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. 2025/26
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

DSP frameworks we consider

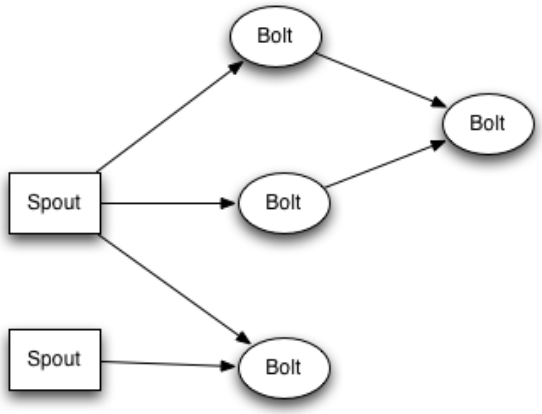
- Apache Storm
- Apache Flink (plus hands-on lesson)
- Apache Spark Streaming (plus hands-on lesson)
- Kafka Streaming (hands-on lesson)
- Cloud-based frameworks
  - Google Cloud Dataflow
  - Amazon Kinesis

Apache Storm

- Open-source, real-time, scalable streaming system https://storm.apache.org/
- Provides a distributed abstraction layer for executing DSP applications
- Supports many use cases: real-time analytics, online ML, continuous computation, distributed RPC, ETL pipelines
- Fast: 1M tuples/sec. per node
- Easy to integrate with different sources (e.g., messaging systems)
- Originally developed by Twitter
- Current version: 2.8

Storm: topology

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

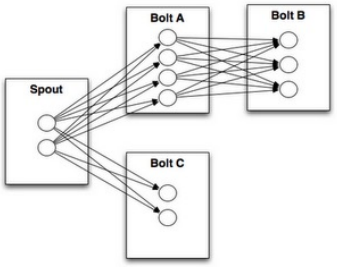


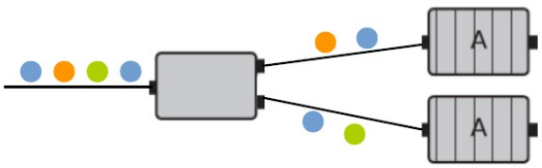
Storm: streams and tuples

- Storm uses streams and tuples as its data model
  - Stream: core abstraction in Storm
    - Unbounded sequence of tuples
    - Storm provides functions for transforming a stream into a new stream in a distributed and reliable way
    - Streams are defined with a schema that names the fields in the stream's tuples
  - Tuple: named list of values
    - Tuple field: 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

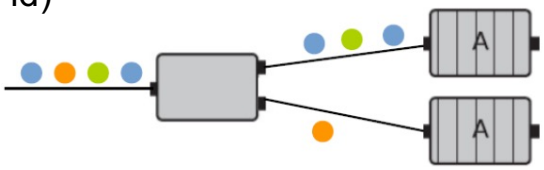
Storm: stream grouping

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



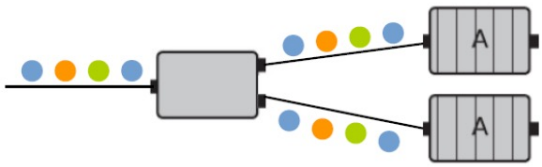


- Field grouping
  - Stream is partitioned by the fields specified in the grouping (e.g., user-id)



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

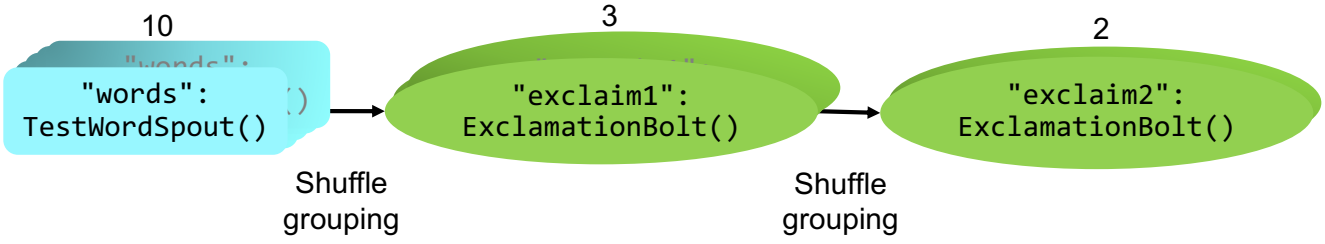
- Example 1: exclamation
  - Spout emits words, each bolt appends "!!!" to its input

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 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 2: 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"));
```

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

- Storm supports both sliding and tumbling windows https://storm.apache.org/releases/current/Windowing.html
- 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 requires windowing support must 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.
```

https://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 supports stateful windowing

Storm: time and out-of-order tuples

- By default, Storm uses processing time
  - Time when an event is actually *processed* by the system
  - In Storm, any timestamp tracked in the window corresponds to the time when the tuple is processed
- Event-time support is limited/user-defined
  - Source-generated timestamps
  - Requires the spout to generate and attach timestamps to emitted tuples
- Out-of-order tuples are handled through
  - Lateness bound (aka *time lag*): specify max time limit for tuples with out-of-order timestamps; by default, late tuples are dropped
  - Watermark: defined as the minimum of the latest tuple timestamps (minus the lag) across all input streams
    - User can change the interval at which watermarks are generated

Storm: Stream API

- Alternative interface: typed API for expressing streaming computations
- Supports a functional style, similar to Spark and Flink
  https://storm.apache.org/releases/current/Stream-API.html
- Provides abstractions such as `Stream` and `PairStream` (key-value pairs)
- Supported operations:
  - Basic transformations, e.g., `filter`, `map`, `flatMap`
  - Windowing
  - Aggregations, e.g., `reduce`, `aggregate`, `reduceByKey`, `countByKey`
  - Joins
  - Output operations (sink), e.g., `print`, `forEach`
  - State management: to save, update and query state

Storm: Stream API example

- WordCount using Stream API

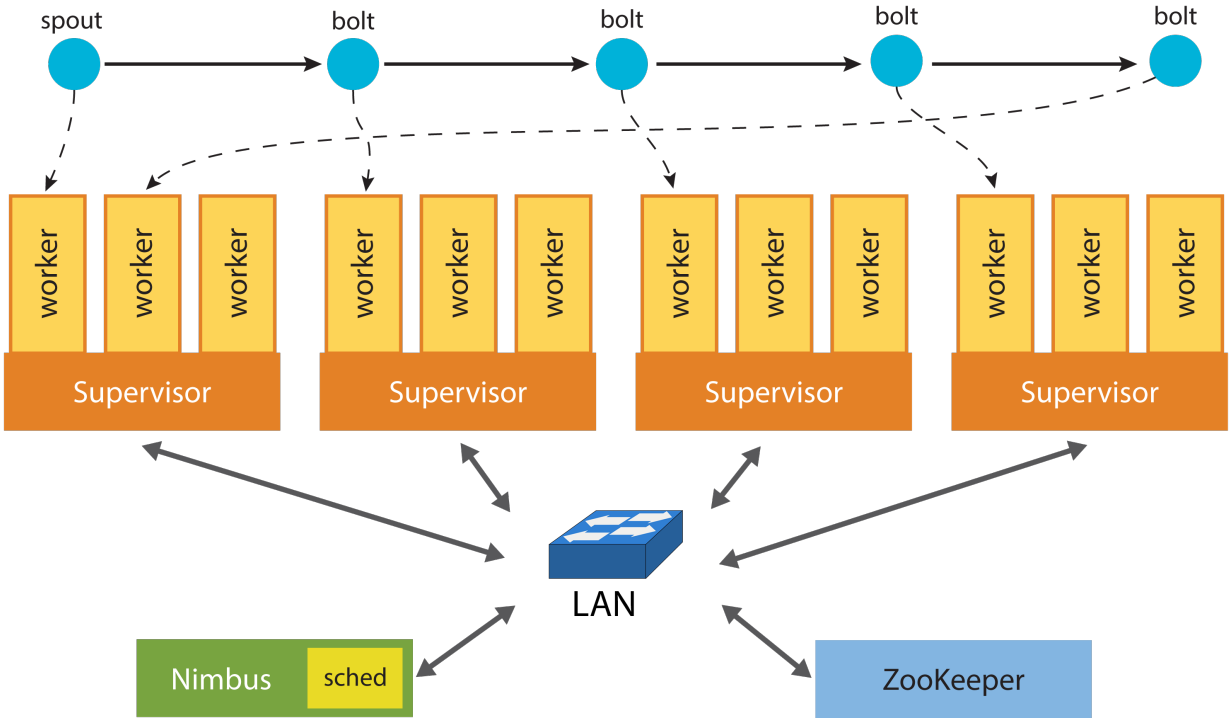
```
StreamBuilder builder = new StreamBuilder();

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();
```

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

Storm: architecture

- Master-worker architecture

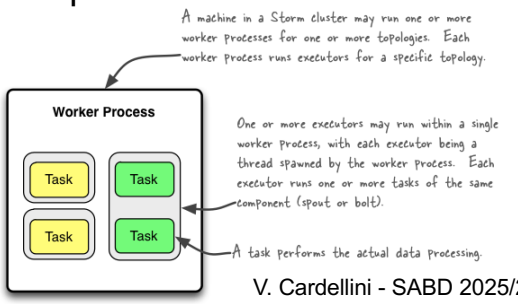


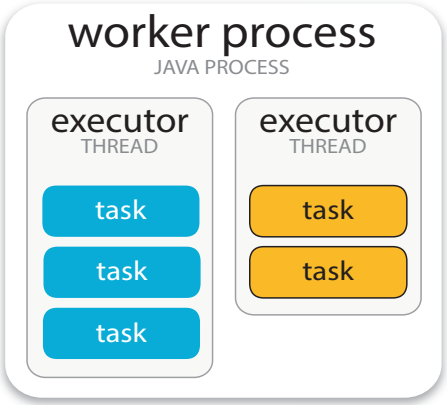
Storm components: master and Zookeeper

- 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, contains one or more worker processes
- Worker process: Java process running one or more executors, belongs to a specific topology
- Executor: thread spawned by a worker process, the smallest schedulable entity
  - May run one or more tasks for the same operator
- Task: operator instance
  - Does the actual work for the operator





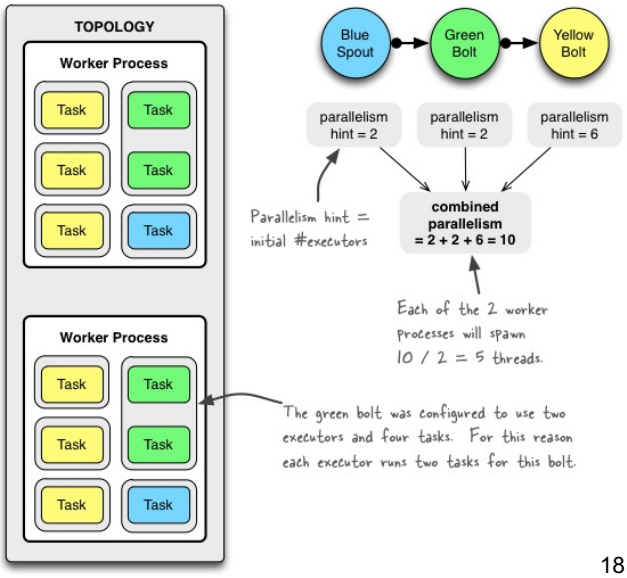
Storm components: supervisor

- 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 a topology

- Developer can configure topology parallelism
  - Number of worker processes: `setNumWorkers` method
  - Number of executors (threads): `parallelism_hint` parameter in `setSpout` or `setBolt`
  - Number of tasks : `setNumTasks` method
- Parallelism of running topology can be *manually* changed using `rebalance` command

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



Storm: reliable message processing

- If a bolt fails to process a tuple, Storm can replay the tuple from the originating spout
  - Storm maintains the relationship between every spout tuple and its tree of tuples using anchoring, so to detect when the tree of tuples has been successfully processed
  - Storm also uses acking: bolts ack tuple appropriately
    - If ack is not received within a timeout period, the tuple processing is considered failed and the tuple is replayed from the spout
- Storm provides at-least-once processing semantics by default https://storm.apache.org/releases/current/Guaranteeing-message-processing.html
  - If acking is disabled, best effort
- Does not natively support exactly-once semantics
  - Trident is a higher-level abstraction to provide it on top of Storm https://storm.apache.org/releases/current/Trident-API-Overview.html

Storm: monitoring and metrics

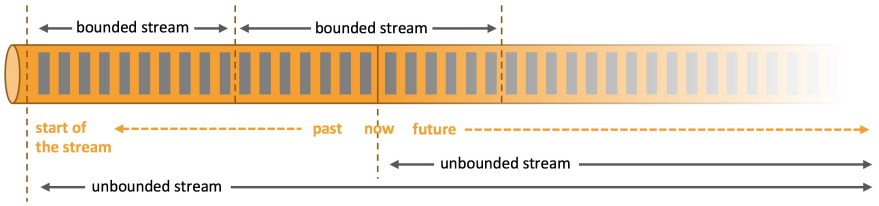
- Storm has a built-in monitoring and metrics system
  - Both built-in and user-defined metrics
- Built-in metrics include:
  - Capacity
    - # of messages executed * average execute latency / time window
  - Latency
    - For spouts: completeLatency (total latency for processing the tuple)
      - 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 accessed via Storm's UI REST API or reported to a registered consumer (e.g., Graphite)

```
"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
},
```

https://storm.apache.org/releases/current/STORM-UI-REST-API.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 them 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

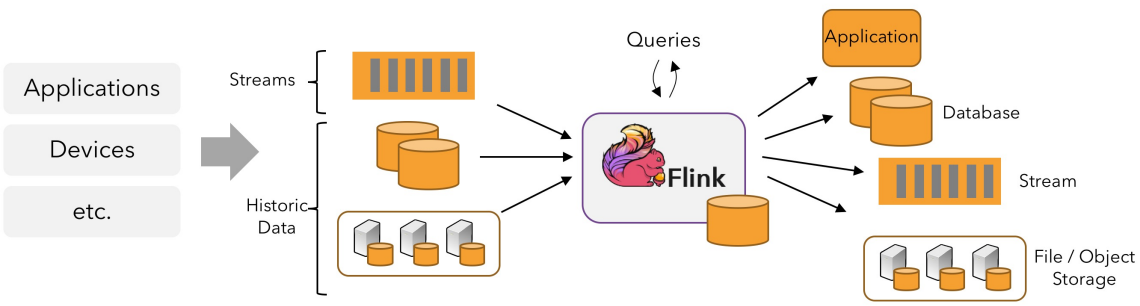
Batch processing vs. stream processing

- 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

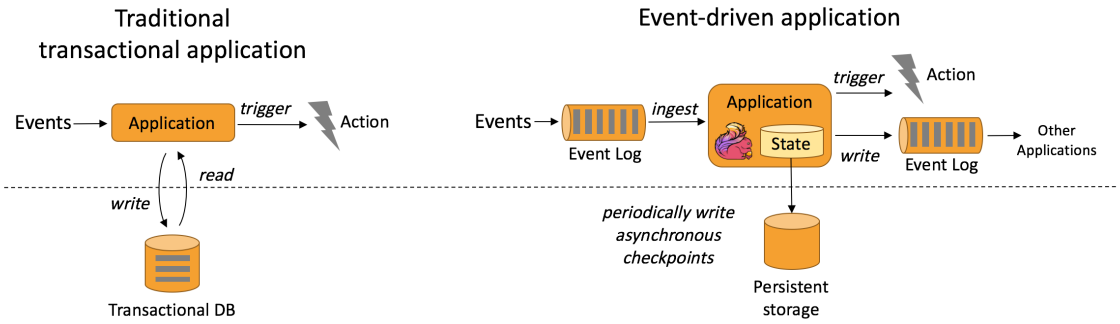
Apache Flink

- Distributed processing system for stateful computation over unbounded and bounded data streams https://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 release: 2.2



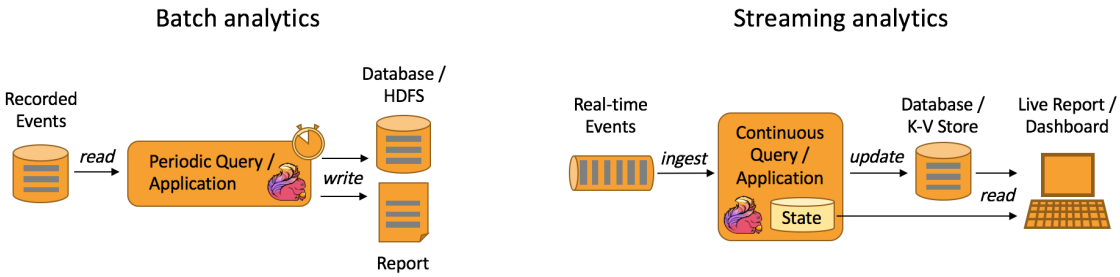
Flink: common use cases

- Event-driven applications
  - Applications that ingest events from one or more event streams and react to incoming events by triggering computations, state updates, or external actions
  - Fraud detection, anomaly detection, rule-based alerting, etc.

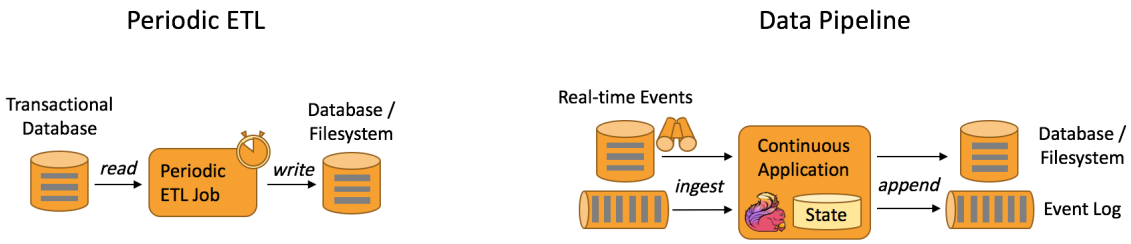


Flink: common use cases

- Data analytics applications
  - Flink supports traditional batch queries on bounded data sets and real-time, continuous queries from unbounded, live data streams

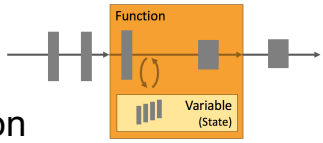


- ETL and data pipelines

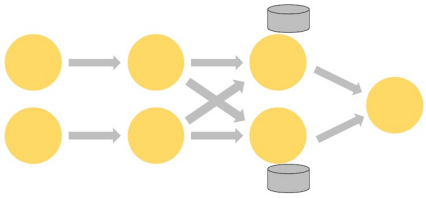


Flink: stateful computation

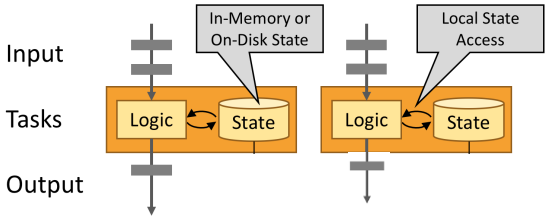
- Flink's operations can be stateful
  - E.g., counting events per minute to display on dashboard, computing features for fraud detection



- State is partitioned: the set of parallel instances of a stateful operator is a sharded key-value store

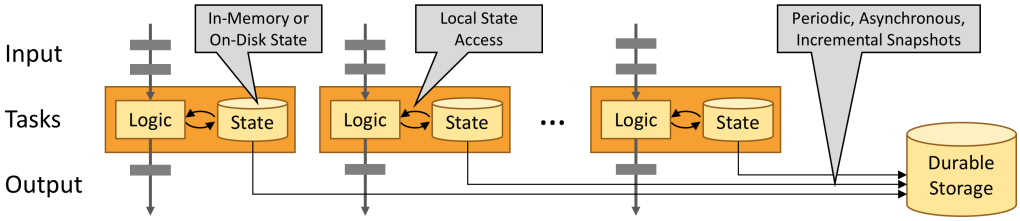


- State is optimized for local access
  - Stored in memory or in access-efficient data structures on disk
  - Goals: high throughput and low latency



Flink: fault tolerance

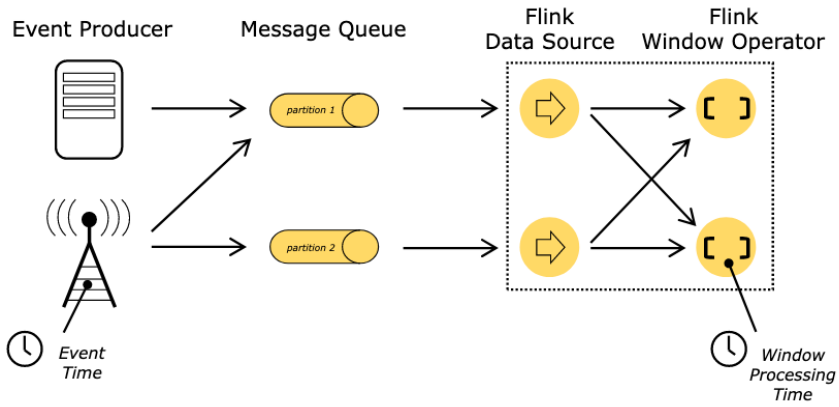
- Flink can guarantee exactly-once state consistency in case of failures by *periodically* and *asynchronously* checkpointing local state to durable storage (state snapshot)
  - State of operators can be restored from checkpoint to an earlier point in time and records are reset to state snapshot



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

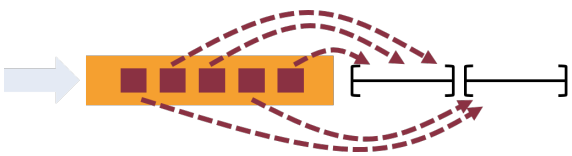
Flink: time

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



Flink: time

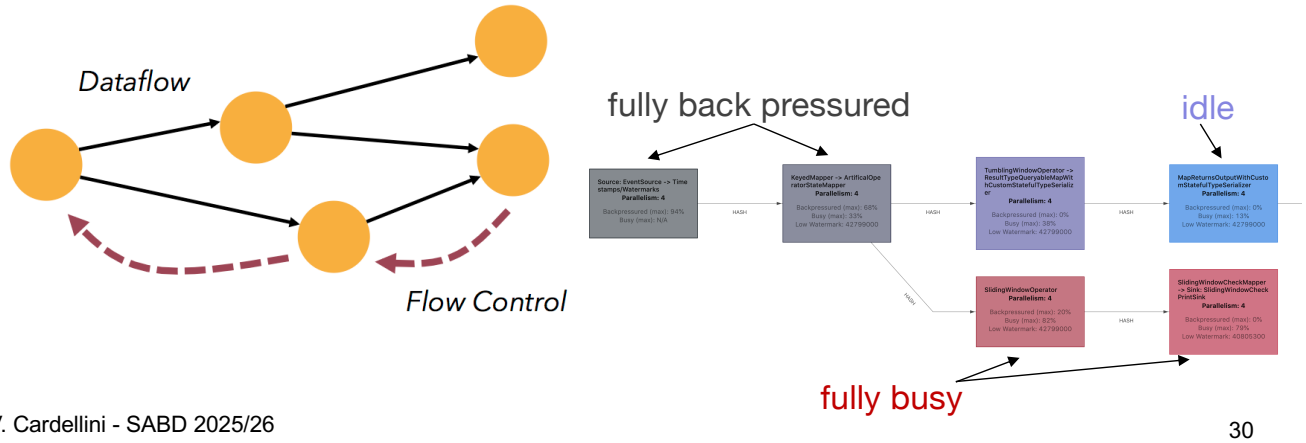
- Flink supports both processing time and event time
- Event time makes it easy to compute over streams where events arrive out-of-order and are late
- To support event time, the DSP system needs to measure the progress of event time: how?
- Flink uses watermarks and lateness bound



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

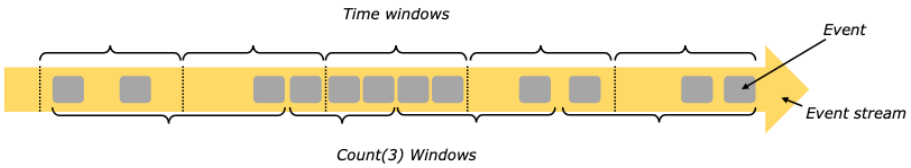
Flink: back pressure

- Continuous streaming with back pressure
  - Flow control mechanism used when data is produced faster than it can be consumed or processed: slow downstream operators backpressure faster upstream operators
  - Flink UI allows to monitor back pressure behavior of running applications
    - Back pressure warning for an upstream operator means it is producing data faster than downstream operators can consume



Flink: windows

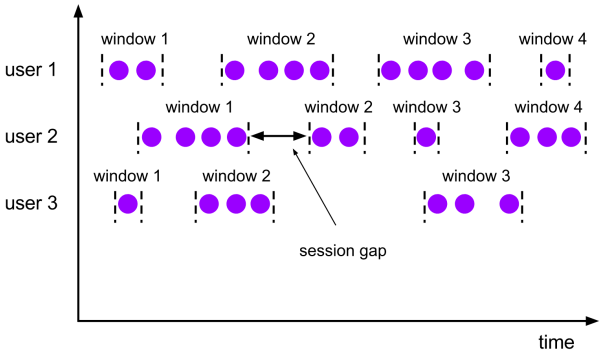
- Highly flexible streaming windows
  - Including user-defined windows



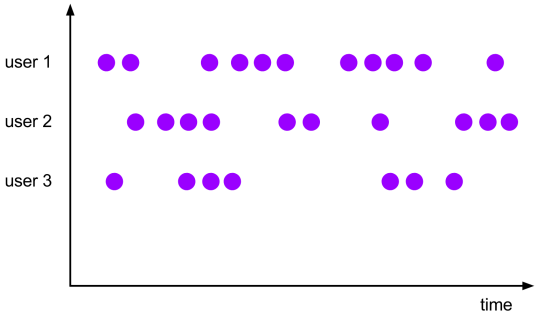
- Supported window types
  - Tumbling: no overlap
  - Sliding: with overlap
  - Session
  - Global

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

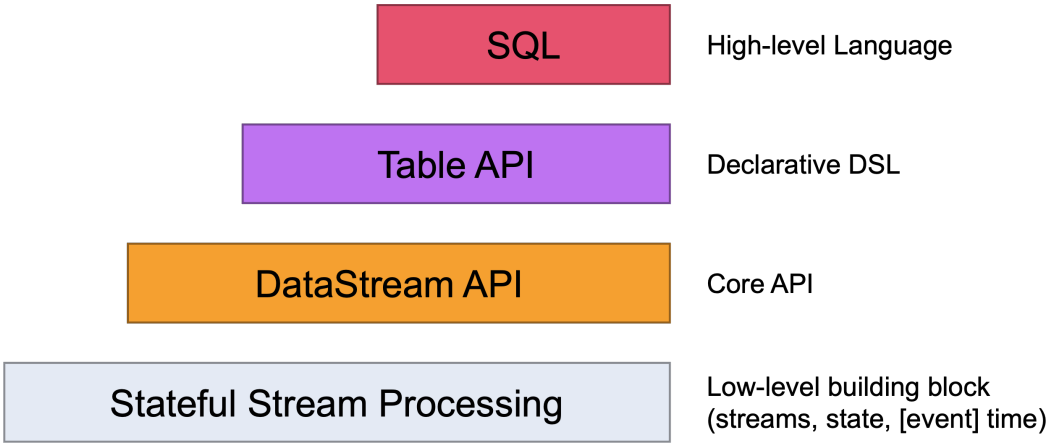


- Global window
  - To assign all elements with the same key to the same single global window
  - Only useful if you also specify a custom trigger



Flink: APIs

- Different levels of abstraction to develop streaming applications
  - Table API & SQL
  - DataStream API
  - ProcessFunction (not properly API)
- APIs in Java, Scala and Python



Flink APIs: Stateful Stream Processing

- Lowest level abstraction, offers stateful and timely stream processing
  - Embedded into DataStream API via `ProcessFunction`
  - Allows users to process events from stream(s), and provides consistent, fault tolerant state; users can also register event time and processing time callbacks
- ProcessFunction: low-level stream processing operation
  - Gives access to basic building blocks of streaming apps
    - Events (stream elements)
    - State (fault-tolerant, consistent, only on keyed stream)
    - Timers (event time and processing time, only on keyed stream)
  - Handles events by being invoked for each event in input stream(s)

https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/operators/process_function/

Flink APIs: DataStream

- Flink's core API for unbounded (and bounded) streams
  - Named from `DataStream` class, which represents an immutable collection of data in a Flink program
  - Data can be unbounded or bounded
  - Initial DataStream is created from source(s) (e.g., message queues, socket streams, files)
  - A new DataStream is derived by means of transformations (provided as API methods) of one or more input DataStreams (e.g., `map`, `filter`, `flatMap`) with user-defined state and flexible windows

See lab lesson

```
DataStream<Tuple2<String, Integer>> dataStream = env
        .socketTextStream("localhost", 9999)
        .flatMap(new Splitter())
        .keyBy(value -> value.f0)
        .window(TumblingProcessingTimeWindows.of(Duration.ofSeconds(5)))
        .sum(1);
```

https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/overview/
https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/operators/overview/

Flink: other APIs

- Table API and SQL
  - Relational APIs for unified stream and batch processing
  - Table API: language-integrated query API for Java, Scala, and Python that allows composition of queries using relational operators (selection, filter, join, …)
  - Flink's SQL support is based on Apache Calcite which implements SQL standard
  - Can combine SQL queries, Table API operations, and DataStream transformations within the same application

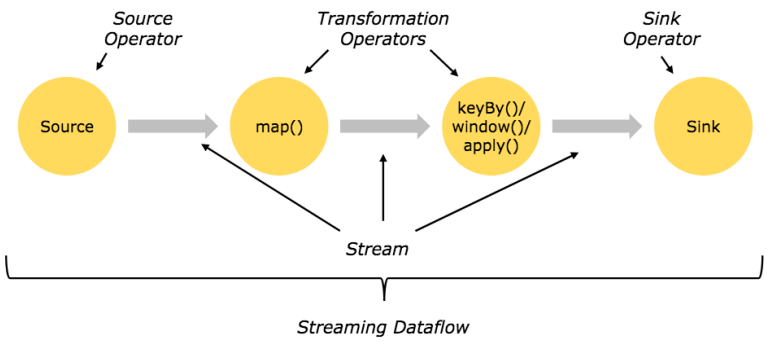
https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/table/overview/

- Python API
  - PyFlink is a Python API for Flink
  - Includes PyFlink DataStream API and PyFlink Table API
  - PyFlink DataStream API provide lower-level control over Flink's core building blocks, state and time

https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/python/overview/

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

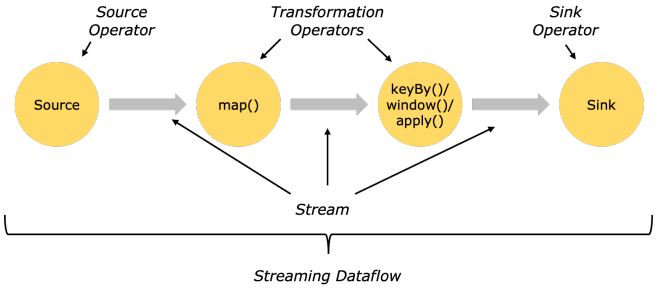
```
DataStream<String> lines = env.addSource(
                           new FlinkKafkaConsumer<>(…));
DataStream<Event> events = lines.map((line) -> parse(line));
DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());
stats.addSink(new BucketingSink(path));
```

Source

Transformation

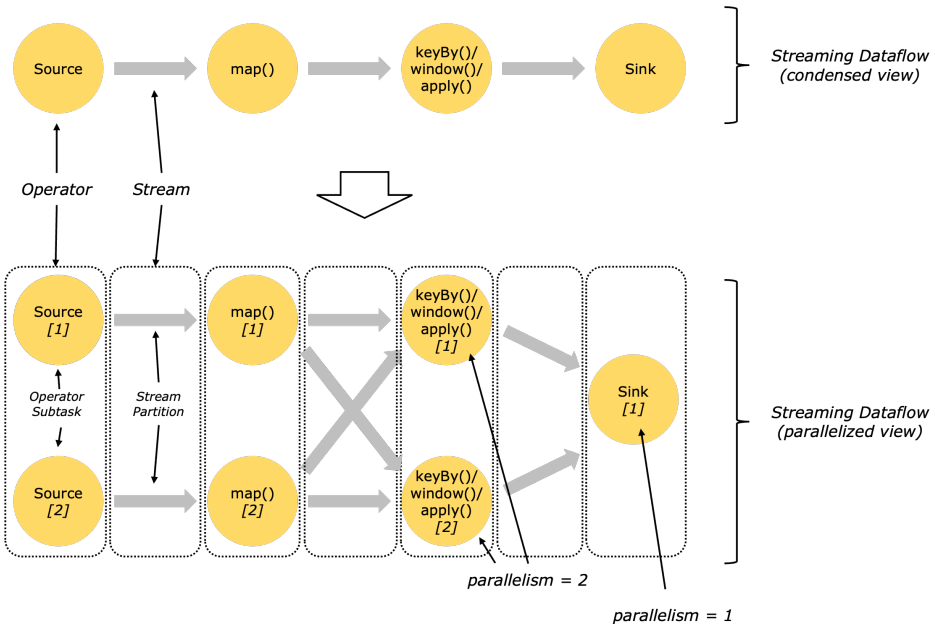
Transformation

Sink



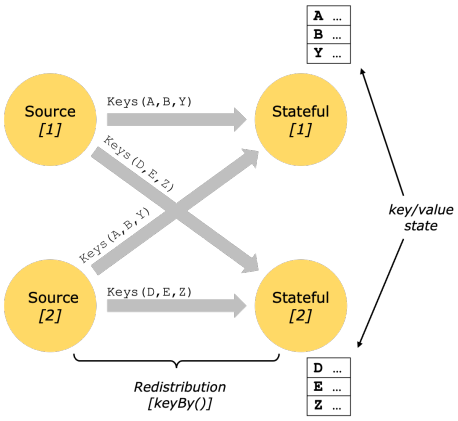
Flink: programming model

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



Flink: programming model

- Stateful operators: require to remember information across multiple events
- Examples:
  - Windowed aggregation: state stores the current aggregates for each window
  - Online machine learning: state stores the current model parameters
- State is maintained in an embedded key/value store
- Access to keyed state is only available on *keyed streams* created by `keyBy()`, which re-partitions tuples by hashing their keys



https://nightlies.apache.org/flink/flink-docs-release-2.2/docs/concepts/stateful-stream-processing/

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, stdin, directories, and sockets, and ingesting data from collections and iterators

  https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/sources/
  - Predefined data sinks support writing to files, to stdout and stderr, and to sockets

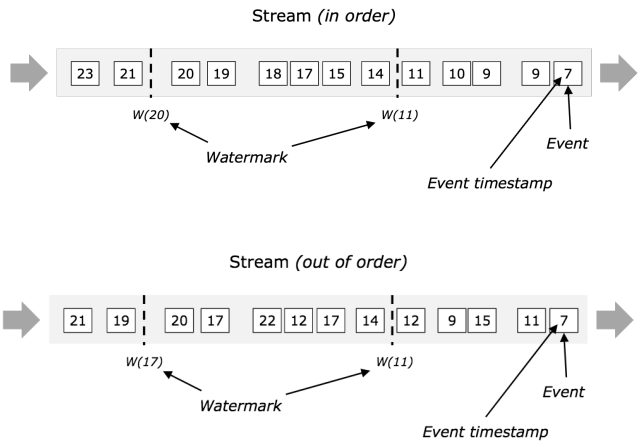
  https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/sinks/
- Connectors provide code for interfacing with various third-party systems
  - Kafka, Rabbit MQ, Pulsar, etc.

Flink: control events

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

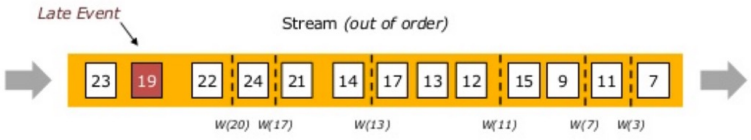
Flink: watermarks

- 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

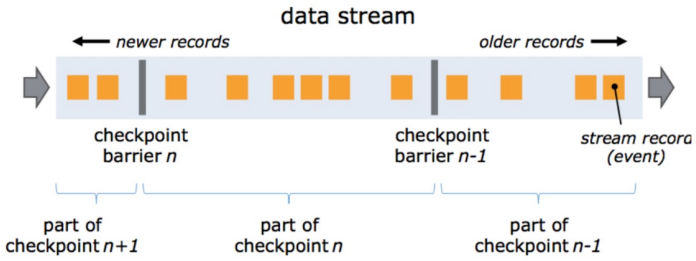


Flink: watermarks

- 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

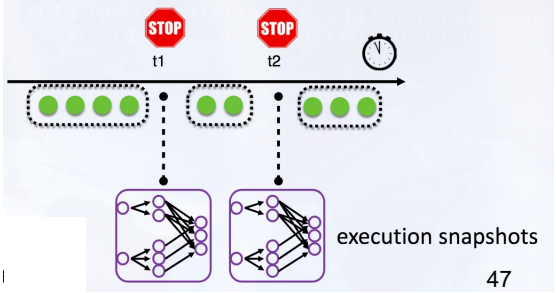
Flink: checkpoint barriers

- 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 (stream flows and operators' state) 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
- Naive approach: rely on periodic global state snapshots, but:
  - X Stalls computation
  - X Eagerly persists tuples in transit along with states, resulting in larger snapshots than required



Flink: snapshot algorithm

- Flink implements a lightweight snapshot algorithm
  - Allows to maintain high throughput while providing strong consistency guarantees
- Properties of snapshot algorithm
  - Draws consistent snapshots of stream flows and operators' state
  - Even in presence of failures, application state reflects every event in data stream exactly once
  - State can be stored in Flink's master (JobManager) heap or file system (including HDFS)
  - 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-stable/docs/concepts/stateful-stream-processing/

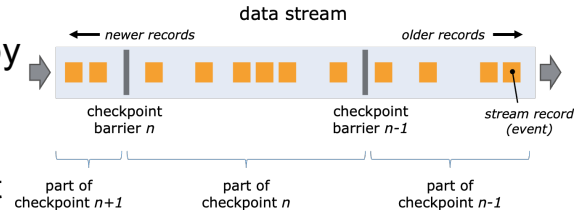
https://arxiv.org/abs/1506.08603

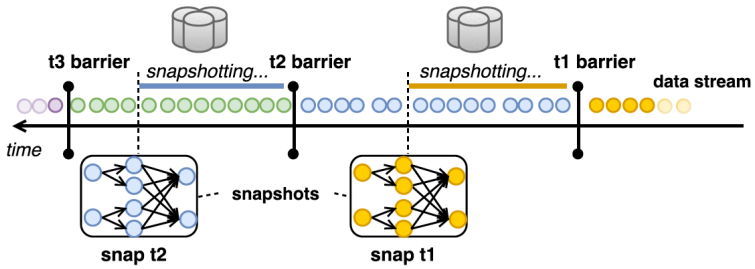
Chandy-Lamport algorithm

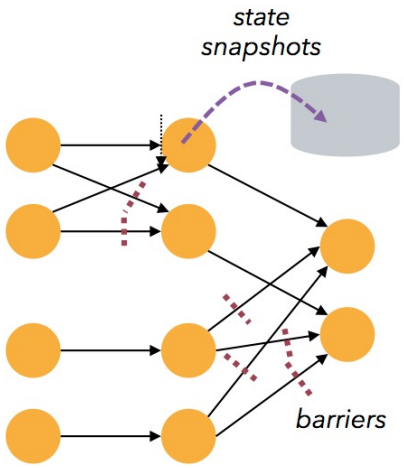
- The observer process (process initiating the snapshot):
  - Saves its own local state
  - Sends a snapshot request message bearing a snapshot token to all other processes
- If a process receives the token *for the first time*:
  - Sends the observer process its own saved state
  - Attaches the snapshot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



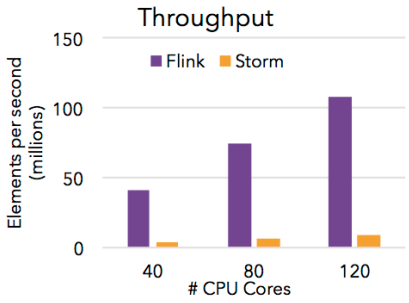


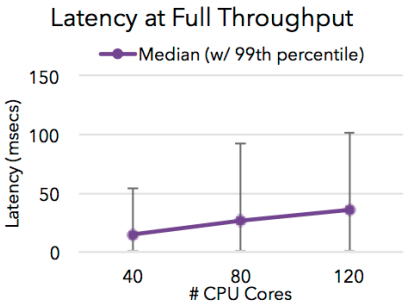


https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/learn-flink/fault_tolerance

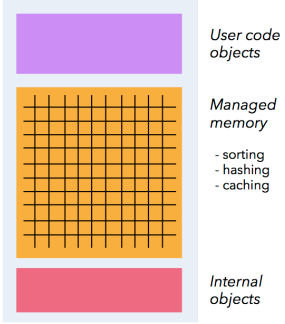
Flink: performance and memory management

- High throughput and low latency



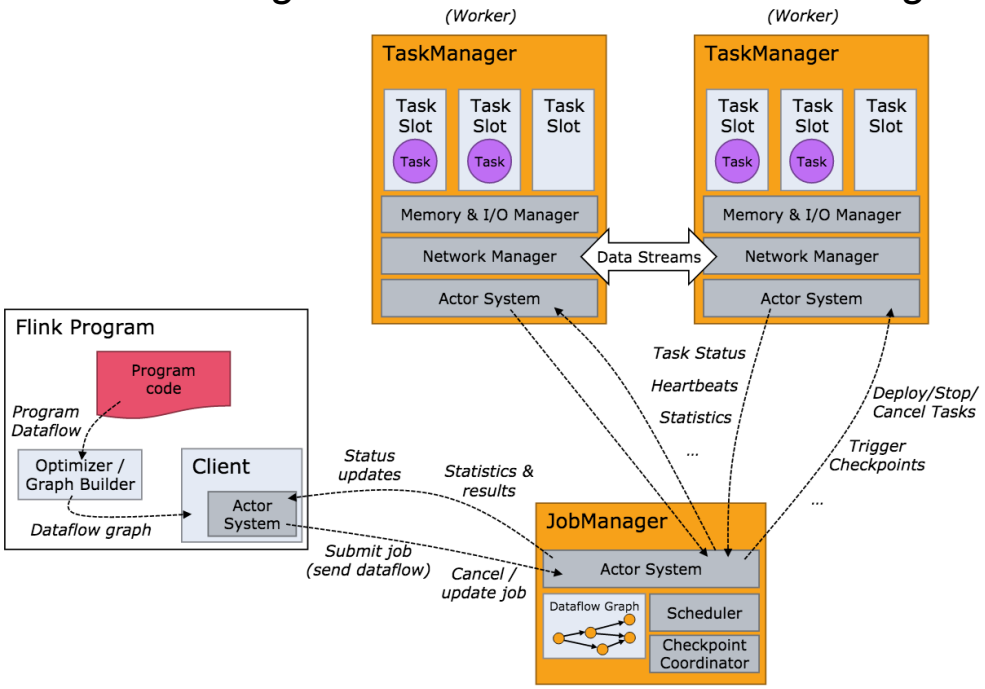


- Memory management
  - Flink implements its own memory management inside JVM



Flink: architecture

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



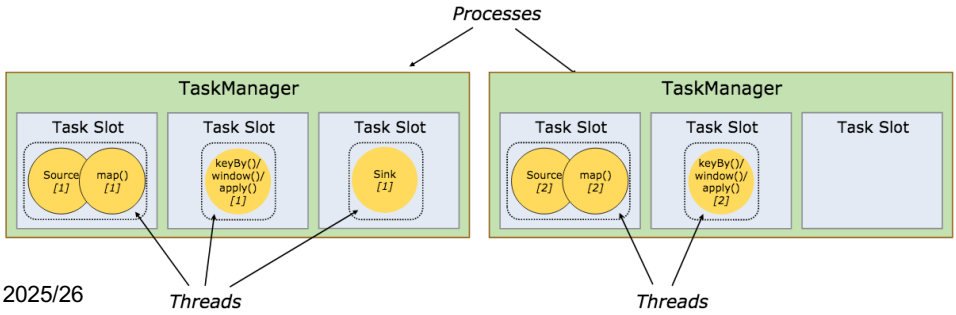
Flink: architecture

- 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

https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/concepts/flink-architecture/

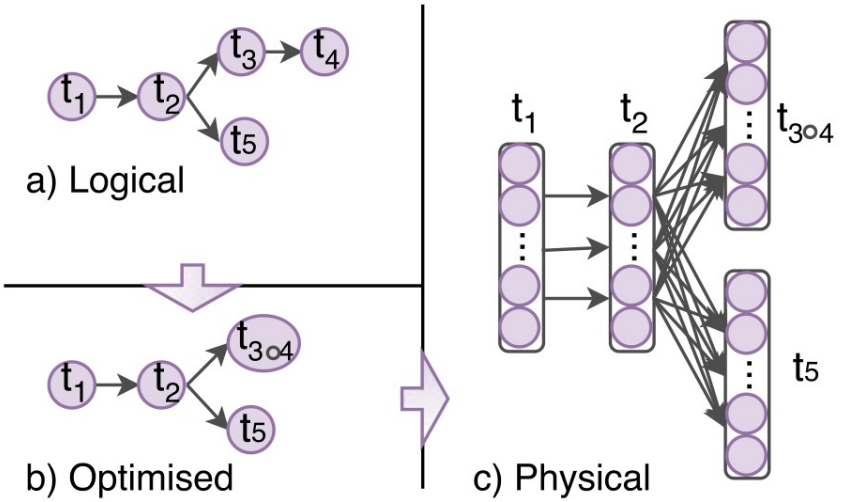
Flink: architecture

- TaskManagers (workers): JVM processes that execute tasks of a dataflow, and buffer and exchange data streams
  - Workers use task slots (threads) to control the number of tasks they accept (at least 1)
  - Each task slot represents a fixed subset of worker resources
    - No CPU isolation, slots only separate memory of tasks
  - By adjusting the number of task slots, user can define how tasks are isolated from each other
    - Tasks in same JVM share TCP connections, heartbeat messages, data sets and data structures



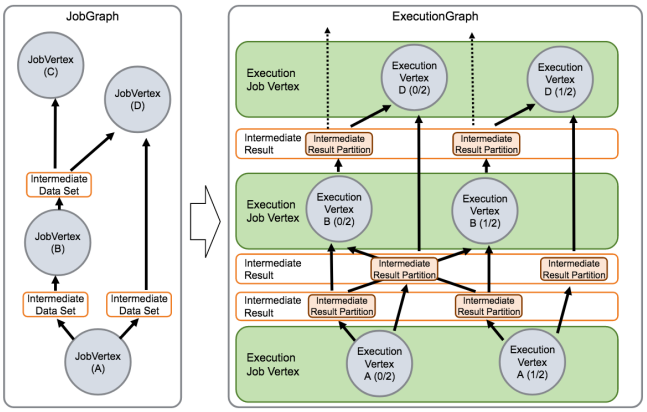
Flink: from logical to physical graph

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



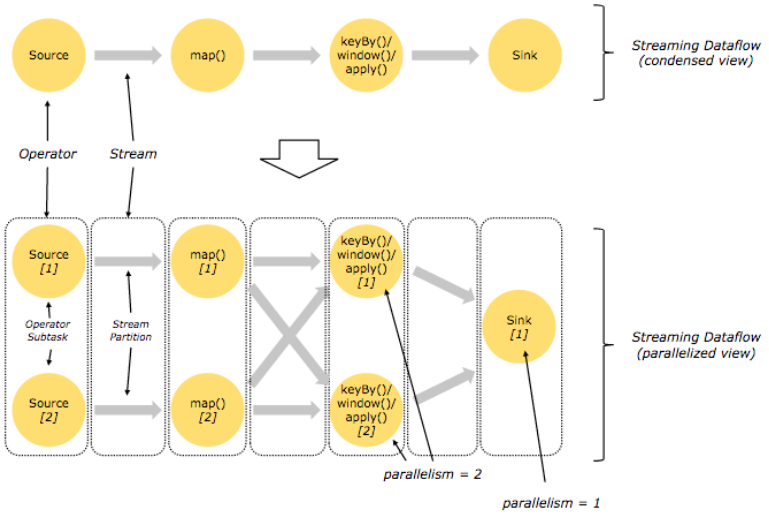
Flink: application execution

- 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



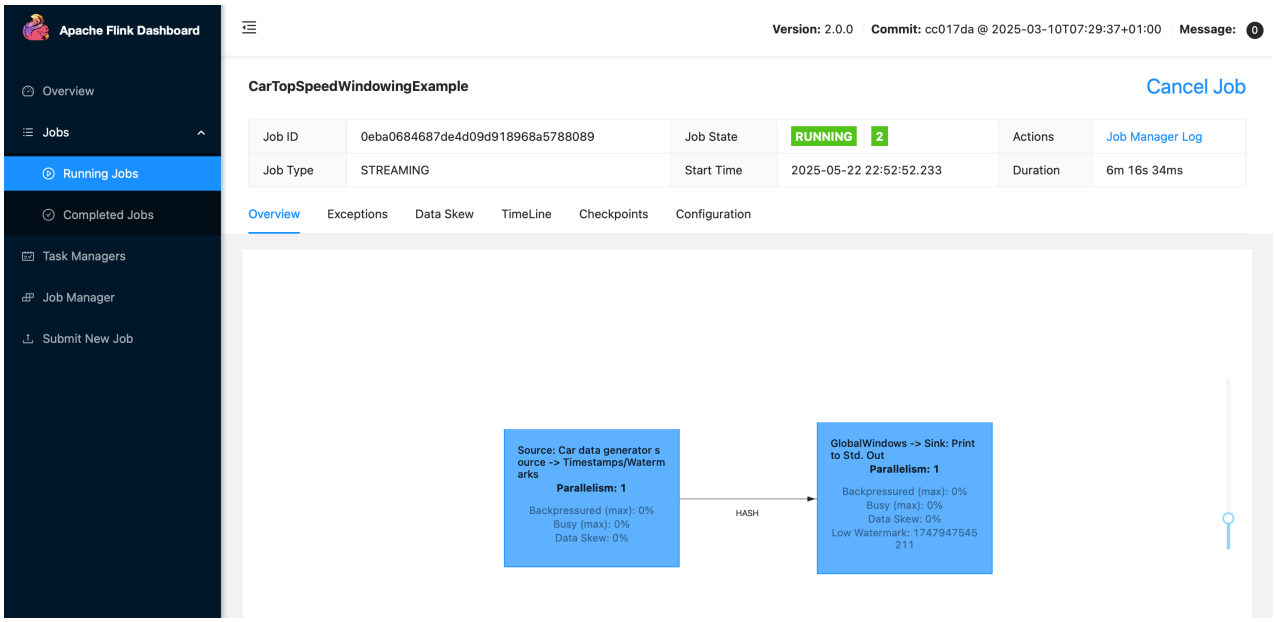
Flink: application execution

- 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 cluster and running jobs
  - Available at localhost:8081



Flink: monitoring and metrics

- Built-in monitoring and metrics system
- Supports exposing metrics to external monitoring systems
- Core components: `Metrics` and `MetricsReporters`
- Built-in metrics include:
  - Throughput
  - Latency: time between event ingestion and when results based on this event become visible
  - Used JVM heap/non-heap/direct memory
  - CPU utilization
  - Availability
  - Backpressure
  - Checkpointing

https://nightlies.apache.org/flink/flink-docs-stable/docs/ops/metrics/

Flink: 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-stable/docs/ops/metrics/#end-to-end-latency-tracking
  - Flink supports end-to-end latency tracking: special events (called LatencyMarker) are periodically inserted at sources to obtain latency distribution between sources and each downstream operator
    - Does not account for time spent in operator processing (or in window buffers)
    - Assumes synchronized clocks across the cluster nodes
    - Disabled by default (can add overhead, useful for debugging): set `latencyTrackingInterval` > 0 to enable

Flink: monitoring

- Application-specific metrics can be added
  - E.g., counter for number of invalid events
- Metrics can be
  - Visualized in Flink's UI dashboard (Metrics tab)
  - Queried through Flink's Monitoring REST API, that accepts HTTP requests and responds with JSON data

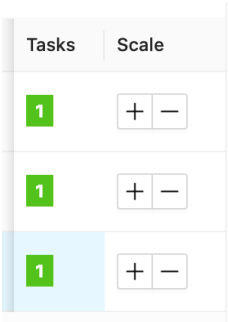
https://nightlies.apache.org/flink/flink-docs-stable/docs/ops/rest_api/
  - Exported to external systems (Graphite, InfluxDB, Prometheus)

https://nightlies.apache.org/flink/flink-docs-stable/docs/deployment/metric_reporters/
- Back pressure: to monitor back pressure behavior of running jobs
  - E.g., back pressure warning (e.g. `High`) for a task means that it produces data faster than downstream operators can consume

https://nightlies.apache.org/flink/flink-docs-stable/docs/ops/monitoring/back_pressure/
- Checkpointing: to monitor job checkpointing

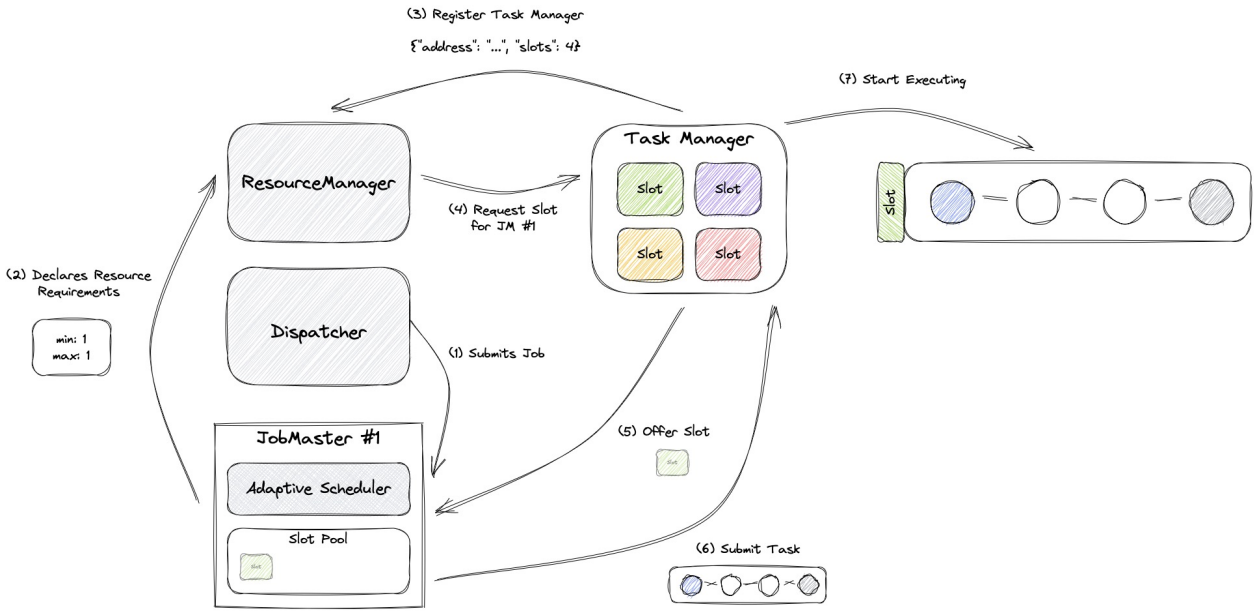
Flink: elastic scaling

- Flink can adjust job's parallelism at runtime based on available task slots
- AdaptiveScheduler: adapts to changing resources and evolving workload requirements
  - Scales in if resources are insufficient
  - Scales out when new slots become available (up to a configured maximum)
- Not fully elastic: manual intervention is required
  - Changes in resource requirements (e.g., higher input rate) require manual updates
  - Adjustments can be done via Web UI



Flink: elastic scaling

- How it works
  - JobMaster declares its desired resources to ResourceManager, which tries to fulfill those resources



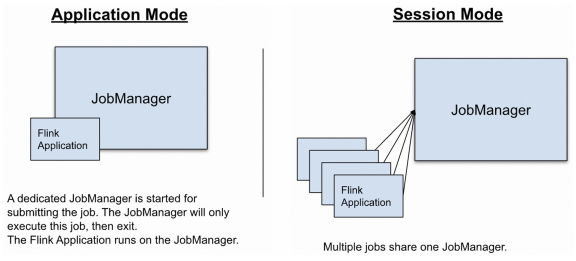
Flink: Complex Event Processing

- FlinkCEP: library for Complex Event Processing (CEP) on top of Flink https://nightlies.apache.org/flink/flink-docs-stable/docs/libs/cep/
- Goal: detect complex event patterns in stream of events
  - CEP continuously matches incoming events against defined patterns to provide insights into what is happening, enabling proactive and effective actions
  - Examples
    - Maritime domain: detect illegal activities at sea based on vessel position streams
    - Financial sector: detect sequences of increasing or decreasing trend events in stock trade time series

```
Pattern<Event, ?> pattern = Pattern.<Event>begin("start")
    .next("middle")
    .where(SimpleCondition.of(value -> value.getName().equals("error")))
    .followedBy("end")
    .where(SimpleCondition.of(value -> value.getName().equals("critical")))
    .within(Duration.ofSeconds(10));
```


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
- Deployment modes
  - Application mode: creates a Flink's cluster per submitted application; application's main() is run by JobManager, application jars are bundled with Flink
  - Session mode: assumes a running cluster and uses cluster resources to execute submitted applications; applications share cluster resources




```
$ ./bin/flink run ./examples/streaming/TopSpeedWindowing.jar
```

Flink: deployment

- Docker images
  - Official image: https://hub.docker.com/_/flink
  - By Flink developers: https://hub.docker.com/r/apache/flink
- Can use Docker images to deploy Session or Application cluster on Docker
- Options to configure Flink cluster
  - Flink configuration file, dynamic properties, environment variables
- Can use Docker Compose
  - Once Flink cluster is running, access JobManager container to launch application

```
$ docker exec -it $(docker ps --filter name=jobmanager
  \ --format={{.ID}}) /bin/bash
```

https://nightlies.apache.org/flink/flink-docs-stable/docs/deployment/resource-providers/standalone/docker/

Delivery guarantees

- Some frameworks offer at-least-once semantics
  - Each tuple is processed, but it may get processed more than once
    - Nothing is lost, but might be duplicated
  - How? Retry delivery until acked (recall RR1 mechanism)
  - E.g., Storm, Flink (depending on configuration)
- To avoid data loss, also source(s) should be *replayable*
  - If DSP system fails before the tuple could be persisted or processed, source provides the tuple again
  - E.g., Kafka is a replayable source: why?
- For stateful non-idempotent operators, at-least-once guarantee can give incorrect results

Delivery guarantees

- Some frameworks offer exactly-once semantics
  - Each tuple is processed once and only once
  - E.g., Storm+Trident, Flink (set checkpointing mode to exactly-once), Spark Structured Streaming, Google's MillWheel
- What is needed?
  - Reliable operators: each operator tracks progress and persists state in fault-tolerant storage
    - Checkpointing for recovery after failure
    - Write-ahead logs to ensure atomicity and durability of ops
  - Replayable sources
  - Idempotent (or transactional) sinks: ensure each tuple affects sink exactly once
- Exactly-once in Flink
  - Every event will *affect the state* exactly once (does not mean that every event will be processed exactly once)

Exactly-once in MillWheel

- MillWheel: low-latency, fault-tolerant DSP framework developed internally by Google to support their real-time data processing needs
- How exactly-once works in MillWheel:
  - Receive tuple for processing
  - Check for duplicates, discard if already processed
  - Run user-defined logic, may produce updates to timers, state, or output tuples
  - Commit pending changes
  - Write updates to persistent storage
  - Acknowledge sender
  - Emit downstream results
  - Optimization: these operations can be coalesced into a single checkpoint for multiple tuples

Akidau et al., MillWheel: Fault-Tolerant Stream Processing at Internet Scale, VLDB 2013
https://research.google/pubs/millwheel-fault-tolerant-stream-processing-at-internet-scale

Comparing DSP frameworks

- Apache Flink vs. Apache Storm

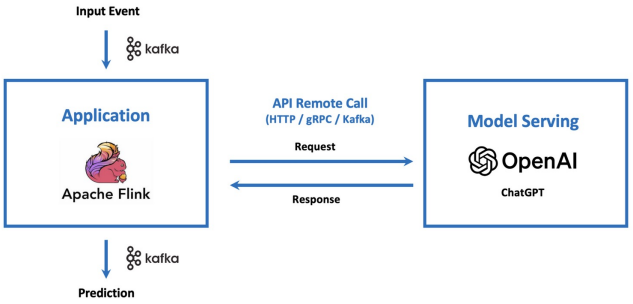
	API	*Windowing*	*Delivery semantics*	*Fault tolerance*	*State mgmt.*	*Flow control*	*Elastic scaling*
Storm	Low-level High-level SQL Only stream	Yes, basic	At-least-once Exactly-once with Trident	Tuple acking, Checkpoint. (similar to Flink)	Limited Yes with Trident	Back pressure	No
Flink	High-level SQL Also batch	Yes, also used-def.	At-least-once Exactly-once	Checkpoint. and state recovery	Yes	Back pressure	Partially

New use case: real-time model inference

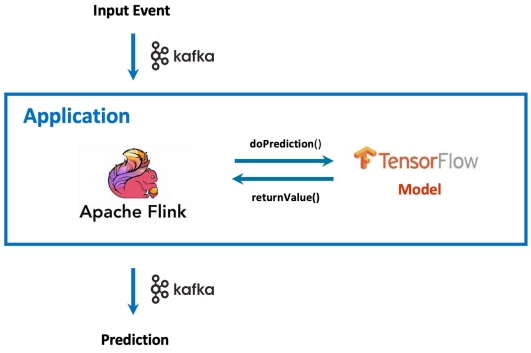
- Model inference
  - Generating predictions using a trained model
  - Involves feeding input data into model to get output (e.g., classification, regression value)
  - In real-time, e.g., for fraud detection in financial services
- Delivery approaches to model inference
  - Remote model inference
    - Model runs on a separate model server
    - Accessed via RPC, API, or HTTP
    - DSP framework handles real-time data streaming, preprocessing, data curation, and postprocessing validation
    - Scalable but may introduce latency
  - Embedded model inference
    - Model is embedded within stream processing application
    - Enables low-latency predictions
    - Requires more resources on processing nodes

New use case: real-time model inference

- Remote model inference using Flink: data is fed into OpenAI API via a Flink SQL UDF to generate context-specific responses using ChatGPT (e.g., ticket rebooking)



- Embedded model inference using Flink: Flink processes data in real-time, embedding a TensorFlow model for immediate model inference
  - How? If Flink job is in Python, you can import TensorFlow, load and run a pre-trained model inside a Flink operator (e.g., MapFunction, ProcessFunction)



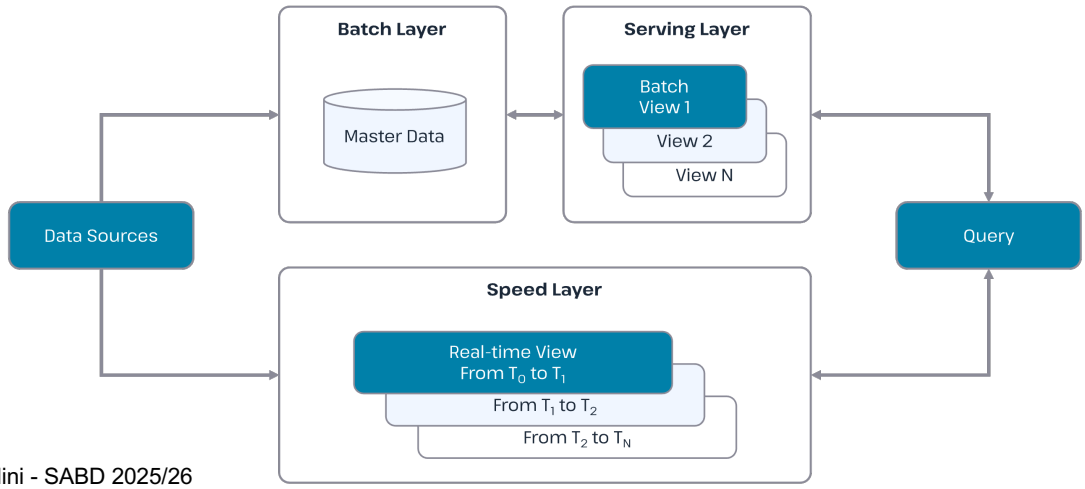
https://www.kai-waehner.de/blog/2024/10/01/real-time-model-inference-with-apache-kafka-and-flink-for-predictive-ai-and-genai

Integrating batch and stream processing

- Common and growing need for companies
  - Run both batch and stream processing
- Options
  1. Lambda architecture: run them *together* in separate layers
  2. Unified frameworks: run them *separately* but in same framework
  3. Unified programming model

Lambda architecture

- Data processing architecture 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 batch and stream layers are then merged



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 source code
- Spark and Flink follow this trend

Unified programming model: Apache Beam

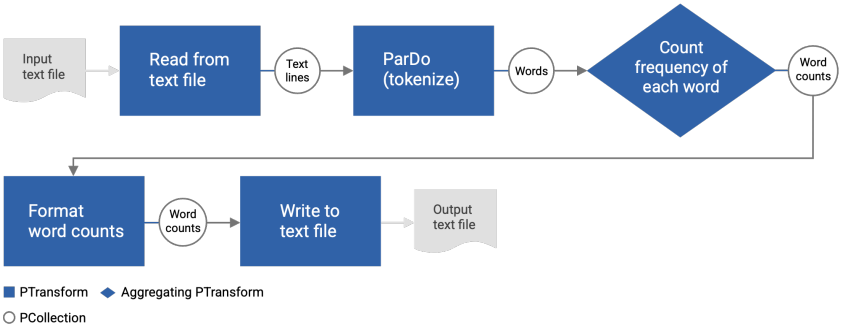
- Layer of abstraction on top of batch and stream processing frameworks https://beam.apache.org/
- Provides a unified programming model
  - Allows to define batch and streaming data processing pipelines that run on supported execution engines, including Flink, Spark, Google Cloud Dataflow
    - Write once, run anywhere
  - Programming languages: Java, Python, Go
- Engine-specific runners translate Beam code to target runtime
- Initially developed by Google, then Apache project

Beam: key concepts

- Define the pipeline
  - `PCollection`: represents a collection of data, which can be bounded or unbounded
  - `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
- Select the execution engine
- Run the pipeline

Beam: key concepts

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



Using Beam: WordCount in Python

```
pipeline = beam.Pipeline(options=pipeline_options)

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

counts = (
    lines
    | 'Split' >> (beam.ParDo(WordExtractingDoFn()).with_output_types(str))
    | 'PairWithOne' >> beam.Map(lambda x: (x, 1))
    | 'GroupAndSum' >> beam.CombinePerKey(sum))

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

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

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

# Execute the pipeline and return the result.
result = pipeline.run()
result.wait_until_finish()
return result
```

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

Beam: pros and cons

- ✓ Portability: single, unified programming model
- ✓ Flexibility: switch underlying processing engine with low effort
- ✓ Integration: pipelines can be embedded into external workflow systems (e.g., Apache Airflow orchestrating Apache Beam pipelines)
- X Potential performance overhead

DSP in the Cloud

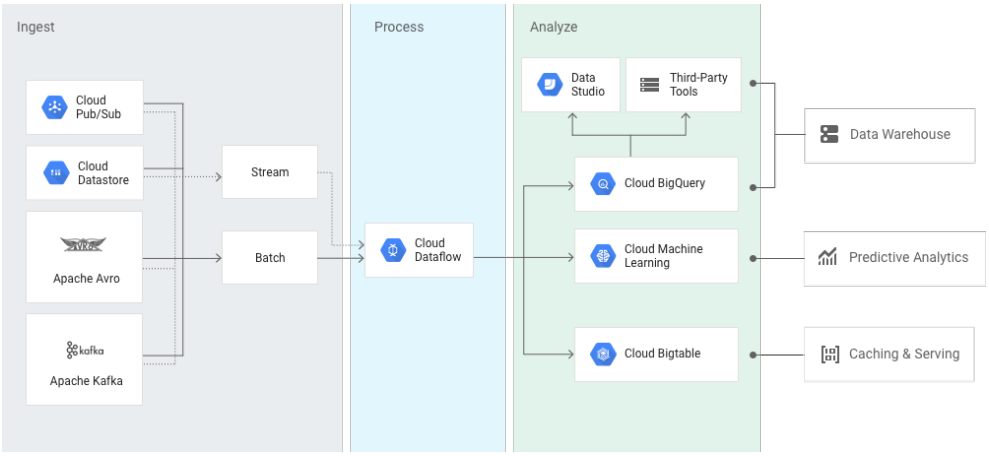
- Data streaming systems as Cloud services
  - Amazon Kinesis Data Streams
  - Google Cloud Dataflow
  - Microsoft Azure Stream Analytics
- Infrastructure is fully abstracted
- Support automatic scaling of compute resources
- Often combined with serverless or fully managed execution models
- Typically execute within a single cloud region, distributed across multiple availability zones, with no automatic geo-distribution

Google Cloud Dataflow 

- Fully-managed data processing service supporting both streaming and batch workloads
  - Automated resource management
  - Dynamic work rebalancing
  - Horizontal auto-scaling
- Provides a unified programming model based on Apache Beam
  - SDKs available in Java and Python
  - Enable developers to implement custom processing logic and run pipelines on different execution engines
- Provides strong consistency guarantees, including exactly-once processing
  - Based on Google's internal system MillWheel

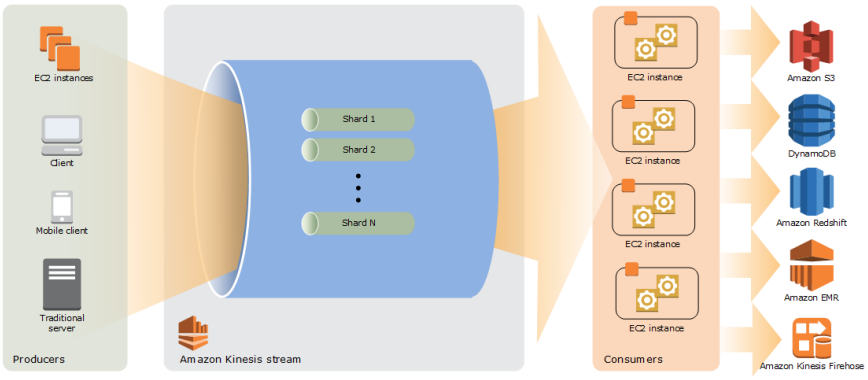
Google Cloud Dataflow

- 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 collection and ingestion of streaming data at scale for real-time analytics
  https://aws.amazon.com/it/kinesis/data-streams
- Fully managed service with automatic provisioning and scaling
- High-level architecture
  - Kinesis data stream is composed of a set of shards; each shard is an ordered sequence of data records; each data record has a sequence number



Flink in the Cloud

- Amazon Managed Service for Apache Flink
  - Fully managed Flink on AWS
  - Uses standard Flink APIs and operators
  - Pricing is based on Kinesis Processing Units (KPUs)
    - Each KPU provides a fixed amount of compute, memory, and networking capacity
  - Supports horizontal scaling by adjusting the number of KPUs
    - Scaling can be automatic based on workload
- Ververica https://www.ververica.com/

References

- Akidau, Streaming 101: The world beyond batch, 2015 https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101
- Carbone et al., Apache Flink: Stream and batch processing in a single engine*, Bulletin IEEE Comp. Soc. Tech. Comm. on Data Eng.,* 2015 https://www.diva-portal.org/smash/get/diva2:1059537/FULLTEXT01.pdf
- Carbone et al., State management in Apache Flink®: consistent stateful distributed stream processing, *Proc. VLDB Endowment*, 2017 https://www.vldb.org/pvldb/vol10/p1718-carbone.pdf
- Carbone et al., A survey on the evolution of stream processing systems, *VLDB Journal*, 2024 https://link.springer.com/content/pdf/10.1007/s00778-023-00819-8.pdf