Kafka UmeJUG

git clone github.com/ryzlab/kafka-kompetensdag

Kafka® is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast...

https://kafka.apache.org

Motivation

Motivation

Traditional batch Processing

- Daily, weekly, monthly
- “I can’t start to analyze today’s data until the overnight ingest process has run”

Real-Time Processing often a better approach

- Real-Time event processing allows for real-time decisions
- Can be used for Fraud Detection, Log monitoring, Fault diagnostics etc

We still have many systems still relying on Batch Processing

- This is changing over time as more ‘stream processing’ systems emerge
  - Kafka Streams, Apache Spark Streaming, Apache Storm, Apache Samza ...

History

Kafka was designed to solve the problems

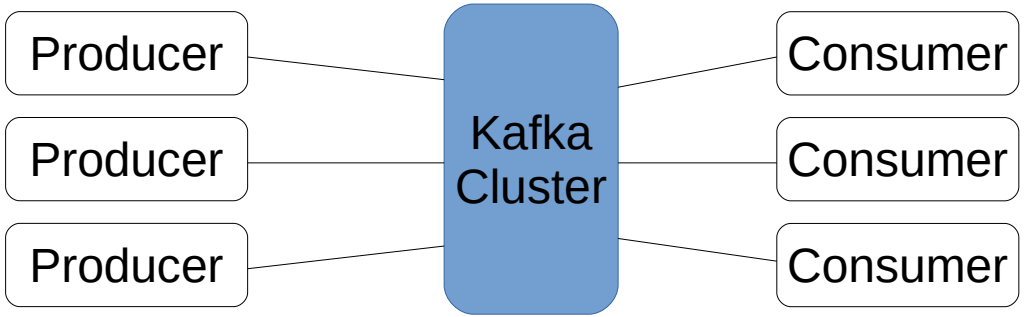
- Simplify data pipelines
- Handling streaming data

Originally created at LinkedIn 2010

- Designed to support batch and real-time analytics
- Kafka is now at the core of LinkedIn’s architecture
- Performs extremely well at very large scale
  - Produces over 2 trillion messages per day
- An open source, top-level Apache project since 2012

A Universal Pipeline

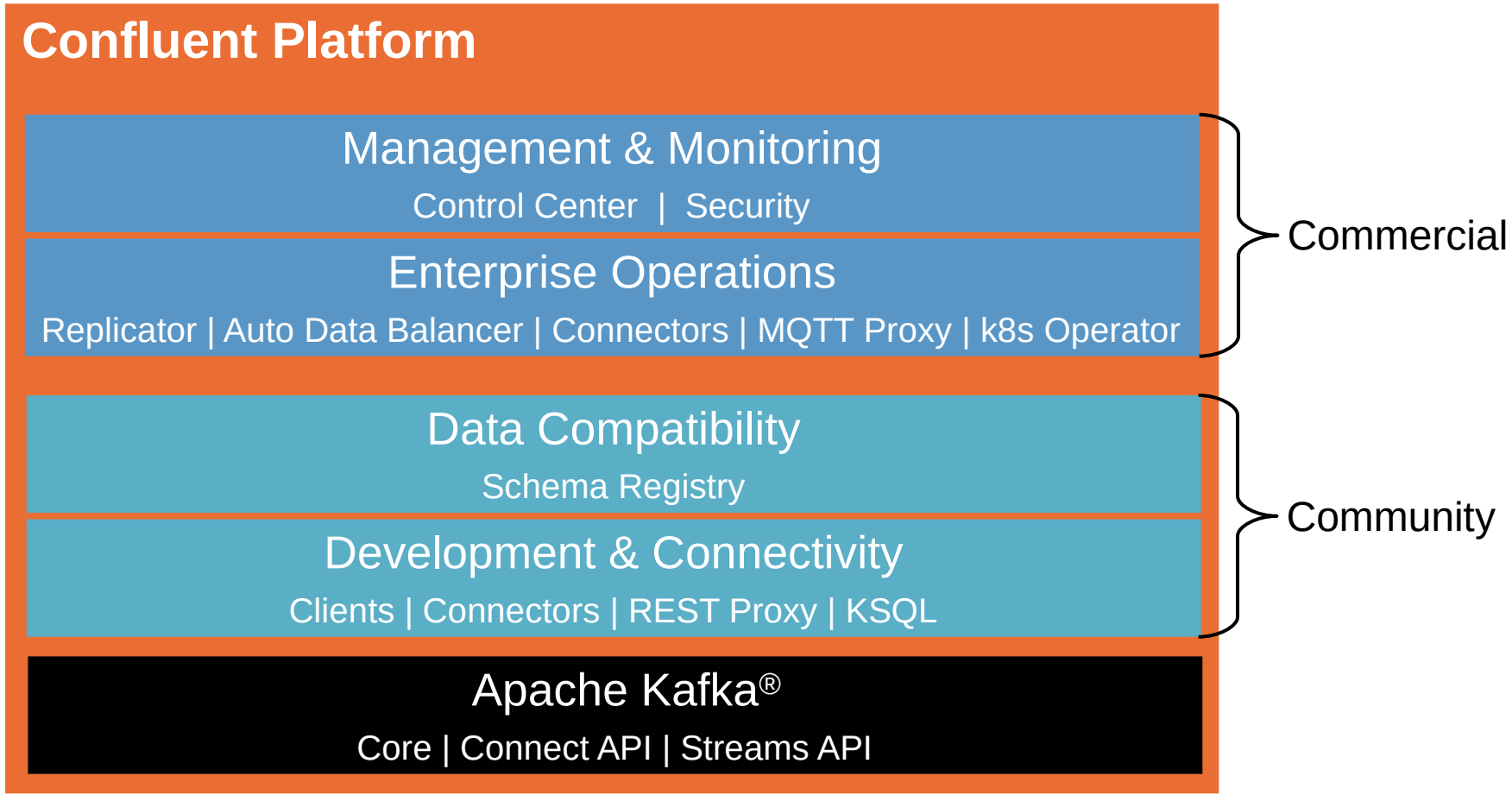
- Kafka decouples data source and destination systems
  - Via a publish/subscribe architecture
- All data sources write their data to the Kafka cluster
- All systems wishing to use the data read from Kafka
- Streaming platform
  - Capture streams of events, facts, records
  - Continuous, real-time data processing and transformation



About Confluent

- Founded in 2014 by the founders of Kafka
  - Include many committers to the Apache Kafka project
- Provides support, consulting and training for Kafka and its ecosystem
- Develops Confluent Open Source
  - Kafka with additional components
  - Completely free, open source

Confluent Platform



Java Code

Producer Example

```
Properties conf = new Properties();
conf.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, “kafka-1:9092,kafka-2:9092”);
// Used to trace activity in Kafka logs
conf.put(ProducerConfig.CLIENT_ID_CONFIG, “producer-client”);
// Messages are key-value and we want to use strings
conf.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
conf.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

KafkaProducer<String, String> producer = new KafkaProducer<>(conf);
ProducerRecord<String, String> record = new ProducerRecord<>(“example-stream”, "thekey", “the message”);
producer.send(record);
producer.flush();
producer.close();
```

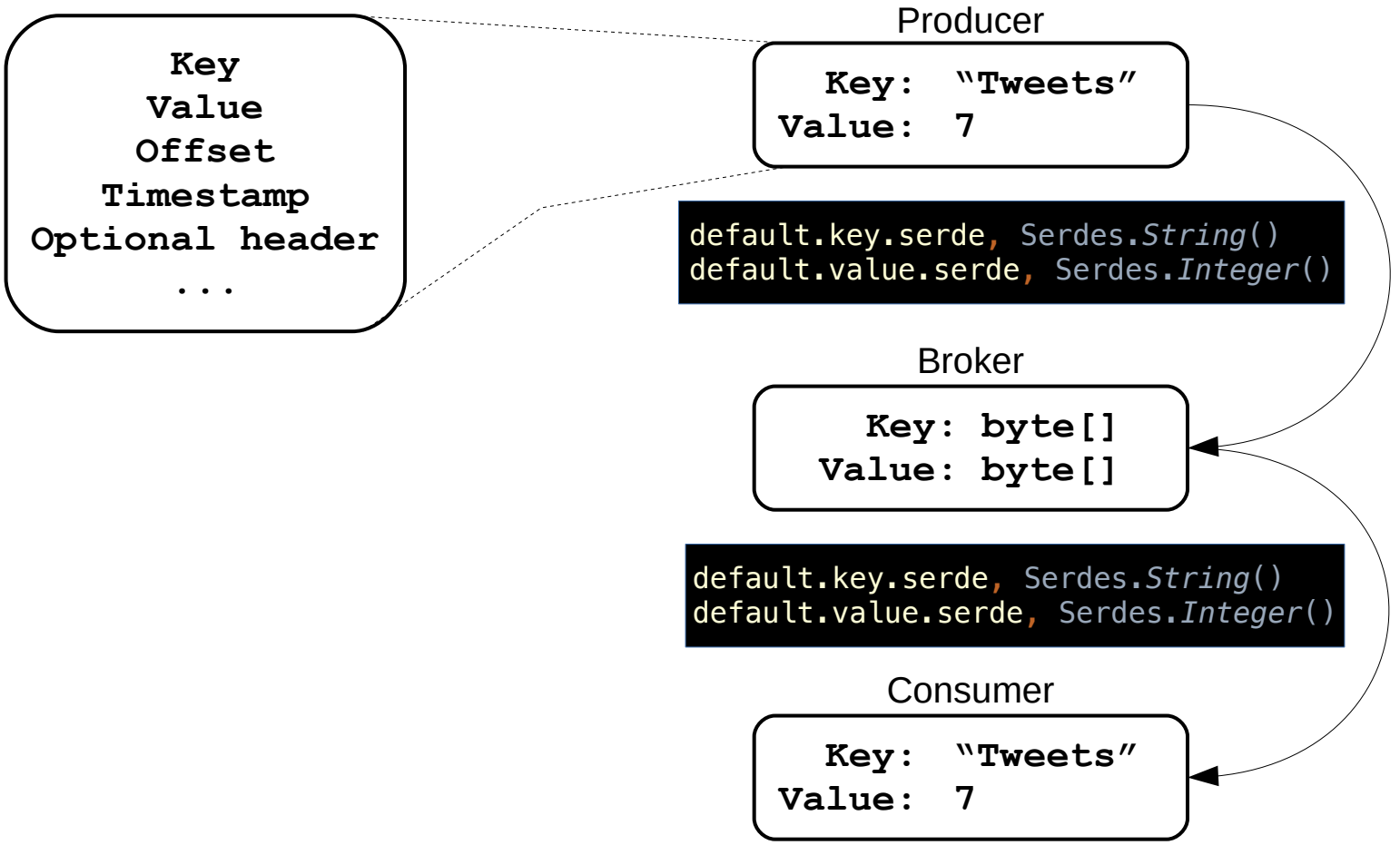
Stream Consumer Example

```
final Properties conf = new Properties();
// Where to find Kafka broker(s).
conf.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-1:9092,kafka-2:9092");
// Give the Streams application a unique name.  The name must be unique in the Kafka cluster
conf.put(StreamsConfig.APPLICATION_ID_CONFIG, "stream-app-v1.0.0");
// Client ID: An optional identifier of a Kafka consumer (in a consumer group) that is passed to a Kafka broker with every request.
conf.put(StreamsConfig.CLIENT_ID_CONFIG, "my-stream-client");
// Specify (de)serializers for record keys values.
conf.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
conf.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
// In the subsequent lines we define the processing topology of the Streams application.
final StreamsBuilder builder = new StreamsBuilder();
final KStream<String, String> messageStream = builder.stream("example-stream");
messageStream.foreach((key, value) -> System.out.println("Key: " + key + ", value: " + value));

final KafkaStreams streams = new KafkaStreams(builder.build(), conf);
streams.start();
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
for (; ; ) {
    Thread.sleep(1000);
}
```

Messages

Messages

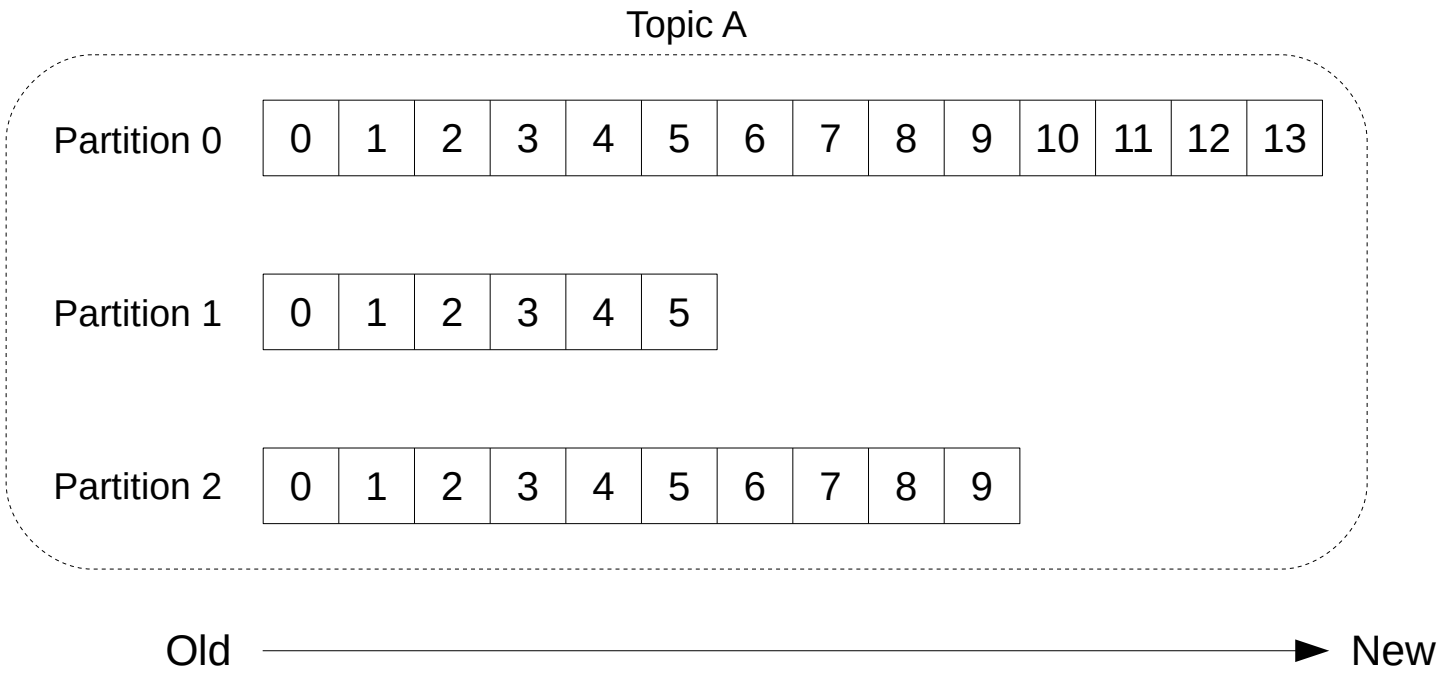

Key
Value
Offset
Timestamp
Optional header
...
Producer
Key: "Tweets"
Value: 7
default.key.serde, Serdes.String()
default.value.serde, Serdes.Integer()
Broker
Key: byte[]
Value: byte[]
default.key.serde, Serdes.String()
default.value.serde, Serdes.Integer()
Consumer
Key: "Tweets"
Value: 7

Timestamp Assignment

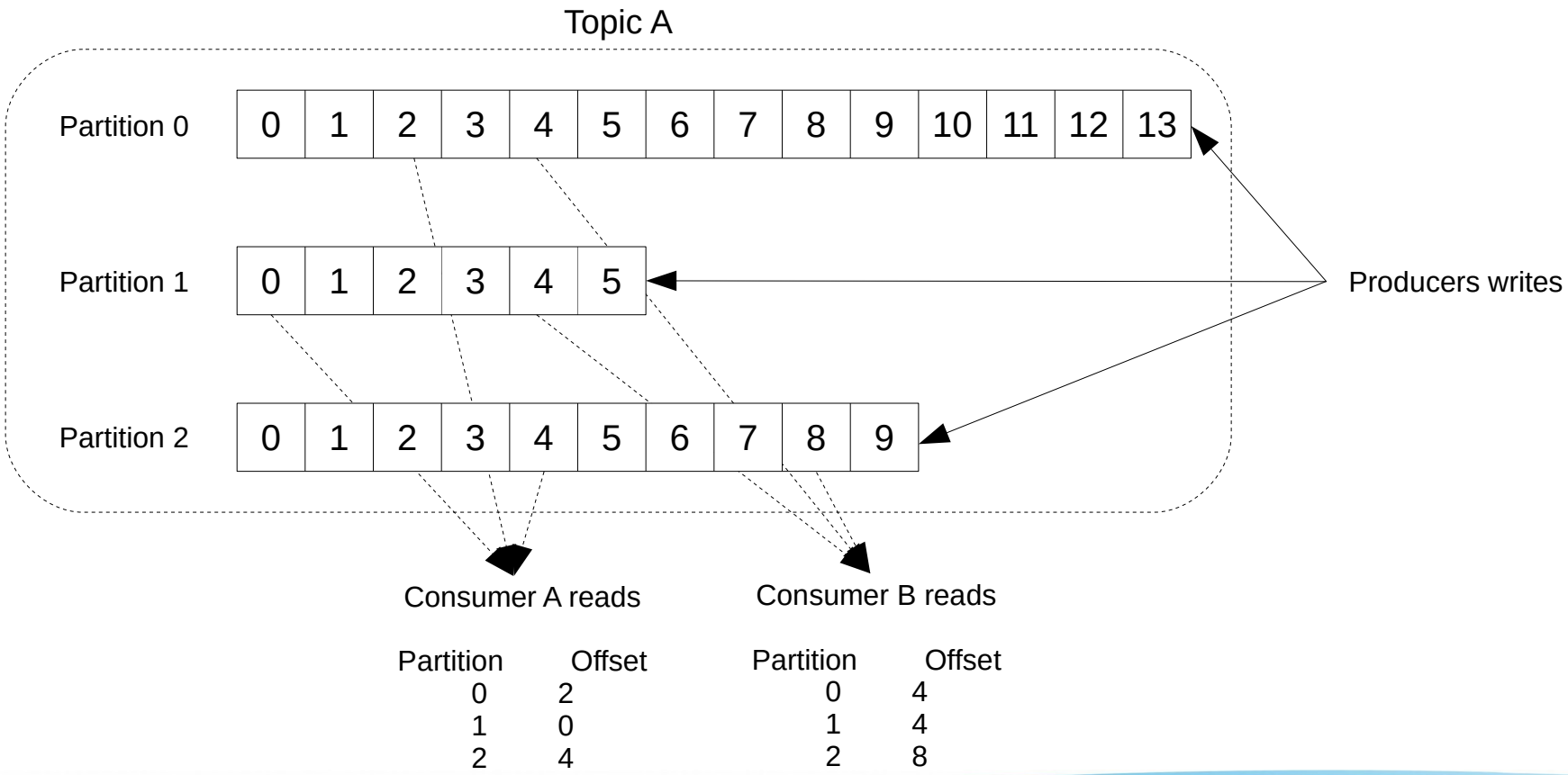
- Can be set in Producer or Broker
- CreateTime (considered to be Event-time), Default
  - When record is created, it does not contain any timestamp by default
  - The Producer can set the timestamp explicitly
  - If not set when the producer calls send(), the current wall-clock time is set automatically
- LogAppendTime
- Timestamp is set by the broker when it receives the message

Topics

Kafka Topic

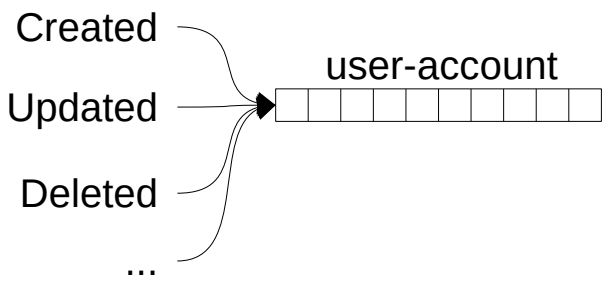

Topic A
Partition 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13
Partition 1
0 1 2 3 4 5
Partition 2
0 1 2 3 4 5 6 7 8 9
Old
New

Kafka Topic


Topic A
Partition 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13
Partition 1
0 1 2 3 4 5
Partition 2
0 1 2 3 4 5 6 7 8 9
Producers writes
Consumer A reads
Partition Offset
0 2
1 0
2 4
Consumer B reads
Partition Offset
0 4
1 4
2 8

What is a Stream

- Unbounded Continuously updating data set
- Stream of Records are
  - Ordered
  - Persistent
  - Replayable
  - Immutable
  - Facts
  - Can be user clicks on web-site, ATM transactions, Application logs ...
- Kafka provides
  - Simple API
  - Exactly once semantics
  - Processor topology



KStream

KStream

Alice	Gunnar	Gunnar	Alice	Gunnar	Alice
Vännäs	Burträsk	Piteå	Kiruna	Luleå	Kalix

KStream KTable

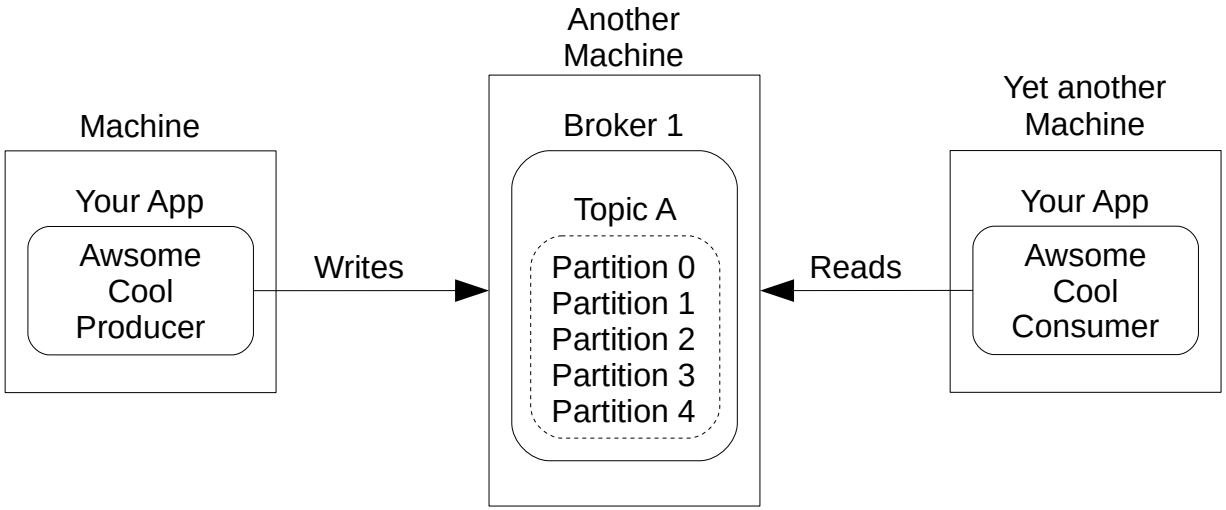
KStream						
	Alice	Gunnar	Gunnar	Alice	Gunnar	Alice
	Vännäs	Burträsk	Piteå	Kiruna	Luleå	Kalix

KTable						
	Alice	Alice	Alice	Alice	Alice	Alice
	Vännäs	Vännäs	Vännäs	Kiruna	Kiruna	Kalix
		Gunnar	Gunnar	Gunnar	Gunnar	Gunnar
		Burträsk	Piteå	Piteå	Luleå	Luleå

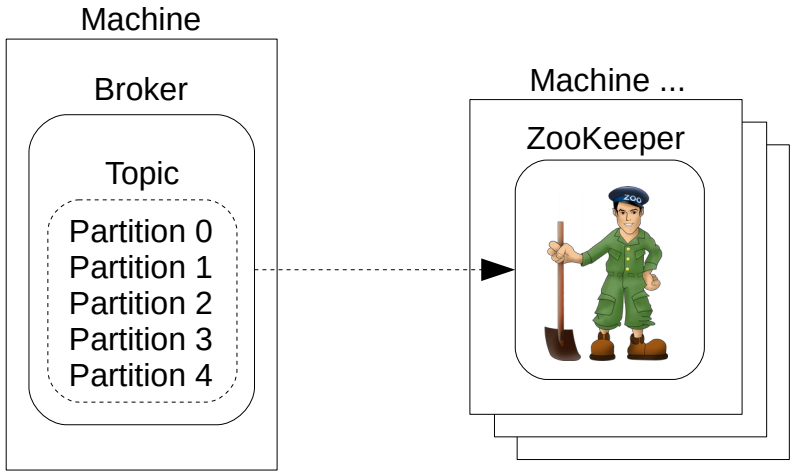
KTable						
	Alice	Alice	Alice	Alice	Alice	Alice
	1	1	1	2	2	3
		Gunnar	Gunnar	Gunnar	Gunnar	Gunnar
		1	2	2	3	3

Broker

Broker

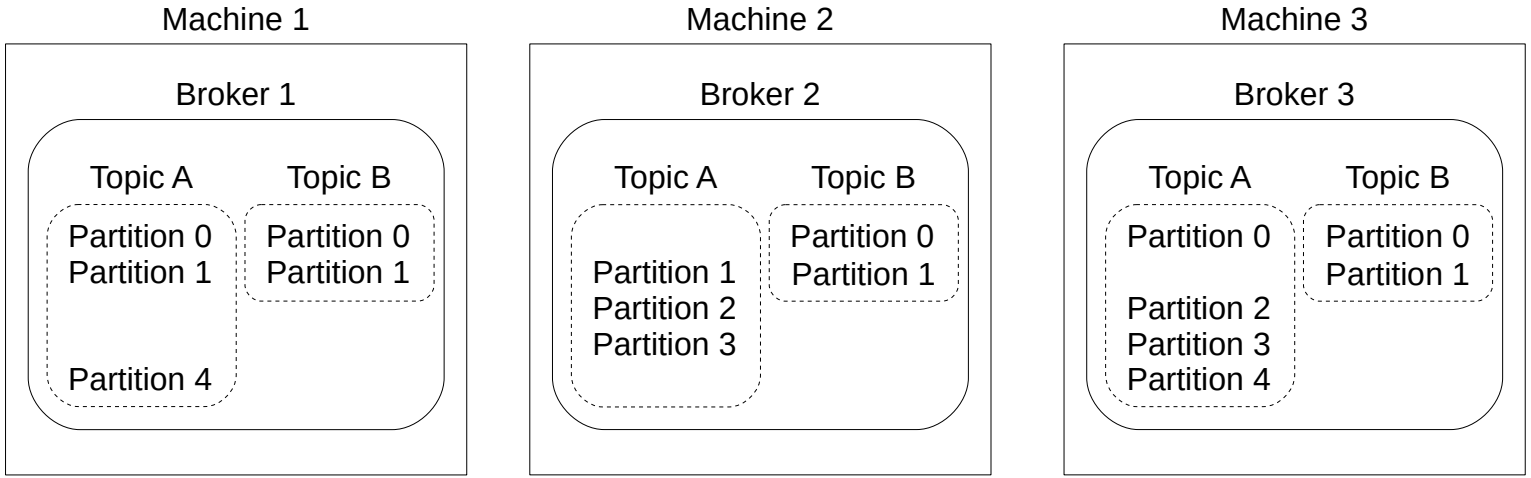

Another
Machine
Broker 1
Topic A
Partition 0
Partition 1
Partition 2
Partition 3
Partition 4
Machine
Your App
Awsome
Cool
Producer
Writes
Reads
Yet another
Machine
Your App
Awsome
Cool
Consumer

Apache ZooKeeper



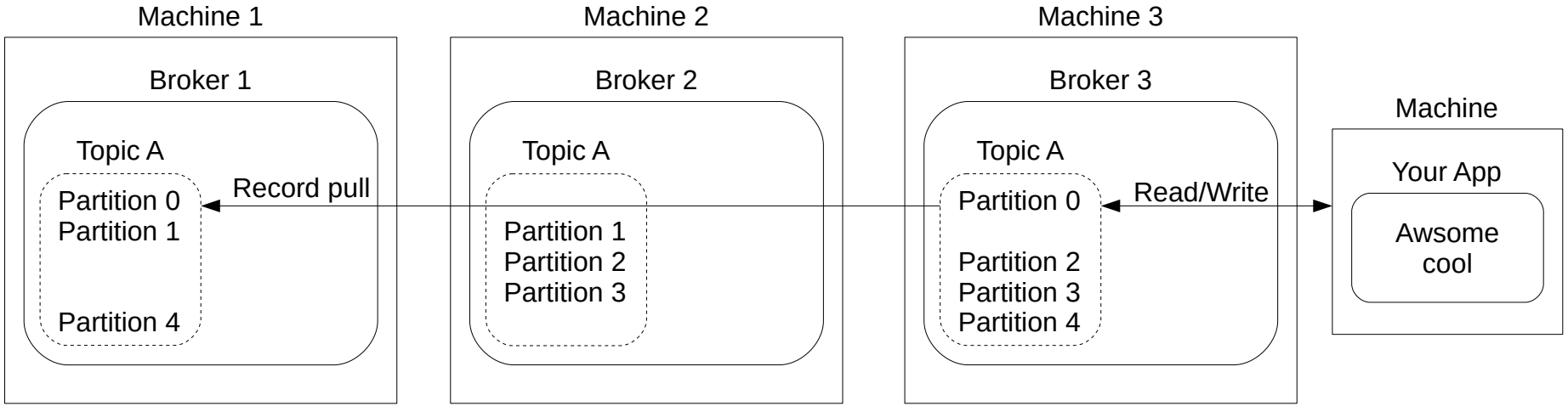
Topic Partition Replica Assignment

Topic A: Partition Count: **5**, Replication Factor: **2**
Topic B: Partition Count: **2**, Replication Factor: **3**



Preferred, Leader and ISR

Topic A: Partition Count: **5**, Replication Factor: **2**



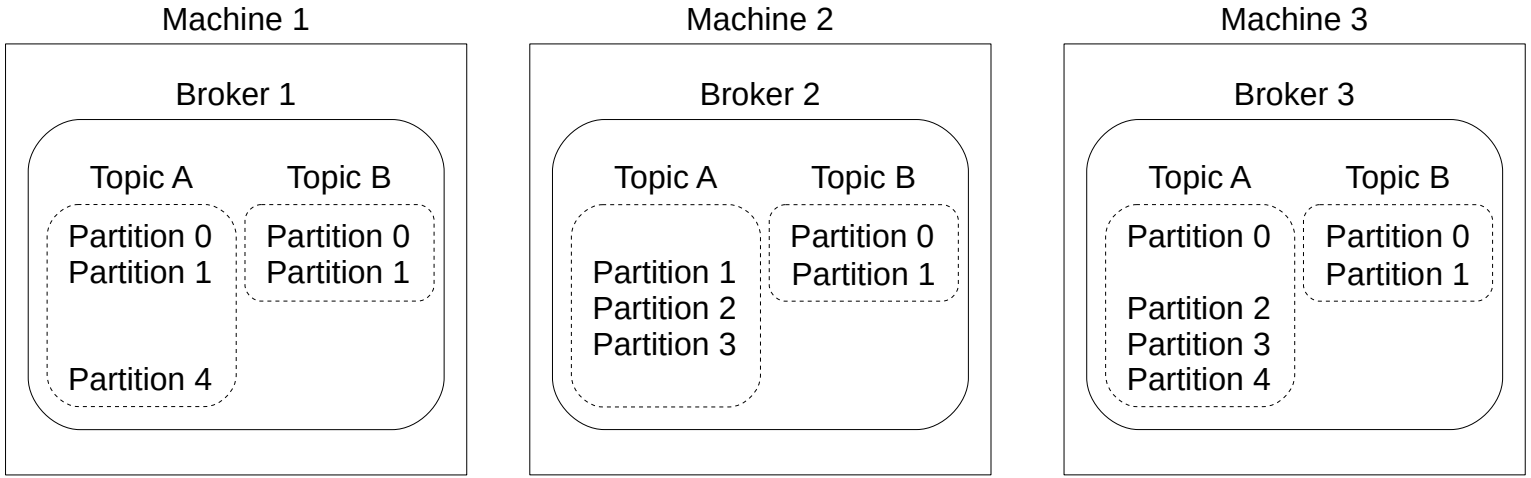
Partition 0 replica list: 3,1 →

- Preferred replica: 3
- Replica leader 3,
- Replica follower: 1

In Sync Replicas (ISR): 3,1

Topic Partitions Replica Assignment

Topic A: Partition Count: **5**, Replication Factor: **2**
Topic B: Partition Count: **2**, Replication Factor: **3**



Topic A

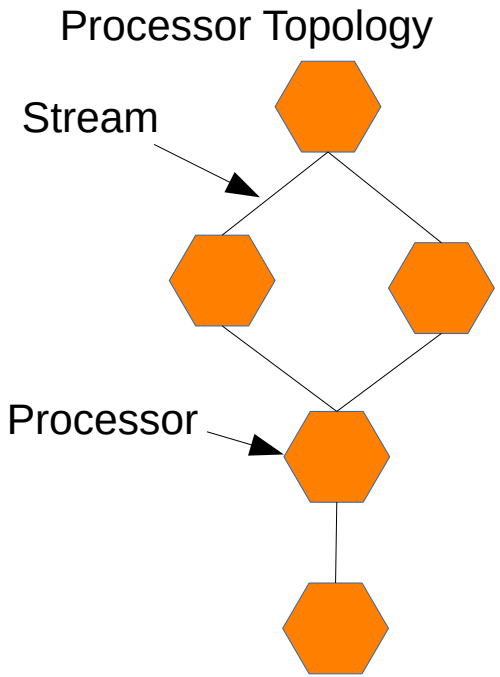
Partition	Latest Offset	Leader	Replicas	In Sync Replicas	Preferred Leader?	Under Replicated?
0	112	3	(3,1)	(3,1)	true	false
1	34	1	(1,2)	(1,2)	true	false
2	19	2	(2,3)	(2,3)	true	false
3	182	3	(3,2)	(3,2)	true	false
4	77	1	(1,3)	(1,3)	true	false

Consumers

Consumers – Streams and Processors

Three APIs

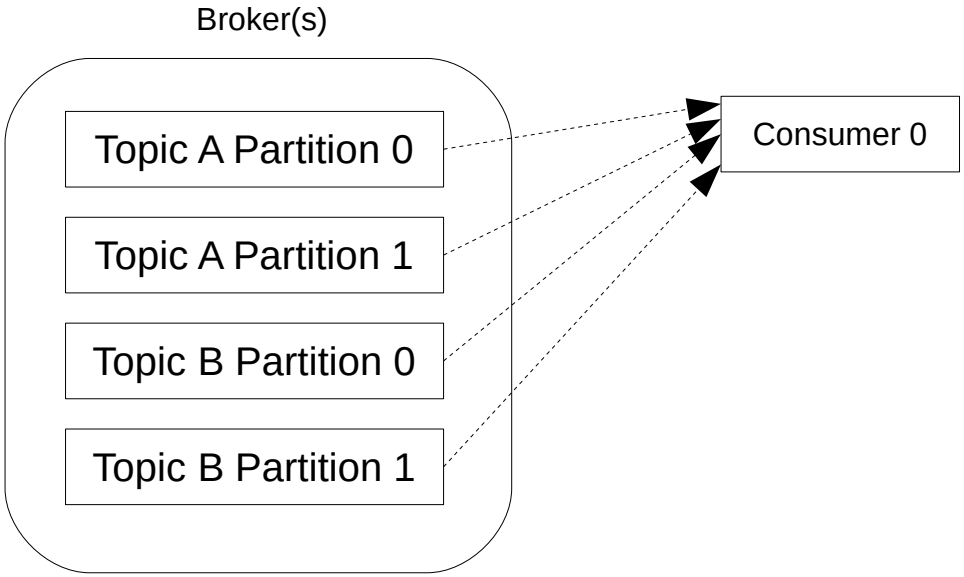
- Topic Processor – Low level
  - Poll or callback
- Stream – High level
  - Unbounded stream of messages
  - forEach, map, reduce, groupBy
  - Stream Joins
- KSQL
  - PRINT ‘topic-name’
  - CREATE STREAM .. AS .. SELECT * FROM .. WHERE ..



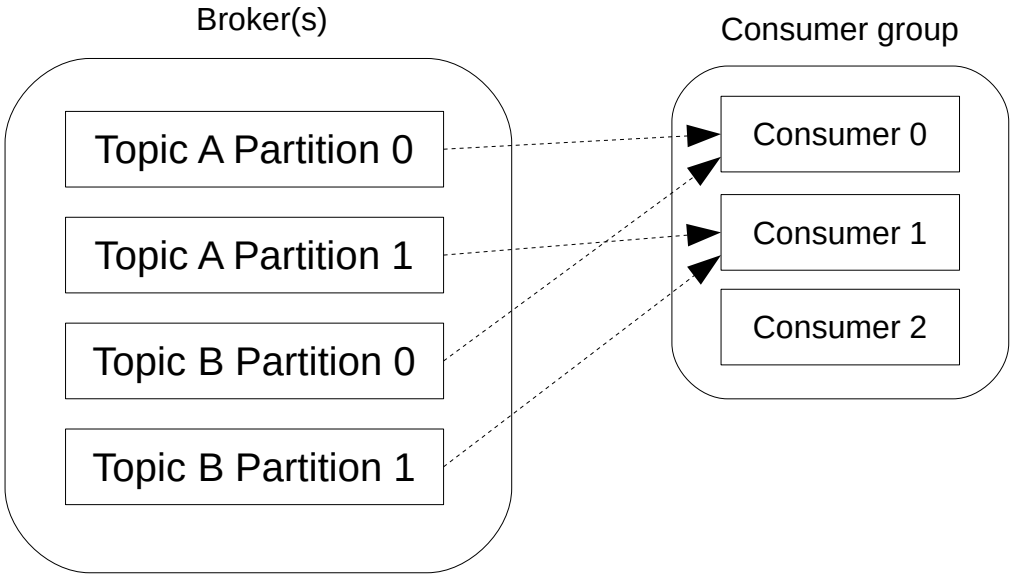
Consumers

- Slow
  - Consumes at their own pace
  - Kafka keeps messages around
- Consumer Groups
  - A named set of consumers
  - Each consumer gets assigned a number of Topic Partitions
  - If Consumers in a group > partitions => idle consumers
  - Command-line tool exists to consume messages from topic

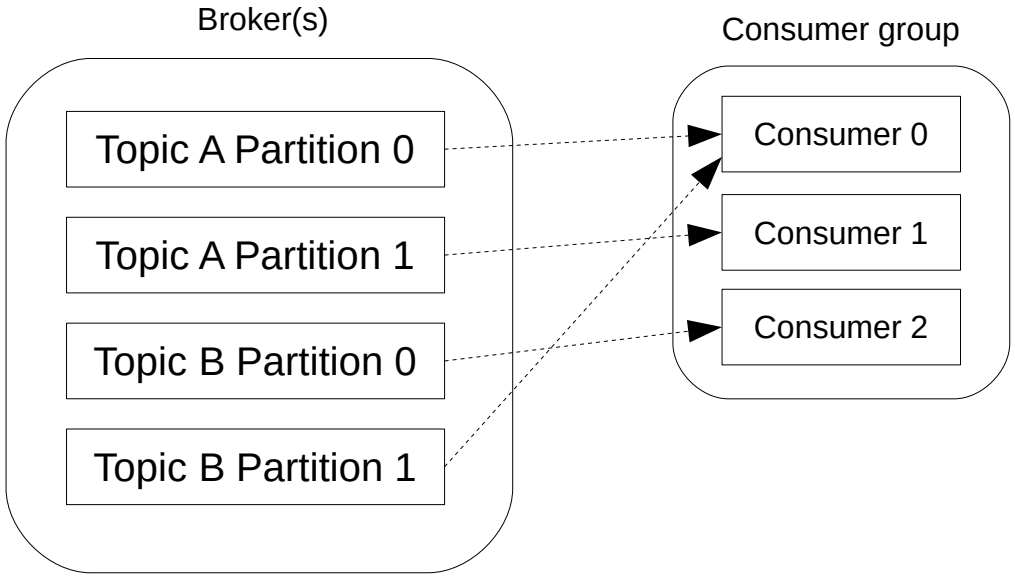
Partition assignment



Partition assignment strategy: Range (default)



Partition assignment strategy: Round-Robin



- Partition assignment is automatically recomputed on changes in Partitions/Consumers

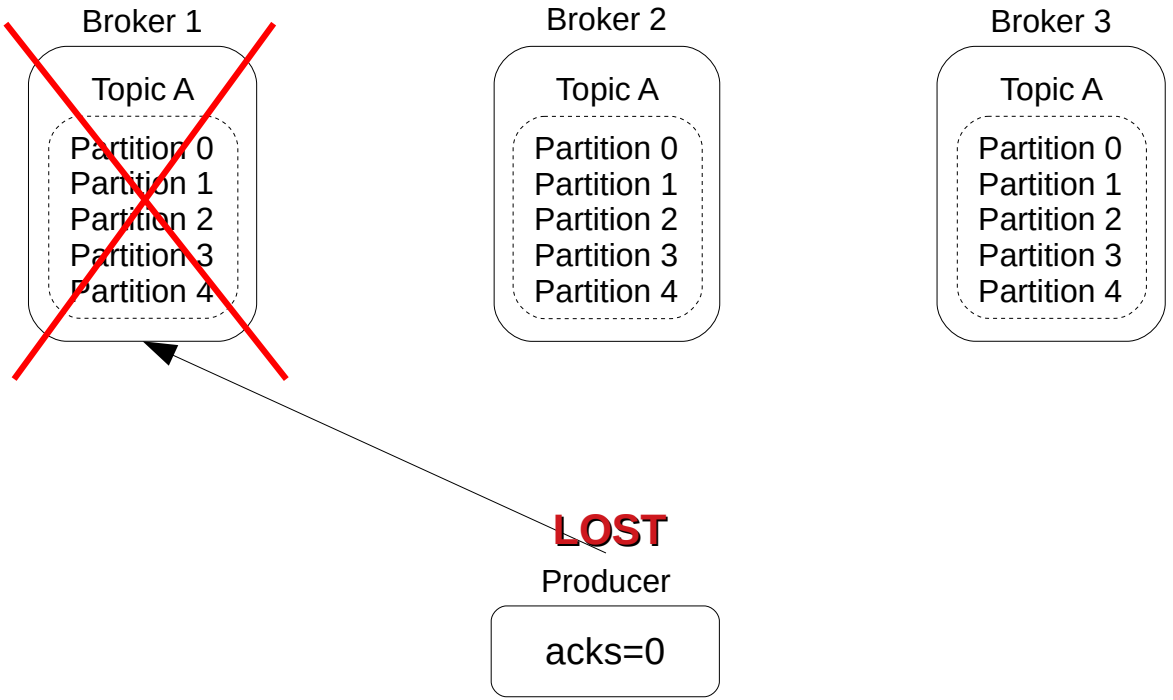
Producers

Producers

- Producers write data in the form of messages to the Kafka cluster
  - Data is persisted
- Producers can be written in any language
  - Native Java, C/C++, Python, Go, .NET, JMS clients are supported by Confluent
  - Clients for many other languages exist
  - REST Server by Confluent which can be used by clients in any language
- Command-line producer exists
  - Useful for testing, debugging etc
- Producer decides on delivery gurarantee
  - acks 0, 1, all (-1)
  - min.insync.replicas – Broker and Topic default, can be overridden

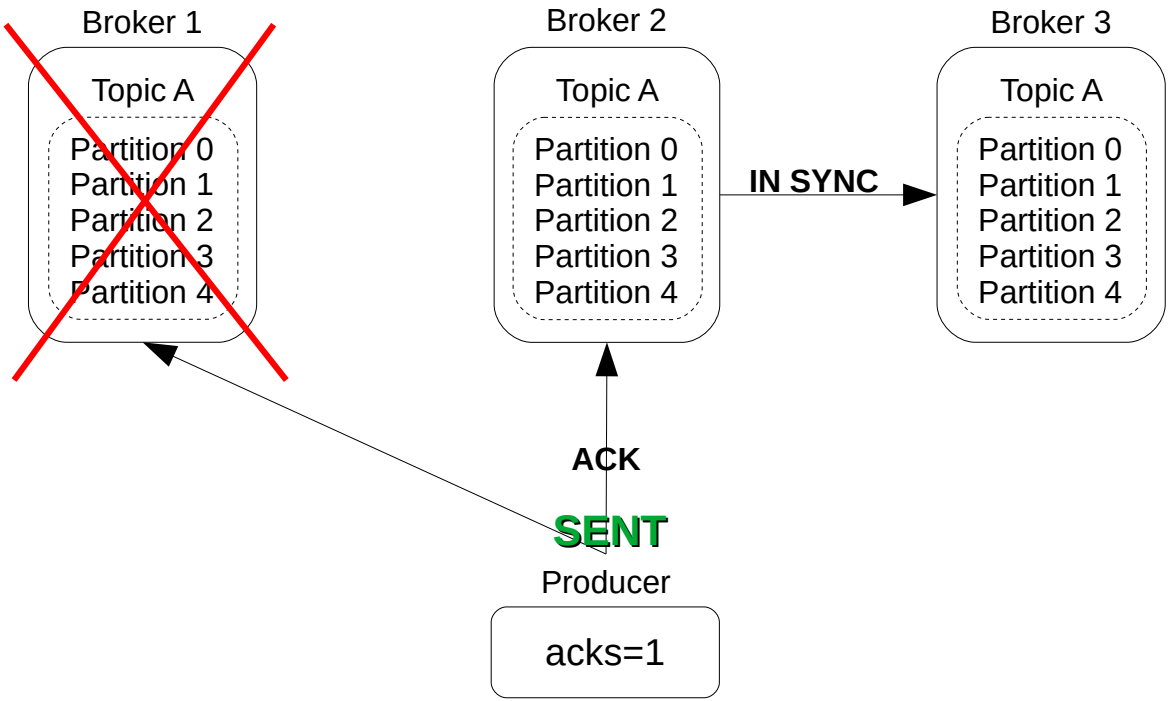
Topic Partitions Replica Assignment

Topic A: Partition Count: **5**, Replication Factor: **3**, min.insync.replicas: **2**



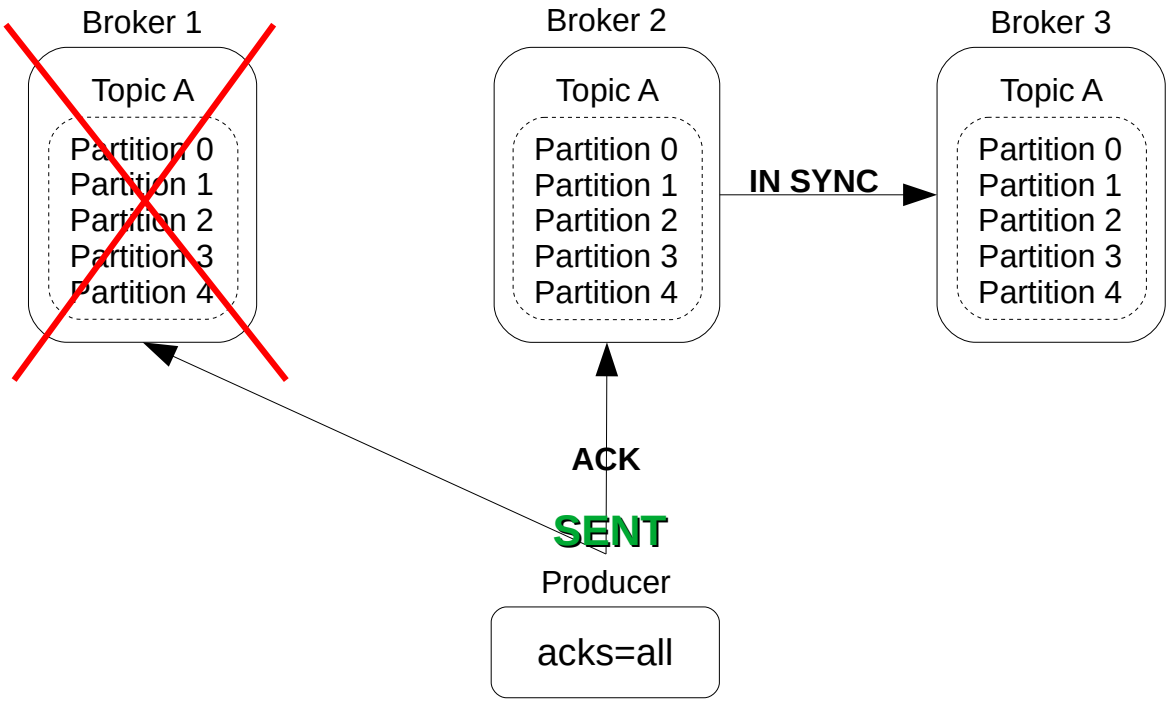
Topic Partitions Replica Assignment

Topic A: Partition Count: **5**, Replication Factor: **3**, min.insync.replicas: **2**

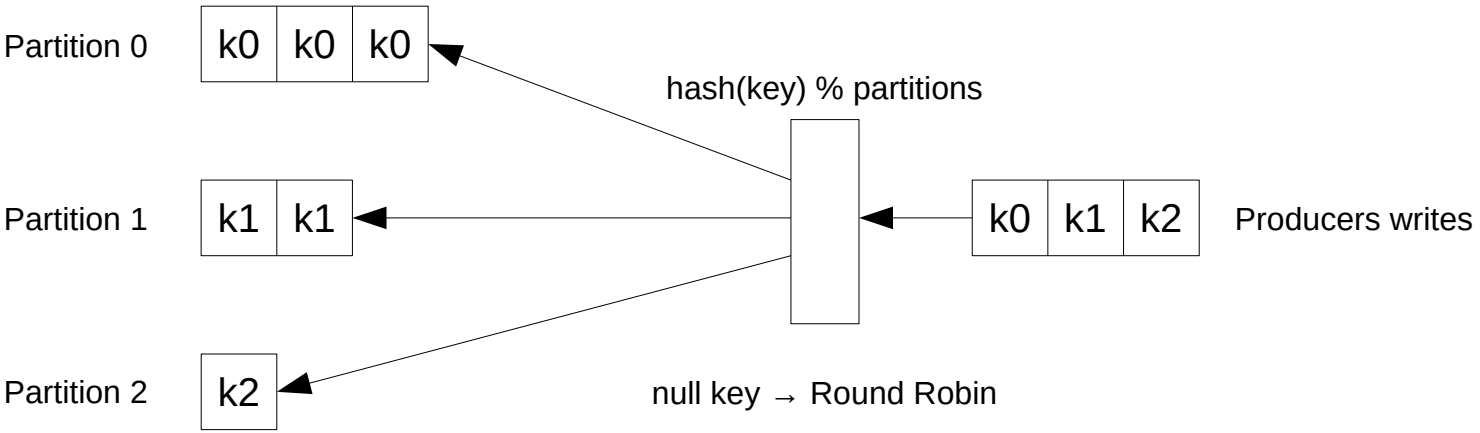


Topic Partitions Replica Assignment

Topic A: Partition Count: **5**, Replication Factor: **3**, min.insync.replicas: **2**

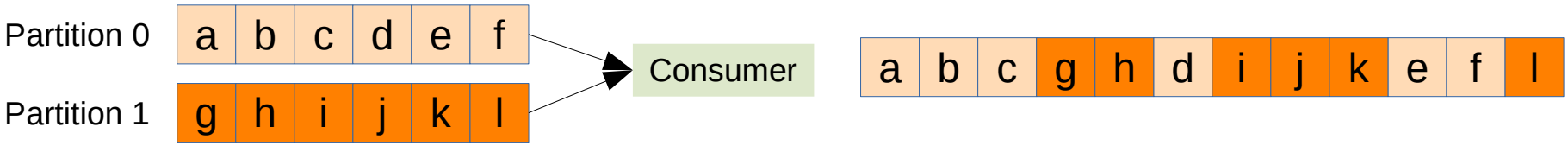


Key hashing


Partition 0
k0
k0
k0
Partition 1
k1
k1
Partition 2
k2
hash(key) % partitions
k0
k1
k2
Producers writes
null key → Round Robin

Message Ordering

- Messages with the same key, from the same Producer, are delivered to the Consumer in order
  - Data within a Partition will be stored in the order written
  - Therefore, data read from a Partition will be read in order *for that Partition*
  - If there are multiple partitions, you will not get total ordering across all messages
    - Question: How can we achieve total ordering if application requires it?



KSQL

KSQL

- Streaming SQL engine
- Provides interactive SQL interface for stream processing
- Scalable, Elastic, Fault Tolerant Real-Time
- Supports wide range of streaming operations
  - Filtering
  - Transformations
  - Aggregations
  - Joins
  - Windowing
  - ...
- CREATE STREAM, TABLE, SELECT FROM, GROUP BY, WINDOW, ...

KSQL

Components

- KSQL Server
  - Runs the engine that executes KSQL queries
- KSQL CLI
  - Interactively write KSQL queries

- https://docs.confluent.io/current/ksql/docs/tutorials/index.html#ksql-tutorials

Join Streams and Tables with KSQL

You can join streams and tables in these ways:

- Two Streams to create a new Stream
- Two Tables to create a new Table
- A Stream and a Table to create a new Stream

Windowed aggregation

- Grouping groups records with the same key
- After grouping, windows allows to further sub-group the records of a key

https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html#windowing

Join Streams and Tables with KSQL

KStream-KStream, KStream-KTable, KTable-KTable

Join Requirements

Co-partitioned Data

- The input records for the join must have the same keying scheme
- The input records must have the same number of partitions on both sides.
- Both sides of the join must have the same partitioning strategy.

If KEY property is set the following must be true

- For every record, the contents of the message key of the Kafka message itself must be the same as the contents of the column set in KEY
- The KEY property must be set to a column of type VARCHAR or STRING

KSQL Example

- Anomaly detection

```
CREATE TABLE possible_fraud AS
  SELECT card_number, count(*)
  FROM authorization_attempts
  WINDOW TUMBLING (SIZE 5 SECONDS)
  GROUP BY card_number
  HAVING count(*) > 3;
```

KSQL Example

- Monitoring

```
CREATE TABLE error_counts AS
  SELECT error_code, count(*)
  FROM monitoring_stream
  WINDOW TUMBLING (SIZE 1 MINUTE)
  WHERE  type = 'ERROR'
  GROUP BY error_code;
```

- More Examples

https://github.com/confluentinc/ksql

Avro

Apache Avro

- Data Serialization System
- Rich Data Structures
- Compact, fast, binary data format
- Container file to store persistent data
- Defines fields, type and optionality of your data
- Multiple language support

Avro example

```
{
    "namespace": "se.ryz.kafka.demo.avro.zombieweapon",
    "type": "record",
    "name": "Chainsaw",
    "doc": "Number 1 option",
    "fields": [
        {"name": "model", "type": "string", "doc": "Zombi ZCS5817, Zombi ZCS12017 etc"}
    ]
}
```

Schema Registry

Schema Registry

- REST interface to managing Avro Schemas
- Versioned history of schemas
- Configurable evolution of schemas
- Serializers that plug into Kafka clients
- Assigns globally unique ID to schema
- Designed to be distributed

Avro - Schema Registry

Two new configurations were added to Schema Registry

- key.subject.name.strategy and value.subject.name.strategy
  - How to construct the subject name for message keys and values respectively

Three strategies

- TopicNameStrategy (Default)
  - Keys: <topic>-key, Values: <topic>-value. This means one schema per topic
- RecordNameStrategy
  - <topic> Subject name is fully-qualified name of the Avro record type of the message -> Any number of different event types in the same topic
- TopicRecordNameStrategy
  - <topic>-<type> Also allows any number of event types in the same topic. Further constrains compatibility check to the current topic only

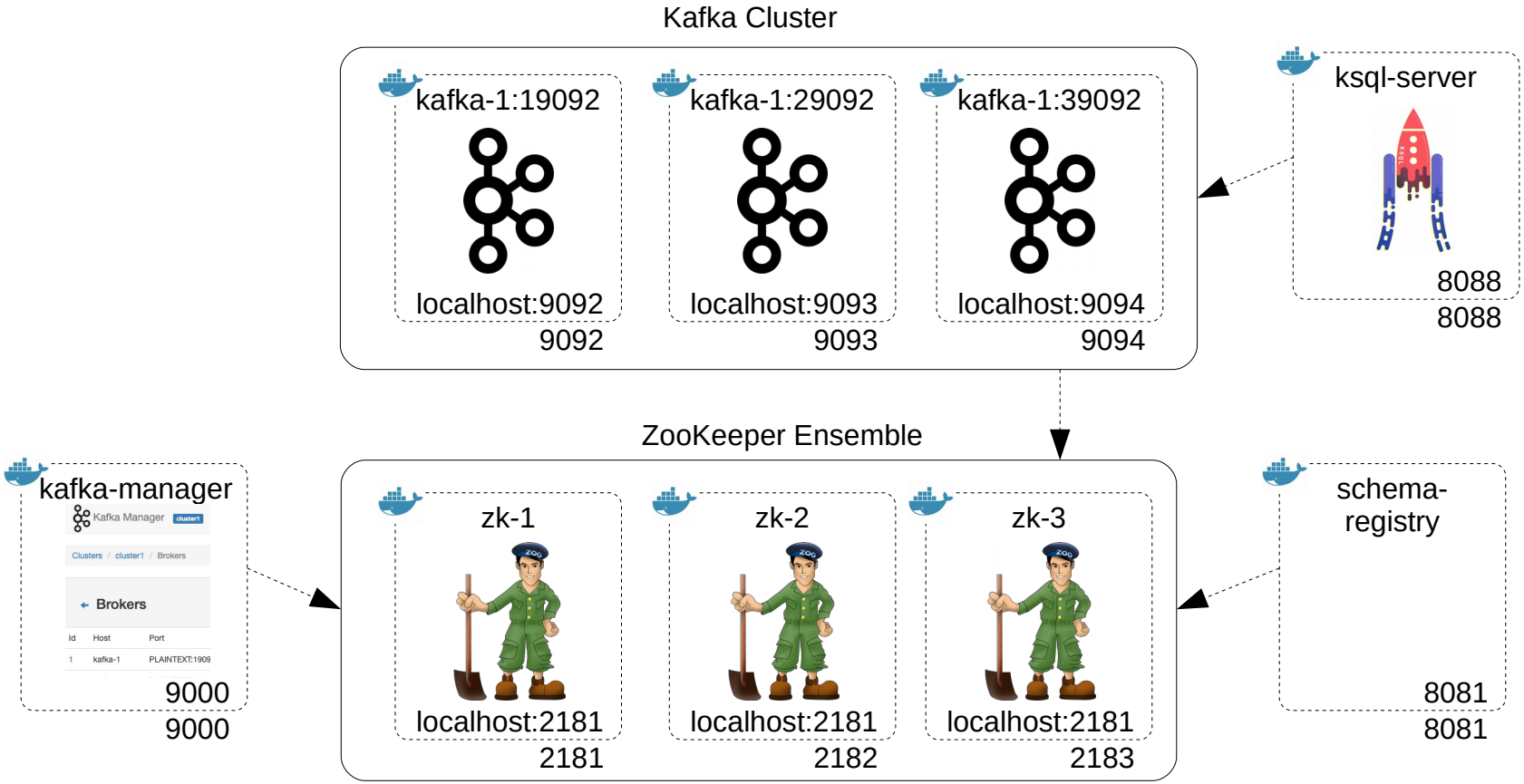
Connectors

Connectors

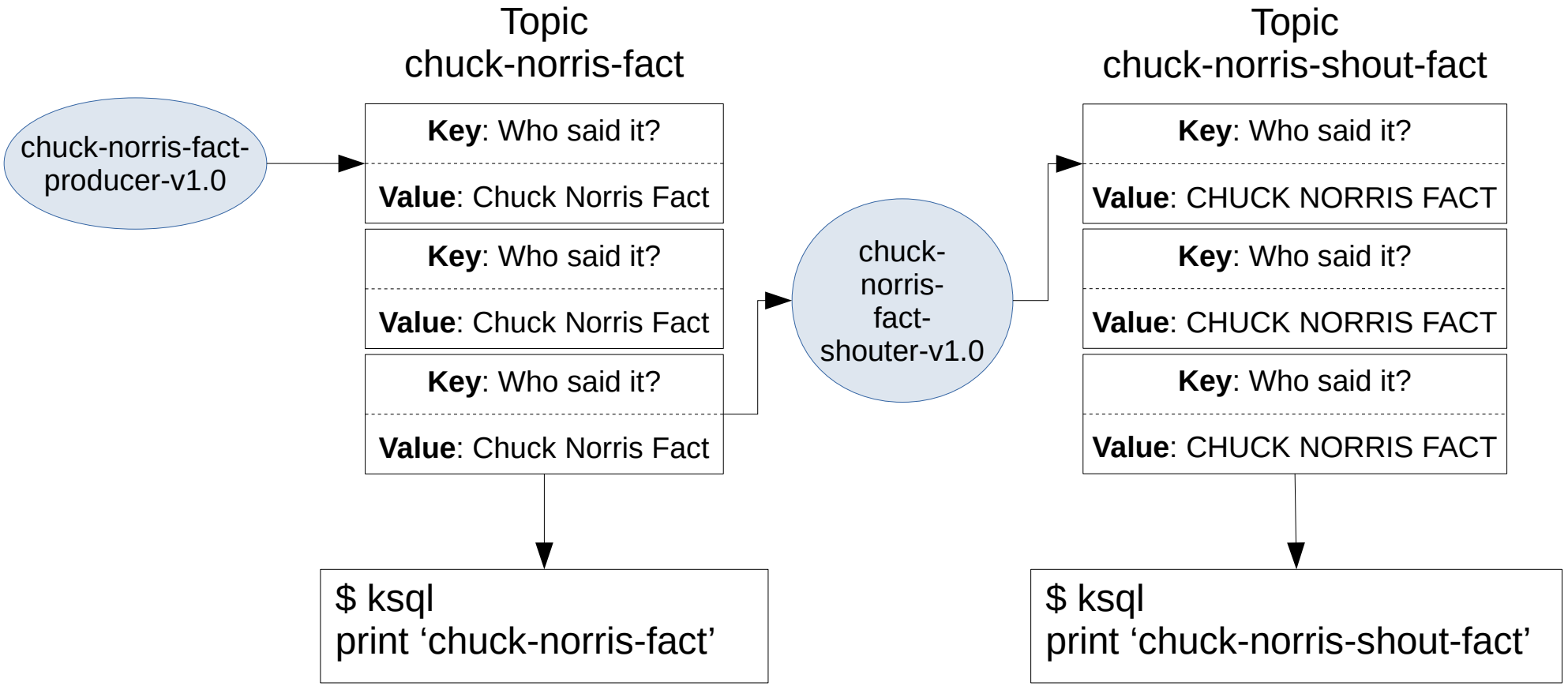
- Open source, a framwework for connecting Kafka to external systems
  - Databases, key-value stores, search indexes, …
- Source connector
  - Reads data and sends it to Kafka
  - Can also collect metrics from application servers and make the data available for stream processing with low latency
- Sink connector
  - Takes data from Kafka and exports it to other system
- Core components
  - Connectors, Tasks, Workers, Converters, Transform
- Example: Connector for MySQL listens to the binlog

Demo

Lab Cluster


Kafka Cluster
kafka-1:19092
localhost:9092
9092
kafka-1:29092
localhost:9093
9093
kafka-1:39092
localhost:9094
9094
ksql-server
8088
8088
ZooKeeper Ensemble
kafka-manager
Kafka Manager
cluster1
Clusters / cluster1 / Brokers
Brokers
Id
Host
Port
1
kafka-1
PLAINTEXT:1909
9000
9000
zk-1
localhost:2181
2181
zk-2
localhost:2181
2182
zk-3
localhost:2181
2183
schema-registry
8081
8081

ChuckNorrisFact



Tack!

Presentation, kluster och labbar

git clone https://github.com/ryzlab/kafka-kompetensdag.git

Många bra exempel och tips

git clone https://github.com/confluentinc/kafka-streams-examples.git