decodable

Use Case Walkthroughs for Real-Time Stream Processing

Executive Summary

Businesses are leveraging stream processing to make smarter and faster business decisions, act on time-sensitive and mission-critical data, obtain real-time analytics and insights, and build applications with features delivered to end-users in real time.

Decodable's mission is to make streaming data engineering easy. Decodable delivers the first real-time data engineering service—that anyone can run. As a platform for real-time data ingestion, integration, analysis, and event driven service development, Decodable eliminates the need for a large data team, clusters to set up, or complex code to write. Decodable will provide its customers with:

- The ease of use, adoption, and ecosystem of Snowflake
- The adoption model of DataDog, GitHub, AWS, GCP, Intercom, and Auth0
- The user experience sophistication of Superhuman and Intercom
- The developer love of dbt, pandas, Kubernetes, and Visual Studio Code

This eBook explores five common use case scenarios, with **additional examples available on our website.**

Customer 360 refers to getting a single view of customer engagement across the entire customer journey. It connects apps and data sources from customer interactions to give businesses a 360-degree customer view. It includes customer data from a variety of sources, including customer demographics, customer relationship management (CRM), social media, eCommerce, marketing, sales, customer service, mobile apps, and many other customer touchpoints.

Food Delivery Tracking explores the dramatic changes that have permeated how the world eats. Twenty years ago, restaurant-quality meal delivery was largely limited to pizza. Today, food delivery has become a global market worth more than $150 billion, having more than tripled in the last 5 years. The advent of appealing, user-friendly apps and tech-enabled driver networks, coupled with changing consumer expectations, has unlocked ready-to-eat food delivery as a major category.

Fraud Detection looks at securing online applications and services, which is a major requirement for businesses of all types. One example of this is mobile device emulators being used to spoof devices and mimic user behavior in an attempt to take over legitimate user accounts. With telemetry data such as accelerator and gyroscope signals, it is possible to train machine learning models to identify fraudulent activity and detect automated bots.

Shipping Logistics Tracking helps enable efficient and economical long-distance transport, which puts it at the center of the world economy. The ability to see, in real-time, logistics and tracking information helps facilitate better transportation decisions leading to reduced costs and enhanced services, which plays a key role in improving the customer experience as well as increasing profitability. Being able to offer order tracking provides customers with peace of mind, can win over hesitant buyers, and can even build customer loyalty.

Website Clickstream Analytics unlocks user activity on the web, and helps provide insight into how visitors get to the website, what they do once there, how long they stay on any given page, the number of page visits visitors make, and the number of unique and repeat visitors. Clickstream analytics have the ability to refine data by processing, cleaning, and transforming the raw data into convenient structures that make analysis of data easy and more accurate.

Creating a Pipeline

Decodable uses SQL to process data, something that will feel familiar to anyone who has used relational database systems. The primary differences you'll notice are that:

- You activate a pipeline to start it, and deactivate a pipeline to stop it.
- All pipeline queries specify a source, which identifies where the input data stream is coming from, and a sink, or where the processed data stream is going to.
- Certain operations, notably JOINs and aggregations, must include window functions.

Transform, Enrich, Aggregate, and More

For these examples, one or more processing pipelines are used to process the raw incoming data into the desired form. While a single pipeline can often be used, it is also possible to use multiple pipelines in a series of stages, with the output of each one being used as the input for the next. This results in pipelines that are easier to test and maintain. Each stage in the sequence of pipelines is used to bring the data closer to its final desired form using SQL queries.

Send to External System

Implicit for each of the use cases being explored, a sink **connection**–one that writes a stream to an external system, such as AWS S3, Kafka, Kinesis, Postgres, Pulsar, or Redpanda–would be created to allow the results to be consumed by your own applications and services.

Customer 360

Customer 360 refers to getting a single view of customer engagement across the entire customer journey. It connects apps and data sources from customer interactions to give businesses a 360-degree customer view. It includes customer data from a variety of sources, including customer demographics, customer relationship management (CRM), social media, eCommerce, marketing, sales, customer service, mobile apps, and many other customer touchpoints.

Businesses can leverage insights gained from a comprehensive customer view to improve and deliver exceptional experiences, increase customer loyalty, create reliable customer profiles to improve marketing and sales initiatives, streamline and connect business processes and workflows to improve efficiency and functionality, and reduce time and cost caused by human error in the customer journey.

In this example, we'll walk through how the Decodable data service is used to clean, transform, and aggregate data from multiple data sources.

Pipeline Architecture

Customer 360 data comes in many forms from many sources, including call logs, clickstream data, ecommerce activity, geolocation, NPS systems, and social media feeds. For this example, we will look at transforming two different data sources into a consistent schema which can then be sent to the same sink **connection** to be used for analysis, regardless of the original source or form of the data.

Below we can see examples of raw call log and clickstream data. Each data source is in a unique data format and uses different field names for similar data. By using one or more Decodable **pipelines**, which are streaming SQL queries that process data, we can transform the raw data into a form that is best suited for how it will be consumed.

Call Log Records


```
{
  "log_datatime": "2020-03-04 13:19:22",
  "xml": "<call_log><activity_id>701C</activity_id><call_id>367e5d7e-
a3e6-4d27a5c7-35706e9dca9d<call_id><user_id>4433a94b-12c5-4397-8837-3eedf11e78e6</user_id><start_time>2020-03-04
13:15:12<start_time><end_time>2020-03-04 13:19:22<
end_time><call_time_seconds>207<call_time_seconds><from_phone_number+37277774841</
from_phone_number><to_phone_number>+37249234343</to_phone_number><outcome>answered</outcome><has_recording>false</
has_recording></call_log>"
}
```


Clickstream Log Records


```
{
  "event_timestamp": "2020-11-16 14:32:19",
  "user_id":
"4433a94b-12c5-4397-8837-3eedf11e78e6",
  "site_id": "wj32-gao1-4w1o-iqp4",
  "pages_visited": 8,
  "total_seconds_on_site": 426,
  "avg_percent_viewed": 28.198543
}
```

Transform call logs

As with most data services pipelines, the first step is to apply a variety of transformations to clean up and simplify the input data. For this example, an inner `select` is used to parse the XML object blob using the xpaths function and extract the desired fields. Then the `start_time` field is converted from a `string` to a `timestamp` type and the `call_time_seconds` field is converted to an integer.

After creating a new pipeline and entering the SQL query, clicking the `Run Preview` button will verify its syntax and then fire up a new executable environment to process the next 10 records coming in from the source stream and display the results.

```
insert into transformed
select
  call_log.user_id as user_id,
  to_timestamp(call_log.start_time) as engagement_datetime,
  'sales call' as engagement_type,
  call_log.call_id as engagement_source_id,
  cast(call_log.call_time_seconds as int) as engagement_seconds
from (
  select
    -- parse XML to a DOM and extract fields using XPath
    expressions
    xpaths(xml,
      'user_id', '//call_log/user_id',
      'start_time', '//call_log/start_time',
      'call_id', '//call_log/call_id',
      'call_time_seconds', '//call_log/call_time_seconds'
    ) as call_log
  from `call_log`
)
```

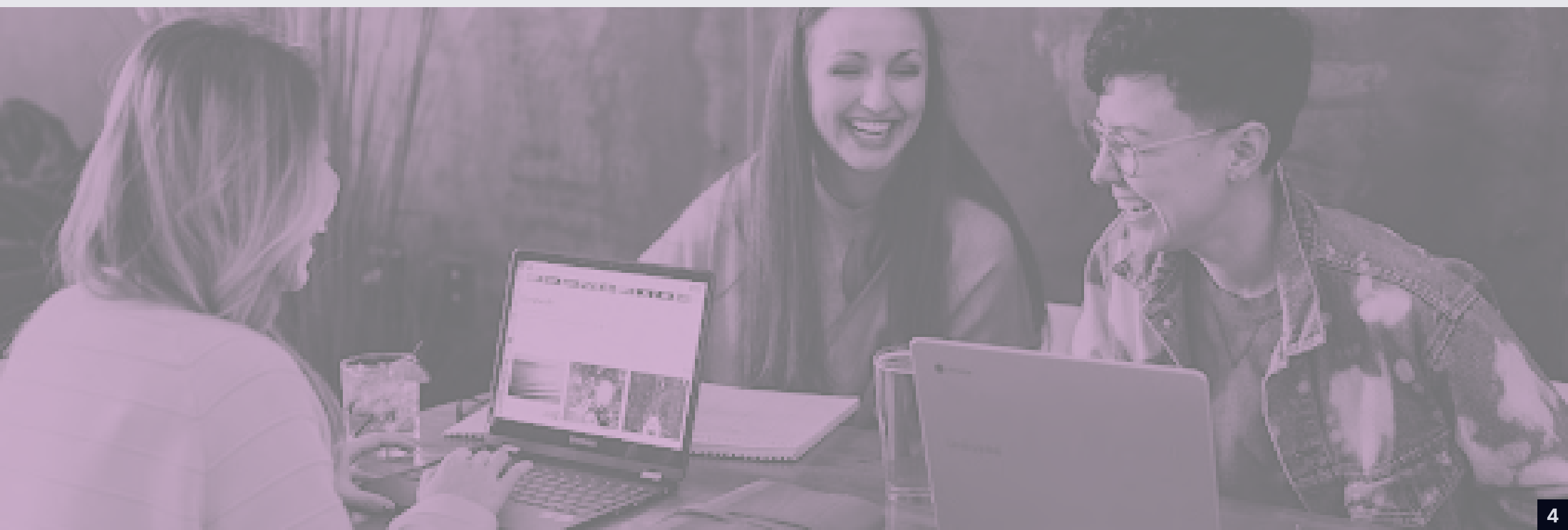
Transform clickstream

For the website clickstream data, the required transformations for this example are fairly minimal. Primarily the field names are changed to match the desired schema for a standardized data stream, and the `event_timestamp` field is converted to a `timestamp`.

```
insert into transformed
select
  user_id,
  to_timestamp(event_timestamp) as engagement_datetime,
  'website' as engagement_type,
  site_id as engagement_source_id,
  total_seconds_on_site as engagement_seconds
from `clickstream`
```

Decodable handles all the heavy lifting on the backend, allowing you to focus on working directly with your data streams to ensure that you are getting the results you need.

```
{
  "user_id": "4433a94b-12c5-4397-8837-3eedf11e78e6",
  "engagement_datetime": "2020-11-16 22:59:59",
  "engagement_type": "website",
  "engagement_source_id": "wj32-gao1-4w1o-iqp4",
  "engagement_seconds": 426
}
```



Food Delivery Tracking

How the world eats is changing dramatically. Twenty years ago, restaurant-quality meal delivery was largely limited to pizza. Today, food delivery has become a global market worth more than $150 billion, having more than tripled in the last 5 years. The advent of appealing, user-friendly apps and tech-enabled driver networks, coupled with changing consumer expectations, has unlocked ready-to-eat food delivery as a major category.

In this example, we'll walk through how the Decodable data service is used to clean, transform, and enrich real-time food delivery data. The processed data can then be used to send customers SMS text messages with progress status updates.

Pipeline Architecture

Below we can see a sample of raw food delivery data. For this example, the source of the data is a legacy system that produces an XML object.


```
{
  "updated_at": "2022-01-01 10:53:11",
  "xml": "<order id=\"629812004\" state=\"8\" state_human_readable=\"dispatched\" tracking_url=\"https://example-app.com/
order/9f0Ljd9g3\"><branch id=\"382102714\"><location><latitude>37.869169</latitude><longitude>-122.206648</longitude></
location></branch><customer id=\"782312311\"><location><latitude>37.874173</latitude><longitude>-122.220741</longitude></
location></customer><driver id=\"429178231\"><location><latitude>37.8924376</latitude><longitude>-122.216880</
longitude></location></driver><timestamps><ordered_at>2022-01-01 10:35:00</ordered_at><pending_at>2022-01-01 10:37:00</
pending_at><vendor_accepted_at>2022-01-01 10:39:00</vendor_accepted_at><driver_accepted_at>2022-01-01 10:44:00</
driver_accepted_at><dispatched_at>2022-01-01 10:49:00</dispatched_at><completed_at/><cancelled_at/></
timestamps><estimations><earliest_completed_at>2022-01-01 10:59:00</earliest_completed_at><latest_completed_at>2022-01-01
11:05:00</latest_completed_at></estimations><message>&lt;span style=&quot;color: red;&quot;&gt;Sorry, your order is
running a little late.&lt;/span&gt;</message><notification>Sorry, your order is running a little late.</notification></
order>"
}
```


Even when the XML is examined in its structured form, it is far more complex and detailed than what customers want to know about their delivery and what is needed to update them with status messages. By using one or more Decodable **pipelines**, which are streaming SQL queries that process data, we can transform the raw data into a form that is best suited for how it will be consumed.


```
<order id="629812004" state="8" state_human_readable="dispatched" tracking_url="https://example-app.com/order/9f0Ljd9g3">
  <branch id="382102714">
    <location>
      <latitude>37.869169</latitude>
      <longitude>-122.206648</longitude>
    </location>
  </branch>
  <customer id="782312311">
    <location>
      <latitude>37.874173</latitude>
      <longitude>-122.220741</longitude>
    </location>
  </customer>
  <driver id="429178231">
    <location>
      <latitude>37.8924376</latitude>
      <longitude>-122.216880</longitude>
    </location>
  </driver>
  <timestamps>
    <ordered_at>2022-01-01 10:35:00</ordered_at>
    <pending_at>2022-01-01 10:37:00</pending_at>
    <vendor_accepted_at>2022-01-01 10:39:00</vendor_accepted_at>
    <driver_accepted_at>2022-01-01 10:44:00</driver_accepted_at>
    <dispatched_at>2022-01-01 10:49:00</dispatched_at>
    <completed_at/>
    <cancelled_at/>
  </timestamps>
  <estimations>
    <earliest_completed_at>2022-01-01 10:59:00</earliest_completed_at>
    <latest_completed_at>2022-01-01 11:05:00</latest_completed_at>
  </estimations>
  <message>&lt;span style=&quot;color: red;&quot;&gt;Sorry, your order is running a little late.&lt;/span&gt;</message>
  <notification>Sorry, your order is running a little late.</notification>
</order>
```

Parse XML object

As with most data services pipelines, the first step is to apply a variety of transformations to clean up and simplify the input data. For this example, the first pipeline is used to parse and restructure the raw data as follows:

- parse the XML object blob using the **xpaths** function and extract the desired fields
- numeric fields will be converted from `string` to integers or floats as needed
- the time-based fields will be converted from `string` to `timestamp` data types, which enables more sophisticated processing in subsequent pipelines

```
insert into parsed
select
  cast(delivery_update.order_id as bigint) as order_id,
  cast(delivery_update.branch_id as bigint) as branch_id,
  cast(delivery_update.customer_id as bigint) as customer_id,
  delivery_update.state_human_readable as state_human_readable,
  delivery_update.notification as notification,
  to_timestamp(delivery_update.earliest) as earliest,
  to_timestamp(delivery_update.latest) as latest,
  to_timestamp(delivery_update.dispatched_at) as dispatched_at,
  to_timestamp(delivery_update.completed_at) as completed_at,
  cast(delivery_update.branch_lat as float) as branch_lat,
  cast(delivery_update.branch_lon as float) as branch_lon,
  cast(delivery_update.customer_lat as float) as customer_lat,
  cast(delivery_update.customer_lon as float) as customer_lon,
  cast(delivery_update.driver_lat as float) as driver_lat,
  cast(delivery_update.driver_lon as float) as driver_lon
from (
  select
    -- parse XML to a DOM and extract fields using XPath expressions
    xpaths(xml,
      'order_id', '//order/@id',
      'branch_id', '//order/branch/@id',
      'customer_id', '//order/customer/@id',
      'state_human_readable', '//order/@state_human_readable',
      'notification', '//order/notification',
      'earliest', '//order/estimations/earliest_completed_at',
      'latest', '//order/estimations/latest_completed_at',
      'dispatched_at', '//order/timestamps/dispatched_at',
      'completed_at', '//order/timestamps/completed_at',
      'branch_lat', '//order/branch/location/latitude',
      'branch_lon', '//order/branch/location/longitude',
      'customer_lat', '//order/customer/location/latitude',
      'customer_lon', '//order/customer/location/longitude',
      'driver_lat', '//order/driver/location/latitude',
      'driver_lon', '//order/driver/location/longitude'
    ) as delivery_update
  from `delivery-raw`
)
```

Enrich data stream

For this example, we want to enrich the data stream with the delivery driver's progress in reaching the customer destination. SQL provides a comprehensive set of powerful **functions**, such as `power` and `sqrt`, which can be leveraged to perform calculations that are useful for subsequent processing. This somewhat complex SQL query example could be broken down into two smaller, simpler queries; but it is also possible to create pipelines of arbitrary complexity based on your requirements.

An inner nested `select` query calculates distances using the latitude and longitude of the origination point (i.e., the branch location), the driver location, and the customer location. By using the SQL `case` statement, we can avoid performing the expensive haversine distance formula based on whether the driver has left the branch location or arrived at the customer location. By taking care to reduce the computational complexity of the pipeline, stream processing throughput can be increased.

Once the distances have been calculated, the surrounding query calculates a progress percentage based on how far the driver is from the customer and the overall distance between the branch and the customer.

```
insert into status
select
  *,
  floor((branch_distance - driver_distance) / branch_distance * 100)
  as  percent_complete
from (
  select
    order_id,
    branch_id,
    customer_id,
    state_human_readable,
    notification,
    earliest,
    latest,
    case
      when coalesce(dispatched_at, '') = '' then 1
      when coalesce(completed_at, '') <> '' then 1
      else 12742 * asin(sqrt(
             power(sin((branch_lat - customer_lat) * 0.008725), 2) +
               cos(customer_lat * 0.01745) *
               cos(branch_lat * 0.01745) *
               power(sin((branch_lon - customer_ lon) * 0.008725), 2)
           ))
    end as branch_distance,
    case
      when coalesce(timestamps.dispatched_at, '') = '' then 1
      when coalesce(timestamps.completed_at, '') <> '' then 0
      else 12742 * asin(sqrt(
             power(sin((driver_lat - customer_lat) * 0.008725), 2) +
               cos(customer_lat * 0.01745) *
               cos(driver_lat * 0.01745) *
               power(sin((driver_lon - customer_lon) * 0.008725), 2)
           ))
    end as driver_distance
  from parsed
)
```

After creating a new pipeline and entering the SQL query, clicking the `Run Preview` button will verify its syntax and then fire up a new executable environment to process the next 10 records coming in from the source stream and display the results. Decodable handles all the heavy lifting on the backend, allowing you to focus on working directly with your data streams to ensure that you are getting the results you need.


```
{
  "order_id": 629812004,
  "branch_id": 382102714,
  "customer_id": 782312311,
  "state_human_readable": "dispatched",
  "notification": "Sorry, your order is running a little late.",
  "earliest": "2022-01-01 10:59:00",
  "latest": "2022-01-01 11:05:00",
  "branch_distance": 11,
  "driver_distance": 8,
  "percent_complete": 27
}
```


Fraud Detection

Securing online applications and services is a major requirement for businesses of all types, and threat actors are constantly increasing the sophistication of their attacks. One example of this is mobile device emulators being used to spoof devices and mimic user behavior in an attempt to take over legitimate user accounts. With telemetry data such as accelerator and gyroscope signals, it is possible to train machine learning models to identify fraudulent activity and detect automated bots.

In this example, we'll walk through how the Decodable data service is used to clean, transform, enrich, and aggregate real-time telemetry data describing a device's accelerometer and touch screen interactions which is being sent from the Moonsense SDK. The processed data can then be sent onward to a fraud detection model for training or evaluation.

Pipeline Architecture

Here we can see a sample of the raw telemetry data from a typical user's device. In its current form, it is not suitable for use by a machine learning model. By using one or more Decodable pipelines, which are streaming SQL queries that process data, we can transform the raw data into a form that is best suited for how it will be used.


```
{
  "bundle": {
    "client_time": {
      "wall_time_millis": "1640122674127",
      "timer_millis": "137168",
      "timer_realtime_millis": "137168"
    },
    "pointer_data": [
      {
        "determined_at": "136893",
        "type": "TOUCH",
        "buttons": "1",
        "delta": {},
        "pos": {
          "dx": 27,
          "dy": 208
        },
        "pressure_range": {
          "upper_bound": 1
        },
        "radius_major": 40.472574869791664,
        "radius_minor": 40.472574869791664,
        "size": 14742.263849318027
      }
    ],
    "index": 1,
    "text_change_data": [
      {
        "determined_at": "136989",
        "target": {
          "target_id": "de8dfdd2-9121-401e-90a5-c9c2b8c2f9e4",
          "target_type": "range"
        },
        "masked_text": "d?"
      }
    ]
  },
  "app_id": "Wyk48mvsheCX4rg5d954tj",
  "credential_id": "a7RZotQQqYn7wbpei2hAhV",
  "session_id": "SjBbhxS88pngxDgzRYssma",
  "server_time_millis": "1640122674226208"
}
```


Clean the input data stream

As with most data services pipelines, the first step is to apply a variety of transformations to clean up and simplify the input data. For this example, the first pipeline is used to parse and restructure the raw data as follows

- the time fields will be converted from strings of integers representing epoch milliseconds to `timestamp` fields, which will enable more sophisticated processing in subsequent pipelines
- several fields that are nested inside the complex JSON source object will be elevated to simple top-level fields, which can then be more easily accessed in subsequent pipelines

- only the fields required by subsequent pipelines will be included in the output stream, filtering out extraneous fields and simplifying the data to be processed

After creating a new pipeline and copying in the SQL query, clicking the `Run Preview` button will verify its syntax and then fire up a new executable environment to process the next 10 records coming in from the source stream and display the results.

```
insert into moonsense_parsed
select
  app_id,
  session_id,
  user_id,
  to_timestamp_ltz(cast(left(server_time_millis, 13) as bigint), 3)
    as server_time,
  to_timestamp_ltz(cast(client_time.wall_time_millis as bigint), 3)
    as wall_time,
  cast(client_time.timer_millis as bigint)
    as timer_millis,
  cast(client_time.timer_realtime_millis as bigint)
    as timer_realtime_millis,
  bundle.location_data as location_data,
  bundle.accelerometer_data as accelerometer_data,
  bundle.magnetometer_data as magnetometer_data,
  bundle.gyroscope_data as gyroscope_data,
  bundle.battery as battery,
  bundle.activity_data as activity_data,
  bundle.orientation_data as orientation_data,
  bundle.pointer_data as pointer_data
from moonsense_raw
```

Unnest data stream array

To help detect non-human bot activity, the raw pointer data from the parsed telemetry data stream can be analyzed. In order to facilitate that, the `pointer_data` field, which contains an array of pointer positions, needs to be unnested (or demultiplexed) into multiple records. To accomplish this, a **cross join** is performed between the `moonsense_parsed` data stream and the results of using the unnest function on the `pointer_data` field.

For example, if a given input record contains an array of 5 pointer positions, this pipeline will transform each input record into 5 separate output records for processing by subsequent pipelines.

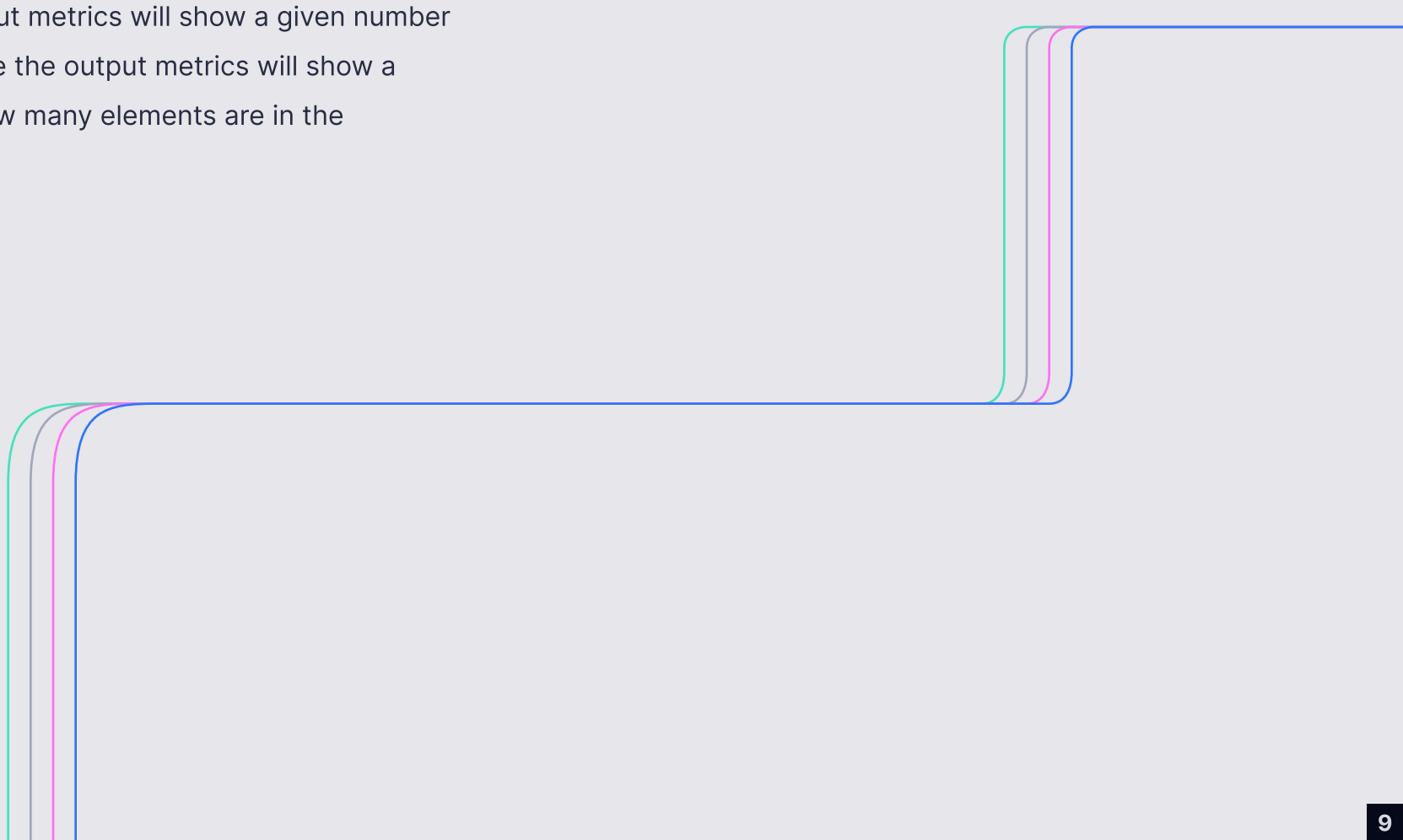
When the pipeline is running, the effects of unnesting the input records can be seen in the Overview tab which shows real-time data flow statistics. The input metrics will show a given number of records per second, while the output metrics will show a higher number based on how many elements are in the `pointer_data` array.

```
insert into moonsense_pointer_records
select

  -- each element of the `pointer_data` array creates a new record
  cast(pointer.determined_at as bigint) as determined_at,
  pointer.pos.dx as dx,
  pointer.pos.dy as dy,
  pointer.radius_major as radius,
  pointer.size as size,

  -- non-array fields common to each record are also included in
  the output
  app_id,
  session_id,
  user_id,
  server_time,
  wall_time,
  timer_millis,
  timer_realtime_millis

from moonsense_parsed
cross join unnest(pointer_data) as pointer
```



Enrich data stream

In the next stage of pipeline processing, we want to determine how quickly the pointer's position is changing. Because SQL provides a comprehensive set of powerful **functions,** such as `power` and `sqrt`, we can leverage these to enrich the data stream with the results of calculations that are more useful for subsequent processing.

This somewhat complex SQL query could be broken down into multiple smaller, simpler queries; but it is also possible to create pipelines of arbitrary complexity based on your requirements.

An inner nested `select` query is used to combine the change in pointer position data from the previous record with the current record using the `lag` **window function,** which provides access to a record at a specified physical offset which comes before the current record (in this case that is simply the previous record). A surrounding `select` query calculates the change in time and position between two consecutive pointer position records. Finally, the outermost `select` query calculates the pointer velocity and outputs that into a new data stream for processing by the next pipeline.

```
insert into moonsense_pointer_velocity
select
  session_id,
  server_time,
  determined_at,
  dx,
  dy,
  size,
  t_delta,
  d_delta,
  case
    when t_delta is null then 0
    when t_delta = 0 then 0
    else d_delta / t_delta
  end as velocity
from (
  select
    *,
    abs(determined_at - determined_at_prev) as t_delta,
    sqrt(power(dx - dx_prev, 2) + power(dy - dy_prev, 2)) as d_delta
  from (
    select
      *,
      lag(dx, 1) over (
        partition by session_id
        order by server_time
      ) as dx_prev,
      lag(dy, 1) over (
        partition by session_id
        order by server_time
      ) as dy_prev,
      lag(determined_at, 1) over (
        partition by session_id
        order by server_time
      ) as determined_at_prev
    from moonsense_pointer_records
  )
)
```

Enrich data stream

On this final pipeline stage, the data is aggregated into summary statistics that can then be fed into a detection model for training or evaluation. By leveraging the SQL `tumble` **group window function,** a data distribution matrix is created across a non-overlapping, continuous window with a fixed duration of 10 seconds. For each set of records, the number of pointer updates and totals for the interval are calculated.

For this example, we have focused only on the pointer position, but the original data stream contains a wealth of additional information, all of which can be proce

```
insert into moonsense_pointer_stats
select
  window_start,
  window_end,
  session_id,
  count(1) as record_count,
  -- # records received during tumble interval

  -- calculate a distribution matrix for the pointer velocities
  min(inst_velocity) as velocity_p0,
  approx_percentile(inst_velocity, 0.25) as velocity_p25,
  approx_percentile(inst_velocity, 0.5) as velocity_p50 ,
  approx_percentile(inst_velocity, 0.75) as velocity_p75,
  max(inst_velocity) as velocity_p100,

  avg(inst_velocity) as velocity_mean,
  sum(d_delta) as total_distance,
  sum(t_delta) as total_duration,
  sum(d_delta)/sum(t_delta) as total_velocity
from table (
    tumble(
      table moonsense_pointer_velocity,
      descriptor(server_time),
      interval '10' seconds
    )
  )
group by
  window_start,
  window_end,
  session_id
```

Aggregate data stream

In this final pipeline stage, the data is aggregated into summary statistics that can then be fed into a detection model for training or evaluation. By leveraging the SQL `tumble` **group window function**, a data distribution matrix is created across a non-overlapping, continuous window with a fixed duration of 10 seconds. For each set of records, the number of pointer updates and totals for the interval are calculated.

For this example, we have focused only on the pointer position, but the original data stream contains a wealth of additional information, all of which can be processed in a similar manner.

```
insert into moonsense_pointer_stats
select
  window_start,
  window_end,
  session_id,
  count(1) as record_count,
  -- # records received during tumble interval

  -- calculate a distribution matrix for the pointer velocities
  min(inst_velocity) as velocity_p0,
  approx_percentile(inst_velocity, 0.25) as velocity_p25,
  approx_percentile(inst_velocity, 0.5) as velocity_p50 ,
  approx_percentile(inst_velocity, 0.75) as velocity_p75,
  max(inst_velocity) as velocity_p100,

  avg(inst_velocity) as velocity_mean,
  sum(d_delta) as total_distance,
  sum(t_delta) as total_duration,
  sum(d_delta)/sum(t_delta) as total_velocity
from table (
    tumble(
      table moonsense_pointer_velocity,
      descriptor(server_time),
      interval '10' seconds
    )
  )
group by
  window_start,
  window_end,
  session_id
```

Send to External System

Clicking the `Run Preview` button will begin the 10-second tumble interval and then display the output data stream of this final step of the multi-stage pipeline for this example, as shown below. Decodable handles all the heavy lifting on the backend, allowing you to focus on working directly with your data streams to ensure that you are getting the results you need.

You can watch a demonstration of this example on the **Decodable YouTube** channel, **ML Feature extraction using SQL pipeline transformations and the Moonsense SDK.**


```
{
  "server_time": "2022-03-31 18:27:40",
  "session_id": "5kj23DQ31ds2SJF23r",
  "record_count": 108,
  "velocity_p0": 0.0623409298,
  "velocity_p25": 0.2609802933,
  "velocity_p50": 0.3587928732,
  "velocity_p75": 0.4160982341,
  "velocity_p100": 0.4620938423,
  "velocity_mean": 0.3220938409,
  "total_distance": 1268.2983742386,
  "total_duration": 4934,
  "total_velocity": 0.2572387498
}
```


Shipping Logistics Tracking

Shipping's ability to offer efficient and economical long-distance transport puts it at the center of the world economy. The ability to see, in real-time, logistics and tracking information helps facilitate better transportation decisions leading to reduced costs and enhanced services, which plays a key role in improving the customer experience as well as increasing profitability. Being able to offer order tracking provides customers with peace of mind, can win over hesitant buyers, and can even build customer loyalty.

In this example, we'll walk through how the Decodable data service is used to clean, transform, and enrich real-time shipping data. The processed data can then be used to update package tracking websites and mobile apps for customers, or to feed into operational models for transport companies.

Pipeline Architecture

Below we can see a sample of raw shipping event data. In its current form, it is more complex and detailed than what customers want to know about their shipments and what is needed to update a mobile app or website. By using one or more Decodable **pipelines**, which are streaming SQL queries that process data, we can transform the raw data into a form that is best suited for how it will be consumed.

```
the raw data into a form that is best suited for how it will be consumed.
{
  "tracking_number": "9405511899223197428490",
  "tracking_url": "https://tools.usps.com/go/TrackConfirmAction.action?tLabels=9405511899223197428490",
  "status_code": "DE",
  "carrier_code": "usps",
  "carrier_id": 1,
  "carrier_detail_code": null,
  "status_description": "Delivered",
  "carrier_status_code": "01",
  "carrier_status_description": "Your item was delivered in or at the mailbox at 2:03 pm on September 20, 2021 in SAN
FRANCISCO, CA 94118.",
  "ship_date": null,
  "estimated_delivery_date": null,
  "actual_delivery_date": null,
  "exception_description": null,
  "events": [
    {
      "occurred_at": "2021-09-20T19:03:00Z",
      "carrier_occurred_at": "2021-09-20T14:03:00",
      "description": "Delivered, In/At Mailbox",
      "city_locality": "SAN FRANCISCO",
      "state_province": "CA",
      "postal_code": "94118",
      "country_code": "",
      "company_name": "",
      "signer": "",
      "event_code": "01",
      "carrier_detail_code": null,
      "status_code": null,
      "status_description": null,
      "carrier_status_code": "01",
      "carrier_status_description": "Delivered, In/At Mailbox",
      "latitude": 37.774,
      "longitude": -122.475
    },
    {
      "occurred_at": "2021-09-20T13:10:00Z",
      "carrier_occurred_at": "2021-09-20T08:10:00",
      "description": "Out for Delivery",
      "city_locality": "SAN FRANCISCO",
      "state_province": "CA",
      "postal_code": "94118",
      "country_code": "",
      "company_name": "",
      "signer": "",
      "event_code": "OF",
      "carrier_detail_code": null,
      "status_code": null,
      "status_description": null,
      "carrier_status_code": "OF",
      "carrier_status_description": "Out for Delivery",
      "latitude": 37.671,
      "longitude": -122.328
    }
  ]
}
```

Unnest data stream array

For this example, each record of the raw tracking stream contains data about the shipment as well as an `events` field, which contains an array of tracking data that needs to be unnested (or demultiplexed) into multiple records. To accomplish this, a **cross join** is performed between the `tracking-raw` data stream and the results of using the `unnest` function on the `events` field.

For example, if a given input record contains an array of 5 shipping event updates, this pipeline will transform each input record into 5 separate output records for processing by subsequent pipelines.

When the pipeline is running, the effects of unnesting the input records can be seen in the Overview tab which shows real-time data flow statistics. The input metrics will show a given number of records per second, while the output metrics will show a higher number based on how many elements are in the `events` array.

After creating a new pipeline and entering the SQL query, clicking the `Run Preview` button will verify its syntax and then fire up a new executable environment to process the next 10 records coming in from the source stream and display the results.

```
insert into parsed
select

  -- each element of the `events` array creates a new record
  to_timestamp(e.occurred_at, 'yyyy-MM-dd''T''HH:mm:ss''Z''') as
  occurred_at,
  e.description as description,
  e.city_locality as city_locality,
  e.state_province as state_province,
  e.postal_code as postal_code,
  e.country_code as country_code,
  e.latitude as latitude,
  e.longitude as longitude,

  -- non-array fields common to each record are also included in
  the output
  tracking_number,
  status_description,
  carrier_code,
  carrier_status_description

from `tracking-raw`
cross join unnest(`events`) as e
```

Transform and enrich data stream

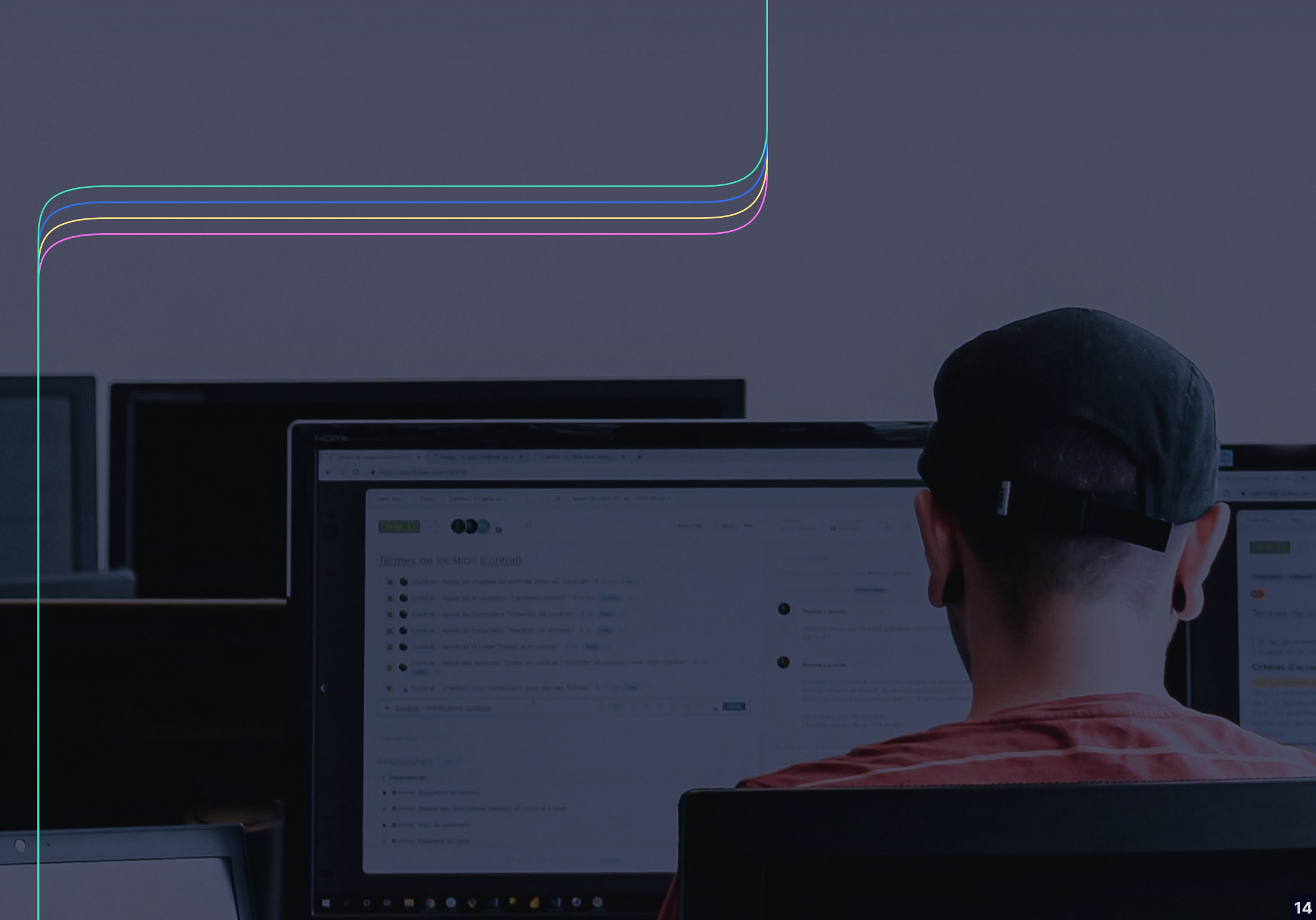
In the next stage of pipeline processing, we want to determine how far the package traveled and how much time has elapsed since the last tracking update. Because SQL provides a comprehensive set of powerful **functions**, such as `cos` and `sqrt`, we can leverage these to enrich the data stream with the results of calculations that are more useful for subsequent processing.

An inner nested `select` query is used to combine the tracking data from the previous record with the current record using the `lag` **window function**, which provides access to a record at a specified physical offset which comes before the current record (in this case that is simply the previous record). Then the outermost `select` query calculates the distance and the difference between the times.

```
insert into summary
select
  *,
  timestampdiff(
      MINUTE,
      occurred_at,
      occurred_at_prev
  ) as elapsed_minutes,
  12742 * asin(sqrt(
    power(sin((latitude_prev - latitude) * 0.008725), 2) +
      cos(latitude * 0.01745) *
      cos(latitude_prev * 0.01745) *
      power(sin((longitude_prev - longitude) * 0.008725), 2)
  )) as distance_traveled -- non-array fields common to each
from (
  select
    *,
    lag(occurred_at, 1) over (
      partition by tracking_number
      order by occurred_at
    ) as occurred_at_prev,
    lag(latitude, 1) over (
      partition by tracking_number
      order by occurred_at
    ) as latitude_prev,
    lag(longitude, 1) over (
      partition by tracking_number
      order by occurred_at
    ) as longitude_prev
  from parsed
)
```

Decodable handles all the heavy lifting on the backend, allowing you to focus on working directly with your data streams to ensure that you are getting the results you need.


```
{
  "occurred_at": "2021-09-20 19:03:00",
  "description": "Delivered, In/At Mailbox",
  "city_locality": "SAN FRANCISCO",
  "state_province": "CA",
  "postal_code": "94118",
  "country_code": "",
  "latitude": 37.774,
  "longitude": -122.475,
  "tracking_number": "9405511899223197428490",
  "status_description": "Delivered",
  "carrier_code": "usps",
  "carrier_status_description": "Your item was delivered in or at the mailbox at 2:03 pm on September 20, 2021 in SAN
FRANCISCO, CA 94118.",
  "occurred_at_prev": "2021-09-20 13:10:00",
  "latitude_prev": 37.671,
  "longitude_prev": -122.328,
  "elapsed_minutes": 353,
  "distance_ traveled": 17
}
```

Website Clickstream Analytics

Clickstream data is collected from user activity on the web, and is used to provide insight into how visitors get to the website, what they do once there, how long they stay on any given page, the number of page visits visitors make, and the number of unique and repeat visitors. Clickstream analytics have the ability to refine data by processing, cleaning, and transforming the raw data into convenient structures that make analysis of data easy and more accurate. Using web data, businesses can not only identify customer needs but can offer customized solutions to cater to the needs of an evolving customer base. The global clickstream analytics market size was valued at $868.8 million in 2018, and is projected to reach $2.5 billion by 2026, indicating a significant focus for businesses.

In this example, we'll walk through how the Decodable data service is used to clean, transform, and enrich real-time clickstream data. The processed data can then be used to inform business decisions.

Pipeline Architecture

Below we can see a sample of raw clickstream data, with one record per page visit for every user of each website monitored. Currently, it is not in the best form for analyzing how well the website is performing. For this, it would be better to have statistics aggregated over time. By using one or more Decodable **pipelines**, which are streaming SQL queries that process data, we can transform the raw data into a form that is best suited for how it will be consumed.

```
{
  "event_datetime": "2020-11-16 22:59:59",
  "event": "view_item",
  "user_id": "f6d4-24d4-4a29-3be1",
  "click_id": "a5cf-179b9-c9d4-83ab",
  "site_id": "wj32-gao1-4w1o-iqp4",
  "page": {
    "id": "b7b1-05fb-bf95-a85a",
    "url": "/product-67890",
    "previous_id": "2905-81e7-be8e-4814",
    "previous_url": "/category-tshirts"
  },
  "engagement": {
    "seconds_on_data": 79,
    "percent_viewed": 39.7
  }
}
```

Aggregate and enrich data stream

For this example, the pipeline leverages the SQL tumble group window function to create a set of records across a non-overlapping, continuous window with a fixed duration of 1 hour. For each interval, the number of pages visited, the total amount of time spent reading or interacting with these pages, and an average of how much of the pages were actually viewed is calculated, grouped by website and user.

As an alternative, the hop window function could be used to create a set of records across a fixed duration that hops (or slides) by a given interval. If the hop interval is smaller than the window duration, the hopping windows overlap, and records from the data stream are assigned to multiple windows. Then a subsequent pipeline could be used to filter the results to one representing the highest level of engagement over a set duration for each user for each website.

```
insert into summary
select
  window_start,
  window_end,
  site_id,
  user_id,
  count(1) as pages_visited,
  sum(engagement.seconds_on_data) as total_seconds_on_site,
  avg(engagement.percent_viewed) as avg_percent_viewed,
from table (
  tumble(
    table clickstream,
    descriptor(to_timestamp(event_datetime)),
    interval '1' hour
  )
)
group by
  window_start,
  window_end,
  site_id,
  user_id
```

After creating a new pipeline and entering the SQL query, clicking the `Run Preview` button will verify its syntax and then fire up a new executable environment to process the next 10 records coming in from the source stream and display the results. Decodable handles all the heavy lifting on the backend, allowing you to focus on working directly with your data streams to ensure that you are getting the results you need.


```
{
  "window_start": "2020-11-16 14:00:00",
  "window_end": "2020-11-16 15:00:00",
  "user_id": "f6d4-24d4-4a29-3be1",
  "site_id": "wj32-gao1-4w1o-iqp4",
  "pages_visited": 8,
  "total_seconds_on_site": 426,
  "avg_percent_viewed": 28.198543
}
```


Conclusion

As we can see from these use case examples, sophisticated business problems can be addressed in a very straight-forward way using Decodable pipelines. It is not necessary to create docker containers, there is no SQL server infrastructure to set up or maintain, all that is needed is a working familiarity with creating the SQL queries themselves.

Today, the world of data is separated into online operational data infrastructure and offline analytical data infrastructure. The two are generally connected by data integration. With real-time data integration infrastructure that can power both high SLA, low latency, operational use cases, there's no reason to maintain a separate data stack for data integration into analytical systems. By feeding both operational and analytical data infrastructure from the system and data, costs are reduced, compliance becomes simpler, data quality increases, and analytical systems better reflect reality.

Many companies are claiming the data warehouse is the center of the universe and, for many people, it is. However, if Snowflake or Databricks suddenly became inexpensive tomorrow, they wouldn't be low latency enough for operational workloads. If they were low latency, they wouldn't be transactional and strongly consistent. If they were transactional and strongly consistent, they wouldn't have application framework support.

The fact is, data warehouses are an enormous destination for data with many workloads, but not all. The operational world is made up of purpose-built systems: streaming systems, messaging systems, OLTP databases, key-value stores (DHTs), online feature stores, ultra-fast in memory caches, document databases, and a myriad of SaaS business applications, each optimized for a specific workload. While the data warehouse is trying to expand, it is simply not feasible for a single system to satisfy all of these workloads. The data warehouse has never been, nor will ever be, the center of the world. Embracing that reality, Decodable will be the platform that ties all of these systems together.

Next Steps

This eBook explored five common use case scenarios, with **additional examples** available on our website.

To follow along with these use case examples, sign up on the **decodable.co** website for a free **Decodable account.**

All code in this guide can be found in our GitHub repo, **github.com/decodableco/examples.**

You can watch demonstrations of several examples on the Decodable YouTube channel, **youtube.com/@decodable.**

Additional documentation for all of Decodable's services is available at **docs.decodable.co.**

Please consider joining us on our **community Slack.**