

[544] Kafka Streaming

Tyler Caraza-Harter

Learning Objectives

- describe the benefits of using streaming for ETL (extract transform load) work
- write code for Kafka **consumers** and **producers** in order to interact with **topic** data that stored by **brokers**
- scale out brokers and consumers by configuring **topic partitions** and **consumer groups**, respectively

Outline: Kafka Streaming

Sending/Receiving Messages

- RPC (Remote Procedure Calls)
- Streaming

ETL (Extract Transform Load)

Kafka Design

Procedure Calls

```
counts = {  
    "A": 123, ...  
}  
  
def increase(key, amt):  
    counts[key] += amt  
    return counts[key]  
  
curr = increase("A", 5)  
print(curr) # 128
```

what if we want many programs running
on different computers to have access to
this dict and the increase function?

Remote Procedure Calls (RPCs)

client

```
curr = increase("A", 5)
print(curr) # 128
```

server

```
counts = {
    "A": 123, ...
}

def increase(key, amt):
    counts[key] += amt
    return counts[key]
```

client

move counts and increase to a server
accessible to many client programs on
different computers

...

Remote Procedure Calls (RPCs)

client

```
def increase(key, amt):  
    ...code to send
```

```
curr = increase("A", 5)  
print(curr) # 128
```

computer 1

server

```
def rpc_server():  
    ...code to receive
```

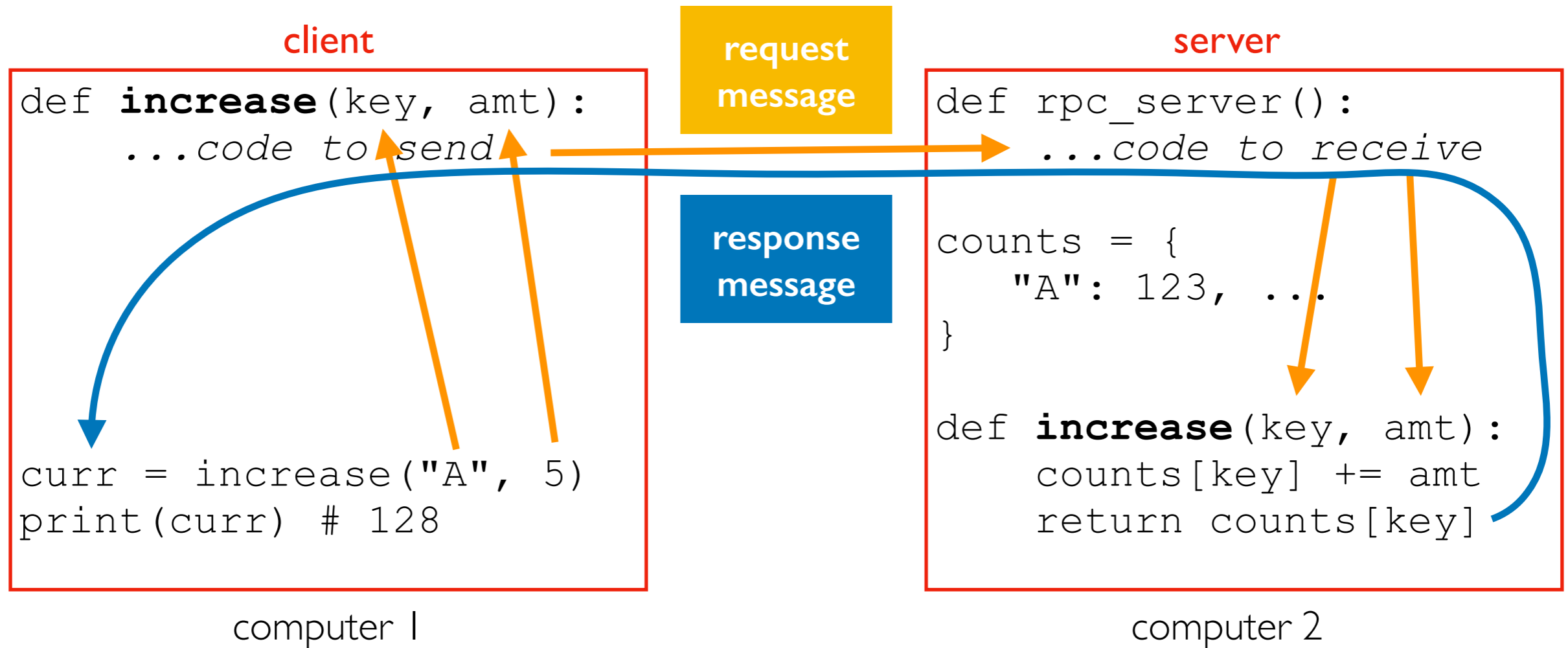
```
counts = {  
    "A": 123, ...  
}
```

```
def increase(key, amt):  
    counts[key] += amt  
    return counts[key]
```

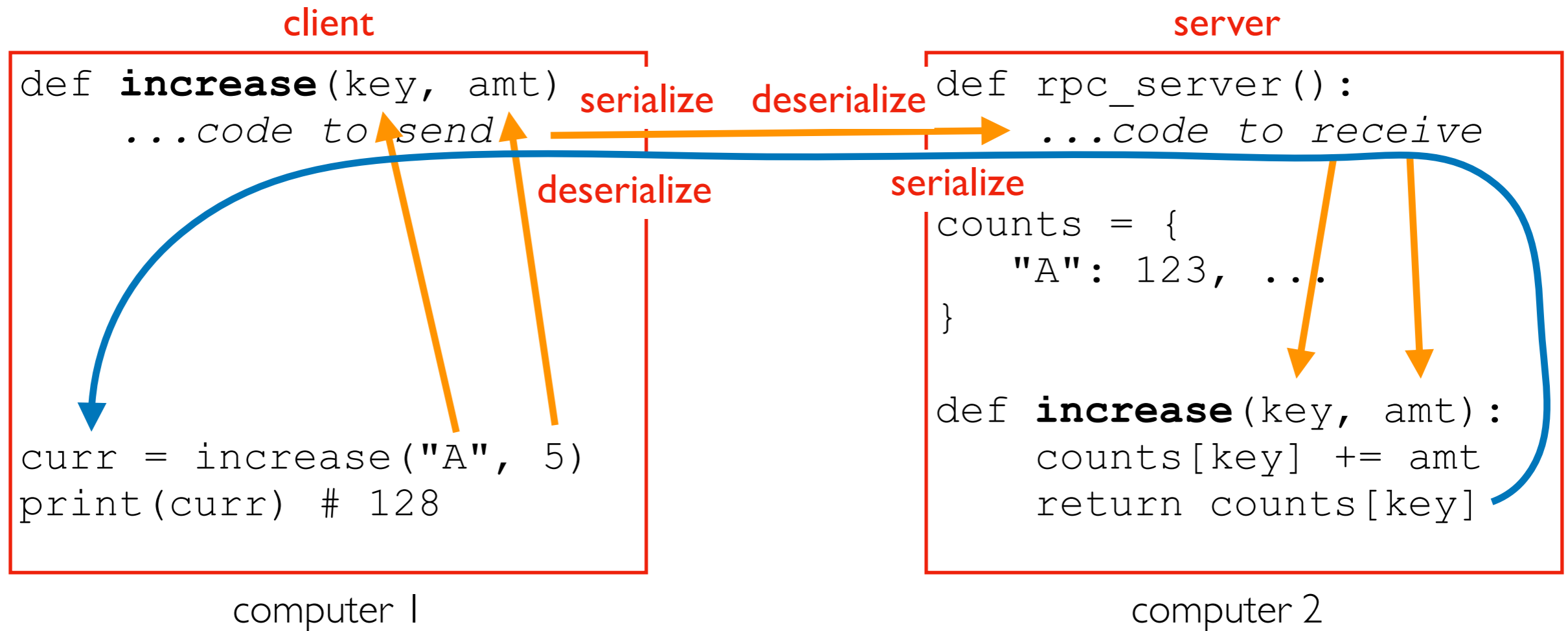
computer 2

need some extra functions to make calling a remote function *feel* the same as calling a regular one

Remote Procedure Calls (RPCs)



Serialization/Deserialization



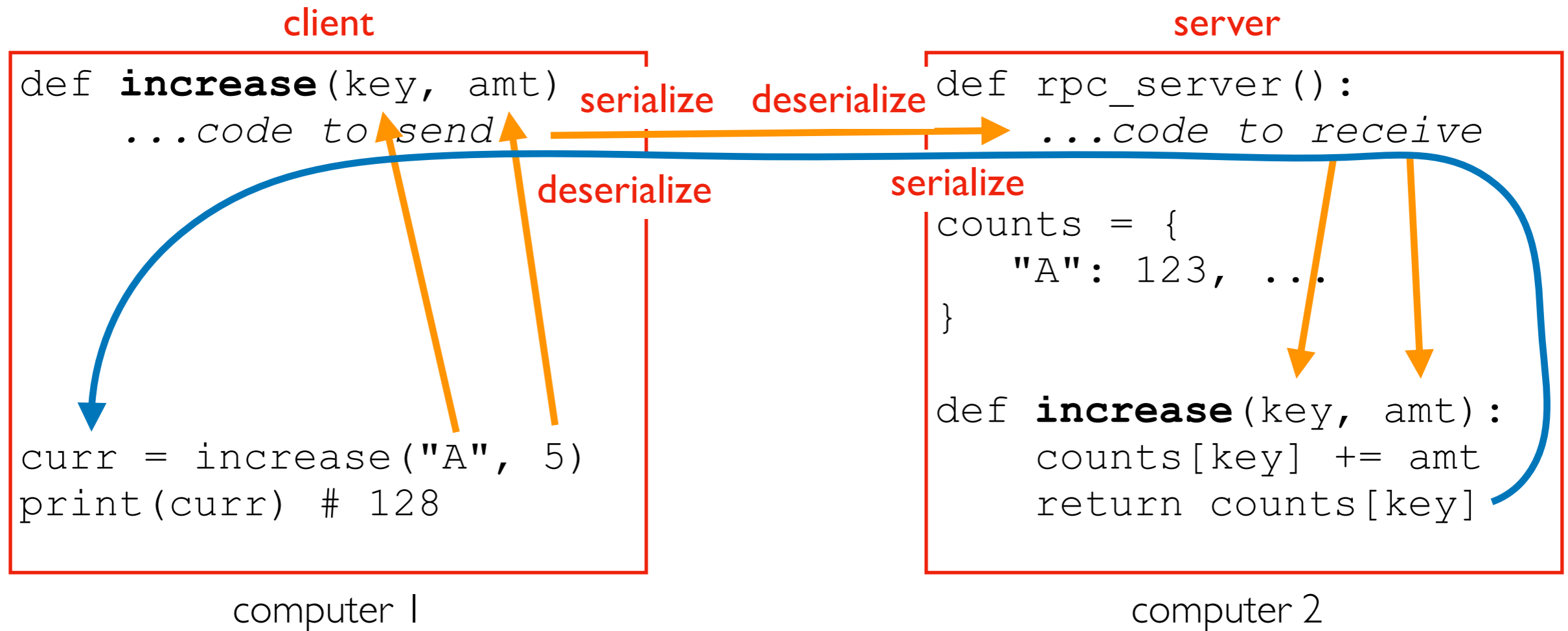
**request
message**

```
args somehow encoded as bytes:
b'{"key": "A"
  "amt": 5}'
```

**response
message**

```
return val as bytes:
b'5'
```


gRPC uses protocol buffers for wire format



**request
message**

```
protobuf (args to bytes)
b'1001000101011111'
(contains "A" and 5)
```

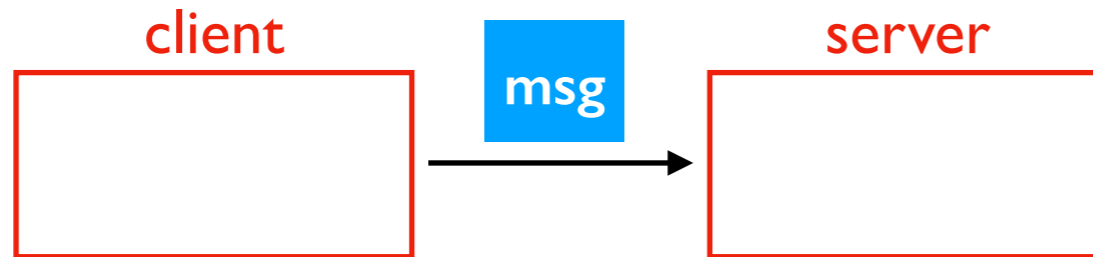
**response
message**

```
protobuf (ret val to bytes)
b'01000000'
(contains 128)
```

Synchronous vs. Asynchronous Communication

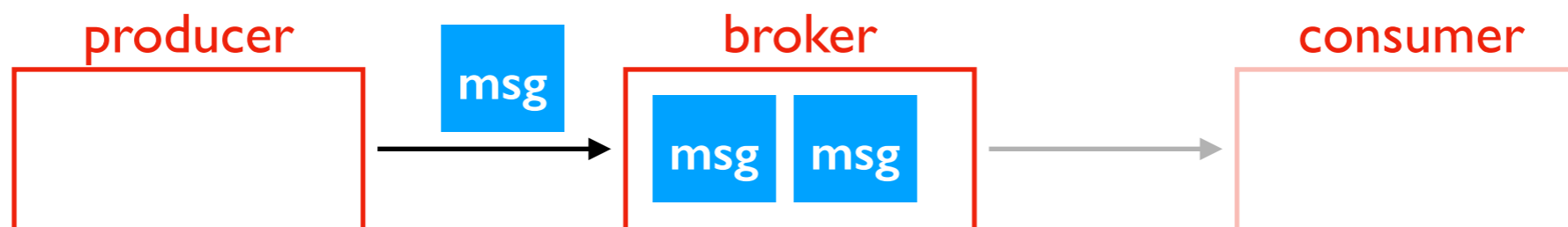
Synchronous

- both parties have to participate at the same time
- examples: phone call, RPC call



Asynchronous

- one party can send any time, the other can receive later
- examples: email, texting, **streaming**



Outline: Kafka Streaming

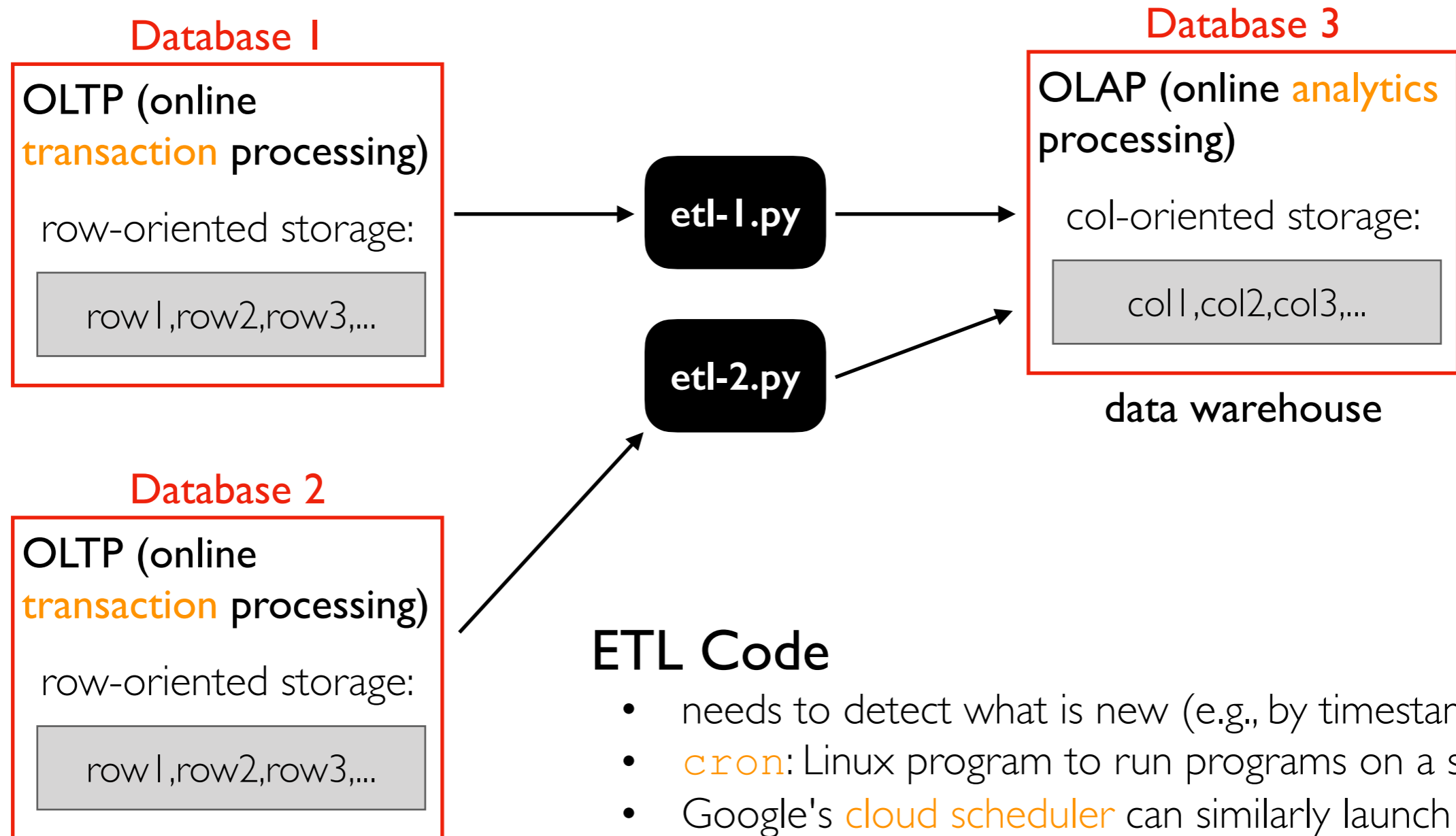
Sending/Receiving Messages

ETL (Extract Transform Load)

- Batch
- Streaming

Kafka Design

Extract Transform Load (ETL)

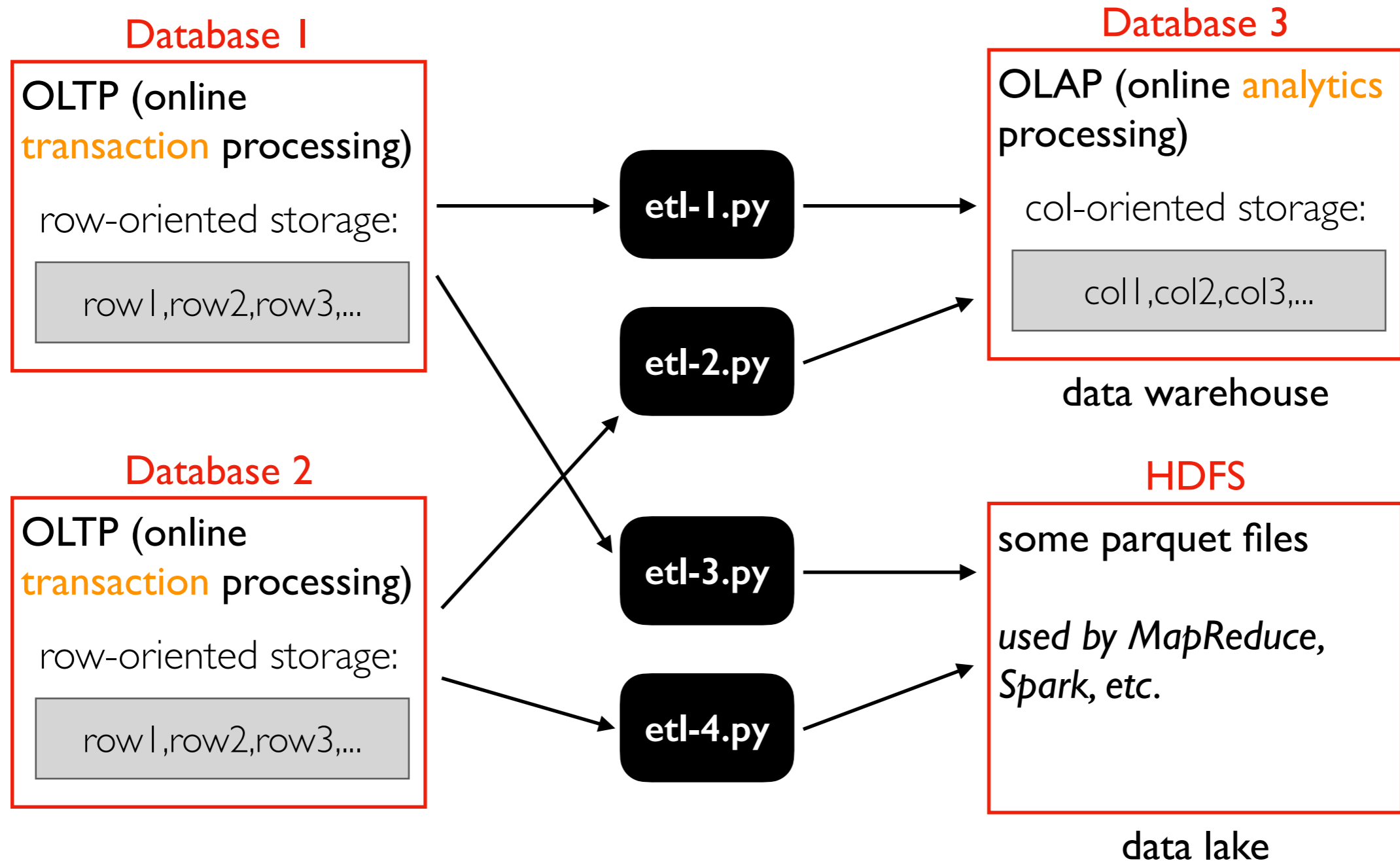


ETL Code

- needs to detect what is new (e.g., by timestamp)
- **cron**: Linux program to run programs on a schedule
- Google's **cloud scheduler** can similarly launch tasks (other clouds have similar options)

issue 1: data freshness

Extract Transform Load (ETL)



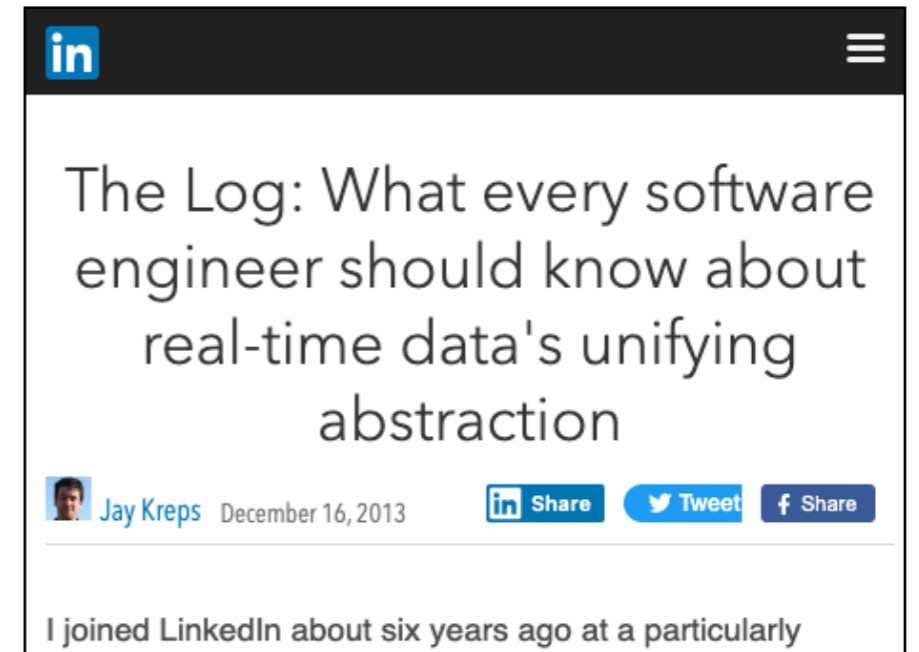
if we have **X** OLTP databases and **Y** derivative stores, how many ETL programs must we write?

issue 2: scaling engineering effort

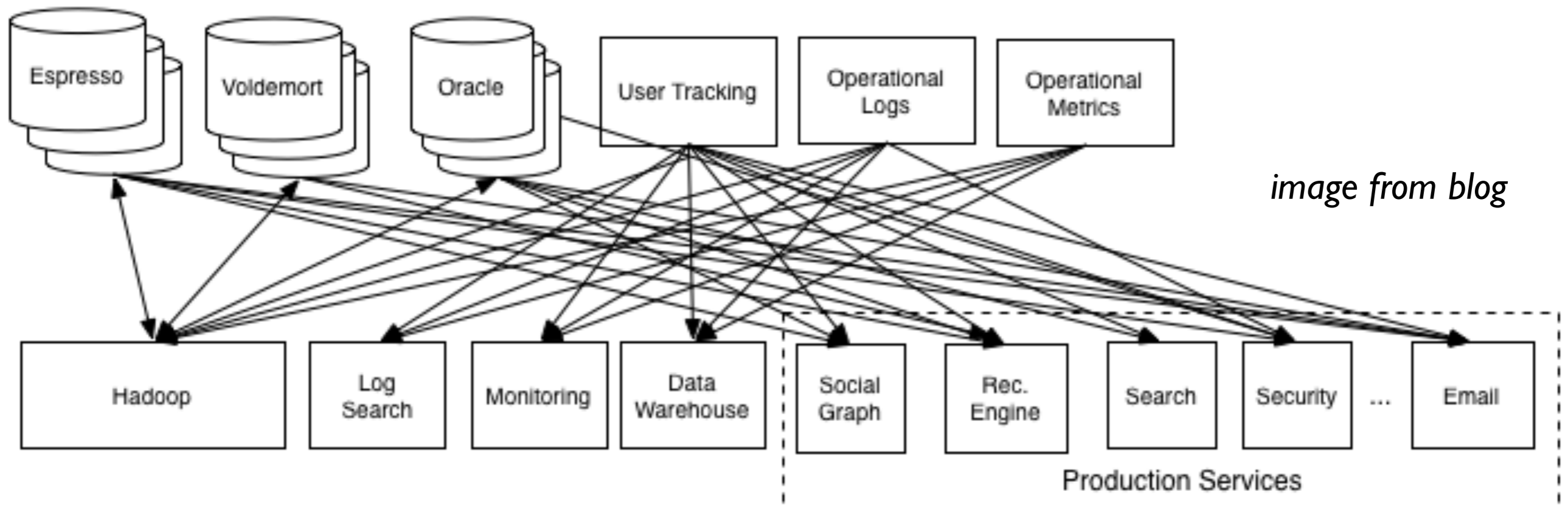
Too much ETL...

Don't want data transfer between every pair of DB/services

- Jay Krepps helped build Kafka at LinkedIn
- Later co-founded Confluent (Kafka-based company)
- Partners with cloud providers to provide Kafka as a service



<https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>

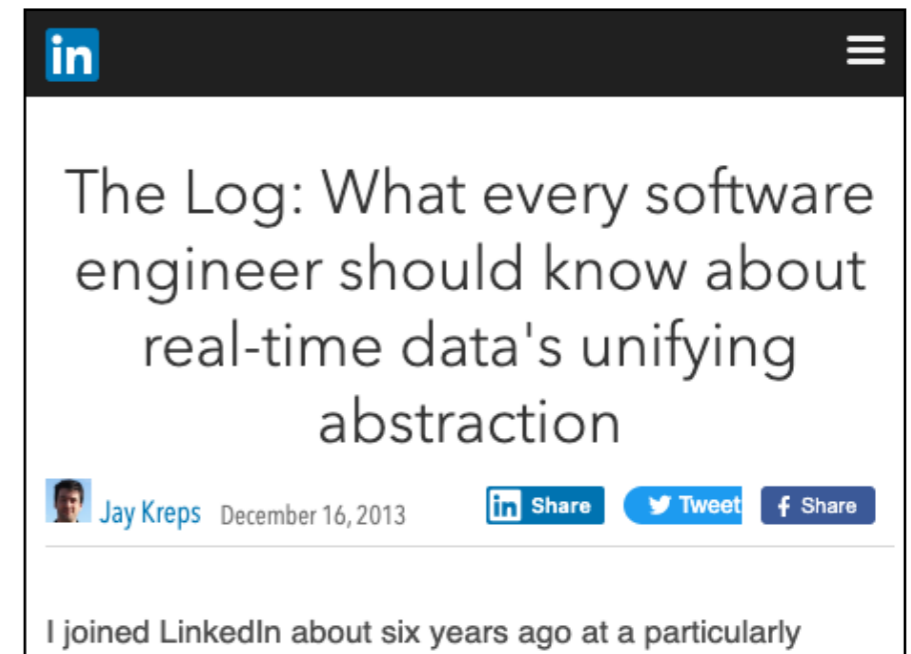


Unified Log

Centralize changes in a distributed logging service

- Many writers (called producers)
- Many readers (called consumers)

Data is constantly flowing, so ETL can be done in realtime (instead of batch jobs with cron)



<https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>

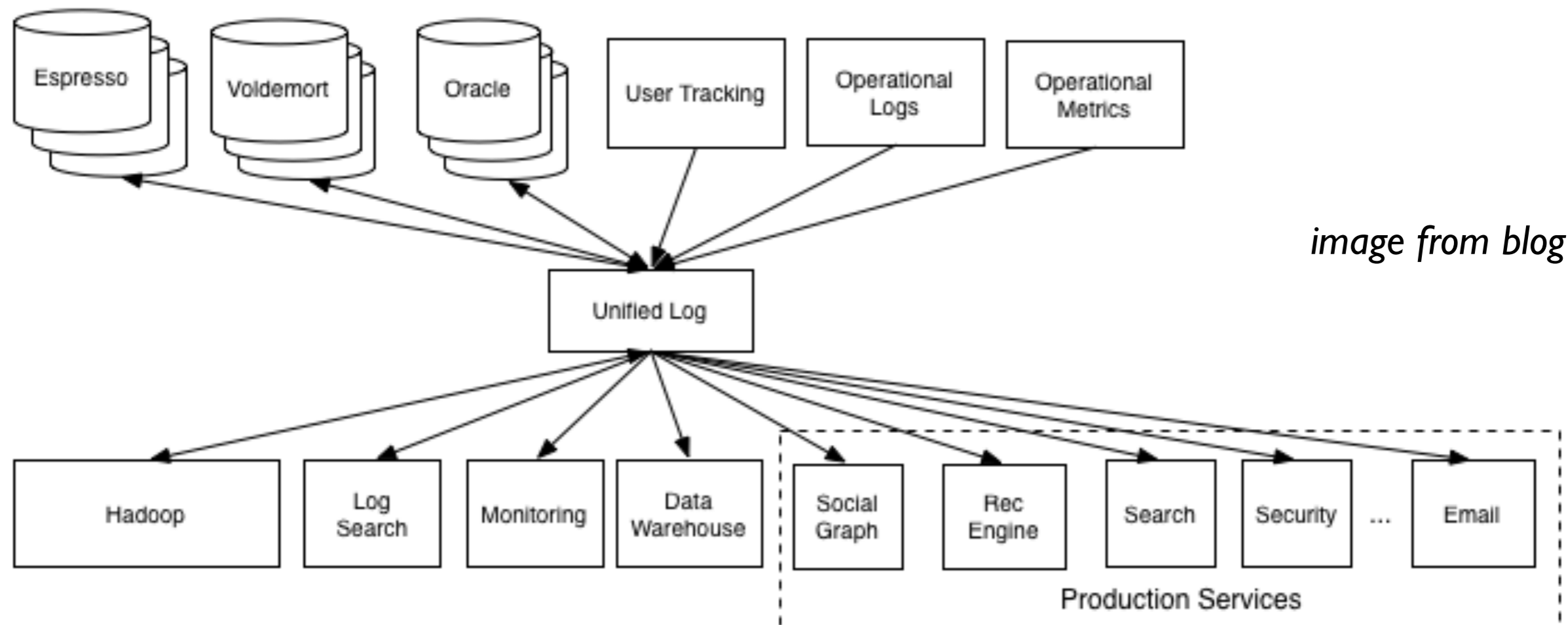


image from blog

Outline: Kafka Streaming

Sending/Receiving Messages

ETL (Extract Transform Load)

Kafka Design

- Topics
- Producers, Consumers, Brokers
- Scalability with Partitioning

Topics

Kafka **topics** (managed by servers called **brokers**)



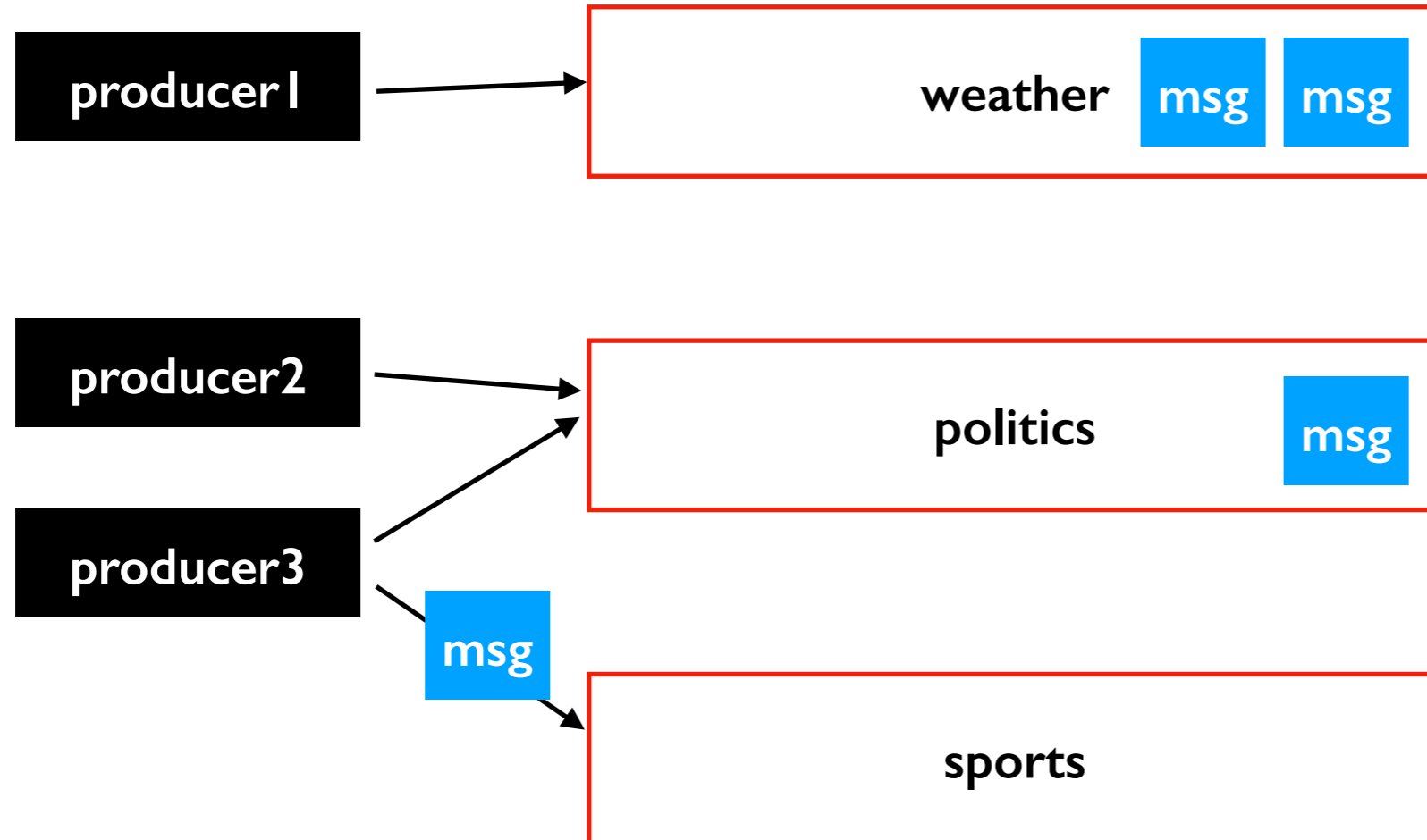
```
admin = KafkaAdminClient(...)  
admin.create_topics([NewTopic("sports", ...)])
```

```
pip install kafka-python
```

Producers Publish (pub/sub)

producers
(code you write)

Kafka topics (managed by
servers called brokers)



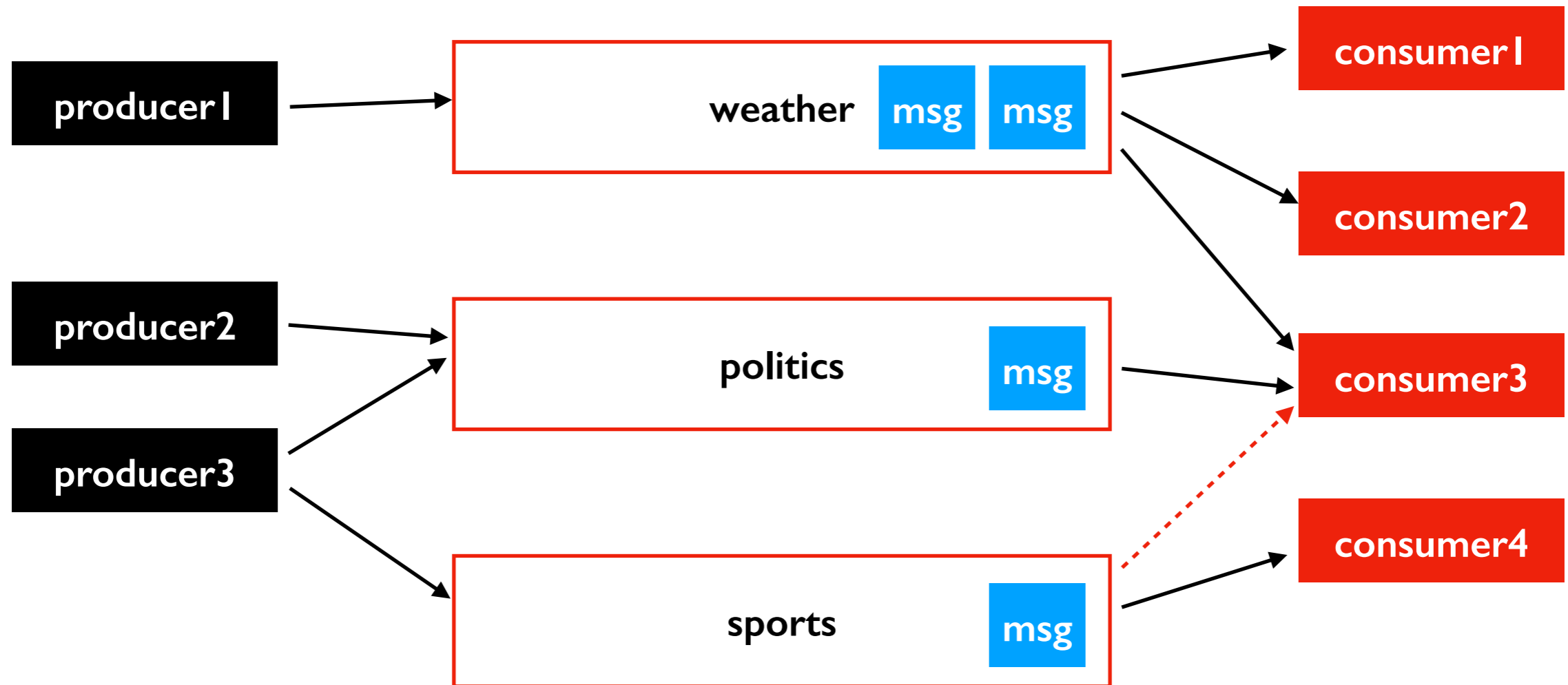
```
producer3 = KafkaProducer(...)  
producer3.send("sports", ...)
```

Consumers Subscribe (pub/sub)

producers
(code you write)

Kafka **topics** (managed by
servers called **brokers**)

consumers
(code you write)



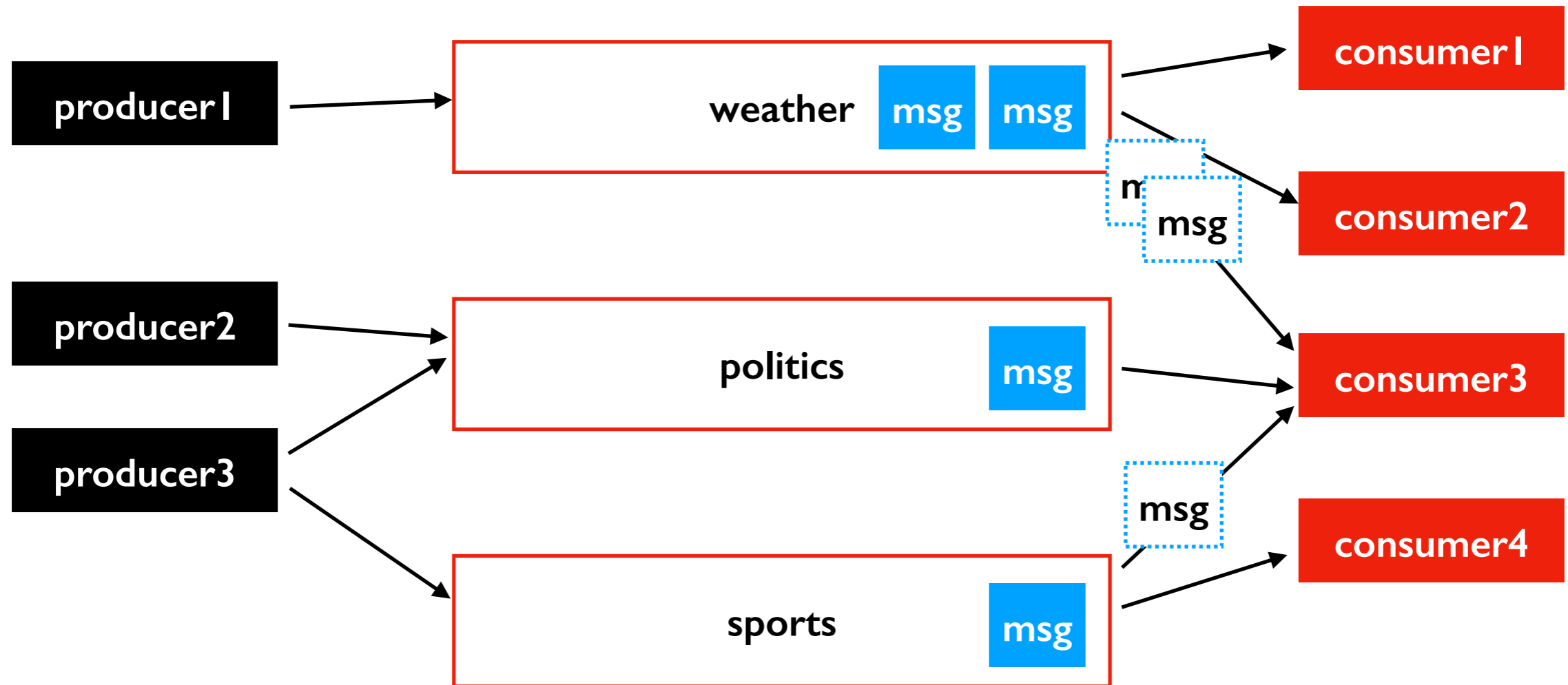
```
consumer3 = KafkaConsumer(...)  
consumer3.subscribe(["sports"])
```

Receiving Messages

producers
(code you write)

Kafka **topics** (managed by
servers called **brokers**)

consumers
(code you write)



`poll()` loop

- generally runs forever
- poll (ideally) returns some messages the consumer hasn't seen before, from any subscribed topic
- leaves messages intact on brokers (for other consumers), unlike many prior streaming systems

```
consumer3 = KafkaConsumer(...)
while True:
    batch = consumer3.poll(???)
    for topic, messages in batch.items():
        for msg in messages:
            ...
```

What's in a Message?

Message parts

- **key** (optional): *some bytes*
- **value** (required): *some bytes*
- other stuff...

```
producer.send("topic", value=????)
```

OR

```
producer.send("topic", value=????, key=????)
```

Common usage: the value is usually some kind of structure with many values. The key is used for partitioning and is usually one of the entries in the value structure.

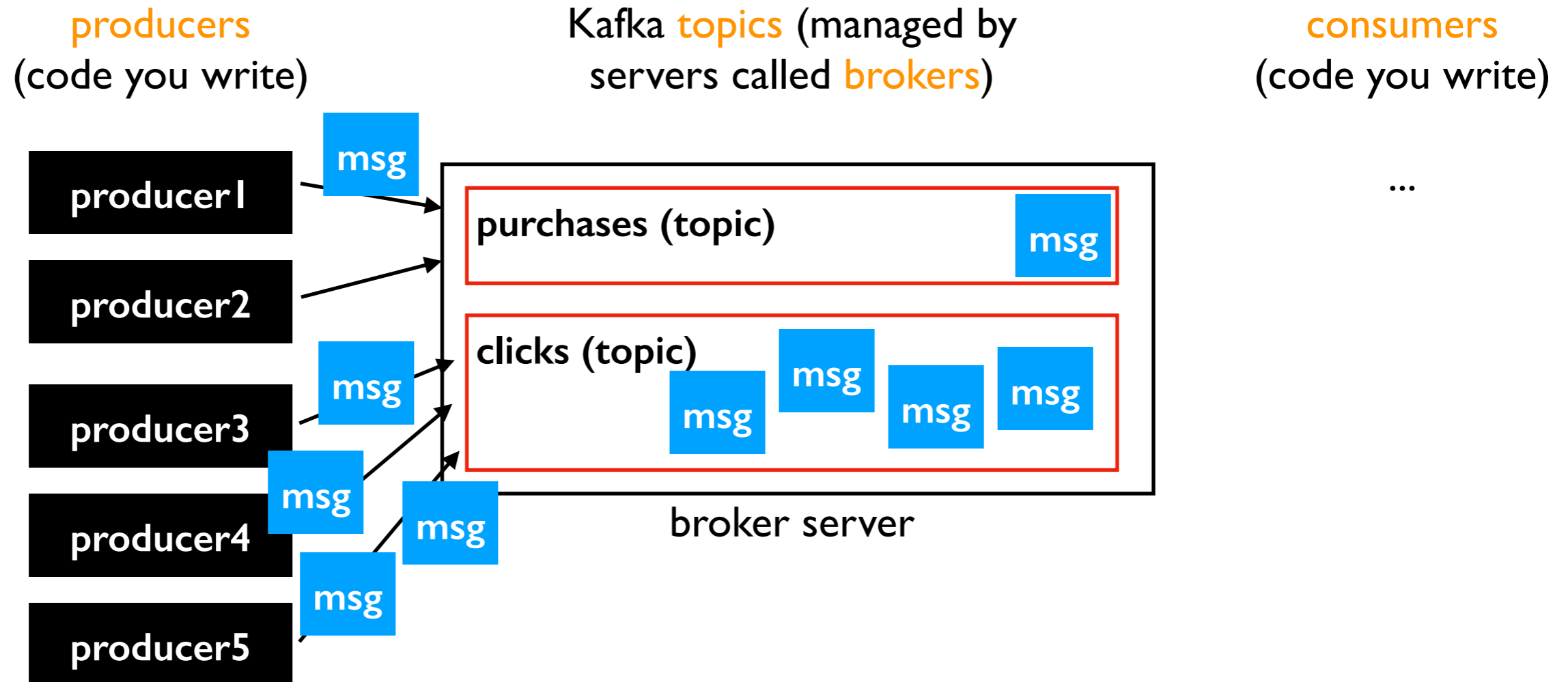
Python dict => bytes:

```
d = {...}
value = bytes(json.dumps(d), "utf-8")
```

Protobuf => bytes:

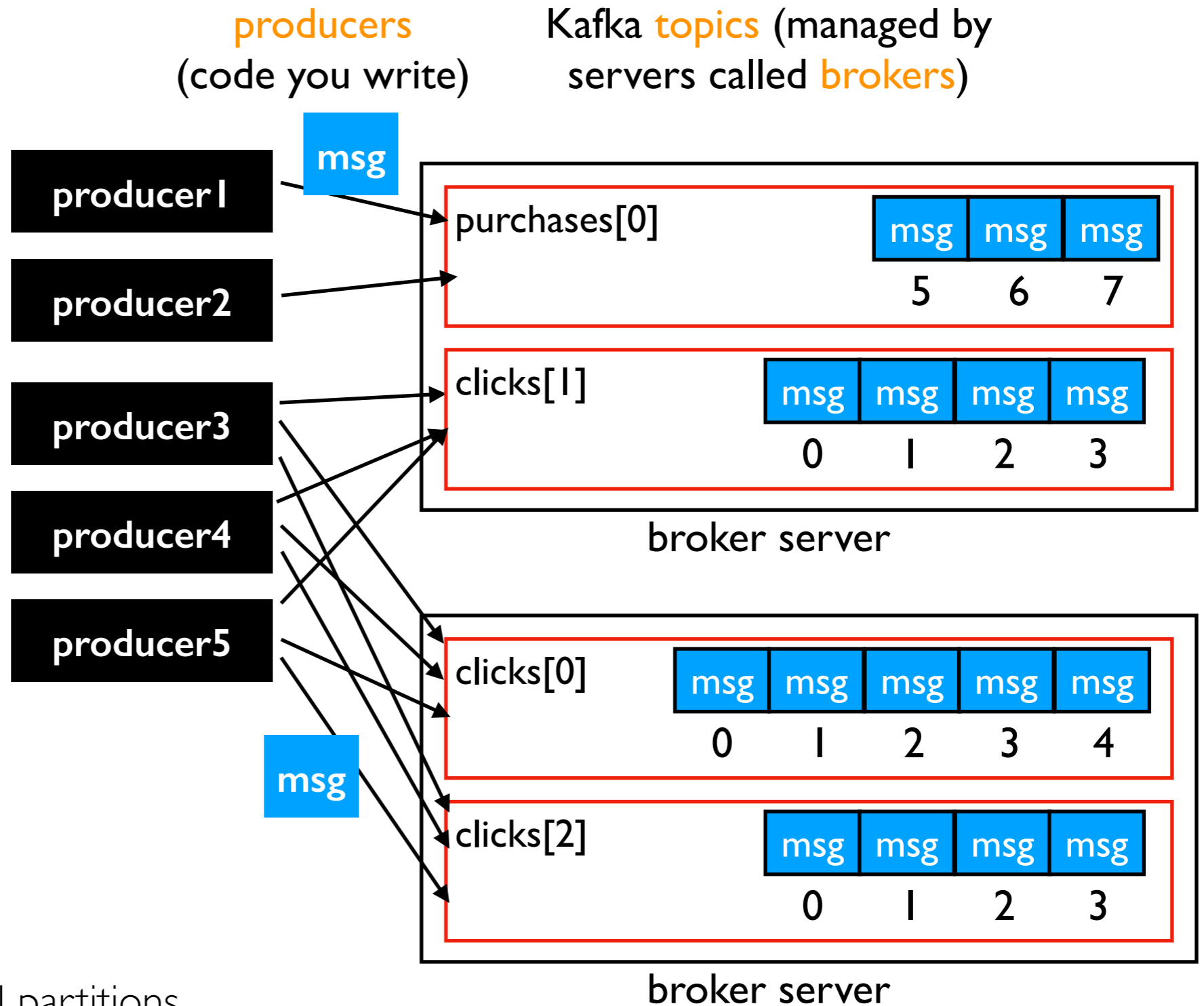
```
msg = mymod_pb2.MyMessage(...)
value = msg.SerializeToString() # actually bytes, not str
```

Scaling the Brokers



problem: some topics might have too many messages for one machine (or set of machines with replicas) to keep up

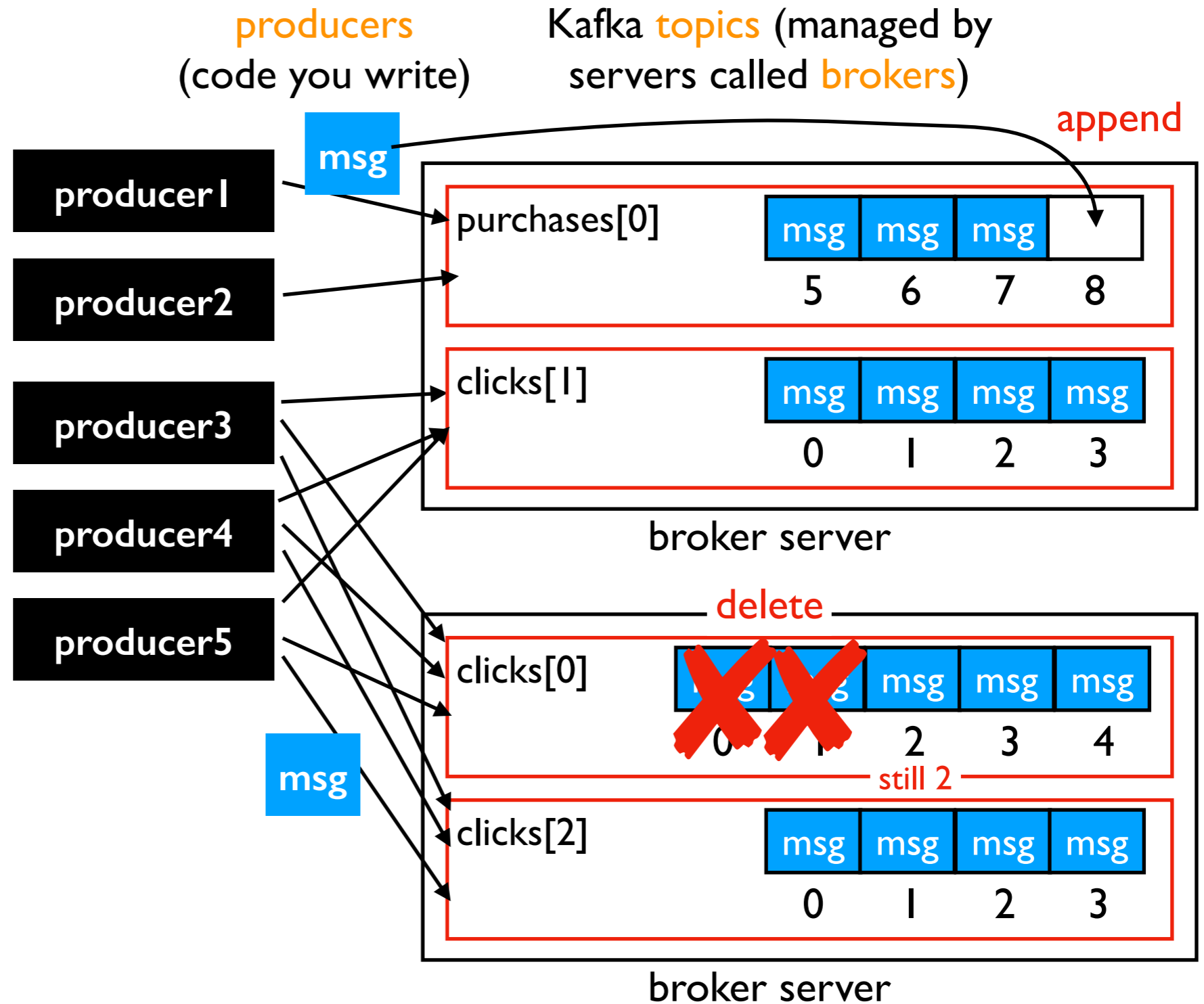
Partitions



Topics can be created with N partitions

- each partition is like an array of messages
- partitions are assigned to brokers
- each producer using a stream works with all partitions

Changing Partitions



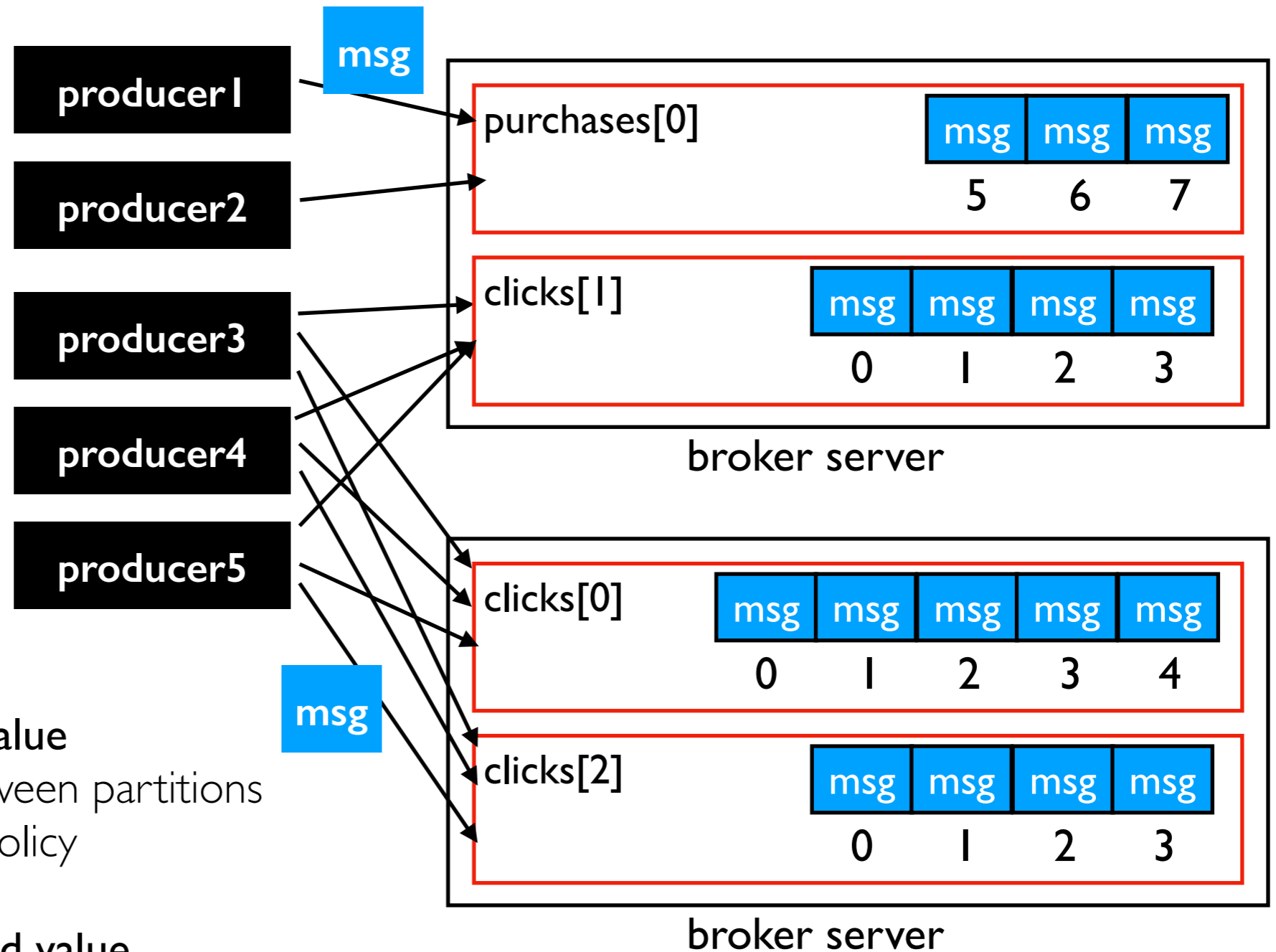
Changes

- **append** right
- **delete** left (depends on "retention" policy)
- **delete** does **NOT** change indexes

Selecting Partitions

producers
(code you write)

Kafka topics (managed by
servers called brokers)



case 1: message only has value

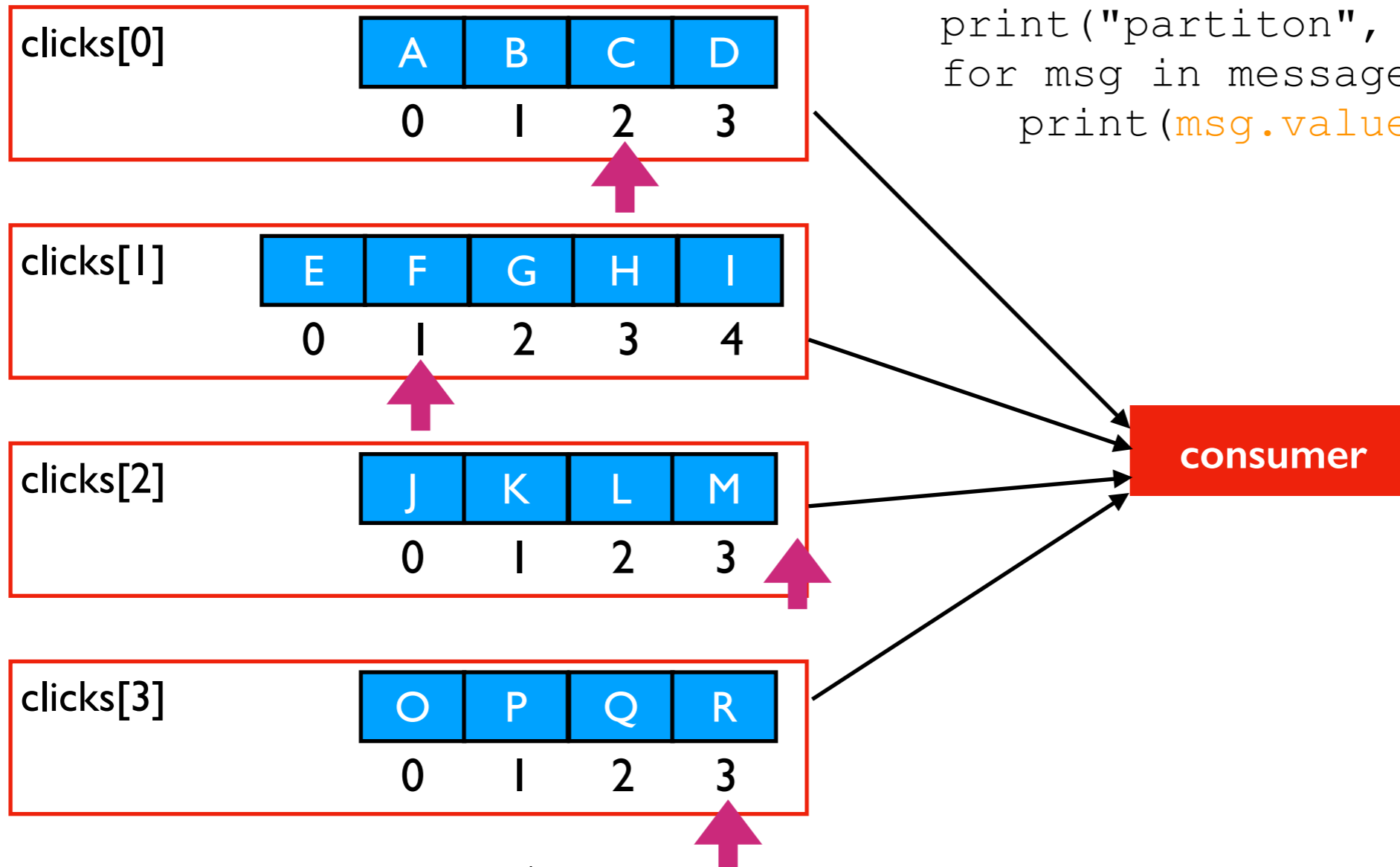
- producer rotates between partitions
- called "round robin" policy

case 2: message has key and value

- calculate partition, for example:
 $\text{hash}(\text{key}) \% \text{partition_count}$
- same keys will go to the same partition
- can plug in alternative partitioning schemes

Consumers: Read Offsets

Topic Partitions



```
batch = consumer.poll(1000)
for topic, messages in batch.items():
    print("partiton", topic.partition)
    for msg in messages:
        print(msg.value)
```

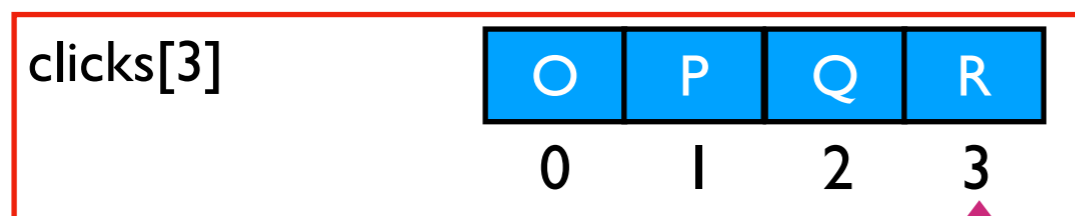
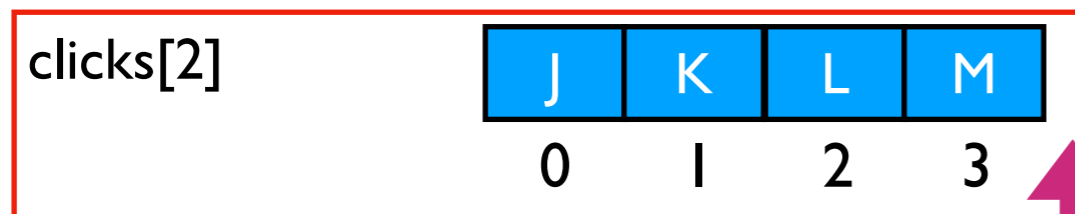
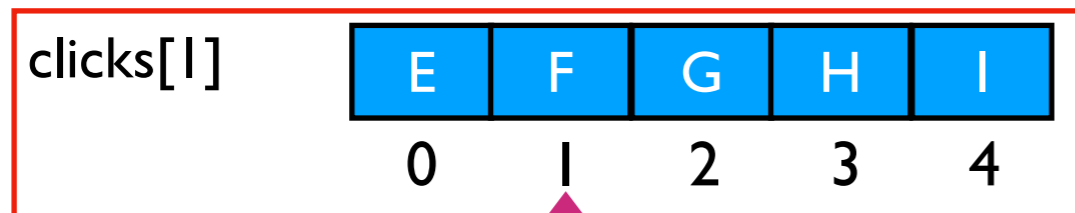
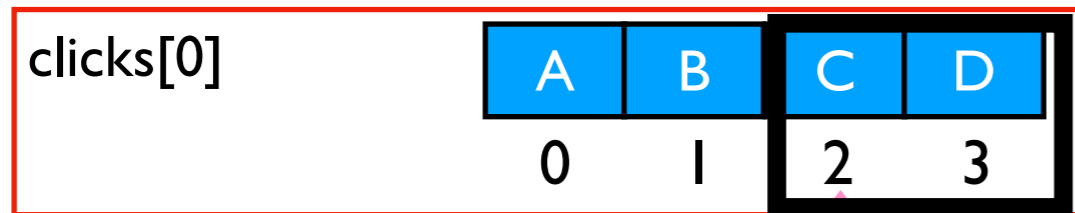
	offset
clicks[0]	2
clicks[1]	1
clicks[2]	4
clicks[3]	3

Batches

- poll returns batches (when enough data or timeout)
- batches contain some subset of partitions
- some number of messages in partition, starting at offset

Example 1

Topic Partitions



```
batch = consumer.poll(1000)
for topic, messages in batch.items():
    print("partition", topic.partition)
    for msg in messages:
        print(msg.value)
```



output:
partition 0
b'C'
b'D'

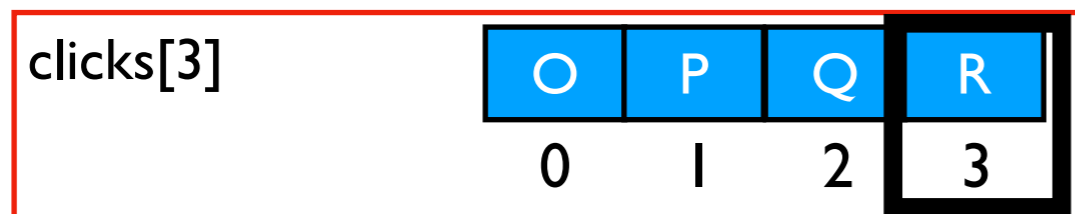
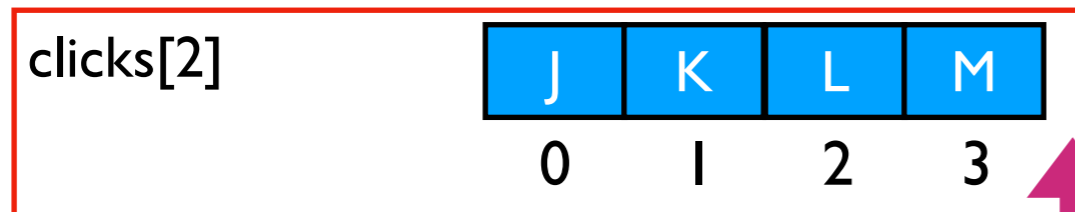
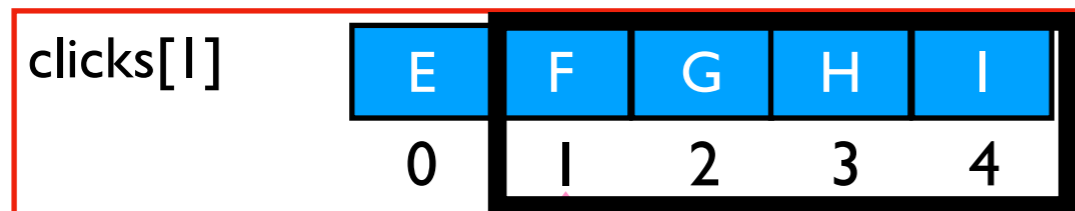
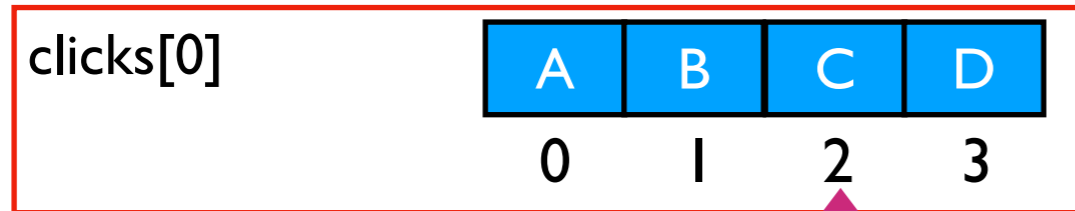
	offset
clicks[0]	4
clicks[1]	1
clicks[2]	4
clicks[3]	3

Batches

- poll returns batches (when enough data or timeout)
- batches contain some subset of partitions
- some number of messages in partition, starting at offset

Example 2

Topic Partitions



```
batch = consumer.poll(1000)
for topic, messages in batch.items():
    print("partition", topic.partition)
    for msg in messages:
        print(msg.value)
```



output:
partition 1
b'F'
b'G'
b'H'
b'I'
partition 3
b'R'

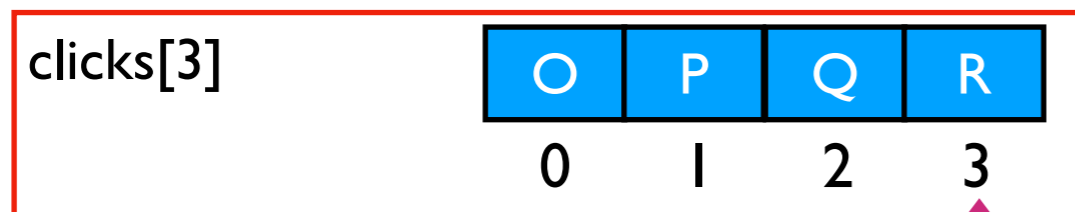
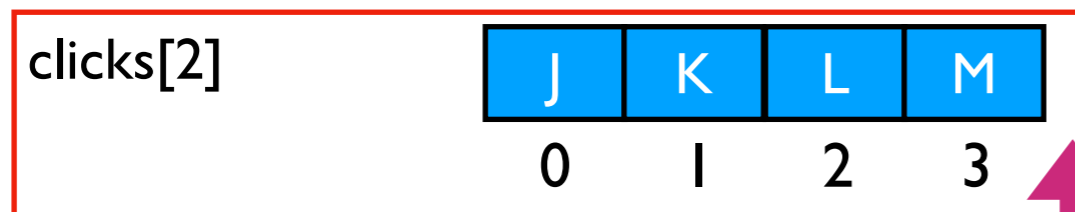
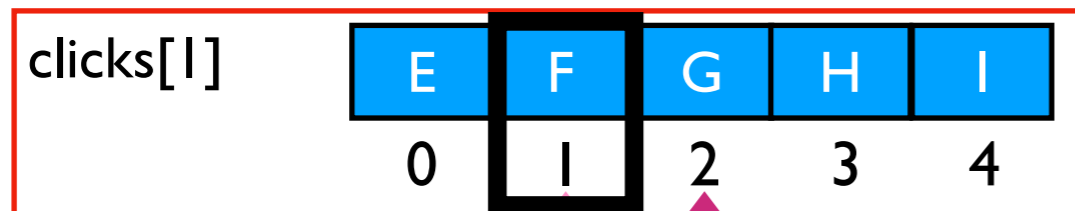
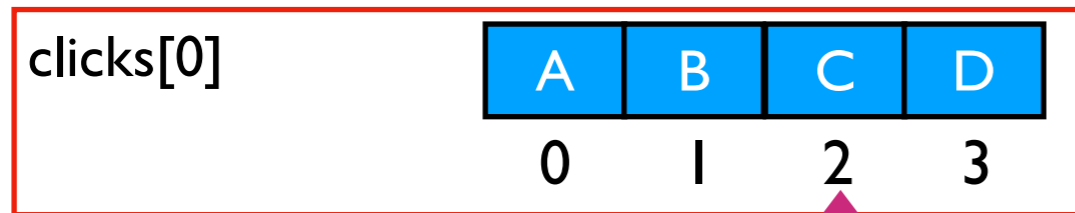
	offset
clicks[0]	2
clicks[1]	5
clicks[2]	4
clicks[3]	4

Batches

- poll returns batches (when enough data or timeout)
- batches contain some subset of partitions
- some number of messages in partition, starting at offset

Example 3

Topic Partitions



```
batch = consumer.poll(1000)
for topic, messages in batch.items():
    print("partition", topic.partition)
    for msg in messages:
        print(msg.value)
```



output:
partition 1
b'F'

	offset
clicks[0]	2
clicks[1]	2
clicks[2]	4
clicks[3]	3

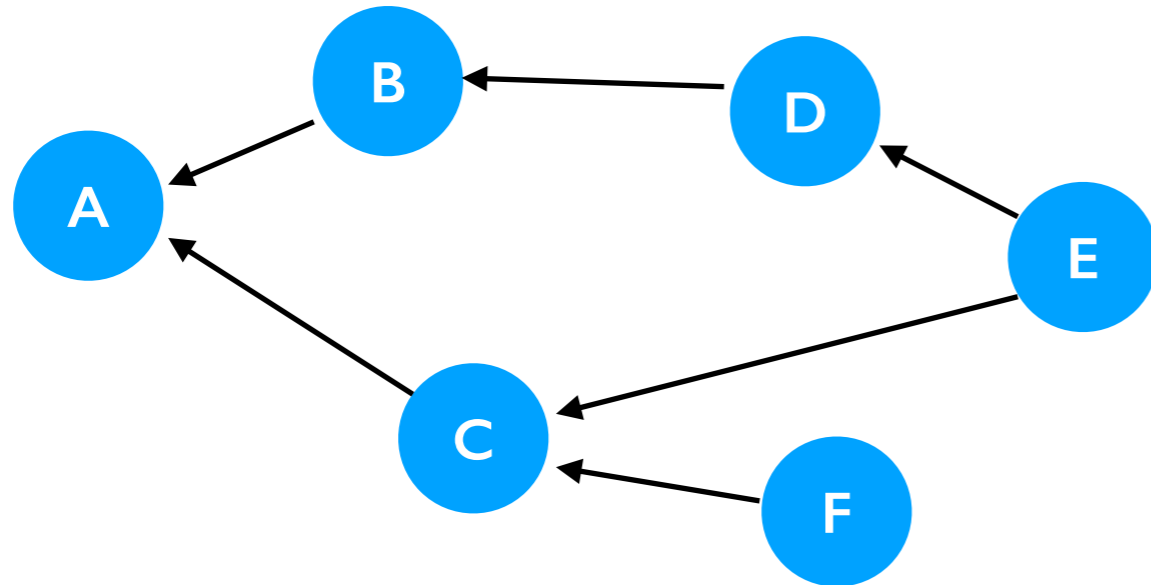
Batches

- poll returns batches (when enough data or timeout)
- batches contain some subset of partitions
- some number of messages in partition, starting at offset

Partially vs. Totally Ordered

Some things are **totally ordered**, like integers. Either $x < y$ or $y \geq x$.

Other things are **partially ordered**, like git commits. Sometimes you can compare, sometimes you can't!



$A < B$ $A < C$ $D < E$...

Can't compare B and C

Can't compare D and F

...

Ordering Kafka Messages

Kafka Messages are **partially ordered**. Messages are consumed from a partition in the order they were written to that partition (no guarantees across topics or across partitions).

If A and B share the same **topic** and **key**, and B was **produced after** A, then:

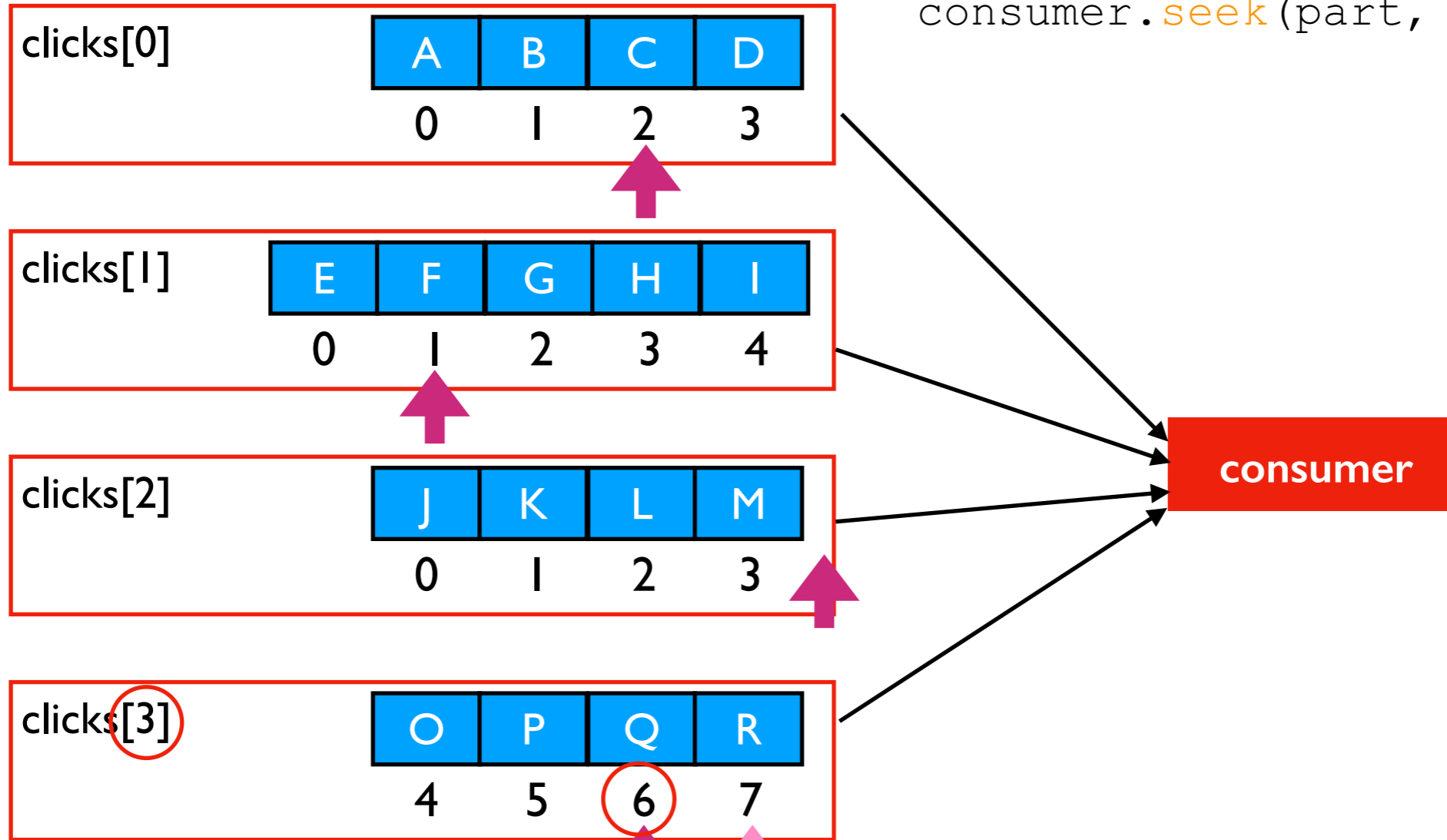
- we say B "happened after" A
- A and B will be in the same partition (assuming partition count is constant)
- each consumer group of the topic will consume A before B

Choose your key carefully! Try to create enough partitions initially and never change it (hash partitioning isn't elastic).

No keys specified => no guarantee about what order messages are consumed.

Seek to an Offset

Topic Partitions



```
part = TopicPartition("clicks", 3)
offset = 6
consumer.seek(part, offset)
```

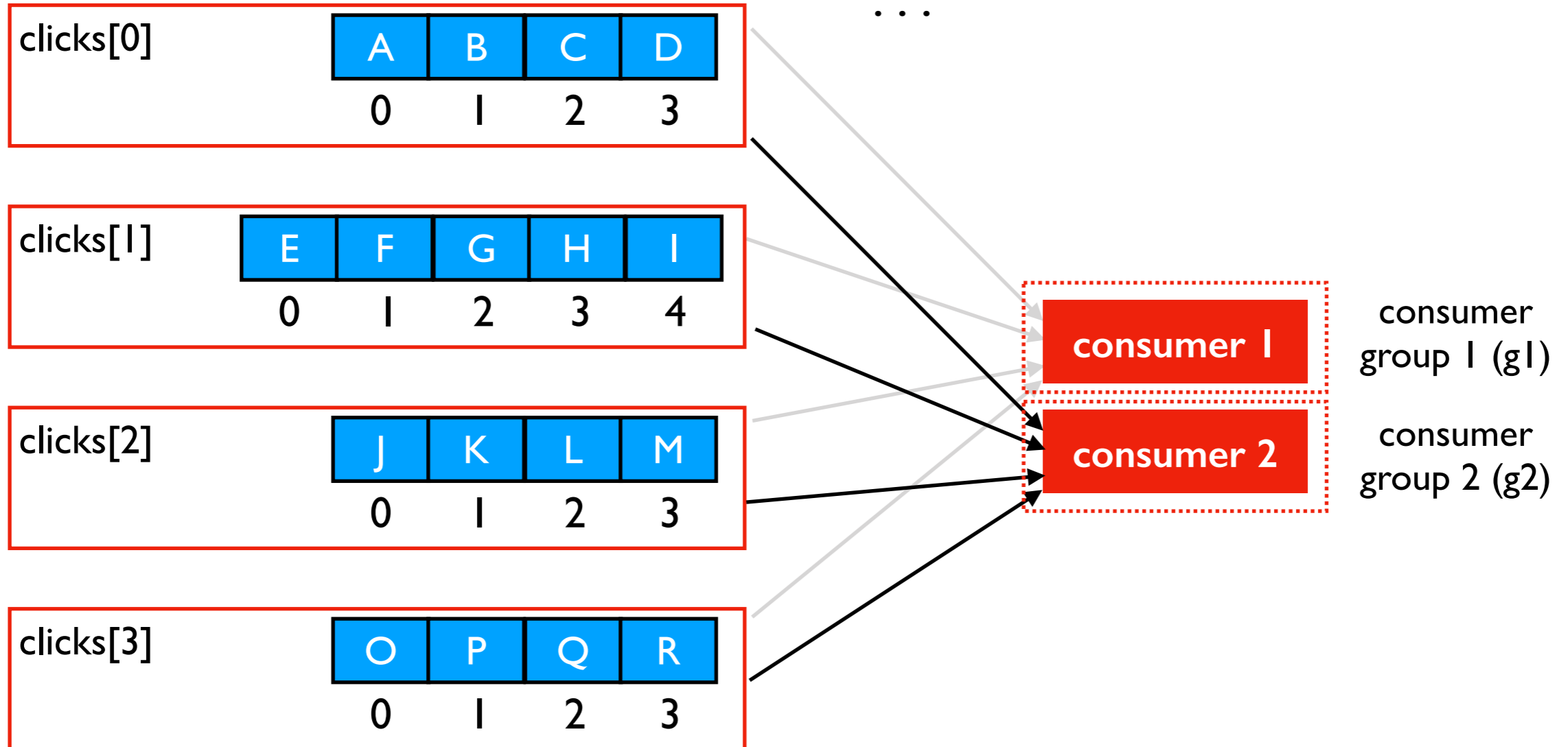
	offset
clicks[0]	2
clicks[1]	1
clicks[2]	4
clicks[3]	7 6

Read pattern

- consumers normally read forward sequentially
- `seek` can jump back (or ahead)
- useful if processing batch failed: just go back and retry

Consumer Groups

Topic Partitions



```
c = KafkaConsumer("clicks",  
                  group_id="g1",  
                  ...)  
batch = c.poll(1000)  
...
```

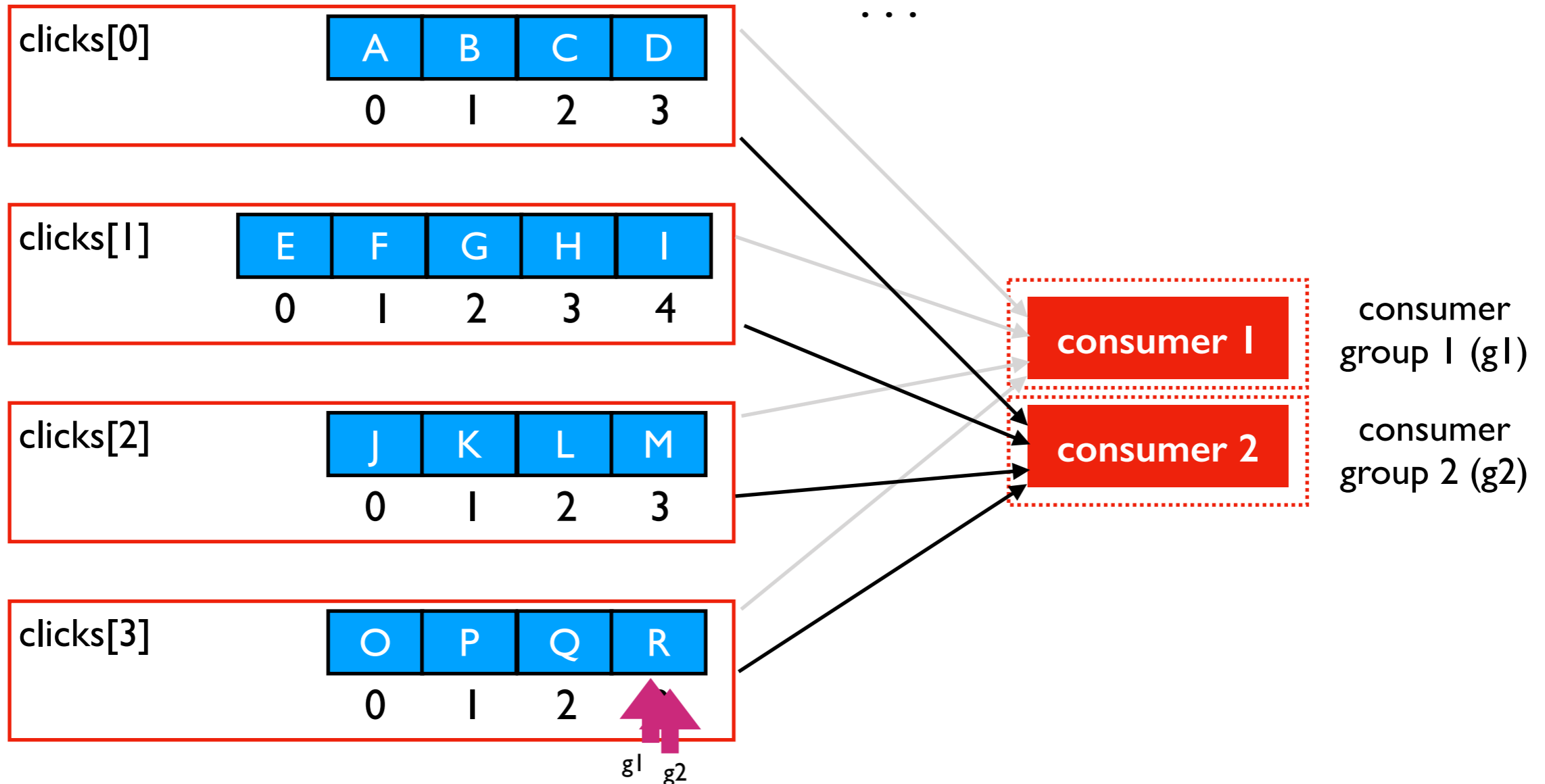
	g1 offsets	g2 offsets
clicks[0]	2	3
clicks[1]	1	2
clicks[2]	4	4
clicks[3]	3	3

Groups

- different applications might operate independently
- they should ALL get a chance to consume messages
- need offsets for each topic/partition/consumer group combination

Consumer Groups

Topic Partitions



```
c = KafkaConsumer("clicks",
                  group_id="g1",
                  ...)
batch = c.poll(1000)
...
```

	g1 offsets	g2 offsets
clicks[0]	2	3
clicks[1]	1	2
clicks[2]	4	4
clicks[3]	3	3

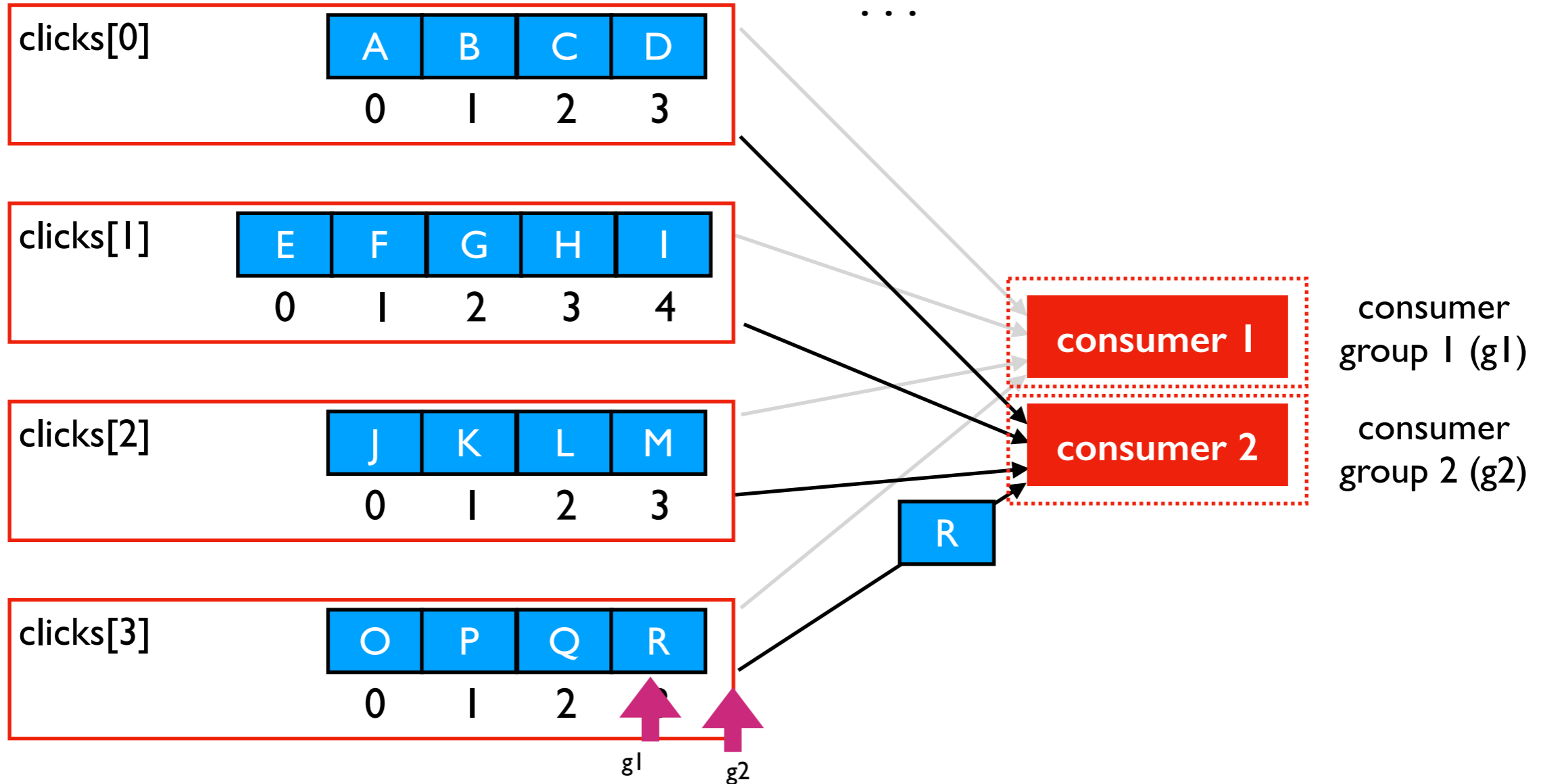
Groups

- different applications might operate independently
- they should ALL get a chance to consume messages
- need offsets for each topic/partition/consumer group combination

Consumer Groups

```
c = KafkaConsumer("clicks",
                  group_id="g1",
                  ...)
batch = c.poll(1000)
...
```

Topic Partitions



	g1 offsets	g2 offsets
clicks[0]	2	3
clicks[1]	1	2
clicks[2]	4	4
clicks[3]	3	4

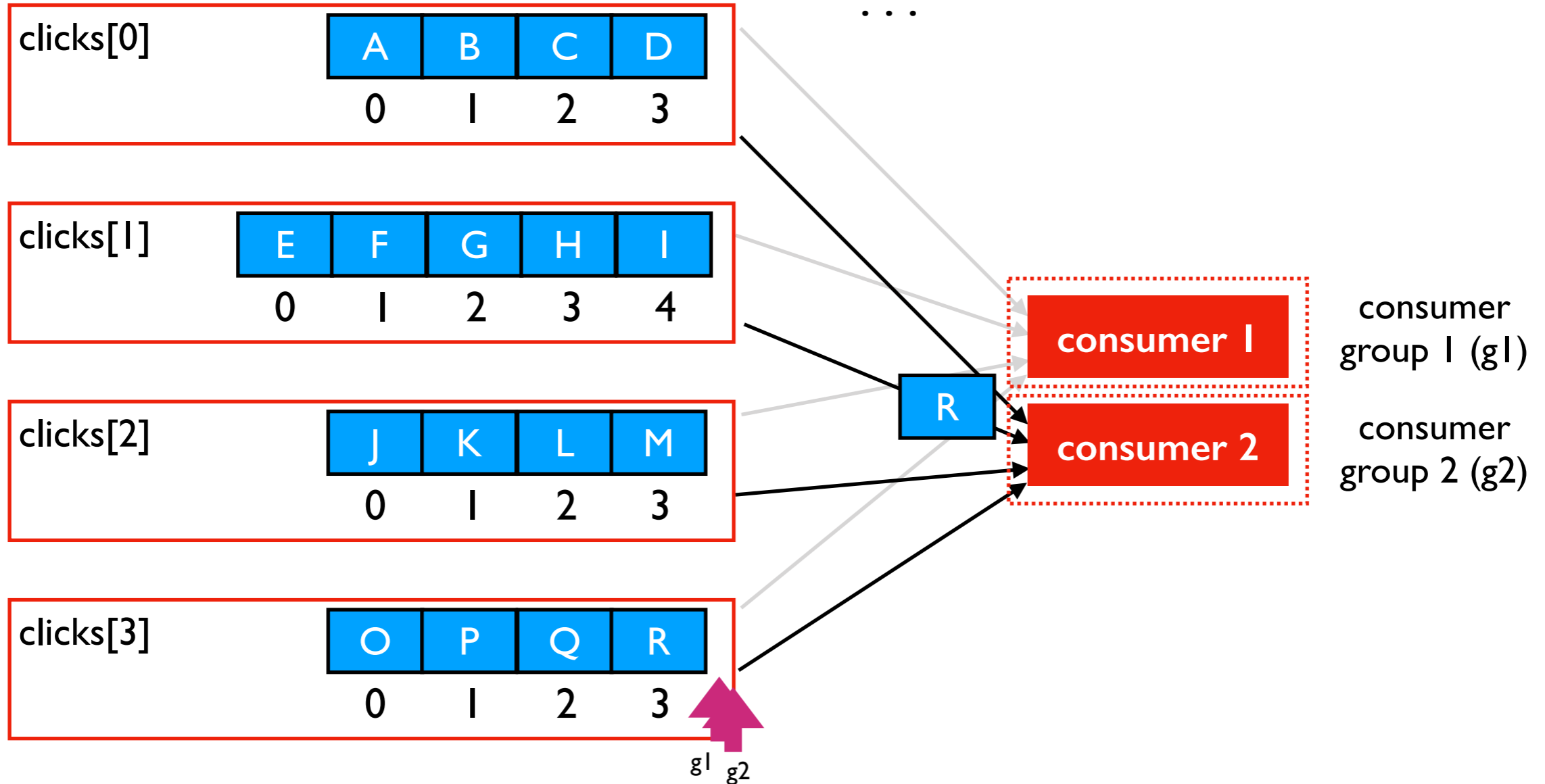
Groups

- different applications might operate independently
- they should ALL get a chance to consume messages
- need offsets for each topic/partition/consumer group combination

Consumer Groups

```
c = KafkaConsumer("clicks",
                  group_id="g1",
                  ...)
batch = c.poll(1000)
...
```

Topic Partitions



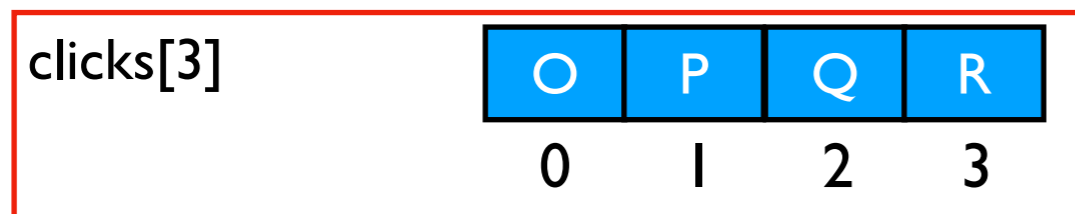
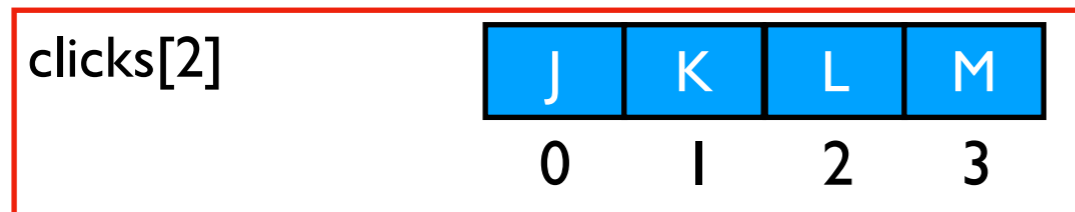
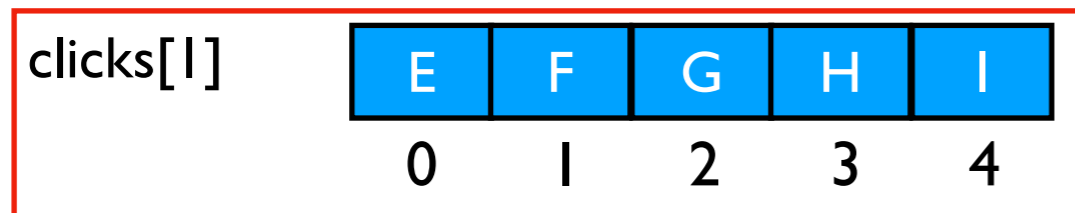
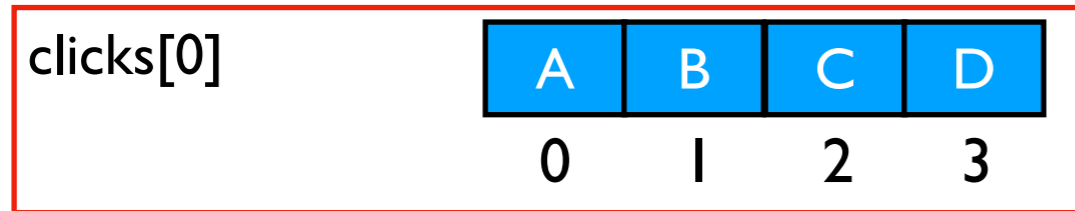
	g1 offsets	g2 offsets
clicks[0]	2	3
clicks[1]	1	2
clicks[2]	4	4
clicks[3]	4	4

Groups

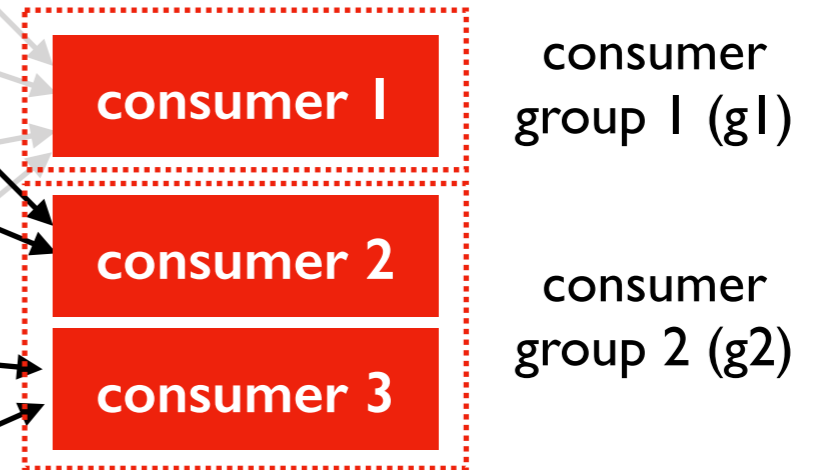
- different applications might operate independently
- they should ALL get a chance to consume messages
- need offsets for each topic/partition/consumer group combination

Partition Assignment: Manual

Topic Partitions



```
tp0 = TopicPartition("clicks", 0)
...
consumer2.assign([tp0, tp1])
consumer3.assign([tp2, tp3])
```

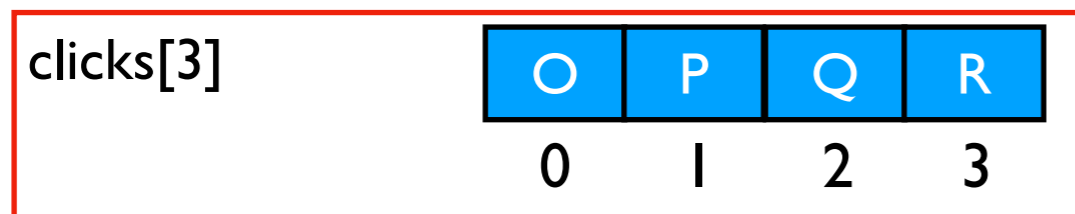
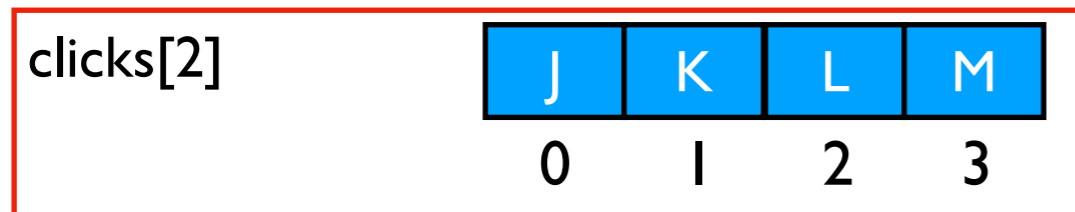
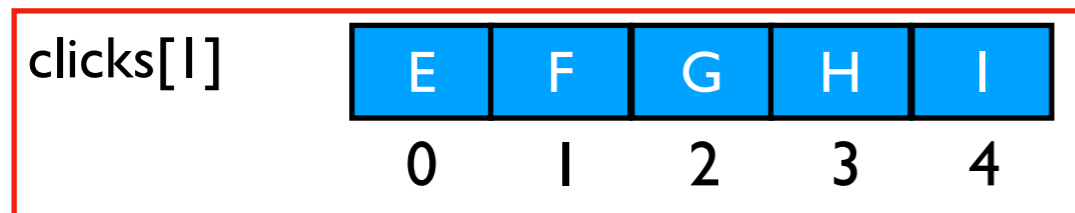
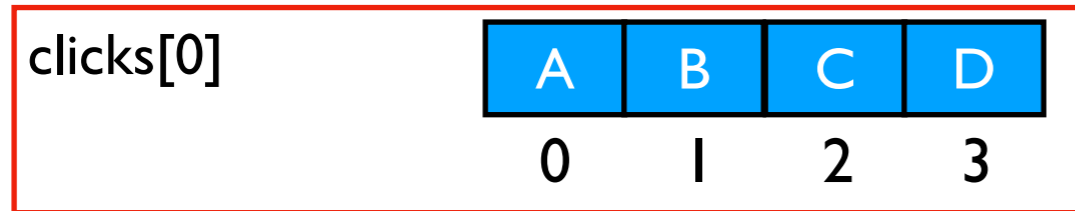


	partition offsets, per group	
	g1 offsets	g2 offsets
clicks[0]	2	3
clicks[1]	1	2
clicks[2]	4	4
clicks[3]	4	4

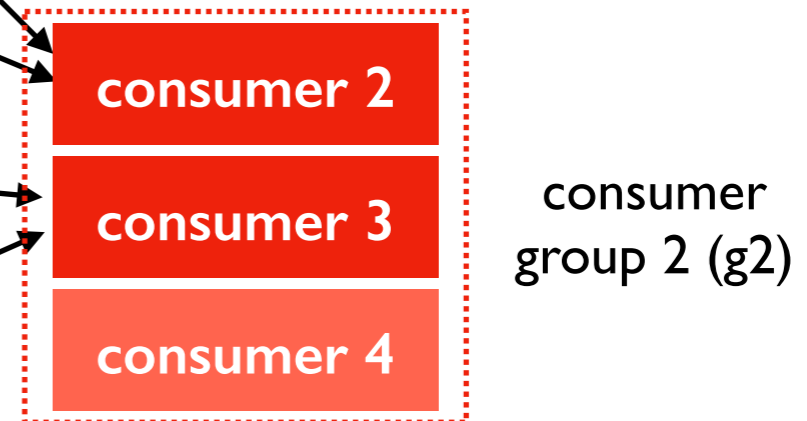
	partition assignments, per group	
	g1 assignment	g2 assignment
clicks[0]	consumer 1	consumer 2
clicks[1]	consumer 1	consumer 2
clicks[2]	consumer 1	consumer 3
clicks[3]	consumer 1	consumer 3

Partition Assignment: Automatic

Topic Partitions



```
# consumer 3: subscribed to clicks
while True:
    batch = consumer.poll(1000)
    for topic, msgs in batch.items():
        for msg in msgs:
            ...
consumer.close()
```



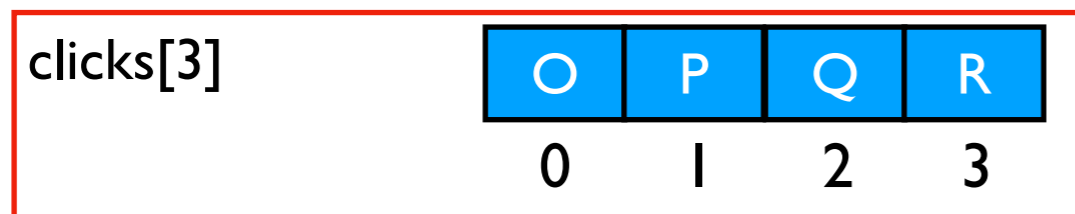
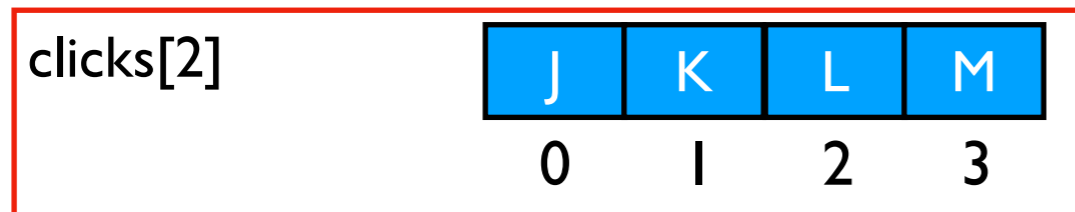
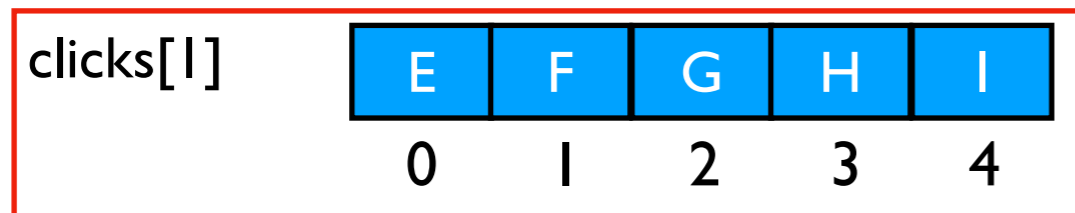
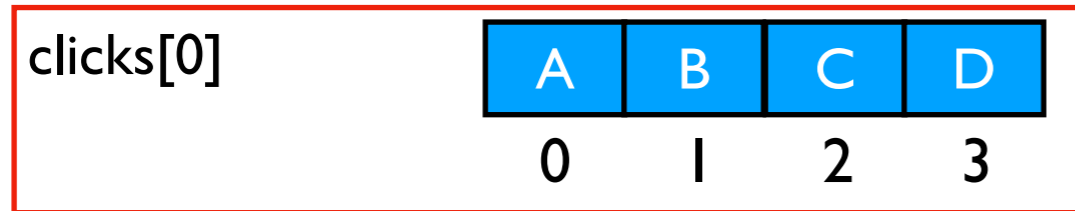
Assignment and re-assignment

- by default, consumers are automatically assigned partitions when they start polling
- **challenge:** Kafka shouldn't re-assign a partition in the middle of a batch (might double process messages)

	partition assignments, per group	
	g1 assignment	g2 assignment
clicks[0]	consumer 1	consumer 2
clicks[1]	consumer 1	consumer 2
clicks[2]	consumer 1	consumer 3
clicks[3]	consumer 1	consumer 3

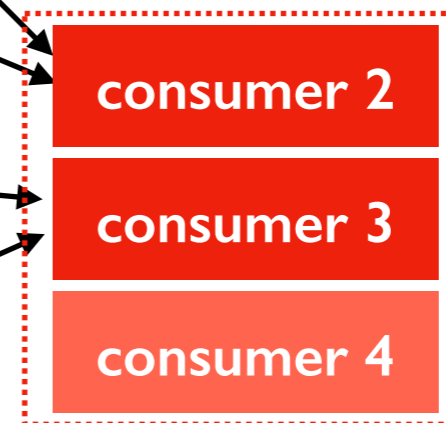
Partition Assignment: Automatic

Topic Partitions



```
# consumer 3: subscribed to clicks
while True:
    batch = consumer.poll(1000)
    for topic, msgs in batch.items():
        for msg in msgs:
            ...
consumer.close()
```

best to take away a partition at these points



consumer group 2 (g2)

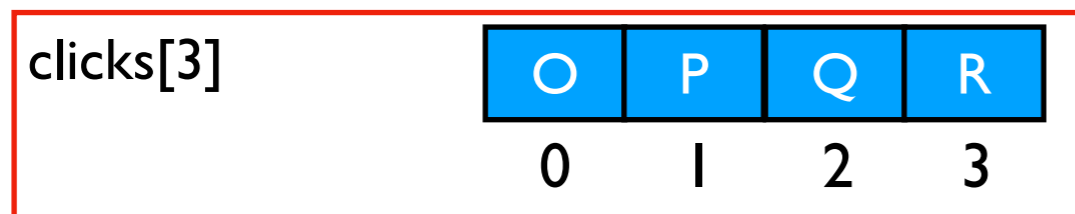
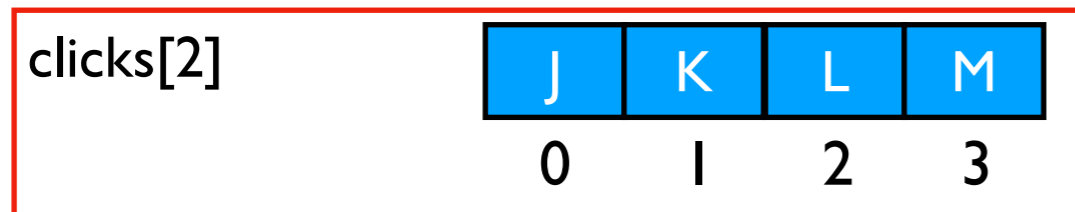
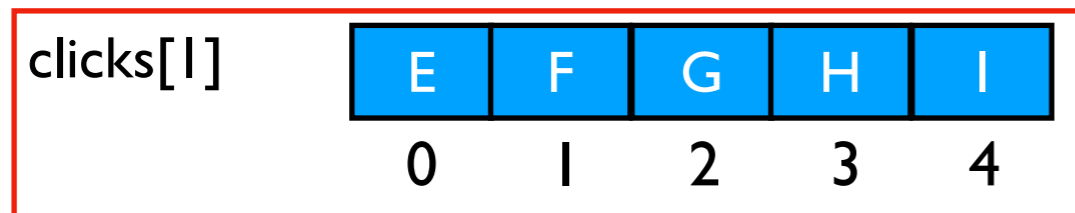
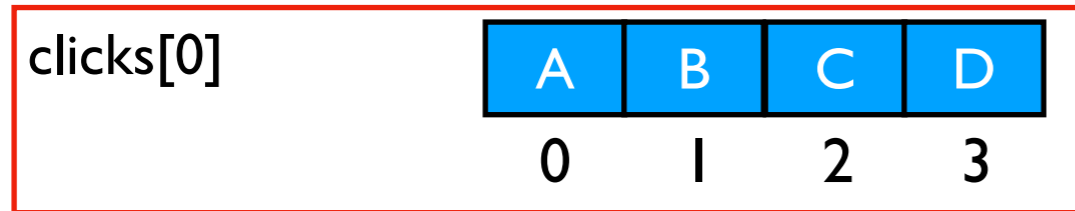
Assignment and re-assignment

- by default, consumers are automatically assigned partitions when they start polling
- **challenge:** Kafka shouldn't re-assign a partition in the middle of a batch (might double process messages)

	partition assignments, per group	
	g1 assignment	g2 assignment
clicks[0]	consumer 1	consumer 2
clicks[1]	consumer 1	consumer 2
clicks[2]	consumer 1	consumer 3
clicks[3]	consumer 1	consumer 3

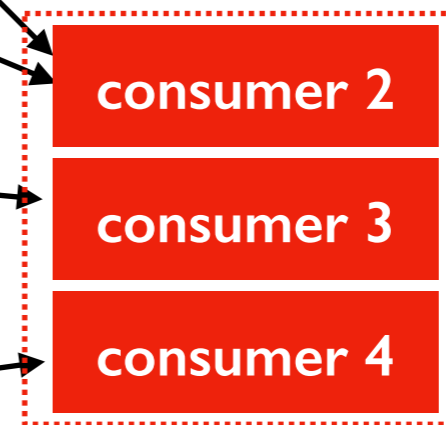
Partition Assignment: Automatic

Topic Partitions



```
# consumer 3: subscribed to clicks
while True:
    batch = consumer.poll(1000)
    for topic, msgs in batch.items():
        for msg in msgs:
            ...
consumer.close()
```

best to take away a partition at these points



consumer group 2 (g2)

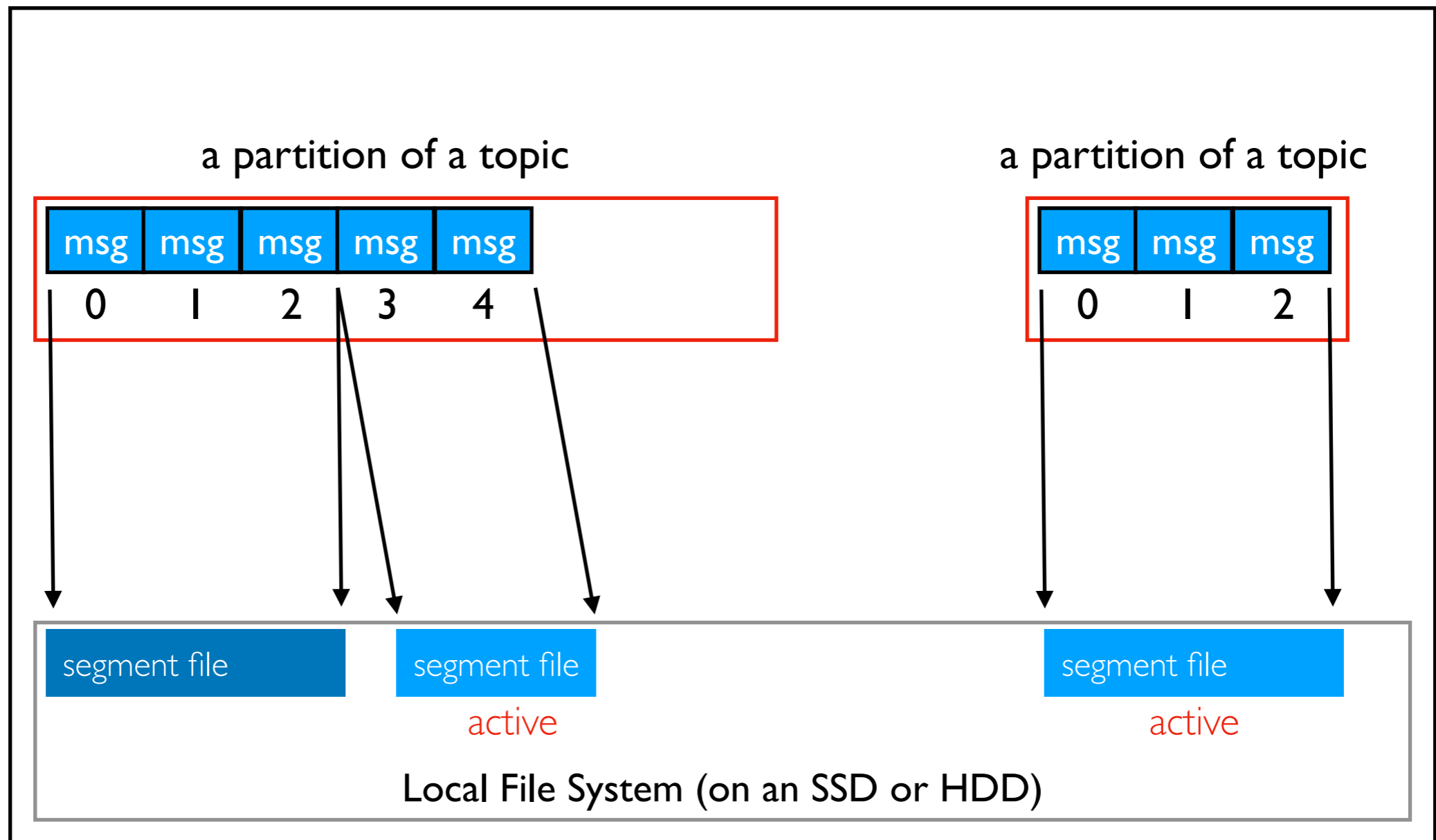
Assignment and re-assignment

- by default, consumers are automatically assigned partitions when they start polling
- **challenge:** Kafka shouldn't re-assign a partition in the middle of a batch (might double process messages)

	partition assignments, per group	
	g1 assignment	g2 assignment
clicks[0]	consumer 1	consumer 2
clicks[1]	consumer 1	consumer 2
clicks[2]	consumer 1	consumer 3
clicks[3]	consumer 1	consumer 4

Segment Files: Log Rollover and Deletion

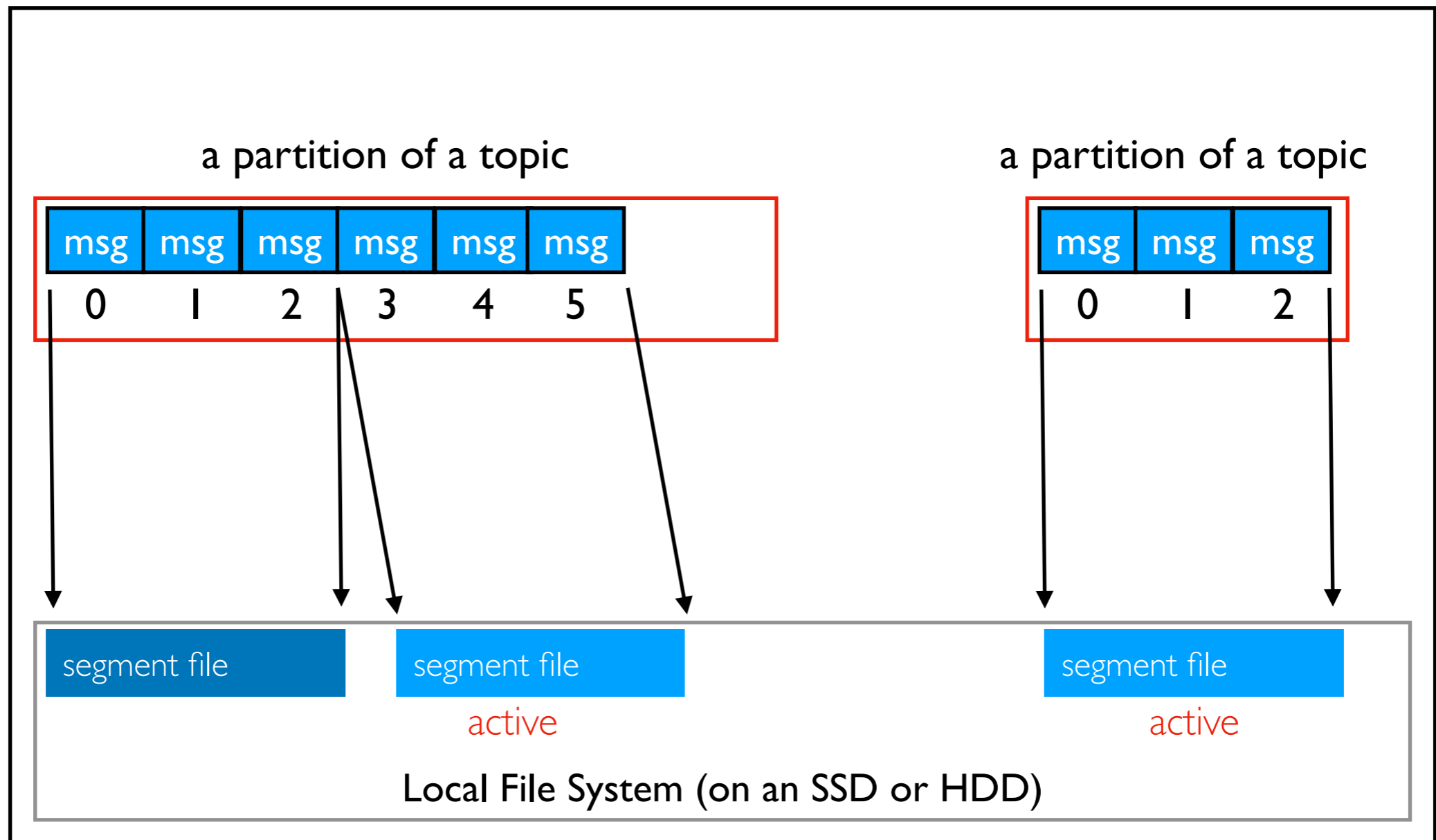
broker server



- partitions are divided into consecutive regions and saved in **segment files**
- all new data is sequentially written to the end of an **active segment**

Segment Files: Log Rollover and Deletion

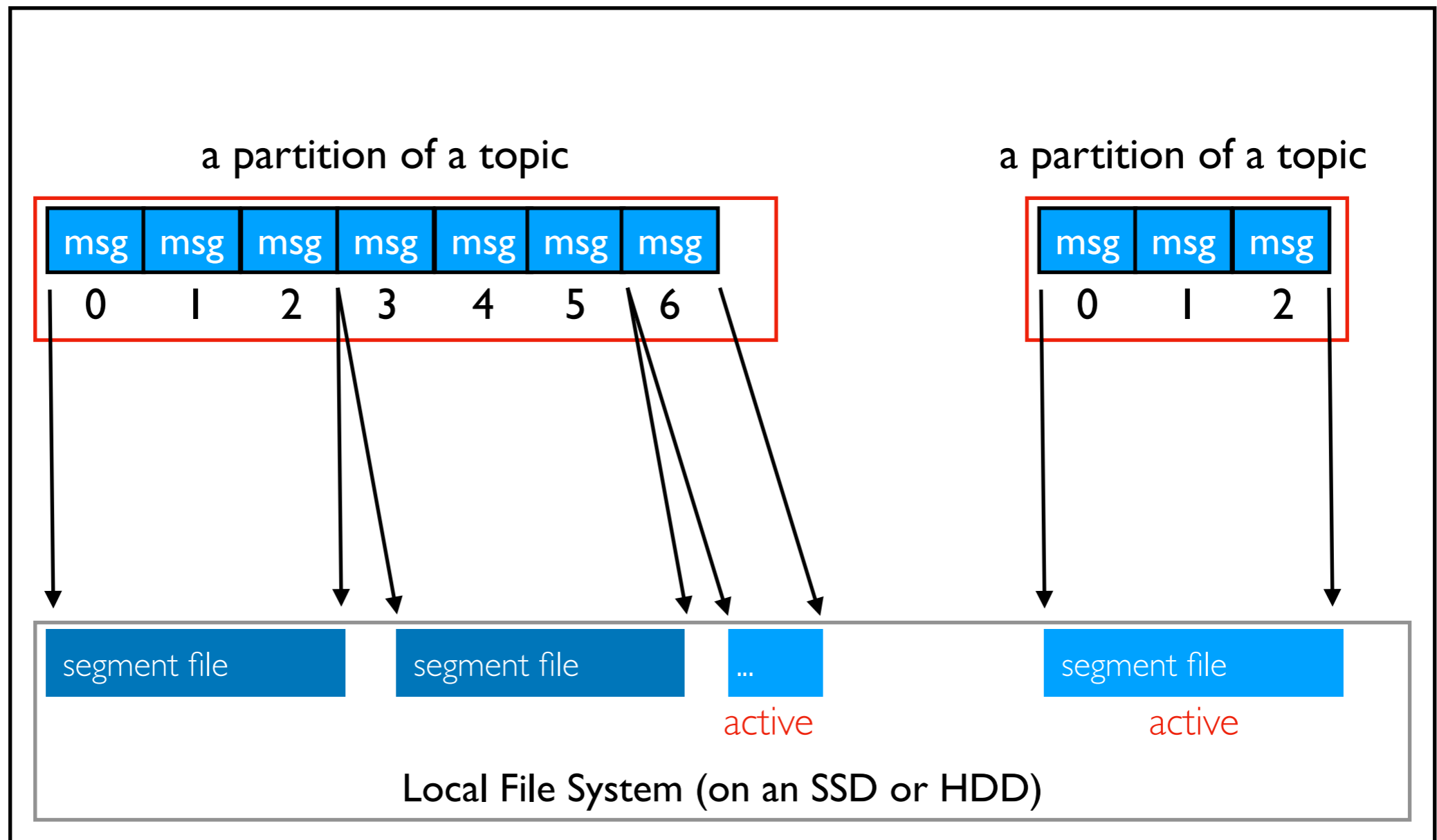
broker server



- partitions are divided into consecutive regions and saved in **segment files**
- all new data is sequentially written to the end of an **active segment**

Segment Files: Log Rollover and Deletion

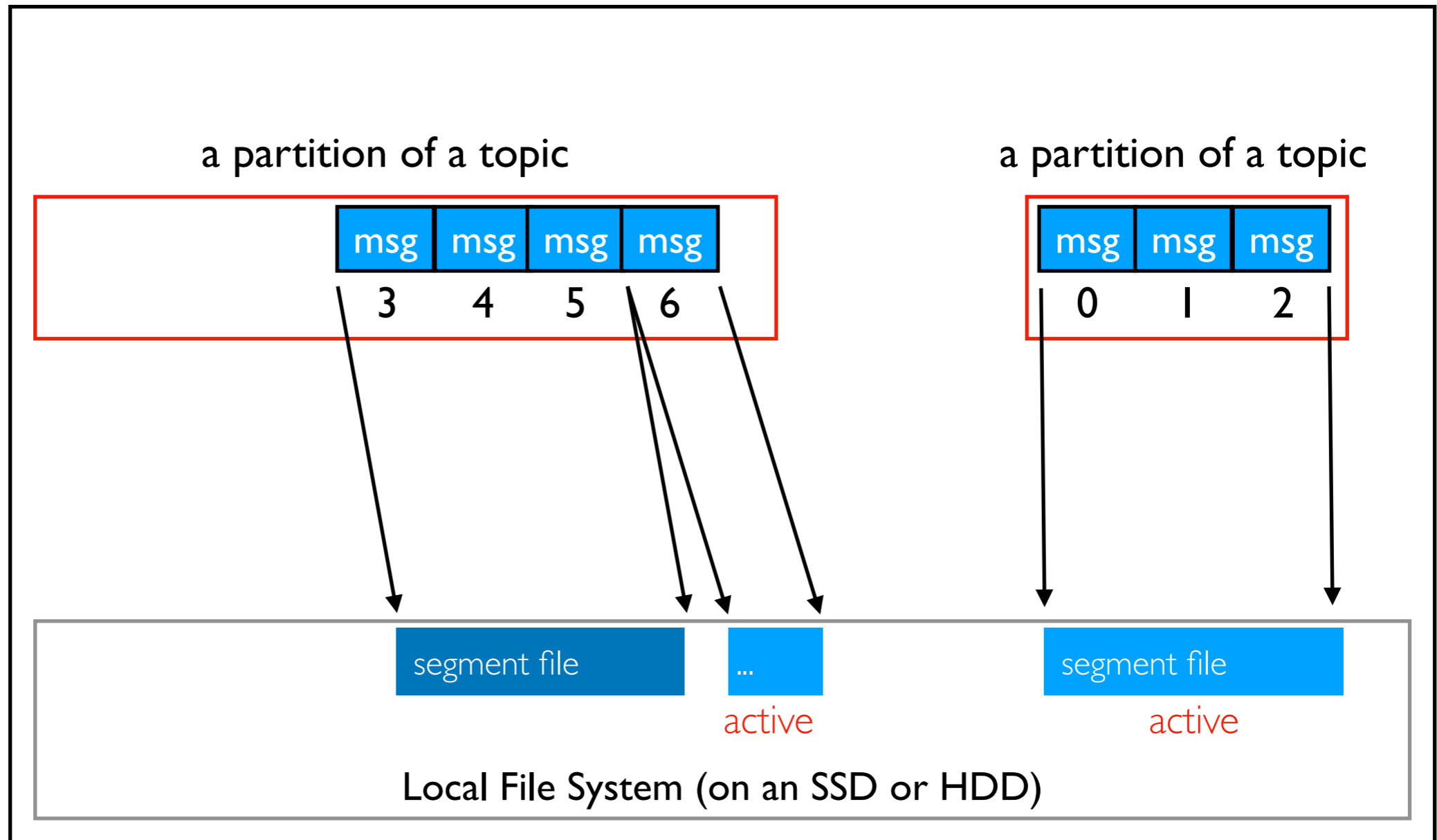
broker server



- **rollover**: current segment is finalized (no more changes)
- new segment is created and becomes active

Segment Files: Log Rollover and Deletion

broker server



- **deletion**: old segment is deleted
- always starts from smallest offset
- active segment is NEVER deleted

Log Policy

Rollover and retention policies are configurable in Kafka.

Rollover

- setting 1: max segment age (`log.roll.hours=7` day by default)
- setting 2: max segment size (`log.segment.bytes=1GB` by default)
- rollover happens when segment gets too big or too old (whichever happens first)

Retention/Deletion

- setting 1: log age cutoff (`log.retention.hours=7` days by default)
- setting 2: log size cutoff (`log.retention.bytes=disabled` by default)
- deletion happens on oldest segment when log is too big or has records too old
- note: age cutoff applies to newest messages in a segment, so there will probably be some older ones in the same segment past the cutoff. *Not useful for legal compliance with data retention laws.*

TopHat