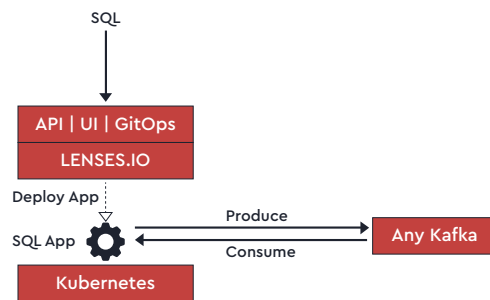


WHAT IS LENSES.IO?

Lenses.io delivers a powerful data operations workspace to build & operate real time applications on Kafka & Kubernetes. Deployed as a container or as a JVM, Lenses.io works with any Kafka & Kubernetes environment including managed and cloud services.

LENSES.IO STREAMING SQL

Streaming SQL allows anyone to build data processing applications with SQL. Queries react to data as soon as it is available in Apache Kafka. They can reshape your data, aggregate it based on any field or time window. Or enrich it with other streaming data. The results are pushed back to a Kafka topic so that downstream applications & processes can consume. Pipelines are deployed and scaled on your own Kubernetes environment.



STATEFUL AND STATELESS PROCESSING

Streams (Stateless) and tables (Stateful) have different semantics and use-cases, but are strongly related nonetheless.

This relationship is known as *stream-table* duality. Every stream can be interpreted as a table, and similarly a table can be interpreted as a stream.

STREAM (Stateless)	Represents an unbounded sequence of independent events over a continuously changing dataset. The dataset is the totality of all data described by every event received so far	SELECT STREAM * FROM website_traffic
TABLE (Stateful)	For each key, a table holds the latest version (state) received of its value. Upon receiving events for keys that already have an associated value, such values will be overridden and state changed	SELECT TABLE * FROM customers

STREAMING SQL EXAMPLE USECASES

Usecase	Example
Filtering	Filter payment events a value greater than 5000 in order to route them through an ML validation process.
Aggregating	Average the bandwidth per network device in a rolling 5 minute window.
Enriching	Combine the customer order information with the customer details.
Reshaping	Unwrap certain field values for a topic containing energy usage information into another topic so that it can be sent to a time-series database.
Re-key	Change the event key to suit a downstream consumer or align the topic for Kafka Streams joins/aggregation.
Reformatting	Translate incoming JSON data to AVRO for better control over schema evolution, rogue messages and so on.
Obfuscate the data	Apply redaction to various sensitive fields when moved to another topic.

SUPPORTED FORMATS

Lenses.io supports reading and writing data from/to Kafka topics in different serialization formats.

Format	Read	Write
INT	yes	yes
BYTES	yes	yes
LONG	yes	yes
STRING	yes	yes
JSON	yes	yes
AVRO	Via schema registry	Via schema registry
XML	yes	yes
CSV	yes	not yet
Google Protobuf	Via custom configuration	no
Custom	Via custom configuration	no
TW[<the_other_formats>]	yes	yes
SW[<the_other_formats>]	yes	yes

FIRST EXAMPLE

```
SET defaults.topic.autocreate=true;
```

```
INSERT INTO speeding_cars
SELECT STREAM car_speed AS speed
    , car_name
FROM car_data;
WHERE car_speed > 100
```

The above example will autocreate any necessary topics (in this case, "speeding_cars" topic) and populate it with the value of car_speed renamed as speed and car_name from the car_data topic only for events where the car_speed is greater than 100 .

KAFKA RECORD

Facet	Lenses.io reference	Example
Key	_key OR _key.<fieldname>	INSERT INTO outputTopic SELECT TABLE _key AS customerId FROM customers
Value	_value OR _value.<fieldname> OR <fieldname> OR *	INSERT INTO outputTopic SELECT TABLE _value.firstName FROM customers

PROJECTIONS

A projection represents the ability to reshape the data layout (Key or Value).

Projections are the main building block of **SELECT** statements

```
INSERT INTO target-topic
SELECT STREAM
    CONCAT('a', 'b') AS result1
    , (1 + field1) AS _key.a
    , _key.field2 AS result3
CASE
    WHEN field3 = 'Robert'
    THEN 'It's bobby'
    WHEN field3 = 'William'
    THEN 'It's willy'
    ELSE 'Unknown'
END AS who_is_it
FROM INPUT-topic;
```

In the above example, *result1*, *_key.a*, *result3* and *who_is_it* are all fields outputted into target-topic.

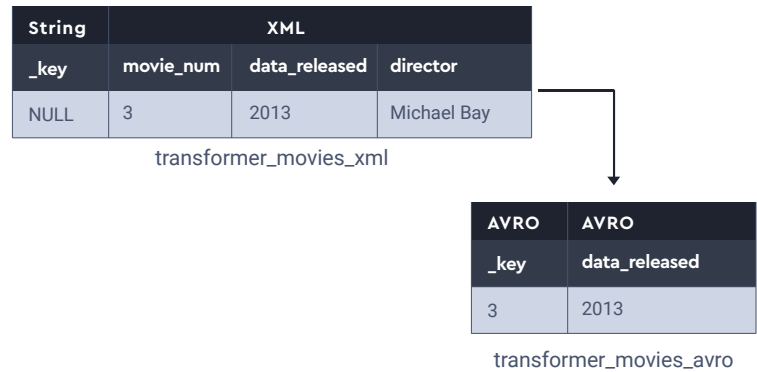
CONDITIONAL EXPRESSIONS

Type	Supported Operators	Example
Logical	AND, OR	WHERE 1 + _value.field1 > 5 AND _value.field2 < 10
Arithmetic	+, -, *, /, % (MOD)	WHERE 1 + _value.field1 > 5
Ordering	>, >=, <, <=	WHERE _value.field1 > _value.field2
Equality	=, !=	WHERE _key != LENGTH (_value.field1)
String	LIKE, NOT LIKE	WHERE _value.field1 NOT LIKE "%foo%"
Case	IN	CASE WHEN field3 = 'Robert' THEN 'Its bobby' WHEN field3 = 'William' THEN 'Its willy' ELSE 'Unknown' END AS who_is_it

EXAMPLE: RE-KEY & CONVERT TO AVRO

```
INSERT INTO transformer_movies_avro
STORE KEY AS AVRO VALUE AS AVRO
SELECT STREAM movie_number AS _key
, date_released
FROM transformer_movies_xml
WHERE movie_number > 0 AND
movie_number < 300
```

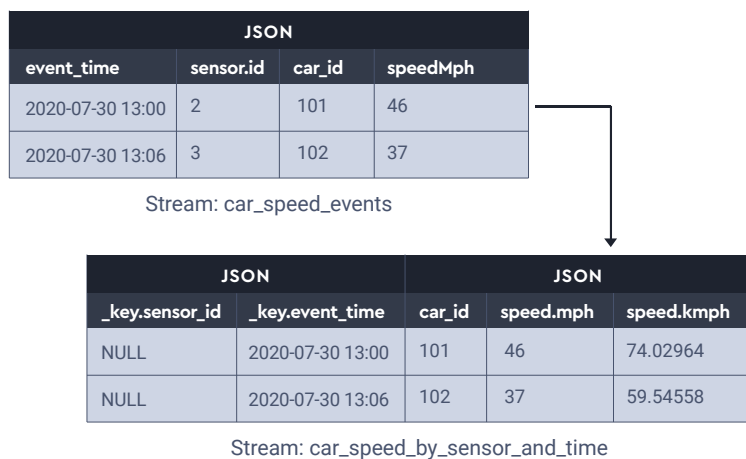
Select movie_number and date_released from transformer_movies_xml topic and put in as AVRO format in transformer_movies_avro topic. Make movie_number as the _key and only select movies where movie_number less than 300.



EXAMPLE: RESHAPE JSON

```
SELECT STREAM sensor.id AS _key.sensor_id
, event_time AS _key.event_time
, car_id
, speedMph AS speed.mph
, speedMph * 1.60934 AS speed.kmph
FROM car_speed_events;
```

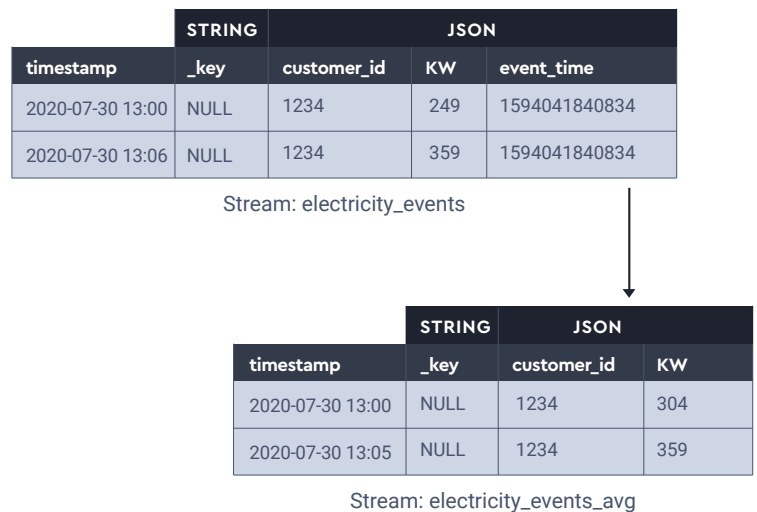
Reshape the car_speed_events stream by setting sensor_id and event_time as keys and nesting mph and a calculated kmph values for speed object in the value field.



EXAMPLE: AGGREGATE OVER TIME WINDOW

```
INSERT INTO electricity_events_avg
SELECT STREAM customer_id ,
AVG (KW) AS KW
FROM electricity_events
WINDOW BY HOP 10m, 5m
GROUP BY customer_id
```

Calculate the rolling average KW for each customer_id for every 10 minute period every 5 minutes. Populate into electricity_event_avg stream



EXAMPLE: JOIN STREAM WITH TABLE & RE-KEY

```
WITH ship_names_rekeyed AS
(SELECT STREAM ship_names.mmsi AS _key
 FROM ship_names);

WITH ship_names_state AS
(SELECT TABLE *
 FROM ship_names_rekeyed);

INSERT INTO ship_speeds_with_names
SELECT STREAM fast_vessel_processor._key AS mmsikey
, fast_vessel_processor.Speed
, ship_names.owner
, ship_names.name
FROM fast_vessel_processor
INNER JOIN ship_names_state ON fast_vessel_processor._key.MMSI
= CAST(ship_names_state._key AS LONG);
```

Populate the ship_speeds_with_names topic with the speed field from fast_vessel_processor topic joined with the ship_names topic using the MMSI value. The ship_names MMSI value is stored as a STRING whereas the fast_vessel_processor is stored as a LONG. The ship_names also doesn't have a key so needs to be rekeyed (ship_names_rekeyed) in order to build a state table (ship_names_state) since states cannot have NULL keys.

STRING	JSON		
_key	name	Owner	mmsi
NULL	The Portia	Isabelle Bray	12334

Stream: ship_names

Re-key

LONG	JSON		
_key	name	Owner	mmsi
1234	The Portia	Isabelle Bray	12334

Stream: ship_names_rekeyed

Generate State Table

LONG	JSON		
_key	name	Owner	mmsi
1234	The Portia	Isabelle Bray	12334

State: ship_names_rekeyed_state

LONG	JSON
_keyMMSI	speed
1234	3

Stream: fast_vessel_processor

JSON	JSON			
_key	mmsi	name	Owner	speed
1234	1234	The Portia	Isabelle Bray	3.6

Stream: ship_speed_with_names

CHANGING STORAGE FORMAT

At times, it is useful to control the resulting Key and/or Value storage of the output topic. If the input is Json, the output for the streaming computation can be set to Avro.

The syntax is the following:

```
INSERT INTO <target topic> STORE KEY AS <format> VALUE AS
<format> ...
```

From/To	INT	LONG	STRING	JSON	AVRO
INT	=	yes	yes	no	yes
LONG	no	=	yes	no	yes
STRING	no	no	=	no	yes
JSON	If the Json storage contains integer only	If the Json storage contains integer or long only	yes	=	yes
AVRO	If Avro storage contains integer only	If the Avro storage contains integer or long only	yes	yes	=

JOIN TYPES

Join Type	Description	Lexicon	Example
Inner	This join type will only emit records where a match has occurred.	JOIN	<pre>INSERT INTO Result SELECT STREAM customersTable.name , ordersStream.item FROM ordersStream JOIN customersTable ON customersTable.id = ordersStream.customer_id;</pre>
Left	Selects all the records from the left side of the join regardless of a match:	LEFT JOIN	<pre>INSERT INTO Result SELECT STREAM customersTable.name , ordersStream.item FROM ordersStream LEFT JOIN customersTable ON customersTable.id = ordersStream.customer_id;</pre>
Right	A mirror of a LEFT JOIN. It selects all the records from the right side of the join regardless of a match:	RIGHT JOIN	<pre>INSERT INTO Result SELECT TABLE customersTable.name , or- dersStream.item FROM customersTable RIGHT JOIN ordersStream ON customersTable.id = ordersStream.customer_id;</pre>
Outer	Union of left and right joins. It selects all records from the left and right side of the join regardless of a match happening:	OUTER	<pre>INSERT INTO Result SELECT TABLE custom- ersStream.name , or- dersStream.item FROM ordersStream OUTER JOIN customersStream ON customersTable.id = ordersStream.customer_id;</pre>

JOIN MATCH EXPRESSIONS

if no *ON* expression is provided, the join will be evaluated based on the equality of the *_key* facet

Equality	<code>customers.id = order.user_id</code> <code>customers.id - 1 = order.user_id - 1</code> <code>substr(customers.name, 5) = order.item</code>	INSERT INTO Result SELECT TABLE customersStream.name, ordersStream.item FROM ordersStream OUTER JOIN customersStream ON substr(customersTable.name, 5) = ordersStream.customerName;
Boolean	<code>len(customers.name) > 10</code> <code>substr(customer.name,1) = "J"</code> <code>len(customer.name) > 10 OR customer_key > 1</code>	INSERT INTO Result SELECT TABLE cars_table.cars_name, car_speeds.speed FROM cars_table OUTER JOIN car_speeds ON car_speeds.speed > 100
Logical	<code>customers._key = order.user_id AND len(customers.name) > 10</code> <code>len(customers.name) > 10 AND substr(customer.name,1) = "J"</code> <code>substr(customers.name, 5) = order.item AND len(customer.name) > 10 OR customer_key > 1</code>	INSERT INTO Result SELECT TABLE customersStream.name, ordersStream.customerId, FROM ordersStream OUTER JOIN customersStream ON customersTable.country = "USA" AND customersTable.id = ordersStream.customerId

JOIN COMPATIBILITY

Left	Right	Allowed Types	Window	Result
Stream	Stream	All	Required	Stream
Table	Table	All	no	Table
Table	Stream	RIGHT JOIN	no	Stream
Stream	Table	INNER, LEFT JOIN	no	Stream

STREAM-TO-STREAM WINDOWING WITH WITHIN

When two streams are joined Lenses needs to know how far away in the past and in the future to look for a matching record.

```

SELECT STREAM customers.name ,
        orders.item
FROM customers
LEFT JOIN orders WITHIN 5s ON customers.id = orders.customer_id WITHIN 5s;

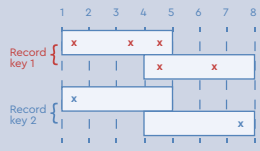
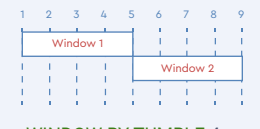

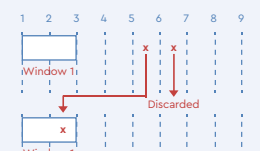
```

The above example means that an event with an orders.item may be generated with a null value for customers.name if a matching customers.id event has not been generated within a 5 second +/- window.

SUPPORTED TIME DESCRIPTORS

Duration	Description	Example
ms	time in milliseconds	100ms
s	time in seconds	10s
m	time in minutes	10m
h	time in hours	10h

SUPPORTED TIME DESCRIPTORS

Hopping Window	WINDOW BY HOP <duration_time>,<hop_interval> Fixed size and overlapping windows. The same event can overlap into multiple windows	 WINDOW BY HOP 4m, 3m
Tumbling Window	WINDOW BY TUMBLE <duration_time> duration and hop interval are equal. An event can only appear in one window.	 WINDOW BY TUMBLE 4m
	WINDOW BY SESSION <inactivity_interval> Defined by a period of activity separated by a specified gap of inactivity at which point current session closes	 WINDOW BY SESSION 3m
Grace period	WINDOW BY <windowing type> GRACE BY <grace time> For a window to accept late-arriving records a grace period can be defined. If a record falls within a window and it arrived before the end of the grace time defined then the record will be processed and the aggregations or joins will update. If not, the record is discarded	 WINDOW BY TUMBLE 4m GRACE BY 3m

Try out SQL on real-time data

Get Workspace