

Blog Post Author: Lioner Ferder

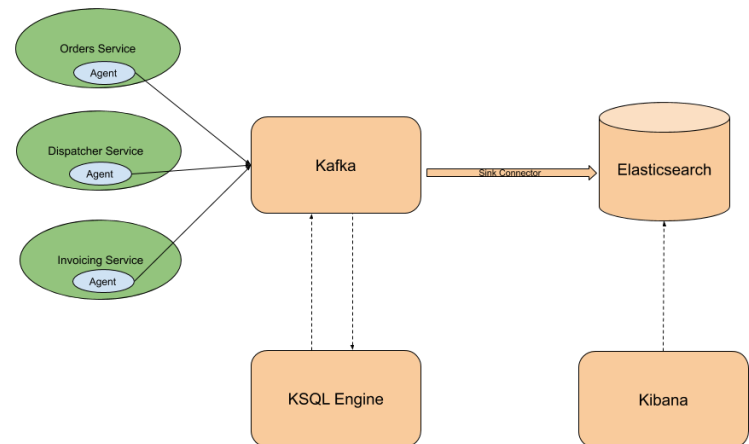
Real Time Business Statistics with Kafka, KSQL and Elasticsearch

One of our clients at Lemonade wanted to have the possibility to view real time business statistics on the usage of the software we've built. The software processes business events received from different actors and acts as a coordinator between them. A customer might be sending a purchase order which has to be forwarded to the different services, for instance warehouse, finance and to external providers before sending the confirmation to the customer. Our client wanted to count the number of business events occurring in a time period, for example the number of orders placed for different products broken down by the customer.

We decided to build this as an independent subsystem which receives and aggregates business events which later on are shown to the users in nice interactive Kibana boards.

Enter [KSQL](#), a SQL-inspired engine built on top of [Kafka Streams](#) facilitating the process of writing Stream applications by allowing to write them in a SQL-like syntax and executing them in the KSQL Engine.

The general process looks like this:



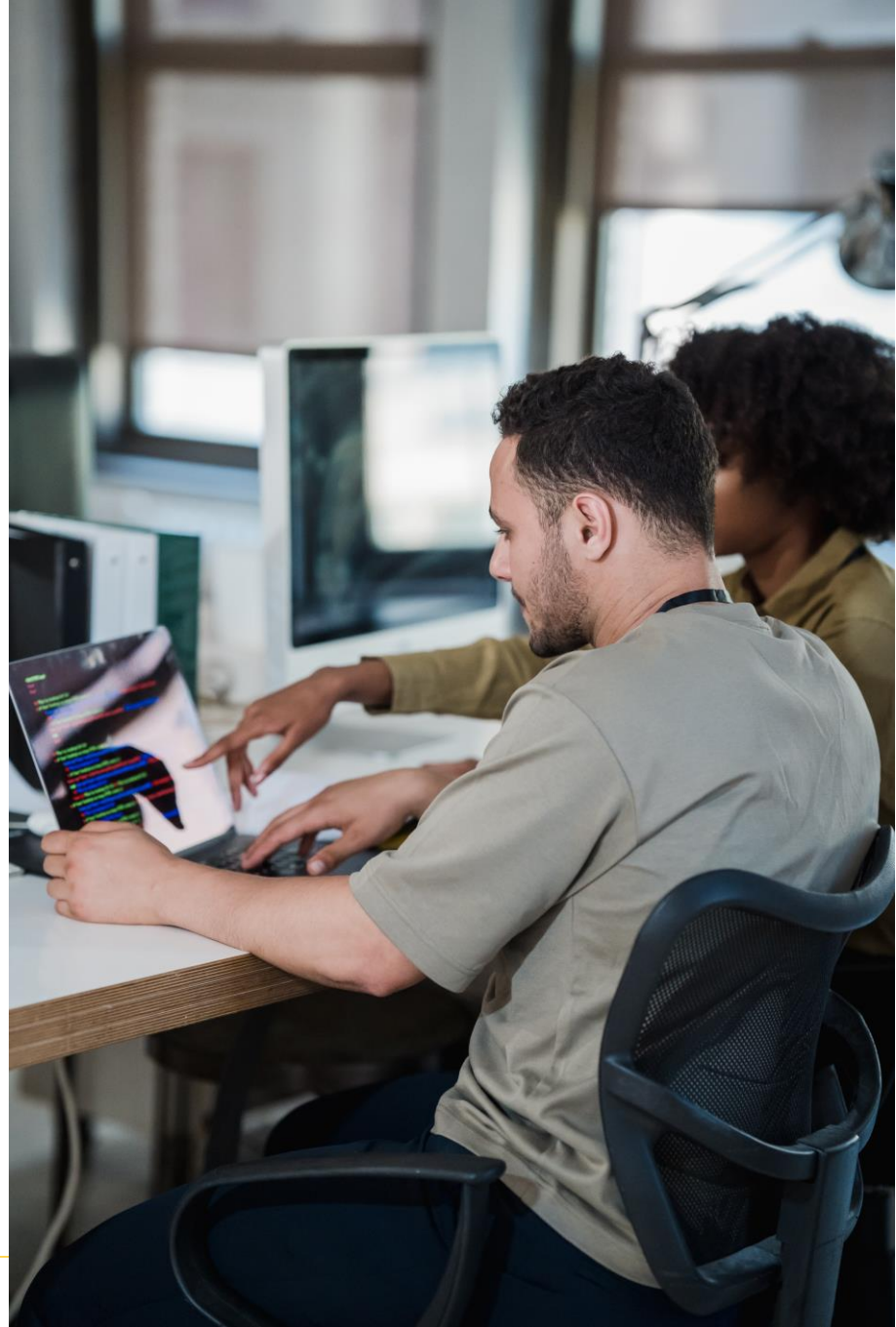
When a business event occurs in our services, it is forwarded to a Kafka topic which is aggregated by a continuous KSQL query that calculates a specific statistic in a time window and forwards it to a separate topic. This statistic topic content is then sent to Elasticsearch by leveraging the Kafka Connect Elasticsearch Sink Connector where the data is persisted and is available for Kibana to consult and visualize.

Getting Messages to Kafka

First step was to write an agent to intercept the events, which was added to the different services handling the events. This interceptor extracts information from the events and acts as a Kafka Producer sending the pertinent information to a Kafka Topic. We decided to follow Confluent's recommendation and use Avro for serializing the data which has a smaller footprint than json given that the schema is not stored within the message but in the Schema Registry. This means that an instance of the Schema Registry must be available for the Kafka Producer, KSQL Engine and the Elasticsearch Sink Connector.

For this article, an example application is used for demonstrating the different technologies. A simple producer is used for publishing the records to two topics in Kafka: `purchase_orders` and `delivery_notes`, here is what the avro messages look like in the topic:

```
developer@~:~$ kafka-console-consumer --topic purchase_orders --bootstrap-server 192.168.56.12:9092
```



The avro definition of the purchase order is the following:

```
{
  "namespace": "be.lemonade.ksqldemo.avro",
  "type": "record",
  "doc": "An incoming purchase order",
  "name": "PurchaseOrder",
  "fields": [
    {"name": "order_id", "type": "long"},
    {"name": "created_on", "type": "long", "logicalType":
      "timestamp-millis"},
    {"name": "client_name", "type": ["null", "string"]},
    {"name": "product_name", "type": ["null", "string"]}
  ]
}
```

While the delivery_note is:

```
{
  "namespace": "be.lemonade.ksqldemo.avro",
  "type": "record",
  "doc": "A delivery note",
  "name": "DeliveryNote",
  "fields": [
    {"name": "order_id", "type": "long"},
    {"name": "created_on", "type": "long", "logicalType":
      "timestamp-millis"},
    {"name": "address", "type": ["null", "string"]}
  ]
}
```

Processing the data in KSQLAs you can see, the avro output is not very readable when looking at the topic directly, luckily things look much nicer in KSQL.

Once our topics are populated we can create a couple of queries to see the purchase orders and delivery notes. KSQL allows defining both streams and tables. While streams are a continuous flow of events where all updates are registered (think of a database transaction log), a table will just contain the current value for a specific record id.

I'll create first two streams, one per topic and then a simple select to show the content of the purchase orders stream:



Few things to note here: first, when creating the stream we need to specify from which kafka topic the stream will be reading and the format of the records, avro in our case. Second, unlike a regular select query executed in a relational database where the results are returned once the operation finishes, a select in KSQL will keep running until interrupted, showing the results as they come.

Now, let's say we have a warehouse that only delivers chairs and is not interested in receiving events where other products are purchased. In this case we could create a new topic called purchase_orders_chairs and have the chairs warehouse service to consume from this topic:

```

ksql> CREATE TABLE handling_statistics WITH (KAFKA_TOPIC = 'handling process statistics', PARTITIONS=1, VALUE FORMAT = 'AVRO') AS
SELECT
  WindowStart() AS "DATE",
  product_name,
  SUM(delivery_notes.created_on - purchase_orders.created_on)/COUNT(*) as AVERAGE,
  MIN(delivery_notes.created_on - purchase_orders.created_on) AS MIN,
  MAX(delivery_notes.created_on - purchase_orders.created_on) AS MAX,
  COUNT(*) AS count
FROM delivery_notes
INNER JOIN purchase_orders WITHIN 1 HOUR ON purchase_orders.order_id = delivery_notes.order_id
WINDOW TUMBLING (SIZE 5 MINUTES)
GROUP BY product_name;

```

In this case we need to create a persistent query:

```

CREATE STREAM purchase_orders_chairs WITH(KAFKA_TOPIC='purchase_orders_chairs', PARTITIONS=1, VALUE_FORMAT='AVRO')
AS SELECT * FROM purchase_orders WHERE product_name = 'Chair';

```

In here we specify the name of the kafka topic where the result will be written to, the format and the actual select query with the filter we are interested in, in this case the purchase orders of chairs. It is important to note that this query will keep running in the background even if I log out from the KSQL console. As long as the KSQL Engine is running it will be processing the events.

Going back to our main focus of creating statistics, lets create a simple one where we count the number of products ordered per minute broken down by client:

```

CREATE TABLE purchase_orders_statistics WITH (KAFKA_TOPIC = 'purchase_orders_statistics', PARTITIONS=1, VALUE_FORMAT =
'AVRO') AS
SELECT WindowStart() AS "DATE", client_name, product_name, COUNT(*) AS count
FROM purchase_orders
WINDOW TUMBLING (SIZE 1 MINUTES)
GROUP BY client_name, product_name;

```

Note that here we are creating a table instead of a stream since we are just interested in the latest result for a specific group (where a group is the combination of window time – client_name – product_name). The results of this query will be stored in a topic called purchase_orders_statistics and the calculation will be done per minute (specified in the window tumbling clause).

```
ksql>
developer@\:~$ kafka-console-consumer --topic purchase_orders_chairs --bootstrap
-server 192.168.56.12:9092
```

Now that we have our three statistics calculated and stored in kafka topics, it is time to visualize the data.

Connecting Kafka with Elasticsearch

The Kafka Connect Elasticsearch Sink Connector provided with the Confluent platform is a Kafka consumer which listens to one or more topics and upon new records sends them to Elasticsearch. In our case we will configure it to listen to the statistics topics so the results of the KQL statistics queries are indexed in Elastic.

Here is the configuration for the connector:

```
name=ksqldemo-elasticsearch-sink
connector.class=io.confluent.connect.elasticsearch.ElasticsearchSinkConnector
tasks.max=1
topics.regex=delivery_notes_statistics|purchase_orders_statistics|handling_process
key.ignore=true
connection.url=http://192.168.56.12:9200
type.name=kafka-connect
schema.ignore=true
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter.schemas.enable=false
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://192.168.56.12:8081
```

Some important properties:

topics.regex contains a regex of the topic names we want to send to Elastic.

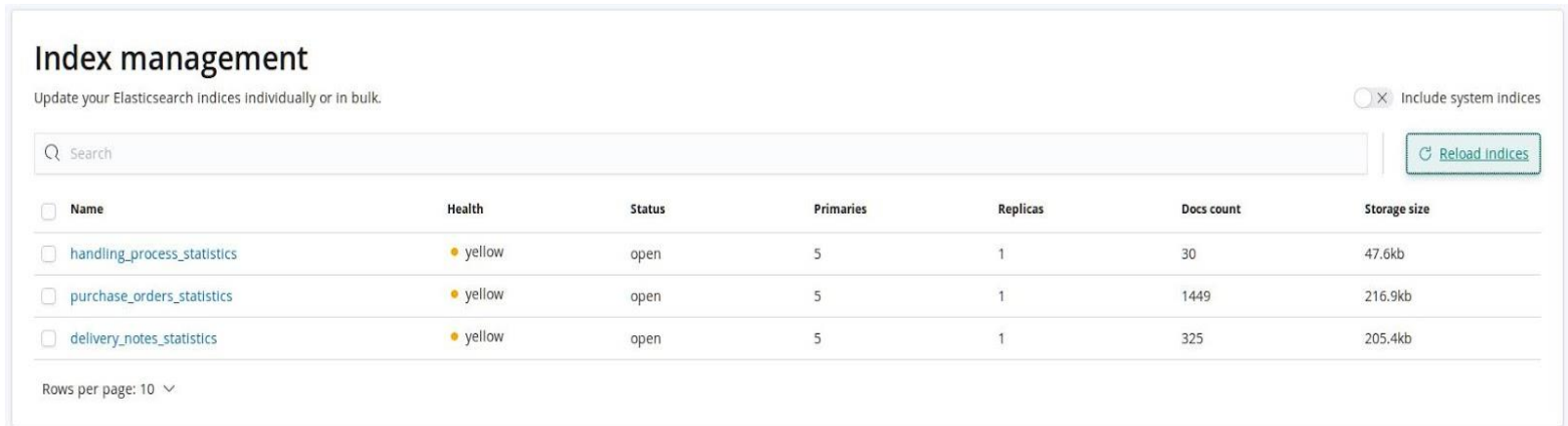
key.ignore when false elastic will use as the document id the same key as the record in kafka, otherwise a new one will be generated. We set it to true since we want all the results of the same aggregation to end up in the same document (since we are just interested in the final one).

schema.ignore: ignore the schema during indexing.

connection.url is the URL where elastic is running.

Key and value converter: since the key is a string and the value is avro we want to use the appropriate connectors. Given that avro is used, the URL of the Schema Registry has to be provided.

Once the connector is up and running the indices will be created. We could check it in Kibana:



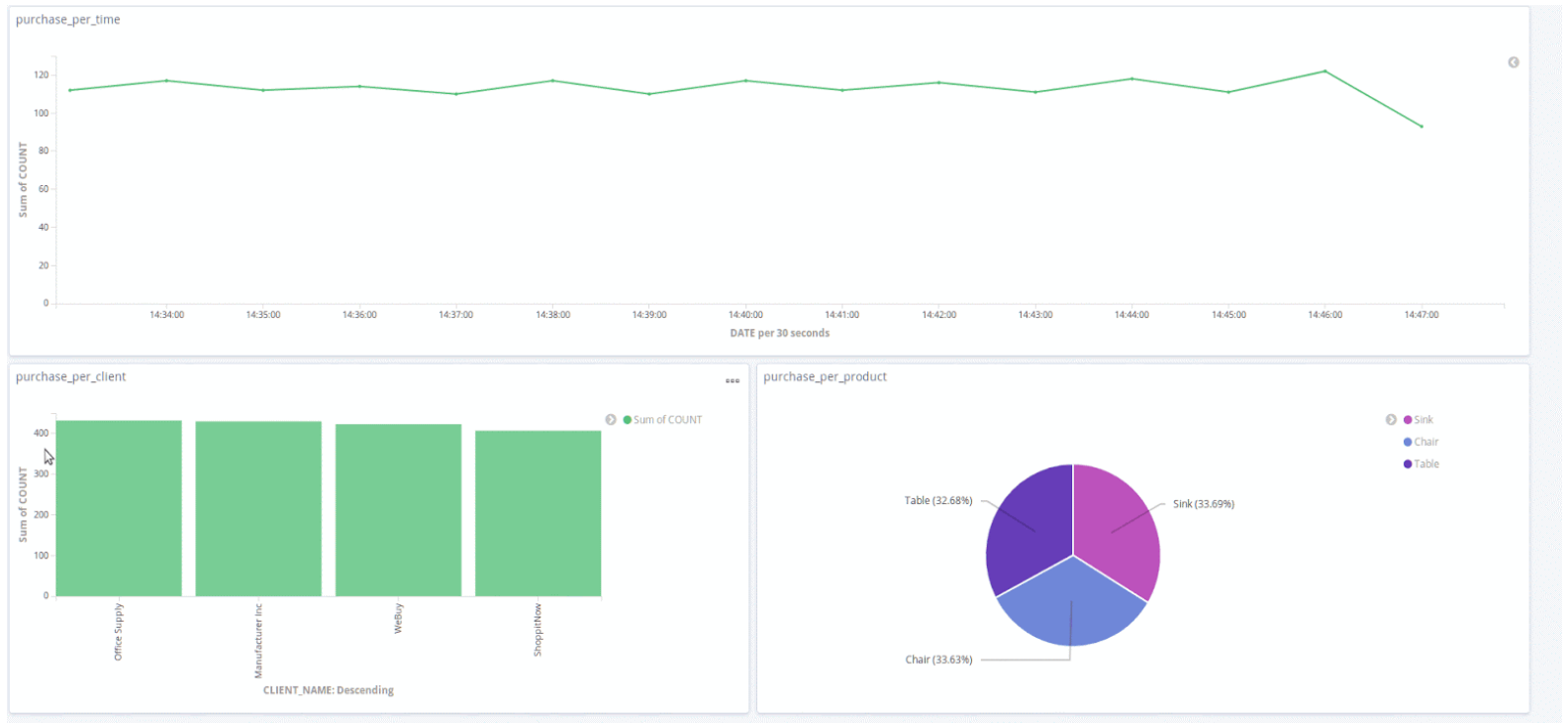
The screenshot shows the 'Index management' interface in Kibana. At the top, there is a search bar and a toggle for 'Include system indices'. Below this is a table with the following columns: Name, Health, Status, Primaries, Replicas, Docs count, and Storage size. The table lists three indices: 'handling_process_statistics', 'purchase_orders_statistics', and 'delivery_notes_statistics'. All three indices have a 'yellow' health status and are 'open'. The 'purchase_orders_statistics' index has the highest document count at 1449. At the bottom left, there is a dropdown menu for 'Rows per page' set to 10.

<input type="checkbox"/>	Name	Health	Status	Primaries	Replicas	Docs count	Storage size
<input type="checkbox"/>	handling_process_statistics	● yellow	open	5	1	30	47.6kb
<input type="checkbox"/>	purchase_orders_statistics	● yellow	open	5	1	1449	216.9kb
<input type="checkbox"/>	delivery_notes_statistics	● yellow	open	5	1	325	205.4kb

Now the only thing left to do is creating the Kibana dashboards.

Visualizing the data

The final step of the setup is to choose the right visualizations for the data. Given that each statistic allowed breaking down the data by multiple fields, we ended up creating one dashboard with multiple visualizations for each statistic. For example, the purchase order statistics is broken down both by product and client, so a dashboard might look like this:





Final thoughts

Kafka, KSQL, Elasticsearch and Kibana is a powerful combination and play nicely together. Writing the queries in KSQL is an easy and familiar way to process streams, but unfortunately it was not enough to cover all of our use cases and we had to refer to writing some Kafka Streams applications. Our problem in particular was the lack of possibility to explode arrays, for which we had to create a Kafka Streams application which explodes the data and stores the result in a topic which KSQL can later query. At the time of writing an [issue](#) exists to add this functionality.

Once the indexes were populated in Elasticsearch I was surprised how easy it was to create interactive dashboards containing multiple visualizations.

The final solution contained about 60 queries for calculating 50 statistics. Those queries were running in a cluster of two KSQL Engines processing around 150 events per second, enough to satisfy our requirements.