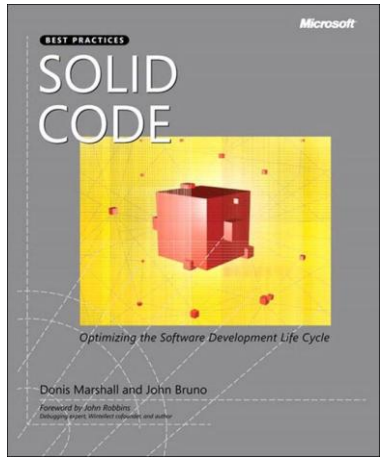
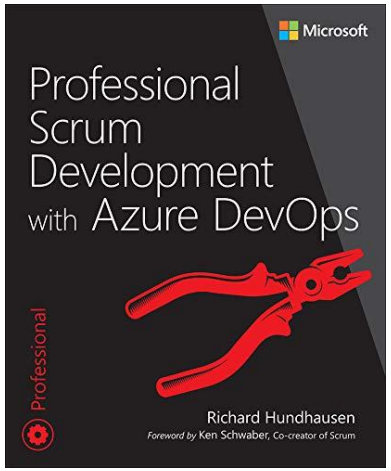
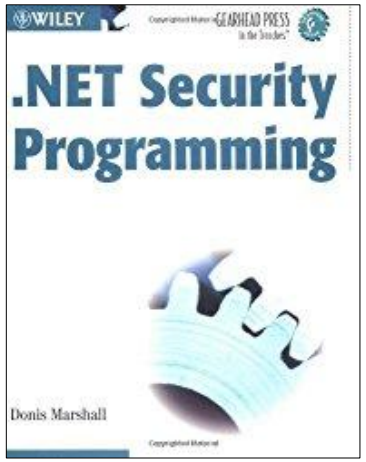
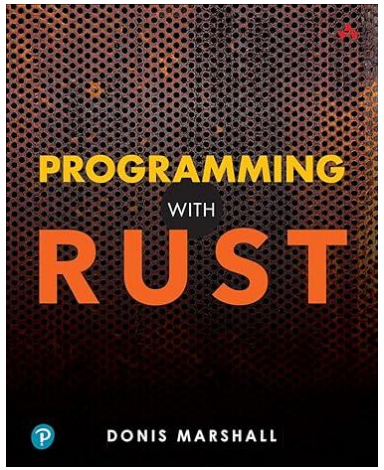
George Niece

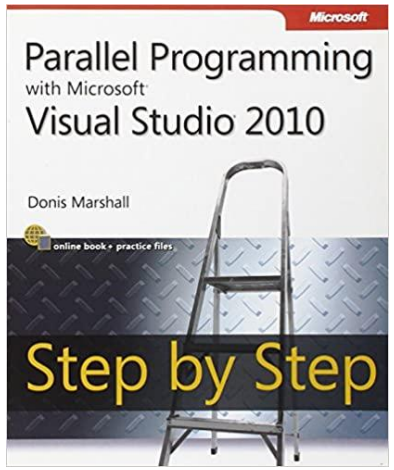
- Data Engineering Pipelines
- Security
- Multicloud – AWS Primary
- Resilience
- Certified in many technologies

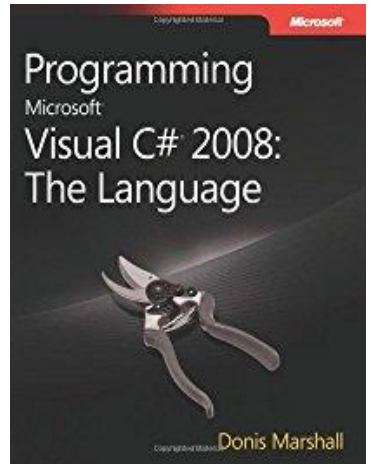












COURSE OUTLINE

Introduction to Stream Processing

- Understanding Data Processing
  - Batch Processing Overview
  - Stream Processing Overview
  - When to Use Stream Processing
  - Common Use Cases
- Stream Processing Concepts
  - Events and Streams
  - Real-time vs Near Real-time
  - Processing Guarantees
- Basic Architecture Patterns

Getting Started with Apache Flink

- Apache Flink Basics
  - What is Apache Flink?
  - Key Features and Components
  - Basic Architecture
  - Development Environment Setup
- Your First Flink Application
  - Project Structure
  - Basic Configuration
  - Hello World Example
  - Running Locally

COURSE OUTLINE

Working with DataStreams

- DataStream Basics
  - Creating DataStreams
  - Basic Operations
  - Data Types
  - Simple Transformations
- Common Operations
  - Map and FlatMap
  - Filter Operations
  - Basic Aggregations
  - Field Selection

Data Sources and Sinks

- Built-in Sources
  - File-based Sources
  - Socket Sources
  - Collection Sources
  - Generating Test Data
- Built-in Sinks
  - File Sinks
  - Print Sink
  - Socket Sink
  - Common Formats

COURSE OUTLINE

Time and Windows

- Understanding Time
  - Event Time vs Processing Time
  - Timestamps
  - Watermarks Basics
  - Dealing with Late Events
- Window Operations
  - Types of Windows
  - Tumbling Windows
  - Sliding Windows
  - Session Windows

Basic State Management

- State Concepts
  - What is State?
  - When to Use State
  - Simple State Examples
  - State Backends
- Working with State
  - Keyed State
  - Value State
  - List State
  - Basic State Patterns

COURSE OUTLINE

Error Handling and Recovery

- Basic Fault Tolerance
  - Understanding Failures
  - Checkpointing Basics
  - Simple Recovery Patterns
- Best Practices
  - Monitoring and Logging
  - Basic Metrics
  - Logging Configuration
  - Common Issues
  - Troubleshooting

Deployment and Next Steps

- Basic Deployment
  - Deployment Options
  - Local Cluster
  - Configuration Basics
  - Resource Planning
- Moving Forward
  - Best Practices Review
  - Advanced Topics Preview
  - Learning Resources
  - Common Use Cases

Introduction to Stream Processing

Introduction to Stream Processing

- Understanding Data Processing
  - Batch Processing Overview
  - Stream Processing Overview
  - When to Use Stream Processing
  - Common Use Cases
- Stream Processing Concepts
  - Events and Streams
  - Real-time vs Near Real-time
  - Processing Guarantees
- Basic Architecture Patterns

UNDERSTANDING DATA PROCESSING

- There are two forms of data processing, real-time (stream) and batch. The majority of large organizations have both, in some cases both pattern for a specific business case, like fraud detection.
- Data engineering enables efficient and scalable data pipelines for various applications, including real-time analytics and event-driven systems.
- We have multiple data processing characteristics:
  - Data processing characteristics encompass the stages and qualities involved in transforming raw data into usable information, including collection, preparation, transformation, and output, all while ensuring data quality, security, and performance.
  - Processing, data enrichment, sinks & sources, transformations, information, analytics, and actionable insight.
- We can structure information in different ways
  - Structured Information – has a defined schema – traditional data
  - Semi- Structured Information – has a structure but not a schema – XML, Documents
  - Unstructured Information – does not have a structure – text, video, images

DATA PROCESSING CHARACTERISTICS

1. Data Collection:

Where is the data coming from? Surveys, sensors, databases, and web scraping.

2. Data Input:

What is the format needed?

3. Data Preparation (Data Cleaning/Wrangling):

What is needed to augment or enrich the data,, de-dup, …

4. Data Transformation:

Are sorting, summarization, or aggregates required?

5. Data Validation:

Are there standards we need to meet or inconsistencies to resolve?

6. Data Output:

How is the data going to used for visualization, analysis or decision support?

7. Data Storage:

Where is this data at rest?

8. Data Quality Characteristics:

What is needed for accuracy, completeness, reliability, relevance, and timeliness?

9. Security Aspects:

Are protections in place for the data from unauthorized access, use, disclosure, disruption, modification, or destruction?

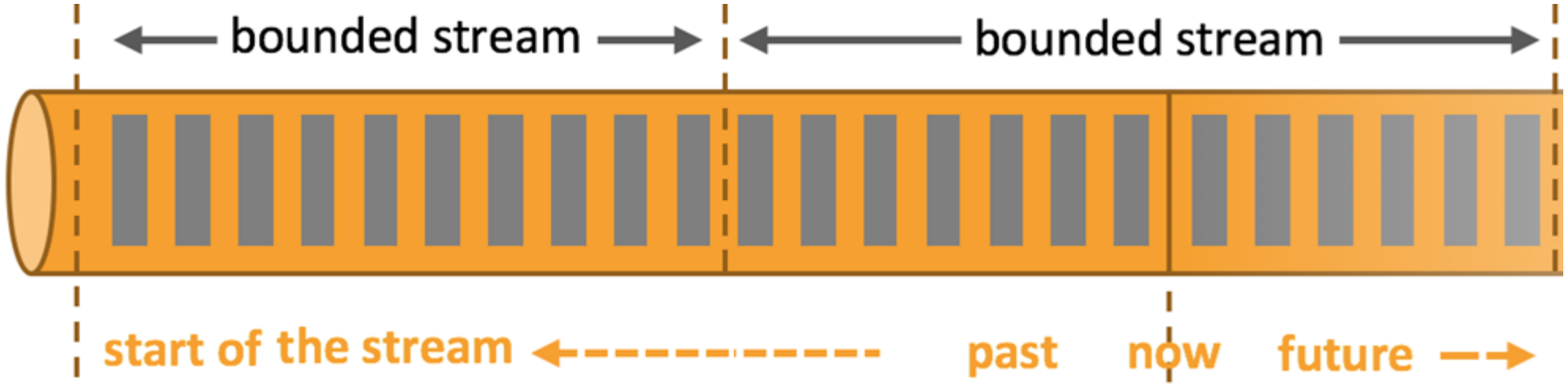
Have we implemented encryption, access control, cyber recovery, and regular audits

10. Performance:

Do we have observability around system for speed, scalability, and resource utilization.

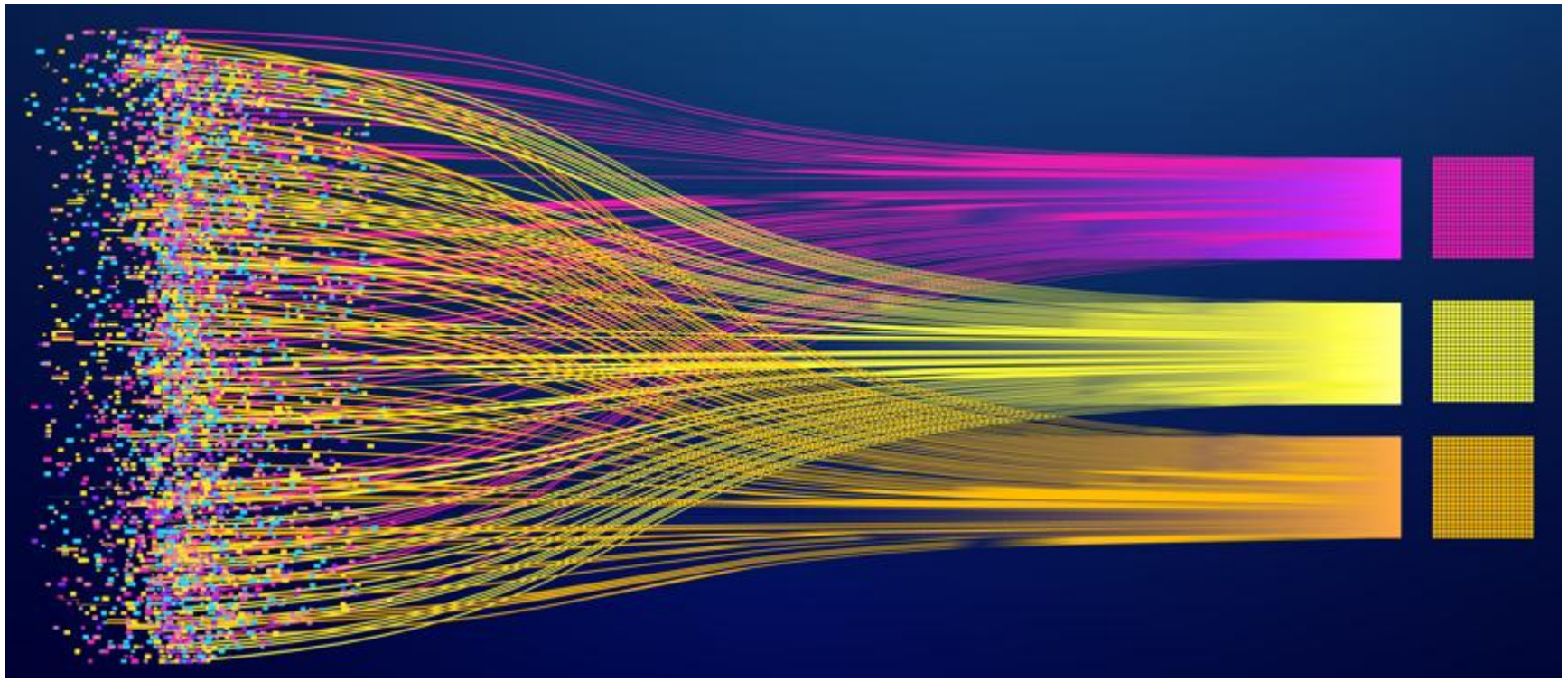
BATCH PROCESSING OVERVIEW

- Batch processing involves processing finite, bounded datasets, often at scheduled intervals, prior to Flink 2.0 the DataSet API, used this as a special case of stream processing where the order and time of records don't matter.
- Flink batch processing is now considered a special case with the steam being finite, this is a path with the DataStream API for a unified approach.
- Using known data sets allows for scheduling and optimization.



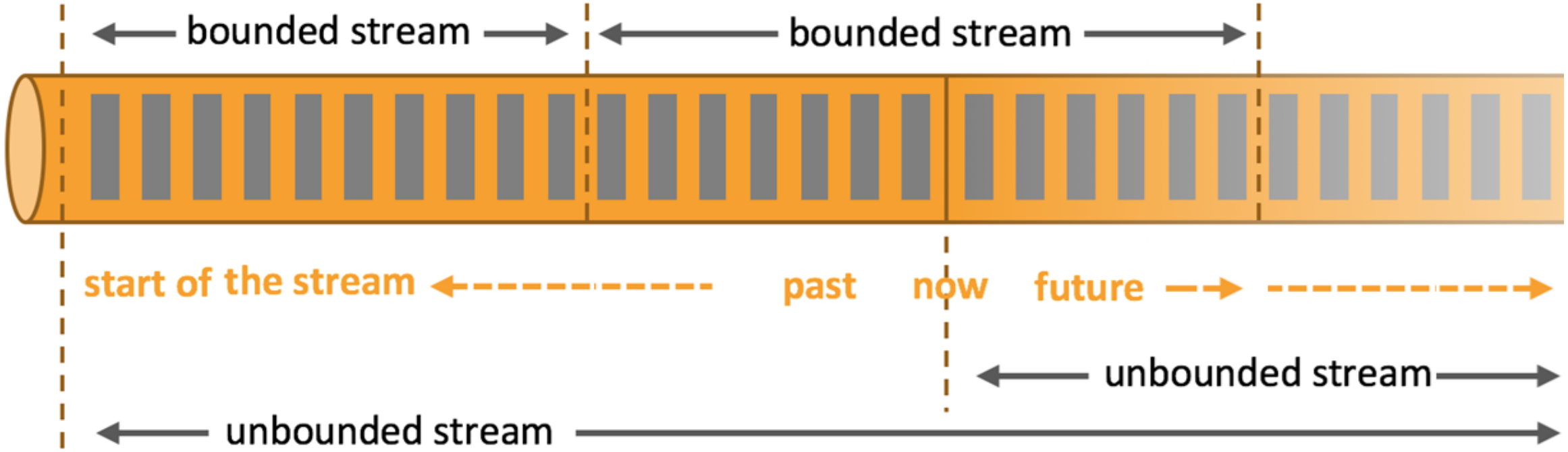
BATCH PROCESSING

- **Batch processing** is the paradigm at work when you process a **bounded data stream**. In this mode of operation, you can choose to ingest the entire dataset before producing any results, which means that it's possible, for example, to sort the data, compute global statistics, or produce a final report that summarizes all of the input.



STREAM PROCESSING OVERVIEW

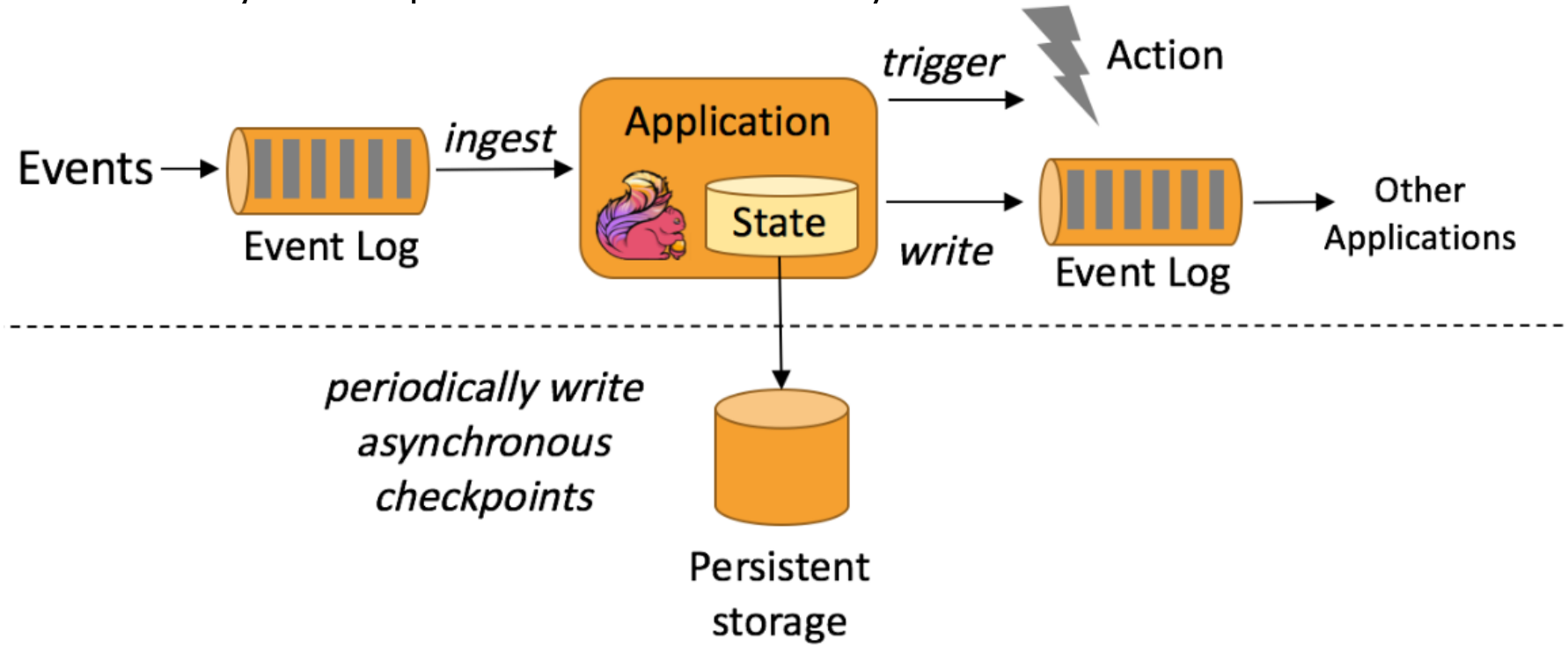
- Stream processing involves unbounded data streams. Conceptually, at least, the input may never end, and so you are forced to continuously process the data as it arrives. This leads to thinking about this time series data over a period, rather than the consumption of the entire data set as you would with batch processes.



WHEN TO USE STREAM PROCESSING?

WHEN TO USE STREAM PROCESSING

Opt for streaming processing if your business **requires real-time insights and immediate action**. This is ideal for applications such as **fraud detection, live traffic management, or real-time customer engagement**, where timely data analysis is critical for decision-making and operational efficiency.



TIMELY STREAM PROCESSING

- For most streaming applications it is very valuable to be able re-process historic data with the same code that is used to process live data – and to produce deterministic, consistent results, regardless.
- The order in which events occurred, rather than the order in which they are delivered for processing, and to be able to reason about when a set of events is (or should be) complete.
- Use event time timestamps that are recorded in the data stream to be able to capture the view of a time period within the stream.

COMMON USE CASES

Flink streams, a powerful stream processing framework, are commonly used for real-time analytics, event-driven applications, and building data pipelines, including tasks like:

- fraud detection
- real-time dashboards
- data processing.

COMMON USE CASES

Real-time analytics and dashboards: Flink can be used to process real-time data streams and generate real-time dashboards and reports.

Fraud detection: Flink can be used to detect fraudulent transactions and activities in real-time.

Event-driven applications: Flink can be used to build event-driven applications that react to real-time events.

Data pipelines and ETL: Flink can be used to build data pipelines and ETL (extract, transform, load) processes.

Financial market analysis: Flink can be used to analyze financial market data in real-time.

Social media analysis: Flink can be used to analyze social media data in real-time.

Internet of Things (IoT): Flink can be used to process data from IoT devices.

STREAM PROCESSING CONCEPTS

Streams are the de-facto way data is created. Whether the data comprises events from web servers, trades from a stock exchange, or sensor readings from a machine on a factory floor, data is created as part of a stream. We may chunk this data up into a finite (bounded) set of transactions for something like Friends Day or Cyber Monday, in Digital Commerce.

EVENTS AND STREAMS

Stream processing systems process data promptly upon arrival, often in small, incremental units known as events. This capability allows organizations to swiftly extract value from their data, for time-sensitive decisions and situations demanding real-time insights. We could consider health for a person as a lifetime stream, but typically we're looking at events, a day/week/year of life, or for specific health events like catching a cold or have your teeth cleaned.

STATE MANAGEMENT

Stream processing frequently integrates state management to effectively handle continuous data streams. This state captures pertinent information necessary for subsequent event processing or assistance. State can take various forms, such as:

- Incremental Aggregates
- Static Data
- Previously Seen Events

REAL-TIME VS NEAR REAL-TIME

Usually when we talk about streaming, we think about two categories:

- Real-time: usually in the realms of sub-milliseconds to seconds
- Near real-time: sub-seconds to hours

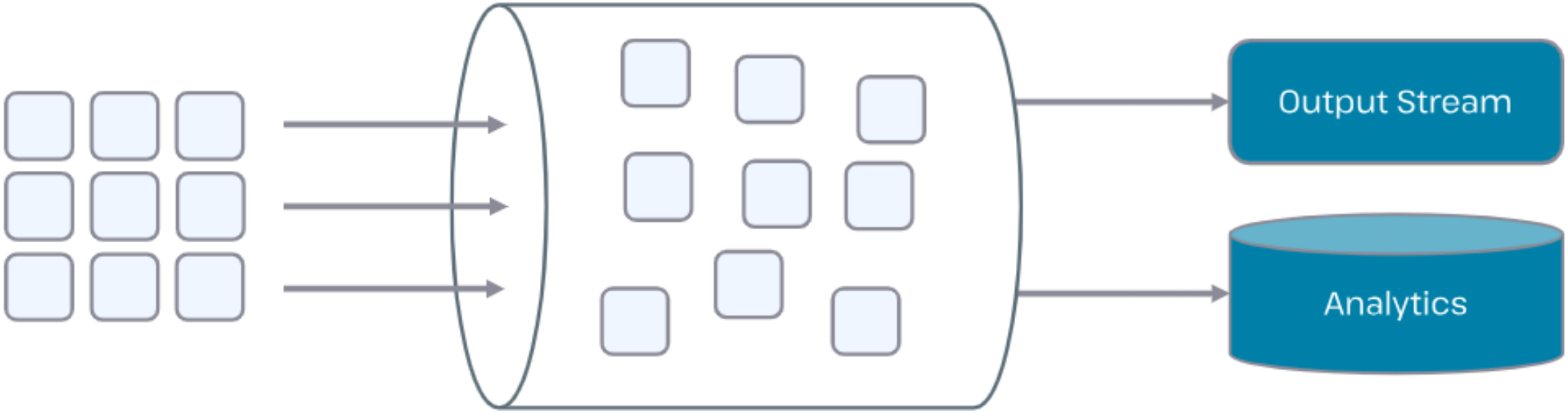
The distinction isn't in the actual timing of the stream, but rather the decision, deadline or response that the event or signal generates. The response should occur similarly under any system load, rather than becoming bi-modal.

PROCESSING GUARANTEES

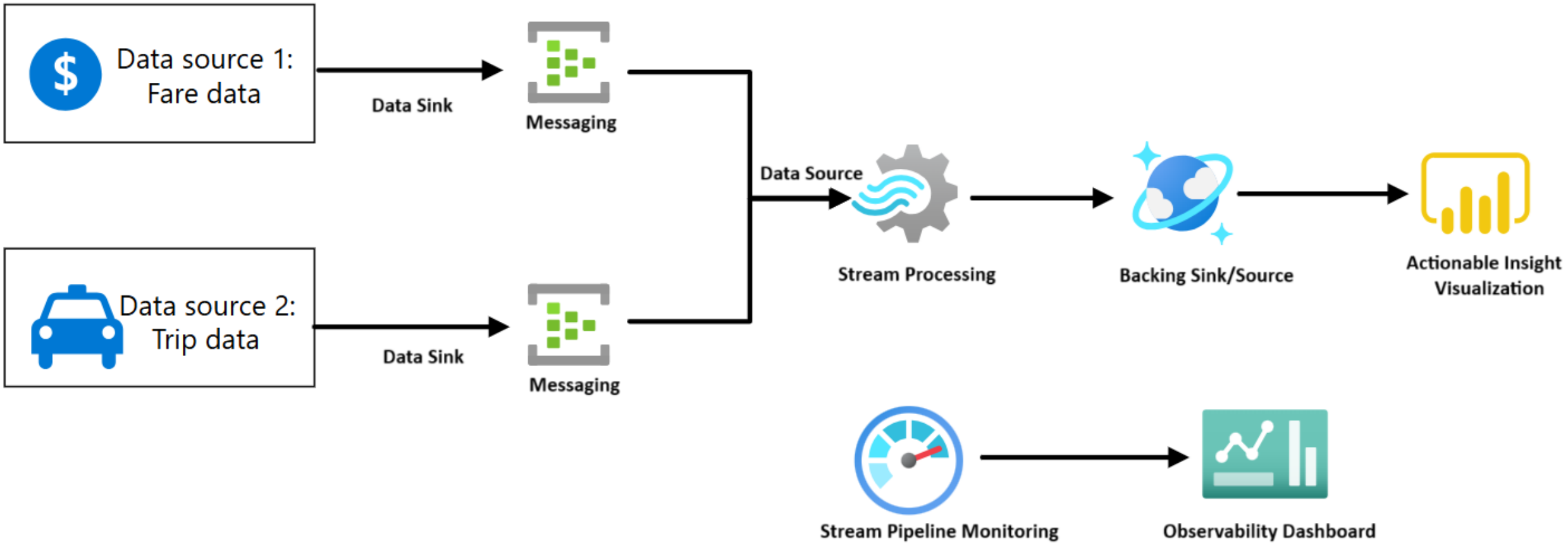
Fault tolerance mechanisms recover processes in the presence of failures and continues to execute them. Such failures include hardware failures, network failures, transient program failures, etc.

Fault tolerance guarantees of data sources and sinks, are tied to the concepts of at least once, at most once, and exactly once. Stream processing pipelines can guarantee exactly-once state updates to user-defined state only when the source or sink participates in the snapshotting or checkpointing mechanism.

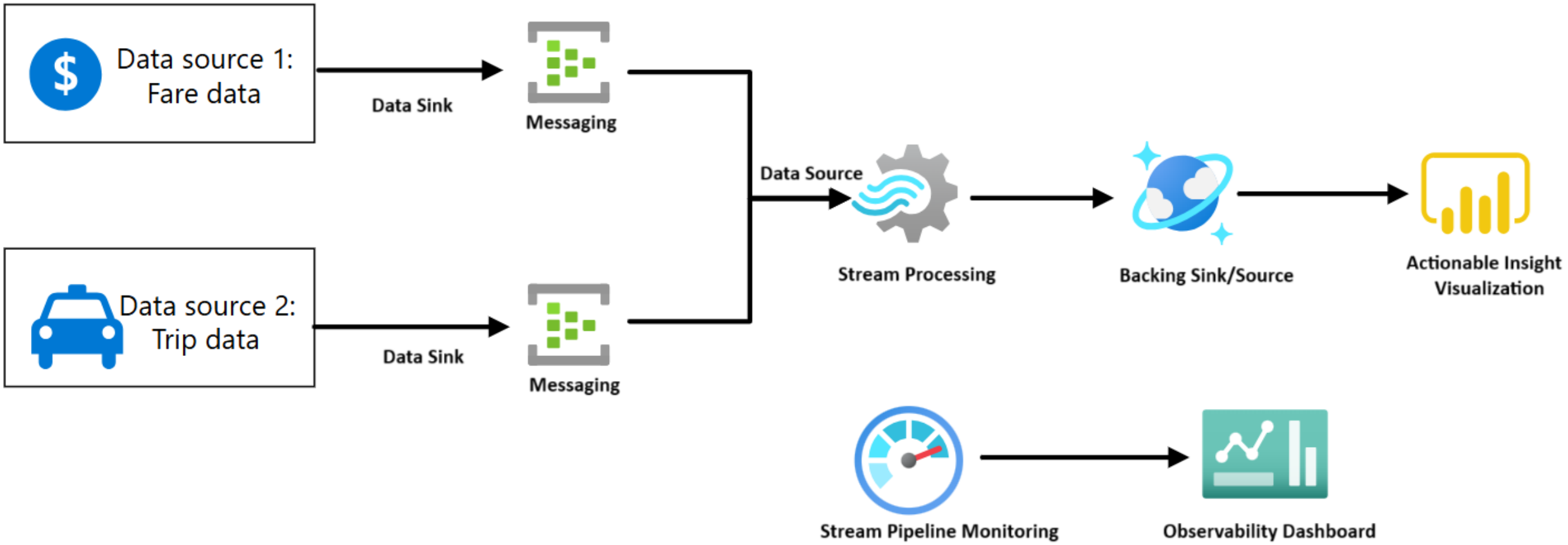
BASIC ARCHITECTURE PATTERNS



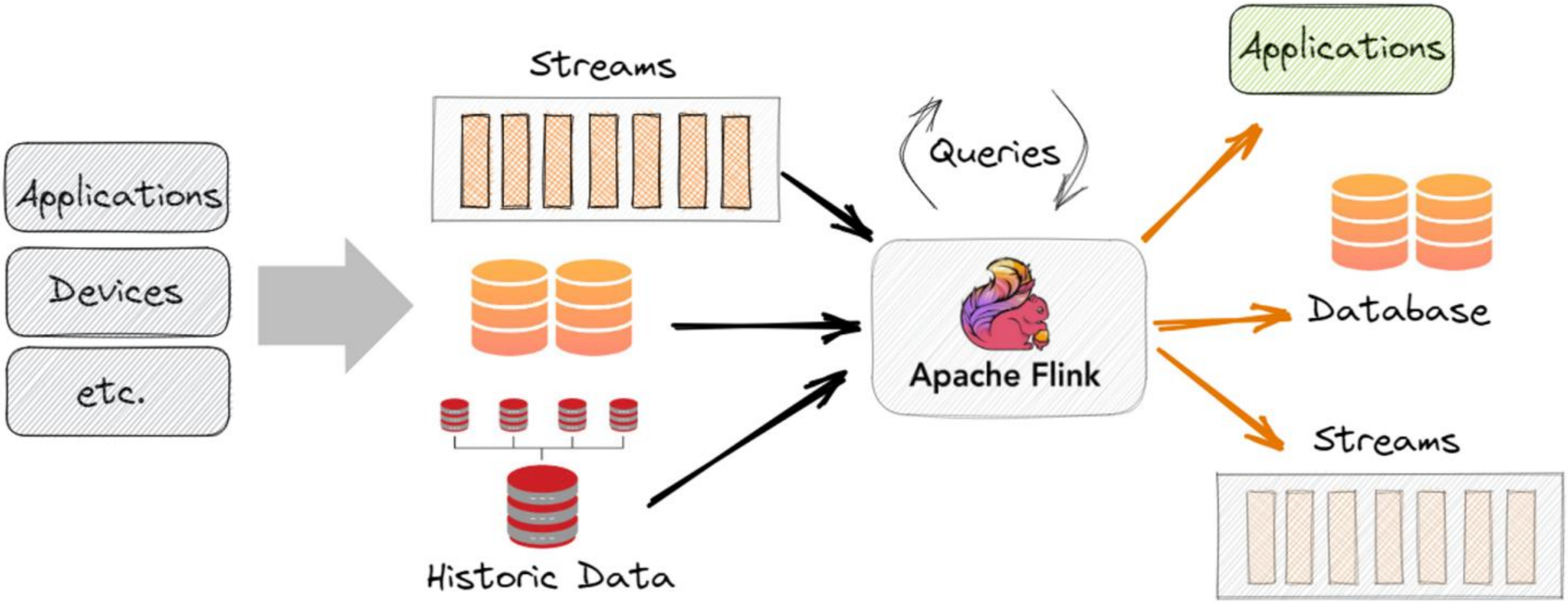
BASIC ARCHITECTURE PATTERNS



BASIC ARCHITECTURE PATTERNS



BASIC ARCHITECTURE PATTERNS



Q&A AND OPEN DISCUSSION

Getting Started with Apache Flink

Getting Started with Apache Flink

- Apache Flink Basics
  - What is Apache Flink?
  - Key Features and Components
  - Basic Architecture
  - Development Environment Setup
- Your First Flink Application
  - Project Structure
  - Basic Configuration
  - Hello World Example
  - Running Locally

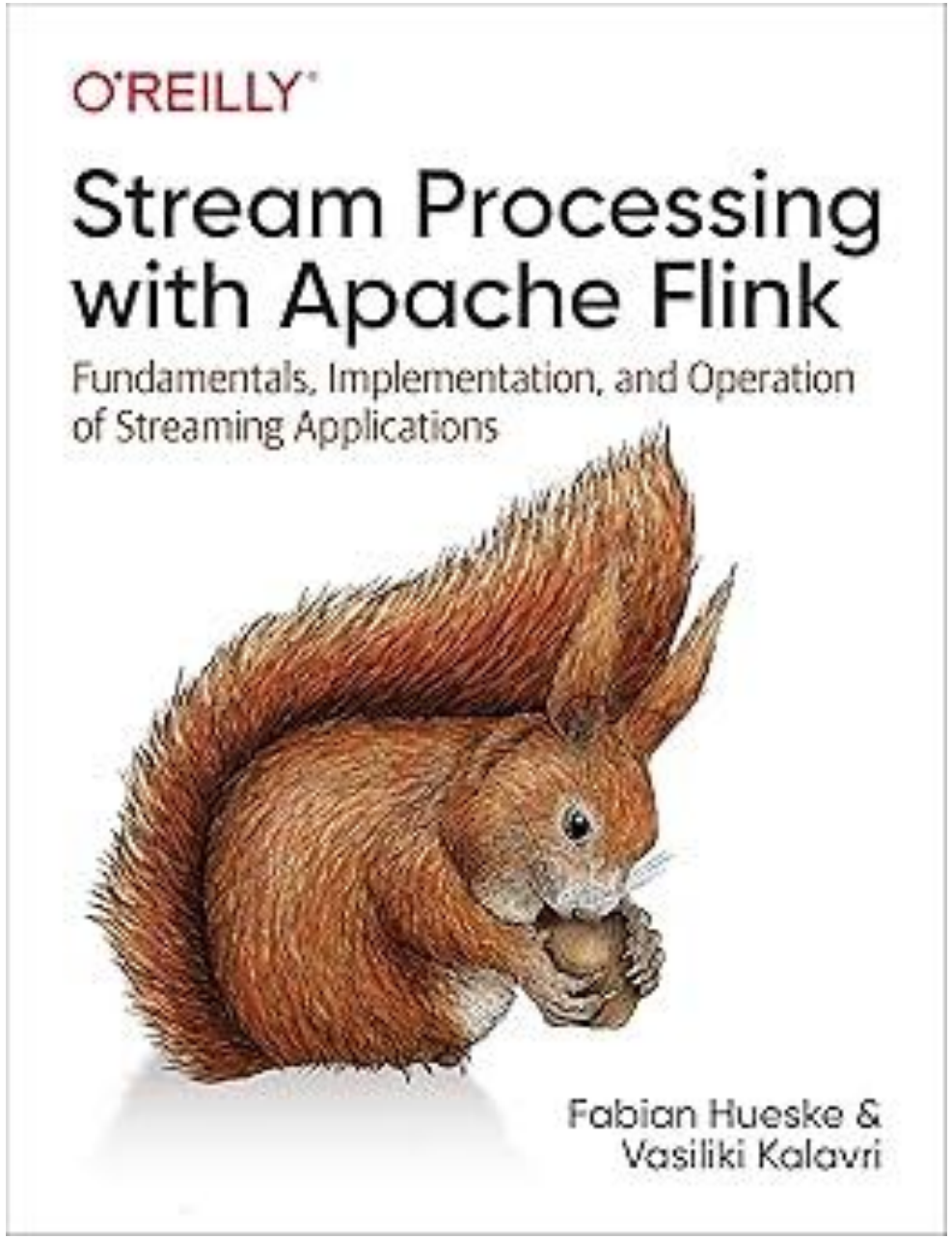
APACHE FLINK BASICS

What is Apache Flink?

Key Features and Components

Basic Architecture

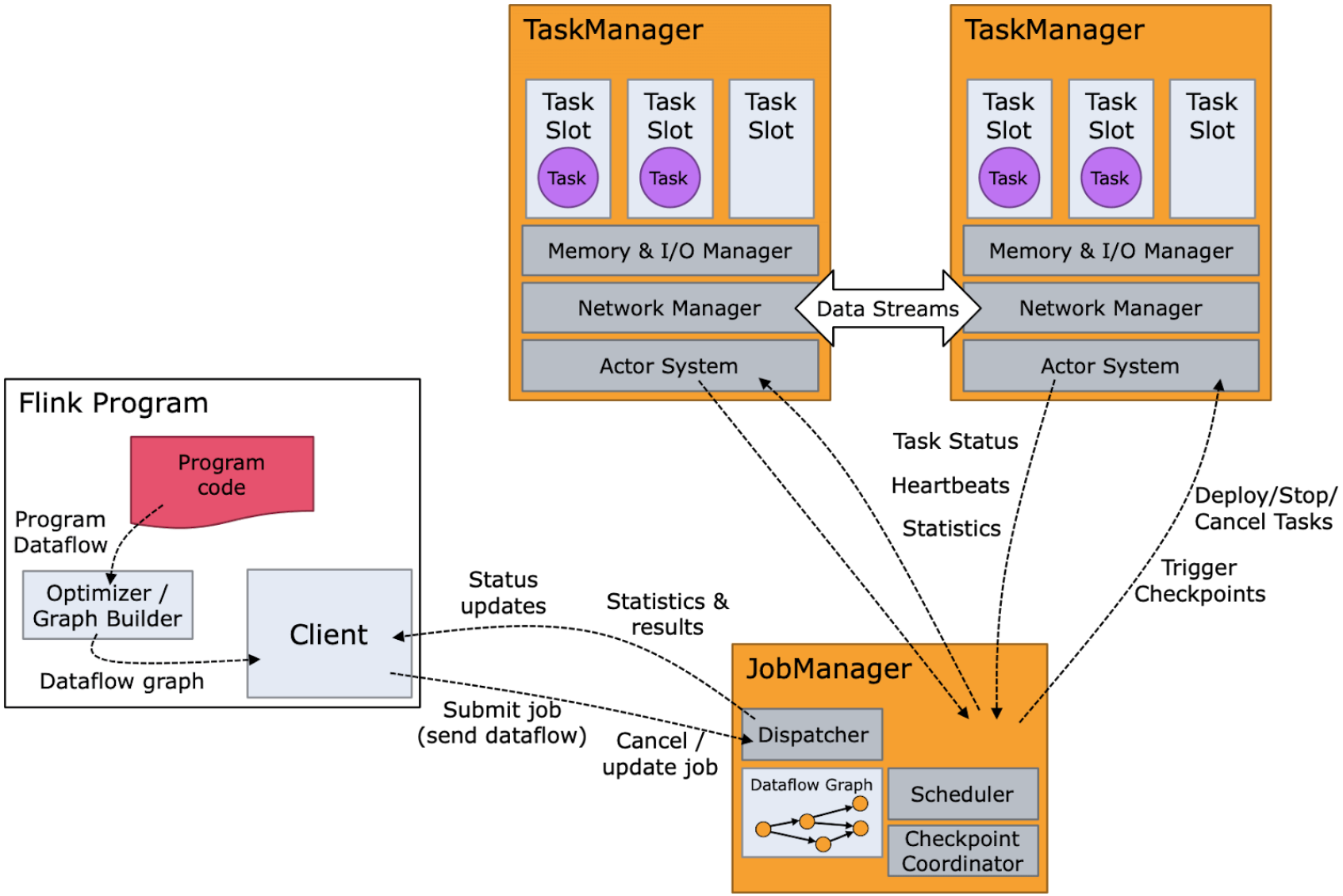
Development Environment Setup



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, and perform computations at in-memory speed, at any scale. Developers build applications for Flink using APIs such as Java or SQL, which are executed on a Flink cluster by the framework.

KEY FEATURES AND COMPONENTS



BASIC ARCHITECTURE – STREAMS & ENVIRONMENT

Apache Flink excels at processing unbounded and bounded data sets. Precise control of time and state enable 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.

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.

KEY FEATURES

Unified Stream and Batch Processing:

Flink provides a unified programming interface for both stream and batch processing, allowing developers to handle real-time and historical data in a single system.

Stateful Computations:

Flink excels at performing stateful computations on data streams, meaning it can maintain and manage data state across different events and time intervals.

Low Latency and High Throughput:

Flink is designed for low-latency, real-time data processing, enabling applications to react quickly to incoming data streams.

Fault Tolerance and Scalability:

Flink is built to be fault-tolerant, meaning it can continue running even if individual nodes fail, and it can scale horizontally to handle large volumes of data.

KEY FEATURES

Exactly-Once Semantics:

Flink provides exactly-once consistency guarantees for state, ensuring that each event is processed exactly once, even in the presence of failures.

Event-Time Processing:

Flink supports event-time processing, which allows applications to process data based on the time of the event itself, rather than the time it was received.

Rich APIs:

Flink offers a variety of APIs, including the DataStream API for stream processing, the Table API for relational data processing, and the SQL API for declarative data processing.

Integration with Common Systems:

Flink integrates well with various data sources and sinks, including Kafka, Hadoop, and cloud storage services.

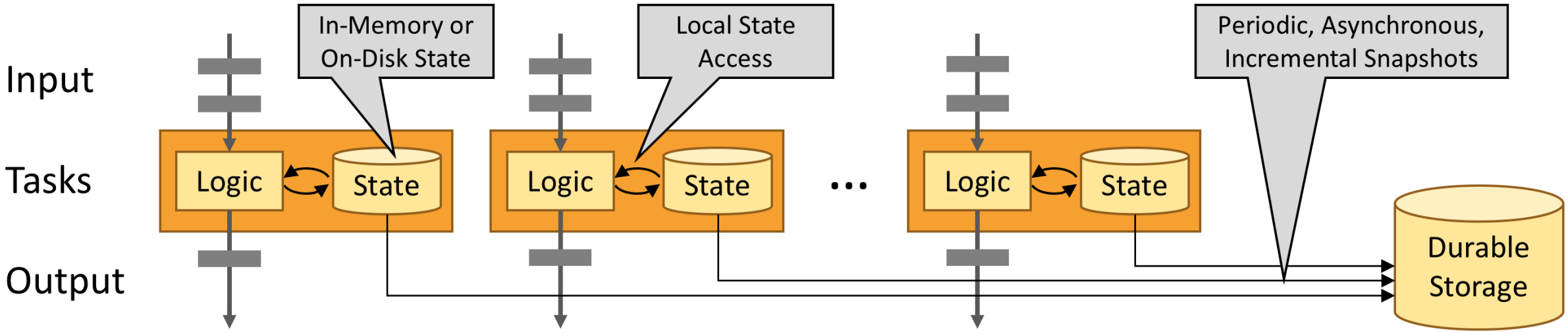
BASIC ARCHITECTURE - SCALE

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.

- applications processing multiple trillions of events per day
- applications maintaining multiple terabytes of state
- applications running on thousands of cores.

BASIC ARCHITECTURE – IN-MEMORY

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.



DEVELOPMENT ENVIRONMENT SETUP

- ***Install Java:*** Flink requires Java 8 or 11, so you need to have one of these versions
- ***Download and Install Apache Flink:*** You can download the latest binary of Apache Flink from the official Flink website.
- ***Start a Local Flink Cluster:*** Start a local Flink cluster using the command. /start-cluster.sh
- ***Set up an Integrated Development Environment (IDE):*** For writing and testing your Flink programs
- ***Create a Flink Project:*** You can create a new Flink using a build tool like Maven or Gradle.

YOUR FIRST FLINK APPLICATION

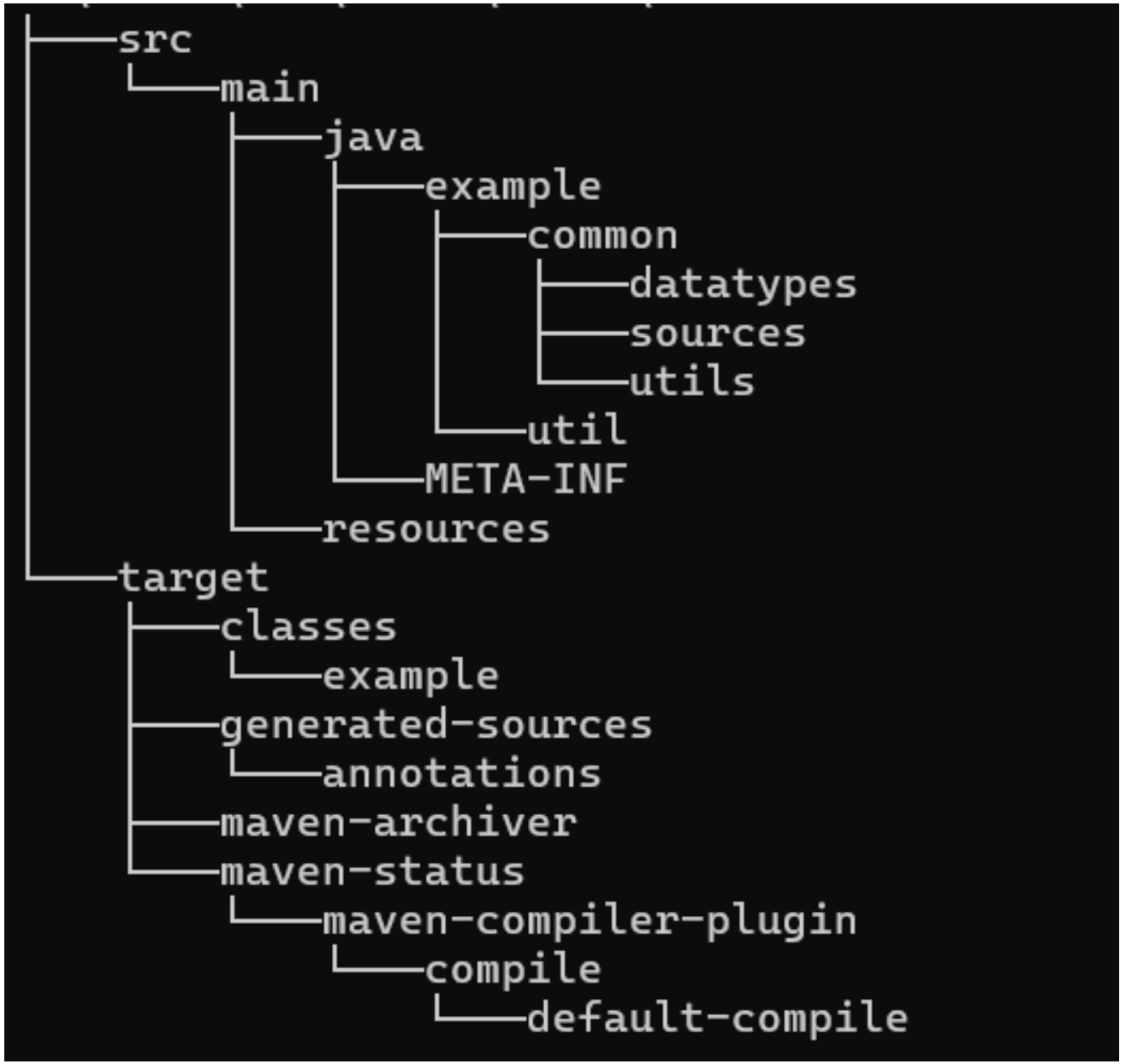
- *Project Structure – Java, Scala or Python. Java for our work*
- *Basic Configuration – easy enough executable Jar*
- *Streaming Example – but that's everything now*
- *Running Local – run a basic cluster with our experiments*

FIRST FLINK APPLICATION ANATOMY

Flink programs look like regular programs that transform DataStreams.
Each program consists of the same basic parts:

- Obtain an execution environment,
- Load/create the initial data,
- Specify transformations on this data,
- Specify where to put the results of your computations,
- Trigger the program execution

PROJECT STRUCTURE



BASIC CONFIGURATION

- *Java 8 or 11*
- *Maven up to 3.8.6*
- *New code in the Apache Flink GitHub requires Java 17, but that won't run in the current LTS Apache Flink Clusters*
- *Code editor – favorites are Eclipse, IntelliJ and VSCode.*
- *Windows, Linux, or MacOS, but we'll have Ubuntu 22 environments for our development cluster. Recommend at least 16GB of RAM.*

```
c:\projects\flink\flink-data-processing-2day>mvn -version
Apache Maven 3.8.6 (84538c9988a25aec085021c365c560670ad80f63)
Maven home: C:\apache-maven-3.8.6
Java version: 11.0.26, vendor: Eclipse Adoptium, runtime: C:\java\jdk-11.0.26+4
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

STREAMING EXAMPLE

```
import ....
public class HelloWorld {
  public static void main(String[] args) throws Exception {
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.fromElements(1, 2, 3, 4, 5)
    .map(i -> 2 * i)
    .print();
  env.execute();
}}
```

FLINK EXECUTION ENVIRONMENT

We always start out with a StreamExecutionEnvironment

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment
();
```

The StreamExecutionEnvironment is the basis for all Flink programs. You can obtain one using these static methods:

- StreamExecutionEnvironment : getExecutionEnvironment();
- createLocalEnvironment();
- createRemoteEnvironment(String host, int port, String… jarFiles);

FLINK EXECUTION ENVIRONMENT

Typically, you only need to use getExecutionEnvironment(), since this will do the right thing depending on the context: if you are executing your program inside an IDE or as a regular Java program it will create a local environment that will execute your program on your local machine. If you created a JAR file from your program, and invoke it through the command line, the Flink cluster manager will execute your main method and getExecutionEnvironment() will return an execution environment for executing your program on a cluster.

For specifying data sources the execution environment has several methods to read from files using various methods: you can just read them line by line, as CSV files, or using any of the other provided sources.

BUILT-IN DATASOURCES

Flink provides special data sources which are backed by Java collections to ease testing.

// Create a DataStream from a list of elements

```
DataStream<Integer> myInts = env.fromElements(1, 2, 3, 4, 5);
```

// Create a DataStream from any Java collection

```
List<Tuple2<String, Integer>> data = ...

DataStream<Tuple2<String, Integer>> myTuples =
env.fromCollection(data);
```

// Create a DataStream from an Iterator

```
Iterator<Long> longIt = ...;

DataStream<Long> myLongs = env.fromCollection(longIt,
Long.class);
```

MAP TRANSFORMATION

In Apache Flink, a "map" refers to a transformation operation on a DataStream or DataSet where a user-defined function is applied to each element, producing a one-to-one mapping. The map function transforms each input element into exactly one output element.

Key aspects of Flink's map transformation:

- One-to-one mapping:

Each input element is processed, and a single output element is produced for it.

- User-defined function:

The map operation requires a user-defined function (like a MapFunction) to specify the transformation logic.

```
.map(i -> 2 * i) // multiply each input element by 2
```

PRINT OPERATION

The print operation can be used to write to stdout which we do in our HelloWorld example, or to the invoker which we will see in a number of our experiments.

```
.print();
```

For the debugging with standard out we can see that in the log folder under the flink installation. The files are named in the format

flink-environment-taskexecutor-n-ip-xx.yy.zz.ww.out

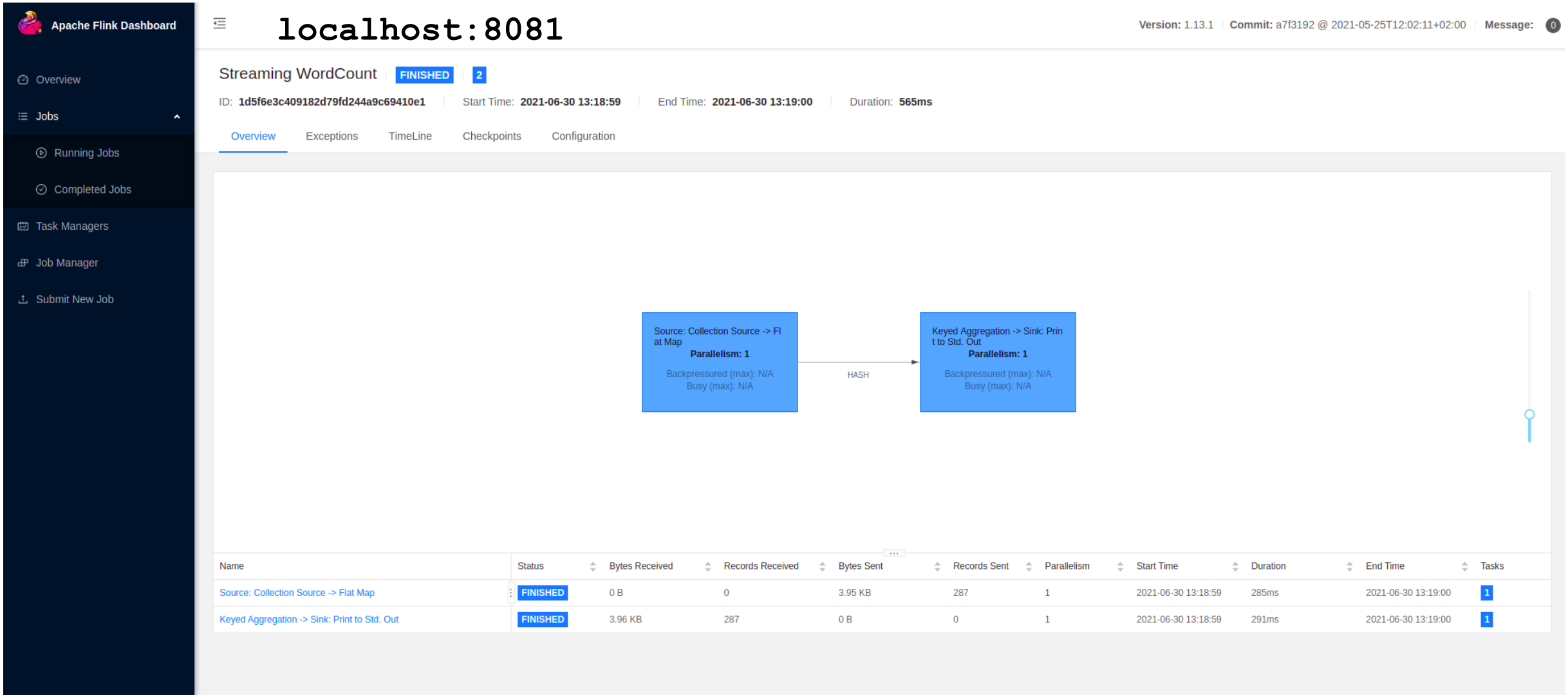
EXECUTION

Once you specified the complete program you need to trigger the program execution by calling execute() on the StreamExecutionEnvironment. Depending on the type of the ExecutionEnvironment the execution will be triggered on your local machine or submit your program for execution on a cluster.

The execute() method will wait for the job to finish and then return a JobExecutionResult, this contains execution times and accumulator results.

```
env.execute();
```

FLINK WEBUI


Apache Flink Dashboard
localhost:8081
Version: 1.13.1 | Commit: a7f3192 @ 2021-05-25T12:02:11+02:00 | Message: 0
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
Streaming WordCount FINISHED 2
ID: 1d5f6e3c409182d79fd244a9c69410e1 | Start Time: 2021-06-30 13:18:59 | End Time: 2021-06-30 13:19:00 | Duration: 565ms
Overview
Exceptions
TimeLine
Checkpoints
Configuration
Source: Collection Source -> Flat Map
Parallelism: 1
Backpressured (max): N/A
Busy (max): N/A
HASH
Keyed Aggregation -> Sink: Print to Std. Out
Parallelism: 1
Backpressured (max): N/A
Busy (max): N/A
Name | Status | Bytes Received | Records Received | Bytes Sent | Records Sent | Parallelism | Start Time | Duration | End Time | Tasks
Source: Collection Source -> Flat Map | FINISHED | 0 B | 0 | 3.95 KB | 287 | 1 | 2021-06-30 13:18:59 | 285ms | 2021-06-30 13:19:00 | 1
Keyed Aggregation -> Sink: Print to Std. Out | FINISHED | 3.96 KB | 287 | 0 B | 0 | 1 | 2021-06-30 13:18:59 | 291ms | 2021-06-30 13:19:00 | 1

Q&A AND OPEN DISCUSSION

Working with DataStreams

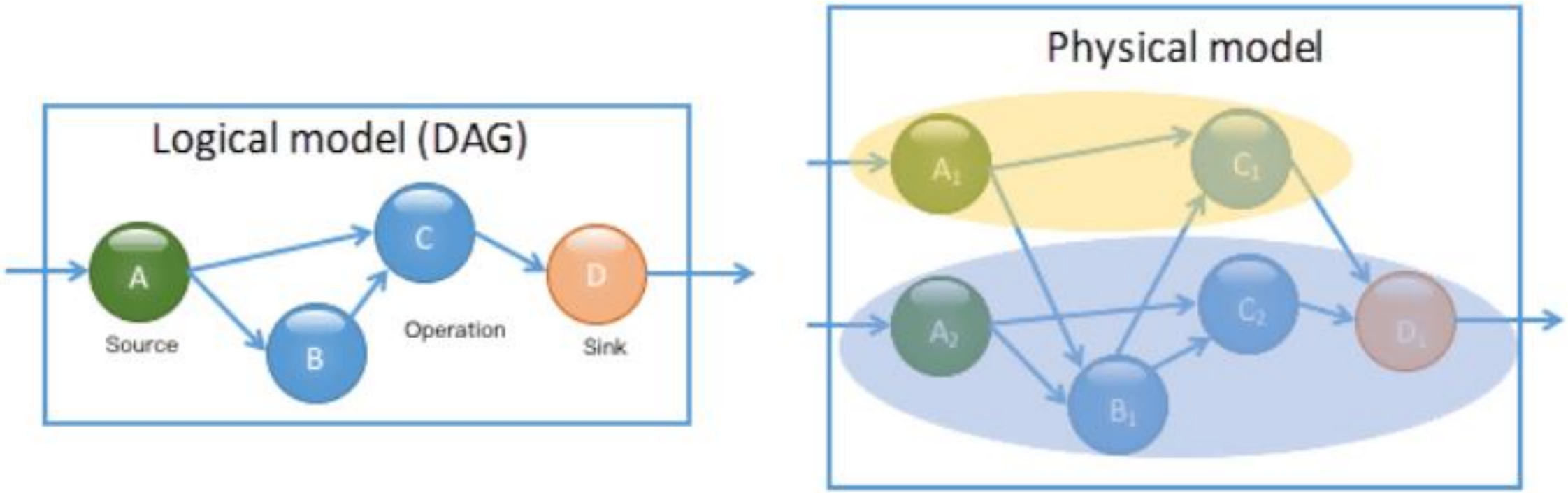
Working with DataStreams

- DataStream Basics
  - Creating DataStreams
  - Basic Operations
  - Data Types
  - Simple Transformations
- Common Operations
  - Map and FlatMap
  - Filter Operations
  - Basic Aggregations
  - Field Selection

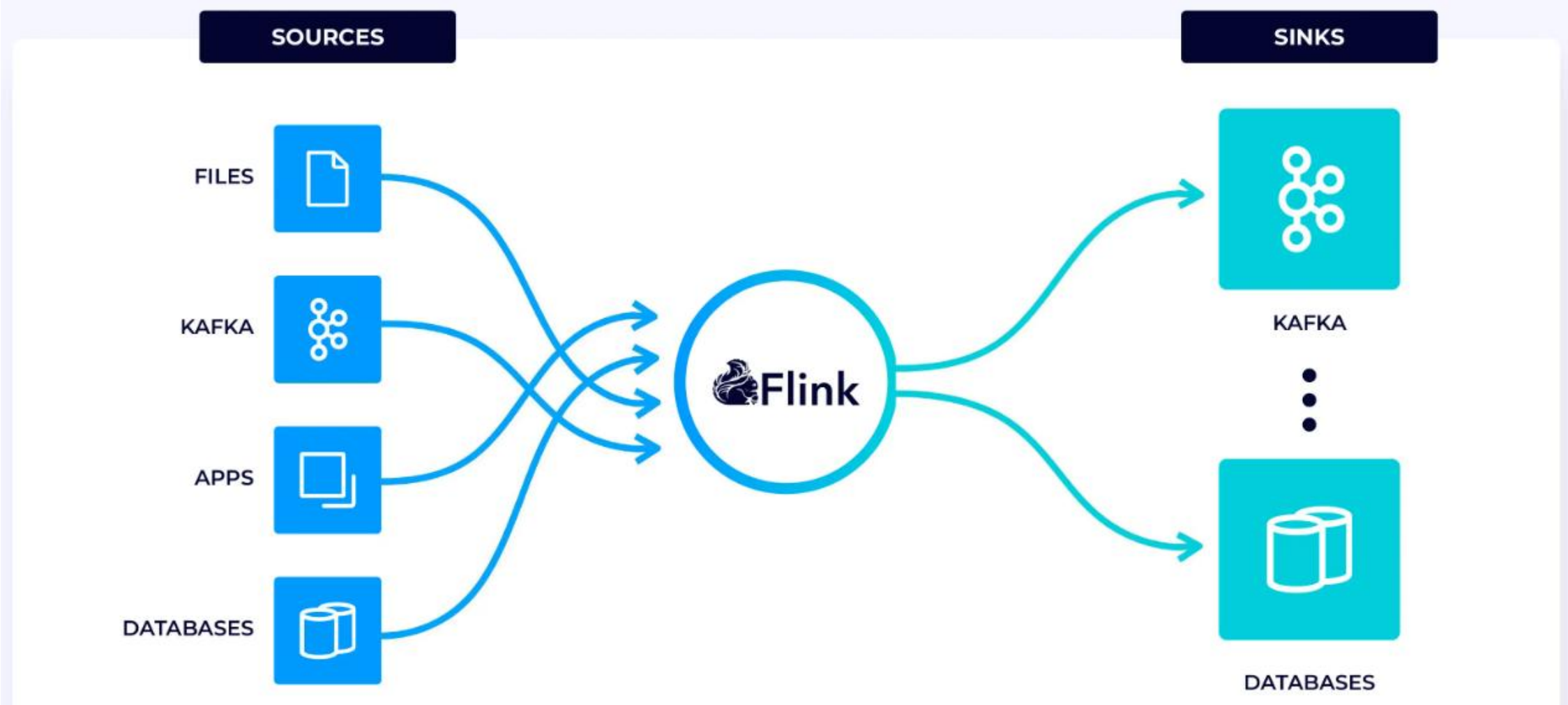
DATASTREAM BASICS

DataStream programs in Flink are regular programs that implement transformations on data streams (e.g., filtering, updating state, defining windows, aggregating). The data streams are initially created from various sources (e.g., message queues, socket streams, files).

DATASTREAM BASICS



DATASTREAM BASICS



DATASTREAMS

- The DataStream API gets its name from the special DataStream class that is used to represent a collection of data in a Flink program. You can think of them as immutable collections of data that can contain duplicates. This data can either be finite or unbounded, the API that you use to work on them is the same.

- A DataStream is similar to a regular Java Collection in terms of usage but is quite different in some key ways. They are immutable, meaning that once they are created you cannot add or remove elements. You can also not simply inspect the elements inside but only work on them using the DataStream API operations, which are also called transformations.

- You can create an initial DataStream by adding a source in a Flink program. Then you can derive new streams from this and combine them by using API methods such as map, filter, and so on.

CREATING DATASTREAMS

The following example creates a datastream from a text file by reading all the lines.

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();


FileSource<String> fileSource = FileSource.forRecordStreamFormat(
        new TextLineInputFormat(), new Path("file:///path/to/file")
    ).build();
DataStream<String> text = env.fromSource(
    fileSource,
    WatermarkStrategy.noWatermarks(),
    "file-input");
```

CREATING DATASTREAMS FROM DATASTREAMS

The following example takes a DataStream and then does a transformation to make a new DataStream. In this case taking all the string numbers in the input stream and making them Integers. DataStreams are **immutable.**

```
DataStream<String> input = ...;

DataStream<Integer> parsed = input.map(new
MapFunction<String, Integer>() {

    @Override

    public Integer map(String value) {

        return Integer.parseInt(value);

    }});
```

DATASTREAMS

DataStream in Flink represents a stream of data that is continuously generated from a source, such as a message queue or a sensor network. DataStreams are processed in a distributed fashion across the Flink cluster, allowing the system to handle large-scale, high-throughput data streams efficiently.

- SocketStream: Reading from socket connections.
- FileStream: Reading from files.
- Kafka: Reading from Kafka topics.
- Custom Source Functions: Defining custom data sources.

DATASTREAMS V2

DataStream programs in Flink are regular programs that implement transformations on data streams (e.g., filtering, updating state, defining windows, aggregating). The data streams are initially created from various sources (e.g., message queues, socket streams, files). Results are returned via sinks, which may for example write the data to files, or to standard output (for example the command line terminal). Flink programs run in a variety of contexts, standalone, or embedded in other programs. The execution can happen in a local JVM, or on clusters of many machines.

DataStream API V2 is a new set of APIs, to gradually replace the original DataStream API. It is currently in the experimental stage and is not fully available for production.

DATASTREAM OPERATIONS

1. **DataStream Transformations:** Map, FlatMap, Filter, Windowing, …
2. **State Management:** state, keyed state and operator state
3. **Failure Recovery and Fault Tolerance:** checkpointing and exactly-once processing
4. **SQL and Table API:** SQL-like queries and operations on tables, and stream-table duality
5. **Other Operations:** sources, sinks, and iterative streams

DATASTREAM OPERATIONS

```
DataStream<Transaction> transactions = // source of transactions;


// Flagging large transactions
DataStream<Alert> alerts = transactions
    .keyBy(Transaction::getAccountId)
    .timeWindow(Duration.ofMinutes(1))
    .apply(new FraudDetectionFunction());
// FraudDetectionFunction implementation
public class FraudDetectionFunction extends ProcessWindowFunction<Transaction, Alert, String,
TimeWindow> {
    @Override
    public void process(String accountId, Context context, Iterable<Transaction> transactions,
Collector<Alert> out) {
        for (Transaction transaction : transactions) {
            if (transaction.getAmount() > 10000) {
                out.collect(new Alert(accountId, transaction.getAmount(), "Potential fraud
detected"));
```

DATATYPES

There are seven different categories of data types:

- Java Tuples and Scala Case Classes
- Java POJOs
- Primitive Types
- Regular Classes
- Values
- Hadoop Writables
- Special Types

TYPE HANDLING

Flink tries to infer a lot of information about the data types that are exchanged and stored during the distributed computation. Think about it like a database that infers the schema of tables. In most cases, Flink infers all necessary information seamlessly by itself. Having the type information allows Flink to do some cool things:

The more Flink knows about data types, the better the serialization and data layout schemes are. That is quite important for the memory usage paradigm in Flink (work on serialized data inside/outside the heap where ever possible and make serialization very cheap).

Finally, it also spares users in the majority of cases from worrying about serialization frameworks and having to register types.

In general, the information about data types is needed during the pre-flight phase - that is, when the program's calls on DataStream are made, and before any call to execute(), print(), count(), or collect().

DATASTREAM TRANSFORMATIONS

Map: Applies a function to each element in the stream.

FlatMap: Similar to Map but can return zero, one, or more elements.

Filter: Filters elements based on a predicate.

KeyBy: Groups the stream by a key, enabling distributed state management.

Reduce: Combines elements in the stream using a reduce function.

Window: Groups elements in the stream into windows for batch processing.

Join: Joins two streams based on a common key.

Union: Combines two or more streams into a single stream.

COMMON OPERATIONS

Map and FlatMap - executed for each element in the stream or the set

Filter Operations - transformations that select elements from a stream or dataset based on a condition

Basic Aggregations - computing a single result from multiple input rows

Field Selection - process of extracting specific fields or attributes

MAP

Take one element of a stream and produce one element. The following doubles the integer value of each element in the input data stream.

```
DataStream<Integer> dataStream = //...
dataStream.map(new MapFunction<Integer, Integer>() {
    @Override
    public Integer map(Integer value) throws Exception {
        return 2 * value;
    }});
```

FLATMAP

Takes one element and produces zero, one, or more elements. The following example is a flatmap function that splits sentences to words.

```
dataStream.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String value, Collector<String>
out)
        throws Exception {
        for(String word: value.split(" ")){
            out.collect(word);
        }}});
```

FILTER OPERATIONS

The filter operation, applied through DataStream.filter(), uses a user-defined function that returns a boolean. If the function returns true for an element, it's included in the resulting stream; otherwise, it's discarded. The following filters out zero values

```
dataStream.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) throws Exception {
        return value != 0;
    }});
```

FILTER OPERATIONS

Purpose:

The primary goal of a filter is to refine a data stream by removing unwanted elements.

Implementation:

The filter() method takes a Predicate (a functional interface) as an argument. This Predicate is applied to each element of the input stream.

Function Return Value:

The Predicate must return true or false for each element.

Result:

The filter transformation creates a new stream containing only the elements for which the Predicate returned true.

BASIC AGGREGATIONS

Window Aggregation: Window aggregations are defined in the GROUP BY clause contains “window_start” and “window_end” columns of the relation applied Windowing TVF. Just like queries with regular GROUP BY clauses, queries with a group by window aggregation will compute a single result row per group.

Group Aggregation: Apache Flink supports aggregate functions; both built-in and user-defined. User-defined functions must be registered in a catalog before use. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the COUNT, SUM, AVG (average), MAX (maximum) and MIN (minimum) over a set of rows.

Over Aggregation: OVER aggregates compute an aggregated value for every input row over a range of ordered rows. In contrast to GROUP BY aggregates, OVER aggregates do not reduce the number of result rows to a single row for every group. Instead OVER aggregates produce an aggregated value for every input row.

WINDOW AGGREGATION

The following is an example of a tumbling window aggregation:

```
Flink SQL> SELECT window_start, window_end, SUM(price) AS
total_price
  FROM TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10'
MINUTES)
  GROUP BY window_start, window_end;
+------------------+------------------+-------------+
|     window_start |       window_end | total_price |
+------------------+------------------+-------------+
| 2020-04-15 08:00 | 2020-04-15 08:10 |       11.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 |       10.00 |
+------------------+------------------+-------------+
```

GROUP AGGREGATION

An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the COUNT, SUM, AVG (average), MAX (maximum) and MIN (minimum) over a set of rows.

```
SELECT COUNT(*) FROM Orders
```

For streaming queries, it is important to understand that Flink runs continuous queries that never terminate. Instead, they update their result table according to the updates on its input tables. For the above query, Flink will output an updated count each time a new row is inserted into the Orders table.

GROUPING SETS

Grouping sets allow for more complex grouping operations than those describable by a standard GROUP BY. Rows are grouped separately by each specified grouping set and aggregates are computed for each group just as for simple GROUP BY clauses.

```
SELECT supplier_id, rating, COUNT(*) AS total
FROM (VALUES
    ('supplier1', 'product1', 4),
    ('supplier1', 'product2', 3),
    ('supplier2', 'product3', 3),
    ('supplier2', 'product4', 4))
AS Products(supplier_id, product_id, rating)
GROUP BY GROUPING SETS ((supplier_id, rating), (supplier_id), ())
```

OVER AGGREGATION

The following query computes for every order the sum of amounts of all orders for the same product that were received within one hour before the current order. Remember that OVER aggregates produce for every input row

```
SELECT order_id, order_time, amount,
  SUM(amount) OVER (
    PARTITION BY product
    ORDER BY order_time
    RANGE BETWEEN INTERVAL '1' HOUR PRECEDING AND CURRENT ROW
  ) AS one_hour_prod_amount_sum
FROM Orders
```

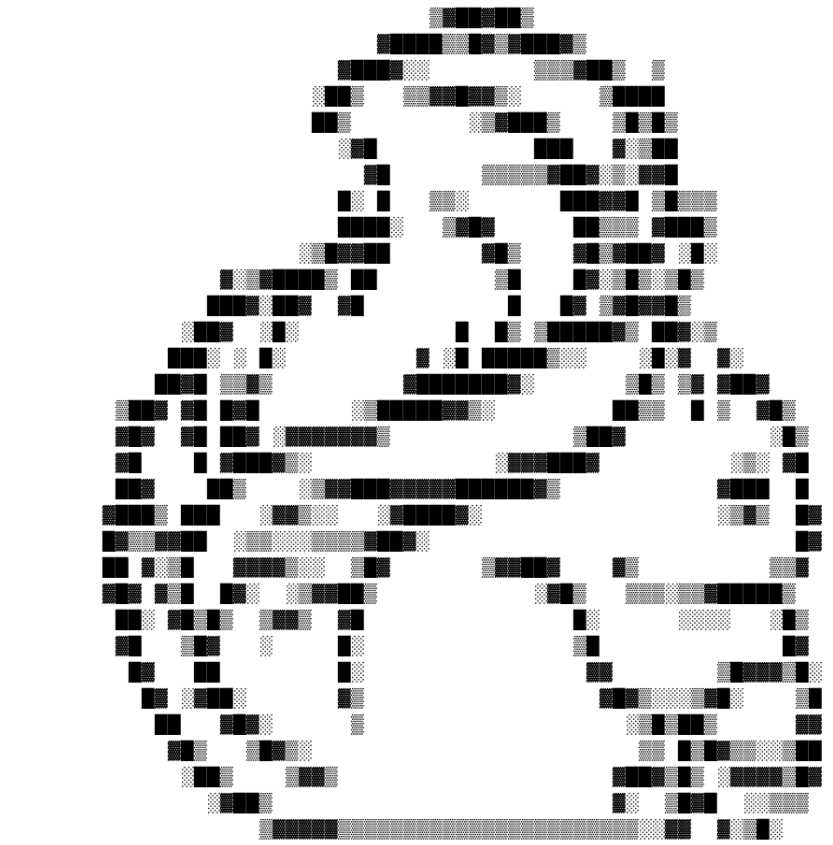
FLINK SQL CLIENT

Bundled in the regular Flink distribution and thus runnable out-of-the-box. Requires a Flink cluster which you start with:

```
./bin/start-cluster.sh
```

The SQL Client scripts are also located in the bin directory of Flink. Users have two options for starting the SQL Client CLI, either by starting an embedded standalone process or by connecting to a remote SQL Gateway. Start the embedded mode client in our dev server environments with:

```
./bin/sql-client.sh
```



FLINK SQL CLIENT VALIDATION

For validating your setup and cluster connection, you can enter the simple query below and press Enter to execute it.

```
SET 'sql-client.execution.result-mode' = 'tableau';
SET 'execution.runtime-mode' = 'batch';

SELECT
  name,
  COUNT(*) AS cnt
FROM
  (VALUES ('Bob'), ('Alice'), ('Greg'), ('Bob')) AS NameTable(name)
GROUP BY name;
```

Q&A AND OPEN DISCUSSION

Data Sources and Sinks

Data Sources and Sinks

- Built-in Sources
  - File-based Sources
  - Socket Sources
  - Collection Sources
  - Generating Test Data
- Built-in Sinks
  - File Sinks
  - Print Sink
  - Socket Sink
  - Common Formats

BUILT-IN SOURCES

Apache Flink offers several built-in sources for ingesting data into stream processing pipelines. These include reading from files and directories, ingesting data from collections and iterators, and connecting to sockets. Furthermore, Flink provides connectors for reading data from various external.

- File and Directory Sources
- Collection and Iterator Sources
- Socket Source
- Streaming Platform Sources (e.g., Kafka, Kinesis)

FILE-BASED SOURCES

Apache Flink uses file systems to consume and persistently store data, both for the results of applications and for fault tolerance and recovery. These are some of most of the popular file systems, including local, hadoop-compatible, Amazon S3, Aliyun OSS and Azure Blob Storage.

The file system used for a particular file is determined by its URI scheme. For example, **file:///home/user/text.txt** refers to a file in the local file system, while **hdfs://namenode:50010/data/user/text.txt** is a file in a specific HDFS cluster.

File system instances are instantiated once per process and then cached/pooled, to avoid configuration overhead per stream creation and to enforce certain constraints, such as connection/stream limits.

FILE-BASED SOURCES

fromSource(FileSource.forRecordStreamFormat(format, paths).build()) - Read record-by-record from files.

readFile(fileInputFormat, path) - Reads (once) files as dictated by the specified file input format

readFile(fileInputFormat, path, watchType, interval, pathFilter, typeInfo) - This is the method called internally by the two previous ones. It reads files in the path based on the given fileInputFormat. Depending on the provided watchType, this source may periodically monitor (every interval ms) the path for new data (FileProcessingMode.PROCESS_CONTINUOUSLY), or process once the data currently in the path and exit (FileProcessingMode.PROCESS_ONCE). Using the pathFilter, the user can further exclude files from being processed.

FILE-BASED SOURCES ANATOMY

Under the hood, Flink splits the file reading process into two sub-tasks, namely directory monitoring and data reading. Each of these sub-tasks is implemented by a separate entity. Monitoring is implemented by a single, non-parallel (parallelism = 1) task, while reading is performed by multiple tasks running in parallel. The parallelism of the latter is equal to the job parallelism. The role of the single monitoring task is to scan the directory (periodically or only once depending on the watchType), find the files to be processed, divide them in splits, and assign these splits to the downstream readers. The readers are the ones who will read the actual data. Each split is read by only one reader, while a reader can read multiple splits, one-by-one.

FILE-BASED SOURCES WATCHTYPE

If the watchType is set to **FileProcessingMode.PROCESS_CONTINUOUSLY**, when a file is modified, its contents are re-processed entirely. This can break the "exactly-once" semantics, as appending data at the end of a file will lead to all its contents being re-processed.

If the watchType is set to **FileProcessingMode.PROCESS_ONCE**, the source scans the path once and exits, without waiting for the readers to finish reading the file contents. The readers will continue reading until all file contents are read. Closing the source leads to no more checkpoints after that point. This may lead to slower recovery after a node failure, as the job will resume reading from the last checkpoint.

LOCAL FILE SYSTEMS

Flink has built-in support for the file system of the local machine, including any NFS or SAN drives mounted into that local file system. It can be used by default without additional configuration. Local files are referenced with the file:// URI scheme.

Other file system types are accessed by an implementation that bridges to the suite of file systems supported by Apache Hadoop, those are the pluggable file systems.

PLUGGABLE FILE SYSTEMS

Amazon S3 object storage is supported by two alternative implementations: flink-s3-fs-presto and flink-s3-fs-hadoop.

Aliyun Object Storage Service is supported by flink-oss-fs-hadoop and registered under the oss:// URI scheme.

Azure Data Lake Store Gen2 is supported by flink-azure-fs-hadoop and registered under the abfs(s):// URI schemes.

Azure Blob Storage is supported by flink-azure-fs-hadoop and registered under the wasb(s):// URI schemes.

Google Cloud Storage is supported by gcs-connector and registered under the gcs:// URI scheme.

SOCKET SOURCES

Flink can be used with sockets. It provides built-in support for both reading data from a socket (source) and writing data to a socket (sink). Specifically, Flink offers the socketTextStream API for reading data from a socket and the writeToSocket API for writing data to a socket. These APIs allow you to easily connect Flink applications to socket-based data streams for processing and analysis.

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

SOCKET SOURCE MECHANICS

Reading from a Socket:

The socketTextStream API allows Flink to read data from a TCP socket as a stream of text lines. You specify the hostname and port of the socket to connect to, and Flink will then continuously receive data from that socket.

For example:

You could use a Flink job to read stock prices from a socket, perform some analysis, and then write the results to another socket for other applications to consume.

COLLECTION SOURCES

In Apache Flink, collection sources are a way to create DataStreams from Java collections for testing and development purposes. They allow you to easily ingest data from in-memory lists, iterators, or other collections into a Flink pipeline without needing to rely on external data sources like files or databases.

Flink provides a simple API to create DataStreams from collections using methods like fromElements(), fromCollection(), and fromCollection(iterator, Class).

Not for Production: While useful for development and testing, collection sources are not intended for production deployments, as they don't scale or handle large amounts of data efficiently.

COLLECTION SOURCES

```
public class CollectionSourceExample {
        public static void main(String[] args) throws Exception {
          StreamExecutionEnvironment env =
StreamExecutionEnvironment.createLocalEnvironment();
          // Create a DataStream from a list of elements
          DataStream<Integer> intStream = env.fromElements(1,2,3,4,5);
          // Create a DataStream from a Java Collection
          List<String> stringList = Arrays.asList("Hello", "Flink");
          DataStream<String> stringStream =
env.fromCollection(stringList);
          // Process the DataStreams
          stringStream.print();
          env.execute(); } }
```

CUSTOM SOURCES

addSource - Attach a new source function. For example, to read from Apache Kafka you can use addSource(new FlinkKafkaConsumer<>(...)).

- Apache Kafka (source/sink)
- Apache Cassandra (source/sink)
- Amazon Kinesis DS (source/sink)
- DataGen (source)
- FileSystem (source/sink)
- RabbitMQ (source/sink)
- Google PubSub (source/sink)
- Hybrid Source (source)
- Apache Pulsar (source)
- MongoDB (source/sink)
- Prometheus (sink)
- Apache ActiveMQ (source/sink)
- Netty (Source)

GENERATING TEST DATA

We need to expect that for any data engineering pipeline that test data will need to be generated. In our experiments we'll look at doing exactly that for Taxi Fares and Taxi Rides.

```
 /** Create a bounded TaxiFareGenerator that runs only for the specified duration.
*/
    public static TaxiFareGenerator runFor(Duration duration) {
        TaxiFareGenerator generator = new TaxiFareGenerator();
        generator.limitingTimestamp = DataGenerator.BEGINNING.plus(duration);
        return generator;
    }
```

BUILT-IN AND CUSTOM SINKS

- Apache Kafka (source/sink)
- Apache Cassandra (source/sink)
- Amazon DynamoDB (sink)
- Amazon Kinesis Data Streams (source/sink)
- Amazon Kinesis Data Firehose (sink)
- Elasticsearch (sink)
- Opensearch (sink)
- FileSystem (source/sink) – Built-in including stdout, stderr
- RabbitMQ (source/sink)
- Google PubSub (source/sink)
- JDBC (sink)
- MongoDB (source/sink)
- Prometheus (sink)
- Apache ActiveMQ (source/sink)
- Apache Flume (sink)
- Redis (sink)
- Akka (sink)

DATA SINKS

Data sinks consume DataStreams and forward them to files, sockets, external systems, or print them. Flink comes with a variety of built-in output formats that are encapsulated behind operations on the DataStreams:

sinkTo(FileSink.forRowFormat(new Path("outputPath"), new SimpleStringEncoder<>()).build()) - Writes elements line-wise as Strings. The Strings are obtained by calling the toString() method of each element.

print() / printToErr() - Prints the toString() value of each element on the standard out / standard error stream. Optionally, a prefix (msg) can be provided which is prepended to the output. This can help to distinguish between different calls to print. If the parallelism is greater than 1, the output will also be prepended with the identifier of the task which produced the output.

writeUsingOutputFormat() / FileOutputFormat - Method and base class for custom file outputs. Supports custom object-to-bytes conversion.

writeToSocket - Writes elements to a socket according to a SerializationSchema

addSink - Invokes a custom sink function. Flink comes bundled with connectors to other systems (such as Apache Kafka) that are implemented as sink functions.

FILE SINKS

Apache Flink uses file systems to consume and persistently store data, both for the results of applications and for fault tolerance and recovery. These are some of most of the popular file systems, including local, hadoop-compatible, Amazon S3, Aliyun OSS and Azure Blob Storage.

The file system used for a particular file is determined by its URI scheme. For example, **file:///home/user/text.txt** refers to a file in the local file system, while **hdfs://namenode:50010/data/user/text.txt** is a file in a specific HDFS cluster.

File system instances are instantiated once per process and then cached/pooled, to avoid configuration overhead per stream creation and to enforce certain constraints, such as connection/stream limits.

PRINT SINK

In our experiments we'll use .print() to print results to the task manager logs (which will appear in your IDE's console, when running in an IDE). This will call toString() on each element of the stream. In SQL these .print() are to our invoker output.

For example, our output could look something like this:

1> Fred: age 35

2> Wilma: age 35

where 1> and 2> indicate which sub-task (i.e., thread) produced the output.

In production, commonly used sinks include the FileSink, various databases, and several pub-sub systems.

SOCKET SINK MECHANICS

Writing to a Socket:

The writeToSocket API allows Flink to write data to a TCP socket. You specify the hostname, port, and the SerializationSchema to be used for converting the data to a byte stream before sending it over the socket.

For example:

You could use a Flink job to read stock prices from a socket, perform some analysis, and then write the results to another socket for other applications to consume.

COMMON FORMATS

Common Flink sink formats include file-based formats like CSV, Avro, Parquet, and ORC, as well as connectors for external systems like Kafka, HBase, Hive, Kinesis, and DynamoDB. Flink also supports custom connectors for integrating with other data sources and sinks.

File-based Formats:

Avro: A data serialization format known for its efficiency and flexibility. Flink provides built-in support for writing data into Avro files.

Parquet: A columnar storage format optimized for analytical queries. Flink offers Parquet writer factories for various use cases, including Avro data.

CSV: A simple, human-readable text format for storing tabular data.

ORC: (Optimized Row Columnar) Another columnar storage format designed for large datasets. Flink supports ORC sinks as a bulk-encoded format.


MySQL
Amazon Aurora
SQL Server
mongoDB
yugabyteDB
HIVE
Amazon S3
Google Sheets
salesforce
zendesk
Google Ads
Flink CDC

Q&A AND OPEN DISCUSSION

Time and Windows

Time and Windows

- Understanding Time
  - Event Time vs Processing
- Time
  - Timestamps
  - Watermark Basics
  - Dealing with Late Events
- Window Operations
  - Types of Windows
  - Tumbling Windows
  - Sliding Windows
  - Session Windows

UNDERSTANDING TIME

Timely stream processing is an extension of stateful stream processing in which time plays some role in the computation. Among other things, this is the case when you do time series analysis, when doing aggregations based on certain time periods (typically called windows), or when you do event processing where the time when an event occurred is important.

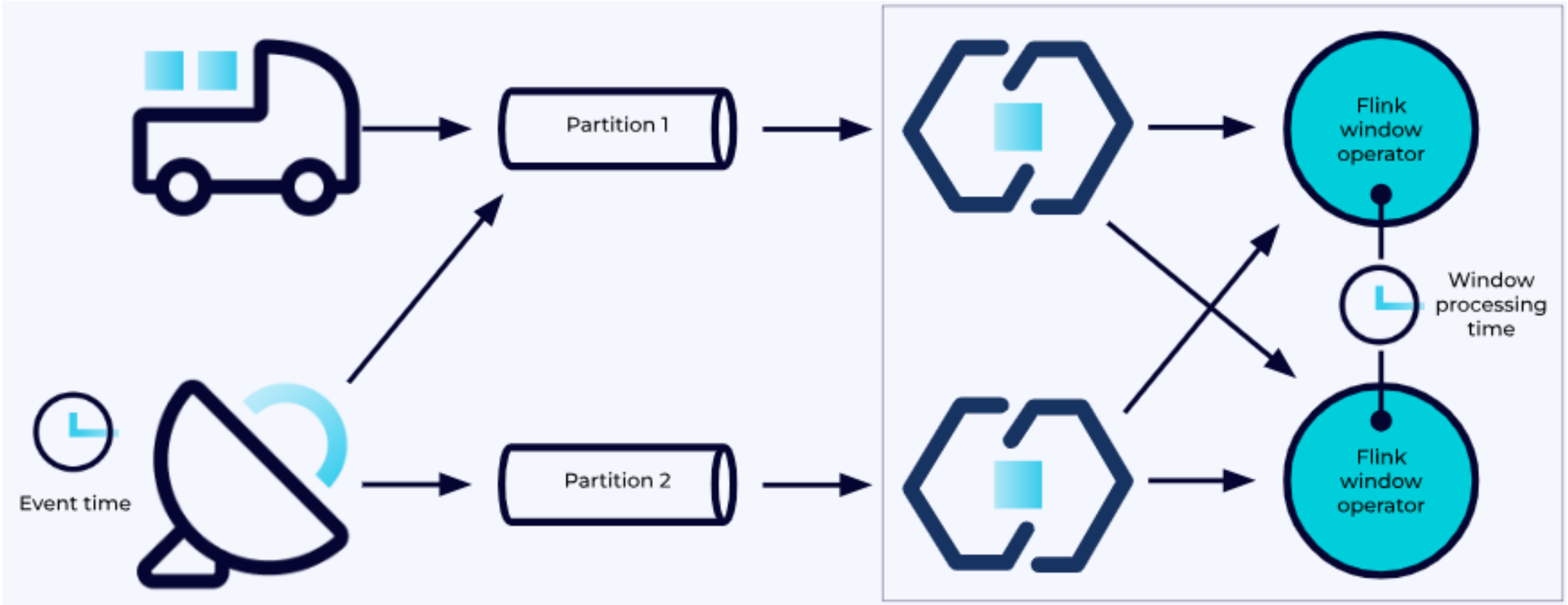
When referring to time in a streaming program (for example to define windows), one can refer to different notions of time, event and processing.

EVENT TIME VS PROCESSING TIME

Event time: The time that each individual event occurred on its producing device. This time is typically embedded within the records before they enter Flink, and that event timestamp can be extracted from each record. In event time, the progress of time depends on the data, not on any wall clocks.

Processing time: This time refers to the system time of the machine that is executing the respective operation.

EVENT TIME VS PROCESSING TIME



EVENT TIME

Event time programs must specify how to generate Event Time Watermarks, which is the mechanism that signals progress in event time.

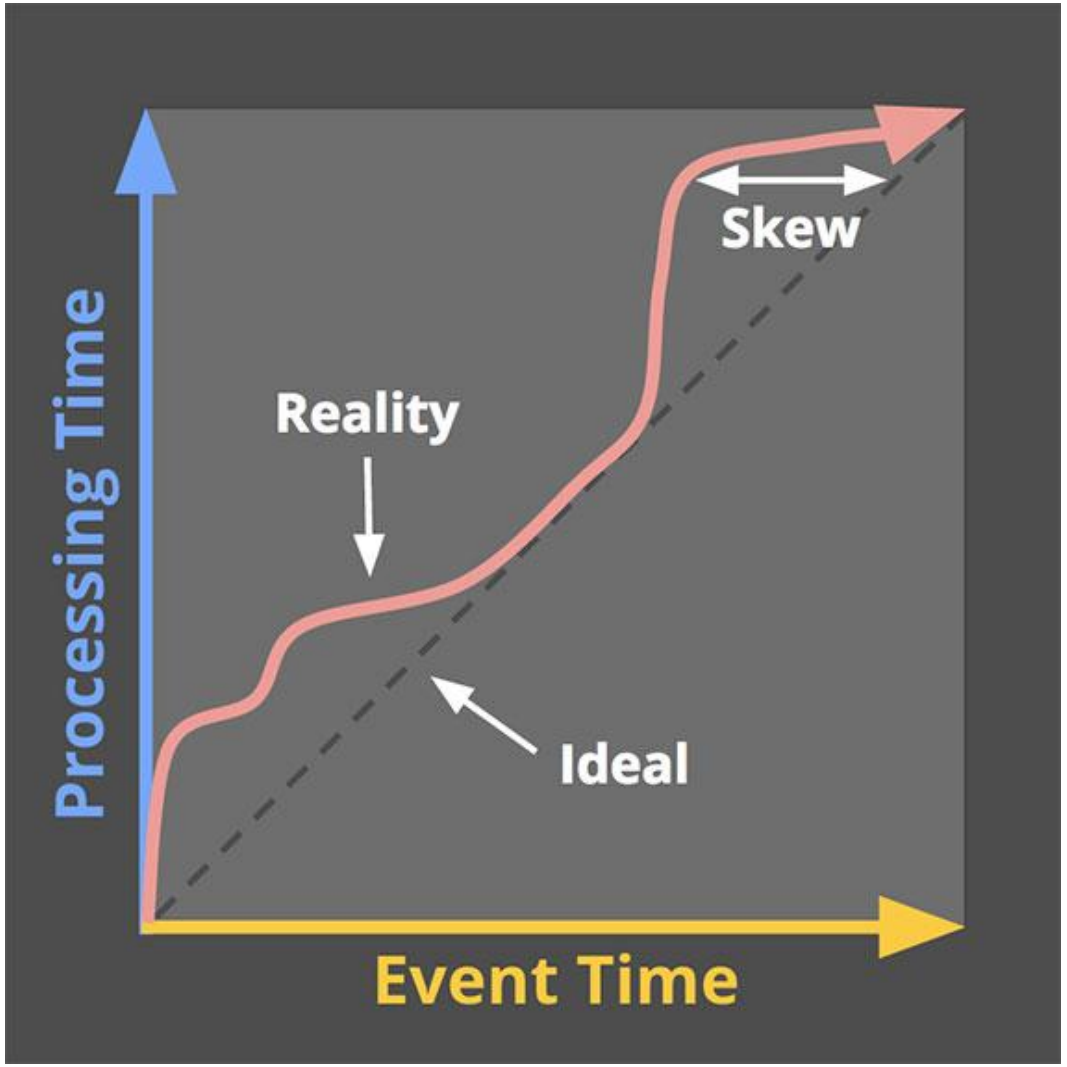
In a perfect world, event time processing would yield completely consistent and deterministic results, regardless of when events arrive, or their ordering. However, unless the events are known to arrive in-order (by timestamp), event time processing incurs some latency while waiting for out-of-order events. As it is only possible to wait for a finite period of time, this places a limit on how deterministic event time applications can be.

PROCESSING TIME

When a streaming program runs on processing time, all time-based operations (like time windows) will use the system clock of the machines that run the respective operator. An hourly processing time window will include all records that arrived at a specific operator between the times when the system clock indicated the full hour.

Processing time is the simplest notion of time and requires no coordination between streams and machines. It provides the best performance and the lowest latency.

TIME



TIMESTAMPS

In Apache Flink, timestamps are primarily used for defining and working with event time, which is the time when an event actually occurred, as opposed to the time when it is processed. Timestamps are used to assign events to windows, sort streams in event time order, and enable time-based operations in Flink SQL.

Flink allows processing data based on the actual event time, which is crucial for stream processing applications where events might be out-of-order or arrive with delays.

Flink needs to know the timestamp for each element in the stream. This is usually done by extracting the timestamp from a field in the event, or by using a timestamp assigner that defines how timestamps and watermarks are generated.

WATERMARK BASICS

Watermarks are a mechanism in Flink that help determine the progress in event time. They indicate that no more elements with a timestamp less than or equal to the watermark's timestamp will arrive.

In Apache Flink, watermarks are special events within a stream that indicate the progress of event time. They essentially tell the system that all events with timestamps less than or equal to the watermark's timestamp have (with high probability) arrived. This helps manage out-of-order events and allows Flink to process data efficiently even when some events are delayed.

WATERMARK BASICS

Watermarks are a mechanism in Flink that help determine the progress in event time. They indicate that no more elements with a timestamp less than or equal to the watermark's timestamp will arrive.

In Apache Flink, watermarks are special events within a stream that indicate the progress of event time. They essentially tell the system that all events with timestamps less than or equal to the watermark's timestamp have (with high probability) arrived. This helps manage out-of-order events and allows Flink to process data efficiently even when some events are delayed.

WATERMARKS IN ACTION

The mechanism in Flink to measure progress in event time is watermarks. Watermarks determine when to make progress during processing or wait for more records.

Certain SQL operations, like windows, interval joins, time-versioned joins, and MATCH_RECOGNIZE require watermarks. Without watermarks, they don't produce output.

A watermark means, "I have seen all records until this point in time". It's a long value that usually represents epoch milliseconds. The watermark of an operator is the minimum of received watermarks over all partitions of all inputs. It triggers the execution of time-based operations within this operator before sending the watermark downstream.

WATERMARK EMITTER

Watermarks can be emitted for every record, or they can be computed and emitted on a wall-clock interval. By default, Flink emits them every 200 ms.

The built-in function, CURRENT_WATERMARK, enables printing the current watermark for the executing operator.

WATERMARK EMITTER

Watermarks can be emitted for every record, or they can be computed and emitted on a wall-clock interval. By default, Flink emits them every 200 ms.

The built-in function, CURRENT_WATERMARK, enables printing the current watermark for the executing operator.

WATERMARK STRATEGIES

In order to work with event time, Flink needs to know the events timestamps, meaning each element in the stream needs to have its event timestamp assigned. This is usually done by accessing/extracting the timestamp from some field in the element by using a TimestampAssigner.

Timestamp assignment goes hand-in-hand with generating watermarks, which tell the system about progress in event time. You can configure this by specifying a WatermarkGenerator.

The Flink API expects a WatermarkStrategy that contains both a TimestampAssigner and WatermarkGenerator. A number of common strategies are available out of the box as static methods on WatermarkStrategy, but users can also build their own strategies when required.

WATERMARK STRATEGY OPTIONS

There are two places in Flink applications where a WatermarkStrategy can be used: 1) directly on sources and 2) after non-source operation.

The first option is preferable, because it allows sources to exploit knowledge about shards/partitions/splits in the watermarking logic. Sources can usually then track watermarks at a finer level and the overall watermark produced by a source will be more accurate. Specifying a WatermarkStrategy directly on the source usually means you have to use a source specific interface.

The second option (setting a WatermarkStrategy after arbitrary operations) should only be used if you cannot set a strategy directly on the source.

DEALING WITH LATE EVENTS

When working with event-time windowing, it can happen that elements arrive late, i.e. the watermark that Flink uses to keep track of the progress of event-time is already past the end timestamp of a window to which an element belongs.

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 operators.

ALLOWED LATENESS

Allowed lateness specifies by how much time elements can be late before they are dropped, and its default value is 0. 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. Depending on the trigger used, a late but not dropped element may cause the window to fire again. This is the case for the EventTimeTrigger.

In order to make this work, Flink keeps the state of windows until their allowed lateness expires. Once this happens, Flink removes the window and deletes its state.

By default, elements that arrive behind the watermark will be dropped.

SPECIFYING ALLOWED LATENESS

```
DataStream<T> input = ...;


input
    .keyBy(<key selector>)
    .window(<window assigner>)
    .allowedLateness(<time>)
    .<windowed transformation>(<window function>);
```

LATE DATA AND SIDE OUTPUTS

Using Flink's side output feature you can get a stream of the data that was discarded as late.

You first need to specify that you want to get late data using sideOutputLateData(OutputTag) on the windowed stream. Then, you can get the side-output stream on the result of the windowed operation.

LATE DATA AND SIDE OUTPUTS

```
final OutputTag<T> lateOutputTag = new OutputTag<T>("late-
data"){};

DataStream<T> input = ...;

SingleOutputStreamOperator<T> result = input

    .keyBy(<key selector>)

    .window(<window assigner>)

    .allowedLateness(<time>)

    .sideOutputLateData(lateOutputTag)

    .<windowed transformation>(<window function>);

DataStream<T> lateStream =
result.getSideOutput(lateOutputTag);
```

WINDOW OPERATIONS

Windows are at the heart of processing infinite streams. Windows split the stream into “buckets” of finite size, over which we can apply computations.

In a nutshell, window lifecycle is a window is created as soon as the first element that should belong to this window arrives, and the window is completely removed when the time (event or processing time) passes its end timestamp plus the user-specified allowed lateness. Flink guarantees removal only for time-based windows and not for other types, e.g. global windows). For example, with an event-time-based windowing strategy that creates non-overlapping (or tumbling) windows every 5 minutes and has an allowed lateness of 1 min, Flink will create a new window for the interval between 12:00 and 12:05 when the first element with a timestamp that falls into this interval arrives, and it will remove it when the watermark passes the 12:06 timestamp.

WINDOW LIFECYCLE MANAGEMENT

Each window will have a Trigger and a function (ProcessWindowFunction, ReduceFunction, or AggregateFunction) attached to it. The function will contain the computation to be applied to the contents of the window, while the Trigger specifies the conditions under which the window is considered ready for the function to be applied.

Apart from the above, you can specify an Evictor which will be able to remove elements from the window after the trigger fires and before and/or after the function is applied.

TRIGGERING POLICY

A triggering policy might be something like “when the number of elements in the window is more than 4”, or “when the watermark passes the end of the window”. A trigger can also decide to purge a window’s contents any time between its creation and removal. Purging in this case only refers to the elements in the window, and not the window metadata. This means that new data can still be added to that window.

KEYED VS NON-KEYED WINDOWS

The first thing to specify is whether your stream should be keyed or not. This has to be done before defining the window. Using the keyBy(...) will split your infinite stream into logical keyed streams. If keyBy(...) is not called, your stream is not keyed.

In the case of keyed streams, any attribute of your incoming events can be used as a key. Having a keyed stream will allow your windowed computation to be performed in parallel by multiple tasks, as each logical keyed stream can be processed independently from the rest. All elements referring to the same key will be sent to the same parallel task.

In case of non-keyed streams, your original stream will not be split into multiple logical streams and all the windowing logic will be performed by a single task, i.e. with parallelism of 1.

TYPES OF WINDOWS

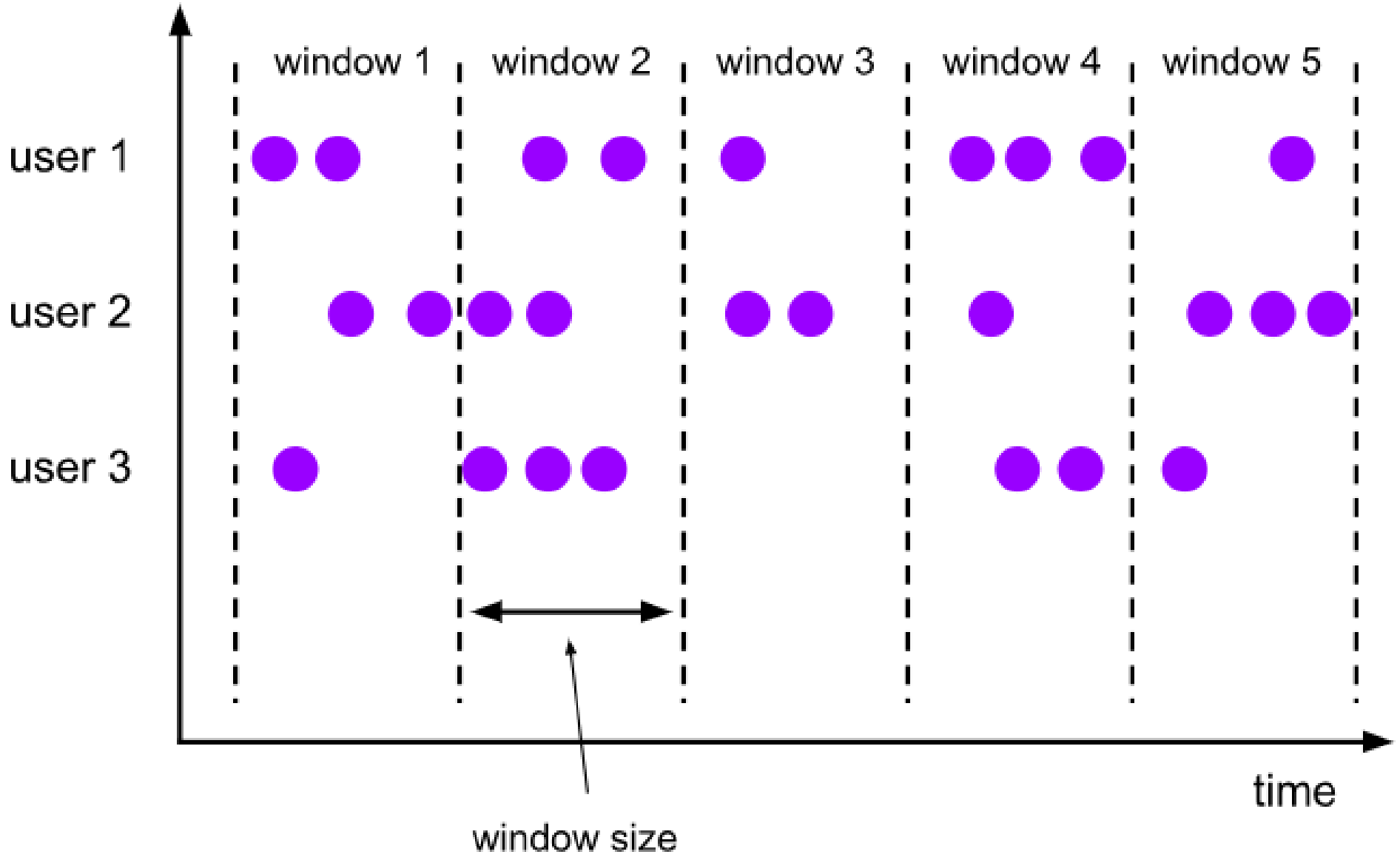
Tumbling - A tumbling windows assigner assigns each element to a window of a specified window size. Tumbling windows have a fixed size and do not overlap.

Sliding - The sliding windows assigner assigns elements to windows of fixed length. Similar to a tumbling windows assigner, the size of the windows is configured by the window size parameter, and windows can overlap. An additional window slide parameter controls how frequently a sliding window is started.

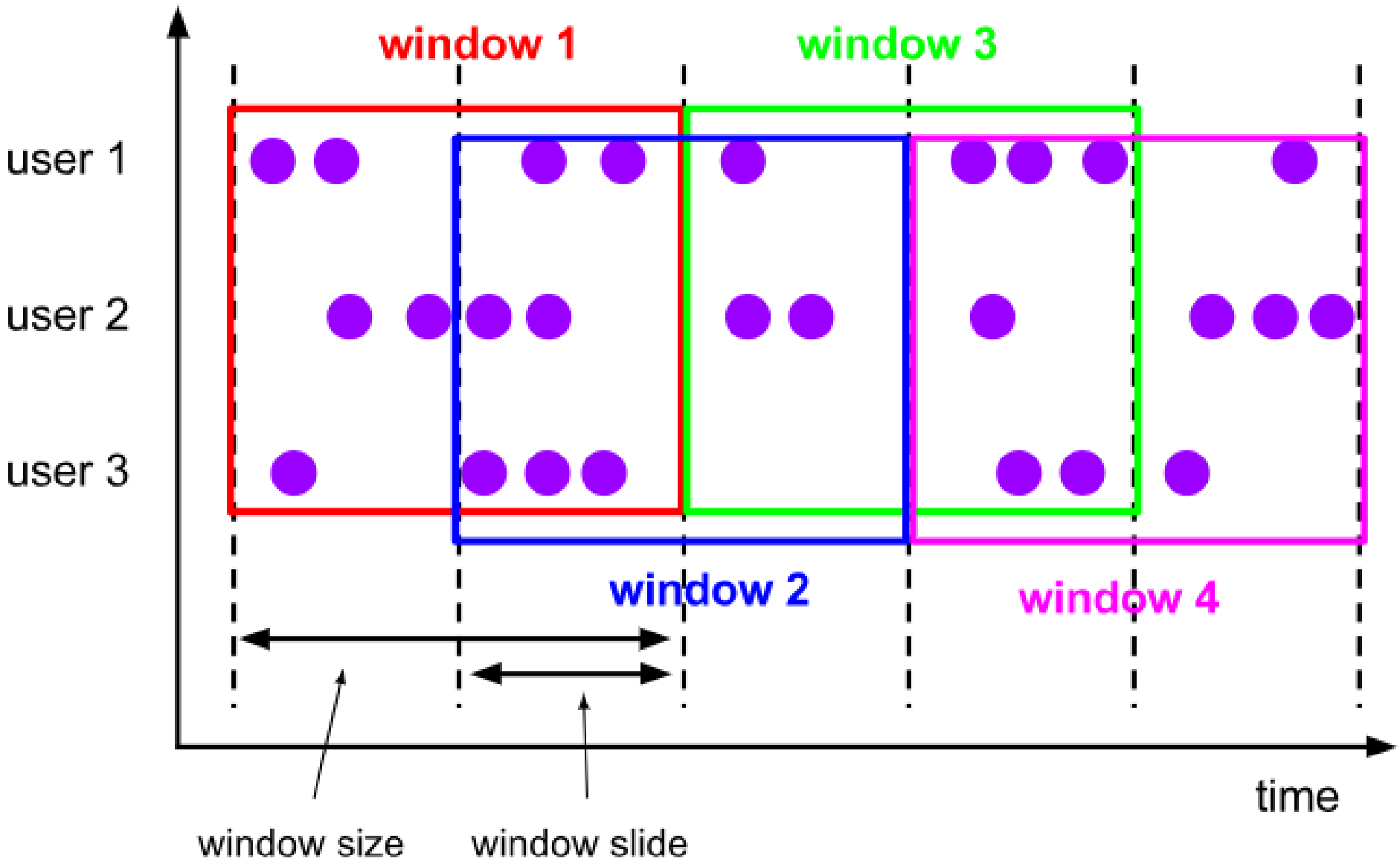
Session - The session windows assigner groups elements by sessions of activity. Session windows do not overlap and do not have a fixed start and end time, in contrast to tumbling windows and sliding windows.

Global - A global windows assigner assigns all elements with the same key to the same single global window. This windowing scheme requires a custom trigger. Global windows do not have a natural end to process the aggregated elements.

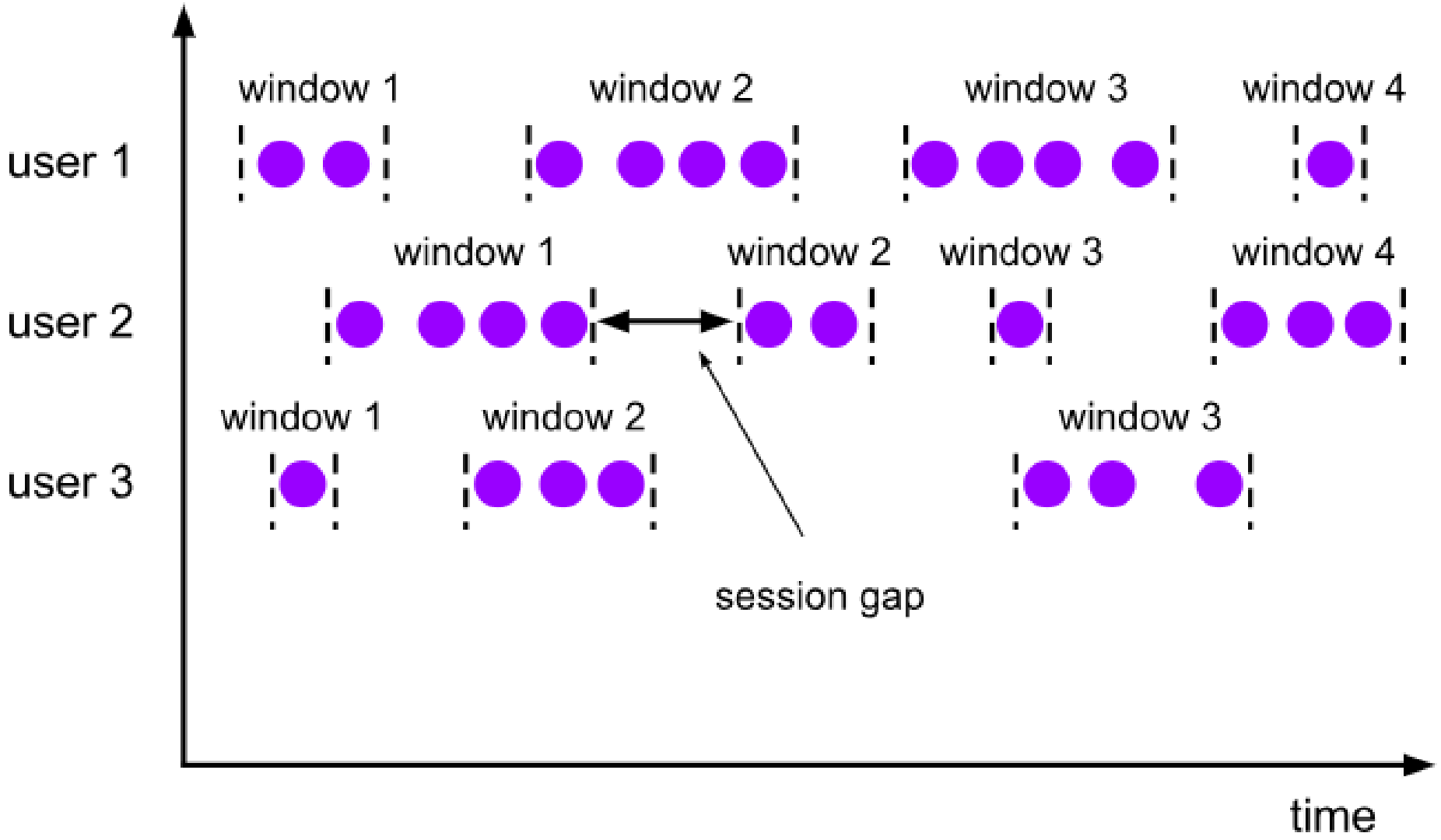
TUMBLING WINDOW



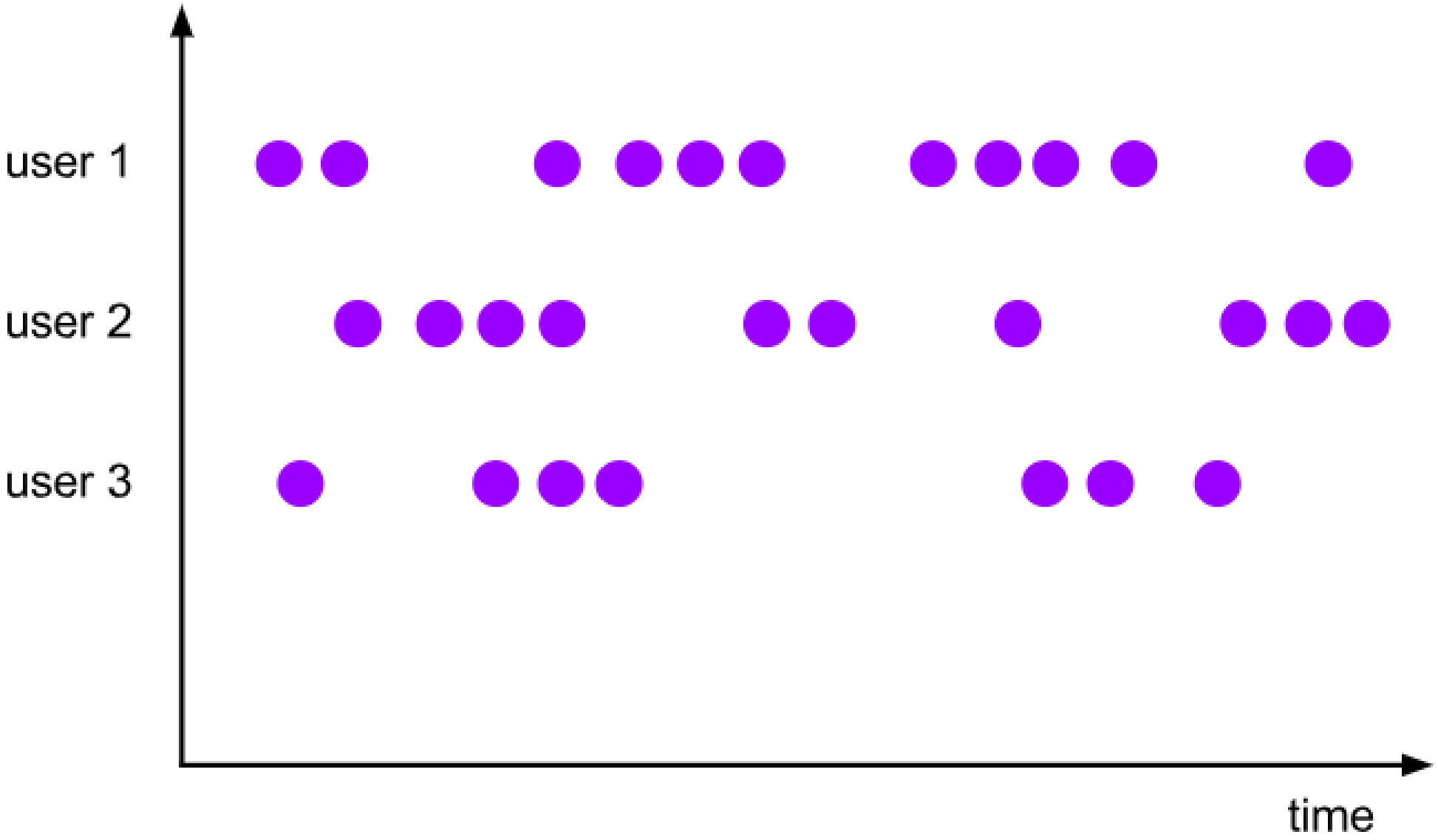
SLIDING WINDOW



SESSION WINDOW



GLOBAL WINDOW



Q&A AND OPEN DISCUSSION

Basic State Management

Basic State Management

- State Concepts
  - What is State?
  - When to Use State
  - Simple State Examples
  - State Backends
- Working with State
  - Keyed State
  - Value State
  - List State
  - Basic State Patterns

STATE MANAGEMENT

In Flink, state management refers to the process of storing and managing data that needs to be remembered across different events or time periods during a streaming application's execution.

This is crucial for operations like aggregations, windowing, and maintaining context for complex data pipelines.

Flink provides mechanisms like checkpoints and savepoints to ensure the accuracy and resilience of real-time data pipelines by capturing consistent snapshots of the application's state and operator state.

STATE TYPES

Keyed State: associated with specific keys in a stream and is designed for partitioned and efficient state management.

Operator State (Non-Keyed State): associated with individual operator instances and can be used in scenarios where state isn't partitioned by keys, such as in source/sink implementations.

Queryable State: allows external applications to access and query Flink's state in real-time, enabling dynamic insights and actions.

WHAT IS STATE

State as a First-Class Citizen: Flink considers application state as a fundamental part of the data processing pipeline, managing its persistence and operations transparently.

In stream processing, “state” refers to any data that needs to be remembered across events to perform operations correctly.

Flink provides a robust system for managing this state, ensuring that it remains consistent and available, even in the face of failures.

STATE EXAMPLE

Keyed State for event counting, if you want to count the number of occurrences of a particular event, you'll need to maintain a count that gets updated as new events arrive. This count is the "state" of the computation.

Operator state can be used to share information across all parallel instances of an operator, making it a powerful tool for scenarios requiring global state access.

STATE BACKENDS

HeapStateBackend: This backend stores state on the Java heap. It's fast for operations, but the size of the state is limited by the available heap memory.

RocksDBStateBackend: RocksDB is a key-value store that persists data on disk, allowing for large state sizes. This backend is well-suited for applications that require a large, persistent state.

JobManager: A default state backend that stores state in memory.

External Storage (S3, HDFS): Allows storing state in external storage for large datasets.

TYPES OF STATE

State as a First-Class Citizen: Flink considers application state as a fundamental part of the data processing pipeline, managing its persistence and operations transparently.

Keyed State: associated with specific keys in a stream and is designed for partitioned and efficient state management.

Operator State (Non-Keyed State): associated with individual operator instances and can be used in scenarios where state isn't partitioned by keys, such as in source/sink implementations.

Queryable State: allows external applications to access and query Flink's state in real-time, enabling dynamic insights and actions.

KEYED STATE: VALUE AND LIST

ValueState<T>: This keeps a value that can be updated and retrieved (scoped to key of the input element as mentioned above, so there will possibly be one value for each key that the operation sees).

ListState<T>: This keeps a list of elements. You can append elements and retrieve an Iterable over all currently stored elements. Elements are added using add(T) or addAll(List<T>), the Iterable can be retrieved using Iterable<T> get().

KEYED STATE: OTHER STATE PRIMITIVES

ReducingState<T>: This keeps a single value that represents the aggregation of all values added to the state. The interface is similar to ListState but elements added using add(T) are reduced to an aggregate using a specified ReduceFunction.

AggregatingState<IN, OUT>: This keeps a single value that represents the aggregation of all values added to the state. Contrary to ReducingState, the aggregate type may be different from the type of elements that are added to the state. The interface is the same as for ListState but elements added using add(IN) are aggregated using a specified AggregateFunction.

MapState<UK, UV>: This keeps a list of mappings. You can put key-value pairs into the state and retrieve an Iterable over all currently stored mappings. Mappings are added using put(UK, UV) or putAll(Map<UK, UV>). The value associated with a user key can be retrieved using get(UK). The iterable views for mappings, keys and values can be retrieved using entries(), keys() and values() respectively.

STATE PATTERNS: FRAUD DETECTION

State management is essential in building a fraud detection system for an e-commerce platform. By maintaining state across events, Flink can track user behavior patterns and detect anomalies that may indicate fraudulent activity.

Use Case: Track user session data, such as login times, page visits, and transaction amounts, to identify unusual patterns.

STATE PATTERNS: ADVANCED ANALYTICS

Flink’s state management allows social media platforms to perform real-time analytics, such as trending topic detection, bullying, and user sentiment analysis.

Use Case: Maintain state across multiple event streams (e.g., posts, comments, likes, shares) to aggregate detailed metrics and detect trends in real-time. Identify abuse by monitoring stream responses.

STATE PATTERNS: PORTFOLIO MANAGEMENT

In financial services, managing user portfolios requires tracking a variety of metrics, such as stock prices, asset allocation, and risk exposure. Flink's state management capabilities enable the efficient computation of these metrics in real time.

Use Case: Use keyed state to maintain portfolio data for each user, enabling real-time updates, risk tolerance, portfolio recommendations, and risk assessment.

CHECKPOINTS

The primary purpose of checkpoints is to provide a recovery mechanism in case of unexpected job failures. A checkpoint's lifecycle is managed by Flink, i.e. a checkpoint is created, owned, and released by Flink - without user interaction. Because checkpoints are being triggered often, and are relied upon for failure recovery, the two main design goals for the checkpoint implementation are i) being as lightweight to create and ii) being as fast to restore from as possible. Optimizations towards those goals can exploit certain properties, e.g., that the job code doesn't change between the execution attempts.

Checkpoints are automatically deleted if the application is terminated by the user (except if checkpoints are explicitly configured to be retained).

Checkpoints are stored in state backend-specific (native) data format (may be incremental depending on the specific backend).

SAVEPOINTS

Savepoints are created internally with the same mechanisms as checkpoints, but are conceptually different and can be a bit more expensive to produce and restore from. Their design focuses more on portability and operational flexibility, especially with respect to changes to the job. The use case for savepoints is for planned, manual operations. For example, this could be an update of your Flink version, changing your job graph, and so on.

Savepoints are created, owned and deleted solely by the user. That means, Flink does not delete savepoints neither after job termination nor after restore.

Savepoints are stored in a state backend independent (canonical) format.

Q&A AND OPEN DISCUSSION

Deployment and Next Steps

Deployment and Next Steps

- Basic Deployment
  - Deployment Options
  - Local Cluster
  - Configuration Basics
  - Resource Planning
- Moving Forward
  - Best Practices Review
  - Advanced Topics Preview
  - Learning Resources
  - Common Use Cases

DEPLOYMENT OPTIONS

Flink can be deployed in several ways, including

- Application Mode
- Per-Job Mode
- Session Mode

These modes offer different approaches to resource management and application lifecycle. Additionally, Flink can be deployed on various platforms like Kubernetes, YARN, and in standalone configurations, each with its own advantages.

DEPLOYMENT MODES

Application Mode: A dedicated cluster is created for each Flink application, with the application's main method executed on the JobManager. This mode provides resource isolation and scalability for individual applications.

Per-Job Mode: Similar to Application Mode, but offers a more streamlined application submission process.

Session Mode: A shared cluster is created, allowing multiple Flink applications to share resources and submit jobs to it. This mode is useful for development and testing environments.

DEPLOYMENT PLATFORMS

Kubernetes: Flink can be deployed natively on Kubernetes, leveraging its resource management and orchestration capabilities. This allows for dynamic allocation of TaskManagers based on resource needs.

YARN: Flink can also be deployed on Hadoop YARN, another popular resource management system.

Standalone: Flink can be deployed in a standalone configuration, which is a simple way to run Flink without relying on external resource managers.

LOCAL CLUSTER

A Flink local cluster is a simplified Flink environment that runs entirely within a single JVM on your local machine, providing a way to develop, test, and experiment with Flink applications without the need for a distributed cluster. It effectively emulates a Flink cluster on your machine, allowing you to run Flink jobs and observe their execution without the overhead of a distributed setup.

LOCAL CLUSTER ANATOMY

Local Execution Environment: Flink provides a LocalExecutionEnvironment that simulates a full Flink cluster, including a JobManager and TaskManager, within a single JVM.

Mini Cluster: This environment essentially creates a mini-cluster locally, where you can run your Flink applications.

Simplified Development: It's ideal for developers who want to quickly prototype and test their Flink applications without setting up a complex, distributed environment.

No Distributed Infrastructure Required: Unlike deploying Flink on a distributed cluster (e.g., using YARN or Kubernetes), a local cluster runs entirely on your local machine, making it easy to get started and experiment.

Development and Testing: Local clusters are useful for development, testing, and debugging your Flink applications. You can use it to verify that your Flink job logic works correctly before deploying it to a production cluster.

Standalone JVM: The LocalExecutionEnvironment runs all the Flink components within the same JVM, which simplifies the setup and allows you to execute Flink jobs directly from your code.

Simulating Distributed Behavior: While it runs on a single machine, the LocalExecutionEnvironment simulates the behavior of a distributed cluster, allowing you to test your Flink applications as if they were running on a distributed environment

CONFIGURATION BASICS

The core of Flink configuration lies in the **config.yaml** file within the **conf** directory of your Flink installation. This file allows you to customize various aspects of the Flink cluster, including resource management, job execution, and more. Understanding the different sections and options within this file is crucial for optimizing Flink's performance and behavior for your specific needs.

CONFIGURATION AREAS

Job Manager Configuration: Controls the memory allocated to the JobManager, the core component responsible for managing Flink jobs.

TaskManager Configuration: Specifies the memory and other resources allocated to TaskManagers, which execute Flink tasks.

Parallelism: Defines the number of parallel instances for different tasks within your Flink application, impacting resource utilization and job execution speed.

State Backend Configuration: Determines how Flink applications store their state, influencing fault tolerance and performance. Checkpointing: Enables fault tolerance by periodically saving the state of the Flink application, allowing it to recover from failures.

High Availability: Configures Flink to be highly available, ensuring that your Flink cluster can continue running even if individual nodes fail.

Metrics: Sets up Flink's monitoring and logging capabilities, allowing you to track the health and performance of your Flink jobs and cluster..

RESOURCE PLANNING

In Apache Flink, resource planning focuses on effectively allocating and managing compute resources for running streaming applications. This involves determining the number of task slots, the optimal memory allocation for each Task Manager, and setting the maximum parallelism for jobs, as well as considering the number of Task Managers needed. Flink's resource scheduling framework, with its two-layer model, ensures efficient resource allocation and utilization.

1. Two-Layer Resource Scheduling Model:

Slot Manager: Responsible for allocating resources (task slots) from the cluster to individual jobs.

Scheduler: Allocates resources (task slots) to tasks within a specific job.

2. Resource Management:

Task Slots: The fundamental unit of resource allocation in Flink, representing a unit of computation in a Task Manager.

Task Manager: Each Flink cluster has one or more Task Managers, each of which manages a fixed number of task slots.

Job Manager: Coordinates the distributed execution of a Flink application, managing the job graph and deploying tasks to Task Managers.

RESOURCE PLANNING

In Apache Flink, resource planning focuses on effectively allocating and managing compute resources for running streaming applications. This involves determining the number of task slots, the optimal memory allocation for each Task Manager, and setting the maximum parallelism for jobs, as well as considering the number of Task Managers needed. Flink's resource scheduling framework, with its two-layer model, ensures efficient resource allocation and utilization.

Two-Layer Resource Scheduling Model:

- Slot Manager: Responsible for allocating resources (task slots) from the cluster to individual jobs.
- Scheduler: Allocates resources (task slots) to tasks within a specific job.

Resource Management:

- Task Slots: The fundamental unit of resource allocation in Flink, representing a unit of computation in a Task Manager.
- Task Manager: Each Flink cluster has one or more Task Managers, each of which manages a fixed number of task slots.
- Job Manager: Coordinates the distributed execution of a Flink application, managing the job graph and deploying tasks to Task Managers.

RESOURCE PLANNING

Resource Allocation Strategies:

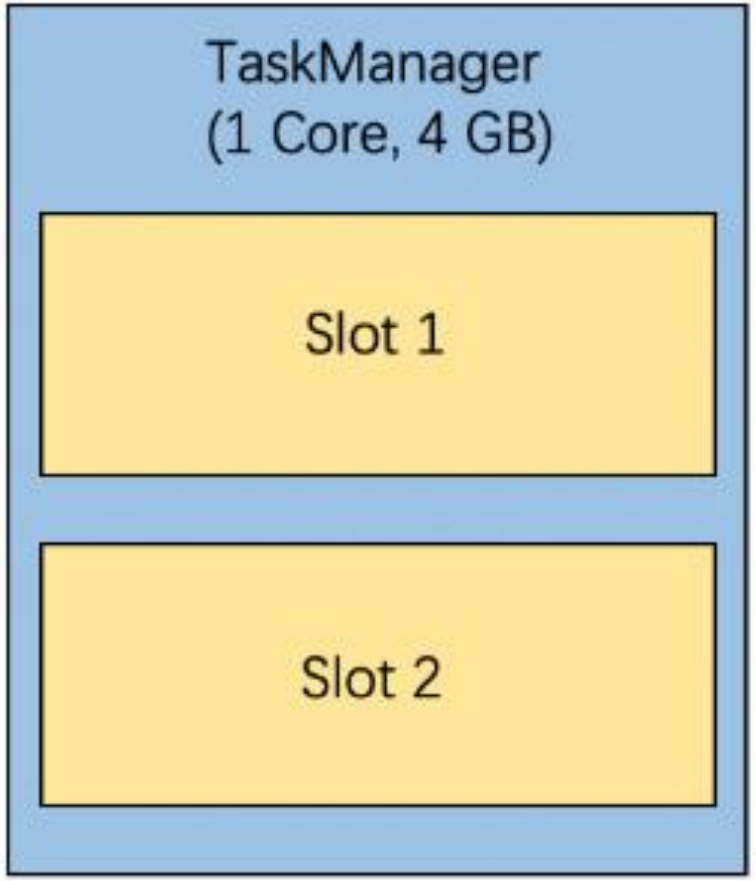
- Slot Caching: Batch jobs and streaming job failover can reuse slots.
- Slot Sharing: Multiple tasks can share the same slot under certain conditions.

Fine-Grained Resource Management: Allows users to specify resource profiles for slots, ensuring that tasks are allocated to slots with matching resources.

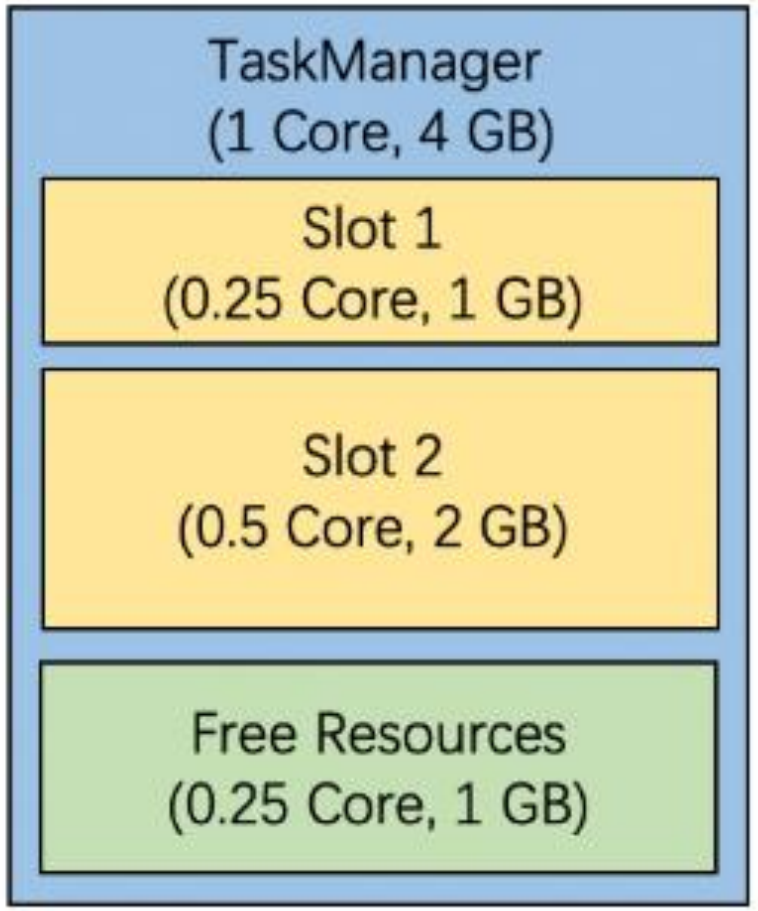
Best Practices for Resource Planning:

- Start Small: Begin with a proof of concept and low data volume.
- Keep Task Managers Under 100: A common recommendation is to keep the number of Task Managers under 100.
- Ensure Sufficient Slots: Make sure the number of task slots is sufficient for the maximum parallelism required by your jobs.

RESOURCE MANAGEMENT



Coarse-Grained
Resource management



Fine-Grained
Resource management

FLINK BEST PRACTICES

Flink best practices focus on optimizing application design, performance, fault tolerance, and resource management. This includes using appropriate APIs, managing state effectively, leveraging event time processing, tuning parallelism, and employing checkpointing for fault tolerance. Additionally, minimizing the size of the uber JAR, choosing the right state backend, and monitoring external dependencies are crucial.

FLINK BEST PRACTICES - APPLICATION DESIGN

Design applications with streaming in mind, considering data as unbounded streams.

Utilize event time semantics to handle out-of-order events and late data.

Choose the right state backend (memory-based or disk-based) based on application needs.

Manage state properly, as stateful processing is a key feature for complex computations and fault tolerance.

FLINK BEST PRACTICES - PERFORMANCE

Minimize the size of the uber JAR to reduce deployment time and memory consumption.

Tune parallelism for operators based on application bottlenecks and resource constraints.

Monitor external dependency resource usage to avoid backpressure and performance bottlenecks.

Consider using custom partitioners for more control over data distribution.

FLINK BEST PRACTICES – FAULT TOLERANCE

Use checkpoints to create snapshots of the application state for fault tolerance.

Choose between aligned and non-aligned checkpoints based on consistency and latency requirements.

Ensure external dependencies are properly provisioned for application throughput.

FLINK BEST PRACTICES – STATE OPTIMIZATION

Use keyed states efficiently, especially with Flink's managed keyed states.

Optimize state for memory consumption and processing time.

Consider using watermarks for event time processing.

FLINK BEST PRACTICES – OBSERVABILITY

Monitor application health, including job state, restarts, and checkpoints.

Use appropriate logging strategies, such as Logback, when running Flink outside of an IDE.

Monitor external dependency resource usage to identify potential bottlenecks.

FLINK BEST PRACTICES – ODDS AND ENDS

Use appropriate APIs and libraries provided by Flink.

Test applications thoroughly using unit tests.

Implement least privilege access for security.

Understand the stream-table duality in Flink SQL.

Be mindful of serialization costs and choose efficient serialization methods.

ADVANCED TOPICS

Advanced topics in Apache Flink include dynamic application logic updates, dynamic data partitioning, custom windowing, advanced SQL operations, and fault tolerance mechanisms. These allow for building sophisticated, event-driven applications that go beyond basic streaming ETL, especially for complex scenarios like fraud detection, real-time alerting, and IoT data processing.

ADVANCED TOPICS – DYNAMIC APPLICATIONS

Flink allows for dynamic updates of application logic without stopping and restarting jobs, enabling adjustments to algorithms and configurations at runtime.

This is achieved through features like

Flink's dynamic class loading https://flink.apache.org/2020/01/15/advanced-flink-application-patterns-vol.1-case-study-of-a-fraud-detection-system/

Stream-table duality https://medium.com/@hivemind_tech/understanding-apache-flink-a-journey-from-core-concepts-to-materialised-views-b8129070acf4, enabling flexible adaptation to changing business requirements.

ADVANCED TOPICS – DYNAMIC DATA

Flink provides the ability to modify how events are distributed and grouped at runtime, allowing for dynamic adjustments based on changing data patterns or performance needs.

This is especially useful when the application logic is dynamically reconfigurable and requires flexible data handling.

ADVANCED TOPICS – CUSTOM WINDOWS

Flink's low-level process function API https://flink.apache.org/2020/07/30/advanced-flink-application-patterns-vol.3-custom-window-processing/ allows for highly customizable windowing mechanisms, beyond the standard window API.

This enables implementing low-latency alerting, controlling state growth, and handling complex windowing scenarios.

ADVANCED TOPICS – FAULT TOLERANCE

Flink's architecture ensures fault tolerance through mechanisms like checkpointing and failover https://nightlies.apache.org/flink/flink-docs-master/docs/dev/configuration/advanced/

This guarantees data consistency even in the face of failures, crucial for applications that cannot tolerate data loss or downtime.

LEARNING RESOURCES

The main location for this is the Apache Flink website, but keep in mind which version of Flink you're targeting. https://flink.apache.org/

Flink GitHub repo - https://github.com/apache/flink/

Medium - https://medium.com/search?q=flink

Stack Overflow - https://stackoverflow.com/search?q=flink

COMMON USE CASES REVISITED

Event-driven applications: Flink excels at processing real-time data streams, making it suitable for applications like fraud detection (analyzing transactions in real-time), rule-based alerting (triggering alerts based on specific events), and real-time UX personalization.

Data analytics: Flink provides tools for analyzing streaming data and extracting insights in real-time, which can be used for real-time dashboards, business process monitoring, and customer behavior analysis.

Streaming data pipelines: Flink can be used to build ETL (Extract, Transform, Load) processes for streaming data, moving data from various sources, transforming it, and loading it into data warehouses or other systems.

Data quality monitoring: Flink can be used to monitor the quality and consistency of data sources by applying validation rules and metrics on streaming data, and reporting or fixing any issues.

IoT data processing: Flink can process data from IoT devices in real-time, enabling applications like anomaly detection, predictive maintenance, and optimization.

Batch processing: While known for stream processing, Flink can also handle batch processing tasks where data is processed in chunks.

Q&A AND OPEN DISCUSSION