Hazelcast Jet Reference Manual

Table of Contents

Welcome to the Hazelcast Jet Reference Manual. This manual includes concepts, instructions, and samples to guide you on how to use Hazelcast Jet to build applications.

Hazelcast Jet is a distributed computing platform. It supports both finite datasets (batch processing) and infinite streams with continuous operators. With Hazelcast In-Memory Data Grid (IMDG) providing storage functionality, Hazelcast Jet performs data-local parallel execution, enabling data-intensive applications to operate in real time. Using directed acyclic graphs (DAG) to model relationships between individual steps in the data processing pipeline, Hazelcast Jet is simple to deploy. Hazelcast Jet is an Apache 2 licensed open source project. Please also refer to the [Hazelcast Jet website](http://jet.hazelcast.org/) for information on its features, use cases and performance considerations.

As the reader of this manual, you must be familiar with the Java programming language and you should have installed your preferred Integrated Development Environment (IDE).

Preface

Naming

- **Hazelcast Jet** or **Jet** both refer to the same distributed data processing engine provided by Hazelcast, Inc.
- **Hazelcast** or **Hazelcast IMDG** both refer to Hazelcast in-memory data grid middleware. **Hazelcast** is also the name of the company (Hazelcast, Inc.) providing Hazelcast IMDG and Hazelcast Jet.

Licensing

Trademarks

Getting Help

The Jet team provides support to its community via these channels:

- Stack Overflow (ask a question on how to use Jet properly and troubleshoot your setup)
- Hazelcast Jet mailing list (propose features and discuss your ideas with the team)
- GitHub's issue tracking (report your confirmed issues)

For information on the commercial support for Hazelcast Jet, please see this page on hazelcast.com.

Chapter 1. Introduction

Hazelcast Jet is a distributed data processing engine, built for high-performance batch and stream processing. It reuses some features and services of Hazelcast In-Memory Data Grid (IMDG), but is otherwise a separate product with features not available in the IMDG.

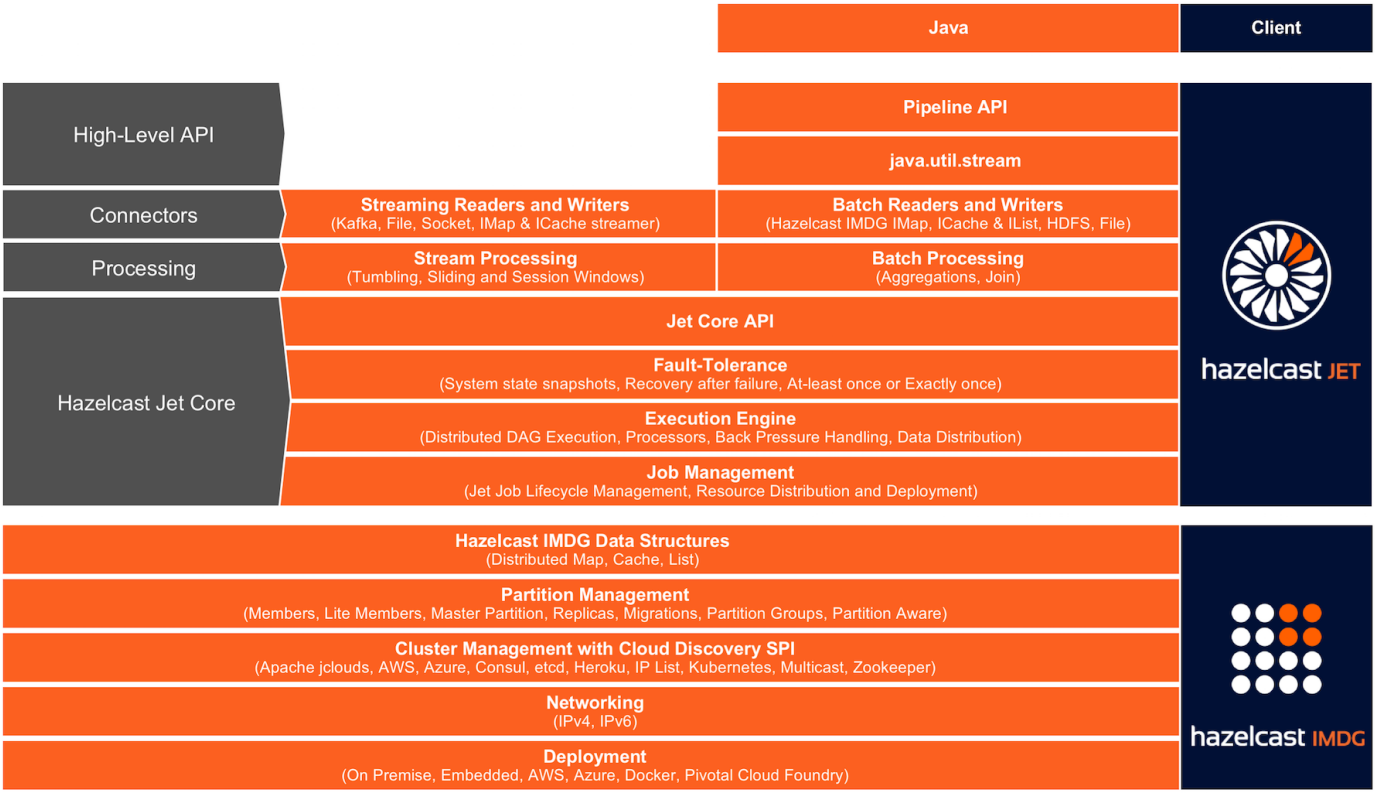
In addition to its own Pipeline API, Jet also offers a distributed implementation of java.util.stream. You can express your computation over any data source Jet supports using the familiar API from the JDK 8.

With Hazelcast's IMDG providing storage functionality, Hazelcast Jet performs parallel execution to enable data-intensive applications to operate in near real-time. Using directed acyclic graphs (DAG) to model relationships between individual steps in the data processing pipeline, Hazelcast Jet can execute both batch and stream-based data processing applications. Jet handles the parallel execution using the *green thread* approach to optimize the utilization of the computing resources.

Breakthrough application speed is achieved by keeping both the computation and data storage in memory. The embedded Hazelcast IMDG provides elastic in-memory storage and is a great tool for storing the results of a computation or as a cache for datasets to be used during the computation. Extremely low end-to-end latencies can be achieved this way.

It is extremely simple to use - in particular, Jet can be fully embedded for OEMs and for Microservices – making it is easier for manufacturers to build and maintain next generation systems. Also, Jet uses Hazelcast discovery for finding the members in the cluster, which can be used in both on-premise and cloud environments.

1.1. Architecture Overview



1.2. The Data Processing Model

Hazelcast Jet provides high performance in-memory data processing by modeling the computation as a *Directed Acyclic Graph (DAG)* where vertices represent computation and edges represent data connections. A vertex receives data from its inbound edges, performs a step in the computation, and emits data to its outbound edges. A single vertex's computation work is performed in parallel by many instances of the `Processor` type around the cluster.

One of the major reasons to divide the full computation task into several vertices is *data partitioning*: the ability to split the data stream traveling over an edge into slices which can be processed independently of each other. To make this work, a function must be defined which computes the *partitioning key* for each item and makes all related items map to the same key. The computation engine can then route all such items to the same processor instance. This makes it easy to parallelize the computation: each processor will have the full picture for its slice of the entire stream.

Edges determine how the data is routed from individual source processors to individual destination processors. Different edge properties offer precise control over the flow of data.

1.3. Clustering and Discovery

Hazelcast Jet typically runs on several machines that form a cluster but it may also run on a single JVM for testing purposes. There are several ways to configure the members for discovery, explained in detail in the Hazelcast IMDG Reference Manual.

1.4. Members and Clients

A Hazelcast Jet *instance* is a unit where the processing takes place. There can be multiple instances per JVM, however this only makes sense for testing. An instance becomes a *member* of a cluster: it can join and leave clusters multiple times during its lifetime. Any instance can be used to access a cluster, giving an appearance that the entire cluster is available locally.

On the other hand, a *client instance* is just an accessor to a cluster and no processing takes place in it.

1.5. Relationship with Hazelcast IMDG

Hazelcast Jet leans on Hazelcast IMDG for cluster formation and maintenance, data partitioning, and networking. For more information on Hazelcast IMDG, see the latest Hazelcast Reference Manual.

As Jet is built on top of the Hazelcast platform, there is a tight integration between Jet and IMDG. A Jet job is implemented as a Hazelcast IMDG proxy, similar to the other services and data structures in Hazelcast. The Hazelcast Operations are used for different actions that can be performed on a job. Jet can also be used with the Hazelcast Client, which uses the Hazelcast Open Binary Protocol to communicate different actions to the server instance.

1.5.1. Reading from and Writing to Hazelcast Distributed Data Structures

Jet embeds Hazelcast IMDG. Therefore, Jet can use Hazelcast IMDG maps, caches and lists on the embedded cluster as sources and sinks of data and make use of data locality. A Hazelcast `IMap` or `ICache` is distributed by partitions across the cluster and Jet members are able to efficiently read from the Map or Cache by having each member read just its local partitions. Since the whole `IList` is stored on a single partition, all the data will be read on the single member that owns that partition. When using a map, cache or list as a Sink, it is not possible to directly make use of data locality because the emitted key-value pair may belong to a non-local partition. In this case the pair must be transmitted over the network to the member which owns that particular partition.

Jet can also use any remote Hazelcast IMDG instance via Hazelcast IMDG connector.

1.6. High Availability and Fault Tolerance

Jet provides highly available and fault tolerant distributed computation. If one of the cluster members fails and leaves the cluster during job execution, the job restarted on the remaining members automatically and transparently. Jet achieves this by maintaining in-memory copies of the job metadata inside the cluster. The user does not need to designate any node as the master.

In case of a failure, a batch job can typically just be restarted from beginning as the data can easily be replayed. For streaming jobs that run continuously, this might not be possible so the engine must be able to detect a failure, recover from it, and resume processing without data loss.

Jet achieves fault tolerance in streaming jobs by making a snapshot of the internal processing state at

regular intervals. If a member of the cluster fails while a job is running, Hazelcast Jet will detect this and restart the job on the new cluster topology. It will restore its internal state from the snapshot and tell the source to start sending data from the last "committed" position (where the snapshot was taken).

1.7. Elasticity

Hazelcast Jet supports the scenario where a new member joins the cluster while a job is running. Currently the ongoing job will not be re-planned to start using the member, though; this is on the roadmap for a future version. The new member can also leave the cluster while the job is running and this won't affect its progress.

One caveat is the special kind of member allowed by the Hazelcast IMDG: a *lite member*. These members don't get any partitions assigned to them and will malfunction when attempting to run a DAG with partitioned edges. Lite members should not be allowed to join a Jet cluster.

Chapter 2. Get Started

In this section we'll get you started using Hazelcast Jet. We'll show you how to set up a Java project with the proper dependencies and a quick Hello World example to verify your setup.

2.1. Requirements

In the good tradition of Hazelcast products, Jet is distributed as a JAR with no other dependencies. It requires JRE version 8 or higher to run.

2.2. Using Maven and Gradle

The easiest way to start using Hazelcast Jet is to add it as a dependency to your project.

Hazelcast Jet is published on the Maven repositories. Add the following lines to your `pom.xml`:

```
<dependencies>
  <dependency>
    <groupId>com.hazelcast.jet</groupId>
    <artifactId>hazelcast-jet</artifactId>
    <version>0.5</version>
  </dependency>
</dependencies>
```

If you prefer to use Gradle, execute the following command:

```
compile 'com.hazelcast.jet:hazelcast-jet:0.5'
```

2.3. Downloading

Alternatively you can download the latest distribution package of Hazelcast Jet and add the `hazelcast-jet-<version>.jar` file to your classpath.

2.3.1. Distribution Package

The distribution package contains the following scripts to help you get started with Hazelcast Jet:

- `bin/jet-start.sh` and `bin/jet-start.bat` start a new Jet member in the current directory.
- `bin/jet-stop.sh` and `bin/jet-stop.bat` stop the member started in the current directory.
- `bin/jet-submit.sh` and `bin/jet-submit.bat` submit a Jet computation job that was packaged in a self-contained JAR file.

- `bin/cluster.sh` provides basic functionality for Hazelcast cluster manager, such as changing the cluster state, shutting down the cluster or forcing the cluster to clean its persisted data.

2.4. Verify Your Setup

You can verify your setup by running this simple program. It processes the contents of a Hazelcast `IList` that contains lines of text, finds the number of occurrences of each word in it, and stores its results in a Hazelcast `IMap`. In a distributed computation job the input and output cannot be simple in-memory structures like a Java `List`; they must reside in the cluster so any member can access them. This is why we use Hazelcast structures.

```
import com.hazelcast.jet.Jet;
import com.hazelcast.jet.JetInstance;
import com.hazelcast.jet.Pipeline;
import com.hazelcast.jet.Sinks;
import com.hazelcast.jet.Sources;

import java.util.List;
import java.util.Map;

import static com.hazelcast.jet.Traversers.traverseArray;
import static com.hazelcast.jet.aggregate.AggregateOperations.counting;
import static com.hazelcast.jet.function.DistributedFunctions.wholeItem;

public class HelloWorld {
    public static void main(String[] args) throws Exception {
        // Create the specification of the computation pipeline. Note that it is
        // a pure POJO: no instance of Jet is needed to create it.
        Pipeline p = Pipeline.create();
        p.drawFrom(Sources.<String>list("text"))
         .flatMap(word -> traverseArray(word.toLowerCase().split("\\W+")))
         .filter(word -> !word.isEmpty())
         .groupBy(wholeItem(), counting())
         .drainTo(Sinks.map("counts"));

        // Start Jet, populate the input list
        JetInstance jet = Jet.newJetInstance();
        try {
            List<String> text = jet.getList("text");
            text.add("hello world hello hello world");
            text.add("world world hello world");

            // Perform the computation
            jet.newJob(p).join();

            // Check the results
            Map<String, Long> counts = jet.getMap("counts");
            System.out.println("Count of hello: " + counts.get("hello"));
            System.out.println("Count of world: " + counts.get("world"));
        } finally {
            Jet.shutdownAll();
        }
    }
}
```

You should expect to see a lot of logging output from Jet (sent to `stderr`) and two lines on `stdout`:

```
Count of hello: 4
Count of world: 5
```

Chapter 3. Work with Jet

3.1. Start Jet and Submit Jobs to It

To create a Jet cluster, we simply start some Jet instances. Normally these would be started on separate machines, but for simple practice we can use the same JVM for two instances. Even though they are in the same JVM, they'll communicate over the network interface.

```
public class WordCount {
    public static void main(String[] args) {
        JetInstance jet = Jet.newJetInstance();
        Jet.newJetInstance();
    }
}
```

These two instances should automatically discover each other using IP multicast and form a cluster. You should see a log output similar to the following:

```
Members [2] {
  Member [10.0.1.3]:5701 - f1e30062-e87e-4e97-83bc-6b4756ef6ea3
  Member [10.0.1.3]:5702 - d7b66a8c-5bc1-4476-a528-795a8a2d9d97 this
}
```

This means the members successfully formed a cluster. Since the Jet instances start their own threads, it is important to explicitly shut them down at the end of your program; otherwise the Java process will remain alive after the `main()` method completes:

```
public class WordCount {
    public static void main(String[] args) {
        try {
            JetInstance jet = Jet.newJetInstance();
            Jet.newJetInstance();

            ... work with Jet ...

        } finally {
            Jet.shutdownAll();
        }
    }
}
```

This is how you submit a Jet pipeline for execution:

```
jet.newJob(pipeline).join();
```

Alternatively, you can submit a Core API DAG:

```
jet.newJob(dag).join();
```

Code samples with the pipeline and the Core API DAG are available at our Code Samples repo.

3.1.1. JobConfig

To gain more control over how Jet will run your job, you can pass in a `JobConfig` instance. For example, you can give your job a human- readable name:

```
JobConfig cfg = new JobConfig();
cfg.setName("my job");
jet.newJob(pipeline, cfg);
```

In the Practical Considerations section we'll deepen this story and explain how to use the `JobConfig` to to submit a job to a Jet cluster in production.

3.1.2. Manage a Submitted Job

`jet.newJob()` returns a `Job` object, which you can use to monitor the job and change its status. You can get the job's name, configuration, and submission time via `job.getName()`, `job.getConfig()`, and `job.getSubmissionTime()` methods. `job.getStatus()` will give you the current status of the job (running, failed, completed etc.). You can also call `Job.getFuture()` to block until the job completes and then get its final outcome (either success or failure).

Jet does not support canceling the job with `future.cancel()`, instead you must call `job.cancel()`. This is due to the mismatch in the semantics between `future.cancel()` on one side and `job.cancel()` plus `job.getStatus()` on the other: the future immediately transitions to "completed by cancellation", but it will take some time until the actual job in the cluster changes to that state. Not to confuse the users with these differences we decided to make `future.cancel()` fail with an exception.

3.1.3. Get a List of all Submitted Jobs

Jet keeps an inventory of all the jobs submitted to it, including those that have already completed. Access the full list with `jet.getJobs()`. You can use any `Job` instance from that list to monitor and manage a job, whether it was you or some other client that submitted it.

To get a more focused list of jobs, you can call `jet.getJobs(name)` to get all the jobs with that name that were submitted since Jet started, or `jet.getJob(name)` to get just the latest such job.

3.2. Build Your Computation Pipeline

3.2.1. The Shape of a Pipeline

The general shape of any data processing pipeline is `drawFromSource → transform → drainToSink` and the natural way to build it is from source to sink. The Pipeline API follows this pattern. For example,

```
Pipeline p = Pipeline.create();
p.drawFrom(Sources.<String>list("input"))
 .map(String::toUpperCase)
 .drainTo(Sinks.writeList("result");
```

In each step, such as `drawFrom` or `drainTo`, you create a pipeline *stage*. The stage resulting from a `drainTo` operation is called a *sink stage* and you can't attach more stages to it. All others are called *compute stages* and expect you to attach stages to them.

In a more complex scenario you'll have several sources, each starting its own pipeline branch. Then you can merge them in a multi-input transformation such as co-grouping:

```
Pipeline p = Pipeline.create();
ComputeStage<String> src1 = p.drawFrom(Sources.list("src1"));
ComputeStage<String> src2 = p.drawFrom(Sources.list("src2"));
src1.coGroup(wholeItem(), src2, wholeItem(), counting2())
    .drainTo(Sinks.writeMap("result"));
```

For further details on `coGroup` please refer to the dedicated section below.

Symmetrically, the output of a stage can be sent to more than one destination:

```
Pipeline p = Pipeline.create();
ComputeStage<String> src = p.drawFrom(Sources.list("src"));
src.map(String::toUpperCase)
   .drainTo(Sinks.writeList("uppercase"));
src.map(String::toLowerCase)
   .drainTo(Sinks.writeList("lowercase"));
```

3.2.2. Choose Your Data Sources and Sinks

Hazelcast Jet has support for these data sources and sinks:

- Hazelcast `IMap` and `ICache`, both as a batch source of just their contents and their event journal as an infinite source
- Hazelcast `IList` (batch)

- Hadoop Distributed File System (HDFS) (batch)
- Kafka topic (infinite stream)
- TCP socket (infinite stream)
- a directory on the filesystem, both as a batch source of the current file contents and an infinite source of append events to the files

You can access most of them via the `Sources` and `Sinks` utility classes. Kafka and HDFS connectors are in their separate modules.

There's a dedicated section that discusses the topic of data sources and sinks in more detail.

3.2.3. Compose the Pipeline Transforms

The simplest kind of transformation is one that can be done on each item individually and independent of other items. The major examples are

`map`, `filter` and `flatMap`. We already saw them in use in the previous examples. `map` transforms each item to another item; `filter` discards items that don't match its predicate; and `flatMap` transforms each item into zero or more output items.

groupBy

Stepping up from the simplest transforms we come to `groupBy`, the quintessential finite stream transform. It groups the data items by a key computed for each item and performs an aggregate operation over all the items in a group. The output of this transform is one aggregation result per distinct grouping key. We saw this one used in the introductory Hello World code with a word count pipeline:

```
Pipeline p = Pipeline.create();
p.drawFrom(Sources.<String>list("text"))
 .flatMap(word -> traverseArray(word.toLowerCase().split("\\W+")))
 .filter(word -> !word.isEmpty())
 .groupBy(wholeItem(), counting())
```

Let's take a moment to analyze the last line. The `groupBy()` method takes two parameters: the function to compute the grouping key and the aggregate operation. In this case the key function is a trivial identity because we use the word itself as the grouping key and the definition of the aggregate operation hides behind the `counting()` method call. This is a static method in our `AggregateOperations` utility class, which provides you with some predefined aggregate operations. You can also implement your own aggregate operations; please refer to the section dedicated to this.

If you don't need grouping and want to aggregate the full data set into a single result, you can supply a constant function to compute the grouping key: `DistributedFunctions.constantKey()`. It always returns the string `"ALL"`.

A more complex variety of pipeline transforms are those that merge several input stages into a single resulting stage. In Hazelcast Jet there are two such transforms of special interest: `coGroup` and `hashJoin`. We discuss these next.

coGroup

`coGroup` is a generalization of `groupBy` to more than one contributing data stream. Instead of a single `accumulate` primitive you provide one for each input stream so the operation can discriminate between them. In SQL terms it can be interpreted as JOIN coupled with GROUP BY. The JOIN condition is constrained to matching on the grouping key.

Here is the example we already used earlier on this page:

```
Pipeline p = Pipeline.create();
ComputeStage<String> src1 = p.drawFrom(Sources.list("src1"));
ComputeStage<String> src2 = p.drawFrom(Sources.list("src2"));
ComputeStage<Tuple2<String, Long>> coGrouped =
        src1.coGroup(wholeItem(), src2, wholeItem(), counting2());
```

These are the arguments:

1. `wholeItem()`: the key extractor function for `this` stage's items
2. `src2`: the other stage to co-group with this one
3. `wholeItem()`: the key extractor function for `src2` items
4. `counting2()`: the aggregate operation

`counting2()` is a factory method returning a 2-way aggregate operation which may be defined as follows:

```
private static AggregateOperation2<Object, Object, LongAccumulator, Long> counting2() {
    return AggregateOperation
            .withCreate(LongAccumulator::new)
            .andAccumulate0((count, item) -> count.add(1))
            .andAccumulate1((count, item) -> count.add(10))
            .andCombine(LongAccumulator::add)
            .andFinish(LongAccumulator::get);
}
```

This demonstrates the individual treatment of input streams: stream 1 is weighted so that each of its items is worth ten items from stream 0.

coGroup Builder

If you need to co-group more than three streams, you'll have to use the co-group builder object. For

example, your goal may be correlating events coming from different systems, where all the systems serve the same user base. In an online store you may have separate streams for product page visits, adding to shopping cart, payments, and deliveries. You want to correlate all the events associated with the same user. The example below calculates statistics per category for each user:

```
Pipeline p = Pipeline.create();
ComputeStage<PageVisit> pageVisit = p.drawFrom(Sources.list("pageVisit"));
ComputeStage<AddToCart> addToCart = p.drawFrom(Sources.list("addToCart"));
ComputeStage<Payment> payment = p.drawFrom(Sources.list("payment"));
ComputeStage<Delivery> delivery = p.drawFrom(Sources.list("delivery"));

CoGroupBuilder<Long, PageVisit> b = pageVisit.coGroupBuilder(PageVisit::userId);
Tag<PageVisit> pvTag = b.tag0();
Tag<AddToCart> atcTag = b.add(addToCart, AddToCart::userId);
Tag<Payment> pmtTag = b.add(payment, Payment::userId);
Tag<Delivery> delTag = b.add(delivery, Delivery::userId);

ComputeStage<Tuple2<Long, long[]>> coGrouped = b.build(AggregateOperation
        .withCreate(() -> new LongAccumulator[] {
                new LongAccumulator(),
                new LongAccumulator(),
                new LongAccumulator(),
                new LongAccumulator()
        })
        .andAccumulate(pvTag, (accs, pv) -> accs[0].add(pv.loadTime()))
        .andAccumulate(atcTag, (accs, atc) -> accs[1].add(atc.quantity()))
        .andAccumulate(pmtTag, (accs, pm) -> accs[2].add(pm.amount()))
        .andAccumulate(delTag, (accs, d) -> accs[3].add(d.days()))
        .andCombine((accs1, accs2) -> {
                    accs1[0].add(accs2[0]);
                    accs1[1].add(accs2[1]);
                    accs1[2].add(accs2[2]);
                    accs1[3].add(accs2[3]);
                })
        .andFinish(accs -> new long[] {
                accs[0].get(),
                accs[1].get(),
                accs[2].get(),
                accs[3].get()
        })
);
```

Note the interaction between the co-group building code and the aggregate operation-building code: the co-group builder gives you type tags that you then pass to the aggregate operation builder. This establishes the connection between the streams contributing to the co-group transform and the aggregate operation processing them. Refer to the section on `AggregateOperation` to learn more about it.

hashJoin

`hashJoin` is a specialization of a general "join" operation, optimized for the use case of *data enrichment*. In this scenario there is a single, potentially infinite data stream (the *primary* stream), that goes through a mapping transformation which attaches to each item some more items found by hashtable lookup. The hashtables have been populated from all the other streams (the *enriching* streams) before the consumption of the primary stream started.

For each enriching stream you can specify a pair of key-extracting functions: one for the enriching item and one for the primary item. This means that you can define a different join key for each of the enriching streams. The following example shows a three-way hash-join between the primary stream of stock trade events and two enriching streams: *products* and *brokers*.

```
Pipeline p = Pipeline.create();

// The primary stream: trades
ComputeStage<Trade> trades = p.drawFrom(Sources.<Trade>list("trades"));

// The enriching streams: products and brokers
ComputeStage<Entry<Integer, Product>> prodEntries =
        p.drawFrom(Sources.<Integer, Product>map("products"));
ComputeStage<Entry<Integer, Broker>> brokEntries =
        p.drawFrom(Sources.<Integer, Broker>map("brokers"));

// Join the trade stream with the product and broker streams
ComputeStage<Tuple3<Trade, Product, Broker>> joined = trades.hashJoin(
        prodEntries, joinMapEntries(Trade::productId),
        brokEntries, joinMapEntries(Trade::brokerId)
);
```

Products are joined on `Trade.productId` and brokers on `Trade.brokerId`. `joinMapEntries()` returns a `JoinClause`, which is a holder of the three functions that specify how to perform a join:

1. the key extractor for the primary stream's item
2. the key extractor for the enriching stream's item
3. the projection function that transforms the enriching stream's item into the item that will be used for enrichment.

Typically the enriching streams will be `Map.Entry`s coming from a key-value store, but you want just the entry value to appear as the enriching item. In that case you'll specify `Map.Entry::getValue` as the projection function. This is what `joinMapEntries()` does for you. It takes just one function, primary stream's key extractor, and fills in `Entry::getKey` and `Entry::getValue` for the enriching stream key extractor and the projection function, respectively.

In the interest of performance the entire enriching dataset resides on each cluster member. That's why

this operation is also known as a *replicated* join. This is something to keep in mind when estimating the RAM requirements for a hash-join operation.

hashJoin Builder

You can hash-join a stream with up to two enriching streams using the API we demonstrated above. If you have more than two enriching streams, you'll use the hash-join builder. For example, you may want to enrich a trade with its associated product, broker, and market:

```
Pipeline p = Pipeline.create();

// The primary stream: trades
ComputeStage<Trade> trades = p.drawFrom(Sources.<Trade>list("trades"));

// The enriching streams: products and brokers
ComputeStage<Entry<Integer, Product>> prodEntries =
        p.drawFrom(Sources.<Integer, Product>map("products"));
ComputeStage<Entry<Integer, Broker>> brokEntries =
        p.drawFrom(Sources.<Integer, Broker>map("brokers"));
ComputeStage<Entry<Integer, Market>> marketEntries =
        p.drawFrom(Sources.<Integer, Market>map("markets"));

HashJoinBuilder<Trade> b = trades.hashJoinBuilder();
Tag<Product> prodTag = b.add(prodEntries, joinMapEntries(Trade::productId));
Tag<Broker> brokTag = b.add(brokEntries, joinMapEntries(Trade::brokerId));
Tag<Market> marketTag = b.add(marketEntries, joinMapEntries(Trade::marketId));
ComputeStage<Tuple2<Trade, ItemsByTag>> joined = b.build();
```

The data type on the hash-joined stage is `Tuple2<Trade, ItemsByTag>`. The next snippet shows how to use it to access the primary and enriching items:

```
ComputeStage<String> mapped = joined.map(
        (Tuple2<Trade, ItemsByTag> t) -> {
            Trade trade = t.f0();
            ItemsByTag ibt = t.f1();
            Product product = ibt.get(prodTag);
            Broker broker = ibt.get(brokTag);
            Market market = ibt.get(marketTag);
            return trade + ": " + product + ", " + broker + ", " + market;
        });
```

3.3. Implement Your Aggregate Operation

The single most important kind of processing Jet does is aggregation. In general it is a transformation

of a set of input values into a single output value. The function that does this transformation is called the "aggregate function". A basic example is `sum` applied to a set of integer numbers, but the result can also be a complex value, for example a list of all the input items.

Jet's library contains a range of predefined aggregate functions, but it also exposes an abstraction, called `AggregateOperation`, that allows you to plug in your own. Since Jet does the aggregation in a parallelized and distributed way, you can't simply supply a piece of Java code that does it; we need you to break it down into several smaller pieces that fit into Jet's processing engine.

The ability to compute the aggregate function in parallel comes at a cost: Jet must be able to give a slice of the total data set to each processing unit and then combine the partial results from all the units. The combining step is crucial: it will only make sense if we're combining the partial results of a *commutative associative* function (CA for short). On the example of `sum` this is trivial: we know from elementary school that `+` is a CA operation. If you have a stream of numbers: `{17, 37, 5, 11, 42}`, you can sum up `{17, 5}` separately from `{42, 11, 37}` and then combine the partial sums (also note the reordering of the elements).

If you need something more complex, like `average`, it doesn't by itself have this property; however if you add one more ingredient, the `finish` function, you can express it easily. Jet allows you to first compute some CA function, whose partial results can be combined, and then at the very end apply the `finish` function on the fully combined result. To compute the `average`, your CA function will output the pair `(sum, count)`. Two such pairs are trivial to combine by summing each component. The `finish` function will be `sum / count`.

In addition to the mathematical side, there is also the practical one: you have to provide Jet with a specific mutable object, called the `accumulator`, which will keep the "running score" of the operation in progress. For the `average` example, it would be something like

```
public class AvgAccumulator {
    private long sum;
    private long count;

    public void accumulate(long value) {
        sum += value;
        count++;
    }

    public void combine(AvgAccumulator that) {
        this.sum += that.sum;
        this.count += that.sum;
    }

    public double finish() {
        return (double) sum / count;
    }
}
```

This object will also have to be serializable, and preferably with Hazelcast's serialization instead of Java's because in a group-by operation there's one accumulator per each key and all of them have to be sent across the network to be combined and finished.

Instead of requiring you to write a complete class from scratch, Jet separates the concern of holding the accumulated state from that of the computation performed on it. This means that you just need one accumulator class per the kind of structure that holds the accumulated data, as opposed to one per each aggregate operation. Jet's library offers in the `com.hazelcast.jet.accumulator` package several such classes, one of them being `LongLongAccumulator`, which is a match for our `average` function. You'll just have to supply the logic on top of it.

Specifically, you have to provide a set of five functions (we call them "primitives"):

- `create` a new accumulator object.
- `accumulate` the data of an item by mutating the accumulator's state.
- `combine` the contents of the right-hand accumulator into the left-hand one.
- `deduct` the contents of the right-hand accumulator from the left-hand one (undo the effects of `combine`).
- `finish` accumulation by transforming the accumulator object into the final result.

So far we mentioned all of these except for `deduct`. This one is optional and Jet can manage without it, but if you are computing a sliding window over an infinite stream, this primitive can give a significant performance boost because it allows Jet to reuse the results of the previous calculations.

If you happen to have a deeper familiarity with JDK's java.util.stream API, you'll find

`AggregateOperation` quite similar to `Collector`, which is also a holder of several functional primitives. Jet's definitions are slightly different, though, and there's also the additional `deduct` primitive.

Let's see how this works with our `average` function. Using `LongLongAccumulator` we can express our `accumulate` primitive as

```
(acc, n) -> {
    acc.setValue1(acc.getValue1() + n);
    acc.setValue2(acc.getValue2() + 1);
}
```

The `finish` primitive will be

```
acc -> (double) acc.getValue1() / acc.getValue2()
```

Now we have to define the other three primitives to match our main logic. For `create` we just refer to the constructor: `LongLongAccumulator::new`. The `combine` primitive expects you to update the left-hand accumulator with the contents of the right-hand one, so:

```
(left, right) -> {
    left.setValue1(left.getValue1() + right.getValue1());
    left.setValue2(left.getValue2() + right.getValue2());
}
```

Deducting must undo the effect of a previous `combine`:

```
(left, right) -> {
    left.setValue1(left.getValue1() - right.getValue1());
    left.setValue2(left.getValue2() - right.getValue2());
}
```

All put together, we can define our counting operation as follows:

```
AggregateOperation1<Long, LongLongAccumulator, Double> aggrOp = AggregateOperation
    .withCreate(LongLongAccumulator::new)
    .andAccumulate((acc, n) -> {
        acc.setValue1(acc.getValue1() + n);
        acc.setValue2(acc.getValue2() + 1);
    })
    .andCombine((left, right) -> {
        left.setValue1(left.getValue1() + right.getValue1());
        left.setValue2(left.getValue2() + right.getValue2());
    })
    .andDeduct((left, right) -> {
        left.setValue1(left.getValue1() - right.getValue1());
        left.setValue2(left.getValue2() - right.getValue2());
    })
    .andFinish(acc -> (double) acc.getValue1() / acc.getValue2());
```

Let's stop for a second to look at the type we got: `AggregateOperation1<Long, LongLongAccumulator, Double>`. Its type parameters are: 1. `Long`: the type of the input item 2. `LongLongAccumulator`: the type of the accumulator 3. `Double`: the type of the result

Specifically note the `1` at the end of the type's name: it signifies that it's the specialization of the general `AggregateOperation` to exactly one input stream. In Hazelcast Jet you can also perform a co-grouping operation, aggregating several input streams together. Since the number of input types is variable, the general `AggregateOperation` type cannot statically capture them and we need separate subtypes. We decided to statically support up to three input types; if you need more, you'll have to resort to the less type-safe, general `AggregateOperation`.

Let us now study a use case that calls for co-grouping. We are interested in the behavior of users in an online shop application and want to gather the following statistics for each user:

1. total load time of the visited product pages
2. quantity of items added to the shopping cart
3. amount paid for bought items

This data is dispersed among separate datasets: `PageVisit`, `AddToCart` and `Payment`. Note that in each case we're dealing with a simple `sum` applied to a field in the input item. We can perform a co-group transform with the following aggregate operation:

```
Pipeline p = Pipeline.create();
ComputeStage<PageVisit> pageVisit = p.drawFrom(Sources.list("pageVisit"));
ComputeStage<AddToCart> addToCart = p.drawFrom(Sources.list("addToCart"));
ComputeStage<Payment> payment = p.drawFrom(Sources.list("payment"));

AggregateOperation3<PageVisit, AddToCart, Payment, LongAccumulator[], long[]> aggrOp =
        AggregateOperation
                .withCreate(() -> new LongAccumulator[] {
                        new LongAccumulator(),
                        new LongAccumulator(),
                        new LongAccumulator()
                })
                .<PageVisit>andAccumulate0((accs, pv) -> accs[0].add(pv.loadTime()))
                .<AddToCart>andAccumulate1((accs, atc) -> accs[1].add(atc.quantity()))
                .<Payment>andAccumulate2((accs, pm) -> accs[2].add(pm.amount()))
                .andCombine((accs1, accs2) -> {
                    accs1[0].add(accs2[0]);
                    accs1[1].add(accs2[1]);
                    accs1[2].add(accs2[2]);
                })
                .andFinish(accs -> new long[] {
                        accs[0].get(),
                        accs[1].get(),
                        accs[2].get()
                });
ComputeStage<Entry<Long, long[]>> coGrouped = pageVisit.coGroup(PageVisit::userId,
        addToCart, AddToCart::userId,
        payment, Payment::userId,
        aggrOp);
```

Note how we got an `AggregateOperation3` and how it captured each input type. When we use it as an argument to a co-group transform, the compiler will ensure that the `ComputeStage`s we attach it to have the correct type and are in the correct order.

On the other hand, if you use the co-group builder object, you'll construct the aggregate operation by calling `andAccumulate(tag, accFn)` with all the tags you got from the co-group builder, and the static type will be just `AggregateOperation`. The compiler won't be able to match up the inputs to their treatment in the aggregate operation.

3.4. Infinite Stream Processing

Distributed stream processing has two major variants: finite and infinite. Let's discuss this difference and some concerns specific to infinite streams.

3.4.1. Finite aka. Batch Processing

Finite stream (batch) processing is the simpler variant where you provide one or more pre-existing datasets and order Jet to mine them for interesting information. The most important workhorse in this area is the "join, group and aggregate" operation: you define a classifying function that computes a grouping key for each of the datasets and an aggregate operation that will be performed on all the items in each group, yielding one result item per distinct key.

3.4.2. The Importance of "Right Now"

In batch jobs the data we process represents a point-in-time snapshot of our state of knowledge (for example, warehouse inventory where individual data items represent items on stock). We can recapitulate each business day by setting up regular snapshots and batch jobs. However, there is more value hiding in the freshest data - our business can win by reacting to minute-old or even second-old updates. To get there we must make a shift from the finite to the infinite: from the snapshot to a continuous influx of events that update our state of knowledge. For example, an event could pop up in our stream every time an item is checked in or out of the warehouse.

A single word that captures the above story is *latency*: we want our system to minimize the latency from observing an event to acting upon it.

3.4.3. Windowing

In an infinite stream, the dimension of time is always there. Consider a batch job: it may process a dataset labeled "Wednesday", but the computation itself doesn't have to know this. Its results will be understood from the outside to be "about Wednesday". An infinite stream, on the other hand, delivers information about the reality as it is unfolding, in near-real time, and the computation itself must deal with time explicitly.

Another point: in a batch it is obvious when to stop aggregating and emit the results: when we have exhausted the whole dataset. However, with infinite streams we need a policy on how to select finite chunks whose aggregate results we are interested in. This is called *windowing*. We imagine the window as a time interval laid over the time axis. A given window contains only the events that belong to that interval.

A very basic type of window is the *tumbling window*, which can be imagined to advance by tumbling over each time. There is no overlap between the successive positions of the window. In other words, it splits the time-series data into batches delimited by points on the time axis. The result of this is very similar to running a sequence of batch jobs, one per time interval.

A more useful and powerful policy is the *sliding window*: instead of splitting the data at fixed boundaries, it lets it roll in incrementally, new data gradually displacing the old. The window (pseudo)continuously slides along the time axis.

Another popular policy is called the *session window* and it's used to detect bursts of activity by correlating events bunched together on the time axis. In an analogy to a user's session with a web

application, the session window "closes" when the specified session timeout elapses with no further events.

3.4.4. Time Ordering and the Watermark

Usually the time of observing an event is explicitly written in the stream item. There is no guarantee that items will occur in the stream ordered by the value of that field; in fact in many cases it is certain that they won't. Consider events gathered from users of a mobile app: for all kinds of reasons the items will arrive to our datacenter out of order, even with significant delays due to connectivity issues.

This disorder in the event stream makes it more difficult to formally specify a rule that tells us at which point all the data for a given window has been gathered, allowing us to emit the aggregated result.

To approach these challenges we use the concept of the *watermark*. It is a timestamped item inserted into the stream that tells us "from this point on there will be no more items with timestamp less than this". Unfortunately, we almost never know for sure when such a statement becomes true and there is always a chance some events will arrive even later. If we do observe such an offending item, we must categorize it as "too late" and just filter it out.

Note the tension in defining the "perfect" watermark for a given use case: it is bad both the more we wait and the less we wait to emit a given watermark. The more we wait, the higher the latency of getting the results of the computation; the less we wait, the worse their accuracy due to missed events.

For the above reasons the policy to compute the watermark is not hardcoded and you as the user must decide which one to use. Hazelcast Jet comes with some predefined policies which you can tune with a few configurable parameters. You can also write your own policy from scratch.

3.4.5. Fault Tolerance and Processing Guarantees

One less-than-obvious consequence of stepping up from finite to infinite streams is the difficulty of forever maintaining the continuity of the output, even in the face of changing cluster topology. A Jet node may leave the cluster due to an internal error, loss of networking, or deliberate shutdown for maintenance. This will cause the computation job to be suspended. Except for the obvious problem of new data pouring in while we're down, we have a much more fiddly issue of restarting the computation in a differently laid-out cluster exactly where it left off and neither miss anything nor process it twice. The technical term for this is the "exactly-once processing guarantee".

Hazelcast Jet transparently coordinates the execution of submitted jobs. One member is assigned the role of the *coordinator*. It tells other members what to do and they report to it any status changes. The coordinator may fail and the cluster will automatically re-elect another one. If any other member fails, the coordinator restarts the job on the remaining members.

In Jet, job submission works on the fire-and-forget principle: once you have submitted it, you can disconnect with no effect on the job. Any other client or Jet member can request a local `Job` instance which allows it to monitor and manage the job.

Snapshotting the State of Computation

To be able to tolerate failures, Jet takes snapshots of the entire state of the computation at regular intervals. The snapshot is coordinated across the cluster and synchronized with a checkpoint on the data source. The source must ensure that, in the case of a restart, it will be able to replay all the data it emitted after the last checkpoint. Each of the other components must ensure it will be able to restore its processing state to exactly what it was at the last snapshot. If a cluster member goes away, Jet will restart the job on the remaining members, rewind the sources to the last checkpoint, restore the state of processing from the last snapshot, and then seamlessly continue from that point.

Level of Safety

Jet stores job metadata and snapshot data in Hazelcast `IMap`s, which means that you don't have to install any other system for fault tolerance. However, the fault tolerance mechanism is at most as safe as the `IMap` itself. Therefore, it is important to configure level of safety for the `IMap`. `IMap` is a replicated in-memory data structure, storing each key-value pair on a configurable number of cluster members. By default it will store one primary copy plus one backup copy, resulting in a system that tolerates the failure of a single member at a time. You can tweak this setting when starting Jet, for example increase the backup count to two:

```
JetConfig config = new JetConfig();
config.getInstanceConfig().setBackupCount(2);
JetInstance = Jet.newJetInstance(config);
```

Exactly-Once

As always when guarantees are involved, the principle of the weakest link applies: if any part of the system is unable to provide it, the system as a whole fails to provide it. The critical points are the sources and sinks because they are the boundary between the domain under Jet's control and the environment. A source must be able to consistently replay data to Jet from a point it asks for, and the sink must either support transactions or be *idempotent*, tolerating duplicate submission of data.

As of version 0.5, Hazelcast Jet supports exactly-once with the source being either a Hazelcast `IMap` or a Kafka topic, and the sink being a Hazelcast `IMap`.

At-Least-Once

A lesser, but still useful guarantee you can configure Jet for is "at-least-once". In this case no stream item will be missed, but some items may get processed again after a restart, as if they represented new events. Jet can provide this guarantee at a higher throughput and lower latency than exactly-once, and some kinds of data processing can gracefully tolerate it. In some other cases, however, duplicate processing of data items can have quite surprising consequences. There is more information about this in our Under the Hood chapter.

We also have an in-between case: if you configure Jet for exactly-once but use Kafka as the sink, after a job restart you may get duplicates in the output. As opposed to duplicating an input item, this is much

more benign because it just means getting the exact same result twice.

Enabling Snapshotting

Fault tolerance is off by default. To activate it for a job, create a `JobConfig` object and set the *processing guarantee*. You can also configure *snapshot interval*.

```
JobConfig jobConfig = new JobConfig();
jobConfig.setProcessingGuarantee(ProcessingGuarantee.EXACTLY_ONCE);
jobConfig.setSnapshotIntervalMillis(SECONDS.toMillis(10));
```

Using less frequent snapshots, more data will have to be replayed and the temporary spike in the latency of the output will be greater. More frequent snapshots will reduce the throughput and introduce more latency variation during regular processing.

Split-Brain Protection

A particularly nasty kind of failure is the "split brain": due to a very specific pattern in the loss of network connectivity the cluster splits into two parts, where within each part the members see each other, but none of those in the other part(s). Each part by itself lives on thinking the other members left the cluster. Now we have two fully-functioning Jet clusters where there was supposed to be one. Each one will recover and restart the same Jet job, making a mess in our application.

Hazelcast Jet offers a mechanism to fight off this hazard: *split-brain protection*. It works by ensuring that a job cannot be restarted in a cluster whose size isn't more than half of what it was before the job was suspended. Enable split-brain protection like this:

```
jobConfig.setSplitBrainProtection(true);
```

A loophole here is that, after the split brain has occurred, you could add more members to any of the sub-clusters and have them both grow to more than half the previous size. Since the job will keep trying to restart itself and by definition one cluster has no idea of the other's existence, it will restart as soon as the quorum value is reached.

3.4.6. Scaling up Jobs

After a job is submitted to the cluster, new nodes can be started and the job can be scaled up. Hazelcast Jet 0.6 introduces a new method into the `Job` interface for this purpose. When `Job.restart()` is invoked, ongoing execution of the job is interrupted and a new execution is scheduled. If the snapshotting mechanism enabled, the job is restarted from the last successful snapshot. Therefore, the restart procedure respects to the configured processing guarantee.

3.4.7. Note for Hazelcast Jet version 0.5

Hazelcast Jet's version 0.5 was released with the Pipeline API still under construction. We started from the simple case of batch jobs and we support the major batch operation of (co)group-and-aggregate, but still lack the API to define the windowing and watermark policies. Other, non-aggregating operations aren't sensitive to the difference between finite and infinite streams and are ready to use. The major example here is data enrichment (hash join), which is essentially a mapping stream transformation. The next release of Jet will feature a fully developed API that supports windowed aggregation of infinite streams and we also plan to add more batch transforms (`sort` and `distinct` for example).

On the other hand, Jet's core has had full-fledged support for all of the windows described above since version 0.4. You can refer to the Under the Hood chapter for details on how to create a Core API DAG that does infinite stream aggregation.

3.5. Source and Sink Connectors

3.5.1. Overview

In an earlier section we briefly listed the resources you can use as sources and sinks of a Jet job's data and where they fit in the general outline of a pipeline. Now let's revisit this topic in more detail.

Jet accesses data sources and sinks via its *connectors*. They are a computation job's point of contact with the outside world. Although the connectors do their best to unify the various kinds of resources under the same "data stream" paradigm, there are still many concerns that need your attention.

Is it Infinite?

The first decision when building a Jet computation job is whether it will deal with finite or infinite data. A typical example of a finite resource is a persistent storage system, whereas an infinite one is usually like a FIFO queue, discarding old data. This is true both for sources and sinks.

Finite data is handled by batch jobs and there are less concerns to deal with. Examples of finite resources are the Hazelcast `IMap`/`ICache` and the Hadoop Distributed File System (HDFS). In the infinite category the most popular choice is Kafka, but a Hazelcast `IMap`/`ICache` can also be used as an infinite source of update events (via the Event Journal feature). You can also set up an `IMap`/`ICache` as a sink for an infinite amount of data, either by ensuring that the size of the keyset will be finite or by allowing the eviction of old entries.

Is it Replayable?

Most finite data sources are replayable because they come from persistent storage. You can easily replay the whole dataset. However, an infinite data source may be of such nature that it can be consumed only once. An example is the TCP socket connector. Such sources are bad at fault tolerance: if anything goes wrong during the computation, it cannot be retried.

Does it Support Checkpointing?

It would be quite impractical if you could only replay an infinite data stream from the very beginning. This is why you need *checkpointing*: the ability of the stream source to replay its data from the point you choose, discarding everything before it. Both Kafka and the Hazelcast Event Journal support this.

Is it Distributed?

A distributed computation engine prefers to work with distributed data resources. If the resource is not distributed, all Jet members will have to contend for access to a single endpoint. Kafka, HDFS, `IMap` and `ICache` are all distributed. On the other hand, an `IList` is not; it iss stored on a single member and all append operations to it must be serialized. When using it as a source, only one Jet member will be pulling its data.

A `file` source/sink is another example of a non-distributed data source, but with a different twist: it's more of a "manually distributed" resource. Each member will access its own local filesystem, which means there will be no contention, but there will also be no global coordination of the data. To use it as a source, you have to prepare the files on each machine so each Jet member gets its part of the data. When used as a sink, you'll have to manually gather all the pieces that Jet created around the cluster.

What about Data Locality?

If you're looking to achieve record-breaking throughput for your application, you'll have to think carefully how close you can deliver your data to the location where Jet will consume and process it. For example, if your source is HDFS, you should align the topologies of the Hadoop and Jet clusters so that each machine that hosts an HDFS member also hosts a Jet member. Jet will automatically figure this out and arrange for each member to consume only the slice of data stored locally.

If you're using `IMap`/`ICache` as data sources, you have two basic choices: have Jet connect to a Hazelcast IMDG cluster, or use Jet itself to host the data (since a Jet cluster is at the same time a Hazelcast IMDG cluster). In the second case Jet will automatically ensure a data-local access pattern, but there's a caveat: if the Jet job causes an error of unrestricted scope, such as `OutOfMemoryError` or `StackOverflowError`, it will have unpredictable consequences for the state of the whole Jet member, jeopardizing the integrity of the data stored on it.

Overview of Sources and Sinks

The table below gives you a high-level overview of all the source and sink connectors we deliver with Jet. There are links to Javadoc and code samples. The sections following this one present each connector in more detail.

Table 1. Sources and Sinks

Resource	Javadoc	Sample	Infinite?	Replayable?	Checkpointing?	Distributed?	Data Locality
IMap	Source Sink	Sample					Src Sink

Resource	Javadoc	Sample	Infinite?	Replayable?	Checkpointing?	Distributed?	Data Locality
ICache	Source Sink	Sample					Src Sink
IMap in another cluster	Source Sink	Sample					
ICache in another cluster	Source Sink						
IMap's Event Journal	Source	Sample					
ICache's Event Journal	Source						
Event Journal of IMap in another cluster	Source	Sample					
Event Journal of ICache in another cluster	Source						
IList	Source Sink	Sample					
IList in another cluster	Source Sink						
HDFS	Source Sink	Sample					
Kafka	Source Sink	Source					
Files	Source Sink	Sample					
File Watcher	Source	Sample (Core API)					

Resource	Javadoc	Sample	Infinite?	Replayable?	Checkpointing?	Distributed?	Data Locality
TCP Socket	Source Sink	Source Sink					
Application Log	Sink	Sink	N/A	N/A			

3.5.2. Hazelcast IMDG

IMap and ICache

Hazelcast IMDG's `IMap` and `ICache` are very similar in the way Jet uses them and largely interchangeable. `IMap` has a bit more features. The simplest way to use them is as finite sources of their contents, but if you enable the Event Journal on a map/cache, you'll be able to use it as a source of an infinite stream of update events (see below).

The most basic usage is very simple, here are snippets to use `IMap` and `ICache` as a source and a sink:

```
Pipeline p = p.create();
ComputeStage<Entry<String, Long>> stage =
        p.drawFrom(Sources.<String, Long>map("myMap"));
stage.drainTo(Sinks.map("myMap"));
```

```
Pipeline p = p.create();
ComputeStage<Entry<String, Long>> stage =
        p.drawFrom(Sources.<String, Long>cache("myCache"));
stage.drainTo(Sinks.cache("myCache"));
```

In these snippets we draw from and drain to the same kind of structure, but you can use any combination.

Update Entries in IMap

When you use an `IMap` as a sink, instead of just pushing the data into it you may have to merge the new with the existing data or delete the existing data. Hazelcast Jet supports this with map-updating sinks which rely on Hazelcast IMDG's Entry Processor feature. An entry processor allows you to atomically execute a piece of code against a map entry, in a data-local manner.

The updating sinks come in three variants:

1. `mapWithMerging`, where you provide a a function that computes the map value from the stream item and a merging function that gets called if a value already exists in the map. Here's an example that concatenates string values:

```
Pipeline pipeline = Pipeline.create();
pipeline.drawFrom(Sources.map("mymap"))
        .drainTo(
                Sinks.mapWithMerging(
                        "mymap",
                        item -> item.getKey(),
                        item -> item.getValue(),
                        (oldValue, newValue) -> oldValue + newValue
                )
        );
```

This operation is **NOT** lock-aware, it will process the entries no matter if they are locked or not. The reason for this behavior is that under the hood, we are applying the update function on keys in batches for performance reasons and this operation does not respect locked entries. So if you use this method on locked entries, your entries will be updated without respecting the lock and mutual exclusion contract will be broken. Use `mapWithEntryProcessor` if you need locking behavior which respects the locks on entries.

2. `mapWithUpdating`, where you provide a single updating function that combines the roles of the two functions in `mapWithMerging`. It will be called on the stream item and the existing value, if any. Here's an example that concatenates string values:

```
Pipeline pipeline = Pipeline.create();
pipeline.drawFrom(Sources.map("mymap"))
        .drainTo(
                Sinks.mapWithUpdating(
                        "mymap",
                        item -> item.getKey(),
                        (oldValue, item) -> (oldValue != null ? oldValue : "")
                                            + item.getValue()
                )
        );
```

This operation is **NOT** lock-aware, it will process the entries no matter if they are locked or not. The reason for this behavior is that under the hood, we are applying the merge function on keys in batches for performance reasons and this operation does not respect locked entries. So if you use this method on locked entries, your entries will be updated without respecting the lock and mutual exclusion contract will be broken. Use `mapWithEntryProcessor` if you need locking behavior which respects the locks on entries.

3. `mapWithEntryProcessor`, where you provide a function that returns a full-blown `EntryProcessor`

instance that will be submitted to the map. This is the most general variant, but can't use batching that the other variants do and thus has a higher cost per item. You should use it only if you need a specialized entry processor that can't be expressed in terms of the other variants. This example takes the values of the map and submits an entry processor that increments the values by 5 :

```
Pipeline pipeline = Pipeline.create();
pipeline.drawFrom(Sources.map("mymap"))
        .drainTo(
                Sinks.mapWithEntryProcessor(
                        "mymap",
                        item -> item.getKey(),
                        item -> new IncrementEntryProcessor(5)
                )
        );

static class IncrementEntryProcessor implements EntryProcessor<Integer, Integer> {

    private int incrementBy;

    public IncrementEntryProcessor(int incrementBy) {
            this.incrementBy = incrementBy;
    }

    @Override
    public Object process(Entry<Integer, Integer> entry) {
            return entry.setValue(entry.getValue() + incrementBy);
    }

    @Override
    public EntryBackupProcessor<Integer, Integer> getBackupProcessor() {
            return null;
    }
}
```

Access an External Cluster

To access a Hazelcast IMDG cluster separate from the Jet cluster, you have to provide Hazelcast client configuration for the connection. In this simple example we use programmatic configuration to draw from and drain to remote `IMap` and `ICache`. Just for variety, we funnel the data from `IMap` to `ICache` and vice versa:

```
ClientConfig cfg = new ClientConfig();
cfg.getGroupConfig().setName("myGroup").setPassword("pAssw0rd");
cfg.getNetworkConfig().addAddress("node1.mydomain.com", "node2.mydomain.com");

Pipeline p = p.create();
ComputeStage<Entry<String, Long>> fromMap =
        p.drawFrom(Sources.<String, Long>remoteMap("inputMap", cfg));
ComputeStage<Entry<String, Long>> fromCache =
        p.drawFrom(Sources.<String, Long>remoteCache("inputCache", cfg));
fromMap.drainTo(Sinks.remoteCache("outputCache", cfg));
fromCache.drainTo(Sinks.remoteMap("outputMap", cfg));
```

For a full discussion on how to configure your client connection, refer to the Hazelcast IMDG documentation on this topic.

Optimize Data Traffic at the Source

If your use case calls for some filtering and/or transformation of the data you retrieve, you can optimize the traffic volume by providing a filtering predicate and an arbitrary transformation function to the source connector itself and they'll get applied on the remote side, before sending:

```
Pipeline p = p.create();
p.drawFrom(Sources.<String, Person, Integer>remoteMap(
        "inputMap", clientConfig,
        e -> e.getValue().getAge() > 21,
        e -> e.getValue().getAge()));
```

The same optimization works on a local `IMap`, too, but has less impact. However, Hazelcast IMDG goes a step further in optimizing your filtering and mapping to a degree that matters even locally. If you don't need fully general functions, but can express your predicate via `Predicates` or `PredicateBuilder`, they will create a specialized predicate instance that can test the object without deserializing it. Similarly, if the mapping you need is of a constrained kind where you just extract one or more object fields (attributes), you can specify a *projection* instead of a general mapping lambda: `Projections.singleAttribute()` or `Projections.multiAttribute()`. These will extract the listed attributes without deserializing the whole object. For these optimizations to work, however, your objects must employ Hazelcast's portable serialization. They are especially relevant if the volume of data you need in the Jet job is significantly less than the volume of the stored data.

Note that the above feature is not available on `ICache`. It is, however, available on `ICache's event journal, which we introduce next.

Receive an Infinite Stream of Update Events

You can use `IMap`/`ICache` as sources of infinite event streams. For this to work you have to enable the Event Journal on your data structure. This is a feature you set in the Jet/IMDG instance configuration,

which means you cannot change it while the cluster is running.

This is how you enable the Event Journal on an `IMap`:

```
JetConfig cfg = new JetConfig();
cfg.getHazelcastConfig()
   .getMapEventJournalConfig("inputMap")
   .setEnabled(true)
   .setCapacity(1000) // how many events to keep before evicting
   .setTimeToLiveSeconds(10); // evict events older than this
JetInstance jet = Jet.newJetInstance(cfg);
```

The default journal capacity is 10,000 and the default time-to-live is 0 (which means "unlimited"). Since the entire event journal is kept in RAM, you should take care to adjust these values to match your use case.

The configuration API for `ICache` is identical:

```
cfg.getHazelcastConfig()
   .getCacheEventJournalConfig("inputCache")
   .setEnabled(true)
   .setCapacity(1000)
   .setTimeToLiveSeconds(10);
```

Once properly configured, you use Event Journal sources like this:

```
Pipeline p = Pipeline.create();
ComputeStage<Entry<String, Long>> fromMap =
        p.drawFrom(Sources.<String, Long>mapJournal("inputMap", START_FROM_CURRENT));
ComputeStage<Entry<String, Long>> fromCache =
        p.drawFrom(Sources.<String, Long>cacheJournal("inputCache", START_FROM_CURRENT));
```

`IMap` and `ICache` are on an equal footing here. The second argument, `START_FROM_CURRENT` here, means "start receiving from events that occur after the processing starts". If you specify `START_FROM_OLDEST`, you'll get all the events still on record.

This version of methods will only emit `ADDED` and `UPDATED` event types. Also, it will map the event object to simple `Map.Entry` with the key and new value. If you want to receive all types of events, use the second version of methods:

```
ComputeStage<EventJournalMapEvent<String, Long>> allFromMap =
        p.drawFrom(Sources.<String, Long, EventJournalMapEvent<String, Long>>mapJournal(
"inputMap",
                alwaysTrue(), identity(), START_FROM_CURRENT));
ComputeStage<EventJournalCacheEvent<String, Long>> allFromCache =
        p.drawFrom(Sources.<String, Long, EventJournalCacheEvent<String, Long>
>cacheJournal("inputCache",
                alwaysTrue(), identity(), START_FROM_CURRENT));
```

Note the type of the stream element: `EventJournalMapEvent` and `EventJournalCacheEvent`. These are almost the same and have these methods:

- `getKey()`
- `getOldValue()`
- `getNewValue()`
- `getType()`

The only difference is the return type of `getType()` which is specific to each kind of structure and gives detailed insight into what kind of event it reports. *Add*, *remove* and *update* are the basic ones, but there are also *evict*, *clear*, *expire* and some others.

Finally, you can get all of the above from a map/cache in another cluster, you just have to prepend `remote` to the source names and add a `ClientConfig`, for example:

```
ComputeStage<Entry<String, Long>> fromRemoteMap = p.drawFrom(
        Sources.<String, Long>remoteMapJournal("inputMap", clientConfig(),
START_FROM_CURRENT));
ComputeStage<Entry<String, Long>> fromRemoteCache = p.drawFrom(
        Sources.<String, Long>remoteCacheJournal("inputCache", clientConfig(),
START_FROM_CURRENT));
```

IList

Whereas `IMap` and `ICache` are the recommended choice of data sources and sinks in Jet jobs, Jet supports `IList` purely for convenience during prototyping, unit testing and similar non-production situations. It is not a partitioned and distributed data structure and only one cluster member has all the contents. In a distributed Jet job all the members will compete for access to the single member holding it.

With that said, `IList` is very simple to use. Here's an example how to fill it with test data, consume it in a Jet job, dump its results into another list, and fetch the results (we assume you already have a Jet instance in the variable `jet`):

```
IList<Integer> inputList = jet.getList("inputList");
for (int i = 0; i < 10; i++) {
    inputList.add(i);
}

Pipeline p = Pipeline.create();
p.drawFrom(Sources.<Integer>list("inputList"))
 .map(i -> "item" + i)
 .drainTo(Sinks.list("resultList"));

jet.newJob(p).join();

IList<String> resultList = jet.getList("resultList");
System.out.println("Results: " + new ArrayList<>(resultList));
```

You can access a list in an external cluster as well, by providing a `ClientConfig` object:

```
ClientConfig clientConfig = new ClientConfig();
clientConfig.getGroupConfig().setName("myGroup").setPassword("pAssw0rd");
clientConfig.getNetworkConfig().addAddress("node1.mydomain.com", "node2.mydomain.com");

Pipeline p = Pipeline.create();
ComputeStage<Object> stage = p.drawFrom(Sources.remoteList("inputlist", clientConfig));
stage.drainTo(Sinks.remoteList("resultList", clientConfig));
```

3.5.3. File and Socket

Hazelcast Jet provides a few connectors that have limited production use, but are simple and can be very useful in an early rapid prototyping phase. These are the connectors for the local file system and TCP/IP sockets. They assume the data is in the form of plain text and emit/receive data items which represent individual lines of text.

Some of these sources are infinite, but when used in a stream processing job they don't offer any fault tolerance because they are not replayable. The finite variant of the file source is trivially replayable: it just reads the same files again.

File

These connectors work with a directory in the file system on each member. Since each member has its own file system, these are to some extent distributed sources and sinks; however there is no unified view of all the data on all members. The user must manually distribute the source data and collect the sink data from all the cluster members.

Source

Jet provides two main ways to use the filesystem as a source:

1. `Sources.files()`: read all the files in a directory and complete. The files should not change while being read.
2. `Sources.fileWatcher()`: first emit the contents of the files present in the directory and then continue watching the directory for further changes. Each time a complete line of text appears in an existing or a newly created file, the source emits another data item. The existing content in the files should not change. This source completes only if the watched directory is deleted.

Sink

The `Sources.files()` sink writes output to several files in the configured directory. Each underlying processor writes to its own file to avoid contention.

The file sink only guarantees that items have been flushed to the operating system on a snapshot, but it doesn't guarantee that the content is actually written to disk.

The socket source can be used to receive text input over a network socket.

Socket

These connectors open a blocking client TCP/IP socket and send/receive data over it. As already noted, the data must be in the form of lines of plain text.

Source

Each underlying processor of the Socket Source connector opens its own client socket and asks for data from it. The user supplies the `host:port` connection details. The server side should ensure a meaningful dispersion of data among all the connected clients, but how it does it is outside of Jet's control.

You can study a comprehensive code sample including a sample socket server using Netty.

Sink

The Socket Sink also opens one client socket per processor and pushes lines of text into it. It is the duty of the server-side system to collect the data from all the concurrently connected clients.

3.5.4. HDFS

The Hadoop Distributed File System is a production-worthy choice for both a data source and sink in a batch computation job. It is a distributed, replicated storage system that handles these concerns automatically, exposing a simple unified view to the client.

The HDFS source and sink require a configuration object of type JobConf which supplies the input and output paths and formats. No actual MapReduce job is created, this config is simply used to describe the required inputs and outputs. The same `JobConf` instance can be shared between the source and the

sink.

```
JobConf jobConfig = new JobConf();
jobConfig.setInputFormat(TextInputFormat.class);
jobConfig.setOutputFormat(TextOutputFormat.class);
TextOutputFormat.setOutputPath(jobConfig, "output-path");
TextInputFormat.addInputPath(jobConfig, "input-path");
```

The word count pipeline can then be expressed using HDFS as follows

```
Pipeline p = Pipeline.create();
p.drawFrom(HdfsSources.hdfs(jobConfig, (k, v) -> v.toString()))
 .flatMap(line -> traverseArray(delimiter.split(line.toLowerCase())).filter(w -> !w
.isEmpty()))
 .groupBy(wholeItem(), counting())
 .drainTo(HdfsSinks.hdfs(jobConfig));
```

Data Locality When Reading

Jet will split the input data across the cluster, with each processor instance reading a part of the input. If the Jet nodes are running along the HDFS datanodes, then Jet can make use of data locality by reading the blocks locally where possible. This can bring a significant increase in read speed.

Output

Each processor will write to a different file in the output folder identified by the unique processor id. The files will be in a temporary state until the job is completed and will be committed when the job is complete. For streaming jobs, they will be committed when the job is cancelled. We have plans to introduce a rolling sink for HDFS in the future to have better streaming support.

Dealing with Writables

Hadoop types implement their own serialization mechanism through the use of Writable. Jet provides an adapter to register a `Writable` for Hazelcast serialization without having to write additional serialization code. To use this adapter, you can register your own `Writable` types by extending `WritableSerializerHook` and registering the hook.

Hadoop JARs and Classpath

When submitting JARs along with a Job, sending Hadoop JARs should be avoided and instead Hadoop JARs should be present on the classpath of the running members. Hadoop JARs contain some JVM hooks and can keep lingering references inside the JVM long after the job has ended, causing memory leaks.

3.5.5. Kafka

Apache Kafka is a production-worthy choice of both source and sink for infinite stream processing jobs. It supports fault tolerance and snapshotting. The basic paradigm is that of a distributed publish/subscribe message queue. Jet's Kafka Source subscribes to a Kafka topic and the sink publishes events to a Kafka topic.

The following code will consume from topics `t1` and `t2` and then write to `t3`:

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "localhost:9092");
props.setProperty("key.serializer", StringSerializer.class.getCanonicalName());
props.setProperty("key.deserializer", StringDeserializer.class.getCanonicalName());
props.setProperty("value.serializer", IntegerSerializer.class.getCanonicalName());
props.setProperty("value.deserializer", IntegerDeserializer.class.getCanonicalName());
props.setProperty("auto.offset.reset", "earliest");

p.drawFrom(KafkaSources.kafka(props, "t1", "t2"))
 .drainTo(KafkaSinks.kafka(props, "t3"));
```

Using Kafka as a Source

The Kafka source emits entries of type `Map.Entry<Key,Value>` which can be transformed using an optional mapping function. It never completes. The job will end only if explicitly cancelled or aborted due to an error.

Internally Jet creates one `KafkaConsumer` per `Processor` instance using the supplied properties. Jet uses manual partition assignment to arrange the available Kafka partitions among the available processors and will ignore the `group.id` property.

Currently there is a requirement that the global parallelism of the Kafka source be at most the number of partitions you are subscribing to. The local parallelism of the Kafka source is 2 and if your Jet cluster has 4 members, this means that a minimum of 8 Kafka partitions must be available.

If any new partitions are added while the job is running, Jet will automatically assign them to the existing processors and consume them from the beginning.

Processing Guarantees

The Kafka source supports snapshots. Upon each snapshot it saves the current offset for each partition. When the job is restarted from a snapshot, the source can continue reading from the saved offset.

If snapshots are disabled, the source will commit the offset of the last record it read to the Kafka cluster. Since the fact that the source read an item doesn't mean that the whole Jet pipeline processed it, this doesn't guarantee against data loss.

Using Kafka as a Sink

The Kafka sink creates one `KafkaProducer` per cluster member and shares it among all the sink processors on that member. You can provide a mapping function that transforms the items the sink receives into `ProducerRecord`s.

3.6. Practical Considerations

3.6.1. Remember that a Jet Job is Distributed

The API to submit a job to Jet is in a way deceptively simple: "just call a method." As long as you're toying around with Jet instances started locally in a single JVM, everything will indeed work. However, as soon as you try to deploy to an actual cluster, you'll face the consequences of the fact that your job definition must travel over the wire to reach remote members which don't have your code on their classpath.

Your custom code must be packaged with the Jet job. For simple examples you can have everything in a single class and use code like this:

```
class JetExample {
    static Job createJob(JetInstance jet) {
        JobConfig jobConfig = new JobConfig();
        jobConfig.addClass(JetExample.class);
        return jet.newJob(createPipeline(), jobConfig);
    }
    ...
}
```

If you forget to do this, or don't add all the classes involved, you may get a quite confusing exception:

```
java.lang.ClassCastException:
cannot assign instance of java.lang.invoke.SerializedLambda
to field com.hazelcast.jet.core.ProcessorMetaSupplier$1.val$addressToSupplier
of type com.hazelcast.jet.function.DistributedFunction
in instance of com.hazelcast.jet.core.ProcessorMetaSupplier$1
```

`SerializedLambda` actually declares `readResolve()`, which would normally transform it into an instance of the correct functional interface type. If this method throws an exception, Java doesn't report it but keeps the `SerializedLambda` instance and continues the deserialization. Later in the process it will try to assign it to a field whose type is the target type of the lambda (`DistributedFunction` in the example above) and at that point it will fail with the `ClassCastException`. So, if you see this kind of error, double-check the list of classes you have added to the Jet job.

For more complex jobs it will become more practical to first package the job in a JAR and then use a

command-line utility to submit it, as explained next.

3.6.2. Submit a Job from the Command Line

Jet comes with the `jet-submit.sh` script, which allows you to submit a Jet job packaged in a JAR file. You can find it in the Jet distribution zipfile, in the `bin` directory. On Windows use `jet-submit.bat`. To use it, follow these steps:

- Write your `main()` method and your Jet code the usual way, except for calling `JetBootstrap.getInstance()` to acquire a Jet client instance (instead of `Jet.newJetClient()`).
- Create a runnable JAR which declares its `Main-Class` in
- `MANIFEST.MF`.
- Run your JAR, but instead of `java -jar jetjob.jar` use `jet-submit.sh jetjob.jar`.
- The script will create a Jet client and configure it from `hazelcast-client.xml` located in the `config` directory of Jet's distribution. Adjust that file to suit your needs.

For example, write a class like this:

```
public class CustomJetJob {
    public static void main(String[] args) {
        JetInstance jet = JetBootstrap.getInstance();
        jet.newJob(buildPipeline()).join();
    }

    static Pipeline buildPipeline() {
        // ...
    }
}
```

After building the JAR, submit the job:

```
$ jet-submit.sh jetjob.jar
```

3.6.3. Watch out for Capturing Lambdas

A typical Jet pipeline involves lambda expressions. Since the whole pipeline definition must be serialized to be sent to the cluster, the lambda expressions must be serializable as well. The Java standard provides an essential building block: if the static type of the lambda is a subtype of `Serializable`, you will automatically get a lambda instance that can serialize itself.

None of the functional interfaces in the JDK extend `Serializable`, so we had to mirror the entire `java.util.function` package in our own `com.hazelcast.jet.function` with all the interfaces subtyped and made `Serializable`. Each subtype has the name of the original with `Distributed` prepended. For

example, a `DistributedFunction` is just like `Function`, but implements `Serializable`. We use these types everywhere in the Pipeline API.

As always with this kind of magic, auto-serializability of lambdas has its flipside: it is easy to overlook what's going on.

If the lambda references a variable in the outer scope, the variable is captured and must also be serializable. If it references an instance variable of the enclosing class, it implicitly captures `this` so the entire class will be serialized. For example, this will fail because `JetJob` doesn't implement `Serializable`:

```
class JetJob {
    private String instanceVar;

    Pipeline buildPipeline() {
        Pipeline p = Pipeline.create();
        p.drawFrom(readList("input"))
         // Refers to instanceVar, capturing "this", but JetJob is not
         // Serializable so this call will fail.
         .filter(item -> item.equals(instanceVar));
        return p;
    }
}
```

Just adding `implements Serializable` to `JetJob` would be a viable workaround here. However, consider something just a bit different:

```
class JetJob {
    private String instanceVar;
    private OutputStream fileOut; // a non-serializable field

    Pipeline buildPipeline() {
        Pipeline p = Pipeline.create();
        p.drawFrom(readList("input"))
         // Refers to instanceVar, capturing "this". JetJob is declared
         // Serializable, but has a non-serializable field and this fails.
         .filter(item -> item.equals(instanceVar));
        return p;
    }
}
```

Even though we never refer to `fileOut`, we are still capturing the entire `JetJob` instance. We might mark `fileOut` as `transient`, but the sane approach is to avoid referring to instance variables of the surrounding class. This can be simply achieved by assigning to a local variable, then referring to that variable inside the lambda:

```
class JetJob {
    private String instanceVar;

    Pipeline buildPipeline() {
        Pipeline p = Pipeline.create();
        String findMe = instanceVar;
        p.drawFrom(readList("input"))
         // By referring to the local variable "findMe" we avoid
         // capturing "this" and the job runs fine.
         .filter(item -> item.equals(findMe));
        return p;
    }
}
```

Another common pitfall is capturing an instance of `DateTimeFormatter` or a similar non-serializable class:

```
DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern("HH:mm:ss.SSS")
                         .withZone(ZoneId.systemDefault());
Pipeline p = Pipeline.create();
ComputeStage<Long> src = p.drawFrom(readList("input"));
// Captures the non-serializable formatter, so this fails
src.map((Long tstamp) -> formatter.format(Instant.ofEpochMilli(tstamp)));
```

Sometimes we can get away by using one of the preconfigured formatters available in the JDK:

```
// Accesses the static final field ISO_LOCAL_TIME. Static fields are
// not subject to lambda capture, they are dereferenced when the code
// runs on the target machine.
src.map((Long tstamp) ->
    DateTimeFormatter.ISO_LOCAL_TIME.format(
        Instant.ofEpochMilli(tstamp).atZone(ZoneId.systemDefault())));
```

This refers to a `static final` field in the JDK, so the instance is available on any JVM. A similar approach is to declare our own `static final` field; however in that case we must add the declaring class as a job resource:

```
class JetJob {

    // Our own static field
    private static final DateTimeFormatter formatter =
            DateTimeFormatter.ofPattern("HH:mm:ss.SSS")
                             .withZone(ZoneId.systemDefault());

    Pipeline buildPipeline() {
        Pipeline p = Pipeline.create();
        ComputeStage<Long> src = p.drawFrom(readList("input"));
        src.map((Long tstamp) -> formatter.format(Instant.ofEpochMilli(tstamp)));
        return p;
    }

    // The job will fail unless we attach the JetJob class as a
    // resource, making the formatter instance available at the
    // target machine.
    void runJob(JetInstance jet) throws Exception {
        JobConfig c = new JobConfig();
        c.addClass(JetJob.class);
        jet.newJob(buildPipeline(), c).join();
    }
}
```

3.6.4. Standard Java Serialization is Slow

When it comes to serializing the description of a Jet job, performance is not critical. However, for the data passing through the pipeline, the cost of the serialize-deserialize cycle can easily dwarf the cost of actual data transfer, especially on high-end LANs typical for data centers. In this context the performance of Java serialization is so poor that it regularly becomes the bottleneck. This is due to its heavy usage of reflection, overheads in the serialized form, etc.

Since Hazelcast IMDG faced the same problem a long time ago, we have mature support for optimized custom serialization and in Jet you can use it for stream data. In essence, you must implement a `StreamSerializer` for the objects you emit from your processors and register it in Jet configuration:

```
SerializerConfig serializerConfig = new SerializerConfig()
        .setImplementation(new MyItemSerializer())
        .setTypeClass(MyItem.class);
JetConfig config = new JetConfig();
config.getHazelcastConfig().getSerializationConfig()
      .addSerializerConfig(serializerConfig);
JetInstance jet = Jet.newJetInstance(config);
```

Consult the chapter on custom serialization in Hazelcast IMDG's reference manual for more details.

Note the limitation implied here: the serializers must be registered with Jet on startup because this is how it is supported in Hazelcast IMDG. There is a plan to improve this and allow serializers to be registered on individual Jet jobs.

3.7. Logging and Debugging

3.7.1. Configuring Logging

Jet, like Hazelcast IMDG does not depend on a specific logging framework and has built-in adapters for a variety of logging frameworks. You can also write a new adapter to integrate with loggers Jet doesn't natively support. To use one of the built-in adapters, set the `hazelcast.logging.type` property to one of the following:

- `jdk`: java.util.logging (default)
- `log4j`: Apache Log4j
- `log4j2`: Apache Log4j 2
- `slf4j`: SLF4J
- `none`: Turn off logging

For example, to configure Jet to use Log4j, you can do one of the following:

```
System.setProperty("hazelcast.logging.type", "log4j");
```

or

```
JetConfig config = new JetConfig() ;
config.getHazelcastConfig().setProperty( "hazelcast.logging.type", "log4j" );
```

For more detailed information about how to configure logging, please refer to the IMDG reference manual.

3.7.2. Inspecting Output of Individual Stages

When building pipelines, it's often useful to see what the output of each stage is. This can be achieved by using the `peek()` stage. For example:

```
p.drawFrom(Sources.<Long, String>map(...))
   .flatMap(e -> traverseArray(delimiter.split(e.getValue().toLowerCase())))
   .filter(word -> !word.isEmpty())
   .groupBy(wholeItem(), counting())
   .peek()
   .drainTo(Sinks.map(COUNTS));
```

In this pipeline, the output of the `groupBy` stage will be logged using the configured logging framework:

```
16:05:29,650  INFO || -
[com.hazelcast.jet.impl.processor.PeekWrappedP.groupByKey.cd6373be.stage2#0]
hz._hzInstance_1_jet.jet.cooperative.thread-1 - [10.0.1.3]:5701 [jet] [0.6-SNAPSHOT]
Output to 0: accusers=6
16:05:29,650  INFO || -
[com.hazelcast.jet.impl.processor.PeekWrappedP.groupByKey.cd6373be.stage2#0]
hz._hzInstance_1_jet.jet.cooperative.thread-1 - [10.0.1.3]:5701 [jet] [0.6-SNAPSHOT]
Output to 0: mutability=2
16:05:29,650  INFO || -
[com.hazelcast.jet.impl.processor.PeekWrappedP.groupByKey.cd6373be.stage2#0]
hz._hzInstance_1_jet.jet.cooperative.thread-1 - [10.0.1.3]:5701 [jet] [0.6-SNAPSHOT]
Output to 0: lovely=53
```

The logger name of `com.hazelcast.jet.impl.processor.PeekWrappedP.groupByKey.cd6373be.stage2#0` can be decomposed as follows:

- `com.hazelcast.jet.impl.processor.PeekWrappedP`: class of the processor writing the log message
- `groupByKey.cd6373be.stage2`: Name of the vertex the processor belongs to
- `#0`: the unique index (per vertex) of the processor instance

Keep in mind that this can create a large amount of output when dealing with large volumes of data, and should strictly be used in a non-production environment.

For more information about logging when using the Core API, see the Best Practices section.

Chapter 4. Comparison of Jet APIs

Hazelcast Jet defines several kinds of API, each of which can be used to build computation jobs. However, there are significant differences between them.

Pipeline API is the one you should use unless you have a specific reason to look at others. It is "the Jet API". Powerful and expressive, yet quite simple to grasp and become productive in.

A secondary choice is Jet's implementation of the java.util.stream API], which some users already familiar with the JDK implementation may find appealing. However, it is less expressive and also suffers from a paradigm mismatch: its contract forces us to deliver the job's result in the return value, which is a cumbersome way to interact with a distributed computation system. The return value's scope is restricted to the scope of the Java variable that holds it, but the actual results remain in the Jet cluster, leading to possible memory leaks if not handled with care.

The most fundamental API is the Core API. All the other APIs are different front ends to it. It establishes strong low-level abstractions that expose all the mechanics of Jet's computation. While it can definitely be used to build a Jet job "from scratch", it is quite unwieldy to do so because for many aspects there is only one right way to set them up and many wrong ones.

This chart summarizes the main differences between the APIs.

Table 2. Differences between the APIs

	Pipeline API	**java.util.stream**	**Core API (DAG)**
When to Use	First choice to use Jet. Build rich data pipelines on a variety of sources and sinks.	Entry-level usage, simple transform-aggregate operations on IMap and IList.	Expert-level API to: * fine-tune a performance-critical computation * build a new high-level API or DSL * implement a custom source or sink * integrate with other libraries or frameworks
Expressiveness	High	Medium	Low
Works with all sources and sinks		(*)	
Transforms (map, flat map, filter)			
Aggregations		(**)	
Joins and forks			
Processing bounded data (batch)			

	Pipeline API	java.util.stream	Core API (DAG)
Processing unbounded data (streaming)	(*)		

*: Any source can be used with j.u.stream, but only `IMap` and `IList` sinks are supported.

**: `j.u.stream` only supports grouping on one input, co-grouping is not supported. Furthermore, aggregation is a terminal operation and additional transforms can't be applied to aggregation results.

*: Windowing support will be added in 0.6.

Chapter 5. Configuration

You can configure Hazelcast Jet either programmatically or declaratively (XML).

5.1. Programmatic Configuration

Programmatic configuration is the simplest way to configure Jet. You instantiate a `JetConfig` object and set the desired properties. For example, the following will configure Jet to use only two threads for cooperative execution:

```
JetConfig config = new JetConfig();
config.getInstanceConfig().setCooperativeThreadCount(2);
JetInstance jet = Jet.newJetInstance(config);
```

5.2. Declarative Configuration

If you don't pass an explicit `JetConfig` object when constructing a Jet instance, it will look for an XML configuration file in the following locations (in that order):

1. Check the system property `hazelcast.jet.config`. If the value is set and starts with `classpath:`, Jet treats it as a classpath resource. Otherwise it treats it as a file pathname.
2. Check for the presence of `hazelcast-jet.xml` in the working directory.
3. Check for the presence of `hazelcast-jet.xml` in the classpath.
4. If all the above checks fail, then Jet loads the default XML configuration that's packaged in the Jet JAR file.

An example configuration looks like the following:

```
<hazelcast-jet xsi:schemaLocation="http://www.hazelcast.com/schema/jet-config hazelcast-
jet-config-0.5.xsd"
               xmlns="http://www.hazelcast.com/schema/jet-config"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <instance>
        <!-- number of threads to use for DAG execution -->
       <cooperative-thread-count>8</cooperative-thread-count>
        <!-- frequency of flow control packets, in milliseconds -->
       <flow-control-period>100</flow-control-period>
        <!-- working directory to use for placing temporary files -->
       <temp-dir>/var/tmp/jet</temp-dir>
        <!-- number of backups for job specifics maps -->
       <backup-count>1</backup-count>
    </instance>
    <properties>
       <property name="custom.property">custom property</property>
    </properties>
    <edge-defaults>
        <!-- number of available slots for each concurrent queue between two vertices -->
       <queue-size>1024</queue-size>

        <!-- maximum packet size in bytes, only applies to distributed edges -->
       <packet-size-limit>16384</packet-size-limit>

        <!-- target receive window size multiplier, only applies to distributed edges -->
       <receive-window-multiplier>3</receive-window-multiplier>
    </edge-defaults>
    <!-- custom properties which can be read within a ProcessorSupplier -->
</hazelcast-jet>
```

The following are the configuration elements for Hazelcast Jet:

- Cooperative Thread Count: Maximum number of cooperative threads to be used for execution of jobs. Its default value is `Runtime.getRuntime().availableProcessors()`
- Temp Directory: Directory where temporary files will be placed, such as JAR files submitted by clients. Jet will create a temp directory, which will be deleted on exit.
- Flow Control Period: While executing a Jet job there is the issue of regulating the rate at which one member of the cluster sends data to another member. The receiver will regularly report to each sender how much more data it is allowed to send over a given DAG edge. This option sets the length (in milliseconds) of the interval between flow-control packets. Its default value is 100ms.
- Backup Count: Sets the number of synchronous backups for storing job metadata and snapshots. Maximum allowed value is 6. Its default value is 1.
- Edge Defaults: The default values to be used for all edges. Please see the section on Tuning Edges.

5.3. Configure the Underlying Hazelcast Instance

Each Jet member or client has its underlying Hazelcast member or client. Please refer to the Hazelcast Reference Manual for specific configuration options for Hazelcast IMDG.

5.3.1. Programmatic

The underlying Hazelcast IMDG member can be configured as follows:

```
JetConfig jetConfig = new JetConfig();
jetConfig.getHazelcastConfig().getGroupConfig().setName("test");
JetInstance jet = Jet.newJetInstance(jetConfig);
```

The underlying Hazelcast IMDG client can be configured as follows:

```
`java ClientConfig clientConfig = new ClientConfig();
clientConfig.getGroupConfig().setName("test"); JetInstance jet = Jet.newJetClient(clientConfig); `
```

5.3.2. Declarative

Hazelcast IMDG can also be configured declaratively as well. Please refer to the Hazelcast Reference Manual for information on how to do this.

Chapter 6. Distributed Implementation of java.util.stream API

Hazelcast Jet adds distributed `java.util.stream` support for Hazelcast IMap, ICache and IList data structures.

For extensive information about `java.util.stream` API please refer to the official javadocs.

6.1. Simple Example

```
JetInstance jet = Jet.newJetInstance();
IStreamMap<String, Integer> map = jet.getMap("latitudes");
map.put("London", 51);
map.put("Paris", 48);
map.put("NYC", 40);
map.put("Sydney", -34);
map.put("Sao Paulo", -23);
map.put("Jakarta", -6);

map.stream().filter(e -> e.getValue() < 0).forEach(System.out::println);
```

In addition to Hazelcast data structures any source processor can be used to create a distributed stream.

```
JetInstance jet = Jet.newJetInstance();
ProcessorSupplier processorSupplier = SourceProcessors.readFilesP("path", UTF_8, "*");

IList<String> sink = DistributedStream
        .<String>fromSource(jet, ProcessorMetaSupplier.of(processorSupplier))
        .flatMap(line -> Arrays.stream(line.split(" ")))
        .collect(DistributedCollectors.toIList("sink"));


sink.forEach(System.out::println);
```

Java specifies that a stream computation starts upon invoking the terminal operation on it (such as `forEach()`). At that point Jet converts the expression into a Core API DAG and submits it for execution.

6.2. Distributed Collectors

Like with the functional interfaces, Jet also includes distributed versions of the classes found in

`java.util.stream.Collectors`. These can be reached from the `DistributedCollectors` utility class. However, keep in mind that the collectors such as `toMap()`, `toCollection()`, `toList()`, and `toArray()` create a local data structure and load all the results into it. This works fine with the regular JDK streams, where everything is local, but usually fails badly in the context of a distributed computing job.

For example the following innocent-looking code can easily cause out-of-memory errors because the whole distributed map will need to be held in the memory at a single place:

```
// get a distributed map with 5GB per member on a 10-member cluster
IStreamMap<String, String> map = jet.getMap("large_map");
// now try to build a HashMap of 50GB
Map<String, String> result = map.stream()
                                .map(e -> e.getKey() + e.getValue())
                                .collect(toMap(v -> v, v -> v));
```

This is why Jet distinguishes between the standard `java.util.stream` collectors and the Jet-specific `Reducer`s. A `Reducer` puts the data into a distributed data structure and knows how to leverage its native partitioning scheme to optimize the access pattern across the cluster.

These are some of the `Reducer` implementations provided in Jet:

- `toIMap()`: Writes the data to a new Hazelcast `IMap`.
- `groupingByToIMap()`: Performs a grouping operation and then writes the results to a Hazelcast `IMap`. This uses a more efficient implementation than the standard `groupingBy()` collector and can make use of partitioning.
- `toIList()`: Writes the data to a new Hazelcast `IList`.

A distributed data structure is cluster-managed, therefore you can't just create one and forget about it; it will live on until you explicitly destroy it. That means it's inappropriate to use as a part of a data item inside a larger collection, a further consequence being that a `Reducer` is inappropriate as a downstream collector; that's where the JDK-standard collectors make sense.

6.3. Word Count

The word count example that was described in the Get Started section can be rewritten using the `java.util.stream` API as follows:

```
IMap<String, Long> counts = lines
                .stream()
                .flatMap(word -> Stream.of(word.split("\\W+")))
                .collect(DistributedCollectors.toIMap(w -> w, w -> 1L, (left, right) ->
left + right));
```

Chapter 7. Under the Hood

7.1. How Distributed Computing Works in Jet

In this section we will take a deep dive into the fundamentals of distributed computing and Jet's specific take on it. We will do this by dissecting one specific problem: the Word Count. This is how you'd describe it in the Jet Pipeline API:

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

We'll step back from this and start from the single-threaded Java code that solves the problem for a basic data structure such as an `ArrayList` and gradually move on to a formulation that allows us to solve it for a data source distributed over the whole cluster, efficiently making use of all the available CPU power. Towards the end of the section we'll show you the Core API code that directly builds the DAG we designed.

Here is the single-threaded code that counts the words in a `List` of lines of text:

```
List<String> lines = ... // a pre-existing list
Map<String, Long> counts = new HashMap<>();
for (String line : lines) {
    for (String word : line.toLowerCase().split("\\W+")) {
        if (!word.isEmpty()) {
            counts.merge(word, 1L, (count, one) -> count + one);
        }
    }
}
```

To move us closer to how this computation is expressed in Jet, let's rewrite it in terms of the Java Streams API, still single-threaded:

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

The Java Streams formulation gives us clear insight into the steps taken to process each data item:

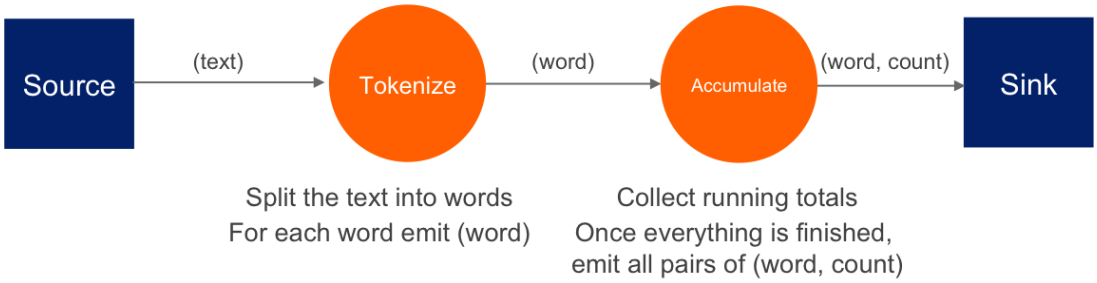
1. `lines.stream()`: read lines from the data source (we'll call this the "source" step).
2. `flatMap()`+`filter()`: split each line into lowercase words, avoiding empty strings (the "tokenizer" step).
3. `collect()`: group equal words together and count them (the "accumulator" step).

Our first move will be modeling the computation as a DAG. We'll start with a single-threaded model and then make several transformation steps to it to reach a parallelized, distributed one, discussing at each step the concerns that arise and how to meet them.

Note that here we are describing a *batch* job: the input is finite and present in full before the job starts. Later on we'll present a *streaming* job that keeps processing an infinite stream forever, transforming it into another infinite stream.

7.1.1. Modeling Word Count in terms of a DAG

We can represent the steps outlined above as a DAG:



The simplest, single-threaded code (shown above) deals with each item as it is produced: the outer loop reads the lines, the inner loop that runs for each line deals with the words on that line, and inside the inner loop we populate the result map with running counts.

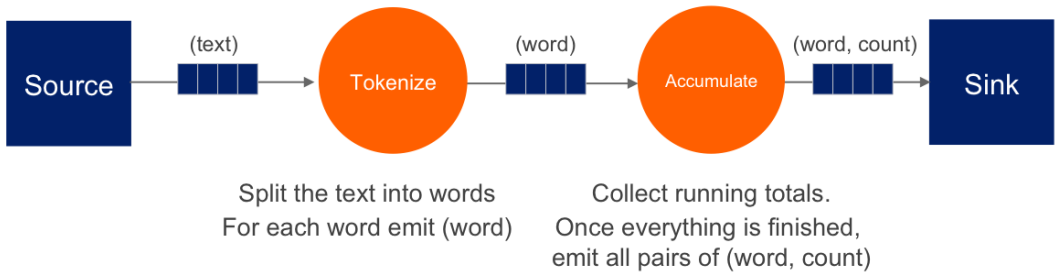
However, just by modeling the computation as a DAG, we've split the work into isolated steps with clear data interfaces between them. We can perform the same computation by running a separate thread for each step. Roughly speaking, these are the snippets the threads would be executing:

```
// Source thread
for (String line : readLines()) {
    emit(line);
}
```

```
// Tokenizer thread
for (String line : receive()) {
    for (String word : line.toLowerCase().split("\\W+")) {
        if (!word.isEmpty()) {
            emit(word);
        }
    }
}
```

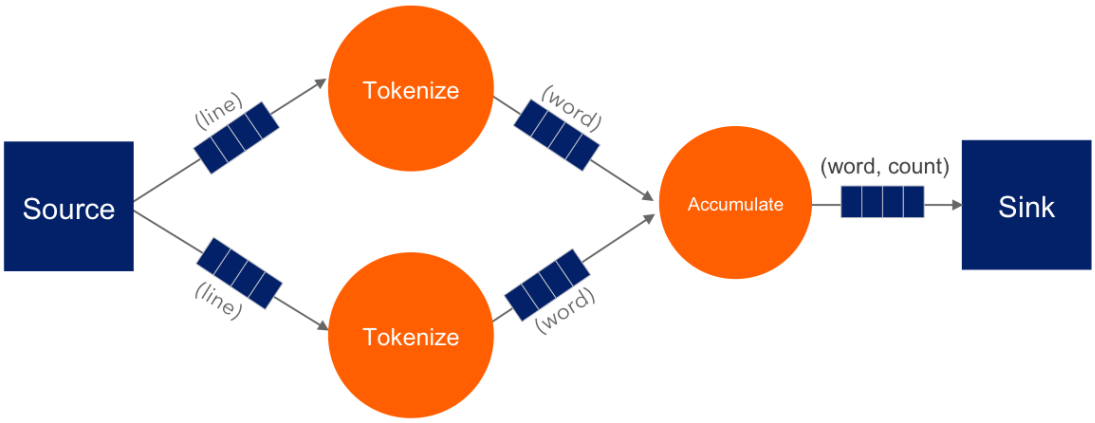
```
// Accumulator thread
Map<String, Long> counts = new HashMap<>();
for (String word : receive()) {
    counts.merge(word, 1L, (count, one) -> count + one);
}
// finally, when done receiving:
for (Entry<String, Long> wordAndCount : counts.entrySet()) {
    emit(wordAndCount);
}
```

The source loop feeds the tokenizer loop over a concurrent queue, the tokenizer feeds the accumulator loop, and after the accumulator is done receiving, it emits its results to the sink. Diagrammatically it looks like this:



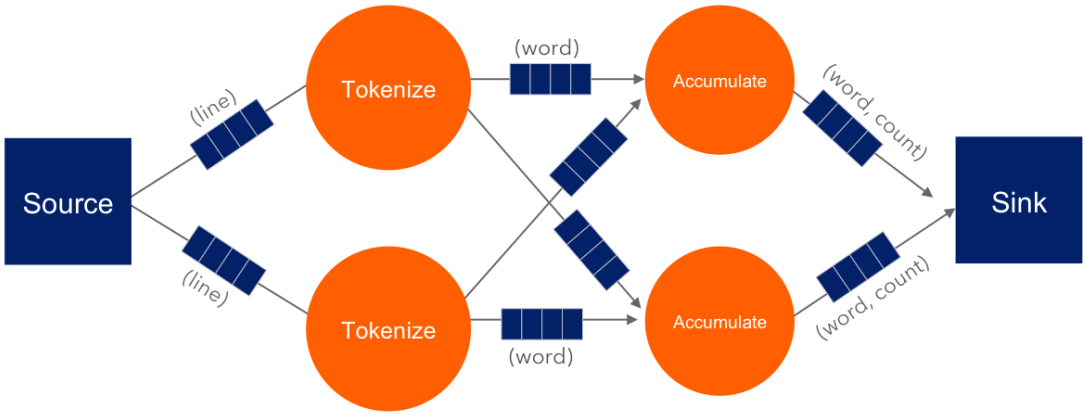
This transformation brought us a *pipelined* architecture: while the tokenizer is busy with the regex work, the accumulator is updating the map using the data the tokenizer is done with; and the source and sink stages are pumping the data from/to the environment. Our design is now able to engage more than one CPU core and will complete that much sooner; however, we're still limited by the number of vertices. We'll be able utilize two or three cores regardless of how many are available. To move forward we must try to parallelize the work of each individual vertex.

Given that our input is an in-memory list of lines, the bottleneck occurs in the processing stages (tokenizing and accumulating). Let's first attack the tokenizing stage: it is a so-called "embarrassingly parallelizable" task because the processing of each line is completely self-contained. At this point we have to make a clear distinction between the notions of *vertex* and *processor*: there can be several processors doing the work of a single vertex. Let's add another tokenizing processor:



The input processor can now use all the available tokenizers as a pool and submit to any one whose queue has some room.

The next step is parallelizing the accumulator vertex, but this is trickier: accumulators count word occurrences so using them as a pool will result in each processor observing almost all distinct words (entries taking space in its hashtable), but the counts will be partial and will need combining. The common strategy to reduce memory usage is to ensure that all occurrences of the same word go to the same processor. This is called "data partitioning" and in Jet we'll use a *partitioned edge* between the tokenizer and the accumulator:



As a word is emitted from the tokenizer, it goes through a "switchboard" stage where it's routed to the correct downstream processor. To determine where a word should be routed, we can calculate its hashcode and use the lowest bit to address either accumulator 0 or accumulator 1.

At this point we have a blueprint for a fully functional parallelized computation job which can max out all the CPU cores given enough instances of tokenizing and accumulating processors. The next challenge is making this work across machines.

For starters, our input can no longer be a simple in-memory list because that would mean each machine processes the same data. To exploit the cluster as a unified computation device, each cluster member must observe only a slice of the dataset. Given that a Jet instance is also a fully functional

Hazelcast IMDG instance and a Jet cluster is also a Hazelcast IMDG cluster, the natural choice is to pre-load our data into an `IMap`, which will be automatically partitioned and distributed across the members. Now each Jet member can just read the slice of data that was stored locally on it.

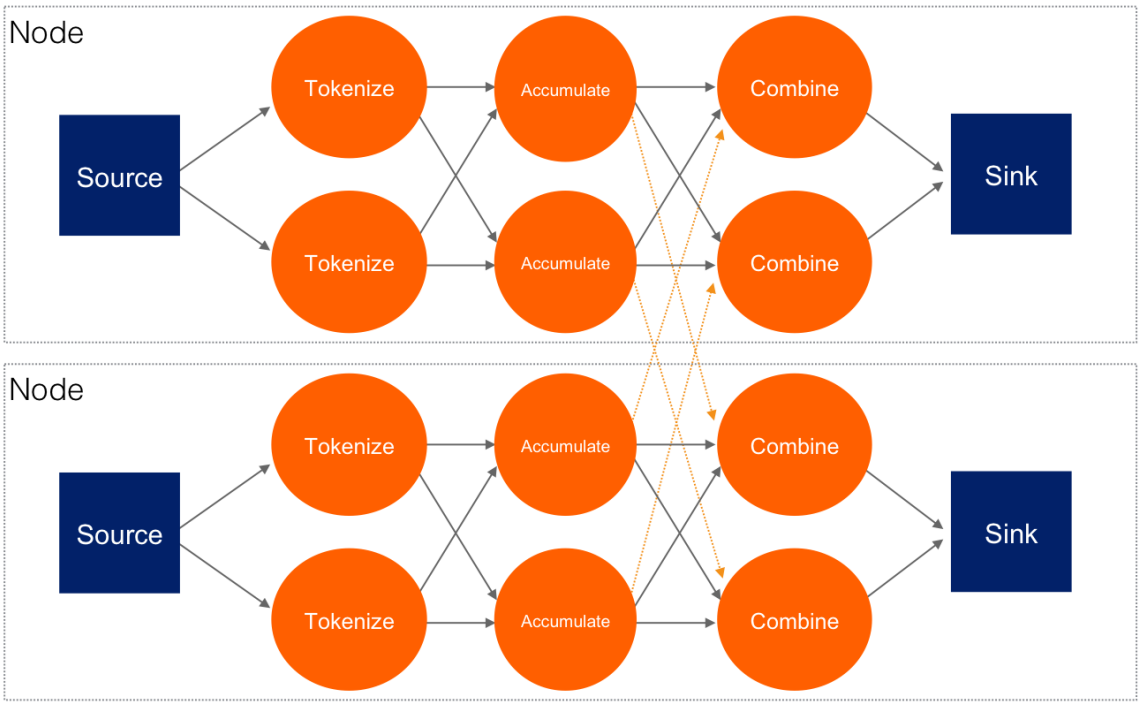
When run in a cluster, Jet will instantiate a replica of the whole DAG on each member. On a two-member cluster there will be two source processors, four tokenizers, and so on. The trickiest part is the partitioned edge between tokenizer and accumulator: each accumulator is supposed to receive its own subset of words. That means that, for example, a word emitted from tokenizer 0 will have to travel across the network to reach accumulator 3, if that's the one that happens to own it. On average we can expect every other word to need network transport, causing both serious network traffic and serialization/deserialization CPU load.

There is a simple trick we can employ to avoid most of this traffic, closely related to what we pointed above as a source of problems when parallelizing locally: members of the cluster can be used as a pool, each doing its own partial word counts, and then send their results to a combining vertex. Note that this means sending only one item per distinct word. Here's the rough equivalent of the code the combining vertex executes:

```
// Combining vertex
Map<String, Long> combined = new HashMap<>();
for (Entry<String, Long> wordAndCount : receive()) {
    combined.merge(wordAndCount.getKey(), wordAndCount.getValue(),
                    (accCount, newCount) -> accCount + newCount);
}
// finally, when done receiving:
for (Entry<String, Long> wordAndCount : combined.entrySet()) {
    emit(wordAndCount);
}
```

As noted above, such a scheme takes more memory due to more hashtable entries on each member, but it saves network traffic (an issue we didn't have within a member). Given that memory costs scale with the number of distinct keys (english words in our case), the memory cost is more-or-less constant regardless of how much book material we process. On the other hand, network traffic scales with the total data size so the more material we process, the more we save on network traffic.

Jet distinguishes between *local* and *distributed* edges, so we'll use a *local partitioned* edge for `tokenize`→`accumulate` and a *distributed partitioned* edge for `accumulate`→`combine`. With this move we've finalized our DAG design, which can be illustrated by the following diagram:



7.1.2. Implementing the DAG in Jet's Core API

Now that we've come up with a good DAG design, we can use Jet's Core API to implement it. We start by instantiating the DAG class and adding the source vertex:

```
DAG dag = new DAG();
Vertex source = dag.newVertex("source", SourceProcessors.readMapP("lines"));
```

Note how we can build the DAG outside the context of any running Jet instances: it is a pure POJO.

The source vertex will read the lines from the `IMap` and emit items of type `Map.Entry<Integer, String>` to the next vertex. The key of the entry is the line number, and the value is the line itself. The built-in map-reading processor will do just what we want: on each member it will read only the data local to that member.

The next vertex is the *tokenizer*, which does a simple "flat-mapping" operation (transforms one input item into zero or more output items). The low-level support for such a processor is a part of Jet's library, we just need to provide the mapping function:

```
// (lineNum, line) -> words
Pattern delimiter = Pattern.compile("\\W+");
Vertex tokenize = dag.newVertex("tokenize",
    Processors.flatMapP((Entry<Integer, String> e) ->
        traverseArray(delimiter.split(e.getValue().toLowerCase()))
              .filter(word -> !word.isEmpty()))
);
```

This creates a processor that applies the given function to each incoming item, obtaining zero or more output items, and emits them. Specifically, our processor accepts items of type `Entry<Integer, String>`, splits the entry value into lowercase words, and emits all non-empty words. The function must return a `Traverser`, which is a functional interface used to traverse a sequence of non-null items. Its purpose is equivalent to the standard Java `Iterator`, but avoids the cumbersome two-method API. Since a lot of support for cooperative multithreading in Hazelcast Jet deals with sequence traversal, this abstraction simplifies many of its aspects.

The next vertex will do the actual word count. We can use the built-in `accumulateByKey` processor for this:

```
// word -> (word, count)
Vertex accumulate = dag.newVertex("accumulate",
        Processors.accumulateByKeyP(wholeItem(), counting())
);
```

This processor maintains a hashtable that maps each distinct key to its accumulated value. We specify `wholeItem()` as the *key extractor* function: our input item is just the word, which is also the grouping key. The second argument is the kind of aggregate operation we want to perform: counting. We are relying on Jet's out-of-the-box definitions here, but it is easy to define your own aggregate operations and key extractors. The processor emits nothing until it has received all the input, and at that point it emits the hashtable as a stream of `Entry<String, Long>`.

Next is the combining step which computes the grand totals from individual members' contributions. This is the code:

```
// (word, count) -> (word, count)
Vertex combine = dag.newVertex("combine",
    Processors.combineByKeyP(counting())
);
```

`combineByKey` is designed to be used downstream of `accumulateByKey`, which is why it doesn't need an explicit key extractor. The aggregate operation must be the same as on `accumulateByKey`.

The final vertex is the sink; we want to store the output in another `IMap`:

```
Vertex sink = dag.newVertex("sink", SinkProcessors.writeMapP("counts"));
```

Now that we have all the vertices, we must connect them into a graph and specify the edge type as discussed in the previous section. Here's all the code at once:

```
dag.edge(between(source, tokenize))
   .edge(between(tokenize, accumulate)
           .partitioned(wholeItem(), Partitioner.HASH_CODE))
   .edge(between(accumulate, combine)
           .distributed()
           .partitioned(entryKey()))
   .edge(between(combine, sink));
```

Let's take a closer look at some of the edges. First, source to tokenizer:

```
.edge(between(tokenize, accumulate)
       .partitioned(wholeItem(), Partitioner.HASH_CODE))
```

We chose a *local partitioned* edge. For each word, there will be a processor responsible for it on each member so that no items must travel across the network. In the `partitioned()` call we specify two things: the function that extracts the partitioning key (`wholeItem()` - same as the grouping key extractor), and the policy object that decides how to compute the partition ID from the key. Here we use the built-in `HASH_CODE`, which will derive the ID from `Object.hashCode()`. As long as the the definitions of `equals()/hashCode()` on the key object match our expected notion of key equality, this policy is always safe to use on a local edge.

Next, the edge from the accumulator to the combiner:

```
.edge(between(accumulate, combine)
       .distributed()
       .partitioned(entryKey()))
```

It is *distributed partitioned*: for each word there is a single `combiner` processor in the whole cluster responsible for it and items will be sent over the network if needed. The partitioning key is again the word, but here it is the key part of the `Map.Entry<String, Long>`. We are using the default partitioning policy here (Hazelcast's own partitioning scheme). It is the slower-but-safe choice on a distributed edge. Detailed inspection shows that hashcode-based partitioning would be safe as well because all of `String`, `Long`, and `Map.Entry` have the hash function specified in their Javadoc.

You can acces a full, self-contained Java program with the above DAG code at the Hazelcast Jet code samples repository.

7.2. How Infinite Stream Processing Works In Jet

Continuing our deep dive into Jet's fundamentals we shall now move on to infinite stream processing. The major challenge in batch jobs was properly parallelizing/distributing a "group by key" operation. To solve it we introduced the idea of partitioning the data based on a formula that takes just the grouping key as input and can be computed independently on any member, always yielding the same result. In the context of infinite stream processing we have the same concern and solve it with the same means, but we also face some new challenges.

7.2.1. Stream Skew

We already introduced the concept of the watermark and how it imposes order onto a disordered data stream. Items arriving out of order aren't our only challenge; modern stream sources like Kafka are partitioned and distributed so "the stream" is actually a set of independent substreams, moving on in parallel. Substantial time difference may arise between events being processed on each one, but our system must produce coherent output as if there was only one stream. We meet this challenge by coalescing watermarks: as the data travels over a partitioned/distributed edge, we make sure the downstream processor observes the correct watermark value, which is the least of watermarks received from the contributing substreams.

7.2.2. Sliding and Tumbling Window

Many quantities, like "the current rate of change of a price" require you to aggregate your data over some time period. This is what makes the sliding window so important: it tracks the value of such a quantity in real time.

Calculating a single sliding window result can be quite computationally intensive, but we also expect it to slide smoothly and give a new result often, even many times per second. This is why we gave special attention to optimizing this computation.

We optimize especially heavily for those aggregate operations that have a cheap way of combining partial results and even more so for those which can cheaply undo the combining. For cheap combining you have to express your operation in terms of a commutative and associative (CA for short) function; to undo a combine you need the notion of "negating" an argument to the function. A great many operations can be expressed through CA functions: average, variance, standard deviation and linear regression are some examples. All of these also support the undoing (which we call *deduct*). The computation of extreme values (min/max) is an example that has CA, but no good notion of negation and thus doesn't support deducting.

This is the way we leverage the above properties: our sliding window actually "hops" in fixed-size steps. The length of the window is an integer multiple of the step size. Under such a definition, the *tumbling* window becomes just a special case with one step per window.

This allows us to divide the timestamp axis into *frames* of equal length and assign each event to its frame. Instead of keeping the event object, we immediately pass it to the aggregate operation's

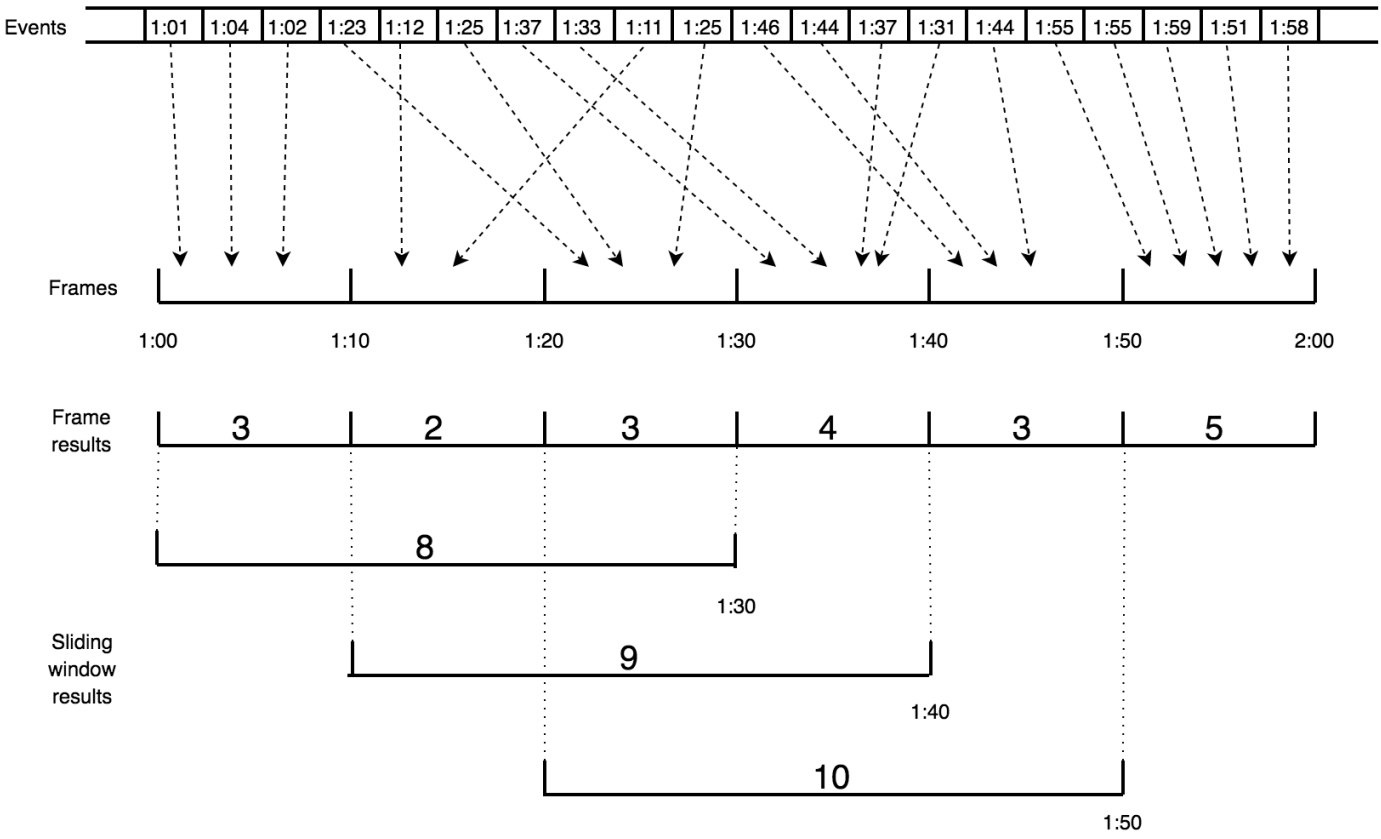
accumulate primitive. To compute a sliding window, we take all the frames covered by it and combine them. Finally, to compute the next window, we just *deduct* the trailing frame and *combine* the leading frame into the existing result.

Even without *deduct* the above process is much cheaper than the most naïve approach where you'd keep all data and recompute everything from scratch each time. After accumulating an item just once, the rest of the process has fixed cost regardless of input size. With *deduct*, the fixed cost approaches zero.

Example: 30-second Window Sliding by 10 Seconds

We'll now illustrate the above story with a specific example: we'll construct a 30-second window which slides by 10 seconds (i.e., three steps per window). The aggregate operation is to simply count the number of events. In the diagrams we label the events as *minutes:seconds*. This is the outline of the process:

1. Throw each event into its "bucket" (the frame whose time interval it belongs to).
2. Instead of keeping the items in the frame, just keep the item count.
3. Combine the frames into three different positions of the sliding window, yielding the final result: the number of events that occurred within the window's timespan.

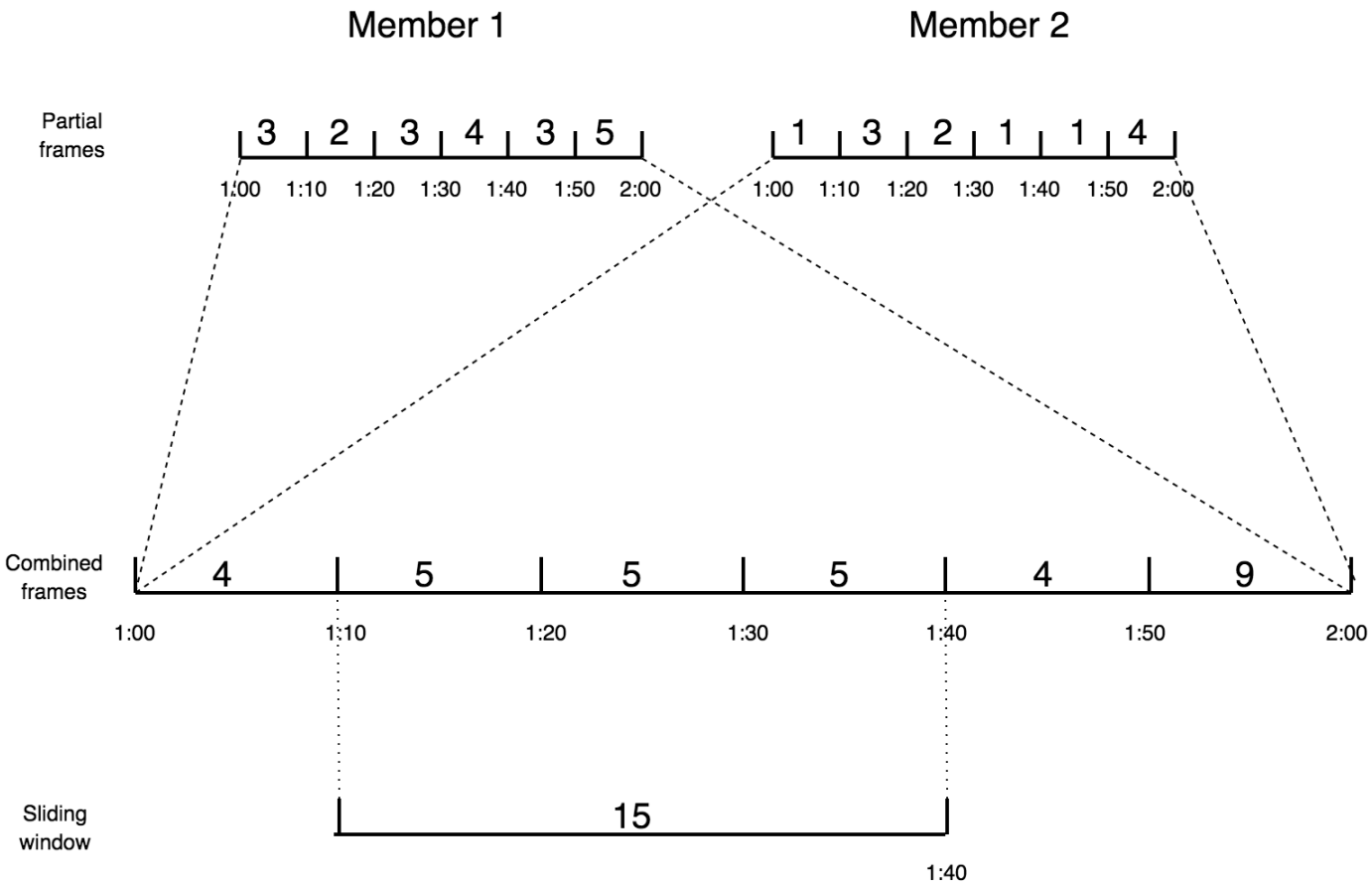


This would be a useful interpretation of the results: "At the time 1:30, the 30-second running average was 8/30 = 0.27 events per second. Over the next 20 seconds it increased to 10/30 = 0.33 events per second."

Keep in mind that the whole diagram represents what happens on just one cluster member and for just one grouping key. The same process is going on simultaneously for all the keys on all the members.

Two-stage aggregation

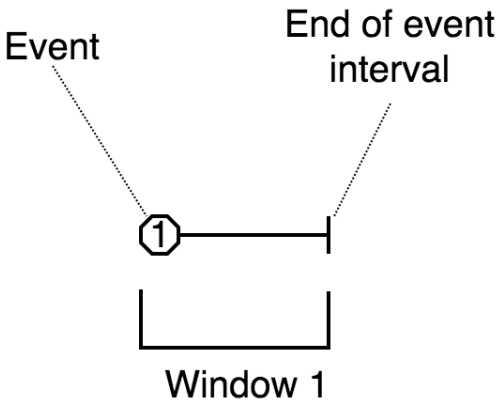
The concept of frame combining helps us implement two-stage aggregation as well. In the first stage the individual members come up with their partial results by frame and send them over a distributed edge to the second stage, which combines the frames with the same timestamp. After having combined all the partial frames from members, it combines the results along the event time axis into the sliding window.



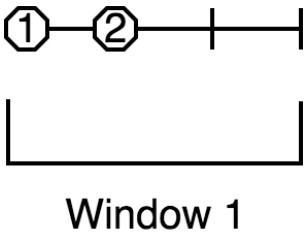
7.2.3. Session Window

In the abstract sense, the session window is a quite intuitive concept: it simply captures a burst of events. If no new events occur within the configured session timeout, the window closes. However, because the Jet processor encounters events out of their original order, this kind of window becomes quite tricky to compute.

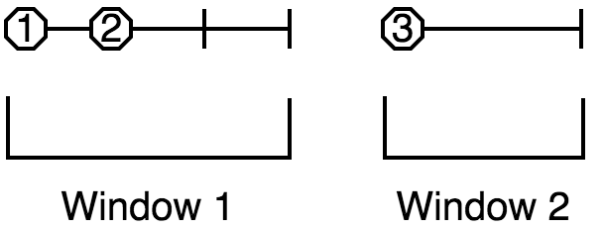
The way Jet computes the session windows is easiest to explain in terms of the *event interval*: the range `[eventTimestamp, eventTimestamp + sessionTimeout]`. Initially an event causes a new session window to be created, covering exactly the event interval.



A following event under the same key belongs to this window iff its interval overlaps it. The window is extended to cover the entire interval of the new event.

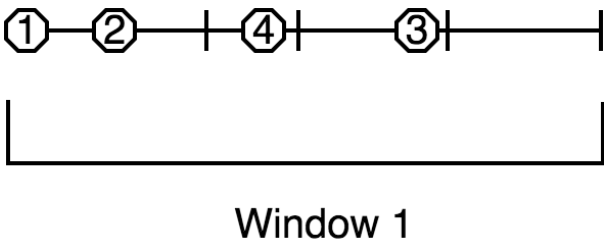


If the event intervals don't overlap, a new session window is created for the new event.




```
width="240"/>
```

An event may happen to belong to two existing windows if its interval bridges the gap between them; in that case they are combined into one.



Once the watermark has passed the closing time of a session window, Jet can close it and emit the result of its aggregation.

7.2.4. Distributed Snapshot

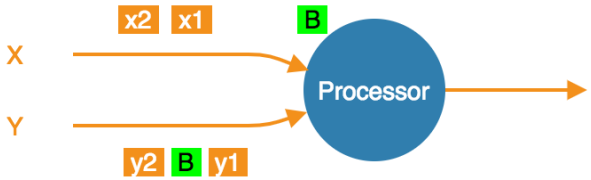
The technique Jet uses to achieve fault tolerance is called a "distributed snapshot", described in a paper by Chandy and Lamport. At regular intervals, Jet raises a global flag that says "it's time for another snapshot". All processors belonging to source vertices observe the flag, create a checkpoint on their source, and emit a barrier item to the downstream processors and resumes processing.

As the barrier item reaches a processor, it stops what it's doing and emits its state to the snapshot storage. Once complete, it forwards the barrier item to its downstream processors.

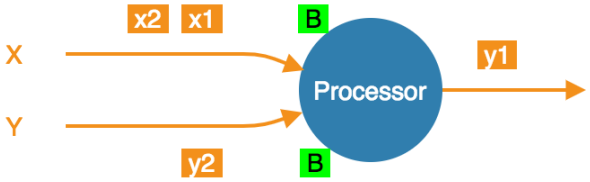
Due to parallelism, in most cases a processor receives data from more than one upstream processor. It will receive the barrier item from each of them at separate times, but it must start taking a snapshot at a single point in time. There are two approaches it can take, as explained below.

Exactly-Once Snapshotting

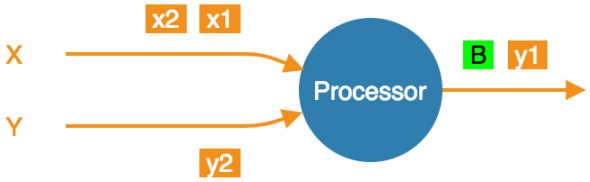
With *exactly-once* configured, as soon as the processor gets a barrier item in any input stream (from any upstream processor), it must stop consuming it until it gets the same barrier item in all the streams:



1. At the barrier in stream X, but not Y. Must not accept any more X items.



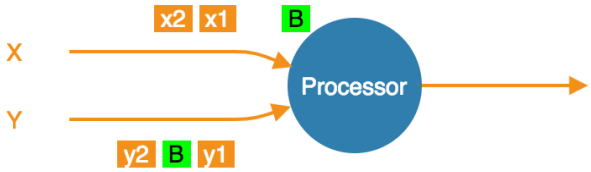
2. At the barrier in both streams, taking a snapshot.



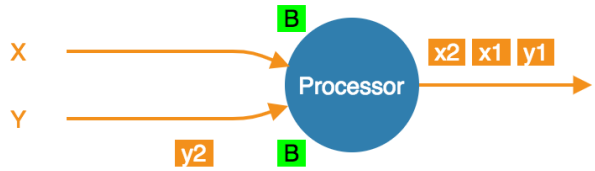
3. Snapshot done, barrier forwarded. Can resume consuming all streams.

At-Least-Once Snapshotting

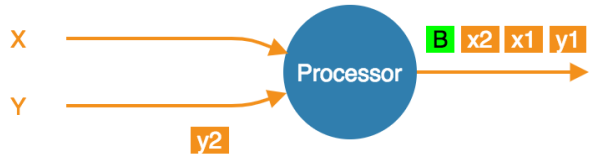
With *at-least-once* configured, the processor can keep consuming all the streams until it gets all the barriers, at which point it will stop to take the snapshot:



1. At the barrier in stream X, but not Y. Carry on consuming all streams.



2. At the barrier in both streams, already consumed `x1` and `x2`. Taking a snapshot.



3. Snapshot done, barrier forwarded.

Even though `x1` and `x2` occur after the barrier, the processor consumed and processed them, updating its state accordingly. If the computation job stops and restarts, this state will be restored from the snapshot and then the source will replay `x1` and `x2`. The processor will think it got two new items.

7.2.5. Rules of Watermark Propagation

Watermark objects are sent interleaved with other stream items, but are handled specially:

- The value of the watermark a processor emits must be strictly increasing. Jet will throw an exception if it detects a non-increasing watermark.
- When a processor receives and handles a watermark, it is automatically emitted to the outbox. Therefore there should be only one processor emitting watermarks in the pipeline.
- The watermark item is always broadcast, regardless of the edge type. This means that all N upstream processors send their watermark to all M downstream processors.
- The processor will observe only the highest watermark received from all upstream processors and from all upstream edges. This is called *watermark coalescing*.

Jet's internal class `WatermarkCoalescer` manages watermarks received from multiple inputs. As it receives watermark items from them, its duty is to decide when to forward the watermark downstream. This happens at two levels: * between multiple queues backing single edge * between multiple input edges to single processor

Idle inputs

A special object called *idle message* can be emitted from source processor when the processor sees no

events for configured *idle timeout*. This can happen in real life when some external partitions have no events while others do.

When an *idle message* is received from an input, that input will be excluded from watermark coalescing. This means that we will not wait to receive watermark from idle input. It will cause that other active inputs can be processed without any delay. When idle timeout is disabled and some processor doesn't emit any watermarks (because it sees no events), the processing will stall indefinitely (unless maximum retention is configured).

7.2.6. The Pitfalls of At-Least-Once Processing

In some cases *at-least-once* semantics can have consequences of quite an unexpected magnitude, as we discuss next.

Apparent Data Loss

Imagine a very simple kind of processor: it matches up the items that belong to a *pair* based on some rule. If it receives item A first, it remembers it. Later on, when it receives item B, it emits that fact to its outbound edge and forgets about the two items. It may also first receive B and wait for A.

Now imagine this sequence: `A → BARRIER → B`. In at-least-once the processor may observe both A and B, emit its output, and forget about them, all before taking the snapshot. After the restart, item B will be replayed because it occurred after the last barrier, but item A won't. Now the processor is stuck forever in a state where it's expecting A and has no idea it already got it and emitted that fact.

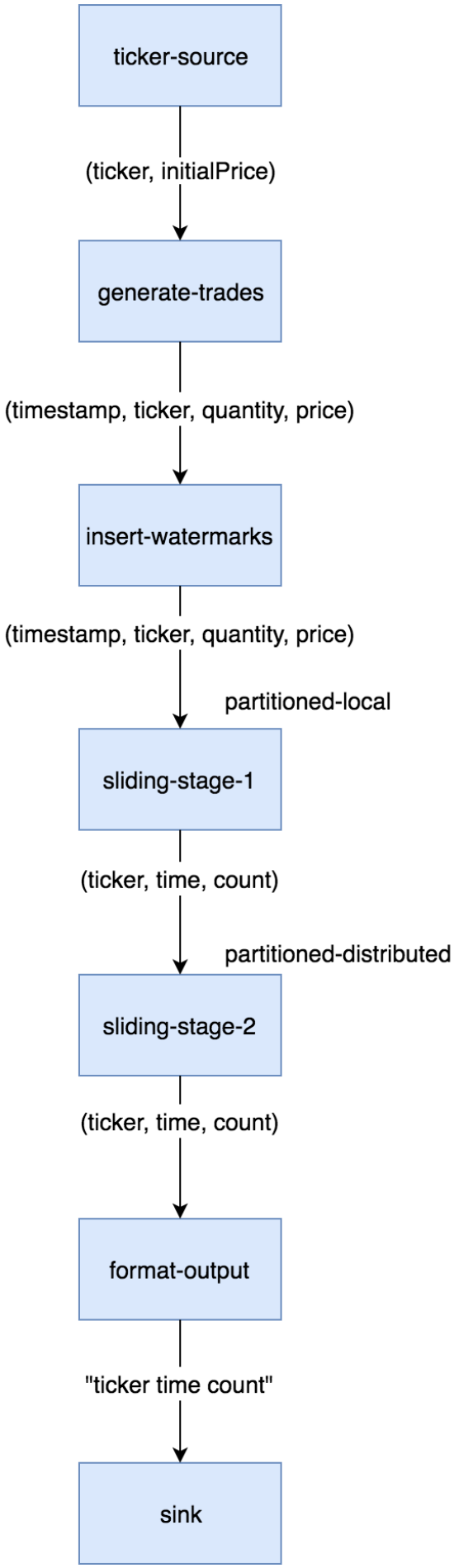
Problems similar to this may happen with any state the processor keeps until it has got enough information to emit the results and then forgets it. By the time it takes a snapshot, the post-barrier items will have caused it to forget facts about some pre-barrier items. After a restart it will behave as though it has never observed those pre-barrier items, resulting in behavior equivalent to data loss.

Non-Monotonic Watermark

One special case of the above story concerns watermark items. Thanks to watermark coalescing, processors are typically implemented against the invariant that the watermark value always increases. However, in *at-least-once* the post-barrier watermark items will advance the processor's watermark value. After the job restarts and the state gets restored to the snapshotted point, the watermark will appear to have gone back, breaking the invariant. This can again lead to apparent data loss.

7.3. Stream-Processing DAG and Code

For this example we'll build a simple Jet job that monitors trading events on a stock market, categorizes the events by stock ticker, and reports the number of trades per time unit (the time window). In terms of DAG design, not much changes going from batch to streaming. This is how it looks:


ticker-source
(ticker, initialPrice)
generate-trades
(timestamp, ticker, quantity, price)
insert-watermarks
(timestamp, ticker, quantity, price)
partitioned-local
sliding-stage-1
(ticker, time, count)
partitioned-distributed
sliding-stage-2
(ticker, time, count)
format-output
"ticker time count"
sink

We have the same cascade of source, two-stage aggregation, and sink. The source part consists of `ticker-source` that loads stock names (tickers) from a Hazelcast IMap and `generate-trades` that retains this list and randomly generates an infinite stream of trade events. A separate vertex is inserting watermark items needed by the aggregation stage and on the sink side there's another mapping vertex, `format-output`, that transforms the window result items into lines of text. The `sink` vertex writes these lines to a file.

Before we go on, let us point out that in the 0.5 release of Hazelcast Jet, the Pipeline API is still in infancy and doesn't support all the features needed for stream processing. Therefore the following example is given only in the Core API; with the next release we'll be able to present the much simpler code to do it in the Pipelines API.

If you studied the DAG-building code for the Word Count job, this code should look generally familiar:

```
WindowDefinition windowDef = WindowDefinition.slidingWindowDef(
        SLIDING_WINDOW_LENGTH_MILLIS, SLIDE_STEP_MILLIS);
Vertex tickerSource = dag.newVertex("ticker-source",
        SourceProcessors.readMapP(GenerateTradesP.TICKER_MAP_NAME));
Vertex generateTrades = dag.newVertex("generate-trades",
        GenerateTradesP.generateTradesP(TRADES_PER_SEC_PER_MEMBER));
Vertex insertWatermarks = dag.newVertex("insert-watermarks",
        Processors.insertWatermarksP(
                Trade::getTime,
                withFixedLag(GenerateTradesP.MAX_LAG),
                emitByFrame(windowDef)));
Vertex slidingStage1 = dag.newVertex("sliding-stage-1",
        Processors.accumulateByFrameP(
                Trade::getTicker,
                Trade::getTime, TimestampKind.EVENT,
                windowDef,
                counting()));
Vertex slidingStage2 = dag.newVertex("sliding-stage-2",
        Processors.combineToSlidingWindowP(windowDef, counting()));
Vertex formatOutput = dag.newVertex("format-output",
        formatOutput());
Vertex sink = dag.newVertex("sink",
        SinkProcessors.writeFileP(OUTPUT_DIR_NAME));

tickerSource.localParallelism(1);
generateTrades.localParallelism(1);

return dag
        .edge(between(tickerSource, generateTrades)
                .distributed().broadcast())
        .edge(between(generateTrades, insertWatermarks)
                .isolated())
        .edge(between(insertWatermarks, slidingStage1)
                .partitioned(Trade::getTicker, HASH_CODE))
        .edge(between(slidingStage1, slidingStage2)
                .partitioned(Entry<String, Long>::getKey, HASH_CODE)
                .distributed())
        .edge(between(slidingStage2, formatOutput)
                .isolated())
        .edge(between(formatOutput, sink)
                .isolated());
```

The source vertex reads a Hazelcast IMap, just like it did in the word counting example. Trade generating vertex uses a custom processor that generates mock trades. It can be reviewed here. The implementation of `complete()` is non-trivial, but most of the complexity just deals with precision timing of events. For simplicity's sake the processor must be configured with a local parallelism of 1;

generating a precise amount of mock traffic from parallel processors would take more code and coordination.

The major novelty is the watermark-inserting vertex. It must be added in front of the windowing vertex and will insert watermark items according to the configured policy. In this case we use the simplest one, `withFixedLag`, which will make the watermark lag behind the top observed event timestamp by a fixed amount. Emission of watermarks is additionally throttled, so that only one watermark item per frame is emitted. The windowing processors emit data only when the watermark reaches the next frame, so inserting it more often than that would be just overhead.

The edge from `insertWatermarks` to `slidingStage1` is partitioned; you may wonder how that works with watermark items, since

1. their type is different from the "main" stream item type and they don't have a partitioning key
2. each of them must reach all downstream processors.

It turns out that Jet must treat them as a special case: regardless of the configured edge type, watermarks are routed using the broadcast policy.

The stage-1 processor will just forward the watermark it receives, along with any aggregation results whose emission it triggers, to stage 2.

The full code of this sample is in StockExchange.java and running it you'll get an endless stream of data accumulating on the disk. To spare your filesystem we've limited the execution time to 10 seconds.

Chapter 8. Expert Zone - The Core API

This section covers the Core API, Jet's low-level API that directly exposes the computation engine's raw features. If you are looking for the API to build your computation pipeline, please refer to the Work With Jet section.

Creating a Core API DAG requires expert-level familiarity with concepts like partitioning schemes, vertex parallelism, distributed vs. local edges, etc. Furthermore, this API offers no static type safety and it is very easy to create a DAG that fails with a `ClassCastException` when executed. Even though it is possible, this API is not intended to create DAGs by hand; its purpose is to serve as the infrastructure on top of which to build high-level DSLs and APIs that describe computation jobs.

Implementing a Core API `Processor` requires even greater expertise than building a DAG. Among other things, you have to be acquainted in detail with Jet's concept of cooperative multithreading. While we provide as much convenience as we can for extending Jet with your custom processors, we cannot remove the dangers of using these facilities improperly.

8.1. Jet Execution Model

At the heart of Jet is the `TaskletExecutionService`. It manages the threads that perform all the computation in a Jet job. Although this class is not formally a part of Jet's public API, understanding how it schedules code for execution is essential if you want to implement a cooperative processor.

8.1.1. Cooperative Multithreading

Cooperative multithreading is one of the core features of Jet and can be roughly compared to green threads. It is purely a library-level feature and does not involve any low-level system or JVM tricks; the `Processor` API is simply designed in such a way that the processor can do a small amount of work each time it is invoked, then yield back to the Jet engine. The engine manages a thread pool of fixed size and on each thread, the processors take their turn in a round-robin fashion.

The point of cooperative multithreading is better performance. Several factors contribute to this:

- The overhead of context switching between processors is much lower since the operating system's thread scheduler is not involved.
- The worker thread driving the processors stays on the same core for longer periods, preserving the CPU cache lines.
- The worker thread has direct knowledge of the ability of a processor to make progress (by inspecting its input/output buffers).

8.1.2. Tasklet

The execution service doesn't deal with processors directly; instead it deals with *tasklets*. `Tasklet` is a very simple functional interface derived from the standard Java `Callable<ProgressState>`. The

execution service manages a pool of worker threads, each being responsible for a list of tasklets. The worker thread simply invokes the `call()` methods on its tasklets in a round-robin fashion. The method's return value tells whether the tasklet made progress and whether it is now done.

The most important tasklet is the one driving a processor (`ProcessorTasklet`); there are a few others that deal with network sending/receiving and taking snapshots.

8.1.3. Work Stealing

When a tasklet is done, its worker will inspect all the other workers' tasklet lists to see if any of them has a longer tasklet list than its own. If it finds such a worker, it will "steal" one of its tasklets to even out the load per thread.

8.1.4. Exponential Backoff

If none of the worker's tasklets report having made progress, the worker will go to a short sleep. If this happens again after it wakes up, it will sleep for twice as long. Once it reaches 1 ms sleep time, it will continue retrying once per millisecond to see if any tasklets can make progress.

8.1.5. ProcessorTasklet

`ProcessorTasklet` is the one that drives a processor. It manages its inbox, outbox, inbound/outbound concurrent queues, and tracks the current processor state so it knows which of its callback methods to call.

During each `tasklet.call()`, `ProcessorTasklet` makes one call into one of its processor's callbacks. It determines the processor's progress status and reports it to the execution service.

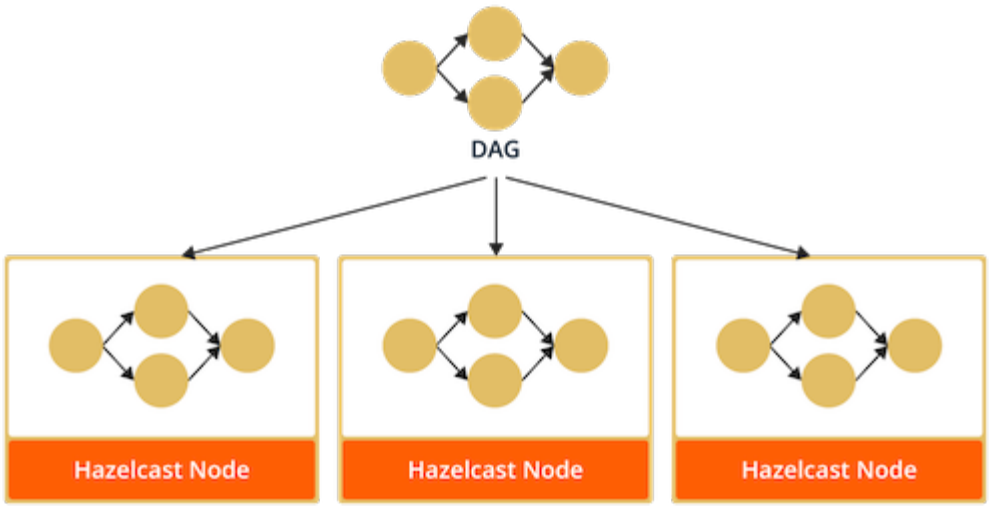
8.1.6. Non-Cooperative Processor

If a processor declares itself as non-cooperative, the execution service will start a dedicated Java thread for its tasklet to run on.

Even if it's non-cooperative, the processor's callback methods must still make sure they don't run for longer than a second or so at a time. Otherwise the tasklet will never be able to initiate a snapshot on the processor.

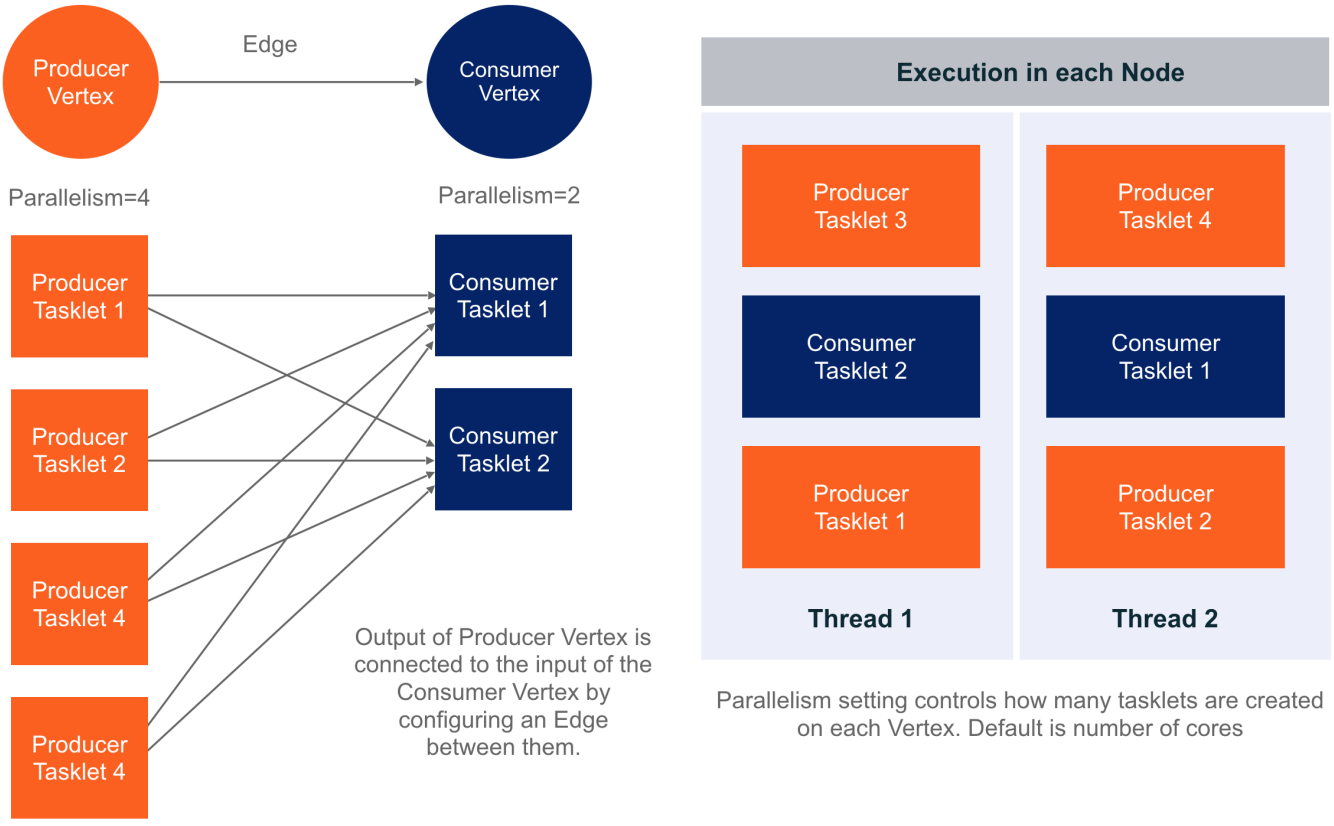
8.1.7. Running a Jet job

When you submit a `Job` to it, Jet replicates the DAG to the whole Jet cluster and executes a copy of it on each member.



Jet executes the job on a user-configurable number of threads which use work stealing to balance the amount of work being done on each thread. Each worker thread has a list of tasklets it is in charge of and as tasklets complete at different rates, the remaining ones are moved between workers to keep the load balanced.

Each instance of a `Processor` is wrapped in one tasklet which the execution service repeatedly executes until it is done. A vertex with a parallelism of 8 running on 4 members would have a total of 32 tasklets running at the same time. Each member has the same number of tasklets running.



When you make a request to execute a Job, the corresponding DAG and additional resources are deployed to the Jet cluster. Jet builds an execution plan for the DAG on each member, which creates the

associated tasklets for each Vertex and connects them to their inputs and outputs.

Jet uses Single Producer/Single Consumer ringbuffers to transfer the data between processors on the same member. They are data-type agnostic, so any data type can be used to transfer the data between vertices.

Ringbuffers, being bounded queues, introduce natural backpressure into the system; if a consumer's ringbuffer is full, the producer will have to back off until it can enqueue the next item. When data is sent to another member over the network, there is no natural backpressure, so Jet uses explicit signaling in the form of adaptive receive windows.

8.2. DAG

The DAG-building API is centered around the `DAG` class. This is a pure data class and can be instantiated on its own, without a Jet instance. This makes it simple to separate the job-describing code from the code that manages the lifecycle of Jet instances. To start building a DAG, you just write

```
DAG dag = new DAG();
```

A good practice is to structure the DAG-building code into the following sections:

1. Create all the vertices.
2. Configure the local parallelism of vertices.
3. Create the edges.

Example:

```
DAG dag = new DAG();

// 1. Create vertices
Vertex source = dag.newVertex("source", Sources.readFiles("."));
Vertex transform = dag.newVertex("transform", Processors.map(
        (String line) -> entry(line, line.length())));
Vertex sink = dag.newVertex("sink", Sinks.writeMap("sinkMap"));

// 2. Configure local parallelism
source.localParallelism(1);

// 3. Create edges
dag.edge(between(source, transform));
dag.edge(between(transform, sink));
```

8.2.1. Creating a Vertex

The two mandatory elements of creating a vertex are its string identifier and the supplier of processors. The latter can be provided in three variants, differing in the degree of explicit control over the lifecycle management of the processors. From simple to complex they are:

1. `DistributedSupplier<Processor>` directly returns processor instances from its `get()` method. It is expected to be stateless and return equivalent instances on each call. It doesn't provide any initialization or cleanup code.
2. `ProcessorSupplier` returns in a single call all the processors that will run on a single cluster member. It may specialize each instance, for example to achieve local data partitioning. It is also in charge of the member-local lifecycle (initialization and destruction).
3. **`ProcessorSupplier`** returns in a single call an object that will be in charge of creating all the processors for a vertex. Given a list of member addresses, the object it returns is a `Function<Address, ProcessorSupplier>` which will then be called with each of the addresses from the list to retrieve the `ProcessorSupplier` specialized for the given member.

Usually you don't have to care, or even know, which of these variants is used. You'll call a library-provided factory method that returns one or the other and they will integrate the same way into your `newVertex()` calls.

8.2.2. Local and Global Parallelism of Vertex

The vertex is implemented by one or more instances of `Processor` on each member. Each vertex can specify how many of its processors will run per cluster member using the `localParallelism` property; every member will have the same number of processors. A new `Vertex` instance has this property set to `-1`, which requests to use the default value equal to the configured size of the cooperative thread pool. The latter defaults to `Runtime.availableProcessors()` and is configurable via `InstanceConfig.setCooperativeThreadCount()`.

In most cases the only value of local parallelism that you'll want to explicitly configure is `1` for the cases where no parallelism is desirable (e.g. on a source processor reading from a file).

The **global parallelism** of the vertex is also an important value, especially in terms of the distribution of partitions among processors. It is equal to local parallelism multiplied by the cluster size.

8.2.3. Edge Ordinal

An edge is connected to a vertex with a given **ordinal**, which identifies it to the vertex and its processors. When a processor receives an item, it knows the ordinal of the edge on which the item came in. Things are similar on the outbound side: the processor emits an item to a given ordinal, but also has the option to emit the same item to all ordinals. This is the most typical case and allows easy replication of a data stream across several edges.

When you use the `between()` edge factory, the edge will be connected at ordinal 0 at both ends. When

you need a different ordinal, use the `from(a, ord1).to(b, ord2)` form. There must be no gaps in ordinal assignment, which means a vertex will have inbound edges with ordinals 0..N and outbound edges with ordinals 0..M.

This example shows the usage of `between()` and `from().to()` forms to build a DAG with one source feeding two computational vertices:

```
DAG dag = new DAG();

Vertex source = dag.newVertex("source", Sources.readFiles("."));
Vertex v1 = dag.newVertex("v1", ...);
Vertex v2 = dag.newVertex("v2", ...);

dag.edge(between(source, v1));
dag.edge(from(source, 1).to(v2));
```

8.2.4. Local and Distributed Edge

A major choice to make in terms of data routing is whether the candidate set of target processors is unconstrained, encompassing all processors across the cluster, or constrained to just those running on the same cluster member. This is controlled by the `distributed` property of the edge. By default the edge is local and calling the `distributed()` method removes this restriction.

With appropriate DAG design, network traffic can be minimized by employing local edges. They are implemented with the most efficient kind of concurrent queue: single-producer, single-consumer array-backed queue. It employs wait-free algorithms on both sides and avoids even the latency of `volatile` writes by using `lazySet`.

The quintessential example of employing local-distributed edge combo is the two-stage aggregation. Here's a review of that setup from the Word Count tutorial:

```
dag.edge(between(source, tokenizer))
   .edge(between(tokenizer, accumulate)
           .partitioned(DistributedFunctions.wholeItem(), Partitioner.HASH_CODE))
   .edge(between(accumulate, combine)
           .distributed()
           .partitioned(DistributedFunctions.entryKey()))
   .edge(between(combine, sink));
```

Note that only the edge from `accumulate` to `combine` is distributed.

8.2.5. Routing Policies

The routing policy decides which of the processors in the candidate set to route each particular item to.

Unicast

This is the default routing policy, the one you get when you write

```
dag.edge(between(source, tokenizer))
```

For each item it chooses a single destination processor with no further restrictions on the choice. The only guarantee given by this pattern is that exactly one processor will receive the item, but typically care will be taken to "spray" the items equally over all the reception candidates.

This choice makes sense when the data does not have to be partitioned, usually implying a downstream vertex which can compute the result based on each item in isolation.

Isolated

This is a more restricted kind of unicast policy: any given downstream processor receives data from exactly one upstream processor. This is needed in some DAG setups to apply selective backpressure to individual upstream source processors. Activate this policy by calling `isolated()` on the edge:

```
dag.edge(between(source, insertWatermarks).isolated());
```

Broadcast

A broadcasting edge sends each item to all candidate receivers. This is useful when some small amount of data must be broadcast to all downstream vertices. Usually such vertices will have other inbound edges in addition to the broadcasting one, and will use the broadcast data as context while processing the other edges. In such cases the broadcasting edge will have a raised priority. There are other useful combinations, like a parallelism-one vertex that produces the same result on each member.

Activate this policy by calling `broadcast()` on the edge:

```
dag.edge(between(source, count).broadcast());
```

Partitioned

A partitioned edge sends each item to the one processor responsible for the item's partition ID. On a distributed edge, this processor will be unique across the whole cluster. On a local edge, each member will have its own processor for each partition ID.

Multiple partitions can be assigned to each processor. The global number of partitions is controlled by the number of partitions in the underlying Hazelcast IMDG configuration. Please refer to the Hazelcast Reference Manual for more information about Hazelcast IMDG partitioning.

This is the default algorithm to determine the partition ID of an item:

1. Apply the key extractor function defined on the edge to retrieve the partitioning key.
2. Serialize the partitioning key to a byte array using Hazelcast serialization.
3. Apply Hazelcast's standard `MurmurHash3`-based algorithm to get the key's hash value.
4. Partition ID is the hash value modulo the number of partitions.

The above procedure is quite CPU-intensive, but has the crucial property of giving repeatable results across all cluster members, which may be running on disparate JVM implementations.

Another common choice is to use Java's standard `Object.hashCode()`. It is often significantly faster. However, it is not a safe strategy in general because `hashCode()'s contract does not require repeatable results across JVMs, or even different instances of the same JVM version. If a given class's Javadoc explicitly specifies the hashing function used, then its instances are safe to partition with `hashCode()`.

You can provide your own implementation of `Partitioner` to gain full control over the partitioning strategy.

We use both partitioning strategies in the Word Count example:

```
dag.edge(between(tokenizer, accumulate)
           .partitioned(wholeItem(), Partitioner.HASH_CODE))
   .edge(between(accumulate, combine)
           .distributed()
           .partitioned(entryKey()))
```

The local-partitioned edge uses partitioning by hash code and the distributed edge uses the default Hazelcast partitioning, to ensure correctness. Note that a detailed inspection of the data types that travel on the distributed edge reveals for that particular case that the hashcode-based partitioning would work on the distributed edge as well. We use Hazelcast partitioning nevertheless, for demonstration purposes. Since much less data travels towards the combiner than towards the accumulator, the performance of the whole job is hardly affected by this choice.

All-To-One

The all-to-one routing policy is a special case of the `partitioned` policy which assigns the same partition ID to all items. The partition ID is randomly chosen at job initialization time. This policy makes sense on a distributed edge when all the items from all the members must be routed to the same member and the same processor instance running on it. Local parallelism of the target vertex should be set to 1, otherwise there will be idle processors that never get any items.

On a local edge this policy doesn't make sense since simply setting the local parallelism of the target vertex to 1 constrains the local choice to just one processor instance.

In the `TopNStocks` example the stream-processing job must find the stocks with fastest-changing prices. To achieve this a single processor must see the complete picture, so an all-to-one edge is employed:

```
dag.edge(between(topNStage1, topNStage2).distributed().allToOne())
```

8.2.6. Priority

By default the processor receives items from all inbound edges as they arrive. However, there are important cases where an edge must be consumed in full to make the processor ready to accept data from other edges. A major example is a "hash join" which enriches the data stream with data from a lookup table. This can be modeled as a join of two data streams where the *enriching* stream contains the data for the lookup table and must be consumed in full before consuming the stream to be enriched.

The `priority` property controls the order of consuming the edges. Edges are sorted by their priority number (ascending) and consumed in that order. Edges with the same priority are consumed without particular ordering (as the data arrives).

We can see a prioritized edge in action in the TF-IDF example:

```
dag.edge(between(stopwordSource, tokenize).broadcast().priority(-1))
```

The `tokenize` vertex performs lookup table-based filtering of words. It must receive the entire lookup table before beginning to process the data.

A Fault Tolerance Caveat

As explained in the section on the Processor API, Jet takes regular snapshots of processor state when fault tolerance is enabled. A processor will get a special item in its input stream, called a *barrier*. When working in the *exactly once* mode, as soon as it receives it, it must stop pulling the data from that stream, wait for the same barrier in all other streams, and then emit its state to the snapshot storage. This is in direct contradiction with the contract of edge prioritization: the processor is not allowed to consume any other streams before having fully exhausted the prioritized ones.

This is why Jet does not initiate a snapshot until all the high-priority edges have been fully consumed.

Although strictly speaking this only applies to the *exactly once* mode, Jet postpones taking the snapshot in *at least once* mode as well. Even though the snapshot could begin early, it would still not be able to complete until the prioritized edges have been consumed. The result would be just that there are many more items processed twice after the restart.

8.2.7. Fine-Tuning Edges

Edges can be configured with an `EdgeConfig` instance, which specifies additional fine-tuning parameters. For example,

```
dag.edge(between(tickerSource, generateTrades)
        .setConfig(new EdgeConfig().setQueueSize(512)));
```

Please refer to the Javadoc of `EdgeConfig` for details.

8.3. Job

A Job is the unit of work which is executed. A Job is described by a DAG, which describes the computation to be performed, and the inputs and outputs of the computation.

`Job` is a handle to the execution of a `DAG`. To create a job, supply the `DAG` to a previously created `JetInstance` as shown below:

```
JetInstance jet = Jet.newJetInstance(); // or Jet.newJetClient();
DAG dag = new DAG();
dag.newVertex(..);
jet.newJob(dag).execute().get();
```

As hinted in the code example, the job submission API is identical whether you use it from a client machine or directly on an instance of a Jet cluster member. This works because the `Job` instance is serializable and the client can send it over the network when submitting the job. The same `Job` instance can be submitted for execution many times.

Job execution is asynchronous. The `execute()` call returns as soon as the Jet cluster has been contacted and the serialized job is sent to it. The user gets a `Future` which can be inspected or waited on to find out the outcome of a computation job. It is also cancelable and can send a cancelation command to the Jet cluster.

Note that the `Future` only signals the status of the job, it does not contain the result of the computation. The DAG explicitly models the storing of results via its **sink** vertices. Typically the results will be in a Hazelcast map or another structure and have to be accessed by their own API after the job is done.

8.3.1. Deploy your Resources

If the Jet cluster has not been started with all the job's computation code already on the classpath, you have to deploy the code together with the Job instance:

```
JobConfig config = new JobConfig();
config.addJar("..");
jet.newJob(dag, config).execute().get();
```

When reading and writing data to the underlying Hazelcast IMDG instance, keep in mind that the deployed code is available **only** within the scope of the executing Jet job.

8.4. Processor

`Processor` is the main type whose implementation is up to the user of the Core API: it contains the code of the computation to be performed by a vertex. There are a number of Processor building blocks in the Core API which allow you to just specify the computation logic, while the provided code handles the processor's cooperative behavior. Please refer to the AbstractProcessor section.

A processor's work can be conceptually described as follows: "receive data from zero or more input streams and emit data into zero or more output streams." Each stream maps to a single DAG edge (either inbound or outbound). There is no requirement on the correspondence between input and output items; a processor can emit any data it sees fit, including none at all. The same `Processor` abstraction is used for all kinds of vertices, including sources and sinks.

The implementation of a processor can be stateful and does not need to be thread-safe because Jet guarantees to use the processor instances from one thread at a time, although not necessarily always the same thread.

8.4.1. Cooperativeness

`Processor` instances are cooperative by default. The processor can opt out of cooperative multithreading by overriding `isCooperative()` to return `false`. Jet will then start a dedicated thread for it.

To maintain an overall good throughput, a cooperative processor must take care not to hog the thread for too long (a rule of thumb is up to a millisecond at a time). Jet's design strongly favors cooperative processors and most processors can and should be implemented to fit these requirements. The major exception are sources and sinks because they often have no choice but calling into blocking I/O APIs.

8.4.2. The Outbox

The processor sends its output items to its `Outbox` which has a separate bucket for each outbound edge. The buckets have limited capacity and will refuse an item when full. A cooperative processor should be implemented such that when the outbox refuses its item, it saves its processing state and returns from the processing method. The execution engine will then drain the outbox buckets.

8.4.3. Data Processing Callbacks

process(ordinal, inbox)

Jet passes the items received over a given edge to the processor by calling `process(ordinal, inbox)`. All items received since the last `process()` call are in the inbox, but also all the items the processor hasn't removed in a previous `process()` call. There is a separate instance of `Inbox` for each inbound edge, so any given `process()` call involves items from only one edge.

The processor must not remove an item from the inbox until it has fully processed it. This is important with respect to the cooperative behavior: the processor may not be allowed to emit all items

corresponding to a given input item and may need to return from the `process()` call early, saving its state. In such a case the item should stay in the inbox so Jet knows the processor has more work to do even if no new items are received.

tryProcessWatermark(watermark)

When new highest watermark is received from all input edges and all input processor instances, the `tryProcessWatermark(watermark)` method is called. The watermark value is always greater than in the previous call.

The implementation may choose to process only partially and return `false`, in which case it will be called again later with the same timestamp before any other processing method is called. When the method returns `true`, the watermark is forwarded to the downstream processors.

tryProcess()

If a processor's inbox is empty, Jet will call its `tryProcess()` method instead. This allows the processor to perform work that is not input data-driven. The method has a `boolean` return value and if it returns `false`, it will be called again before any other methods are called. This way it can retry emitting its output until the outbox accepts it.

An important use case for this method is the emission of watermark items. A job that processes an infinite data stream may experience occasional lulls - periods with no items arriving. On the other hand, a windowing processor is not allowed to act upon each item immediately due to event skew; it must wait for a watermark item to arrive. During a stream lull this becomes problematic because the watermark itself is primarily data-driven and advances in response to the observation of event timestamps. The watermark-inserting processor must be able to advance the watermark even during a stream lull, based on the passage of wall-clock time, and it can do it inside the `tryProcess()` method.

complete()

Jet calls `complete()` when all the input edges are exhausted. It is the last method to be invoked on the processor before disposing of it. Typically this is where a batch processor emits the results of an aggregating operation. If it can't emit everything in a given call, it should return `false` and will be called again later.

8.4.4. Snapshotting Callbacks

Hazelcast Jet supports fault-tolerant processing jobs by taking distributed snapshots. In regular time intervals each of the source vertices will perform a snapshot of its own state and then emit a special item to its output stream: a *barrier*. The downstream vertex that receives the barrier item makes its own snapshot and then forwards the barrier to its outbound edges, and so on towards the sinks.

At the level of the `Processor` API the barrier items are not visible; `ProcessorTasklet` handles them internally and invokes the snapshotting callback methods described below.

saveToSnapshot()

Jet will call `saveToSnapshot()` when it determines it's time for the processor to save its state to the current snapshot. Except for source vertices, this happens when the processor has received the barrier item from all its inbound streams and processed all the data items preceding it. The method must emit all its state to the special *snapshotting bucket* in the Outbox, by calling `outbox.offerToSnapshot()`. If the outbox doesn't accept all the data, it must return `false` to be called again later, after the outbox has been flushed.

When this method returns `true`, `ProcessorTasklet` will forward the barrier item to all the outbound edges.

restoreFromSnapshot()

When a Jet job is restarting after having been suspended, it will first reload all the state from the last successful snapshot. Each processor will get its data through the invocations of `restoreFromSnapshot()`. Its parameter is the `Inbox` filled with a batch of snapshot data. The method will be called repeatedly until it consumes all the snapshot data.

finishSnapshotRestore()

After it has delivered all the snapshot data to `restoreFromSnapshot()`, Jet will call `finishSnapshotRestore()`. The processor may use it to initialize some transient state from the restored state.

8.4.5. Best Practice: Document At-Least-Once Behavior

As we discuss in the Under the Hood chapter, the behavior of a processor under *at-least-once* semantics can deviate from correctness in extremely non-trivial and unexpected ways. Therefore the processor should always document its possible behaviors for that case.

8.5. AbstractProcessor

`AbstractProcessor` is a convenience class designed to deal with most of the boilerplate in implementing the full `Processor` API.

8.5.1. Receiving items

On the reception side the first line of convenience are the `tryProcessN()` methods. While in the inbox the watermark and data items are interleaved, these methods take care of the boilerplate needed to filter out the watermarks. Additionally, they get one item at a time, eliminating the need to write a suspendable loop over the input items.

There is a separate method specialized for each edge from 0 to 4 (`tryProcess0`..`tryProcess4`) and a catch-all method `tryProcess(ordinal, item)`. If the processor doesn't need to distinguish between the inbound edges, the latter method is a good match; otherwise, it is simpler to implement one or more of

the ordinal-specific methods. The catch-all method is also the only way to access inbound edges beyond ordinal 4, but such cases are very rare in practice.

Paralleling the above there are `tryProcessWm(ordinal, wm)` and `tryProcessWmN(wm)` methods that get just the watermark items.

8.5.2. Emitting items

`AbstractProcessor` has a private reference to its outbox and lets you access all its functionality indirectly. The `tryEmit()` methods offer your items to the outbox. If you get a `false` return value, you must stop emitting items and return from the current callback method of the processor. For example, if you called `tryEmit()` from `tryProcess0()`, you should return `false` so Jet will call `tryProcess0()` again later, when there's more room in the outbox. Similar to these methods there are `tryEmitToSnapshot()` and `emitFromTraverserToSnapshot()`, to be used from the `saveToSnapshot()` callback.

Implementing a processor to respect the above rule is quite tricky and error-prone. Things get especially tricky when there are several items to emit, such as:

- when a single input item maps to many output items
- when the processor performs a group-by-key operation and emits each group as a separate item

You can avoid most of the complexity if you wrap all your output in a `Traverser`. Then you can simply say `return emitFromTraverser(myTraverser)`. It will:

1. try to emit as many items as possible
2. return `false` if the outbox refuses an item
3. hold on to the refused item and continue from it when it's called again with the same traverser.

There is one more layer of convenience relying on `emitFromTraverser()`: the nested class `FlatMapper`, which makes it easy to implement a flatmapping kind of transformation. It automatically handles the concern of creating a new traverser when the previous one is exhausted and reusing the previous one until exhausted.

8.5.3. Traverser

`Traverser` is a very simple functional interface whose shape matches that of a `Supplier`, but with a contract specialized for the traversal over a sequence of non-null items: each call to its `next()` method returns another item of the sequence until exhausted, then keeps returning `null`. A traverser may also represent an infinite, non-blocking stream of items: it may return `null` when no item is currently available, then later return more items.

The point of this type is the ability to implement traversal over any kind of dataset or lazy sequence with minimum hassle: often just by providing a one-liner lambda expression. This makes it very easy to integrate with Jet's convenience APIs for cooperative processors.

`Traverser` also supports some `default` methods that facilitate building a simple transformation layer

over the underlying sequence: `map`, `filter`, `flatMap`, etc.

The following example shows how you can implement a simple flatmapping processor:

```
public class ItemAndSuccessorP extends AbstractProcessor {
    private final FlatMapper<Integer, Integer> flatMapper =
        flatMapper(i -> traverseIterable(asList(i, i + 1)));

    @Override
    protected boolean tryProcess(int ordinal, Object item) {
        return flatMapper.tryProcess((int) item);
    }
}
```

For each received `Integer` this processor emits the number and its successor. If the outbox refuses an item, `flatMapper.tryProcess()` returns `false` and stays ready to resume the next time it is invoked. The fact that it returned `false` signals Jet to invoke `ItemAndSuccessorP.tryProcess()` again with the same arguments.

8.6. WatermarkPolicy

As mentioned in the Work_with_Jet chapter, determining the watermark is somewhat of a black art; it's about superimposing order over a disordered stream of events. We must decide at which point it stops making sense to wait even longer for data about past events to arrive. There's a tension between two opposing forces here:

- wait as long as possible to account for all the data;
- get results as soon as possible.

While there are ways to (kind of) achieve both, there's a significant associated cost in terms of complexity and overall performance. Hazelcast Jet takes a simple approach and strictly triages stream items into "still on time" and "late", discarding the latter.

`WatermarkPolicy` is the abstraction that computes the value of the watermark for a (sub)stream of disordered data items. It takes as input the timestamp of each observed item and outputs the current watermark value.

8.6.1. Predefined watermark policies

We provide some general, data-agnostic watermark policies in the `WatermarkPolicies` class. They vary in how well they deal with advancing the watermark during a stream lull. The better they deal with it, the more assumptions they must make on the nature of the events' timestamp values and on the relationship between the timestamps and the locally observed wall-clock time.

"With Fixed Lag"

The `withFixedLag()` policy will maintain a watermark that lags behind the highest observed event timestamp by a configured amount. In other words, each time an event with the highest timestamp so far is encountered, this policy advances the watermark to `eventTimestamp - lag`. This puts a limit on the spread between timestamps in the stream: all events whose timestamp is more than the configured `lag` behind the highest timestamp are considered late.

"Limiting Lag and Delay"

The `limitingLagAndDelay()` policy applies the same fixed-lag logic as above and adds another limit: maximum delay from observing an item and advancing the watermark to at least that item's timestamp. A stream may experience a lull (no items arriving) and this added limit will ensure that the watermark doesn't stay behind the highest timestamp observed before the onset of the lull. However, the skew between substreams may still cause the watermark that reaches the downstream vertex to stay behind some timestamps. This is because the downstream will only get the lowest of all substream watermarks.

The advantage of this policy is that it doesn't assume anything about the unit of measurement used for event timestamps.

"Limiting Lag and Lull"

The `limitingLagAndLull()` policy is similar to `limitingLagAndDelay` in addressing the stream lull problem and goes a step further by addressing the issues of lull combined with skew. To achieve this it must introduce an assumption, though: that the time unit used for event timestamps is milliseconds. After a given period passes with the watermark not being advanced by the arriving data (i.e., a lull happens), it will start advancing it in lockstep with the passage of the local system time. The watermark isn't adjusted *towards* the local time; the policy just ensures the difference between local time and the watermark stays the same during a lull. Since the system time advances equally on all substream processors, the watermark propagated to downstream is now guaranteed to advance regardless of the lull.

There is, however, a subtle issue with `limitingLagAndLull()`: if there is any substream that never observes an item, that substream's policy instance won't be able to initialize its "last seen timestamp" and will cause the watermark sent to the downstream to forever lag behind all the actual data.

"Limiting Timestamp and Wall-Clock Lag"

The `limitingTimestampAndWallClockLag()` policy makes a stronger assumption: that the event timestamps are in milliseconds since the Unix epoch and that they are synchronized with the local time on the processing machine. It puts a limit on how much the watermark can lag behind the local time. As long as its assumption holds, this policy gives straightforward results. It also doesn't suffer from the subtle issue with `limitingLagAndLull()`.

8.6.2. Watermark Throttling

The policy objects presented above will return the "ideal" watermark value according to their logic; however it would be too much overhead to insert a watermark item each time the ideal watermark advances (typically a thousand times per second). `WatermarkEmissionPolicy` is the object that decides whether to emit a watermark item given the last emitted and the current value of the watermark. For the purpose of sliding windows there is an easy answer: suppress all watermark items that belong to the same frame as the already emitted one. Such items would have no effect since the watermark must advance beyond a frame's end for the aggregating vertex to consider the frame completed and act upon its results. The method `emitByFrame()` will return a policy with this kind of throttling applied. For other cases there is `emitByMinStep()` which suppresses watermark items until the watermark has advanced at least `minStep` ahead of the previously emitted one.

8.6.3. Maximum watermark retention on substream merge

When two input streams are merged into one for downstream processing, Jet waits for the watermark to advance in all substreams in order to advance the overall watermark. The process that does this is called *watermark coalescing* and it results in increased latency of the output with respect to the input and possibly also increased memory usage due to the retention of all the pending data.

The skew between two distributed streams is defined as the difference in their watermark values. There is always some skew in the system and it's acceptable, but it can grow very large due to various causes such as a hiccup on one of the cluster members (a long GC pause, for example), external source hiccup, non-balanced partitions and so on.

An option to limit the watermark retention is available using `JobConfig.setMaxWatermarkRetainMillis() `. The option sets the maximum time to retain the watermarks while coalescing them. A negative value disables the limit and Jet will retain the watermark as long as needed. With this setting you choose a trade-off between latency and correctness that arises when dealing with stream skew.

8.7. Vertices in the Library

While formally there's only one kind of vertex in Jet, in practice there is an important distinction between the following:

- A **source** is a vertex with no inbound edges. It injects data from the environment into the Jet job.
- A **sink** is a vertex with no outbound edges. It drains the output of the Jet job into the environment.
- A **computational** vertex has both kinds of edges. It accepts some data from upstream vertices, performs some computation, and emits the results to downstream vertices. Typically it doesn't interact with the environment.

The `com.hazelcast.jet.processor` package contains static utility classes with factory methods that return suppliers of processors, as required by the `dag.newVertex(name, procSupplier)` calls. There is a convention in Jet that every module containing vertex implementations contributes a utility class to

the same package. Inspecting the contents of this package in your IDE should allow you to discover all vertex implementations available on the project's classpath. For example, there are modules that connect to 3rd party resources like Kafka and Hadoop Distributed File System (HDFS). Each such module declares a class in the same package, `com.hazelcast.jet.processor`, exposing the module's source and sink definitions.

The main factory class for the source vertices provided by the Jet core module is `SourceProcessors`. It contains sources that ingest data from Hazelcast IMDG structures like `IMap`, `ICache`, `IList`, etc., as well as some simple sources that get data from files and TCP sockets (`readFiles`, `streamSocket` and some more).

Paralleling the sources there's `SinkProcessors` for the sink vertices, supporting the same range of resources (IMDG, files, sockets). There's also a general `writeBuffered` method that takes some boilerplate out of writing custom sinks. The user must implement a few primitives: create a new buffer, add an item to it, flush the buffer. The provided code takes care of integrating these primitives into the `Processor` API (draining the inbox into the buffer and handling the general lifecycle).

Finally, the computational vertices are where the main action takes place. The main class with factories for built-in computational vertices is `Processors`.

8.8. Implement a Custom Source or Sink

The Hazelcast Jet distribution contains vertices for the sources and sinks exposed through the Pipeline API. You can extend Jet's support for sources and sinks by writing your own vertex implementations.

One of the main concerns when implementing a source vertex is that the data source is typically distributed across multiple machines and partitions, and the work needs to be distributed across multiple members and processors.

8.8.1. How Jet Creates and Initializes a Job

These are the steps taken to create and initialize a Jet job:

1. The user builds the DAG and submits it to the local Jet client instance.
2. The client instance serializes the DAG and sends it to a member of the Jet cluster. This member becomes the **coordinator** for this Jet job.
3. The coordinator deserializes the DAG and builds an execution plan for each member.
4. The coordinator serializes the execution plans and distributes each to its target member.
5. Each member acts upon its execution plan by creating all the needed tasklets, concurrent queues, network senders/receivers, etc.
6. The coordinator sends the signal to all members to start job execution.

The most visible consequence of the above process is the `ProcessorMetaSupplier` type: one must be provided for each `Vertex`. In Step 3, the coordinator deserializes the meta-supplier as a constituent of the `DAG` and asks it to create `ProcessorSupplier` instances which go into the execution plans. A separate

instance of `ProcessorSupplier` is created specifically for each member's plan. In Step 4, the coordinator serializes these and sends each to its member. In Step 5 each member deserializes its `ProcessorSupplier` and asks it to create as many `Processor` instances as configured by the vertex's `localParallelism` property.

This process is so involved because each `Processor` instance may need to be differently configured. This is especially relevant for processors driving a source vertex: typically each one will emit only a slice of the total data stream, as appropriate to the partitions it is in charge of.

ProcessorMetaSupplier

The client serializes an instance of `ProcessorMetaSupplier` as part of each `Vertex` in the `DAG`. The **coordinator** member deserializes the instance and uses it to create `ProcessorSupplier`s by calling the `ProcessorMetaSupplier.get()` method. Before that, coordinator calls the `init()` method with a context object that you can use to get useful information. The `get()` method takes `List<Address>` as a parameter, which you should use to determine cluster members that will run the job, if needed.

ProcessorSupplier

Usually this type will be custom-implemented in the same cases where the meta-supplier is custom-implemented and will complete the logic of a distributed data source's partition assignment. It creates instances of `Processor` ready to start executing the vertex's logic.

Another use is to open and close external resources, such as files or connections. We provide `CloseableProcessorSupplier` for this.

8.8.2. Example - Distributed Integer Generator

Let's say we want to write a simple source that will generate numbers from 0 to 1,000,000 (exclusive). It is trivial to write a single `Processor` which can do this using `java.util.stream` and [`Traverser`](AbstractProcessor#page_Traverser).

```
class GenerateNumbersP extends AbstractProcessor {

    private final Traverser<Integer> traverser;

    GenerateNumbersP(int upperBound) {
        traverser = Traversers.traverseStream(IntStream.range(0, upperBound).boxed());
    }

    @Override
    public boolean complete() {
        return emitFromTraverser(traverser);
    }
}
```

Now we can build our DAG and execute it:

```
JetInstance jet = Jet.newJetInstance();

int upperBound = 10;
DAG dag = new DAG();
Vertex generateNumbers = dag.newVertex("generate-numbers",
        () -> new GenerateNumbersP(upperBound));
Vertex logInput = dag.newVertex("log-input",
        DiagnosticProcessors.writeLogger(i -> "Received number: " + i));
dag.edge(Edge.between(generateNumbers, logInput));

try {
    jet.newJob(dag).execute().get();
} finally {
    Jet.shutdownAll();
}
```

When you run this code, you will see the output as below:

```
Received number: 4
Received number: 0
Received number: 3
Received number: 2
Received number: 2
Received number: 2
```

Since we are using the default parallelism setting on this vertex, several instances of the source processor were created, all of which generated the same sequence of values. Generally we want the ability to parallelize the source vertex, so we have to make each processor emit only a slice of the total data set.

So far we've used the simplest approach to creating processors: a `DistributeSupplier<Processor>` function that keeps returning equal instances of processors. Now we'll step up to Jet's custom interface that gives us the ability to provide a list of separately configured processors: `ProcessorSupplier` and its method `get(int processorCount)`.

First we must decide on a partitioning policy: what subset will each processor emit. In our simple example we can use a simple policy: we'll label each processor with its index in the list and have it emit only those numbers `n` that satisfy `n % processorCount == processorIndex`. Let's write a new constructor for our processor which implements this partitioning logic:

```
GenerateNumbersP(int upperBound, int processorCount, int processorIndex) {
    traverser = Traversers.traverseStream(
            IntStream.range(0, upperBound)
                     .filter(n -> n % processorCount == processorIndex)
                     .boxed());
}
```

Given this preparation, implementing `ProcessorSupplier` is trivial:

```
class GenerateNumbersPSupplier implements ProcessorSupplier {

    private final int upperBound;

    GenerateNumbersPSupplier(int upperBound) {
        this.upperBound = upperBound;
    }

    @Override @Nonnull
    public List<? extends Processor> get(int processorCount) {
        return IntStream.range(0, processorCount)
                .mapToObj(index -> new GenerateNumbersP(upperBound, processorCount,
index))
                .collect(Collectors.toList());
    }
}
```

Let's use the custom processor supplier in our DAG-building code:

```
DAG dag = new DAG();
Vertex generateNumbers = dag.newVertex("generate-numbers",
        new GenerateNumbersPSupplier(10));
Vertex logInput = dag.newVertex("log-input",
        DiagnosticProcessors.writeLogger(i -> "Received number: " + i));
dag.edge(Edge.between(generateNumbers, logInput));
```

Now we can re-run our example and see that each number indeed occurs only once. However, note that we are still working with a single-member Jet cluster; let's see what happens when we add another member:

```
JetInstance jet = Jet.newJetInstance();
Jet.newJetInstance();

DAG dag = new DAG();
...
```

Running after this change we'll see that both members are generating the same set of numbers. This is because `ProcessorSupplier` is instantiated independently for each member and asked for the same number of processors, resulting in identical processors on all members. We have to solve the same problem as we just did, but at the higher level of cluster-wide parallelism. For that we'll need the `ProcessorMetaSupplier`: an interface which acts as a factory of `ProcessorSupplier`s, one for each cluster member. Under the hood it is actually always the meta-supplier that's created by the DAG-building code; the above examples are just implicit about it for the sake of convenience. They result in a simple meta-supplier that reuses the provided suppliers everywhere.

The meta-supplier is a bit trickier to implement because its method takes a list of Jet member addresses instead of a simple count, and the return value is a function from address to `ProcessorSupplier`. In our case we'll treat the address as just an opaque ID and we'll build a map from address to a properly configured `ProcessorSupplier`. Then we can simply return `map::get` as our function.

```
class GenerateNumbersPMetaSupplier implements ProcessorMetaSupplier {

    private final int upperBound;

    private transient int totalParallelism;
    private transient int localParallelism;

    GenerateNumbersPMetaSupplier(int upperBound) {
        this.upperBound = upperBound;
    }

    @Override
    public void init(@Nonnull Context context) {
        totalParallelism = context.totalParallelism();
        localParallelism = context.localParallelism();
    }

    @Override @Nonnull
    public Function<Address, ProcessorSupplier> get(@Nonnull List<Address> addresses) {
        Map<Address, ProcessorSupplier> map = new HashMap<>();
        for (int i = 0; i < addresses.size(); i++) {
            // We'll calculate the global index of each processor in the cluster:
            int globalIndexBase = localParallelism * i;
            // Capture the value of the transient field for the lambdas below:
            int divisor = totalParallelism;
            // processorCount will be equal to localParallelism:
            ProcessorSupplier supplier = processorCount ->
                    range(globalIndexBase, globalIndexBase + processorCount)
                            .mapToObj(globalIndex ->
                                new GenerateNumbersP(upperBound, divisor, globalIndex)
                            ).collect(toList());
            map.put(addresses.get(i), supplier);
        }
        return map::get;
    }

}
```

We change our DAG-building code to use the meta-supplier:

```
DAG dag = new DAG();
Vertex generateNumbers = dag.newVertex("generate-numbers",
        new GenerateNumbersPMetaSupplier(upperBound));
Vertex logInput = dag.newVertex("log-input",
        DiagnosticProcessors.writeLogger(i -> "Received number: " + i));
dag.edge(Edge.between(generateNumbers, logInput));
```

After re-running with two Jet members, we should once again see each number generated just once.

8.8.3. Sinks

Like a source, a sink is just another kind of processor. It accepts items from the inbox and pushes them into some system external to the Jet job (Hazelcast IMap, files, databases, distributed queues, etc.). A simple way to implement it is to extend `AbstractProcessor` and override `tryProcess`, which deals with items one at a time. However, sink processors must often explicitly deal with batching. In this case directly implementing `Processor` is better because its `process()` method gets the entire `Inbox` which can be drained to a buffer and flushed out.

8.8.4. Example - File Writer

In this example we'll implement a vertex that writes the received items to files. To avoid contention and conflicts, each processor must write to its own file. Since we'll be using a `BufferedWriter` which takes care of the buffering/batching concern, we can use the simpler approach of extending `AbstractProcessor`:

```
class WriteFileP extends AbstractProcessor implements Closeable {

    private final String path;

    private transient BufferedWriter writer;

    WriteFileP(String path) {
        setCooperative(false);
        this.path = path;
    }

    @Override
    protected void init(@Nonnull Context context) throws Exception {
        Path path = Paths.get(this.path, context.jetInstance().getName()
                + '-' + context.globalProcessorIndex());
        writer = Files.newBufferedWriter(path, StandardCharsets.UTF_8);
    }

    @Override
    protected boolean tryProcess(int ordinal, Object item) throws Exception {
        writer.append(item.toString());
        writer.newLine();
        return true;
    }

    @Override
    public void close() throws IOException {
        if (writer != null) {
            writer.close();
        }
    }
}
```

Some comments:

- The constructor declares the processor non-cooperative because it will perform blocking IO operations.
- `init()` method finds a unique filename for each processor by relying on the information reachable from the `Context` object.
- Note the careful implementation of `close()`: it first checks if writer is null, which can happen if `newBufferedWriter()` fails in `init()`. This would make `init()` fail as well, which would make the whole job fail and then our `ProcessorSupplier` would call `close()` to clean up.

Cleaning up on completion/failure is actually the only concern that we need `ProcessorSupplier` for: the other typical concern, specializing processors to achieve data partitioning, was achieved directly from

the processor's code. This is the supplier's code:

```
class WriteFilePSupplier implements ProcessorSupplier {

    private final String path;

    private transient List<WriteFileP> processors;

    WriteFilePSupplier(String path) {
        this.path = path;
    }

    @Override
    public void init(@Nonnull Context context) {
        File homeDir = new File(path);
        boolean success = homeDir.isDirectory() || homeDir.mkdirs();
        if (!success) {
            throw new JetException("Failed to create " + homeDir);
        }
    }

    @Override @Nonnull
    public List<WriteFileP> get(int count) {
        processors = Stream.generate(() -> new WriteFileP(path))
                           .limit(count)
                           .collect(Collectors.toList());
        return processors;
    }

    @Override
    public void complete(Throwable error) {
        for (WriteFileP p : processors) {
            try {
                p.close();
            } catch (IOException e) {
                throw new JetException(e);
            }
        }
    }
}
```

8.9. Best Practices

8.9.1. Inspecting Processor Input and Output

The structure of the DAG model is a very poor match for Java's type system, which results in the lack of compile-time type safety between connected vertices. Developing a type-correct DAG therefore usually requires some trial and error. To facilitate this process, but also to allow many more kinds of diagnostics and debugging, Jet's library offers ways to capture the input/output of a vertex and inspect it.

Peeking with Processor Wrappers

The first approach is to decorate a vertex declaration with a layer that will log all the data traffic going through it. This support is present in the `DiagnosticProcessors` factory class, which contains the following methods:

- `peekInput()`: logs items received at any edge ordinal.
- `peekOutput()`: logs items emitted to any ordinal. An item emitted to several ordinals is logged just once.

These methods take two optional parameters:

- `toStringF` returns the string representation of an item. The default is to use `Object.toString()`.
- `shouldLogF` is a filtering function so you can focus your log output only on some specific items. The default is to log all items.

Example Usage

Suppose we have declared the second-stage vertex in a two-stage aggregation setup:

```
Vertex combine = dag.newVertex("combine",
    combineByKey(counting()));
```

We'd like to see what exactly we're getting from the first stage, so we'll wrap the processor supplier with `peekInput()`:

```
Vertex combine = dag.newVertex("combine",
    peekInput(combineByKey(counting())));
```

Keep in mind that logging happens on the machine running hosting the processor, so this technique is primarily targeted to Jet jobs the developer runs locally in his development environment.

Attaching a Sink Vertex

Since most vertices are implemented to emit the same data stream to all attached edges, it is usually possible to attach a diagnostic sink to any vertex. For example, Jet's standard `writeFileP()` sink vertex

can be very useful here.

Example Usage

In the example from the Word Count tutorial we can add the following declarations:

```
Vertex diagnose = dag.newVertex("diagnose",
        Sinks.writeFile("tokenize-output"))
        .localParallelism(1);
dag.edge(from(tokenize, 1).to(diagnose));
```

This will create the directory `tokenize-output` which will contain one file per processor instance running on the machine. When running in a cluster, you can inspect on each member the input seen on that member. By specifying the `allToOne()` routing policy you can also have the output of all the processors on all the members saved on a single member (although the choice of exactly which member will be arbitrary).

8.9.2. How to Unit-Test a Processor

We provide some utility classes to simplify writing unit tests for your custom processors. You can find them in the `com.hazelcast.jet.core.test` package. Using these utility classes you can unit test your processor by passing it some input items and asserting the expected output.

Start by calling `TestSupport.verifyProcessor()` by passing it a processor supplier or a processor instance.

The test process does the following:

- initialize the processor by calling `Processor.init()`
- do a snapshot+restore (optional, see below)
- call `Processor.process(0, inbox)`. The inbox always contains one item from the `input` parameter
- every time the inbox gets empty, do a snapshot+restore
- call `Processor.complete()` until it returns `true` (optional)
- do a final snapshot+restore after `complete()` is done

The optional snapshot+restore test procedure:

- call `saveToSnapshot()`
- create a new processor instance and use it instead of the existing one
- restore the snapshot using `restoreFromSnapshot()`
- call `finishSnapshotRestore()`

The test optionally asserts that the processor made progress on each call to any processing method. To

be judged as having made progress, the callback method must do at least one of these:

- take something from the inbox
- put something to the outbox
- return `true` (applies only to `boolean`-returning methods)

Cooperative Processors

The test will provide a 1-capacity outbox to cooperative processors. The outbox will already be full on every other call to `process()`. This tests the edge case: `process()` may be called even when the outbox is full, giving the processor a chance to process the inbox without emitting anything.

The test will also assert that the processor doesn't spend more time in any callback than the limit specified in `cooperativeTimeout(long)`.

Cases Not Covered

This class does not cover these cases:

- testing of processors which distinguish input or output edges by ordinal
- checking that the state of a stateful processor is empty at the end (you can do that yourself afterwards with the last instance returned from your supplier)
- it never calls `Processor.tryProcess()`

Example Usage

This will test one of the jet-provided processors:

```
TestSupport.verifyProcessor(Processors.map((String s) -> s.toUpperCase()))
           .disableCompleteCall()             // enabled by default
           .disableLogging()                  // enabled by default
           .disableProgressAssertion()        // enabled by default
           .disableSnapshots()                // enabled by default
           .cooperativeTimeout(<timeoutInMs>) // default is 1000
           .outputChecker(<function>)         // default is `Objects::equal`
           .input(asList("foo", "bar"))       // default is `emptyList()`
           .expectOutput(asList("FOO", "BAR"));
```

Other Utility Classes

`com.hazelcast.jet.test` contains these classes that you can use as implementations of Jet interfaces in tests:

- `TestInbox`
- `TestOutbox`

- `TestProcessorContext`
- `TestProcessorSupplierContext`
- `TestProcessorMetaSupplierContext`

The class `JetAssert` contains a few of the `assertX()` methods normally found in JUnit's `Assert` class. We had to reimplement them to avoid a dependency on JUnit from our production code.

8.10. Custom DAG - Inverted TF-IDF Index

In this tutorial we'll explore what the Core API DAG model offers beyond the capabilities of the Pipeline API. Our DAG will feature splits, joins, broadcast, and prioritized edges. We'll access data from the file system and show a simple technique to distribute file reading across Jet members. Several vertices we use can't be implemented in terms of out-of-the-box processors, so we'll also show you how to implement your own with minimum boilerplate.

The full code is available at the `hazelcast-jet-code-samples` repository:

TfIdfJdkStreams.java

TfIdf.java

Let us first introduce the problem. The inverted index is a basic data structure in the domain of full-text search. First used in the 1950s, it is still at the core of modern information retrieval systems such as Lucene. The goal is to be able to quickly find the documents that contain a given set of search terms, and to sort them by relevance. To understand it we'll need to throw in some terminology.

- A *document* is treated as a list of words that has a unique ID. It is useful to define the notion of a *document index* which maps each document ID to the list of words it contains. We won't build this index; it's just for the sake of explanation.
- The *inverted index* is the inverse of the document index: it maps each word to the list of documents that contain it. This is the fundamental building block in our search algorithm: it will allow us to find in O(1) time all documents relevant to a search term.
- In the inverted index, each entry in the list is assigned a *TF-IDF score* which quantifies how relevant the document is to the search request.
- Let DF (*document frequency*) be the length of the list: the number of documents that contain the word.
- Let D be the total number of documents that were indexed.
- IDF (*inverse document frequency*) is equal to `log(D/DF)`.
- TF (*term frequency*) is the number of occurrences of the word in the document.
- TF-IDF score is simply the product of `TF * IDF`.

Note that IDF is a property of the word itself: it quantifies the relevance of each entered word to the search request as a whole. The list of entered words can be perceived as a list of filtering functions that

we apply to the full set of documents. A more relevant word will apply a stronger filter. Specifically, common words like "the", "it", "on" act as pure "pass-through" filters and consequently have an IDF of zero, making them completely irrelevant to the search.

TF, on the other hand, is the property of the combination of word and document, and tells us how relevant the document is to the word, regardless of the relevance of the word itself.

When the user enters a search phrase:

1. each individual term from the phrase is looked up in the inverted index;
2. an intersection is found of all the lists, resulting in the list of documents that contain all the words;
3. each document is scored by summing the TF-IDF contributions of each word;
4. the result list is sorted by score (descending) and presented to the user.

Let's have a look at a specific search phrase:

```
the man in the black suit murdered the king
```

The list of documents that contain all the above words is quite long… how do we decide which are the most relevant? The TF-IDF logic will make those stand out that have an above-average occurrence of words that are generally rare across all documents. For example, "murdered" occurs in far fewer documents than "black"… so given two documents where one has the same number of "murdered" as the other one has of "black", the one with "murdered" wins because its word is more salient in general. On the other hand, "suit" and "king" might have a similar IDF, so the document that simply contains more of both wins.

Also note the limitation of this technique: a phrase is treated as just the sum of its parts; a document may contain the exact phrase and this will not affect its score.

8.10.1. Building the Inverted Index with Java Streams

To warm us up, let's see what it takes to build the inverted index with just thread parallelism and without the ability to scale out across many machines. It is expressible in Java Streams API without too much work. The full code is here.

We'll start by preparing a `Stream<Entry<Long, String>> docWords`: a stream of all the words found in all the documents. We use `Map.Entry` as a holder of a pair of values (a 2-tuple) and here we have a pair of `Long docId` and `String word`:

```
Stream<Entry<Long, String>> docWords = docId2Name
        .entrySet()
        .parallelStream()
        .flatMap(TfIdfJdkStreams::docLines)
        .flatMap(this::tokenize);
```

We know the number of all documents so we can compute `double logDocCount`, the logarithm of the document count:

```
double logDocCount = Math.log(docId2Name.size());
```

Calculating TF is very easy, just count the number of occurrences of each distinct pair and save the result in a `Map<Entry<Long, String>, Long>`:

```
// TF map: (docId, word) -> count
final Map<Entry<Long, String>, Long> tfMap = docWords
        .parallel()
        .collect(groupingBy(identity(), counting()));
```

And now we build the inverted index. We start from `tfMap`, group by word, and the list under each word already matches our final product: the list of all the documents containing the word. We finish off by applying a transformation to the list: currently it's just the raw entries from the `tf` map, but we need pairs `(docId, tfIDfScore)`.

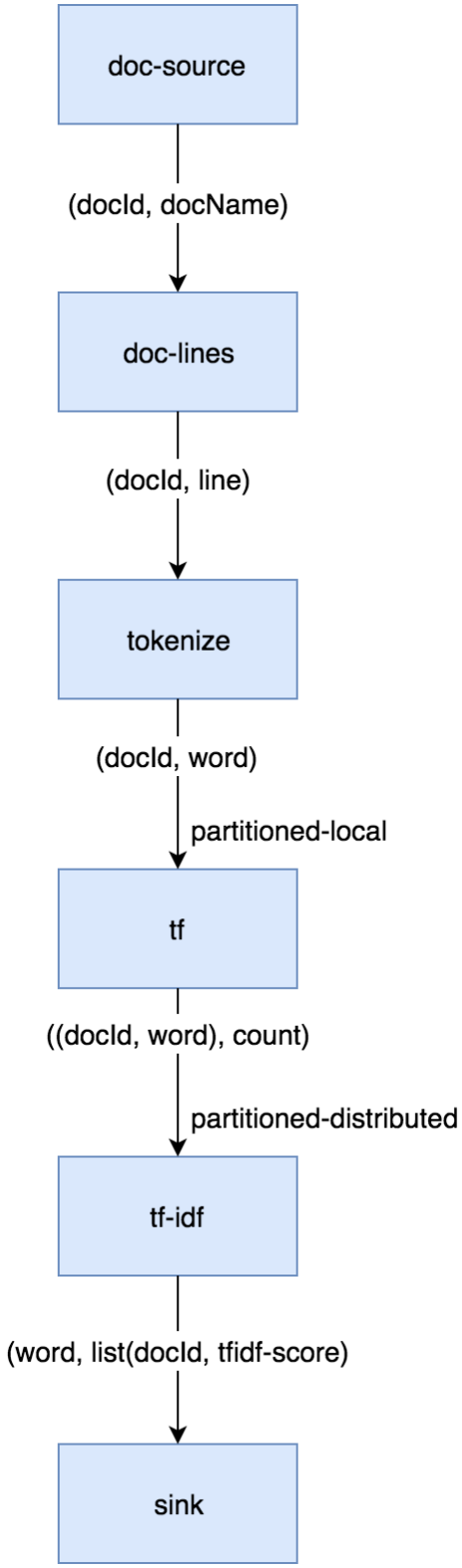
```
invertedIndex = tfMap
    .entrySet() // set of ((docId, word), count)
    .parallelStream()
    .collect(groupingBy(
        e -> e.getKey().getValue(),
        collectingAndThen(
            toList(),
            entries -> {
                double idf = logDocCount - Math.log(entries.size());
                return entries.stream()
                              .map(e -> tfidfEntry(e, idf))
                              .collect(toList());
            }
        )
    ));

// ((docId, word), count) -> (docId, tfIdf)
private static Entry<Long, Double> tfidfEntry(
        Entry<Entry<Long, String>, Long> tfEntry, Double idf
) {
    final Long tf = tfEntry.getValue();
    return entry(tfEntry.getKey().getKey(), tf * idf);
}
```

The search function can be implemented with another Streams expression, which you can review in the SearchGui class. You can also run the TfIdfJdkStreams class and take the inverted index for a spin, making actual searches.

There is one last concept in this model that we haven't mentioned yet: the *stopword set*. It contains those words that are known in advance to be common enough to occur in every document. Without treatment, these words are the worst case for the inverted index: the document list under each such word is the longest possible, and the score of all documents is zero due to zero IDF. They raise the index's memory footprint without providing any value. The cure is to prepare a file, `stopwords.txt`, which is read in advance into a `Set<String>` and used to filter out the words in the tokenization phase. The same set is used to cross out words from the user's search phrase, as if they weren't entered. We'll add this feature to our DAG based model in the following section.

8.10.2. Translating to Jet DAG

Our DAG as a whole will look relatively complex, but it can be understood as a "backbone" (cascade of vertices) starting from a source and ending in a sink with several more vertices attached on the side. This is just the backbone:

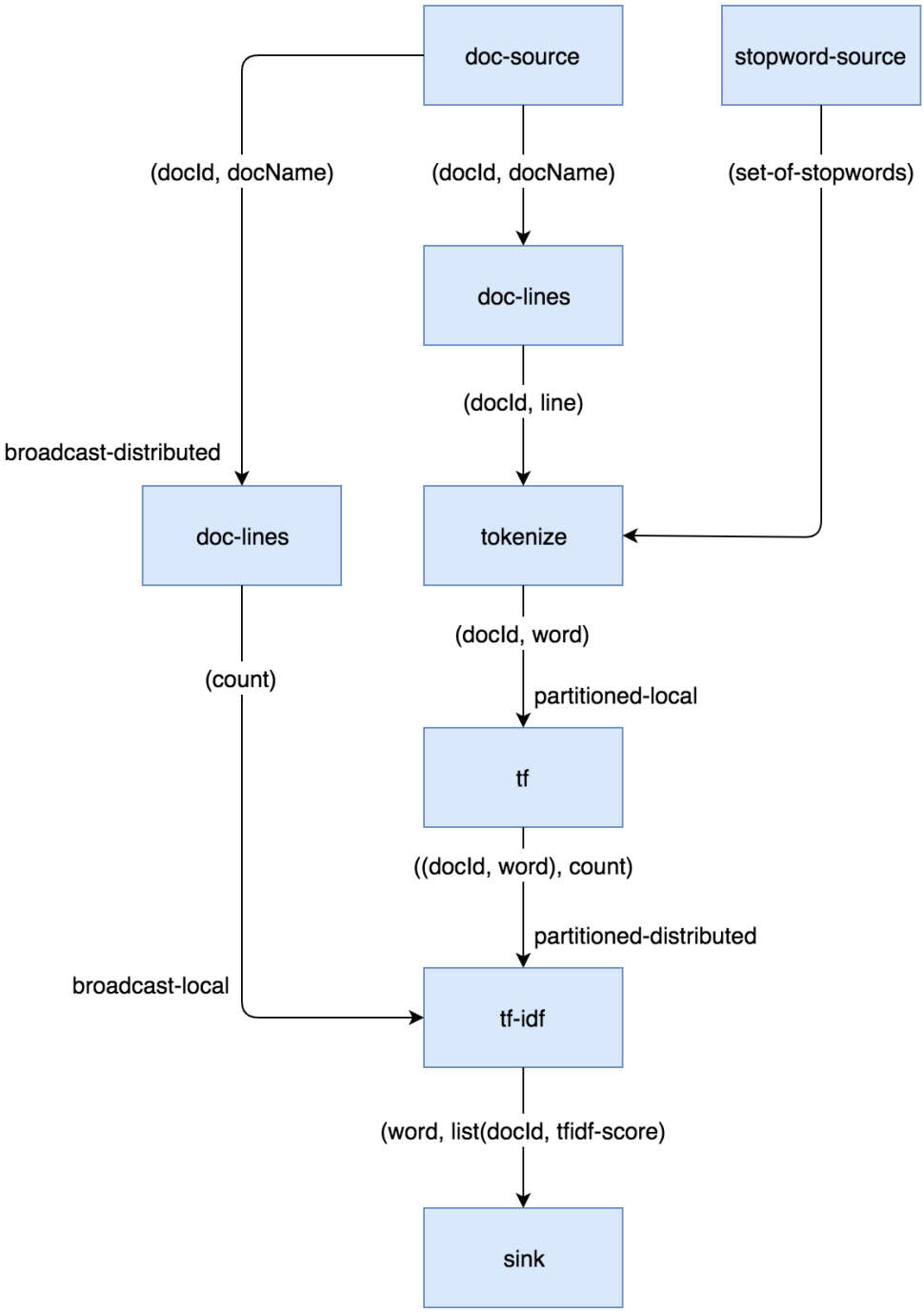


1. The data source is a Hazelcast `IMap` which holds a mapping from document ID to its filename. The source vertex will emit all the map's entries, but only a subset on each cluster member.
2. `doc-lines` opens each file named by the map entry and emits all its lines in the `(docId, line)` format.
3. `tokenize` transforms each line into a sequence of its words, again paired with the document ID, so it emits `(docId, word)`.

4. `tf` builds a set of all distinct pairs emitted from `tokenize` and maintains the count of each pair's occurrences (its TF score).
5. `tf-idf` takes that set, groups the pairs by word, and calculates the TF-IDF scores. It emits the results to the sink, which saves them to a distributed `IMap`.

Edge types follow the same pattern as in the word-counting job: after flatmapping there is first a local, then a distributed partitioned edge. The logic behind it is not the same, though: TF can actually compute the final TF scores by observing just the local data. This is because it treats each document separately (document ID is a part of the grouping key) and the source data is already partitioned by document ID. The TF-IDF vertex does something similar to word count's combining, but there's again a twist: it will group the TF entries by word, but instead of just merging them into a single result per word, it will keep them all in lists.

To this cascade we add a `stopword-source` which reads the stopwords file, parses it into a `HashSet`, and sends the whole set as a single item to the `tokenize` vertex. We also add a vertex that takes the data from `doc-source` and simply counts its items; this is the total document count used in the TF-IDF formula. We end up with this DAG:



The choice of edge types into and out of `doc-count` may look surprising, so let's examine it. We start with the `doc-source` vertex, which emits one item per document, but its output is distributed across the cluster. To get the full document count on each member, each `doc-count` processor must get all the items, and that's just what the distributed broadcast edge will achieve. We'll configure `doc-count` with local parallelism of 1, so there will be one processor on every member, each observing all the `doc-source` items. The output of `doc-count` must reach all `tf-idf` processors on the same member, so we use the local broadcast edge.

Another thing to note are the two flat-mapping vertices: `doc-lines` and `tokenize`. From a purely semantic standpoint, composing flatmap with flatmap yields just another flatmap. As we'll see below, we're using custom code for these two processors… so why did we choose to separate the logic this

way? There are actually two good reasons. The first one has to do with Jet's cooperative multithreading model: `doc-lines` makes blocking file IO calls, so it must be declared *non-cooperative*; tokenization is pure computation so it can be in a *cooperative* processor. The second one is more general: the workload of `doc-lines` is very uneven. It consists of waiting, then suddenly coming up with a whole block of data. If we left tokenization there, performance would suffer because first the CPU would be forced to sit idle, then we'd be late in making the next IO call while tokenizing the input. The separate vertex can proceed at full speed all the time.

8.10.3. Implementation Code

As we announced, some of the processors in our DAG will need custom implementation code. Let's start from the source vertex. It is easy, just the standard `IMap` reader:

```
dag.newVertex("doc-source", SourceProcessors.readMapP(DOCID_NAME));;
```

The stopwords-producing processor has custom code, but it's quite simple:

```
dag.newVertex("stopword-source", StopwordsP::new);
```

```
private static class StopwordsP extends AbstractProcessor {
    @Override
    public boolean complete() {
        return tryEmit(docLines("stopwords.txt").collect(toSet()));
    }
}
```

Since this is a source processor, all its action happens in `complete()`. It emits a single item: the `HashSet` built directly from the text file's lines.

The `doc-count` processor can be built from the primitives provided in Jet's library:

```
dag.newVertex("doc-count", Processors.aggregateP(counting()));
```

The `doc-lines` processor is more of a mouthful, but still built from existing primitives:

```
dag.newVertex("doc-lines",
    Processors.nonCooperativeP(
        Processors.flatMapP((Entry<Long, String> e) ->
            traverseStream(docLines("books/" + e.getValue())
                           .map(line -> entry(e.getKey(), line))))));
```

Let's break down this expression... `Processors.flatMap` returns a standard processor that emits an arbitrary number of items for each received item. We already saw one in the introductory Word Count example. There we created a traverser from an array, here we create it from a Java stream. We additionally apply the `nonCooperative()` wrapper which will declare all the created processors non-cooperative. We already explained why we do this: this processor will make blocking I/O calls.

`tokenizer` is another custom vertex:

```
dag.newVertex("tokenize", TokenizeP::new);

private static class TokenizeP extends AbstractProcessor {
    private Set<String> stopwords;
    private final FlatMapper<Entry<Long, String>, Entry<Long, String>> flatMapper =
        flatMapper(e -> traverseStream(
                    Arrays.stream(DELIMITER.split(e.getValue()))
                          .filter(word -> !stopwords.contains(word))
                          .map(word -> entry(e.getKey(), word))));

    @Override
    protected boolean tryProcess0(@Nonnull Object item) {
        stopwords = (Set<String>) item;
        return true;
    }

    @Override
    protected boolean tryProcess1(@Nonnull Object item) {
        return flatMapper.tryProcess((Entry<Long, String>) item);
    }
}
```

This is a processor that must deal with two different inbound edges. It receives the stopword set over edge 0 and then it does a flatmapping operation on edge 1. The logic presented here uses the same approach as the implementation of the provided `Processors.flatMap()` processor: there is a single instance of `FlatMapper` that holds the business logic of the transformation, and `tryProcess1()` directly delegates into it. If `FlatMapper` is done emitting the previous items, it will accept the new item, apply the user-provided transformation, and start emitting the output items. If the outbox refuses a pending item, it will return `false`, which will make the framework call the same `tryProcess1()` method later, with the same input item.

Let's show the code that creates the `tokenize's two inbound edges:

```
dag.edge(between(stopwordSource, tokenize).broadcast().priority(-1))
   .edge(from(docLines).to(tokenize, 1));
```

Especially note the `.priority(-1)` part: this ensures that there will be no attempt to deliver any data

coming from `docLines` before all the data from `stopwordSource` is already delivered. The processor would fail if it had to tokenize a line before it has its stopword set in place.

`tf` is another simple vertex, built purely from the provided primitives:

```
dag.newVertex("tf", Processors.aggregateByKeyP(wholeItem(), counting()));
```

`tf-idf` is the most complex processor:

```
dag.newVertex("tf-idf", TfIdfP::new);

private static class TfIdfP extends AbstractProcessor {
    private double logDocCount;

    private final Map<String, List<Entry<Long, Double>>> wordDocTf = new HashMap<>();
    private final Traverser<Entry<String, List<Entry<Long, Double>>>>
invertedIndexTraverser =
            lazy(() -> traverseIterable(wordDocTf.entrySet()).map(this:
:toInvertedIndexEntry));

    @Override
    protected boolean tryProcess0(@Nonnull Object item) throws Exception {
        logDocCount = Math.log((Long) item);
        return true;
    }

    @Override
    protected boolean tryProcess1(@Nonnull Object item) throws Exception {
        final Entry<Entry<Long, String>, Long> e = (Entry<Entry<Long, String>, Long>)
item;
        final long docId = e.getKey().getKey();
        final String word = e.getKey().getValue();
        final long tf = e.getValue();
        wordDocTf.computeIfAbsent(word, w -> new ArrayList<>())
                 .add(entry(docId, (double) tf));
        return true;
    }

    @Override
    public boolean complete() {
        return emitFromTraverser(invertedIndexTraverser);
    }

    private Entry<String, List<Entry<Long, Double>>> toInvertedIndexEntry(
            Entry<String, List<Entry<Long, Double>>> wordDocTf
    ) {
```

```
        final String word = wordDocTf.getKey();
        final List<Entry<Long, Double>> docidTfs = wordDocTf.getValue();
        return entry(word, docScores(docidTfs));
    }

    private List<Entry<Long, Double>> docScores(List<Entry<Long, Double>> docidTfs) {
        final double logDf = Math.log(docidTfs.size());
        return docidTfs.stream()
                       .map(tfe -> tfidfEntry(logDf, tfe))
                       .collect(toList());
    }

    private Entry<Long, Double> tfidfEntry(double logDf, Entry<Long, Double> docidTf) {
        final Long docId = docidTf.getKey();
        final double tf = docidTf.getValue();
        final double idf = logDocCount - logDf;
        return entry(docId, tf * idf);
    }
}
```

This is quite a lot of code, but each of the three pieces is not too difficult to follow:

1. `tryProcess0()` accepts a single item, the total document count.
2. `tryProcess1()` performs a boilerplate `groupBy` operation, collecting a list of items under each key.
3. `complete()` outputs the accumulated results, also applying the final transformation on each one: replacing the TF score with the final TF-IDF score. It relies on a *lazy* traverser, which holds a `Supplier<Traverser>` and will obtain the inner traverser from it the first time `next()` is called. This makes it very simple to write code that obtains a traverser from a map after it has been populated.

Finally, our DAG is terminated by a sink vertex:

```
dag.newVertex("sink", SinkProcessors.writeMapP(INVERTED_INDEX));
```

Chapter 9. Miscellaneous

9.1. Phone Homes

Hazelcast uses phone home data to learn about the usage of Hazelcast Jet.

Hazelcast Jet instances call our phone home server initially when they are started and then every 24 hours. This applies to all the instances joined to the cluster.

What is sent in?

The following information is sent in a phone home:

- Hazelcast Jet version
- Local Hazelcast Jet member UUID
- Download ID
- A hash value of the cluster ID
- Cluster size bands for 5, 10, 20, 40, 60, 100, 150, 300, 600 and > 600
- Number of connected clients bands of 5, 10, 20, 40, 60, 100, 150, 300, 600 and > 600
- Cluster uptime
- Member uptime
- Environment Information:
  - Name of operating system
  - Kernel architecture (32-bit or 64-bit)
  - Version of operating system
  - Version of installed Java
  - Name of Java Virtual Machine
- Hazelcast IMDG Enterprise specific:
  - Number of clients by language (Java, C++, C#)
  - Flag for Hazelcast Enterprise
  - Hash value of license key
  - Native memory usage

Phone Home Code

The phone home code itself is open source. Please see here.

Disabling Phone Homes

Set the `hazelcast.phone.home.enabled` system property to false either in the config or on the Java command line.

Starting with Hazelcast Jet 0.5, you can also disable the phone home using the environment variable `HZ_PHONE_HOME_ENABLED`. Simply add the following line to your `.bash_profile`:

```
export HZ_PHONE_HOME_ENABLED=false
```

Phone Home URL

The URL used for phone home requests is http://phonehome.hazelcast.com/ping

9.2. License Questions

Hazelcast Jet is distributed using the Apache License 2, therefore permissions are granted to use, reproduce and distribute it along with any kind of open source and closed source applications.

Depending on the used feature-set, Hazelcast Jet has certain runtime dependencies which might have different licenses. Following are dependencies and their respective licenses.

9.2.1. Embedded Dependencies

Embedded dependencies are merged (shaded) with the Hazelcast Jet codebase at compile-time. These dependencies become an integral part of the Hazelcast Jet distribution.

For license files of embedded dependencies, please see the `license` directory of the Hazelcast Jet distribution, available at our download page.

minimal-json

minimal-json is a JSON parsing and generation library which is a part of the Hazelcast Jet distribution. It is used for communication between the Hazelcast Jet cluster and the Management Center.

minimal-json is distributed under the MIT license and offers the same rights to add, use, modify, and distribute the source code as the Apache License 2.0 that Hazelcast uses. However, some other restrictions might apply.

Runtime Dependencies

Depending on the used features, additional dependencies might be added to the dependency set. Those runtime dependencies might have other licenses. See the following list of additional runtime dependencies.

Apache Hadoop

Hazelcast integrates with Apache Hadoop and can use it as a data sink or source. Jet has a dependency on the libraries required to read from and write to the Hadoop File System.

Apache Hadoop is distributed under the terms of the Apache License 2.

Apache Kafka

Hazelcast integrates with Apache Kafka and can make use of it as a data sink or source. Hazelcast has a dependency on Kafka client libraries.

Apache Kafka is distributed under the terms of the Apache License 2.

9.3. FAQ

You can refer to the FAQ page to see the answers to frequently asked questions related to topics such as the relationship and differences between Hazelcast Jet and Hazelcast IMDG, Jet's APIs and roadmap.

9.4. Common Exceptions

You may see the following exceptions thrown when working with Jet:

- `JetException`: A general exception that will be thrown if a job failure occurs. It will have the original exception in the cause field.
- `TopologyChangedException`: This exception is thrown when a member participating in a job leaves the cluster. The job will typically be restarted automatically without throwing the exception to the user if auto-restart is enabled.
- `JobNotFoundException`: Thrown when the coordinator node is not able to find the metadata for a given job.

Furthermore, there are several Hazelcast exceptions that might be thrown when interacting with `JetInstance`. For description of Hazelcast IMDG exceptions, please refer to the IMDG Reference manual.

Chapter 10. Glossary

Term	Definition
Accumulation	The act of building up an intermediate result inside a mutable object (called the *accumulator*) as a part of performing an aggregate operation. After all accumulation is done, a *finishing* function is applied to the object to produce the result of the operation.
Aggregate Operation	A set of functional primitives that instructs Jet how to calculate some aggregate function over one or more data sets. Used in the group-by, co-group and windowing transforms.
Aggregation	The act of applying an *aggregate function* to a stream of items. The result of the function can be simple, like a sum or average, or complex, like a collection of all aggregated items.
At-Least-Once Processing Guarantee	The system guarantees to process each item of the input stream(s), but doesn’t guarantee it will process it just once.
Batch Processing	The act of processing a finite dataset, such as one stored in Hazelcast IMDG or HDFS.
Client Server Topology	Hazelcast topology where members run outside the user application and are connected to clients using client libraries. The client library is installed in the user application.
Co-Grouping	An operation that is a mix of an SQL JOIN and GROUP BY with specific restrictions. Sometimes called a "stream join". Each item of each stream that is being joined must be mapped to its grouping key. All items with the same grouping key (from all streams) are aggregated together into a single result. However, the result can be structured and preserve all input items separated by their stream of origin. In that form the operation effectively becomes a pure JOIN with no aggregation.
DAG	Directed Acyclic Graph which Hazelcast Jet uses to model the relationships between individual steps of the data processing.

Term	Definition
Edge	A DAG element which holds the logic on how to route the data from one vertex's processing units to the next one's.
Embedded Topology	Hazelcast topology where the members are in-process with the user application and act as both client and server.
Event time	A data item in an infinite stream typically contains a timestamp data field. This is its *event time*. As the stream items go by, the event time passes as the items' timestamps increase. A typical distributed stream has a certain amount of event time disorder (items aren't strictly ordered by their timestamp) so the "passage of event time" is a somewhat fuzzy concept. Jet uses the *watermark* to superimpose order over the disordered stream.
Exactly-Once Processing Guarantee	The system guarantees that it will process each item of the input stream(s) and will never process an item more than once.
Fault Tolerance	The property of a distributed computation system that gives it resilience to changes in the topology of the cluster running the computation. If a member leaves the cluster, the system adapts to the change and resumes the computation without loss.
Hash-Join	A special-purpose stream join optimized for the use case of data enrichment. Each item of the *primary* stream is joined with one item from each of the *enriching* streams. Items are matched by the join key. The name "hash-join" stems from the fact that the contents of the enriching streams are held in hashtables for fast lookup. Hashtables are replicated on each cluster member, which is why this operation is also known as a "replicated join".
Hazelcast IMDG	An In-Memory Data grid (IMDG) is a data structure that resides entirely in memory, and is distributed among many machines in a single location (and possibly replicated across different locations). IMDGs can support millions of in-memory data updates per second, and they can be clustered and scaled in ways that support large quantities of data. Hazelcast IMDG is the in-memory data grid offered by Hazelcast.

Term	Definition
HDFS	Hadoop Distributed File System. Hazelcast Jet can use it both as a data source and a sink.
Jet Job	A unit of distributed computation that Jet executes. One job has one DAG specifying what to do. A distributed array of Jet processors performs the computation.
Kafka	Apache Kafka is a product that offers a distributed publish-subscribe message queue with guarantees of delivery and message persistence. The most commonly used component over which heterogenous distributed systems exchange data.
Latency	The time that passes from the occurrence of an event that triggers some response to the occurrence of the response. In the case of Hazelcast Jet's stream processing, latency refers to the time that passes from the point in time the last item that belongs to a window enters the system to the point where the result for that window appears in the output.
Member	A Hazelcast Jet instance (node) that is a member of a cluster. A single JVM can host one or more Jet members, but in production there should be one member per physical machine.
Partition (Data)	To guarantee that all items with the same grouping key are processed by the same processor, Hazelcast Jet uses a total surjective function to map each data item to the ID of its partition and assigns to each processor its unique subset of all partition IDs. A partitioned edge then routes all items with the same partition ID to the same processor.
Partition (Network)	A malfunction in network connectivity that splits the cluster into two or more parts that are mutually unreachable, but the connections among nodes within each part remain intact. May cause each of the parts to behave as if it was "the" cluster that lost the other members. Also known as "split brain".

Term	Definition
Pipeline	Hazelcast Jet's name for the high-level description of a computation job constructed using the Pipeline API. Topologically it is a DAG, but the vertices have different semantics than the Core API vertices and are called *pipeline stages*. Edges are implicit and not expressed in the API. Each stage (except for source/sink stages) has an associated *transform* that it performs on its input data.
Processor	The unit which contains the code of the computation to be performed by a vertex. Each vertex's computation is implemented by a processor. On each Jet cluster member there are one or more instances of the processor running in parallel for a single vertex.
Session Window	A window that groups an infinite stream's items by their timestamp. It groups together bursts of events closely spaced in time (by less than the configured session timeout).
Sliding Window	A window that groups an infinite stream's items by their timestamp. It groups together events that belong to a segment of fixed size on the timeline. As the time passes, the segment slides along, always extending from the present into the recent past. In Jet, the window doesn't literally slide, but hops in steps of user-defined size. ("Time" here refers to the stream's own notion of time, i.e., *event time*.)
Source	A resource present in a Jet job's environment that delivers a data stream to it. Hazelcast Jet uses a *source connector* to access the resource. Alternatively, *source* may refer to the DAG vertex that hosts the connector.
Sink	A resource present in a Jet job's environment that accepts its output data. Hazelcast Jet uses a *sink connector* to access the resource. Alternatively, *sink* may refer to the vertex that hosts the connector.

Term	Definition
Skew	A generalization of the term "clock skew" applied to distributed stream processing. In this context it refers to the deviation in *event time* as opposed to wall-clock time in the classical usage. Several substreams of a distributed stream may at the same time emit events with timestamps differing by some delta, due to various lags that accumulate in the delivery pipeline for each substream. This is called *stream skew*. *Event skew* refers to the disorder within a substream, where data items appear out of order with respect to their timestamps.
Split Brain	A popular name for a *network partition*, which see above.
Stream Processing	The act of processing an infinite stream of data, typically implying that the data is processed as soon as it appears. Such a processing job must explicitly deal with the notion of time in order to make sense of the data. It achieves this with the concept of *windowing*.
Throughput	A measure for the volume of data a system is capable of processing per unit of time. Typical ways to express it for Hazelcast Jet are in terms of events per second and megabytes per second.
Tumbling Window	A window that groups an infinite stream's items by their timestamp. It groups together events that belong to a segment of fixed size on the timeline. As the time passes, the segment "tumbles" along, never covering the same point in time twice. This means that each event belongs to just one tumbling window position. ("Time" here refers to the stream's own notion of time, i.e., *event time*.)
Vertex	The DAG element that performs a step in the overall computation. It receives data from its inbound edges and sends the results of its computation to its outbound edges. There are three kinds of vertices: source (has only outbound edges), sink (has only inbound edges) and computational (has both kinds of edges).

Term	Definition
Watermark	A concept that superimposes order over a disordered underlying data stream. An infinite data stream's items represent timestamped events, but they don't occur in the stream ordered by the timestamp. The value of the watermark at a certain location in the processing pipeline denotes the lowest value of the timestamp that is expected to occur in the upcoming items. Items that don't meet this criterion are discarded because they arrived too late to be processed.
Windowing	The act of splitting an infinite stream's data into *windows* according to some rule, most typically one that involves the item's timestamps. Each window becomes the target of an aggregate function, which outputs one data item per window (and per grouping key).