[544] Kafka Streaming

Meenakshi Syamkumar

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)

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




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

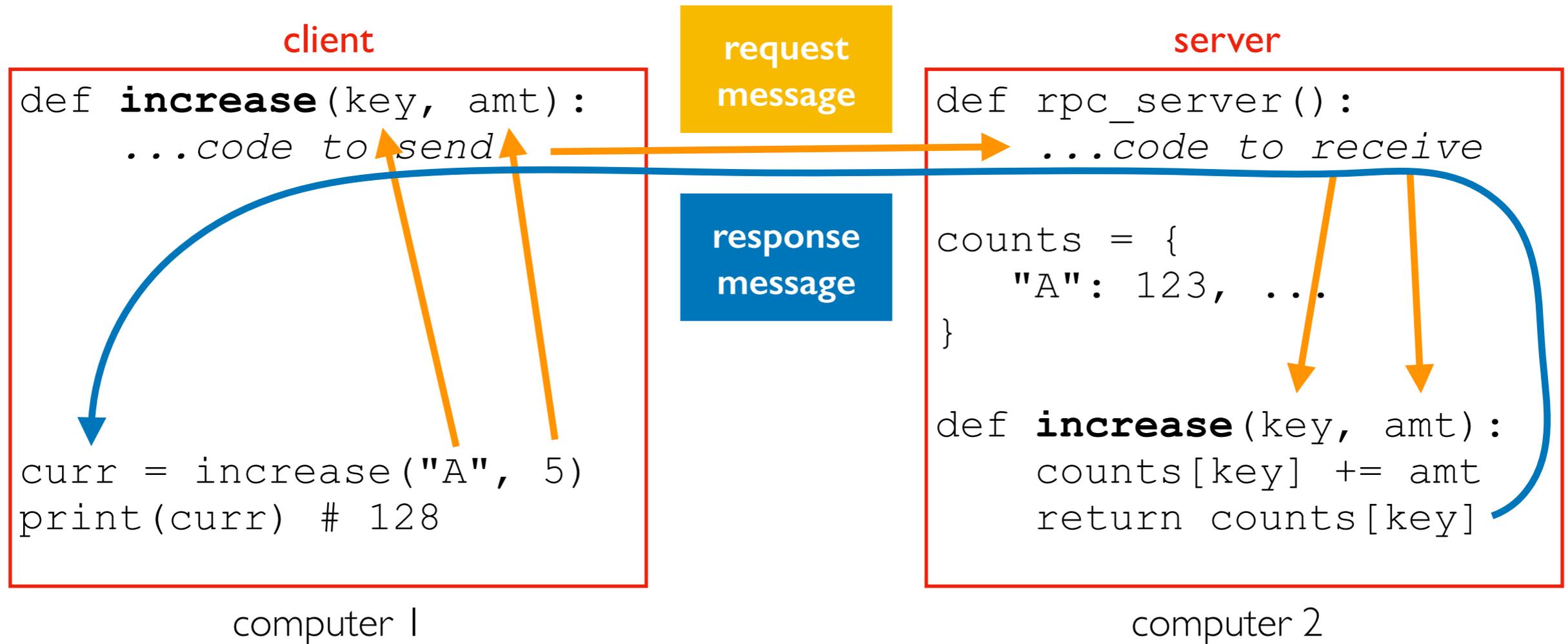computer 1

```
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)

client

request message

server

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



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

response message

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

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

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

computer 1

computer 2

Serialization/Deserialization

client

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

serialize   deserialize

deserialize   serialize

```
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
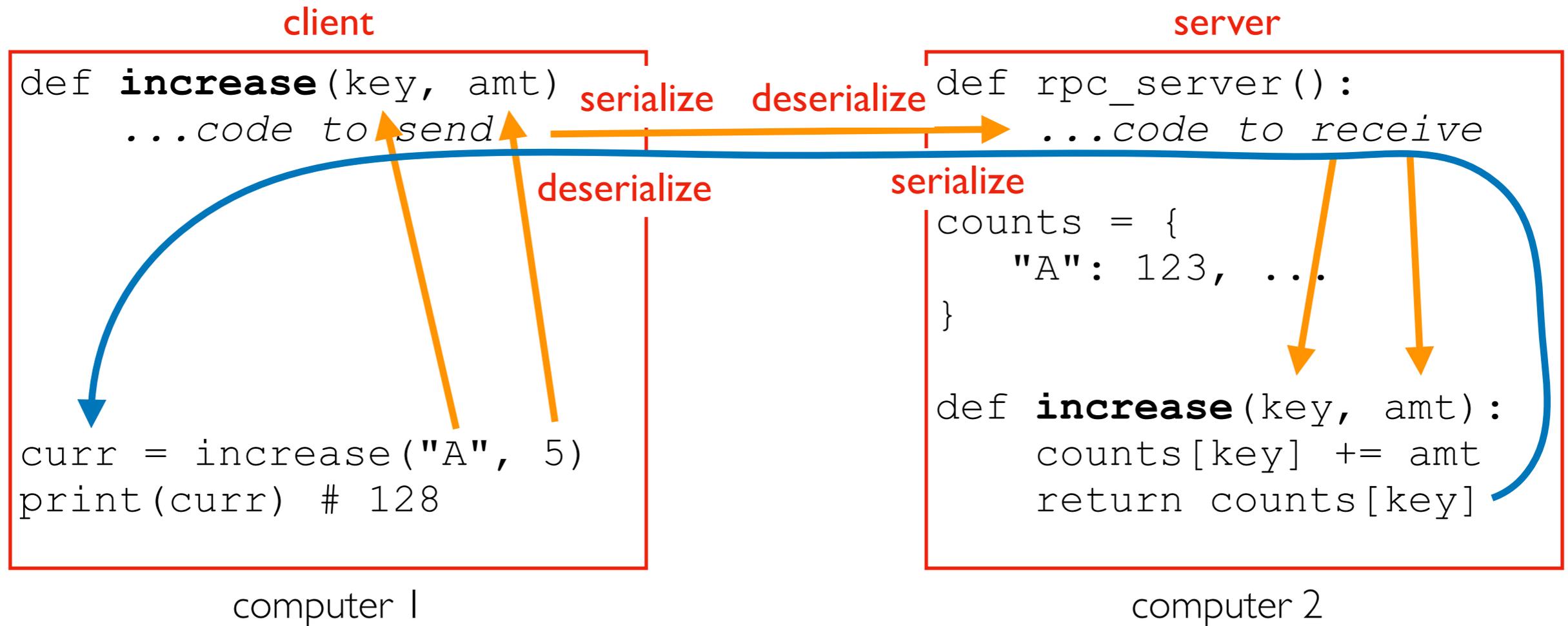
request message

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

response message

```
return val as bytes:
b'128'
```

gRPC uses protocol buffers for wire format

client

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

```
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]
```

serialize

computer 2
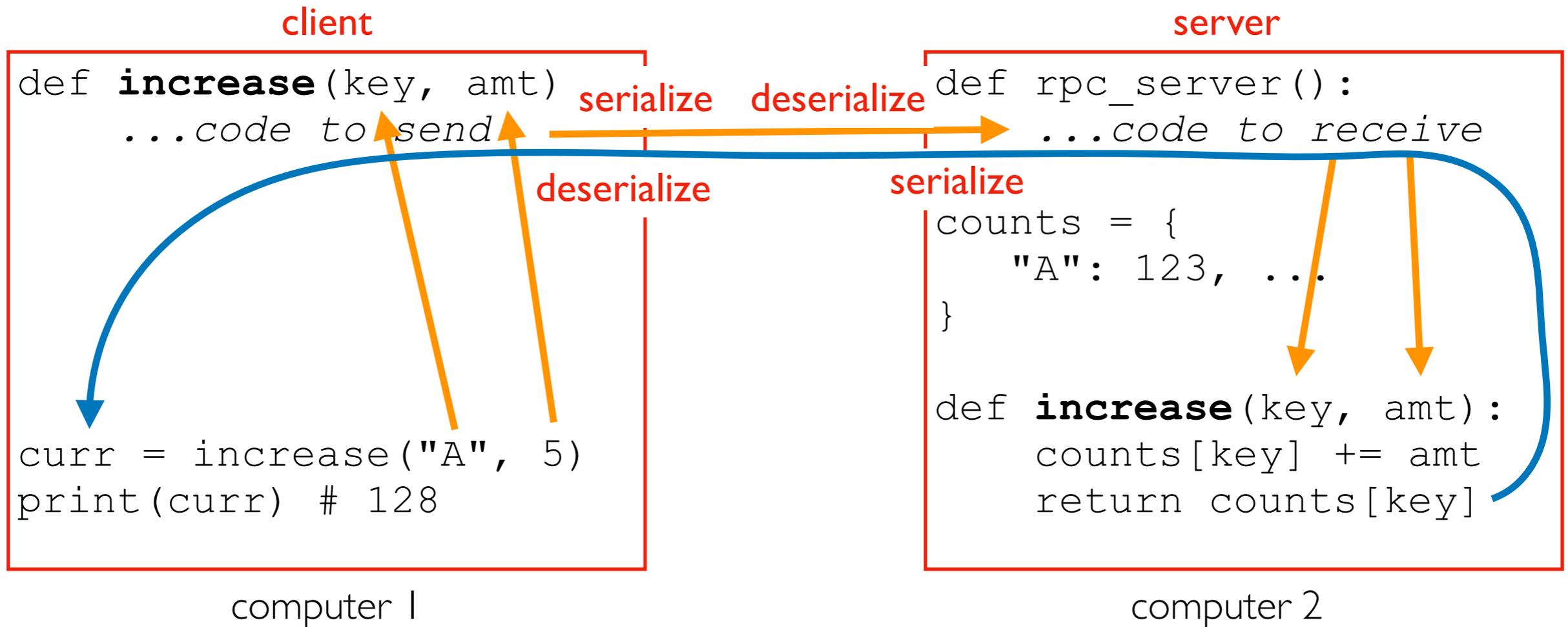
request message

protobuf (args to bytes)
100100010101111
(contains "A" and 5)

response message

protobuf (ret val to bytes)
01000000
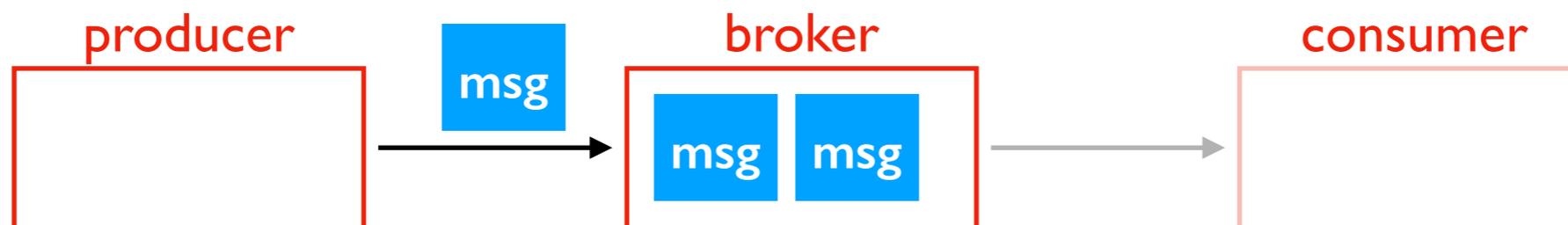(contains 128)

Synchronous vs. Asynchronous Communication

Synchronous

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

client → msg → server
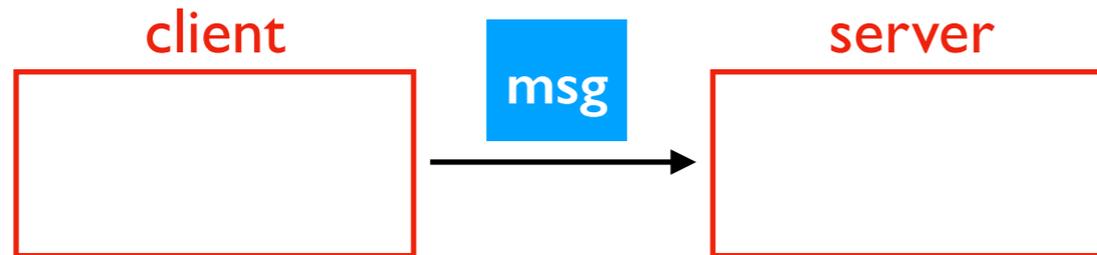
Asynchronous

- one party can send any time, the other can receive later
- examples: email, texting, streaming
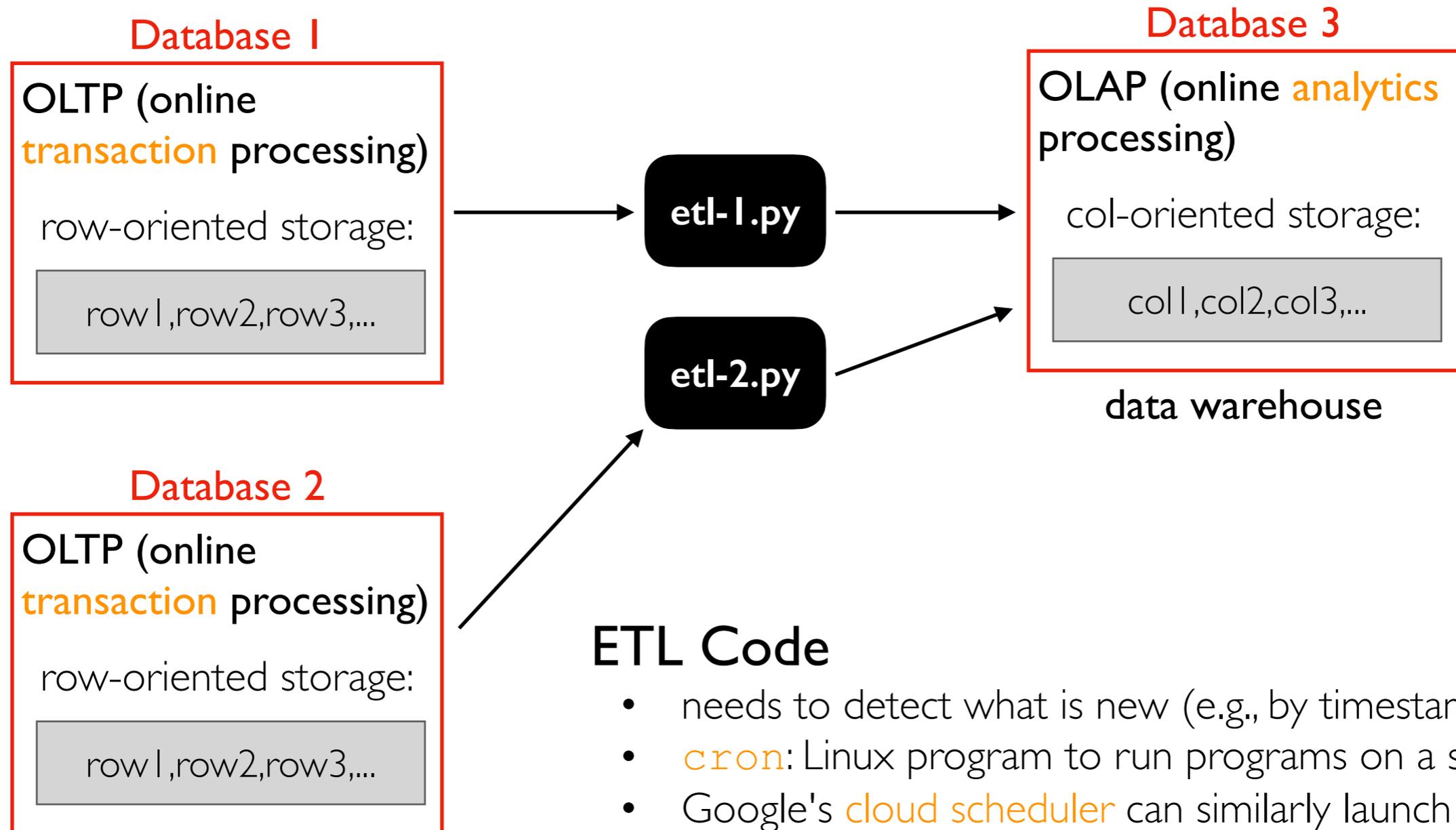
producer → msg → broker [msg msg] → consumer

Outline: Kafka Streaming

Sending/Receiving Messages

ETL (Extract Transform Load)
- Batch
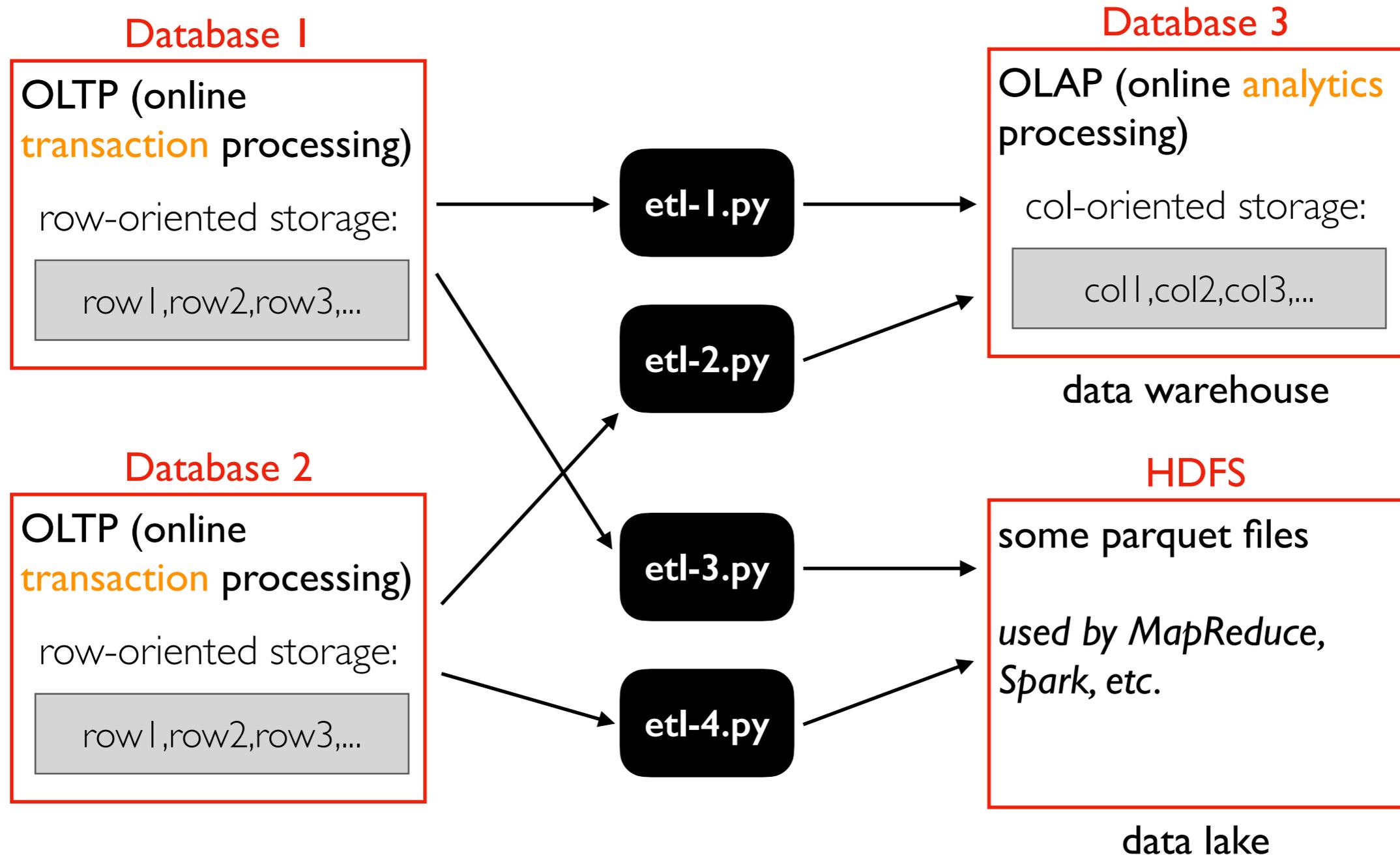- Streaming

Kafka Design

Extract Transform Load (ETL)

Database 1

OLTP (online transaction processing)

row-oriented storage:

row1,row2,row3,...

etl-1.py

etl-2.py

Database 3

OLAP (online analytics processing)

col-oriented storage:

col1,col2,col3,...

data warehouse

Database 2

OLTP (online transaction processing)

row-oriented storage:

row1,row2,row3,...

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)
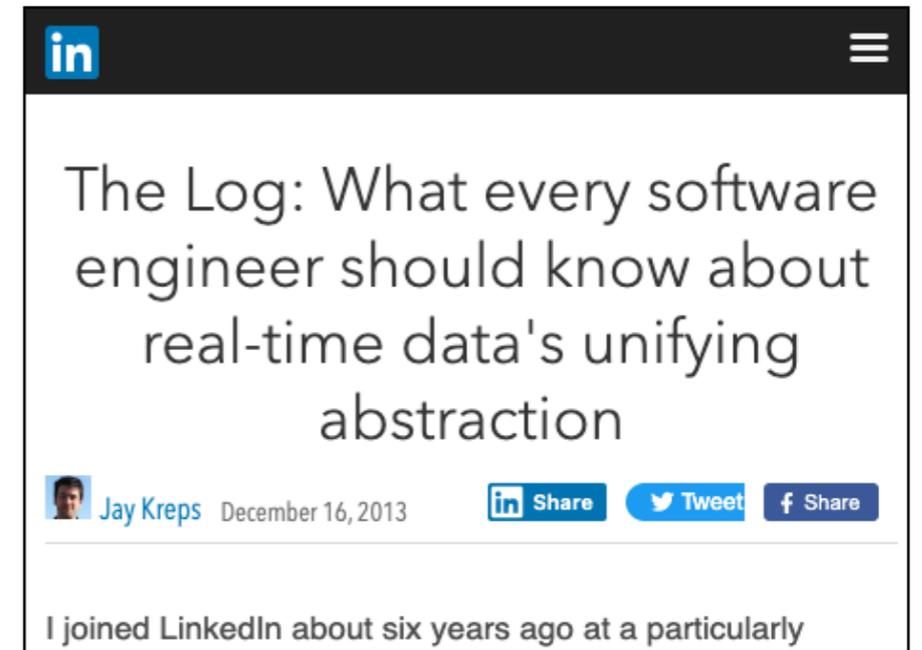
issue 1: data freshness

Extract Transform Load (ETL)

Database 1

OLTP (online transaction processing)

row-oriented storage:

| row1,row2,row3,... |

Database 3

OLAP (online analytics processing)

col-oriented storage:

| col1,col2,col3,... |

data warehouse

etl-1.py

etl-2.py

Database 2

OLTP (online transaction processing)

row-oriented storage:

| row1,row2,row3,... |

etl-3.py

etl-4.py

HDFS

some parquet files

used by MapReduce, Spark, etc.

data lake

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

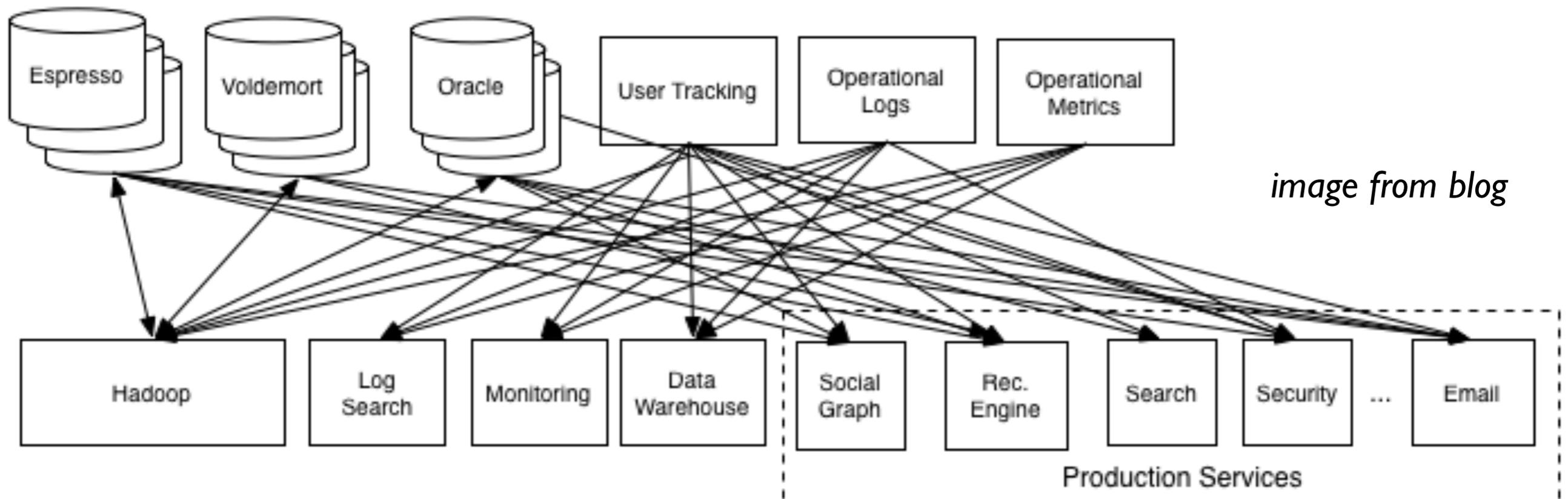**issue 2:** scaling engineering effort

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

The Log: What every software engineer should know about real-time data's unifying abstraction

Jay Kreps December 16, 2013

I joined LinkedIn about six years ago at a particularly

image from blog

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
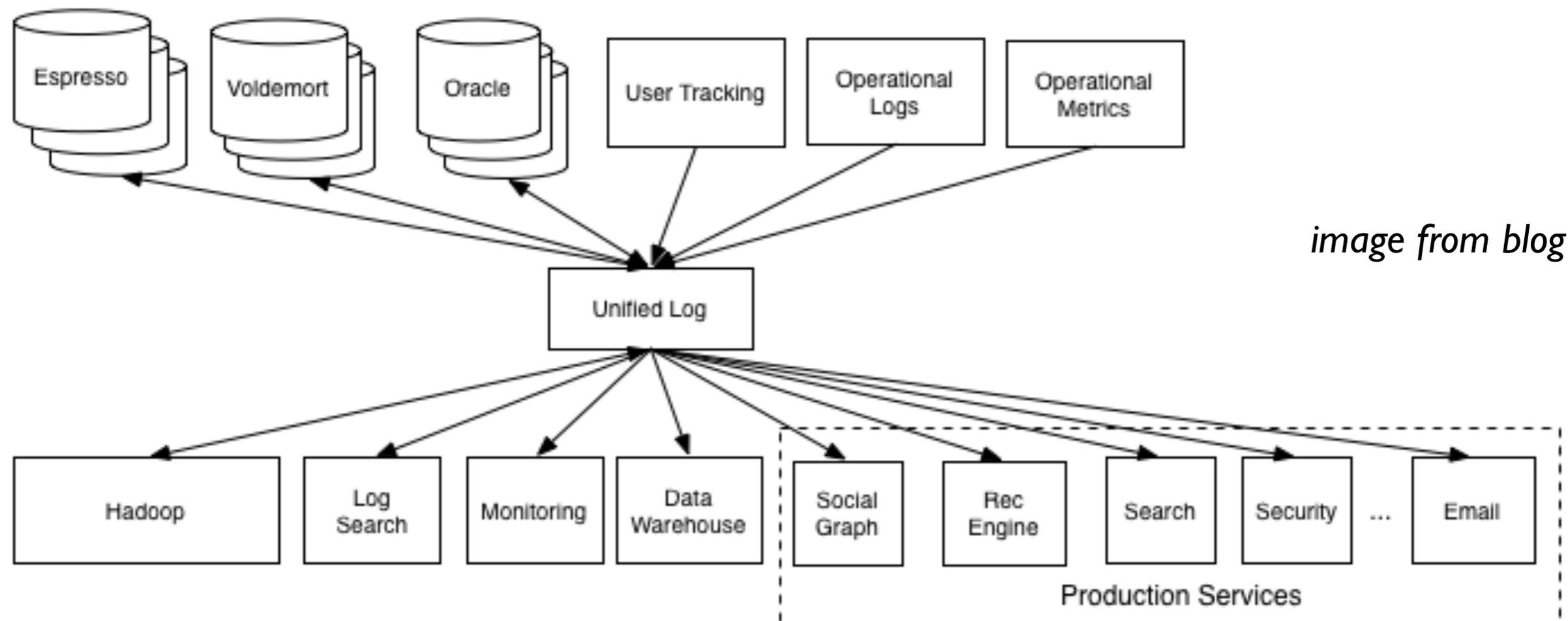(instead of batch jobs with cron)
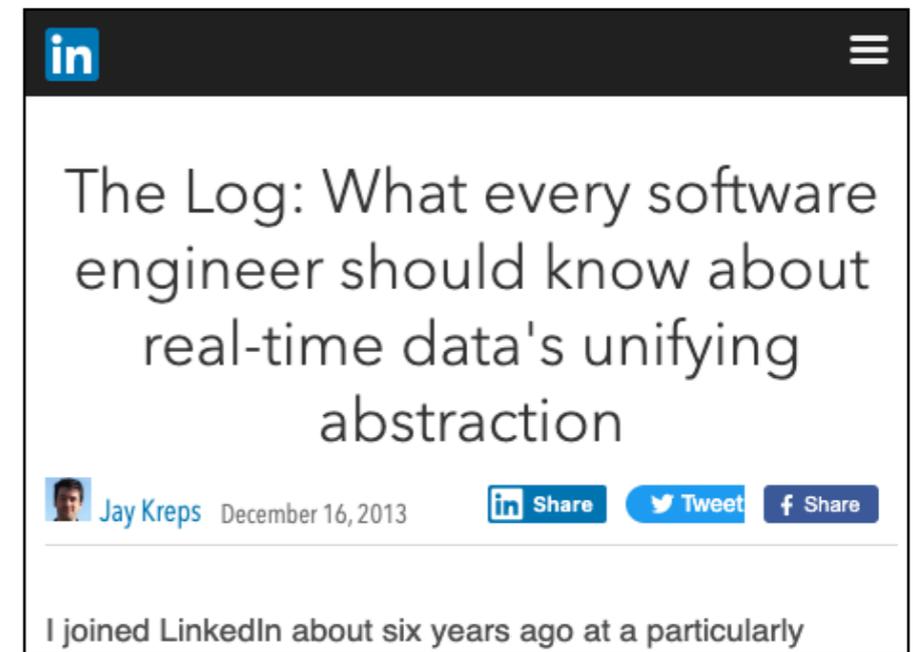
image from blog

Outline: Kafka Streaming

Sending/Receiving Messages

ETL (Extract Transform Load)

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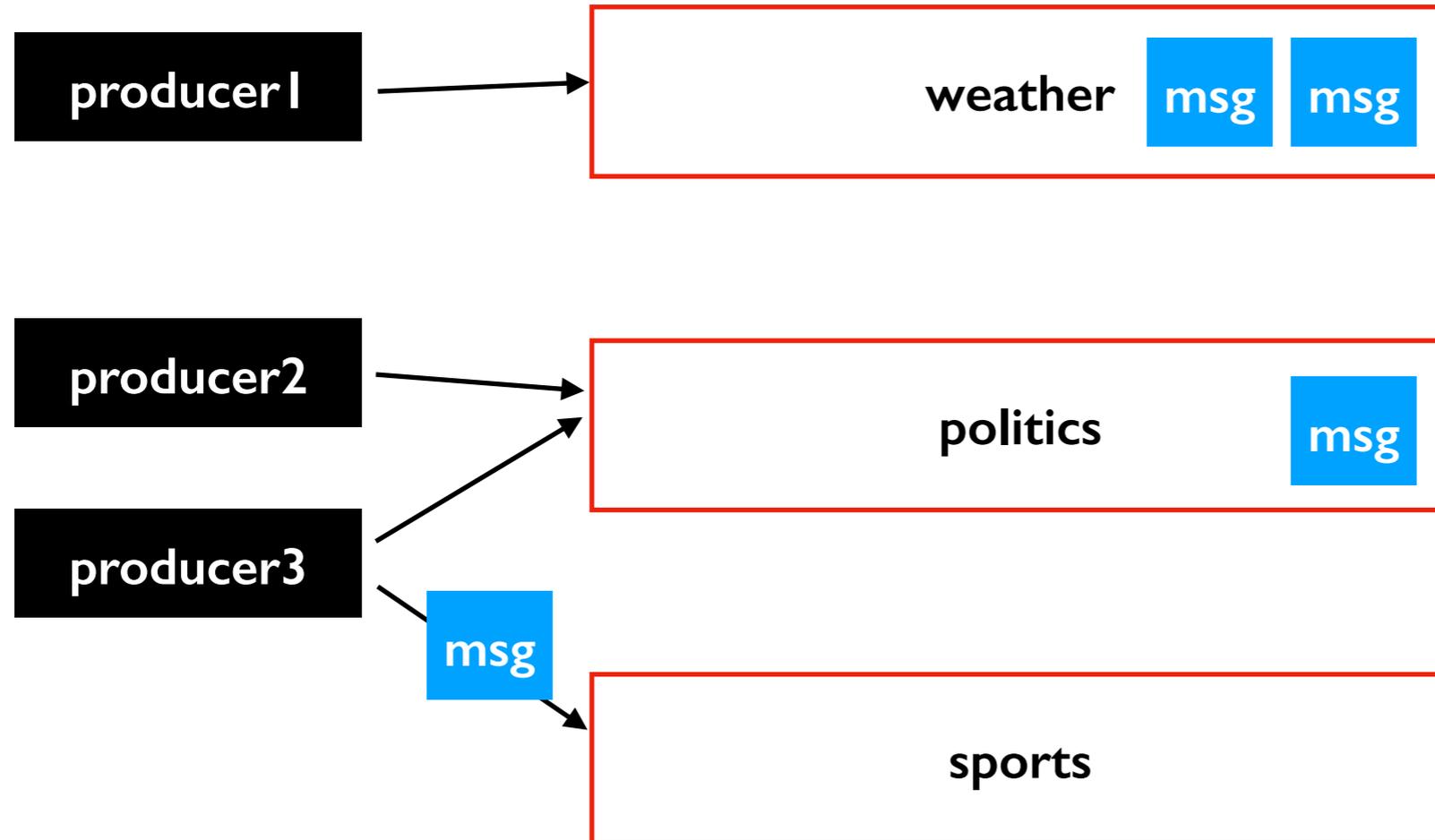
Topics

Kafka topics (managed by
servers called brokers)

weather	msg msg

politics	msg

sports	

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

```
pip install kafka-python
```

Producers **Pub**lish (pub/sub)

producers
(code you write)

Kafka topics (managed by
servers called brokers)

| producer1 | → | weather | msg | msg |

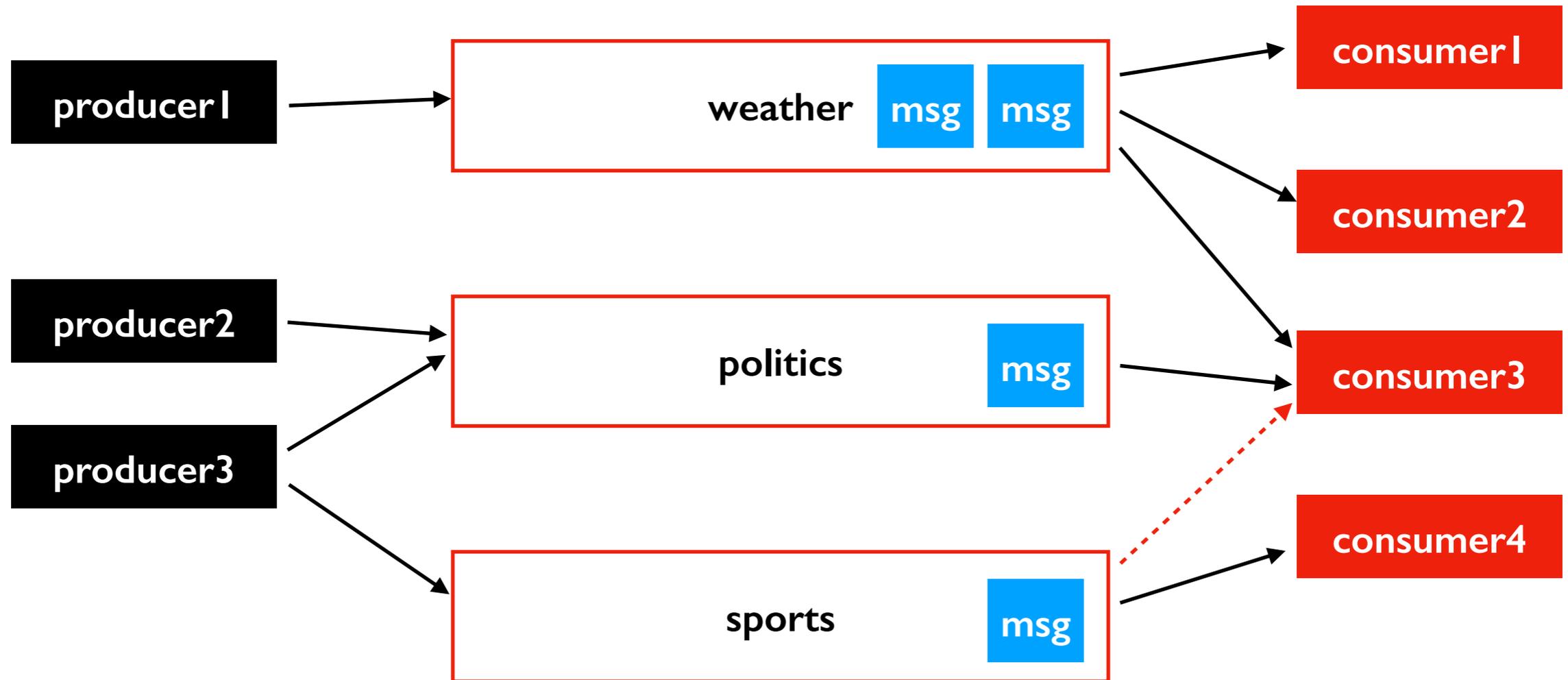| producer2 |
| producer3 |

politics | msg

msg

sports

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

Consumers **Sub**scribe (pub/sub)

producers
(code you write)

Kafka topics (managed by
servers called brokers)

consumers
(code you write)
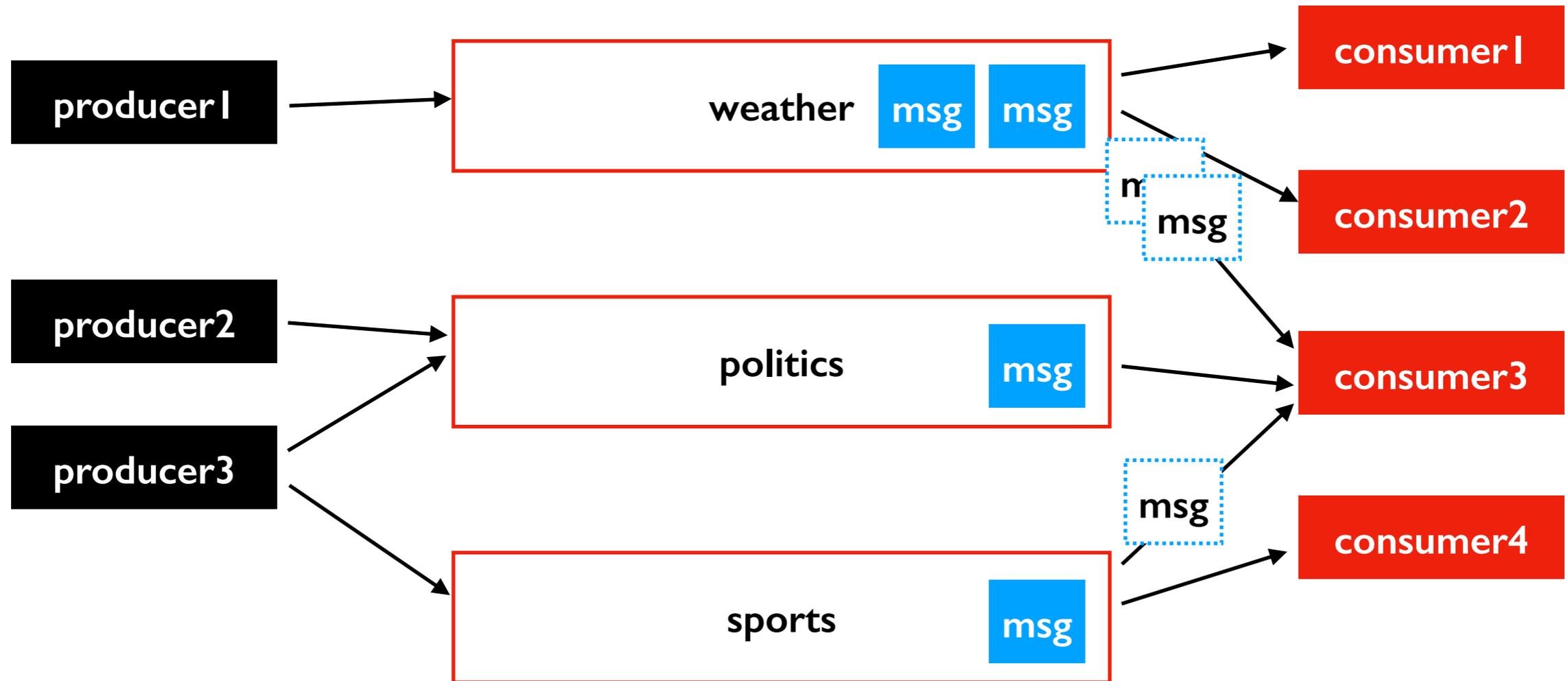
producer1

producer2

producer3

weather msg msg

politics msg

sports msg

consumer1

consumer2

consumer3

consumer4

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

Receiving Messages

producers
(code you write)

Kafka topics (managed by
servers called brokers)

consumers
(code you write)

producer1 → weather | msg | msg

producer2

producer3 → politics | msg

sports | msg

consumer1

consumer2

msg

consumer3

msg

consumer4

`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?

- 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.
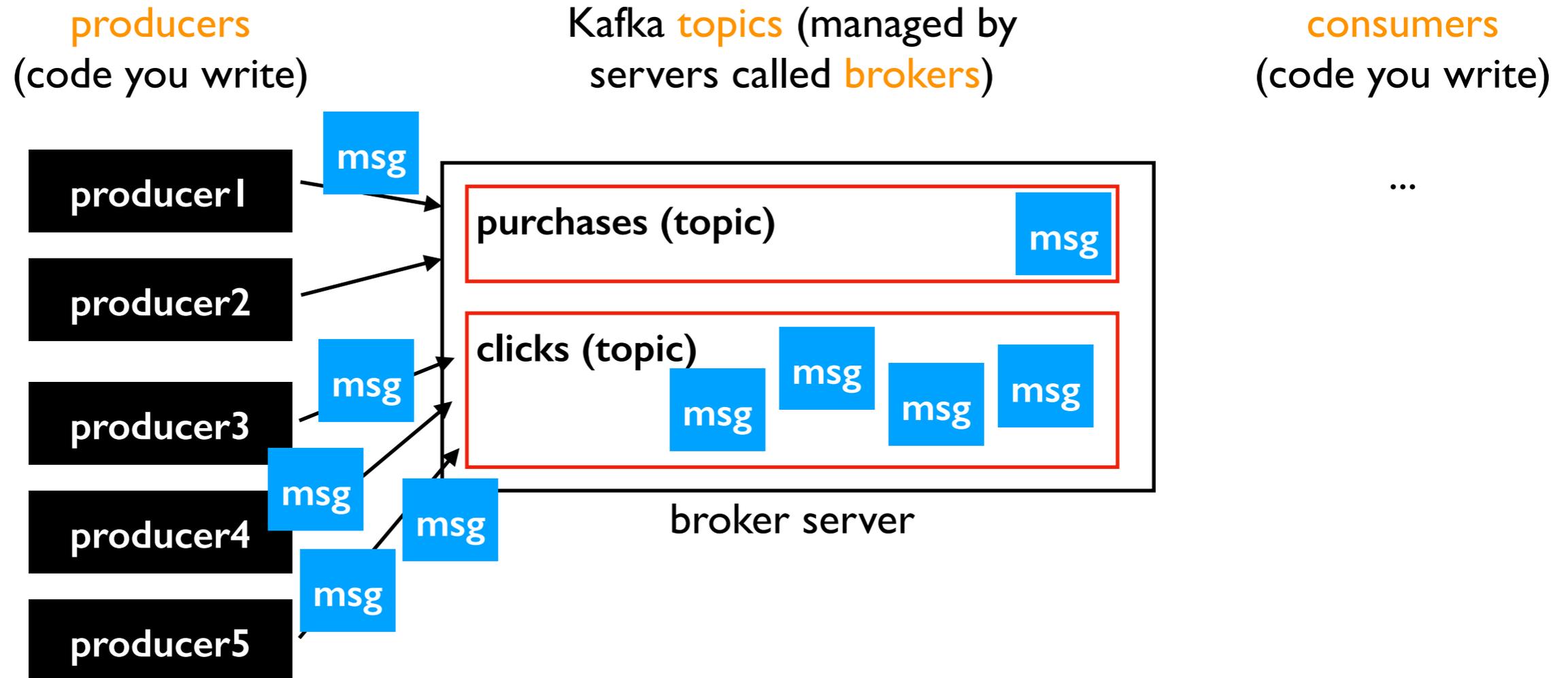
Python dict => bytes:

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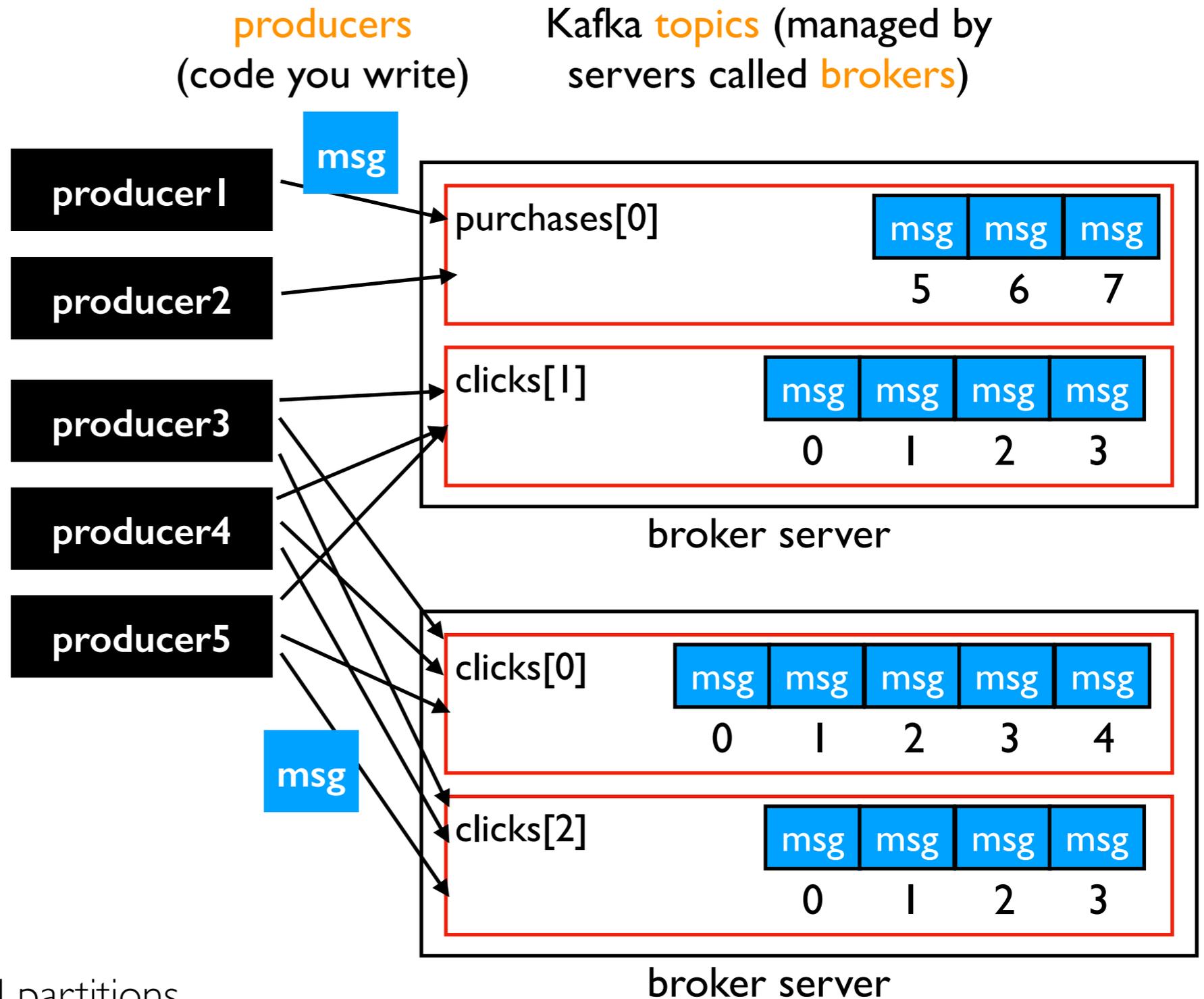
Protobuf => bytes:

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

Scaling the Brokers

producers
(code you write)

Kafka topics (managed by
servers called brokers)

consumers
(code you write)

...

producer1

msg

purchases (topic)

msg

producer2

producer3

msg

clicks (topic)

msg

msg

msg

msg

producer4

msg

msg

broker server

producer5

msg

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

Partitions

producers
(code you write)

Kafka topics (managed by
servers called brokers)

msg

producer1

producer2

producer3

producer4

producer5

msg

purchases[0]

msg	msg	msg
5	6	7

clicks[1]

msg	msg	msg	msg
0	1	2	3

broker server

clicks[0]

msg	msg	msg	msg	msg
0	1	2	3	4

clicks[2]

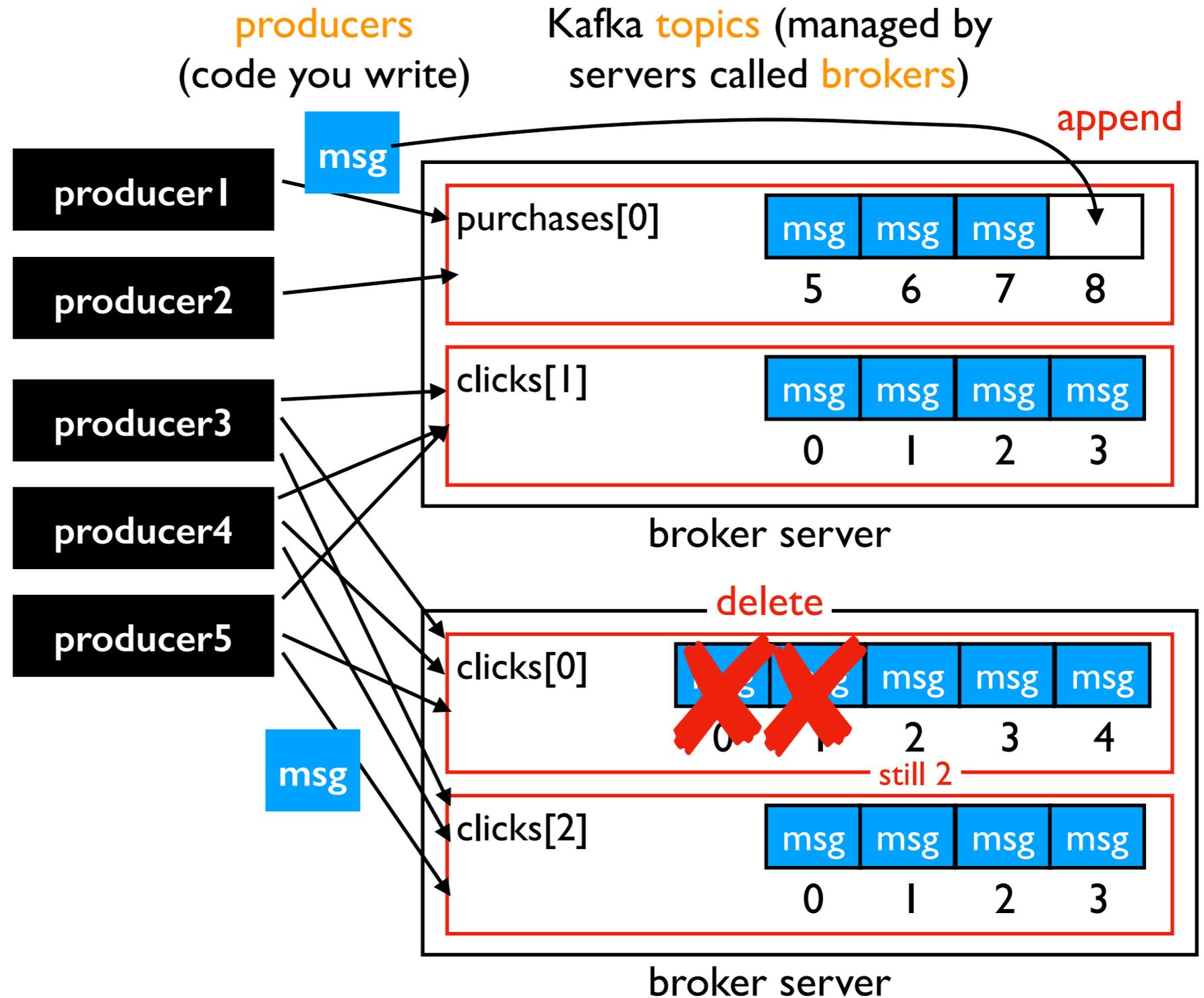| msg	msg	msg	msg
0	1	2	3

broker server

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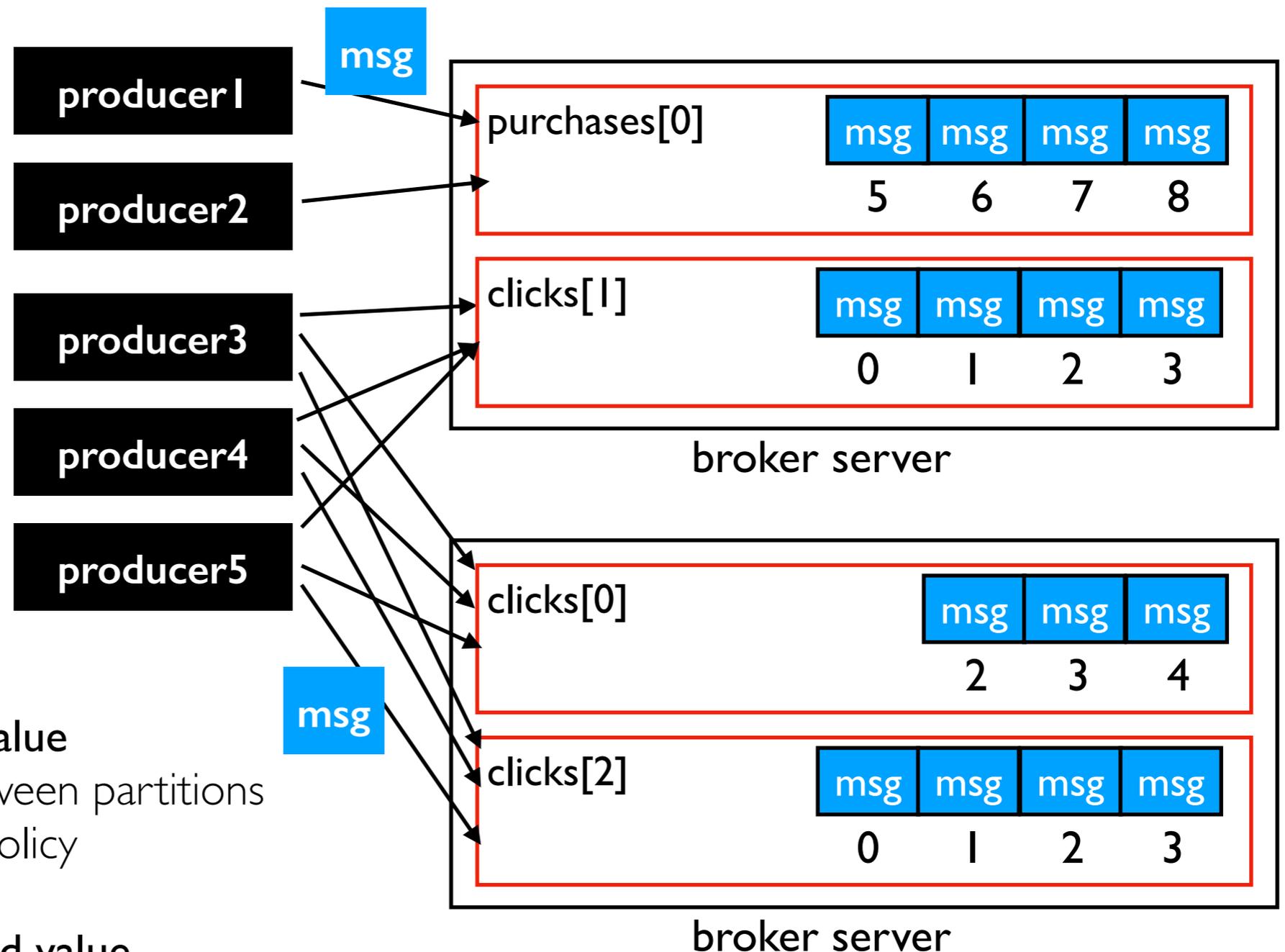- each producer using a stream works with all partitions

Changing Partitions

producers
(code you write)

Kafka topics (managed by
servers called brokers)

append

msg

producer1

producer2

producer3

producer4

producer5

msg

purchases[0]

msg	msg	msg	
5	6	7	8

clicks[1]

msg	msg	msg	msg
0	1	2	3

broker server

delete

clicks[0]

msg	msg	msg	msg	msg
0	1	2	3	4

still 2

clicks[2]

msg	msg	msg	msg
0	1	2	3

broker server

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)

msg

producer1

producer2

producer3

producer4

producer5

purchases[0]

msg	msg	msg	msg
5	6	7	8

clicks[1]

msg	msg	msg	msg
0	1	2	3

broker server

clicks[0]

msg	msg	msg
2	3	4

msg

clicks[2]

msg	msg	msg	msg
0	1	2	3

broker server

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:
  `hash(key) % partition_count`
- same keys will go to the same partition
- *can plug in alternative partitioning schemes*

Consumers: Read Offsets
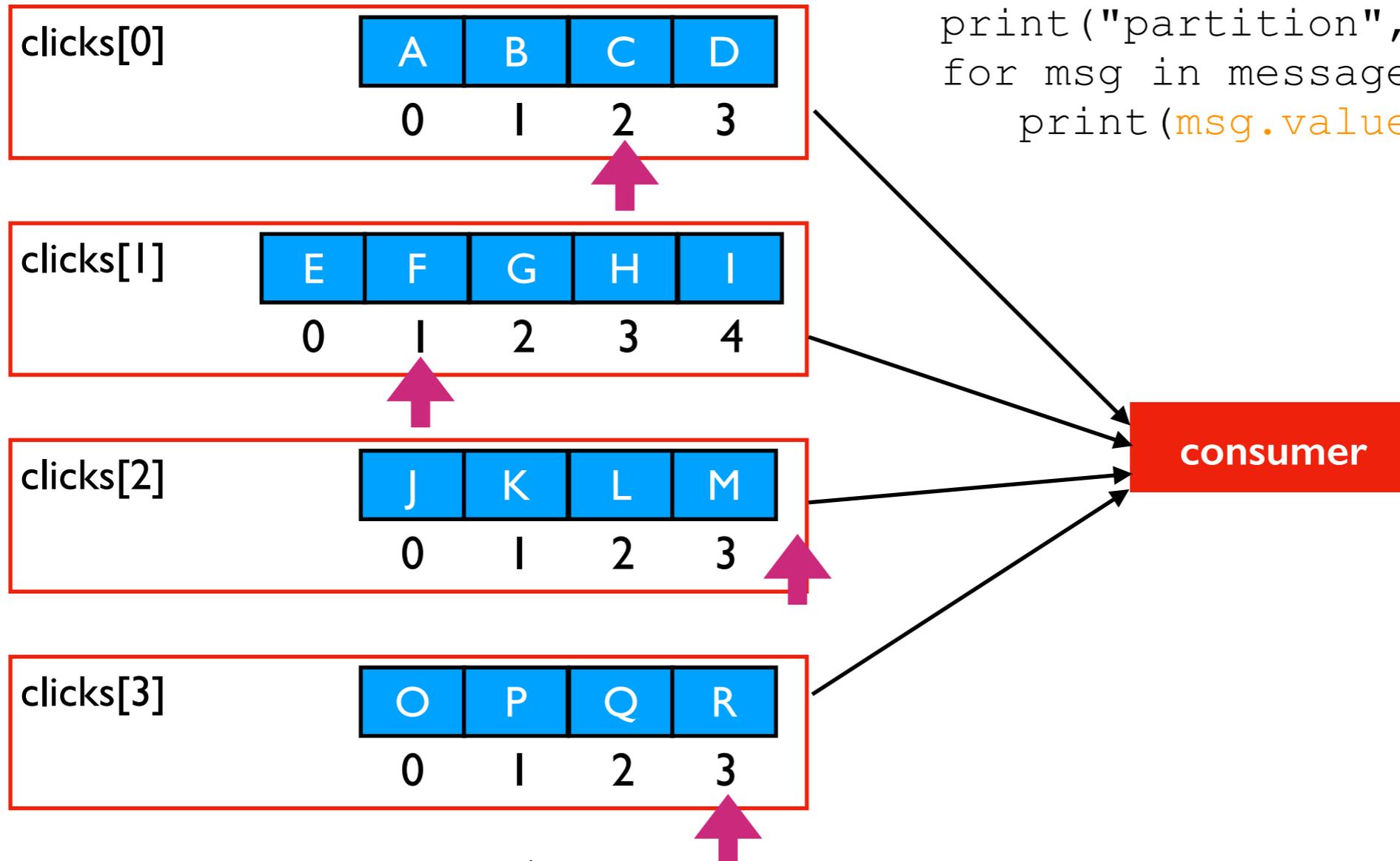
```
batch = consumer.poll(1000)
for topic, messages in batch.items():
    print("partition", topic.partition)
    for msg in messages:
        print(msg.value)
```
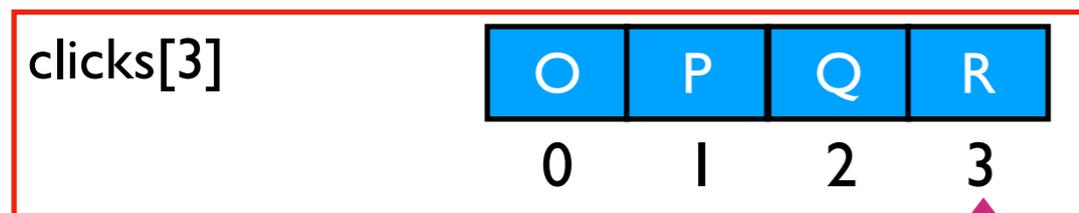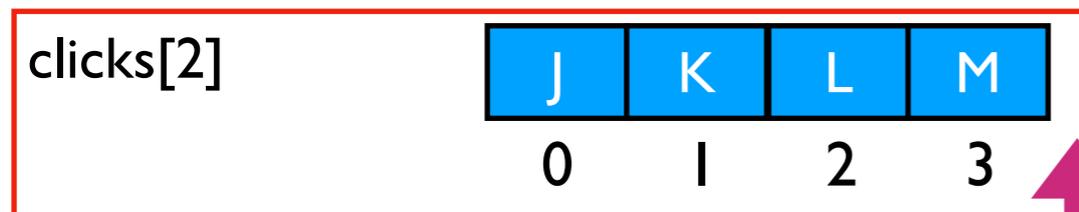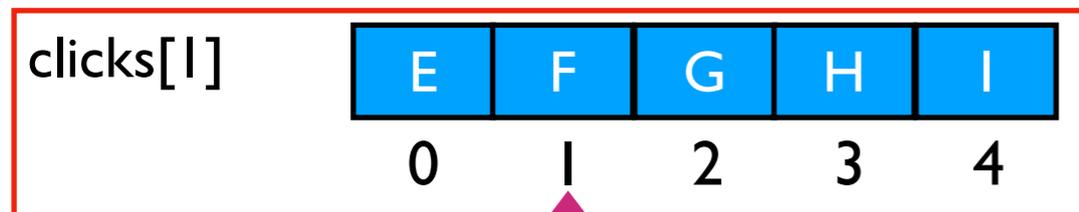
clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	3

consumer

	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

```
batch = consumer.poll(1000)
for topic, messages in batch.items():
    print("partiton", topic.partition)
    for msg in messages:
        print(msg.value)
```
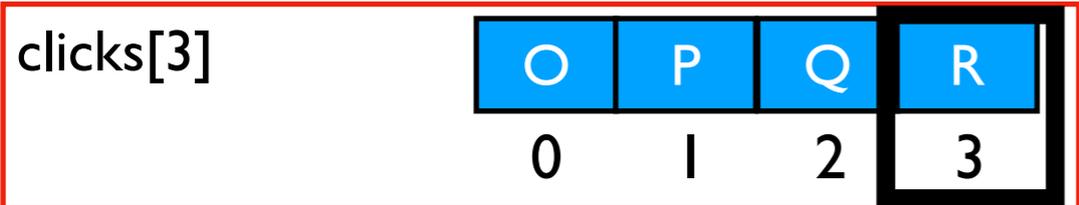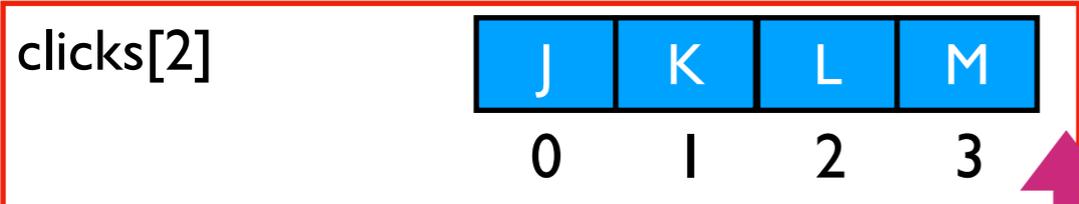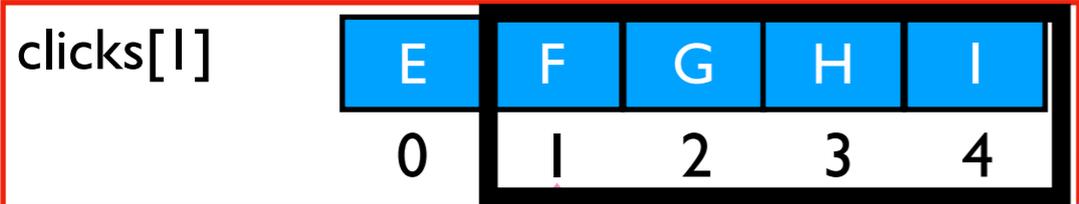
clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

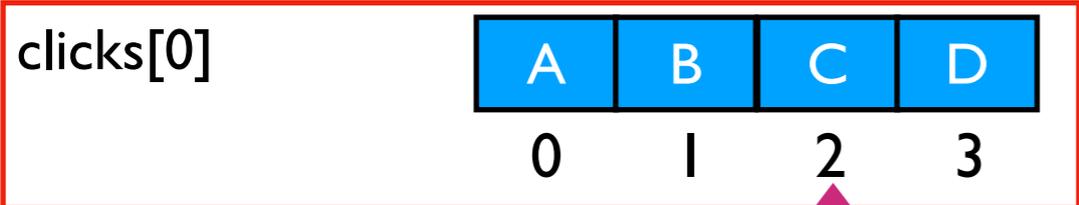| O	P	Q	R
0	1	2	3
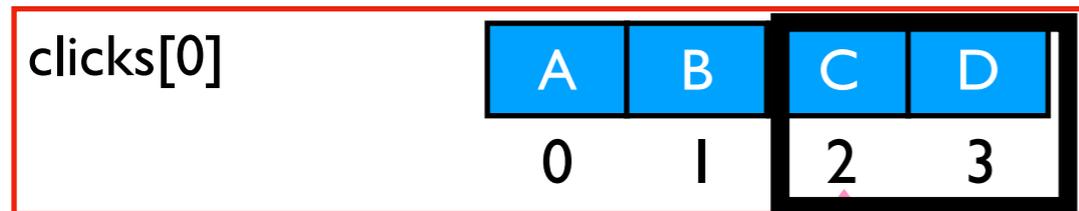
consumer

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

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

clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	3

consumer

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

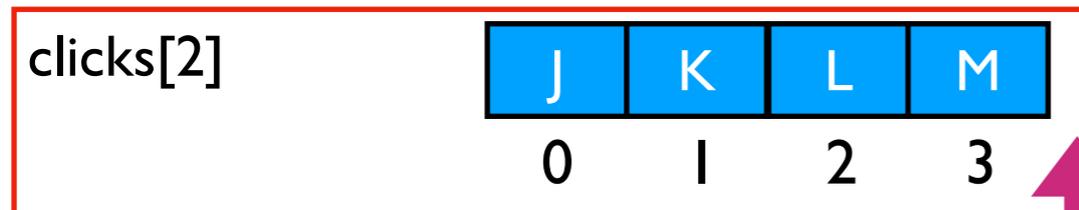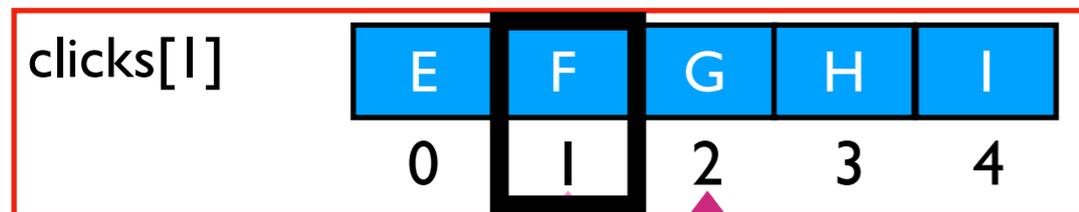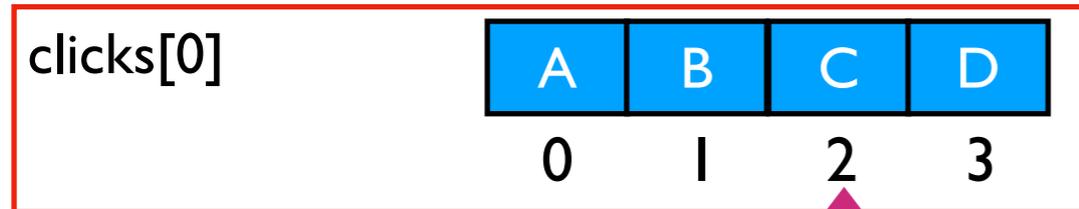|	offset
clicks[0]	2
clicks[1]	5
clicks[2]	4
clicks[3]	4

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

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

clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	3

consumer

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 x >= y.

Other things are partially ordered, like git commits. Sometimes you can compare, sometimes you can't!
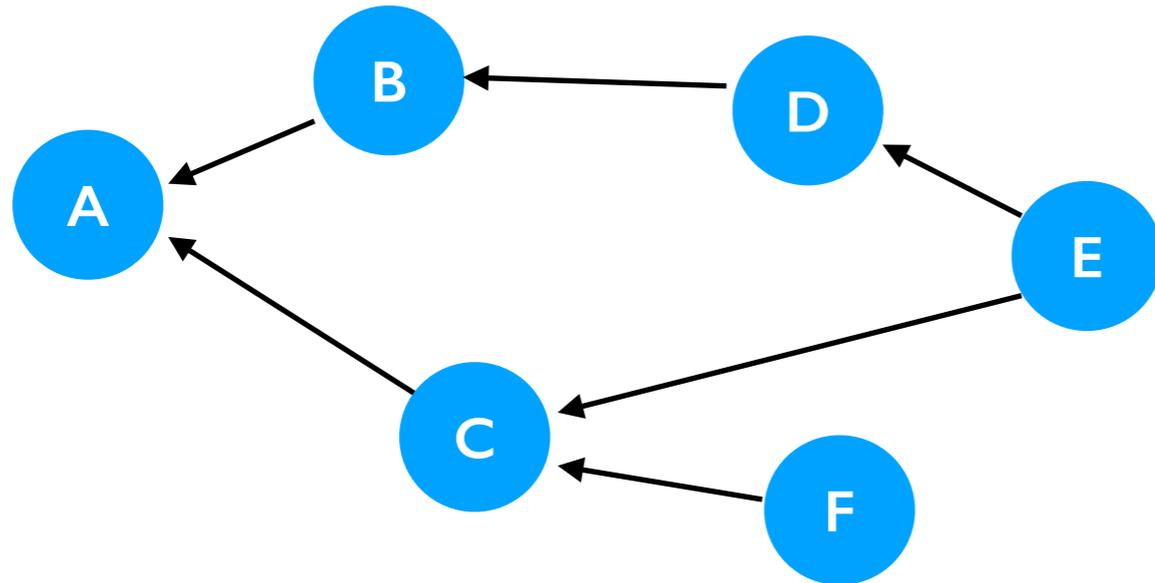
A < B   A < C   D < E   ...

Can't compare B and C
Can't compare D and F
...

Ordering Kafka Messages

Kafka Messages are generally partially ordered (though if you have one partition only, they are totally 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:
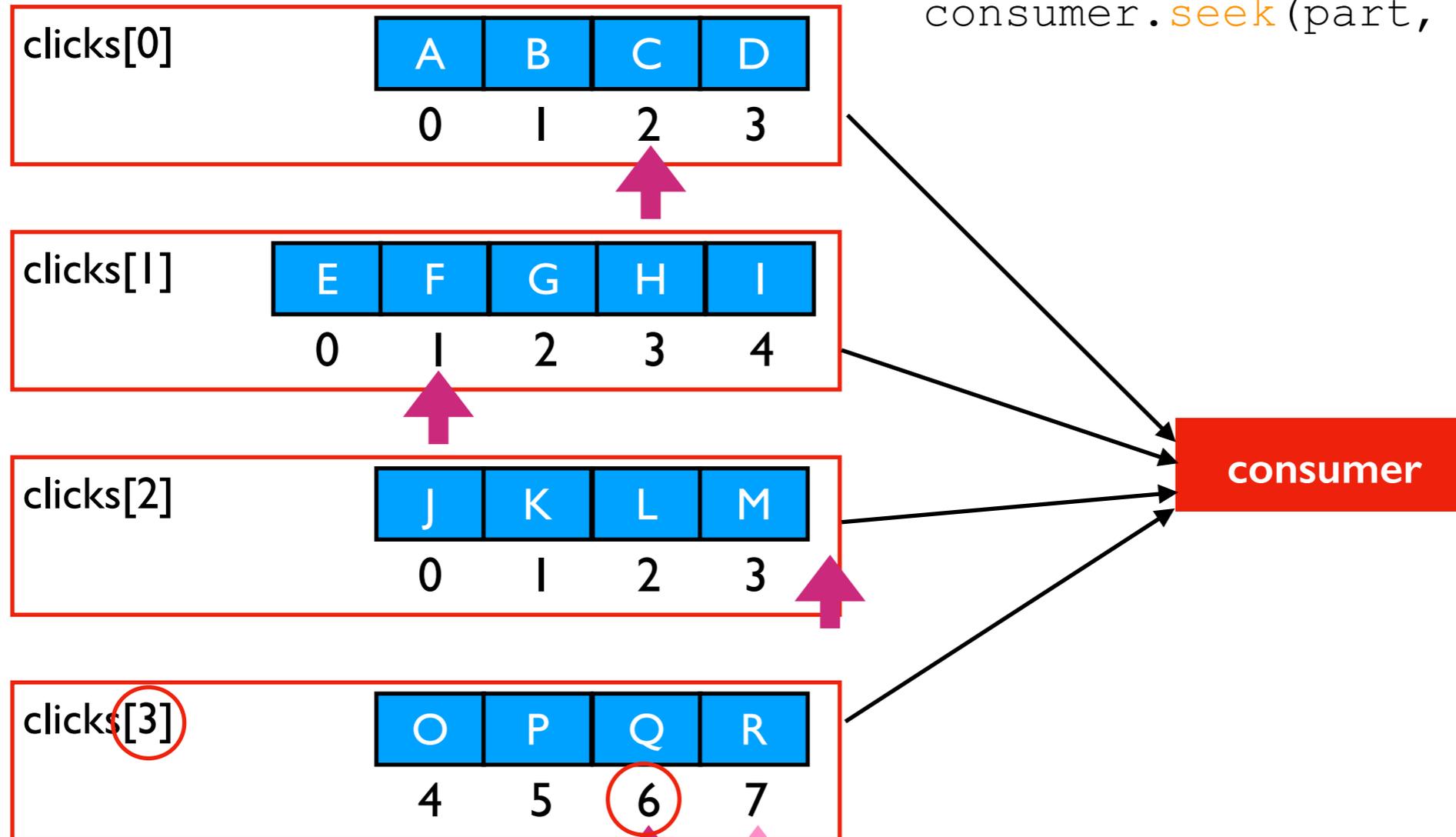- we say B "happened after" A
- A and B will be in the same partition (assuming partition count is constant)
- each consumer group of the topic will consume A before B

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

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

Topic Partitions
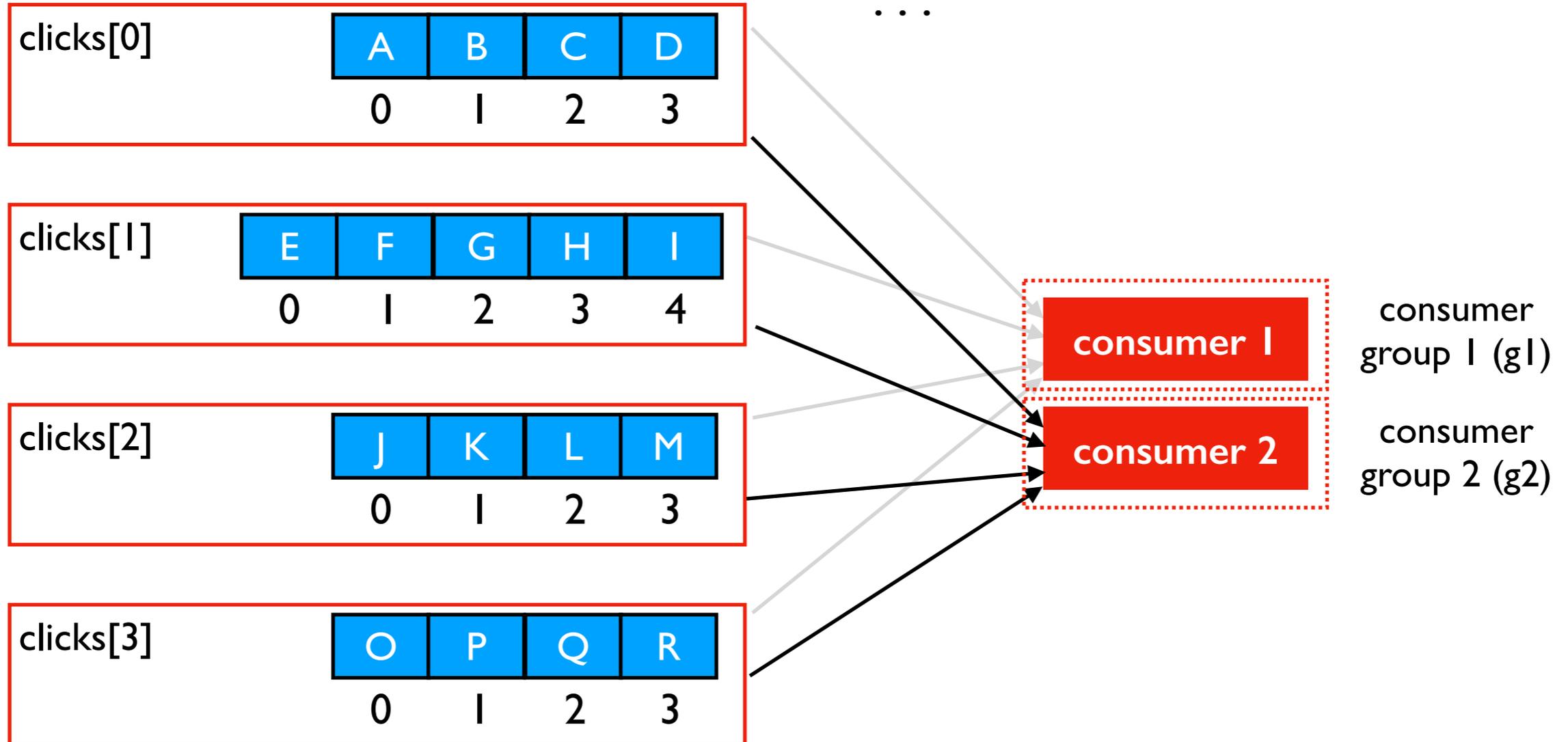
clicks[0]	A	B	C	D
	0	1	2	3

clicks[1]	E	F	G	H	I
	0	1	2	3	4

clicks[2]	J	K	L	M
	0	1	2	3

clicks[3]	O	P	Q	R
	4	5	6	7

consumer

	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

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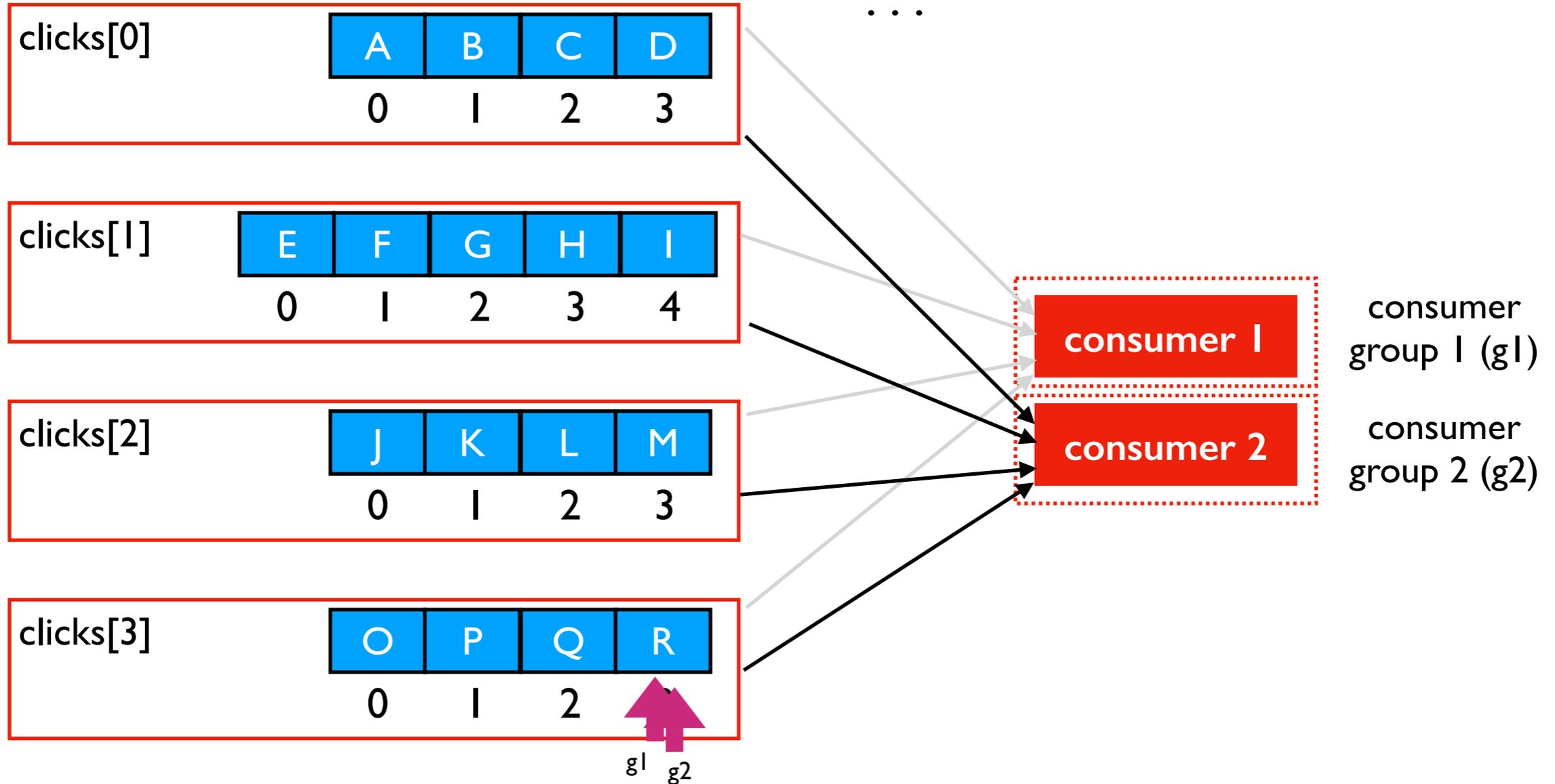
Topic Partitions

clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	3

consumer 1   consumer group 1 (g1)

consumer 2   consumer group 2 (g2)

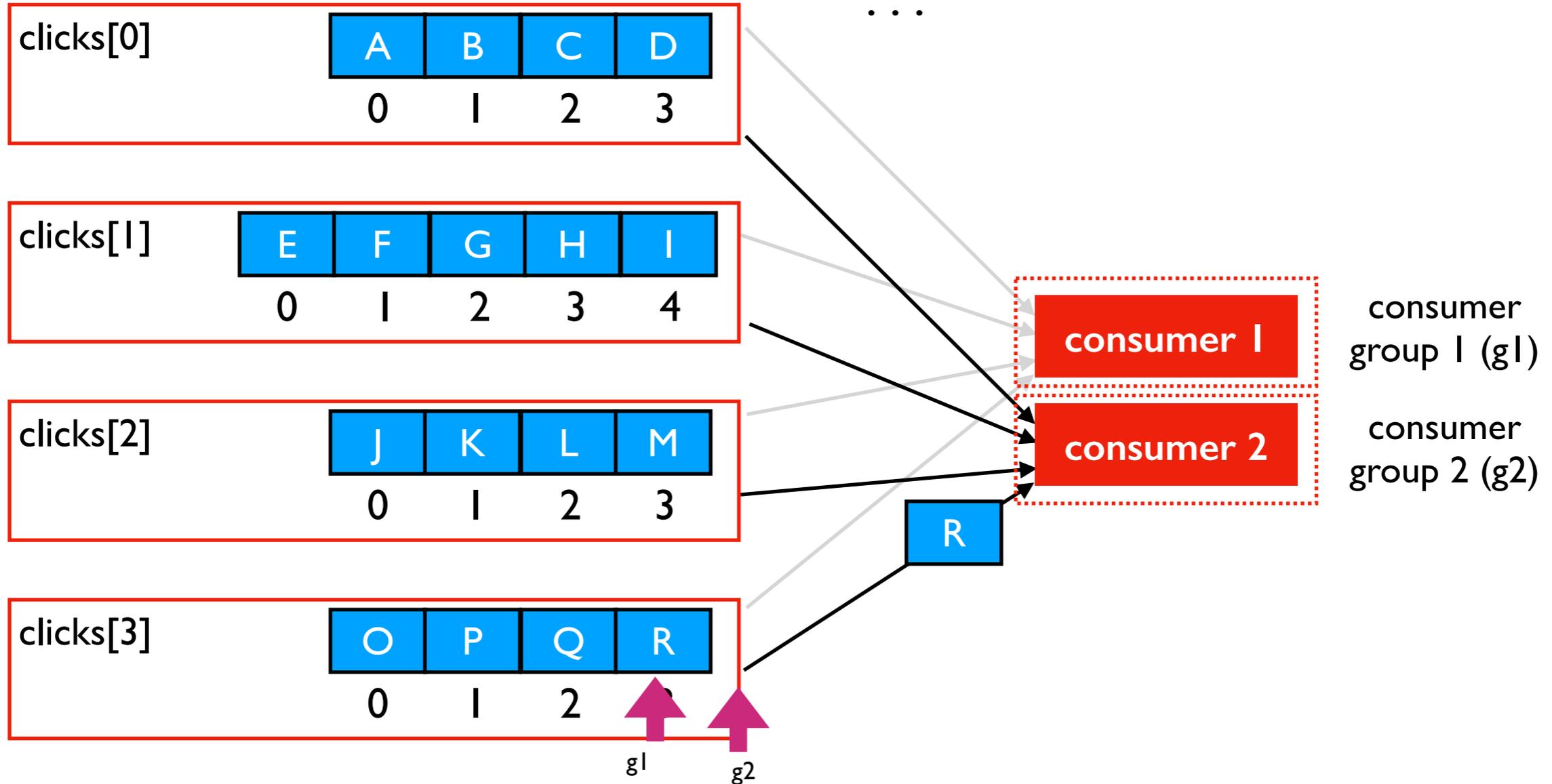|	g1 offsets	g2 offsets
clicks[0]	2	3
clicks[1]	1	2
clicks[2]	4	4
clicks[3]	3	3

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

clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	

g1 g2

consumer 1 — consumer group 1 (g1)

consumer 2 — consumer group 2 (g2)

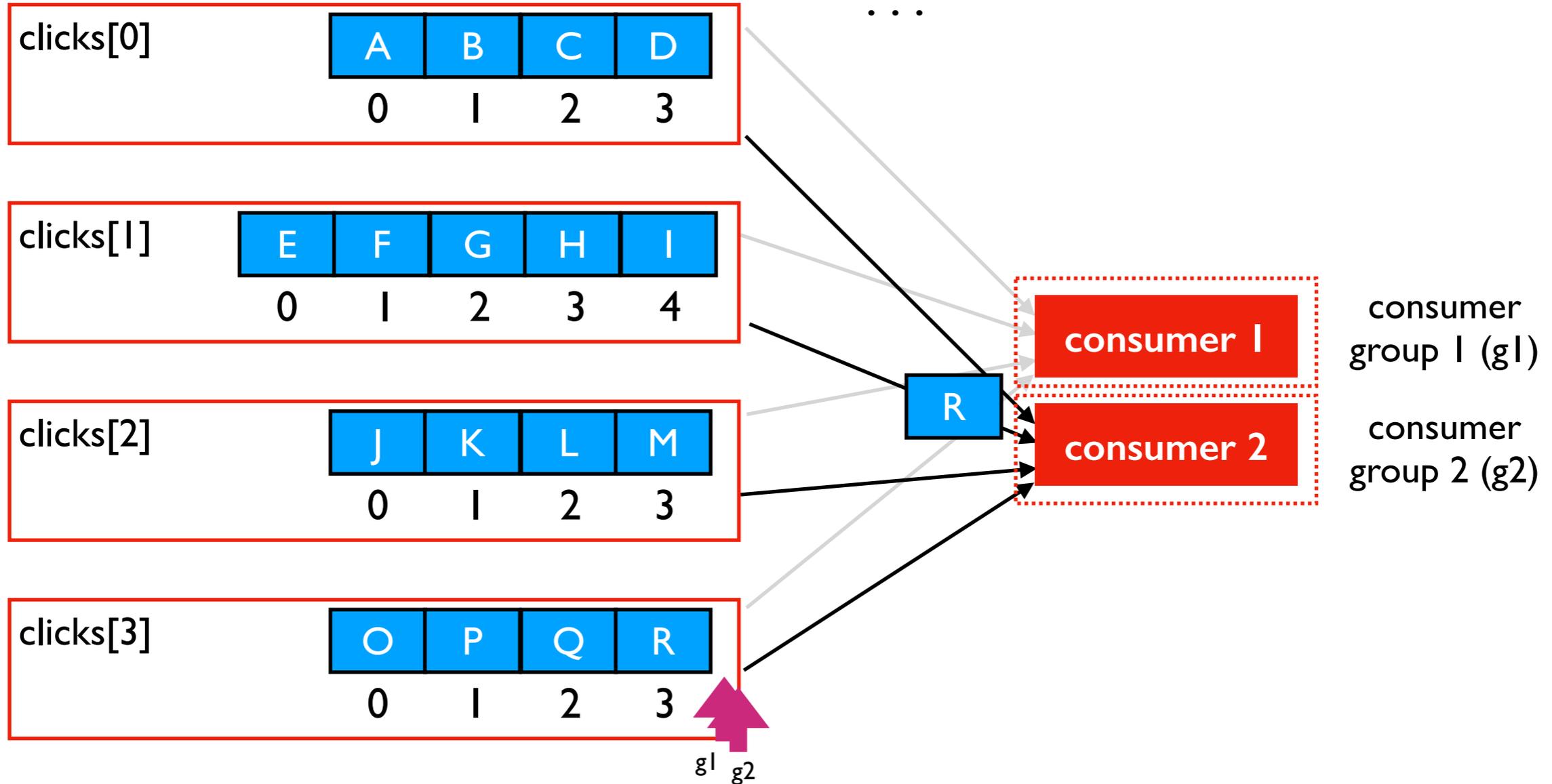|	g1 offsets	g2 offsets
clicks[0]	2	3
clicks[1]	1	2
clicks[2]	4	4
clicks[3]	3	3

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)
...
```
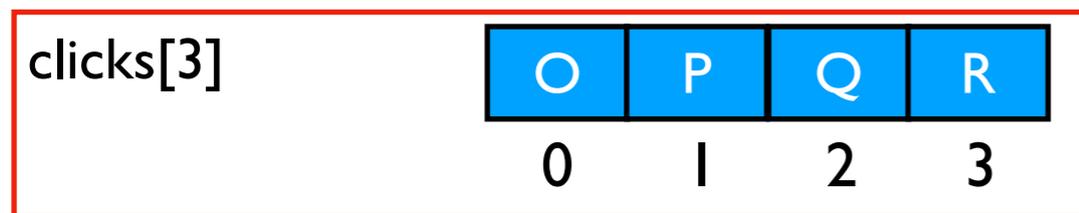
Topic Partitions

clicks[0] A B C D 0 1 2 3

clicks[1] E F G H I 0 1 2 3 4

clicks[2] J K L M 0 1 2 3

clicks[3] O P Q R 0 1 2

consumer 1 — consumer group 1 (g1)

consumer 2 — consumer group 2 (g2)

R

g1   g2

	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)
...
```
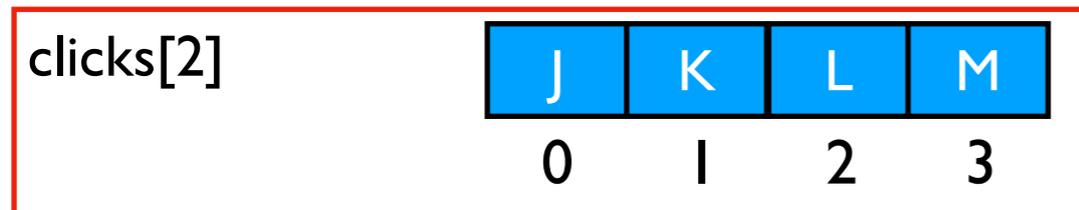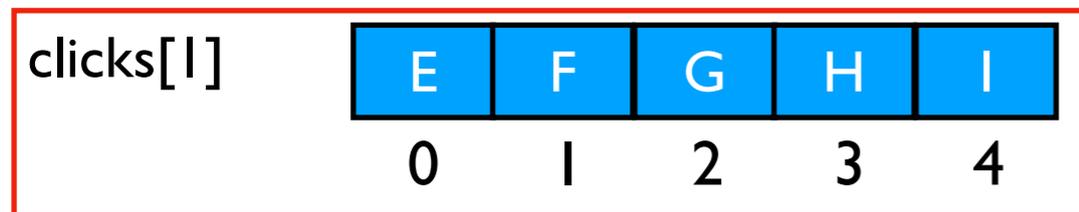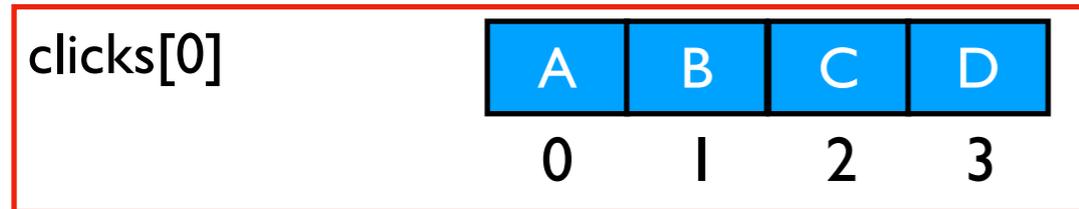
Topic Partitions

clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	3

R

consumer 1 — consumer group 1 (g1)

consumer 2 — consumer group 2 (g2)

g1   g2

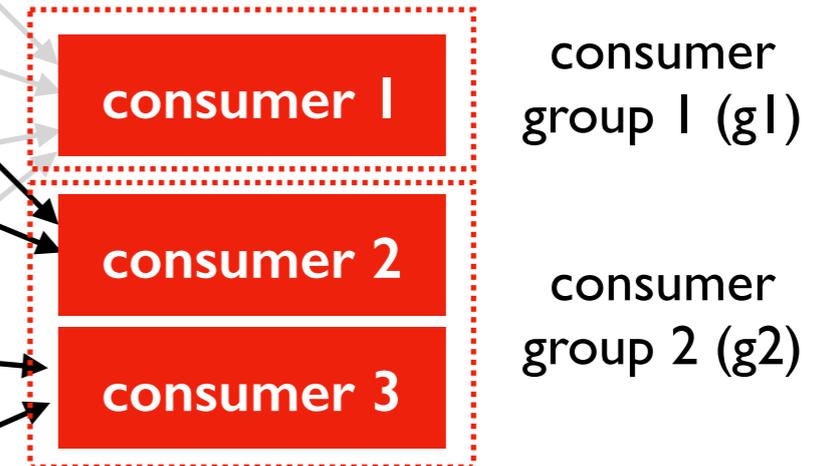|	g1 offsets	g2 offsets
clicks[0]	2	3
clicks[1]	1	2
clicks[2]	4	4
clicks[3]	4	4

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

clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	3

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

consumer 1 — consumer group 1 (g1)
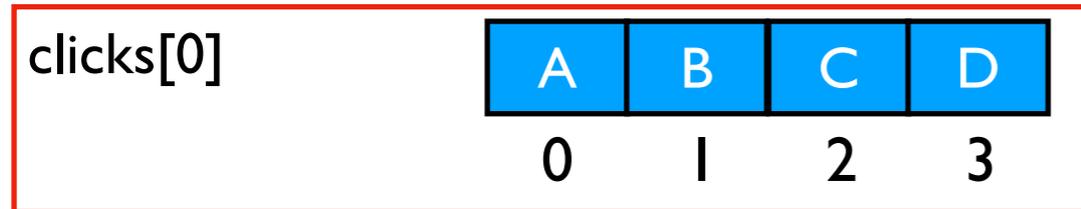
consumer 2

consumer 3 — consumer group 2 (g2)

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

clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	3

```
# consumer 3: subscribed to clicks
while True:
    batch = consumer.poll(1000)
    for topic, msgs in batch.items():
        for msg in msgs:
            ...
consumer.close()
```
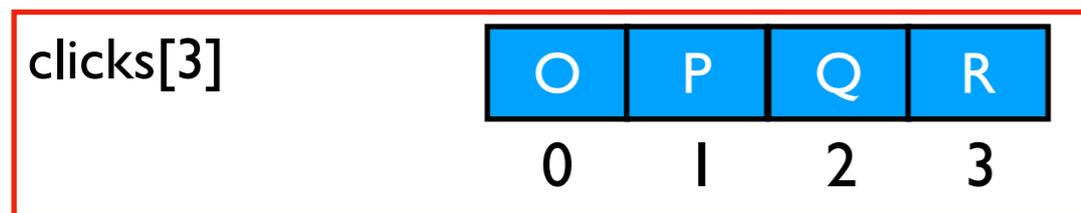
consumer 2

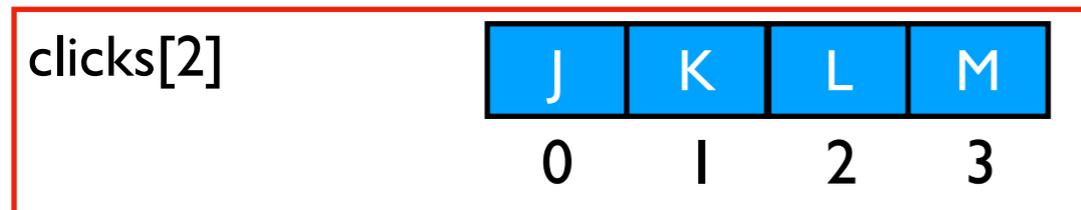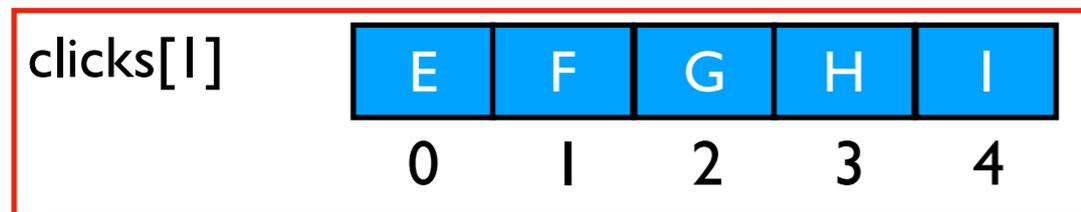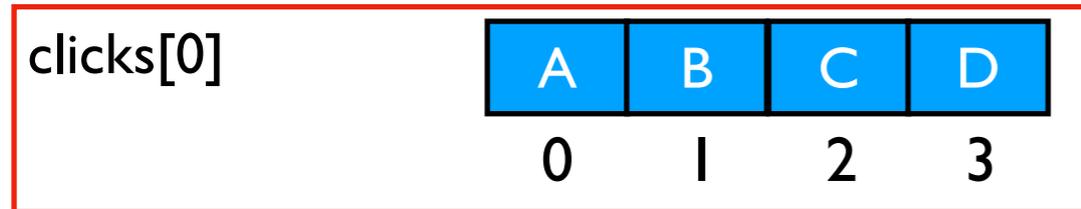consumer 3

consumer 4

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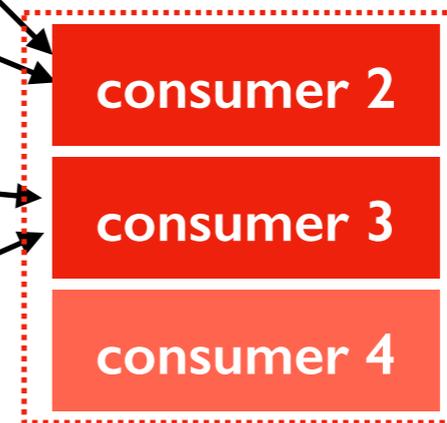
```
# consumer 3: subscribed to clicks
while True:
    batch = consumer.poll(1000)
    for topic, msgs in batch.items():
        for msg in msgs:
            ...
consumer.close()
```

clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	3

best to take away a partition at these points
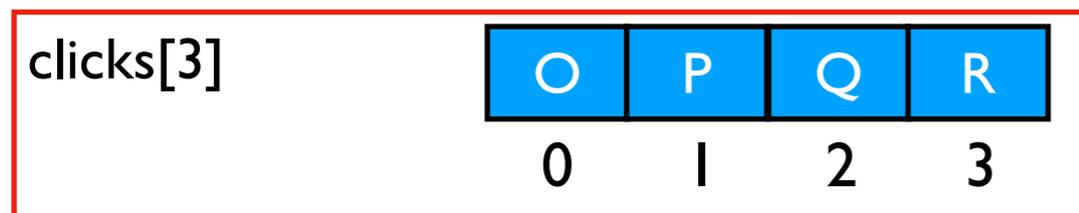
consumer 2

consumer 3

consumer 4

consumer group 2 (g2)
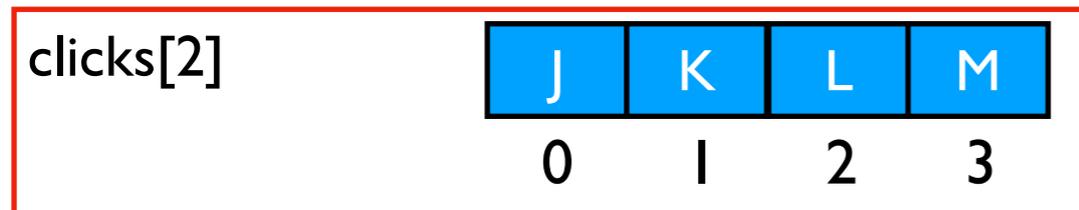
Assignment and re-assignment
- by default, consumers are automatically assigned partitions when they start polling
- **challenge:** Kafka shouldn't re-assign a partition in the middle of a batch (might double process messages)

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

clicks[0]

A	B	C	D
0	1	2	3

clicks[1]

E	F	G	H	I
0	1	2	3	4

clicks[2]

J	K	L	M
0	1	2	3

clicks[3]

O	P	Q	R
0	1	2	3

```
# 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
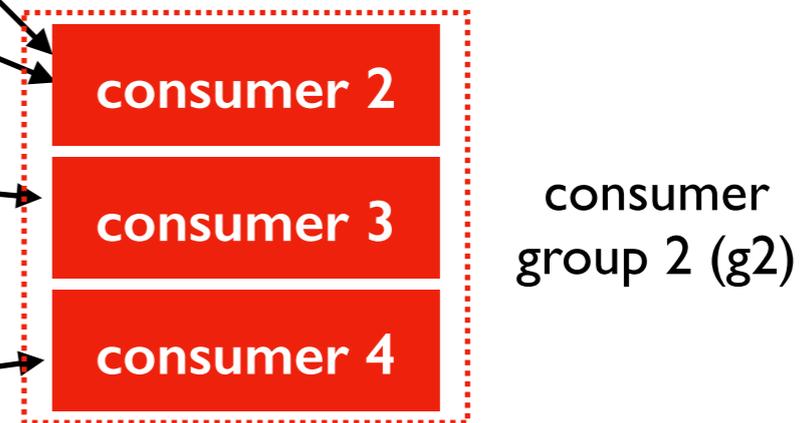
consumer 2

consumer 3

consumer 4

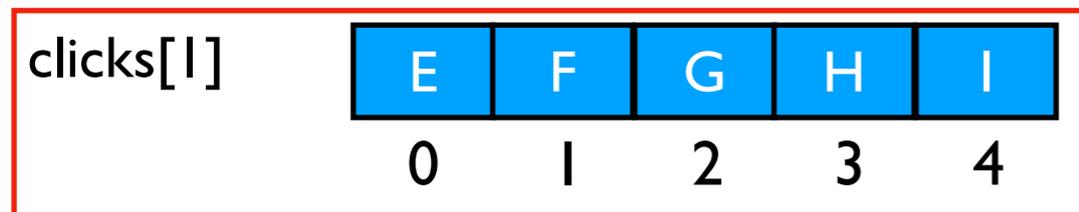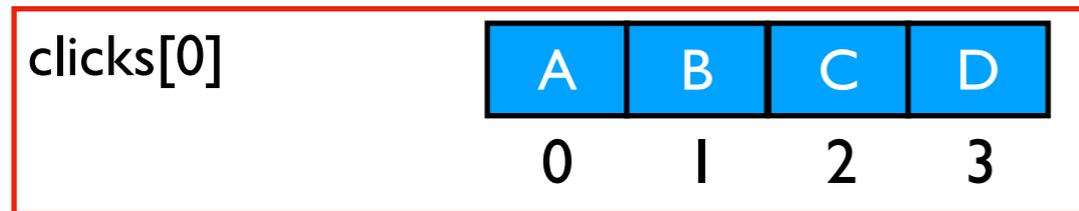consumer group 2 (g2)

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
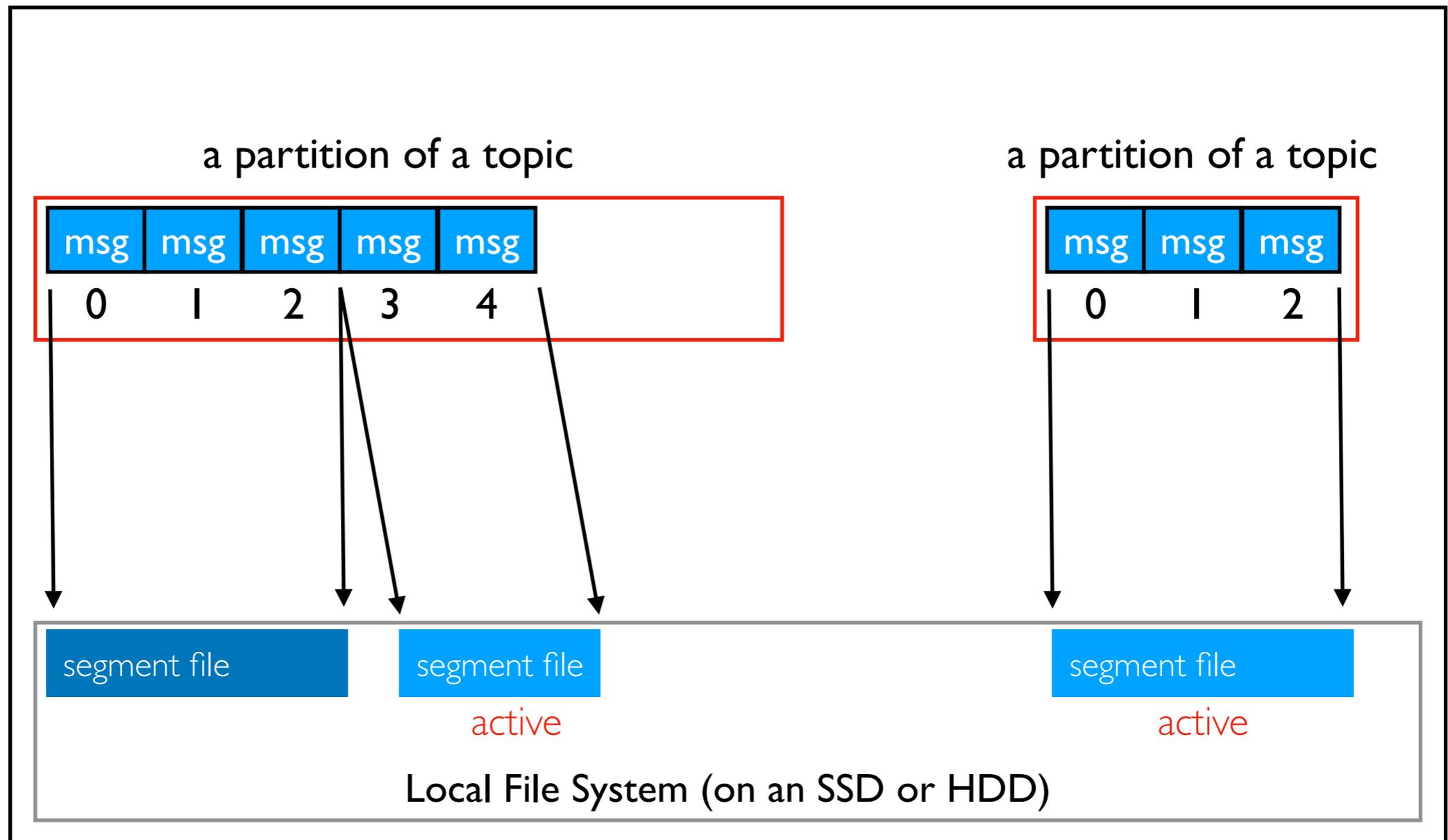
broker server

a partition of a topic

msg	msg	msg	msg	msg
0	1	2	3	4

a partition of a topic

msg	msg	msg
0	1	2

segment file

segment file

active

segment file

active

Local File System (on an SSD or HDD)

- 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

a partition of a topic

msg	msg	msg	msg	msg	msg
0	1	2	3	4	5

a partition of a topic

msg	msg	msg
0	1	2

segment file

segment file

active

segment file

active

Local File System (on an SSD or HDD)
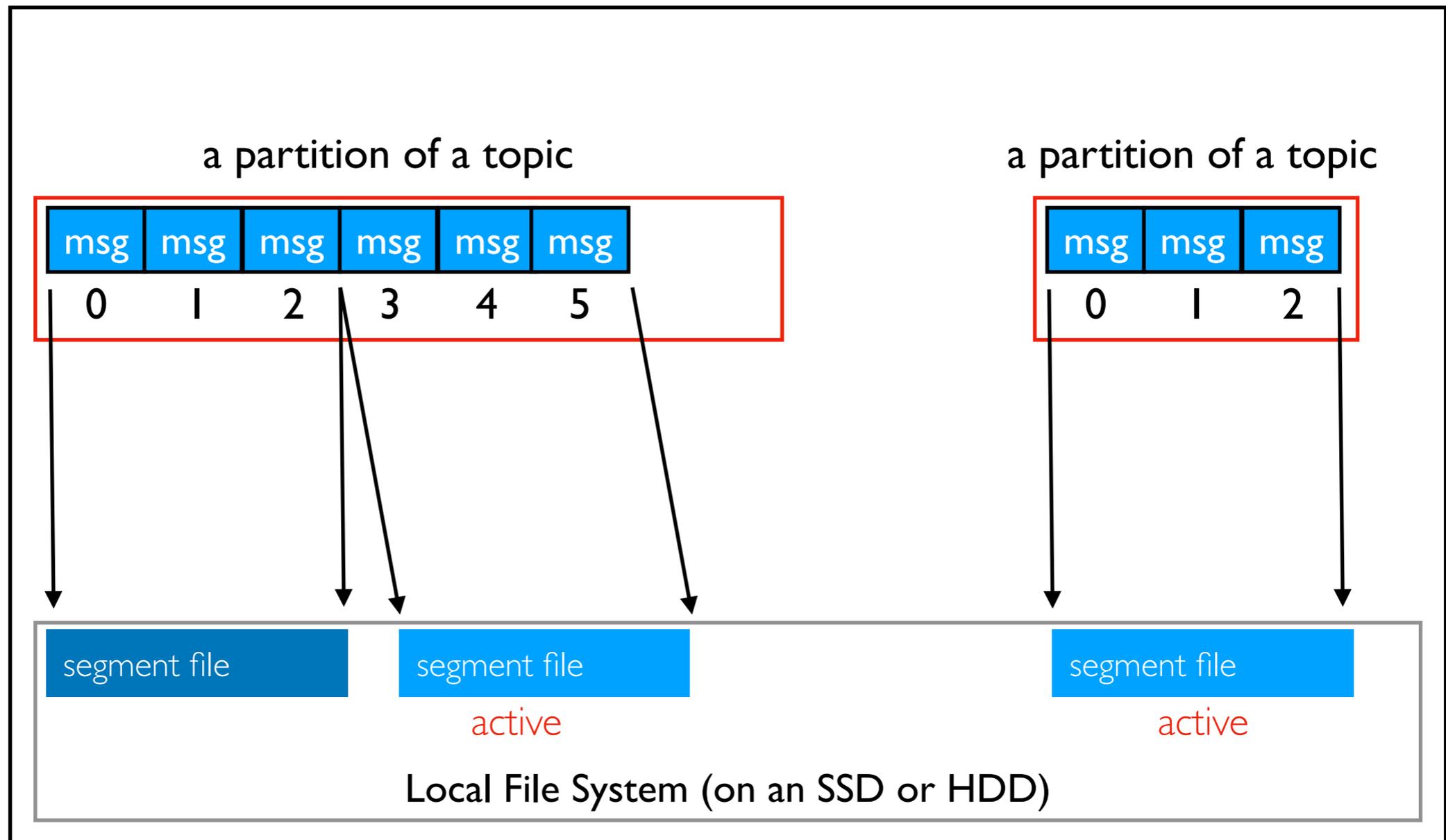
- partitions are divided into consecutive regions and saved in segment files
- all new data is sequentially written to the end of an active segment

Segment Files: Log Rollover and Deletion

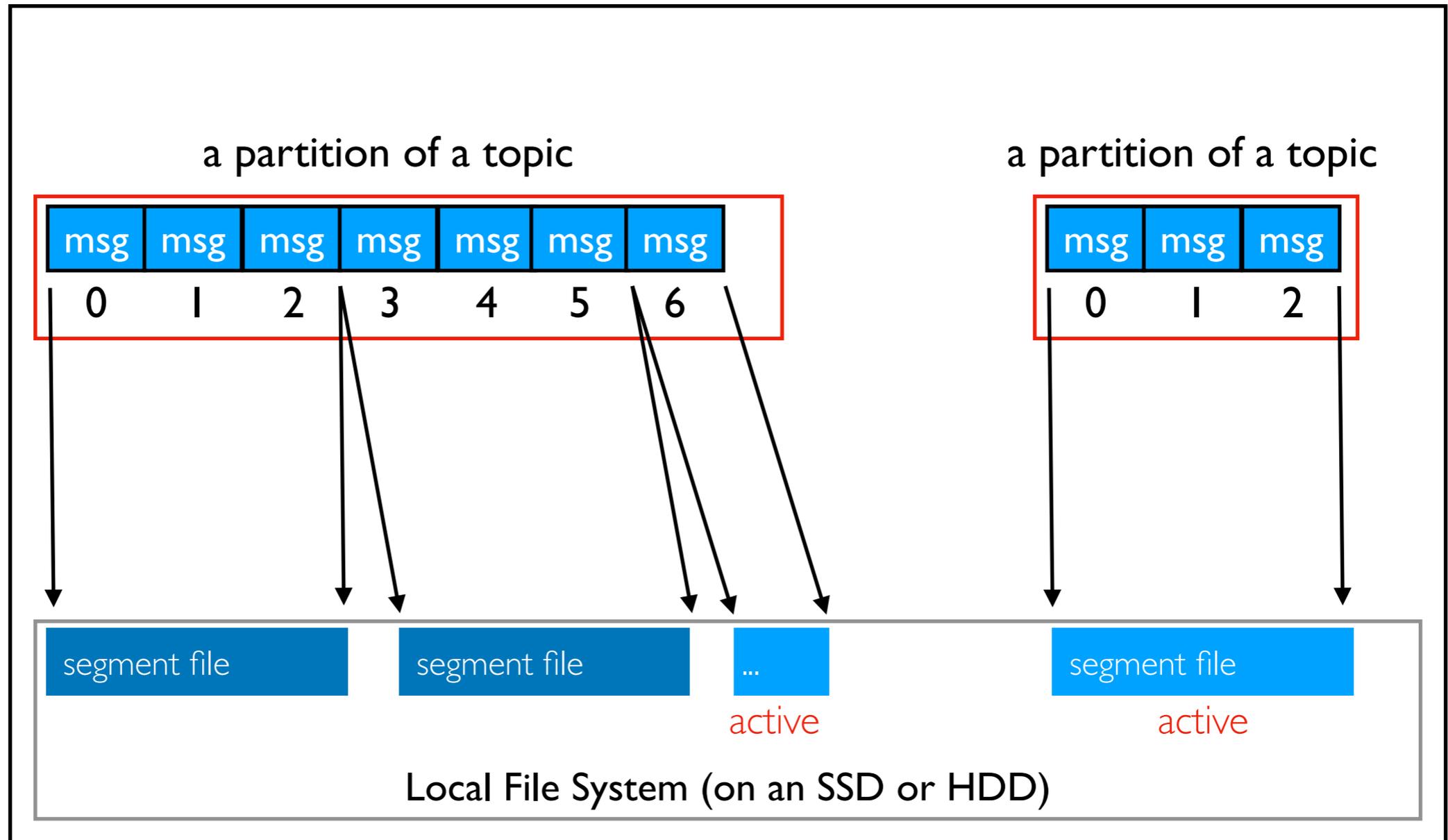broker server

a partition of a topic

msg	msg	msg	msg	msg	msg	msg
0	1	2	3	4	5	6

a partition of a topic

msg	msg	msg
0	1	2

segment file

segment file

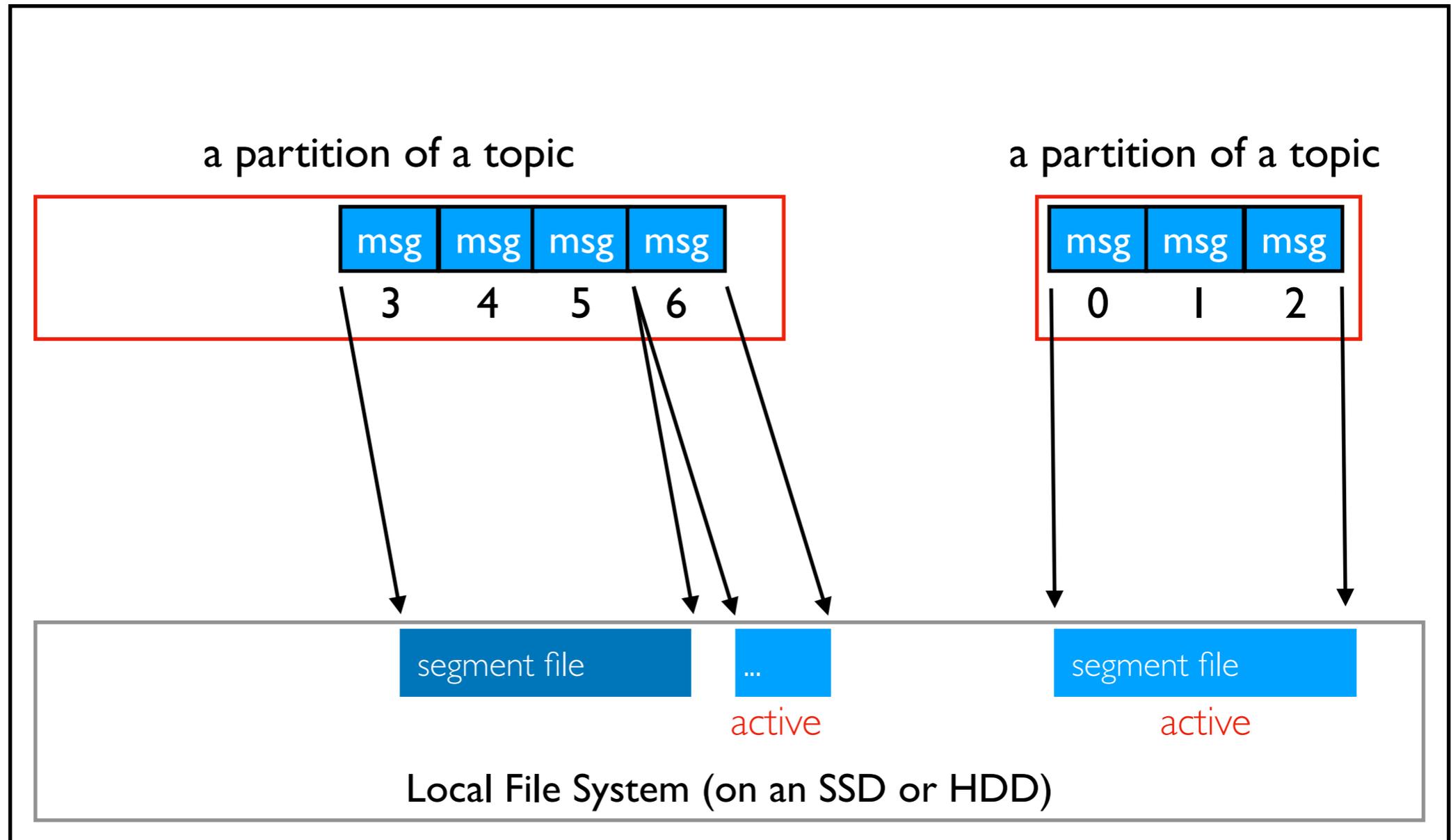...
active

segment file
active

Local File System (on an SSD or HDD)

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

Segment Files: Log Rollover and Deletion

broker server

a partition of a topic

msg	msg	msg	msg
3	4	5	6

a partition of a topic

msg	msg	msg
0	1	2

segment file

...

active

segment file

active

Local File System (on an SSD or HDD)

- **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