Data Processing Architectures

Dr. Sonali Agarwal
Associate Professor, Department of IT
IIIT Allahabad

Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- From a practical viewpoint, Internet of Things (IoT) represents any device that is connected to the Internet.
- This includes your PC, mobile phone, smart watch, smart thermostat, smart refrigerator, connected automobile, heart monitoring implants, and anything else that connects to the Internet and sends or receives data.
- The number of connected devices grows every day, as does the amount of data collected from them. Often this data is being collected in highly constrained, sometimes high-latency environments.

IoT / Event-Driven Architecture

Concept: Designed specifically for billions of distributed devices and sensors

Field Gateway

Edge computing for near-device preprocessing, filtering, and aggregation

Cloud Gateway

Reliable message ingestion using Kafka and other protocols

Stream Processors

Real-time analytics engines for immediate insights and alerts

Cold Storage

Data lake for archiving historical data (HDFS, S3, Azure Data Lake)

IoT Architecture Example: Smart City Traffic

Components

- Traffic cameras
- Road sensors
- Vehicle counters
- Weather stations
- Signal control systems

Data Characteristics

- High frequency (readings every few seconds)
- Geographically distributed
- Mixed data types (video, numeric, text)

Edge Processing

Field gateways filter and aggregate sensor data to reduce bandwidth needs

Data Ingestion

Kafka ingests data streams from thousands of sensors across the city

Real-time Analytics

Spark Streaming detects congestion patterns and triggers signal adjustments

Historical Analysis

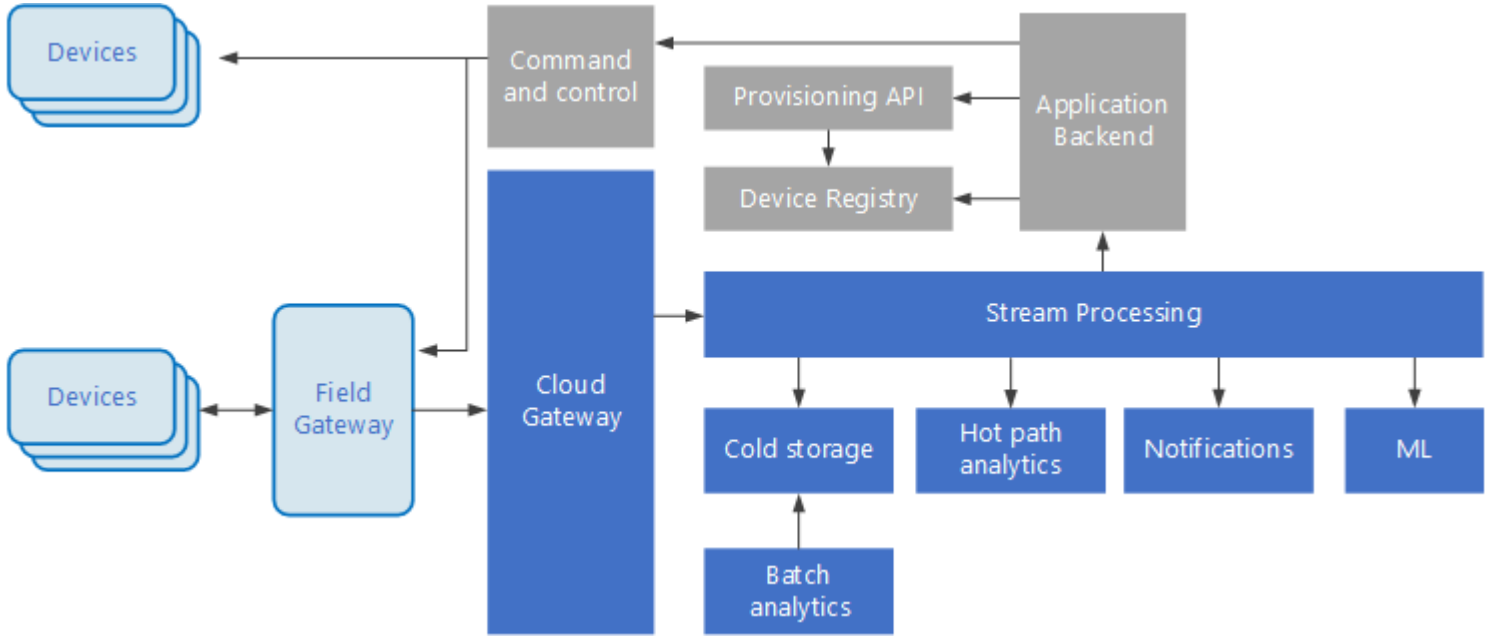
Cold storage retains 1+ year of traffic data for urban planning

Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- In these cases, data is sent from low-latency environments by thousands or millions of devices, requiring the ability to rapidly ingest the data and process accordingly.
- Therefore, proper planning is required to handle these constraints and unique requirements.

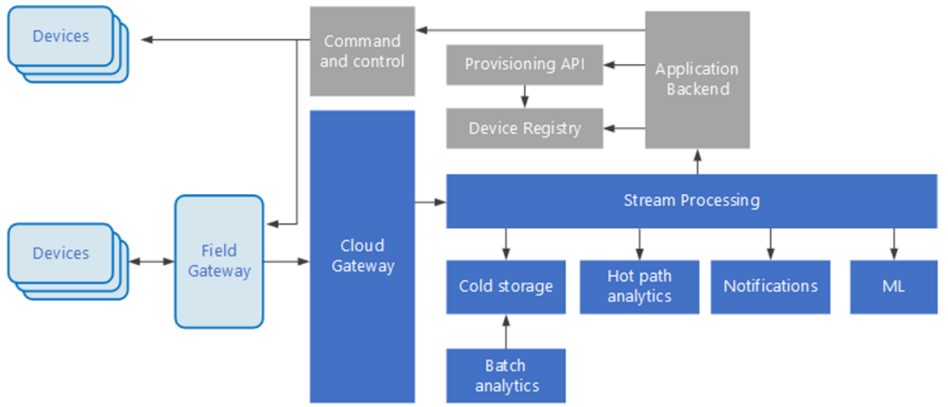
Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- Event-driven architectures are central to IoT solutions.
- The following diagram shows a possible logical architecture for IoT.
- The diagram emphasizes the event-streaming components of the architecture.



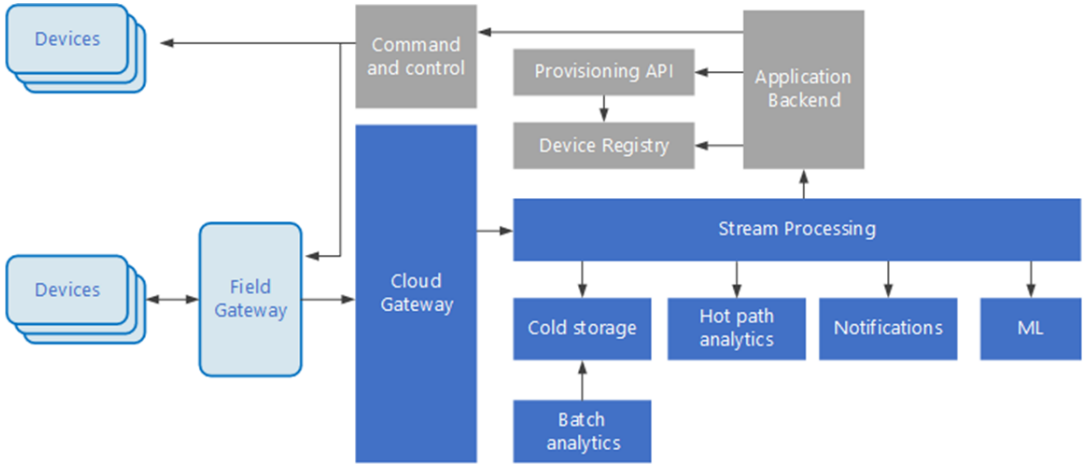
Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- The **cloud gateway** ingests device events at the cloud boundary, using a reliable, low latency messaging system.
- Devices might send events directly to the cloud gateway, or through a **field gateway**. A field gateway is a specialized device or software, usually collocated with the devices, that receives events and forwards them to the cloud gateway. The field gateway might also preprocess the raw device events, performing functions such as filtering, aggregation, or protocol transformation.
- After ingestion, events go through one or more **stream processors** that can route the data (for example, to storage) or perform analytics and other processing.



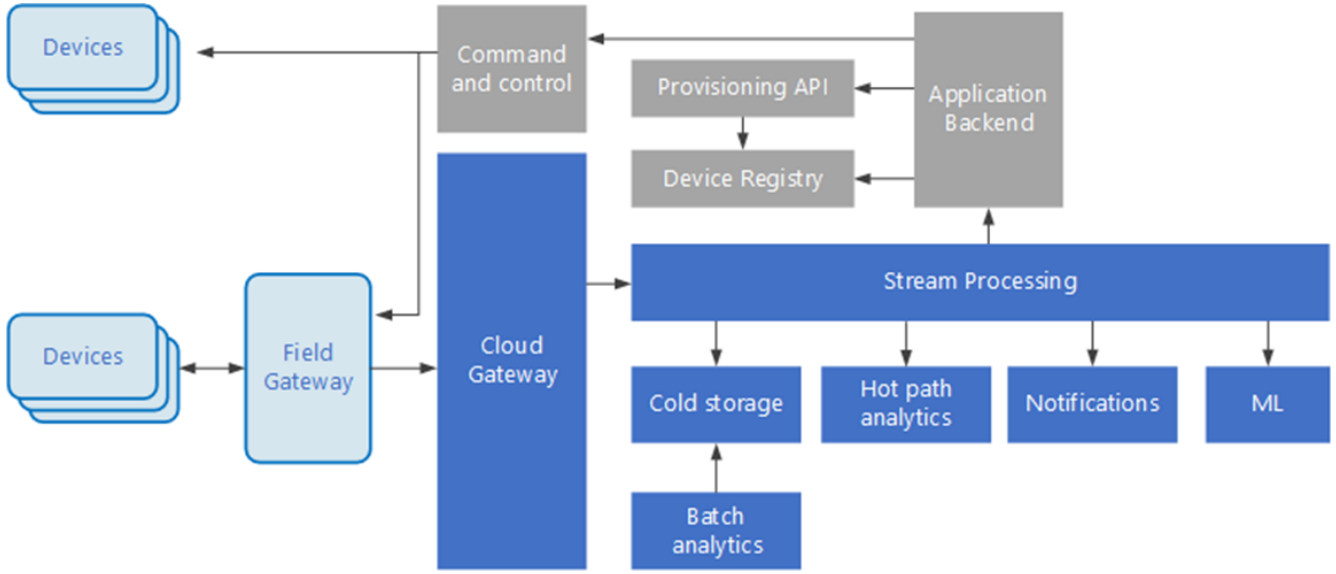
Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- The following are some common types of processing.
  - Writing event data to cold storage, for archiving or batch analytics.
  - Hot path analytics, analyzing the event stream in (near) real time, to detect anomalies, recognize patterns over rolling time windows, or trigger alerts when a specific condition occurs in the stream.
  - Handling special types of nontelemetric messages from devices, such as notifications and alarms.
  - Machine learning.

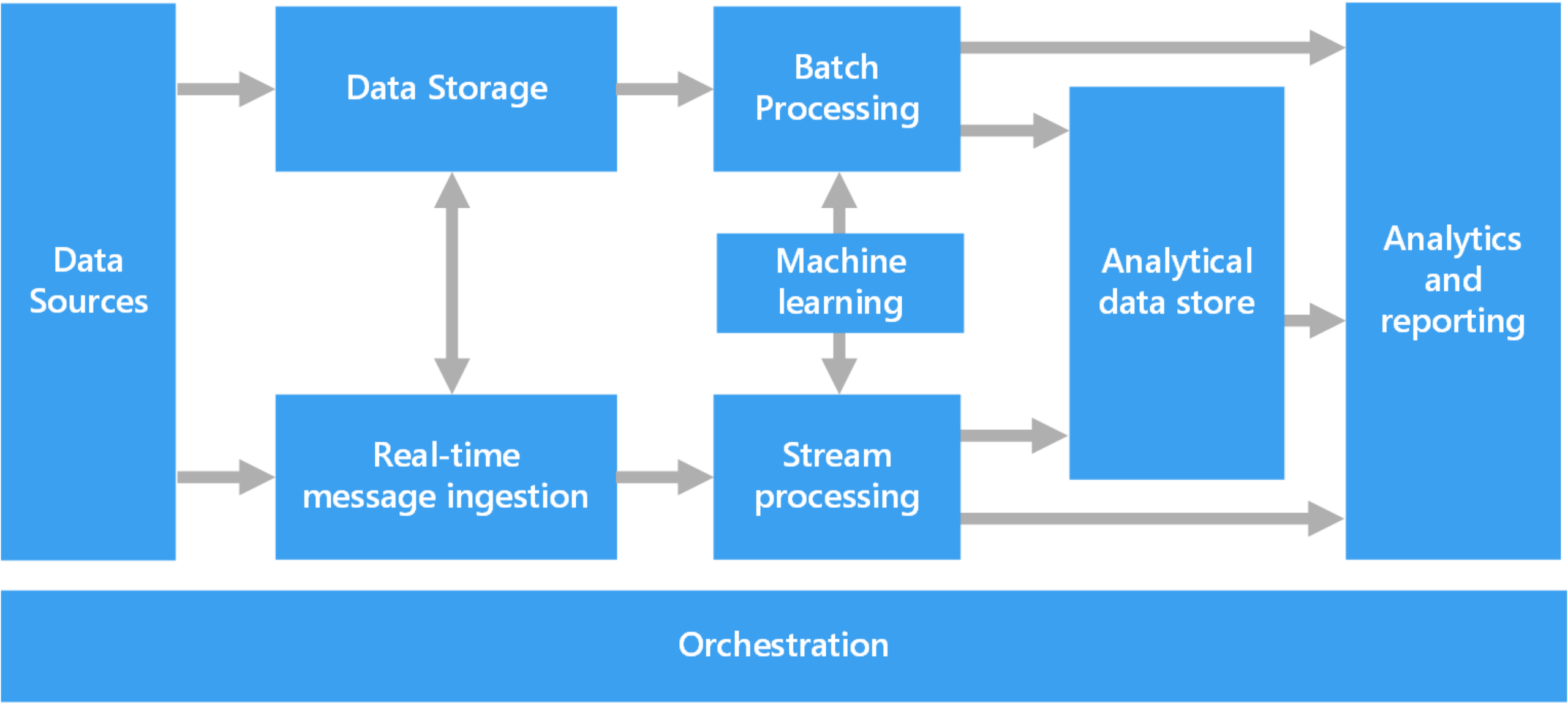


Data Processing Architectures

- **Internet of Things (IoT) based architectures**
- The **device registry** is a database of the provisioned devices, including the device IDs and usually device metadata, such as location.
- The **provisioning API** is a common external interface for provisioning and registering new devices.
- Some IoT solutions allow **command and control messages** to be sent to devices.



Components of a big data architecture

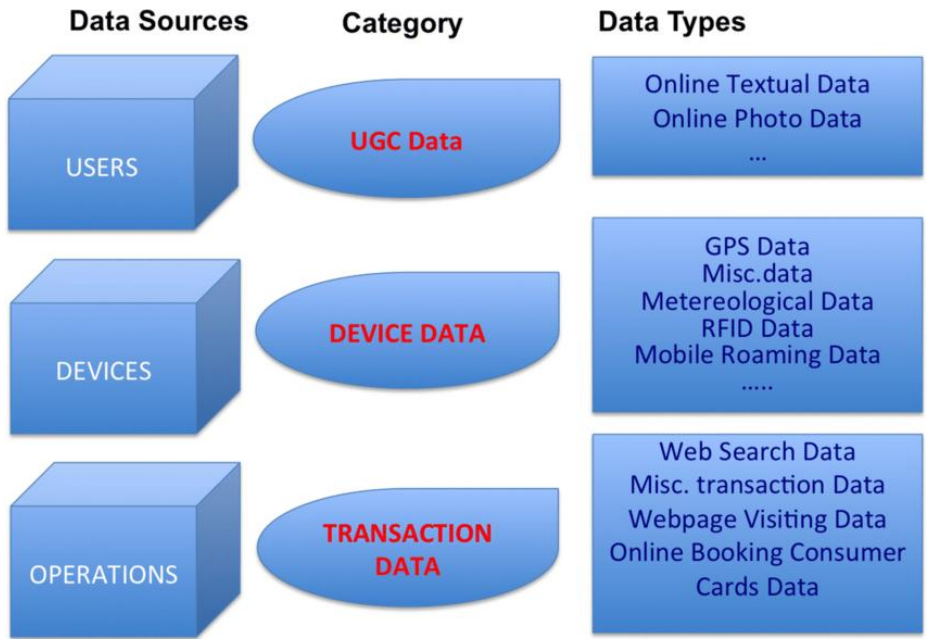


Data Source Layer

- The Big Data sources are the ones that govern the Big Data architecture.
- The designing of the architecture depends heavily on the data sources.
- The data is arriving from numerous sources that too in different formats.
- This data can be both batch data as well as real-time data.
- These sources pile up a huge amount of data in no time.
- The Big Data architecture is designed such that it is capable of handling this data.

Data Source Layer

- **Data sources**. All big data solutions start with one or more data sources.
- Examples include:
  - Application data stores, such as relational databases.
  - Static files produced by applications, such as web server log files.
  - Real-time data sources, such as IoT devices.



Data source layer

- Data from SQL, NoSQL stores (MySQL, Oracle, PostgreSQL, MongoDB, etc. – Mostly structured)
- (Semi/Un)-structured data (CRM, marketing, campaign, spend, revenue, leads, etc.)
- Web logs or other log files (weblogs, user clicks, user visits, activity, etc.)

Unstructured data

The university has 5600 students.
John's ID is number 1, he is 18 years old and already holds a B.Sc. degree.
David's ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert's ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

Semi-structured data

```
<University>
 <Student ID="1">
   <Name>John</Name>
   <Age>18</Age>
   <Degree>B.Sc.</Degree>
 </Student>
 <Student ID="2">
   <Name>David</Name>
   <Age>31</Age>
   <Degree>Ph.D. </Degree>
 </Student>
....
</University>
```

Structured data

ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

Data storage Layer

- **Data storage**. Data for batch processing operations is typically stored in a distributed file store that can hold high volumes of large files in various formats.
- As the volume of data generated and stored by companies has started to explode, sophisticated but accessible systems and tools have been developed – such as Apache Hadoop DFS (distributed file system).
- A data lake can also be an example of data storage.

Data storage Layer

- **Direct-attached storage (DAS)**
- Direct-attached storage stands for all types of physical data storage devices you can connect to a computer.
- Portable and affordable — yet only accessible by one computer at a time — DAS is a standard solution for keeping small-scale records data backups or for transferring data between devices.
- Popular types of direct-attached storage include external hard drives or solid-state drives (SSD), flash drives (USB sticks) and, although now dwindling in popularity, CD/DVD disks and other older methods.

Data storage Layer

- **Network-attached storage (NAS)**
- Network-attached storage (NAS) is a special hardware unit, featuring file-level architecture that can be accessed by more than one device as long as all users are connected to the internal network.
- In essence, a NAS unit features several storage disks or hard drives, processors, RAM, and lightweight operating systems (OS) for managing access requests.

Data storage Layer

- **Storage area networks (SAN)**
- Storage area networks (SANs) help assemble an even more complex on-premises data management architecture that features two components:
- A dedicated network for data exchanges with network switches for load balancing
- Data storage system, consisting of on-premises hardware.
- The purpose of SAN is to act as a separate “highway” for transmitting data between servers and storage devices across the organization, in a bypass of local area networks (LANs) and wide-area networks (WANs). Featuring a management layer, SANs can be configured to speed up and strengthen server-to-server, storage-to-server and storage-to-storage connections.

Data storage Layer

- **Software-defined storage (SDS)**
- A software-defined storage (SDS) system is a hardware-independent, software-based storage architecture that can be used on any computing hardware platform.
- The two major benefits of SDS include:
  - Scalability:
  - Total cost of ownership (TCO)

Data storage Layer

- **Cloud data storage**
- Unlike other options, cloud data storage assumes you will (primarily) use offsite storage of data in public, private, hybrid or multicloud environments that are managed and maintained by cloud services providers such as Amazon, Microsoft and Google, among others.
- Unlike NAS or SAN, public cloud data storage doesn’t require a separate internal network — all data is accessible via the internet. Also, there are virtually no limits on scalability since you are renting storage resources from a third party that effectively offers an endless supply of servers.

Data storage Layer

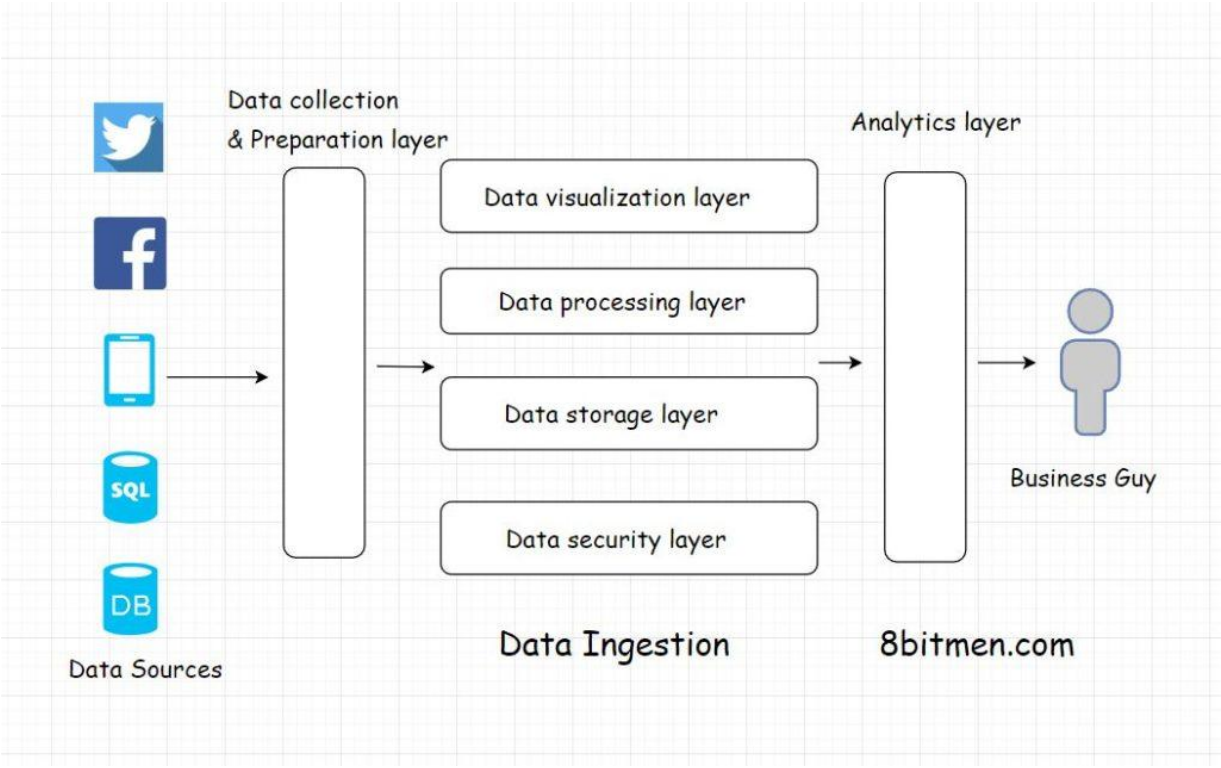
- **What is a data lake?**
- A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale.
- You can store your data as-is, without having to first structure the data, and run different types of analytics—from dashboards and visualizations to big data processing, real-time analytics, and machine learning to guide better decisions.

Data storage Layer

Characteristics	Data Warehouse	Data Lake
Data	Relational from transactional systems, operational databases, and line of business applications	Non-relational and relational from IoT devices, web sites, mobile apps, social media, and corporate applications
Schema	Designed prior to the DW implementation (schema-on-write)	Written at the time of analysis (schema-on-read)
Price/Performance	Fastest query results using higher cost storage	Query results getting faster using low-cost storage
Data Quality	Highly curated data that serves as the central version of the truth	Any data that may or may not be curated (ie. raw data)
Users	Business analysts	Data scientists, Data developers, and Business analysts (using curated data)
Analytics	Batch reporting, BI and visualizations	Machine Learning, Predictive analytics, data discovery and profiling

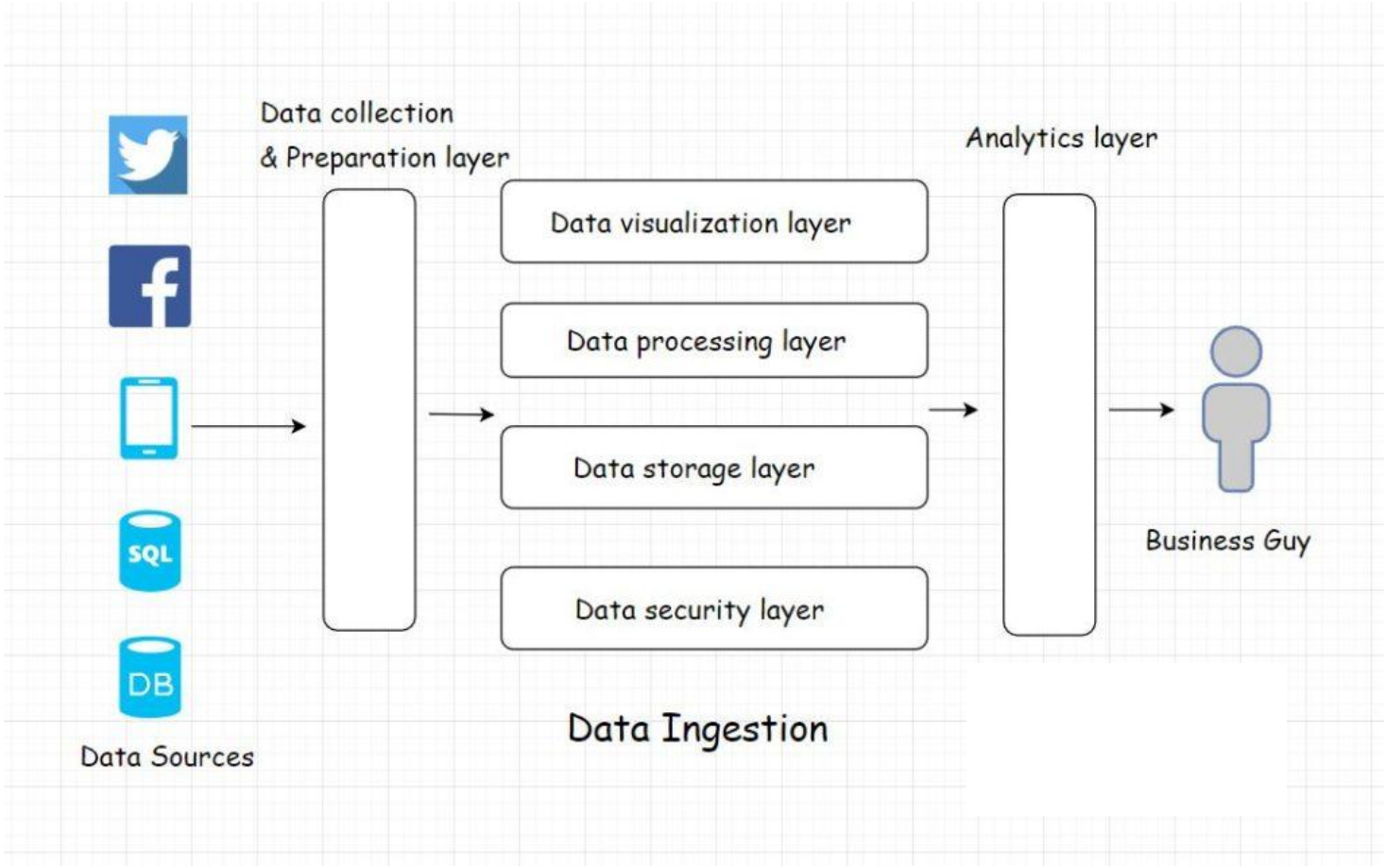
Data Ingestion Layer

- Data ingestion is the initial & the toughest part of the entire data processing architecture.
- The key parameters which are to be considered when designing a data ingestion solution are:
- Data Velocity, size & format: Data streams in through several different sources into the system at different speeds & size. Data streams from social networks, IoT devices, machines & what not. And every stream of data streaming in has different semantics. A stream might be structured, unstructured or semi-structured.



Data Ingestion Based Architecture

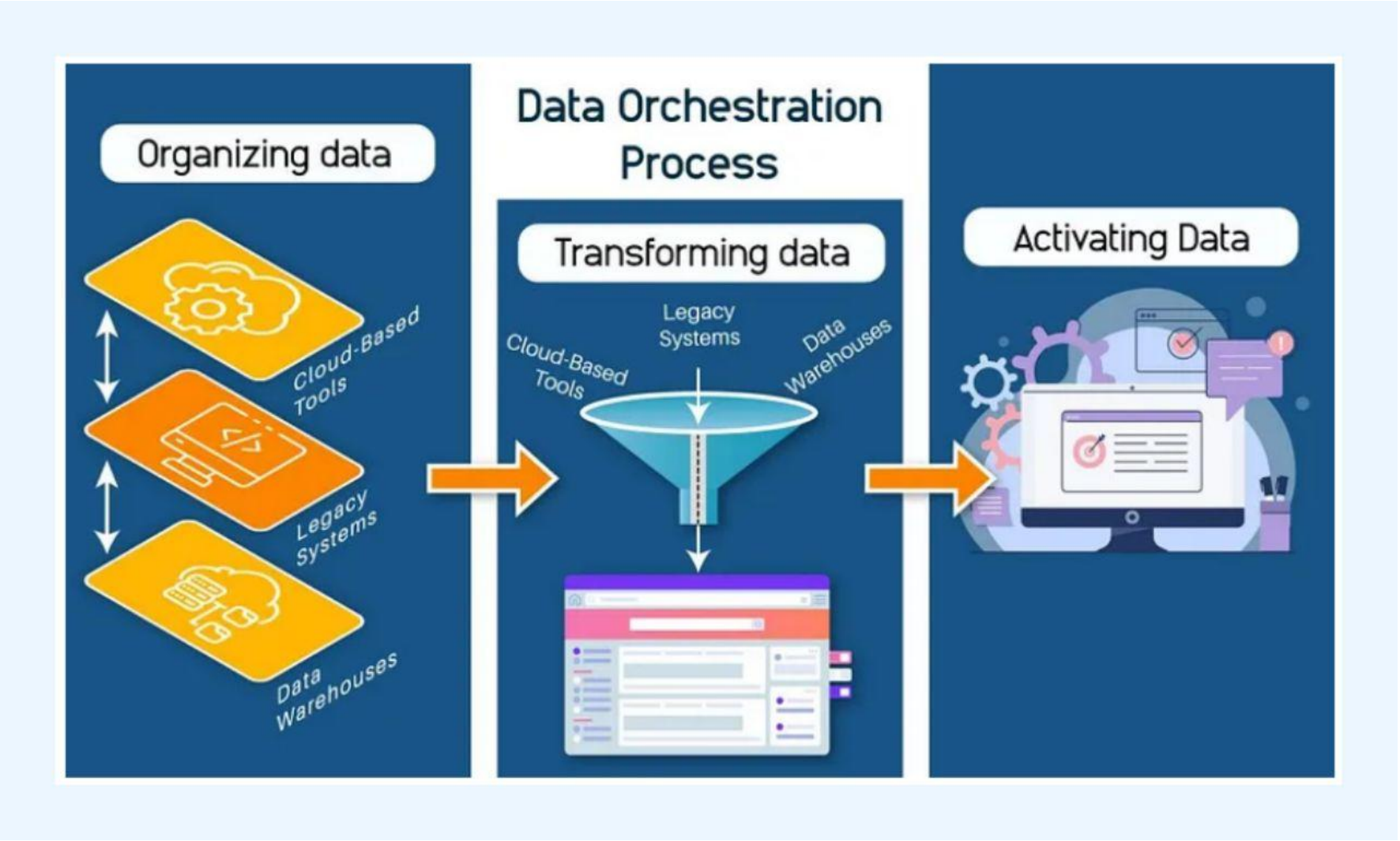
- The frequency of data streaming: Data can be streamed in continually in real-time or at regular batches. We would need weather data to stream in continually. On the other hand, to study trends social media data can be streamed in at regular intervals.



Orchestration

- Most big data solutions consist of repeated data processing operations, encapsulated in workflows, that transform source data, move data between multiple sources and sinks, load the processed data into an analytical data store, or push the results straight to a report or dashboard.
- To automate these workflows, you can use an orchestration technology such Apache Oozie and Sqoop.

Orchestration



Big Data Architecture – Layered Components

Data Source Layer

Databases, CRM systems, IoT sensors, web logs, clickstreams, third-party APIs

Data Ingestion Layer

Kafka, Flume, Sqoop, NiFi, Event Hubs - moving data reliably into the system

Data Storage Layer

HDFS, Cassandra, HBase, Cloud Data Lakes (S3, Azure Blob, GCS)

Processing Layer

Batch (MapReduce, Spark) and Streaming (Flink, Spark Streaming, Storm)

Big Data Architecture – Layered Components (cont.)

Orchestration Layer

Apache Oozie, Airflow, Control-M - managing complex workflows and dependencies

Analytics Layer

SQL engines (Hive, Presto, SparkSQL) and ML libraries (MLlib, TensorFlow)

Visualization Layer

Tableau, Power BI, Kibana, Superset - translating insights into actionable views

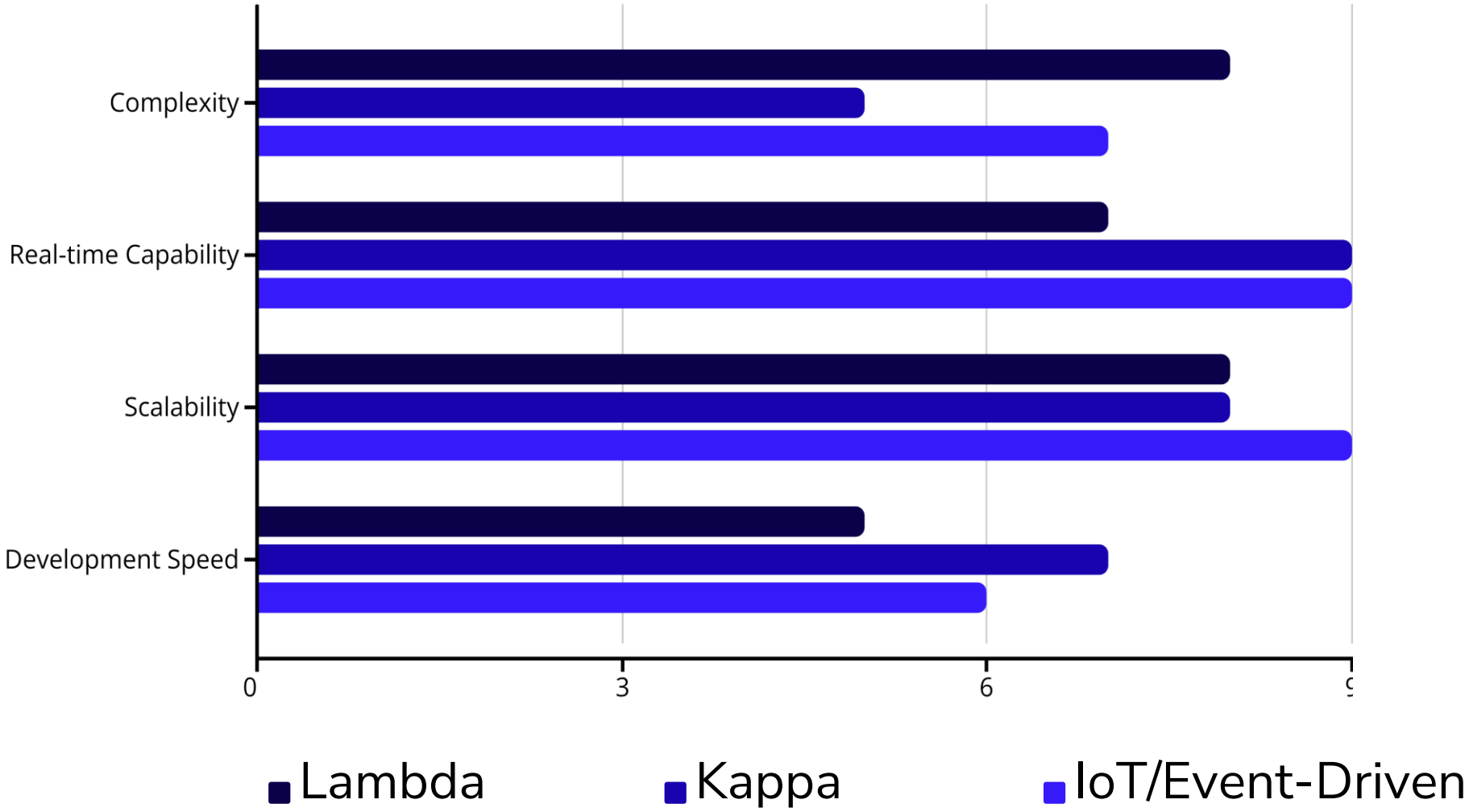
Security & Governance Layer

Ranger, Knox, Atlas - ensuring compliance and protecting sensitive data

Layer Implementation Examples

Company	Layer	Technologies	Scale/Impact
Netflix	Ingestion	Kafka for streaming events	500B+ events daily
Amazon	Storage	S3 for video logs	Exabytes of data
Uber	Processing	Spark Streaming	Near real-time matching
Airbnb	Orchestration	Airflow for ETL	1000s of daily workflows
Facebook	Analytics	Presto for SQL queries	Petabyte-scale analytics
Spotify	Visualization	Custom dashboards	Music streaming patterns

Comparative Analysis: Choosing the Right Architecture



Scale: 1-10, with 10 being highest/best

Architecture Feature Comparison

Feature	Lambda	Kappa	IoT/Event-Driven
Complexity	High (dual logic)	Lower (single pipeline)	Medium (device mgmt)
Batch Support	Native	Via stream replay	Sometimes
Streaming Support	Yes	Native	Yes
Recomputation	Batch layer	Replay stream	Replay events
Primary Use Cases	Social media, financial	Fraud detection, logs	Smart devices, industrial

Architecture Selection Decision Tree

Do you need both historical and real-time processing?

Yes → Consider Lambda first

No, just real-time → Consider Kappa

Is your data coming from distributed IoT devices?

Yes → IoT/Event-Driven architecture likely best

No → Continue with Lambda or Kappa evaluation

Do you have development resources for dual pipelines?

Yes → Lambda provides accuracy and performance advantages

No → Kappa offers simplicity with acceptable trade-offs

Are your batch processing windows very long (hours/days)?

Yes → Lambda separates these concerns efficiently

No → Kappa may provide sufficient performance

The Future: Serverless Big Data

Moving toward managed services and unified processing models

Emerging Trends: Serverless Big Data

Cloud-Native Solutions

AWS Lambda + Kinesis

Event-driven, serverless data pipelines with automatic scaling

Google Dataflow

Unified batch and streaming programming model with autoscaling

Azure Functions + Event Hubs

Serverless compute triggered by streaming data events

Unified Platforms

Databricks Delta Lake

ACID transactions on data lakes with unified batch/stream processing

Apache Beam

Single programming model for batch and streaming with multiple runners

Snowflake

Cloud data platform unifying storage, compute, and analytics

Data Stream Processing Fundamentals

Dr. Sonali Agarwal
Associate Professor, Department of IT
IIIT Allahabad

Introduction

What is Scalable Data Analytics ?

Scalable data analytics refers to the ability to handle large datasets efficiently and effectively.

Scalable Data is continuously growing…..

It contains massive volumes of structured and unstructured data.

Decision-makers can gain deeper and more comprehensive insights into market trends, customer behavior, economic indicators

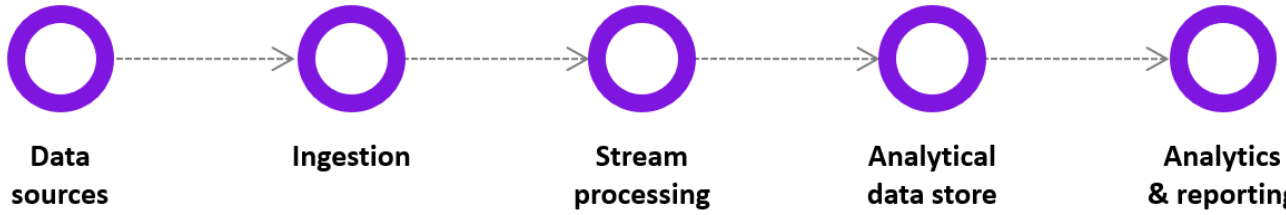
Scalable Data Analytics help to gain better outcomes

A global study of 682 executives in financial services and insurance companies *Aite Group Study commissioned by TransUnion: Current State Assessment: Global Analytics Ecosystem (October 2019)*

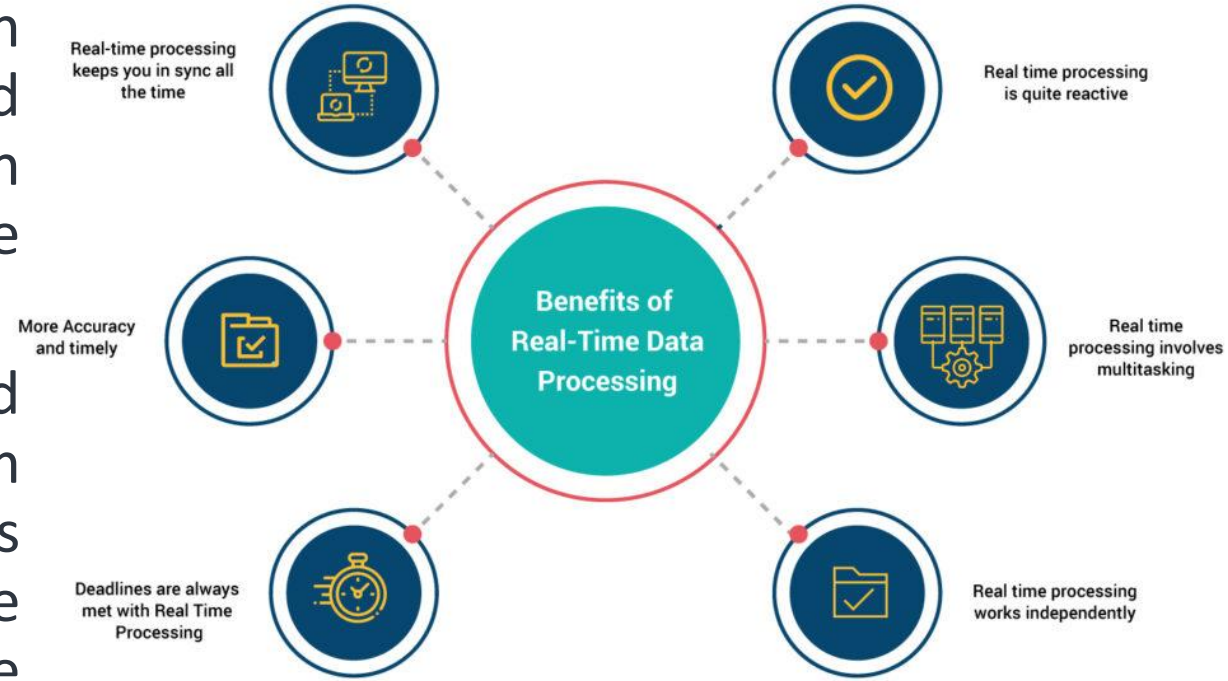
[1] Source: https://newsroom.transunion.com/drowning-in-data-financial-services-and-insurance-industries-seek-technology-and-talent-to-close-global-insights-gap/

Real-Time Data Processing

- **Real-time data processing refers to the ability to collect, process, and analyze data as it is generated.**

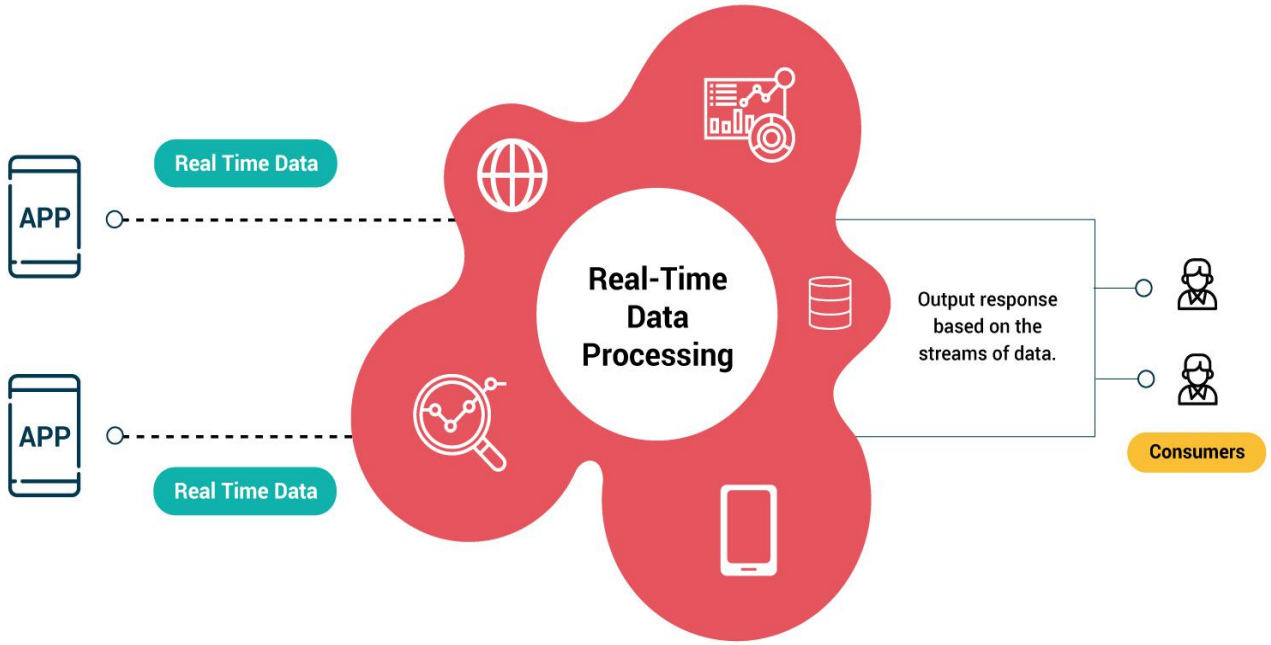


- Typically, real-time processing is used in situations where data must be processed and made immediately accessible, such as in financial markets, eCommerce, and online gaming.
- Real-time processing should not be confused with stream processing. While stream processing refers to the continuous processing of data streams as they are generated, real-time processing refers to the timely processing of input data.



Types of Real Time Data Processing

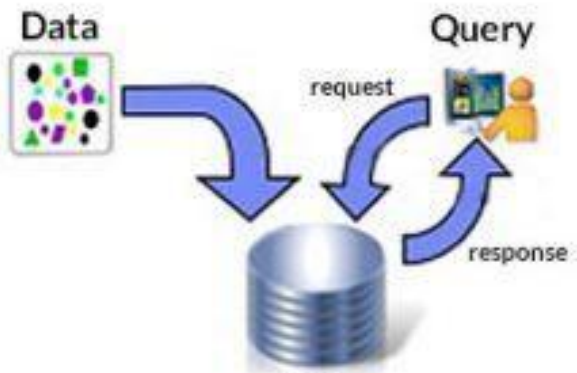
- **Hard real-time:**
  - Hard real-time systems must always meet deadlines.
  - These systems are frequently utilized in mission-critical applications, such as aviation or healthcare, where a missed deadline could have catastrophic effects.
  - Missed deadlines are seen as system failures.



- **Soft real-time:**
  - Soft real-time systems try to fulfill deadlines most of the time but may miss one without any real consequence.
  - These methods are widely used in video streaming or online gaming when missing a deadline is not important but should be avoided.
- **Firm real-time processing:**
  - Firm real-time systems prioritize completing deadlines over other metrics.
  - These systems are utilized when fulfilling deadlines is more important than data accuracy or system performance. Data delivered after the deadline may be considered invalid but isn't a system failure.

Stream Processing

Traditional Processing



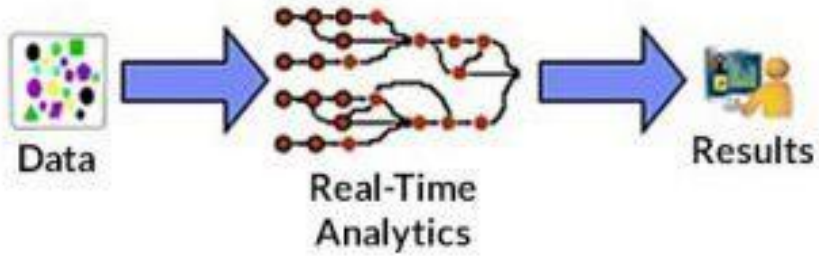
Historical fact finding

Find and analyze information stored on disk

Batch paradigm, pull model

Query-driven: submits queries to static data

Stream Processing



Current fact finding

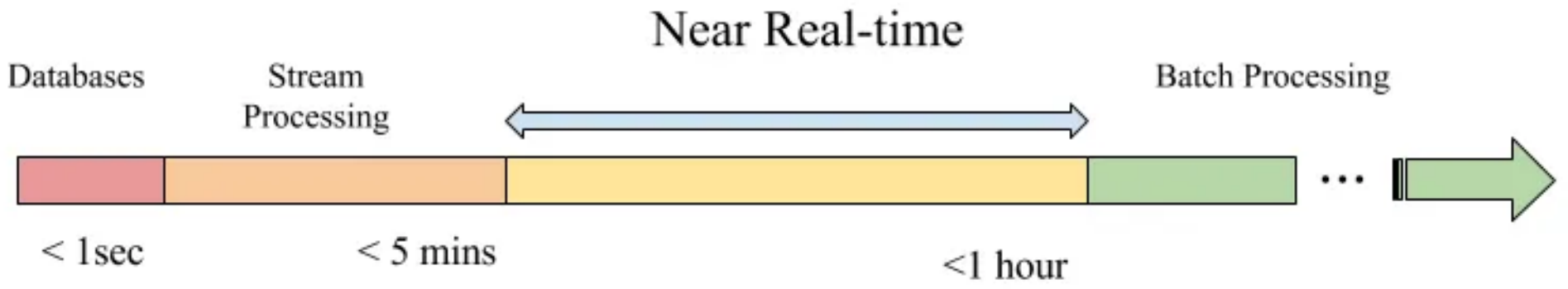
Analyze data in motion – before it is stored

Low latency paradigm, push model

Data driven: bring data to the analytics

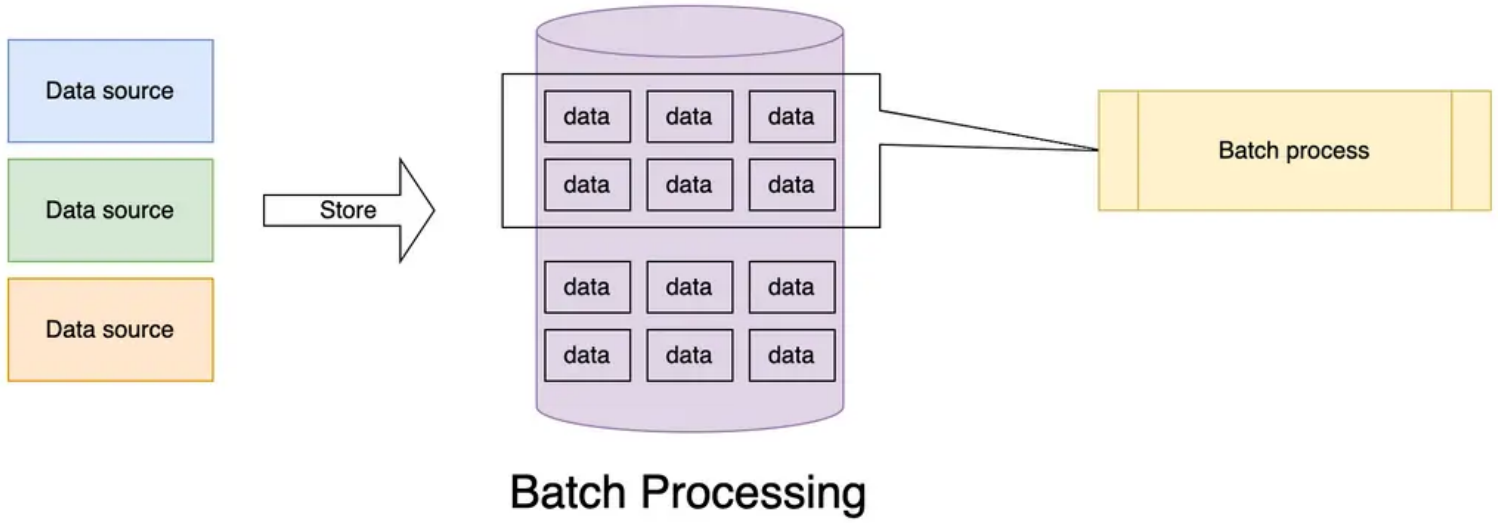
Near Real Time Systems

- Near-real-time processing, also known as “low-latency” processing, involves processing data with a few seconds to a few minutes of delay.
- Near-real-time processing is employed when a business has to handle data quickly, but a delay isn’t critical. Applications where near real-time processing may be suitable include handling online transactions and managing inventory levels.
- Some benefits of near-real-time include: Scalable, Flexible, Cost-effective:

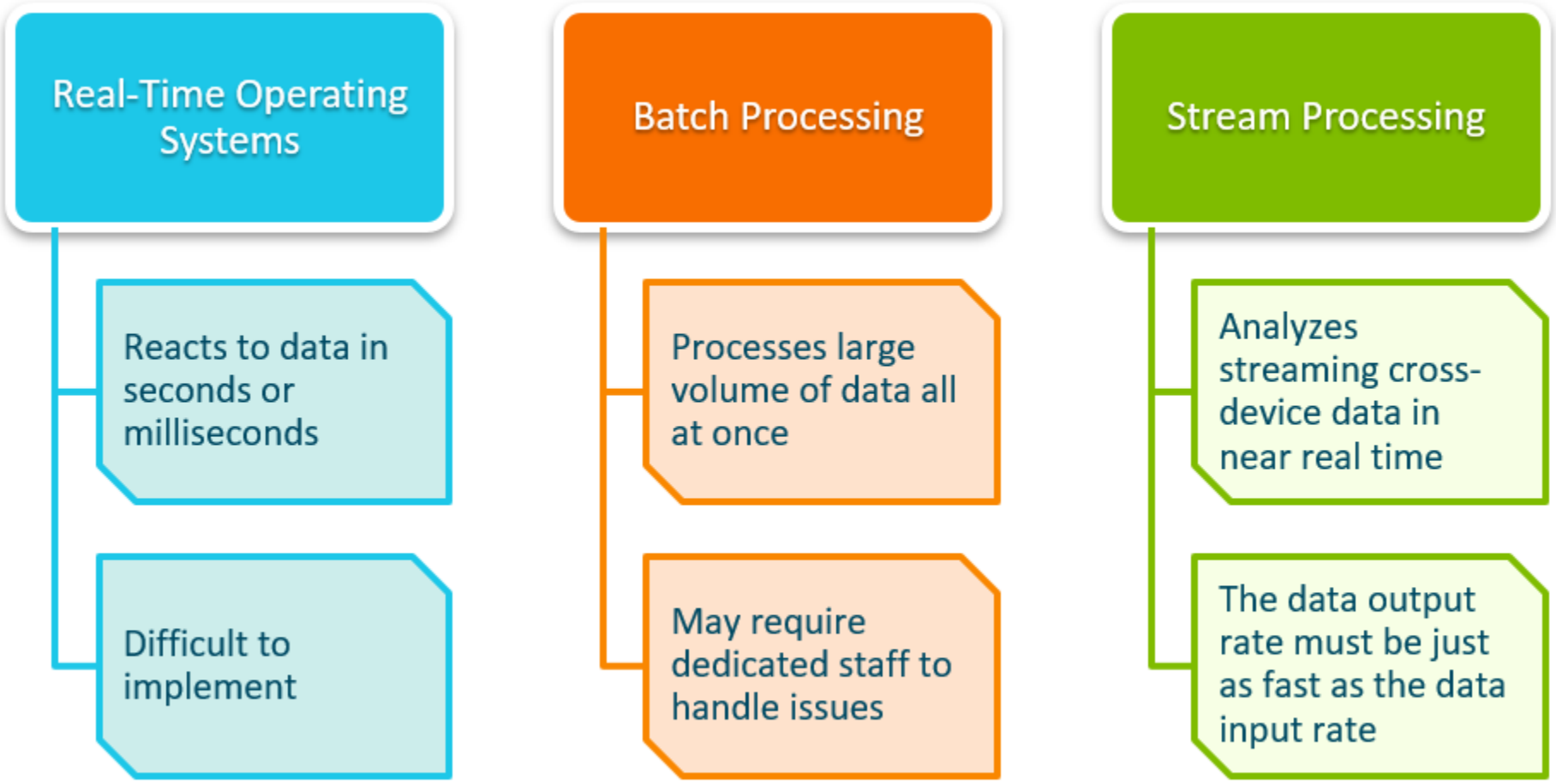


Batch Processing

- Batch processing involves processing data in large groups, or “batches,” rather than individually.



Comparison between Data Processing Techniques



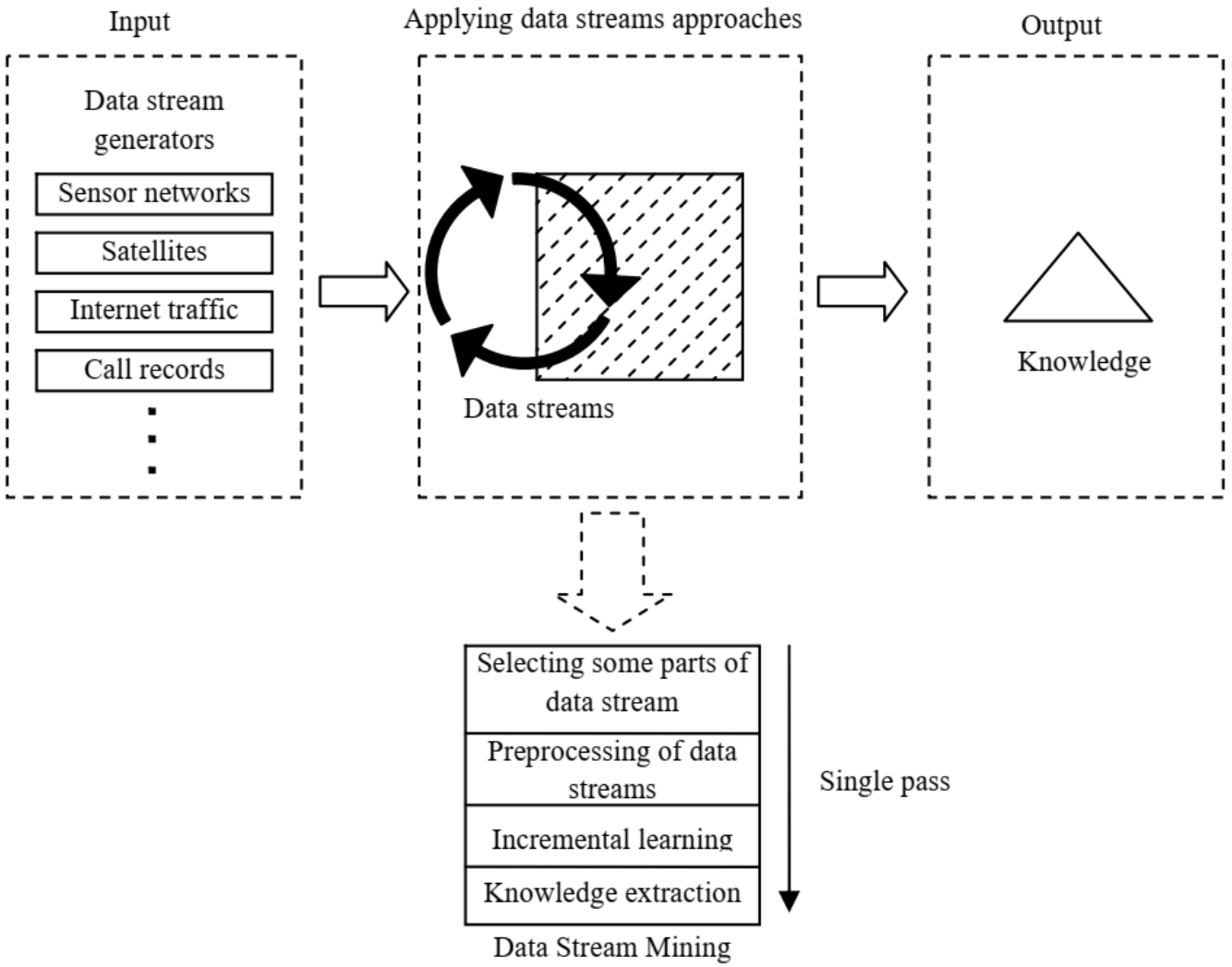
Benefits & drawbacks of common data processing types

Comparison between Data Processing Techniques

Factor	Real-Time Processing	Near Real-Time Processing	Batch Processing
Latency	Low (milliseconds)	Moderate (seconds to minutes)	High (hours to days)
Cost	High	Moderate	Low
Complexity	High	Moderate	Low
Use Case	Time-sensitive tasks (e.g., fraud detection, stock trading)	Activities with high delay tolerance (e.g., social media analysis, customer service)	Non-time-sensitive tasks (e.g., financial reporting, sales analysis)

Data Streams Mining

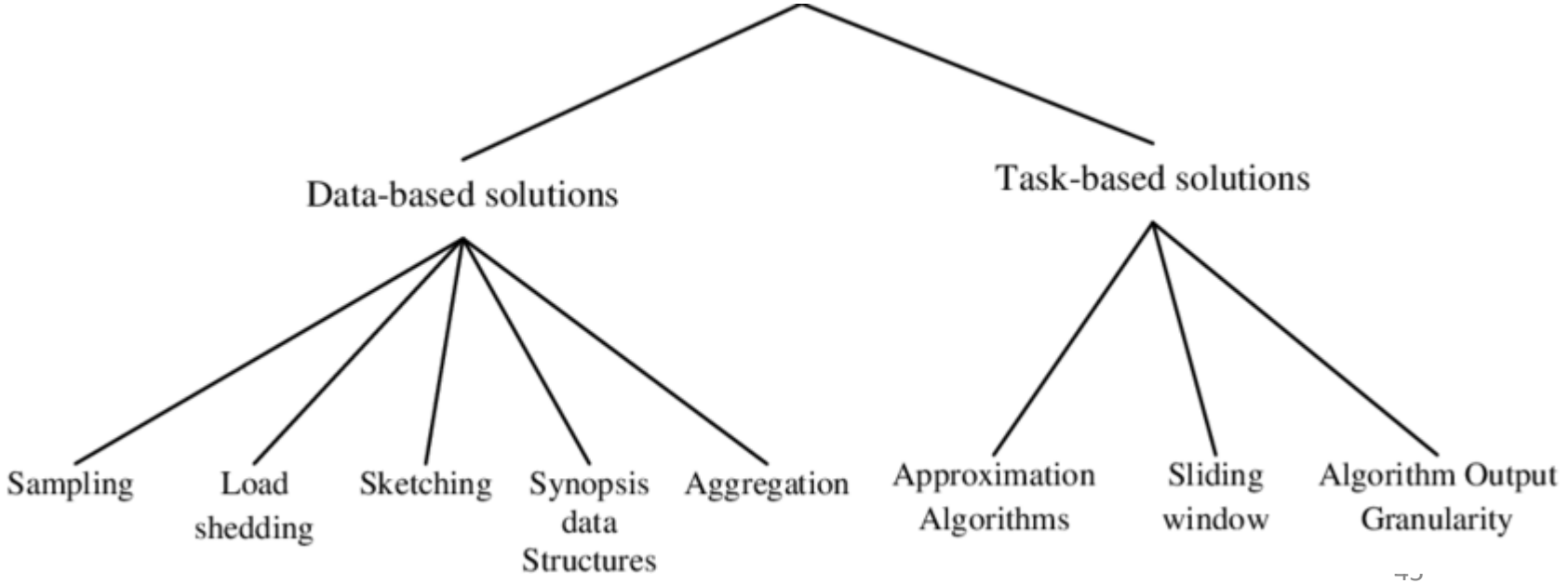
Stream Data Mining



Sampling

Sampling: Probabilistic choice of data items for processing.

- Random Sampling: Selecting data points randomly with equal probability.
- Reservoir Sampling: Maintaining a fixed-size sample with equal probability inclusion.
- Windowed Sampling: Sampling from fixed-size windows.
- Frequency-based Sampling: Sampling based on data point occurrence frequency.



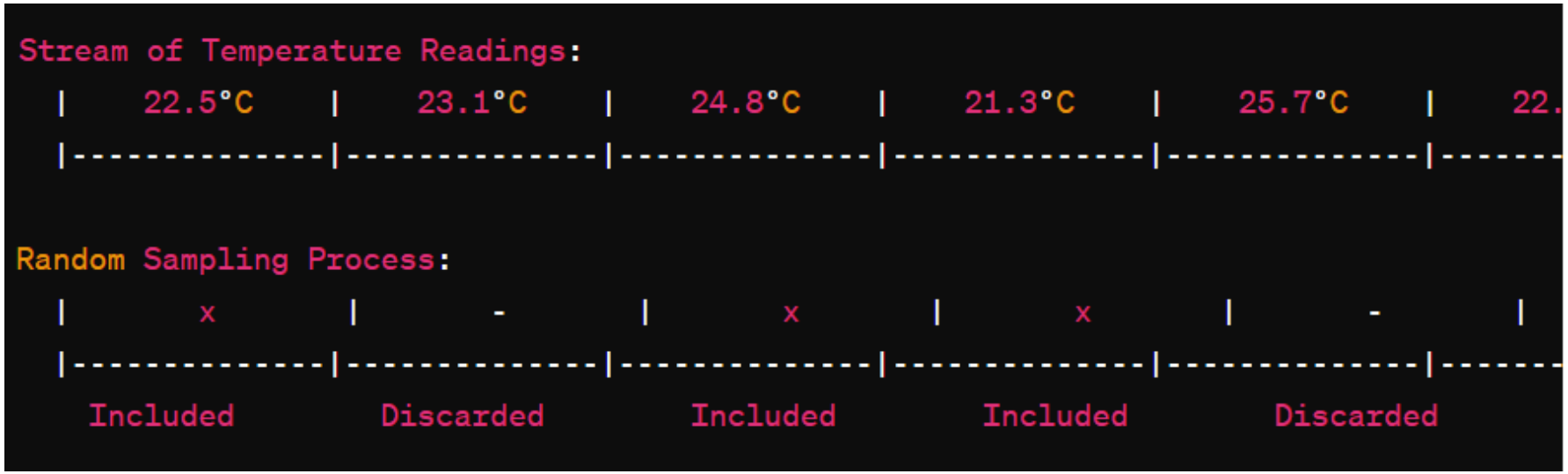
Data Streams Sampling

Data Stream Sampling

- Data stream sampling is a technique used in data analysis to efficiently extract a representative subset of data from a continuous stream of data points.
- In scenarios where processing or storing the entire stream of data is impractical due to its volume or speed, data stream sampling offers a way to approximate characteristics of the data using a smaller, manageable subset.

Sampling technique types

- Random Sampling: Selecting data points randomly from the stream with equal probability.
- Sampling rate: 20%



Sampling technique types

Reservoir Sampling: Maintaining a fixed-size sample (reservoir) and selecting data points with specific probabilities, ensuring each point has an equal chance of inclusion.

Reservoir Sampling Algorithm:

- Let's consider a stream of numbers: 2, 7, 1, 8, 4, 6, 3, 5, 9.
- We want to sample k = 3 numbers from this stream using reservoir sampling.
- Initialize reservoir = [2, 7, 1] with the first k elements.

For the 4th element (8):

- Generate a random integer between 1 and 4. Let's say we get j = 3.
- Since j <= k, replace the j-th element (1) in the reservoir with 8.
- Updated reservoir: [2, 7, 8].

Sampling technique types

2, 7, 1, 8, 4, 6, 3, 5, 9.

[2, 7, 8].

Reservoir Sampling

For the 5th element (4):

- Generate a random integer between 1 and 5. Let's say we get j = 5.
- Since j > k, we don't update the reservoir.

For the 6th element (6):

- Generate a random integer between 1 and 6. Let's say we get j = 2.
- Since j <= k, replace the j-th element (7) in the reservoir with 6. Updated reservoir: [2, 6, 8].
- Continue this process until all elements are processed.

Sampling technique types

Windowed Sampling:

Dividing the stream into fixed-size windows and selecting a sample from each window.

Sampling technique types

Fixed-size Window Sampling:

Description: In fixed-size window sampling, the stream is divided into windows of a pre-defined fixed size.

Characteristics:

- Each window contains the same number of data points.
- Once a window is filled, it moves forward by one data point, and the oldest data point is removed.

Applications:

- Used for time-series analysis, where each window represents a fixed period of time.
- Useful for detecting trends or patterns over time.

Sampling technique types

Fixed size windows

Example Stream of Temperature Readings:

```
Stream: 22.5°C, 23.1°C, 24.8°C, 21.3°C, 25.7°C, 22.9°C, 24.5°C, 23.6°C, 25.1°C, 21.9
```

```
Window 1: [22.5°C, 23.1°C, 24.8°C, 21.3°C, 25.7°C]
```

```
Window 2: [22.9°C, 24.5°C, 23.6°C, 25.1°C, 21.9°C]
```

Sampling technique types

Sliding Window Sampling:

Description: Sliding window sampling maintains a window of a fixed size that slides along the stream.

Characteristics:

- Overlapping windows allow for continuous analysis of the stream.
- Each new data point adds to the window, and the oldest data point is removed as the window slides.

Applications:

- Useful for real-time analysis where continuous monitoring of the stream is required.
- Used in anomaly detection, where anomalies are detected based on changes within the sliding window.

Sampling technique types

- Stream 22.5 C 23.1 C 24.8 C 21.3 C 25.7 C 22.9 C 24.5 C 23.6 C 25.1 C 21.9 C
- Window 1 22.5 C 23.1 C 24.8 C 21.3 C 25.7 C
- Window 2 23.1 C 24.8 C 21.3 C 25.7 C 22.9 C
- Window 3 24.8 C 21.3 C 25.7 C 22.9 C 24.5 C

Sampling technique types

Tumbling Window Sampling:

Description: Tumbling window sampling divides the stream into non-overlapping, fixed-size windows.

Characteristics:

- Each window is independent and does not overlap with other windows.
- Once a window is filled, it is processed independently, and then the next window starts.

Sampling technique types

- **Tumbling Window Sampling**
- **Input stream is**
- **Stream 22.5 C 23.1 C 24.8 C 21.3 C 25.7 C 22.9 C 24.5 C 23.6 C 25.1 C 21.9 C**

- Window 1 22.5 C 23.1 C 24.8 C 21.3 C 25.7 C
- Window 2 22.9 C 24.5 C 23.6 C 25.1 C 21.9 C

Sampling Applications

Applications:

- Used for batch processing of data streams, where each window represents a separate batch.
- Useful for aggregating statistics or performing calculations on non-overlapping segments of the stream.

Sampling technique types

- **Frequency-based Sampling**: Sampling data points based on their frequency of occurrence in the stream.
- **Example Stream of Events:**
- Stream
- Page A Page B Page C Page A Page B Page A Page C Page A Page A Page B

Sampling technique types

- Stream
- Page A Page B Page C Page A Page B Page A Page C Page A Page A Page B

Count Page Frequencies:

Count the frequency of each page in the stream:

- Page A: 5 occurrences
- Page B: 3 occurrences
- Page C: 2 occurrences

Calculate Sampling Probabilities

Sampling technique types

Calculate Sampling Probabilities:

- Calculate the sampling probability for each page based on its frequency:
  - Page A: 5/10=0.5
  - Page B: 3/10=0.3
  - Page C: 2/10=0.2

Sampling technique types

Perform Sampling:

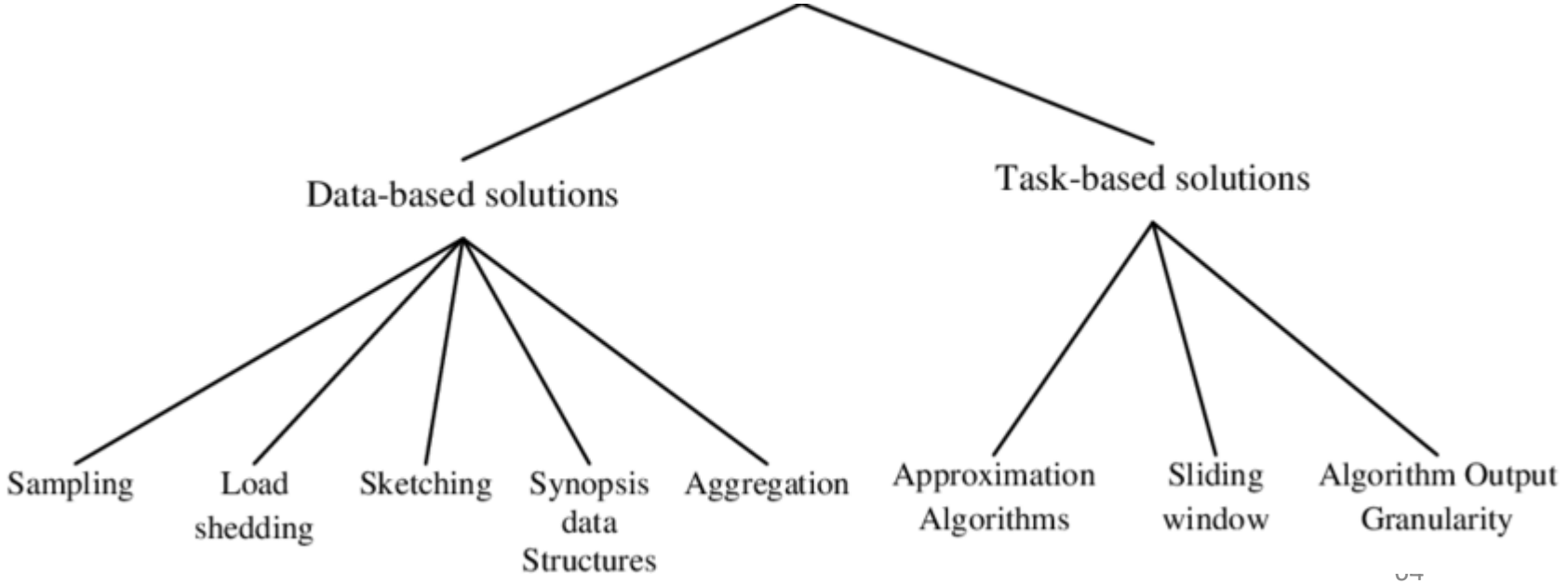
- Randomly select pages based on their sampling probabilities:
- Suppose we randomly select 3 samples from the stream.
- Sample 1: Page A (probability 0.5)
- Sample 2: Page B (probability 0.3)
- Sample 3: Page A (probability 0.5)

- Sample: Page A, Page B, Page A

Data Streams: Load Shedding

Stream Data processing types

- Load Shedding: Dropping sequences of data streams.
- Challenges: Similar to sampling, unknown dataset size, difficult for mining algorithms.



Load Shedding

Load Shedding Technique:

Threshold-based Load Shedding

- **Description:** Threshold-based load shedding involves setting a threshold for incoming data volume. If the incoming data volume exceeds this threshold, some data is selectively dropped to prevent overload.
- **Scenario:** In our example, let's assume that the system can handle up to 1000 data points per second without overloading. However, due to sudden spikes in data volume, the incoming data rate occasionally exceeds this limit.

Load Shedding

Implementation:

- **Monitoring Data Rate:** Continuously monitor the incoming data rate in the system.
- **Threshold Setting:** Set a threshold, such as 1000 data points per second, beyond which load shedding will be triggered.
- **Load Shedding Policy:** If the incoming data rate exceeds the threshold, apply a load shedding policy to selectively drop some data points.
- **Example Policy:** One simple load shedding policy could be to drop every nth data point beyond the threshold. For instance, if the threshold is exceeded by 10%, every 10th data point could be dropped to bring the rate back under the threshold.

Load Shedding

Example:

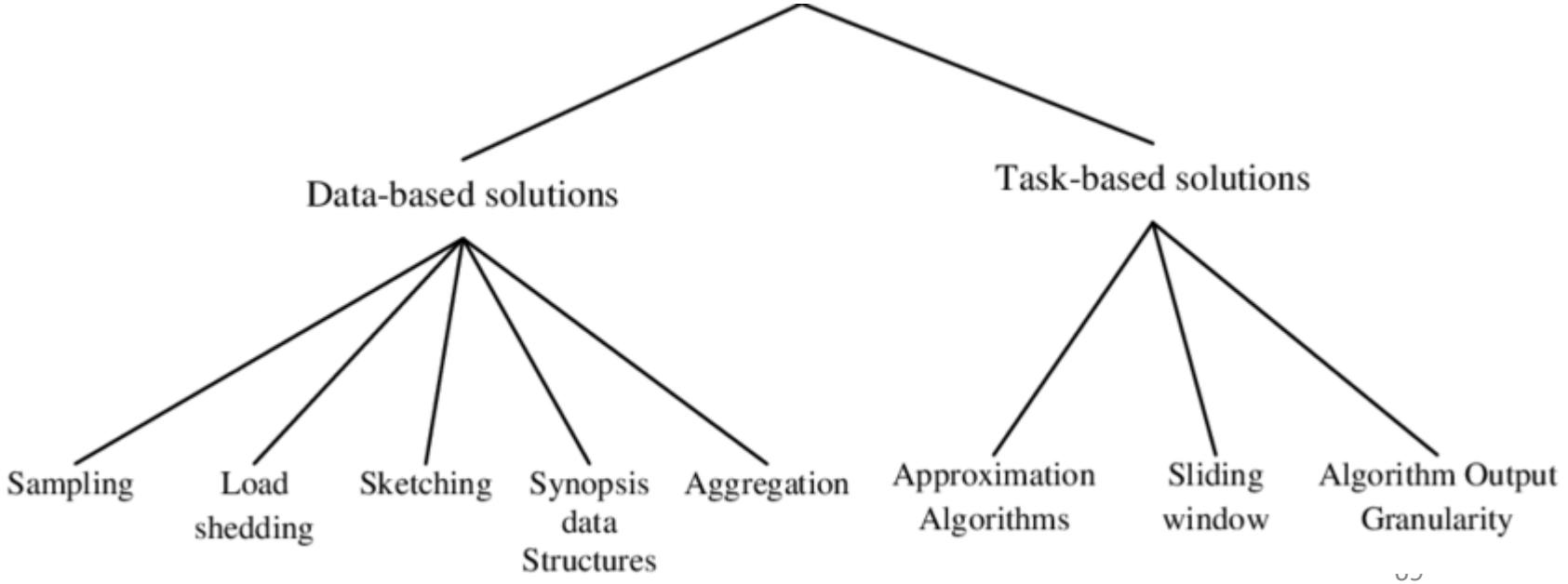
- **Incoming data rate:** 1200 data points per second.
- **Threshold:** 1000 data points per second.
- **Excess data rate:** 200 data points per second.
- **Load shedding policy:** Drop every 10th data point.
- **Result:** Out of the 200 excess data points, 20 (every 10th) are dropped, and the remaining 180 data points are processed and stored.

Data Streams Synopsis

Stream Data processing types

Data Synopsis: Summarization techniques like wavelet analysis, histograms, etc.

- Purpose: Summarize incoming stream for further analysis.
- Limitation: Approximate answers due to incomplete representation.



Data Synopsis types

- **Synopsis Trees**:
- **Type**: Hierarchical data structure.
- **Purpose**: Summarize data streams at different levels of granularity for efficient aggregation and querying.
- **Example**: In sensor networks, a hierarchical synopsis tree can summarize environmental data (e.g., temperature, humidity) collected from multiple sensors deployed in different locations.

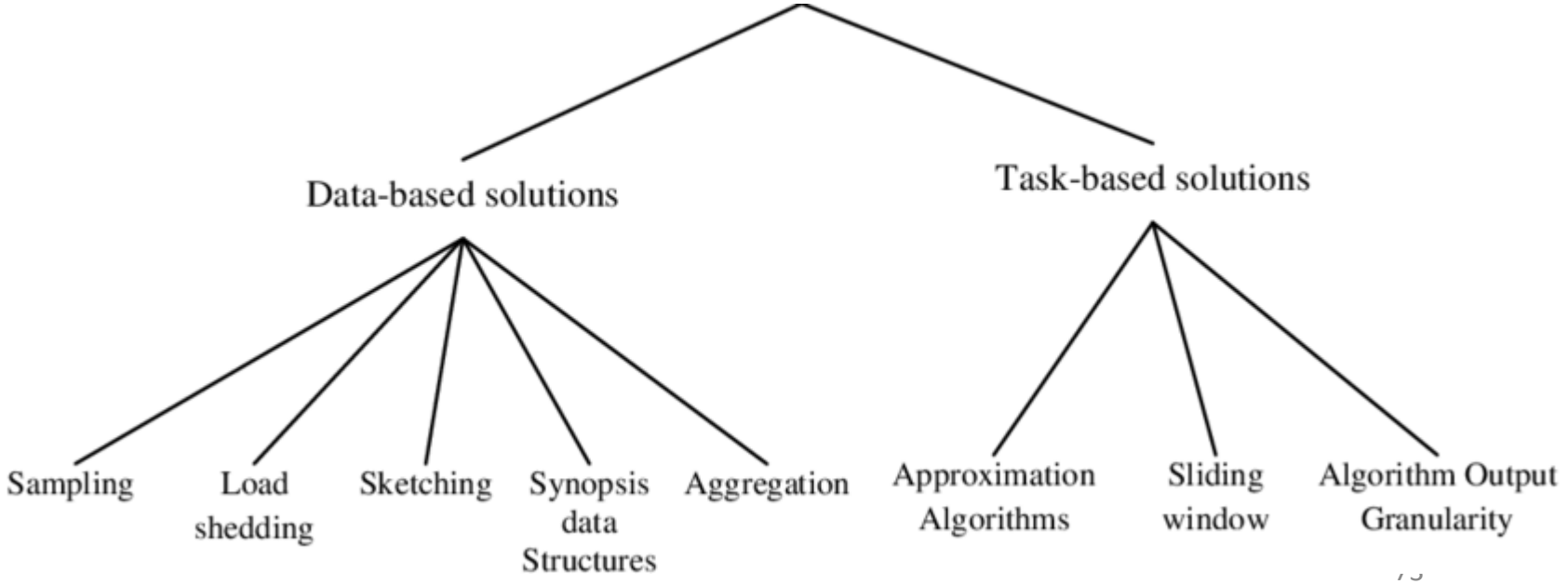
Data Synopsis types

- **Synopsis Matrices**:
- **Type**: Matrix-based data structure.
- **Purpose**: Summarize statistical information (e.g., frequency, correlation) of data streams in a compact format.
- **Example**: In social network analysis, a synopsis matrix can summarize the interactions between users (e.g., likes, comments) over time to identify influential users or detect communities.

Data Streams Sketching

Stream Data processing types

- Sketching: Randomly projecting a subset of features or vertically sampling.
- Applications: Comparing different data streams and aggregate queries.
- Drawback: Accuracy issues, challenging for data stream mining.



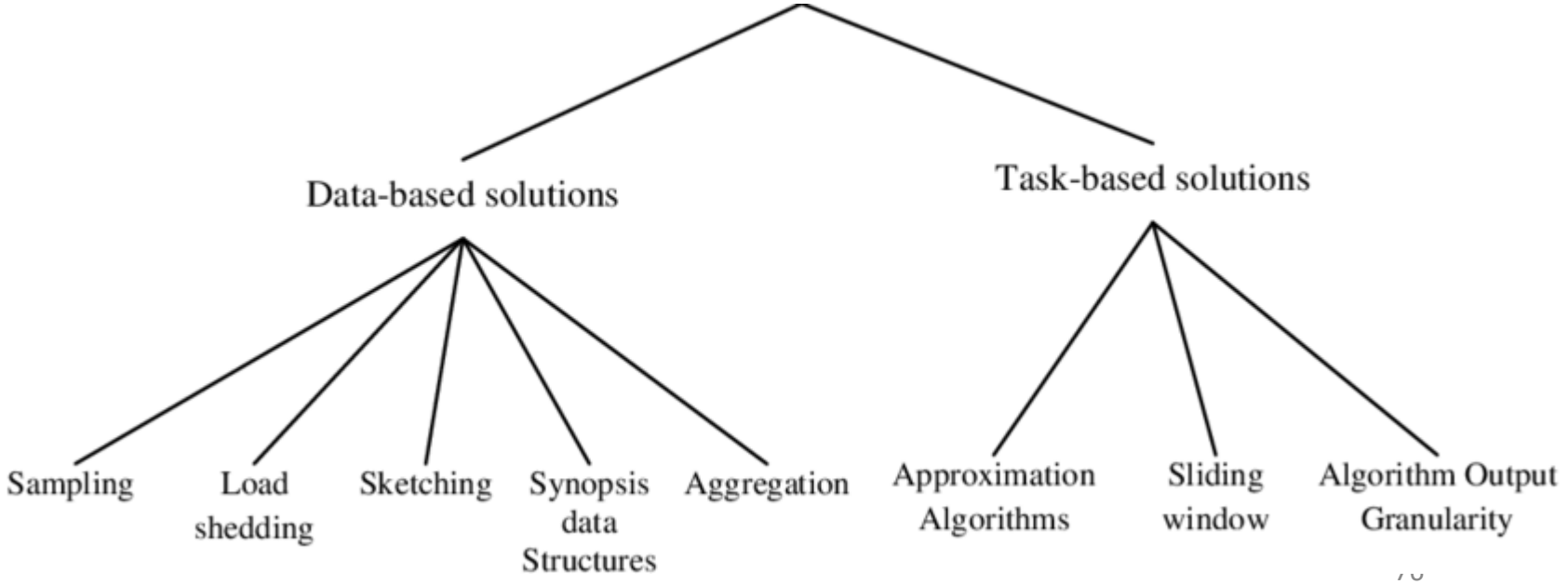
Sketching

- Data stream sketching refers to the process of summarizing a data stream using compact data structures called sketches, which provide approximate answers to queries about the data stream while conserving memory and processing resources.

Data Streams Aggregation

Stream Data processing types

- Aggregation: Representing input stream in a summarized form for data mining.
- Challenge: Efficiency reduction with fluctuating data distributions.



Aggregation

Temperature Readings:
[25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]

Sliding Window Parameters:

- Window Size: 5 (i.e., aggregate temperature over 5 readings)
- Window Step: 2 (i.e., move the window by 2 readings at a time)

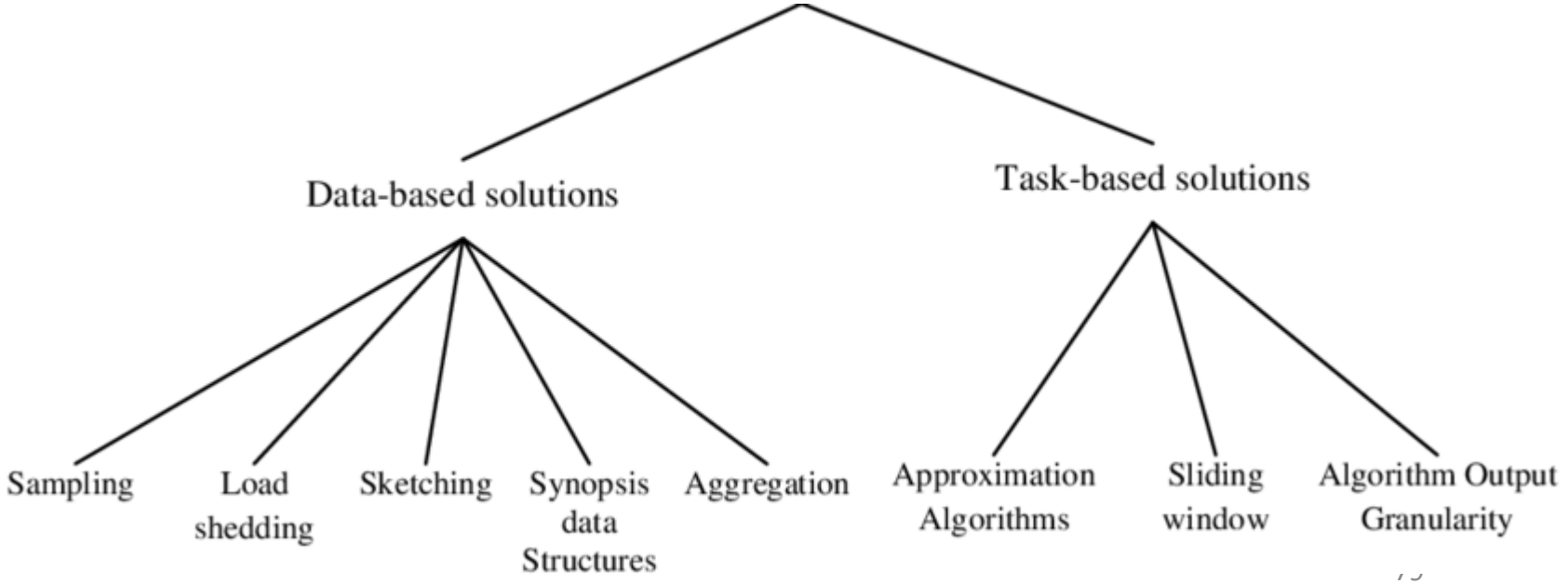
Aggregation

Temperature Readings:
[25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]

1. **Initial Window:**
   - Aggregate the first 5 temperature readings: `[25, 26, 27, 28, 29]`
   - Compute the average temperature: $\frac{25+26+27+28+29}{5} = 27.0$
   - Move the window by 2 readings.
2. **Second Window:**
   - Aggregate the next 5 temperature readings: `[27, 28, 29, 30, 31]`
   - Compute the average temperature: $\frac{27+28+29+30+31}{5} = 29.0$
   - Move the window by 2 readings.
3. **Third Window:**
   - Aggregate the next 5 temperature readings: `[29, 30, 31, 32, 33]`
   - Compute the average temperature: $\frac{29+30+31+32+33}{5} = 31.0$
   - Move the window by 2 readings.
4. **Fourth Window:**
   - Aggregate the next 5 temperature readings: `[31, 32, 33, 34, 35]`
   - Compute the average temperature: $\frac{31+32+33+34+35}{5} = 33.0$
   - Move the window by 2 readings.

Stream Data processing types

- Approximation Algorithms: Designing algorithms for computationally hard problems.
- Purpose: Provide approximate solutions with error bounds.
- Usage: Addressing data stream mining problems but may not solve resource constraints alone.



Statistical Estimation Algorithms

Statistical Estimation Algorithms:

- **Streaming Variance Algorithms:** Estimate variance and standard deviation of a data stream using incremental calculations to reduce memory requirements.
- **Streaming Histograms**: Approximate histograms of data stream distributions to analyze data distribution characteristics.

Stream Data processing types

Machine Learning Approximation Algorithms:

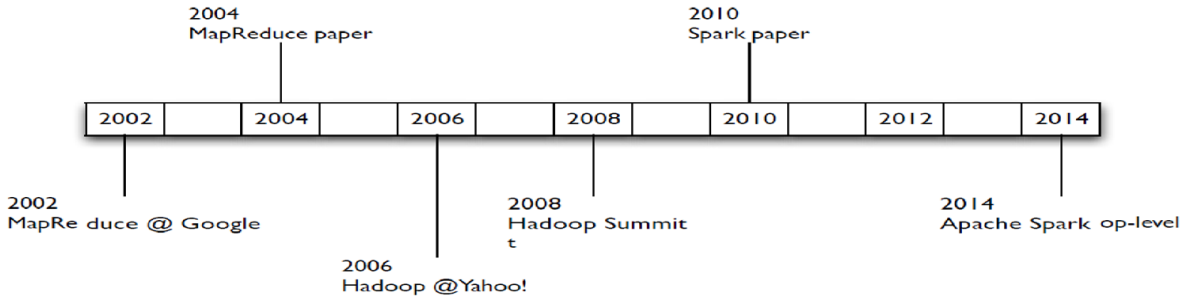
- **Streaming Clustering Algorithms:** Perform approximate clustering of streaming data to identify clusters and patterns efficiently.
- **Online Learning Algorithms:** Adapt traditional machine learning algorithms for streaming data, providing approximate models with continuous learning capabilities.

Introduction to Apache Spark

Basics of Apache Spark and Resilient Distributed Datasets (RDDs)

Introduction to Apache Spark

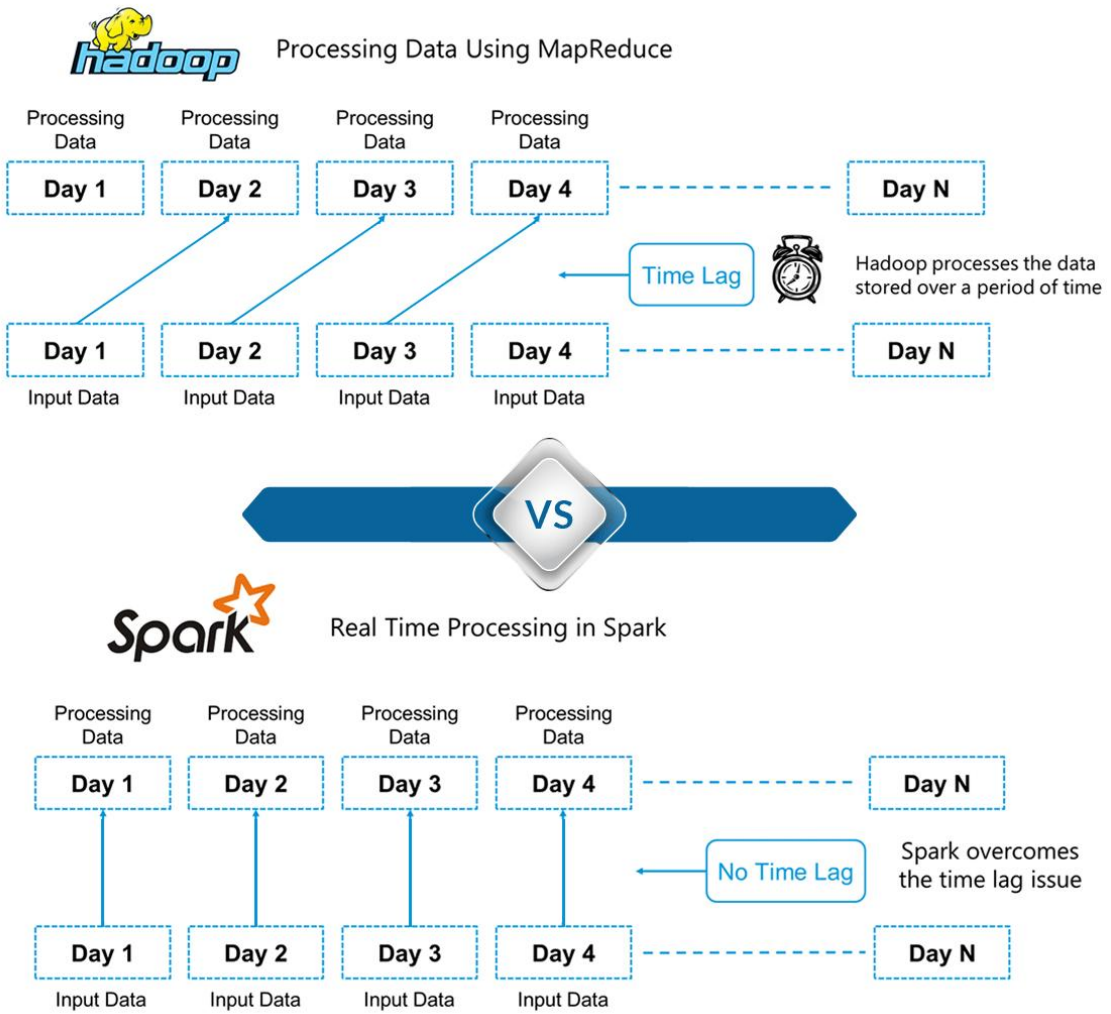
- Powerful **cluster computing** engine.
- Primarily based on Hadoop, offers several new computations such
  - Interactive queries
  - Stream processing
- The most Sparkling feature of Apache Spark is it offers **in-memory cluster computing.**
- In-memory cluster computing enhances the processing speed of an application.



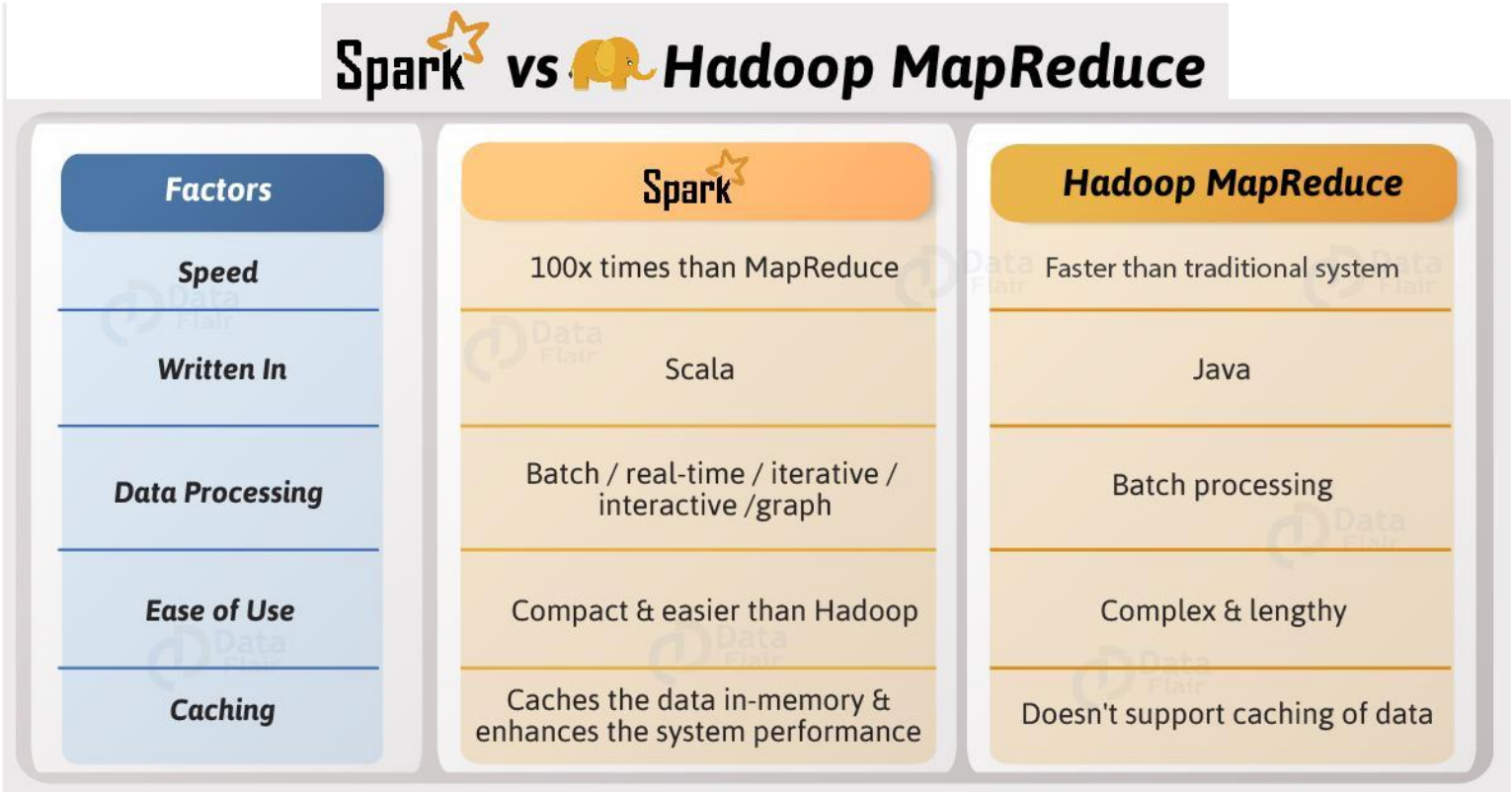
Introduction to Apache Spark

- Workloads in Apache Spark:
  - **Streaming and batch applications**
  - iterative algorithms
  - interactive queries
- Spark supports high-level APIs such a Java, Scala, Python and R. It is basically built upon Scala language.
- Framework for large scale stream processing
  - Scales to 100s of nodes
  - Can achieve second scale latencies
  - Integrates with Spark's batch and interactive processing
  - Provides a simple batch-like API for implementing complex algorithm
  - Can absorb live data streams from Kafka, Flume etc.

Why Spark is better than Hadoop

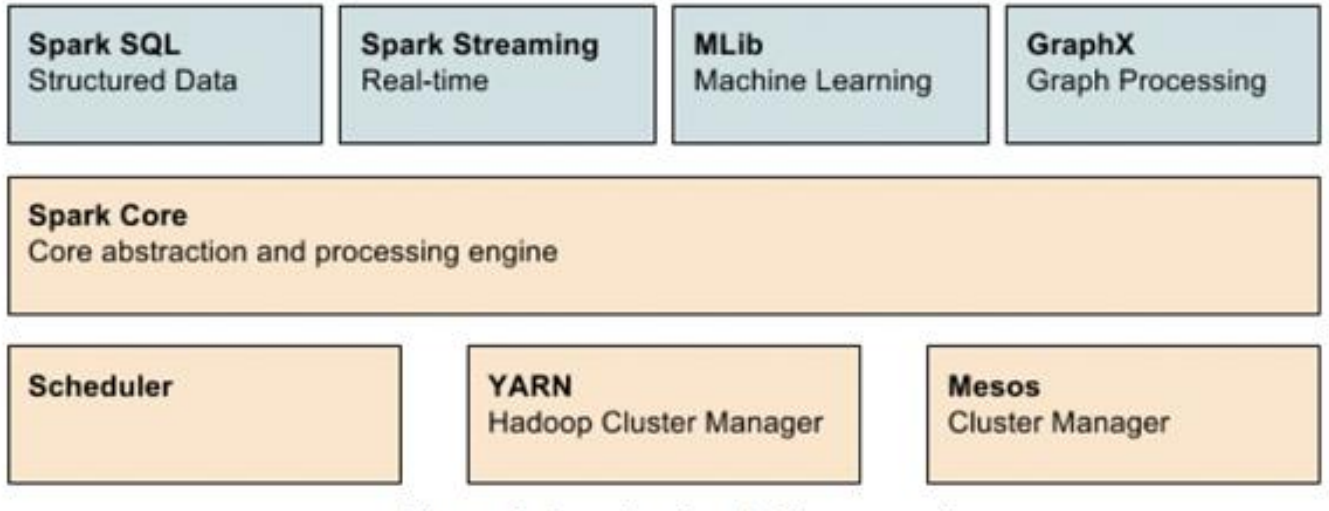


Why Spark is better than Hadoop



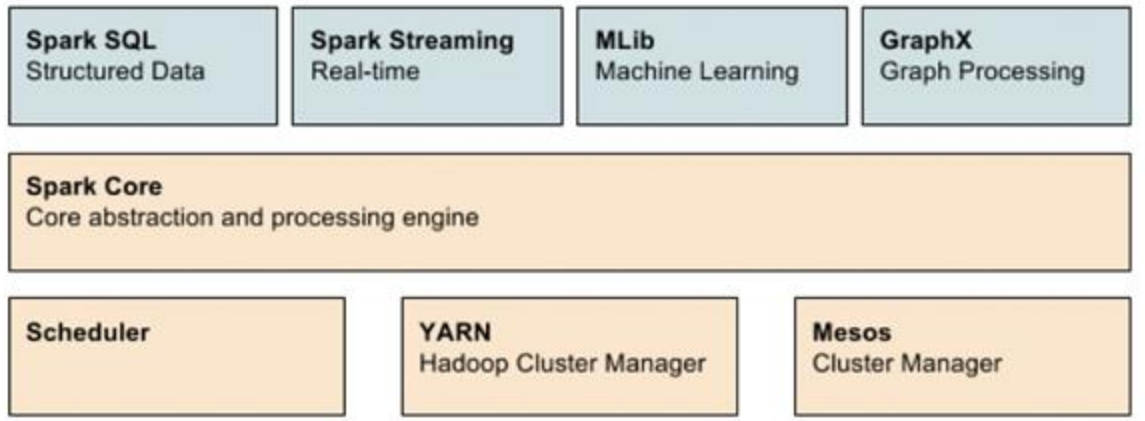
Apache Spark Component

- **Apache Spark Core**
  - It is accountable for all basic I/O Functionalities.
  - It monitors the role of the cluster. This also observes the jobs of the cluster.
  - Significant in programming with Spark and supports fault recovery
  - As we are using In-memory computation it enhances the productivity. It also overcomes the drawbacks of MapReduce.



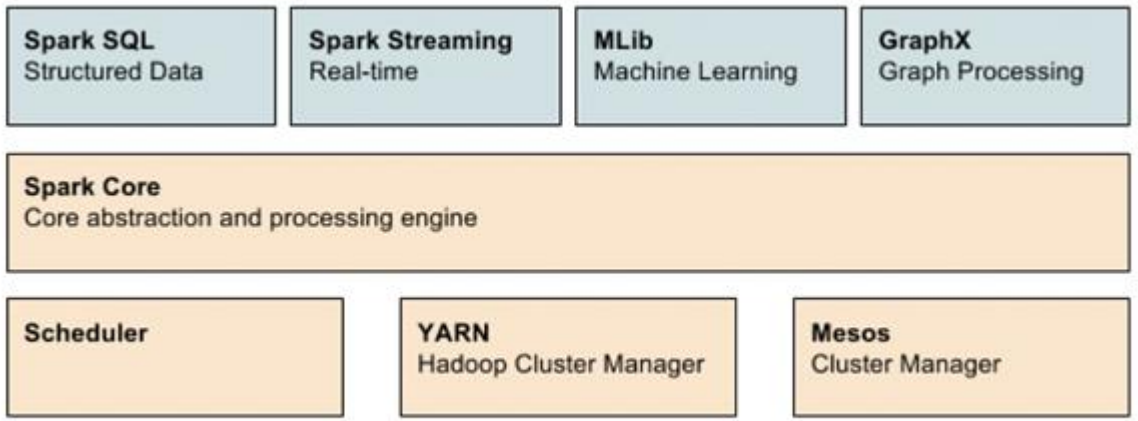
Apache Spark Component

- **Apache Spark SQL**
  - Spark SQL gives provision to perform structured as well as semi-structured data analysis.
  - In Spark SQL, we have extensible optimizer for the core functioning in SQL and fully compatible with HIVE, Avro, Parquet, ORC, JSON, and JDBC.
  - This also helps in querying and analyzing large datasets stored in Hadoop files.
  - Along with dataFrame, SQL provides a common way to access several data sources.



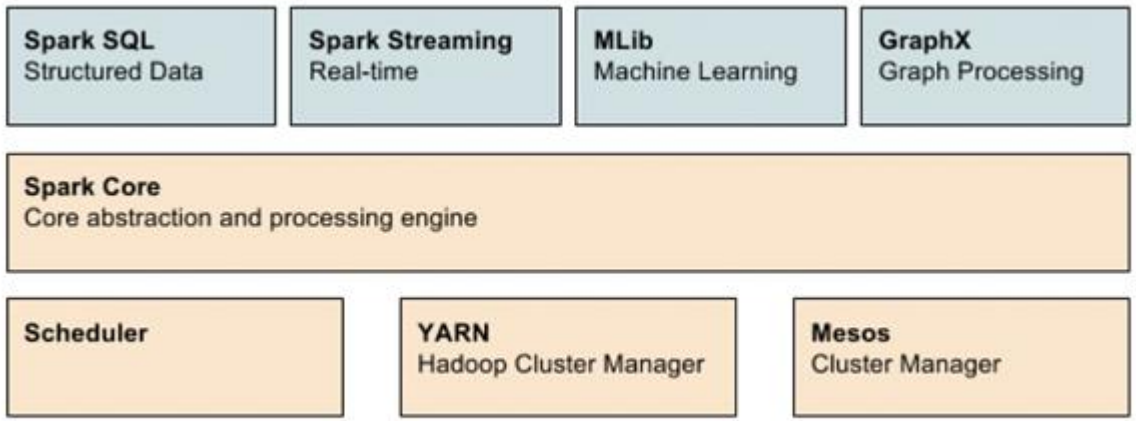
Apache Spark Component

- **Apache Spark Streaming**
  - There are 3 phases of Spark Streaming:
  - GATHERING: The Spark Streaming provides two categories of built-in streaming sources:
  - **Basic sources:** These are the sources which are available in the StreamingContext API. Examples: file systems, and socket connections.
  - **Advanced sources:** These are the sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. Hence Spark access data from different sources like Kafka, Flume, Kinesis, or TCP sockets.



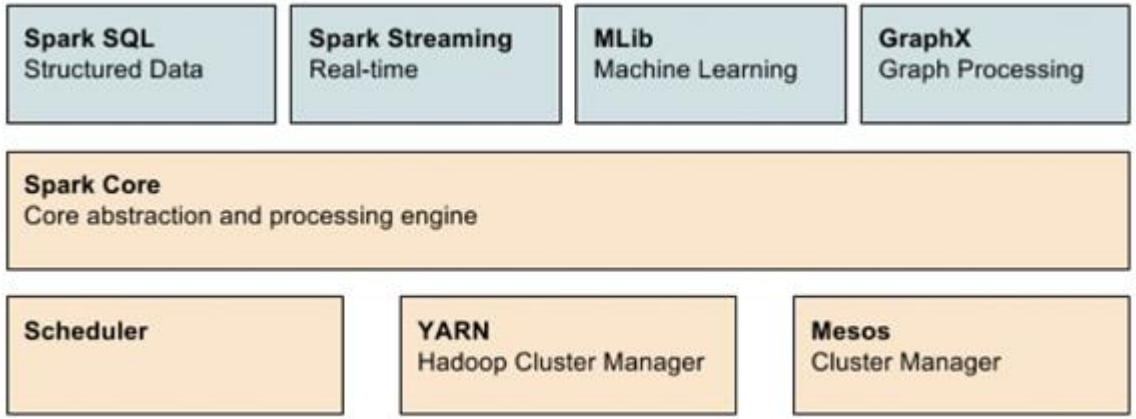
Apache Spark Component

- **Apache Spark Streaming**
  - **PROCESSING:** The gathered data is processed using complex algorithms expressed with a high-level function. For example, map, reduce, join and window. Refer this guide to learn Spark Streaming transformations operations.
  - **DATA STORAGES:** The Processed data is pushed out to file systems, databases, and live dashboards.
  - Spark Streaming also provides high-level abstraction. It is known as **discretized stream or DStream.**



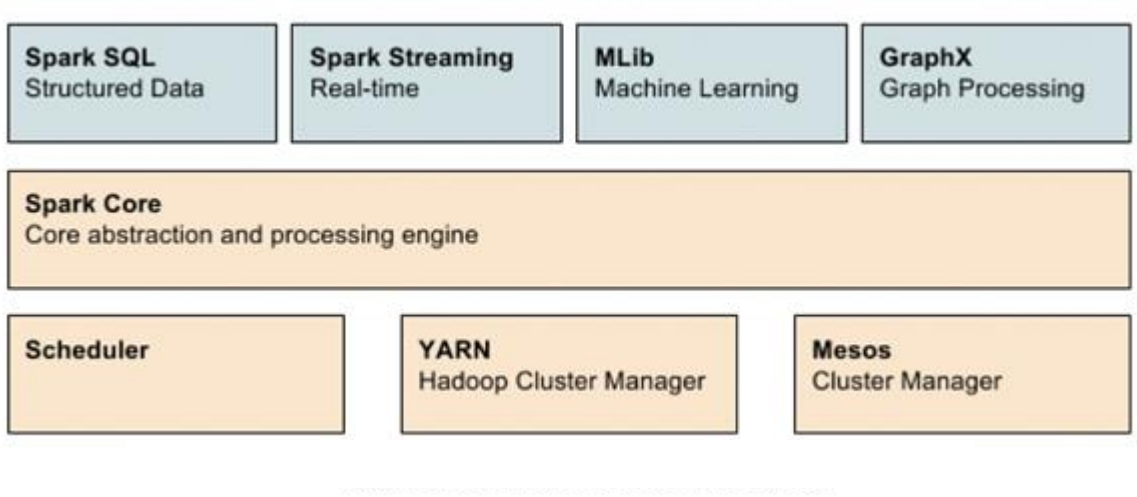
Apache Spark Component

- **Apache Spark Streaming**
  - **DStream** in Spark signifies continuous stream of data. We can form DStream in two ways either from sources such as Kafka, Flume, and Kinesis or by high-level operations on other DStreams. Thus, DStream is internally a sequence of RDDs.
  - In Spark streaming, integration of the streaming data with historical data is possible. We can also reuse the same code for stream and batch processing.
  - Spark Streaming allows exactly-once message guarantees. It helps to recover lost data without having to write adding extra configurations.



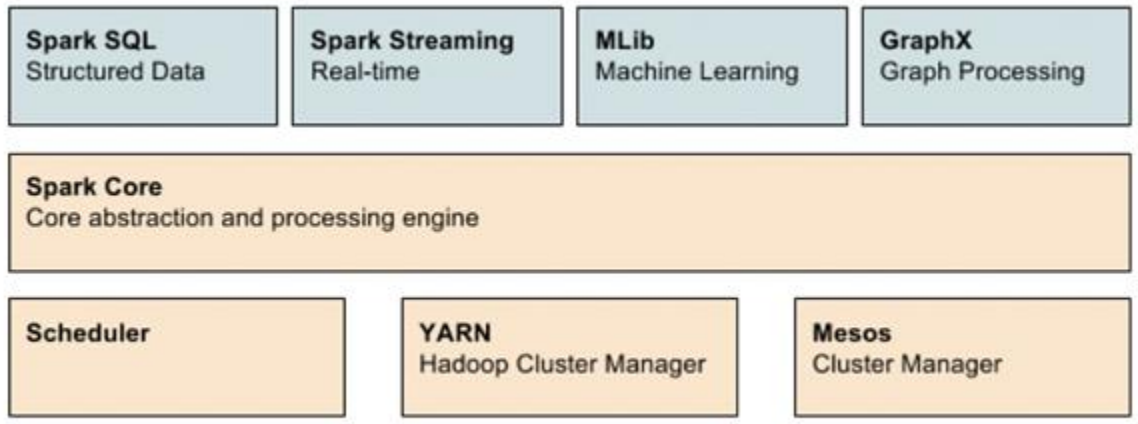
Apache Spark Component

- **MLlib (Machine Learning Library)**
  - It consists of common learning algorithms as well as utilities.
  - This library also includes classification, regression, clustering & many more.
  - It is also capable of performing in-memory data processing.
  - That enhances the performance of iterative algorithm drastically.



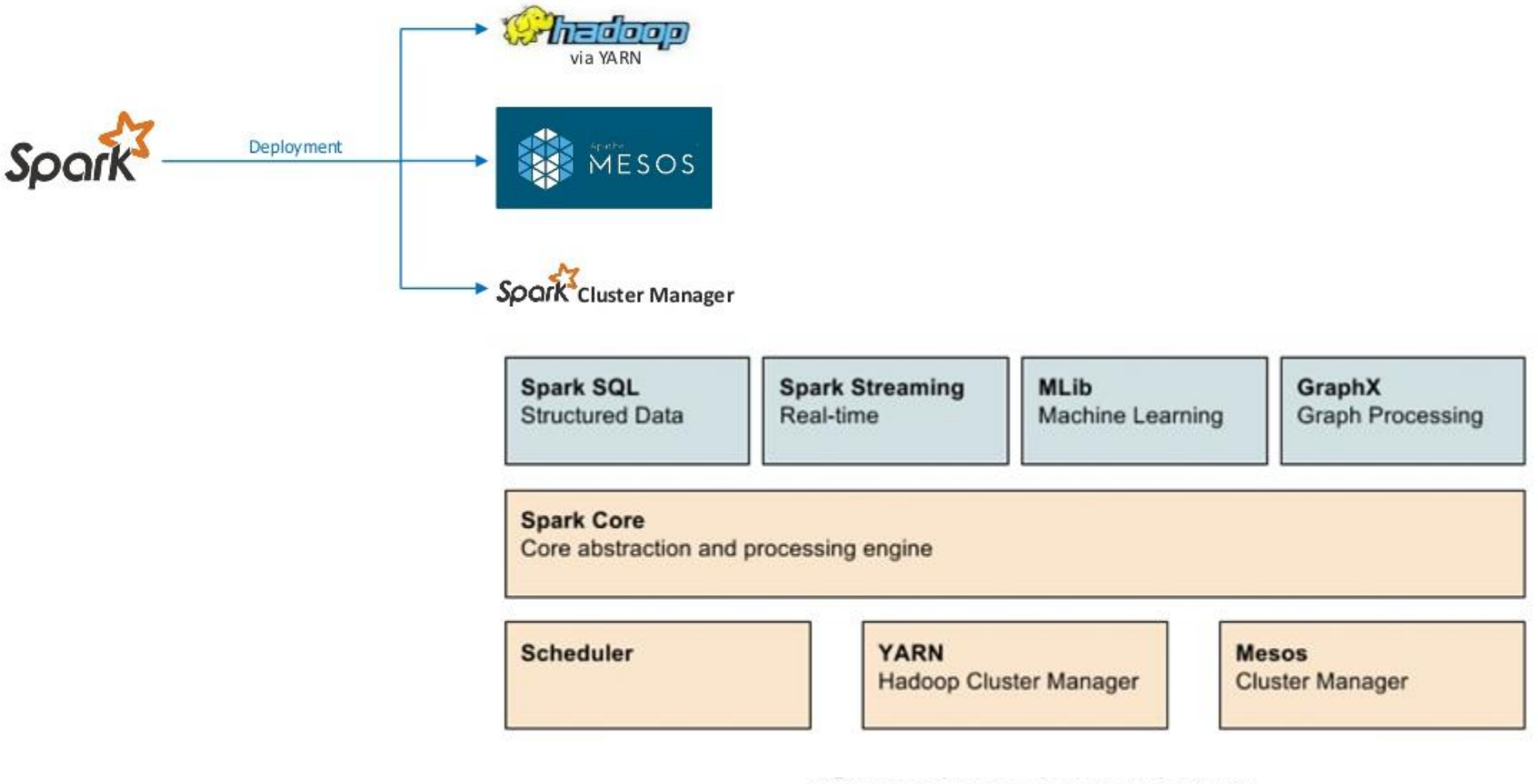
Apache Spark Component

- **GraphX**
  - API for graphs and graph parallel execution.
  - It is network graph analytics engine and data store.
  - Clustering, classification, traversal, searching, and pathfinding is also possible in graphs.
  - It has its own Graph Computation Engine for graphs and graphical computations.
  - GraphX also optimizes the way in which we can represent vertex and edges when they are primitive data types.

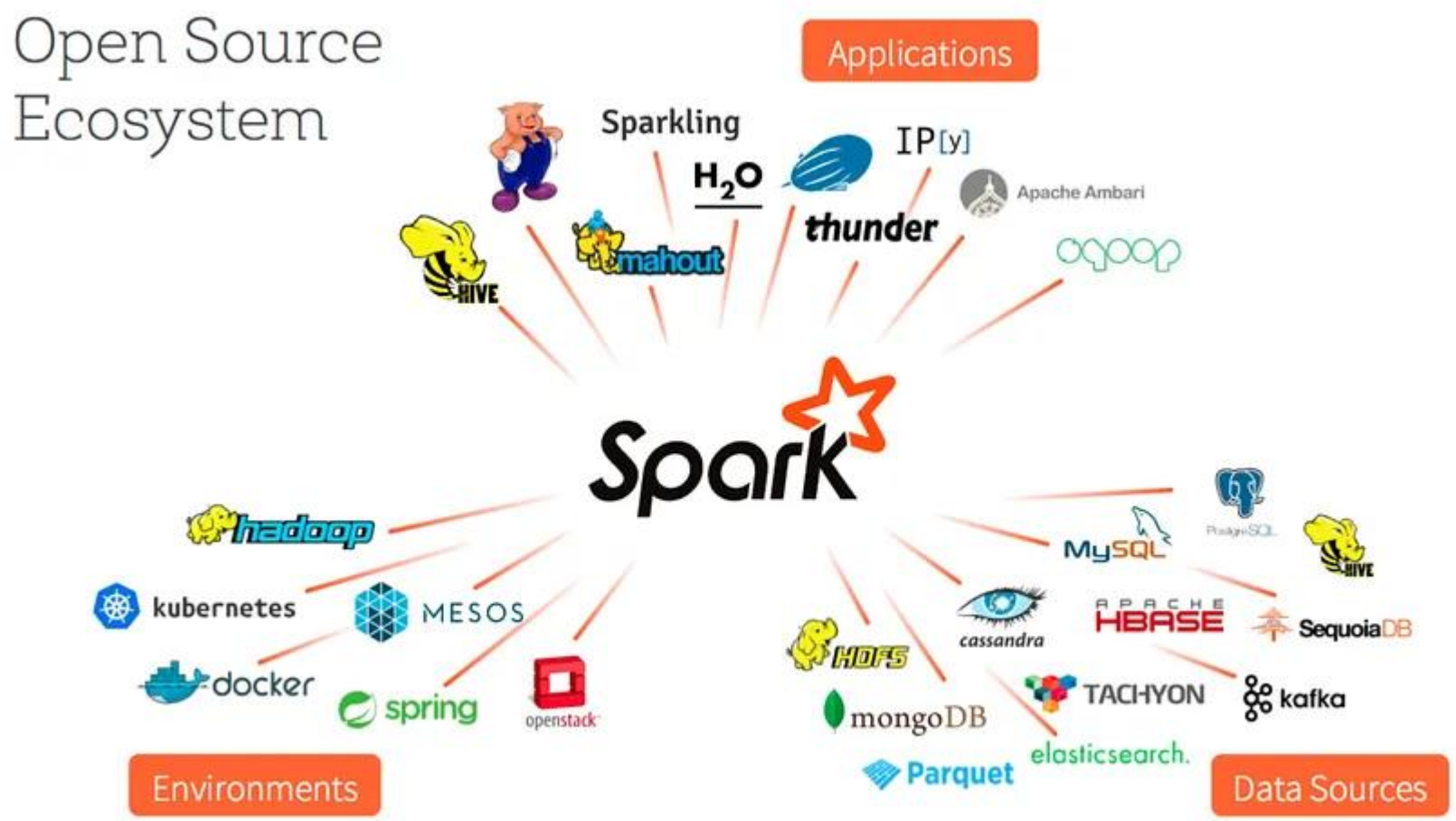


Apache Spark Component

- The two cluster managers, Apache Mesos and Hadoop YARN, manage the underlying data that we want to analyze.



Apache Spark Component



Apache Spark Features

- **Speed:** Spark performs up to 100 times faster than MapReduce for processing large amounts of data. It is also able to divide the data into chunks in a controlled way.
- **Powerful Caching:** Powerful caching and disk persistence capabilities are offered by a simple programming layer.
- **Deployment:** Mesos, Hadoop via YARN, or Spark's own cluster manager can all be used to deploy it.
- **Real-Time:** Because of its in-memory processing, it offers real-time computation and low latency.

Polyglot: Java, Scala, Python, and R, Spark supports all four of these languages. You can write Spark code in any one of these languages. Spark also provides a command-line interface in Scala and Python.

Supported Formats

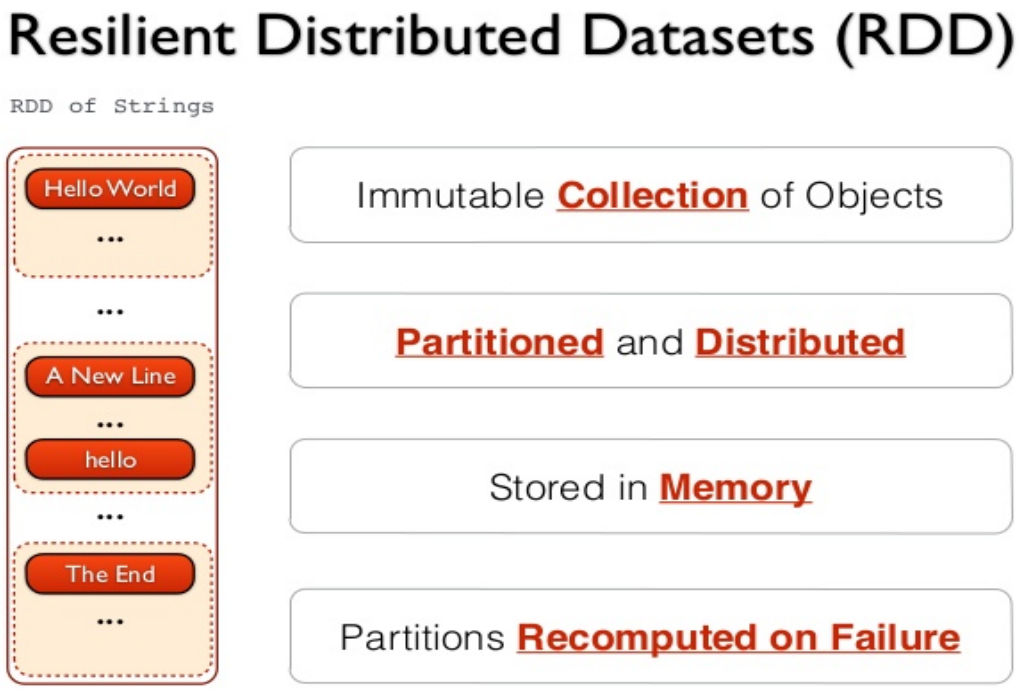
- JSON files
- Parquet files
- Hive tables
- local file systems
- distributed file systems (HDFS)
- cloud storage (S3)
- external relational database systems via JDBC

Introduction to Apache Spark

Basics of Apache Spark and Resilient Distributed Datasets (RDDs)

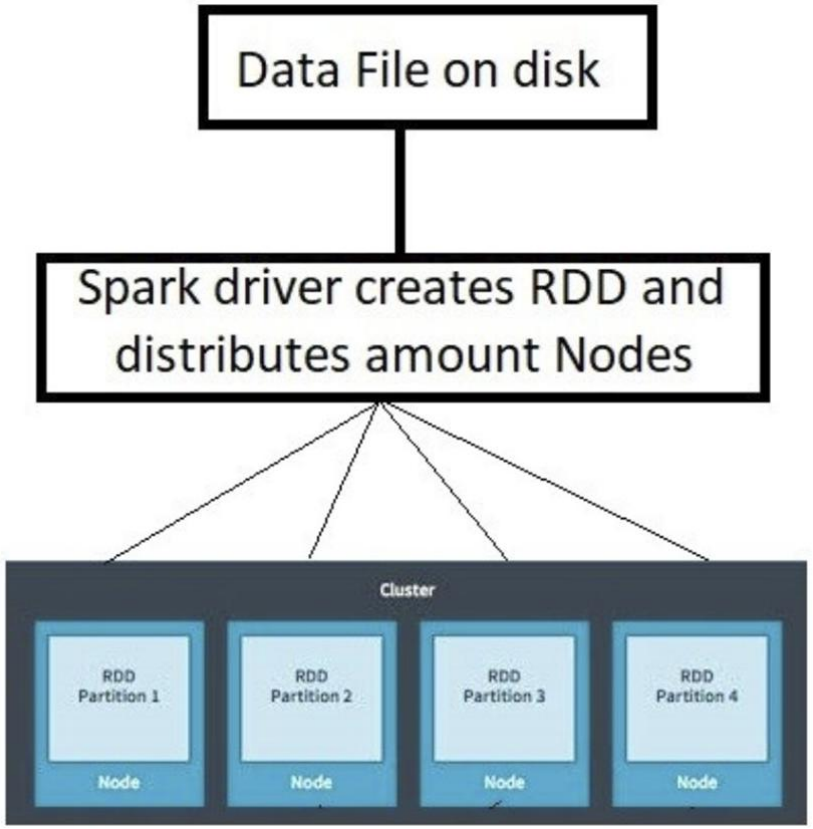
Resilient Distributed Datasets (RDDs)

- Resilient Distributed Datasets (RDDs) in Spark is an API of Spark. It collects all the elements of the data in the cluster which are well partitioned.
- ***Resilient - meaning ability to re-computed from history which in turn fault tolerant***.



Resilient Distributed Datasets (RDDs)

- A distributed memory abstraction that enables in-memory computations on large clusters in a fault-tolerant manner
- Motivation: iterative algorithms, interactive data mining tools
  - In both cases above keeping data in memory will help enormously for performance improvement
- RDDs are parallel data structures allowing coarse grained transformations
- It provides fault-tolerance by storing the lineage as opposed to the actual data as done in Hadoop



Resilient Distributed Datasets (RDDs)

The building block of the Spark API

(http://spark.apache.org/docs/latest/programming-guide.html#resilient-d istributed-datasets-rdds)

In RDD API there are two types of operations:

1. *Transformations* that define a new data set based on previous ones
2. *Actions* which kick off a job to execute on a cluster

Creation of RDD

- **Using Parallelized collection:**
  - This is a basic method to create RDD which is applied at the very initial stage of spark.
  - It creates RDD very quickly. It also initializes further operations on them at the same time.
  - To operate this method, we need entire dataset on one machine.
  - **RDDs** are generally created by parallelized collection i.e. by taking an existing collection in the program and passing it to **SparkContext's** parallelize() method.

Using Parallelized collection

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.sparkContext.parallelize([(1, 2, 3, 'a b c'),
                (4, 5, 6, 'd e f'),
                (7, 8, 9, 'g h i')]).toDF(['col1', 'col2', 'col3','col4'])
```

```
df.show()

+----+----+----+-----+
```

```
|col1|col2|col3| col4|
+----+----+----+-----+
|   1|   2|   3|a b c|
|   4|   5|   6|d e f|
|   7|   8|   9|g h i|
+----+----+----+-----+
```

Once created RDD performs two types of operations: Transformations and Actions

By using creatDataFrame() function

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

Employee = spark.createDataFrame([
                        ('1', 'Joe',   '70000', '1'),
                        ('2', 'Henry', '80000', '2'),
                        ('3', 'Sam',   '60000', '2'),
                        ('4', 'Max',   '90000', '1')],
                        ['Id', 'Name', 'Sallary','DepartmentId']
                       )
```

Then you will get the RDD data:

```
+---+-----+-------+------------+
| Id| Name|Sallary|DepartmentId|
+---+-----+-------+------------+
|  1|  Joe|  70000|           1|
|  2|Henry|  80000|           2|
|  3|  Sam|  60000|           2|
|  4|  Max|  90000|           1|
+---+-----+-------+------------+
```

Once created RDD performs two types of operations:
Transformations and Actions

By using read and load functions

- Reading data from a CSV file

```
## set up  SparkSession
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.read.format('com.databricks.spark.csv').\
                                options(header='true', \
                                inferschema='true').\
                load("/home/feng/Spark/Code/data/Advertising.csv",
↪header=True)

df.show(5)
df.printSchema()
```

Once created RDD performs two types of operations: Transformations and Actions

By using **read** and **load** functions

- Than you will get your RDD data

```
+---+-----+-----+---------+-----+
|_c0|   TV|Radio|Newspaper|Sales|
+---+-----+-----+---------+-----+
|  1|230.1| 37.8|     69.2| 22.1|
|  2| 44.5| 39.3|     45.1| 10.4|
|  3| 17.2| 45.9|     69.3|  9.3|
|  4|151.5| 41.3|     58.5| 18.5|
|  5|180.8| 10.8|     58.4| 12.9|
+---+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- _c0: integer (nullable = true)
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

Once created RDD performs two types of operations: Transformations and Actions

Read data from dataset

```
## set up  SparkSession
from pyspark.sql import SparkSession

spark = SparkSession \
            .builder \
            .appName("Python Spark create RDD example") \
            .config("spark.some.config.option", "some-value") \
            .getOrCreate()

## User information
user = 'your_username'
pw   = 'your_password'

## Database information
table_name = 'table_name'
url = 'jdbc:postgresql://##.###.###.##:5432/dataset?user='+user+'&
↪password='+pw
properties ={'driver': 'org.postgresql.Driver', 'password': pw,'user
↪': user}

df = spark.read.jdbc(url=url, table=table_name,␣
↪properties=properties)

df.show(5)
df.printSchema()
```

Once created RDD performs two types of operations: Transformations and Actions

Read data from dataset

- Than you will get your RDD data

```
+---+-----+-----+---------+-----+
|_c0|   TV|Radio|Newspaper|Sales|
+---+-----+-----+---------+-----+
|  1|230.1| 37.8|     69.2| 22.1|
|  2| 44.5| 39.3|     45.1| 10.4|
|  3| 17.2| 45.9|     69.3|  9.3|
|  4|151.5| 41.3|     58.5| 18.5|
|  5|180.8| 10.8|     58.4| 12.9|
+---+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- _c0: integer (nullable = true)
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

Once created RDD performs two types of operations: Transformations and Actions

Read dataset from HDFS

```
from pyspark.conf import SparkConf
from pyspark.context import SparkContext
from pyspark.sql import HiveContext
```

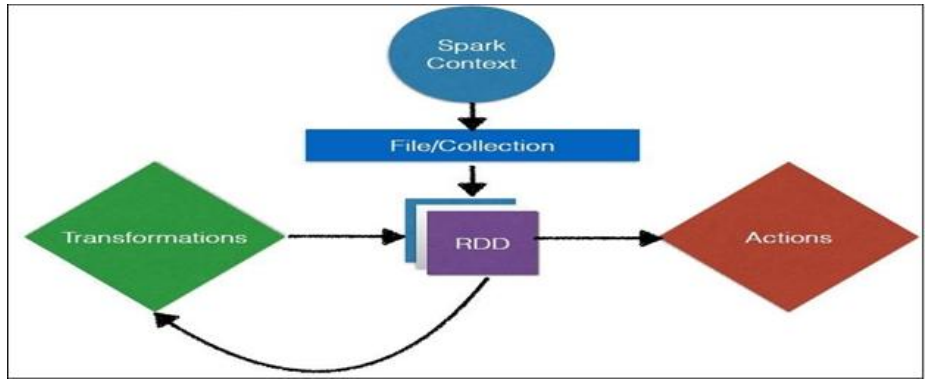
```
sc= SparkContext('local','example')
hc = HiveContext(sc)
tf1 = sc.textFile("hdfs://cdhstltest/user/data/demo.CSV")
print(tf1.first())

hc.sql("use intg_cme_w")
spf = hc.sql("SELECT * FROM spf LIMIT 100")
print(spf.show(5))
```

Once created RDD performs two types of operations: Transformations and Actions

Apache Spark RDD

- We can perform different operations on RDD
  - **Transformations** mean to create a new data sets from the existing ones. As we know RDDs are Immutable, we can transform the data from one to another.
  - **Actions** are operations that return a value to the program. All the transformations done on a Resilient Distributed Datasets are later applied when an action is called.



Transformations	Actions
map (func)	reduce(func)
flatMap(func)	collect()
filter(func)	count()
groupByKey()	first()
reduceByKey(func)	take(n)
mapValues(func)	saveAsTextFile(path)
sample(...)	countByKey()
union(other)	foreach(func)
distinct()	...
sortByKey()	
...	

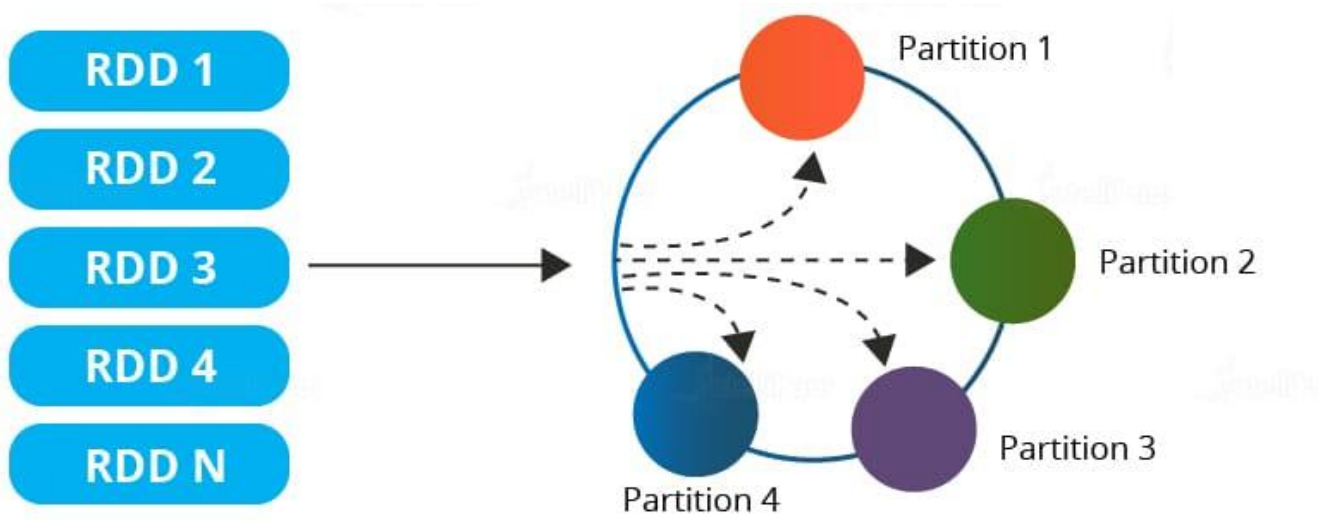
Representing RDDs

Each RDD is represented through a common interface that exposes 5 pieces of information:

1. A set of partitions, atomic pieces of datasets
2. Set of dependencies on the parent RDDs
3. Function for computing the RDD from the parents
4. Metadata about partitioning scheme
5. Data placement

RDD Operations

- The **coarse-grained** operation means to apply operations on all the objects at once. (Entire cluster simultaneously)
- **Fine-grained** operations mean to apply operations on a smaller set.
- Through its name, RDD itself indicating its properties like:
  - **Resilient** – Means that it is able to withstand all the losses itself.
  - **Distributed** – Indicates that the data at different locations or partitioned.
  - **Datasets** – Group of data on which we are performing different operations.



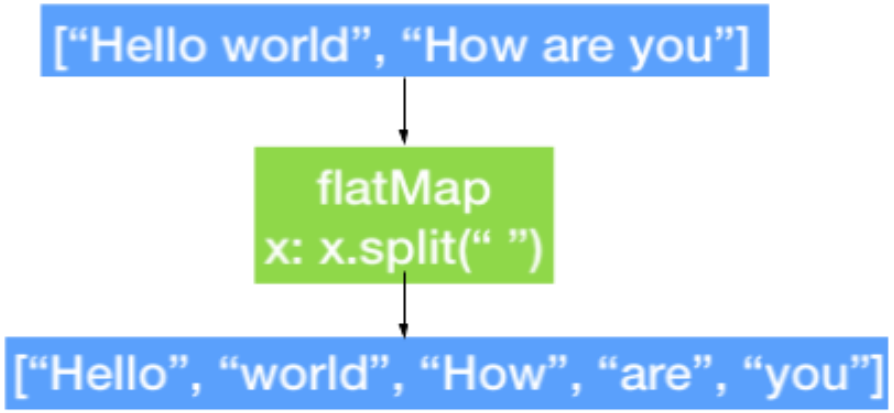
Python Lambda Functions

- Lambda functions are anonymous functions (i.e., they do not have a name) that are created at runtime.
- They can be used wherever function objects are required and are syntactically restricted to a single expression. The following example shows a lambda function that returns the sum of its two arguments:

  lambda a, b: a + b
- Lambdas are defined by the keyword lambda, followed by a comma-separated list of arguments.
- A colon separates the function declaration from the function expression.

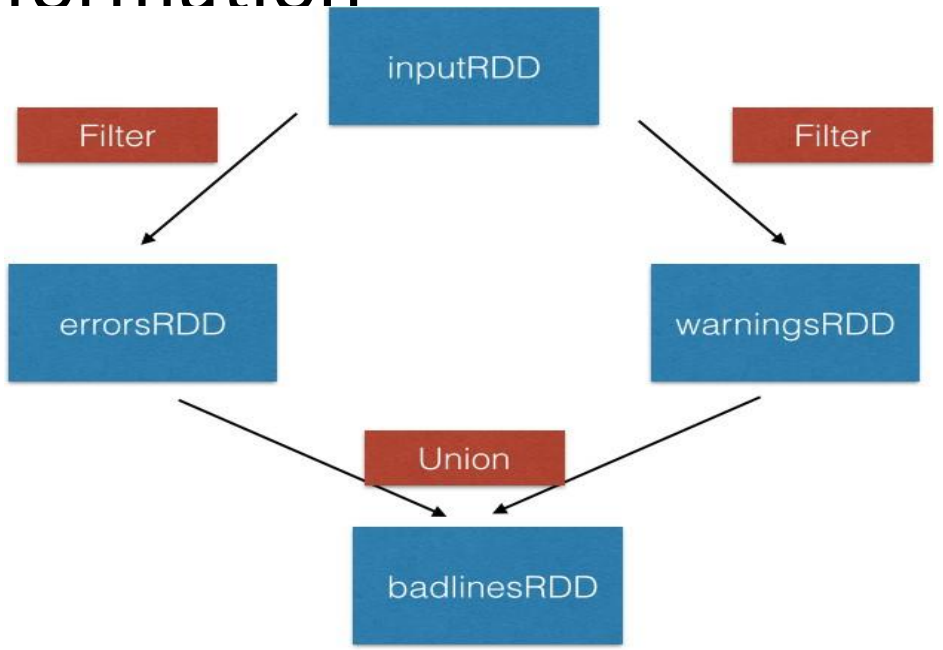
flatMap() Transformation

flatMap() transformation returns multiple values for each element in the original RDD



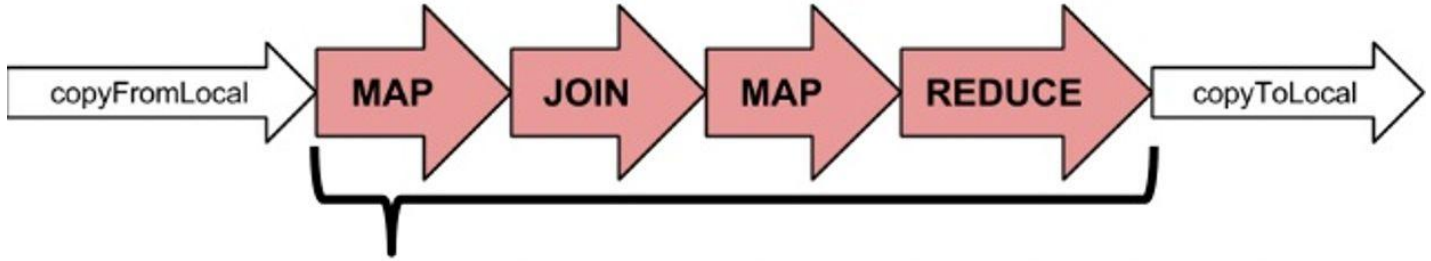

```
RDD = sc.parallelize(["hello world", "how are you"])
RDD_flatmap = RDD.flatMap(lambda x: x.split(" "))
```

union() Transformation




```
inputRDD = sc.textFile("logs.txt")
errorRDD = inputRDD.filter(lambda x: "error" in x.split())
warningsRDD = inputRDD.filter(lambda x: "warnings" in x.split())
combinedRDD  =  errorRDD.union(warningsRDD)
```

RDD Transformation vs. Action



- **Transformations are lazy:** nothing actually happens when this code is evaluated
- **RDDs are computed only when an *action* is called on them, e.g.,**
  - Calculate statistics over the elements of an RDD (`count`, `mean`)
  - Save the RDD to a file (`saveAsTextFile`)
  - Reduce elements of an RDD into a single object or value (`reduce`)
- **Allows you to define partitioning/caching behavior *after* defining the RDD but *before* calculating its contents**

Transformations
map (func)
flatMap(func)
filter(func)
groupByKey()
reduceByKey(func)
mapValues(func)
sample(...)
union(other)
distinct()
sortByKey()
...

Actions
reduce(func)
collect()
count()
first()
take(n)
saveAsTextFile(path)
countByKey()
foreach(func)
...

Apache Spark Transformation Operations

map(func)

- **Description:** Applies a function to each element in the RDD and returns a new RDD.
- **Example:** Doubling each number in an RDD.

- **numbers = sc.parallelize([1, 2, 3, 4, 5])**
- **doubled_numbers = numbers.map(lambda x: x * 2)**
- **doubled_numbers.collect() # Output: [2, 4, 6, 8, 10]**

Apache Spark Transformation Operations

filter(func)

- **Description**: Returns a new RDD containing only the elements that satisfy a condition.
- **Example**: Filtering even numbers from an RDD.

- **numbers = sc.parallelize([1, 2, 3, 4, 5])**
- **even_numbers = numbers.filter(lambda x: x % 2 == 0)**
- **even_numbers.collect() # Output: [2, 4]**

Apache Spark Transformation Operations

flatMap(func)

- **Description:** Similar to map, but each input item can be mapped to 0 or more output items.
- **Example:** Splitting sentences into words.

- **sentences = sc.parallelize(["Hello world", "PySpark is fun"])**
- **words = sentences.flatMap(lambda sentence: sentence.split(" "))**
- **words.collect() # Output: ['Hello', 'world', 'PySpark', 'is', 'fun']**

Apache Spark Transformation Operations

reduceByKey(func)

- Description: When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function.
- Example: Word count.

- **words = sc.parallelize(["apple", "banana", "apple", "orange", "banana", "apple"])**
- **word_counts = words.map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)**
- **word_counts.collect() # Output: [('banana', 2), ('apple', 3), ('orange', 1)]**

Apache Spark Transformation Operations

sortByKey(ascending=True, numPartitions=None, keyfunc=lambda x: x)

- **Description:** When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset sorted by keys.
- **Example:** Sorting word count pairs by the word (key).

- **word_counts = sc.parallelize([("apple", 2), ("banana", 3), ("orange", 1)])**
- **sorted_word_counts = word_counts.sortByKey()**
- **sorted_word_counts.collect() # Output: [('apple', 2), ('banana', 3), ('orange', 1)]**

Apache Spark Transformation Operations

union(otherDataset)

- **Description:** Returns a new RDD containing all elements from the source RDD and another dataset.
- **Example:** Union of two RDDs.

- **rdd1 = sc.parallelize([1, 2, 3])**
- **rdd2 = sc.parallelize([4, 5, 6])**
- **union_rdd = rdd1.union(rdd2)**
- **union_rdd.collect() # Output: [1, 2, 3, 4, 5, 6]**

Apache Spark Transformation Operations

intersection(otherDataset)

- **Description:** Returns a new RDD that contains the intersection of elements in the source RDD and another dataset.
- **Example:** Intersection of two RDDs.

- **rdd1 = sc.parallelize([1, 2, 3, 4])**
- **rdd2 = sc.parallelize([3, 4, 5, 6])**
- **intersection_rdd = rdd1.intersection(rdd2)**
- **intersection_rdd.collect() # Output: [3, 4]**

Apache Spark Transformation Operations

distinct(numPartitions=None)

- **Description:** Returns a new RDD containing distinct elements from the source RDD.
- **Example:** Removing duplicates from an RDD.

- **rdd = sc.parallelize([1, 1, 2, 3, 3, 4])**
- **distinct_rdd = rdd.distinct()**
- **distinct_rdd.collect() # Output: [1, 2, 3, 4]**

Apache Spark Transformation Operations

groupByKey(numPartitions=None)

- **Description:** When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as sum or average) over each key, using reduceByKey or aggregateByKey will yield better performance.

- **Example:** Grouping values by key.
- **rdd = sc.parallelize([("apple", 2), ("banana", 1), ("apple", 1)])**
- **grouped_rdd = rdd.groupByKey()**
- **grouped_rdd.mapValues(list).collect() # Output: [('apple', [2, 1]), ('banana', [1])]**

Apache Spark Transformation Operations

sortBy(lambda x: x, ascending=True)

- **Description:** Sorts the RDD by the given function, which should return the value to sort by. This can be used to sort by key, value, or any derived value from the RDD's elements.
- **Example:** Sorting an RDD of numbers in descending order.

- **rdd = sc.parallelize([5, 3, 1, 2, 4])**
- **sorted_rdd = rdd.sortBy(lambda x: x, ascending=False)**
- **sorted_rdd.collect() # Output: [5, 4, 3, 2, 1]**

Apache Spark Transformation Operations

mapValues(func)

- **Description**: Applies a function to each value of a pair RDD without changing the key.
- **Example**: Adding one to each value in a pair RDD.

- **rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])**
- **incremented_values_rdd = rdd.mapValues(lambda x: x + 1)**
- **incremented_values_rdd.collect() # Output: [('a', 2), ('b', 2), ('a', 2)]**

Apache Spark Transformation Operations

flatMapValues(func)

- **Description**: Applies a function to each value of a pair RDD and flattens the result.
- **Example**: Expanding values in a pair RDD.

- **rdd = sc.parallelize([("a", [1, 2, 3]), ("b", [4, 5])])**
- **flattened_values_rdd = rdd.flatMapValues(lambda x: x)**
- **flattened_values_rdd.collect() # Output: [('a', 1), ('a', 2), ('a', 3), ('b', 4), ('b', 5)]**

How RDD can be helpful

- Consolidate operations
  - Combine transformations
- Iterative operations
  - Keep the output of an iteration in
- memory till the next iteration
  - Data sharing
- Reuse the same data without reading it multiple times

Apache Spark action operations

collect()

- **Description:** Returns all the elements of the dataset as an array.
- **Example:** Collecting the elements of an RDD.

- **rdd = sc.parallelize([1, 2, 3, 4, 5])**
- **result = rdd.collect()**
- **print(result) # Output: [1, 2, 3, 4, 5]**

Apache Spark action operations

count()

- **Description**: Returns the number of elements in the dataset.
- **Example**: Counting the elements in an RDD.

- **rdd = sc.parallelize([1, 2, 3, 4, 5])**
- **result = rdd.count()**
- **print(result) # Output: 5**

Apache Spark action operations

take(n)

- **Description**: Returns an array with the first n elements of the dataset.
- **Example**: Taking the first 3 elements of an RDD.

- **rdd = sc.parallelize([1, 2, 3, 4, 5])**
- **result = rdd.take(3)**
- **print(result) # Output: [1, 2, 3]**

Apache Spark action operations

first()

- **Description**: Returns the first element in the dataset.
- **Example**: Getting the first element of an RDD.

- **rdd = sc.parallelize([1, 2, 3, 4, 5])**
- **result = rdd.first()**
- **print(result) # Output: 1**

Apache Spark action operations

reduce(func)

- **Description**: Aggregates the elements of the dataset using the function (func) supplied.
- **Example**: Adding all elements in an RDD.

- **rdd = sc.parallelize([1, 2, 3, 4, 5])**
- **result = rdd.reduce(lambda a, b: a + b)**
- **print(result) # Output: 15**

Apache Spark action operations

foreach(func)

- **Description**: Applies a function to each element of the dataset. This is an action operation that returns nothing.
- **Example**: Printing each element of an RDD.

```
def print_element(element):
    print(element)
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.foreach(print_element)
# Output: 1 2 3 4 5 (possibly not in order due to distribution)
```

Apache Spark action operations

saveAsTextFile(path)

- **Description**: Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system.
- **Example**: Saving RDD contents to a text file.

- **rdd = sc.parallelize([1, 2, 3, 4, 5])**
- **rdd.saveAsTextFile("path/to/output")**
- **# This saves the RDD contents to the specified path.**

Apache Spark action operations

countByKey()

- **Description**: Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
- **Example**: Counting the number of occurrences of each key in a pair RDD.

- **rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])**
- **result = rdd.countByKey()**
- **print(result) # Output: defaultdict(<class 'int'>, {'a': 2, 'b': 1})**

Apache Spark action operations

takeOrdered(n, key=None)

- **Description**: Returns the first n elements of the RDD, ordered by the given function if specified, or the natural order if not.
- **Example**: Getting the first 3 smallest elements from an RDD.

- **rdd = sc.parallelize([5, 3, 12, 2, 9])**
- **result = rdd.takeOrdered(3)**
- **print(result) # Output: [2, 3, 5]**

Apache Spark action operations

top(n, key=None)

- **Description**: Returns the top n elements from the RDD, using the natural order for comparison or a custom comparison function if specified.
- **Example**: Getting the top 3 elements.

- **rdd = sc.parallelize([10, 4, 2, 12, 3])**
- **result = rdd.top(3)**
- **print(result) # Output: [12, 10, 4]**

Apache Spark action operations

aggregate(zeroValue, seqOp, combOp)

- **Description**: Aggregates the elements of each partition and then the results for all the partitions, using a given combine functions and a neutral "zero value."
- **Example**: Calculating the sum and the number of elements in an RDD to compute the average.

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
# Define the zero value
zero_val = (0, 0)  # Sum, Count
# Define the sequence operation (within partition)
def seqOp(acc, value):
    return (acc[0] + value, acc[1] + 1)
# Define the combination operation (across
partitions)
def combOp(acc1, acc2):
    return (acc1[0] + acc2[0], acc1[1] + acc2[1])
result = rdd.aggregate(zero_val, seqOp, combOp)
average = result[0] / result[1]
print(f"Average: {average}")
# Output: Average: 3.0
```

Apache Spark action operations

fold(zeroValue, op)

- **Description**: Aggregates the elements of each partition and then the results for all the partitions, using a given associative function and a neutral "zero value."
- **Example**: Summing all elements in an RDD using fold.

- **rdd = sc.parallelize([1, 2, 3, 4, 5])**
- **result = rdd.fold(0, lambda a, b: a + b)**
- **print(result) # Output: 15**

Apache Spark action operations

countByValue()

- **Description**: Returns the count of each unique value in the RDD as a dictionary of (value, count) pairs.
- **Example**: Counting the frequency of each element in an RDD.

- **rdd = sc.parallelize([1, 2, 1, 3, 2, 1, 4, 4])**
- **result = rdd.countByValue()**
- **print(result) # Output: {1: 3, 2: 2, 3: 1, 4: 2}**

References

- **Advanced Analytics with Spark** by S. Ryza, U. Laserson, S. Owen and J. Wills
- **Apache Spark documentation**
  - http://spark.apache.org/
  - http://spark.apache.org/docs/latest/programming-guide.html
- **Pyspark**
  - http://spark.apache.org/docs/latest/api/python/pyspark.html
- **Resilient Distributed Dataset: A Fault-tolerant Abstraction for in-Memory Cluster Computing.** M. Zaharia et al.
  - https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf