

# Joining Dynamic Tables

---

Flink SQL Training

<https://github.com/ververica/sql-training>

# Joining Static Tables Is Well-Understood

- Join Types
  - INNER JOIN, [LEFT, RIGHT, FULL] OUTER JOIN, CROSS JOIN
- Join Predicates
  - Equality Predicates & Non-Equality Predicates
- Join Algorithms
  - Nested-Loops, Index-Nested-Loops, Sort-Merge, Hybrid-Hash, ...
- Static tables are completely available when a join is processed



# Joining Dynamic Streaming Tables Is Considered a Challenge

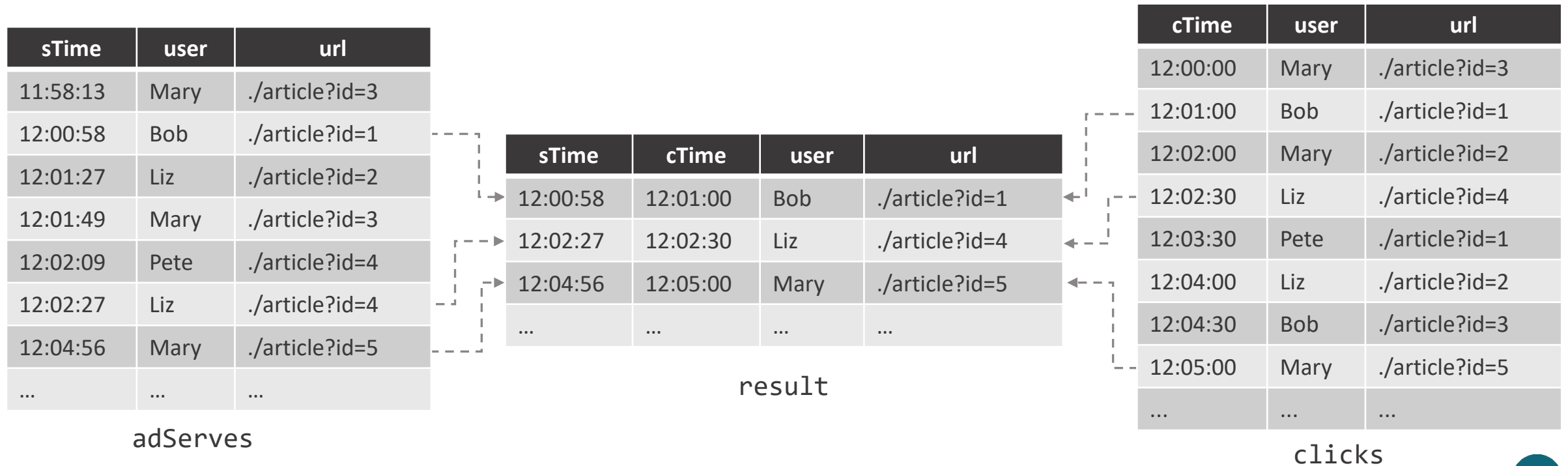
- Dynamic tables are constantly changing
  - Append-only tables can only be joined with time-constraining conditions
  - Updating tables need to be fully materialized
- Different requirements & terminology
  - People have different semantics in mind when talking about joining streams
  - There are different use cases for joining streams
- Flink SQL supports three common ways to join dynamic tables
  - Time-windowed joins
  - Joins with temporal tables
  - Regular / default joins



# Time-windowed Joins

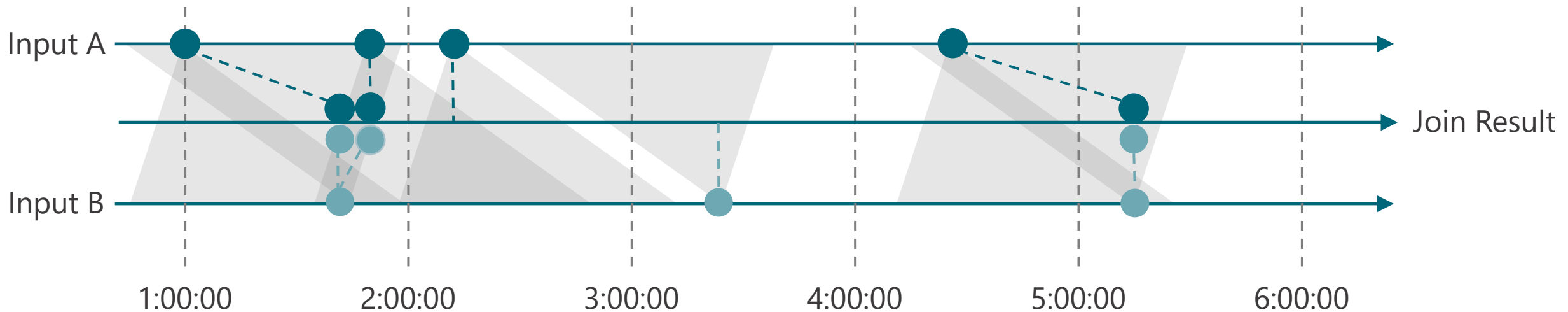
# Time-windowed Join: Use Case

- Given
  - A table `adServes` with served ads events: [servingTime, userId, URL]
  - A table `clicks` with click events: [clickTime, userId, URL]
- Find URLs that were clicked less than 5 seconds after being served as an ad



# Time-windowed Join

*A time-windowed join joins records of **two append-only tables** such that the **time attributes** of joined records **are not more than a specified window interval apart from each other.***



Every record of Input A is joined with all records of Input B that arrived at most 15 minutes earlier or 1 hour later.  
Every record of Input B is joined with all records of Input B that arrived at most 1 hour earlier or 15 minutes later.



# Time-windowed Join: Requirements and Syntax

- Tables A and B are append-only (records are never updated!)
- The join condition includes
  - An equality predicate
  - A predicate defining a closed window around A's and B's time attributes

```
SELECT *  
FROM A, B  
WHERE A.id = B.id AND  
      A.t BETWEEN B.t - INTERVAL '15' MINUTE AND  
      B.t + INTERVAL '1' HOUR
```

Every record of A is joined with all records of B that arrived at most 15 minutes earlier or 1 hour later.  
Every record of B is joined with all records of A that arrived at most 1 hour earlier or 15 minutes later.



# Time-windowed Join: Use Case

- Find URLs that were clicked less than 5 seconds after being served as an ad

```
SELECT url
FROM clicks c, serves s
WHERE s.url = c.url AND
      s.user = c.user AND
      c.cTime BETWEEN s.sTime AND
                     s.sTime + INTERVAL '5' SECOND
```



Time attributes





# Time-windowed Join: Execution

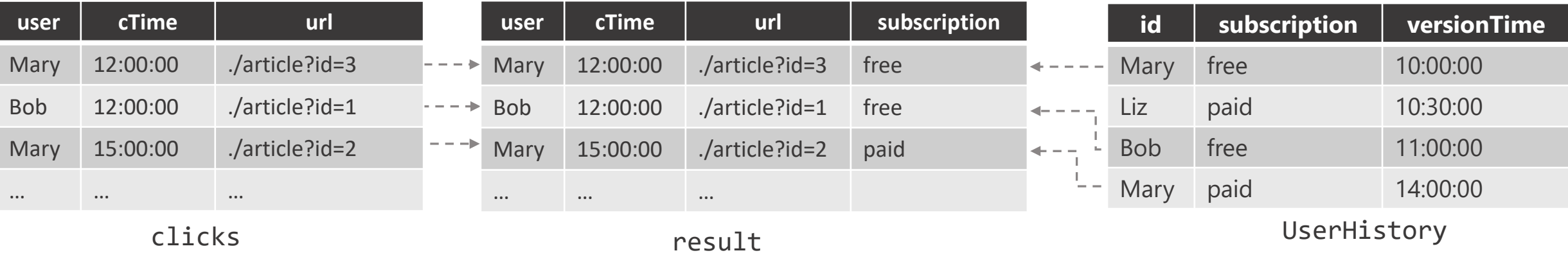
- The tails of both append-only tables that correspond to their active join-window are kept in state
- Rows are immediately removed from state once they cannot be joined anymore
- Event-time skew between tables increases state size



# Joins with Temporal Tables

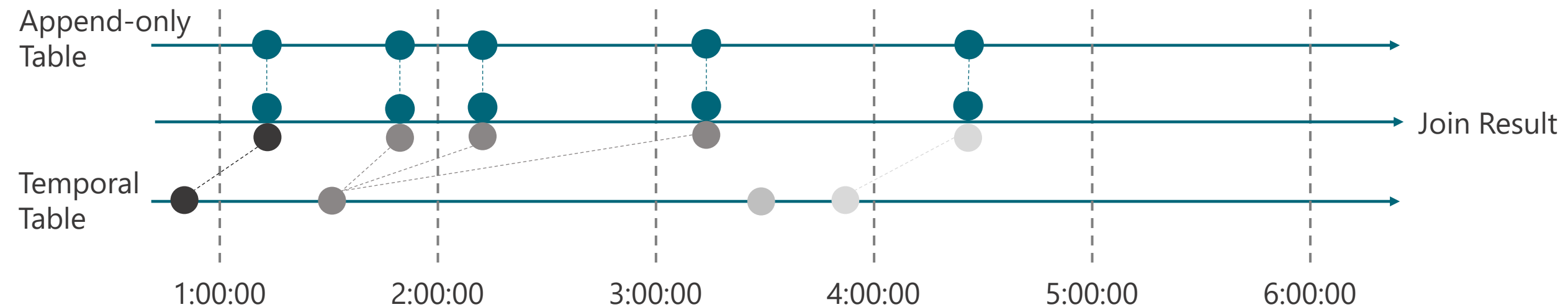
# Join with Temporal Table: Use Case

- Given
  - A table `clicks` with click events: `[URL, userId, clickTime]`
  - A table `userHistory` with user information `[id, subscription, versionTime]` that is continuously updated and stores the history of changes
- Enrich every click with the most recent version of the user information



# Join with Temporal Table

*A temporal table gives access to the history of a dynamic table.  
By joining with a temporal table, records of an append-only table can be joined with the version of the dynamic table that corresponds to their timestamp.*



Every record of the append-only table is joined with the latest version in the temporal table.



# Temporal Tables

- A temporal table gives access to a version of an append-only history table.
  - Temporal table has a unique key
  - Updates are versioned by timestamp

<u>Key</u>	VersionTime	Value
A	12:00:00	1
B	12:00:00	1
A	12:01:00	2
C	12:02:00	1
B	12:04:00	2
D	12:04:00	1
A	12:05:00	3
C	12:07:00	2
E	12:08:00	1

Append-only History Table

<u>Key</u>	VersionTime	Value
A	12:01:00	2
B	12:00:00	1
C	12:02:00	1

Temporal Table at  
12:03:00

<u>Key</u>	VersionTime	Value
A	12:05:00	3
B	12:04:00	2
C	12:02:00	1
D	12:04:00	1

Temporal Table at  
12:06:00



# Temporal Tables

- Temporal tables are defined as a built-in table-valued function in Flink SQL
- Defining the table-valued function requires:
  - Append-only history table
  - Declaration of unique key attribute
  - Declaration of time attribute (event or processing time) as version time

See documentation for details:

[https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/streaming/temporal tables.html](https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/streaming/temporal%20tables.html)



# Join with Temporal Table: Requirements and Syntax

- Table A is append-only (records cannot be updated!)
- Table B is a Temporal Table function and takes the time attribute of A as parameter
- An mandatory equality predicate on B's unique key.

```
SELECT *  
FROM A, LATERAL TABLE(B(A.t))  
WHERE A.id = B.id
```

For each row of table A, temporal table B returns the version that corresponds to the timestamp of the A row. The row of A is joined with the rows returned by B according to the join condition.



# Join with Temporal Table: Use Case

Enrich click data with the current version of user information.  
Users is a Temporal Table defined on UserHistory.

```
SELECT c.url, c.cTime, u.id, u.subscription
FROM
  clicks c,
  LATERAL TABLE(users(c.cTime)) u
WHERE c.user = u.id
```



Time attribute

Proper event-time support:

Correct semantics during failure recovery, backfill, etc!





# Join with Temporal Table: Execution

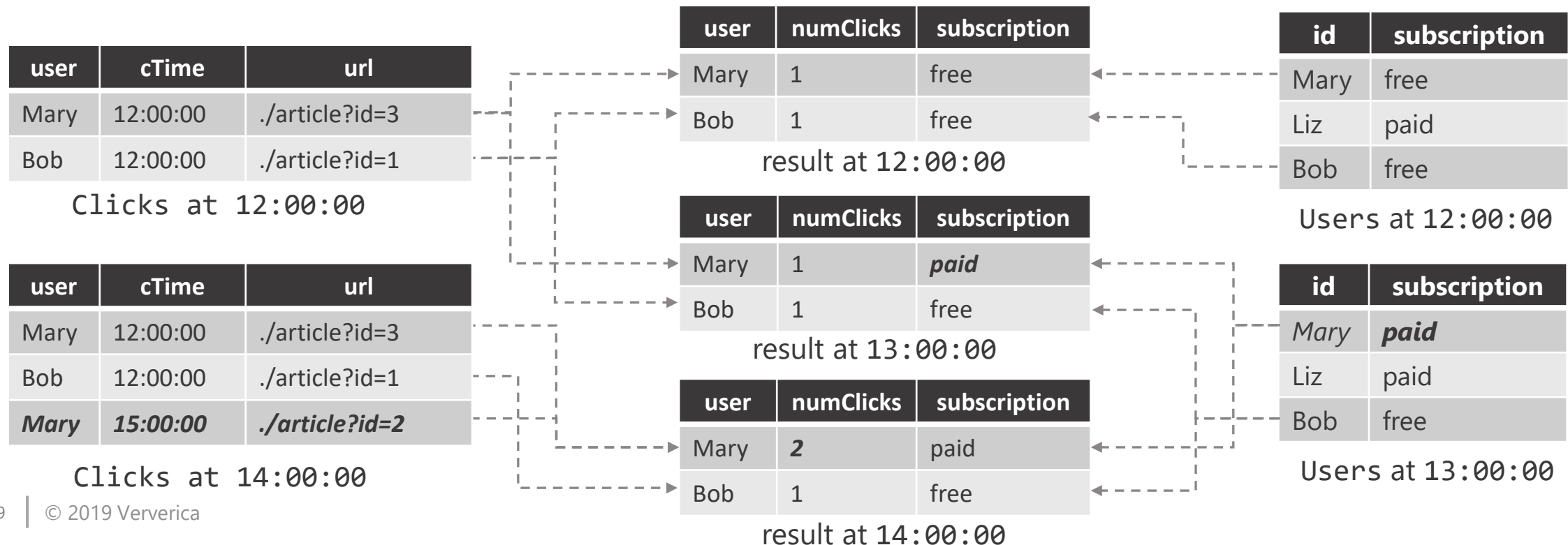
- Event time:
  - Temporal table:
    - For each unique key at least one row with the latest version is kept in state
    - Possibly more versions depending on event time skew
  - Append-only table:
    - Rows with timestamps later than the current event-time are temporarily buffered in state
- Processing time:
  - Temporal table:
    - For each unique key exactly one row with the latest version is kept in state
  - Append-only table:
    - All rows are immediately joined and not buffered
- Event-time skew between tables increases state size



# Regular / Default Joins

# Regular Join: Use Case

- Given
  - A table `clicks` with click events: [`URL`, `userId`, `clickTime`]
  - A table `users` with user information [`id`, `subscription`] that is continuously updated. The table does not store the history of changes.
- Keep track the number of clicks and the subscription status of all users



# Regular / Default Join

*Joins without a temporal join condition  
(i.e., joins that are neither time-windowed nor temporal table joins)  
are handled as regular joins. Input tables can be updating.*

The join result is updated as records of either input table are inserted, deleted, or updated.



# Regular Join: Requirements and Syntax

- Tables A and B should be updating (but can be append-only as well)
- The join condition includes at least one equality predicate

```
SELECT *  
FROM A, B  
WHERE A.id = B.id
```

All pairs of records of A and B that satisfy the join condition are joined.  
Records can be inserted into, deleted from, or updated in A and B.



# Regular Join: Use Case

- Keep track the number of clicks and the subscription status of all users

```
SELECT user, numClicks, subscription
FROM (SELECT user, COUNT(*) AS numClicks
      FROM clicks
      GROUP BY user) c,
Users u
WHERE c.user = u.id
```

GROUP BY subquery  
produces updating table

Users table is  
updating



# Regular Join: Execution

- The join operator fully materializes both input tables in state
  - A new record can join with any other record of the other table
- Regular joins only work well if both input tables are not growing too large
  - Table is only slowly growing (or not growing at all)
- Flink can be configured to remove rows that have not been used for x time.
  - Query result becomes inconsistent if state is discarded too early



# Attention!

- Regular joins cannot forward time attributes
  - The order in which result rows are emitted is not defined
  - Time attributes lose their orderliness / watermark alignment property
- Time attributes must be either projected out or casted to regular `TIMESTAMP`
- If you mess up the join condition for a time-windowed join, the query will be planned with a regular join.
  - You will get an exception about the time attribute in your result.





# Hands On Exercises

# Joining Dynamic Tables

Continue with the hands-on exercises in  
“Joining Dynamic Tables”

<https://github.com/ververica/sql-training/wiki/Joining-Dynamic-Tables>

We are here to help!





# ververica

---

[www.ververica.com](http://www.ververica.com)

@VervericaData