UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
PROGRAMA DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Jean Carlos Gonçalves

PROCESSADOR DE EVENTOS COMPLEXOS DE ALTO DESEMPENHO PARA FLUXOS MASSIVOS DE EVENTOS

Florianópolis, Santa Catarina – Brasil
2023

Jean Carlos Gonçalves

PROCESSADOR DE EVENTOS COMPLEXOS DE ALTO DESEMPENHO PARA FLUXOS MASSIVOS DE EVENTOS

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Sistemas de Informação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Sistemas de Informação.
Orientador: Ronaldo dos Santos Mello, Dr.

Florianópolis, Santa Catarina – Brasil
2023

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina.
Arquivo compilado às 01:01h do dia 11 de julho de 2023.

Jean Carlos Gonçalves
Processador de Eventos Complexos de Alto Desempenho para Fluxos Massivos de Eventos / Jean Carlos Gonçalves; Orientador, Ronaldo dos Santos Mello, Dr. - Florianópolis, Santa Catarina - Brasil, 29 de junho de 2023.
108 p.

Trabalho de Conclusão de Curso - Universidade Federal de Santa Catarina, INE - Departamento de Informática e Estatística, CTC - Centro Tecnológico, Programa de Graduação em Sistemas de Informação.

Inclui referências

1. Complex Event Processing, 2. Complex Event Recognition, 3. Streaming, 4. Otimização, 5. Arquitetura Orientada a Eventos, 6. Event Sourcing, 7. Big Data, 8. In-Memory Index, I. Ronaldo dos Santos Mello, Dr. II. Programa de Graduação em Sistemas de Informação III. Processador de Eventos Complexos de Alto Desempenho para Fluxos Massivos de Eventos

CDU 02:141:005.7

Jean Carlos Gonçalves

PROCESSADOR DE EVENTOS COMPLEXOS DE ALTO DESEMPENHO PARA FLUXOS MASSIVOS DE EVENTOS

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Sistemas de Informação, e foi aprovado em sua forma final pelo Programa de Graduação em Sistemas de Informação do INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, 29 de junho de 2023.

Renato Cislaghi, Dr.
Coordenador do Programa de Graduação em Sistemas de Informação

Banca Examinadora:

Ronaldo dos Santos Mello, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Renato Fileto, Dr.
Avaliador
Universidade Federal de Santa Catarina

Luiz Henrique Zambom Santana, Dr.
Avaliador
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Gostaria de agradecer primeiramente à minha família. Aos meus pais e irmãos pelo apoio moral, e ao meu cunhado Arthur pela iluminação quando eu mais precisei. Agradeço também ao professor Rafael de Santiago por me ensinar a escrever algoritmos em LaTeX.

Um agradecimento especial para os meus amigos Marcos Laurentino e Juliano Souza dos Passos. Por estarem ao meu lado durante todo esse tempo. Por não me deixarem desistir. Por me inspirarem a buscar uma versão melhor de mim mesmo. Pelas incontáveis vezes em que vocês me ajudaram, mesmo que a distância. Amo vocês, meus irmãos.

Agradeço a UFSC por tudo o que ela me proporcionou durante todos esses anos. Ao corpo docente do CTC pelo aprendizado, e ao meu professor orientador Ronaldo, pela paciência e determinação em me fazer passar pela linha de chegada.

Por último, gostaria não só de agradecer como também dedicar minha graduação à minha esposa, Manoela. Só eu sei o quanto você sacrificou para realizar o meu sonho. Sem você, meu amor, eu jamais teria chegado aonde cheguei. Você me trouxe determinação quando pensei em desistir, esperança quando deixei de acreditar e me ajudou a enxergar quando eu não conseguia ver. Muito obrigado por tudo. Eu te amo.

RESUMO

O Complex Event Processing (CEP) é um tipo de arquitetura orientada a eventos que visa detectar situações de interesse na forma de eventos complexos. Para tais sistemas, a latência é uma métrica significativamente importante. Entretanto, a análise/processamento de consultas CEP é conhecida por ser computacionalmente desafiadora. Diversas técnicas de otimização foram propostas para minimizar a latência desses sistemas. Porém, uma análise dos trabalhos recentes no campo de CEP evidencia que, apesar de efetivas, as técnicas de otimização propostas ainda possuem limitações e oportunidades de melhoria. Neste contexto é proposto o desenvolvimento de um processador de CEP, com o diferencial de possuir tempos de resposta e vazão constantes ou quase constantes, independente da quantidade de eventos processada anteriormente.

Palavras-chaves: Complex Event Processing. Complex Event Recognition. Streaming. Otimização. Arquitetura Orientada a Eventos. Event Sourcing. Big Data. In-Memory Index.

ABSTRACT

Complex Event Processing (CEP) is a type of event-driven architecture that aims to detect situations of interest in the form of complex events. For such systems, latency is a significantly important metric. However, parsing/processing CEP queries is known to be computationally challenging. Several optimization techniques have been proposed to minimize the latency of these systems. However, an analysis of recent works in the CEP field shows that, despite being effective, the proposed optimization techniques still have limitations and opportunities for improvement. In this context, the development of a CEP processor is proposed, with the differential of having constant or almost constant response times and throughput, regardless of the number of events previously processed.

Keywords: Complex Event Processing. Complex Event Recognition. Streaming. Optimization. Event-Driven Architecture. Event Sourcing. Big Data. In-Memory Index.

LISTA DE FIGURAS

Figura 1 – Arquitetura orientada a eventos . . . 17
Figura 2 – Arquitetura genérica de CEP . . . 18
Figura 3 – Arquitetura genérica de CEP 2 . . . 19
Figura 4 – NFA de um evento complexo . . . 21
Figura 5 – Árvore de um evento complexo . . . 22
Figura 6 – *Pattern reordering* . . . 23
Figura 7 – *Pattern sharing* . . . 24
Figura 8 – Arquitetura do pSPICE . . . 26
Figura 9 – Arquitetura do processador . . . 29
Figura 10 – Expressão de um evento complexo . . . 29
Figura 11 – Definição de um evento complexo . . . 30
Figura 12 – Evento (i): visita em uma página . . . 31
Figura 13 – Evento (ii): abertura de e-mail . . . 31
Figura 14 – Evento (iii) click . . . 31
Figura 15 – Índice de predicados . . . 33
Figura 16 – Evento irrelevante . . . 34
Figura 17 – Predicado sem o campo “dispositivo” . . . 35
Figura 18 – Índice de predicados . . . 35
Figura 19 – Adicionando um item a um *sortedset* . . . 38
Figura 20 – Lendo um item de um *sortedset* . . . 38
Figura 21 – Ferramenta CLI do Cloud Bigtable . . . 39
Figura 22 – Carregamento do estado dos predicados . . . 42
Figura 23 – Estrutura da requisição de registro . . . 44
Figura 24 – Estrutura da requisição de processamento . . . 44
Figura 25 – Configurações do Docker . . . 47
Figura 26 – Métricas agregadas do experimento 1 . . . 48
Figura 27 – Throughput do experimento 1 . . . 49
Figura 28 – Tempos de resposta do experimento 1 . . . 49
Figura 29 – Consumo de memória do experimento 1 . . . 50
Figura 30 – *Footprint* base de memória . . . 50
Figura 31 – Quantidade de estados intermediários do experimento 1 . . . 50
Figura 32 – Armazenamento usado pelo Redis no experimento 1 . . . 51
Figura 33 – Throughput do experimento 2 . . . 51
Figura 34 – Tempos de resposta do experimento 2 . . . 52
Figura 35 – Consumo de memória do experimento 2 . . . 52
Figura 36 – Throughput do experimento 3 . . . 57
Figura 37 – Tempos de resposta do experimento 3 . . . 58
Figura 38 – Consumo de memória do experimento 3 . . . 58

Figura 39 – Quantidade de estados intermediários do experimento 3 59
Figura 40 – Armazenamento usado pelo Redis no experimento 3 59

LISTA DE TABELAS

Tabela 1 – Comparação entre os trabalhos relacionados e a ferramenta proposta . . . 27
Tabela 2 – Estado inicial 40
Tabela 3 – Estado após evento (ii) 40
Tabela 4 – Estado após evento (iii) 40

LISTA DE CÓDIGOS

Código 1 – Formato da resposta para requisição de registro. 44
Código 2 – Formato da resposta para requisição de processamento. 45
Código 3 – Consulta 1 47
Código 4 – Consulta 2 52

SUMÁRIO

1 INTRODUÇÃO . . . 13
1.1 MOTIVAÇÃO . . . 13
1.2 JUSTIFICATIVA . . . 14
1.3 OBJETIVOS . . . 14
1.3.1 Objetivo geral . . . 14
1.3.2 Objetivos específicos . . . 15
1.4 ORGANIZAÇÃO DESTE TRABALHO . . . 15

2 FUNDAMENTAÇÃO TEÓRICA . . . 16
2.1 ARQUITETURA ORIENTADA A EVENTOS . . . 16
2.2 PROCESSAMENTO DE EVENTOS COMPLEXOS . . . 17
2.2.1 Operadores . . . 20
2.2.2 Abordagens . . . 21
2.2.2.1 Desafios e Otimizações . . . 22

3 TRABALHOS RELACIONADOS . . . 25
3.1 REAL-TIME MULTI-PATTERN DETECTION OVER EVENT STREAMS . 25
3.2 PSPICE: PARTIAL MATCH SHEDDING FOR COMPLEX EVENT PROCESSING . . . 25
3.3 ZSTREAM: A COST-BASED QUERY PROCESSOR FOR ADAPTIVELY DETECTING COMPOSITE EVENTS . . . 26
3.4 COMPARATIVO . . . 27

4 PROPOSTA . . . 28
4.1 VISÃO GERAL . . . 28
4.2 CATÁLOGO . . . 32
4.3 PROCESSO DE FILTRO . . . 33
4.4 PROCESSO DE PERSISTÊNCIA . . . 36
4.4.1 Modelo de dados . . . 37
4.4.2 Alternativas . . . 38
4.5 PROCESSO DE TESTE DA EXPRESSÃO . . . 40
4.6 API . . . 44
4.6.1 Registro . . . 44
4.6.2 Processamento . . . 44

5 EXPERIMENTOS . . . 46
5.1 EXPERIMENTO 1: OPERADOR DE SEQUÊNCIA COM *DATASET* ORDENADO . . . 47

5.2 EXPERIMENTO 2: DATASET DESORDENADO 51
5.3 EXPERIMENTO 3: DEC COM MAIS PREDICADOS 52

6 CONCLUSÃO 60

REFERÊNCIAS 62

APÊNDICE A – CÓDIGO-FONTE 65

APÊNDICE B – ARTIGO NO FORMATO SBC 89

1 INTRODUÇÃO

A arquitetura orientada a eventos tem se tornado cada vez mais proeminente no desenvolvimento de aplicações modernas, principalmente em aplicações construídas como microsserviços, devido ao seu potencial para facilitar a construção de aplicações na nuvem. Nesta abordagem, eventos são usados para acionar e comunicar serviços independentes e desacoplados. Isso permite que os serviços trabalhem de maneira mais flexível e escalável, o que pode ser uma vantagem em ambientes em constante mudança (AWS, 2022a).

O *Complex Event Processing (CEP)* é um tipo de arquitetura orientada a eventos. Ele surgiu para suportar aplicações de *streaming*, como por exemplo, análise em tempo real e detecção de intrusão de rede. Ele opera sobre *streams* de eventos primitivos de alta velocidade e analisa/processa consultas para detectar eventos complexos, ou seja, coleções de eventos primitivos que satisfazem algum padrão (BUCCHI *et al.*, 2021).

No contexto da detecção de fraudes, eventos relacionados ao uso de cartão de crédito são transmitidos continuamente para um mecanismo de processamento de eventos complexos. Esses eventos são chamados de eventos primitivos ou simplesmente eventos. Cada evento pertence a um tipo que define um conjunto de atributos a serem associados a ele. Por exemplo, o tipo *BigTransReq* pode representar uma solicitação de transação para uma grande soma, com atributos como *cardID* e *valor*.

Os padrões CEP especificam cenários de interesse a serem detectados. Suponha que queremos detectar uma possível tentativa de fraude reconhecendo uma sequência de duas tentativas falhas de verificação do *CVV (código de segurança do cartão)* seguidas por uma tentativa bem-sucedida. Em casos de fraude, isso geralmente é seguido por uma solicitação inicial de transação pequena e, em seguida, uma segunda solicitação de transação grande para um valor superior a R$ 10.000,00 e pelo menos 10 vezes maior que o valor da solicitação de transação pequena anterior. Tudo isso ocorre dentro de um período de 5 minutos (CHAPNIK; KOLCHINSKY; SCHUSTER, 2021).

Os sistemas CEP visam, portanto, identificar situações de interesse na forma de eventos complexos, fornecendo informações relevantes para que sejam tomadas medidas adequadas. Como tal, eles se esforçam para realizar um processamento de consultas com baixa latência, o que permite que as respostas sejam dadas de forma rápida e eficiente (BUCCHI *et al.*, 2021). Na detecção de fraudes, por exemplo, a falha em identificar uma situação de interesse pode levar a prejuízos significativos. É justamente neste contexto de processamento de consultas com baixa latência que este trabalho está inserido.

1.1 MOTIVAÇÃO

A análise/processamento de consultas CEP é conhecida por ser computacionalmente desafiadora, e esse desafio aumenta conforme a quantidade de eventos processados (BUCCHI *et al.*, 2021). Diversas técnicas de otimização foram propostas para minimizar a latência de

processamento em sistemas CEP. As propostas vão desde a proposição de diversos modelos de execução, otimização de consulta com base em custo (como é feito por sistemas de gerência de bancos de dados - SGBDs), foco em fragmentos de consulta específicos (por exemplo, políticas de seleção de eventos), uso de técnicas de rejeição de carga (*load shedding*) para obter baixa latência às custas de potencialmente perder alguns resultados, e o emprego de computação distribuída (BUCCHI *et al.*, 2021; CHAPNIK; KOLCHINSKY; SCHUSTER, 2021).

Uma análise de trabalhos recentes na área de CEP evidencia que, apesar de efetivas, as técnicas de otimização propostas ainda possuem limitações e oportunidades de melhoria. Por exemplo, o acúmulo de estados intermediários em sistemas CEP abordado por Kolchinsky e Schuster (2019) e Slo *et al.* (2019), ou a falta de suporte a alguns operadores abordada por Mei e Madden (2009). Além disso, a maioria das propostas de otimização de CEP assume que os estados intermediários estão disponíveis na memória, o que reduz drasticamente a escalabilidade horizontal do sistema (KOLCHINSKY; SCHUSTER, 2019; SLO *et al.*, 2019; CHAPNIK; KOLCHINSKY; SCHUSTER, 2021; MEI; MADDEN, 2009).

A partir desta motivação, surge o propósito deste Trabalho de Conclusão de Curso (TCC): o desenvolvimento de um processador de eventos complexos eficiente, flexível e que escale bem com o número de eventos processados.

1.2 JUSTIFICATIVA

Trabalhos recentes (KOLCHINSKY; SCHUSTER, 2019; SLO *et al.*, 2019; CHAPNIK; KOLCHINSKY; SCHUSTER, 2021) expõem os desafios da implementação de CEP usando autômatos finitos, e a alternativa baseada em árvore analisada (MEI; MADDEN, 2009) possuem escalabilidade limitada, fortemente influenciada pela carga de trabalho. Apesar de ligeiramente diferentes, todas as abordagens analisadas usam alguma estrutura de dados em memória para indexar as definições dos eventos complexos e os estados intermediários. Porém, as duas informações possuem cardinalidades diferentes e, consequentemente, características de escalabilidade completamente distintas. Ainda assim, pouco foi feito no sentido de separá-los.

Sendo assim, a ferramenta proposta neste TCC tem como objetivo aproveitar a escalabilidade dos SGBDs modernos para desacoplar os sistemas CEP da indexação de estados intermediários.

1.3 OBJETIVOS

1.3.1 Objetivo geral

Desenvolver um sistema CEP com tempos de resposta e vazão constantes ou quase constantes, independente da quantidade de eventos processada anteriormente. Esse sistema poderá servir de base para a construção de aplicações de maior porte que processam grandes quantidades de eventos.

1.3.2 Objetivos específicos

Os objetivos específicos deste trabalho são:

1. Desenvolver um método eficiente para o processamento de eventos complexos;
2. Propor um modelo de dados para armazenar o estado do processador;
3. Avaliar a ferramenta proposta através de um estudo de caso;

1.4 ORGANIZAÇÃO DESTE TRABALHO

O restante deste trabalho está organizado conforme descrito a seguir. O capítulo 2 apresenta a fundamentação teórica, incluindo os conceitos de arquitetura orientada a eventos e CEP, além dos principais desafios na implementação desse tipo de arquitetura e as técnicas de otimização mais comuns. O capítulo 3 apresenta os trabalhos relacionados, focando principalmente em trabalhos que realizam algum tipo de otimização em sistemas CEP. O capítulo 4 apresenta a ferramenta proposta, sua arquitetura, os seus componentes, a sua API e os algoritmos principais. O capítulo 5 apresenta uma análise de desempenho do processador. O capítulo 6 apresenta a conclusão obtida com o desenvolvimento do trabalho e as ideias para o desenvolvimento de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

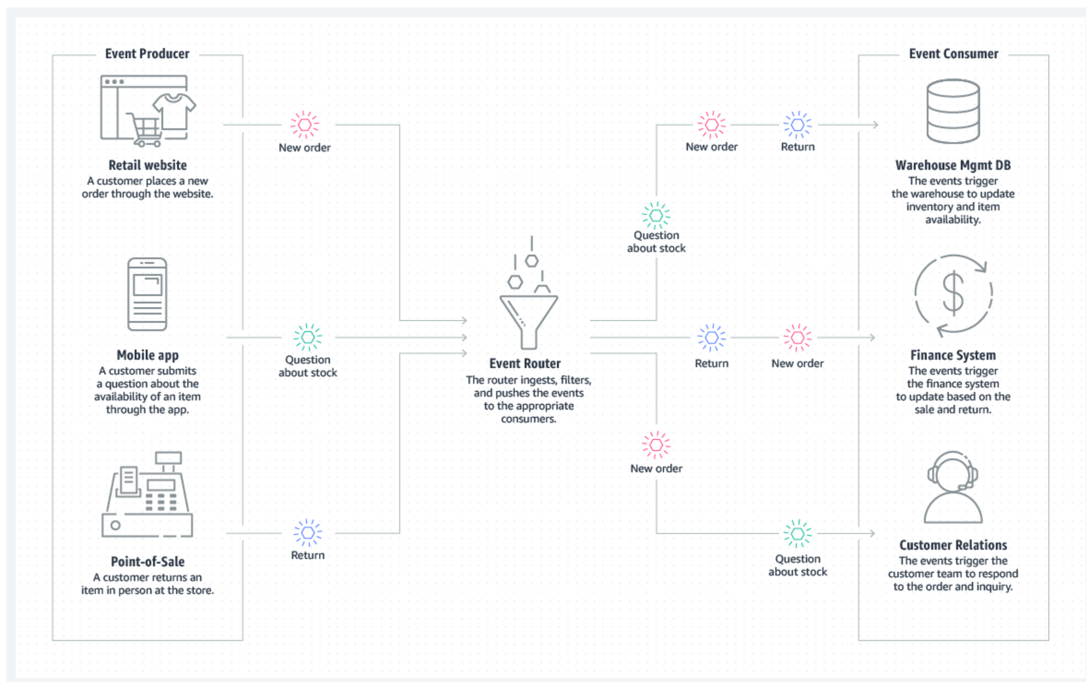
2.1 ARQUITETURA ORIENTADA A EVENTOS

Uma arquitetura orientada a eventos usa eventos para acionamento e comunicação entre serviços desacoplados. É uma abordagem bastante comum entre aplicações modernas estruturadas como microsserviços. Neste contexto, um evento é uma mudança ou uma atualização no estado, como um item adicionado a um carrinho de compras em um site de comércio eletrônico. Eles podem conter o estado (o item comprado, o preço ou um endereço de entrega) ou podem ser identificadores (uma notificação de que um pedido foi despachado, por exemplo) (AWS, 2022b). Os interessados avaliam o evento e, opcionalmente, tomam providências. A ação orientada a eventos pode incluir a invocação de um serviço, o acionamento de um processo de negócios e/ou publicação de informações adicionais (MICHELSON, 2006).

Por sua natureza, uma arquitetura orientada a eventos é fracamente acoplada e altamente distribuída. O criador (fonte) do evento só conhece o evento ocorrido. O criador não tem conhecimento do processamento posterior do evento, nem dos interessados. A rastreabilidade de um evento por meio de uma rede dinâmica de eventos multicaminho pode ser difícil. Assim, as arquiteturas orientadas a eventos são melhor usadas para fluxos assíncronos de trabalho e informações. Existem três estilos gerais de processamento de eventos: *simples, stream* e *complexo*. No processamento simples, um evento ocorre e desencadeia uma ou mais ações subsequentes. No processamento de *stream*, um fluxo contínuo de eventos é publicado em tempo-real. Paralelamente a isso, os interessados aguardam, e ao identificar a ocorrência de um novo evento, desencadeiam uma ou mais ações subsequentes. No processamento de eventos complexos, os eventos são filtrados e correlacionados para identificar padrões, chamados de eventos complexos. Os três estilos são frequentemente usados juntos em uma arquitetura madura orientada a eventos (MICHELSON, 2006).

A Figura 1 mostra um exemplo de arquitetura orientada a eventos para comércio eletrônico. Esta arquitetura é comum neste segmento pois nela é possível propagar um pedido para diversas aplicações, como por exemplo, um sistema financeiro ou um sistema de pós-venda, sem deixar o cliente esperando ou acoplá-las. A Figura mostra que os produtores e os consumidores dos eventos não se comunicam diretamente. Os eventos são enviados para um roteador e então encaminhados para os consumidores apropriados.

Figura 1 – Arquitetura orientada a eventos



Fonte: AWS (2022b)

Segundo a AWS (2022b), os principais casos de uso para a utilização dessa arquitetura são: (i) replicação de dados; (ii) distribuição e processamento paralelo; (iii) monitoramento e geração de alertas de estados de recursos; e (iv) integração de sistemas. No caso de uso (i), os eventos podem ser usados para sincronizar um ou mais sistemas. Por exemplo, um sistema distribuído globalmente pode sincronizar todas as suas regiões através de eventos. Uma dessas regiões poderia estar no leste dos Estados Unidos e a outra no Brasil.

No caso de uso (ii), o evento é emitido uma única vez e consumido paralelamente por diversas aplicações. Na Figura 1, por exemplo, o website emite uma única vez um evento para notificar a criação de um novo pedido, e esse evento é consumido paralelamente por três aplicações. Já no caso de uso (iii) pode-se emitir um evento sempre que houver alteração no estado de um recurso. Dessa forma, é possível identificar a ocorrência de anomalias ou situações de interesse, e reagir de acordo. Um sistema de comércio eletrônico, por exemplo, pode enviar um e-mail para o cliente sempre que o estado do seu pedido mudar.

Por fim, no caso de uso (iv) pode-se usar eventos para integrar sistemas sem acoplá-los. Uma aplicação de envio de e-mails, por exemplo, pode consumir um *stream* de eventos, onde cada evento é um e-mail. Qualquer aplicação poderia se integrar sem sequer saber quem está enviando os e-mails.

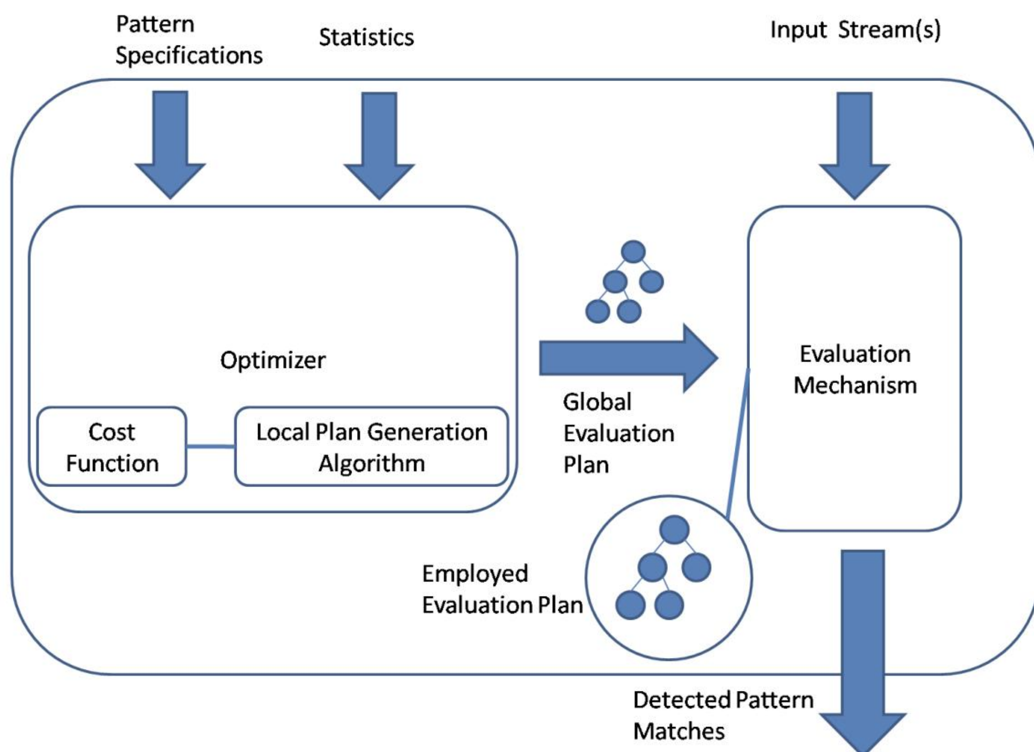
2.2 PROCESSAMENTO DE EVENTOS COMPLEXOS

Semelhante ao processamento de *stream* de eventos, o processamento de eventos complexos é uma tecnologia para agregar, processar e analisar fluxos massivos de dados para obter *insights*

em tempo real dos eventos à medida que eles ocorrem. O CEP separa os fatos irrelevantes dos demais ao transformar dados de baixo nível em informações de negócios de alto nível com as quais as empresas se preocupam (CONFLUENT, 2022).

A arquitetura de um sistema CEP varia de acordo com a sua implementação. A Figura 2 ilustra a arquitetura utilizada por Kolchinsky e Schuster (2019). Nesta arquitetura o processador recebe três informações: (i) as especificações dos eventos complexos (*Pattern Specifications*), (ii) um conjunto de estatísticas sobre os eventos primitivos (*Statistics*) e (iii) um *stream* de eventos (*Input Stream(s)*). Os eventos complexos são associados a uma função de custo, que é posteriormente utilizada pelo processador para decidir qual é o melhor plano de execução de acordo com as propriedades estatísticas dos *streams*. Nesta arquitetura o processamento dos eventos primitivos acontece no componente *Evaluation Mechanism*.

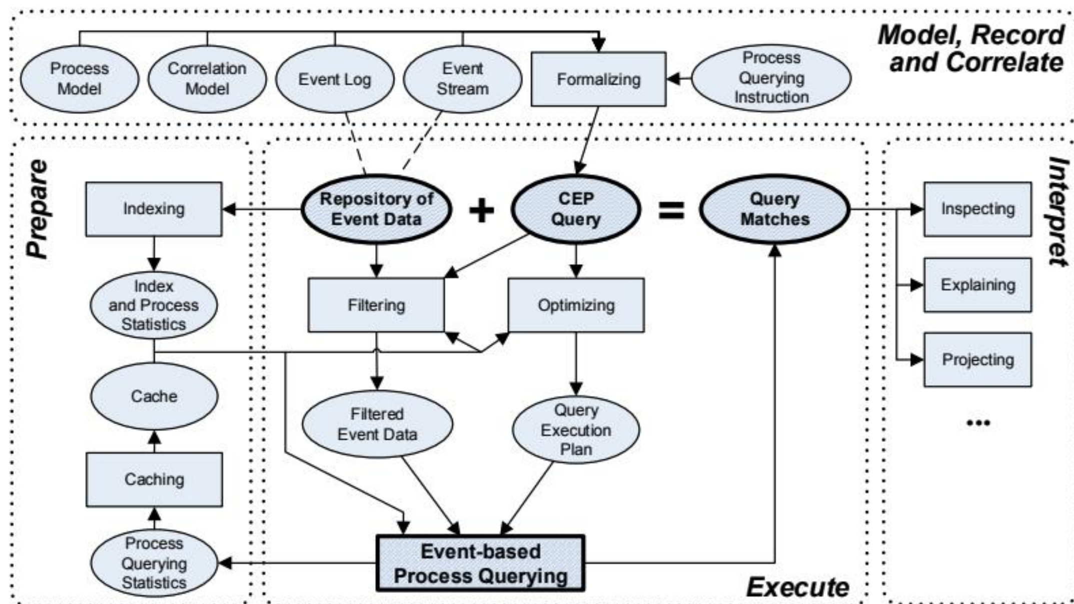
Figura 2 – Arquitetura genérica de CEP



Fonte: Kolchinsky e Schuster (2019)

A arquitetura utilizada por Aa, Artikis e Weidlich (2022) é ilustrada na Figura 3. Apesar de diferente, é possível observar a sua semelhança com a arquitetura anterior em alguns aspectos. Neste exemplo, o processador formaliza as especificações de eventos compostos a partir de um *Process Model*. As especificações são usadas para filtrar um *stream* de eventos (*Event Stream*), gerando eventos eventos complexos (*Query Matches*) e estatísticas.

Figura 3 – Arquitetura genérica de CEP 2



Fonte: Aa, Artikis e Weidlich (2022)

A distinção entre CEP e os demais tipos de processamento de eventos nem sempre é clara. Alguns autores, como Dayarathna e Perera (2018), abordam o CEP como um tipo especial de processamento de eventos. Outros autores, como Confluent (2022), abordam o CEP como uma generalização do processamento de *stream* tradicional. Porém, segundo Confluent (2022) existem algumas características que os distinguem. Por exemplo, as hierarquias conceituais e relações causais são incomuns em sistemas de processamento de *stream* de eventos, mas são geralmente encontradas em sistemas CEP.

As relações de hierarquia conceitual entre os eventos permitem que eventos de baixo nível, como cliques de um mouse, se tornem menos abstratos quando analisados em conjunto. Por exemplo, medições de sensores de umidade, temperatura, vento e etc, podem formar um evento meteorológico. Enquanto um processador de *stream* tradicional se preocupa em encontrar padrões de baixo nível nos dados, como uma trilha de GPS, o CEP busca padrões de alto nível, como intenções e sentimentos.

As relações causais são relacionamentos de causa e efeito entre dois eventos. Por exemplo, para cada evento “e-mail entregue” espera-se um evento “e-mail enviado”.

Segundo Confluent (2022), entre os principais casos de uso para a utilização dessa arquitetura estão: (i) prevenção e detecção de fraudes; (ii) marketing em tempo real; (iii) análise preditiva; e (iv) IoT. No caso de uso (i), as instituições financeiras podem usar o CEP para inspecionar e identificar transações fraudulentas, rastreando eventos em tempo real em relação a vários padrões. Um login de um novo dispositivo pode ser combinado com uma alteração de senha e outras atividades da conta para criar um evento complexo que sinaliza a possibilidade de fraude. Vários alertas de fraude podem ser combinados em um evento de nível superior que identifica uma violação em todo o sistema.

No caso de uso (ii), os varejistas de comércio eletrônico podem usar o CEP para oferecer recomendações personalizadas com base em uma combinação de dados de GPS, atividade de rede social, feriados e hábitos de compras anteriores. A capacidade de combinar diferentes fontes de dados com dados históricos é um dos principais pontos fortes do CEP.

No caso de uso (iii), ao combinar eventos gerados por vendas de farmácias, sites de redes sociais, twitter e fluxos de GPS, podemos prever o surgimento de novos clusters de coronavírus. Quase todas as formas de previsões dependem da descoberta de padrões complexos em grandes quantidades de dados de várias fontes. Portanto, o CEP é uma parte natural do cenário de análise preditiva.

No caso de uso (iv), ao combinar informações de várias fontes, o CEP tem um efeito transformador ao coletar fluxos de sensores de IoT para monitoramento, análise e solução de problemas em tempo real. Por exemplo, combinando dados distribuídos de iluminação, alarmes e outros dispositivos com clima, data e hora em tempo real, um edifício inteligente pode prever o comportamento de seus ocupantes e otimizar o uso de luzes e aquecimento enquanto fornece serviços automatizados aos ocupantes. Esse sistema também pode identificar convidados ou intrusos e tomar as medidas apropriadas.

2.2.1 Operadores

Diversas linguagens de consulta foram propostas para sistemas CEP nos últimos anos. Essas linguagens diferem em sintaxe e terminologia, dificultando uma comparação. Entretanto, é possível observar a existência de conjunto de operações comuns a muitas delas (AA; ARTIKIS; WEIDLICH, 2022). Essas operações são:

- *Conjunção ("and")*: identifica a ocorrência de um conjunto de eventos primitivos. Suponha a existência de dois eventos primitivos EV_i e EV_j. A conjunção EV_i *and* EV_j é verdadeira se e somente se EV_i e EV_j ocorrerem;

- *Disjunção ("or")*: identifica a ocorrência de pelo menos um dos eventos primitivos de um conjunto. A disjunção EV_i *or* EV_j é verdadeira se EV_i ou EV_j ocorrerem;

- *Sequência (";")*: identifica a ocorrência de um conjunto de eventos primitivos em uma ordem específica. A sequência $EV_i \; ; \; EV_j$ é verdadeira se EV_i ocorrer antes de EV_j;

- *Negação ("not")*: identifica a não ocorrência de um evento primitivo. A negação *not* EV_i é verdadeira se EV_i não ocorrer;

- *Fecho de Kleene ($A^*/A^+/A^{num}$)*: identifica a ocorrência de um evento primitivo um número específico de vezes. O fecho de Kleene EV_i^* é verdadeiro se EV_i ocorrer zero ou mais vezes. O fecho de Kleene EV_i^+ é verdadeiro se EV_i ocorrer uma ou mais vezes. O fecho de Kleene EV_i^{num} é verdadeiro se EV_i ocorrer *num* vezes;

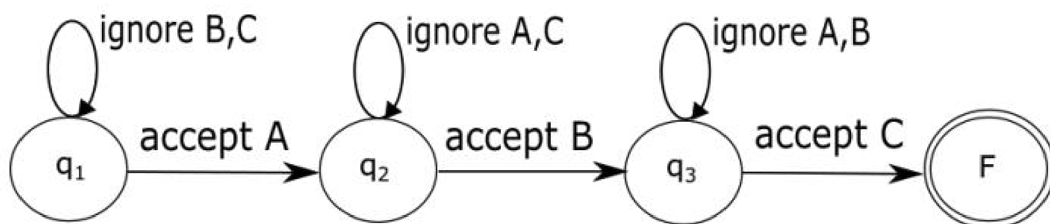
- *Janelas de tempo (within)*: identifica a ocorrência de um ou mais eventos primitivos dentro

de um intervalo de tempo. A janela de tempo EV_i *and* EV_j *within* t é verdadeira se EV_i e EV_j ocorrerem dentro de um intervalo de tempo t.

2.2.2 Abordagens

Segundo Aa, Artikis e Weidlich (2022), existem três abordagens principais de CEP: (i) baseados em autômatos finitos não-determinísticos (NFA); (ii) baseados em árvores e; (iii) baseados em lógica. Na abordagem (i), as definições de eventos complexos são representadas como um NFA. O processamento de eventos primitivos modifica o estado do autômato. Um estado pode ser final ou não-final. Os estados não-finais são chamados de estados intermediários (*partial matches*) e representam potenciais eventos complexos. Os estados finais representam ocorrências de eventos complexos. Segundo Aa, Artikis e Weidlich (2022), este é o mais proeminente dos três tipos.

Figura 4 – NFA de um evento complexo



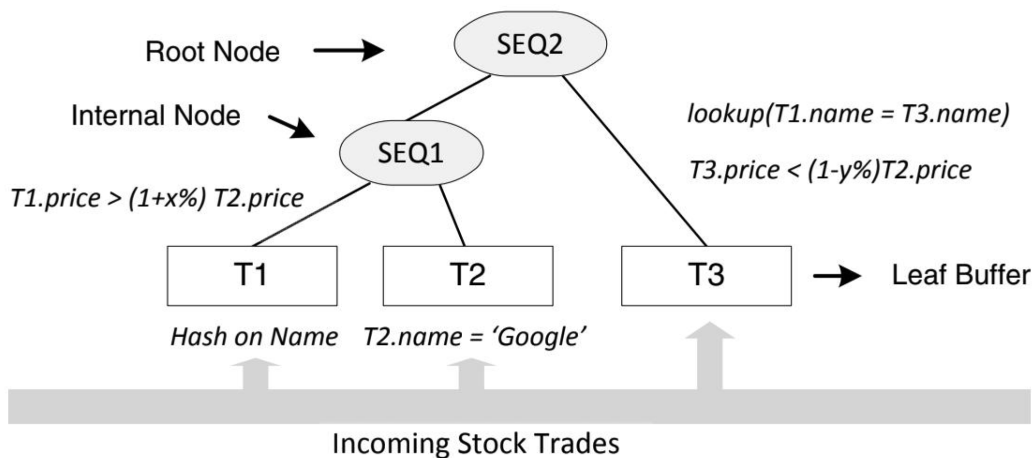
Fonte: Kolchinsky e Schuster (2019)

A Figura 4 ilustra o NFA de um evento complexo. Neste exemplo o evento complexo é definido pela sequência de eventos primitivos $A \rightarrow B \rightarrow C$. O estado inicial do autômato é q_1. Neste estado, o autômato espera um evento A. Ao receber A, o estado muda para q_2, onde espera B. Ao receber B, o estado muda para q_3 onde espera C. Ao receber C o estado muda para o estado F, que é final, e portanto emite um evento complexo.

Um NFA consegue representar de forma eficaz um evento complexo simples, mas sua estrutura impõe algumas limitações. Por exemplo, é difícil representar eventos complexos cujos eventos primitivos não acontecem em uma ordem pre-determinada. Além disso, é difícil representar operações como conjunção ou negação. A incorporação de lógica temporal também pode ser desafiadora. O processamento de eventos complexos muitas vezes requer a especificação de restrições temporais, como janelas de tempo e atrasos. A representação dessas restrições temporais em NFAs pode ser complexa e exigir transformações adicionais para garantir a corretude e eficiência do processamento.

Na abordagem (ii), as definições de eventos complexos são representados como uma árvore. Nesta estrutura, os nodos folhas são eventos primitivos, e os nodos internos são operadores (conjunção, disjunção, negação, fecho de Kleene, etc.). Já o nodo raiz é o evento complexo. Cada nodo interno é responsável por armazenar os estados intermediários das sub-árvores.

Figura 5 – Árvore de um evento complexo



Fonte: Mei e Madden (2009)

A Figura 5 ilustra uma árvore representando um evento complexo. Neste exemplo, o evento complexo é definido como uma ação cujo preço seja primeiro $x\%$ maior em relação a ação do Google e depois $y\%$ menor. Os três nodos raiz representam os eventos primitivos $T1$, $T2$ e $T3$. O nodo intermediário $SEQ1$ está associado ao predicado $T1.price > (1+x\%) * T2.price$. Esse operador é chamado de multiclasse, pois opera sobre duas classes diferentes de eventos ($T1$ e $T2$). Por fim, o nodo raiz $SEQ2$, também multiclasse, está associado ao predicado $T3.price < (1-y\%) * T2.price$. Um evento complexo ocorre quando a expressão $SEQ1\&SEQ2$ for verdadeira.

A abordagem baseada em árvores, proposta inicialmente por Mei e Madden (2009), endereça algumas das limitações dos NFAs. As principais são a simplificação dos operadores de conjunção e negação, e a flexibilização da ordem de avaliação dos eventos primitivos. Além disso, essa abordagem permite armazenar eventos compostos nos nodos intermediários, o que simplifica a implementação de operadores.

Por fim, na abordagem (iii) as definições de eventos complexos são representados usando formalismos lógicos, como lógica temporal e cálculo de eventos (AA; ARTIKIS; WEIDLICH, 2022).

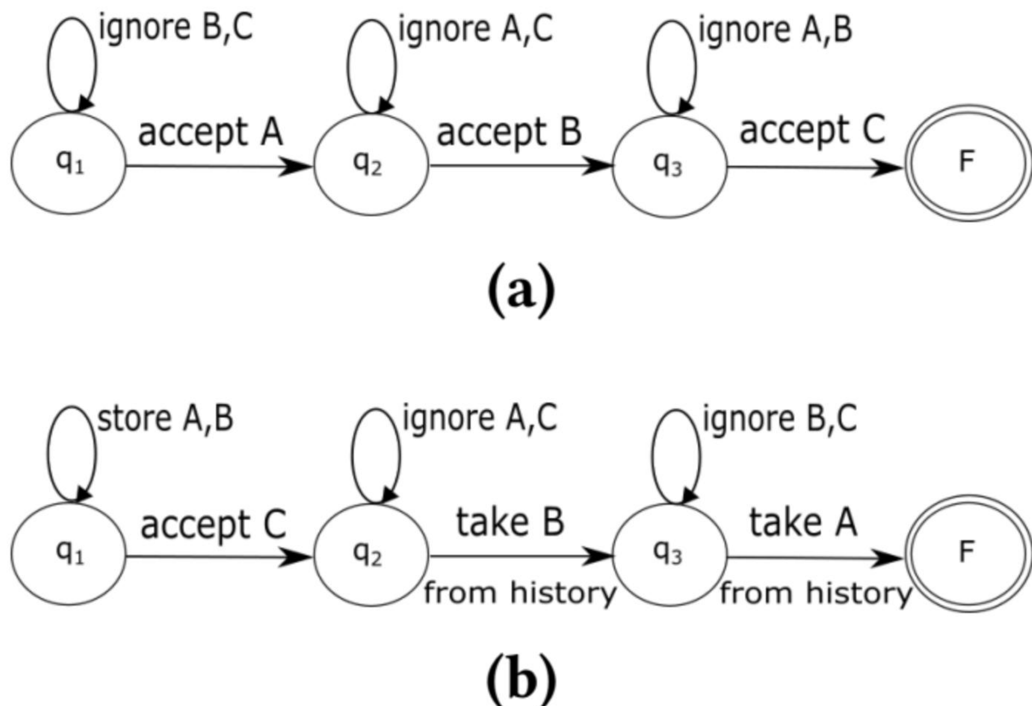
2.2.2.1 Desafios e Otimizações

Cada abordagem possui desafios e otimizações específicas. No entanto, existem alguns desafios comuns. Um deles é a quantidade de estados intermediários. Esses estados impactam o desempenho de sistemas CEP, pois é necessário recalcular o conjunto de estados intermediários a cada novo evento, e mesmo para eventos complexos simples a quantidade pode facilmente se tornar exponencial em relação ao número N de eventos de previamente processados. Portanto, a cada novo evento seria necessário executar computação superlinear em N, o que é incompatível com os requisitos de latência desejados nesse tipo de sistema (BUCCHI *et al.*, 2021).

Hirzel *et al.* (2014) catalogam algumas otimizações que podem ser aplicadas em sistemas CEP. Dentre elas, as mais relevantes para o escopo deste trabalho são *pattern rewriting* e *load shedding*. O *pattern rewriting* utiliza técnicas estatísticas para derivar uma versão mais eficiente do evento complexo. O *load shedding* descarta eventos primitivos ou estados intermediários com baixa probabilidade de se tornarem um evento complexo. Essas otimizações são aplicadas em tempo de execução, e portanto não alteram a semântica do evento complexo.

O *pattern rewriting* é uma técnica bastante conhecida (KOLCHINSKY; SCHUSTER, 2019). Dentro desta categoria, alguns métodos específicos de otimização são *pattern reordering* e o *pattern sharing*.

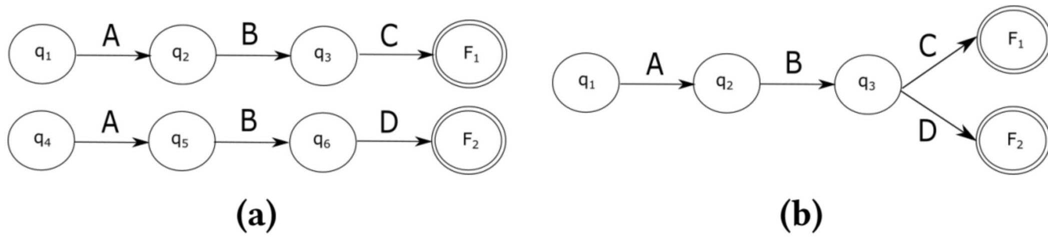
Figura 6 – *Pattern reordering*



Fonte: Kolchinsky e Schuster (2019)

O *pattern reordering* organiza a sequência de eventos primitivos que compõem o evento complexo (*pattern*) de tal forma que as operações mais leves em termos de volume de estados intermediários sejam executadas primeiro. Na Figura 6, o padrão é reescrito de $A \rightarrow B \rightarrow C$ (a) para $C \rightarrow B \rightarrow A$ (b), pois C ocorre menos que B, que ocorre menos que A.

Figura 7 – *Pattern sharing*



Fonte: Kolchinsky e Schuster (2019)

O *pattern sharing*, por sua vez, aproveita as similaridades entre diferentes eventos complexos para unificar o processamento de subexpressões. No exemplo da Figura 7, os grafos apresentados em (a) são bastante similares: ambos esperam a ocorrência dos eventos A e B. Portanto, um novo grafo é apresentado em (b) unificando o que há em comum.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta e compara as propostas de Kolchinsky e Schuster (2019), Slo *et al.* (2019) e Mei e Madden (2009) para otimizar o desempenho de processadores de eventos complexos através da redução da quantidade de estados intermediários. Estes trabalhos foram selecionados através de uma estratégia de busca. A estratégia iniciou com a definição de palavras-chave (*complex event processing, stream processing, stream analytics*) e termos relevantes (*optimization, methods*). Após as definições iniciais, foram efetuadas buscas em algumas bases de dados bibliográficos (DBLP[1] e Google Scholar[2]). Os artigos foram filtrados por título, resumo e breve leitura da introdução para manter somente aqueles com foco próximo ao objetivo deste trabalho.

3.1 REAL-TIME MULTI-PATTERN DETECTION OVER EVENT STREAMS

Os autores Kolchinsky e Schuster (2019) propõem um novo framework de CEP que utiliza uma combinação de duas técnicas de otimização, *pattern reordering* e *pattern sharing*, para minimizar a quantidade de estados intermediários. A quantidade desses estados tem influência direta no desempenho de processadores CEP, pois um número elevado deles reduz muito a eficiência do processador.

Ainda que efetiva na minimização de estados intermediários, a proposta de Kolchinsky e Schuster (2019) não lida com o seu impacto. Portanto, ainda existe risco de degradação caso a premissa usada na simplificação do grafo seja invalidada. Por exemplo, na Figura 6, a reescrita parte do premissa que, se a ocorrência de C for menor, é mais barato processá-lo primeiro. Porém, essa premissa pode deixar de ser verdadeira em algum momento. Além disso, ainda é necessário armazenar A e B para que seja possível consultá-los nos estados q_2 e q_3. Logo, dependendo do volume desses eventos, iterar cada ocorrência pode se tornar custoso.

Este trabalho se diferencia da solução aqui proposta pois visamos reduzir o custo dos estados intermediários, não só o seu volume. Também propomos um novo modelo de processamento capaz de maximizar a reutilização de subexpressões sem reescrever o evento complexo.

3.2 PSPICE: PARTIAL MATCH SHEDDING FOR COMPLEX EVENT PROCESSING

Neste trabalho, Slo *et al.* (2019) propõem uma estratégia para lidar com degradações de sistemas CEP através da eliminação de parte do estado do processador. Diferente das abordagens de *load shedding*, como a apresentada em Hirzel *et al.* (2014), a estratégia proposta por Slo *et al.* (2019) descarta os estados intermediários com menor probabilidade de se tornarem uma ocorrência de evento complexo. Nas estratégias de *load shedding* convencionais, são os eventos que normalmente são descartados.

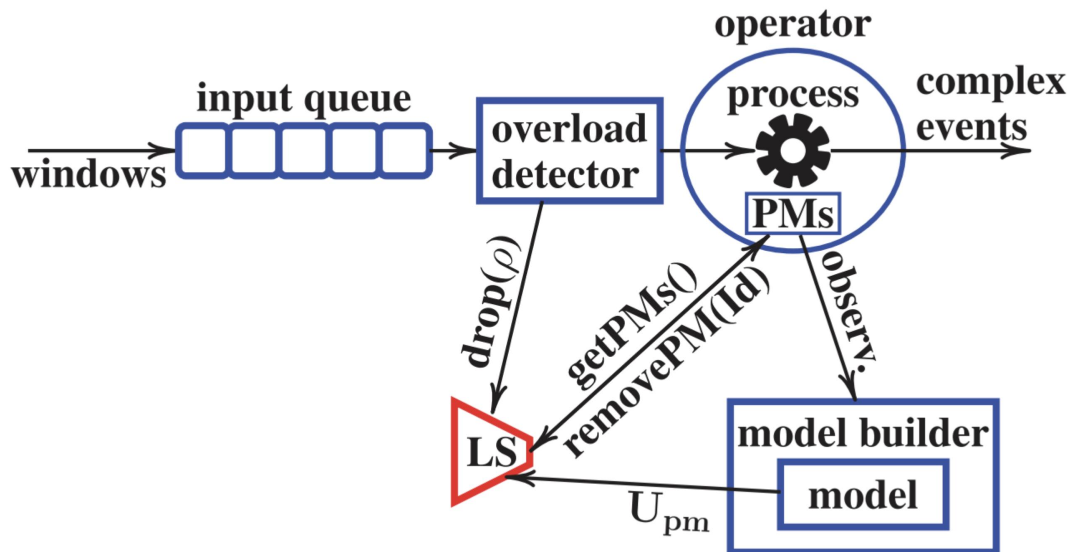
---

[1] https://dblp.org/

[2] https://scholar.google.com/

A arquitetura do proposta, representada na Figura 8, é composta por três componentes principais: um detector de sobrecarga (*overload detector*), o responsável pela limpeza do estado (*Load Shedder*, ou apenas LS), e o responsável pela predição da importância de cada estado intermediário (*model builder*).

Figura 8 – Arquitetura do pSPICE



Fonte: Slo *et al.* (2019)

Os autores defendem que a eliminação de uma fração do estado é igualmente efetiva na redução de carga do sistema, mas impacta menos a dos resultados do que a eliminação de eventos. Entretanto, ainda que o impacto seja menor, a qualidade dos resultados (QoR) ainda pode ser prejudicada. Além disso, o processo de limpeza de estado pode sobrecarregar ainda mais o sistema.

A solução proposta se diferencia deste trabalho pois explora otimizações sem impacto na qualidade dos resultados, e que não sobrecarregam o processamento de eventos.

3.3 ZSTREAM: A COST-BASED QUERY PROCESSOR FOR ADAPTIVELY DETECTING COMPOSITE EVENTS

Os autores Mei e Madden (2009) propõem um novo mecanismo de avaliação baseado em árvores. Este trabalho difere da maioria dos trabalhos recentes, que utilizam autômatos finitos não-determinísticos. Neste novo mecanismo, um evento complexo produz uma ou mais árvores com diferentes custos computacionais. A escolha da árvore acontece em tempo de execução, utilizando uma função de custo, e pode ser alterada a qualquer instante de forma transparente. Segundo Kolchinsky e Schuster (2019), este foi o primeiro trabalho a descrever um mecanismo desse tipo.

A proposta dos autores se mostra rápida em uma série de consultas (eventos complexos). Entretanto, seu funcionamento depende de um buffer em memória, o que sugere que certos tipos

de carga de trabalho podem ser um problema. Por exemplo, uma consulta que relaciona dois tipos de eventos, $T1$ e $T2$, em uma janela de tempo grande, pode gerar um grande acúmulo de eventos e potencialmente consumir toda a memória disponível. Além disso, é possível que o desempenho varie de acordo com a quantidade de eventos nos buffers.

Este trabalho se diferencia da solução aqui proposta pois não precisamos armazenar os eventos, em memória ou em disco, para relacioná-los ou filtrá-los.

3.4 COMPARATIVO

A Tabela 1 apresenta um sumário das principais diferenças entre os trabalhos relacionados e a ferramenta proposta. Os trabalhos de Kolchinsky e Schuster (2019) e Slo *et al.* (2019) representam os eventos complexos e estados intermediários juntos, como autômatos finitos não-determinísticos, enquanto Mei e Madden (2009) os representam utilizando árvores. A ferramenta proposta utiliza árvores para indexar os eventos complexos, porém, armazena os estados intermediários em um banco de dados.

Os critérios usados na comparação foram:

- Peso dos estados intermediários: capacidade de escalar independente da quantidade de eventos processados interiormente;
- Quantidade de estados intermediários: capacidade de reduzir a quantidade de informação necessária para correlação;
- Sobrecarga: capacidade de otimizar o desempenho sem sobrecarregar o sistema;
- Impacto na QoR: capacidade de otimizar o desempenho sem perder qualidade dos resultados;
- Armazenar eventos: capacidade de correlacionar eventos primitivos sem armazená-los. Os efeitos colaterais do processamento do evento são persistentes, mesmo que os eventos não sejam persistidos;

Tabela 1 – Comparação entre os trabalhos relacionados e a ferramenta proposta

Trabalho	**Reduz peso dos estados intermediários**	**Reduz quantidade de estados intermediários**	**Sem sobrecarga**	**Sem impacto na QoR**	**Sem armazenar eventos**
Kolchinsky e Schuster (2019)		✓	✓	✓	✓
Slo *et al.* (2019)		✓			✓
Mei e Madden (2009)		✓	✓	✓	
Ferramenta proposta	✓	✓	✓	✓	✓

Conforme ilustra a Tabela 1, o diferencial da ferramenta proposta é a sua capacidade de minimizar a quantidade e o peso dos estados intermediários sem sobrecarga no processamento ou impacto na qualidade dos resultados. Outra característica importante é que não precisamos manter uma cópia dos eventos. A proposta deste trabalho é apresentada no próximo capítulo.

4 PROPOSTA

Este capítulo apresenta um método diferente de processamento que minimiza o impacto de estados intermediários na escalabilidade de sistemas CEP. Esse método combina diversas técnicas de otimização, como políticas de seleção de eventos e compartilhamento de estado. As principais contribuições deste trabalho são: (i) uma nova representação de estado intermediário que os dissocia dos eventos complexos e possibilita armazená-los em um sistemas de bancos de dados; (ii) um modelo de dados para a escrita e leitura eficiente de estados intermediários; e (iii) uma versão experimental desse processador escrita em Go[1].

Por questões de simplificação, apenas alguns operadores (sequência, conjunção, disjunção e negação) serão suportados pela versão inicial do processador. Apesar de limitado, veremos a seguir que a combinação desses operadores possibilita ao processador representar uma vasta gama de eventos complexos. Esses operadores também podem ser usados em janelas de tempo deslizantes, que permitem delinear um período de tempo no qual todas as condições precisam ser satisfeitas. Neste momento ainda não há suporte para operadores multiclasse (que usam dois tipos diferentes de eventos) e operações de agregação (soma, máximo, mínimo, percentis, etc.)

As próximas seções detalham o seu funcionamento, incluindo um exemplo em execução para demonstrar seu potencial.

4.1 VISÃO GERAL

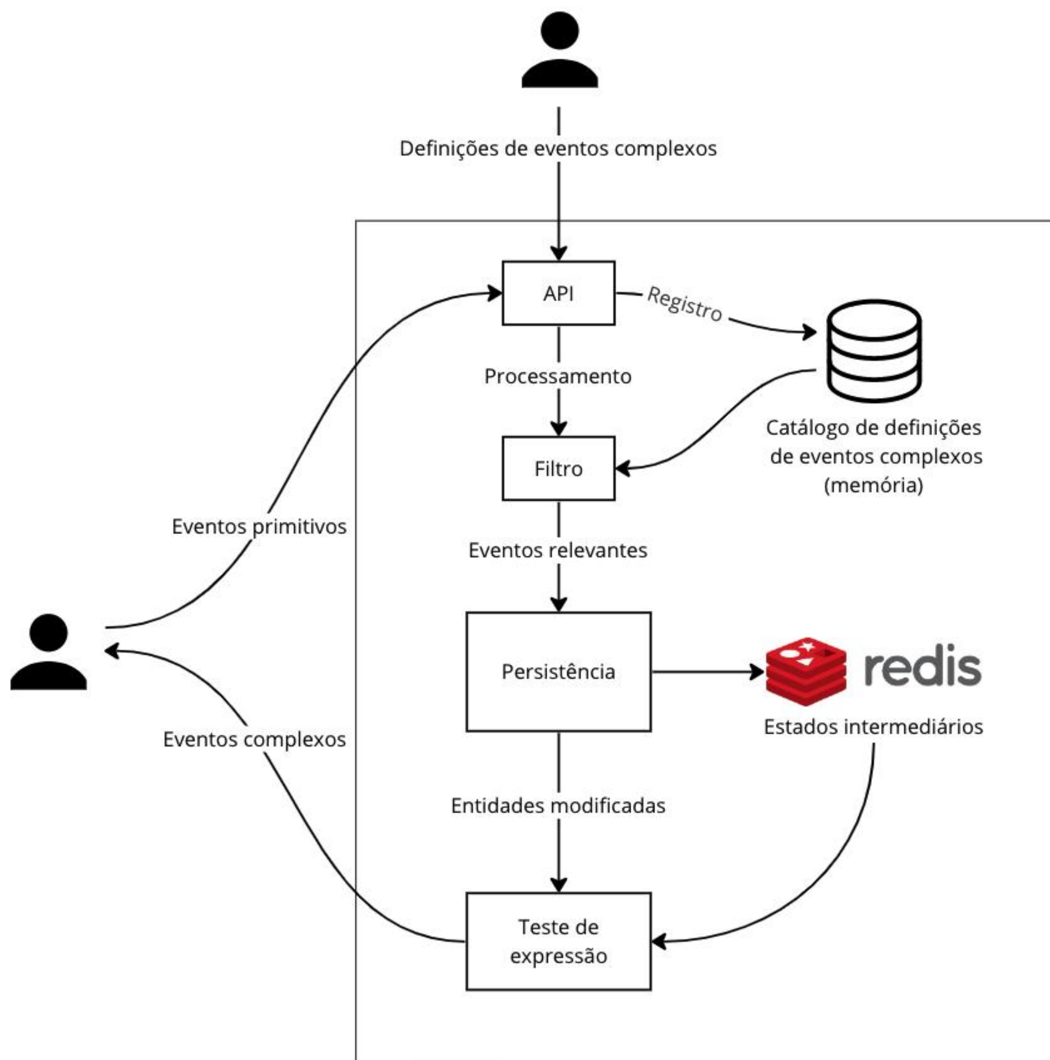
O método proposto divide o estado do processador em duas partes: (i) catálogo de definições de eventos complexos (DEC); e (ii) estados intermediários. O catálogo possui um índice em memória que permite verificar quais condições (as chamaremos de predicados), dentre todas as DECs, um evento primitivo satisfaz. Já os estados intermediários são armazenados em um banco de dados e representam quais predicados são atualmente satisfeitos.

A Figura 9 ilustra a arquitetura do processador. Além do catálogo e do banco de dados usado pra armazenar os estados intermediários, o processador possui uma API para registro de DECs e processamento de eventos primitivos. O processamento acontece em três etapas: (i) filtro; (ii) persistência e (iii) teste da expressão. Na etapa (i), são selecionados apenas os eventos relevantes para algum predicado. Na etapa (ii), os estados intermediários são criados. Na etapa (iii), é verificado quais das DECs impactadas possuem todas as condições satisfeitas, e o resultado é retornado pela API. As DECs, representadas como um objeto JSON[2], são registradas a priori por um usuário especialista, e os eventos primitivos são enviados para a API por um agente externo.

[1] https://go.dev/

[2] https://www.json.org/json-en.html

Figura 9 – Arquitetura do processador



Uma DEC é composta por um identificador, algumas configurações e uma expressão lógica muito semelhante à **seleção** da álgebra relacional. Na expressão da Figura 10, os textos em amarelo são os predicados (“A” e “B”). O texto em azul é um operador lógico (“AND“), usado para representar operações de conjunção. Por fim, o texto em verde é o operador unário “NOT”, que representa a não-ocorrência da subexpressão (neste caso, “B”).

Figura 10 – Expressão de um evento complexo

A B
(E'.tipo == "viu_video" AND E'.duracao > 10) AND NOT (E''.tipo = "assinou_canal")

Para fins de exemplificação do funcionamento do processador, suponha a ocorrência de três eventos primitivos: (i) visita de uma página; (ii) abertura de um e-mail; (iii) *click* em um link. Suponha também a existência de uma única DEC no catálogo, caracterizada pela ocorrência dos seguintes eventos primitivos:

A) Evento do tipo “abriu_email” com “periodo_segundos” maior que 10 e “dispositivo” igual a “telefone”;

B) Evento do tipo “clicou_link” com “url” igual a “http://site.org/quero-comprar”.

A Figura 11 exemplifica como seria o objeto JSON desta DEC. O primeiro predicado representa a condição *A* (evento do tipo “abriu_email” com “periodo_segundos” maior que 10 e “dispositivo” igual a “telefone”). O segundo predicado representa a condição *B* (evento do tipo “clicou_link” com “url” igual a “http://site.org/quero-comprar”). As condições de um predicado não precisam de conector pois parte-se do princípio que todas precisam ser verdadeiras para que ele seja impactado por um evento.

Figura 11 – Definição de um evento complexo

```
{
    "id": "complex-event-1",
    "logical_expression": {
        "connector": "and",
        "operands": [
            {
                "predicate": {
                    "id": "A",
                    "event_type": "abriu_email",
                    "conditions": [
                        {
                            "field": "dispositivo",
                            "operator": "equal",
                            "value": "telefone"
                        },
                        {
                            "field": "periodo_segundos",
                            "operator": "more_than",
                            "value": 10
                        }
                    ]
                }
            },
            {
                "predicate": {
                    "id": "B",
                    "event_type": "clicou_link",
                    "conditions": [
                        {
                            "field": "url",
                            "operator": "equal",
                            "value": "http://site.org/quero-comprar"
                        }
                    ]
                }
            }
        ]
    }
}
```

Suponha agora que surgiu um evento primitivo (i) mostrado na Figura 12. Ele será descartado pelo processador, pois seu tipo, “visitou_pagina”, é irrelevante para a definição de evento

complexo no catálogo, uma vez que os únicos tipos de eventos primitivos potencialmente relevantes são “abriu_email” e “clicou_link”. Consequentemente, não emitimos nenhuma notificação durante o seu processamento.

Figura 12 – Evento (i): visita em uma página

```
{
    "id": "1",
    "type": "visitou_pagina",
    "timestamp": 1000000000,
    "payload": {
        "url": "http://site.org/quero-comprar"
    }
}
```

Suponha agora um evento (ii) mostrado na Figura 13. Este evento já é relevante pois ele possui características presentes na definição de evento complexo (“type” igual a “abriu_email”, “dispositivo” igual a “telefone” e “período_segundos” maior que 10). No entanto, neste momento apenas uma das condições (A) do evento complexo foi satisfeita. Portanto, nenhuma notificação é emitida durante o processamento.

Figura 13 – Evento (ii): abertura de e-mail

```
{
    "id": "2",
    "type": "abriu_email",
    "timestamp": 1000000000,
    "payload": {
        "dispositivo": "telefone",
        "periodo_segundos": 60
    }
}
```

Por fim, suponha o evento (iii), mostrado na Figura 14. Este evento também é relevante pois possui características presentes na definição do evento complexo (“type” igual a “clicou_link” e “url” igual a “http://site.org/quero-comprar”). Neste momento, ambas as condições (A e B) são satisfeitas e uma notificação é enviada informando que esse evento complexo ocorreu.

Figura 14 – Evento (iii) click

```
{
    "id": "3",
    "type": "clicou_link",
    "timestamp": 1000000000,
    "payload": {
        "url": "http://site.org/quero-comprar"
    }
}
```

As próximas seções detalham cada um dos componentes do processador.

4.2 CATÁLOGO

O catálogo é composto por três componentes: (i) um hashmap com referências para todas as DECs; (ii) um hashmap que relaciona predicados e DECs; e (iii) um índice do tipo árvore onde são indexados os predicados. Esse índice é usado posteriormente na filtragem de eventos. O algoritmo de indexação utilizado é o Algorítmo 1.

Algorithm 1: Algoritmo de Indexação

Input: árvore de predicados T, hashmap ligando os predicados e DECs PD, definição de eventos complexos d

Output: árvore de predicados T atualizada, hashmaps PD atualizado

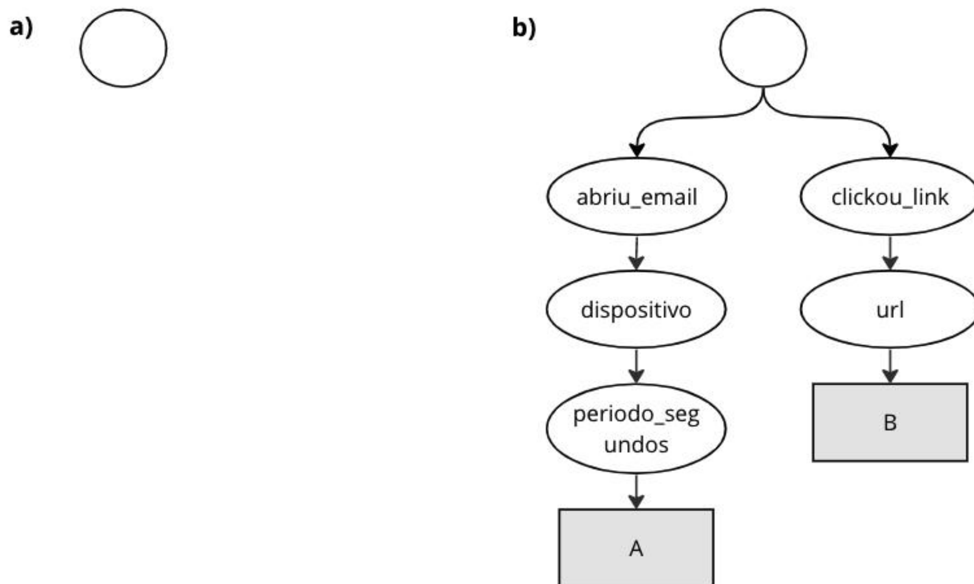
```
1  foreach p ∈ predicates(d) do
       /* Deriva um ID para o predicado com base no seu conteúdo.
          Dois predicados iguais devem sempre gerar o mesmo ID     */
2      p.id ← hash(p)
       // Relaciona o predicado p a DEC d
3      PD_p ← PD_p ∪ {d}
       /* Extrai os campos usados nas condições. Por exemplo, para um
          evento complexo com condições dispositivo=Android,
          periodo_segundos=10 essa função retornaria ⟨ dispositivo,
          periodo_segundos ⟩. A lista está sempre em ordem
          lexicográfica                                              */
4      ck ← extractConditionFields(p)
       /* Inicializa o caminho K com o tipo do evento e adiciona as
          chaves das condições ck. O resultado é algo como K = ⟨
          abriu_email, dispositivo, periodo_segundos ⟩              */
5      K ← ⟨p.eventType⟩ ∪ ck
       // Percorre o caminho K dentro da árvore e retorna o nodo
          final. Os nodos que não existirem são criados
6      n ← findNode(T, K)
7      n.predicates ← n.predicates ∪ {p}
8  return (T, PD)
```

No Algorítmo 1, a função *hash* é responsável por gerar um identificador único para o predicado com base no seu conteúdo. Uma propriedade importante dessa função é que predicados iguais sempre geram o mesmo ID, minimizando a quantidade de predicados. Veremos a seguir que essa propriedade é essencial para a escalabilidade solução.

A Figura 15 mostra o estado do índice de predicados antes (“a”) e depois (“b”) da execução do Algorítmo 1 para a DEC da Figura 11. O predicado A está em $abriu_email \rightarrow dispositivo \rightarrow periodo_segundos$, sendo relevante quando houverem os campos “dispositivo”e “periodo_-segundos” em um evento do tipo “abriu_email”. O predicado B está em $clicou_link \rightarrow url$,

sendo relevante sempre que houver um campo “url” em um evento de “clicou_link”.

Figura 15 – Índice de predicados



Assumindo que a árvore não esteja vazia, ela sempre terá pelo menos um nível: o tipo do evento. Esse nível nos permite descartar mais rápido os eventos irrelevantes, pois já no primeiro nível é possível saber se há ou não algum predicado para o tipo do evento sendo processado. A próxima seção detalha a utilização do catálogo no processamento de eventos.

4.3 PROCESSO DE FILTRO

A primeira etapa do processamento consiste em um filtro para descartar eventos irrelevantes. Neste contexto, são irrelevantes os eventos cujo conteúdo não atende as condições de nenhum predicado de nenhuma DEC do catálogo. Por exemplo, um evento do tipo “clicou_link” seria irrelevante para o predicado *A* da Figura 11, porque ele só está interessado em eventos do tipo “abriu_email”. Seria igualmente irrelevante tanto para o predicado *A* quanto para o predicado *B* um evento cujo valor do campo “dispositivo” seja “computador”. O descarte é importante para economizar recursos computacionais que seriam usados no processamento destes eventos.

Ao final desta etapa sabemos quais eventos primitivos são relevantes para alguma DEC, e o porquê. Seu resultado é um conjunto de tuplas, compostas por um evento, quais predicados foram impactados por esse evento, e quais DECs usam esses predicados. São essas tuplas que são encaminhadas para a próxima etapa. Os eventos irrelevantes geram tuplas sem nenhum predicado que devem ser descartadas.

O processo de filtro, detalhado no Algorítmo 2, é dividido em três subetapas: (i) identificar predicados potencialmente relevantes; (ii) verificar se as condições do predicado são todas verdadeiras, e então; (iii) buscar as expressões que utilizam os predicados encontrados. A subetapa (i), detalhada no Algorítmo 3, tem por objetivo identificar quais predicados podem

estar interessados no evento que está sendo processado. Na Figura 16, por exemplo, o evento foi considerado irrelevante pelo predicado A da Figura 11 por não possuir um campo “dispositivo”. Já se, por exemplo, registrarmos um segundo evento complexo (Figura 17), interessado em eventos contendo somente o campo “periodo_segundos”, o evento da Figura 16 seria considerado relevante, pois este campo está presente em uma condição. É importante ressaltar que nesta etapa ainda não verificamos o conteúdo das condições, somente a presença dos campos.

Algorithm 2: Algoritmo de Busca

Input: árvore de predicados T, evento primitivo e

Output: lista de predicados impactados pelo evento e

`/* Função similar à` *`extractConditionFields`* `utilizada no Algorítmo 1, porém com um tipo diferente. Ao invés de receber um predicado, recebe um evento. A lista está sempre em ordem lexicográfica */`

1 $pk \leftarrow extractPayloadKeys(e)$

`/* Inicializa o caminho` K `com o tipo do evento e adiciona as chaves do payload` pk`. O resultado será algo como` $K = \langle$ `abriu_email, dispositivo, periodo_segundos` $\rangle$ `*/`

2 $K \leftarrow \langle p.eventType \rangle \cup pk$

`// Percorre todas as permutações de K e retorna todos os predicados encontrados pelo caminho. Esse algoritmo é descrito no Algorítmo 3`

3 $P \leftarrow searchPredicates(T, K)$

4 $P' \leftarrow \emptyset$

5 **foreach** $p \in P$ **do**

  `/* Esta função retorna verdadeiro se todas as condições do predicado` p `forem satisfeitas pelo evento` e `*/`

6 **if** $evaluateConditions(p, e)$ **then**

7   $P' \leftarrow P' \cup \{p\}$

8 **return** P'

Figura 16 – Evento irrelevante

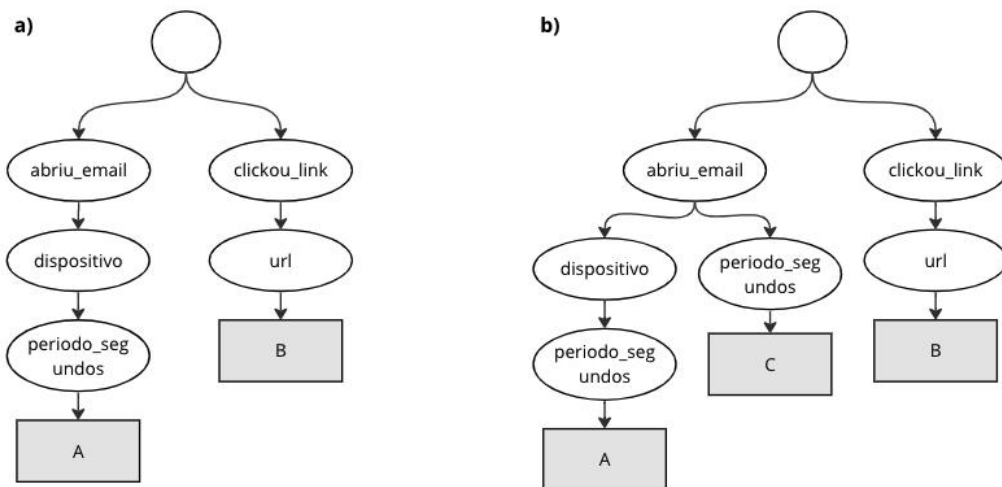

```
{
    "id": "4",
    "type": "abriu_email",
    "timestamp": 1,
    "payload": {
        "periodo_segundos": 60
    }
}
```

Figura 17 – Predicado sem o campo “dispositivo”

```
{
    "id": "complex-event-2",
    "logical_expression": {
        "connector": "and",
        "operands": [
            {
                "predicate": {
                    "id": "C",
                    "event_type": "abriu_email",
                    "conditions": [
                        {
                            "field": "periodo_segundos",
                            "operator": "more_than",
                            "value": 10
                        }
                    ]
                }
            }
        ]
    }
}
```

A Figura 18 mostra o estado do índice antes (“a”) e depois (“b”) da execução do Algorítmo 1 para a DEC da Figura 17. O predicado C foi indexado em $abriu_email \rightarrow periodo_segundos$ pois não possui o campo “dispositivo”.

Figura 18 – Índice de predicados



Na subetapa (ii), detalhada na linha do 6 do Algorítmo 2, o objetivo é descobrir quais predicados possuem todas as condições verdadeiras. No caso do predicado A da Figura 11, por exemplo, o evento precisa ter “dispositivo” igual a “telefone” e “periodo_segundos” maior que 10. Os predicados que não tiverem todas as condições verdadeiras serão desconsiderados.

Por fim, na subetapa (iii), detalhada na linha 7 do Algorítmo 2, identifica-se quais expressões utilizam os predicados impactados, pois se um evento EV_i modificar o estado de um predicado P_j, então os resultados das expressões que usam P_j podem ter sido alterados por EV_i. Nesta

etapa ainda não sabemos efetivamente se houve ou não mudança no resultado da expressão por que para tal precisamos testá-la novamente. O foco desta etapa é apenas identificar quais expressões precisam ser testadas.

Algorithm 3: Algoritmo auxiliar "$searchPredicates$"

Input: árvore de predicados T e um caminho a ser percorrido K

Output: todos os predicados encontrados no caminho K

```
1 P ← ∅
2 foreach p ∈ T.predicates do
3     P ← P ∪ p
4 foreach (i, k) ∈ K do
      // Subárvore identificada pela chave k
5     T′ ← T_k
6     if T′ ≠ ∅ then
          // Percorre verticalmente a árvore para todas as chaves que
             ainda faltam visitar
7         P ← P ∪ searchPredicates(T′, K_{i..|K|})
8 return P
```

4.4 PROCESSO DE PERSISTÊNCIA

A segunda etapa do processamento consiste em persistir os estados intermediários de modo que no futuro seja possível saber quais condições para a ocorrência de um evento complexo já foram satisfeitas. Só chegam nesta etapa as tuplas de eventos que satisfazem ou invalidam algum predicado, pois sem que haja mudança no estado é impossível que o resultado de uma expressão seja alterado. Por exemplo, só chegam nesta etapa os eventos (ii) e (iii) (Figura 13 e Figura 14), que satisfazem, respectivamente, os predicados A e B da Figura 11. O evento (i) (Figura 12) é descartado porque não satisfaz ou invalida nenhum predicado.

Por questões de simplicidade, o banco de dados escolhido para a persistência dos estados intermediários foi o Redis[3]. O Redis é um repositório de dados em memória, fácil de usar e com padrões de acesso chave-valor extremamente rápidos. Ele está amplamente disponível em qualquer *cloud* e pode ser executado em qualquer computador usando o Docker[4]. Além disso, ele possui uma API simples e bem documentada, o que facilita a implementação do processador.

O algoritmo de persistência é descrito no Algorítmo 4. O modelo de dados utilizado é descrito a seguir, na Subseção 4.4.1.

[3] https://redis.io/

[4] https://www.docker.com/

Algorithm 4: Algoritmo de Persistência.

Input: lista de predicados impactados P, hashmap com a relação entre predicados e expressões PD, evento primitivo e

Output: DECs que precisam ser testadas D

```
   // Timestamp que será usado para versionar o estado.
1  t ← e.timestamp
   // Inicializa S e D como listas vazias.
2  S ← ∅
3  D ← ∅
   // Para cada predicado p.
4  foreach p ∈ P do
       // Define o predicado p como verdadeiro no momento t.
5      S_{p,t} ← true
       // Testar todas as DECs associadas ao predicado p.
6      D ← D ∪ PD_p
   // Persiste o estado no banco de dados.
7  save(S)
8  return D
```

Na linha 5 do Algorítmo 4, é especificado o momento em que estado intermediário foi criado. Essa informação é necessária para que seja possível suportar janelas de tempo e o operador de sequência. Por exemplo, considere as seguintes DECs: (i) "A e B em um intervalo de 30 segundos" e (ii) "A depois de B". No caso (i), temos uma janela de tempo. Neste exemplo precisamos saber para um evento qualquer EV_i que impactou A ou B, se houve, respectivamente, um B ou A a 30 segundos de distância. No caso (ii), temos um operador de sequência. Nele precisamos verificar para um evento qualquer EV_j que impactou A ou B, se há, respectivamente, um B ou A posterior. Em ambos os casos é preciso saber quando A e B ocorreram.

4.4.1 Modelo de dados

A estrutura de dados *sortedset* do Redis é uma ótima alternativa para modelar séries temporais de dados simples. A maioria das operações sobre *sortedsets* possuem complexidade de tempo O(log(N))[5], o que escala bem para caso de uso proposto. Nesta modelagem, a chave, o *score* e o valor serão, respectivamente, o ID do predicado, o *timestamp* do evento, e o ID do evento.

A Figura 19 ilustra o comando usado para adicionar um item no *sortedset*. No exemplo, o evento "2" impactou o predicado A no *timestamp* "1000000000".

[5] https://redis.io/commands/?group=sorted-set

Figura 19 – Adicionando um item a um *sortedset*

```
127.0.0.1:6379> ZADD A 1000000000 2
(integer) 1
127.0.0.1:6379>
```

A Figura 20 ilustra o comando usado para buscar um item no *sortedset*. No exemplo, será retornado para os predicados A e B se algum evento os impactou entre os *timestamps* "999999700" e "1000000300". No caso, o evento "2" impactou o predicado A no *timestamp* "1000000000". Nenhum evento impactou o predicado B no intervalo de tempo especificado.

Figura 20 – Lendo um item de um *sortedset*

```
127.0.0.1:6379> ZRANGE A 999999700 1000000300 BYSCORE LIMIT 0 1 WITHSCORES
1) "2"
2) "1000000000"
127.0.0.1:6379> ZRANGE B 999999700 1000000300 BYSCORE LIMIT 0 1 WITHSCORES
(empty array)
127.0.0.1:6379>
```

Diferentemente dos processadores baseados em NFA e em árvore, os estados intermediários não são o estado atual de uma instância de evento complexo. Em vez disso, são os predicados que já foram satisfeitos. Essa abordagem maximiza o compartilhamento de estado entre diferentes DECs, pois, durante o processamento de um evento, apenas um estado intermediário será criado mesmo que centenas de DECs usem o mesmo predicado. Outra vantagem dessa abordagem é que ela permite que o processador correlacione eventos primitivos sem armazená-los. Isso é possível pois o estado intermediário contém tudo que o processador precisa saber sobre os eventos primitivos que o criaram.

Considere a DEC da Figura 11, por exemplo. Independente da quantidade de eventos primitivos previamente processados, durante o processamento de um evento EV_i, no máximo dois valores precisariam ser lidos, A e B. Essa propriedade tem forte influência sobre a escalabilidade do processador, pois enquanto métodos convencionais escalam de acordo com a quantidade de eventos, o método proposto escala de acordo com a quantidade de predicados. Isso significa que o método proposto é extremamente eficiente em cenários onde há menos predicados que eventos, como por exemplo, cenários de IoT. Por outro lado, ele pode ser menos eficiente em cenários onde há muitos predicados, como por exemplo, cenários de análise de dados.

4.4.2 Alternativas

Os padrões de leitura e escrita utilizados pelo processador podem ser modelados em diversos tipos de sistemas de bancos de dados. Em bancos colunares, por exemplo, as linhas poderiam ser partições, como o ano atual, ou algum outro atributo do evento. As colunas poderiam ser os predicados, e cada célula (linha x coluna) da tabela poderia armazenar uma lista com os *timestamps* dos eventos. Duas opções populares nesta categoria são o Apache

Cassandra[6], criado pelo Facebook, e o Cloud Bigtable[7], do Google. Ambos são bancos de dados distribuídos altamente escaláveis, e suportam uma grande quantidade de colunas. No caso do Apache Cassandra, não há limite para o número de colunas, mas sim para a quantidade de células dentro de uma partição. O número máximo de células é de 2 bilhões[8]. No caso do Cloud Bigtable, não há limite no número de colunas, mas de tamanho de linha. O tamanho máximo de uma linha é de 256 MB[9]. Uma vantagem do Bigtable em relação ao Cassandra é que no Bigtable não é preciso definir o esquema da tabela, enquanto no Cassandra é preciso definir o nome das colunas e seus tipos. Outra vantagem do Bigtable é o suporte nativo a uma terceira dimensão, a versão do dado. Isso significa que seria possível armazenar cada ocorrência do predicado como uma versão, usando o *timestamp* do evento como identificador da versão.

A Figura 21 mostra a utilização da ferramenta *Command Line interface* (CLI) do Cloud Bigtable para especificar a estrutura da tabela, e depois ler e escrever dados. No exemplo, o “r1” é o ID da linha, “cf1” é uma família de colunas, “c1” é o nome da coluna, “val1” é o valor da célula e “2021/12/16-10:10:00.000000” é a sua versão.

Figura 21 – Ferramenta CLI do Cloud Bigtable

```
t3[283;0]run-yaml$ cbt createtable my-table
t3[284;0]run-yaml$ cbt createfamily my-table cf1
t3[285;0]run-yaml$ cbt set my-table r1 cf1:c1=val1@1639617000000000
t3[286;0]run-yaml$ cbt set my-table r1 cf1:c1=val2@1639616000000000
t3[287;0]run-yaml$ cbt read my-table
----------------------------------------
r1
  cf1:c1                                @ 2021/12/16-10:10:00.000000
    "val1"
  cf1:c1                                @ 2021/12/16-09:53:20.000000
    "val2"
t3[288;0]run-yaml$ cbt set my-table r1 cf1:c1=val3@1639615000000000
t3[289;0]run-yaml$ cbt read my-table
----------------------------------------
r1
  cf1:c1                                @ 2021/12/16-10:10:00.000000
    "val1"
  cf1:c1                                @ 2021/12/16-09:53:20.000000
    "val2"
  cf1:c1                                @ 2021/12/16-09:36:40.000000
    "val3"
t3[290;0]run-yaml$ cbt read my-table cells-per-column=1
----------------------------------------
r1
  cf1:c1                                @ 2021/12/16-10:10:00.000000
    "val1"
t3[291;0]run-yaml$
```

Fonte: Stackoverflow (2021)

As tabelas Tabela 2, Tabela 3 e Tabela 4 exemplificam como seria o estado inicialmente

[6] https://cassandra.apache.org/

[7] https://cloud.google.com/bigtable

[8] https://cassandra.apache.org/doc/latest/cassandra/data_modeling/data_modeling_refining.html

[9] https://cloud.google.com/bigtable/quotas#limits

(quando nenhum evento foi processado), após o processamento do evento (ii), e após o processamento do evento (iii), respectivamente. Bancos de dados colunares são adequados neste contexto onde o número de colunas pode variar com o passar do tempo.

Tabela 2 – Estado inicial

partition

Tabela 3 – Estado após evento (ii)

partition	**A**
2023	t@1

Tabela 4 – Estado após evento (iii)

partition	**A**	**B**
2023	t@1	t@1

4.5 PROCESSO DE TESTE DA EXPRESSÃO

A terceira e última etapa do processamento consiste em testar as expressões impactadas pela modificação no estado. Nesta etapa, o processador carrega do banco de dados todos os estados intermediários necessários para testar as expressões. Em seguida, para cada expressão, o processador testa se as condições especificadas na DEC são satisfeitas. O Algorítmo 5 detalha o processo de teste da expressão.

Algorithm 5: Algoritmo de Teste da Expressão

```
Input: lista de DECs cujas expressões devem ser testadas D, evento primitivo e
Output: hashmap com o resultado das expressões R
1  R ← ∅
2  foreach d ∈ D do
3      S ← ∅
       /* Delimita a janela de tempo na qual as condições devem ser
          satisfeitas. Assume-se para efeito de simplificação que a
          janela de tempo também esteja em segundos                  */
4      if d.within > 0 then
5          t_min ← e.timestamp − d.within
6          t_max ← e.timestamp + d.within
7      else
8          t_min ← −∞
9          t_max ← ∞
       /* Carrega o estado dos predicados da DEC d do banco de dados.
          O funcionamento dessa função será detalhado a seguir.      */
10     S ← load(d, t_min, t_max)
       // Invoca o algoritmo auxiliar Algorítmo 6 para testar a
          expressão
11     R_d ← evaluateExpression(S, d.logicalExpression)
12 return R
```

No Algorítmo 5, a função $load$ é responsável por carregar do banco de dados os estados dos predicados que compõem a expressão. A função recebe como parâmetro a DEC cuja expressão deve ser testada, o tempo mínimo e máximo que delimitam a janela de tempo na qual as condições devem ser satisfeitas. A função retorna um hashmap com o estado dos predicados, onde o valor é um booleano, e a chave é um array com os identificadores dos predicados. Por exemplo, se a DEC possui dois predicados, A e B, o hashmap retornado pela função terá dois elementos, um para cada predicado. O primeiro elemento terá como chave a tupla (A) e o segundo elemento terá como chave a tupla (B). O valor será verdadeiro quando houver alguma ocorrência de A ou B dentro da janela $t_{min}..t_{max}$.

A Figura 22 ilustra o carregamento do estado dos predicados. Neste exemplo (a), os predicados A e B possuem cinco e três ocorrências, respectivamente. Suponha o processamento de um evento com um *timestamp* igual a “400000”, e uma DEC com uma janela de tempo de “100000” segundos. Neste caso (b), ambos os predicados retornariam verdadeiro, pois ambos possuem ocorrências dentro da janela de tempo.

Figura 22 – Carregamento do estado dos predicados

A	B
100000	300000
200000	410000
300000	550000
400000	
500000	

a)

A	B
100000	300000
200000	410000
300000	550000
400000	
500000	

b)

A	B
100000	300000
200000	410000
300000	550000
400000	
500000	

c)

Há um outro caso que deve ser considerado. Quando dois predicados estão conectados por um operador de sequência ($B; A$), não basta que hajam ocorrências de ambos os predicados dentro da janela de tempo $t_{min}..t_{max}$. É preciso que as ocorrências estejam na ordem correta. Por exemplo, suponha que a expressão da DEC do exemplo anterior seja $B; A$. Neste caso (c), é preciso que haja um B entre $t_{min}..t_{max}$, e que haja um A entre $t_b..t_{max}$, onde t_b é o *timestamp* da primeira ocorrência de B dentro da janela de tempo. Neste caso, o hashmap resultante seria $\{(A) \rightarrow true, (B, A) \rightarrow true\}$.

O Algorítmo 6 detalha o algoritmo auxiliar que testa a expressão (função *evaluateExpression*). Uma expressão é composta por um ou mais operandos conectados por um operador. Um operando pode ser um predicado ou uma expressão lógica. Quando ele for uma expressão lógica, o algoritmo é invocado recursivamente (linha 4). O funcionamento do algoritmo para cada um dos operadores é detalhado a seguir.

Algorithm 6: Algoritmo auxiliar "*evaluateExpression*"

Input: seção do estado S, expressão lógica que deve ser testada e

Output: booleano resultante r

```
   // Array com o valor dos operandos da expressão. O valor é sempre
      um booleano
1  V ← ∅
2  foreach o ∈ e.operands do
3      if o.logicalExpression then
           // Invoca recursivamente o algoritmo para avaliar a
              subexpressão
4          V ← V ∪ evaluateExpression(S, o.logicalExpression)
5      else
6          p ← o.predicate
           /* Quando o conector é um operador de sequência, retorna
              uma lista P com os predicados que aparecem antes de p na
              expressão. Quando o conector é "and" ou "or", retorna
              uma lista vazia.                                      */
7          P ← p.dependsOn
           // Carrega o estado de p em S.
8          v ← S_{P∪{p}}
           // Inverte o valor do predicado se ele for negado
9          if o.negated then
10             v ← ¬v
11         V ← V ∪ {v}
12 r ← false
   // Conecta os valores com o operador lógico
13 if e.operator = "or" then
       // Retorna true quando algum dos valores for true
14     r ← ∃v ∈ V : v = true
15 else
       // Retorna true quando todos os valores são true
16     r ← ∀v ∈ V : v = true
17 return r
```

4.6 API

O processador possui uma API RESTful[10] com dois *endpoints*: "POST /expression" para o registro de DECs e "POST /event" para o processamento síncrono de eventos primitivos.

4.6.1 Registro

Este *endpoint* adiciona uma nova DEC ao catálogo. A Figura 23 ilustra a estrutura de uma requisição e o Código 1 ilustra o retorno.

Figura 23 – Estrutura da requisição de registro

```
curl -X POST localhost:8080/expression \
    -H 'Content-Type: application/json ' \
    -d '{
   "id": "complex-event-1",
   "tenant_id": "1",
   "logical_expression": { ... }
 }'
```

Código 1 – Formato da resposta para requisição de registro.

```
1 { "ok": true }
```

O catálogo precisa ser mantido em memória. Logo, ele está sujeito a perda total no caso de falhas ou interrupções. Por esse motivo o processador mantém uma cópia de segurança desse catálogo no sistema de arquivos. Essa cópia é atualizada a cada registro de DEC, e automaticamente carregada em caso de reinicialização.

4.6.2 Processamento

Este *endpoint* processa sincronamente um evento. Ele recebe um evento primitivo e retorna o resultado das expressões impactados pelo evento. A Figura 24 ilustra a estrutura de uma requisição.

Figura 24 – Estrutura da requisição de processamento

```
curl -X POST localhost:8080/event \
    -H 'Content-Type: application/json ' \
    -d '{
  "id": "1",
  "tenant_id": "1",
  "entity_id": "1",
  "type": "abriu_email",
  "payload": {
    "email": "john@example.org",
  }
}'
```

[10] https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api

No exemplo do Código 2, o resultado para a DEC "complex-event-1" é falso porque, apesar de ter sido impactada pelo evento, as condições para a sua ocorrência ainda não foram satisfeitas. Já o resultado para a DEC "complex-event-2" é verdadeiro, sinalizando a sua ocorrência.

Código 2 – Formato da resposta para requisição de processamento.

```
1 {
2     "complex-event-1":false,
3     "complex-event-2":true
4 }
```

5 EXPERIMENTOS

Este capitulo descreve experimentos realizados com o processador de eventos proposto neste trabalho. Para isso, foi utilizada uma implementação em Go, disponível no Github[1]. Os experimentos utilizaram um *dataset*[2] de transações da NASDAQ. O *dataset* é uma amostra de 224.473 transações de compra e venda de ações, e possui as seguintes informações: código da ação, ID da transação, volume, preço, hora (convertida para *timestamp*), tipo (compra[B]/venda[S]). Todos os testes rodaram por 120 segundos, e durante esse período um mesmo evento pode ser processado duas vezes. Por esse motivo, não utilizamos o ID da transação presente no *dataset* e sim uma *string* randômica. Isso garante que o processador entenda que se trata de um evento diferente e o processe novamente.

O Apache JMeter[3] foi utilizado para carregar o *dataset* e simular requisições HTTP para o processador a partir de 200 threads. Foram também utilizados o *Prometheus*[4] para a coleta de métricas e o *Grafana*[5] para visualização. O processador foi executado em um container Docker com 8 CPUs, 8GB de RAM, 1GB de swap e 200GB de disco, como mostrado na Figura 25. Os containers rodaram de um Macbook Air M1 com 8 núcleos, 4 de performance e 4 de eficiência, e 16GB de RAM. O Redis rodou como um container separado, dentro da mesma rede.

---

[1] https://github.com/jeangnc/complex-event-processor

[2] https://davis.wpi.edu/datasets/Stock_Trace_Data/

[3] https://jmeter.apache.org/

[4] https://prometheus.io/

[5] https://grafana.com/

Figura 25 – Configurações do Docker

5.1 EXPERIMENTO 1: OPERADOR DE SEQUÊNCIA COM *DATASET* ORDENADO

Este experimento avalia o desempenho do processador com uma única DEC no catálogo (Código 3) e nada no Redis. A DEC usada identifica transações de venda da ação MSFT com preço superior a 26, seguidas de transações de compra da ação INTC com qualquer valor, em uma janela de 1 hora.

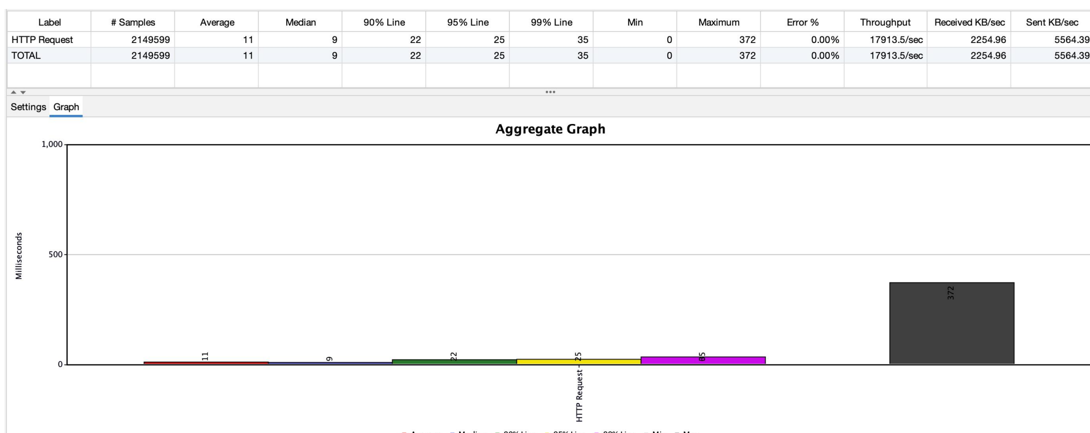
Código 3 – Consulta 1

```
{
  "id": "complex-event-1",
  "within": "1h",
  "logical_expression": {
    "connector": "sequence",
    "operands": [
      {
        "predicate": {
          "id": "A",
          "event_type": "S",
          "conditions": [
            {
              "field": "stock",
              "operator": "equal",
```

```
15                 "value": "MSFT"
16               },
17               {
18                 "field": "price",
19                 "operator": "greater_than",
20                 "value": 26
21               }
22             ]
23           }
24         },
25         {
26           "predicate": {
27             "id": "B",
28             "event_type": "C",
29             "conditions": [
30               {
31                 "field": "stock",
32                 "operator": "equal",
33                 "value": "INTC"
34               }
35             ]
36           }
37         }
38       ]
39     }
40   }
```

Na Figura 26 podemos ver as métricas agregadas do JMeter. Foram feitas 2.149.599 requisições durante os 120 segundos de teste. O tempo de resposta médio foi de 11 milisegundos, e o *throughput* médio foi de 17.913 requisições por segundo.

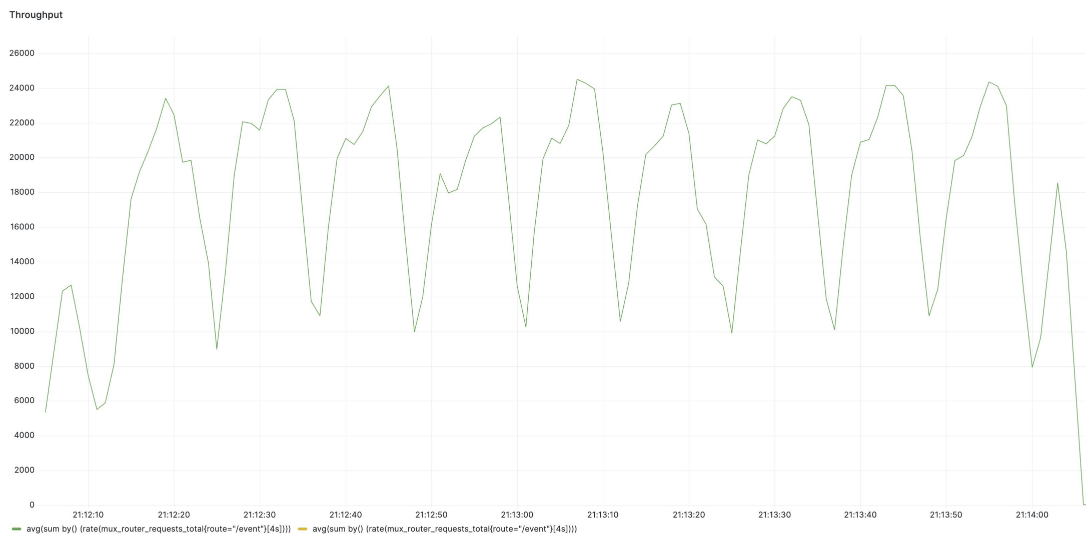
Figura 26 – Métricas agregadas do experimento 1



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/sec	Sent KB/sec
HTTP Request	2149599	11	9	22	25	35	0	372	0.00%	17913.5/sec	2254.96	5564.39
TOTAL	2149599	11	9	22	25	35	0	372	0.00%	17913.5/sec	2254.96	5564.39

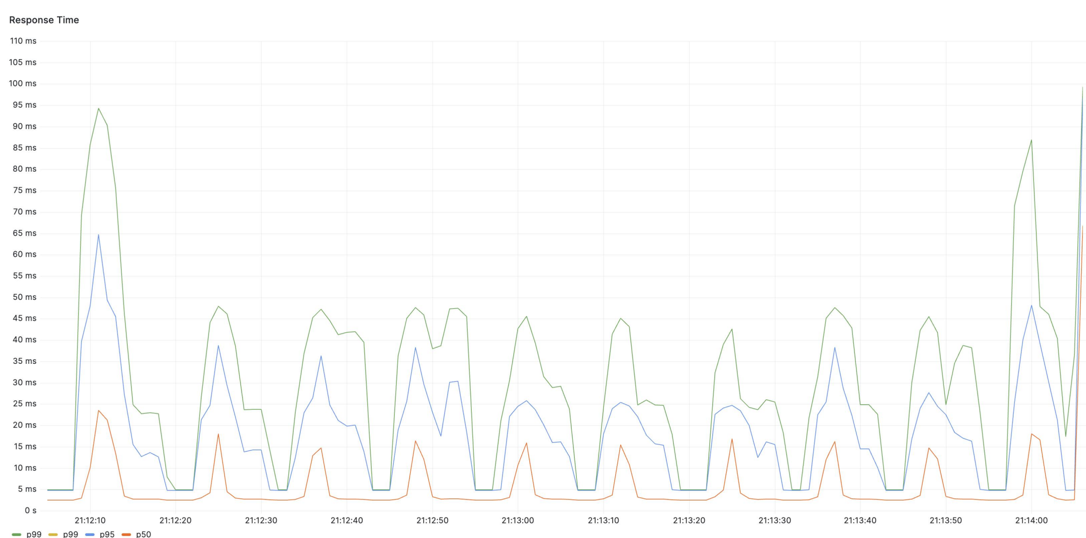
Figura 27 e Figura 28 mostram, respectivamente, o *throughput* e os tempos de resposta em função do tempo. Os picos de *throughput* são explicados pelo processo de filtro descrito no capítulo anterior. O JMeter percorre o *dataset* em ordem. Logo, nos momentos em que estão sendo processados eventos irrelevantes, o *throughput* é maior, pois os eventos são descartados já na primeira etapa. É possível ver pelos gráficos que o *dataset* foi percorrido aproximadamente dez vezes.

Figura 27 – Throughput do experimento 1



Na Figura 28, os percentis 99, 95 e 50 são representados, respectivamente, pelas linhas verde, azul e laranja.

Figura 28 – Tempos de resposta do experimento 1



A Figura 29 mostra o consumo de memória durante o experimento. O consumo se manteve constante (entre 23 e 25MB) durante todo o experimento. Percebe-se uma relação inversa entre os picos de *throughput* e de consumo de memória. Isso é explicado pelo fato de que um evento descartado não precisa passar pelas etapas de persistência e teste, consumindo muito menos

memória. A Figura 30 mostra quanta memória o processador consome quando ocioso. Desta forma, se descontarmos a memória usada pelas funções básicas do processo, a memória usada pelo experimento foi de aproximadamente 5MB.

Figura 29 – Consumo de memória do experimento 1

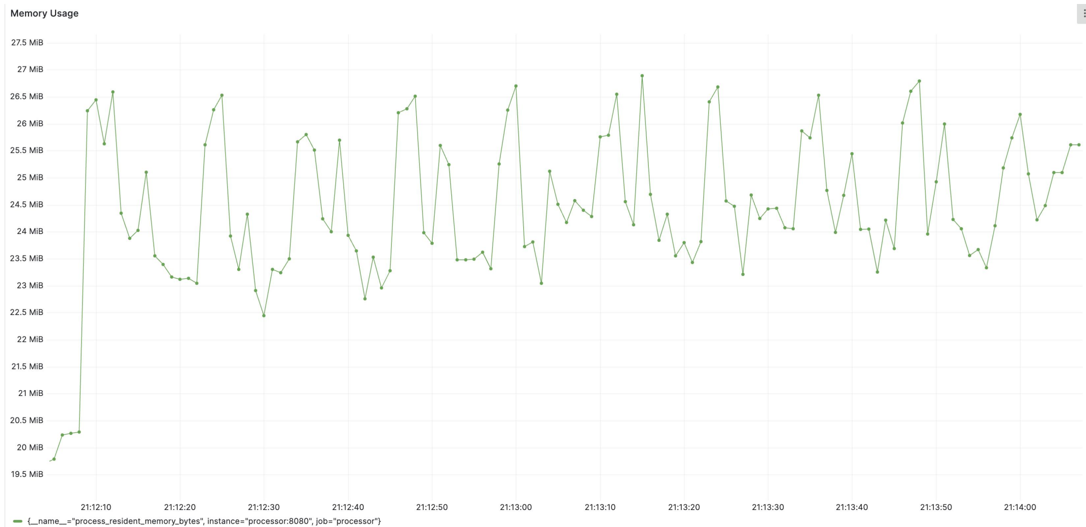
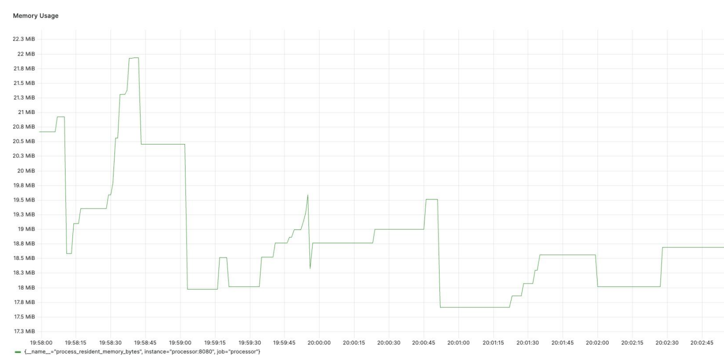


Figura 30 – *Footprint* base de memória



A Figura 31 mostra a quantidade de estados intermediários criados pelo experimento. Os predicados A e B (ver Código 3) foram impactados 78.480 e 310.484 vezes, respectivamente. Esses estados intermediários pesam cerca de 37,64MB no disco, como pode ser visto na Figura 32. Esse número é três vezes menor que o peso dos eventos processados.

Figura 31 – Quantidade de estados intermediários do experimento 1

```
127.0.0.1:6379> zcard A
(integer) 78480
127.0.0.1:6379> zcard B
(integer) 310484
127.0.0.1:6379>
```

Figura 32 – Armazenamento usado pelo Redis no experimento 1

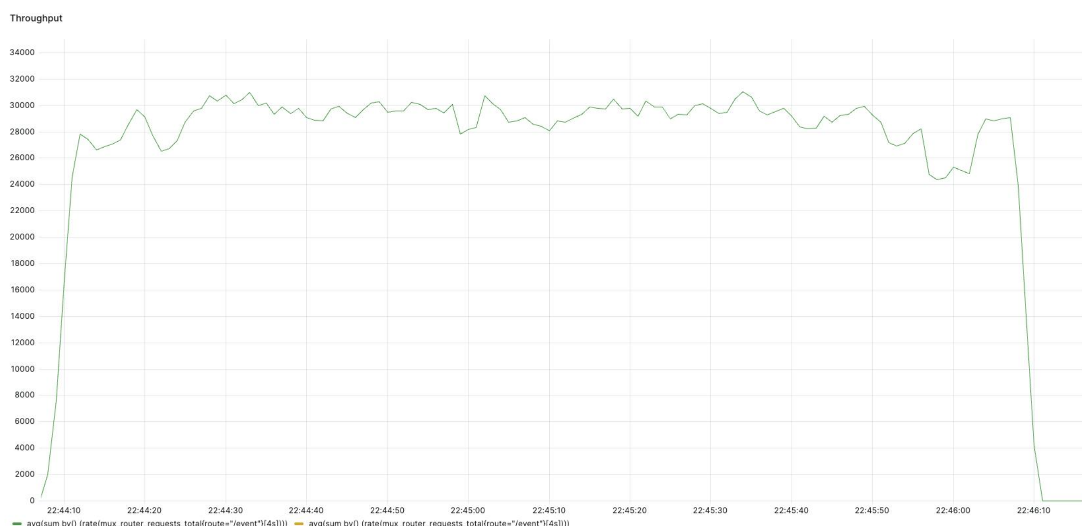
```
127.0.0.1:6379> info memory
# Memory
used_memory:39470712
used_memory_human:37.64M
used_memory_rss:47558656
used_memory_rss_human:45.36M
used_memory_peak:42146920
used_memory_peak_human:40.19M
```

5.2 EXPERIMENTO 2: DATASET DESORDENADO

Este experimento usa a mesma DEC do experimento anterior, mas dessa vez, o *dataset* é percorrido fora de ordem. Figura 33, Figura 34 e Figura 35 mostram o *throughput*, os tempos de resposta e o consumo de memória, respectivamente. Diferentemente do que aconteceu no primeiro experimento, o *throughput*, o tempo de resposta e o consumo de memória se mantiveram praticamente constantes durante todo o experimento. Isso aconteceu porque os eventos relevantes foram distribuídos aleatoriamente no *dataset*. Portanto, não houve um momento em que todos os eventos foram descartados ou selecionados. Como visto anteriormente, nos momentos em que todos os eventos são descartados o *throughput* é maior.

É possível observar pelo *throughput* e pelo tempo de resposta que o processador não escala em função da quantidade de eventos previamente processados. Essa característica contrasta com os processadores baseados em NFA e árvore, que precisam percorrer todos os estados intermediários criados até o momento para cada evento processado.

Figura 33 – Throughput do experimento 2



Na Figura 34, os percentis 99, 95 e 50 são representados, respectivamente, pelas linhas verde, azul e laranja.

Figura 34 – Tempos de resposta do experimento 2

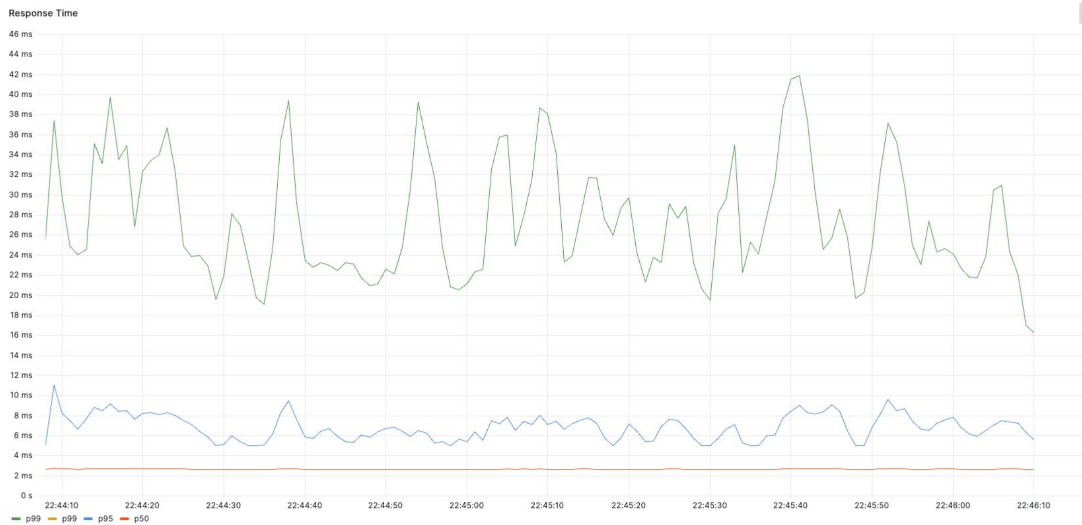
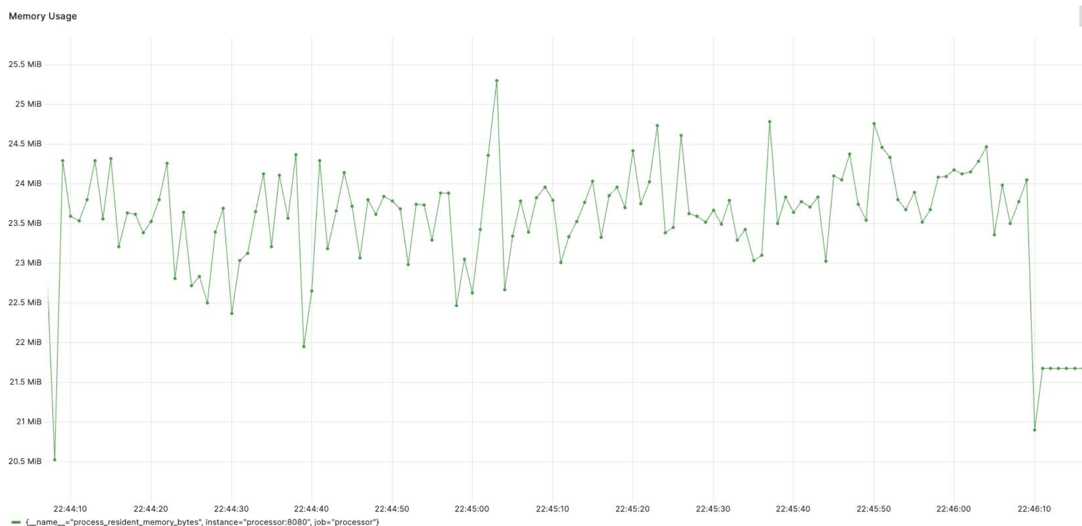


Figura 35 – Consumo de memória do experimento 2



5.3 EXPERIMENTO 3: DEC COM MAIS PREDICADOS

Este experimento mostra como o desempenho do processador varia em função da quantidade de predicados em uma DEC. Um exemplo é mostrado no Código 4. A DEC usada identifica a coocorrência de vendas das ações AMAT, CSCO, DELL, INTC IPIX, MSFT, ORCL, QQQ, RIMM e YAHOO, todas com um valor acima de 10.

Código 4 – Consulta 2

```
{
  "id": "complex-event-2",
  "logical_expression": {
    "connector": "and",
    "operands": [
```

```
6         {
7           "predicate": {
8             "id": "C",
9             "event_type": "S",
10            "conditions": [
11              {
12                "field": "stock",
13                "operator": "equal",
14                "value": "AMAT"
15              },
16              {
17                "field": "price",
18                "operator": "greater_than",
19                "value": 10
20              }
21            ]
22          }
23        },
24        {
25          "predicate": {
26            "id": "D",
27            "event_type": "S",
28            "conditions": [
29              {
30                "field": "stock",
31                "operator": "equal",
32                "value": "CSCO"
33              },
34              {
35                "field": "price",
36                "operator": "greater_than",
37                "value": 10
38              }
39            ]
40          }
41        },
42        {
43          "predicate": {
44            "id": "F",
45            "event_type": "S",
46            "conditions": [
47              {
```

```
48                  "field": "stock",
49                  "operator": "equal",
50                  "value": "DELL"
51                },
52                {
53                  "field": "price",
54                  "operator": "greater_than",
55                  "value": 10
56                }
57              ]
58            }
59          },
60          {
61            "predicate": {
62              "id": "G",
63              "event_type": "S",
64              "conditions": [
65                {
66                  "field": "stock",
67                  "operator": "equal",
68                  "value": "INTC"
69                },
70                {
71                  "field": "price",
72                  "operator": "greater_than",
73                  "value": 10
74                }
75              ]
76            }
77          },
78          {
79            "predicate": {
80              "id": "H",
81              "event_type": "S",
82              "conditions": [
83                {
84                  "field": "stock",
85                  "operator": "equal",
86                  "value": "IPIX"
87                },
88                {
89                  "field": "price",
```

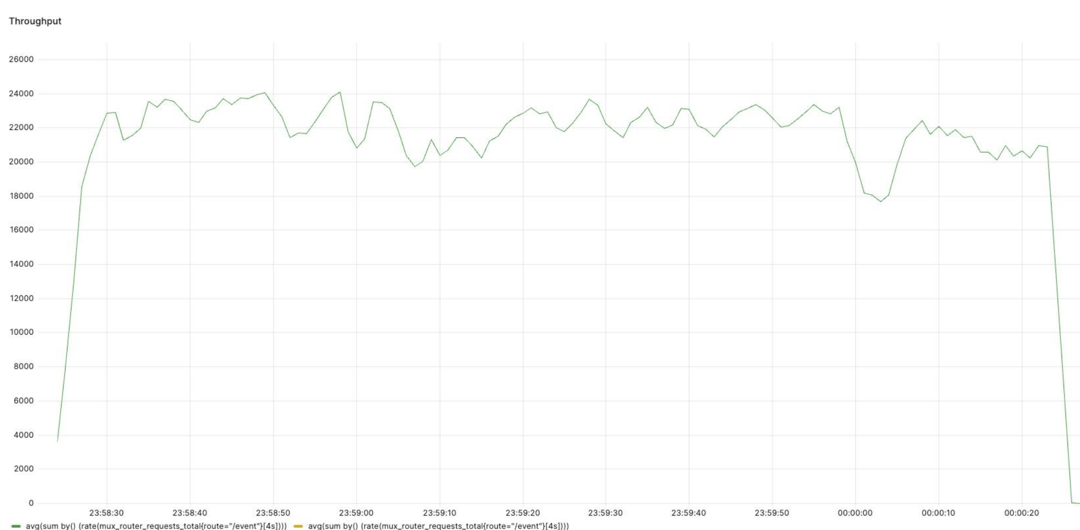
```
90              "operator": "greater_than",
91              "value": 10
92            }
93          ]
94        }
95    },
96    {
97      "predicate": {
98        "id": "I",
99        "event_type": "S",
100       "conditions": [
101         {
102           "field": "stock",
103           "operator": "equal",
104           "value": "MSFT"
105         },
106         {
107           "field": "price",
108           "operator": "greater_than",
109           "value": 10
110         }
111       ]
112     }
113   },
114   {
115     "predicate": {
116       "id": "J",
117       "event_type": "S",
118       "conditions": [
119         {
120           "field": "stock",
121           "operator": "equal",
122           "value": "ORCL"
123         },
124         {
125           "field": "price",
126           "operator": "greater_than",
127           "value": 10
128         }
129       ]
130     }
131   },
```

```
132        {
133          "predicate": {
134            "id": "K",
135            "event_type": "S",
136            "conditions": [
137              {
138                "field": "stock",
139                "operator": "equal",
140                "value": "QQQ"
141              },
142              {
143                "field": "price",
144                "operator": "greater_than",
145                "value": 10
146              }
147            ]
148          }
149        },
150        {
151          "predicate": {
152            "id": "L",
153            "event_type": "S",
154            "conditions": [
155              {
156                "field": "stock",
157                "operator": "equal",
158                "value": "RIMM"
159              },
160              {
161                "field": "price",
162                "operator": "greater_than",
163                "value": 10
164              }
165            ]
166          }
167        },
168        {
169          "predicate": {
170            "id": "M",
171            "event_type": "S",
172            "conditions": [
173              {
```

```
174                 "field": "stock",
175                 "operator": "equal",
176                 "value": "YAHOO"
177               },
178               {
179                 "field": "price",
180                 "operator": "greater_than",
181                 "value": 10
182               }
183             ]
184           }
185         }
186       ]
187     }
188   }
```

Na Figura 36 pode-se ver que o *throughput* se manteve em um nível aceitável, na casa dos 20.000 eventos por segundo. A Figura 37 demonstra pouca variação na mediana do tempo de resposta, mas os percentis 95 e 99 aumentaram. Este efeito se explica pela baixa seletividade dos predicados, que agora são sempre impactados, uma vez que há um predicado para cada código de ação disponível no *dataset*. Apesar disso, o tempo de resposta se manteve bastante baixo, com mediana em 5 milissegundos. Por fim, a Figura 38 mostra uma variação mínima no consumo de memória.

Figura 36 – Throughput do experimento 3



Na Figura 37, os percentis 99, 95 e 50 são representados, respectivamente, pelas linhas verde, azul e laranja.

Figura 37 – Tempos de resposta do experimento 3

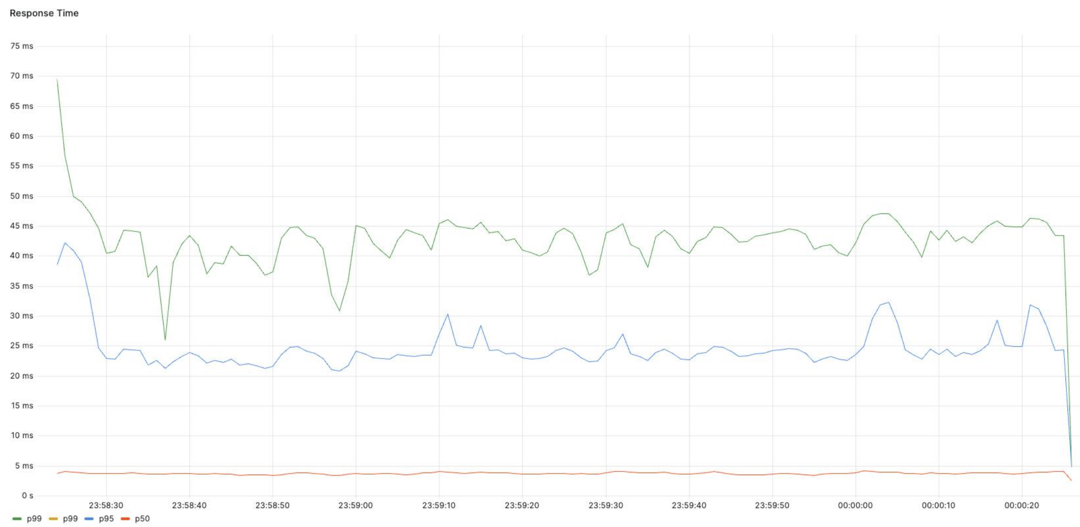
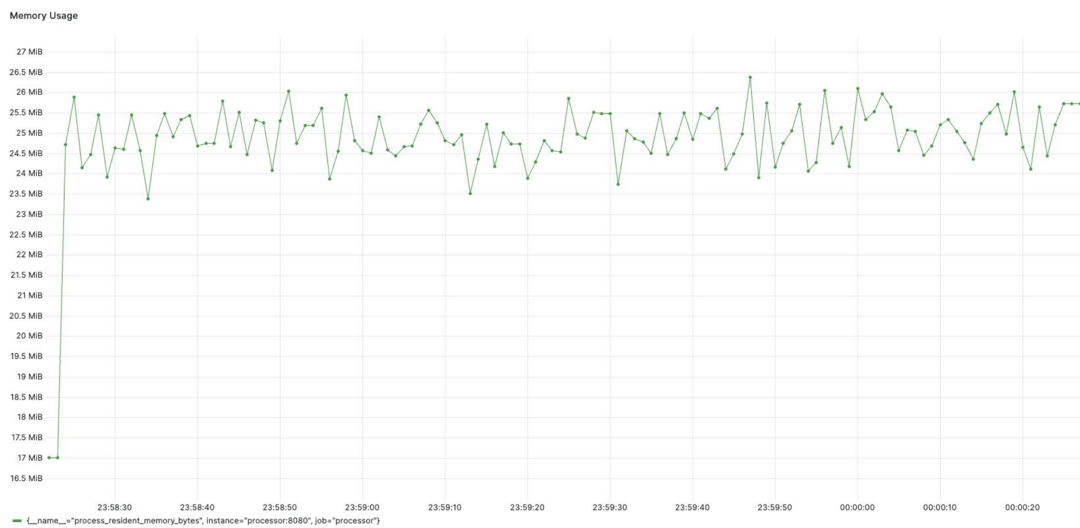


Figura 38 – Consumo de memória do experimento 3



Um efeito colateral negativo da baixa seletividade dos predicados pode ser observado na Figura 39 e na Figura 40. O número de estados intermediários cresceu bastante, assim como o armazenamento usado pelo Redis.

Figura 39 – Quantidade de estados intermediários do experimento 3

```
127.0.0.1:6379> zcard K
(integer) 154121
127.0.0.1:6379> zcard J
(integer) 96765
127.0.0.1:6379> zcard D
(integer) 114693
127.0.0.1:6379> zcard L
(integer) 92935
127.0.0.1:6379> zcard G
(integer) 146326
127.0.0.1:6379> zcard I
(integer) 136256
127.0.0.1:6379> zcard C
(integer) 95387
127.0.0.1:6379> zcard H
(integer) 205272
127.0.0.1:6379> zcard F
(integer) 82472
127.0.0.1:6379>
```

Figura 40 – Armazenamento usado pelo Redis no experimento 3

```
127.0.0.1:6379> info memory
# Memory
used_memory:106191800
used_memory_human:101.27M
used_memory_rss:113856512
used_memory_rss_human:108.58M
used_memory_peak:107795728
```

Avalia-se isso ao fato de que, como todos os eventos são impactados por exatamente um predicado, o número total de estados intermediários é igual ao número de eventos processados. Entretanto, também percebe-se uma oportunidade de otimização, pois como não há janela de tempo ou operador de sequência, o valor armazenado poderia ser uma simples caractere, ao invés de um *sortedset*.

6 CONCLUSÃO

Este trabalho de conclusão de curso teve como objetivo principal o desenvolvimento de um processador de eventos complexos eficiente, flexível e que escale bem com o número de eventos processados. Para isso, foi proposto um método diferente de processamento que minimiza o impacto de estados intermediários na escalabilidade de sistemas CEP.

A principal motivação para a realização desse trabalho são as evidentes oportunidades de melhoria em sistemas de CEP, principalmente no que diz respeito a escalabilidade. Ainda que existam trabalhos que abordam o tema, a maioria deles enfatiza a redução da criação de estados intermediários e não a minimização do impacto deles na escalabilidade do sistema.

As principais contribuições desse trabalho são:

- Uma nova representação de estado intermediário que os dissocia dos eventos complexos e permite armazená-los em sistemas de bancos de dados;
- Um modelo de dados para a escrita e leitura eficiente de estados intermediários;
- O projeto de um processador de CEP.
- Uma versão experimental desse processador escrita em Go[1].

Os experimentos realizados no Capítulo 5 demonstram a capacidade do método proposto de processar um grande volume de eventos em tempo constante, e com baixo consumo de memória. Dessa forma, o objetivo geral do trabalho pode ser considerado atingido.

Como atividades futuras relacionadas a este trabalho considera-se:

- Adicionar suporte a mais operadores, como o *fecho de Kleene*. Suportar mais operadores ampliaria a expressividade da linguagem de consulta e permitiria suportar uma variedade maior de eventos complexos.
- Adicionar diferentes estratégias de janelas de tempo, como janelas fixas. Atualmente só é suportada a janela deslizante (últimas x horas, por exemplo), mas a janela fixa é uma estratégia comum em sistemas de CEP.
- Resolver o problema da repetição de eventos. Atualmente, uma expressão como $A;B;C$ testará positivo pela primeira vez que A, B e C ocorrerem em sequência, mas testará positivo para cada A, B ou C que ocorrerem depois disso. Idealmente, a expressão deveria testar positivo apenas para novas triplas A, B, C.
- Eliminar estados intermediários que não são mais relevantes para DECs. Por exemplo, se só há uma DEC cuja janela de tempo é de uma hora, é possível que os estados intermediários do dia anterior não sejam mais relevantes.

[1] https://go.dev/

- Comparar o desempenho desta proposta com outros processadores de CEP disponíveis.

REFERÊNCIAS

AA, Han van der; ARTIKIS, Alexander; WEIDLICH, Matthias. Complex event processing methods for process querying. *In:* PROCESS Querying Methods. [*S.l.*]: Springer, 2022. P. 479–510. Citado nas pp. 18–22.

AWS. **Getting Started with Event-Driven Architecture**. [*S.l.*: *s.n.*], 2022. Website. Disponível em: <https://aws.amazon.com/blogs/compute/getting-started-with-event-driven-architecture/>. Acesso em: 21 set. 2022. Citado na p. 13.

AWS. **What is an Event-Driven Architecture?** [*S.l.*: *s.n.*], 2022. Website. Disponível em: <https://aws.amazon.com/event-driven-architecture/>. Acesso em: 21 set. 2022. Citado nas pp. 16, 17.

BUCCHI, Marco *et al.* CORE: a COmplex event Recognition Engine. **arXiv preprint arXiv:2111.04635**, 2021. Citado nas pp. 13, 14, 22.

CHAPNIK, Koral; KOLCHINSKY, Ilya; SCHUSTER, Assaf. DARLING: data-aware load shedding in complex event processing systems. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 15, n. 3, p. 541–554, 2021. Citado nas pp. 13, 14.

CONFLUENT. **What is CEP? Complex Event Processing Guide**. [*S.l.*: *s.n.*], 2022. Website. Disponível em: <https://www.confluent.io/learn/complex-event-processing/>. Acesso em: 21 set. 2022. Citado nas pp. 18, 19.

DAYARATHNA, Miyuru; PERERA, Srinath. Recent advancements in event processing. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 51, n. 2, p. 1–36, 2018. Citado na p. 19.

HIRZEL, Martin *et al.* A catalog of stream processing optimizations. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 46, n. 4, p. 1–34, 2014. Citado nas pp. 23, 25.

KOLCHINSKY, Ilya; SCHUSTER, Assaf. Real-time multi-pattern detection over event streams. *In:* PROCEEDINGS of the 2019 International Conference on Management of Data. [*S.l.*: *s.n.*], 2019. P. 589–606. Citado nas pp. 14, 18, 21, 23–27.

MEI, Yuan; MADDEN, Samuel. Zstream: a cost-based query processor for adaptively detecting composite events. *In:* PROCEEDINGS of the 2009 ACM SIGMOD International Conference on Management of data. [*S.l.*: *s.n.*], 2009. P. 193–206. Citado nas pp. 14, 22, 25–27.

MICHELSON, Brenda M. Event-driven architecture overview. **Patricia Seybold Group**, v. 2, n. 12, p. 10–1571, 2006. Citado na p. 16.

SLO, Ahmad *et al.* pSPICE: partial match shedding for complex event processing. *In:* IEEE. 2019 IEEE International Conference on Big Data (Big Data). [*S.l.*: *s.n.*], 2019. P. 372–382. Citado nas pp. 14, 25–27.

STACKOVERFLOW. **Can I read bigtable records with cbt by time?** [*S.l.*: *s.n.*], 2021. [Online; acessado em 12 de junho de 2023]. Disponível em: <https://stackoverflow.com/questions/70357169/can-i-read-bigtable-records-with-cbt-by-time>. Citado na p. 39.

Apêndices

APÊNDICE A – CÓDIGO-FONTE

```go
// go.mod
module github.com/jeangnc/complex-event-processor

go 1.18

require (
	github.com/gorilla/mux v1.8.0
	github.com/prometheus/client_golang v1.8.0
	github.com/redis/go-redis/v9 v9.0.5
	gitlab.com/msvechla/mux-prometheus v0.0.2
	golang.org/x/exp v0.0.0-20220414153411-bcd21879b8fd
)

require (
	github.com/beorn7/perks v1.0.1 // indirect
	github.com/cespare/xxhash/v2 v2.2.0 // indirect
	github.com/dgryski/go-rendezvous v0.0.0-20200823014737-9f7001d12a5f // indirect
	github.com/golang/protobuf v1.4.3 // indirect
	github.com/matttproud/golang_protobuf_extensions v1.0.1 // indirect
	github.com/prometheus/client_model v0.2.0 // indirect
	github.com/prometheus/common v0.14.0 // indirect
	github.com/prometheus/procfs v0.2.0 // indirect
	golang.org/x/sys v0.0.0-20211019181941-9d821ace8654 // indirect
	google.golang.org/protobuf v1.23.0 // indirect
)

// pkg/tree/tree.go
package tree

type Node struct {
	value ValueMap
	nodes NodeMap
}
type ValueMap map[string]interface{}
type NodeMap map[string]Node

func NewNode() Node {
```

```
38      return Node{
39          value: ValueMap{},
40          nodes: NodeMap{},
41      }
42  }
43
44  func (n *Node) Set(key string, value interface{}) {
45      n.value[key] = value
46  }
47
48  func (n *Node) Get(key string) interface{} {
49      return n.value[key]
50  }
51
52  func (n *Node) Unset(key string) {
53      delete(n.value, key)
54  }
55
56  func (n *Node) Traverse(path []string) *Node {
57      target := n
58
59      for _, k := range path {
60          next, ok := target.nodes[k]
61
62          if !ok {
63              newNode := NewNode()
64              target.nodes[k] = newNode
65              next = newNode
66          }
67
68          target = &next
69      }
70
71      return target
72  }
73
74  // traverses different arrangements of keys retrieving values along
        the way
75  func (n Node) Values(path []string) []interface{} {
76      var found []interface{}
77
78      if n.value != nil {
```

```
79          for _, v := range n.value {
80              found = append(found, v)
81          }
82      }
83
84      for i, k := range path {
85          c, ok := n.nodes[k]
86
87          if ok {
88              found = append(found, c.Values(path[i:])...)
89          }
90      }
91
92      return found
93  }
94
95  // pkg/types/types.go
96  package types
97
98  const CONNECTOR_AND string = "and"
99  const CONNECTOR_OR string = "or"
100 const CONNECTOR_SEQUENCE string = "sequence"
101
102 type Event struct {
103     Id        string                 `json:"id"`
104     Type      string                 `json:"type"`
105     Timestamp int64                  `json:"timestamp"`
106     Payload   map[string]interface{} `json:"payload"`
107 }
108
109 type State struct {
110     Predicates map[string]bool
111 }
112
113 type Impact struct {
114     Predicates map[string]bool
115 }
116
117 type Expression struct {
118     Id                string            `json:"id"`
119     LogicalExpression LogicalExpression `json:"logical_expression"`
120     Within            string            `json:"within"`
```

```
121 }
122
123 type Operand struct {
124     Negated           bool
125     Predicate         *Predicate         `json:"predicate,omitempty"`
126     LogicalExpression *LogicalExpression
        `json:"logical_expression,omitempty"`
127 }
128
129 type Predicate struct {
130     Id         string      `json:"id"`
131     EventType  string      `json:"event_type"`
132     Conditions []Condition `json:"conditions"`
133 }
134
135 type Condition struct {
136     Field    string      `json:"field"`
137     Operator string      `json:"operator"`
138     Value    interface{} `json:"value"`
139 }
140
141 // pkg/types/logical_expression.go
142 package types
143
144 type LogicalExpression struct {
145     Connector string    `json:"connector"`
146     Operands  []Operand `json:"operands"`
147 }
148
149 func (le LogicalExpression) DeepPredicates() []*Predicate {
150     r := make([]*Predicate, 0)
151
152     for _, o := range le.Operands {
153         if o.Predicate != nil {
154             r = append(r, o.Predicate)
155             continue
156         }
157
158         r = append(r, o.LogicalExpression.DeepPredicates()...)
159     }
160
161     return r
```

```
162 }
163
164 // pkg/types/value/float_value.go
165 package value
166
167 import "fmt"
168
169 type FloatValue struct {
170     value float64
171 }
172
173 func (v FloatValue) ToInt() int64 {
174     return int64(v.value)
175 }
176
177 func (v FloatValue) ToFloat() float64 {
178     return v.value
179 }
180
181 func (v FloatValue) ToString() string {
182     return fmt.Sprintf("%v", v.value)
183 }
184
185 func (v FloatValue) LessThan(v2 GenericValue) bool {
186     return v.value < v2.ToFloat()
187 }
188
189 func (v FloatValue) LessThanEqual(v2 GenericValue) bool {
190     return v.value <= v2.ToFloat()
191 }
192
193 func (v FloatValue) GreaterThan(v2 GenericValue) bool {
194     return v.value > v2.ToFloat()
195 }
196
197 func (v FloatValue) GreaterThanEqual(v2 GenericValue) bool {
198     return v.value >= v2.ToFloat()
199 }
200
201 func (v FloatValue) Equal(v2 GenericValue) bool {
202     return v.value == v2.ToFloat()
203 }
```

```
204
205 func (v FloatValue) Different(v2 GenericValue) bool {
206     return v.value != v2.ToFloat()
207 }
208
209 // pkg/types/value/int_value.go
210 package value
211
212 import "fmt"
213
214 type IntValue struct {
215     value int64
216 }
217
218 func (v IntValue) ToInt() int64 {
219     return v.value
220 }
221
222 func (v IntValue) ToFloat() float64 {
223     return float64(v.value)
224 }
225
226 func (v IntValue) ToString() string {
227     return fmt.Sprintf("%v", v.value)
228 }
229
230 func (v IntValue) LessThan(v2 GenericValue) bool {
231     return v.value < v2.ToInt()
232 }
233
234 func (v IntValue) LessThanEqual(v2 GenericValue) bool {
235     return v.value <= v2.ToInt()
236 }
237
238 func (v IntValue) GreaterThan(v2 GenericValue) bool {
239     return v.value > v2.ToInt()
240 }
241
242 func (v IntValue) GreaterThanEqual(v2 GenericValue) bool {
243     return v.value >= v2.ToInt()
244 }
245
```

```
246 func (v IntValue) Equal(v2 GenericValue) bool {
247     return v.value == v2.ToInt()
248 }
249
250 func (v IntValue) Different(v2 GenericValue) bool {
251     return v.value != v2.ToInt()
252 }
253
254 // pkg/types/value/string_value.go
255 package value
256
257 import (
258     "fmt"
259     "strconv"
260 )
261
262 type StringValue struct {
263     value string
264 }
265
266 func (v StringValue) ToInt() int64 {
267     i, err := strconv.ParseInt(v.value, 10, 64)
268
269     if err != nil {
270         fmt.Println(err)
271     }
272
273     return i
274 }
275
276 func (v StringValue) ToFloat() float64 {
277     f, err := strconv.ParseFloat(v.value, 64)
278
279     if err != nil {
280         fmt.Println(err)
281     }
282
283     return f
284 }
285
286 func (v StringValue) ToString() string {
287     return v.value
```

```
288 }
289 
290 func (v StringValue) LessThan(v2 GenericValue) bool {
291     return v.value < v2.ToString()
292 }
293 
294 func (v StringValue) LessThanEqual(v2 GenericValue) bool {
295     return v.value <= v2.ToString()
296 }
297 
298 func (v StringValue) GreaterThan(v2 GenericValue) bool {
299     return v.value > v2.ToString()
300 }
301 
302 func (v StringValue) GreaterThanEqual(v2 GenericValue) bool {
303     return v.value >= v2.ToString()
304 }
305 
306 func (v StringValue) Equal(v2 GenericValue) bool {
307     return v.value == v2.ToString()
308 }
309 
310 func (v StringValue) Different(v2 GenericValue) bool {
311     return v.value != v2.ToString()
312 }
313 
314 // value.go
315 package value
316 
317 import "fmt"
318 
319 func NewValue(v interface{}) GenericValue {
320     switch v := v.(type) {
321     case float64:
322         return FloatValue{value: v}
323     case int64:
324         return IntValue{value: v}
325     case string:
326         return StringValue{value: v}
327     default:
328         fmt.Println(fmt.Sprintf("I don't know about type %T!\n", v))
329         return nil
```

```
330        }
331    }
332
333    type GenericValue interface {
334        ToInt() int64
335        ToFloat() float64
336        ToString() string
337        LessThan(GenericValue) bool
338        LessThanEqual(GenericValue) bool
339        GreaterThan(GenericValue) bool
340        GreaterThanEqual(GenericValue) bool
341        Equal(GenericValue) bool
342        Different(GenericValue) bool
343    }
344
345    // pkg/expression/index.go
346    package expression
347
348    import (
349        "encoding/gob"
350        "fmt"
351        "os"
352        "sort"
353        "sync"
354
355        "github.com/jeangnc/complex-event-processor/pkg/tree"
356        "github.com/jeangnc/complex-event-processor/pkg/types"
357        "github.com/jeangnc/complex-event-processor/pkg/types/value"
358
359        "golang.org/x/exp/maps"
360    )
361
362    const OPERATOR_EQUAL string = "equal"
363    const OPERATOR_DIFFERENT string = "not_equal"
364    const OPERATOR_LESS_THAN string = "less_than"
365    const OPERATOR_GREATER_THAN string = "greater_than"
366    const OPERATOR_LESS_THAN_OR_EQUAL = "less_than_or_equal"
367    const OPERATOR_GREATER_THAN_OR_EQUAL = "greater_than_or_equal"
368
369    type Index struct {
370        filename            string
371        expressions         map[string]*types.Expression
```

```
372     predicateExpressions map[string][]*types.Expression
373     predicateTree        tree.Node
374     mutex                sync.Mutex
375 }
376
377 func NewIndex(filename string) Index {
378     return Index{
379         filename:             filename,
380         expressions:          map[string]*types.Expression{},
381         predicateExpressions: map[string][]*types.Expression{},
382         predicateTree:        tree.NewNode(),
383     }
384 }
385
386 func NewTemporaryIndex() Index {
387     return NewIndex("")
388 }
389
390 func (i *Index) Load() {
391     f, err := os.Open(i.filename)
392
393     if err != nil {
394         fmt.Println(err)
395         return
396     }
397
398     var es []types.Expression
399
400     d := gob.NewDecoder(f)
401     err = d.Decode(&es)
402     if err != nil {
403         fmt.Println("Decode error:", err)
404     }
405
406     for _, e := range es {
407         i.Append(e)
408     }
409 }
410
411 func (i Index) Save() {
412     e := i.Expressions()
413     f, _ := os.Create(i.filename)
```

```
414     c := gob.NewEncoder(f)
415     c.Encode(e)
416 }
417
418 func (i Index) SearchImpactedPredicates(e types.Event) types.Impact {
419     keys := extractPayloadKeys(e)
420     values := i.predicateTree.Values(keys)
421
422     result := map[string]bool{}
423
424     for _, v := range values {
425         p := v.(types.Predicate)
426         r := evaluateConditions(e, p)
427
428         if r {
429             result[p.Id] = r
430         }
431     }
432
433     return types.Impact{Predicates: result}
434 }
435
436 func (i Index) FilterImpactedExpressions(impact types.Impact)
       []*types.Expression {
437     r := make([]*types.Expression, 0, 0)
438
439     for p, _ := range impact.Predicates {
440         es, ok := i.predicateExpressions[p]
441
442         if !ok {
443             continue
444         }
445
446         for _, e := range es {
447             r = append(r, e)
448         }
449     }
450
451     return r
452 }
453
454 func (i *Index) Append(e types.Expression) {
```

```
455     i.mutex.Lock()
456     defer i.mutex.Unlock()
457
458     i.expressions[e.Id] = &e
459
460     for _, p := range e.LogicalExpression.DeepPredicates() {
461         keys := extractPredicateKeys(p)
462
463         n := i.predicateTree.Traverse(keys)
464         n.Set(p.Id, *p)
465
466         if _, ok := i.predicateExpressions[p.Id]; !ok {
467             i.predicateExpressions[p.Id] = make([]*types.Expression,
                    0)
468         }
469         i.predicateExpressions[p.Id] =
                append(i.predicateExpressions[p.Id], &e)
470     }
471 }
472
473 func (i Index) Expressions() []types.Expression {
474     r := make([]types.Expression, 0)
475
476     for _, e := range i.expressions {
477         r = append(r, *e)
478     }
479
480     return r
481 }
482
483 func extractPayloadKeys(e types.Event) []string {
484     fields := maps.Keys(e.Payload)
485     sort.Strings(fields)
486
487     keys := append([]string{e.Type}, fields...)
488
489     return keys
490 }
491
492 func extractPredicateKeys(p *types.Predicate) []string {
493     fields := make([]string, 0, len(p.Conditions))
494
```

```
495     for _, c := range p.Conditions {
496         fields = append(fields, c.Field)
497     }
498
499     sort.Strings(fields)
500
501     keys := []string{p.EventType}
502     keys = append(keys, fields...)
503
504     return keys
505 }
506
507 func evaluateConditions(e types.Event, p types.Predicate) bool {
508     result := true
509
510     for _, c := range p.Conditions {
511         payloadValue, ok := e.Payload[c.Field]
512
513         if !ok {
514             return false
515         }
516
517         genericValue := value.NewValue(payloadValue)
518         expectedValue := value.NewValue(c.Value)
519
520         switch c.Operator {
521         case OPERATOR_EQUAL:
522             result = result && genericValue.Equal(expectedValue)
523         case OPERATOR_DIFFERENT:
524             result = result && genericValue.Different(expectedValue)
525         case OPERATOR_LESS_THAN:
526             result = result && genericValue.LessThan(expectedValue)
527         case OPERATOR_GREATER_THAN:
528             result = result &&
                   genericValue.GreaterThan(expectedValue)
529         case OPERATOR_LESS_THAN_OR_EQUAL:
530             result = result &&
                   genericValue.LessThanEqual(expectedValue)
531         case OPERATOR_GREATER_THAN_OR_EQUAL:
532             result = result &&
                   genericValue.GreaterThanEqual(expectedValue)
533         default:
```

```
534             panic(fmt.Sprintf("invalid operator %s", c.Operator))
535         }
536     }
537 
538     return result
539 }
540 
541 // pkg/util/slice.go
542 package util
543 
544 func SliceAll(s []bool) bool {
545     r := true
546 
547     for _, b := range s {
548         r = r && b
549     }
550 
551     return r
552 }
553 
554 func SliceAny(s []bool) bool {
555     for _, b := range s {
556         if b {
557             return true
558         }
559     }
560 
561     return false
562 }
563 
564 // pkg/expression/evaluation.go
565 package expression
566 
567 import (
568     "fmt"
569 
570     "github.com/jeangnc/complex-event-processor/pkg/types"
571     util "github.com/jeangnc/complex-event-processor/pkg/util"
572 )
573 
574 func EvaluateExpression(e types.State, ex *types.Expression) bool {
575     return evaluateLogicalExpression(e, &ex.LogicalExpression)
```

```
576 }
577
578 func evaluateLogicalExpression(e types.State, l
        *types.LogicalExpression) bool {
579     values := make([]bool, 0, 0)
580     prefix := ""
581
582     for _, o := range l.Operands {
583         if o.LogicalExpression != nil {
584             values = append(values, evaluateLogicalExpression(e,
                    o.LogicalExpression))
585             continue
586         }
587
588         value, ok := e.Predicates[prefix+o.Predicate.Id]
589
590         if !ok {
591             value = false
592         }
593
594         if o.Negated {
595             value = !value
596         }
597
598         if l.Connector == types.CONNECTOR_SEQUENCE {
599             prefix += o.Predicate.Id + ";"
600         }
601
602         values = append(values, value)
603     }
604
605     result := false
606     switch l.Connector {
607     case types.CONNECTOR_AND, types.CONNECTOR_SEQUENCE:
608         result = util.SliceAll(values)
609     case types.CONNECTOR_OR:
610         result = util.SliceAny(values)
611     default:
612         panic(fmt.Sprintf("invalid connector %s", l.Connector))
613     }
614
615     return result
```

```
616 }
617
618 // pkg/handlers/event_handler.go
619 package handlers
620
621 import (
622     "encoding/json"
623     "fmt"
624     "log"
625     "net/http"
626
627     "github.com/jeangnc/complex-event-processor/pkg/event"
628     "github.com/jeangnc/complex-event-processor/pkg/expression"
629     "github.com/jeangnc/complex-event-processor/pkg/state"
630     "github.com/jeangnc/complex-event-processor/pkg/types"
631 )
632
633 func NewEventHandler(index *expression.Index, repository
       state.Repository) func(w http.ResponseWriter, r *http.Request) {
634     return func(w http.ResponseWriter, r *http.Request) {
635         var e types.Event
636
637         err := json.NewDecoder(r.Body).Decode(&e)
638         if err != nil {
639             w.WriteHeader(http.StatusBadRequest)
640             fmt.Fprintf(w, "Invalid event: ", err)
641             return
642         }
643
644         response := event.Process(index, repository, e)
645         log.Print("Response: ", response)
646
647         w.WriteHeader(http.StatusCreated)
648         json.NewEncoder(w).Encode(response)
649     }
650 }
651
652 // pkg/handlers/expression_handler.go
653 package handlers
654
655 import (
656     "encoding/json"
```

```
657     "fmt"
658     "net/http"
659
660     "github.com/jeangnc/complex-event-processor/pkg/expression"
661     "github.com/jeangnc/complex-event-processor/pkg/types"
662 )
663
664 func NewExpressionHandler(index *expression.Index) func(w
        http.ResponseWriter, r *http.Request) {
665     return func(w http.ResponseWriter, r *http.Request) {
666         var e types.Expression
667
668         err := json.NewDecoder(r.Body).Decode(&e)
669         if err != nil {
670             w.WriteHeader(http.StatusBadRequest)
671             fmt.Fprintf(w, "Invalid expression: ", err)
672             return
673         }
674
675         index.Append(e)
676         index.Save()
677
678         w.WriteHeader(http.StatusCreated)
679         json.NewEncoder(w).Encode(map[string]bool{"ok": true})
680     }
681 }
682
683 // pkg/state/redis_repository.go
684 package state
685
686 import (
687     "context"
688     "io/ioutil"
689     "strconv"
690     "strings"
691     "time"
692
693     "github.com/jeangnc/complex-event-processor/pkg/types"
694     "github.com/redis/go-redis/v9"
695 )
696
697 type RedisRepository struct {
```

```
698     client *redis.Client
699 }
700
701 func NewRedisRepository(redisAddr string, functionsFile string)
        RedisRepository {
702     client := redis.NewClient(&redis.Options{
703         Addr:     redisAddr,
704         Password: "",
705     })
706
707     loadLib(client, functionsFile)
708
709     return RedisRepository{
710         client: client,
711     }
712 }
713
714 func (r RedisRepository) Save(ctx context.Context, event
        types.Event, impact types.Impact) error {
715     _, err := r.client.Pipelined(ctx, func(pipe redis.Pipeliner)
            error {
716         for k := range impact.Predicates {
717             member := redis.Z{
718                 Score:  float64(event.Timestamp),
719                 Member: event.Id,
720             }
721
722             pipe.ZAdd(ctx, k, member)
723         }
724
725         return nil
726     })
727
728     return err
729 }
730
731 func (r RedisRepository) Load(ctx context.Context, event
        types.Event, expressions []*types.Expression)
        (map[*types.Expression]types.State, error) {
732     expressionCommands :=
            make(map[*types.Expression]map[string]*redis.Cmd)
733
```

```
734     _, err := r.client.Pipelined(ctx, func(pipe redis.Pipeliner)
          error {
735         for _, e := range expressions {
736             expressionCommands[e] = make(map[string]*redis.Cmd, 0)
737 
738             min := "-inf"
739             max := "+inf"
740 
741             if e.Within != "" {
742                 t := time.Unix(event.Timestamp, 0)
743                 duration, _ := time.ParseDuration(e.Within)
744                 min = strconv.FormatInt(t.Add(-duration).Unix(), 10)
745                 max = strconv.FormatInt(t.Add(duration).Unix(), 10)
746             }
747 
748             for _, keys := range
                  gambiKeysToLoad(e.LogicalExpression) {
749                 id := strings.Join(keys, ";")
750 
751                 var fname string
752 
753                 if len(keys) > 1 {
754                     fname = "zsequence"
755                 } else {
756                     fname = "zvalue"
757                 }
758 
759                 expressionCommands[e][id] = pipe.FCall(ctx, fname,
                      keys, min, max)
760             }
761         }
762 
763         return nil
764     })
765 
766     if err != nil {
767         panic(err)
768     }
769 
770     states := make(map[*types.Expression]types.State)
771 
772     for e, commands := range expressionCommands {
```

```
773         values := make(map[string]bool)
774
775         for id, command := range commands {
776             keys := strings.Split(id, ";")
777
778             if len(keys) > 1 {
779                 prefix := ""
780                 response, _ := command.Slice()
781
782                 for i, k := range keys {
783                     values[prefix+k] = response[i] != nil
784                     prefix += k + ";"
785                 }
786             } else {
787                 response, _ := command.Slice()
788                 values[id] = len(response) > 0
789             }
790         }
791
792         states[e] = types.State{Predicates: values}
793     }
794
795     return states, nil
796 }
797
798 func gambiKeysToLoad(l types.LogicalExpression) [][]string {
799     r := make([][]string, 0, 0)
800
801     if l.Connector == types.CONNECTOR_SEQUENCE {
802         keys := make([]string, 0, 0)
803
804         for _, o := range l.Operands {
805             keys = append(keys, o.Predicate.Id)
806         }
807
808         r = append(r, keys)
809         return r
810     }
811
812     for _, o := range l.Operands {
813         if o.LogicalExpression != nil {
814             r = append(r, gambiKeysToLoad(*o.LogicalExpression)...)
```

```
815             continue
816         }
817
818         r = append(r, []string{o.Predicate.Id})
819     }
820
821     return r
822 }
823
824 func loadLib(client *redis.Client, functionsFile string) {
825     ctx := context.Background()
826     content, err := ioutil.ReadFile(functionsFile)
827
828     if err != nil {
829         panic(err)
830     }
831
832     client.FunctionLoadReplace(ctx, string(content))
833 }
834
835 // pkg/state/repository.go
836 package state
837
838 import (
839     "context"
840
841     "github.com/jeangnc/complex-event-processor/pkg/types"
842 )
843
844 type Repository interface {
845     Save(ctx context.Context, event types.Event, impact
           types.Impact) error
846     Load(ctx context.Context, event types.Event, expressions
           []*types.Expression) (map[*types.Expression]types.State,
           error)
847 }
848
849 // main.go
850 package main
851
852 import (
853     "bytes"
```

```
854        "io/ioutil"
855        "log"
856        "net/http"
857        "time"
858
859        "github.com/gorilla/mux"
860        "github.com/prometheus/client_golang/prometheus/promhttp"
861
862        "github.com/jeangnc/complex-event-processor/pkg/expression"
863        "github.com/jeangnc/complex-event-processor/pkg/handlers"
864        "github.com/jeangnc/complex-event-processor/pkg/state"
865        muxprom "gitlab.com/msvechla/mux-prometheus/pkg/middleware"
866  )
867
868  const (
869        port = ":8080"
870  )
871
872  func jsonMiddleware(next http.Handler) http.Handler {
873        return http.HandlerFunc(func(w http.ResponseWriter, r
              *http.Request) {
874              w.Header().Add("Content-Type", "application/json")
875              next.ServeHTTP(w, r)
876        })
877  }
878
879  func loggingMiddleware(next http.Handler) http.Handler {
880        return http.HandlerFunc(func(w http.ResponseWriter, r
              *http.Request) {
881              bodyBytes, _ := ioutil.ReadAll(r.Body)
882              r.Body = ioutil.NopCloser(bytes.NewBuffer(bodyBytes))
883              log.Print("Received: ", string(bodyBytes))
884              next.ServeHTTP(w, r)
885        })
886  }
887
888  // lib.lua
889  #!lua name=lib
890  local function zsequence(keys, args)
891     local lower_bound, upper_bound = unpack(args)
892     local values = {}
893
```

```
894     for i, k in ipairs(keys) do
895        local value = redis.call('ZRANGE', k, lower_bound,
              upper_bound, 'BYSCORE', 'LIMIT', 0, 1, 'WITHSCORES')
896
897        if #value > 0 then
898           value = value[2]
899           lower_bound = value + 1
900        else
901           value = false
902        end
903
904        table.insert(values, value)
905     end
906
907     return values
908  end
909
910  local function zvalue(keys, args)
911     local lower_bound, upper_bound = unpack(args)
912     local value = redis.call('ZRANGE', keys[1], lower_bound,
           upper_bound, 'BYSCORE', 'LIMIT', 0, 1, 'WITHSCORES')
913
914     return {value[2]} or {}
915  end
916
917  redis.register_function('zvalue', zvalue)
918  redis.register_function('zsequence', zsequence)
919
920
921  func main() {
922      index := expression.NewIndex("./tmp/expressions")
923      index.Load()
924
925      repository := state.NewRedisRepository("redis:6379", "./lib.lua")
926
927      instrumentation := muxprom.NewDefaultInstrumentation()
928
929      router := mux.NewRouter().StrictSlash(true)
930      router.Use(jsonMiddleware)
931      router.Use(loggingMiddleware)
932      router.Use(instrumentation.Middleware)
933
```

```
934     router.HandleFunc("/event", handlers.NewEventHandler(&index,
           &repository)).Methods("POST")
935     router.HandleFunc("/expression",
           handlers.NewExpressionHandler(&index)).Methods("POST")
936     router.Path("/metrics").Handler(promhttp.Handler())
937
938     srv := &http.Server{
939         Handler:      router,
940         Addr:         port,
941         WriteTimeout: 15 * time.Second,
942         ReadTimeout:  15 * time.Second,
943     }
944
945     log.Print("Listening to ", port)
946     log.Fatal(srv.ListenAndServe())
947 }
```

APÊNDICE B – ARTIGO NO FORMATO SBC

Processador de Eventos Complexos de Alto Desempenho para Fluxos Massivos de Eventos

Jean Carlos Gonçalves[1], **Ronaldo dos Santos Mello**[2]

[1]Departamento de Informatica e Estatatística (INE)
Universidade Federal de Santa Catarina (UFSC)
Florianopolis – SC – Brasil

jean.g.c@grad.ufsc.br, r.mello@ufsc.br

Abstract. *Complex Event Processing (CEP) is a type of event-driven architecture that aims to detect situations of interest in the form of complex events. For such systems, latency is a significantly important metric. However, parsing/-processing CEP queries is known to be computationally challenging. Several optimization techniques have been proposed to minimize the latency of these systems. However, an analysis of recent works in the CEP field shows that, despite being effective, the proposed optimization techniques still have limitations and opportunities for improvement. In this context, the development of a CEP processor is proposed, with the differential of having constant or almost constant response times and throughput, regardless of the number of events previously processed.*

Resumo. *O Complex Event Processing (CEP) é um tipo de arquitetura orientada a eventos que visa detectar situações de interesse na forma de eventos complexos. Para tais sistemas, a latência é uma métrica significativamente importante. Entretanto, a análise/processamento de consultas CEP é conhecida por ser computacionalmente desafiadora. Diversas técnicas de otimização foram propostas para minimizar a latência desses sistemas. Porém, uma análise dos trabalhos recentes no campo de CEP evidencia que, apesar de efetivas, as técnicas de otimização propostas ainda possuem limitações e oportunidades de melhoria. Neste contexto é proposto o desenvolvimento de um processador de CEP, com o diferencial de possuir tempos de resposta e vazão constantes ou quase constantes, independente da quantidade de eventos processada anteriormente.*

1. Introdução

A arquitetura orientada a eventos tem se tornado cada vez mais proeminente no desenvolvimento de aplicações modernas, principalmente em aplicações construídas como microsserviços, devido ao seu potencial para facilitar a construção de aplicações na nuvem. Nesta abordagem, eventos são usados para acionar e comunicar serviços independentes e desacoplados. Isso permite que os serviços trabalhem de maneira mais flexível e escalável, o que pode ser uma vantagem em ambientes em constante mudança [AWS 2022].

O *Complex Event Processing (CEP)* é um tipo de arquitetura orientada a eventos. Ele surgiu para suportar aplicações de *streaming*, como por exemplo, análise em tempo

real e detecção de intrusão de rede. Ele opera sobre *streams* de eventos primitivos de alta velocidade e analisa/processa consultas para detectar eventos complexos, ou seja, coleções de eventos primitivos que satisfazem algum padrão [Bucchi et al. 2021].

Os sistemas CEP visam identificar situações de interesse na forma de eventos complexos. Para tais sistemas, a latência é uma métrica significativamente importante. Na detecção de fraudes, por exemplo, a falha em identificar uma situação de interesse pode levar a prejuízos significativos.

A análise/processamento de consultas CEP é conhecida por ser computacionalmente desafiadora, e esse desafio aumenta conforme a quantidade de eventos processados [Bucchi et al. 2021]. Diversas técnicas de otimização foram propostas para minimizar a latência de processamento em sistemas CEP. As propostas vão desde a proposição de diversos modelos de execução, otimização de consulta com base em custo (como é feito por sistemas de gerência de bancos de dados - SGBDs), foco em fragmentos de consulta específicos (por exemplo, políticas de seleção de eventos), uso de técnicas de rejeição de carga (*load shedding*) para obter baixa latência às custas de potencialmente perder alguns resultados, e o emprego de computação distribuída [Bucchi et al. 2021, Chapnik et al. 2021].

Uma análise de trabalhos recentes na área de CEP evidencia que, apesar de efetivas, as técnicas de otimização propostas ainda possuem limitações e oportunidades de melhoria. Por exemplo, o acúmulo de estados intermediários em sistemas CEP abordado por [Kolchinsky and Schuster 2019]. Além disso, a maioria das propostas de otimização de CEP assume que os estados intermediários estão disponíveis na memória, o que reduz drasticamente a escalabilidade horizontal do sistema [Kolchinsky and Schuster 2019, Slo et al. 2019, Chapnik et al. 2021, Mei and Madden 2009].

Sendo assim, este trabalho tem como objetivo desenvolver um sistema CEP com tempos de resposta e vazão constantes ou quase constantes, independente da quantidade de eventos processada anteriormente. Esse sistema poderá servir de base para a construção de aplicações de maior porte que processam grandes quantidades de eventos.

2. Trabalhos relacionados

A Tabela 1 apresenta um sumário das principais diferenças entre os trabalhos relacionados e a ferramenta proposta. Os trabalhos de [Kolchinsky and Schuster 2019] e [Slo et al. 2019] representam os eventos complexos e estados intermediários juntos, como autômatos finitos não-determinísticos, enquanto [Mei and Madden 2009] os representam utilizando árvores. A ferramenta proposta utiliza árvores para indexar os eventos complexos, porém, armazena os estados intermediários em um banco de dados.

Os critérios usados na comparação foram: (i) peso dos estados intermediários: capacidade de escalar independente da quantidade de eventos processados interiormente; (ii) quantidade de estados intermediários: capacidade de reduzir a quantidade de informação necessária para correlação; (iii) sobrecarga: capacidade de otimizar o desempenho sem sobrecarregar o sistema; (iv) impacto na QoR: capacidade de otimizar o desempenho sem perder qualidade dos resultados; e (v) armazenar eventos: capacidade de correlacionar eventos primitivos sem armazená-los. Os efeitos colaterais do processamento do evento são persistentes, mesmo que os eventos não sejam persistidos.

Tabela 1. Comparação

Trabalho	Reduz peso dos estados intermediários	Reduz quantidade de estados intermediários	Sem sobrecarga	Sem impacto na QoR	Sem armazenar eventos
[Kolchinsky and Schuster 2019]		✓	✓	✓	✓
[Slo et al. 2019]		✓			✓
[Mei and Madden 2009]		✓	✓	✓	
Ferramenta proposta	✓	✓	✓	✓	✓

Conforme ilustra a Tabela 1, o diferencial da ferramenta proposta é a sua capacidade de minimizar a quantidade e o peso dos estados intermediários sem sobrecarga no processamento ou impacto na qualidade dos resultados. Outra característica importante é que não precisamos manter uma cópia dos eventos. A proposta deste trabalho é apresentada na próxima seção.

3. Proposta

Esta seção apresenta um método diferente de processamento que minimiza o impacto de estados intermediários na escalabilidade de sistemas CEP. Esse método combina diversas técnicas de otimização, como políticas de seleção de eventos e compartilhamento de estado. As principais contribuições deste trabalho são: (i) uma nova representação de estado intermediário que os dissocia dos eventos complexos e possibilita armazená-los em um sistemas de bancos de dados; (ii) um modelo de dados para a escrita e leitura eficiente de estados intermediários; e (iii) uma versão experimental desse processador escrita em Go[1].

Por questões de simplificação, apenas alguns operadores (sequência, conjunção, disjunção e negação) serão suportados pela versão inicial do processador. Apesar de limitado, veremos a seguir que a combinação desses operadores possibilita ao processador representar uma vasta gama de eventos complexos. Esses operadores também podem ser usados em janelas de tempo deslizantes, que permitem delinear um período de tempo no qual todas as condições precisam ser satisfeitas. Neste momento ainda não há suporte para operadores multiclasse (que usam dois tipos diferentes de eventos) e operações de agregação (soma, máximo, mínimo, percentis, etc.)

As próximas seções detalham o seu funcionamento, incluindo um exemplo em execução para demonstrar seu potencial.

3.1. Visão geral

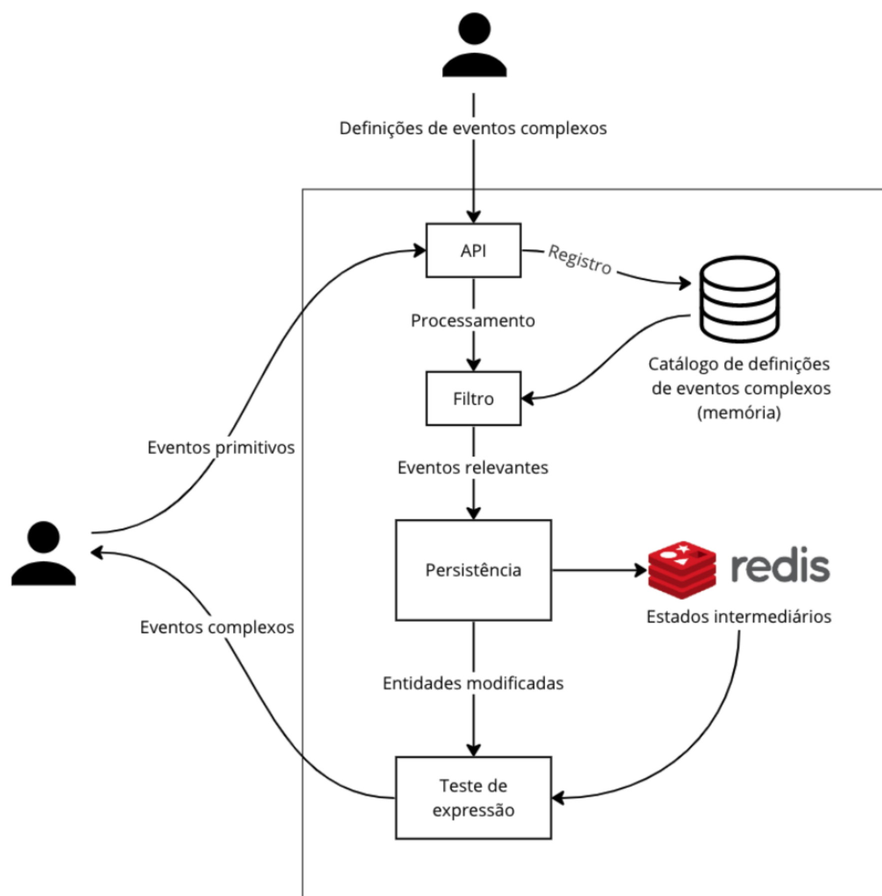
O método proposto divide o estado do processador em duas partes: (i) catálogo de definições de eventos complexos (DEC); e (ii) estados intermediários. O catálogo possui um índice em memória que permite verificar quais condições (as chamaremos de predicados), dentre todas as DECs, um evento primitivo satisfaz. Já os estados intermediários são armazenados em um banco de dados e representam quais predicados são atualmente satisfeitos.

A Figura 1 ilustra a arquitetura do processador. Além do catálogo e do banco de dados usado pra armazenar os estados intermediários, o processador possui uma API para registro de DECs e processamento de eventos primitivos. O processamento acontece em três etapas: (i) filtro; (ii) persistência e (iii) teste da expressão. Na etapa (i), são

[1] https://go.dev/

selecionados apenas os eventos relevantes para algum predicado. Na etapa (ii), os estados intermediários são criados. Na etapa (iii), é verificado quais das DECs impactadas possuem todas as condições satisfeitas, e o resultado é retornado pela API. As DECs, representadas como um objeto JSON[2], são registradas a priori por um usuário especialista, e os eventos primitivos são enviados para a API por um agente externo.

Figura 1. Arquitetura do processador



Uma DEC é composta por um identificador, algumas configurações e uma expressão lógica muito semelhante à **seleção** da álgebra relacional. Na expressão da Figura 2, os textos em amarelo são os predicados (“A” e “B”). O texto em azul é um operador lógico (“AND“), usado para representar operações de conjunção. Por fim, o texto em verde é o operador unário “NOT”, que representa a não-ocorrência da subexpressão (neste caso, “B”).

Para fins de exemplificação do funcionamento do processador, suponha a ocorrência de três eventos primitivos: (i) visita de uma página; (ii) abertura de um e-mail; (iii) *click* em um link. Suponha também a existência de uma única DEC no catálogo, caracterizada pela ocorrência dos seguintes eventos primitivos:

[2]https://www.json.org/json-en.html

Figura 2. Expressão de um evento complexo

A B
(E'.tipo == "viu_video" AND E'.duracao > 10) AND NOT (E''.tipo = "assinou_canal")

A) Evento do tipo "abriu_email" com "periodo_segundos" maior que 10 e "dispositivo" igual a "telefone";
B) Evento do tipo "clicou_link" com "url" igual a "http://site.org/quero-comprar".

A Figura 3 exemplifica como seria o objeto JSON desta DEC. O primeiro predicado representa a condição A (evento do tipo "abriu_email" com "periodo_segundos" maior que 10 e "dispositivo" igual a "telefone"). O segundo predicado representa a condição B (evento do tipo "clicou_link" com "url" igual a "http://site.org/quero-comprar"). As condições de um predicado não precisam de conector pois parte-se do princípio que todas precisam ser verdadeiras para que ele seja impactado por um evento.

Figura 3. Definição de um evento complexo

```
{
    "id": "complex-event-1",
    "logical_expression": {
        "connector": "and",
        "operands": [
            {
                "predicate": {
                    "id": "A",
                    "event_type": "abriu_email",
                    "conditions": [
                        {
                            "field": "dispositivo",
                            "operator": "equal",
                            "value": "telefone"
                        },
                        {
                            "field": "periodo_segundos",
                            "operator": "more_than",
                            "value": 10
                        }
                    ]
                }
            },
            {
                "predicate": {
                    "id": "B",
                    "event_type": "clicou_link",
                    "conditions": [
                        {
                            "field": "url",
                            "operator": "equal",
                            "value": "http://site.org/quero-comprar"
                        }
                    ]
                }
            }
        ]
    }
}
```

Suponha agora que surgiu um evento primitivo (i) mostrado na Figura 4. Ele será descartado pelo processador, pois seu tipo, "visitou_pagina", é irrelevante para a definição de evento complexo no catálogo, uma vez que os únicos tipos de eventos primitivos potencialmente relevantes são "abriu_email" e "clicou_link". Consequentemente, não emitimos nenhuma notificação durante o seu processamento.

Figura 4. Evento (i): visita em uma página

```
{
    "id": "1",
    "type": "visitou_pagina",
    "timestamp": 1000000000,
    "payload": {
        "url": "http://site.org/quero-comprar"
    }
}
```

Suponha agora um evento (ii) mostrado na Figura 5. Este evento já é relevante pois ele possui características presentes na definição de evento complexo ("type" igual a "abriu_email", "dispositivo" igual a "telefone" e "período_segundos" maior que 10). No entanto, neste momento apenas uma das condições (A) do evento complexo foi satisfeita. Portanto, nenhuma notificação é emitida durante o processamento.

Figura 5. Evento (ii): abertura de e-mail

```
{
    "id": "2",
    "type": "abriu_email",
    "timestamp": 1000000000,
    "payload": {
        "dispositivo": "telefone",
        "periodo_segundos": 60
    }
}
```

Por fim, suponha o evento (iii), mostrado na Figura 6. Este evento também é relevante pois possui características presentes na definição do evento complexo ("type" igual a "clicou_link" e "url" igual a "http://site.org/quero-comprar"). Neste momento, ambas as condições (A e B) são satisfeitas e uma notificação é enviada informando que esse evento complexo ocorreu.

Figura 6. Evento (iii) click

```
{
    "id": "3",
    "type": "clicou_link",
    "timestamp": 1000000000,
    "payload": {
        "url": "http://site.org/quero-comprar"
    }
}
```

As próximas seções detalham cada um dos componentes do processador.

3.2. Catálogo

O catálogo é composto por três componentes: (i) um hashmap com referências para todas as DECs; (ii) um hashmap que relaciona predicados e DECs; e (iii) um índice do tipo árvore onde são indexados os predicados. Esse índice é usado posteriormente na filtragem de eventos. O algoritmo de indexação utilizado é o Algorítmo 1.

Algorithm 1: Algoritmo de Indexação

Input: árvore de predicados T, hashmap ligando os predicados e DECs PD, definição de eventos complexos d

Output: árvore de predicados T atualizada, hashmaps PD atualizado

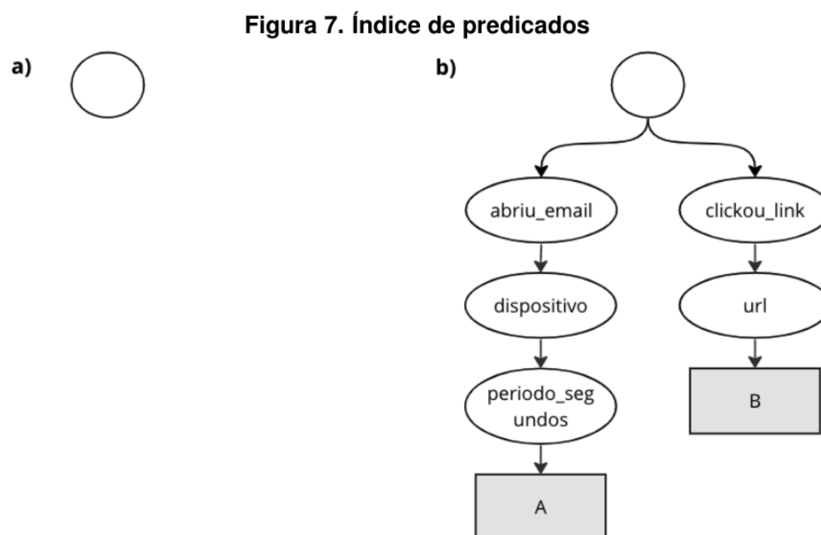
```
1 foreach p ∈ predicates(d) do
     /* Deriva um ID para o predicado com base no seu
        conteúdo. Dois predicados iguais devem sempre
        gerar o mesmo ID                                */
2    p.id ← hash(p)
     // Relaciona o predicado p a DEC d
3    PD_p ← PD_p ∪ {d}
     /* Extrai os campos usados nas condições. Por
        exemplo, para um evento complexo com condições
        dispositivo=Android, periodo_segundos=10 essa
        função retornaria ⟨ dispositivo,
        periodo_segundos ⟩. A lista está sempre em
        ordem lexicográfica                              */
4    ck ← extractConditionFields(p)
     /* Inicializa o caminho K com o tipo do evento e
        adiciona as chaves das condições ck. O
        resultado é algo como K = ⟨ abriu_email,
        dispositivo, periodo_segundos ⟩                  */
5    K ← ⟨p.eventType⟩ ∪ ck
     // Percorre o caminho K dentro da árvore e
        retorna o nodo final. Os nodos que não
        existirem são criados
6    n ← findNode(T, K)
7    n.predicates ← n.predicates ∪ {p}
8 return (T, PD)
```

No Algorítmo 1, a função $hash$ é responsável por gerar um identificador único para o predicado com base no seu conteúdo. Uma propriedade importante dessa função é que predicados iguais sempre geram o mesmo ID, minimizando a quantidade de predicados. Veremos a seguir que essa propriedade é essencial para a escalabilidade solução.

A Figura 7 mostra o estado do índice de predicados antes ("a") e depois ("b") da execução do Algorítmo 1 para a DEC da Figura 3. O predicado A está em $abriu_email \rightarrow dispositivo \rightarrow periodo_segundos$, sendo relevante quando houverem os campos "dispositivo"e "periodo_segundos" em um evento do tipo "abriu_email". O predicado B está em $clicou_link \rightarrow url$, sendo relevante sempre que houver um campo "url" em um evento de

"clicou_link".

Figura 7. Índice de predicados



Assumindo que a árvore não esteja vazia, ela sempre terá pelo menos um nível: o tipo do evento. Esse nível nos permite descartar mais rápido os eventos irrelevantes, pois já no primeiro nível é possível saber se há ou não algum predicado para o tipo do evento sendo processado. A próxima seção detalha a utilização do catálogo no processamento de eventos.

3.3. Processo de filtro

A primeira etapa do processamento consiste em um filtro para descartar eventos irrelevantes. Neste contexto, são irrelevantes os eventos cujo conteúdo não atende as condições de nenhum predicado de nenhuma DEC do catálogo. Por exemplo, um evento do tipo "clicou_link" seria irrelevante para o predicado A da Figura 3, porque ele só está interessado em eventos do tipo "abriu_email". Seria igualmente irrelevante tanto para o predicado A quanto para o predicado B um evento cujo valor do campo "dispositivo" seja "computador". O descarte é importante para economizar recursos computacionais que seriam usados no processamento destes eventos.

Ao final desta etapa sabemos quais eventos primitivos são relevantes para alguma DEC, e o porquê. Seu resultado é um conjunto de tuplas, compostas por um evento, quais predicados foram impactados por esse evento, e quais DECs usam esses predicados. São essas tuplas que são encaminhadas para a próxima etapa. Os eventos irrelevantes geram tuplas sem nenhum predicado que devem ser descartadas.

O processo de filtro, detalhado no Algorítmo 2, é dividido em três subetapas: (i) identificar predicados potencialmente relevantes; (ii) verificar se as condições do predicado são todas verdadeiras, e então; (iii) buscar as expressões que utilizam os predicados encontrados. A subetapa (i), detalhada no Algorítmo 3, tem por objetivo identificar quais predicados podem estar interessados no evento que está sendo processado. É importante ressaltar que nesta etapa ainda não verificamos o conteúdo das condições, somente a presença dos campos.

```
Algorithm 2: Algoritmo de Busca
   Input: árvore de predicados T, evento primitivo e
   Output: lista de predicados impactados pelo evento e
   /* Função similar à extractConditionFields
      utilizada no Algorítmo 1, porém com um tipo
      diferente.  Ao invés de receber um predicado,
      recebe um evento.  A lista está sempre em ordem
      lexicográfica                                      */
 1 pk ← extractPayloadKeys(e)
   /* Inicializa o caminho K com o tipo do evento e
      adiciona as chaves do payload pk.  O resultado
      será algo como K = ⟨ abriu_email, dispositivo,
      periodo_segundos ⟩                                 */
 2 K ← ⟨p.eventType⟩ ∪ pk
   // Percorre todas as permutações de K e retorna
      todos os predicados encontrados pelo caminho.
      Esse algoritmo é descrito no Algorítmo 3
 3 P ← searchPredicates(T, K)
 4 P' ← ∅
 5 foreach p ∈ P do
      /* Esta função retorna verdadeiro se todas as
         condições do predicado p forem satisfeitas
         pelo evento e                                   */
 6    if evaluateConditions(p, e) then
 7       P' ← P' ∪ {p}
 8 return P'
```

Na subetapa (ii), detalhada na linha do 6 do Algorítmo 2, o objetivo é descobrir quais predicados possuem todas as condições verdadeiras. No caso do predicado A da Figura 3, por exemplo, o evento precisa ter "dispositivo" igual a "telefone" e "periodo_segundos" maior que 10. Os predicados que não tiverem todas as condições verdadeiras serão desconsiderados.

Por fim, na subetapa (iii), detalhada na linha 7 do Algorítmo 2, identifica-se quais expressões utilizam os predicados impactados, pois se um evento EV_i modificar o estado de um predicado P_j, então os resultados das expressões que usam P_j podem ter sido alterados por EV_i. Nesta etapa ainda não sabemos efetivamente se houve ou não mudança no resultado da expressão por que para tal precisamos testá-la novamente. O foco desta etapa é apenas identificar quais expressões precisam ser testadas.

Algorithm 3: Algoritmo auxiliar "$searchPredicates$"

Input: árvore de predicados T e um caminho a ser percorrido K
Output: todos os predicados encontrados no caminho K

```
1 P ← ∅
2 foreach p ∈ T.predicates do
3     P ← P ∪ p
4 foreach (i, k) ∈ K do
      // Subárvore identificada pela chave k
5     T' ← T_k
6     if T' ≠ ∅ then
          // Percorre verticalmente a árvore para todas
             as chaves que ainda faltam visitar
7         P ← P ∪ searchPredicates(T', K_{i..|K|})
8 return P
```

3.4. Processo de persistência

A segunda etapa do processamento consiste em persistir os estados intermediários de modo que no futuro seja possível saber quais condições para a ocorrência de um evento complexo já foram satisfeitas. Só chegam nesta etapa as tuplas de eventos que satisfazem ou invalidam algum predicado, pois sem que haja mudança no estado é impossível que o resultado de uma expressão seja alterado. Por exemplo, só chegam nesta etapa os eventos (ii) e (iii) (Figura 5 e Figura 6), que satisfazem, respectivamente, os predicados A e B da Figura 3. O evento (i) (Figura 4) é descartado porque não satisfaz ou invalida nenhum predicado.

Por questões de simplicidade, o banco de dados escolhido para a persistência dos estados intermediários foi o Redis[3]. O Redis é um repositório de dados em memória, fácil de usar e com padrões de acesso chave-valor extremamente rápidos. Ele está amplamente disponível em qualquer *cloud* e pode ser executado em qualquer computador usando o Docker[4]. Além disso, ele possui uma API simples e bem documentada, o que facilita a implementação do processador.

[3]https://redis.io/
[4]https://www.docker.com/

O algoritmo de persistência é descrito no Algorítmo 4. O modelo de dados utilizado é descrito a seguir, na Subsubseção 3.4.1.

Algorithm 4: Algoritmo de Persistência.

Input: lista de predicados impactados P, hashmap com a relação entre predicados e expressões PD, evento primitivo e
Output: DECs que precisam ser testadas D

```
   // Timestamp que será usado para versionar o
      estado.
1  t ← e.timestamp
   // Inicializa S e D como listas vazias.
2  S ← ∅
3  D ← ∅
   // Para cada predicado p.
4  foreach p ∈ P do
      // Define o predicado p como verdadeiro no
         momento t.
5     S_{p,t} ← true
      // Testar todas as DECs associadas ao predicado
         p.
6     D ← D ∪ PD_p
   // Persiste o estado no banco de dados.
7  save(S)
8  return D
```

Na linha 5 do Algorítmo 4, é especificado o momento em que estado intermediário foi criado. Essa informação é necessária para que seja possível suportar janelas de tempo e o operador de sequência. Por exemplo, considere as seguintes DECs: (i) “A e B em um intervalo de 30 segundos” e (ii) “A depois de B”. No caso (i), temos uma janela de tempo. Neste exemplo precisamos saber para um evento qualquer EV_i que impactou A ou B, se houve, respectivamente, um B ou A a 30 segundos de distância. No caso (ii), temos um operador de sequência. Nele precisamos verificar para um evento qualquer EV_j que impactou A ou B, se há, respectivamente, um B ou A posterior. Em ambos os casos é preciso saber quando A e B ocorreram.

3.4.1. Modelo de dados

A estrutura de dados *sortedset* do Redis é uma ótima alternativa para modelar séries temporais de dados simples. A maioria das operações sobre *sortedsets* possuem complexidade de tempo O(log(N))[5], o que escala bem para caso de uso proposto. Nesta modelagem, a chave, o *score* e o valor serão, respectivamente, o ID do predicado, o *timestamp* do evento, e o ID do evento.

A Figura 8 ilustra o comando usado para adicionar um item no *sortedset*. No exemplo, o evento “2” impactou o predicado A no *timestamp* “1000000000”.

[5]https://redis.io/commands/?group=sorted-set

Figura 8. Adicionando um item a um *sortedset*

```
127.0.0.1:6379> ZADD A 1000000000 2
(integer) 1
127.0.0.1:6379>
```

A Figura 9 ilustra o comando usado para buscar um item no *sortedset*. No exemplo, será retornado para os predicados A e B se algum evento os impactou entre os *timestamps* “999999700” e “1000000300”. No caso, o evento “2” impactou o predicado A no *timestamp* “1000000000”. Nenhum evento impactou o predicado B no intervalo de tempo especificado.

Figura 9. Lendo um item de um *sortedset*

```
127.0.0.1:6379> ZRANGE A 999999700 1000000300 BYSCORE LIMIT 0 1 WITHSCORES
1) "2"
2) "1000000000"
127.0.0.1:6379> ZRANGE B 999999700 1000000300 BYSCORE LIMIT 0 1 WITHSCORES
(empty array)
127.0.0.1:6379>
```

Diferentemente dos processadores baseados em NFA e em árvore, os estados intermediários não são o estado atual de uma instância de evento complexo. Em vez disso, são os predicados que já foram satisfeitos. Essa abordagem maximiza o compartilhamento de estado entre diferentes DECs, pois, durante o processamento de um evento, apenas um estado intermediário será criado mesmo que centenas de DECs usem o mesmo predicado. Outra vantagem dessa abordagem é que ela permite que o processador correlacione eventos primitivos sem armazená-los. Isso é possível pois o estado intermediário contém tudo que o processador precisa saber sobre os eventos primitivos que o criaram.

Considere a DEC da Figura 3, por exemplo. Independente da quantidade de eventos primitivos previamente processados, durante o processamento de um evento EV_i, no máximo dois valores precisariam ser lidos, A e B. Essa propriedade tem forte influência sobre a escalabilidade do processador, pois enquanto métodos convencionais escalam de acordo com a quantidade de eventos, o método proposto escala de acordo com a quantidade de predicados. Isso significa que o método proposto é extremamente eficiente em cenários onde há menos predicados que eventos, como por exemplo, cenários de IoT. Por outro lado, ele pode ser menos eficiente em cenários onde há muitos predicados, como por exemplo, cenários de análise de dados.

3.5. Processo de teste da expressão

A terceira e última etapa do processamento consiste em testar as expressões impactadas pela modificação no estado. Nesta etapa, o processador carrega do banco de dados todos os estados intermediários necessários para testar as expressões. Em seguida, para cada expressão, o processador testa se as condições especificadas na DEC são satisfeitas. O Algorítmo 5 detalha o processo de teste da expressão.

No Algorítmo 5, a função $load$ é responsável por carregar do banco de dados os estados dos predicados que compõem a expressão. A função recebe como parâmetro a DEC cuja expressão deve ser testada, o tempo mínimo e máximo que delimitam a janela

Algorithm 5: Algoritmo de Teste da Expressão

Input: lista de DECs cujas expressões devem ser testadas D, evento primitivo e

Output: hashmap com o resultado das expressões R

```
1  R ← ∅
2  foreach d ∈ D do
3      S ← ∅
       /* Delimita a janela de tempo na qual as
          condições devem ser satisfeitas.  Assume-se
          para efeito de simplificação que a janela de
          tempo também esteja em segundos              */
4      if d.within > 0 then
5          t_min ← e.timestamp − d.within
6          t_max ← e.timestamp + d.within
7      else
8          t_min ← −∞
9          t_max ← ∞
       /* Carrega o estado dos predicados da DEC d do
          banco de dados.  O funcionamento dessa função
          será detalhado a seguir.                     */
10     S ← load(d, t_min, t_max)
       // Invoca o algoritmo auxiliar Algorítmo 6 para
          testar a expressão
11     R_d ← evaluateExpression(S, d.logicalExpression)
12 return R
```

de tempo na qual as condições devem ser satisfeitas. A função retorna um hashmap com o estado dos predicados, onde o valor é um booleano, e a chave é um array com os identificadores dos predicados. Por exemplo, se a DEC possui dois predicados, A e B, o hashmap retornado pela função terá dois elementos, um para cada predicado. O primeiro elemento terá como chave a tupla (A) e o segundo elemento terá como chave a tupla (B). O valor será verdadeiro quando houver alguma ocorrência de A ou B dentro da janela $t_{min}..t_{max}$.

Há um outro caso que deve ser considerado. Quando dois predicados estão conectados por um operador de sequência $(B; A)$, não basta que hajam ocorrências de ambos os predicados dentro da janela de tempo $t_{min}..t_{max}$. É preciso que as ocorrências estejam na ordem correta. Por exemplo, suponha que a expressão da DEC do exemplo anterior seja $B; A$. Neste caso (c), é preciso que haja um B entre $t_{min}..t_{max}$, e que haja um A entre $t_b..t_{max}$, onde t_b é o *timestamp* da primeira ocorrência de B dentro da janela de tempo. Neste caso, o hashmap resultante seria $\{(A) \rightarrow true, (B, A) \rightarrow true\}$.

O Algorítmo 6 detalha o algoritmo auxiliar que testa a expressão (função *evaluateExpression*). Uma expressão é composta por um ou mais operandos conectados por um operador. Um operando pode ser um predicado ou uma expressão lógica. Quando ele for uma expressão lógica, o algoritmo é invocado recursivamente (linha 4). O funcionamento do algoritmo para cada um dos operadores é detalhado a seguir.

4. Experimentos

Esta seção descreve o experimento realizado com o processador de eventos proposto neste trabalho. O experimento utilizou um *dataset*[6] de transações da NASDAQ. O *dataset* é uma amostra de 224.473 transações de compra e venda de ações, e possui as seguintes informações: código da ação, ID da transação, volume, preço, hora (convertida para *timestamp*), tipo (compra[B]/venda[S]). Todos os testes rodaram por 120 segundos, e durante esse período um mesmo evento pode ser processado duas vezes. Por esse motivo, não utilizamos o ID da transação presente no *dataset* e sim uma *string* randômica. Isso garante que o processador entenda que se trata de um evento diferente e o processe novamente.

O Apache JMeter[7] foi utilizado para carregar o *dataset* e simular requisições HTTP para o processador a partir de 200 threads. Foram também utilizados o *Prometheus*[8] para a coleta de métricas e o *Grafana*[9] para visualização. O processador foi executado em um container Docker com 8 CPUs, 8GB de RAM, 1GB de swap e 200GB de disco, como mostrado na Figura 10. Os containers rodaram de um Macbook Air M1 com 8 núcleos, 4 de performance e 4 de eficiência, e 16GB de RAM. O Redis rodou como um container separado, dentro da mesma rede.

O experimento avalia o desempenho do processador com uma única DEC no catálogo e nada no Redis. A DEC usada identifica transações de venda da ação MSFT com preço superior a 26, seguidas de transações de compra da ação INTC com qualquer valor, em uma janela de 1 hora. Figura 11, Figura 12 e Figura 13 mostram o *throughput*,

[6]https://davis.wpi.edu/datasets/Stock_Trace_Data/
[7]https://jmeter.apache.org/
[8]https://prometheus.io/
[9]https://grafana.com/

Algorithm 6: Algoritmo auxiliar "$evaluateExpression$"

Input: seção do estado S, expressão lógica que deve ser testada e
Output: booleano resultante r

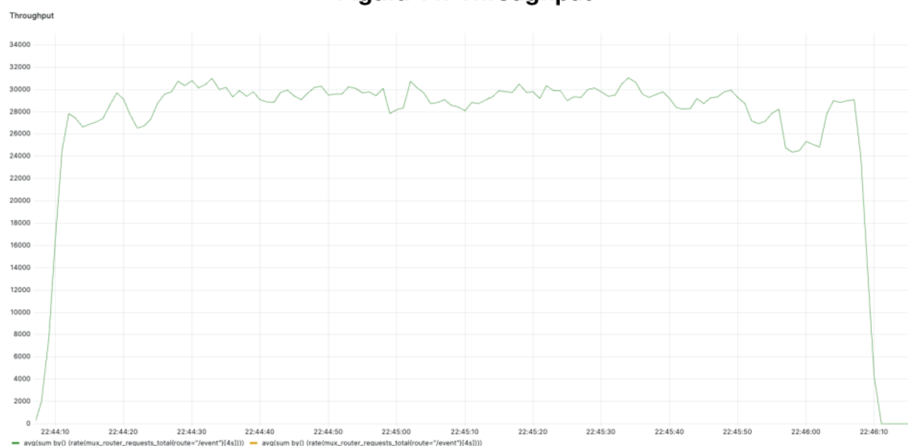
```
   // Array com o valor dos operandos da expressão.  O
      valor é sempre um booleano
1  V ← ∅
2  foreach o ∈ e.operands do
3      if o.logicalExpression then
           // Invoca recursivamente o algoritmo para
              avaliar a subexpressão
4          V ← V ∪ evaluateExpression(S, o.logicalExpression)
5      else
6          p ← o.predicate
           /* Quando o conector é um operador de
              sequência, retorna uma lista P com os
              predicados que aparecem antes de p na
              expressão.  Quando o conector é ``and'' ou
              ``or'', retorna uma lista vazia.         */
7          P ← p.dependsOn
           // Carrega o estado de p em S.
8          v ← S_{P∪{p}}
           // Inverte o valor do predicado se ele for
              negado
9          if o.negated then
10             v ← ¬v
11         V ← V ∪ {v}
12 r ← false
   // Conecta os valores com o operador lógico
13 if e.operator = "or" then
       // Retorna true quando algum dos valores for true
14     r ← ∃v ∈ V : v = true
15 else
       // Retorna true quando todos os valores são true
16     r ← ∀v ∈ V : v = true
17 return r
```

Figura 10. Configurações do Docker

os tempos de resposta e o consumo de memória, respectivamente. O *throughput*, o tempo de resposta e o consumo de memória se mantiveram praticamente constantes durante todo o experimento.

É possível observar pelo *throughput* e pelo tempo de resposta que o processador não escala em função da quantidade de eventos previamente processados. Essa característica contrasta com os processadores baseados em NFA e árvore, que precisam percorrer todos os estados intermediários criados até o momento para cada evento processado.

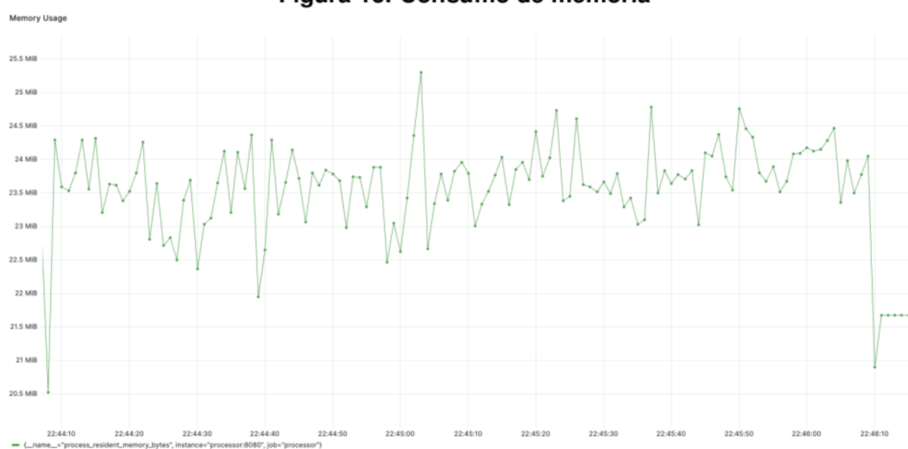
Figura 11. Throughput



Na Figura 12, os percentis 99, 95 e 50 são representados, respectivamente, pelas linhas verde, azul e laranja.

Figura 12. Tempos de resposta

Figura 13. Consumo de memória



5. Conclusão

Este trabalho teve como objetivo principal o desenvolvimento de um processador de eventos complexos eficiente, flexível e que escale bem com o número de eventos processados. Para isso, foi proposto um método diferente de processamento que minimiza o impacto de estados intermediários na escalabilidade de sistemas CEP.

As principais contribuições desse trabalho são: (i) uma nova representação de estado intermediário que os dissocia dos eventos complexos e permite armazená-los em sistemas de bancos de dados; (ii) um modelo de dados para a escrita e leitura eficiente de estados intermediários; (iii) O projeto de um processador de CEP; e (iv) Uma versão experimental desse processador escrita em Go[10].

Os experimentos realizados demonstram a capacidade do método proposto de processar um grande volume de eventos em tempo constante, e com baixo consumo de memória. Dessa forma, o objetivo geral do trabalho pode ser considerado atingido.

Como atividades futuras relacionadas a este trabalho considera-se: (i) adicionar suporte a mais operadores, como o *fecho de Kleene*. Suportar mais operadores ampliaria a expressividade da linguagem de consulta e permitiria suportar uma variedade maior de eventos complexos; (ii) adicionar diferentes estratégias de janelas de tempo, como janelas fixas. Atualmente só é suportada a janela deslizante (últimas x horas, por exemplo), mas a janela fixa é uma estratégia comum em sistemas de CEP. (iii) resolver o problema da repetição de eventos. Atualmente, uma expressão como $A; B; C$ testará positivo pela primeira vez que A, B e C ocorrerem em sequência, mas testará positivo para cada A, B ou C que ocorrerem depois disso. Idealmente, a expressão deveria testar positivo apenas para novas triplas A, B, C; (iv) eliminar estados intermediários que não são mais relevantes para DECs. Por exemplo, se só há uma DEC cuja janela de tempo é de uma hora, é possível que os estados intermediários do dia anterior não sejam mais relevantes; e (v) comparar o desempenho desta proposta com outros processadores de CEP disponíveis.

Referências

[AWS 2022] AWS (2022). Getting started with event-driven architecture. Website.

[Bucchi et al. 2021] Bucchi, M., Grez, A., Quintana, A., Riveros, C., and Vansummeren, S. (2021). Core: a complex event recognition engine. *arXiv preprint arXiv:2111.04635*.

[Chapnik et al. 2021] Chapnik, K., Kolchinsky, I., and Schuster, A. (2021). Darling: data-aware load shedding in complex event processing systems. *Proceedings of the VLDB Endowment*, 15(3):541–554.

[Kolchinsky and Schuster 2019] Kolchinsky, I. and Schuster, A. (2019). Real-time multi-pattern detection over event streams. In *Proceedings of the 2019 International Conference on Management of Data*, pages 589–606.

[Mei and Madden 2009] Mei, Y. and Madden, S. (2009). Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 193–206.

[10]https://go.dev/

[Slo et al. 2019] Slo, A., Bhowmik, S., Flaig, A., and Rothermel, K. (2019). pspice: partial match shedding for complex event processing. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 372–382. IEEE.