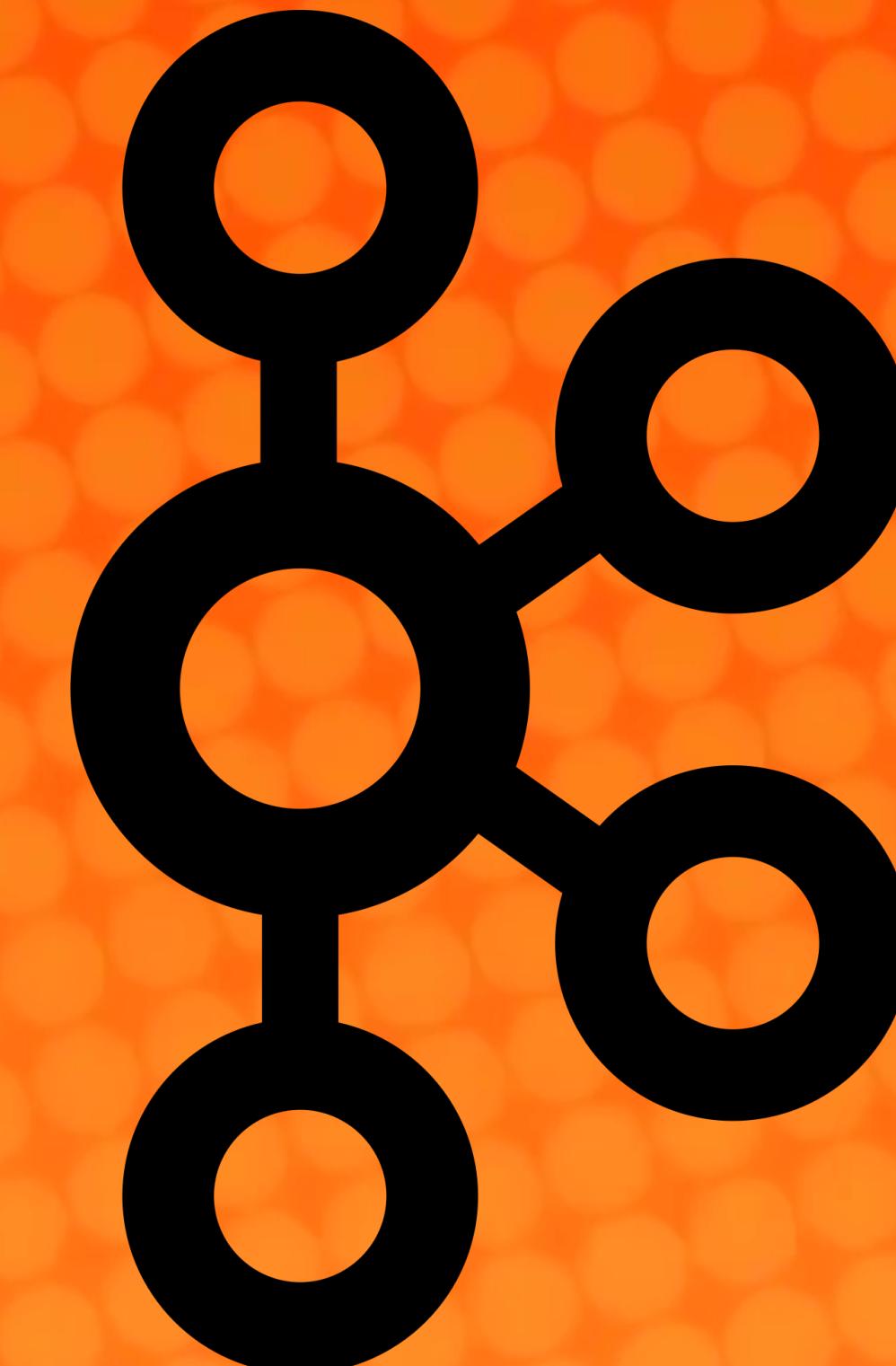


Apache Kafka Internals

Daniel Hinojosa



About Me...

Daniel Hinojosa

Programmer, Consultant, Trainer

Testing in Scala (Book)

Beginning Scala Programming (Video)

Scala Beyond the Basics (Video)

Contact:

dhinojosa@evolutionnext.com

@dhinojosa

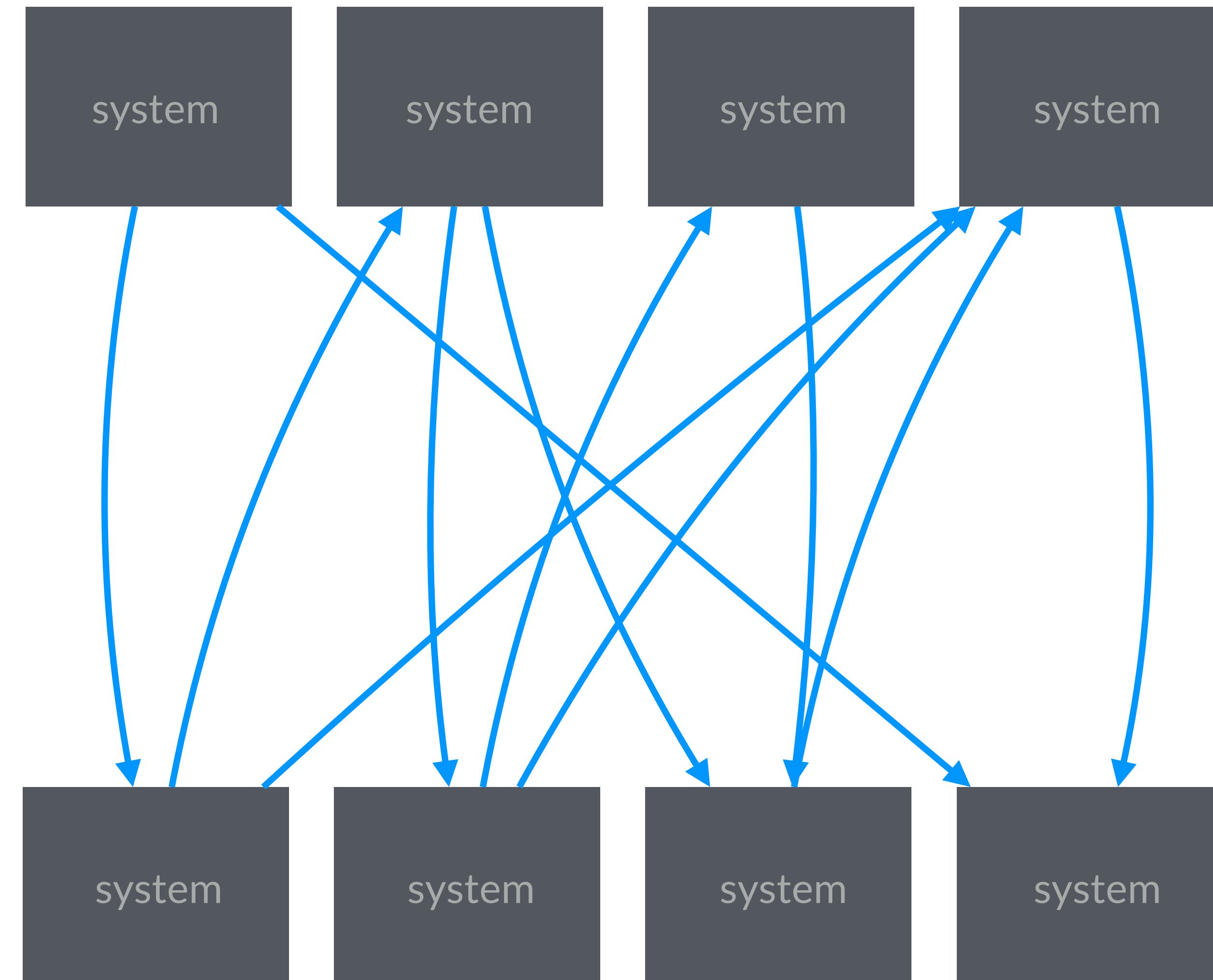


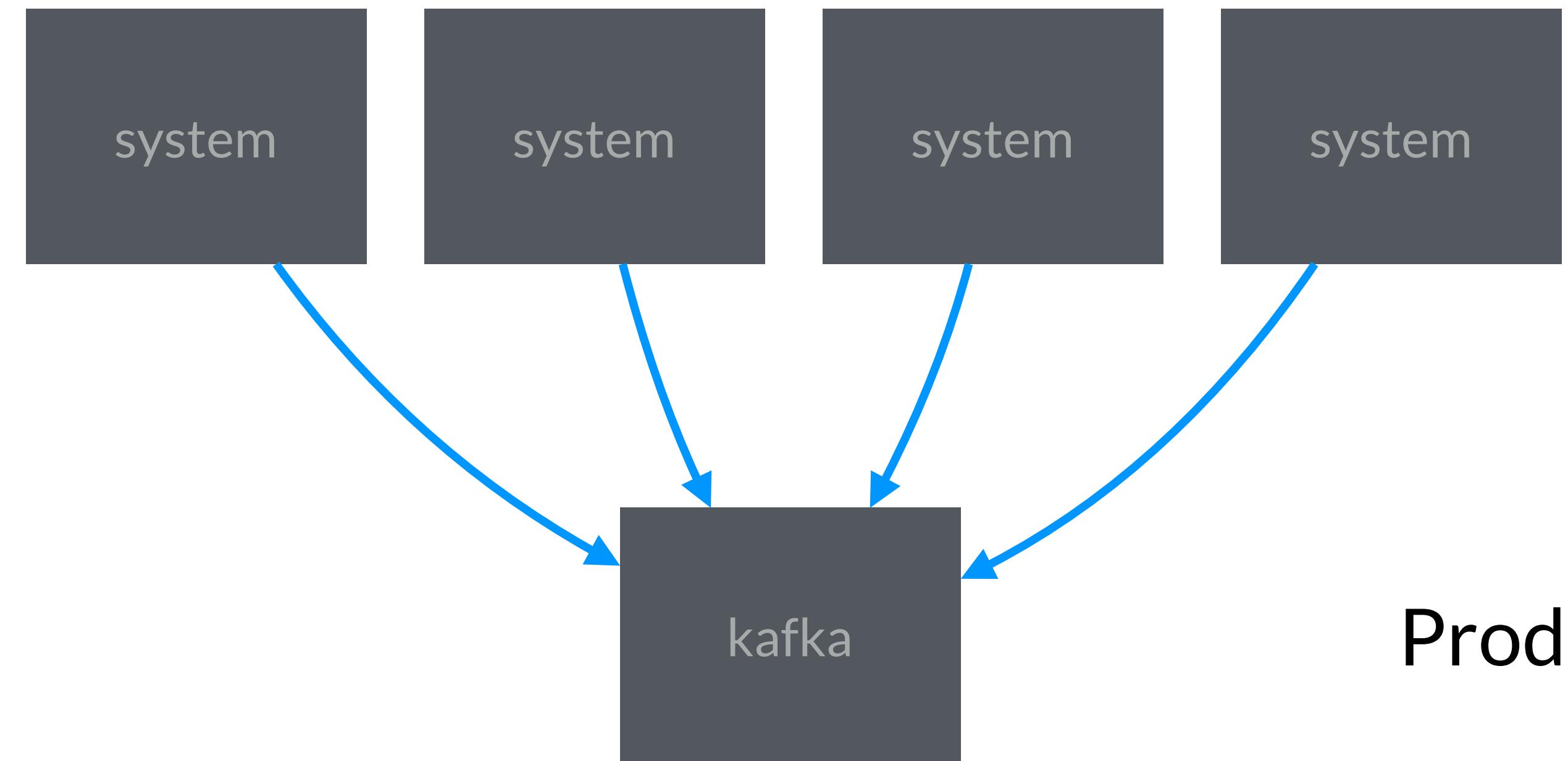
Agenda

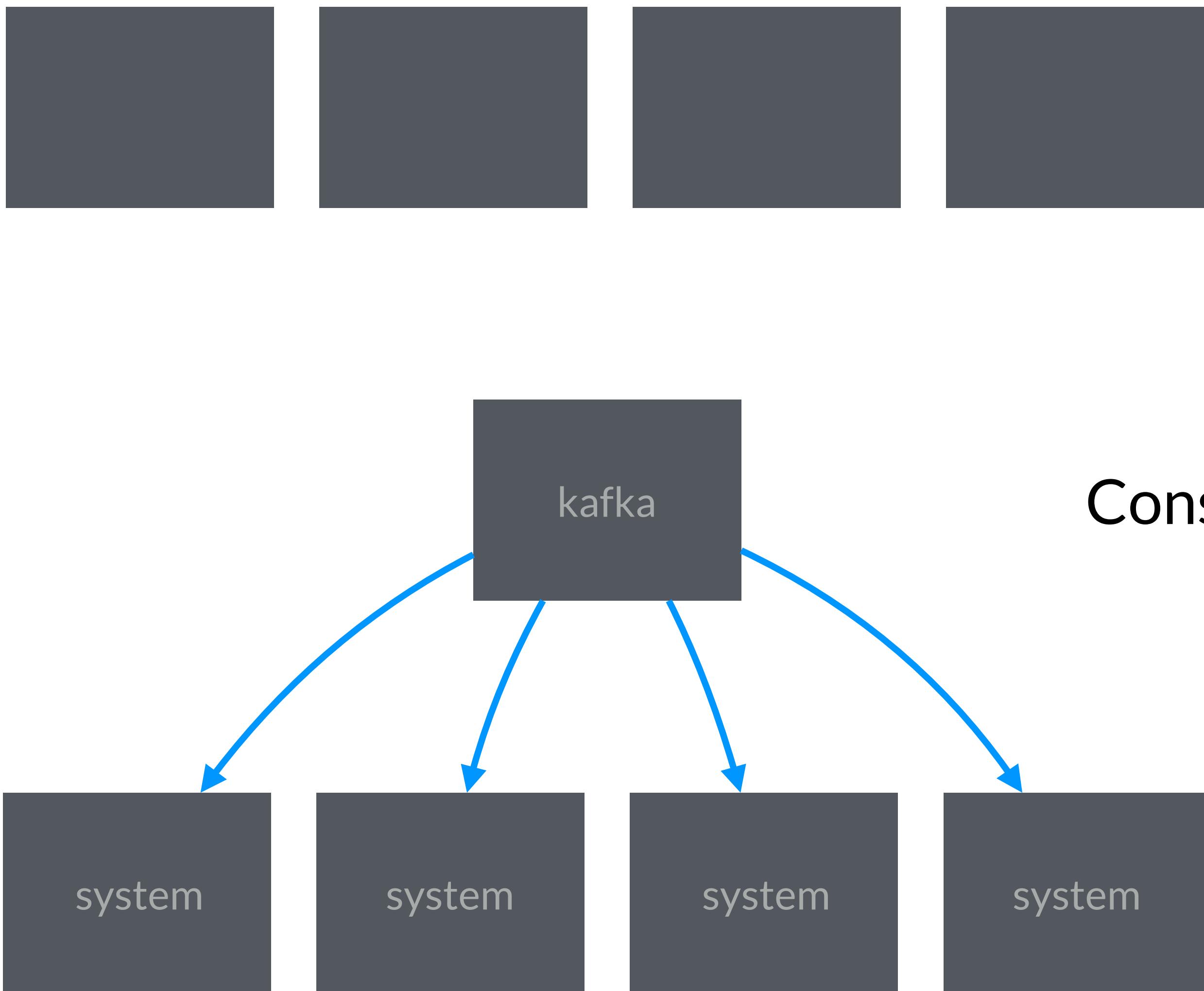
Agenda

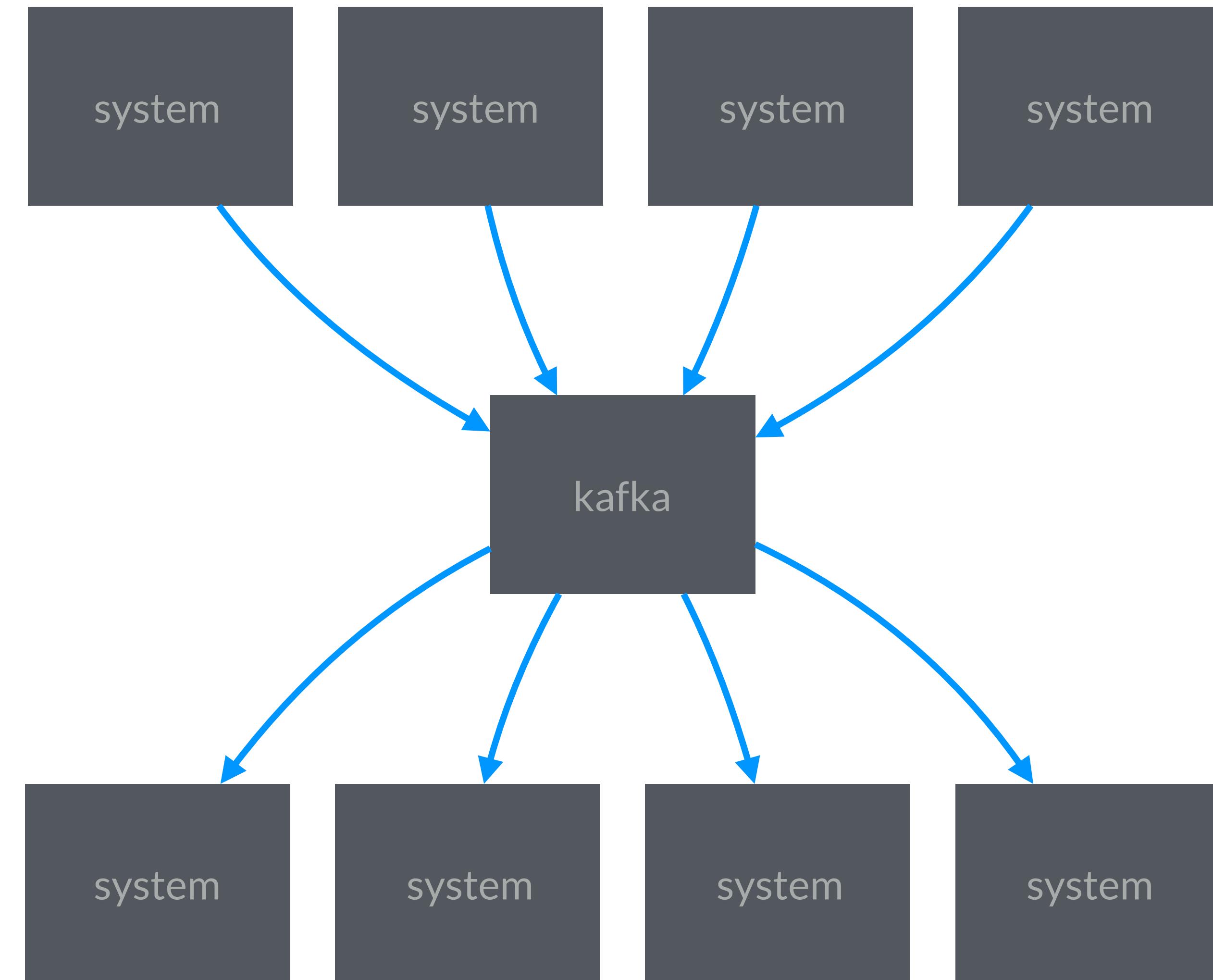
- ★ Understand Kafka
- ★ Understanding K8s Operators
- ★ Understanding Strimzi
- ★ Understand Producer
- ★ Understand Consumer
- ★ Understand Rebalancing
- ★ Understand Replication
- ★ Understand Common Properties
- ★ Understand Stream v Tables
- ★ Understanding Kafka Connect
- ★ Understanding Monitoring
- ★ Understanding Security
- ★ Performance Tuning
- ★ Understanding KSQLDB

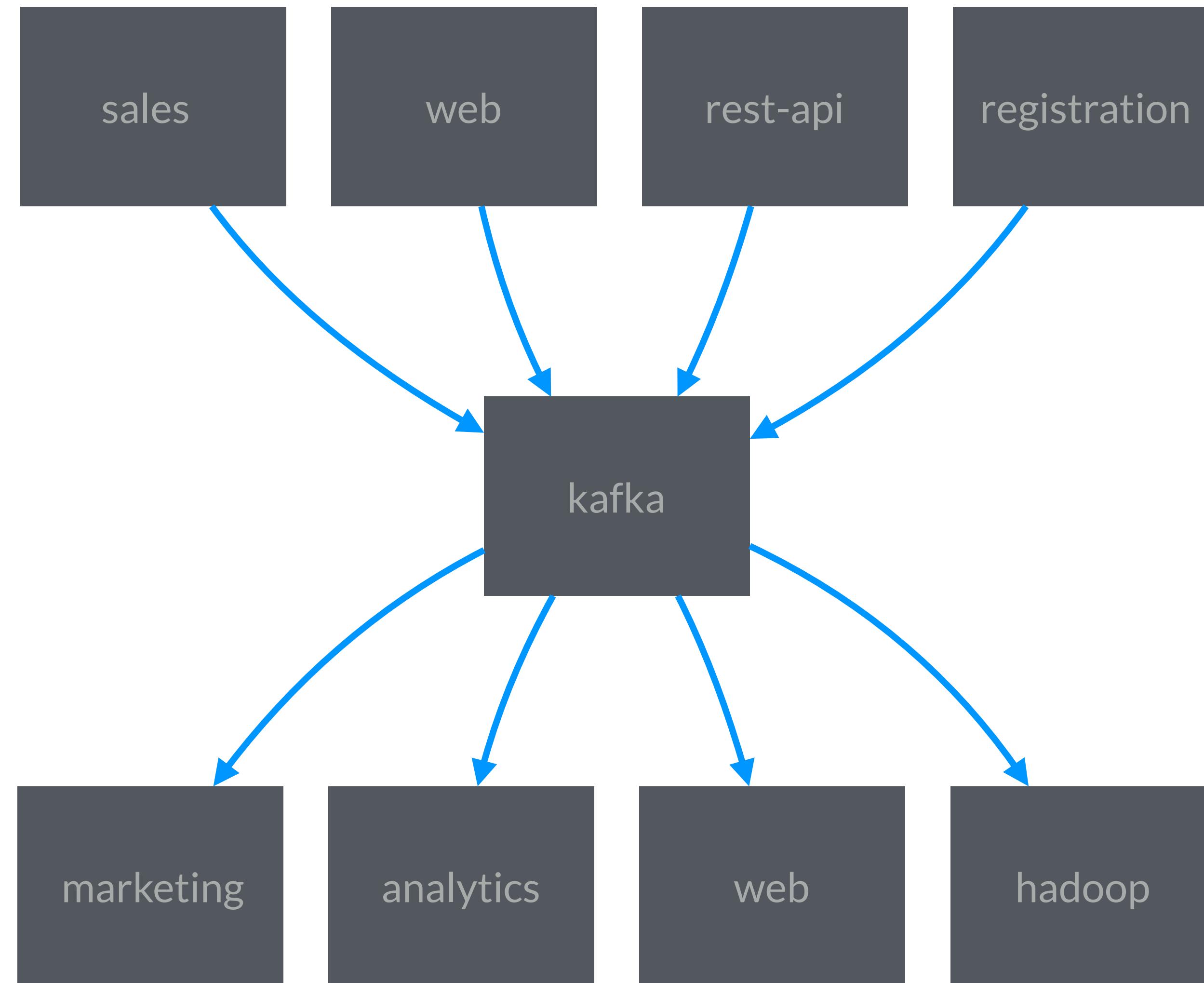
Kafka Introduction











Note: A Producer can be a Consumer

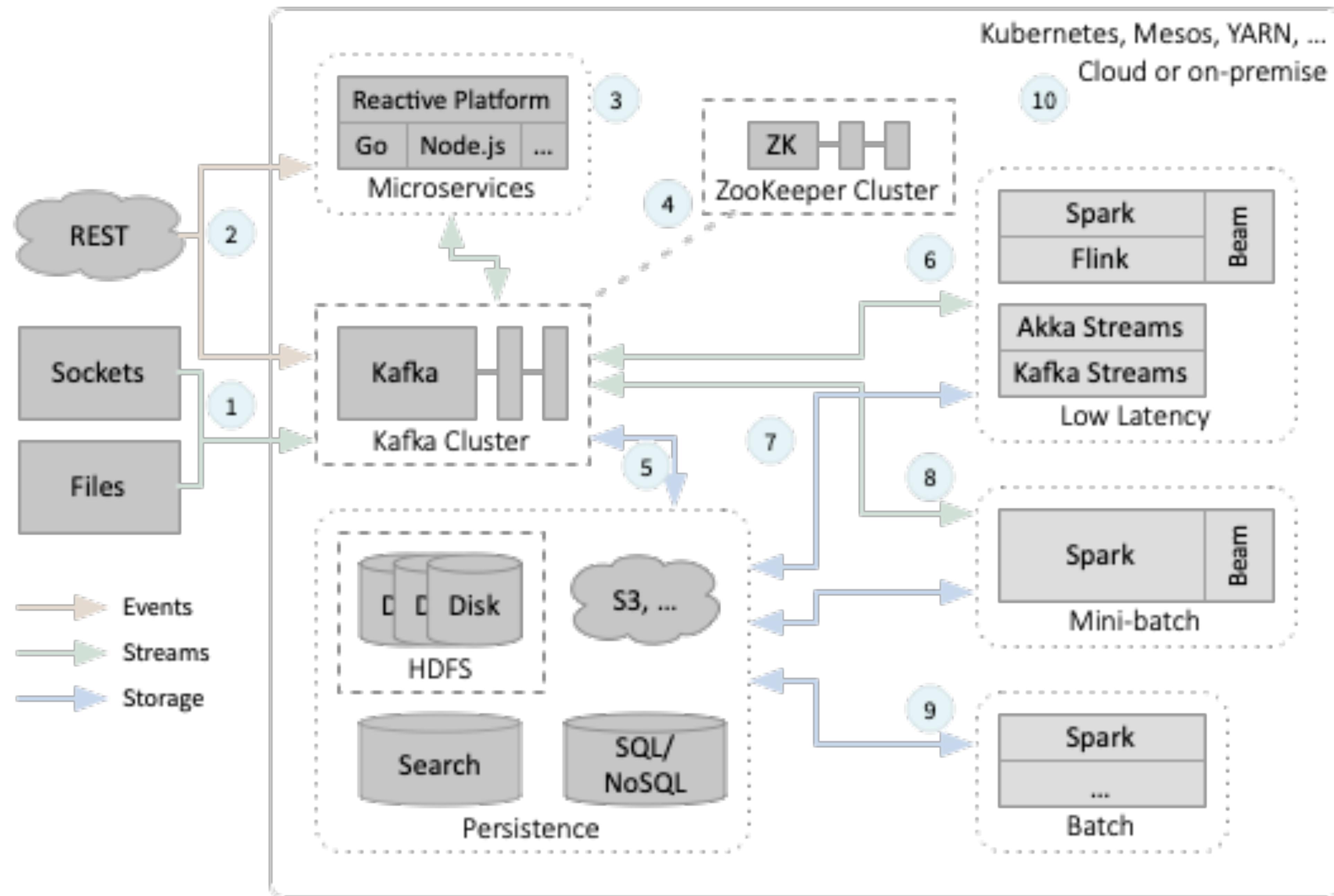
About Kafka

- Handles millions of messages per second, high throughput, high volume
- Distributed and Replicated Commit-log
- Real Time Data Processing
- Stream processing

Kafka Use-Cases

- Activity Tracking
- Page Views
- Click Tracking
- Advertising Patronage
- Feed Artificial Intelligence
- Internet of Things

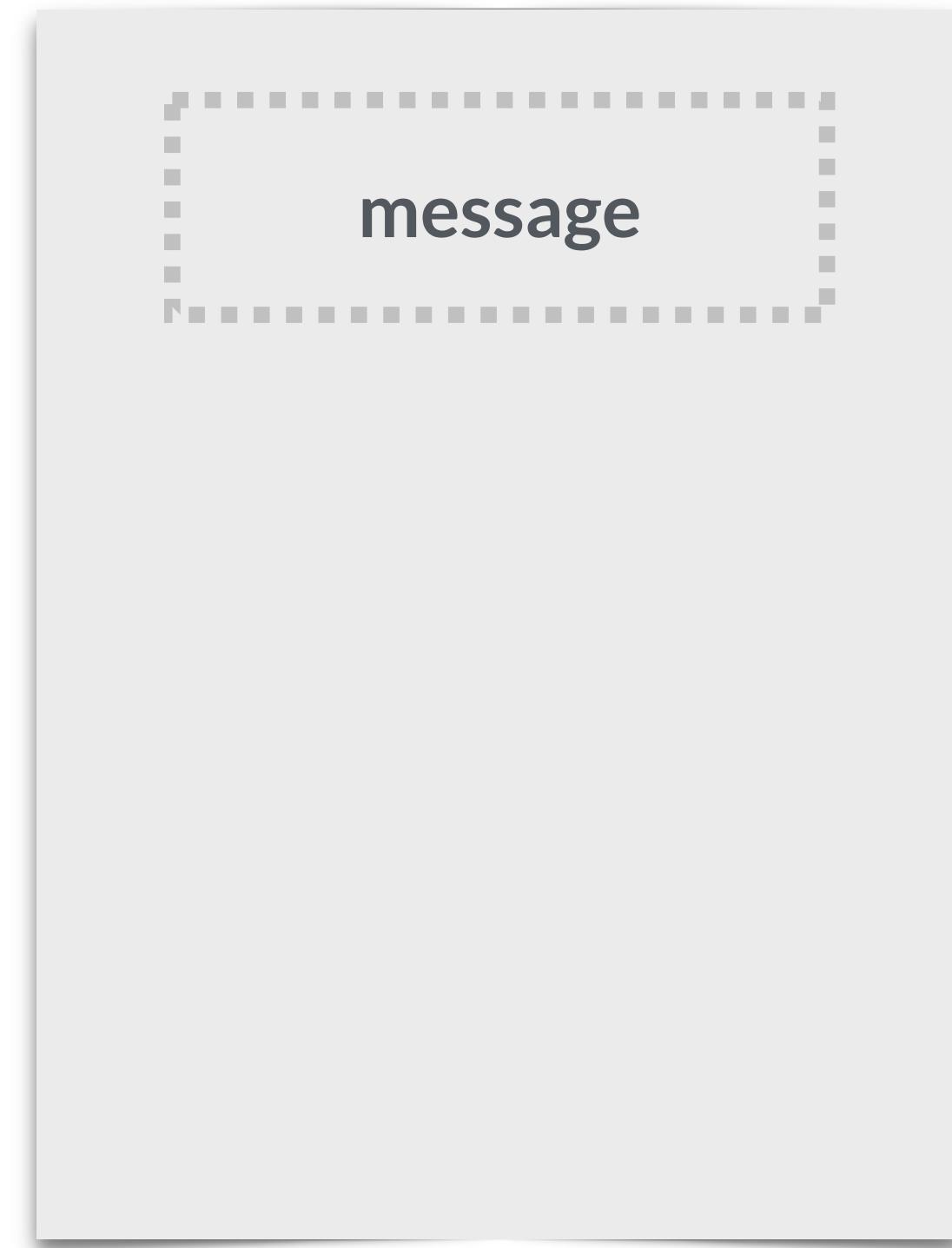
Dean Wampler Architecture



Kafka Messages

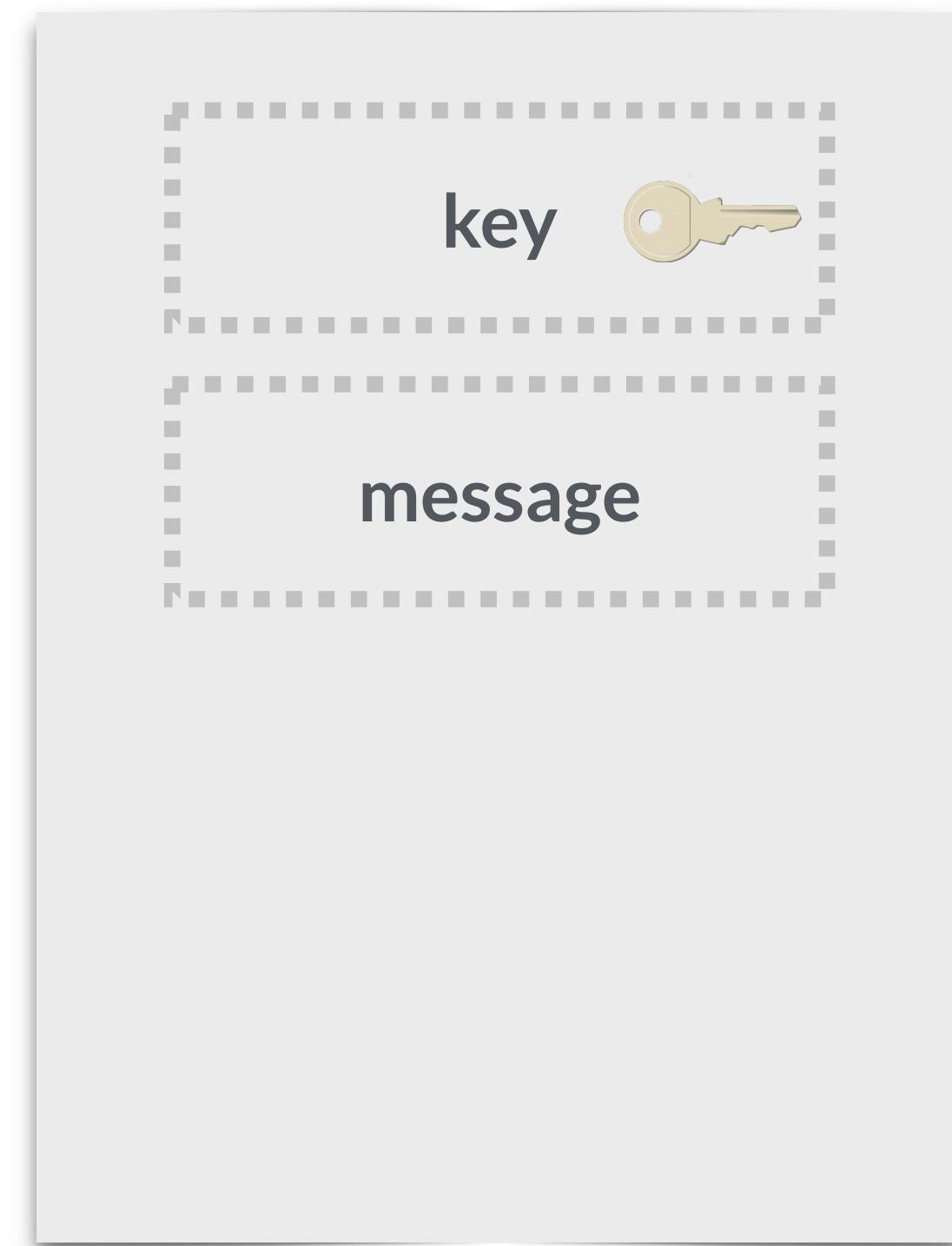
A Kafka Message

- Similar to a *row* or a *record*
- Message is an array of bytes
- No special serialization, that is done at the producer or consumer



A Kafka Message Key

- Message may contain a *key* for better distribution to partitions
- The *key* is also an array of bytes
- If a *key* is provided, a partitioner will hash the key and map it to a single partition
- Therefore it is the only time that something is guaranteed to be in order



Kafka Producers

kafka broker: 0

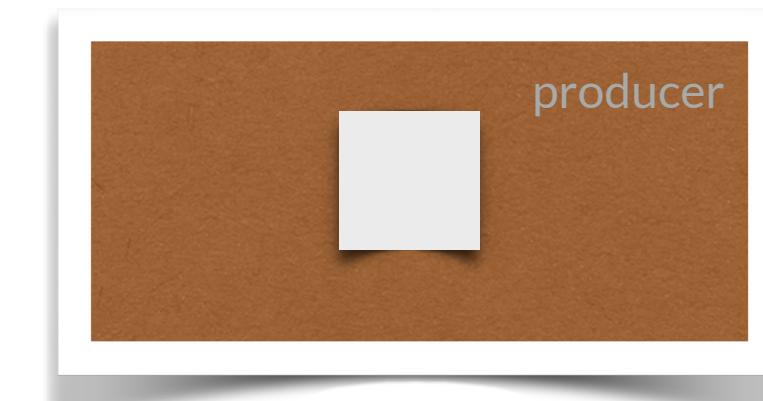
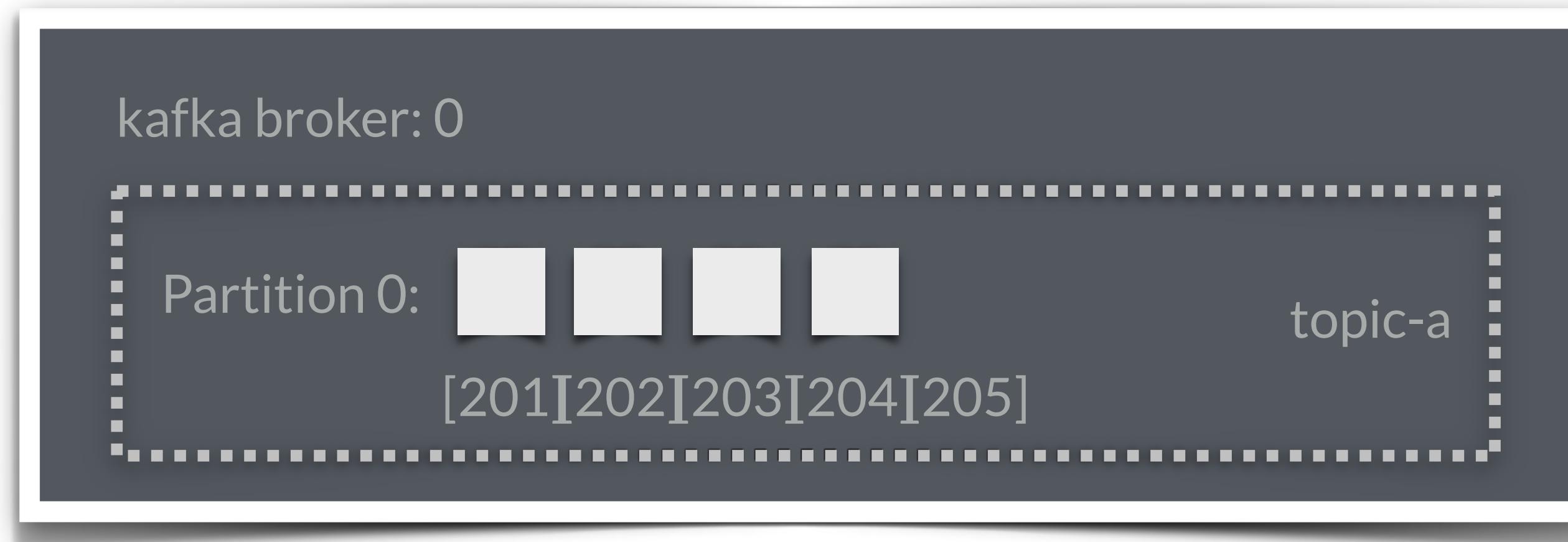
Partition 0:



[0] [1] [2] [3] [4]

topic-a

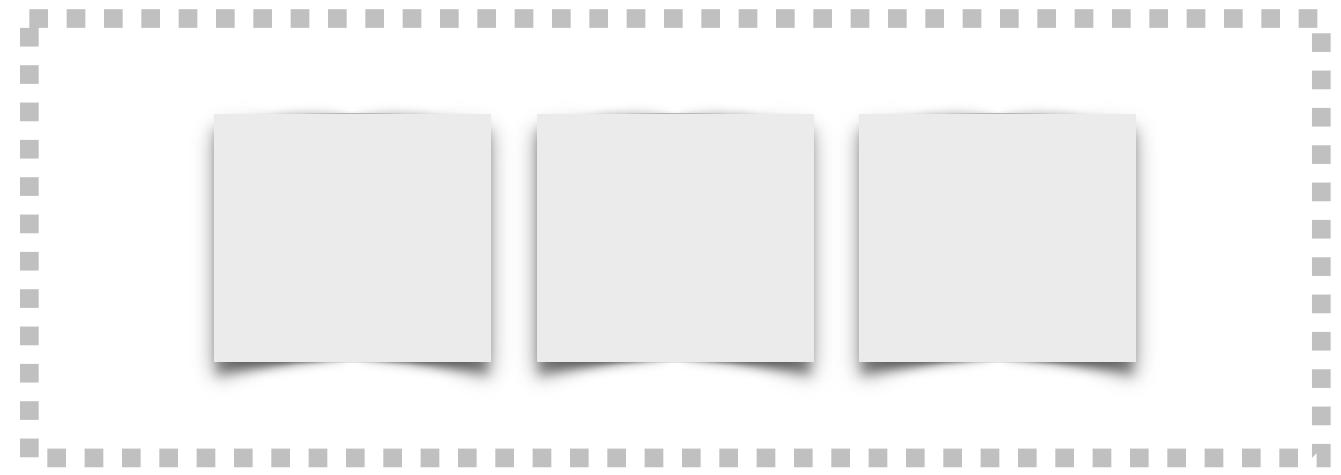
producer

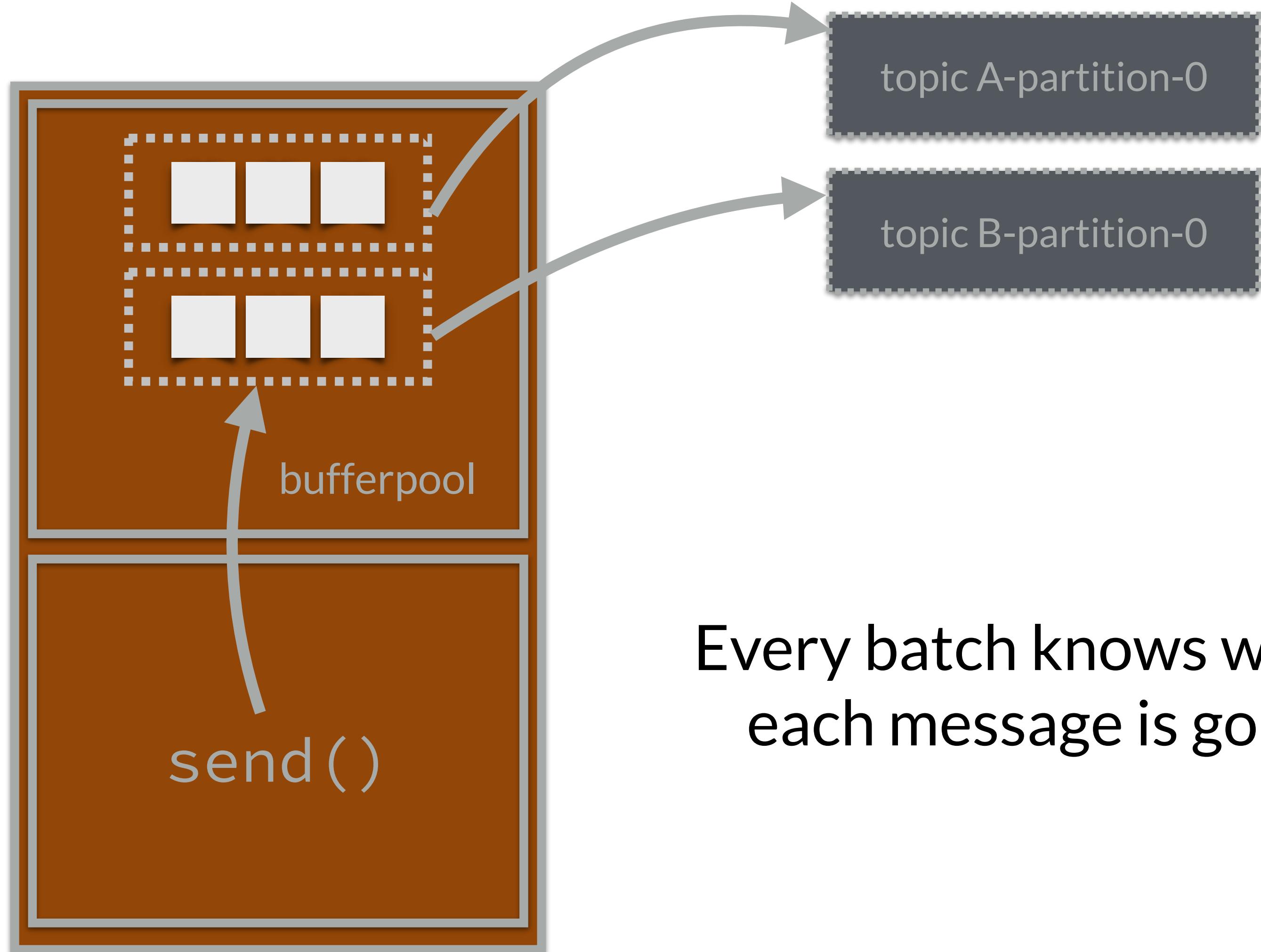


Retention: The data is temporary

Kafka Batch

- A collection of messages, that is sent, configured in *bytes*
- Sent to the same *topic* and the same *partition*
- *Avoids overhead of sending multiple message over the wire*





Every batch knows where
each message is going

$\text{murmur2(bytes)} \% \text{ number partitions}$

This could be changed with custom partitioner

```
murmur2(bytes) % number partitions  
if (key == "CSCO") {  
    else return 0;  
} else {  
    (murmur2(bytes) % number partitions - 1) + 1  
}
```

A-E



F-K



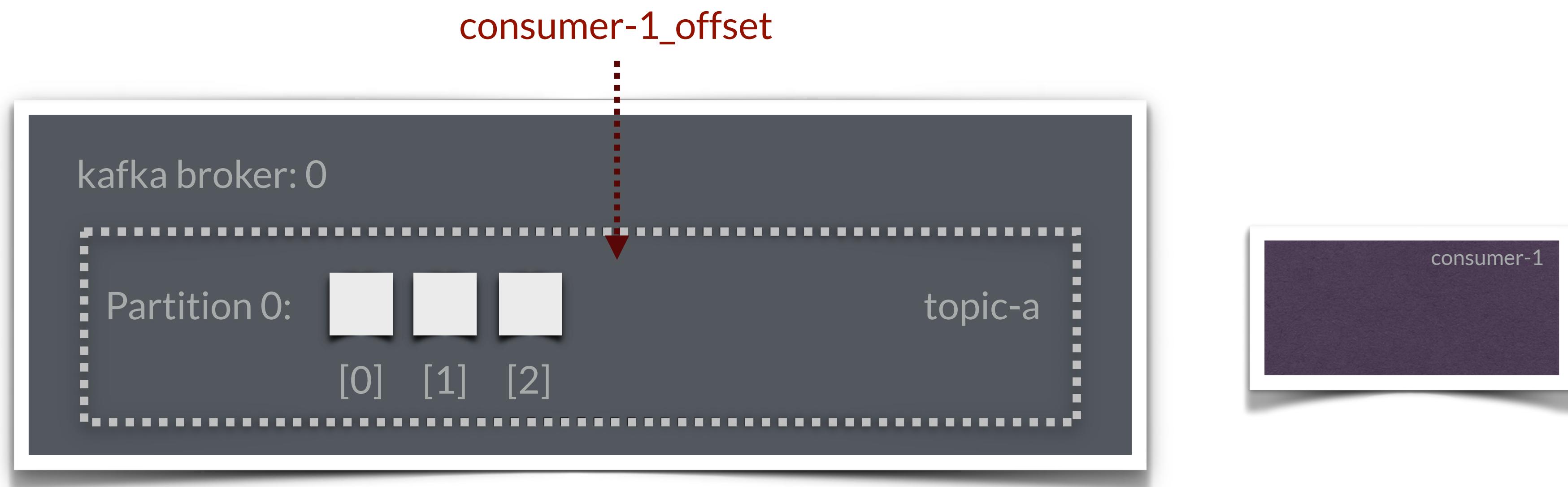
L-S

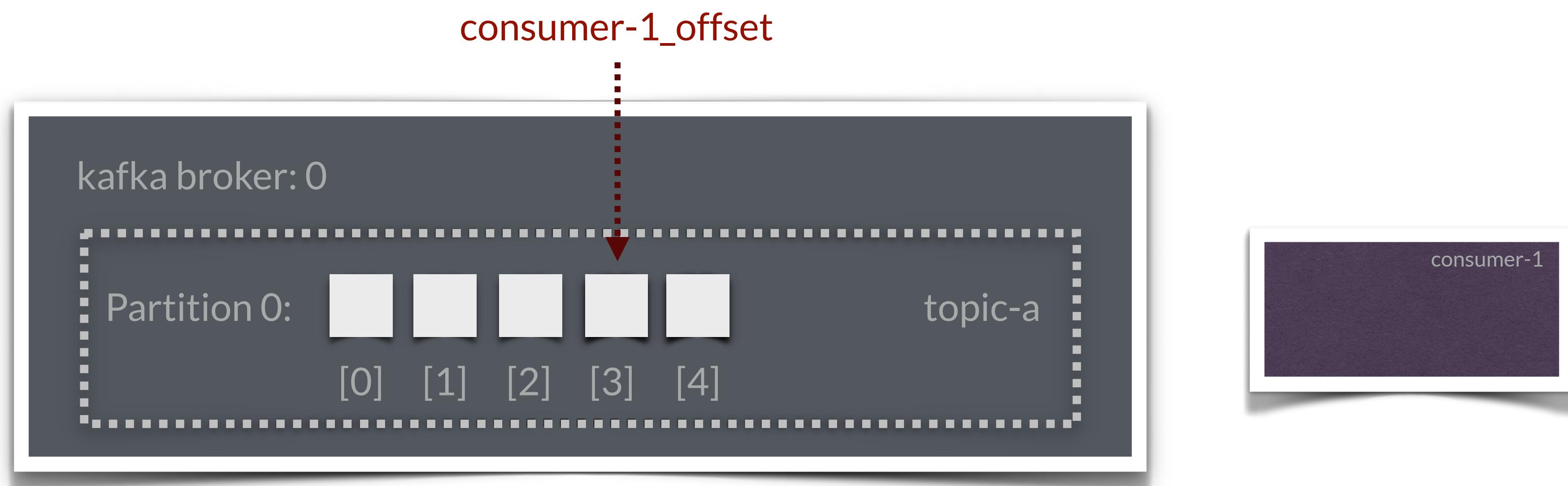


T-Z

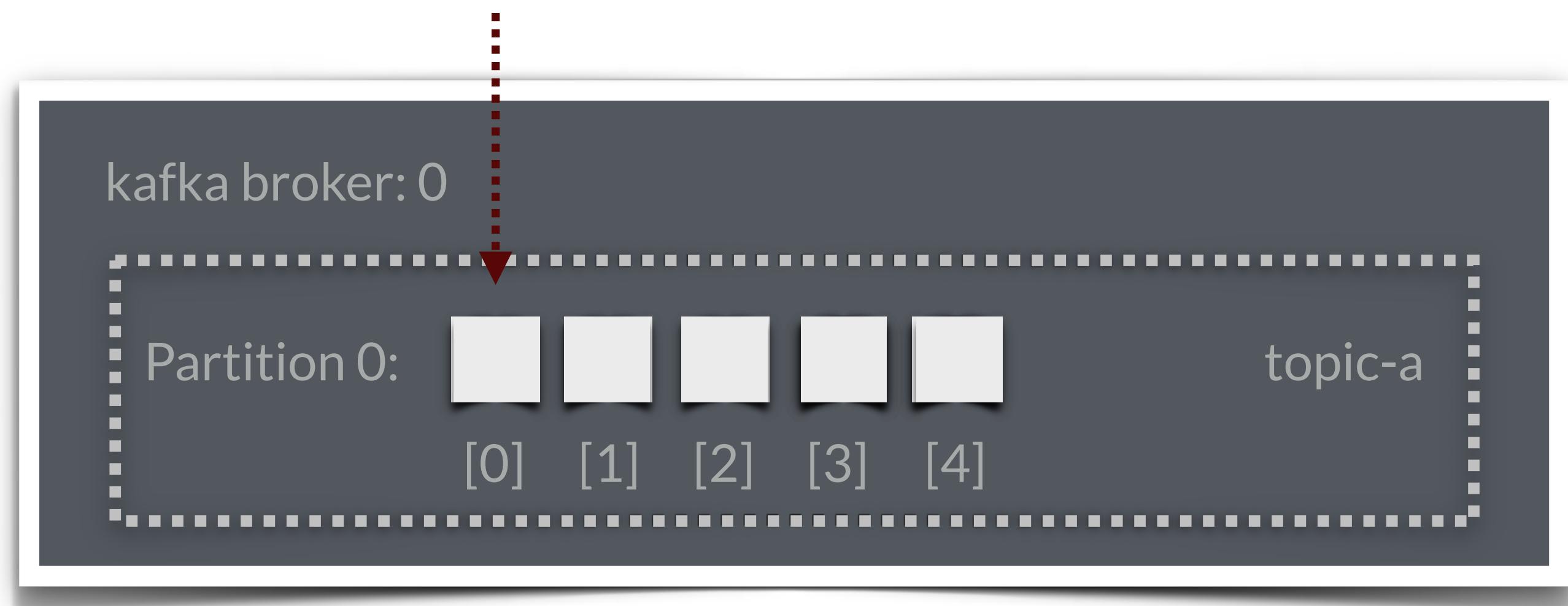


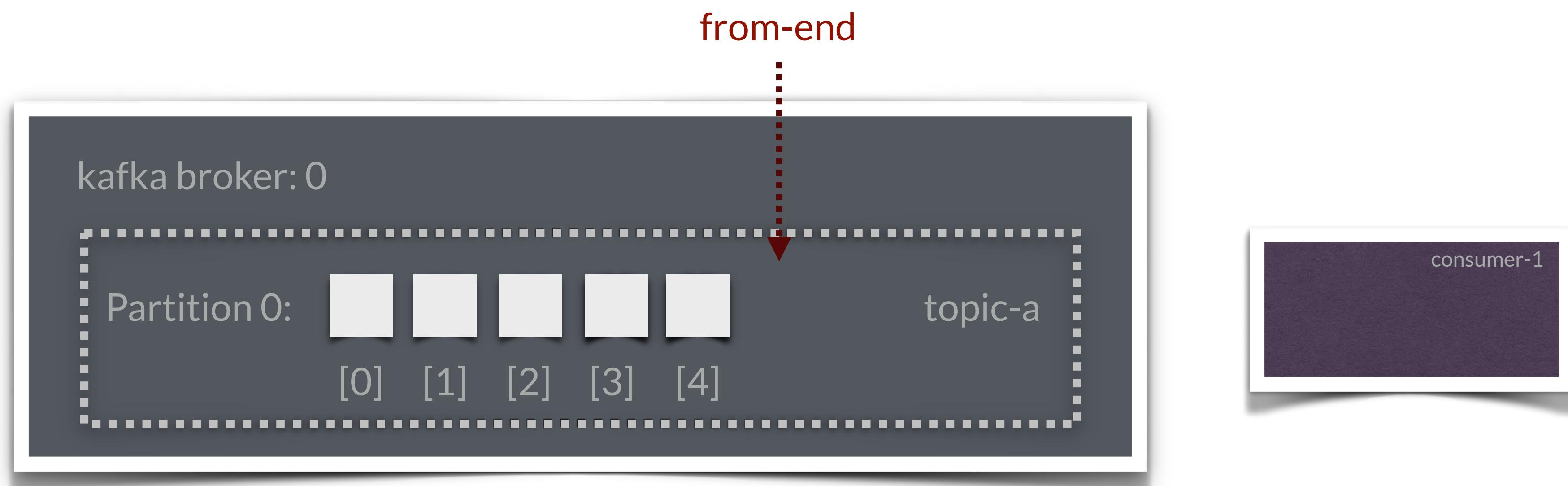
Kafka Consumers

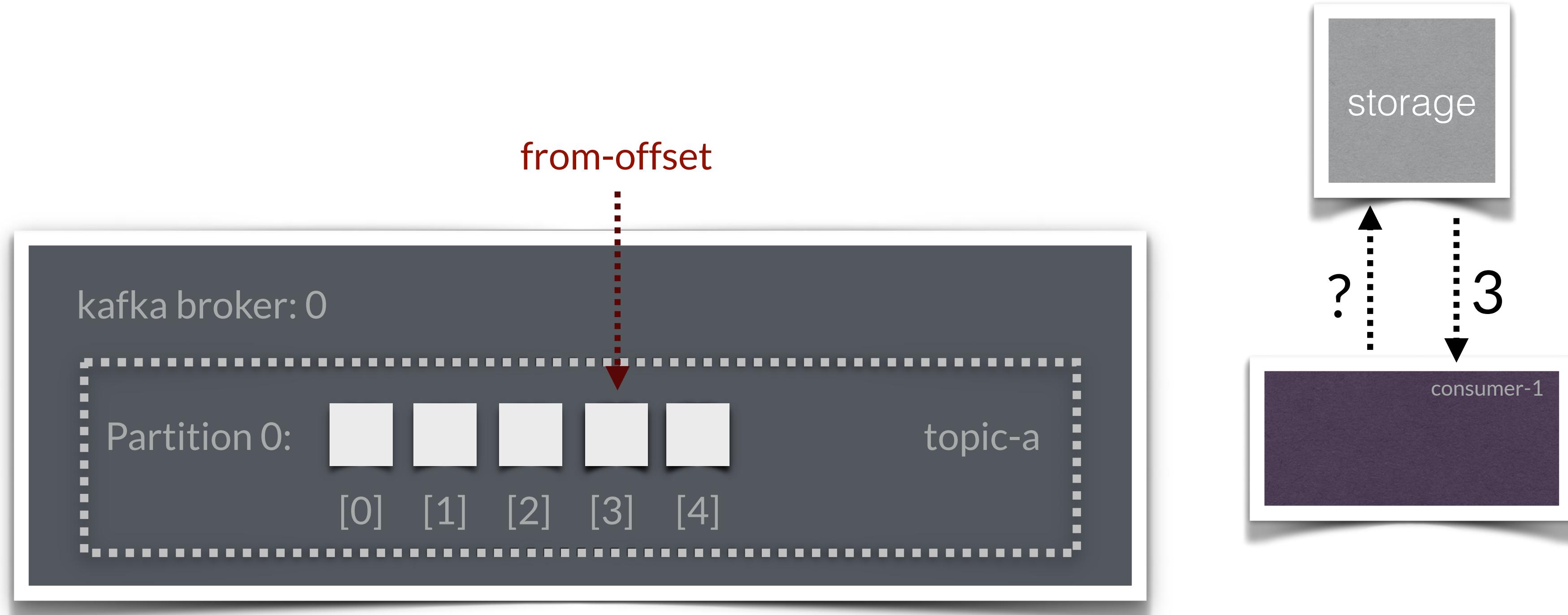




from-beginning



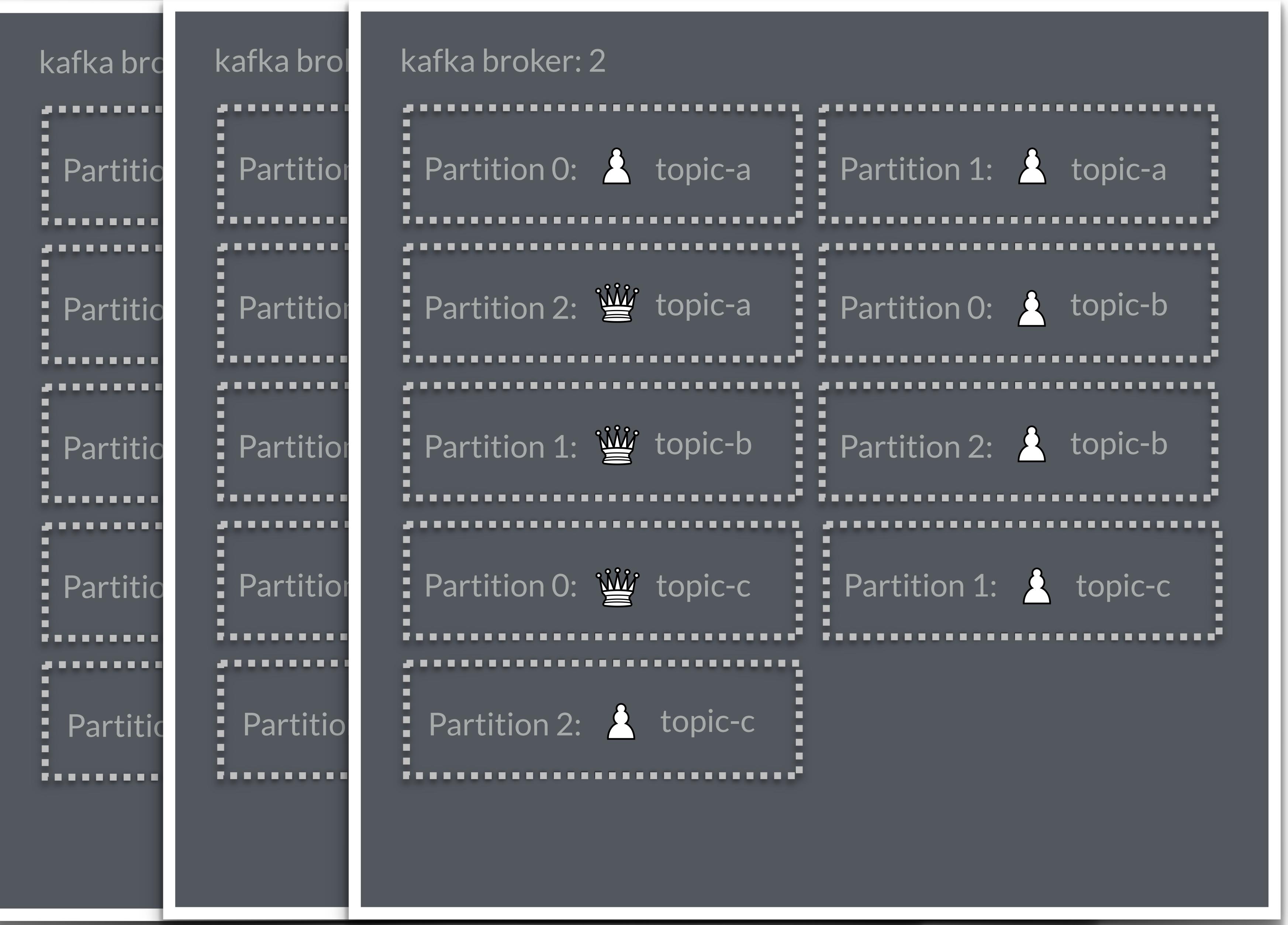




Kafka Partitions



Each partition is on a different broker,
therefore a single topic is scaled



A-E



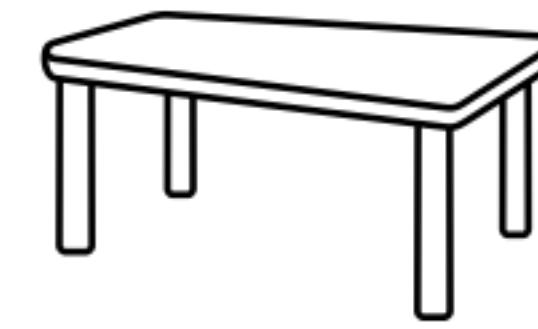
F-K



L-S



T-Z



Kafka Pub/Sub Comparisons

Comparison Pub/Sub

- Kafka as the distributed "git log"
- Ability to replayed in a consistent manner
- Kafka is also stored durable, in order, and deterministic, if key is available
- Data can also be distributed for resiliency
- Scalable and Elastic

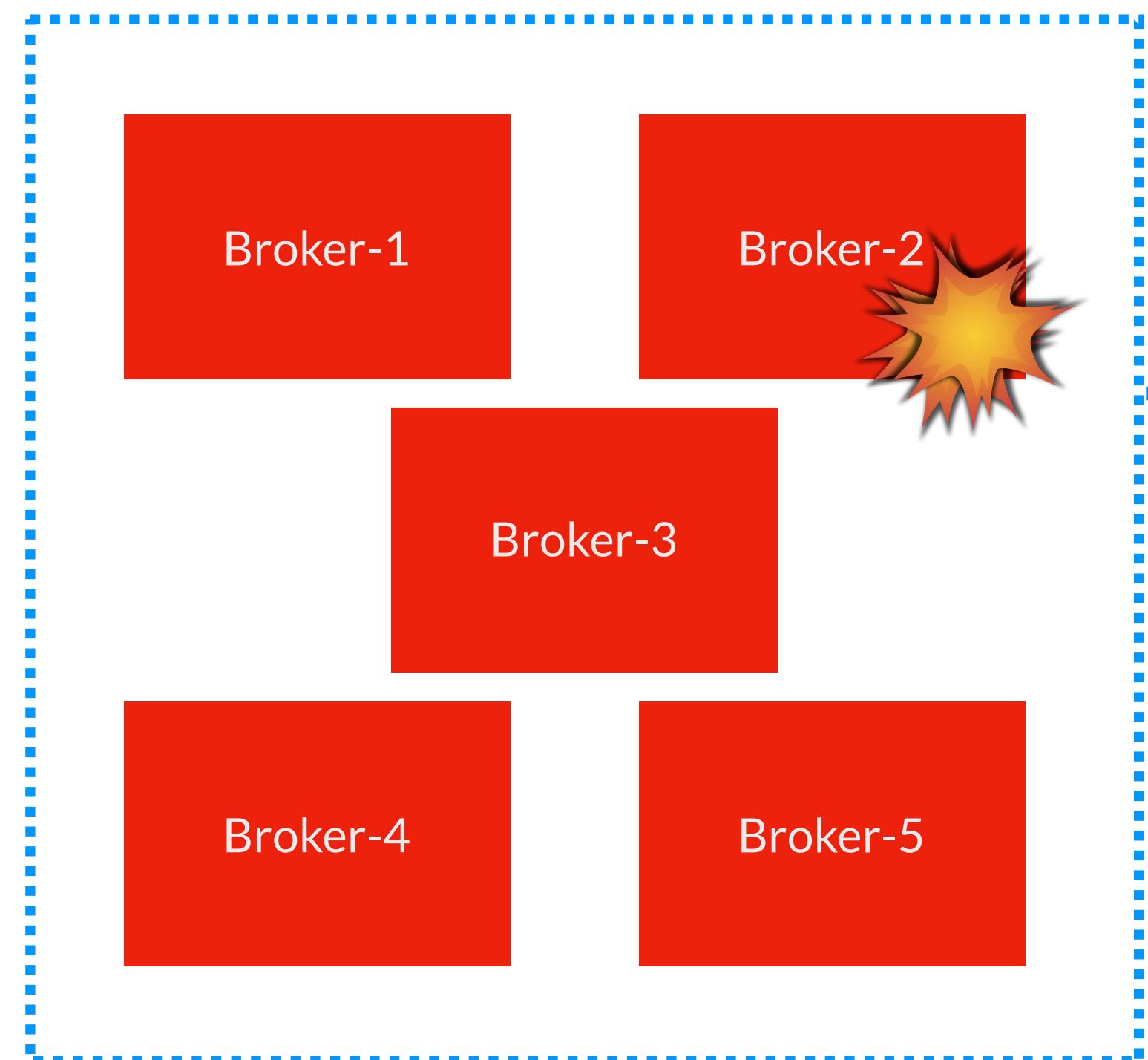
Zookeeper

Zookeeper Essence

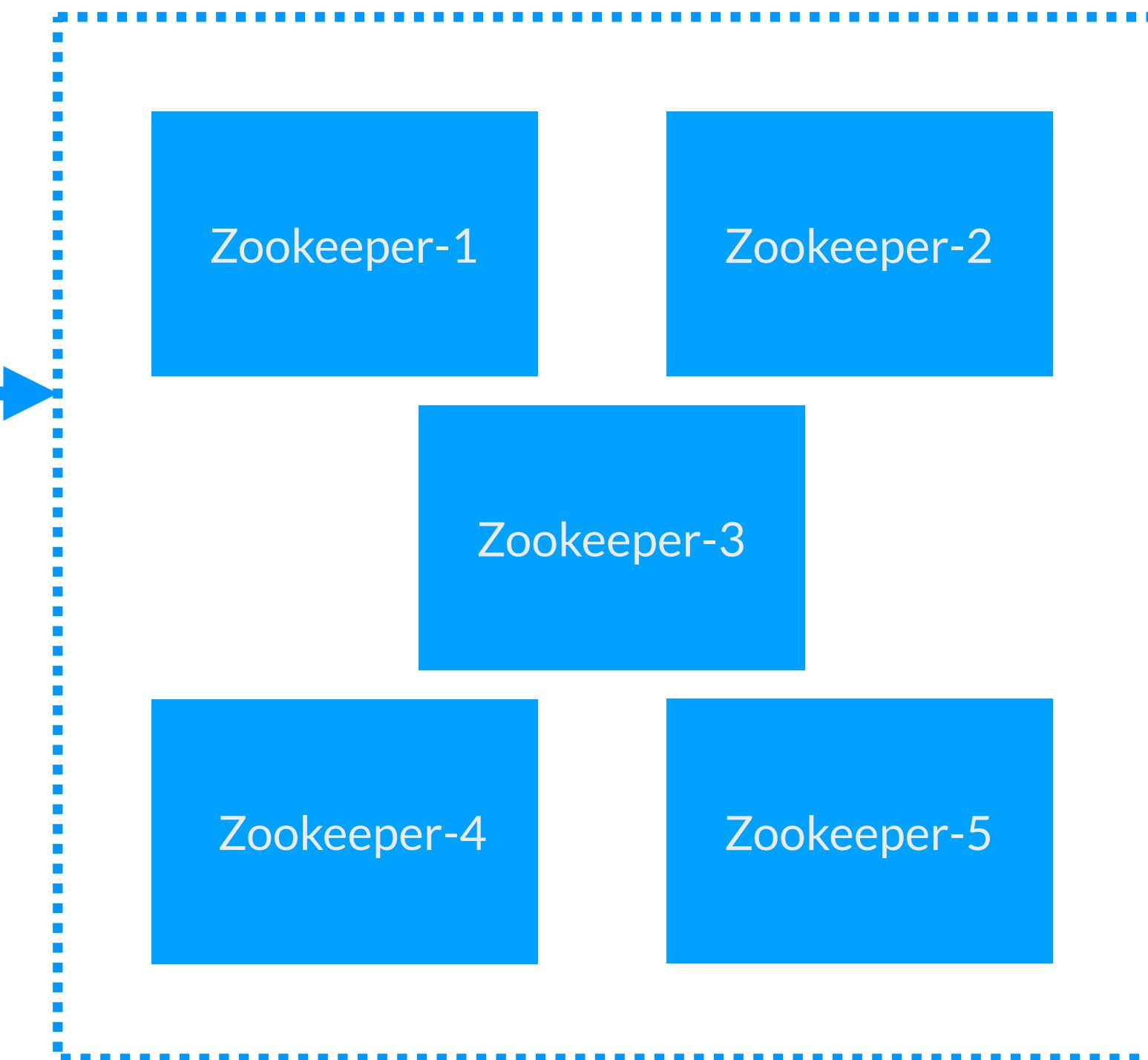
- Centralized Coordination Service
- Maintains:
 - Metadata
 - Naming
 - Configuration



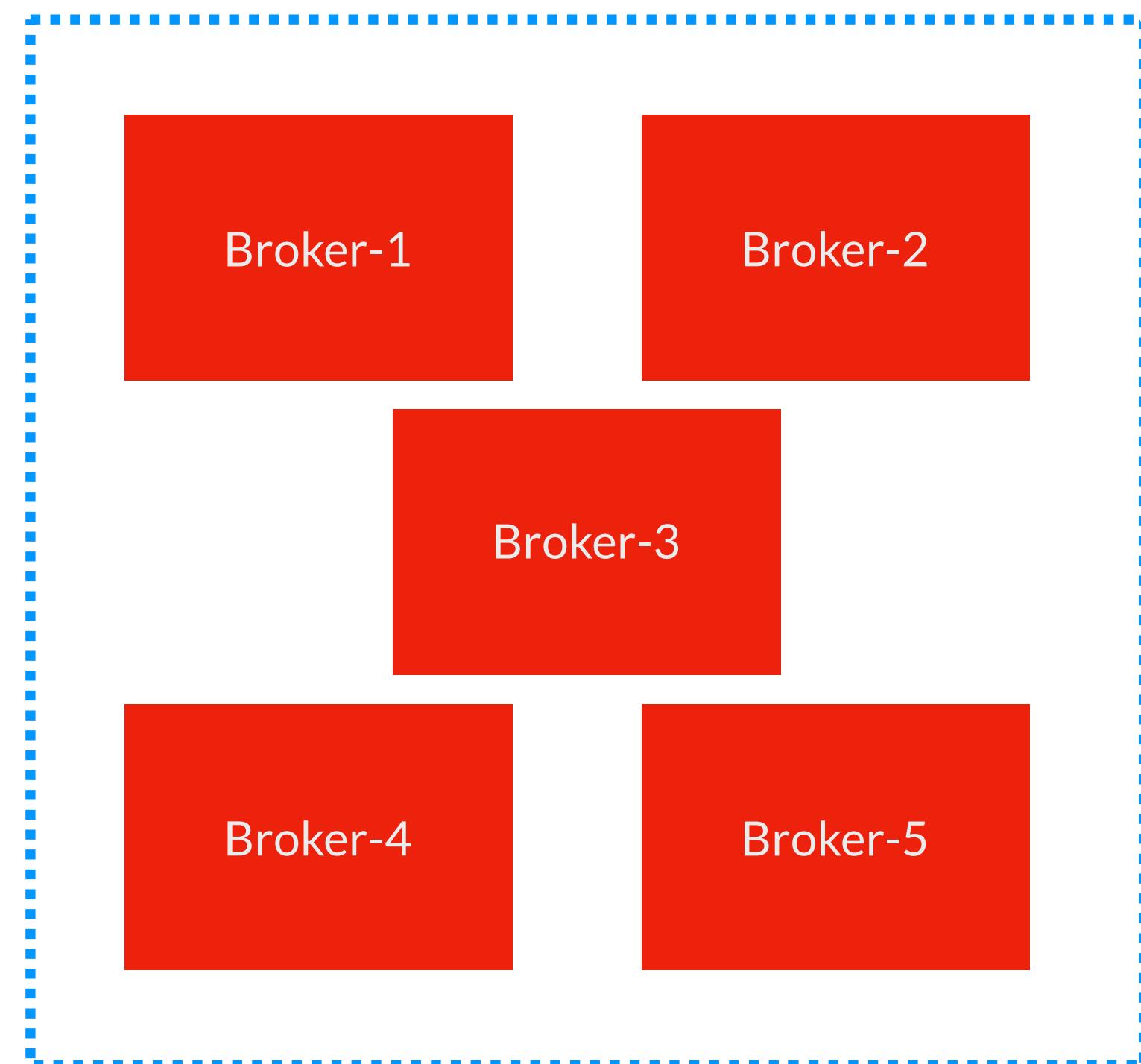
Kafka Cluster



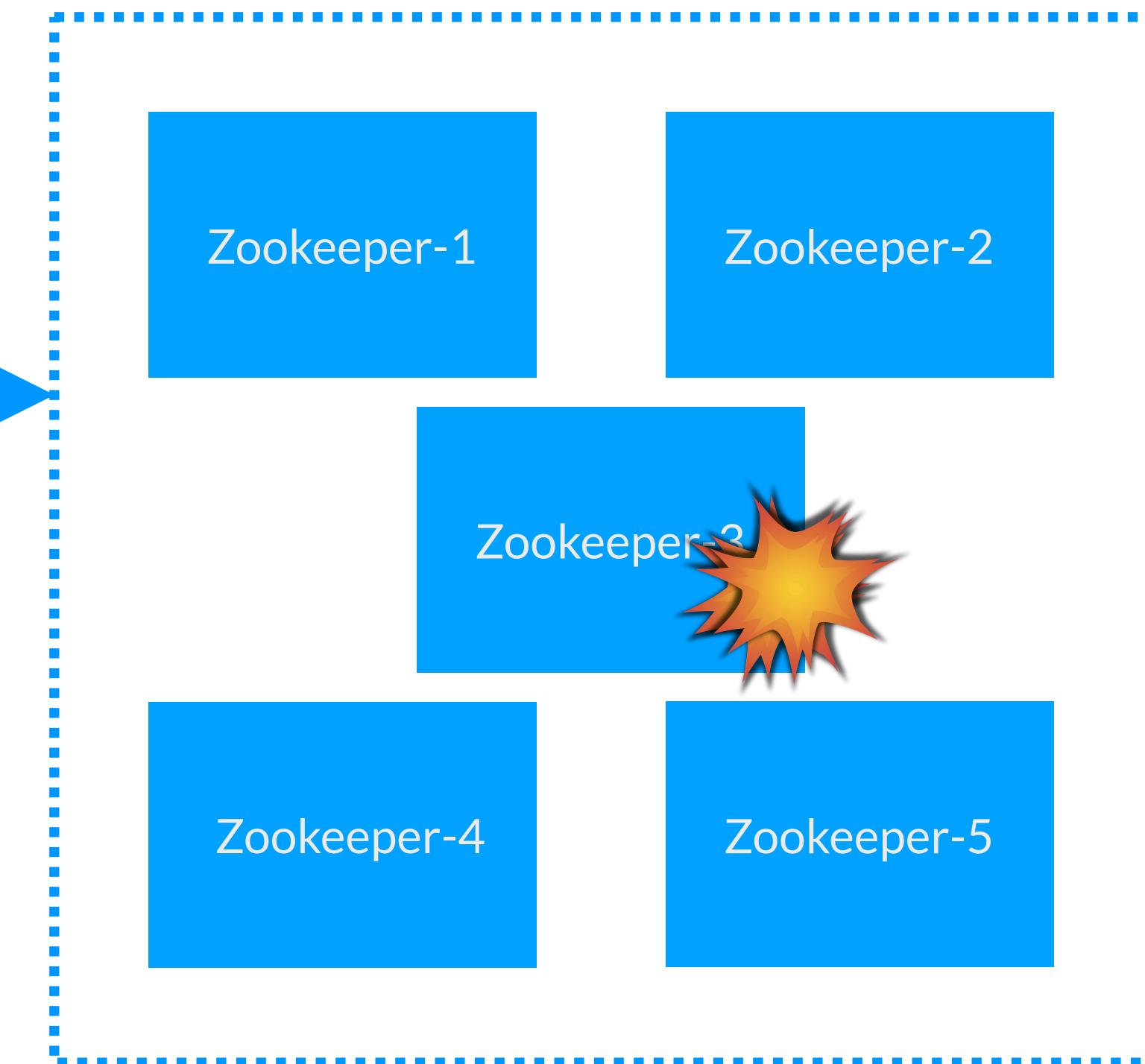
Zookeeper Ensemble



Kafka Cluster



Zookeeper Ensemble



Requirements



Hardware Requirements

- Do not co-locate other applications due to memory page cache pollution and will degrade performance
- Performant drive is required
- Storage capacity will need to be calculated by the expected messages per day and retention
- Slower networks can degrade the rate in which messages are produced

Cloud Requirements

- Analyze by data retention
- Analyze performance need by the producers
- If low latency is required, SSD should be considered
- Ephemeral Storage may be required (Elastic Block Storage)

Kafka Guarantees



Kafka Guarantees

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent.
- A consumer instance sees records in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any records committed to the log.



Kubernetes & Operators

Strimzi Operator

Kubernetes Operators

- Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components.
- Operators follow Kubernetes principles, notably the control loop.
- Extend the cluster's behaviour without modifying the code of Kubernetes itself by linking controllers to one or more custom resources.
- Operators are clients of the Kubernetes API that act as controllers for a Custom Resource

Strimzi Operator

- <https://strimzi.io/>
- Strimzi provides a way to run an Apache Kafka cluster on Kubernetes in various deployment configurations.
- Once deployed it makes use of Custom Resource Definitions to deploy Kafka, and various other artifacts like:
 - Kafka Connect and Connectors
 - Kafka Brokers
 - More..

Lab: Running a Strimzi Operator

Kafka CLI

Creating a Topic

Creating the topic orders with replication factor 2 and 4 partitions

```
$ /usr/bin/kafka-topics --create --zookeeper zoo:2181 \  
--replication-factor 2 --partitions 4 --topic orders
```

Listing Topics

List the topics using one of the zookeeper nodes

```
$ /usr/bin/kafka-topics --zookeeper <zookeeper>:2181 --list
```

Sending a Message

Send a message to list of brokers for a particular topic

```
$ /usr/local/kafka/bin/kafka-console-producer.sh \
--broker-list <kafka-broker>:9092 --topic <topic>
> Hello
> I
> am
> sending
> six
> messages
```

Receiving a Message

Receiving the messages from the topic that was posted by the CLI producer

```
$ /usr/local/kafka/bin/kafka-console-consumer.sh \
--bootstrap-server <kafka-broker>:9092 \
--topic <topic> \
--from-beginning

> Hello
> I
> am
> sending
> six
> messages
```

Showing Distributed Partitions

Showing how partitions are distributed

```
$ /usr/local/kafka/bin/kafka-topics.sh --describe \  
--topic <topic-name> \  
--zookeeper <zookeeper>:2181
```

Lab: Command-Line

Kafka Programming Producers

Establishing Properties

- Construct a `java.util.Properties` object
- Provide two or more locations where the bootstrap servers are located
- Provide a Serializer for the key
- Provide a Serializer for the value

```
Properties properties = new Properties();
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
    "localhost:9092");
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class);
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    IntegerSerializer.class);
```

Create a Producer Object

- Construct a `org.apache.kafka.clients.producer.KafkaProducer` object
- A Kafka Producer is thread-safe
- Inject the Properties into the Producer object

```
KafkaProducer<String, Integer> producer = new KafkaProducer<>(properties);
```

Creating a Record/Message

- Create a `org.apache.kafka.clients.producer.ProducerRecord`
- Accepts many parameters but the main ones are:
 - Topic
 - Key (if applicable)
 - Value

```
ProducerRecord<String, Integer> producerRecord =  
    new ProducerRecord<>("my_orders", state, amount);
```

Sending a message

- Send the record by calling send on the Producer
- Returns a Future object to process the results on another Thread.
- The Future will contain RecordMetadata, an object that has information about your send.

```
Future<RecordMetadata> send = producer.send(producerRecord);
```

RecordMetadata

Contains information about your send including the messages

```
if (metadata.hasOffset()) {  
    System.out.format("offset: %d\n",  
        metadata.offset());  
}  
System.out.format("partition: %d\n",  
    metadata.partition());  
System.out.format("timestamp: %d\n",  
    metadata.timestamp());  
System.out.format("topic: %s\n", metadata.topic());  
System.out.format("toString: %s\n",  
    metadata.toString());
```

Sending with a Callback

- Alternately, you can send with a Callback (lambda)
- Callback is an interface, can be used as a lambda
- If RecordMetadata is null there was an error, if Exception is null the send was successful

```
producer.send(producerRecord, new Callback() {  
    @Override  
    public void onCompletion(RecordMetadata metadata,  
                            Exception e) {  
        ...  
    }  
}
```

Using a Closure to capture Key and Value

- RecordMetadata does not have information on key and value
- Using a closure you can obtain that in the block of your lambda

```
producer.send(producerRecord, (metadata, e) -> {
    if (metadata != null) {
        System.out.println(producerRecord.key());
        System.out.println(producerRecord.value());
    }
})
```

Be a good citizen, close your resources

- When you need to terminate, flush messages from the bufferpool
- Close the Producer

```
producer.flush();
producer.close();
```

Closing Resources in a Shutdown Hook

- `Runtime.getRuntime().addShutdownHook(...)` will listen for SIGTERM (CTRL+C)
- This would make an excellent place to flush and close, and close any loops that you may have created

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    done.set(true);
    producer.flush();
    producer.close();
}));
```

Lab: Running a Producer

Acknowledgements & Retries

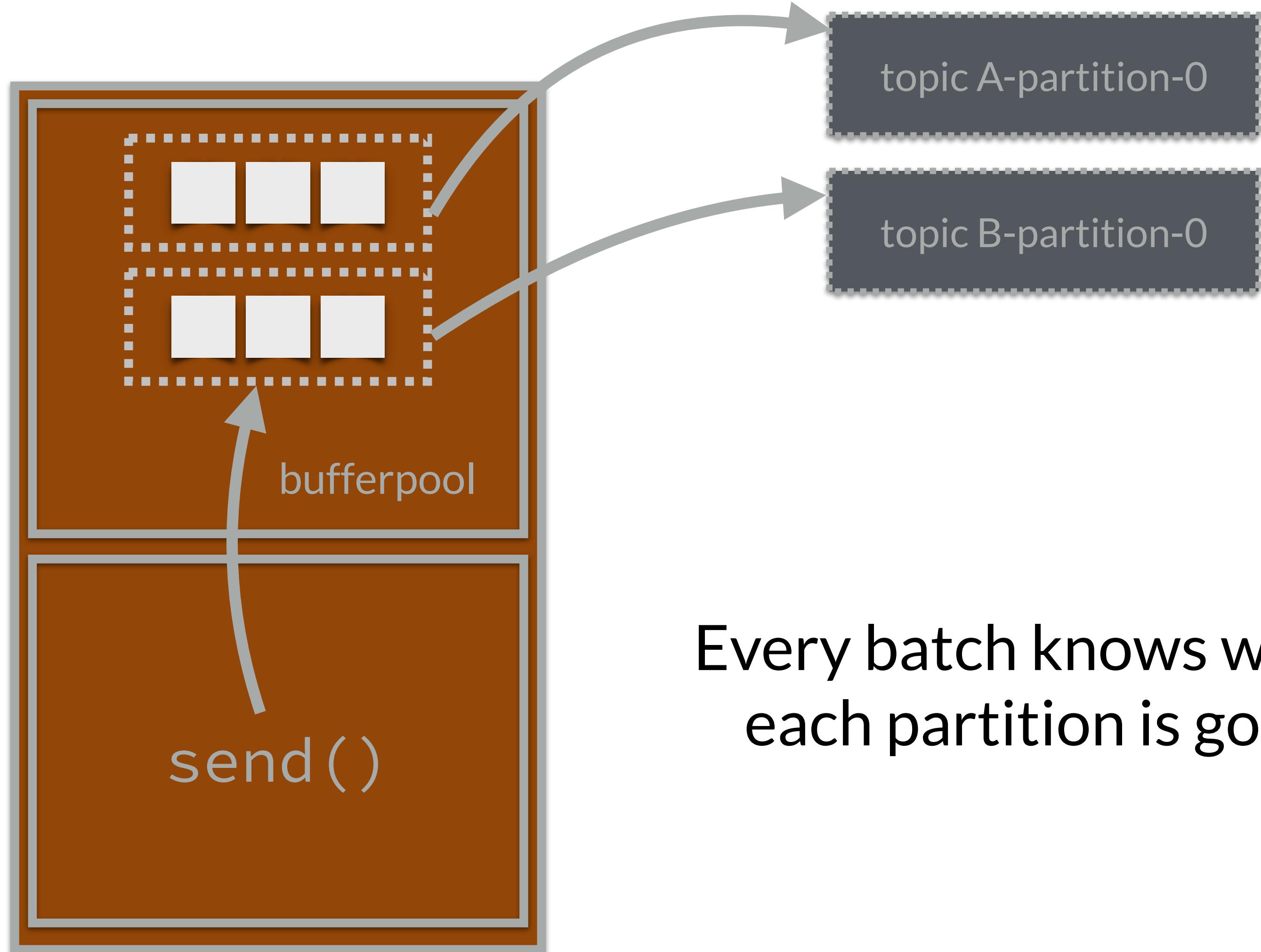
Acks

`acks` controls how many partition replicas must receive the record before the write is considered a success.

```
Properties properties = new Properties();
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
    "localhost:9092");
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class);
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    IntegerSerializer.class);
properties.put(ProducerConfig.ACKS_CONFIG, "all");
properties.put(ProducerConfig.RETRIES_CONFIG, 20);
properties.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 3000);
```

Acks

acks	description
0	No acknowledgment, assume all is well
1	At least one replica will Producer will receive a success response, error if unsuccessful, up to client to handle
all	All replicas must acknowledge. Higher latency, safest



Every batch knows where
each partition is going

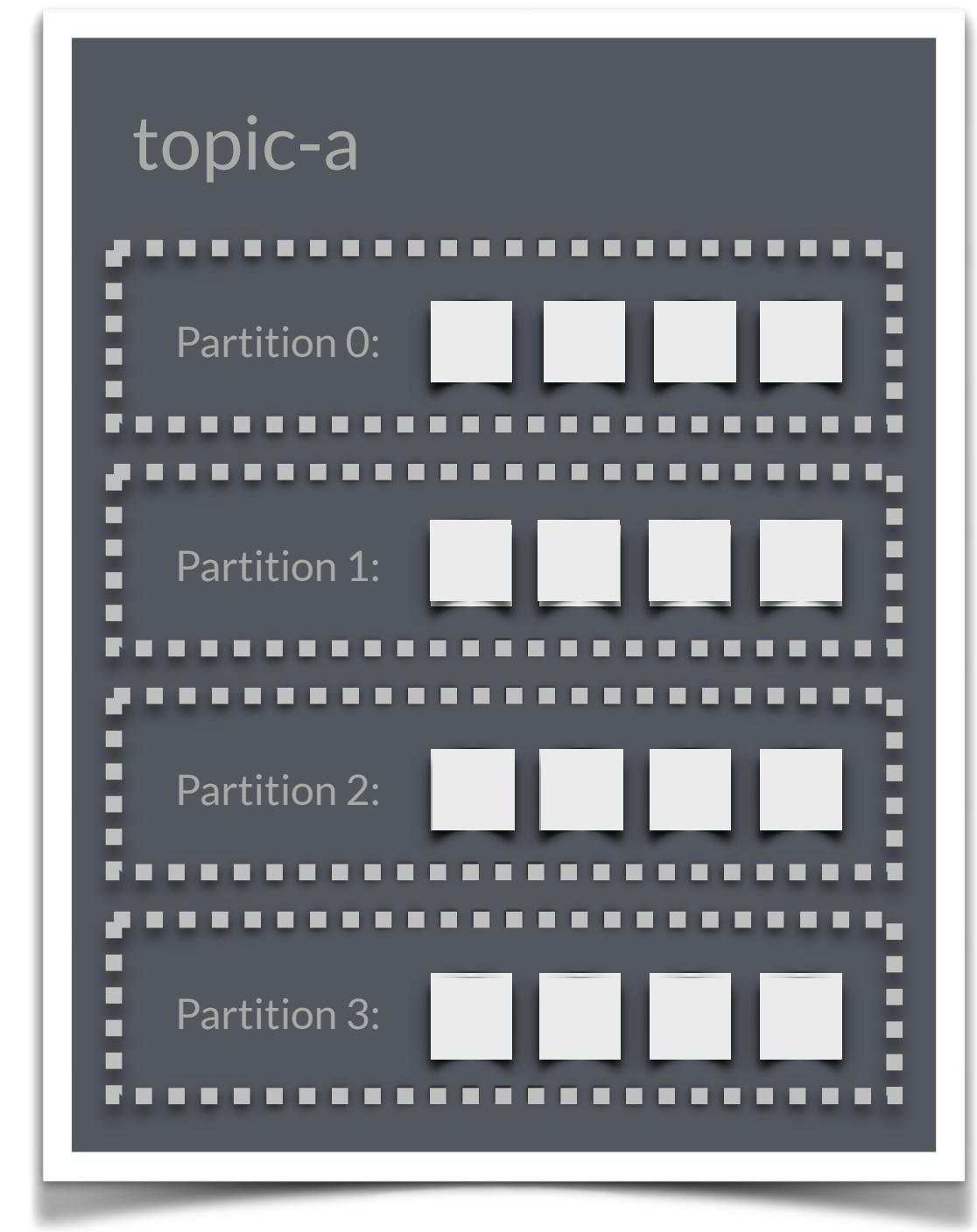
$\text{murmur2(bytes)} \% \text{ number partitions}$

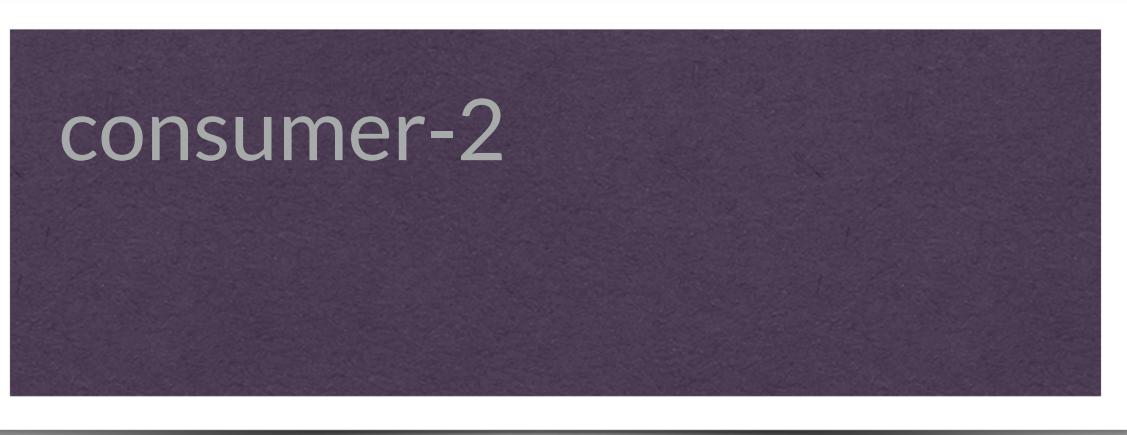
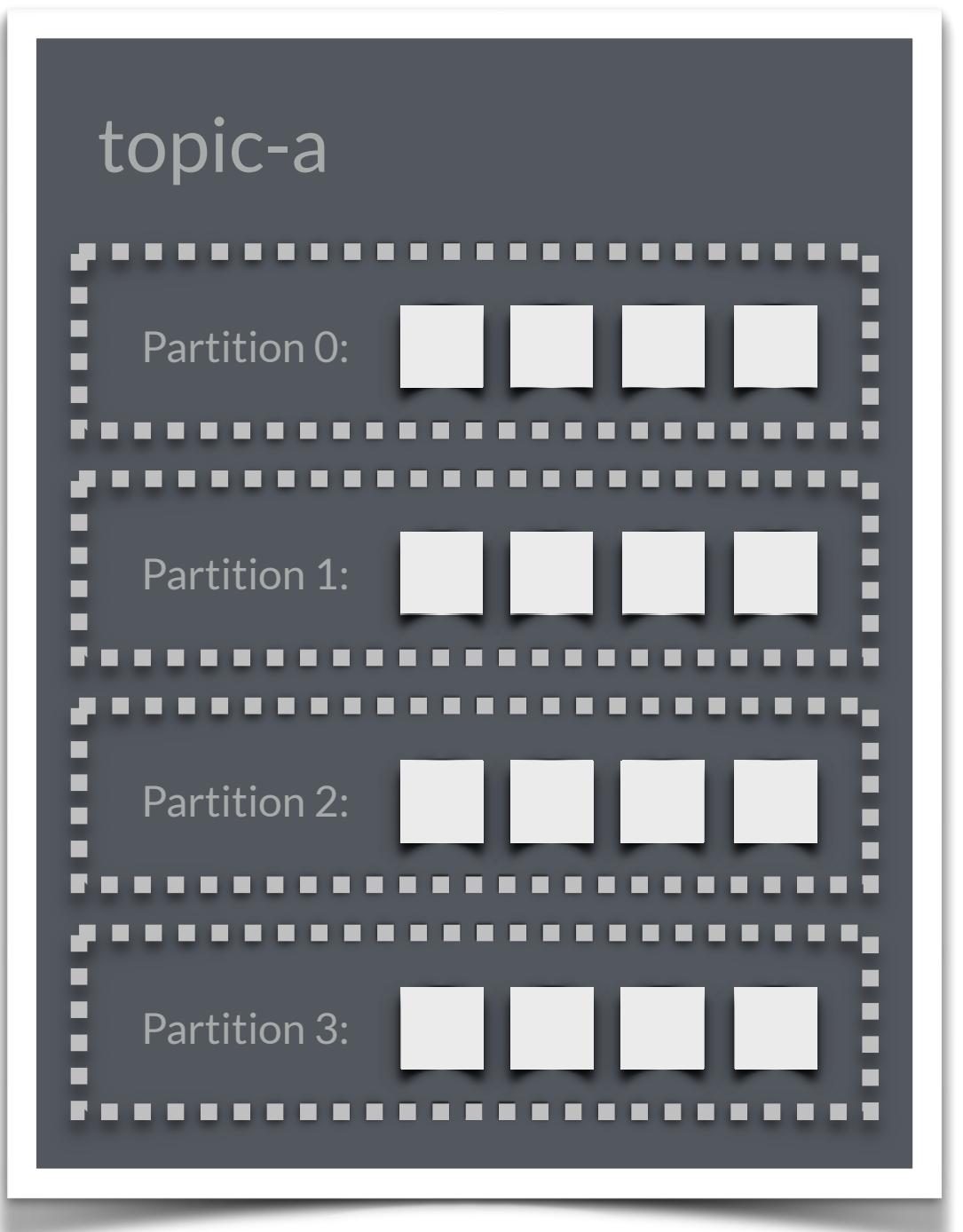
Compression & Batching

Kafka Programming Consumers & Groups

Kafka Consumer Groups

- Consumers are typically done as a group
- A single consumer will end up inefficient with large amounts of data
- A consumer may never catch up
- **Every consumer should be on it's own machine, instance, pod**





Kafka's Goal

Kafka scales to large amount of different consumers without affecting performance

Kafka Consumer Threading

- There are no multiple consumers that belong to the same group in one thread.
One consumer per one thread.
- There are not multiple threads running one consumer, **One consumer per one thread.**

__consumer_offsets

- Topic on Kafka brokers that contain information about consumer and their offsets, stored in Kafka's data directory

Establishing Properties

- Construct a `java.util.Properties` object
- Provide two or more locations where the bootstrap servers are located
- Provide a "Team Name", officially called a `group.id`
- Provide a DeSerializer for the key
- Provide a DeSerializer for the value

```
Properties properties = new Properties();
properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
    "localhost:9092");
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "my_group");
properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringDeserializer");
properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.IntegerDeserializer");
```

Create a Consumer Object

- Construct a `org.apache.kafka.clients.consumer.KafkaConsumer` object
- A Kafka Consumer is **not thread-safe**
- Inject the Properties into the Consumer object

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Processing Messages

- Use the consumer that you have constructed, and call poll with pulse time
- The poll is a max sleep time, if a message is ready it will download a batch, ConsumerRecords
- Iterate through each record in the batch with a for-loop, process the message

```
while (!done.get()) {  
    ConsumerRecords<String, String> records =  
        consumer.poll(Duration.of(500, ChronoUnit.MILLISECONDS));  
    for (ConsumerRecord<String, String> record : records) {  
        System.out.format("offset: %d\n", record.offset());  
        System.out.format("partition: %d\n", record.partition());  
        System.out.format("timestamp: %d\n", record.timestamp());  
        System.out.format("timeStampType: %s\n", record.timestampType());  
        System.out.format("topic: %s\n", record.topic());  
        System.out.format("key: %s\n", record.key());  
        System.out.format("value: %s\n", record.value());  
    }  
}
```

Be a good citizen, close your resources

- When you need to terminate, let the *Kafka group coordinator* know you are done
- Close the Consumer

```
consumer.close();
```

Consumer Offset Reset

Consumer Offset Reset

- When a consumer is assigned it must determine what offset for the partition to start
- Position is set according to your consumer's reset policy, either earliest or latest
- This is only applicable if there is no valid offset in the `__consumer_offset__` topic

```
Properties properties = new Properties();
...
properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Manual Commits

Manual Commits

- Consumers will commit automatically every 5 seconds
- If your consumer fails within those 5 seconds another consumer may get those messages
- Might be too long before commit, in such case you may want to handle commits yourself
- Turn off your auto-commit by setting `ENABLE_AUTO_COMMIT_CONFIG` to false

```
Properties properties = new Properties();
...
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Synchronous Commit

- After turning off auto-commit
- Committing and blocking synchronously yourself
- Synchronous commits will block

Synchronous Commit

- After turning off auto-commit
- Committing and blocking synchronously yourself
- Synchronous commits will block

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);  
...  
consumer.commitSync(); //Block
```

Asynchronous Commit

- After turning off auto-commit
- Committing and letting it commit asynchronously
- Advantage: Speed, Higher throughput
- Disadvantage: You will get error reports asynchronously

```
consumer.commitAsync((offsets, exception) -> {  
    //offsets  
    //exception  
});
```

Using both kinds of commits

- You can commit asynchronously within the loop
- Then commit synchronously outside the loop
- Reason, if you are out of the loop, likely you want to shut down

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> done.set(true)));

while (!done.get()) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.of(500, ChronoUnit.MILLIS));
    for (ConsumerRecord<String, String> record : records) {
        ...
    }
    consumer.commitAsync((offsets, exception) -> {
        //offsets
        //exception
    });
}
consumer.commitSync(); //Block
consumer.close();
```

Consumer Rebalancing

Partition Rebalance

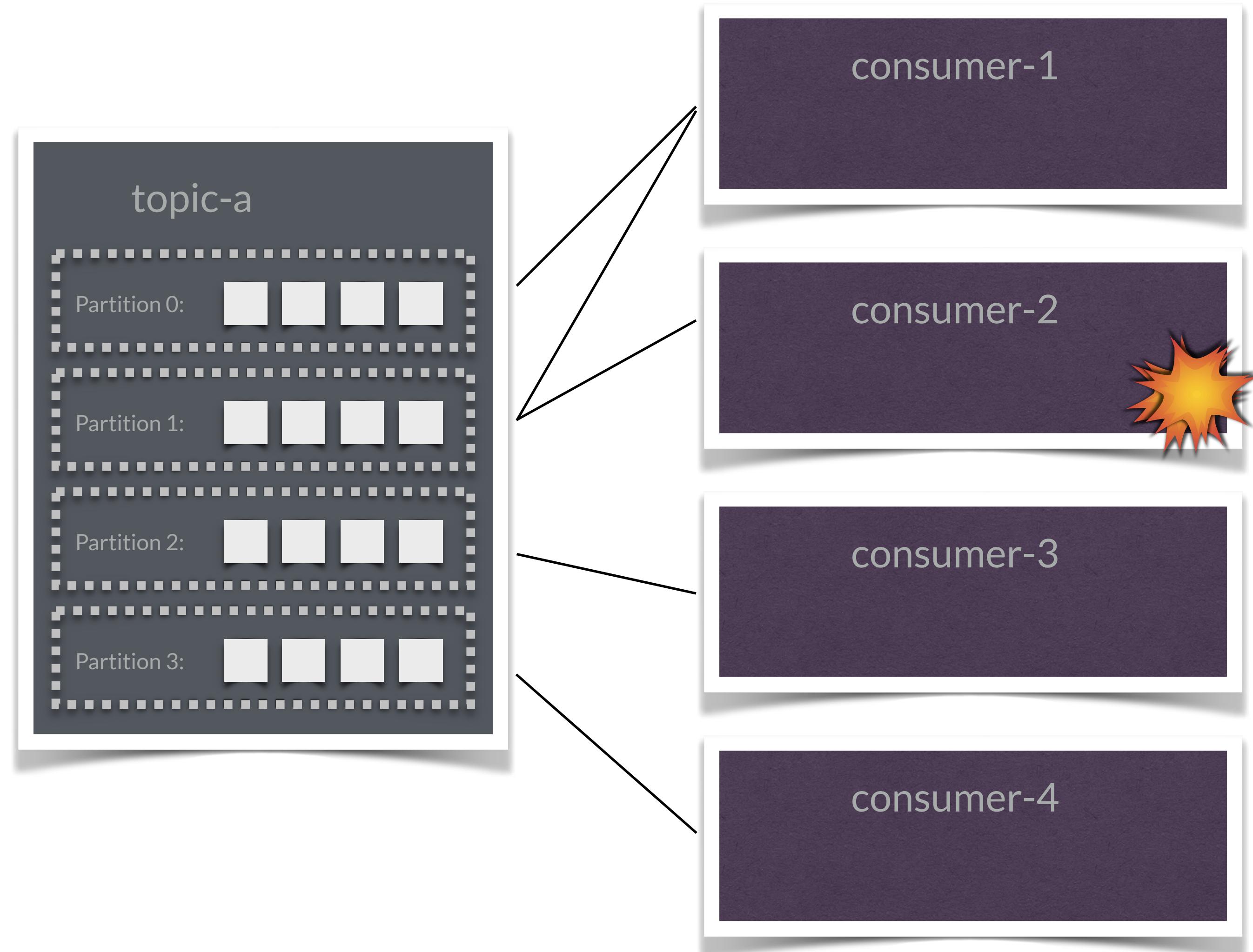
- When one partition is moved from one consumer to another, this is known as a *rebalance*.
- A way to mitigate when either consumers go down, or when consumers are added.
- Although unavoidable, this will cause an unfortunate pause, and it ***will lose state***.

Groups and Heartbeats

- At regular intervals, consumers will send heartbeats to the broker, to let it know it is alive and still reading data
- Heartbeats are sent to a Kafka broker called the *group coordinator*
- They are sent when the consumer polls and consumes a record

Leaving the Group

- When a consumer leaves the group, it lets the *group coordinator* know it is done
- The Kafka broker then triggers a rebalance of the group.



Consumer Rebalance Listener

- You can implement a ConsumerRebalanceListener and be alerted when a rebalance has occurred
- The rebalance listener requires that you implement onPartitionsRevoked, and onPartitionsAssigned

```
consumer.subscribe(Collections.singletonList("my_orders"),
    new ConsumerRebalanceListener() {
        @Override
        public void onPartitionsRevoked(Collection<TopicPartition> collection) {
            System.out.println("Partition revoked:" +
                collectionTopicPartitionToString(collection));
            consumer.commitAsync();
        }

        @Override
        public void onPartitionsAssigned(Collection<TopicPartition> collection) {
            System.out.println("Partition assigned:" +
                collectionTopicPartitionToString(collection));
        }
    });
}
```

Lab: Running a Consumer

Retention Settings

Retention

- Durable Storage over a Period of Time
- Can either be configured in time or size
- Once reached messages are expunged

Retention Formula

Who administers Kafka can update the default retention time or size

```
log.retention.hours = 120  
log.retention.bytes = 3221225472
```

Default Hours are set to (168 hours) or one week

You may also use **log.retention.minutes** or **log.retention.ms**, whichever is smallest wins.

You can also configure retention per topic, like the following

```
bin/kafka-configs --zookeeper localhost:2181 --entity-type topics --entity-name my-topic --alter --add-config retention.ms=128000
```

Retention Evaluation

- Durable Storage over a Period of Time
- Can either be configured in time or size
- Once reached messages are expunged

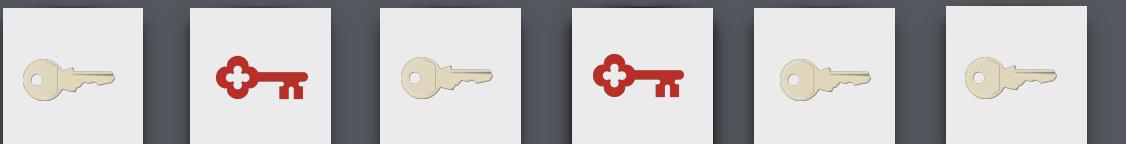
Compaction

Compaction

- A form of retention where messages of the same key where only the latest message will be retained.
- Compaction is performed by a *cleaner thread*

kafka broker: 0

Partition 0:

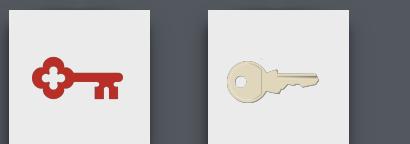


[0] [1] [2] [3] [4] [5]



kafka broker: 0

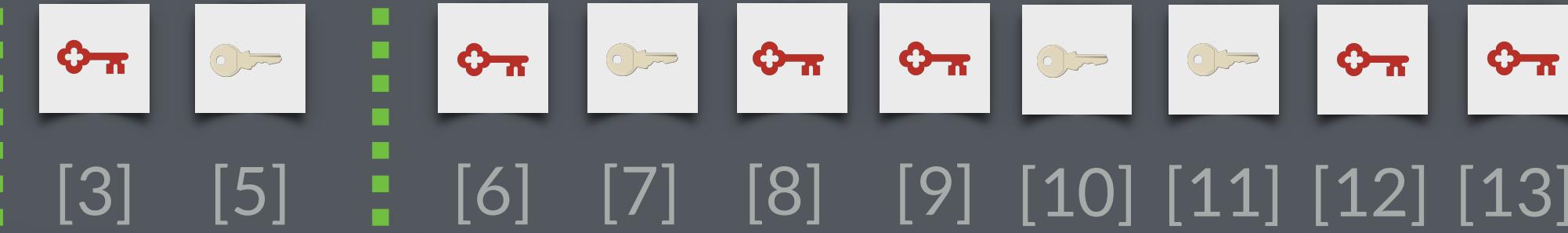
Partition 0:



[3] [5]

kafka broker: 0

Partition 0:



clean

dirty

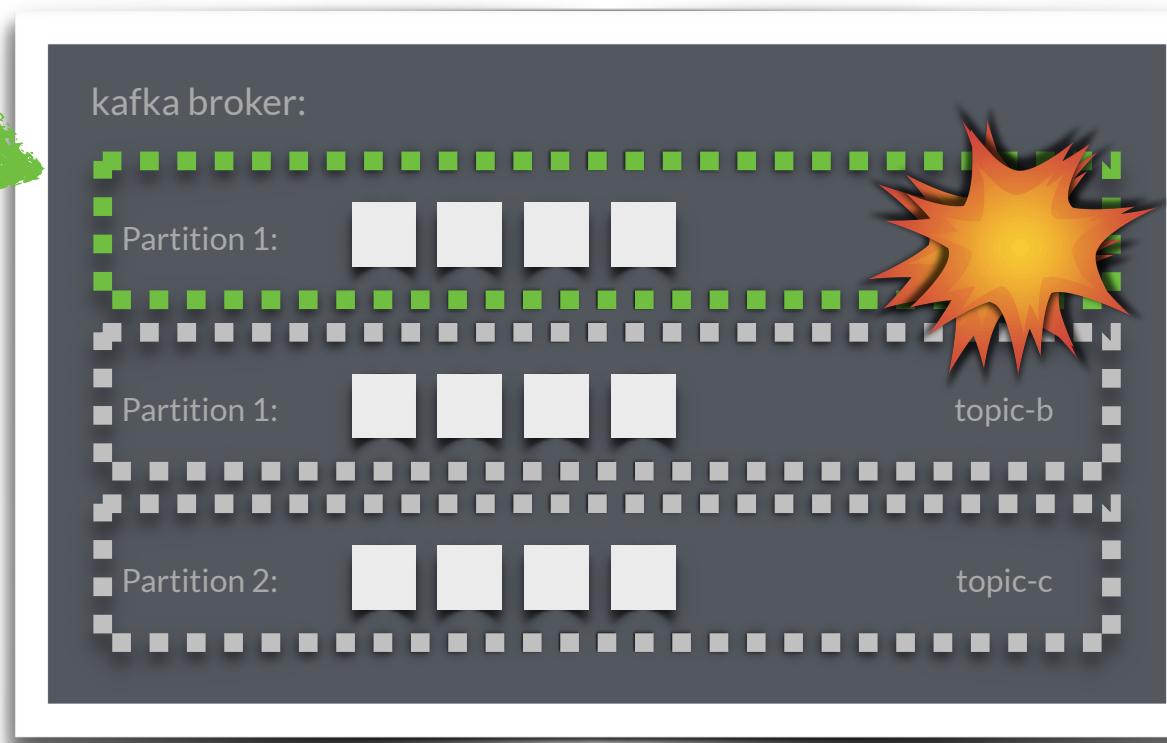
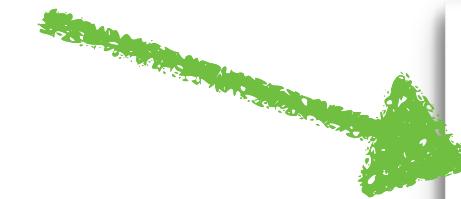
Kafka will start compacting when 50% of the topic contains dirty records

Resiliency

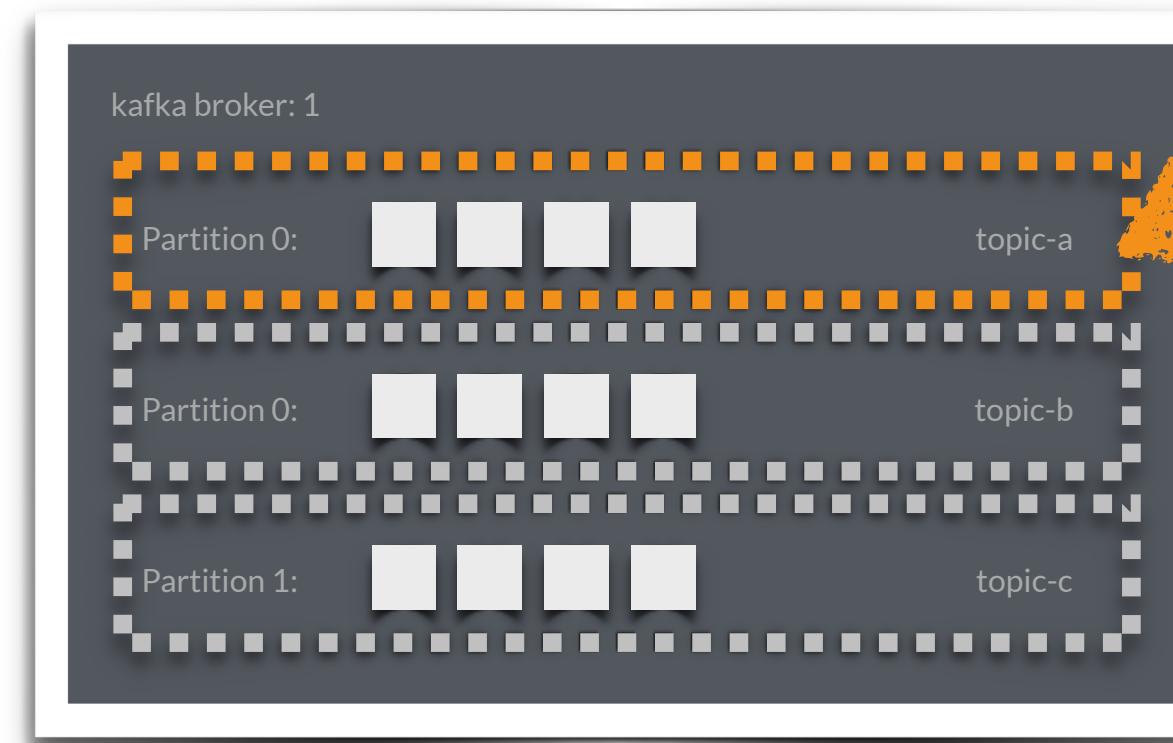
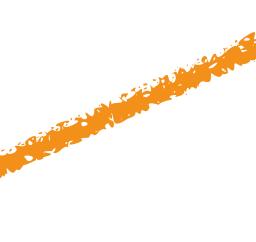
ISR (In sync replicas)

- Given a leader partition, an in sync replica is one that has been kept up to date within the last 10 seconds
- This is configurable
- During a crash, the closes ISR will retain control for failover

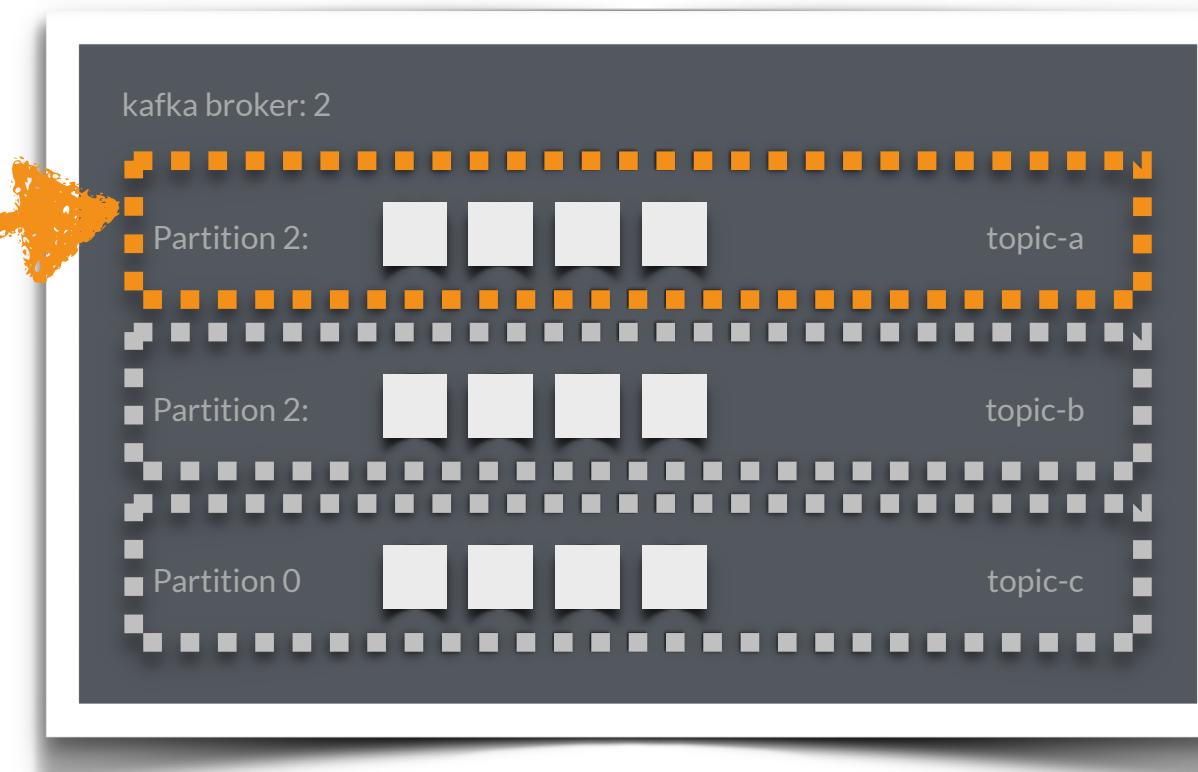
Leader



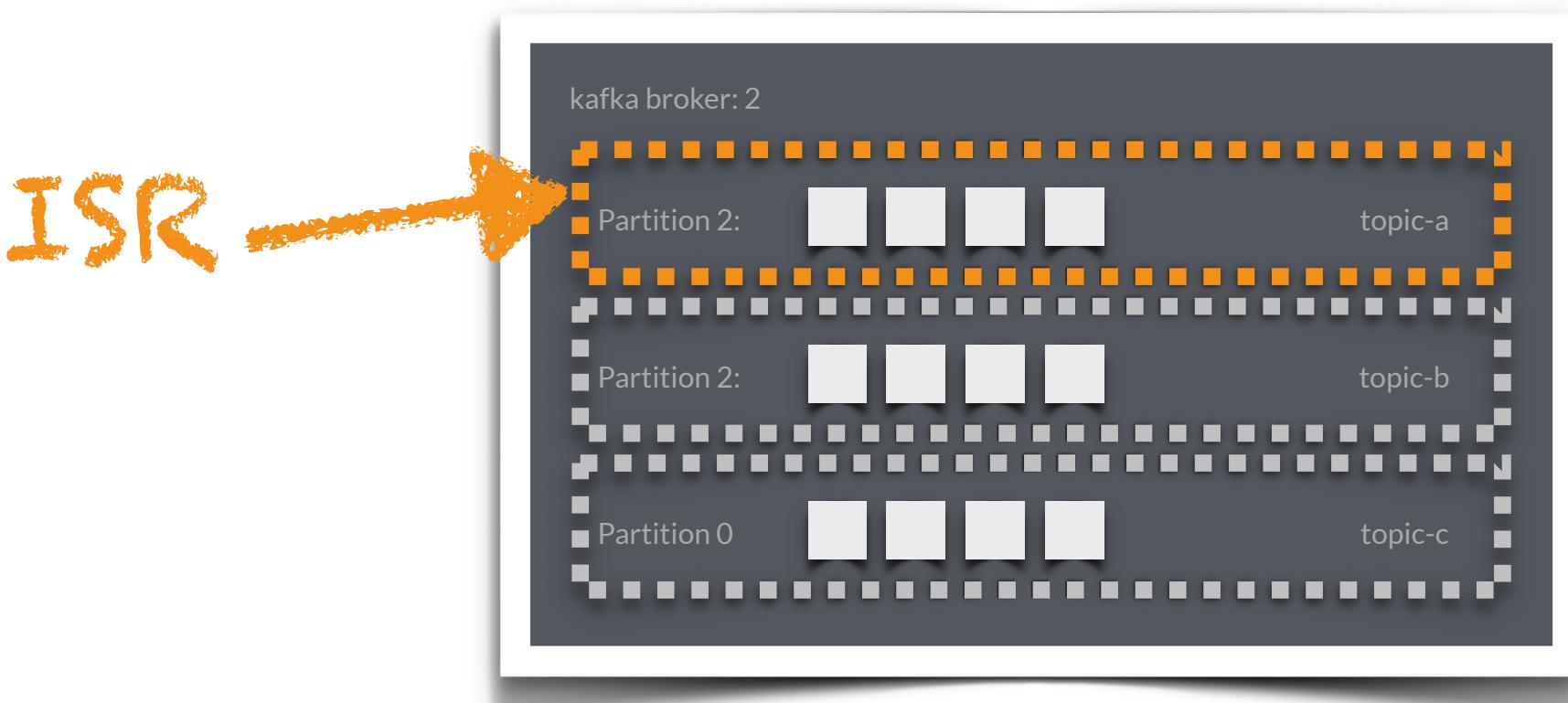
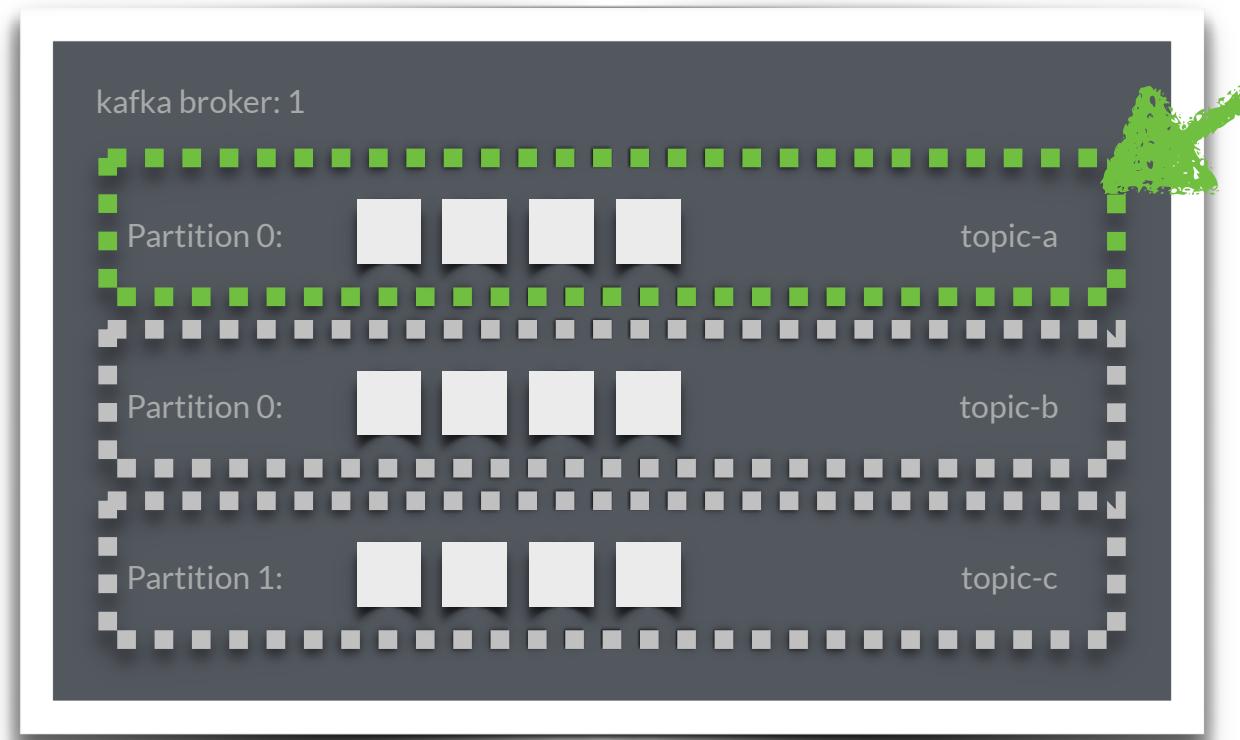
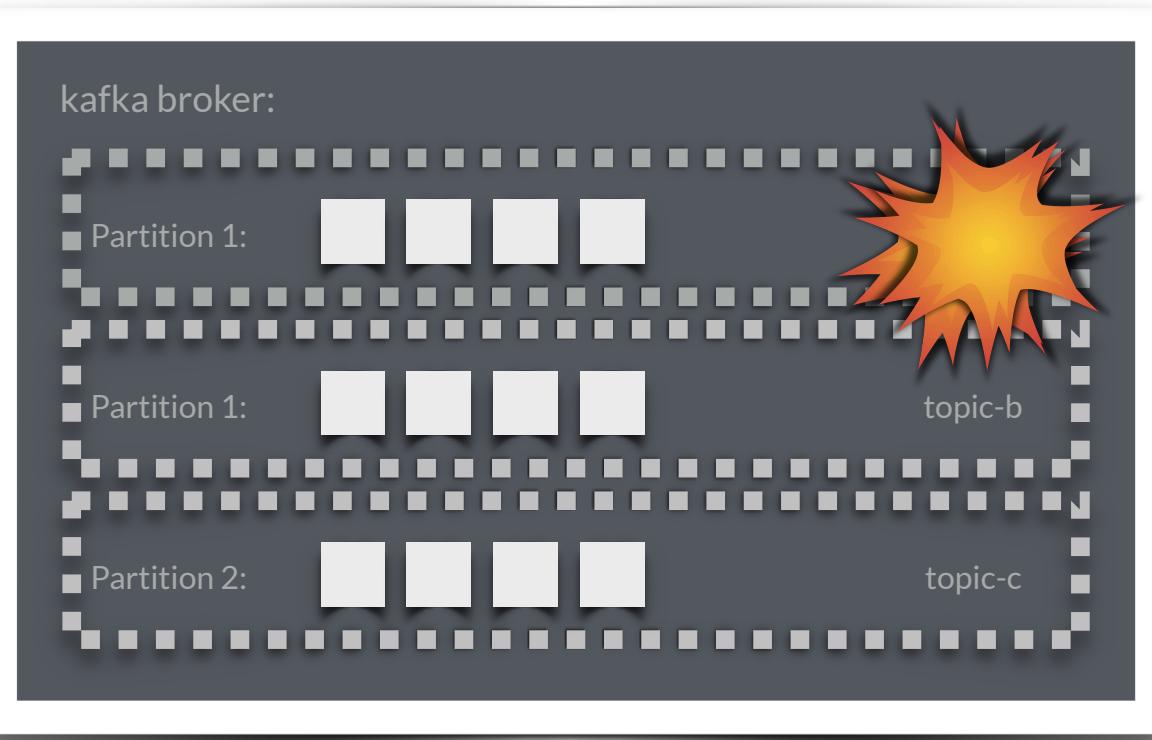
ISR



ISR



Leader



ISR

Exactly Once Semantics

**Mathias Verraes**

@mathiasverraes

Follow



There are only two hard problems in distributed systems:
2. Exactly-once delivery
Guaranteed order of messages
2. Exactly-once delivery

RETWEETS

6,775

LIKES

4,727

10:40 AM - 14 Aug 2015



69



6.8K

**4.7K**

Exactly Once Semantics

- If a stream processing application consumes message A and produces message B such that $B = F(A)$
- Exactly once processing means that A is considered consumed if and only if B is successfully produced, and vice versa.

How Exactly Once Processing is Lost (1)

- The producer `.send()` could result in duplicate writes of message *B* due to internal retries.
- This is addressed by the idempotent producer
- Exactly once processing means that *A* is considered consumed if and only if *B* is successfully produced, and vice versa.

How Exactly Once Processing is Lost (2)

- We may reprocess the input message A , resulting in duplicate B messages being written to the output
- Reprocessing may happen if the stream processing application crashes after writing B but before marking A as consumed.
- Thus when it resumes, it will consume A again and write B again, causing a duplicate.

How Exactly Once Processing is Lost (3)

- Finally, in distributed environments, applications will crash or—worse!—temporarily lose connectivity to the rest of the system.
- Typically, new instances are automatically started (like Kubernetes) to replace the ones which were deemed lost.
- Through this process, we may have multiple instances processing the same input topics and writing to the same output topics, causing duplicate outputs and violating the exactly once processing semantics.
- We call this the problem of “zombie instances.”



Requirements for EOS

- Transaction API
 - All of the messages included in the transaction will be successfully written or none of them will be.
- Zombie Fencing
 - We solve the problem of zombie instances by requiring that each transactional producer be assigned a unique identifier called the `transactional.id`.

transaction.id and epochs

- Producers register the Producer's transaction.id
- Identifies producer with a unique identifier across process restarts
- Registers transaction.id with broker
- Kafka broker checks for open transactions with the given transactional.id and completes them
- Increments an epoch associated with the transactional.id
- Any producers with same transactional.id and an older epoch are considered zombies and are fenced off, ie. future transactional writes from those producers are rejected.

Consumers and transactions

- Guarantees with consumers fall short
- Consumers will
 - not know whether these messages were written as part of a transaction
 - not know when transactions start or end
 - not guaranteed to be subscribed to all partitions which are part of a transaction
- Kafka guarantees that a consumer will eventually deliver only non-transactional messages or committed transactional messages

Transactions in the Producer

To "turn on" transactions in the Producer

- Add transaction id to identify the producer
- Initialize the transaction

```
Properties properties = new Properties();
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
    "localhost:9092");
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class);
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    IntegerSerializer.class);
properties.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "my-transaction-id");

producer.initTransactions();
```

Sending Transactionally

```
producer.initTransactions();

try {
    producer.beginTransaction();
    producer.send(record1);
    producer.send(record2);
    producer.commitTransaction();
} catch(ProducerFencedException e) {
    producer.close();
} catch(KafkaException e) {
    producer.abortTransaction();
}
```

Using transactions:

- Beginning Transaction with `beginTransaction`
- Send records as normal
- commit the Transaction
- If fenced, close it, nothing more we can do
- Other Exceptions, abort that transaction

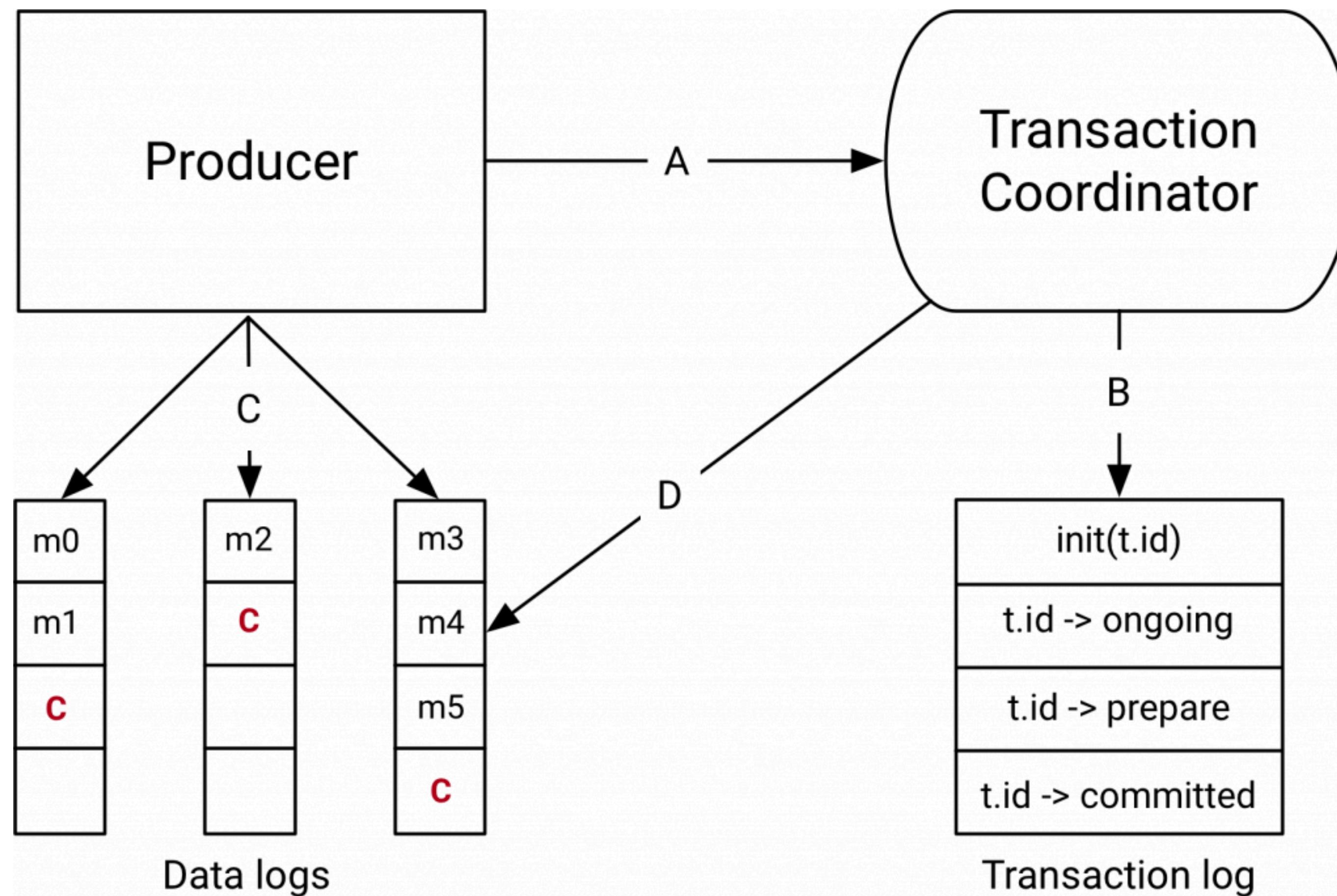
What about the consumer?

For the consumer, we will need to establish:

- `read_committed` - In addition to reading messages that are not part of a transaction, you can also read ones that are, after the transaction is committed
- `read_uncommitted` - Read all messages in offset order without waiting for transactions to be committed. This option is similar to the current semantics of a Kafka consumer.

```
Properties properties = new Properties();
properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
    "localhost:9092");
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "sf_giants");
...
properties.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");
```

How Transactions Work



Enable Idempotence

- We talk about transactions which is a part of ensuring EOS
- We also need to ensure that we handle idempotence
 - *Idempotence* - denoting an element of a set which is unchanged in value when multiplied or otherwise operated on by itself
 - Each batch of messages sent to Kafka will contain a sequence number that the broker will use to dedupe any duplicate send
 - Turn it on in the Producer with enable.idempotence=true

```
properties.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
```

Security

SSL/TLS

Kafka Authn, Authz, and Transport

- Authentication
 - One component can verify another component
- Authorization
 - After we authenticate who it is, we can then determine if we are allowed to let them in
- Transport
 - Are conversations secure.

Security in Kafka

- Authentication
 - Mutual TLS
 - SASL - Embedded Principals allowing access, outside-in
- Authorization
 - LDAP
 - ACL (managed by Kafka Tools and Zookeeper)
- Transport
 - TLS
 - Plain Text

Impact on Security Implementations

- Implementing TLS/SSL will impact performance
- Expect lower throughput
- Expect higher use of CPU and Broker
- <https://blog.mimacom.com/apache-kafka-with-ssltls-performance/>

Encryption at Rest

- Kafka encryption “at rest” is not out of the box
- Kafka is incapable for doing so
- Push Encryption to the Clients keeps the Brokers from having to do the encryption
- You can run Brokers on a machine with an encrypted filesystem or encrypted RAID controllers

Truststores vs Keystores

- Keystore is used to store *private key* and *identity certificates* that a specific program should present to both parties (server or client) for verification.
- Truststore is used to store certificates from *Certified Authorities (CA)* that verify the certificate presented by the server in SSL/TLS connection.

Client and Brokers

- Client Truststore
 - CA certificate used to verify signature on broker cert
 - If the signature is valid, the broker is trusted
- Broker Keystore
 - Uses private key to create a certificate
 - Certificate must be signed by a certificate authority
 - Certificate includes public key which client will use to establish secure connection

CA and TLS

- Broker creates public/private key pair
- Broker private key is used to generate a certificate
- Certificate is signed by the certificate authority by using the CA's own private key
- Broker then sends cert, including public key to client
- Client verifies the CA signature using CA certificate
- Once trusted, clients creates symmetric encryption to communicate with the broker. Communication uses the same shared key.

SSL/TLS Process

1. Generate certificate (X.509) in Broker keystore
2. Create your own Certificate Authority (CA) or use a well known CA
3. Sign Broker certificate with CA
4. Import signed certificate and CA certificate to Broker keystore
5. Import CA certificate to client truststore

Set up Properties Files

- Add truststores and keystore in every *.properties file

```
ssl.keystore.location = /var/private/ssl/kafka.server.keystore.jks  
ssl.keystore.password = password-to-keystore-file  
ssl.key.password = password-to-private-key  
ssl.truststore.location = /var/private/ssl/kafka.server.truststore.jks  
ssl.truststore.password = password-to-truststore-file
```

Broker Setup

Set up the following in the server.properties file of the server to setup SSL communication for brokers

```
listeners = SSL://<host>:<port>
ssl.client.auth = required
inter.broker.listener.name = SSL
```

- Broker Listeners can be setup with different protocols, even different names
<https://www.confluent.io/blog/kafka-listeners-explained/>
- ssl.client.auth is set to required to ensure that communicated is secure
- inter.broker.listener.name stipulates that inner-broker communication is also secure

Client Setup

- Set up the following in the producer.properties or consumer.properties of the server to setup SSL communication for clients.
- For more information visit: https://docs.confluent.io/platform/current/kafka/authentication_ssl.html#kafka-ssl-authentication

```
security.protocol = SSL
```

SASL

SASL

- SASL: Simple Authentication and Security Layer
 - Challenge/response protocols
 - Server issues challenge; client sends response
 - Continues until server is satisfied
- Preferred to send over the wire using TLS/SSL

SASL Mechanisms

- SASL supports varying mechanisms
 - GSSAPI: Kerberos
 - SCRAM-SHA-256, SCRAM-SHA-512
 - “salted” and hashed passwords
 - SCRAM (Salted Challenge Response Authentication Mechanism)
 - Username/password stored in ZK or delegation token (OAuth 2)
 - Plain Text
 - OAUTHBEARER - Token based authentication
 - Intended to be used with SSL/TLS

Role of Zookeeper

- Used for the credential store
 - Creates credentials for Brokers and clients
 - Credentials must be created before Brokers are started
 - Zookeeper should be secure
 - Zookeeper should be on a private network

Configuring SASL SCRAM

- Create credentials using kafka-configs
- Changes do not require a reboot

```
$ kafka-configs --bootstrap-server broker_host:port \
  --alter \
  --add-config \
  'SCRAM-SHA-256=[password=alice-secret],SCRAM-SHA-512=[password=alice-secret]' \
  --user alice
```

SASL using JAAS on the Broker

- JAAS (Java Authentication and Authorization Service) can be included in *.properties files on clients and Brokers using the sasl.jaas.config property
- Changes do not require a reboot

```
...
listeners=SASL_SSL://<host>:<port>
inter.broker.listener.name=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256
sasl.enabled.mechanisms=SCRAM-SHA-256
listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="admin" \
    password="admin-secret";
...
...
```

Kerberos on the Broker

- Kerberos provides secure single sign-on
 - An organization may provide multiple services
 - A user just needs a single Kerberos
- One password to use all services
 - More convenient where there are many users
 - Requires a Key Distribution Center (KDC)
 - Each service and each user must register a Kerberos principal in the KDC
- https://www.youtube.com/watch?v=_44CHD3Vx-0

SASL using JAAS on the Broker

- JAAS (Java Authentication and Authorization Service) can be included in *.properties files on clients and Brokers using the sasl.jaas.config property
- Changes do not require a reboot

```
...
listeners=SASL_SSL://<host>:<port>
inter.broker.listener.name=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256
sasl.enabled.mechanisms=SCRAM-SHA-256
listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="admin" \
    password="admin-secret";
...
```

Avro

Avro

- Created by Doug Cutting; Creator of Hadoop
- Serialization is defined by schema
- Schemas are JSON Based
- Codegen available at command line
- Codegen available by Maven, Gradle, SBT Plugin
- Supports the following languages:
 - C, C++, Java, Perl, Python, Ruby, PHP
- Named after WWI, WWII Aircraft Company, A.V. Roe



Types of Avro Serialization

- Generic
 - Develop code that reflects your schema
- Specific
 - Code generate your class from a schema, an `.avsc` file
 - This is the most common form
- Reflection
 - Auto-create schema from an existing class

Avro Primitive Types

Avro Type	Java Type
null	null
double	double
float	float
int	int
long	long
bool	bool
string	Unicode CharSequence
bytes	Sequence of 8-bit unsigned bytes

Avro Complex Types

```
{  
  "namespace": "com.xyzcorp",  
  "type": "record",  
  "doc" : "An music album",  
  "name": "Album",  
  "fields": [  
    {  
      "name": "name",  
      "type": "string"  
    },  
    {  
      "name": "yearReleased",  
      "type": [  
        "int",  
        "null"  
      ]  
    }  
  ]  
}
```

Avro Field Options

Avro Field	Description
name	Name of the Field
doc	Documentation
type	Type of the Field
default	Default Value
order	What order does this impact record
aliases	Any other names

Avro Array

List of Strings, but can be any type for items

```
{"type": "array", "items": "string"}
```

Avro Map

All keys in an Avro map are string

```
{"type": "map", "values": "long"}
```

Avro Enum

When autogen in Java, RainbowColors will be an enum

```
{ "type" : "enum",
  "name" : "RainbowColors",
  "doc" : "Colors of the Rainbow",
  "symbols" : ["RED", "ORANGE", "YELLOW",
               "GREEN", "BLUE", "INDIGO",
               "VIOLET"]}
```

Full Specification for Avro

- Read more about Avro, <https://avro.apache.org/docs/current/spec.html>
- Includes other types and feature:
 - UUID
 - Timestamp
 - Date/Time
 - Sorting
 - Codecs

Running Avro Tools

Avro Tools is a jar file that can be executed to generate Java from schema

```
java -jar avro-tools-1.8.2.jar compile schema <schema file> <destination>
```

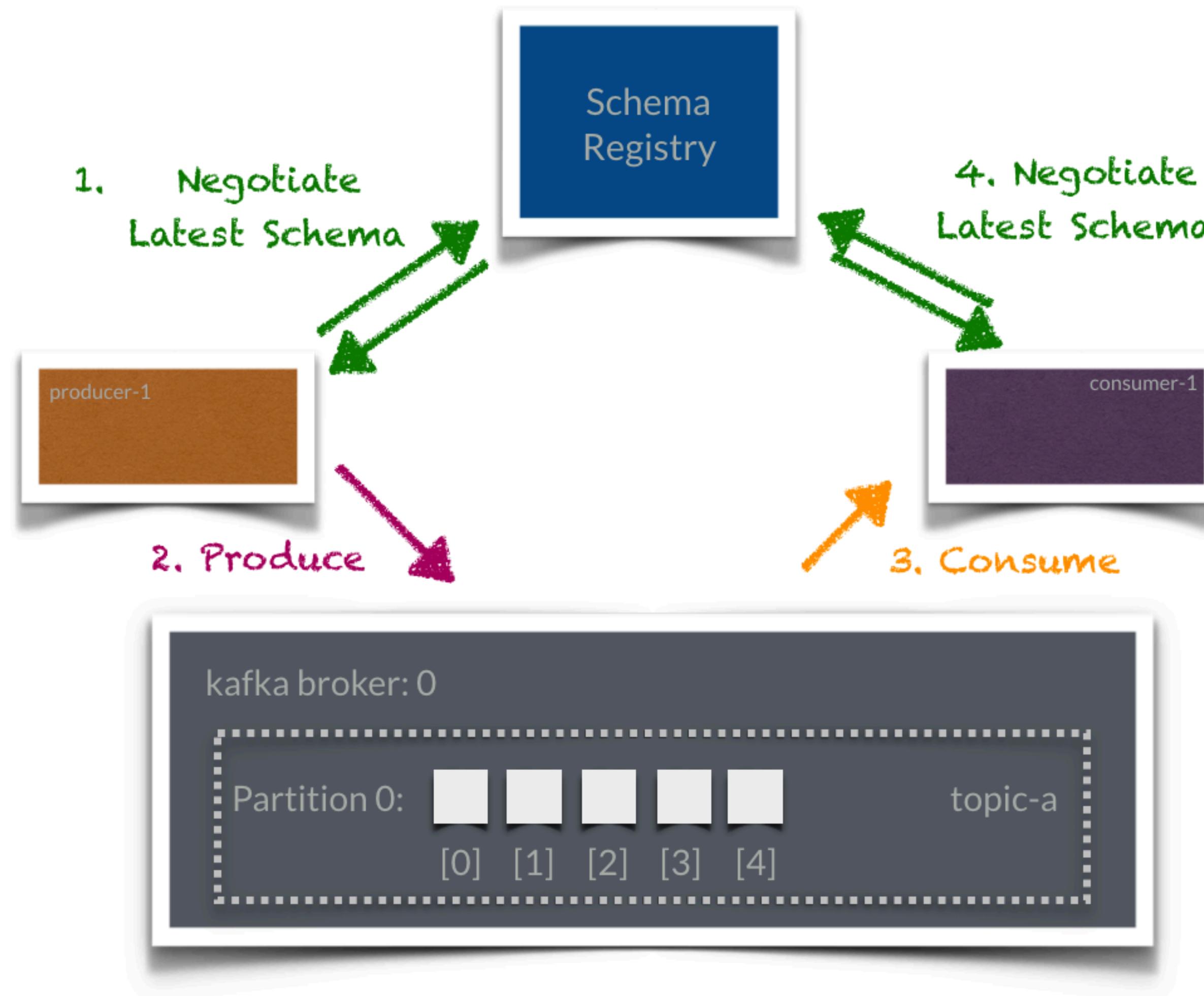
Avro Tools is available for most IDEs like Maven, and Gradle

Schema Registry

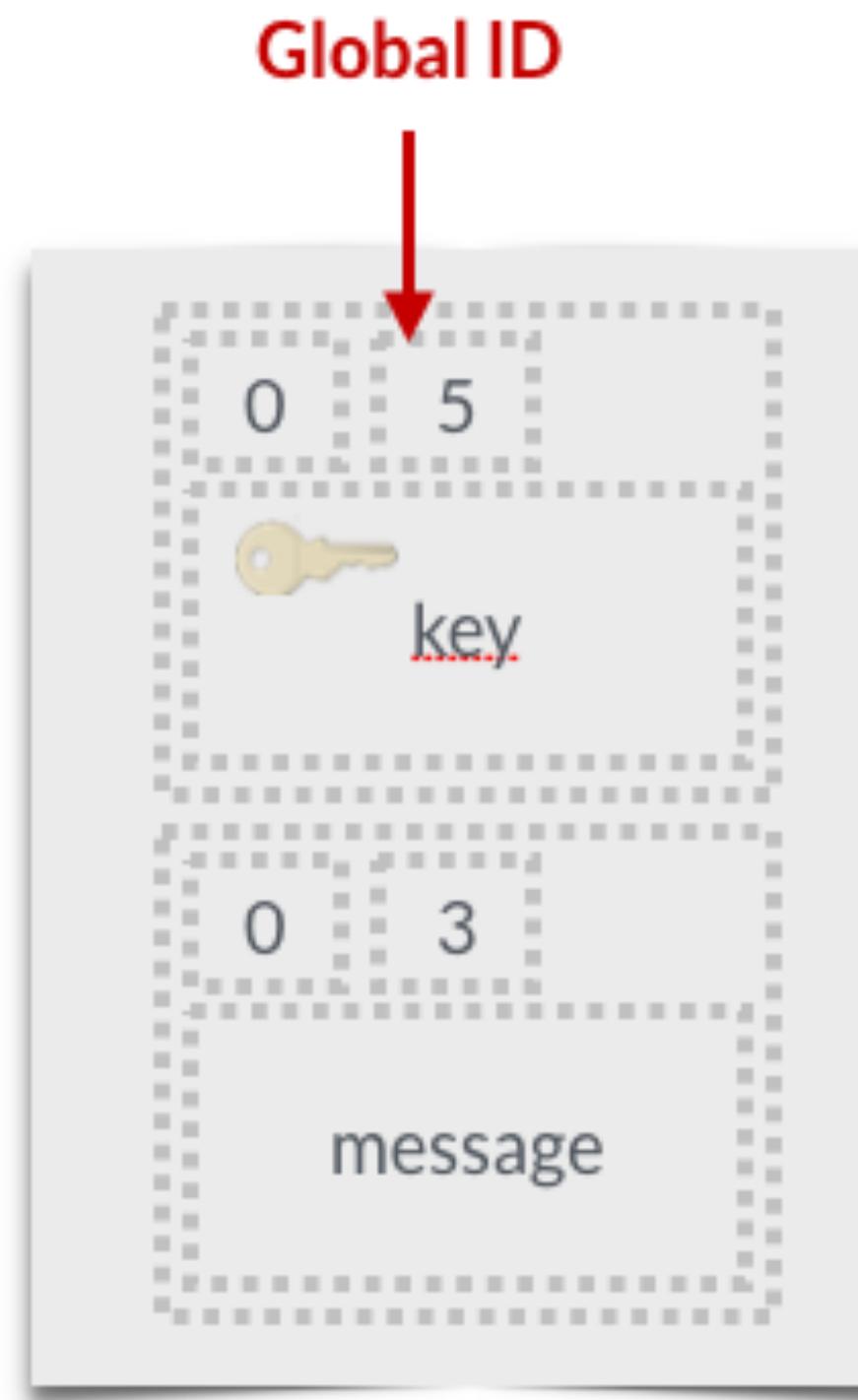
Schema Registry

- Service that constrains and typechecks messages before sending
- Stores schemas in registry for and assigns a global ID
- Ensures *backward* and *forward* compatibility
- Does so through Serializers and Deserializers
- Automatically done through Java Kafka Client API

Schema Registry



Schema Registry Message Format



- The first byte is a zero byte
- The second byte is the global identifier of the Schema that registered in the Schema Registry
- The Global ID is used to as a means of communicating which Schema to use with messages

Registering a New Schema Manually

```
POST /subjects/my_topic-key/versions HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json,
         application/vnd.schemaregistry+json,
         application/json

{
  "schema":
    "{
      \"type\": \"record\",
      \"name\": \"test\",
      \"fields\": [
        {
          \"type\": \"string\",
          \"name\": \"field1\"
        },
        {
          \"type\": \"int\",
          \"name\": \"field2\"
        }
      ]
    }"
}
```

- Posts to a subject
- A subject is combination of
 - Topic
 - -key or -value
- Example of a subject
 - stocks-key
 - sales-value



Note: We aren't going manually do this we just wanted to show what an avro producer will do

Getting Response from the Registry

```
HTTP/1.1 200 OK
```

```
Content-Type: application/vnd.schema.registry.v1+json
```

```
{"id": 1}
```



The {"id": 1} is the global id, this is used as reference to this schema

Schema Registry Java Producer

```
Properties properties = new Properties();
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
    "localhost:9092");
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class);
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    io.confluent.kafka.serializers.KafkaAvroSerializer.class);
properties.setProperty("schema.registry.url", "http://localhost:8081");
```

- What makes schema registry work is the Serializer
- You must add schema.registry.url and specify the location of the registry

Example of Sending Custom Avro Object

```
Album album = new Album("Purple Rain", "Prince", 1984,  
    Arrays.asList("Purple Rain", "Let's go crazy"));  
  
ProducerRecord<String, Album> producerRecord =  
    new ProducerRecord<>("music_albums", "Prince", album);  
  
producer.send(producerRecord);
```

- The ProducerRecord's value is accepting an actual Java Object
- We can just send the object just like any other type

Schema Registry Consumer

```
Properties properties = new Properties();
properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "my_group");
properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringDeserializer");
properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    io.confluent.kafka.serializers.KafkaAvroDeserializer.class);
properties.setProperty("schema.registry.url", "http://localhost:8081");
properties.setProperty("specific.avro.reader", "true");
```

- What makes schema registry work is the Deserializer
- You must add schema.registry.url and specify the location of the registry
- You must add specific.avro.reader and specify that you used specific avro mode

Receiving a Custom Object from Kafka

```
while (true) {
    ConsumerRecords<String, Album> records =
        consumer.poll(Duration.of(500, ChronoUnit.MILLIS));
    for (ConsumerRecord<String, Album> record : records) {
        System.out.format("offset: %d\n", record.offset());
        System.out.format("partition: %d\n", record.partition());
        System.out.format("timestamp: %d\n", record.timestamp());
        System.out.format("timeStampType: %s\n",
                          record.timestampType());
        System.out.format("topic: %s\n", record.topic());
        System.out.format("key: %s\n", record.key());
        Album a = record.value();
        System.out.format("value: %s\n", a.getTitle());
        System.out.format("value: %s\n", a.getArtist());
    }
}
```

Lab: Using Avro & Schema Registry

Streaming Concepts



Stream Processing

Endless supply of data

“Replayable”

Real Time Processing for Fraud Detection, High End Sales Detection, Internet of Things, and More.

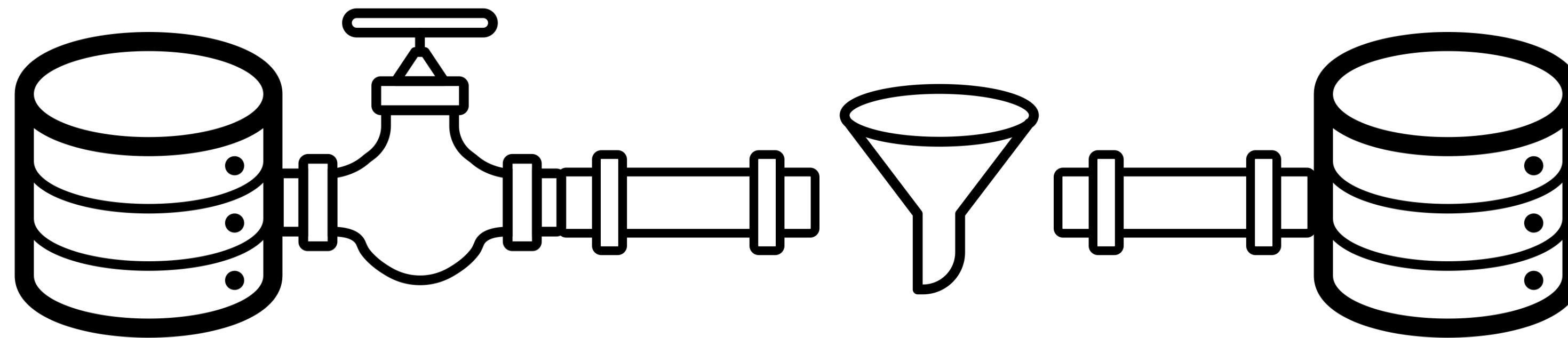
Will require built-in accumulator table to perform real time data processing, like counting, and grouping

NY

100.00

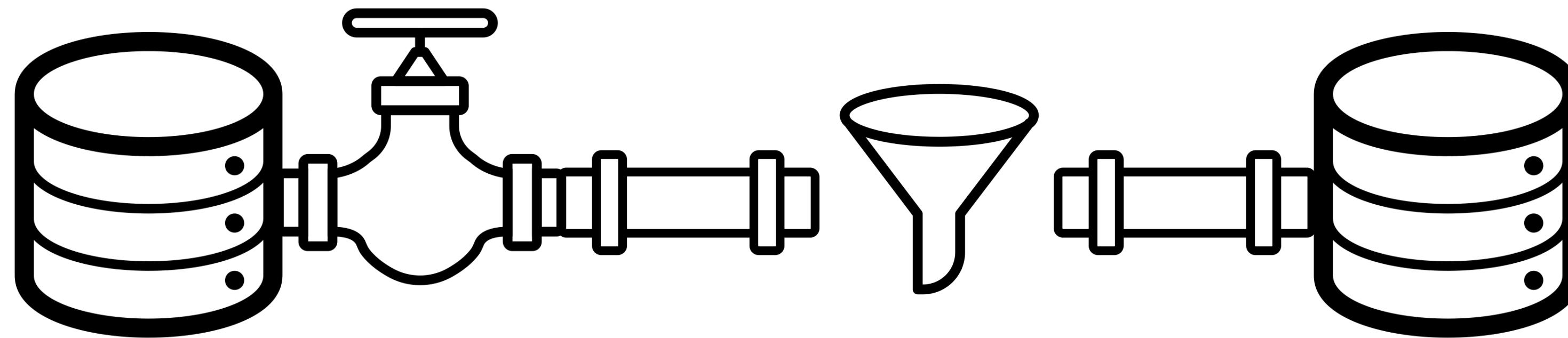
Filtering

NY
.....
100.00



value > 10000

OH
.....
20000

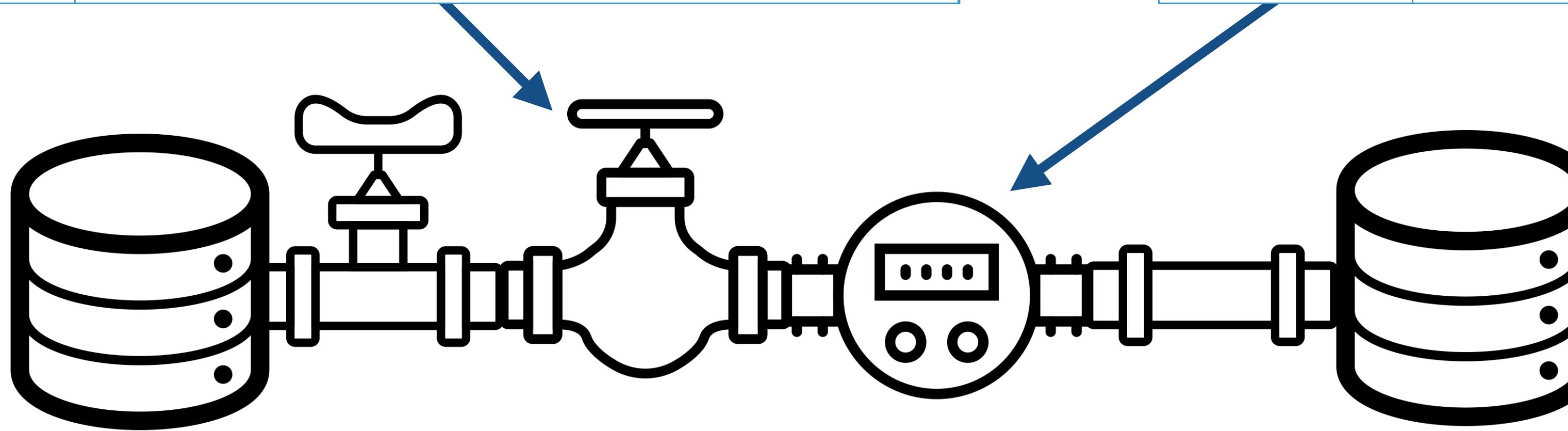


value > 10000

Aggregating

Key	Value
OH	100.0

Key	Value
OH	100.0

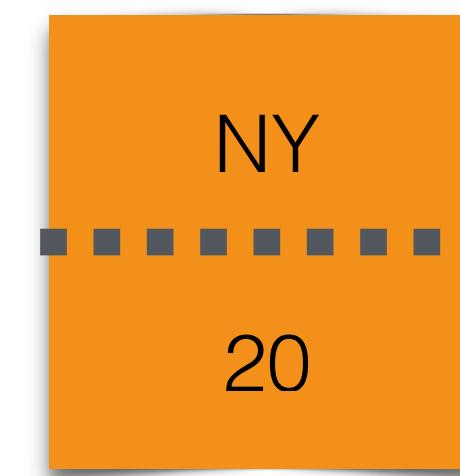
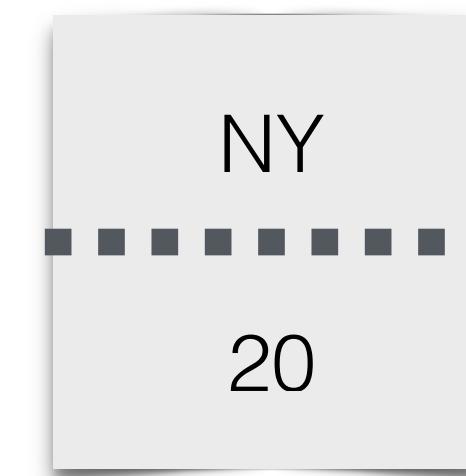
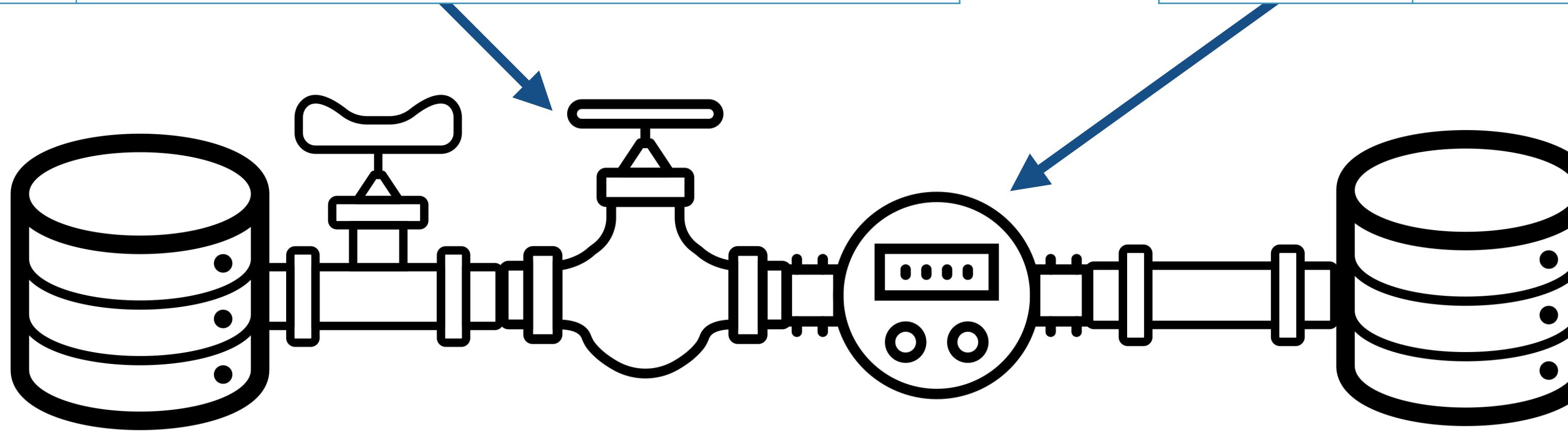


OH
.....
100.00

OH
.....
100.00

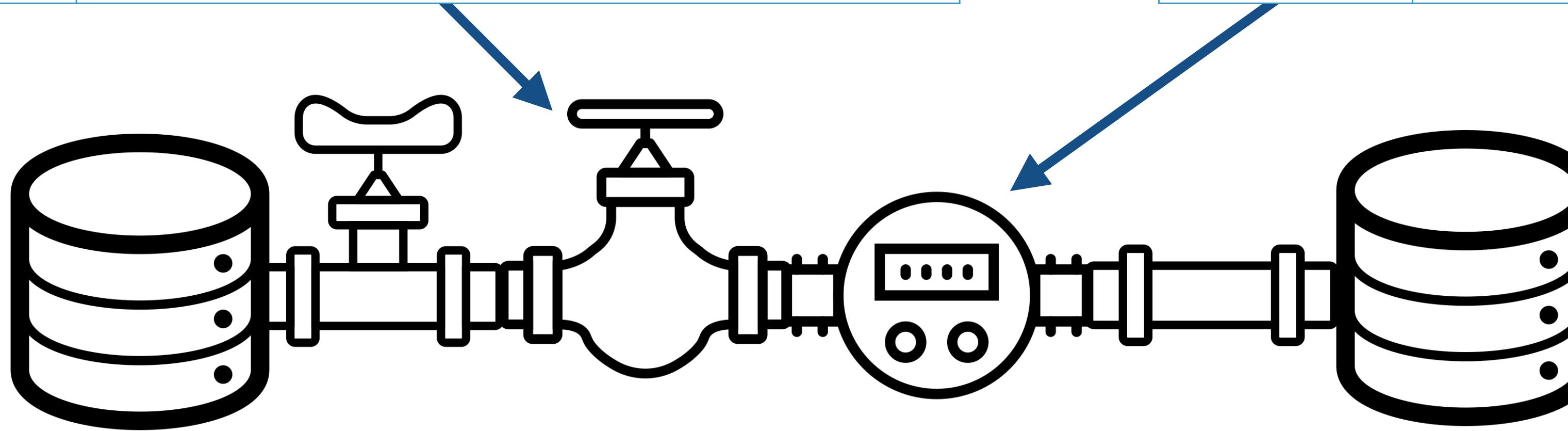
Key	Value
OH	100.0
NY	20.0

Key	Value
OH	100.0
NY	20.0



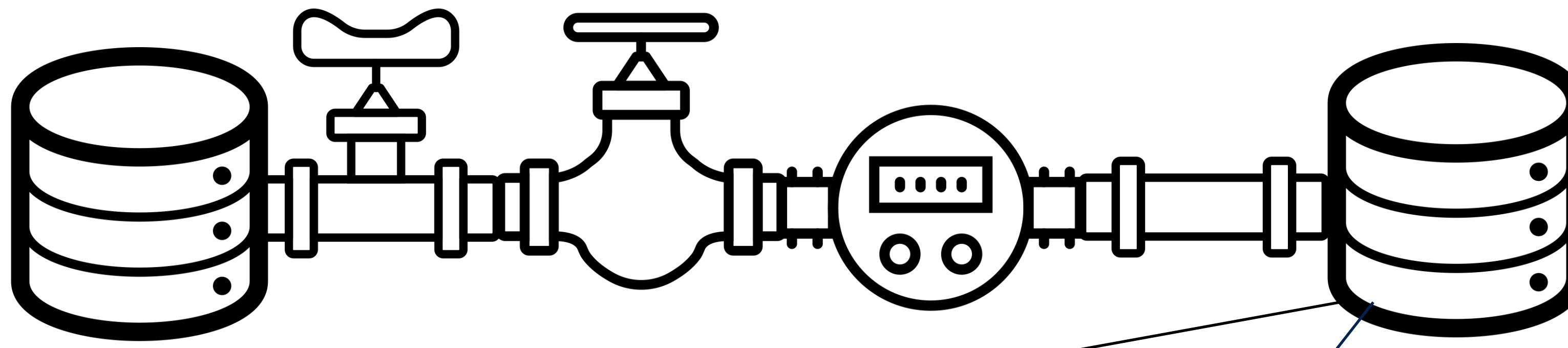
Key	Value
OH	100.0
NY	20.0, 60.0

Key	Value
OH	100.0
NY	80.0



NY
.....
60

NY
.....
80.0

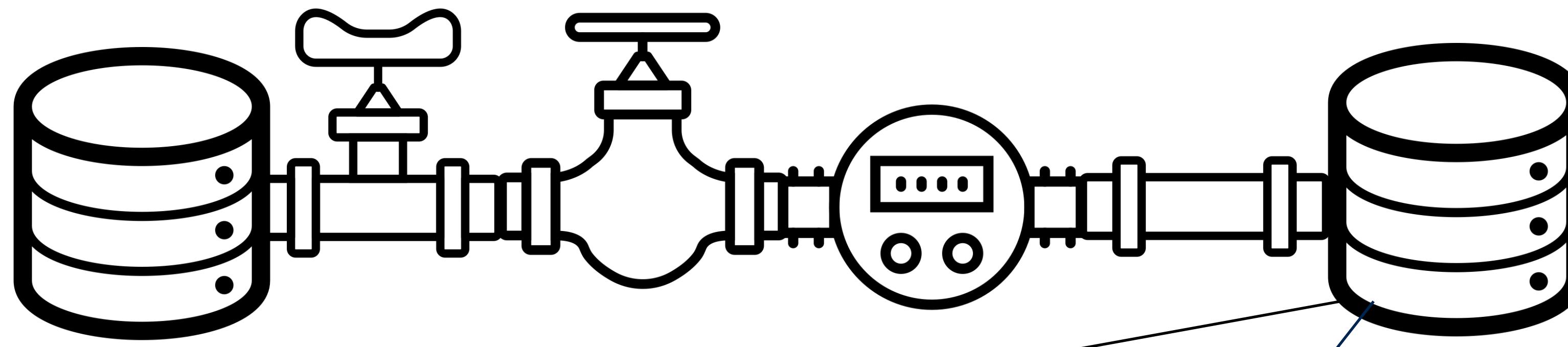


Offset	Key	Value
0	OH	100.0

Partition 0

Offset	Key	Value
0	NY	20.0
1	NY	80.0

Partition 1



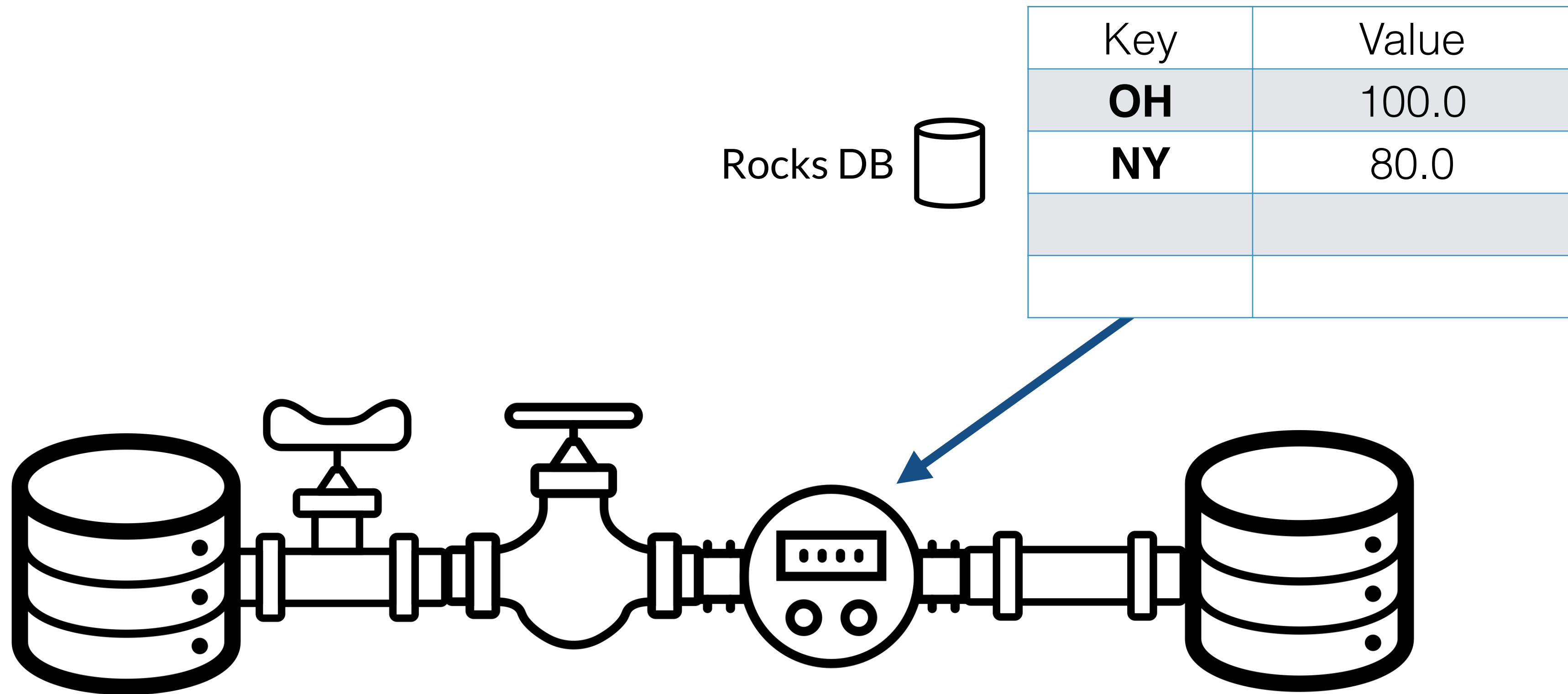
Offset	Key	Value
0	OH	100.0

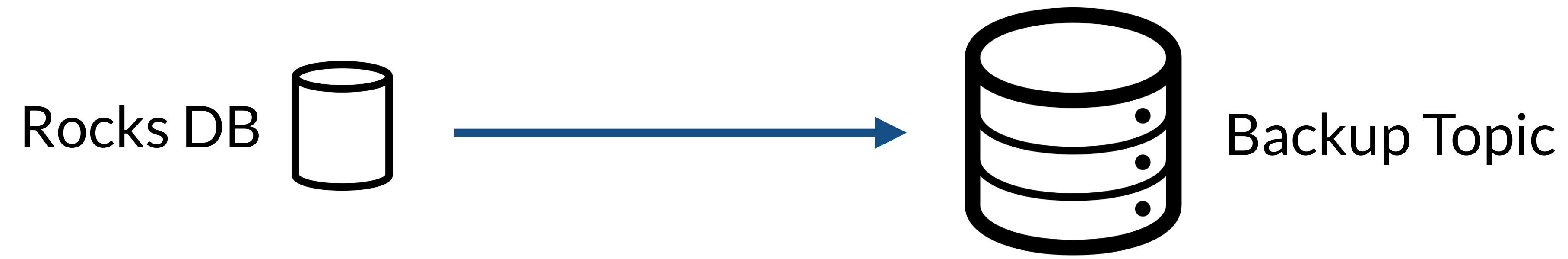
Partition 0

Offset	Key	Value
1	NY	80.0

Partition 1

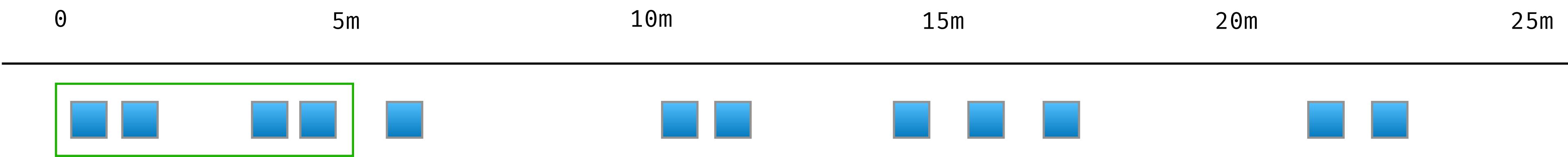
After Compaction



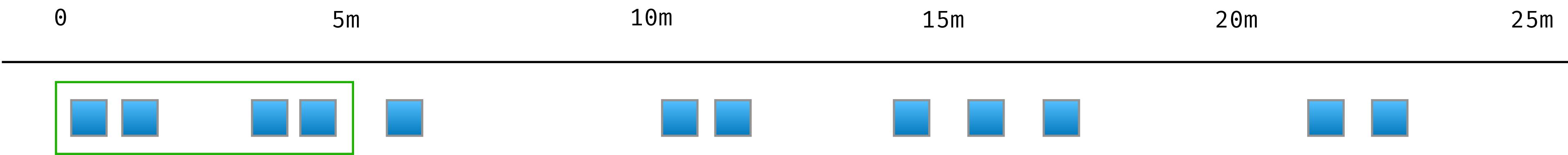


Windowing

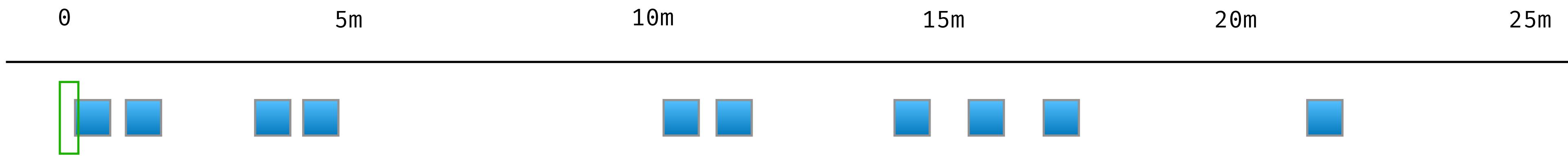
Tumbling Window



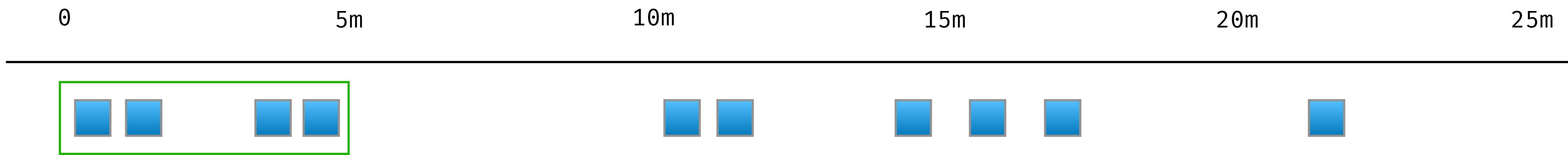
Hopping Window



Session Window

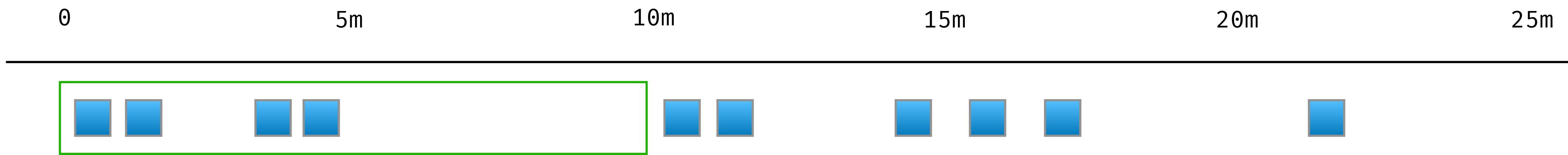


Session Window



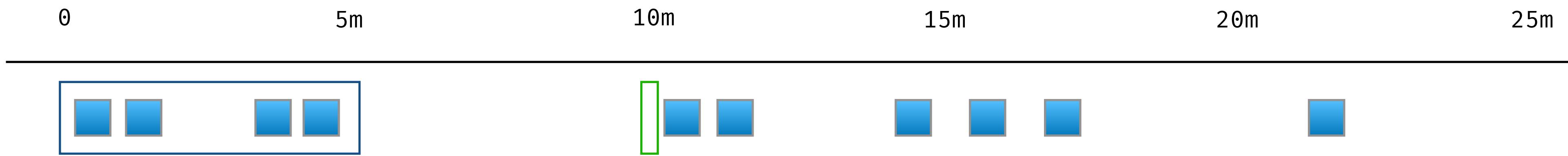
the last element has occurred; now we wait

Session Window



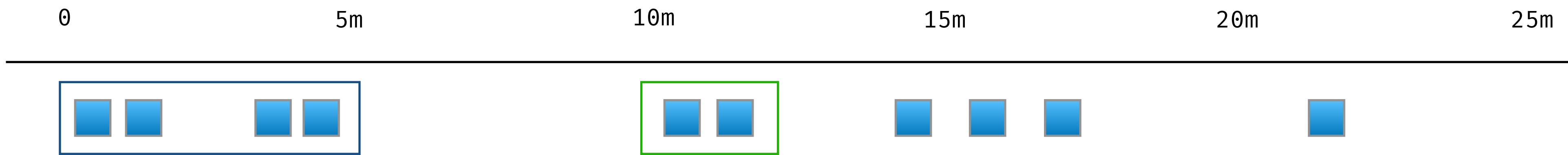
waited 5 minutes = make it a window

Session Window



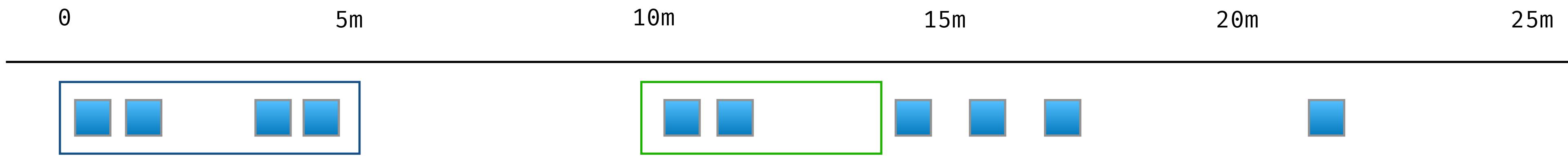
we have a new element; we start a new window

Session Window



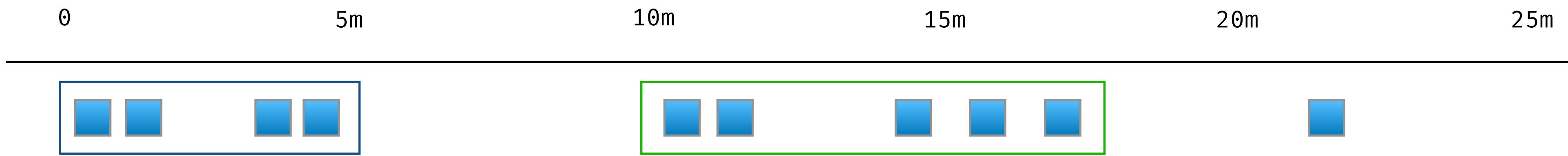
we wait for 5 minutes

Session Window



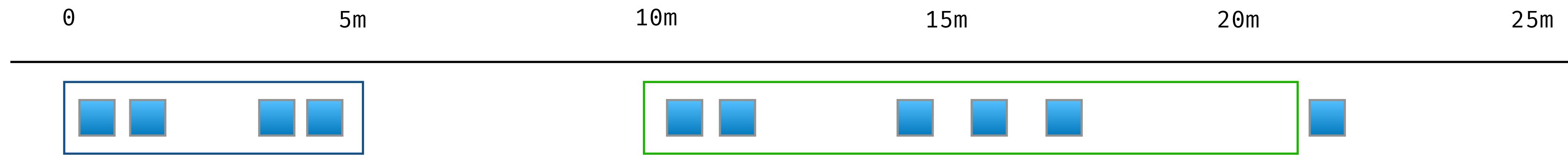
2 minutes elapsed; no windowing yet

Session Window



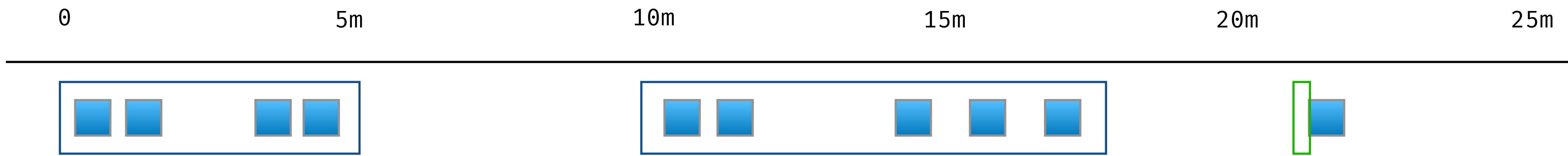
we wait five minutes

Session Window



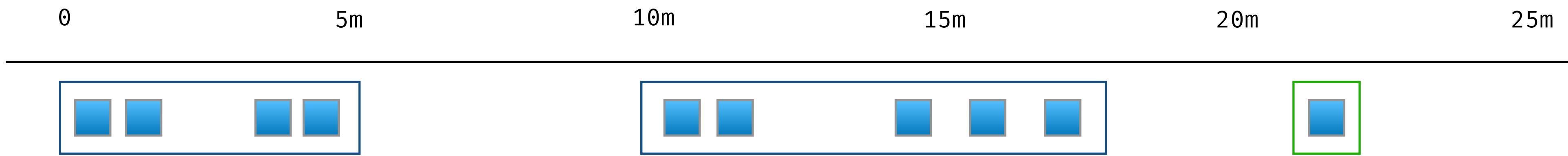
no new elements; we make it a window

Session Window



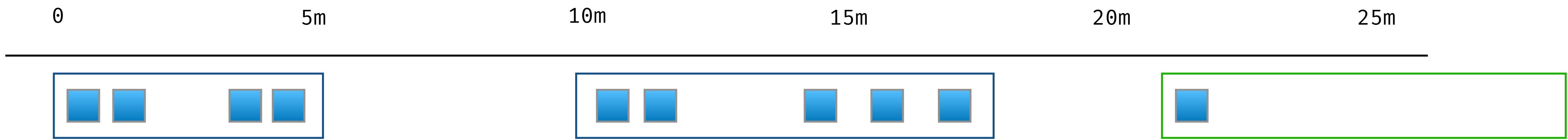
we see something new

Session Window



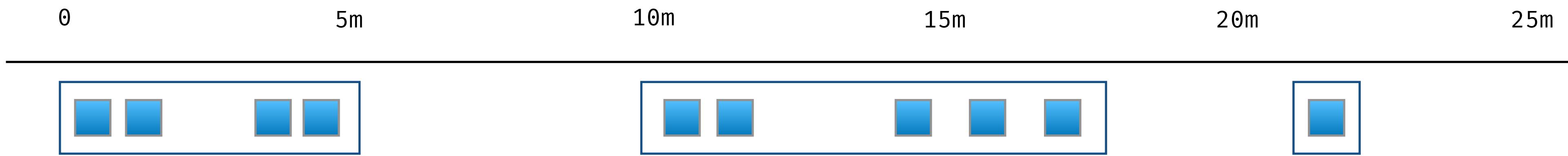
last element; we wait five minutes

Session Window



nothing else; we wait 5 minutes

Session Window



that's a new window

Stream API

Establishing Properties

- Create an application.id that represents your application group or "team"
- Serde is combination of Serializer/Deserializer
- Every stream application and KSQL app (later) is a consumer-producer

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "my_stream_app");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.Integer().getClass());
```

Create a Stream Builder

- Always start with a StreamBuilder object
- This is the GoF builder pattern, where we will create a Topology object that represents our data pipeline

```
StreamsBuilder builder = new StreamsBuilder();
```

Read from a Topic

- Recall there are two representations of how to ingest data from a topic
 - Stream - Flow of data
 - Table - An aggregation; Or a database
- How do you read the topic is up to you
- Result is either a stream or table object that can take functional programming DSL.

```
KStream<String, Integer> stream = builder.stream("my_topic");
```

```
KTable<String, Integer> table = builder.table("my_topic");
```

Standard Functional Programming

- map
- filter
- flatMap
- groupBy
- reduce
- window
- join
- leftJoin
- outerJoin

Mapping

- Given a Stream

```
(1, "Hello"), (2, "Zoom"), (3, "Fold")
```

- Applying map

```
stream.map((key, value) -> new KeyValue<>(key + 1, value + "!"));
```

- Resulting in

```
(2, "Hello!"), (3, "Zoom!"), (4, "Fold!")
```

Filtering

- Given a Stream

```
(1, "Hello"), (2, "Zoom"), (3, "Fold"), (4, "Past")
```

- Applying filter

```
stream.filter((key, value) -> key % 2 == 0);
```

- Resulting in

```
(2, "Zoom"), (4, "Past")
```

FlatMap

- Given a Stream

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

- Applying flatMap

```
stream.flatMap((key, value) -> Arrays.asList(  
    new KeyValue(key, value),  
    new KeyValue(key * 100, value + " Hundred")))
```

- Resulting in

```
(1, "One"), (100, "One Hundred"), (2, "Two"), (200, "Two Hundred"), (3,  
"Three"), (300, "Three Hundred"), (4, "Four"), (400, "Four Hundred")
```

Peek

- Peers into the Stream and view what goes through

```
stream1.peek((key, value) ->  
    System.out.printf("key: %s, value: %d", key, value));
```

GroupBy

- Given a Stream

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

- Applying groupBy

```
stream.groupBy((key, value) -> key % 2 == 0 ? "Even", "Odd")
```

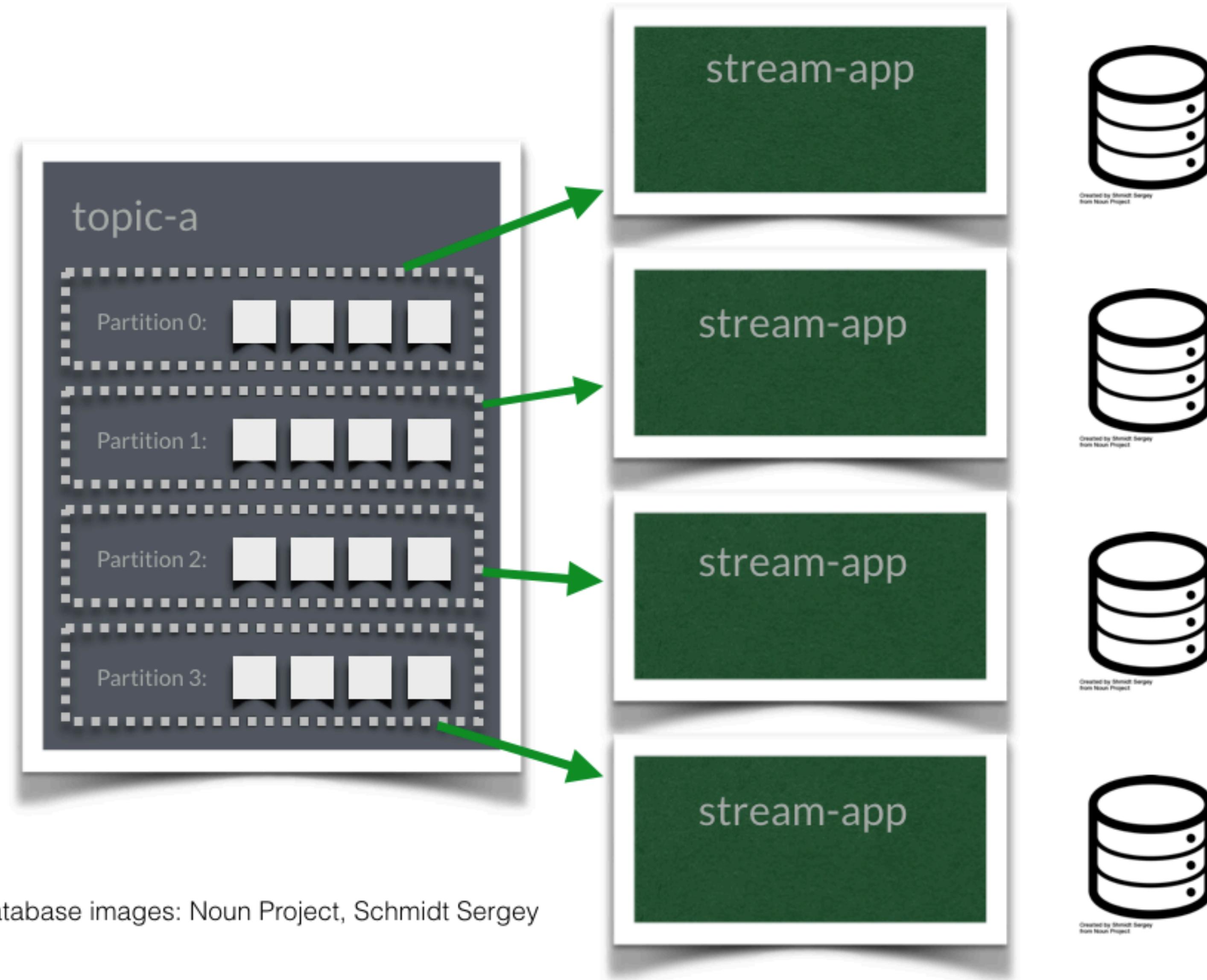
- Resulting in KGroupedStream

Once we have KGroupedStream

- `groupBy` creates a `KGroupedStream` object
- You have to decide what to do with this group
 - `count`
 - `aggregate`
 - `reduce`
 - `windowBy`
- Once you call any of the above, you have a `KTable`

KTable

- Represents a changelog, where the oldest records are not important
- Ideal for counters, and state
- KTable values are stored at each application's ephemeral storage
- Most ephemeral storage is backed by Rocks DB
- All streaming is then backed up as a topic for resilience



GroupByKey

- Given a Stream

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

- Applying groupByKey

```
stream.groupByKey()
```

- Resulting in KGroupedStream
- Is Synonymous with the Following

```
stream.groupBy((key, value) -> key)
```

Count

- Aggregates the count of what has been grouped
- Records with null key or value are ignored
- Returns KTable that represents the latest rolling count
- Will maintain the count in ephemeral storage, (like RocksDB)

Count

- Given a Stream

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

- Applying groupBy with count

```
stream.groupBy((key, value) -> key % 2 == 0 ? "Even" : "Odd").count();
```

- Will result in the RocksDB ephemeral storage

```
("Even", 2)  
("Odd", 2)
```

Reduce

- Given a Stream

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

- Applying groupBy with count

```
stream.groupBy((key, value) -> key % 2 == 0 ? "Even" : "Odd")
    .reduce((value1, value2) -> value1 + "," + value2);
```

- Will result in the RocksDB ephemeral storage

```
("Even", "Two,Four")
("Odd", "One,Three")
```

Aggregate

- Given a Stream

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

- Applying groupBy with count

```
groupedStream.aggregate(() -> "0:Zero",
(nextKey, nextValue, aggString) -> aggString + nextKey + ":" + nextValue ",");
```

- Will result in the RocksDB ephemeral storage

```
("Even", "0:Zero,2:Two,4:Four")
("Odd", "0:Zero,1:One,3:Three")
```

Hopping Windows

- Creating a Time Window

```
import java.util.concurrent.TimeUnit;
import org.apache.kafka.streams.kstream.TimeWindows;

long windowSizeMs = TimeUnit.MINUTES.toMillis(5); // 5 * 60 * 1000L
long advanceMs =    TimeUnit.MINUTES.toMillis(1); // 1 * 60 * 1000L
TimeWindows.of(windowSizeMs).advanceBy(advanceMs);
```

- Applying the Window

```
streams.groupBy(...).windowedBy(TimeWindows.of(windowSizeMs).advanceB
y(advanceMs));
```

Session Windows

- Creating a Time Window

```
import java.util.concurrent.TimeUnit;
import org.apache.kafka.streams.kstream.SessionWindows;

SessionWindows.with(TimeUnit.MINUTES.toMillis(5));
```

- Applying the Window

```
streams.groupBy(...).windowedBy(SessionWindows.with(TimeUnit.MINUTES.
toMillis(5)));
```

Dump results to a topic

- Dump the results to a topic using to

```
KStream<String, Integer> stream = builder.stream("my_topic");  
stream.filter(...).to("new_topic");
```

- Dump the results to a topic using through to post to topic and continue:

```
KStream<String, Integer> stream = builder.stream("my_topic");  
stream.filter(...).through("new_topic").flatMap(...).to("other_topic")
```

Build the Topology and Stream

- Once you create your stream, you can build a Topology object:

```
Topology topology = builder.build();
```

- Create a Kafka Streams object with both Topology and Properties you created earlier

```
KafkaStreams streams = new KafkaStreams(topology, props);
```

- And Run...

```
streams.start();
```

Adding a Shutdown Hook

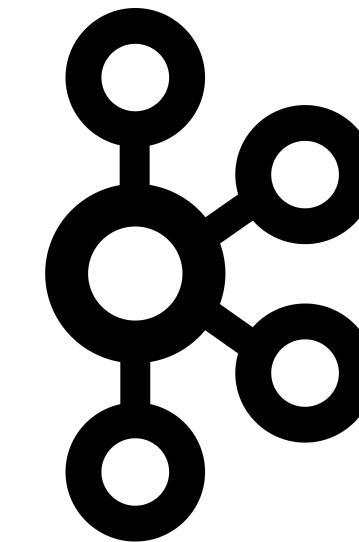
- As always, be a good citizen, properly shutdown resources

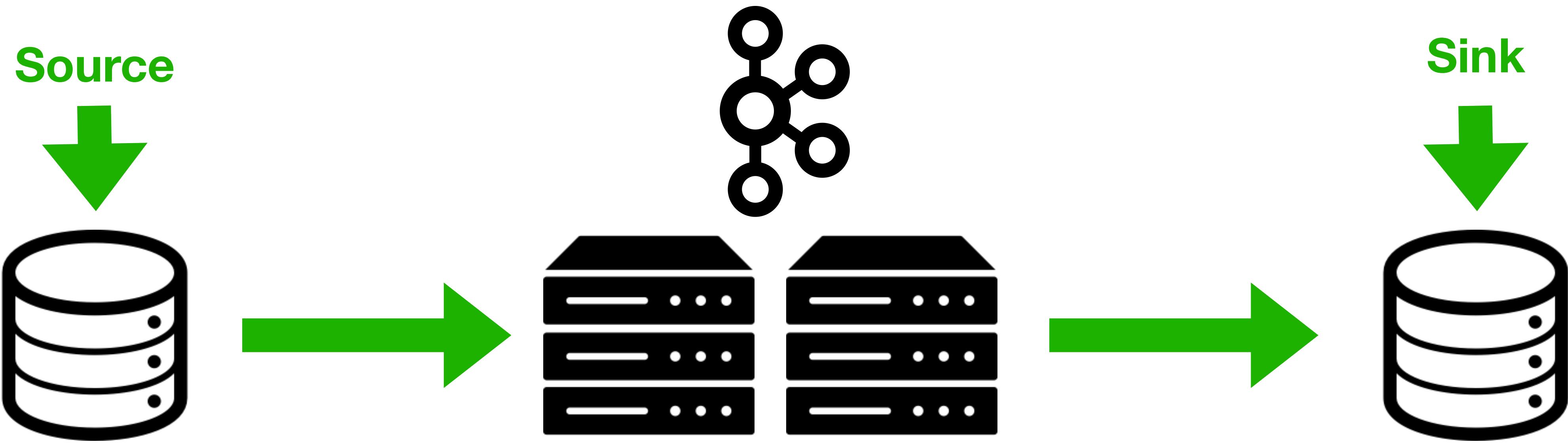
```
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

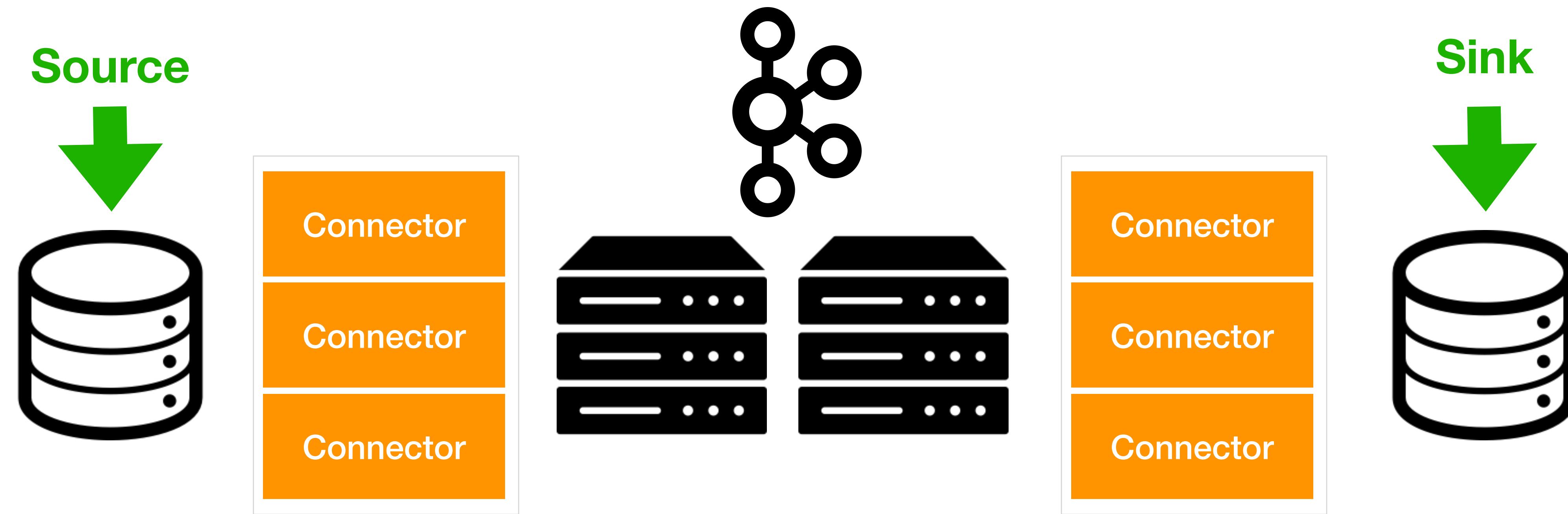
Kafka Connect

Kafka Connect

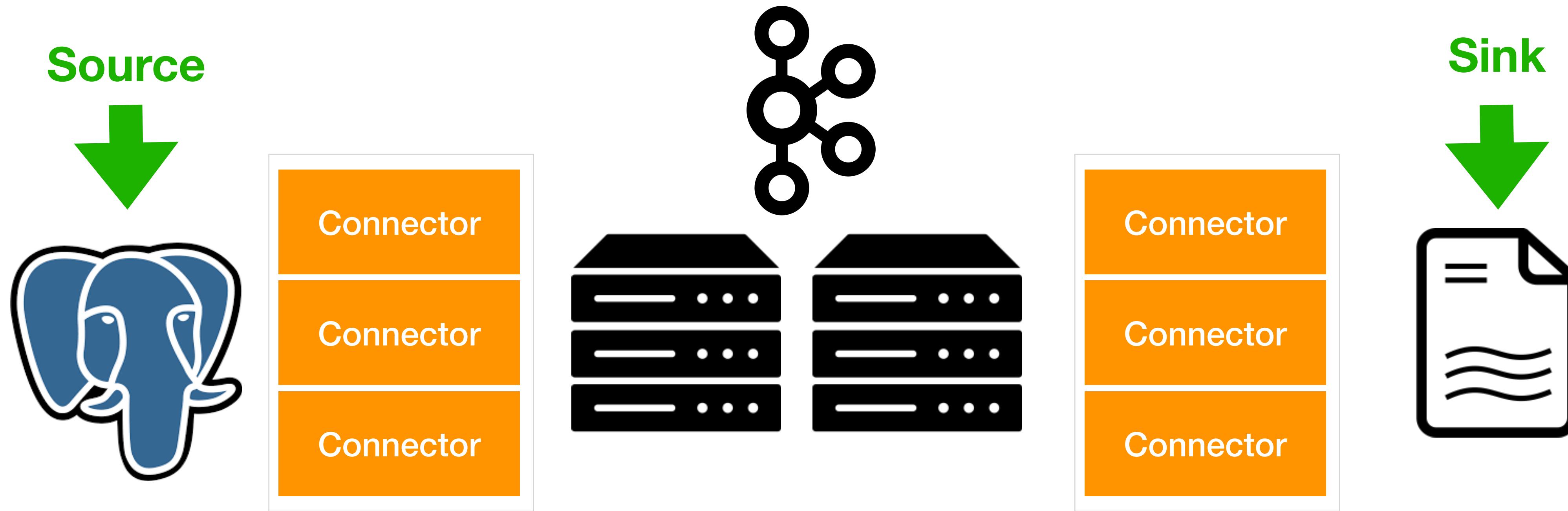
- Automatic Scaling
- Fault Tolerance
- All Configuration
- Pre-Engineered
- Transformations
- Confluent Community License



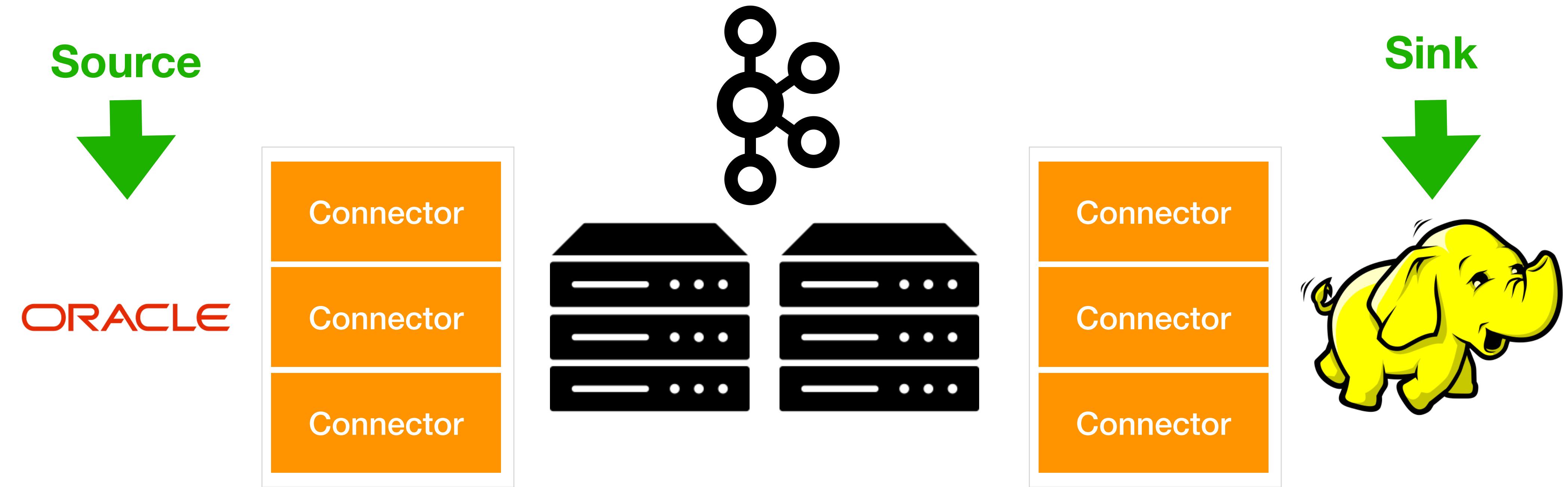




Kafka Connect is an Open Source Framework to stream data to and from Kafka



Connect a PostgreSQL database to a single file



Connect an Oracle database to Hadoop

[Product](#)[Cloud](#)[Developers](#)[Blog](#)[Docs](#)[Download](#)

Verified Standard

**CockroachDB Change
Data Capture**

Cockroach Labs

[Read More](#)

Verified Standard

**Vertica Analytics
Platform**

Vertica

[Read More](#)

Confluent Supported

**Debezium MySQL CDC
Connector**

Debezium Community

[Read More](#)

Confluent Supported

**Debezium PostgreSQL
CDC Connector**

Debezium Community

[Read More](#)

Confluent Supported

**Debezium SQL Server
CDC Connector**

Debezium Community

[Read More](#)

Confluent Supported

**Debezium MongoDB
CDC Connector**

Debezium Community

[Read More](#)

Confluent Supported

**Kafka Connect ActiveMQ
Source**

Confluent, Inc.

[Read More](#)

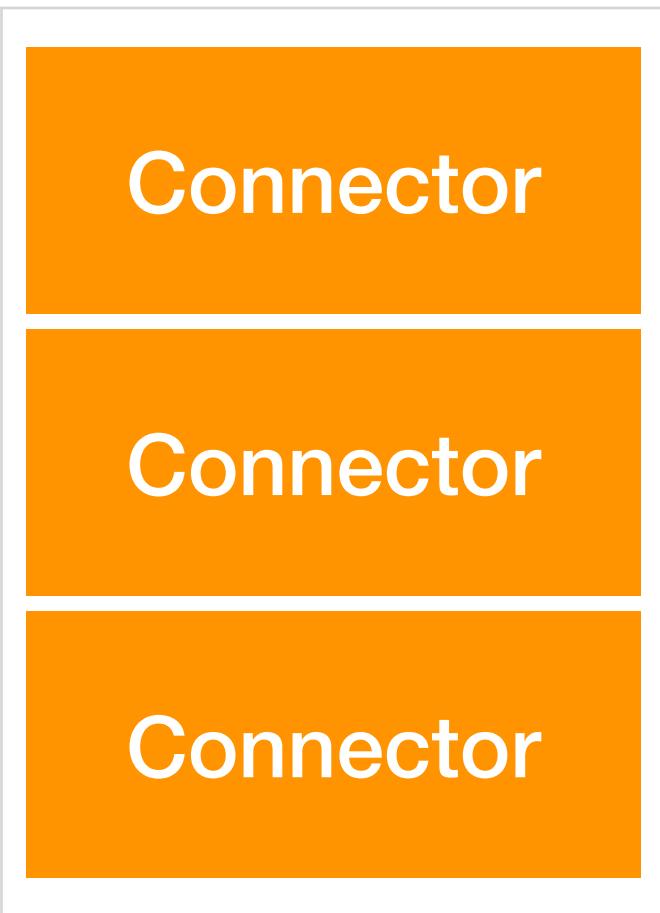
Confluent Supported

**Kafka Connect AWS
CloudWatch Logs Source
Connector**

Confluent, Inc.

[Read More](#)<https://confluent.io/hub>

Connectors



- **Logical Jobs** that copy the data to and from databases
- Internally a *source connector* is a Producer
- Internally a *sink connector* is a Consumer



Kafka Connectors

Confluent Open Source Connectors

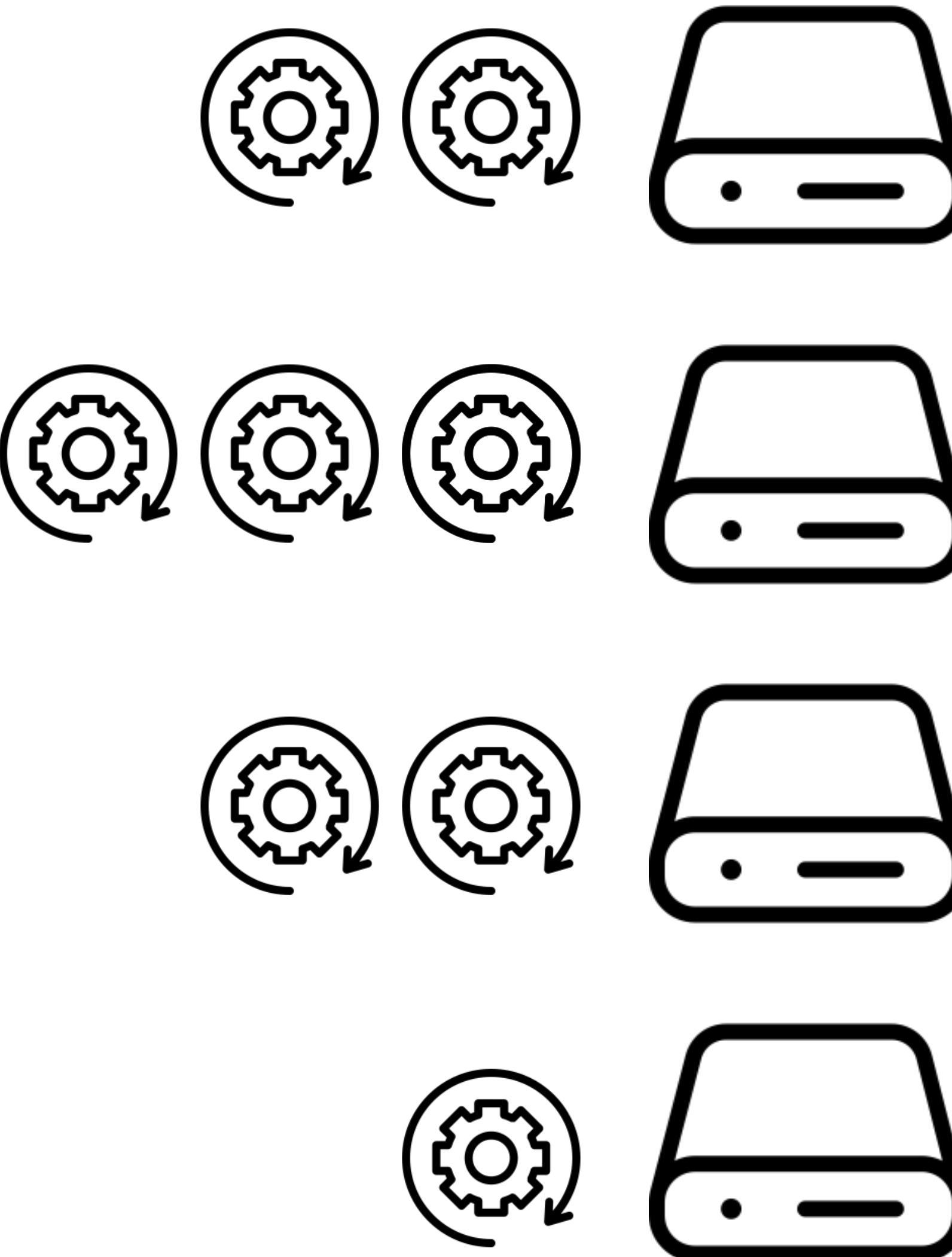
File Stream - JDBC - HDFS - S3 - Elastic Search

Confluent Enterprise

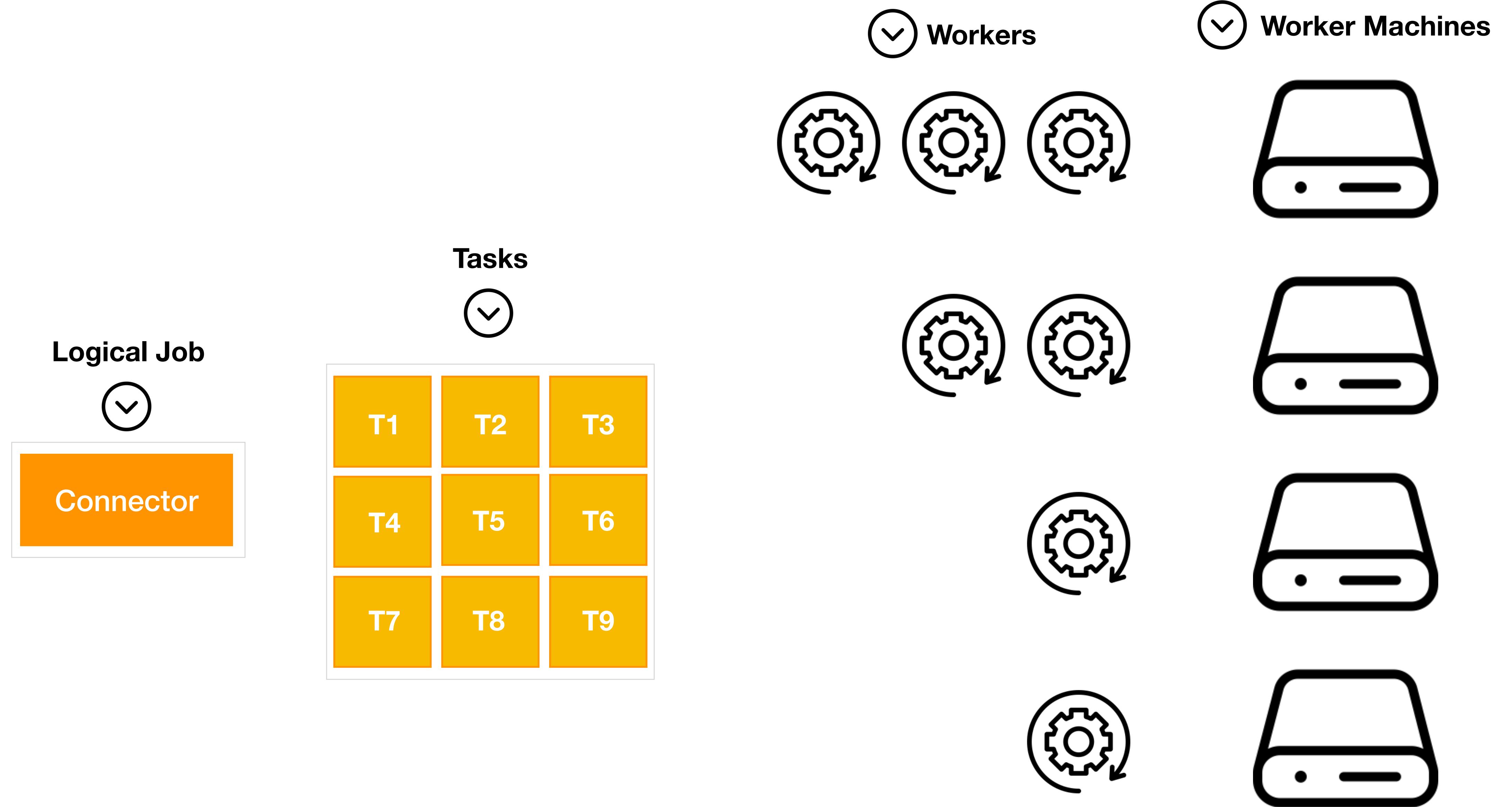
Replicator - GCS - JMS - IBM MQ - Active MQ - Cassandra

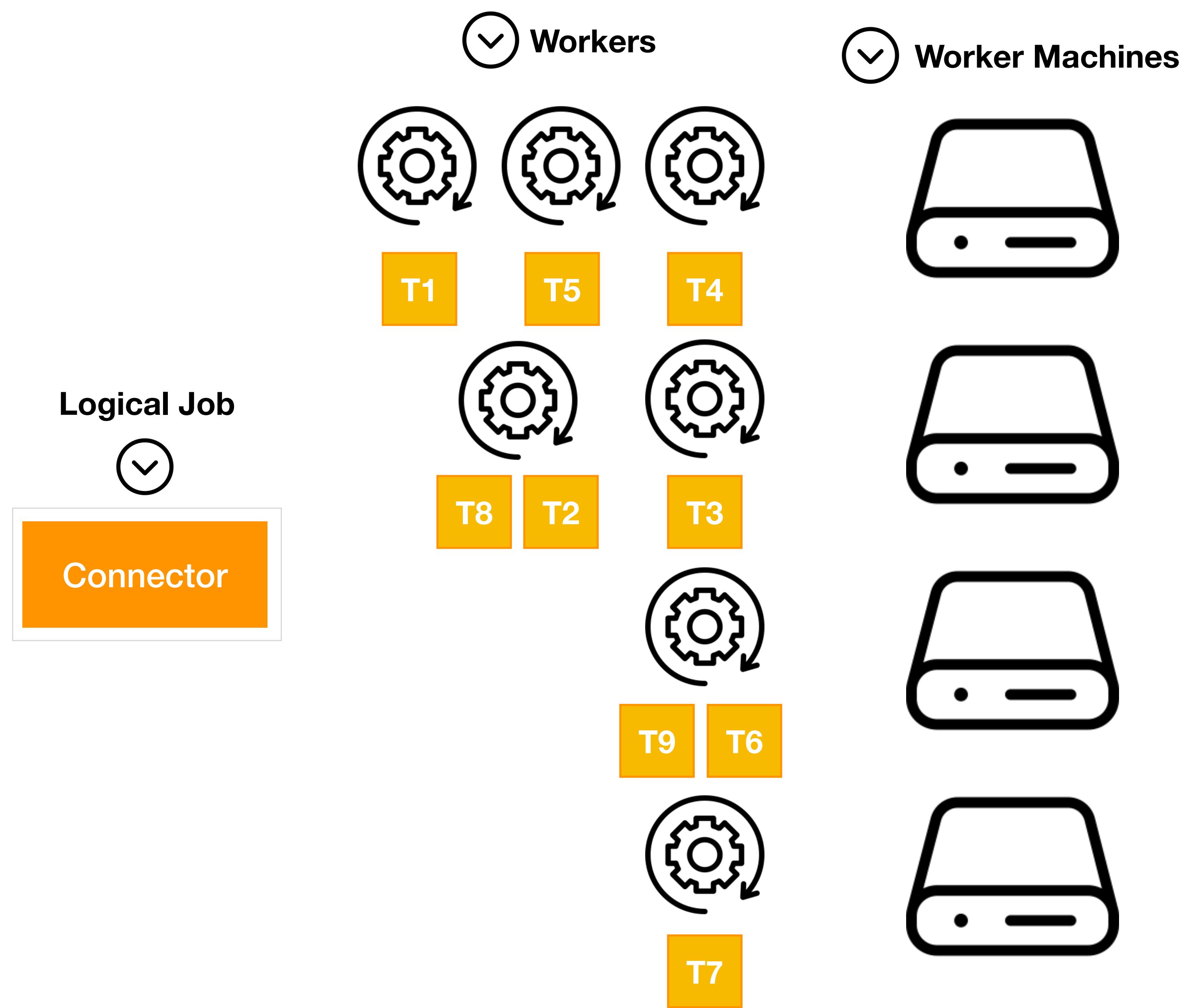
Architecture

Workers >



< **Worker Machines**

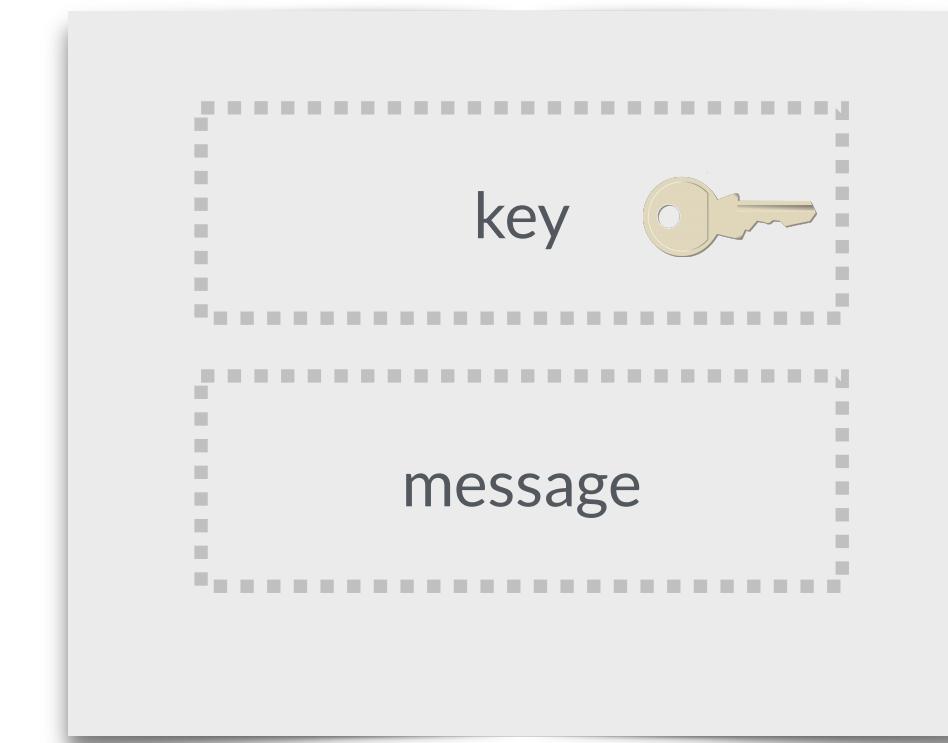




Source Architecture

Kafka Messages

- Message may contain a *key* for better distribution to partitions
- The *key* is also an array of bytes
- If a *key* is provided, a partitioner will hash the key and map it to a single partition
- Therefore it is the only time that something is guaranteed to be in order



A-E



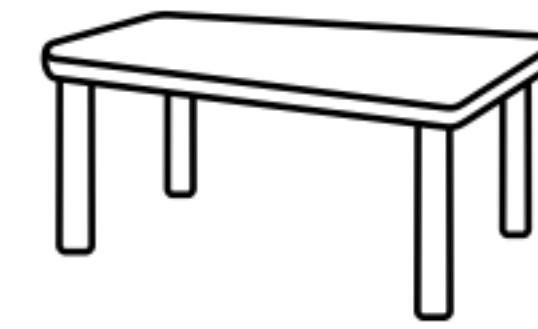
F-K



L-S



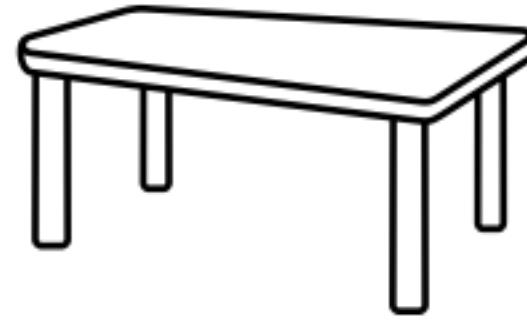
T-Z





murmur2 % partitions

MSFT, AMZN



AAPL, F



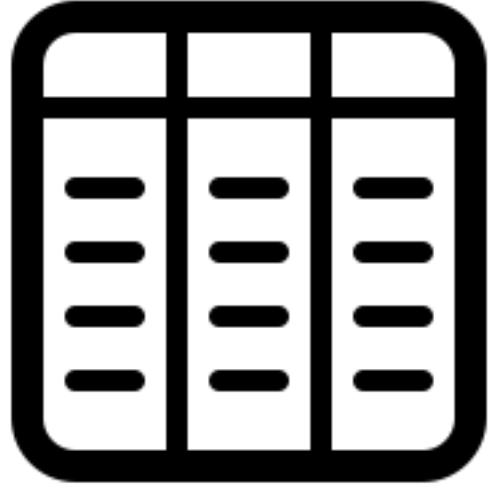
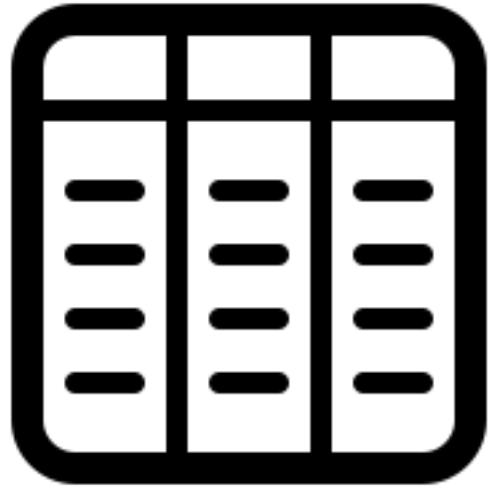
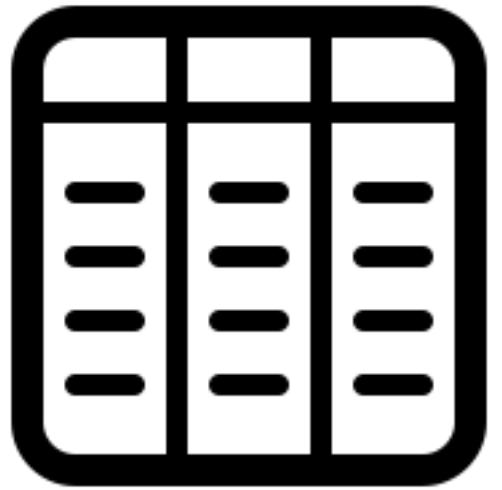
AXP, XOM,T



TWX, NOK, CSCO



Database Table to Kafka Topic



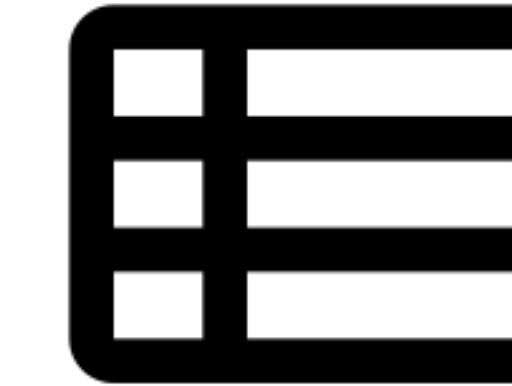
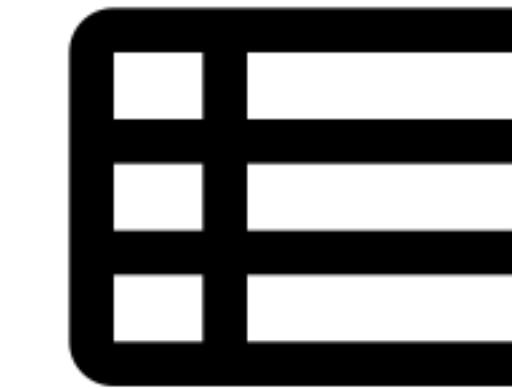
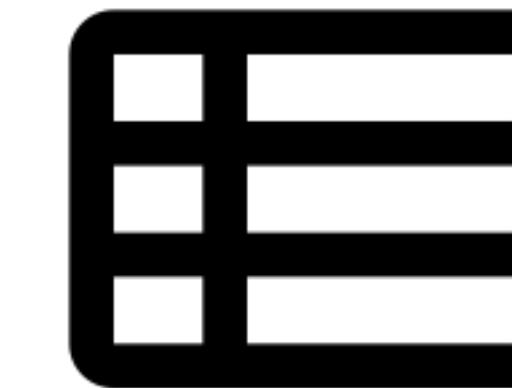
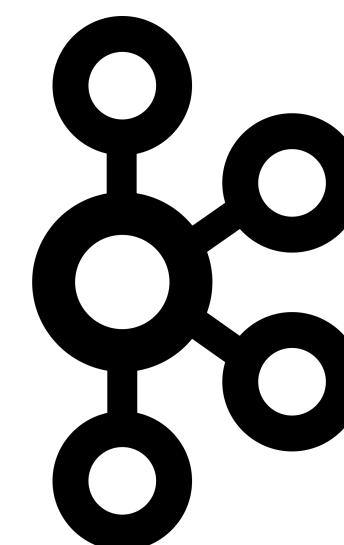
Tasks



T2

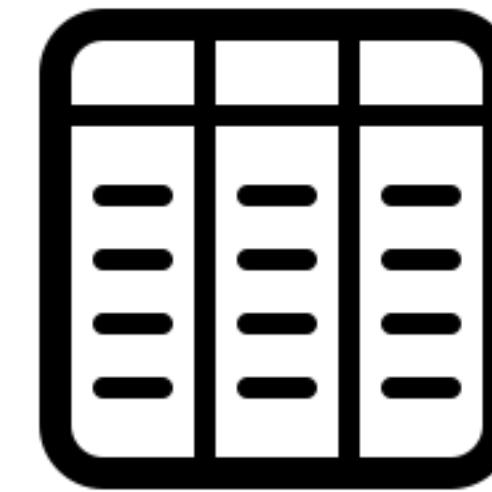
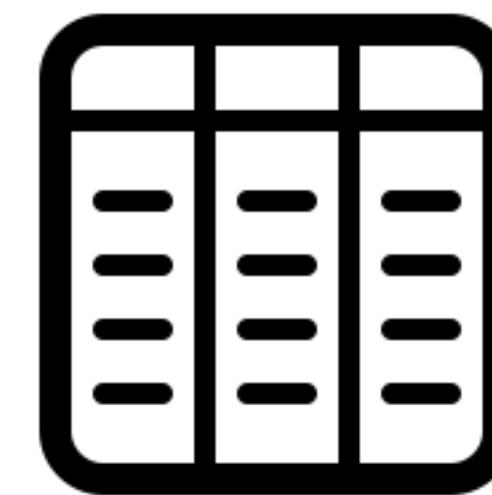
T6

T9



Topics

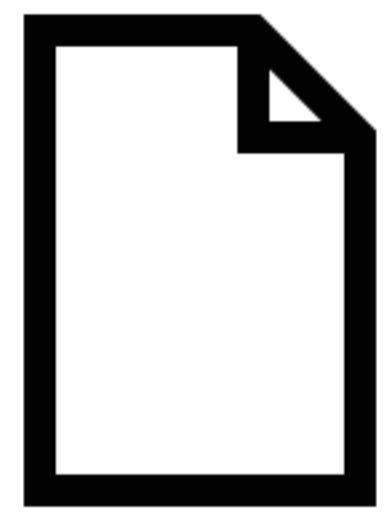
Partitions 



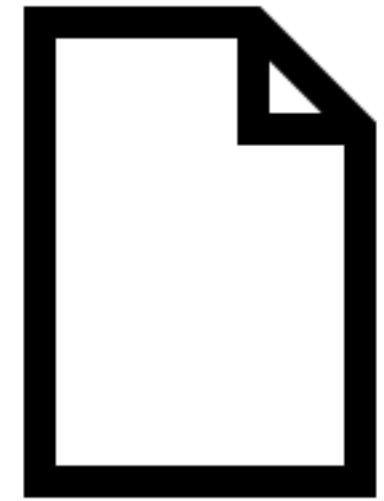
Tasks

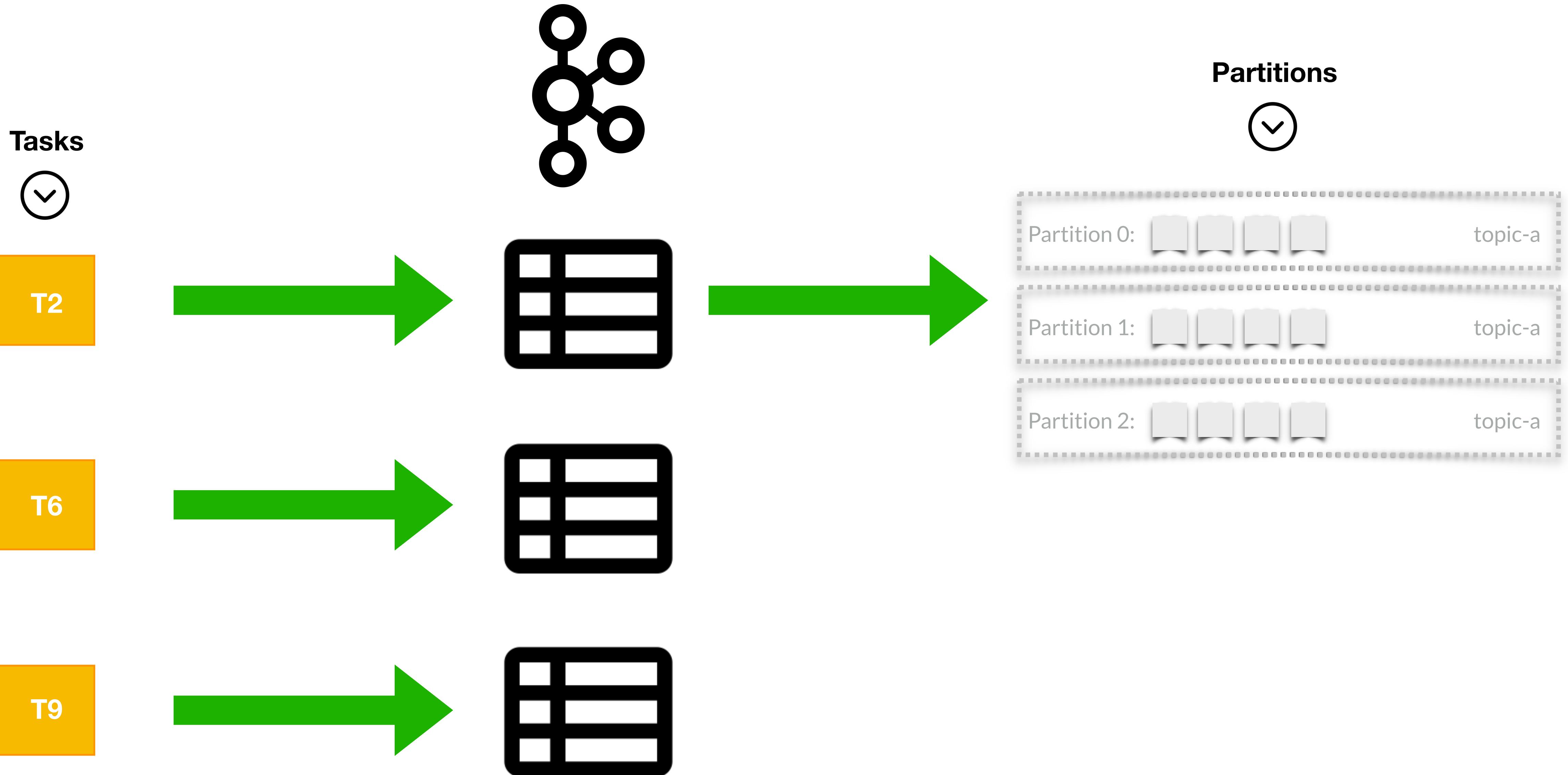


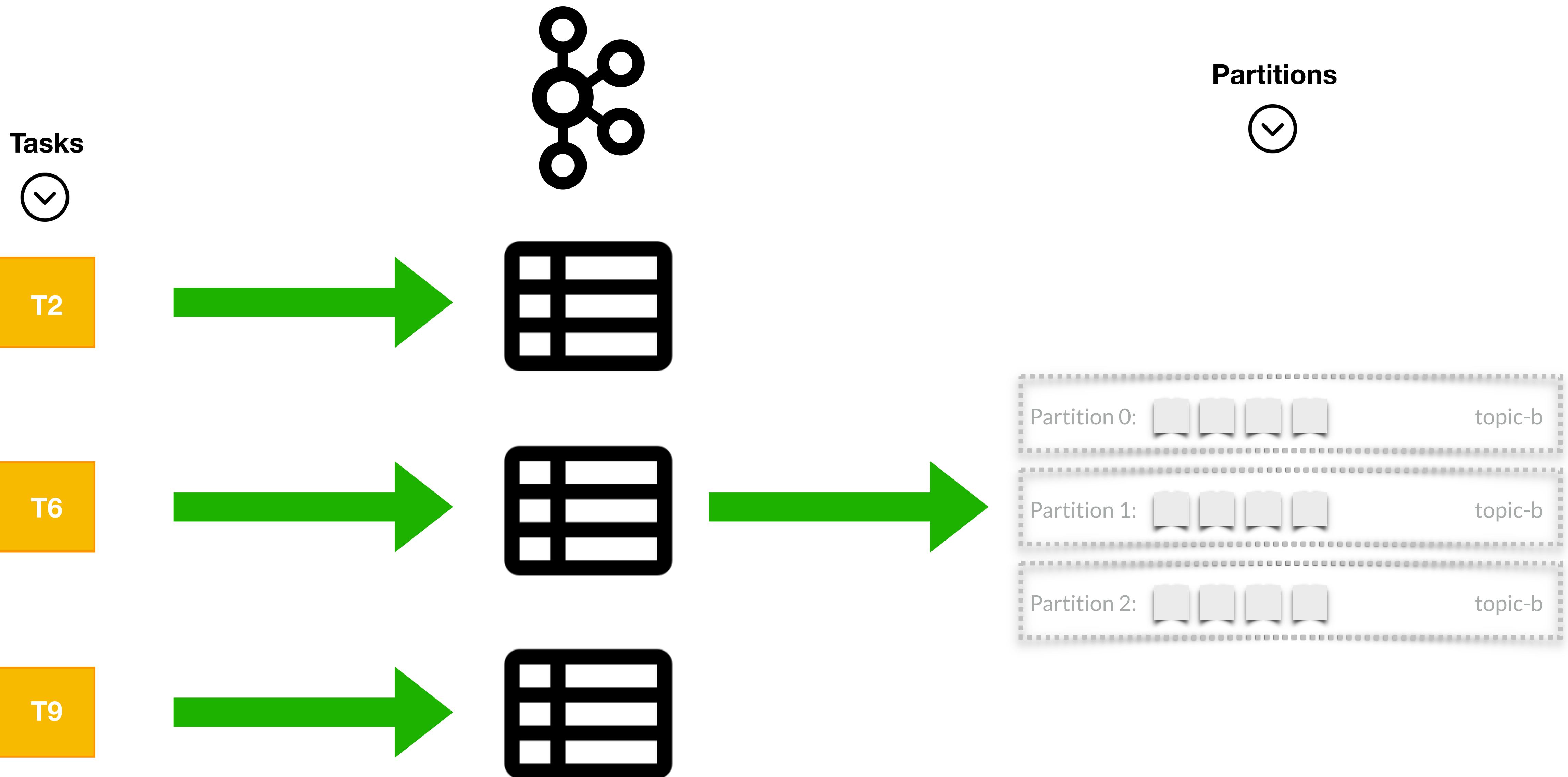
Partitions 

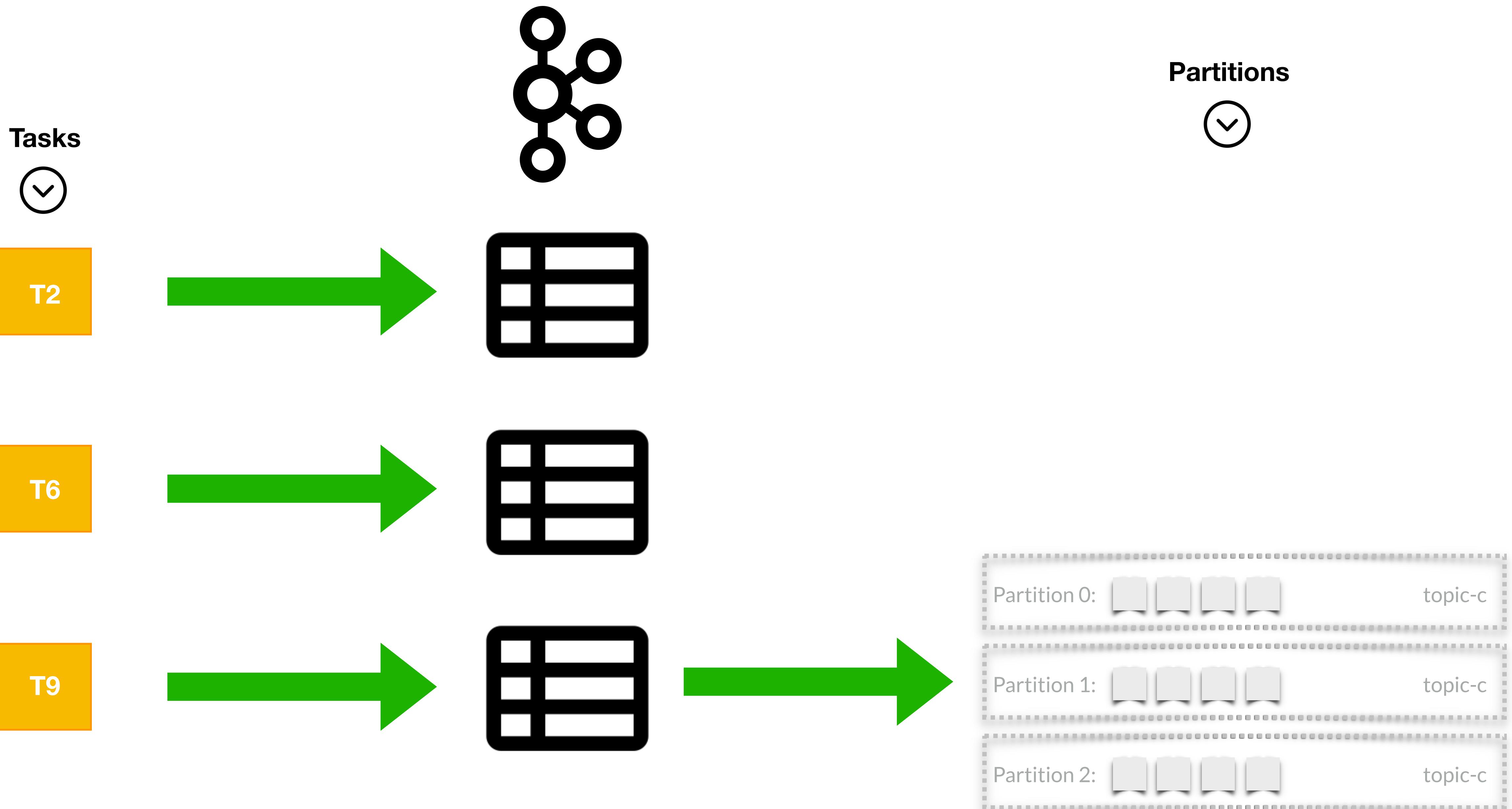


Tasks

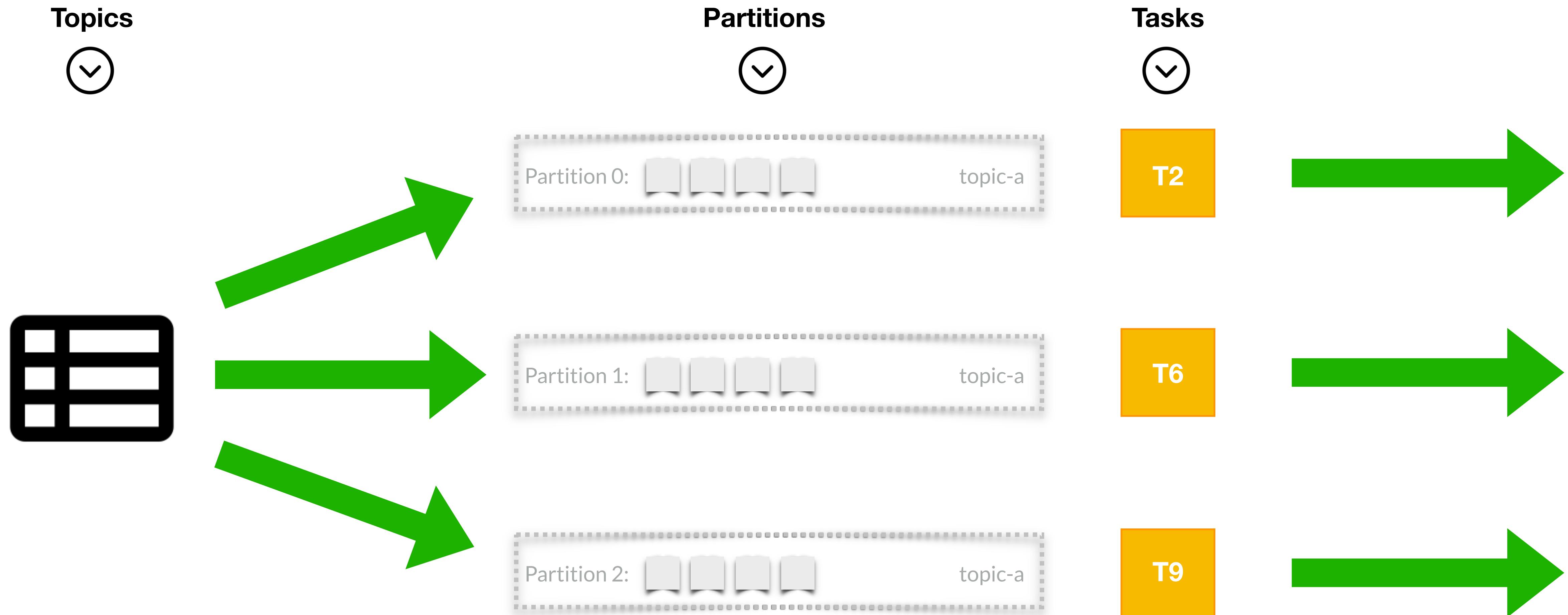


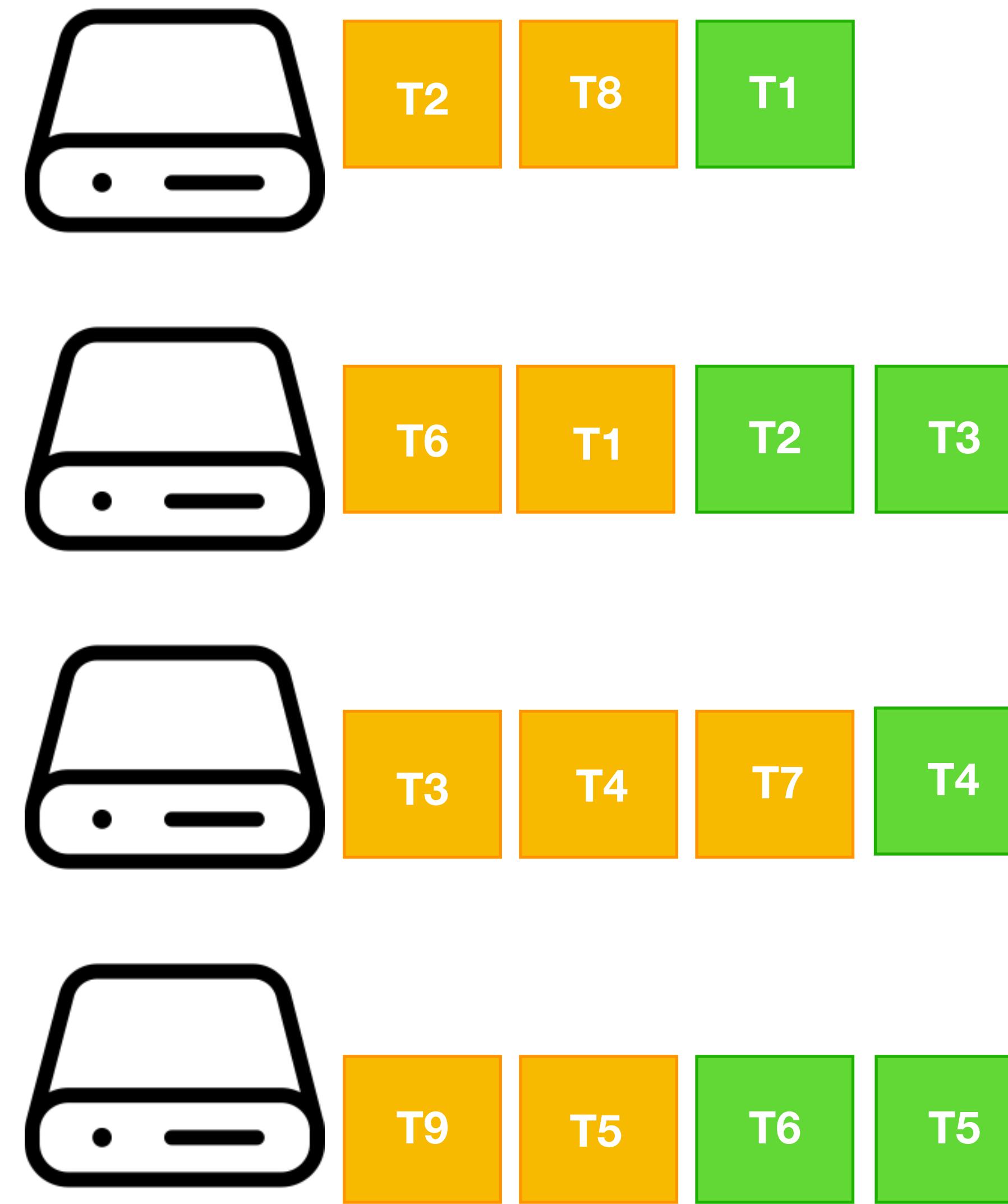






Sink Architecture





Offsets

Offsets

- Connect tracks the location of a file or database and marks its positions after each read
- This allows Connect to start at the right place
- Different Connectors do different things in regards to offsetting

Which Offsets?

- **File Source:** Position in that file
- **Database Source:** id or timestamp

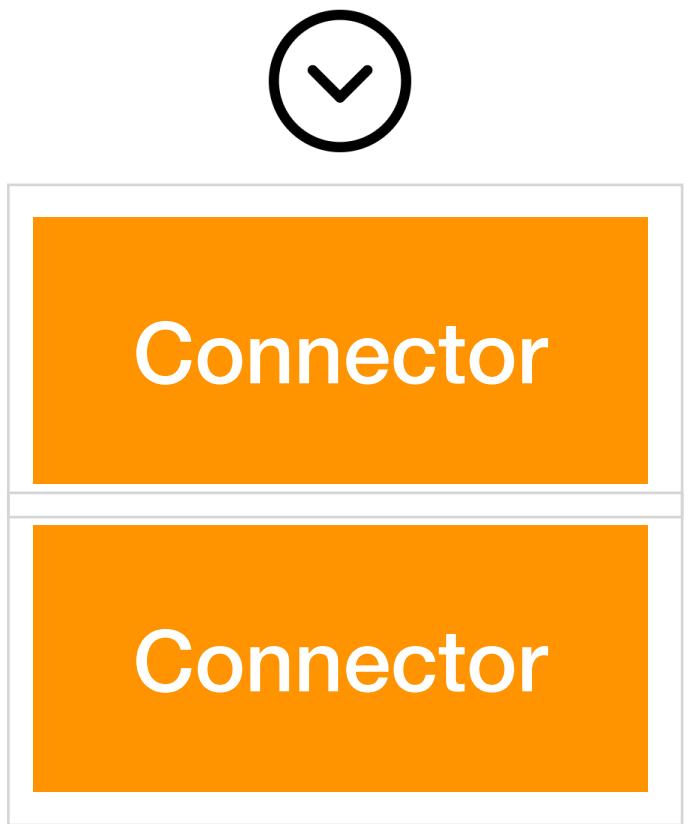
Where do we track the offsets

- **JDBC:** A preconfigured Kafka Topic
- **HDFS Sink:** HDFS File
- **FileStream Source:** A Separate Local File

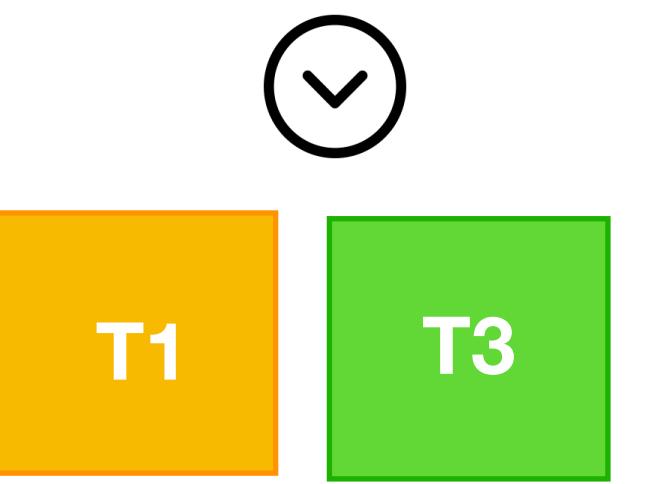
Connect Modes

Standalone Mode

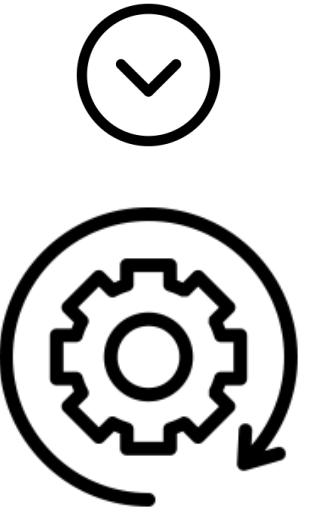
Logical Jobs



Tasks



One Worker



One Machine

Common Configuration for ALL Connectors



```
% connect-standalone connect-standalone.properties \
    connector1.properties [connector2.properties connector3.properties ...]
```

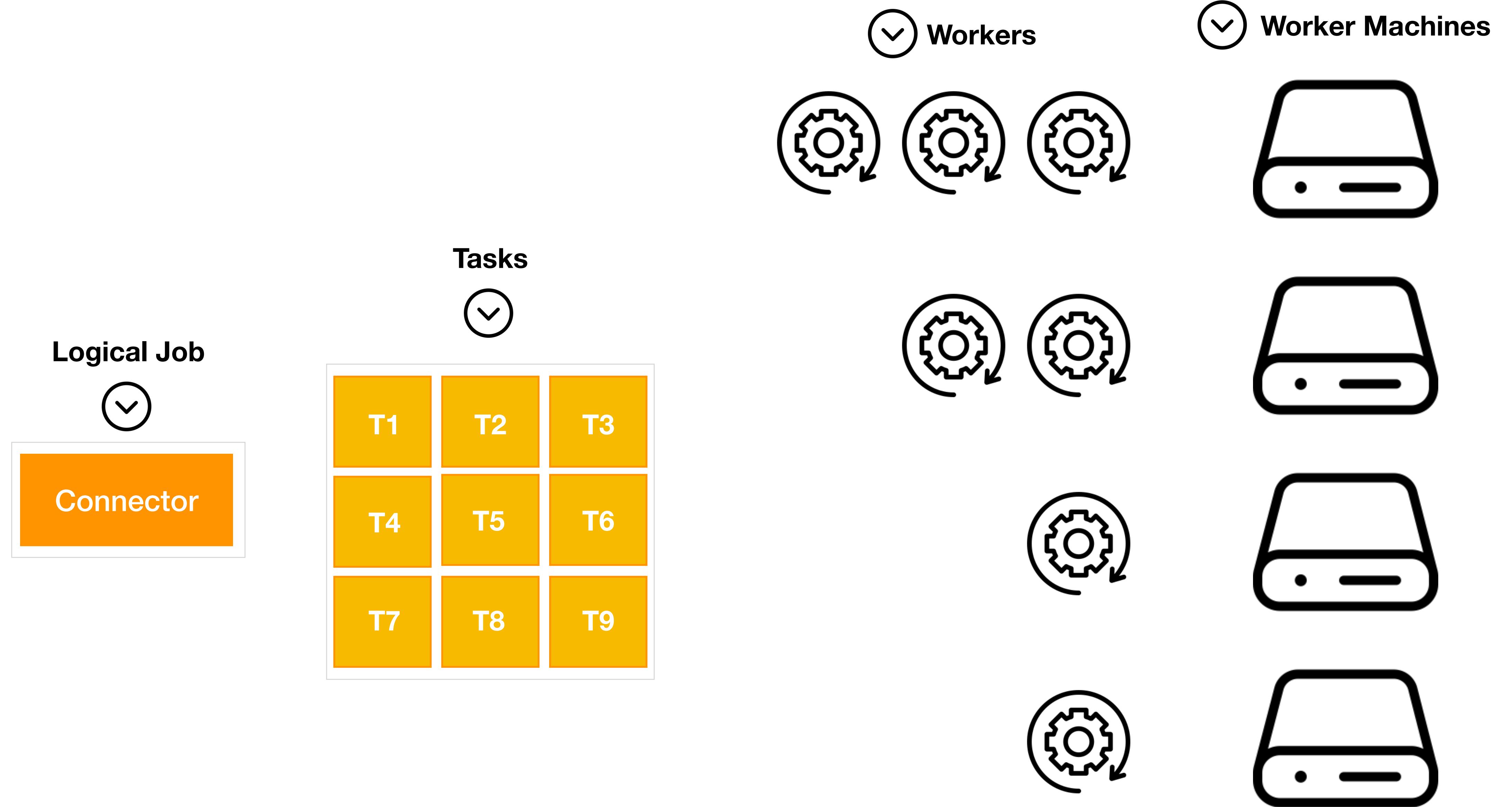


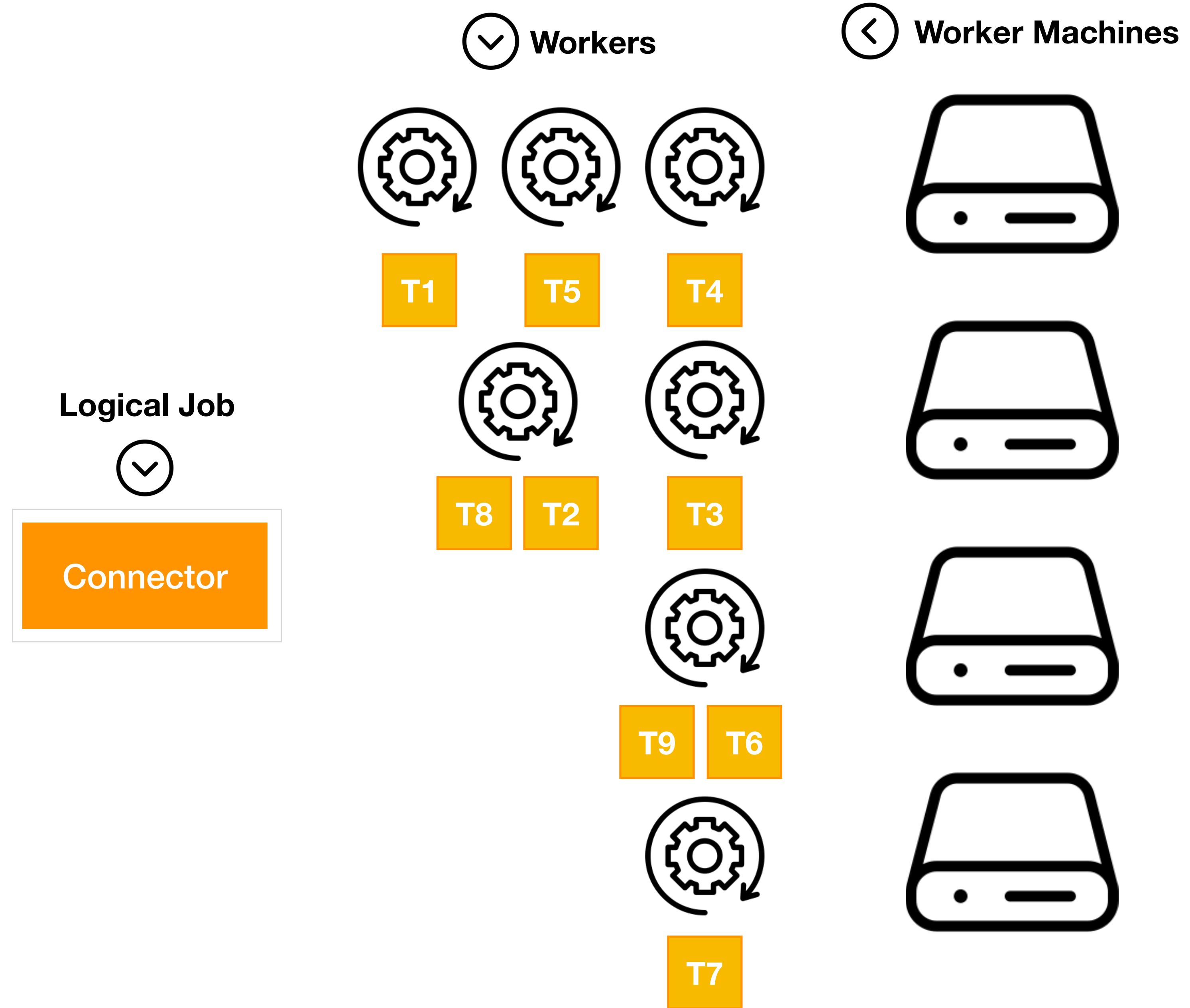
Configuration for Specific Connector



One Machine

Distributed Mode





Common Configuration for ALL Workers



```
% connect-distributed connect-distributed.properties
```

group.id=sales group.id=sales group.id=sales group.id=sales



Worker Machines

What is in a Common Configuration?

Common Configuration

bootstrap.servers	List of host/ports
key.converter	Converter Class for Key
value.converter	Converter Class for Value

Additional Configuration for Standalone

`offset.storage.file.filename`

The filename in which to store offset data for the Connectors
(Default: ""). This enables a standalone process to be stopped
and then resume where it left off.

Additional Configuration for Distributed

<code>group.id</code>	Group that the cluster belongs to
<code>session.timeout.ms</code>	Timeout used to detect failures when using Kafka's group management facilities fails
<code>heartbeat.interval.ms</code>	Expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Must be smaller than <code>session.timeout.ms</code>
<code>config.storage.topic</code>	Topic in which to store Connector & task config. data
<code>offset.storage.topic</code>	Topic in which to store offset data for Connectors
<code>status.storage.topic</code>	Topic in which to store connector & task status

What is in a Specific Configuration?

Specific Connector Configuration

name	Unique Name of Connector
connector.class	Connector class
tasks.max	Number of tasks to run; Connect decides if possible
key.converter	Converter for the key
value.converter	Value for the key
topics	Sink Topics

Single Message Transforms

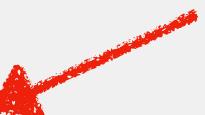
Transformation Chains

- **transforms** - List of aliases for the transformation, specifying the order in which the transformations will be applied.
- **transforms.\$alias.type** - Fully qualified class name for the transformation.
- **transforms.\$alias.\$transformationSpecificConfig** - Configuration properties for the transformation

Alias Declarations

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=test.txt
topic=connect-test
transforms=MakeMap, InsertSource
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=line
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.InsertSource.static.field=data_source
transforms.InsertSource.static.value=test-file-source
```

Alias Declarations



Hoist Field Transformation

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=test.txt
topic=connect-test
transforms=MakeMap, InsertSource
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=line
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.InsertSource.static.field=data_source
transforms.InsertSource.static.value=test-file-source
```

Make Map Configuration



org.apache.kafka.connect.transforms.HoistField

Wrap data using the specified field name in a Struct when schema present, or a Map in the case of schemaless data.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.HoistField$Key`) or value (`org.apache.kafka.connect.transforms.HoistField$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
field	Field name for the single field that will be created in the resulting Struct or Map.	string			medium

"Hello World"

"Kool And The Gang"

"Roger, Roger"

"Everyone Wang Chung Tonight"

"Get off my plane!"

```
{"line":"Hello World"}  
{"line":"Kool And The Gang"}  
{"line":"Roger, Roger"}  
{"line":"Everyone Wang Chung Tonight"}  
{"line":"Get off my plane!"}
```

Insert Source Transformation

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=test.txt
topic=connect-test
transforms=MakeMap, InsertSource
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=line
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.InsertSource.static.field=data_source
transforms.InsertSource.static.value=test-file-source
```

←
Insert Source Configuration

org.apache.kafka.connect.transforms.InsertField

Insert field(s) using attributes from the record metadata or a configured static value.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.InsertField$Key`) or value (`org.apache.kafka.connect.transforms.InsertField$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
static.field	Field name for static data field. Suffix with <code>!</code> to make this a required field, or <code>?</code> to keep it optional (the default).	string	null		medium
static.value	Static field value, if field name configured.	string	null		medium

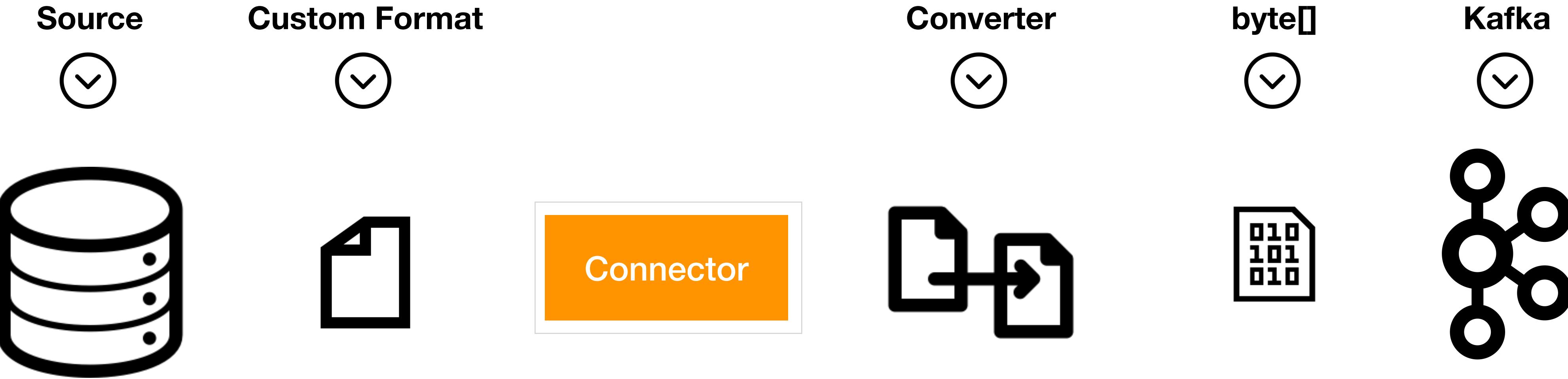
```
{"line":"Hello World"}  
{"line":"Kool And The Gang"}  
{"line":"Roger, Roger"}  
{"line":"Everyone Wang Chung Tonight"}  
{"line":"Get off my plane!"}
```

```
{"line":"Hello World", "data-source":"test-file-source"}  
{"line":"Kool And The Gang", "data-source":"test-file-source"}  
{"line":"Roger, Roger", "data-source":"test-file-source:"}  
{"line":"Everyone Wang Chung Tonight", "data-source":"test-file-source:"}  
{"line":"Get off my plane!", "data-source":"test-file-source:"}
```

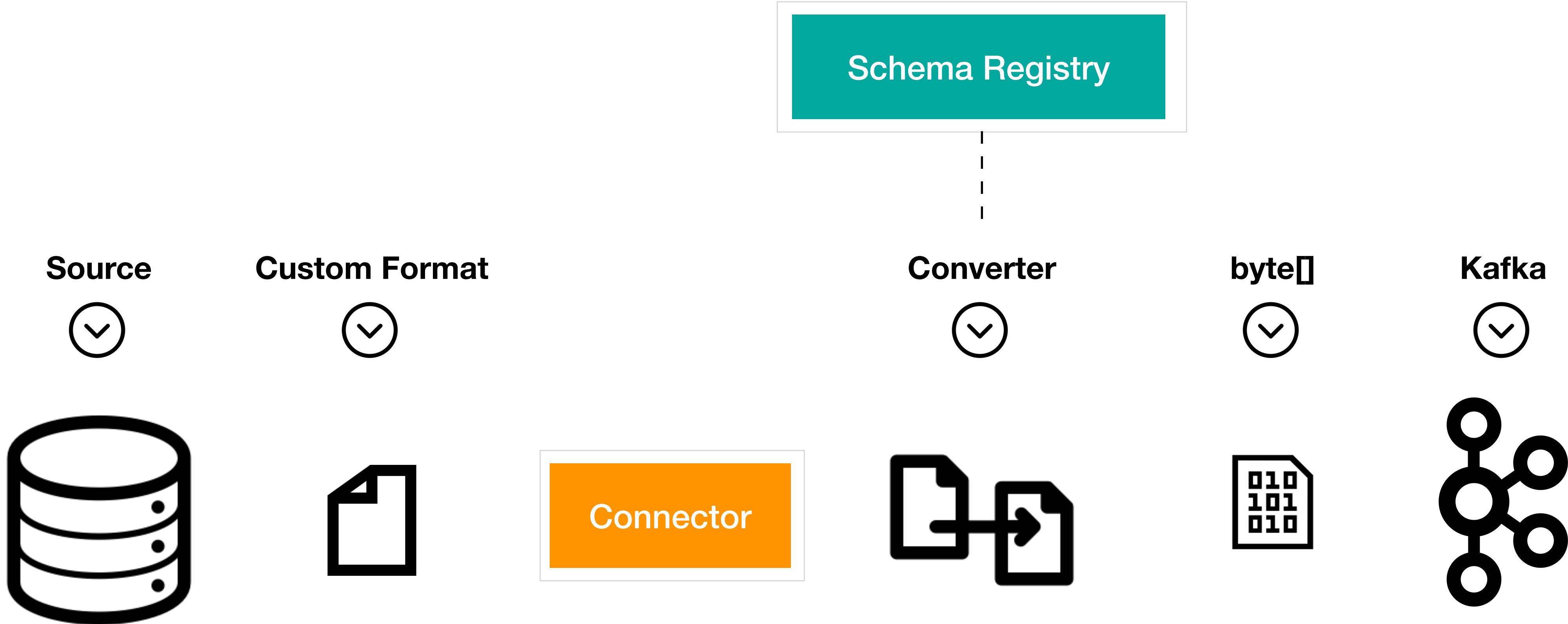
Available Transformations

- **InsertField** - Add a field using either static data or record metadata
- **ReplaceField** - Filter or rename fields
- **MaskField** - Replace field with valid null value for the type (0, empty string, etc)
- **ValueToKey**
- **HoistField** - Wrap the entire event as a single field inside a Struct or a Map
- **ExtractField** - Extract a specific field from Struct and Map and include only this field in results
- **SetSchemaMetadata** - modify the schema name or version
- **TimestampRouter** - Modify the topic of a record based on original topic and timestamp.
Useful when using a sink that needs to write to different tables or indexes based on timestamps
- **RegexRouter** - modify the topic of a record based on original topic, replacement string and a regular expression

Converters



Converters Exist Independently



```
key.converter = org.apache.kafka.connect.storage.StringConverter
```

```
value.converter = org.apache.kafka.connect.storage.StringConverter
```

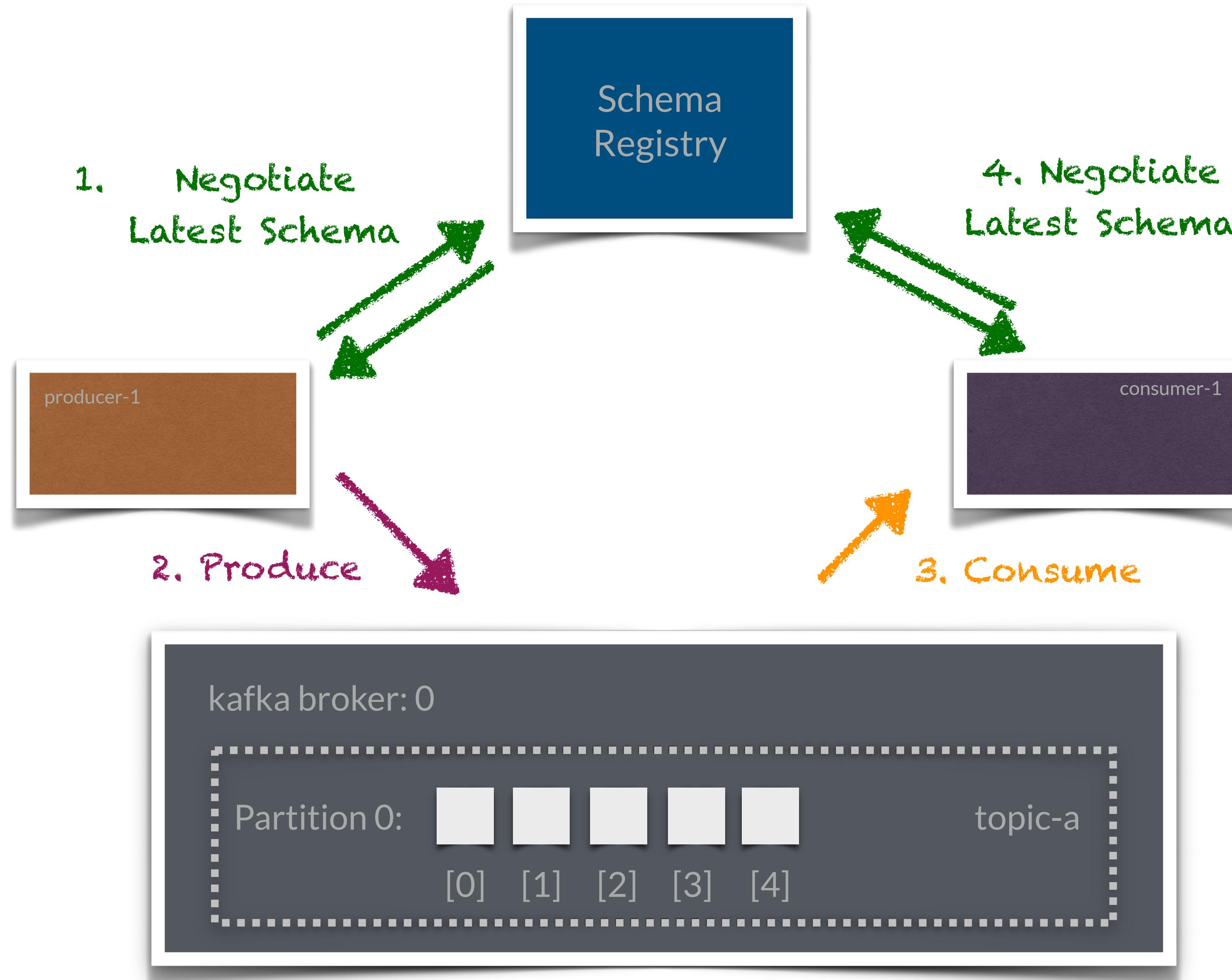
```
key.converter = org.apache.kafka.connect.json.JsonConverter
```

```
value.converter = org.apache.kafka.connect.json.JsonConverter
```

```
key.converter = io.confluent.connect.avro.AvroConverter
```

```
value.converter = io.confluent.connect.avro.AvroConverter
```

Uses Schema Registry



Configuring Connect to use Avro with Schema Registry

```
key.converter=io.confluent.connect.avro.AvroConverter  
key.converter.schema.registry.url=http://schemaregistry1:8081  
value.converter=io.confluent.connect.avro.AvroConverter  
value.converter.schema.registry.url=http://schemaregistry1:8081
```

JDBC Connector

JDBC Source Connector

- Polls a JDBC Connection to Database for any new and updated changes
- Creates a record in Kafka based on the changes
- Table to Topic Relationship
- New and Deleted Tables are propagated automatically

JDBC Connector Properties

connection.url	JDBC connection URL for the database to load.
topic.prefix	Prefix to prepend to table names to generate the name of the Kafka topic
mode	The mode for updating a table each time it is polled (See following slides).
poll.interval.ms	Frequency in ms to poll for new data in each table.
timestamp.delay.interval.ms	How long to wait after a row with certain timestamp appears before we include it in the result

Ensure that your database driver is in the classpath of your connector

JDBC Connector Properties

incrementing.column.name	The name of the strictly incrementing column to use to detect new rows
query	If specified, the query to perform to select new or updated row. Great for joins
table.blacklist	List of tables to exclude from copying
table.whitelist	List of tables to include in copying
timestamp.column.name	The name of the timestamp column to use to detect new or modified rows

JDBC Connector Properties

batch.max.rows	Maximum number of rows to include in a single batch when polling for new data
table.poll.interval.ms	Frequency in ms to poll for new or removed tables
validate.non.null	By default, the JDBC connector will validate that all incrementing and timestamp tables have NOT NULL

JDBC Connector Modes

JDBC Connector Modes

bulk	Perform a bulk load of the entire table each time it is polled
incrementing	Use a strictly incrementing column on each table to detect only new rows. Note that this will not detect modifications or deletions of existing rows
timestamp	Use a timestamp (or timestamp-like) column to detect new and modified rows.
timestamp+incrementing	Use two columns, a timestamp column that detects new and modified rows and a strictly incrementing column which provides a globally unique ID

RESTFul Configuration

REST API Configuration

- Connectors can be deployed via RestFul Configuration rather than file deployment
- For Distributed Mode
 - This is the only way to do so in distributed mode
 - Changes will persist even after restarts
 - REST Calls can be made to any worker
- For Standalone
 - REST Calls can be made to the single
 - Changes **will not persist**

REST Examples

GET /connectors	Get a list of active connectors
POST /connectors	Create a new connector, returning the current connector info if successful.
GET /connectors/(string:name)	Get information about the connector.
GET /connectors/(string:name)/config	Get the configuration for the connector
GET /connectors/(string:name)/status	Get current status of the connector, including whether it is running, failed or paused

Creating an HDFS Sink Example

```
POST /connectors HTTP/1.1
Host: connect.example.com
Content-Type: application/json
Accept: application/json

{
    "name": "hdfs-sink-connector",
    "config": {
        "connector.class": "io.confluent.connect.hdfs.HdfsSinkConnector",
        "tasks.max": "10",
        "topics": "test-topic",
        "hdfs.url": "hdfs://fakehost:9000",
        "hadoop.conf.dir": "/opt/hadoop/conf",
        "hadoop.home": "/opt/hadoop",
        "flush.size": "100",
        "rotate.interval.ms": "1000"
    }
}
```

Lab: Using Connect

KSQLDB

KSQL

Open source streaming SQL engine for Apache Kafka

Provides SQL interface for stream processing on Kafka

Scalable, elastic, fault-tolerant, and real-time.

KSQL

Supports a wide range of streaming operations, including:

- Data filtering
- Transformations
- Aggregations
- Joins
- Windowing
- Sessionization

KSQL Server



KSQL CLI

KSQL Architecture

KSQL Server communicates with the broker

KSQL servers are run separately from the KSQL CLI client and Kafka brokers.

You can deploy servers on remote machines, VMs, or containers

KSQL CLI interacts with KSQL Servers via a REST API

KSQL Server runs in two modes: CLI, Headless

KSQL

Kafka Streams

Producer/Consumer API

Starting in CLI Mode

```
/bin/ksql http://localhost:8088
```

Starting in Headless Mode

```
/bin/ksql-start-server \  
server_path/etc/ksql/ksql-server.properties \  
--queries-file /path/to/queries.sql
```



Tapping a Stream Based Topic

```
CREATE STREAM pageviews \
(viewtime BIGINT, \
userid VARCHAR, \
pageid VARCHAR) \
WITH (KAFKA_TOPIC='pageviews', \
VALUE_FORMAT='DELIMITED');
```

Tapping a Table Based Topic

```
CREATE TABLE users \
  (registertime BIGINT, \
  userid VARCHAR, \
  gender VARCHAR, \
  regionid VARCHAR) \
WITH (KAFKA_TOPIC = 'users', \
      VALUE_FORMAT='JSON', \
      KEY = 'userid');
```

Showing the Streams

SHOW STREAMS;

Stream Name	Kafka Topic	Format
PAGEVIEWS	pageviews	DELIMITED

Showing the Tables

```
SHOW TABLES;
```

Table Name	Kafka Topic	Format
USERS	users	JSON

Supported SQL Types

- BOOLEAN
- INTEGER
- BIGINT
- DOUBLE
- VARCHAR (or STRING)
- ARRAY<ArrayType> (JSON and AVRO only. Index starts from 0)
- MAP<VARCHAR, ValueType> (JSON and AVRO only)
- STRUCT<FieldName FieldType, ... > (JSON and AVRO only)

Various Operators

Scalar Operators:

ABS, ARRAYCONTAINS, CEIL, CONCAT,
EXTRACTJSONFIELD, FLOOR, GEO_DISTANCE, IFNULL, LC
ASE, LEN, MASK, RANDOM, ROUND,
STRINGTOTIMESTAMP, SUBSTRING, TIMESTAMPTOSTRING,
TRIM, UCASE

Aggregator Operators:

COUNT, MAX, MIN, SUM, TOPK, TOPKDISTINCT, WINDOWS
TART, WINDOWEND

Displays Information

SHOW TOPICS	Show all topics
SHOW <TOPIC>	Display Topic
SHOW STREAMS	Show Streams
SHOW TABLES	Show Tables
SHOW FUNCTIONS	Show Available Functions
SHOW QUERIES	Show Persistent Queries

Describing

DESCRIBE

List columns in stream or table

DESCRIBE EXTENDED

Describe in detail stream or table; runtime statistics, and queries that populate the stream or table

Persistent Query

```
CREATE TABLE users_female AS \  
SELECT userid, gender, regionid FROM users \  
WHERE gender='FEMALE';
```

What makes this a persistent query is the form

```
CREATE (TABLE|STREAM) AS SELECT
```

Once executed, it will continuously run, until terminated
with TERMINATE command

Non-Persistent Query

```
SELECT * FROM pageviews  
WHERE ROWTIME >= 1510923225000  
AND ROWTIME <= 1510923228000;
```

A **LIMIT** can be used to limit the number of rows returned.
Once the limit is reached the query will terminate.

Persistent v. Non Persistent

	Persistent	Non Persistent
Where does data go?	Will store into a topic	Will display on screen
How do I stop it?	Find query id using SHOW QUERIES and TERMINATE <id>	CTRL + C
How to create?	CREATE STREAM AS SELECT ...	SELECT

Selecting Customized Fields

```
CREATE STREAM pageviews_transformed \
  WITH (TIMESTAMP='viewtime', \
        PARTITIONS=5, \
        VALUE_FORMAT='JSON') AS \
  SELECT viewtime, \
         userid, \
         pageid, \
         TIMESTAMPTOSTRING \
           (viewtime, 'yyyy-MM-dd HH:mm:ss.SSS') \
             AS timestamp \
  FROM pageviews \
  PARTITION BY userid;
```

Start from the Beginning

```
SET 'auto.offset.reset' = 'earliest';
```

Tumbling Window

```
SELECT item_id, SUM(quantity)  
FROM orders  
WINDOW TUMBLING (SIZE 20 SECONDS)  
GROUP BY item_id;
```

Hopping Window

```
SELECT item_id, SUM(quantity)
FROM orders
WINDOW HOPPING (SIZE 20 SECONDS,
                  ADVANCE BY 5 SECONDS)
GROUP BY item_id;
```

Session Window

```
SELECT item_id, SUM(quantity)  
FROM orders  
WINDOW SESSION (20 SECONDS)  
GROUP BY item_id;
```

Lab: Using KSQLDB

Please Fill in Survey



Conclusion

Agenda

- ★ Understand Kafka
- ★ Understanding K8s Operators
- ★ Understanding Strimzi
- ★ Understand Producer
- ★ Understand Consumer
- ★ Understand Rebalancing
- ★ Understand Replication
- ★ Understand Properties
- ★ Understand Stream v Tables
- ★ Understanding Kafka Connect
- ★ Understanding Monitoring
- ★ Understanding Security
- ★ Performance Tuning
- ★ Understanding KSQLDB