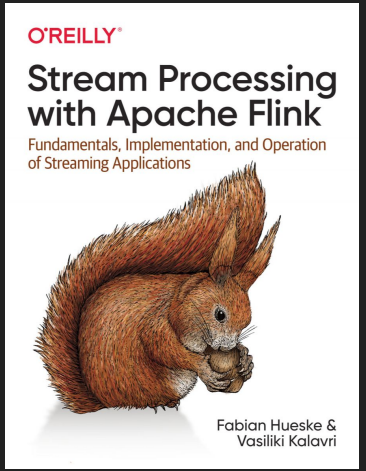
Stream Processing for Everyone with Continuous SQL Queries

Fabian Hueske, Ververica

YOW Data Online 2020

About Me

- Apache Flink Committer and PMC Member
  - Working on Flink since day one
- Co-Author of *“Stream Processing with Apache Flink”*
- Co-Founder of data Artisans
  - acquired by Alibaba in 2019, now Ververica



Background

- Flink and Beam communities are working on SQL support since late 2015
  - 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

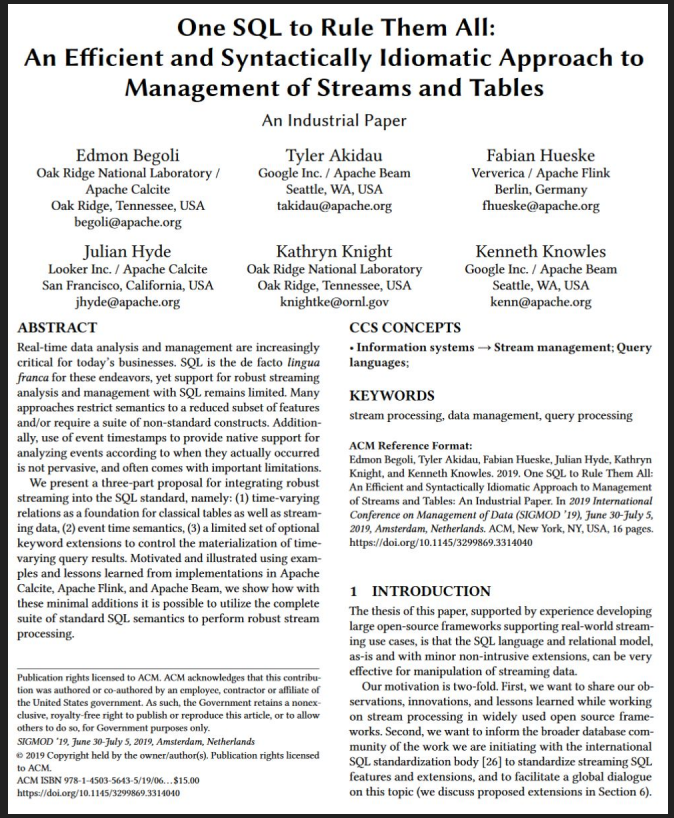
“One SQL to Rule Them All”

“One SQL to Rule Them All:
An Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables”

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

Accepted at SIGMOD 2019 Industry Track

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



One SQL to Rule Them All:
An Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables

An Industrial Paper

Edmon Begoli
Oak Ridge National Laboratory / Apache Calcite
Oak Ridge, Tennessee, USA
begoli@apache.org

Tyler Akidau
Google Inc. / Apache Beam
Seattle, WA, USA
takidau@apache.org

Fabian Hueske
Ververica / Apache Flink
Berlin, Germany
fhueske@apache.org

Julian Hyde
Looker Inc. / Apache Calcite
San Francisco, California, USA
jhyde@apache.org

Kathryn Knight
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
knightke@ornl.gov

Kenneth Knowles
Google Inc. / Apache Beam
Seattle, WA, USA
kenn@apache.org

ABSTRACT

Real-time data analysis and management are increasingly critical for today's businesses. SQL is the de facto *lingua franca* for these endeavors, yet support for robust streaming analysis and management with SQL remains limited. Many approaches restrict semantics to a reduced subset of features and/or require a suite of non-standard constructs. Additionally, use of event timestamps to provide native support for analyzing events according to when they actually occurred is not pervasive, and often comes with important limitations.

We present a three-part proposal for integrating robust streaming into the SQL standard, namely: (1) time-varying relations as a foundation for classical tables as well as streaming data, (2) event time semantics, (3) a limited set of optional keyword extensions to control the materialization of time-varying query results. Motivated and illustrated using examples and lessons learned from implementations in Apache Calcite, Apache Flink, and Apache Beam, we show how with these minimal additions it is possible to utilize the complete suite of standard SQL semantics to perform robust stream processing.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3314040

CCS CONCEPTS

• **Information systems** → **Stream management**; **Query languages**;

KEYWORDS

stream processing, data management, query processing

ACM Reference Format:
Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All: An Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables: An Industrial Paper. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3299869.3314040

1 INTRODUCTION

The thesis of this paper, supported by experience developing large open-source frameworks supporting real-world streaming use cases, is that the SQL language and relational model, as-is and with minor non-intrusive extensions, can be very effective for manipulation of streaming data.

Our motivation is two-fold. First, we want to share our observations, innovations, and lessons learned while working on stream processing in widely used open source frameworks. Second, we want to inform the broader database community of the work we are initiating with the international SQL standardization body [26] to standardize streaming SQL features and extensions, and to facilitate a global dialogue on this topic (we discuss proposed extensions in Section 6).

Our Motivation

- Share our experience of building OS streaming systems with SQL support
  - Our software powers many real-world streaming use cases
- The SQL Standards Committee is investigating extensions for streaming SQL
  - One of our co-authors is a member of the SQL standards committee
  - We want to offer our experience and help to shape the standard

What we came up with

- 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 is threefold
  1. Time-varying relations
  2. Event time semantics
  3. Controlling the materialization of time-varying results

Time-Varying Relations

Regular and Streaming SQL

- Regular SQL queries process point-in-time relations
  - Transactions & isolation levels ensure consistency of relations
- Time is the new dimension of streaming SQL
  - Streaming SQL queries process relations that evolve over time

Time-Varying Relations (TVRs)

- A time-varying relation (TVR) is a regular relation that changes over time
  - A table that is updated by transactional applications
  - A stream that is interpreted as the changelog of a table
- For each point in time, a TVR can return a static relation
  - The full set of SQL operations remains valid!
- A SQL query on a TVR is continuously evaluated and produces a result TVR
  - Result TVR can be computed in lock step with input TVRs
  - Equivalent to maintaining a materialized view

Key Insight

Streams and Tables are different representations 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)
```

Type of change

Time of change

Changed data

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 |
-------------------------
```

SQL Extension?

- SQL extensions are not needed for TVRs!
- All SQL operations remain valid on TVRs

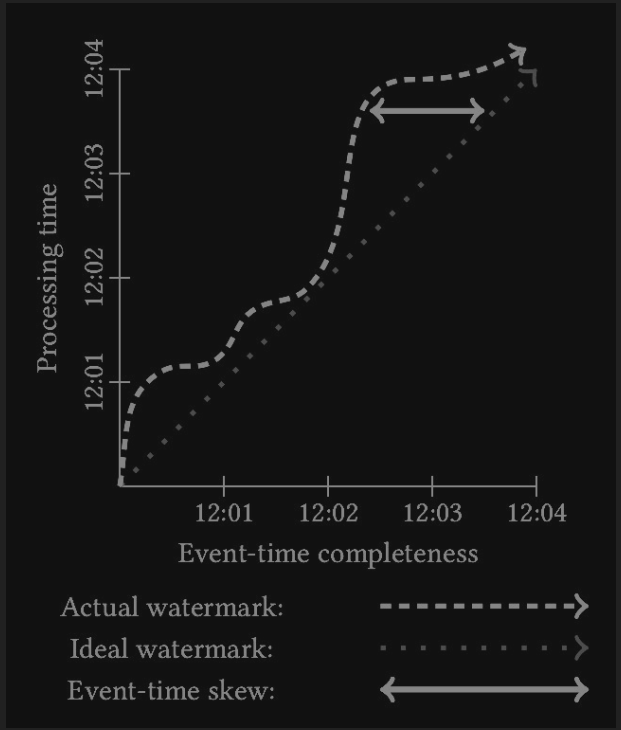
Event Time Semantics

Event Time vs. Processing Time

- Time-based operations are very common in stream processing
  - Count clicks per URL and hour
  - Join events that are at most 5 minutes apart from each other
- A system needs a notion of time to evaluate such queries
  - Using arrival time of events (a.k.a. processing time) results in arbitrary results
- Event time semantics are required for correct and consistent results
  - Using timestamps that are provided by/embedded in the data
  - Correct results when live data is delayed or out-of-order is processed
  - Correct results when recorded data is processed

Implementing Event Time Semantics

- Event timestamps and watermarks* to implement event time semantics
  - Event timestamps define point in time of an event
  - Watermarks define a temporal margin of completeness for a stream
- General approach to support full breadth of streaming use cases
  - Temporal aggregations
  - Notifications and alerts

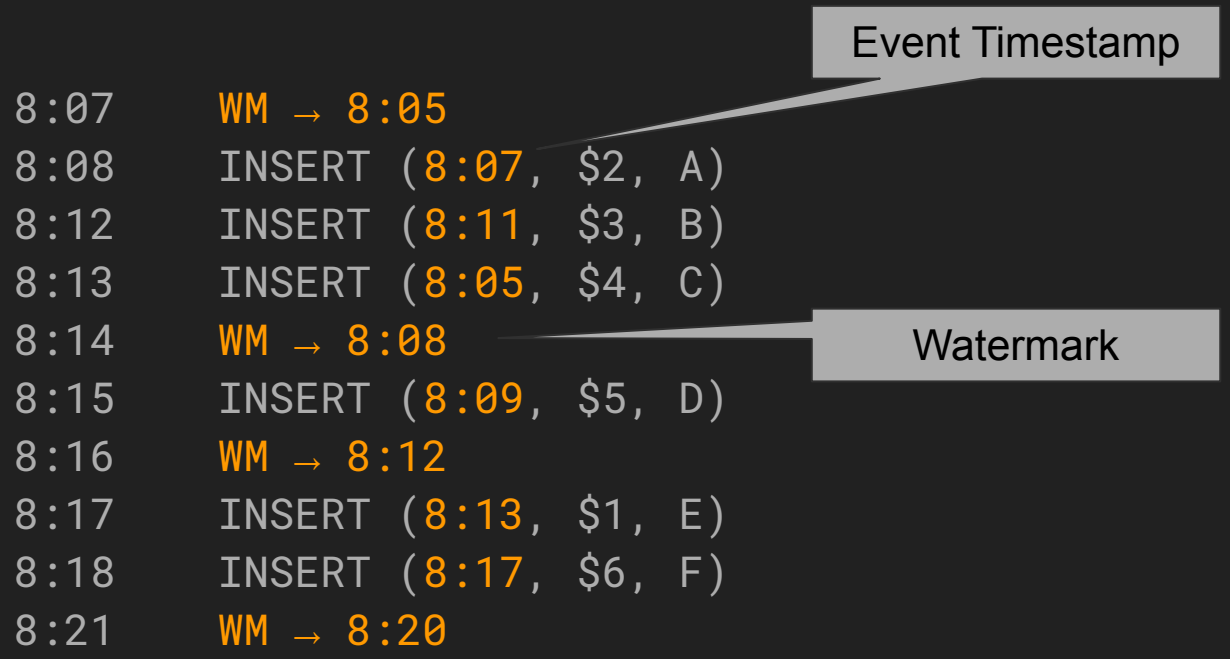


*Millwheel (VLDB'13) proposed watermarks. Later adopted by Cloud Dataflow, Beam, and Flink

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 Materialization 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 materialization is the default
  - Stream materialization 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 Materialization

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

SQL Extension 3: Stream Materialization

```
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

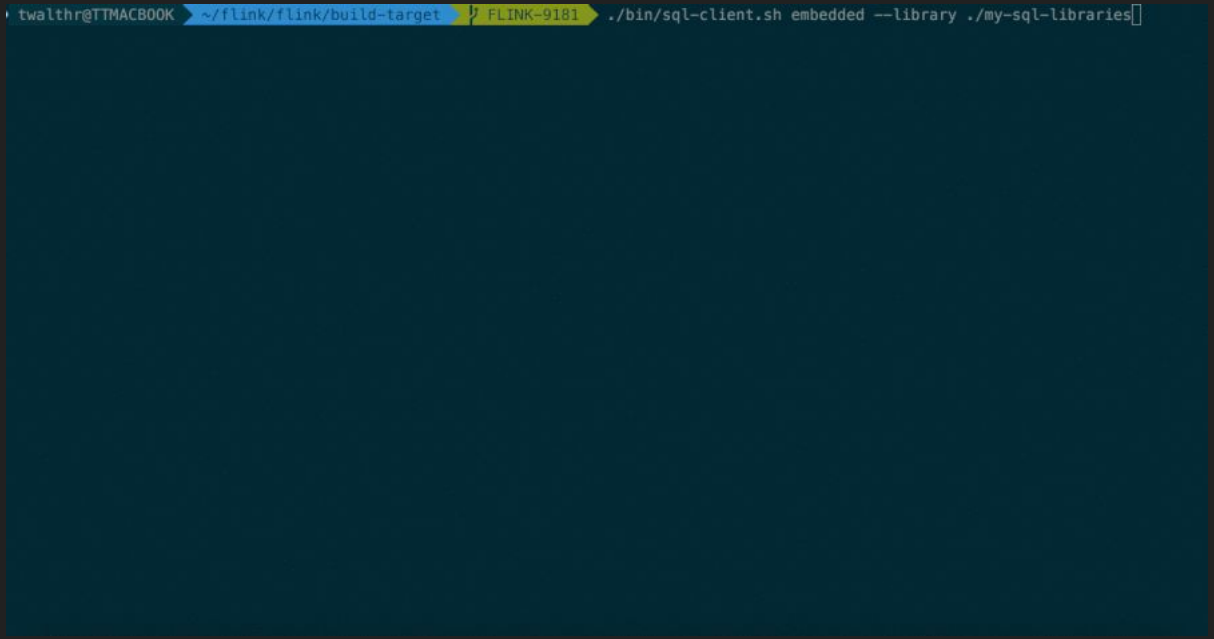
- Materializing every change of a TVR can result in many updates
  - Can overload downstream systems
  - Often not necessary / required

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

Adoption of Streaming SQL

- Calcite
  - Community is implementing proposed syntax extensions https://issues.apache.org/jira/browse/CALCITE-3271
- Flink
  - Available via Table and SQL APIs
  - In use at several companies, including Alibaba, Cloudera, Huawei, Lyft, Uber, Yelp
- Beam
  - Available via Java SQL API, SQL CLI, and Google Cloud Dataflow UI.
  - In use by companies such as eBay, Spotify

Apache Flink Streaming SQL Demo



Try it for yourself at: https://github.com/ververica/sql-training/

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, materialization, 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 materialization of time-varying results
     - Ext 3: Stream materialization ← EMIT STREAM
     - Ext 4: Watermark delays for completeness ← EMIT AFTER WATERMARK
     - Ext 5: Periodic delays for rate limiting ← EMIT AFTER DELAY

Thank you!
Questions?

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