TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

DSP Frameworks

Corso di Sistemi e Architetture per Big Data
A.A. 2023/24
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

DSP frameworks we consider

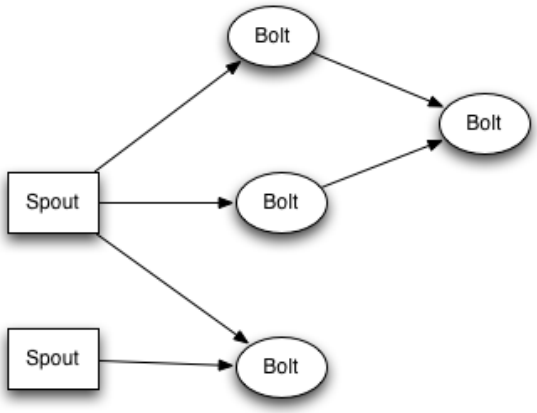
- Apache Storm
- Apache Flink (with hands-on lesson)
- Apache Spark Streaming (with hands-on lesson)
- Kafka Streaming (hands-on lesson)
- Cloud-based frameworks
  - Google Cloud Dataflow
  - Amazon Kinesis

Apache Storm

- Open-source, real-time, scalable streaming system storm.apache.org
- Provides an abstraction layer to execute DSP applications
- Many use cases, including real-time analytics, online ML, continuous computation, distributed RPC, ETL
- Fast: 1M tuples processed per second per node
- Easy to integrate with different sources (e.g., messaging systems)
- Initially developed by Twitter
- Current version: 2.6.2

Storm: topology

- Main Storm's concept: topology
  - Where the application logic is packaged into
  - Long-running
  - DAG of spouts (sources of streams) and bolts (operators that do processing and data sinks)
  - Top-level abstraction submitted to Storm for execution

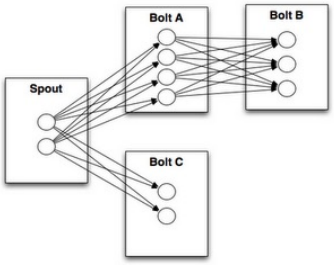


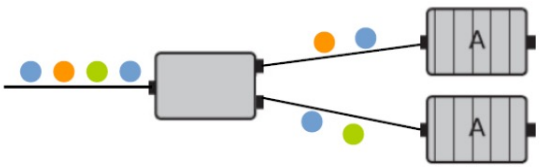
Storm: streams and tuples

- Storm uses streams and tuples as its data model
  - Stream: core abstraction in Storm
    - Unbounded sequence of tuples that is processed and created in parallel in a distributed fashion
    - Streams are defined with a schema that names the fields in the stream's tuples
  - Tuple: named list of values
    - A field in a tuple can be an object of any type
    - Storm supports all primitive types, strings, and byte arrays as tuple field values
    - To use a custom data type, you need to define the corresponding serializer

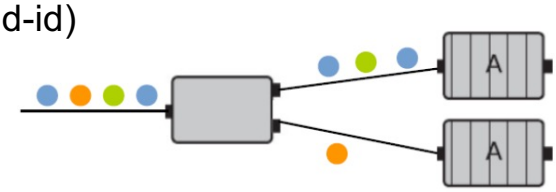
Storm: stream grouping

- Stream grouping defines how to send tuples between two adjacent nodes in the topology
  - Remember of data parallelism: spouts and bolts execute in parallel (multiple threads of execution)
- Shuffle grouping
  - Tuples are randomly partitioned



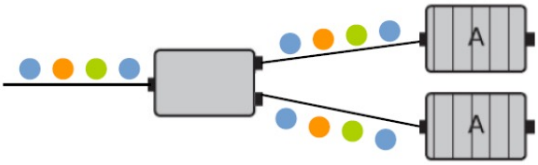


- Field grouping
  - Stream is partitioned by the fields specified in the grouping (e.g., used-id)



Storm: stream grouping

- All grouping (i.e., broadcast)
  - Stream is replicated across all the bolt's replicas (use with care)



- Global grouping
  - Stream goes to a single one of the bolt's replicas (specifically, to the replica with the lowest id)
- Direct grouping
  - The producer of the tuple decides which replica of the consumer will receive this tuple

Storm: a simple topology

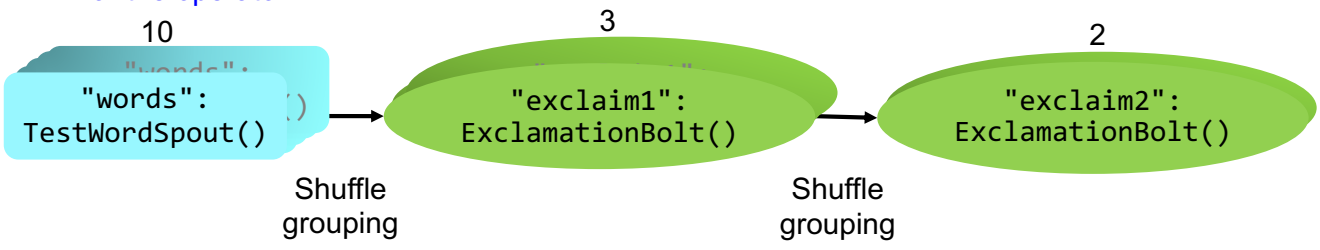
- First example: exclamation
  - Spout emits words, each bolt appends "!!!" to its input

github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/ExclamationTopology.java

setSpout and setBolt methods take as input:

- user-specified id
- object containing the processing logic of the operator
- amount of parallelism for the operator

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
        .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
        .shuffleGrouping("exclaim1");
```



Storm: another topology

- Example: WordCount

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("sentences", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8)
        .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 12)
        .fieldsGrouping("split", new Fields("word"));
```

See github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/WordCountTopology.java

- Bolts can be defined in any language
  - Bolts written in another language are executed as subprocesses, and Storm communicates with them using JSON messages over stdin/stdout
  - Communication protocol for Python available in an adapter library streamparse.readthedocs.io

Storm: windowing

- Storm supports sliding and tumbling windows
- Windows can be based on time duration or event count
  - Count-based windows
    - Based on tuples count (no relation to clock time)
  - Time-based windows
    - Based on time duration
- Bolt that needs windowing support needs to implement `IWindowedBolt` interface

```
public interface IWindowedBolt extends IComponent {
    void prepare(Map stormConf, TopologyContext context, OutputCollector collector);
    /**
     * Process tuples falling within the window and optionally emit
     * new tuples based on the tuples in the input window.
     */
    void execute(TupleWindow inputWindow);
    void cleanup();
}
```

`execute` is invoked every time the window activates

Storm: windowing

- Different window configurations, including
  - Sliding windows

```
withWindow(Count windowLength, Count slidingInterval)
Tuple count based sliding window that slides after `slidingInterval` number of tuples.

withWindow(Duration windowLength, Duration slidingInterval)
Time duration based sliding window that slides after `slidingInterval` time duration.
```

  - Tumbling windows

```
withTumblingWindow(BaseWindowedBolt.Count count)
Count based tumbling window that tumbles after the specified count of tuples.

withTumblingWindow(BaseWindowedBolt.Duration duration)
Time duration based tumbling window that tumbles after the specified time duration.
```

  See github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/SlidingWindowTopology.java
- By default, tuples in the window are stored in memory until they are processed and expired
  - ✗ Windows need to fit entirely in memory
- Storm also provides stateful windowing support

Storm: Stream API

- Alternative interface to Storm: provides a typed API for expressing streaming computations and supports functional style operations
  - Similarly to Spark and Flink, but still experimental

  storm.apache.org/releases/2.6.2/Stream-API.html
- Stream APIs: Stream and PairStream (key-value pair streams)
  - Support a wide range of operations, including
    - Transformations, which produce another stream from current one (e.g., `filter`, `map`, `flatMap`)
    - Windowing
    - Aggregations (e.g., `reduce`, `aggregate`, `reduceByKey`, `countByKey`)
    - Joins
    - Output, which produce a result (e.g., `print`, `forEach`)

Storm: Stream API example

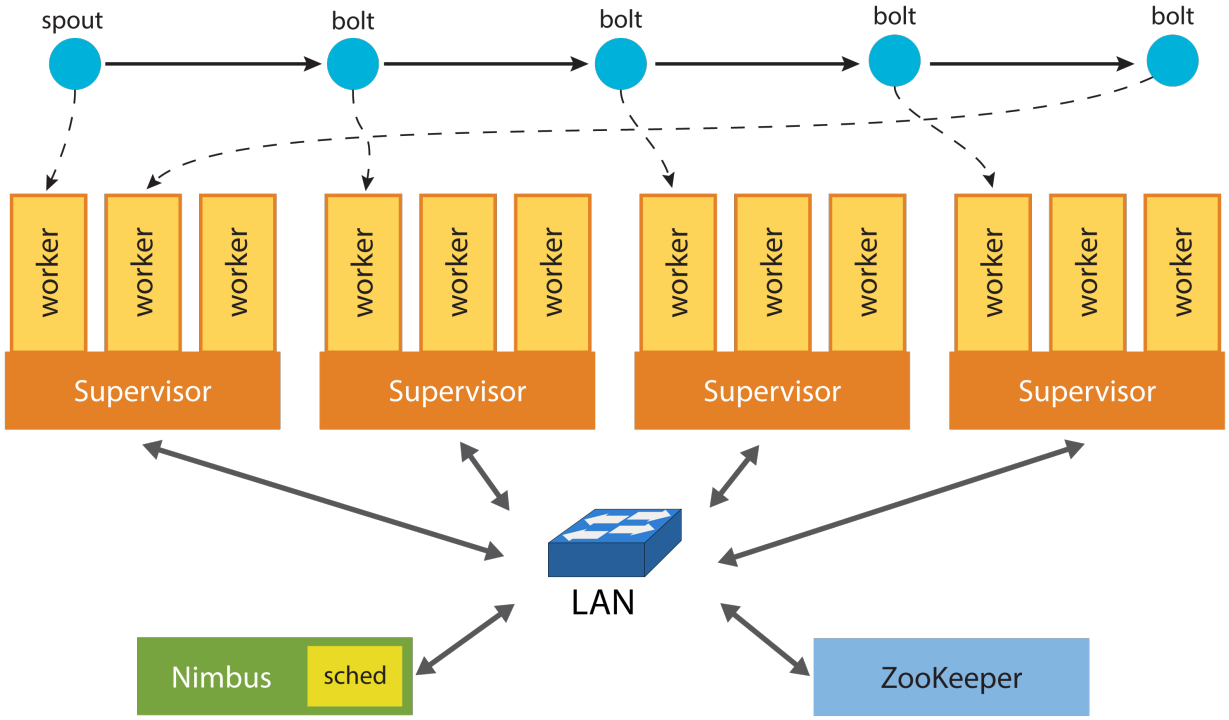
- The usual WordCount using Stream API

```
StreamBuilder builder = new StreamBuilder();

builder
   // A stream of random sentences with two partitions
   .newStream(new RandomSentenceSpout(), new ValueMapper<String>(0), 2)
   // a two seconds tumbling window
   .window(TumblingWindows.of(Duration.seconds(2)))
   // split the sentences to words
   .flatMap(s -> Arrays.asList(s.split(" ")))
   // create a stream of (word, 1) pairs
   .mapToPair(w -> Pair.of(w, 1))
   // compute the word counts in the last two second window
   .countByKey()
   // print the results to stdout
   .print();
```

See github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/streams/WindowedWordCount.java

Storm: architecture

- Master-worker architecture

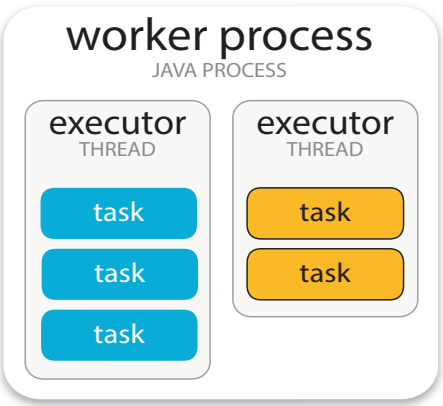


Storm components: master and Zookeeper

- Nimbus
  - Master node
  - Users submit topologies to it
  - Responsible for distributing and coordinating the topology execution
- Zookeeper
  - Nimbus uses a combination of local disk(s) and Zookeeper to store info about application topology

Storm components: worker node

- Worker node: computing resource, a kind of container for one or more worker processes
- Worker process: Java process running one or more executors
- Executor: smallest schedulable entity
  - Execute one or more tasks related to same operator
- Task: operator instance
  - Actual work for bolt or spout is done by task



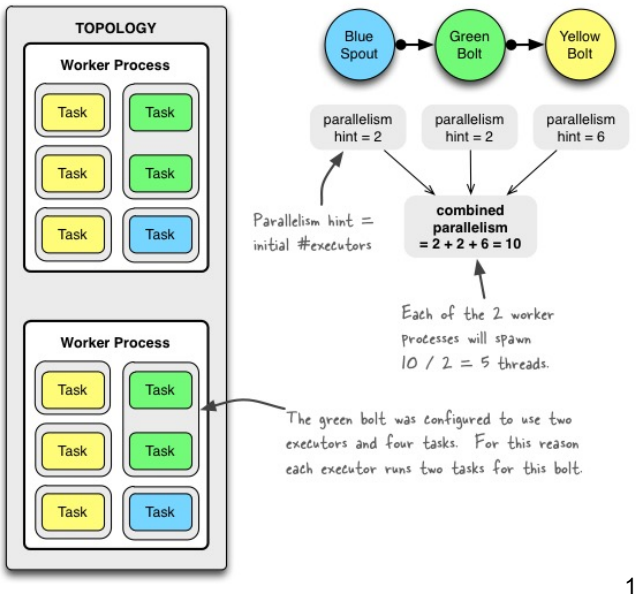
Storm components: supervisor

- Each worker node runs a supervisor
- Each supervisor:
  - Receives assignments from Nimbus (through ZooKeeper) and spawns workers based on the assignment
  - Sends to Nimbus (through ZooKeeper) a periodic heartbeat
  - Advertises the topologies that the worker node is currently running, and any vacancies that are available to run more topologies

Storm: running topology

- Application developer can configure the topology parallelism
  - Number of worker processes
  - Number of executors (threads)
  - Number of tasks
- Parallelism of running topology can be changed *manually* using `rebalance` command

See storm.apache.org/releases/current/Understanding-the-parallelism-of-a-Storm-topology.html



Storm: reliable message processing

- What happens if a bolt fails to process a tuple?
- Storm provides a mechanism by which the originating spout can replay the failed tuple
  - Storm needs to maintain the link between every spout tuple and its tree of tuples so to detect when the tree of tuples has been successfully processed: anchoring
  - And needs to ack (or fail) the spout tuple appropriately
    - If ack is not received within a specified timeout time period, the tuple processing is considered as failed and the tuple is replayed
- Storm provides at-least-once semantics
  - Best effort semantics if acking is disabled

storm.apache.org/releases/2.6.2/Guaranteeing-message-processing.html

Storm: application monitoring

- Storm has a built-in monitoring and metrics system
  - Built-in and user-defined metrics
- Built-in metrics include:
  - Capacity
    - # of messages executed * average execute latency / time window
  - Latency
    - For spouts: completeLatency (total latency for processing the message)
      - Ignore value if acking is disabled
    - For bolts: executeLatency (avg time the bolt spends to run the execute method) and processLatency (avg time from starting execute to ack)
  - JVM memory usage and garbage collection
- Metrics can be queried via Storm's UI REST API or reported to a registered consumer (e.g., Graphite)

See storm.apache.org/releases/current/STORM-UI-REST-API.html


```
"bolts": [
{
    "executors": 12,
    "emitted": 184580,
    "transferred": 0,
    "acked": 184640,
    "executeLatency": "0.048",
    "tasks": 12,
    "executed": 184620,
    "processLatency": "0.043",
    "boltId": "count",
    "lastError": "",
    "errorLapsedSecs": null,
    "capacity": "0.003",
    "failed": 0
},
```

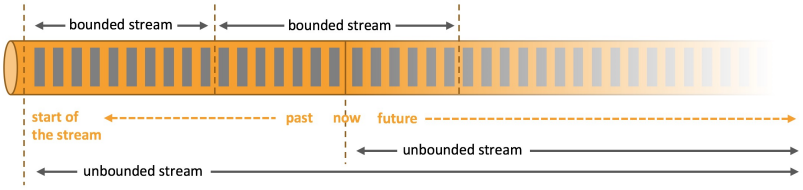
Layers on top of Storm

- Trident
  - High-level abstraction on top of Storm to provide exactly-once processing
- SQL
  - To run SQL queries over streaming data

storm.apache.org/releases/2.4.0/storm-sql.html

Unbounded vs. bounded streams

- Data can be processed as bounded or unbounded streams



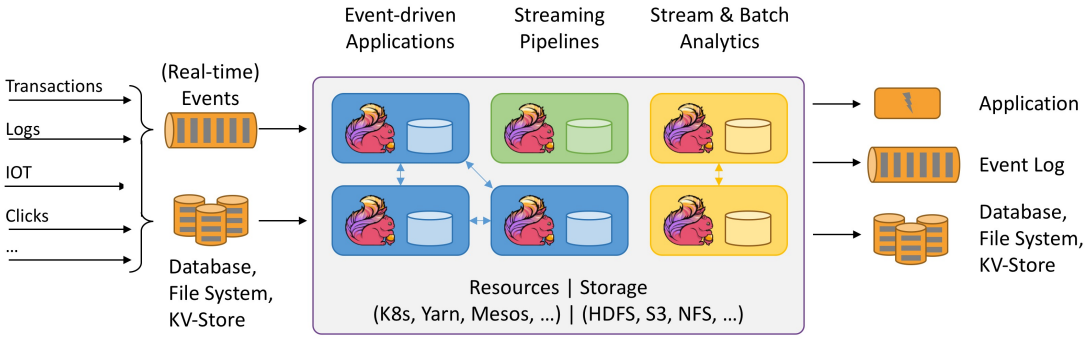
- Unbounded streams
  - Have a start but no defined end
  - Provide data as it is generated
  - Must be continuously processed
  - Not possible to wait for all input data to arrive
  - Processing unbounded data often requires that events are ingested in a specific order
- Bounded streams
  - Have a defined start and end
  - Can be processed by ingesting all data before performing any computations
  - Ordered ingestion is not required because bounded data can always be sorted

Batch processing vs. stream processing

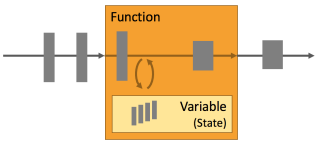
- Batched/stateless: scheduled in batches
  - Short-lived tasks (Hadoop, Spark)
  - Distributed streaming over batches (Spark Streaming)
- Dataflow/stateful: continuously processed, typically scheduled once (Storm, Flink)
  - Long-lived task execution
  - State is kept inside tasks

Apache Flink

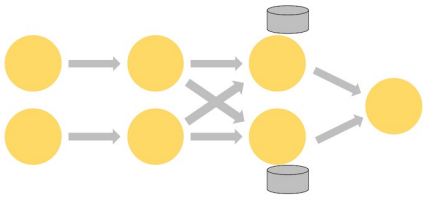
- Distributed processing system for stateful computation over bounded and bounded data streams flink.apache.org
  - One common runtime for data streaming and batch processing
- Integrated with many other projects in Big data open-source ecosystem
- Originated from Stratosphere project by TU Berlin, Humboldt Univ. and Hasso Plattner Institute
- Current version: 1.19



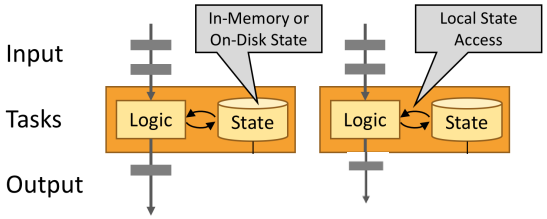
Flink: Stateful computation



- Flink's operations can be stateful
  - E.g., counting events per minute to display on dashboard, computing features for fraud detection model
- State is partitioned: the set of parallel instances of a stateful operator is a sharded key-value store

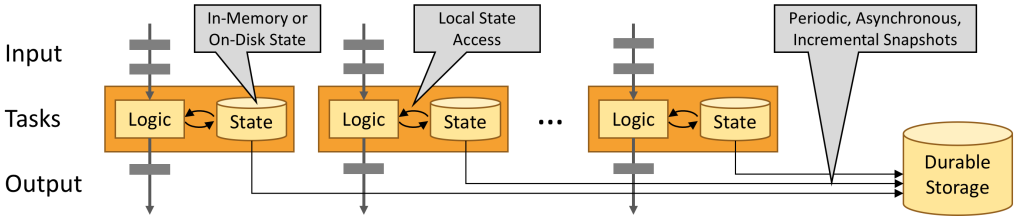


- State is optimized for local access
  - Stored in memory or in access-efficient data structures on disk
  - Goals: high throughput and low latency



Flink: fault tolerance

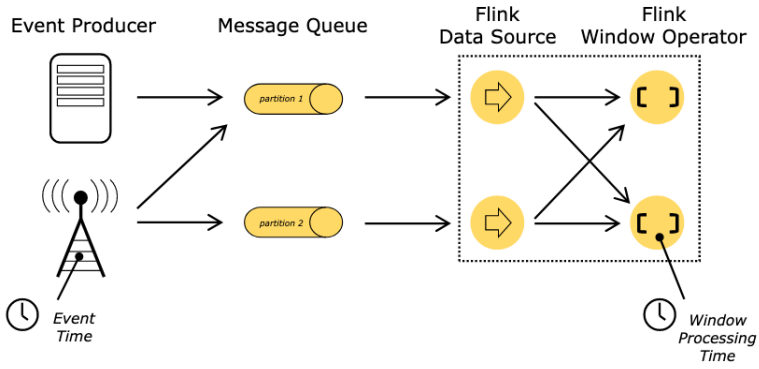
- Flink can guarantee exactly-once state consistency in case of failures by *periodically* and *asynchronously* checkpointing local state to durable storage (state snapshot)
  - State of operators can be restored from checkpoint to an earlier point in time and records are reset to the point of the state snapshot



Flink docs: nightlies.apache.org/flink/flink-docs-stable/

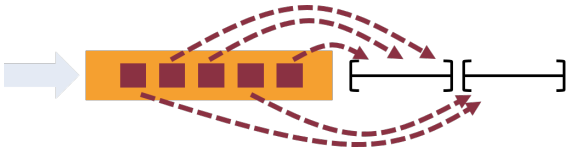
DSP and time

- Different notions of time in a DSP application:
  - Processing time: time at which an event is observed in the system (wall-clock time of the machine executing the operator)
  - Event time: time at which an event actually occurred on its producing device
    - Typically embedded within the event before it enters Flink and represented by a *timestamp*



Flink: time

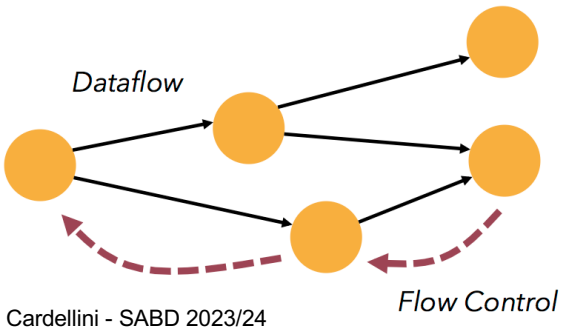
- Flink supports both processing time and event time
- Event time makes it easy to compute over streams where events arrive *out-of-order* and are late
- To support event time, the DSP needs a way to measure the progress of event time; how to measure it?
- Flink uses watermarks

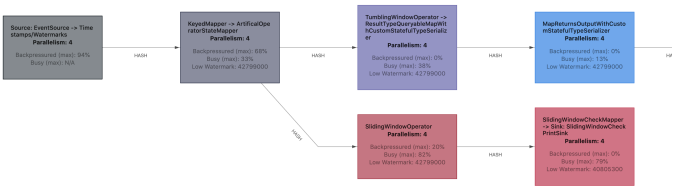


nightlies.apache.org/flink/flink-docs-release-1.19/docs/concepts/time/

Flink: back pressure

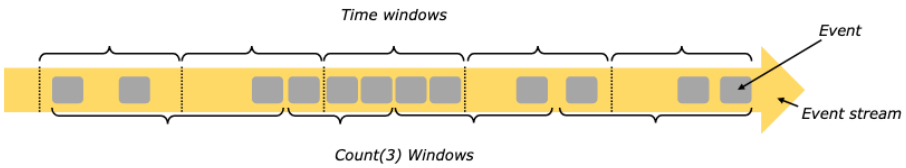
- Continuous streaming with back pressure
  - Flink runtime provides flow control: slow downstream operators backpressure faster upstream operators
  - Flink UI allows us to monitor back pressure behavior of running applications
    - Back pressure warning (e.g., `High`) for an upstream operator means it it is producing data faster than the downstream operators can consume





Flink: windows

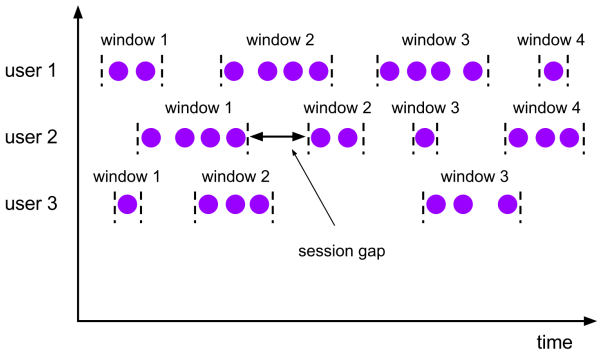
- Highly flexible streaming windows
  - Including user-defined windows



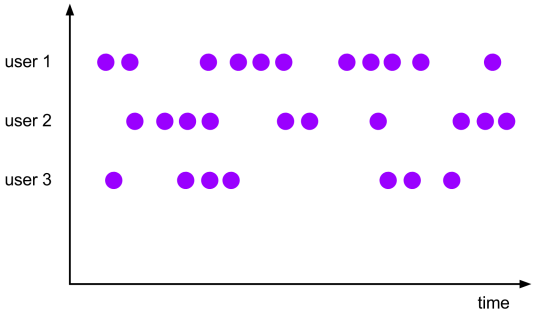
- Supported window types
  - Tumbling: no overlap
  - Sliding: with overlap
  - Session
  - Global

Flink: windows

- Session window
  - To group elements by sessions of activity
  - Differently from tumbling and sliding windows, no overlap and no fixed start and end time
  - Closes when a gap of inactivity occurs

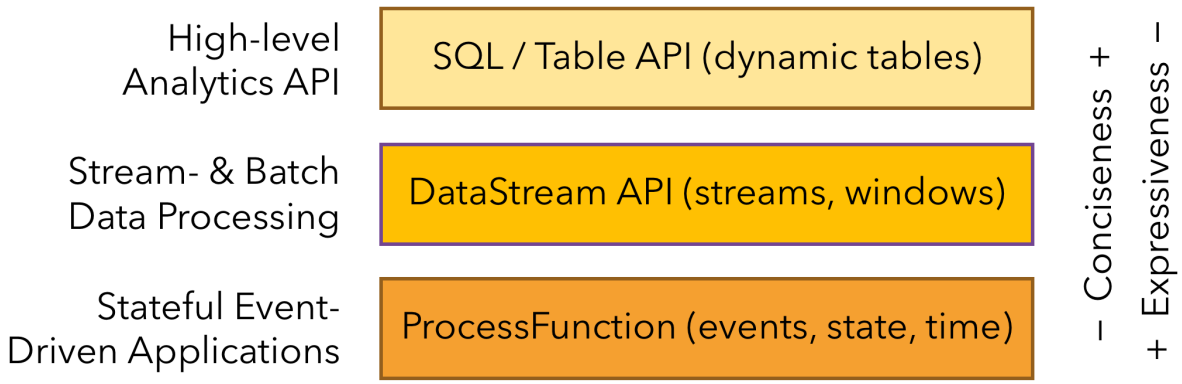


- Global window
  - To assign all elements with the same key to the same single global window
  - Only useful if you also specify a custom trigger



Flink: APIs

- Different levels of abstraction to develop streaming applications
  - SQL & Table API
  - DataStream API
  - ProcessFunctions
- APIs in Java, Scala and Python



Flink: APIs

- ProcessFunction: low-level stream processing operation, giving access to basic building blocks of (acyclic) streaming applications
  - events (stream elements)
  - state (fault-tolerant, consistent, only on keyed stream)
  - timers (event time and processing time, only on keyed stream)

nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/datastream/operators/process_function/

Flink: APIs

- DataStream API: streaming (and batch) applications
  - Supports transformations on data streams (e.g., filter, update state, define windows, aggregate), with user-defined state and flexible windows
  - Provides fine-grained control over state and time
  - Supports different runtime execution modes
    - STREAMING: classic one for unbounded streams
    - BATCH: reminiscent of batch processing frameworks

nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/datastream/overview/

Flink: APIs

- Table API & SQL
  - Relational APIs for unified stream and batch processing

nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/table/overview/

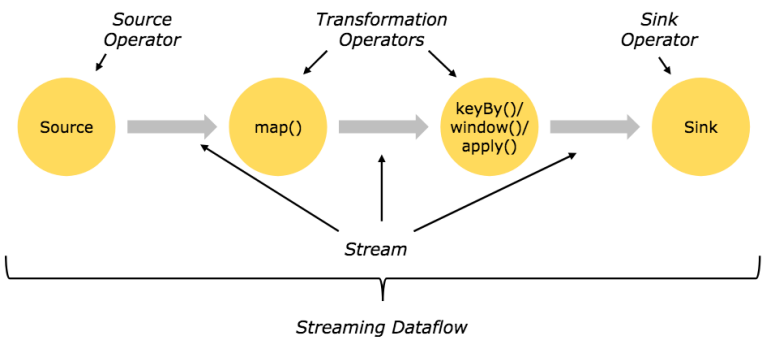
- Python API
  - Python API for Apache Flink: PyFlink DataStream API and PyFlink Table API

nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/python/overview/

- See lab lesson

Flink: programming model

- Applications are composed of streaming dataflows that are transformed by user-defined operators
  - Streams: unbounded, partitioned, immutable sequence of events
- Streaming dataflows form directed graphs (usually DAGs) that start with one or more sources, and end in one or more sinks
  - DAGs basic building blocks: streams and transformation operators



Flink: programming model

- Stream operators
  - Stream transformations that take one or more streams as input, and produce one or more output streams as a result

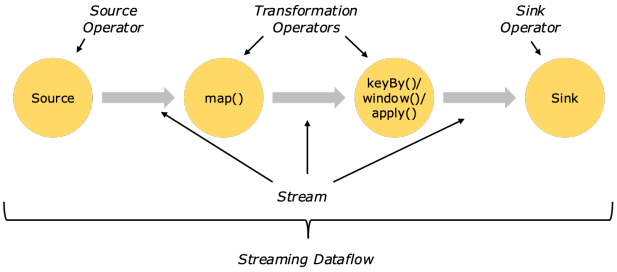
```
DataStream<String> lines = env.addSource(
                            new FlinkKafkaConsumer<>(…));
DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new BucketingSink(path));
```

Source

Transformation

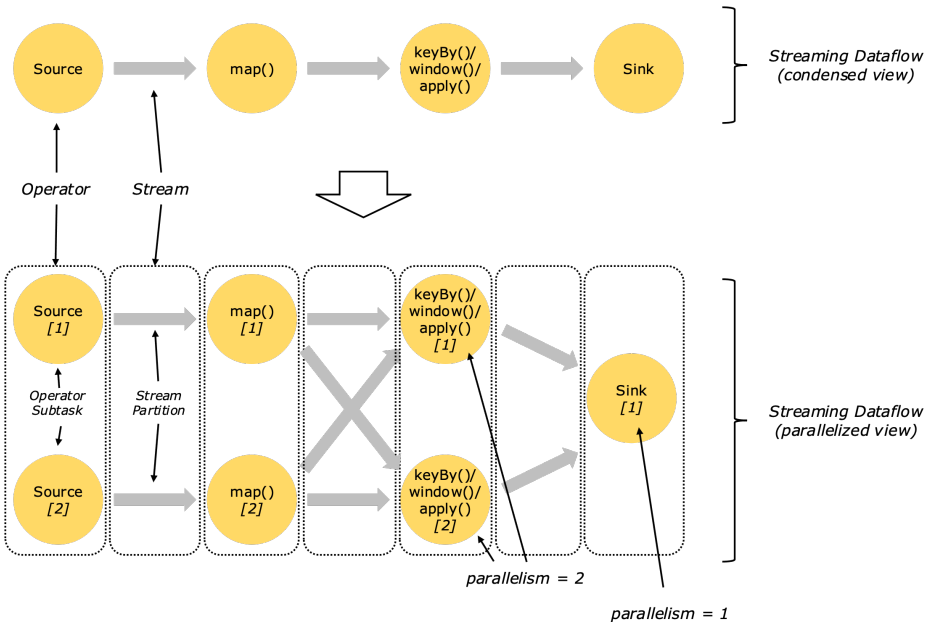
Transformation

Sink



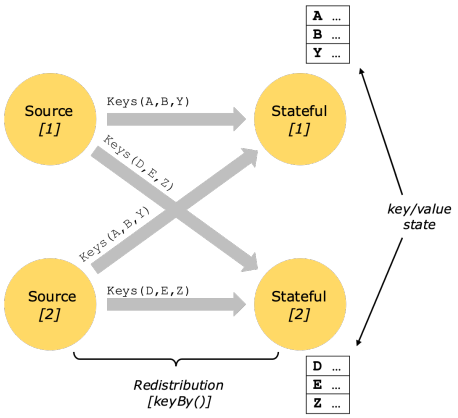
Flink: programming model

- Parallel dataflows: operator parallelism
  - Same solution as in Storm



Flink: programming model

- Stateful operators: require to remember information across multiple events
  - E.g., counting events per minute to display on a dashboard, or computing features for a fraud detection model
  - State is maintained in a sort of embedded key/value store
  - Access to key/value state is only possible on *keyed streams* after `keyBy()`, which re-partitions by hashing the key



Flink: sources and sinks

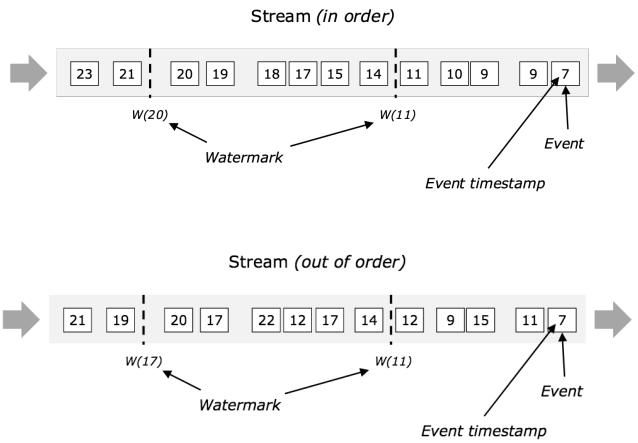
- Sources ingest input data from external systems, while sinks send result data to external systems
- Basic data sources and sinks are built into Flink
  - predefined data sources include reading from files, directories, and sockets, and ingesting data from collections and iterators
  - predefined data sinks support writing to files, to stdout and stderr, and to sockets
- Connectors provide code for interfacing with various third-party systems
  - Apache Kafka, Rabbit MQ, Apache Pulsar, etc.

Flink: control events

- Control events: special events injected in the data stream by operators
- Two types of control events in Flink
  - Watermarks
  - Checkpoint barriers

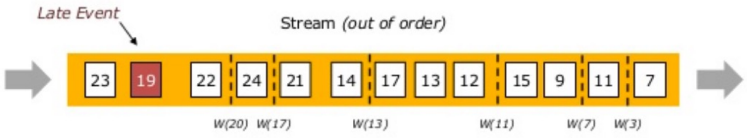
Flink: watermarks

- Watermarks (*W*) mark the progress of event time within a data stream
- Generated at, or directly after, source functions
- Flow as part of data stream and carry a timestamp *t*
  - *W(t)* declares that event time has reached time *t* in that stream, meaning that there should be no more elements with timestamp *t'* <= *t*
  - Crucial for *out-of-order* streams, where events are not ordered by their timestamps



Flink: watermarks

- By default, late elements are dropped when the watermark is past the end of the window
- However, Flink allows to specify a maximum *allowed lateness* for window operator
  - By how much time elements can be late before they are dropped (0 by default)
  - Late elements that arrive after the watermark has passed the end of the window but before it passes the end of the window plus the allowed lateness, are still added to the window

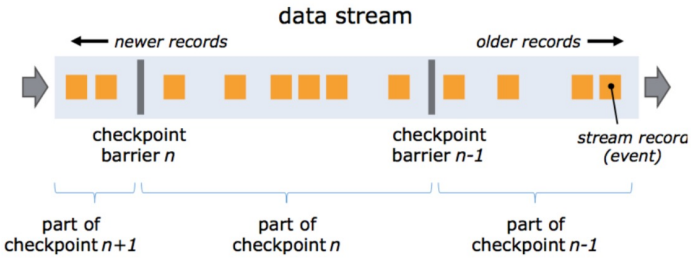


Flink: watermarks

- Flink does not provide ordering guarantees after any form of stream partitioning or broadcasting
  - In such case, dealing with out-of-order tuples is left to the operator implementation

Flink: checkpoint barriers

- To provide fault tolerance special barrier markers (called checkpoint barriers) are periodically injected at streams sources and then pushed downstream up to sinks

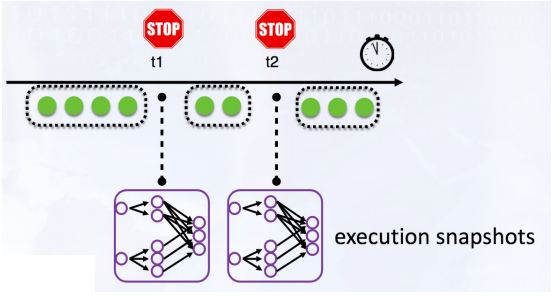


Fault tolerance

- To provide consistent results, DSP systems need to be resilient to failures
- How? By periodically capturing a snapshot of execution graph which can be used later to restart in case of failures (checkpointing)

  Snapshot: global state of execution graph, capturing all necessary information to restart computation from that specific execution state

- Common approach is to rely on periodic global state snapshots, but has drawbacks:
  - Stalls overall computation
  - Eagerly persists all tuples in transit along with states, which results in larger snapshots than required



Flink: snapshot algorithm

- Flink implements a lightweight snapshot algorithm
  - Allows to maintain high throughput while providing strong consistency guarantees
- Snapshot algorithm properties:
  - Draws consistent snapshots of stream flows and operators' state
  - Even in presence of failures, application state reflects every record from data stream exactly once
  - State can be stored at JobManager's heap or HDFS
  - Disabled by default
- Inspired by Chandy-Lamport algorithm for distributed snapshot and tailored to Flink's execution model

nightlies.apache.org/flink/flink-docs-release-1.19/docs/concepts/stateful-stream-processing/

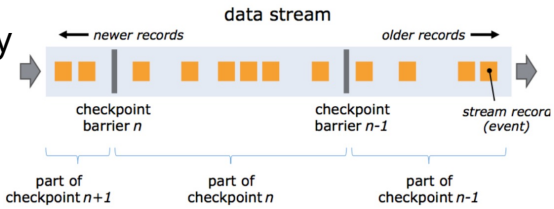
arxiv.org/abs/1506.08603

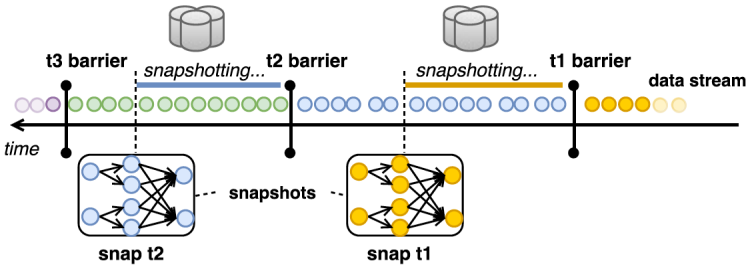
Chandy-Lamport algorithm

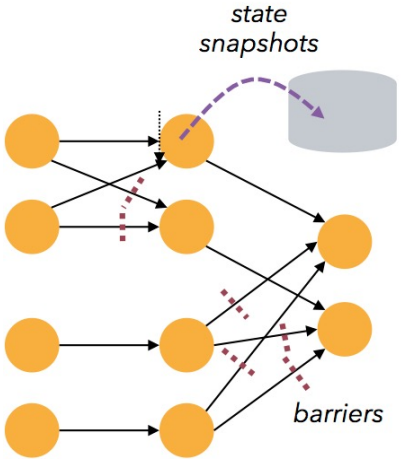
- The observer process (process initiating the snapshot):
  - Saves its own local state
  - Sends a snapshot request message bearing a snapshot token to all other processes
- If a process receives the token *for the first time*:
  - Sends the observer process its own saved state
  - Attaches the snapshot token to all subsequent messages (to help propagate the snapshot token)
- When a process that has *already* received the token receives a message not bearing the token, it will forward that message to the observer process
  - This message was sent before the snapshot "cut off" (as it does not bear a snapshot token) and needs to be included in the snapshot
- The observer builds up a complete snapshot: a saved state for each process and all messages "in the ether" are saved

Flink: snapshot algorithm

- Based on checkpoint barriers injected by checkpoint coordinator
  - When an operator receives a barrier for snapshot *n* from all of its input streams, it emits a barrier for snapshot *n* into all of its outgoing streams. Once a sink operator has received barrier *n* from all of its input streams, it acknowledges snapshot *n* to the checkpoint coordinator (master). After all sinks have acknowledged a snapshot, it is considered completed



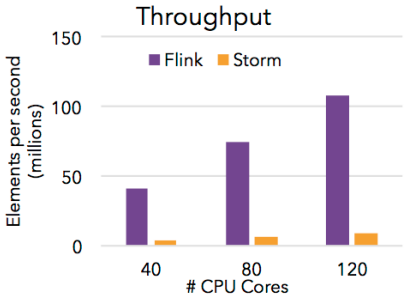


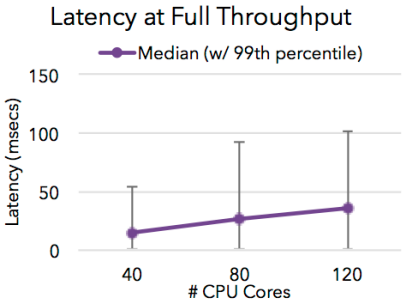


nightlies.apache.org/flink/flink-docs-release-1.19/docs/learn-flink/fault_tolerance

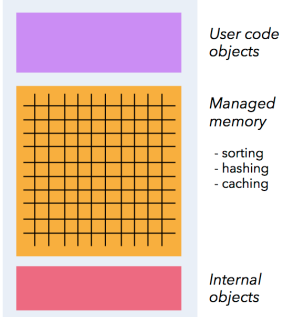
Flink: performance and memory management

- High throughput and low latency



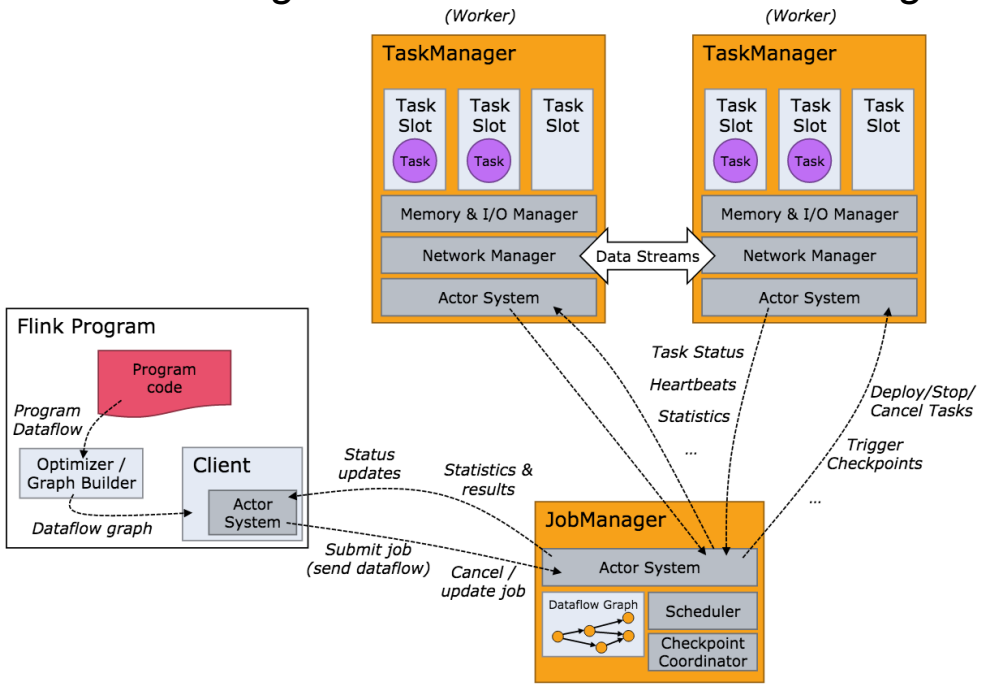


- Memory management
  - Flink implements its own memory management inside JVM



Flink: architecture

- The usual master-worker architecture
  - A *JobManager* and one or more *TaskManagers*



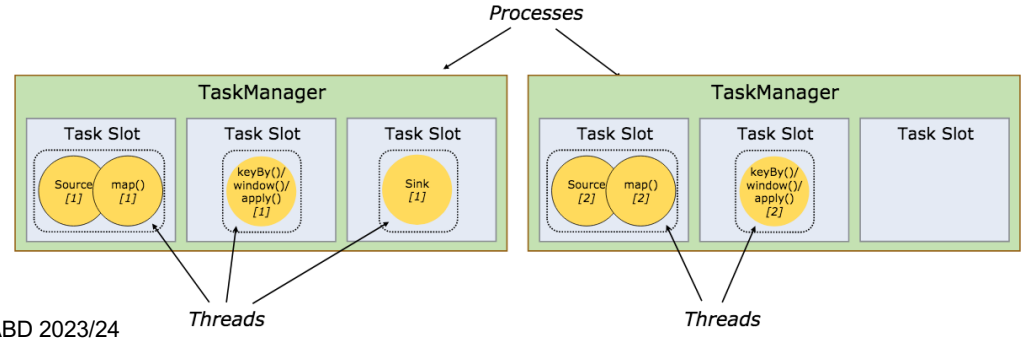
Flink: architecture

- JobManager (master): responsible to coordinate distributed execution of Flink applications
  - Schedules tasks, coordinates checkpoints, coordinates recovery on failures, etc.
- Composed by:
  - **ResourceManager**: responsible for resource de-/allocation and provisioning, manages task slots (unit of resource scheduling in Flink cluster)
  - **Dispatcher**: provides a REST interface to submit Flink applications for execution and starts a new JobMaster for each submitted job; also runs Flink Web UI
  - **JobMaster**: responsible for managing the execution of a single JobGraph

nightlies.apache.org/flink/flink-docs-release-1.19/docs/concepts/flink-architecture/

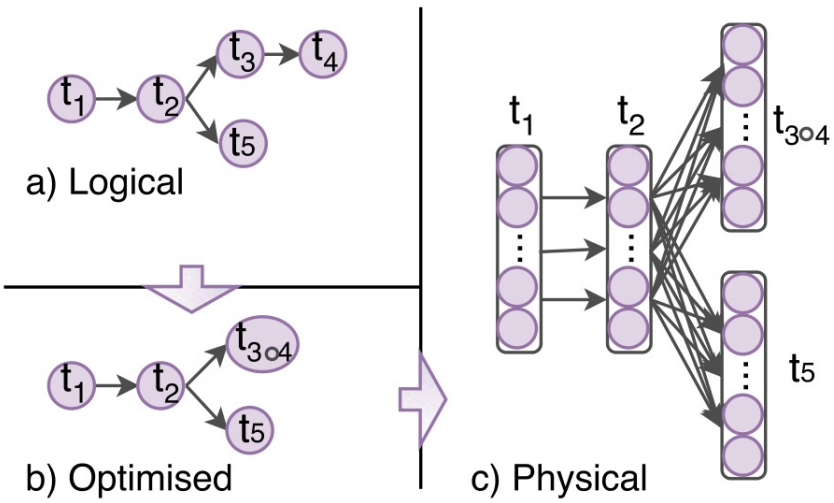
Flink: architecture

- TaskManagers (workers): JVM processes that execute tasks of a dataflow, and buffer and exchange data streams
  - Workers use task slots to control the number of tasks they accept (at least 1)
  - Each task slot represents a fixed subset of worker resources
  - By adjusting the number of task slots, users can define how tasks are isolated from each other
    - Tasks in same JVM share TCP connections, heartbeat messages, data sets and data structures



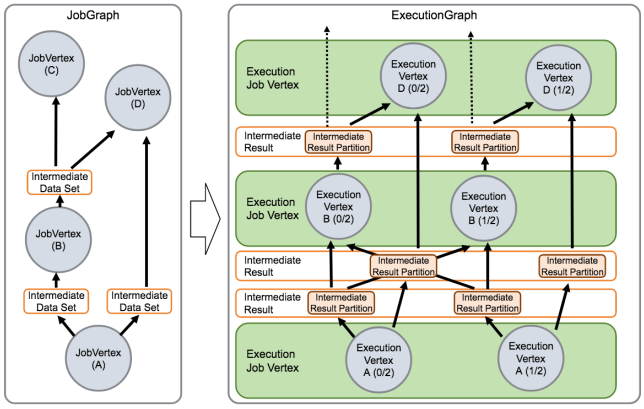
Flink: from logical to physical graph

- Optimizer takes user-specified logical plan and creates an optimized plan



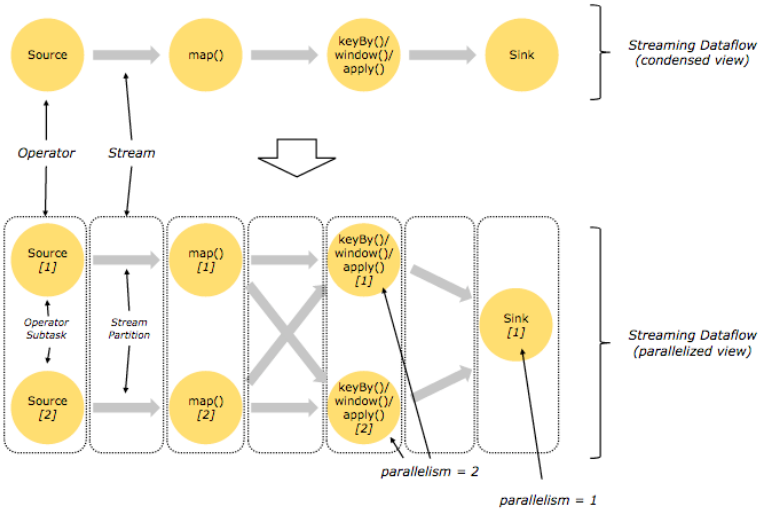
Flink: application execution

- JobManager receives JobGraph (or Logical Graph)
  - Representation of dataflow consisting of operators (JobVertex) and intermediate results (IntermediateDataSet)
  - Each operator has properties, like parallelism and code that it executes
- JobManager transforms JobGraph into ExecutionGraph (or physical graph)
  - Parallel version of JobGraph
  - Graph nodes are tasks and graph edges indicate input/output relationships or partitions of data streams



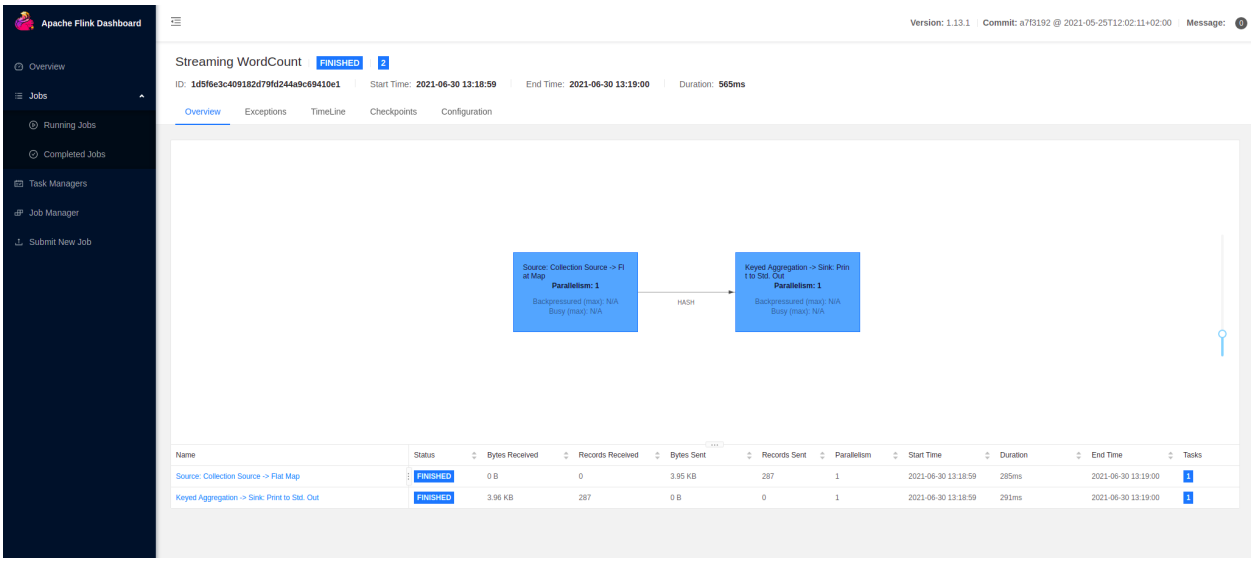
Flink: application execution

- Data parallelism
  - Different operators of same application may have different levels of parallelism
  - Parallelism of individual operator, data source, or data sink can be defined by calling its `setParallelism()` method



Flink: application execution

- Flink provides a Web UI to monitor the status of the cluster and running job



Flink: monitoring

- Built-in monitoring and metrics system
- Allows gathering and exposing metrics to external systems
- Built-in metrics include
  - Throughput
  - Latency: delay between event creation and time at which results based on this event become visible
  - Used JVM heap/non-heap/direct memory
  - CPU
  - Availability
  - Backpressure
  - Checkpointing

Flink: application monitoring

- Throughput
  - In terms of rate of outgoing number of records (per operator/task), e.g.,
    - `numRecordsOutPerSecond`: number of records operator/task sends per second
- Latency

  nightlies.apache.org/flink/flink-docs-release-1.19/docs/ops/metrics/#end-to-end-latency-tracking
  - Flink supports end-to-end latency tracking: special markers (called LatencyMarker) are periodically inserted at all sources in order to obtain a distribution of latency between sources and each downstream operator
    - But does not account for time spent in operator processing (or in window buffers)
    - Assume that all machines clocks are sync
    - Disabled by default (can significantly impact performance, use for debugging): to enable, set `latencyTrackingInterval` > 0

Flink: application monitoring

- Back pressure: to monitor back pressure behavior of running jobs
  - E.g., High warning for a task means that it is producing data faster than downstream operators can consume

nightlies.apache.org/flink/flink-docs-release-1.19/docs/ops/monitoring/back_pressure

- Checkpointing: to monitor the checkpoints of jobs
- Application-specific metrics can be added
  - E.g., counters for number of invalid records
- Metrics can be
  - Queried via Flink's Monitoring REST-ful API, that accepts HTTP requests and responds with JSON data

nightlies.apache.org/flink/flink-docs-release-1.19/docs/ops/rest_api/

  - Visualized in Flink dashboard (use Metrics tab)
  - Sent to external systems (e.g., Graphite and InfluxDB)

nightlies.apache.org/flink/flink-docs-release-1.19/docs/ops/metrics/

Flink: deployment

- Designed to run on large-scale clusters with thousands of nodes
- Can be run in a fully distributed fashion on a *static* (possibly heterogeneous) standalone cluster
- For a *dynamically shared* cluster, can be deployed on YARN, Mesos or Kubernetes
- Docker images for Apache Flink available on Docker Hub
  - Docker official image: https://hub.docker.com/_/flink
  - By Flink developers: https://hub.docker.com/r/apache/flink

  See nightlies.apache.org/flink/flink-docs-master/docs/deployment/resource-providers/standalone/docker/

Delivery guarantees

- Some frameworks provide at-least-once delivery guarantees (e.g., Storm)
  - Each tuple is guaranteed to be processed, but it may get processed more than once
    - Nothing is lost, but might be duplicated
  - How? Deliveries are retried until they are acked (recall RR1 mechanism)
- To avoid data loss, also the source should be *replayable*
  - If the DSP system fails before the tuple could be persisted or processed, the source must provide the same tuple again
  - E.g., Kafka is a replayable source
- For stateful non-idempotent operators (e.g., counting), at-least-once delivery guarantees can give incorrect results

Towards strict delivery guarantees

- Flink, Storm plus Trident, Spark Structured Streaming, and Google's MillWheel offer stronger delivery guarantees (i.e., exactly-once) from the user's perspective
  - In strict sense: each tuple is guaranteed to be processed once and only once
  - What is needed?
    - Replayable sources
    - Reliable operators, where each operator keeps track of its progress and persists its state into fault-tolerant storage
      - Write-ahead log to provide atomicity and durability
      - Checkpointing
    - Idempotent (or transactional) sinks, so that every tuple affects the sinks exactly once

Towards strict delivery guarantees

- Exactly-once in MillWheel works as follows:
  - Upon receipt of an input tuple for a computation
    - The tuple is checked against de-duplication data from previous deliveries; duplicates are discarded
    - User code is run for the tuple, possibly resulting in pending changes to timers, state, and productions
    - Pending changes are committed to backing store
    - Senders are acked
    - Pending downstream productions are sent
  - As an optimization, these operations may be coalesced into a single checkpoint for multiple tuples

Akidau et al., MillWheel: Fault-Tolerant Stream Processing at Internet Scale, VLDB 2013

Comparing DSP frameworks

- Let's compare open source DSP frameworks according to some features

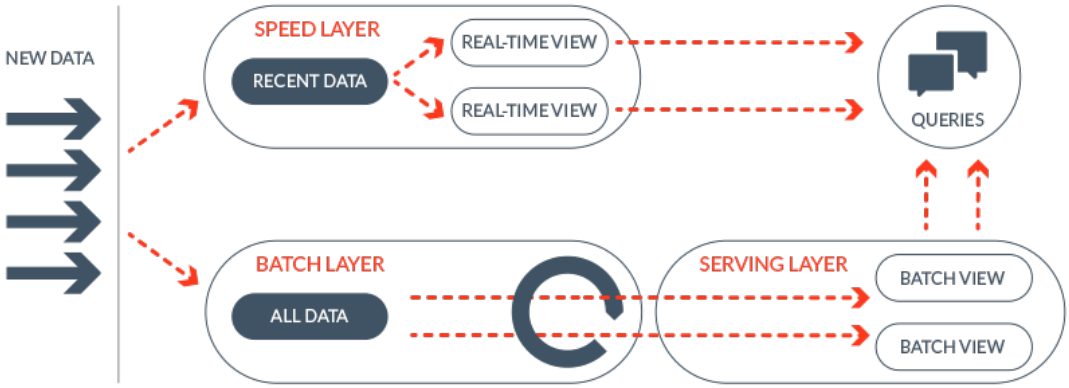
	API	*Windows*	*Delivery semantics*	*Fault tol.*	*State mgmt.*	*Flow control*	*Operator elasticity*
Storm	Low-level High-level SQL No batch	Yes	At-least-once Exactly-once with Trident	Acking Checkpoint. (similar to Flink)	Limited Yes with Trident	Back pressure	No
Flink	High-level SQL Also batch	Yes, also used-def.	At-least-once Exactly-once	Checkpoint.	Yes	Back pressure	No

A recent need

- A common need for many companies
  - Run both batch and stream processing
- Alternative solutions
  1. Lambda architecture
  2. Unified frameworks
  3. Unified programming model

Lambda architecture

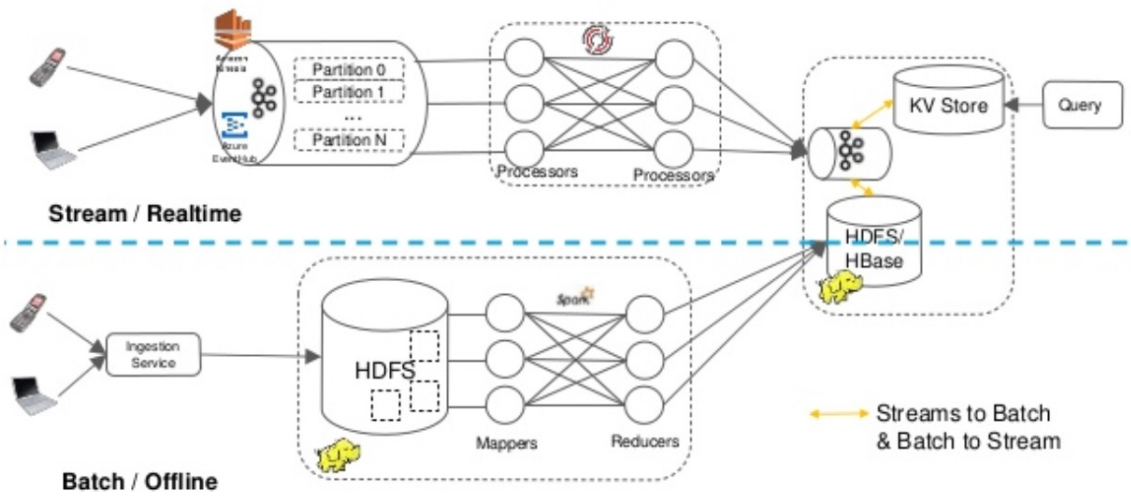
- Data-processing design pattern to integrate batch and real-time stream processing
- Composed of 3 layers
  - Batch and speed (stream) layers: batch framework to process the entire dataset and, in parallel, streaming framework used to process real-time events
  - Results from the two layers are then merged



Source: https://voltdb.com/products/alternatives/lambda-architecture

Lambda architecture: example

- LinkedIn's lambda architecture
  - Before Samza development



Lambda architecture: pros and cons

- Pros:
  - Flexibility in frameworks' choice
- Cons:
  - Implementing and maintaining two separate frameworks for batch and stream processing can be hard and error-prone
  - Overhead of developing and managing multiple source codes
    - The logic in each fork evolves over time, and keeping them in sync involves duplicated and complex manual effort, often with different languages

Unified frameworks

- Use a unified (Lambda-less) design for processing both real-time as well as batch data using the same data structure
- Spark and Flink follow this trend

Unified programming model: Apache Beam

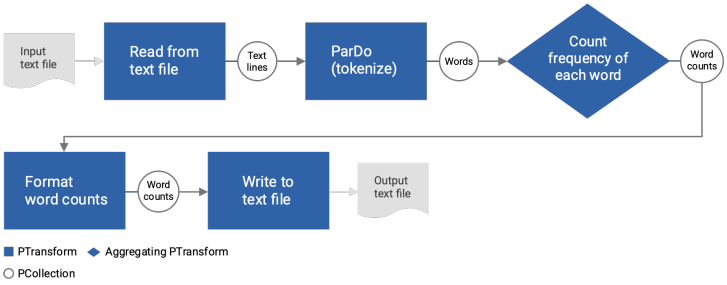
- A layer of abstraction on top of processing frameworks
- Provides a unified programming model
  - Allows to define batch and streaming data processing pipelines that run on supported execution engines, including Flink, Spark, Google Cloud Dataflow
    - Write once, run anywhere
  - Programming languages: Java, Python, Go
- Engine-specific runners translate Beam code to target runtime
- Initially developed by Google, now open-source top-level Apache project

Using Beam: key concepts

- PCollection: represents a collection of data, which could be bounded or unbounded in size
- PTransform: represents a computation that transforms input PCollections into output PCollections.
- Pipeline: manages a DAG of PTransforms and PCollections that is ready for execution
- PipelineRunner: specifies where and how the pipeline should execute

Using Beam: key concepts

- Create the Pipeline
  - PipelineOptions object
- Read data input
  - E.g., text files
- Apply pipeline transformations
- Write output
  - E.g., to text file
- Run the Pipeline



Using Beam: WordCount in Python

```
# We use the save_main_session option because one or more DoFn's in this
# workflow rely on global context (e.g., a module imported at module level).
pipeline_options = PipelineOptions(pipeline_args)
pipeline_options.view_as(SetupOptions).save_main_session = save_main_session
with beam.Pipeline(options=pipeline_options) as p:

  # Read the text file[pattern] into a PCollection.
  lines = p | ReadFromText(known_args.input)

  # Count the occurrences of each word.
  counts = (
      lines
      | 'Split' >> (
          beam.FlatMap(lambda x: re.findall(r'[A-Za-z\']+', x)).
          with_output_types(unicode))
      | 'PairWithOne' >> beam.Map(lambda x: (x, 1))
      | 'GroupAndSum' >> beam.CombinePerKey(sum))

  # Format the counts into a PCollection of strings.
  def format_result(word_count):
    (word, count) = word_count
    return '%s: %s' % (word, count)

  output = counts | 'Format' >> beam.Map(format_result)

  # Write the output using a "Write" transform that has side effects.
  # pylint: disable=expression-not-assigned
  output | WriteToText(known_args.output)
```

See github.com/apache/beam/blob/master/sdks/python/apache_beam/examples/wordcount.py

Beam: pros and cons

- Pros
  - Single, unified programming model
  - Flexibility to switch underlying processing system with low effort
- Con:
  - Impact on performance
    - Dated evaluation (2019): slowdown >= 3x with respect to same programs developed using native system APIs

      Quantitative Impact Evaluation of an Abstraction Layer for Data Stream Processing Systems, ICDCS '19

DSP in the Cloud

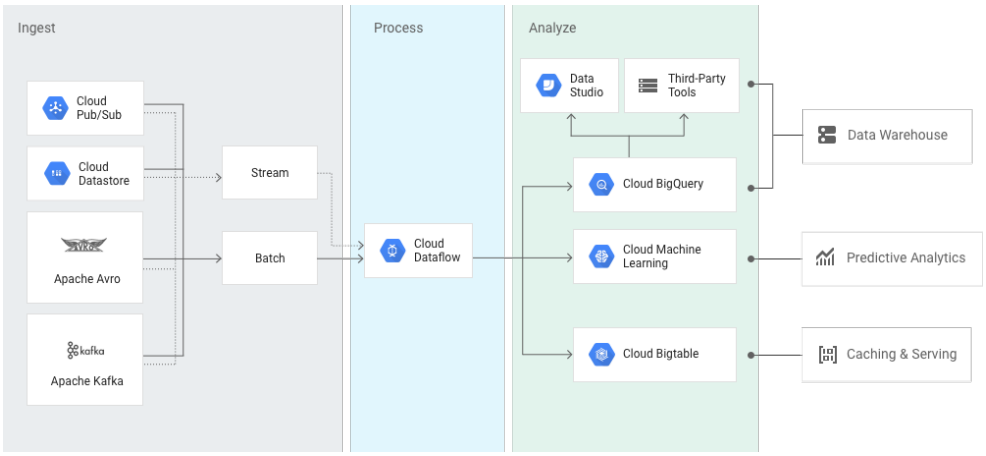
- Data streaming systems also as Cloud services
  - Amazon Kinesis Data Streams
  - Google Cloud Dataflow
  - IBM Streaming Analytics
  - Microsoft Azure Stream Analytics
- Abstract underlying service infrastructure and support dynamic scaling of computing resources
- Appear to execute in a single data center (i.e., no geo-distribution)

Google Cloud Dataflow

- Fully-managed data processing service, supporting both stream and batch data processing
  - Automated resource management
  - Dynamic work rebalancing
  - Horizontal auto-scaling
- Provides a unified programming model based on Apache Beam
  - Apache Beam SDK in Java and Python
  - Enable developers to implement custom extensions and choose other execution engines
- Provides exactly-once processing
  - MillWheel is Google's internal version of Cloud Dataflow

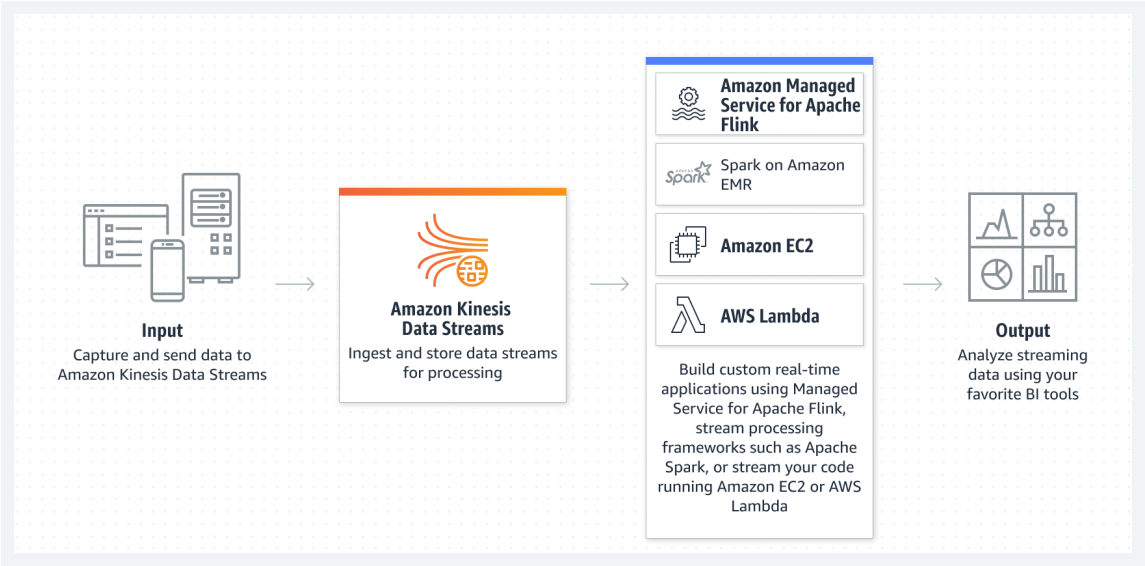
Google Cloud Dataflow

- Can be seamlessly integrated with GCP services for streaming events ingestion (Cloud Pub/Sub), data warehousing (BigQuery), machine learning (Cloud Machine Learning)



Amazon Kinesis Data Streams

- Allows to collect and ingest streaming data at scale for real-time analytics



Amazon Kinesis Data Streams

- Serverless, fully managed Apache Flink: allows to process data streams in real time
  - Based on Flink: same operators to filter, aggregate and transform streaming data
  - Per-hour pricing based on number of Kinesis Processing Units (KPUs) used to run application
    - Horizontal auto-scaling of KPUs

References

- Akidau, Streaming 101: The world beyond batch, 2015
- Carbone et al., Apache Flink: Stream and batch processing in a single engine*, Bulletin IEEE Comp. Soc. Tech. Comm. on Data Eng.,* 2015
- Carbone et al., State management in Apache Flink®: consistent stateful distributed stream processing, *Proc. VLDB Endowment*, 2017
- Carbone et al., A survey on the evolution of stream processing systems, *VLDB Journal*, 2023