[544] Spark Streaming

Tyler Caraza-Harter

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)
]
```

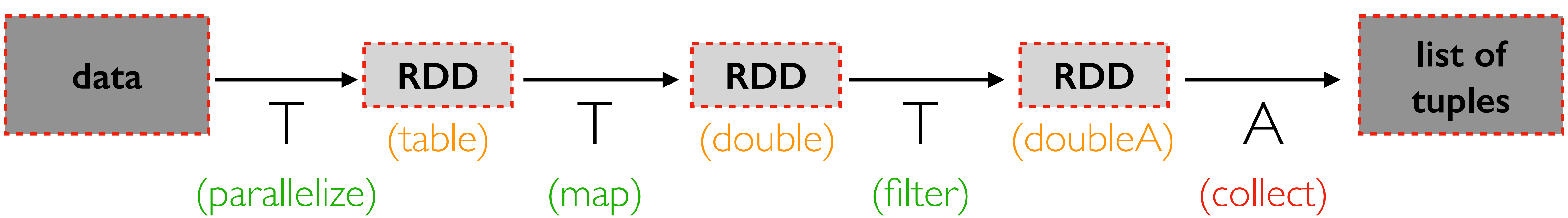
```
def mult2(row):
    return (row[0], row[1] * 2)

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),
```

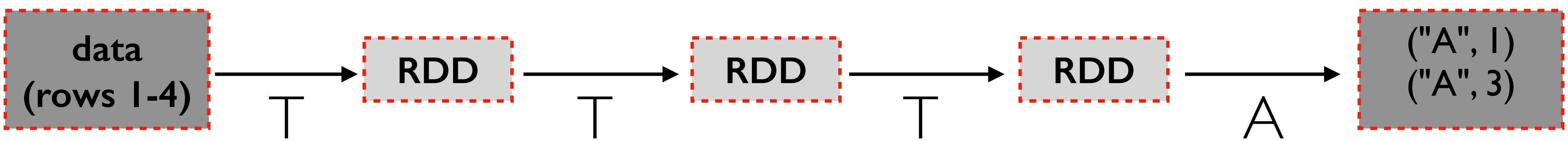
new data

```
        ("A", 5),
        ("C", 6)
]
```

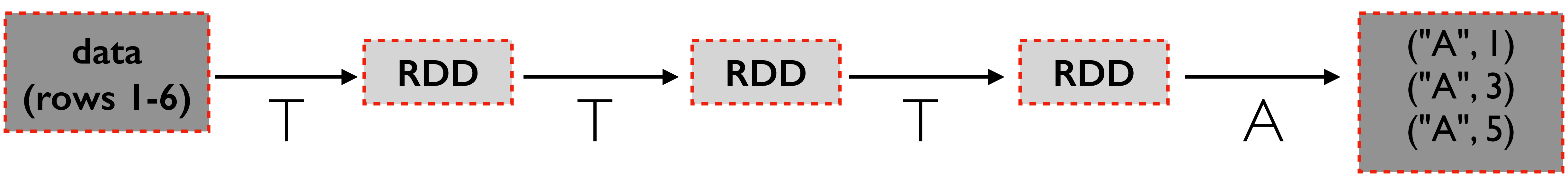
```
def mult2(row):
    return (row[0], row[1] * 2)

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

Round 1



Round 2



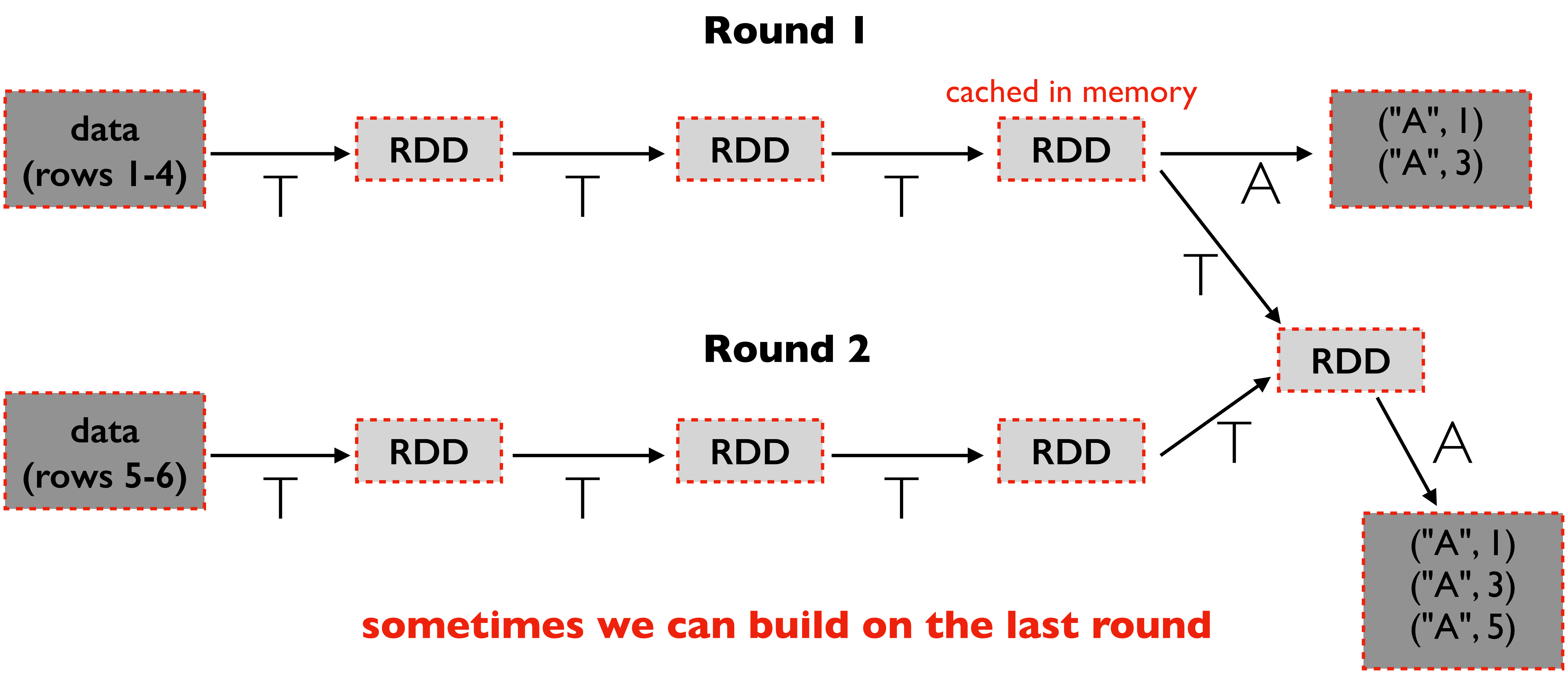
re-doing work is wasteful!

Handling Data Changes: Incremental Computation

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

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

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

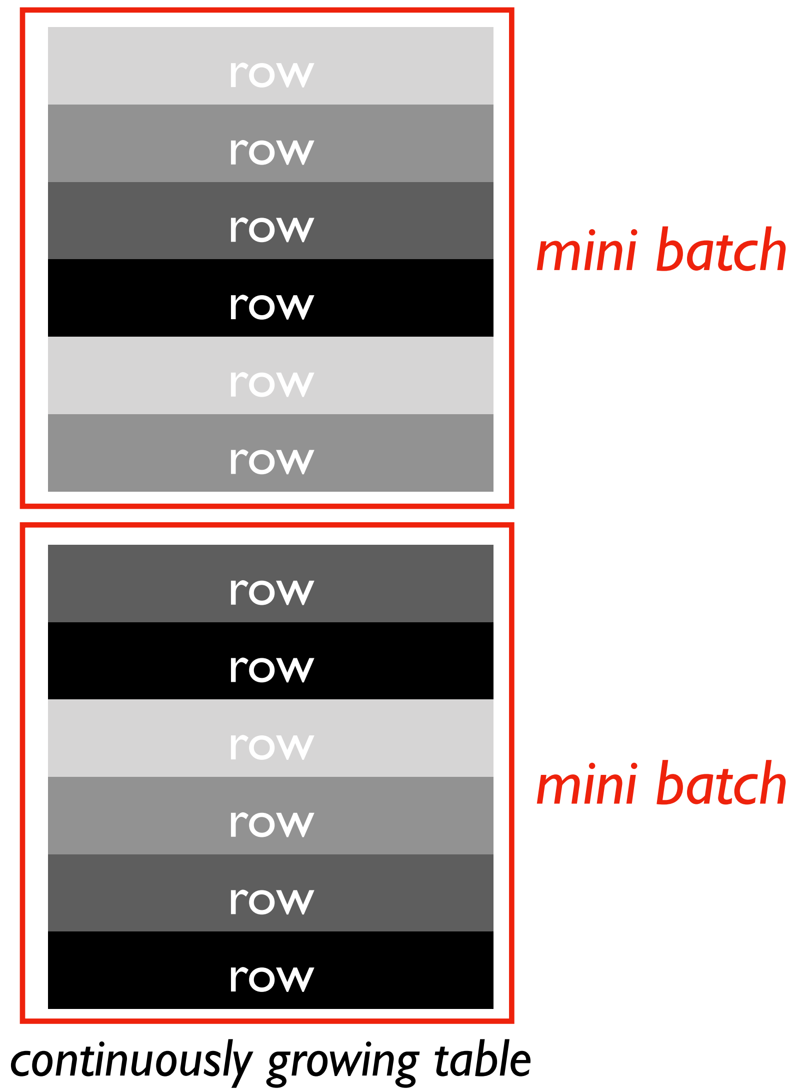


sometimes we can build on the last round

Some DataFrames constantly grow

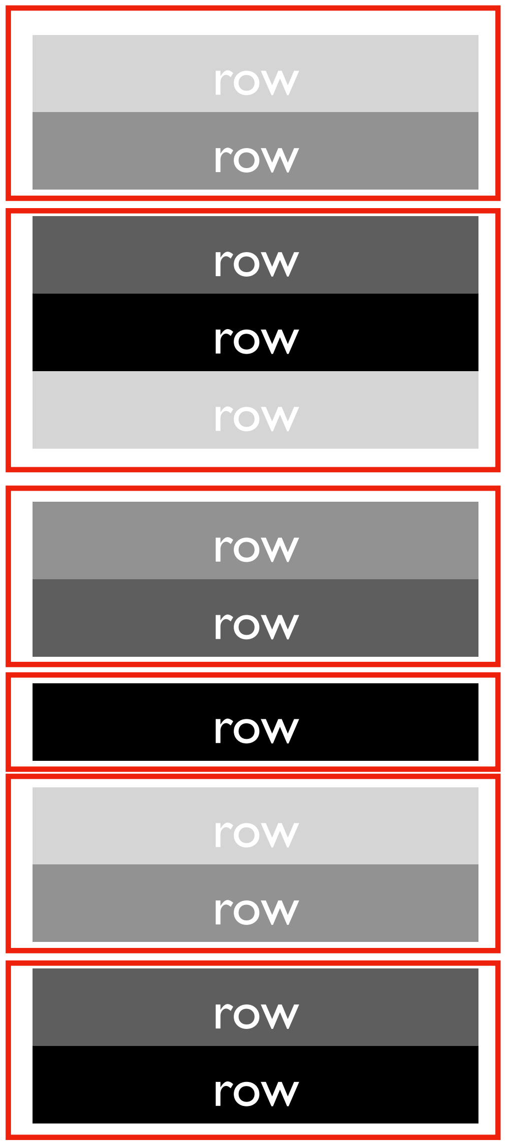
continuously growing table

Mini Batches




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

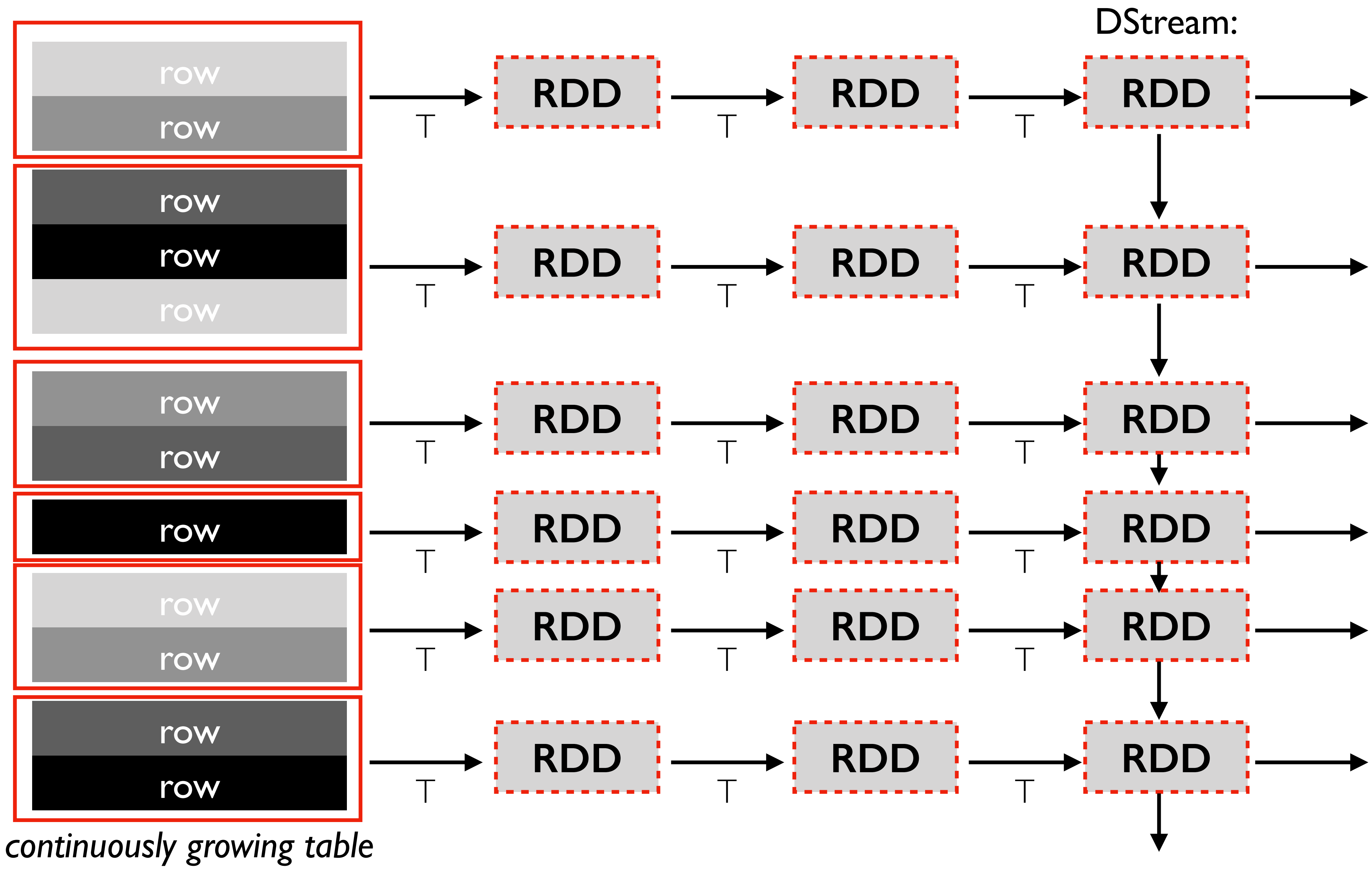
Trigger Frequency



continuously growing table

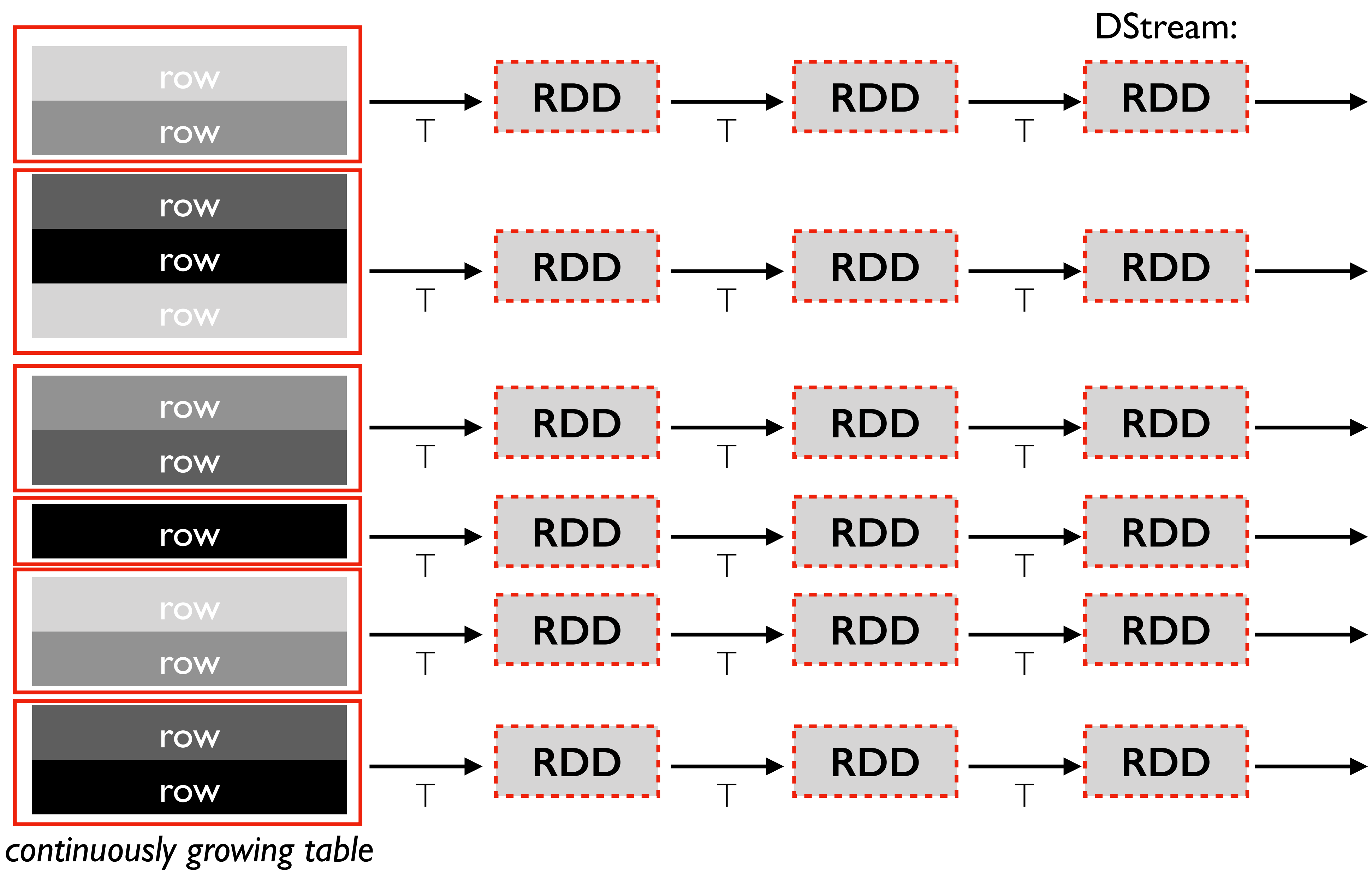
```
trigger(processingTime="4 seconds")
```

DStream (Stateful)



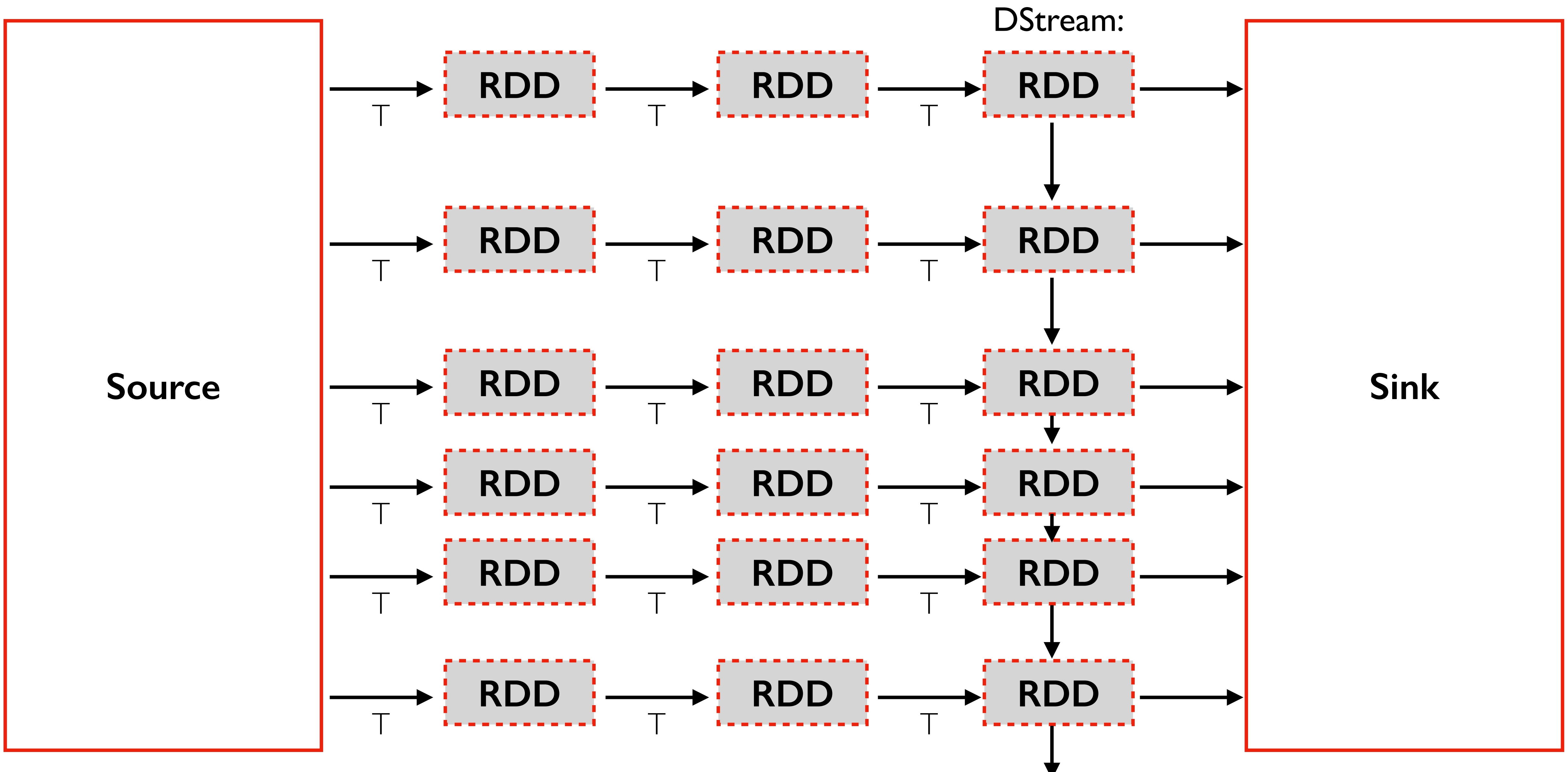
A Spark DStream is a series of RDDs

DStream (Stateless)



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

Source => DStream => Sink



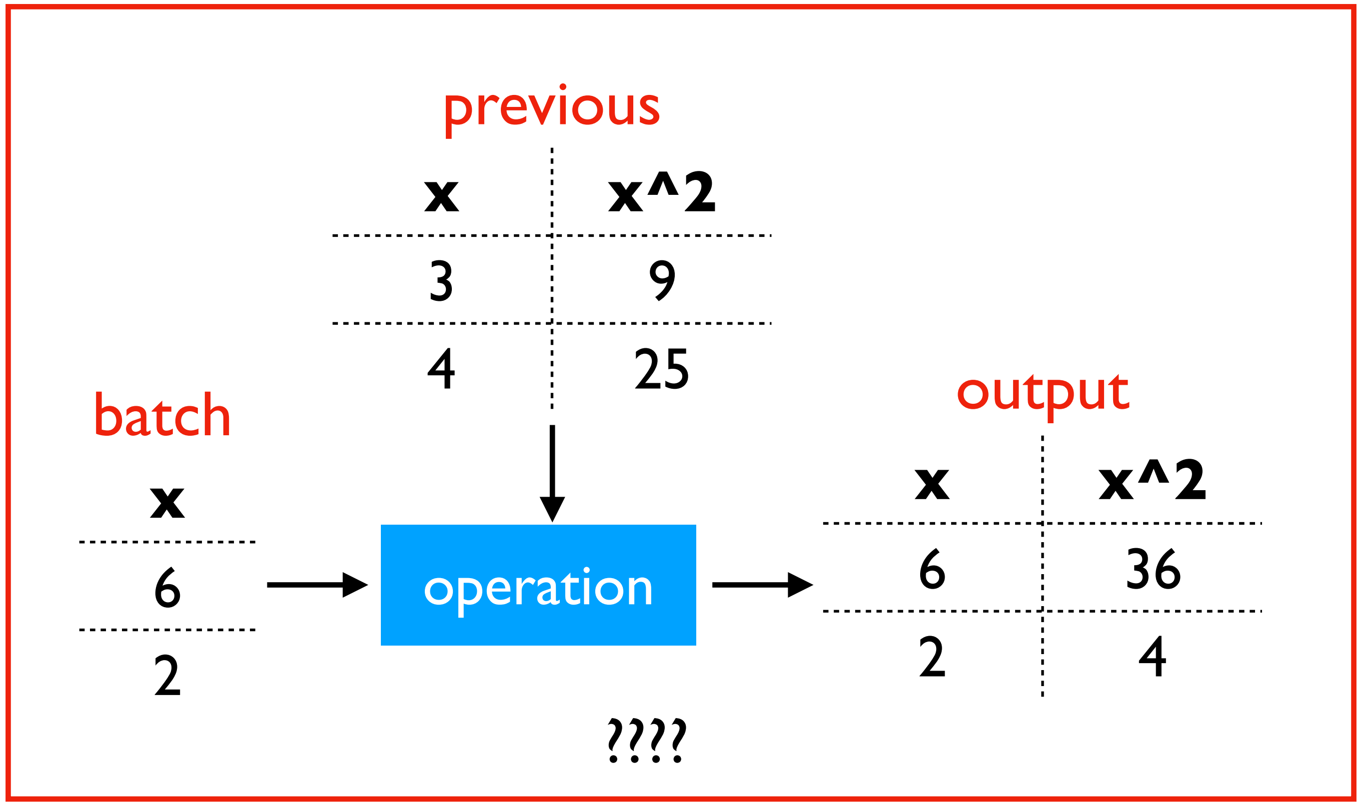
- Kafka
- HDFS files
- Cassandra
- etc.

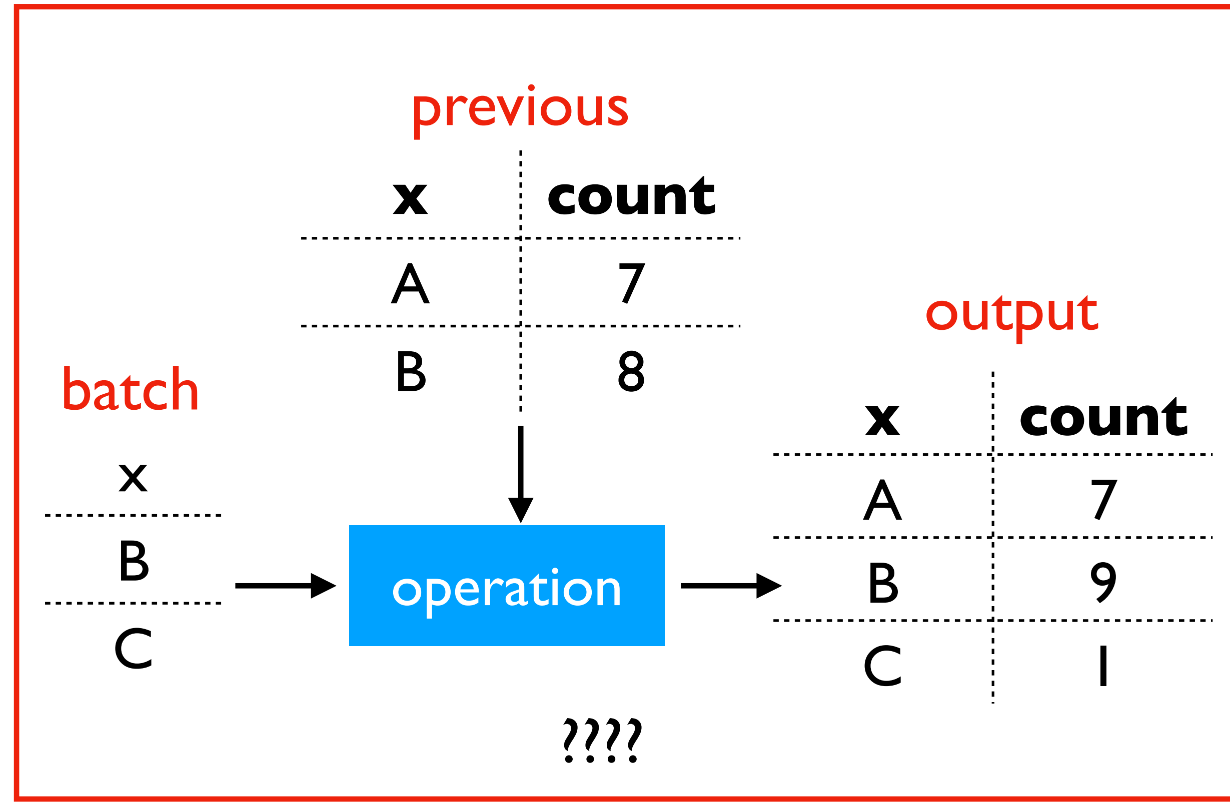
A DStream continuously pulls data from a source, transforms it, and sends it to a sink

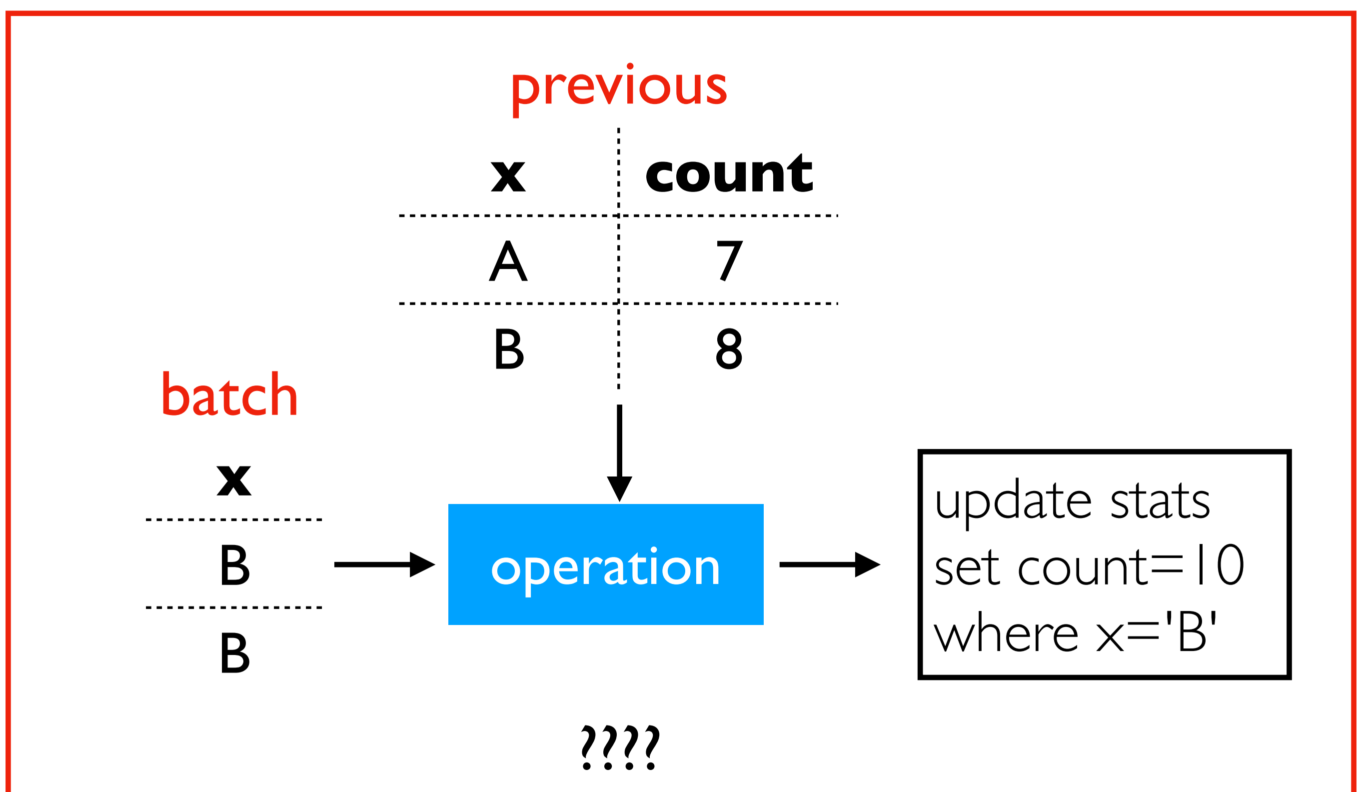
- Kafka
- HDFS files
- console
- etc.

many possible source/sink formats

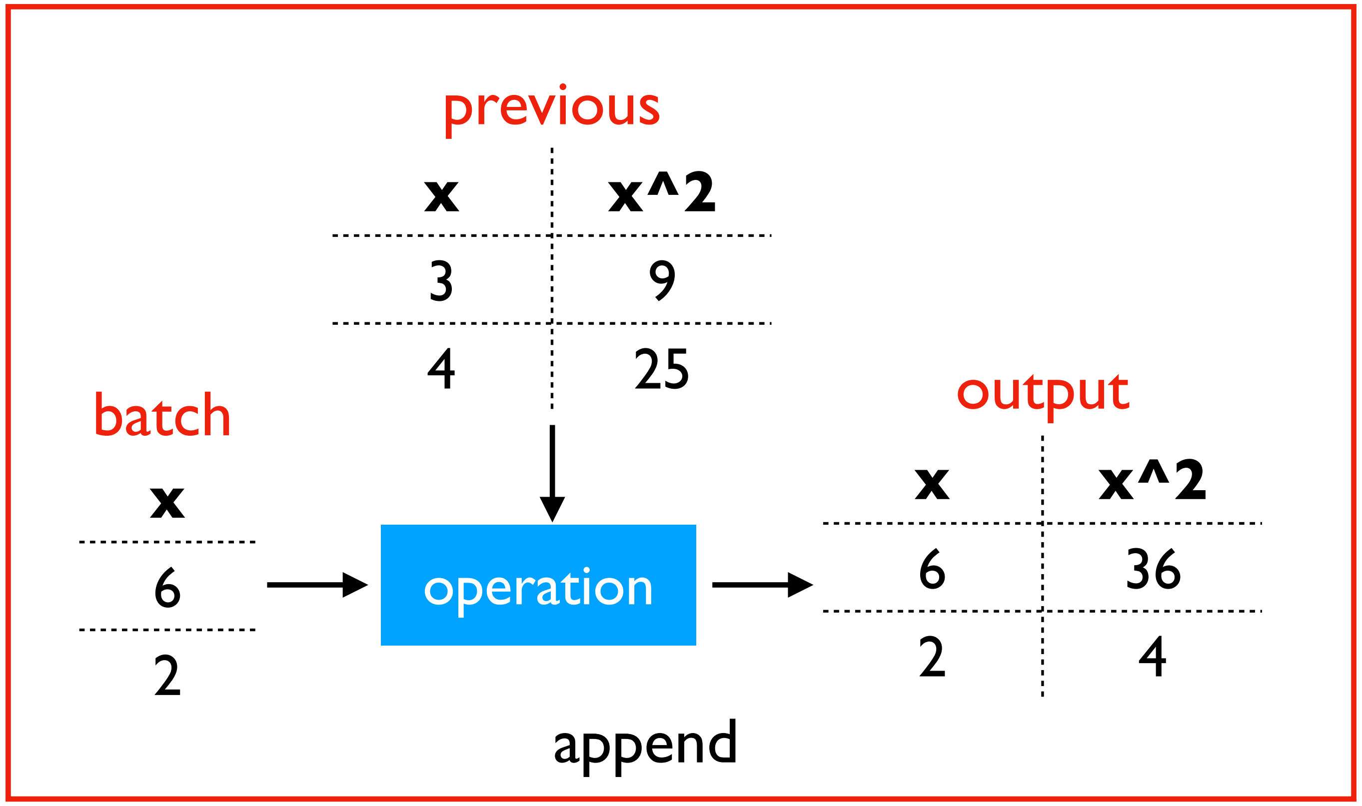
Output Modes: Update, Complete, Append

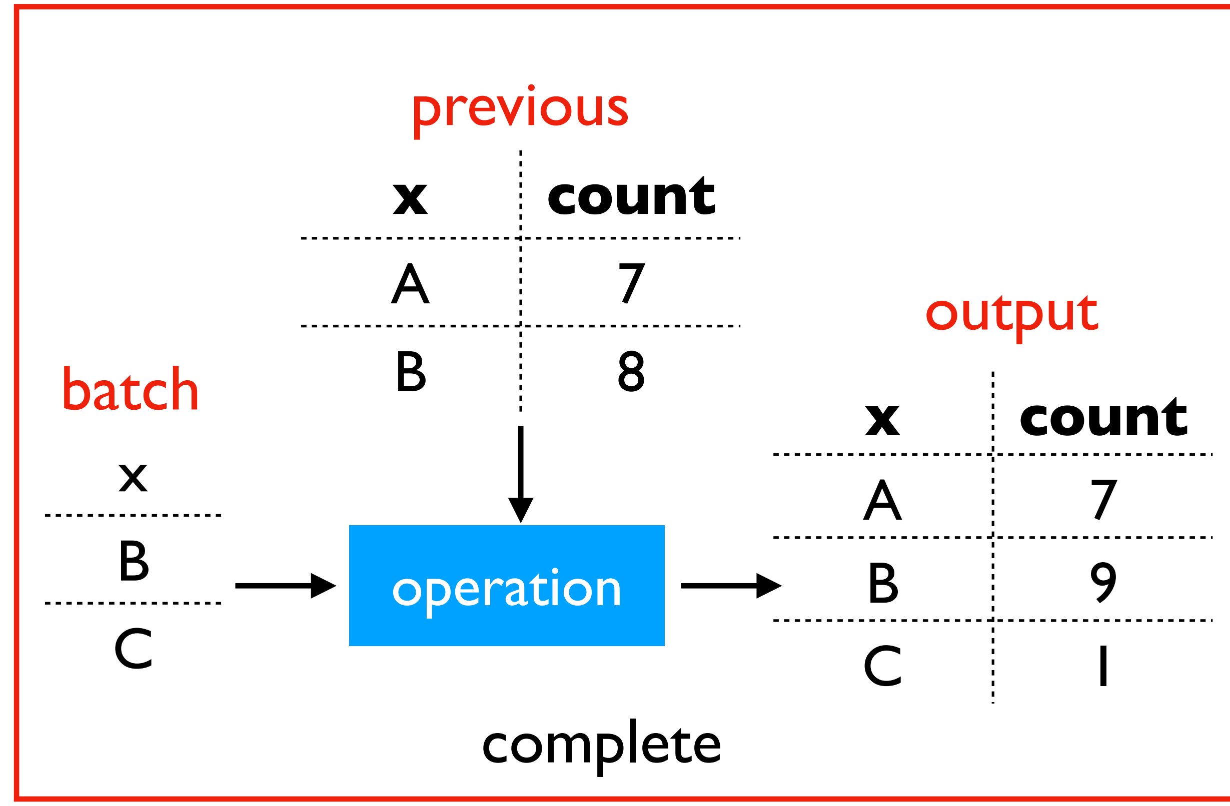


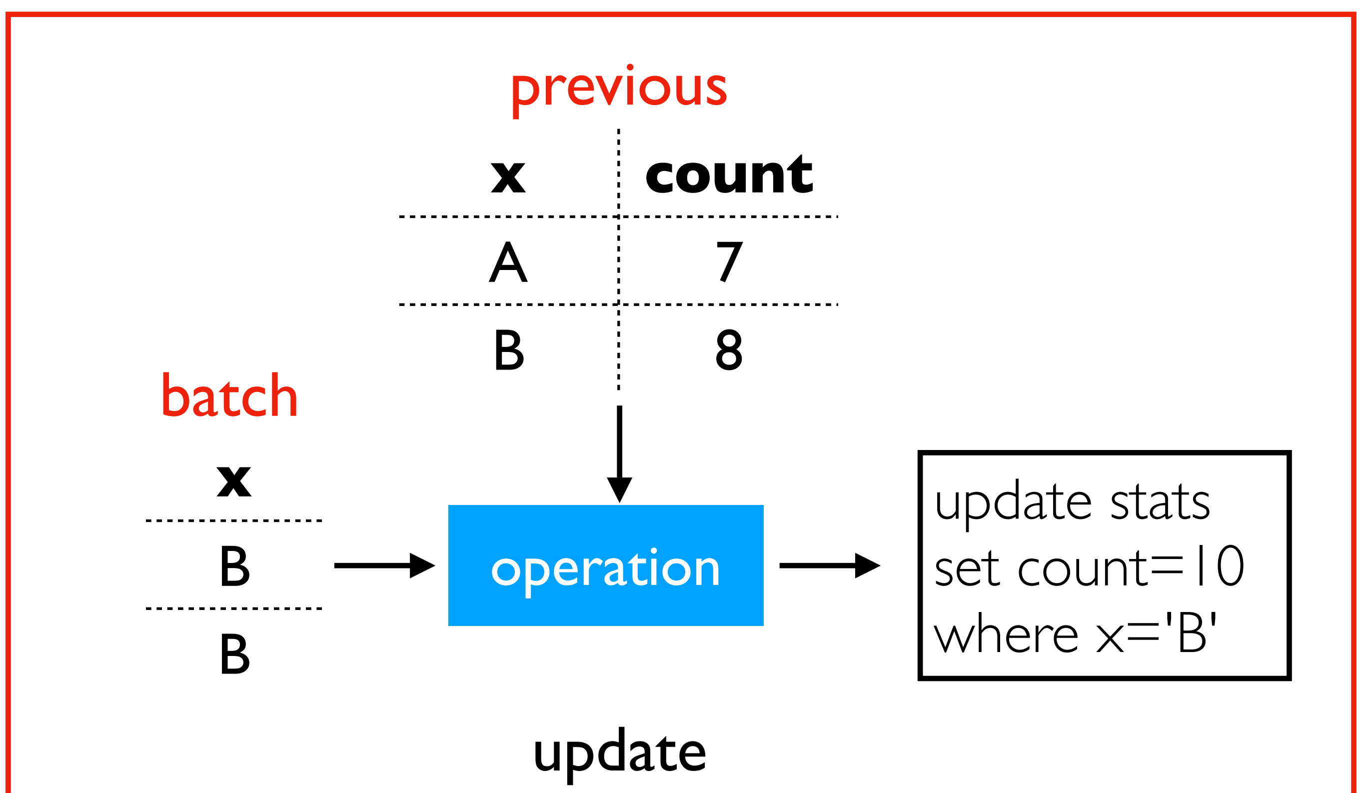




Output Modes: Update, Complete, Append





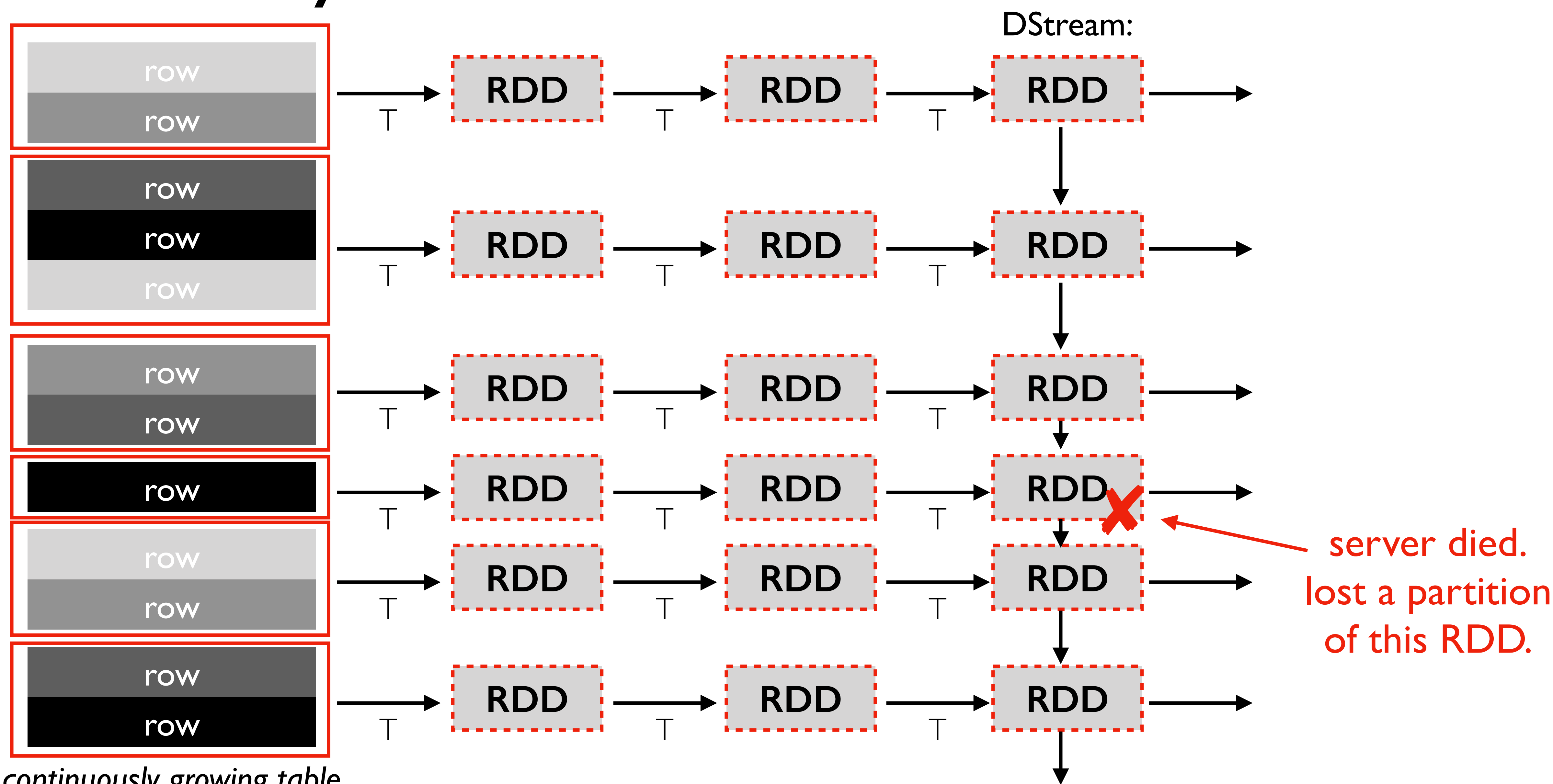


Different modes are available depending on transformation and output format.

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

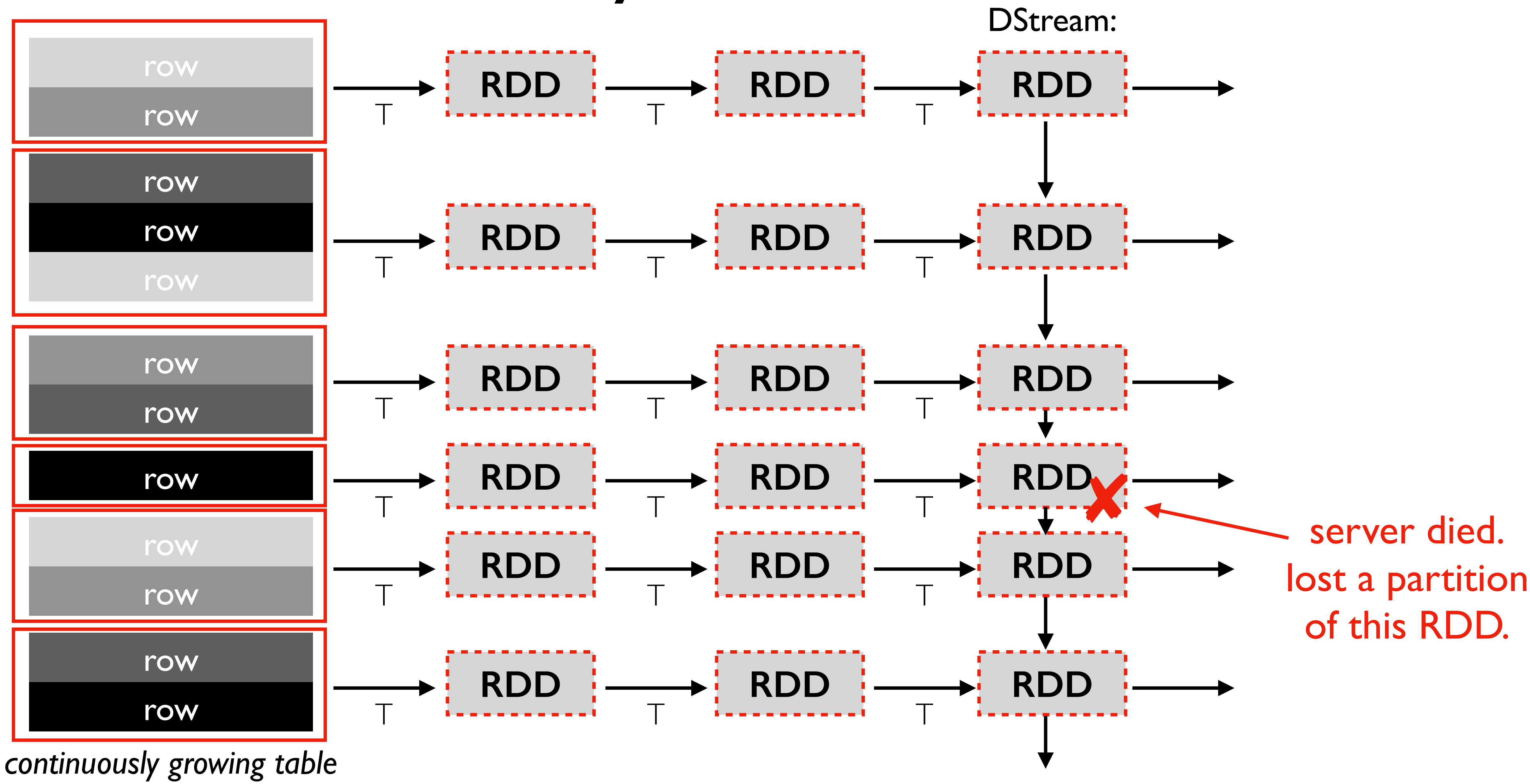


continuously growing table

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

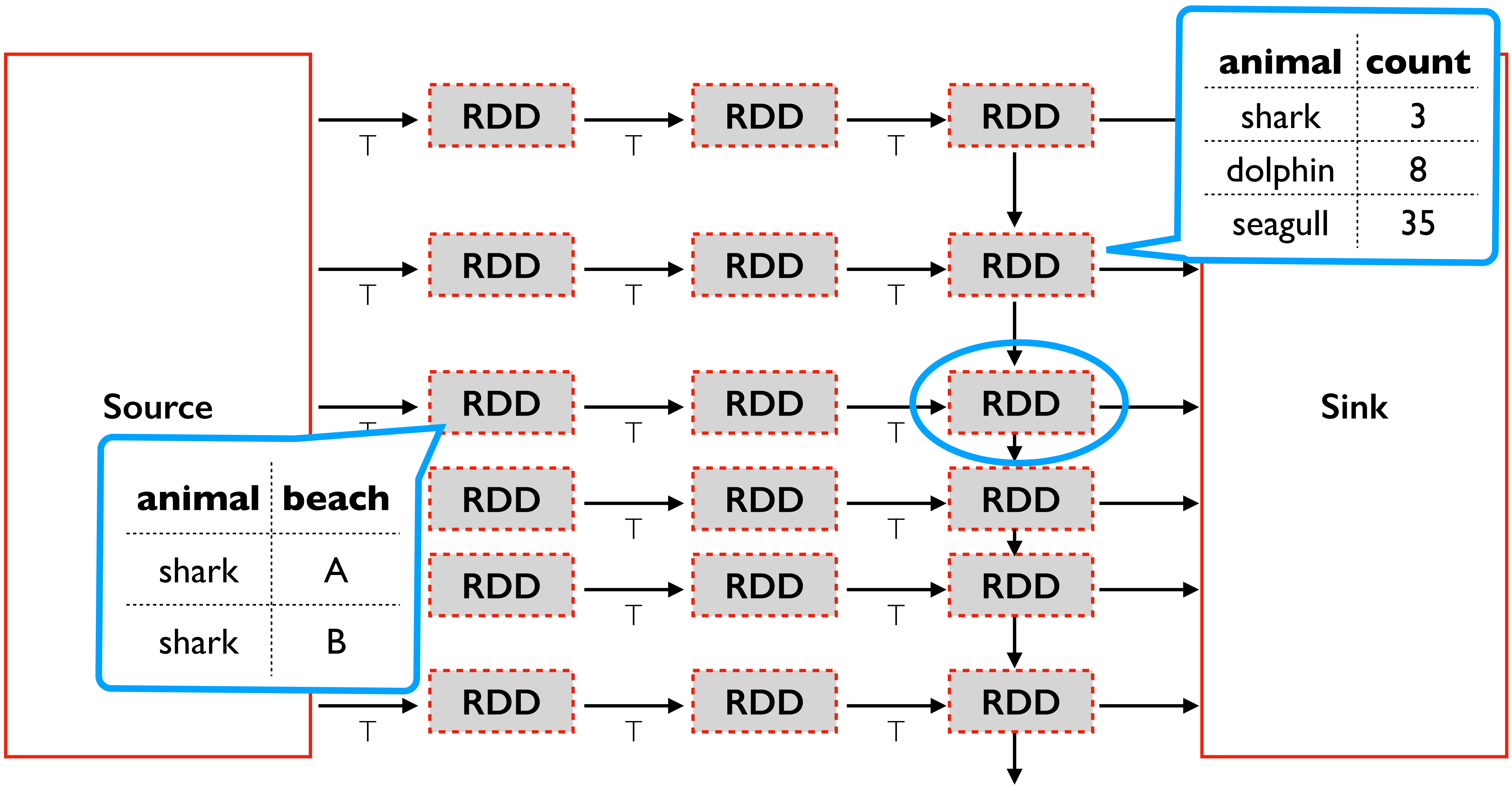
Watermarks

Pivoting

Joining

Exactly-Once Semantics

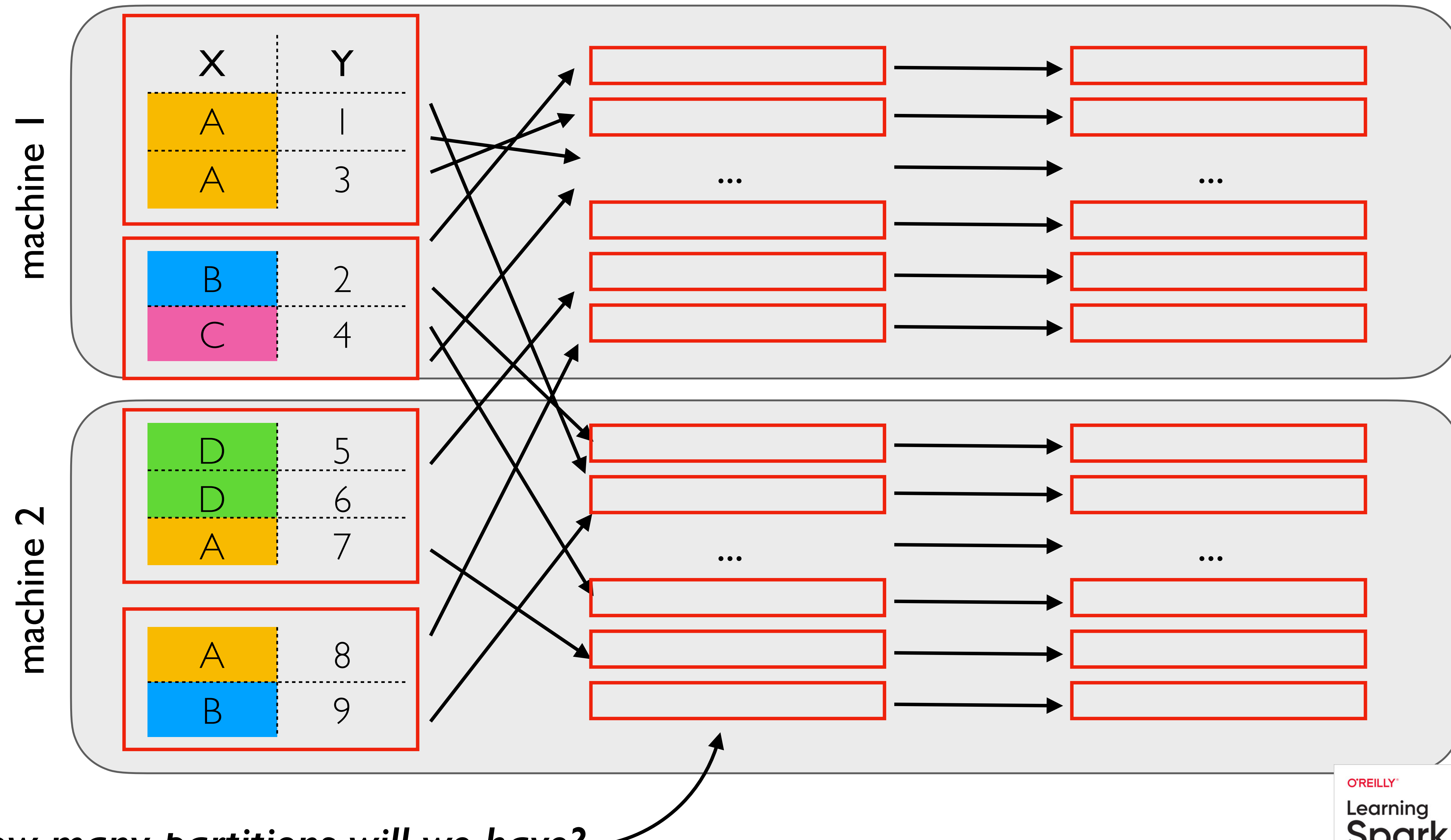
Incremental Aggregations




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

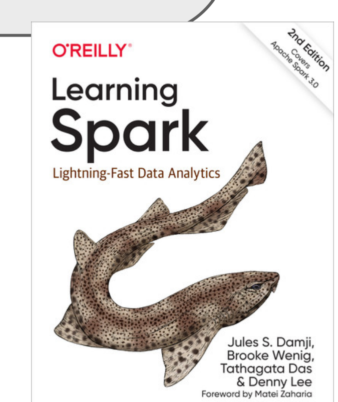
- many aggregations are easy to compute incrementally
- mode: update or complete (append usually not valid because previous rows change)
- space for state proportional to unique categories

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

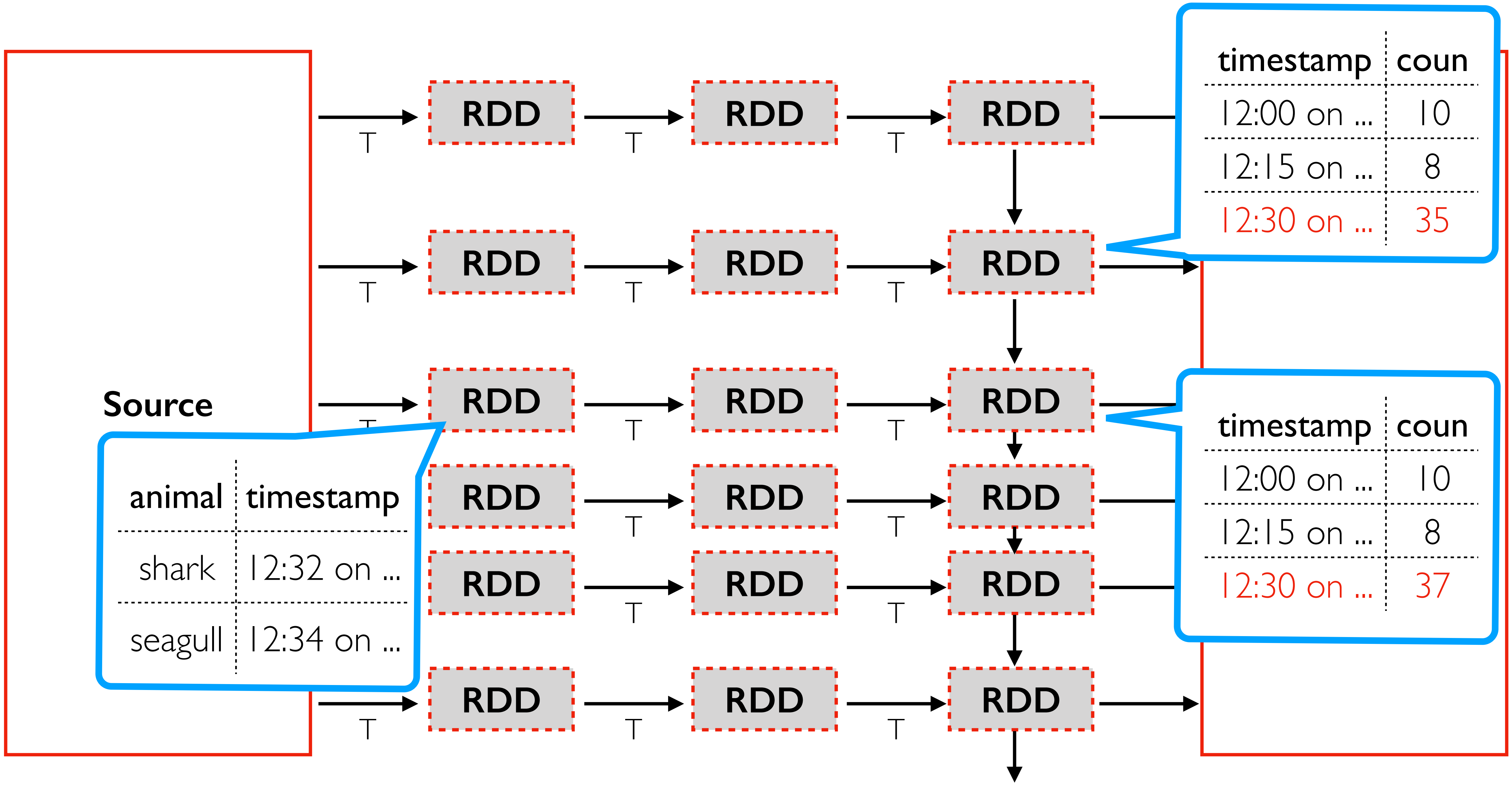
Watermarks

Pivoting

Joining

Exactly-Once Semantics

Grouping By Time Intervals

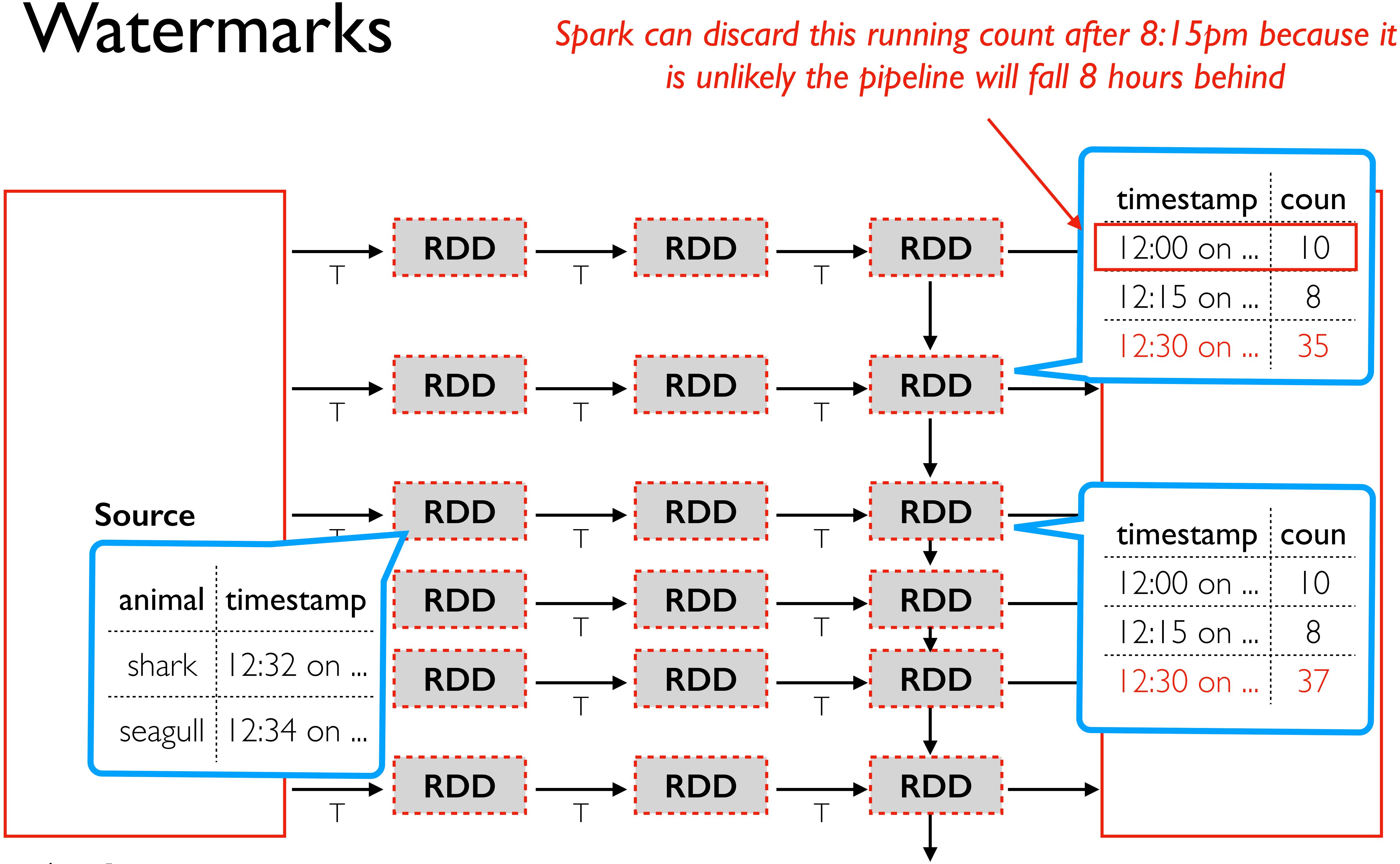



```
(animals
 .groupBy(window("timestamp",
                 "15 minute))
 .count())
```

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




```
(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

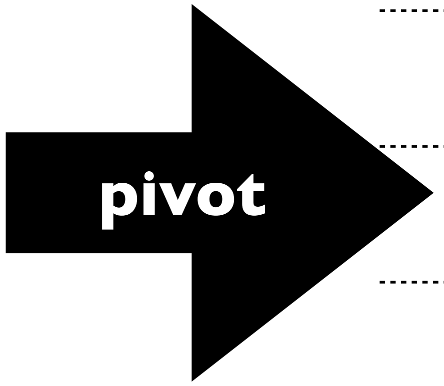
Pivoting

Joining

Exactly-Once Semantics

Pivots

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



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

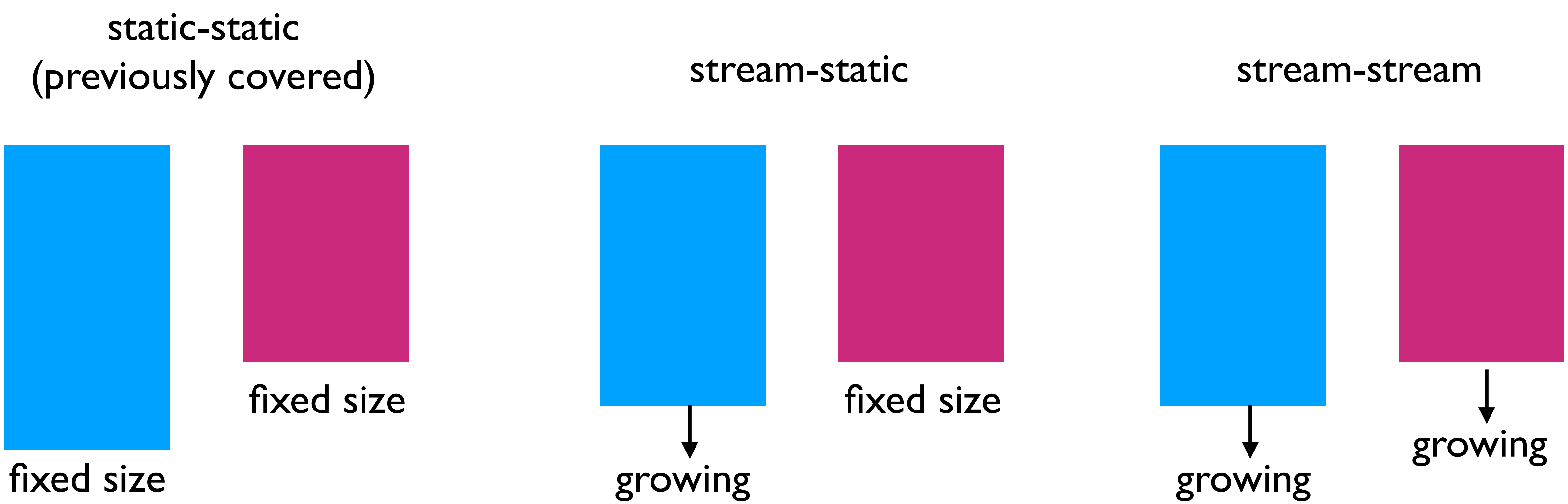
Watermarks

Pivoting

Joining

Exactly-Once Semantics

JOIN Scenarios

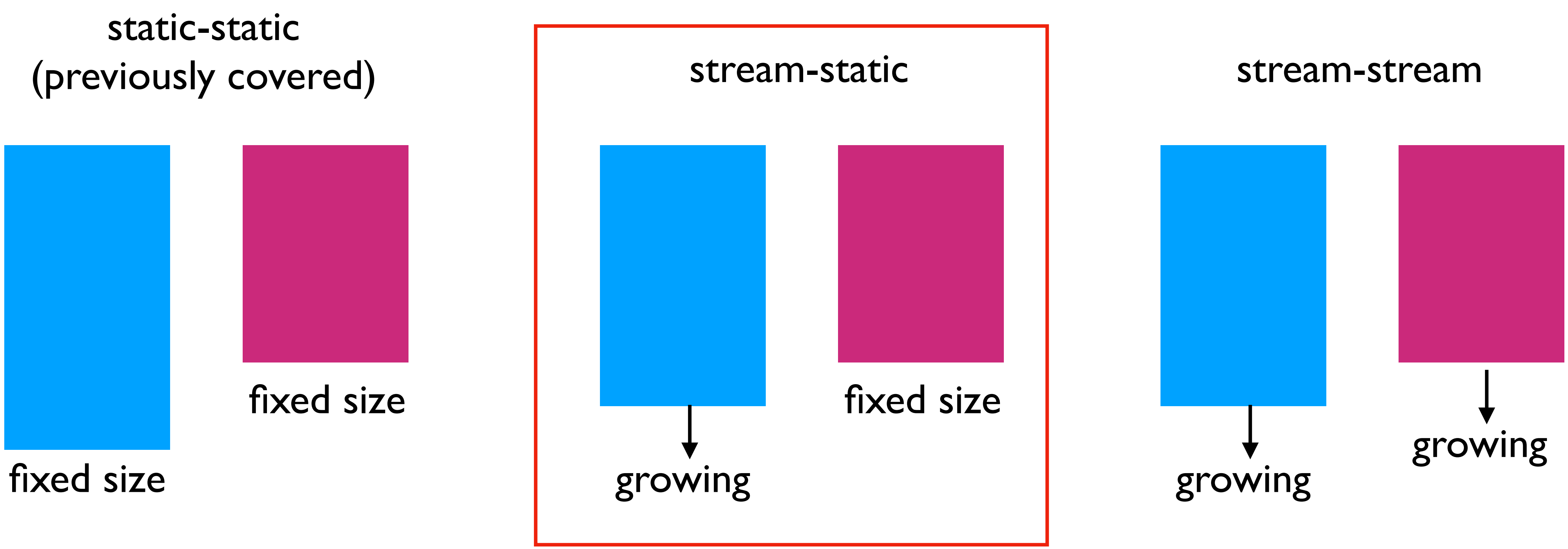


static-static review:

- 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
```

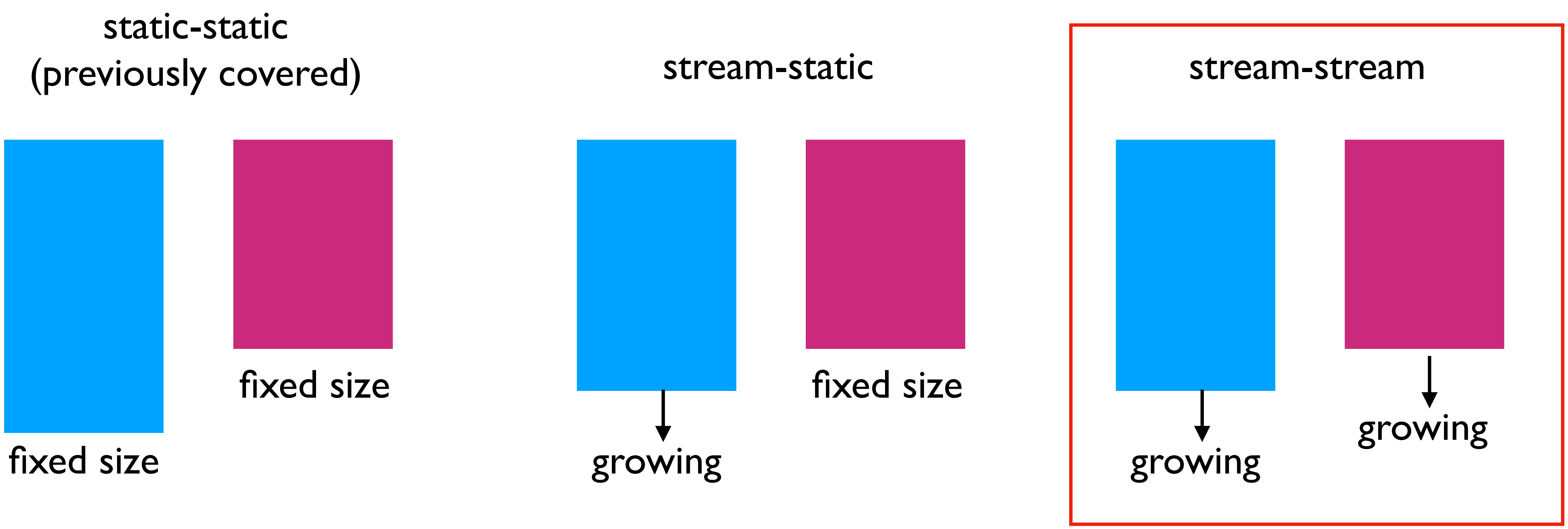
results

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