Processamento de Streams
Análise a Corridas de Táxis

André Lopes - 45617
Nelson Coquenim - 45694
Departamento de Informática
Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
Almada, Portugal

I. Introdução

Este projecto foi desenvolvido no âmbito da cadeira de Processamento de *Streams* (PStr). O objectivo passa por implementar de novo as *queries* do primeiro projecto de PStr, nomeadamente as do DEBS Grand Challenges 2015. A diferença é que desta vez serão implementadas usando SiddhiQL do WSO2 no lugar do SparkStreaming. Para além destas, serão implementadas mais 3 *queries* que serão descritas mais à frente.

O WSO2 Complex Event Processor (CEP) ajuda a identificar os eventos e padrões mais significativos vindos de múltiplas *sources*, analisando o seu impacto e agindo em *real-time*. O WSO2 CEP apoia-se numa linguagem designada SiddhiQL, que é fácil de usar devido à sua semelhança ao SQL, sendo esta especializada em *queries* complexas envolvendo janelas e detecção tanto de padrões como de sequências.

A implementação de *queries* semelhantes às do primeiro projecto permitirá uma comparação entre as duas tecnologias em termos de manutenção e flexibilidade. A manutenção será avaliada através da comparação da simplicidade do código, pois será possível fazer o paralelo entre o código das *queries* em SiddhiQL e em SparkStreaming, assinalando os benefícios de utilizar uma linguagem de alto nível. No que toca à flexibilidade, o SiddhiQL oferece uma semântica de sequência que permite a identificação de padrões complexos.

II. *Setup*

A *source* dos eventos processados pelo WSO2 CEP é o Apache Kafka. De forma a executar o Kafka é necessário primeiro executar uma instância do Zookeper que tem como benefício a coordenação dos *brokers* do Kafka, o que neste caso não seria necessário, mas o Kafka tem essa obrigatoriedade.

Tendo o *broker* activo, é necessário executar um produtor Kafka, neste caso implementado em Java, que emite os eventos de corridas de táxis com uma cadência configurável. Este produtor lê de um *dataset*, no qual cada linha corresponde a uma viagem de táxi, e emite para o *broker*. Sendo o Kafka um sistema *topic-based*, o tópico no qual estes eventos são publicados é designado ”debs”. A cadência de eventos foi definida como sendo 60, o que significa que 1 minuto corresponderá a 1 segundo.

De forma a registar as *queries* SiddhiQL que implementam as interrogações definidas, utilizou-se o Stream Processor Studio do WSO2. Este permite, entre outros, instalar *queries* e definir os adaptadores necessários, que permitem consumir a *stream* de eventos através do *broker* Kafka. Para isto, é necessário definir qual o tópico que se pretende subscrever, e como referido anteriormente no caso da produção dos eventos, será o ”debs”.

A cada evento que chega ao CEP *engine* serão executadas as *queries* instaladas, resultando destas novas *streams*. Por este motivo é importante definir um conjunto de *sinks*, para as quais serão emitidos os novos eventos produzidos pelo processamento dos eventos emitidos pelas *sources*. Nesta implementação foi definido que a *sink* é o *broker* do Kafka, sendo que para cada *query* existirá um produtor de eventos com um tópico diferente. Esta implementação permitirá que qualquer aplicação subscreva a *stream* de *output* que desejar, tendo os dados processados em *real-time*.

III. *Queries*

Para cada *query* será apresentada a sua descrição e a sua implementação acompanhada de uma explicação. Importa também referir que sempre que se referir a minutos, na *query* corresponde a segundos (como referido na introdução). Outro dos pontos importantes a ter em conta nesta secção é a descrição da *query* **TaxiRidesStr** que é transversal a todas as interrogações. Esta consiste numa *query* que tem como *input* a *stream* de produção, sendo que esta efetua algum pré-processamento necessário, como é o caso da transformação das coordenadas esféricas em *cells*.

A. *Frequent Routes*

O objetivo desta primeira *query* é calcular o *top* 10 das rotas mais frequentes durante um período de 30 minutos. Uma rota é representada por uma *cell* inicial e uma *cell* final.

Na Figura 1 pode-se observar a estrutura desta *query*:

- **TopFreqRoutesStr**: Esta *query* aplica uma janela deslizante de 30 segundos. Após isto, agrupa por rota e conta a frequência de corridas de táxi em cada uma das rotas. Finalmente, faz *output* apenas do *top* 10 das rotas mais frequentadas.

O código SiddhiQL que implementa esta *query* é o seguinte:

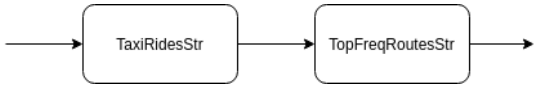

Fig. 1. Diagrama da query *Frequent Routes*.

```
from TaxiRidesStr#window.time(30 sec)
select pickup_gridID,
       dropoff_gridID,
       count(*) as frequency
group by pickup_gridID, dropoff_gridID
order by frequency DESC
limit 10
insert into TopFreqRoutesStr;
```

B. Profitables Areas

Nesta query pretende-se identificar, de forma contínua, as áreas que são mais lucrativas para os taxistas. Para tal, o lucro de uma área é definido pelas receitas geradas nessa área a dividir pelo número de táxis vazios nessa mesma área.

A receita gerada numa área é a média das *fare* + *tip* de todas as corridas que originaram nessa área e que acabaram nos 15 minutos seguintes.

O número de táxis vazios num dada área consiste na soma dos táxis que efetuaram uma *dropoff* nessa área mas que após 30 minutos ainda não efetuaram uma *pickup*.

Na Figura 2 pode-se observar um diagrama que demonstra o fluxo desta *query*:

- **EmptyTaxisStr:** Detecção dos táxis vazios através de um padrão de *absence* de uma corrida de táxis.
- **EmptyTaxisPerAreaStr:** Determinação do número de táxis vazios por área.
- **RevenuePerAreaStr:** Cálculo da receita média gerada nessa mesma área.
- **ProfitableAreasStr:** Junção das duas *streams* anteriores e determinação do lucro por área através da divisão da receita gerada na área pelo número de táxis vazios da mesma.

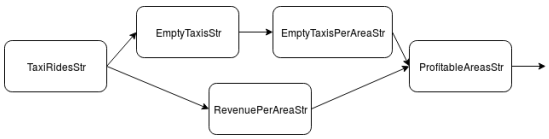

Fig. 2. Diagrama da query *Profitables Areas*.

O código siddhi que implementa esta *query* é o seguinte:

```
partition with
  (medallion of TaxiRidesStr)
begin
  from e1 = TaxiRidesStr ->
    not TaxiRidesStr[medallion ==
    e1.medallion] for 30 sec
  select e1.medallion,
    e1.dropoff_gridID
    insert into EmptyTaxisStr;
end;

from EmptyTaxisStr#window.time(30 sec)
select dropoff_gridID as areaID,
count(*) as emptyTaxis
group by dropoff_gridID
insert into EmptyTaxisPerAreaStr;

from TaxiRidesStr#window.time(15 sec)
select pickup_gridID as areaID,
  avg(fare_amount + tip_amount) as revenue
group by pickup_gridID
insert into RevenuePerAreaStr;

from RevenuePerAreaStr#window.time(15s) as A
join
EmptyTaxisPerAreaStr#window.time(15s) as B
on A.areaID == B.areaID
select A.areaID,
  revenue/emptyTaxis as profit
group by A.areaID
order by profit DESC
limit 10
insert into ProfitableAreasStr;
```

C. Idle Taxis

Neste *use case* espera-se que seja emitido um alerta quando o número de táxis disponíveis torna-se superior ao pretendido. Para tal, deverá ser publicado um aviso quando o tempo de paragem médio (*idle time*) de todos os táxis é superior a 10 minutos. Define-se como tempo de paragem, o período de tempo entre uma *dropoff* e uma *pickup*. Finalmente, assume-se que um táxi encontra-se disponível se tiver realizado pelo menos uma viagem na última hora.

O diagrama na Figura 3 demonstra a lógica da implementação desta *query*:

- **AvailableTaxisStr:** Através de um janela temporal deslizante de uma hora são emitidos os táxis disponíveis, ou seja, os que emitiram eventos na última hora.
- **IdleTimeTaxisStr:** Nesta *query* é efetuada a determinação do período de tempo em que um táxi está desocupado. Este cálculo é feito pela subtração do instante em que ocorreu uma *pickup* pelo instante da *dropoff* da corrida anterior.
- **AlertTooMuchAvailable:** Por último, é emitido um alerta das áreas cuja a média do tempo em que um táxi está desocupado é superior ao limite estabelecido (30 minutos).

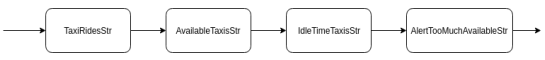

Fig. 3. Diagrama da query *Idle Taxis*.

Em seguida, apresenta-se o excerto de código da *query* em questão:

```
from TaxiRidesStr#window.time(1 hour)
```

```
select *
insert into AvailableTaxisStr;

from e1 = AvailableTaxisStr ->
  e2 = AvailableTaxisStr[medallion
    == e1.medallion]
select e1.medallion as taxi,
  (e2.p_time-e1.d_time) as idle_time
insert into IdleTimeTaxisStr;

from IdleTimeTaxisStr
select avg(idle_time) as avg_idle_time
having avg_idle_time > 10 * 60
insert into IdleTaxisStr;
```

D. Congested Areas

Nesta secção ir-se-à descrever a implementação de uma *query* que emite as localizações onde, possivelmente, poderá haver congestionamentos no trânsito. Para tal, dever-se-à detectar picos nas durações das viagens dos táxis que são seguidos por pelo menos 3 viagens, todas estas com durações crescentes.

Na Figura 4 pode-se observar a estrutura desta *query*:

- **CongestedAreaStr**: Esta *query* consiste na identificação do padrão anteriormente referido. Por cada deteção desse padrão, será feito *output* de um evento que tem como atributos o pico de duração de uma viagem naquele táxi e um identificador da área no qual foi feito o *pickup* da corrida de táxi na qual a duração atingiu o pico.

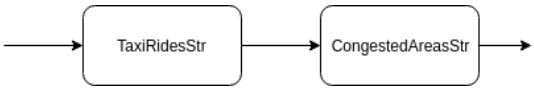

Fig. 4. Diagrama da query *Congested Areas*.

O código SiddhiQL que implementa esta *query* é o seguinte:

```
from every e1 = TaxiRidesStr ->
  e2=TaxiRidesStr[medallion == e1.medallion
  and ride_duration > e1.ride_duration] ->
  e3=TaxiRidesStr[medallion == e2.medallion
  and ride_duration < e2.ride_duration] ->
  e4=TaxiRidesStr[medallion == e3.medallion
  and ride_duration > e3.ride_duration] ->
  e5=TaxiRidesStr[medallion == e4.medallion
  and ride_duration > e4.ride_duration] ->
  e6=TaxiRidesStr[medallion == e5.medallion
  and ride_duration > e5.ride_duration]
select e2.pickup_gridID as areaID,
       e2.ride_duration as peak_duration
insert into CongestedAreasStr;
```

E. Most Pleasant Taxi Drivers

Para recompensar os condutores de táxis mais simpáticos é necessário que seja emitido, uma vez por dia, o taxista que recebeu mais gorjetas nesse dia. Na Figura 5 pode-se observar a estrutura desta *query*:

- **PleasantDriverStr**: Esta *query* aplica uma janela em *batch* de 24 minutos. Após isto, agrupa por número de carta de condução (identificação do taxista) e calcula a soma das gorjetas de cada taxista. Finalmente, faz *output* apenas do *top* 10 dos taxistas com a quantia mais elevada de gorjeta.

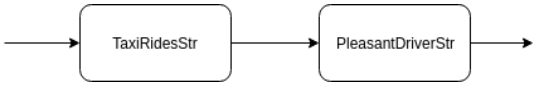

Fig. 5. Diagrama da query *Most Pleasant Taxi Driver*.

O código SiddhiQL que implementa esta *query* é o seguinte:

```
from TaxiSecStr#window.timeBatch(24 min)
select hack_license,
       sum(tip_amount) as tips_total
group by hack_license
order by tips_total DESC
limit 1
insert into PleasantDriverStr;
```

IV. Visualização

Como foi referido na secção III., os streams produzidos pelas *queries* são publicados no *broker* Kafka, sendo que cada uma das *queries* tem um tópico diferente. Esta arquitectura permite recorrer ao Jupyter Notebook para ajudar na visualização em *real-time* das *queries*. O *notebook* desenvolvido tem duas componentes principais: consumidor da stream produzida pelas *queries* e uma outra com a visualização. Em relação a esta última, recorreu-se a um cliente Python do Google Maps. Sendo assim possível, por exemplo no caso da *query* das *Profitable Areas*, identificar as 10 áreas através de um *marker* no centro destas.

V. Agradecimentos

Uma palavra de agradecimento ao Daniel Murteira pois foi ele que ajudou na tarefa de ligar o Kafka ao WSO2 CEP, a qual se tornou no principal percalço deste projecto.

VI. Conclusão

Comparando o SparkStreaming e o SiddhiQL em termos de manutenção, o segundo, como apresenta uma linguagem de alto nível, permite que os projectos escalem em complexidade mantendo a simplicidade do código.

Em termos da flexibilidade das *queries*, o SiddhiQL permite a identificação de padrões e sequências, algo que em SparkStreaming seria mais complicado simular. A verdade é que o SiddhiQL sendo uma linguagem mais alto nível, é bastante mais expressiva.

Um dos principais problemas que se teve ao desenvolver a solução para estas *queries* foi o facto da documentação ser muito reduzida. Neste aspecto perde claramente em relação ao SparkStreaming, no qual a quantidade e qualidade de

informação disponível é bastante mais elevada. Um dos problemas que surgiu deste facto foi a dificuldade em conseguir conectar o Kafka ao WSO2 CEP.

Outro dos obstáculos à implementação destas *queries* foi o Stream Processor Studio, no qual foram implementadas. O editor deste, muitas das vezes, alertava para erros que mais tarde (sem qualquer alteração) desapareciam aleatoriamente. Sendo que estes erros não eram suficientemente informativos para se perceber o que estava errado.

Por fim, ao longo do desenvolvimento o desconhecimento da implementação deste CEP foi uma dificuldade. Esta surgiu principalmente em identificar as *queries* que precisavam de janelas. A verdade é que se assumiu que a implementação é semelhante à do CQL (estudada na aula).