Apache Kafka Bootcamp

Agile Brains Consulting

CLOUD NATVZ

Welcome: Kafka Essentials

- Logistics
- Ground Rules
- Instructor Introduction
- Expectations

What to expect from this class

- Course is designed with participation in mind
- Learn the importance of messaging systems
- Learn the Essentials of Apache Kafka
- Hands on Practice with Kafka
- Learn to think about problems from a Big Data Perspective

Expected Outcomes

- Comfortable with how data moves through a data pipeline
- Understand where Kafka fits in the big data ecosystems
- Know the benefits of using Kafka over traditional pub/sub systems
- Strong Understanding of how Kafka Publishers and Consumers work
- Understand who is using Kafka and why
- The basics of deploying Kafka Clusters

Introductions

Name
Job Functionality
Experience Messaging, Microservices, Big Data

About Your Instructor

Who am I?

- Experience
- Big Data Background
- Interesting Fact

Housekeeping

1. One lunch break and 1 short morning and afternoon break
2. Please raise your hands for questions
3. If you must do work in the class please ask for a break.
4. This is a multi-day course. We will be talking in general terms for the sake of time. If you have something to add please do so in a manner that the entire class will benefit from otherwise we can discuss esoteric cases during break.

Connect to Lab Setup:

1. Find your VM instances and credentials provided by instructor

https://docs.google.com/spreadsheets/d/1mC8CjRr7GB5om4ExtuvaAbTS7okuHgZsxI1LQ02D_S4/edit?usp=sharing

1. Follow instruction to ssh into machine.

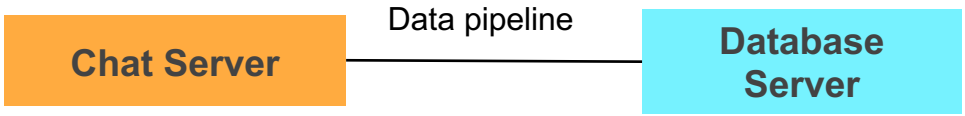
https://drive.google.com/file/d/1oaR0BTLjUh9uy57DTv5CBC8-cfDc4oKL/view?usp=sharing

Data Intensive Applications

If an application is data intensive its primary challenge is the quantity of data. As opposed to compute intensive where the main challenge is CPU speed. Some of the solutions we have used for coping with DIA

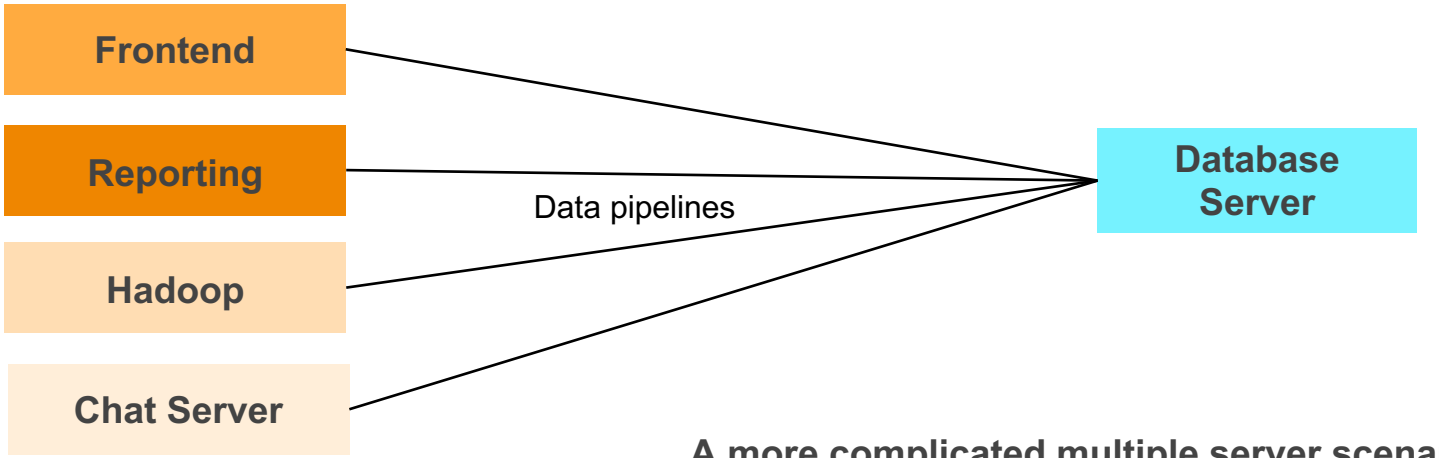
Need of Messaging Systems

Communication is required between different systems in the real-time scenario, which is done by data pipelines



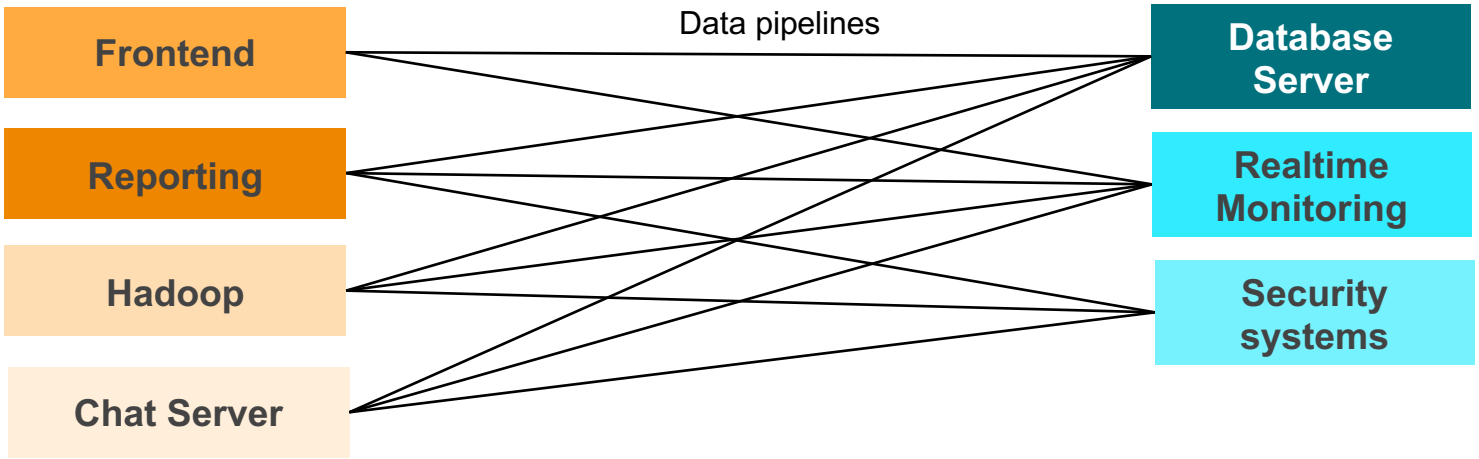
Simple two server scenario:
Chat server needs to communicate with database server for storing messages

Need of Messaging Systems



A more complicated multiple server scenario:
Multiple servers needs to be connected to database server

Need of Messaging Systems



A complex multiple server scenario:
Multiple servers needs to be connected to multiple server

Need of Messaging Systems

Transactional data pipeline

- Robust and Fault Tolerant
- Cannot miss a single transaction

Click stream data pipeline

- May be Fragile

Catalog Server to Web Server data pipeline

- Needs to be more reliable (cannot show a product that is not present)

Varied requirements

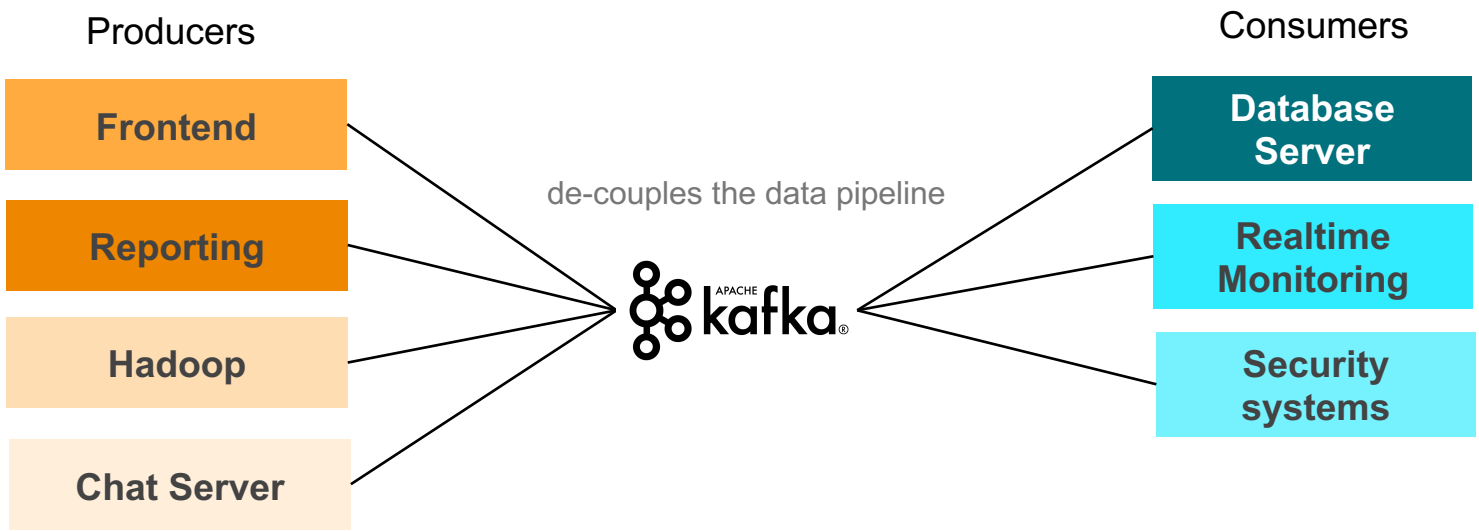
Increasing data and nodes

Messaging Systems

Why Messaging Systems

- Reduces complexity of data pipelines
- Communication b/w systems manageable
- Establish remote communication and send data across networks
- Provides common paradigm for different platform and languages
- Async communication between systems
- Reliable communication by persistence

How Kafka Helps



Data Intensive Applications

If an application is data intensive its primary challenge is the quantity of data. As opposed to compute intensive where the main challenge is CPU speed. Some of the solutions we have used for coping with DIA

DIA Examples

- Messaging queues
- Caches
- Search indexes
- Frameworks for batch
- Frameworks for stream
- NoSQL DB’s

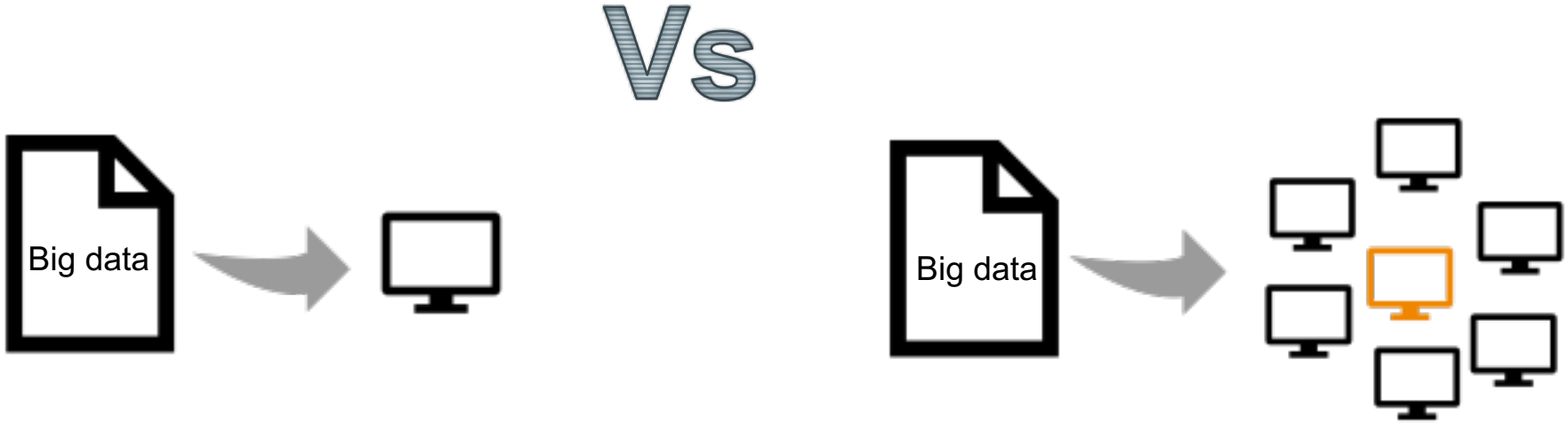
What are some specific products?

Data Intensive Applications

Distributed Computing is designed for DIA

A **distributed system** serves to co-ordinate the actions of several computers/systems. This co-ordination is achieved by exchanging **messages**.

Distributed Computing



Properties of Distributed System

- Reliability
- Scalability
- Availability
- Efficiency
- Partition Tolerant

- Data Replication
- Consistency
- Failure Management
- Failure recovery
- Distributed Transactions

CAP Theorem (Consistency, Availability, Partition Tolerant)

Data Replication and Consistency

- Loss of server holding unique copy of data is an unrecoverable damage
- Important to distribute read/write operations

Issues with replication:

- Performance: needs more disk, writing to several copies takes more time
- Consistency: same copy of information should be present in all replicas

Types of Consistency

- Strong Consistency
    - Data will get passed to all replicas as soon as a write request comes to one of the replicas, but during this time when the replicas are being updated with new data, response to any subsequent read/write requests by any replicas will get delayed as all replicas are busy in maintaining consistency
- Eventual Consistency
    - Make sure that data of each node of the database gets consistent eventually, but will take some time. Thus offer low latency at a risk of returning stale data

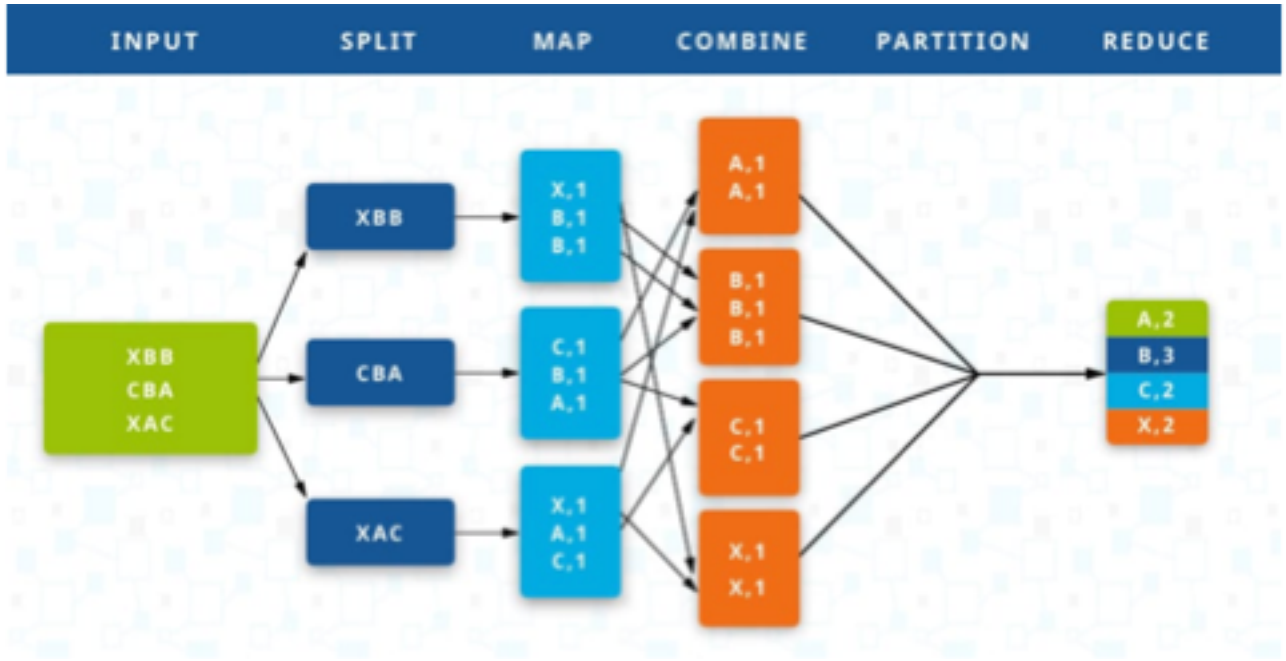
Failure Management and Recovery

In large distributed systems, failures are inevitable due to network errors, program bugs, out of disk|memory, etc.

- Independence – task handled by one node is independent of others
- Replication – Same data is replicated in multiple nodes
- Multiple master – master is single point of failure, so have multiple masters
- Database writing logs to centralized servers

Distributed Transactions

MapReduce – one of the most common algorithm, invented and used by Google in Big Table.



Reliable System

Ability of a distributed system to deliver its services even when one or several of its software of hardware components fail

Goals

- Performs functions as expected
- Can tolerate user making mistakes
- Performance is sufficient for given use case under given load and data volume

Pre-requisites

- Fault Tolerance
- Redundancy
- Eliminate single point of failure

Scalability

Ability of a system to continuously evolve in order to support a growing amount of tasks, so to respond to increases in load

- What does your systems current load look like?
- How does your system perform as load increases?
- What are your approaches to coping with load?

Scalable Applications

Vertical Scaling

Horizontal Scaling

Fault Tolerance

- Fault vs. Failure vs. Error

Fault - when a component of a system deviates from expected behavior

Error - the part of a system state that deviates from expected output

Failure - when the system quits providing the required function or service.

Designing fault tolerant systems become even more important now that we are working in service oriented architecture where we may be using API’s and services that our organization has very little control over.

Fault Tolerance

We assume that systems will never be 100% fault proof so we design our systems to be fault tolerant. These systems are designed to fail safely when faults occur. This can be achieved in different ways.

- Fail backwards
- Fail forward
- Redundant Backups

Fault Tolerance

Optimal fault tolerant systems often have a mechanism that randomly injects faults into the system to test how the system responds to those faults, and help build confidence in the system.

Netflix Chaos Monkey

https://www.gremlin.com/blog/adrian-cockroft-chaos-engineering-what-it-is-and-where-its-going-chaos-conf-2018/

Fault Tolerant Systems

Not Fault Tolerant

We were not able to save files(error) to the AWS Simple Storage Service(failure) today due to an error in the code(fault) that retrieve info from the DNS service. This resulted in our application being down for half the day(failure). This made our team rethink the fault tolerance of our application.

Fault Tolerant

We were not able to save files(error) to the AWS Simple Storage Service(failure) today due to an error in the code(fault) that retrieve info from the DNS service. We have a two tier system that helped our application to respond and recover once S3 came back online causing no downtime in our application. The files were saved to a drive in our corporate cloud and our Kafka cluster had all the S3 request queued and ready to continue where we left off, writing to S3 as soon as S3 came back on line.

Availability

Capacity of a system to limit as much latency as possible, caused by unavailable servers (like crashed servers)

- Failures (crash, unavailability, etc.) must be detected as soon as possible
  - Needs periodically monitoring of system heartbeat (typically done by Master node)
- Failover - Quick recovery mechanism
  - Replication and redundancy

Partition Tolerant

The cluster continues to function even if there is a "partition" (communication break) between two nodes (both nodes might be up, but can't communicate)

Example:- Gossip Protocol (used by Cassandra)

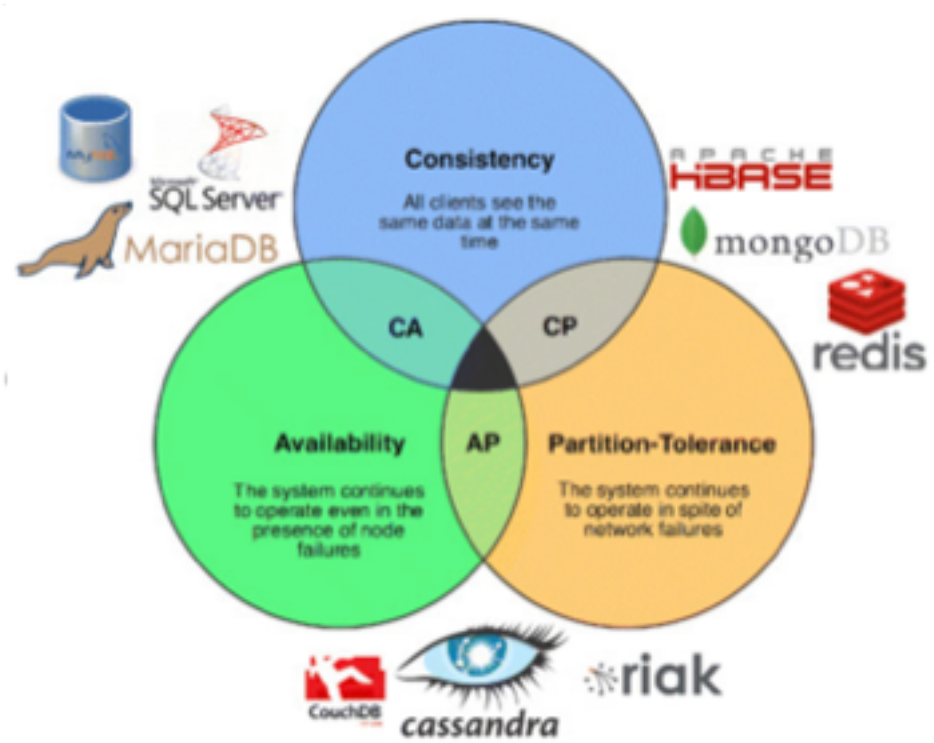
- Node added to the cluster gets registered with the **gossiper** to receive communication
- The gossiper selects a random node to check if it is alive
- Differentiate between failure detection and long running transactions, Cassandra implemented {*Phi Accural Failure Detection Algorithm*}

Putting it all together – CAP Theorem

CAP – Consistency, Availability Partition Tolerant

YOU CANT GET ALL THREE !!!!

Trade-off between consistency and availability, partition-tolerant has to be dealt anyway



Ultimately - Efficiency

How do you measure efficiency of a distributed system.

- Response Time or latency
  - Needs periodically monitoring of system heartbeat (typically done by Master node)
- Throughput (bandwidth)
  - Replication and redundancy

Few more Properties of Distributed System

- Partitioning
- Consistent Hashing

Partitioning

Basics

- Divide data into multiple chunks
- Place chunks on different nodes
- Both read and write load gets distributed
- Chunks are called shards or partitions or vnodes etc.

Must for a scalable solution !!

When you add more nodes in the cluster:

- New node will hold a new partitions with new data

Partitioning

Pre-requisites of partitioning

- Fair partition – to ensure uniform load
- In skewed partitioning, a few partitions will handle most of the requests
- Highly loaded partitions becomes bottleneck for the system – called **hotspots**
- Can’t be round-robin – loses track of where data goes
- quick lookup capability

*Each data record has an **Id** or **Key***

Partitioning Strategies

1. Key Range Partitioning

Divide the entire keyset into continuous ranges and assign each range to a partition.

Advantage: keys can be kept in a sorted manner within a partition. So, range queries like time series queries would have a better performance with this strategy.
Downside: Most data distribution is non-uniform, thus partitions doesn't get uniform range leading to hotspots

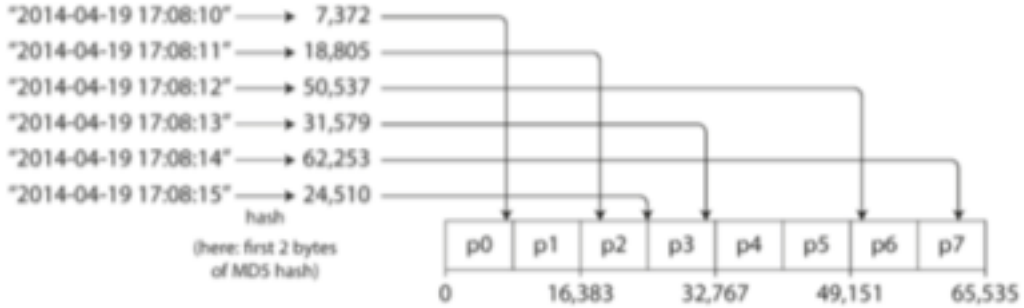
E.g. a bookshelf

Partitioning Strategies

2. Key Hash Partitioning

Find hash function of the key and take “mod N” where N is total number of partitions

Advantage: quick O(1) retrieval of keys; solves hotspot problems by randomness.
Downside: What happens when number of partitions are changed ? – “mod N” changes and all data previously assigned to “i” partition is no longer in “i” partition. Thus there is a need of re-partition means re-hashing – too expensive in large distributed system



Consistent Hashing

How to rebalance partitions when a new node is added/removed

Use a distribution scheme that does not depend directly on the number of partitions

- Assign partitions (servers) a position on an abstract circle, or hash ring by pseudo-randomly assigning them angles (from 0-360)
- Calculate “hash(key)” in “radians or degrees” that has a range from 0-360
- Keys for both data and partitions are on the same circle
- Define a association rule – data will belong to the partition whose key is closest, in a counterclockwise|clockwise direction

Consistent Hashing

KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"C"	C
"kate"	3421831276	123.1	"A"	A
"jane"	5000648311	180	"A"	A
"bill"	7594873884	273.4	"B"	B
"steve"	9786437450	352.3	"C"	C

Minimize re-partition

1. Operation - Add a node

If a 4th node D is added between A and C, only node A needs to be re-partitioned. Rest un-touched

2. Operation - Delete a node

If node A dies, only node B needs to be re-partitioned. Rest un-touched

** Trick for uniform load - Assign not one, but many labels (angles) to each server

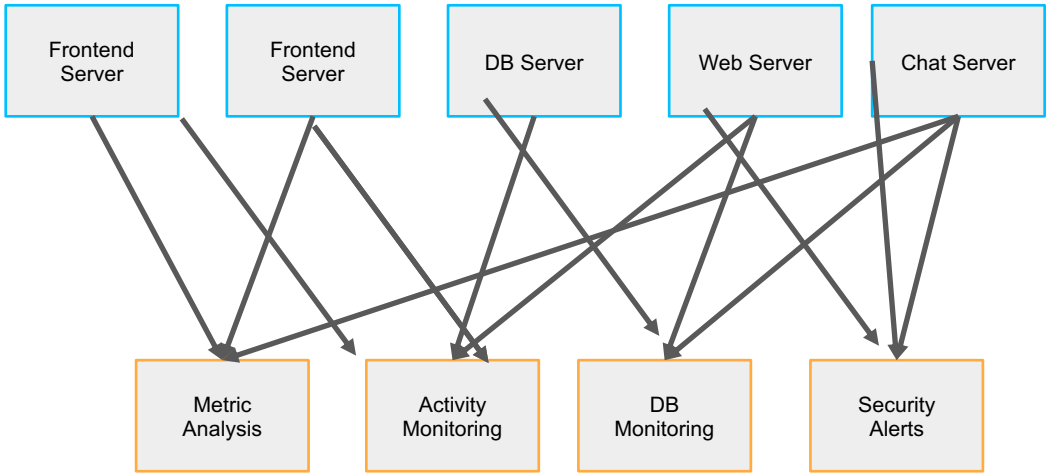
Coming back to

Messaging Systems

Traditional Architecture

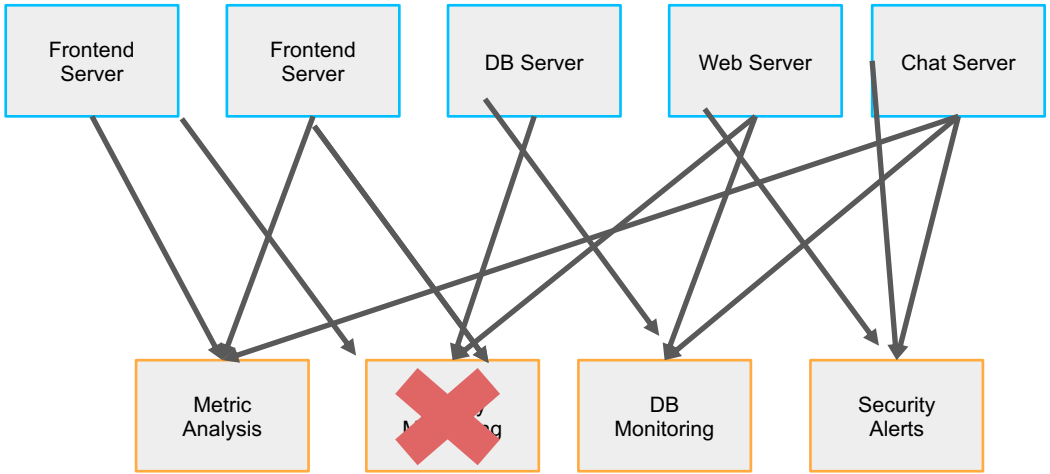
Traditional direct service to service communications are fast but the durability of the messages can be compromised if a service goes down or becomes overloaded.

What is the problem here?



Traditional Architecture

If one server goes down, the whole application is compromised



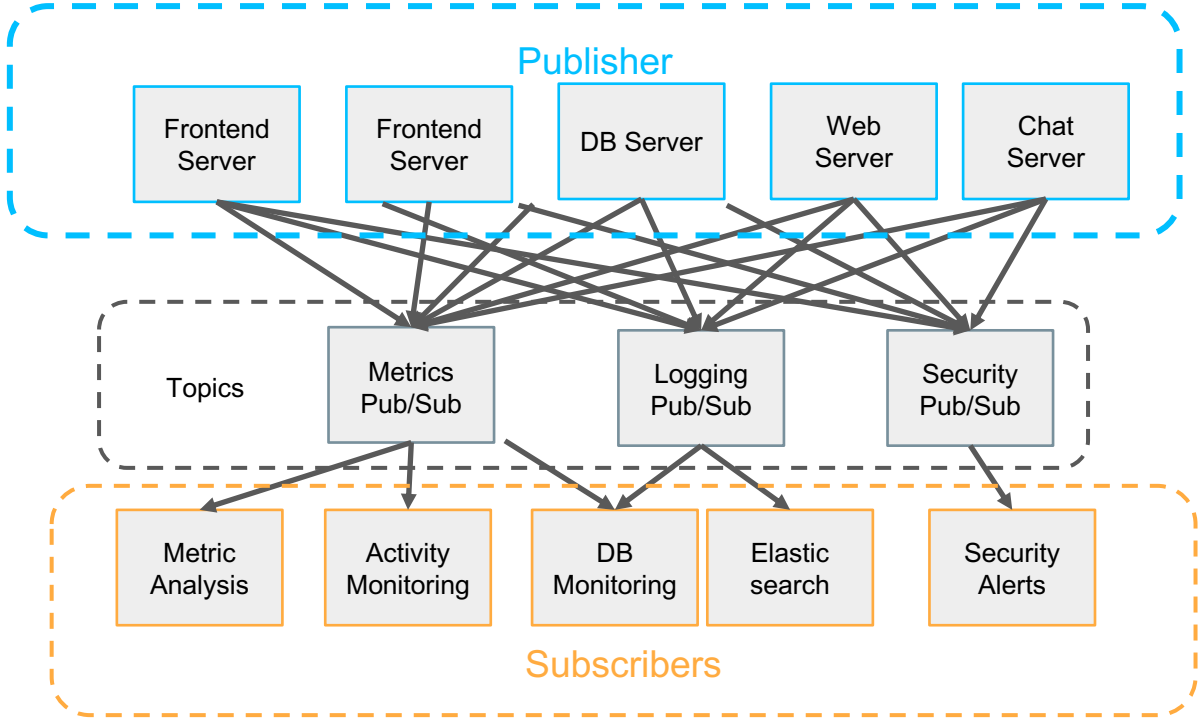
Pub/Sub Messaging System

Publish/subscribe messaging, or pub/sub messaging,

- A form of asynchronous service-to-service communication used in serverless and microservices architectures.
- In a pub/sub model, any message published to a topic is immediately received by all of the subscribers to the topic.
- Pub/sub messaging can be used to enable event-driven architectures
- Decouple applications in order to increase performance,
- Reliability and scalability

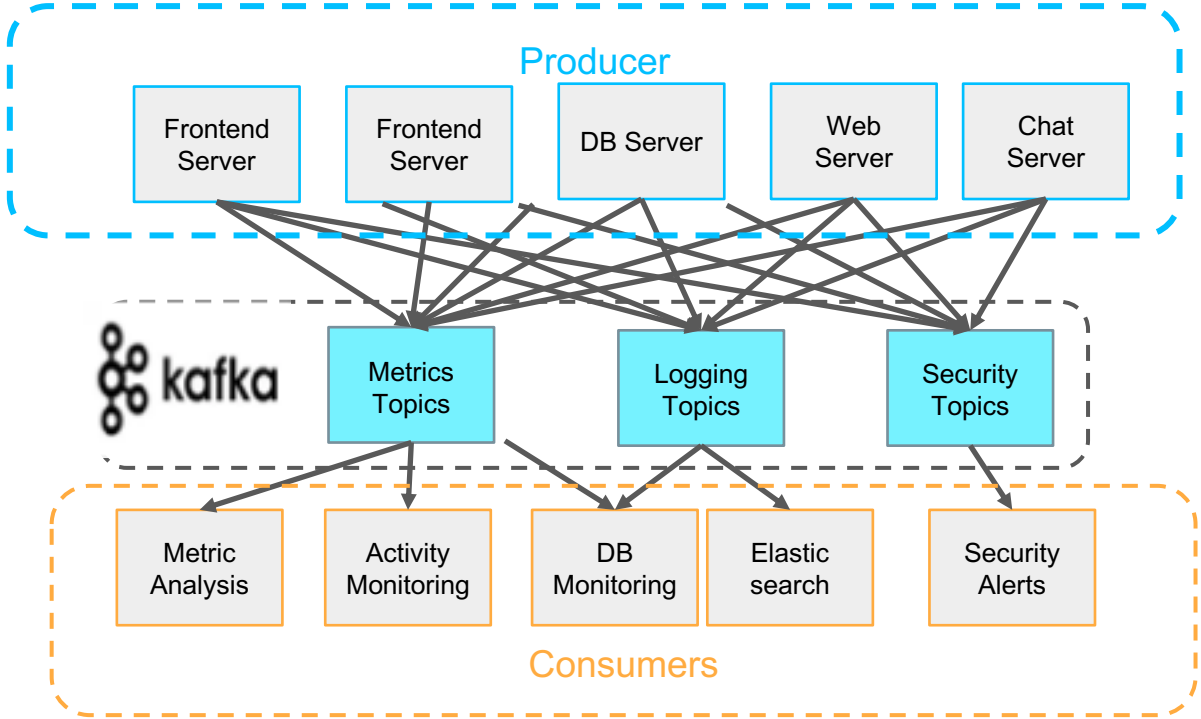
Pub/Sub Messaging

Pub/Sub messaging systems allow messages to be queued and communication between services to be buffered by a messaging service that allows service to pull data down in a reliable controlled manner.



Pub/Sub Messaging

Kafka is a decentralized distributed pub/sub messaging system that provides reliable communication between producers and consumer. It's distributed nature allows for a level of scalability and low latency that has made it the go to messaging system for service to service communication.



What is Apache Kafka?

- Pub/sub messaging system
- Developed at LinkedIn
- Low Latency
- Fault tolerant
- Scalable
- Durable
- Distributed

Why Kafka

- Great performance
- Operational Simplicity, Stable, Reliable Durability,
- Flexible Publish-subscribe/queue (scales with N-number of consumer groups),
- Robust Replication
- Message Process Guarantees
- Ordering Preserved at Topic Level
- Works well with systems that have data streams to process, aggregate, transform & load into other stores

Why Kafka (cont)

- Real time streaming data processed for real time analytics
- Service calls, track every call, IOT sensors
- Apache Kafka is a fast, scalable, durable, and fault tolerant publish-subscribe messaging system. Has higher throughput & reliability
- Kafka is often used instead of Amazon Simple Queue Service (SQS), Java Message Service (JMS), RabbitMQ, Advanced Message Queuing Protocol (AMQP)
- Kafka Support in EC2/AWS/Spark/Cassandra/etc.

Kafka Use Cases

- Stream Processing
- Website Activity Tracking
- Metrics Collection and Monitoring
- Log Aggregation
- Real time analytics
- Capture and ingest data into Spark / Hadoop
- CQRS
- Guaranteed distributed commit log for in-memory computing

Real Time Analysis

Kafka can works in combination with Real time Big Data processing systems Flume/Flafka, Spark Streaming, Storm, HBase and Spark for real-time analysis and processing of streaming data.

More on that later…..

Real Time Analysis

- Real time analysis for streams
  - Data analytics
  - ETL workloads
  - Complex event Processing
- Data Pipelines
  - In memory microservices
  - Message filtering

Batch Analysis

- Feed your data lakes with data streams
- Kafka brokers support massive message streams for follow up analysis in Hadoop or Spark
- Kafka Streaming (subproject) can be used for batch processing

Decoupling Services

- Decouples data streams
- Producers don’t know about consumers
- Flexible message consumption
- Kafka broker delegates log partition offset (location) to Consumers (clients)
- Consumers go offline messages are queued until ready for consumption

Kafka in Production

- 1/3 of all Fortune 500 companies
- Top ten travel companies, 7 of ten top banks, 8 of ten top insurance companies, 9 of ten top telecom companies
- LinkedIn, Microsoft and Netflix process 4 comma message a day with Kafka (1,000,000,000,000)
- Real-time streams of data, used to collect big data or to do real time analysis (or both) Cassandra / Kafka Support in EC2/AW

Who Uses Kafka

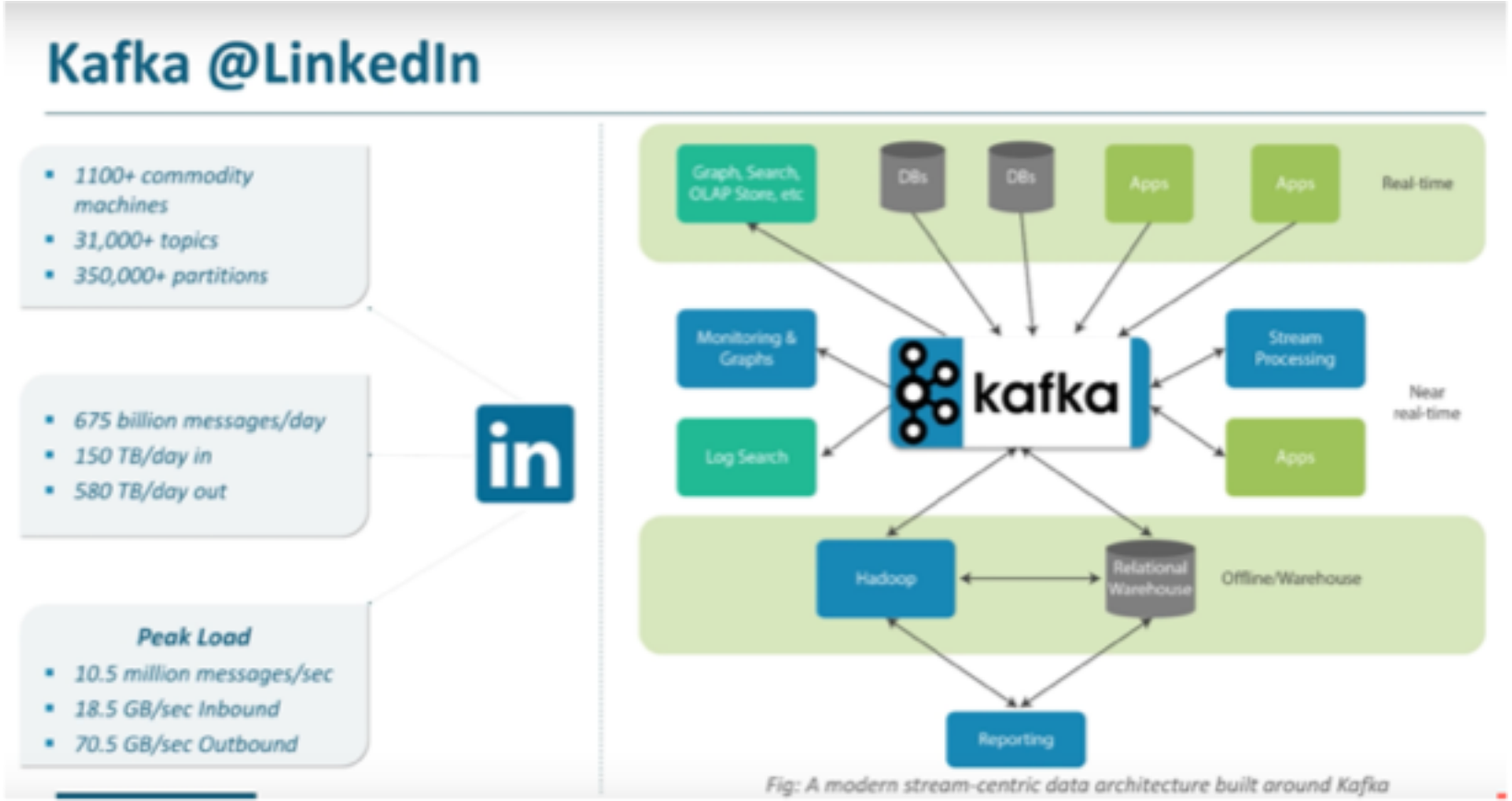
LinkedIn: Activity data and operational metrics

Twitter: Uses it as part of Storm – stream processing infrastructure

Square: Kafka as bus to move all system events to various Square data centers (logs, custom events, metrics, an so on). Outputs to Splunk, Graphite, Esper-like alerting systems

Spotify, Uber, Tumbler, Goldman Sachs, PayPal, Box, Cisco, CloudFlare, DataDog, LucidWorks, MailChimp, NetFlix, etc

Kafka in LinkedIn



Speed

- **Zero Copy** - calls the OS kernel direct to move data fast
- **Batch Data in Chunks** - Batches data into chunks
  - End to end from Producer to file system to Consumer
  - Provides More efficient data compression. Reduces I/O latency
- **Sequential Disk Write**s - Avoids Random Disk Access
- **Writes to immutable commit log-** No slow disk seeking. No random I/O operations. Disk accessed in sequential manner
- **Horizontal Scale -** can uses a large number of partitions for a single topic
  - spread out to thousands of servers
  - handle massive load

Lab 1: Install Zookeeper/Kafka (45 Min)

Objective:

Install a single instance of Zookeeper and Kafka on AWS EC-2 instance

https://github.com/shekhar2010us/kafka_t/blob/master/labs/01_install_zk_kafka_single_broker.md

Kafka Components

Kafka Components @ High Level

Topic
Category of a feed name to which records are published

Partition
Topics are broken up into ordered commit logs called partitions

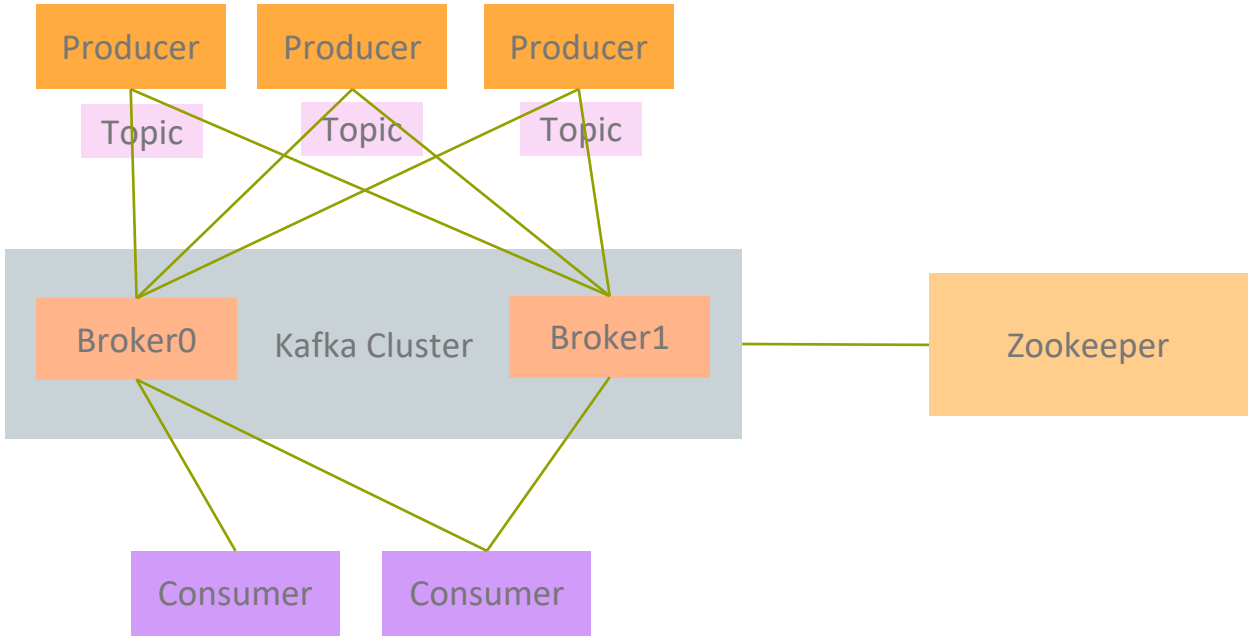
Producer
Application who can publish messages to a topic

Consumer
Application who subscribe to a topic and consume messages

Broker
Kafka cluster is a set of servers, each is called a broker

Zookeeper
Managing & coordinating Kafka brokers

Kafka Cluster



Kafka Topics

Agile Brains Consulting

CLOUD NATVZ

Kafka Topics

Topic is a category/feed name to which stream of messages are stored and published.

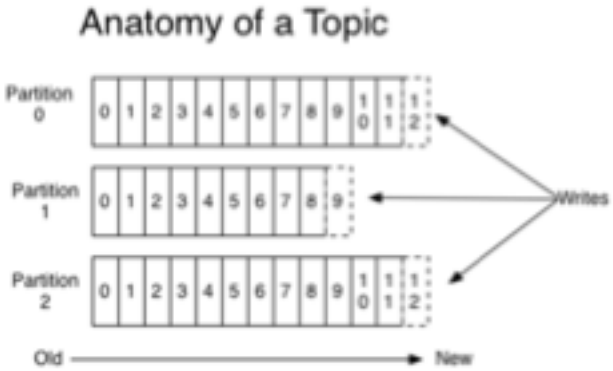
Producers publishes messages and consumer consumes messages.

Messages –

- Byte arrays that can store any object in any format.
- Messages are organized into topics.
- Each message has a key, a value, a timestamp

Topic Partition

- Topics are divided into number of partitions
- Each partition is an **ordered**, **immutable** sequence of messages that is continually appended to – a structured commit log
- Each message in a partition is assigned and identified by its unique **offset**
- Replication is implemented at the partition level
- Partitions allow to parallelize a topic by splitting data across multiple brokers
- Each specific message in a Kafka cluster can be uniquely identified by a tuple consisting of the (topic name, partition number, and offset in the partition) within the partition.

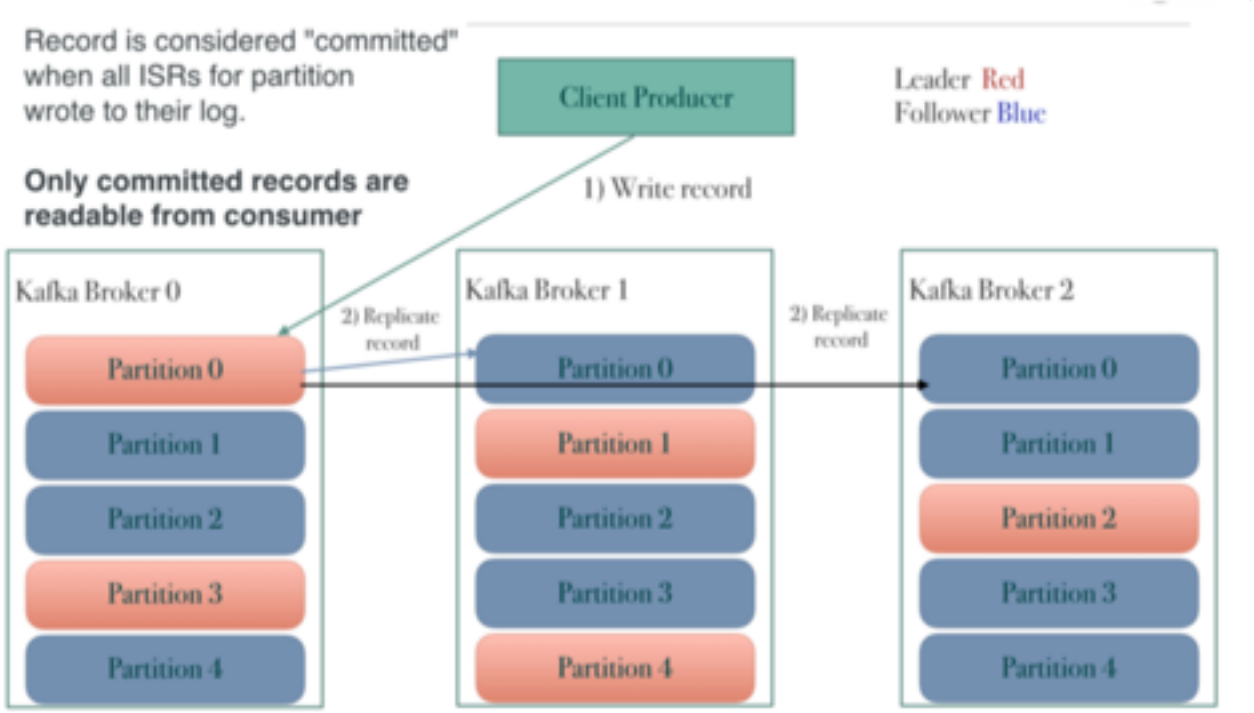


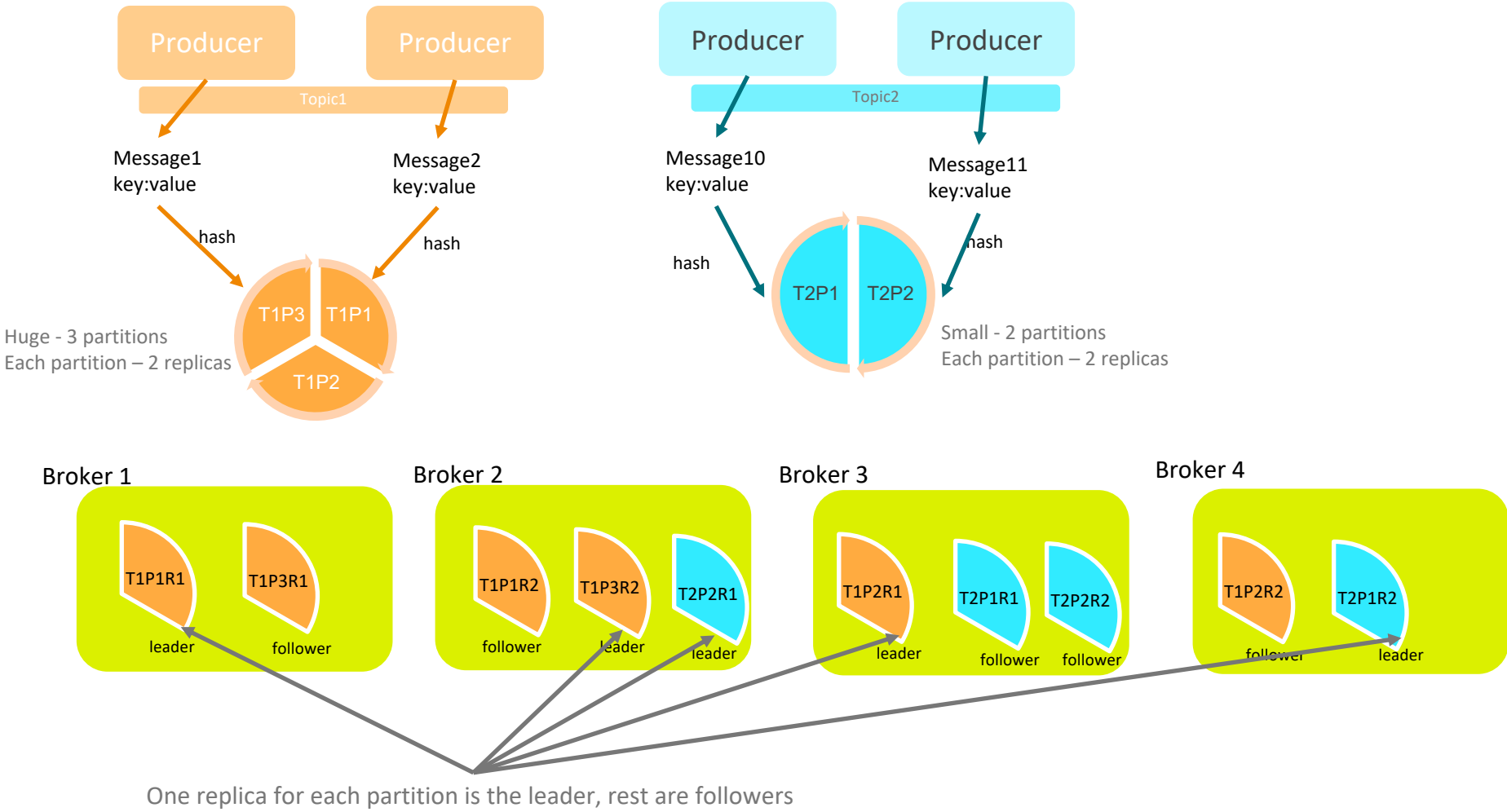
- Implements parallelism for Producers and Consumers
- Partition offers scalability
- Replica offers fault tolerance

Partition Leader & Followers

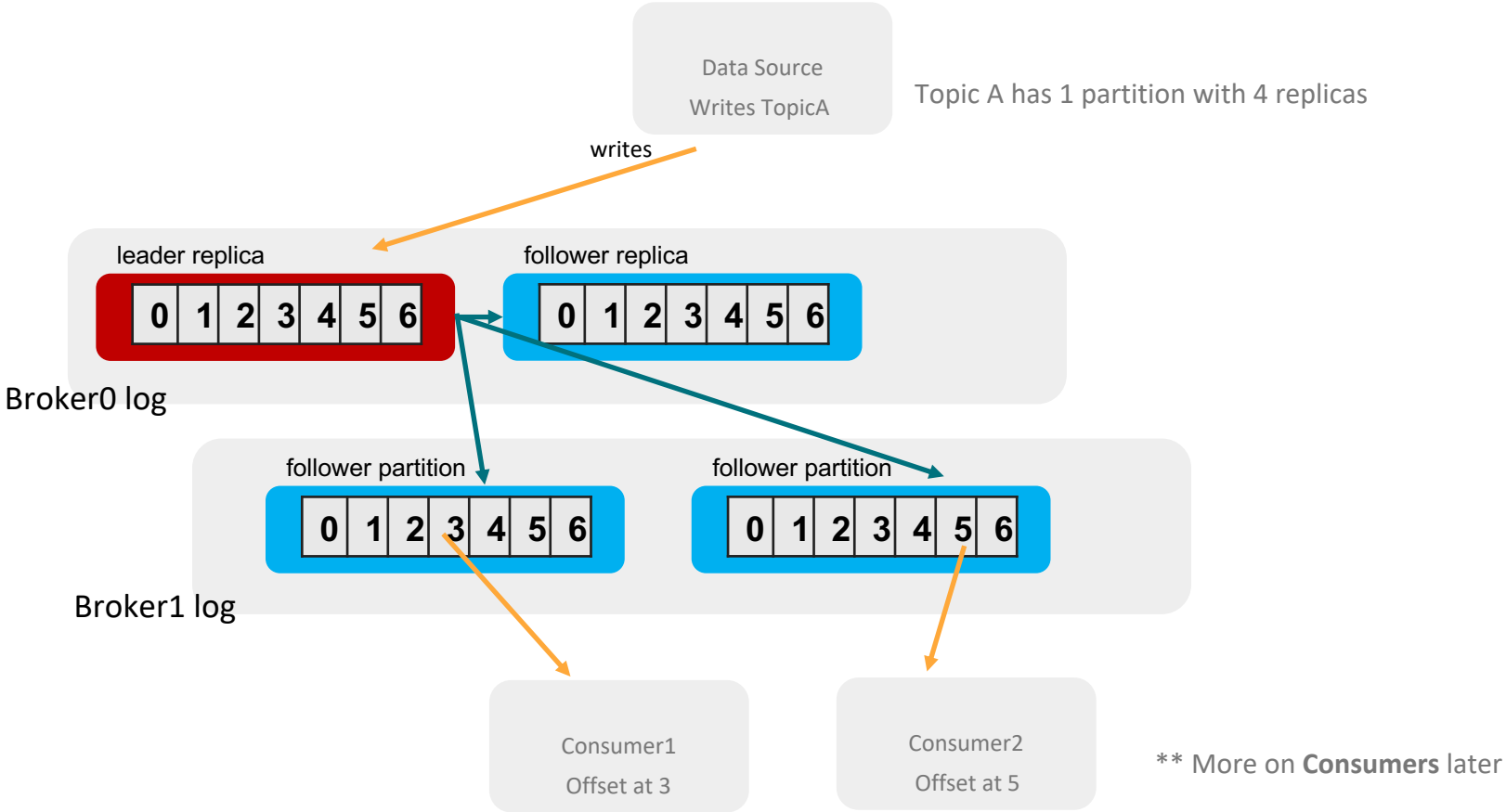
- Every partition has one leader replica and zero or more follower replicas
- Leader replica handles all read-write requests for the specific partition and the followers replicate the leader
- If the leader replica fails, one of the follower replica become the leader by ISR (in-sync replica) strategy
- When a producer publishes a message to a partition in a topic, it is forwarded to the leader replica
- The leader appends the message to its commit log and increments its message offset.
- Kafka only exposes a message to a consumer after it has been committed

Topic Partitions




Producer
Producer
Topic1
Message1
key:value
Message2
key:value
hash
hash
T1P3
T1P1
T1P2
Huge - 3 partitions
Each partition – 2 replicas
Producer
Producer
Topic2
Message10
key:value
Message11
key:value
hash
hash
T2P1
T2P2
Small - 2 partitions
Each partition – 2 replicas
Broker 1
T1P1R1
T1P3R1
leader
follower
Broker 2
T1P1R2
T1P3R2
T2P2R1
follower
leader
leader
Broker 3
T1P2R1
T2P1R1
T2P2R2
leader
follower
follower
Broker 4
T1P2R2
T2P1R2
follower
leader
One replica for each partition is the leader, rest are followers

Log Anatomy – Another way to understand Partition



Choose # of Partitions

- Topic partition is the unit of parallelism in Kafka
- Single partition's data to one consumer thread
- More partitions means higher throughput

- More partitions requires more open file handles – 2 per partition (1 for index, other for log segment)
- Partial unavailability – Say a broker dies which had "X" leader partitions, then all X partitions becomes unavailable at the same time. Total time taken to elect "X" new leaders = time of electing (1) * X
- May increase end-to-end latency – Kafka exposes message to consumer only after message is replicated in all in-sync partitions.

Choose # of partitions wisely

Choose # of Partitions

A rough formula for picking the number of partitions is based on **throughput**.

p = measured throughout that can be achieved on a single partition for production

c = measured throughout that can be achieved on a single partition for consumption

t = target throughput

You will need **at least *max(t/p, t/c)*** partitions.

Producer and Consumer throughputs

- Producer – Per-partition throughput for producer depends on configurations such as the batching size, compression codec, type of acknowledgement, replication factor, etc.
- Consumer – Application dependent, on how the messages is processed. So, you need to measure it.

Topics Configurations

Kafka allows configurations to be set per topic but users should first set a baseline or these in global configurations.

Num.partitions(default: 1) - how many partitions a topic is created with. Partitions can only be increased not decreased.

Log.retention.ms - how long Kafka will retain messages. This is the smallest of the time expiration options.

Log.retention.bytes - applied at the partition level this sets message expiration based on the size of the data retained. If you have specified a value for both log.retention.bytes and log.retention.ms (or another parameter for retention by time), messages may be removed when either criteria is met.

Message.max.bytes(default: 1MB) - maximum compressed size of a message that can be produced

Retention Policy

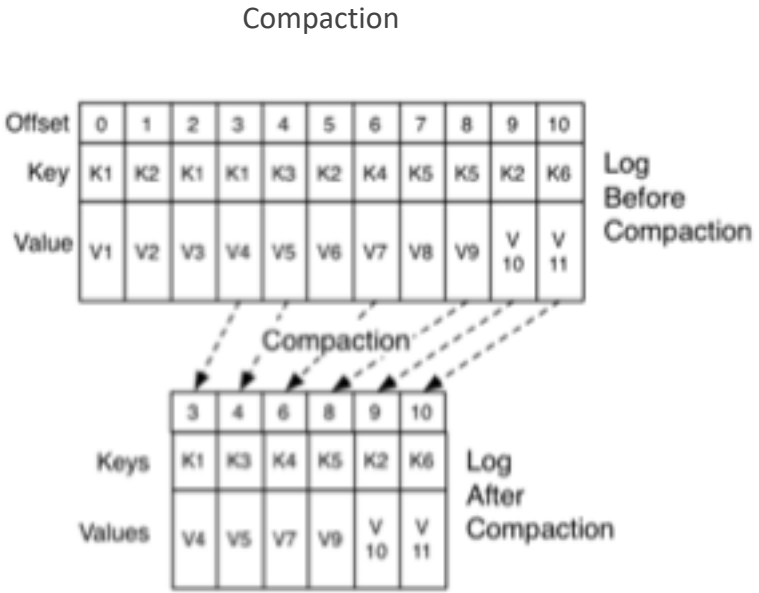
- Message sent to a Kafka Cluster is appended to the end of one of the logs
- The message remains in the topic for a configurable period of time OR a configurable size is reached
- The message stays even if it is consumed by one or more consumers
- After retention period or size is reached, the message will be discarded
- Default is 7 days.

Set config in broker to change: {offsets.retention.minutes}

You can also set retention time for a topic and log file

Retention Strategies

- Delete
  - Default cleanup policy, simply discard old segments when their retention time or size limit has been reached
- Compaction
  - Enables Log Compaction on a topic. Selectively remove records for each partition where there is a recent update with the same primary key
- Delete and Compact
  - Does both



Kafka Topic CLI

- Create a topic “test” with 1 partition and 1 replication factor

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

Messages

Agile Brains Consulting

CLOUD NATVZ

Kafka Messages

Examples - Log messages, Database records, etc.

Properties –

- Byte arrays that can store any object in any format.
- Messages are organized into topics.
- Each message has a key, a value, a timestamp
- Directing message to partition is done using message key. (Partition = hash(key) mod N), where N is number of partitions
- Message from same key will always be in the same partition
- If message has no key, Kafka will assign partition using round-robin

Message Schemas and Serialization

Messages are opaque byte arrays to Kafka itself, it is recommended that additional structure, or schema, be imposed on the message content so that it can be easily understood.

- JSON and XML are sufficient, but lack robust features and are inefficient.

Serialization – process of converting an object into stream of bytes and that bytes are used for transmission.

De-serialization – process of converting stream of bytes back to the object

Kafka offers **ser** and **deser** only for few data types – String, Long, Double, Integer and Bytes. For custom class of objects, write your custom serde

Why use Custom SerDe

Serialization – Before transmitting message from Producer to Kafka queue, let the producer know how to convert the message into byte array

DeSerialization – Provide ability to consumer to convert serialized message back to readable message.

Kafka offers **(De)Serialization** only for few data types – String, Long, Double, Integer and Bytes. For custom class of objects, use custom.

- Better for blob objects or unstructured text
- Efficient and less resource consumption
- Easy retrieval and usage by the consumer

Custom Kafka SerDe Example

Implement: org.apache.kafka.common.serialization.Serializer

Implement: org.apache.kafka.common.serialization.Deserializer

User.java - POJO

```
public class User {
  String name;
  int age;

// constructor
// getter and setter
}
```

Custom Kafka SerDe Example

CustomSer.java –
implements serializer

```
// Implement Serializer
public class CustomSer implements Serializer {

  @Override void configure(Map<String, ?>
     map, boolean b) {}

  @Override byte[] serialize(User user) {
      byte[] val;
      ObjectMapper mapper = new ObjectMapper();
      val = mapper.writeValueAsString(user).getBytes();
      return val;
  }
}
```

CustomDeser.java –
implements deserializer

```
// Implements deserializer
public class CustomDeser implements Deserializer {

 @Override
 void close() {}

 @Override
 void configure(Map<String, ?>
    arg0, boolean arg1) {}

 @Override
 User deserialize(byte[] user_bytes) {
   ObjectMapper mapper = new ObjectMapper();
   User user = mapper.readValue(user_bytes, User.class);
   return user;
 }
}
```

Apache Avro

Avro is an open source, language neutral data serialization api that was originally created for Hadoop.
Avro can serve as an exchange between programs written in any language.

- Avro stores the data definition in JSON format making it easy to interpret
- Data itself is stored in binary format making it **compact** and **efficient**.
- Schema Evolution, robust support for data schemas that change over time

Apache Avro

- Create an Avro schema according to your data.
- Read the schemas into your program.
- Serialize the data using the serialization API provided for Avro
- Producers sends the message
- Consumer receives and deserialize the data using deserialization API provided for Avro

Brokers and Kafka Cluster

Agile Brains Consulting

CLOUD NATVZ

Brokers

A Kafka cluster consists of one or more servers called Kafka Brokers, which are running Kafka.

Producers

- Receives messages from producers
- Assigns offsets to messages
- Commits them to storage on disk

Consumers

- Responding to fetch request for partitions
- Responds with confirmations that messages have been committed

Clusters

In a scalable system Kafka brokers are designed to be part of a cluster. One broker in a cluster will function as the cluster **controller**. It will be in charge of nominating and replacing partition leaders and replicas.

The Controller

- One broker in the cluster acts as controller
- Monitor the liveness of brokers
- Elect new leaders on broker failure
- Communicate new leaders to brokers
- The process of promoting a broker to be the active controller is called Kafka Controller Election
- When started, KafkaController emits Startup controller event. That starts controller election

Broker Configurations

Broker.id - Unique integer identifier that defines the broker.

Port - default port for Kafka listener is 9092. Custom port number below 1024 requires root access.

Zookeeper.connect - the location of the Zookeeper service used for storing broker metadata. The format for this parameter is port/path

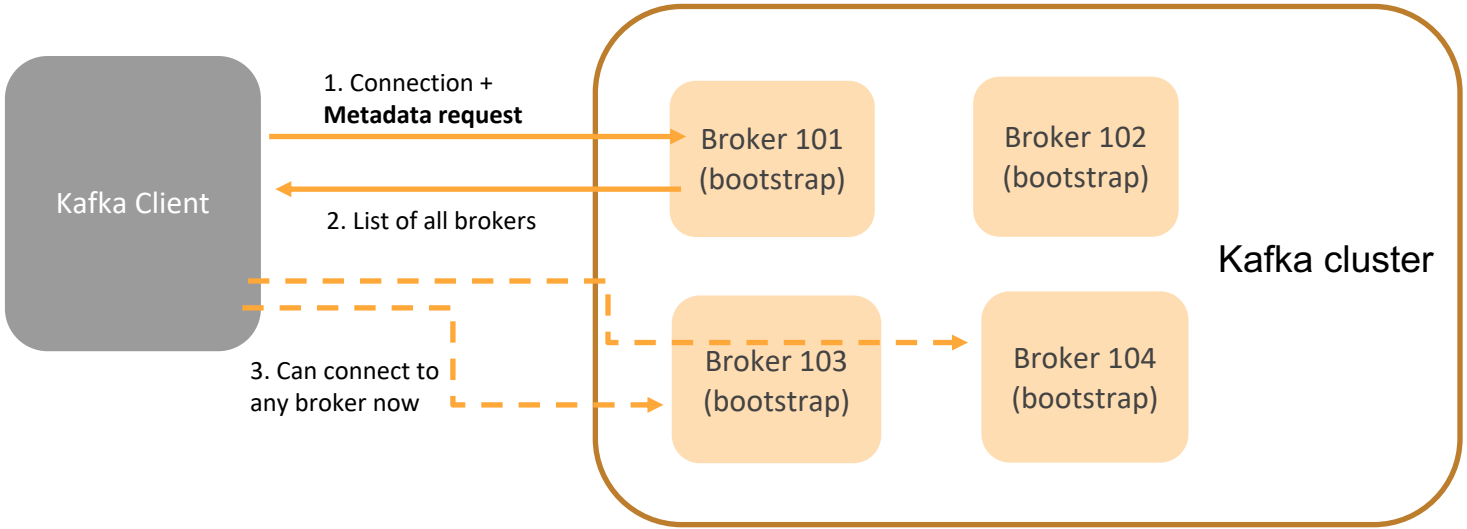
Log.dirs - the directory where Kafka stores all messages.

Num.recovery.threads.per.data.dir - The number of threads per data directory to be used for log recovery at startup and flushing at shutdown

Auto.create.topics.enable - specifies that broker should automatically create a topic when a producer or consumer start reading/writing from a topic or when a client request metadata for topic.

Kafka Broker Discovery

- Every Kafka broker is called “**bootstrap server**”
- You only need to connect to one broker, and you will get connected to the entire cluster
- Each broker knows about all other brokers, topics and partitions (just metadata)



Kafka Storage Internals

Agile Brains Consulting

CLOUD NATVZ

Segments

- Each partition is sub-divided into **segments**.
- Segment is a collection of messages of a partition
- Kafka doesn’t store all message in a partition to a single log file. Instead it splits them into chunks called segments
- When Kafka writes to a partition it writes to the **active segment**. Once the segment’s size limit is reached (default value: 1GB), a new segment file is created
- Each segment file is created with the offset of the first message as its file name.
- The base offset of a segment is an offset greater than offsets in previous segments and less than or equal to offsets in that segment.

Segment Advantages

- Parallelism
- Easy Purge – Kafka remove messages using the retention policy. It’s not easy to remove message from a file when producer is writing to it. Instead removing segments is easier

Segment Components

Each segment has three kind of files

- *.log file ---- to store actual message and metadata
- *.index file ---- Consumer seldom reads message at a particular offset. For this, it has to go to the log file to find the offset, which is an expensive task. Index file stores the offsets and physical position of the message in the log file
- *.timeindex file ---- Some time information so it can be used along with *.index file to quickly locate an offset

Index File

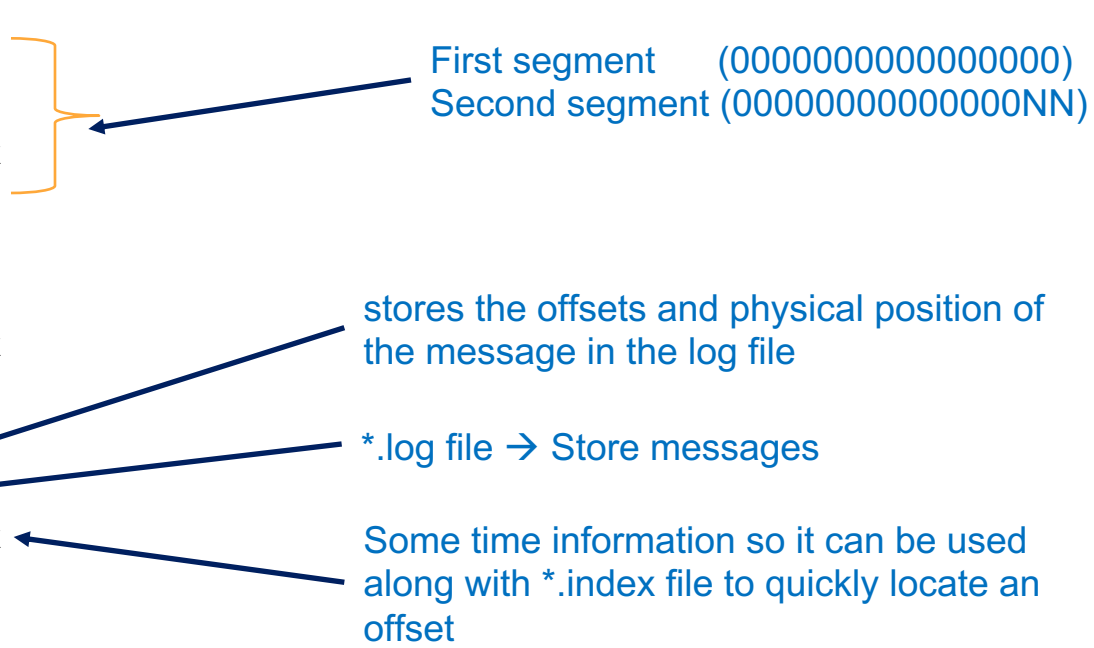
- Memory mapped
- Offset look up uses binary search to find the nearest offset less than or equal to the target offset.
- made up of 8 byte entries,
    - 4 bytes to store the offset relative to the base offset
    - 4 bytes to store the position.
- offset is relative to the base offset so that only 4 bytes is needed to store the offset.

Topic Log Directory

Create a topic with P partitions

```
./kafka-topics.sh --create --zookeeper <ZK_URL:PORT> --replication-factor <R> --partitions <P=3> --topic <T_NAME>
```

```
<T_Name>-0
|-- 00000000000000000000.index
|-- 00000000000000000000.log
|-- 00000000000000000000.timeindex
`-- leader-epoch-checkpoint
<T_Name>-1
|-- 00000000000000000000.index
|-- 00000000000000000000.log
|-- 00000000000000000000.timeindex
`-- leader-epoch-checkpoint
<T_Name>-2
|-- 00000000000000000000.index
|-- 00000000000000000000.log
|-- 00000000000000000000.timeindex
`-- leader-epoch-checkpoint
```



Topic Log File

Inside log file, a line appears like below:

```
@^@^BÂ°Â£Ã¦Ãƒ^@^K^XÃ¿Ã¿Ã¿Ã¿Ã¿^@^@^@^A"^@^@^A^VHello World^@
```

You can use kafka.tools.DumplofSegments to understand the binary expression:

```
./kafka-run-class.sh kafka.tools.DumpLogSegments --deep-iteration --print-
data-log --files logs\<log_file_path>.log
```

It should output something like below:

```
offset: 0 position: 0 CreateTime: 1533443377944 isvalid: true keysize: -1
valuesize: 11 producerId: -1 headerKeys: [] payload: Hello World
```

Zero Copy and Sequential I/O

- Kafka use filesystem for storing and caching messages (not RAM)
- Low latency is achieved through **Sequential I/O and Zero Copy Principle**

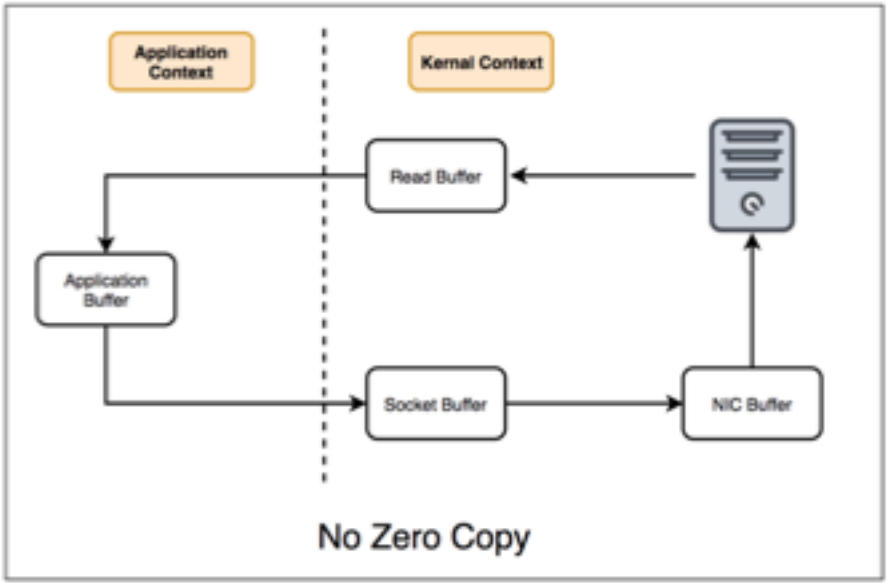
Sequential I/O

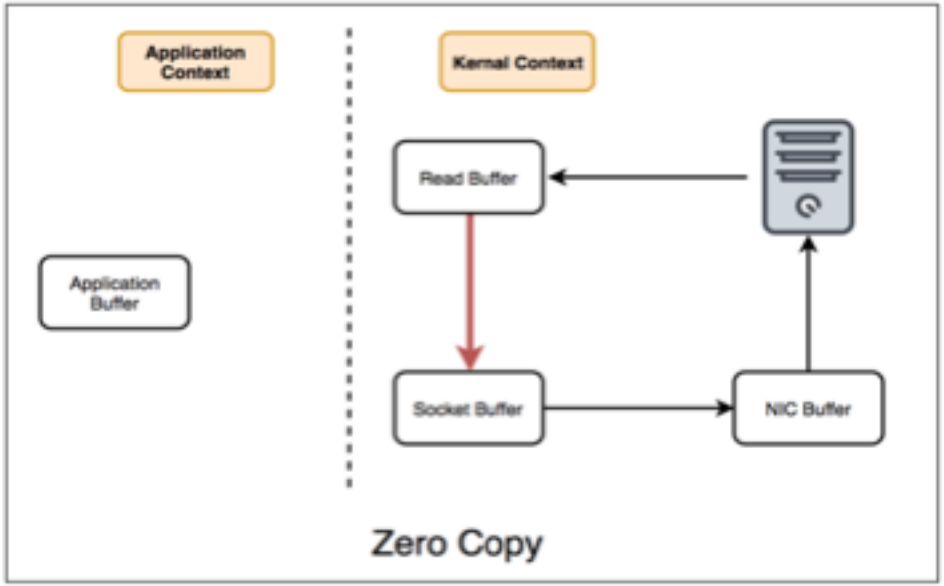
- avoids disk seek time by maintaining offsets and ordered messages

Zero copy –

- data transfer beyond the user-boundary kernel consumes CPU cycles
- request kernel to move the data to sockets rather than moving it via the application

Zero Copy and Sequential I/O





Producers and Consumers

Agile Brains Consulting

CLOUD NATVZ

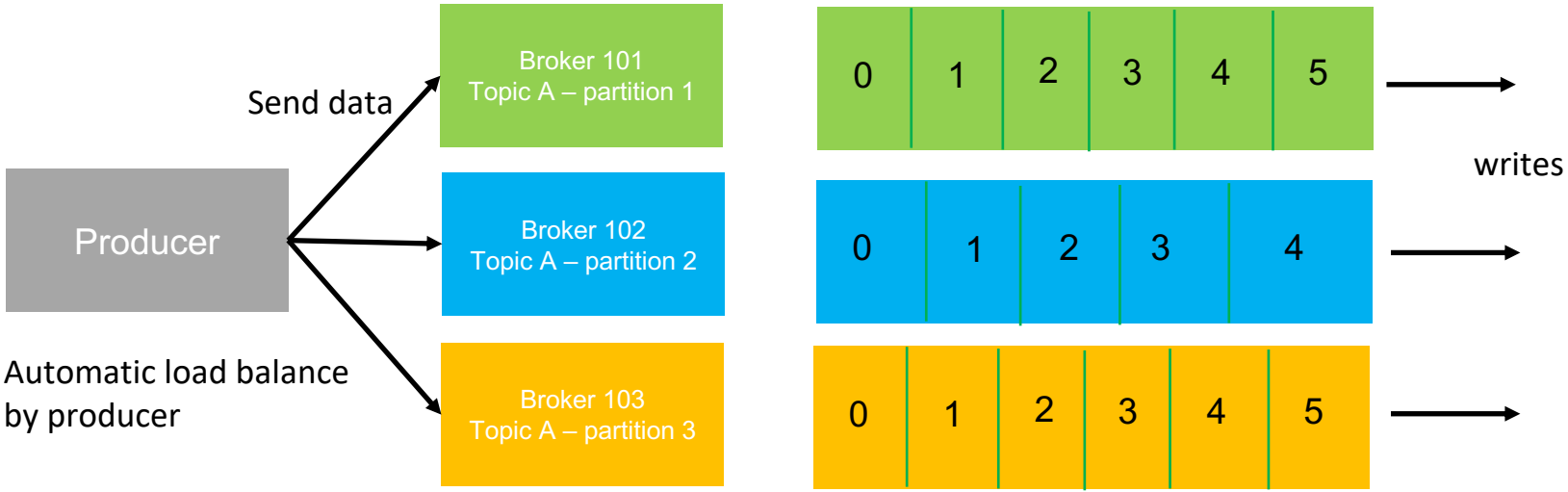
Producers and Consumers

Whether you use Kafka as a queue, message bus, or data storage platform, you will always use Kafka by writing a producer that writes data to Kafka, a consumer that reads data from Kafka, or an application that serves both roles.

Kafka Producers

- Write data to topics (which is made of partitions)
- Automatically know to which broker and partition to write to
- In case broker fails, producers will automatically recover
- Pick partitions and configure their consistency/durability
- Append records to end of topic log
- Producers can write to multiple topics
- Can use the API or CLI/Console

Kafka Producers



Producer Acknowledgement

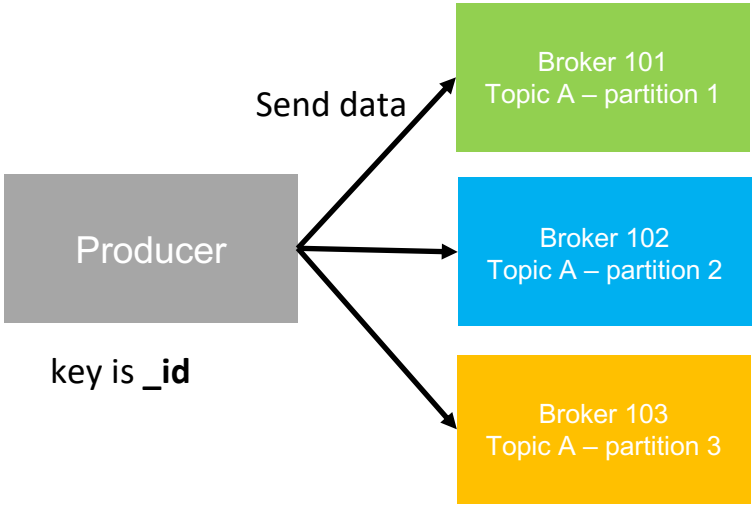
- Producers can choose to receive acknowledgement of data writes
  - acks=0; won’t wait (possible data loss)
  - acks=1; will wait for leader acknowledgement (Default) (Limited data loss)
  - acks=all; Leader + replicas acknowledgement (No data loss)

Producer Message Keys

- Producers can choose to send a key with the message (String, Number, etc.)
  - If key=null, data is sent round robin across partitions
  - If key is sent, then all messages for that key will always go to the same partition.
- When key is sent, Kafka use partitioner
  - Default is hash partitioner
  - Can use custom partitioner

  partition number = hash(key) Mod p; where p is total # of partitions

Kafka Producers



Example:

key _id123 will always be in partition 1
key _id456 will always be in partition 1

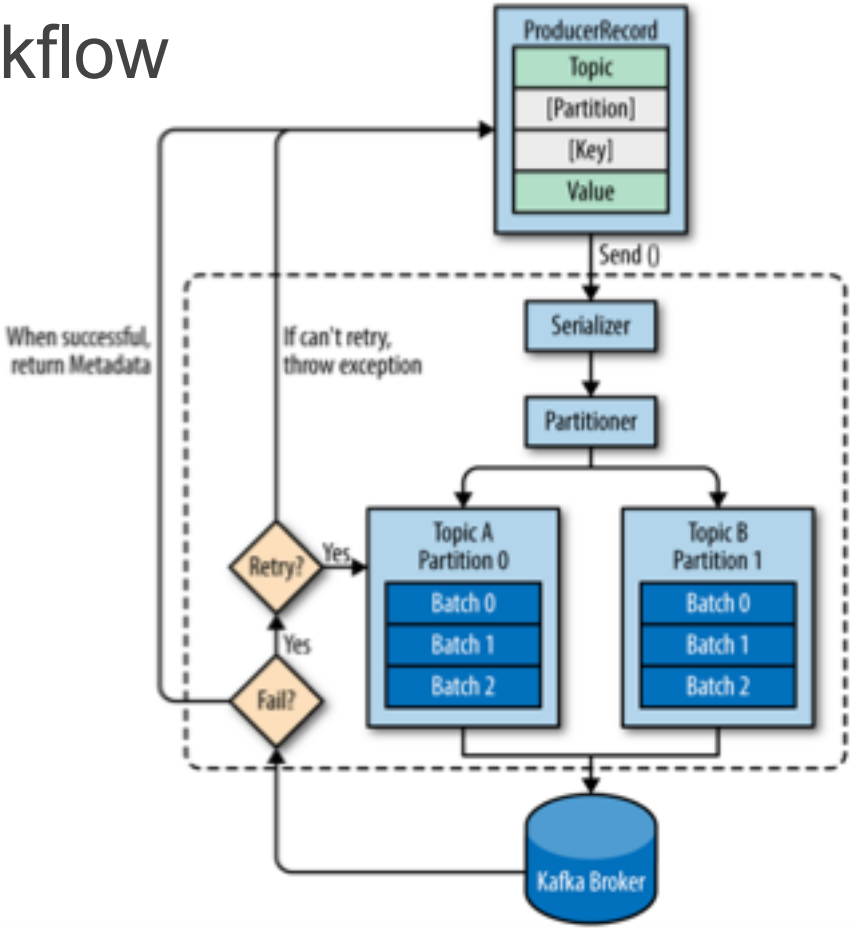
key _id675 will always be in partition 2

key _id897 will always be in partition 3

Benefits:

- Less Shuffle
- Easy Sorting and Aggregation
- Helps in retention policy

Producer Workflow



Partitioning and Order

Example: Because your producer application determines order you could have all the events of a certain ‘student_id’ go to the same partition.

If you needed something a little more random you could combine student id and the timestamp and hash them to ensure data is evenly partitioned.

If order within a partition is not needed, a ‘Round Robin’ partition strategy can be used, so Records get evenly distributed across partitions

KafkaProducer API

In order to send messages to a topic, KafkaProducer class provides send method.

```
producer.send(new ProducerRecord<byte[],byte[]>(topic,partition, key1, value1) , callback);
// ProducerRecord – producer manages a buffer of records waiting to be sent
// callback – when the record has been acknowledged by the server, a user-supplied callback to execute

public void send(KeyedMessaget<k,v> message)
// send single data

public void send(List<KeyedMessage<k,v>>messages)
// send multiple data
```

Producer Settings

“serializer.class”
defines what Serializer to use when preparing the message for transmission to the Broker.

"partitioner.class"
defines what class to use to determine which Partition in the Topic the message is to be sent to. This is optional, but for any non-trivial implementation you are going to want to implement a partitioning scheme.

”acks"
Control acknowledgement protocols

”batch.size"
Buffer size

Producer Settings

Q: Which broker should my producer use, and do I need to manually switch the broker to balance the load?

A: No need to worry about figuring out which Broker is the leader for the topic (and partition), the Producer knows how to connect to the Broker and ask for the metadata then connect to the correct Broker.

Producer Protocols

Fire-and-Forget

We send a message to the server and don’t really care if it arrives successfully or not.

Synchronous Send

We send a message, the send method returns a Future object, and we use get to wait on the future and see if the send was successful or not.

Asynchronous Send

We call the send method with a callback function, which gets triggered when it receives a response from the Kafka broker.

Kafka Producer Console (CLI)

kafka-console-producer is a consumer command line to write data to a Kafka topic

- Start a consumer

  ```
  bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic test
  ```

- The above gives a console to type|send messages

Kafka Producer (Python)

*Need python, pip, python-kafka

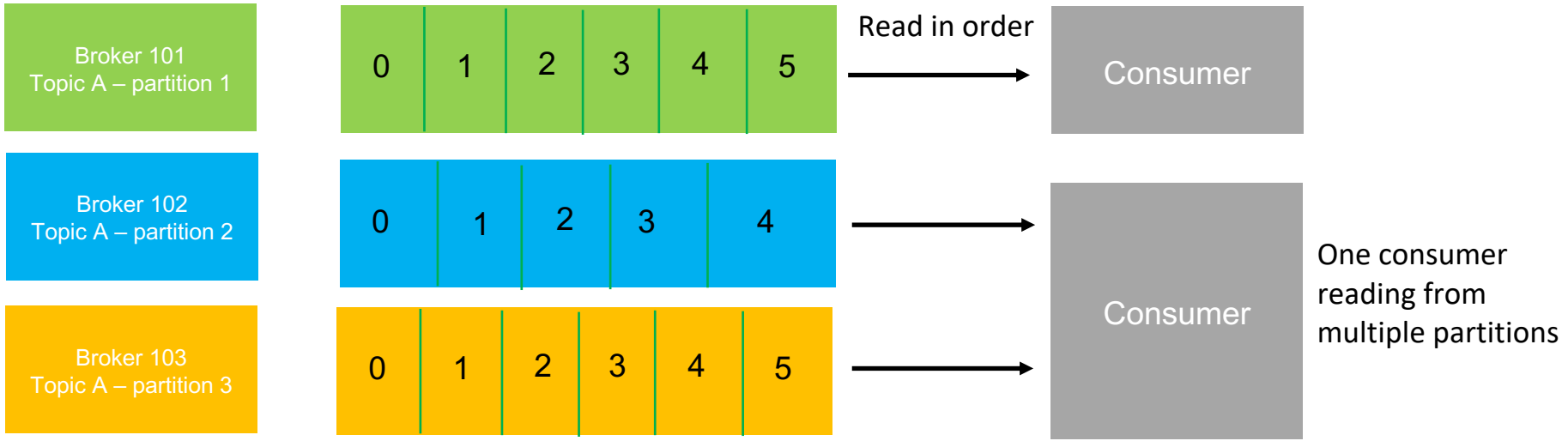
*kafka_test.py

```
from kafka import KafkaProducer
import time
partition=3


#create topic
topic = 'test'
producer = KafkaProducer(bootstrap_servers='localhost:9092', value_serializer=str.encode)

#produce logging to topic
with open("logs/test-logs.txt", mode="r") as log:
    for line in log:
        print(line)
        producer.send(topic, value=line)
        time.sleep(3)
```

Consumer

- Read from a kafka topic (identified by name)
- Consumers know which broker to read from
- In case of broker failures, consumers know how to recover
- Data is read in order **within each partition**
- Read kafka at their own pace
- Can go on and offline with no issues
- Consumers can read from multiple topics and same topic can be read by multiple consumer

Consumer



* Order is guaranteed in a partition, not at a topic level

Kafka Consumer Console (CLI)

kafka-console-consumer is a consumer command line to read data from a Kafka topic and write it to standard output.

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
```

Kafka Consumer (Python)

```
1  from kafka import KafkaConsumer
2  import pprint
3  from log_parser import LogRow
4
5
6  kafka_topic = 'test'
7  group_id = 'log_consumer_group'
8
9  #create Kafka consumer
10
11 consumer = KafkaConsumer(kafka_topic)
12 lr = LogRow()
13
14 #loop through
15
16 for msg in consumer:
17     msg = msg.value.decode('ascii')
18     msg= lr.parseRow(msg)
19     pp = pprint.PrettyPrinter(indent=4)
20     pp.pprint(msg)
```

Lab 2: Console Consumer and Producer (single broker)

Objective:

Create topics and run some console producers and consumers on Apache Kafka, and get familiar with the Kafka command line interface.

https://github.com/shekhar2010us/kafka_t/blob/master/labs/02_single_broker_producer_consumer_console.md

Consumer Groups and Consumer Offsets

Agile Brains Consulting

CLOUD NATVZ

Consumer Group

Consumer group – Group of related consumer doing a specific task

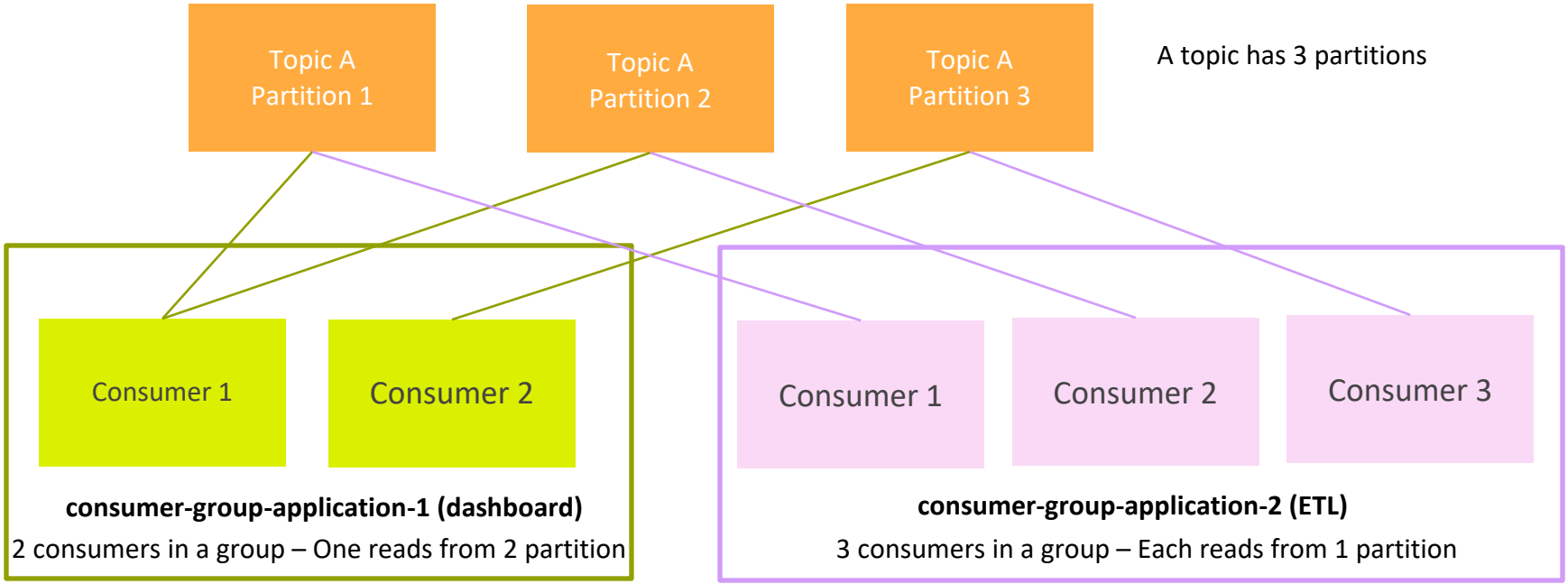
Need – It is a means to scale system and have single application reading data in parallel

- A group maintains its offset per topic partition.
- All consumers (collectively) in a group ensures reading all messages in a topic
- different consumer groups can read from different locations in a partition

Consumer Group Know and How

- Consumers read data in consumer groups
- Approach towards multi-threaded or multi-machine consumption from Kafka topics
- A group has a unique id and name to identify them from other consumer groups.
- A record gets delivered to only one consumer in a consumer group
- Each consumer within a group reads from **exclusive partitions**
- Consumers in the same consumer group **do not share partitions**
- If you have more consumers than partitions, some consumers will be inactive
- If you have less consumers than partitions, some consumers will read from more than one partition

Consumer Groups



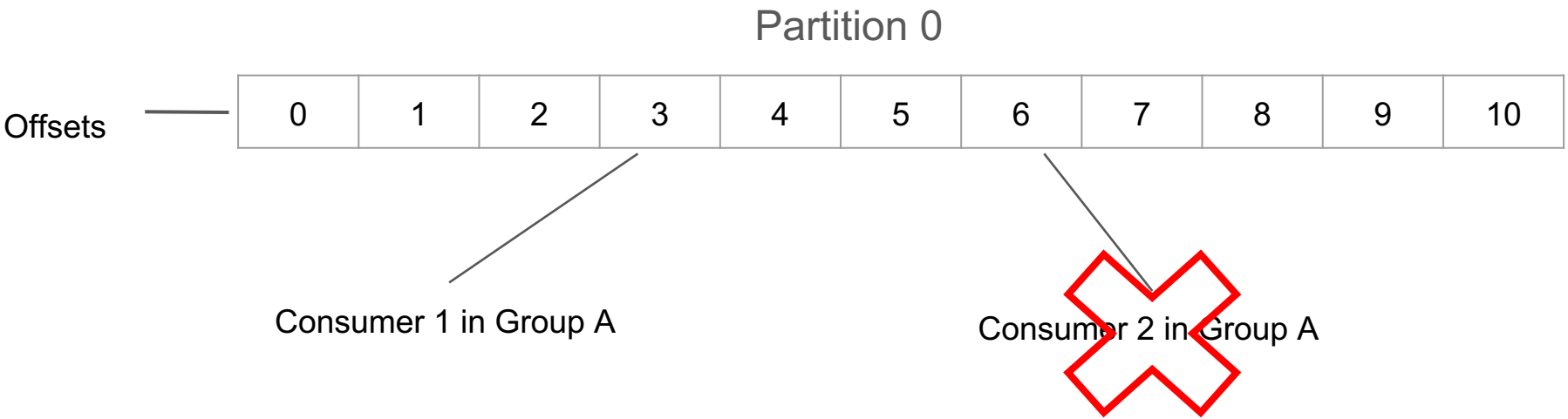
NO TWO CONSUMERS IN A GROUP READS FROM THE SAME PARTITION

Consumer Groups

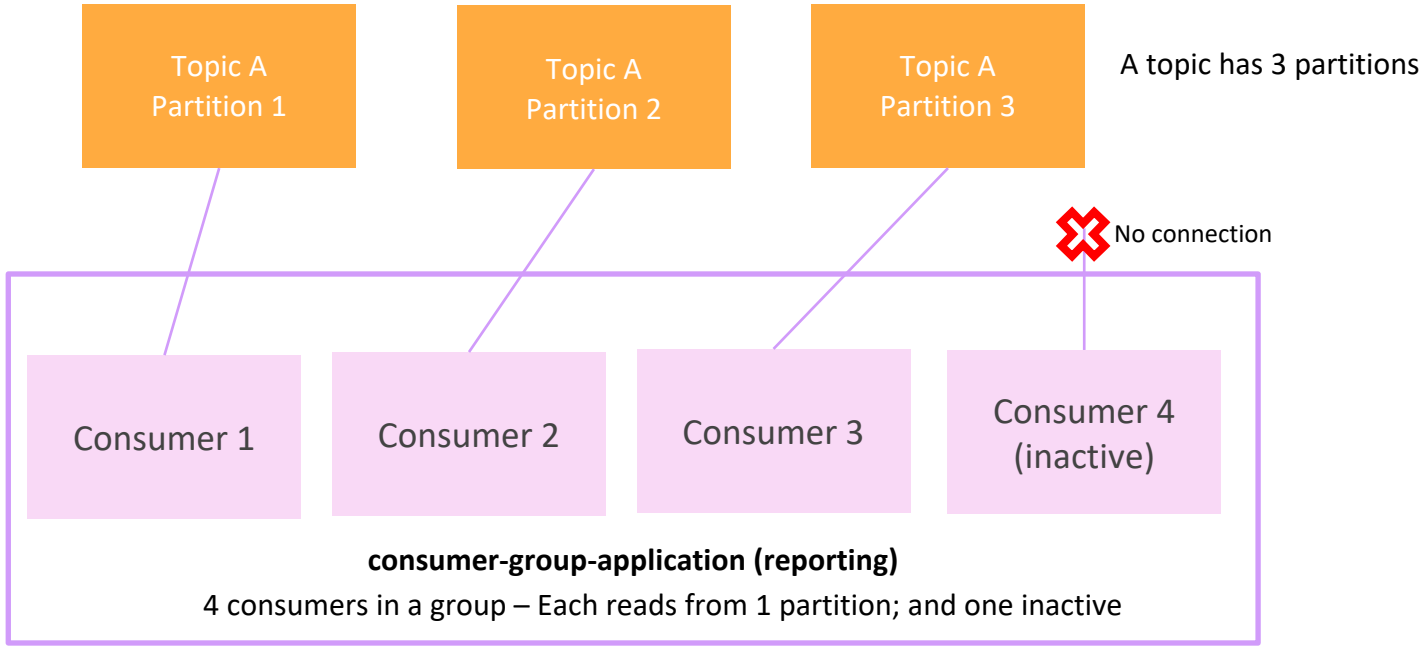
Consumers in the same consumer group **do not share partitions**

Means no two consumers from the same group reads from the same partition

WHY ?



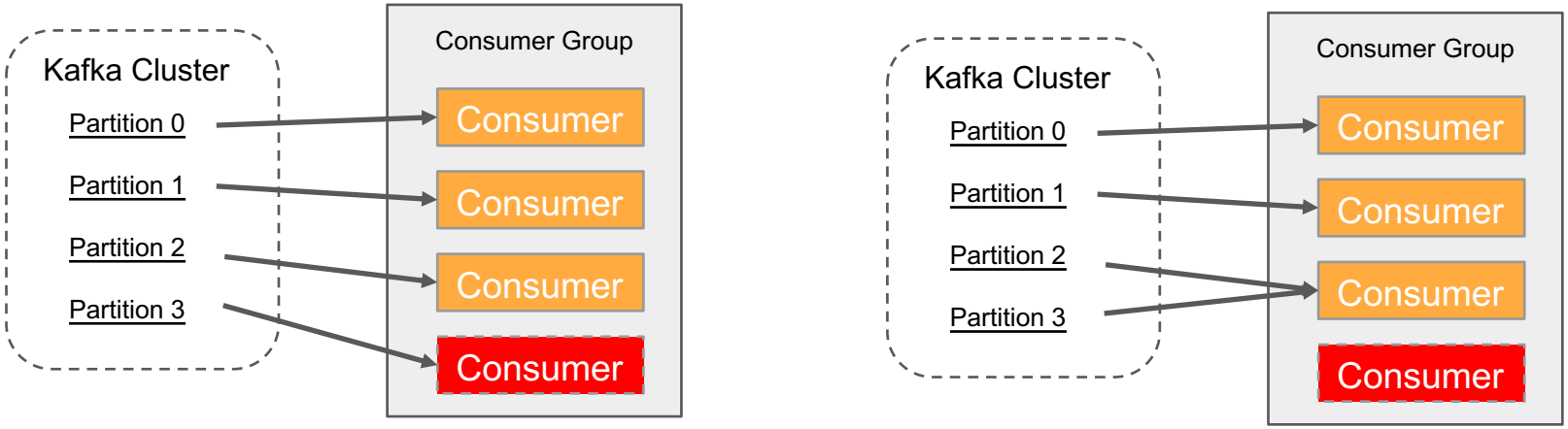
Too Many Consumers in a Group



Sometimes you want this in a critical path, so if one consumer dies, you have another one ready to take the job
If not, you are wasting resources

Consumer Group

What happens when a consumer goes down in a Consumer Group?



The partition goes down, and the partition is reassigned, but more on this later!

Kafka Consumer Offsets

Consumers:

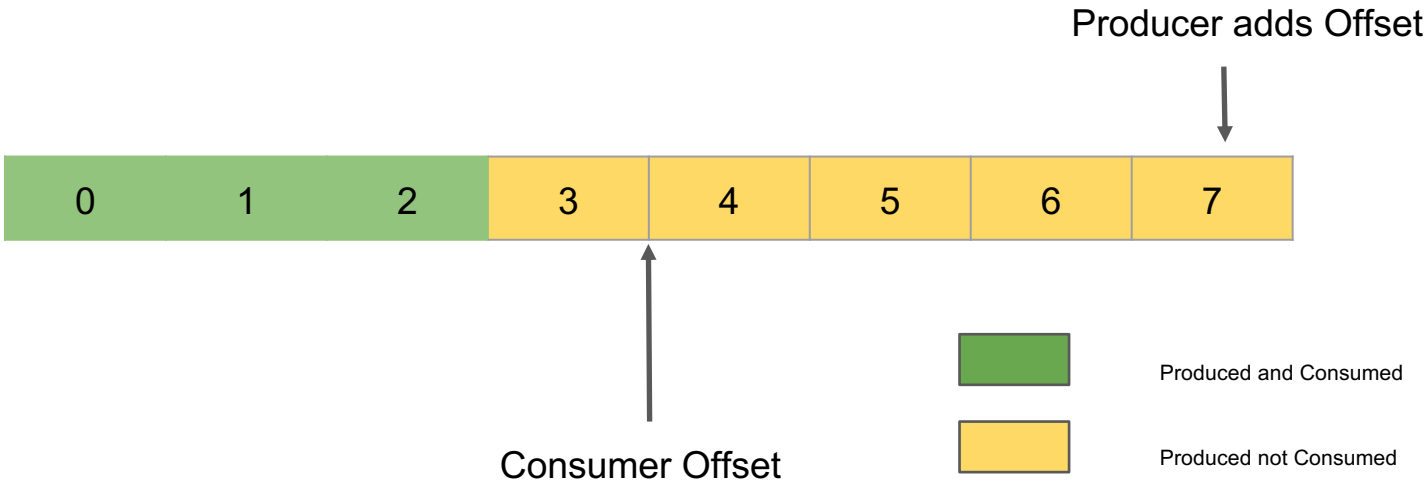
- Can read data from beginning of a topic
- Can read data from where it has left
- Once dead, alive again and read from where it left
- Another consumer from the group read data from where it left
- Can read data from specified position

How? Through **Offset**

Kafka Consumer Offsets

- Integer number used to maintain the current position of a consumer
- Pointer to the last record that has been sent to a consumer
- Consumer read from a specific offset allowing the consumers to join the cluster and topic at any point.
- This makes a consumer more resilient allowing it to pick back up where it left off.
- Each message in a given partition has a unique offset and position in the partition.

Kafka Consumer Offsets

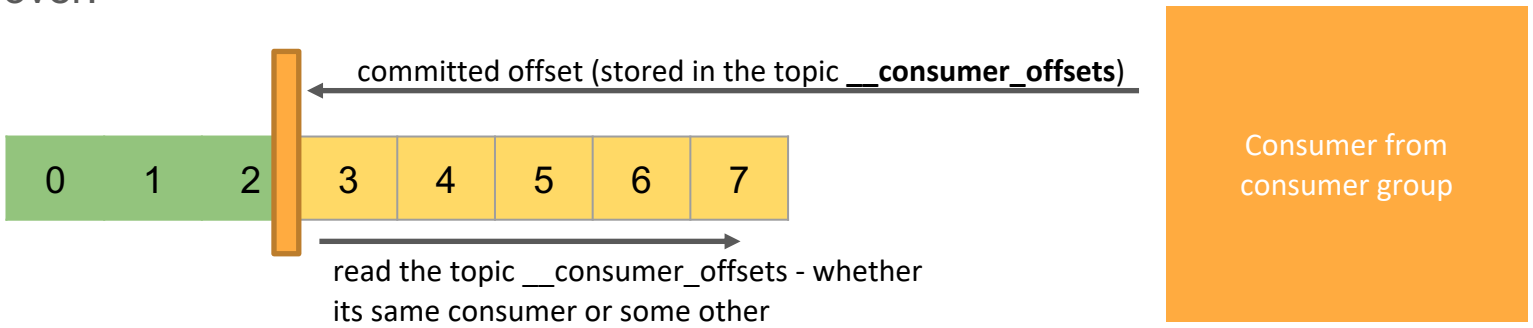


Kafka Consumer Offsets

When a consumer has processed data, it should commit offsets.

Kafka stores offset data in a topic called "**__consumer_offsets**". These topics use log compaction, which means they only save the most recent value per key.

If consumer process dies, it will be able to start up and start reading where it left off based on offset stored in "**__consumer_offsets**" or another consumer in the consumer group can take over.



Delivery Semantics for Consumers

Consumers choose when to commit offsets

Three delivery semantics:

- **At most once** (not preferred)
  - Offsets committed as soon as message is received
  - If processing goes wrong, message gets lost (won't be read again)
- **At least once** (preferred)
  - Offsets committed after message is processed
  - If processing goes wrong, message will be read again
  - Since same message can be read & processed twice, make sure system is idempotent – processing a message twice shouldn't impact the system
- **Exactly once**
  - Only be achieved for Kafka => Kafka workflows using Kafka Stream API

Ordered Messages at the Topic Level ?

Q: What is the way to get ordered messages at a topic level?

A: Messages are ordered at the partition level not at topic level

- Create just one partition
- Use same key for all messages – so it goes to just one partition
- Order messages in consumer

Q: What if message is not a KeyedMessage?

A: No key means no HashPartition, Kafka will assign partitions in round-robin strategy

Kafka Core APIs

Agile Brains Consulting

CLOUD NATVZ

Kafka Core APIs

- **Producer API**
    - Contains a set of API’s which allows us to publish a stream of data to one or more of the named/categorized Kafka topics in the cluster.
- **Consumer API**
    - Contains a set of API’s which allows us to consume a stream of data
- **Streams API**
    - Contains relevant API’s which acts on the stream of data. They can process this stream data and can transform it from existing form to a designated form according to your use case demands. These are relatively new API's as against existing producer and consumer API’s.
- **Connector API**
    - API’s which allows Kafka to be extensible. It contains methods which can be used to build Kafka connectors for the inputting and outputting of data into Kafka.

Kafka Producer API

https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html

Configuration Property

- bootstrap.servers – to connect (default: -)
- acks – wait for leader acknowledgement (default:1)
- retries – # of retries if fails (default:2147483647)
- batch.size – buffer of unsent records (default: 16384 bytes)
- linger.ms – microbatch delay (default: 0 ms)
- buffer.memory – buffer memory (default: 33554432 bytes)
- key.serializer – serializer class for key (default: -)
- value.serializer – serializer class for value (default: -)

Functions

- **send** – send(ProducerRecord<k,v>, Callback)
- **partitionsFor** – partitionsFor(topic) – get partition metadata for the given topic
- **metrics** – metrics() – get full set of internals metrics maintained by the producer
- **flush** – flush() – immediately send all buffered records
- **close** – close() – close the producer object

Kafka Consumer API

https://kafka.apache.org/10/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html

Configuration Property

- bootstrap.servers – to connect (default: -)
- group.id – group name (default: -)
- enable.auto.commit – commit offset (default: true)
- auto.commit.interval.ms – time for commit offset (default: 5000 ms)
- key.serializer – serializer class for key (default: -)
- value.serializer – serializer class for value (default: -)

Functions

- **seek** – seek(<TopicPartition>, offset) – seek to offset
- **subscribe** – subscribe(List<String> topics)
- **unsubscribe** – unsubscribe()
- **partitionsFor** – partitionsFor(topic) – get partition metadata for the given topic
- **metrics** – metrics() – get full kept by the consumer
- **pause** – pause(<TopicPartition>) – suspend fetching from given partition
- **position** – position(TopicPartition) – offset of next record
- **close** – close() – close the producer object

Kafka Connect API

More on this later !!

Kafka Stream API

More on this later !!

Kafka Internals

Agile Brains Consulting

CLOUD NATVZ

Leaders

- Clients only produce and consume from partition leaders
- Published messages are forwarded to leader
- Leader appends message and increments its message offset
- Followers replicate leaders

In Sync Replicas (ISR)

- A broker which has fully caught up to partition it is following.
- ISR can not be behind on the latest message for a given partition.
- Partition leaders are responsible for keeping track of which broker is an ISR and which is not.
- State data is stored in zookeeper
- Every partition stores a list of in sync replicas

Clean Leader Election

For a broker to be promoted to a leader it must be an ISR. The idea of a clean leader election is necessary for Kafka's durability and availability guarantees. Users can change this setting to favor **availability over consistency**.

If we elected a leader with stale data it would cause the cluster to lose messages. Not only would we lose messages, but we could have conflicts in the consumers since the lost messages' offsets will be taken by newer messages.

Clean Leader Election

This conflict might still be a possibility even with clean leader election (hint: it’s not, but more on that later). An in-sync replica might not have the latest copy. Since replication is done asynchronously, it is impossible to guarantee that a follower is up to the very latest message.

Downside – of a scalable solution

Clean Leader Election

We can adjust our definition of ISR settings to make this scenario less likely.

A broker is considered ISR if:

replica.lag.time.max.ms

It has fetched messages from the partition leader in the last X seconds. It is not enough to fetch any messages—the fetch request must have requested all messages up to the leader log's end offset. This ensures that it is as in-sync as possible.

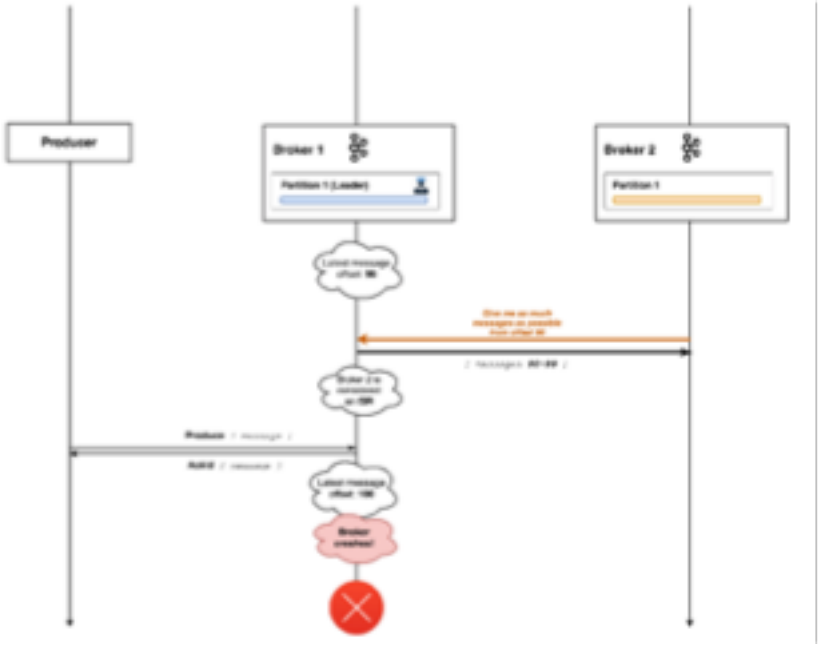
zookeeper.session.timeout.ms

It has sent a heartbeat to Zookeeper in the last X seconds.

ISR The perfect Storm

So there is a sad path scenario in which the leader dies and in-sync replica is elected as its replacement and we lose a small amount of data.

Example: A leader saves some new records right after it responds to a follower's fetch request, there will be a time frame in which those new messages will not have been propagated to the follower yet. If the leader happens to crash after acknowledging the new message, the replica would still be considered in-sync (perhaps X seconds have not passed yet) and would be elected as the new leader.



ISR The perfect Storm

If you have an application that requires 100% durability you can in theory use the configuration acks=all. This setting forces the leader to acknowledges messages once all the ISRs have successfully replicated the messages. This will put some **serious restrictions on the clusters throughput**.

Kafka Guarantess (re-iterates)

- Messages are appended to a topic-partition in the order they are sent
- Consumers read messages in the order stored in a topic-partition
- With a replication factor of N, producers and consumers can tolerate up to N-1 brokers being down
- This is why a replication factor of 3 is good
    - Allows for one broker to be taken down for maintenance
    - Allows for another broker to be taken down unexpectedly
- As long as the number of partitions remains constant for a topic (no new partitions), the same key goes to the same partition

Add Partition to Kafka Topic

```
## create a topic “my-topic” with 2 partitions
./bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic my-topic --replication-factor 1 --partitions 2

## change partitions to 3
./bin/kafka-topics.sh --alter --zookeeper localhost:2181 --topic my-topic --partitions 3

## describe the topic
./bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-topic
```

Partitions cannot be deleted !!

Adding more partitions may be un-desired !!

Add Partition to Kafka Topic

Old Partitions=m, new partitions=n; replicas=r

Re-partition will trigger a **rebalance** (not at the message level, but at the consumer level)

- Kafka will NOT shuffle existing partitions
- Updates the zookeeper path "/admin/reassign_partitions" with the list of topic partitions and the list of their new assigned replicas
- Will create ‘r’ replicas for (n-m partitions)
- Leader selection of the new partition
- Producers and consumers will get notified
- Producers will start sending messages to n partitions
- Consumers will be re-distributed for n partitions now

- **Bottleneck: If you worry about same key strictly be in the same partition, re-partitioning will violate that**

Zookeeper

Agile Brains Consulting

CLOUD NATVZ

ZooKeeper

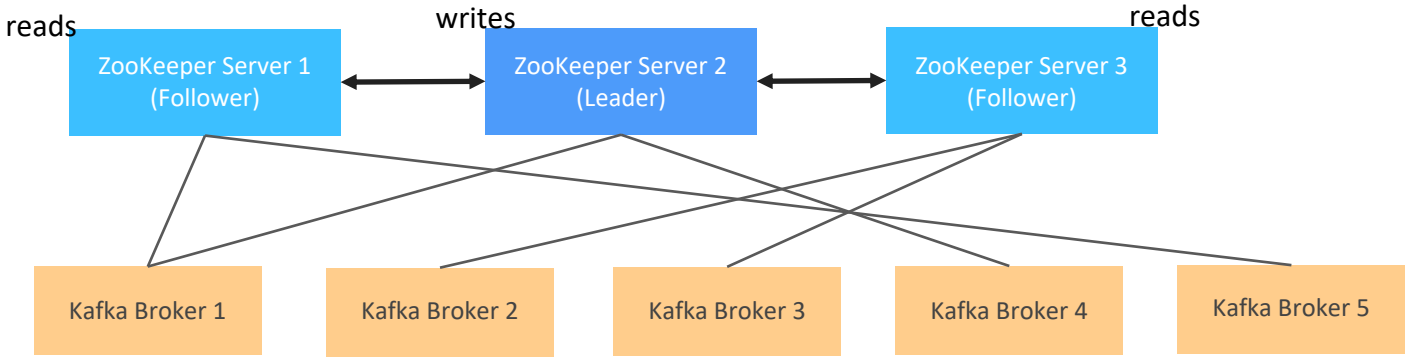
Cluster management tool that Kafka uses to store metadata about the cluster. When a broker process starts, it registers itself with Zookeeper with a Unique ID.

- Manages brokers (keep a list of brokers)
- Broker Leadership election
- Service discovery
- Topic Partitions
- Notify Kafka when there are changes to cluster
    - New brokers
    - Brokers died
    - New topics
    - Topics removed
- Cluster configuration information

Kafka cannot work without zookeeper !!

ZooKeeper

- By design, operates with an odd number only (1,3,5,7 servers)
  - Bcoz ZK works on quorum and majority
- **zk** has a leader (handle writes) and followers (handle reads)
- **zk** does not store consumer offsets with Kafka > v0.10

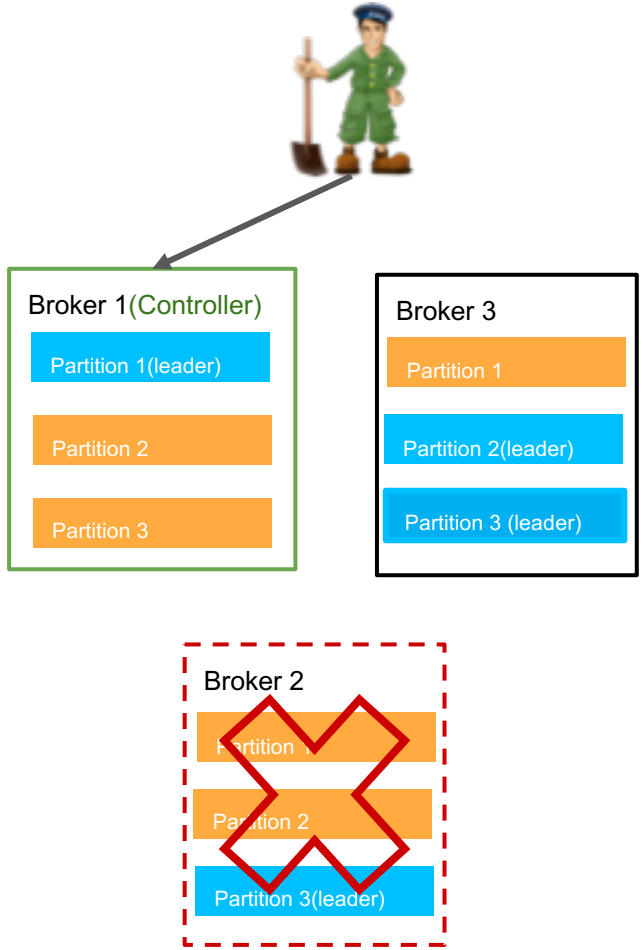


Doesn’t matter which broker connected to which zk server, just it has to connect to any one

Zookeeper

Example: Broker 2 goes offline

- Ephemeral Node is removed from Zookeeper
- Metadata associated with it is not deleted
- The Controller gets notified
- The Controller elects a new leader for partition 3
- Kafka components will be notified that the broker list has changed
- Broker 3 is informed that they are new leader for partition 3
- Broker 1 starts following for Partition 3 updates.

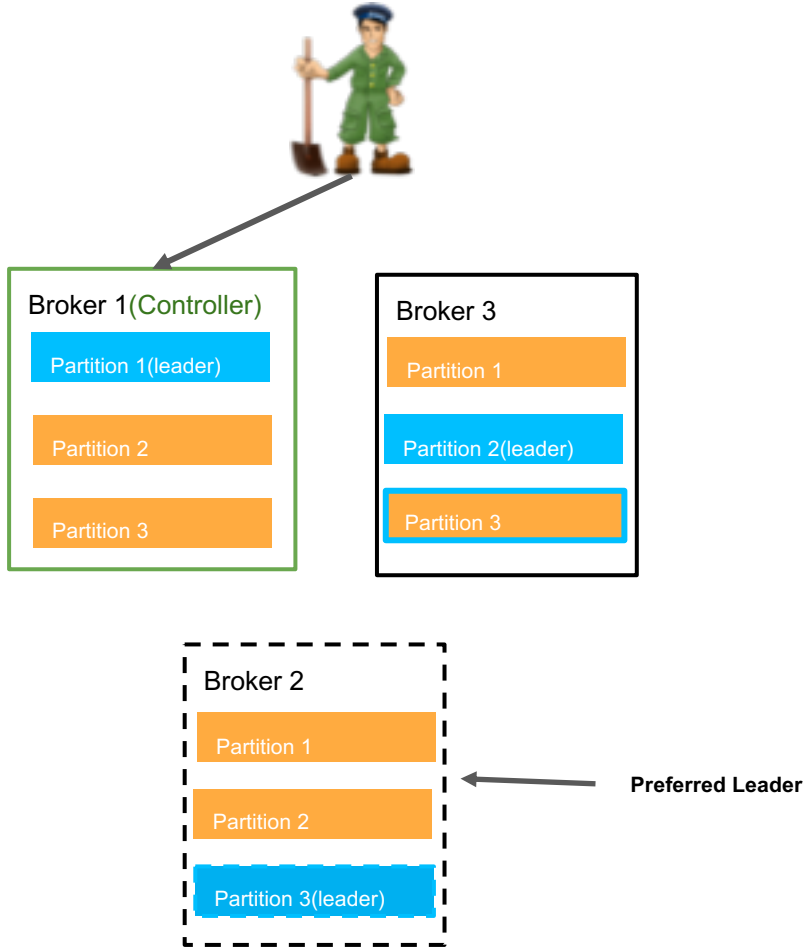


Zookeeper

Example: A node rejoins cluster

- Kafka assumes that original leader assignment is optimal (preferred leaders)
- By default Kafka will check if the preferred leader replica is not the current leader and, if it's alive, try to elect it.*
- When the controller notices, it check to see if their partitions that exist on this broker.
- The new broker starts replicating messages again
- Eventually the controller will attempt to reassign leadership back to the broker in order to optimally balance the cluster.

*(auto.leader.rebalance.enabled=true)



Hardware

Agile Brains Consulting

CLOUD NATVZ

Hardware - Disk Throughput

- Disk throughput directly effects Kafka’s ability to confirm commits.
- In most cases Kafka log segments must be written to a local storage before the client considered the write a success.
- SSD drives are usually recommended as the latency is much less than HDD.

Hardware - Disk Capacity

The amount of disk space needed will be determined by your retention policies, size of messages, and replication strategy

Hardware - Memory

High performance Kafka consumers ideally read from the end of partitions in the systems page cache sitting in the brokers memory. Therefore the more memory a broker has the less cache misses occur that result in the broker reading messages from the disk. This means stand alone Kafka brokers are superior to systems sharing page cache with other applications.

Hardware - Networking

Often networking resources can prove to be a bottleneck for your cluster. One must consider the outbound traffic multiplier that occurs if a producer writing to 1 topic has multiple consumers for that topic.

Choosing Partitions

- Throughout on future usage
  - per topic?
  - per partition/consumer?
  - per partition/producer?
- Available disk space and bandwidth per broker.
- Overestimating partitions could lead to inefficient use of resources.

Record Retention

Kafka will retain all published records based on your configuration. Your retention policies can be Time, Size, or compaction based.

Kafka and Microservices

Agile Brains Consulting

CLOUD NATVZ

Kafka and Microservices

A typical microservices solutions:

- dozens of “independent” services interacting with each other.
- services may scale up and down dynamically bringing new nodes online.
- huge problem if services are tightly coupled.

How can Kafka help?

Use Case: Kafka and Microservices

Problem: We have a simple service responsible for taking an order from an eCommerce website. Once this order is approved the shipping service needs to be called.

Monolith Solution

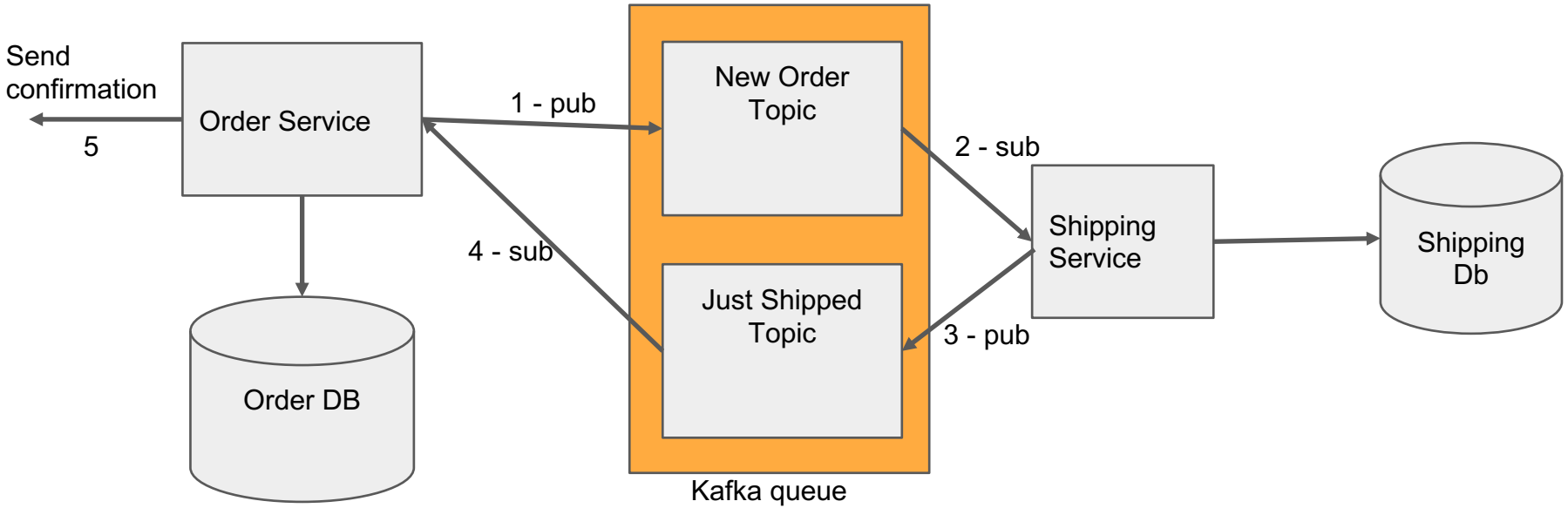
In a monolith system we would probably have all this logic in the same codebase, in a synchronous way using the multiple tables in the same DB. When the order was placed you would create a transaction that would update the sales table and the shipping table.

Microservice Solution

In a microservices environment we have decoupled these responsibilities in two different services and now we need to make sure the transaction is processed across all services.

Use Case: Kafka and Microservices

Solution: Because the DB queries now span multiple services and databases. We need to alert the other services that an event has occurred so they can respond and complete their portion of the transaction.



Without Message Broker

It is not feasible for each service to have a direct connection with every service that it wants to talk for a few reasons:

- **Scaling** : the number of such connections could grow rapidly and become difficult to manage.
- **Reliability :** the service being called may be down or may have moved to another server.
- **Durability:** The service may become overloaded with traffic causing the service to become overloaded requiring the message to be resent.

Microservices with Message Brokers

Message Brokers are a way of decoupling the sending and receiving services through the concept of Publish & Subscribe.

Scalability: consumers need only to subscribe to a Kafka topic when they come online decoupling the services from each other

Reliability: queue or retain the message till the consumer picks it up. If the consumer service is down or busy when the sender sends the message, it can always pick it up later.

Durability: Messages remain in the queue and can be pulled down when consumer is ready ensuring that no messages are lost.because a buffer is full or service is down.

Case Study: Uber

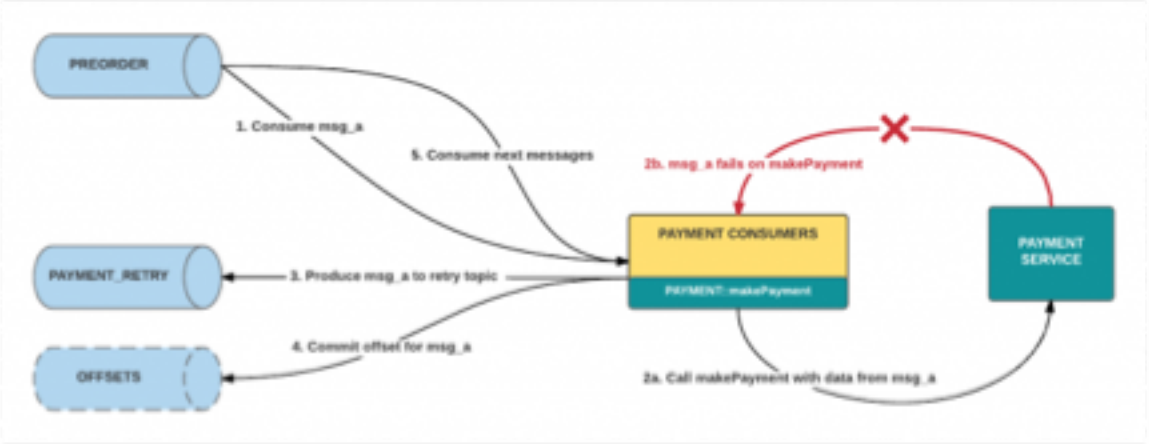
Agile Brains Consulting

CLOUD NATVZ

Case Study: Reliable Reprocessing and DLQ at Uber

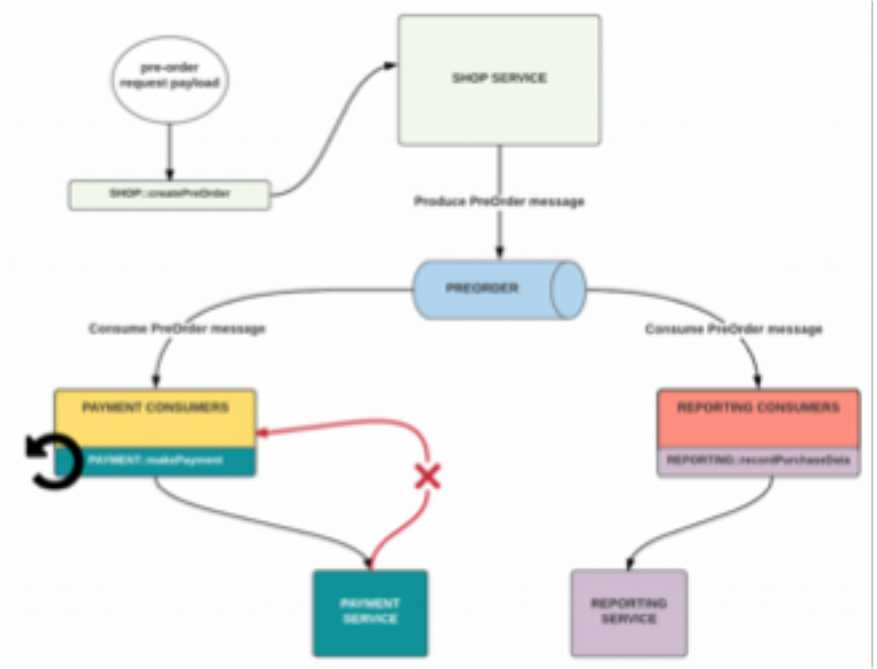
The Uber Insurance Engineering team had already been using event-driven architecture in the larger microservices ecosystem but they wanted to extend its usage to achieve decoupled, observable error handling without disrupting real-time traffic.

- non-blocking request reprocessing
- dead letter queues (DLQ)



Case Study: Reliable Reprocessing and DLQ at Uber

- Pre-order request is received,
- Shop Service publishes a PreOrder message containing relevant data about the request.
- Each of the two sets of listeners reads the produced event to execute its own business logic and call its corresponding service to do two things:
    - make a payment
    - create a separate record capturing data for each product pre-order per user to generate real-time product analytics



Case Study: Reliable Reprocessing and DLQ at Uber

How to handle failures in this system?

Strategy: Retry by using feedback cycle at the point of the client call.

Example: Payment service prolonged latency and starts throwing timeout exceptions, the shop service would continues to makePayment under some prescribed retry limit until it succeeds or another stop condition is reached.

Case Study: Reliable Reprocessing and DLQ at Uber

Problem: Retries at a large scale can be subject to batch processing and difficulty retrieving metadata.

Clogged batch processing - a large number of repeatedly failed messages can clog batch processing. Without a success response, the Kafka consumer will not commit a new offset and the batches with these bad messages would be blocked as they are re-consumed again and again.

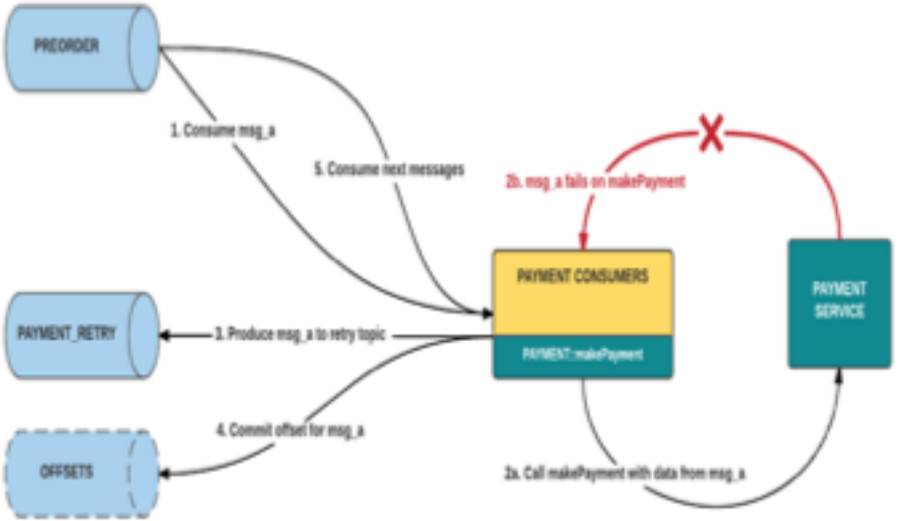
Difficulty retrieving metadata. It can be cumbersome to obtain metadata on the retries, such as timestamps and *nth* retry.

Case Study: Reliable Reprocessing and DLQ at Uber

Optimized Strategy: When a consumer handler returns a failed response after certain number of retries, the consumer publishes that message to a corresponding retry topic. The handle returns true to the original consumer which commits its offset.

A separate group of retry consumers will read off their corresponding retry queue. These consumers behave like those in the original architecture, except that they consume from a different Kafka topic.

Executing multiple retries is accomplished by creating multiple topics, with a different set of listeners subscribed to each retry topic.



Case Study: Reliable Reprocessing and DLQ at Uber

Example:

Step A:

1. Original message fails
2. Consumer receives an error response
3. Consumer publishes msg_a to the payment_retry queue
4. Commits message offset relative to pre-orders at the original processing topic.
5. The consumption of message is complete
6. Consumer moves onto the next message.

DLQ (Dead Letter Queue)

dead letter queue is a service implementation to store messages that meet one or more of the following criteria:

- Message that is sent to a queue that does not exist.
- Queue length limit exceeded.
- Message length limit exceeded.
- Message is rejected by another queue exchange.
- Message reaches a threshold read counter number, because it is not consumed. Sometimes this is called a "back out queue".

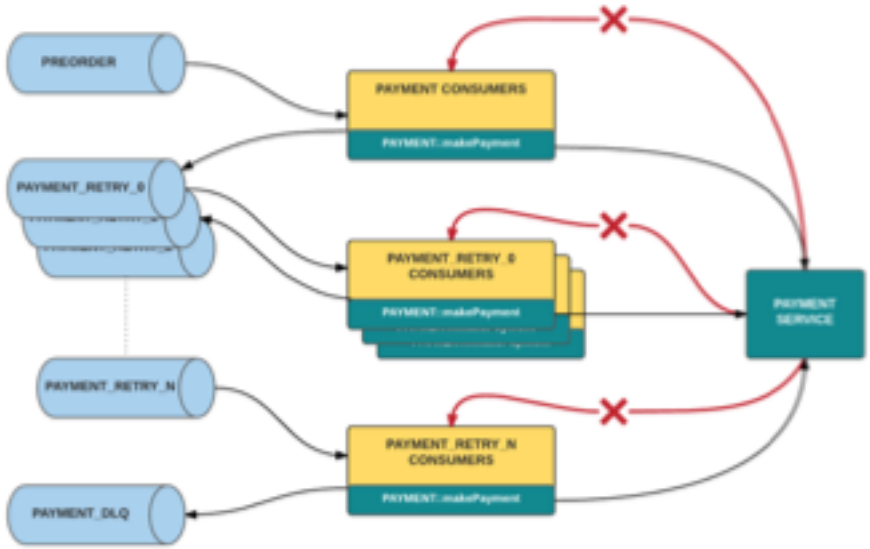
Case Study: Reliable Reprocessing and DLQ at Uber

Step B:

1. Retry consumer begins consuming messages from retry topic
2. Retry topic fails
3. Publishes message to next level retry topic*
4. The DLQ is the end-of-line Kafka topic

   From the DLQ a bunch of techniques can be employed for handling failed messages but they should not be put back into live traffic.

*Retries should not be processed immediately and each level of retries should extend the delay between processing. You can think of the retry queues tas more of a delayed processing queue.



Case Study: Reliable Reprocessing and DLQ at Uber

Benefits of Queue Based Processing

Unblocked batch processing

Failed messages enter their own designated channels, enabling successes in the same batch to proceed instead of requiring them to be reprocessed along with the failures. Thus, consumption of incoming requests moves forward unblocked, achieving higher real-time throughput.

Decoupling

Independent work streams that operate on the same event each have their own consumer flows, with separate reprocessing and dead letter queues. Failure in one dependency does not require retrying that particular message for others that had succeeded.

Flexibility

Kafka supports client libraries in several languages. For example, many services at Uber use Go for their Kafka client.

Case Study: Reliable Reprocessing and DLQ at Uber

Configurability

Creating new topics incurs practically no overhead, and the messages produced to these topics can abide by the same schema.

Observability

Segmentation of message processing into different topics facilitates the easy tracing of an errored message’s path, when and how many times the message has been retried, and the exact properties of its payload.

Performance and dependability

Kafka offers at-least-once semantics by default. When it comes to delivering business-critical data this guarantee is paramount. Kafka’s parallelism model and pull-based system enable high throughput and low latency.

Lab 3: Custom consumer and producer application

Objective: Create a producer and consumer applications that read and write to a Kafka topic.

https://github.com/shekhar2010us/kafka_t/blob/master/labs/03_single_broker_producer_consumer_python.md

Lab 4: Multiple Node Brokers

Objective: Create a Multi Node Kafka Cluster on 2 Ubuntu VM's

https://github.com/smhillin/kafka_essentials/blob/master/multi_node_multi_broker_kafka.md

Kafka Extended APIs

- **Kafka Connect**

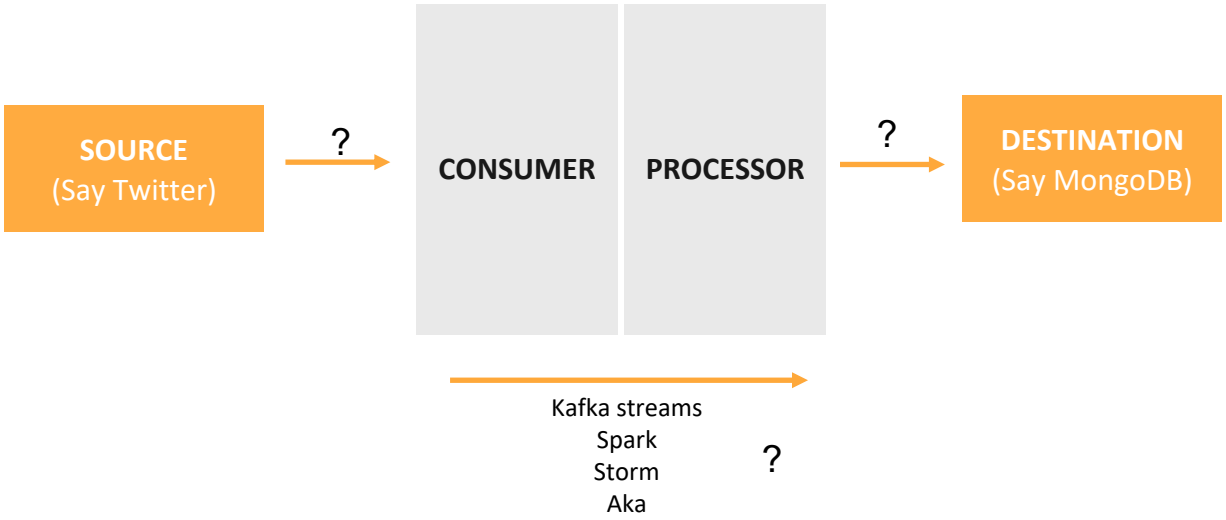
Agile Brains Consulting

CLOUD NATVZ

Kafka Connect

Common use-case

Connectors are very common, Kafka allows to extend their library for different connectors



Kafka Connect History

2013 Kafka 0.8.x

- Topic replication, Log compaction
- Simplified producer client API

2015 Kafka 0.9.x

- Simplified high level consumer APIs, without zookeeper dependency to store offset
- Kafka connect APIs

2016 Kafka 0.10.0

- Kafka Stream APIs

Relatively new and advanced

2017 Kafka 0.10.2

- Improved and Matured Connect and Stream APIs

Kafka Common Use Cases

Source => Kafka	(Producer API) or Kafka connect source
Kafka => Kafka	(Kafka stream)
Kafka => Sink	(consumer api) or Kafka connect sink
Kafka => App	(consumer api)

Kafka Connect

Kafka Connectors are a reusable component that knows how to talk to specific source and sinks

- Copies between Kafka and some other data system
- Simplify and improve getting data in and out of Kafka
- Simplify transforming data within Kafka without relying on external libs
- Distributed/Scalable
- Reusable
- Extensible

Kafka Connect

- Programmers always want to import data from **some sources** like:

RDBMS, Couchbase, SAP, Cassandra, FTP, IOT, Twitter, Solr, etc.

- Programmers always want to import data from **some sink** like:

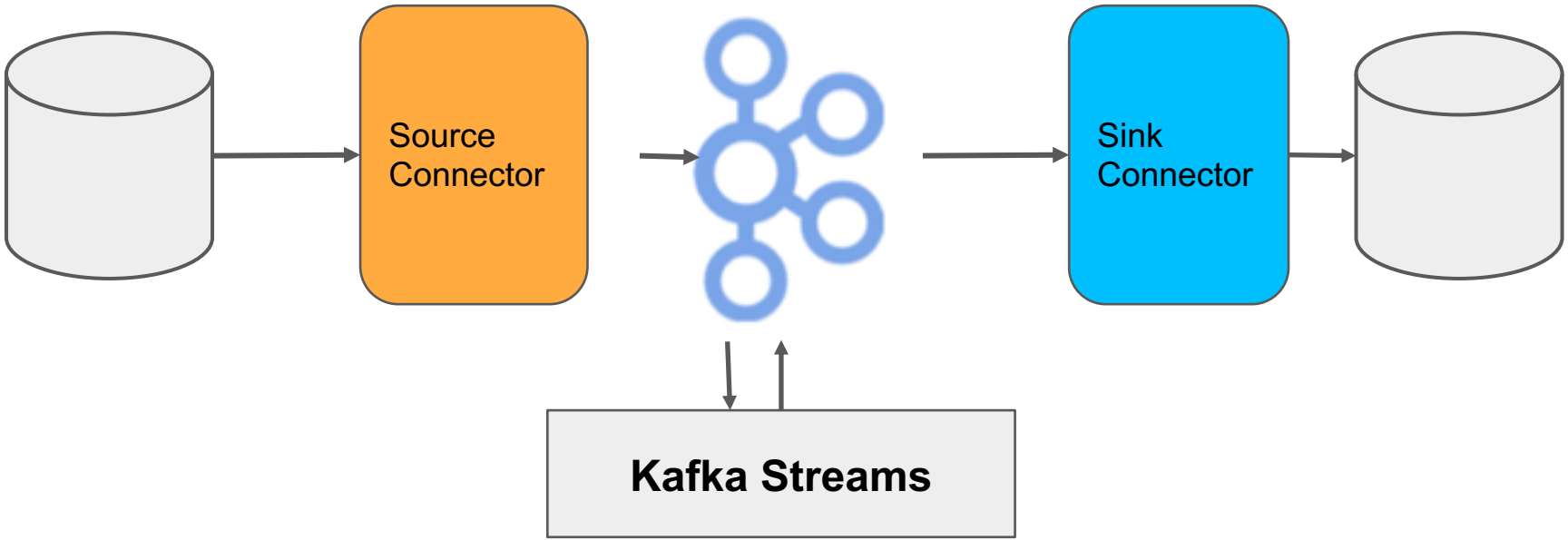
S3, Elasticsearch, HDFS, RDBMS, Couchbase, SAP, Cassandra, Solr, etc.

- It is tough to achieve Fault Tolerance, Distribution, Idempotence, Ordering – if you code for all source and sink connections

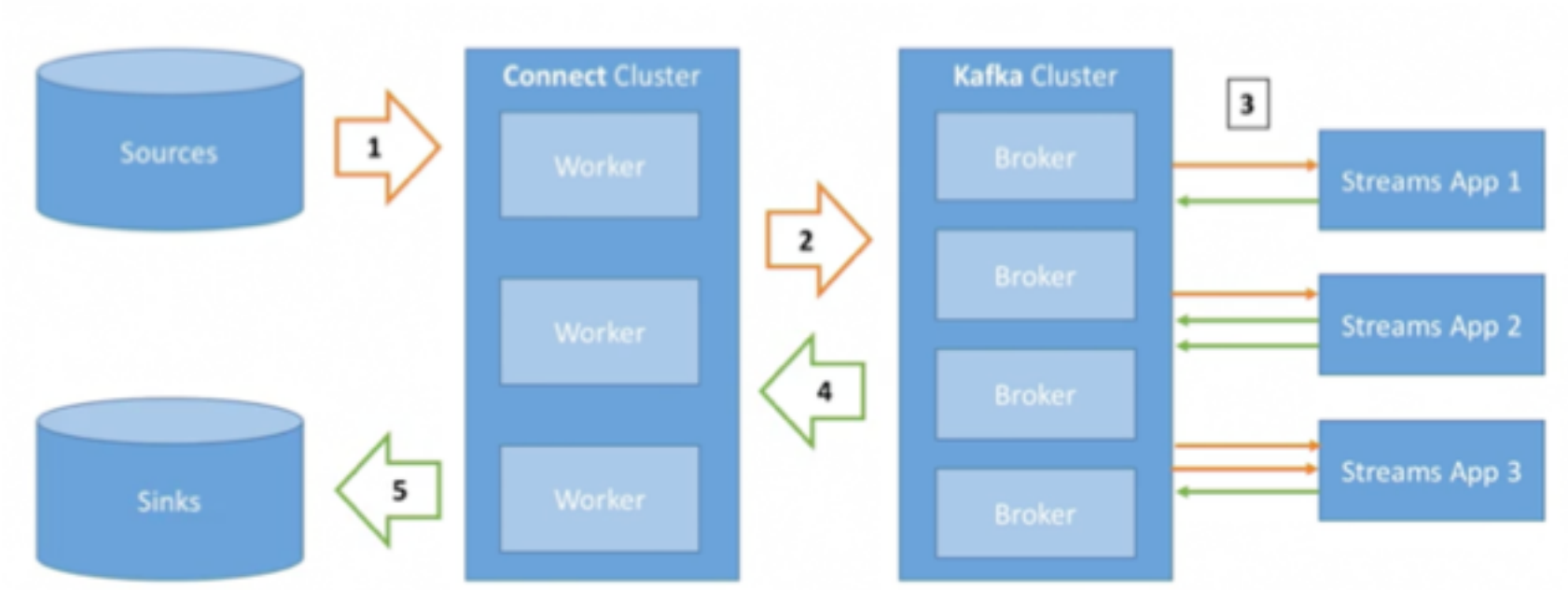
Other programmers have already solved it, and made their solution open source

That's the idea behind **Kafka Connect**

Kafka Connect Architecture



Kafka Connect Architecture



Kafka Connect Repository

Library of connectors that can be reused for popular technologies.

https://www.confluent.io/hub
https://www.confluent.io/product/connectors-repository

Confluent Connectors
Certified Connectors
Community Connectors

Twitter – Source Connector

Library are two Twitter Source Connectors, One of them is below:

https://github.com/jcustenborder/kafka-connect-twitter

Go over this connector !!!!

Add lab on connect

https://www.linkedin.com/learning/learn-apache-kafka-for-beginners/kafka-connect-twitter-hands-on-example

Kafka Stream

Agile Brains Consulting

CLOUD NATVZ

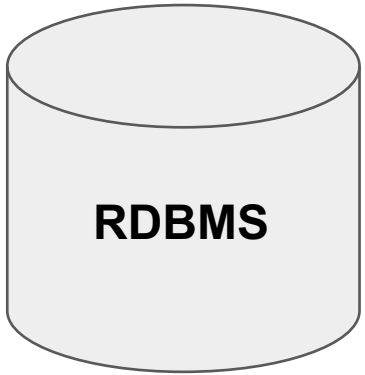
1. Streaming Data

Agile Brains Consulting

CLOUD NATVZ

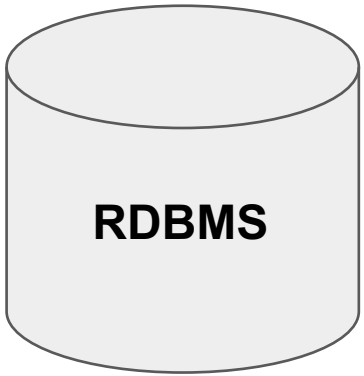
Traditional Way

Data represented by current state that is updated as needed, but does only the current state matters??



Traditional Way

{
“Time”: 14757577575376
“student_id” : 12345
“class_id” : 5672
“Class_detail: ….
}



The Streaming Way

- All data can be represented by an event stream
- A stream adds the dimension of time to your data
- All software does is process a stream of events
- Kafka process events without assuming anything about the end state of the data

Time based Events (Table with time)

Time = 0

Key 1	Value 1.0

Time = 1

Key 1	Value 1.0
Key 2	Value 2.0

Time = 2

Key 1	Value 1.1
Key 2	Value 2

Time based Events (Table with time)

Time = 0

PUT(Key1,value1)

Key 1	Value 1.0

Time = 1

PUT(Key2,value2)

Key 1	Value 1.0
Key 2	Value 2.0

Time = 2

PUT(Key1,value1.1)

Key 1	Value 1.1
Key 2	Value 2

Use Case: Logs to Business Dashboards

Application Log

Stream Processing

Process

Show on dashboard

- Technical Errors
- Business KPIs

Use Case: Twitter Sentiments

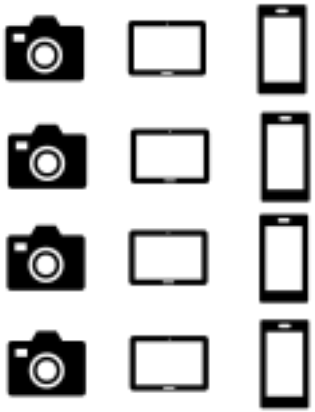
Tweet

Stream Processing

Filter tweet
Predict sentiment

Show results on a dashboard

Use Case: IoT (internet of things)



Devices

Stream Processing

Collect data
Process
Predict, Persist

Show reports on a dashboard

Stream Processing – Additional **Time** Dimension

- Add “time” as a new dimension
- Stream operations are performed on time-windows or micro-batching
- Time become a complex topic when discussing distributed systems
- Having a clear notion of time is important for your system

Stream Processing – Additional **Time** Dimension

- No system is 100% reliable
- Need of a reliable design

- System processing on a 2 minute time-windows
- Goes offline for a few hours

What do we do with all the unprocessed data?

Time

Event timing - The time that events occurred and the record was created

Log append time - The time that the event arrived to the Kafka broker and was stored

Processing Time - The time at which a stream-processing application received the event

State

- Single events processing is relatively straightforward
- Stream processing with multiple events is complex.
- Not enough to look at each event by itself
- Keep track of more information - how many events of each type did you see this hour, all events that require joining, sums, averages, etc.

State of an event: Current state of an event – but the event may or may not be mutually exclusive

Time Windows and States

When designing streaming systems you have to know the details of the time-window

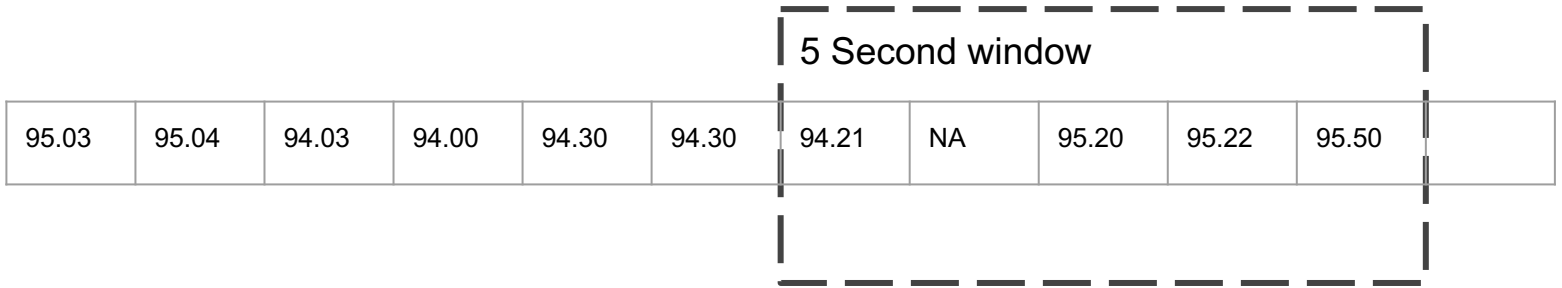
Example: Find the moving average of a stock index

Size of the window: Find 5 second moving average

How often the window moves: Our stock ticker updates every second

How long the window remains updateable: If events arrive late what do we do with those events. The eternal debate on speed vs. consistency.

Stock Ticker 5 Second Moving Average



At every second, find average of previous 5 seconds

Finite State Machines

Example: Find number of minutes each customer has watched a certain program or a channel

How often do you get a batch: every second (for thousands of customers) -> tune events

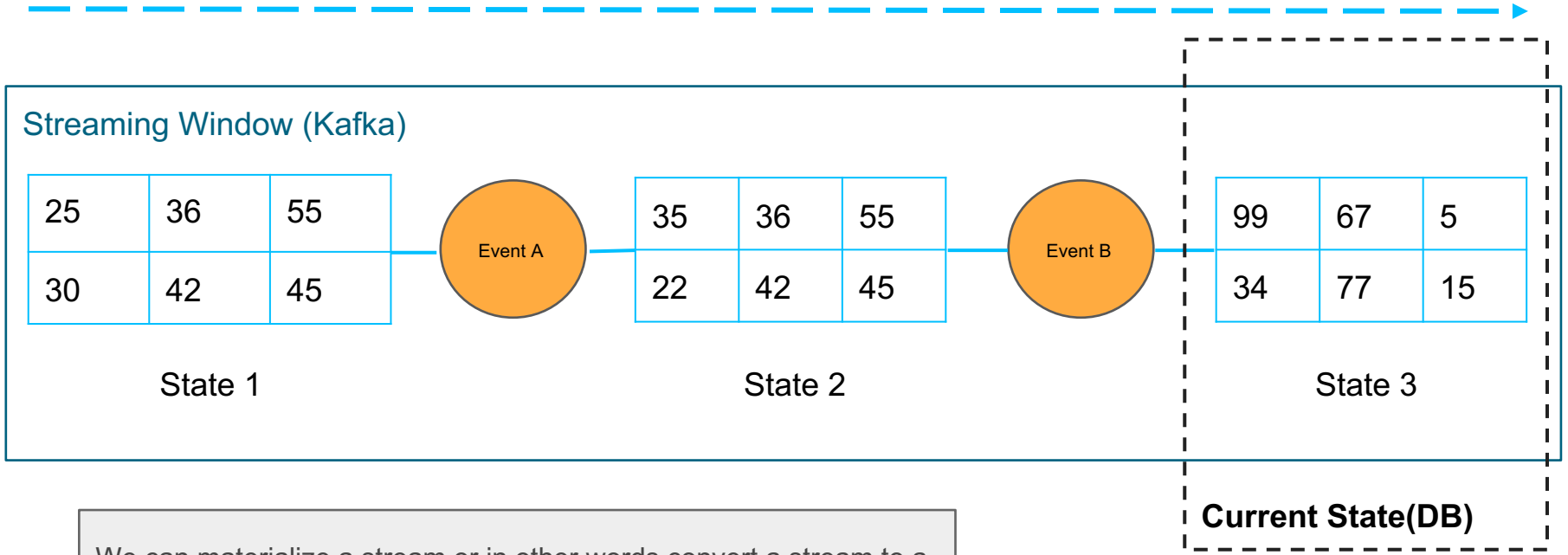
A million customer is watching TV at a given point in time

Goal: Find how much time a customer has watched a channel or program. Also, keep a tab of which channel the customer has changed to

How: You need data for the current (thousand customer) and the previous (million customers) state.

Check the **session end**, if session ended for a customer, persist. Otherwise keep the customer in the state

Matrix of States from Streams



We can materialize a stream or in other words convert a stream to a table by applying all the changes that the stream contains.

2. Kafka Streams

Agile Brains Consulting

CLOUD NATVZ

Kafka Streams

Traditionally Kafka served as the messaging system that enabled a reliable stream of events to other platform such as Apache Spark Streaming, Storm, and Flink to name a few.

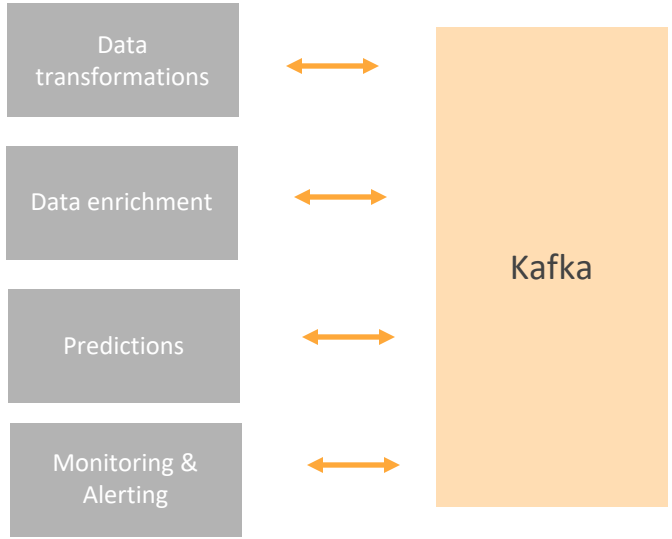
Starting with version 0.10

- provides an API for processing its streams
- in the context of your application
- Kafka Streams is part of native Apache Kafka project
- JAR file that executes inside your application

What is Kafka Streams

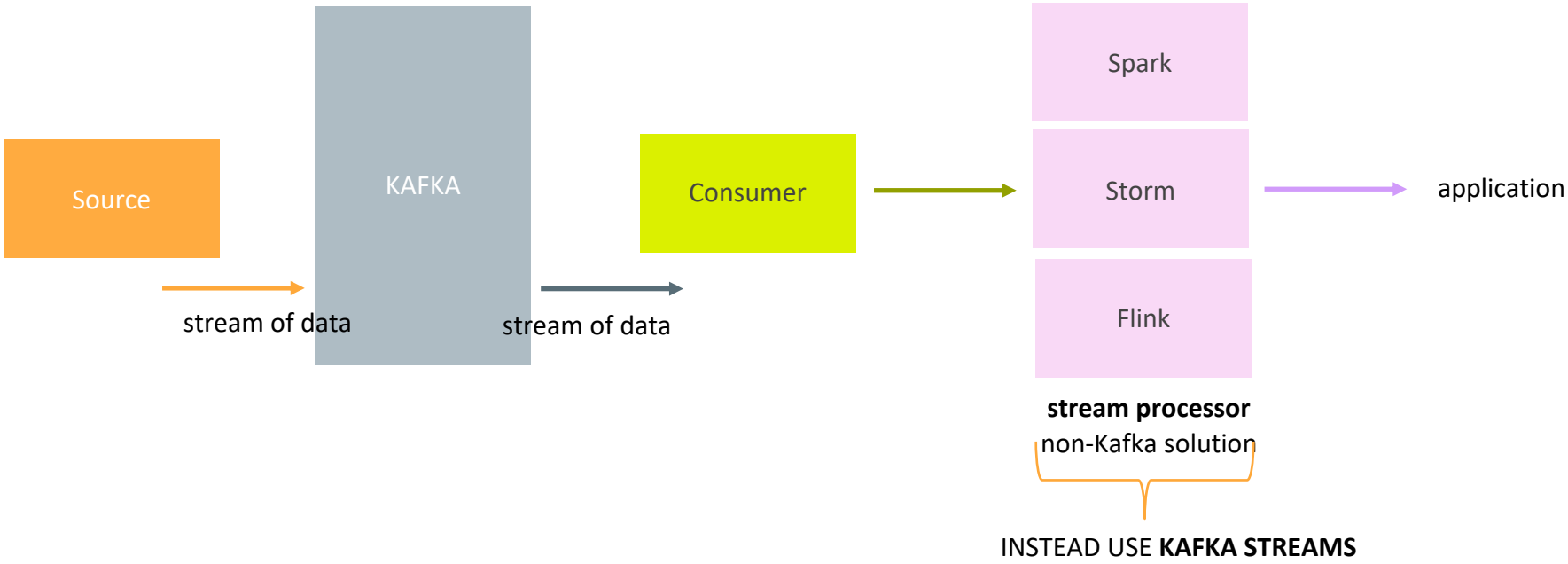
Easy **data processing and transformation library** within Kafka

- Standard Java Application
- No need to create a separate cluster
- Highly scalable, elastic and fault tolerant
- **Exactly Once** capabilities
- One record at a time (no micro-batching)
- Works for any application size
- Scales just like consumer groups

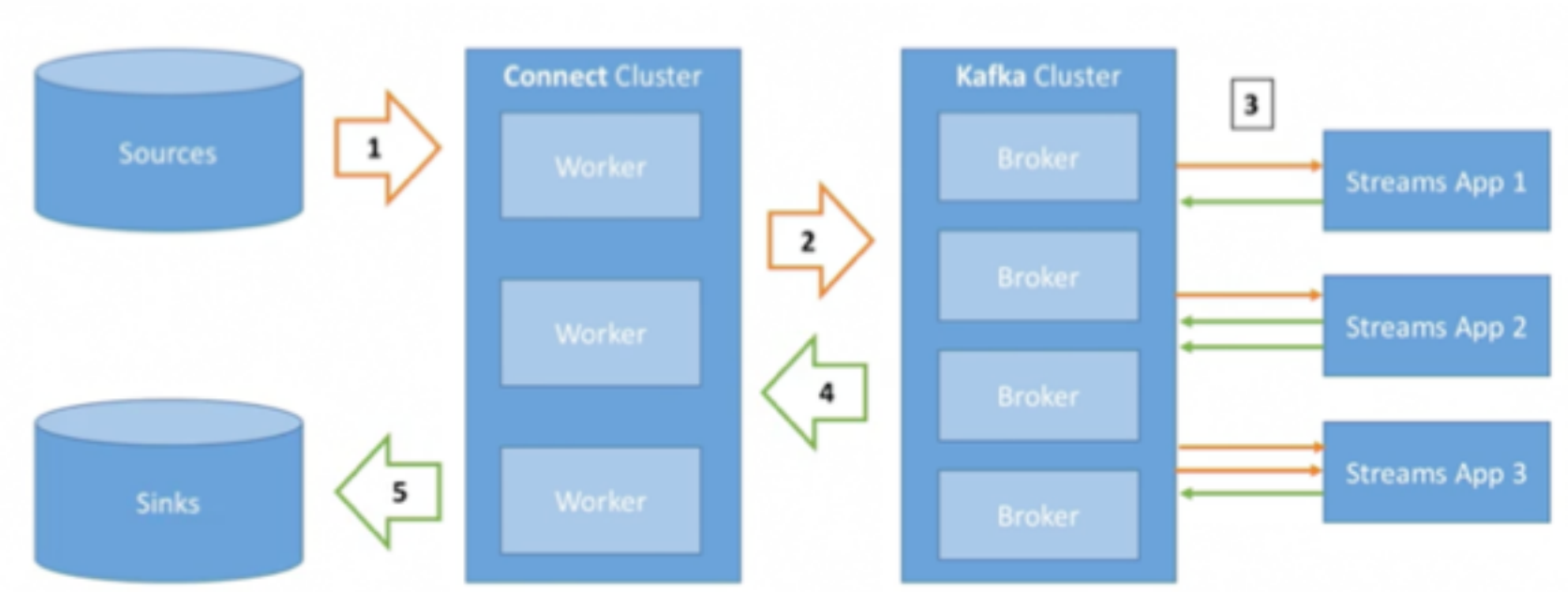


All within Kafka

Kafka Streams



Kafka Stream Architecture



Data Transformation and Processing

Kafka Stream History

2016 Kafka 0.10

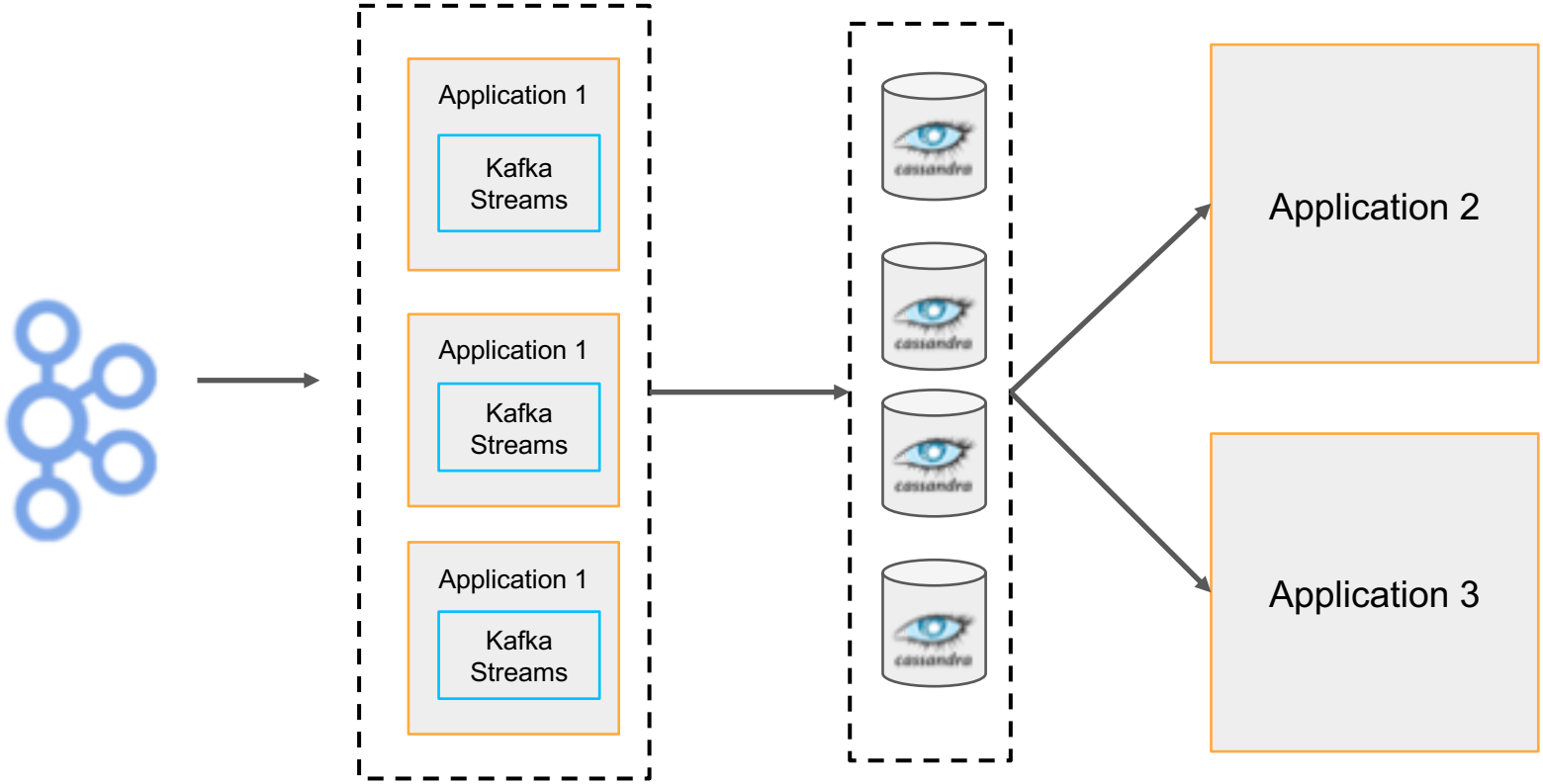
Introduced Kafka Stream

2017 Kafka 0.11

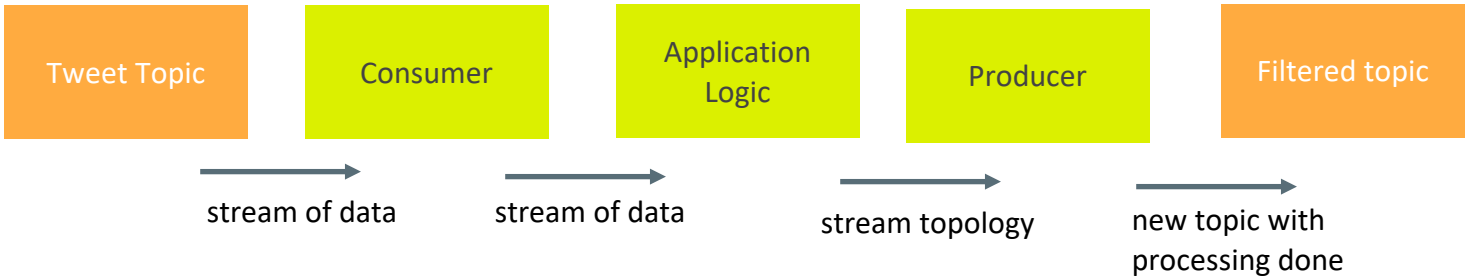
Matured Kafka Stream

- It’s the only library that can leverage the new exactly once capabilities from Kafka 0.11
- Contender to Spark, Flink, Storm

Stream Processing Design Patterns



Kafka Stream Example



Add lab on Kafka Streams

https://www.linkedin.com/learning/learn-apache-kafka-for-beginners/kafka-streams-hands-on-example

Lab 5: Kafka Streams

Objective: Get a high level view of Apache Streams Architecture

https://github.com/smhillin/kafka_essentials/blob/master/simple_kafka_streaming.md

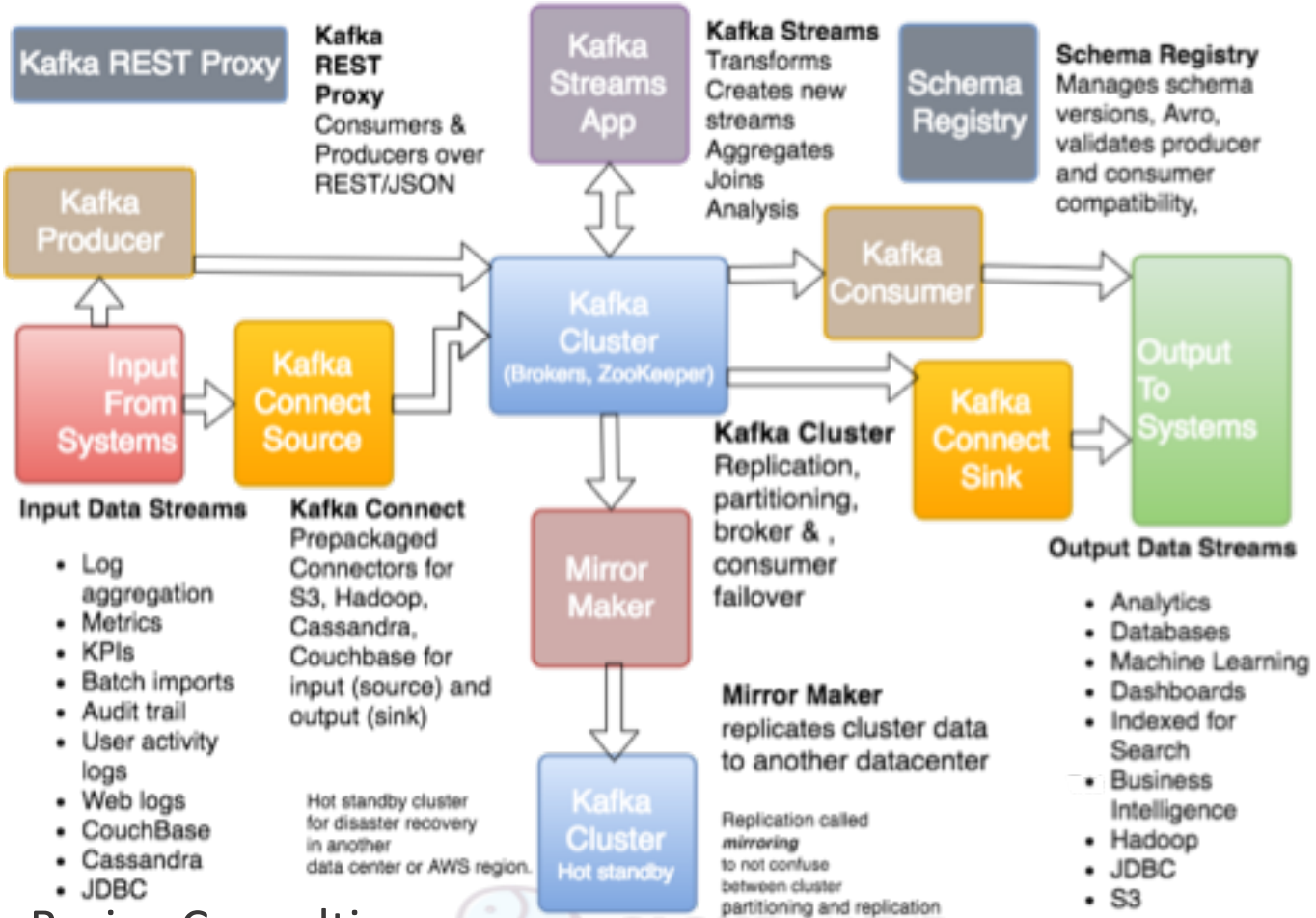
Kafka Ecosystem

Agile Brains Consulting

CLOUD NATVZ

Kafka Ecosystem

The Kafka ecosystem consists of Kafka Core, Kafka Streams, Kafka Connect, Kafka REST Proxy, and the Schema Registry. Most of the additional pieces of the Kafka ecosystem comes from **Confluent** and is not part of Apache.



Kafka Ecosystem

Kafka Core

the core of Apache distribution of Kafka is the brokers, topics, logs, partitions, and cluster. The core also consists of related tools like MirrorMaker.

Kafka Stream

the Streams API to transform, aggregate, and process records from a stream and produces derivative streams.

Kafka Connect

the connector API to create reusable producers and consumers (e.g., stream of changes from DynamoDB). The Kafka REST Proxy is used to producers and consumer over REST (HTTP).

Schema Registry

manages schemas using Avro for Kafka records.

Kafka Schema Registry

Kafka properties

- No Data Verification
- Kafka doesn't know what data is being transmitted
- Kafka doesn't know the schema or type of data

What would happen when

- Producer sends bad data
- Fields are renamed
- Data format changes

→ **Consumers will break !!**

Kafka Schema Registry

- Need data to be self describable
- Need to be able to evolve data without breaking downstream

We need Schema Registry !!

Can Kafka Broker do schema registry ?

- It can but Kafka won't be as good as it is
- Kafka do not consume CPU, just gets and send bytes

Kafka Schema Registry

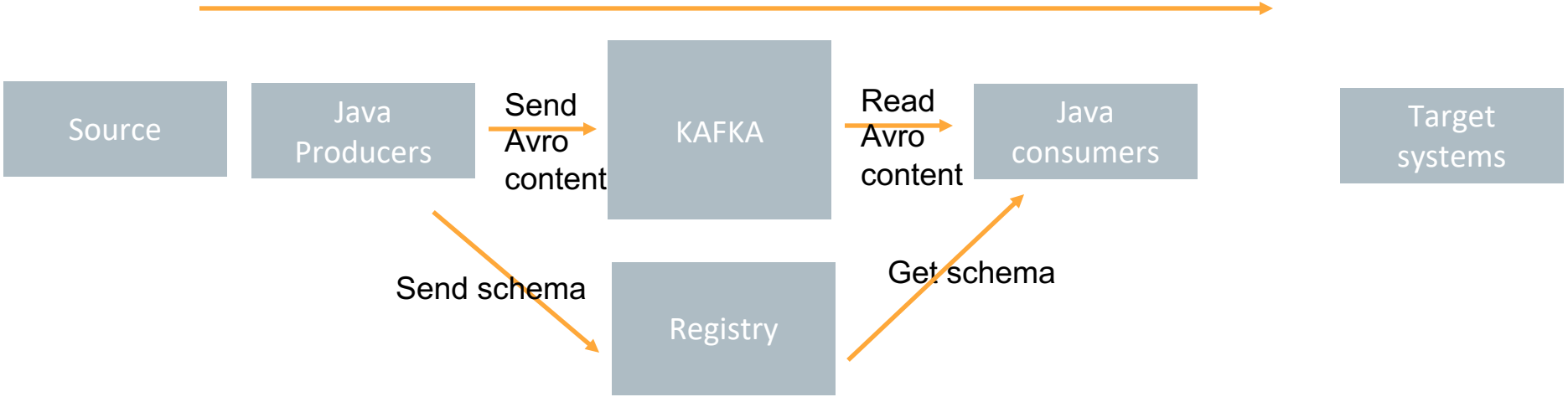
Schema Registry Properties

- Schema registry – separate component
- Producer and Consumer – should talk to the Schema registry
- Ability to reject bad data
- Common data format must be agreed upon
- Needs to support schemas
- Lightweight

Confluent Schema Registry
They chose Apache Avro as data format

Pipeline without Registry

Pipeline with Registry



Client Libraries

Agile Brains Consulting

CLOUD NATVZ

Kafka Client Libraries

C/C++
Python
Go (AKA golang)
Erlang
.NET
Clojure
Ruby
Node.js
Proxy (HTTP REST, etc)

Perl
stdin/stdout
PHP
Rust
Alternative Java
Storm
Scala DSL
Clojure
Swift
Client Libraries Previously Supported

Multiple Libraries for each language...
https://cwiki.apache.org/confluence/display/KAFKA/Clients

Kafka Client Libraries

Pure Python implementation with full protocol support. Consumer and Producer implementations included, GZIP, LZ4, and Snappy compression supported.

http://github.com/dpkp/kafka-python

Kafka Version: 0.8.x, 0.9.x, 0.10.x, 0.11.x, 1.0.x

Maintainer: Dana Powers

License: Apache 2.0

Kafka and Big Data
- Netflix Case Study

Agile Brains Consulting

CLOUD NATVZ

Big Data - 3 V’s

As the size and complexity of data grows unique problems emerge.

- **Volume** - Very large data sets
- **Velocity -** Generated at high speeds
- **Variety** - Sourced from many end devices and applications

Data Ingestion

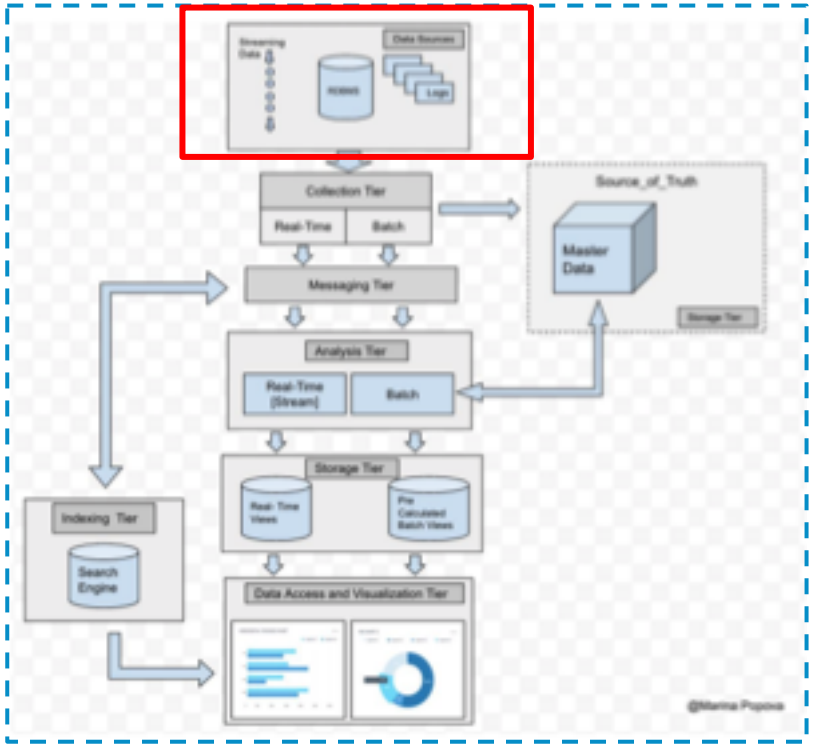
In the era of Big Data companies want the ability to analyze large amounts of data in batch and more increasingly real-time. Before we can start to make sense of the data we need the ability to ingest the date and then route it to the write places to be stored, processed and analyzed.

Lambda Architecture (batch and stream)

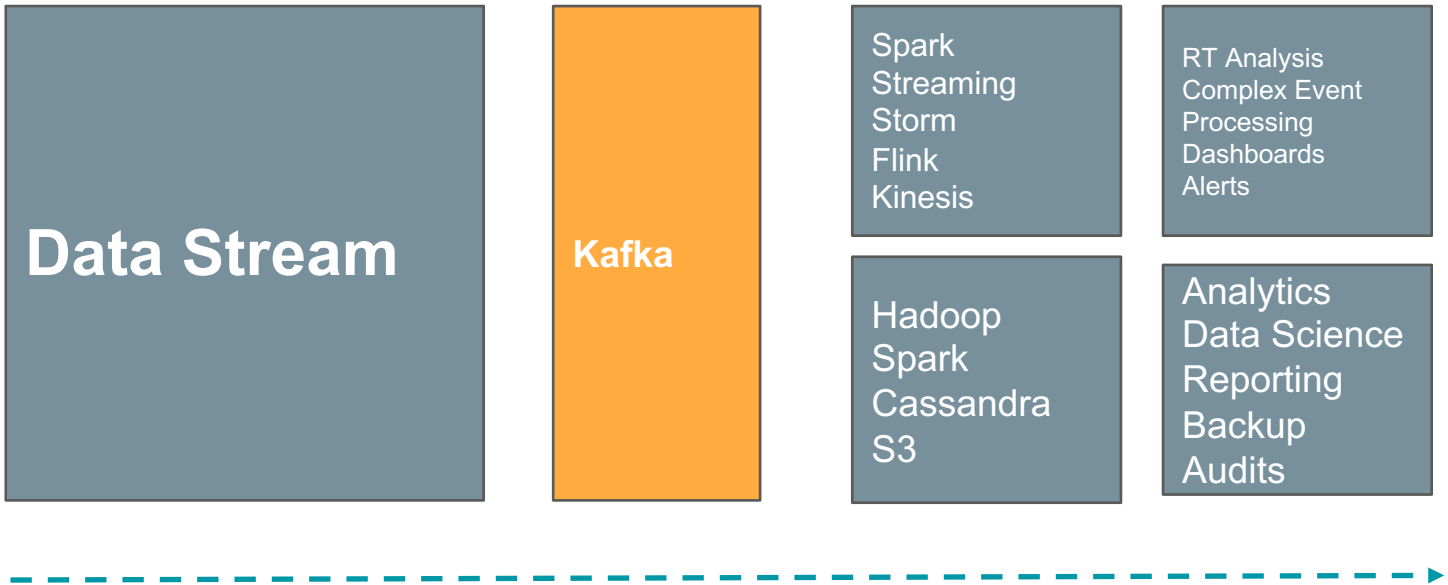
Kappa Architecture (stream processing)

Lambda Architecture

1. Collection tier
2. Messaging tier
3. Analysis tier
    a. Fast real-time
    b. Batch analysis
4. Data storage tier
    a. Fast short-term real-time access
    b. Slower long term storage
5. Data access tier / UI
    a. Indexing/ Search Engine tiers
    b. Data access/ visualization tiers



Messaging and Collection Tier



Case Study: Netflix

Agile Brains Consulting

CLOUD NATVZ

Case Study: Kafka at Netflix

Almost every application at Netflix uses some sort of a data pipeline because decisions at Netflix rely on the ability to collect, aggregate, process large amounts of Data. Netflix Keystone Big Data Pipeline went live in December 2015.

https://www.youtube.com/watch?v=WuRazsX-MBY

Case Study: Kafka at Netflix

The ability to ingest events at scale is the first step to reliably capturing the data for analysis. And Netflix has a lot of events.

- ~500 billion events and ~1.3 PB per day
- ~8 million events and ~24 GB per second during peak hours

Case Study: Kafka at Netflix

These events flow through several hundred streams.

For example:

- Video viewing activities
- UI activities
- Error logs
- Performance events
- Troubleshooting & diagnostic events

Case Study: Kafka at Netflix

V1.0 Chukwa pipeline

Netflix data pipeline has had to evolve rapidly over the last few years to keep up with proliferation of data. The original pipeline was simple architecturally the purpose was to aggregate and upload events to Hadoop/Hive for batch processing

Case Study: Kafka at Netflix

Apache Chukwah

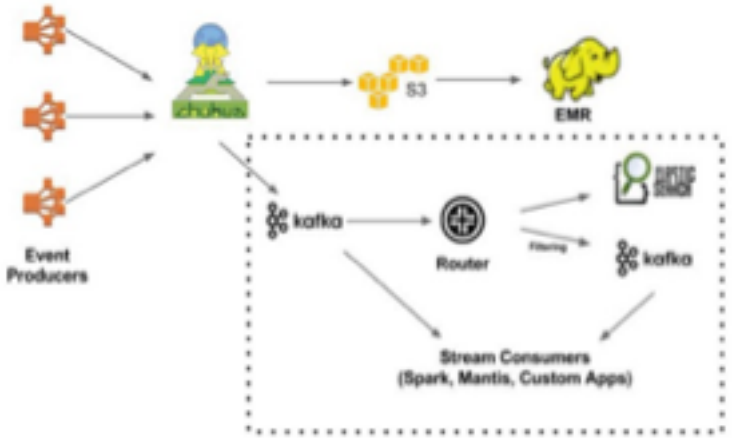
Open source data collection system for monitoring large distributed systems.
Built on top of the HDFS and MR

Case Study: Kafka at Netflix

V1.5 Chukwa pipeline with real-time branch

Elasticsearch has enabled data pipelines to do some impressive real time analytics

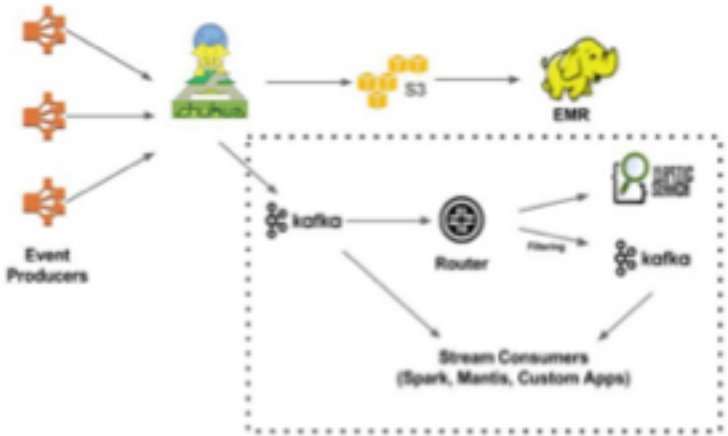
(latency < min).



Case Study: Kafka at Netflix

V1.5 Chukwa pipeline with real-time branch

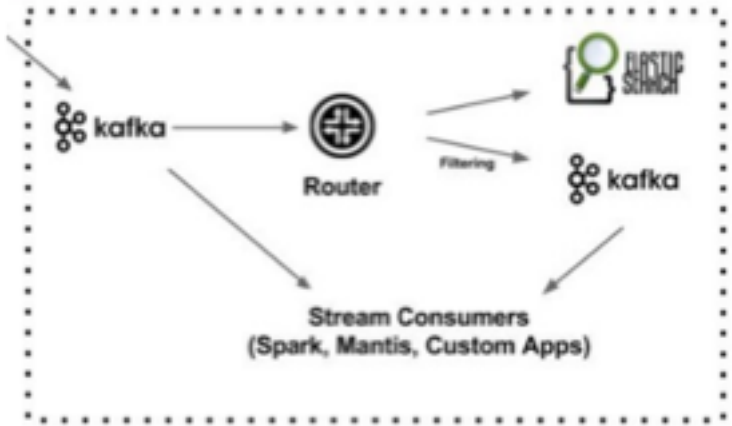
- Added the ability see traffic to Kafka (the front gate of real-time branch). In V1.5,
- Approximately 30% of the events are branched to the real-time pipeline.
- Router is responsible for routing data from Kafka to the various sinks: Elasticsearch or secondary Kafka.
- Elasticsearch uses exploded to 150 clusters totaling ~3,500 instances hosting ~1.3 PB of data.



Case Study: Kafka at Netflix

V1.5 Chukwa pipeline with real-time branch

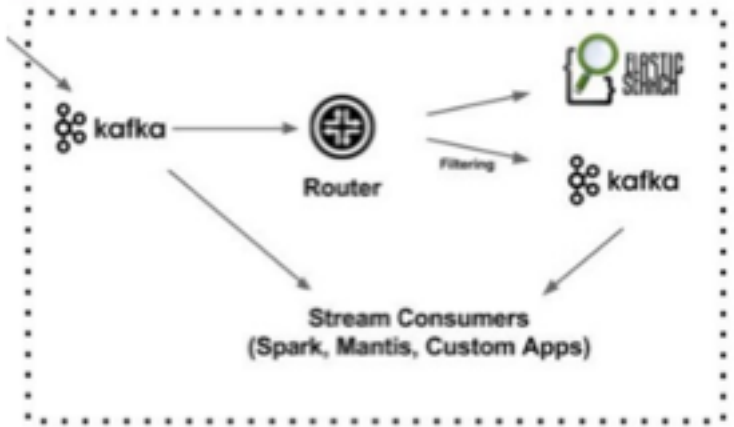
- Chukwa can deliver full or filtered streams to kafka
- Router consumes from one Kafka topic provides further filtering and produces to a seperate Kafka topic.
- From second Kafka topic real-time stream processing occurs: Mantis, Spark, or custom applications.
- Because Kafka Kafka decouples applications user can choose their own stream processing tool



Case Study: Kafka at Netflix

V1.5 Lessons Learned

- The Kafka high-level consumer can lose partition ownership and stop consuming some partitions after running stable for a while. This requires us to bounce the processes.
- When pushing new code, sometimes the high-level consumer can get stuck in a bad state during rebalance.
- Grouping hundreds of routing jobs into a dozen of clusters results in a an operational overhead of that is an increasing burden. They needed a better platform to manage the routing jobs.



Case Study: Kafka at Netflix

V2.0 Keystone pipeline (Kafka fronted)

Three major components of this architecture

- Data Ingestion
- Data Buffering
- Data Routing

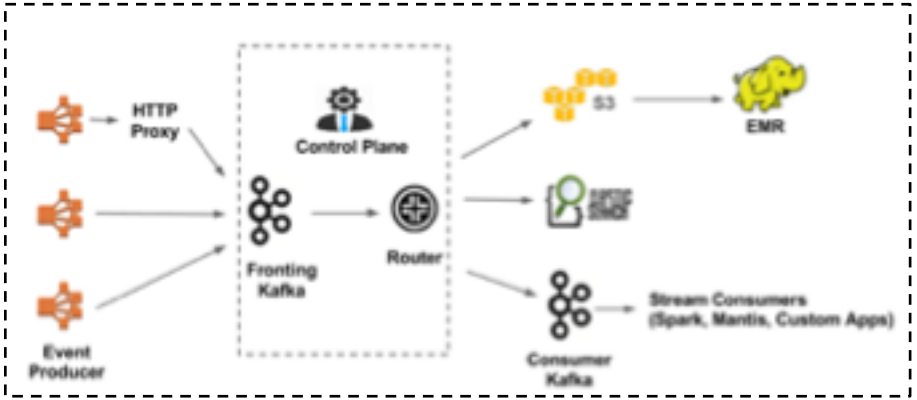
Case Study: Kafka at Netflix

V2.0 Keystone pipeline (Kafka fronted)

Data Ingestion:

Two ways for applications to ingest data:

- use Java library and write to Kafka directly.
- send to an HTTP proxy which then writes to Kafka.

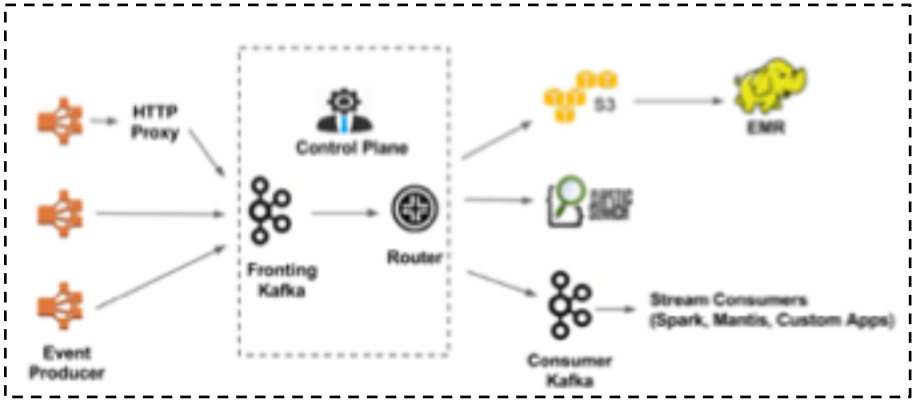


Case Study: Kafka at Netflix

V2.0 Keystone pipeline (Kafka fronted)

Data Buffering:

Kafka serves as the replicated persistent message queue. It also helps absorb temporary outages from downstream sinks.

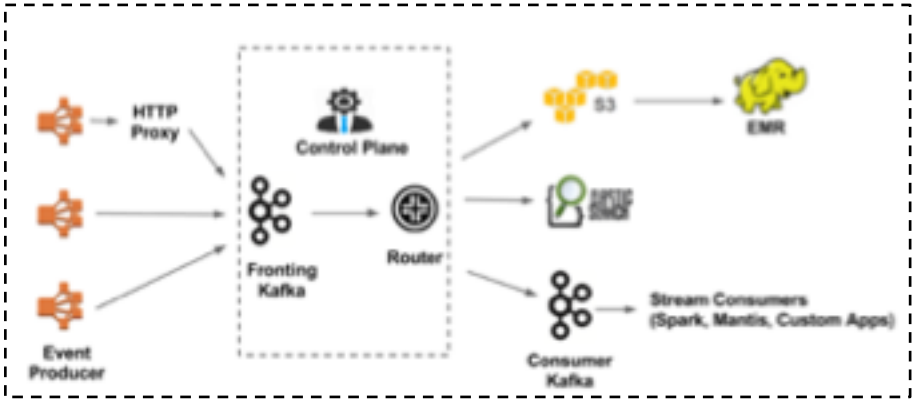


Case Study: Kafka at Netflix

V2.0 Keystone pipeline (Kafka fronted)

Data Routing:

The routing service is responsible for moving data from fronting Kafka to various sinks: S3, Elasticsearch, and secondary Kafka.

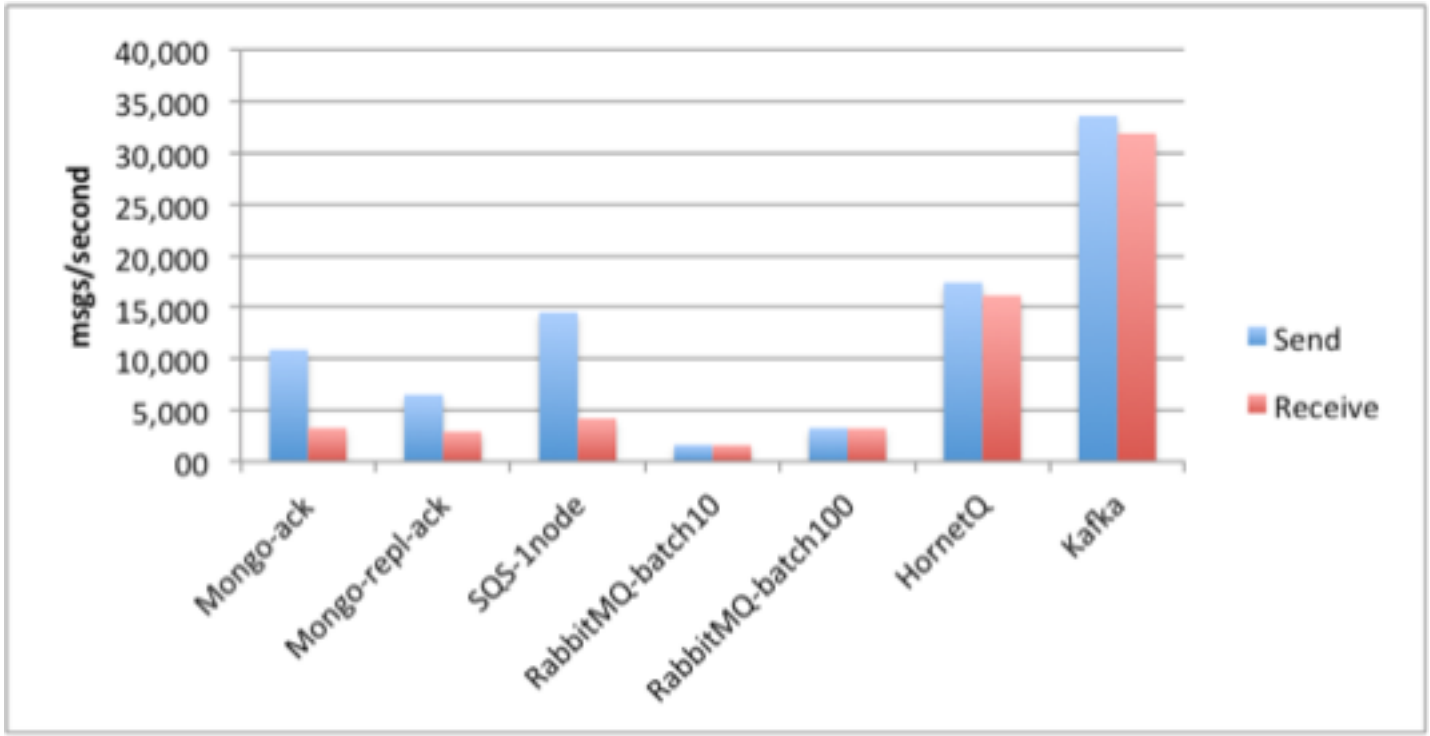


End Notes

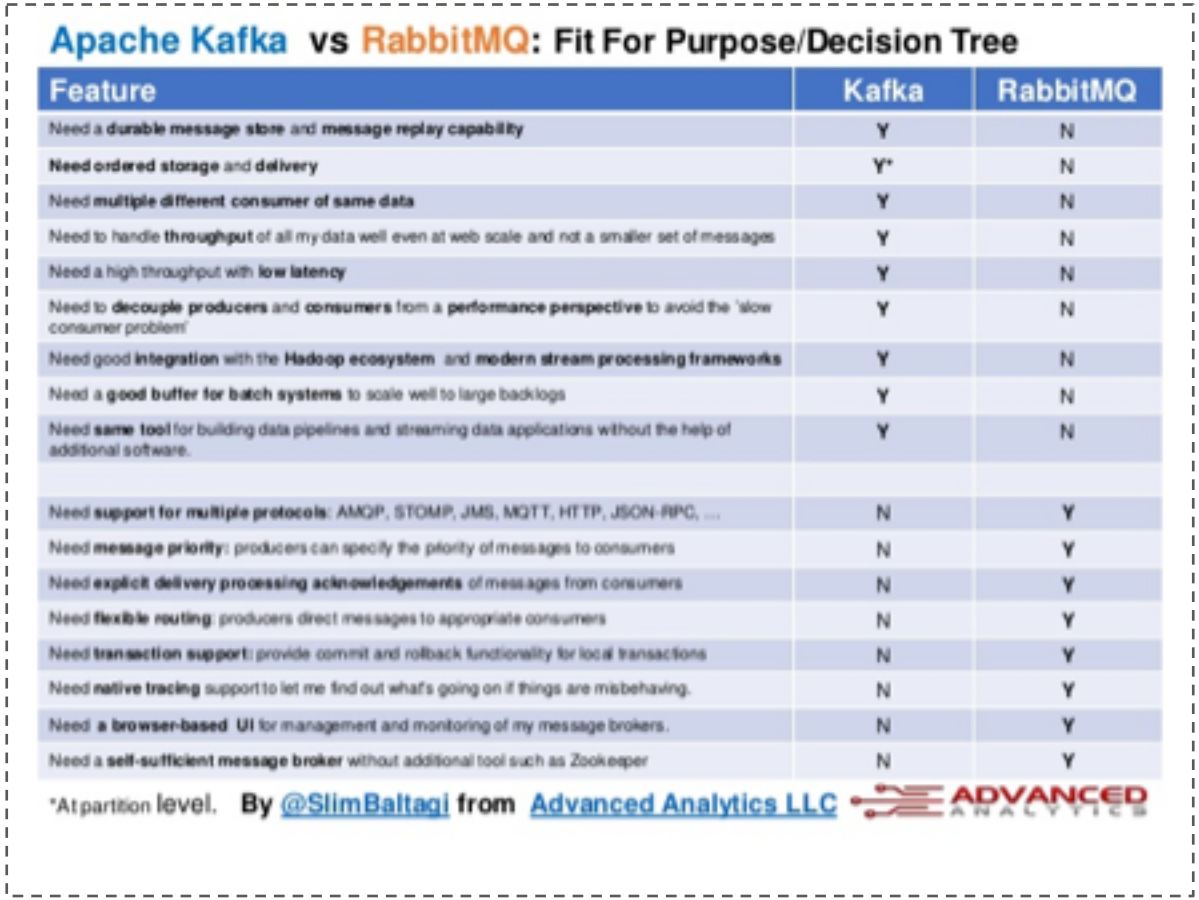
Agile Brains Consulting

CLOUD NATVZ

How does Kafka Stack Up



Kafka vs RabbitMQ decision tree



Apache Kafka vs RabbitMQ: Fit For Purpose/Decision Tree

Feature	Kafka	RabbitMQ
Need a **durable message store** and **message replay capability**	Y	N
Need ordered storage and **delivery**	Y*	N
Need **multiple different consumer of same data**	Y	N
Need to handle **throughput** of all my data well even at web scale and not a smaller set of messages	Y	N
Need a high throughput with **low latency**	Y	N
Need to **decouple producers** and **consumers** from a **performance perspective** to avoid the 'slow consumer problem'	Y	N
Need good **integration** with the **Hadoop ecosystem** and **modern stream processing frameworks**	Y	N
Need a **good buffer for batch systems** to scale well to large backlogs	Y	N
Need **same tool** for building data pipelines and streaming data applications without the help of additional software.	Y	N
Need **support for multiple protocols**: AMQP, STOMP, JMS, MQTT, HTTP, JSON-RPC, ...	N	Y
Need **message priority**: producers can specify the priority of messages to consumers	N	Y
Need **explicit delivery processing acknowledgements** of messages from consumers	N	Y
Need **flexible routing**: producers direct messages to appropriate consumers	N	Y
Need **transaction support**: provide commit and rollback functionality for local transactions	N	Y
Need **native tracing** support to let me find out what's going on if things are misbehaving.	N	Y
Need **a browser-based UI** for management and monitoring of my message brokers.	N	Y
Need a **self-sufficient message broker** without additional tool such as Zookeeper	N	Y

*At partition level. By @SlimBaltagi from Advanced Analytics LLC

Confluent

Confluent is a hosted event streaming service created by the founder of Kafka. It is a turnkey solution to managing Kafka clusters in production.

Additional Resources

Kafka White Paper
http://notes.stephenholiday.com/Kafka.pdf

Confluent Blog
https://www.confluent.io/blog/tag/white-paper/

15 Minutes to getting Kafka running on Kubernetes
https://technology.amis.nl/2018/04/19/15-minutes-to-get-a-kafka-cluster-running-on-kubernetes-and-start-producing-and-consuming-from-a-node-application/

THANK YOU !!