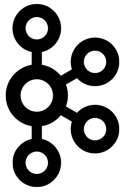
Apache Kafka  -- Unified Logging Platform

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

- The slides are adapted from the following source materials:
  - Jay Kreps, "The Log: What every software engineer should know about real-time data's unifying abstraction," Linkedin Engineering Blog, https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying, Dec 2013.
  - Jun Rao, "Intra-cluster Replication for Apache Kafka," ApacheCon 2013.
  - Joel Koshy, "Building a Real-Time Data Pipeline: Apache Kafka at Linkedin," Hadoop Summit, June 2013.
  - Martin Kleppmann, "Apache Samza: Taking stream processing to the next level," 2014
  - Martin Kleppmann:"Moving faster with data streams:The rise of Samza at LinkedIn." 14 July 2014. http://engineering.linkedin.com/stream-processing/moving-faster-data-streams-rise-samza-linkedin
  - Jeff Holoman, Cloudera, "Kafka Introduction," Apache Kafka ATL Meetup, 2015.
  - Michael G. Noll, Verisign, "Apache Kafka 0.8 basic training," July 2014.
  - David Tucker (Confluent), David Ostrovsky (Couchbase), "State of the Streaming Platform 2016 – What's new in Apache Kafka and Confluent platform," Nov 2016.
  - Cloudurable, "Introduction to Kafka", May 2017.
  - Cloudurable, "Kafka Tutorial," May 2017. http://cloudurable.com/blog/kafka-tutorial/index.html

-

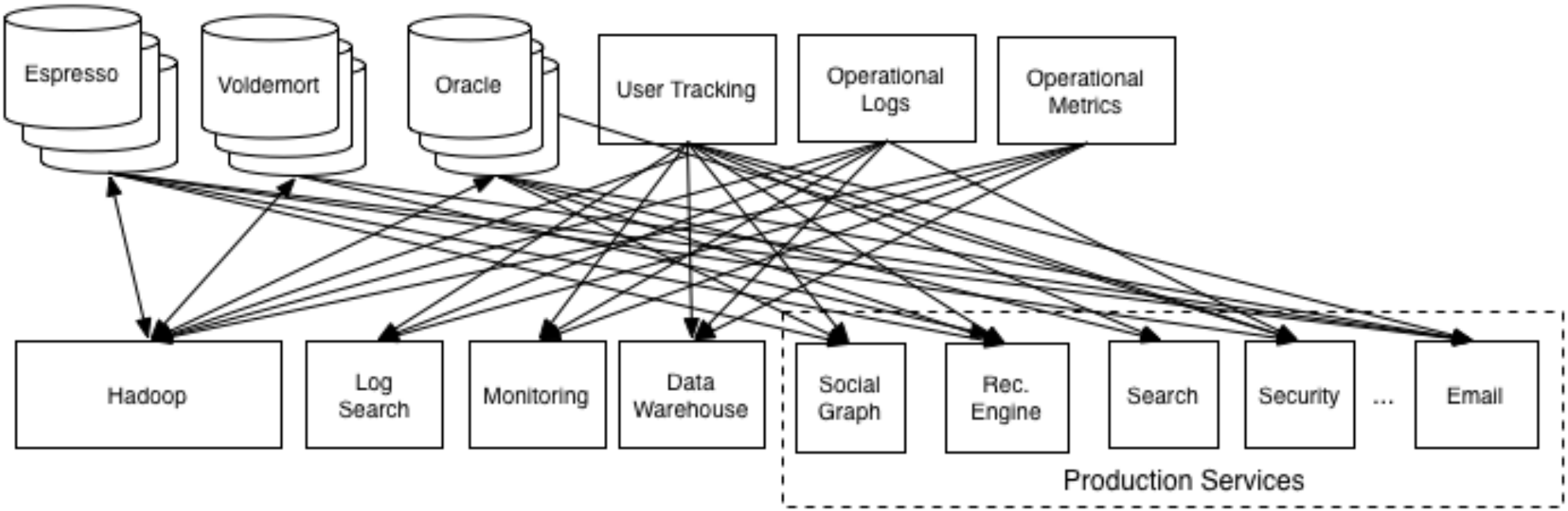
Apache Kafka

A high-throughput distributed messaging system.

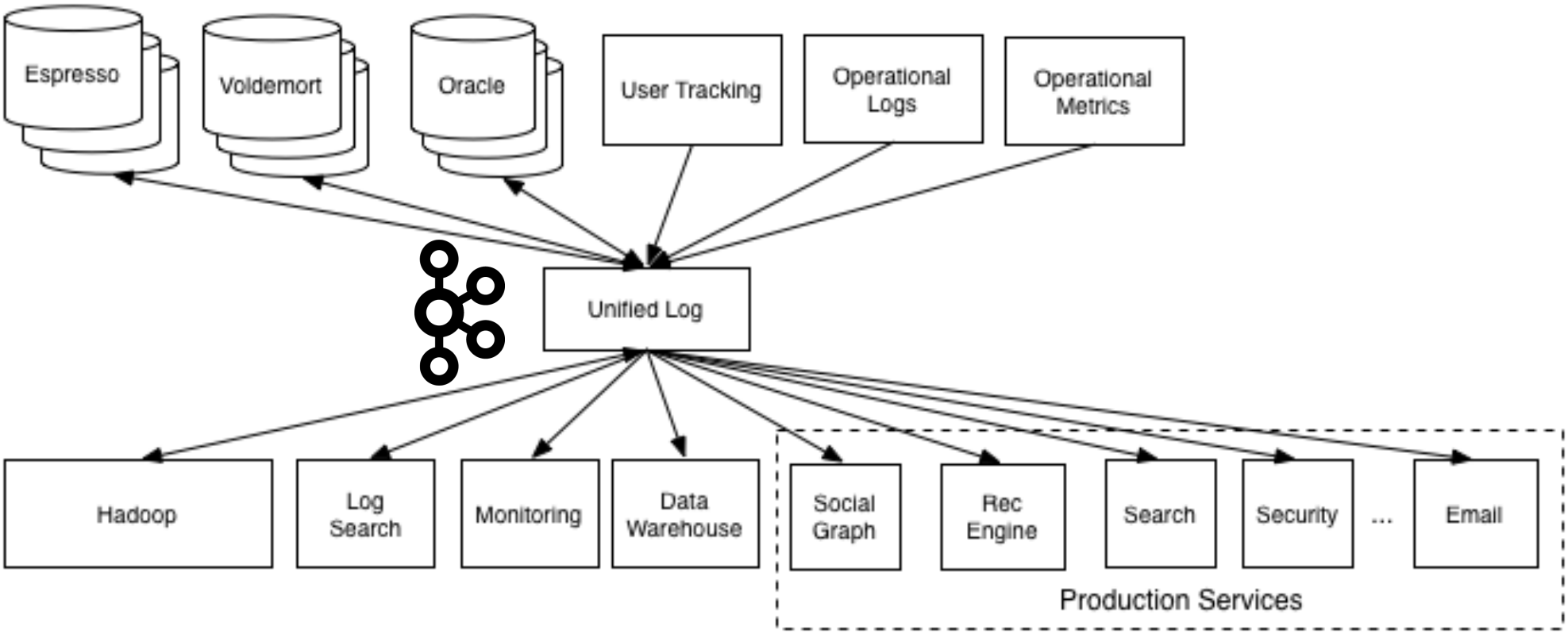
Outline

- Motivation for Kafka
- What is Kafka ?
- Real-world Use Cases
- System Architecture
- Key Concepts/ Terminologies in Kafka
- Replication Support
- Performance

Motivation: Linkedin Before Kafka



Linkedin After Kafka



Motivation for Building Kafka in Linkedin

- Goal: To provide a High-Performance, Reliable Distributed Unified Logging Platform for LinkedIn
  - Named after German author/writer Franz Kafka by one of its developers (Jay Kreps) because

    "It is a system optimized for writing".
- Originated at LinkedIn, open sourced in early 2011
  - http://kafka.apache.org/
  - In 2014, core development team of Kafka at Linkedin formed Confluent, a start-up to further develop a Kafka-centric real-time big data processing platform.
- Implemented in Scala, some Java

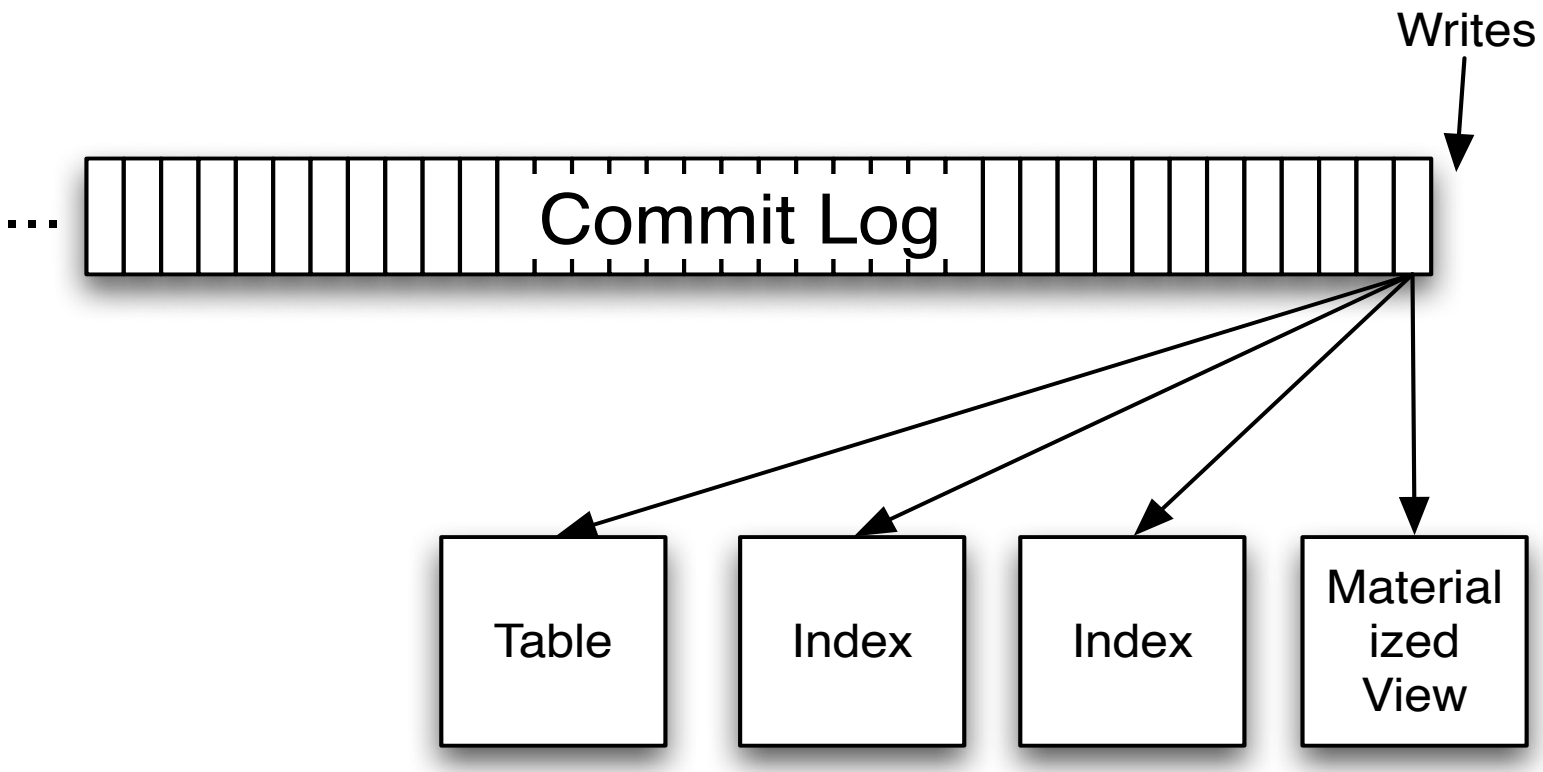
Requirements for Kafka

- Linkedin's motivation for Kafka was:
  - "A unified platform for handling all the real-time data feeds a large company might have."
- Must have:
  - High throughput to support **high volume event feeds**.
  - Support real-time processing of these feeds to create **new, derived feeds**.
  - Support large data backlogs to handle periodic ingestion from **offline systems**.
  - Support low-latency delivery to handle more traditional **messaging use cases**.
  - Guarantee **fault-tolerance** in the presence of machine failures.

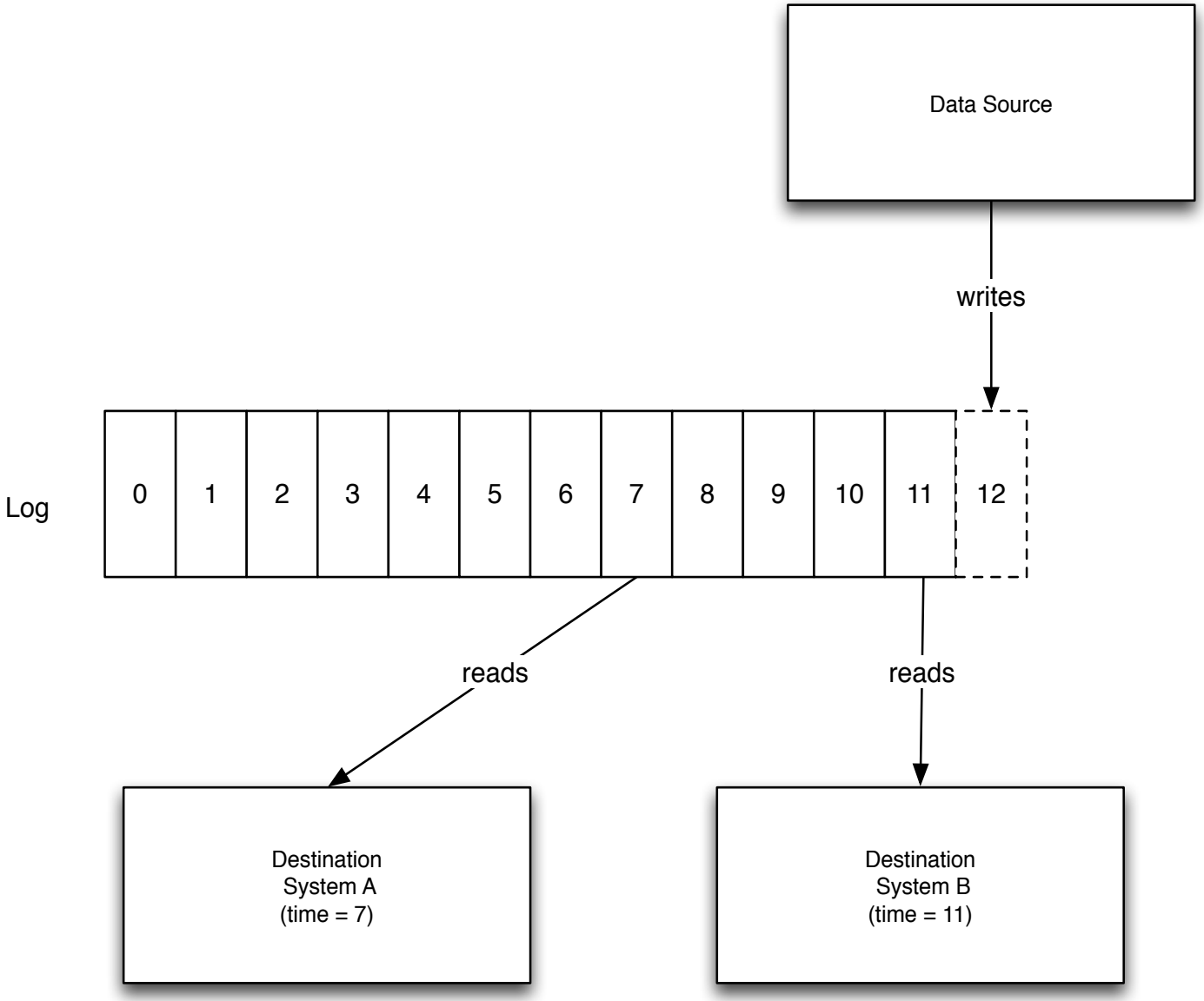
What is Kafka ?

- A Distributed Unified Logging Platform
  - Kafka maintains feeds (streams) of records (messages) organized in different categories called topics.
  - Support the “Publish and Subscribe” model for streams of records (messages)
  - Fault Tolerant Storage via Replication to multiple servers
  - Process records as they occur
  - Fast, Efficient I/O, Batching, Compression and more
  - Decouple the Producers and Consumers of those records (messages)

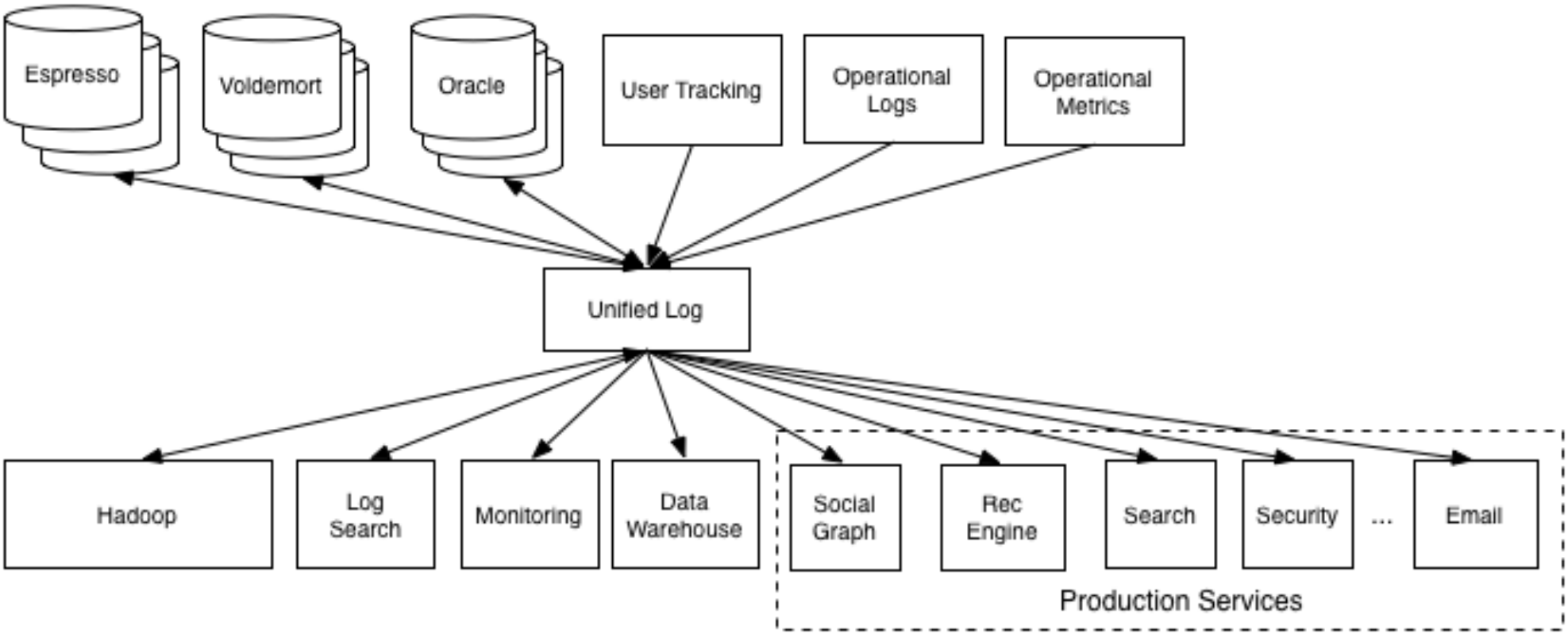
What is a commit log?



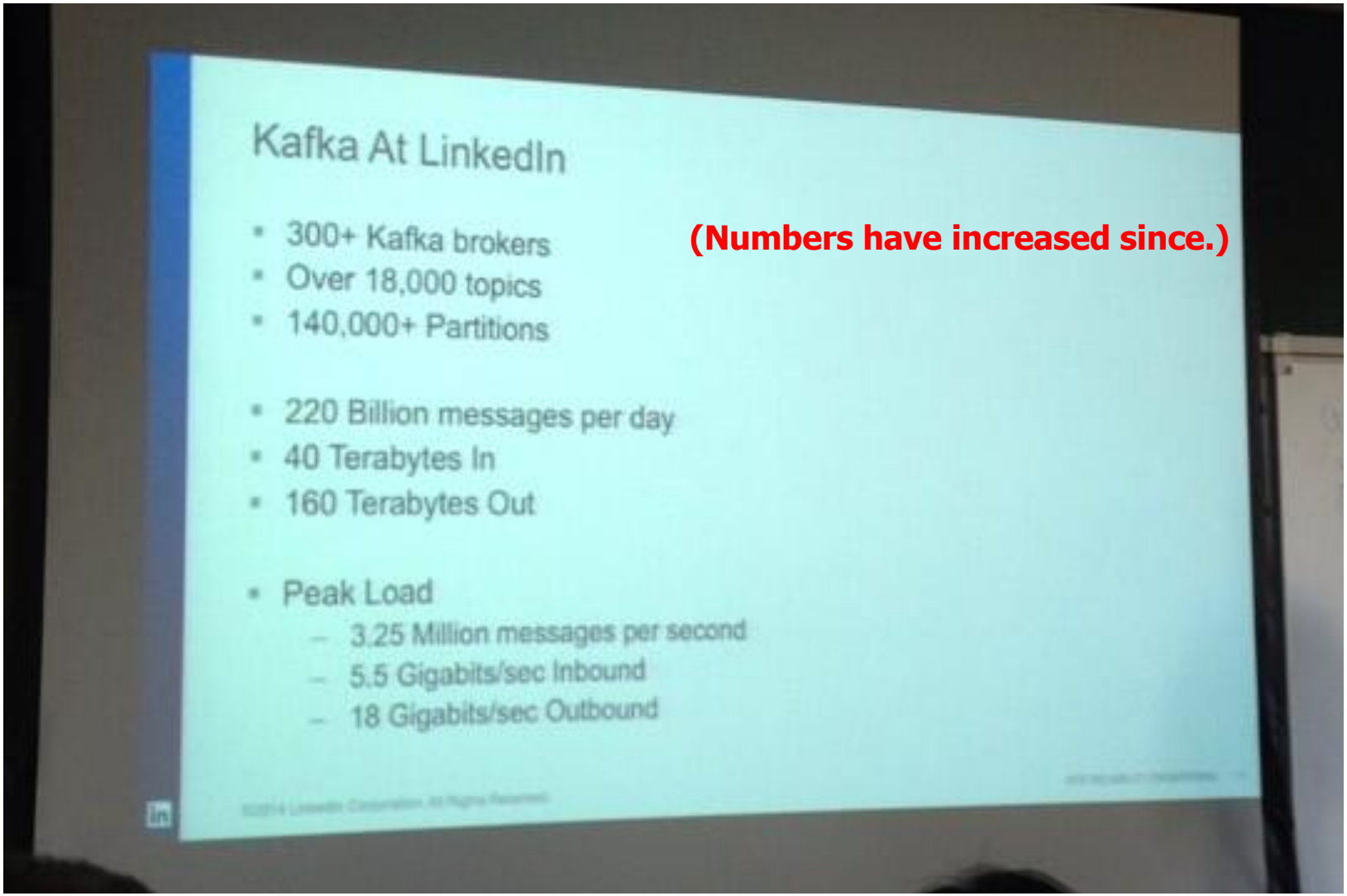
The Log as a Messaging system



Linkedin After Kafka



Kafka @ LinkedIn, 2014



https://twitter.com/SalesforceEng/status/466033231800713216/photo/1
http://www.hakkalabs.co/articles/site-reliability-engineering-linkedin-kafka-service

Kafka @ LinkedIn, 2014

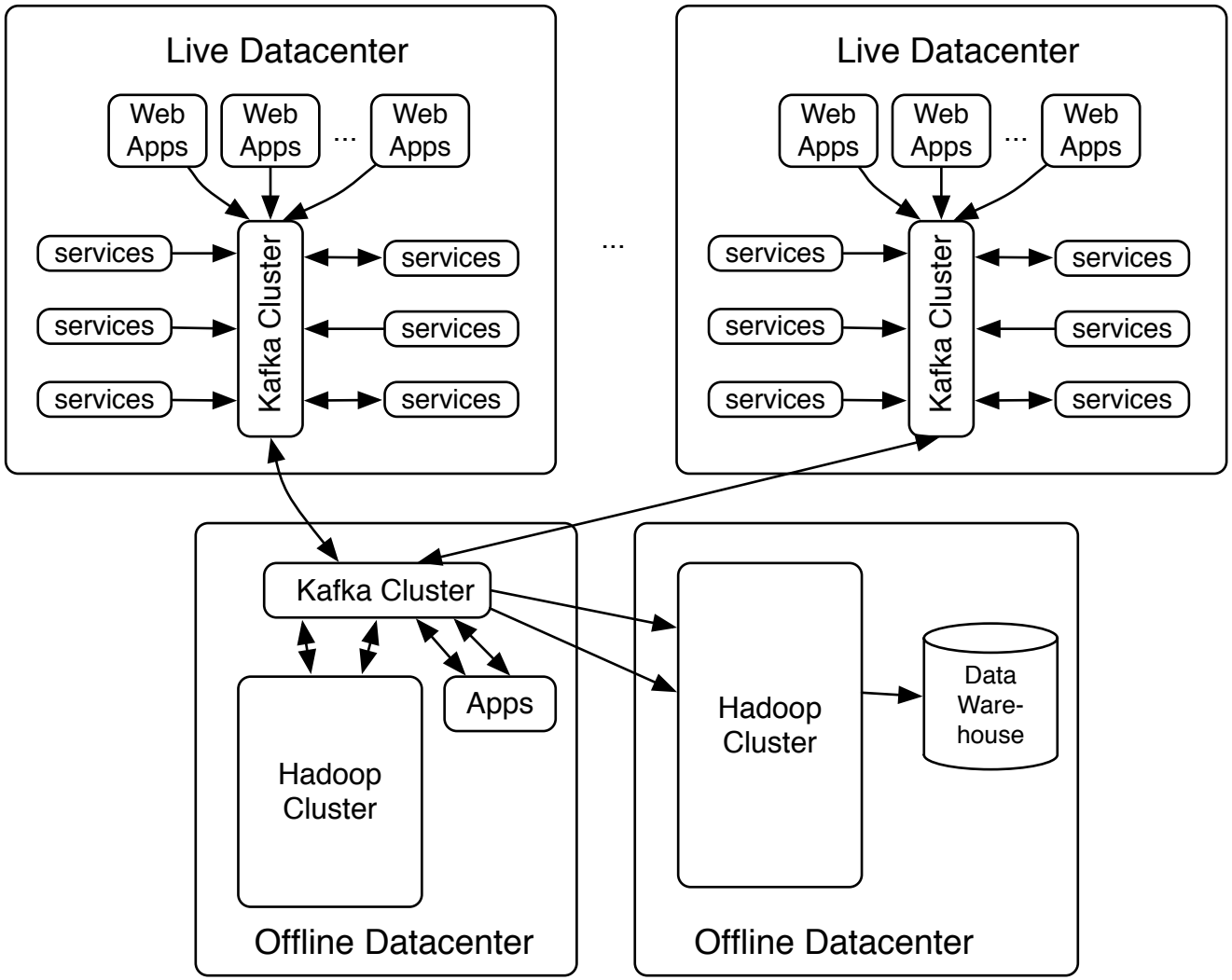
- Multiple data centers, multiple clusters
  - Mirroring between clusters / data centers

- What type of data is being transported through Kafka?
  - **Metrics**: operational telemetry data
  - **Tracking**: everything a LinkedIn.com user does
  - **Queuing**: between LinkedIn apps, e.g. for sending emails

- To transport data from LinkedIn's apps to Hadoop, and back
- In total **~ 200 billion events/day** via Kafka
  - Tens of thousands of data producers, thousands of consumers
  - 7 million events/sec (write), 35 million events/sec (read) <<< may include replicated events
  - But: LinkedIn is not even the largest Kafka user anymore as of 2014

http://www.hakkalabs.co/articles/site-reliability-engineering-linkedin-kafka-service
http://www.slideshare.net/JayKreps1/i-32858698
http://search-hadoop.com/m/4TaT4qAFQW1

Kafka Usage at LinkedIn (circa 2013)



Kafka @ LinkedIn, 2014

"For reference, here are the stats on one of LinkedIn's busiest clusters (at peak):

15 brokers
15,500 partitions (replication factor 2)
400,000 msg/s inbound
70 MB/s inbound
400 MB/s outbound"

https://kafka.apache.org/documentation.html#java

General Use Cases of Kafka

Major Role (via the original Kafka "Core components"):

- Log Aggregation
- Capture and Ingest Data into Spark/ Hadoop/ Storm/ Flink etc
- Command-Query Responsibility Segregation (CRQS), Replay, Error Recovery
- Guaranteed Distributed Commit Log for in-memory computing
- Metrics Collection and Monitoring

Supporting Role (with recent extensions from Confluent):

- Stream Processing
- Website Activity Tracking
- Real Time Analytics

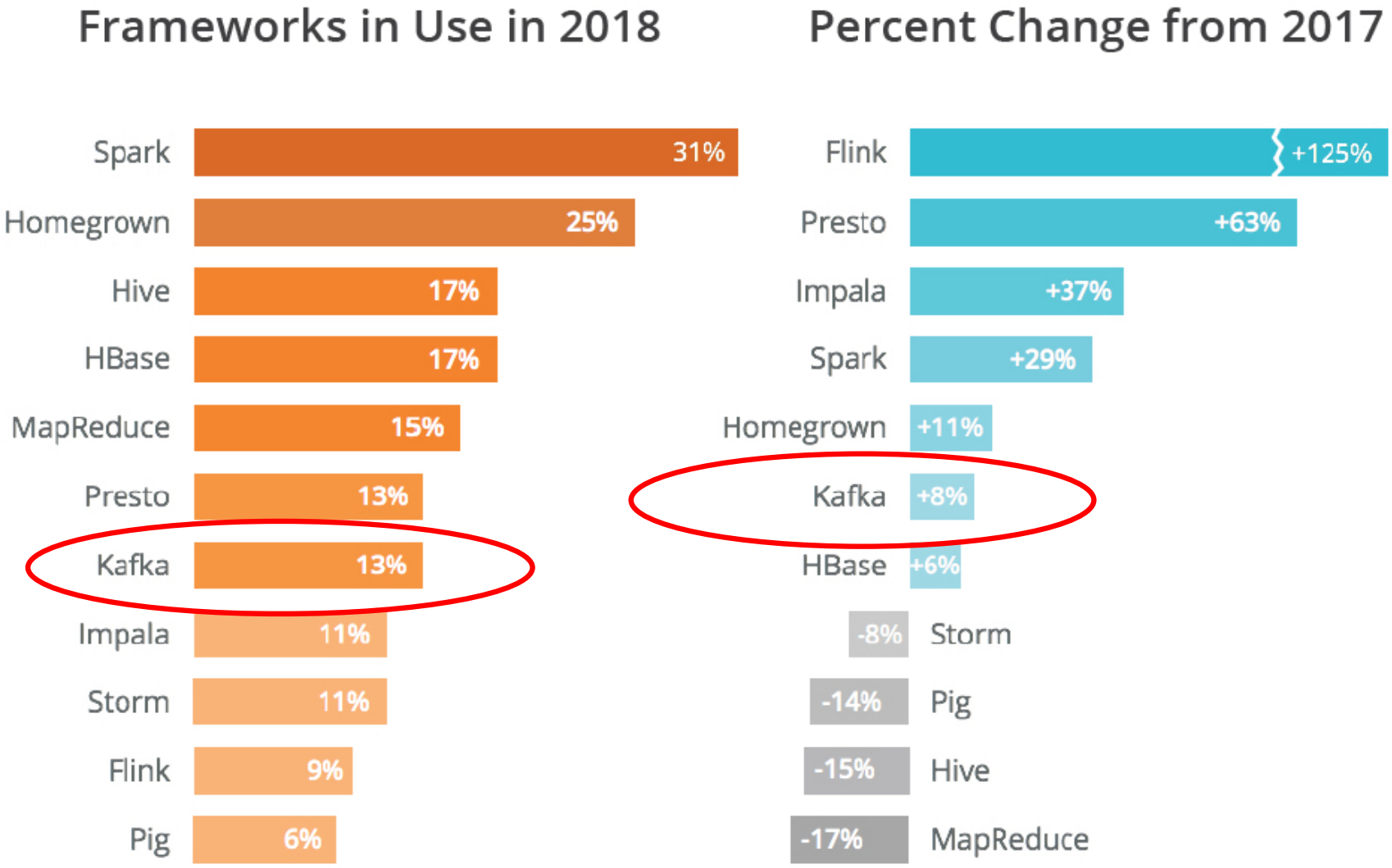
Sample Use Cases of Kafka

- Linkedin: Activity streams, Operational Metrics, data bus
  - 400 nodes, 18k topics, 220B msg/day (peak 3.2M msg/sec) (circa May 2014)
- Netflix: Real-time Monitoring and Event Processing
- Twitter: Use it with Storm for stream processing pipelines
- Spotify: Log delivery (for 4 hrs down to 10sec), Hadoop
- Loggy: Log collection and processing
- Mozilla: Telemetry data
- Square: Kafka as “data-bus” to move all system events to various Square datacenters (logs, custom events, metrics, etc). Outputs to other systems, e.g. Alert generation

Other Users of Kafka

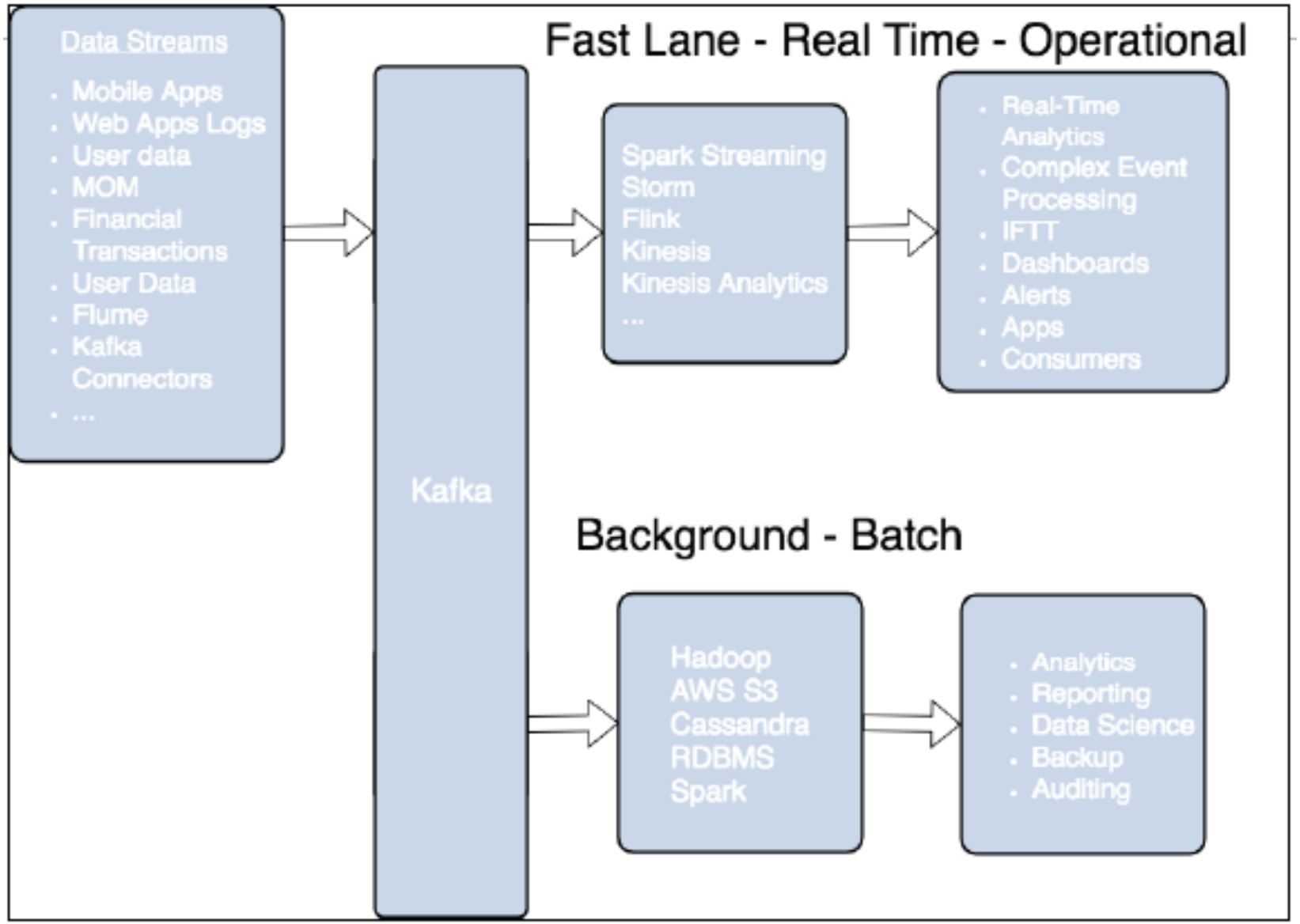
- 1/3 of all Fortunate 500 companies
  - Top 10 Travel companies ; 7 of Top 10 Banks ; 8 of Top 10 Insurance companies ; 9 of Top 10 Telecom coms.
- Airbnb, Uber, Tumbler, Goldman Sachs, PayPal, Cisco, CloudFlare, etc.
- LinkedIn, Microsoft and Netflix process 4 comma messages a day with Kafka (1,000,000,000,000) (circa 2017)

Big Data Frameworks Adoption Trends



Qubole

Typical Service Architecture with Kafka

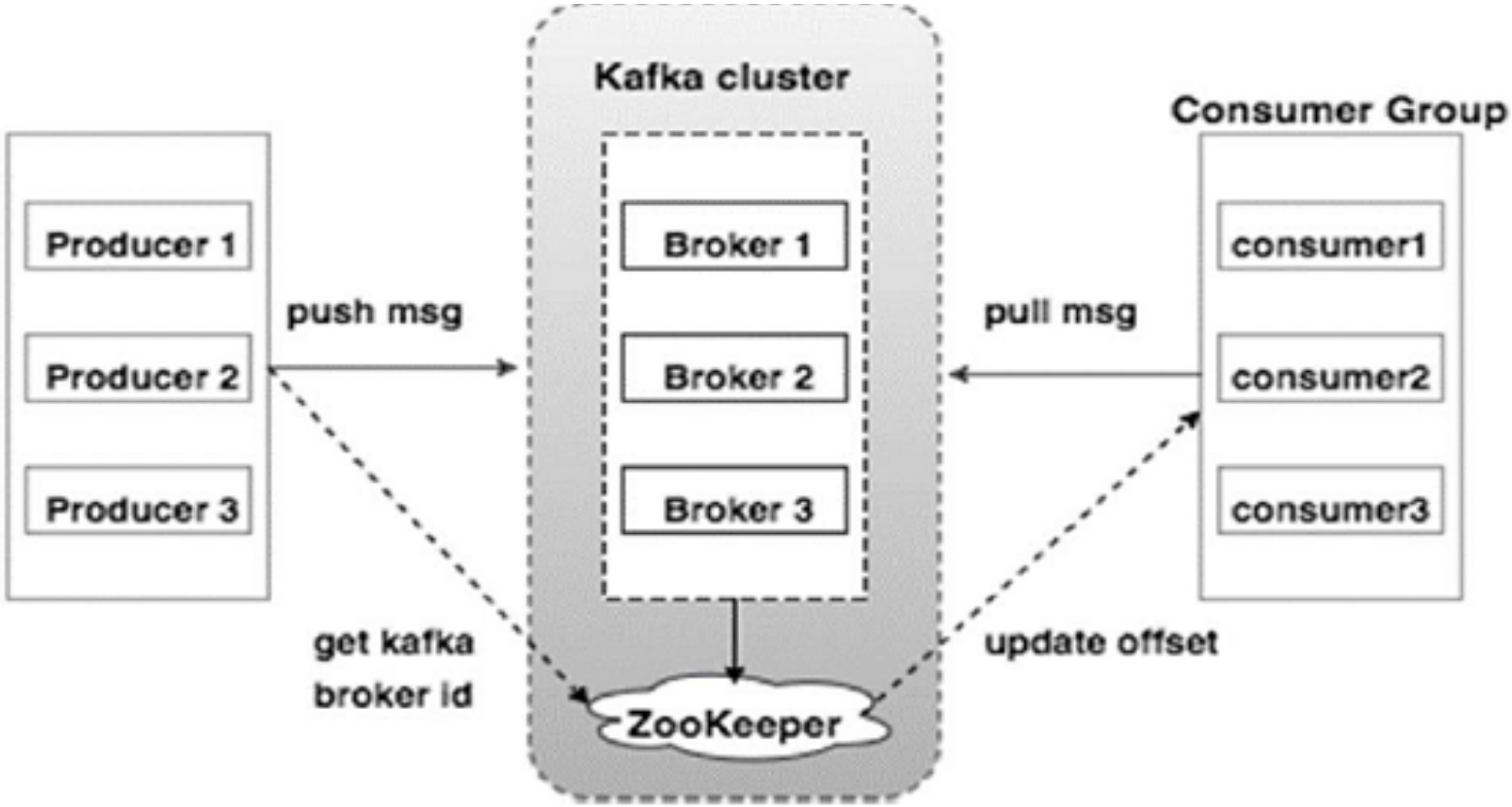


Kafka + X for processing the data?

- Kafka + **Storm** often used in combination, e.g. Twitter
- Kafka + **custom**
  - "Normal" Java multi-threaded setups
  - Akka actors with Scala or Java, e.g. Ooyala
- Additional "partners":
  - **Samza** (since Aug '13) – also by LinkedIn
  - **Spark Streaming**, part of Spark (since Feb '13)
- Kafka + **Camus** for Kafka->Hadoop ingestion
  - Camus phased out/ replaced by **Globbin**

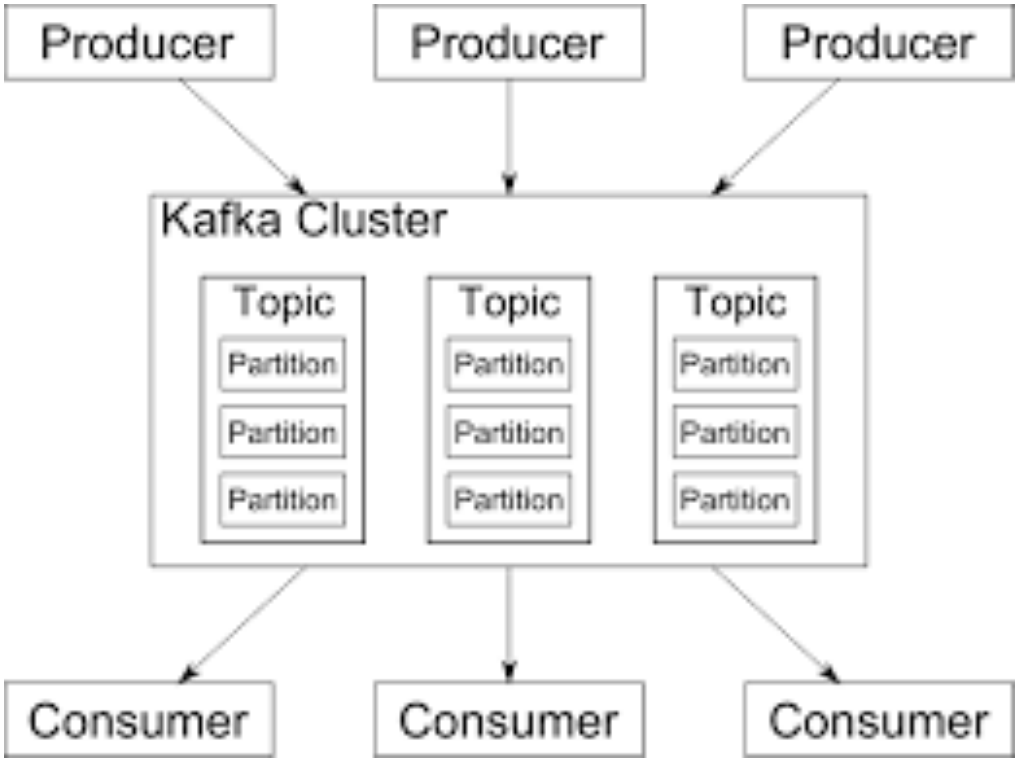
  https://cwiki.apache.org/confluence/display/KAFKA/Powered+By
- Kafka (core) + extended Open-source frameworks from Confluent e.g.
  - KSQL (Kafka SQL), Kafka Streams, Kafka Connect & Connectors, Schema Registry, REST Proxy, MQTT Proxy ;

System Architecture of Kafka (its Core Components)



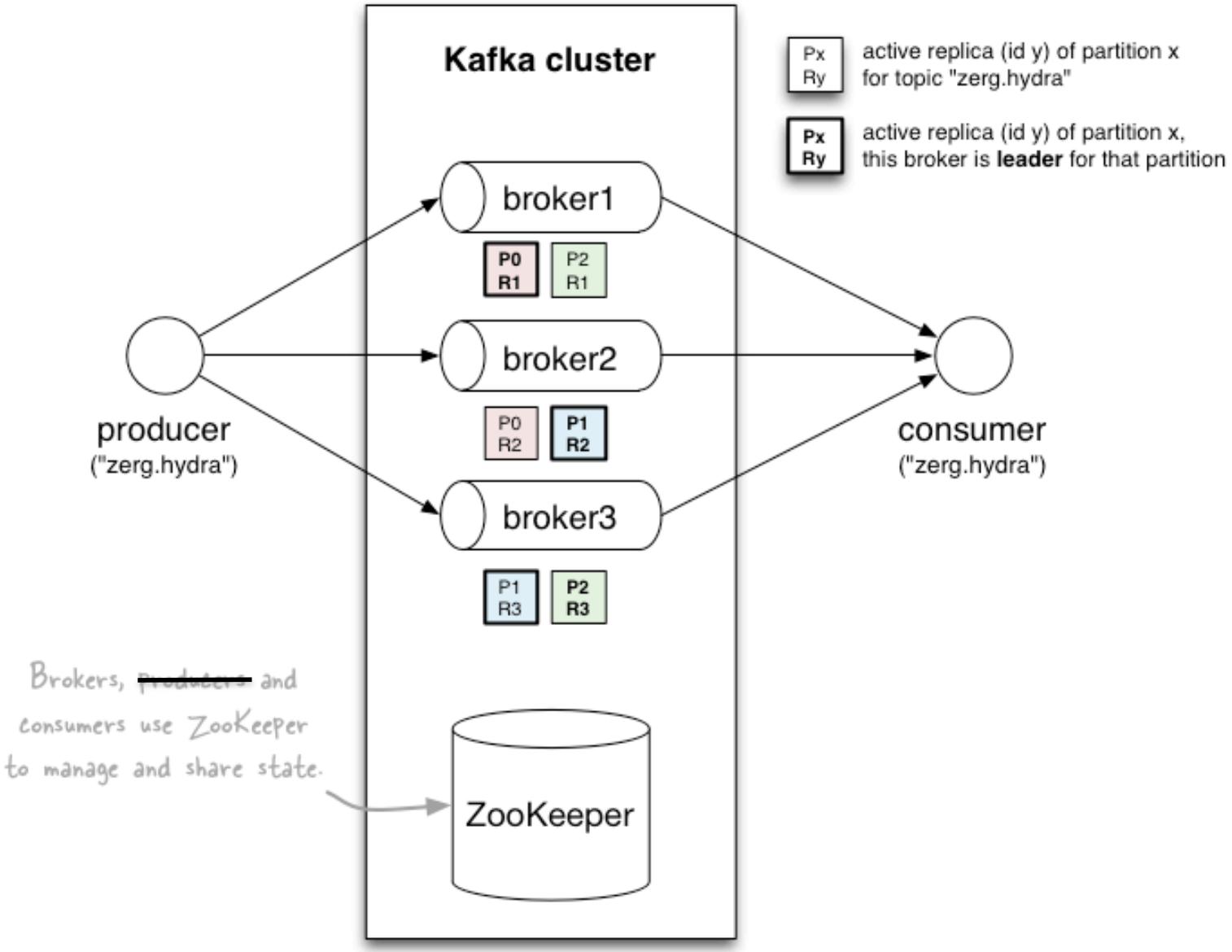
- **Broker: Kafka server that runs in a Kafka Cluster. Each Cluster has 1+ Broker ; Each Broker has a Unique Broker ID**
- **ZooKeeper: Coordinate Brokers/Cluster topology:**
  - **Stable storage for Cluster configuration**
  - **Election for Broker and Partition Leaders ; Coordinate Cluster changes**
  - **Used by Consumers to track message (reading) offsets in v0.8 [replaced by the use of Special Topics in v0.9]**

Key Concepts/ Terminologies of Kafka



- **Producers** write data to **Brokers**.
- **Consumers** read data from **Brokers**.
- Data is stored in **Topics**.
- **Topics** are split into **Partitions**, which are **Replicated**.
- **All this is distributed.**

Illustration of a Multi-Broker Kafka Cluster

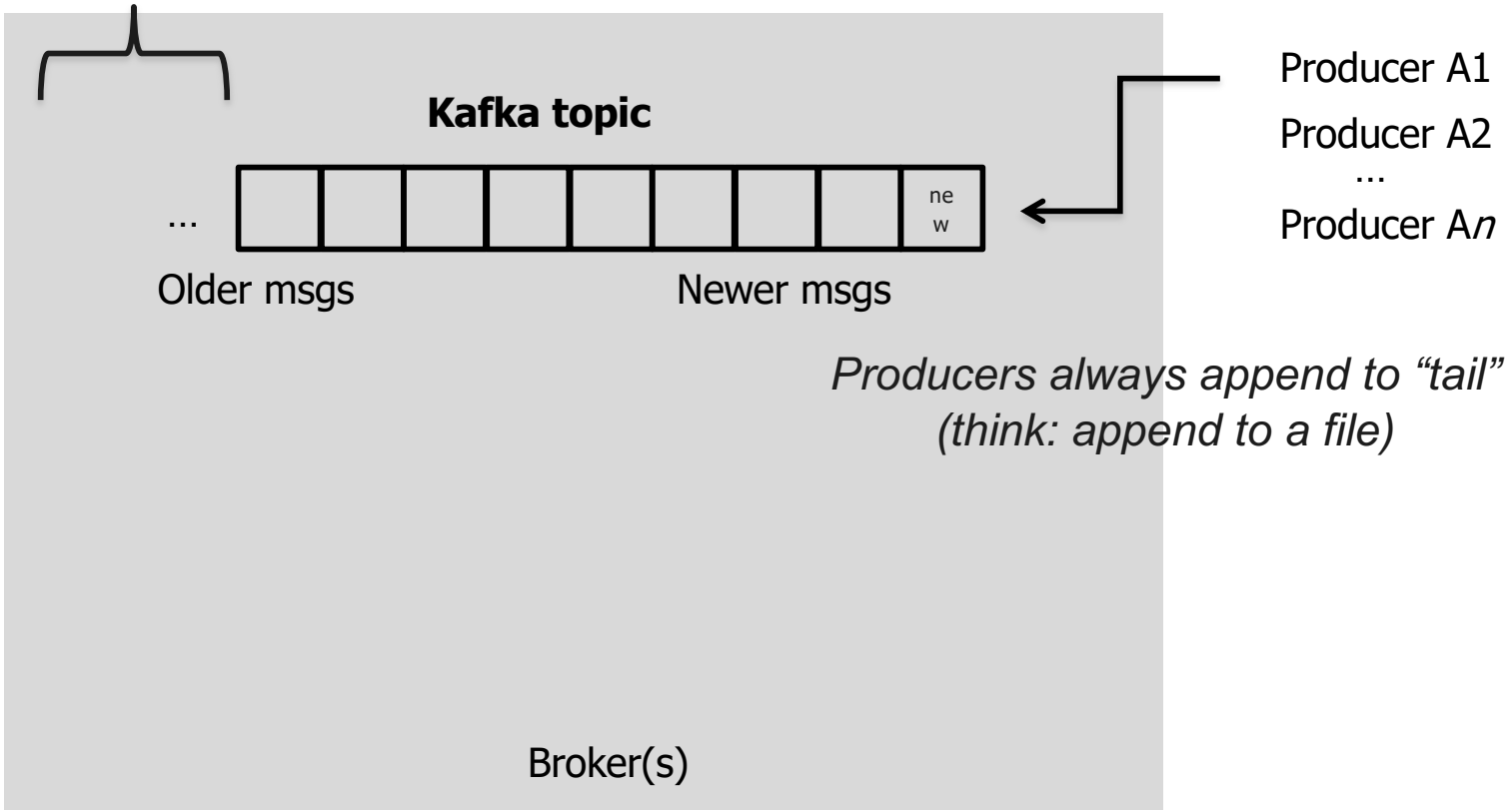


http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

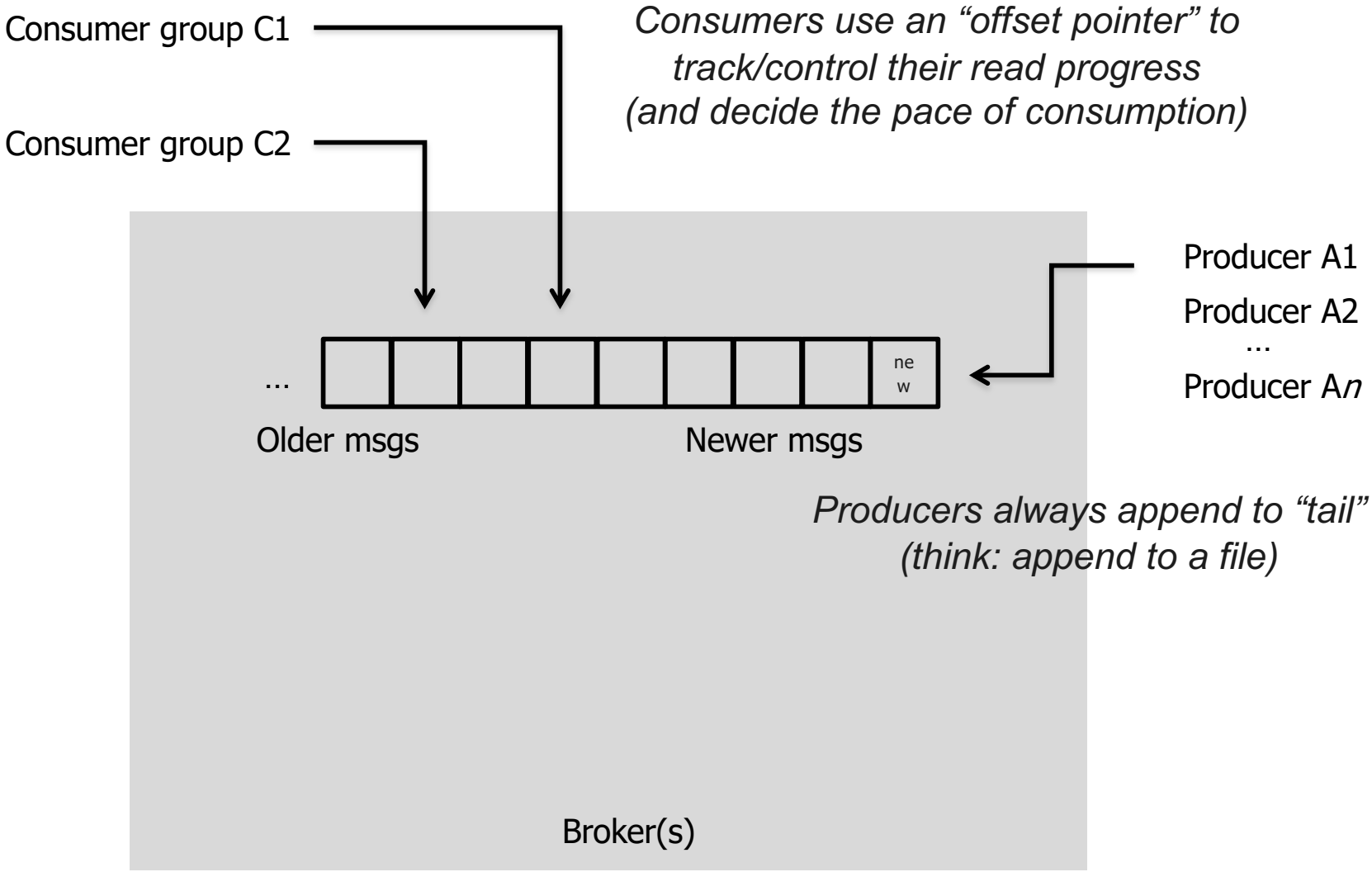
Topics

- **Topic:** feed name to which messages are published
  - Example: “zerg.hydra”

Kafka prunes “head” based on age or max size or “key”



Topics


Consumer group C1
Consumer group C2
Consumers use an "offset pointer" to track/control their read progress (and decide the pace of consumption)
Producer A1
Producer A2
...
Producer An
...
new
Older msgs
Newer msgs
Producers always append to "tail" (think: append to a file)
Broker(s)

Topics

- Creating a topic
  - CLI (following example for Kafka version 0.8x only)

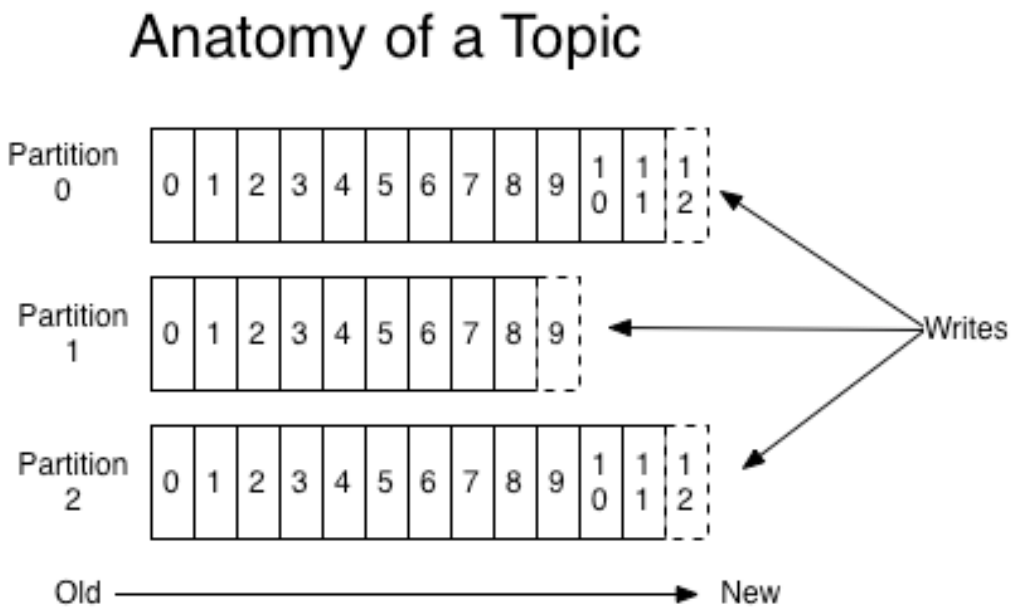
```
$ kafka-topics.sh --zookeeper zookeeper1:2181 --create --topic zerg.hydra \
    --partitions 3 --replication-factor 2 \
    --config x=y
```

  - API
    https://github.com/miguno/kafka-storm-starter

  - Auto-create via `auto.create.topics.enable = true`
- Modifying a topic
  - https://kafka.apache.org/documentation.html#basic_ops_modify_topic
- Deleting a topic

Partitions

- A topic consists of **partitions.**
- Partition: **ordered + immutable** sequence of messages that is continually appended to

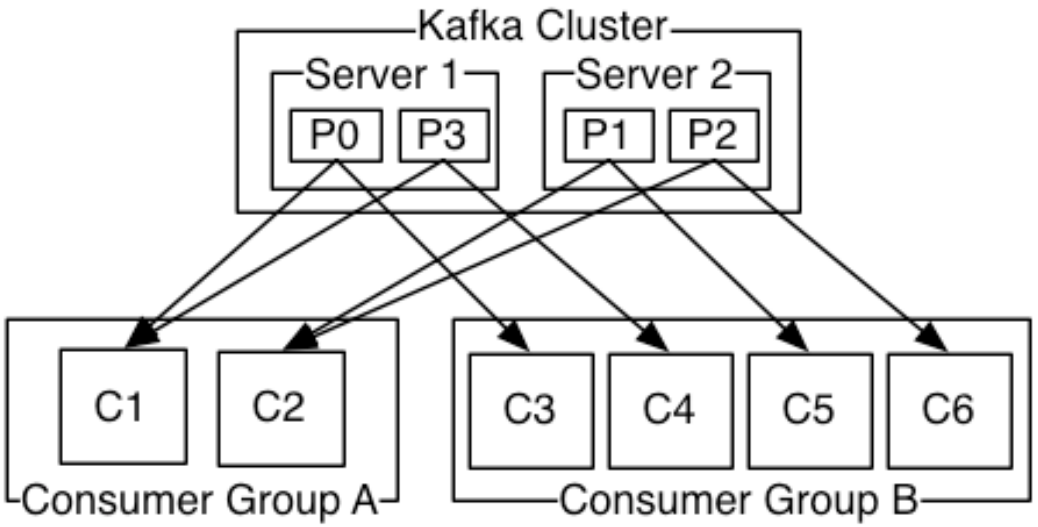


Producers

- Producers publish data to the **topics** of their choice.
- Producer is responsible for choosing which message to assign to which **partition** within the topic.
- This can be done in a **round-robin** fashion simply to balance load or it can be done according to some **semantic partition function**

Partitions

- #partitions of a topic is configurable
- #partitions determines **max** # of consumers (threads) in each Consumer Group
  - Parallelism (same idea as sharding in database)
  - Cf. parallelism of Storm's KafkaSpout via `builder.setSpout(,,N)`



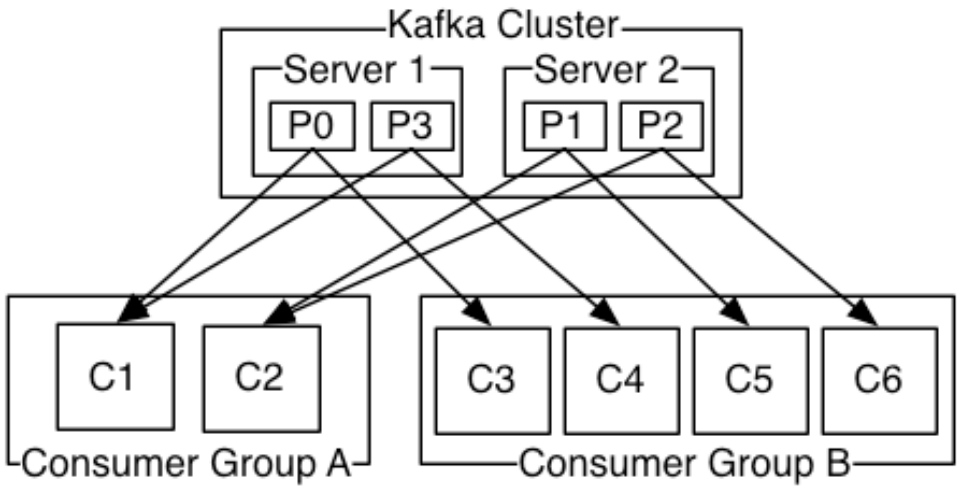
- Consumer group A, with 2 consumers, reads from a 4-partition topic
- Consumer group B, with 4 consumers, reads from the same topic

Reading data from Kafka

Consumer “groups”

- Allow multi-threaded and/or multi-machine consumption from Kafka topics.
- Consumers “join” a group by using the same `group.id`
- Kafka guarantees a message is only ever read by a single consumer in a group => processing order guarantee for messages within a partition.
  - Kafka assigns the partitions of a topic to the consumers in a group so that each partition is consumed by exactly one consumer in the group.
  - Maximum parallelism of a consumer group:

    #consumers (in the group) =< **#partitions**



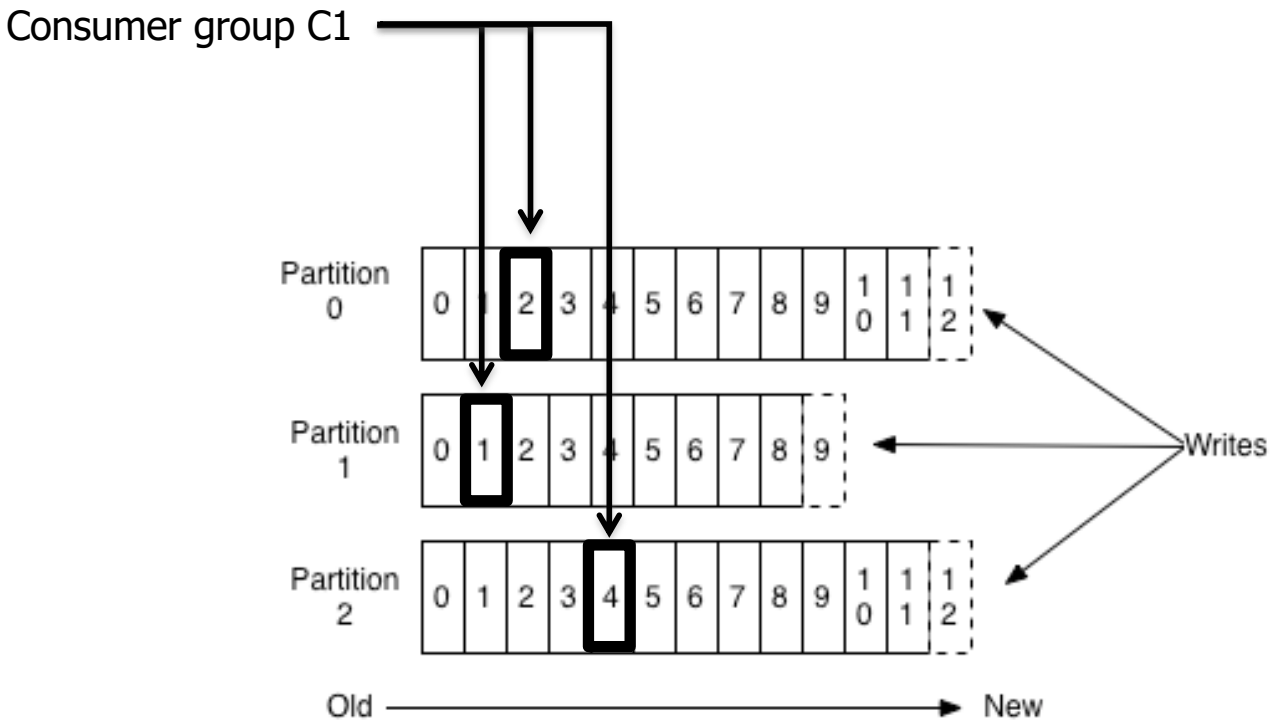
Consumers

- Kafka offers a single consumer abstraction that generalizes both queuing and publish-subscribe mode.
- Consumers label themselves with a consumer group name, and each message published to a topic is delivered to **one consumer instance** within each subscribing consumer group.
- Kafka is able to provide both **ordering guarantees** and **load balancing** over a pool of consumer processes, by guarantee each partition is consumed by *exactly one consumer in the group*.

  => There cannot be more consumer instances in a consumer group than partitions.
- Kafka only provides a total order over messages **within** a partition, not between different partitions in a topic.

Partition offsets

- **Offset**: messages in the partitions are each assigned a unique (per partition) and sequential id called the *offset*
  - Consumers track their pointers via *(offset, partition, topic)* tuples ; This offset is controlled by the Consumer



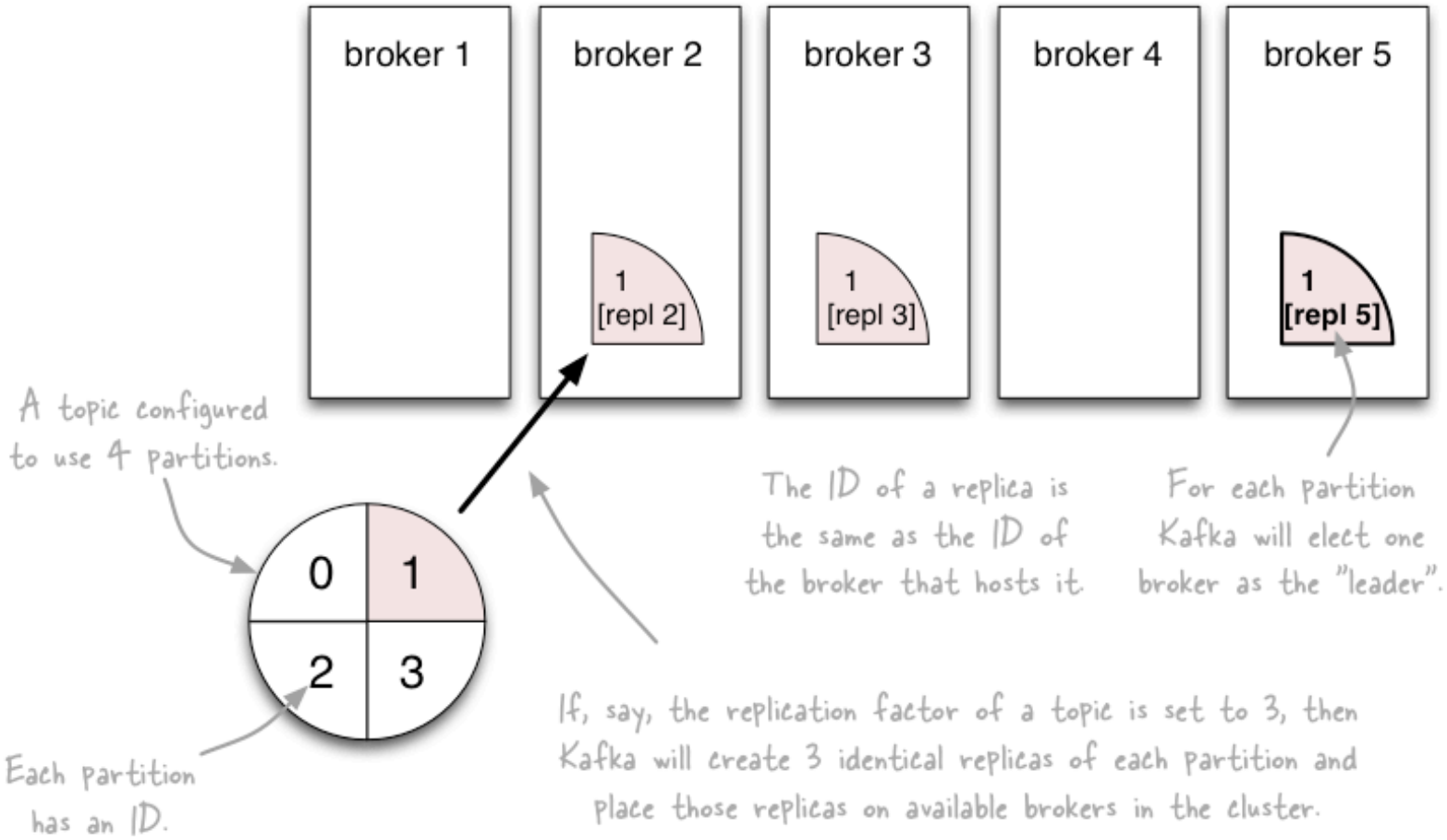
Guarantees supported by Kafka

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent.
- A consumer instance sees messages in the order they are stored in the log.

Distribution and Replication of a Partition

- The partitions of the log are distributed over the servers (brokers) in the Kafka cluster
- Each partition is replicated across a **configurable number** of servers for fault tolerance.
- Each partition has one server which acts as the "**leader**" and zero or more servers which act as "**followers**".
- If the leader fails, one of the followers will automatically become the new leader.
- Each server acts as a leader for some of its partitions and a follower for others so load is well **balanced within the cluster**

Topics vs. Partitions vs. Replicas



http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

Distribution and Replication of a Partition (cont'd)

- The leader of a partition handles all read and write requests for the partition while the followers (Replicas) ONLY passively replicate the leader.
  - Replicas exist solely to prevent data loss.
  - Replicas are never read from, never written to.
    - They do NOT help to increase producer or consumer parallelism!
  - Kafka tolerates *(numReplicas - 1)* dead brokers before losing data
    - LinkedIn: numReplicas == 2 → 1 broker can die

Inspecting the current state of a topic

- `--describe` the topic

```
$ kafka-topics.sh --zookeeper zookeeper1:2181 --describe --topic zerg.hydra
Topic:zerg2.hydra PartitionCount:3 ReplicationFactor:2 Configs:
  Topic: zerg2.hydra Partition: 0 Leader: 1 Replicas: 1,0 Isr: 1,0
  Topic: zerg2.hydra Partition: 1 Leader: 0 Replicas: 0,1 Isr: 0,1
  Topic: zerg2.hydra Partition: 2 Leader: 1 Replicas: 1,0 Isr: 1,0
```

  - Leader: brokerID of the currently elected leader broker
    - Replica ID's = broker ID's
  - ISR = "in-sync replica", replicas that are in sync with the leader
- In this example:
  - Broker 0 is leader for partition 1.
  - Broker 1 is leader for partitions 0 and 2.
  - All replicas are in-sync with their respective leader partitions.

Message Delivery Semantics

- **At Least Once** (default)
  - Messages are never lost but may be redelivered
- **At Most Once**
  - Messages can be lost but never redelivered
- **Exactly Once**

Achieving Exactly Once Semantics

- Must consider 2 components
  - Durability guarantees when publishing a message (by the Producer)
  - Durability guarantees when reading a message (by a Consumer)
- For the Producer
  - What happens when a produce request was sent but a network error returned before an ACK ?
  - Use a Single writer per partition and check the latest committed value after network errors
- For a Consumer
  - Include a Unique ID (e.g. UUID) and de-duplicate records
  - Consider storing offsets with data

Kafka Record Retention

- Kafka cluster retains all published records (as long as there is enough storage)
  - Time-based – Configurable Retention period, e.g. 3 days, 2 weeks or 1 month
  - Size-based – Configurable based on size
  - Compaction – keeps latest records
- Records written to Kafka are persisted to disk and replicated to other servers for fault-tolerance
- Records are available for consumption until discarded by time, size or compaction
- Consumption speed not impacted by size as Kafka always write to the end of the (topic) log
- Record (message) Producers *can* wait on Acknowledgement
  - s.t. Write not complete until properly replicated

Writing Data to Kafka

For more detail tutorials, see:
http://cloudurable.com/blog/kafka-tutorial-kafka-producer/index.html

Writing data to Kafka

- Use Kafka “Producers” to write data to Kafka brokers.
  - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.
- Two modes of writing: “async” and “sync”
  - Different semantics
  - Sync Producer “send” call will block !
  - Async Producer is preferred to achieve high throughput (non-blocking)
- Important Configuration settings for Producers:

`client.id`	identifies producer app, e.g. in system logs
`producer.type`	async or sync
`acks`	acking semantics
`serializer.class`	configure encoder, e.g. using Avro
`Bootstrap.servers`	For bootstrapping from a list of well-known brokers

Async Producer

- Sends messages in background = no blocking in client.
- Provides more powerful batching of messages.
- Wraps a sync producer, or rather a pool of them.
- Communication from async->sync producer happens via a queue.

Caveats

- Async producer may drop messages if its queue is full.
  - Solution 1: Don't push data to producer faster than it is able to send to brokers.
  - Solution 2: Queue full == need more brokers, add them now! Use this solution in favor of solution 3 particularly if your producer cannot block (async producers).
  - Solution 3: Set queue.enqueue.timeout.ms to -1 (default). Now the producer will block indefinitely and will never willingly drop a message.
  - Solution 4: Increase queue.buffering.max.messages (default: 10,000).

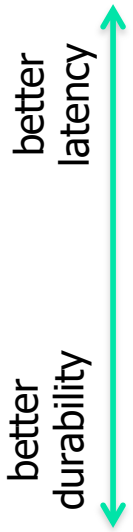
Message ACKing for Producers

- Background:
  - In Kafka, a message is considered *committed* when “any required” ISR (in-sync replicas) for that partition have applied it to their data log.
  - Message acking is about conveying this “Yes, committed!” information back from the brokers to the producer client.
  - Exact meaning of “any required” is defined by `request.required.acks`.

- Only **producers** must configure acking
  - Exact behavior is configured via **`request.required.acks`**, which determines when a produce request is considered completed.
  - Allows you to trade **latency (speed)** <-> **durability (data safety)**.
  - Consumers: Acking and how you configured it on the side of producers do not matter to consumers because only committed messages are ever given out to consumers. They don’t need to worry about potentially seeing a message that could be lost if the leader fails.

Message ACKing (cont'd)

- Typical values of `request.required.acks`



  - **`0`**: producer never waits for an ack from the broker.
    - Gives **the lowest latency** but the weakest durability guarantees.
  - **`1`**: producer gets an ack after the leader replica has received the data.
    - Gives better durability as the we wait until the lead broker acks the request. Only msgs that were written to the now-dead leader but not yet replicated will be lost.
  - **`-1`**: producer gets an ack after *all* ISR have received the data.
    - Gives **the best durability** as Kafka guarantees that no data will be lost as long as at least one ISR remains.

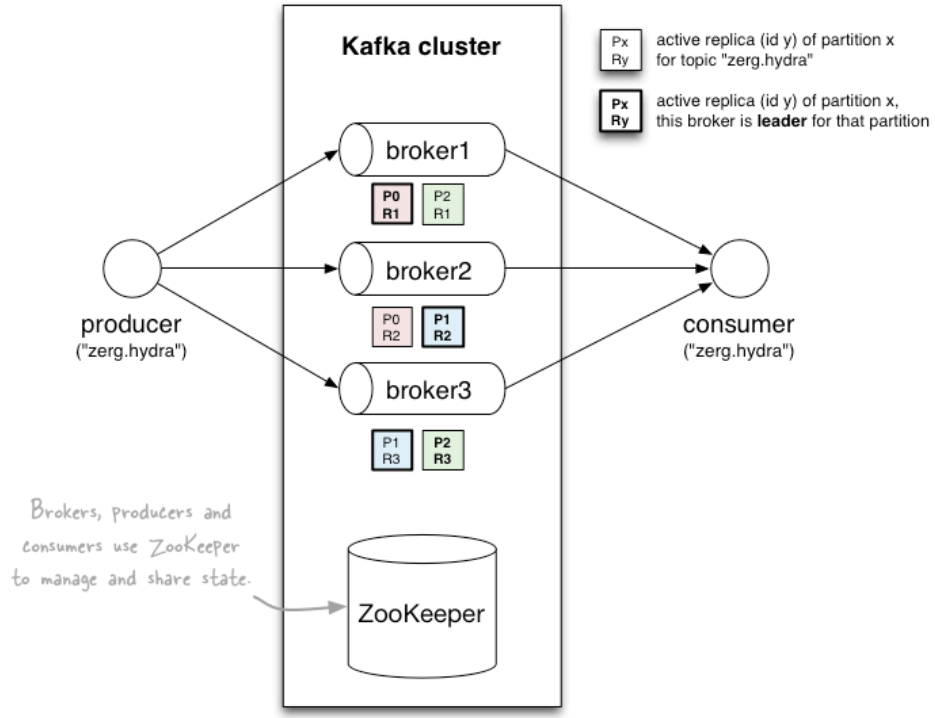
- Beware of interplay with `request.timeout.ms`!
  - "The amount of time the broker will wait trying to meet the `request.required.acks` requirement before sending back an error to the client."
  - Caveat: Message may be committed even when broker sends timeout error to client (e.g. because not all ISR ack'ed in time). One reason for this is that the producer acknowledgement is independent of the leader-follower replication, and ISR's send their acks to the leader, the latter of which will reply to the client.

Sample Java code for a Kafka Producer

```
package de.predic8.h_performance;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import javax.json.Json;
import javax.json.JsonObject;
import java.util.Properties;
import static java.lang.Math.random;
import static java.lang.Math.round;
import static org.apache.kafka.clients.producer.ProducerConfig.*;

public class PerformanceProducer {

    public static void main(String[] args) throws InterruptedException {

        Properties props = new Properties();
        props.put(BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ACKS_CONFIG, "all");
        props.put(RETRIES_CONFIG, 0);
        props.put(BATCH_SIZE_CONFIG, 32000);
        props.put(LINGER_MS_CONFIG, 100);
        props.put(BUFFER_MEMORY_CONFIG, 33554432);
        props.put(KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.LongSerializer");
        props.put(VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.LongSerializer");

        Producer<Long, Long> producer = new KafkaProducer<>(props);

        long t1 = System.currentTimeMillis();

        long i = 0;
        for(; i < 1000000; i++) {

            producer.send(new ProducerRecord<>("produktion", i, i));
    }
        producer.send(new ProducerRecord<Long,Long>("production", (long) -1, (long)-1));
        System.out.println("fertig " + i  + " Nachrichten in " + (System.currentTimeMillis() - t1 + " ms"));

        producer.close();
  }
}
```

Write operations behind the scenes

- When writing to a topic in Kafka, producers write directly to the partition leaders (brokers) of that topic
  - Remember: Writes always go to the leader ISR of a partition!



- This raises two questions:
  - How to know the “right” partition for a given topic?
  - How to know the current leader broker/replica of a partition?

1) How to know the “right” partition when sending?

- In Kafka, a producer – i.e. the ***client*** – decides to which target partition a message will be sent.
  - Can be random ~ load balancing across receiving brokers.
  - Can be semantic based on message “key”, e.g. by user ID or domain name.
    - Here, Kafka guarantees that all data for the same key will go to the same partition, so consumers can make locality assumptions.

```
1  // Java example.  Topic is "zerg.hydra".
2  KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myValue");
3  KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myKey", "myValue");
```

- But there’s one catch with line 2 (i.e. no key)

Keyed vs. non-keyed messages in Kafka

- If a key **is not** specified:

```
2   KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myValue");
```

  - Producer will *ignore* any configured partitioner.
  - It will pick a random partition from the list of *available* partitions **and stick to it** for some time before switching to another one = NOT round robin or similar!
    - Why? To reduce number of open sockets in large Kafka deployments (KAFKA-1017).
    - Default: 10mins, cf. `topic.metadata.refresh.interval.ms`
    - See implementation in `DefaultEventHandler#getPartition()`
  - If there are fewer producers than partitions at a given point of time, some partitions may not receive any data. How to fix if needed?
    - Try to reduce the metadata refresh interval `topic.metadata.refresh.interval.ms`
    - Specify a message key and a customized random partitioner.
  - In practice it is not trivial to implement a correct "random" partitioner in Kafka 0.8.
    - Partitioner interface in Kafka 0.8 lacks sufficient information to let a partitioner select a random and *available* partition. Same issue with DefaultPartitioner.

Keyed vs. non-keyed messages in Kafka

- If a key **is** specified:

```
3  KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myKey", "myValue");
```

  - Key is retained as part of the msg, will be stored in the broker.
  - One can design a partition function to route the msg based on key.
  - The *default partitioner* assigns messages to a partition based on their key hashes, via `key.hashCode % numPartitions`.
  - Caveat:
    - If you specify a key for a message but do not explicitly wire in a custom partitioner via `partitioner.class`, your producer will use the default partitioner.
    - So without a custom partitioner, messages with the same key will still end up in the same partition! (cf. default partitioner's behavior above)

2) How to know the current leader of a partition?

- Producers: broker discovery aka bootstrapping
  - Producers don't talk to ZooKeeper, so it's not through ZK.
  - Broker discovery is achieved by providing producers with a "bootstrapping" broker list, cf. `metadata.broker.list`
    - These brokers inform the producer about all alive brokers and where to find current partition leaders. The bootstrap brokers do use ZK for that.
- Impacts on failure handling
  - In Kafka 0.8 the bootstrap list is static/immutable during producer run-time. This has limitations and problems as shown in next slide.
  - The bootstrap approach has been improved in Kafka 0.9. This change makes the life of Ops easier.

Bootstrapping in Kafka

- Scenario: N=5 brokers total, 2 of which are for bootstrap

```
1  Properties props = new Properties();
2  props.put("metadata.broker.list", "broker1:9092,broker2:9092");
3  ProducerConfig config = new ProducerConfig(props);
```

- Do's:
  - Take down one bootstrap broker (e.g. broker2), repair it, and bring it back.
  - In terms of impacts on *broker discovery*, you can do whatever you want to brokers 3-5.
- Don'ts:
  - Stop all bootstrap brokers 1+2. If you do, the producer stops working!
- To improve operational flexibility, use VIP's or similar for values in `metadata.broker.list`.

Reading Data from Kafka

For more detail tutorials, see:
http://cloudurable.com/blog/kafka-tutorial-kafka-consumer/index.html

Reading data from Kafka

- You use Kafka "consumers" to read data from Kafka brokers.
  - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.
  - The Kafka project only provides the JVM implementation.
    - Has risk that a new Kafka release will break non-JVM clients.

- Three API options for JVM users:
  1. High-level consumer API <<< in most cases you want to use this one!
  2. Simple consumer API
  3. Hadoop consumer API

- Most noteworthy: The "simple" API is anything but simple. ☺
  - Prefer to use the high-level consumer API if it meets your needs (it should).
  - Counter-example: Kafka spout in Storm 0.9.2 uses simple consumer API to integrate well with Storm's model of guaranteed message processing.

Reading data from Kafka

- Consumers *pull* from Kafka (there's no push)
  - Allows consumers to control their pace of consumption.
  - Allows to design downstream apps for **average** load, not peak load (cf. Loggly talk)
- Consumers' responsibility to track their read positions i.e. offsets
  - High-level consumer API: takes care of this for you, stores offsets in ZooKeeper
  - Simple consumer API: nothing provided, it's totally up to you (the programmer)
  - What does this offset management allow you to do?
    - Consumers can deliberately rewind "in time" (up to the point where Kafka prunes), e.g. to replay older messages.
      - Cf. Kafka spout in Storm 0.9.2.
    - Consumers can decide to only read a specific subset of partitions for a given topic.
      - Cf. Loggly's setup of (down)sampling a production Kafka topic to a manageable volume for testing
    - Run offline, batch ingestion tools that write (say) from Kafka to Hadoop HDFS every hour.
      - Cf. LinkedIn Camus, Pinterest Secor

Reading data from Kafka

- Important consumer configuration settings

`group.id`	assigns an individual consumer to a "group"
`zookeeper.connect`	to discover brokers/topics/etc., and to store consumer state (e.g. when using the high-level consumer API)
`fetch.message.max.bytes`	number of message bytes to (attempt to) fetch for each partition; must be >= broker's `message.max.bytes`

Sample Java code for a Kafka Consumer

```
package com.cloudurable.kafka;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.common.serialization.LongDeserializer;
import org.apache.kafka.common.serialization.StringDeserializer;


import java.util.Collections;
import java.util.Properties;

public class KafkaConsumerExample {

    private final static String TOPIC = "my-example-topic";
    private final static String BOOTSTRAP_SERVERS =
            "localhost:9092,localhost:9093,localhost:9094";
    …
}
public class KafkaConsumerExample {

  private static Consumer<Long, String> createConsumer() {
      final Properties props = new Properties();
      props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                                BOOTSTRAP_SERVERS);
      props.put(ConsumerConfig.GROUP_ID_CONFIG,
                                "KafkaExampleConsumer");
      props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
              LongDeserializer.class.getName());
      props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
              StringDeserializer.class.getName());

      // Create the consumer using props.
      final Consumer<Long, String> consumer =
                                new KafkaConsumer<>(props);

      // Subscribe to the topic.
      consumer.subscribe(Collections.singletonList(TOPIC));
      return consumer;
  }
```

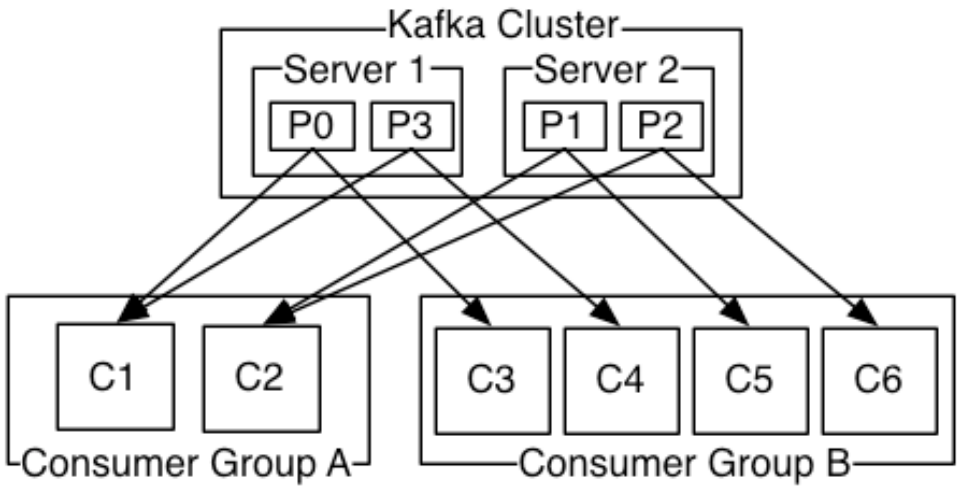
```
…
    static void runConsumer() throws InterruptedException {
        final Consumer<Long, String> consumer = createConsumer();

        final int giveUp = 100;   int noRecordsCount = 0;

        while (true) {
            final ConsumerRecords<Long, String> consumerRecords =
                    consumer.poll(1000);

            if (consumerRecords.count()==0) {
                noRecordsCount++;
                if (noRecordsCount > giveUp) break;
                else continue;
            }

            consumerRecords.forEach(record -> {
                System.out.printf("Consumer Record:(%d, %s, %d, %d)\n",
                        record.key(), record.value(),
                        record.partition(), record.offset());
            });

            consumer.commitAsync();
        }
        consumer.close();
        System.out.println("DONE");
    }

 public static void main(String... args) throws Exception {
     runConsumer();
  }
}
```

Source: http://cloudurable.com/blog/kafka-tutorial-kafka-consumer/index.html

Recall: Reading data from Kafka

Consumer “groups”

- Allow multi-threaded and/or multi-machine consumption from Kafka topics.
- Consumers “join” a group by using the same `group.id`
- Kafka guarantees a message is only ever read by a single consumer in a group.
  - Kafka assigns the partitions of a topic to the consumers in a group so that each partition is consumed by exactly one consumer in the group.
  - Maximum parallelism of a consumer group: **#consumers** (in the group) <= **#partitions**

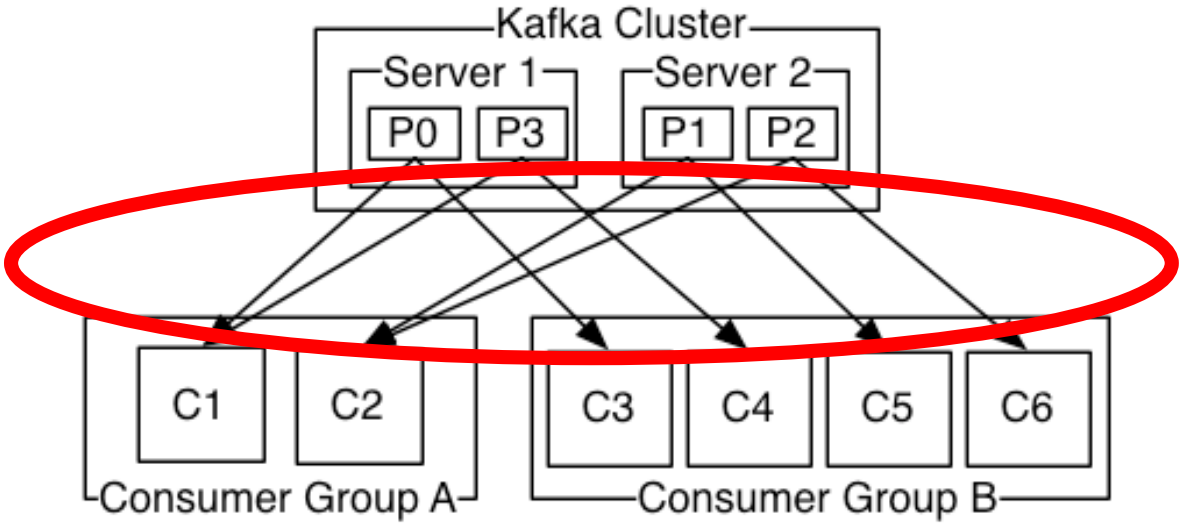


Guarantees when reading data from Kafka

- A message is only ever read by a single consumer in a group.
- A consumer sees messages in the order they were stored in the log.
- The order of messages is only guaranteed within a partition.
  - No order guarantee across partitions, which includes no order guarantee per-topic.
  - If total order (per topic) is required you can consider, for instance:
    - Use #partition = 1. Good: total order. Bad: Only 1 consumer process at a time.
    - “Add” total ordering in your consumer application, e.g. a Storm topology.
- Some gotchas:
  - If you have multiple partitions per thread there is NO guarantee about the order you receive messages, other than that within the partition the offsets will be sequential.
    - Example: You may receive 5 messages from partition 10 and 6 from partition 11, then 5 more from partition 10 followed by 5 more from partition 10, even if partition 11 has data available.
  - Adding more processes/threads will cause Kafka to rebalance, possibly changing the assignment of a partition to a thread (whoops).

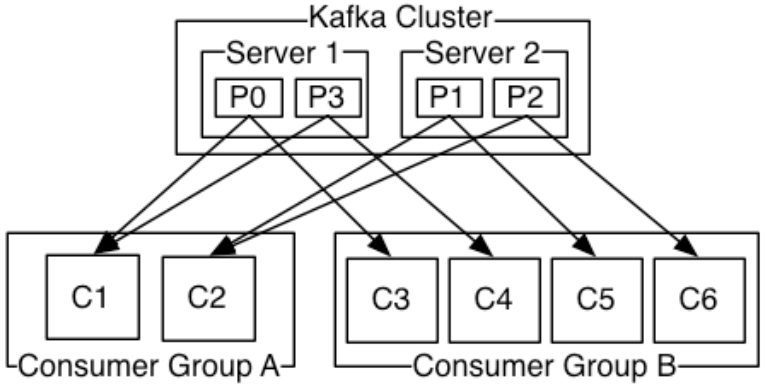
Rebalancing: how consumers meet brokers

Remember?



- The assignment of brokers – via the partitions of a topic – to consumers is quite **important**, and it is **dynamic** at run-time.

Rebalancing: how consumers meet brokers



- Why “dynamic at run-time”?
  - Machines can die, be added, …
  - Consumer apps may die, be re-configured, added, …
- Whenever this happens a **rebalancing** occurs.
  - Rebalancing is a normal and expected lifecycle event in Kafka.
  - But it’s also a nice way to shoot yourself or Ops in the foot.
- Why is this important?
  - Most Ops issues are due to

    1) **rebalancing** and 2) **consumer lag**.
  - So Dev + Ops must understand what goes on.

Rebalancing: how consumers meet brokers

- Rebalancing?
  - Consumers in a group come into consensus on which consumer is consuming which partitions → required for distributed consumption
  - Divides broker partitions evenly across consumers, tries to reduce the number of broker nodes each consumer has to connect to
- When does it happen? Each time:
  - a **consumer** joins or leaves a consumer group, OR
  - a **broker** joins or leaves, OR
  - a topic "joins/leaves" via a filter, cf. `createMessageStreamsByFilter()`
- Examples:
  - If a consumer or broker fails to heartbeat to ZK → rebalance!
  - `createMessageStreams()` registers consumers for a topic, which results in a rebalance of the consumer-broker assignment.

Kafka Polyglot Clients/ Wire Protocol

- Kafka communication from clients and servers wire protocol over TCP
- Protocol versioned
- Maintains backwards compatibility
- Many Languages supported
- Kafka REST proxy allows easy integration with other systems via HTTP and JSON
- Also provide Avro/ Schema registry support via an extended Kafka ecosystem

Serialization in Kafka

- Kafka does not care about data format of msg payload
- Up to developer (= you) to handle serialization/deserialization
  - Common choices in practice: Avro, JSON

```
/**
 *  Create a list of message streams of type T for each topic.
 *
 *  @param topicCountMap  a map of (topic, #streams) pair
 *  @param decoder a decoder that converts from Message to T
 *  @return a map of (topic, list of  KafkaStream) pairs.
 *          The number of items in the list is #streams. Each stream supports
 *          an iterator over message/metadata pairs.
 */
public <K,V> Map<String, List<KafkaStream<K,V>>>
  createMessageStreams(Map<String, Integer> topicCountMap, Decoder<K> keyDecoder, Decoder<V> valueDecoder);

/**
 *  Create a list of message streams of type T for each topic, using the default decoder.
 */
public Map<String, List<KafkaStream<byte[], byte[]>>> createMessageStreams(Map<String, Integer> topicCountMap);
```

(Code above is from the High Level Consumer API)

Serialization in Kafka: using Avro

- One way to use Avro in Kafka is via Twitter Bijection.
  - https://github.com/twitter/bijection
- Approach: Convert pojo to byte[], then send byte[] to Kafka.
  - Bijection in Scala:

```
5  val tweet = new Tweet("miguno", "Terran is the cheese race.", 1366190000)
6
7  // From POJO to byte array
8  val bytes = Injection[Tweet, Array[Byte]](tweet)
9
10 // From byte array back to POJO
11 val recovered_tweet = Injection.invert[Tweet, Array[Byte]](bytes)
```

  - Bijection in Java: https://github.com/twitter/bijection/wiki/Using-bijection-from-java

- Full Kafka/Bijection example:
  - KafkaSpec in kafka-storm-starter
- Alternatives to Bijection:
  - e.g. https://github.com/miguno/kafka-avro-codec

Data compression in Kafka

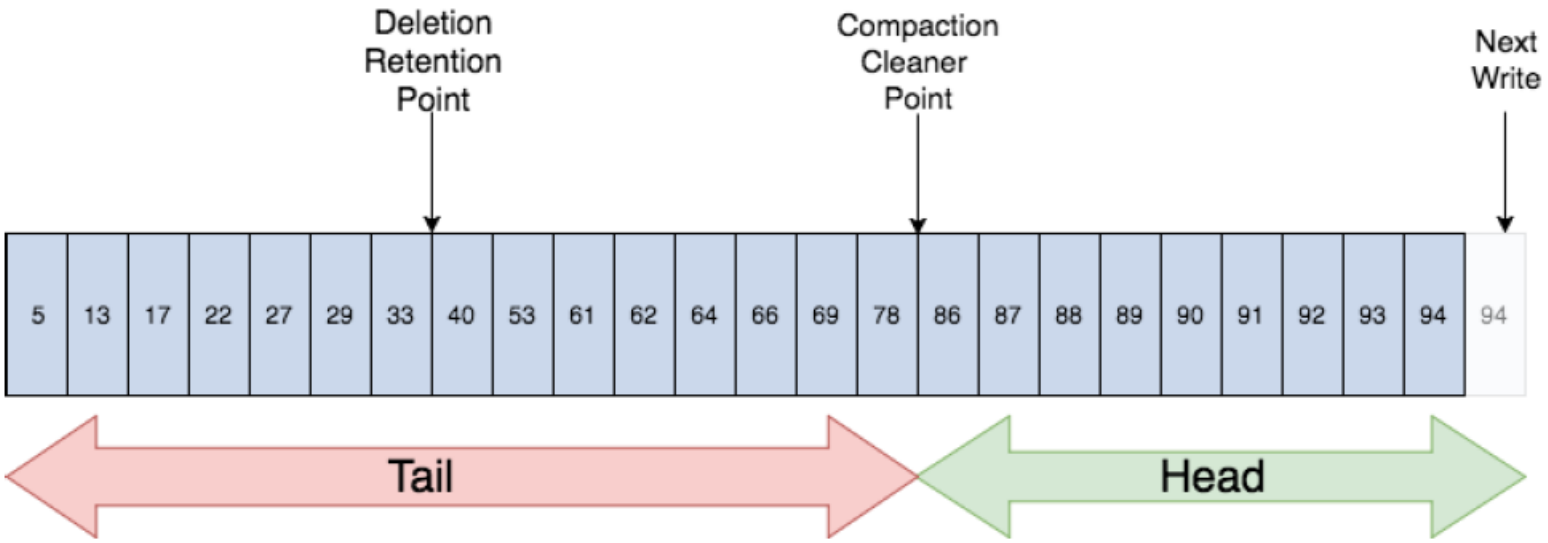
- Kafka provides E2E Batch Compression
- Bottleneck is not always CPU or disk but often network bandwidth
  - Especially in Cloud, Containerized and Virtualized environments
  - Especially when communicating between Datacenters or via WAN
- Instead of compressing messages one at a time, compresses whole batch and sent to Kafka Broker in one go !
  - Interplay with batching of messages, e.g. larger batches typically achieve better compression ratios.
  - Message batch gets written in compressed form in log partition and do NOT get decompressed until they are consumed.
- GZIP, Snappy and LZ4 compression algorithms supported
- Details about compression in Kafka:
  - https://cwiki.apache.org/confluence/display/KAFKA/Compression

Log Compaction Overview

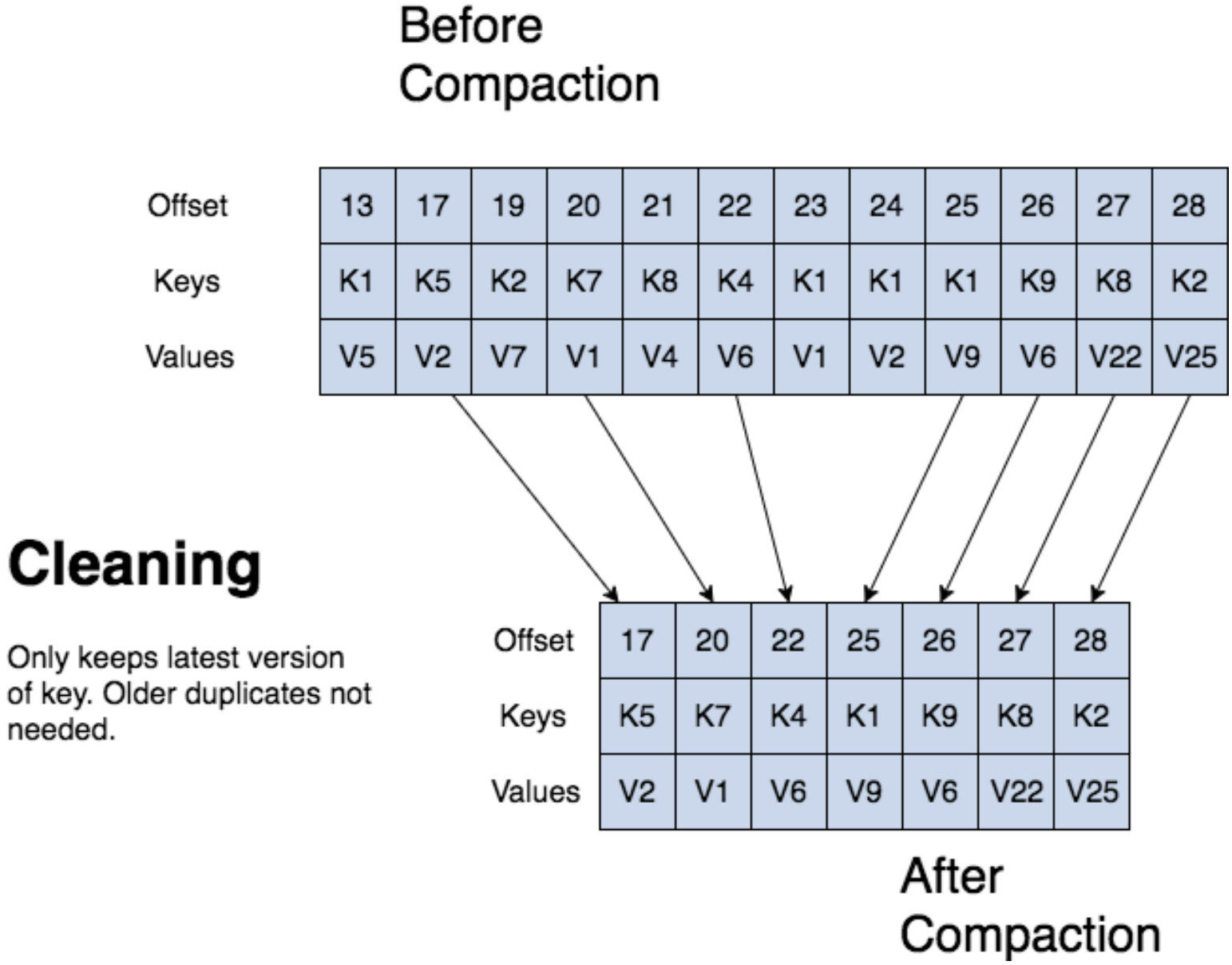
- Recall Kafka can delete older records based on
  - time period
  - size of a log
- Kafka also supports log compaction for **Record Key** Compaction
- Log compaction: keep latest version of record and delete older versions

Log Compaction Structure

- Log has head and tail
- Head of compacted log identical to a traditional Kafka log
- New records get appended to the head
- Log compaction works at tail of the log
- Tail gets compacted
- Records in tail of log retain their original offset when written after compaction



Log Compaction Cleaning for Key-value Data



Log Compaction Overview

- Log compaction retains last known value for each record key
- Useful for restoring state after a crash or system failure, e.g., in memory service, persistent data store, reloading a cache
- Data streams is to log changes to keyed, mutable data,
  - e.g., changes to a database table, changes to object in in-memory microservice
- Topic log has full snapshot of final values for every key - not just recently changed keys
- Downstream consumers can restore state from a log compacted topic

Log Compaction Details

- All offsets remain valid, even if record at offset has been compacted away (next highest offset)
- Compaction also allows for deletes. A message with a key and a null payload acts like a tombstone (a delete marker for that key)
  - Tombstones get cleared after a period.
- Log compaction periodically runs in background by recopying log segments.
- Compaction does not block reads and can be throttled to avoid impacting I/O of producers and consumers

Log Compaction Guarantee

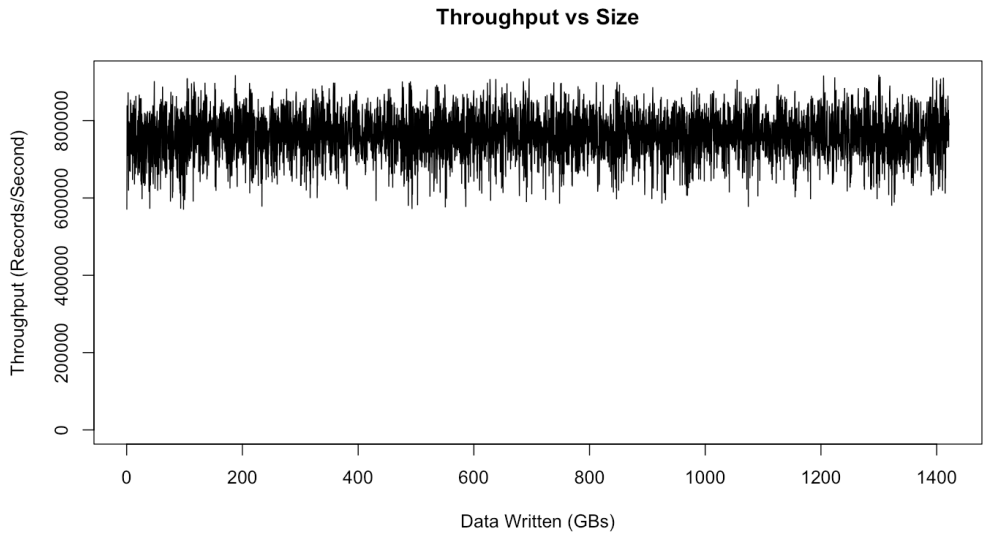
- If consumer stays caught up to head of the log, it sees every record that is written.
  - Topic config ***min.compaction.lag.ms*** used to guarantee minimum period that must pass before message can be compacted.
  - Consumer sees all tombstones as long as the consumer reaches head of log in a period less than the topic config ***delete.retention.ms*** (the default is 24 hours).
- Compaction will never re-order messages, just remove some.
- Offset for a message never changes.
- Any consumer reading from start of the log, sees at least final state of all records in order they were written

Failover vs. Disaster Recovery

- Kafka Replication is for Failover
  - Replication of Kafka Topic Log partitions allows for failure of a rack or AWS availability zone
- Mirror Maker is used for Disaster Recovery
  - Mirror Maker replicates a Kafka cluster to another datacenter or AWS region
  - Called Mirroring since replication happens within a cluster.

How fast is Kafka?

- **"Up to 2 million writes/sec on 3 cheap machines"**
  - Using 3 producers on 3 different machines, 3x async replication
    - Only 1 producer/machine because NIC already saturated
- **Sustained throughput as stored data grows**
  - Slightly different test config than 2M writes/sec above.



- Test setup
  - Kafka trunk as of April 2013, but 0.8.1+ should be similar.
  - 3 machines: 6-core Intel Xeon 2.5 GHz, 32GB RAM, 6x 7200rpm SATA, 1GigE

http://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

Why is Kafka so Fast ?

- Adopt the principle of Zero Copy
  - Using sendfile (Java's NIO FileChannel transferTo method)
  - Implement Linux sendfile( ) which skips unnecessary copies
- Batch Data in Chunks during Disk and Network access
  - The I/O scheduler will batch together consecutive small writes into bigger physical writes which improve system throughput
  - The I/O scheduler will attempt to re-sequence writes to minimize movement of the disk head which improves system throughput
  - Provide more efficient data compression
- Sequential Disk Writes
  - Write to immutable commit log => Disk accessed in sequential manner
  - Eliminate slow disk seek or random I/O operations

Why is Kafka so Fast ? (cont'd)

- Heavily relies on Linux PageCache
  - It automatically uses all free memory on the machine
  - In a system where consumers are roughly caught up with producers, you are essentially reading data from cache
- Scale out Horizontally
  - Use 100's to 1000's of partitions (i.e. sharding) for a single topic (group of messages of interest)
  - Spread workload to thousands of servers to handle massive load

Why is Kafka so fast (cont'd) ?

- Fast **writes**:
  - While Kafka persists all data to disk, essentially all writes go to the **page cache** of OS, i.e. RAM.
  - Cf. hardware specs and OS tuning

- Fast **reads**:
  - Very efficient to transfer data from page cache to a network **socket**
  - Linux: **`sendfile()`** system call

- Combination of the two = fast Kafka!
  - Example (Operations): On a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks as they will be serving data entirely from cache.

http://kafka.apache.org/documentation.html#persistence

Why is Kafka so fast?

- Example: Loggly.com, who run Kafka & Co. on Amazon AWS
  - “99.99999% of the time our data is coming from disk cache and RAM; only very rarely do we hit the disk.”
  - “One of our consumer groups (8 threads) which maps a log to a customer can process about 200,000 events per second draining from 192 partitions spread across 3 brokers.”
    - Brokers run on m2.xlarge Amazon EC2 instances backed by provisioned IOPS

http://www.developer-tech.com/news/2014/jun/10/why-loggly-loves-apache-kafka-how-unbreakable-infinitely-scalable-messaging-makes-log-management-better/

Recap: Key Strength of Kafka

- High Performance – High Throughput, Low-Latency
- Stable, Robust Replication => Reliable, Durability
- Support Flexible Publish-Subscribe model for Data Producers and Consumers
- Producer Tunable Consistency Guarantees
- Ordering Preserved (at Shard Level, i.e. Topic Partition)
- Works well with Systems that have Data Streams to process, aggregate, transform and load to other stores
  - “Connector” modules available for most mainstream Big Data processing frameworks.

Comparing Kafka with other Alternatives

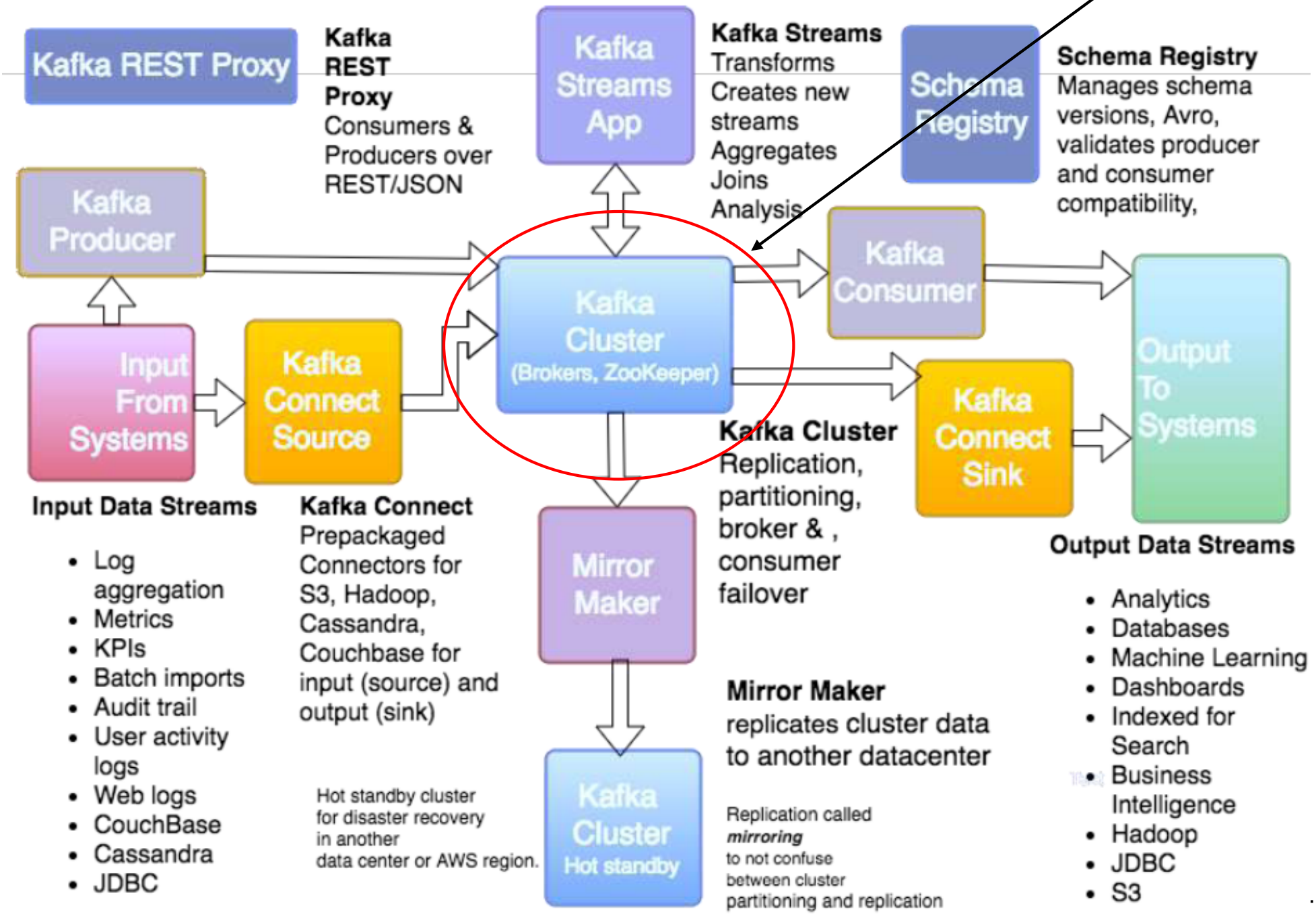
Kafka vs. Message Oriented Middleware (MOM) systems

- MOM = JMS, ActiveMQ, RabbitMQ, IBM MQ Series, Tibco, etc.
- Is Kafka a Queue or a Pub/Sub/Topic?
  - Yes
- Kafka is like a Queue per consumer group
  - Kafka is a queue system per consumer in consumer group so load balancing like JMS, RabbitMQ queue
- Kafka is like Topics in JMS, RabbitMQ, MOM
  - Topic/pub/sub by offering Consumer Groups which act like subscriptions
  - Broadcast to multiple consumer groups

Kafka vs. Message Oriented Middleware (MOM) systems

- By design, Kafka is better suited for scale than traditional MOM systems due to partition topic log
  - Load divided among Consumers for read by partition
  - Handle parallel consumers better than traditional MOM
- Also by moving location (partition offset) in log to client/consumer side of equation instead of the broker, less tracking required by Broker and more flexible consumers
- Kafka written with mechanical sympathy, modern hardware, cloud in mind
  - Disks are faster
  - Servers have tons of system memory
  - Easier to spin up servers for scale out

The Extended Kafka Universe (beyond "Kafka Core")



Kafka and Amazon Kinesis are similar

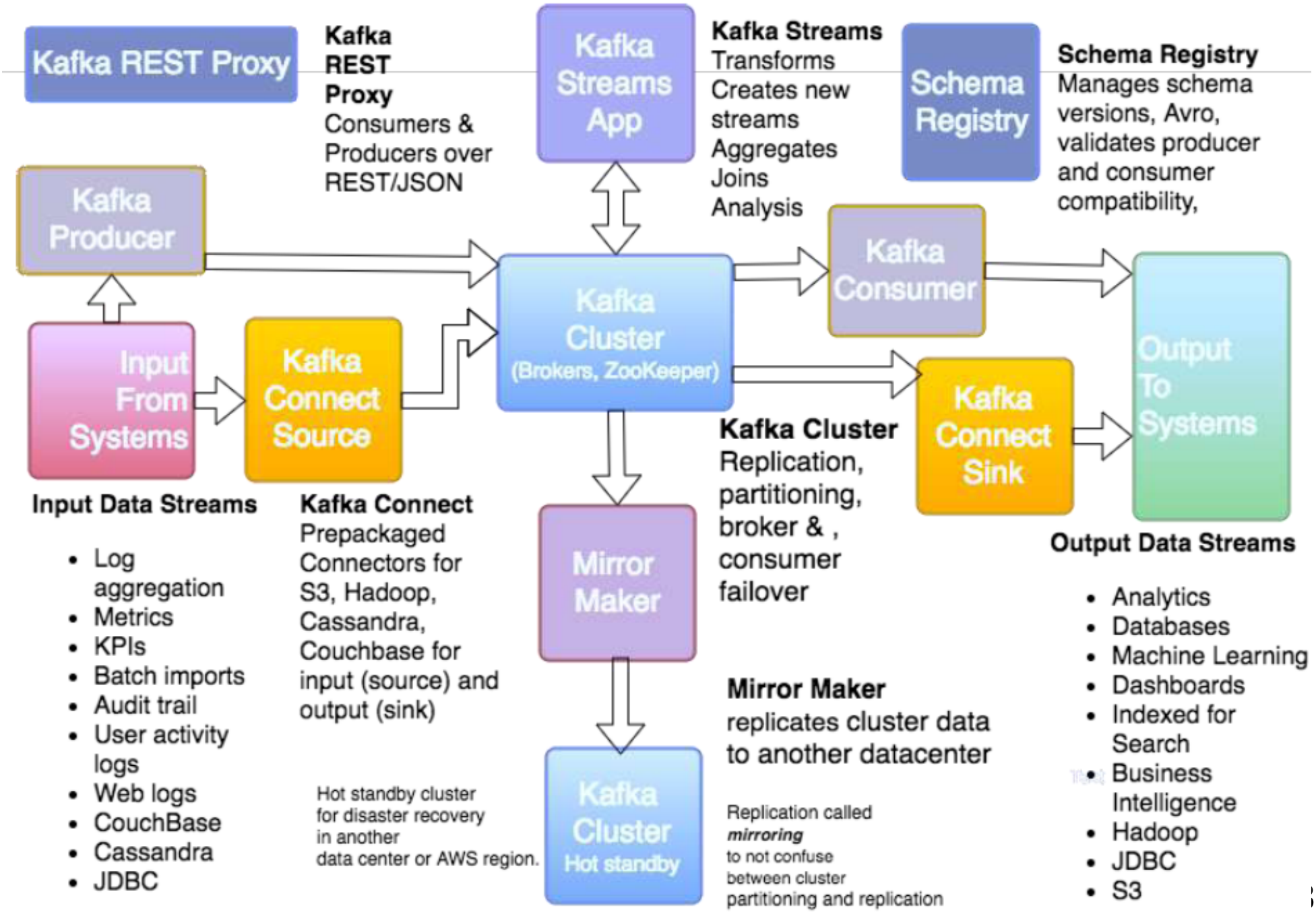
- Kinesis Streams is like Kafka Core
- Kinesis Analytics is like Kafka Streams
- Kinesis Shard is like Kafka Partition
- Similar and get used in similar use cases
- In Kinesis, data is stored in shards. In Kafka, data is stored in partitions
- Kinesis Analytics allows you to perform SQL like queries on data streams
- Kafka Streaming allows you to perform functional aggregations and mutations
- Kafka integrates well with Spark and Flink which allows SQL like queries on streams

Kafka vs. Amazon Kinesis

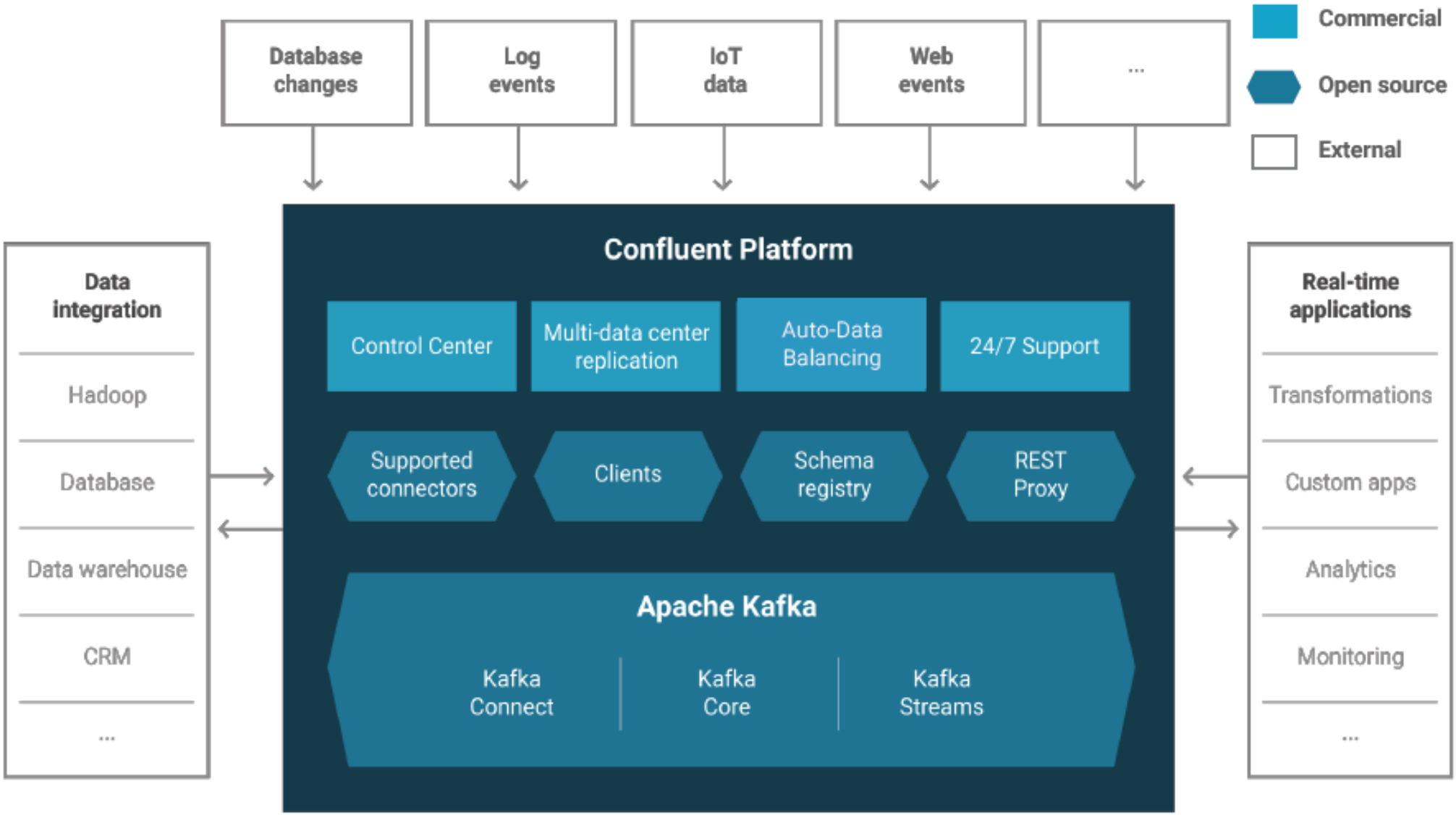
- Data is stored in Kinesis for default 24 hours, and you can increase that up to 7 days.
- Kafka records default stored for 7 days
  - can increase until you run out of disk space.
  - Decide by the size of data or by date.
  - Can use compaction with Kafka so it only stores the latest timestamp per key per record in the log
- With Kinesis data can be analyzed by Lambda before it gets sent to S3 or RedShift
- With Kinesis you pay for use, by buying read and write units.
- Kafka is more flexible than Kinesis but you have to manage your own clusters, and requires some dedicated DevOps resources to keep it going
- Kinesis is sold as a service and does not require a DevOps team to keep it going (can be more expensive and less flexible, but much easier to setup and run)

The Extended Universe of Kafka by Confluent

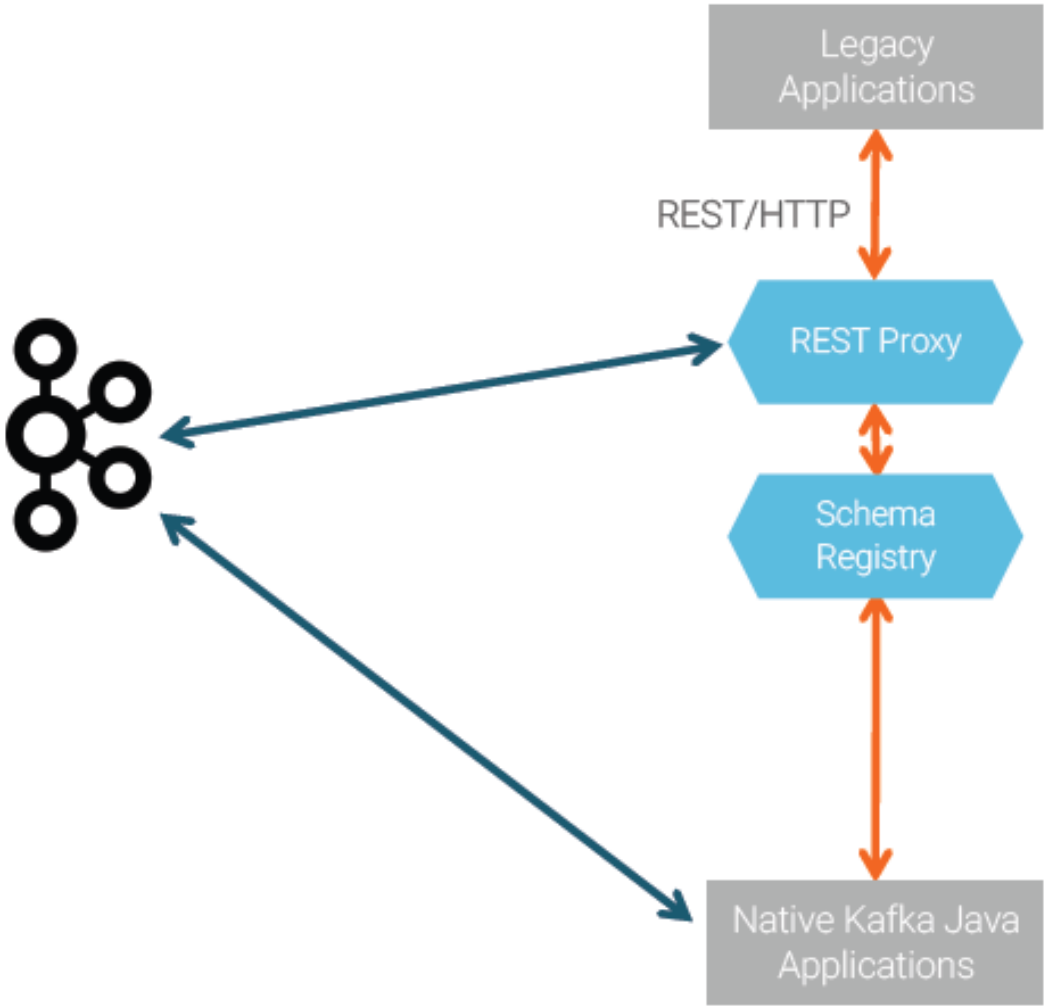
The “Extended” Kafka Universe



The Confluent Enterprise Streaming Platform

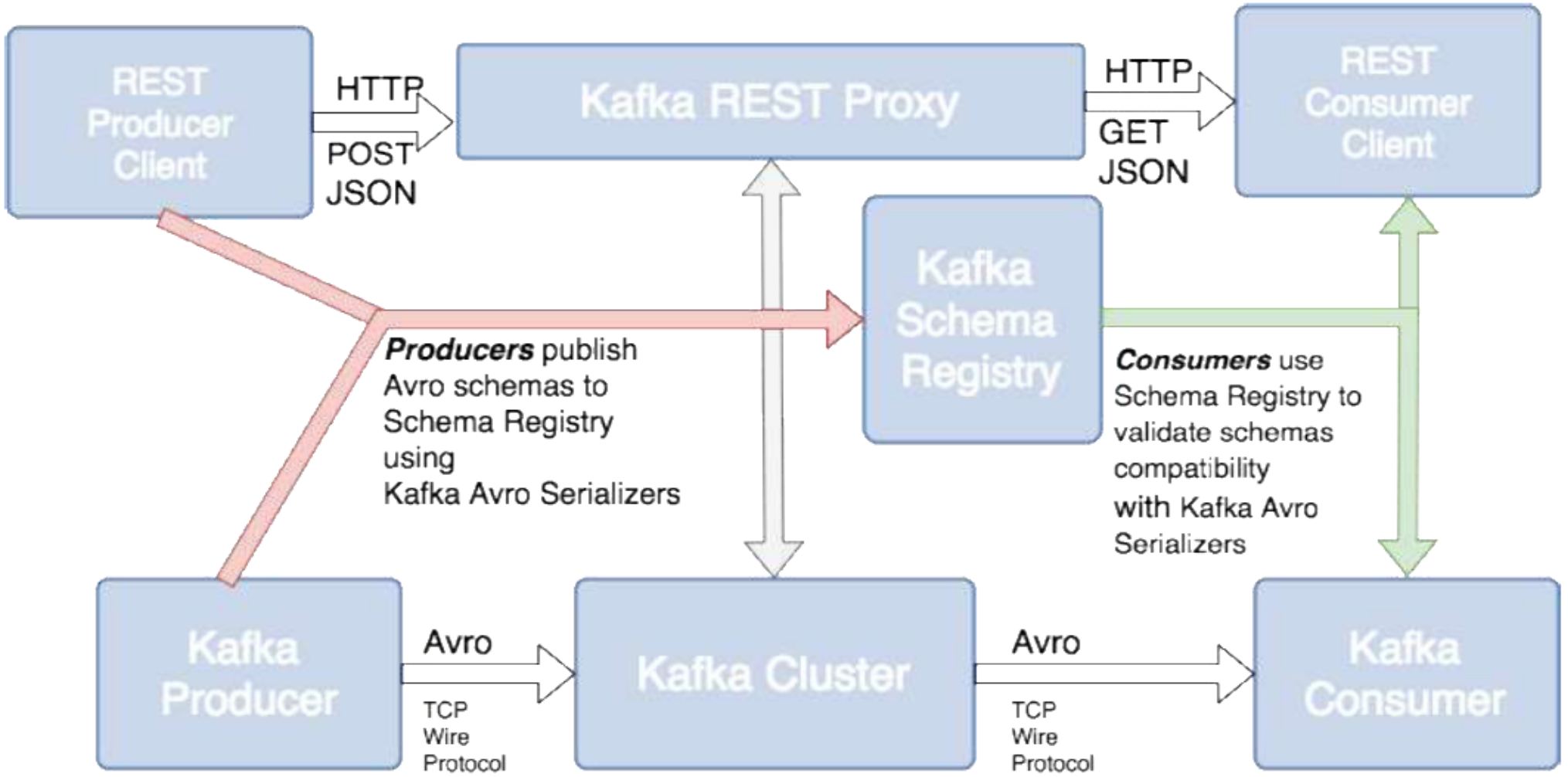


REST Proxy: Enable other Applications to access Kafka data

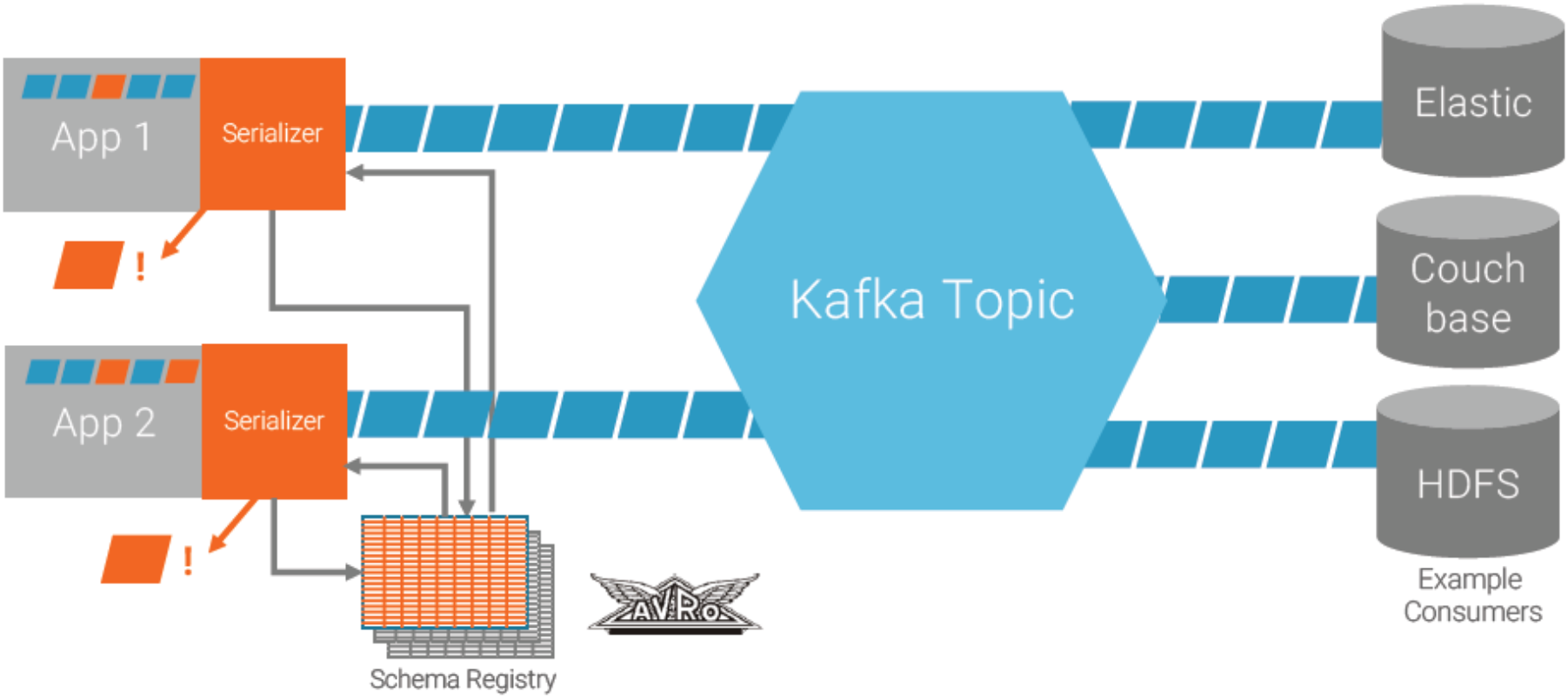


- Provide a RESTful Interface to a Kafka Cluster
  - Simplify message creation and consumption
  - Simplify system administration

Kafka REST Proxy and Schema Registry



Sample use of the Confluent Schema Registry

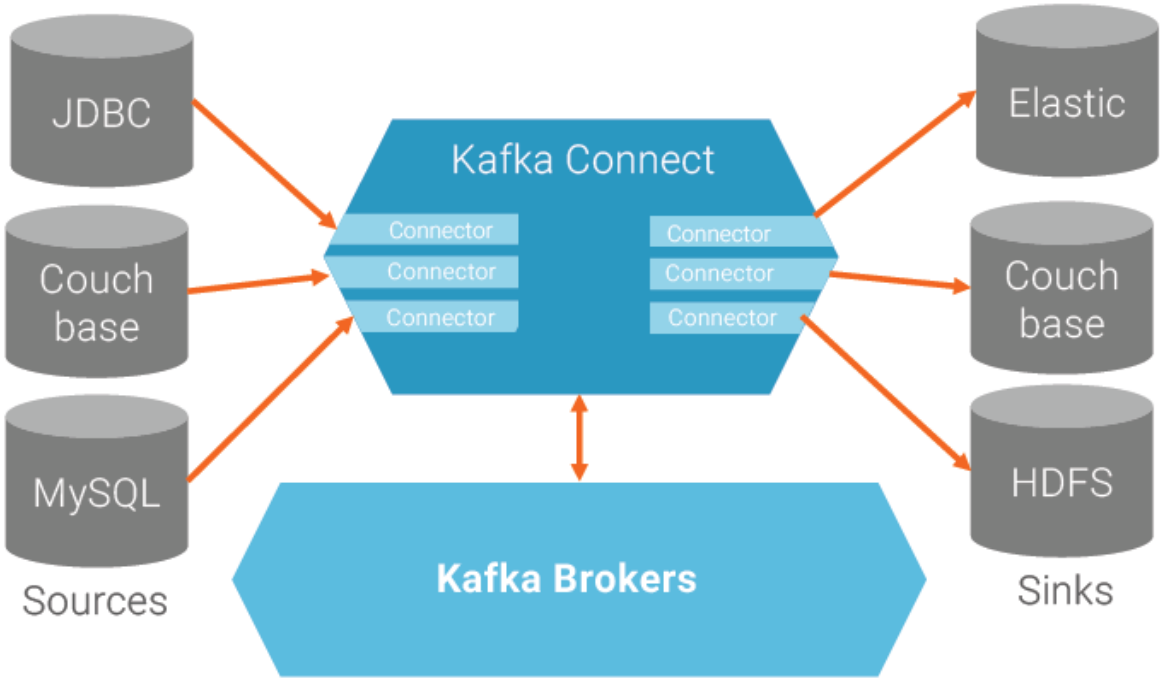


- Define the expected fields (schema) for each Kafka topic
- Leverage Avro to automatically handle schema changes (e.g. new fields) => enhance backward compatibility

Kafka Clients

- Define the expected fields (schema) for each Kafka topic
- Leverage Avro to automatically handle schema changes (e.g. new fields) => enhance backward compatibility

Apache Kafka Connect Library of Connectors for Streaming Data Capture



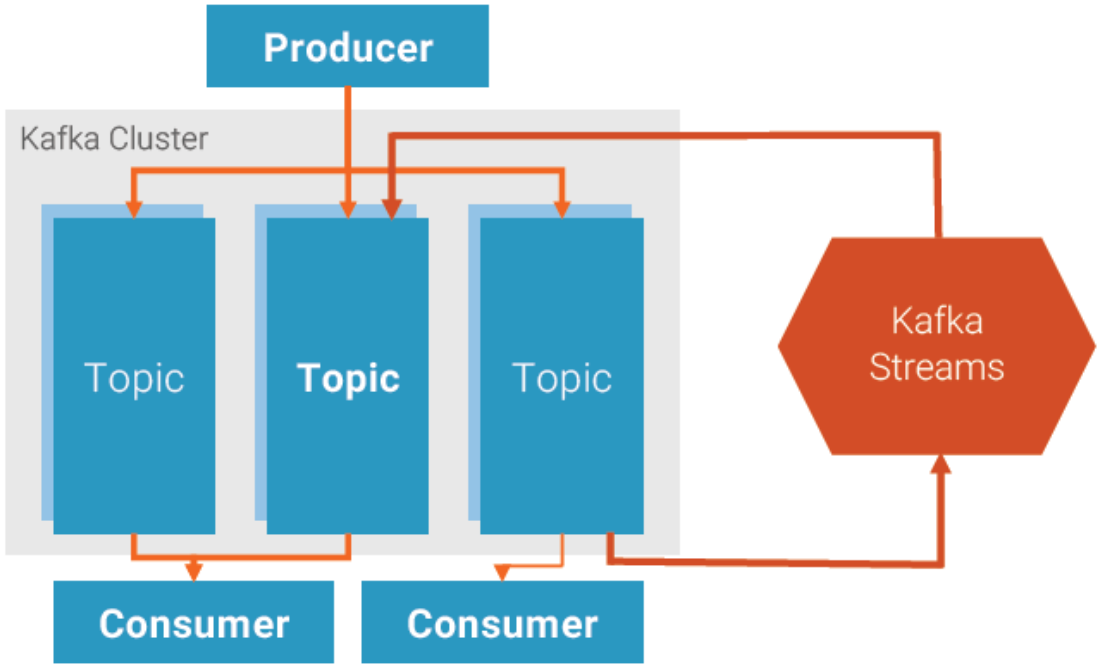
Databases	Datastore / File Store
JDBC* Couchbase Datastax / Cassandra GoldenGate JustOne DynamoDB MongoDB Hbase InfluxDB Kudu RethinkDB	HDFS* Apache Ignite FTP Syslog Hazelcast

Analytics	Applications / Other
Elasticsearch* Veritca Mixpanel	Attunity AWS / S3 Bloomberg Ticker Striim Solr Syncsort Twitter

* Denotes Connectors developed at Confluent and distributed with the Confluent Platform.

- Fault tolerant
- Preserve Data Schema
- Manage/Support Heterogeneous Sources and Sinks

Architecture of Kafka Streams

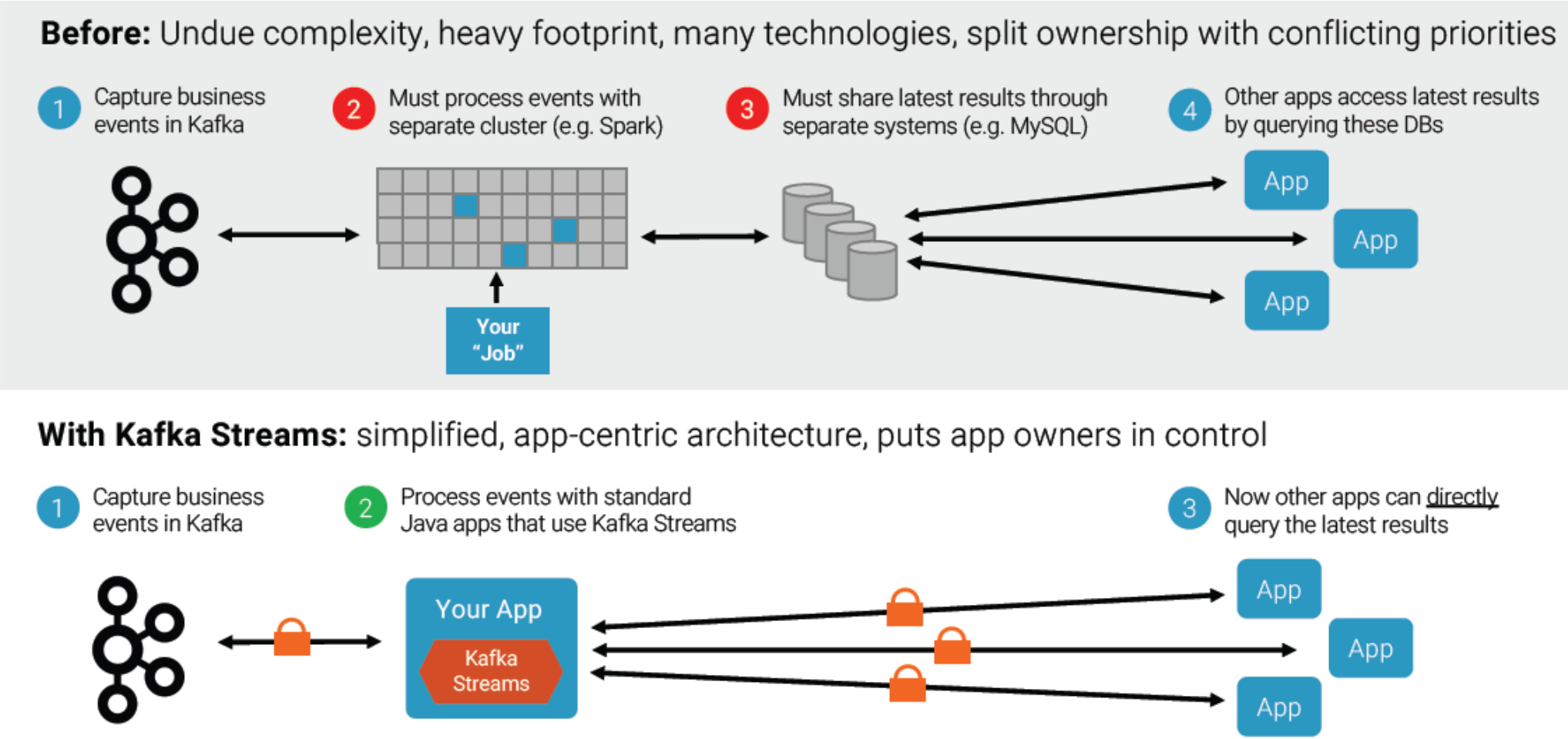


Example Use Cases for "Kafka Streams"

- Microservices
- Continuous queries
- Continuous transformations
- Event-triggered processes

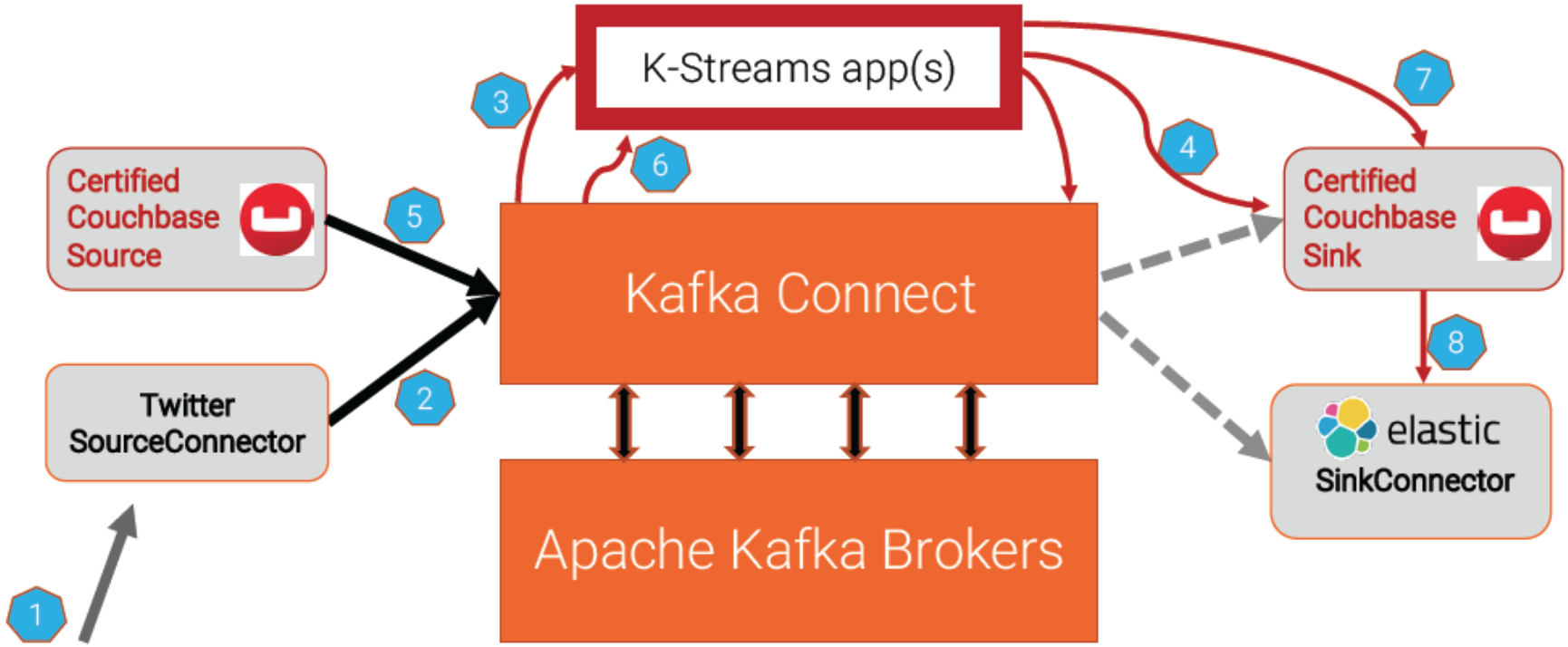
- Available as high-level DSL as well as low-level API to enable flexible application development
- No additional cluster required
- Security and permission integrated from Kafka

Comparison of Alternative Service Architectures to support Enterprise-level Stream Processing



"Grain of Salt" Alert: This slide is from Confluent

A Sample Streaming Data Pipeline using the Kafka ecosystem



1. Twitter feed with sentiment data
2. Use Twitter Source connector to publish data to Kafka topic
3. Kafka Streams application augments Twitter records with sentiment analysis
4. Kafka Streams output to Couchbase
5. Couchbase Source Connector to pull data from Couchbase bucket back to Kafka topic
6. 2nd stage Kafka Streams app saves data to another Couchbase bucket and then onto Elastic Search

Operating Kafka

Kafka broker hardware specs @ LinkedIn (circa 2014)

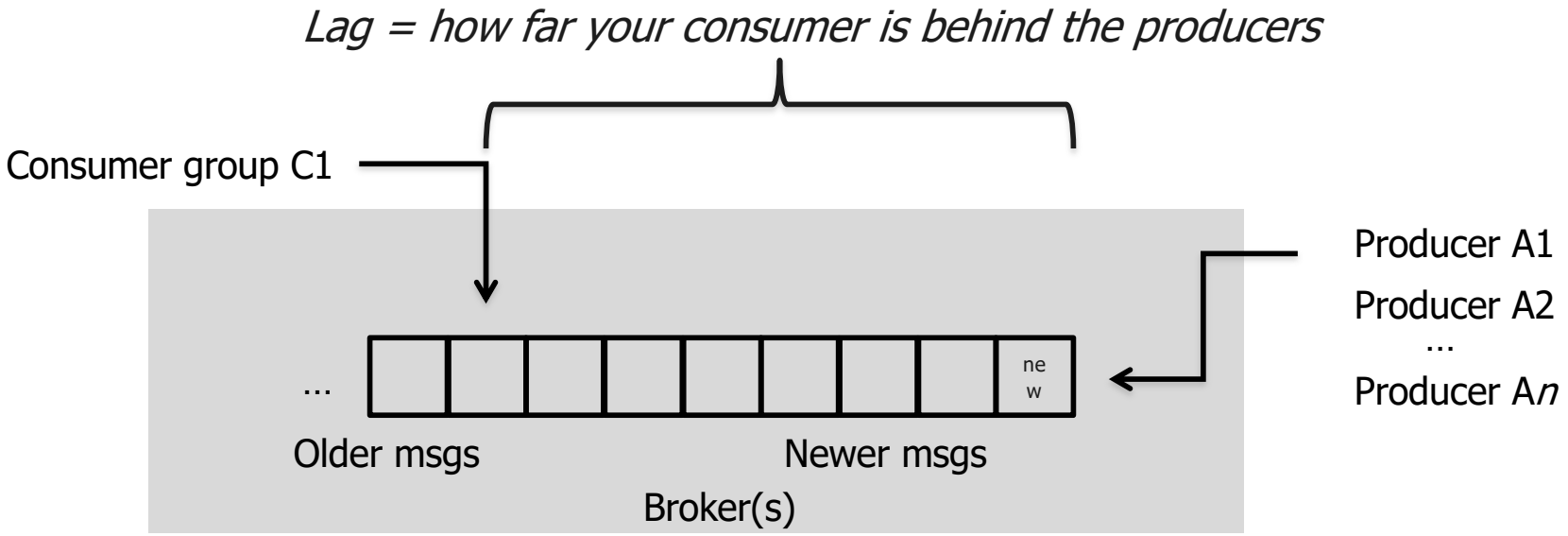
- Solely dedicated to running Kafka, run nothing else.
  - 1 Kafka broker instance per machine
- 2x 4-core Intel Xeon (info outdated?)
- **64 GB RAM (up from 24 GB)**
  - Only 4 GB used for Kafka broker, remaining 60 GB for page cache
  - Page cache is what makes Kafka **fast**
- **RAID10 with 14 spindles**
  - More spindles = higher disk throughput
  - Cache on RAID, with battery backup
  - Before H/W upgrade: 8x SATA drives (7200rpm), not sure about RAID
- 1 GigE (?) NICs

- EC2 example: m2.2xlarge @ $0.34/hour, with provisioned IOPS

Operating Kafka

- Typical operations tasks include:
  - Adding or removing brokers
    - Example: ensure a newly added broker actually receives data, which requires moving partitions from existing brokers to the new broker
    - Kafka provides helper scripts but still manual work involved
  - Balancing data/partitions to ensure best performance
  - Add new topics, re-configure topics
    - Example: Increasing #partitions of a topic to increase max parallelism
  - Apps management: new producers, new consumers
- Biggest Challenges to handle growth of Kafka adoption, increase in Producers, Consumers
  - Most problems are due to: 1) Consumer Lag ; and 2) Rebalancing
- Original design was not created with security in mind.
  - Discussion started in June 2014 to add security features.
  - Covers transport layer security, data encryption at rest, non-repudiation, A&A, …

Monitoring Kafka apps: Consumer lag



- **Lag** is a consumer problem:
  - Too slow, too much GC, losing connection to ZK or Kafka, …
  - Bug or design flaw in Consumer
  - Operational mistakes: e.g. you brought up 6 servers in parallel, each one in turn triggering rebalancing, then hit Kafka's rebalance limit;
    cf. `rebalance.max.retries` (default: 4) & friends

Monitoring Kafka itself (1 of 3)

- **Under-replicated partitions**
  - For example, because a broker is down.
  - Means cluster runs in degraded state.
    - FYI: LinkedIn runs with replication factor of 2 => 1 broker can die.

- Offline partitions
  - Even worse than under-replicated partitions!
  - *Serious* problem (data loss) if anything but `0` offline partitions.

Monitoring Kafka itself (1 of 3)

- Data size on disk
  - Should be balanced across disks/brokers
  - Data balance even more important than partition balance
  - There are scripts for balancing data/partitions across brokers

- Broker partition balance
  - Count of partitions should be balanced evenly across brokers

Monitoring Kafka itself (1 of 3)

- Leader partition count
  - Should be balanced across brokers so that each broker gets the same amount of load
  - Only 1 broker is ever the leader of a given partition, and only this broker is going to talk to producers + consumers for that partition
    - Non-leader replicas are used solely as safeguards against data loss
  - Recent Feature to support auto-rebalance the leaders and partitions in case a broker dies
- Network utilization
  - Maxed network one reason for under-replicated partitions
  - LinkedIn don't run anything but Kafka on the brokers, so network max is due to Kafka. Hence, when they max the network, they need to add more capacity across the board.

Monitoring ZooKeeper

- Ensemble (= cluster) availability
  - LinkedIn run 5-node ensembles = tolerates 2 dead
  - Twitter run 13-node ensembles = tolerates 6 dead
- Latency of requests
  - Metric target is `0` ms when using SSD's in ZooKeeper machines.
    - Why? Because SSD's are so fast they typically bring down latency below ZK's metric granularity (which is per-ms).
- Outstanding requests
  - Metric target is `0`.
  - Why? Because ZK processes all incoming requests serially. Non-zero values mean that requests are backing up.

“Auditing” Kafka

- Every **producer** is also writing messages into a special topic about how many messages it produced, every 10mins.
  - Example: "Over the last 10mins, I sent N messages to topic X.”
  - This metadata gets mirrored like any other Kafka data.

- **Audit consumer**
  - 1 audit consumer per Kafka cluster
  - Reads *every* single message out of “its” Kafka cluster. It then calculates counts for each topic, and writes those counts back into the same special topic, every 10mins.
    - Example: "I saw M messages in the last 10mins for topic X in THIS cluster”
  - And the next audit consumer in the next, downstream cluster does the same thing.

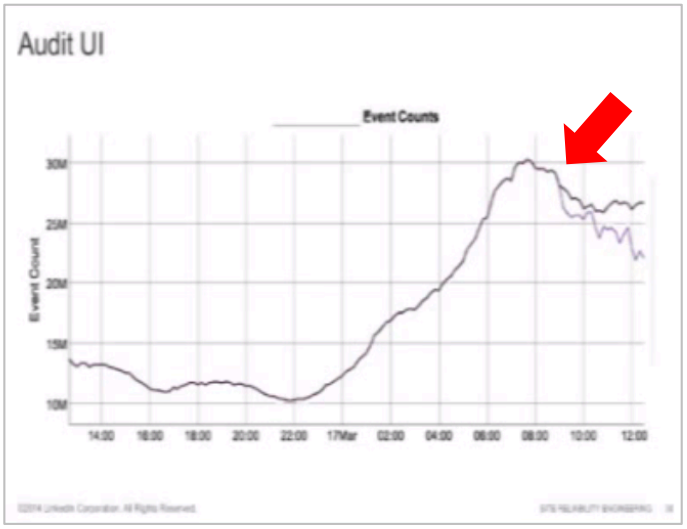
“Auditing” Kafka

- Monitoring audit consumers
  - Completeness check
    - "#msgs according to producer == #msgs seen by audit consumer?"
  - Lag
    - "Can the audit consumers keep up with the incoming data rate?"
    - If audit consumers fall behind, then all your tracking data falls behind as well, and you don't know how many messages got produced.

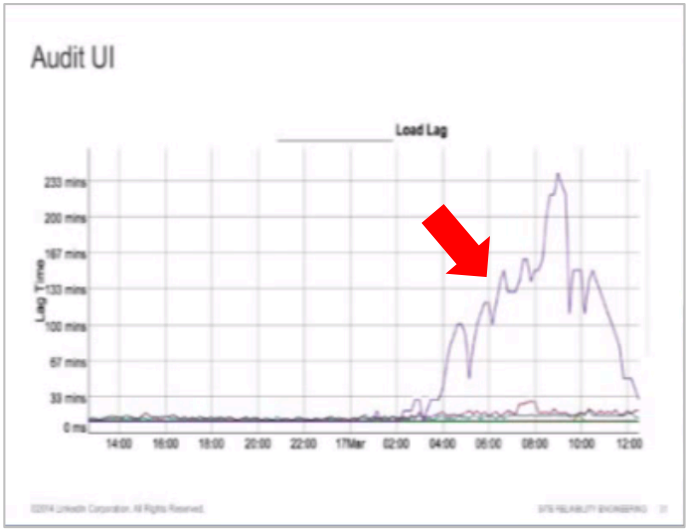
“Auditing” Kafka

- Audit UI
  - Only reads data from that special "metrics/monitoring" topic, but this data is read from every Kafka cluster at LinkedIn.
    - What they producers said they wrote in.
    - What the audit consumers said they saw.
  - Shows correlation graphs (producers vs. audit consumers)
    - For each tier, it shows how many messages there were in each topic over any given period of time.
    - Percentage of how much data got through (from cluster to cluster).
    - If the percentage drops below 100%, then emails are sent to Kafka SRE+DEV as well as their Hadoop ETL team because that stops the Hadoop pipelines from functioning properly.

LinkedIn's Audit UI: a closing look



- Example 1: Count discrepancy
  - Caused by messages failing to reach a downstream Kafka cluster



- Example 2: Load lag

Kafka Performance Tuning

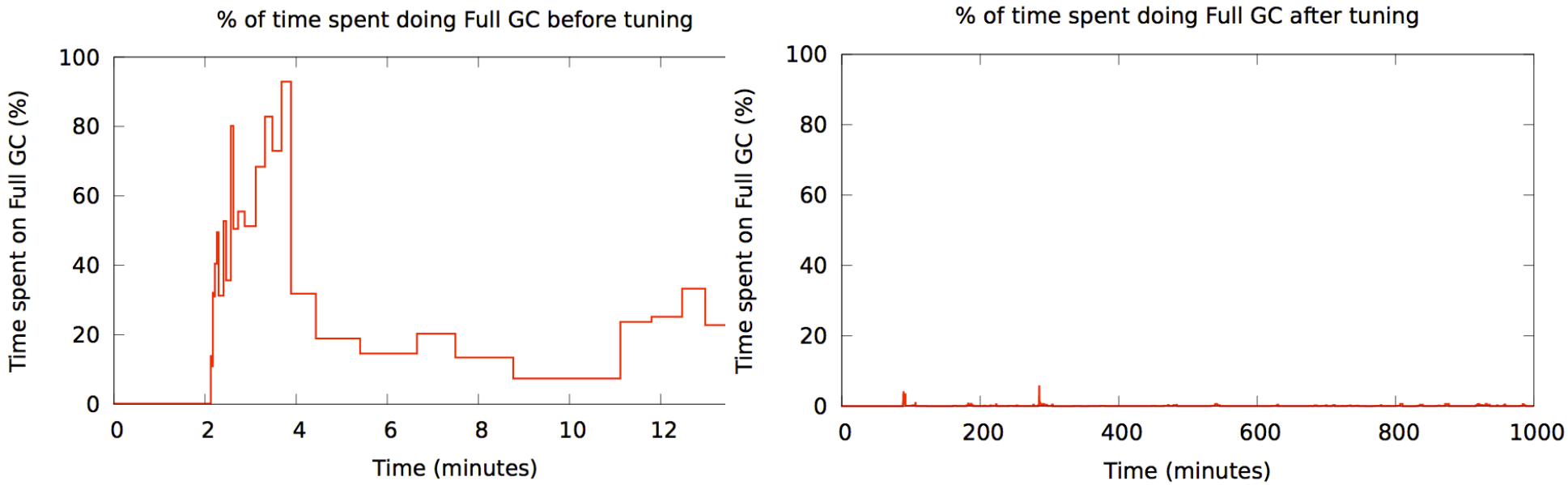
- OS Kernel tuning
  - Don't swap! `vm.swappiness = 0` (RHEL 6.5 onwards: `1`)
  - Allow more dirty pages but less dirty cache.
    - LinkedIn have lots of RAM in servers, most of it is for page cache (60 of 64 GB). They let dirty pages built up, but cache should be available as Kafka does lots of disk and network I/O.
    - See `vm.dirty_*_ratio` & friends
- Disk throughput
  - Longer commit interval on mount points. (ext3 or ext4)
    - Normal interval for ext3 mount point is 30s between flushes; LinkedIn: 120s. They can tolerate losing 2mins worth of data (because of partition replicas) so they rather prefer higher throughput here.
  - More spindles (RAID10 w/ 14 disks)

Java/ JVM tuning for Kafka

- Biggest issue: Garbage Collection (GC)
  - And, most of the time, the only issue

- Goal is to **minimize GC pause times**
  - Aka “stop-the-world” events – apps are halted until GC finishes

Java garbage collection in Kafka @ Spotify



Before tuning **After tuning**

https://www.jfokus.se/jfokus14/preso/Reliable-real-time-processing-with-Kafka-and-Storm.pdf

Kafka configuration tuning

- Often not much to do beyond using the defaults,
- Key candidates for tuning:

`num.io.threads`	should be >= #disks (start testing with == #disks)
`num.network.threads`	adjust it based on (concurrent) #producers, #consumers, and replication factor

Kafka usage tuning – lessons learned from others

- Don't break things up into separate topics unless the data in them is truly independent.
  - Consumer behavior can (and will) be extremely variable, don't assume you will always be consuming as fast as you are producing.

- Keep time related messages in the same partition.
  - Consumer behavior can extremely variable, don't assume the lag on all your partitions will be similar.
  - Design a partitioning scheme, so that the owner of one partition can stop consuming for a long period of time and your application will be minimally impacted (for example, partition by transaction id)

http://grokbase.com/t/kafka/users/145qtx4z1c/topic-partitioning-strategy-for-large-data

Further Reading

- Neha Narkhede, Gwen Shapira, Todd Palino, “Kafka – The Definitive Guide,” published by O’Reilly, July 2017.
  - Complimentary Free Copy available from the Confluent website

    https://www.confluent.io/wp-content/uploads/confluent-kafka-definitive-guide-complete.pdf