[544] Kafka Reliability

Tyler Caraza-Harter

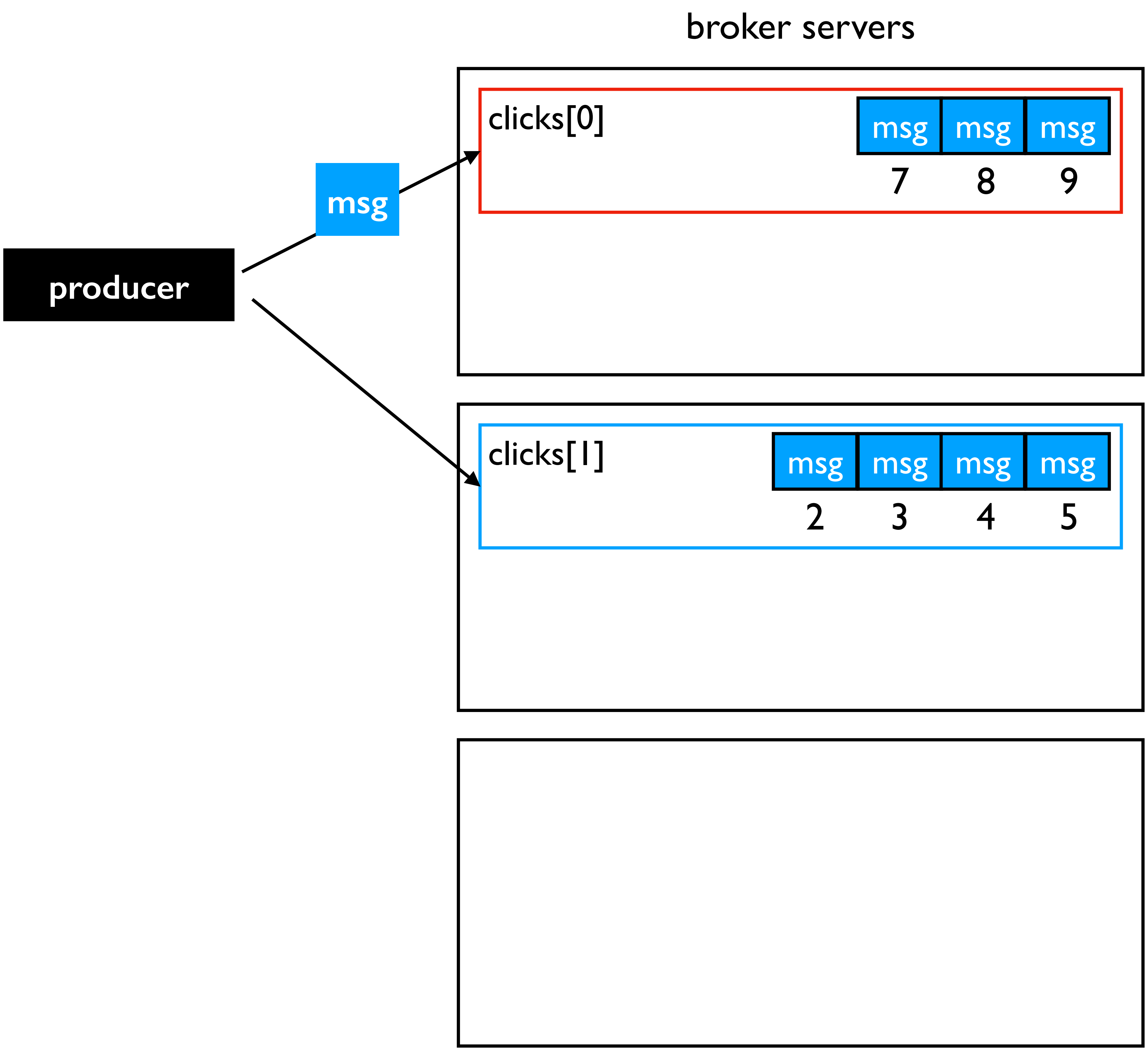
Outline: Kafka Reliability

Kafka Replication

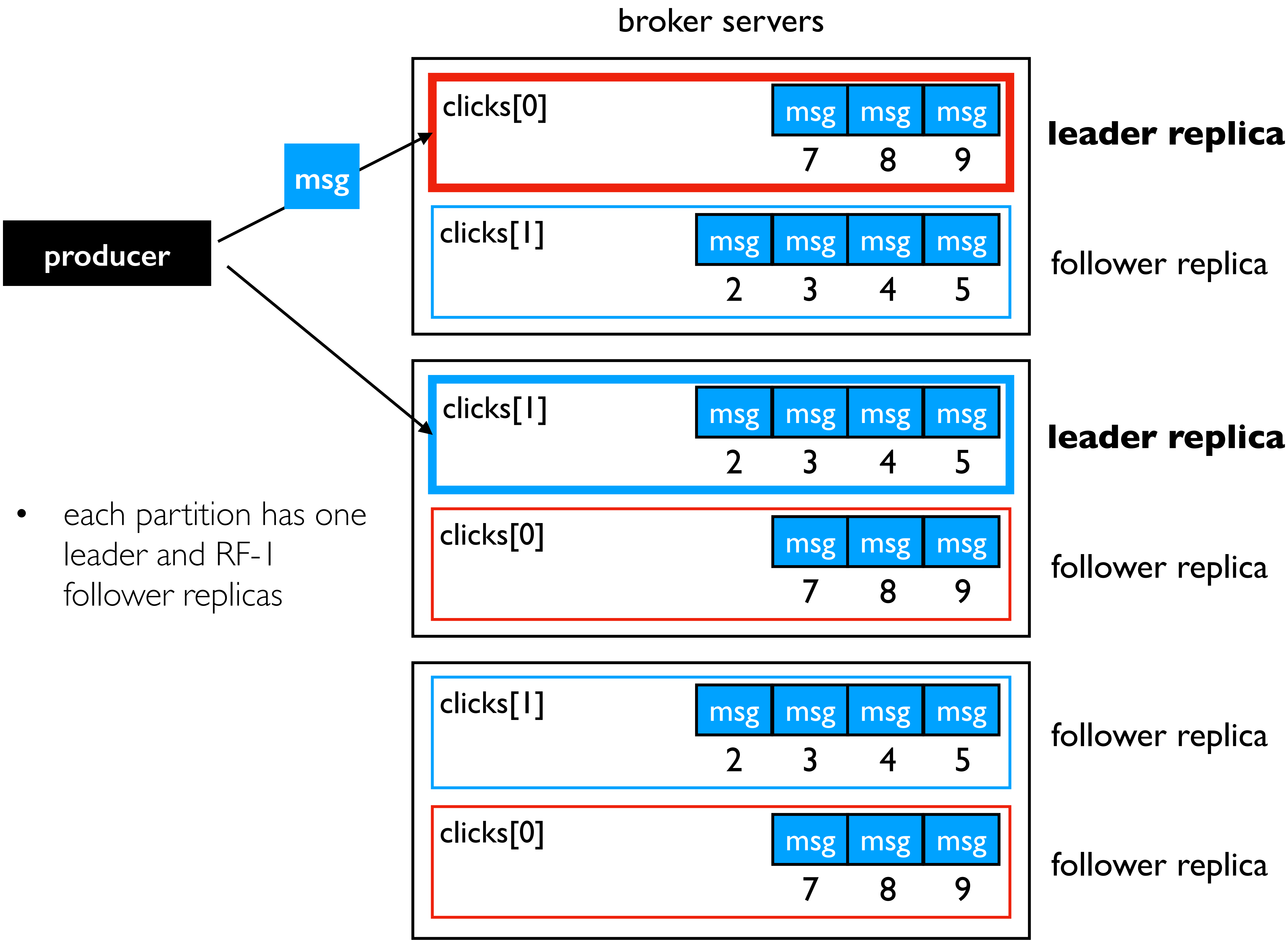
Fault Tolerance

Exactly-Once Semantics

Three brokers, 2 partitions, replication factor=1

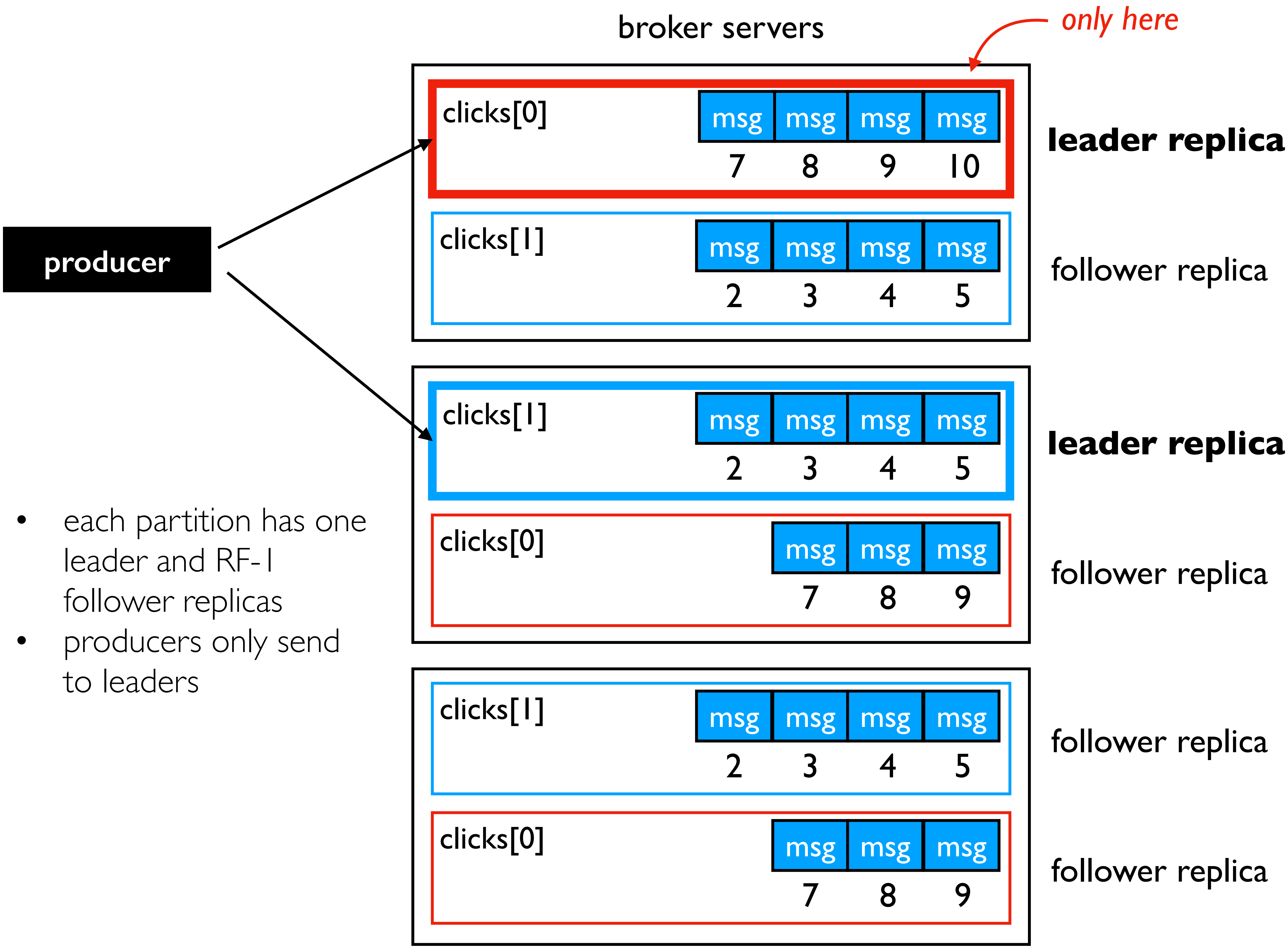


Three brokers, 2 partitions, replication factor=3



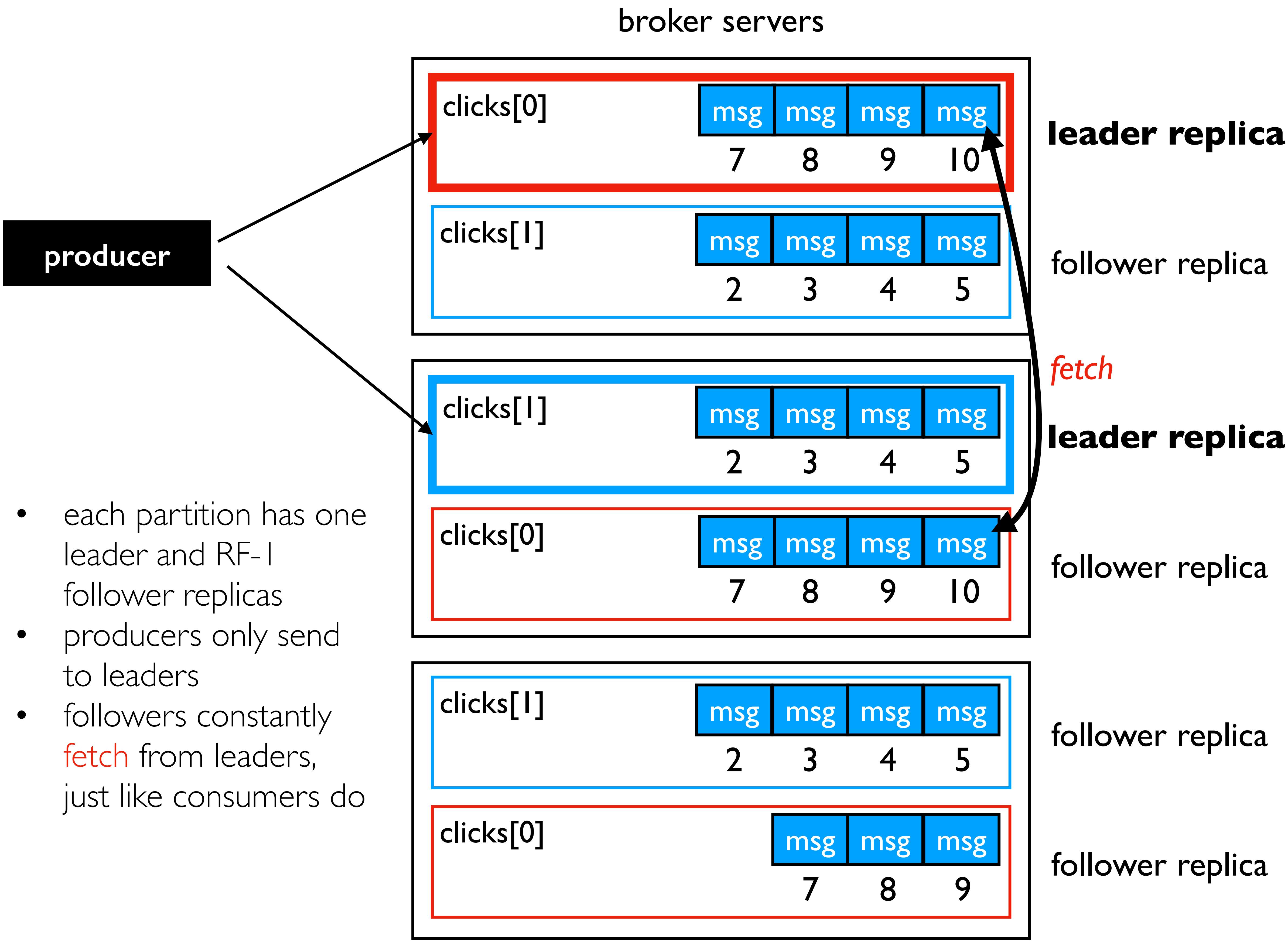
- each partition has one leader and RF-1 follower replicas

Three brokers, 2 partitions, replication factor=3



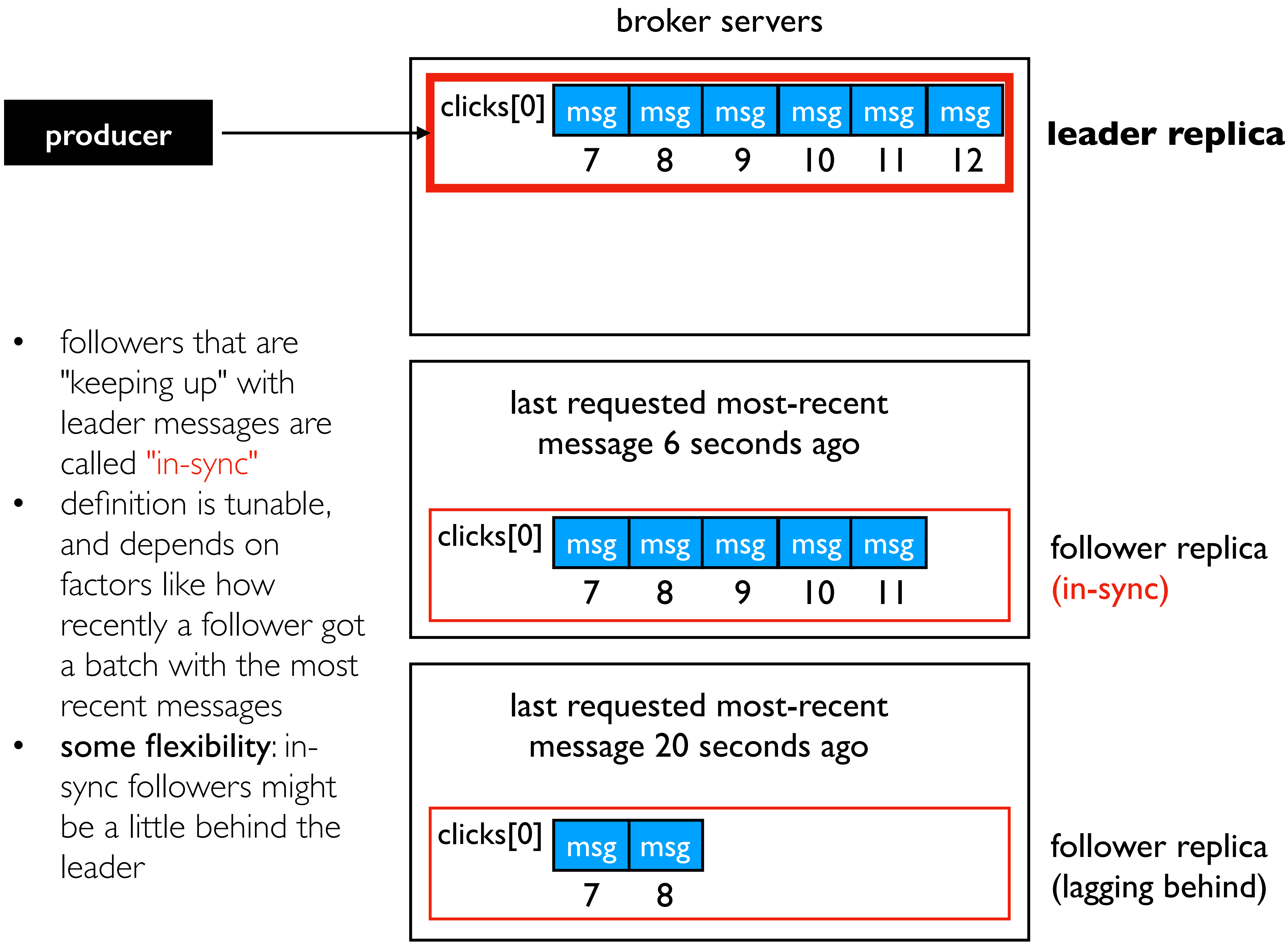
- each partition has one leader and RF-1 follower replicas
- producers only send to leaders

Fetch Requests



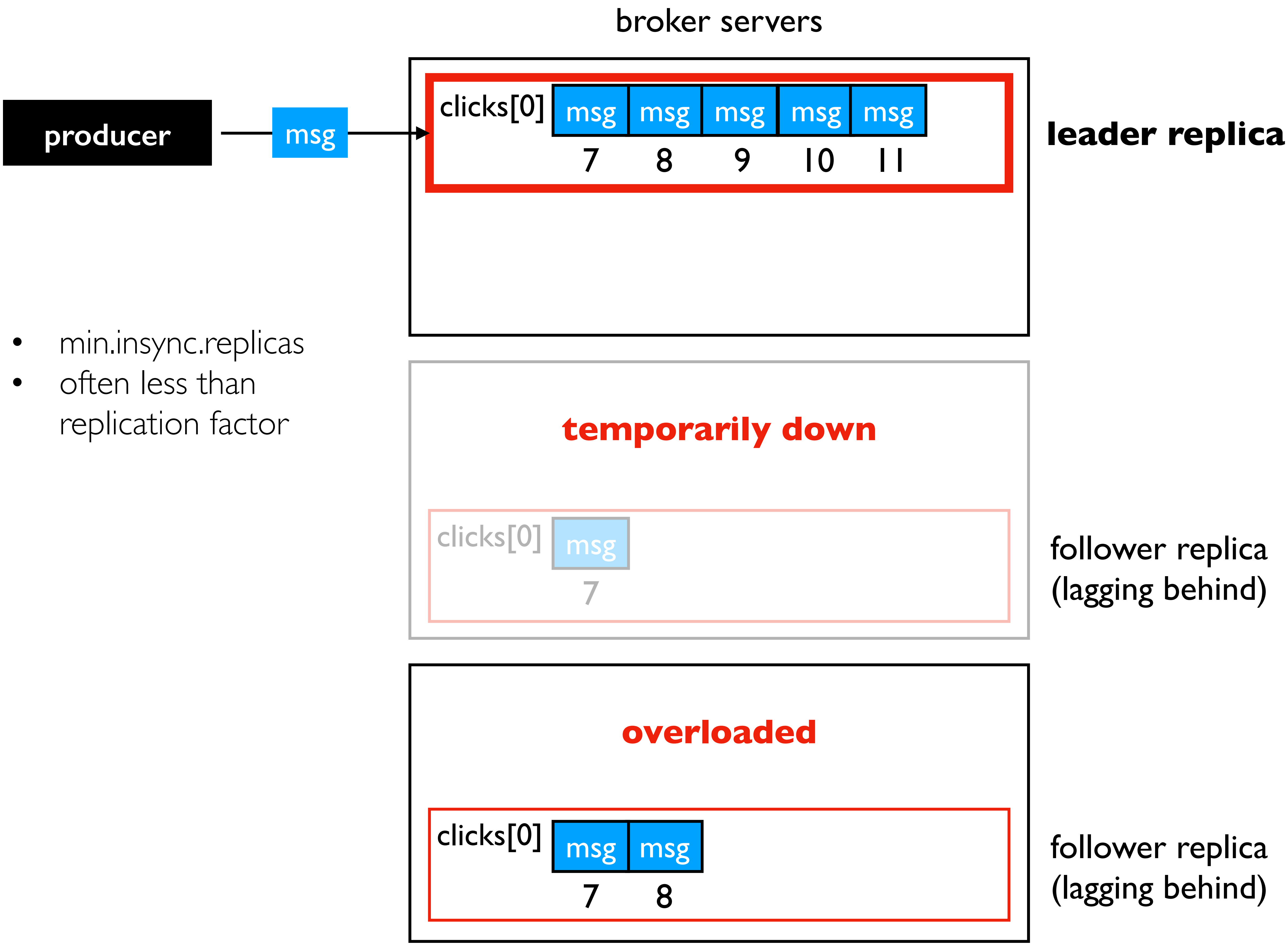
- each partition has one leader and RF-1 follower replicas
- producers only send to leaders
- followers constantly fetch from leaders, just like consumers do

Followers: In-Sync vs. Lagging Too Far Behind



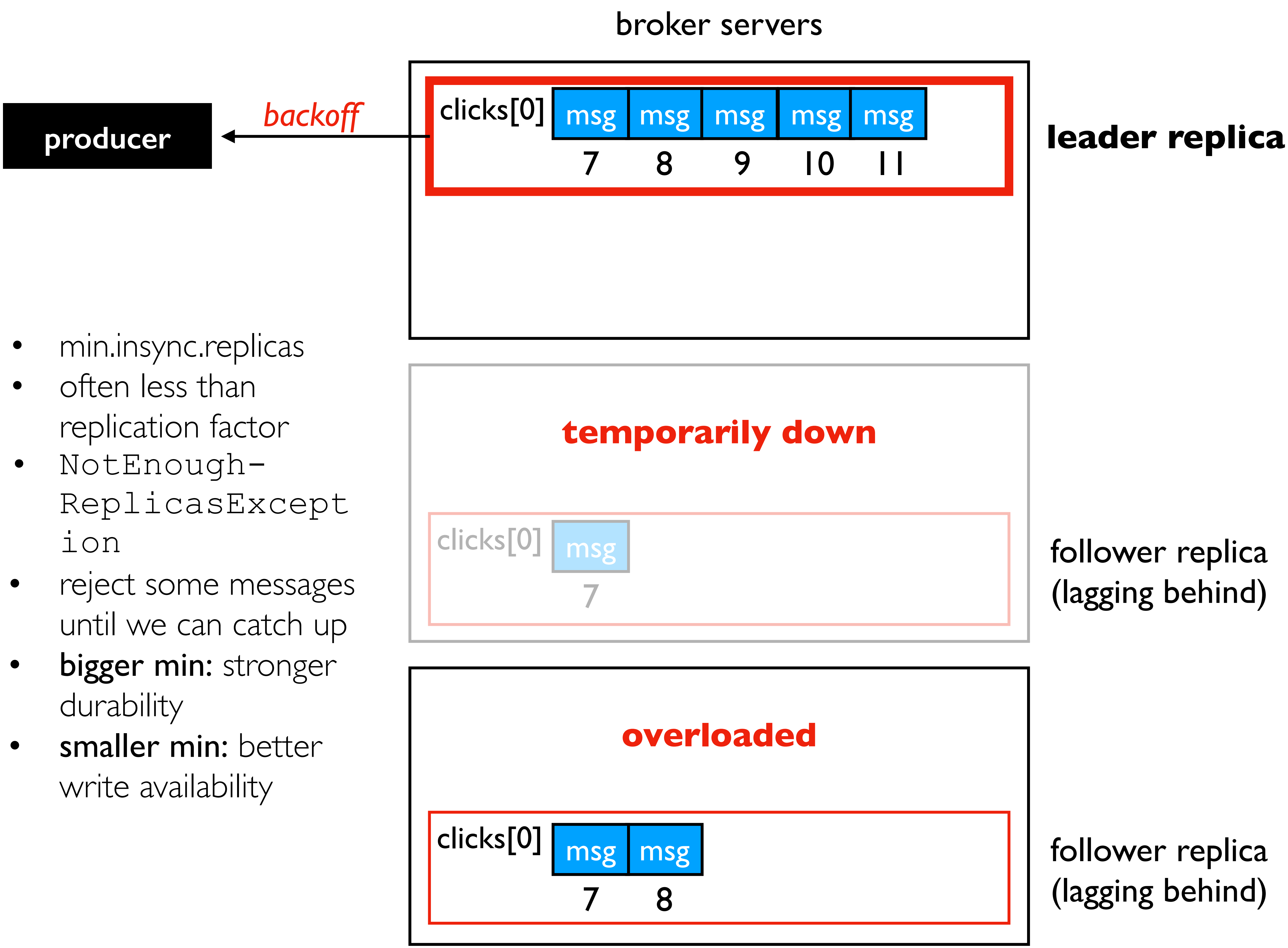
- followers that are "keeping up" with leader messages are called "in-sync"
- definition is tunable, and depends on factors like how recently a follower got a batch with the most recent messages
- **some flexibility**: in-sync followers might be a little behind the leader

Minimum In-Sync Replicas (Assume 2 Here)



- min.insync.replicas
- often less than replication factor

Backoff: Not Enough Replicas Exception



- min.insync.replicas
- often less than replication factor
- `NotEnough-ReplicasException`
- reject some messages until we can catch up
- **bigger min:** stronger durability
- **smaller min:** better write availability

Outline: Kafka Reliability

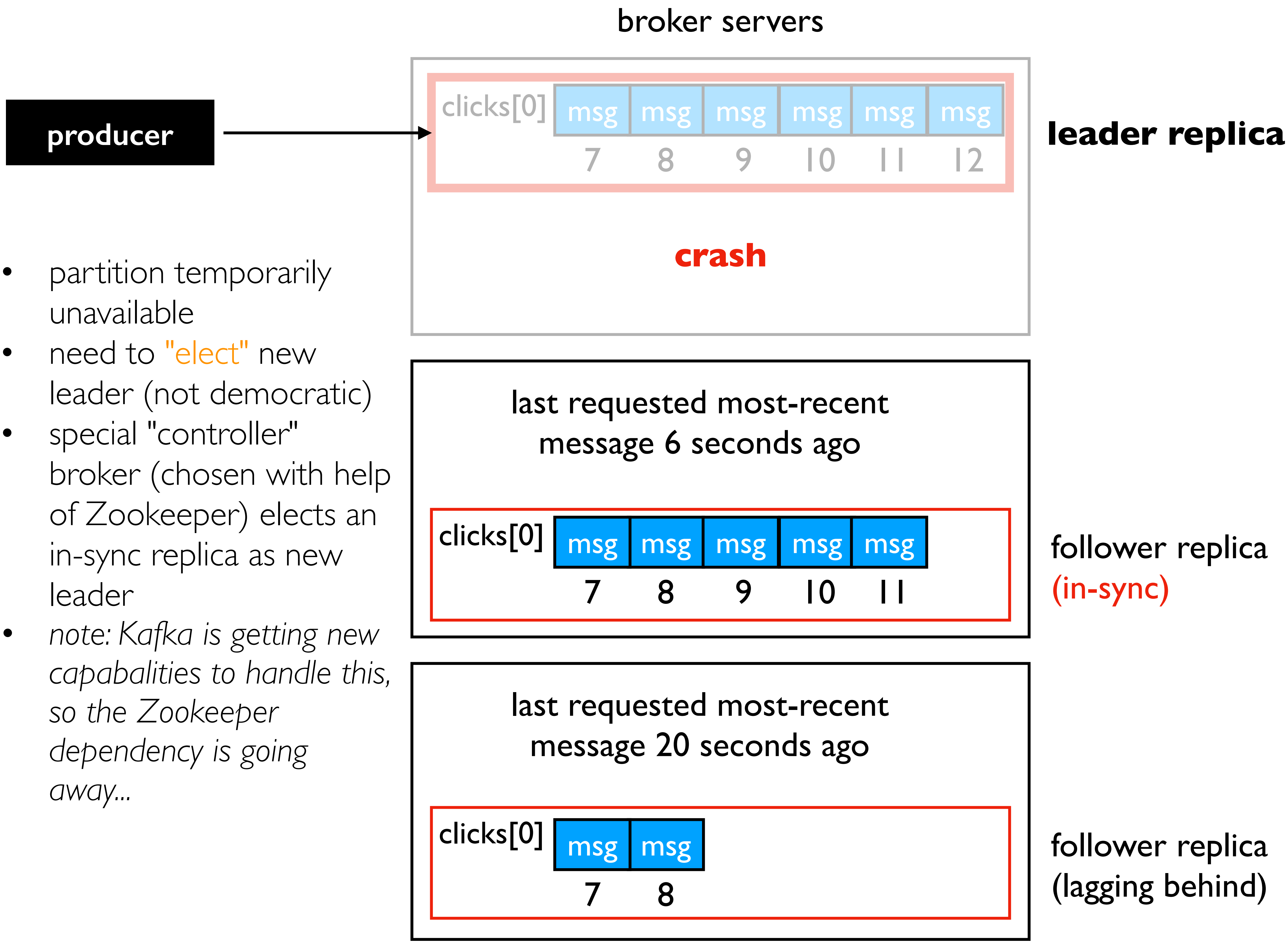
Kafka Replication

Fault Tolerance

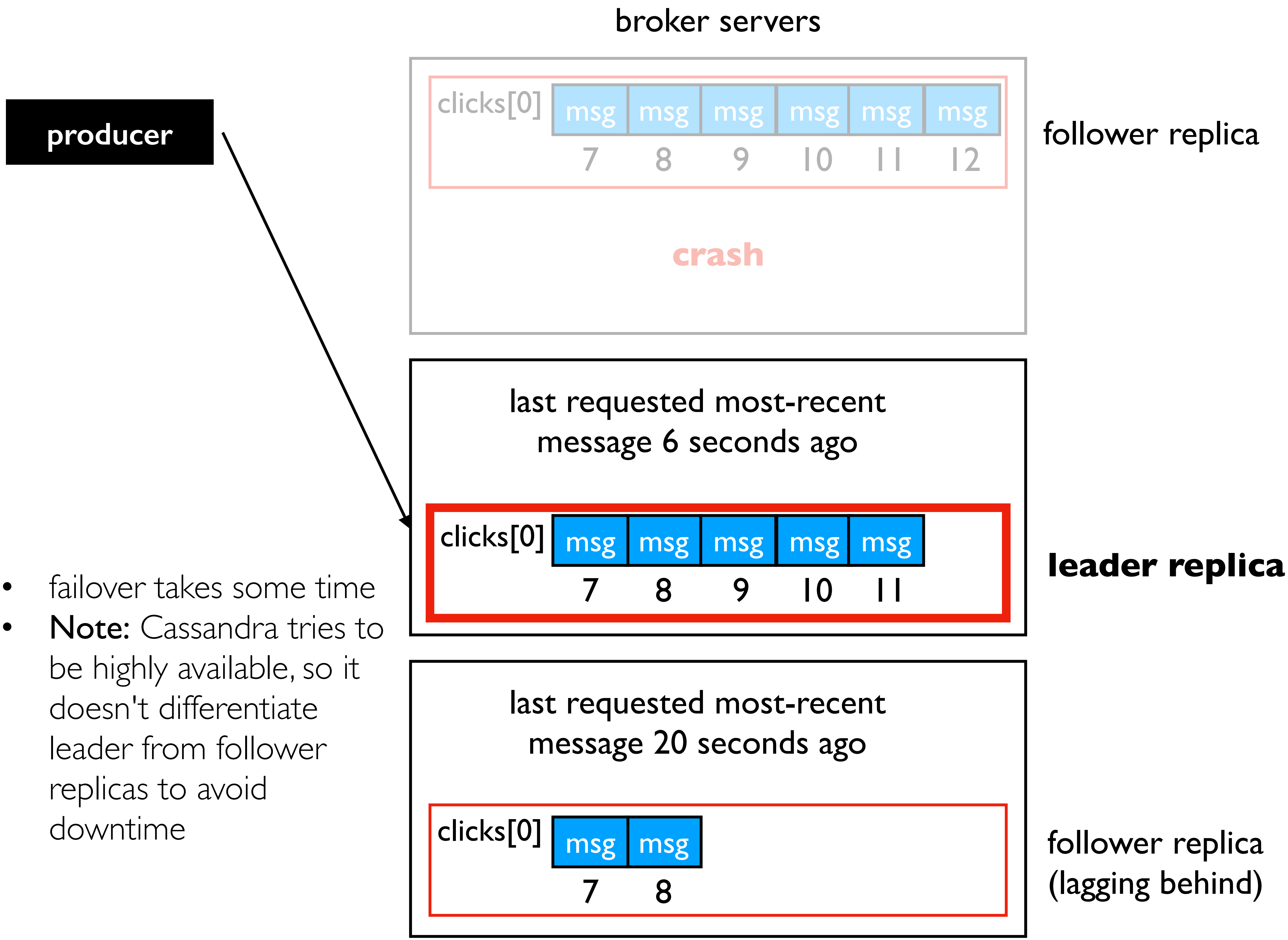
Exactly-Once Semantics

What if the leader fails? Elect a new one!

- partition temporarily unavailable
- need to "elect" new leader (not democratic)
- special "controller" broker (chosen with help of Zookeeper) elects an in-sync replica as new leader
- *note: Kafka is getting new capabalities to handle this, so the Zookeeper dependency is going away...*

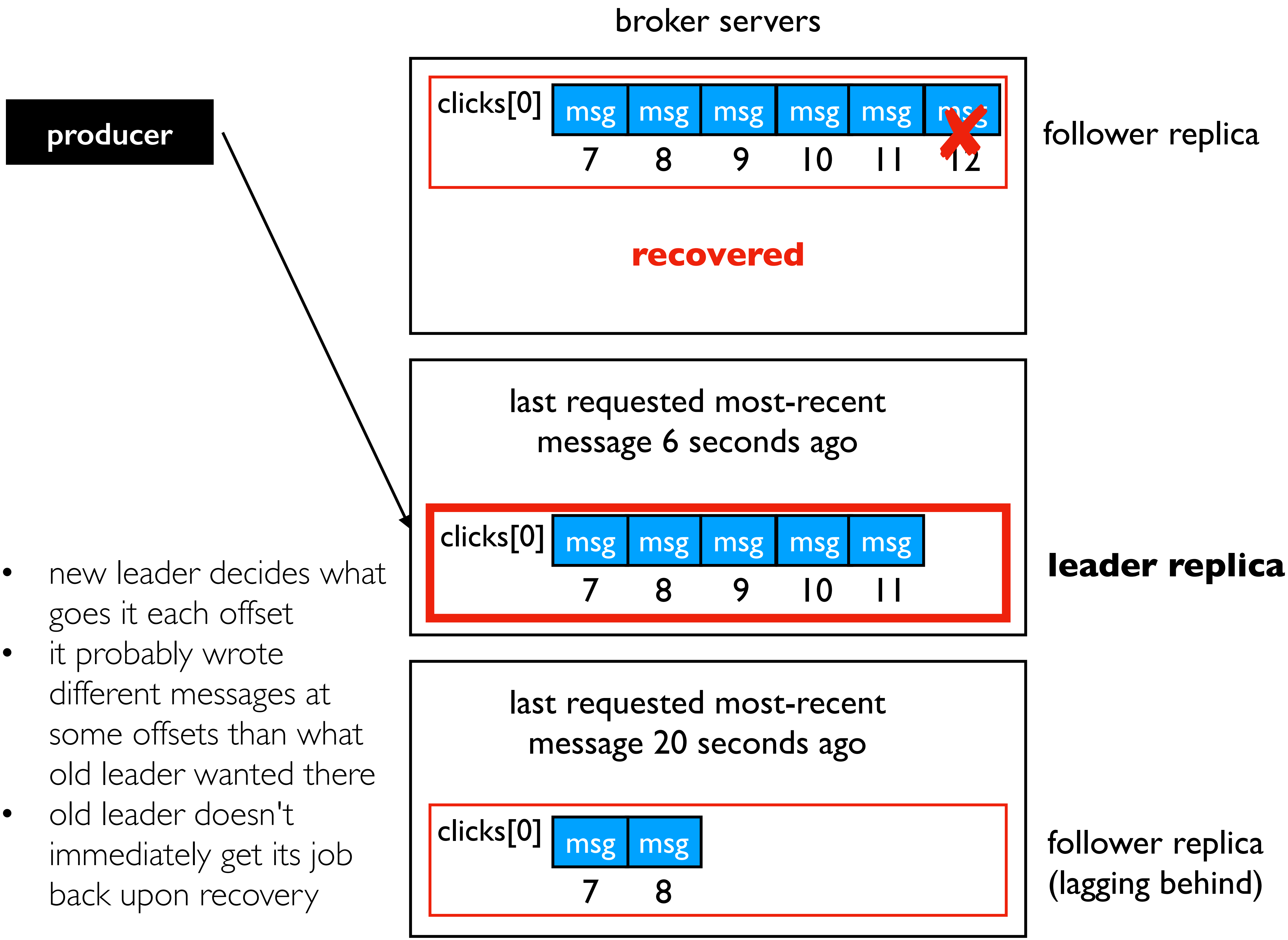


Kafka Replica Failover



- failover takes some time
- **Note:** Cassandra tries to be highly available, so it doesn't differentiate leader from follower replicas to avoid downtime

Some Messages Seen by Old Leader Lost



- new leader decides what goes it each offset
- it probably wrote different messages at some offsets than what old leader wanted there
- old leader doesn't immediately get its job back upon recovery

Review "Committed": WhatsApp Acks Example

How to check read receipts

Copy link

Android iPhone KaiOS

Check marks will appear next to each message you send. Here's what each one indicates:

- ✓ The message was successfully sent.
- ✓✓ The message was successfully delivered to the recipient's phone or any of their linked devices.
- ✓✓ The recipient has read your message.

https://faq.whatsapp.com/665923838265756

these are examples of "acks" (acknowledgements)

In distributed storage systems/databases, an ack means our data is *committed*.

"Committed" means our data is "safe", even if bad things happen. The definition varies system to system, based on what bad things are considered. For example:

- a node could hang until rebooted; a node's disk could permanently fail
- a rack could lose power; a data center could be destroyed

In Kafka's leader/follower replica design, what are some "bad things" we might worry about?

Kafka: Committed Messages

Messages are "commited" when written to ALL in-sync replicas.

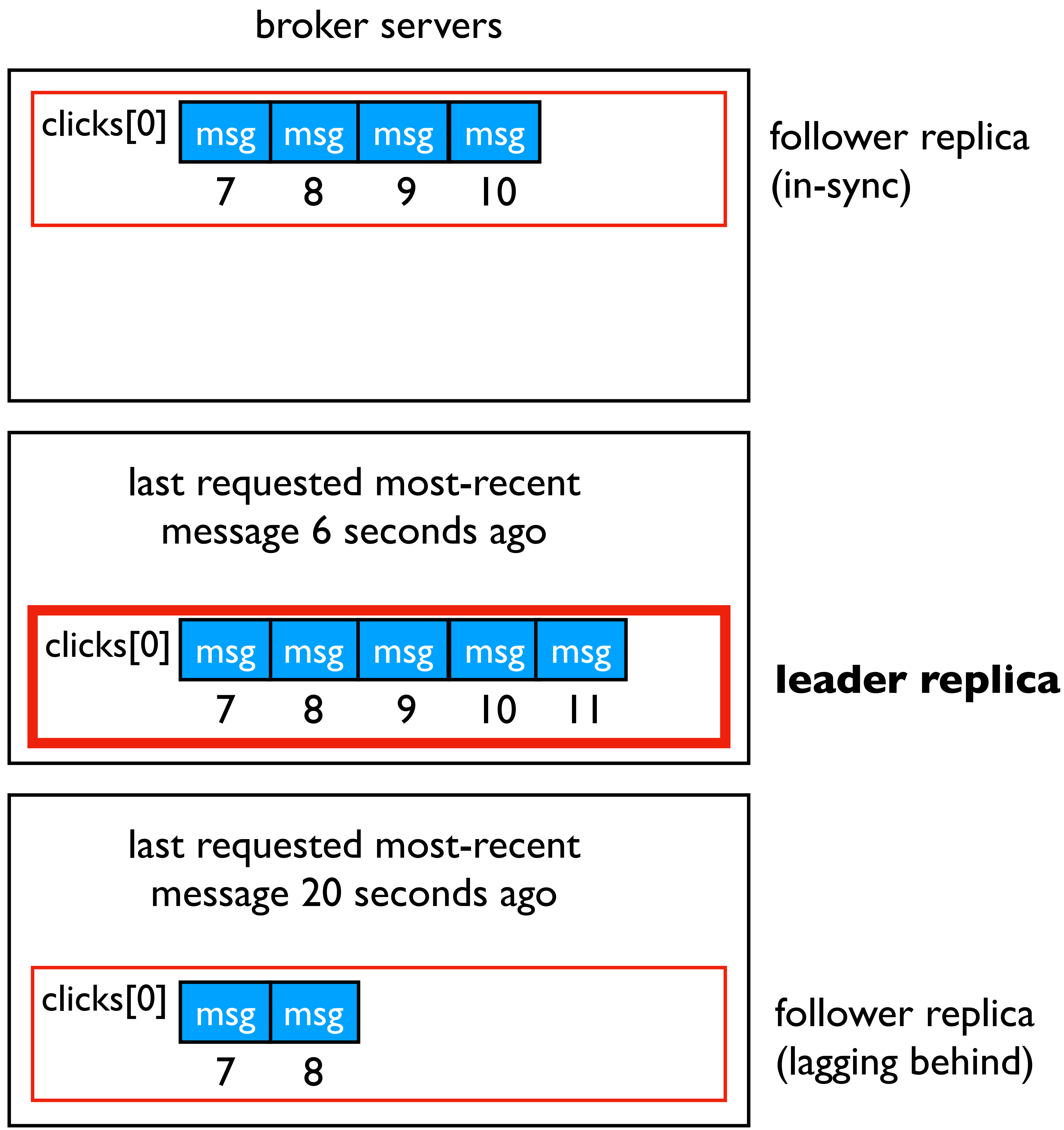
Depending on how many are in-sync, the strength of the guarantee can vary, but min.insync.replicas lets us specify a worst case.

If number of concurrent broker failures < min.insync.replicas, then our committed data is safe, even if the leader fails (because we can elect another in-sync replica, and all in-sync replicas have all committed data).

Committed Messages

What is committed?

- assume RF=3 and minimum in-sync=2
- is message 8 commited?
- message 10?
- message 11?



TopHat

Working with Committed Data

How can we avoid "anomalies" (unexpected system behavior) by taking advantage of commited data?

Example 1: Write Anomaly

Scenario:
- producer writes a message
- produce receives an ACK (acknowledgement) from the broker
- consumers never see the message

Cause: maybe the leader sent an ACK back, then crashed, before replicating the message to the followers.

How to avoid it? *Use strong acks.*

Consumer initialization:
- `KafkaConsumer(..., acks=0)`
  don't wait for leader to send back ACK
- `KafkaConsumer(..., acks=1)`
  ACK after leader writes to its own log
- `KafkaConsumer(..., acks="all")`
  ACK after data is committed (slowest but strongest)

If you don't get an ACK that data is commited, usually best to retry in a loop.

Example 2: Read Anomaly

Scenario:

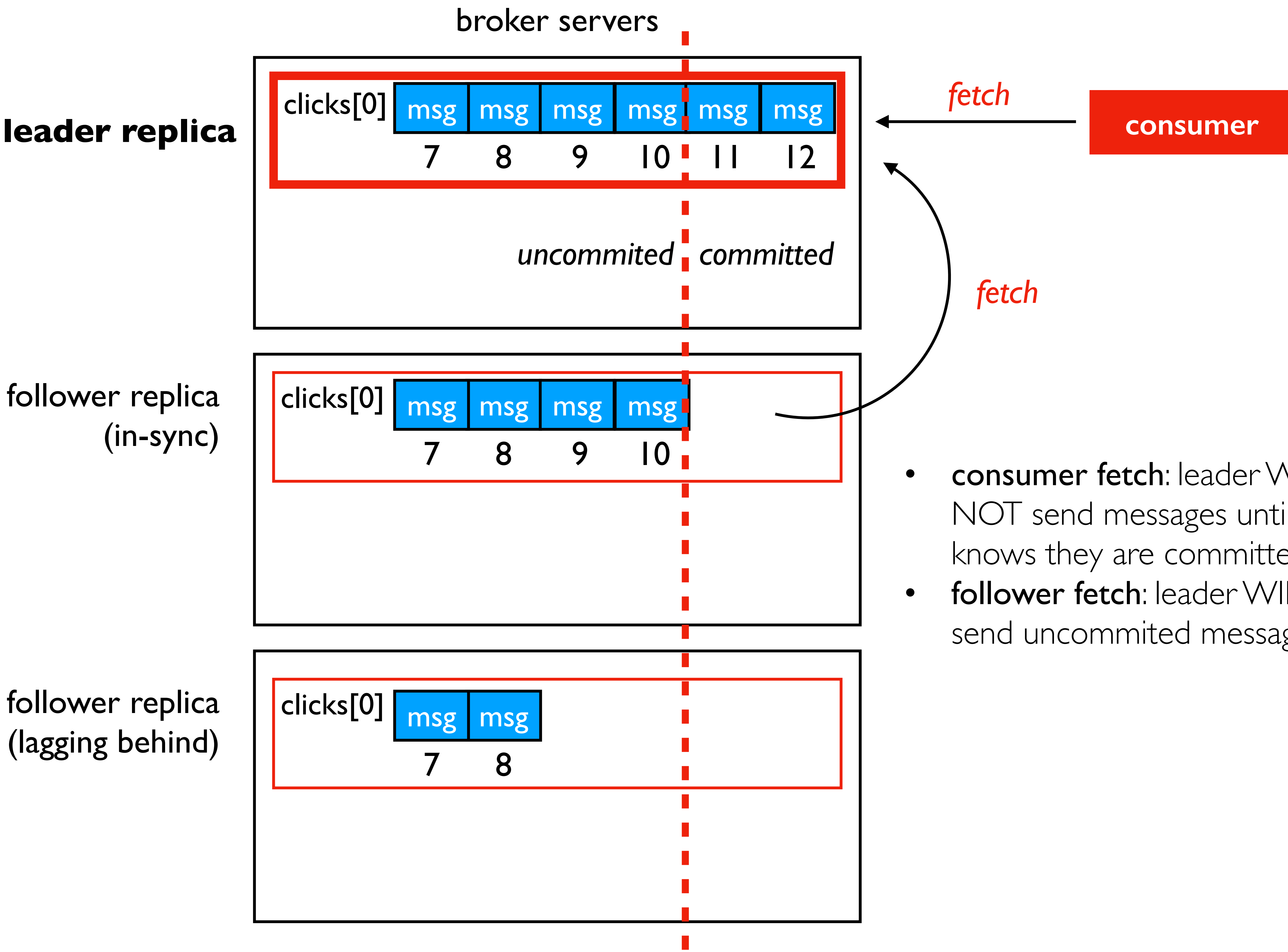
- a consumer reads a message
- there is an attempt to read the message again later (same consumer, or other)
- message is gone, or changed

Cause: maybe the message was consumed from the leader before it was replicated to the followers; then the leader crashed and the new leader doesn't have that message for future consumption.

How to avoid it? *Never read un-committed data.*

The leader does this automatically.

Fetch Behavior: Consumer vs. Follower



- **consumer fetch**: leader WILL NOT send messages until it knows they are committed
- **follower fetch**: leader WILL send uncommited messages

Outline: Kafka Reliability

Kafka Replication

Fault Tolerance

Exactly-Once Semantics

Semantics (Meaning)

Dictionary

Definitions from Oxford Languages · Learn more

se·man·tics

noun

noun: **semantics**; noun: **logical semantics**; noun: **lexical semantics**

the branch of linguistics and logic concerned with meaning. There are a number of branches and

Programming Example:

- **Runtime bug:** the program crashed, there was clearly a problem
- **Semantic bug:** you need to understand the meaning of the results to say whether or not the program behaved correctly

In Systems:

- what does it mean when we get we get an **ACK**, or a **write returns**?
- the meaning depends on how we configured things...

At-most-once semantics

```
producer = KafkaProducer(..., acks=1)
producer.send("my-topic", b"some-value")
```

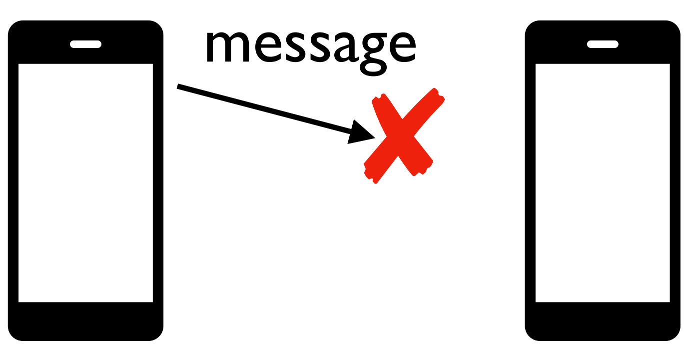
With acks as 0 or 1 and no retry, a successful write means the data was recorded at most once (ideally once, but if the leader crashes at a bad time, maybe zero times).

Using strong ACKs and retry

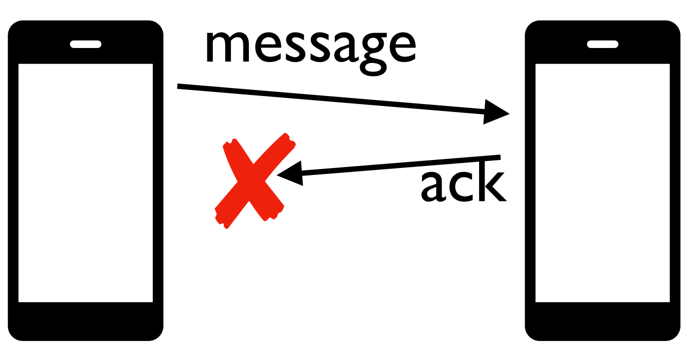
```
producer = KafkaProducer(..., acks="all", retries=10)
producer.send("my-topic", b"some-value")
```

Keep retrying until success (within reason -- for example, 10 times)

Problem: their are two reasons we might not get an ACK:



scenario 1



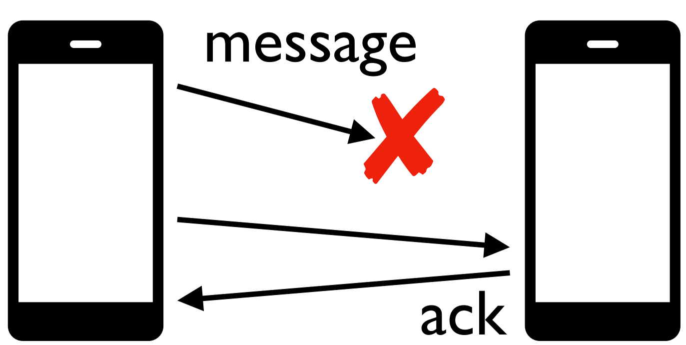
scenario 2

Using strong ACKs and retry

```
producer = KafkaProducer(..., acks="all", retries=10)
producer.send("my-topic", b"some-value")
```

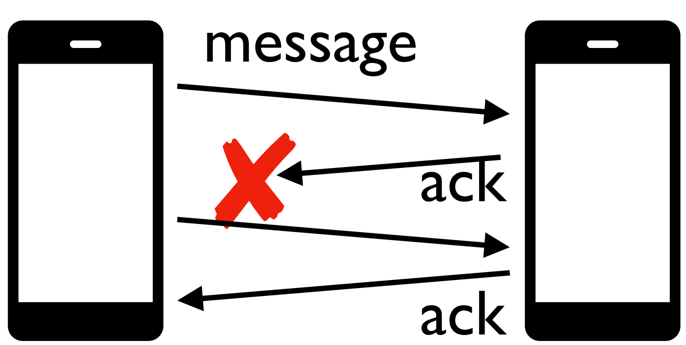
Keep retrying until success (within reason -- for example, 10 times)

Problem: their are two reasons we might not get an ACK:



scenario 1

message written once



scenario 2

message written twice

A strong ACK with retry provides *at-least-once semantics* because we're guaranteed 1+ messages upon success

Are duplicate messages OK?

Yes, if they're idemponent.

"An operation is called idempotent when the effect of performing the operation multiple times is equivalent to the effect of performing the operation a single time"
~ *Operating Systems: Three Easy Pieces, by Arpaci-Dusseau*

```
x = 0
y = 0

def set_x(value):
    global x
    x = value

def inc_y(value):
    global y
    y += value
```

```
# if we just do once, is it the
same?
set_x(123)
set_x(123)
set_x(123)

# if we just do once, is it the
same?
inc_y(3)
inc_y(3)
inc_y(3)
```

TopHat

Supressing Duplicates

With some cleverness, we can make ANYTHING idempotent.

```
y = 0
completed_ops = set()

def inc_y(value, operation_id):
    global y
    if not operation_id in completed_ops:
        y += value
        completed_ops.add(operation_id)

inc_y(3, 1251253)
inc_y(3, 1251253)    # no effect
inc_y(3, 1251253)    # no effect

inc_y(3, 9876)
inc_y(3, 9876)       # no effect

inc_y(1, 5454)
```

Exactly-Once Semantics: Producer Side

Upon a successful write, the message will be considered exactly once (duplicates will be supressed by brokers or consumers).

Producer settings:

- acks="all"
- retry=N
- enable.idempotence=True

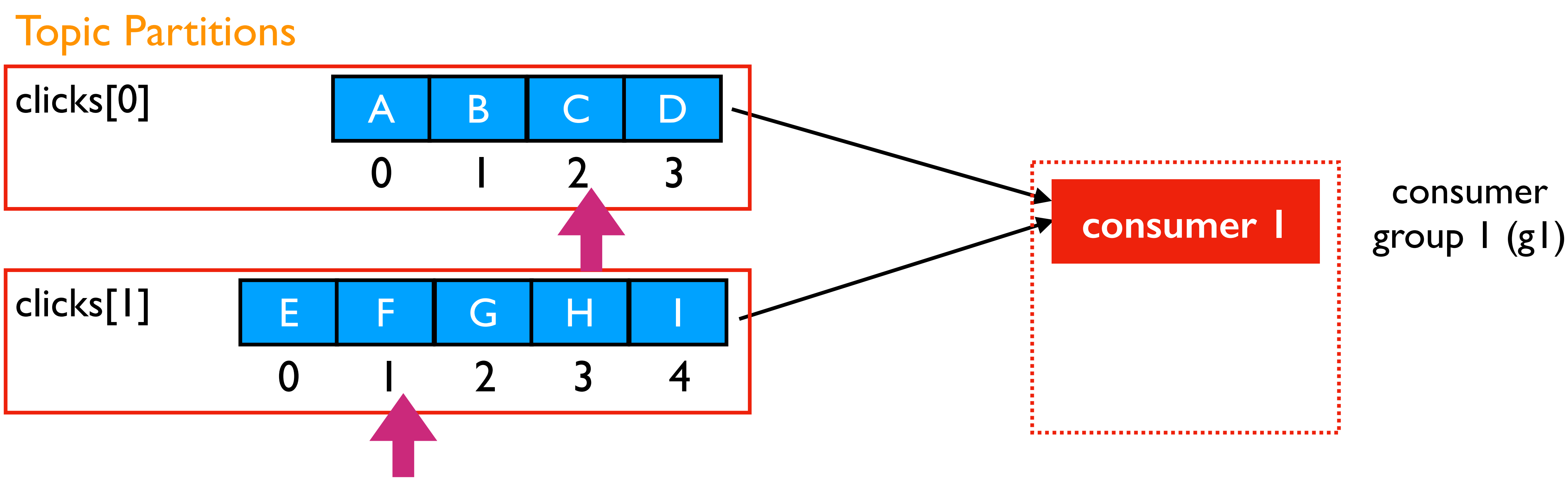
With idempontent enabled, producers automatically generate unique operation IDs and brokers supress duplicates (this has an extra cost).

You can use enable.idempotence in Java, but the kafka-python package doesn't support it.:

- need to handle it yourself
- often, messages have a unique ID anyway, and your consumer code can ignore it
- Example: weather stations that emit one record per day -- if a consumer sees a date for a station it has seen before, ignore it

Exactly-Once Semantics: Consumer Side

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



Suppose consumer dies and is replaced by another in the same group

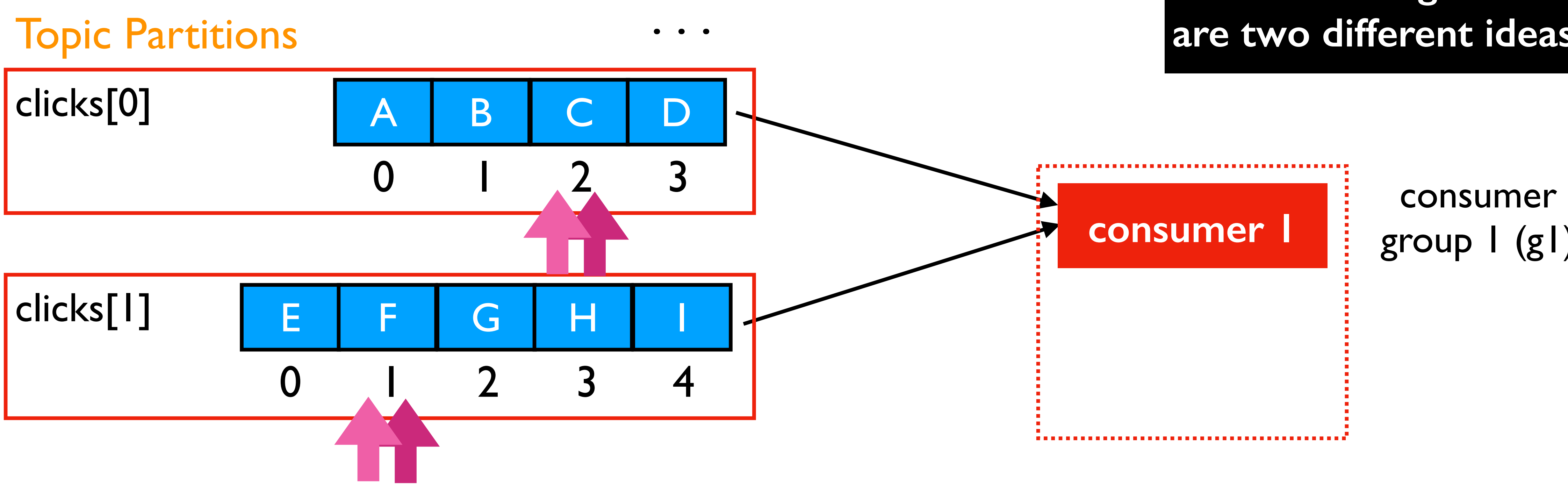
- don't want replacement to miss any messages
- don't want replacement to repeat any processing

	g1 offsets
clicks[0]	2
clicks[1]	1

Exactly-Once Semantics: Consumer Side

```
c = KafkaConsumer("clicks",
                  group_id="g1",
                  enable_auto_commit=True,
                  auto_commit_interval_ms=5000,
                  ...)
while True:
   batch = c.poll(1000)
   ...
```

Note! Committing messages and commiting read offsets are two different ideas.



	g1 offsets
clicks[0]	2
clicks[1]	1

Kafka

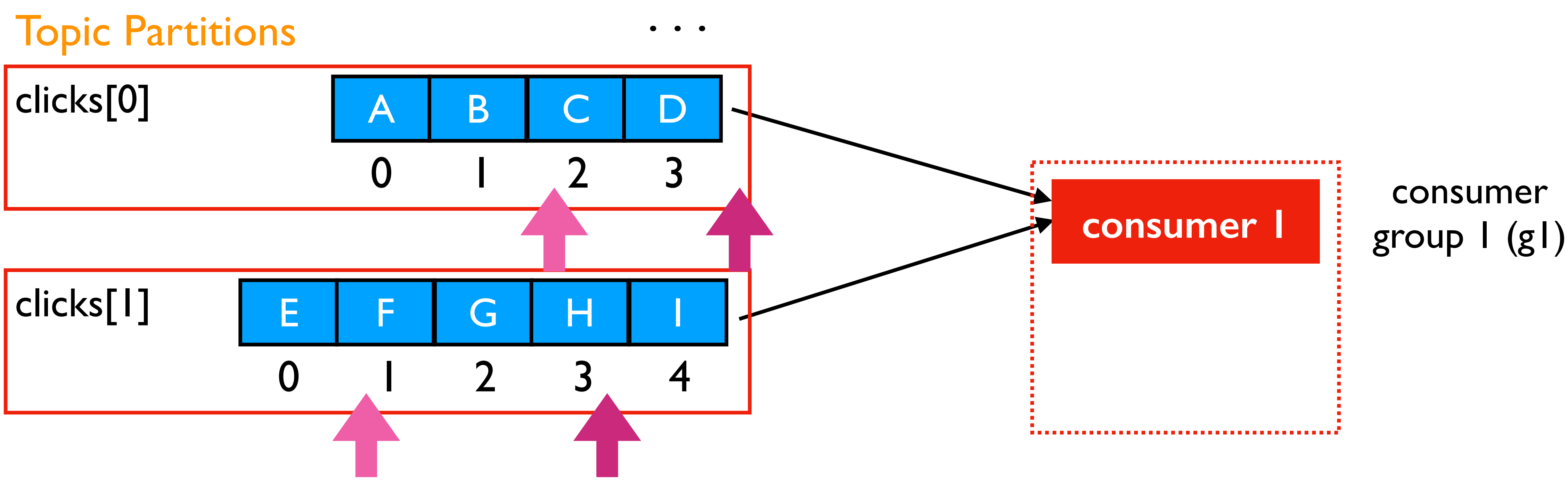
occasionally commit offsets

	g1 offsets
clicks[0]	2
clicks[1]	1

consumer

Exactly-Once Semantics: Consumer Side

```
c = KafkaConsumer("clicks",
                  group_id="g1",
                  enable_auto_commit=True,
                  auto_commit_interval_ms=5000,
                  ...)
while True:
   batch = c.poll(1000)
   ...
```



	g1 offsets
clicks[0]	2
clicks[1]	1

Kafka

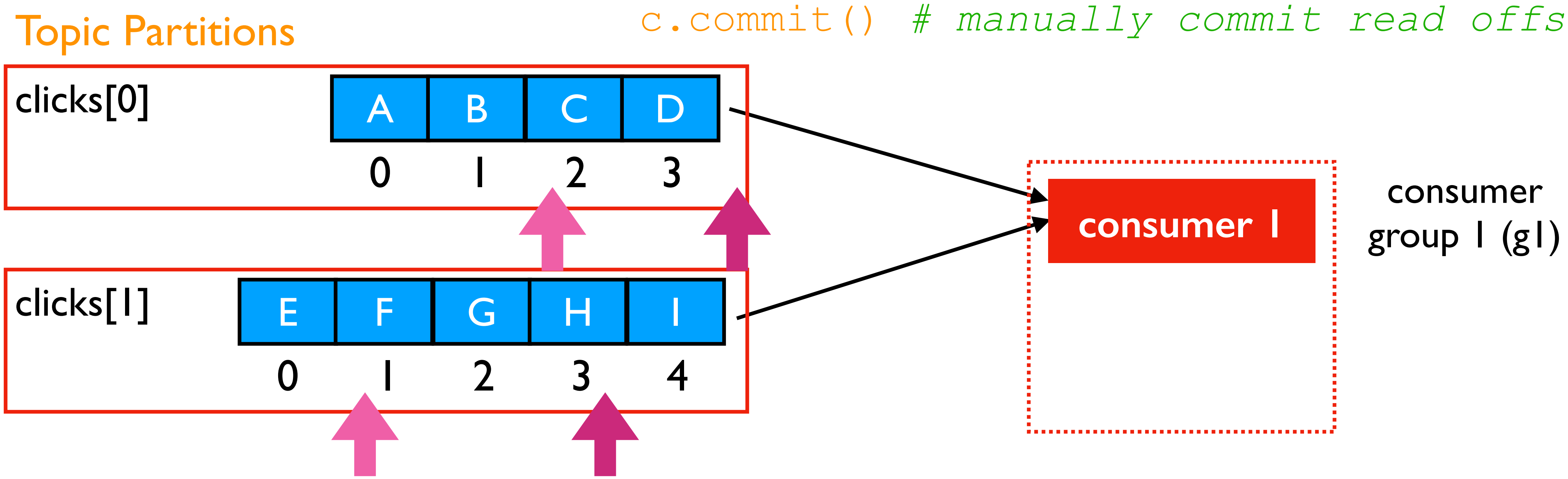
If we crash at a bad time, the offsets the next consumer gets from Kafka will only be approximately correct.

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

consumer

Approach 1: Manually Commit Offsets

```
c = KafkaConsumer("clicks",
                  group_id="g1",
                  enable_auto_commit=False,
                  ...)
while True:
   batch = c.poll(1000)
   ...
   c.commit() # manually commit read offsets
```



	g1 offsets
clicks[0]	2
clicks[1]	1

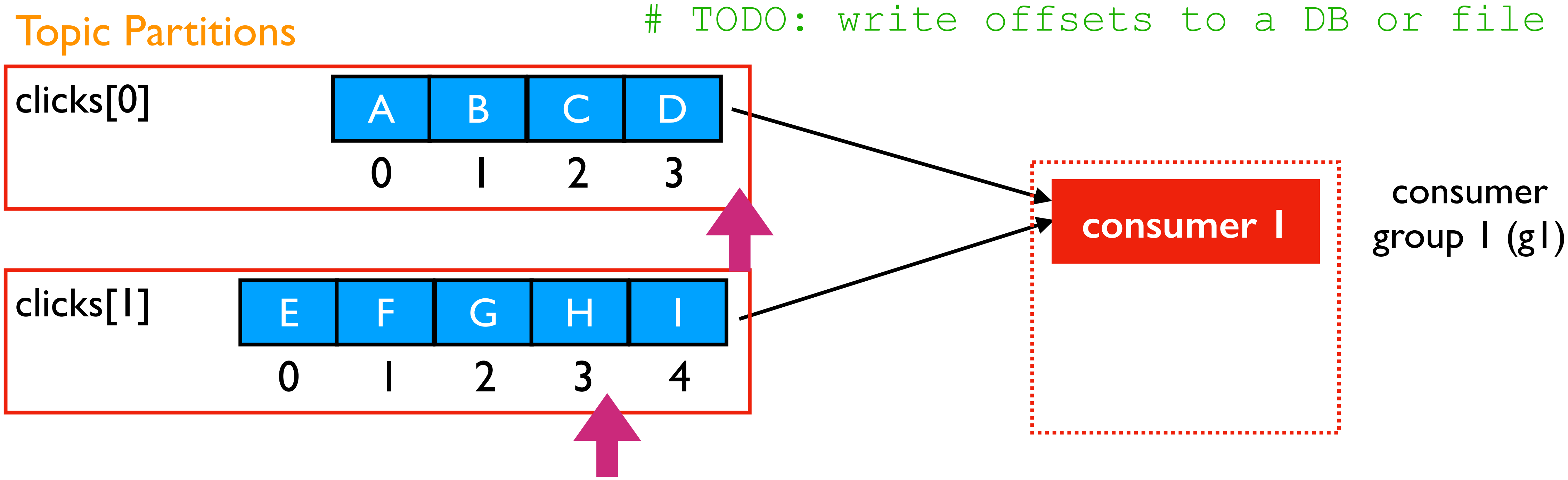
Kafka

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

consumer

Approach 2: Externally Save Commits

```
c = KafkaConsumer("clicks",
                  group_id="g1",
                 ...)
# TODO: seek to previous position
while True:
   batch = c.poll(1000)
   ...
   # TODO: write offsets to a DB or file
```



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

consumer

Conclusion

Every part of the system has a part to play in reliability and exactly-once semantics.

Producer:

- requesting strong acks
- retry
- idempotence

Broker:

- replicating data to followers
- failing over to new leader
- sending acks
- helping producer supress duplicates
- keeping uncommitted data hidden from consumers

Consumer:

- carefully handling read offsets
- sometimes supressing duplicates (if not handled by producers+brokers)

[544] Spark Streaming

Tyler Caraza-Harter

Analysis Demos: Kafka and Spark

Outline: Spark Streaming

DStreams

Grouped Aggregates

Watermarks

Pivoting

Joining

Exactly-Once Semantics

Review Data Lineage: Transformations and Actions

```
data = [
    ("A", 1),
    ("B", 2),
    ("A", 3),
    ("B", 4)
]
```

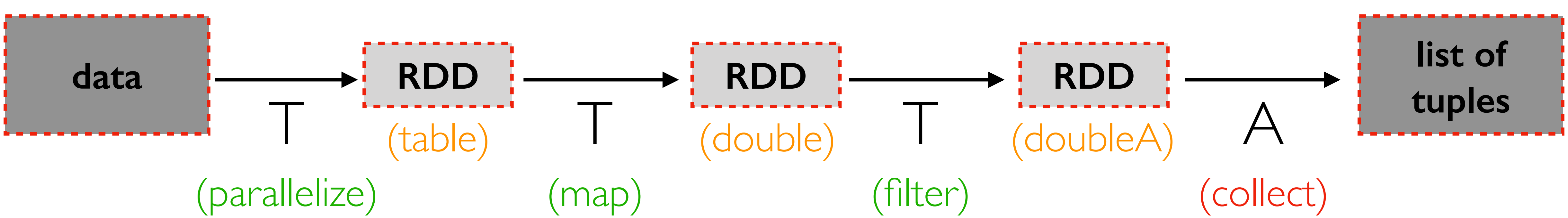
```
def mult2(row):
    return (row[0], row[1] * 2)

def onlyA(row):
    return row[0] == "A"
```

goal: get 2 times the second column wherever the first column is "A"

```
table = sc.parallelize(data)
double = table.map(mult2)
doubleA = double.filter(onlyA)
doubleA.collect()
```

```
[('A', 2),
 ('A', 6)]
```



Handling Data Changes: Re-Calculate Everything

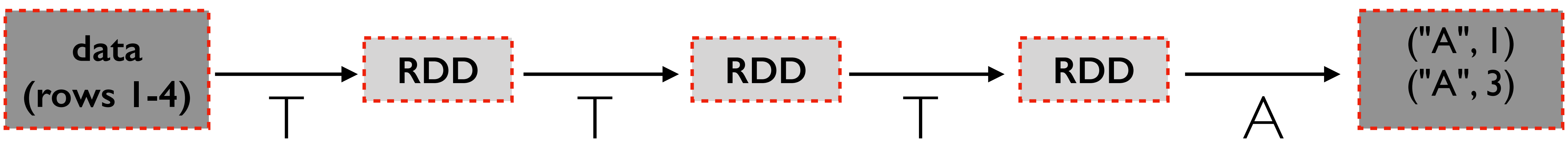
```
data = [
     ("A", 1),
     ("B", 2),
     ("A", 3),
     ("B", 4),
     ("A", 5),
     ("C", 6)
]
```

new data

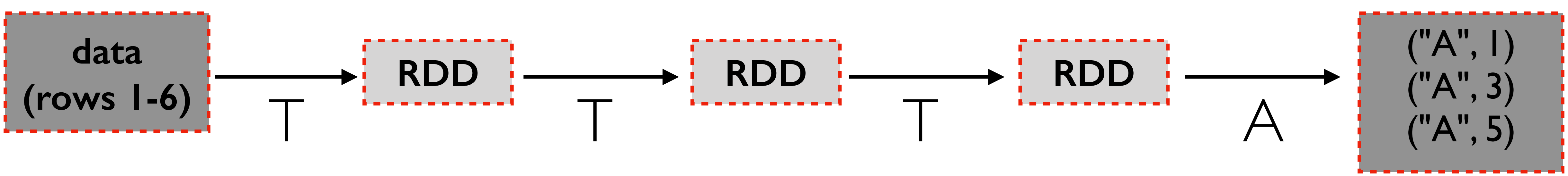
```
def mult2(row):
    return (row[0], row[1] * 2)

def onlyA(row):
    return row[0] == "A"
```

Round 1



Round 2



re-doing work is wasteful!

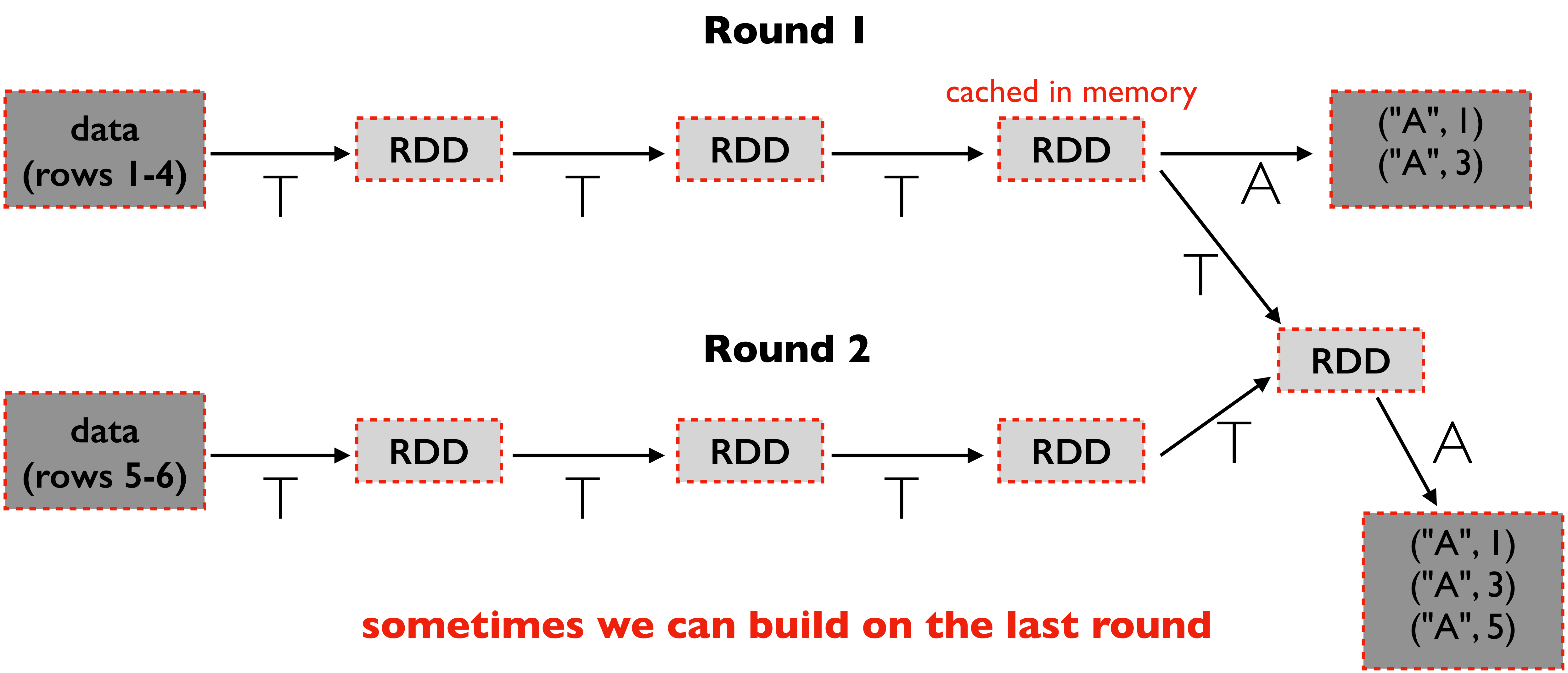
Handling Data Changes: Incremental Computation

```
data = [
       ("A", 1),
       ("B", 2),
       ("A", 3),
       ("B", 4),
       ("A", 5),
       ("C", 6)
]
```

new data

```
def mult2(row):
    return (row[0], row[1] * 2)

def onlyA(row):
    return row[0] == "A"
```



sometimes we can build on the last round

Some DataFrames constantly grow

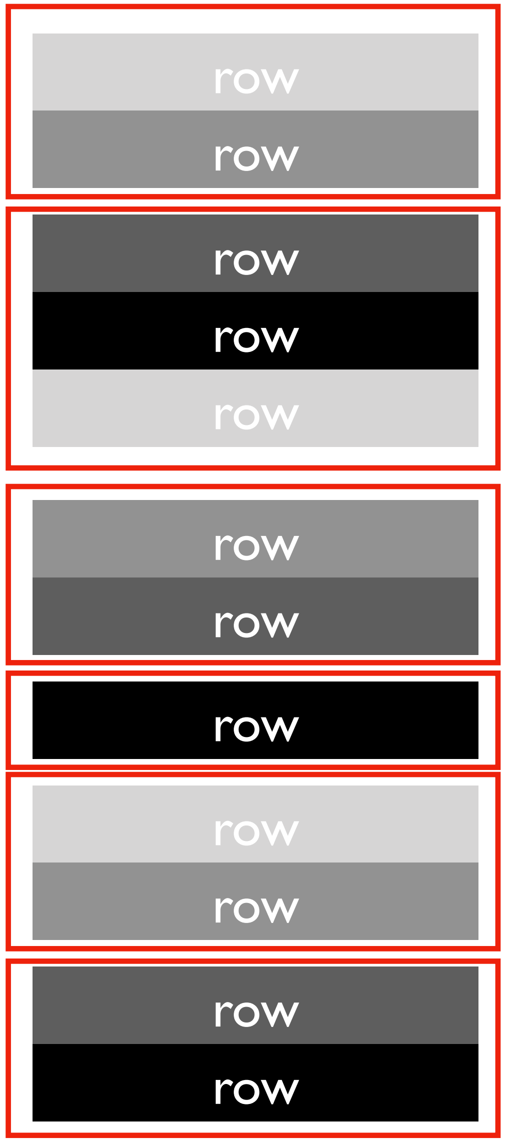
continuously growing table

Mini Batches

continuously growing table

```
trigger(processingTime="12 seconds")
```

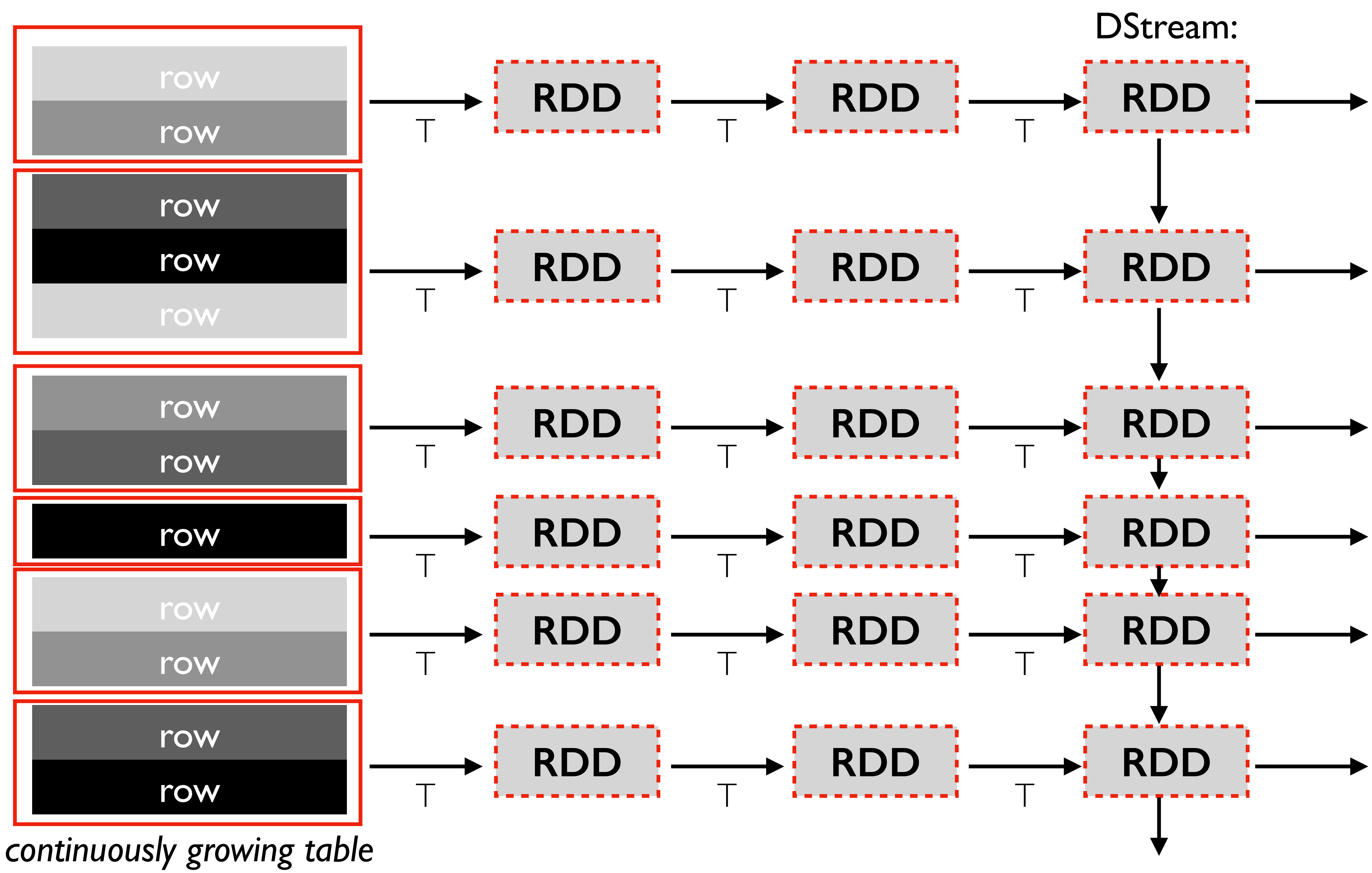
Trigger Frequency



continuously growing table

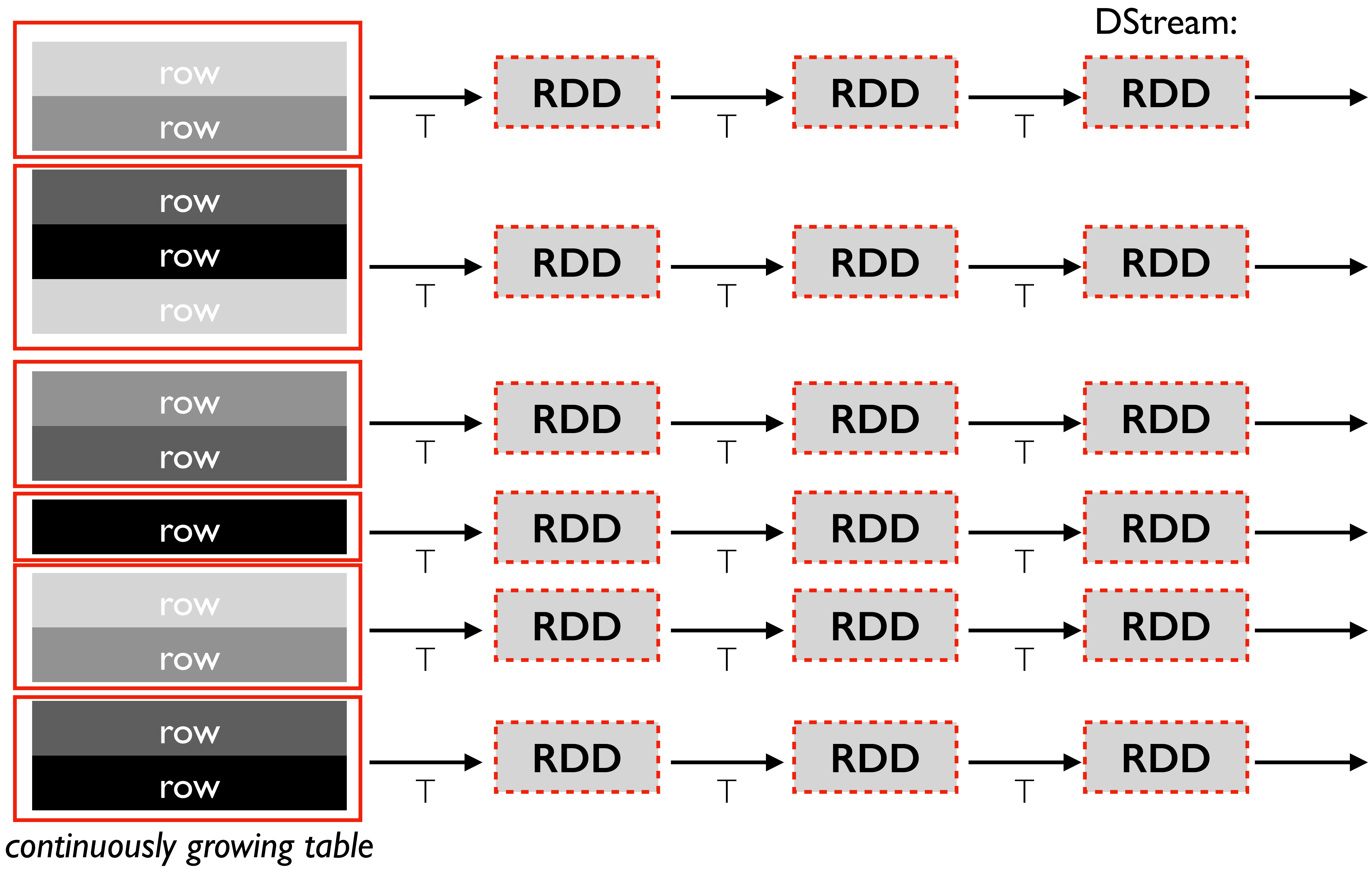
```
trigger(processingTime="4 seconds")
```

DStream (Stateful)



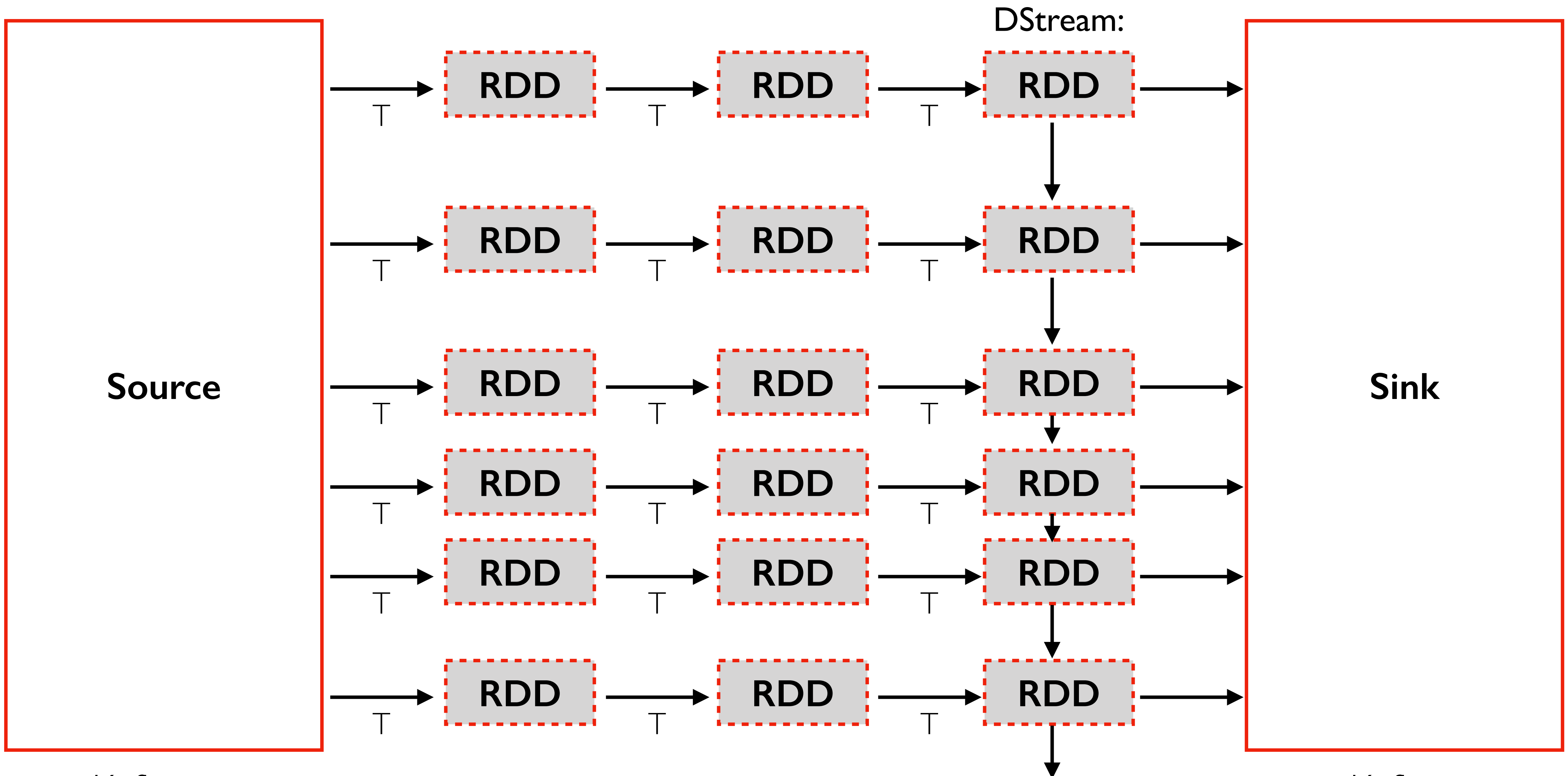
A Spark DStream is a series of RDDs

DStream (Stateless)



If we can compute on each batch without using state from previous computations, it is stateless.

Source => DStream => Sink



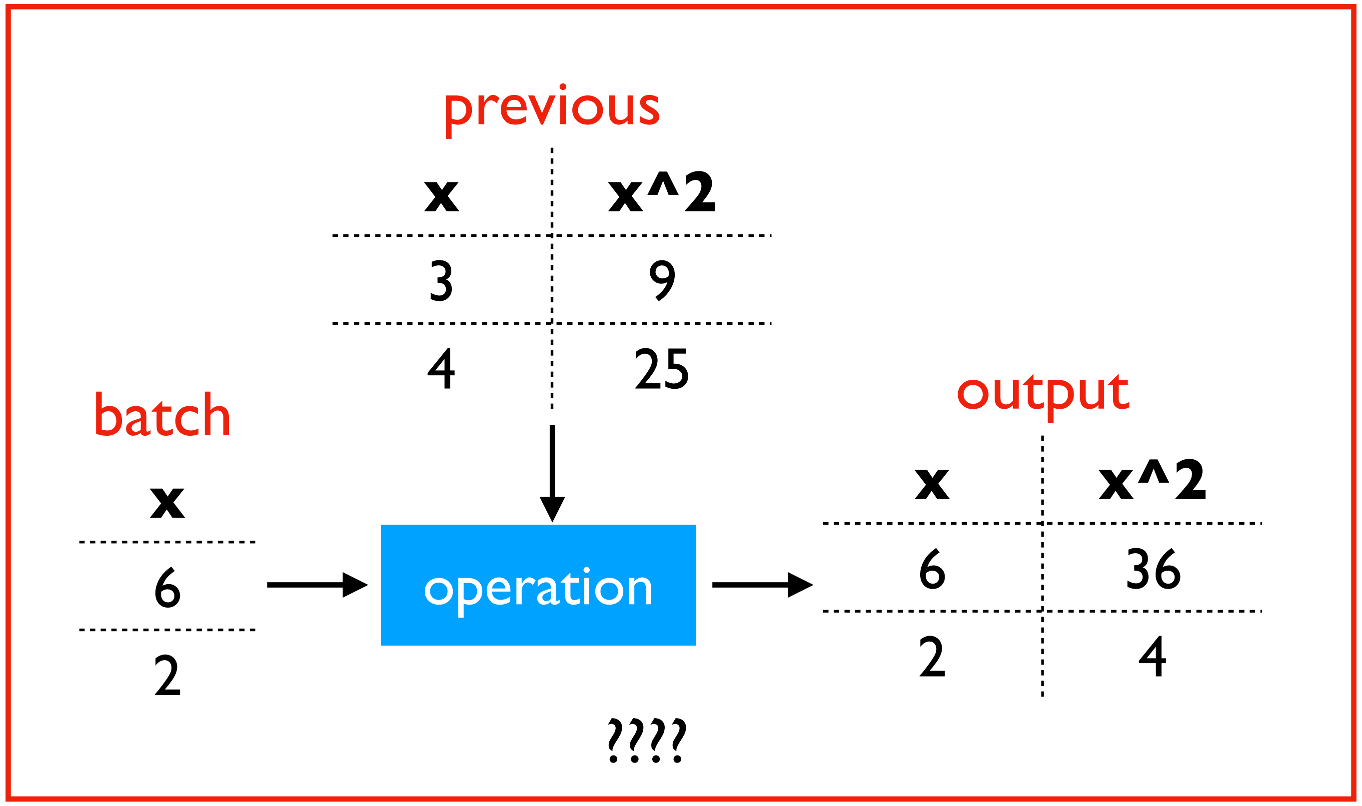
- Kafka
- HDFS files
- Cassandra
- etc.

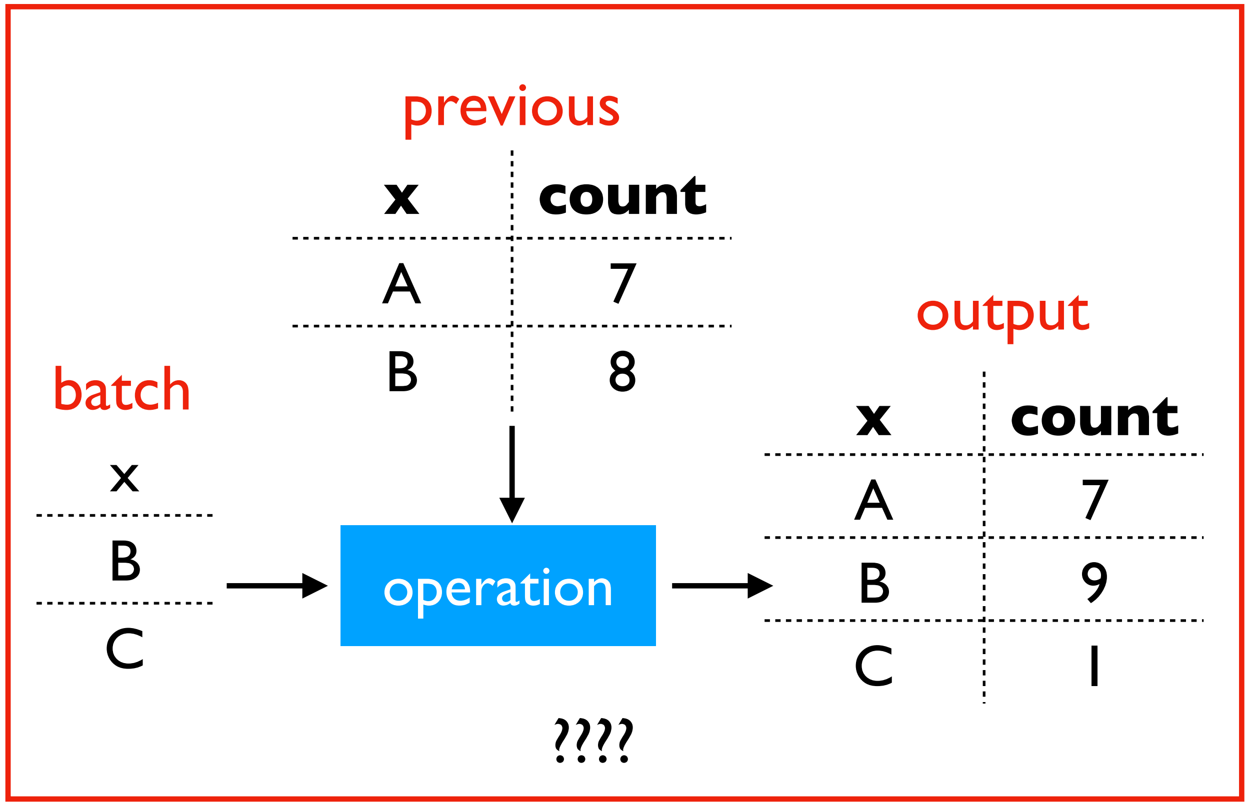
A DStream continuously pulls data from a source, transforms it, and sends it to a sink

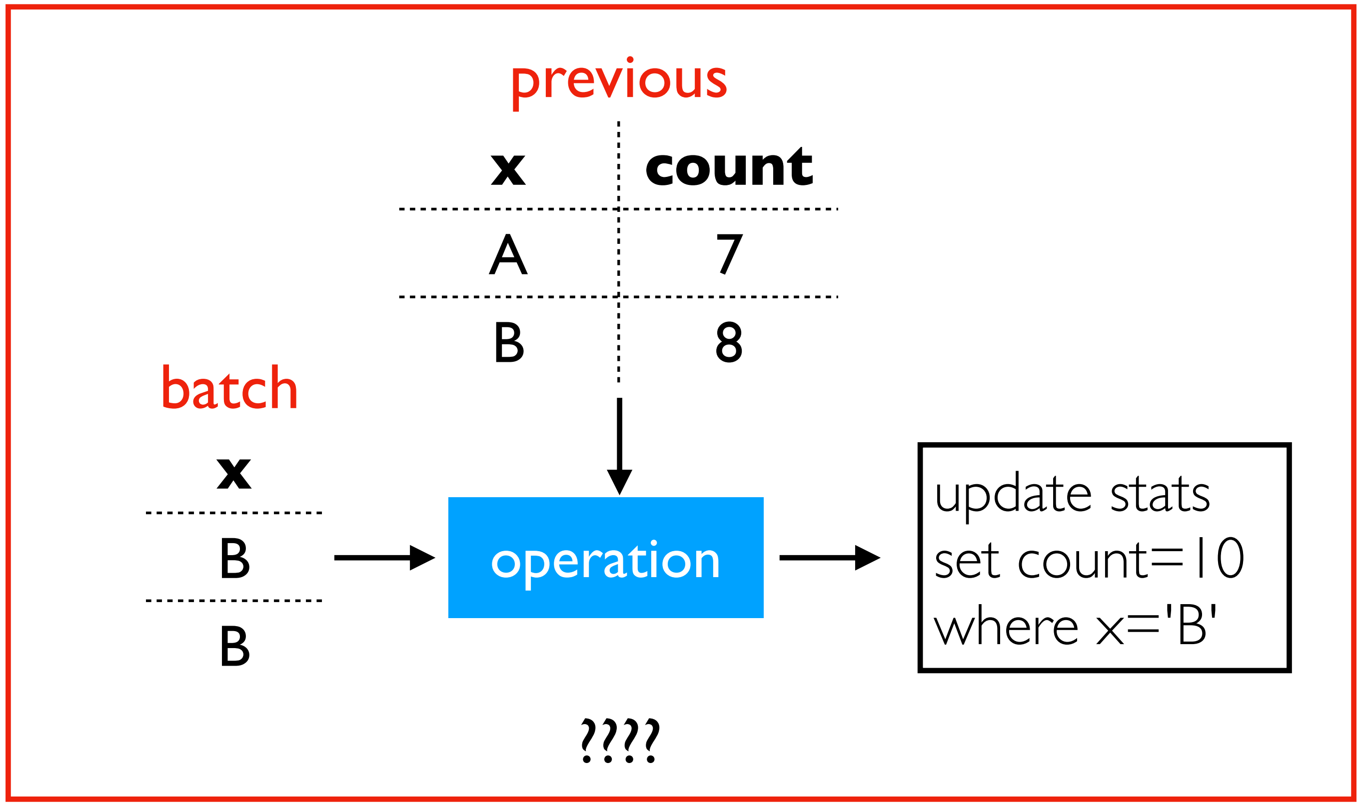
- Kafka
- HDFS files
- console
- etc.

many possible sink/source formats

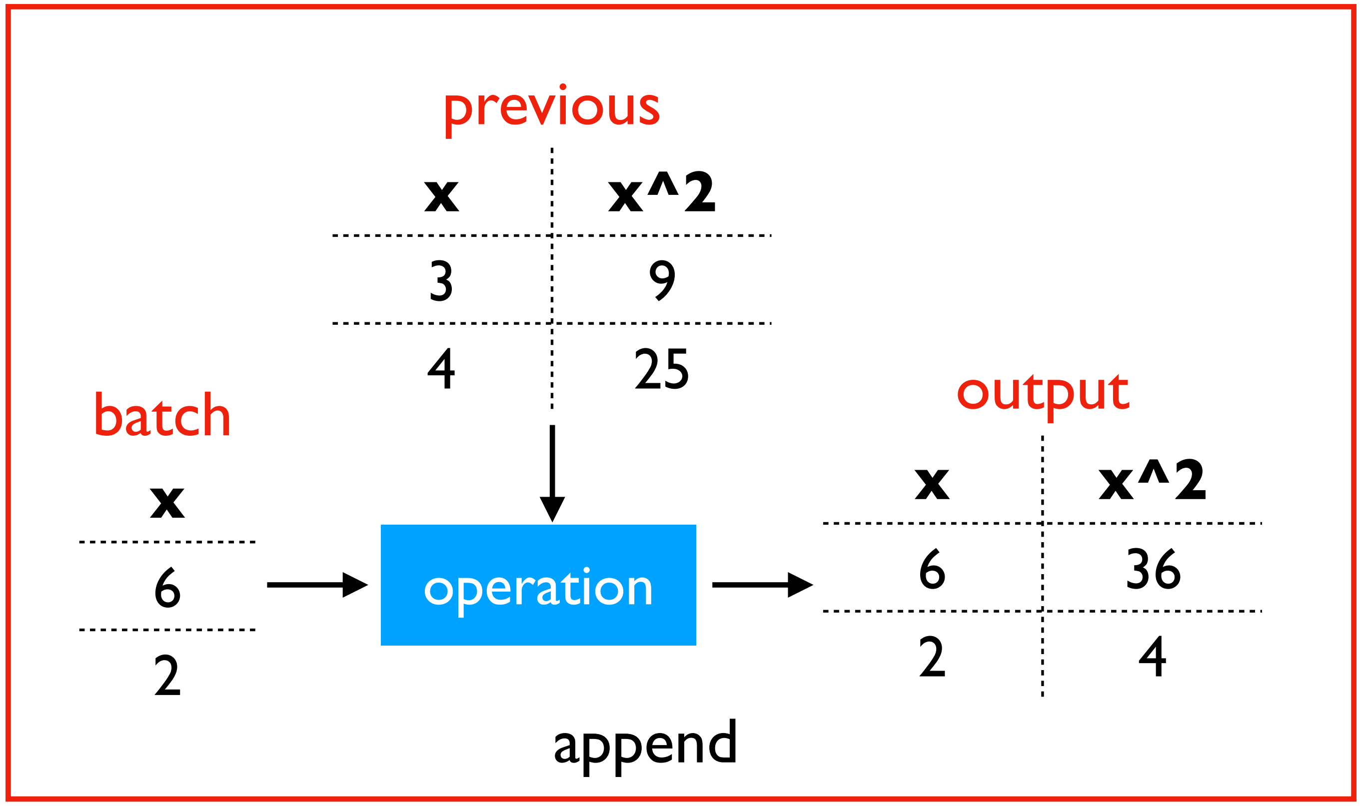
Output Modes: Update, Complete, Append

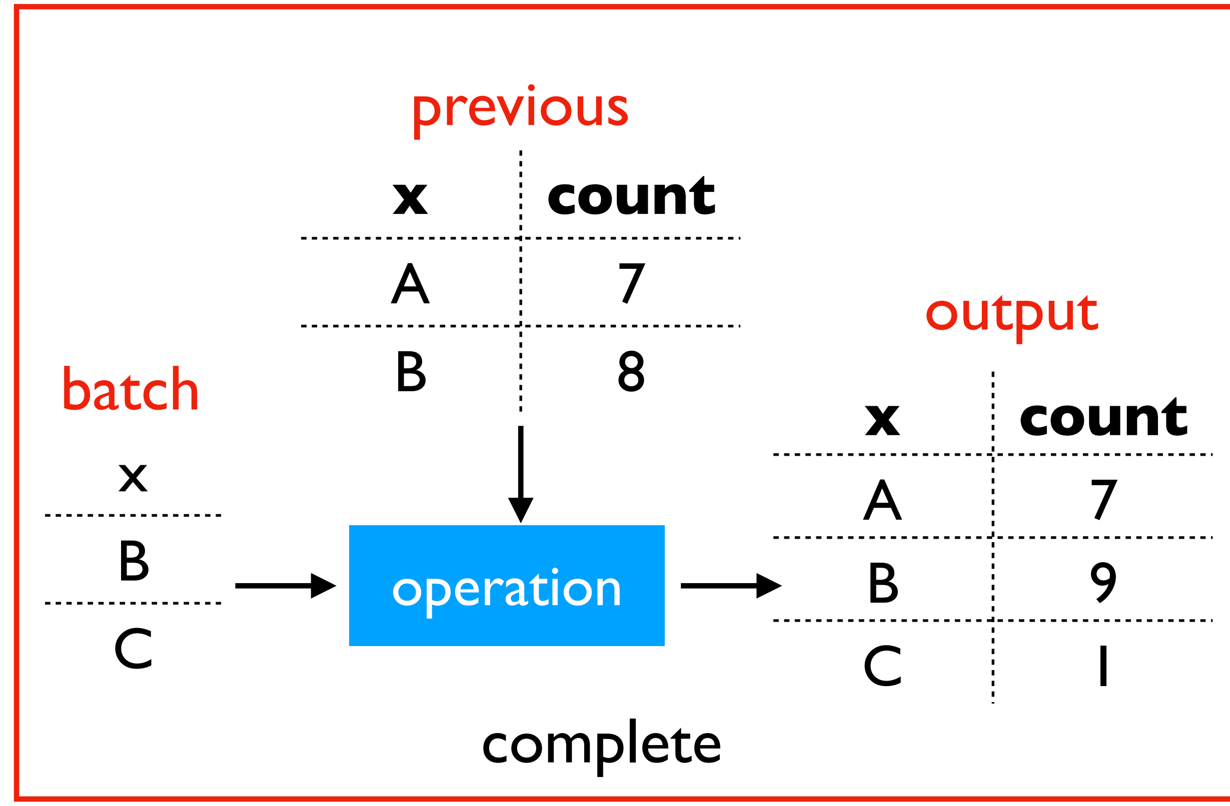


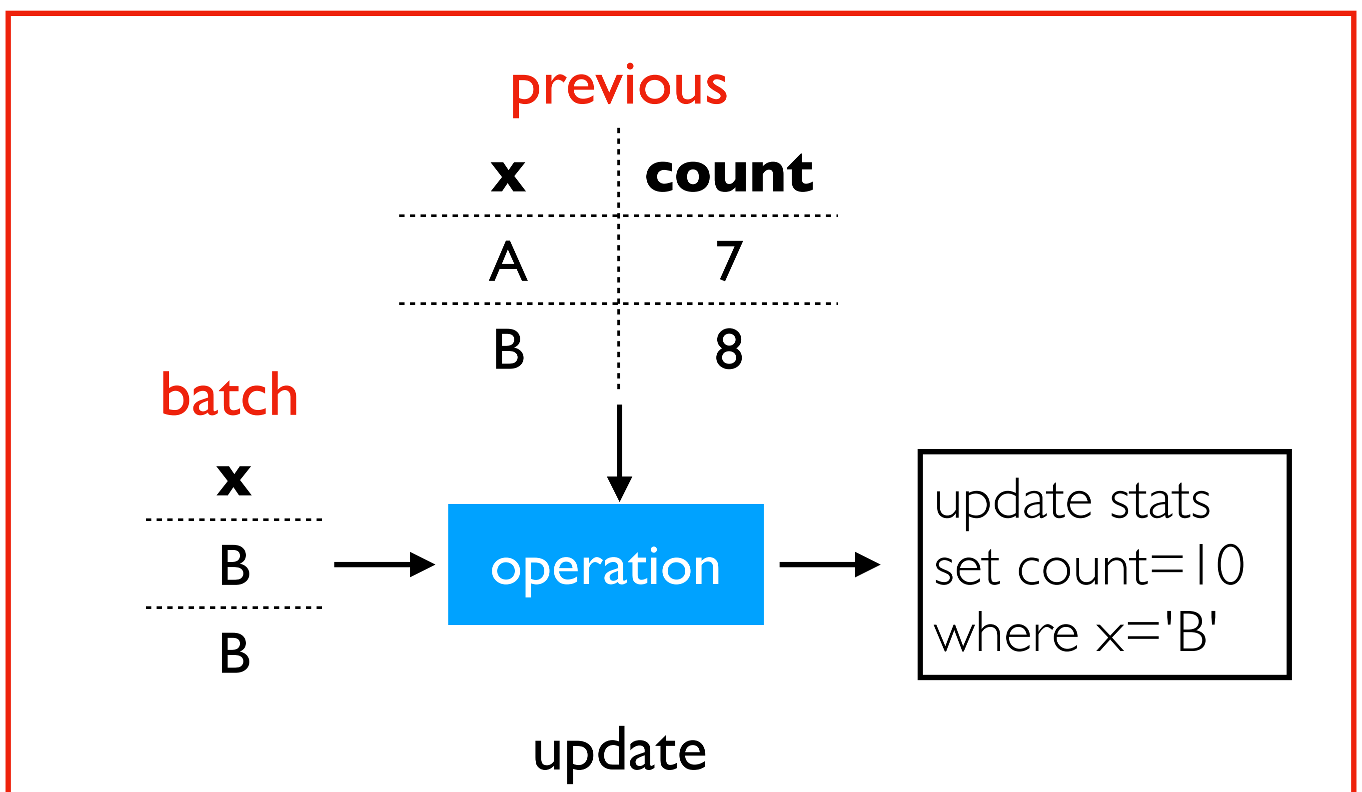




Output Modes: Update, Complete, Append





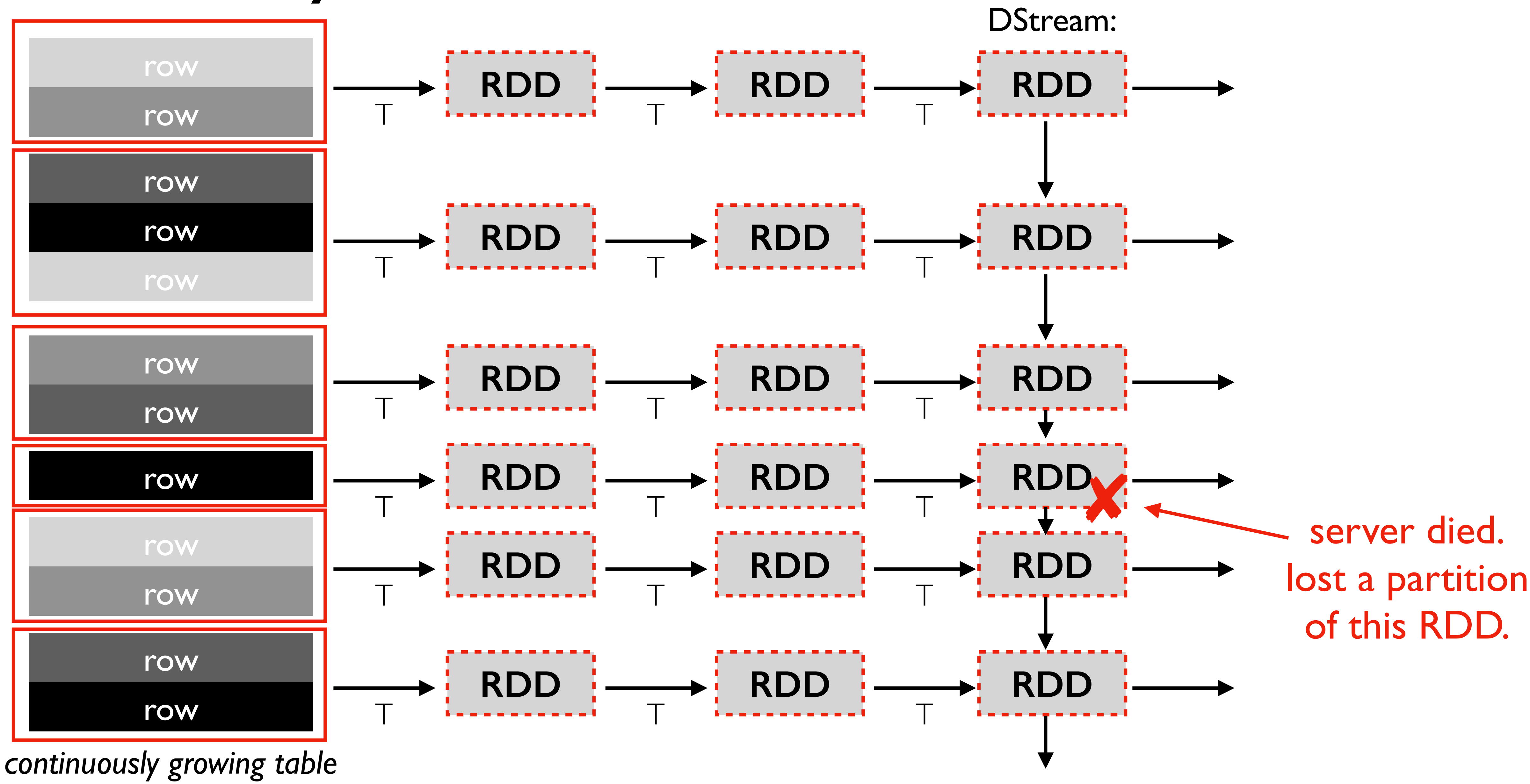


Different modes are available depending on transformation and output format.

Examples:

- **update**: output is usually a DB
- **append**: generally narrow transformations (previous output rows cannot change)
- **complete**: often for aggregates (otherwise too expensive so not allowed)

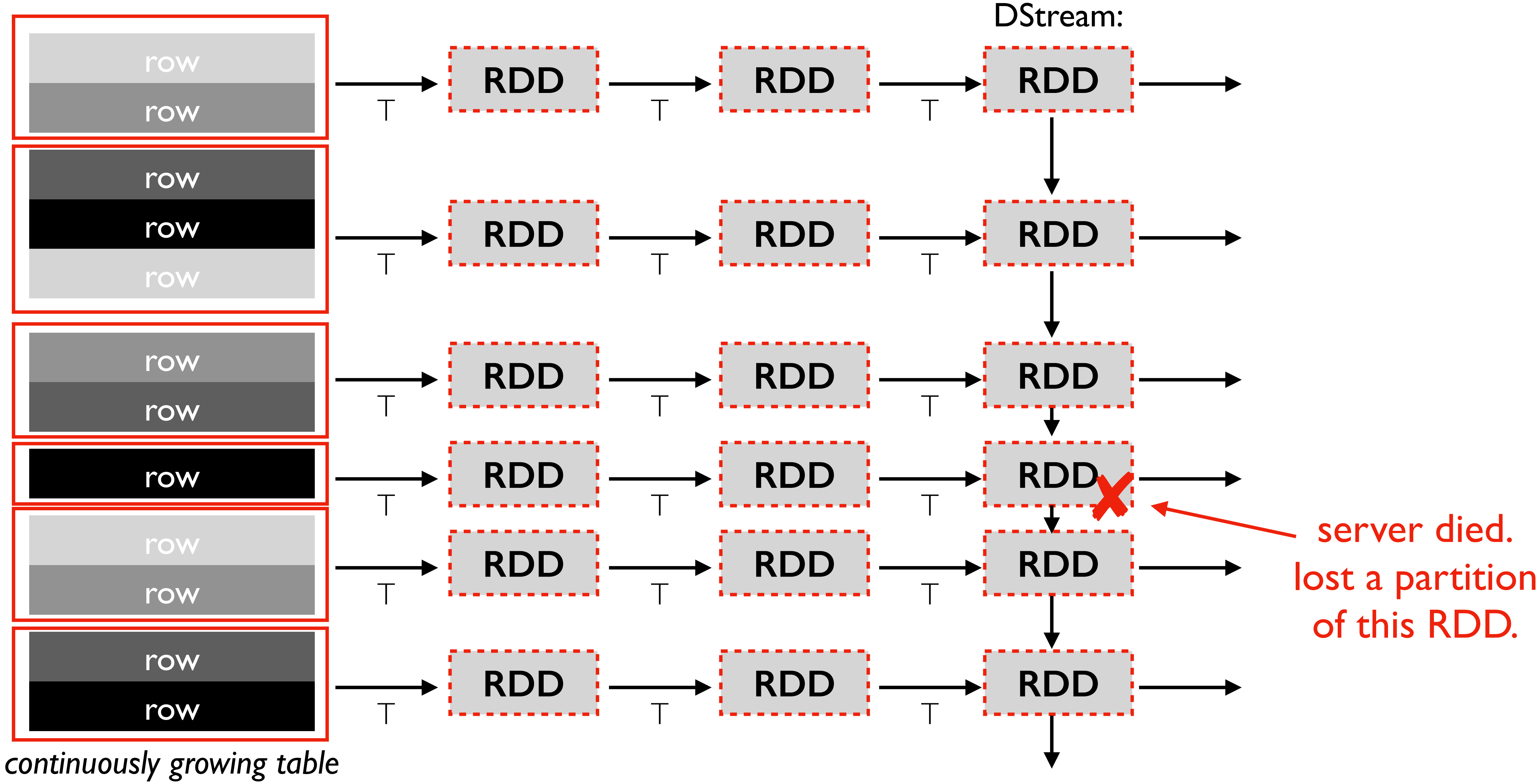
Recovery



Recovery:

- Spark usually doesn't replicate data because RDDs tell us how to recompute lost data
- What if source data is no longer available? (e.g., beyond Kafka retention time)
- What if it takes too long to recover?

Effecient Recovery



Recovery:

- Spark usually doesn't replicate data because RDDs tell us how to recompute lost data
- What if source data is no longer available? (e.g., beyond Kafka retention time)
- What if it takes too long to recover?

Spark Optimizations:

- Often, every worker can help with recovery work (i.e., recomputing data for an RDD)
- Checkpoint DStream once every 10 batches.

Outline: Spark Streaming

DStreams

Grouped Aggregates

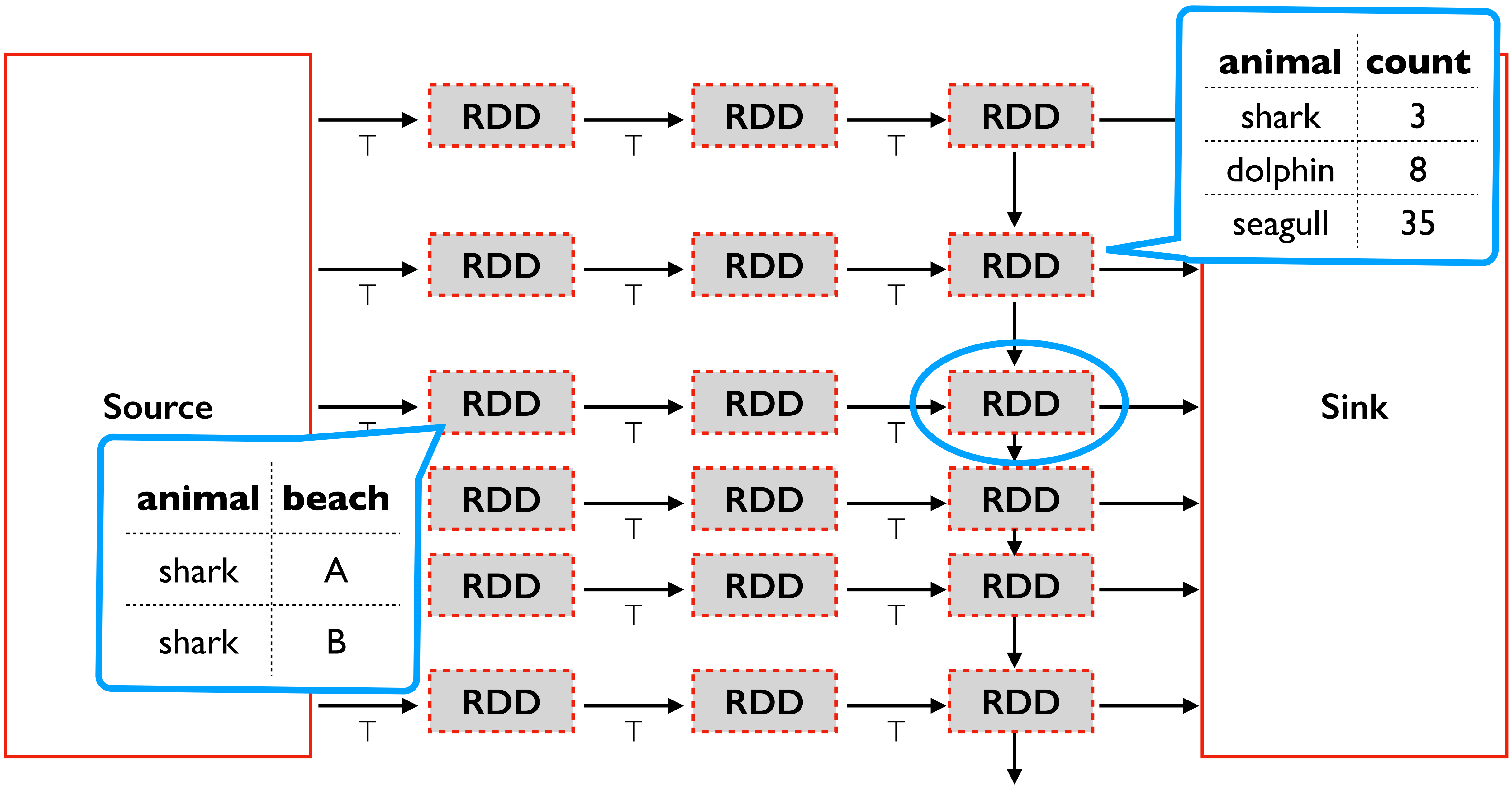
Watermarks

Pivoting

Joining

Exactly-Once Semantics

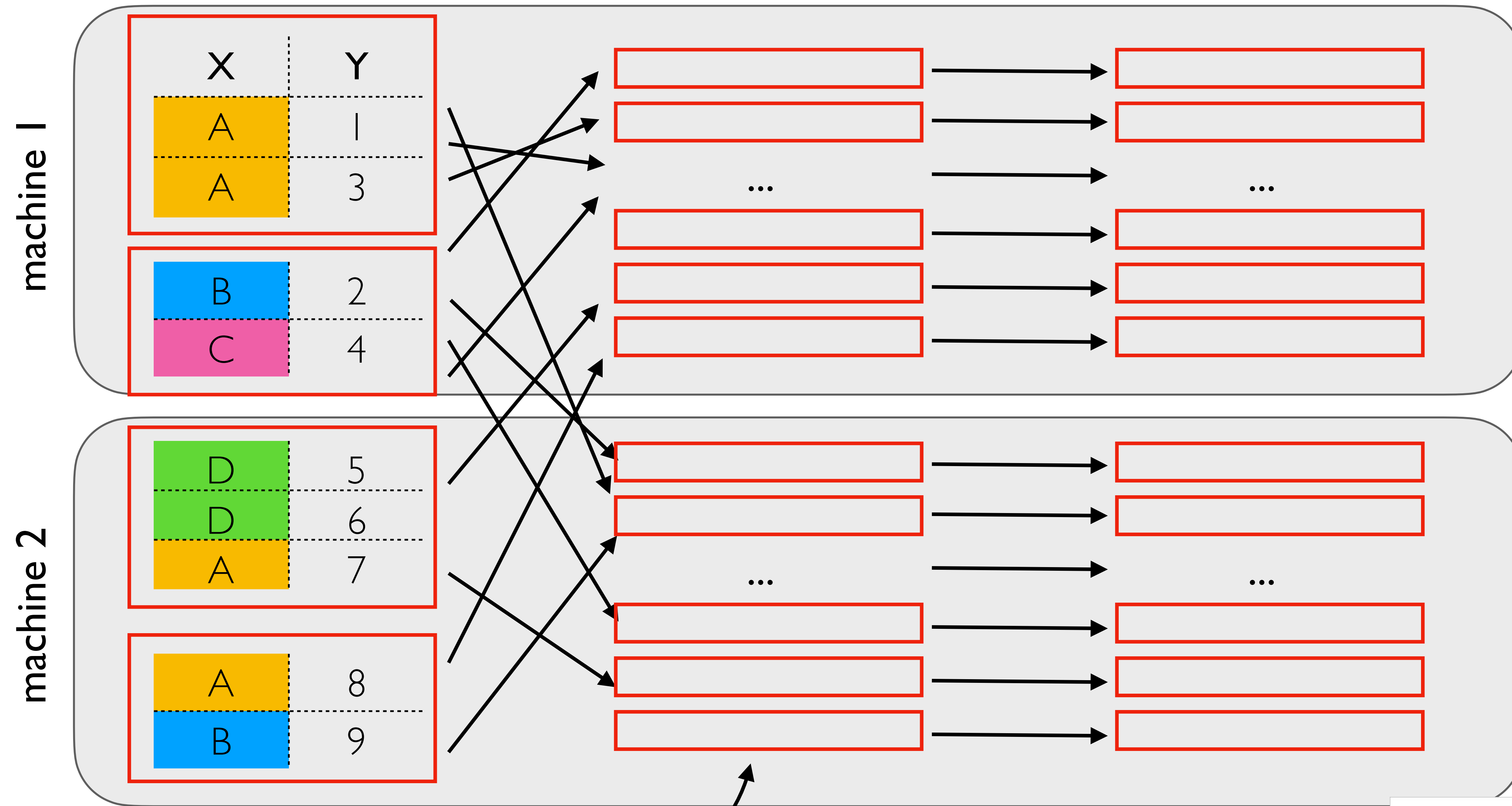
Incremental Aggregations




```
SELECT animal, COUNT(*)
FROM sightings
GROUP BY animal
```

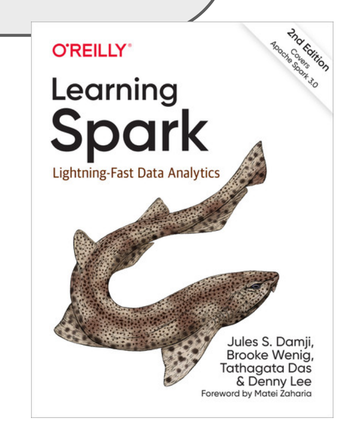
- many aggregations are easy to compute incrementally
- mode: update or complete (append usually not valid because previous rows change)
- space for state proportional to unique categories

Grouped Aggregate Internals: Shuffle Partitions



How many partitions will we have?

- spark.sql.shuffle.partitions (default 200) sets this -- fixed for whole application
- Often need to reduce for streaming jobs
- Batch jobs can automatically coallesce small partitions into bigger ones?
- Why not optimized for streaming? One challenge: coallescing based on data so far probably isn't good for future data. Avoid re-shuffling existing counts.



see Epilogue: Apache Spark 3.0

Outline: Spark Streaming

DStreams

Grouped Aggregates

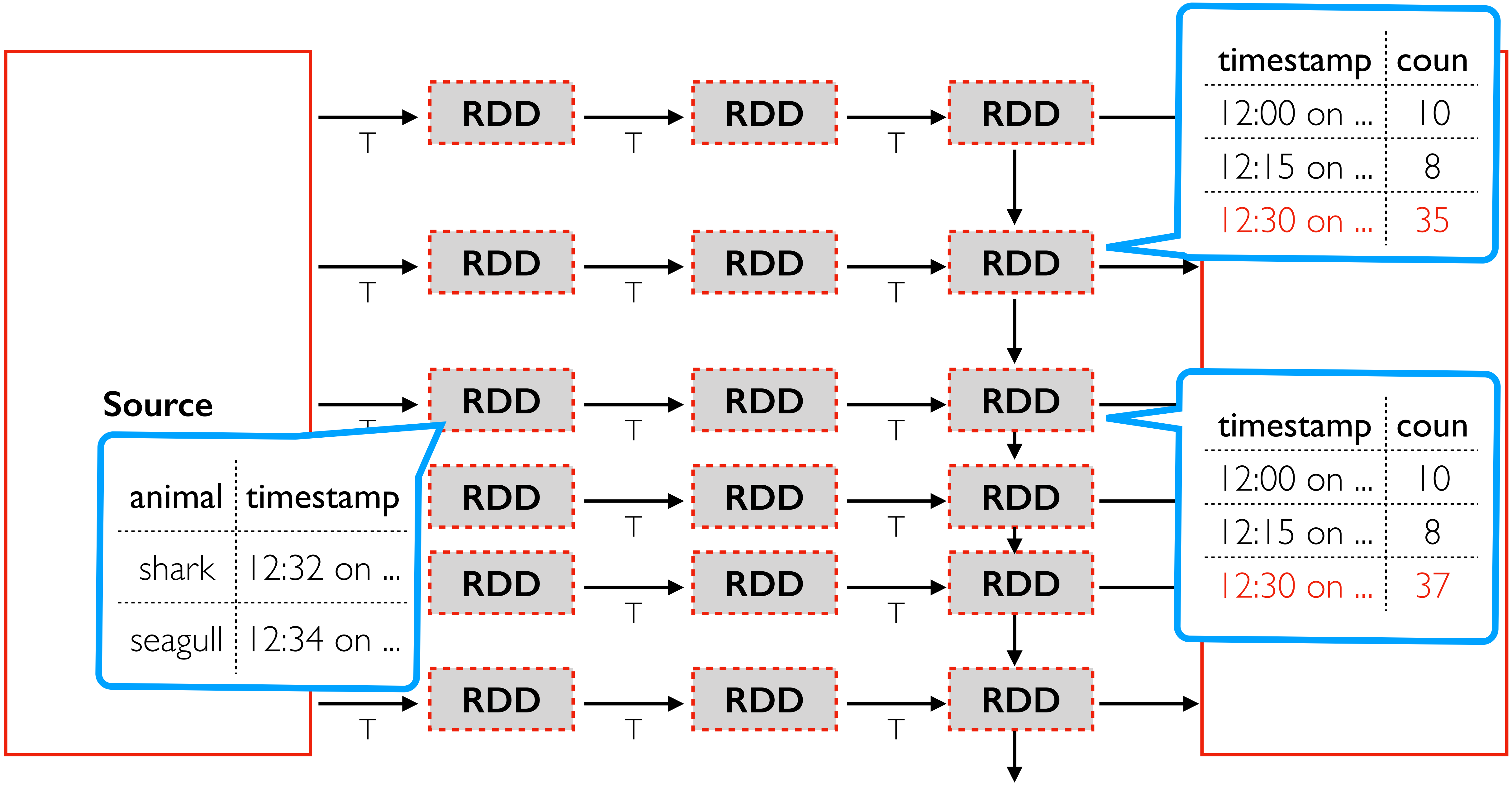
Watermarks

Pivoting

Joining

Exactly-Once Semantics

Grouping By Time Intervals



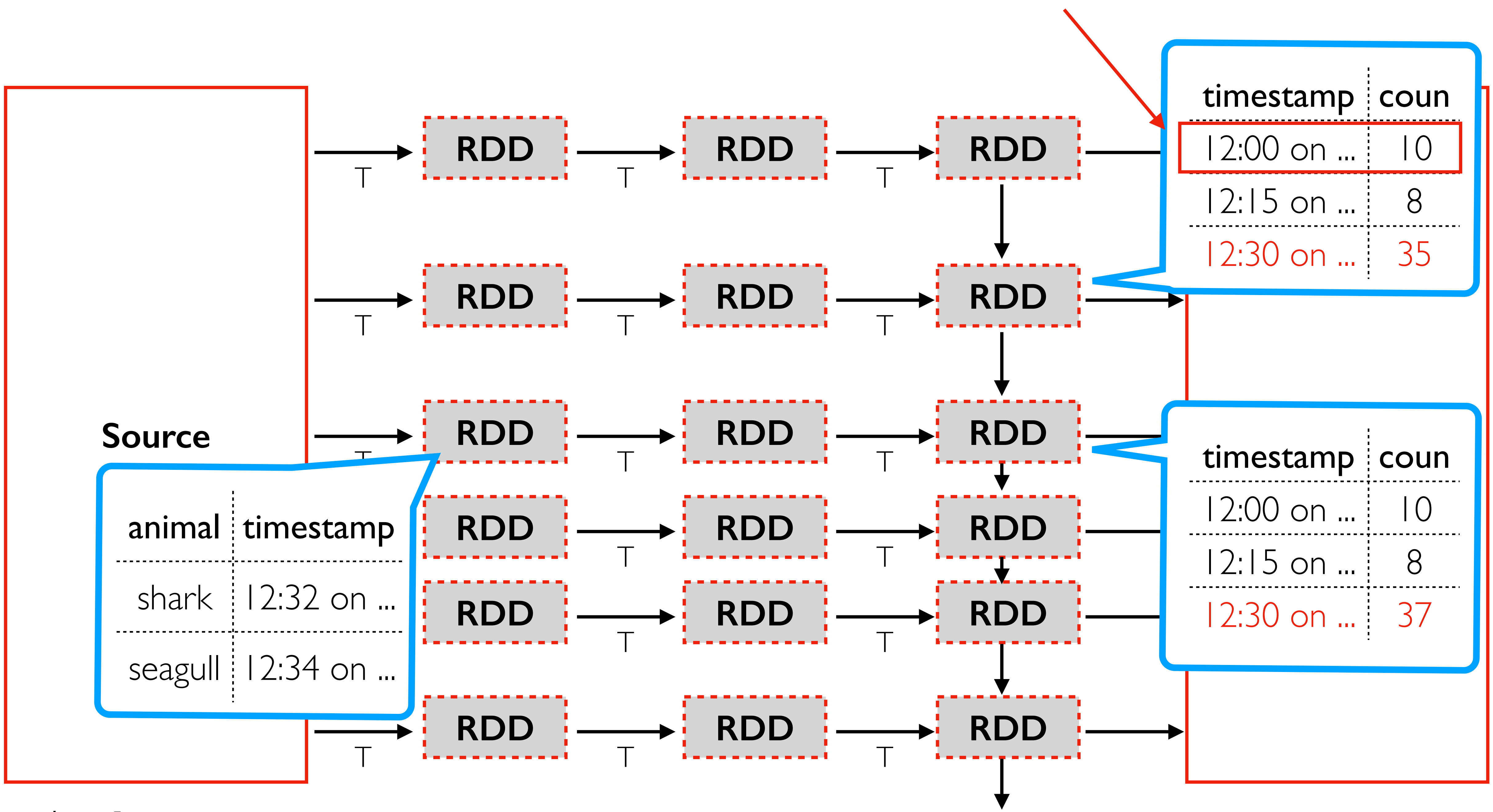

```
(animals
 .groupBy(window("timestamp",
                 "15 minute))
 .count())
```

Observations:

- number of groups (and RAM needed) grows indefinitely with time
- new batches contain recent times
- old times might occasionally pop up (Kafka delays)

Watermarks

Spark can discard this running count after 8:15pm because it is unlikely the pipeline will fall 8 hours behind




```
(animals
 .withWatermark("timestamp",
                "8 hours")
 .groupBy(window("timestamp",
                 "15 minute))
 .count())
```

Behavior:

- never throw away data newer than watermark time
- might throw away older data to save space

Outline: Spark Streaming

DStreams

Grouped Aggregates

Watermarks

Pivoting

Joining

Exactly-Once Semantics

Pivots

beach	animal
A	seagull
B	seagull
B	dolphin
C	seagull
A	seagull
A	dolphin
B	dolphin

pivot

beach	seagull	dolphin
A	2	1
B	1	2
C	1	0

what if we add a row with previously unseen values?

Pivots

beach	animal
A	seagull
B	seagull
B	dolphin
C	seagull
A	seagull
A	dolphin
B	dolphin
D	shark

pivot

beach	seagull	dolphin	shark
A	2	1	0
B	1	2	0
C	1	0	0
D	0	0	1

- **new row**: OK for batching and streaming
- **new col:** only OK for batching
- with streaming, it would cause consfusion if columns were adding mid query (how would somebody even query from our results?)
- some operations like pivot are supported for batching but not streaming

Outline: Spark Streaming

DStreams

Grouped Aggregates

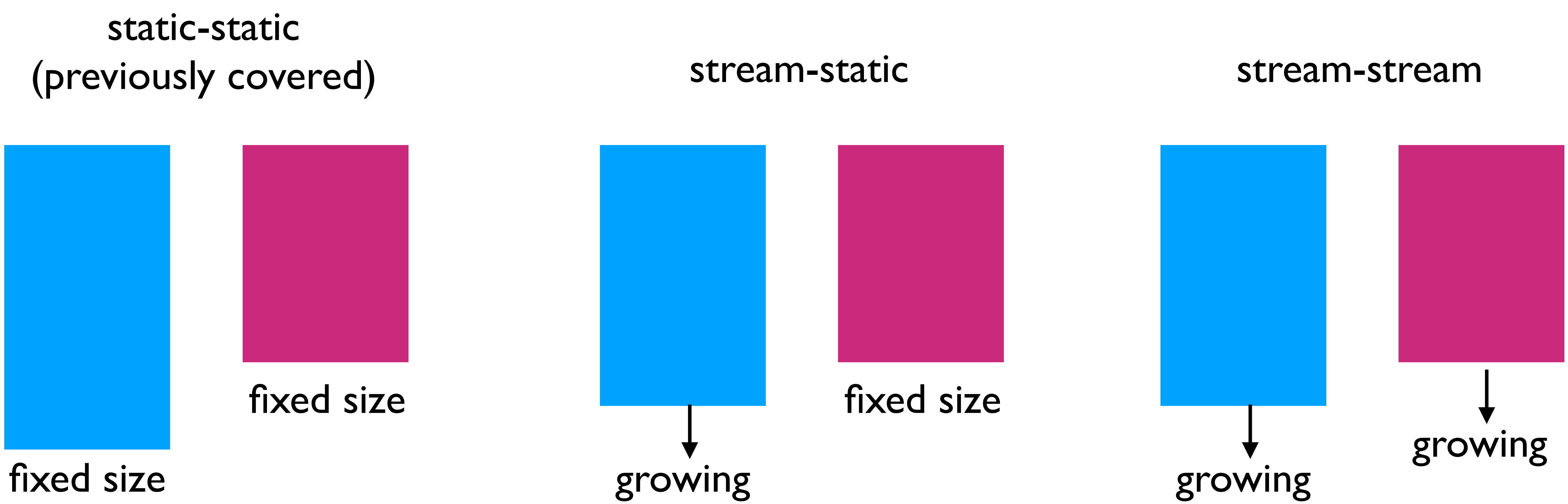
Watermarks

Pivoting

Joining

Exactly-Once Semantics

JOIN Scenarios

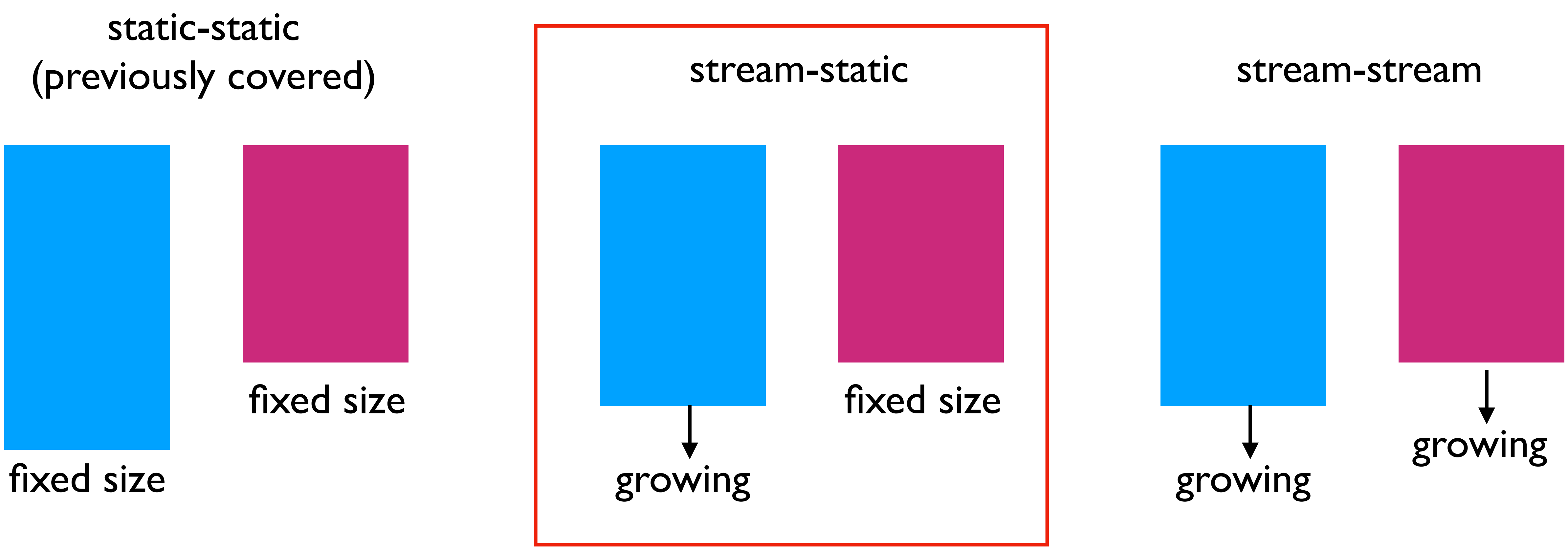


static-static review:

- shuffle sort merge join
- broadcast hash join

- Spark has at least some support for each scenario
- stream-stream can use an every increasing amout of memory if we're not carefuly (need watermarking)

JOIN Scenarios



static-static review:

- shuffle sort merge join
- broadcast hash join

- Spark has at least some support for each scenario
- stream-stream can use an every increasing amout of memory if we're not carefuly (need watermarking)

Stream-Static INNER JOIN

animals

id	name
1	dolphin
2	shark
3	seagull

fixed

sightings

beach	animal_id
A	3
B	3
A	2
C	4

growing

what *known* animals do we see?

```
SELECT beach, name
FROM sightings
INNER JOIN animals
ON sightings.animal_id=animals.id
```

results

beach	name
A	seagull
B	seagull
A	shark

growing

Stream-Static LEFT JOIN

animals

id	name
1	dolphin
2	shark
3	seagull

fixed

sightings

beach	animal_id
A	3
B	3
A	2
C	4

growing

are there any sightings of unknown animals?

```
SELECT beach, animal_id
FROM sightings
LEFT JOIN animals
ON sightings.animal_id=animals.id
WHERE name IS NULL
```

results

beach	name
C	4

growing

Stream-Static RIGHT JOIN

animals

id	name
1	dolphin
2	shark
3	seagull

fixed

sightings

beach	animal_id
A	3
B	3
A	2
C	4

growing

are there any animals that are never seen?

```
SELECT name, beach
FROM sightings
RIGHT JOIN animals
ON sightings.animal_id=animals.id
WHERE beach IS NULL
```

results

name	beach
dolphin	NULL

fixed

why is it impossible to compute the results, even though it would be easy for static-static?

Cannot RIGHT JOIN if right is static; Cannot LEFT JOIN if left is static

animals

id	name
1	dolphin
2	shark
3	seagull

fixed

sightings

beach	animal_id
A	3
B	3
A	2
C	4

growing

are there any animals that are never seen?

```
SELECT name, beach
FROM sightings
RIGHT JOIN animals
ON sightings.animal_id=animals.id
WHERE beach IS NULL
```

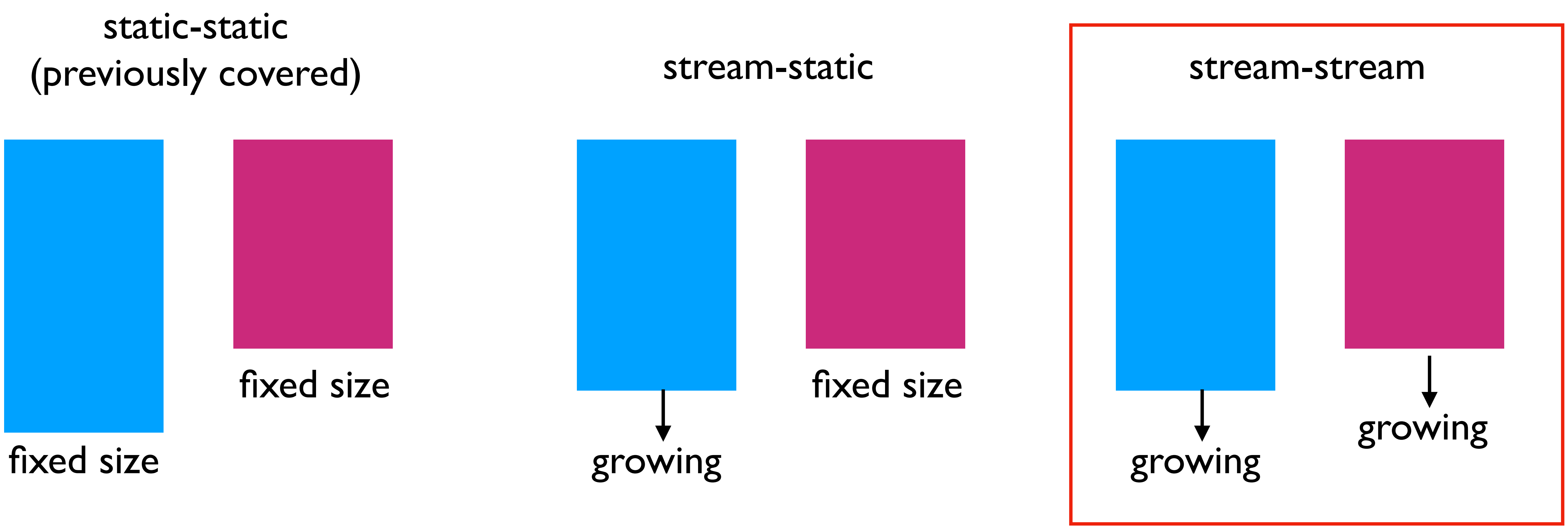
results

name	beach
dolphin	NULL

fixed

we can never say an animal is never seen if we keep seeing animals forever, so this query is illogical (and unsupported by Spark)

JOIN Scenarios



static-static review:

- shuffle sort merge join
- broadcast hash join

- Spark has at least some support for each scenario
- stream-stream can use an every increasing amout of memory if we're not carefuly (need watermarking)

Stream-Stream

closures

date	type
4/10/23	"all day"
4/15/23	"part day"
4/20/23	"all day"

growing

↓

sightings

date	animal
4/13/23	seagull
4/14/23	seagull
4/14/23	shark
4/15/23	dolphin

growing

↓

how many sharks are seen on days when the beach is closed?

```
SELECT COUNT(*)
FROM sightings
INNER JOIN closures
ON sightings.date=closures.date
WHERE animal = 'shark'
```

challenge: we can't "forget" about this row if we might later learn about a beach closure on the 14th (for example, from a lagging Kafka stream)

solution: use watermarks (like for grouped aggregates)

note: Spark works without watermarks; it just keeps using more memory indefinitely

Outline: Spark Streaming

DStreams

Grouped Aggregates

Watermarks

Pivoting

Joining

Exactly-Once Semantics

Exactly-Once Semantics

If a task crashes, we can restart a new one, but we don't want to:

- double count any row
- miss any row

Spark can achieve exactly-once semantics given 3 features

- your code is "deterministic" (does same thing each time given same inputs)
- **source:** it's possible to go back and re-read older inputs that the previous task was processing when it crashed (Kafka makes this easy, within the retention period)
- **sink:** it is "idempotent" (can supress duplicates)

file sink (parquet files on HDFS) supports this -- Spark writes checkpoint files that identify which output files correspond to which input messages

Parquet on HDFS

HDFS directory that will accumulate parquet files

HDFS directory where Spark stores info about how to supress duplicates when reading those parquet files

```
query = (df
  .writeStream
  .format("parquet")
  .option("path", ????)
  .option("checkpointLocation", ????)
  .start())
```

When Spark reads a directory of parquet files, it automatically supresses duplicates. But be careful reading individual parquet files in a directory yourself, because then you might see those duplicates.

Conclusion

Spark streaming is frequent batch computing

- DStream is series of RDDs
- Most things we can do with regular DataFrames can be done with streams
- Not quite realtime, but fast crash recovery

Performance

- choose shuffle partition count carefully
- apply watermarks to limit memory consumption