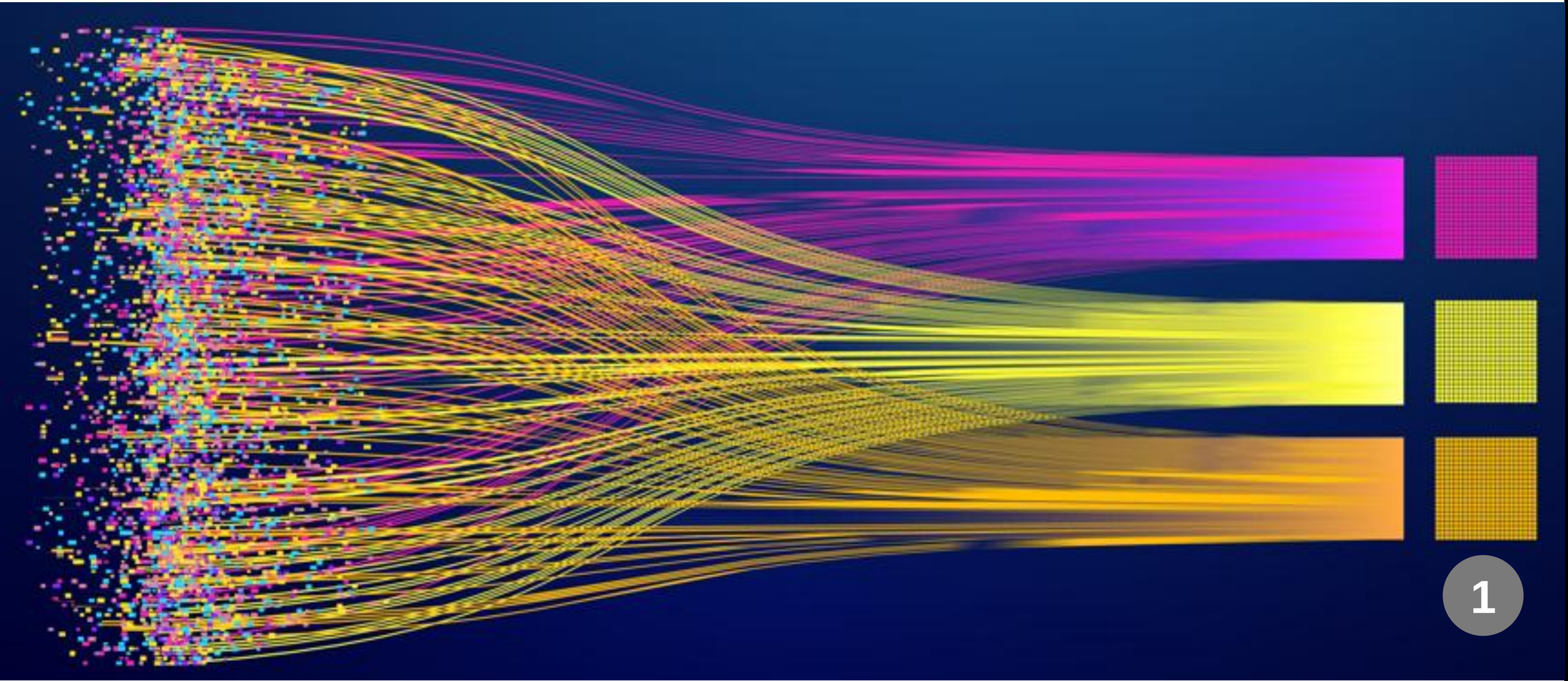
“Develop a passion for learning.”

Real-Time Data Processing using Apache Flink



logistics

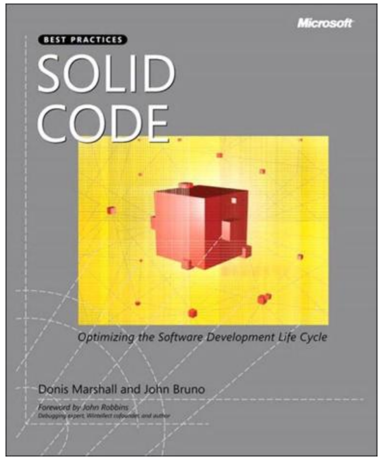
- Class Hours:
- Instructor will set class start and end times.
- There will be regular breaks in class.
- Planned lunch breaks for 1:15

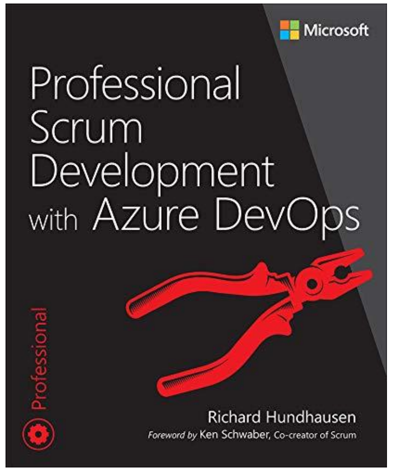
- Telecommunication:
- Turn off or set electronic devices to silent (not vibrate)
- Reading or attending to devices can be distracting to other students
- Try to delay until breaks or after class

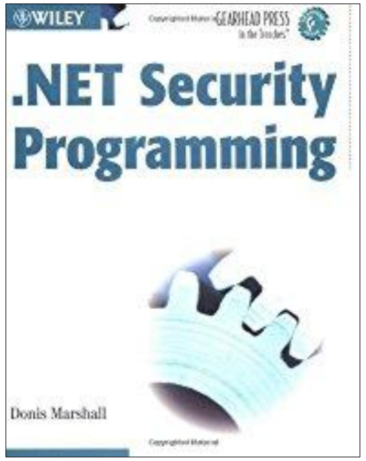
- Miscellaneous:
- Courseware
- Bathroom
- Fire drills

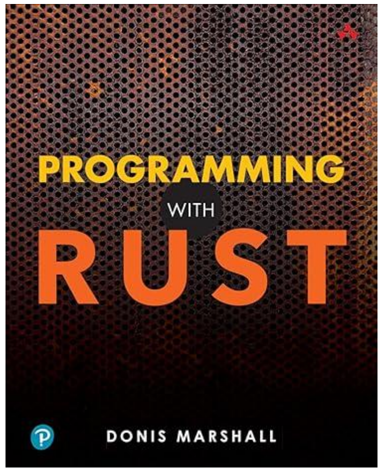
George Niece

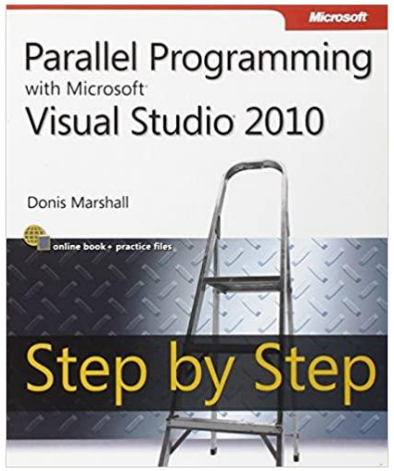
- Data Engineering
- Security
- Multicloud
- Resilience
- Certified in many technologies

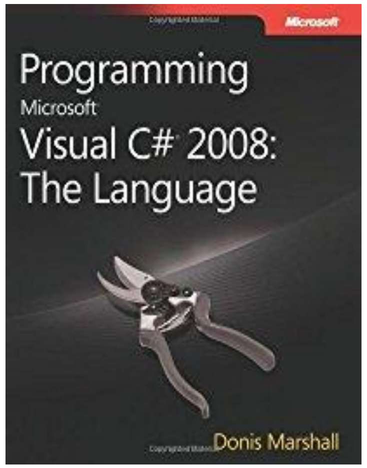

Microsoft
BEST PRACTICES
SOLID CODE
Optimizing the Software Development Life Cycle
Donis Marshall and John Bruno


Microsoft
Professional Scrum Development with Azure DevOps
Professional
Richard Hundhausen
Foreword by Ken Schwaber, Co-creator of Scrum






Microsoft
Parallel Programming with Microsoft Visual Studio 2010
Donis Marshall
online book + practice files
Step by Step



Introduce yourself

- Time to introduce yourself:
- Name
- What is your role in the organization
- Indicate OWASP experience

labs

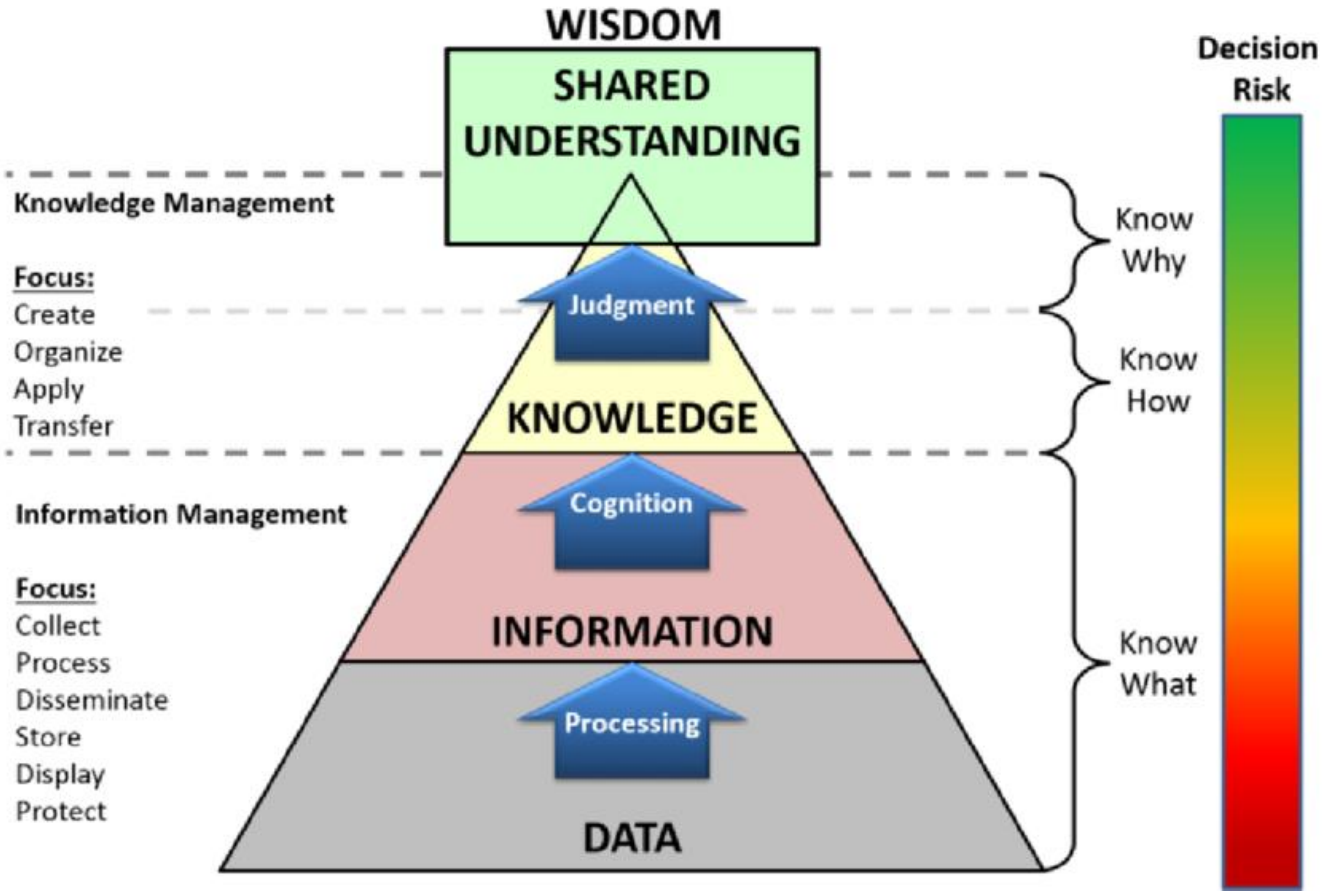
- In this class, some labs are completed as teams.
- The labs are intended as a collaborative exercise and team members work together in breakout rooms.
- Teams are expected to present their lab solutions to the class.

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

SINGLE LARGE IMAGE



BULLETED LIST

- Data that has been processed, structured, or organized to provide context or meaning.
- Characteristics:
  - Answers questions like who, what, when, and where.
  - Give insights into the behaviors and characteristics of entities in the real world
  - Provides insights but lacks a deeper level of understanding.
- 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

IMAGE RIGHT

Chen’s four levels correspond to the three level architecture proposed for data base development

- Has a wide range of applications, as we shall see

The correspondences are:

- **External views:** a specific group of users’ view of the data. Data at this level is *discovered* through investigation
- **Conceptual level:** a common, domain specific *defined* view of the data that is rigorous and complete
- **Internal level:** How the data is *structured*: relational model or dimensional model or network model for example
- **Physical level:** The specific *implementation* of the structure defined at the internal level

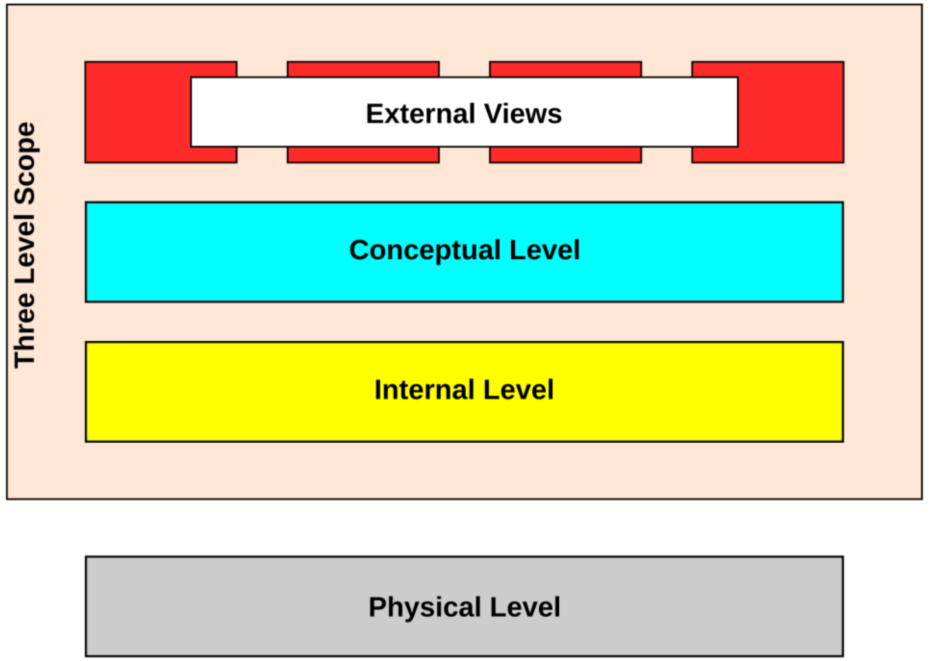
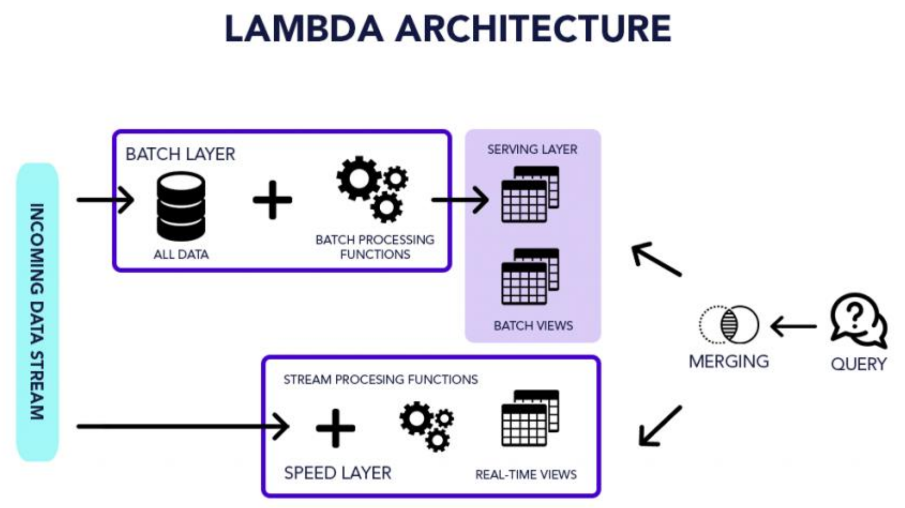


IMAGE BOTTOM

- Used when we have streaming data and historical data
  - We set up a two part architecture
  - A real time speed layer, that could be part of a OLTP transactional system
  - The real time layer keeps track of data being used in transactions
  - The batch storage layer keeps historical data for training and BI



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?

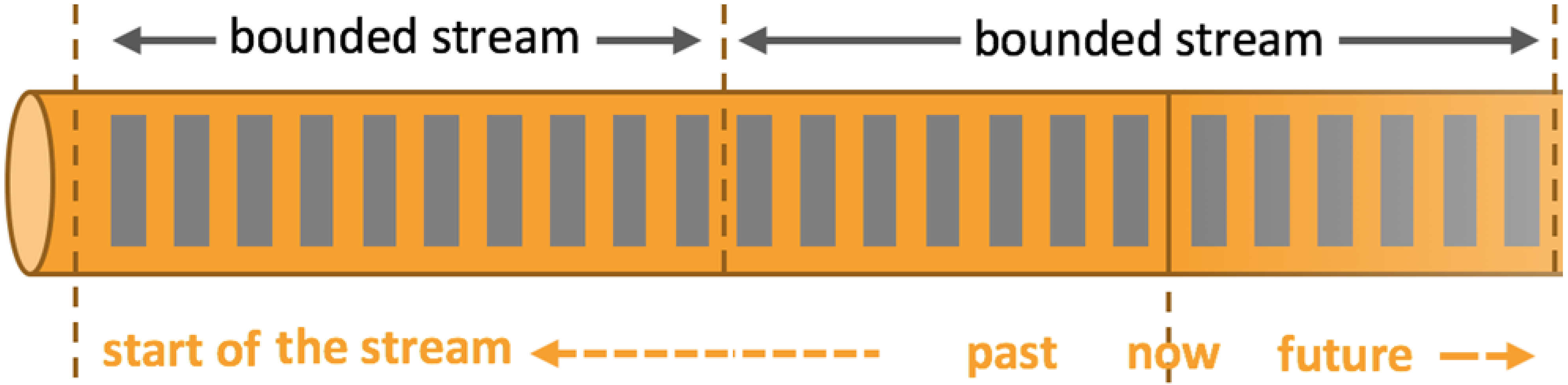
Have we implemented encryption, access control, cyber recovery, and regular audits

10. Performance:

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

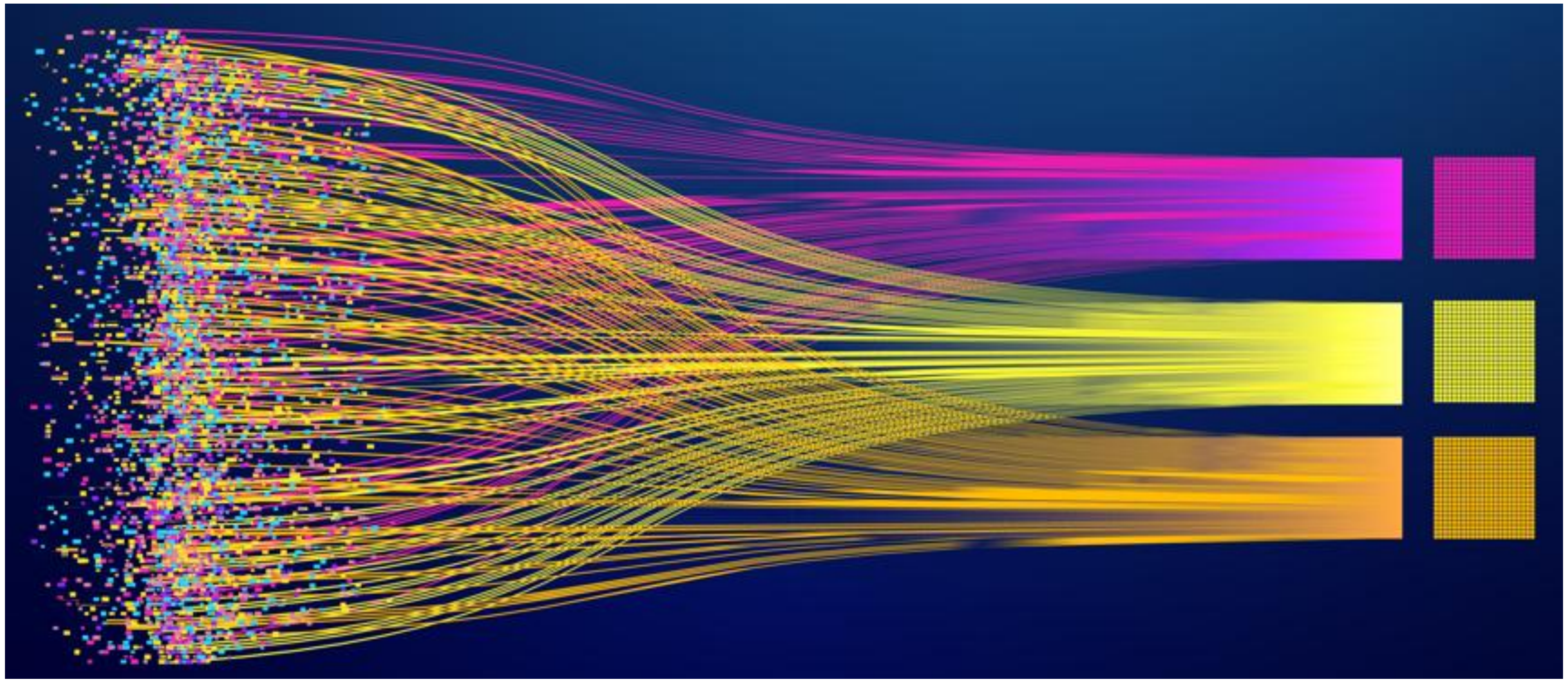
BATCH PROCESSING OVERVIEW

- 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.



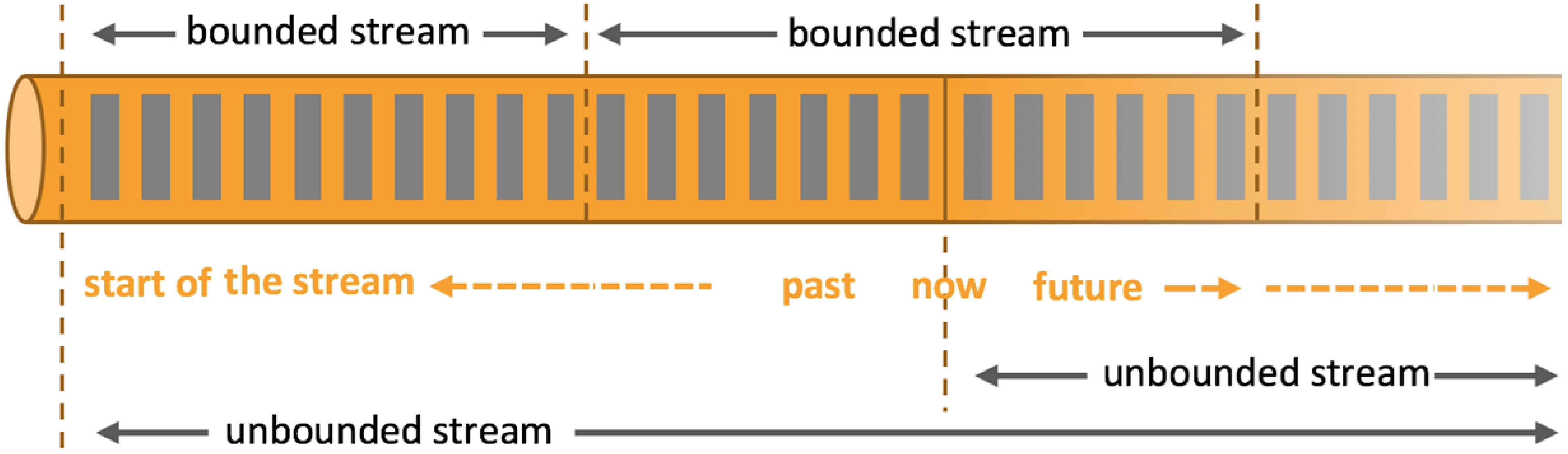
BATCH PROCESSING

- **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.

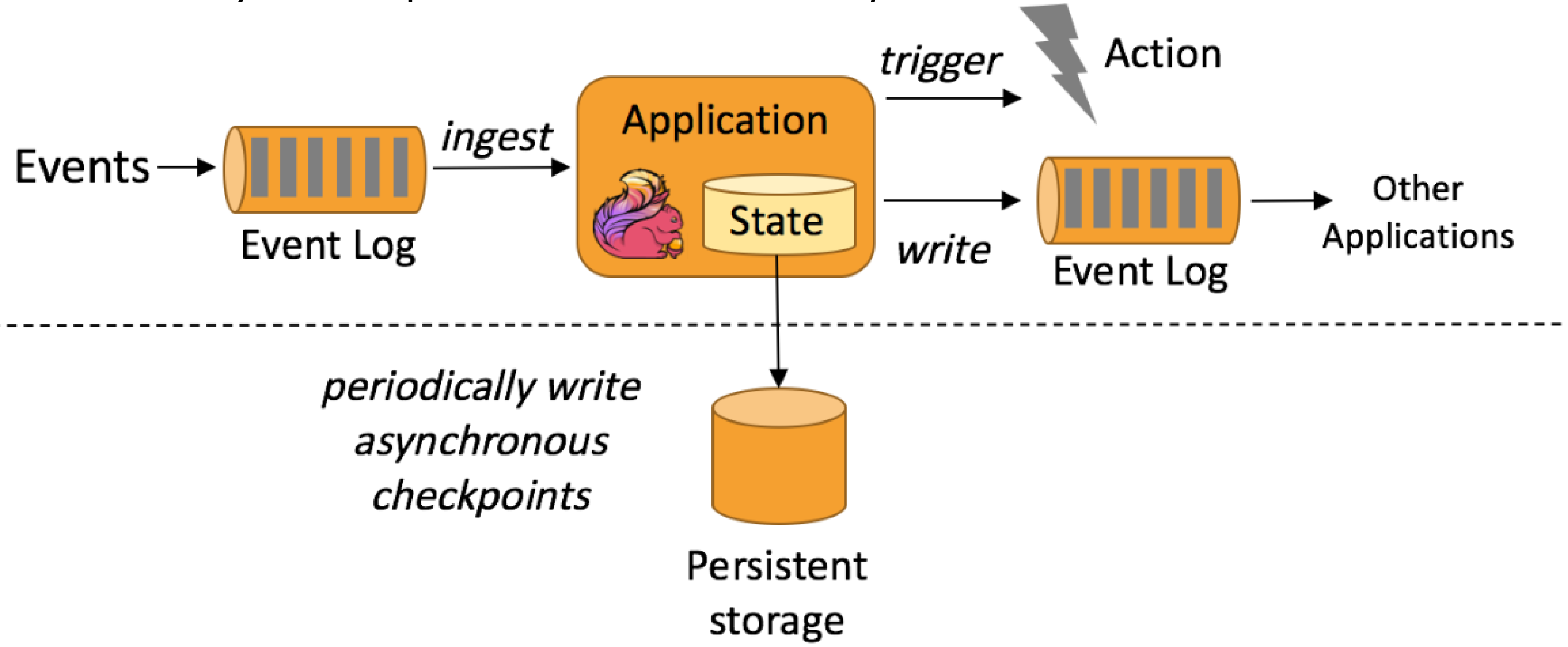


WHEN TO USE STREAM PROCESSING?

Always

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

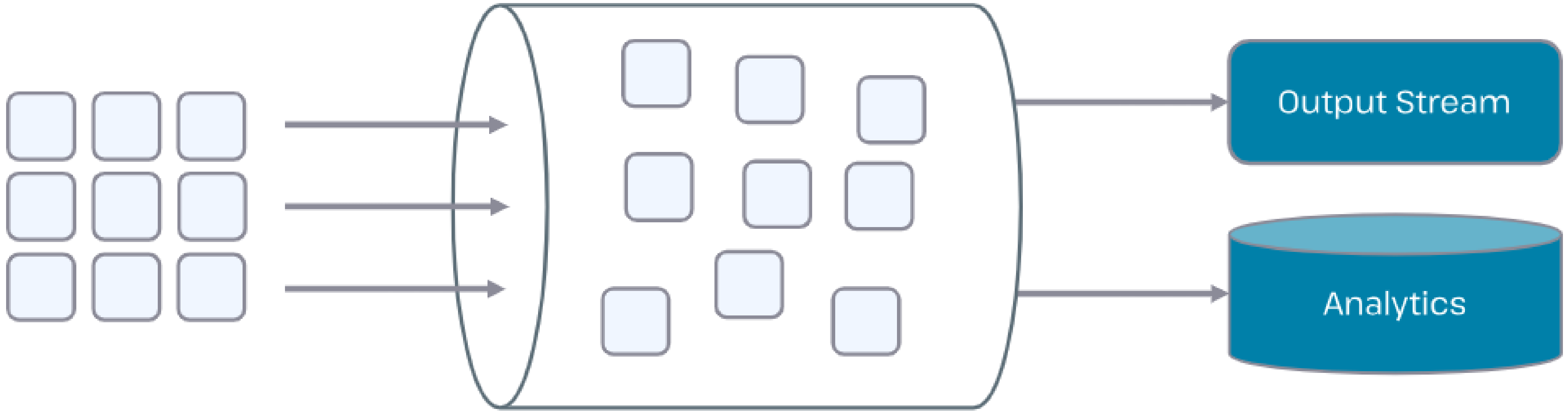
The distinction isn’t in the actual timing of the stream, but rather the decision, deadline or response that the event or signal generates. The response should occur similarly under any system load, rather than becoming bi-modal.

PROCESSING GUARANTEES

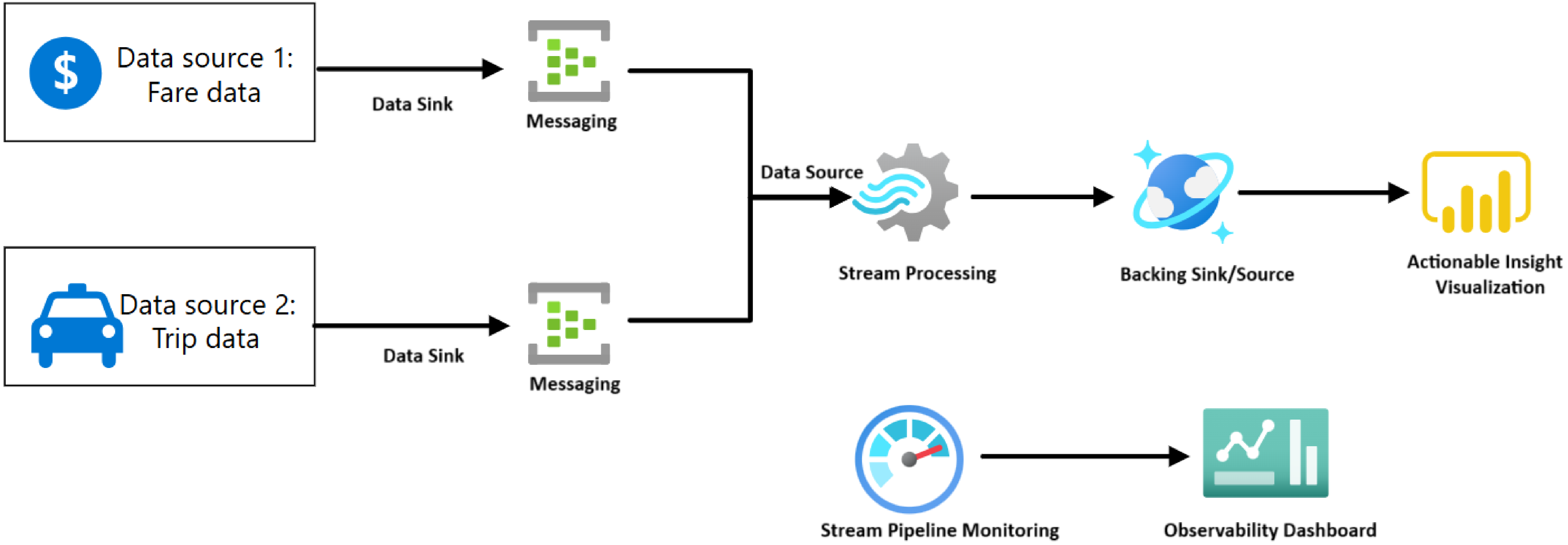
Fault tolerance mechanisms recover processes in the presence of failures and continues to execute them. Such failures include hardware failures, network failures, transient program failures, etc.

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.

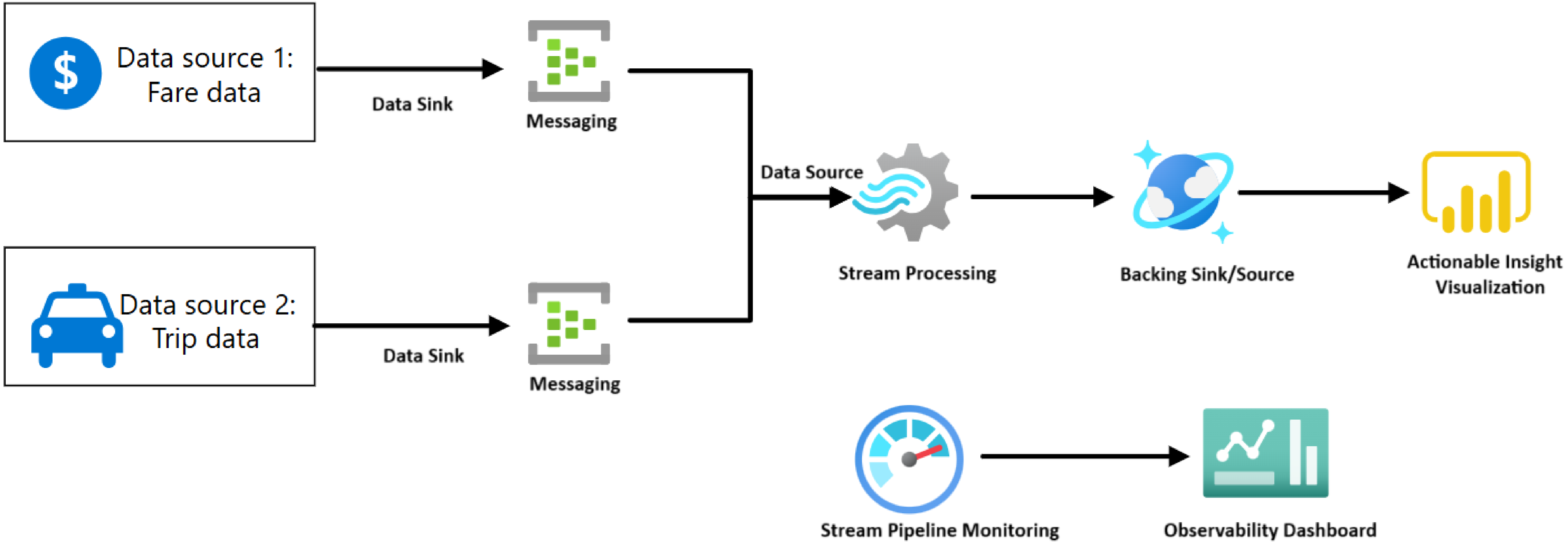
BASIC ARCHITECTURE PATTERNS



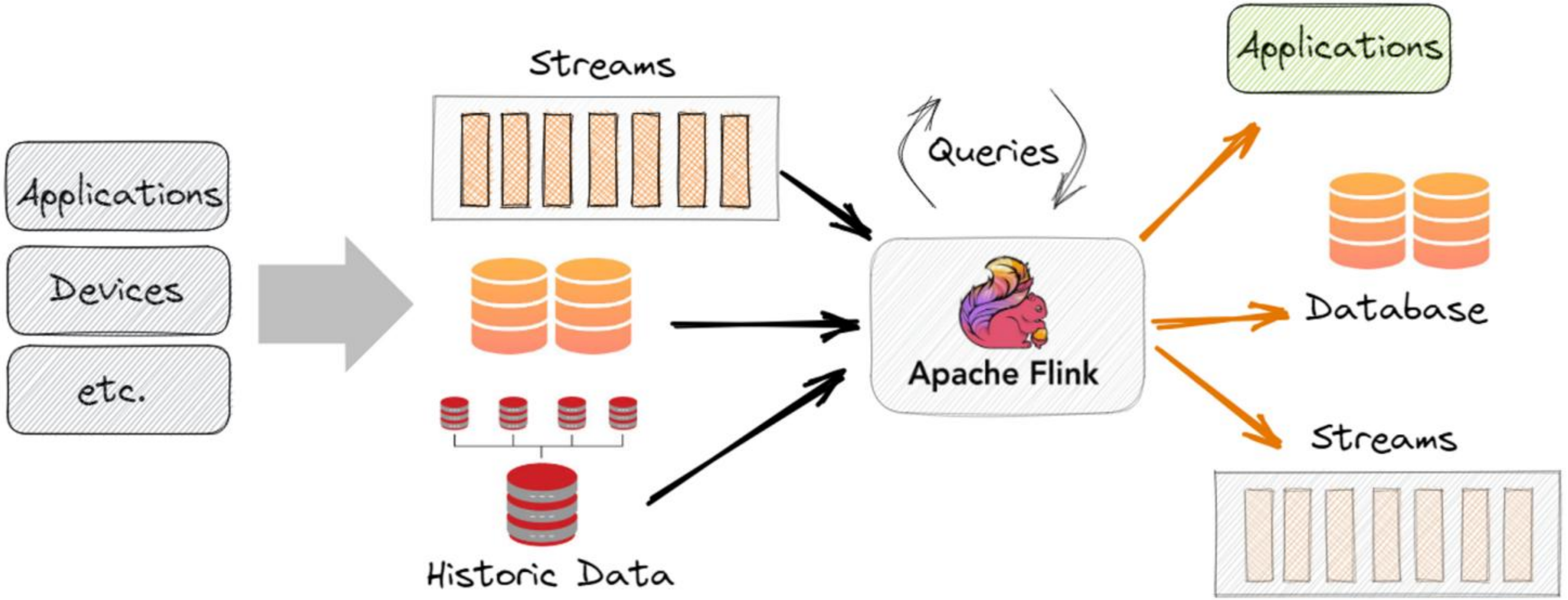
BASIC ARCHITECTURE PATTERNS



BASIC ARCHITECTURE PATTERNS



BASIC ARCHITECTURE PATTERNS



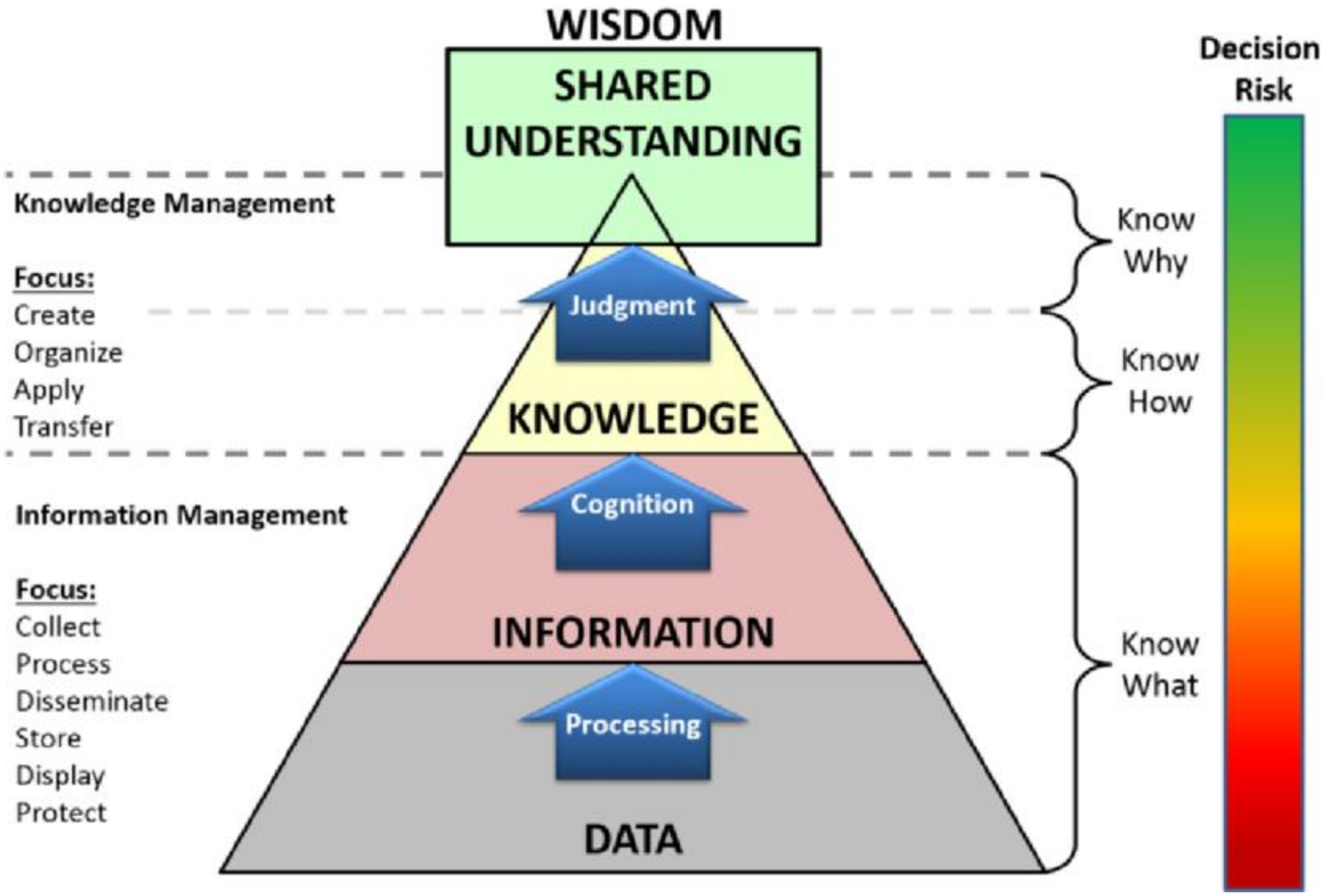
Q&A AND OPEN DISCUSSION

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

SINGLE LARGE IMAGE



BULLETED LIST

- Data that has been processed, structured, or organized to provide context or meaning.
- Characteristics:
  - Answers questions like who, what, when, and where.
  - Give insights into the behaviors and characteristics of entities in the real world
  - Provides insights but lacks a deeper level of understanding.
- 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

IMAGE RIGHT

Chen’s four levels correspond to the three level architecture proposed for data base development

- Has a wide range of applications, as we shall see

The correspondences are:

- **External views:** a specific group of users’ view of the data. Data at this level is *discovered* through investigation
- **Conceptual level:** a common, domain specific *defined* view of the data that is rigorous and complete
- **Internal level:** How the data is *structured*: relational model or dimensional model or network model for example
- **Physical level:** The specific *implementation* of the structure defined at the internal level

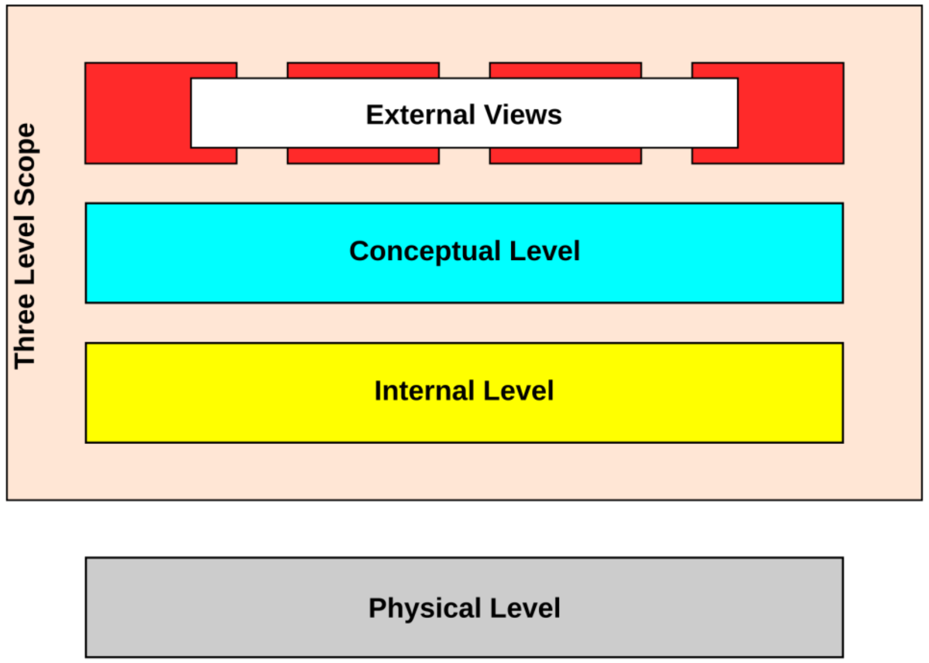
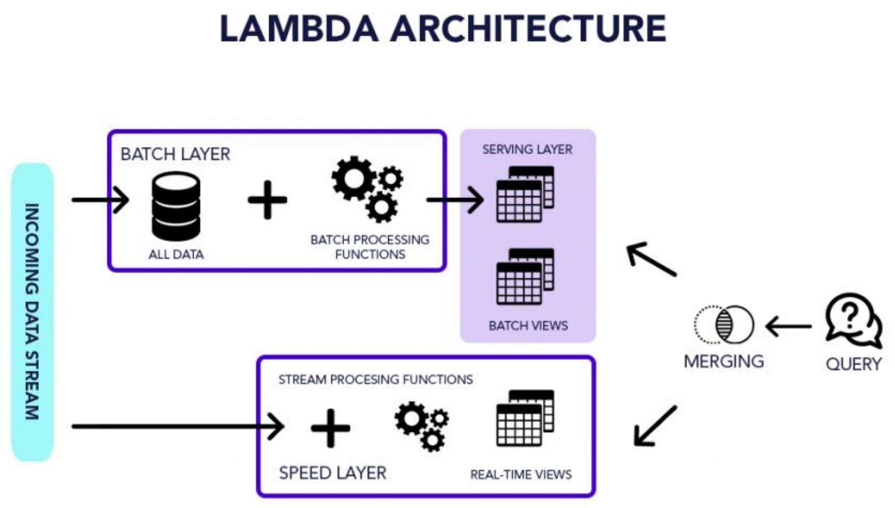


IMAGE BOTTOM

- Used when we have streaming data and historical data
  - We set up a two part architecture
  - A real time speed layer, that could be part of a OLTP transactional system
  - The real time layer keeps track of data being used in transactions
  - The batch storage layer keeps historical data for training and BI



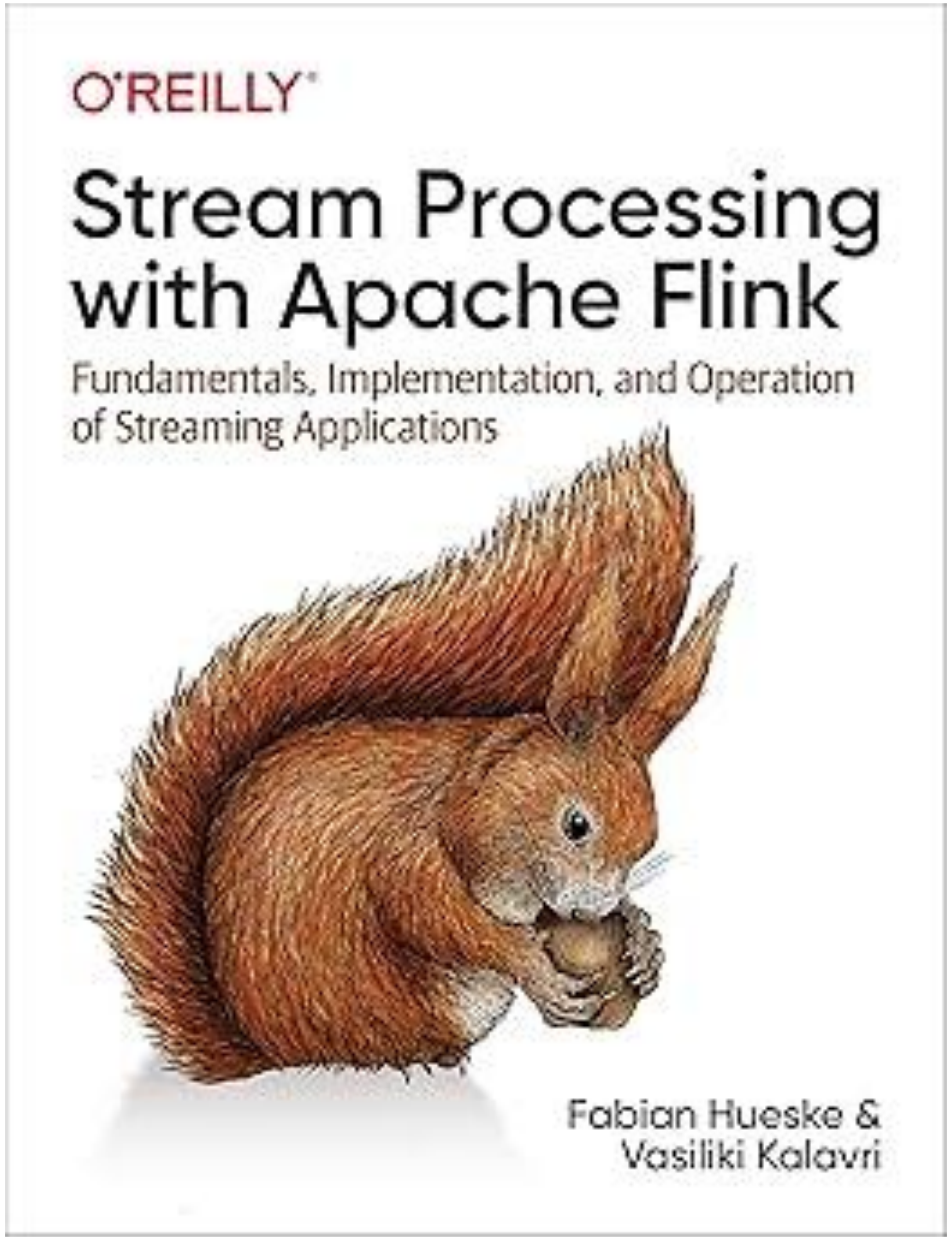
APACHE FLINK BASICS

What is Apache Flink?

Key Features and Components

Basic Architecture

Development Environment Setup


O'REILLY

Stream Processing with Apache Flink

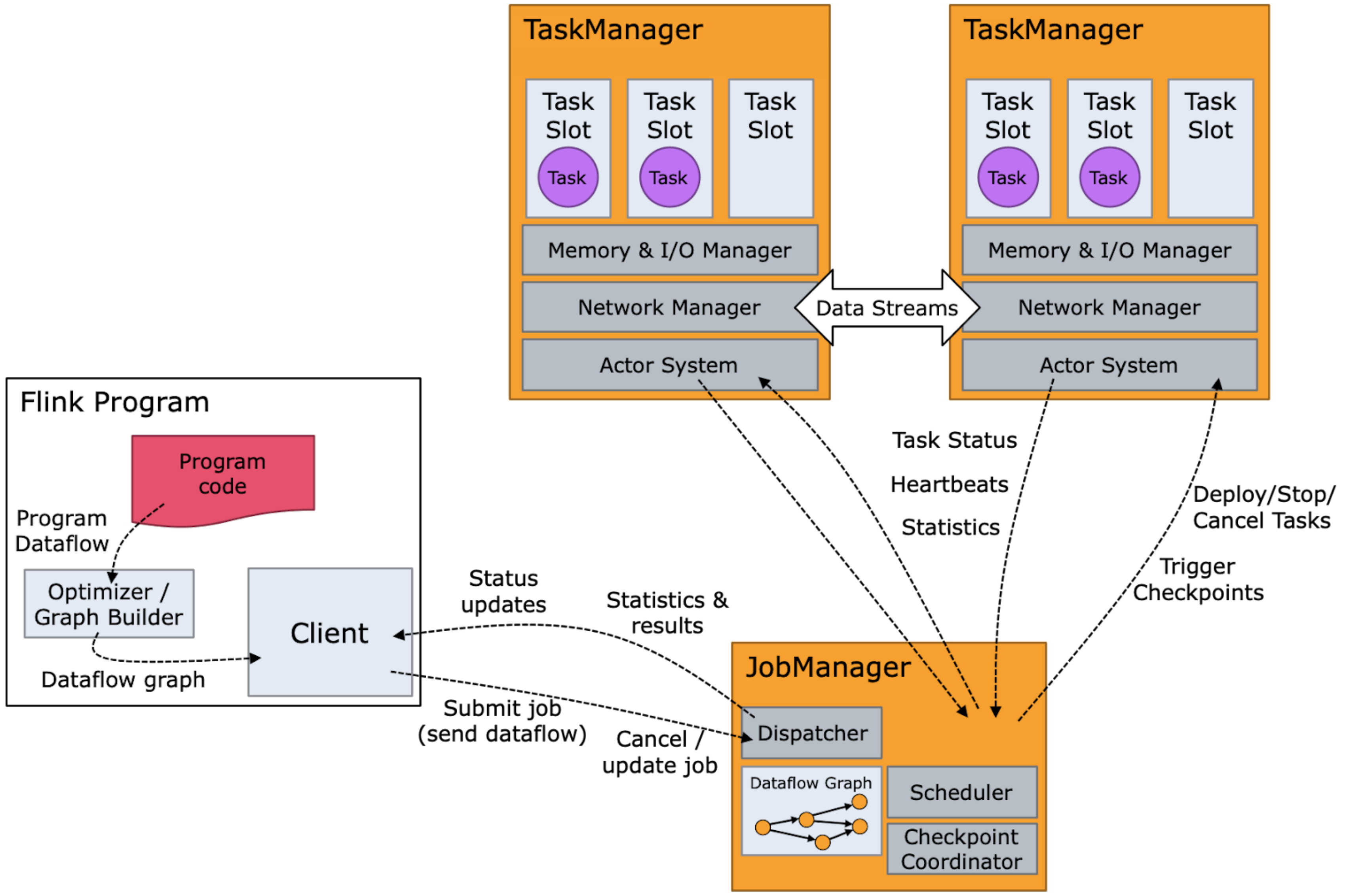
Fundamentals, Implementation, and Operation of Streaming Applications

Fabian Hueske & Vasiliki Kalavri

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.

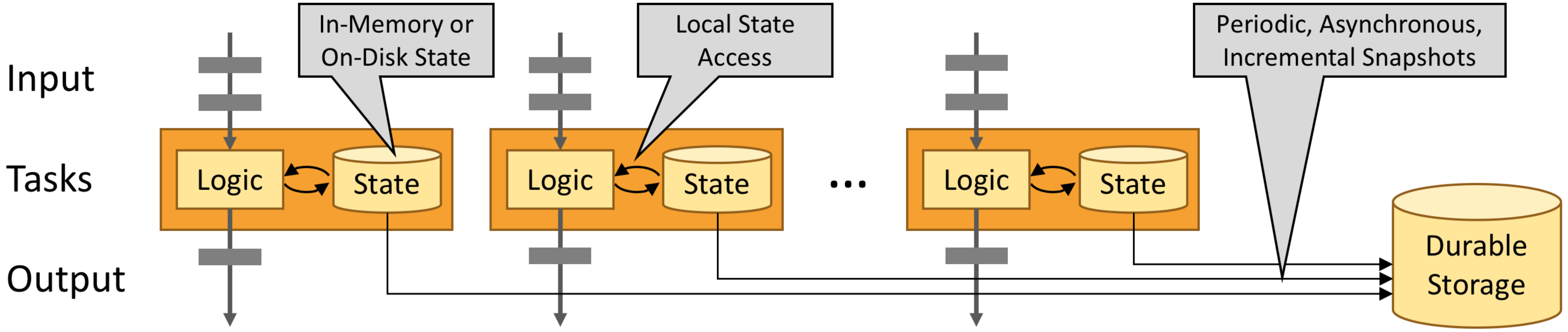
BASIC ARCHITECTURE - SCALE

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,

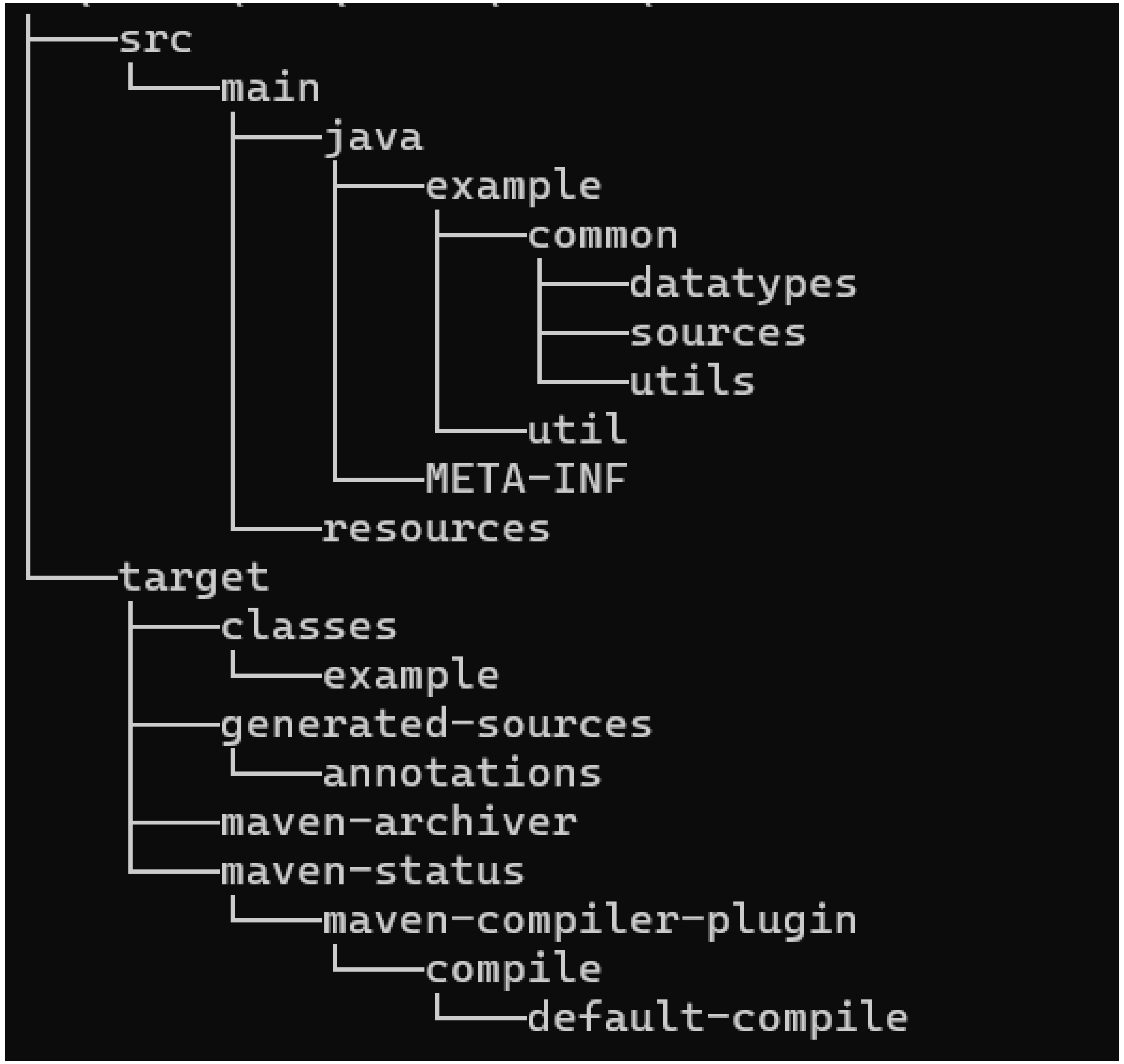
•Load/create the initial data,

•Specify transformations on this data,

•Specify where to put the results of your computations,

•Trigger the program execution

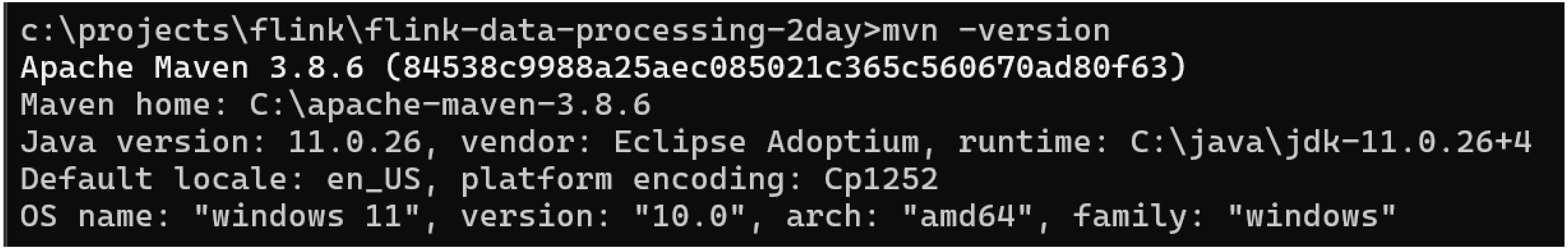
PROJECT STRUCTURE



```
├───src
│   └───main
│       ├───java
│       │   ├───example
│       │   │   ├───common
│       │   │   │   ├───datatypes
│       │   │   │   ├───sources
│       │   │   │   └───utils
│       │   │   └───util
│       │   └───META-INF
│       └───resources
└───target
    ├───classes
    │   └───example
    ├───generated-sources
    │   └───annotations
    ├───maven-archiver
    └───maven-status
        └───maven-compiler-plugin
            └───compile
                └───default-compile
```

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
```

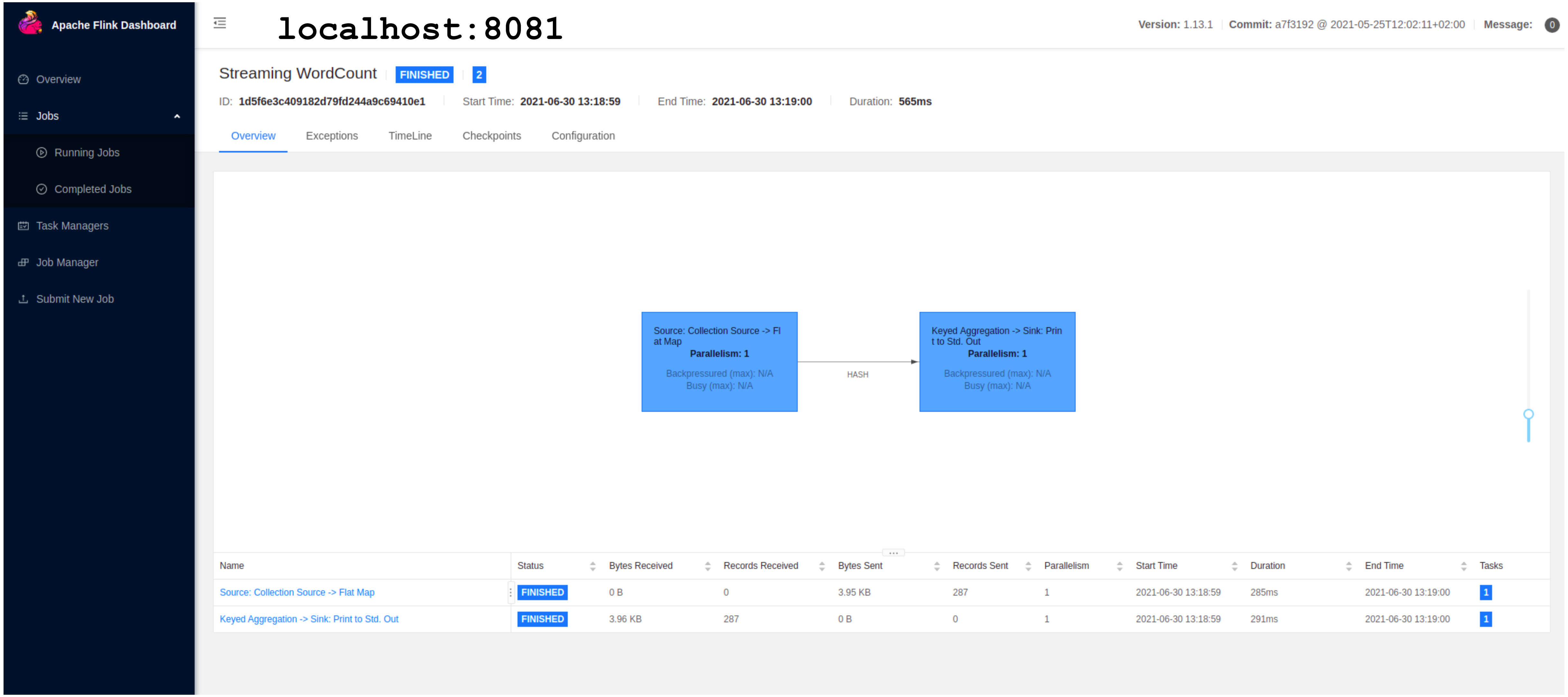
EXECUTION

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



localhost:8081

Apache Flink Dashboard

- Overview
- Jobs
  - Running Jobs
  - Completed Jobs
- Task Managers
- Job Manager
- Submit New Job

Version: 1.13.1 | Commit: a7f3192 @ 2021-05-25T12:02:11+02:00 | Message: 0

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

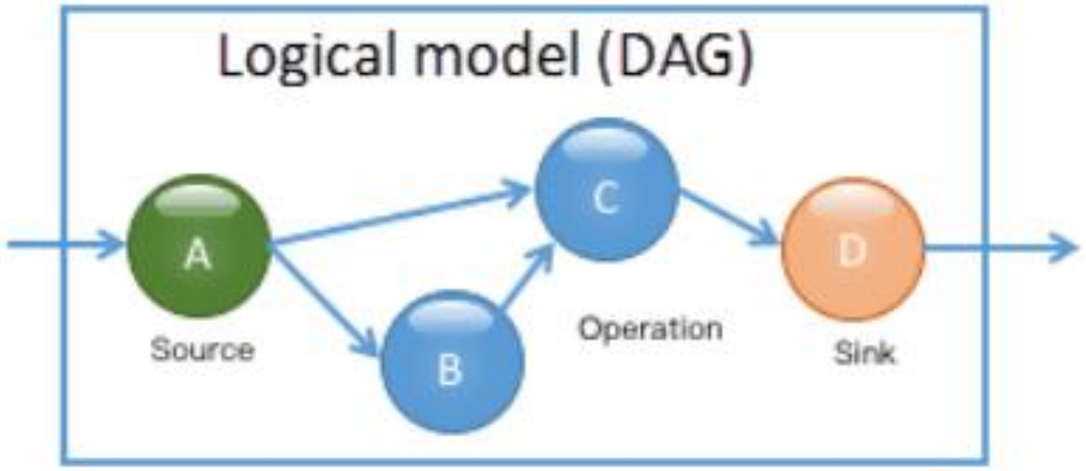
Working with DataStreams

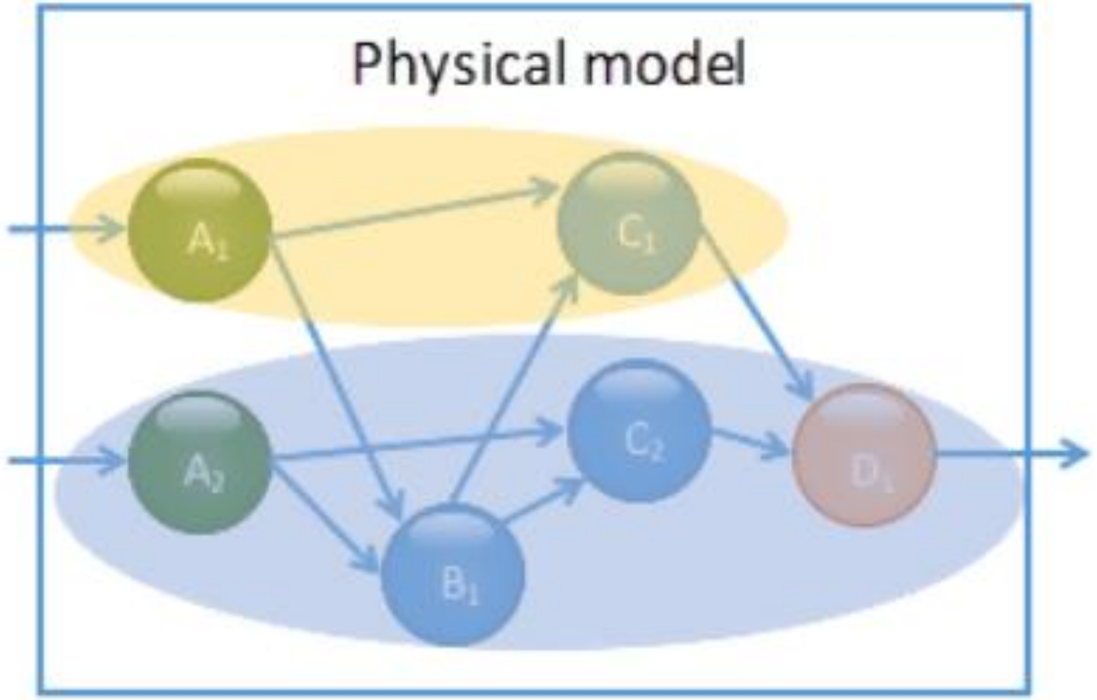
- DataStream Basics
  - Creating DataStreams
  - Basic Operations
  - Data Types
  - Simple Transformations
- Common Operations
  - Map and FlatMap
  - Filter Operations
  - Basic Aggregations
  - Field Selection

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).

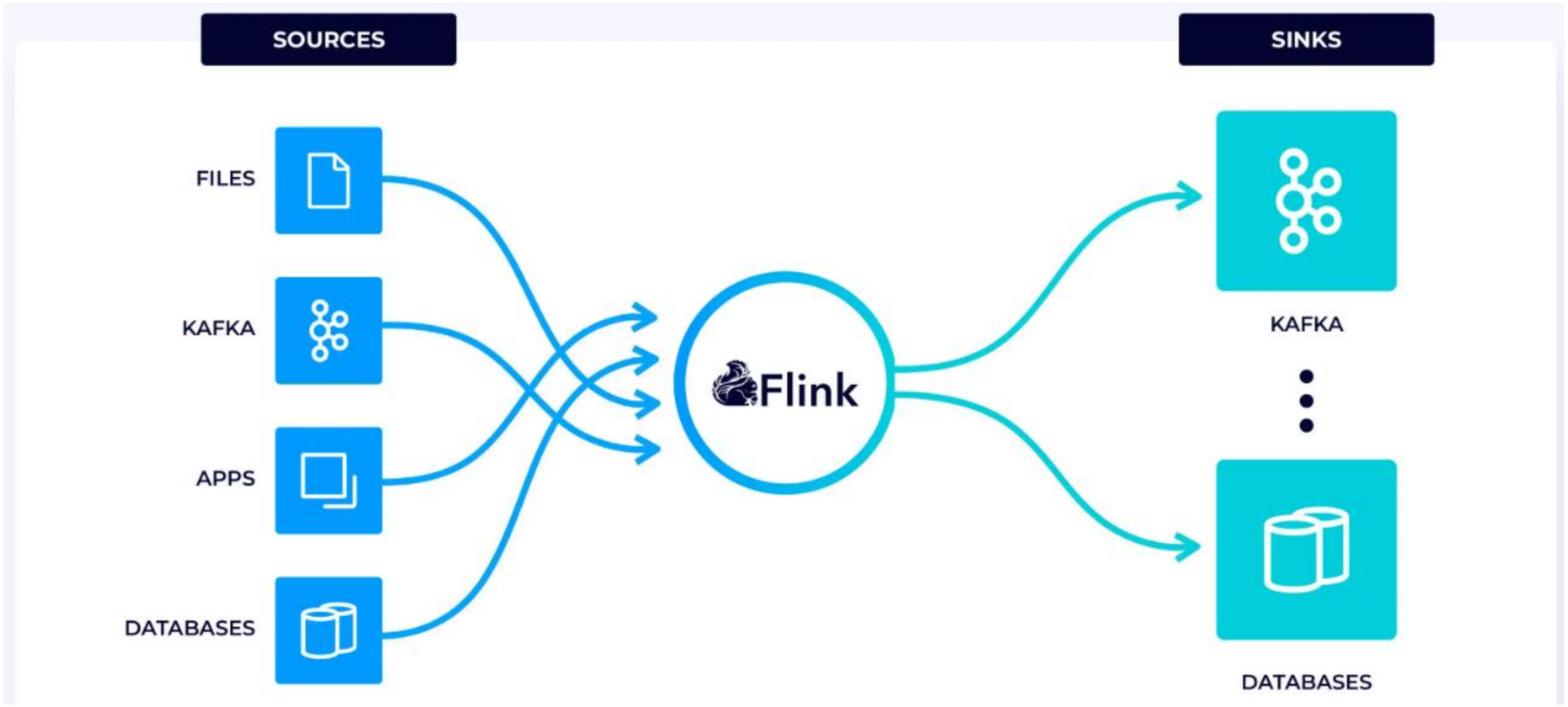
DATASTREAM BASICS





Footer

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
```

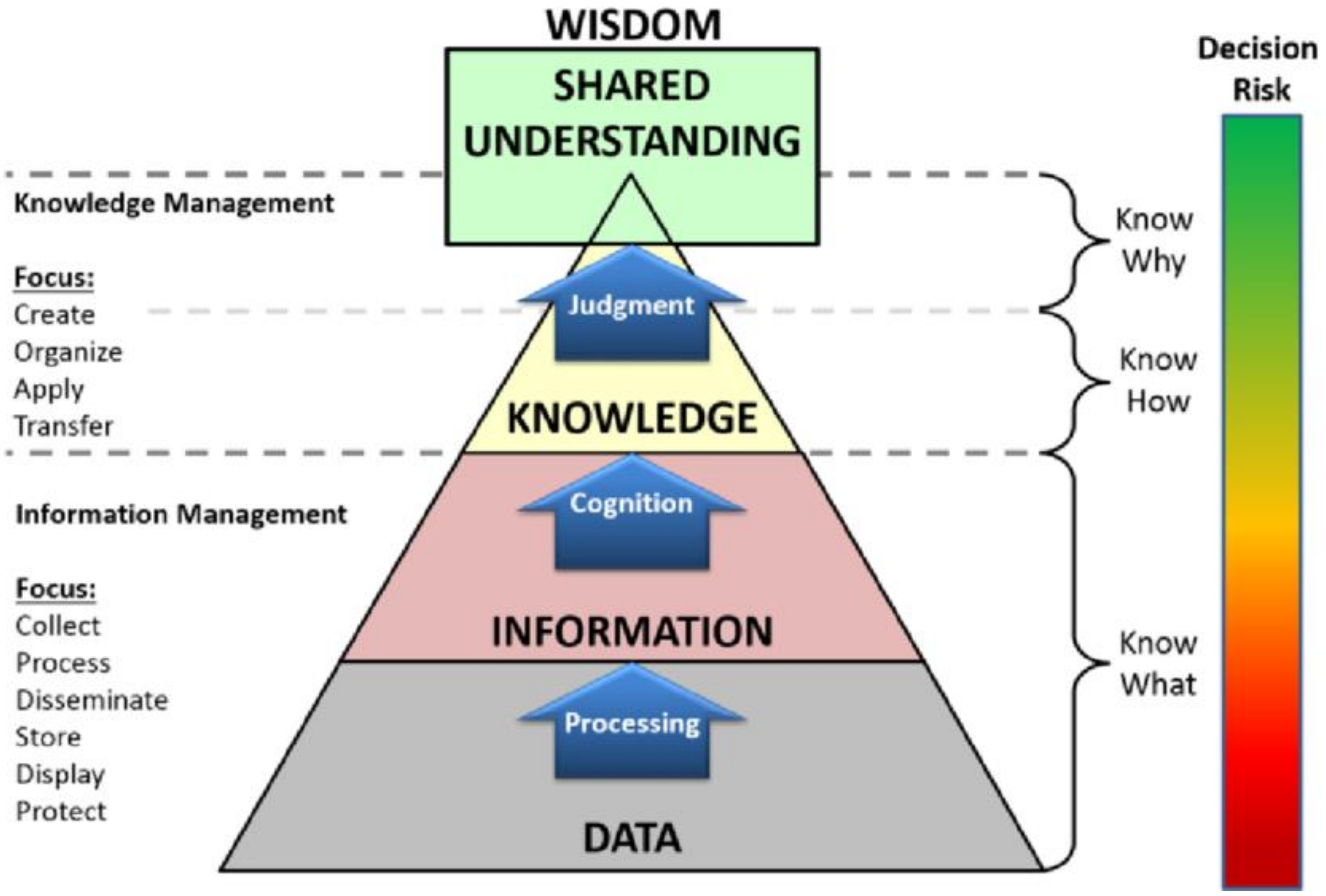
Q&A AND OPEN DISCUSSION

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

SINGLE LARGE IMAGE



BULLETED LIST

- Data that has been processed, structured, or organized to provide context or meaning.
- Characteristics:
  - Answers questions like who, what, when, and where.
  - Give insights into the behaviors and characteristics of entities in the real world
  - Provides insights but lacks a deeper level of understanding.
- 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

IMAGE RIGHT

Chen’s four levels correspond to the three level architecture proposed for data base development

- Has a wide range of applications, as we shall see

The correspondences are:

- **External views:** a specific group of users’ view of the data. Data at this level is *discovered* through investigation
- **Conceptual level:** a common, domain specific *defined* view of the data that is rigorous and complete
- **Internal level:** How the data is *structured*: relational model or dimensional model or network model for example
- **Physical level:** The specific *implementation* of the structure defined at the internal level

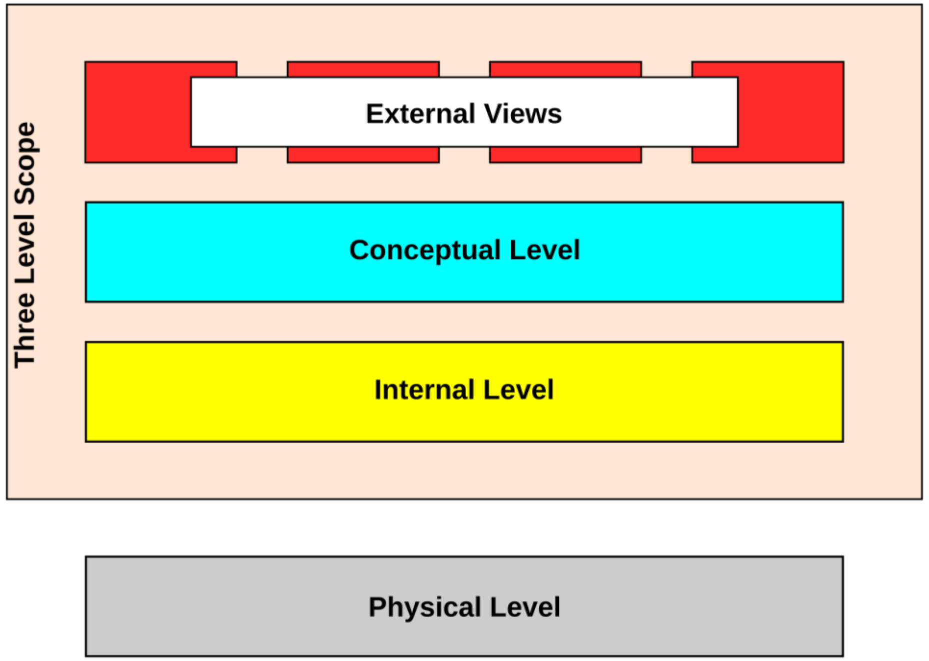
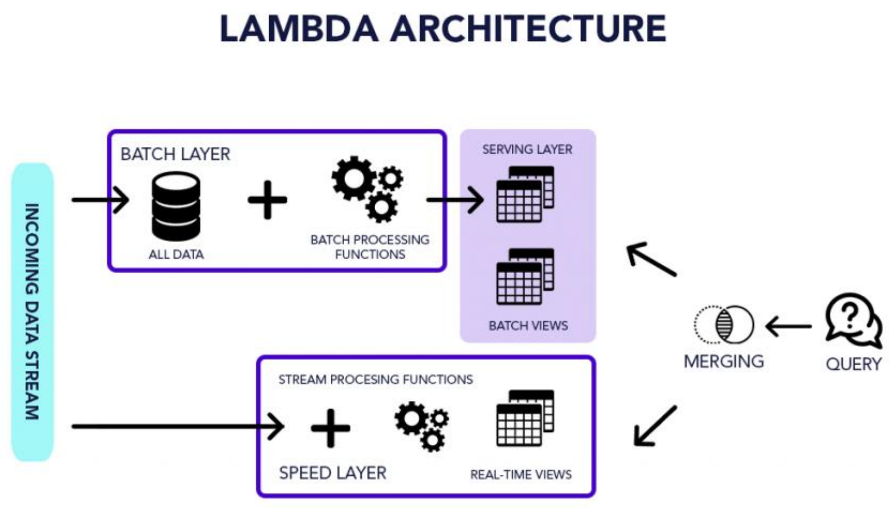


IMAGE BOTTOM

- Used when we have streaming data and historical data
  - We set up a two part architecture
  - A real time speed layer, that could be part of a OLTP transactional system
  - The real time layer keeps track of data being used in transactions
  - The batch storage layer keeps historical data for training and BI



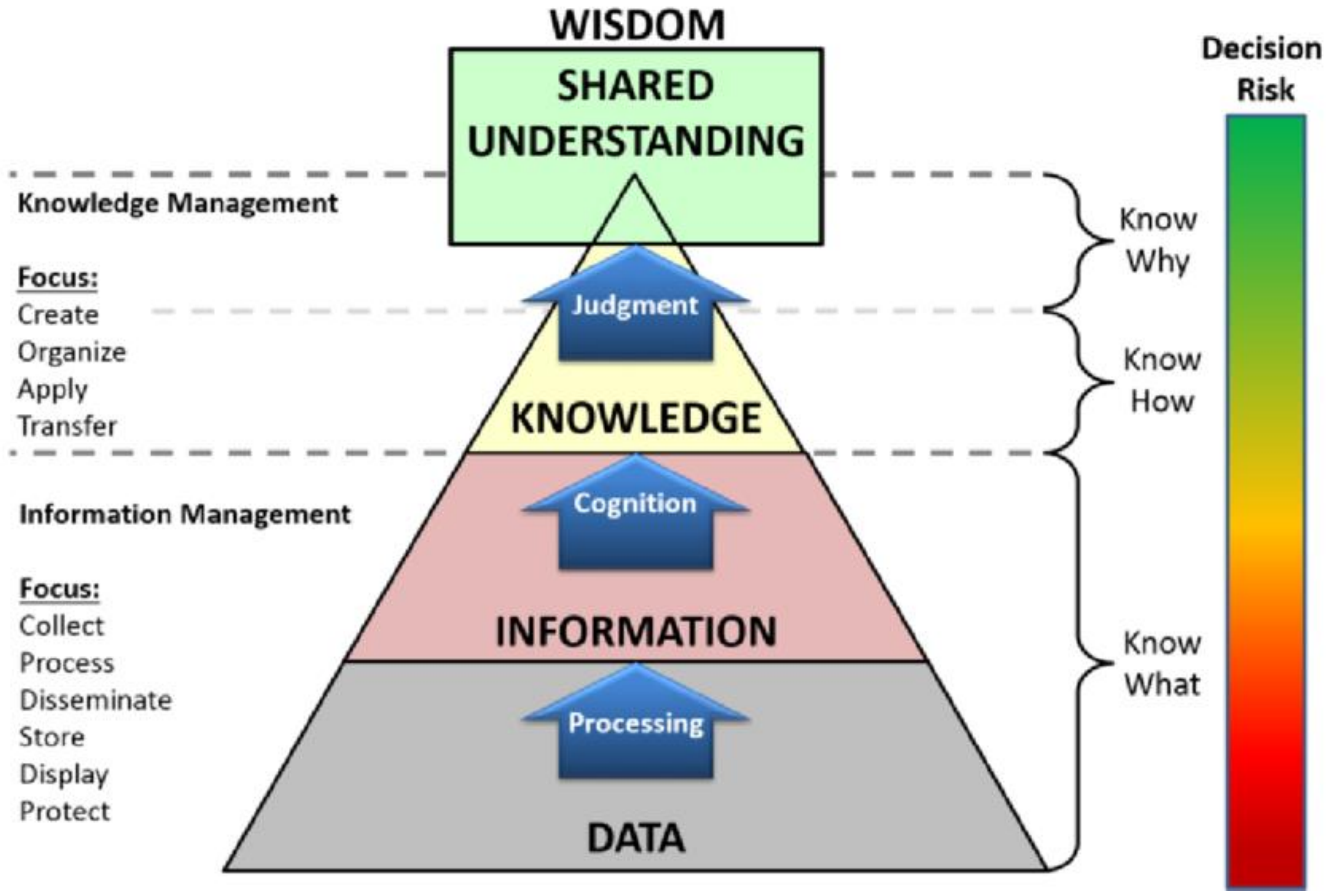
Q&A AND OPEN DISCUSSION

Time and Windows

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

SINGLE LARGE IMAGE



BULLETED LIST

- Data that has been processed, structured, or organized to provide context or meaning.
- Characteristics:
  - Answers questions like who, what, when, and where.
  - Give insights into the behaviors and characteristics of entities in the real world
  - Provides insights but lacks a deeper level of understanding.
- 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

IMAGE RIGHT

Chen’s four levels correspond to the three level architecture proposed for data base development

- Has a wide range of applications, as we shall see

The correspondences are:

- **External views:** a specific group of users’ view of the data. Data at this level is *discovered* through investigation
- **Conceptual level:** a common, domain specific *defined* view of the data that is rigorous and complete
- **Internal level:** How the data is *structured*: relational model or dimensional model or network model for example
- **Physical level:** The specific *implementation* of the structure defined at the internal level

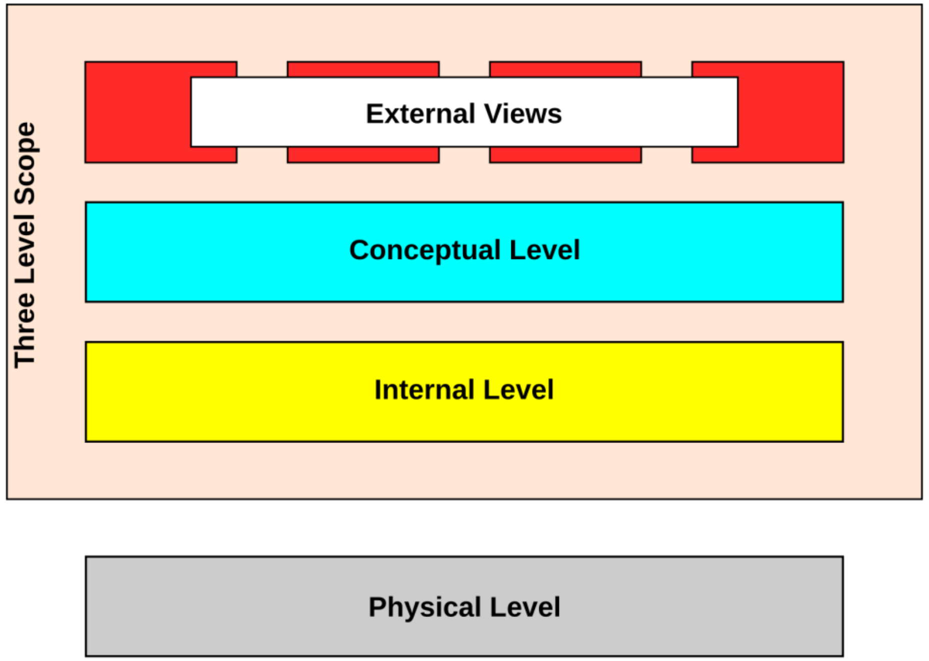
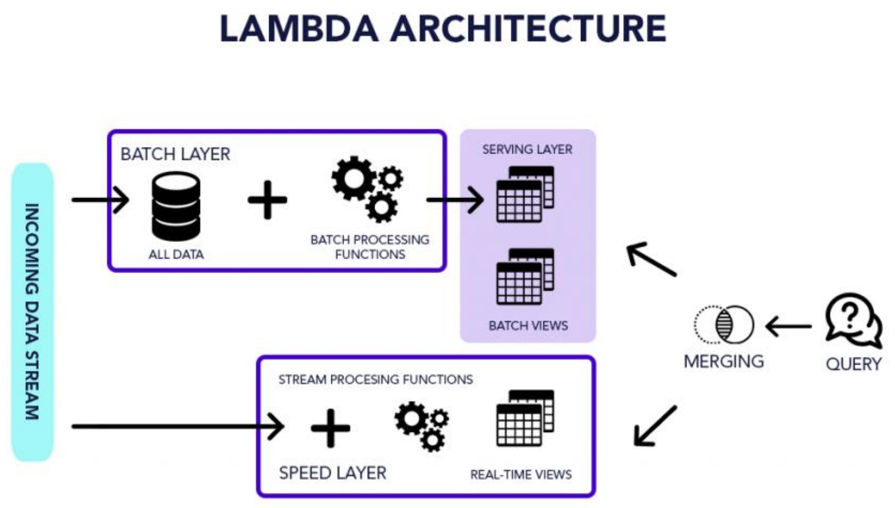


IMAGE BOTTOM

- Used when we have streaming data and historical data
  - We set up a two part architecture
  - A real time speed layer, that could be part of a OLTP transactional system
  - The real time layer keeps track of data being used in transactions
  - The batch storage layer keeps historical data for training and BI



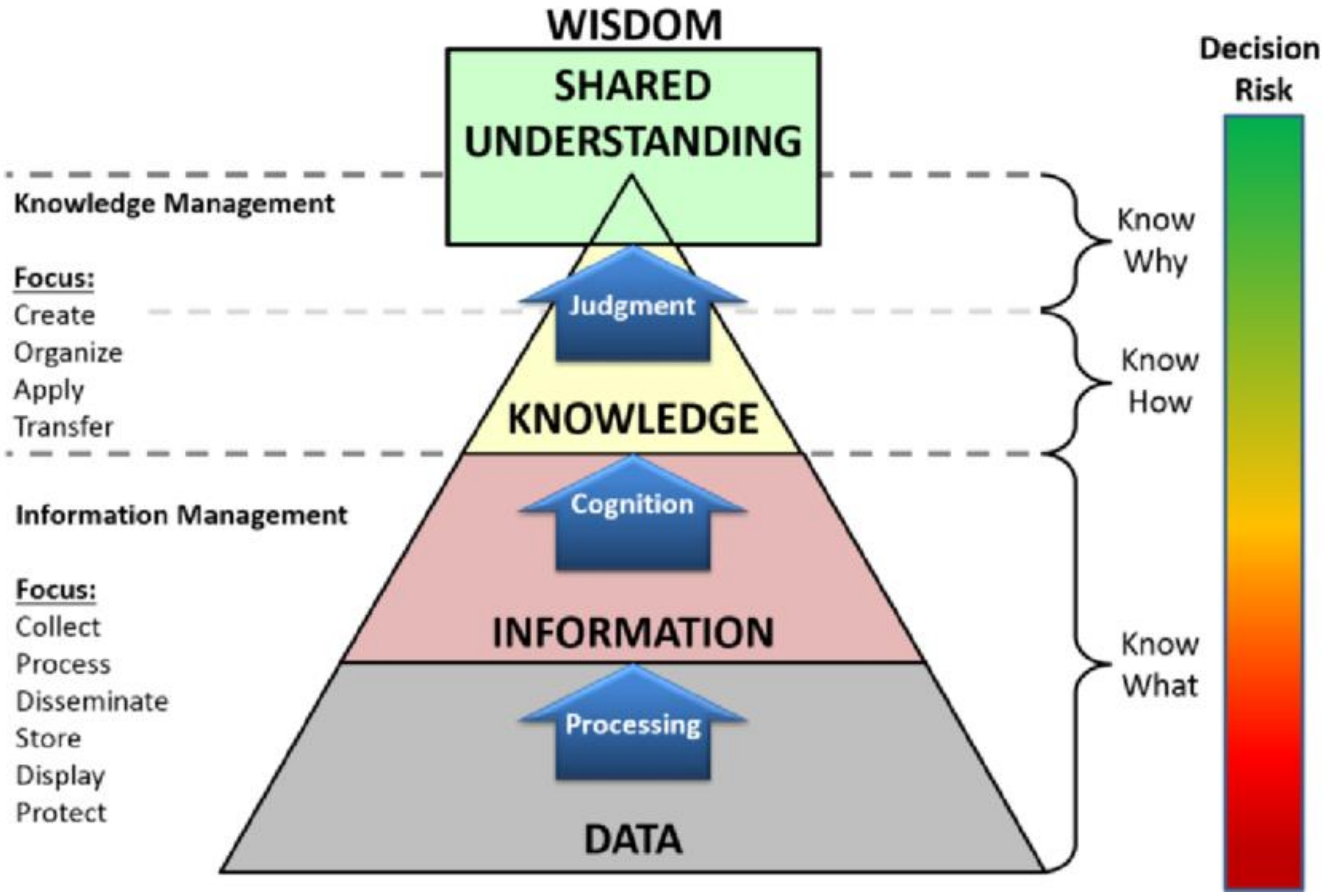
Q&A AND OPEN DISCUSSION

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

SINGLE LARGE IMAGE



BULLETED LIST

- Data that has been processed, structured, or organized to provide context or meaning.
- Characteristics:
  - Answers questions like who, what, when, and where.
  - Give insights into the behaviors and characteristics of entities in the real world
  - Provides insights but lacks a deeper level of understanding.
- 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

IMAGE RIGHT

Chen’s four levels correspond to the three level architecture proposed for data base development

- Has a wide range of applications, as we shall see

The correspondences are:

- **External views:** a specific group of users’ view of the data. Data at this level is *discovered* through investigation
- **Conceptual level:** a common, domain specific *defined* view of the data that is rigorous and complete
- **Internal level:** How the data is *structured*: relational model or dimensional model or network model for example
- **Physical level:** The specific *implementation* of the structure defined at the internal level

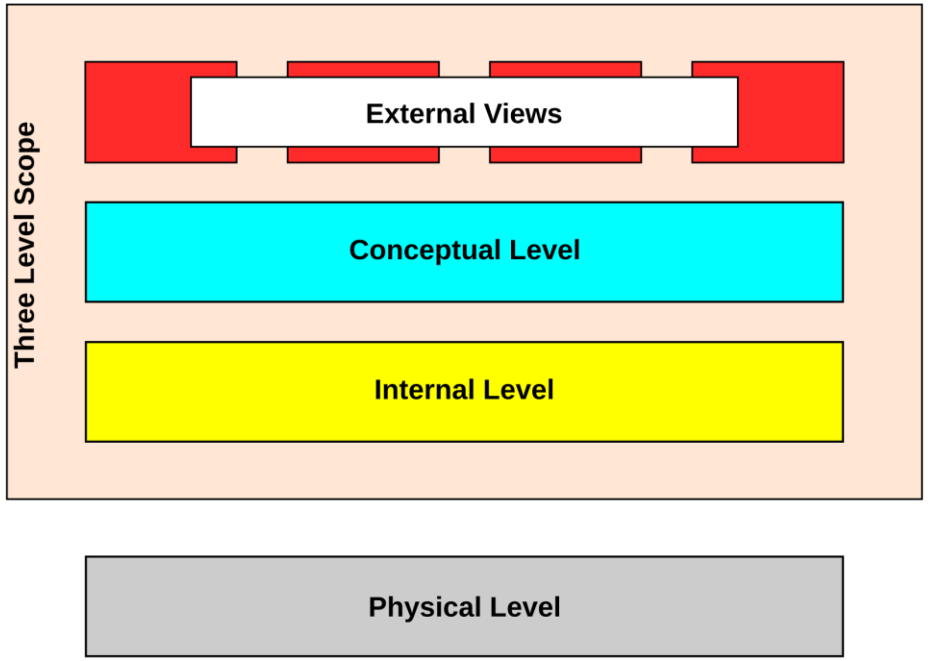
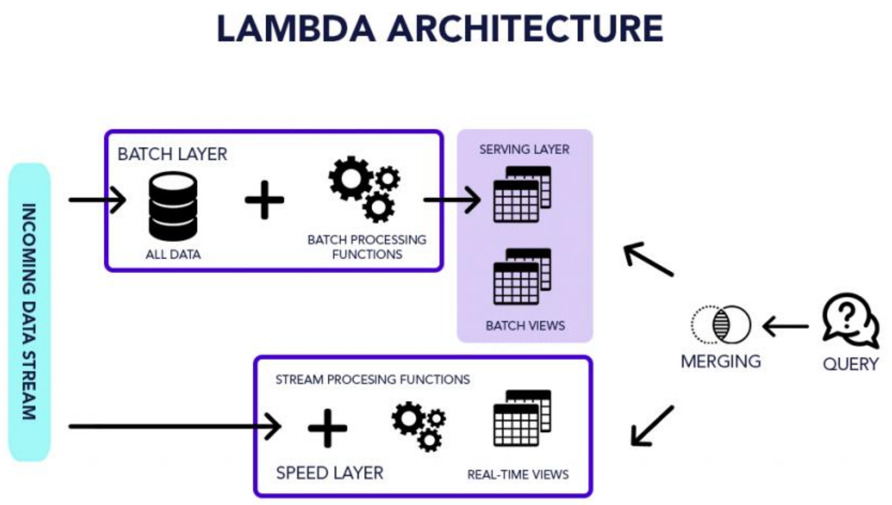


IMAGE BOTTOM

- Used when we have streaming data and historical data
  - We set up a two part architecture
  - A real time speed layer, that could be part of a OLTP transactional system
  - The real time layer keeps track of data being used in transactions
  - The batch storage layer keeps historical data for training and BI



Q&A AND OPEN DISCUSSION

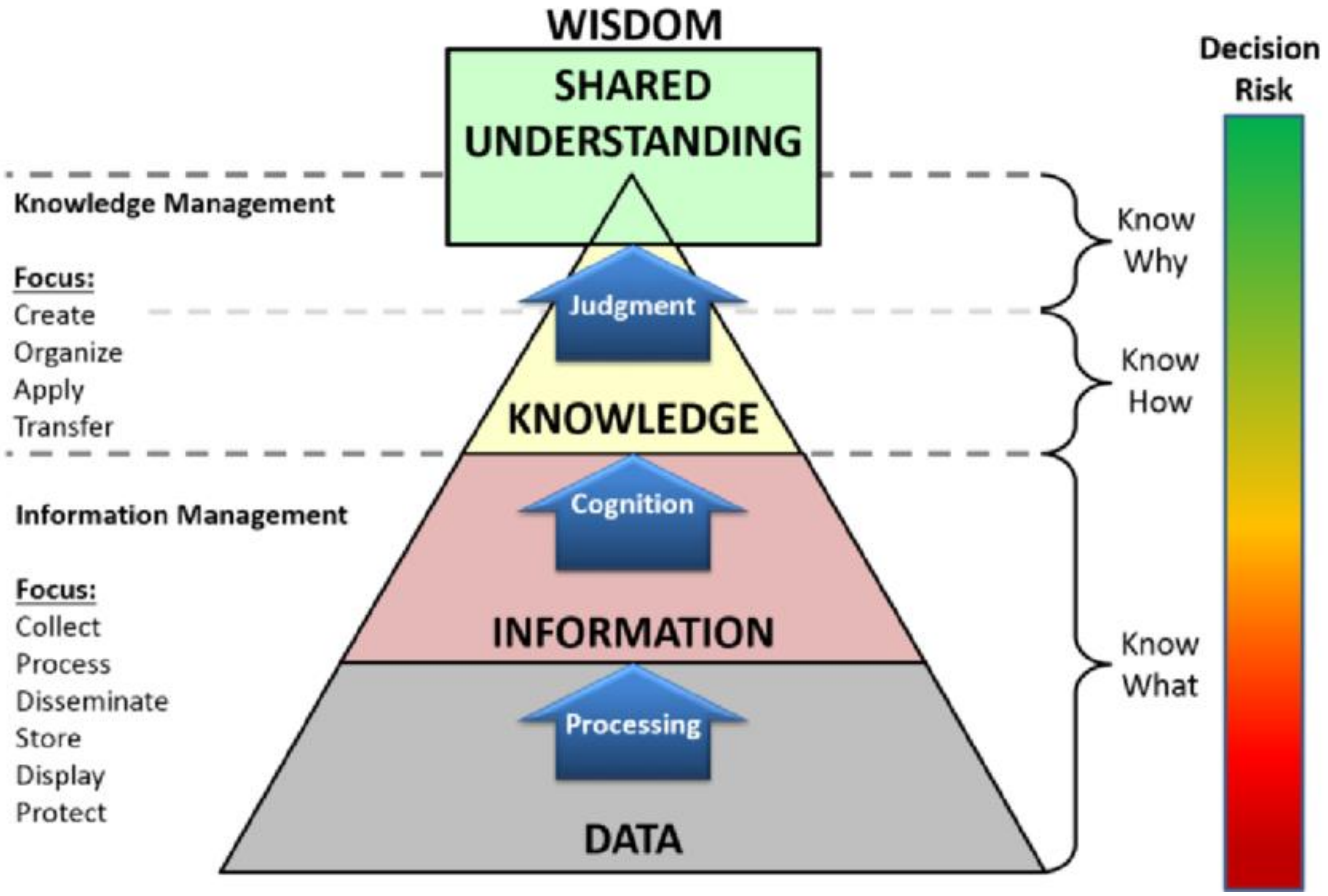
Deployment and Next Steps

1

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

SINGLE LARGE IMAGE



BULLETED LIST

- Data that has been processed, structured, or organized to provide context or meaning.
- Characteristics:
    - Answers questions like who, what, when, and where.
    - Give insights into the behaviors and characteristics of entities in the real world
    - Provides insights but lacks a deeper level of understanding.
- 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

IMAGE RIGHT

Chen's four levels correspond to the three level architecture proposed for data base development

- Has a wide range of applications, as we shall see

The correspondences are:

- **External views:** a specific group of users' view of the data. Data at this level is *discovered* through investigation
- **Conceptual level:** a common, domain specific *defined* view of the data that is rigorous and complete
- **Internal level:** How the data is *structured*: relational model or dimensional model or network model for example
- **Physical level:** The specific *implementation* of the structure defined at the internal level

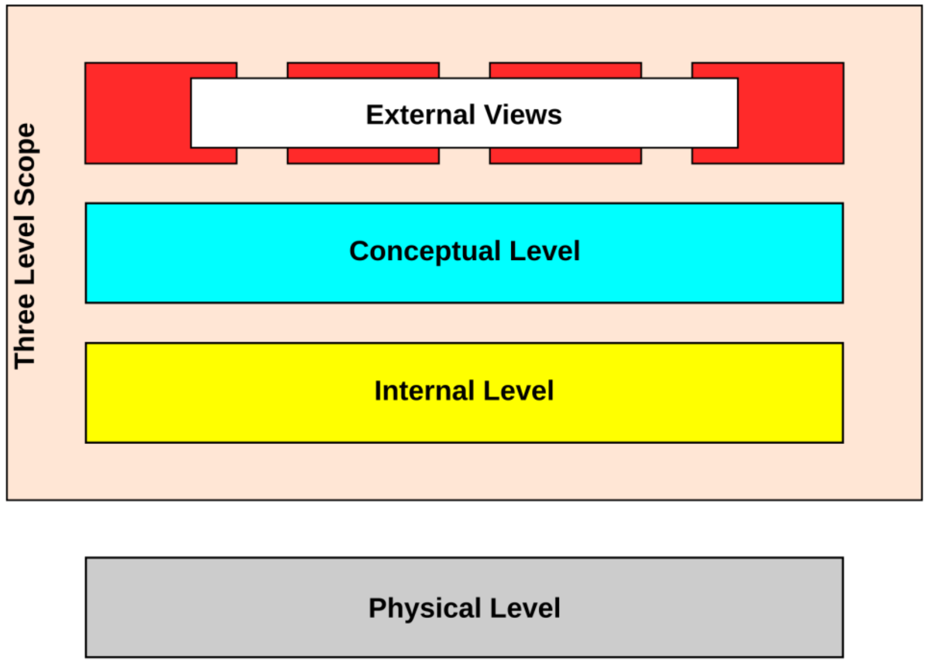
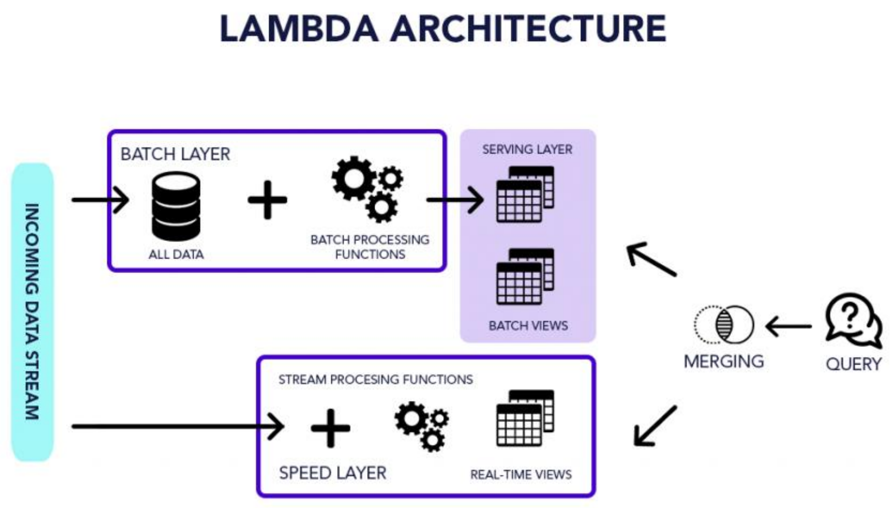


IMAGE BOTTOM

- Used when we have streaming data and historical data
  - We set up a two part architecture
  - A real time speed layer, that could be part of a OLTP transactional system
  - The real time layer keeps track of data being used in transactions
  - The batch storage layer keeps historical data for training and BI



Q&A AND OPEN DISCUSSION