# 01: Introduction to Flink

CONFLUENT
**Global Education**

# Module Overview

This module contains four lessons:

- Batch Processing vs. Stream Processing
- What is Apache Flink?
- Apache Flink's APIs
- Flink Job & Topology

# 01a: Origin of Stream Processing

## Description

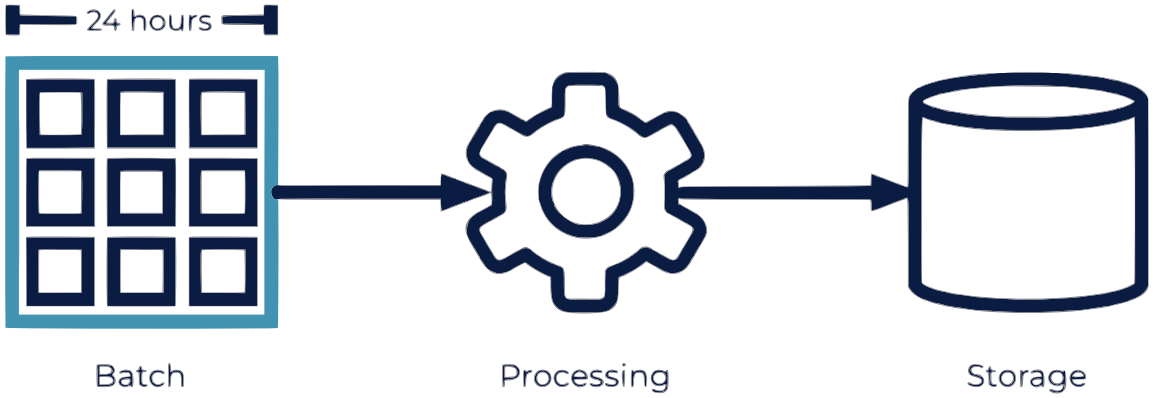
Natural Evolution from Batch Processing to Stream Processing



# Batch Jobs

In the past, when the volume of data was smaller, periodic batch jobs were used to process data once a day




---

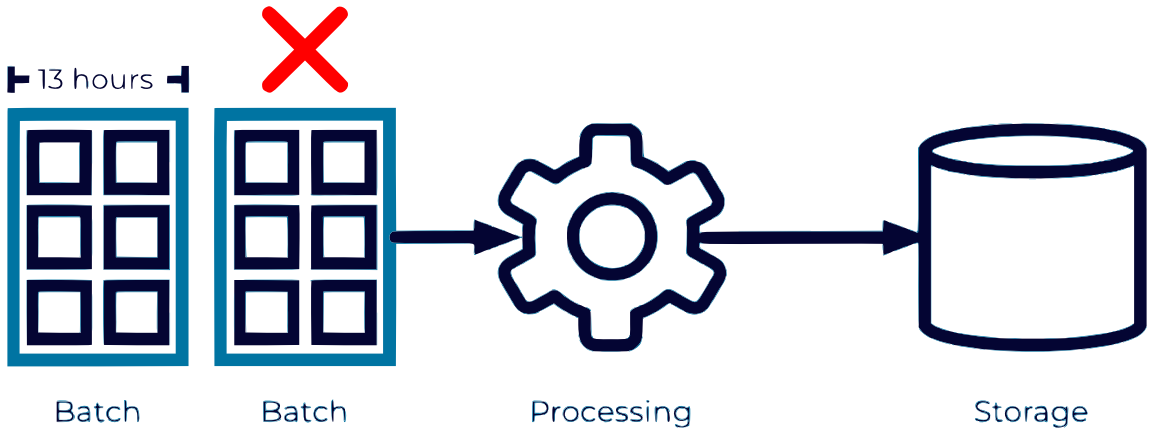
In the past, data was often processed by periodic batch jobs. Initially, those jobs might have run once a day, or even once a week.



# Batch Jobs

What if processing a full day of data takes 13 hours and fails in the 12th hour?



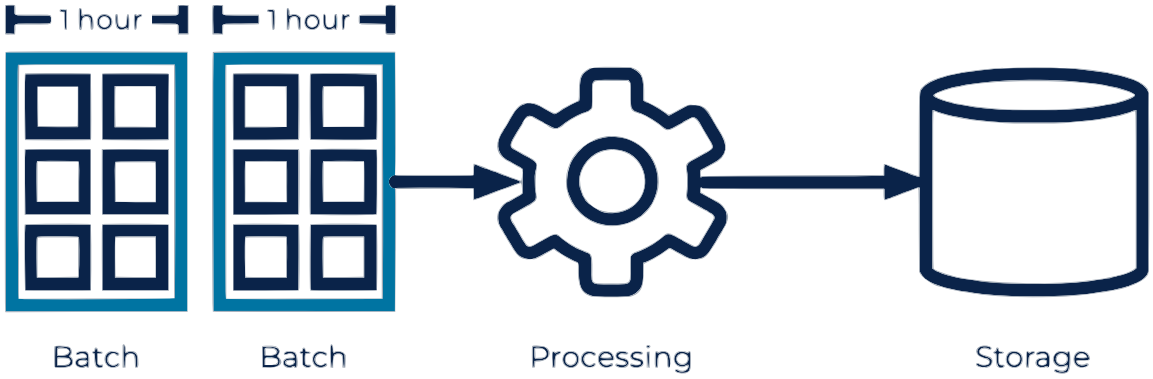

Imagine if processing a full day of data took 13 hours. That means that if you fail on the 12th hour of processing, there's not enough time left in the day to repeat all the work.



# Batch Jobs

Mitigating expensive failures resulted in smaller batches



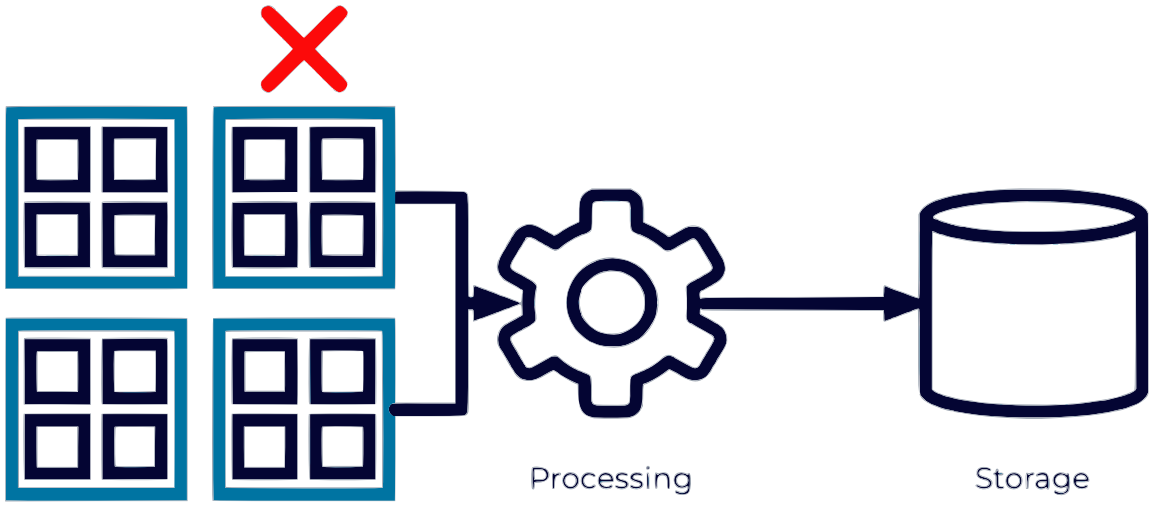

But, as the amount of data increased, the time intervals needed to shrink to avoid costly failures.



# Batch Jobs

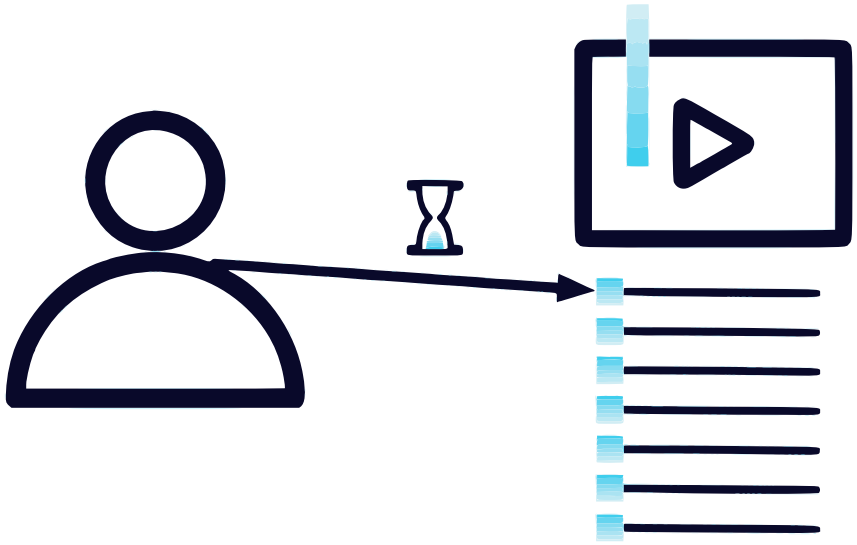
Smaller batches mitigate the impact of those failures




By shortening the cycle, and reducing the batch size, we can reduce the impact of the failures. Rather than having to reprocess a full day, maybe we only need to redo an hour. That leaves us plenty of breathing room to process the rest.

# User Expectations - Real Time

Modern users expect instant results



A user posts a comment on a video

---

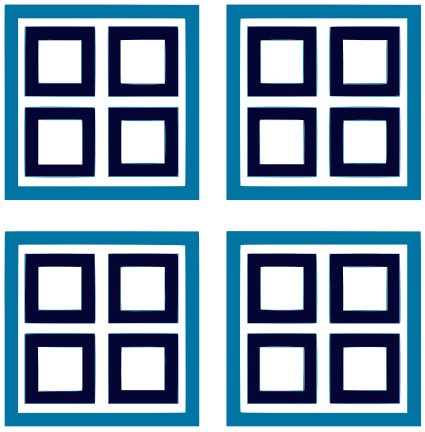


Even if the cycle times are shortened to hours or even minutes, are still too long.

Modern users have been conditioned to expect instant results. If they log in to Facebook or TikTok and comment on a video, they don't want to wait an hour for a batch process to pick up their comment. They want to see the results immediately.

# Micro-Batching

We can keep reducing batch sizes



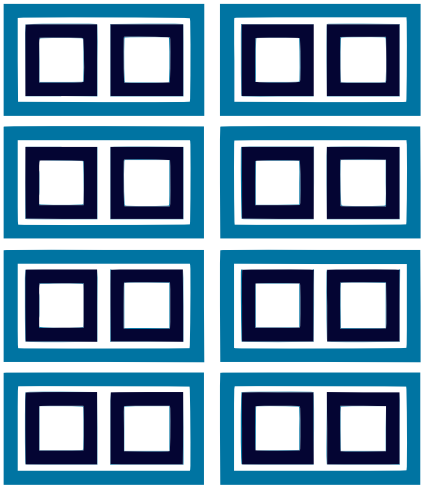
Of course, we could shrink our batch sizes even further.



# Micro-Batching

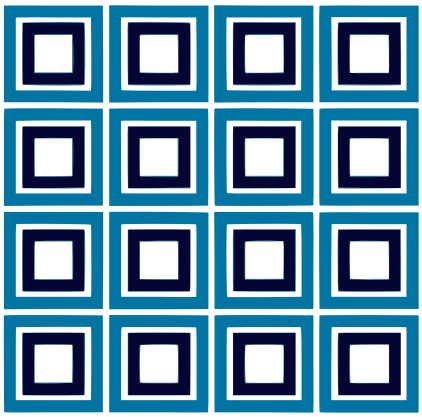
We can reduce them to minutes or even seconds



Rather than processing batches every minute or every hour, we could process them every second, or even less. This is sometimes known as micro-batching.

# Micro-Batching

But eventually, the batch sizes will contain only one record

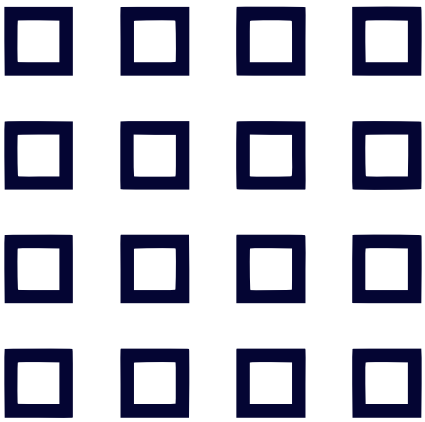


If we keep shrinking our batches to smaller and smaller sizes, eventually we are going to end up with batches that contain only a single record.

If single record batches are the natural endpoint of micro-batching...

# Micro-Batching

What if we just eliminated the batches and processed each record individually?



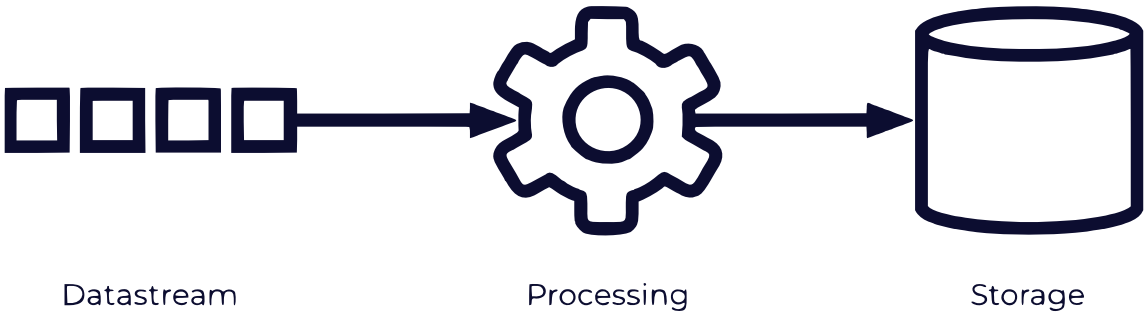
Why wouldn't we skip batching completely and process our records one at a time?



# Stream Processing

Stream Processing handles records as they arrive (without batching)

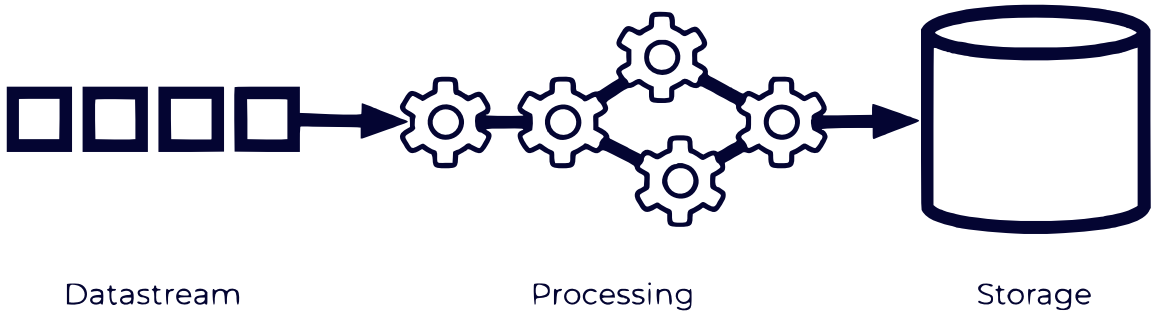



This is one of the goals of datastream programming. Rather than dealing with batches of data, datastream programming prefers to process records as they arrive.

These records are transmitted in the form of events.

# Stream Processing

Events are pushed through operators connected in a directed graph




---

They are pushed through a series of operators connected in a directed graph.

Each operator performs a transformation to the data and emits the result as a new event.



# Managing State

Ideally, each event would be completely independent

Datastream

---

We also run into challenges managing the state across operations.

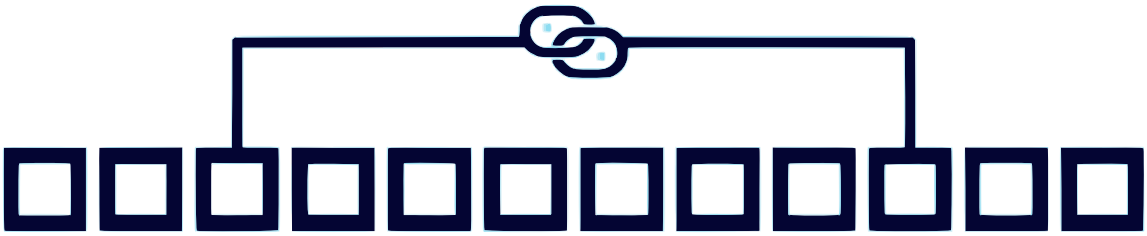
In a perfect world, each event could be treated as an independent unit. It would not rely on any other events. But that's often not the case.



# Managing State

In reality, events are often interconnected



Datastream

---

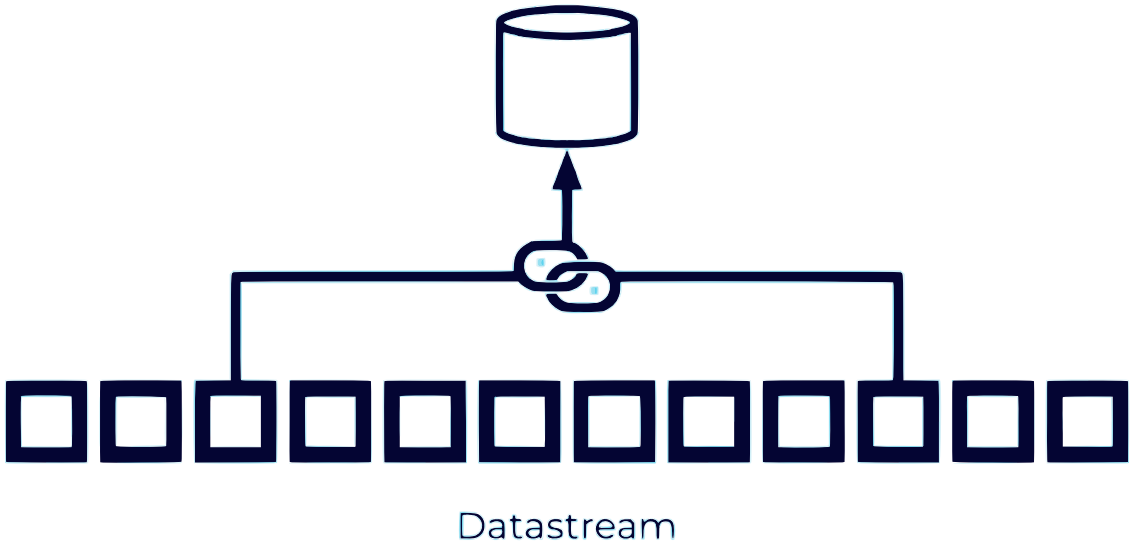
Instead, events are interconnected, and sorting out those connections requires us to keep track of state.



# Managing State

This requires storing state somewhere like a database




---

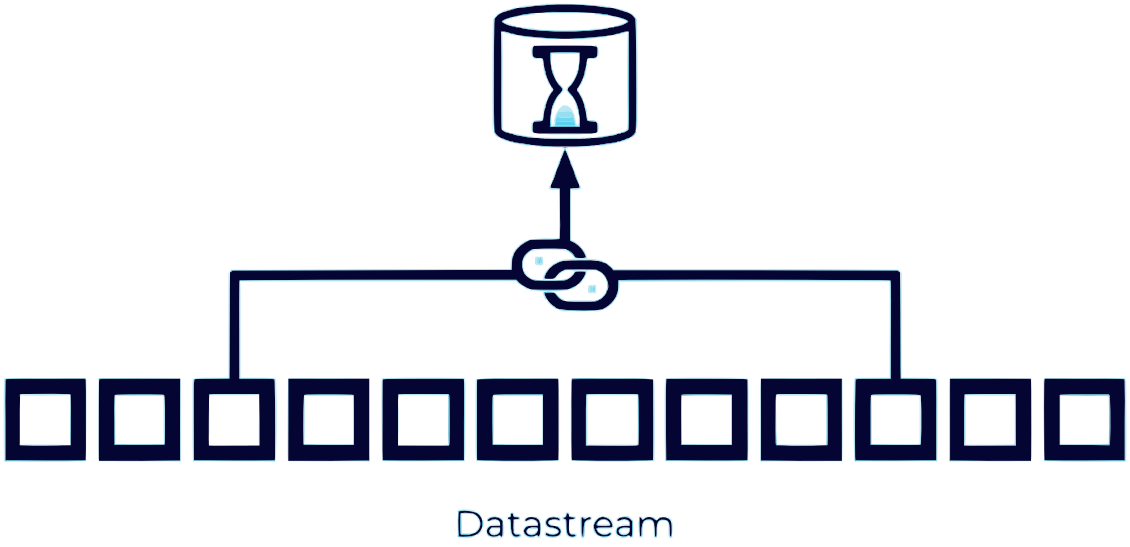
We could keep that state in a database ...



# Managing State

However, at scale, databases can be slow



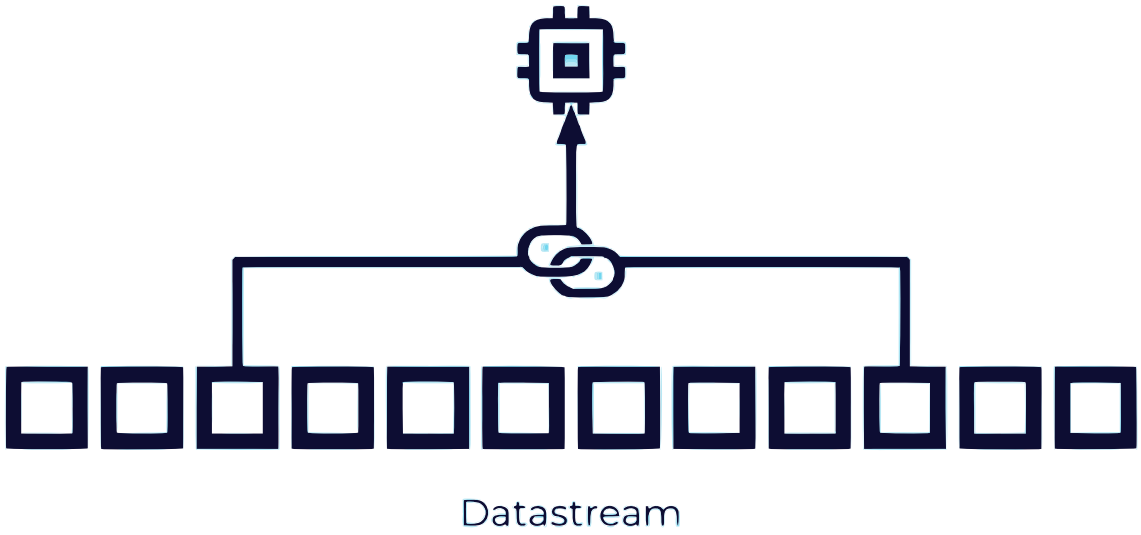

... but at scale this tends to be slow.



# Managing State

We try to speed this up using in-memory caches




---

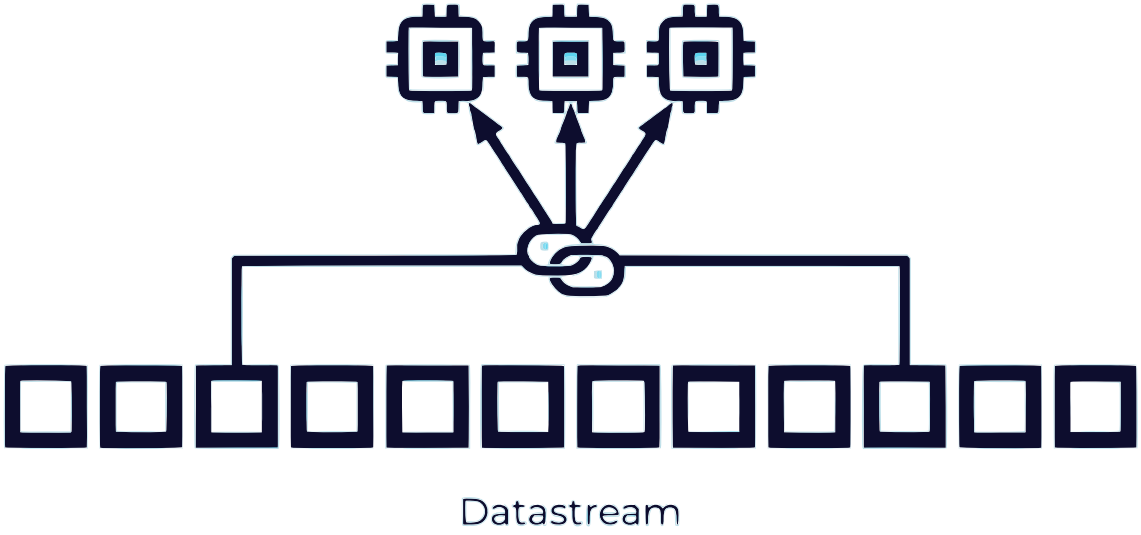
We often try to speed it up with in-memory caching ...



# Managing State

But in-memory caches are difficult to distribute




... but in-memory caches are difficult to distribute effectively

# Flink to the Rescue

Apache Flink is designed to solve these problems

---

This is where Flink steps in to help us.

Flink is a distributed datastream programming engine.

It includes facilities for managing and distributing your datastream operations.

It provides built-in support for stateful operations that can be distributed using clearly defined rules.

As a result, it has rapidly become one of the key tools for building large-scale streaming platforms.

# 01b What is Apache Flink?

## Description

Defining what Apache Flink is.



# What is Apache Flink?

Apache Flink is a framework and distributed processing engine for stateful computations over ***unbounded and bounded data streams***

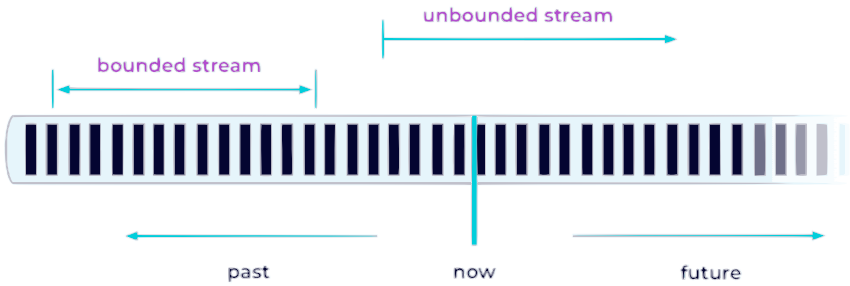
# Process Unbounded and Bounded Data Streams

**Bounded Streams:**

- Have a Start and End
- Periodic Processing
- All data is ingested before processing

**Unbounded Streams:**

- Have a Start but No End
- Continuous Processing
- Data is processed as it arrives






Any kind of data is produced as a stream of events. Credit card transactions, sensor measurements, machine logs, or user interactions on a website or mobile application, all of these data are generated as a stream. Data can be processed as unbounded or bounded streams.

Unbounded streams have a start but no defined end. They do not terminate and provide data as it is generated. Unbounded streams must be continuously processed, i.e., events must be promptly handled after they have been ingested. It is not possible to wait for all input data to arrive because the input is unbounded and will not be complete at any point in time. Processing unbounded data often requires that events are ingested in a specific order, such as the order in which events occurred, to be able to reason about result completeness.

Bounded streams have a defined start and end. Bounded streams can be processed by ingesting all data before performing any computations. Ordered ingestion is not required to process bounded streams because a bounded data set can always be sorted. Processing of bounded streams is also known as batch processing.

Apache Flink excels at processing unbounded and bounded data sets. Control of state enables Flink's runtime to run any kind of application on unbounded streams. Bounded streams are internally processed by algorithms and data structures that are specifically designed for fixed sized data sets, yielding excellent performance.

# What is Apache Flink?

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams

Flink has been designed to run in ***all common cluster environments***



# All common cluster environments

- Flink requires a *cluster of computing resources* to execute applications

- Can integrate with cluster resource managers such as:
  - Hadoop YARN
  - Kubernetes
  - Standalone mode

---

Apache Flink is a distributed system and requires compute resources in order to execute applications. Flink integrates with all common cluster resource managers such as Hadoop YARN and Kubernetes but can also be setup to run as a stand-alone cluster.

When deploying a Flink application, Flink automatically identifies the required resources based on the application's configured parallelism and requests them from the resource manager. All communication between Flink user/clients and the Flink cluster happens via REST calls. This eases the integration of Flink in many environments. Flink users/clients are developers, administrators, or automated systems that need to interact with the Flink cluster to submit, control, or monitor Flink jobs.

The standalone mode is the most barebone way of deploying Flink: The Flink services described in the deployment overview are just launched as processes on the operating system. Unlike deploying Flink with a resource provider such as Kubernetes or YARN, you have to take care of restarting failed processes, or allocation and de-allocation of resources during operation.

# What is Apache Flink?

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams

Flink has been designed to run in all common cluster environments, ***perform computations at in-memory speed***

---

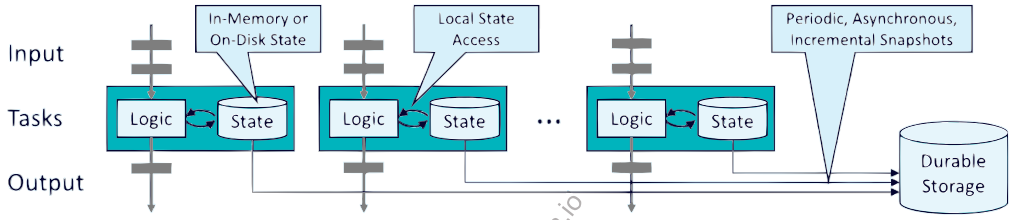


# Local In-Memory State Access

- Flink achieves very low processing latencies for stateful applications:
  - Local state access
  - Task state is always maintained in memory and disk

- Flink guarantees exactly-once state consistency by checkpointing the local state to durable storage






Stateful Flink applications are optimized for local state access. Task state is always maintained in memory or, if the state size exceeds the available memory, in access-efficient on-disk data structures. Hence, tasks perform all computations by accessing local, often in-memory, state yielding very low processing latencies. Flink guarantees exactly-once state consistency in case of failures by periodically and asynchronously checkpointing the local state to durable storage.

Out of the box, Flink bundles these state backends:

- HashMapStateBackend (default): Keeps state in memory
- EmbeddedRocksDBStateBackend: Keeps state in memory and disk

Flink's state backends use a copy-on-write mechanism to allow stream processing to continue unimpeded while older versions of the state are being asynchronously snapshotted. This is a strategy where modifications to data structures are done on a copy rather than the original. Only when the snapshots have been durably persisted will these older versions of the state be garbage collected. Find more information about copy-on-write in these references: ref-1, ref-2

# What is Apache Flink?

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams

Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and ***at any scale***

---



# Run Applications at any Scale

- Apache Flink is a distributed system
- Applications can be parallelized into possibly thousands of tasks distributed in a cluster

Users reported Flink applications running in their production environments:

- Processing multiple trillions of events per day
- Maintaining multiple terabytes of state
- Running on thousands of cores

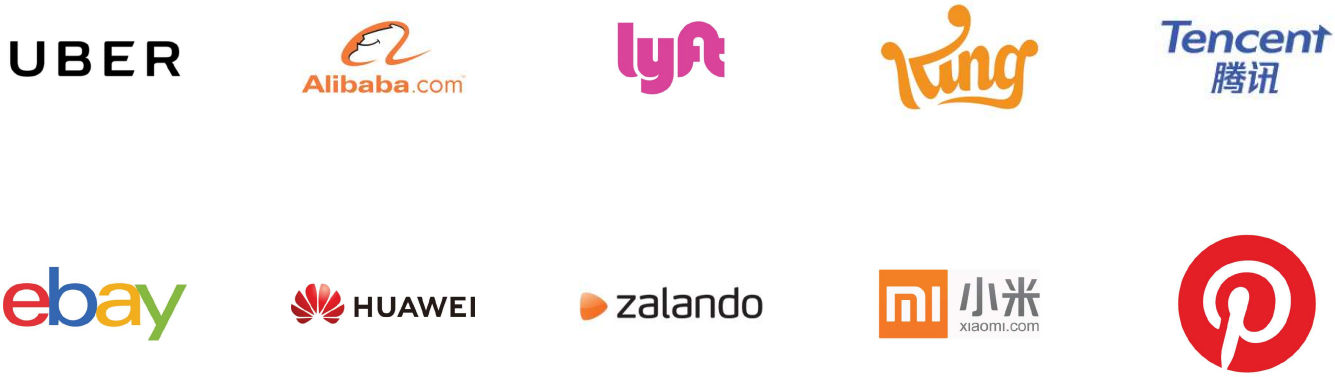
---

Flink is designed to run stateful streaming applications at any scale. Applications are parallelized into possibly thousands of tasks that are distributed and concurrently executed in a cluster. Therefore, an application can leverage virtually unlimited amounts of CPUs, main memory, disk and network IO. Moreover, Flink easily maintains very large application state. Its asynchronous and incremental checkpointing algorithm ensures minimal impact on processing latencies while guaranteeing exactly-once state consistency.



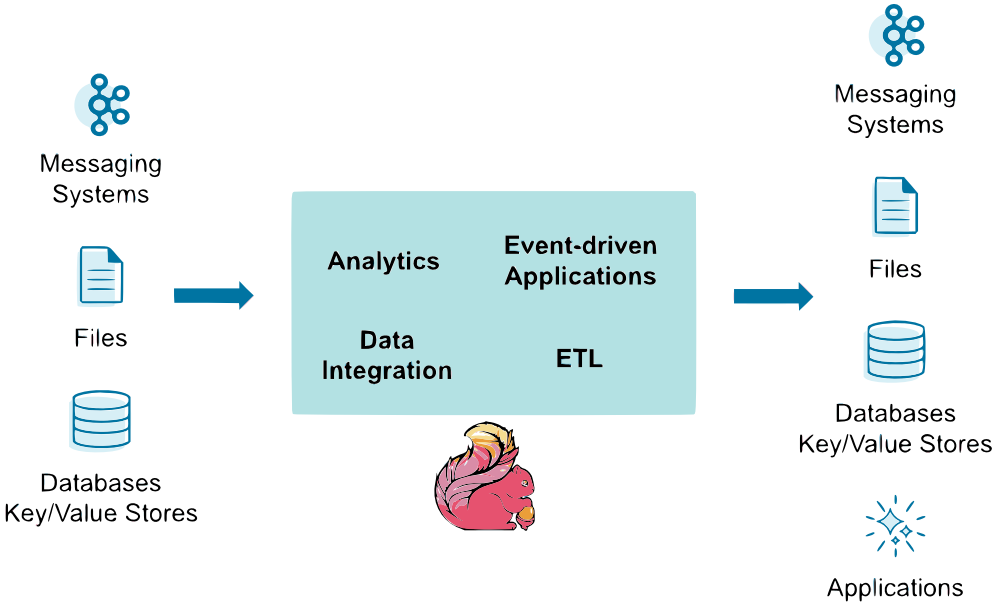
# Flink Users



# What is Apache Flink used for?

- Transactions
- Logs
- IoT
- Events
- Interactions

...



# 01c: Apache Flink's APIs

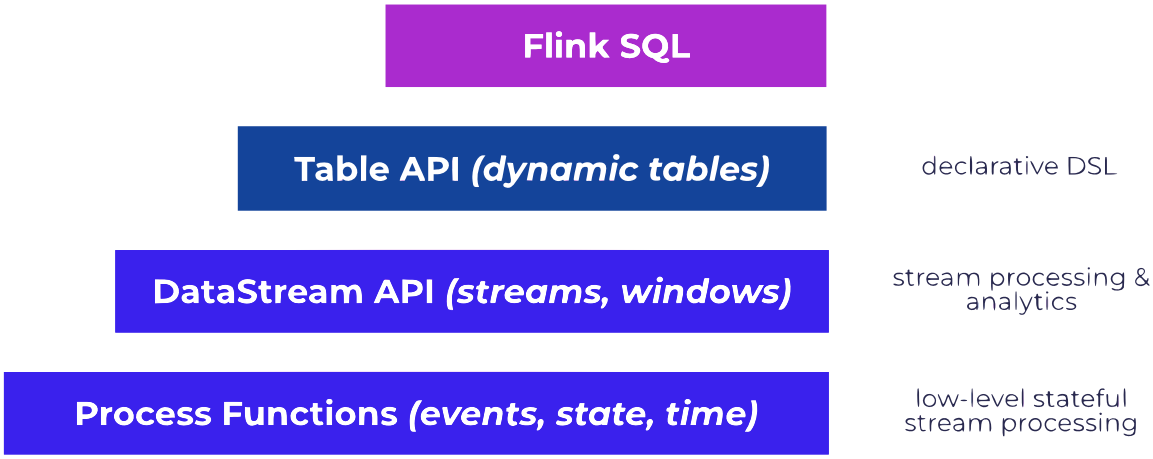
## Description

Defining what Apache Flink is.



# Flink's APIs




Flink SQL is just one of several APIs available to Flink developers. Flink's other APIs operate at lower levels of abstraction.

Working our way up from the bottom of this diagram, we find Process Functions. A process function is a primitive building block, capable of implementing almost any operation by directly manipulating Flink's state backends and timer services. At this level of the API stack you are writing code that reacts to each event as it arrives, one at a time.



Technically speaking, process functions are actually part of the DataStream API. Most of the DataStream API is at a slightly higher level of abstraction, where the building blocks include abstractions like streams and windows.

Approaching the top of this diagram, we find the Table API. In terms of the abstractions involved, the Table API is roughly equivalent to Flink SQL, but as a developer using the Table API you are writing Java or Python code, rather than SQL.

It's worth noting that these different APIs are all interoperable. It's not that you must choose one of these to the exclusion of the others – in fact, a single Flink application could use all of these APIs together.

Programs can be written in Java, Scala, Python, and SQL and are automatically compiled and optimized into dataflow programs that are executed in a cluster or cloud environment.

# 01d: Flink Job & Topology

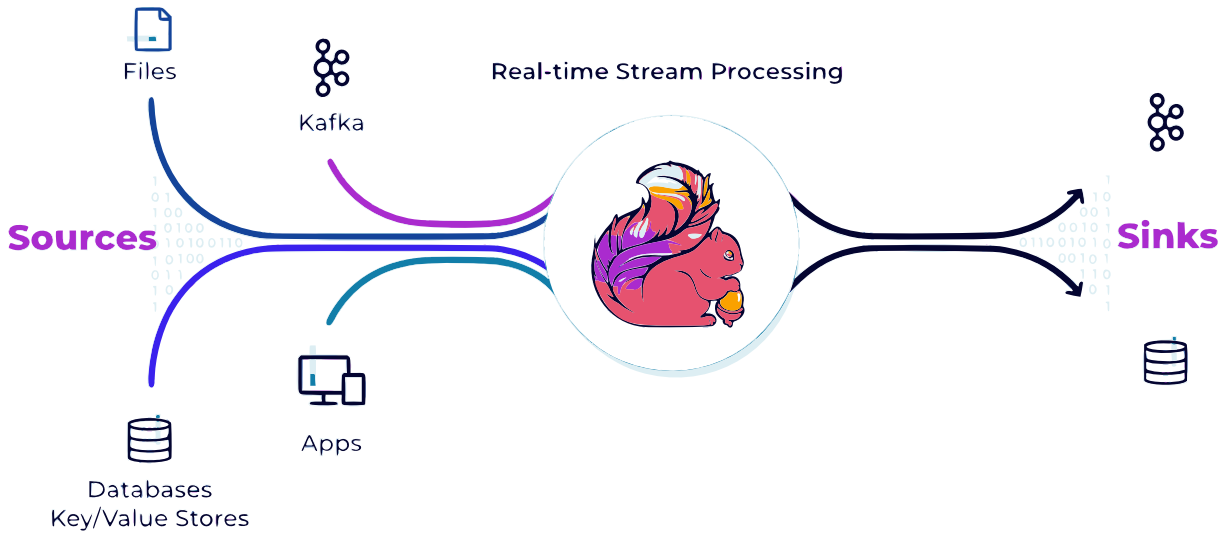
## Description

Defining a Flink Job, topology and its components.



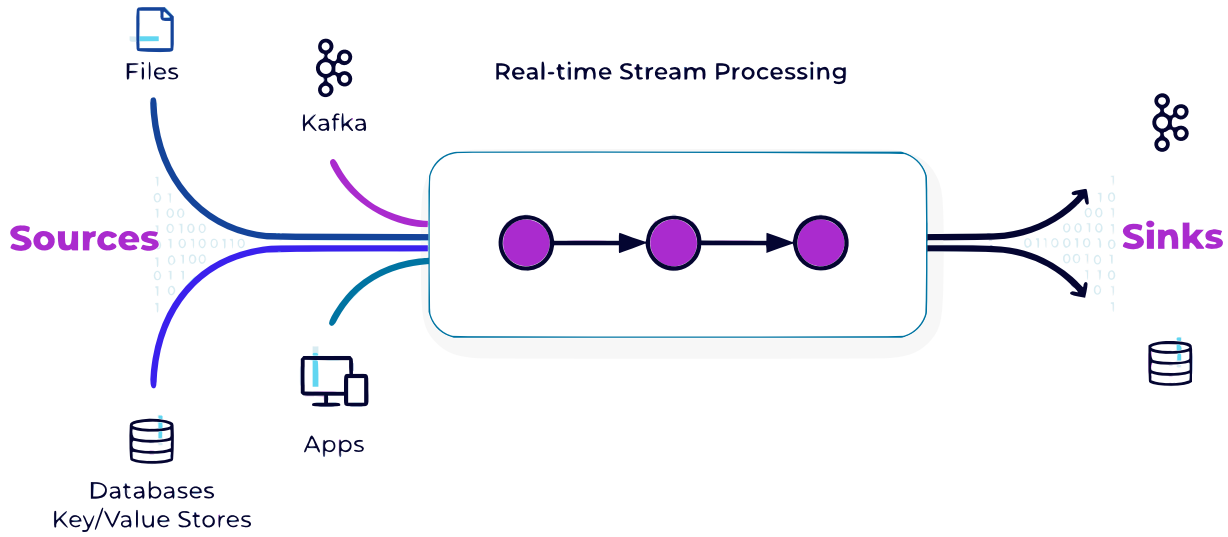
 

# Real-time services rely on Stream Processing



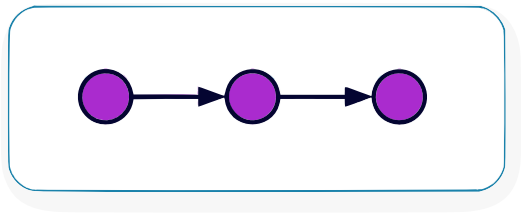

Flink applications consume data from one or more event sources, and produce data to one or more sinks. These sources and sinks can be messaging systems, such as Kafka, or files, or databases, or any service or application that produces and consumes data.

# Real-time services rely on Stream Processing




Our focus in this course will be on this part in the middle, sitting in-between the sources and sinks. This is where your stream processing business logic goes.

# The Job Graph (or Topology)

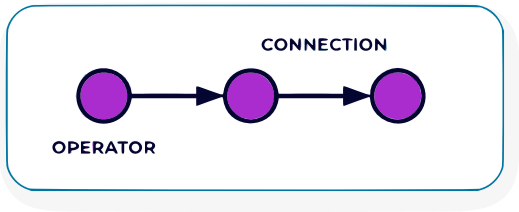


As a Flink developer, you will define your business logic using one of Flink's APIs, and Flink will execute your code in a Flink cluster. A running Flink application is called a Job, and the event data is streaming through a data processing pipeline that we call the Job Graph or Topology. The nodes in the Job Graph represent the processing steps in your pipeline.



# The Job Graph (or Topology)

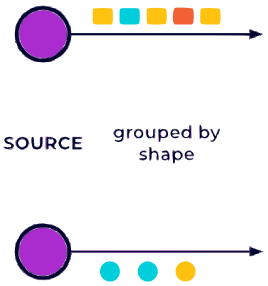



Each of these processing steps is executed by an operator. Operators transform event streams.

The operators in the job graph are connected to one another, as shown here by the arrows that form the edges of the graph. Technically speaking, the Job Graph is always a directed, acyclic graph, where the event data is always flowing from the sources toward the sink, and being processed along the way by the operators

# Job Graph Example

- **Parallel**
- Forward
- Repartition
- Rebalance

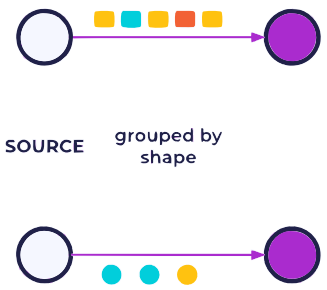



Stream processing is done in parallel by partitioning event streams into parallel sub-streams, each of which can be independently processed. This is a key point – this partitioning into independently processed pipelines is crucial for scalability. These independent parallel operators share nothing, and can run at full speed.

It's typically the case that the input to a Flink job can be consumed in parallel. Often these parallel input streams will have been partitioned upstream, before being ingested by Flink. In this example, some upstream process has produced data that is partitioned by shape.

# Job Graph Example

- Parallel
- **Forward**
- Repartition
- Rebalance




* The source is grouped by shape (source partition).

---

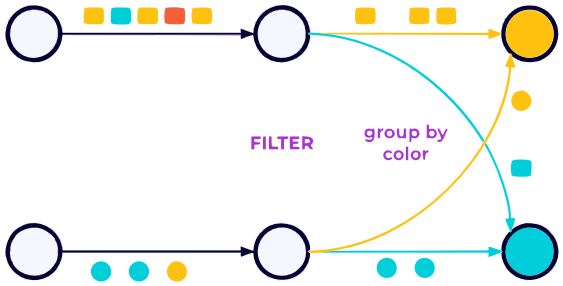
At each stage of the Job Graph, your application code will use one of Flink's APIs to specify both what to do in that operator, and where that operator should send its results.

In this example, the first operator is collecting the input from the source, and forwarding it downstream.

Forwarding an event stream is the simplest thing an operator can do, and Flink will optimize this type of connection so that it is handled very efficiently.

# Job Graph Example

- Parallel
- Forward
- **Repartition**
- Rebalance




---

The second operator in this job is filtering the event streams to remove any orange events, and then it is reorganizing the streams so that they are grouped by color, rather than by shape.

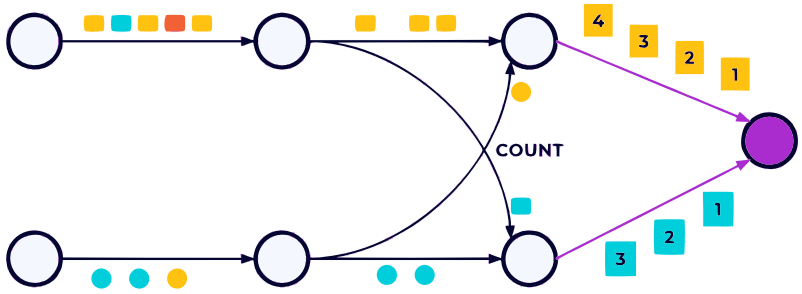
Sometimes a shuffle, or repartitioning like this, is necessary.

This particular shuffle brings together all of the yellow events so they will be processed by the same node, and similarly, all of the blue events will also be processed together. This makes it possible to count all of the yellow events, and all of the blue events.

Shuffling event streams is much more expensive than forwarding them. Each event has to be serialized, and depending on where an event is being routed, it may have to be sent over the network to the next operator downstream.

# Job Graph Example

- Parallel
- Forward
- Repartition
- **Rebalance**




At the right-hand side of this diagram you see the results of the COUNT operation streaming towards the sink.

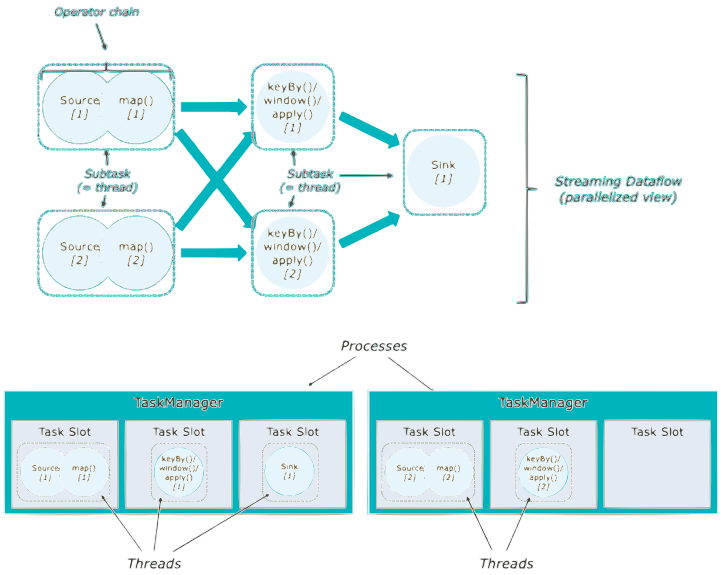
For the yellow events, a total of 1, then 2, then 3, and now 4 events have been processed. And for the blue, the count has progressed from 1 to 2 and then to 3. Instead of having each instance of the COUNT operator forward its results to one of two parallel sinks, I've chosen instead to change the parallelism so that there's only one sink This is accomplished by a stream operation called rebalancing, which means that the output elements are distributed evenly to instances of the next operation in a round-robin fashion. For example, if the upstream operation has parallelism 2 and the downstream operation has parallelism 4, then one upstream operation would distribute elements to two downstream operations while the other upstream operation would distribute to the other two downstream operations. If, on the other hand, the downstream operation has parallelism 2 while the upstream operation has parallelism 4 then two upstream operations will distribute to one downstream operation while the other two upstream operations will distribute to the other downstream operations.

In this scenario, since rebalancing changes parallelism to 1, round-robin does not make much sense since all events are sent to the same operator. In case the parallelism was changed to a higher number, the events would be redistributed in a round-robin fashion.

Since in this example the parallelism is being reduced from two to one, the streams are being merged at the sink.

Just to be clear here – this sink could, and probably should, operate at the same parallelism as the rest of the job graph. I reduced the parallelism merely for the sake of having an example of a rebalance.

# Flink's Architecture




For distributed execution, Flink chains operator subtasks together into tasks. Each subtask is executed by one thread. Chaining operators together into tasks is a useful optimization: it reduces the overhead of thread-to-thread handover and buffering, and increases overall throughput while decreasing latency. The dataflow in the figure is executed with five subtasks, and hence with five parallel threads.

Each worker (TaskManager) is a JVM process, and may execute one or more subtasks in separate threads. To control how many tasks a TaskManager accepts, it has so called task slots (at least one).

Each task slot represents a fixed subset of resources of the TaskManager. A TaskManager with three slots, for example, will dedicate 1/3 of its managed memory to each slot. Slotting the resources means that a subtask will not compete with subtasks from other jobs for managed memory, but instead has a certain amount of reserved managed memory. Note that no CPU isolation happens here; currently slots only separate the managed memory of tasks.

By adjusting the number of task slots, users can define how subtasks are isolated from each other. Having one slot per TaskManager means that each task group runs in a separate JVM (which can be started in a separate container, for example). Having multiple slots means more subtasks share the same JVM. Tasks in the same JVM share TCP connections (via multiplexing) and heartbeat messages. They may also share data sets and data structures, thus reducing the per-task overhead.

By default, Flink allows subtasks to share slots even if they are subtasks of different tasks,

so long as they are from the same job. The result is that one slot may hold an entire pipeline of the job.



# Lab Module 01: Setting up the Lab Environment

Please work on **Lab Module 01: Setting up the Lab Environment**.

Refer to the Exercise Guide.