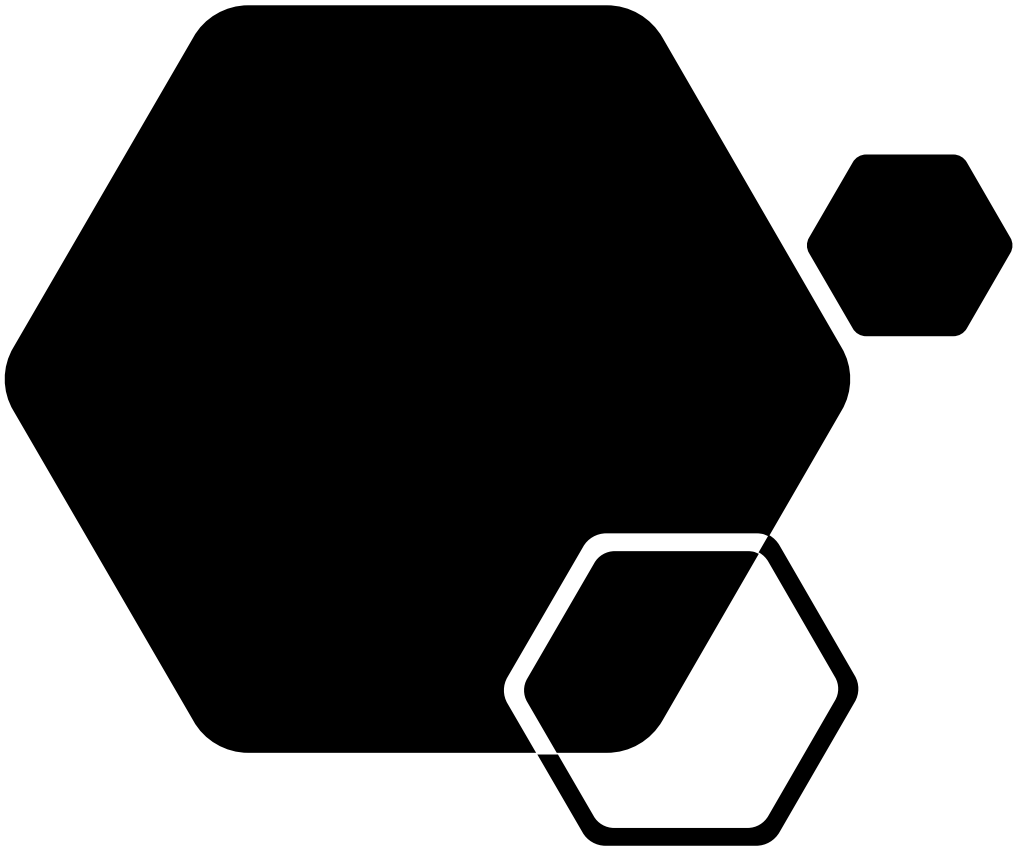
Develop performance oriented streaming application with Kafka

Devoxx Poland Handson

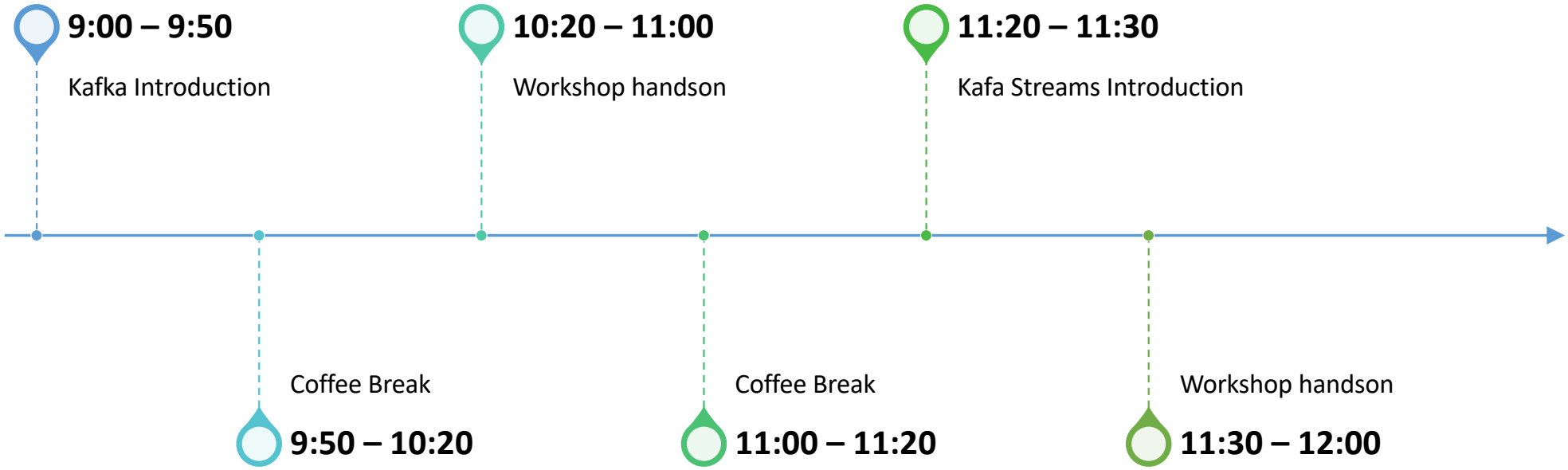
Muhammad Arslan

Lead/Staff Software Engineer

INVIDI Technologies AB



Schedule


9:00 – 9:50
Kafka Introduction
Coffee Break
9:50 – 10:20
10:20 – 11:00
Workshop handson
Coffee Break
11:00 – 11:20
11:20 – 11:30
Kafa Streams Introduction
Workshop handson
11:30 – 12:00

Agenda

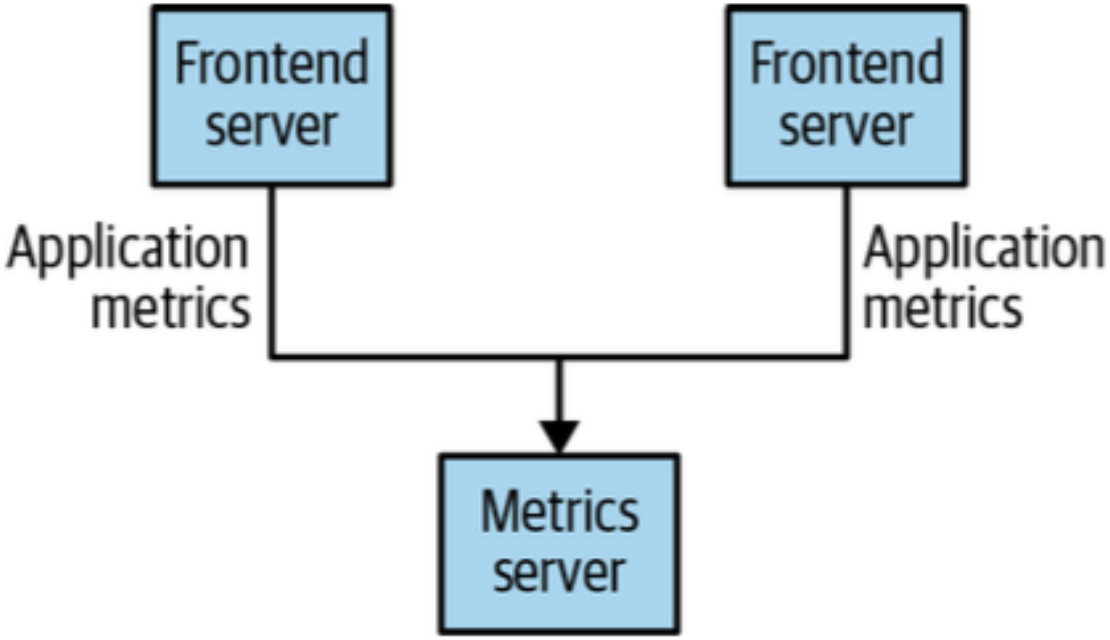
- Kafka Introduction
- Data Serialization Formats (avro, parquet, protobuf)
- Kafka Producer
- Kafka Consumer
- Kafka Streams

Kafka Introduction

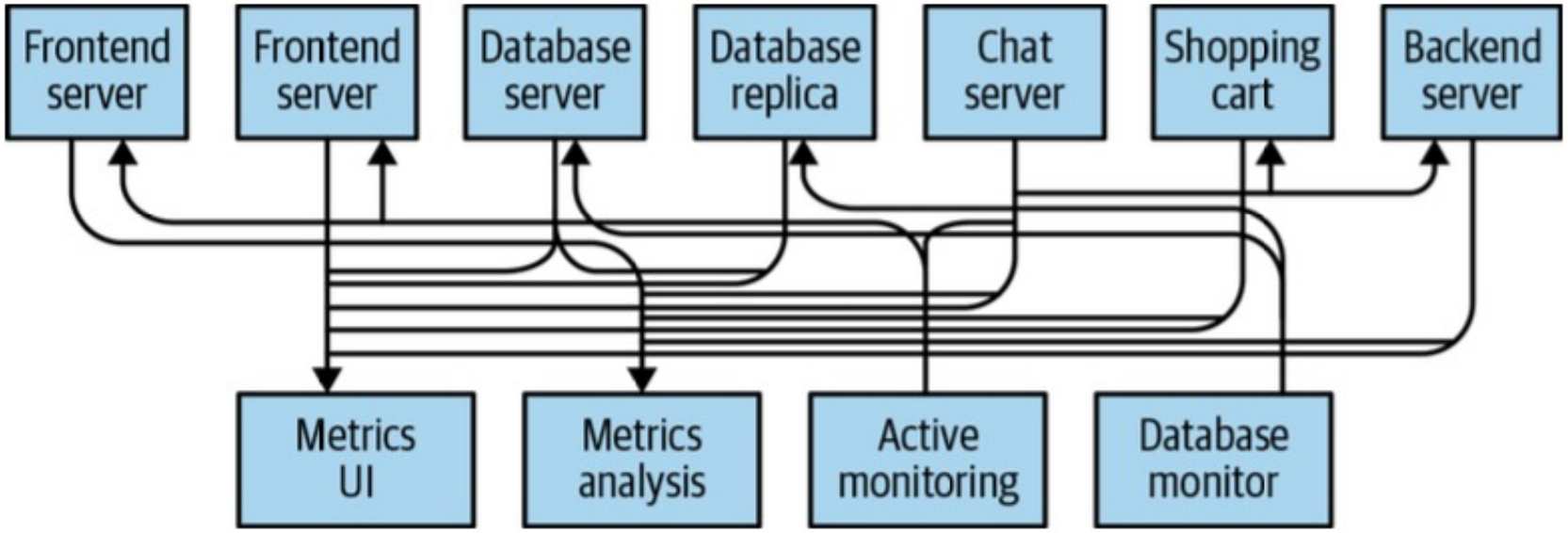
Publish/Subscribe Pattern

- Publish/Subscribe is a Pattern
- Publisher
- Subscriber
- Pub/Sub has broker, as central point where message are published

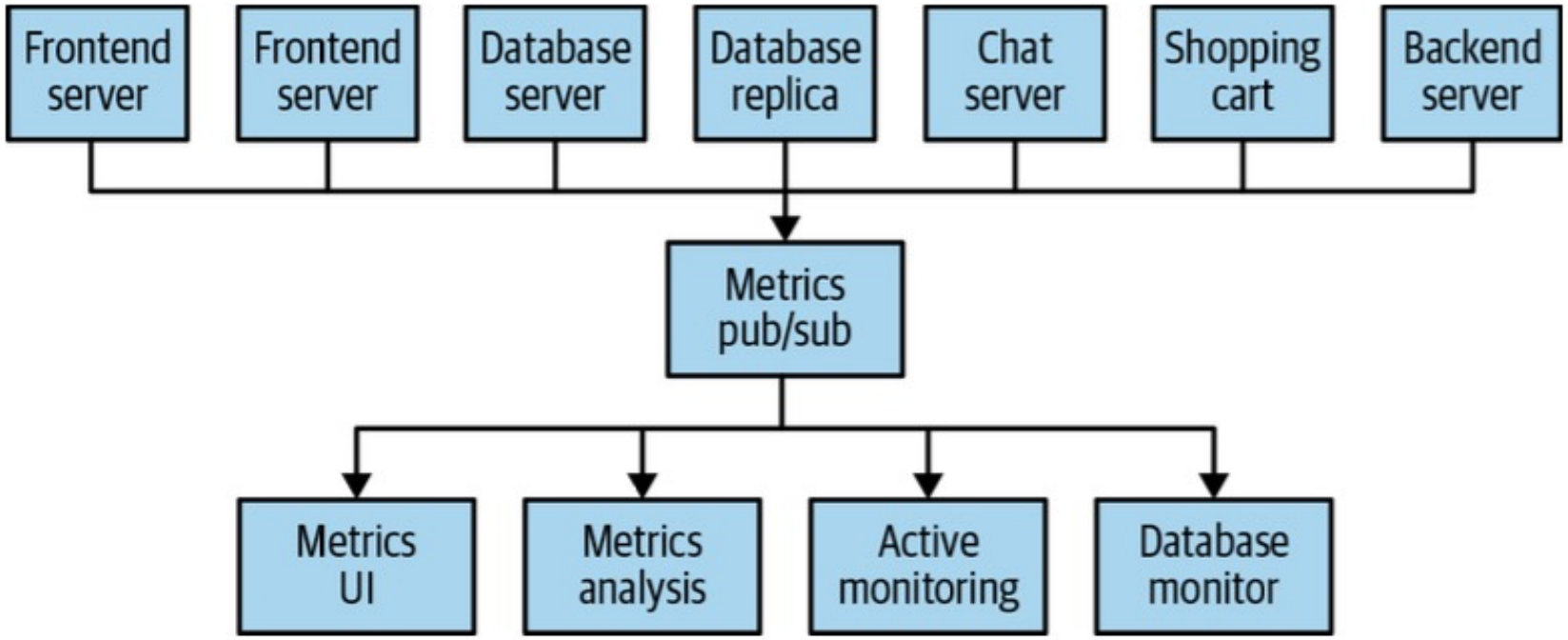
Direct Metrics Publisher



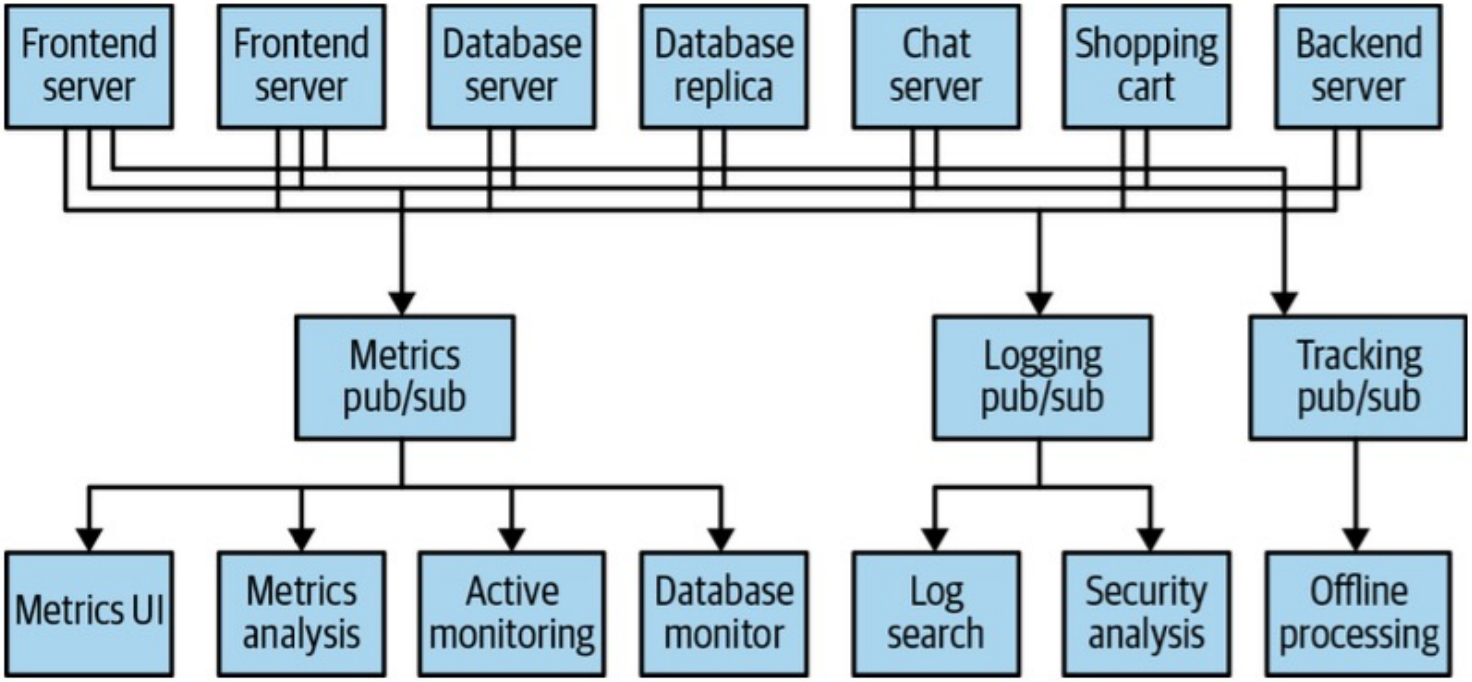
Many Metrics Publisher



Publish/Subscribe System



Individual/Many Queue Systems



Kafka

Apache Kafka was developed as a publish/subscribe messaging system designed to solve this problem.

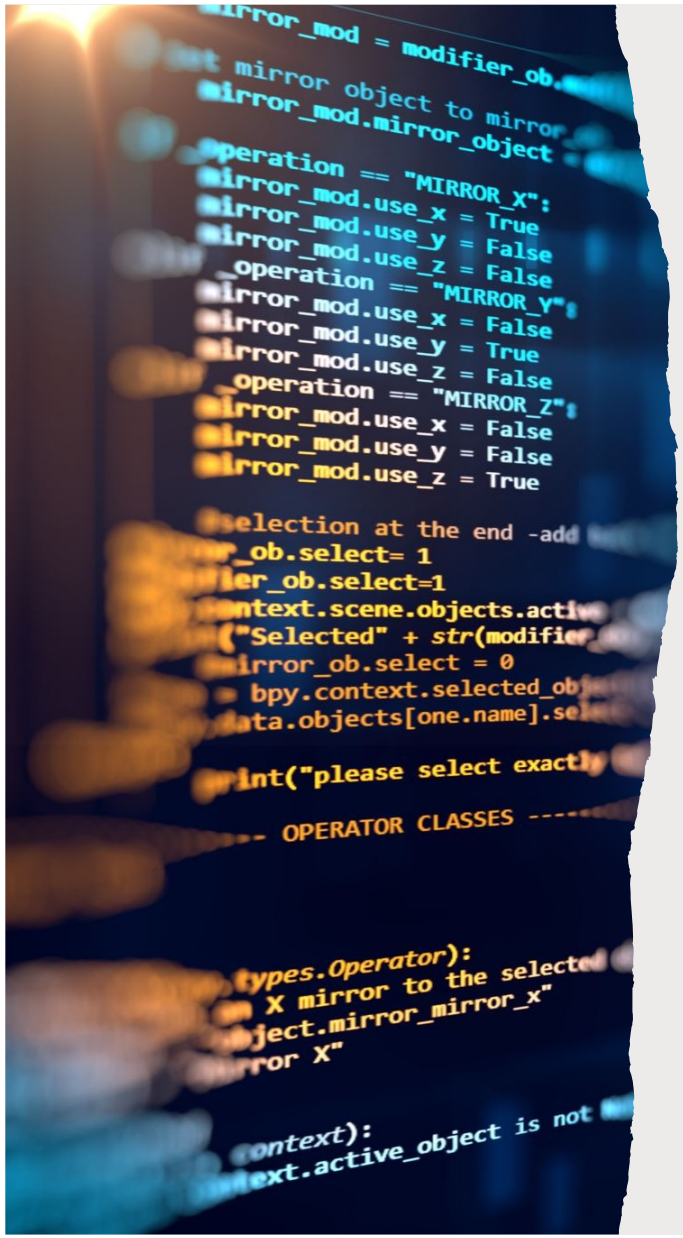
It is often described as a “distributed commit log” or more recently as a “distributing streaming platform.”

Developed in LinkedIn, 1st release was in 2011

The scalability of Kafka has helped LinkedIn’s usage grow in excess of seven trillion messages produced (as of February 2020) and over five petabytes of data consumed daily.

Kafka Fundamentals

Topics	Partition	Partition Offset	Replicas and partition
Brokers	Kafka Cluster	Producers	Cosumers
	Leader	Follower	



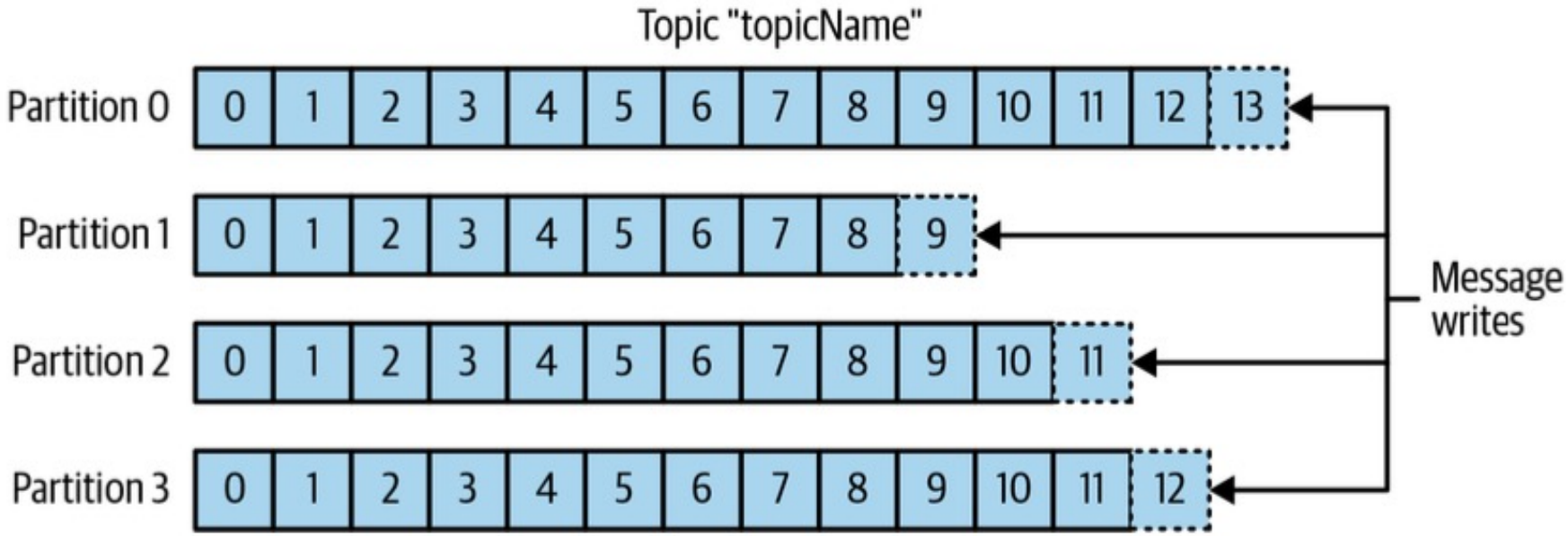
Schemas

- While messages are opaque byte arrays to Kafka
- JSON/XML are good and readable
  - Robust type handling
  - Compatibility between schema versions
- Consistent data format is important
- Decoupling between reading and writing
- Support multiple versions
- Central registry to store schema

Topics and Partitions

- Messages in Kafka are categorized into topics, like database table
- Topics are additional broken down to number of partitions
- Messages are written in append-only fashion and read in order from beginning to end
- No guarantee of message ordering across entire topic in multiple partiions
- Partitions provides redundancy and scalability
- Partitions can be hosted on different servers
- Single topic can scale horizontally across multiple servers

Partitions


Topic "topicName"
Partition 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13
Partition 1
0 1 2 3 4 5 6 7 8 9
Partition 2
0 1 2 3 4 5 6 7 8 9 10 11
Partition 3
0 1 2 3 4 5 6 7 8 9 10 11 12
Message writes

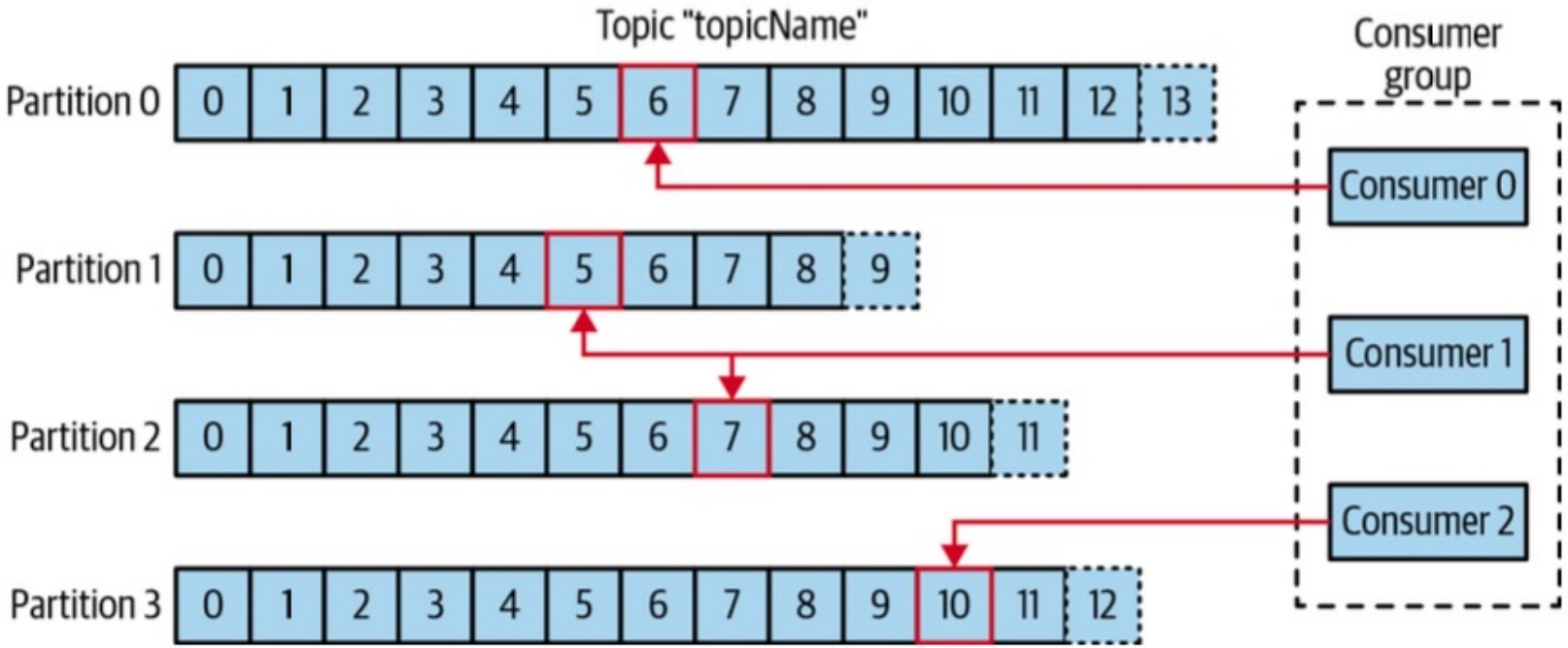
Producers and Consumers

- Kafka clients are users of the system
  - Producers and Consumers
- Advanced client APIs
  - Kafka Connect API
  - Kafka Streams
- Producer writes to topic, kafka balance messages over all partitions of topic evenly
- In some cases, the producer will direct messages to specific partitions.
  - Message key and a partitioner hash

Consumers

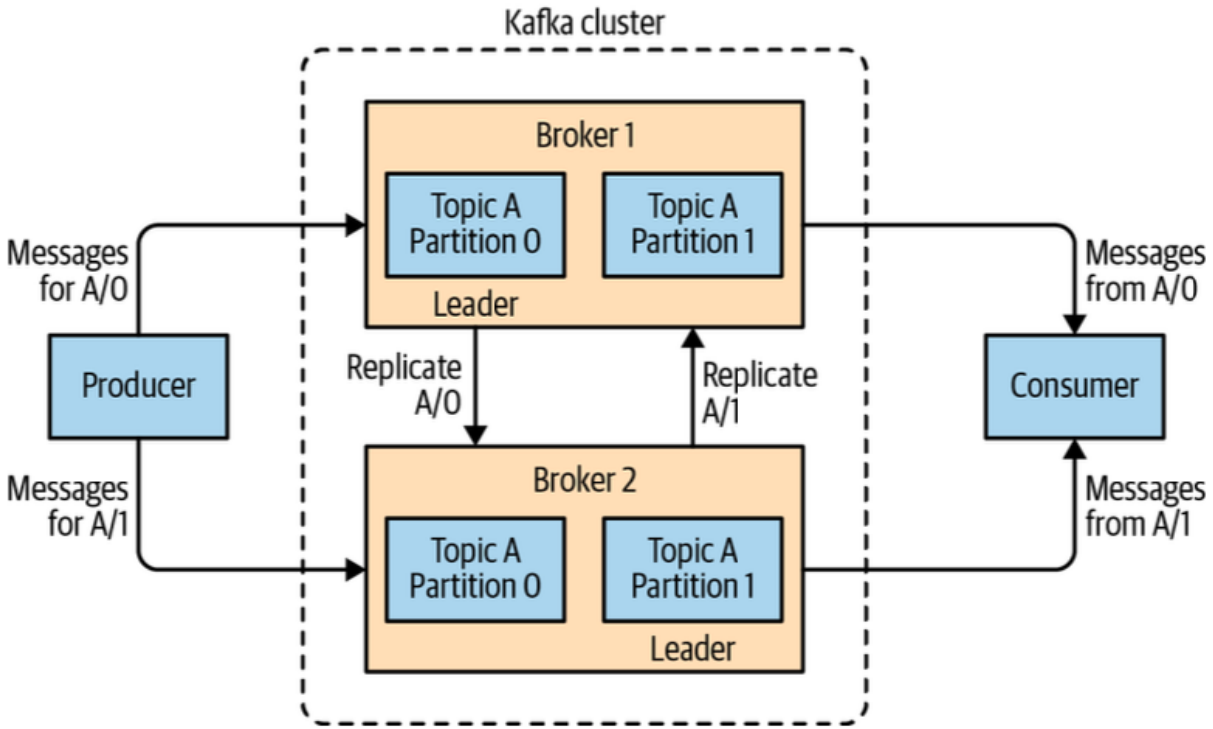
- Each partition has offset
- Each message produced adds up to offset of partition
- Kafka store offset of each consumer
- Consumer can stop and restart without losing its place
- Consumers works as part of Consumer Group
- Consumers can horizontally scale
- Partitions are reassigned between consumer group members

Consumer Groups



Brokers and Clusters

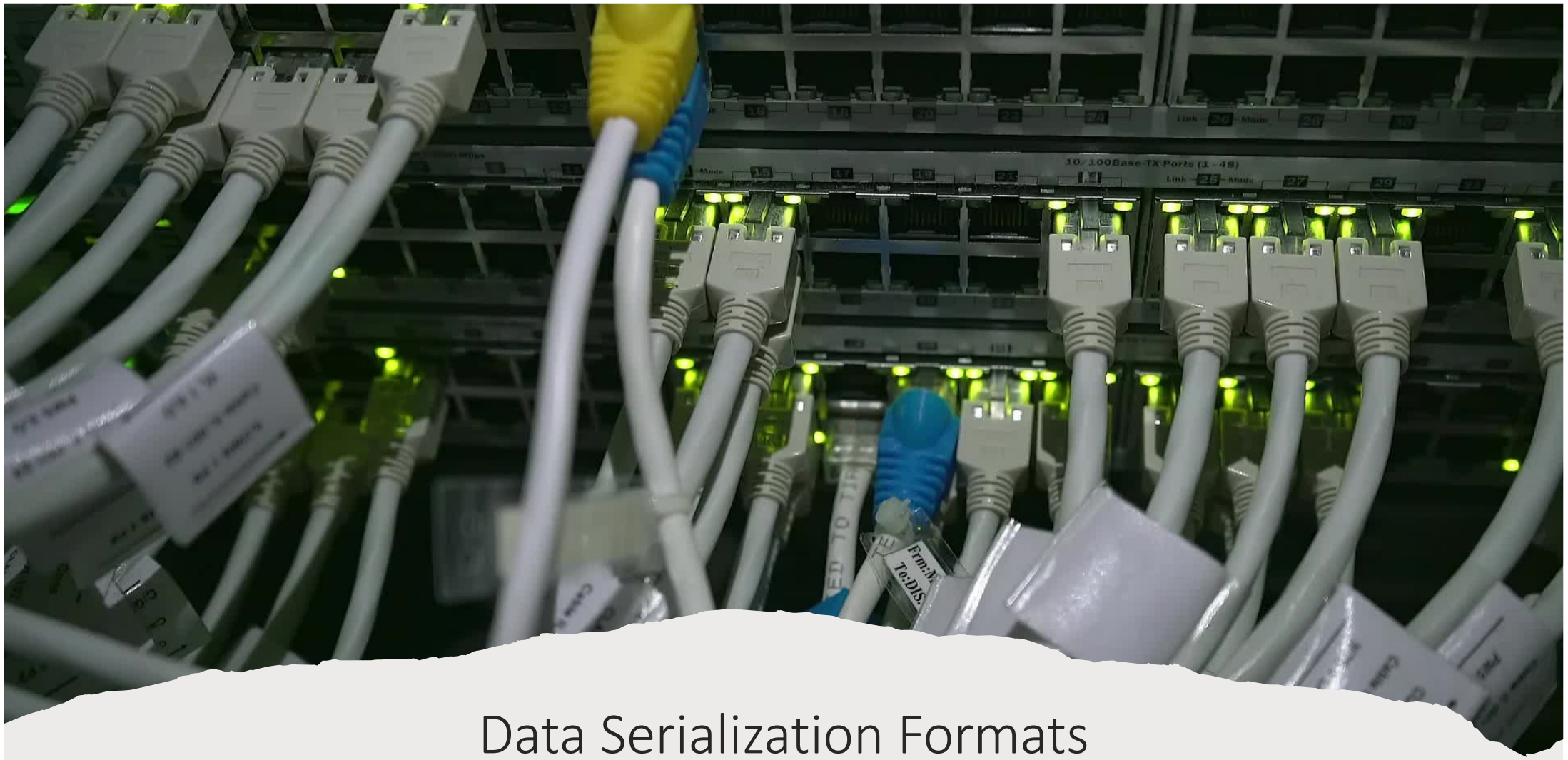
- Single Kafka server is called broker
- Broker recieves messages from producers and assign offsets to them and write it to disk
- a single broker can easily handle thousands of partitions and millions of messages per second.
- Message retention can be configured either days or amount of data
- Kafka brokers are designed to be a part of cluster
  - 1 Kafka broker as cluster controller (automatically elected)
  - Responsible for admistrative operations includes partition assigning, monitoring and broker failure



Brokers and Clusters

Why Kafka

- Multiple Producers
- Multiple Consumers
- Dish Based Retention
- Scalable
- High Performance



Data Serialization Formats

Data Serialization Formats

- Avro
- Protobuffs
- Parquet

Data Serialization Formats: Avro

- Apache project, comes from Hadoop, since 2009
- Supports C, C++, C#, Go, Haskell, Java, Perl, PHP, Python, Ruby, Scala, JavaScript
- Includes RPC specification
- Data is always accompanied by schema (in file header or protocol handshake)

Data Serialization Formats: Avro

```
{
    "type": "record",
    "name": "Person",
    "fields": [
      {"name": "userName", "type": "string"},
      {"name": "birthday", "type": ["null", "long"]},
      {"name": "friends",  "type": {"type": "array", "items": "string"}}
    ]
}
```

First Version:

```
{
    "type": "record",
    "name": "Person",
    "fields": [
      {"name": "userName", "type": "string"},
      {"name": "birthday", "type": ["null", "long"]},
      {"name": "friends",  "type": {"type": "array", "items": "string"}}
      {"name": "livesIn",  "type": ["string", "null"], "default": "World"},
    ]
}
```

Updated Version:

Data Serialization Formats: Parsing in Avro

- Fields are defined with their names, not numerical IDs
- There are no field IDs and no encoding of field IDs in binary representation
- Schema must be known before reading
- No optional fields
- instead use union of null and type

Data Serialization Formats: Schema Evolution in Avro

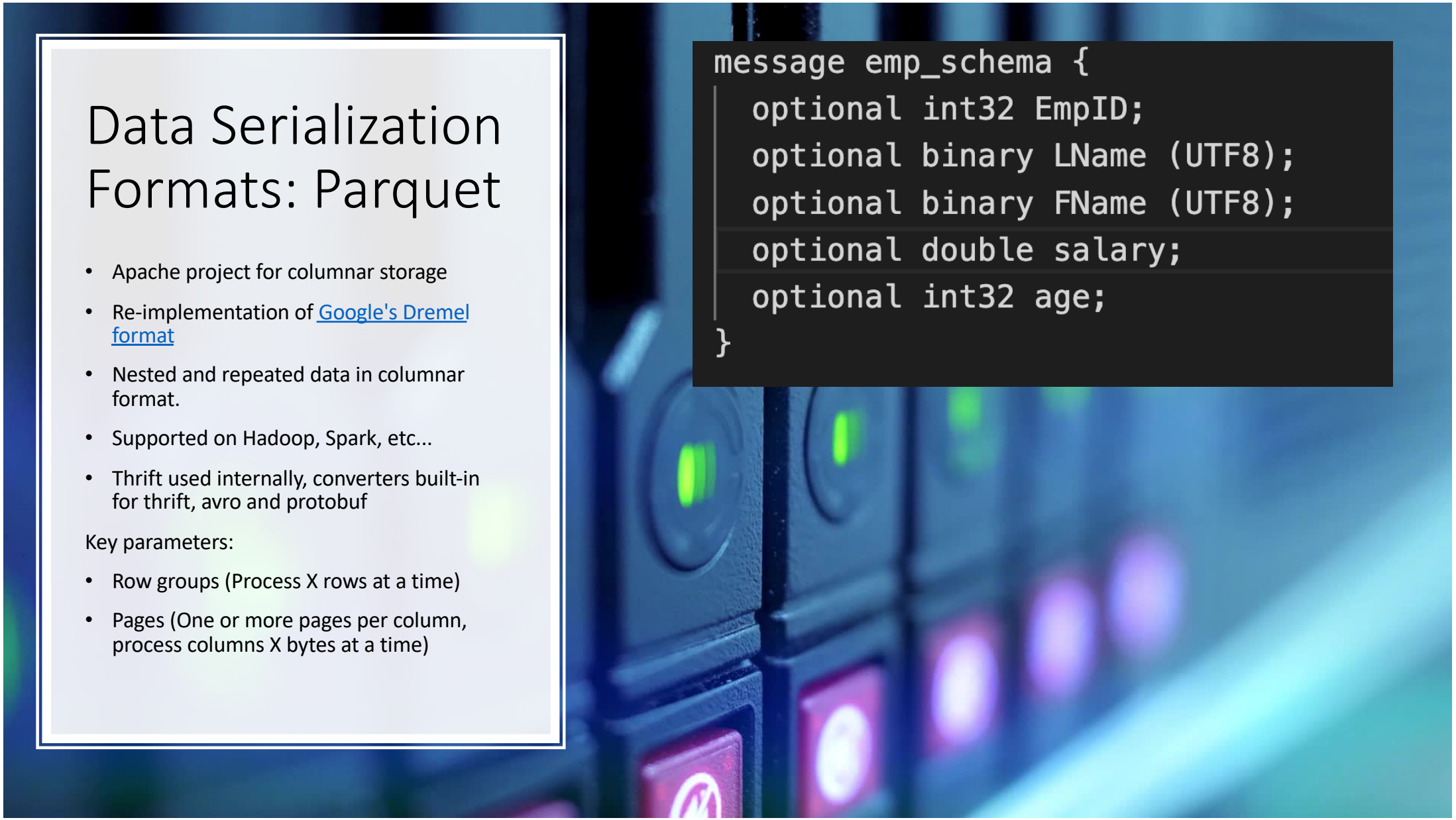
- Fields can be added only if they have a default value (null is OK)
- when reading old records with new reader, they will be assigned the default value
- Fields can be reordered, order is reconstructed from order in accompanying schema
- Changing field names is possible, but not easy
- likewise for adding types to unions and new values to Enums
- Schema resolution:
- writer's schema comes in file/schema registry
- reader's schema is encoded in generated code.
- Reader figures out what to do with difference between its own and writer's schema

Data Serialization Formats: Parquet

- Apache project for columnar storage
- Re-implementation of Google's Dremel format
- Nested and repeated data in columnar format.
- Supported on Hadoop, Spark, etc...
- Thrift used internally, converters built-in for thrift, avro and protobuf

Key parameters:

- Row groups (Process X rows at a time)
- Pages (One or more pages per column, process columns X bytes at a time)



```
message emp_schema {
  optional int32 EmpID;
  optional binary LName (UTF8);
  optional binary FName (UTF8);
  optional double salary;
  optional int32 age;
}
```

Data Serialization Formats: Parquet File Structure

- Parquet files consist of row groups - sets of rows processed together
- Larger row groups require more memory, could get better compression
- Row groups contain columns one after another, wrapped in pages.
- Larger pages mean better compression BUT:
- Page size represents worst case size for reading single record
- If there is no compression, reading a single record requires reading full original size of data page
- Footer of the file contains metadata and statistics on all row groups and pages
- Sometimes reading the metadata and statistics can short-circuit reading any block from the file.

Data Serialization Formats: Protocol Buffers

- Comes from Google, available since 2007-08
- Directly supports Java, C++, Python, Go, Dart, Ruby, JavaScript, Objective-C, and C#
- Lots of 3rd party implementations
- Home page
- Core includes just data serialization, RPC framework is gRPC
- Versions 2 and 3, both active and supported
- v3 recommended for gRPC
- v3 removes optional/required distinction
- All fields in v3 are optional
- Up to readers to interpret what to do with empty fields

Data Serialization Formats: Protocol Buffers

```
message Person {
    string userName          = 1;
    int64  birthday          = 2;
    repeated string friends = 3;
}
```

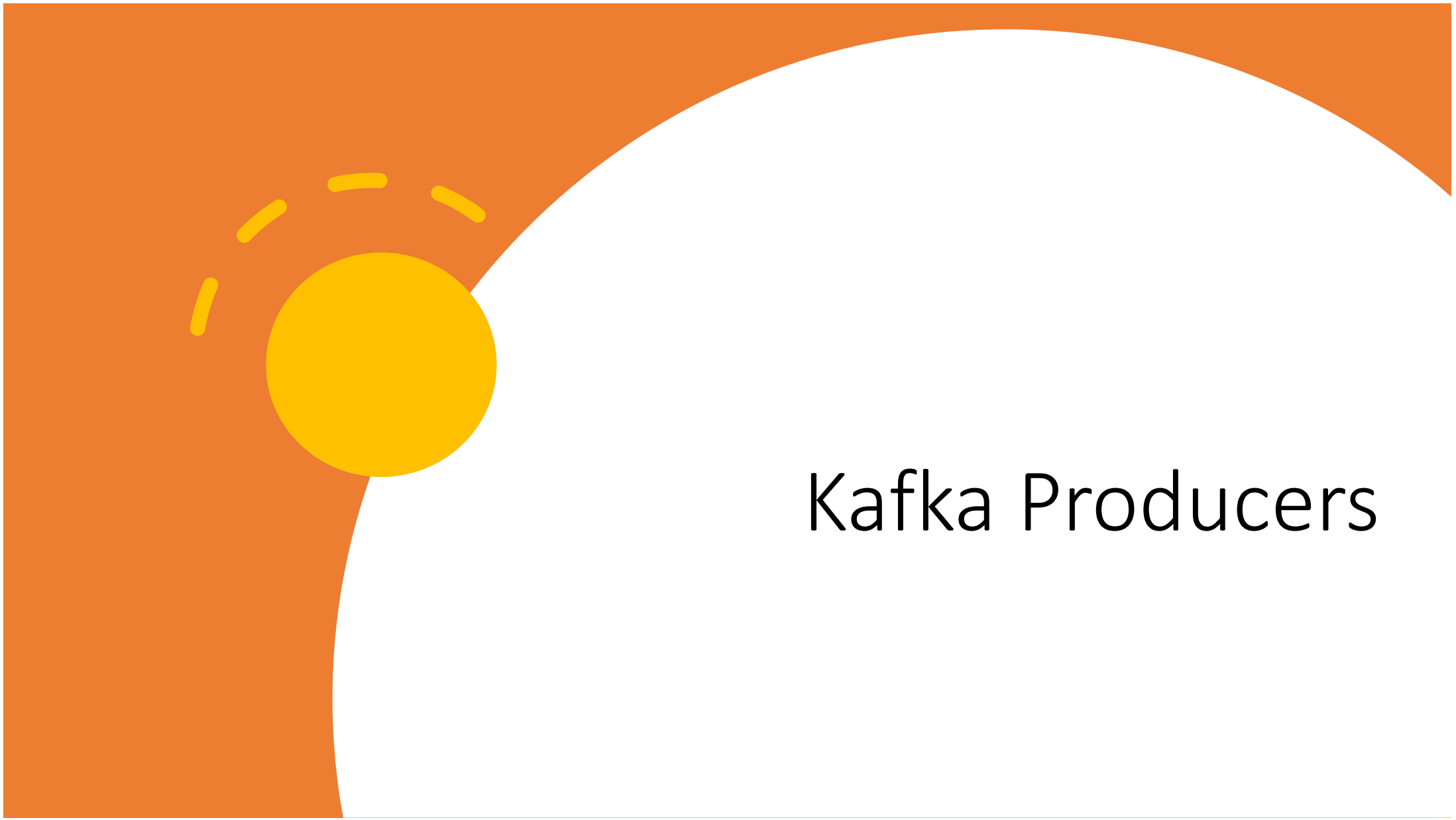
First Version:

```
message Person {
    string userName          = 1;
    int64  birthday          = 2;
    repeated string friends = 3;
    string livesIn           = 4;
}
```

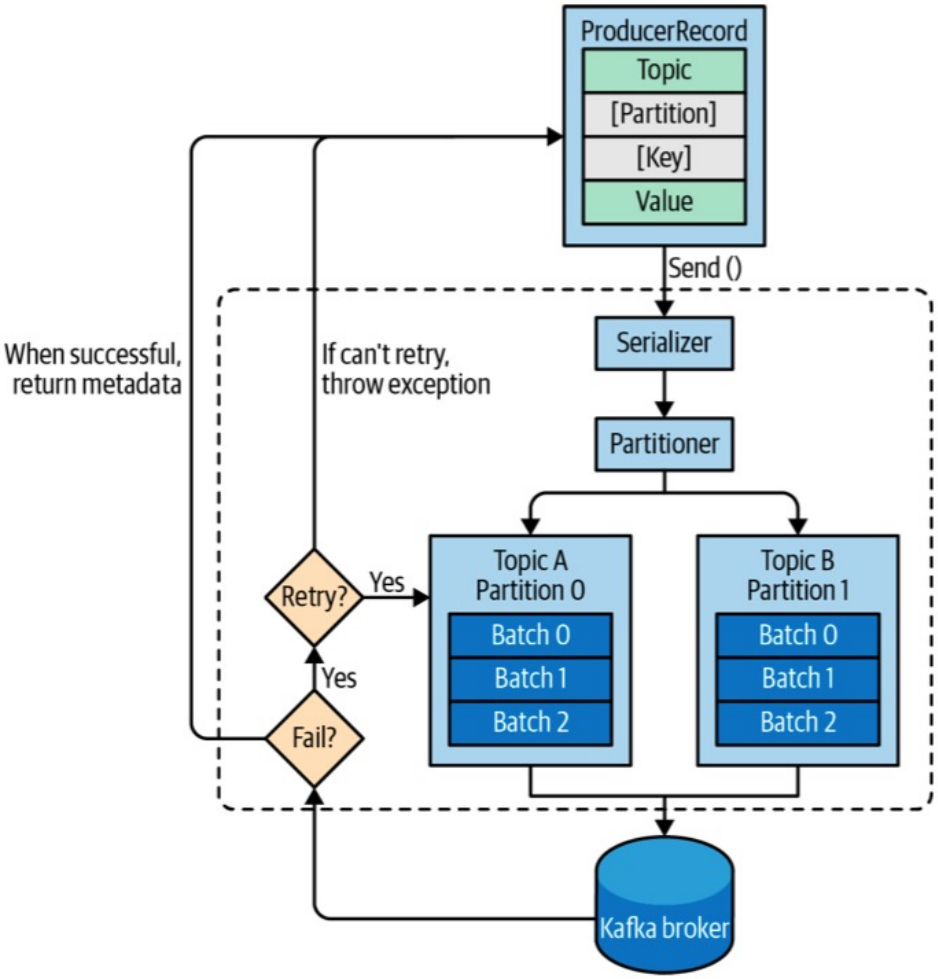
Updated Version:

Parsing and schema evolution in Protobuf

- Numerical field IDs and their types are encoded into binary message
- Unknown field IDs are ignored by reader =>
  - new fields can be added
- Writer's schema is not needed on read side to do basic parsing =>
  - unknown/new fields are skipped
- Optional fields can be removed
- Fields can be renamed, as the ID is key
- IDs can not be reused


Kafka Producers

Kafka Producers: Writing messages to Kafka



```
ProducerRecord<String, String> record =
new ProducerRecord<>("CustomerCountry", "Precision Products", "France");
try {
    producer.send(record).get();
} catch (Exception e) {
    e.printStackTrace();
}
```

Consutructing a Kafka Producer

- bootstrap.servers
- key.serializer
- Value.serializer
- Fire-and forget
  - Synchronous send
  - Asynchronous send

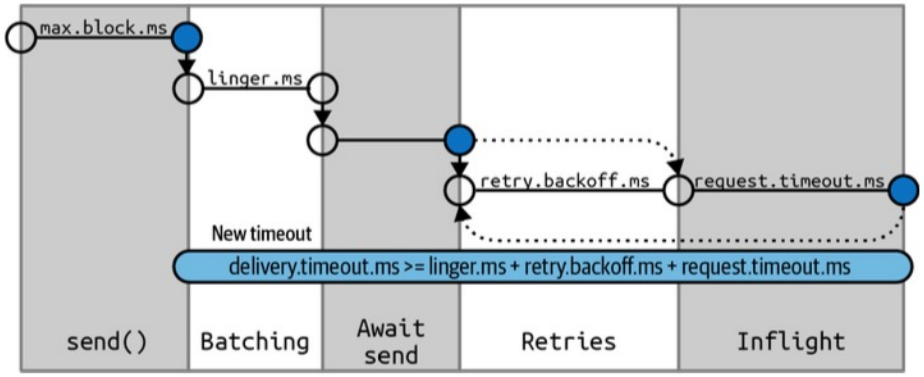
Sending a Message Asynchronously

- To use callbacks, you need a class that implements the *.Callback interface,
- If Kafka returned an error, onCompletion() will have a nonnull exception.
- The records are the same as before.
- And we pass a Callback object along when sending the record.

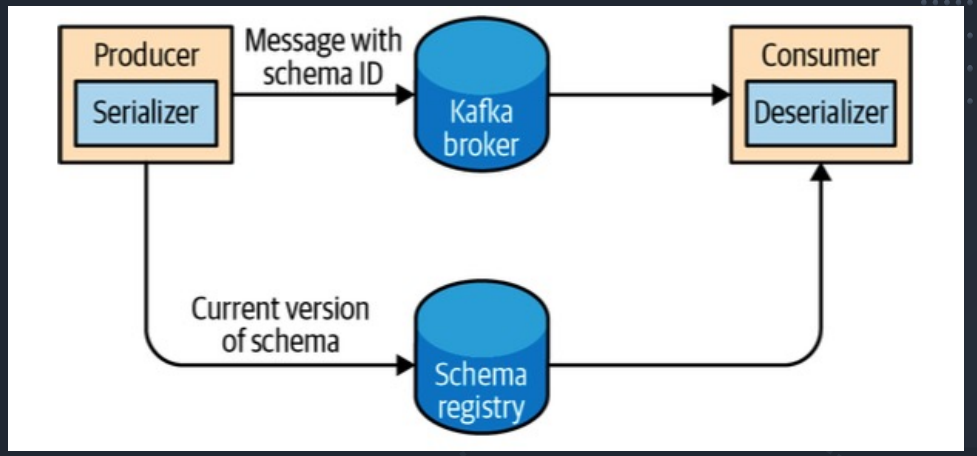
```
private class DemoProducerCallback implements Callback {
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null) {
            e.printStackTrace();
        }
} }
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");
producer.send(record, new DemoProducerCallback());
```

Configuring Producers

- Client.id
- Acks
  - Acks=0
  - Acks=1
  - Acks=all
- Max.block.ms
- Delivery.timeout.ms
- Request.timeout.ms
- Retry.backoff.ms
- Linger.ms
- Buffer.memory
- Compression.type
- Batch.size
- …



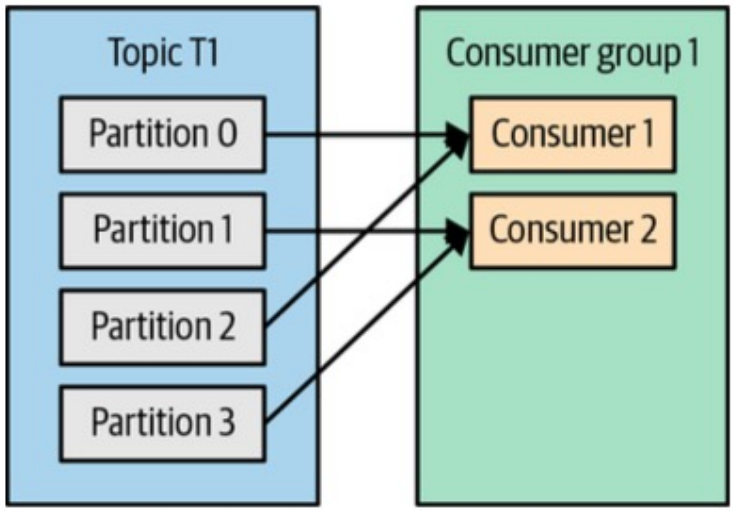
```
Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092");
    props.put("key.serializer",
       "io.confluent.kafka.serializers.KafkaAvroSerializer");
    props.put("value.serializer",
       "io.confluent.kafka.serializers.KafkaAvroSerializer");
    props.put("schema.registry.url", schemaUrl);
    String topic = "customerContacts";
    Producer<String, Customer> producer = new KafkaProducer<>(props);
    // We keep producing new events until someone ctrl-c
    while (true) {
        Customer customer = CustomerGenerator.getNext();
        System.out.println("Generated customer " +
            customer.toString());
        ProducerRecord<String, Customer> record =
            new ProducerRecord<>(topic, customer.getName(), customer);
        producer.send(record);
}
```

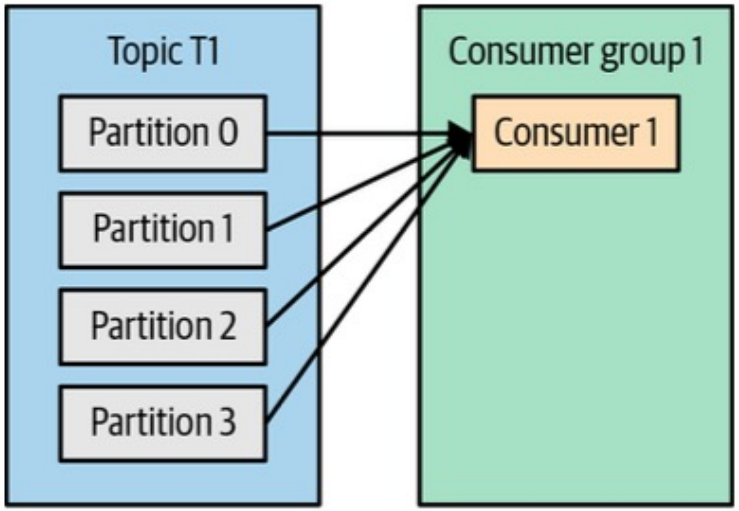


Using Avro records with Kafka

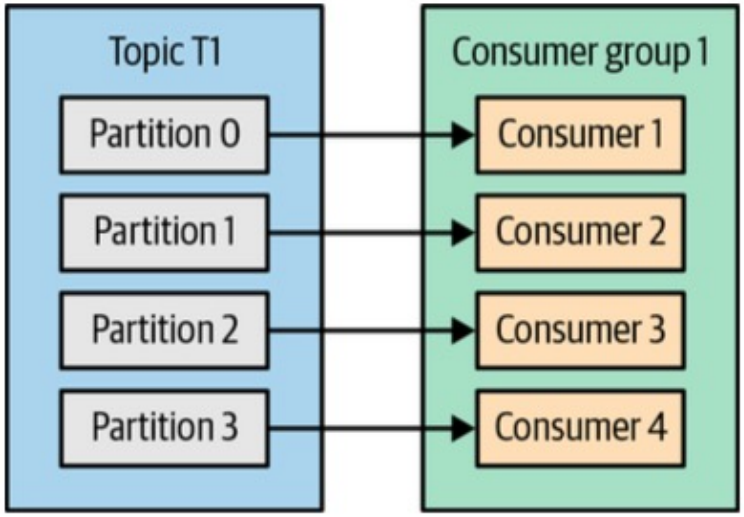

Kafka Consumers

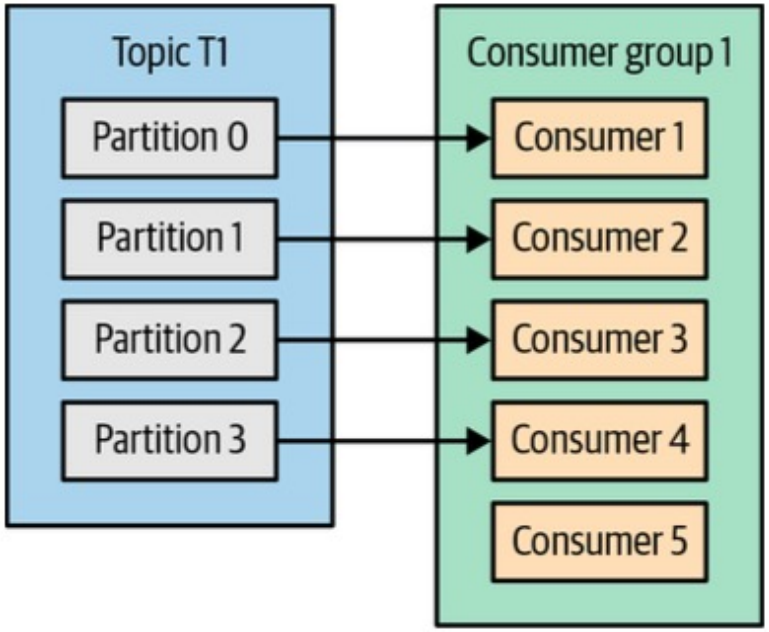
Kafka Consumers: Reading data from Kafka





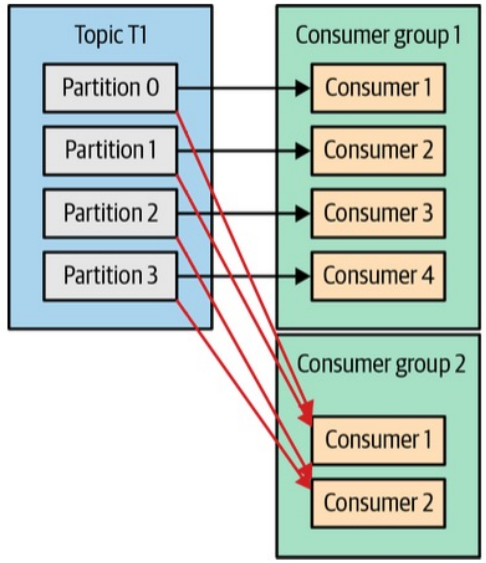
Kafka Consumers: Reading data from Kafka





Kakfa Consumers: Reading data from Kafka

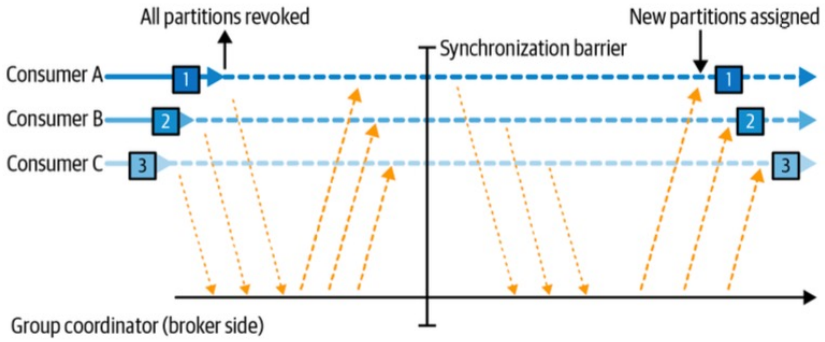
- Multiple applications
- Multiple groups
  - Multiple consumers
  - Consumer group 1
  - Consumer group 2



Consumer Groups and Partition Rebalance

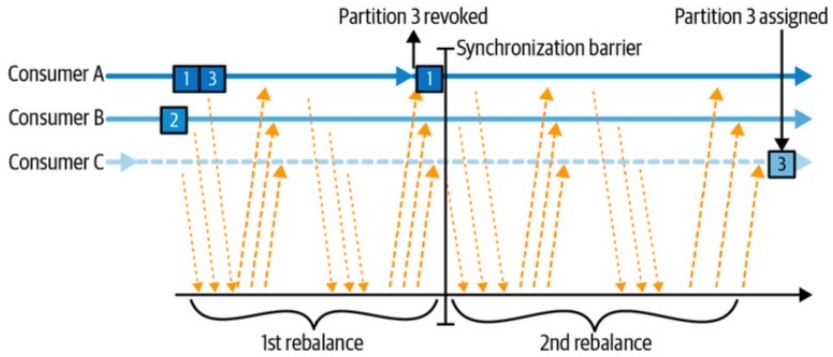
- Consumers in 1 group shares the partitions
- Consumer leave others take care of the partition
- Resassignment in partitions also happens when group is modified
- **Moving partition ownership from one cosumer to another is called rebalance.**
- Two typs of rebalances:
  - Eager rebelances
  - Cooperative relabances

Partition Rebalance: Eager rebalances



- Consumers stop consuming
- Give up their ownership
- Rejoin consumer group
- Get a brand new partition assignment
- Essentially short window of unavailability of entire consumer group

Partition Rebalance: Cooperative rebalances



- Also called incremental rebalances
- Involves reassigning only small subset of partitions
- Allow consumers to continue processing records
- Achieved by rebalancing in two or more phases

```
Properties props = new Properties();
    props.put("bootstrap.servers", "broker1:9092,broker2:9092");
    props.put("group.id", "CountryCounter");
    props.put("key.deserializer",
        "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
        "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer =
        new KafkaConsumer<String, String>(props);
// Single topic
consumer.subscribe(Collections.singletonList("customerCountries"));
//Multiple topics with wildcard
consumer.subscribe(Pattern.compile("test.*"));

Duration timeout = Duration.ofMillis(100);
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %d, offset = %d, " +
                              "customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(),
                    record.key(), record.value());
            int updatedCount = 1;
            if (custCountryMap.containsKey(record.value())) {
                updatedCount = custCountryMap.get(record.value()) + 1;
            }
            custCountryMap.put(record.value(), updatedCount);
            JSONObject json = new JSONObject(custCountryMap);
            System.out.println(json.toString());
        }
}
```

Commits and Offsets

Consumers to use Kafka to track their position (offset) in each partition.

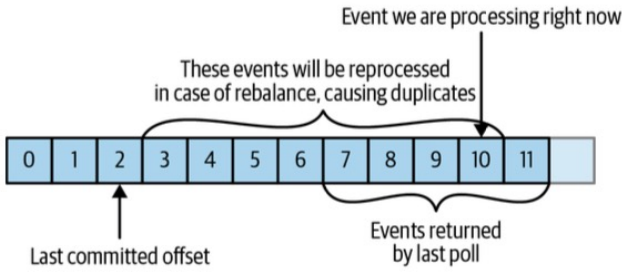
Offset Commit:

Updating the current position in the partition

How does a consumer commit an offset?

__consumer_offsets topic with the committed offset for each partition

Consumer crashes or a new consumer joins the group, *trigger a rebalance*



Kafka Handson

Let’s get our hands dirty with code

https://github.com/muhammadarslan/kafka-handson-workshop

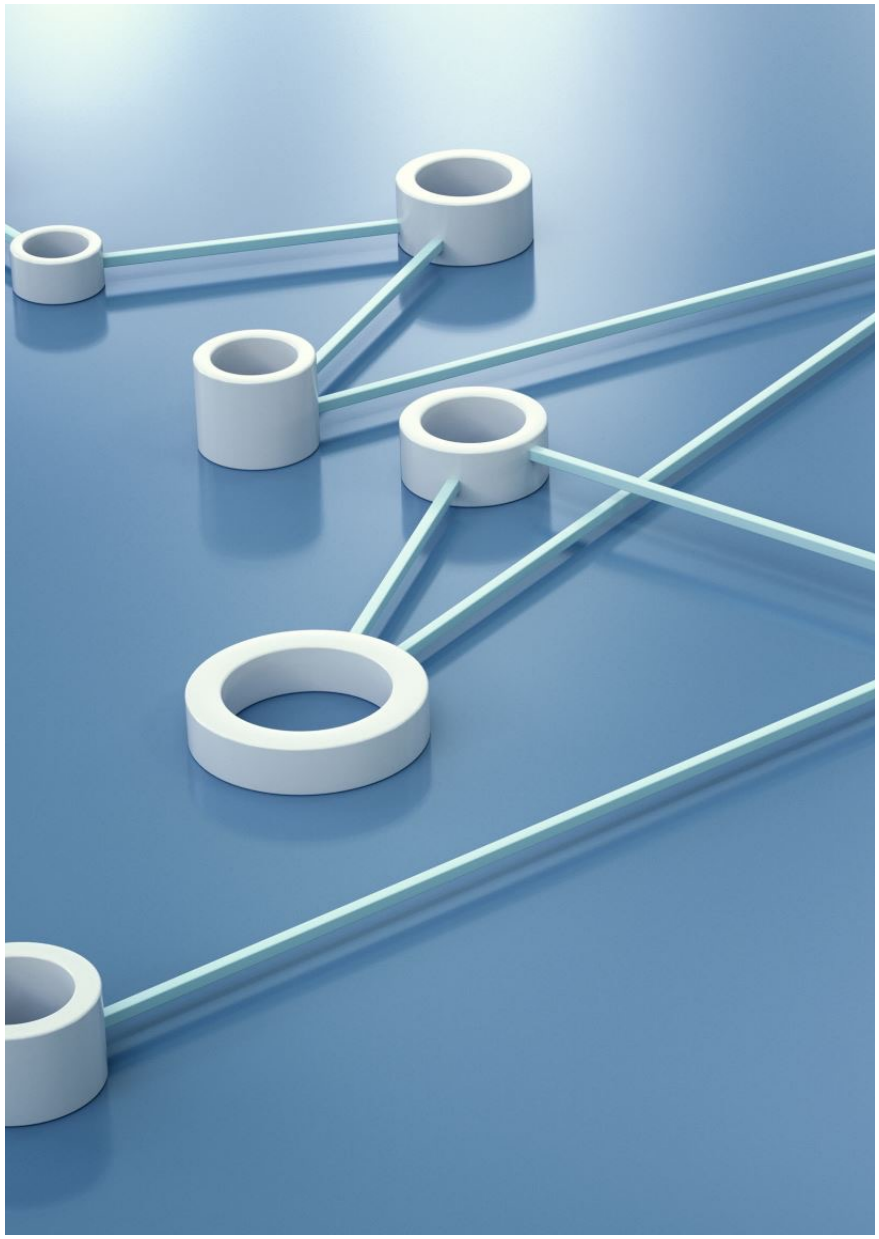

Kafka Streams

Kafka Streams

- First and foremost, a *data stream* is an abstraction representing an unbounded dataset. *Unbounded* means infinite and ever growing.
- The dataset is unbounded because over time, new records keep arriving. This definition is used by Google, Amazon, and pretty much everyone else.

Kafka Streams

- *Event streams are ordered*
- *Immutable data records*
- Request/Response
- Batch Processing
- Stream Processing

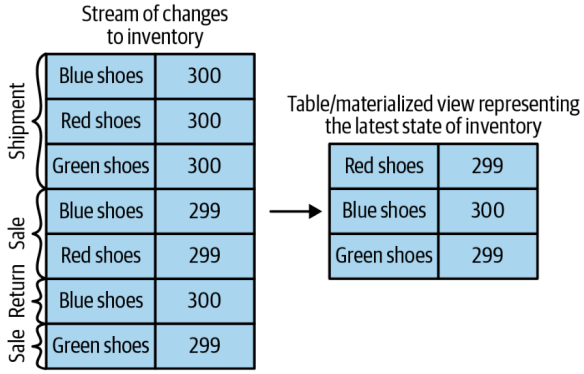


Stream Processing Concepts

- Topology
- Time
  - Event Time
  - Log Append Time
  - Processing Time
- State
- Stream-Table Duality
- Time Windows

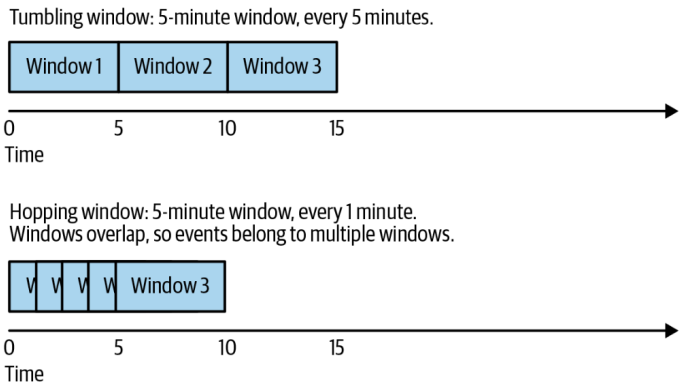
Stream Table Duality

- “Shipment arrived with red, blue, and green shoes.”
- “Blue shoes sold.”
  “Red shoes sold.”
  “Blue shoes returned.”
- “Green shoes sold.”

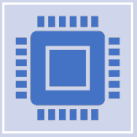


Time Windows

- Size of windows
- *How often the window moves (advance interval)*
- *How long the window remains updatable (grace period)*

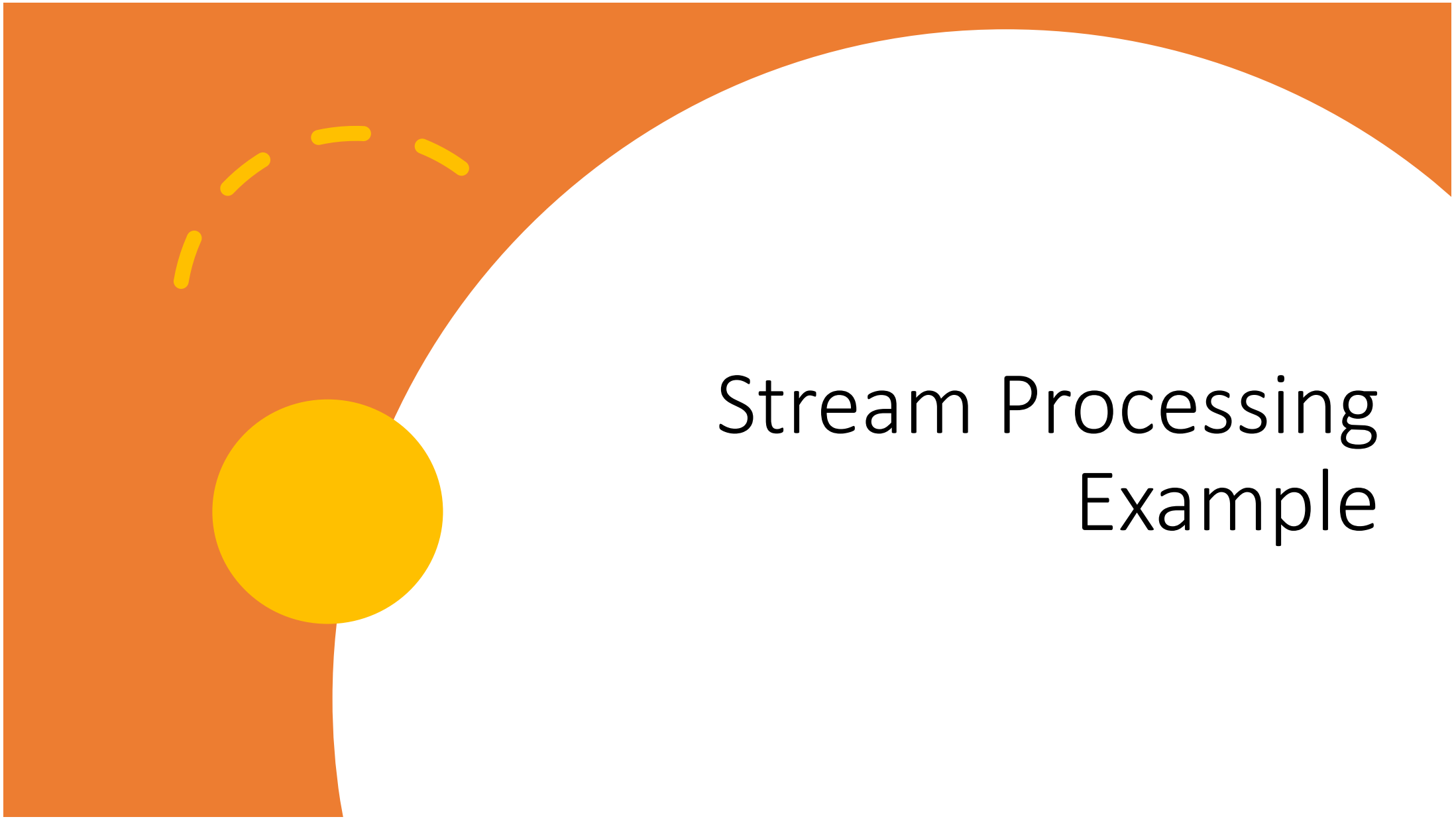


Processing Guarantees



A key requirement for stream processing applications is the ability to process each record exactly once, regardless of failures.

Without exactly-once guarantees, stream processing can’t be used in cases where accurate results are needed.



Stream Processing Example

Thank You!