A conceptual Introduction to Apache Kafka

Introduction:

- Every application creates data, whether it is log messages, metrics, user activity, outgoing messages, or sth else.
- Every byte of data has a story to tell, sth of importance that will inform the next thing to be done.
  - In order to know what is, we need to get the data from where it is created to where it can be analyzed.
    - The faster we do this, the more agile and responsive our organizations can be.
    - The less effort we spend on moving data around, the more we can focus on the core business at hand.

  ⟹ This is why the pipeline is a critical component in the data-driven enterprise. How we move the data becomes nearly as important as the data itself.

Publish/Subscribe Messaging:

Definition: A messaging pattern where the sender (publisher) of a piece of data (message) does not send it directly to a specific receiver.

Key Components:

Publisher:

* The sender of the message
* Classifies messages into categories without targeting specific recipients

Subscriber:

* Receives messages by subscribing to certain categories or classes of messages.

Broker:

* A central point in some pub/sub systems where messages are published to facilitate the distribution to subscribers.

⟹ Pub/Sub messaging is a critical component of data-driven applications, supporting scalable and loosely-coupled communication between systems.

- refers to a system design where the components or services interact with each other with minimal dependencies
  - In the context of pub/sub messaging, loosely-coupled communication means that publishers (data senders) and subscribers (data receivers) do not need direct, one-to-one connections. They communicate through a broker that manages the distribution, allowing publishers and subscribers to work independently.
    - This decoupling enables flexible and scalable communication in complex applications

Use cases & Evolution:

Initial use case:

- Many applications begin with:
  - * simple message queue: a messaging system where messages are sent directly from one producer (sender) to a specific consumer (receiver)
    - ↳ This setup ensures point-to-point communication, with each message delivered to only one recipient.
  - * interprocess communication channel (IPC): is a mechanism that allows different processes (programes or parts of a program) to exchange data and messages.

eg: an application might send monitoring data directly to a dashboard, creating a straightforward solution for a single metric display.

⇒ This is a simple solution to a simple problem that works when you are getting started with monitoring.

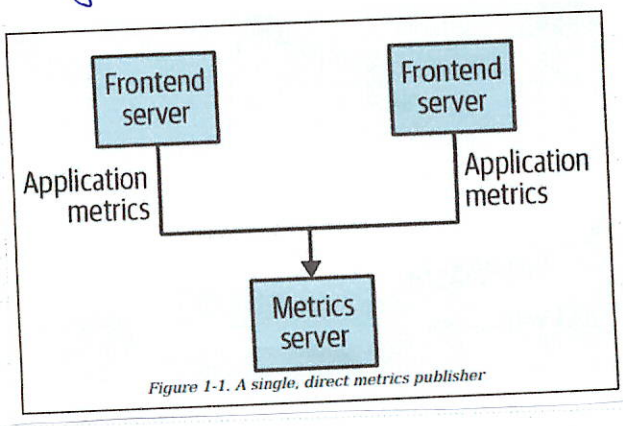


Figure 1-1. A single, direct metrics publisher

Scaling and complexity:

- As applications and systems grow, the simple point-to-point connections used for messaging become increasingly complex and harder to manage: Each new application or service added requires more connections
  - ↳ As applications scale, the complex point-to-point connections creates "technical debt" (a term referring to the growing maintenance burden and lack of scalability in the architecture)
    - ↳ Managing all these connections can become chaotic, leading to issues with reliability and increasing the risk of errors in data transmission.

eg: As the need for analysis grows, simple dashboards may no longer suffice, prompting the creation of new services to store and analyze metrics over time.

When additional applications need to generate and send metrics, each new app establishes its own direct connections to multiple services. This quickly becomes challenging to manage, as every new application or service requires multiple, hard-to-trace connections.

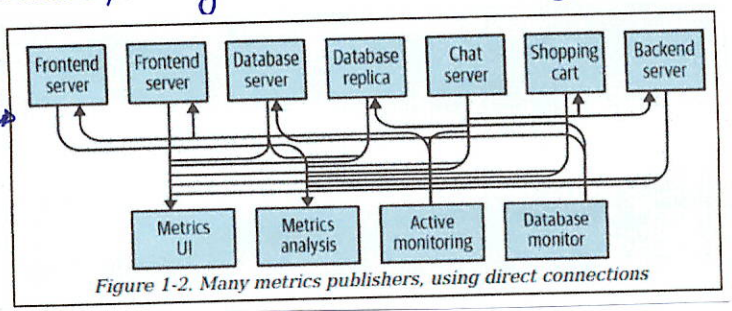


Figure 1-2. Many metrics publishers, using direct connections

Architecture Simplification:

- The scaling challenge highlights the need for a centralized messaging system that can efficiently handle multiple data streams without direct, individual connections for each service.

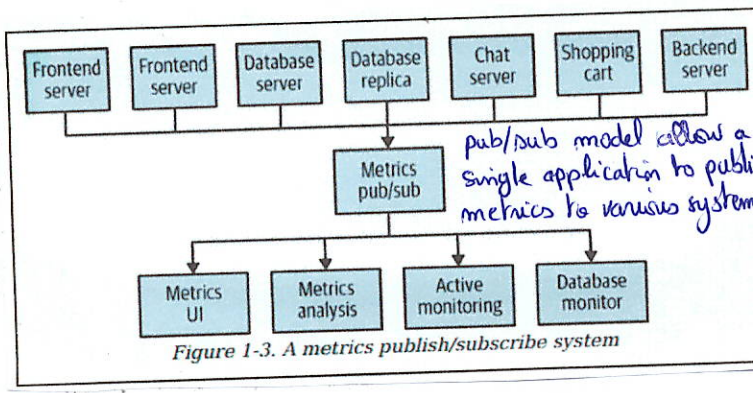


Figure 1-3. A metrics publish/subscribe system

- Adopting a publish/subscribe (pub/sub) model addresses this challenge by reducing connection complexity:
  * Instead of each service connecting directly to every other service, a single application publishes data once to a central broker, which then distributes to all interested services.
  * This approach consolidates data in one location, simplifying the architecture and making it more manageable and scalable.

Individual queue systems:

- Over time, an organization can end up with too many separate messaging systems, which can be inefficient and challenging to maintain:
  * In large organizations, different teams often set up their own pub/sub systems for specific needs
  * Each team manages their own pub/sub system independently. This means they build, configure, and maintain it in a way that best suits their specific goals.

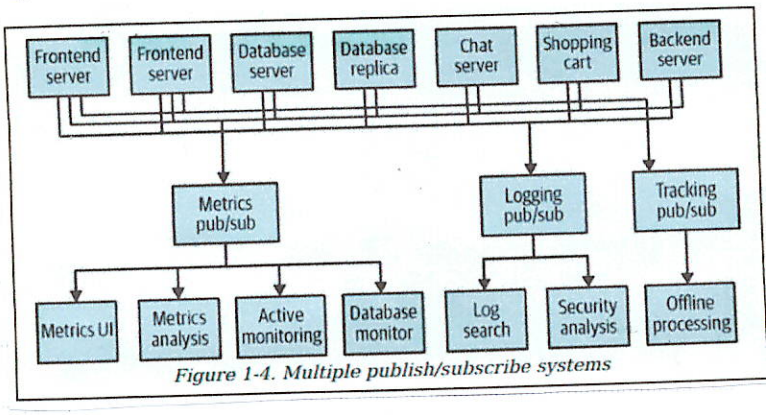


Figure 1-4. Multiple publish/subscribe systems

  * As each team creates its own system, the organization ends up with several isolated pub/sub setups. These setups aren't connected, which means they work in silos, often doing similar things, but without sharing resources.
    → operating independently without effectively sharing information or resources with each other.
  * Having many separate systems that do similar tasks can become inefficient. It leads to duplicate work, as each team is essentially managing their own version of the same type of infrastructure ⇒ This increases complexity and makes it harder to keep everything consistent across the organization.

⇒ To avoid redundancy and manage scalability, companies often move towards single, centralized pub/sub system for handling diverse data types.

⇒ Apache Kafka was developed as a publish/subscribe messaging system designed to solve this problem, by providing a unified pub/sub platform.

↳ Kafka's design helps companies replace numerous isolated systems with one, robust infrastructure, reducing complexity and operational overhead while supporting data growth and business expansion.

What is Kafka?

Logs and commit logs:

Logs:

Def: A log is a general term that refers to any record of events or actions that occur within a system serving various purposes from monitoring to debugging → They are a source of truth.

Purpose & Applications:

- Logs serve as a crucial tool in software systems, enabling debugging, monitoring, auditing, and diagnostics by providing a detailed record of activity.
- They are instrumental in:
  * Understanding failures: Logs help identify the flow of operations, errors, and performance issues, making them the go-to resource when diagnosing why something went wrong.
  * Tracing Operations: By recording events and actions across a system, logs allow developers to analyze interactions and dependencies within complex workflows.
  * Auditing and Security: Logs maintain to trail of operations, enabling audits to ensure compliance and detect potential security breaches.
- Logs are especially effective in scenarios where multiple autonomous components form a single system: For instance, in microservices architectures or monolithic systems with isolated processes.
  ↳ They provide visibility into interdependent operations, ensuring smooth functioning even in complex environment.

Commit logs:

Def:

- A commit log is a specialized type of log that records transact... and persisted, in a reliable immutable manner to ensure c... and facilitate recovery.
- They are a sequence of records, each with a unique identif...

> A commit log is a data structure that:
> * Maintains a durable record of all transactions or events
> * stores messages in order, allowing consistent state rebuilding and deterministic reads
> * Is essential in systems requiring reliability and fault tolerance.

Key characteristics:

- Immutability: Once a record is written to the commit log, it cannot be altered. This immutability ensu... that the history of transactions remains intact, which is crucial for recovery and auditing purposes.
- Sequential record keeping: a commit log maintains a sequence of records, where each record represents a transaction that has been applied to the system. This allows for tracking the state of the system over tim...
- Sequential appending: New records are always added to the end of the log.
- Deterministic reads: The sequence of records ensures ordered and consistent reading.
- Efficient writes: Writing to a commit log is fast, even on disk.

- **Recovery mechanism:** In the event of a system failure, the commit log can be used to reconstruct the state of the database by replaying the recorded transactions. This is essential for ensuring data durability and consistency.

Why commit logs are fundamental:

- **Durability:** whether it's data in databases or code changes, commit logs ensure that information is never lost.
- **Traceability:** Logs create an audit trail, which is crucial for debugging, accountability, and compliance.
- **Consistency:** They enable systems to recover from crashes and maintain consistent states across distributed environments.
- **Scalability:** Commit logs in analytics platforms (eg: Kafka) allow systems to handle vast amounts of data by replaying logs as needed for different consumers.

⟹ By enabling scalability, durability, and traceability, commit logs form the backbone of systems that require reliable data management and historical tracking, from databases to modern data pipelines and version control systems.

Applications:

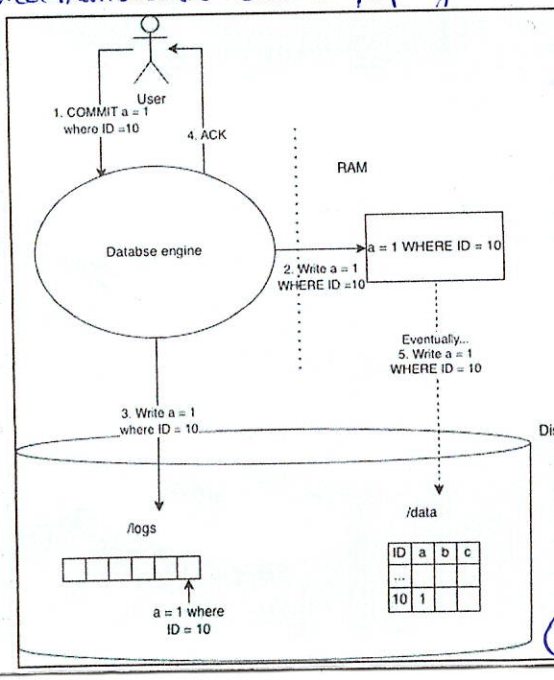
- Commit logs are a fundamental concept in software development due to their ability to track changes and ensure consistency, reliability, and durability in various systems.
- Here's how they are applied across different domains:

Relational Databases:

* **Purpose:** commit logs (or transaction logs) are used to ensure durability (the "D" in ACID properties)
* **How they work:**
  - Every change or transaction (eg: update, insert, or delete) is first recorded in the commit log.
  - The log ensures that even if the database crashes, all committed transactions can be replayed to restore the database to a consistent state.

eg: Analyzing, step by step, how a write operation happens within the database:

1. **User Operation:**
   - The user initiates a transaction with the command: (...)
     "COMMIT a = 1 WHERE ID = 10"
     ↳ This action signifies the user's intention to write a new value "a = 1" into the database.
2. **Database Engine Verification:**
   - The database engine checks the request, ensuring it satisfies



all constraints and rules (eg: data types, integrity constraints, ...)
▸ If the request passes, it proceeds with the next steps.

3. In memory Update: (step 2)
▸ The database engine temporarily saves the change "a=1 WHERE ID=10" in RAM.
↳ This ensures quick accessibility for subsequent read operations, avoiding the need for immediate disk access.

4. Durable persistence in Commit log: (step 3)
▸ The transaction is written to the commit log stored on disk.
▸ This log acts as the database's journal, designed to ensure durability.
↳ If a system crash occurs, the database can recover by replaying transactions from this log.
⚠ RAM alone doesn't suffice for durability because it is volatile and loses data in case of a crash.

5. Acknowledgment: (step 4)
▸ Once the transaction is safely written to the commit log, the database sends an ACK (acknowledgment) to the user
↳ The user can now trust that the transaction is durable and will persist even in the event of a crash.

6. Final Disk write: (step 5)
▸ At some point (based on system configuration or threshold like memory pressure), the operating system flushes the in-memory data to permanent storage on disk.
▸ At this stage, the new value "a=1" is stored in the appropriate table, row, and column structure on disk.
↳ This ensures that the database's permanent data storage is now up-to-date with the write operation.
▸ Once the data has been safely written to its final location on disk, the corresponding record (a=1) in the commit log is no longer needed and can be deleted.
↳ This frees up space in the log for new transactions while maintaining system efficiency.

* Benefits:
- Guarantees no data loss for committed transactions
- Facilitates recovery and fault tolerance.

Version Control Systems (eg: Git)

* Purpose: Commit logs provide a history of changes to the codebase.

* How they work:
- Each commit in Git records a snapshot of the repository at a specific point in time, along with metadata (eg: author, timestamp, and message)
- The commit log is a chronological list of these snapshots, enabling developers to trace the evolution of the codebase.

* Benefits:
  - Makes debugging easier by showing what changes were made and why.
  - Facilitates collaboration by allowing teams to review and merge changes effectively
  - Provides a mechanism to revert to a previous version if needed.

Analytics Platforms:

* Purpose: Commit logs serve as a stream of immutable events for systems like Apache Kafka and Apache Flink
* How they work: Commit logs act as a "master record" of all events, allowing the system to:
  - Store every event in sequence
  - Replay events whenever needed
  - Share events with multiple systems or consumers.

* Benefits:
  - Powers real-time and batch processing workflows.
    ↳ Commit logs are perfect for systems that need to process data reliably and in order.
  - They make sure data flows correctly, stay durable, and can be reused whenever needed.

Streaming Platform:

Def: A streaming platform is a system designed to process and manage continuous flows of data (called streams) in real time.

Core Functions:

* Data Collection: Receiving real-time data from multiple sources
* Data storage: Storing the data durably (for further processing) so they can be consumed at any time
* Real-time Processing: It processes continuous streams of data as they are generated, instead of waiting for batch processing.
* Data Distribution: Distributes data to multiple consumers simultaneously, ensuring scalability.

Kafka:

Apache Kafka was developed as a publish/subscribe messaging system designed to address inefficiencies in traditional systems. It is often described as a "distributed commit log" or more recently as a "distributed streaming platform".

Kafka as a publish/subscribe messaging system:

. Kafka solves key challenges in older messaging systems, such as:
  * Point-to-point connections: Difficult to scale when the number of systems grows.
  * Multiple messaging systems: Leading to duplicated work and inefficiencies
. Kafka acts as a central hub for managing data streams, addressing these limitations by offering:
  * Scalability: Handling generic data streams as businesses grow.
  * Centralized management: Consolidating data ingestion, storage, and distribution under one system.

Kafka as a distributed commit log:

- Kafka extends the concept of a traditional commit log by adding, distribution, scalability, and fault tolerance:
  - * Durable Data Storage: Stores events immutably and sequentially in partitions, ensuring consistent and reliable record keeping
  - * Order and consistency: Kafka ensures data is written in an immutable, ordered format.
  - * Event replay: Consumers can reprocess past events by reading specific log offsets.
  - * scalability: Logs are divided into partitions and distributed across multiple servers, enabling Kafka to handle high data volumes and throughput.
  - * Fault Tolerance: Data is replicated across servers, ensuring availability even in the event of node failures.
  - * Resilience: Components interact via the log, allowing the system to function independently even if some parts are offline

Kafka as a Distributed Streaming Platform:

- Apache Kafka is a distributed streaming platform designed for:
  - * Data Ingestion: Collecting and transporting large volumes of real-time or batch data from diverse sources (eg: logs, sensors, applications).
  - * Data Storage: Storing data durably and reliably with configurable retention policies.
  - * Data Distribution: Streaming data to multiple consumers in real time using a publish/subscribe model
- ⚠ While Kafka's primary purpose is data ingestion and distribution, it also supports lightweight data processing through its "Kafka Streams" API

Kafka's Architecture:

Messages and Batches:

Message:

- A message is the fundamental unit of data in kafka, comparable to a row or record in a database.
- A message in kafka is an array of bytes with no predefined format or meaning (Kafka does not impose a form
- A message can have an optional piece of metadata, which is referred to as a key:
  - * A key is a piece of metadata provided by the producer. (It is optional)
  - * The key is a byte array, similar to the message itself, and has no specific meaning to Kafka.
  - * When provided, keys are used for more controlled assignment of messages to partitions.
    - ▸ used to determine which partition the message is sent to.
    - ▸ A consistent hash of the key is generated (simplest way)
      - ↳ A hash of the key is calculated, and the partition is determined by: hash(key) % total_partitions
        - → This ensures that messages with the same key always go to the same partition.
        - → This consistency holds as long as the number of partitions in the topic does not change.

taking the hash result modulo the total number of partitions in the topic.

* The key does not inherently relate to the order of the message within the partition.

Note: The key is not the offset in Kafka: The key helps with partitioning decisions, while the offset identifies the position of a message within a partition.

- Throughput refers to the amount of data or number of messages a system can process in a given amount of time
- Higher throughput indicates the system can handle more messages efficiently over time

Batches:

Messages in batches:

- Messages are grouped into batches for efficiency
- Each batch consists of messages sent to the same topic and partition.

Purpose of batching:

- Sending each message individually across the network would cause too much overhead
  - Batching reduces the overhead caused by individual network trips for each message
    - number of network operations
- Grouping messages into batches allows more messages to be processed at once.
  - Batching
    - Improves throughput by handling more messages per unit of time
    - allows Kafka to handle a larger number of messages in less time.

Trade-off:

- Trade-off between latency (smaller batches) (speed) and throughput (larger batches) (efficiency):
  - * Larger batches improve throughput by reducing the frequency of network operations
  - * However, waiting to fill a batch may delay the propagation of individual messages, increasing latency.
  - → Larger batches increase throughput but may increase latency (delivery).

Batch Compression:

- Batches are typically compressed to save storage space and reduce data transfer size
- This compression adds a small processing cost but improves the overall storage and data transfer efficiency.

Streams:

- Kafka handles messages as opaque byte arrays (raw data without predefined structure
- Kafka recommends structured schemas to:
  - * Enhance readability and understanding of message content
  - * Decouple producers and consumers:
    - ▸ Producers can send data independently of how consumers process it
    - ▸ Enable independent updates to producers and consumers.
  - * Support flexibility for changing or evolving data formats.

- Common schema formats are: JSON, XML, and Apache Avro depending on the application's needs:

* JSON & XML:
  - Human-readable: are easy for developers to read and understand, making them user-friendly.
  - Lack of advanced type handling: do not enforce strict data types (eg: integers/strings ...), leading to potential type-related errors.
  - No schema evolution support: They don't provide built-in mechanisms for managing changes in data structure across versions, making backward and foward compatibility harder to maintain

→ Many Kafka developers favor the use of Apache Avro

* Apache Avro:
  - Def: A serialization framework originally developed for Hadoop.
  - Key features:
    - Compact serialization format: Avro efficiently encodes data to reduce storage space and transmission size
    - Schema Separation: schemas are stored independently from the actual message data, simplifying management and updates
    - No code Regeneration: schema changes don't require regenerating or updating application code, making updates seamless.
    - Strong Data Typing: Ensures that data types are explicitly defined and enforced for consistency.
    - Schema evolution: Supports changes to schemas while maintaining compatibility
      - Backward compatibility: New schemas can read data written with older schemas
      - Foward compatibility: Older schemas can read data written with newer schemas

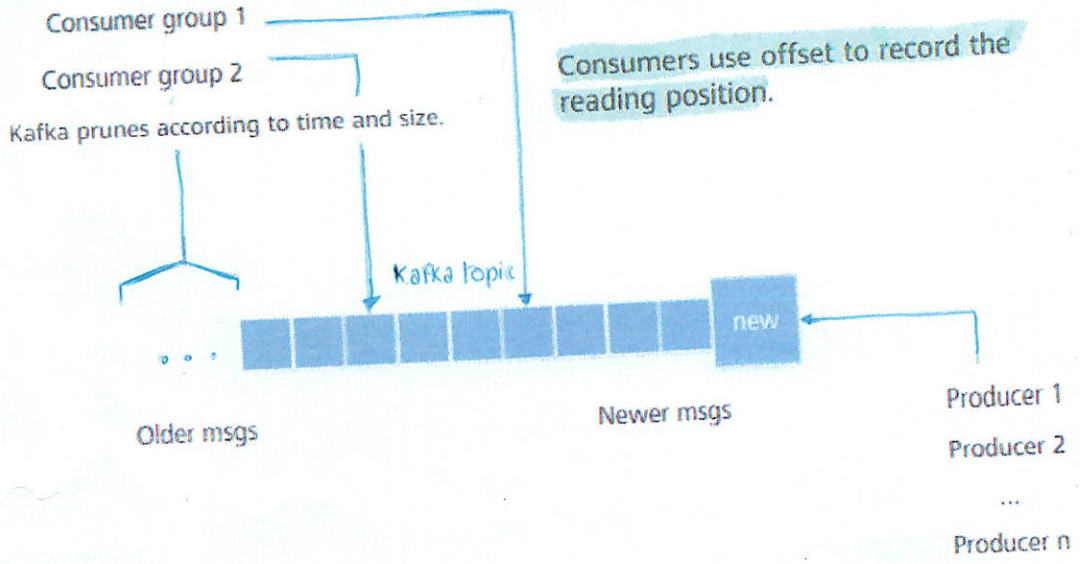
Topics and Partitions:

Topics:

Definition:

For instance let's consider the broker as the Kafka server node

- Kafka organizes (categorizes) messages into topics
  * Each message published to Kafka belongs to a category which is called topic
  * Topics are logical categories & organizational units for messages, similar to tables in a database or folders in filesystems.
- A topic can be considered as a combination of one or multiple message queues
  * Eg: Weather can be considered as a topic (queue) that stores daily temperature information.
  * Each queue may contain numerous messages, which are ordered based on their generation time



Producers always append messages to the end of a queue.

Characteristics:

- **Logical grouping:** Messages in a topic are logically related (eg: a weather topic for temperature data)
- **Subscriber Model:** Consumers subscribe to topics to receive relevant data.
- **Partitioning:** Topics are divided into a number of partitions.

Stream:

- Refers to:
  - a single topic of data (Regardless of the number of partitions)
  - the continuous flow of data within a specific topic.
- ⟶ Represents data moving from producers to consumers.

Partitions:

Definition:

- Topics are divided into partitions, which are fundamental storage and processing units
  - a topic typically has multiple partitions
  - In most cases there are many queues in each topic (because multiple clients can concurrently access different nodes to read and write messages of a topic)
    - ↳ we give each queue a name called partition
- Each partition represents a single commit log:
  - **Ordered Log:** Each partition is an ordered and immutable sequence of messages:
    - Messages within a partition maintain a strict order: They are read from beginning to end
    - **Note:** There's no guarantee of message ordering across the entire topic.
  - **Append only:** New messages are always added at the end.
  - **Unique offsets:** Messages in a partition are uniquely identified by their offset.

- Offset is a unique identifier for each message in a partition
- The order is recorded by the offset.

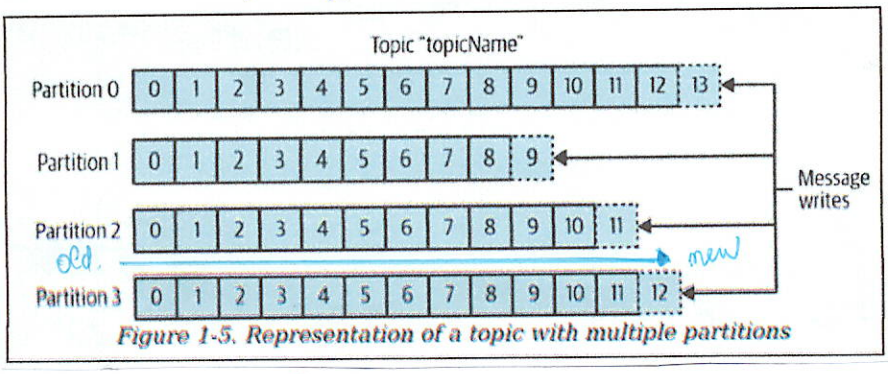


Figure 1-5. Representation of a topic with multiple partitions

Partition Replication and distribution:

- Partitions are also the way that kafka provides redundancy and scalability
- Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers to provide performance far beyond the ability of a single server.
- **Replication:** Creating backup copies of partitions to ensure data redundancy, such that different servers will store a copy of the same partition.
- **Distribution:** Partitions are distributed across multiple servers (brokers) to balance load and improve scalability.
- Partition replication and distribution has several benefits including: → performance benefits
  - * Facilitates concurrent data processing, improving throughput and enabling high performance.
  - * Ensures fault tolerance by enabling failover to replicas during server or leader failures
    - ↳ If a broker fails, replicas can take over to maintain availability. → Reliability benefits
  - * Allow efficient processing of massive data volumes (supports large-scale data processing) → Scalability benefits

Tasswin
replicat

Consumers and Producers:

- Kafka clients are users of the system, and there are 2 basic types: producers and consumers.
- There are also advanced client APIs:
  - Kafka Connect API for data integration
  - Kafka Streams for stream processing.
- The advanced clients use producers and consumers as building blocks and provide higher-level functionality on top.

Producers:

- In other publish/subscribe systems, these may be called publishers or writers.
- **Purpose:** Create and send messages (data) to Kafka brokers, which store them under specific topics for later consumption by consumers.
- **Functionality:**
  - A message will be produced to a specific topic
  - The producer handles the partitioning logic
    - By default (If no specific key provided), the producer distributes messages evenly across all partitions of a topic → This ensures load balancing.
    - If a key is provided with a message, the producer uses a partitioner to consistently send messages with the same key to the same partition
      → This is useful when order or grouping is important.
    - Developers can implement custom logic to control how messages are distributed across partitions, overriding the default partitioner.

Note: The broker simply stores the messages in the partitions assigned by the producer.

Consumers:

- In other publish/subscribe systems, these may be called subscribers or readers.
- **Purpose:** Reads messages from Kafka brokers by subscribing to specific topics.
- **Functionality:**
  - A consumer subscribes to one or more topics and consumes data from their partitions.
  - Consumers read messages from topics in the order they were produced to each partition.
  - The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages.
    - The offset is an integer value that continually increases. It is a piece of metadata that Kafka adds to each message as it is produced.
    - Each message in a given partition has a unique offset. By storing the next possible offset for each partition, typically in Kafka itself, a consumer can stop and restart without losing its place.

- **Consumer groups:**
  - Consumers work as part of a consumer group, which is one or more consumers that work together to consume all partitions in a topic
  - Consumer instances in the same group share a common GroupID
  - The group ensures that:
    - Workload is balanced across group members for scalability.
    - If a consumer fails other members of the group will reassign the partitions being consumed to take over for the missing member.
    - Each consumer belongs to a consumer group
    - Each message can be consumed by multiple consumer groups.
    - Each partition is consumed by only one group member.
  - → Data is shared between groups but exclusive within a group.

eg1:

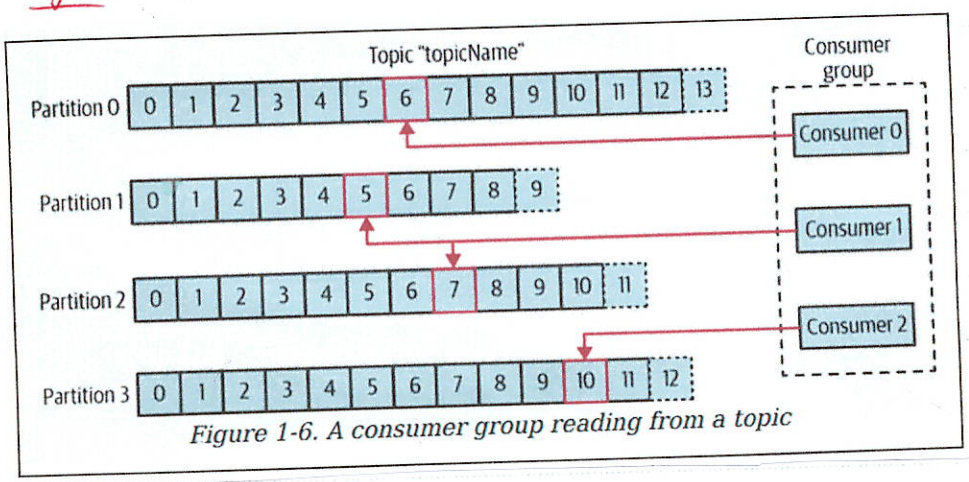
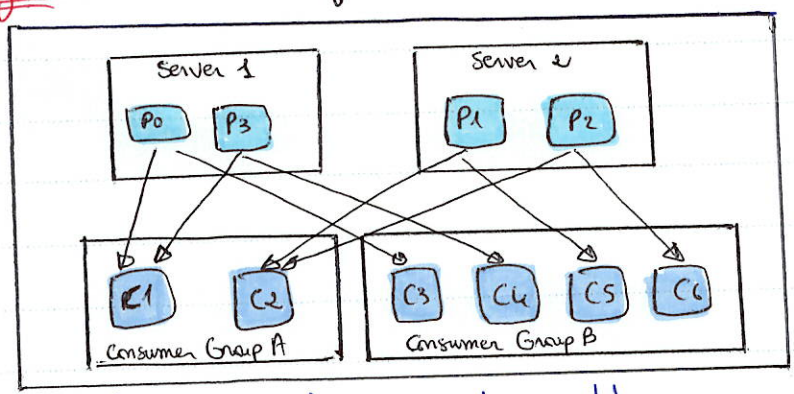


Figure 1-6. A consumer group reading from a topic

- There are 3 consumers in a single group consuming a topic.
- 2 of the consumers are working from one partition each, while the third consumer is working from 2 partitions.

eg2:

Kafka Cluster.



- In this figure, there are 2 servers and each server has exactly 2 partitions
- In group A, there are 2 consumers
  - → If C1 consumes P0 then C2 can't consume P0 anymore: That's why, each consumer in the group A consumes only 2 partitions.
- In Group B, there are 4 consumers:
  - → If C3 consumes P0 then C4, C5, C6 cannot consume P0 anymore: That's why, each consumer in the group B consumes only 1 parti

- The mapping of a consumer to a partition is often called ownership of the partition
  - → In this way, consumers can horizontally scale to consume topics with a large number of messages.

Brokers and Clusters:

Broker:

Definition and Role:

- A kafka broker is a single server that:
  - * Receives messages from producers
  - * Assigns offsets to messages.
  - * Writes messages to disk for storage
  - * Serves consumers by providing requested data
    - ↳ Responding to fetch requests for partitions and responding with the message that have been published.

⟹ Brokers are passive: Producers push data to brokers and consumers pull data when needed.

Performance and scalability:

- Depending on the specific hardware and its performance characteristics, a single broker can handle:
  - * Thousands of partitions
  - * Millions of messages per second
- Kafka brokers operate as part of a cluster for scalability and reliability

Data Retention:

What is Retention?

- Retention means the durable storage of messages for a set amount of time or until a certain size limit is reached

Default Retention settings:

- Retention by time: retaining messages for some period of time (eg: 7 days)
- Retention by size: retaining messages until the partition reaches a certain size in bytes (eg: 1GB per partition)

What happens after retention ends:

- Once the time or size limit is reached, old messages are expired and deleted.

→ This ensures only relevant and recent data is kept, saving storage space.

Custom Retention for topics:

- Each topic can have (can be configured with) its own retention settings:
- eg:
  - * A tracking topic might keep data for several days
    - ▸ A tracking topic is used to store and process data related to user activity, system events, or any kind of tracking information.
  - * Application metrics might only need to be stored for a few hours

Log Compaction: (Special Retention Mode)

I'll delve more into this later

- Topics can also be configured as log compacted.
  * Kafka retains only the last message produced with a specific key.
- Useful for changelog-type data where only the most recent update matters
  * Changelog-type data refers to data that represents changes or updates to specific items or entities over time.

Data Management within a cluster:

Overview:

- A kafka cluster is a group of brokers working together to:
  - * Distribute and replicate data
  - * Provide fault tolerance and high availability.
- Kafka brokers are designed to operate as part of a cluster.

Leadership and Replication:

Replication:

- Kafka allows replication to improve the data reliability
- Each partition in Kafka has its own primary and secondary replicas:
  - * The primary replicas is called leader.
  - * The secondary replicas is called Follower
- The leader and follower needs to be synchronized and they should share the same messages all the time
  - ↳ Data is synchronized between replicas.
- Although having the same pieces of data, the leader and follower have different responsibilities

Leader Broker:

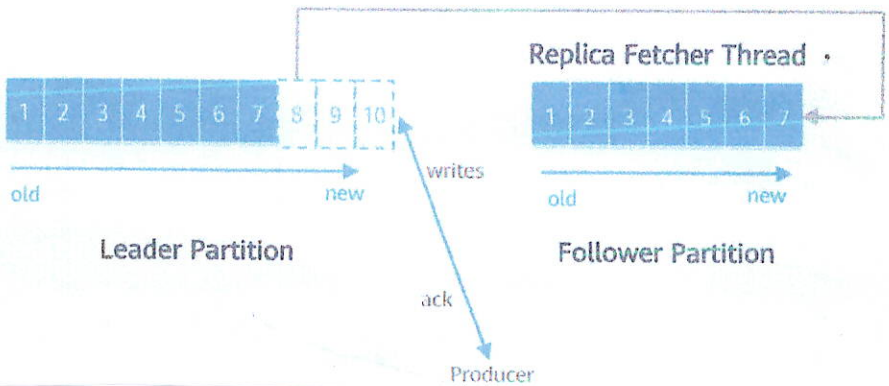
- A partition is owned by a single broker in the cluster, and that broker is called the leader of the partition.
- The Leader broker handles all read/write operations for its partitions.
  - * Producers do not interact with followers.
  - * Producers connect to the leader to publish messages.

Follower Brokers:

- A replicated partition is assigned to additional brokers called followers of the partition:
  - * Follower Brokers replicate replicate data from the leader.
  - * Replication provides redundancy of messages in the partition, such that one of the followers can take over leadership if there's a broker failure.

Note: All producers must connect to the leader in order to publish messages, but consumers may fetch from either the leader or one of the followers.

Follower Pulls Data from Leader



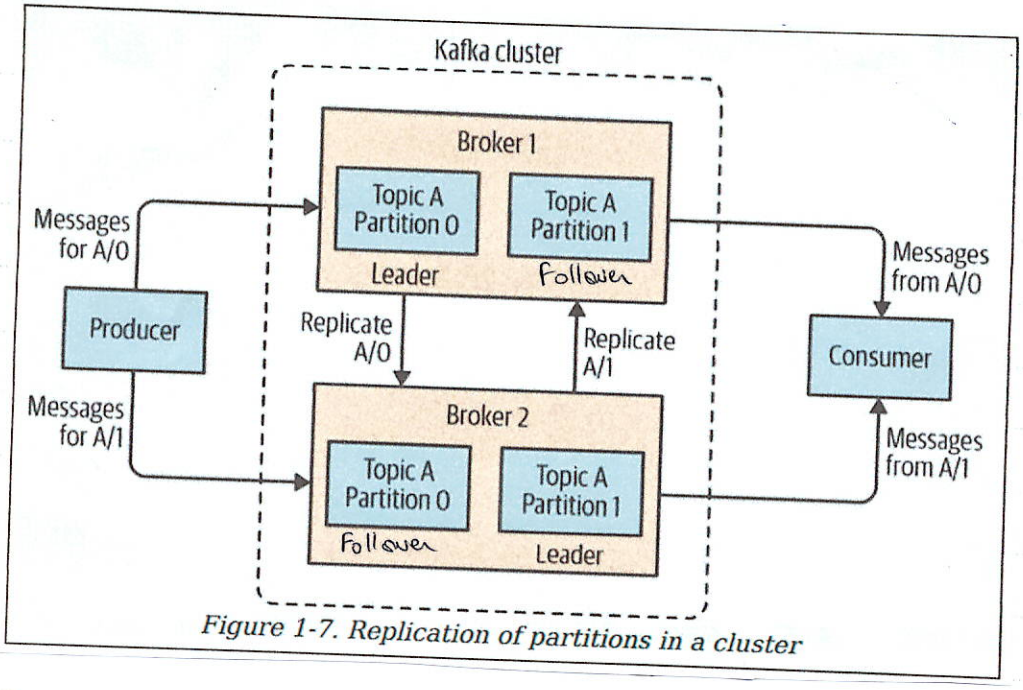


Figure 1-7. Replication of partitions in a cluster

Controller Broker:

- Within a cluster of brokers, one broker will also function as the cluster controller.
  - elected automatically from the live members of the cluster
- A controller broker is automatically elected to be responsible for administrative tasks, including:
  - * Assigning partitions to brokers.
  - * Monitoring broker health (failure) and handling failovers.

Zookeeper (Controller election)