Capítulo 2

Introdução ao processamento de fluxo de dados: uma abordagem orientada a eventos complexos

Marcos Roriz Junior[1], Álan L. Vasconcelos[2], Fernando B. V. Magalhães[2], Sérgio Colcher[2], Markus Endler[2]

[1]Faculdade de Ciências e Tecnologia – Engenharia de Transportes, UFG

[2]Departamento de Informática, PUC-Rio

marcosroriz@ufg.br, alan@telemidia.puc-rio.br, fmagalhaes@inf.puc-rio.br

colcher@inf.puc-rio.br, endler@inf.puc-rio.br

Abstract

Most of information technologies courses are focused on exposing the traditional static data (e.g., SGBD) processing model and lack on presenting the data stream model. Such model can be applied to fields such as Internet of Things, busyness and finances, logistics and smart cities) of the industry. Complex Event Processing consists of a programming approach to handle Data Stream Processing. It provide primitives to process and detect the occurrence of patterns in data streams. This chapter has the objective of presenting the complex event processing (CEP) programming model as a means of dealing with the specificities of data flows.

Resumo

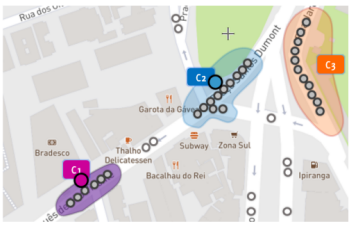
A maioria dos cursos de tecnologias da informação está focada em expor o modelo de processamento tradicional de dados estáticos (e.g., SGBD) e a falta de apresentação do modelo de fluxo de dados. Esse modelo pode ser aplicado a áreas como Internet das Coisas, ocupação e finanças, logística e cidades inteligentes) da indústria. O processamento complexo de eventos consiste em uma abordagem de programação para lidar com o processamento de fluxo de dados. Ele fornece primitivas para processar e detectar a ocorrência de padrões nos fluxos de dados. Este capítulo tem o objetivo de apresentar o modelo de programação de processamento de eventos complexos (CEP) como um meio de lidar com as especificidades dos fluxos de dados.

2.1. Introdução

Avanços técnicos na arquitetura de computadores e computação móvel têm possibilitado não somente a popularização de dispositivos portáteis e embarcados, com conectividade com a Internet e capacidade de sensoriamento e atuação, mas também a utilização dos mesmos para a construção de aplicações de Cidades Inteligentes (*Smart City*) e Internet das Coisas (*Internet of Things*) [van der Zee and Scholten 2013].

Coletivamente, esses dispositivos podem produzir grandes fluxos de dados, que ao serem interconectados possibilitem gerar uma inteligência competitiva ou detectar situações de interesse [Zheng et al. 2014, McAfee and Brynjolfsson 2012]. Por exemplo, pode-se utilizar sensores para controlar plantas na indústria (Figura 2.1a), detectar engarrafamento de veículos a partir de dados de localização (Figura 2.1b) e monitorar a reputação de uma empesa em redes sociais a partir de mensagens dos usuários. Desta forma, cada vez mais é importante explorar técnicas que possibilitem não somente a extração de informação, mas também a reação a anomalias em fluxo de dados.

(a) Sensor de temperatura fixo[1]



(b) Engarrafamento [Roriz Junior 2017]

Figura 2.1: Cenários de uso de aplicações orientadas a fluxo de dados.

Para tais tarefas existem várias tecnologias focadas na análise de fluxos de dados, as quais vão desde bancos de dados ativos até o processamento de eventos complexos. O processamento de eventos complexos [Luckham 2001, Etzion and Niblett 2010], também conhecido como CEP (Complex Event Processing), consiste em um modelo de programação que fornece primitivas para processar e derivar eventos (informações) mais complexas a partir de fluxos de dados.

Em uma indústria, por exemplo, o evento de incêndio pode ser gerado a partir de eventos de aumento de temperatura e detecção de Fumaça. Em *Smart Cities*, por exemplo, o evento de engarrafamento pode ser detectado a partir de eventos mais simples que indicam que vários veículos não estão se movendo em um determinado período de tempo.

Neste contexto, este capítulo tem o objetivo apresentar e exemplificar o modelo de programação de processamento de eventos complexos (CEP) como meio de lidar com as especificidades de fluxos de dados. O capítulo está organizado da seguinte maneira. A Seção 2.2 apresenta os conceitos e problemas fundamentais de processamento de fluxo de dados e o modelo CEP. Já a Seção 2.3 apresenta a tecnologia Esper e a sua linguagem de regras EPL como meio de tratar os fluxo de dados. A Seção 2.4 exemplifica um caso de uso da aplicação de CEP para tratamento de dados de aplicações IoT, enquanto que a Seção 2.5 apresenta um outro caso de uso de aplicação de CEP como meio de processar

[1] `https://www.electronicdesign.com/iot/wireless-sensor-networking-industrial-iot`

dados de aplicações para *Smart Cities*. Por fim, a Seção 2.6 apresenta nossas considerações finais.

2.2. Fundamentação teórica e introdução a CEP

O Processamento de Eventos Complexos (*Complex Event Processing* - CEP) é um paradigma de programação voltado a tratar, analisar e reagir a um fluxo de dados (encapsulados como eventos) em aproximadamente tempo real, isto é, em poucos segundos [Luckham 2001, Kudyba 2014, Flouris et al. 2016]. Por exemplo, utilizando os conceitos de CEP é possível detectar, em aproximadamente tempo real (poucos segundos), engarrafamentos e acidentes no trânsito [Dunkel et al. 2011, Roriz Junior et al. 2019], padrões em redes sociais [Cameron et al. 2012, Yadranjiaghdam et al. 2017] e anomalias em fluxos de áudios de usuários [Maison et al. 2013].

Para possibilitar a detecção em tempo real, ao contrário dos sistemas gerenciadores de banco de dados, nos quais os dados são primeiramente armazenados e depois consultados posteriormente, o CEP inverte esta lógica armazenando as consultas continuamente e executando os dados diretamente nelas. Precisamente, em vez de armazenar os dados, o CEP implementa as consultas em um estado contínuo, sempre funcionando, com intuito de analisar e reagir aos eventos ao passo que os mesmos apareçam no fluxo de dados. Cada consulta implementa continuamente uma ou mais primitivas de processamento de fluxo de evento em tempo real, como *filtro*, *sequência* e *negação* [Cugola and Margara 2012, Suhothayan et al. 2011].

Os fluxos de eventos são as principais fontes de entrada de uma consulta contínua do CEP. No CEP, os eventos são criados por meio de produtores, que são entidades (*e.g.*, sensores, usuários) que encapsulam os dados do domínio da aplicação, como localização, imagem, texto e *clicks*. Estes eventos são denominados de eventos simples (*raw events*), visto que ainda não foram processados. Os eventos são caracterizados por um tipo, a hora que foi gerado (*timestamp*) e uma carga de dados (*payload*) [Luckham 2001]. O tipo do evento define o esquema do *payload*, isto é, o nome e o domínio dos atributos correspondentes que representam a ocorrência do evento em questão. Por exemplo, podemos definir o tipo de evento `LocationUpdate` para representar a atualização da posição de um objeto em movimento usando o seguinte esquema de carga útil: *id*, latitude, longitude e velocidade. Já o esquema de eventos originados em uma rede social pode conter o nome do usuário, o texto da mensagem transmitida, imagens anexadas, localização e tempo de envio.

Um fluxo de dados de eventos é a sequência resultante dos eventos criados e transmitidos pelos produtores [Luckham 2001, Etzion and Niblett 2010]. Neste sentido, os eventos em um fluxo de dados seguem o mesmo tipo e sua ordem é baseada no tempo de origem (*timestamp*) do mesmo. Desta maneira, uma consulta contínua pode utilizar as primitivas de processamento em tempo real, como *filtro*, *divisão* e *sequência*, para reagir e processar o fluxo de eventos à medida que este chegam. Por exemplo, podemos utilizar a primitiva de *filtro* para reter o fluxo de evento `LocationUpdate` com intuito de descobrir os veículos que se encontram próximos de um determinado ponto de interesse. Semelhantemente, uma empresa área pode utilizar a primitiva *dividir* para continuamente extrair e analisar as palavras-chave de mensagens de uma rede social que contenham o

nome da companhia.

As consultas contínuas podem usar várias primitivas em tempo real para reagir, processar e derivar outros eventos (complexos) de nível superior a partir de fluxos de eventos puros. Os eventos de saída das consultas contínuas são dito complexos, pois representam eventos processos e contém informações derivadas [Luckham 2001]. Ambos, eventos simples e complexos, podem ser usados como parte da definição de um outro evento complexo. Precisamente, é possível criar hierarquias de eventos, nas quais eventos intermediários podem ser usados para definir outros eventos complexos de nível superior. Por exemplo, um evento complexo `TrafficJam` pode ser criado pela combinação de vários eventos `LocationUpdate` na mesma área e período. Outro exemplo, podemos construir um evento complexo `PossibleDelay` a partir da composição de múltiplas mensagens de uma rede social que contenham palavras-chave associadas ao atraso de vôos de uma companhia em um pequeno espaço de tempo.

Cada consulta contínua é executada por um Agente de Processamento de Eventos do CEP, também conhecido por *Event Processing Agent* (EPA), que podem ser interligados para processarem os eventos [Suhothayan et al. 2011, Cugola and Margara 2012]. Especificamente, um estágio EPA continuamente faz as seguintes tarefas: reage aos eventos recebidos; analisa e manipula-os; e gera eventos complexos (derivados) para consumidores de eventos, podendo ser outros estágios EPAs ou aplicações finais. Ao interconectar os EPAs é possível construir uma rede de processamento de eventos (*Event Processing Network* - EPN), um *workflow* topológico que analisa o fluxo de eventos de entrada à medida que passa. A estrutura topológica da EPN, um gráfico direcionado, facilita a distribuição do processamento de EPAs para diferentes máquinas. Cada EPA nessa rede é responsável por receber eventos, processar a consulta contínua e, se houver uma etapa derivativa, enviar os eventos complexos aos próximos estágios.

Para exemplificar esses conceitos considere a EPN ilustrada na Figura 2.2 que visa detectar anomalias no trânsito a partir da localização dos veículos e de uma rede social. No cenário descrito, os EPAs de borda da EPN recebem continuamente a localização de cada ônibus da cidade, junto com mensagens dos usuários da rede social. Se os eventos

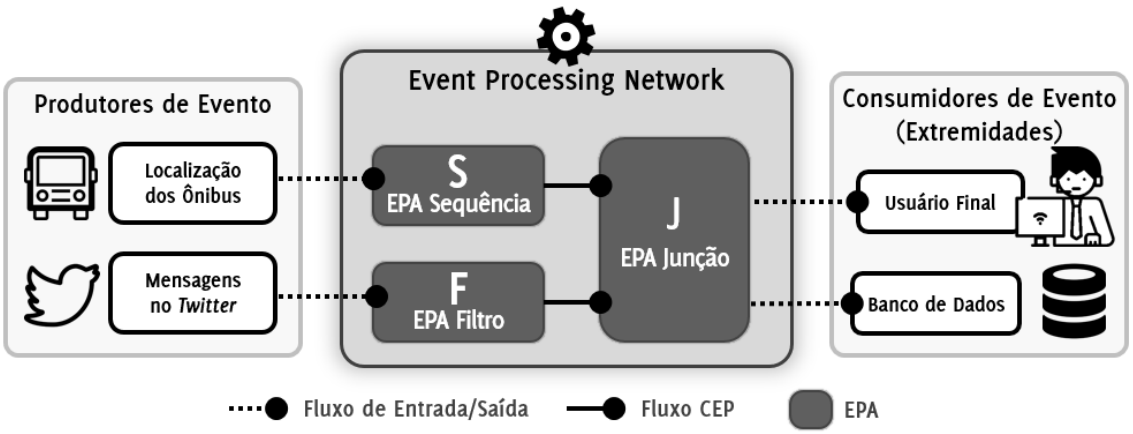


Figura 2.2: Exemplo de EPN para detecção de anomalias no trânsito.

analisados pelos EPAs satisfizerem a lógica da primitiva de processamento, o EPA emitirá um evento complexo como saída. Tais eventos são entregues a EPAs interessados (conectados) para serem processados em sequência.

Por exemplo, a unidade de processamento `S` identifica se os ônibus estão parados (ou andando devagar) verificando se a distância percorrida pelo mesmo, na sequência de eventos de localização enviados, for menor que um limiar (*e.g.*, 250 metros). Caso positivo, o EPA deriva um evento complexo agregando o identificador do veículo e a sequência de posições. Paralelamente, o EPA `F` filtra as mensagens da rede social que contenham palavras-chave relacionados a anomalias no trânsito, tais como acidente e engarrafamento, que por sua vez são adicionalmente processadas se possuírem um valor maior que um determinado limiar dentro de um período (*e.g.*, 50 mensagens nos últimos 5 minutos). Finalmente, `J` correlaciona os dados de ônibus que não se moveram com o grupo de mensagens com palavras-chave que ocorreram na mesma região espacial. Como saída, o EPA gera um evento complexo de possível anomalia de trânsito, que consequentemente, é repassado para entidades interessadas na borda da EPN, como o usuário final ou banco de dados.

2.2.1. Motor de Inferência CEP e Linguagens de Consultas Contínuas

Um motor de inferência (*engine*) CEP permite instanciar os conceitos apresentados. Ele fornece operações para definir os tipos de eventos (esquema e carga útil) e primitivas em tempo real para expressar as consultas contínuas (EPAs) e para interconectá-las (EPN). Há várias implementações de moto de inferência CEP, por exemplo, Esper [EsperTech 2019], Siddhi [Suhothayan et al. 2011], STREAM [Arasu et al. 2003], Microsoft StreamInsight [Ali et al. 2011], Sase+ [Yanlei Diao Neil Immerman and Gyllstrom 2007], Apache Flink [Carbone et al. 2015] e Red Hat Drools Fusion [Bali 2009].

As principais diferenças entre as *engines* são os construtores e semântica das linguagens fornecidas aos desenvolvedores para definir os eventos e primitivas de processamento em tempo real. Em geral, as semânticas das linguagens CEP podem ser divididas em duas categorias: orientada a fluxo e orientado a regras [Etzion and Niblett 2010, Cugola and Margara 2012, Kudyba 2014, Zhao et al. 2017].

As linguagens de processamento de eventos complexos orientadas a fluxos tipicamente são baseadas em um design formal da linguagem de consulta contínua, também denominada *Continuous Query Language* (CQL) [Arasu et al. 2005]. A CQL é uma extensão da linguagem *Structured Query Language* (SQL) com operadores e primitivas específicas para tratamento e análise de fluxo de dados.

Para ilustrar a expressividade do modelo de linguagem orientada a fluxo do CEP, considere a consulta contínua escrita na linguagem EPL (*Event Processing Language*) do motor de inferência CEP Esper [EsperTech 2019] no Código 2.1. A EPL descreve um EPA que detecta um evento complexo `AlertMessage` quando existir mais de 50 mensagens em uma janela de tempo de 60 segundos contendo as palavras-chave `acidente` ou `engarrafamento` e que estejam dentro do intervalo delimitado (*e.g.*, limites do município).

Para implementar esse EPA, o motor de inferência CEP realiza os seguintes passos. Ao receber um evento do fluxo `TwitterMessages`, a consulta utiliza o *timestamp* do mesmo para deslizar a janela de tempo e recuperar todos os eventos recebidos nos últimos

60 segundos. O resultado desta operação, um subconjunto do fluxo `TwitterMessages`, é transformado em uma relação temporária. Utilizando essa relação, a consulta contínua filtra os eventos cujos valores de carga útil possuem pelo menos uma das palavras-chave pré-determinada e estão dentro dos intervalos de latitude e longitude especificado. Em seguida, a consulta contínua conta o número de tuplas filtradas. Caso este número seja superior a 50 mensagens, será gerado um novo evento complexo `AlertMessages` contendo a projeção (seleção) das palavras-chave e localização dos eventos resultantes do filtro.

```
INSERT INTO AlertMessages
SELECT keyword, lat, lng
  FROM TwitterMessages#TIME(60 s)
 WHERE keyword IN ("acidente", "engarrafamento")
   AND (lat > −21 AND lat < −23 AND lng > −42 AND lng < −43)
HAVING COUNT(*) > 50;
```

Código 2.1: Exemplo de EPA escrito na linguagem EPL da *engine* Esper.

Observe que as consultas contínuas de uma EPA, escritas em uma linguagem orientada a fluxo, podem implementar uma ou mais primitivas em tempo real. No caso ilustrado, a consulta EPL implementa as primitivas *filtro*, *agregação* e *projeção*. Além disso, como dito, observe que a sintaxe da linguagem EPL (baseada em CQL) é semelhante a uma consulta SQL. Uma das diferenças é que a saída de uma consulta contínua produz um fluxo de eventos que, consequentemente, pode ser consumido e processado por outras consultas contínuas. Nesse caso, o EPA produz continuamente eventos `AlertMessages` que podem ser analisados por outras consultas. Outra diferença frente ao SQL padrão é a capacidade de usar janelas de tempo para analisar e correlacionar eventos recebidos em subconjuntos do fluxo. Abordaremos o conceito de janelas de tempo em uma subseção posterior deste capítulo.

Ao contrário das linguagens semelhantes a CQL, *engines* CEP que empregam linguagens orientadas a regras são baseados em cláusulas de inferência evento-condição-ação (ECA) [Wu et al. 2006, Anicic et al. 2010, Zhao et al. 2017]. Regras ECA separam a manipulação de eventos, a verificação de condições e a ação em diferentes cláusulas. Primeiro, a consulta contínua da EPA é acionada quando o evento especificado acontece. Em seguida, uma restrição ou condição é verificada. Finalmente, se a condição for atendida, as regras executam a cláusula de ação.

Para exemplificar uma linguagem orientada a regra, considere a definição do EPA ilustrado no Código 2.2. É uma consulta contínua semelhante ao exemplo anterior, descrito em EPL, mas agora escrita na linguagem de regras da *engine* CEP da plataforma Red Hat Drools Fusion [Bali 2009]. A regra é acionada ao chegar um novo evento do fluxo `TwitterMessages` que possua pelo menos uma das palavras-chave mencionadas e esteja dentro da área de interesse. Em seguida, a *engine* retorna o acúmulo de mensagens do fluxo que situam no mesmo intervalo de localização e contenham as palavras-chave indicadas. Após essa etapa, é feito a contagem de mensagens (`$numMsgs`) que passaram

pelo respectivo filtro. Caso possua mais de 50 eventos a regra prossegue para a etapa de consequência (`THEN`). Em seguida realizamos a projeção dos atributos `keyword`, `lat` e `lng` dos eventos para criar a lista resultante `$listMsgs`. Por fim, utilizamos a lista de eventos filtrada na etapa anterior para construir e enviar o evento complexo `AlertMessages`.

```
RULE "Detectar possível anomalia no trânsito em uma dada região"
  WHEN
    TwitterMessages (keyword IN ("acidente", "engarrafamento")
                     AND (lat > -21 AND lat < -23)
                     AND (lng > -42 AND lng < -43))
    ACCUMULATE (
      $msgs = TwitterMessages (keyword IN ("acidente", "engarrafamento")
                               AND (lat > -21 AND lat < -23)
                               AND (lng > -42 AND lng < -43)
                               OVER WINDOW:TIME(60 s),
                               $numMsgs = COUNT($msgs);
                               $numMsgs > 50)
    )
  THEN
    $listMsgs = collectList($msgs.keyword, $msgs.lat, $msgs.lng);
    INSERT(AlertMessages($listMsgs))
END
```

Código 2.2: Exemplo de EPA escrito na linguagem EPL da *engine* Red Hat Drools.

Como dito, a principal diferença entre as semântica das linguagens CEP é a sintaxe fornecida para criar consultas contínuas. Precisamente, aquelas orientadas a fluxo são baseadas na CQL (uma extensão da SQL), enquanto as orientadas a regras usam cláusulas evento-condição-ação (ECA) para processar os eventos recebidos. Note que os dois modelos *podem* implementar as mesmas primitivas CEP. Por exemplo, tanto o Código 2.1, quanto o Código 2.2, implementam as primitivas de *filtro*, *agregação* e *projeção*.

No entanto, na prática, as *engines* CEP suportam um conjunto diferente de primitivas de processamento [Cugola and Margara 2012]. Em geral, os motores CEP que utilizam linguagens orientadas a fluxos se concentram no suporte às primitivas de *transformação*, tais como *filtrar*, *dividir* e *combinar*, enquanto as orientadas a regras são focadas em primitivas de detecção de padrão, como *sequência* e *negação* de padrões de eventos.

A Figura 2.3 ilustra uma classificação taxonômica das principais primitivas CEP, adaptadas de [Etzion and Niblett 2010, Cugola and Margara 2012], que são tipicamente suportadas em cada modelo semântico do CEP. Entretanto, ressaltamos que alguns motores de inferência CEP, tais como Esper [EsperTech 2019], Microsoft StreamInsight [Microsoft 2015] e Apache Flink [Carbone et al. 2015] oferecem suporte aos dois tipos semânticos para construção das consultas contínuas. Na subseção a seguir, definiremos as classes de primitivas e apresentaremos concisamente a função de cada uma delas.

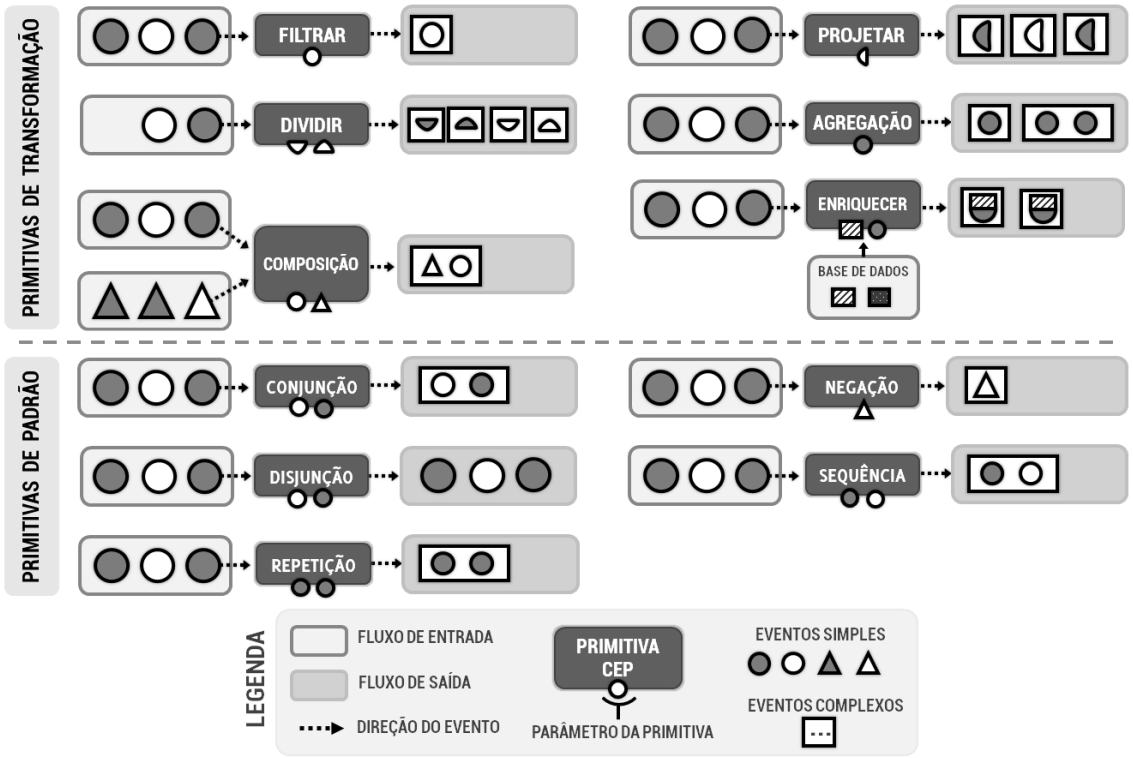


Figura 2.3: Taxonomia das primitivas CEP.

2.2.2. Primitivas CEP

As primitivas de transformação CEP são aquelas que filtram, modificam ou correlacionam os eventos do fluxo de dados [Luckham 2001, Anicic et al. 2010, Cugola and Margara 2012]. Elas podem converter o evento de entrada em um complexo usando os atributos do mesmo ou uma fonte de dados externa. Como visto ao longo do texto, a primitiva de *filtrar* possibilita reter ou passar um conjunto de eventos conforme um ou mais predicados. Já a primitiva de *dividir* possibilita transformar um evento em vários. Por exemplo, podemos utilizar a primitiva para separar uma mensagem de uma rede social em múltiplos eventos complexos com base nas palavras-chave da mesma.

Como na álgebra relacional, a primitiva *projetar* cria um evento complexo usando um subconjunto de seus atributos, enquanto a primitiva *enriquecer* utiliza uma fonte de dados externa (*e.g.*, tabela na memória ou em um banco de dados) para criar um evento derivado contendo novas informações ou modificações de atributos originais [Luckham 2001]. Por exemplo, um evento de ocorrência de engarrafamento pode ser enriquecido com pontos de interesses de um banco de dados para verificar se os mesmos se encontram próximo (*e.g.*, acidente próximo ao aeroporto).

Já as primitivas de agregação permitem combinar eventos de entrada em um único de saída, por exemplo, múltiplas mensagens de reclamação podem ser combinada em um evento de alerta. Além disso, essa primitiva pode empregar as funções clássicas de agregação do SQL, tais como média (*avg*), mínimo (*min*) e máximo (*max*). A primitiva de composição possibilita realizar a junção entre dois ou mais fluxos de eventos de entrada,

utilizando um critério, para derivar eventos complexos oriundos dessa correspondência. Como previamente exemplificado, podemos combinar os fluxos de eventos de localização de veículo com as mensagens de acidentes em redes sociais.

Por outro lado, as primitivas de detecção de padrões são baseadas em modelo de regras (*templates*) de evento [Wu et al. 2006, Cugola and Margara 2012]. Tais *templates* usam operadores lógicos – como *conjunção*, *disjunção*, *repetição*, *negação* e *sequência* – para definir o padrão que será utilizado pela consulta contínua [Anicic et al. 2010].

Por exemplo, a primitiva de *repetição* define um padrão de evento que detecta quando um determinado tipo de evento se repete pelo menos n vezes em uma determinada janela de tempo. Por exemplo, considere a primitiva de repetição com o seguinte padrão de evento E [n] [3 min]. Essa primitiva consome e processa continuamente eventos até encontrar pelo menos n eventos do tipo E dentro no período especificado. Podemos utilizar tal primitiva para detectar surtos de mensagens contendo palavras-chave semelhante em um pequeno espaço de tempo em uma rede social.

Em seguida, destacamos a primitiva *conjunção* que possibilita definir um padrão de evento que devem ocorrer em conjunto. Para tal, devemos utilizar o seguinte *template*: E_1 `AND` E_2 `AND` ... `AND` E_n. Esta primitiva continuamente analisa o fluxo a procura do padrão, sendo ativada quando encontrar os eventos E_1, E_2, ..., e E_n, no mesmo. Note que a definição não impõe uma ordem, isto é, a regra não depende da ordem de chegada da aparição dos eventos. Para ilustrar essa primitiva considere o seguinte *template*: *Atraso* [50] [30 min] `AND` *ProblemaAeroporto (nome = "Congonhas")*. A consulta contínua detectará o padrão quando observar pelo menos 50 eventos do tipo *Atraso* em meia hora e um evento que indica um problema no aeroporto de Congonhas. Como resultado, as primitivas de padrão criam um evento complexo contendo os elementos que dispararam a mesma.

Semelhante à primitiva de *conjunção*, a *disjunção* define um padrão de evento usando o operador lógico OU (`OR`), por exemplo, E_1 `OR` E_2 `OR` ... `OR` E_n. A *disjunção* é satisfeita quando a consulta contínua detecta pelo menos um dos eventos especificados no *template*.

O CEP também provê outros operadores de relações de padrão, como a *negação* e *sequência*. A primitiva de *negação* define um padrão que detecta a ausência de um determinado evento. Por exemplo, o padrão $\neg E$ [10 min] significa que a consulta contínua deve ser acionada se nenhum evento E for detectado dentro de 10 minutos. Para exemplificar o funcionamento dessa primitiva, considere o seguinte padrão: $\neg$*ProblemaAeroporto (nome = "Congonhas")* [30 min]. Essa consulta contínua será acionada se não receber mais nenhum evento *ProblemaAeroporto*, cuja carga útil se refere ao aeroporto de Congonhas, em um período de 30 minutos.

A primitiva de *sequência* define um *template* para capturar o relacionamento entre os eventos, especificamente a ordem de chegada e tipo dos mesmos. Por exemplo, uma consulta contínua contendo o padrão $E_1 \rightarrow E_2 \rightarrow E_1 \rightarrow E_3$ é acionada quando a mesma recebe os seguintes eventos em ordem E_1, E_2, outro E_1 e E_3. Para ilustrar essa primitiva, considere um exemplo onde pretendemos detectar o surgimento e término de engarrafamentos no trânsito a partir de mensagens de uma rede social. O padrão *Acidente* $\rightarrow$ *MsgEngarrafamento* [50] $\rightarrow$ *Engarrafamento* $\rightarrow \neg$*Engarrafamento* [60 min], inicia a de-

tecção a partir de um evento de acidente, seguido de 50 mensagens de congestionamento. Após isso, o mesmo verifica pela detecção de um evento do tipo *Engarrafamento*. Depois disso, o padrão procura a ausência de um evento de congestionamento por 60 minutos. Essa situação pode capturar a indicação de que o engarramento gerado por um dado acidente.

2.2.3. Contexto e Janelas de Tempo

Várias primitivas apresentadas requerem o conceito de uma janela, seja de tempo ou de tamanho, para processar o fluxo de eventos recebidos. Por exemplo, a primitiva *agregação* permite combina eventos correlatos que ocorrem dentro de um subconjunto do fluxo em um único evento complexo. Para tal, o CEP fornece meios de agrupar eventos relacionados em um *contexto* (janela) para permitir que eles sejam processados de maneira relacionada. Um contexto CEP subdivide o fluxo de eventos em uma ou mais partições [Luckham 2001, Etzion and Niblett 2010, Cugola and Margara 2012] usando predicados lógicos e/ou temporais. Desta maneira, cada partição de contexto representa um subconjunto do fluxo de eventos recebidos.

Por exemplo, considere a declaração da partição de contexto *MesmoVeículo* ilustrada na Figura 2.4. Esse contexto subdivide o fluxo de eventos *PosiçãoVeículo* de acordo com o identificador de cada veículo (utilizando o atributo `id`). A partição de contexto resultante gera vários subfluxos, de modo que todos os eventos localizados em uma determinada partição contenham o mesmo *id*, ou seja, contenham apenas eventos emitidos pelo mesmo veículo. Como a partição de contexto é um fluxo de eventos (um subconjunto), todas as primitivas de CEP funcionam normalmente nelas. Por exemplo, podemos utilizar o seguinte padrão de sequência, $MesmoVeículo \rightarrow \neg MesmoVeículo$ [2 min], para detectar a situação onde um dado veículo não transmite sua posição em um intervalo de 2 minutos.

Semelhantemente, podemos utilizar o conceito de contexto para particionar o fluxo de eventos de mensagens de uma rede social utilizando as palavras-chave da mesma. Nesse contexto, cada partição irá conter apenas mensagens que possuem a palavra-chave em questão.

As janelas de tempo são partições de contexto. Precisamente, uma janela de tempo é um contexto temporal que subdivide o fluxo de evento em intervalos de tempo utilizando o atributo *timestamp* de cada evento [Luckham 2001, Cugola and Margara 2012]. Desta maneira, a janela irá particionar o fluxo para incluir apenas os eventos cuja chegada esteja no intervalo especificado, *e.g.*, $(t - \Delta, t)$ onde t é a hora atual [Matysiak 2012, Amini et al. 2014]. Por exemplo, a declaração da janela de tempo *PosiçãoVeículo* [*Deslizar* 30 seg] cria automaticamente uma partição de contexto temporal que retém apenas os eventos deste fluxo recebidos nos últimos 30 segundos. Quando uma EPA processa um evento com tempo de chegada t nessa janela, as primitivas irão processar apenas os eventos que estão no intervalo $(t - 30, t)$. O conceito de janela é útil, pois, além de permitir tratar e lidar com o possível crescimento infinito dos fluxos de dados, também possibilita que as primitivas considerem apenas os eventos mais atuais na etapa de processamento.

Os dois principais tipos de janelas de tempo no CEP são lotes (*landmark/batch*) e deslizantes (*sliding*) [Matysiak 2012, Arasu et al. 2005] como ilustrado na Figura 2.5. As

janelas *landmark* provê a capacidade de processar o fluxo de eventos em lotes. Para tal, a janela armazena temporariamente (em um *buffer*) todos os eventos recebidos durante um intervalo de tempo e aplica as consultas contínuas considerando todos os eventos do lote. Neste caso, haverá um atraso entre a hora de chegada do evento e o tempo de processamento. Por exemplo, na Figura 2.5 (a), enquanto os eventos E_1 e E_2 chegaram no tempo $t = 1$, eles serão processados somente quando o período do lote terminar ($t = 2$).

Ao armazenar temporariamente os eventos em um *buffer*, é possível usar os predicados da primitiva de agregação, como *min* e *max*, para sintetizar todo o conteúdo do lote em um evento complexo. No entanto, se os eventos que devem ser processados juntos, por exemplo, por uma primitiva de sequência, forem colocados em lotes adjacentes, a mesma falhará em detectar a correlação entre os eventos, uma vez que a mesma considera apenas o conteúdo em análise. Uma maneira de atenuar esse problema é aumentar o período do lote, mas isso causa um atraso adicional para processar os eventos.

As janelas deslizantes tratam desse problema ao mover a janela de tempo junto com os eventos recebidos. Portanto, em vez de ter períodos de lote predefinidos, a borda da janela deslizam conforme o evento em análise. Mais especificamente, podemos definir uma janela deslizante como uma janela de lote em movimento que contém os eventos dos últimos Δ unidade de tempo. Para ilustrar esse conceito, considere o fluxo de eventos com uma janela de tempo deslizante de 1 segundo na Figura 2.5 (b). Por exemplo, quando $t = 3$, a janela de tempo inclui os eventos do segundo passado ($t = 2$) até o horário atual.

O movimento da janela, que desliza automaticamente os limites da janela, atenua o problema de correlacionar eventos situados próximos, mas em diferentes lotes. Por exemplo, os eventos E_3 e E_4 são colocados em lotes diferentes mesmo estando próximos um do outro. Este problema seria mitigado ao utilizar uma janela deslizante. Precisamente, a movimentação da janela possibilita incluir os eventos adjacentes e, ao mesmo tempo, fornece um processamento e reação ao fluxo em aproximadamente tempo real.

É importante destacar a existência de outros tipos de janela, como as de decaimento e de pulo [Ali et al. 2011, Cugola and Margara 2012, Amini et al. 2014]. A janela

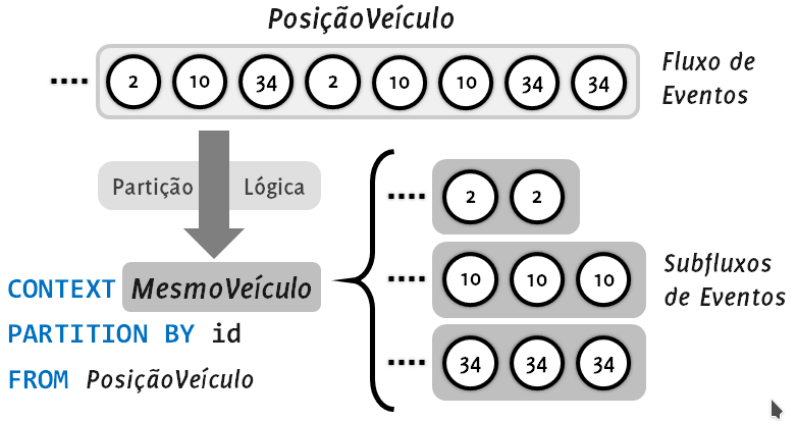


Figura 2.4: Exemplo de uma partição de contexto (*MesmoVeículo*) que subdivide o fluxo de eventos *PosiçãoVeículo* usando o valor *id* de cada evento.

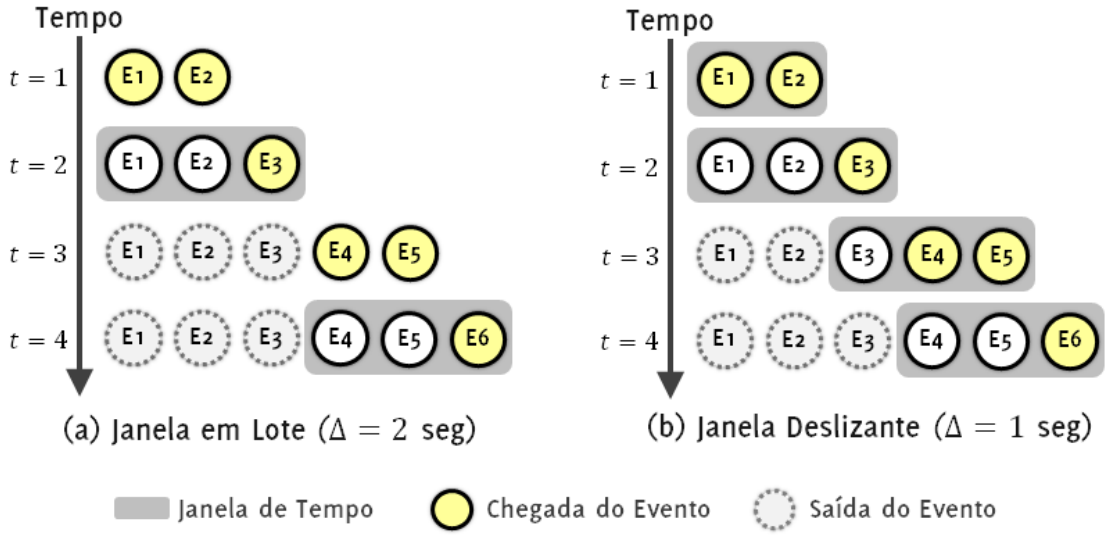


Figura 2.5: Exemplo de Janela em Lote e Deslizante.

de decaimento é uma variação de uma janela deslizante em que se aplica um fator de esquecimento λ aos eventos de acordo com a idade dos mesmos. Isto é feito para diferenciar a importância e a atualidade dos eventos na etapa de processamento, ou seja, eventos mais recentes têm maior importância que os eventos mais antigos. Para tal, é necessário que o usuário informe o peso λ e uma função de esquecimento f que determina como aplicar o peso aos eventos. No entanto, destacamos que o cálculo contínuo dos pesos de cada evento usando o fator de decaimento é muito caro para o processamento em tempo real dos dos fluxos [Amini et al. 2014].

Já a janela de pulo (*hopping*) mistura os conceitos de lote e de deslizamento [Ali et al. 2011, Cugola and Margara 2012]. Precisamente, a janela de pulo recebe dois atributos, b e h, sendo b o tamanho da janela em lote, h o tempo a ser deslizado e $h < b$. A idéia é que a janela de lote não deva pular diretamente para o próximo lote e sim deslizar em h unidades de tempo. Como $h < b$ a janela de lote irá avançar para um novo lote, mas irá levar consigo parte dos eventos acumulados anteriormente. Para exemplificar esta janela considere que a janela atualmente contenha os eventos no intervalo $(t-b,\ t)$. A próxima janela irá conter os eventos no intervalo $(t-b+h,\ t+h)$. Como $h < b$ a janela conterá eventos do lote anterior (especificamente do intervalo $(t-b+h,\ t)$.

Podemos ver que as janelas de tempo são um conceito central no processamento das primitivas do CEP. Entretanto, ressaltamos que nem todas as *engines* CEP fornecem suporte aos tipos de janelas vistas aqui. Na próxima seção será apresentado o motor de inferência Esper [EsperTech 2019], que suporta um bom conjunto das primitivas e janelas descritas.

2.3. A *engine* Esper e sua linguagem EPL

Esper é uma *engine* CEP de código aberto disponibilizada como um conjunto de bibliotecas Java. Isso significa que Esper pode ser embutida desde em aplicações para desktop

quanto em aplicações para servidores, essa característica dá flexibilidade a *engine* e permite por exemplo um prototipagem e testes locais anteriores ao *deployment*. Além de uma versão para .NET chamada Nesper, Esper possui um conjunto de adaptadores para entrada e saída de dados chamado EsperIO. Estes permitem acoplar a *engine* Esper diretamente a canais de comunicação como o Apache Kafka ou o protocolo HTTP. Nessa sessão apresentaremos de forma geral e daremos alguns exemplos de uso das bibliotecas que compõem a Esper e a EsperIO, adotamos a versão 8.2 da *engine*.

Os exemplos de código discutidos nesse capítulo estão disponíveis no repositório deste capítulo: https://github.com/fmagalhaes-inf-puc-rio/2019-CEP-Webmedia.

2.3.1. Arquitetura Geral

A *engine* Esper, desde a versão 8.0, consiste em uma linguagem (Esper EPL), um compilador (Esper Compiler) e um ambiente de execução (Esper Runtime).

- A Esper EPL é uma linguagem declarativa para descrever as regras CEP que definem o processamento de eventos. Ela é compatível com o padrão SQL-97 porém o estende com uma série de operadores focados no processamento de fluxos de eventos e detecção de padrões.
- O Esper Compiler converte regras EPL em bytecode java o que permite que as regras sejam processadas mais rapidamente no ambiente de execução.
- O Esper Runtime é executado sobre a máquina virtual java (JVM) tendo como entrada os eventos e como saída o resultado do seu processamento. A API java do Esper oferece métodos e interfaces para entrada de eventos assim como o consumo dos resultados porém também é possível utilizar a EsperIO que permite acoplar middleware diretamente a entrada e ou saída do Esper Runtime.

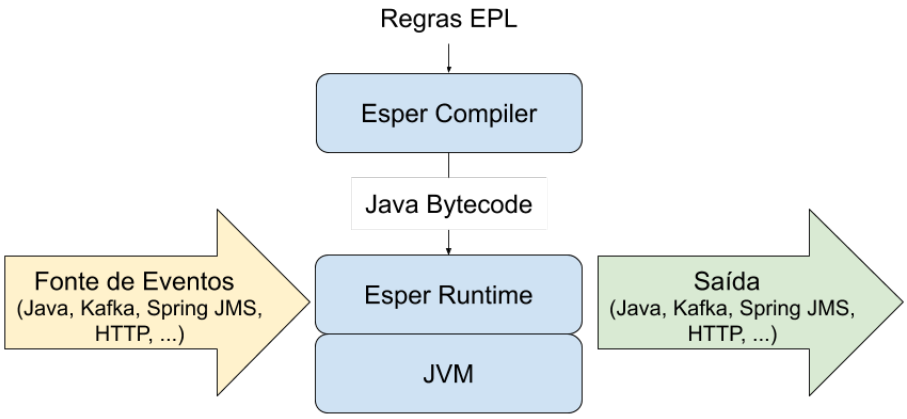


Figura 2.6: Arquitetura Geral da *engine* Esper.

2.3.2. Configuração do Motor CEP

A configuração permite especificar os seguintes parâmetros antes de iniciar o processamentos de eventos:

- Acesso a um Banco de Dados, permitindo que as regras EPL acessem dados contidos no banco e, ou insiram dados no banco;

- Importar classes e pacotes para o ambiente Esper permitindo que métodos declarados nessas classes sejam acessíveis dentro das regras CEP.

- Adição de tipos de eventos, o que também pode ser feito em tempo de execução a partir de regras EPL;

Essas configurações são feitas a partir da classe **Configuration**. É possível configurar a **engine** de forma programática no próprio código java como apresentado no código 2.3 ou a partir de ou carregar um aquivo de configuração em XML, como apresentado nos códigos 2.4 e 2.5.

```
Configuration c = new Configuration()
c.getCommon().addEventType(MyEvent.class);
c.getCommon().addImport("mypackage.MyClass");
Properties props = new Properties();
props.put("username", "myusername"); props.put("password", "mypassword");
props.put("driverClassName", "com.mysql.jdbc.Driver");
ConfigurationCommonDBRef cDB = new ConfigurationCommonDBRef();
cDB.setDataSourceFactory(props, BasicDataSourceFactory.class.getName());
c.getCommon().addDatabaseReference("mydb", cDB);
```

Código 2.3: Exemplo de configuração da *engine* ESPER como código Java.

```
Configuration c = new Configuration()
c.configure(myfile.xml);
```

Código 2.4: Exemplo de configuração da *engine* ESPER a partir de um arquivo XML.

```
<?xml version="1.0" encoding="UTF-8"?>
  <esper-configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.espertech.com/schema/esper"
    xsi:schemaLocation=
          "http://www.espertech.com/schema/esper/esper-configuration-8-0.xsd">
  <common>
    <event-type name="MyEvent" class="mypackage.MyEvent"/>
  </common>
</esper-configuration>
```

Código 2.5: Exemplo de arquivo de configuração XML.

2.3.3. Definição de regras em Esper EPL

A linguagem Esper EPL extende o padrão SQL-97, portanto operações padrões como `SELECT, FROM, WHERE, JOIN, CREATE, DELETE, ...` são suportadas. Por exemplo a regra a seguir é válida em Esper EPL e seleciona todo novo evento do tipo LeituraSensor que for recebido no Esper Runtime.

```
1 SELECT * FROM MeuEvento
```

Assumimos que o(a) leitor(a) já possui alguma familiaridade com a definição de consultas em SQL e apresentaremos os pontos onde a EPL Esper estende os padrões do SQL incluindo operadores idealizados para o processamento de fluxos de eventos.

2.3.3.1. Janelas de Eventos

Como os fluxos de eventos são potencialmente infinitos, é necessário fornecer um método para delimitar o escopo de uma regra a um faixa do fluxo, em Esper EPL isso é feito com janelas de eventos. Essas janelas podem ser delimitadas por tempo ou por número de eventos, além de processadas de forma deslizante ou em lotes.

Na regra a seguir, o símbolo # indica o início da definição de uma janela e o termo `LENGHT` denota que a janela é deslizante e delimitada por número de eventos. Nesse caso a cada novo evento do tipo `MeuEvento` recebido, a regra será ativada e retornará os valores dos atributos do novo evento além disso a regra retornará como valor de `avg1` a média do valor do `atributo1` nos últimos 10 eventos, ou seja o novo evento e os 9 eventos imediatamente anteriores se houverem.

```
1 SELECT *, AVG(MeuEvento.atributo1) AS avg1 FROM MeuEvento#LENGHT(10)
```

Para utilizar uma janela deslizante delimitada por tempo, bastaria trocar o termo `LENGHT` por `TIME` e incluir um unidade de tempo logo após o numeral 10. Se nenhuma unidade for especificada, o compilador de regras assume que o tempo está em segundos. Por exemplo, se substituíssemos o numeral `10` por `10 min` a consulta continuaria produzindo saídas a cada novo evento porém `avg1` passaria a conter a média do do valor do `atributo1` nos eventos recebidos no últimos 10 minutos. Ao utilizar janelas de processamento em lotes, as consultas passam a retornar resultados em lote ao invés de a cada evento recebido, ou seja cada vez que o lote é preenchido a regra retorna um resultado com todos os eventos do lote. As janelas de lote também podem ser delimitadas por tempo ou por número de eventos e para definir indicar uma janela de lote, basta adicionar o termo `BATCH` como na regra a seguir.

```
1 SELECT * FROM MeuEvento#TIME_BATCH(10)
```

Também é possível criar uma janela de eventos nomeada. Essa opção é especialmente útil quando múltiplas regra consultam a mesma janela de eventos. Para definir uma

janela de eventos basta usar a cláusula `CREATE WINDOW`, é necessário especificar o tamanho da janela, que assim como as janelas não nomeadas pode ser delimitado por tempo ou por número de eventos. A seguir um exemplo de regra que define uma janela nomeada que armazena 10 segundos de eventos.

```
1 CREATE WINDOW MeuEvento15sTIME(10) AS MeuEvento
```

O segundo passo é popular a janela isso é feito a partir de uma regra CEP que utiliza a cláusula `INSERT INTO`. Dessa maneira é possível filtrar os eventos que iram preencher a janela. Por exemplo, se janela deve conter apenas os eventos com `atributo1` maior que 5, pode-se fazer isso com a seguinte regra

```
1 INSERT INTO MeuEvento15s SELECT * FROM MeuEvento WHERE atributo1 > 5
```

Então é possível definir regras que consultam a janela nomeada. A regra abaixo por exemplo consulta os eventos dos últimos 15 segundos com atributo1 superior a 5 e atributo2 superior a > 20.

```
1 SELECT * FROM MeuEvento15s WHERE atributo2 > 20
```

2.3.3.2. Filtros

Filtros funcionam de forma semelhante a cláusula `WHERE`, permitindo especificar valores (ou faixas de valores) para as propriedades de um evento nas consultas. Porém enquanto um filtro restringe os eventos que entraram na janela, as cláusulas `WHERE` especificam dentre os eventos da janela quais serão retornados pela consulta. Por exemplo, as duas regras a seguir selecionam eventos com temperatura acima de 100:

```
1 SELECT temperatura FROM LeituraSensor(temperatura>100)#LENGHT_BATCH(10)
```

Código 2.6: Seleciona lotes de 10 eventos `LeituraSensor` com temperatura maior que 100, sempre retorna 10 eventos.

```
1 SELECT temperatura FROM LeituraSensor(temperatura>100)#LENGHT_BATCH(10)
```

Código 2.7: Seleciona dentre lotes de 10 eventos `LeituraSensor` aqueles com temperatura superior a 100, retorna 10 ou menos eventos.

2.3.3.3. Definições de padrões utilizando `PATTERN` e `MATCH_RECOGNIZE`

A EPL Esper permite descrever padrões de sequências de eventos utilizando cláusulas `PATTERN` ou `MATCH_RECOGNIZE`, elas possuem sintaxe e um conjunto de operadores diferentes. Como exemplo de regra de uso de `PATTERN` temos a regra abaixo, ela dispara quando um evento de `FornalhaLigada` não é seguida por um evento de `FornalhaDesligada` correspondente após passadas 4 horas.

```
SELECT a.*
FROM PATTERN
[
 EVERY a=FornalhaLigada -> (TIMER:INTERVAL(4 hours)
 AND NOT FornalhaDesligada(idFornalha=a.idFornalha))
]
```

As cláusulas `PATTERN` possuem 3 tipos de operadores:

1. Operadores de repetição:
   (a) `EVERY` seleciona todas as sequências de eventos que se encaixam em um padrão.
   (b) `EVERY-DISTINCT` análogo ao operador `EVERY` porém elimina resultados duplicados.
   (c) `[n]` seleciona n sequências de eventos que se encaixam em um padrão.
   (d) `UNTIL` especifica o fim de um padrão.
2. Operadores de sequência:
   (a) `->` (seguido por) define um padrão caracterizado pela ocorrência de uma sequencia de eventos em uma ordem especifica.
   (b) `OR` define um padrão caracterizado pela ocorrência de um ou outro evento.
   (c) `AND` define a ocorrência de uma sequência de eventos em qualquer ordem. Isso significa que `(EventoA AND EventoB)` é equivalente a `(EventoA->EventoB) OR (EventoB->EventoA)`.
3. Operadores de controle:
   (a) `TIMER:WITHIN(tempo)` serve para definir um padrão que deve acontecer dentro de um espaço de tempo.
   (b) `TIMER:INTERVAL(tempo)` define um intervalo de tempo que fará parte do padrão.
   (c) `WHILE` define que um padrão é valido apenas enquanto uma condição for verdadeira.

Já a especificação de padrões utilizando `MATCH_RECOGNIZE` é baseada em expressões regulares. Possuindo os operadores de:

- Agrupamento: `()`
- Quantificadores: `*, +, ?, {min, max}`
- Concatenação: `AB`
- Alternância: `|`

Como exemplo de regra de uso de `MATCH_RECOGNIZE` temos a regra a seguir, ela dispara quando um evento de `LeituraSensor` com temperatura superior a 50 é imediatamente seguido por um evento com temperatura ainda maior ou por um evento com radiação superior a 4.

```
SELECT * FROM LeituraSensor
MATCH_RECOGNIZE
(
   MEASURES A.temperatura AS aTemp, B.temperatura AS bTemp, C.radiacao AS cRad
    PATTERN (A (B | C))
      DEFINE A AS A.temperatura >= 50, B AS B.temperatura > A.temperatura,
             C AS C.radiacao >= 4
)
```

Outros exemplos de regras com `PATTERN` e `MATCH_RECOGNIZE` estão disponíveis no repositório deste capítulo.

2.3.4. Usando a biblioteca EsperIO

A biblioteca EsperIO provê um conjunto de adaptadores de entrada e saída pra o EsperRuntime, permitindo consumir ou exportar eventos diretamente de ou para arquivos, páginas HTML, Apache Kafka, ou outros. Nessa subseção apresentaremos um exemplo de uso dessa biblioteca para ler eventos de entrada de um arquivo e exportar eventos complexos gerados para arquivos de registro.

O objetivo desse exemplo é processar eventos do tipo `AtualizacaoSensor` com as propriedades `temperatura, Umidade, idSala, data_hora`. E produzir dois tipos de eventos complexos que representam estados de alerta: `BaixaUmidade` e `AltaTemperatura` que herdam as propriedades do evento simples.

Assumindo que uma leitura de umidade inferior a 35% é considerada baixa e uma temperatura acima de 35°C é considerada alta, utilizamos as seguintes regras para produzir o eventos complexos a partir de eventos de `AtualizacaoSensor`:

```
INSERT INTO BaixaUmidade -- Gerar eventos de BaixaUmidade
SELECT s.temperatura AS temperatura, s.umidade AS umidade,
       s.idSala AS idSala, s.data_hora AS data_hora
FROM PATTERN
[
 EVERY-DISTINCT(s.data_hora) s=SensorUpdate(humidade<0.35)
]
```

```
8
9  INSERT INTO AltaTemperatura -- Gerar eventos de AltaTemperatura
10 SELECT s.temperatura AS temperatura, s.umidade AS umidade,
11        s.idSala AS idSala, s.data_hora AS data_hora
12 FROM PATTERN
13 [
14    EVERY-DISTINCT(s.data_hora) s=SensorUpdate(temperatura>35)
15 ]
```

Agora o código EPL que define a leitura dos eventos de `AtualizacaoSensor` de um arquivo:

```
1  CREATE DATAFLOW SensorCSVEntrada
2     -- A regra define a geracao de um fluxo de evento a partir de um arquivo
3     FileSource -> sensorstream<SensorUpdate> {
4        --- O arquivo que sera lido
5        file: 'input.csv',
6        -- A ordem em que cada propriedade aparece em cada linha do arquivo csv
7        propertyNames: ['temperature','humidity','roomId', 'data_hora']
8     }
9  --Faz com que o EsperRuntime consuma o fluxo de eventos
10 EventBusSink(sensorstream){}
```

Por fim definimos o código que faz o registro dos eventos complexos:

```
1  CREATE DATAFLOW BaixaUmidadeCSVSaida
2     -- Coleta o fluxo de eventos de BaixaUmidade
3     EventBusSource -> outstream<BaixaUmidade> {}
4        -- Define uma saida em arquivo
5        FileSink(outstream) {
6        -- O arquivo onde sera salvo o registro
7        file: 'RegistroBaixaUmidade.csv'
8        -- Se o arquivo ja existir adiciona os valores novos no final
9        append: true
10    }
11 -- Fazemos o mesmo para os eventos de AltaTemperatura
12 CREATE DATAFLOW AltaTemperaturaCSVSaida
13    EventBusSource -> outstream<AltaTemperatura> {}
14       FileSink(outstream) {
15       file: 'RegistroAltaTemperatura.csv'
16       append: true
17    }
```

O código completo desse exemplo, incluindo a configuração necessária para usar a EsperIO está no diretório *EsperIODemo* do repositório online.

2.4. Estudo de Caso IoT: Monitoramento e Controle de temperatura

Apresentaremos aqui um exemplo onde CEP foi usado para monitorar e controlar a temperatura em um pequeno reator químico. Durante uma reação química exotérmica (que libera calor) para a produção de algum composto, é necessário um controle de resfriamento. A temperatura no recipiente deve ser mantida o mais próximo possível da temperatura ideal para a reação. Além disso é interessante um sistema de monitoramento que emita notificações caso essa temperatura sofra mudanças bruscas ou se distancie muito da ideal. O exemplo apresentado aqui foi testado em um modelo físico em um reator onde implementamos sistema que usa CEP para monitorar a temperatura e controlar o resfriamento de um reator. O resfriamento foi feito usando uma bomba que mantinha um fluxo de água gelada em uma camisa ao redor do reator.

Foi utilizado um sensor de temperatura no reator e outro no tanque de água gelada, os dados gerados por esses sensores foram encapsulados em eventos do tipo `EventoTemperatura` com as propriedades `temperaturaReator`, `temperaturaTanque` e `data_hora`. Como os sensores produzem dados com ruído, a primeira regra criada gera eventos com a média das últimas 5 leituras de modo a dissolver o ruído:

```
INSERT INTO TemperaturaMedia
SELECT AVG(temperaturaReator) AS temperaturaReator,
       AVG(temperaturaTanque) AS temperaturaTanque,
       CAST(AVG(data_hora), long) AS data_hora
FROM EventoTemperatura#LENGTH(5)
```

A bomba possui um potenciômetro para controlar a intensidade do fluxo de água. O ângulo dele é modificado sempre que a *engine* CEP produz eventos do tipo `MudarAngulo`. Esse evento foi definido usando o operador `CREATE SCHEMA` na regra a seguir:

```
CREATE SCHEMA MudarAngulo AS (angulo double, acao string, fonte string)
```

Nos eventos `MudarAngulo`, `acao` pode conter o valor “abrir” ou “fechar”, respectivamente representando um aumento ou diminuição do fluxo de água gelada. Existem dois tipos de regras controlam o sistema gerando esses eventos:

1. **Regras de incidente:** São disparadas caso de incidentes onde a temperatura chega a um nível de alerta ou crítico.

2. **Regras de ajuste fino:** São disparadas quando a temperatura está fora dos níveis de alerta ou crítico e buscam estabilizar a temperatura próximo de seu valor ideal.

2.4.1. Regras de incidente

Nesse caso de uso, o ângulo mínimo suportado pela bomba é 50° e o máximo é 170° e dada a temperatura ideal de 30°C considerou-se estado de alerta valores abaixo de 22.5°C ou acima de 37.5°C e estado crítico valores abaixo de 19.5°C ou acima de 40.5°C. As regras de estado critico buscam agir o mais rápido possível utilizando sempre o ân-

gulo máximo ou o angulo mínimo da bomba. Já as regras de alerta aumentam ou diminuem o ângulo em 5 graus sempre que uma delas é disparada, elas utilizam a variável `var_AnguloBomba` que armazena o valor atual do ângulo da bomba. Posteriormente explicaremos como o valor dessa variável é modificado automaticamente. As regras de incidente estão listada a seguir:

```
1
2  -- Regra para estado critico de temperatura baixa
3  INSERT INTO MudarAngulo
4  SELECT 50 AS angulo, "fechar" AS acao, "incidente" AS fonte
5  FROM TemperaturaMedia(temperaturaReator < 19.5)
6
7  -- Regra para estado critico de temperatura alta
8  INSERT INTO MudarAngulo
9  SELECT 170 AS angulo, "abrir" AS acao, "incidente" AS fonte
10 FROM TemperaturaMedia(temperaturaReator > 40.5)
11
12 -- Regra para estado de alerta de temperatura baixa.
13 INSERT INTO MudarAngulo
14 SELECT max(50, (var_AnguloBomba-5)) AS angulo, "fechar" AS acao,
15       "incidente" AS fonte
16 FROM TemperaturaMedia(temperaturaReator < 22.5 AND temperaturaReator >= 19.5)
17
18 -- Regra para estado de alerta de temperatura alta.
19 INSERT INTO MudarAngulo
20 SELECT min(170, (var_AnguloBomba+5)) AS angulo, "abrir" AS acao,
21       "incidente" AS fonte
22 FROM TemperaturaMedia(temperaturaReator > 37.5 AND temperaturaReator =< 40.5)
```

Além da variável `var_AnguloBomba`, temos a variável `var_UltimaAbertura` para armazenar se o último `MudarAngulo` com `acao` "abrir" foi gerado a partir uma regra de incidente ou de ajuste e a variável `var_UltimoFechamento` que armazena o mesmo para `acao` "fechar". As regras abaixo modificam o valor dessas variáveis sempre que há um novo evento `MudarAngulo` utilizando o comando `SET` dentro de uma cláusula `ON <EVENTO>`.

```
1  -- Modifica o valor de var_AnguloBomba e de var_UltimoFechamento
2  ON MudarAngulo(acao = "fechar") AS ma
3  SET var_AnguloBomba = ma.angulo, var_UltimoFechamento = ma.fonte
4
5  -- Modifica o valor de var_AnguloBomba e de var_UltimaAbertura
6  ON MudarAngulo(acao = "abrir") AS ma
7  SET var_AnguloBomba = ma.angulo, var_UltimaAbertura = ma.fonte
```

2.4.2. Regras de ajuste fino

As regras de ajuste fino tentam estabilizar a temperatura quando ela está subindo ou descendo além do valor ideal. Primeiramente, elas calculam a diferença entre o valor atual

da temperatura e o valor ideal. Então o ajuste feito no ângulo da bomba é determinado multiplicando essa diferença por quatro.

```
-- Regra para corrigir aquecimentos
INSERT INTO MudarAngulo
-- O angulo nao deve ultrapassar 170
SELECT min(170, var_AnguloBomba+((e2.temperaturaReator-30)*4)) AS angulo,
      "abrir" AS acao, "ajuste" AS fonte
-- A temperatura deve estar superior ao valor ideal, mas fora da faixa de alerta
FROM PATTERN [
  EVERY-DISTINCT(e2.data_hora) e1=TemperaturaMedia
  ->
  e2=TemperaturaMedia(temperaturaReator in (30.1, 37.5)
]
-- A temperatura deve estar subindo
WHERE e2.temperaturaReator > e1.temperaturaReator

-- Regra para corrigir resfriamentos
INSERT INTO MudarAngulo
-- O angulo nao deve pode ser inferior a 50
SELECT max(50, var_AnguloBomba-((30-e2.temperaturaReator)*4)) AS angulo,
      "fechar" AS acao, "ajuste" AS fonte
-- A temperatura deve estar inferior ao valor idel, mas fora da faixa de alerta
FROM PATTERN [
  EVERY-DISTINCT(e2.data_hora) e1=TemperaturaMedia
  ->
  e2=TemperaturaMedia(temperaturaReator in (22.5, 29.9)
]
-- A temperatura deve estar descendo
WHERE e2.temperaturaReator<e1.temperaturaReator
```

2.4.3. Regras de monitoramento

Além das regras de controle discutidas anteriormente também foram definidas regras de monitoramento, que geram notificações caso a temperatura irregularidades. Algumas das regras são muito semelhantes as regras de incidente, detectando quando a temperatura está em níveis críticos ou de alerta. Porém ao invés de gerar eventos `MudarAngulo` elas retornam os atributos do evento `TemperaturaMedia` que disparou a regra. Por exemplo a regra para níveis de alerta de temperatura baixa é a seguinte:

```
SELECT *
FROM TemperaturaMedia(temperaturaReator < 22.5 AND temperaturaReator >= 19.5)
```

Também foram definidas regras para detectar mudanças repentinas na temperatura que representem saltos para longe da temperatura ideal. Eles analisam os quatro últimos eventos de temperatura média e são acionadas se os eventos em sequência se afastam

cada vez mais do valor ideal e a temperatura do último evento difere em mais de 5% do primeiro.

```
-- Detecta aumento repentino
SELECT * FROM TemperaturaMedia
MATCH_RECOGNIZE
(
 MEASURES
    A.temperaturaReator AS temperaturaInicial,
    D.temperaturaReator AS temperaturaFinal,
    ((D.temperaturaReator-A.temperaturaReator)/A.temperaturaReator) AS aumento
 PATTERN (A B C D)
 DEFINE
    A AS A.temperaturaReator > 30
    B AS B.temperaturaReator > A.temperaturaReator
    C AS C.temperaturaReator > B.temperaturaReator
    D AS D.temperaturaReator > C.temperaturaReator
 AND D.temperaturaReator > (A.temperaturaReator * 1.05)
)
-- Detecta queda repentina
SELECT * FROM TemperaturaMedia
MATCH_RECOGNIZE
(
 MEASURES
    A.temperaturaReator AS temperaturaInicial,
    D.temperaturaReator AS temperaturaFinal,
    ((A.temperaturaReator-D.temperaturaReator)/A.temperaturaReator) AS queda
 PATTERN (A B C D)
 DEFINE
    A AS A.temperaturaReator < 30
    B AS B.temperaturaReator < A.temperaturaReator
    C AS C.temperaturaReator < B.temperaturaReator
    D AS D.temperaturaReator < C.temperaturaReator
 AND D.temperaturaReator < (A.temperaturaReator * 0.95)
)
```

2.4.3.1. Resultados obtidos

Nesse estudo de caso, as regras CEP conseguiram manter a temperatura no reator em uma faixa de 3°C ao redor do valor ideal. A figura 2.7 demonstra esses resultados. Nos testes iniciamos o controlador CEP com a temperatura do reator 18°C graus acima do valor ideal para verificar se o sistema conseguiria estabilizar a temperatura.

2.5. Estudo de Caso de Smart City (Cidades Inteligentes)

Com intuito de ilustrar os conceitos de CEP e da *engine* Esper descritos esta seção apresenta um estudo de caso de cidade inteligente que utiliza o fluxo de dados da posição dos

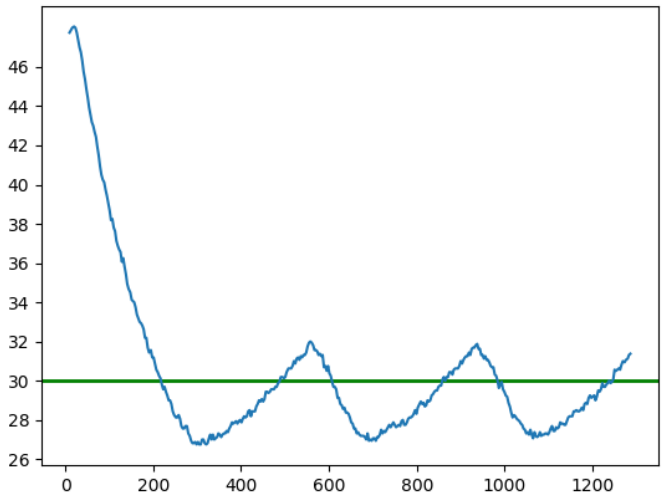


Figura 2.7: Temperatura ao longo do tempo usando as regras CEP para controle

ônibus da cidade do Rio de Janeiro para detectar possíveis locais de congestionamentos em tempo real. A arquitetura da aplicação, ilustrada pela Figura 2.8, descreve uma visão geral da interação entre as diferentes entidades necessárias para construir tal sistema, como produtor de evento, recepção de fluxo de entrada e EPAs utilizados pela máquina de inferência CEP.

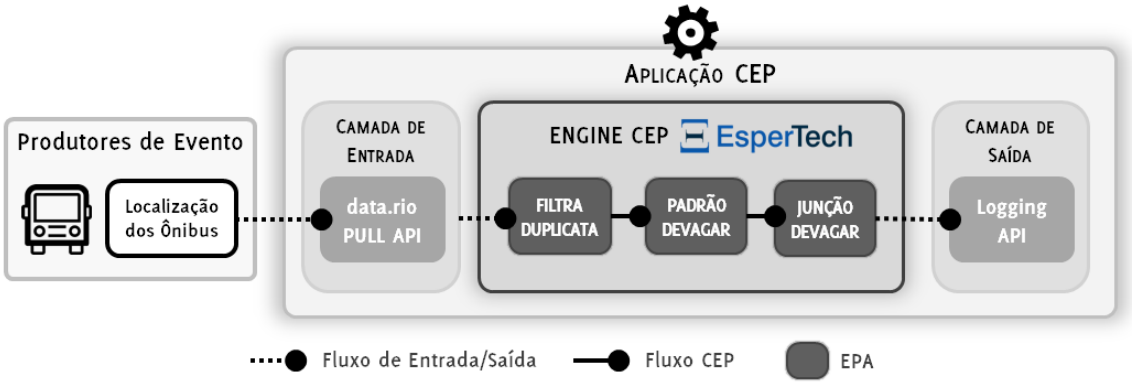


Figura 2.8: Arquitetura da aplicação CEP de cidades inteligentes para detecção de possíveis congestionamentos no trânsito.

O primeiro passo para a construção da aplicação é a definição da fonte do fluxo de eventos. Neste caso, será utilizado a plataforma de dados abertos `data.rio`[1] para obtenção da localização de todos os ônibus da cidade. A Tabela 2.1 ilustra um subconjunto

[1]O `data.rio` é uma plataforma de dados abertos que divulga em tempo real diversas informações da cidade do Rio de Janeiro. A posição de todos os ônibus pode ser obtida na seguinte URL: http://dadosabertos.rio.rj.gov.br/apiTransporte/apresentacao/rest/index.cfm/obterTodasPosicoes

deste fluxo de dados. Cada dado contém informações referentes ao horário de envio (`DATAHORA`), identificador (`ORDEM`), posição (`LATITUDE` e `LONGITUDE`), além da velocidade do veículo, em metros por segundo, durante a medição (`VELOCIDADE`). Note que como o serviço retorna a última posição de todos os ônibus, pode ocorrer de ter dados com datas anteriores à atual, visto que os mesmos podem não ter atualizados suas posições ou terem parados de operar. Utilizando os campos descritos na fonte de dados podemos definir o primeiro evento da aplicação: `LocalizacaoOnibusAPI`. Adota-se o termo API para ilustrar que o evento não foi pré-processado e reflete diretamente o dado da fonte do fluxo.

Tabela 2.1: Exemplo do formato de dados de ônibus da API `data.rio`

DATAHORA	ORDEM (ID)	LINHA	LATITUDE	LONGITUDE	VELOCIDADE
09-14-2019 18:04:44	A72062	14	-22.921499	-43.185966	0.56
09-14-2019 18:04:45	A72160	105	-22.982031	-43.240974	2.5
09-14-2019 18:04:47	B10124	323	-22.80504	-43.206684	0

Os dados podem ser obtidos em tempo real consultando a *Application Programing Interface* (API) do serviço `data.rio`, isto é, aplicações interessadas podem puxar (*pull*) os dados da API para obter um reflexo do estado da posição dos veículos. Isto pode ser realizado periodicamente para reconstruir o fluxo de eventos. Precisamente, no estudo de caso buscamos os dados a cada 30 segundos. Entretanto, como a API retorna as últimas posições de todos os ônibus, pode ocorrer de que leituras subsequentes da API contenham dados duplicados.

Para reter eventos duplicados, o primeira EPA utiliza a primitiva de *filtrar* e *sequência*, como descrito no Código 2.8. O EPA funciona da seguinte maneira. Ao receber um evento do tipo `LocalizacaoOnibusAPI` a consulta contínua executa as seguintes etapas. Primeiro, verifica-se se o evento é diferente dos eventos prévios que tenha recebido. Para cada evento `b1` distinto (`EVERY-DISTINCT`) inicia-se a busca por um evento subsequente do mesmo veículo, mas com data diferente. Caso o evento seja repetido, descarta-se o mesmo. Além disso, verifica-se se o evento distinto recebido casa com alguma sequência prévia, isto é, de um evento passado do mesmo veículo. Nesta situação, gera-se o evento do tipo `LocalizacaoOnibus`, que tem estrutura equivalente ao `LocalizacaoOnibusAPI`, mas que representa um evento não duplicado.

```
1 INSERT INTO LocalizacaoOnibus
2 SELECT b2.data AS data, b2.id AS id,
3        b2.linha AS linha, b2.velocidade AS velocidade,
4        b2.latitude AS latitude, b2.longitude AS longitude
5 FROM PATTERN
6 [ EVERY-DISTINCT(b1.data, b1.id) b1 = LocalizacaoOnibusAPI
7    -> b2 = LocalizacaoOnibusAPI(id = b1.id, data != b1.data) ]
```

Código 2.8: EPA para filtrar dados de localização duplicados.

O próximo EPA consome o fluxo de eventos `LocalizacaoOnibus` para detectar quando os veículos estão se movendo lentamente (evento `OnibusDevagar`). Para isso, utiliza-se novamente a primitiva de `sequência` em conjunto com as de `filtrar` e `projetar`, como descrito no Código 2.9. A consulta contínua inicia a detecção do padrão ao receber um evento do tipo `LocalizacaoOnibus`, representado pela variável `p1`. Após isso, espera-se por um intervalo de 3 minutos. Em seguida, ao receber a próxima localização do veículo (`p2`) calcula-se a movimentação do veículo através da função `distancia`, uma função externa que computa a distância em metros entre duas posições de latitude e longitude. Caso a distância percorrida seja inferior a 750 m, isto é, menor que $\frac{750}{180} \times 3.6 = 15$ km/h, gera-se um evento `OnibusDevagar` com as localizações `p1` e `p2`, além da distância e do tempo entre os eventos. Caso negativo, o padrão é cancelado e considera-se que o veículo moveu normalmente. Note que é importante considerar um fluxo que não tenha eventos duplicados, como o utilizado, uma vez que ao considerar eventos idênticos de `LocalizacaoOnibus` chegar-se-ia a conclusão que o veículo não moveu durante o período, entretanto, isso apenas significa que o mesmo não atualizou sua posição neste intervalo.

```
INSERT INTO OnibusDevagar
SELECT p1 AS primPosicao, p2 AS ultPosicao, p1.id AS id,
       distancia(p1, p2) AS distViajada, (p2.data - p1.data) AS periodo
FROM PATTERN
[
   EVERY p1=BusLocationUpdateEvent
   ->
   timer:interval(3 min)
   ->
   p2=BusLocationUpdateEvent(id = p1.id)
]
WHERE distancia(p1, p2) <= 750
```

Código 2.9: EPA para detectar ônibus que estão se movendo lentamente.

A próxima etapa é correlacionar, isto é, agrupar, eventos de `OnibusDevagar` que ocorrem próximos um dos outros. Para tal, primeiramente se cria a janela `OnibusDevagarWindow`, como descrito no Código 2.10, que irá contemplar apenas os últimos eventos do fluxo dos últimos 5 minutos através dos operadores `#UNIQUE`(id)) e `#TIME`(5 min) respectivamente. Observe que ao passo que o veículo se move, os novos eventos de `OnibusDevagar` sobrescrevem os eventos prévios na janela de eventos nomeada.

```
CREATE WINDOW OnibusDevagarWindow#UNIQUE(id)#TIME(5 min) AS OnibusDevagar
```

Código 2.10: EPL para criar uma janela do fluxo `OnibusDevagar`.

Para encontrar a concentração de ônibus que estão se movendo lentamente e, que possivelmente representa um congestionamento, o último EPA irá correlacionar o surgimento de um novo evento do tipo `OnibusDevagar` com os restantes da janela nomeada `OnibusDevagarWindow`, como descrito no Código 2.11. Ao receber um evento `p1` do tipo `OnibusDevagar` o EPA consultará a janela `OnibusDevagarWindow`, aqui denominada `pw`. Em seguida, correlaciona `p1` com todos eventos de `pw`. . Caso exista pelo menos três eventos no mesmo período (de 5 minutos) e de ônibus diferentes localizados em até 500 metros de p1 Os eventos são ditos próximos se a distância entre os ônibus lentos for menor do que 500 metros. Caso existam pelo menos três veículos nesta situação (com pouca movimentação), em uma janela de 5 minutos, gera-se um evento complexo do tipo `PossivelCongestionamento` composto pelos eventos individuais de cada ônibus lento.

```
1 ON OnibusDevagar AS p1
2 INSERT INTO PossivelCongestionamento
3 SELECT p1, pw
4 FROM OnibusDevagarWindow AS pw
5 WHERE distancia(p1, pw) <= 500
6 HAVING COUNT(*) >= 3
```

Código 2.11: EPA para detectar aglomerados de ônibus lentos.

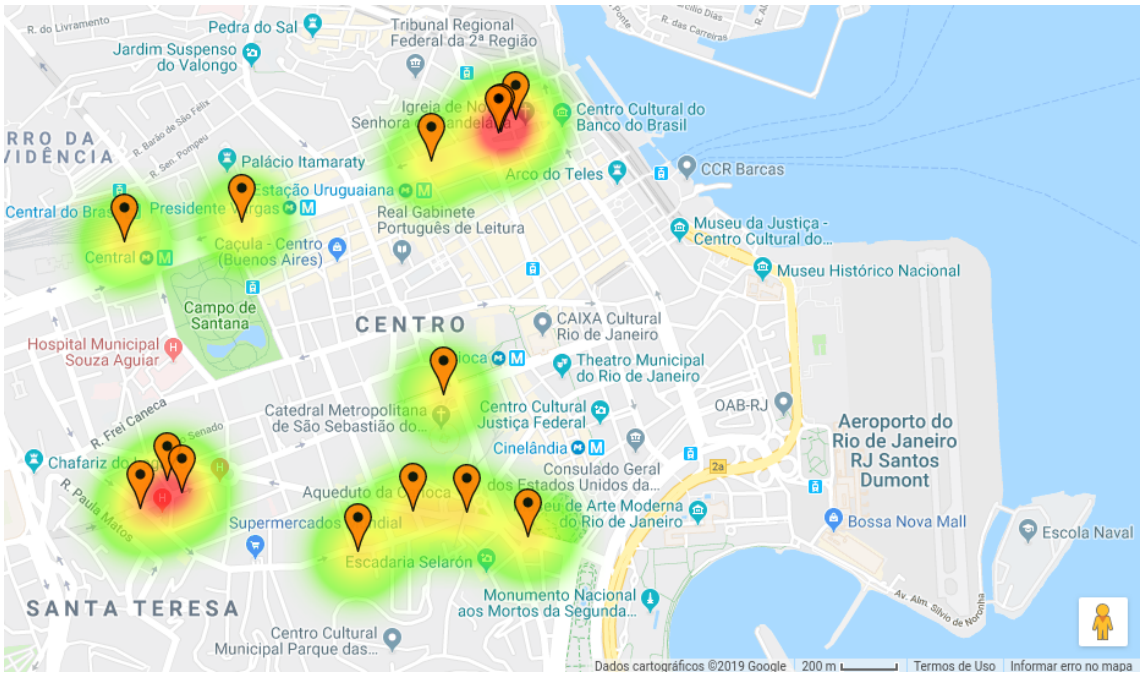


Figura 2.9: Possíveis locais de lentidão no trânsito conforme o processamento do fluxo de dados.

A Figura 2.9 ilustra o mapa de calor dos diversos eventos de `PossivelCongestionamento` detectados na região do centro do Rio de Janeiro. Cada evento desses é composto pelos

ônibus lentos próximos um dos outros, isto é, de eventos do tipo `OnibusDevargar`. As áreas e regiões de calor encontradas refletem regiões comumente associadas a lentidão e congestionamento, como Av. Presidente Vargas e Praça da Cruz Vermelha.

Além disso, ressalta-se que é possível estender essa aplicação para não somente detectar as áreas de lentidão e, possivelmente, de congestionamento no trânsito, mas também de acompanhar a evolução das mesmas. Por exemplo, pode-se monitorar os eventos `PossivelCongestionamento` para verificar se os mesmos estão aumentando ou diminuindo através da intersecção com eventos passados do mesmo tipo.

2.6. Considerações Finais

O *Complex Event Processing* (CEP – Processamento de Eventos Complexos) é um modelo de programação importante para sistemas orientados a fluxos de dados, como dados de sensores, análise de tráfego e análise de redes sociais. Este capítulo teve o objetivo de apresentar o modelo de programação CEP como meio de lidar com as especificidades de fluxos de dados.

Especificamente, o capítulo apresentar ao leitor: (1) os conceitos e problemas fundamentais de processamento de fluxo de dados e o modelo CEP; (2) o motor de inferência Esper e a sua linguagem de regras EPL como meio de tratar os fluxo de dados; (3) exemplo um caso de uso da aplicação de CEP para tratamento de dados de aplicações IoT; (4) exemplo de um caso de uso de aplicação de CEP como meio de processar dados de aplicações para *Smart Cities*.

Nesse contexto, acreditamos que a proposta do minicurso é importante para estudantes, pesquisadores e profissionais interessados em explorar as potencialidades de fluxos de dados de aplicações de Internet das Coisas e Cidades Inteligentes.

Referências

[Ali et al. 2011] Ali, M., Chandramouli, B., Goldstein, J., and Schindlauer, R. (2011). The extensibility framework in microsoft streaminsight. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1242–1253, Washington, DC, USA. IEEE Computer Society.

[Amini et al. 2014] Amini, A., Wah, T., and Saboohi, H. (2014). On Density-Based Data Streams Clustering Algorithms: A Survey. *Journal of Computer Science and Technology*, 29(1):116–141.

[Anicic et al. 2010] Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., and Studer, R. (2010). A rule-based language for complex event processing and reasoning. In Hitzler, P. and Lukasiewicz, T., editors, *Web Reasoning and Rule Systems*, pages 42–57, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Arasu et al. 2003] Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. (2003). Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA. ACM.

[Arasu et al. 2005] Arasu, A., Babu, S., and Widom, J. (2005). The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142.

[Bali 2009] Bali, M. (2009). *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing.

[Cameron et al. 2012] Cameron, M. A., Power, R., Robinson, B., and Yin, J. (2012). Emergency situation awareness from twitter for crisis management. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 695–698, New York, NY, USA. ACM.

[Carbone et al. 2015] Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., and Tzoumas, K. (2015). Apache Flink: Unified Stream and Batch Processing in a Single Engine. *Data Engineering*, pages 28–38.

[Cugola and Margara 2012] Cugola, G. and Margara, A. (2012). Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys*, 44(3):1–62.

[Dunkel et al. 2011] Dunkel, J., Fernández, A., Ortiz, R., and Ossowski, S. (2011). Event-driven architecture for decision support in traffic management systems. *Expert Systems with Applications*, 38(6):6530 – 6539.

[EsperTech 2019] EsperTech (2019). Esper - Complex Event Processing. `http://www.espertech.com/esper/`.

[Etzion and Niblett 2010] Etzion, O. and Niblett, P. (2010). *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

[Flouris et al. 2016] Flouris, I., Giatrakos, N., Deligiannakis, A., Garofalakis, M., Kamp, M., and Mock, M. (2016). Issues in complex event processing: Status and prospects in the Big Data era. *Journal of Systems and Software*, pages 1–20.

[Kudyba 2014] Kudyba, S. (2014). *Big Data, Mining, and Analytics*. Auerbach Publications, Boca Raton, Florida, 1st edition.

[Luckham 2001] Luckham, D. C. (2001). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Maison et al. 2013] Maison, R., Majda, E., Dobrowolski, A. P., and Zakrzewicz, M. (2013). Similarity based join over audio feeds in a multimedia data stream management system. *Bell Labs Technical Journal*, 18(1):195–212.

[Matysiak 2012] Matysiak, M. (2012). Data Stream Mining: Basic Methods and Techniques. Technical report, Rheinisch-Westfälische Technische Hochschule Aachen.

[McAfee and Brynjolfsson 2012] McAfee, A. and Brynjolfsson, E. (2012). Big data: the management revolution. *Harvard business review*, 90(10):61–68.

[Microsoft 2015] Microsoft (2015). Microsoft StreamInsight.

[Roriz Junior 2017] Roriz Junior, M. (2017). *DG2CEP: An On-line Algorithm for Real-Time Detection of Spatial Clusters from Large Data Streams Through Complex Event Processing*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brazil.

[Roriz Junior et al. 2019] Roriz Junior, M., de Oliveira, R. P., Carvalho, F., Lifschitz, S., and Endler, M. (2019). Mensageria: A smart city framework for real-time analysis of traffic data streams. In Oliveira, J., Farias, C. M., Pacitti, E., and Fortino, G., editors, *Big Social Data and Urban Computing*, pages 59–73, Cham. Springer International Publishing.

[Suhothayan et al. 2011] Suhothayan, S., Gajasinghe, K., Loku Narangoda, I., Chaturanga, S., Perera, S., and Nanayakkara, V. (2011). Siddhi: A Second Look at Complex Event Processing Architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments - GCE '11*, page 43, New York, New York, USA. ACM Press.

[van der Zee and Scholten 2013] van der Zee, E. and Scholten, H. (2013). Application of geographical concepts and spatial technology to the internet of things. WorkingPaper 2013-33, Faculty of Economics and Business Administration.

[Wu et al. 2006] Wu, E., Diao, Y., and Rizvi, S. (2006). High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA. ACM.

[Yadranjiaghdam et al. 2017] Yadranjiaghdam, B., Yasrobi, S., and Tabrizi, N. (2017). Developing a real-time data analytics framework for twitter streaming data. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 329–336.

[Yanlei Diao Neil Immerman and Gyllstrom 2007] Yanlei Diao Neil Immerman and Gyllstrom, D. (2007). SASE+: An Agile Language for Kleene Closure over Event Streams. Technical Report UM-CS-07-03, Department of Computer Science, University of Massachusetts Amherst.

[Zhao et al. 2017] Zhao, X., Garg, S., Queiroz, C., and Buyya, R. (2017). Chapter 11 - a taxonomy and survey of stream processing systems. In Mistrik, I., Bahsoon, R., Ali, N., Heisel, M., and Maxim, B., editors, *Software Architecture for Big Data and the Cloud*, pages 183 – 206. Morgan Kaufmann, Boston.

[Zheng et al. 2014] Zheng, Y., Capra, L., Wolfson, O., and Yang, H. (2014). Urban Computing. *ACM Transactions on Intelligent Systems and Technology*, 5(3):1–55.

BIO

Marcos Roriz. É professor adjunto da Universidade Federal de Goiás (UFG). Possui graduação em Ciência da Computação e mestrado pela UFG, e doutorado em Informática pela PUC-Rio. Suas áreas de interesse concentram-se em Sistemas Inteligentes de Transportes e Sistemas Distribuídos, com foco em algoritmos e plataformas de middleware para cidades inteligentes, inteligência artificial, computação móvel e computação ubíqua. Curriculo Lattes: `http://lattes.cnpq.br/1356289387726731`

Álan Guedes. É pesquisador de pós-doutorado no laboratório TeleMídia da PUC-Rio. Obteve graduação (2009) e mestrado(2012) pela UFPB, e doutorado em Informática (2017) pela PUC-Rio. Atuou em projetos de pesquisa em vídeo interativo, como os premiados GingaStore e Brasil4. Seus interesses de pesquisa incluem multimídia, vídeo interativo, mídia imersiva e Deep Learning para Multimídia. Curriculo Lattes: `http://lattes.cnpq.br/1481576313942910`.

Fernando B. V. Magalhães. É aluno de mestrado em Informática pela PUC-Rio e pesquisador no laboratório LAC. Possui graduação em computação (2018) pela Universidade Federal do Maranhão. Atualmente desenvolve projetos de CEP distribuído e aplicado a processos bioquímicos. Curriculo Lattes: `http://lattes.cnpq.br/9225104372311945`.

Sérgio Colcher. É professor do quadro principal da PUC-Rio desde 2001 e coordenador do laboratório TeleMídia. Obteve os títulos de Engenheiro de Computação (1991), Mestre (1993) e Doutor em Informática (1999), todos pela PUC-Rio, além do Pós-Doutorado (2003) no ISIMA (Institute Supérieur D'Informatique et de Modelisation des Applications, França). Suas áreas de interesse incluem redes de computadores, análise de desempenho de sistemas computacionais, sistemas multimídia/hipermídia e sistemas de TV digital.. Curriculo Lattes: `http://lattes.cnpq.br/1104157433492666`.

Markus Endler. Bacharel em Matemática e Mestre em Informática pela PUC-Rio (1984 e 1987), obteve o título de Dr.rer.nat. em Informatica da Technische Universitat Berlim (1992), e o título de Professor livre-docente pela Universidade de São Paulo (2001). Atualmente é professor associado da PUC-Rio. Tem experiência na área de Sistemas Distribuídos, com ênfase em: computação móvel e ubíqua, protocolos distribuídos para redes móveis, middleware, ciência de contexto e colaboração móvel. Currículo Lattes:`http://lattes.cnpq.br/6505039023842313`