ESPER TECH

CHAPTER 4. CONTEXT AND CONTEXT PARTITIONS

DA SILVA MARQUES SABRINA
ARAYA FUENTES BEATRIZ
BAH SALIFOU ANNE
FUENTES ESTRELIA
MIGUT ERIK

SUMMARY

INTRODUCTION

Importance of context on EPL.

Context : ***"the set of circumstances or facts that surround a particular event, situation, etc."***

Context is built on many events that are organized in various sets: we call it context partition.
Context partition will be adapted to a specific context: each customer has a different partition.
You can use aggregations, data windows, time or other patterns of EPL to :

- analyze banking transactions
- monitor speed violations

GOAL

To analyze data without doing it one by one

Benefits of context :

- **To reduce the need of duplication** :
- **Time flexibility** : define a period of analysis
- **Life cycle**: context partition has an independent cycle from statements, you can determine a start and an ending.
- **Easier to read**
- **Nested context :** use various contexts
- **Events aggregation**
- **Boundaries coordination**

STEP 1

CONTEXT DECLARATION

Create context : statement to specifiy which subject is analyzed

Partition_def : sets of events that will be use to detail context

Context isn't initiated until you enter in the application a statement that refers to the settled context.

You must give it material work.

Initiated by : starting point of the application

Terminated by : end of application

EXAMPLE

```
create schema StockTick(symbol string, price double);
create schema ServiceOrder(custId string, name string, price double);
create schema ProductOrder(custId string, name string, price double);

select a.custId, sum(a.price + b.price) from pattern [every a=ServiceOrder ->
b=ProductOrder(custId=a.custId) where timer:within(1 min)]#time(2 hour) where a.name in ('Repair',
b.name) group by a.custId having sum(a.price + b.price)>100
```

EXAMPLE

EPL Statements

EPL Module Text

Enter EPL Here:

```
create schema StockTick(symbol string, price double);
create schema ServiceOrder(custId string, name string,
price double);
create schema ProductOrder(custId string, name string,
price double);

select a.custId, sum(a.price + b.price) from pattern [every
a=ServiceOrder -> b=ProductOrder(custId=a.custId) where
timer:within(1 min)]#time(2 hour) where a.name in
('Repair', b.name) group by a.custId having sum(a.price +
b.price)>100
```

Time And Event Sequence

Beginning Of Time

Provide a timestamp to start at:

2001-01-01 08:00:00.000 **Submit**

Advance Time and Send Events

Enter sequence of time and events:

```
t=t.plus(45 seconds)
ServiceOrder={custId='ID1', name='Repair', price=142}
ProductOrder={custId='ID1', name='Repair', price=2}

t=t.plus(60 seconds)
ServiceOrder={custId='ID1', name='Repair', price=171}
ProductOrder={custId='ID1', name='Repair', price=22}
```

Scenario Results

All Output Events | Output Per Statement | All Audit Text | Audit Text Per Statement

```
At: 2001-01-01 08:00:45.000
  Statement: stmt-4
    Insert
      stmt-4-output=
      {a.custId='ID1',
      sum(a.price+b.price)=144.0}
At: 2001-01-01 08:01:45.000
  Statement: stmt-4
    Insert
      stmt-4-output=
      {a.custId='ID1',
      sum(a.price+b.price)=337.0}
```

HASH SEGMENTED CONTEXT

This setup establishes rules for a task management app, restricting task addition or editing to specific working hours, from 9:00 AM to 5:00 PM.

1. **Creation of WorkingHours Context:** This defines the time frame for adding or editing tasks, starting at 9:00 AM and ending at 5:00 PM.
2. **Functionality to Add a Task:** Checks if the current time is between 9:00 AM and 5:00 PM. If so, it allows adding a task; otherwise, it shows a message saying tasks can only be added during working hours.
3. **Functionality to Edit a Task:** Similar to adding tasks, it checks if the current time is within working hours. If yes, it allows editing a task by updating its name; otherwise, it displays a message indicating tasks can only be edited during working hours.

In summary, this setup ensures tasks can only be added or edited during the specified working hours, maintaining consistency and organization in the task management a

CONTEXTE SEGMENTÉ DE HACHAGE

- Scenario: We aim to create a task management application that only allows adding or modifying tasks during working hours, specifically from 9:00 AM to 5:00 PM.

Creation of **WorkingHours Context**:

create context WorkingHours start (1, 9, *, *, *) end (1, 17, *, *, *);

- Creation of **Functionality to Add a Task**:

```
context WorkingHours function addTask(taskName string) {
if (current_time() >= 9 && current_time() <= 17) {
insert into Tasks (taskName) values (taskName);
} else {
 print("Tasks can only be added between 9:00 AM and 5:00 PM.");
} }
```

- Creation of **Functionality to Edit a Task**:

```
context WorkingHours function editTask(taskID string, newTaskName string) { if
(current_time() >= 9 && current_time() <= 17) { update Tasks set taskName = newTaskName where taskID = taskID;
 } else {
print("Tasks can only be edited between 9:00 AM and 5:00 PM.");
 } }
```

CONTEXT CONDITIONS

Les conditions de contexte sont les clauses pour spécifier les conditions dans laquelle une règle doit être appliquée.

Exemple de conditions : filtres / periodes / modeles

- filtres sur les évènements

SELECT * FROM Event
WHERE value > 5

Illustration :
contexte global:
rule R1:
select * from SpeedEvent where speed > 90 and road = "Autoroute A1" context PeakHours action raiseAlert("Plusieurs véhicules dépassent la vitesse autorisée sur l'Autoroute A1 pendant les heures de pointe !")
contexte PeakHours: condition: timeOfDay >= 7:00 and timeOfDay <= 9:00
rule R2: select * from SpeedEvent where speed > 90 and road = "Autoroute A1" action notifyFleetManager("Dépassement de vitesse sur l'Autoroute A1 pendant les heures de pointe !")

CONTEXT NESTING

L’imbrication de contexte est la capacité a utiliser plusieurs conditions de contexte dans une seule requête

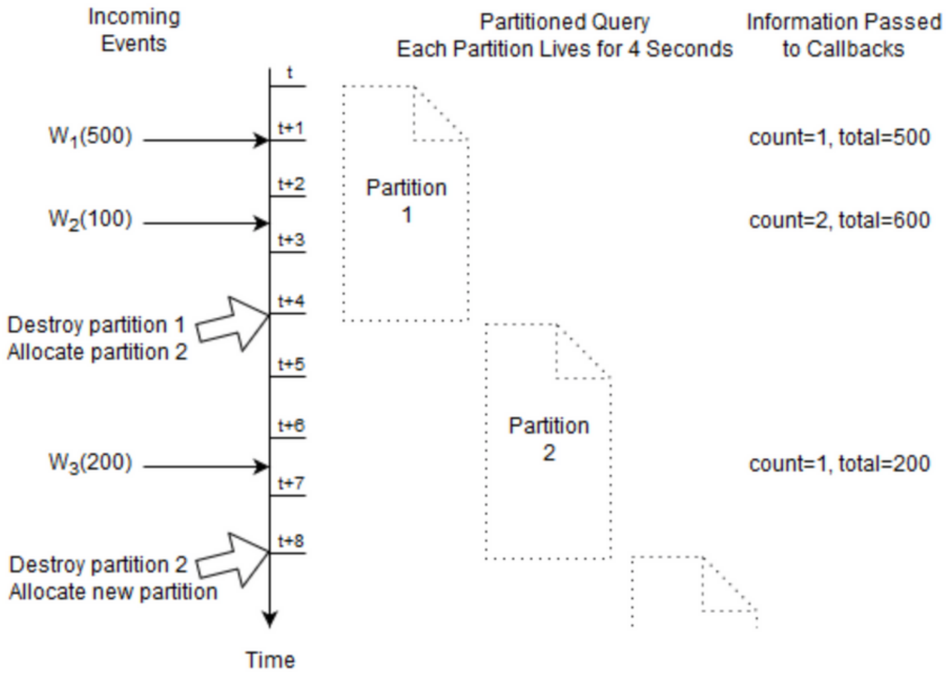
Exemple :

SELECT * FROM EventA(value > 10) WHERE EXISTS (SELECT * FROM EventB(value < 5) WHERE timer:within(10 seconds)

Illustration :

contexte global:
rule R1: select * from TemperatureEvent where temperature > 30 action raiseAlert("Il fait trop chaud dans la maison !") contexte spécifique: rule R2: select * from TemperatureEvent where room = "Cuisine" and temperature > 25 action turnOnFan("Cuisine") rule R3: select * from TemperatureEvent where room = "Salon" and temperature > 28 action turnOnAC("Salon")

PARTITIONING WITHOUT CONTEXT DECLARATION



- **No need to declare a context to partition data windows.**

OUTPUT WHEN A CONTEXT PARTITION STARTS (NON-OVERLAPPING CONTEXT) OR INITIATES (OVERLAPPING CONTEXT)

Context: **What is the capital of France?**

When we ask the model to answer this question, the output could be:

output**: Answer: The capital of France is Paris."**

The **output** **provides an answer to the question asked in the context**

Text segment began to be analyzed by the model.

OUTPUT WHEN A CONTEXT PARTITION ENDS (NON-OVERLAPPING CONTEXT) OR TERMINATES (OVERLAPPING CONTEXT)

Context: " In a distant land, there was a young boy named Peter. He lived in a small house by the river, where he spent his days fishing and dreaming of adventures."
When the model comes to the end of this context, the output could be a natural continuation of the story.
(example) Output: "One day, while sitting by the water, Peter saw something shiny at the bottom of the river. Carefully, he dipped his hand in cold water and came out with a silver ring adorned with a sparkling gem. It was the beginning of an incredible adventure for the boy."

Text segment was analyzed in its entirety by the model

Context and Named Window

For 9:00:
Context: Outdoor temperature at 9:00
For 10:00:
Context Outdoor temperature at 10:00
For 11:00:
Context: Outdoor temperature at 11:00
...
For 5:00pm:
Context: Outdoor temperature at 17:00

Each named window captures the outside temperature at a specific time, with no overlap between hours. This makes it possible to analyze temperature variations throughout the day, with separate data for each hour

CONTEXT AND TABLES

Tables in EPL are global data structures that store rows of data organized by primary keys. These tables can be referenced by multiple statements and can interact with contexts. It's crucial that both the table and any interacting statements share the same context to maintain consistency. This restriction ensures that operations like aggregation, selection, updates, merges, and deletions on the table are only performed within the defined context parameters.

EXAMPLE

Scenario: we're managing reservations for a restaurant that wants to track average spending by customers, but only during dinner hours, say from **6** PM to **11** PM.

Name of the context : **DinnerHours**

create context DinnerHours start (0, 18, *, *, *) end (0, 23, *, *, *);

- Creation of a table to store the average in **DinnerHours**

context DinnerHours create table AverageSpendTable (customerId string primary key, avgSpend avg(double));

- We aggregate data into this table, ensuring to declare the same DinnerHours context. This ensures that data is only aggregated during the specified dinner hours:

context DinnerHours into table AverageSpendTable select avg(spend) as avgSpend from ReservationEvent group by customerId;

In this example, any operation on **AverageSpendTable**, whether it's updating, querying, or deleting data, must also declare the **DinnerHours** context.

CONTEXT AND VARIABLES

Variables can be either global variables or context variables, and are linked to the context. They differ significantly in their scope and on how their values are maintained across different contexts.

Global variables : if you change the value of a global variable in one part of your application, that change is reflected throughout the entire application, regardless of any context partitions.

Context variables : each partition of the context can have its own, independent value for the variable and it’s specific to a context partition.

EXAMPLE

- Scenario: with two parking lots, A and B, each with a different capacity threshold for when to display a "Full" sign.

- **Global Variable Example**: If we had a **global variable var_global_full_sign_threshold**, set to 50, this would mean both parking lots A and B would display the "Full" sign when they reach 50 parked cars. This is a universal threshold, not considering the different sizes or capacities of the lots.
- **Context Variable Example:** With context variables, we could have **var_parkinglot_threshold** as a context variable within a **ParkingLotContext**. Parking lot A, being larger, could have its threshold set to 100, whereas the smaller lot B could have its threshold set at 30. This allows each parking lot to operate based on its own capacity, providing a more flexible and realistic management of the "Full" sign display.

OPERATIONS ON SPECIFIC CONTEXT PARTITIONS

Both functionalities rely on the ability to select and operate on specific context partitions, enhancing the flexibility and efficiency of data processing within EPL applications.

Iterating through context partitions : the iteration over data within specific context partitions, usefull to analyze or process data segment by segment, based on the defined context.

Executing fire-and-forget queries : for on-demand data retrieval from specific context partitions without maintaining a continuous query listener. This approach is suitable for scenarios where immediate, one-time data retrieval is needed from certain partitions. Again, any order-by clauses will apply within, not across, the partitions queried.

We first defined a **TemperatureEvent** that represents a temperature reading in a room, with attributes for the room identifier (roomId), the temperature reading (temperature), the direction (to identify south-facing rooms, direction), and a timestamp.
We created a **RoomContext** context that partitions the data by roomId, enabling calculations and analyses specific to each room.
We set a temperature threshold for south-facing rooms (**southRoomThreshold**) and wrote a query to iterate over the average temperatures of south-facing rooms over a day, to determine whether they exceed this threshold.
Finally, we defined a fire-and-forget query to obtain the maximum temperature recorded in the kitchen over the course of the day, which you can run on demand without maintaining continuous monitoring.

```
-- Définition de l'événement pour les relevés de température
create schema TemperatureEvent(roomId string, temperature double, direction string, timestamp long);

-- Création d'un contexte pour chaque pièce, partitionné par roomId
create context RoomContext partition by roomId from TemperatureEvent;

-- Déclaration des seuils pour les températures
create variable double southRoomThreshold = 25.0; -- Exemple de seuil pour les pièces orientées au sud

-- Requête pour itérer sur les températures des pièces orientées au sud
context RoomContext
select roomId, avg(temperature) as avgTemp
from TemperatureEvent(direction='south').win:time(1 day) -- Fenêtre de temps de 1 jour
group by roomId
having avg(temperature) > southRoomThreshold;

-- Définition d'une requête fire-and-forget pour la cuisine
-- Cette requête peut être exécutée à la demande pour récupérer la température la plus élevée de la journée dans la cuisine
-- Notez que dans un environnement réel, vous utiliseriez une API ou un mécanisme d'interface pour exécuter cette requête spécifique selon les besoins
select max(temperature) as maxTemp
from TemperatureEvent(roomId='kitchen').win:time(1 day);
```

SYNTAX

s

SOURCE

http://esper.espertech.com/release-8.9.0/reference-esper/html_single/index.html#context