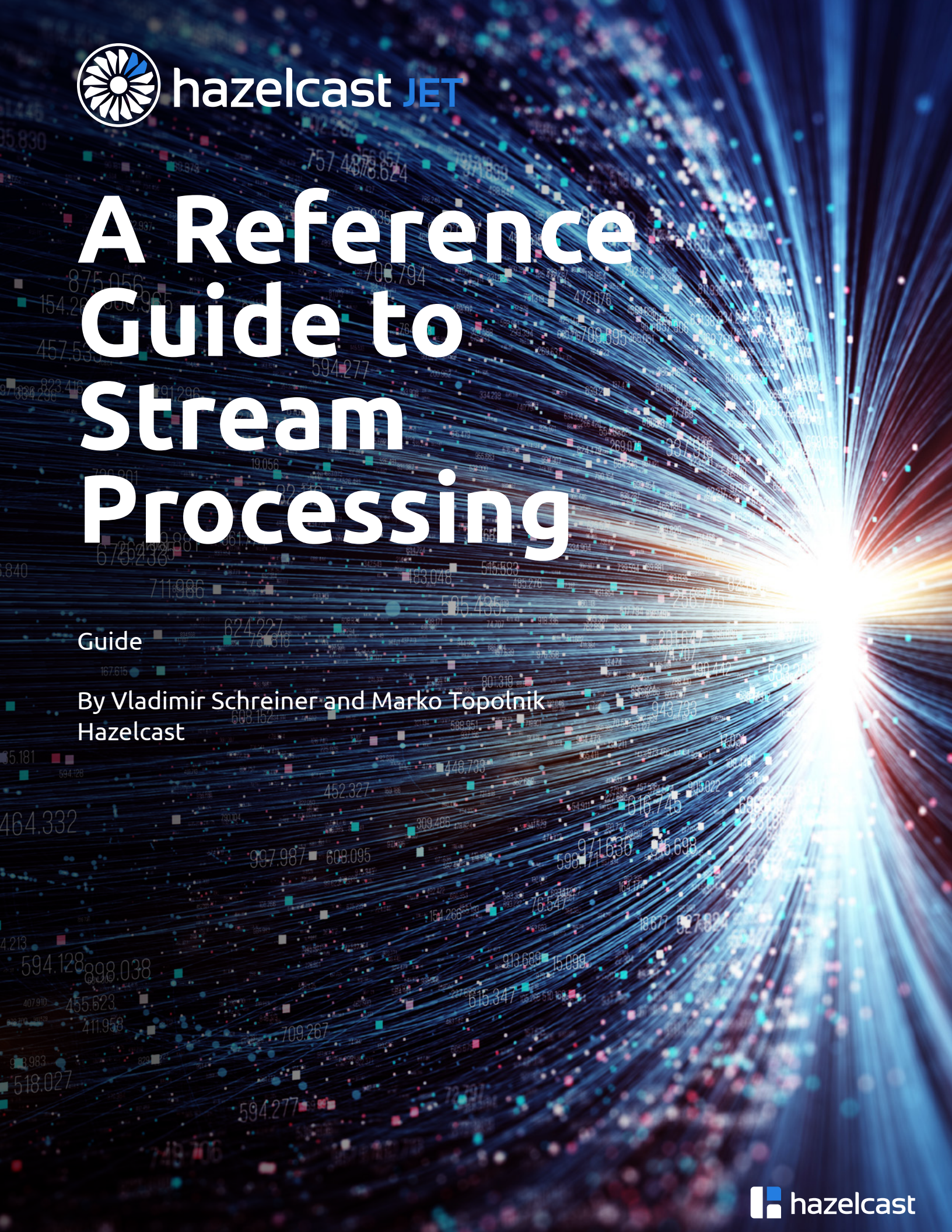

hazelcast JET
A Reference Guide to Stream Processing
Guide
By Vladimir Schreiner and Marko Topolnik
Hazelcast
hazelcast

Table of Contents

Understanding Stream Processing

Fast Processing of Infinite and Big Data

This white paper introduces you to the domain of stream processing. It covers these topics:

- Use cases that benefit from stream processing
- Building blocks of a stream processing solution
- Key concepts used when building a streaming pipeline: definition of the data flow, keyed aggregation, windowing
- Runtime aspects and tradeoffs between performance and correctness
- Overview of distributed stream processing engines
- Hands-on examples based on Hazelcast Jet®

What is Stream Processing

The goal of streaming systems is to process big data volumes and provide useful insights into the data prior to saving it to long-term storage.

The traditional approach to processing data at scale is batching. For example, a bank collects the transactional data during the day into a storage system such as HDFS or a data warehouse. Then, after closing time, it processes it into an offline batch job. The premise of this approach is that all the data is available in the system of record before the processing starts. In the case of failures, the whole job can be simply restarted.

While quite simple and robust, this approach clearly introduces a large latency between gathering the data and being ready to act upon it.

The goal of stream processing is to overcome this latency. It processes the live, raw data immediately as it arrives and meets the challenges of incremental processing, scalability and fault tolerance.

When to Use Stream Processing

Fast Big Data

Stream processing should be used in systems which handle big data volumes and where real-time results matter. In other words, when the value of the information contained in the data stream decreases rapidly as it gets older.

This mostly applies to:

- Real-time analytics — for fast business insights and decision making.
- Anomaly, fraud or pattern detection.
- Complex event processing.
- Real-time stats — monitoring, feeding the real-time dashboards.
- Real-time ETL (extract, transform, load).
- Implementing event driven architectures — stream processor builds materialized views on top of the event stream.

Continuous Data

Batch processing forces you to split your data into isolated blocks. The information in the data that crosses the border of batches gets lost.

Example: You analyze the behavior of users browsing your website. You run an analytic task on a daily basis, processing one day batches. What if somebody keeps browsing your site around midnight? His or her behavior is divided between two batches and the correlation is lost.

Streams, on the other hand, are unbounded by definition. As the data stream is potentially infinite, as is the computation, your insight into the data isn't limited by the underlying technical solution.

Consistent Resource Consumption

Also, processing data as it arrives spreads out workloads more evenly over time. Stream processing should be used to make the consumption of resources more consistent and predictable.

The Building Blocks

A **stream** is a sequence of **records**. Each record holds information about an event that happened, such as a user's access to a web site, temperature update from an Internet-of things (IoT) sensor, or trade being completed. The records are immutable, as they reflect something that had already happened — the user accessed the web site, the thermometer captured the temperature, the trade was processed. This cannot be undone. Also, the stream is potentially infinite, as the events just keep happening.

The **stream processing application** provides insight into the stream, the current state computed on top of the individual events flowing in the stream. Such as current count of users accessing a web site, or the maximum temperature measured in an engine in the last hour.

This is being done by application of **transformations** on top of the data stream. Multiple transformations could be composed to do more complex transformations. The next chapter will introduce the most frequently used transformations.

The stream processing application ingests one or more streams from **stream sources**. Source connectors are used to connect the stream processing application to the system that feeds the data, such as Apache Kafka, JMS broker, IoT sensor or custom enterprise system.

Sinks are used to pass the transformed data downstream for storage or further processing. One streaming application can read data from multiple sources and output data to multiple sinks.

The stream processing application runs on a **stream processing engine (SPE) or platform** — infrastructure allowing you to focus on business logic and transformations instead of low-level distributed computation concerns.

To build a stream processing application, you must:

- Define the transformations.
- Connect the streaming application to stream sources and sinks. Stream processing platforms mostly provide a set of connectors, you have to configure them properly.
- Define the dataflow of the application by wiring the sources to transformations and sinks.
- Execute the stream processing application using the stream processing engine. The engine then takes care of passing the records between the system components (sources, processors, sinks) and invoking them when the record arrives. The application consumes and processes the stream until it is stopped.

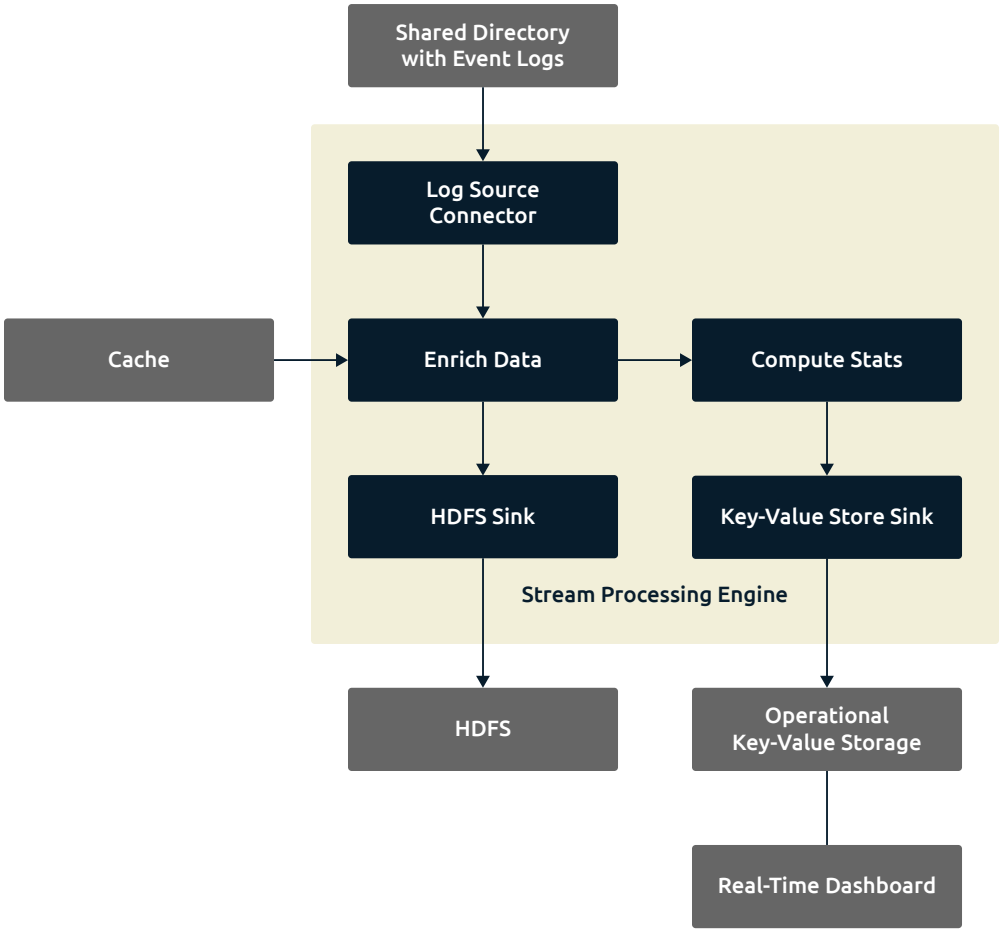
This white paper will guide you through the building blocks in detail.

Example

Let's consider an application that processes a stream of system log events in order to power a dashboard giving immediate insight into what's going on with the system. These are the steps it will take:

- Read the raw data from log files in a watched directory.
- Parse and filter it, transforming it into event objects.
- *Enrich* the events: using a machine ID field in the event object, look up the data we have on that machine and attach it to the event.
- Save the enriched, denormalized data to persistent storage such as Cassandra or HDFS for deeper offline analysis (first data sink).
- Perform *windowed aggregation*: using the attached data calculate real-time statistics over the events that happened within the time window of the last minute.
- Push the results to an in-memory Key-Value store that backs the dashboard (second data sink).

We can represent each of the above steps with a box in a diagram and connect them using arrows representing the data flow:



The boxes and arrows form a graph, specifically a directed acyclic graph (DAG). This model is at the core of modern stream processing engines.

Here's how you would describe the above DAG in the Hazelcast Jet Pipeline API. It uses the paradigm of a "pipeline" that consists of interconnected "stages".

```
Pipeline p = Pipeline.create();
// Read the logs, parse each line, keep only success responses
StreamStage<LogEvent> events = p.drawFrom(fileWatcher("log-directory"))
       .withoutTimestamps()
       .map(LogEvent::parse)
       .filter(event -> event.responseCode() >= 200 && event.responseCode() < 400);

// Enrich with machine data
StreamStage<Map.Entry<Long, LogEvent>> enrichedEvents = events.hashJoin(
       p.drawFrom(Sources.<String, Machine>map("machines")),
       joinMapEntries(LogEvent::machineID),
       (e, m) -> entry(e.sequenceNumber(), e.withMachine(m)));

// Save enriched events to HDFS
JobConf jobConfig = new JobConf();
jobConfig.setOutputFormat(TextOutputFormat.class);
TextOutputFormat.setOutputPath(jobConfig, new Path("output"));
enrichedEvents.drainTo(HdfsSinks.hdfs(jobConfig));

// Calculate requests per minute for each machine,
// save stats to IMap
events.addTimestamps(LogEvent::timestamp, 0)
       .window(sliding( MINUTES.toMillis(1), SECONDS.toMillis(1)))
       .groupingKey(LogEvent::machineID)
       .aggregate(counting())
       .drainTo(Sinks.map("machine-stats"));
```

Transformations

Transformations are used to express the business logic of a streaming application. On the low level, a processing task receives some stream items, performs arbitrary processing, and emits some items. It may emit items even without receiving anything (acting as a *stream source*) or it may just receive and not emit anything (acting as a *sink*).

Due to the nature of distributed computation you can't just provide arbitrary imperative code that processes the data, you must describe it declaratively. This is why streaming applications share some principles with functional and dataflow programming. This requires some time to get used to when coming from the imperative programming.

The Hazelcast Jet Pipeline API is one such example: you compose a *pipeline* from individual *stages*, each performing one kind of transformation. A simple map stage transforms items with a stateless function; a more complex *windowed group-and-aggregate* stage groups events by key in an infinite stream and calculates an aggregated value over a sliding window. You provide just the business logic such as the function to extract the grouping key, the definition of the aggregate function, the definition of the sliding window, etc.

Basic Transformations

We have already mentioned that a stream is a sequence of isolated records. Many basic transformations process each record independently. Such a transformation is **stateless**.

These are the main types of stateless transformation:

- **Map** transforms one record to one record, for example: change format of the record, enrich record with some data.
- **Filter** filters out the records that doesn't satisfy the predicate.
- **FlatMap** is the most general type of stateless transformation, outputting zero or more records for each input record, for example: tokenize a record containing a sentence into individual words.

However, many types of computation involve more than one record. In this case the processor must maintain internal state across the records. When counting the records in the stream, for example, you have to maintain the current count.

Stateful transformations:

- **Aggregation** combines all the records to produce a single value. Examples: min, max, sum, count, avg.
- **Group-and-aggregate** extracts a grouping key from the record and computes a separate aggregated value for each key.
- **Join** joins same-keyed records from several streams.
- **Sort** sorts the records observed in the stream.

This code samples shows both stateless and stateful transformations:

```
    logLines.map(LogEvent::parse)
           .filter((LogEvent event) -> event.responseCode() >= 200 && event.responseCode() < 400)
           .flatMap(LogEvent::urlSegments)
           .groupingKey(wholeItem())
           .aggregate(counting());
```

In the most general case, the state of stateful transformations is affected by all the records observed in the stream, all the ingested records are involved in the computation. However, we're mostly interested in something like "stats for last 30 seconds" instead of "stats since streaming app was started" (remember — the stream is generally infinite). This is where the concept of *windowing* enters the picture — it meaningfully bounds the scope of the aggregation. See the Windowing section.

Keyed or Non-keyed Aggregations

You often need to classify records by a grouping key, thereby creating sub-streams containing just the records with the same grouping key. Each record group is then processed separately. This fact can be leveraged for easy parallelization by *partitioning* the data stream on the grouping key and letting independent threads/processes/machines handle records with different keys.

Examples of keyed aggregations

This example processes a stream of text snippets (tweets or anything else) by first splitting them into individual words and then performing a windowed group-and-aggregate operation. The aggregate function is simply counting the items in each group. This results in a live word frequency histogram that updates as the window slides along the time axis. Although quite a simple operation, it gives a powerful insight into the contents of the stream.

```
tweets.flatMap(tweet -> traverseArray(tweet.toLowerCase().split("\\W+")))
       .groupingKey(wholeItem())
       .window(sliding(SECONDS.toMillis(10), 100))
       .aggregate(counting())
       .map( windowResult -> entry(windowResult.key(), windowResult.getValue()));
```

Another example of keyed aggregation could be gaining insight into the activities of all your users. You key the stream by user ID and write your aggregation logic focused on a single user. The processing engine will automatically distribute the load of processing user data across all the machines in the cluster and all their CPU cores.

Example of non-keyed (global) aggregation

This example processes a stream of reports from weather stations. Among the reports from the last hour it looks for the one that indicated the strogest winds.

```
weatherStationReports
       .window(sliding(HOURS.toMillis(1), MINUTES.toMillis(1)))
       .aggregate(maxBy(comparing(WeatherStationReport::windSpeed)))
       .map(windowResult -> entry(windowResult.end(), windowResult.result()));
```

A general class of use cases where non-keyed aggregation is useful are complex event processing (CEP) applications. They search for complex patterns in the data stream. In such a case there is no a-priori partitioning you can apply to the data; the pattern-matching operator needs to see the whole dataset to be able to detect a pattern.

Windowing

Windows provide you with a finite, bounded view on top of an infinite stream. The window defines how records are selected from the stream and grouped together to a meaningful frame. Your transformation is executed just on the records contained in the window.

How to Define Windows?

A very simple example is a **tumbling window**. It divides the continuous stream into discrete parts that don't overlap. The window is usually defined by a time duration or record count. The new window is opened as soon as the time passes (for time-based windows) or as soon as the count reaches the limit (count-based windows).

Examples:

- Time-based tumbling windows: counting system usage stats, i. e. a count of accesses in the last minute.
- Count-based tumbling windows: maximum score in a gaming system over last 1,000 results.

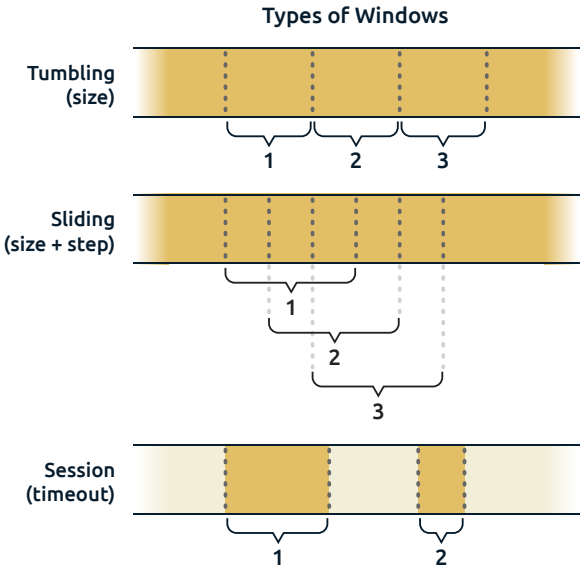
The **sliding window** is also of a fixed size, however consecutive windows can overlap. It is defined by the size and sliding step.

Example:

- Time-based sliding windows: counting system usage stats. Number of accesses in last minute with updates every 10 seconds.

A session is a burst of user activity followed by period of inactivity (timeout). A **session window** collects the activity belonging to the same session. As opposed to tumbling or sliding windows, session windows don't have a fixed start or duration, their scope is data-driven.

Example: When analysing the web site traffic data, then the activity of one user forms one session. The session is considered closed after some period of inactivity (let's say one hour). When the user starts browsing later, it's considered a new session.



Take into account, that the windows may be keyed or global. The keyed window contains just the records that belong to that window and have the same key. See keyed or global transformations section.

Dealing with Late Events

Very often a stream record represents an event that happened in the real world and has its own timestamp (called the *event time*), which can be quite different from the moment the record is processed (this is called the *processing time*). This can happen due to the distributed nature of the system — the record could be delayed on its way from source to stream processing system, the link speed from various sources may vary, the originating device may be offline when the event occurred (e.g. IoT sensor or mobile device in flight mode). The difference between processing time and event time is called **event time skew**.

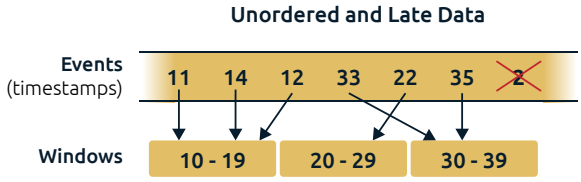
Processing time semantics should be used when:

- The business use case emphasizes low latency; results are computed as soon as possible, without waiting for stragglers.
- The business use case builds on the time when the stream processing engine observed the event, ignoring when it originated; sometimes you simply don't trust the timestamp in the event coming from a 3rd party.

Event time semantics should be used when the timestamp of the event origin matters for the correctness of the computation. When your use case is event time sensitive, it requires more of your attention. If event time is ignored, records can be assigned to improper windows resulting in incorrect computation results.

The system has to be instructed where in the record is the information about event time. Also, it has to be decided how long to wait for late events. If you tolerate long delays, it is more probable that you've captured all the events. Short waiting on the other hand gives you faster responses (lower latency) and less resource consumption — there is less data to be buffered.

In a distributed system, there is no "upper limit" on event time skew. Imagine an extreme scenario, when there is a game score from a mobile game. The player is on a plane, so the device is in flight mode and the record cannot be send to be processed. What if the mobile device never comes online again? The event just happened, so in theory the respective window has to stay open forever for the computation to be correct.



Stream processing frameworks provide heuristic algorithms to help you assess window completeness, sometimes called **watermarks**.

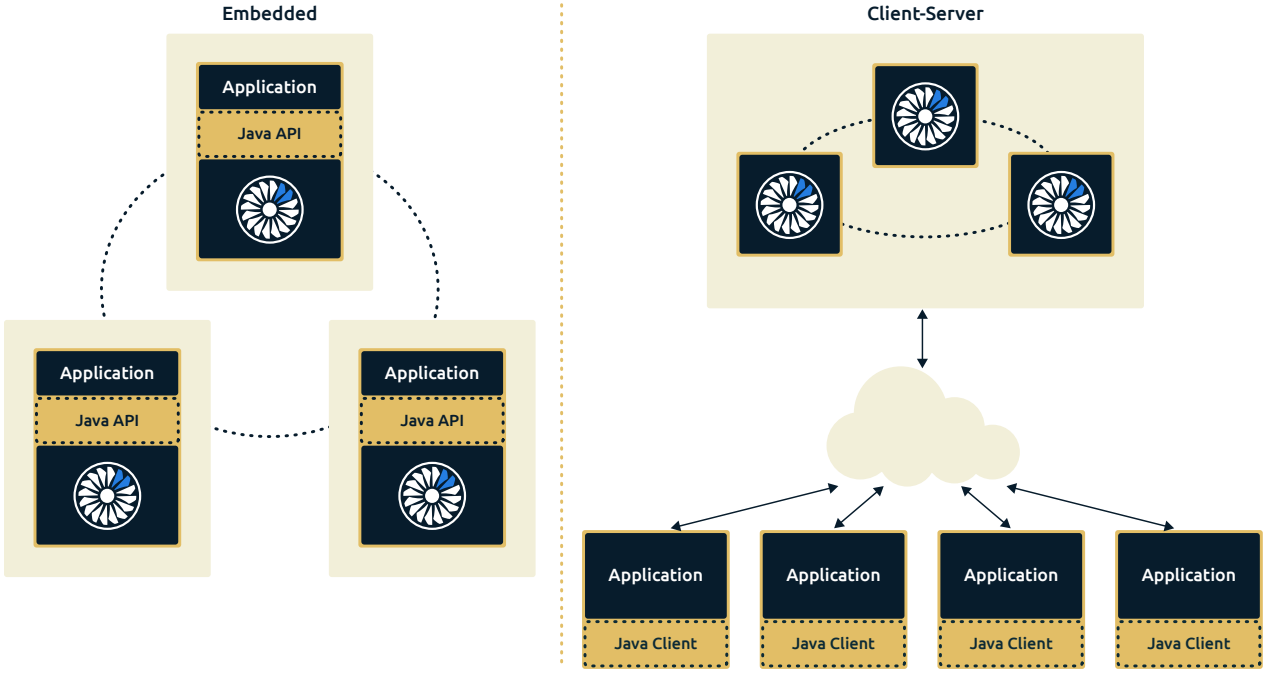
When your system handles event time sensitive data, make sure that the underlying stream processing platform supports event time-based processing.

Running Jobs

The definition of the data processing pipeline (composed of sources, sinks and transformations wired together) is the equivalent of a program, but the environment in which you can execute it is not simply your local machine. It describes a program that will run distributed in the stream processing cluster. Therefore, you must perform the additional step of getting a handle to the stream processing engine and submitting your program to it.

Once you have submitted the pipeline, the SPE allocates the cluster's resources (CPU, memory, disk) for it and runs it. Each vertex in the DAG model becomes one or more tasks executing in parallel, and each edge (i.e., arrow) becomes a data connection between tasks: either a concurrent queue (if both tasks are on the same machine) or a network connection. The tasks implementing the data source start pulling the data and sending it into the pipeline. The SPE coordinates the execution of the tasks, monitors the data connections for congestion and applies backpressure as needed. It also monitors the whole cluster for topology changes (a member leaving or joining the cluster) and compensates for them by remapping the resources devoted to your processing job, so that it can go on unaffected by these changes. If the job's data source is unbounded, it will keep on running until the user explicitly stops it (or a failure occurs within the processing logic).

Stream processing engines usually provide a Client (a programming API or a command line tool) which is used for submitting the Job and its resources to a cluster. The library approach of Hazelcast Jet allows you to follow the "embedded member" mode, where the JVM containing application code participates in the Hazelcast Jet cluster directly and can be used to submit the job.



When to use Client-Server:

- Cluster shared for multiple jobs
- Isolating client application from the cluster

When to use Embedded:

- Simplicity — it's simple, no separate moving parts to manage
- OEM — SPE is embedded in your application
- Microservices

Fault Tolerance

A major challenge of infinite stream processing is maintaining state in the face of inevitable system failures. The data that was already ingested, but not yet emitted as final results, is at risk of disappearing forever if the processing system fails. To avoid loss, the original input items can be stored until the results of their processing have been emitted. Upon resuming after failure these items can then be replayed to the system. The internal state of the computation engine can also be persisted. However, in each case there is an even greater challenge if there must be a strict guarantee of correctness. That means that each individual item must be accounted for and it must be known at each processing stage whether that particular item was processed or not. This is called the *exactly* once processing guarantee. A more relaxed variant is called *at least once* and in this case the system may end up replaying an item that was already processed.

Exactly-once guarantee is achievable only at a cost in terms of latency, throughput, and storage overheads. If you can come up with an *idempotent* function that processes your data, the cheaper at-least-once guarantee will be enough because processing the same event twice with such a function has the same effect as processing it just once.

Current stream processing frameworks do support the exactly-once processing. However, assess whether your use case really cannot tolerate some duplicities in the case of fault, as exactly-once isn't free in the sense of complexity, latency, throughput and resources.

Example: in a system processing access logs and detecting fraud patterns on top of millions of events per second, minor duplicities could be tolerated. Duplicities just lead to possible false positives in exchange for a performance. On the other hand — in a billing system, consistency and exactly-once is an absolute must have.

Sources and Sinks

Stream processing application 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 as they limit what you can do within your Stream processing application.

Is it Unbounded?

The first decision when building a computation job is to decide whether it will deal with bounded (finite) or unbounded (infinite) data.

Bounded data is handled in batch jobs and there are less concerns to deal with as data boundaries are within the dataset itself. You don't have to worry about windowing, late events or event time skew. Examples of bounded, finite resources are plain files, Hadoop Distributed File System (HDFS) or iterating through database query results.

Unbounded data streams allow for continuous processing, however you have to use windowing for operations that cannot effectively work on infinite input (such as sum, avg or sort). In the unbounded category the most popular choice is Kafka. Some databases can be turned into unbounded data sources by exposing it's journal, the stream of all data changes, as an API for 3rd parties.

```
Pipeline p = Pipeline.create();
p.drawFrom(Sources.<Integer, Integer>mapJournal(MAP_NAME, START_FROM_OLDEST));
```

Hazelcast Jet reading the journal of an IMap – the distributed map provided by Hazelcast In-Memory Data Grid (unbounded data source)

Is it Replayable?

The data source is replayable if you can easily replay the data. This is generally necessary for fault tolerance: if anything goes wrong during the computation, the data can be replayed and the computation can be restarted from the beginning.

Bounded data sources are mostly replayable (plain file, HDFS file). The replayability of infinite data sources is limited by the disk space necessary to store the whole data stream — Apache Kafka source is replayable with this limitation. On the other hand, some sources can be read only once (TCP socket source, JMS Queue source).

It would be quite impractical if you could only replay a 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 (offset) you choose. Both Kafka and the Hazelcast Event Journal support this.

Is it Distributed?

A distributed computation engine prefers to work with distributed data resources to maximize performance. If the resource is not distributed, all SPE cluster members will have to contend for access to a single data source endpoint. Kafka, HDFS and Hazelcast IMap are all distributed. On the other hand, a file is not; it is stored on a single machine.

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 the Stream Processing Application will consume and process it. For example, if your source is HDFS, you should align the topologies of the Hadoop and SPE clusters so that each machine that hosts an HDFS member also hosts a node of the SPE cluster. Hazelcast Jet will automatically figure this out and arrange for each member to consume only the slice of data stored locally.

Hazelcast Jet makes use of data locality when reading from co-located Hazelcast IMap or HDFS.

Overview of Stream Processing Platforms

This list provides an overview of the major distributed stream processing platforms:

- Apache Flink: https://flink.apache.org/, open-source, unified platform for batch and stream processing.
- Apache Storm: http://storm.apache.org/, first streaming platform (originated in 2010), thus having a big mind share and install base. It's limited to at-least once and doesn't support stateful stream processing. Project Trident adds another layer for exactly-once using micro-batches. Another project based on Storm is Twitter Heron.
- Google Cloud DataFlow: https://cloud.google.com/dataflow/, a managed service in a Google cloud. It introduced Apache Beam (https://beam.apache.org/) — the unified programming model and SDK for batch and stream processing.
- Hazelcast Jet: https://jet.hazelcast.org/, open source, lightweight and embeddable platform for batch and stream processing, contains distributed in-memory data structures to store operational data and publish results.
- Kafka Streams: https://kafka.apache.org/documentation/streams/, open source SPE integrated into Apache Kafka ecosystem. It's optimized for processing Kafka topics.
- Spark Streaming: https://spark.apache.org/streaming/, open source, utilizes Spark batch processing platform by dividing continuous stream to sequence of discrete micro-batches. Big ecosystem around Spark and thus a big mind share.

Conclusion

This refcard guided you through the key aspects of stream processing. It covered the building blocks of streaming application:

- Source and sink connectors to connect your application into the data pipeline.
- Transformations to process and "query" the data stream using filtering, converting, grouping, aggregating and joining it.
- Windows to select finite sub-streams from generally infinite data streams.

We also covered how to run a streaming application in a stream processing engine, listing the most popular distributed SPE.

350 Cambridge Ave, Suite 100, Palo Alto, CA 94306 USA
Email: sales@hazelcast.com Phone: +1 (650) 521-5453
Visit us at www.hazelcast.com