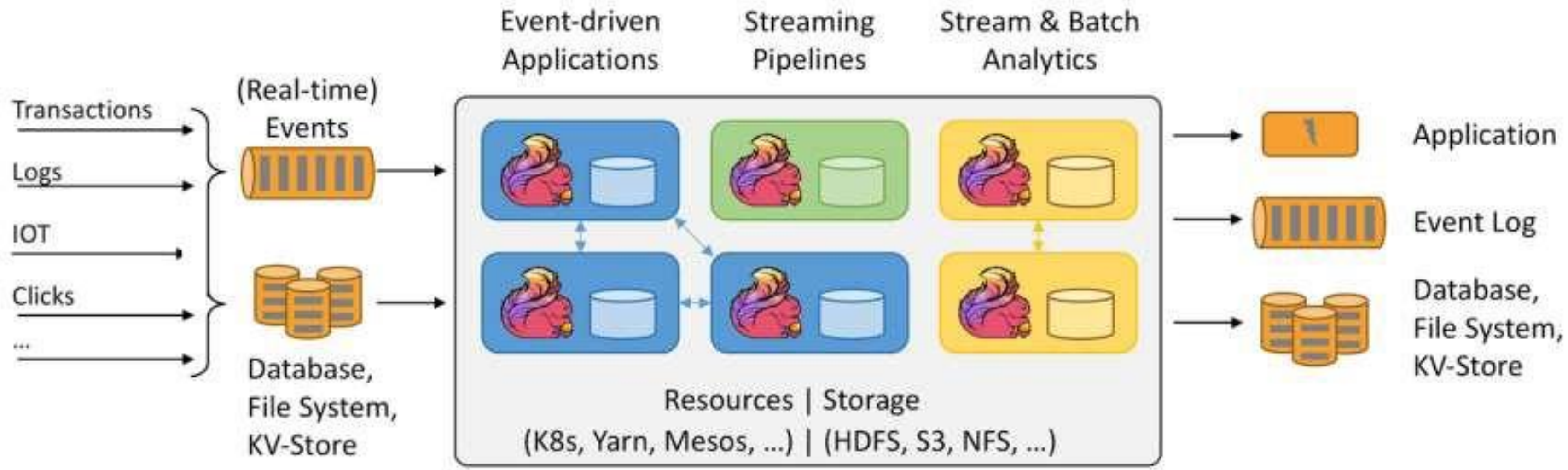
Apache Flink® SQL

Based on slides by Fabian Hueske and others

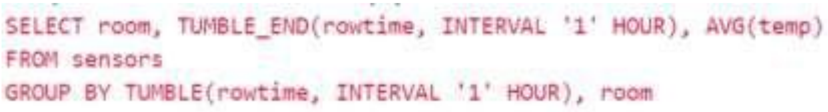
What is Apache Flink?

Stateful computations over streams
real-time and historic
fast, scalable, fault tolerant, in-memory
event time, large state, exactly-once

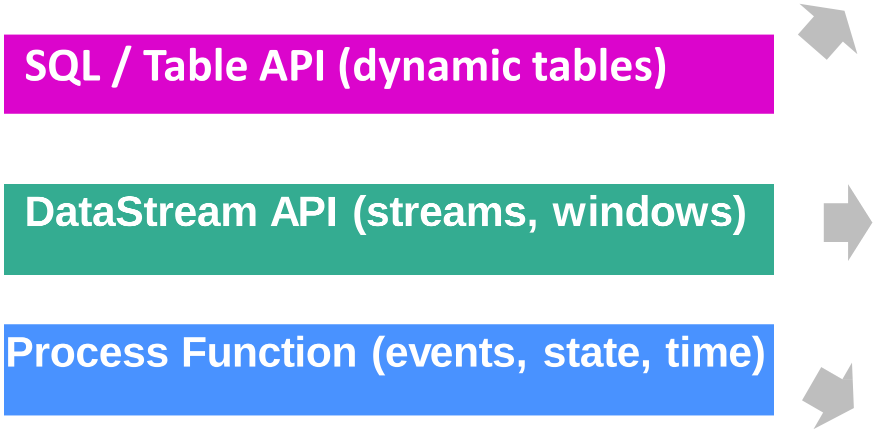

Event-driven Applications
Streaming Pipelines
Stream & Batch Analytics
Transactions
Logs
IOT
Clicks
...
(Real-time) Events
Database, File System, KV-Store
Resources | Storage
(K8s, Yarn, Mesos, ...) | (HDFS, S3, NFS, ...)
Application
Event Log
Database, File System, KV-Store

Flink’s Powerful Abstractions

Layered abstractions to
navigate simple to complex use cases



```
SELECT room, TUMBLE_END(rowtime, INTERVAL '1' HOUR), AVG(temp)
FROM sensors
GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR), room
```



High-level
Analytics API

SQL / Table API (dynamic tables)

Stream- & Batch
Data Processing

DataStream API (streams, windows)

Stateful Event-
Driven Applications

Process Function (events, state, time)

```
val stats = stream
    .keyBy("sensor")
    .timeWindow(Time.seconds(5))
    .sum((a, b) -> a.add(b))
```

```
def processElement(event: MyEvent, ctx: Context, out: Collector[Result]) = {
  // work with event and state
  (event, state.value) match { … }

  out.collect(…) // emit events
  state.update(…) // modify state

  // schedule a timer callback
  ctx.timerService.registerEventTimeTimer(event.timestamp + 500)
}
```

Flink’s Relational APIs

ANSI SQL

```
SELECT user, COUNT(url) AS cnt
FROM clicks
GROUP BY user
```

LINQ-style Table API

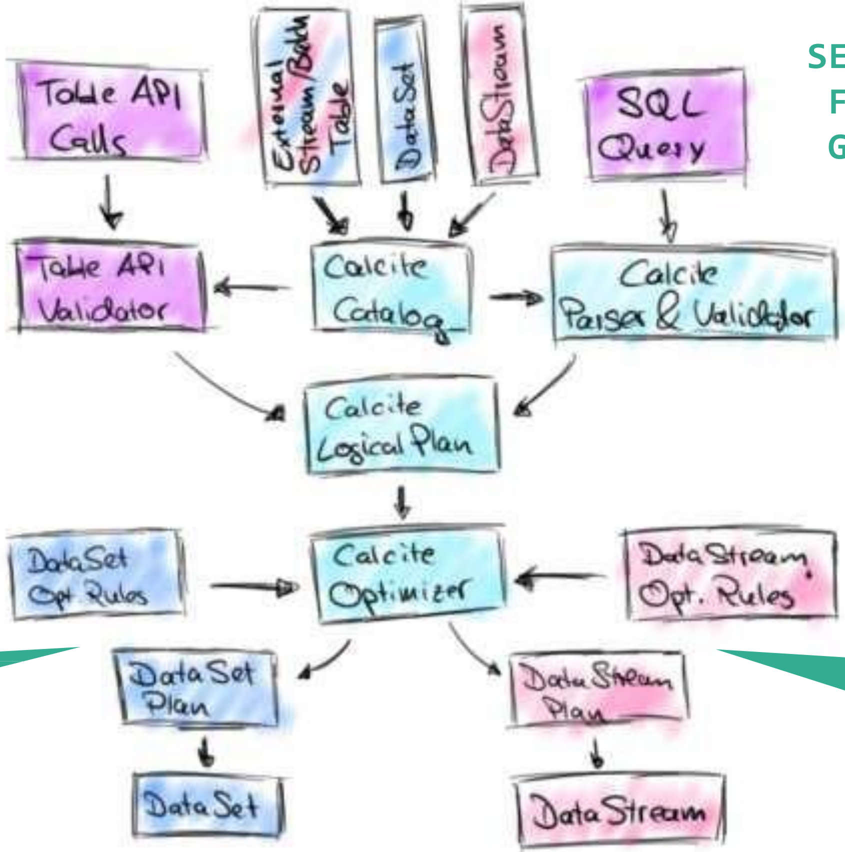
```
tableEnvironment
  .scan("clicks")
  .groupBy('user)
  .select('user, 'url.count as 'cnt)
```

Unified APIs for batch & streaming data

A query specifies exactly the same result regardless whether its input is static batch data or streaming data.

Query Translation

```
tableEnvironment
  .scan("clicks")
  .groupBy('user)
  .select('user, 'url.count)
```

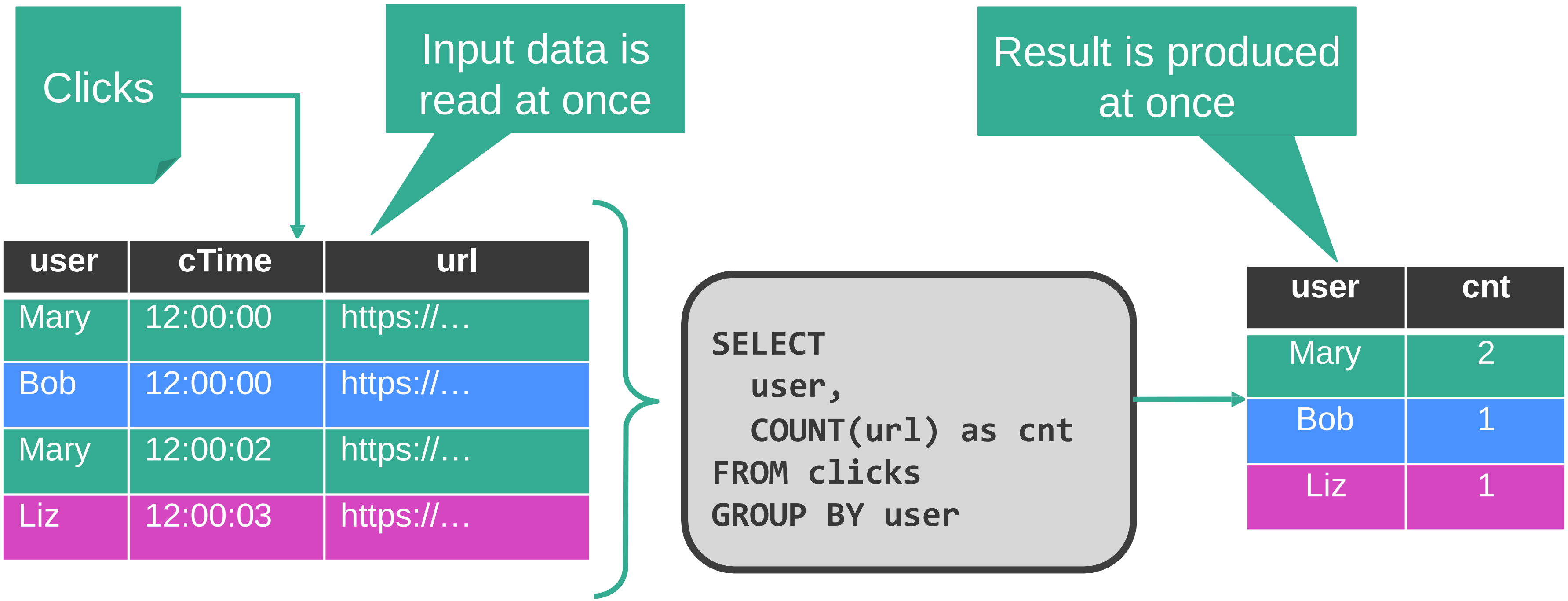


```
SELECT user, COUNT(url)
 FROM clicks
 GROUP BY user
```

Input data is
bounded
(batch)

Input data is
unbounded
(streaming)

What if “Clicks” is a File?



Clicks

Input data is read at once

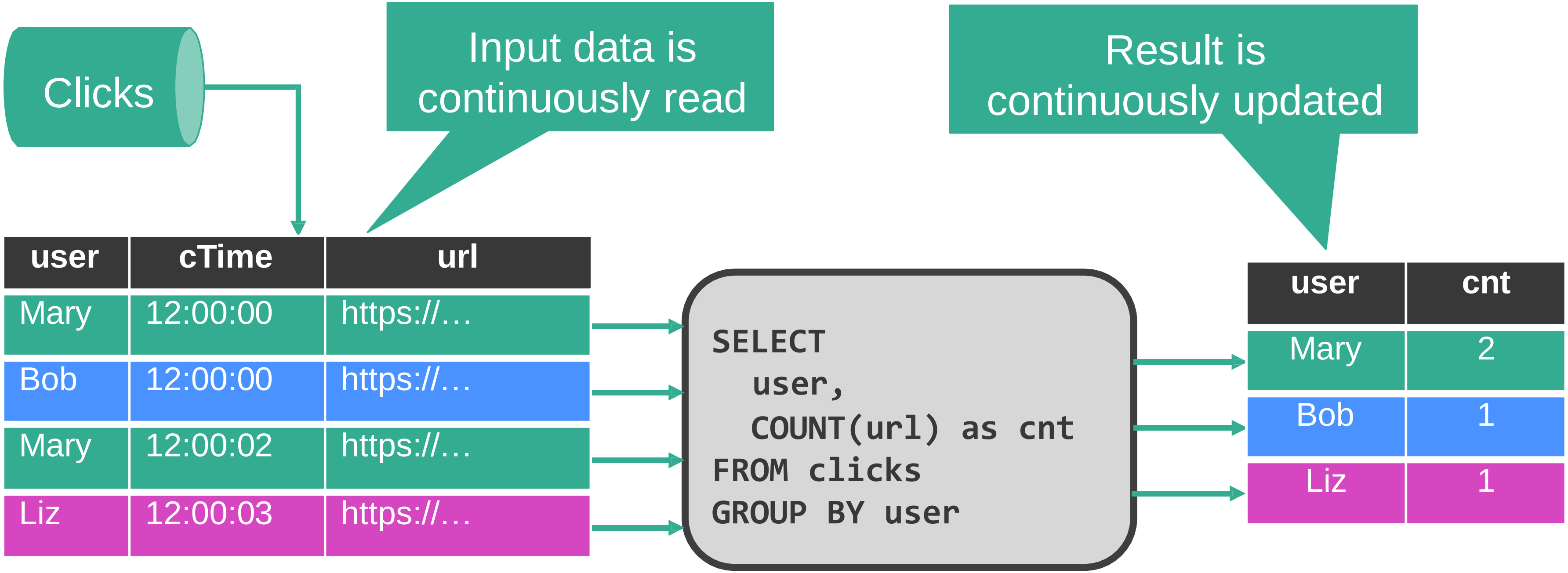
user	cTime	url
Mary	12:00:00	https://…
Bob	12:00:00	https://…
Mary	12:00:02	https://…
Liz	12:00:03	https://…

```
SELECT
  user,
  COUNT(url) as cnt
FROM clicks
GROUP BY user
```

Result is produced at once

user	cnt
Mary	2
Bob	1
Liz	1

What if “Clicks” is a Stream?



Clicks

Input data is continuously read

Result is continuously updated

user	cTime	url
Mary	12:00:00	https://…
Bob	12:00:00	https://…
Mary	12:00:02	https://…
Liz	12:00:03	https://…

```
SELECT
  user,
  COUNT(url) as cnt
FROM clicks
GROUP BY user
```

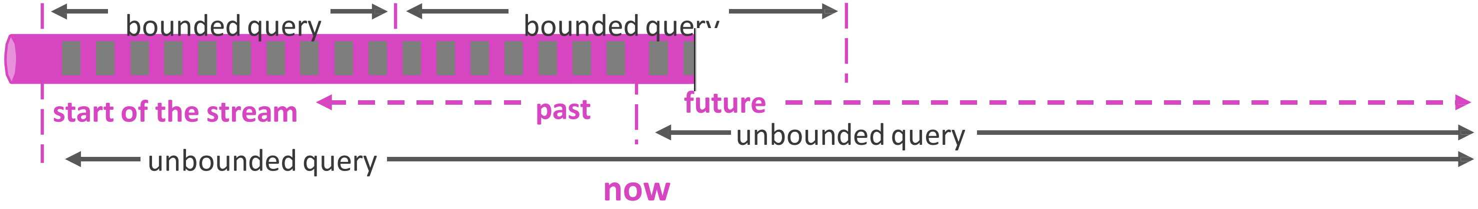
user	cnt
Mary	2
Bob	1
Liz	1

The result is the same!

Why is Stream-Batch Unification Important?

- Usability
  - ANSI SQL syntax: No custom “StreamSQL” syntax.
  - ANSI SQL semantics: No stream-specific result semantics.

- Portability
  - Run the same query on *bounded* & *unbounded* data
  - Run the same query on *recorded* & *real-time* data
  - *Bootstrapping* query state or *backfilling* results from historic data



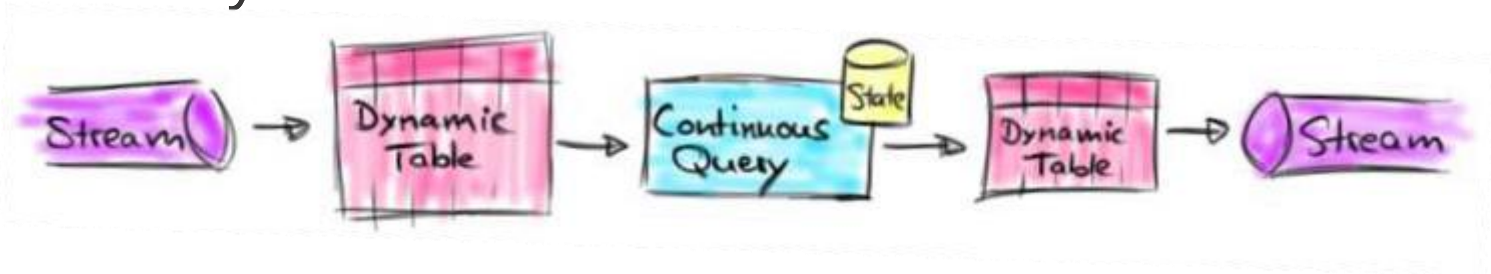
Database Systems Run Queries on Streams

- Materialized views (MV) are similar to regular views, but persisted to disk or memory
  - Used to speed-up analytical queries
  - MVs need to be updated when the base tables change

- MV maintenance is very similar to SQL on streams
  - Base table updates are a stream of DML statements
  - MV definition query is evaluated on that stream
  - MV is query result and continuously updated

Continuous Queries in Flink

- Core concept is a *“Dynamic Table”*
  - Dynamic tables are changing over time
- Queries on dynamic tables
  - produce new dynamic tables (which are updated based on input)
  - do not terminate
- Stream ↔ Dynamic table conversions



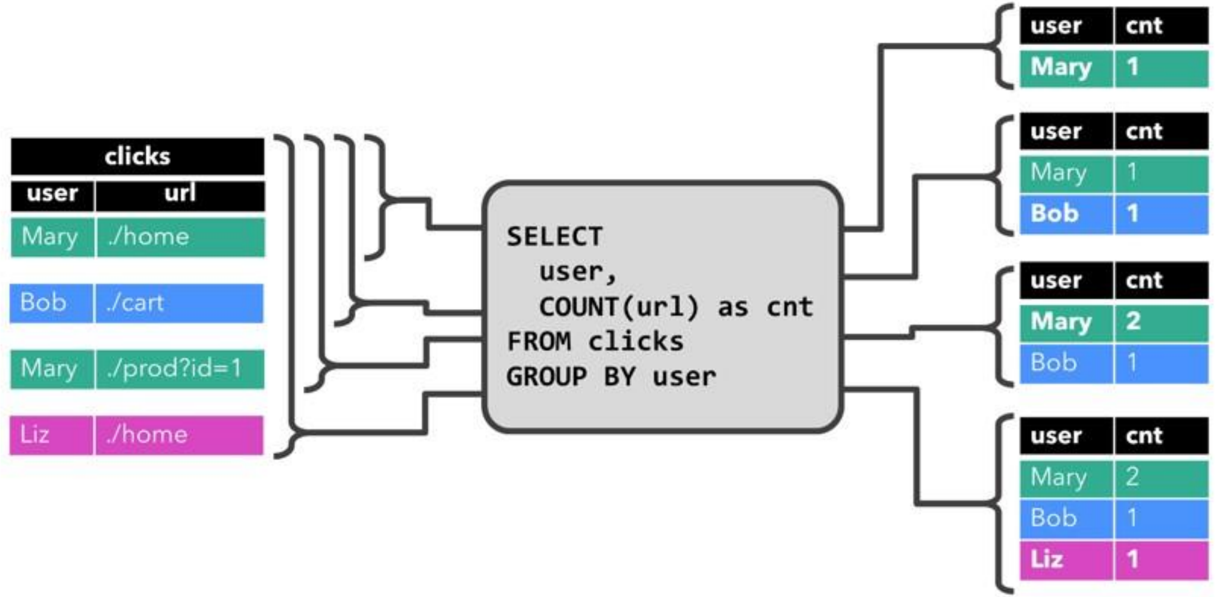
Stream ↔ Dynamic Table Conversions

- A stream is the changelog of a dynamic table
  - As change messages are ingested from a stream, a table evolves
  - As a table evolves, change messages are emitted to a stream

- Different changelog interpretations
  - Append-only change messages
  - Upsert change messages
  - Add/Retract change messages

Continuous Queries

Based on the nature of the query, the result table might contain insert, update, upsert or delete records



clicks	
user	url
Mary	./home
Bob	./cart
Mary	./prod?id=1
Liz	./home

```
SELECT
  user,
  COUNT(url) as cnt
FROM clicks
GROUP BY user
```

user	cnt
Mary	**1**

user	cnt
Mary	1
Bob	**1**

user	cnt
Mary	**2**
Bob	1

user	cnt
Mary	2
Bob	1
Liz	**1**

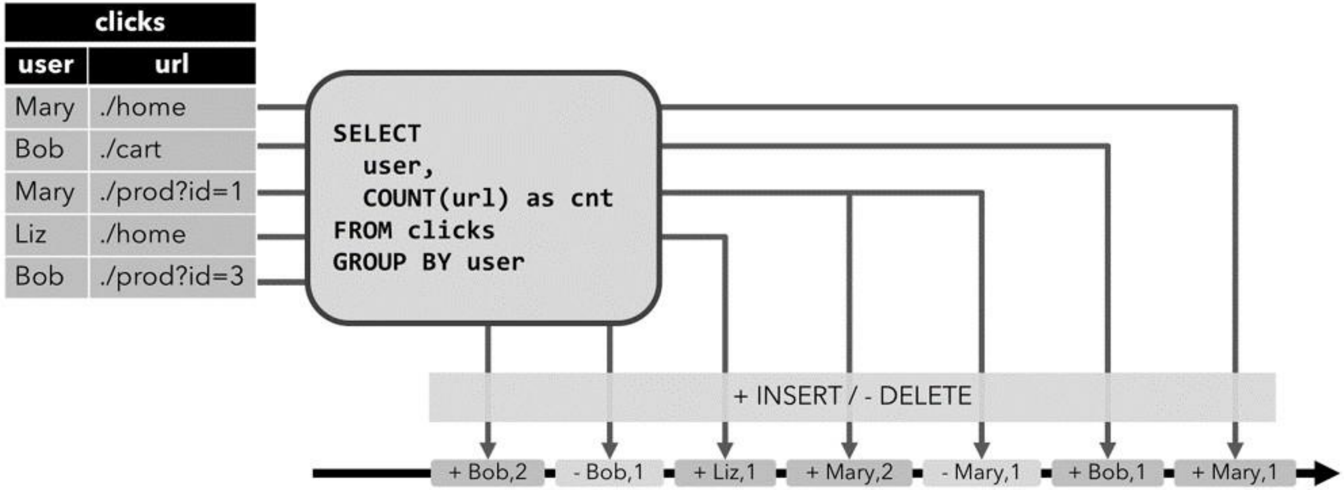
source: https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/table/streaming/dynamic_tables.html

Table to Stream (Concepts)

- When writing a dynamic table result back to a stream, we have to encode the type of change that occurred to the table: insert, delete or update

- Flink supports three ways to encode changes, recall the Dataflow model,
  - Append-only streams: only insert records are emitted on the stream
  - Retract streams: two messages are sent add and retract
    - Add: add message
    - Update: retract old value then add the new value
    - Delete: retract message
  - Upsert streams: both insert and update are handled as an upsert message. Delete is a retract message. Not yet implemented in Flink.

Retract Stream



clicks	
user	**url**
Mary	./home
Bob	./cart
Mary	./prod?id=1
Liz	./home
Bob	./prod?id=3

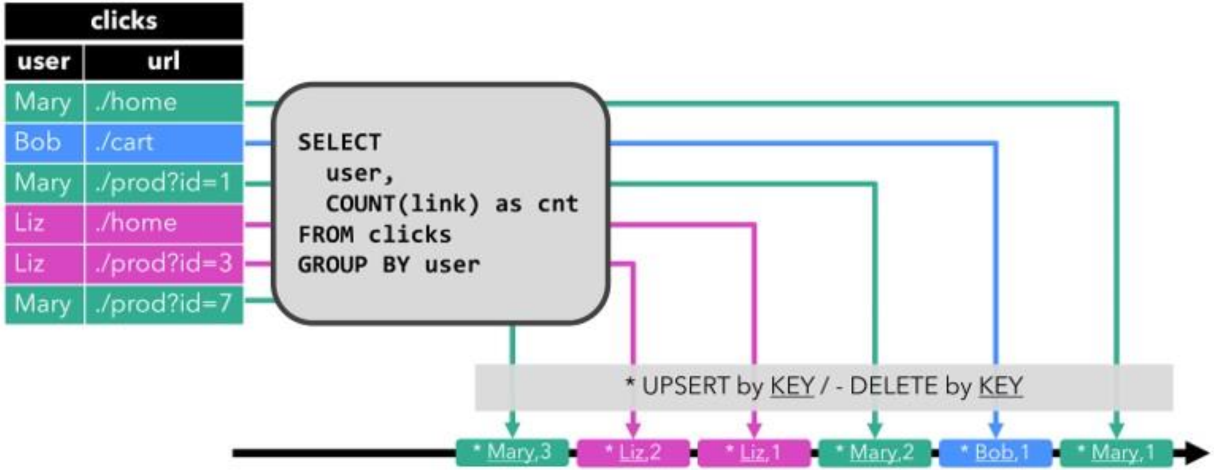
```
SELECT
  user,
  COUNT(url) as cnt
FROM clicks
GROUP BY user
```

+ INSERT / - DELETE

+ Bob,2 - Bob,1 + Liz,1 + Mary,2 - Mary,1 + Bob,1 + Mary,1

Upsert Stream

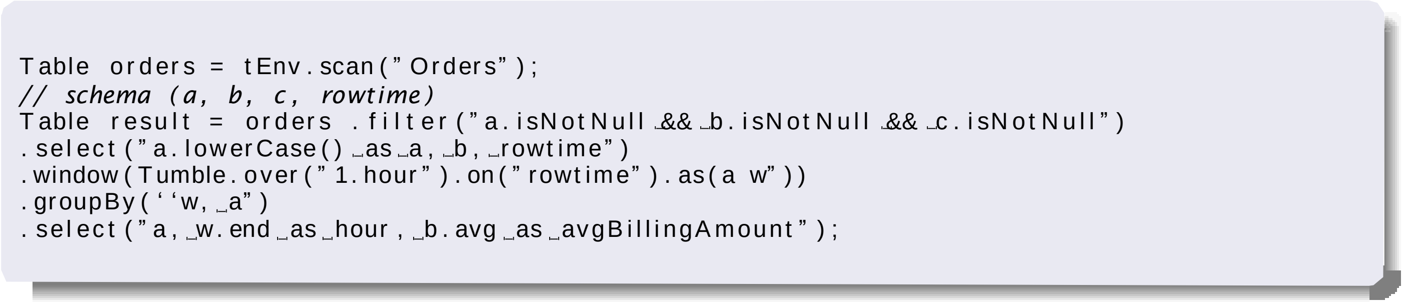
- Less number of messages to be emitted
- Receiver operator has to be aware of message encoding
- A unique key is required



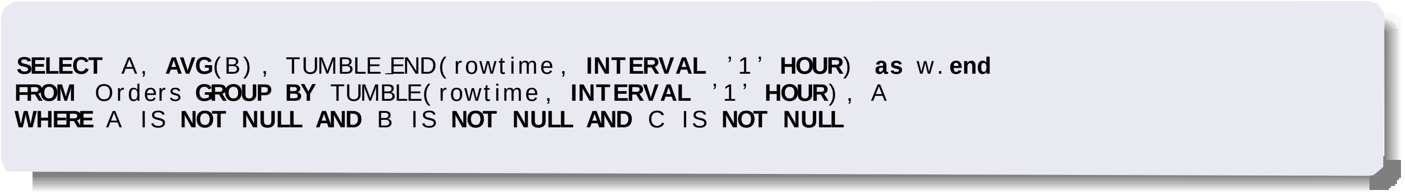
clicks	
user	url
Mary	./home
Bob	./cart
Mary	./prod?id=1
Liz	./home
Liz	./prod?id=3
Mary	./prod?id=7

```
SELECT
  user,
  COUNT(link) as cnt
FROM clicks
GROUP BY user
```

Table API/SQL Examples



```
Table orders = tEnv.scan(”Orders”);
// schema (a, b, c, rowtime)
Table result = orders .filter(”a.isNotNull␣&&␣b.isNotNull␣&&␣c.isNotNull”)
.select(”a.lowerCase()␣as␣a,␣b,␣rowtime”)
.window(Tumble.over(”1.hour”).on(”rowtime”).as(a w”))
.groupBy(‘‘w,␣a”)
.select(”a,␣w.end␣as␣hour,␣b.avg␣as␣avgBillingAmount”);
```



```
SELECT A, AVG(B), TUMBLE_END(rowtime, INTERVAL ’1’ HOUR) as w.end
FROM Orders GROUP BY TUMBLE(rowtime, INTERVAL ’1’ HOUR), A
WHERE A IS NOT NULL AND B IS NOT NULL AND C IS NOT NULL
```

Table API Examples

```
Table orders = tEnv.scan(”Orders”);
// schema (a, b, c, rowtime)
Table result = orders .filter(”a.isNotNull␣&&␣b.isNotNull␣&&␣c.isNotNull”)
.select(”a.lowerCase()␣as␣a,␣b,␣rowtime”)
.window(Slide.over(”10.minutes”).every(”5.minutes”).on(”rowtime”).as(”w”))
.groupBy(‘‘w, a”)
.select(”a, w.end as hour, b.avg as avgBillingAmount”);
```

```
SELECT A, AVG(B), HOP_END(rowtime, INTERVAL ’10’ MINUTES, INTERVAL ’5’ MINUTES ) as w.end
FROM Orders GROUP BY HOP(rowtime, INTERVAL ’10’ MINUTES, INTERVAL ’5’ MINUTES ), A
WHERE A is NOT NULL AND B IS NOT NULL AND C IS NOT NULL
```

Table API Examples

```
Table orders = tEnv.scan(”Orders”);
// schema (a, b, c, rowtime)
Table result = orders .filter(”a.isNotNull␣&&␣b.isNotNull␣&&␣c.isNotNull”)
.select(”a.lowerCase()␣as␣a,␣b,␣rowtime”)
.window(Slide.over(”10.rows”).every(”5.rows”).on(”proctime”).as(”w”))
.groupBy(”w,␣a”)
select(”a,␣w.end␣as␣hour,␣b.avg␣as␣avgBillingAmount”);
```

Table API Examples

```
Table orders = tEnv.scan(”Orders”);
// schema (a, b, c, rowtime)
Table result = orders .filter(”a.isNotNull␣&&␣b.isNotNull␣&&␣c.isNotNull”)
.select(”a.lowerCase()␣as␣a,␣b,␣rowtime”)
.window(Session.withGap(”10.minutes”).on(”rowtime”).as(”w”))
.groupBy(”w,␣a”)
.select(”a,␣w.end␣as␣hour,␣b.avg␣as␣avgBillingAmount”);
```

```
SELECT A, AVG(B), SESSION_END(rowtime, INTERVAL ’10’ MINUTES) as w.end
FROM Orders GROUP BY SESSION(rowtime, INTERVAL ’10’ MINUTES), A
WHERE A IS NOT NULL AND B IS NOT NULL AND C IS NOT NULL
```

SQL Feature Set in Flink >= 1.8.0

STREAMING & BATCH

- SELECT FROM WHERE
- GROUP BY [HAVING]
  - Non-windowed
  - TUMBLE, HOP, SESSION windows
- JOIN
  - Time-Windowed INNER + OUTER JOIN
  - Non-windowed INNER + OUTER JOIN
- User-Defined Functions
  - Scalar
  - Aggregation
  - Table-valued

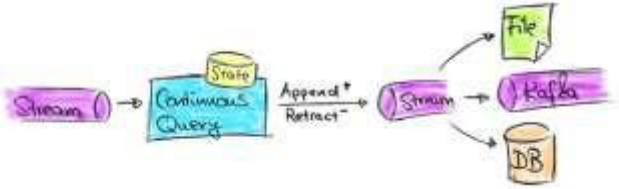
STREAMING ONLY

- OVER / WINDOW
  - UNBOUNDED / BOUNDED PRECEDING
- INNER JOIN with time-versioned table
- MATCH_RECOGNIZE
  - Pattern Matching/CEP (SQL:2016)

BATCH ONLY

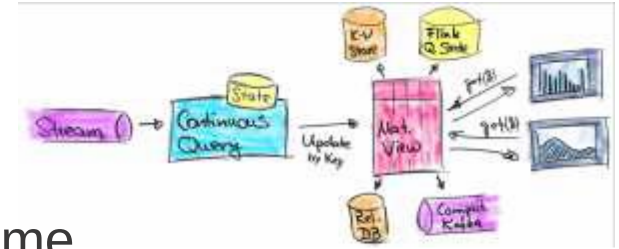
- UNION / INTERSECT / EXCEPT
- ORDER BY

What Can I Build With That?



- Data Pipelines & Low-latency ETL
  - Transform, aggregate, and move events in real-time
  - Write streams to file systems, DBMS, K-V stores, …
  - Ingest appearing files to produce streams

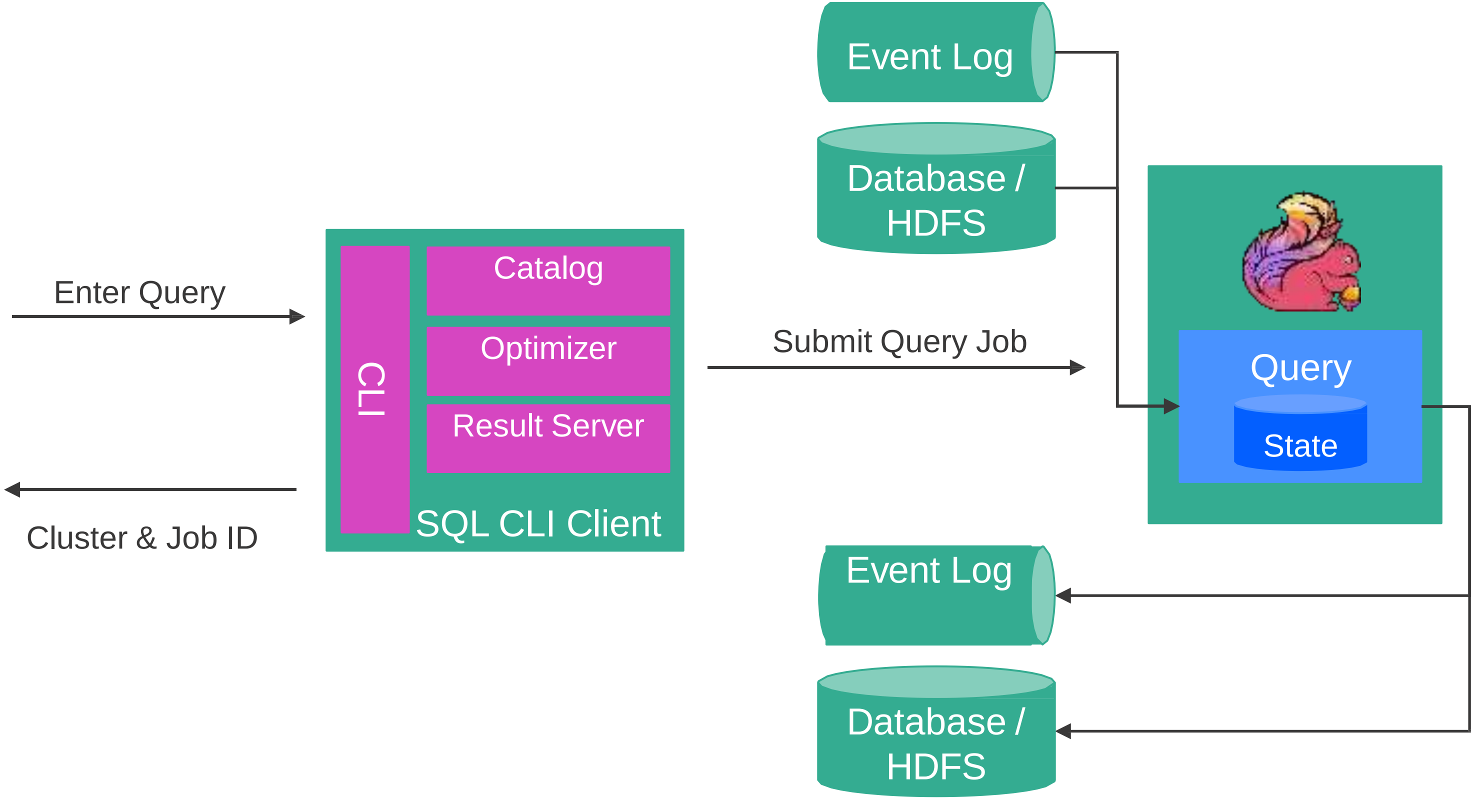
- Stream & Batch Analytics
  - Run analytical queries over bounded and unbounded data
  - Query and compare historic and real-time data



- Power Live Dashboards
  - Compute and update data to visualize in real-time

SQL CLI Client – Detached Queries

```
INSERT INTO
sinkTable
SELECT
  user,
  COUNT(url) AS cnt
FROM clicks
GROUP BY user
```



The New York Taxi Rides Data Set

- A public data set about taxi rides in New York City
- Rides are ingested as append-only (streaming) table
  - Each ride is represented by a start and an end event
- Table: **`Rides`**

```
rideId:    BIGINT     // ID of the taxi ride
taxiId:    BIGINT     // ID of the taxi
isStart:   BOOLEAN    // flag for pick-up (true) or drop-off (false) event
lon:       DOUBLE     // longitude of pick-up or drop-off location
lat:       DOUBLE     // latitude of pick-up or drop-off location
rideTime:  TIMESTAMP  // time of pick-up or drop-off event
psgCnt:    INT        // number of passengers
```

Compute Basic Statistics

- ***Count rides*** per ***number of passengers***.

```
SELECT
  psgCnt,
  COUNT(*) as cnt
FROM Rides
WHERE isStart
GROUP BY
  psgCnt
```

Identify Popular Pick-Up / Drop-Off Locations

- Compute ***every 5 minutes*** for ***each area*** the ***number of departing and arriving taxis***.

```
SELECT
  area,
  isStart,
  TUMBLE_END(rideTime, INTERVAL '5' MINUTE) AS cntEnd,
  COUNT(*) AS cnt
FROM (SELECT rideTime, isStart, toAreaId(lon, lat) AS area
      FROM Rides)
GROUP BY
  area,
  isStart,
  TUMBLE(rideTime, INTERVAL '5' MINUTE)
```

Average Tip Per Hour of Day

- Compute the ***average tip*** per ***hour of day***. Fare data is stored in a separate table ***Fares*** *that needs to be **joined**.*

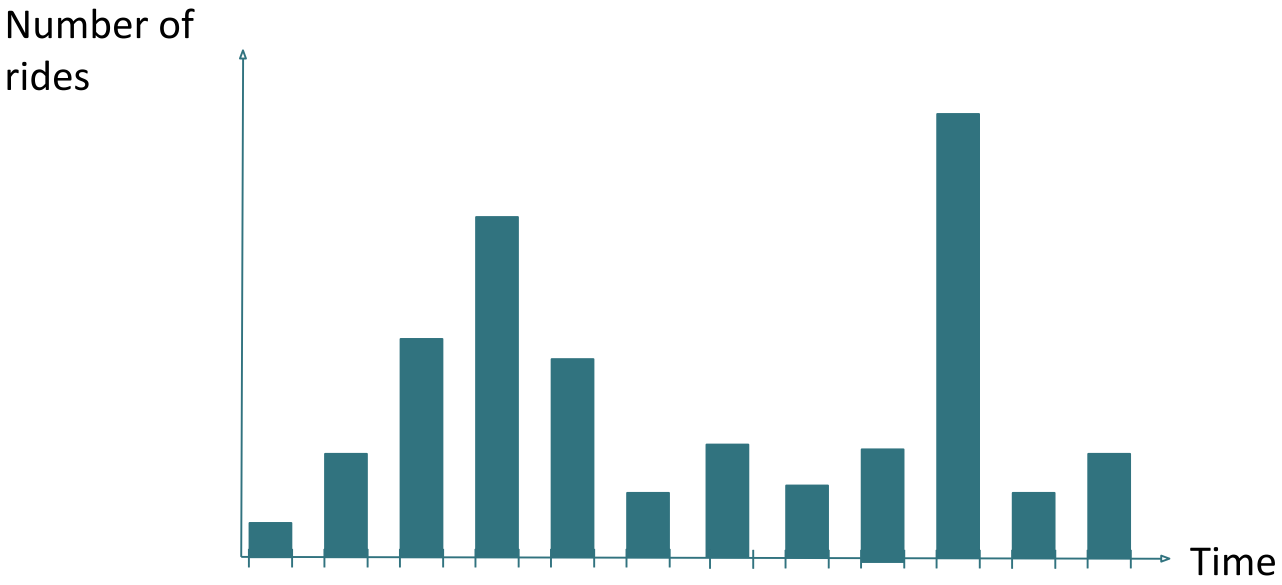
```
SELECT
  HOUR(r.rideTime) AS hourOfDay,
  AVG(f.tip) AS avgTip
FROM
  Rides r,
  Fares f
WHERE
  NOT r.isStart AND
  r.rideId = f.rideId AND
  f.payTime BETWEEN r.rideTime - INTERVAL '5' MINUTE AND r.rideTime
GROUP BY
  HOUR(r.rideTime);
```

Average Ride Duration Per Pick-Up Location

- ***Join start ride*** and ***end ride*** events ***on rideId*** and compute ***average ride duration per pick-up location***.

```
SELECT pickUpArea,
       AVG(timeDiff(s.rowTime, e.rowTime) / 60000) AS avgDuration
FROM (SELECT rideId, rowTime, toAreaId(lon, lat) AS pickUpArea
      FROM TaxiRides
      WHERE isStart) s
   JOIN
     (SELECT rideId, rowTime
      FROM TaxiRides
      WHERE NOT isStart) e
    ON s.rideId = e.rideId AND
       e.rowTime BETWEEN s.rowTime AND s.rowTime + INTERVAL '1' HOUR
GROUP BY pickUpArea
```

Number of rides per 30 minutes per area

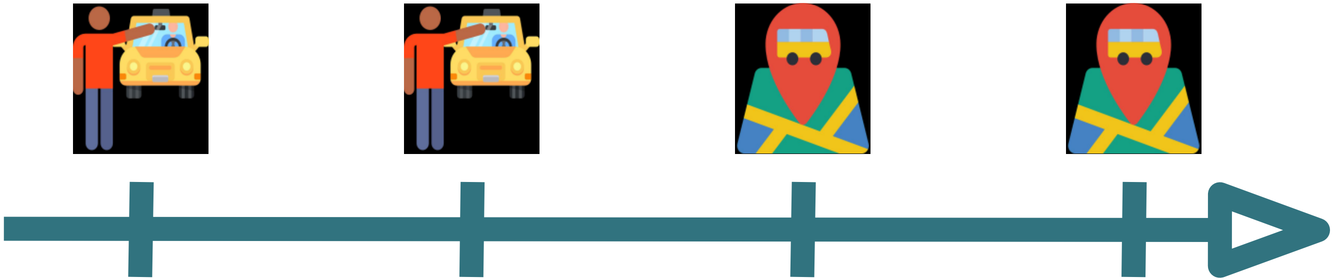


SQL Example

```
SELECT
    toAreaId(lat, lon) as cellId,
    COUNT(distinct rideId) as rideCount,
    TUMBLE_ROWTIME(rowTime, INTERVAL '30' minute) AS rowTime,
    TUMBLE_START(rowTime, INTERVAL '30' minute) AS startTime,
    TUMBLE_END(rowTime, INTERVAL '30' minute) AS endTime
FROM
    TaxiRides
GROUP BY
    toAreaId(lat, lon),
    TUMBLE(rowTime, INTERVAL '30' minute)
```

What is hard with SQL?

- Find rides with mid-stops



Mid Stops: Pure SQL

Rides table

```
rideId:   BIGINT     // ID of the taxi ride
taxiId:   BIGINT     // ID of the taxi
isStart:  BOOLEAN    // flag for pick-up (true) or drop-off (false) event
lon:      DOUBLE     // longitude of pick-up or drop-off location
lat:      DOUBLE     // latitude of pick-up or drop-off location
rideTime: TIMESTAMP  // time of pick-up or drop-off event
psgCnt:   INT        // number of passengers
```

Mid Stops: Pure SQL

```
select start.rideId from rides as start, rides as end
where start.taxiId = end.taxiId
and start.rideTime < end.rideTime
and start.isStart and end.isStart
and not exists (select 1 from rides as inbetweenNewRide
                    where inbetweenNewRide.taxiId = start.taxiId
                    and inbetweenNewRide.isStart
                    and inbetweenNewRide.rideTime > start.rideTime
                    and inbetweenNewRide.rideTime < end.rideTime     )
and not exists (select 1 from rides as inbetweenDrop
          where inbetweenDrop.taxiId = start.taxiId
          and inbetweenDrop.isStart = 0
          and inbetweenDrop.rideTime > start.rideTime
          and inbetweenDrop.rideTime < end.rideTime                           )
```

Mid Stops : Pattern Matching

```
Pattern.<Row>begin("S”).where(
(row) -> {
return row.isStart == true;
}).next("E").where( (row) -> {
return row.isStart == true;
});
```

```
CEP.pattern(input.keyBy(“driverId”),
pattern)
.flatSelect(
new PatternFlatSelectFunction<Row,
Row>() {
@Override
public void flatSelect(
Map<String, List<Row>> pattern,
Collector<Row> out) throws Exception {
out.collect((
pattern.get(“S”).get(0).getRideId
));
}
);
```

MATCH_RECOGNIZE

SQL:2016 extension

Common use-cases

- stock market analysis
- customer behaviour
- tracking money laundering
- service quality
- network intrusion detection

Position in a SQL Query

SELECT ...

FROM ...

MATCH_RECOGNIZE

(...)

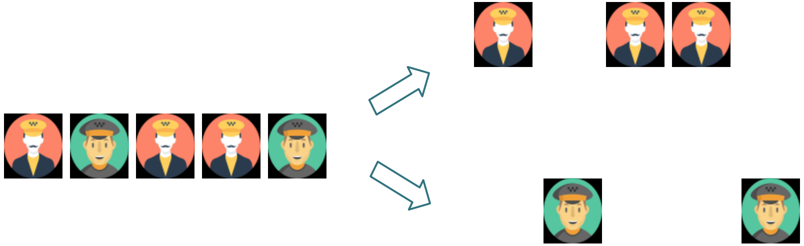
WHERE ...

GROUP BY ...

Rides with mid-stops

```
SELECT *
FROM TaxiRides
MATCH_RECOGNIZE (
    PARTITION BY driverId
    ORDER BY rowTime
    MEASURES
        S.rideId as sRideId
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (S E)
    DEFINE
      S AS S.isStart = true,
        E AS E.isStart = true
)
```

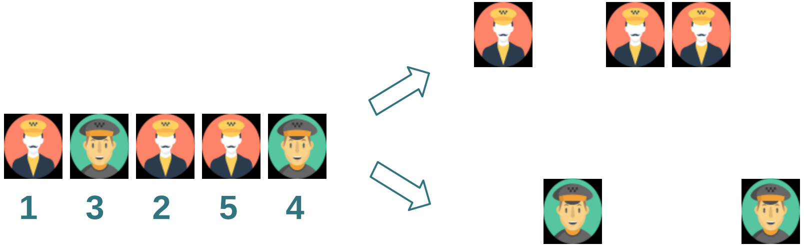
partition the data by given field = keyBy



Rides with mid-stops

```
SELECT *
FROM TaxiRides
MATCH_RECOGNIZE (
    PARTITION BY driverId
    ORDER BY rowTime
    MEASURES
        S.rideId as sRideId
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (S E)
    DEFINE
     S AS S.isStart = true,
        E AS E.isStart = true
)
```

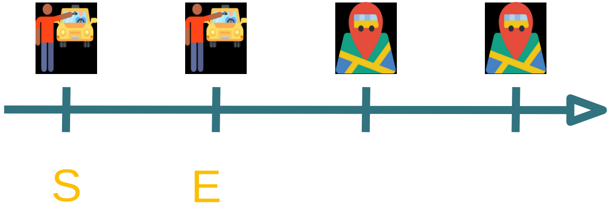
specify order
primary order = Event
or Processing time



Rides with mid-stops

```
SELECT *
FROM TaxiRides
MATCH_RECOGNIZE (
    PARTITION BY driverId
    ORDER BY rowTime
    MEASURES
        S.rideId as sRideId
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (S E)
    DEFINE
     S AS S.isStart = true,
     E AS E.isStart = true
)
```

construct pattern



PATTERN: Defining a Pattern

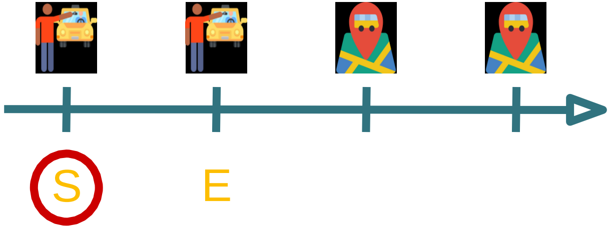
- *Concatenation:*
  - All rows of a pattern must be mapped to pattern variables
  - A pattern like (A B) means that the contiguity is strict between A and B
  - In other words: No rows between A and B
- *Quantifiers*
  - Number of rows mapped to a pattern variable

*	0 or more rows
+	1 or more rows
?	0 or 1 rows
{ n }, { n, }, { n, m }, { , m }	Define intervals (inclusive)
B*?	Perform mapping *reluctant* instead of *greedy* (default behavior)

Rides with mid-stops

```
SELECT *
FROM TaxiRides
MATCH_RECOGNIZE (
    PARTITION BY driverId
    ORDER BY rowTime
    MEASURES
        S.rideId as sRideId
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (S E)
    DEFINE
     S AS S.isStart = true,
     E AS E.isStart = true
)
```

extract measures from matched sequence

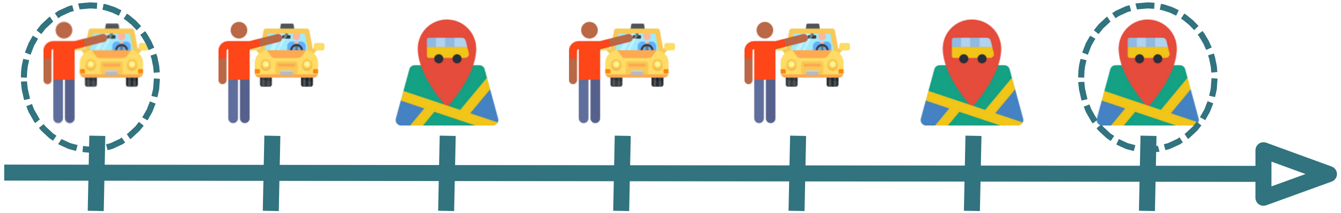


DEFINE/MEASURES: Define/Access Variables

- MEASURES
  - Defines what will be included in the output of a matching pattern
  - Project columns and define expressions for evaluation
  - Number of produced rows depends on the output mode.
    Currently, `ONE ROW PER MATCH` = one output summary row per match only
  - Output schema: *[partitioning columns]* + *[measures columns]*

- DEFINE
  - Conditions that a row has to fulfill to be classified to the corresponding variable
  - No condition for a variable evaluates to *TRUE*

Multi-Stops



Rides with more than one mid-stop

```
SELECT *
FROM TaxiRides
MATCH_RECOGNIZE (
    PARTITION BY driverId
    ORDER BY rowTime
    MEASURES
        S.rideId as sRideId
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (S E)
    DEFINE
     S AS S.isStart = true,
     E AS E.isStart = true
)
```

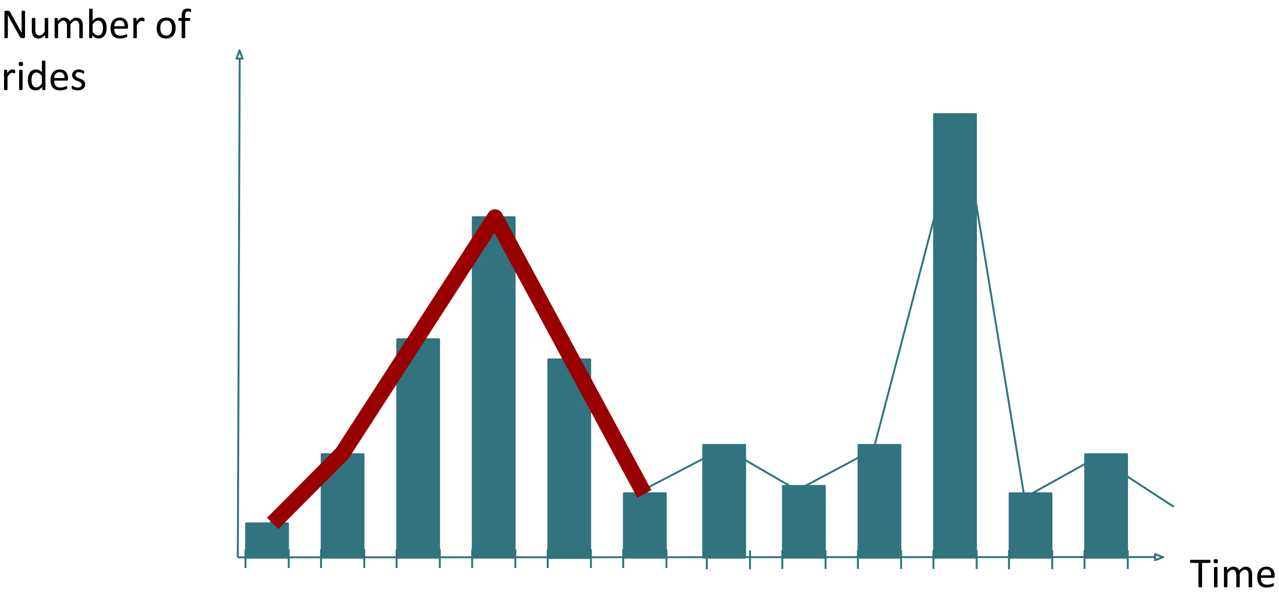
```
SELECT *
FROM TaxiRides
MATCH_RECOGNIZE (
    PARTITION BY driverId
    ORDER BY rowTime
    MEASURES
        S.rideId as sRideId,
        COUNT(M.rideId) as countMidStops
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (S M{2,} E)
    DEFINE
     S AS S.isStart = true,
     M AS M.rideId <> S.rideId,
     E AS E.isStart = false AND E.rideId = S.rideId
)
```

Rides with more than one mid-stop

```
SELECT *
FROM TaxiRides
MATCH_RECOGNIZE (
      PARTITION BY driverId
      ORDER BY rowTime
      MEASURES
            S.rideId as sRideId
      AFTER MATCH SKIP PAST LAST ROW
      PATTERN (S E)
      DEFINE
        S AS S.isStart = true,
        E AS E.isStart = true
)
```

```
SELECT *
FROM TaxiRides
MATCH_RECOGNIZE (
      PARTITION BY driverId
      ORDER BY rowTime
      MEASURES
            S.rideId as sRideId,
            COUNT(M.rideId) as countMidStops
      AFTER MATCH SKIP PAST LAST ROW
      PATTERN (S M{2,} E)
      DEFINE
        S AS S.isStart = true,
        M AS M.rideId <> S.rideId,
        E AS E.isStart = false AND E.rideId = S.rideId
)
```

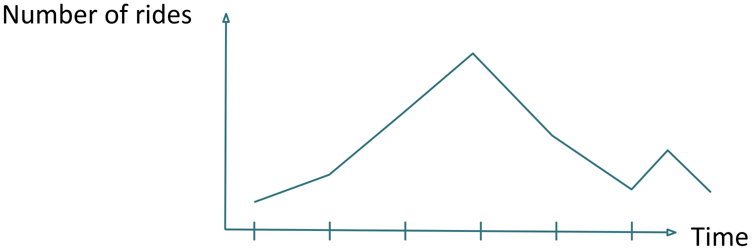
Rush (peak) hours – V Shape Upside down



Statistics per Area

```
CREATE VIEW RidesInArea AS
SELECT
        toAreaId(lat, lon) as cellId,
        COUNT(distinct rideId) as rideCount,
        TUMBLE_ROWTIME(rowTime, INTERVAL '30' minute) AS rowTime,
        TUMBLE_START(rowTime, INTERVAL '30' minute) AS startTime,
        TUMBLE_END(rowTime, INTERVAL '30' minute) AS endTime
FROM
        TaxiRides
GROUP BY
        toAreaId(lat, lon),
        TUMBLE(rowTime, INTERVAL '30' minute)
```

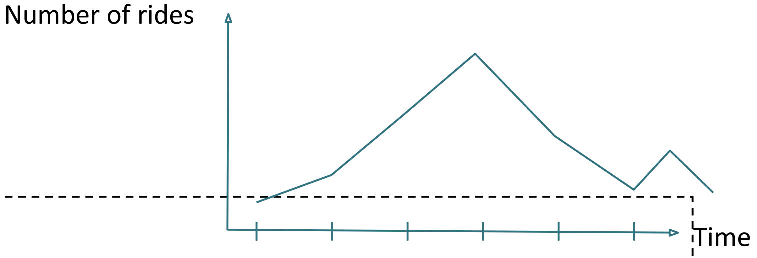
Rush hours



Use previous table/view

```
SELECT *  FROM RidesInArea
MATCH_RECOGNIZE(
        PARTITION BY cellId
        ORDER BY rowTime
        MEASURES
                FIRST(UP.startTime) as rushStart,
                LAST(DOWN.endTime) AS rushEnd,
                SUM(UP.rideCount) + SUM(DOWN.rideCount) AS rideSum
        AFTER MATCH SKIP PAST LAST ROW
        PATTERN (UP{4,} DOWN{2,} E)
        DEFINE
                UP AS UP.rideCount > LAST(UP.rideCount, 1) or LAST(UP.rideCount, 1) IS NULL,
                DOWN AS DOWN.rideCount < LAST(DOWN.rideCount, 1) OR
                                LAST(DOWN.rideCount, 1) IS NULL,
                E AS E.rideCount > LAST(DOWN.rideCount)
)
```

Rush hours

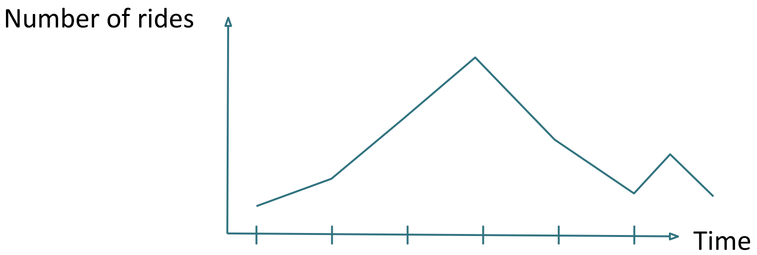


Use previous table/view

```
SELECT *  FROM RidesInArea
MATCH_RECOGNIZE(
        PARTITION BY cellId
        ORDER BY rowTime
        MEASURES
                FIRST(UP.startTime) as rushStart,
                LAST(DOWN.endTime) AS rushEnd,
                SUM(UP.rideCount) + SUM(DOWN.rideCount) AS rideSum
        AFTER MATCH SKIP PAST LAST ROW
        PATTERN (UP{4,} DOWN{2,} E)
        DEFINE
                UP AS UP.rideCount > LAST(UP.rideCount, 1) or LAST(UP.rideCount, 1) IS NULL,
                DOWN AS DOWN.rideCount < LAST(DOWN.rideCount, 1) OR
                                LAST(DOWN.rideCount, 1) IS NULL,
                E AS E.rideCount > LAST(DOWN.rideCount)
)
```

Apply match to the result of the inner query

Rush hours

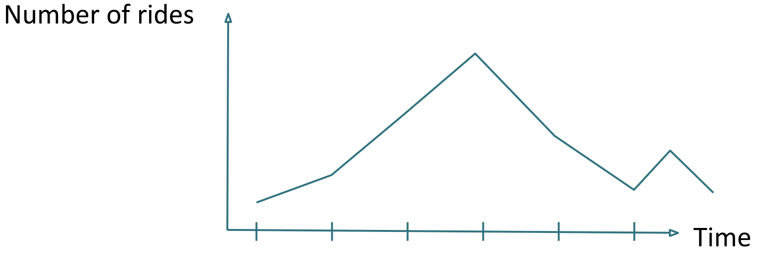


```
SELECT *  FROM RidesInArea
MATCH_RECOGNIZE(
          PARTITION BY cellId
          ORDER BY rowTime
          MEASURES
                    FIRST(UP.startTime) as rushStart,
                    LAST(DOWN.endTime) AS rushEnd,
                    SUM(UP.rideCount) + SUM(DOWN.rideCount) AS rideSum
          AFTER MATCH SKIP PAST LAST ROW
          PATTERN (UP{4,} DOWN{2,} E)
          DEFINE
                    UP AS UP.rideCount > LAST(UP.rideCount, 1) or LAST(UP.rideCount, 1) IS NULL,
                    DOWN AS DOWN.rideCount < LAST(DOWN.rideCount, 1) OR
                                        LAST(DOWN.rideCount, 1) IS NULL,
                    E AS E.rideCount > LAST(DOWN.rideCount)
)
```

access elements of looping pattern in measures

Access elements in looping pattern define

Rush hours



```
SELECT *  FROM RidesInArea
MATCH_RECOGNIZE(
        PARTITION BY cellId
        ORDER BY rowTime
        MEASURES
                FIRST(UP.startTime) as rushStart,
                LAST(DOWN.endTime) AS rushEnd,
                SUM(UP.rideCount) + SUM(DOWN.rideCount) AS rideSum
        AFTER MATCH SKIP PAST LAST ROW
        PATTERN (UP{4,} DOWN{2,} E)
        DEFINE
                UP AS UP.rideCount > LAST(UP.rideCount, 1) or LAST(UP.rideCount, 1) IS NULL,
                DOWN AS DOWN.rideCount < LAST(DOWN.rideCount, 1) OR
                                LAST(DOWN.rideCount, 1) IS NULL,
                E AS E.rideCount > LAST(DOWN.rideCount)
)
```

aggregate values from looping patterns

DEFINE/MEASURES: Define/Access Variables

- *Pattern Variable Referencing*
  - Access to the set of rows mapped to a particular pattern variable (so far)
  - `A.price` = set of rows mapped so far to A plus the current row if we try to match the current row to A
  - If A is a set, the last row is selected for scalar operations.
  - If no pattern variable is specified (e.g. `SUM(price)`), the default pattern variable `"*"` is used. This set contains all rows matched for pattern + current row.

```
PATTERN (A B+)
DEFINE
          A AS A.price > 10,
          B AS B.price > A.price AND
                    SUM(price) < 100 AND SUM(B.price) < 80
```

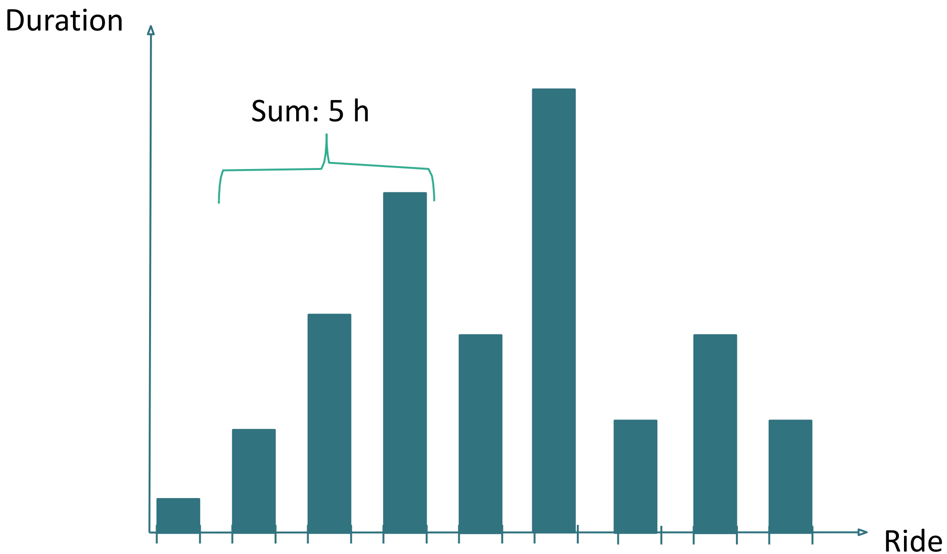
DEFINE/MEASURES: Define/Access Variables

- *Pattern Variable Navigation*
  - Logical offsets enable navigation within the events that were mapped to a particular pattern variable.
  - `FIRST(variable.field, n)` n starts from the beginning
  - `LAST(variable.field, n)` n starts from the end

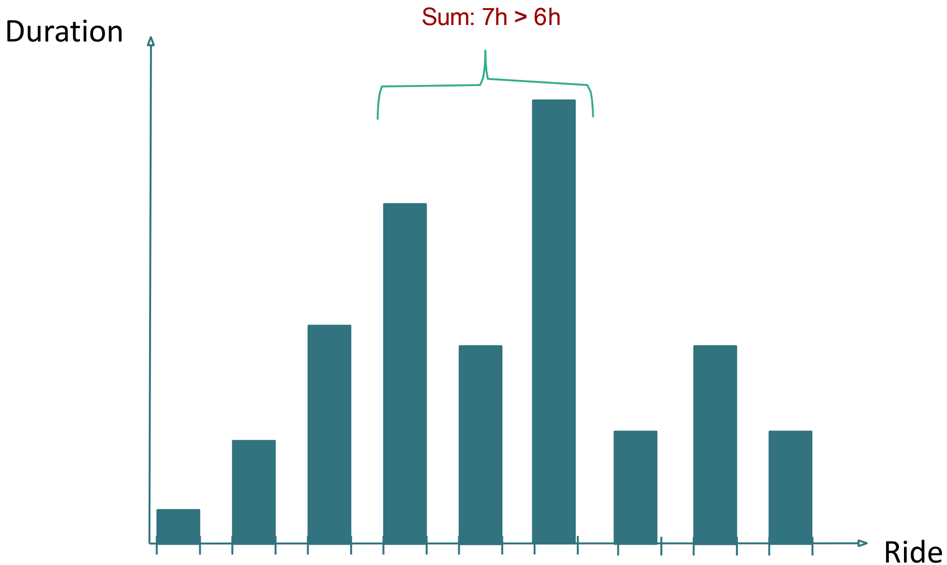
```
PATTERN (A B+)
DEFINE
            A AS A.price > 10,
            B AS (LAST(B.price, 1) IS NULL OR B.price > LAST(B.price, 1)) AND
                        (LAST(B.price, 2) IS NULL OR B.price > 2 * LAST(B.price, 2))
```

  - Expressions on same "list" are supported: `LAST(A.price * A.tax)`

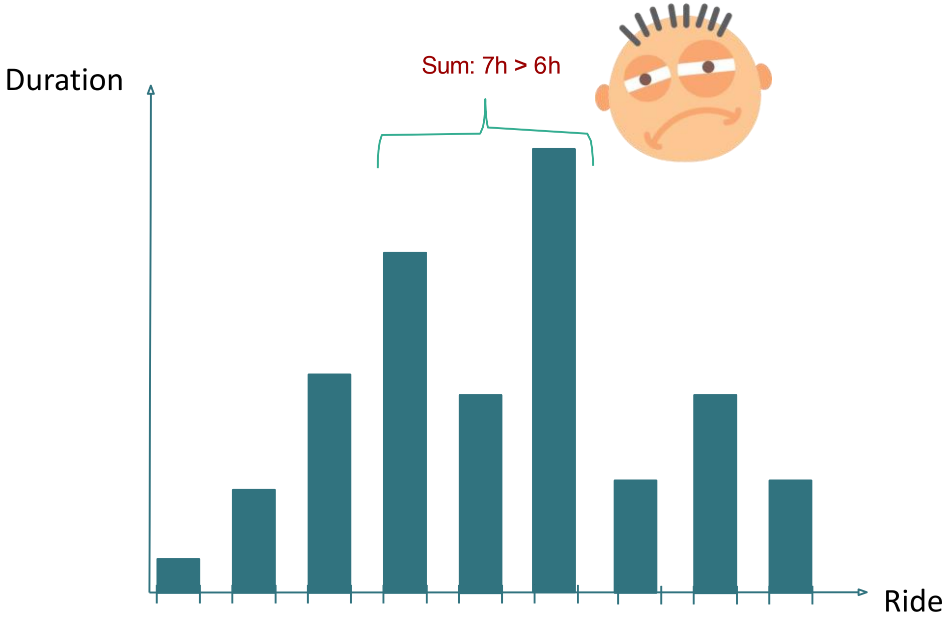
Driver Fatigue



Driver Fatigue



Driver Fatigue



Rides durations

```
CREATE VIEW RidesDurations AS
SELECT *  FROM TaxiRides
MATCH_RECOGNIZE (
    PARTITION BY driverId
    ORDER BY rowTime
    MEASURES
        rideId as rideId,
        timeDiff(S.rowTime, E.rowTime) as rideDuration,
        MATCH_ROWTIME() as rowTime,
        S.rowTime as startTime,
        E.rowTime AS endTime
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (S E)
    DEFINE
        S AS S.isStart = true,
        E AS E.isStart = false AND E.rideId = S.rideId
);
```

Rides durations

```
CREATE VIEW RidesDurations AS
SELECT *  FROM TaxiRides
MATCH_RECOGNIZE (
   PARTITION BY driverId
   ORDER BY rowTime
   MEASURES
         rideId as rideId,
         timeDiff(S.rowTime, E.rowTime) as rideDuration,
         MATCH_ROWTIME() as rowTime,
         S.rowTime as startTime,
         E.rowTime AS endTime
   AFTER MATCH SKIP PAST LAST ROW
   PATTERN (S E)
   DEFINE
         S AS S.isStart = true,
         E AS E.isStart = false AND E.rideId = S.rideId
);
```

Time attribute of the match

Drivers Fatigue

```
SELECT *
FROM RidesDurations
MATCH_RECOGNIZE (
    PARTITION BY driverId
    ORDER BY rowTime
    MEASURES
        formatDuration(SUM(rideDuration)) as totalRideDuration,
        FIRST(R.startTime) as startTime,
        LAST(R.endTime) as endTime
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (R+? E) WITHIN INTERVAL '1' DAY
    DEFINE
        E AS SUM(rideDuration) >= durationOfHours(2)
);
```

Drivers Fatigue

```
SELECT *
FROM RidesDurations
MATCH_RECOGNIZE (
  PARTITION BY driverId
  ORDER BY rowTime
  MEASURES
      formatDuration(SUM(rideDuration)) as totalRideDuration,
      FIRST(R.startTime) as startTime,
      LAST(R.endTime) as endTime
  AFTER MATCH SKIP PAST LA
  PATTERN (R+? E) WITHIN IN
  DEFINE
      E AS SUM(rideDuration) >= durationOfHours(2)
);
```

Aggregation in condition

Features set of MATCH_RECOGNIZE

- Quantifiers support:
  - + (one or more), * (zero or more), {x,y} (times)
  - greedy(default), ?(reluctant)
    - with some restrictions (not working for last pattern)
- After Match Skip
  - skip_to_first/last, skip_past_last, skip_to_next
- Aggregates (since 1.8)
- Allow time attribute extraction (since 1.8)
- Not supported:
  - alter(|), permute, exclude ‘{- -}’

AFTER MATCH SKIP: Continuation strategy

- Location where to start a new matching procedure after a complete match was found

SKIP PAST LAST ROW	next row after the last row of the current match
SKIP TO NEXT ROW	next row after the starting row of the match
SKIP TO LAST variable	last row that is mapped to the specified pattern variable
SKIP TO FIRST variable	first row that is mapped to the specified pattern variable

- Thus, also specifies how many matches a single event can belong to