VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMMING INTEGRATION PROJECT (CO3127)

Report

Apache Kafka and Apache Spark

Instructor: **Assoc. Prof. Thoại Nam**
Đinh Phúc Hưng
La Quốc Nhựt Huân

Student: Trần Tuấn Kiệt - 2252410
Phạm Duy Tường Phước - 2252662
Phan Hồng Quân - 2252685

HO CHI MINH CITY, JANUARY 2025

Contents

List of Figures

CHAPTER 1

APACHE KAFKA

1 Introduction to Apache Kafka

1.1 Publish/Subscribe Messaging

To understand Apache Kafka, it is first necessary to grasp the concept of publish/subscribe messaging and its significance. Publish/subscribe messaging is a communication pattern where the sender (publisher) of information (message) does not send it to a specific receiver. Instead, the publisher categorizes the message in some way, and the receiver (subscriber) opts to receive messages from certain categories. These systems typically rely on a broker, which serves as a central hub where messages are published, streamlining the process.

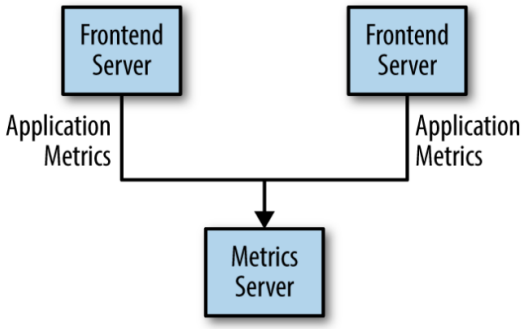


Figure 1: A solitary metrics publisher with a direct connection [16]

Many publish/subscribe use cases start with a simple setup, such as a message queue or interprocess communication channel. Initially, an application may send monitoring data directly to a dashboard for displaying metrics. However, as requirements evolve, such as the need for long-term analysis, additional services are introduced, and the application is modified to send data to multiple systems. With more applications generating metrics and features like active polling added, the architecture becomes increasingly complex and difficult to manage.

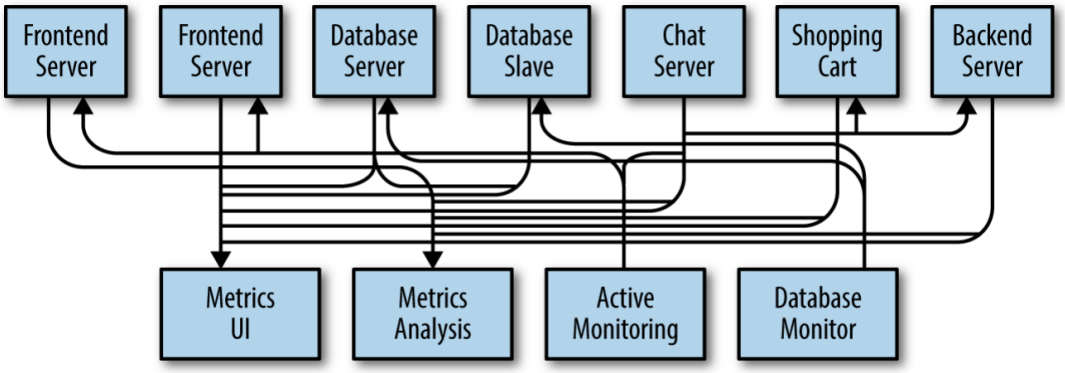


Figure 2: Numerous metrics publishers rely on direct connections [16]

Simplification often involves centralizing metrics collection and providing a single interface

for querying data. Similar challenges arise with managing logs, tracking user behavior, and supporting analytics, leading to separate pub/sub systems that reduce complexity but result in duplication. Maintaining multiple systems introduces inefficiencies, and a centralized platform for scalable, unified data publishing is a more sustainable solution.

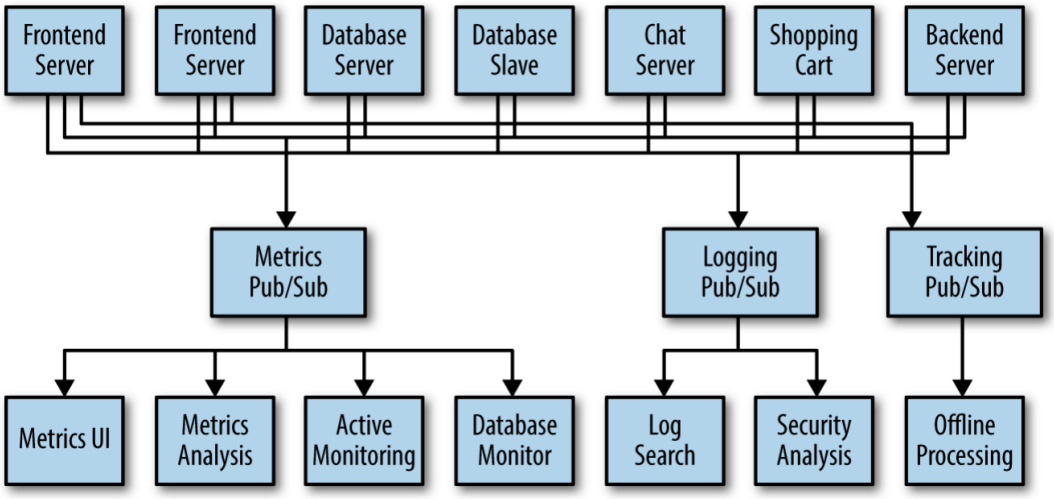


Figure 3: Several publish/subscribe systems [16]

Apache Kafka serves as a publish/subscribe messaging system specifically designed to address this challenge. Often referred to as a "distributed commit log" or more recently as a "distributed streaming platform", Kafka functions similarly to the commit log of a filesystem or database. A commit log ensures a durable record of all transactions, allowing them to be replayed to reliably recreate a system's state. In the same way, Kafka stores data in a durable, ordered manner, enabling deterministic reads. Furthermore, Kafka distributes this data across the system, providing resilience against failures and offering significant scalability for enhanced performance.

1.2 Apache Kafka Components

1.2.1 Messages and Batches

In Kafka, the fundamental unit of data is called a message. For those familiar with databases, a message can be likened to a row or record. From Kafka's perspective, a message is simply a byte array, meaning it has no predefined format or meaning. Messages can include an optional piece of metadata known as a key, which is also a byte array and similarly lacks any inherent meaning to Kafka. Keys are primarily used for controlling how messages are distributed across partitions. A common approach is to compute a consistent hash of the key and use the hash modulo the total number of partitions in the topic to determine the partition number. This ensures that messages with the same key are always routed to the same partition.

To optimize efficiency, Kafka writes messages in batches rather than individually. A batch is a collection of messages destined for the same topic and partition. Sending each message individually over the network would introduce significant overhead, so grouping messages into batches reduces this inefficiency. However, batching involves a tradeoff between latency and throughput: larger batches increase the number of messages processed in a given time but delay

the propagation of individual messages. Additionally, batches are often compressed to improve data transfer and storage efficiency, although this requires additional processing power.

1.2.2 Schemas

Although messages are opaque byte arrays to Kafka itself, it is recommended to impose additional structure or schema on the message content to ensure it can be easily interpreted. Various schema options are available depending on application requirements. Simple formats like JSON (JavaScript Object Notation) and XML (Extensible Markup Language) are human-readable and easy to implement but lack advanced features such as robust type handling and compatibility across schema versions. A popular choice among Kafka developers is Apache Avro, a serialization framework initially created for Hadoop. Avro offers a compact serialization format, schemas that are independent of message payloads (eliminating the need for code regeneration when schemas change), and strong data typing with support for schema evolution, enabling both backward and forward compatibility.

Maintaining a consistent data format is critical in Kafka, as it decouples the processes of writing and reading messages. Tight coupling requires subscriber applications to be updated to handle both the old and new data formats before publisher applications can adopt the new format. Using well-defined schemas stored in a centralized repository eliminates this need for coordination, allowing messages in Kafka to be interpreted seamlessly.

1.2.3 Topics and Partitions

In Kafka, messages are organized into topics, which are comparable to a database table or a folder in a filesystem. Topics are further divided into multiple partitions, with each partition functioning as an individual log. Messages are written to partitions in an append-only manner and are read sequentially from beginning to end. Since topics usually consist of multiple partitions, message time-ordering is guaranteed only within a single partition, not across the entire topic. Partitions also enable Kafka to achieve both redundancy and scalability. By hosting each partition on a separate server, a single topic can be distributed horizontally across multiple servers, allowing performance levels far beyond the capacity of a single server.

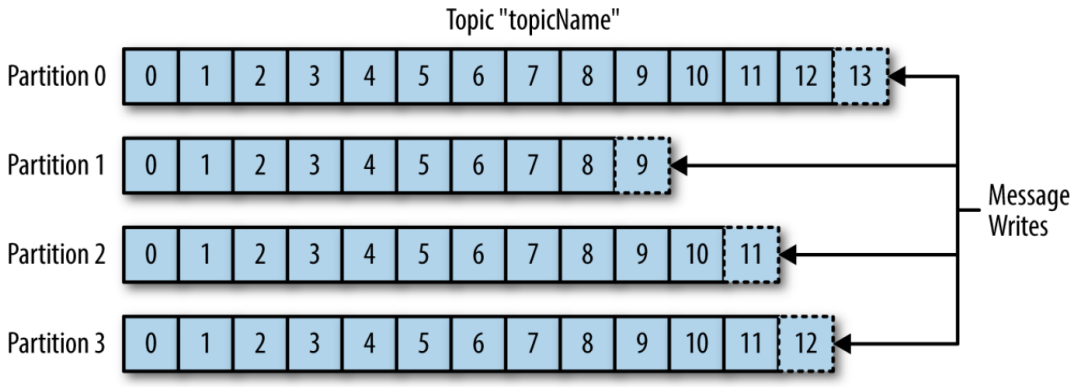


Figure 4: Representation of a topic with multiple partitions [16]

The term "stream" is frequently used in the context of systems like Kafka. Typically, a stream refers to a single topic of data, regardless of how many partitions it contains, representing a

continuous flow of data from producers to consumers. This terminology is particularly common in discussions of stream processing, where frameworks such as Kafka Streams, Apache Samza, and Storm process messages in real time. Stream processing contrasts with offline frameworks like Hadoop, which are designed to handle bulk data at a later time.

1.2.4 Producers and Consumers

Kafka clients are the users of the system, categorized into two primary types: producers and consumers. In addition to these, Kafka offers advanced client APIs, such as the Kafka Connect API for data integration and Kafka Streams for stream processing. These advanced clients are built upon producers and consumers, offering higher-level functionalities.

Producers are responsible for creating new messages and are analogous to publishers or writers in other publish/subscribe systems. Typically, messages are produced to a specific topic. By default, the producer distributes messages evenly across all partitions within a topic. However, in certain scenarios, producers may direct messages to specific partitions. This is usually achieved by utilizing the message key along with a partitioner, which hashes the key and maps the message to a corresponding partition. This ensures that all messages with the same key are consistently written to the same partition. Alternatively, producers can implement a custom partitioner to map messages to partitions based on specific business rules.

Consumers in Kafka are responsible for reading messages and are comparable to subscribers or readers in other publish/subscribe systems. A consumer subscribes to one or more topics and processes messages in the order they were produced. To track its progress, the consumer maintains the offset of messages it has already consumed. The offset, a continuously increasing integer value added by Kafka to each message, is unique within a given partition. By storing the offset of the last consumed message for each partition, either in Zookeeper or within Kafka itself, a consumer can pause and resume processing without losing its place.

Consumers operate as part of a consumer group, which is a collection of one or more consumers working together to consume messages from a topic. Each partition within the topic is assigned to only one consumer in the group, ensuring no overlap in processing. For example, in a setup with three consumers in a single group, one consumer might process two partitions while the other two handle one partition each. This assignment of partitions to consumers is referred to as partition ownership.

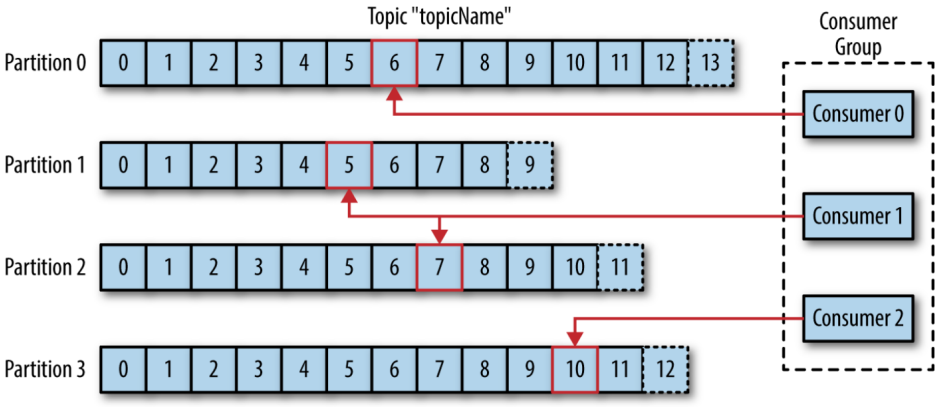


Figure 5: A consumer group reading from a topic [16]

This design enables consumers to scale horizontally, allowing them to efficiently process topics with large volumes of messages. Furthermore, if a consumer fails, the group automatically rebalances the partitions among the remaining members to ensure continuity of message consumption.

1.2.5 Brokers and Clusters

A single Kafka server is referred to as a broker. The broker handles several key responsibilities, including receiving messages from producers, assigning offsets to those messages, and storing them on disk. It also serves consumers by responding to fetch requests for partitions and providing messages that have been committed to disk. Depending on the hardware and its performance capabilities, a single broker can manage thousands of partitions and process millions of messages per second.

Kafka brokers are designed to function within a cluster. In a cluster, one broker is automatically elected as the cluster controller, which oversees administrative tasks such as assigning partitions to brokers and monitoring for broker failures. Each partition is managed by a single broker, known as the partition leader. A partition can also be replicated across multiple brokers to ensure redundancy. This replication allows another broker to assume leadership of the partition if the leader fails. However, producers and consumers working with a partition must always connect to its leader.

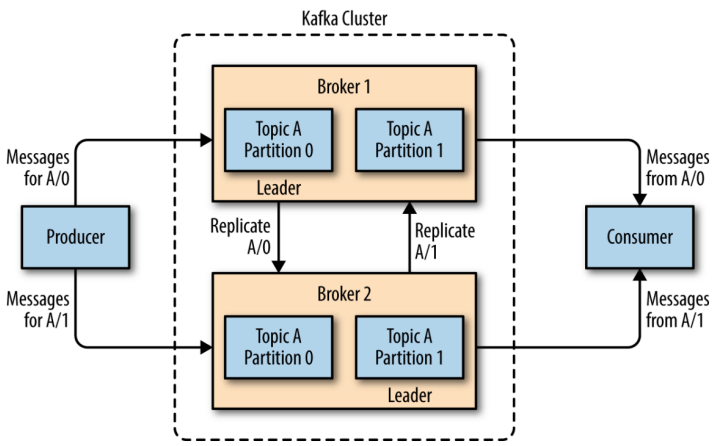


Figure 6: Replication of partitions in a cluster [16]

A key feature of Apache Kafka is its retention capability, which ensures the durable storage of messages for a specified period. Kafka brokers are configured with a default retention policy for topics, either based on a time limit (e.g., 7 days) or a size limit (e.g., 1 GB). Once these thresholds are reached, older messages are expired and deleted, ensuring that the retention settings define the minimum amount of data available at any given time. Individual topics can also be configured with custom retention settings based on their specific use case. For instance, a tracking topic may retain messages for several days, while application metrics might only be stored for a few hours. Additionally, topics can be configured for log compaction, where Kafka retains only the most recent message for each specific key. This is particularly useful for changelog-style data, where only the latest update is relevant.

1.2.6 Multiple Clusters

As Kafka deployments expand, it is often beneficial to utilize multiple clusters. This approach can serve several purposes, including segregating different types of data, meeting security requirements through isolation, and supporting multiple datacenters for disaster recovery. In scenarios involving multiple datacenters, it is frequently necessary to copy messages between them. This ensures that online applications can access user activity across all locations. For instance, if a user updates their public profile information, that change must be reflected in search results regardless of the datacenter being accessed. Similarly, monitoring data from various sites can be centralized in one location for analysis and alerting.

However, Kafka's native replication mechanisms are designed to operate only within a single cluster and do not support replication across clusters. To address this, the Kafka project provides a tool called MirrorMaker. MirrorMaker functions as a combination of a Kafka consumer and producer connected via a queue. It consumes messages from one Kafka cluster and produces them for another. For example, MirrorMaker can aggregate messages from multiple local clusters into a central cluster and replicate that central cluster across other datacenters.

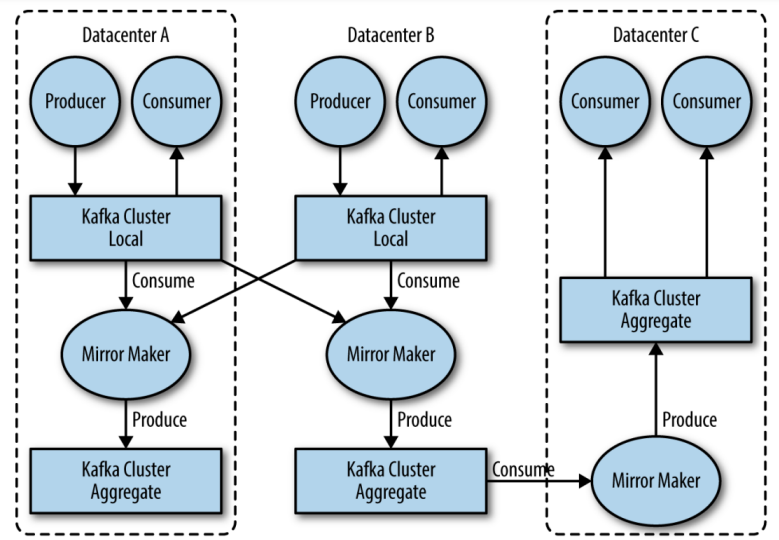


Figure 7: Multiple datacenter architecture [16]

1.3 Advantages of Apache Kafka

With numerous options available for publish/subscribe messaging systems, Apache Kafka stands out due to its unique combination of features. A comparission

1.3.1 Multiple Producers

Kafka efficiently handles multiple producers, whether they are working with multiple topics or writing to the same topic. This capability makes Kafka particularly well-suited for aggregating data from various frontend systems and ensuring consistency. For instance, a website delivering content through multiple microservices can utilize a single topic for page views, with all services writing to it in a standardized format. Consumer applications can then access a unified stream

of page views for all site applications, eliminating the need to coordinate consumption across separate topics for each service.

1.3.2 Multiple Consumers

Kafka is designed to support multiple consumers reading from the same stream of messages without interfering with one another. This contrasts with many traditional queuing systems, where a message becomes unavailable to other clients once it is consumed. Kafka consumers can also form a group to share a stream, ensuring that each message in the stream is processed by only one member of the group.

1.3.3 Disk-Based Retention

Kafka not only supports multiple consumers but also offers durable message retention, in the meaning that consumers do not always need to work in real time. Messages are committed to disk and stored according to configurable retention policies, which can be customized on a per-topic basis to meet specific consumer requirements. Durable retention ensures that even if a consumer falls behind due to slow processing or a traffic spike, no data is lost. This feature also allows for maintenance on consumer applications, enabling them to go offline temporarily without risking message loss or causing backups on the producer side. Consumers can be stopped, and Kafka will retain the messages, allowing them to resume processing from where they left off without any data loss.

1.3.4 Scalability

Kafka's scalability allows it to efficiently handle any volume of data. Users can begin with a single broker for a proof of concept, expand to a small development cluster of three brokers, and eventually scale to a production cluster consisting of tens or even hundreds of brokers as data volumes increase. Cluster expansions can be carried out while the system remains online, ensuring uninterrupted availability. Additionally, a multi-broker cluster is resilient to individual broker failures, maintaining service for clients. For greater fault tolerance, clusters can be configured with higher replication factors to withstand multiple simultaneous failures.

1.3.5 High Performance

All these features combine to make Apache Kafka a highly efficient publish/subscribe messaging system, capable of delivering excellent performance even under heavy load. Producers, consumers, and brokers can be scaled out to effortlessly handle large message streams while maintaining subsecond message latency from production to consumer availability.

2 Kafka Streams API

2.1 Introduction to Kafka Streams API

The Kafka Streams API is a powerful and lightweight library that enables developers to build real-time, distributed stream processing applications on data stored in Apache Kafka. Unlike traditional stream processing frameworks that require separate clusters or resource managers, Kafka Streams integrates directly with standard Java applications. This integration eliminates the need for additional infrastructure and simplifies the development and deployment of stream processing pipelines. By leveraging Kafka Streams, developers can create highly scalable, fault-tolerant, and efficient data processing systems.

The library provides an intuitive API that supports real-time transformations, aggregations, and event-driven computations. It enables applications to process data as it flows through Kafka topics, applying operations like filtering, joining, and aggregating in real time. These capabilities make Kafka Streams an indispensable tool for organizations that rely on timely insights and event-driven architectures.

The picture below shows the anatomy of an application that uses the Kafka Streams library.

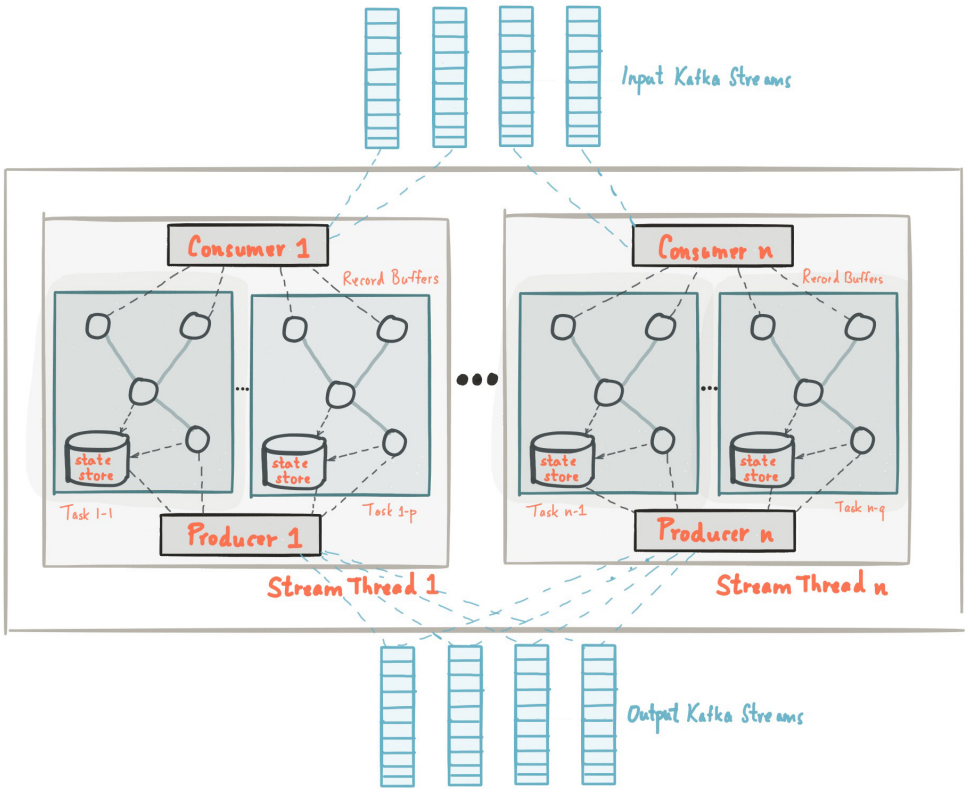


Figure 8: Anatomy of a Kafka Streams Application [15].

2.2 Kafka Streams Architecture

2.2.1 Core Components

The Kafka Streams library enables distributed, scalable, and fault-tolerant stream processing directly on data stored in Apache Kafka. Its architecture is built on the following foundational components:

2.2.1.a KStream (Key-Value Streams)

- A **KStream** represents a continuous flow of records, where each record is a key-value pair [18].
- It is ideal for real-time processing of unbounded streams of data, such as filtering or transforming incoming records.
- Example: In a financial application, a KStream could represent transaction events where each record corresponds to a transaction.

2.2.1.b KTable (Key-Value Tables)

- A **KTable** represents a view of data as a changelog stream, essentially mapping each key to its latest value [17].
- It is designed for stateful processing and is ideal for scenarios where updates overwrite previous values, such as tracking account balances.
- Internally, KTables are backed by **state stores** to maintain the most recent key-value pair, ensuring efficient updates.
- Example: A customer profile store in an e-commerce system.

2.2.1.c GlobalKTable

- A **GlobalKTable** is similar to a KTable but is replicated to all instances of the Kafka Streams application [19].
- It is optimized for use cases where data joins require looking up the state of the entire dataset.
- Example: Joining customer transaction data (KStream) with a global product catalog (GlobalKTable).

2.2.1.d Processor Topology

- At the heart of Kafka Streams is the **processor topology**, a directed acyclic graph (DAG) where:
  - **Source nodes** ingest records from Kafka topics.
  - **Processor nodes** perform computations, such as filtering, mapping, and aggregation.
  - **Sink nodes** write results to output Kafka topics.
- The topology is defined programmatically using the Kafka Streams DSL or the Processor API.

2.2.1.e State Stores

- **State stores** are local storage mechanisms embedded within a Kafka Streams application for maintaining intermediate states required for stateful operations like aggregations or joins.
- Backed by storage engines such as **RocksDB**, state stores ensure fault tolerance by persisting state and using **changelog topics** to replicate data across instances.
- Kafka Streams automatically manages state store updates and recovery, ensuring consistency.

2.2.2 Distributed Processing

Kafka Streams' distributed nature ensures robust and scalable stream processing. It uses **partitioning and locality-based task assignment** to align processing tasks with topic partitions, reducing network overhead and ensuring data locality. As a result, the system minimizes latency and maximizes efficiency.

The platform supports **horizontal scalability** by dynamically redistributing tasks across instances whenever nodes are added or removed. This automatic rebalancing ensures consistent processing of all topic partitions and robust fault tolerance, making Kafka Streams highly resilient for distributed applications.

2.3 Programming Model

2.3.1 DSL (Domain-Specific Language)

Kafka Streams provides a high-level **Domain-Specific Language (DSL)** for defining stream-processing workflows. Using this DSL, developers can perform common transformations like `map`, `filter`, and `groupBy` with minimal effort. The DSL also supports the concept of **stream-table duality**, enabling seamless transitions between streams and tables. This allows developers to perform stateful operations, such as aggregations and joins, in a simple and declarative manner.

2.3.2 Processor API

For advanced use cases, Kafka Streams offers the **Processor API**, which provides low-level control over data processing. This API enables developers to define custom processors and control how data is transformed and routed. By exposing metadata and state, it is ideal for integrating external systems or applying custom logic not available in the DSL. For example, a custom processor might enrich a stream of transactions with external customer data, enabling more complex operations.

2.4 Key Features

2.4.1 Fault Tolerance

Kafka Streams ensures high reliability through **fault tolerance** mechanisms. State information is stored in Kafka changelog topics, which provide a durable backup of intermediate states. In case of a failure, the application can recover its state and resume processing seamlessly, making it a dependable choice for critical applications.

2.4.2 Windowing

Kafka Streams supports **windowing**, a powerful feature for time-based data aggregations. Developers can choose from various window types, including tumbling, hopping, and sliding windows, to fit their use cases. For instance, tumbling windows can be used to calculate hourly sales totals, while sliding windows are ideal for monitoring performance metrics in real time.

2.4.3 Stateful Operations

The platform's ability to handle **stateful operations** is one of its standout features. By leveraging embedded state stores, Kafka Streams allows applications to maintain intermediate results efficiently. These stores are backed by changelog topics, ensuring durability and fault tolerance for stateful computations such as aggregations and joins.

2.4.4 Stream-Thread Model

Kafka Streams employs a **stream-thread model** to enable efficient parallel processing of data streams. Each stream thread operates independently and is responsible for processing data from one or more partitions. This architecture allows applications to scale vertically by increasing the thread count or horizontally by deploying additional instances. The model ensures optimal resource utilization and supports high-throughput processing.

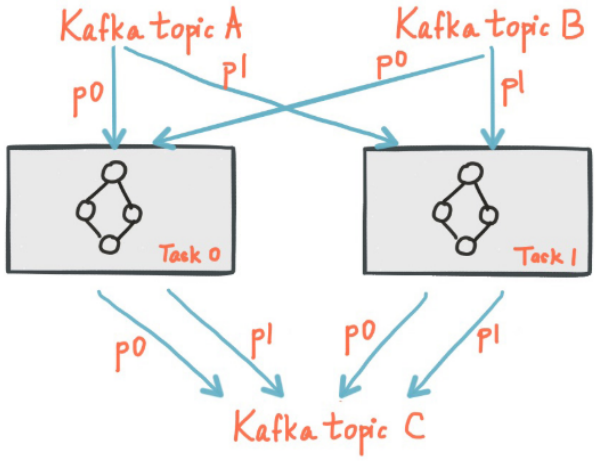


Figure 9: Each thread processes independent partitions [16].

2.4.5 Stream-Table Duality

Kafka Streams embraces the concept of **stream-table duality**, which allows seamless transitions between streams and tables. A stream (continuous flow of records) can be converted into a table (a view of aggregated state), and vice versa. This duality simplifies operations like joining streams with tables, performing aggregations, or maintaining real-time materialized views. It forms the theoretical backbone for many advanced stream processing applications.

2.5 Integration with Kafka Ecosystem

2.5.1 Kafka Connect

Kafka Streams integrates seamlessly with **Kafka Connect**, enabling real-time data ingestion and export. This allows developers to bring data from external sources, such as relational databases or cloud storage, into Kafka topics and process it with Kafka Streams. Similarly, processed data can be exported to downstream systems like dashboards or data warehouses, creating an end-to-end data pipeline.

2.5.2 Schema Registry

The **Schema Registry** ensures consistent data serialization and deserialization, supporting formats like Avro and Protobuf. With schema evolution capabilities, developers can update schemas without breaking existing applications, making Kafka Streams suitable for long-term projects requiring stability and compatibility.

2.6 Use Cases

2.6.1 Real-Time Analytics

Kafka Streams is widely used for **real-time analytics**, enabling businesses to gain immediate insights from their data. For example, an e-commerce platform might analyze user activity streams to identify trends, optimize recommendations, or personalize the shopping experience.

2.6.2 Event-Driven Applications

The platform supports **event-driven applications**, where real-time event streams trigger actions or workflows. Use cases include fraud detection in financial transactions, where Kafka Streams monitors patterns and flags anomalies in real time, or alerting systems that notify operators of unusual system behavior.

2.6.3 ETL Pipelines

Kafka Streams is a powerful tool for building **ETL pipelines**, transforming raw data streams into enriched datasets for downstream systems. For example, it can combine sales data from multiple sources with customer profiles to create structured datasets for analytics. Its ability to handle high-throughput data streams ensures efficiency and scalability.

2.7 Advantages Over Alternatives

2.7.1 Integration with Kafka

Kafka Streams' seamless **integration with Kafka** makes it a unified solution for data ingestion, processing, and publishing. Unlike standalone frameworks, it eliminates the need for a separate cluster, simplifying infrastructure management and reducing operational overhead.

2.7.2 Ease of Development

The platform's high-level DSL simplifies the development process, allowing developers to focus on business logic rather than boilerplate code. At the same time, the Processor API provides flexibility for custom use cases, making Kafka Streams suitable for a wide range of applications.

2.7.3 Robustness

Kafka Streams is known for its **robustness**, with features like built-in fault tolerance, state management, and scalability. These capabilities make it a reliable choice for building distributed, real-time stream-processing applications, enabling businesses to harness the power of streaming data effectively.

3 KStream Class in Kafka Streams [18]

3.1 Definition

The `KStream` interface in Kafka Streams represents an abstraction for a stream of records, each expressed as a key-value pair. Each record is treated as an independent event in the real world. For example, if a user purchases two items, the records might appear as `<User:Item1>` and `<User:Item2>` in the stream.

Key characteristics include:

- A `KStream` can be created from one or more Kafka topics or from the transformation of another stream.
- It supports various operations, including record-by-record transformations, joins with other streams or tables, and aggregations into a `KTable`.
- A `KStream` can be created from one or more Kafka topics or from the transformation of another stream.

The `KStream` interface also supports seamless integration with the Processor API, enabling custom processing through methods such as `process`, `transform`, and `transformValues`.

3.2 Constructor

A `KStream` can be initialized in various ways using the `StreamsBuilder` class. These methods allow developers to create streams tailored to specific use cases and configurations. Here are the primary approaches to initialize a `KStream`:

3.2.1 Stream from a Single Topic

You can create a `KStream` from a single topic by specifying the topic name and an instance of `Consumed`, which defines optional parameters like deserializers, timestamp extractors, and offset reset strategies:

```
<K, V> KStream<K, V> stream(String topic, Consumed<K, V> consumed);
```

3.2.1.a Parameters

- **topic:** The name of the topic (cannot be null).
- **consumed:** An instance of `Consumed` to define configurations for deserialization, timestamp extraction, and offset reset.

3.2.1.b Example

```
KStream<String, String> stream = builder.stream(
    "input-topic",
    Consumed.with(Serdes.String(), Serdes.String())
);
```

3.2.1.c Note

- The topic must be partitioned by key for key-based operations like joins or aggregations. If not, the user must repartition the data.

3.2.2 Stream from Multiple Topics

You can also create a `KStream` from multiple topics. The `stream()` method allows specifying a collection of topic names and a `Consumed` instance:

```
<K, V> KStream<K, V> stream(Collection<String> topics, Consumed<K, V> consumed);
```

3.2.2.a Parameters

- **topics:** A collection of topic names (must contain at least one name).
- **consumed:** An instance of `Consumed` for optional configurations.

3.2.2.b Example

```
List<String> topics = Arrays.asList("topic1", "topic2", "topic3");
KStream<String, String> stream = builder.stream(
    topics,
    Consumed.with(Serdes.String(), Serdes.String())
);
```

3.2.2.c Note

- When multiple topics are specified, there is no guarantee of ordering between records from different topics.
- Ensure topics are partitioned by key for key-based operations.

3.2.3 Stream from a Topic Pattern

The `stream()` method also supports defining a `KStream` based on a regular expression pattern that matches topic names:

```
<K, V> KStream<K, V> stream(Pattern topicPattern, Consumed<K, V> consumed);
```

3.2.3.a Parameters

- **topicPattern:** A regular expression pattern to match topic names.
- **consumed:** An instance of `Consumed` to define configurations for deserialization, timestamp extraction, and offset reset.

3.2.3.b Example

```
Pattern topicPattern = Pattern.compile("topic.*");
KStream<String, String> stream = builder.stream(
    topicPattern,
    Consumed.with(Serdes.String(), Serdes.String())
);
```

3.2.3.c Note

- Records from matched topics are included in the stream, but there is no ordering guarantee across topics.
- Ensure topics matched by the pattern are partitioned by key for downstream key-based operations.

These initialization methods provide flexibility to define `KStream` instances for diverse use cases, whether working with a single topic, multiple topics, or dynamically matched topics via patterns.

3.3 ETL Methods in KStream

The `KStream` interface provides a comprehensive suite of methods for performing Extract, Transform, and Load (ETL) operations, as well as debugging and logging. These methods allow developers to process and manipulate data streams effectively. Below are detailed descriptions of each category of ETL methods:

3.3.1 Extract Operations

Extract operations in `KStream` allow you to filter or branch records from a stream based on custom logic. These operations are stateless and operate record by record, providing the foundation for narrowing down or splitting the data for further processing.

It is important to note that there are no direct data extraction methods in `KStream` because the `KStream` instance automatically consumes the topic partition(s) as soon as it is initialized by the `stream()` method from the `StreamsBuilder` class. The operations in this section are applied to the records after they have already been consumed into the `KStream` instance.

Below are detailed explanations of the key extract methods:

3.3.1.a Filter Records

`filter(Predicate<? super K,? super V> predicate)`

The `filter` method creates a new `KStream` containing only those records from the original stream that satisfy the specified predicate. Records that do not match the predicate are excluded from the resulting stream. This is a stateless operation and processes each record independently.

Parameters:

- **predicate:** A filter condition applied to each record. Records that evaluate to `true` are included in the output stream.

Returns:

- A `KStream` containing only records that satisfy the given predicate.

Example: To filter records where the value contains the string `"important"`:

```
KStream<String, String> filteredStream = stream.filter(
    (key, value) -> value.contains("important")
);
```

In this example:

- Records with values containing `"important"` are retained.
- Records not meeting the condition are dropped.

3.3.1.b Exclude Records

`filterNot(Predicate<? super K,? super V> predicate)`

The `filterNot` method is the inverse of `filter`. It creates a new `KStream` containing only those records that do **not** satisfy the specified predicate. This is also a stateless operation, and each record is evaluated independently.

Parameters:

- **predicate:** A filter condition applied to each record. Records that evaluate to `false` are included in the output stream.

Returns:

- A `KStream` containing only records that do **not** satisfy the given predicate.

Example: To exclude records with null values:

```
KStream<String, String> nonNullStream = stream.filterNot(
    (key, value) -> value == null
);
```

In this example:

- Records with non-null values are retained.
- Records with null values are dropped.

3.3.1.c Branch Records

`branch(Predicate<? super K,? super V>... predicates)`

The `branch` method splits the original stream into multiple substreams based on the provided predicates. Each predicate corresponds to a substream, and a record is assigned to the first substream whose predicate evaluates to `true`. If none of the predicates match, the record is dropped. This is a stateless operation and processes records one at a time.

Parameters:

- **predicates:** An ordered list of predicates used to evaluate records. The first predicate that evaluates to `true` determines the substream to which the record is assigned.

Returns:

- An array of `KStream` objects, where each substream corresponds to one of the predicates.

Example: To branch records into three categories based on their values:

```
KStream<String, String>[] branches = stream.branch(
    (key, value) -> value.startsWith("A"), // Substream 0: Values starting with "A"
    (key, value) -> value.startsWith("B"), // Substream 1: Values starting with "B"
    (key, value) -> true                   // Substream 2: All other values
);
```

In this example:

- Records with values starting with `"A"` are sent to the first substream (`branches[0]`).
- Records with values starting with `"B"` are sent to the second substream (`branches[1]`).

- All other records are sent to the third substream (`branches[2]`).

Key Notes:

- A record is assigned to only one substream (the first match).
- If no predicate matches, the record is dropped.
- This method is useful for routing records into specific processing paths based on conditions.

These extract operations are foundational tools in Kafka Streams, allowing developers to build precise data pipelines by filtering and branching streams according to business logic. By using `filter`, `filterNot`, and `branch`, developers can effectively narrow down or distribute data for targeted processing.

3.3.2 Transform Operations

Transformation methods in Kafka Streams are used to manipulate the data within a stream by modifying keys, values, or both. These operations are generally stateless and operate record by record unless otherwise specified. Below are detailed descriptions of key transformation methods:

3.3.2.a Transform Key and Value

`map(KeyValueMapper<? super K,? super V,? extends KeyValue<? extends KR,? extends VR» mapper)`

The `map` method transforms both the key and value of each record into a new key-value pair. The provided `KeyValueMapper` is applied to each record, enabling arbitrary modifications of the key and value types. This method is stateless, processing records independently.
Parameters:

- **mapper:** A `KeyValueMapper` function that computes a new key-value pair for each record.

Returns:

- A `KStream` containing records with new keys and values, which can be of different types.

Example: The following example normalizes the key to uppercase and counts the number of words in the value:

```
KStream<String, String> inputStream = builder.stream("topic");
KStream<String, Integer> outputStream = inputStream.map((key, value) ->
    new KeyValue<>(key.toUpperCase(), value.split(" ").length)
);
```

Key Notes:

- This operation might result in internal data redistribution if a key-based operation (e.g., aggregation or join) is applied downstream.
- The `KeyValueMapper` must return a `KeyValue` object and cannot return `null`.

3.3.2.b Transform Value Only

`mapValues(ValueMapper<? super V,? extends VR> mapper)`

The `mapValues` method modifies only the value of each record while keeping the key unchanged. It is a stateless operation, and no internal data redistribution occurs since the key remains the same.

Parameters:

- **mapper:** A `ValueMapper` function that computes a new value for each record.

Returns:

- A `KStream` containing records with unmodified keys and transformed values.

Example: The following example counts the number of words in the value:

```
KStream<String, String> inputStream = builder.stream("topic");
KStream<String, Integer> outputStream = inputStream.mapValues(value -> value.split(" ").length);
```

Key Notes:

- This method preserves data co-location with respect to the key, avoiding internal data redistribution.
- It is often used when key-based operations are required later in the pipeline.

3.3.2.c Transform Value with Access to Key

`mapValues(ValueMapperWithKey<? super K,? super V,? extends VR> mapper)`

The `mapValues` method with `ValueMapperWithKey` extends the functionality of the standard `mapValues` by providing access to both the key and value when transforming the record. This is useful for scenarios where the transformation logic depends on the key.

Parameters:

- **mapper:** A `ValueMapperWithKey` function that computes a new value for each record using both the key and value.

Returns:

- A `KStream` containing records with unmodified keys and transformed values.

Example: The following example counts the total number of characters in both the key and value:

```
KStream<String, String> inputStream = builder.stream("topic");
KStream<String, Integer> outputStream = inputStream.mapValues((key, value) ->
    key.length() + value.split(" ").length
);
```

Key Notes:

- The key is read-only and should not be modified as it could result in corrupt partitioning.
- This operation preserves data co-location with respect to the key, ensuring that no data redistribution occurs.

Comparison of `map` and `mapValues`

Feature	`map`	`mapValues`
Modifies Key	Yes	No
Modifies Value	Yes	Yes
Resulting Data Types	Both key and value can change	Only value type can change
Internal Data Redistribution	Possible if a key-based operation follows	None
Key Usage	Can change the key	Key remains unchanged

Table 1: Comparison of `map` and `mapValues`

Stateful Transformations

Stateful transformations allow records to be processed with additional context from state stores or other operations.

3.3.2.d `transform`

The `transform` method enables stateful transformation of each record in the stream. It applies a `Transformer`, which can access and modify state stores and emit zero or one output record for each input record.
Parameters:

- `transformerSupplier`**:** Supplies the `Transformer` for the operation.
- `stateStoreNames`**:** Names of the state stores used by the processor.

Returns:

- A `KStream` containing transformed records, potentially with new keys and values.

Example: The following example uses a `Transformer` to modify records with access to a state store:

```
// Create and register a state store
StoreBuilder<KeyValueStore<String, String>> storeBuilder =
    Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("myTransformState"),
        Serdes.String(),
        Serdes.String()
    );
builder.addStateStore(storeBuilder);

// Apply the transformation
KStream<String, String> transformedStream = inputStream.transform(
    new TransformerSupplier<String, String, KeyValue<String, String>>() {
        @Override
        public Transformer<String, String, KeyValue<String, String>> get() {
            return new Transformer<>() {
                private KeyValueStore<String, String> stateStore;
```

```
                @Override
                public void init(ProcessorContext context) {
                    this.stateStore = (KeyValueStore<String, String>)
                        context.getStateStore("myTransformState");
                    context.schedule(Duration.ofSeconds(1),
                        PunctuationType.WALL_CLOCK_TIME,
                        timestamp -> {
                            // Periodic actions can be performed here
                        });
                }

                @Override
                public KeyValue<String, String> transform(String key, String value) {
                    // Access state and modify record
                    stateStore.put(key, value);
                    return new KeyValue<>(key.toUpperCase(), value.toLowerCase());
                }

                @Override
                public void close() {
                    // Clean up resources
                }
            };
        }
    },
    "myTransformState"
);
```

Key Notes:

- A state store must be registered beforehand.
- The `Transformer` can emit multiple records using `context.forward()`.
- This method does not trigger auto-repartitioning if upstream operations are key-changing.

3.3.2.e `transformValues`

The `transformValues` method modifies the value of each record in the stream. This method operates on record values only and can access state stores for additional context.
Parameters:

- `valueTransformerSupplier`**:** Supplies the `ValueTransformer` for the operation.
- `stateStoreNames`**:** Names of the state stores used by the processor.

Returns:

- A `KStream` with transformed values.

Example: The following example uses a `ValueTransformer` to modify record values:

```
// Create and register a state store
StoreBuilder<KeyValueStore<String, String>> storeBuilder =
    Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("myValueTransformState"),
        Serdes.String(),
        Serdes.String()
    );
builder.addStateStore(storeBuilder);

// Apply the value transformation
KStream<String, String> transformedStream = inputStream.transformValues(
    new ValueTransformerSupplier<String, String>() {
        @Override
        public ValueTransformer<String, String> get() {
            return new ValueTransformer<>() {
                private KeyValueStore<String, String> stateStore;

                @Override
                public void init(ProcessorContext context) {
                    this.stateStore = (KeyValueStore<String, String>)
                        context.getStateStore("myValueTransformState");
                }

                @Override
                public String transform(String value) {
                    // Modify the value
                    return value.toUpperCase();
                }

                @Override
                public void close() {
                    // Clean up resources
                }
            };
        }
    },
    "myValueTransformState"
);
```

Key Notes:

- This method does not modify keys, preserving data co-location for key-based operations.
- No additional `KeyValue` pairs can be emitted in this method.

3.3.2.f `transformValues` with Key Access

This variation of `transformValues` allows the use of both keys and values for transformation. A `ValueTransformerWithKey` is applied to compute new values.
Parameters:

- `valueTransformerSupplier`: Supplies the `ValueTransformerWithKey` for the operation.
- `stateStoreNames`: Names of the state stores used by the processor.

Returns:

- A `KStream` with new values computed using both keys and values.

Example: The following example uses a `ValueTransformerWithKey` to compute a new value:

```
KStream<String, Integer> transformedStream = inputStream.transformValues(
    new ValueTransformerWithKeySupplier<>() {
        @Override
        public ValueTransformerWithKey<String, String, Integer> get() {
            return new ValueTransformerWithKey<>() {
                @Override
                public void init(ProcessorContext context) {
                    // Initialization logic
                }

                @Override
                public Integer transform(String key, String value) {
                    // Compute the new value using both key and value
                    return key.length() + value.length();
                }

                @Override
                public void close() {
                    // Cleanup logic
                }
            };
        }
    }
);
```

Key Notes:

- The key remains read-only and should not be modified.
- This method ensures no internal data redistribution for key-based operations.

These methods expand the capabilities of Kafka Streams, enabling stateful and advanced transformations for building robust stream-processing applications. They are essential for scenarios requiring periodic actions, stateful processing, or customized record transformations.

3.3.2.g Flat Transformations

The "flat" versions of transformation methods, `flatMap` and `flatMapValues`, extend the functionality of `map` and `mapValues` respectively. They are designed for scenarios where a single input record should result in multiple output records.
Key Differences Between `map` and `flatMap`:

- `map`: Each input record produces exactly one output record.

- `flatMap`**:** Each input record can produce zero, one, or multiple output records.

Key Differences Between `mapValues` and `flatMapValues`:

- `mapValues`**:** Modifies the value of each input record, producing one output record per input record.
- `flatMapValues`**:** Allows the value of a single input record to generate multiple values in the output stream.

Below are detailed descriptions of flat transformation methods.

3.3.2.h `flatMap`

The `flatMap` method allows each input record to generate zero, one, or multiple output records. This is useful for scenarios like tokenizing a sentence into individual words or splitting nested data structures into flattened records.
Example:

```
KStream<String, String> inputStream = builder.stream("input-topic");
KStream<String, String> wordStream = inputStream.flatMap((key, value) -> {
    List<KeyValue<String, String>> words = new ArrayList<>();
    for (String word : value.split(" ")) {
        words.add(new KeyValue<>(key, word));
    }
    return words;
});
```

For an input record `key: "1", value: "hello world"`, the output would be:

```
key: "1", value: "hello"
key: "1", value: "world"
```

3.3.2.i `flatMapValues`

The `flatMapValues` method is similar to `flatMap`, but it only operates on the value of each record. It allows the value of a single input record to produce multiple output values while keeping the key unchanged.
Example:

```
KStream<String, String> inputStream = builder.stream("input-topic");
KStream<String, String> wordStream = inputStream.flatMapValues(value ->
    Arrays.asList(value.split(" "))
);
```

For an input record `key: "1", value: "hello world"`, the output would be:

```
key: "1", value: "hello"
key: "1", value: "world"
```

3.3.2.j `flatTransform`

The `flatTransform` method is a stateful transformation similar to `flatMap`, but it allows access to state stores and supports emitting multiple output records for each input record. This method is ideal for advanced scenarios requiring context-aware transformations.
Example:

```
// Create and register a state store
StoreBuilder<KeyValueStore<String, String>> storeBuilder =
    Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("myFlatTransformState"),
        Serdes.String(),
        Serdes.String()
    );
builder.addStateStore(storeBuilder);

// Apply flatTransform
KStream<String, String> transformedStream = inputStream.flatTransform(
    new TransformerSupplier<String, String, Iterable<KeyValue<String, String>>>() {
        @Override
        public Transformer<String, String, Iterable<KeyValue<String, String>>> get() {
            return new Transformer<>() {
                private KeyValueStore<String, String> stateStore;

                @Override
                public void init(ProcessorContext context) {
                    this.stateStore = (KeyValueStore<String, String>)
                        context.getStateStore("myFlatTransformState");
                }

                @Override
                public Iterable<KeyValue<String, String>> transform(String key, String value) {
                    List<KeyValue<String, String>> results = new ArrayList<>();
                    for (String word : value.split(" ")) {
                        results.add(new KeyValue<>(key, word));
                    }
                    return results;
                }

                @Override
                public void close() {}
            };
        }
    },
    "myFlatTransformState"
);
```

3.3.2.k `flatTransformValues`

The `flatTransformValues` method extends the functionality of `flatMapValues` by supporting stateful transformations and accessing state stores. It computes multiple output values for each input record's value.
Example:

```
// Create and register a state store
StoreBuilder<KeyValueStore<String, String>> storeBuilder =
    Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("myFlatTransformValueState"),
        Serdes.String(),
        Serdes.String()
    );
builder.addStateStore(storeBuilder);

// Apply flatTransformValues
KStream<String, String> transformedStream = inputStream.flatTransformValues(
    new ValueTransformerWithKeySupplier<String, String, Iterable<String>>() {
        @Override
        public ValueTransformerWithKey<String, String, Iterable<String>> get() {
            return new ValueTransformerWithKey<>() {
                private KeyValueStore<String, String> stateStore;

                @Override
                public void init(ProcessorContext context) {
                    this.stateStore = (KeyValueStore<String, String>)
                        context.getStateStore("myFlatTransformValueState");
                }

                @Override
                public Iterable<String> transform(String key, String value) {
                    return Arrays.asList(value.split(" "));
                }

                @Override
                public void close() {}
            };
        }
    },
    "myFlatTransformValueState"
);
```

For an input record `key: "1", value: "hello world"`, the output would be:

```
key: "1", value: "hello"
key: "1", value: "world"
```

These "flat" transformation methods expand the capabilities of Kafka Streams by supporting one-to-many transformations and stateful processing, making them essential for complex stream-processing scenarios.

3.3.3 Aggregation and Join Operations

Aggregation and join operations in Kafka Streams allow for advanced stateful stream processing. These operations enable grouping, combining, and correlating records within or across streams. Below is a detailed breakdown of the key methods:

Grouping Records

Before applying aggregation operators like `count`, `sum`, or `reduce`, the records in a stream must be grouped by a key. Kafka Streams provides two key methods for grouping records: `groupByKey` and `groupBy`.

3.3.3.a groupByKey

The `groupByKey` method groups records by their current key. This method is typically used when the key already represents the logical grouping for aggregation.
Key Features:

- Retains the existing keys and values of the records.
- Requires the specification of serializers and deserializers (via the `Grouped` instance).
- If no repartitioning has occurred after a key-changing operation (e.g., `selectKey`, `map`), an internal repartitioning topic may be created to ensure proper partitioning.

Example:

```
Grouped<String, String> grouped = Grouped.with(Serdes.String(), Serdes.String());
KGroupedStream<String, String> groupedStream = inputStream.groupByKey(grouped);
```

Repartitioning Considerations: If the key has been modified upstream, Kafka Streams will create an internal repartitioning topic named `${applicationId}-<name>-repartition`. You can inspect these topics using `Topology.describe()`.

3.3.3.b groupBy

The `groupBy` method allows records to be grouped by a newly selected key, using a `KeyValueMapper` to compute the grouping key. This is useful when the logical grouping key differs from the current key.
Key Features:

- The new key is selected using a `KeyValueMapper`.
- If the new key differs from the current partitioning, an internal repartitioning topic may be created.
- Preserves the original values of the records.

Example:

```
KGroupedStream<String, String> groupedStream = inputStream.groupBy(
    (key, value) -> value.split(" ")[0],  // Group by the first word in the value
    Grouped.with(Serdes.String(), Serdes.String())
);
```

Comparison with groupByKey:

- Use `groupByKey` when the existing key represents the logical grouping.
- Use `groupBy` when a new key needs to be computed for grouping.

Joining Streams

Kafka Streams supports joining records from one stream with another, or with a table, using the `join`, `leftJoin`, and `outerJoin` methods. These joins are commonly used for enriching data or combining related records.

3.3.3.c join

A standard inner join that combines records with matching keys from both streams. Only records with keys present in both streams are included in the result.
Example:

```
KStream<String, String> joinedStream = stream1.join(
    stream2,
    (value1, value2) -> value1 + ":" + value2,
    JoinWindows.of(Duration.ofMinutes(5))  // 5-minute window for matching
);
```

3.3.3.d leftJoin

A left join includes all records from the left stream, even if there is no matching record in the right stream. If no match is found, the join result includes the left record with a `null` value for the right-hand side.
Example:

```
KStream<String, String> leftJoinedStream = stream1.leftJoin(
    stream2,
    (value1, value2) -> value1 + ":" + (value2 == null ? "no match" : value2),
    JoinWindows.of(Duration.ofMinutes(5))
);
```

3.3.3.e outerJoin

An outer join includes all records from both streams, even if no matching key is found. When no match is found, the result includes the record with a `null` value for the non-matching side.
Example:

```
KStream<String, String> outerJoinedStream = stream1.outerJoin(
    stream2,
    (value1, value2) -> (value1 == null ? "no match" : value1) + ":" +
                        (value2 == null ? "no match" : value2),
    JoinWindows.of(Duration.ofMinutes(5))
);
```

Repartitioning and Internal Topics

For all the above grouping and joining operations:

- Kafka Streams may create internal repartitioning topics if the keys need to be redistributed across partitions for proper alignment.
- These topics can be inspected using `Topology.describe()` and typically follow the naming pattern:

```
${applicationId}-<name>-repartition
```

Summary Table of Join Types

Join Type	Description	When to Use
Inner Join	Includes only matching records from both streams.	When you only need matches from both.
Left Join	Includes all records from the left stream and matches from the right.	When left data is primary, and right data is supplementary.
Outer Join	Includes all records from both streams, with `null` for non-matches.	When you need a full dataset with all matches and non-matches.

Table 2: Comparison of Join Types

This section provides comprehensive coverage of aggregation and join operations, enabling flexible and efficient stream processing pipelines.

3.3.4 Load Operations

Load operations in Kafka Streams allow for materializing processed data into Kafka topics. These methods enable the final output of the stream processing pipeline or intermediate materialization for further processing.

3.3.4.a `to(String topic, Produced<K,V> produced)`

This version of `to` allows for additional configurations using the `Produced` instance, such as specifying serializers, partitioners, and other producer settings.
Parameters:

- `topic`**:** The name of the Kafka topic.
- `produced`**:** The `Produced` instance used to configure serialization and other options.

Example:

```
Produced<String, String> produced = Produced.with(Serdes.String(), Serdes.String());
stream.to("output-topic", produced);
```

In this example, custom serializers for the key and value are specified.

3.3.4.b `to(TopicNameExtractor<K,V> topicExtractor, Produced<K,V> produced)`

The dynamic version of `to` determines the output topic for each record using a `TopicNameExtractor`. This is useful when records need to be routed to different topics based on their content.
Parameters:

- `topicExtractor`: A `TopicNameExtractor` to dynamically determine the topic for each record.
- `produced`: The `Produced` instance used to configure serialization and other options.

Example:

```
stream.to((key, value, recordContext) -> key + "-topic", produced);
```

In this example, records are sent to topics based on their keys, appending `"-topic"` to each key.

3.3.4.c `through(String topic, Produced<K,V> produced)`

The `through` method writes the records to a Kafka topic and simultaneously creates a new `KStream` from the same topic. This is useful for intermediate materialization of data or ensuring correct partitioning for downstream processing.
Parameters:

- `topic`: The name of the Kafka topic.
- `produced`: The `Produced` instance used to configure serialization and other options.

Returns: A `KStream` containing the same records as the original stream.
Example:

```
KStream<String, String> intermediateStream = stream.through(
    "intermediate-topic", Produced.with(Serdes.String(), Serdes.String())
);
```

In this example, the processed data is written to `intermediate-topic` and then read back into `intermediateStream`.
Comparison with `to`: - `to`: Sends the records to a Kafka topic but does not provide a new `KStream`. - `through`: Writes records to a topic and reads them back into a new `KStream`.

Key Notes

- All topics specified in `to` or `through` must be created manually before starting the Kafka Streams application.
- The `Produced` instance allows for configuring serializers, partitioners, and other producer settings for fine-grained control.
- Use `through` when repartitioning or intermediate materialization is required for downstream processing.
- Dynamic topic routing with `to(TopicNameExtractor)` is particularly useful for multi-tenant or topic-routing scenarios.

This section highlights the versatility of Kafka Streams in materializing and routing processed data efficiently.

3.3.5 Debugging and Logging

Debugging and logging operations in Kafka Streams provide mechanisms to inspect, log, or perform side effects on stream records. These methods are stateless and operate on each record independently. They are particularly useful for testing, debugging, and monitoring purposes.

3.3.5.a `foreach(ForeachAction<? super K,? super V> action)`

The `foreach` method is a terminal operation that performs an action on each record in the stream. It is typically used for logging, auditing, or performing side effects such as writing to an external system. Since `foreach` is terminal, it does not return a new stream and concludes the processing pipeline for the records.
Parameters:

- `action`: A `ForeachAction` that specifies the operation to perform on each record.

Example: Logging Stream Records:

```
stream.foreach((key, value) -> System.out.println(key + ": " + value));
```

In this example, each record in the stream is logged to the console in the format `key: value`.
Key Notes:

- `foreach` is a terminal operation; it does not return a new stream.
- If an action fails, processing for that record is stopped, but other records may still be processed.

3.3.5.b `peek(ForeachAction<? super K,? super V> action)`

The `peek` method is a non-terminal operation that performs an action on each record in the stream and returns an unchanged stream. It is typically used for debugging purposes, such as inspecting records at a specific stage in the pipeline, without altering the records themselves.
Parameters:

- `action`: A `ForeachAction` that specifies the operation to perform on each record.

Example: Debugging Stream Records:

```
stream.peek((key, value) -> System.out.println("Debugging: " + key + " -> " + value))
      .mapValues(value -> value.toUpperCase());
```

In this example, the records are logged with the prefix `"Debugging:"`, and then their values are transformed to uppercase using `mapValues()`.
Key Notes:

- `peek` is non-terminal; the stream remains unchanged.
- Since it is stateless, it may execute multiple times for a single record in failure cases.
- Ideal for side effects like logging or metrics collection.

Comparison of `foreach` and `peek`

Feature	`foreach`	`peek`
Terminal?	Yes, ends the pipeline.	No, stream remains unchanged.
Returns Stream?	No.	Yes.
Use Cases	Logging, external side effects, auditing.	Debugging, monitoring, statistics collection.
Stateful?	Stateless.	Stateless.

Table 3: Comparison of `foreach` and `peek`

These methods provide essential tools for inspecting and debugging Kafka Streams pipelines while ensuring flexibility in performing side effects or logging without modifying the original data.

4 Custom Aggregation in Kafka Streams

Custom aggregation in Kafka Streams enables developers to process and summarize streaming data according to unique business requirements. By leveraging stateful operations, Kafka Streams facilitates advanced real-time data processing.

4.1 Introduction to Aggregation

Aggregation is the process of combining data records to compute a summary or transformation over a stream of data. In Kafka Streams, aggregations are stateful operations requiring data to be grouped by a key. The aggregated results are stored in state stores, which ensure fault tolerance and scalability.

Key Features of Aggregation:

- Aggregations operate on grouped streams (`KGroupedStream`) or grouped tables (`KGroupedTable`).
- Aggregations result in a new `KTable`, representing the evolving state of the aggregation.
- Kafka Streams supports custom aggregation logic through the `aggregate()` method, in addition to built-in operations like `count()`, `sum()`, and `reduce()`.

4.2 Custom Aggregation with `aggregate()`

The `aggregate()` method allows developers to define their custom aggregation logic. This method provides flexibility by enabling the initialization of an aggregate value, the accumulation of new records, and the reversal of previous aggregations when records are updated.

Key Components:

- **Initializer**: Defines the initial state of the aggregation.
- **Aggregator**: Specifies how new records are combined with the current aggregate.
- **Materialized**: Configures the serialization and state store used to store the aggregation result.

Method Signature:

```
<VR> KTable<K, VR> aggregate(
    Initializer<VR> initializer,
    Aggregator<? super K, ? super V, VR> aggregator,
    Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized
);
```

Parameters:

- `initializer`: A function that creates the initial aggregate value.
- `aggregator`: A function that merges each incoming record into the aggregate.
- `materialized`: Specifies the state store configuration.

4.3 Example: Building a Custom Aggregation

Scenario: Calculating the total sales amount for each product category from a stream of sales transactions.

```
KGroupedStream<String, Double> groupedStream = inputStream.groupBy(
    (key, value) -> value.getCategory(),
    Grouped.with(Serdes.String(), Serdes.Double())
);

KTable<String, Double> totalSales = groupedStream.aggregate(
    () -> 0.0,  // Initializer: Start with zero sales
    (key, sale, aggregate) -> aggregate + sale,  // Aggregator: Add each sale
    Materialized.<String, Double, KeyValueStore<Bytes, byte[]>>as("sales-store")
        .withKeySerde(Serdes.String())
        .withValueSerde(Serdes.Double())
);
```

Explanation:

- **Initialization**: The total sales start at `0.0`.
- **Aggregation Logic**: Each sales record is added to the current total.
- **State Store**: The aggregated results are stored in a durable and queryable state store named `sales-store`.

4.4 Aggregation with Windows

Windowed aggregation is used when summarizing data over fixed time intervals. Kafka Streams supports various windowing types such as tumbling, hopping, sliding, and session windows.
Key Components of Windowed Aggregation:

- **Window Types**:
  - **Tumbling Windows**: Fixed, non-overlapping intervals.
  - **Hopping Windows**: Fixed intervals with overlaps.
  - **Sliding Windows**: Based on event occurrence.
  - **Session Windows**: Based on activity gaps.
- **Window Size**: The duration of the aggregation window.
- **Grace Period**: The additional time allowed for late-arriving records.

Code Example:

```
KGroupedStream<String, Double> groupedStream = inputStream.groupBy(
    (key, value) -> value.getCategory(),
    Grouped.with(Serdes.String(), Serdes.Double())
);

KTable<Windowed<String>, Double> windowedSales = groupedStream
```

```
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(10)))
    .aggregate(
        () -> 0.0,  // Initializer
        (key, sale, aggregate) -> aggregate + sale,  // Aggregator
        Materialized.<String, Double, WindowStore<Bytes, byte[]>>as("windowed-sales-store")
            .withKeySerde(Serdes.String())
            .withValueSerde(Serdes.Double())
    );
```

Explanation:

- Aggregates sales data for each category in 10-minute intervals.
- Late-arriving records are ignored because no grace period is defined.

4.5 Performance Optimization

To optimize performance during aggregation, consider the following strategies:

- **State Store Configuration**: Use compacted topics for backing state stores to minimize disk usage.
- **Parallelism**: Ensure proper partitioning of input topics to leverage Kafka Streams' distributed architecture.
- **Memory Management**: Use Serdes that are efficient for serializing/deserializing large objects.
- **Windowing**: Choose appropriate window sizes to balance granularity and computational overhead.

Example: Using `Materialized.withCachingDisabled()` for low-latency aggregations.

```
Materialized<String, Double, KeyValueStore<Bytes, byte[]>> materialized =
    Materialized.<String, Double, KeyValueStore<Bytes, byte[]>>as("optimized-store")
        .withKeySerde(Serdes.String())
        .withValueSerde(Serdes.Double())
        .withCachingDisabled();
```

4.6 Use Cases for Custom Aggregation

Custom aggregation can be applied in a variety of real-world scenarios:

- **Real-Time Analytics**: Calculating metrics like average response time, total sales, or engagement rates.
- **Monitoring and Alerting**: Summarizing log data to identify anomalies or trends.
- **Event Processing**: Aggregating events for fraud detection, user activity tracking, or IoT device monitoring.
- **Financial Services**: Summarizing transactional data for compliance or risk analysis.

Example: Aggregating User Clicks for Personalized Recommendations.

```
KTable<String, Long> userClicks = groupedStream.aggregate(
    () -> 0L,
    (key, click, aggregate) -> aggregate + 1,
    Materialized.with(Serdes.String(), Serdes.Long())
);
```

This section demonstrates how Kafka Streams enables custom aggregation to meet diverse processing requirements. From simple accumulations to complex windowed computations, Kafka Streams provides powerful tools to build scalable and performant stream-processing applications.

5 Demonstration on Streams Processing with Kafka Streams

This section provides a comprehensive demonstration of stream processing using Kafka Streams. The goal is to showcase the practical implementation of a Kafka Streams application, from setting up the environment to analyzing the results.

5.1 Setting Up the Environment

Before starting with Kafka Streams, a proper environment setup is crucial. The following components are required:

5.1.1 Prerequisites

- Java Development Kit (JDK) 8 or higher.
- Apache Kafka installed and running (version 2.0 or above).
- A development environment or IDE (e.g., IntelliJ IDEA, Eclipse).
- Maven or Gradle for managing dependencies.

5.1.2 Installing Apache Kafka

- Download the latest version of Kafka from the Apache Kafka website.
- Extract the downloaded archive and navigate to the Kafka directory.
- Set up Kafka broker properties:

```
KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c config/kraft/server.properties
```

- Start the Kafka broker with Kraft:

```
bin/kafka-server-start.sh config/kraft/server.properties
```

5.1.3 Creating Topics

Create the input and output topics for the demonstration:

```
bin/kafka-topics.sh --create --topic input-topic --bootstrap-server localhost:9092
--replication-factor 1     --partitions 1
bin/kafka-topics.sh --create --topic output-topic --bootstrap-server localhost:9092
--replication-factor 1     --partitions 1
```

After creating the topics, we can use this to view the current topics on the broker:

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe
```

The result should look like this:

```
tuki@DESKTOP-OLS62J1:/mnt/d/0_Academic/H241/PIPKafka/kafka_2.13-3.9.0$ bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe
Topic: input-topic      TopicId: yjdwPTWiT9STiL7Bdhgl4w PartitionCount: 1       ReplicationFactor: 1    Configs: segment.bytes=1073741824
        Topic: input-topic      Partition: 0    Leader: 1       Replicas: 1     Isr: 1  Elr:    LastKnownElr:
Topic: output-topic     TopicId: 3HatCA6CTRWArs9V8rDQXw PartitionCount: 1       ReplicationFactor: 1    Configs: segment.bytes=1073741824
        Topic: output-topic     Partition: 0    Leader: 1       Replicas: 1     Isr: 1  Elr:    LastKnownElr:
```

Figure 10: Description of current topics

5.1.4 Adding Dependencies

Add Kafka Streams dependencies in your `pom.xml` (for Maven):

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>3.0.0</version>
</dependency>
```

5.2 Use Case Description

The use case involves processing streaming sales transactions. The goal is to aggregate sales data by product category and calculate the total sales amount for each category in real-time.

5.2.1 Scenario

- **Input**: A stream of sales records with fields for `category`, and `amount`.
- **Processing**: Group records by `category` and calculate the total sales for each category.
- **Output**: The aggregated results are written to an output topic.

5.2.2 Input Data Example

```
{"category": "electronics", "amount": 100.0}
{"category": "clothing", "amount": 50.0}
{"category": "electronics", "amount": 200.0}
```

5.2.3 Expected Output

```
electronics   300.0
clothing      50.0
```

5.3 Implementing the Topology

5.3.1 Define the Kafka Streams Configuration

```
// Kafka Streams Configuration
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "sales-aggregator");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
```

5.3.2 Build the Stream Topology

```
// Build the Stream Topology
StreamsBuilder builder = new StreamsBuilder();

// Create the stream from the input topic with specific consumption configurations
KStream<String, String> inputStream = builder.stream("input-topic",
    Consumed.with(Serdes.String(), Serdes.String())); // Specify the key and value serdes

// Process and map the incoming data to (category, sale) pair
KStream<String, Double> processedStream = inputStream
    .map((key, value) -> {
        try {
            // Parse the JSON value and extract the fields
            JsonNode node = new ObjectMapper().readTree(value);
            String category = node.get("category").asText();
            Double amount = node.get("amount").asDouble();

            // Return a KeyValue pair with the correct types
            return new KeyValue<>(category, amount); // Key is category, value is amount
        } catch (JsonProcessingException e) {
            e.printStackTrace(); // Handle the exception
            return new KeyValue<>(null, 0.00); // Return a default value in case of error
        }
    });

// Now, group by category and aggregate the sales per category
KGroupedStream<String, Double> groupedStream = processedStream
    .groupByKey(Grouped.with(Serdes.String(), Serdes.Double())); // Group by category

// Aggregate the sales per category
KTable<String, Double> totalSales = groupedStream.aggregate(
    () -> 0.0, // Initial value for the aggregator (total starts at 0.0)
    (key, sale, aggregate) -> aggregate + sale, // Sum the sales for each category
    Materialized.with(Serdes.String(), Serdes.Double()) // Specify the key and value Serdes
);

// Write the aggregated result to the "output-topic"
totalSales.toStream().mapValues(value -> value.toString()) // Convert Double to String
   .to("output-topic", Produced.with(Serdes.String(), Serdes.String()));
```

5.3.3 Create and Start the Kafka Streams Application

```
KafkaStreams streams = new KafkaStreams(builder.build(), props);
streams.start();
```

5.4 Running the Application

5.4.1 Start Kafka and Create Topics

Ensure the Kafka broker and ZooKeeper are running, and the topics are created.

5.4.2 Produce Sample Data

Use the Kafka console producer to send sample records to the `input-topic`:

```
bin/kafka-console-producer.sh --topic input-topic --bootstrap-server localhost:9092
>{"category": "electronics", "amount": 100.0}
>{"category": "clothing", "amount": 50.0}
>{"category": "electronics", "amount": 200.0}
```

5.4.3 Consume the Output

Use the Kafka console consumer to view the aggregated results from the `output-topic`:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic output-topic
--from-beginning --property print.key=true --property print.value=true --property
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer --property
value.deserializer=org.apache.kafka.common.serialization.DoubleDeserializer
{"electronics": 300.0}
{"clothing": 50.0}
```

5.5 Enhancing the Demonstration

5.5.1 Adding Windowing

Enhance the application by adding a time window to group sales by category within specific intervals:

```
KTable<Windowed<String>, Double> windowedSales = groupedStream
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(10)))
    .aggregate(
        () -> 0.0,
        (key, sale, aggregate) -> aggregate + sale,
        Materialized.with(Serdes.String(), Serdes.Double())
    );
windowedSales.toStream().to("output-topic",
            Produced.with(WindowedSerdes.timeWindowedSerdeFrom(String.class), Serdes.Double()));
```

5.5.2 Integrating External Systems

Write aggregated data to an external database or dashboard system for real-time analytics.

5.5.3 Error Handling

Implement deserialization error handling and retry mechanisms for robustness.

5.6 Observations and Metrics

5.6.1 Key Observations

- **Data Accuracy**: Aggregated sales data matched the expected output.
- **Latency**: The pipeline processed data in near real-time.
- **Fault Tolerance**: Stream processing resumed seamlessly after restarting the Kafka Streams application.

5.6.2 Metrics to Monitor

- **Throughput**: Measure the number of records processed per second.
- **State Store Size**: Monitor the size of state stores used for aggregations.
- **Lag**: Ensure minimal consumer lag in processing records.

5.6.3 Monitoring Tools

- Use Kafka Streams' built-in metrics accessible via JMX.
- Visualize metrics using monitoring tools like Prometheus or Grafana.

This demonstration highlights the power and flexibility of Kafka Streams in implementing real-time stream processing solutions. It showcases practical steps, potential enhancements, and key metrics for successful application deployment.

CHAPTER 2
APACHE SPARK

1 Introduction

1.1 What is Apache Spark?

1.1.1 Apache Spark: An Overview

Apache Spark is an open-source distributed computing framework designed for big data analytics, graph processing, and machine learning. It supports programming languages such as Scala, Python, R, Java, and SQL, making it versatile and widely applicable across various industries.

1.1.2 Applications of Apache Spark

Spark is renowned for its ability to process massive datasets, making it invaluable for industries like healthcare, banking, and stock exchanges. Its framework supports batch processing and stream processing, enabling organizations to analyze vast amounts of data efficiently.

The popularity of Spark lies in its robust and well-layered architecture, which is designed for seamless integration with libraries such as Spark SQL, MLlib, and GraphX. Built on a master-slave architecture, it comprises two main programs:

- Master program: Responsible for managing the cluster and resources.
- Worker program: Executes tasks assigned by the master.

The two important aspects of a Spark application are Spark ecosystem and RDD

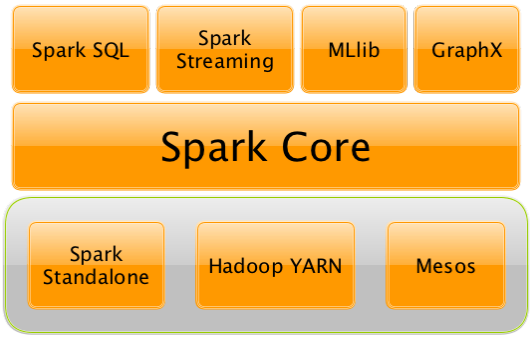


Figure 11: Spark Ecosystem

Spark Ecosystem: Spark ecosystem consists of Spark Core, Spark SQL, Spark Streaming, MLlib, and GraphX. Each module is designed to cater to specific requirements.

- Spark Core: the foundation for parallel and distributed data processing.
- Spark SQL: Simplifies working with structured data and is ideal for users transitioning from relational database systems (RDBMS)
- MLlib: Supports scalable machine learning algorithms.

- GraphX: Facilitates graph processing.

(Details about these modules would be included in later parts of this report)

Resilient Distributed Datasets (RDD): The RDD is the fundamental data structure in Spark. It is immutable and allows for distributed storage and processing. RDDs enable:

- Transformations: Operations that define a new RDD based on existing ones.
- Actions: Operations that return results to the driver after computations.

1.2 Features of Apache Spark

Apache Spark stands out in the big data landscape due to its robust architecture, extensive community support, and diverse library ecosystem. It provides a fault-tolerant and parallelized interface for clusters, enabling efficient and scalable data processing. Below are the primary features of Apache Spark architecture:

- **Speed:** Spark offers unparalleled processing speed, outperforming Hadoop MapReduce by up to 100 times in batch processing. This speed advantage is achieved through controlled partitioning, which divides data into smaller chunks for parallel distribution, minimizing network latency.
- **Polyglot Support:** Apache Spark supports high-level APIs for multiple programming languages, including R, Python, Java, and Scala. This flexibility allows developers to write applications in their preferred language. Additionally, Spark provides interactive shells for Scala (./bin/spark-shell) and Python (./bin/pyspark) for ease of use.
- **Real-Time Computation:** Spark excels at low-latency, in-memory computation, making it ideal for real-time data processing. Its highly scalable architecture can operate on clusters with thousands of nodes, supporting a wide range of computational workloads.
- **Hadoop Integration:** While Spark replaces Hadoop's MapReduce functionality, it remains fully compatible with the Hadoop ecosystem. Spark can run on top of a Hadoop cluster using YARN for resource scheduling, making it a natural choice for organizations transitioning from Hadoop to more advanced data processing frameworks.
- **Machine Learning with MLlib:** Spark's MLlib library provides powerful machine learning capabilities directly integrated into the framework. This eliminates the need for separate tools, offering data engineers a unified engine for tasks such as clustering, classification, and regression.
- **Lazy Evaluation:** One of Spark's key efficiency mechanisms is lazy evaluation. Transformations are not executed immediately but are instead added to a Directed Acyclic Graph (DAG). The DAG is only executed when the driver requests the data, optimizing the execution process and improving overall performance.

1.3 Spark Terminologies

RDD and Dataframe and Dataset:

RDD stands for Resilient Distributed Dataset with each work in its name corresponding to one of its characteristics:

- Resilient: fault-tolerant with the help of lineage graph(DAG)
- Distributed: data in RDD is stored not in one site but on multiple nodes
- Dataset: it represents the records of data which we can work with which can be either JSON, CSV, text file or database via JDBC with no specific data structure

To clarify the use of DAG in RDD, we need to consider the concepts of immutable, type of operations on RDD: transformation and action.

- **Immutable**: RDD cannot be changed once created, instead new RDD would be created to represent the change in the old RDD.
  The new RDD would store with it the information of the previous RDD that it originates from and the operations on that that RDD to achieve the current one. This explains for the fault-tolerant characteristic of the RDD as it can recalculate from the previous RDD instead of calculating with the initial raw data.
- **Transformation**: Operations on RDD involve creating new RDD from the old one. Transformation is further divided into wide and narrow transformation:
  - **Narrow transformation**: represent one-to-one mapping, one partition from child RDD is constructed from at most one partition in parent RDD. This simplicity means that it does not need shuffling data across network, therefore, offering faster execution
  - **Wide transformation**: represent many-to-one mapping, one partition from child RDD might be constructed from many partitions from parent RDD. These transformations need to redistribute data across the data network, therefore, showing higher execution time.
- **Actions**: Operations on RDD involve doing computations on RDD and return that RDD to the driver program

Let put these concepts into an example for better understanding of them

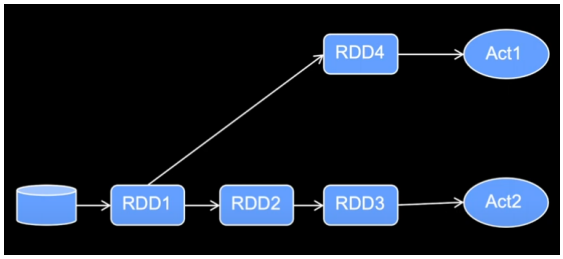


Figure 12: Sequence of operations on RDD

Suppose we need to conduct series of changes on RDD at different stages of time until we achieve a wanted state of the dataset. Only after achieving the wanted condition that we conduct some computations on the dataset and return the result to the driver program. If every step in adjustment period is conducted, there may be two problems arising. Firstly, the changing may be duplicated or not in the order that is optimized for execution. Secondly, this requires scanning through the dataset many times, which is a huge overhead. Spark solve this problem by noting the changes on RDD in this period in a Directed Acyclic Graph(DAG) but not execute any

changes yet. Only when the result is required at the driver program that the plan in DAG is triggered executing all planned operations. By doing this, Spark can read the dataset only once and also optimize the plan for conducting a series of operations on RDD. This explains for many things including: lazy-evaluation property for non-immediate execution; the separation of operations into transformation and actions to distinguish between planned optimized operation and triggered operation.

Dataframe and Dataset are later introduced concepts that are built based on RDD. Therefore, they inherit from RDD many characteristics like: fault-tolerant, lazy evaluation, distributed, ...However, by providing a higher abstraction on the RDD they offer some critical differences that are summarized in the table below

Context	RDD	Dataframe	Dataset
Data representation	Distributed collection of data elements without any schema	Distributed collection organized into named columns	Extension of DataFrames with type safety and object-oriented interface
Type safety	No type-safe	Offer run-time type-safe	Compile-time type checking
APIs	Low-level API that requires more code to perform transformations and actions on data	Provide a high-level API that makes it easier to perform transformations and actions on data	Support both functional and object-oriented programming paradigms and provide a more expressive API for working with data
Schema enforcement	Do not require a schema	Enforce schema at runtime	Enforce schema at compile time.
Optimization	No inbuilt optimization	Catalyst optimizer	Catalyst optimizer
Programming language support	Java, Scala, Python and R	Java, Scala, Python and R	Scala and Java

Figure 13: RDD vs DataFrame vs Dataset

Spark Shell

The Spark Shell is an interactive command-line interface. It enables users to explore Spark features and develop standalone Spark applications. The shell provides auto-completion and a simple way to test and debug code. One of the main advantages of Spark Shell is its ability to handle datasets of varying sizes, making it a crucial tool for prototyping and learning Spark operations.

Spark Application

A Spark Application refers to the computational program that runs on user-provided code to produce results. These applications can continue running in the background even when not actively processing a job, ensuring availability for future computations.

Job

A job in Spark is a high-level parallel computation triggered by a Spark action. It consists of multiple tasks that are executed across the cluster.

Stage

Every job in Spark is divided into smaller units called stages. A stage represents a boundary of computation, with each stage depending on the output of the previous one. Multiple stages are required to complete a job.

Task

A task is the smallest unit of execution in Spark. Each stage is further divided into tasks, with one task assigned to each partition of the data. These tasks are sent to executors for processing.

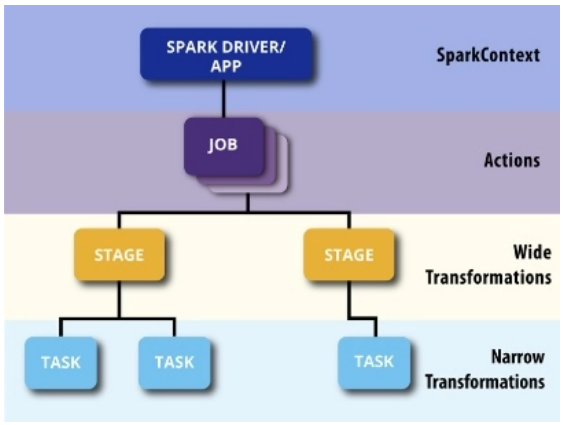


Figure 14: Spark Terminologies

2 Spark Architecture

2.1 Components of Spark run-time architecture

2.1.1 SparkContext

The SparkContext is a central component of the Spark framework. It is responsible for creating Resilient Distributed Datasets (RDDs), accessing Spark services, running jobs, and broadcasting variables. Additionally, SparkContext establishes a connection with the Spark execution environment, functioning as the master of a Spark application. Through SparkContext, users can monitor the status of their application, cancel jobs or stages, and execute jobs synchronously or asynchronously.

2.1.2 Spark Driver

The Spark Driver is the central component where the main computation method runs. It is responsible for:

- Creating user-defined code to generate RDDs and the SparkContext.
- Splitting Spark application tasks into smaller tasks that can be scheduled on executors.
- Converting the user program into tasks and managing their execution.

When a user starts a Spark Shell, the Spark Driver is initiated. The lifecycle of a Spark application is tied to the driver, as the application is considered complete once the driver is terminated.

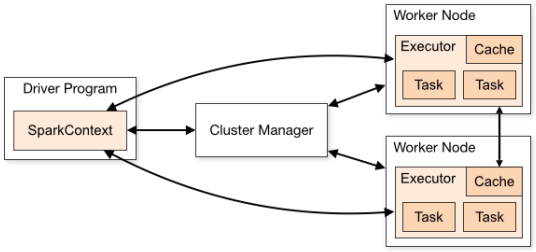


Figure 15: Spark Runtime Architecture

2.1.3 Cluster Manager

The Cluster Manager is responsible for launching both executors and drivers. It uses the Spark Scheduler, often employing scheduling strategies such as FIFO (First In, First Out) to manage jobs and actions.

The cluster manager dynamically adjusts resource allocation based on workload demands. This flexibility ensures optimal resource utilization, as free resources can be reclaimed and reallocated as needed. Spark supports various cluster managers, such as YARN, Mesos, and Standalone Mode, providing versatility across deployment environments.

2.1.4 Executor

Executors are responsible for running the individual tasks within a Spark job. They are launched at the start of the application and continue running throughout its lifecycle. Key responsibilities of executors include:

- Executing tasks assigned by the driver and returning the results back to it.
- Providing in-memory storage for RDDs cached by the user, enabling faster data access.

Even if an executor fails, the Spark application continues to function, as the system is fault-tolerant and can recover tasks as needed. Executors play a crucial role in the distributed nature of Spark, as they handle the actual processing of data across the cluster.

2.2 Working of Spark Architecture

The Spark architecture operates through a collaborative process involving the master node, worker nodes, and the SparkContext. Below is a detailed explanation of its working mechanism

2.2.1 Master Node and Driver Program

The master node houses the driver program, which is responsible for managing your Spark application.

- In a Spark application, the user's code serves as the driver program.
- When using an interactive shell (e.g., Spark Shell), the shell itself functions as the driver.

The driver program initializes the SparkContext, which acts as the gateway to all Spark functionalities. Similar to a database connection in relational databases, every Spark operation passes through the SparkContext.

2.2.2 SparkContext and Cluster Manager

The SparkContext communicates with the cluster manager, which oversees the allocation and scheduling of resources across the cluster. Together, the SparkContext and cluster manager divide the submitted job into smaller tasks and distribute them to worker nodes for execution.

2.2.3 Worker Nodes and RDD Partitions

The worker nodes are slave nodes responsible for executing the tasks assigned by the master node. These tasks operate on partitioned RDDs (Resilient Distributed Datasets) to ensure parallel processing and efficient execution.

- The results of the computations are sent back to the SparkContext.
- As the number of worker nodes increases, the job can be further partitioned into smaller tasks, enabling faster and more efficient execution.

This distributed processing framework, supported by the interaction between the master and worker nodes, ensures scalability, fault tolerance, and optimized performance.

2.3 Principle of Apache Spark in a program

This section explains in detail how Spark operates in a program with a specific data set. The example covered here is used to process web server log files stored in HDFS. It analyzes logs to count requests per (IP, URL) pair, aggregates these requests by URL, and identifies the top 10 most requested URLs.

```
192.168.0.1 - - [22/Nov/2024:10:15:32 +0000] "GET /home HTTP/1.1" 200 512
10.0.0.2 - - [22/Nov/2024:10:15:45 +0000] "POST /login HTTP/1.1" 200 348
172.16.0.3 - - [22/Nov/2024:10:16:12 +0000] "GET /about HTTP/1.1" 404 123
192.168.0.1 - - [22/Nov/2024:10:17:04 +0000] "GET /home HTTP/1.1" 200 512
10.0.0.2 - - [22/Nov/2024:10:17:22 +0000] "GET /dashboard HTTP/1.1" 200 2345
```

Figure 16: Some lines of the dataset used in the example

2.3.1 Program Code

```
import org.apache.spark.sql.SparkSession
import org.apache.log4j._

object WebServerLogAnalysis {

  def main(args: Array[String]): Unit = {
    // Set log level to only print errors
    Logger.getLogger("org").setLevel(Level.ERROR)

    // Step 1
    val spark = SparkSession.builder()
      .appName("WebServerLogAnalysis")
      .config("spark.executor.instances", "5")
      .master("local[*]")
      .getOrCreate()

    // Step 2
    val rawLogs = spark.sparkContext.textFile("hdfs:///user/group1/Alan_
        DF/web-server.log")

    // Step 3
    val parsedLogs = rawLogs
      .map(line => {
        val parts = line.split(" ")
        val ip = parts(0)
        val url = parts(6)
        (ip, url)
      })

    // Step 4
    val requestCountsByPair = parsedLogs
      .map(log => ((log._1, log._2), 1))
      .reduceByKey((x, y) => x + y)
```

```
33        .collect()                              // Action
34
35      println("Request counts by (IP, URL):")
36      requestCountsByPair.foreach(println)
37
38      // Step 5
39      val requestCountsByUrl = parsedLogs
40        .map(log => (log._2, 1))
41        .reduceByKey((x, y) => x + y)
42        .collect()                              // Action
43
44      println("Request counts by URL:")
45      requestCountsByUrl.foreach(println)
46
47      // Step 6
48      val sortedUrls = spark.sparkContext
49        .parallelize(requestCountsByUrl)
50        .sortBy(_._2, ascending = false)
51        .take(10)                               // Action
52
53      println("Top 10 requested URLs:")
54      sortedUrls.foreach(println)
55
56      // Step 7
57      spark.sparkContext.parallelize(sortedUrls)
58        .saveAsTextFile("hdfs:///output/sorted-urls")  // Action
59
60      // Step 8
61      spark.stop()
62    }
63  }
```

2.3.2 Explanation

The above program can be categorized into the following steps:

1. Create a `SparkSession`
2. Load the log file from HDFS
3. Extract (IP, URL) pairs from each log line
4. Count requests for each (IP, URL) pair
5. Aggregate counts by URL
6. Sort URLs by request count in descending order and take the first 10 rows
7. Save the results to a file in HDFS
8. Stop the SparkSession

When running this program (using the `spark-submit`), the Spark driver creates a `sparkContext` that is an entry point into the application, and all operations (transformations and actions) are

executed on the worker nodes, and the resources are managed by the cluster manager (yarn).

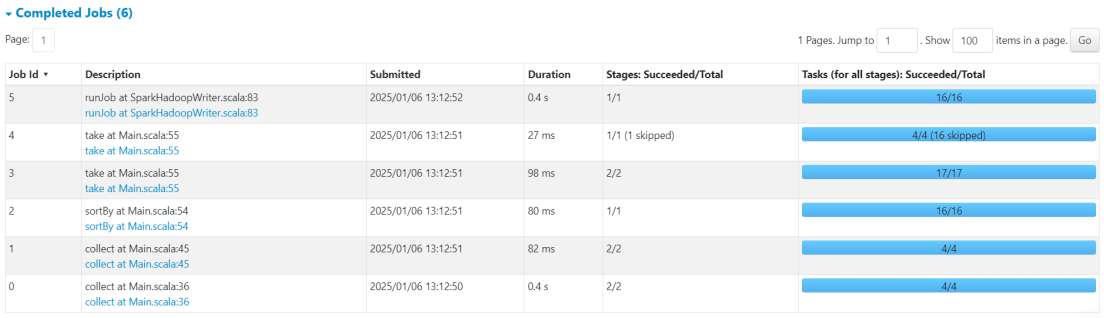


Figure 17: Jobs in the program

In **Figure 16**, the application shows six jobs, but in theory there are four jobs only (due to the principle of Catalyst Optimizer, Action 3 is divided into 3 jobs). Therefore, only 4 jobs are created:

1. Counting requests by (IP, URL)

The `collect()` action gathers all results of the `reduceByKey` transformation into the driver. This triggers a job with multiple stages:

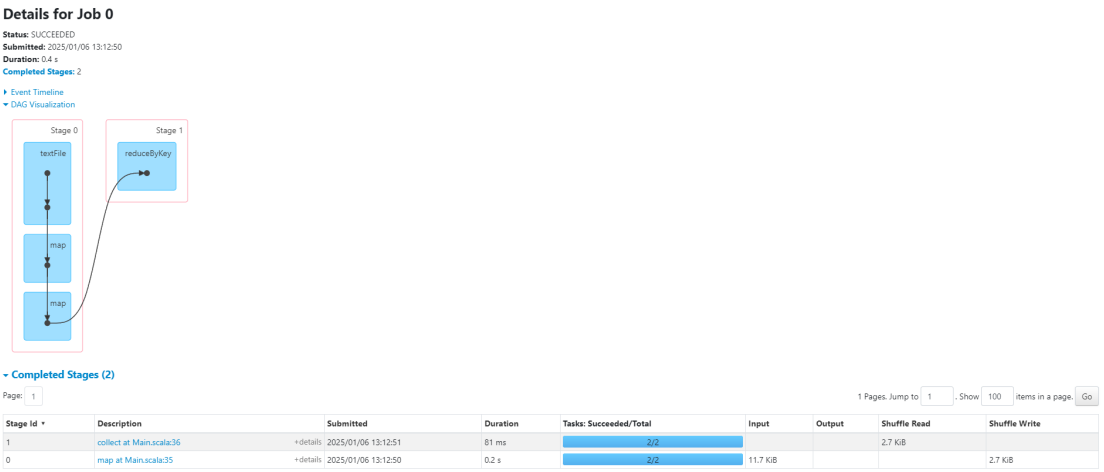


Figure 18: Action 1: Trigger by `collect()`

- Stage 0: Performs the `textFile` and `map` operation to create (IP, URL) pairs with initial counts of 1 **(Narrow Stage)**.
- Stage 1: Executes the `reduceByKey` operation, which involves shuffling data to group and count keys **(Wide Stage)**.

2. Counting requests by URL

The `collect()` action forces Spark to process the preceding transformations. The job executes:

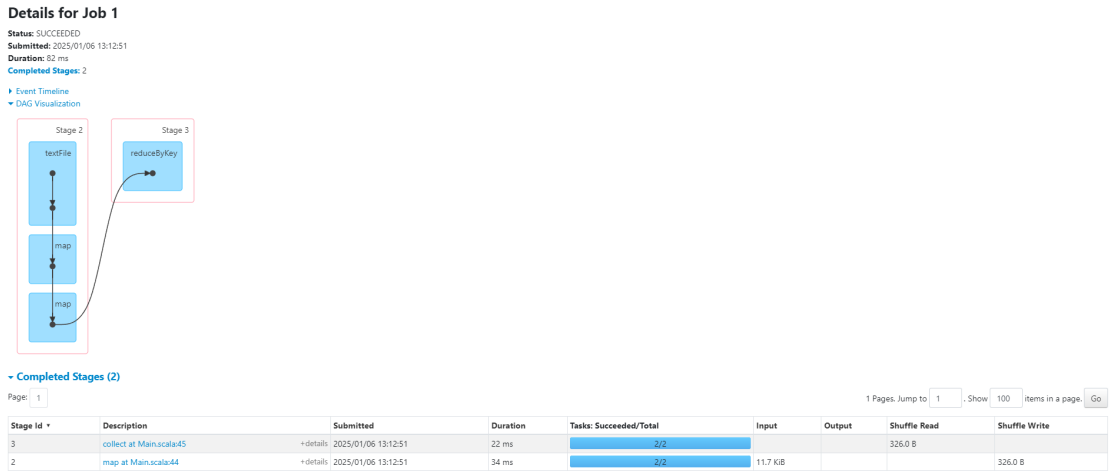


Figure 19: Action 2: Trigger by `collect()`

- Stage 2: Maps log entries to `(URL, 1)` pairs **(Narrow Stage)**. Note that the first `textFile` and `map` operation in this stage is re-evaluate from the RDD `parsedLogs`
- Stage 3: Performs the `reduceByKey` operation with a shuffle write to aggregate counts by URL **(Wide Stage)**.

3. Counting requests by URL

The `take(10)` action triggers sorting followed by fetching the top results. This job includes:

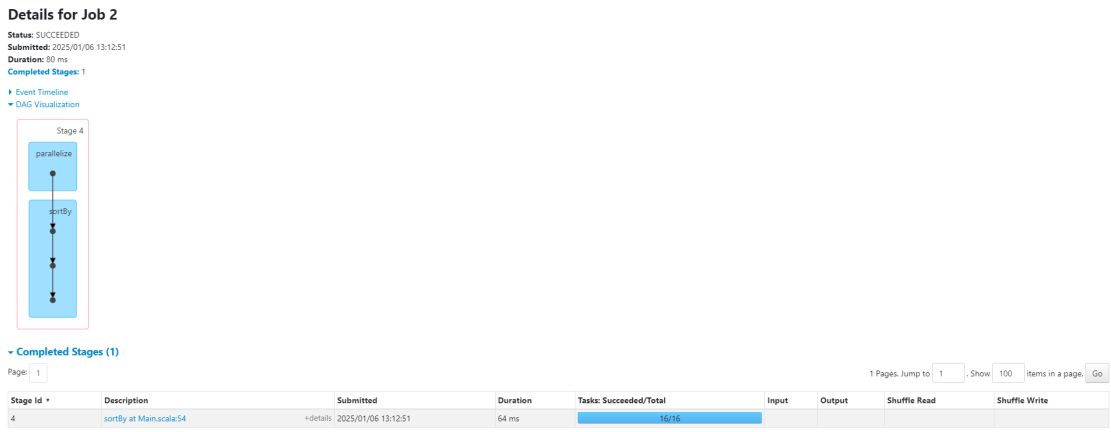


Figure 20: Action 3 (1)

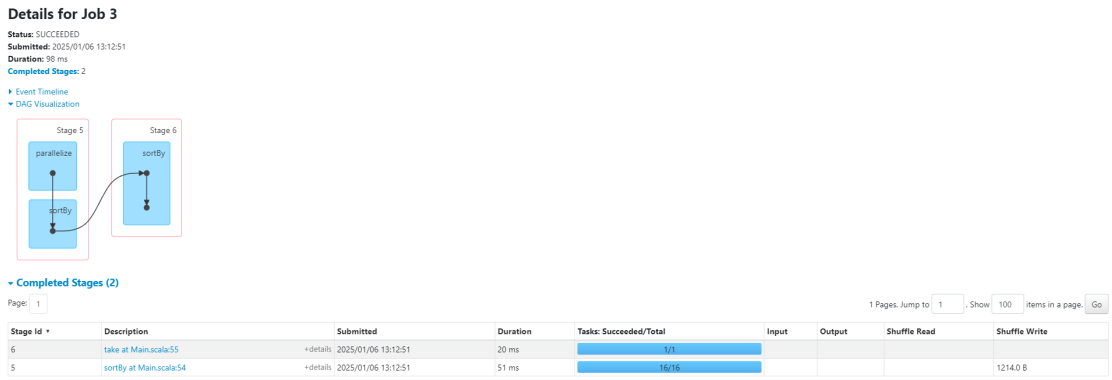


Figure 21: Action 3 (2)

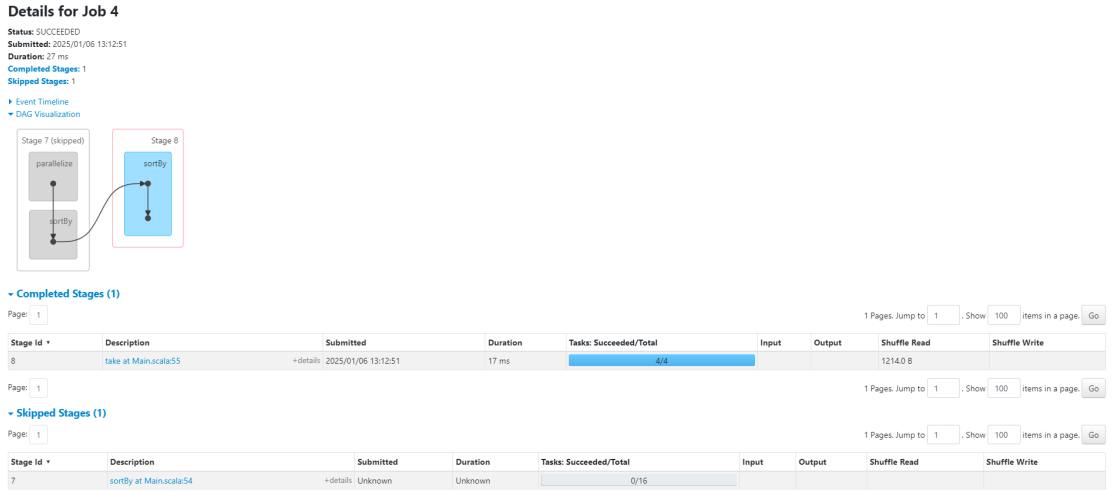


Figure 22: Action 3 (3): Trigger by `take()`

- Parallelizes the RDD.
- Shuffles the data to sort them by count (descending) **(Wide stage)**.

4. Saving results to HDFS

The `saveAsTextFile` action creates a job with tasks to write the partitioned results to the specified HDFS directory.

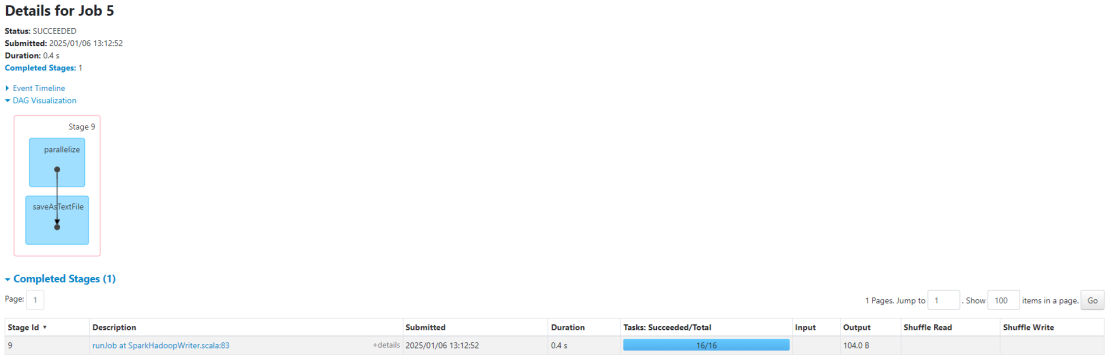


Figure 23: Action 4: Trigger by `saveAsTextFile()`

Each stage in a job is further divided into tasks, and each executor is allocated resources by YARN to execute these tasks within a stage. Note that a stage must be completed before the job can proceed to the next stage.

In this example, the application splits the RDD into five partitions (i.e., the RDD is divided into five parts) to enable more efficient parallel execution. Each worker utilizes its allocated resources to process some of the RDD partitions. The number of partitions processed by a worker depends on the number of virtual CPU cores and the cluster manager's resource management.

Since the RDD is divided into five partitions, each stage is divided into five tasks. The number of partitions is determined using the `getNumPartitions` method on the RDD `parsedLogs`. This number can be increased or decreased using the `repartition` or `coalesce` methods.

3 Principle of Apache Spark' Data Structures

3.1 Principle of RDD

An RDD is an abstraction of distributed data with several interesting features. It is immutable, meaning that once it is created, it cannot be modified. This immutability is beneficial for resilience; when executing operations that change an RDD, Spark creates a new one and keeps track of its lineage. For example, if a RDD1 is initialized and the program wants to convert some fields from character strings to numbers, the new RDD (RDD2) will contain these modifications while also remembering that they were derived from RDD1. If the creation of RDD2 fails, Spark can revert to RDD1 instead of having to read the data from disk again.

Figure 24: Create a new RDD2 based on some requirements from RDD1

A big question arises, if all these copies consume too much memory or not? While that's a possibility, Spark manages memory efficiently and removes unnecessary data. For instance, if some requirements requires the creation RDD3 from RDD2, Spark may not need to keep RDD1 around anymore, because while RDD2 is correct, RDD3 just need to be derived from RDD2.

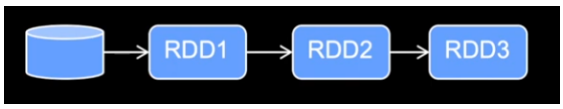


Figure 25: Remove RDD1 from memory

However, the situation can be more complex if RDD4 is created based on RDD1, leading to a tree-like structure of RDDs. In this case, RDD1 must be kept in memory, since if RDD4 fails, Spark can go back from RDD1 and re-execute again.

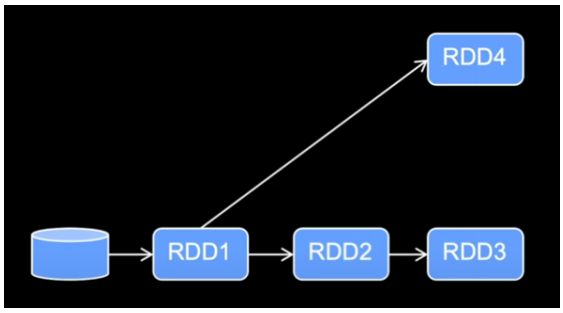


Figure 26: Create RDD4 from RDD1

A question arises: why does Spark separate methods into transformations and actions? The answer lies in a key feature of Spark, which is lazy evaluation. When an action is called, Spark

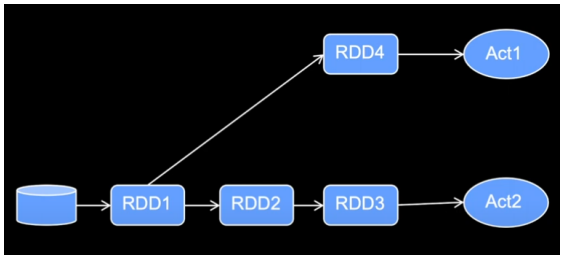


Figure 27: Principle of action and transformations

begins evaluating its previous transformations. In the above example, by applying this feature to `action 2`, the creation of RDD2 and RDD3 can be merged to reduce the amount of data stored in memory. This merging is evaluated by a Directed Acyclic Graph (DAG), which is a powerful data structure in Spark used to create a plan for performing transformations for each action. This process allows Spark to optimize the execution path by analyzing all transformations and determining the most efficient way to achieve the desired outcome.

3.2 Principle of DataFrame

In Apache Spark, a DataFrame represents a high-level abstraction built on top of RDDs (Resilient Distributed Datasets). Unlike RDDs, which consist of unstructured data, DataFrames offer a structured, table-like format with named columns and defined data types resembling SQL tables. Internally, a DataFrame is reliant on one or more RDDs as its underlying foundation. When operations, such as filtering or selection, are conducted on a DataFrame, Spark automatically translates these queries into RDD transformations and actions, effectively integrating structure with efficiency.

A DataFrame does not store data directly; instead, it serves as a structured interface to access the underlying RDDs. The raw data is maintained within these RDDs, while the schema, which includes column names and data types, is stored as metadata that defines the data's structure. For instance, if an RDD holds the data [(1, "Alice"), (2, "Bob")] and is transformed into a DataFrame with columns labeled "id" and "name," the DataFrame provides a structured representation of the data contained in the RDD.

When operations such as filter or select occur on a DataFrame, Spark utilizes the Catalyst Optimizer. This system translates the operations into a logical plan, optimizes it, and subsequently converts it into a physical execution plan designed to run on the cluster. The final physical plan operates via the RDD API, applying transformations like map and filter to process the data. For example, a query like `df.filter($"age" > 25).select("name")` is parsed and optimized by Catalyst before being executed as RDD transformations. This integration empowers DataFrames to deliver structured processing capabilities while preserving the scalability and resilience characteristic of RDDs.

Catalyst Optimizer

The **Catalyst Optimizer** is a vital component of Spark SQL that enhances the performance of queries executed on DataFrames and Datasets. It operates through four main stages, each contributing to efficient query execution.

1. Logical Plan

The **Logical Plan** represents the initial steps of query execution on a DataFrame or Dataset and consists of two sub-stages:

- **Unresolved Logical Plan**: This is the initial representation where columns or tables may not yet be defined.
- **Resolved Logical Plan**: After schema validation, Spark refines the plan into a resolved state with accurate definitions for columns and tables.

Example: Consider the following Scala code:

```
val df = spark.read.json("data.json")
val result = df.filter($"age" > 25).select("name")
```

The Logical Plan for this query includes:

- A filter operation to select rows where `age > 25`.
- A projection to select the `name` column.

2. Optimized Logical Plan

In this stage, Catalyst applies rule-based optimizations to improve the Logical Plan. Common optimization techniques include:

- **Predicate Pushdown**: Moving filter conditions closer to the data source to minimize unnecessary data processing.
- **Column Pruning**: Selecting only the required columns for processing.
- **Constant Folding**: Precomputing static expressions involving constants.
- **Projection Pruning**: Discarding unnecessary columns at each step of the query.

Example: For the query:

```
val result = df.filter($"age" > 25).select("name")
```

Catalyst optimizes the plan by:

- Pushing the `age > 25` filter to the data source (if supported).
- Reading only the `age` and `name` columns instead of the entire dataset.

3. Physical Plan

The **Physical Plan** specifies how the query will be executed on the cluster. It employs physical operators, such as **HashAggregate** and **Exchange**, to define the execution steps. Among multiple possible Physical Plans, Spark selects the optimal one using **Cost-Based Optimization (CBO)**:

- Catalyst estimates the cost of each plan based on data statistics.
- The plan with the lowest cost is chosen for execution.

Example: When performing a join operation, Spark might:

- Use a **Broadcast Join** if one table is small.
- Opt for a **Sort-Merge Join** for larger tables.

4. Code Generation (Tungsten Execution Engine)

The final step involves generating executable code for the optimized Physical Plan. Utilizing the **Tungsten Execution Engine**, Catalyst creates efficient Java bytecode to minimize runtime overhead and maximize computational efficiency.

By following these stages, the Catalyst Optimizer empowers Spark to execute queries efficiently, blending ease of use with powerful optimizations suitable for large-scale data processing.

4 Modules of Apache Spark

4.1 Core Module

The Core Module is the foundation of Apache Spark and provides the essential building blocks for all other modules. It handles the basic functionalities required for distributed data processing, such as:

- **Distributed Task Scheduling:** Manages the allocation of tasks across a cluster of nodes.
- **RDD API:** Resilient Distributed Datasets (RDDs) are immutable distributed collections of objects. The RDD API allows transformations (e.g., map, filter) and actions (e.g., collect, count) for distributed data manipulation.
- **Fault Tolerance:** Automatic recovery from node failures using lineage information.
- **Memory Management:** Efficient in-memory computation and storage.

The Core Module also supports input and output operations with various file systems, including HDFS, S3, and local file systems.

4.2 SQL Module (Spark SQL)

The SQL Module enables structured data processing using SQL and DataFrame APIs. It integrates seamlessly with relational databases and other data sources. Key features include:

- **DataFrames and Datasets:**
  1. DataFrames are distributed collections of data organized into named columns.
  2. Datasets provide a type-safe, object-oriented programming interface.
- **Query Optimization:** The Catalyst Optimizer automatically optimizes queries for better performance.
- **Integration with BI Tools:** Spark SQL can act as a distributed SQL query engine for tools like Tableau or Power BI.
- **Support for Standard Formats:** Works with formats like JSON, Parquet, ORC, and Avro.

4.3 Streaming Module (Structured Streaming)

The Streaming Module allows Apache Spark to process real-time data streams. It enables processing of live data and integrates seamlessly with Spark's batch processing capabilities. Key features of Structured Streaming include:

- **Unified batch and streaming API:** Both batch and streaming workloads can be handled using the same DataFrame and Dataset API, making it easier to develop real-time applications.
- **Event time processing:** Spark can process late-arriving data based on event time, handling out-of-order data.

- **Fault tolerance:** It supports checkpointing, write-ahead logs, and other mechanisms to ensure data consistency and fault tolerance.
- **Sources and sinks:** Structured Streaming supports various data sources, including Kafka, file systems, and socket streams, with the ability to output results to consoles, files, and external systems like databases.

4.4 Machine Learning Module (MLlib)

The MLlib Module is Spark's library for scalable machine learning. It provides algorithms and tools for classification, regression, clustering, and collaborative filtering. Some key features include:

- **Machine learning algorithms:** Spark includes a wide range of machine learning algorithms such as decision trees, logistic regression, K-means, and ALS (Alternating Least Squares).
- **Pipelines:** MLlib includes utilities for building machine learning pipelines, which allow for the automation of model training and evaluation.
- **Feature engineering:** It offers tools for feature extraction, transformation, and selection.
- **Model persistence:** Supports saving and loading models, making it easy to deploy models in production.

4.5 Graph Processing Module (GraphX)

The GraphX Module enables graph processing and analysis in Spark. It provides an API for working with graphs, which are composed of vertices (nodes) and edges (relationships). Key features include:

- **Graph representation:** GraphX provides a graph abstraction for representing graphs as vertex and edge RDDs.
- **Built-in algorithms:** It includes common graph algorithms such as PageRank, connected components, and triangle counting.
- **RDD integration:** GraphX utilizes RDDs for efficient distributed computation, which helps in processing large-scale graphs.

5 Demonstration for Each Module (Mostly Spark SQL)

5.1 Spark Session

In order to start working on Apache Spark, a *SparkSession* is created to connect to the Spark Cluster. Spark Session controls communicating with *SparkContext* - main executor of Apache Spark.

To created a Spark Session, the following code can be executed:

```
1    val spark = SparkSession
2      .builder()
3      .master("yarn") // maybe local[*], mesos, k8s
4      .appName("Testing")
5      .getOrCreate()
```

For the above code, there are a few methods to note:

- *builder*(): This initializes the builder for creating a SparkSession. The builder is a configuration utility that allows the user to specify how your SparkSession should be set up. It's the starting point for setting later parameters.
- *master*(): This specifies the cluster manager that Spark should use to execute your application. The above code indicates that Spark will run on a $YARN$ (Yet Another Resource Negotiator) cluster. Besides YARN, there are many cluster managers to use, such as $local[*]$, $mesos$, $k8s$.
- *appName*(): This method defines the name of the Spark Application. This name is displayed in the Spark UI and hels identify the job among others running on the cluster.
- *getOrCreate*(): In order not to accidentally create multiple SparkSession instances in the same application, this method is used to check if a SparkSession already exists, it will reuse that instead of creating a new session.

5.2 Spark SQL Module

Spark provides *Spark SQL*, which is a powerful module used for retrieving useful information from structured data. This section covers some general methods for each functionality in Spark SQL.

In order to demonstrate the following methods, some values should be declared before testing. First, the data collected from a 1800.*csv* file is used for testing. This dataset contains 1800 rows, with ... columns.

```
ITE00100554,18000101,TMAX,-75,,,E,
ITE00100554,18000101,TMIN,-148,,,E,
GM000010962,18000101,PRCP,0,,,E,
EZE00100082,18000101,TMAX,-86,,,E,
EZE00100082,18000101,TMIN,-135,,,E,
ITE00100554,18000102,TMAX,-60,,I,E,
ITE00100554,18000102,TMIN,-125,,,E,
GM000010962,18000102,PRCP,0,,,E,
EZE00100082,18000102,TMAX,-44,,,E,
EZE00100082,18000102,TMIN,-130,,,E,
ITE00100554,18000103,TMAX,-23,,,E,
ITE00100554,18000103,TMIN,-46,,I,E,
GM000010962,18000103,PRCP,4,,,E,
EZE00100082,18000103,TMAX,-10,,,E,
EZE00100082,18000103,TMIN,-73,,,E,
ITE00100554,18000104,TMAX,0,,,E,
ITE00100554,18000104,TMIN,-13,,,E,
```

Figure 28: Dataset used for demonstration

Note that for this demonstration, *Scala* - the main API of Apache Spark - is used to implement these later methods. Apache Spark provides a powerful data structure for Scala, which is the *Dataset* (The definition of the *Dataset* in this context is different from *Dataset* in common situations. For convenience, the data structure will be called *DS*). In this demonstration, a DS is created by using a case class and a schema:

```
1  case class Temperature(id: String, date: Int, measure: String, temp:
       Double)
2
3  import spark.implicits._
4
5  val schema = new StructType()
6    .add("id", StringType, nullable = true)
7    .add("date", IntegerType, nullable = true)
8    .add("measure", StringType, nullable = true)
9    .add("temp", FloatType, nullable = true)
10
```

```
val data = spark.read
  .schema(schema)
  .csv("hdfs:///user/group1/Alan_DF/1800.csv")
  .as[Temperature]
```

5.2.1 Methods for Manipulating Data

5.2.1.a SQL query

Spark SQL provides a method for passing a SQL query to manipulate the data. But before working on a SQL query, a global (or temporary) view should be created to work for SQL queries. Note that a global view can be used on multiple sessions of a Spark application, while a temporary view can only be used on the session that create it.

```
data.createOrReplaceTempView("Temperature")
spark.sql("SELECT * FROM Temperature WHERE measure LIKE '%TMIN%'")
```

```
+-----------+--------+-------+------+
|         id|    date|measure|  temp|
+-----------+--------+-------+------+
|ITE00100554|18000101|   TMIN|-148.0|
|EZE00100082|18000101|   TMIN|-135.0|
|ITE00100554|18000102|   TMIN|-125.0|
|EZE00100082|18000102|   TMIN|-130.0|
|ITE00100554|18000103|   TMIN| -46.0|
|EZE00100082|18000103|   TMIN| -73.0|
|ITE00100554|18000104|   TMIN| -13.0|
|EZE00100082|18000104|   TMIN| -74.0|
|ITE00100554|18000105|   TMIN|  -6.0|
|EZE00100082|18000105|   TMIN| -58.0|
|ITE00100554|18000106|   TMIN|  13.0|
|EZE00100082|18000106|   TMIN| -57.0|
|ITE00100554|18000107|   TMIN|  10.0|
|EZE00100082|18000107|   TMIN| -50.0|
|ITE00100554|18000108|   TMIN|  14.0|
|EZE00100082|18000108|   TMIN| -31.0|
|ITE00100554|18000109|   TMIN|  23.0|
|EZE00100082|18000109|   TMIN| -46.0|
|ITE00100554|18000110|   TMIN|  31.0|
|EZE00100082|18000110|   TMIN| -75.0|
+-----------+--------+-------+------+
only showing top 20 rows
```

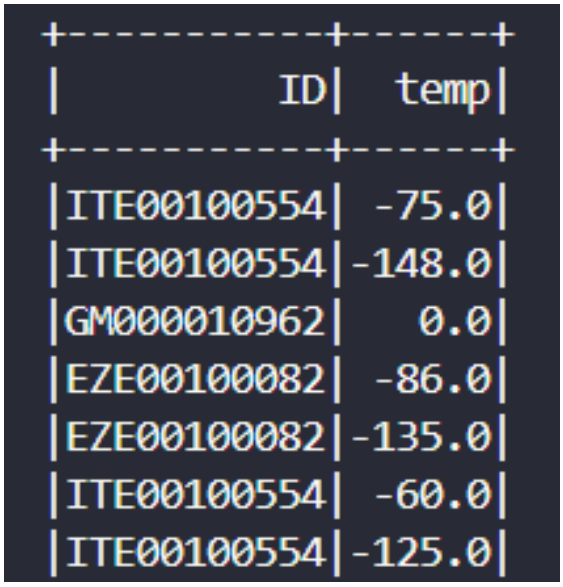
Figure 29: Result when using normal SQL query on Apache Spark

5.2.1.b *SELECT* methods

Instead of writing a whole SQL query, Spark provides separate methods for retrieving multiple kinds of data in a dataset.

```
data.select("id", "temp").show()
data.selectExpr("id AS ID", "temp").show()
```

In the first line, the method takes multiple parameters, which is the string of the name of the column specified in the previous case class. On the other hand, the second method takes parameters similar to the first method, but can rename the returned column inside each string, which helps the user have a clearer view about the returned table.



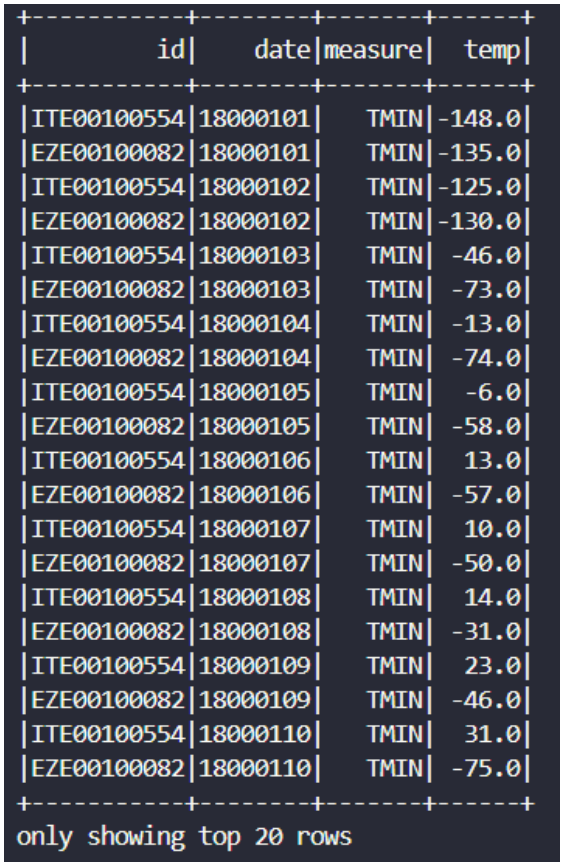
ID	temp
ITE00100554	-75.0
ITE00100554	-148.0
GM000010962	0.0
EZE00100082	-86.0
EZE00100082	-135.0
ITE00100554	-60.0
ITE00100554	-125.0

Figure 30: Result when applying SELECT methods on *id* and *temp* column

5.2.1.c Conditional methods

Spark SQL provides methods for filtering columns based on some criteria. In the code, the data is filtered by checking for value "TMIN" in column *measure* in the table. The first line of code used an lambda function to check for each record in the table, while the second code displays the method in a similar manner when writing a *WHERE* clause in SQL.

```
data.filter(row => row.measure == "TMIN").show()
data.where("measure == 'TMIN'").show()
```



id	date	measure	temp
ITE00100554	18000101	TMIN	-148.0
EZE00100082	18000101	TMIN	-135.0
ITE00100554	18000102	TMIN	-125.0
EZE00100082	18000102	TMIN	-130.0
ITE00100554	18000103	TMIN	-46.0
EZE00100082	18000103	TMIN	-73.0
ITE00100554	18000104	TMIN	-13.0
EZE00100082	18000104	TMIN	-74.0
ITE00100554	18000105	TMIN	-6.0
EZE00100082	18000105	TMIN	-58.0
ITE00100554	18000106	TMIN	13.0
EZE00100082	18000106	TMIN	-57.0
ITE00100554	18000107	TMIN	10.0
EZE00100082	18000107	TMIN	-50.0
ITE00100554	18000108	TMIN	14.0
EZE00100082	18000108	TMIN	-31.0
ITE00100554	18000109	TMIN	23.0
EZE00100082	18000109	TMIN	-46.0
ITE00100554	18000110	TMIN	31.0
EZE00100082	18000110	TMIN	-75.0

only showing top 20 rows

Figure 31: Filtering on column *measure*

5.2.1.d Aggregate methods

Similar to SQL, SparkSQL provides methods for grouping attributes in order to calculate some aggregate functions such as COUNT, MIN, MAX, AVG, SUM,... In the code, the data is grouped based on the column *measure*, and the column *temp* is chosen to calculate the average value. Note that we can group many columns or calculate many aggregate functions on many columns by adding more columns (as string) in the method (*groupBy* and *agg*, respectively).

```
1 data.groupBy("measure").agg(avg("temp")).show()
```

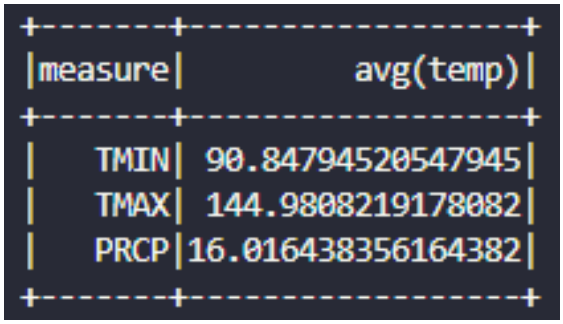


Figure 32: Filtering on column *measure*

5.2.1.e Join and Union methods

* Join Methods:

We can join two tables in SparkSQL to retrieve multiple data at the same time. By using *join* method, we can simply connect the two tables based on the value of the column(s).

```
val another_data = data

data.join(another_data, data.col("id") === another_data.col("id"), "inner
    ").show()

data.joinWith(another_data, data.col("id") === another_data.col("id"), "
    inner").show()
```

The above code returns tables when joining two tables with the same column *id*. Note that for join methods, we can specify multiple kinds of joins (inner join, outer join, left join, right join, cross join) by adding a type of join (as string) as the last parameter of the method.

The first line of code returns a table with join records having the same value of column *id* (similar to the representation of the table in SQL); on the other hand, the remaining line returns a table with only two columns corresponding to 2 joining tables, with each row has the same value of *id*. Each value in each column will contain the value of the record corresponding to that table, as shown in Figure...

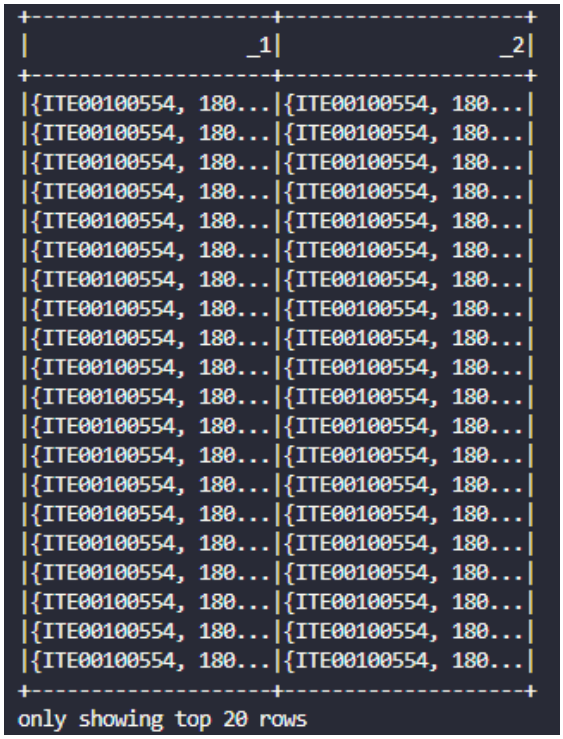


Figure 33: Result by using *joinWith* method on column *id*

* Union Methods:

Spark also provides Union method for merging data from multiple tables to create a new table containing all the values. Note that in order to use the Union, the tables participating in the Union method must be type compatible, which means they have to have the same number (and same order) of columns, and each column has the same data type with one another.

```
val df1 = Seq((1, 2, 3)).toDF("col0", "col1", "col2")
val df2 = Seq((1, 2, 3)).toDF("col1", "col4", "col0")
val df3 = Seq((1, 2, 3)).toDF("col0", "col1", "col2")
df1.union(df3).show
```

The result of the *union* method will be all the values of the two tables in $df1$ and $df3$. Note that UNION in Spark SQL will not remove duplicate values.

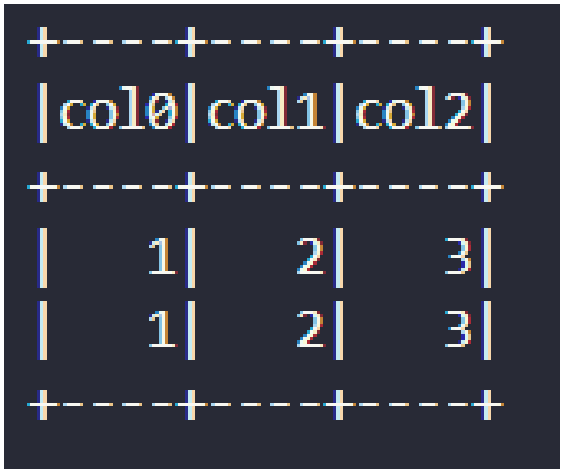


Figure 34: Union of the two tables

On the other hand, if some users still want to union two tables but with different order of columns, *unionByName* can help identifying the columns by checking the column names and union them. If two columns don't match their names, a boolean variable *allowMissingColumns* can be used to determine whether the two columns can be kept in the result or not.

```
df1.unionByName(df2, allowMissingColumns = true).show()
```

The result of the code is as following figure:

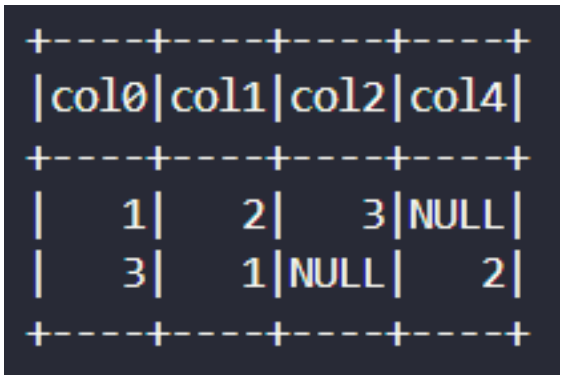


Figure 35: Union two tables by using column names

5.2.2 Descriptive methods

Similar to many API for statistics, Scala API in Spark provides many methods for pre-processing and summarizing data.

```
data.summary().show()
```

The above code returns the overall information about the dataset, includes the count, mean, standard deviation,... as the shown figure:

summary	id	date	measure	temp
count	1825	1825	1825	1825
mean	NULL	1.800066832328767E7	NULL	97.53479452054795
stddev	NULL	345.0964453798429	NULL	93.40225576440513
min	EZE00100082	18000101	PRCP	-148.0
25%	NULL	18000402	NULL	11.0
50%	NULL	18000702	NULL	85.0
75%	NULL	18001001	NULL	175.0
max	ITE00100554	18001231	TMIN	323.0

Figure 36: Summarizing a dataset

Besides summarizing, Spark SQL provides many methods for dealing with N/A (or NULL) values. The following code is used to fill N/A values with value 15.

```
1 data.na.fill(15).show()
```

5.2.3 Display methods

In order to print out the records into the console, several methods can be used. The following methods will help printing out the values into the console, but in a different manner.

```
1 data.show(10)
2
3 data.head(10).foreach(row => println(s"This row has temperature ${row.
      temp} with type ${row.measure}"))
4
5 println(data.first)
```

- The first line uses *show*() method, which displays the first 10 rows directly into the console
- The second line uses *head*() method, which returns the first 10 rows as an array of rows, and to print them, a search through each value to call the *println*() method is necessary.
- The last line uses *first*() method, which returns only the first row of the dataset as 1 row, and to print them, *println*() method is required.

```
This row has temperature -75.0 with type TMAX
This row has temperature -148.0 with type TMIN
This row has temperature 0.0 with type PRCP
This row has temperature -86.0 with type TMAX
This row has temperature -135.0 with type TMIN
This row has temperature -60.0 with type TMAX
This row has temperature -125.0 with type TMIN
This row has temperature 0.0 with type PRCP
This row has temperature -44.0 with type TMAX
This row has temperature -130.0 with type TMIN
```

Figure 37: Results when applying the second line of code

5.2.4 System controlling methods

To improve the efficiency while working with big data, a good strategy in storage management is required to achieve best performance. In SparkSQL, some methods can help keeping some parts of the dataset in memory/disk

```
data.cache()
data.persist(StorageLevel.MEMORY_AND_DISK)
```

In the first code, all the rows of the *value* will be kept in the main memory (if possible), to improve the runtime of the application while using the data. However, if the data set is too large to fit in the memory, the remaining data will be kept in the disk (but leads to low performance). Apart from that, the second code use *persist*(), which allows the user to specify whether the data should be kept. In this example, the data will be kept in memory and disk.

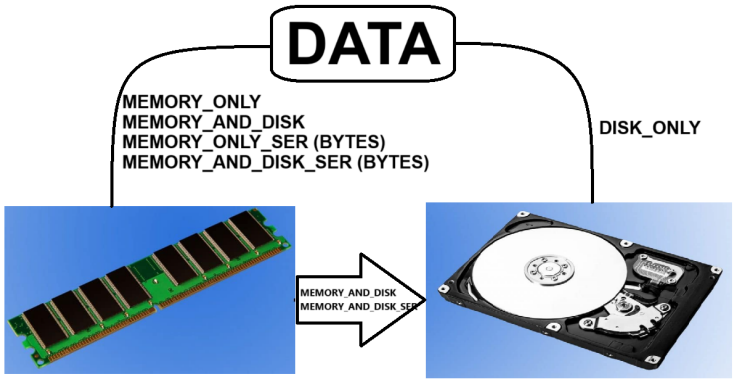


Figure 38: Flow of Data using `persist()`

When using `persist()` with multiple options, the data is kept in multiple formats and places in the system:

- `MEMORY_ONLY`: The data is kept only in memory. If the data is too large to fit in the memory, the left will be kept in the disk –> Only use this option when the data is small and the main memory is large enough to contain all the data.

- `MEMORY_AND_DISK`: The data is kept in memory. If the memory cannot fit all the data, the remaining will be kept in the disk (but this option ensures efficient processing in working with memory and disk when comparing with `MEMORY_ONLY`
- `MEMORY_ONLY_SER`: The data is kept into the memory only, a similar manner with `MEMORY_ONLY`. The main difference is that the data before being kept in memory, it is serialized, meaning they will be converted into an array of bytes. This option reduces the amount of bytes each object being kept in the memory by eliminating unnecessary information (references, methods,...) from its object.
- `MEMORY_AND_DISK_SER`: This option works analogical with option `MEMORY_AND_DISK` and serialize similar to `MEMORY_ONLY_SER`.
- `DISK_ONLY`: The data will be kept in the disk only.

5.3 Spark MLlib Module

MLlib, short for Machine Learning Library, is a scalable and distributed machine learning framework provided by Apache Spark. Designed to handle large-scale data processing, MLlib offers a variety of high-performance machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as tools for dimensionality reduction, feature extraction, and model evaluation. Built on Spark's distributed computing capabilities, MLlib can efficiently process data in parallel across a cluster, making it suitable for big data applications. It supports APIs in multiple programming languages such as Scala, Java, Python, and R, allowing developers to seamlessly integrate machine learning into their Spark workflows. Additionally, MLlib provides compatibility with Spark's DataFrame and RDD abstractions, enabling easy data preprocessing and analysis. Its versatility and performance make it a popular choice for building machine learning pipelines in distributed environments.

This section covers a quick demonstration of MLlib Module in Apache Spark, which is Linear Regression Model on a DataFrame.

5.3.1 Linear Regression in Spark MLlib Module

```
1  object LinearRegressionDataFrameDataset {
2
3      case class RegressionSchema(label: Double, features_raw: Double)
4
5      /** Our main function where the action happens */
6      def main(args: Array[String]) {
7          // Set the log level to only print errors
8          Logger.getLogger("org").setLevel(Level.ERROR)
9
10         val spark = SparkSession.builder
11         .appName("LinearRegressionDF")
12         .master("local[*]")
13         .getOrCreate()
14
15         // Load up our page speed / amount spent data in the format
               required by MLLib
16         // (which is label, vector of features)
17
```

```
18          // In machine learning lingo, "label" is just the value you're
                trying to predict, and
19          // "feature" is the data you are given to make a prediction with.
                 So in this example
20          // the "labels" are the first column of our data, and "features"
                are the second column.
21          // You can have more than one "feature" which is why a vector is
                required.
22          val regressionSchema = new StructType()
23          .add("label", DoubleType, nullable = true)
24          .add("features_raw", DoubleType, nullable = true)
25
26          import spark.implicits._
27          val dsRaw = spark.read
28          .option("sep", ",")
29          .schema(regressionSchema)
30          .csv("hdfs:///user/group1/Alan_DF/regression.txt")
31          .as[RegressionSchema]
32
33          val assembler = new VectorAssembler()
34          .setInputCols(Array("features_raw"))
35          .setOutputCol("features")
36          val df = assembler
37          .transform(dsRaw)
38          .select("label", "features")
39
40          // Let's split our data into training data and testing data
41          val trainTest = df.randomSplit(Array(0.5, 0.5))
42          val trainingDF = trainTest(0)
43          val testDF = trainTest(1)
44
45          // Now create our linear regression model
46          val lir = new LinearRegression()
47          .setRegParam(0.3) // regularization
48          .setElasticNetParam(0.8) // elastic net mixing
49          .setMaxIter(100) // max iterations
50          .setTol(1e-6) // convergence tolerance
51
52          // Train the model using our training data
53          val model = lir.fit(trainingDF)
54
55          // Now see if we can predict values in our test data.
56          // Generate predictions using our linear regression model for all
                features in our
57          // test dataframe:
58          val fullPredictions = model.transform(testDF).cache()
59
60          // This basically adds a "prediction" column to our testDF
                dataframe.
61
62          // Extract the predictions and the "known" correct labels.
63          // val predictionAndLabel = fullPredictions.select("prediction",
                "label").collect()
```

```
64
65          // // Print out the predicted and actual values for each point
66          // for (prediction <- predictionAndLabel) {
67          //   println(prediction)
68          // }
69
70          fullPredictions.selectExpr("*").show()
71
72          // Stop the session
73          spark.stop()
74
75      }
76  }
```

5.3.2 Detailed Explanation

The above code is designed to implement a linear regression model using Apache Spark's MLlib and focuses on an end-to-end machine learning workflow from data preparation to model evaluation. The program starts by importing essential libraries such as `org.apache.spark.sql` for working with Spark SQL, `org.apache.spark.ml.feature.VectorAssembler` for feature transformations, and `org.apache.spark.ml.regression.LinearRegression` for implementing the regression model. Logging utilities are also imported to control the verbosity of execution messages.

The `LinearRegressionDataFrameDataset` object defines the application, and within it, a case class `RegressionSchema` is declared. This case class serves as a schema definition for the data being processed, consisting of two fields: `label` (the target variable to predict) and `features_raw` (the independent variable used for prediction). This structure helps create a strongly typed Dataset in Spark.

The `main` function begins by configuring Spark's logging level to suppress non-error messages, ensuring a cleaner console output. A `SparkSession` is then initialized using the `builder` pattern. The session is named *"LinearRegressionDF"* and is set to run locally with all available CPU cores using the `master("local[*]")` configuration. The `getOrCreate()` method ensures that a Spark session is either created or reused if one already exists.

Next, the schema for the input data is defined explicitly using `StructType`. It specifies two columns: `label`, which represents the dependent variable, and `features_raw`, which represents the independent variable. Both columns are declared as `DoubleType`. The data is loaded from a CSV file stored on HDFS at `hdfs:///user/group1/Alan_DF/regression.txt`. The `spark.read` method is used with the specified schema and delimiter options, and the resulting DataFrame is converted into a strongly typed Dataset using the `as[RegressionSchema]` method.

The program then prepares the data for use in a machine learning model. In Spark MLlib, features must be represented as a single vector column. To achieve this, a `VectorAssembler` is configured to take the `features_raw` column as input and produce a new column called `features`. This transformation is applied to the raw Dataset, and the resulting DataFrame contains only the `label` and `features` columns, which are required for the regression model.

To enable model training and evaluation, the data is split into two parts: a training set and a testing set. The `randomSplit` method divides the data into two subsets, with 50% allocated to

each. The training dataset is used to fit the regression model, while the testing dataset is used to evaluate its predictive performance.

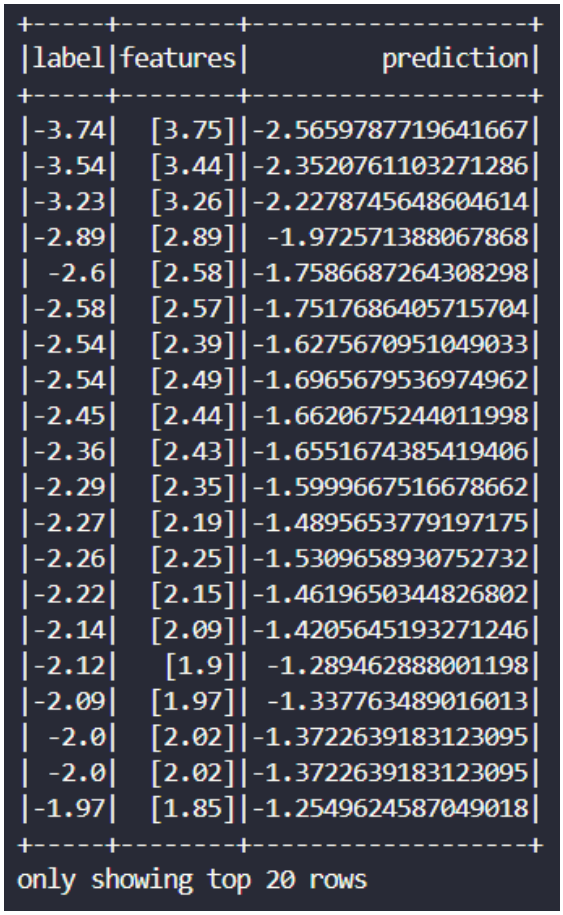
A linear regression model is then initialized with specific hyperparameters. These include a regularization parameter (`regParam`) set to 0.3 to prevent overfitting, an elastic net mixing parameter (`elasticNetParam`) set to 0.8 to control the balance between L1 and L2 regularization, a maximum of 100 optimization iterations (`maxIter`), and a convergence tolerance (`tol`) of 1×10^{-6}. The model is trained using the `fit` method on the training DataFrame.

After the model is trained, predictions are generated for the testing dataset using the `transform` method. This adds a new `prediction` column to the test DataFrame, containing the model's predicted values for each row. To optimize performance, the resulting DataFrame is cached in memory using the `cache` method. The predictions and their corresponding labels are displayed in a tabular format using the `selectExpr("*")` and `show()` methods, providing a view of the model's outputs alongside the original data.

The program concludes by stopping the Spark session using the `spark.stop()` method to release resources. This step ensures that all Spark-related processes terminate gracefully. Overall, the code demonstrates the process of building, training, and evaluating a linear regression model, while also highlighting Spark's ability to handle large-scale, distributed machine learning workflows.

5.3.3 Result

After applying Linear Regression model based on the given dataset, the application returns the result as the figure below.



label	features	prediction
-3.74	[3.75]	-2.5659787719641667
-3.54	[3.44]	-2.3520761103271286
-3.23	[3.26]	-2.2278745648604614
-2.89	[2.89]	-1.972571388067868
-2.6	[2.58]	-1.7586687264308298
-2.58	[2.57]	-1.7517686405715704
-2.54	[2.39]	-1.6275670951049033
-2.54	[2.49]	-1.6965679536974962
-2.45	[2.44]	-1.6620675244011998
-2.36	[2.43]	-1.6551674385419406
-2.29	[2.35]	-1.5999667516678662
-2.27	[2.19]	-1.4895653779197175
-2.26	[2.25]	-1.5309658930752732
-2.22	[2.15]	-1.4619650344826802
-2.14	[2.09]	-1.4205645193271246
-2.12	[1.9]	-1.289462888001198
-2.09	[1.97]	-1.337763489016013
-2.0	[2.02]	-1.3722639183123095
-2.0	[2.02]	-1.3722639183123095
-1.97	[1.85]	-1.2549624587049018

only showing top 20 rows

Figure 39: Applying Linear Regression Model in Apache Spark

5.4 GraphX

5.4.1 Introduction

GraphX is a powerful library within Apache Spark that enables distributed graph processing at scale. It bridges the gap between graph processing and data processing by combining Spark's distributed data processing capabilities with a graph computation framework. GraphX allows users to work with graph structures (vertices and edges) seamlessly alongside RDDs (Resilient Distributed Datasets) and DataFrames.

The library is designed to handle large-scale graphs and perform computations such as PageRank, connected components, and triangle counting efficiently. It offers a user-friendly API for defining graphs, running custom graph algorithms, and performing analytics on graph-based data. GraphX is particularly suited for use cases like social network analysis, recommendation systems, and fraud detection.

GraphX provides a range of built-in functions and abstractions to work with graph data. These include:

1. Graph Construction

- **Graph Class:** The primary abstraction representing a directed graph where each vertex has an associated attribute, and each edge has an associated attribute.
- **Graph Creation Functions:**
  - `Graph.fromEdges`: Creates a graph from an edge RDD.
  - `Graph.fromEdgesAndVertices`: Builds a graph using both vertex and edge RDDs.

2. Graph Transformations

- `mapVertices`: Applies a function to transform the attributes of the vertices.
- `mapEdges`: Applies a function to transform the attributes of the edges.
- `reverse`: Reverses the direction of all edges in the graph.
- `subgraph`: Extracts a subgraph by filtering vertices and edges based on user-defined predicates.

3. Graph Algorithms

GraphX includes several built-in graph algorithms to simplify complex analytics:

- **PageRank:** Measures the importance of each vertex in the graph (e.g., for web link analysis).
- **Connected Components:** Identifies all the connected subgraphs in an undirected graph.
- **Shortest Path:** Calculates the shortest path between vertices.
- **Triangle Count:** Computes the number of triangles passing through each vertex.
- **Label Propagation:** Used for community detection in graphs.

4. Structural Operations

- `triplets`: Generates an RDD of triplets, where each triplet contains a source vertex, a destination vertex, and the edge connecting them. This is particularly useful for relational computations.
- `aggregateMessages`: Aggregates information from neighboring vertices and edges to compute properties like in-degree, out-degree, or custom metrics.

5. Graph Queries

- **Vertex and Edge Queries:**
  - `vertices`: Returns the RDD of all vertices in the graph.
  - `edges`: Returns the RDD of all edges.
- **Degree Computation:**
  - `inDegrees`: Computes the number of incoming edges for each vertex.
  - `outDegrees`: Computes the number of outgoing edges for each vertex.
  - `degrees`: Computes the total degree (in + out) for each vertex.

5.4.2 An application of GraphX

This section demonstrates the use of GraphX in analysing social network data. Specifically, it focuses on identifying common friends between pairs of users based on a graph representation of relationships. In this context, each user is represented as a vertex, and the relationships (friendships) between users are represented as edges in the graph.

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

// Initialize SparkContext
val sc = spark.sparkContext

// Task 1: Load edges from CSV file
val edgesPath = "/path/to/edges.csv" // Replace with your file path
val edgesFile = sc.textFile(edgesPath)

// Task 2: Construct a graph
// Parse edges and create RDD of Edge
val edges: RDD[Edge[Int]] = edgesFile
  .filter(!_.contains("User1")) // Skip header row
  .map { line =>
    val Array(user1, user2) = line.split(",").map(_.trim.toInt)
    Edge(user1, user2, 1) // Attribute '1' to indicate relationship
  }

// Create vertices RDD
val vertices: RDD[(VertexId, String)] = edges
  .flatMap(edge => Seq(edge.srcId, edge.dstId))
  .distinct()
  .map(id => (id, s"User_$id"))

// Create GraphX graph
val graph = Graph(vertices, edges)

// Reverse the graph to ensure bidirectional relationships
// val reversedGraph = graph.reverse

// Task 3: Compute common friends for each pair of users
val commonFriends = graph.triplets
  .flatMap { triplet =>
    Seq(
      ((triplet.srcId, triplet.dstId), 1),  // Direct relation (User1, User2)
      ((triplet.dstId, triplet.srcId), 1)   // Reverse relation (User2, User1)
    )
  }
  .reduceByKey(_ + _) // Sum up occurrences of pairs
  .filter { case (_, count) => count > 1 } // Keep only pairs with common friends

// Task 4: Collect and display results
println("Common Friends:")
```

```
45      commonFriends.collect().foreach { case ((user1, user2), count) =>
46        println(s"Users $user1 and $user2 have $count common friends.")
47      }
```

The main objective of this code is to compute and display pairs of users who share common friends. Given a dataset in the form of a CSV file that contains user relationships, the code performs the following tasks:

1. The relationships (edges) are loaded from the CSV file, where each line represents a friendship between two users.
2. Construct a graph using these relationships, where the vertices are users, and the edges represent the friendships.
3. Computes the number of common friends for each pair of users, which is determined by looking at the shared neighbors (i.e., mutual friends) in the graph.
4. Filters the results to show only those pairs of users who have more than one common friend and prints the number of common friends for each such pair.

6 Compare Apache Spark with Other Tools

This section covers the overview of other existing tools (Apache Hadoop, Apache Storm, Apache Flink) and compares them with Apache Spark on multiple criteria.

6.1 Apache Hadoop

6.1.1 Overview

Apache Hadoop (Figure 40) is a widely used framework designed for distributed storage and processing of large datasets across clusters of computers. It has four primary components that work together to ensure efficient data management and computation.

1. Hadoop Distributed File System (HDFS)

HDFS is the backbone of Hadoop's data storage. It splits large datasets into smaller blocks and distributes them across multiple nodes in a cluster. By replicating these blocks across several nodes, HDFS ensures fault tolerance and high availability, even if some nodes fail. This architecture makes Hadoop well-suited for handling petabytes of data efficiently.

2. MapReduce

MapReduce is a programming model and processing engine in Hadoop for analyzing large-scale datasets. It operates in two distinct phases:

- **Map Phase:** This phase processes input data, converting it into key-value pairs for easier organization and analysis.
- **Reduce Phase:** The output from the Map phase is grouped and consolidated to generate meaningful results, such as counts, averages, or summaries.

MapReduce's ability to divide tasks into independent units allows it to leverage the computational power of a distributed cluster.

3. YARN (Yet Another Resource Negotiator)

YARN is Hadoop's resource management layer. It dynamically allocates computational resources like CPU and memory to various applications running on the Hadoop cluster. YARN also handles task scheduling and monitors application performance, ensuring optimal utilization of the cluster's resources.

4. Hadoop Common

This module contains essential libraries, utilities, and tools shared by other Hadoop components. It includes file system abstractions, configuration management, and common I/O utilities, providing a foundation for HDFS, MapReduce, and YARN to work seamlessly.

Figure 40: Apache Hadoop

By integrating these components, Apache Hadoop enables organizations to store, process, and analyze massive amounts of structured and unstructured data, making it a cornerstone of big data ecosystems. Its scalability and fault tolerance make it an excellent choice for applications in areas like data warehousing, machine learning, and analytics.

6.1.2 Compare Apache Spark and Apache Hadoop

Criteria	Apache Hadoop	Apache Spark
Architecture	Based on HDFS and MapReduce	Based on RDD (Resilient Distributed Datasets)
Processing Model	Batch processing via MapReduce	Supports both batch and streaming, real-time processing
Processing Speed	Slower due to frequent disk I/O	Faster due to in-memory processing
Ease of Programming	More complex, requires deep understanding of MapReduce	Easier with high-level APIs for Scala, Java, Python, and R
Fault Tolerance	Automatic through data replication in HDFS	Supported through RDDs with automatic recovery
Interactive Data Processing	Not optimized for interactive queries	Well-suited for interactive queries and data analysis
Ecosystem	Large ecosystem with tools like Hive, Pig, HBase	Also has a rich ecosystem with Spark SQL, MLlib, GraphX
Resource Management	Uses YARN for resource management	Integrates with YARN or can operate standalone
Scalability	Good horizontal scalability	Also supports horizontal scalability and large data processing
Data Type Support	Primarily structured data	Supports structured, semi-structured, and unstructured data

6.2 Apache Storm

6.2.1 Overview

Apache Storm (Figure 41) is a powerful distributed real-time computation system designed to process data streams as they arrive. It is highly scalable, fault-tolerant, and efficient, making it suitable for a wide variety of applications, including real-time analytics, data transformation, and event-driven processing. Below are the detailed features and insights into Apache Storm:

Figure 41: Apache Storm

Real-Time Processing

- Apache Storm specializes in processing unbounded streams of data in real-time.
- It enables applications to react instantly to incoming data, making it ideal for time-sensitive use cases such as real-time monitoring and analytics.

Distributed and Scalable Architecture

- Storm operates in a distributed environment, processing data across multiple nodes in a cluster.
- It supports horizontal scaling, allowing users to add more nodes to handle increasing workloads.

Fault Tolerance and Reliability

- Fault tolerance is a core feature of Storm. When a node or task fails, Storm reassigns the task to healthy nodes, ensuring uninterrupted processing.
- The system provides at-least-once processing guarantees, with an option for exactly-once guarantees in specific scenarios.

Event-Driven Model

- Storm uses an event-driven architecture where computations are triggered by incoming data events.
- This approach eliminates the need for scheduled batch processing, enabling more dynamic and responsive data pipelines.

Component-Based Architecture

Apache Storm's architecture is built around two key components:

- **Spouts**: These are data sources that feed streams of data into the topology. Spouts can pull data from external systems like Kafka or APIs.
- **Bolts**: These are the processing units that perform tasks such as filtering, aggregating, and transforming data. Bolts can also emit results to other bolts or external systems.

Storm topologies, represented as directed acyclic graphs (DAGs), define the flow of data between spouts and bolts.

Integration with Other Systems

- Apache Storm integrates seamlessly with various technologies such as Apache Kafka, HDFS, and relational databases.
- It can work alongside machine learning frameworks and stream processing tools to build robust data pipelines.

High Throughput and Low Latency

- Storm delivers high throughput, capable of processing millions of tuples per second per node.
- Its low-latency design makes it suitable for applications requiring near-instantaneous processing.

6.2.2 Comparison between Apache Storm and Apache Spark

Criteria	Apache Storm	Apache Spark
Processing Model	Real-time stream processing	Batch processing and real-time (Structured Streaming)
Fault Tolerance	Automatic task reassignment in case of failure	Automatic recovery via RDD lineage
Ease of Use	Requires more effort for defining topologies	Easier with high-level APIs (DataFrames, Datasets)
Latency	Low-latency, ideal for real-time analytics	Higher latency, though Structured Streaming is also low-latency
Stateful Processing	Supported with frameworks like Trident	Built-in support for stateful streaming
Programming Language Support	Java, Clojure, Python	Java, Scala, Python, R
Scalability	Highly scalable and can handle high-throughput data streams	Highly scalable for both batch and real-time workloads
Streaming Support	Native, real-time stream processing	Batch processing by default, real-time streaming via add-ons

6.3 Apache Flink

6.3.1 Overview

Apache Flink (Figure 42) is an open-source stream processing framework designed for distributed, high-performance, and low-latency data processing. It supports both real-time (stream) and batch processing workloads, making it versatile for a wide range of big data applications. Below are its features and highlights:

Key Features of Apache Flink

- **True Stream Processing**: Processes data as it arrives, providing genuine real-time capabilities without requiring batch intervals.
- **Event Time and Windowing**: Provides sophisticated handling of event time (the time when an event was generated) and advanced windowing techniques, enabling analyses of streams where events may arrive late or out of order.
- **Fault Tolerance**: Maintains system reliability by periodically creating distributed snapshots of its state, ensuring recovery in case of failure.
- **Stateful Stream Processing**: Allows Flink to maintain and manage the state of each computation across events, facilitating complex stream operations.
- **Unified API for Batch and Streaming**: Simplifies development by providing a single, consistent API for handling both streaming and batch data.
- **Integration with Big Data Ecosystem**: Connects seamlessly with data storage and messaging systems like Kafka, HDFS, Cassandra, and Elasticsearch, making it a versatile choice for big data architectures.

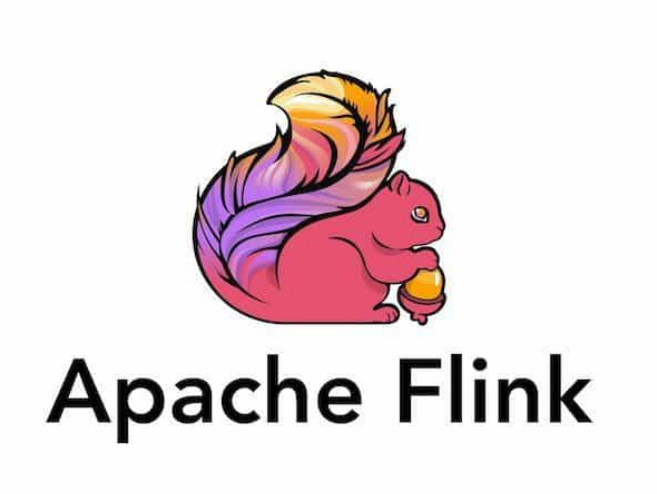


Figure 42: Apache Flink

6.3.2 Comparison between Apache Flink and Apache Spark

Criteria	Apache Flink	Apache Spark
Processing Model	True real-time, event-driven processing	Batch processing by default; streaming with micro-batching
Fault Tolerance	Distributed snapshots for stateful recovery	Recovery via lineage in RDDs (micro-batch)
Latency	Low latency, better suited for high-throughput real-time	Low latency in streaming, but higher than Flink in some cases
Windowing and Event Time	Advanced, supports late-arriving events	Supports event-time processing but with more limited features
Stateful Processing	Built-in, efficient state management for complex apps	Supports stateful streaming, less optimized than Flink
Batch Processing	Supported, but optimized for streaming	Optimized for batch; also supports streaming
Ease of Use	Unified API, but more complex for new users	Easier with high-level APIs for DataFrames and Datasets
Programming Language Support	Java, Scala, Python	Java, Scala, Python, R
Streaming Support	Native, continuous processing	Supported with Structured Streaming (micro-batch)

References

[1] 012-Spark RDDs - https://www.youtube.com/watch?v=nH6C9vqtyYU&t=133s

[2] Apache Spark with Scala - Hands On with Big Data! - https://www.youtube.com/watch?v=SLZS1mSc_VY

[3] Apache Spark 3.5 Tutorial with Examples - https://sparkbyexamples.com/

[4] What is Spark Job - https://sparkbyexamples.com/spark/what-is-spark-job/

[5] What is the Spark Stage? Explained - https://sparkbyexamples.com/spark/what-is-spark-stage/

[6] What is Spark Executor - https://sparkbyexamples.com/spark/what-is-spark-executor/

[7] What is Apache Spark Driver? - https://sparkbyexamples.com/spark/what-is-apache-spark-driver/

[8] What is DAG in Spark or PySpark - https://sparkbyexamples.com/spark/what-is-dag-in-spark/

[9] What is a Lineage Graph in Spark? - https://sparkbyexamples.com/spark/what-is-lineage-graph-in-spark/

[10] How to Submit a Spark Job via Rest API? - https://sparkbyexamples.com/spark/submit-spark-job-via-rest-api/

[11] Spark RDD vs DataFrame vs Dataset - https://sparkbyexamples.com/spark/spark-rdd-vs-dataframe-vs-dataset/?fbclid=IwY2xjawHkKQVleHRuA2FlbQIxMAABHaeEtOqiI3u4Mp5IPonuzKjs4IaBtOwS3V99TUYO45yammYBM5b6Xgg-DQ_aem_o46v64TfZHK8Mb24TEAqBQ

[12] RDDs vs. Dataframes vs. Datasets: What is the Difference and Why Should Data Engineers Care? - https://www.analyticsvidhya.com/blog/2020/11/what-is-the-difference-between-rdds-dataframes-and-datasets/

[13] Data Engineering for Beginners – Get Acquainted with the Spark Architecture - https://www.analyticsvidhya.com/blog/2020/11/data-engineering-for-beginners-get-acquainted-with-the-spark-architecture/

[14] Apache Spark 3.5 Tutorial with Examples - https://sparkbyexamples.com/

[15] Kafka Streams Architecture - https://kafka.apache.org/39/documentation/streams/architecture

[16] Narkhede N., Shapira G. & Palino T. (2017) Kafka: The Definitive Guide, OReilly Media, USA.

[17] KTable (kafka 2.3.0 API) - https://kafka.apache.org/23/javadoc/org/apache/kafka/streams/kstream/KTable.html

[18] KStream (kafka 2.3.0 API) - https://kafka.apache.org/23/javadoc/org/apache/kafka/streams/kstream/KStream.html

[19] GlobalKTable (kafka 2.3.0 API) - https://kafka.apache.org/23/javadoc/org/apache/kafka/streams/kstream/GlobalKTable.html