Why use Apache Storm?

Apache Storm is a free and open source distributed real time computation system. Apache Storm makes it easy to reliably process unbounded streams of data, doing for real time processing what Hadoop did for batch processing. Apache Storm is simple, can be used with any programming language, and is a lot of fun to use!

Apache Storm has many use cases: real time analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Apache Storm is fast: a benchmark clocked it at over **a million tuples processed per second per node**. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

Apache Storm integrates with the queueing and database technologies you already use. An Apache Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed. Read more in the tutorial.

What is Apache Storm?

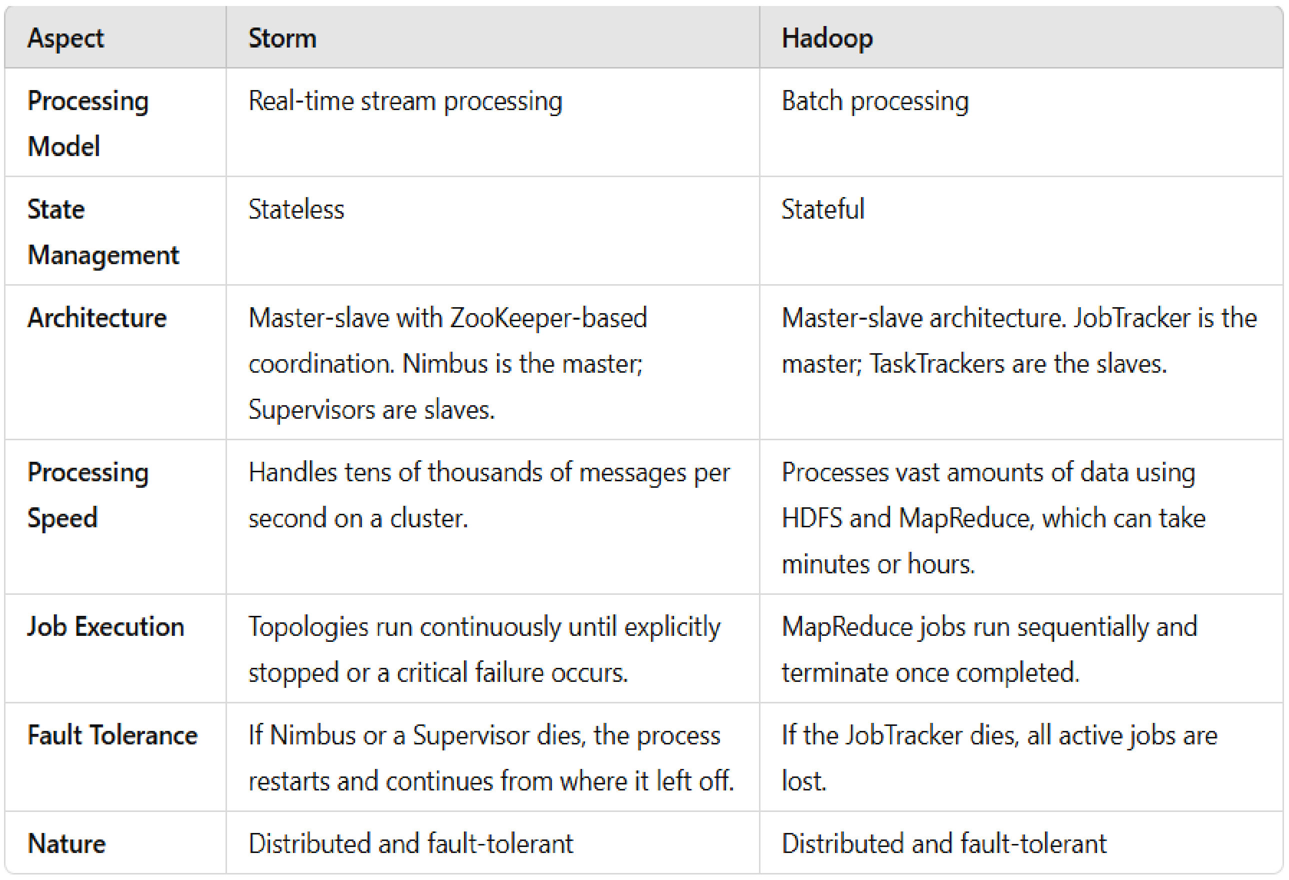
Apache Storm is a distributed real-time big data-processing system. Storm is designed to process vast amount of data in a fault-tolerant and horizontal scalable method. It is a streaming data framework that has the capability of highest ingestion rates. Though Storm is stateless, it manages distributed environment and cluster state via Apache ZooKeeper. It is simple and you can execute all kinds of manipulations on real-time data in parallel.

Apache Storm is continuing to be a leader in real-time data analytics. Storm is easy to setup, operate and it guarantees that every message will be processed through the topology at least once.

Apache Storm vs Hadoop

Basically, Hadoop and Storm frameworks are used for analysing big data. Both of them complement each other and differ in some aspects. Apache Storm does all the operations except persistency, while Hadoop is good at everything but lags in real-time computation. The following table compares the attributes of Storm and Hadoop.

Structured comparison between Apache Storm and Hadoop



Aspect	Storm	Hadoop
Processing Model	Real-time stream processing	Batch processing
State Management	Stateless	Stateful
Architecture	Master-slave with ZooKeeper-based coordination. Nimbus is the master; Supervisors are slaves.	Master-slave architecture. JobTracker is the master; TaskTrackers are the slaves.
Processing Speed	Handles tens of thousands of messages per second on a cluster.	Processes vast amounts of data using HDFS and MapReduce, which can take minutes or hours.
Job Execution	Topologies run continuously until explicitly stopped or a critical failure occurs.	MapReduce jobs run sequentially and terminate once completed.
Fault Tolerance	If Nimbus or a Supervisor dies, the process restarts and continues from where it left off.	If the JobTracker dies, all active jobs are lost.
Nature	Distributed and fault-tolerant	Distributed and fault-tolerant

Use Cases of Apache Storm

1. **Twitter**
   - Used for **Publisher Analytics Products** to process every tweet and click in real time.
   - Integrated deeply with Twitter's infrastructure.
2. **NaviSite**
   - Implements an **event log monitoring/auditing system**.
   - Logs are processed through Storm, matching messages against configured regular expressions. Matches are saved to a database.
3. **Wego**
   - A **travel metasearch engine** that processes real-time travel data from global sources.
   - Resolves concurrency issues and ensures the best data matches for end-users.

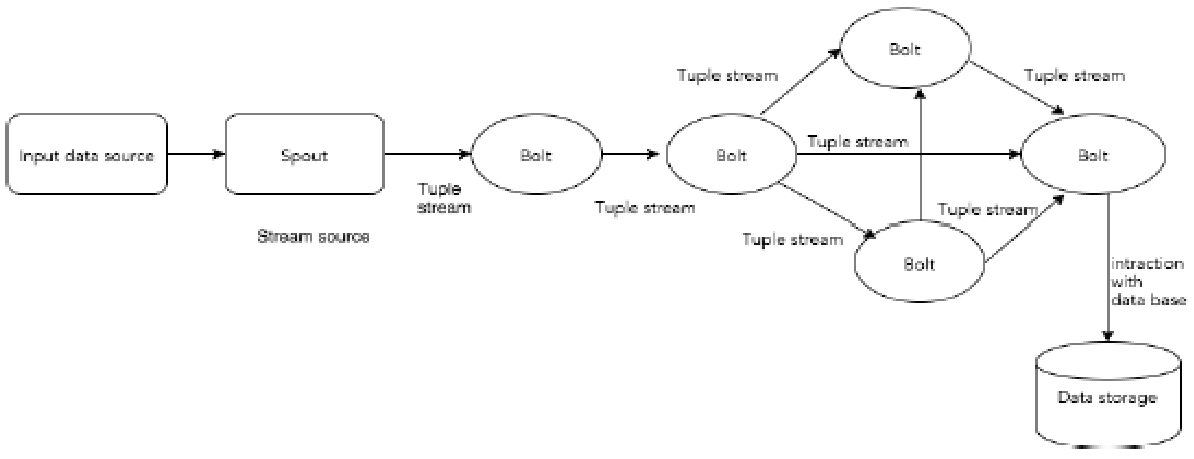
Benefits of Apache Storm

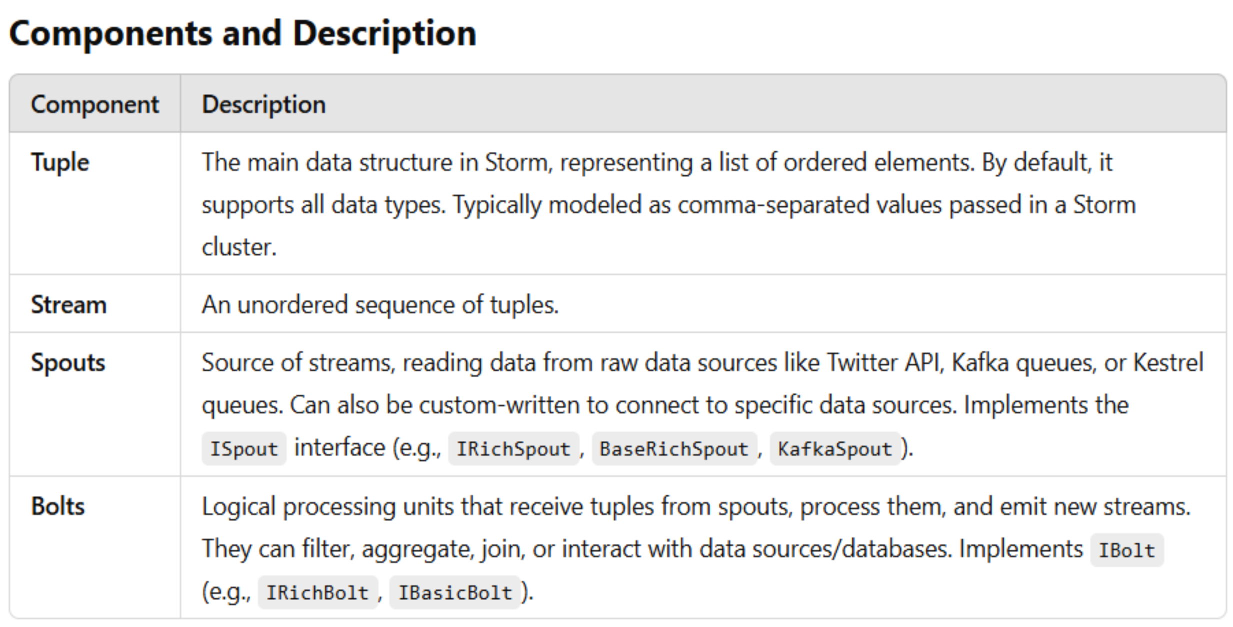
1. **Open Source and Versatile**
   - Robust, user-friendly, and suitable for both small and large organizations.
2. **Fault Tolerant and Reliable**
   - Maintains stability and guarantees data processing even with node failures.
3. **Real-Time Stream Processing**
   - Designed for low-latency data processing, completing tasks in seconds or minutes.
4. **High Speed and Scalability**
   - Processes enormous amounts of data quickly and scales linearly with increased resources.
5. **Flexibility**
   - Supports **multiple programming languages**, making it adaptable to various development environments.
6. **Low Latency**
   - Ensures rapid data refresh and delivery response.
7. **Operational Intelligence**
   - Provides insights into real-time data processing and decision-making capabilities.

Apache Storm - Core Concepts

Apache Storm reads raw streams of real-time data from one end, processes it through a sequence of small units, and outputs the useful information at the other end.

The core components of Apache Storm are explained below:





Components and Description

Component	Description
Tuple	The main data structure in Storm, representing a list of ordered elements. By default, it supports all data types. Typically modeled as comma-separated values passed in a Storm cluster.
Stream	An unordered sequence of tuples.
Spouts	Source of streams, reading data from raw data sources like Twitter API, Kafka queues, or Kestrel queues. Can also be custom-written to connect to specific data sources. Implements the `ISpout` interface (e.g., `IRichSpout`, `BaseRichSpout`, `KafkaSpout`).
Bolts	Logical processing units that receive tuples from spouts, process them, and emit new streams. They can filter, aggregate, join, or interact with data sources/databases. Implements `IBolt` (e.g., `IRichBolt`, `IBasicBolt`).

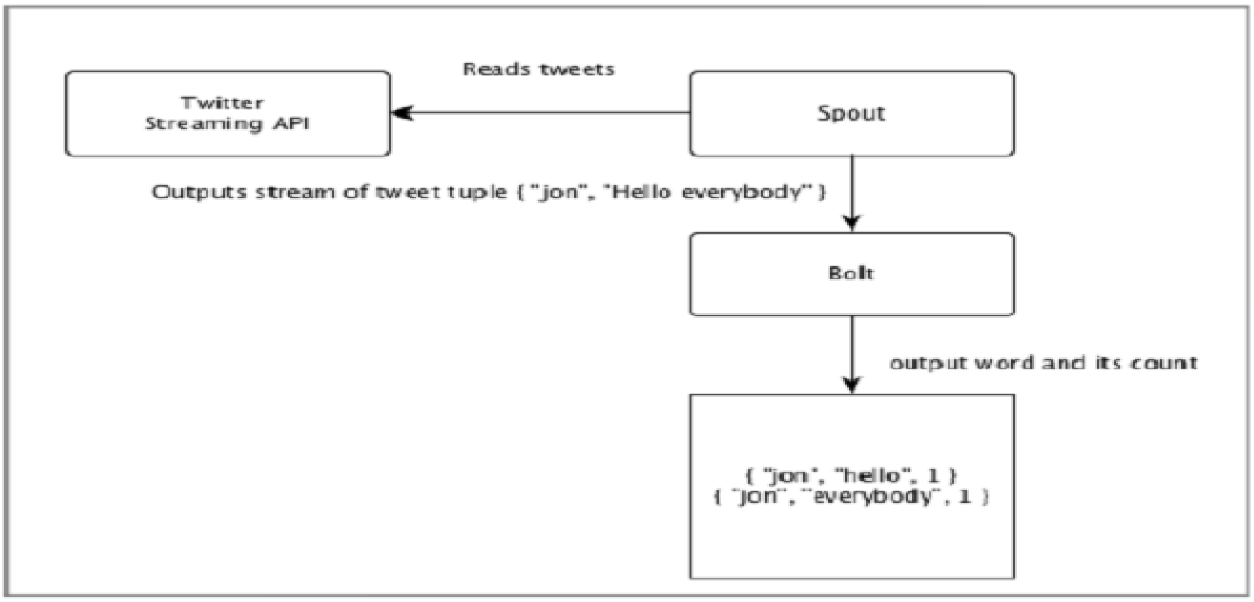
Example: Twitter Analysis

Input: Tweets are fetched from the Twitter Streaming API by a spout, which emits them as tuples, e.g., ["username", "tweet"].

Processing:

- Bolt 1 splits the tweet into words.
- Bolt 2 calculates the word count.
- Data is persisted to a configured database.

Output: Results can be queried from the database.


Twitter Streaming API
Reads tweets
Spout
Outputs stream of tweet tuple { "jon", "Hello everybody" }
Bolt
output word and its count
{ "jon", "hello", 1 }
{ "jon", "everybody", 1 }

Topology

Spouts and bolts are connected to form a **topology**, which represents real-time application logic.

Topology:

A **directed graph**, where:

- **Vertices**: Computation units (spouts and bolts).
- **Edges**: Stream of data.

Starts with spouts that emit data to bolts.

Storm keeps the topology running continuously until manually terminated.

Tasks

- Execution of spouts and bolts is referred to as a **task**.
- Tasks run in parallel, with multiple instances of spouts and bolts executing in separate threads.

Workers

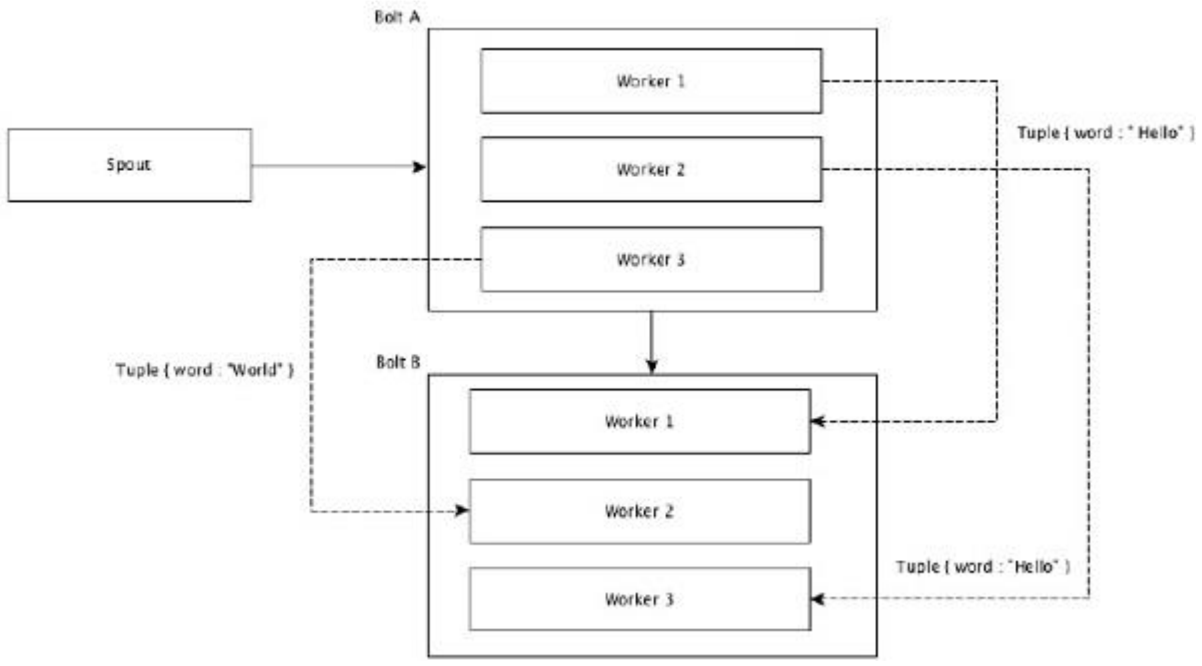
- Topologies run on multiple worker nodes in a distributed manner.
- Worker nodes listen for jobs and manage their execution.

Stream Grouping

Stream of data flows from spouts to bolts or from one bolt to another bolt. Stream grouping controls how the tuples are routed in the topology and helps us to understand the tuples flow in the topology. There are four in-built groupings as explained below.

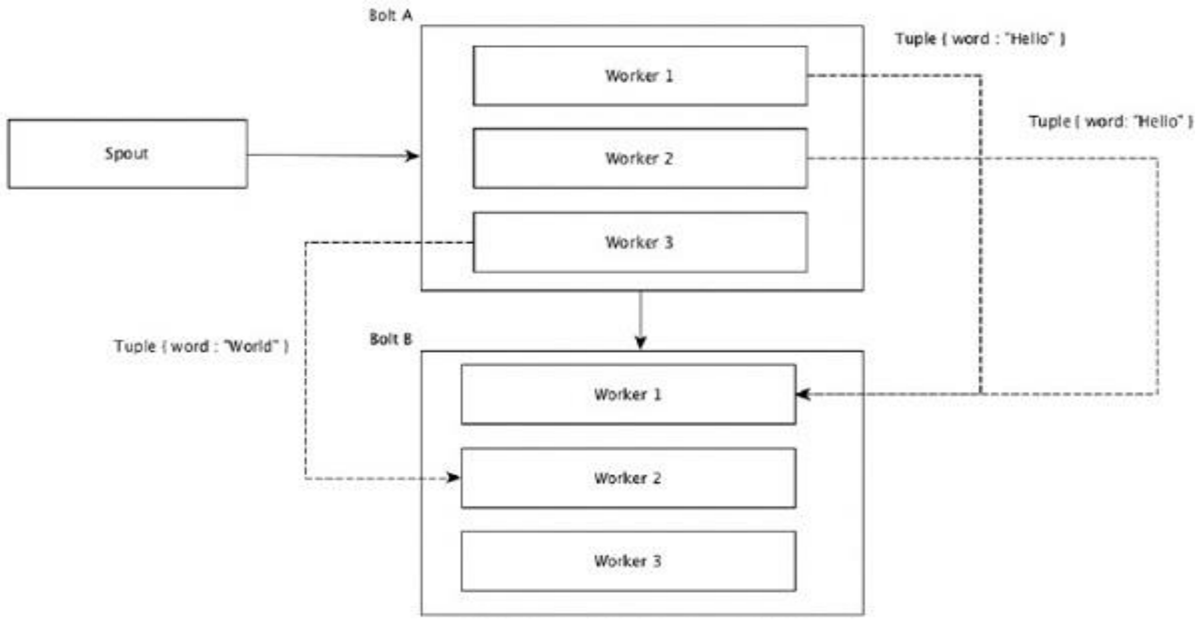
Shuffle Grouping

In shuffle grouping, an equal number of tuples is distributed randomly across all of the workers executing the bolts. The following diagram depicts the structure.



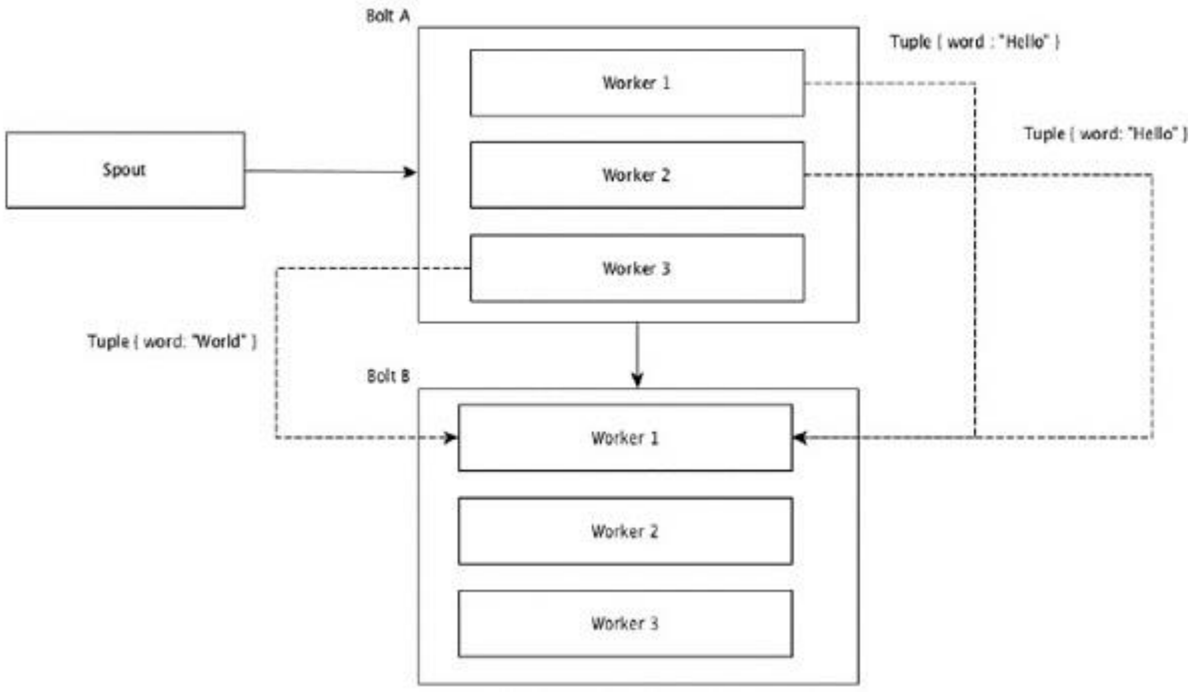
Field Grouping

The fields with same values in tuples are grouped together and the remaining tuples kept outside. Then, the tuples with the same field values are sent forward to the same worker executing the bolts. For example, if the stream is grouped by the field "word", then the tuples with the same string, "Hello" will move to the same worker. The following diagram shows how Field Grouping works.



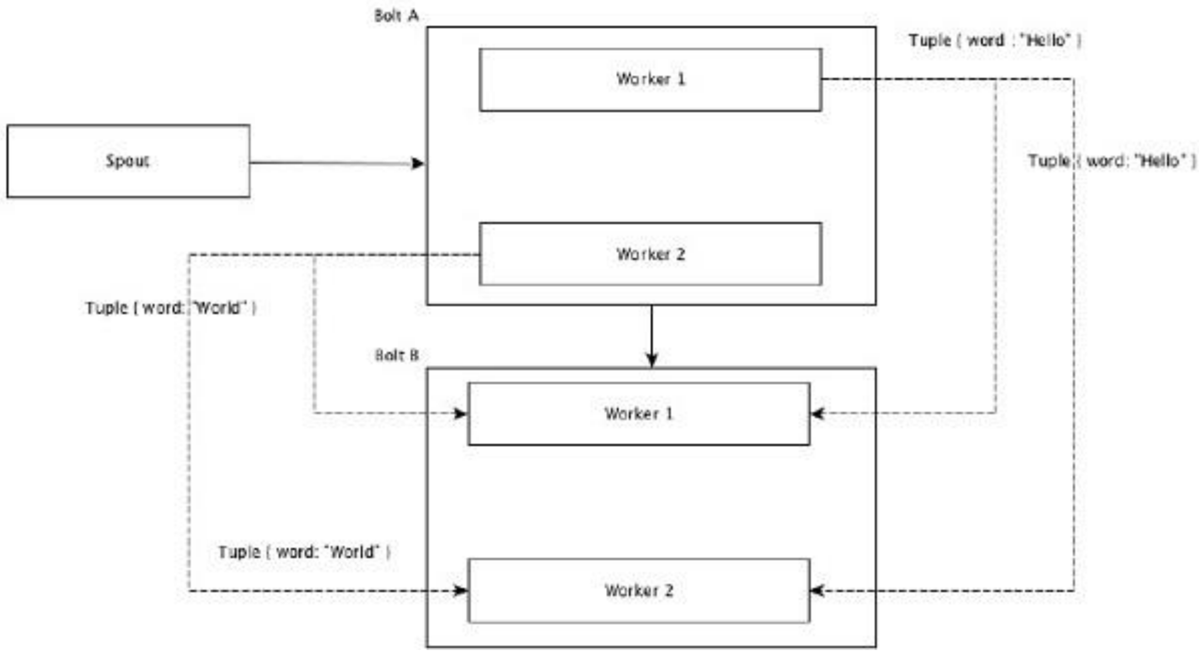
Global Grouping

All the streams can be grouped and forward to one bolt. This grouping sends tuples generated by all instances of the source to a single target instance (specifically, pick the worker with lowest ID).



All Grouping

All Grouping sends a single copy of each tuple to all instances of the receiving bolt. This kind of grouping is used to send signals to bolts. All grouping is useful for join operations.



Apache Storm - Cluster Architecture

Apache Storm - Cluster Architecture
Apache Storm is a fault-tolerant, distributed, and highly scalable real-time data processing framework. It is designed with no Single Point of Failure (SPOF), enabling efficient data processing by scaling across multiple systems.

Highlights of Apache Storm

- Fault-tolerant: Ensures the system continues functioning even in the case of component failures.
- Scalable: Can add more systems to increase capacity.
- Distributed architecture: Tasks are divided across multiple nodes.
- Stateless processing: Processes real-time data efficiently.

Cluster Design

The Apache Storm cluster consists of the following key components:

1. Nimbus (Master Node)

- The central component of the Storm cluster.
- Responsibilities:
    - Runs the Storm topology.
    - Analyzes the topology and breaks it into tasks.
    - Distributes tasks to available Supervisors.
    - Monitors failures and reassigns tasks if needed.
- Stateless: Relies on Apache ZooKeeper for state management.

2. Supervisor (Worker Node)

- Follows instructions provided by Nimbus.
- Responsibilities:
    - Hosts one or more worker processes.
    - Delegates tasks to worker processes for execution.

3. Worker Process

- Executes tasks related to a specific Storm topology.
- Does not perform tasks directly; instead, it spawns executors to handle specific tasks.

4. Executor

- A thread spawned by a worker process.
- Executes one or more tasks for either a spout or a bolt.

5. Task

- The smallest unit of execution in Apache Storm.
- Performs actual data processing, implemented as a spout or a bolt.

6. ZooKeeper Framework

- A distributed coordination service used by the cluster for:
  - Maintaining state across nodes.
  - Synchronizing data between Nimbus and Supervisors.
  - Monitoring the health of nodes.
- ZooKeeper enables statelessness for Nimbus and Supervisors by storing their state, ensuring quick recovery in case of failure.

Stateless Design and Recovery

- Stateless Nature:
  - Improves real-time data processing by not retaining state locally.
  - Minimizes downtime.
- State Storage:
  - Critical state information is stored in ZooKeeper.
  - Allows a failed Nimbus or Supervisor to restart and resume from the last known state.
- Tools like Monit monitor and restart Nimbus in case of failure.

Trident Topology

- An advanced topology in Apache Storm.
- Features:
  - State maintenance: Provides reliable processing with state preservation.
  - High-level API: Similar to Pig, making it easier to process complex workflows.

Apache Storm - Workflow

Apache Storm provides a robust architecture to handle distributed real-time data processing. Below is a detailed explanation of the Storm workflow:

Storm Cluster Overview

A functioning Storm cluster consists of:

1. **Nimbus (Master Node)**:
   - Central component for processing and distributing tasks.
2. **Supervisors (Worker Nodes)**:
   - Execute tasks assigned by Nimbus.
3. **Apache Zookeeper**:
   - Ensures coordination between Nimbus and Supervisors.

Step-by-Step Workflow

1. **Topology Submission**:
   - Nimbus waits for a "Storm Topology" to be submitted by the user.
   - Once submitted, Nimbus:
     - Processes the topology.
     - Gathers the tasks to be executed.
     - Determines the execution order.
2. **Task Distribution**:
   - Nimbus distributes tasks evenly among all available Supervisors.
3. **Heartbeat Mechanism**:
   - Supervisors periodically send **heartbeats** to Nimbus to indicate they are active and operational.
   - If Nimbus doesn’t receive a heartbeat from a supervisor:
     - The tasks assigned to that Supervisor are reallocated to another available Supervisor.
4. **Handling Failures**:
   - **Supervisor Failure**:
     - If a supervisor dies, Nimbus reassigns its tasks to another Supervisor.
     - Restarted Supervisors resume operations as before.
   - **Nimbus Failure**:
     - Supervisors continue executing already assigned tasks without interruption.
     - Nimbus is automatically restarted by **service monitoring tools** like Monit.
     - The restarted Nimbus resumes from where it left off.

5. **Task Completion**:
   - Supervisors wait for new tasks after completing current ones.
   - Nimbus waits for new topologies to be submitted.
6. **Guaranteed Processing**:
   - Apache Storm ensures all tasks are processed **at least once**, even in the event of failures.

Storm Cluster Modes

1. **Local Mode**:
   - Used for **development, testing, and debugging**.
   - Topologies run locally in a **single JVM**.
   - Useful for experimenting with various configurations.
2. **Production Mode**:
   - Used for **real-time deployments**.
   - Topologies run on a distributed Storm cluster composed of multiple machines.
   - Clusters run **indefinitely** until manually shut down.

Apache Storm - Distributed Messaging System

Apache Storm processes real-time data streams, and its architecture relies heavily on distributed messaging systems for communication, both externally and internally.

Role of Messaging Systems in Storm

1. **Input Data Source**:
   - Apache Storm consumes real-time input from external distributed messaging systems such as **Apache Kafka**, **RabbitMQ**, or others.
   - A **Spout** in Storm reads this input, converts it into tuples, and feeds them into the Storm topology.
2. **Internal Messaging**:
   - Apache Storm employs its own **distributed messaging system** for communication between **Nimbus** and **Supervisors**.

What is a Distributed Messaging System?

A **Distributed Messaging System** queues messages asynchronously between clients and servers, enabling reliable and scalable message delivery. It is designed to support features such as:

- **Reliability**: Ensures no data loss.
- **Scalability**: Easily handles increased load.
- **Persistence**: Messages are stored for guaranteed delivery.

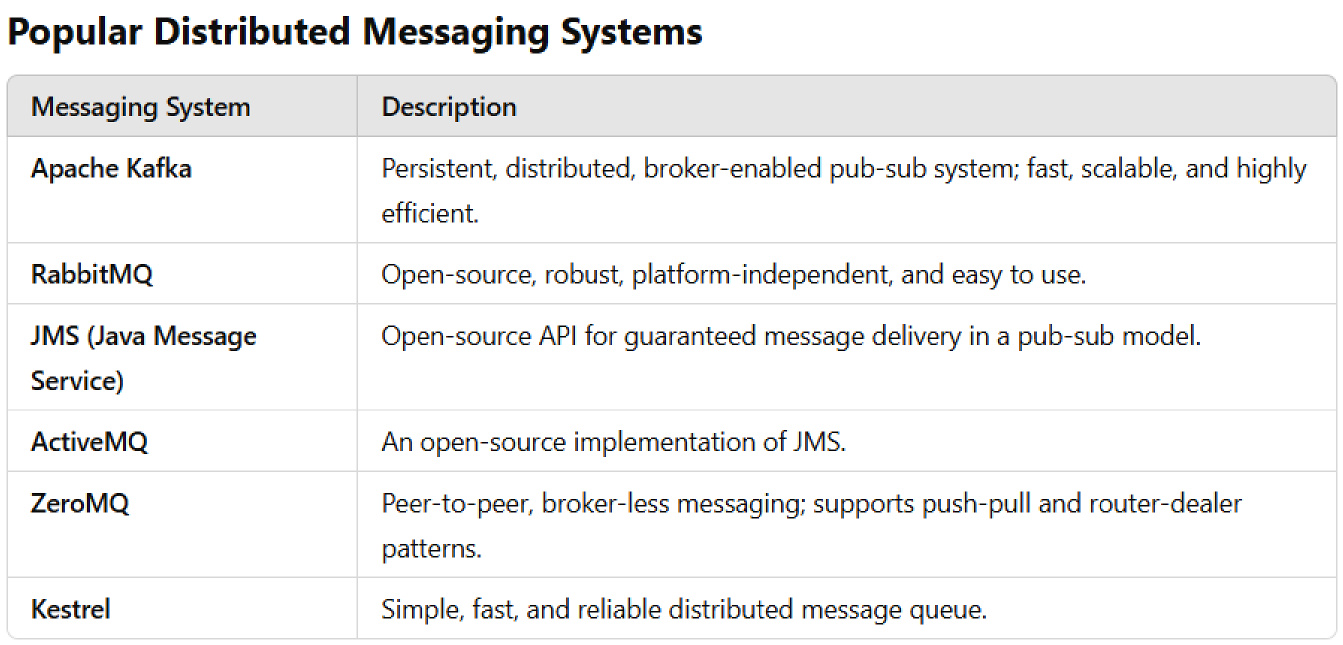
Publish-Subscribe (Pub-Sub) Model

The **Pub-Sub Model** is widely used in distributed messaging:

- **Publishers**: Send messages (e.g., channels like sports, movies).
- **Subscribers**: Receive messages based on preferences (e.g., choosing specific channels to view).
- **Filtering**:
    - **Topic-based Filtering**: Messages are categorized into topics.
    - **Content-based Filtering**: Filters based on specific content.

This architecture is loosely coupled; publishers do not need to know their subscribers. For example:

- **Dish TV** publishes channels, and subscribers choose their desired ones.



Popular Distributed Messaging Systems

Messaging System	Description
Apache Kafka	Persistent, distributed, broker-enabled pub-sub system; fast, scalable, and highly efficient.
RabbitMQ	Open-source, robust, platform-independent, and easy to use.
JMS (Java Message Service)	Open-source API for guaranteed message delivery in a pub-sub model.
ActiveMQ	An open-source implementation of JMS.
ZeroMQ	Peer-to-peer, broker-less messaging; supports push-pull and router-dealer patterns.
Kestrel	Simple, fast, and reliable distributed message queue.

Thrift Protocol

Apache Thrift, initially developed by Facebook, is:

- **A Communication Framework**: Supports cross-language communication for distributed applications.
- **Key Features**:
    - Modular, flexible, high-performance.
    - Enables streaming, messaging, and RPC (Remote Procedure Calls).

Thrift in Apache Storm:

- **Topology Definition**: Storm topology is structured using Thrift.
- **Communication**: Storm Nimbus (the master node) operates as a Thrift service.

Apache Storm - Working Example

Apache Storm - Trident

Apache Storm's **Trident** is a high-level abstraction for stream processing built on top of Storm, offering stateful stream processing, exactly-once processing semantics, and low-latency distributed querying. It is particularly useful for use cases where transactional consistency and stateful operations are required. Below is a concise breakdown of Trident's key features and concepts:

Key Features

1. **Batch Processing**: Trident processes data in batches (transactions) rather than tuple-by-tuple as in Storm. Each batch has a unique transaction ID, ensuring exactly-once processing.
2. **Abstraction**: It simplifies stream processing by introducing functions, filters, aggregations, and joins, abstracting away the low-level complexities of Storm.
3. **Stateful Stream Processing**: Trident provides tools for state maintenance, supporting memory or external storage for stateful operations.
4. **Fault Tolerance**: Failed transactions are retried entirely, maintaining the consistency of operations.

Core Components

1. **TridentTopology**: The starting point for creating a Trident topology.
2. **Spouts**:
   - Use ITridentSpout for transactional or opaque transactional semantics.
   - Example: FeederBatchSpout simplifies feeding data into the topology.
3. **Operations**:
   - **Filter**: Filters tuples based on a condition.
   - **Function**: Processes tuples to emit zero or more fields.
   - **Aggregation**: Supports batch-wide, partition-wise, or persistent aggregation.
   - **Grouping**: Groups tuples based on specified fields.
4. **Merging and Joining**:
   - **Merge**: Combines multiple streams.
   - **Join**: Joins two streams based on a key.

State Maintenance

- Trident ensures consistent state updates by ordering transactions and retrying failed batches with the same transaction ID.
- Example:

```
topology.newStream("spout", spout)
     .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"));
```

Distributed RPC

- Enables querying Trident topologies in real-time using an RPC server.
- Example usage of **LocalDRPC** for testing:

```
LocalDRPC drpc = new LocalDRPC();
topology.newDRPCStream("query", drpc)
        .stateQuery(callCounts, new Fields("args"), new MapGet(), new Fields("result"));
```

Example: Call Log Analyzer

The application demonstrates a call log analyzer using Trident, implementing features like formatting call information, filtering, grouping, and distributed querying.

Components

1. **FormatCall**: Formats call information.
2. **CSVSplit**: Splits CSV input for queries.
3. **LogAnalyserTrident**:
    - Creates a Trident topology.
    - Feeds data using FeederBatchSpout.
    - Processes and aggregates call counts.
    - Sets up a **DRPC Server** for querying.

Execution Steps

1. Compile:
    ```
    javac -cp "/path/to/storm/apache-storm-0.9.5/lib/*" *.java
    ```
2. Run:
    ```
    java -cp "/path/to/storm/apache-storm-0.9.5/lib/*":. LogAnalyserTrident
    ```

Expected Output

```
DRPC : Query starts
[["1234123401 - 1234123402",10]]
DEBUG: [1234123401 - 1234123402, 10]
DEBUG: [1234123401 - 1234123403, 10]
[[20]]
DRPC : Query ends
```

When to Use Trident

- **Exactly-once Processing**: Applications requiring strong guarantees on data processing.
- **Stateful Processing**: Scenarios involving state maintenance (e.g., counting, aggregations).
- **Distributed RPC**: Real-time querying and analysis of stream data.

Apache Storm in Twitter

This example demonstrates how Apache Storm, in combination with the twitter4j library, can be used to create a real-time application for analyzing hashtags on Twitter. Here's a summary of the process:

Key Components

1. **Spout: TwitterSampleSpout**
   - Fetches real-time tweets using the Twitter Streaming API.
   - Emits tweets to bolts for processing.
2. **Bolt: HashtagReaderBolt**
   - Extracts hashtags from tweets using getHashtagEntities provided by the twitter4j library.
   - Emits each hashtag to the next bolt.
3. **Bolt: HashtagCounterBolt**
   - Counts occurrences of each hashtag.
   - Stores the count in memory using a Java HashMap.
   - Prints the results upon shutdown.
4. **Topology: TwitterHashtagStorm**
   - Combines the spout and bolts into a topology:
     - **Spout**: TwitterSampleSpout for fetching tweets.
     - **Bolt**: HashtagReaderBolt for extracting hashtags.
     - **Bolt**: HashtagCounterBolt for counting hashtags.

Steps to Build and Run the Application

1. Set Up Twitter API Credentials

- Sign up for a Twitter Developer Account.
- Generate the following credentials:
  - ConsumerKey
  - ConsumerSecret
  - AccessToken
  - AccessTokenSecret

2. Code Compilation

Use the following command to compile the Java files:

```
javac -cp "/path/to/storm/apache-storm-0.9.5/lib/*:/path/to/twitter4j/lib/*" *.java
```

3. Execute the Application

Run the compiled application with required arguments:

```
java -cp "/path/to/storm/apache-storm-0.9.5/lib/*:/path/to/twitter4j/lib/*:." TwitterHashtagStorm
<ConsumerKey> <ConsumerSecret> <AccessToken> <AccessTokenSecret> <Keyword1> <Keyword2> ...
<KeywordN>
```

4. View Output

- Hashtag counts will be displayed in the console.
- Example:

makefile

```
Result: food : 2
Result: android : 1
Result: SundayRoast : 1
```

Practical Use Cases

This setup can be extended to:

- Monitor trending topics in real time.
- Perform sentiment analysis on tweets.
- Identify popular hashtags for marketing or research purposes.