Aggregating Streaming Data in Real Time

- Introduction
- Tutorial steps
Introduction

In the previous tutorial, you looked at the Siddhi real time data summarization capabilities by calculating the total production in the past minute. Now let's consider a more advanced scenario where you need to calculate the total value for a specific time period.

In this scenario, the foreman of the Sweet Factory needs to know the total production of Sherbet Lemon during each hour in November 2017.

It is costly to do this by recalculating the total for each and every event. What you need is a time based aggregation of the events in real time and retrieval on demand. Siddhi supports this functionality through the Incremental Aggregation concept.

Incremental Aggregation calculates the aggregated values continuously and stores them. These values can be retrieved efficiently from the store on demand. Furthermore, Incremental Aggregators support out of order event arrival with in-memory buffers for higher accuracy.

This tutorial covers the following concepts:

- Introduction to incremental aggregation
- Retrieval from incremental aggregation
Before you begin:

In this scenario, information sent by the Sweet Bots are stored in a MySQL table named SweetFactoryDB. You need to download and install MySQL, and create this table before you carry out the tutorial steps.

1. Download and install MySQL Server.
2. Download the MySQL JDBC driver.
3. Unzip the downloaded MySQL driver zipped archive, and copy the MySQL JDBC driver JAR (mysql-connector-java-x.x.xx-bin.jar) into the `<SP_HOME>/lib` directory.
4. Enter the following command in a terminal/command window, where `username` is the username you want to use to access the databases.
   ```
   mysql -u username -p
   ```
   When prompted, specify the password you are using to access the databases with the username you specified.
5. When prompted, specify the password you are using to access the databases with the username you specified.
6. Add the following configuration under the Data Sources Configuration section of the `<SP_HOME>/conf/editor/deployment.yaml` file.

   ```yaml
   - name: SweetFactoryDB
     description: Datasource used for Sweet Factory Supply Records
     jndiConfig:
       name: jdbc/test
       useJndiReference: true
     definition:
       type: RDBMS
       configuration:
         jdbcUrl: 'jdbc:mysql://localhost:3306/test'
         username: root
         password: root
         driverClassName: com.mysql.jdbc.Driver
         maxPoolSize: 50
         idleTimeout: 60000
         connectionTestQuery: SELECT 1
         validationTimeout: 30000
         isAutoCommit: false
   ```

7. To create a database table named SweetFactoryDB, issue the following commands from the terminal.

   ```sql
   mysql> create database SweetFactoryDB;
   mysql> use SweetFactoryDB;
   mysql> source <SP_HOME>/wso2/editor/dbscripts/metrics/mysql.sql;
   mysql> grant all on SweetFactoryDB.* TO username@localhost identified by "password";
   ```
Tutorial steps

Let's get started!

**User Scenario 1: Defining incremental aggregation**

In this scenario, let's define an incremental aggregation to calculate the total production in an incremental manner, and store the results.

1. Let's define an input stream as follows based on the data received from Sweet Bots. This is the same stream definition used in the previous tutorials to capture the name of the sweet category and the amount produced.

   ```
   define stream SweetProductionStream (name string, amount long);
   ```

2. Now, let's define an aggregation for the input data. Here, you can assume that the foreman would like to know the production per hour, month and year for each sweet.

   ```
   define aggregation SweetProductionAggregation from SweetProductionStream
   select name, sum(amount) as totalAmount
   group by name
   aggregate every hour...year
   ```

   This calculates the total amount per hour, day, month and year by the arrival time of each event. Incremental Aggregation can also be done for seconds, minutes, hours, days, months and years. However, in this sweet production scenario, aggregating by second holds no information value. Therefore, the sweet production is aggregated from hour to year.

3. Now, comes the question of when the production occurs. In the above aggregation, event arrival time is the time used in aggregation. The Sweet Bots send information directly from the factory floor to the server in the same network. Therefore, we can assume that the event arrival time is the production time.

   - If you want you can be more accurate by appending the data sent by the SweetBots to include time as shown below.
     a. First define the input stream to include a timestamp:

       ```
       define stream SweetProductionStream (name string, amount long, timestamp long);
       ```

     b. Then use the timestamp for aggregation as shown below.

       ```
       aggregate by SweetProductionStream.timestamp every hour...year
       ```

   For this tutorial, let's continue to use the format mentioned first instead of the format in these substeps because the time differences are very slight in the hourly calculations.
The completed Siddhi application looks as follows.

```
from SweetProductionStream
select name, sum(amount) as totalAmount
group by name
```

This part of the aggregation specifies the following:

- From where the information to be processed is taken (i.e., `SweetProductionStream` input stream)
- The value you are aggregating. In this scenario, `sum(amount) as totalAmount` aggregates only the summation of values. The aggregation can also be `minimum`, `maximum` or `average`.
- The `group by` clause is optional and can be ignored if all production must be aggregated.

User Scenario 2: Retrieval of data on demand

In the previous scenario, you defined the aggregation. Now let's see how to retrieve from it. Siddhi supports this functionality through correlation of data. In this tutorial, you are retrieving data via aggregation joins. For more information on correlating data through joins see [Siddhi Query Guide - Joins](#).

1. First, let's define a stream to retrieve data. The foreman needs to see the hourly production of Sherbet Lemon for November 2016. Therefore, the criteria to retrieve values are as follows.

<table>
<thead>
<tr>
<th>Sweet</th>
<th>Sherbet Lemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval</td>
<td>Hourly</td>
</tr>
<tr>
<td>Duration</td>
<td>November 2017</td>
</tr>
</tbody>
</table>

Therefore, the input stream needs to be defined as follows:

```
define stream GetTotalSweetProductionStream (name string, start_duration string, end_duration string, interval string);
```

2. A possible output of this retrieval is the timestamp (beginning of each hour), the name of the sweet and the total amount. Therefore, let's define an output stream with these values as follows.

```
define stream SweetProductionStream(name string, amount long);
@store(

define aggregation SweetProductionAggregation
from SweetProductionStream
select name, sum(amount) as totalAmount
group by name
aggregate every min...year
```

User Scenario 2: Retrieval of data on demand

In the previous scenario, you defined the aggregation. Now let's see how to retrieve from it. Siddhi supports this functionality through correlation of data. In this tutorial, you are retrieving data via aggregation joins. For more information on correlating data through joins see [Siddhi Query Guide - Joins](#).

1. First, let's define a stream to retrieve data. The foreman needs to see the hourly production of Sherbet Lemon for November 2016. Therefore, the criteria to retrieve values are as follows.

<table>
<thead>
<tr>
<th>Sweet</th>
<th>Sherbet Lemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval</td>
<td>Hourly</td>
</tr>
<tr>
<td>Duration</td>
<td>November 2017</td>
</tr>
</tbody>
</table>

Therefore, the input stream needs to be defined as follows:

```
define stream GetTotalSweetProductionStream (name string, start_duration string, end_duration string, interval string);
```

2. A possible output of this retrieval is the timestamp (beginning of each hour), the name of the sweet and the total amount. Therefore, let's define an output stream with these values as follows.
define stream HourlyProductionStream(AGG_TIMESTAMP long, name string, totalAmount long);

In the above definition, AGG_TIMESTAMP is the internal reference of the aggregation defining the start of the time interval.

3. Now, let's use the aggregation, retrieval stream, and the output stream to define data correlation from an aggregation. Aggregation for the selected period contains aggregation for all sweets. Therefore, let's join the aggregation, and the retrieval stream based on the sweet name to filter aggregations for Sherbet Lemon.

   ```sql
   from GetTotalSweetProductionStream as b join SweetProductionAggregation as a on a.name == b.name
   ```

4. You need to retrieve data relevant only for November 2017. Therefore, let's add it in the retrieval stream as the duration.

   ```sql
   from GetTotalSweetProductionStream as b join SweetProductionAggregation as a on a.name == b.name within b.duration
   ```

   In the output event, the duration for which the data is retrieved must be represented in a specific format. For example, November 2017 can be represented as 2017-11-** **:**:**: **. The supported date formats are <yyy>-<MM>-<dd> <HH>:<mm>:<ss> (if time is in GMT) and <yyyy>-<MM>-<dd> <HH>:<mm>:<ss> <Z> (if the time is not in GMT), here the ISO 8601 UTC offset must be provided for <z> (e.g., +05:30, -11:00).

   If the user needs a specific time duration, the query must be changed as follows. Both durations specified must adhere to the data formats required by Siddhi.

   ```sql
   from GetSweetProductionStream as b join SweetProductionAggregation as a on a.name == b.name within b.start_duration, b.end_duration
   ```

5. Let's add interval for the retrieval to specify for which intervals you want the data to be retrieved.

   ```sql
   from GetTotalSweetProductionStream as b join SweetProductionAggregation as a on a.name == b.name within b.duration per b.interval
   ```

   Interval can be in the format of SECONDS, MINUTES, HOURS, DAYS, MONTHS or YEARS (these values are not case sensitive).

The completed statement including the output stream looks as follows:
from GetTotalSweetProductionStream as b join SweetProductionAggregation as a
  on a.name == b.name
  within b.duration
  per b.interval
  select a.AGG_TIMESTAMP, a.name, a.totalAmount
  insert into HourlyProductionStream;

In the above definition, a.AGG_TIMESTAMP is the internal data of the aggregation defining the start of the time interval. For instance, in the November 2017 duration, there is a 24*30 hourly production aggregation. The first output event has the timestamp of the date and time of 1st November 2017 00:00:00.

The completed Siddhi application with the possible sink and source configurations is as follows.

```siddhi
@App:name('TotalProductionHistoryApp')
@source(type = 'http', @map(type = 'json'))
define stream SweetProductionStream(name string, amount long);
@source(type = 'http', @map(type = 'json'))
define stream GetTotalSweetProductionStream (name string, duration string, interval string);

@sink(type='log', prefix='Hourly Production Stream')
define stream HourlyProductionStream(AGG_TIMESTAMP long, name string, totalAmount long);

@index('name')
@store(type='rdbms', jdbc.url="jdbc:mysql://localhost:3306/SweetFactoryDB", username="root", password="root", jdbc.driver.name="com.mysql.jdbc.Driver")
define aggregation SweetProductionAggregation
  from SweetProductionStream
  select name, sum(amount) as totalAmount
  group by name
  aggregate every hour ... year;
from GetTotalSweetProductionStream as b join SweetProductionAggregation as a
  on a.name == b.name
  within b.duration
  per b.interval
  select a.AGG_TIMESTAMP, a.name, a.totalAmount
  insert into HourlyProductionStream;
```

What's Next

Analyzing KPIs