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. 2021/22
Valeria Cardellini

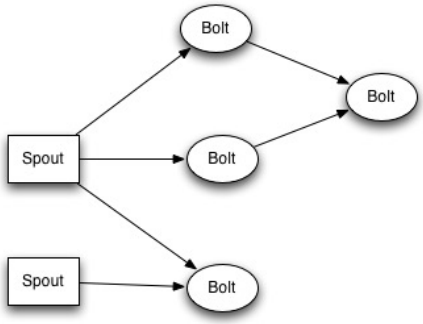
Laurea Magistrale in Ingegneria Informatica

DSP frameworks we consider

- Apache Storm
- Twitter Heron
- Apache Flink (with lab)
- Apache Spark Streaming (with lab)
- Kafka Streaming (lab)
- Cloud-based frameworks
  - Google Cloud Dataflow
  - Amazon Kinesis

Apache Storm

- Apache Storm https://storm.apache.org
  - Open-source, real-time, scalable streaming system
  - Provides an abstraction layer to execute DSP applications
  - Initially developed by Twitter
  - Current version: 2.4
- Main concept: topology
  - DAG of spouts (sources of streams) and bolts (operators and data sinks)
  - Top-level abstraction submitted to Storm for execution

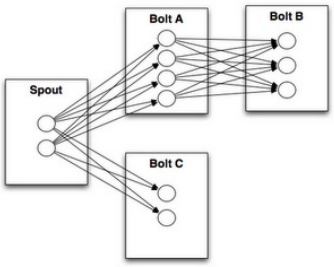


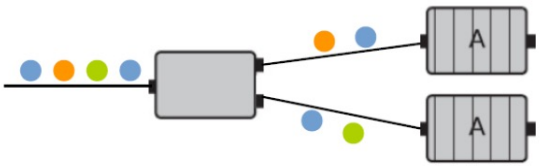
Storm: topology API

- Storm uses tuples as its data model
  - Tuple: named list of values, and 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 an object of another type, you need to implement a serializer for the type

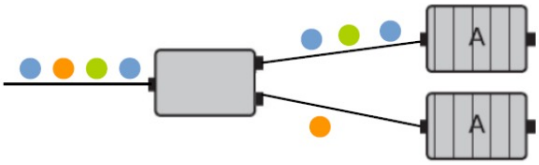
Storm: stream grouping

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



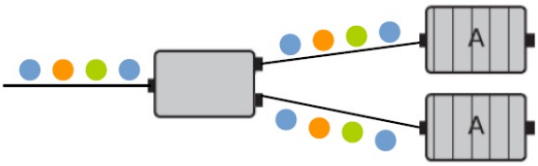


- Field grouping
  - Stream is partitioned by the fields specified in the grouping



Storm: stream grouping

- All grouping (i.e., broadcast)
  - Stream is replicated to all the bolt's tasks



- Global grouping
  - Stream goes to a single one of the bolt's task
- Direct grouping
  - The producer of the tuple decides which task of the consumer will receive this tuple

Storm: a simple topology

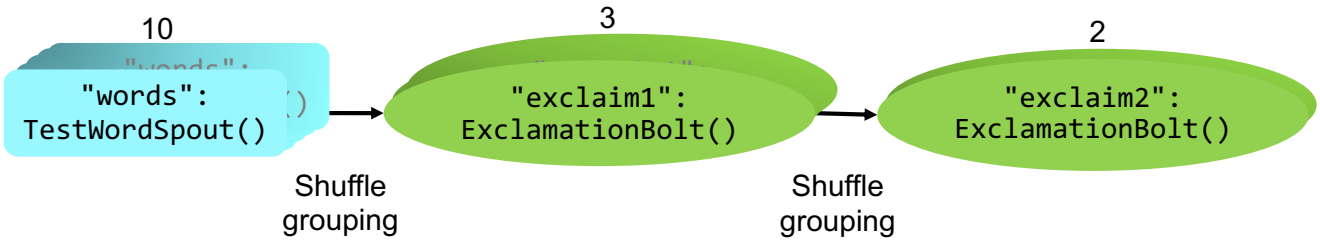
- First example: exclamation
  - Spout emits words, each bolt appends "!!!" to its input

https://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
- 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"));
```

Full example at https://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 https://streamparse.readthedocs.io/

Storm: windowing

- Windowing support in core Storm: 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

```
// time based sliding window
stream.window(SlidingWindows.of(Duration.minutes(10), Duration.minutes(1)));

// count based sliding window
stream.window(SlidingWindows.of(Count.(10), Count.of(2)));

// tumbling window
stream.window(TumblingWindows.of(Duration.seconds(10));

// specifying timestamp field for event time based processing and a late tuple stream.
stream.window(TumblingWindows.of(Duration.seconds(10)
                    .withTimestampField("ts")
                    .withLateTupleStream("late_events"));
```


Storm: Stream APIs

- Alternative interface to Storm: provides a typed API for expressing streaming computations and supports functional style operations (similar to Spark and Flink)
  - Still experimental

  https://storm.apache.org/releases/2.4.0/Stream-API.html
- Stream APIs: Stream and PairStream
- Support a wide range of operations: transformations, filters, windowing, aggregations, branching, joins, stateful, output and debugging operations

Stream APIs

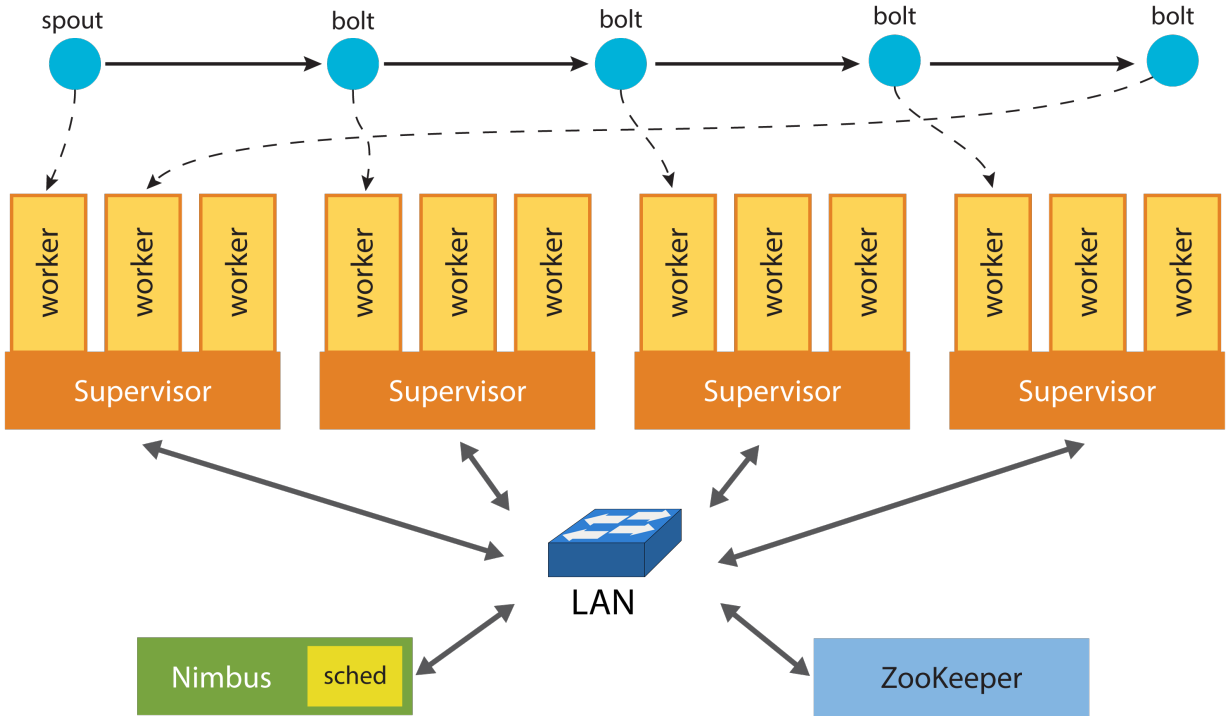
- Example

```
// imports
import org.apache.storm.streams.Stream;
import org.apache.storm.streams.StreamBuilder;
...

StreamBuilder builder = new StreamBuilder();

// a stream of sentences obtained from a source spout
Stream<String> sentences = builder.newStream(new RandomSentenceSpout()).map(tuple -> tuple.getString(0));

// a stream of words obtained by transforming (splitting) the stream of sentences
Stream<String> words = sentences.flatMap(s -> Arrays.asList(s.split(" ")));

// output operation that prints the words to console
words.forEach(w -> System.out.println(w));
```

- A Stream supports two kinds of operations:
  - **Transformations:** produce another stream from the current one (e.g., `map`, `flatMap`)
  - **Output operations**: produce a result (e.g., `forEach`)

Storm: architecture

- Master-worker architecture

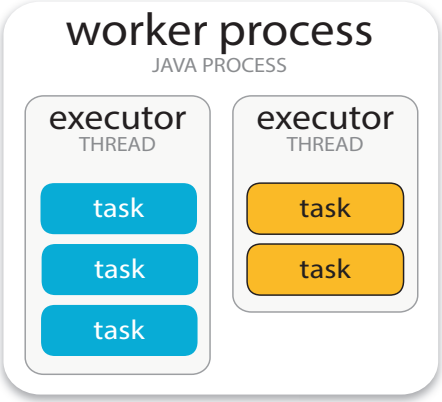


Storm components: Nimbus and Zookeeper

- Nimbus
  - Master node
  - Clients 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 state about the topology

Storm components: worker

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



Storm components: supervisor

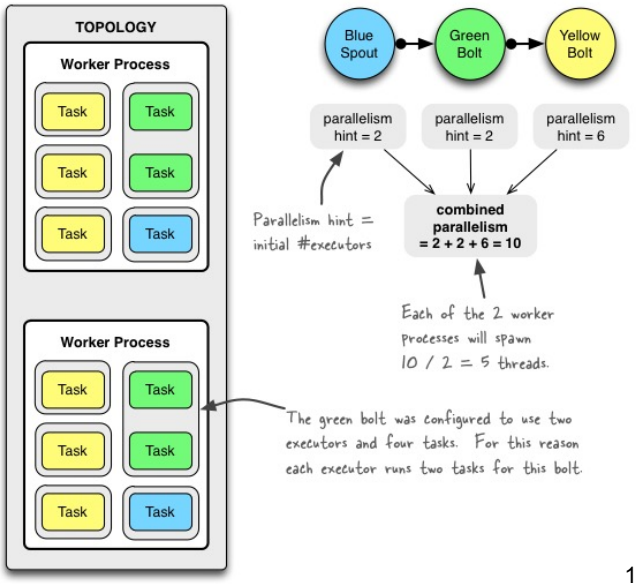
- Each worker node runs a supervisor

The 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 they are currently running, and any vacancies that are available to run more topologies

Storm: running topology

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



See https://storm.apache.org/releases/2.4.0/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
  - Needs to maintain the link between the spout tuple and its child tuples so to detect when the tree of tuples is fully 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
- Storm offers at-least-once semantics
  - Use Trident (high-level abstraction on top of Storm) for exactly-once semantics

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 in 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 https://storm.apache.org/releases/2.4.0/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
  - Alternative interface to Storm, provides exactly-once processing
- SQL
  - To run SQL queries over streaming data

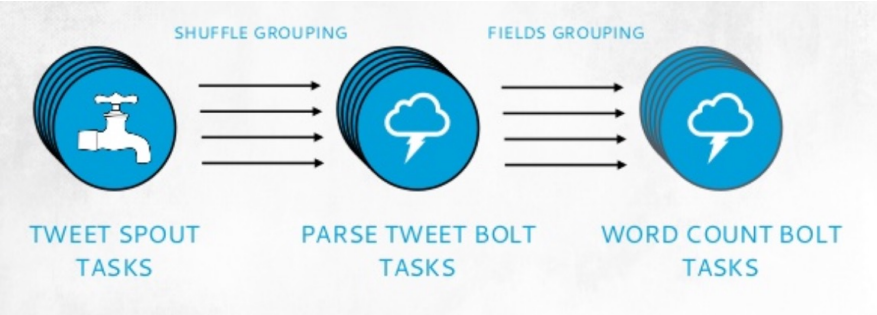
https://storm.apache.org/releases/2.4.0/storm-sql.html

Twitter Heron

- Real-time, distributed, fault-tolerant stream processing engine from Twitter
- Developed as direct successor of Storm
  - Released as open source in 2016 https://heron.incubator.apache.org/
  - Stream data processing engine used at Twitter
- Goal of overcoming Storm's performance, reliability, and other shortcomings
- Compatibility with Storm
  - API compatible with Storm: no code change is required for migration

Heron: in common with Storm

- Same terminology of Storm
  - Topology, spout, bolt
- Same stream groupings
  - Shuffle, fields, all, global
- Example: WordCount topology



Heron: design goals

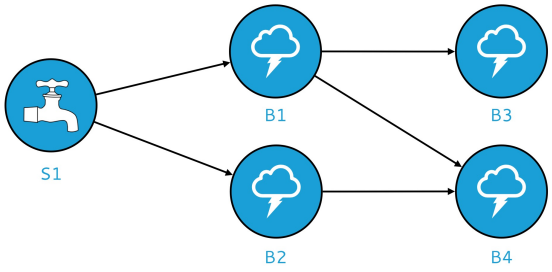
- Isolation
  - Process-based topologies rather than thread-based
  - Each process runs in isolation (easy debugging, profiling, and troubleshooting)
  - Goal: overcoming Storm's performance, reliability, and other shortcomings
- Resource constraints
  - Safe to run in shared infrastructure: topologies use only initially allocated resources and never exceed bounds
- Compatibility
  - Fully API and data model compatible with Storm

Heron: design goals

- Backpressure
  - Built-in rate control mechanism to ensure that topologies can self-adjust in case components lag
  - Heron dynamically adjusts the rate at which data flows through the topology using backpressure
- Performance
  - Higher throughput and lower latency than Storm
  - Enhanced configurability to fine-tune potential latency/throughput trade-offs
- Semantic guarantees
  - Support for both at-most-once and at-least-once processing semantics
- Efficiency
  - Minimum possible resource usage

Heron: topology

- Similarly to Storm, a Heron topology is a DAG used to process streams of data and consists of spouts and bolts
  - Spouts inject data from external sources like pub-sub messaging systems (Apache Kafka, Apache Pulsar, etc.)
  - Bolts apply user-defined processing logic to data supplied by spouts, can be stateless or stateful



Heron APIs

- Main APIs
  - Heron Topology API
  - Heron Streamlet API
- Topology API: lower-level API based on Storm topology API
  - Specify spout and bolt logic directly
  - Same window types as in Storm: tumbling and sliding
  - As in Storm windows can be based on count or time (4 total types of windows)
  - Available in Java and Python

Heron Streamlet API

- As in Storm, shift from procedural to functional style
- Let's examine Heron Streamlet API

Domain	Original topology API	Heron Streamlet API
Programming style	Procedural, processing component based	Functional
Abstraction level	**Low level.** Developers must think in terms of "physical" spout and bolt implementation logic.	**High level.** Developers can write processing logic in an idiomatic fashion in the language of their choice, without needing to write and connect spouts and bolts.
Processing model	Spout and bolt logic must be created explicitly, and connecting spouts and bolts is the responsibility of the developer	Spouts and bolts are created for you automatically on the basis of the processing graph that you build

Heron Streamlet API

- Processing graphs consist of streamlets
  - One or more supplier streamlets inject data into DAG to be processed by downstream operators
- Operations (similar to Spark)

Operation	Description	Example
`map`	Create a new streamlet by applying the supplied mapping function to each element in the original streamlet	Add 1 to each element in a streamlet of integers
`flatMap`	Like a map operation but with the important difference that each element of the streamlet is flattened	Flatten a sentence into individual words
`filter`	Create a new streamlet containing only the elements that satisfy the supplied filtering function	Remove all inappropriate words from a streamlet of strings
`union`	Unifies two streamlets into one, without modifying the elements of the two streamlets	Unite two different `Streamlet<String>`s into a single streamlet
`clone`	Creates any number of identical copies of a streamlet	Create three separate streamlets from the same source

Heron API: shift to functional style

- Operations (continued)

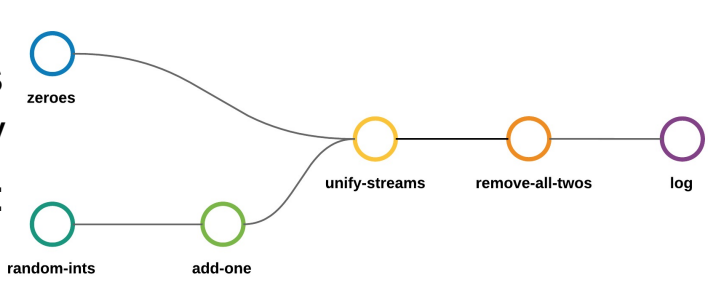
`transform`	Transform a streamlet using whichever logic you'd like (useful for transformations that don't neatly map onto the available operations)	
`join`	Create a new streamlet by combining two separate key-value streamlets into one on the basis of each element's key	Combine key-value pairs listing current scores (e.g. `("h4x0r", 127)`) for each user into a single per-user stream
`reduceByKeyAndWindow`	Produces a streamlet out of two separate key-value streamlets on a key, within a time window, and in accordance with a reduce function that you apply to all the accumulated values	Count the number of times a value has been encountered within a specified time window
`repartition`	Create a new streamlet by applying a new parallelism level to the original streamlet	Increase the parallelism of a streamlet from 5 to 10
`toSink`	Sink operations terminate the processing graph by storing elements in a database, logging elements to stdout, etc.	Store processing graph results in an AWS Redshift table
`log`	Logs the final results of a processing graph to stdout. This *must* be the last step in the graph.	
`consume`	Consume operations are like sink operations except they don't require implementing a full sink interface (consume operations are thus suited for simple operations like logging)	Log processing graph results using a custom formatting function

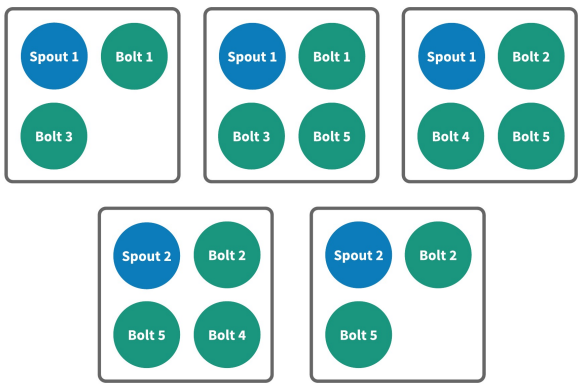
Heron: topology lifecycle

- Topology lifecycle managed through Heron's CLI tool
- Stages
  - Submit topology to cluster
  - Activate topology
  - Restart active topology if, e.g., after updating its configuration
  - Deactivate topology
  - Kill topology to completely remove it from cluster

Heron topology: logical and physical plans

- Topology's **logical plan**: maps out basic operations associated with a topology
- Topology's **physical plan**: determines the "physical" execution logic of a topology, i.e. how topology processes are divided between Heron containers
- Both are automatically created by Heron

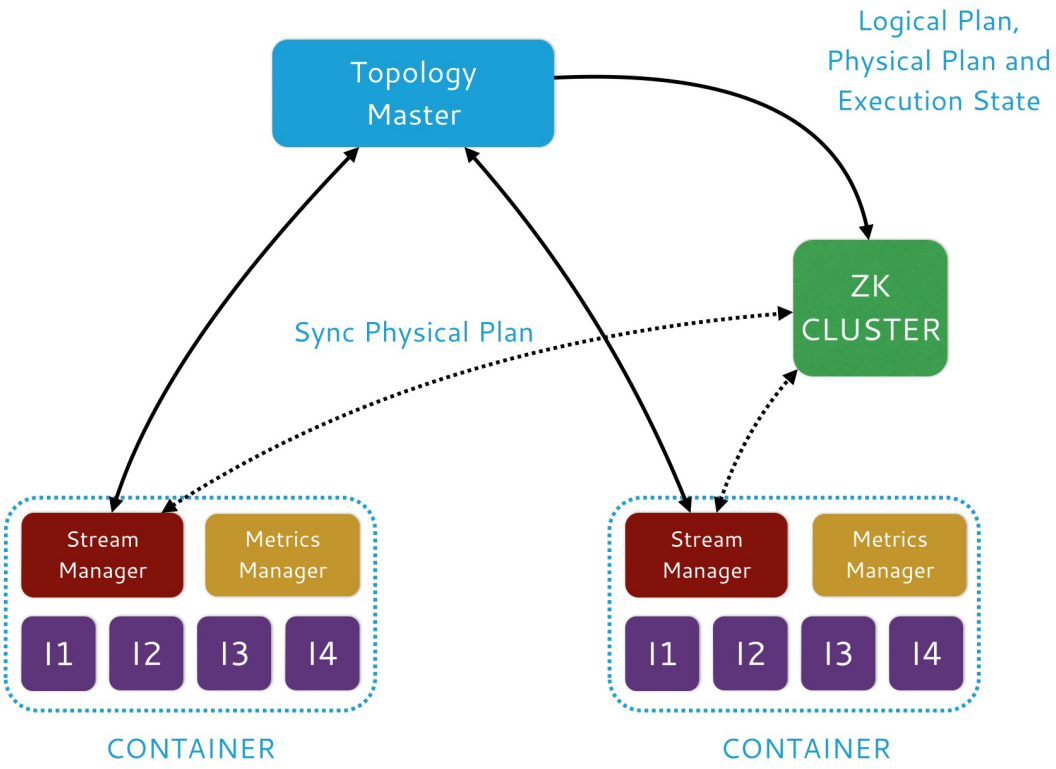




Heron architecture per topology

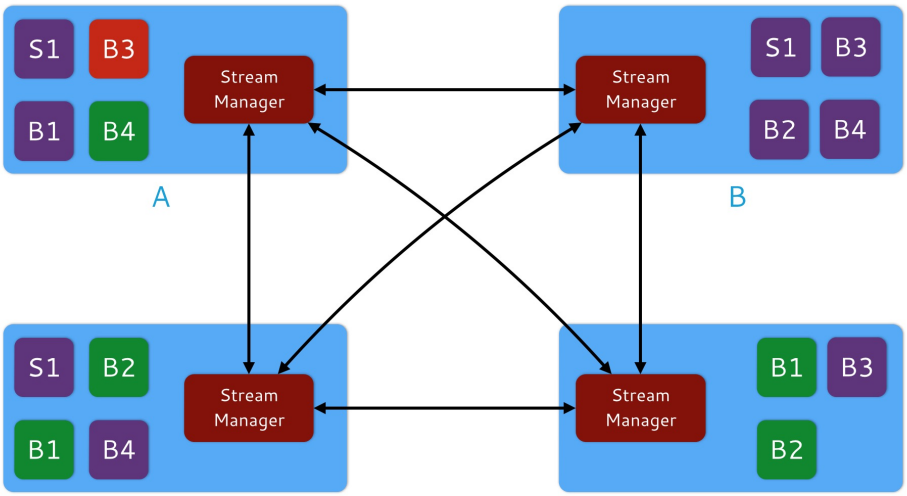
- Master-work architecture
- One Topology Master (TM)
  - Manages a topology throughout its entire lifecycle
- Multiple Containers
  - Each Container contains:
    - multiple Heron Instances
    - a Stream Manager
    - a Metrics Manager
  - *Heron Instance*: process that handles a single task of a spout or bolt
  - Containers communicate with TM to ensure that the topology forms a fully connected graph

Heron architecture per topology



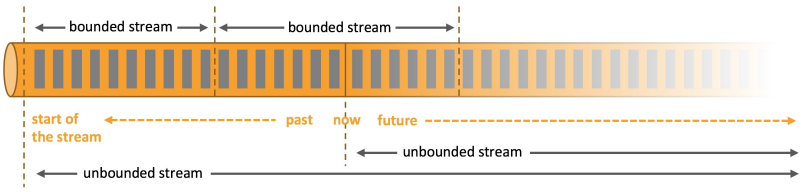
Heron architecture per topology

- Stream Manager (SM): routing engine for data streams
  - Each Heron container connects to its local SM, while all of the SMs in a given topology connect to one another to form a network
  - Responsible for propagating backpressure



Unbounded vs. bounded streams

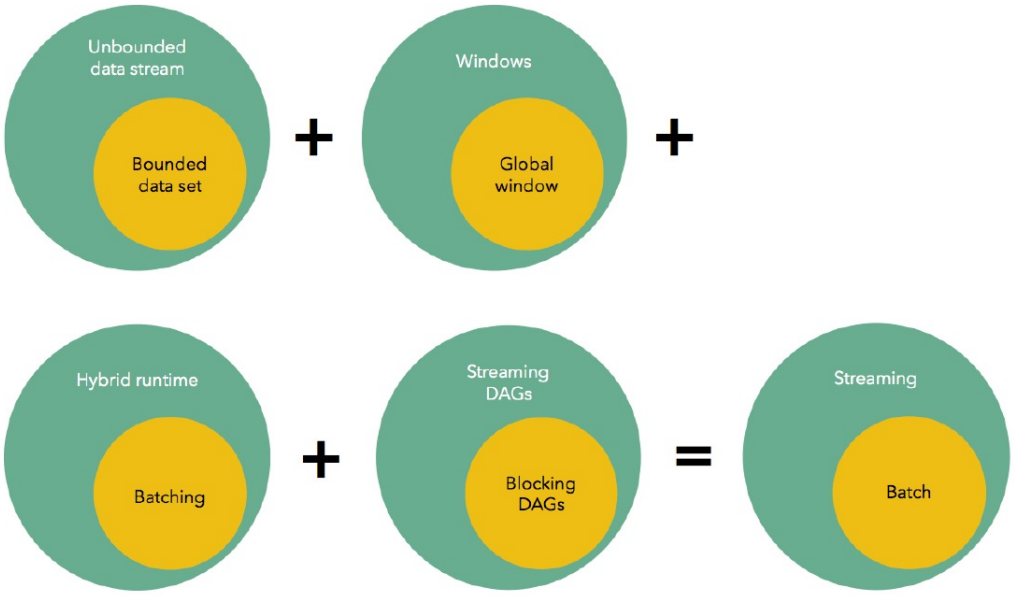
- Data can be processed as bounded or unbounded streams



- 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

- Batch processing as special case of stream processing

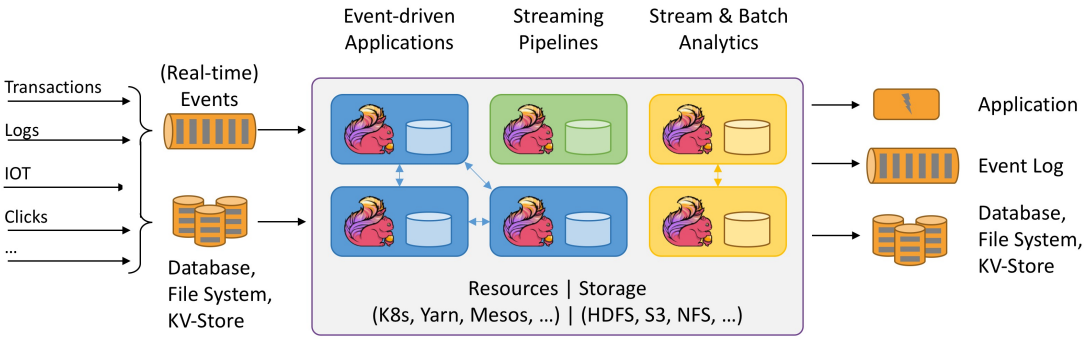


Batch processing vs. stream processing

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

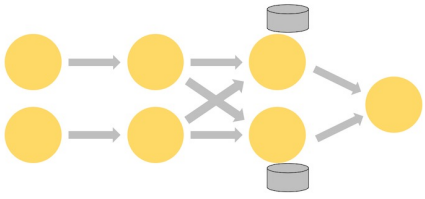
Apache Flink

- Distributed processing system for stateful computation over bounded and bounded data streams
  - 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 stable version: 1.15

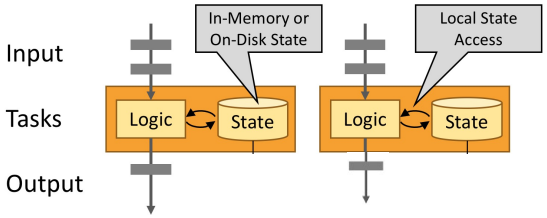


Flink: Stateful computation

- 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

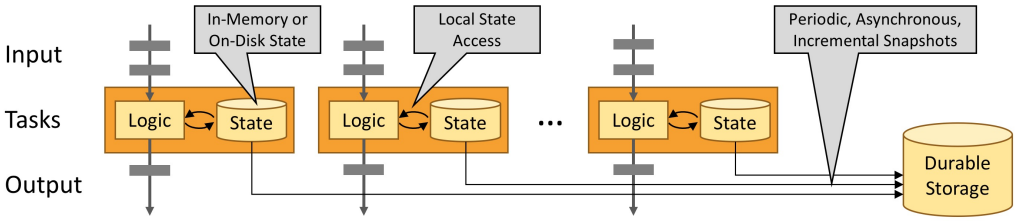


- State is optimized for local access
  - Maintained in memory or in access-efficient on-disk data structures
  - Goals: high throughput and low latency



Flink: fault tolerance

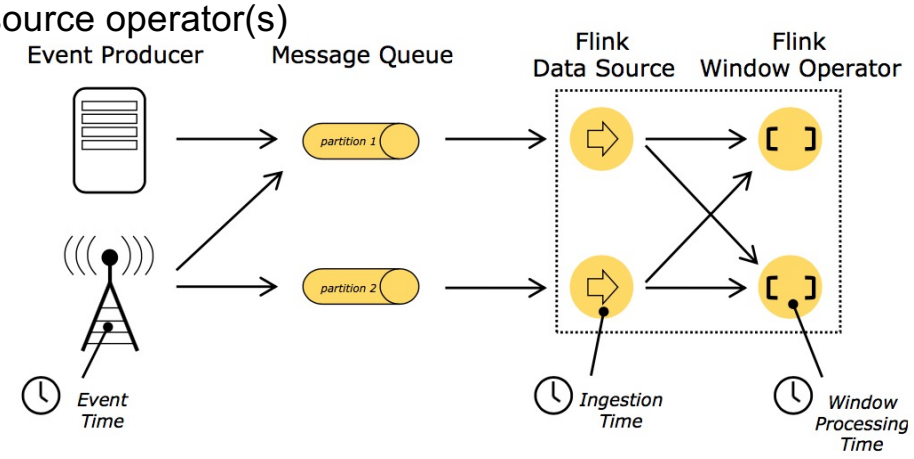
- Flink guarantees exactly-once state consistency in case of failures by *periodically* and *asynchronously* checkpointing local state to durable storage (state snapshot)
  - State of operators can be restored from checkpoint to an earlier point in time and records are reset to the point of the state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



Flink docs: https://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 (system time of the machine executing the operator)
  - Event time: time at which an event actually occurred on its producing device
    - Usually described by a timestamp in the event
  - Ingestion time: time when an event enters the dataflow at the source operator(s)



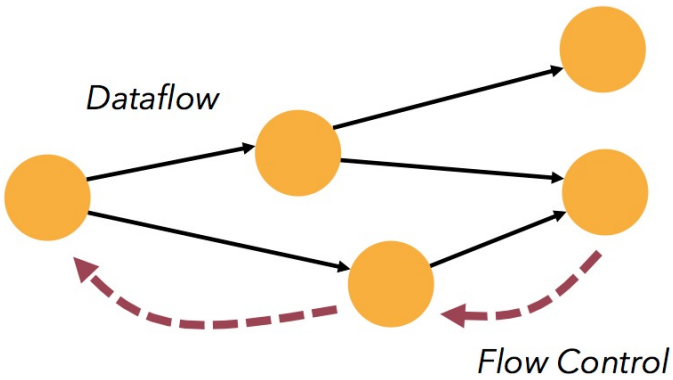
Flink: time

- Flink supports all the 3 notions of time
  - Internally, ingestion time is treated similarly to event time
- Event time makes it easy to compute over streams where events arrive *out-of-order*, and where events may arrive delayed
- How to measure the progress of event time?
  - Flink uses watermarks

https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/concepts/time/

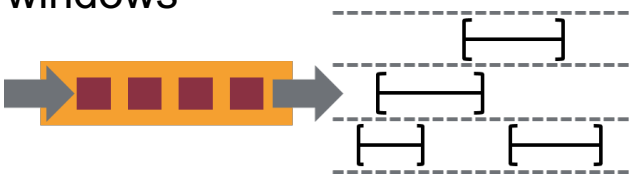
Flink: backpressure

- Continuous streaming with backpressure
  - Flink's streaming runtime provides flow control: slow downstream operators backpressure faster upstream operators
  - Flink's UI allows to monitor backpressure behavior of running jobs
    - Back pressure warning (e.g., `High`) for an upstream operator



Flink: windows

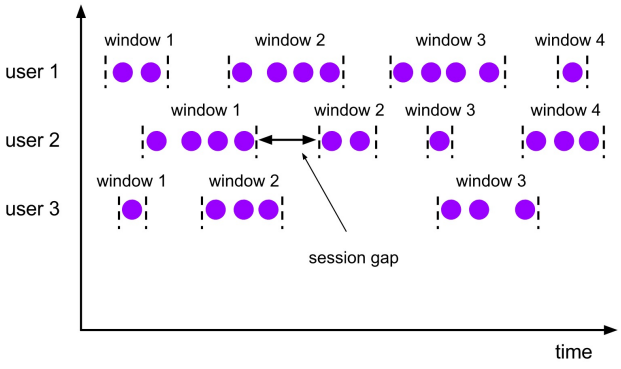
- Highly flexible streaming windows
  - Also user-defined windows



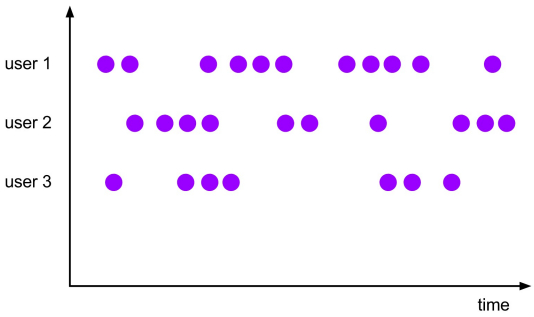
- Supported types of windows:
  - Tumbling: no overlap
  - Sliding: overlap
  - Session
  - Global

Flink: windows

- Session windows
  - To group elements by sessions of activity
  - Differently from tumbling and sliding windows, do not overlap and do not have a fixed start and end time
  - A session window closes when a gap of inactivity occurs

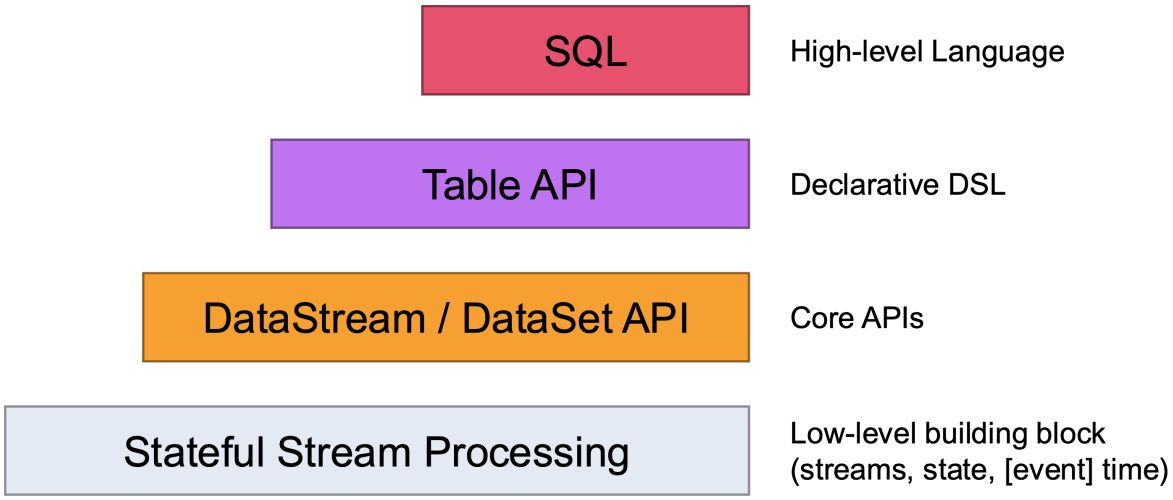


- Global windows
  - To assign all elements with the same key to the same single global window
  - Only useful if you also specify a custom trigger



Flink: APIs

- Different levels of abstraction to develop streaming/batch applications
- On top: libraries with high-level APIs for different use cases
- APIs in Java, Scala and Python

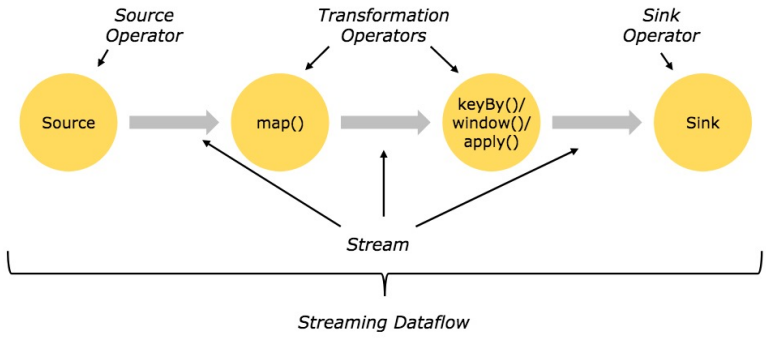


Flink: APIs

- Data streaming applications: DataStream API
  - Supports transformations on data streams (e.g., filtering, updating state, defining windows, aggregating), with user-defined state and flexible windows
  - Provides fine-grained control over state and time
- Table API & SQL
  - Two relational APIs for unified stream and batch processing
- Python API
  - Python API for Apache Flink: PyFlink DataStream API and PyFlink Table API
- See lab lesson

Flink: programming model

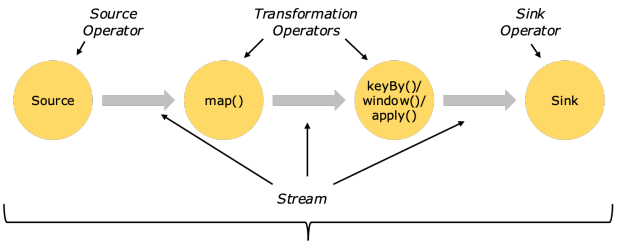
- Applications are composed of streaming dataflows that are transformed by user-defined operators
  - Streams: unbounded, partitioned, immutable sequence of events
- Streaming dataflows form directed graphs (usually DAGs) that start with one or more sources, and end in one or more sinks
  - DAGs basic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



Flink: programming model

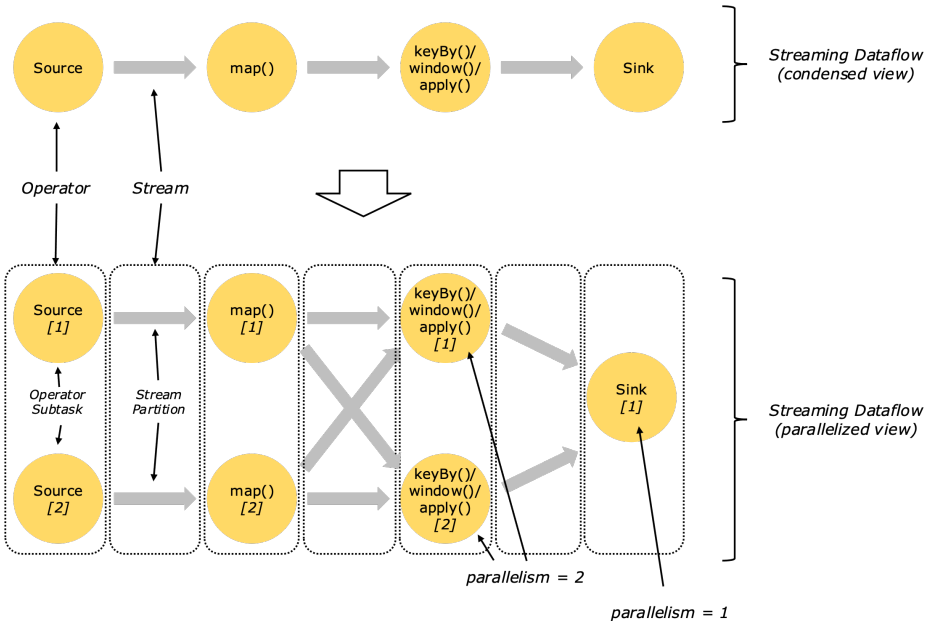
- Stream operators
  - Stream transformations that take one or more streams as input, and produce one or more output streams as a result

```
DataStream<String> lines = env.addSource(
                           new FlinkKafkaConsumer<>(…));          Source

DataStream<Event> events = lines.map((line) -> parse(line));      Transformation

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

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



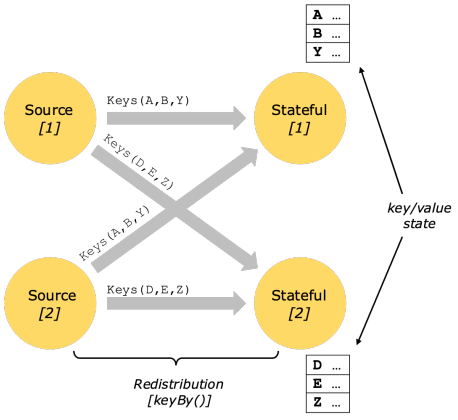
Flink: programming model

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



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

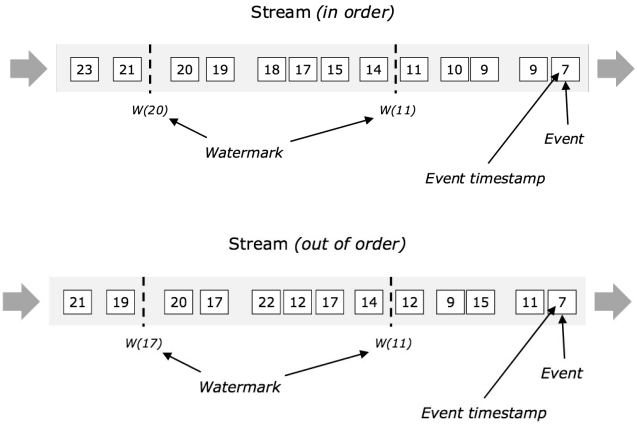


Flink: control events

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

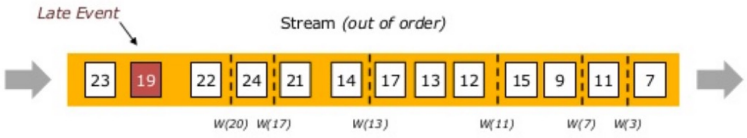
Flink: watermarks

- 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

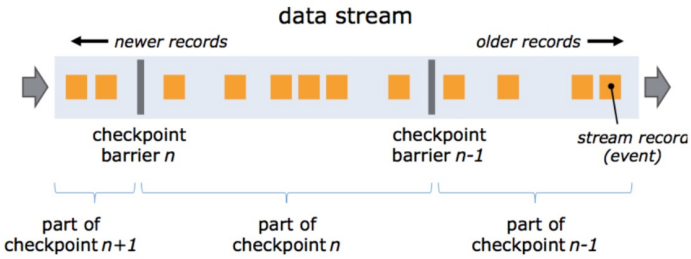


Flink: watermarks

- 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

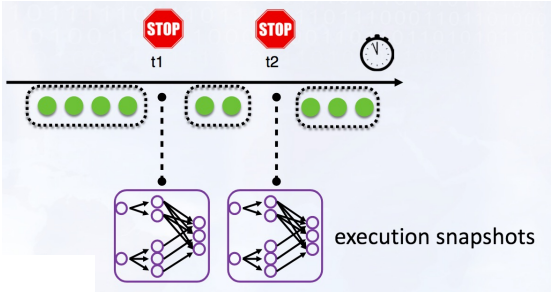
- To provide fault tolerance special barrier markers (called checkpoint barriers) are periodically injected at streams sources and then pushed downstream up to sinks



Fault tolerance

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

  Snapshot: global state of the 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: fault tolerance

- Flink offers a lightweight snapshotting mechanism
  - Allows to maintain high throughput and provides strong consistency guarantees at the same time
- Such mechanism:
  - Draws consistent snapshots of stream flows and operators' state
  - Even in presence of failures, the application state will reflect every record from the data stream exactly once
  - State stored at configurable place
  - Disabled by default
- Inspired by Chandy-Lamport algorithm for distributed snapshot and tailored to Flink's execution model

https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/concepts/stateful-stream-processing/

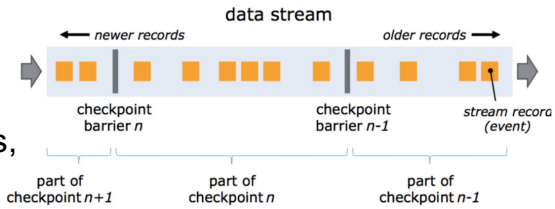
https://arxiv.org/abs/1506.08603

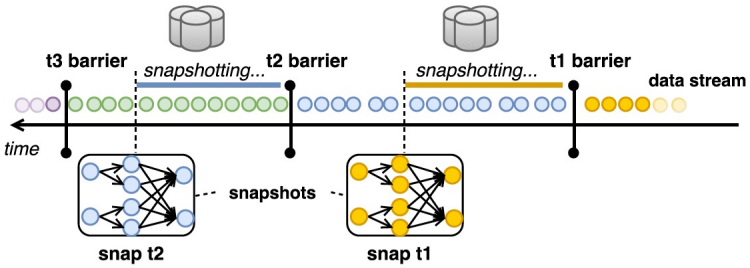
Chandy-Lamport algorithm

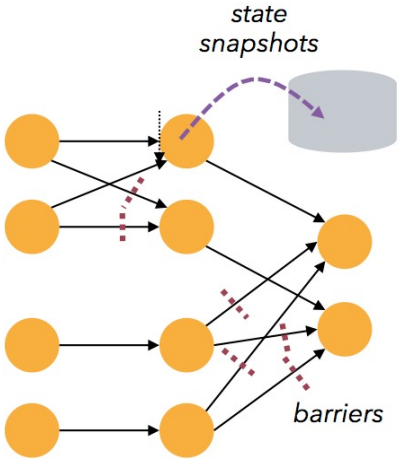
- The observer process (process initiating the snapshot):
  - Saves its own local state
  - Sends a snapshot request message bearing a snapshot token to all other processes
- If a process receives the token *for the first time*:
  - Sends the observer process its own saved state
  - Attaches the snapshot 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: fault tolerance

- Uses checkpoint barriers
  - When an operator has received 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 that snapshot *n* to the checkpoint coordinator. After all sinks have acknowledged a snapshot, it is considered completed



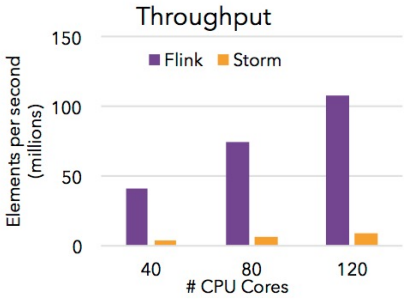


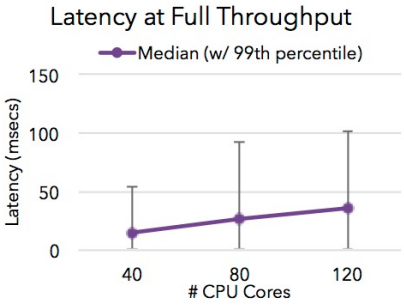


https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/learn-flink/fault_tolerance/

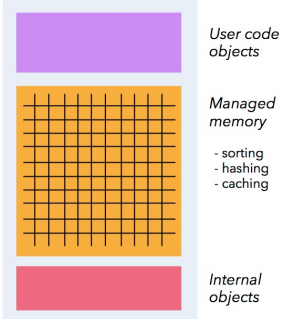
Flink: performance and memory management

- High throughput and low latency



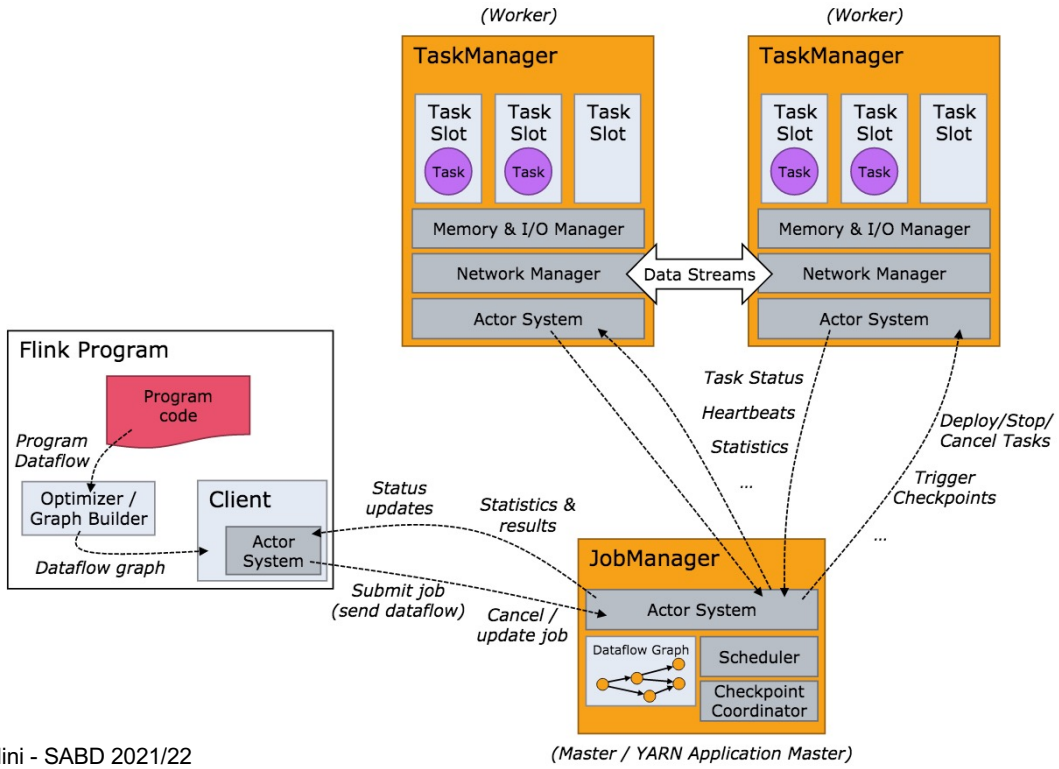


- Memory management
  - Flink implements its own memory management inside JVM



Flink: architecture

- The usual master-worker architecture



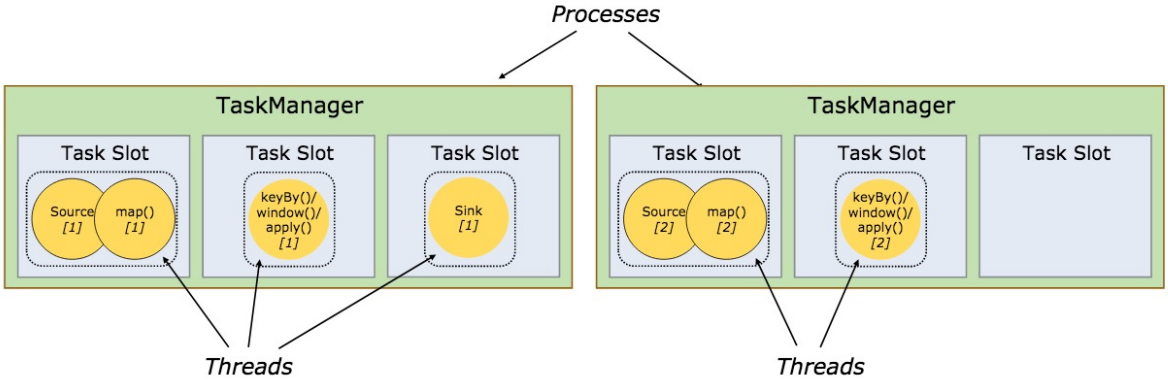
Flink: architecture

- JobManager (master): is responsible to responsibilities related to coordinate the 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 a 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 WebUI
  - **JobMaster**: responsible for managing the execution of a single JobGraph

https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/

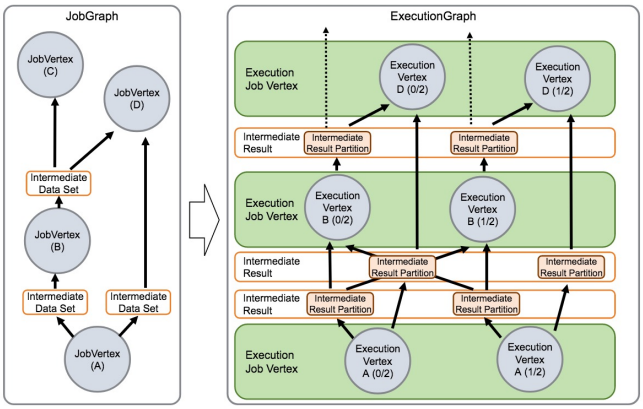
Flink: architecture

- TaskManagers (workers): JVM processes that execute tasks of a dataflow, and buffer and exchange the data streams
  - Workers use task slots to control the number of tasks they accept (at least one)
  - Each task slot represents a fixed subset of resources of the worker



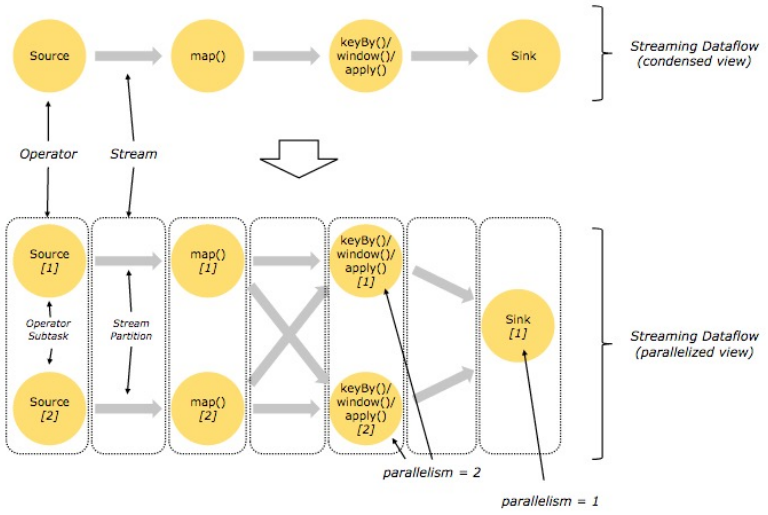
Flink: application execution

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



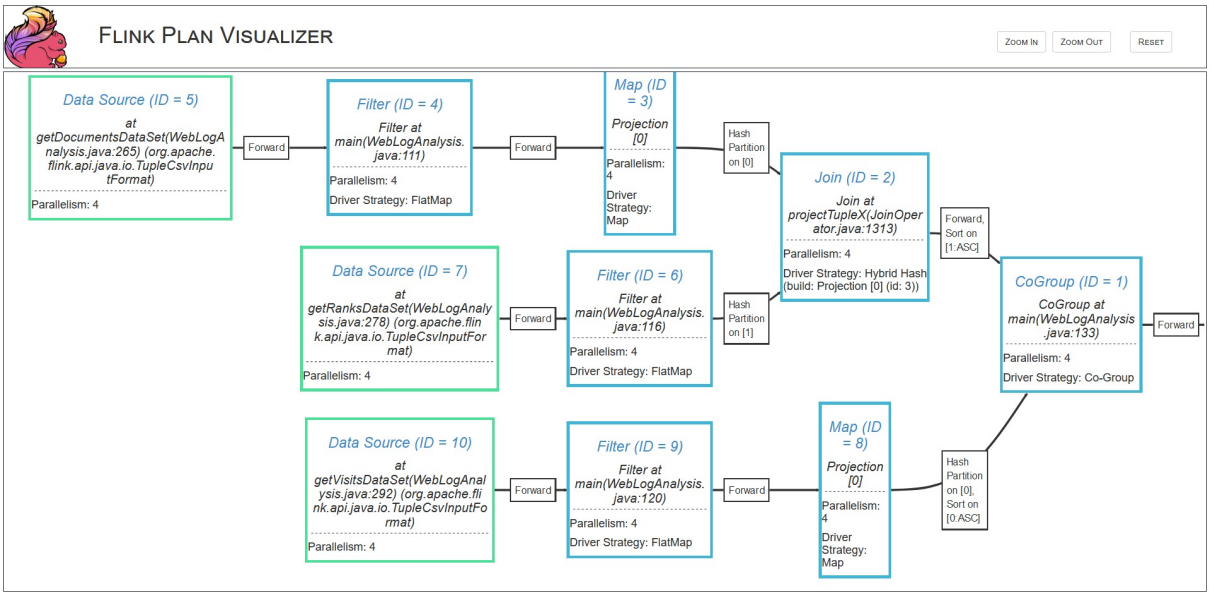
Flink: application execution

- Data parallelism
  - Different operators of same program 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 visualization tool for execution plans



Flink: application 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
  - Availability
  - 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

  https://nightlies.apache.org/flink/flink-docs-release-1.15/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 impact performance): to enable `latencyTrackingInterval` must be > 0

Flink: application monitoring

- Application-specific metrics can be added
  - E.g., counters for number of invalid records
- All metrics can be
  - Queried via Flink's Monitoring REST-ful API that accepts HTTP requests and responds with JSON data https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/ops/rest_api/
  - Visualized in Flink dashboard (Metrics tab)
  - Sent to external systems (e.g., Graphite and InfluxDB)

See https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/ops/metrics/

Flink: deployment

- Designed to run on large-scale clusters with many thousands of nodes
- Can be run in a fully distributed fashion on a *static* (but 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

Towards strict delivery guarantees

- Most frameworks provide at-least-once delivery guarantees (e.g., Storm, Samza)
  - For stateful non-idempotent operators such as counting, at-least-once delivery guarantees can give incorrect results
- Flink, Storm plus Trident, and Google's MillWheel offer stronger delivery guarantees (i.e., exactly-once)
  - Exactly-once low latency stream processing in MillWheel works as follows:
    - The record is checked against de-duplication data from previous deliveries; duplicates are discarded
    - User code is run for the input record, possibly resulting in pending changes to timers, state, and productions
    - Pending changes are committed to the backing store
    - Senders are acked
    - Pending downstream productions are sent

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
Heron	Low-level High-level No SQL No batch	Yes	At least once Effectively once		Limited	Back pressure	Yes (with Dhalion)
Flink	High-level SQL Also batch	Yes, also used-def.	At least once Exactly once	Checkpoint.	Yes	Back pressure	No

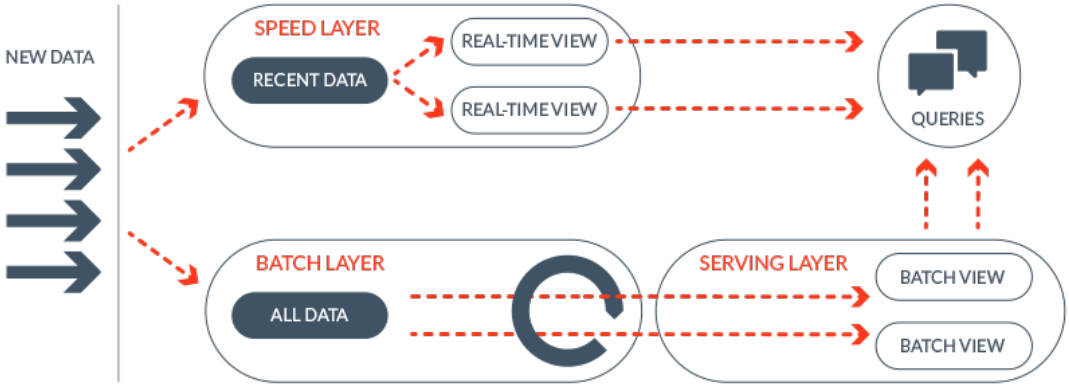
A recent need

- 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

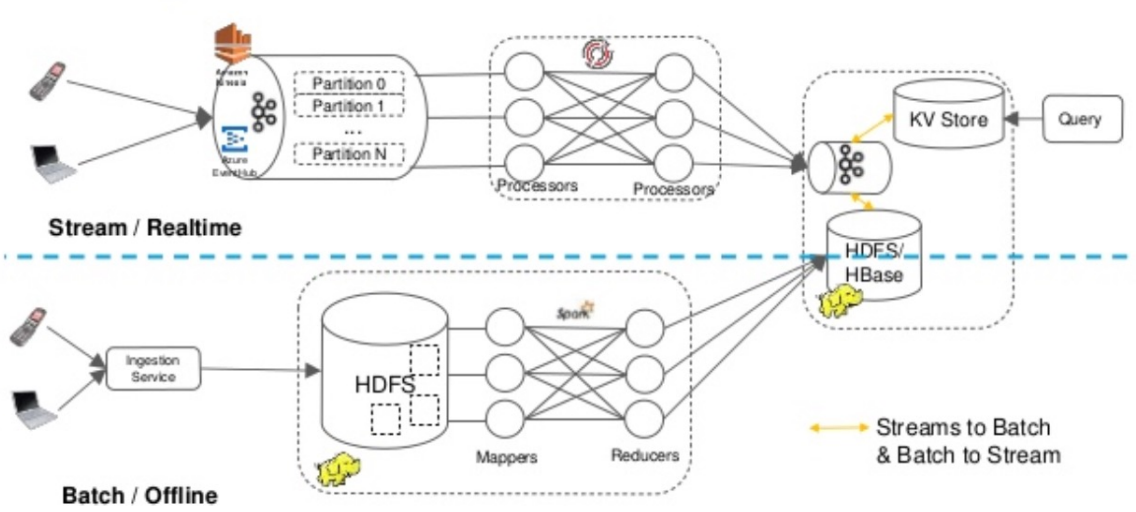
- Data-processing design pattern to integrate batch and real-time processing
- Streaming framework used to process real-time events, and, in parallel, batch framework to process the entire dataset
- Results from the two parallel pipelines are then merged



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

Lambda architecture: example

- Lambda architecture used at LinkedIn before Samza development



Lambda architecture: pros and cons

- Pros:
  - Flexibility in the 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

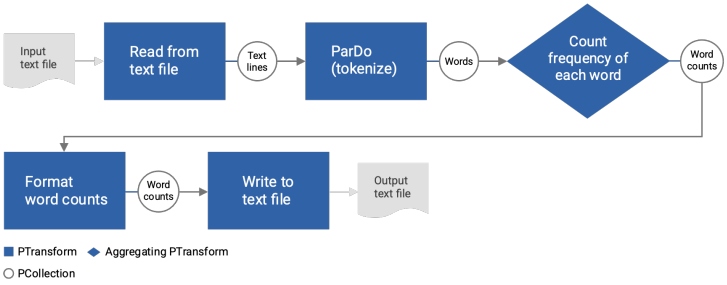
- Use a unified (Lambda-less) design for processing both real-time as well as batch data using the same data structure
- Spark and Flink follow this trend

Unified programming model: Apache Beam

- A new layer of abstraction
- Provides advanced unified programming model
  - Allows to define batch and streaming data processing pipelines that run on any supported execution engine (Flink, Spark, Samza, Google Cloud Dataflow)
  - Java, Python and Go as programming languages
- Engine-specific runners translate the Apache Beam code to the target runtime
- Developed by Google and released as open-source top-level project

Using Beam: key concepts

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



Example: WordCount in Python using Beam

```
# 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 https://bit.ly/3dk5RLe

Beam: pros and cons

- Pros
  - A single, unified programming model
  - Flexibility to switch underlying DSP system with relatively low effort

- Cons:
  - Noticeable impact on performance of DSP systems
    - 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

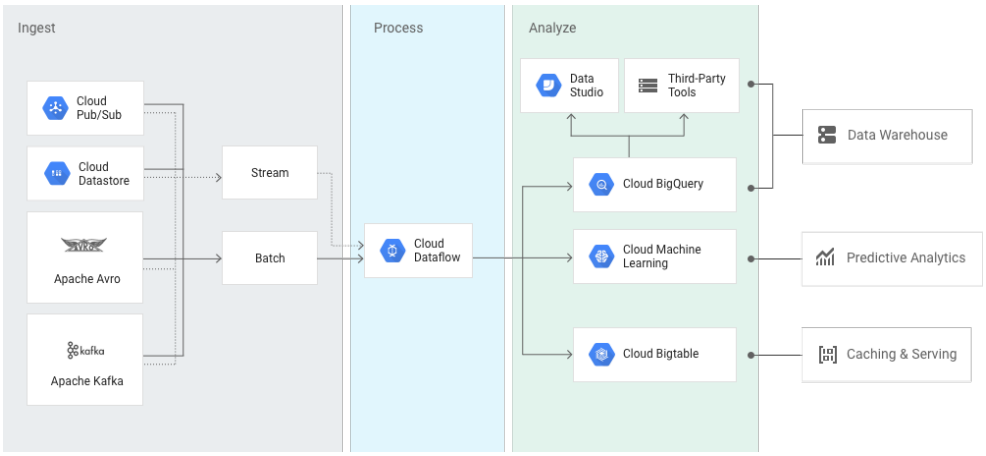
- Appear to execute in a single data center (i.e., no geo-distribution)

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

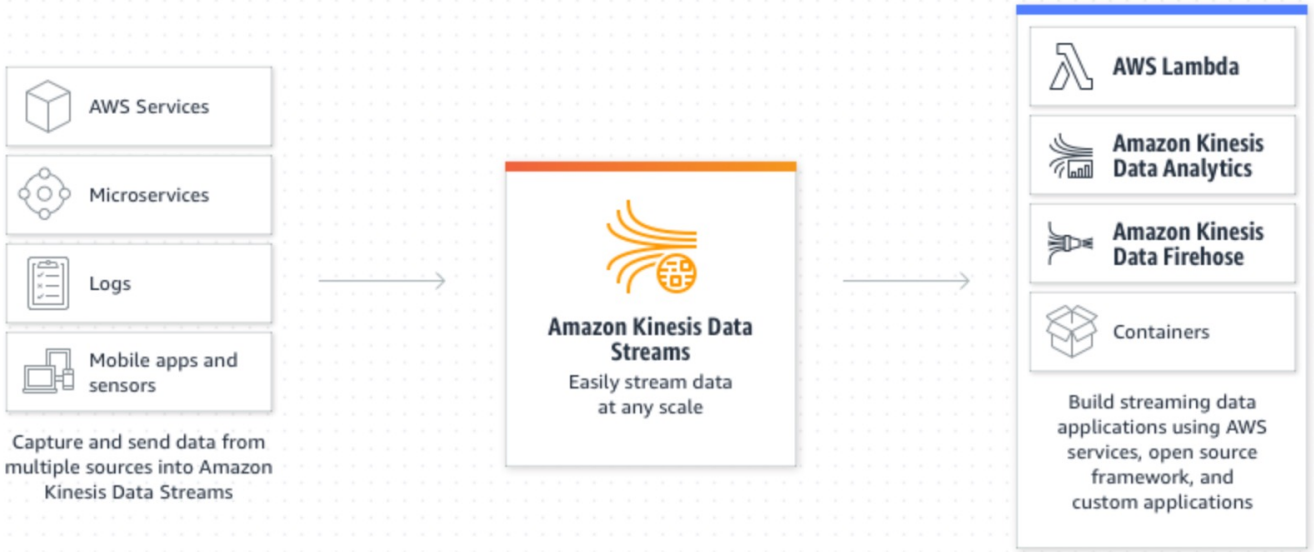
Google Cloud Dataflow

- 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 Analytics

- Serverless, fully managed Apache Flink: allows to process data streams in real time
  - Based on Apache 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 scaling of KPUs

References

- Akidau, Streaming 101: The world beyond batch, 2015.
- Carbone et al., Beyond Analytics: The Evolution of Stream Processing Systems, *ACM SIGMOD '20.*
- Kulkarni et al., Twitter Heron: stream processing at scale, *ACM SIGMOD '15.*
- 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.