Introduction to Big Data Streaming

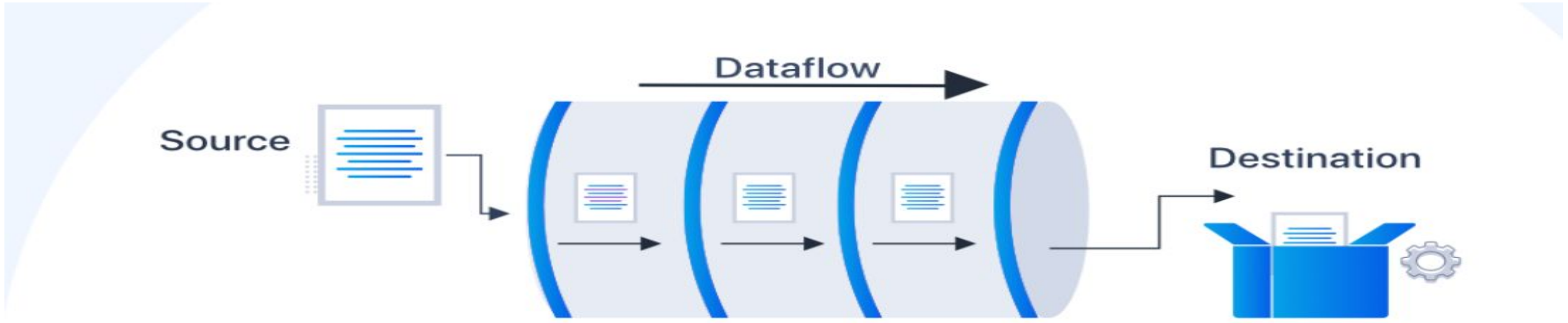
By-Shraddha Nikam

What is a Data Pipeline?

A **data pipeline** is a systematic and automated process for the efficient and reliable movement, transformation, and management of data from one point to another within a computing environment. It plays a crucial role in modern data-driven organizations by enabling the seamless flow of information across various stages of data processing.

A data pipeline consists of a series of data processing steps. If the data is not currently loaded into the data platform, then it is ingested at the beginning of the pipeline. Then there are a series of steps in which each step delivers an output that is the input to the next step. This continues until the pipeline is complete. In some cases, independent steps may be run in parallel.

Data pipelines consist of three key elements: a source, a processing step or steps, and a destination. In some data pipelines, the destination may be called a sink. Data pipelines enable the flow of data from an application to a data warehouse, from a data lake to an analytics database, or into a payment processing system , for example. Data pipelines also may have the same source and sink, such that the pipeline is purely about modifying the data set. Any time data is processed between point A and point B (or points B, C, and D), there is a data pipeline between those points.



What Is a Big Data Pipeline?

As the volume, variety, and velocity of data have dramatically grown in recent years, architects and developers have had to adapt to “big data.” The term “big data” implies that there is a huge volume to deal with. This volume of data can open opportunities for use cases such as predictive analytics, real-time reporting, and alerting, among many examples.

Like many components of data architecture, data pipelines have evolved to support big data. Big data pipelines are data pipelines built to accommodate one or more of the three traits of big data. The velocity of big data makes it appealing to build streaming data pipelines for big data. Then data can be captured and processed in real time so some action can then occur.

The volume of big data requires that data pipelines must be scalable, as the volume can be variable over time. In practice, there are likely to be many big data events that occur simultaneously or very close together, so the big data pipeline must be able to scale to process significant volumes of data concurrently. The variety of big data requires that big data pipelines be able to recognize and process data in many different formats—structured, unstructured, and semi-structured.

Benefits of a Data Pipeline

Efficiency

Data pipelines automate the flow of data, reducing manual intervention and minimizing the risk of errors. This enhances overall efficiency in data processing workflows.

Real-time Insights

With the ability to process data in real-time, data pipelines empower organizations to derive insights quickly and make informed decisions on the fly.

Scalability

Scalable architectures in data pipelines allow organizations to handle growing volumes of data without compromising performance, ensuring adaptability to changing business needs.

Data Quality

By incorporating data cleansing and transformation steps, data pipelines contribute to maintaining high data quality standards, ensuring that the information being processed is accurate and reliable.

Cost-Effective

Automation and optimization of data processing workflows result in cost savings by reducing manual labor, minimizing errors, and optimizing resource utilization.

Types of Data Pipelines

Batch Processing

Batch processing involves the execution of data jobs at scheduled intervals. It is well-suited for scenarios where data can be processed in non-real-time, allowing for efficient handling of large datasets.

Streaming Data

Streaming data pipelines process data in real-time as it is generated. This type of pipeline is crucial for applications requiring immediate insights and actions based on up-to-the-moment information.

How Data Pipelines Work

A typical data pipeline involves several key stages:

1. **Ingestion**
   Data is collected from various sources and ingested into the pipeline. This can include structured and unstructured data from databases, logs, APIs, and other sources.
2. **Processing**
   The ingested data undergoes processing, which may involve transformation, cleansing, aggregation, and other operations to prepare it for analysis or storage.
3. **Storage**
   Processed data is stored in a suitable data store, such as a database, data warehouse, or cloud storage, depending on the requirements of the organization.

Analysis

Analytical tools and algorithms are applied to the stored data to extract meaningful insights, patterns, and trends.

Visualization

The results of the analysis are presented in a visual format through dashboards or reports, making it easier for stakeholders to interpret and act upon the information.

Data Pipeline Architecture

A robust data pipeline architecture is essential for ensuring the effectiveness and scalability of the pipeline. Common components include:

Data Source

The origin of data, which could be databases, external APIs, logs, or other repositories.

Data Processing Engine

The core component responsible for transforming and manipulating the data according to predefined rules and logic.

Data Storage

Where the processed data is stored, ranging from traditional databases to fast data stores to hybrid cloud-based solutions.

Data Orchestration

The mechanism that coordinates the flow of data through the pipeline, ensuring that each step is executed in the correct sequence.

Data Monitoring and Management

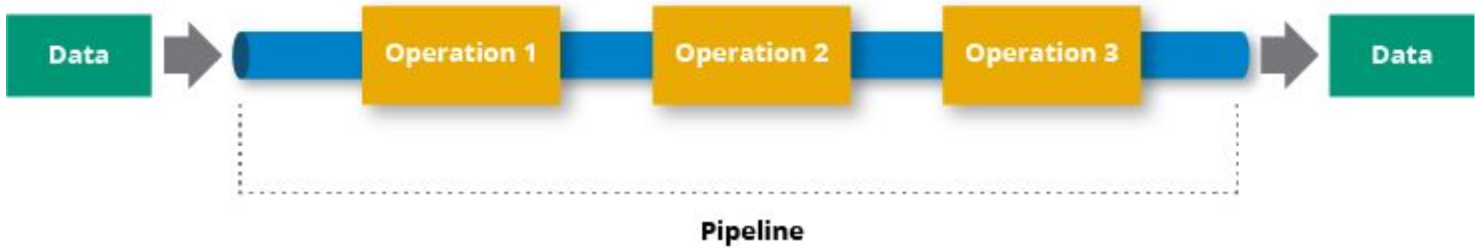
Tools and processes for monitoring the health and performance of the data pipeline, as well as managing errors and exceptions.

Data Pipeline Architecture

Data pipelines may be architected in several different ways.

1.Batch-based Data Pipeline

One common example is a batch-based data pipeline. In that example, you may have an application such as a point-of-sale system that generates a large number of data points that you need to push to a data warehouse and an analytics database. Here is an example of what that would look like:

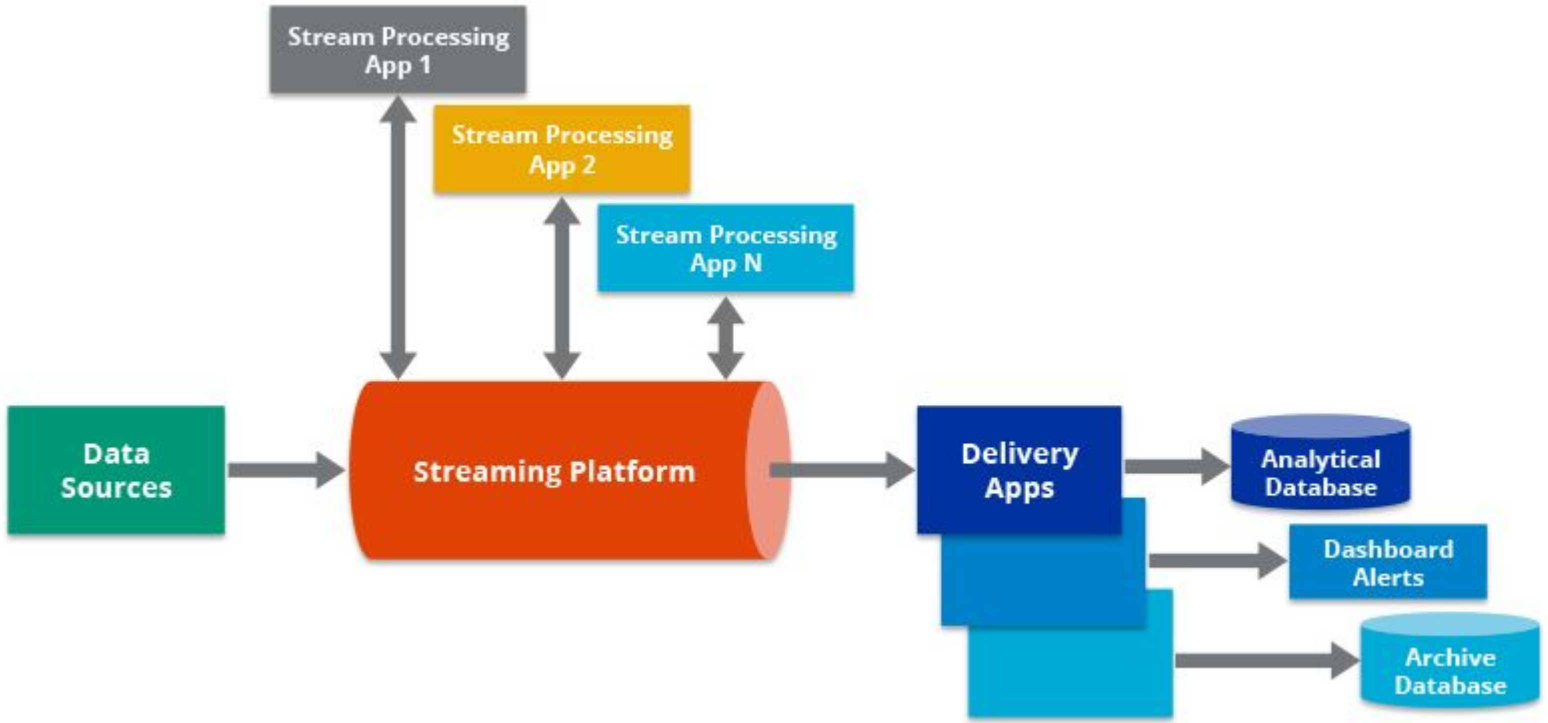


A basic example of a data pipeline.

2. Streaming Data Pipeline

Another example is a streaming data pipeline. In a streaming data pipeline, data from the point of sales system would be processed as it is generated.

The stream processing engine could feed outputs from the pipeline to data stores, marketing applications, and CRMs, among other applications, as well as back to the point of sale system itself.

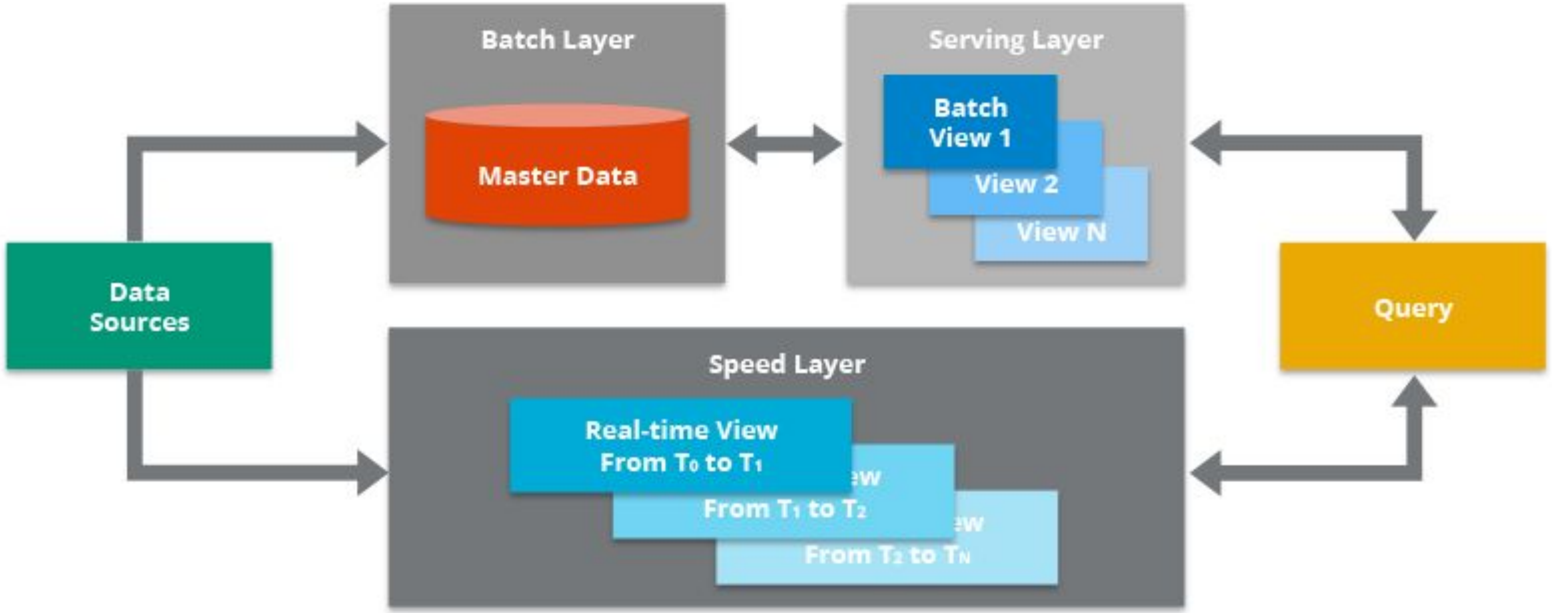


This diagram models a streaming data pipeline. The data stream is is managed by the stream processing framework where it can be processed and delivered to apps and/or solutions.

3. Lambda Architecture

A third example of a data pipeline is the Lambda Architecture, which combines batch and streaming pipelines into one architecture.

The Lambda Architecture is popular in big data environments because it enables developers to account for both real-time streaming use cases and historical batch analysis. One key aspect of this architecture is that it encourages storing data in raw format so that you can continually run new data pipelines to correct any code errors in prior pipelines, or to create new data destinations that enable new types of queries.



The Lambda Architecture accounts for both a traditional batch data pipeline and a real-time data streaming pipeline. It also has a serving layer that responds to queries.

In the diagram above, you can see the main components of the Lambda Architecture:

1.Data Sources:

Data can be obtained from a variety of sources, which can then be included in the Lambda Architecture for analysis.

This component is oftentimes a streaming source like Apache Kafka, which is not the original data source, but is an intermediary store that can hold data in order to serve both the batch layer and the speed layer of the Lambda Architecture.

The data is delivered simultaneously to both the batch layer and the speed layer to enable a parallel indexing effort.

2.Batch Layer:

This component saves all data coming into the system as batch views in preparation for indexing.

The input data is saved in a model that looks like a series of changes/updates that were made to a system of record, similar to the output of a change data capture (CDC) system.

Oftentimes this is simply a file in the comma-separated values (CSV) format. The data is treated as immutable and append-only to ensure a trusted historical record of all incoming data.

A technology like Apache Hadoop is often used as a system for ingesting the data as well as storing the data in a cost-effective way.

3. Serving Layer:

This layer incrementally indexes the latest batch views to make it queryable by end users. This layer can also reindex all data to fix a coding bug or to create different indexes for different use cases.

The key requirement in the serving layer is that the processing is done in an extremely parallelized way to minimize the time to index the data set.

While an indexing job is run, newly arriving data will be queued up for indexing in the next indexing job.

4. Speed Layer:

This layer complements the serving layer by indexing the most recently added data not yet fully indexed by the serving layer. This includes the data that the serving layer is currently indexing as well as new data that arrived after the current indexing job started.

Since there is an expected lag between the time the latest data was added to the system and the time the latest data is available for querying (due to the time it takes to perform the batch indexing work), it is up to the speed layer to index the latest data to narrow this gap.

This layer typically leverages stream processing software to index the incoming data in near real-time to minimize the latency of getting the data available for querying.

When the Lambda Architecture was first introduced, Apache Storm was a leading stream processing engine used in deployments, but other technologies have since gained more popularity as candidates for this component (like Hazelcast Jet, Apache Flink, and Apache Spark Streaming).

5. Query:

This component is responsible for submitting end user queries to both the serving layer and the speed layer and consolidating the results.

This gives end users a complete query on all data, including the most recently added data, to provide a near real-time analytics system.

Introduction to Spark Streaming

Introduction

We, as a learner, are in the stage of analyzing the data mostly in the **CSV format. Still, we** need to understand that at the enterprise level, most of the work is done in **real-time,** where we need skills to stream live data. For that, we have **Spark Streaming from** Apache Spark.

What is Spark Streaming?

Apache Spark Streaming is a scalable fault-tolerant streaming processing system that natively supports both batch and streaming workloads. Spark Streaming is an extension of the core Spark API that allows data engineers and data scientists to process real-time data from various sources including Kafka, Flume, and Amazon Kinesis.

This processed data can be pushed out to file systems, databases, and live dashboards. Its key abstraction is a Discretized Stream (DStream), which represents a stream of data divided into small batches.

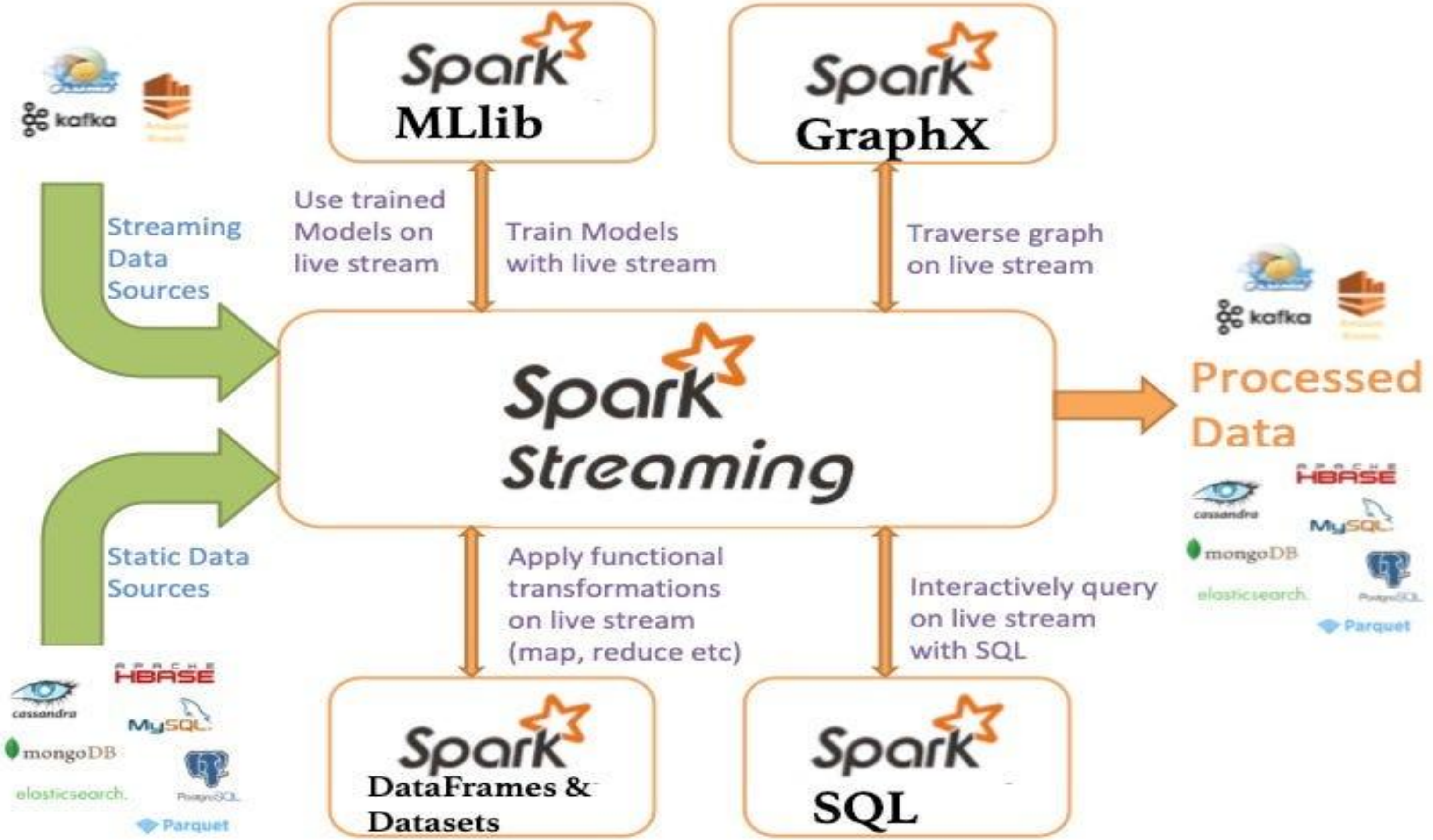
DStreams are built on RDDs, Spark’s core data abstraction. This allows Spark Streaming to seamlessly integrate with any other Spark components like MLlib and Spark SQL.

Spark Streaming is different from other systems that either have a processing engine designed only for streaming, or have similar batch and streaming APIs but compile internally to different engines.

Spark’s single execution engine and unified programming model for batch and streaming lead to some unique benefits over other traditional streaming systems.

Four Major Aspects of Spark Streaming

- Fast recovery from failures and stragglers
- Better load balancing and resource usage
- Combining of streaming data with static datasets and interactive queries
- Native integration with advanced processing libraries (SQL, machine learning, graph processing)


kafka
Streaming Data Sources
Static Data Sources
cassandra
HBASE
MySQL
mongoDB
elasticsearch
PostgreSQL
Parquet
Spark MLlib
Spark GraphX
Use trained Models on live stream
Train Models with live stream
Traverse graph on live stream
Spark Streaming
Processed Data
Apply functional transformations on live stream (map, reduce etc)
Interactively query on live stream with SQL
Spark DataFrames & Datasets
Spark SQL

What is Spark Streaming

When we hear Spark Streaming as a term, the first thing that pops up is that it will only deal with real-time data (like **Twitter developer API**). Still, we can also work with **batch processing,** i.e., a dataset in the system, though other libraries and utilities are there; hence it is widely used for **stream processing**.

Let’s discuss a few major characteristics of it.

> **Scalable:** when we say stream processing, that means it takes real-time data for the pipeline needs to be scalable, then only the purpose will be served.
>
> **Fault-tolerant:** The secret behind the fault-tolerant capacity of spark streaming is its **worker nodes** which are built on top of Spark only so similar to Spark it deals with **high uptimes** and sudden **fault detection**.

Load balancing: Whenever working with live data, we must take care of the **high traffic of data** coming up for processing. We have **load balancers** in this option. We can equally **distribute the traffic** to each resource by this; we gain 2 objectives, one is the proper usage of resources, and another is dealing with high data traffic.

Streaming and static: Spark Streaming combination can deal with **a static** dataset and **a live** dataset. For that, we need to work with **interactive queries, providing native** support to the end users.

Note: Spark Streaming is just another extension of the core Spark API

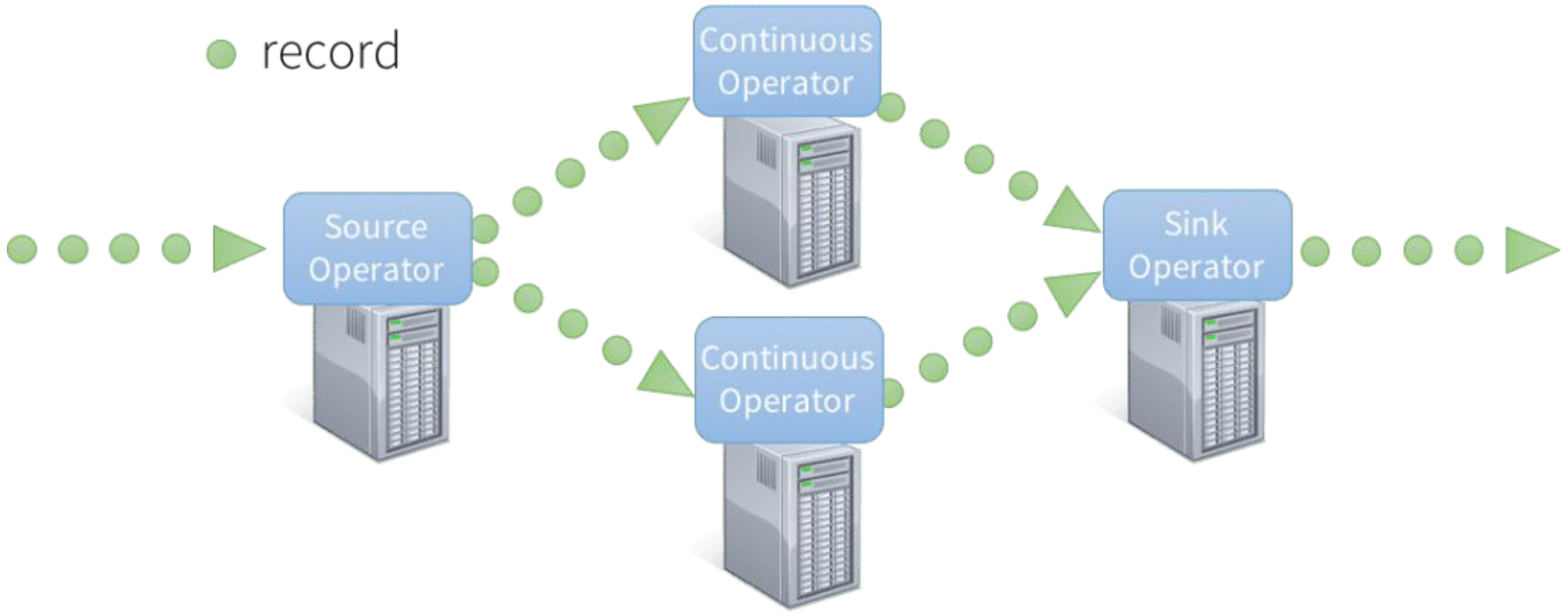
How does Spark Streaming Work?

Before digging deeper to spark Streaming specifically, let’s scratch the surface and understand how **the most modern distributed system** works:

1. **Receiving data:** This is the data ingestion process where real/static data is being received (e.g., from IoT devices) and ingested to any system like **Amazon kinesis** or **Apache Kafka**
2. **Process data:** In real-time processing, the data is processed in **parallel on clusters. From** this approach, stream data is being processed effectively and efficiently.
3. **Output:** After the data is processed, we need to collect the data as the output, which is stored on downstream systems like Cassandra, Kafka, and Hbase.

Traditional stream processing systems

continuous operator model

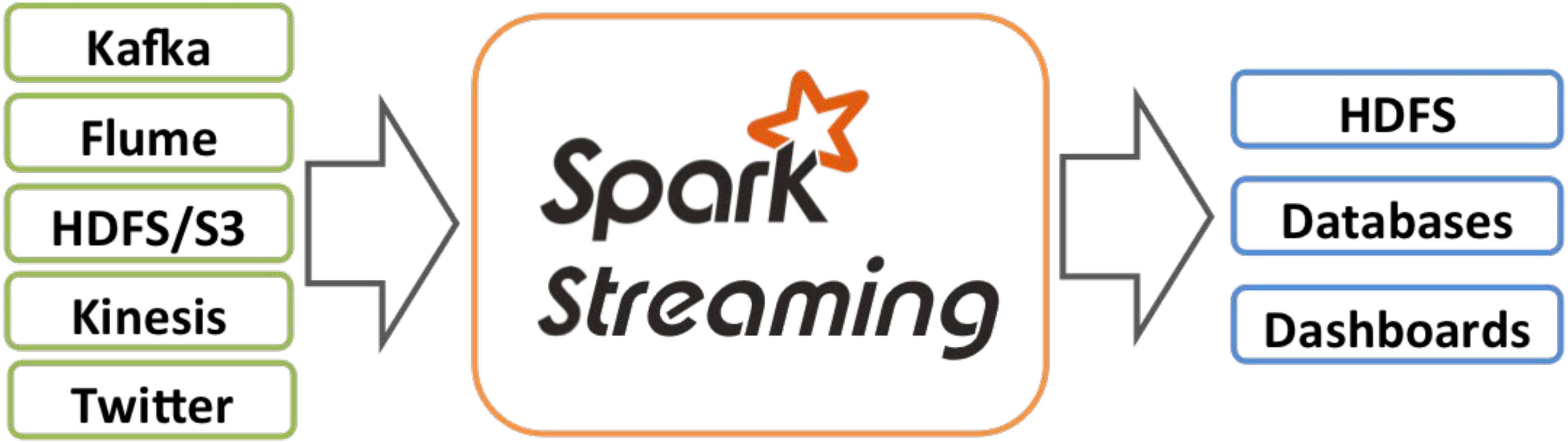


records processed one at a time

Now let’s discuss how Spark streaming works and what is the architecture model of the same; the architecture model is identical to the traditional model but with some advanced tools and techniques, which we will look into further:

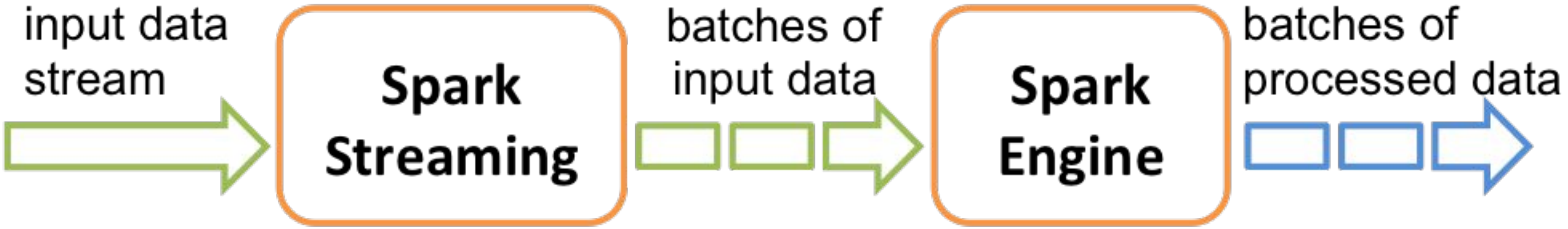
- In Spark, Streaming data is first ingested from single or multiple sources such as through networking (**TCP sockets**), **Kafka**, **Kinesis**, **IoT devices**, and so on.
- Then the data is pushed to the processing part, where Spark Streaming has several complex algorithms powered by high-throughput functions such as **window**, **map**, **join**, **reduce**, and more.
- The last step of this process will be to push the data to **large databases and dashboards** to develop analytical solutions, more specifically, the live dashboards where we can see the data visualization of those advanced graphs changing in real-time. We can also **use PySpark’s Machine learning techniques** or **graphical processing** on the streamed data.

- Ingest data from many sources: Kafka, Twitter, HDFS, TCP sockets
- Results can be pushed out to file-systems, databases, live dashboards



How does Spark Streaming Works Internally?

As it is essential to know what kind of processing technique Spark Streaming follows! The answer is simple and self-understandable as we all know that all traditional live streaming architecture works **with batch processing,** so is Apache spark. In this method, Spark streaming will simply take in the **input streams** and then **break down into batches** and process them **parallelly**.



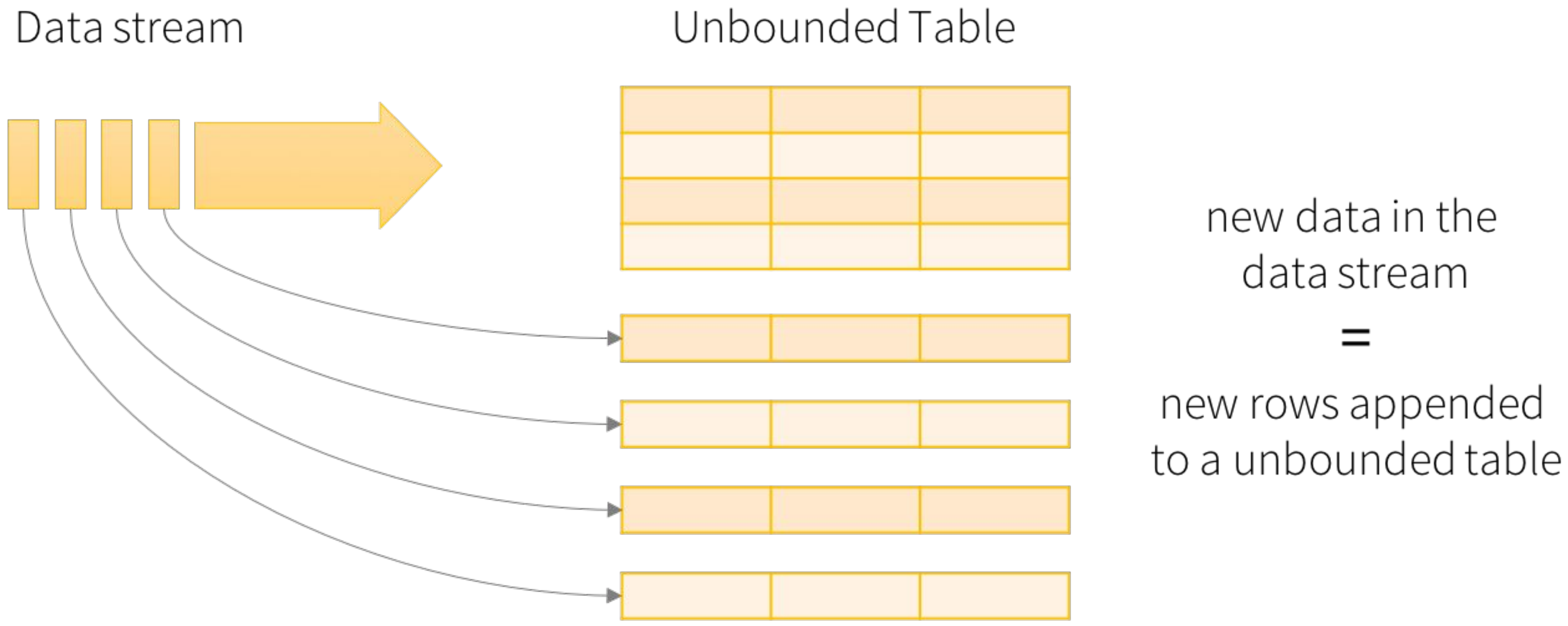
Structured Streaming

A spark supports two types of streaming; one is the **legacy project,** i.e., the **spark model,** which we have already discussed; now it’s time to have an understanding of the other one, i.e., **Structured Architecture**

Structured Streaming: Unlike Spark Streaming, This one works on top of **Spark SQL API/engine**. The best part is that there is no confusion about whether we have to compute the batch data or live streamed data, as the implementation part is almost the same. The results keep updating as long as it receives the final data from input streams for such operations; one can use the **Dataset/DataFrame API** in either of the available languages (Python, Java, Scala, and so on).

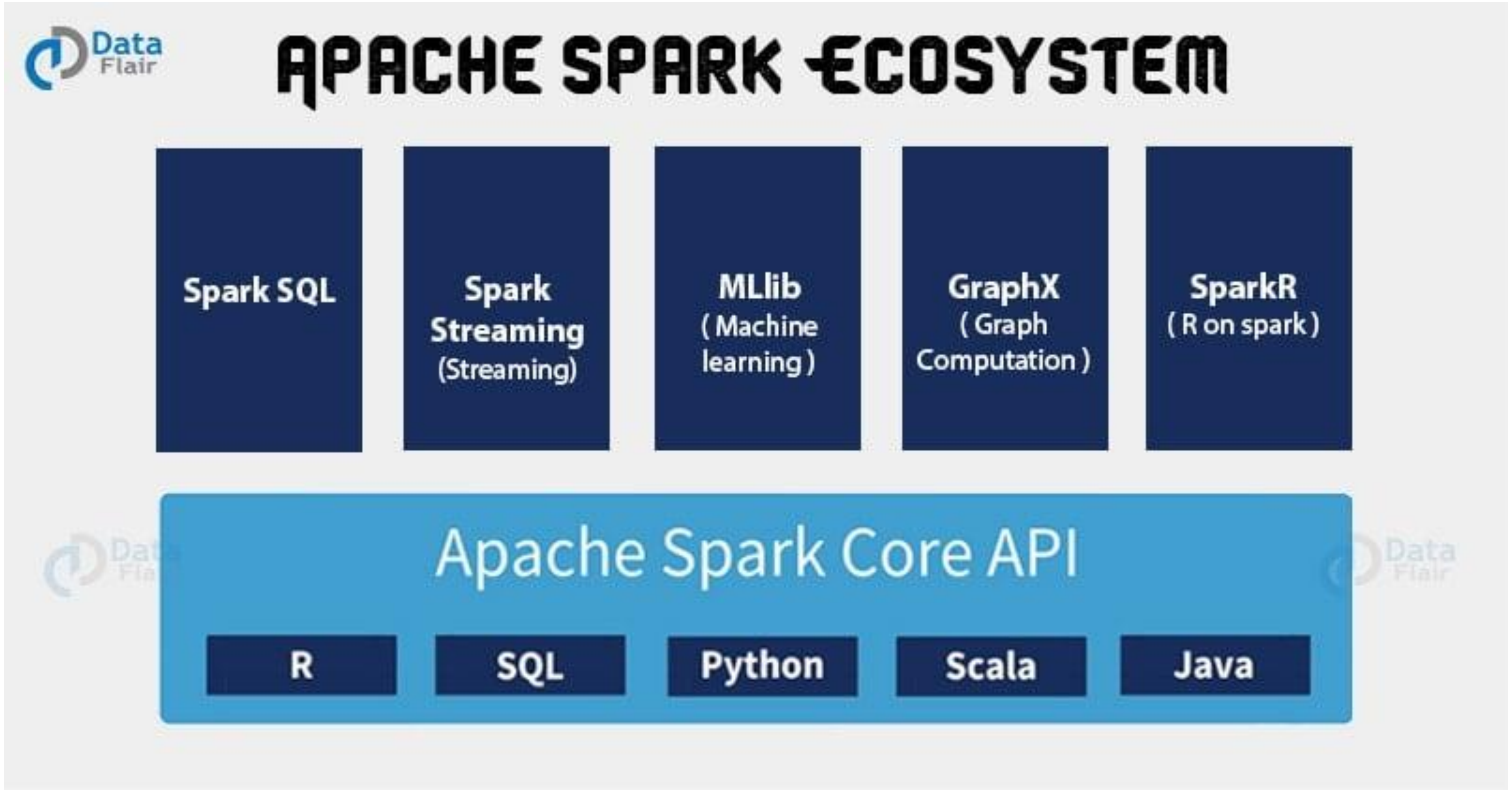
Architecture model of Structured Streaming

The main idea behind the structure streaming model is to treat it like an **unbounded table** (from unbounded, I mean, the table with no limit to store the records, or we can say it tends to increase its size every time the new data arrives). This format makes the live streaming the same as the batch processing concept, making it a bit easier in terms of implementation compared to other streaming models.


Data stream
Unbounded Table
new data in the data stream
=
new rows appended to a unbounded table

Data stream as an unbounded table

Introduction to Apache Spark Ecosystem Components



Following are 6 components in Apache Spark Ecosystem which empower to Apache Spark-

Spark Core, Spark SQL, Spark Streaming, Spark MLlib, Spark GraphX, and SparkR.

Let us now learn about these Apache Spark ecosystem components in detail below:

1. Apache Spark Core

All the functionalities being provided by Apache Spark are built on the top of **Spark Core**. It delivers speed by providing **in-memory computation** capability. Thus Spark Core is the foundation of parallel and distributed processing of huge dataset.

The key features of Apache Spark Core are:

- It is in charge of essential I/O functionalities.
- Significant in programming and observing the role of the **Spark cluster**.
- Task dispatching.
- Fault recovery.
- It overcomes the snag of **MapReduce** by using in-memory computation.

Spark Core is embedded with a special collection called **RDD** (resilient distributed dataset).

2. Apache Spark SQL

The Spark SQL component is a distributed framework for *structured* data processing. Using Spark SQL, Spark gets more information about the structure of data and the computation. With this information, Spark can perform extra optimization. It uses same execution engine while computing an output. It does not depend on API/ language to express the computation.

Spark SQL works to access structured and semi-structured information. It also enables powerful, interactive, analytical application across both streaming and historical data. Spark SQL is Spark module for structured data processing. Thus, it acts as a distributed SQL query engine.

Features of Spark SQL include:

- Cost based optimizer.
- Mid query fault-tolerance: This is done by scaling thousands of nodes and multi-hour queries using the Spark engine.
- Full compatibility with existing **Hive** data.
- **DataFrames** and SQL provide a common way to access a variety of data sources. It includes Hive, Avro, Parquet, ORC, JSON, and JDBC.
- Provision to carry structured data inside Spark programs, using either SQL or a familiar Data Frame API.

3. Apache Spark Streaming

It is an add-on to core Spark API which allows scalable, high-throughput, fault-tolerant stream processing of live data streams. Spark can access data from sources like **Kafka**, **Flume**, **Kinesis** or **TCP socket.** It can operate using various algorithms. Finally, the data so received is given to file system, databases and live dashboards. Spark uses *Micro-batching* for real-time streaming.

Micro-batching is a technique that allows a process or task to treat a stream as a sequence of small batches of data. Hence Spark Streaming, groups the live data into small batches. It then delivers it to the batch system for processing. It also provides fault tolerance characteristics

4. Apache Spark MLlib (Machine Learning Library)

MLlib in Spark is a scalable Machine learning library that discusses both high-quality algorithm and high speed.

The motive behind MLlib creation is to make machine learning scalable and easy. It contains machine learning libraries that have an implementation of various machine learning algorithms. For example, *clustering, regression, classification and collaborative filtering*. Some lower level machine learning primitives like generic gradient descent optimization algorithm are also present in MLlib.

5. Apache Spark GraphX

GraphX in Spark is 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. Furthermore, GraphX extends Spark RDD by bringing in light a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge.

GraphX also optimizes the way in which we can represent vertex and edges when they are primitive data types. To support graph computation it supports fundamental operators (e.g., subgraph, join Vertices, and aggregate Messages) as well as an optimized variant of the *Pregel API*.

6. Apache SparkR

The key component of SparkR is SparkR DataFrame. DataFrames are a fundamental data structure for data processing in **R.**

The concept of DataFrames extends to other languages with libraries like *Pandas* etc.

R also provides software facilities for data manipulation, calculation, and graphical display.

Hence, the main idea behind SparkR was to explore different techniques to integrate the usability of R with the scalability of Spark. It is R package that gives light-weight frontend to use Apache Spark from R.

There are various benefits of SparkR:

- **Data Sources API:** By tying into Spark SQL's data sources API SparkR can read in data from a variety of sources. For example, Hive tables, JSON files, Parquet files etc.
- **Data Frame Optimizations:** SparkR DataFrames also inherit all the optimizations made to the computation engine in terms of code generation, memory management.
- **Scalability to many cores and machines:** Operations that executes on SparkR DataFrames get distributed across all the cores and machines available in the **Spark cluster**. As a result, SparkR DataFrames can run on terabytes of data and clusters with thousands of machines.

Resilient Distributed Datasets (Spark RDD)

Introduction

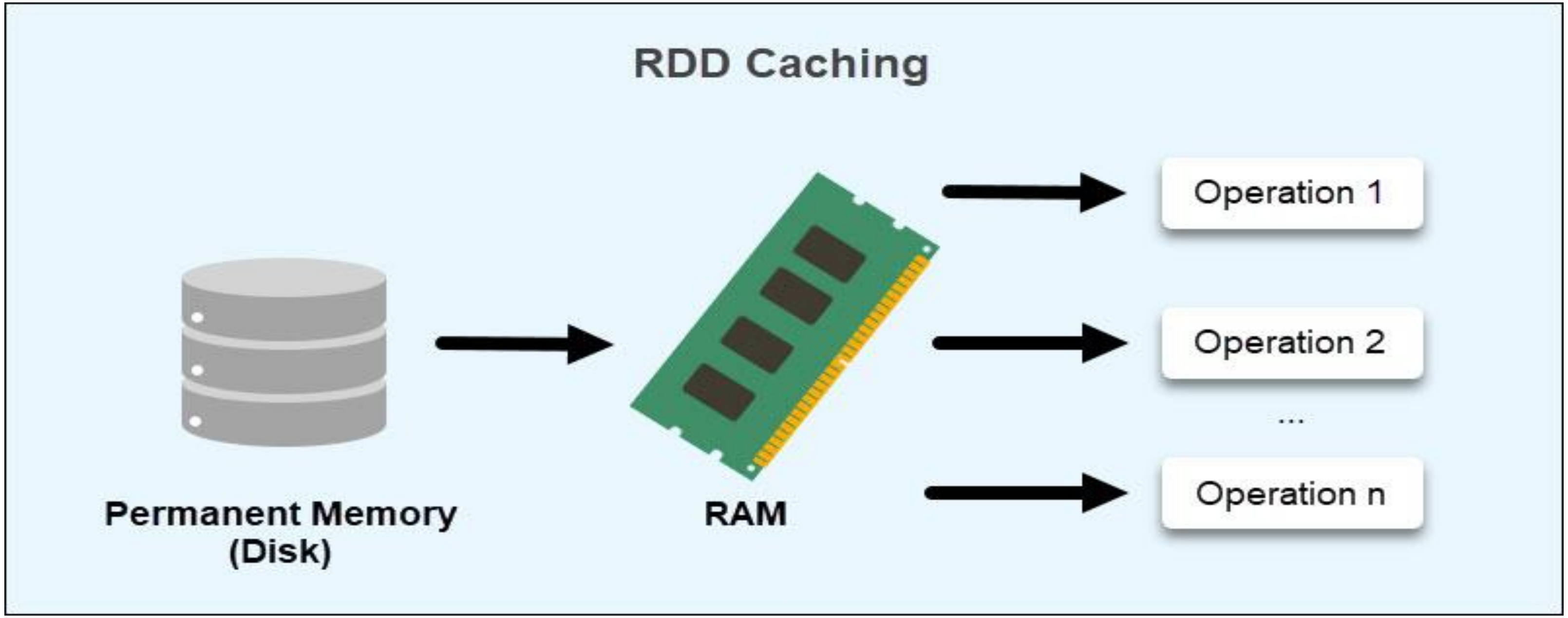
Resilient Distributed Datasets (RDDs) are the primary data structure in Spark. RDDs are reliable and memory-efficient when it comes to parallel processing. By storing and processing data in RDDs, Spark speeds up MapReduce processes.

What Is a Resilient Distributed Dataset?

A Resilient Distributed Dataset (RDD) is a low-level API and Spark's underlying data abstraction. An RDD is a static set of items distributed across clusters to allow parallel processing. The data structure stores any Python, Java, Scala, or user-created object.

Why Do We Need RDDs in Spark?

RDDs address MapReduce's shortcomings in data sharing. When reusing data for computations, MapReduce requires writing to external storage (HDFS, Cassandra, HBase, etc.)



The read and write processes between jobs consume a significant amount of memory.

Furthermore, data sharing between tasks is slow due to replication, serialization, and increased disk usage

RDDs aim to reduce the usage of external storage systems by leveraging **in-memory compute operation storage.** This approach improves data exchange speeds between tasks by 10 to 100 times.

Speed is critical when working with large data volumes. Spark RDDs make it easier to train machine learning algorithms and handle large amounts of data for analytics.

When to Use RDDs?

Spark RDDs are useful in the following scenarios:

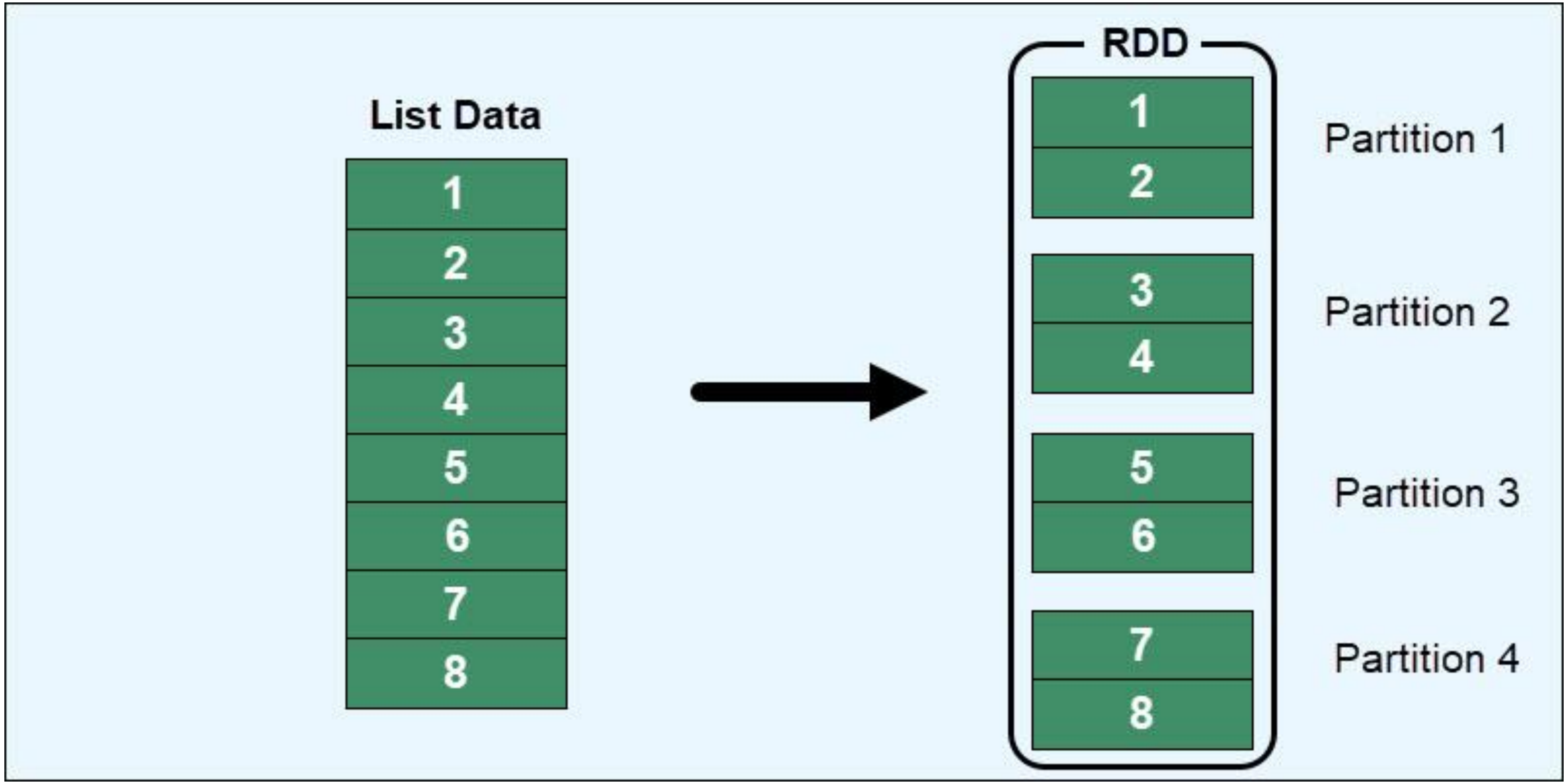
- If the data is unstructured like text and media streams, RDD will be beneficial in terms of performance.
- If the transformation is of a low level, RDD will be beneficial to fasten and straightforward the data manipulation when closer to the source of data.
- If the schema is not important, RDD will not impose it, but it will use schema to access specific data based on the column.

How Does RDD Store Data?

An RDD stores data in read-only mode, making it immutable. Performing operations on existing RDDs creates new objects without manipulating existing data.

RDDs reside in RAM through a caching process. Data that does not fit is either recalculated to reduce the size or stored on a permanent storage. Caching allows retrieving data without reading from disk, reducing disk overhead.

RDDs further distribute the data storage across multiple partitions. Partitioning allows data recovery in case a node fails and ensures the data is available at all times.


List Data
1
2
3
4
5
6
7
8
RDD
1
2
Partition 1
3
4
Partition 2
5
6
Partition 3
7
8
Partition 4

Spark's RDD uses a **persistence optimization technique** to save computation results. Two methods help achieve RDD persistence:

- **cache()**
- **persist()**

These methods provide an interactive storage mechanism by choosing different storage levels. The cached memory is fault-tolerant, allowing the recreation of lost RDD partitions through the initial creation operations.

Spark RDD Features

The main features of a Spark RDD are:

- **In-memory computation**. Data calculation resides in memory for faster access and fewer I/O operations.
- **Fault tolerance**. The tracking of data creation helps recover or recreate lost data after a node failure.
- **Immutability**. RDDs are read-only. The existing data cannot change, and transformations on existing data generate new RDDs.
- **Lazy evaluation**. Data does not load immediately after definition - the data loads when applying an action to the data.

How to create RDD?

In Apache Spark, RDDs can be created in three ways.

- Parallelize method by which already existing collection can be used in the driver program.
- By referencing a dataset that is present in an external storage system such as HDFS, HBase.
- New RDDs can be created from an existing RDD.

Spark RDD Operations

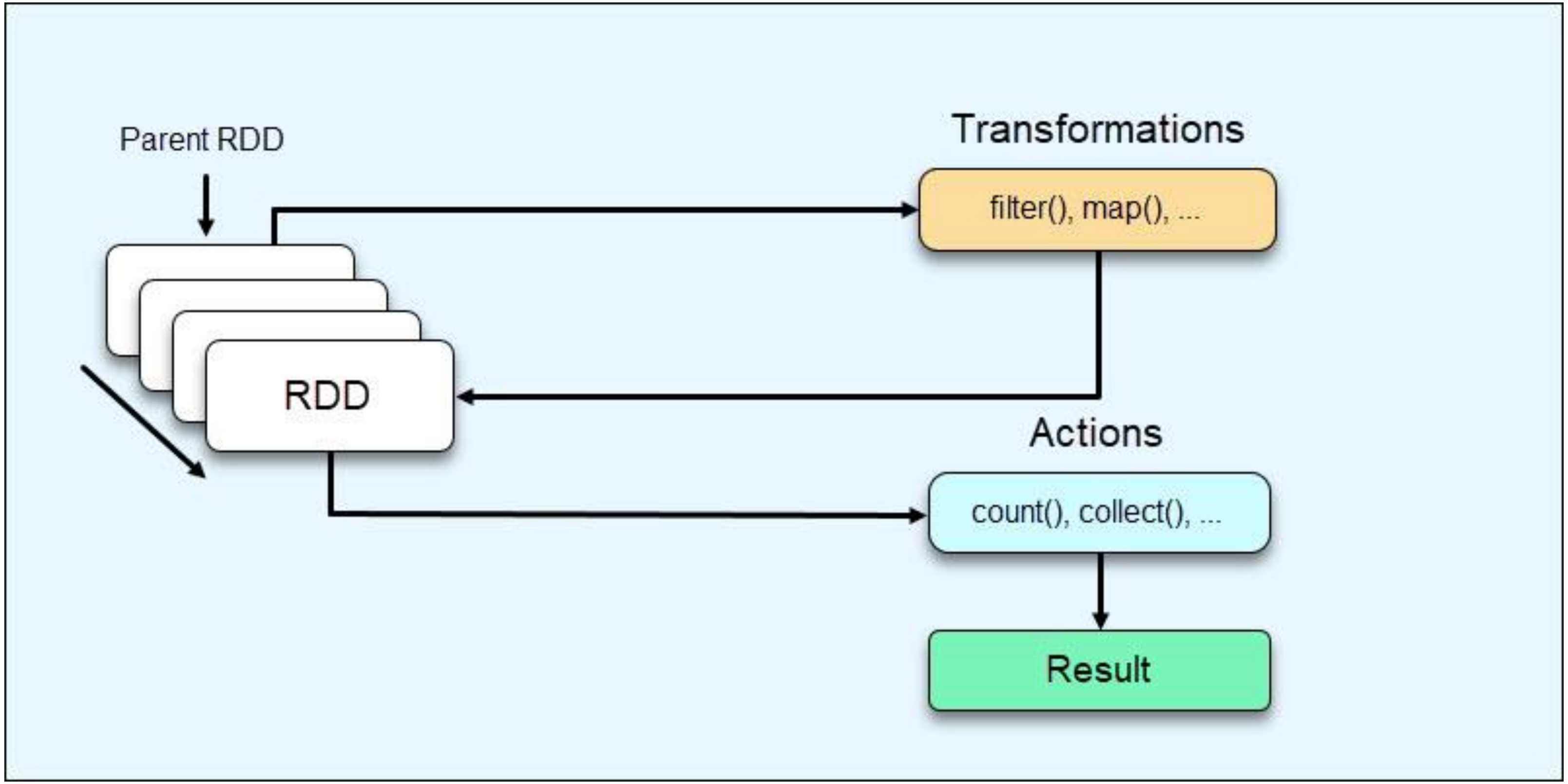
1. Transformations:

Transformations are the processes that you perform on an RDD to get a result which is also an RDD. The example would be applying functions such as filter(), union(), map(), flatMap(), distinct(), reduceByKey(), mapPartitions(), sortBy() that would create an another resultant RDD. Lazy evaluation is applied in the creation of RDD.

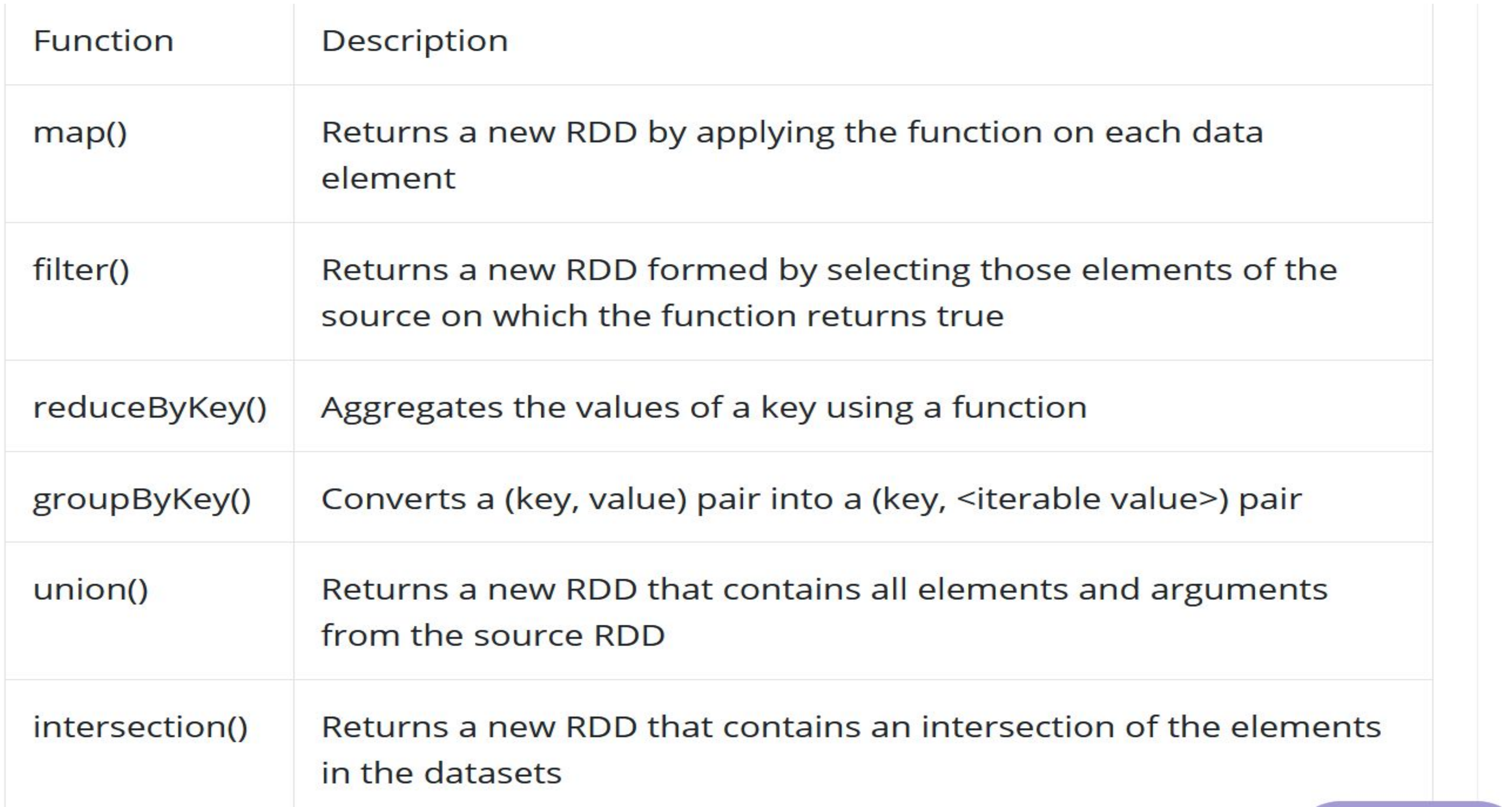
2. Actions:

Actions are operations that do not result in RDD creation and provide some other value. Actions return results to the driver program or write it in a storage and kick off a computation.examples are count(), first(), collect(), take(), countByKey(), collectAsMap(), and reduce().

Transformations will always return RDD whereas actions return some other data type.

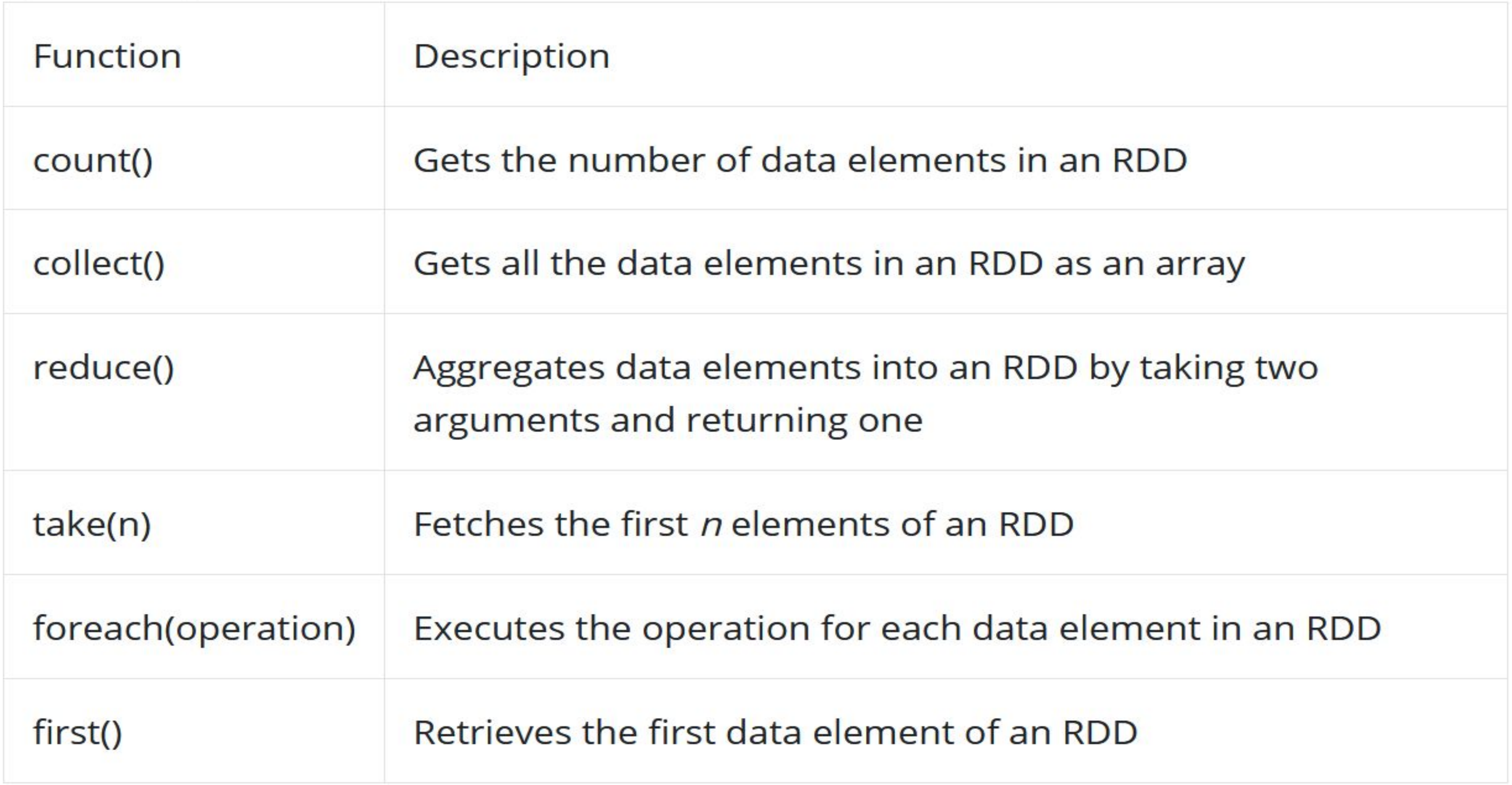

Parent RDD
Transformations
filter(), map(), ...
RDD
Actions
count(), collect(), ...
Result

Transformation Operations



Function	Description
map()	Returns a new RDD by applying the function on each data element
filter()	Returns a new RDD formed by selecting those elements of the source on which the function returns true
reduceByKey()	Aggregates the values of a key using a function
groupByKey()	Converts a (key, value) pair into a (key, <iterable value>) pair
union()	Returns a new RDD that contains all elements and arguments from the source RDD
intersection()	Returns a new RDD that contains an intersection of the elements in the datasets

Actions used in Spark



Function	Description
count()	Gets the number of data elements in an RDD
collect()	Gets all the data elements in an RDD as an array
reduce()	Aggregates data elements into an RDD by taking two arguments and returning one
take(n)	Fetches the first *n* elements of an RDD
foreach(operation)	Executes the operation for each data element in an RDD
first()	Retrieves the first data element of an RDD

What is Spark SQL DataFrame?

DataFrame appeared in Spark Release 1.3.0. We can term DataFrame as Dataset organized into named columns. DataFrames are similar to the table in a relational database or data frame in R /Python. It can be said as a relational table with good optimization technique.

The idea behind DataFrame is it allows processing of a large amount of structured data. DataFrame contains rows with Schema. The **schema** is the illustration of the structure of data.

DataFrame in Apache Spark prevails over RDD but contains the features of RDD as well. The features common to RDD and DataFrame are **immutability**, **in-memory**, resilient, distributed computing capability.

It allows the user to impose the structure onto a distributed collection of data. Thus provides higher level abstraction.

We can build DataFrame from different data sources. For Example structured data file, tables in Hive, external databases or existing RDDs. The Application Programming Interface (APIs) of DataFrame is available in various languages. Examples include Scala, Java, Python, and R.

Both in Scala and Java, we represent DataFrame as Dataset of rows. In the Scala API, DataFrames are type alias of Dataset[Row]. In Java API, the user uses Dataset<Row> to represent a DataFrame.

Why DataFrame?

DataFrame is one step ahead of **RDD.** Since it provides memory management and optimized execution plan.

a. Custom Memory Management: This is also known as Project **Tungsten.** A lot of memory is saved as the data is stored in off-heap memory in binary format. Apart from this, there is no Garbage Collection overhead. Expensive Java serialization is also avoided. Since the data is stored in binary format and the schema of memory is known.

b. Optimized Execution plan: This is also known as the **query optimizer**. Using this, an optimized execution plan is created for the execution of a query. Once the optimized plan is created final execution takes place on RDDs of Spark.

When to Use Dataframes?

Spark Dataframes are useful in the following scenarios:

- If the data is structured or semi-structured and you want high-level abstractions, Dataframe provides a schema for such data.
- If you want to store one-dimensional or multidimensional data matrices in tabular form.
- If high-level processing is required in datasets, Dataframe provides high-level functions and ease to use.

Features of Apache Spark DataFrame

Some of the limitations of Spark RDD were-

- It does not have any built-in optimization engine.
- There is no provision to handle structured data.

Thus, to overcome these limitations the picture of DataFrame came into existence. Some of the key features of DataFrame in Spark are:
i. DataFrame is a distributed collection of data organized in named column. It is equivalent to the table in RDBMS.

ii. It can deal with both structured and unstructured data formats. For Example Avro, CSV, elastic search, and Cassandra. It also deals with storage systems HDFS, HIVE tables, MySQL, etc.

iii. Catalyst supports optimization. It has general libraries to represent trees. DataFrame uses **Catalyst tree transformation** in four phases:

- Analyze logical plan to solve references
- Logical plan optimization
- Physical planning
- Code generation to compile part of a query to Java bytecode.

iv. The DataFrame API's are available in various programming languages. For example Java, Scala, Python, and R.
v. It provides Hive compatibility. We can run unmodified Hive queries on existing Hive warehouse.
vi. It can scale from kilobytes of data on the single laptop to petabytes of data on a large cluster.
vii. DataFrame provides easy integration with Big data tools and framework via **Spark core**.

Creating DataFrames in Apache Spark

To all the functionality of Spark, **SparkSession** class is the entry point. For the creation of basic SparkSession just use

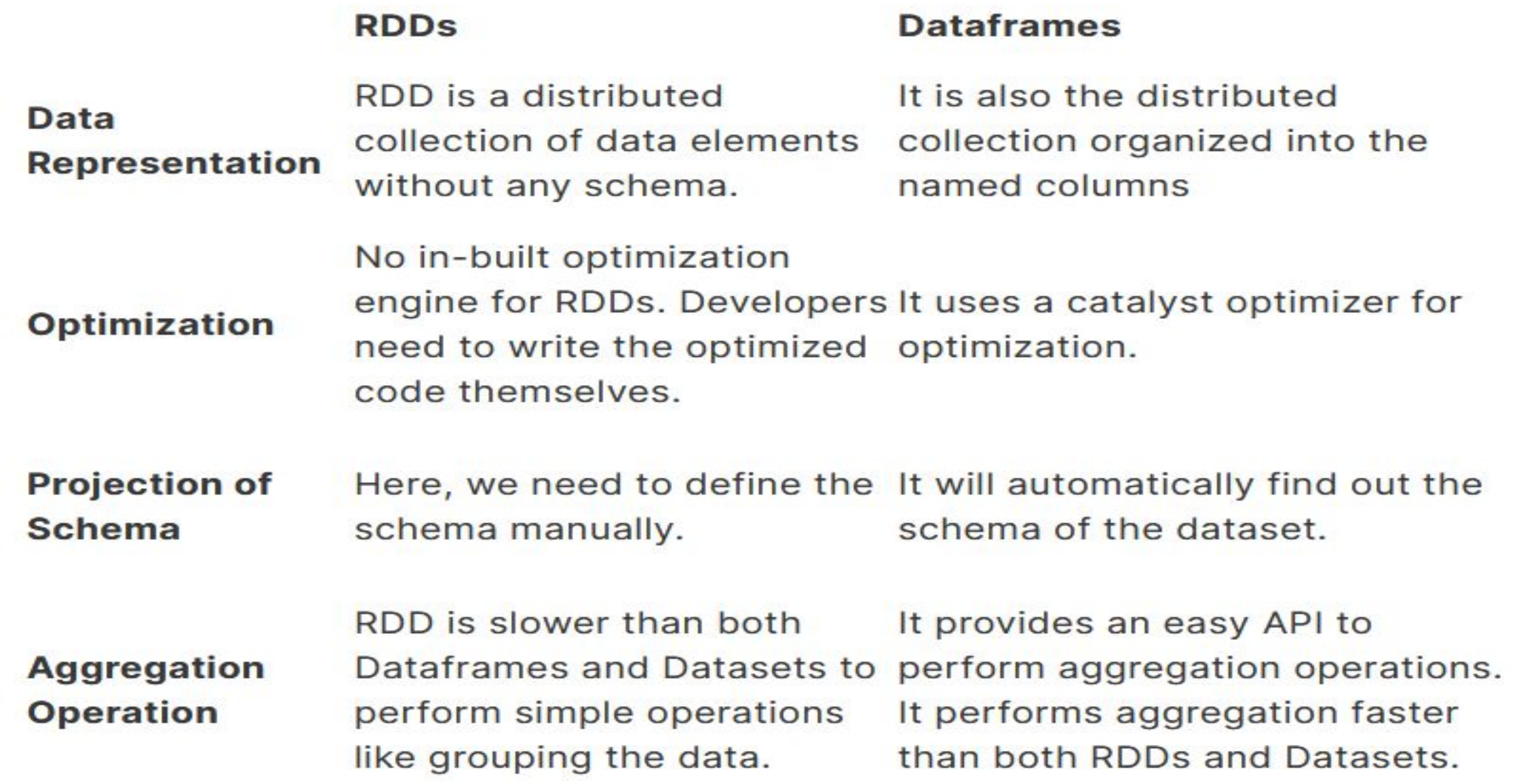
SparkSession.builder()

Using Spark Session, an application can create DataFrame from an existing RDD, Hive table or from Spark data sources. Spark SQL can operate on the variety of data sources using DataFrame interface. Using Spark SQL DataFrame we can create a temporary view. In the temporary view of dataframe, we can run the SQL query on the data.

Limitations of DataFrame in Spark

- Spark SQL DataFrame API does not have provision for **compile time type safety**. So, if the structure is unknown, we cannot manipulate the data.
- Once we convert the domain object into data frame, the regeneration of domain object is not possible.

RDD vs Dataframe



	RDDs	Dataframes
Data Representation	RDD is a distributed collection of data elements without any schema.	It is also the distributed collection organized into the named columns
Optimization	No in-built optimization engine for RDDs. Developers need to write the optimized code themselves.	It uses a catalyst optimizer for optimization.
Projection of Schema	Here, we need to define the schema manually.	It will automatically find out the schema of the dataset.
Aggregation Operation	RDD is slower than both Dataframes and Datasets to perform simple operations like grouping the data.	It provides an easy API to perform aggregation operations. It performs aggregation faster than both RDDs and Datasets.

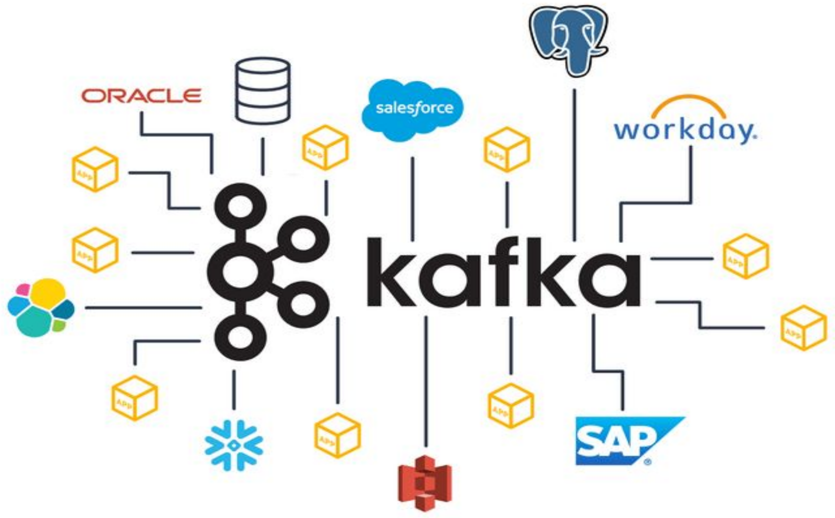
What is Kafka?

Apache Kafka is a distributed data store optimized for ingesting and processing streaming data in real-time. Streaming data is data that is continuously generated by thousands of data sources, which typically send the data records in simultaneously. A streaming platform needs to handle this constant influx of data, and process the data sequentially and incrementally.

Kafka provides three main functions to its users:

- Publish and subscribe to streams of records
- Effectively store streams of records in the order in which records were generated
- Process streams of records in real time

Kafka is primarily used to build real-time streaming data pipelines and applications that adapt to the data streams. It combines messaging, storage, and stream processing to allow storage and analysis of both historical and real-time data.



What is Kafka used for?

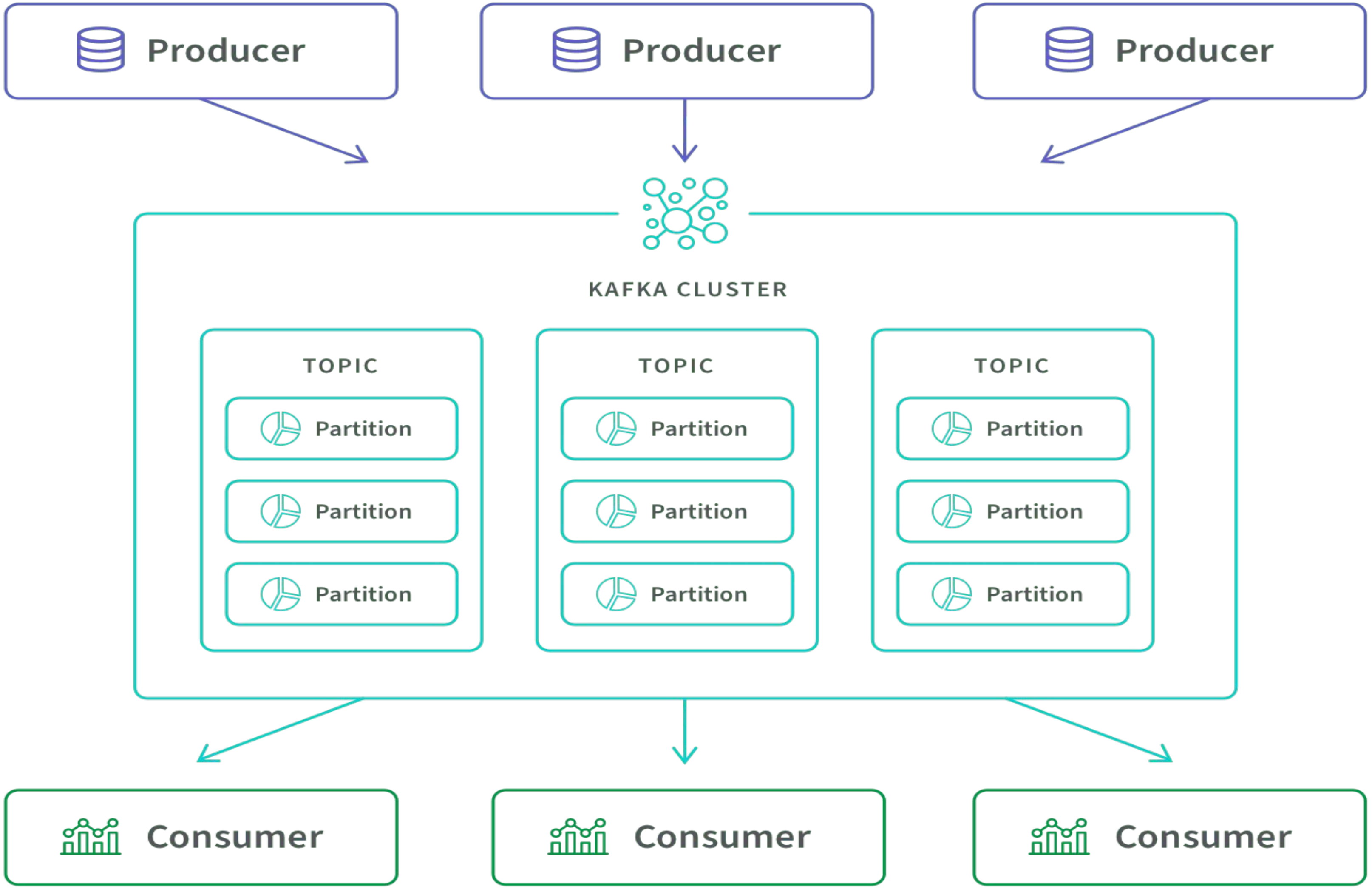
Kafka is used to build real-time streaming data pipelines and real-time streaming applications. A data pipeline reliably processes and moves data from one system to another, and a streaming application is an application that consumes streams of data. For example, if you want to create a data pipeline that takes in user activity data to track how people use your website in real-time, Kafka would be used to ingest and store streaming data while serving reads for the applications powering the data pipeline. Kafka is also often used as a message broker solution, which is a platform that processes and mediates communication between two applications.

How Apache Kafka Works

At a high level, Apache Kafka allows you to publish and subscribe to streams of records, store these streams in the order they were created, and process these streams in real time.

Running on a horizontally scalable cluster of commodity servers, Apache Kafka ingests real time data from multiple "producer" systems and applications—such as logging systems, monitoring systems, sensors, and IoT applications—and at very low latency makes the data available to multiple "consumer" systems and applications such as real time analytics.

Consumers can range from analytics platforms to applications that rely on real-time data processing. Examples include logistics or location-based micromarketing applications.


Producer
Producer
Producer
KAFKA CLUSTER
TOPIC
Partition
Partition
Partition
TOPIC
Partition
Partition
Partition
TOPIC
Partition
Partition
Partition
Consumer
Consumer
Consumer

Here we define the terms shown above:

Producer: Client application that push events into topics

Cluster: One or more servers (called brokers) running Apache Kafka.

Topic: The method to categorize and durably store events. There are two types of topics: compacted and regular. Records in compacted topics do not expire based on time or space bounds. Newer topic messages update older messages that possess the same key and Apache Kafka does not delete the latest message unless deleted by the user. For regular topics, records can be configured to expire, deleting old data to free storage space.

Partition: The mechanism to distribute data across multiple storage servers (brokers). Messages are indexed and stored together with a timestamp and ordered by the position of the message within a partition. Partitions are distributed across a node cluster and are replicated to multiple servers to ensure that Apache Kafka delivers message streams in a fault-tolerant manner.

Consumers: Client applications which read and process the events from partitions. The Apache Kafka Streams API allows writing Java applications which pull data from Topics and write results back to Apache Kafka. External stream processing systems such as Apache Spark, Apache Apex, Apache Flink, Apache NiFi and Apache Storm can also be applied to these message streams

What are the benefits of Kafka's approach?

Scalable:

Kafka’s partitioned log model allows data to be distributed across multiple servers, making it scalable beyond what would fit on a single server.

Fast :

Kafka decouples data streams so there is very low latency, making it extremely fast.

Durable:

Partitions are distributed and replicated across many servers, and the data is all written to disk. This helps protect against server failure, making the data very fault-tolerant and durable.