One SQL to Rule them All

An Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables

https://arxiv.org/pdf/1905.12133.pdf

Published & presented at SIGMOD 2019

Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, Kenneth Knowles

ApacheCon NA 2019, Presented by Kenneth Knowles & Julian Hyde.

Slides: https://s.apache.org/streaming-sql-apachecon-na-2019

Background

- Flink and Beam communities have been working on SQL support
  - Leveraging Apache Calcite as building block
- All three communities aim for unified batch/streaming semantics for SQL
  - Frequent exchange among communities
  - Agreement on a common model / feature set
- Idea: Write a paper to summarize our ideas and get it peer-reviewed
  - Paper accepted & presented at SIGMOD 2019 Industry Track



Background

- Idea: Write a paper to summarize our ideas and get it peer-reviewed
  - Paper accepted & presented at SIGMOD 2019 Industry Track

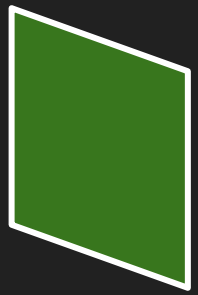
- Paper accepted & presented at SIGMOD 2019 Industry Track!

Our Motivation(s)

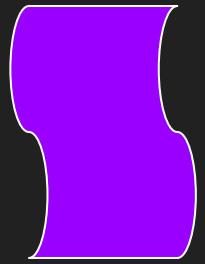
Big Data Research & OSS	***Streaming SQL Standardization***
Share our experience of building OSS streaming systems with SQL support Our software powers many real-world streaming use cases	The SQL Standards Committee is investigating extensions for streaming SQL We want to offer our experience and help to shape the standard

First attempt: CREATE STREAM

One obvious way to solve the problem: streams as first-class SQL objects, distinct from tables, but with similar capabilities.



```
CREATE TABLE PastBids
SELECT * FROM PastBids
```



```
CREATE STREAM NewBids
SELECT * FROM NewBids
```

But there are problems. For example, how to write a query that combines past with future?

What we came up with (1 of 2)

Guiding principles

- Unified semantics for SQL over tables and streams
- Minimal additions to the standard
- Use as much as possible of SQL in a streaming context

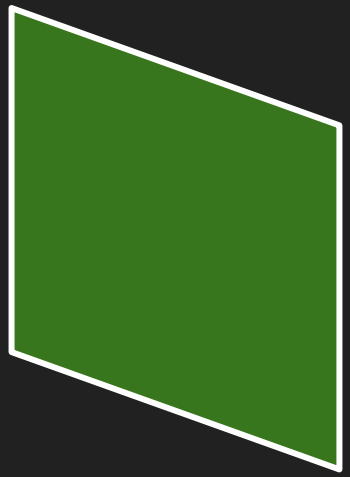
What we came up with (2 of 2)

Three Concrete Proposals:

1. Time-varying relations
2. Event time semantics
3. Controlling the realization of time-varying results

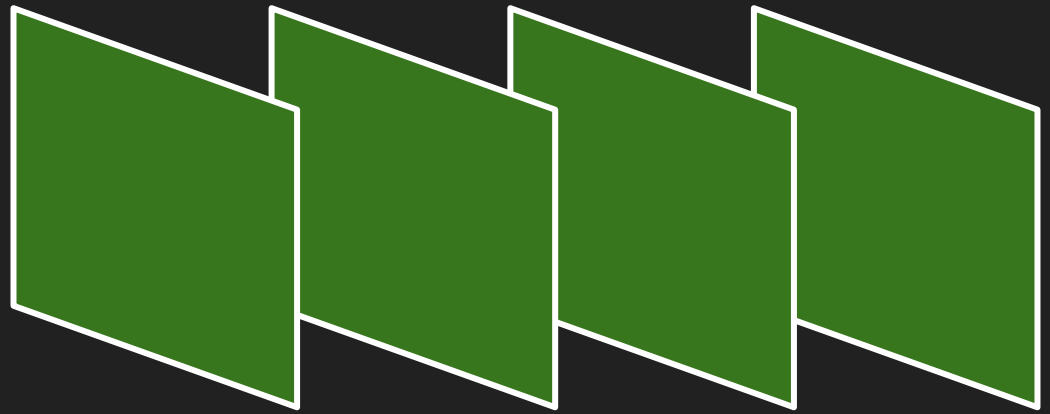
Time-Varying Relations

SQL data: relations

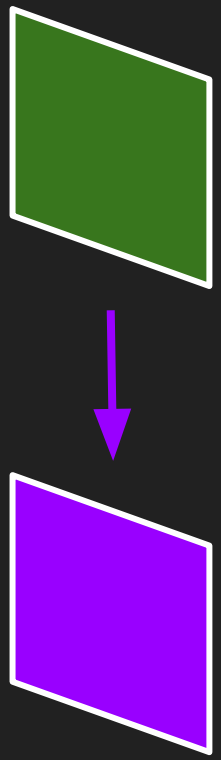


SQL data: time-varying relations (TVRs)

- Table updated by transactional edits
- Stream of events being appended

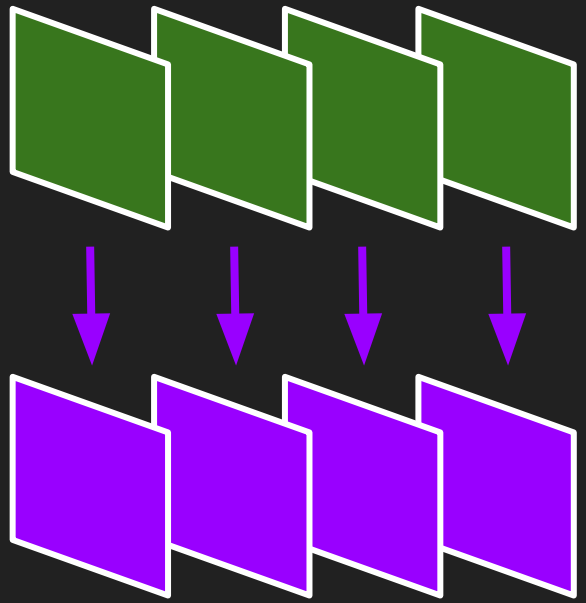


SQL queries map relations to relations



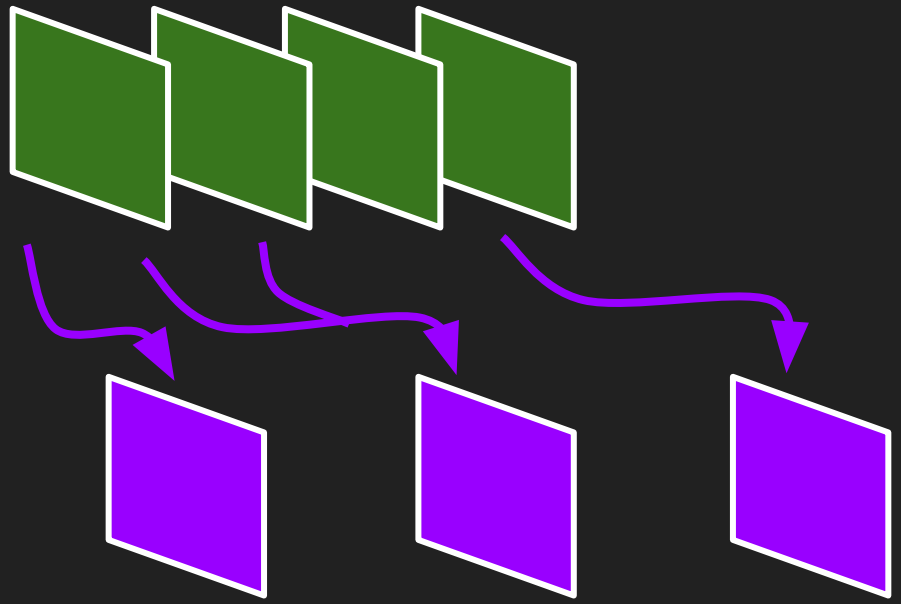
SQL queries apply to time-varying relations

- Classic SQL queries map time-varying relations at each point-in-time
  - Transactional isolation ensures consistency



Streaming SQL queries apply "across" TVRs

- Time is the new dimension of streaming SQL
  - Streaming SQL queries process relations that evolve over time



Key Insight

Streams and Tables are different instances of the same semantic object - a TVR.

*“Streams **are** Tables” instead of “Streams **and** Tables”*

TVR Representations

TVR of auction bids

```
--------------------------
| bidtime | price | item |
--------------------------
```

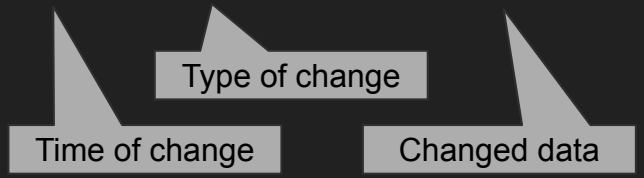
TVR Representations

TVR of auction bids

```
--------------------------
| bidtime | price | item |
--------------------------
```

Stream representation of TVR

```
8:08    INSERT (8:07, $2, A)
8:12    INSERT (8:11, $3, B)
8:13    INSERT (8:05, $4, C)
8:15    INSERT (8:09, $5, D)
8:17    INSERT (8:13, $1, E)
8:18    INSERT (8:17, $6, F)
```



TVR Representations

TVR of auction bids

```
-------------------------
| bidtime | price | item |
-------------------------
```

Stream representation of TVR

```
8:08    INSERT (8:07, $2, A)
8:12    INSERT (8:11, $3, B)
8:13    INSERT (8:05, $4, C)
8:15    INSERT (8:09, $5, D)
8:17    INSERT (8:13, $1, E)
8:18    INSERT (8:17, $6, F)
```

Time of query

Table representation of TVR

```
8:14> SELECT * FROM bids;
-------------------------
| bidtime | price | item |
-------------------------
| 8:07    |    $2 |    A |
| 8:11    |    $3 |    B |
| 8:05    |    $4 |    C |
-------------------------
```

TVR Representations

TVR of auction bids

```
-------------------------
| bidtime | price | item |
-------------------------
```

Stream representation of TVR

```
8:08    INSERT (8:07, $2, A)
8:12    INSERT (8:11, $3, B)
8:13    INSERT (8:05, $4, C)
8:15    INSERT (8:09, $5, D)
8:17    INSERT (8:13, $1, E)
8:18    INSERT (8:17, $6, F)
```

Table representation of TVR

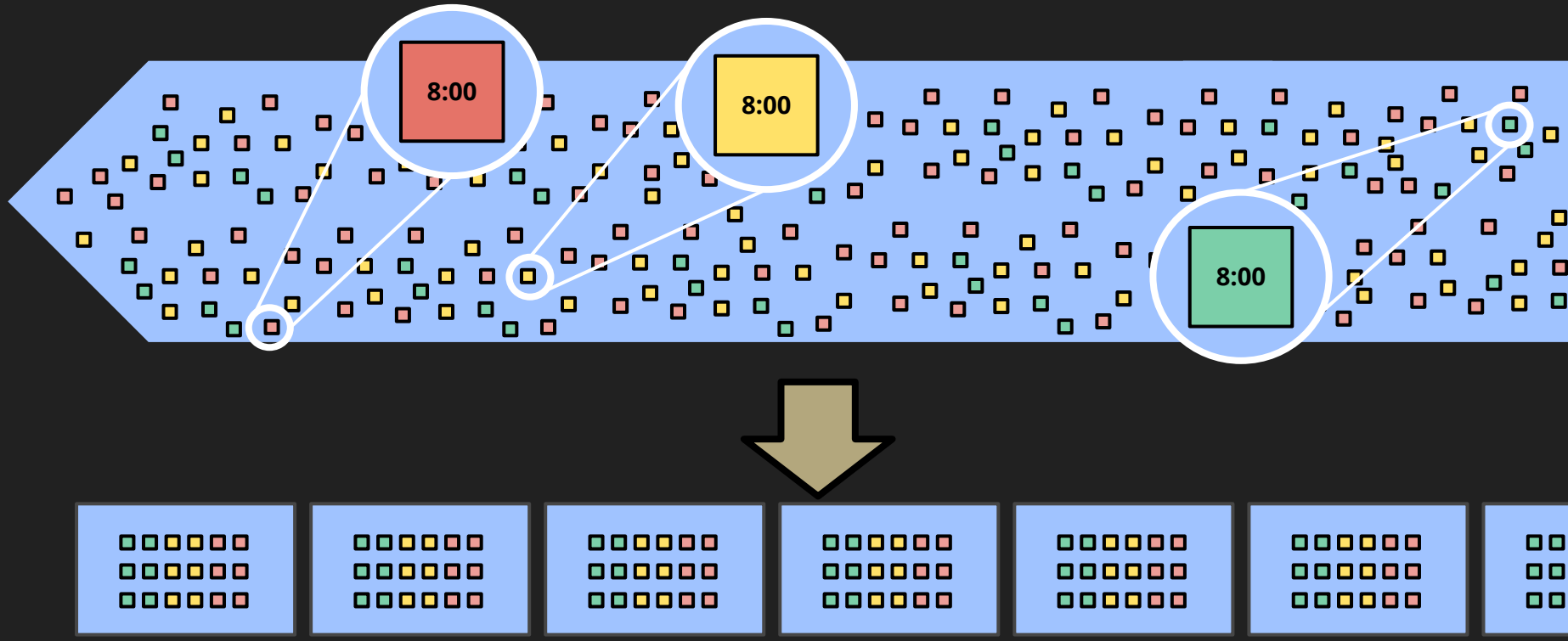
```
8:14> SELECT * FROM bids;
-------------------------
| bidtime | price | item |
-------------------------
| 8:07    |    $2 |    A |
| 8:11    |    $3 |    B |
| 8:05    |    $4 |    C |
-------------------------

8:20> SELECT * FROM bids;
-------------------------
| bidtime | price | item |
-------------------------
| 8:07    |    $2 |    A |
| 8:11    |    $3 |    B |
| 8:05    |    $4 |    C |
| 8:09    |    $5 |    D |
| 8:13    |    $1 |    E |
| 8:17    |    $6 |    F |
-------------------------
```

No SQL Extension

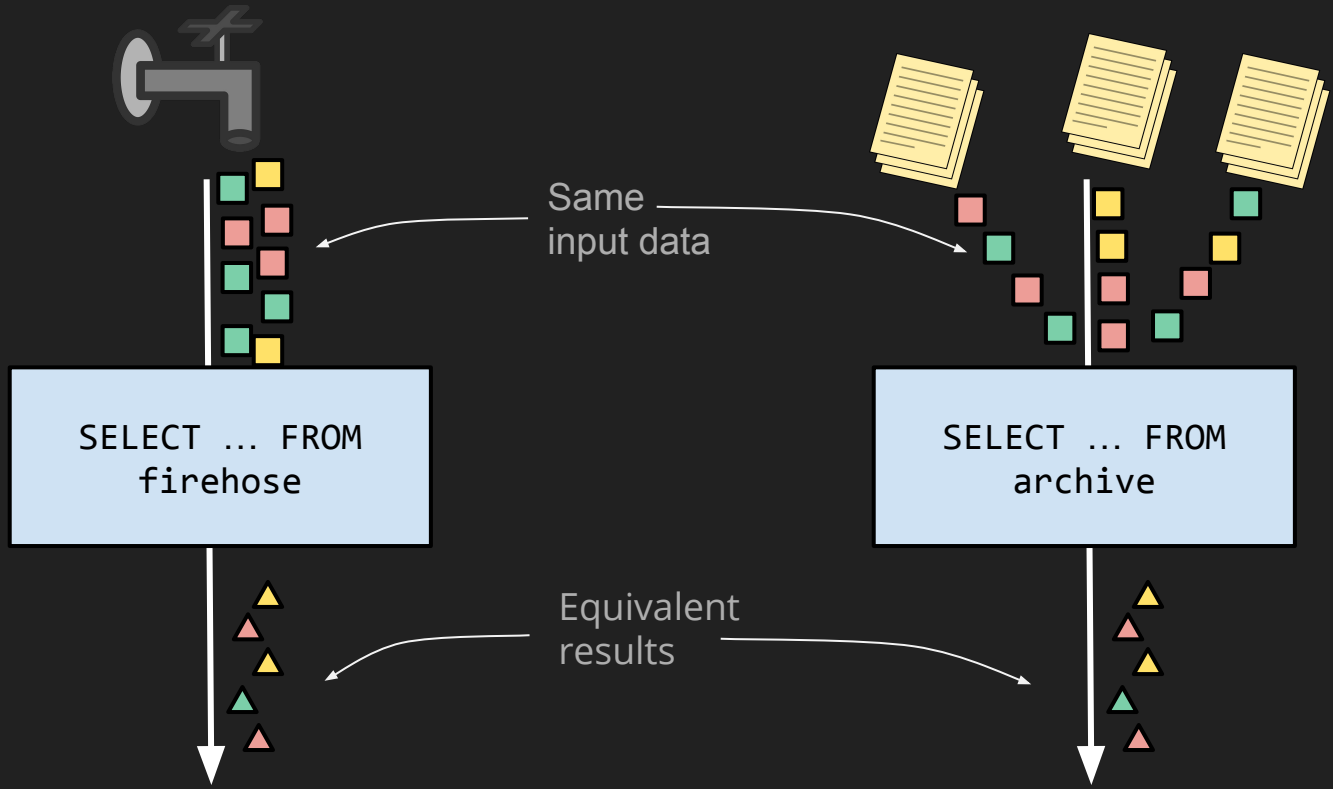
- No SQL extensions needed!
- All SQL operations remain valid on TVRs

Event Time Semantics

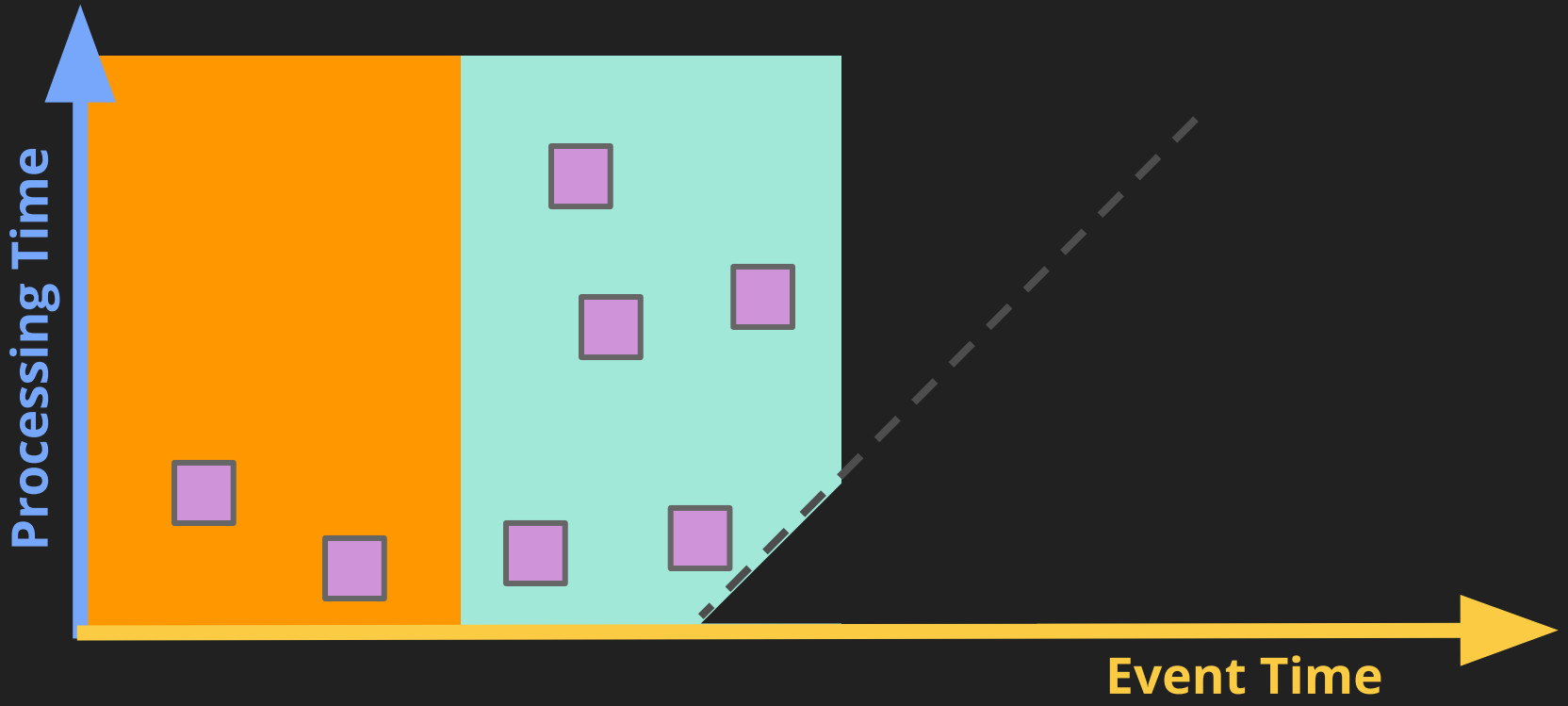
Event Time vs. Processing Time



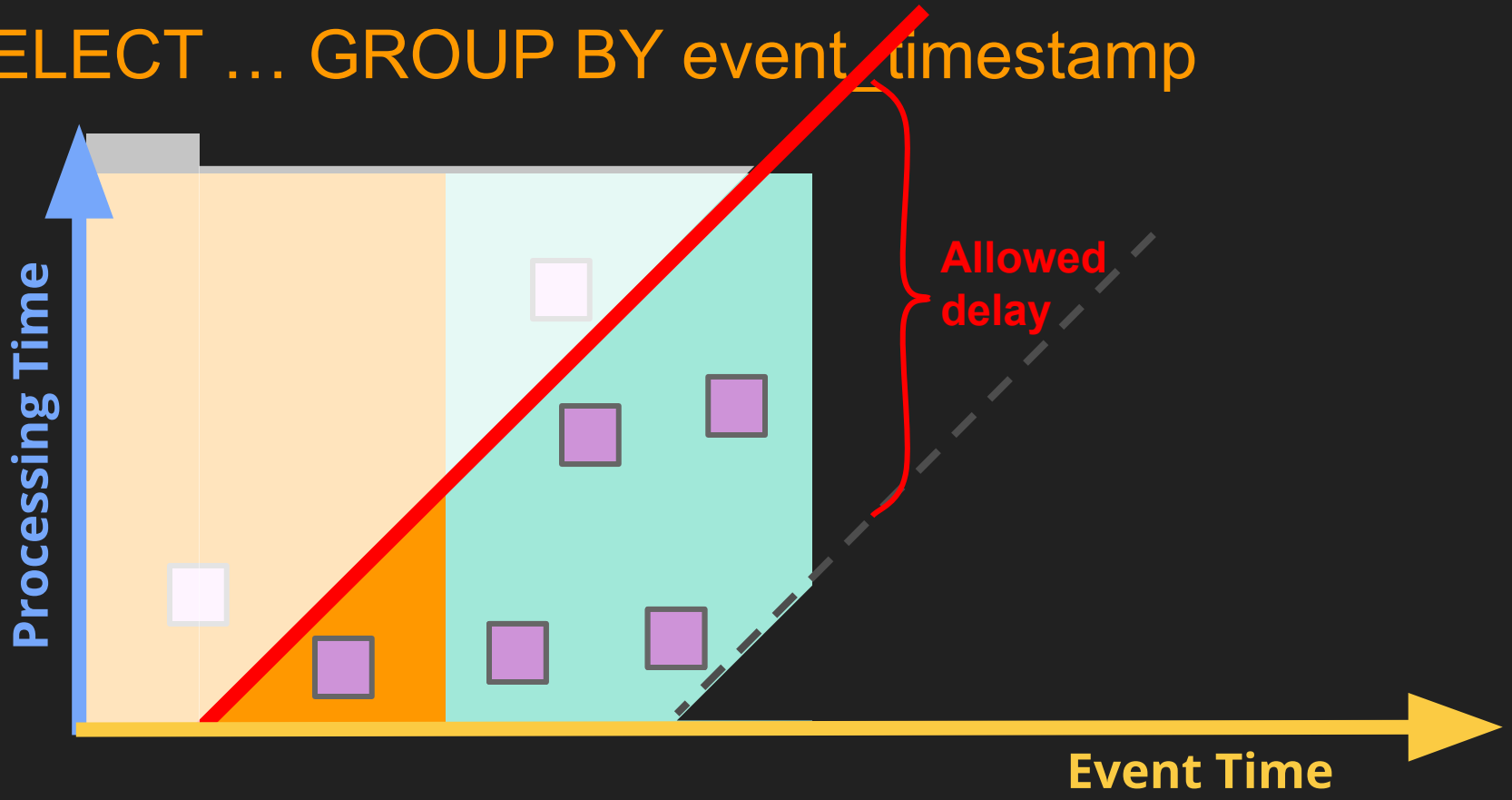
Event Time: unified batch/streaming



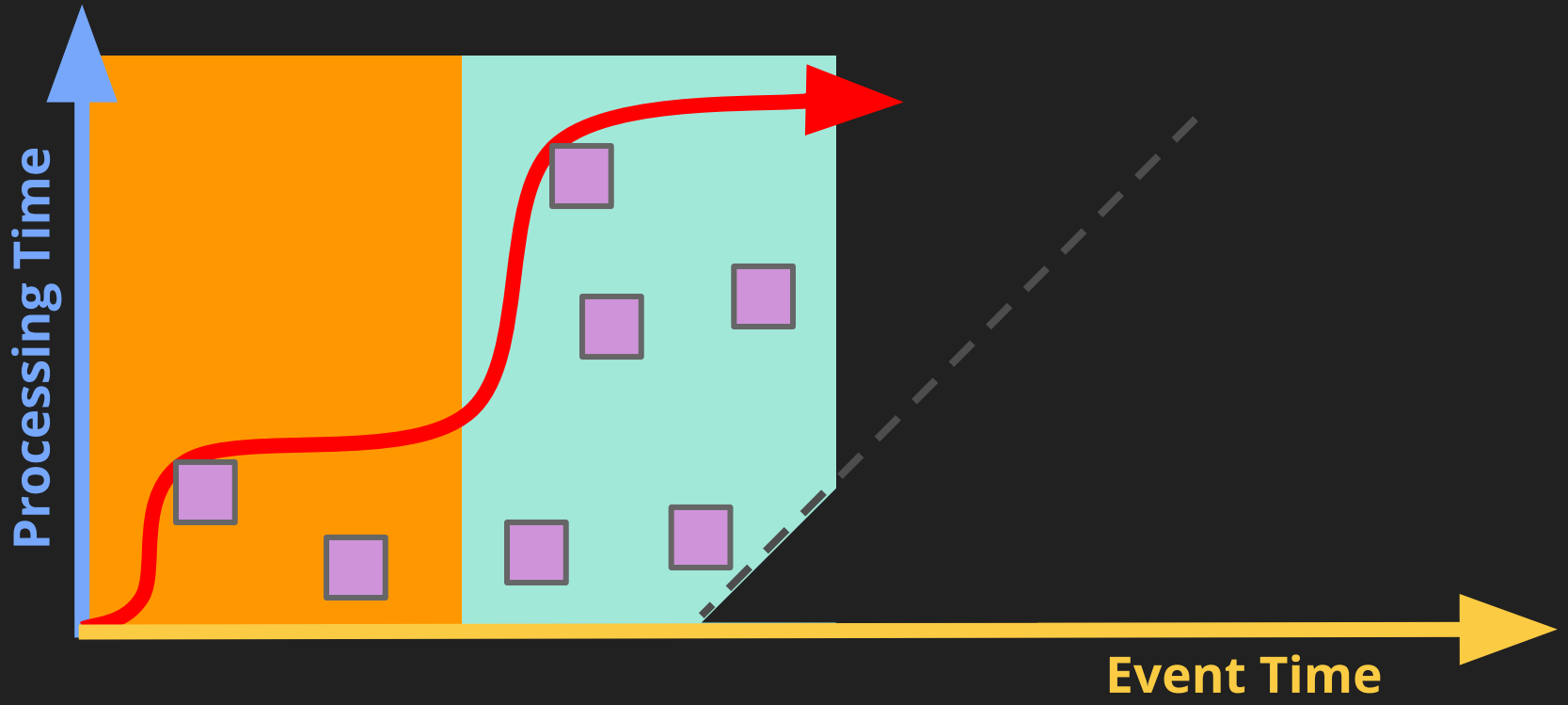
SELECT … GROUP BY event_timestamp



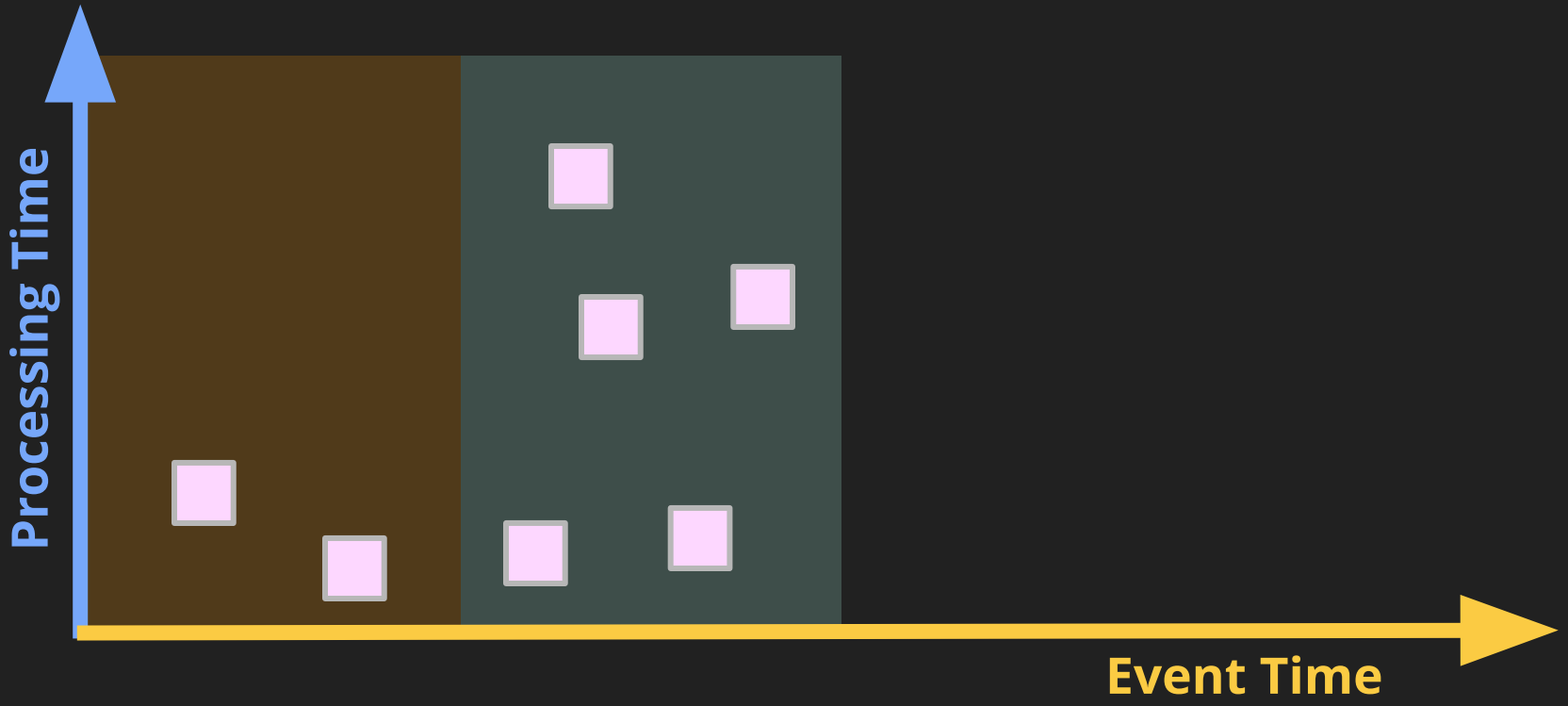
SELECT … GROUP BY event_timestamp



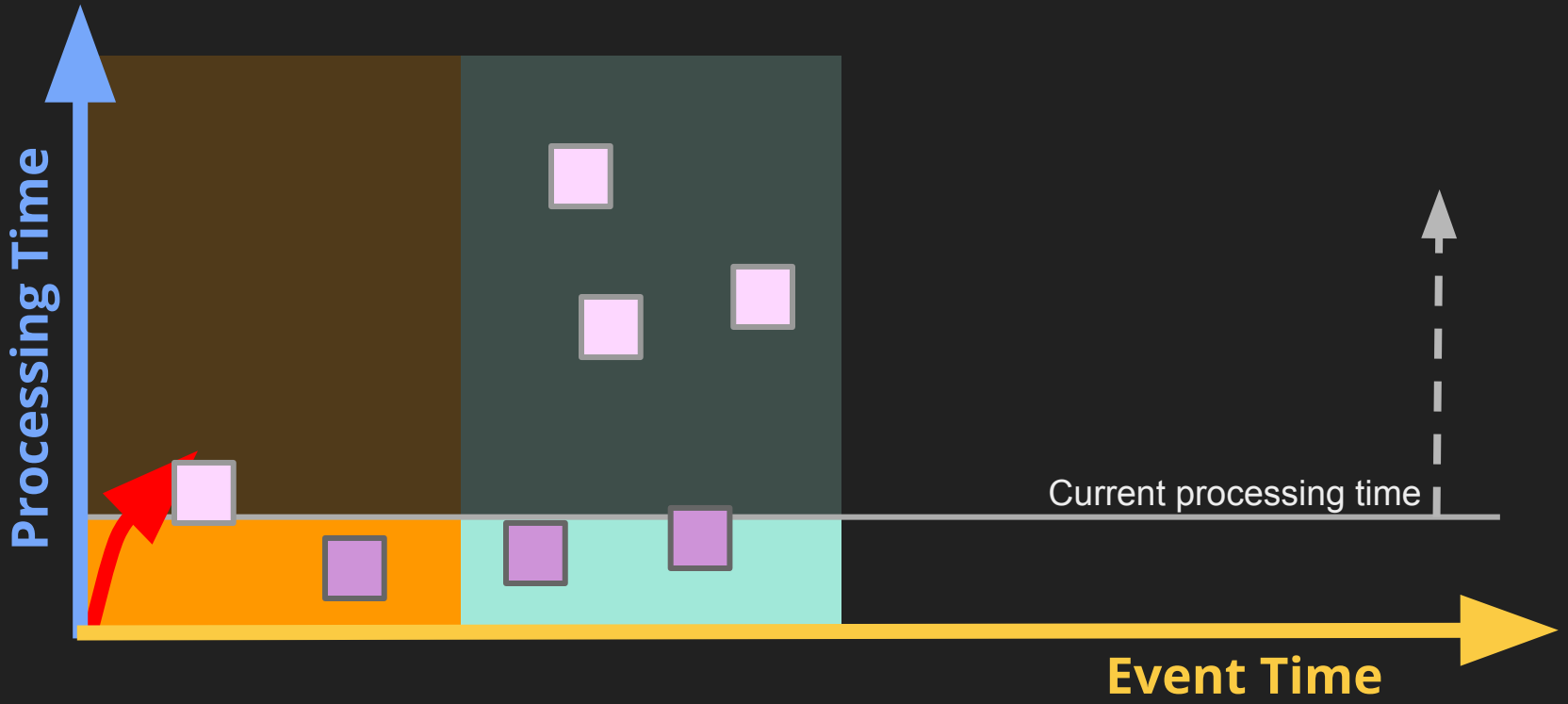
SELECT … GROUP BY event_timestamp



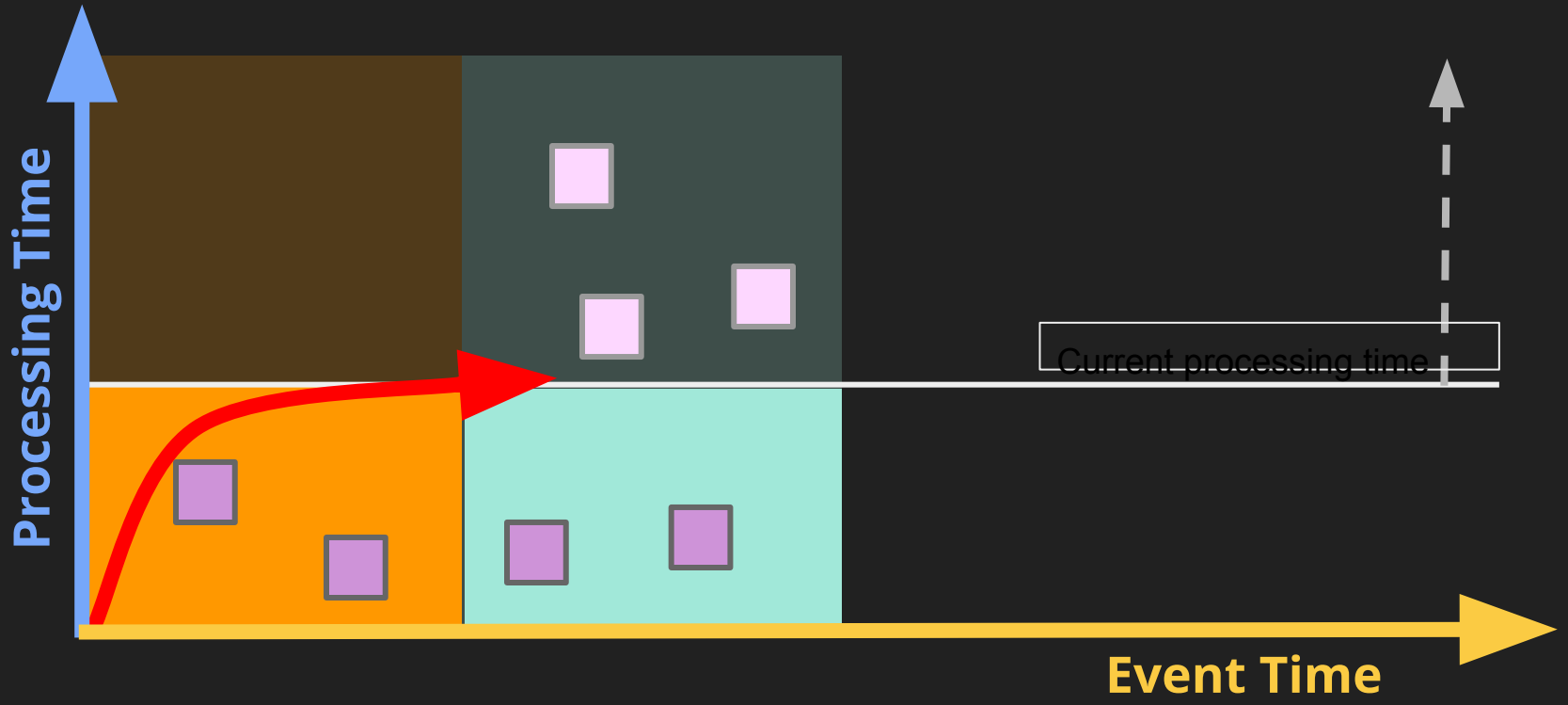
SELECT … GROUP BY event_timestamp



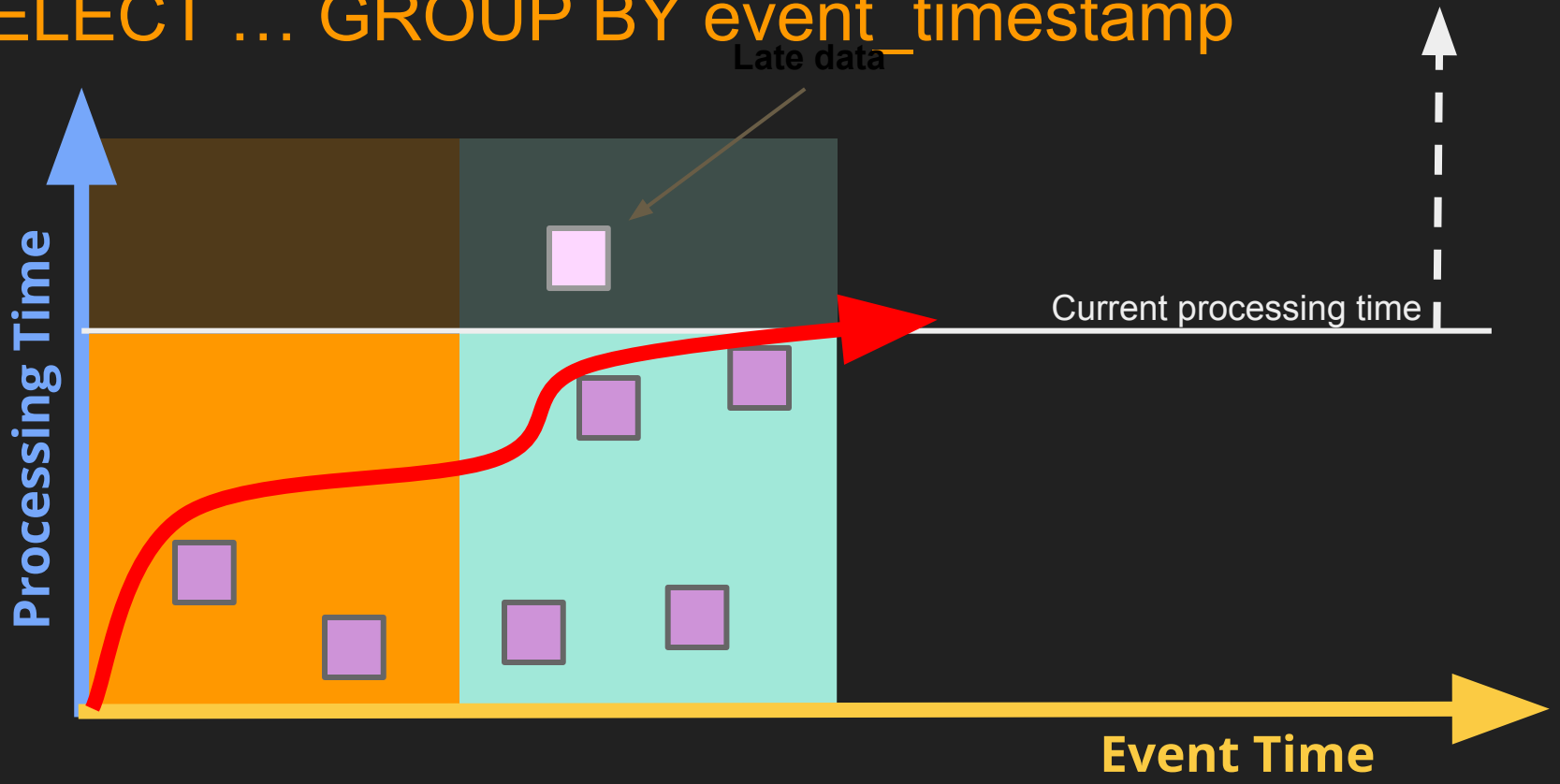
SELECT … GROUP BY event_timestamp



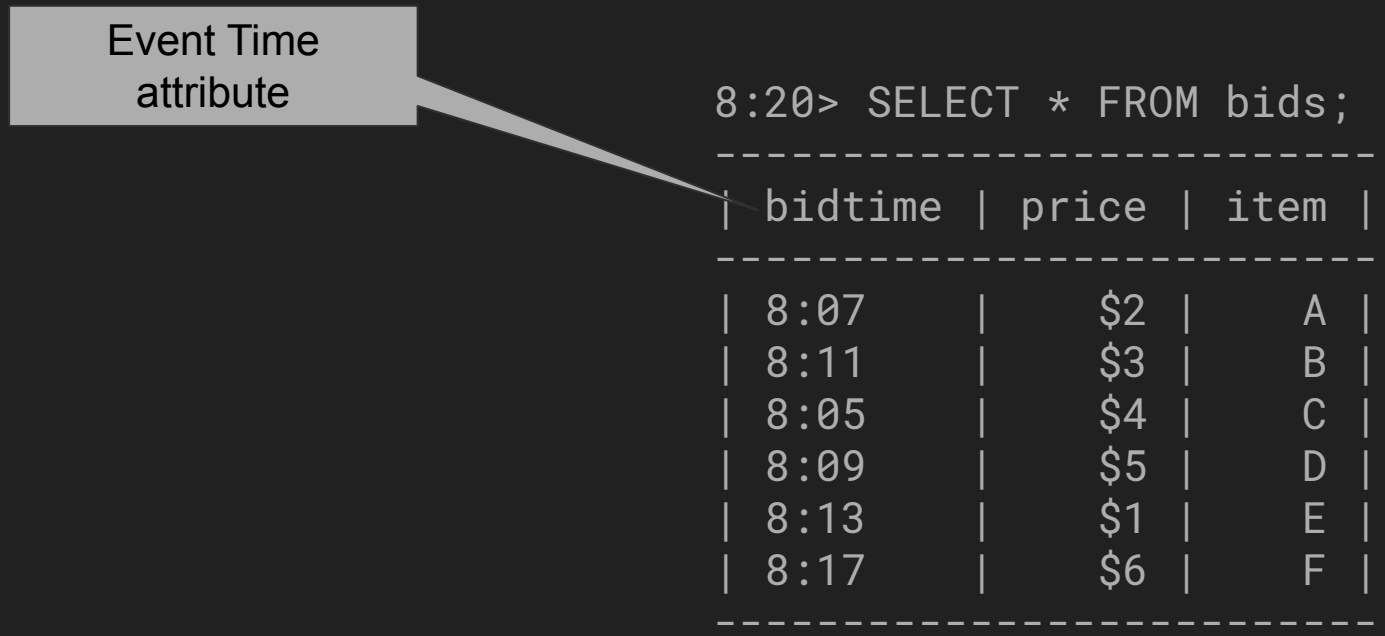
SELECT … GROUP BY event_timestamp



SELECT … GROUP BY event_timestamp



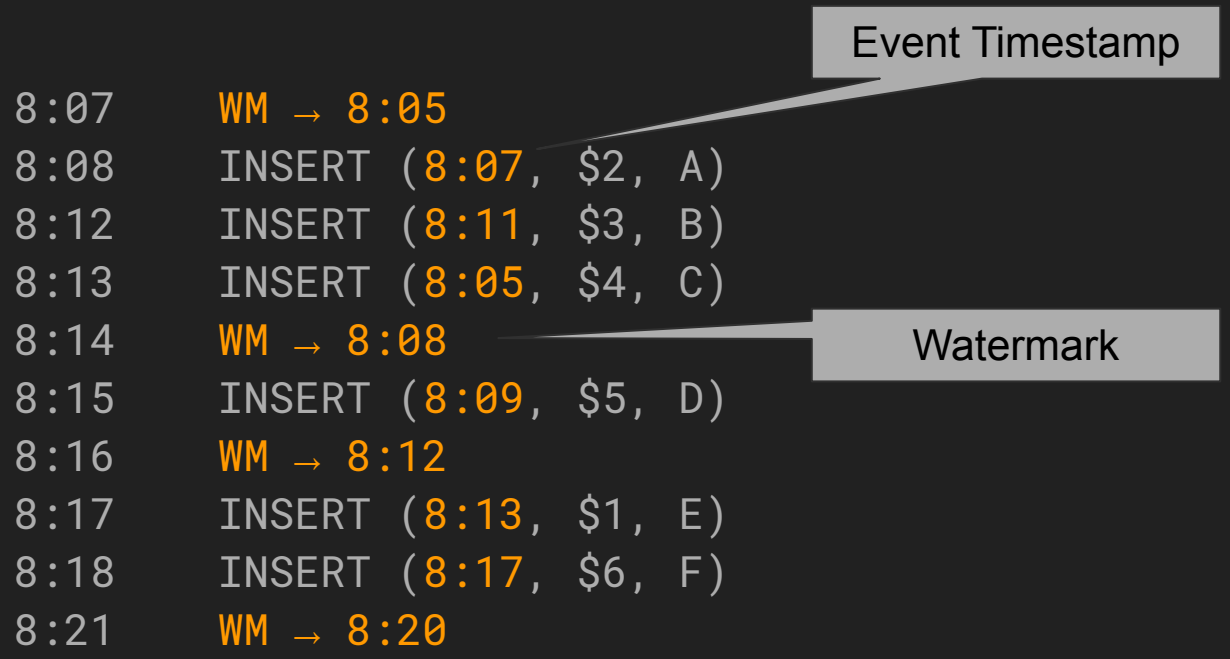
SQL Extension 1: Event Time Attributes



SQL Extension 1: Event Time Attributes

- Add DDL syntax to declare watermarked event time attributes
  - Similar to `PRIMARY KEY`, `UNIQUE`, `NOT NULL`, or other constraints
- Event time attribute is a regular `TIMESTAMP` type
  - Attribute can be used like any other `TIMESTAMP` attribute
  - Attribute is “roughly” increasing.
  - Watermarks report minimum of future values
- Optimizer uses knowledge of event time attributes to build plans that leverage watermarks to reason about progress

SQL Extension 1: Event Time Attributes



SQL Extension 2: Event Time Windowing Functions

- Add built-in table-valued functions to assign rows to event time windows
- TUMBLE function assigns each row to a fixed sized window
  - Enriches rows of a TVR with start and end timestamps

```
8:21> SELECT *
      FROM
        Tumble (
          data    => TABLE(bids),
          timecol => DESCRIPTOR(bidtime),
          dur     => INTERVAL '10 ' MINUTES);
```

wstart	wend	bidtime	price	item
8:00	8:10	8:07	$2	A
8:10	8:20	8:11	$3	B
8:00	8:10	8:05	$4	C
8:00	8:10	8:09	$5	D
8:10	8:20	8:13	$1	E
8:10	8:20	8:17	$6	F

SQL Extension 2: Event Time Windowing Functions

- Aggregate rows per event time window

```
8:21> SELECT MAX(wstart), wend, SUM(price)
      FROM
        Tumble (
          data    => TABLE(Bid),
          timecol => DESCRIPTOR(bidtime),
          dur     => INTERVAL '10 ' MINUTES)
      GROUP BY wend;
```

```
------------------------
| wstart | wend | price |
------------------------
| 8:00   | 8:10 |   $11 |
| 8:10   | 8:20 |   $10 |
------------------------
```

Controlling the realization of Time-Varying Results

Control *How* and *When* to Materialize a TVR

- A query on a TVR produces a result TVR
  - There are several options how to materialize a TVR
- Materialize a TVR as table or as stream?
  - Table realization is the default
  - Stream realization needs to be explicitly chosen
- Choose when or how often to materialize TVR changes
  - Only materialize complete results
  - Only materialize changes once per minute

SQL Extension 3: Stream realization

- Add `EMIT STREAM` clause for stream realization
- Materializes the changes of a TVR in a changelog TVR
  - All operations on TVR are supported

SQL Extension 3: Stream realization

```
8:08    INSERT (8:07, $2, A)
8:12    INSERT (8:11, $3, B)
8:13    INSERT (8:05, $4, C)
8:15    INSERT (8:09, $5, D)
8:17    INSERT (8:13, $1, E)
8:18    INSERT (8:17, $6, F)
```

```
8:20> SELECT ...;
-------------------------
| wstart | wend | price |
-------------------------
| 8:00   | 8:10 |   $11 |
| 8:10   | 8:20 |   $10 |
-------------------------
```

```
8:08> SELECT ... EMIT STREAM;
-----------------------------------------------
| wstart | wend | price | undo | ptime | ver |
-----------------------------------------------
| 8:00   | 8:10 |    $2 |      | 8:08  |   0 |
| 8:10   | 8:20 |    $3 |      | 8:12  |   0 |
| 8:00   | 8:10 |    $2 | undo | 8:13  |   1 |
| 8:00   | 8:10 |    $6 |      | 8:13  |   2 |
| 8:00   | 8:10 |    $6 | undo | 8:15  |   3 |
| 8:00   | 8:10 |   $11 |      | 8:15  |   4 |
| 8:10   | 8:20 |    $3 | undo | 8:17  |   1 |
| 8:10   | 8:20 |    $4 |      | 8:17  |   2 |
| 8:10   | 8:20 |    $4 | undo | 8:18  |   3 |
| 8:10   | 8:20 |   $10 |      | 8:18  |   4 |
...
```

SQL Extension 4: Delay for Completeness

- Add `EMIT AFTER WATERMARK` clause to materialize only complete results
  - Watermark indicates completeness

```
8:07    WM → 8:05
8:08    INSERT (8:07, $2, A)
8:12    INSERT (8:11, $3, B)
8:13    INSERT (8:05, $4, C)
8:14    WM → 8:08
8:15    INSERT (8:09, $5, D)
8:16    WM → 8:12
8:17    INSERT (8:13, $1, E)
8:18    INSERT (8:17, $6, F)
8:21    WM → 8:20
```

```
8:15> SELECT ... EMIT AFTER WATERMARK;
------------------------
| wstart | wend | price |
------------------------
------------------------

8:17> SELECT ... EMIT AFTER WATERMARK;
------------------------
| wstart | wend | price |
------------------------
| 8:00   | 8:10 |   $11 |
------------------------
```

SQL Extension 4: Delay for Completeness

- Add `EMIT AFTER WATERMARK` clause can be combined with `EMIT STREAM`

```
8:07    WM → 8:05
8:08    INSERT (8:07, $2, A)
8:12    INSERT (8:11, $3, B)
8:13    INSERT (8:05, $4, C)
8:14    WM → 8:08
8:15    INSERT (8:09, $5, D)
8:16    WM → 8:12
8:17    INSERT (8:13, $1, E)
8:18    INSERT (8:17, $6, F)
8:21    WM → 8:20
```

```
8:08> SELECT ... EMIT STREAM AFTER WATERMARK;
------------------------------------------------
| wstart | wend | price | undo | ptime | ver |
------------------------------------------------
| 8:00   | 8:10 |   $11 |      | 8:16  |   0 |
| 8:10   | 8:20 |   $10 |      | 8:21  |   0 |
...
```

SQL Extension 5: Periodic Delays

- Materializing every change of a TVR can result in many updates
  - Can overload downstream systems
  - Often not necessary / required

- Add `EMIT AFTER DELAY` clause to control frequency of updates

SQL Extension 5: Periodic Delays

```
8:08    INSERT (8:07, $2, A)
8:12    INSERT (8:11, $3, B)
8:13    INSERT (8:05, $4, C)
8:15    INSERT (8:09, $5, D)
8:17    INSERT (8:13, $1, E)
8:18    INSERT (8:17, $6, F)
```

```
8:08> SELECT ... EMIT STREAM
      AFTER DELAY INTERVAL '6' MINUTES;
-----------------------------------------------
| wstart | wend | price | undo | ptime | ver |
-----------------------------------------------
| 8:00   | 8:10 |    $6 |      | 8:14  | 0   |
| 8:10   | 8:20 |   $10 |      | 8:18  | 0   |
| 8:00   | 8:10 |    $6 | undo | 8:21  | 1   |
| 8:00   | 8:10 |   $11 |      | 8:21  | 2   |
...
```

```
8:15> SELECT ...;
-------------------------
| wstart | wend | price |
-------------------------
| 8:00   | 8:10 |    $6 |
-------------------------
```

```
8:19> SELECT ...;
-------------------------
| wstart | wend | price |
-------------------------
| 8:00   | 8:10 |    $6 |
| 8:10   | 8:20 |   $10 |
-------------------------
```

```
8:21> SELECT ...;
-------------------------
| wstart | wend | price |
-------------------------
| 8:00   | 8:10 |   $11 |
| 8:10   | 8:20 |   $10 |
-------------------------
```

Adoption

- Flink
  - Available via Table and SQL APIs
  - In use at companies such as Alibaba, Huawei, Lyft, Uber
- Beam
  - Available via Java SQL API, SQL CLI, and Google Cloud Dataflow UI.
  - In use by companies such as eBay, Spotify

Future Work

- Expanded / Custom Event-Time Windowing
  - Pre-built: transitive closure sessions, keyed sessions, calendar months, etc.
  - Custom windows defined via a SQL expression
- Time-progressing expressions:
  - E.g., (bidtime > CURRENT_TIME - INTERVAL ’1’ HOUR)
  - Expressions like CURRENT_TIME are fixed to query evaluation time in current standard
- Correlated access to temporal tables
  - Need dynamic AS OF SYSTEM TIME expressions for time-varying correlations
- Streaming changelog options
  - E.g, render changelog as sequence of deltas rather than updates.

Future Work

- Nested EMIT
  - EMIT within nested queries could add additional power, at cost of increased complexity
- Graceful evolution
  - Need clean ways to evolve stateful streaming pipelines over time
- More rigorous formal semantics
  - What are the formal properties of streaming systems in general?
  - Watermarks, latency, realization, etc.
  - Greater understanding of differences between different systems.

Summary

- Guiding principles
  - Unified semantics for SQL over tables and streams
  - Minimal additions to the standard
  - Use as much as possible of SQL in a streaming context

- Our proposal
  1. Time-varying relations
  2. Event time semantics
     - Ext 1: Event time attributes ← DDL for watermarks, etc.
     - Ext 2: Event time windowing functions ← TUMBLE, HOP, etc. via table-valued functions
  3. Controlling the realization of time-varying results
     - Ext 3: Stream realization ← EMIT STREAM
     - Ext 4: Watermark delays for completeness ← EMIT AFTER WATERMARK
     - Ext 5: Periodic delays for rate limiting ← EMIT AFTER DELAY

Thank you!
Questions?

https://s.apache.org/streaming-sql-apachecon-na-2019

https://arxiv.org/pdf/1905.12133.pdf