Streaming Systems

Instructor: Matei Zaharia

cs245.stanford.edu

Outline

Motivation

Streaming query semantics

Query planning & execution

Fault tolerance

Parallel processing

Outline

Motivation

Streaming query semantics

Query planning & execution

Fault tolerance

Parallel processing

Motivation

Many datasets arrive in real time, and we want to compute queries on them continuously (efficiently update result)

Example Query 1

Users visit pages and we want to compute # of visits to each page by hour

```
SELECT page,
       FORMAT(time, “YYYYMMDD-HH”) AS hour,
       COUNT(*) AS cnt
FROM visits
GROUP BY page, hour
```

Example Query 2

Users visit pages and we want to compute # of visits by hour and user’s service plan

```
SELECT users.plan,
  FORMAT(visits.time, “YYYYMMDD-HH”) AS hour,
  COUNT(*) AS cnt
FROM visits JOIN users
GROUP BY users.plan, hour
```

Challenges

1. What do these queries even mean?
   - » E.g. in Q2, what if a user's `plan` attribute changes over time?
   - » Even in Q1, what is "time" – the time of the visit or the time we got the event?

2. What does consistency mean here?
   - » Can't say "serializability" since these are infinitely long queries

3. How to implement this in real systems?
   - » Query planning, execution, fault tolerance

Timeline of Streaming Systems

Early 2000s: lots of research on streaming database (SQL) systems

- Stanford's STREAM, Berkeley's TelegraphCQ, MIT's Aurora & Borealis
- Let to several startups, e.g. Truviso, StreamBase

2004-2011: open source systems including ActiveMQ, Apache Kafka, Storm, Flink, Spark

2017-2020: many of the open source systems add streaming SQL support

Outline

Motivation

Streaming query semantics

Query planning & execution

Fault tolerance

Parallel processing

Streaming Query Semantics

Kind of hard to define!

Many variants out there, but we’ll cover one reasonable set of approaches

» Based on Stanford CQL, Google Dataflow and Spark Structured Streaming

» Combine streams & relations

Streams

A stream is a sequence of tuples, each of which has a special `processing_time` attribute that indicates when it arrives at the system

New tuples in a stream have **non-decreasing** processing times

(user1,	index.html,	2020-01-01 01:00)
(user1,	checkout.html,	2020-01-01 01:20)
(user2,	index.html,	2020-01-01 01:20)
(user2,	search.html,	2020-01-01 01:25)
(user2,	checkout.html,	2020-01-01 01:30)

Relations

We’ll also consider relations in our system, which may change over time

Assume we have serializable transactions, and tuples change when a txn commits

Dealing with Time: Event Time

One subtle issue is that the time when an event occurred in the world may be different than the `processing_time` when we got it

» E.g. clicks on mobile app with slow upload, inventory in a warehouse, etc

Solution: make the real-world time, `event_time`, be an attribute in each record

⇒ Tuples may be **out-of-order** in event time!

Event Time Example

user	page	event_time	processing_time
user1	index.html	01:00	01:00
user1	checkout.html	01:19	01:20
user2	index.html	01:21	01:20
user2	search.html	01:22	01:25
user2	checkout.html	01:23	01:30
user1	search.html	01:15	01:35

Could be out-of-order, maybe even for 1 user;

Could be incorrect clock

Always non-decreasing, set via DB system clock

Queries on Event Time

Event time is just another attribute, so you can use group by, etc:

```
SELECT page,
       FORMAT(event_time, “YYYYMMDD-HH”) AS hour,
       COUNT(*) AS cnt
FROM visits
GROUP BY page, hour
```

What if records keep arriving really late?

Bounding Event Time Skew

Some systems allow setting a max delay on late records to avoid keeping an unbounded amount of state for event time queries

Usually combined with **watermarks**: track event times *currently* being processed and set the threshold based on that

- Helps handle case of processing system being slow!
- E.g. min `event_time` allowed = (min seen in past 5 minutes) – 30 minutes

Back to Streams & Relations

What does it mean to do a query on a stream?

```
SELECT * FROM visits WHERE page="checkout.html"
```

→ Easy, the output is a stream…

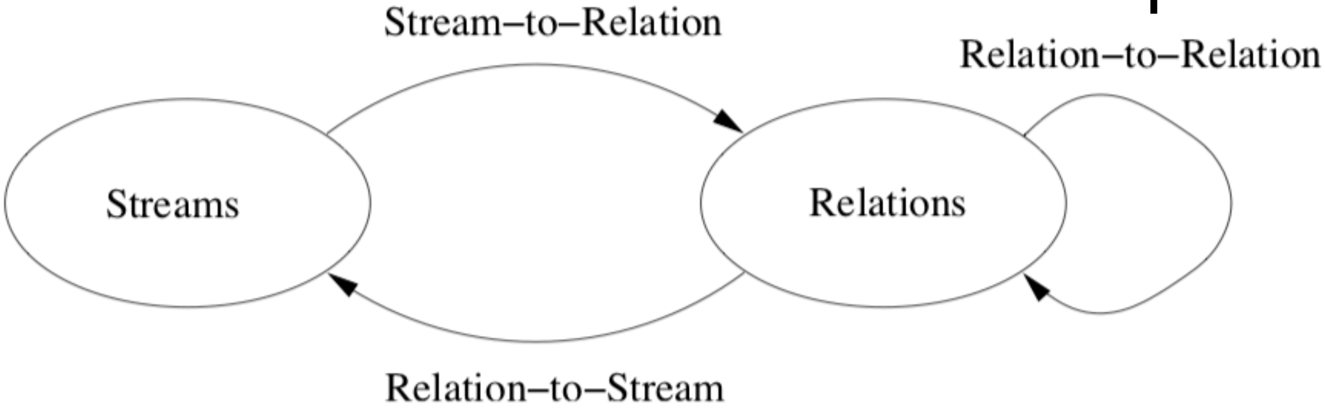
```
SELECT page, COUNT(*) FROM visits GROUP BY page
```

→ What is the output? A relation?

Stanford CQL Semantics

CQL = Continuous Query Language; research project by our dean Jennifer Widom!

“SQL on streams” semantics based on SQL over relations + stream ↔ relation operators



CQL Stream-to-Relation Ops

Windowing: select a contiguous range of a stream in processing time

Time-based window: `S [RANGE T]`

» E.g. `visits [range 1 hour]`

All visits with processing time in the past hour

Tuple-based window: `S [ROWS N]`

» E.g. `visits [rows 10]`

Last 10 visits received at system

Partitioned: `S [PARTITION BY attrs ROWS N]`

» E.g. `visits [partition by page rows 1]`

Last visit received for each page

CQL Stream-to-Relation Ops

Many downstream operations could only be done on bounded windows!

CQL also allows `S [RANGE UNBOUNDED]` but not all operations are allowed after that

- » Only those that can be done with a finite amount of *state*; we'll see more on this later

CQL Relation-to-Relation Ops

All of SQL! Join, select, aggregate, etc

CQL Relation-to-Stream Ops

Capture changes in a relation (each relation has a different version at each proc. time t):

`ISTREAM(R)` contains a tuple (s, t) when tuple s was inserted in R at processing time t.

`DSTREAM(R)` contains (s, t) whenever tuple s was deleted from R at processing time t

`RSTREAM(R)` contains (s, t) for every tuple in R at processing time t

Example Query 1

```
SELECT ISTREAM(*)
FROM visits [RANGE UNBOUNDED]
WHERE page=“checkout.html”
```

Returns a **stream** of all visits to checkout

- Step 1: convert `visits` stream to a relation via “`[RANGE UNBOUNDED]`” window
- Step 2: selection on this relation ($\sigma_{page=checkout}$)
- Step 3: convert the resulting relation to an `ISTREAM` (just output new items)

Example Query 2

```
SELECT *
FROM visits [RANGE UNBOUNDED]
WHERE page="checkout.html"
```

Maintains a **table** of all visits to checkout

- » Step 1: convert `visits` stream to a relation via "`[RANGE UNBOUNDED]`" window
- » Step 2: selection on this relation ($\sigma_{\text{page=checkout}}$)

Note: table may grow indefinitely over time

Example Query 3

```
SELECT page, COUNT(*)
FROM visits [RANGE 1 HOUR]
GROUP BY page
```

Maintains a **table** of visit counts by page for the past 1 hour (in processing time)

» Step 1: convert `visits` stream to a relation via "`[RANGE 1 HOUR]`" window

» Step 2: aggregation on this relation

Example Query 4

```
SELECT page,
       FORMAT(event_time, …) AS hour,
       COUNT(*)
FROM visits [RANGE UNBOUNDED]
GROUP BY page, hour
```

Maintains a **table** of visit counts by page and by hour of event time

This table will grow indefinitely unless we bound event times we accept

Syntactic Sugar in CQL

```
SELECT ISTREAM(*)
FROM visits [RANGE UNBOUNDED]
WHERE page=“checkout.html”
```

↓

```
SELECT * FROM visits WHERE page=“checkout.html”
```

Automatically infer “range unbounded” and “istream” for queries on streams

When Do Stream↔Relation Interactions Happen?

In CQL, every relation has a new version at each **processing time**

Example: joins are against the version at each proc. time, unless you use `RSTREAM` on the table to access an older version

Can also use `RSTREAM` for self-joins of a stream (e.g. what was the user doing 1h ago)

When Does the System Actually Write Output?

In CQL, the system updates all tables or output streams at each processing time (whenever an event or query arrives)

In practice, may want **triggers** for when to output, especially if writing to another system

- » E.g. update visits report only every minute
- » E.g. update visits by event-time only after the watermark for that event-time passes

Google Dataflow Model

More recent API, used at Google and open sourced (API only) as Apache Beam

Somewhat simpler approach: streams only, but can still output either streams or relations

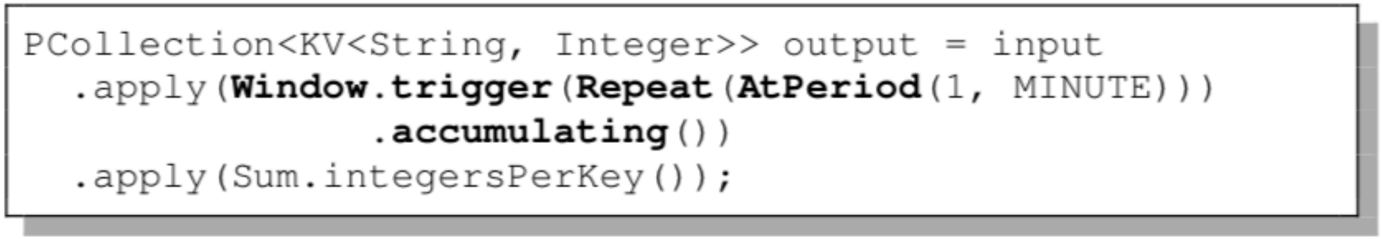
Many operators and features specifically for event time & windowing

Google Dataflow Model

Each operator has several properties:

- **Windowing:** how to group input tuples (can be by processing time or event time)
- **Trigger:** when the operator should output data downstream
- **Incremental processing mode:** how to pass changing results downstream (e.g. retract an old result due to late data)

Example



```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
               .accumulating())
  .apply(Sum.integersPerKey());
```

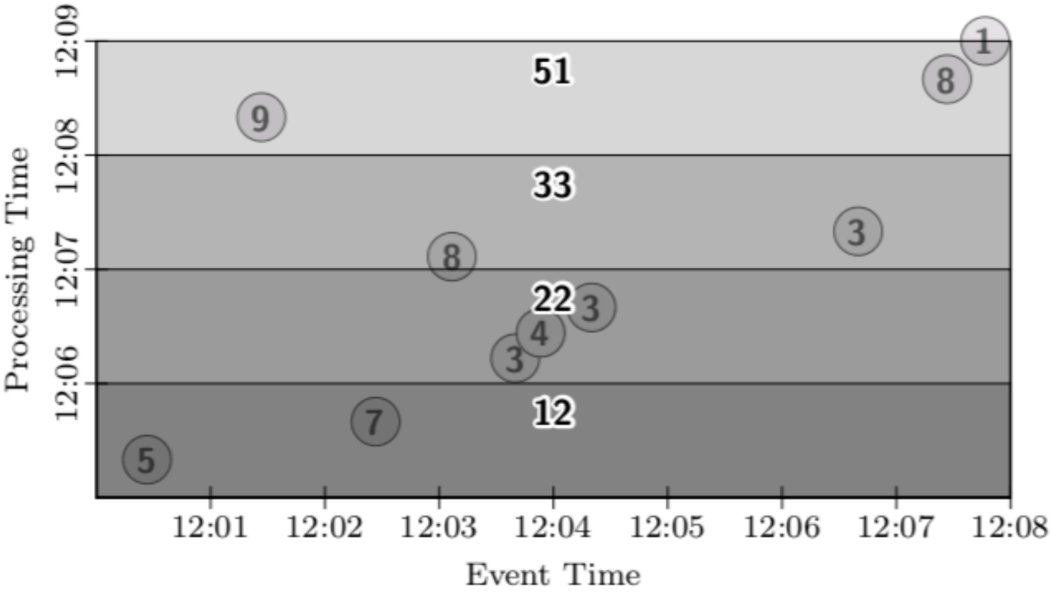
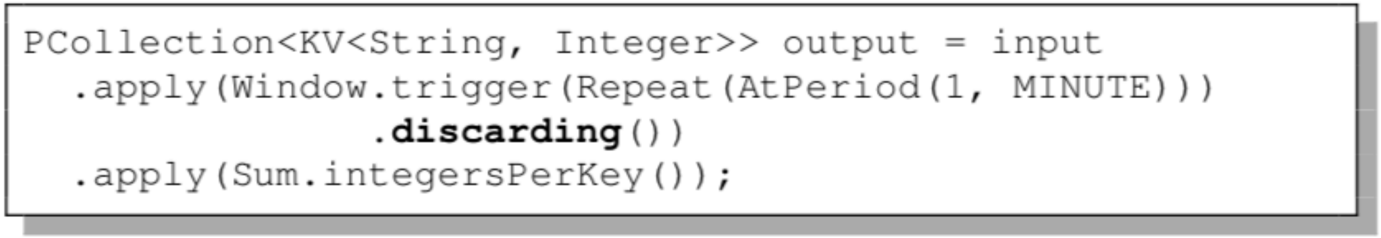


Figure 7: GlobalWindows, AtPeriod, Accumulating

Example



```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
               .discarding())
  .apply(Sum.integersPerKey());
```

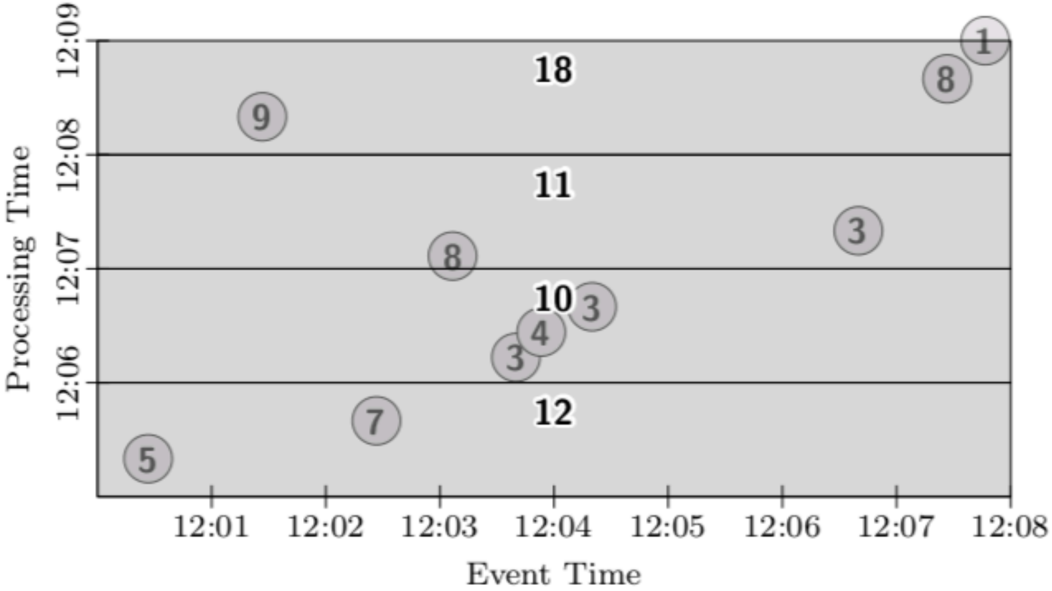
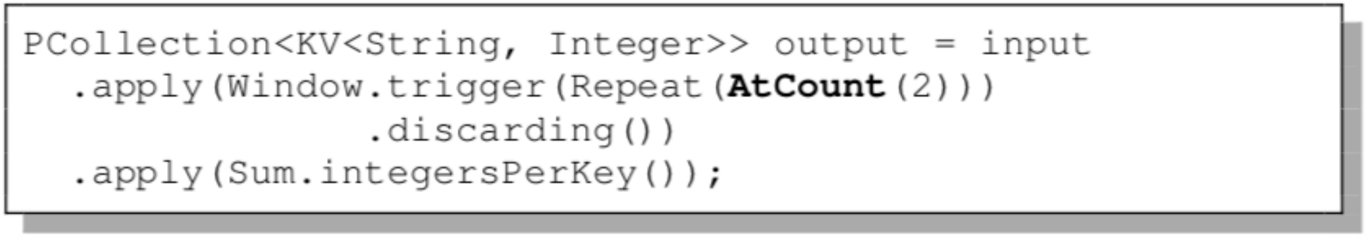


Figure 8: GlobalWindows, AtPeriod, Discarding

Example



```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtCount(2)))
                .discarding())
  .apply(Sum.integersPerKey());
```

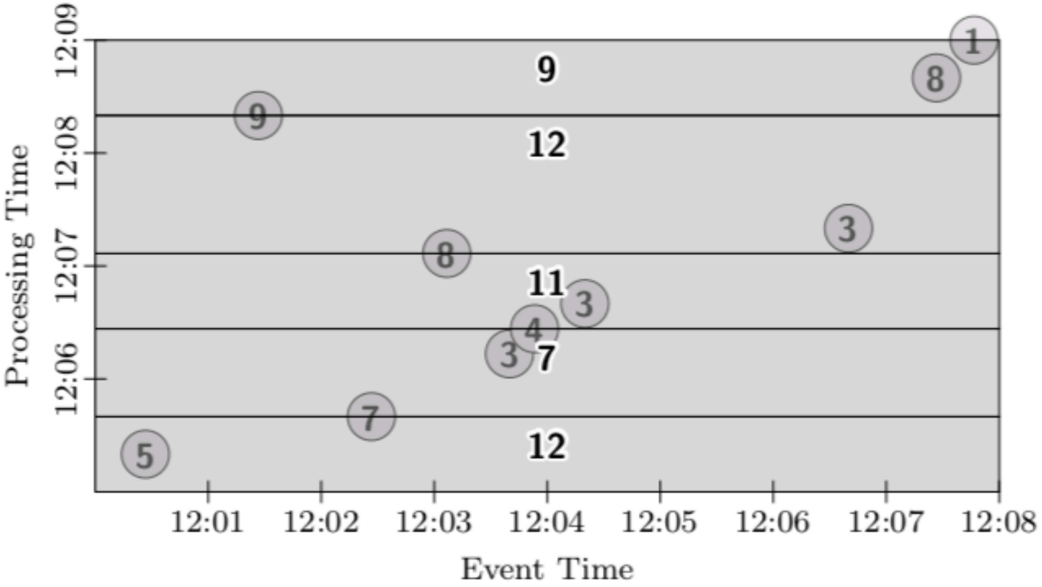
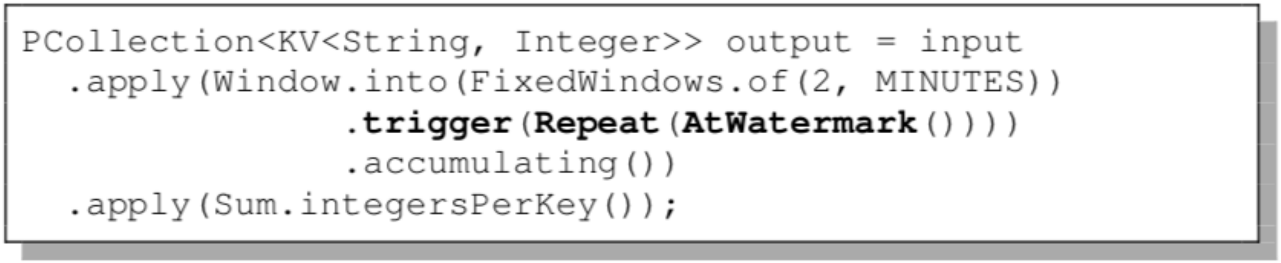


Figure 9: GlobalWindows, AtCount, Discarding

Example



```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(FixedWindows.of(2, MINUTES))
               .trigger(Repeat(AtWatermark()))
               .accumulating())
  .apply(Sum.integersPerKey());
```

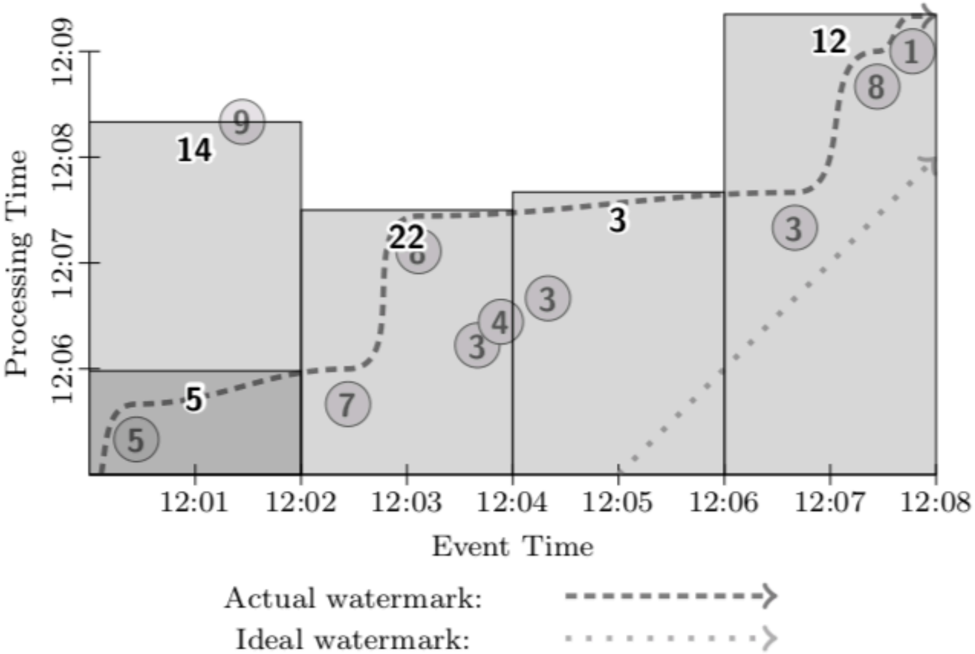
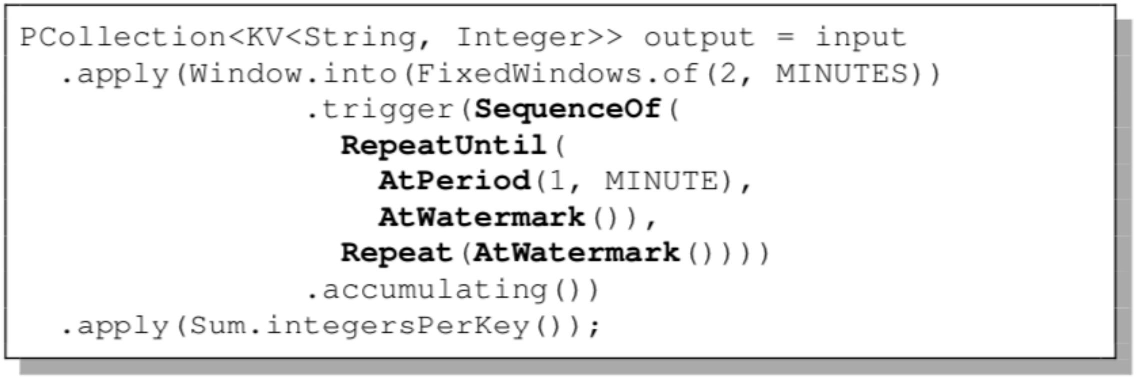


Figure 12: FixedWindows, Streaming

Example



```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(FixedWindows.of(2, MINUTES))
               .trigger(SequenceOf(
                 RepeatUntil(
                   AtPeriod(1, MINUTE),
                   AtWatermark()),
                 Repeat(AtWatermark())))
               .accumulating())
  .apply(Sum.integersPerKey());
```

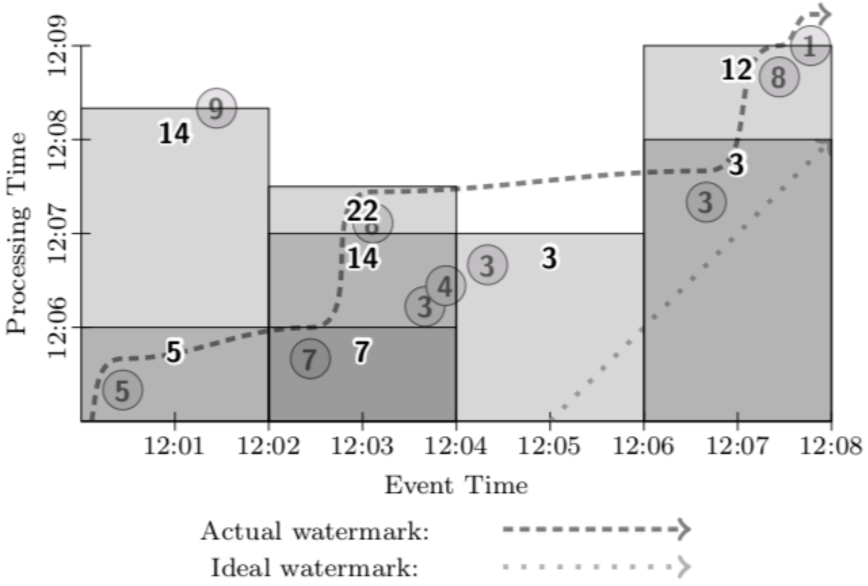


Figure 13: FixedWindows, Streaming, Partial

Spark Structured Streaming

Even simpler model: specify an end-to-end SQL query, triggers, and output mode

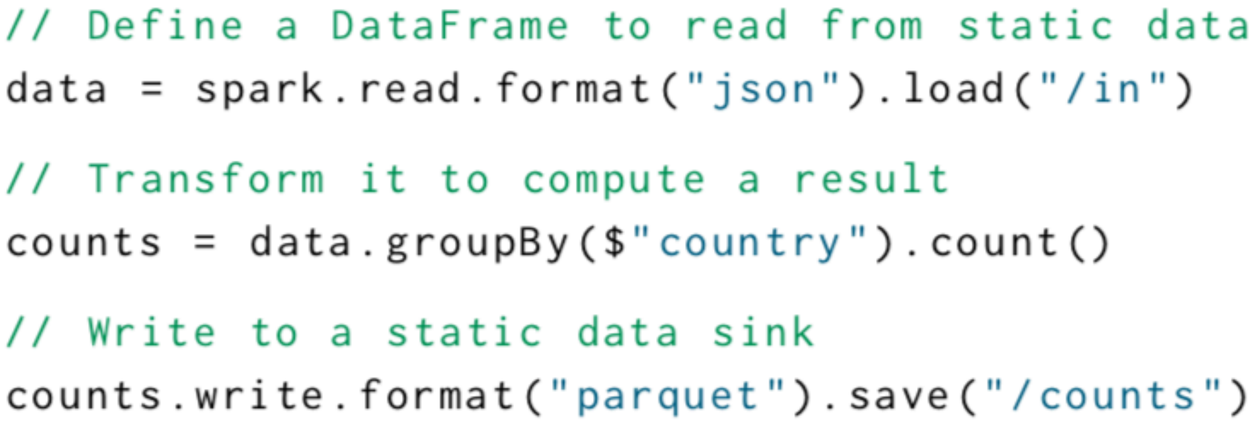
- » Spark will automatically incrementalize query

Spark Structured Streaming

Even simpler model: specify an end-to-end SQL query, triggers, and output mode

» Spark will automatically incrementalize query

Example Spark SQL batch query:



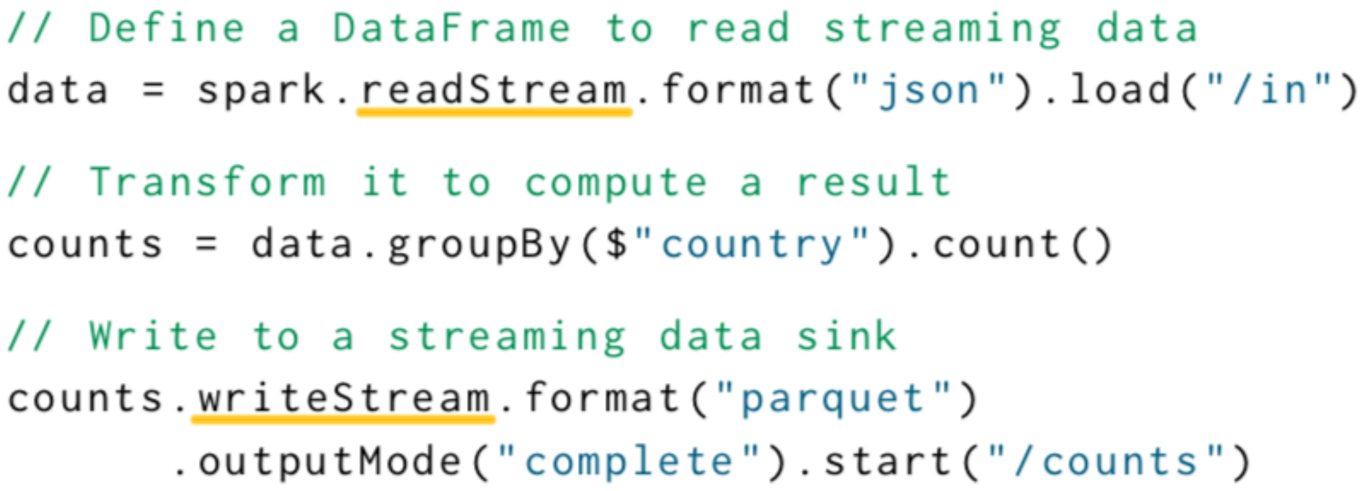
```
// Define a DataFrame to read from static data
data = spark.read.format("json").load("/in")

// Transform it to compute a result
counts = data.groupBy($"country").count()

// Write to a static data sink
counts.write.format("parquet").save("/counts")
```

Spark Structured Streaming

Even simpler model: specify an end-to-end SQL query, triggers, and output mode

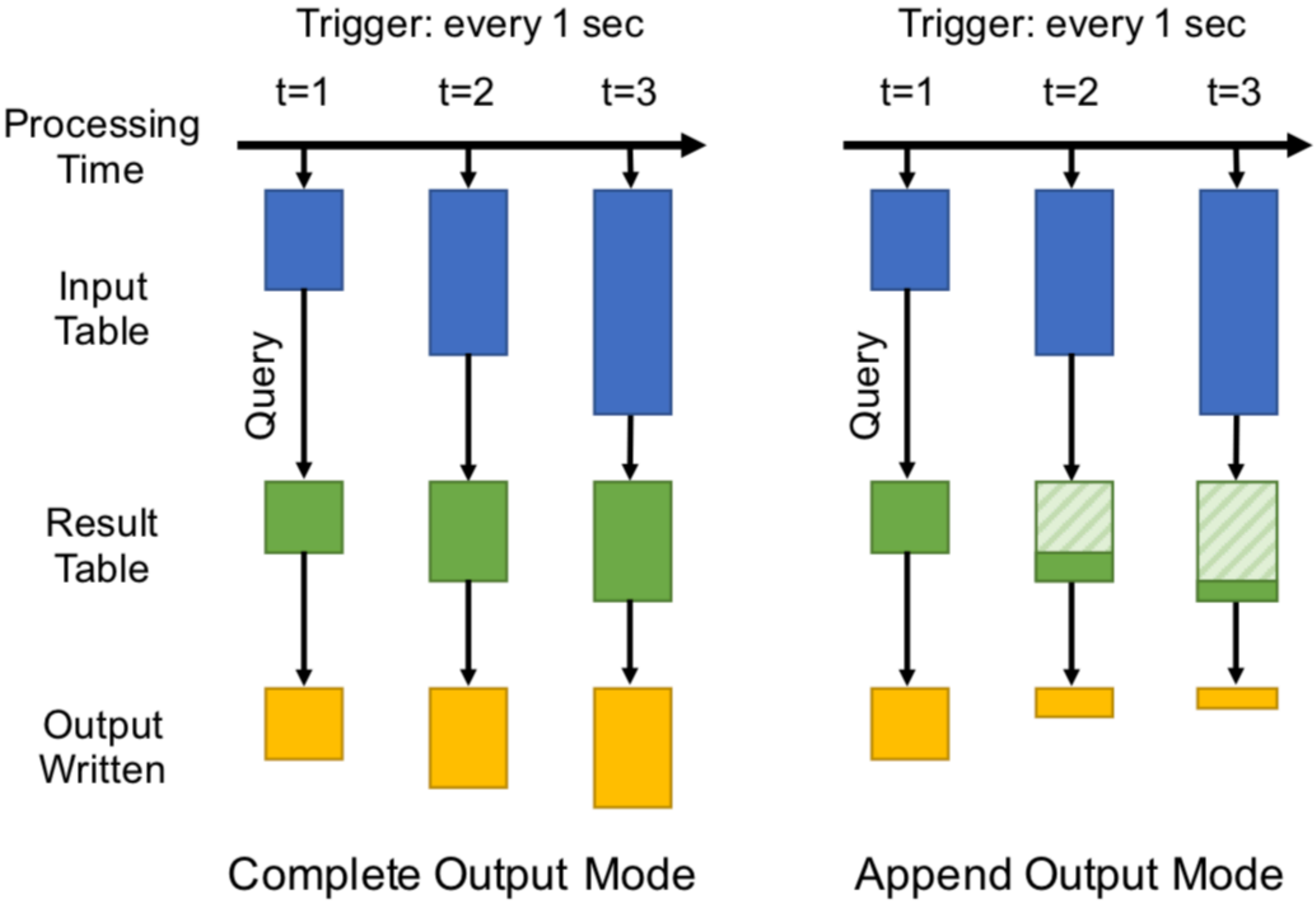
- » Spark will automatically incrementalize query

Spark SQL streaming query:



```
// Define a DataFrame to read streaming data
data = spark.readStream.format("json").load("/in")

// Transform it to compute a result
counts = data.groupBy($"country").count()

// Write to a streaming data sink
counts.writeStream.format("parquet")
      .outputMode("complete").start("/counts")
```

Query Semantics



Other Streaming API Features

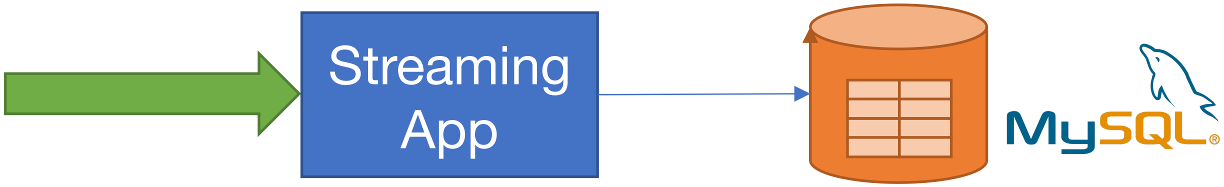
Session windows: each window is a user session (e.g. 2 events count as part of the same session if they are <30 mins apart)

Custom stateful operators: let users write custom functions that maintain a “state” object for each key

Outputs to Other Systems

CQL had a “closed world” model where all relations are in the DB, but this is unrealistic

In general, if you output data to another system, you either need transactions on that system or “at least once” outputs



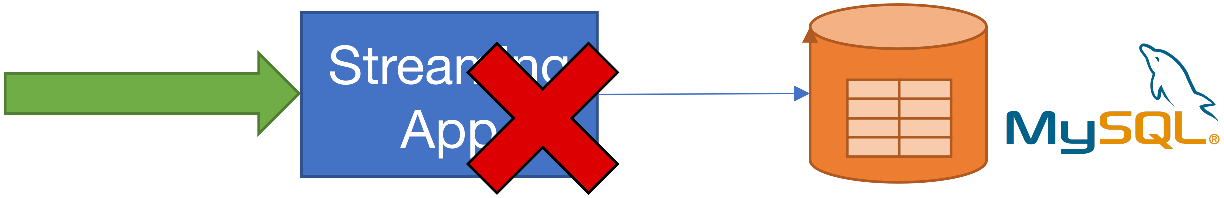
E.g. compute visits by hour

Table read by users

Outputs to Other Systems

CQL had a “closed world” model where all relations are in the DB, but this is unrealistic

In general, if you output data to another system, you either need transactions on that system or “at least once” outputs



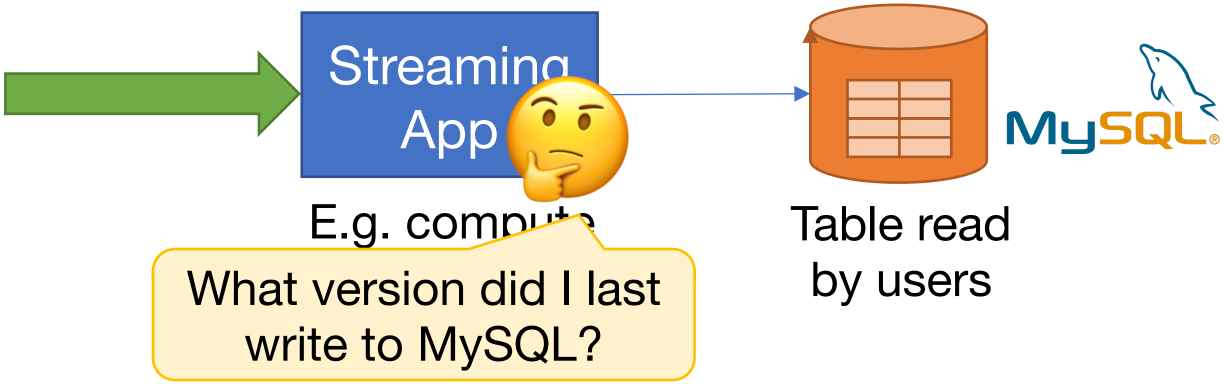
E.g. compute visits by hour

Table read by users

Outputs to Other Systems

CQL had a “closed world” model where all relations are in the DB, but this is unrealistic

In general, if you output data to another system, you either need transactions on that system or “at least once” outputs



Streaming App

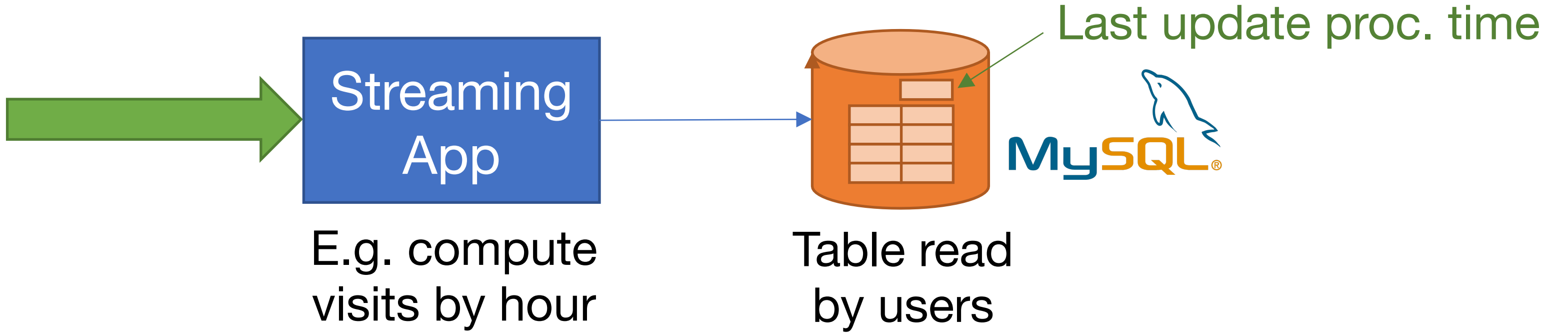
E.g. compute

What version did I last write to MySQL?

Table read by users

Outputs to Other Systems

Transaction approach: streaming system maintains some “last update time” field in the output *transactionally* with its writes



At-least-once approach: for queries that only insert data (maybe by key), just run again from last proc. time known to have succeeded

Outline

Motivation

Streaming query semantics

Query planning & execution

Fault tolerance

Parallel processing

How to Run Streaming Queries?

1) Query planning: convert the streaming query to a set of physical operators
   - » Usually done via rules

2) Execute physical operators
   - » Many of these are “stateful”: must remember data (e.g. counts) across tuples

3) Maintain some state reliably for recovery
   - » Can use a write-ahead log

Streaming Operators

Similar to the physical ops in a batch engine, but some extra ones with (more) state

Examples:

ReadStream
(e.g. from message bus or TCP port)

σ

Π

$\bowtie$
(with bounded event time range)

Aggregate
(maybe with bounded event time range)

WriteStream
(using transactions or at-least-once)

Query Planning

We don’t have time to cover this in detail, but there are good algorithms to “incrementalize” a SQL query

E.g. convert CQL query to windows, `ISTREAM`, `DSTREAM`, and relational ops on bounded-size intermediate tables

Fault Tolerance

Need to maintain:

- What data we **outputted** in external systems (usually, up to which processing time)
- What data we **read** from each source at each proc. time (can also ask sources to replay)
- **State** for operators, e.g. partial count & sum

What order should we log these items in?

Fault Tolerance

Need to maintain:

- What data we **outputted** in external systems (usually, up to which processing time)
- What data we **read** from each source at each proc. time (can also ask sources to replay)
- **State** for operators, e.g. partial count & sum

What order should we log these items in?

- Typically must log what we read at each proc. time **before** we output for that proc. time
- Can log operator state asynchronously if we can replay our input streams

Example: Structured Streaming

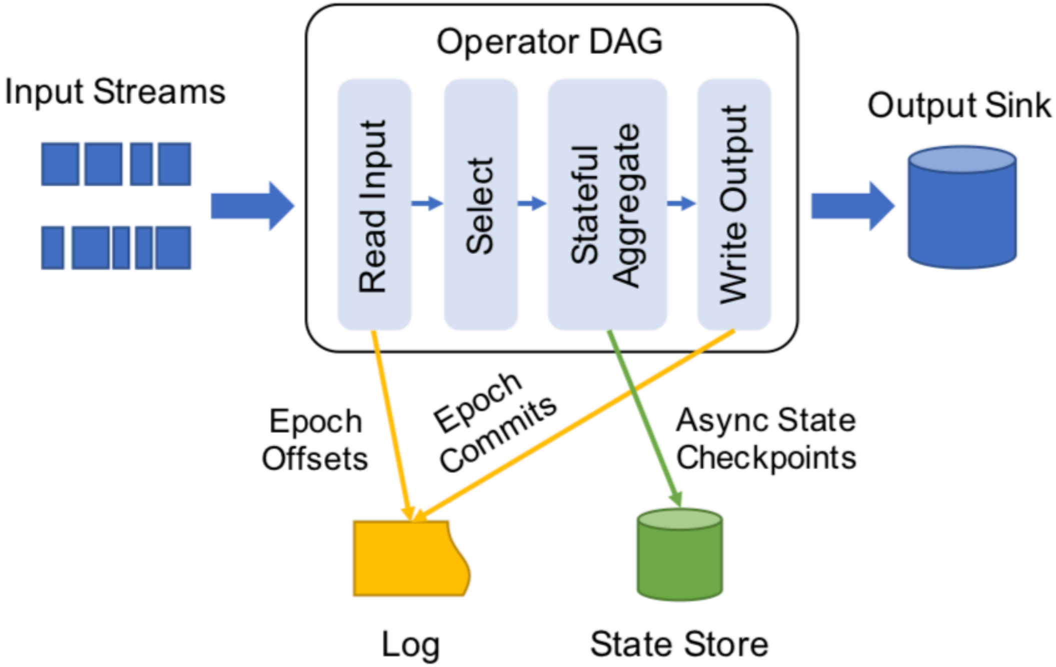


Figure 4: State management during the execution of Structured Streaming. Input operators are responsible for defining epochs in each input source and saving information about them (e.g., offsets) reliably in the write-ahead log. Stateful operators also checkpoint state asynchronously, marking it with its epoch, but this does not need to happen on every epoch. Finally, output operators log which epochs' outputs have been reliably committed to the idempotent output sink; the very last epoch may be rewritten on failure.

Outline

Motivation

Streaming query semantics

Query planning & execution

Fault tolerance

Parallel processing

Parallel Stream Processing

Required for very large streams, e.g. app logs or sensor data

Additional complexity from a few factors:

- » How to recover quickly from faults & stragglers?
- » How to log in parallel?
- » How to write parallel output atomically?
(An issue for parallel jobs in general; see Delta)

Parallel Stream Processing

Typical implementation:

How to recover quickly from faults & stragglers?

- » Split up the recovery work (like MapReduce)

How to log in parallel?

- » Master node can log input offsets for all readers on each "epoch"; state logged asynchronously

How to write parallel output atomically?

- » Use transactions or only offer "at-least-once"

Summary

Streaming apps require a different semantics

They can be implemented using many of the techniques we saw before

- » Rule-based planner to transform SQL ASTs into incremental query plans
- » Standard relational optimizations & operators
- » Write-ahead logging & transactions