iterate through all the lines
2. Split the line into words
3. Update running total of counts with each word

```
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);
}
```

© 2018 Hazelcast Inc.

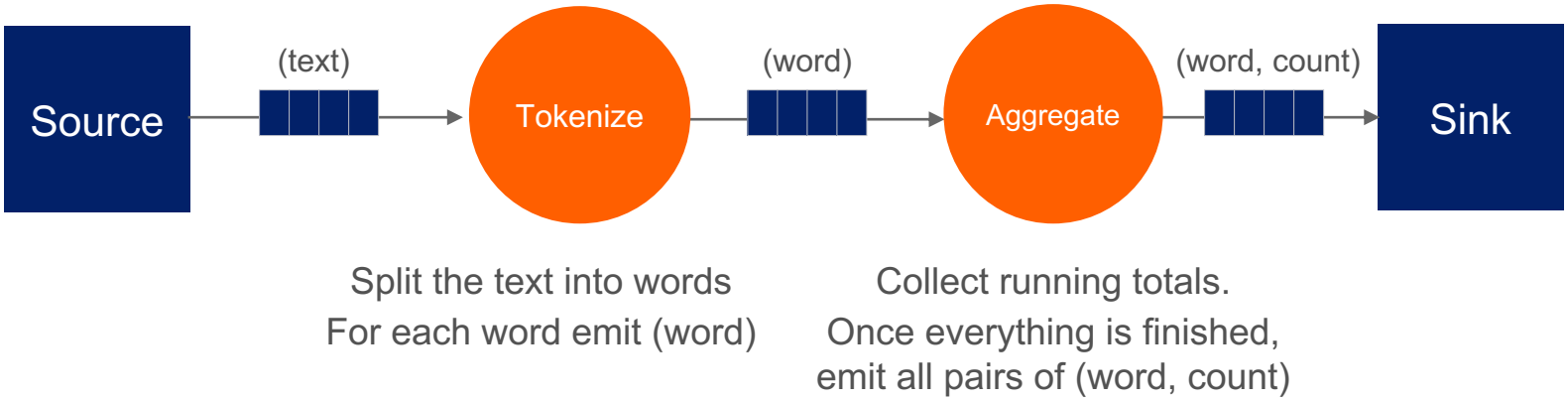
We can represent the computation as a **DAG**



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

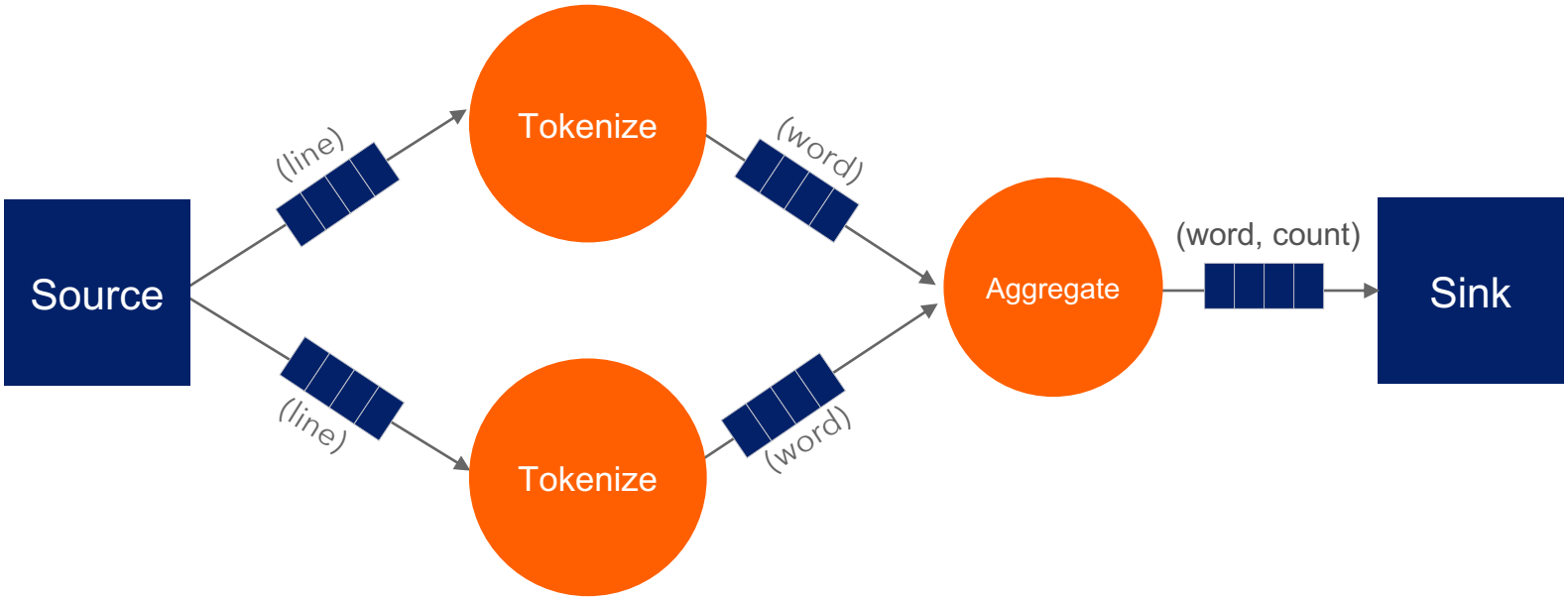
© 2018 Hazelcast Inc.

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



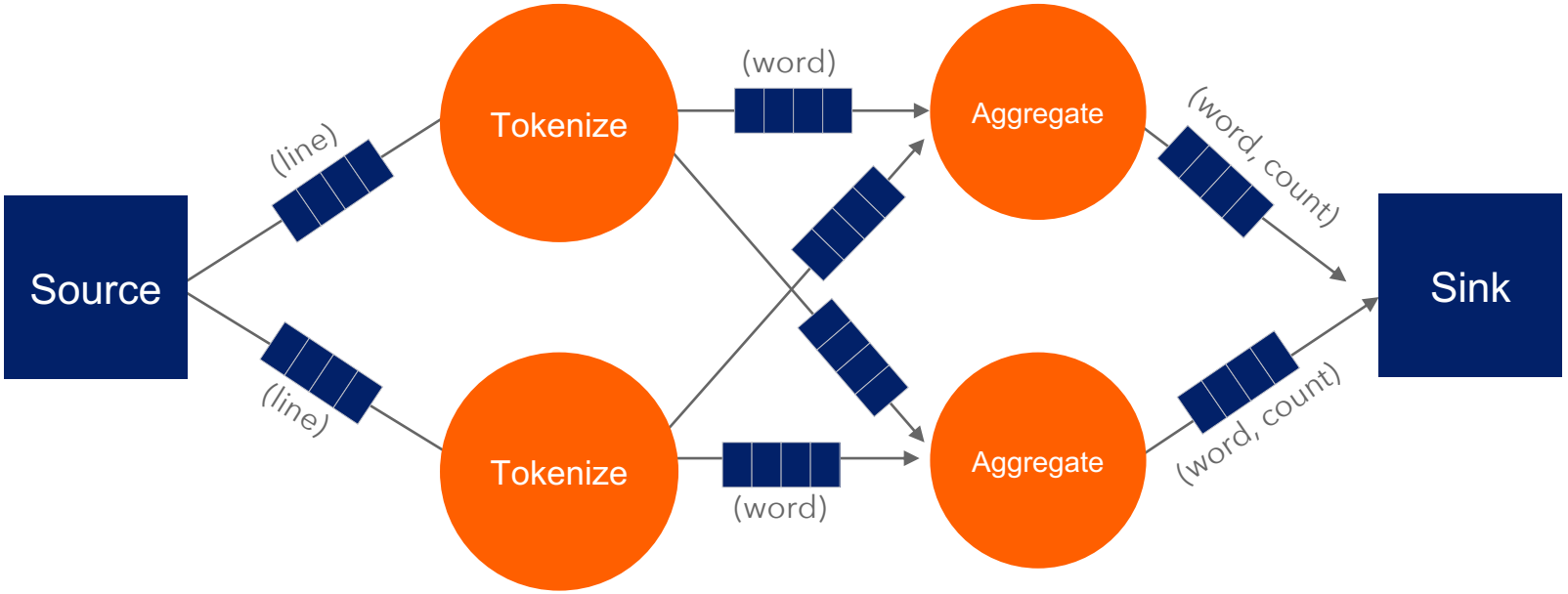
© 2018 Hazelcast Inc.

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



© 2018 Hazelcast Inc.

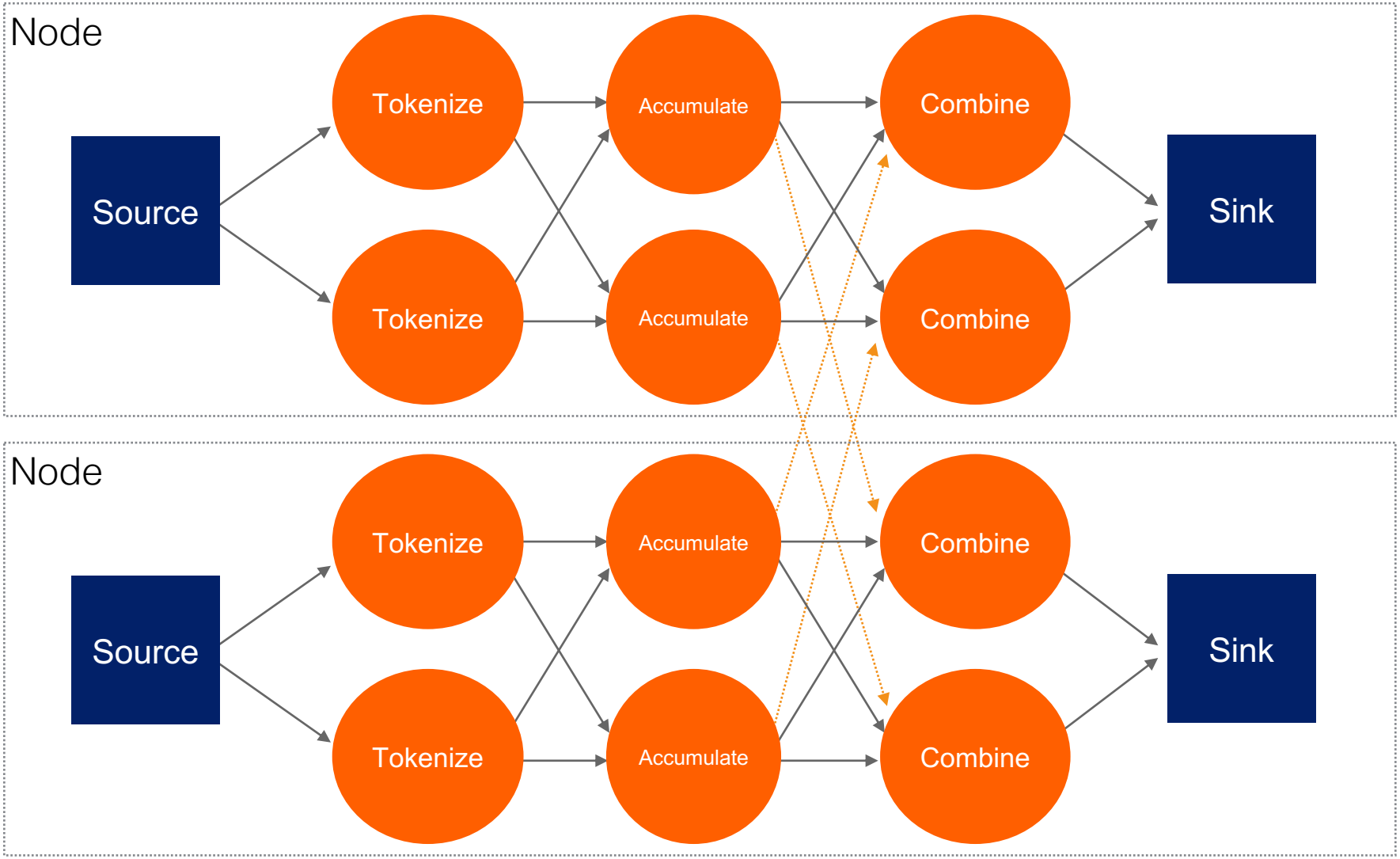
Aggregator can also be executed in parallel by **partitioning** by the individual words.



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

© 2018 Hazelcast Inc.

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



© 2018 Hazelcast Inc.

Flight Telemetry

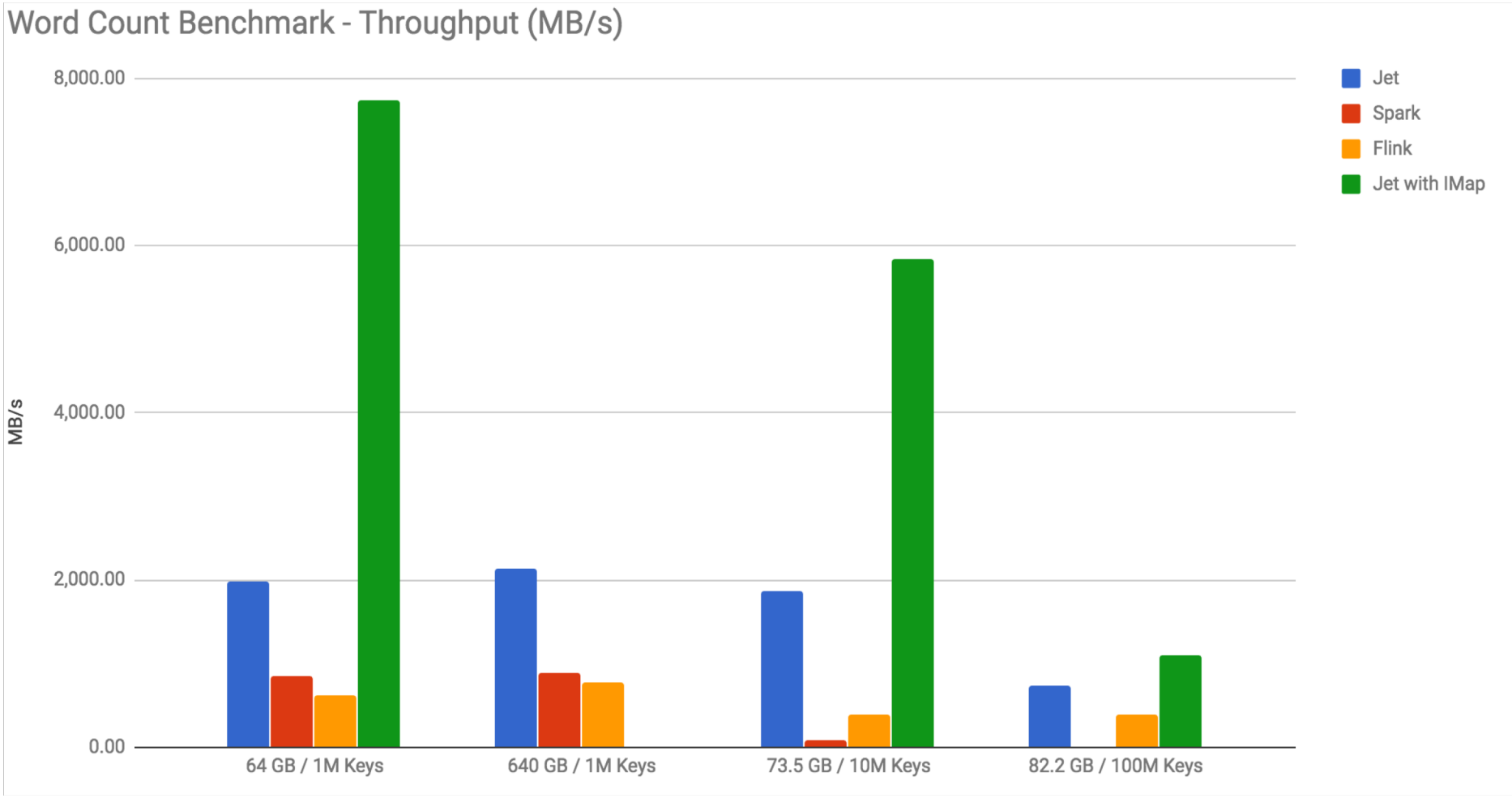
https://github.com/hazelcast/hazelcast-jet-demos

© 2018 Hazelcast Inc.

Performance

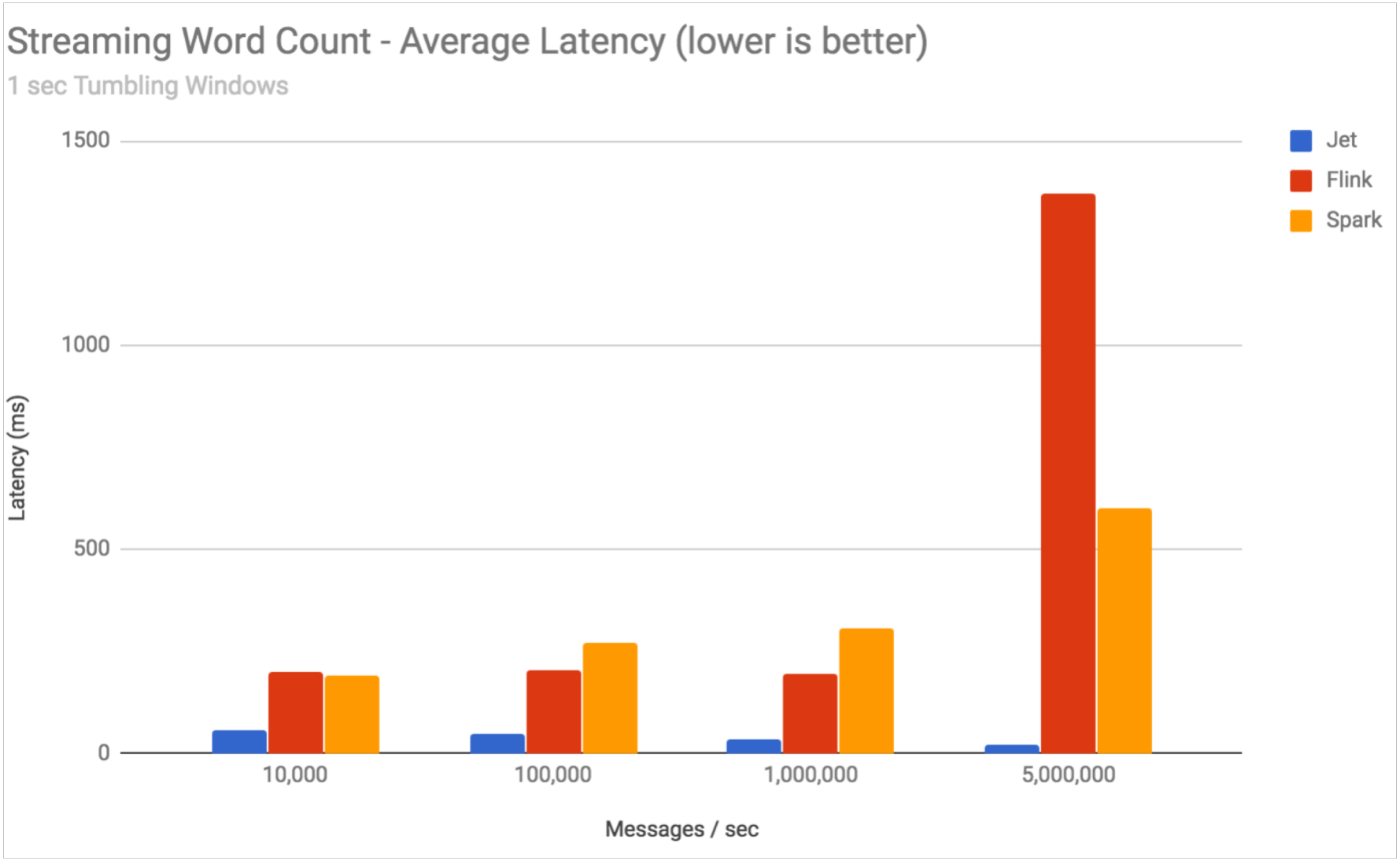
© 2018 Hazelcast Inc.

Throughput



© 2018 Hazelcast Inc.

Latency



© 2018 Hazelcast Inc.

Hazelcast Jet Architecture

© 2018 Hazelcast Inc.

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.

© 2018 Hazelcast Inc.

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

© 2018 Hazelcast Inc.

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

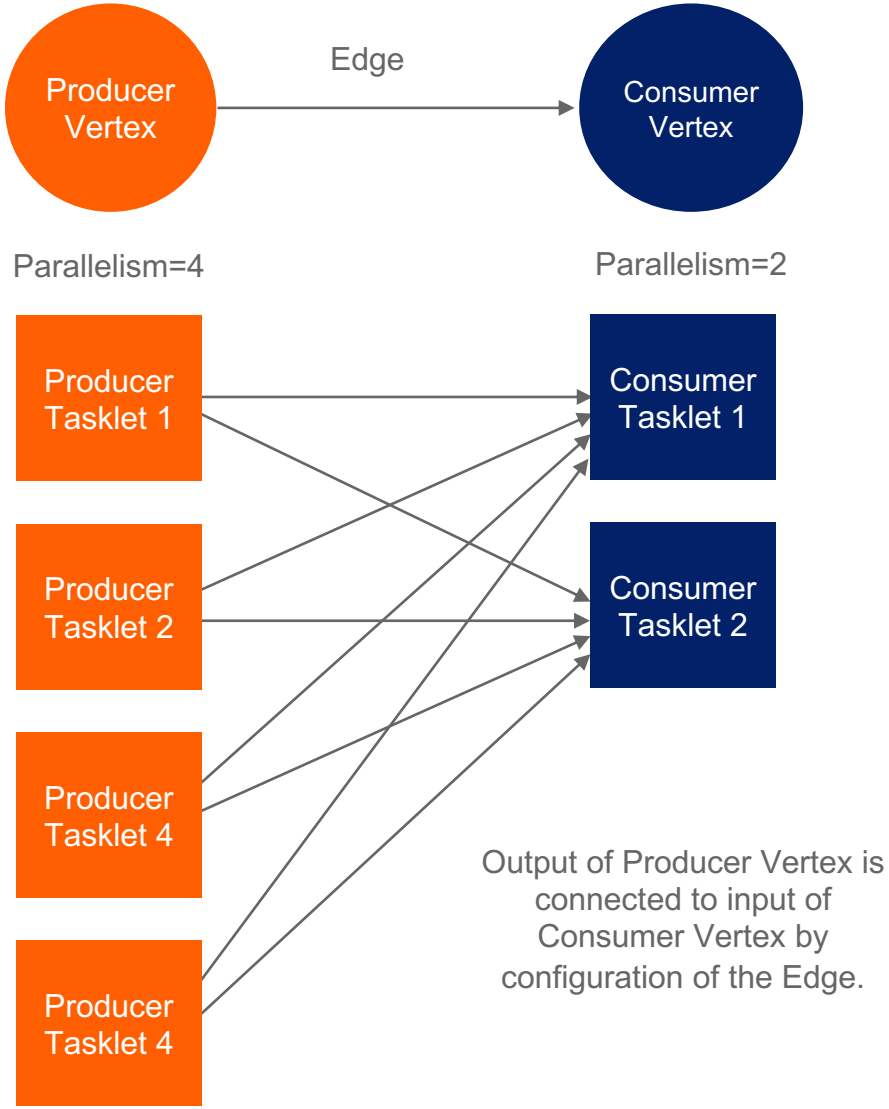
© 2018 Hazelcast Inc.

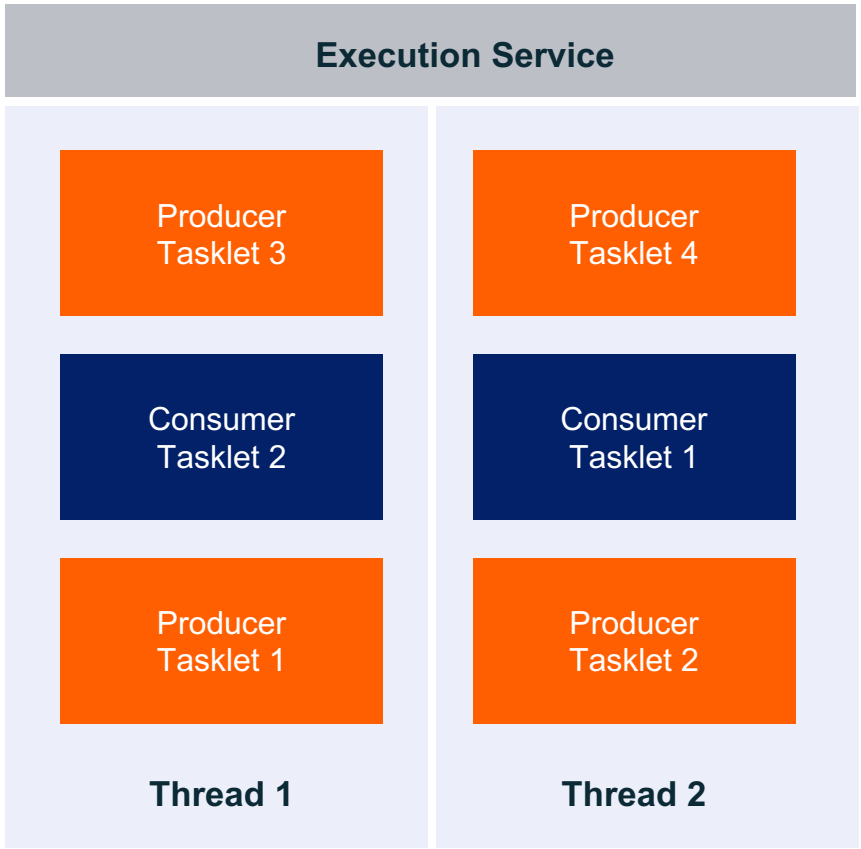
Cooperative Multithreading

- 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.

© 2018 Hazelcast Inc.

Tasklets – Unit of Execution





© 2018 Hazelcast Inc.

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"));
```

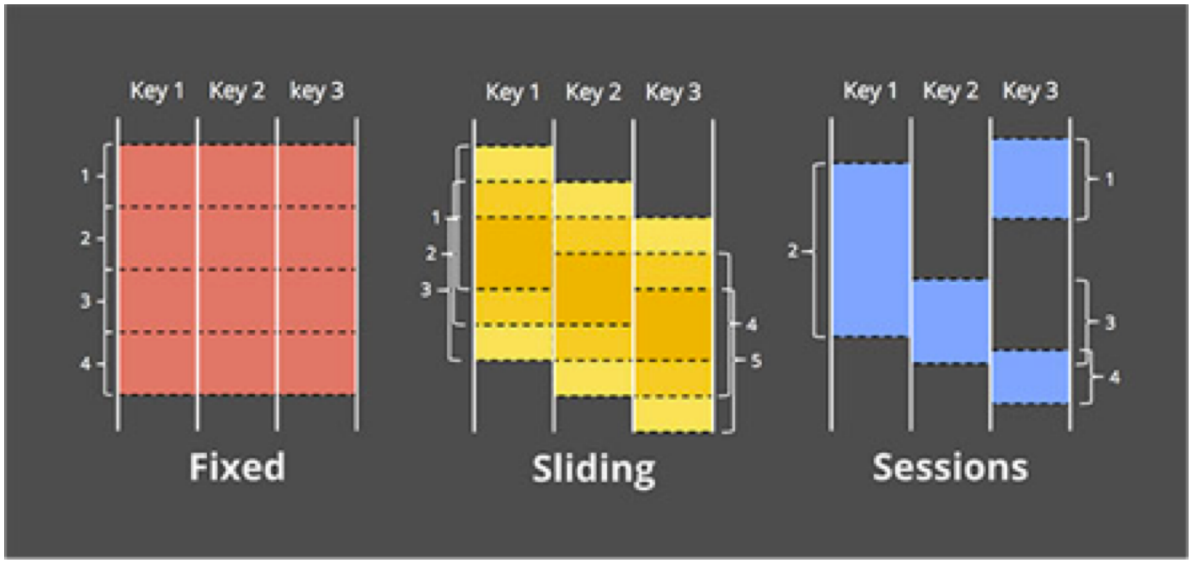
DAG API – Low level, verbose

```
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))
…
```

© 2018 Hazelcast Inc.

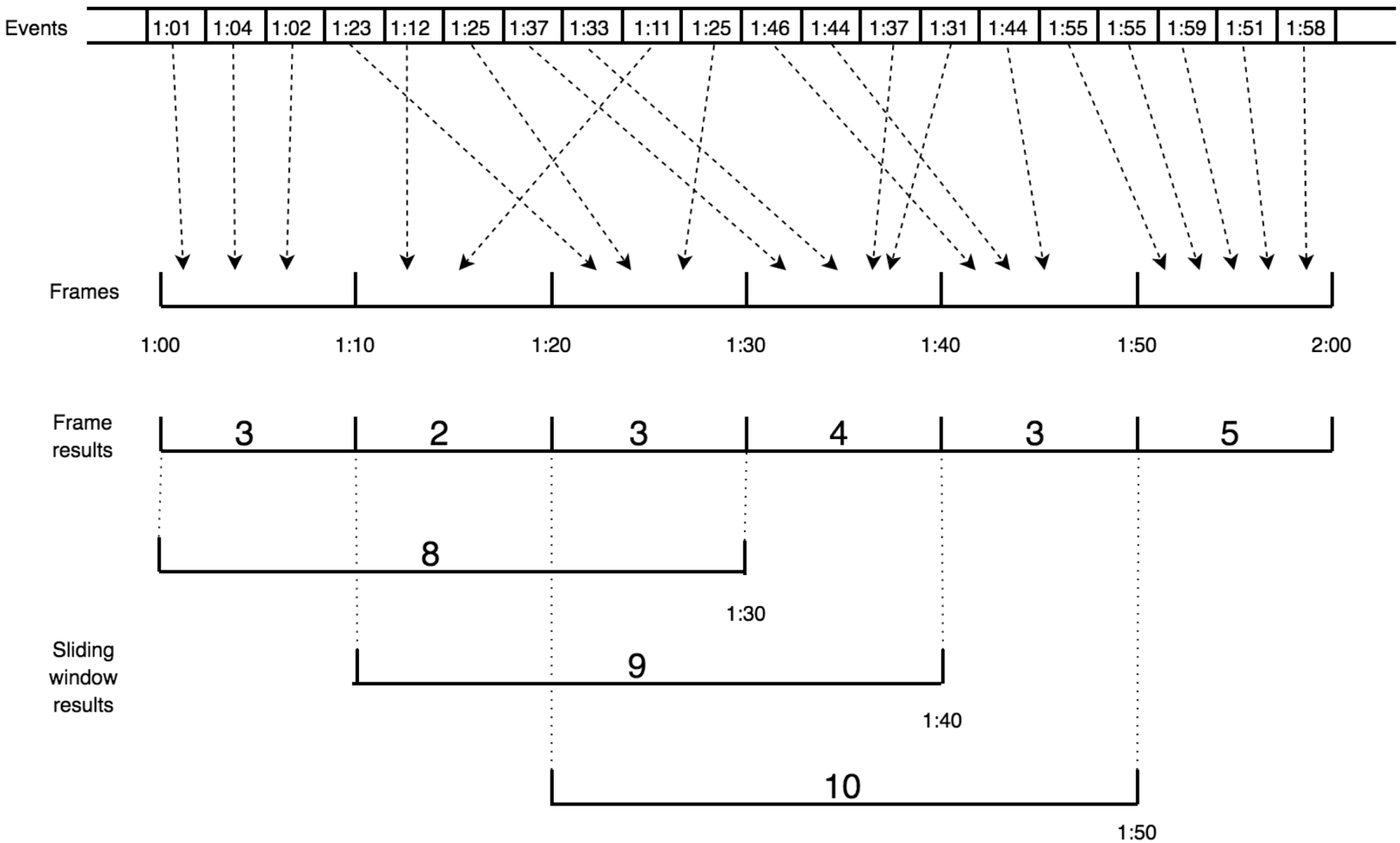
Windowing

- Makes it possible to do aggregations on a stream of events
- Count based – e.g. 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



© 2018 Hazelcast Inc.

Aggregation with Sliding Windows



© 2018 Hazelcast Inc.

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” ?

© 2018 Hazelcast Inc.

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

© 2018 Hazelcast Inc.

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 behavior is determined by how the barrier is treated.

© 2018 Hazelcast Inc.

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).

© 2018 Hazelcast Inc.

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*

© 2018 Hazelcast Inc.

Questions?

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

http://jet.hazelcast.org

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

© 2018 Hazelcast Inc.