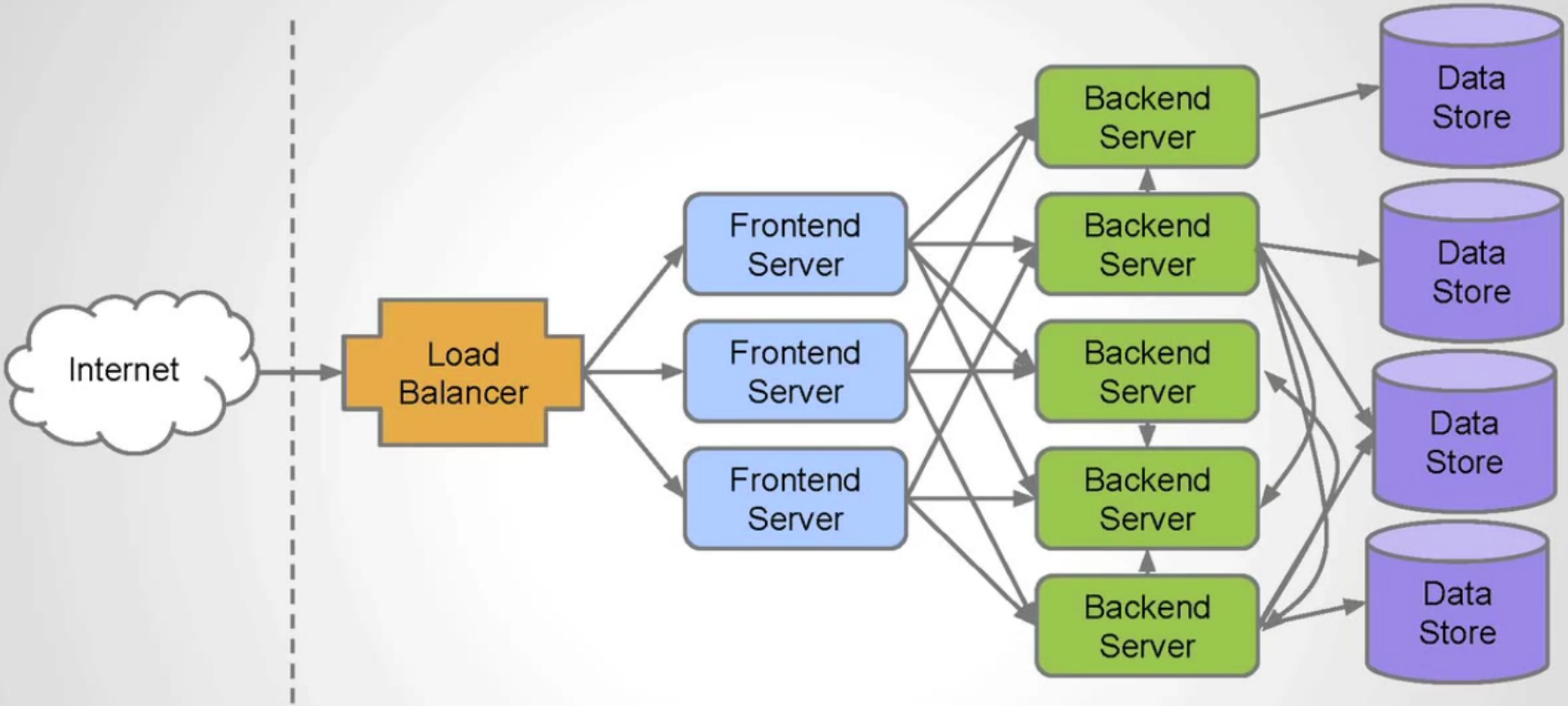
Data Infrastructure @ LinkedIn

World’s largest professional network

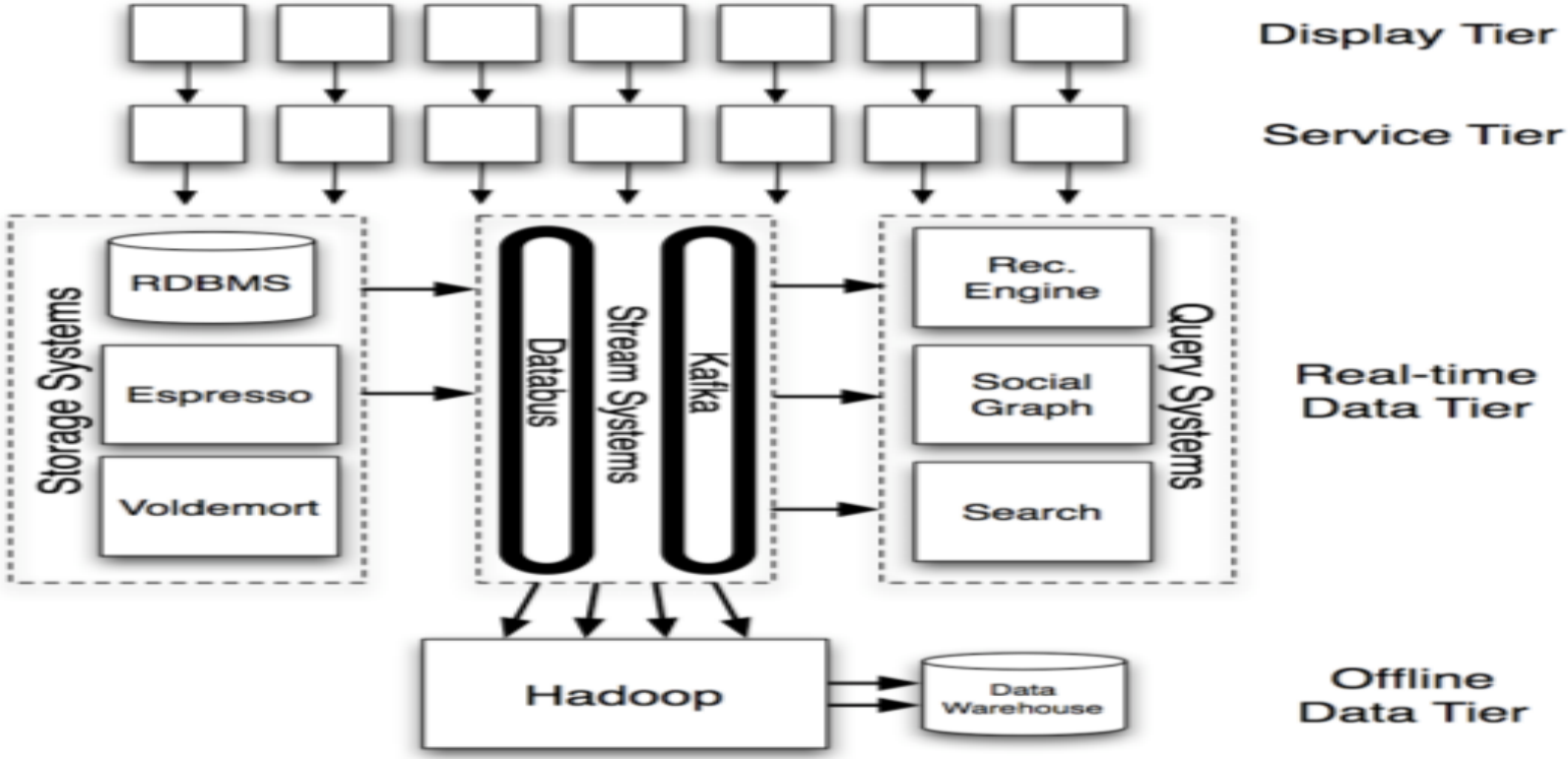
Mostly built on open source stack

- Voldemort
- Espresso
- Data bus
- Kafka



For data, LinkedIn uses a Service Oriented Architecture

LinkedIn Site's Architecture



Architecture Highlights

- 3 tier.
- Stateless.(machines interchangeable).
- Push state down to data systems.
- Partitioning data over multiple machines
- Intelligently routing requests to the machines where data resides.
- Availability using replication of data across servers

Concerns at LinkedIn

- Availability requires replicating data onto multiple machines.
- Reasoning about consistency .
- Expansion requires redistributing data over new machines without downtime or interruption.

LinkedIn Core Data Systems

- Live storage
- Stream systems
- Search
- Social Graph
- Recommendations

Live storage systems

- Simple variations on OLTP databases
- Live storage systems are work horses of web applications, serving the majority of data requests that make up the user experience.
- Voldemort (fast, simple, consistent key value store).
- Espresso (timeline consistency model).

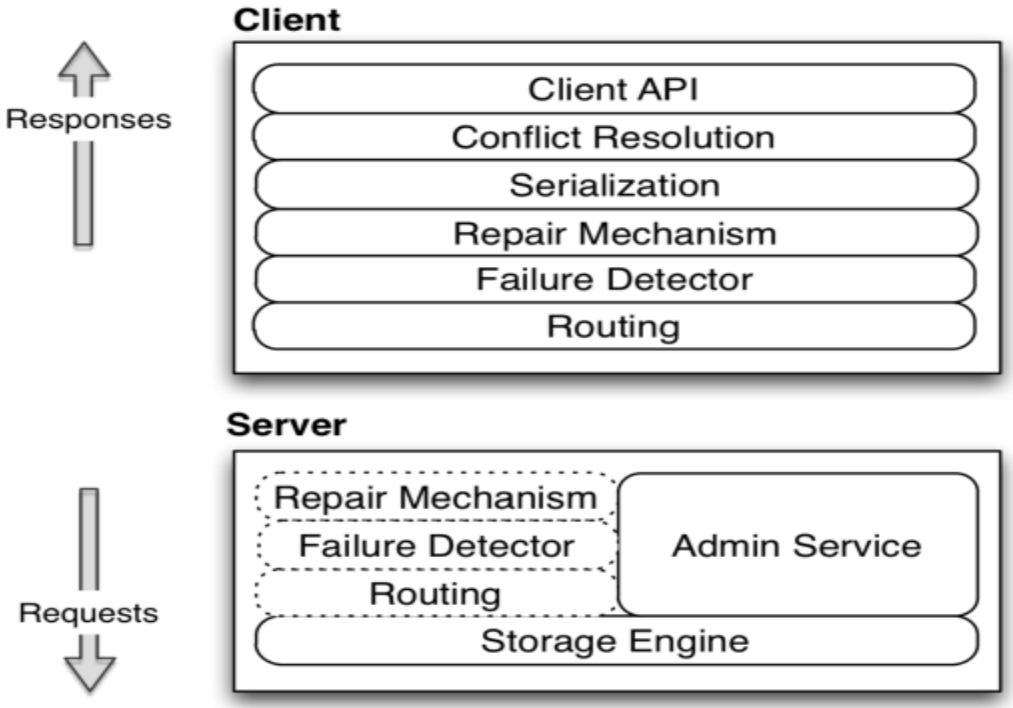
Voldemort

- Inspired by Amazon’s dynamo
- Data is automatically replicated over multiple servers.
- Data is automatically partitioned so each server contains only a subset of the total data
- Provides tunable consistency (strict quorum or eventual consistency) Server failure is handled transparently
- Pluggable Storage Engines -- BDB-JE, MySQL, Read-Only
- Pluggable serialization -- Protocol Buffers, Thrift, Avro and Java Serialization

Voldemort

- Data items are versioned to maximize data integrity in failure scenarios without compromising availability of the system.
- Each node is independent of other nodes with no central point of failure or coordination
- Good single node performance: you can expect 10-20k operations per second depending on the machines, the network, the disk system, and the data replication factor
- Support for pluggable data placement strategies to support things like distribution across data centers that are geographically far apart.

Pluggable Architecture of Voldemort



Espresso

- Espresso is a distributed, timeline consistent, scalable, document store that supports local secondary indexing and local transactions.

- ESPRESSO relies on Data Bus for internal replication and therefore provides a Change Data Capture pipeline to downstream consumers.

- Espresso bridges the semantic gap between a simple Key Value store like Voldemort and a full RDBMS.

Data Model and API

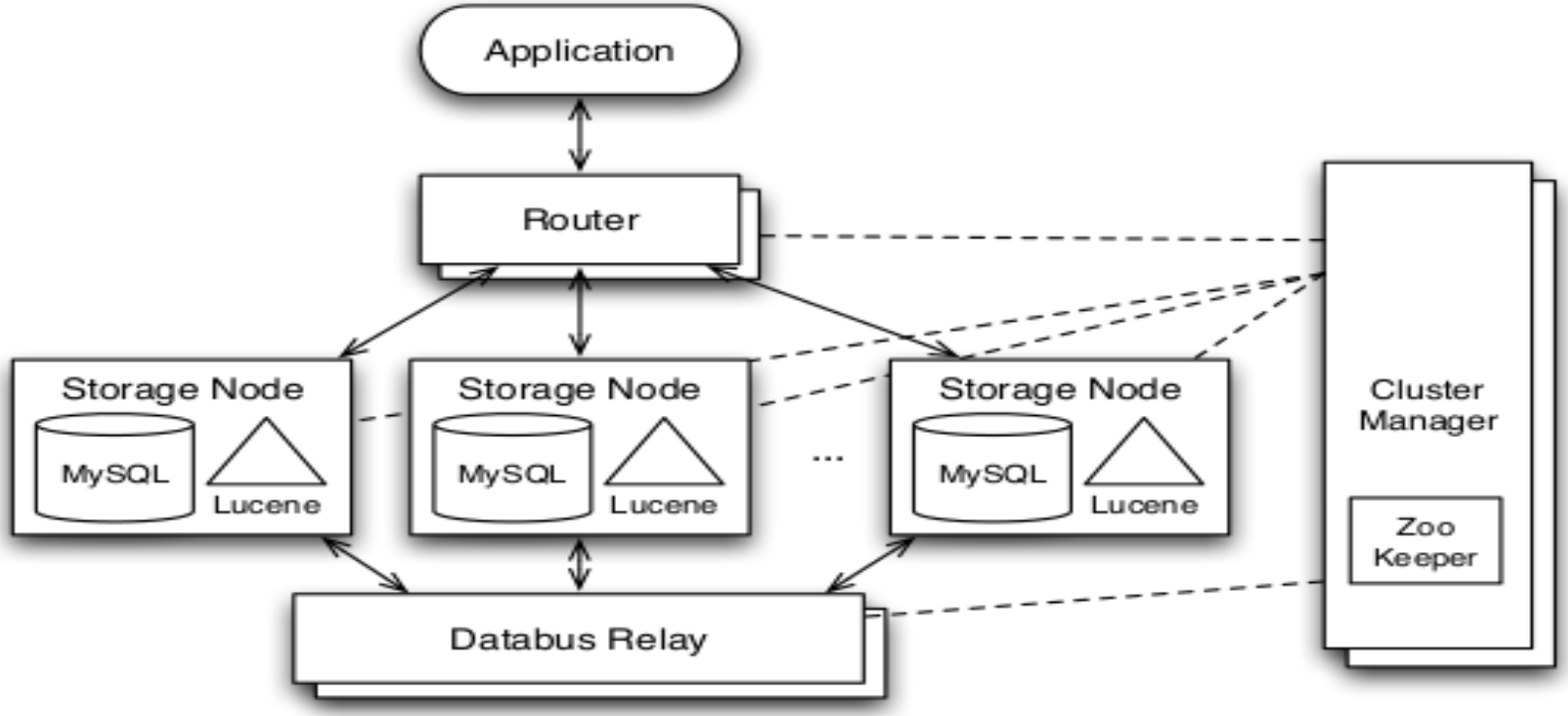
- Espresso is a document-oriented data store that provides hierarchical ordering of documents and local secondary indexing.
- In Espresso, a database is a container of tables. A table is a container of documents.
- Each database, table and document has an associated schema.
- Schemas are represented in JSON format.

Espresso System Architecture

The Espresso system consists of four major components:

- Router
- Storage node
- Relays
- Cluster managers

Architecture Design



Router

- The router accepts HTTP requests, inspects the URI and forwards the request to the appropriate storage node.
- Next it consults the routing table maintained by the cluster manager to determine the Storage node.
- Finally router forwards the HTTP request to selected node.

Storage Node

- The requests for a document are routed to the storage node that is the master of partition.
- Storage node maintains a consistent view of each document in local data store.
- The timestamp and etag fields are used to implement conditional HTTP requests.

Storage Node

- Requests for specific resources can be satisfied via direct lookup in the local data store.

- Queries first consult a local secondary index then return the matching documents from the local data store.

Relay

- Espresso replication lays the foundation for fault-tolerant and elastic solution.

- The Relay is designed to be
  - Timeline consistent.
  - Efficient.
  - Robust.

Cluster Manager

- The cluster manager is called Helix.
- It is a generic platform for managing a cluster of nodes including espresso storage nodes and Data Bus relay nodes.
- Helix is a state machine. It has the following states:
  - Ideal state.
  - Current state.
  - Best possible state.

Components of Cluster Manager

- Load balancing
- Robust hosted services
- Service discovery
- Server lifecycle management
- Health check

Databus

- Modern Internet-based systems are increasingly facing the difficult challenge of performing complex processing over large amounts of valuable data with strict upper bounds on latency.

- For example,LinkedIn have to continuously maintain indexes and statistical models over many aspects of the online professional identity of constantly increasing user base of 135+ million.

- LinkedIn built Databus, a system for change data capture (CDC), that is being used to enable complex online and near-line computations under strict latency bounds.
- It provides a common pipeline for transporting CDC events from LinkedIn primary databases to various applications.
- Among these applications are the Social Graph Index, the People Search Index, Read Replicas, and near-line processors like the Company Name and Position Standardization

Important requirement for Databus

Strong Timeline Consistency:

To avoid having the subscribers see partial and/or inconsistent data they need to capture

User-space Processing:

The ability to perform the computation triggered by the data change outside the database server.

Support for long look-back queries:

Subscribers are interested only in recent changes that they have not processed yet. Yet, there are occasions when subscribers may want to read older changes.

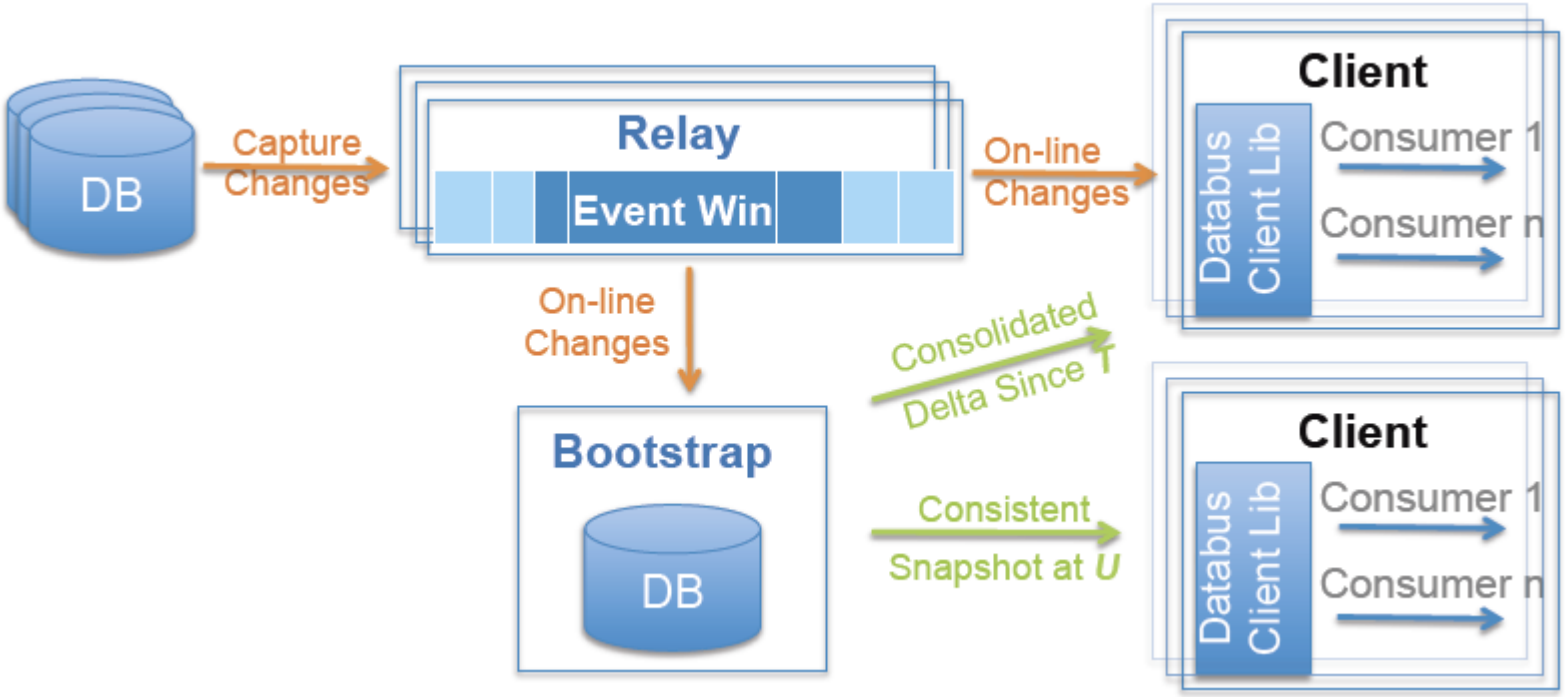
Data source / subscriber isolation:

Isolate the source database from the number of subscribers so that increasing the number of the latter should not impact the performance of the former

Databus Guarantees

- Timeline consistency with the data source with transactional semantics and at-least-once delivery.
- No loss in durability (relies on bootstrap and primary store on failure)
- High availability (replicated availability)
- Low latency (low milliseconds)

Architecture



The Databus pipeline consists of three main components: the Relay, the Bootstrap Server, and the Databus Client Library.

Relay :The Relay captures changes in the source database, serializes them to a common binary format and buffers those. Each change is represented by a Databus CDC event which contains a sequence number in the commit order of the source database, metadata, and payload with the serialized change.

Features:

The relay with the in-memory circular buffer provides:

- Default serving path with very low latency (<1 ms)
- Efficient buffering of tens of GB of data with hundreds of millions of Databus events
- Index structures to efficiently serve to Databus clients events from a given sequence number S
- Server-side filtering for support of multiple partitioning schemes to the clients
- Support of hundreds of consumers per relay with no additional impact on the source database

Bootstrap server:

- The main task of the bootstrap server is to listen to the stream of Data Bus events and provide long-term storage for them.

- It serves all requests from clients that cannot be processed by the relay because of its limited memory.

- Thus, the bootstrap server isolates the primary database from having to serve those clients.

Databus client library: The Databus client library is the glue between the Relays and Bootstrap servers and the business logic of the Databus consumers. It provides:

- Tracking of progress in the Databus event stream with automatic switchover between the Relays and Bootstrap servers when necessary.
- Push (callbacks) or pull interface
- Local buffering and flow control to ensure continuous stream of Databus events to the consumers
- Support for multi-thread processing
- Retry logic if consumers fail to process some events

Kafka

- A Scalable and Efficient messaging system for collecting various user activity events and log data.
- Kafka has large amount of event data generated at any sizable internet company. Data includes user activity events corresponding to logins, page-views, clicks, sharing, comments, and search queries.
- Kafka adopts a messaging API to support both real time and offline consumption of this data.
- Since event data is 2-3 orders magnitude larger than data handled in traditional messaging systems, yet practical design choices to make the system simple, efficient and scalable.

Architecture

- A stream of messages of a particular type is defined by a topic.
- A producer publishes messages to a topic. The published messages are stored at a set of servers called brokers.
- A consumer subscribes to one or more topics, and consumes the subscribed messages by pulling data from the brokers.
- Each message stream provides an iterator interface over the continual stream of messages being produced.
- If there are currently no more messages to consume, the iterator blocks until new messages are published to the topic.

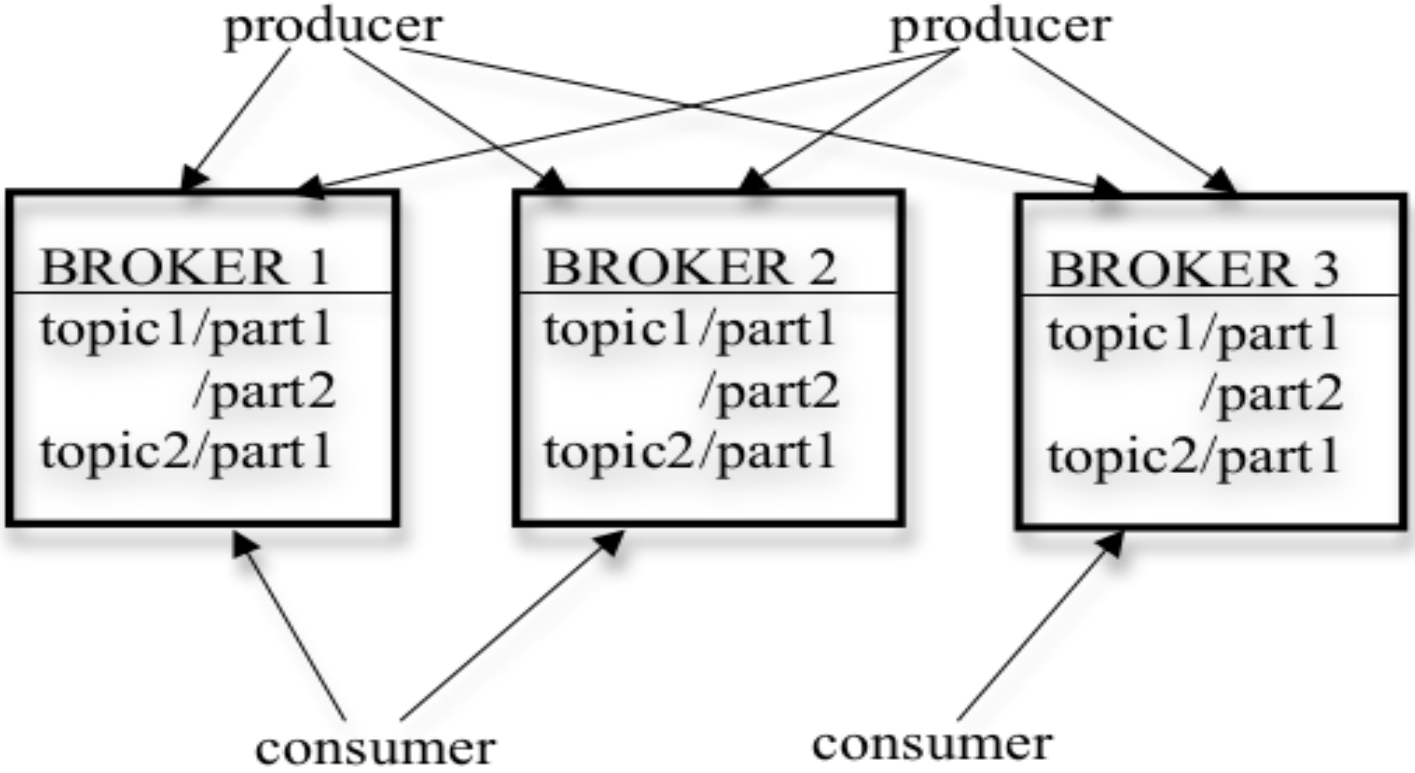


Figure V.1 Architecture

Storage

- Every time a producer publishes a message to a partition, the broker simply appends the message to the last segment file.
- For better performance, we flush the segment files to disk only after a configurable number of messages have been published or a certain amount of time has elapsed.
- Messages stored in Kafka doesn’t have an explicit message id. Instead, each message is addressed by its logical offset in the log.
- The broker locates the segment file where the requested message resides by searching the offset list, and sends the data back to the consumer.

Efficient Transfer

- Kafka supports compression, so producer can compress a set of messages and send it to the broker.
- The compressed data is stored in the broker and is eventually delivered to the consumer, where it is uncompressed.
- It can save about 2/3rd of the network bandwidth with compression enabled.
- Kafka doesn’t cache messages but it rely on filesystem page cache which avoids double buffering.
- Additional benefit of retaining warm cache even when a broker process is restarted.

Distributed Coordination

- Each consumer group consists of one or more consumers that jointly consume a set of subscribed topics, each message is delivered to only one of the consumers within the group.
- Different consumer groups each independently consume the full set of subscribed messages and no coordination is needed across consumer groups. The consumers within the same group can be in different processes or on different machines.
- To facilitate the coordination, Kafka uses Zookeeper for tasks like detecting the addition and the removal of brokers and consumers, triggering a rebalance process, keeping track of the consumed offset of each partition.