Timelines

Declarative, temporal queries with Kaskada

July 2023

Agenda

Challenges of temporal queries

1. **Challenges of temporal queries**
2. The timeline abstraction
3. Querying with timelines
4. Executing efficiently
5. Next steps

Events are ubiquitous

Events

- System Logs
- IoT Data
- Derived Events
- API Call Logs
- DB Change Capture

- ML Training & Features
- Real-Time Analytics
- Real-Time Dashboards
- Historic Analytics
- Predictive Analytics

Insights

Example events from a game

Victories

Time	User
6	Aaron
2	Aaron
10	Aaron
18	Aaron
1	Carla
9	Carla

Losses

Time	User
5	Carla
12	Carla
12	Aaron
15	Aaron
3	Brad
16	Carla
19	Carla
4	Carla
6	Brad
4	Brad
1	Aaron
3	Carla
2	Brad
5	Brad
16	Aaron
17	Aaron
15	Carla
11	Carla
17	Carla
18	Carla

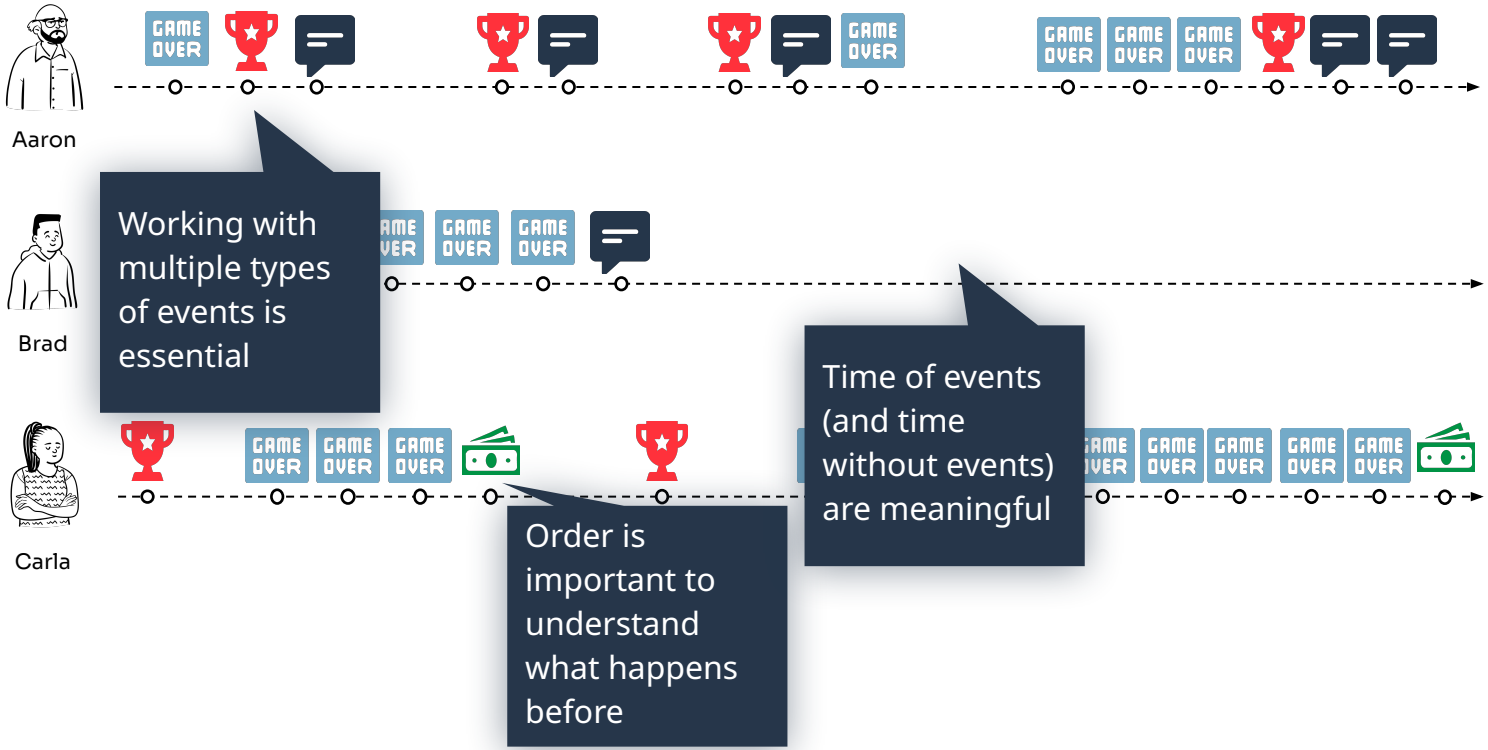
Messages

Time	User
20	Aaron
11	Aaron
7	Brad
3	Aaron
19	Aaron
7	Aaron

Purchases

Time	User
20	Carla
6	Carla

Complete User Stories

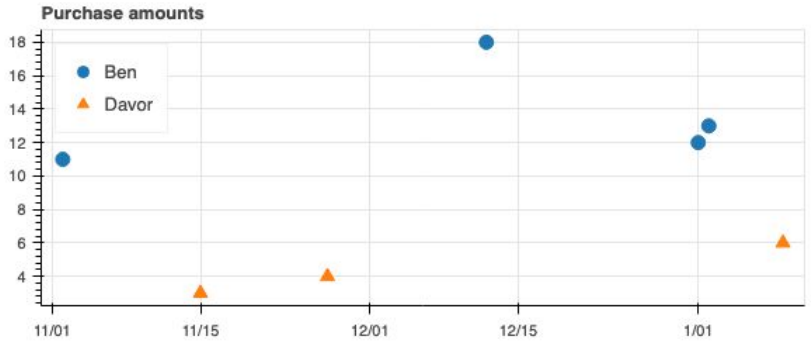


Agenda

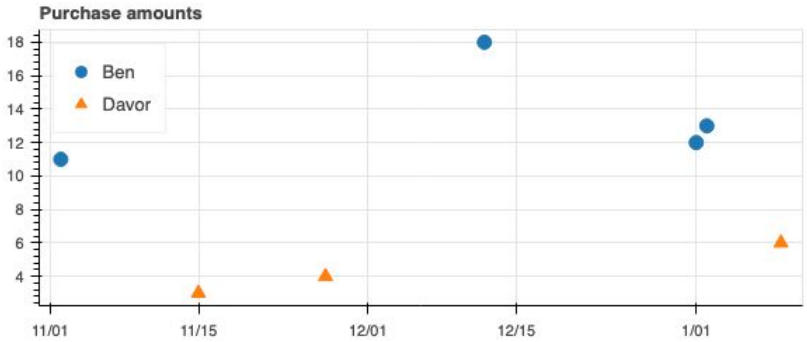
The timeline abstraction

1. Challenges of temporal queries
2. **The timeline abstraction**
3. Querying with timelines
4. Executing efficiently
5. Next steps

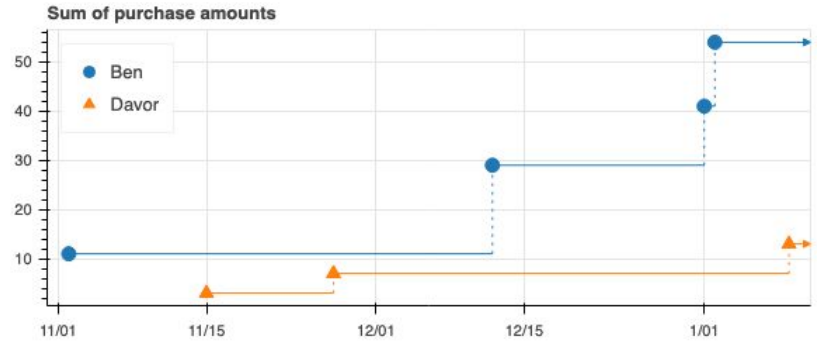
Timelines: Ordered by Time, Grouped by Entity



Timelines: Discrete or Continuous



Discrete



Continuous

Timelines all the way down

Two Views of a Timeline

Snapshot (at 12/20)

{ user: “Ben”, time: “12/20”, value: 11 }

{ user: “Davor”, time: “12/20”, value: 3 }

{ user: “Ben”, time: “11/02”, value: 11 }

{ user: “Davor”, time: “11/15”, value: 3 }

{ user: “Davor”, time: “11/15”, value: 7 }

{ user: “Ben”, time: “12/12”, value: 29 }

{ user: “Ben”, time: “01/01”, value: 41 }

{ user: “Davor”, time: “11/15”, value: 13 }

History

Timelines are the abstraction for temporal values

Agenda

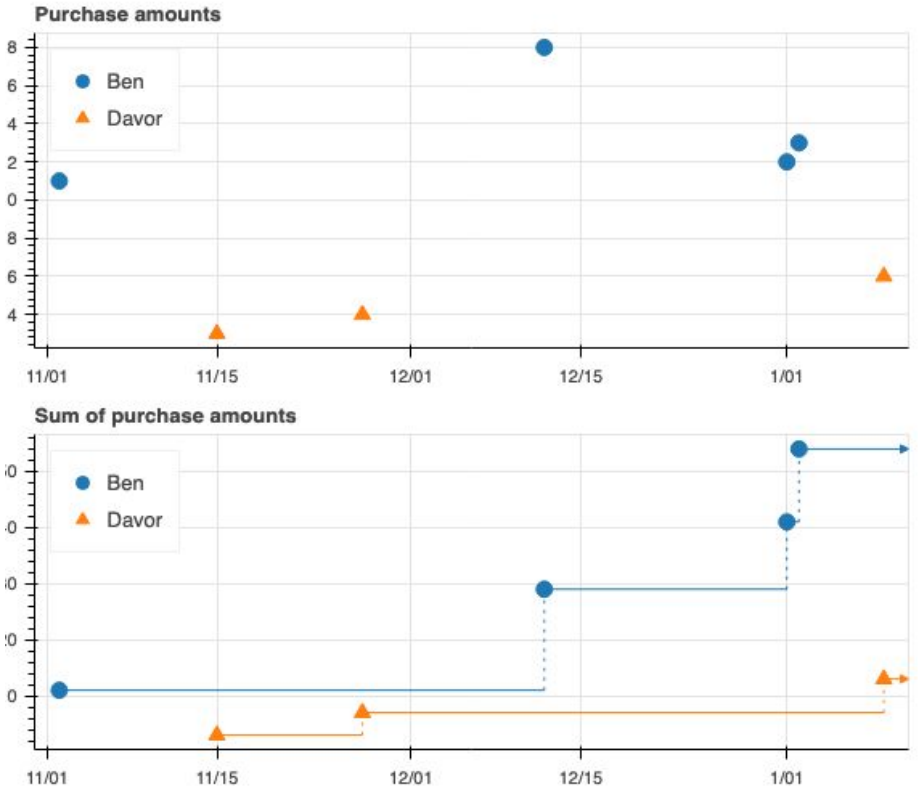
Querying with timelines

1. Challenges in real-time AI/ML
2. The timeline abstraction
3. **Querying with timelines**
4. Executing efficiently
5. Next steps

Aggregation

How much did each user spend?

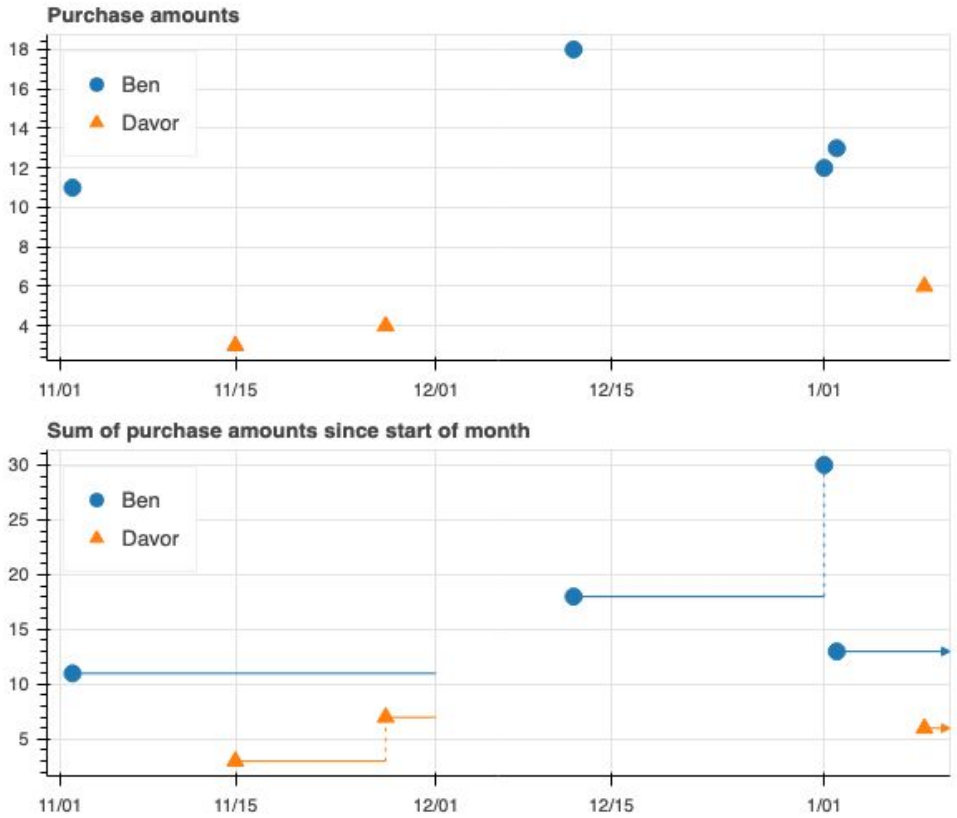
```
sum(Purchases.amount)

# OR

Purchases.amount
| sum()
```



Windowed Aggregation

How much has each user spent this month?

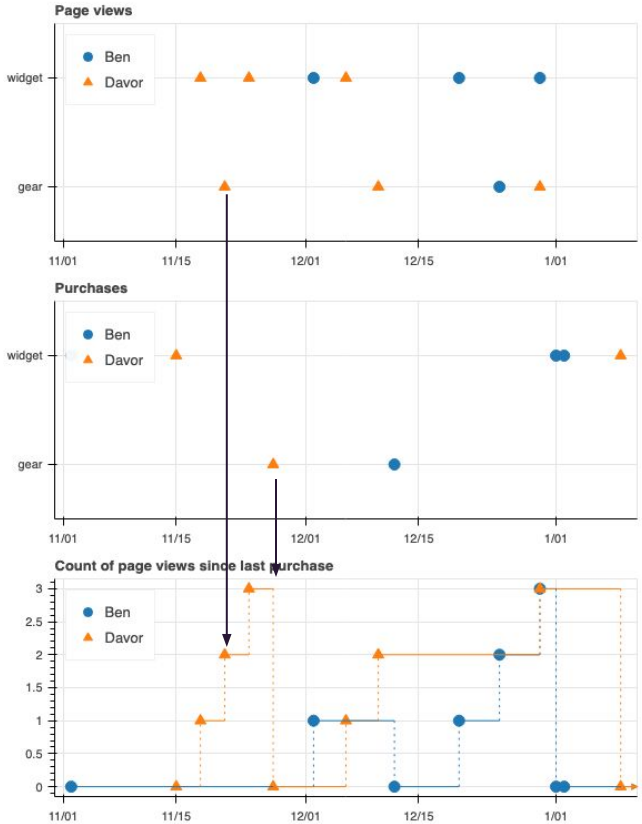
```
Purchases.amount
| sum(window=since(monthly()))
```



Data-Defined Windowed Aggregation

For each user, how many page-views have occurred since the last purchase?

```
PageViews
| count(window=since(Purchases))
```



Aside: Data-Defined Windows in SQL

For each user, how many page-views have occurred since the last purchase?

```
PageViews
| count(window=since(Purchases))
```

(Kaskada)

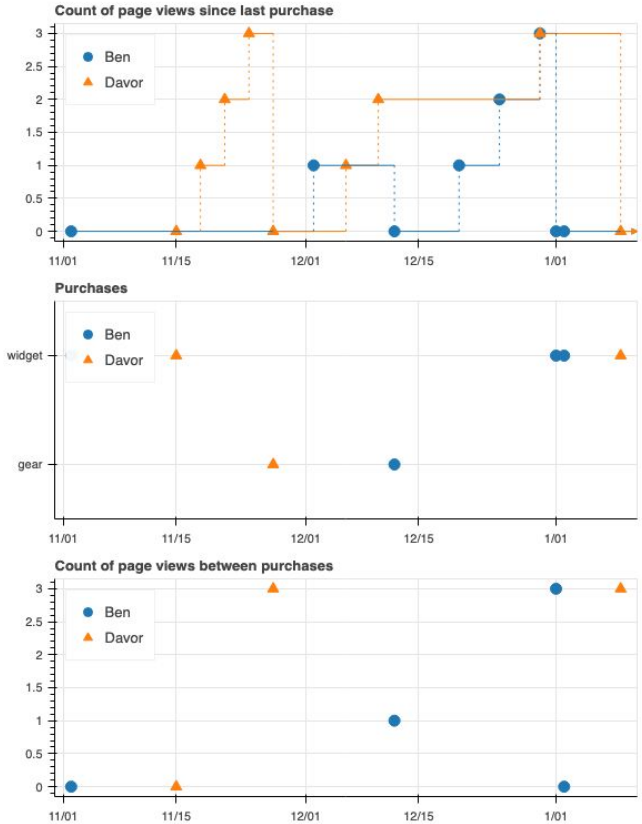
```
WITH activity AS (
  (SELECT k, t, 1 as is_page_view FROM page_views)
  UNION
  (SELECT k, t, 0 as is_page_view FROM purchases)
), purchase_counts AS (
  SELECT
    k, t, is_page_view,
    SUM(CASE WHEN is_page_view = 0 THEN 1 ELSE 0 END)
      OVER (PARTITION BY k ORDER BY t) AS purchase_count
    FROM activity
)
SELECT
  k, t,
  SUM(CASE WHEN is_page_view = 1 THEN 1 ELSE 0 END)
    OVER (PARTITION BY k, purchase_count ORDER BY t) AS views
  FROM purchase_counts
)
```

(SQL - History)

Data-Defined Windowed Aggregation

For each user, what is the number of page-views between each purchase?

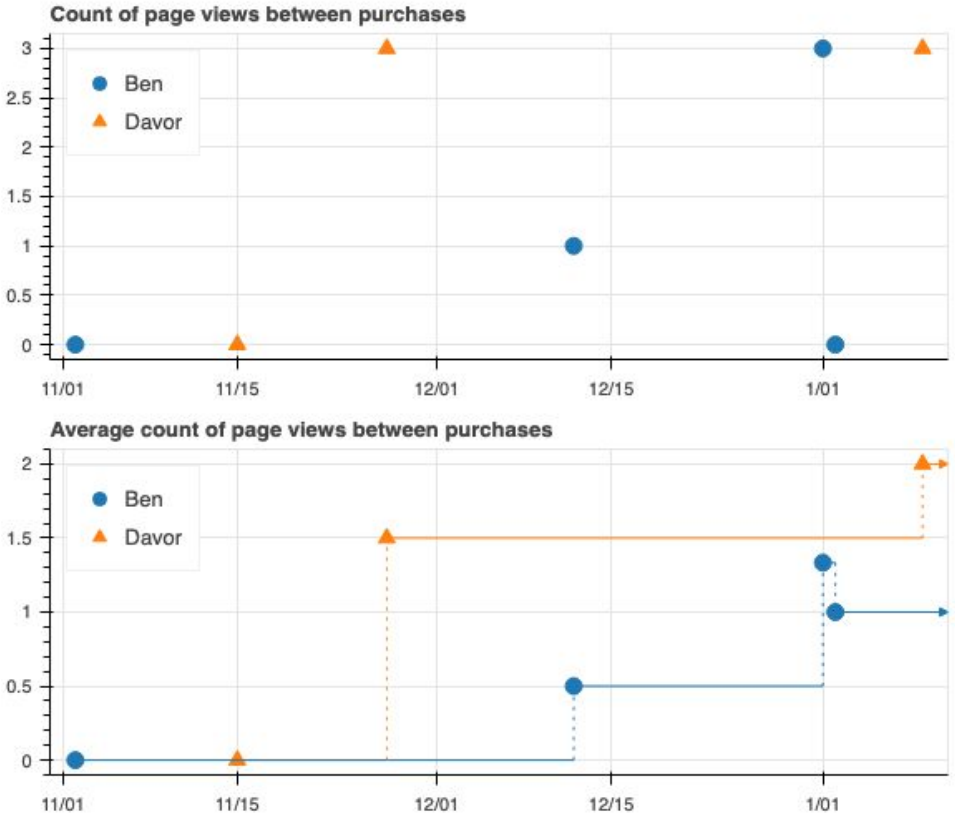
```
PageViews
| count(window=since(Purchases))
| when(is_valid(Purchases))
```



Data-Defined Windowed Aggregation

For each user, what is the average number of page-views between each purchase?

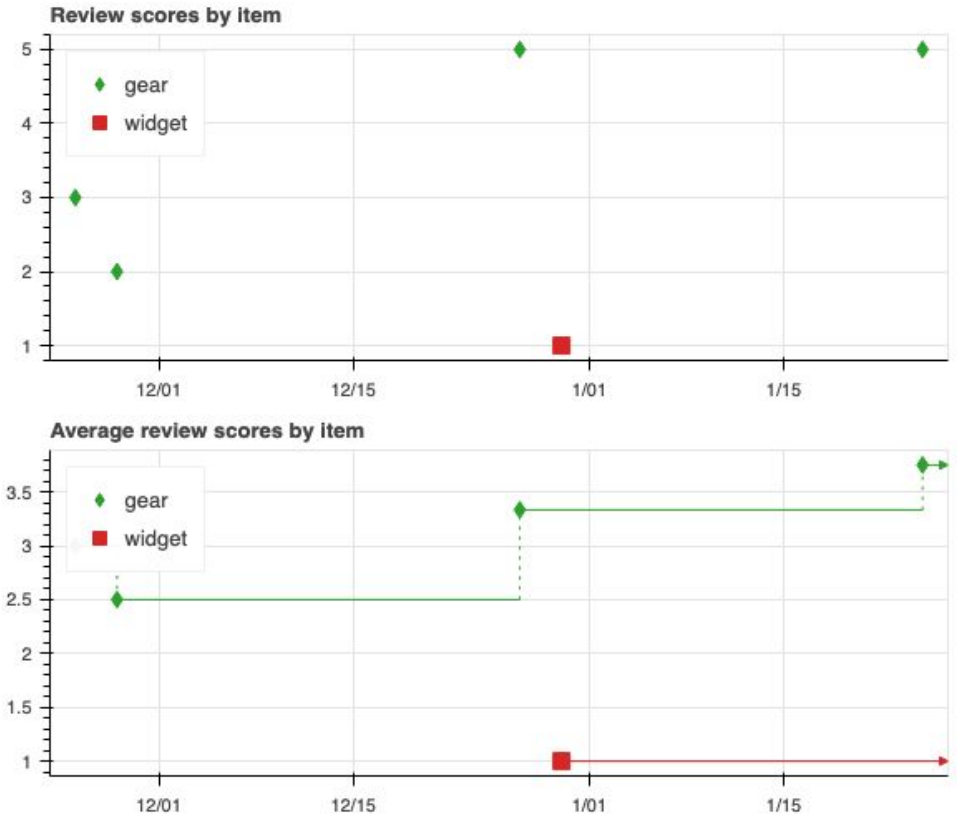
```
PageViews
| count(window=since(Purchases))
| when(is_valid(Purchases))
| mean()
```



Temporal Join

What is the average product review (score) at time of purchase?

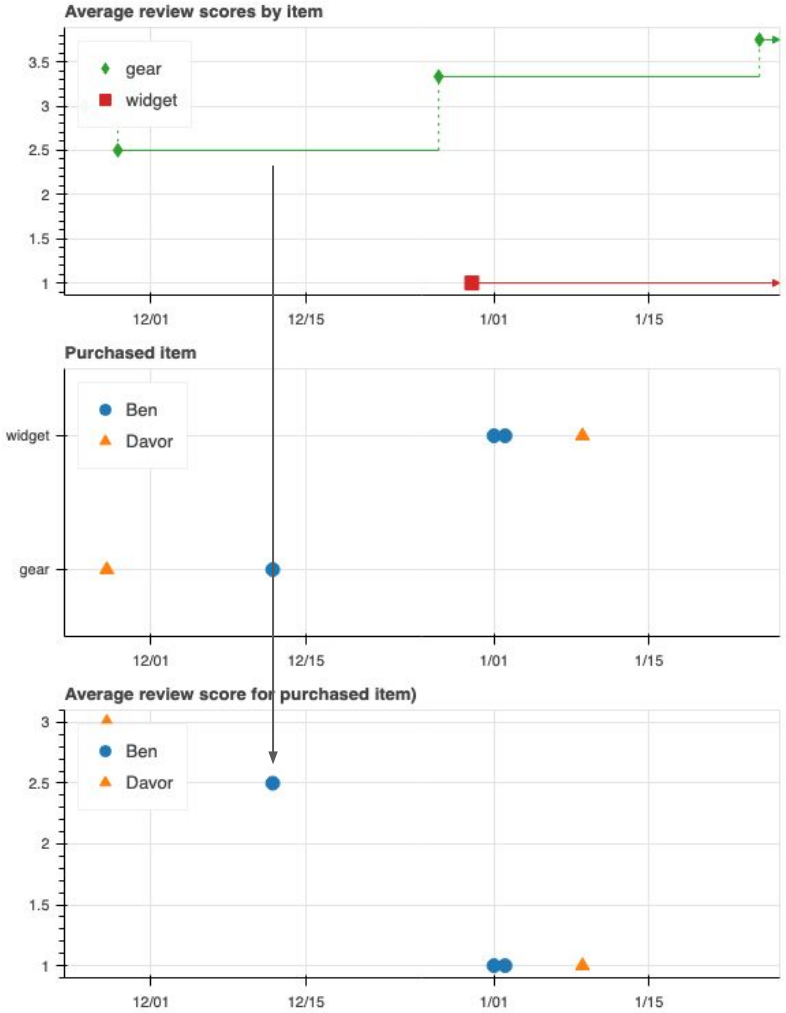
```
Reviews.score
| with_key(Review.item)
| mean()
```



Temporal Join

What is the average product review (score) at time of purchase?

```
Reviews.score
| with_key(Review.item)
| mean()
| lookup(Purchase.item)
```



Temporal Join

What is the average product review (score) at time of purchase?

```
Reviews.score
| with_key(Review.item)
| mean()
| lookup(Purchase.item)
```

(Kaskada)

```
WITH review_avg AS (
  SELECT product_id, t as time,
    AVG(score) OVER (PARTITION BY product_id ORDER BY t) as avg_score
  FROM Reviews
), review_times AS (
  SELECT product_id, time, time AS r_time,
    CAST(NULL AS TIMESTAMP) as p_time
  FROM review_avg
), purchase_times AS (
  SELECT product_id, t AS time, t as p_time
    CAST(NULL AS TIMESTAMP) AS r_time,
  FROM Purchases
), all_times AS (
  (SELECT * FROM review_times) UNION (SELECT * FROM purchase_times)
), spline AS (
  SELECT product_id, time, max(r_time) OVER w AS last_r_time,
  FROM all_times
  WINDOW w AS (PARTITION BY product_id ORDER BY time)
)
SELECT k, t, avg_score
FROM Purchases
LEFT JOIN spline
  ON Purchases.t = spline.time AND Purchases.product_id = spline.product_id
LEFT JOIN review_avg
  ON spline.last_r_time = review_avg.time
 AND Purchases.product_id = review_avg.product_id
```

1

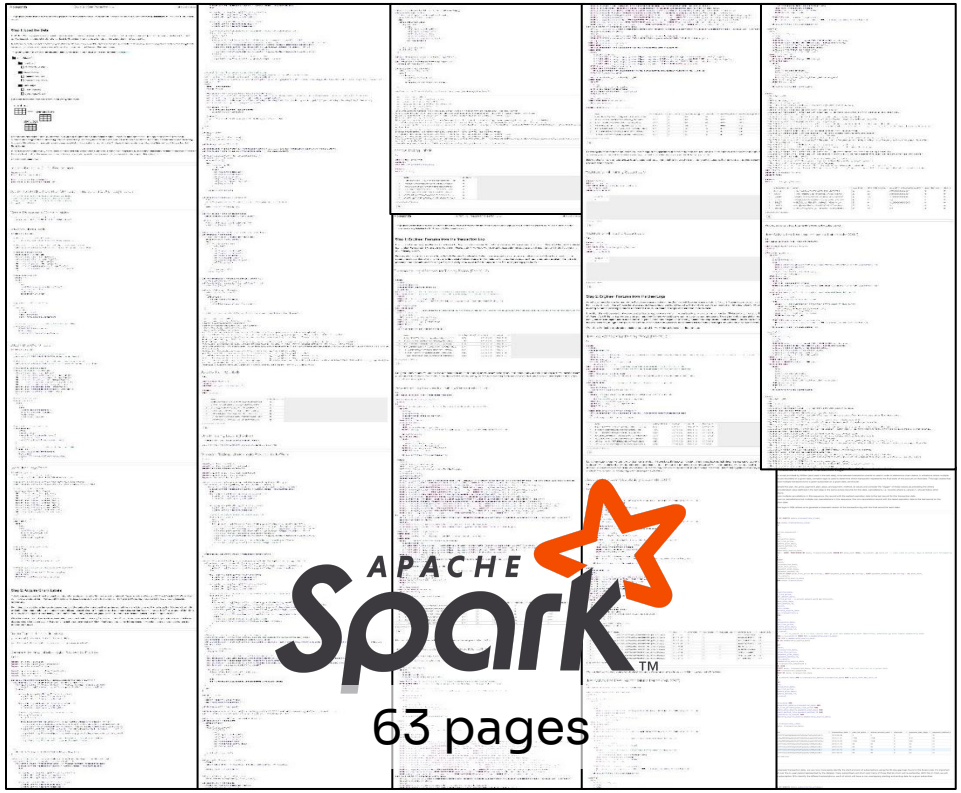
2

3

4

(SQL - Snapshot)

Abstractions Matter



APACHE Spark™

63 pages

KASKADA

2 pages

Change the abstraction, not the question

Agenda

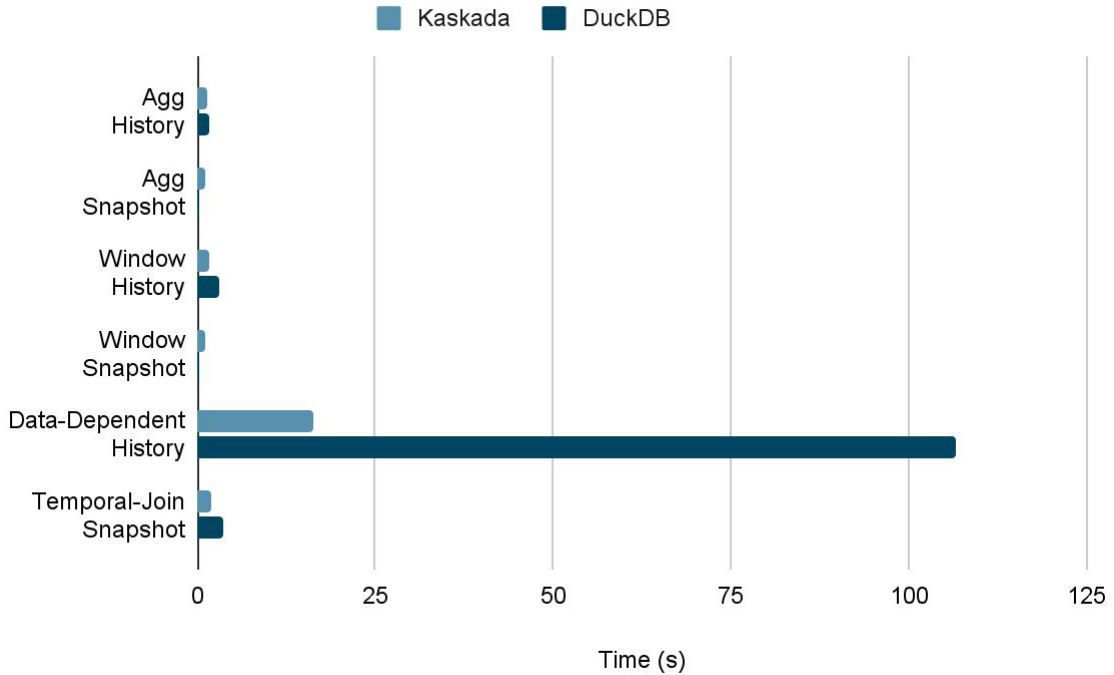
Executing Efficiently

1. Challenges of temporal queries
2. The timeline abstraction
3. Querying with timelines
4. **Executing efficiently**
5. Next steps

Benchmarks vs. DuckDB

Random Parquet Files

- 50K users
- 50K items
- 10M Purchases (784MB)
- 50M Page Views (3.7GB)
- 200K Reviews (15MB)



Efficient Execution

Incremental

Trade-off cost vs. latency by adjusting how often computations are executed.

Leveraging timelines

Take advantage of inherent ordering and keys to efficiently implement functions on timelines.

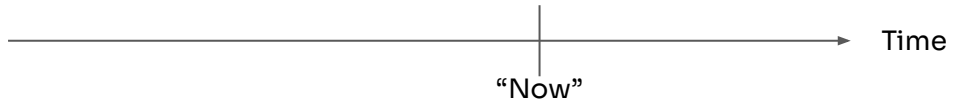
Columnar compute

Built in Rust, on Apache Arrow for fast columnar computations.

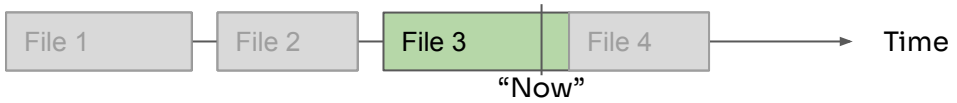
Temporal Locality: Taking Advantage of Order

Temporal queries tend to use values from the same time.

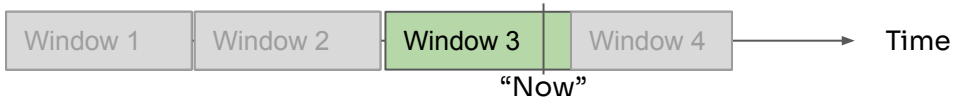
Many operations can take advantage of order.



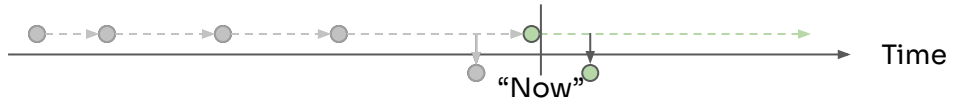
Reading Input Files



Windowed Aggregation

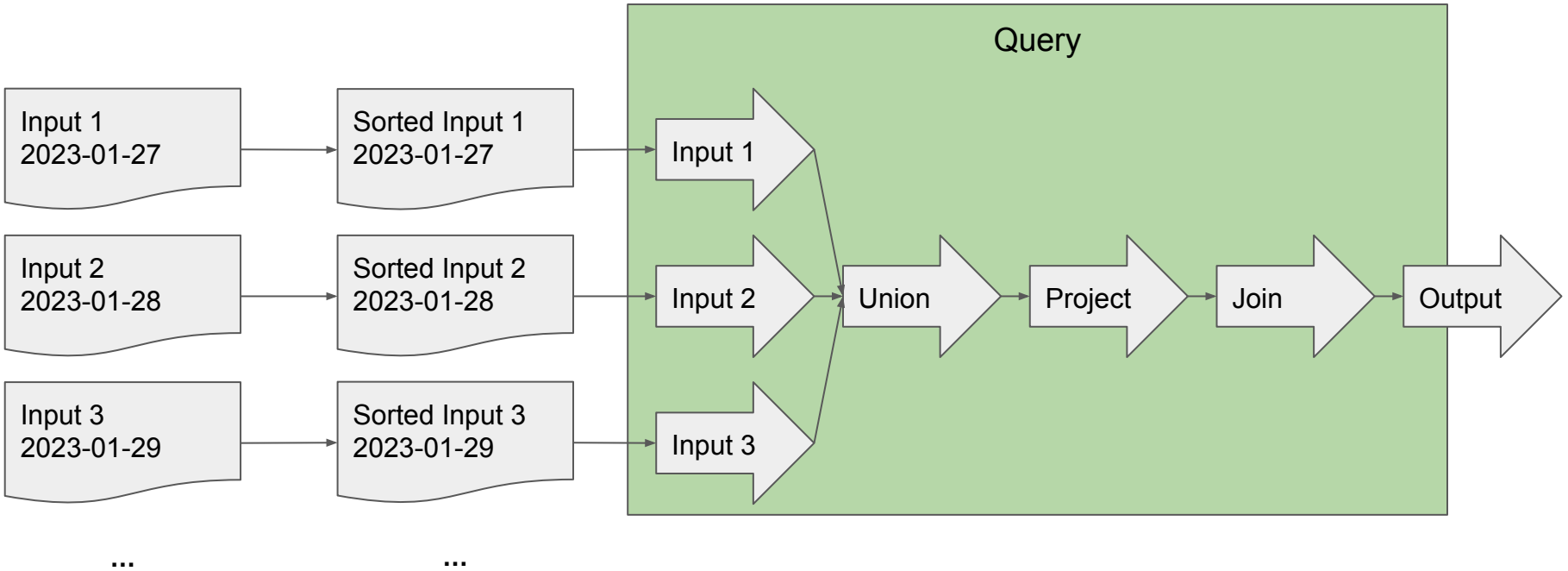


As-Of Join (Lookup)



Efficiently Providing Order

It is faster to *maintain* order than to *sort* when needed.



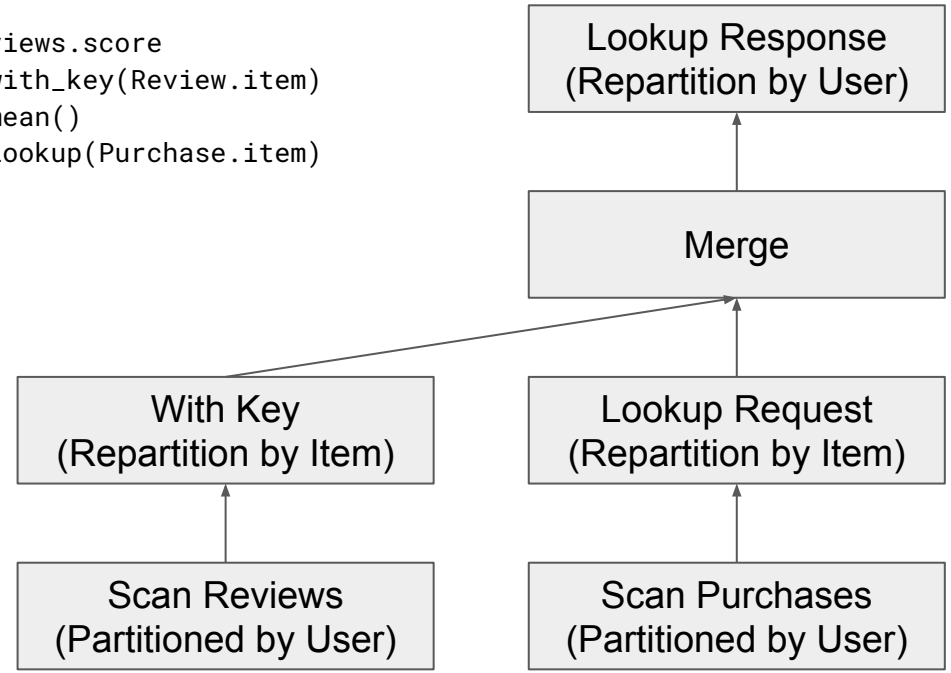
(Nearly) Embarrassingly Parallel

Partitioned by key suggests the problem is “embarassingly parallel”.

BUT the need for ordered operations between keys seems to violate this!

Similar to SQL joins, we can compile the temporal query to a plan that allows each operation to run in parallel with minimal synchronization happening between operations

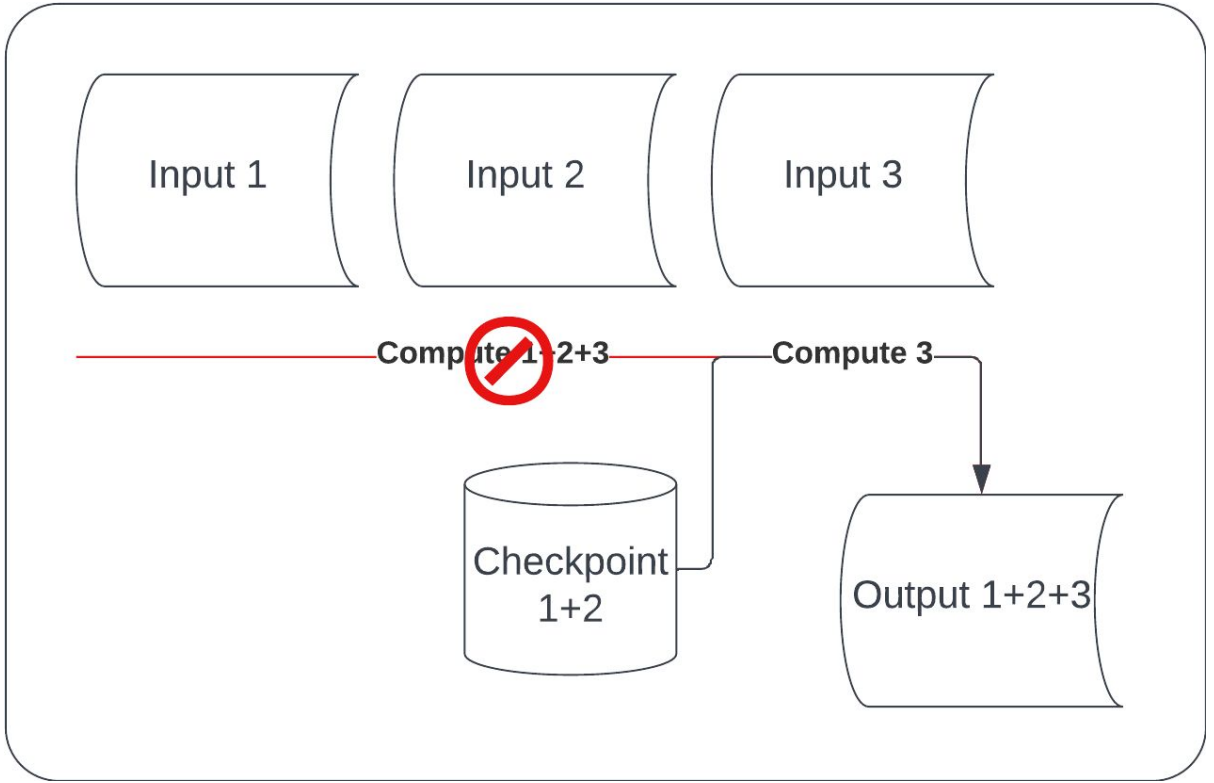
```
Reviews.score
| with_key(Review.item)
| mean()
| lookup(Purchase.item)
```



Checkpoints for Incremental

Checkpoints enable incremental batch, running as needed to update outputs

Instead of computing over all of the input, the most recent checkpoint can be used to run over only the “new” input



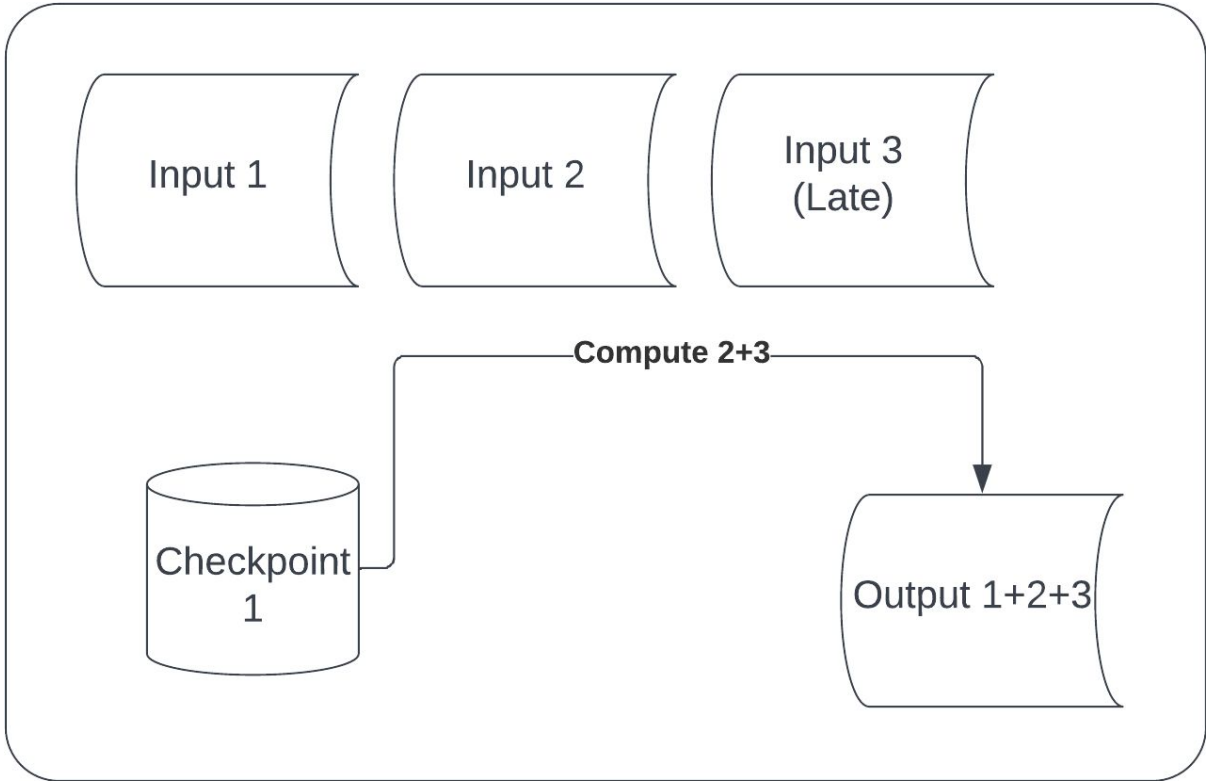
Optimistic Late Data

Checkpoints can be used to allow “optimistic” late data handling

As data arrives, it is processed immediately, assuming no late data will arrive

When late data ***does*** arrive, state can be restored from an earlier checkpoint and input including the late data processed in order

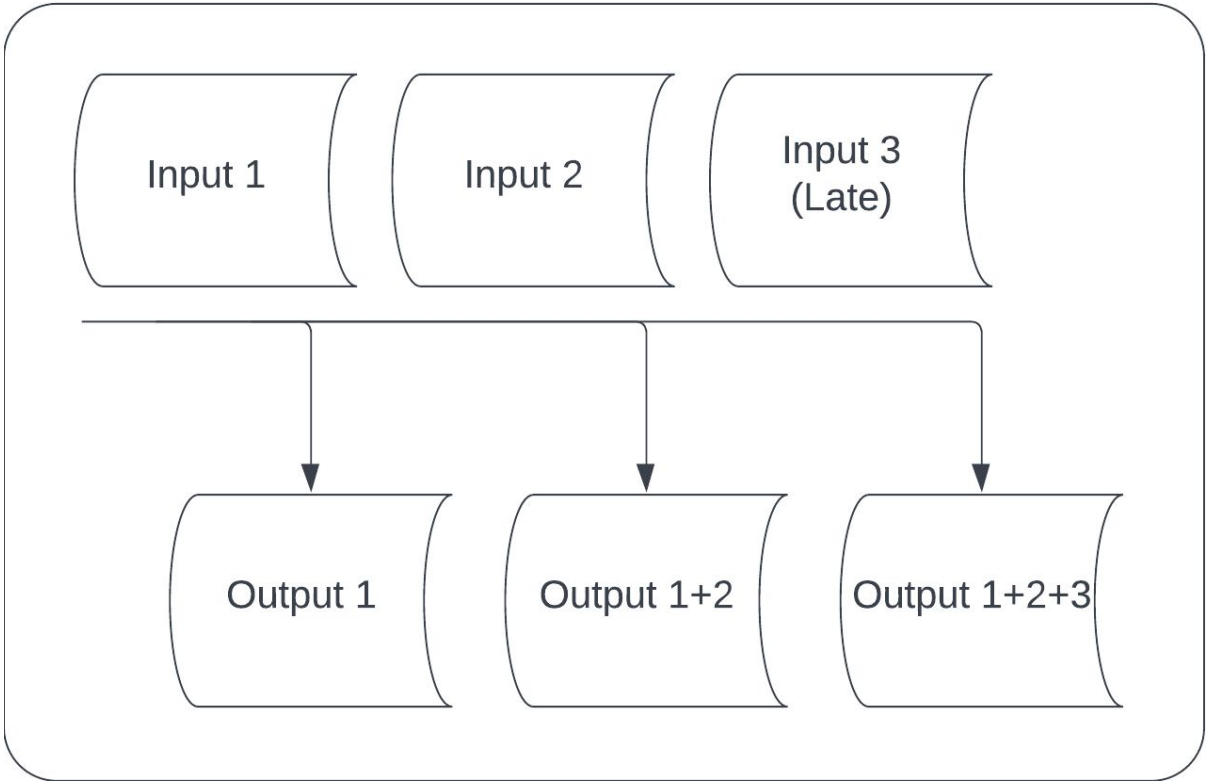
Roll-back can be per affected entity



Streaming

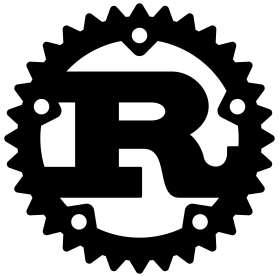
Streaming uses the same logic, but continues processing input as it arrives

The same checkpoints are produced by streaming and batch, allowing you to switch between modes of execution as needed



Open-Source Kaskada implements Timelines

- Timeline-based query language and execution engine
- Type-checked queries with useful, located error diagnostics
- High-performance engine built on Apache Arrow and Rust
- End-to-end incremental operations
- Apache-2 licensed
- Actively supported by DataStax



DATASTAX

Agenda

Next steps

1. Challenges of temporal queries
2. The timeline abstraction
3. Querying with timelines
4. Executing efficiently
5. **Next steps**

Next steps

Sign up for a hands-on workshop

https://kaskada.io/workshop

Try it out yourself

http://kaskada.io

Join the community

https://kaskada.io/community