FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Complex Event Processing (CEP) - Using SQL Server StreamInsight for near real-time visualization and monitoring

Rolando Emanuel Lopes Pereira

Mestrado Integrado em Engenharia Informática e Computação

19 de junho de 2013

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Complex Event Processing (CEP) - Using SQL Server StreamInsight for near real-time visualization and monitoring

Rolando Emanuel Lopes Pereira

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Armando Sousa

19 de junho de 2013

Complex Event Processing (CEP) - Using SQL Server StreamInsight for near real-time visualization and monitoring

Rolando Emanuel Lopes Pereira

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: José Manuel de Magalhães Cruz

Arguente: Luís Paulo Reis

Vogal: Armando Jorge Miranda de Sousa

---

19 de junho de 2013

Resumo

Atualmente, com o conceito de *Internet of Things*, existem cada vez mais dados que têm de ser processados em tempo real. Estes dados podem ter origens variadas, como por exemplo, dados de um sensor de temperatura ou dados de um sistema de *tracking online*. Esses dados surgem sob a forma de *streams* de eventos que ocorrem a um dado instante de tempo.

Uma forma popular de processar essas *streams* de eventos é através da utilização de sistemas de *Complex Event Processing*. Estes permitem processar *streams* de eventos em tempo real e construir janelas temporais sobre essas *streams*, podendo depois aplicar agregações sobre as mesmas. Normalmente esta funcionalidade é obtida através da adição de funcionalidades à linguagem SQL por parte de um motor de *Complex Event Processing*, permitindo que se possa utilizar uma linguagem semelhante SQL para processar *streams* através da construção de *queries* e criar janelas temporais sobre as mesmas.

Infelizmente muitos sistemas de CEP requerem conhecimento *à priori* do tipo (*payload*) de eventos que terão de processar bem como do tipo de *queries* que irão ser executadas sobre eles.

O motor de CEP utilizado nesta dissertação foi o *Microsoft StreamInsight*.

Nesta dissertação é proposta uma arquitetura que consegue processar eventos genéricos em *runtime* e criar várias janelas temporais sobre esses eventos através de uma adição ao motor do *StreamInsight*, transformando este último num sistema que consiga processar eventos genéricos sem ter conhecimento *à priori* do seu *payload* desde que esses eventos estejam representados no formato XML. Para validar essa arquitetura foi criado o *CmStream*.

Mostramos nesta dissertação que, para problemas envolvendo agregações temporais, o desempenho do *CmStream* é superior, em tempo e espaço, aos sistemas de base de dados, mesmo para conjuntos de dados pequenos. Mostramos também que, para *queries* temporais simples, o desempenho do *CmStream* é semelhante ao desempenho dos sistemas de NoSQL. No entanto mostramos também que o desempenho do *CmStream* é ligeiramente inferior a uma solução implementada utilizando apenas o *StreamInsight*.

Abstract

Nowadays, there is a lot of data that needs to be processed in real time and it is expected that with the proliferation of the Internet of Things. This data comes from different places, from temperature sensors to an online user tracking system. This data appears in the form of streams of events that happen at a given moment of time.

A popular way to process those event streams, is by using Complex Event Processing. This technology allows the processing, in real time, of event streams, the creating of time windows over those streams and the use of aggregations on those windows. Usually this functionality is gained by using a CEP engine that extends the SQL language allowing the latter to process streams by constructing queries and create temporal windows on them.

Unfortunately before using a CEP system, many require à priori knowledge regarding the type (i.e. payload) of events that can appear on their streams and what queries it can run.

The CEP engine used in this dissertation was Microsoft's StreamInsight.

We propose in this dissertation an architecture that can process at runtime events with generic payloads and allows the creation of several time windows over those events. This architecture is obtained through an addition to the StreamInsight engine that allows it to processing process generic events without à priori knowledge of their payload as long as those event have a XML representation. To validate this architecture we created CmStream.

We show in this dissertation that for problems requiring temporal aggregations the performance of CmStream is better, in both time and space, than the performance of a database system, even for small datasets. We also show that for simple temporal queries the performance of CmStream is similar to the performance of a NoSQL system. However we also show that the performance of CmStream is slightly worse than a solution implemented using only StreamInsight.

Agradecimentos

Gostaria de agradecer em primeiro lugar à minha família e amigos, especialmente aos meus pais que sempre me incentivaram e apoiaram durante o meu percurso académico.

Gostaria também de agradecer ao meu orientador o Prof. Armando Jorge Sousa pelos conselhos e apoio fornecidos durante o decorrer desta dissertação.

Finalmente gostaria de agradecer à Critical Manufacturing por me fornecer a oportunidade de poder trabalhar neste projeto e ao Eng. Ricardo Magalhães por me acompanhar durante o percurso desta dissertação.

Rolando Pereira

"Everything flows, nothing stands still."

Heraclitus

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

CEP	Complex Event Processing
EPL	Event Processing Language
CQL	Continuous Query Language
SQL	Structured Query Language
LINQ	Language Integrated Query
API	Application Programming Interface
UDF	User Defined Function
UDA	User Defined Aggregate
UDO	User Defined Operator
UDSO	User Defined Stream Operator
CSV	Comma-Separated Values
XML	Extensible Markup Language
ECA	Event-Condition-Action
CTI	Current Time Increment

Capítulo 1

Introdução

1.1 Contexto

Existe cada vez mais informação disponível aos sistemas de *software*: estima-se que a *Internet of Things*, atinja os milhares de milhões de dispositivos em 2015 [WBC+09], fazendo com que exista mais dispositivos ligados à *Internet* do que pessoas. Existe também um grande aumento de sensores e leitores *RFIDs* nesses dispositivos [CES04, Dav11].

A informação vinda desses dispositivos surge sobre a forma de eventos. Estes eventos representam acontecimentos que são válidos durante um dado intervalo de tempo podendo, no extremo, só serem válidos durante um instante de tempo, sendo nesse caso designados por "eventos instantâneos". Por exemplo o evento "Entrada de peça na máquina" é um evento instantâneo enquanto que o evento "Processamento de peça" é um evento que existe durante um intervalo de tempo.

Um evento contem também um conjunto de dados, designado por *payload* e representado por um tuplo, que indica informação sobre esse evento. No exemplo do evento "Entrada de peça na máquina" um evento podia ter os dados "MachineId" e "PieceId".

Essa informação tem um tempo de vida útil muito reduzido, por isso tem de ser processado em tempo real, o que não é possível fazer utilizando bases de dados tradicionais e *data warehouses* pois estas não estão preparadas para realizar processamento em tempo real. [JAF+06]. Isso faz com que quando estes sistemas têm uma resposta a uma *query* esta já não é útil para o negócio.

Esta informação pode também surgir em grande quantidade. Por exemplo o sistema *ATLAS* do *Large Hadron Collider* tem que processar 100 milhões eventos por segundo[1] [dABB+08].

Como tal foi necessário desenvolver sistemas que consigam processar esses eventos. Esses sistemas designam-se por Complex Event Processing.

Ao contrário dos sistemas de bases de dados relacionais que só dão uma resposta por *query*, os sistemas de *Complex Event Processing* implementam o conceito de "*Continuous Queries*", que são *queries* que são aplicadas sobre *streams* de dados potencialmente infinitas [LWZ11] e que atualizam a sua resposta imediatamente quando ocorre um evento. Isto permite que as "*Continuous Queries*" possam atualizar as suas respostas em tempo real sem terem que analisar novamente a

[1] Esta capacidade de processamento só é obtida através da utilização de hardware específico

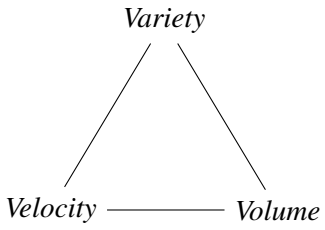

Figura 1.1: A pirâmide da *Big Data*. Os elementos nos extremos designam os 3 Vs da Big Data.

stream de dados completa, ao contrário de uma base de dados relacional, que precisa de aceder aos dados todos das tabelas cada vez que a *query* é atualizada.

Um outro desafio que existe atualmente é o processamento de "Big Data", estimando-se que o volume de dados mundial cresça 59% por ano. [Gar11]

Em [Gar11] são indicadas as 3 características da *Big Data*: Processamento de um elevado volume de dados, processamento de dados com elevada velocidade e processamento de eventos com grande variedade. Juntas, estas três características criam aquilo que se designa "The 3 Vs of Big Data". Esta é representada na figura 1.1 que também é conhecida como "The Big Data Pyramid".

Neste trabalho foi utilizado o sistema de *Complex Event Processing* desenvolvido pela *Microsoft*, o *StreamInsight*. Este é um sistema de *Complex Event Processing*, distribuído como um conjunto de ficheiros ".dll" que podem ser utilizados com qualquer programa escrito em C#.

Como será vista na secção 4.2, um dos problemas do *StreamInsight* é que este requer o conhecimento à priori de todos os tipos de eventos que possam surgir durante a sua execução. Isto impede que o *StreamInsight* atinja o último V da pirâmide da *Big Data* da figura 1.1, o processamento de eventos com grande variedade.

O projeto *CmStream* passou pela criação de uma plataforma de *Complex Event Processing* implementada sobre o *StreamInsight* utilizando as funcionalidades de extensão do mesmo que permitisse a criação de *queries* em *runtime* e que estas pudessem ser utilizadas sobre qualquer tipo de eventos. Como se pode ver na figura 1.2 isto faz com que o *CmStream* atinja o último V da *Big Data*: processamento de eventos com grande variedade.

Neste trabalho foi utilizado a versão *Standard* do *StreamInsight* 2.1 que é disponibilizada gratuitamente pela *Microsoft*. A versão *Premium* do *StreamInsight* contêm algumas funcionalidades extra como a recuperação do estado interno da aplicação caso esta seja repentinamente interrompida[MSDf]. No entanto essas funcionalidades extras não foram necessárias para este trabalho, pelo que se escolheu utilizar a versão *Standard* do *StreamInsight* 2.1.

1.2 Contribuições da dissertação

As contribuições desta dissertação são três:

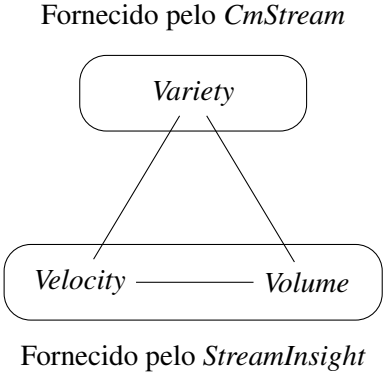

Figura 1.2: Posicionamento do *StreamInsight* e do *CmStream* na pirâmide da *Big Data*. O *StreamInsight* fornece processamento de dados com grande volume e velocidade, e o *CmStream* fornece o processamento de eventos com grande variedade.

1. A especificação formal de um conjunto de gramáticas que permita utilizar um sistema de *Complex Event Processing* a criação de janelas temporais sobre *streams* através do evento de eventos contendo XML.

2. A descrição de um algoritmo que permita gerar numa *stream* de eventos um conjunto de janelas temporais de tamanho variável, designadas de *Session Windows*, através da especificação de eventos que representam o início e fim de cada janela.

3. É também especificado um problema de análise de *logs*, sendo feita uma análise do desempenho dos sistemas de *Complex Event Processing* e esta comparada com o desempenho de um sistema de base de dados.

1.3 Estrutura da dissertação

O resto da dissertação está organizada da seguinte forma: o capítulo 2 faz revisão da literatura e descreve o estado da arte em sistemas de *Complex Event Processing*. O capítulo 3 faz uma análise ao sistema *StreamInsight*, que é uma tecnologia de Complex *Event Processing* desenvolvida pela *Microsoft*. O capítulo 4 fala do *CmStream*, tendo sido esse o projeto desenvolvido nesta dissertação. O capítulo 5 define o caso de estudo que foi utilizado para validar o desempenho da arquitetura do *CmStream* e comparar-lo com o desempenho de um sistema de base de dados tradicional. Finalmente o capítulo 6 conclui a dissertação.

Capítulo 2

Revisão da Literatura e do Estado da Arte

A secção seguinte descreve o trabalho recente realizado na área dos sistemas de *Event Processing*, em particular na área dos sistemas de *Complex Event Processing*.

A ideia por detrás dos sistemas de *Complex Event Processing* é realizar processamento de eventos, em tempo real, de forma a que se consiga extrair informação nova [TGP05] através do processamento de eventos. Essa informação nova é representada como sendo novos eventos, designados "Eventos Complexos", sendo um evento algo que acontece num determinado instante de tempo, ou válido por um intervalo de tempo, e que tem associado a si, para além de um tempo de início e de fim, um tuplo de dados que constituem o "*payload*" do evento [GHM+]. Estes eventos são modelados por um sistema de *Complex Event Processing* como sendo "*streams*", potencialmente infinitas, de eventos. Um exemplo de uma *stream* de eventos cujos "*payloads*" são números pode ser vista na figura 2.1.

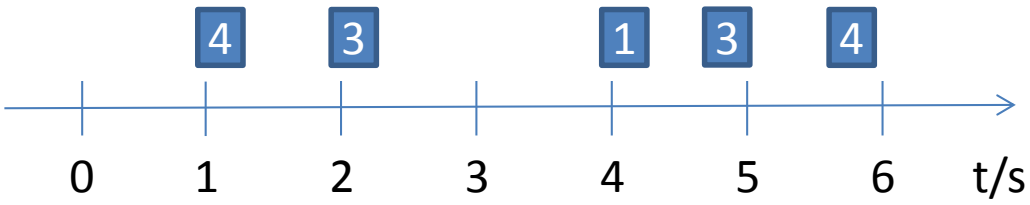

Figura 2.1: Exemplo de uma *stream* de eventos cujos *payloads* são números.

Um exemplo da deteção de um evento complexo seria detetar um incêndio através da utilização de sensores de fumo e de temperatura. Apesar de não existir nenhum sensor de incêndio neste exemplo, é possível detectar-los usando um sistema de *Complex Event Processing* através da seguinte regra: "Existe um incêndio quando há um sensor de fumo que é ativado num espaço de 5 minutos de um sensor de temperatura medir uma temperatura superior a 45ºC".

Um fator importante nesta regra é o conceito de tempo, pois só estamos interessados em detetar eventos que ocorram passados 5 minutos após a ocorrência de outro evento. Esse conceito está exemplificado na figura 2.2. Vê-se assim que o conceito de tempo é bastante importante na área

do processamento de eventos. Será visto na secção 4.1 que o conceito de tempo interno das *streams*, designado por "*Application Time*" é independente do tempo no "mundo real", designado de "*System Time*" [SW04].

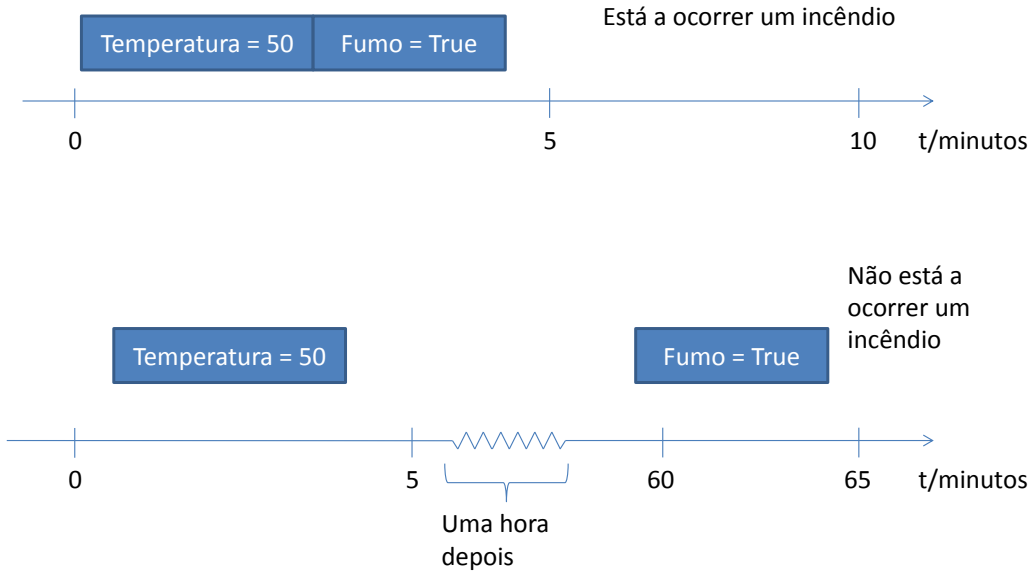

Figura 2.2: Exemplo da importância do tempo nos sistemas de processamento de eventos.

Inicialmente os sistemas de processamento de eventos foram implementados sobre os sistemas de base de dados através da criação de funcionalidades que permitissem especificar comportamentos reativos na base de dado. Normalmente estas funcionalidades era obtidas através da adição do conceito de "triggers" ao sistema de base de dados, criando assim as "Active Databases" [DGG95], que são um exemplo do paradigma *Event-Condition-Action*, normalmente designado por ECA. Este paradigma é das maneiras mais simples de realizar processamento de eventos utilizando os sistemas de base de dados [TGP05, ZU99].

No entanto, ao longo da década de 90 começou-se a detetar problemas no uso de "triggers" de base de dados para o processamento eventos [SD95]. Apesar de alguns desses problemas terem sido corrigidos ao longo dessa década, continuaram a existir problemas que ainda não tinham sido resolvidos. O artigo [CCW00] mostra algumas dessas soluções.

Mais recentemente, foi também estudada a capacidade de implementar sistemas de processamento de eventos sobre base de dados relacionais, através da utilização das funcionalidades de programação que estas fornecem. Um exemplo de um desses sistemas pode ser visto em [GHM$^+$] que mostra como implementar um sistema de processamento de eventos sobre uma base de dados utilizando a tecnologia JBDC que permite aceder a várias bases de dados utilizando a linguagem Java. Devido ao fato de a tecnologia JBDC ser independente da base de dados utilizada, é possível facilmente substituir a base de dados utilizada (por exemplo, trocar uma base de dados *PostgreSQL* por uma base de dados *MySQL*) pois a tecnologia JBDC abstrai a API de ligação à base de dados. É também indicado em [GHM$^+$] que esses sistemas implementados sobre JBDC são práticos para sistemas de processamento de eventos de complexidade pequena e média. A título de exemplo é

indicado que esse sistema consegue processar 100 eventos por segundo quando está a efetuar 20 operações de filtro, agregações, correlações e "*pattern matching*".

O uso destes sistemas de processamento de eventos pode ser útil para empresas que não queiram adquirir um novo sistema de *Complex Event Processing*, mas que já tenham conhecimento sobre sistemas de base de dados relacionais.

A tecnologia de *Complex Event Processing* pode ser utilizada em vários domínios, existindo sistemas implementados em áreas como a saúde [WRWE10, WHRN13], manufatura [RZS+12], na indústria náutica [MTC+11], na indústria de anúncios online [AGR+09] e análise de Social Media [RSS12], na integração de com sistemas de BPM[1] [SGA+12], em redes de sensores sem fio [LZLL10, MSS+06] ou na "*Semantic Web*" [SGA+12]. Como algumas dessas áreas, como a saúde, têm um forte requisito do ponto de vista de privacidade do utilizador, existem também estudos feitos sobre o impacto que esses sistemas de *Complex Event Processing* têm sobre a privacidade [HBWN11, WHRN13].

Um dos problemas que afeta a adoção de sistemas de *Complex Event Processing* tem a ver com a falta de *standards* na área, em termos de linguagens utilizadas e em sistemas de *benchmark* para testar esses sistemas. Apesar desses problemas existe algum trabalho feito na tentativa de criar *standards* para a sintaxe das várias componentes das linguagens utilizadas por esses sistemas: [ZWCC07] oferece um *standard* para a componente de deteção de padrões e [JMS+08] oferece um *standard* para a componente de processamento de *streams*.

Apesar de cada sistemas de *Complex Event Processing* oferecer as suas linguagens próprias, quase todas oferecem as mesmas funcionalidades, sendo normalmente estas linguagens implementadas como uma extensão à linguagem SQL, mantendo assim uma sintaxe familiar aos utilizadores [LWZ11].

Uma linguagem utilizada por alguns sistemas é a "*Continuous Query Language*" ou CQL. Esta é uma linguagem, introduzida em [ABB+04], é semelhante ao SQL utilizado nas bases de dados, e introduz o conceito de "*sliding windows*" [ABW06], janelas temporais que são criadas periodicamente sobre a *stream* de eventos e que a delimita em secções. Os eventos que estão nessas secções são depois utilizados para realizar operações de agregações e "*pattern matching*". O conceito de janelas temporais será visto com mais detalhe na secção 3.4.

Outro conceito importante das linguagens utilizadas em sistemas de *Complex Event Processing* é o conceito de "*Continuous Queries*" que são *queries* que nunca terminam, atualizando sempre a sua resposta quando surgem novos eventos e dando sempre a resposta mais atualizada aos utilizadores. Isto em contraste com as *queries* dos sistemas de base de dados que apenas calculam a resposta uma vez. Outra característica das "*Continuous Queries*" é que como estas vão atualizando as suas respostas em tempo real conforme vão surgindo os eventos na *stream*, estas não precisam de aceder a todos os eventos que alguma vez surgiram na *stream* (visto esta poder ser infinita) para atualizar a resposta, ao contrário das *queries* de base de dados que requerem sempre o ao mesmo conteúdo da base de dados para responder a uma *query*.

[1] Business Process Management

Para que os sistemas de *Complex Event Processing* consigam atualizar as suas respostas estes só podem fazer processamento em memória, pois só assim conseguem atingir as velocidades necessárias [SUZ05].

A figura 2.3 mostra o contraste entre a forma de processar *queries* nos sistemas de processamento de eventos (que utilizam *streams*) com os sistemas de base de dados. No caso dos sistemas de base de dados, as *queries* vão buscar a informação para dar as respostas. Isto é designado como o modelo "*Pull-based*". No caso dos sistemas de processamento de eventos, os dados vêm ter com as *queries*. Isto é designado como o modelo "*Push-based*".

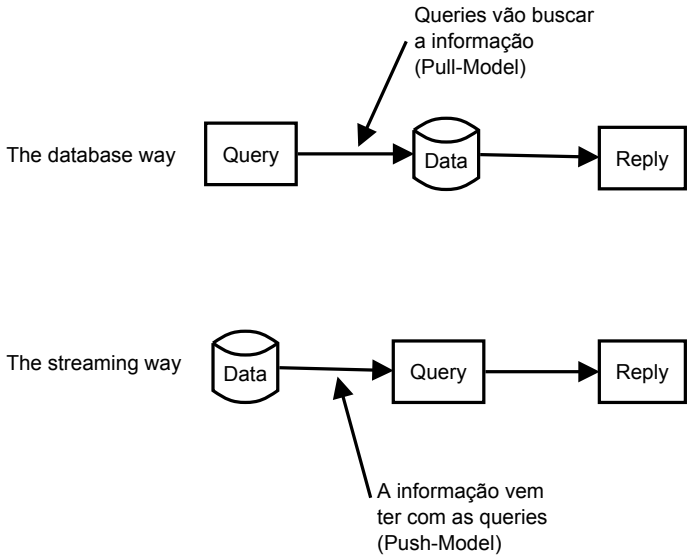

Figura 2.3: Comparação entre os modelos de processamento "*Pull*" e "*Push*". Adaptado de: [EGA]

Tal como foi visto anteriormente, o conceito de tempo está muito interligado com os sistemas de *Complex Event Processing*. Isso causa problemas em processar dados em tempo real, porque na prática os objetos que criam os eventos (por exemplo, sensores) podem ter problemas de comunicação e fazer com que os eventos cheguem com "*payloads*" errados e fora de ordem (os eventos não chegam ordenados pelo tempo) ao sistema de *Complex Event Processing* [JAF+06]. Este último causa problemas para os sistemas de processamento de eventos, obrigando a que o sistema necessite de esperar um certo tempo pela chegada de possíveis eventos que estejam atrasados. No entanto, quando mais tempo o sistema esperar, maior será a latência do sistema, o que pode remover a vantagem de realizar o processamento em temporal que os sistemas de *Complex Event Processing* fornecem. Existe portanto um "*trade-off*" entre ter uma resposta mais correta (porque espera mais tempo pelos eventos fora de ordem) e entre ter uma resposta mais rápida.

Uma forma que os sistemas de *Complex Event Processing* têm para diminuir esse problema é através de uma da introdução de um evento do tipo "*Puntuation Mark*" [TM03] (ou "*Heartbeats*" [SW04]). Estes "*Puntuation Marks*" são eventos extras, que contêm só um "*timestamp*" que são adicionados à *stream* de eventos e que indicam que não irão surgir eventos com um "*timestamp*" inferior a esse "*Puntuation Mark*" [TM03]. Caso chegue um evento com um "*timestamp*" inferior

esse evento pode ser descartado, (ou seja, não processado) ou pode ser modificado de forma a ficar com um "*timestamp*" superior ao "*Puntuation Mark*". O *StreamInsight* chama estes "*Puntuation Marks*" de CTIs e estes serão vistos com mais detalhe na secção 4.1.

Uma outra maneira de resolver problemas com as leituras de objetos reais, como por exemplo sensores que podem conter erros, por parte dos sistemas de processamento de eventos é através da criação de "*wrappers*" virtuais que expõem esses sensores ao sistema de processamento de eventos como sendo sensores ideais [JAF+05].

Para que o uso dos sistemas de *Complex Event Processing* seja o mais eficiente possível, estes devem ser colocados, do ponto de vista da arquitetura, o mais próximo possível das fontes de dados (por exemplo, dos sensores de uma máquina) como está na figura 2.4. Nestas situações o sistema de *Complex Event Processing* recebe os eventos o mais rapidamente possível e vindos diretamente dos sensores, isto é, os dados são recebidos utilizando o modelo "*Push-Based*". [Riz05].

Outra forma de utilização de um sistema de *Complex Event Processing* é colocar este a extrair informação de uma base de dados que guarda a informação vinda dos sensores. Isto implementa um modelo "*Pull-based*", isto é o sistema tem que ir buscar os eventos à base de dados. Esta forma de utilização não é recomendada porque não aproveita a velocidade de processamento dos sistemas de *Complex Event Processing*, ficando o acesso à base de dados um "*bottleneck*" da arquitetura de processamento de eventos.

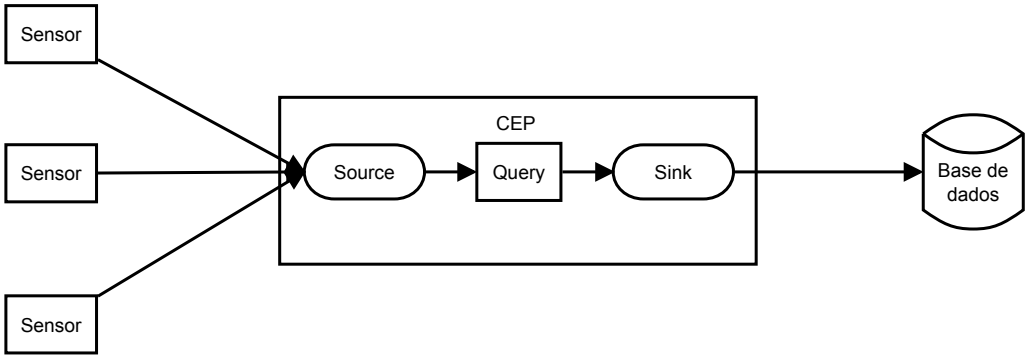

Figura 2.4: Arquitetura recomendada para a utilização de um sistema de *Complex Event Processing*.

Um dos requisitos para este trabalho era a necessidade de escolher um sistema de *Complex Event Processing* que funcionasse em ".Net". Apesar de haverem vários sistemas de *Complex Event Processing* como o *Esper*, *Nesper*, *StreamBase*, *Drools Fusion*, *StreamInsight*, só os sistemas *Nesper*, que é uma implementação do sistema *Esper* em C#, e *StreamInsight* da *Microsoft*, é que são implementados em ".Net".

Para este trabalho decidiu-se utilizar o sistema *StreamInsight* devido à sua integração com as tecnologias da *Microsoft* bem como pela sua forma de definir *queries*. Ao contrário do *Nesper*, que utiliza concatenação de *strings* para escrever as *queries* utilizando a linguagem *EPL* (*Event Processing Language*), o *StreamInsight* utiliza a tecnologia *LINQ* que permite detetar alguns erros de

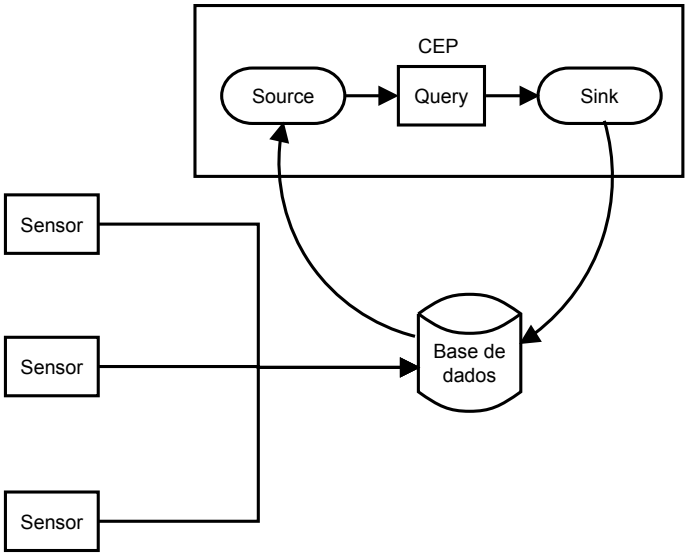

Figura 2.5: Arquitetura não recomendada para a utilização de um sistema de *Complex Event Processing*.

"*type-checking*" em "*compile-time*", bem como permitir utilizar as funcionalidades de *IntelliSense* dos IDEs como o *Visual Studio*, como se pode ver na figura 2.6.

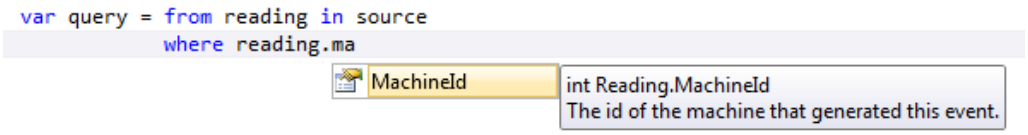

Figura 2.6: Exemplo da utilização do *IntelliSense* na construção de uma *query* no *StreamInsight*.

O fato das *queries* do *StreamInsight* terem "*type-checking*" permite evitar um problema detetado em [TGP05] que indica que há problemas na utilização de sistemas de *Complex Event Processing* devido a estes não terem um sistema que detete erros do tipo "*type-checking*" durante a construção das *queries*.

O *StreamInsight* será analisado com maior detalhe na secção 3.

Capítulo 3

Streaminsight

O *StreamInsight* é uma tecnologia desenvolvida pela *Microsoft* para a criação de sistemas de *Complex Event Processing*, tendo sido criada em 2009 [AGR+09]. Existem duas versões disponíveis: a versão *Standard* que é gratuita e a versão *Premium* que requer o uso de uma licença do *SQL Server*. A única diferença entre as duas é na taxa de eventos que conseguem processar e na tolerância de latência que apresentam [MSDa]. Para este trabalho foi utilizada a versão *Standard* do *StreamInsight*.

A versão atual do *StreamInsight* é a 2.1, tendo sido essa a versão utilizada neste trabalho. Esta disponibiliza duas arquiteturas para a criação de sistemas de *Complex Event Processing*, uma baseada em *Input Adapters* e *Output Adapters* e outra baseada na biblioteca *Reactive Extensions*, que é uma biblioteca *open source* para o processamento assíncrono de eventos. Como a partir da versão 2.1 do *StreamInsight* a arquitetura de *Input Adapters* e *Output Adapters* é considerada como sendo *legacy* [MSDk, MSDd] foi utilizado neste trabalho a arquitetura baseada na biblioteca *Reactive Extensions*.

A biblioteca *Reactive Extensions* [MSDe] fornece uma interface baseada em *Observers* e *Observables*. Esta é uma *design pattern* que permite ter um conjunto de objetos, designados de "Observers", que são notificados quando o estado interno de um outro objeto, designado de "Observable", é alterado [GHJV95].

No caso da biblioteca *Reactive Extensions*, os objetos "Observer" são aqueles que implementam a interface genérica "IObserver<T>". Esta requer a implementação de 3 métodos:

1. OnNext(T value) – Indica as ações a tomar quando o objeto recebe um evento vindo de um "Observable"

2. OnError(Exception e) – Indica as ações a tomar quando ocorre uma exceção no OnNext ou no "Observable"

3. OnComplete() – Indica as ações a tomar quando o "Observable" termina.

A título de exemplo, é mostrado aqui uma implementação de um Observer simples que recebe valores do tipo "int" e imprime para a consola aqueles que são pares:

```
class WritePairsObserver : IObserver<int>
{
    public void OnNext(int value)
    {
        if (value % 2 == 0)
        {
            Console.WriteLine(value);
        }
    }

    public void OnError(Exception e)
    {
        Console.WriteLine("Exception");
    }

    public void OnCompleted()
    {
        Console.WriteLine("Completed");
    }
}
```

De forma análoga os objetos “Observable” são aqueles que implementam a interface genérica “IObservable<T>”. Esta requer a implementação de um método “Subscribe” que associa um objeto do tipo “IObserver” a si e devolve um objeto do tipo “IDisposable” que pode ser usado no futuro para dissociar o “IObserver” ao “IObservable”.

Segue-se uma implementação de um “Observable” simples que gera números aleatórios de 1 em 1 segundos e envia estes para os seus “Observer”. Para simplificar o exemplo não é mostrado o código que implementa o “IDisposable”. Esta última interface requer a implementação de um método “Dispose()” que permite libertar, de forma segura, recursos que tenham sido alocados durante o tempo de vida do “IObservable” [MSDc].

```
class RandomNumberObservable : IObservable<int>
{
    private Random randomGenerator;
    private Thread workThread;
    private List<IObserver<int>> subscribers;

    public RandomNumberObservable()
    {
        subscribers = new List<IObserver<int>>();
        randomGenerator = new Random();
        workThread = new Thread(() =>
        {
            // Generate a random number and send them to the subscribers
            while (true)
            {
```

```
16                 var randomNumber = randomGenerator.Next();
17
18                 foreach (var observer in subscribers)
19                 {
20                     observer.OnNext(randomNumber);
21                 }
22
23                 // Wait for one second
24                 Thread.Sleep(1000);
25             }
26         });
27
28         workThread.Start();
29     }
30
31     public IDisposable Subscribe(IObserver<int> observer)
32     {
33         subscribers.Add(observer);
34         return null;
35     }
36 }
```

A biblioteca *Reactive Extensions* também fornece a interface genérica "ISubject<TInput, TOutput>". Esta interface indica que o objeto pode funciona quer como um "Observer", que como um "Observable", isto é, implementa as interfaces "IObservable" e "IObserver".

O diagrama de classes das interfaces fornecidas pela biblioteca *Reactive Extensions* pode ser vista na figura 3.1.

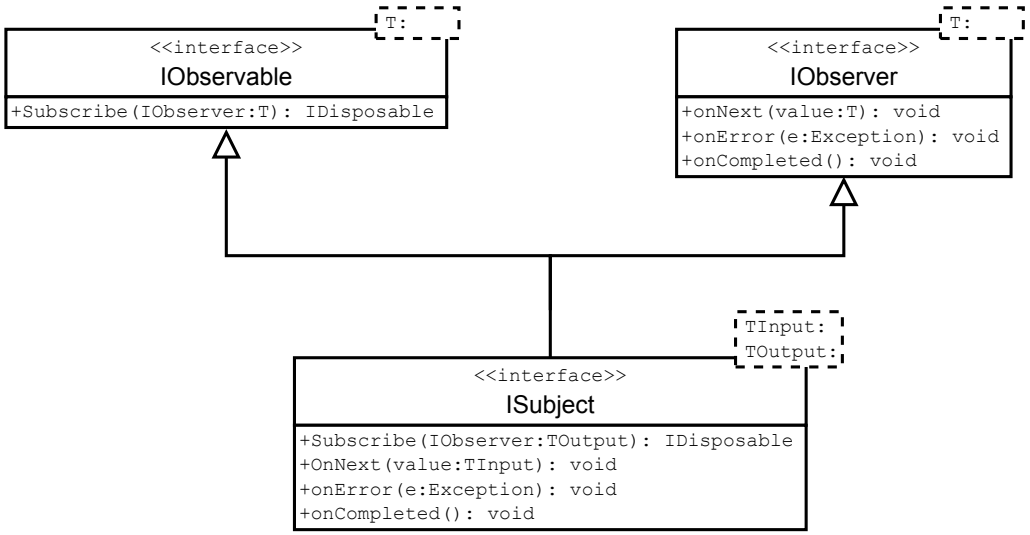

Figura 3.1: Diagrama de classes UML das classes fornecidas pela biblioteca *Reactive Extensions*.

Quando o *StreamInsight* está a utilizar a arquitetura baseada na biblioteca *Reactive Extensions*, este considera os "IObservable" como sendo a fonte de eventos, sendo estes designados por "*sour-*

ces", e os "IObserver" como sendo o destino, designados por "*sinks*", desses eventos. Como os objetos "ISubject" implementam ambas interfaces, uma classe que implemente a interface "ISubject" pode ser simultaneamente um "*sink*" e um "*source*".

Para o *StreamInsight* os eventos têm origem num "*source*" e são enviadas para um ou mais "*sinks*", sendo utilizadas "*queries*" escritas em LINQ para definir o caminho que os eventos seguem dos *sources* até aos *sinks* [MSDj]. Essas "*queries*" podem realizar várias operações como filtrar eventos através dos campos do *payload*, projetar campos dos *payload*, alterar a duração dos eventos, etc. É possível que existam eventos que não consigam aceder a nenhum "*sink*" porque não satisfazem alguma condição de filtro de uma "*query*". Nesse caso esses eventos são descartados.

O *StreamInsight* considera que todos os eventos existem numa *stream* temporal, tendo cada evento, em adição ao seu *payload*, um tempo de início e um tempo de fim. É portanto necessário inserir na posição correta um evento que surge num "*source*" numa *stream*. Só assim é que o *StreamInsight* poderá aplicar *queries* sobre esse evento.

Isto é feito utilizando uma família de funções designada de "To*Streamable" que converte uma "*source*" numa *stream*, fazendo com que quando um evento surja na "*source*" este seja automaticamente inserido numa *stream*. A razão pela qual há um conjunto diferente de funções que inserem um evento numa *stream* é que estas indicam o modelo do evento. O *StreamInsight* modela 3 tipos de eventos:

1. *Point Events* – Eventos que só têm um *Start Time*, existindo apenas num instante de tempo. São adicionados a uma *stream* utilizando a função "ToPointStreamable"

2. *Interval Events* – Eventos que têm um *Start Time* e um End Time que indicam em que intervalo de tempo são válidos. São adicionados à *stream* utilizando a função "ToIntervalStreamable"

3. *Edge Events* – Eventos que têm um *Start Time* mas que inicialmente não se conhece qual o seu *End Time*, sendo esse determinado depois no futuro. Isto faz com que existam dois tipos de *Edge Events*: os *Start* e os *End*. Ambos são adicionados à *stream* utilizando a função "ToEdgeStreamable".

A representação destes modelos pode ser vista na figura 3.2.

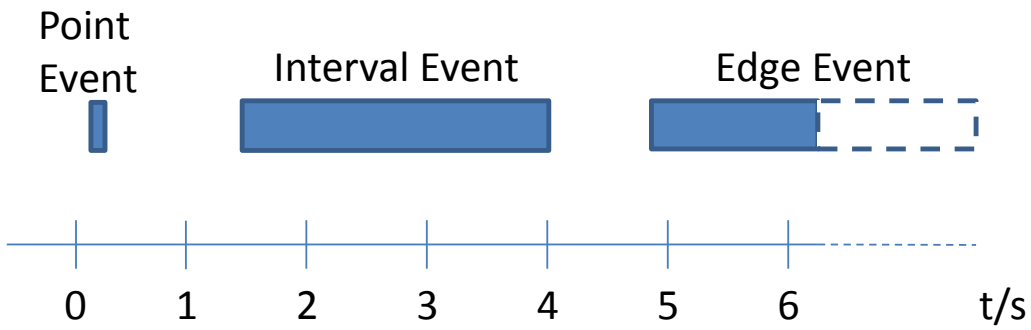

Figura 3.2: Os diferentes modelos de eventos fornecidos pelo *StreamInsight*.

Apesar de o *StreamInsight* fornecer estes modelos para modelar eventos, internamente ele considera tudo como sendo *Interval Events* [RJA12]. Para transformar os outros tipos de eventos em *Interval Events* é feita a seguinte conversão:

- *Point Events* – Convertidos em *Interval Events* com o mesmo *Start Time* e com o *End Time* igual ao *Start Time* + 1 *tick* do relógio
- *Edge Events* – Se for um evento do tipo *Start*, então este é convertido num *Interval Events* com o mesmo *Start Time* e com o *End Time* igual a "DateTime.MaxValue". Caso seja um evento do tipo *End*, então equivalente a um *Interval Event*.

Pode ser ver um exemplo de uma arquitetura do *StreamInsight* na figura 3.3 que mostra uma arquitetura que contêm um gerador de números aleatórios como "*source*" e que envia os números pares para um ficheiro.

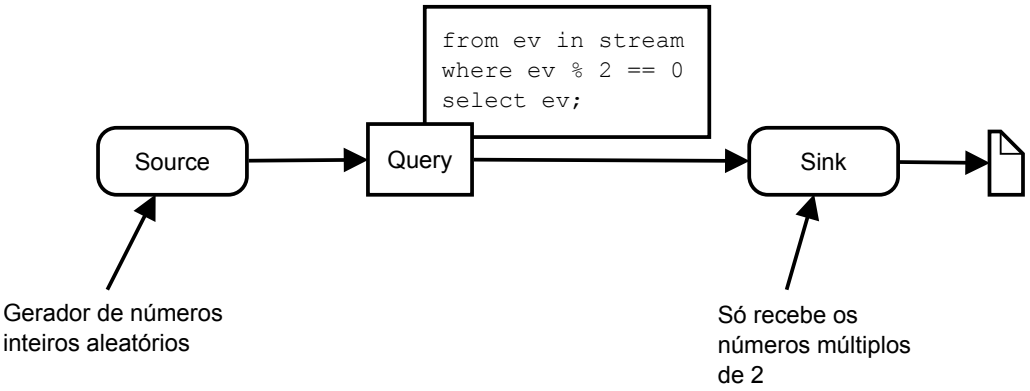

Figura 3.3: Arquitetura do *StreamInsight* que escreve números pares num ficheiro.

O código que implementa essa arquitetura no *StreamInsight* é:

```
1  using (var server = Server.Create("instance-name"))
2  {
3      var myApp = server.CreateApplication("output-to-files experiment");
4
5      var random = new Random();
6
7      // Create the source
8      var source = myApp.DefineObservable(
9          () => Observable.Generate(random.Next(),
10                                   _ => true,
11                                   _ => random.Next(),
12                                   _ => random.Next(),
13                                   _ => TimeSpan.FromSeconds(1)));
14
15     // Convert it to a stream
16     var stream = source.ToPointStreamable(
17         number => PointEvent.CreateInsert(DateTimeOffset.Now, number),
```

```
18          AdvanceTimeSettings.IncreasingStartTime);
19
20      // Create different queries over the stream
21      var queryPairs = from ev in stream
22                       where ev % 2 == 0
23                       select ev;
24
25      // Create the sink
26      var fileSink = myApp.DefineObserver(
27          (string filename) => Observer.Create<int>(
28              payload =>
29              File.AppendAllText("c:/Users/re-pereira/Documents/" + filename,
30                                 payload.ToString() + "\r\n")));
31
32      // Actually start running the queries
33      using (queryPairs.Bind(fileSink("output-pairs-numbers.txt"))
34                       .Run("pairs numbers query"))
35      {
36          Console.WriteLine("StreamInsight is now running");
37          Console.WriteLine("Press any key to stop");
38          Console.ReadKey();
39      }
40  }
```

Após correr este programa o ficheiro "output-pair-numbers.txt" contêm o seguinte *output*:

```
1018937924
1774143626
1210365538
368237368
332067872
```

Tal como já foi dito anteriormente, é possível que um evento vá para múltiplos *sinks*. Para demonstrar isso pode-se modificar a arquitetura da figura 3.3 para adicionar uma outra *query* que extrai da *stream* os números múltiplos de 4. Esta arquitetura pode ser vista na figura 3.4 e o seu código é:

```
1   using (var server = Server.Create("instance-name"))
2   {
3       var myApp = server.CreateApplication("output-to-files experiment");
4
5       var random = new Random();
6
7       // Create the source
8       var source = myApp.DefineObservable(
9           () => Observable.Generate(random.Next(),
10                                    _ => true,
```

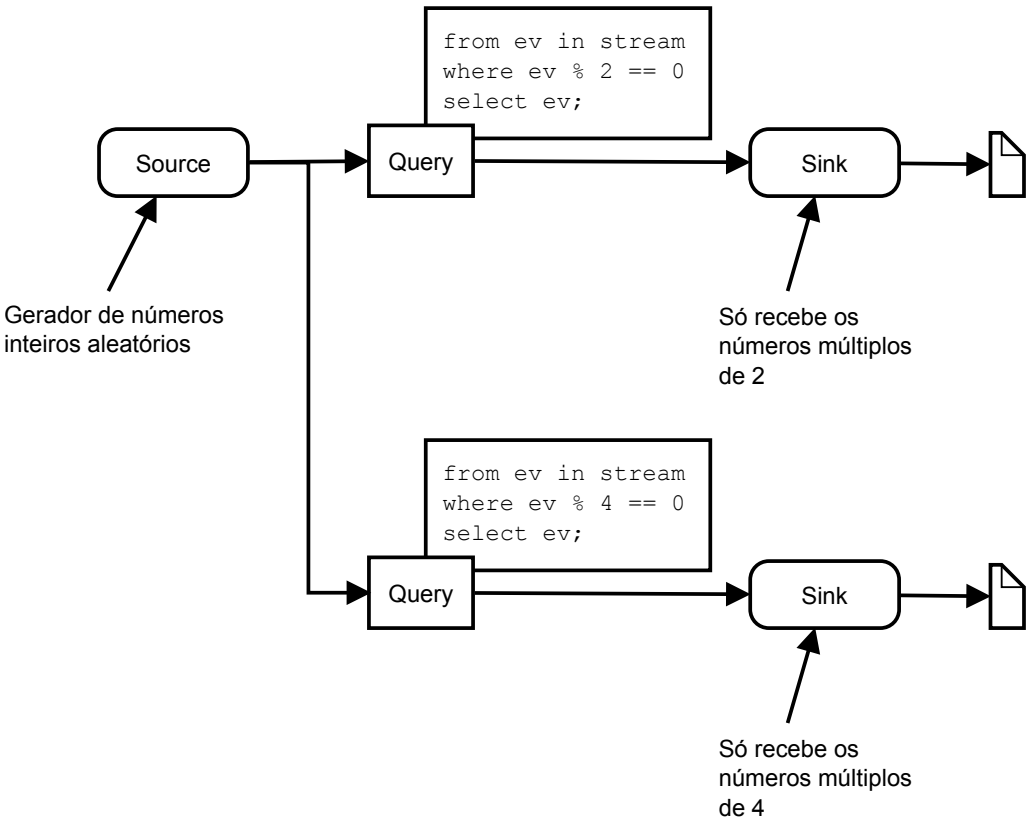

Figura 3.4: Exemplo de uma arquitetura do *StreamInsight* com vários "*sinks*".

```
11                                  _ => random.Next(),
12                                  _ => random.Next(),
13                                  _ => TimeSpan.FromSeconds(1)));
14
15      // Convert it to a stream
16      var stream = source.ToPointStreamable(
17          number => PointEvent.CreateInsert(DateTimeOffset.Now, number),
18          AdvanceTimeSettings.IncreasingStartTime);
19
20      // Create different queries over the stream
21      var queryMul2 = from ev in stream
22                      where ev % 2 == 0
23                      select ev;
24
25      var queryMul4 = from ev in stream
26                      where ev % 4 == 0
27                      select ev;
28
29      // Create the sink
30      var fileSink = myApp.DefineObserver(
31          (string filename) => Observer.Create<int>(
32              payload =>
33              File.AppendAllText("c:/Users/re-pereira/Documents/" + filename,
```

```
34                                  payload.ToString() + "\r\n")));
35
36      // Actually start running the queries
37      using (queryMul2.Bind(fileSink("output-mul2-numbers.txt"))
38                      .Run("mul2 numbers query"))
39      {
40          using (queryMul4.Bind(fileSink("output-mul4-numbers.txt"))
41                          .Run("mul4 numbers query"))
42          {
43              Console.WriteLine("StreamInsight is now running");
44              Console.ReadKey();
45          }
46      }
47  }
```

A após correr esta *query* o conteúdo do ficheiro "output-mul2-numbers.txt" é:

```
1018937924
1774143626
1210365538
368237368
332067872
```

e o conteúdo do ficheiro "output-mul4-numbers.txt" é:

```
1018937924
368237368
332067872
187283924
2088938988
```

Pode-se ver que, por exemplo, o número 1018937924 aparece nos dois ficheiros porque este é divisível por 2 e por 4.

3.1 LINQ

Tal como já foi descrito anteriormente, o *StreamInsight* utiliza a tecnologia LINQ [MSDj]. Isto permite escrever as *queries* utilizando um "*type-system*" que consegue detetar, sem necessidade de executar o programa, algumas classes de erros.

A tecnologia LINQ não é especifica ao *StreamInsight*, existindo várias implementações de LINQ para outro tipo de domínios para além da escrita de *queries* sobre *streams* (por exemplo o LINQ2XML que descreve como converter um conjunto de dados numa expressão de XML).

A escrita de uma expressão de LINQ pode ser feita utilizando duas sintaxes [Mic10]:

- *Query Expression* – Permite escrever uma *query* utilizando uma sintaxe semelhante ao SQL

- *Method Chain* – Permite escrever uma *query* utilizando a combinação de função para criar a *query*. Internamente, o compilador de C# transforma as expressões de LINQ do tipo "*Query Expression*" numa expressão do tipo "*Method Chain*" [Mic10].

Segue-se um exemplo de uma expressão LINQ escrita no formato *Query Expression*:

```
var query = from value in array
            where value > 10
            select value * 2;
```

devolve uma lista com os elementos da variável "*array*" maiores que 10. Esses elementos são depois multiplicados por 2 e guardados na variável "*query*".

A mesma expressão escrita no formato "*Method Chain*" seria:

```
var query = array.Where(value => value > 10)
                 .Select(value => value * 2);
```

Neste trabalho foi utilizado a versão "*Method Chain*" do LINQ devido às funcionalidades de composição que este fornece. Esta permite que uma *query* possa ser facilmente modificada dependendo do valor de outras variáveis. Por exemplo, o seguinte código modifica o cálculo da variável "*query*" dependendo do valor da variável "*var1*".

```
var query = array.Where(value => value > 10);

switch (var1)
{
case "foo":
    query = query.Select(value => value * 3);
    break;
case "bar":
    query = query.Select(value => value * 5);
    break;
}
```

Tal expressão não é possível escrever utilizando a sintaxe "*Query Expression*", isto é, não é possível escrever este código:

```
var query = from value in array
            where value > 10;

switch (var1)
{
case "foo":
    query = select value * 3;
```

```
8       break;
9   case "bar":
10      query = select value * 5;
11      break;
12  }
```

A única forma de escrever esta *query* é:

```
1   // Tem que se declarar a query para se poder aceder-la
2   // fora do switch
3   List<int> query = null;
4
5   switch (var1)
6   {
7   case "foo":
8       query = from value in array
9               where value > 10
10              select value * 3;
11      break;
12  case "bar":
13      query = from value in array
14              where value > 10
15              select value * 5;
16      break;
17  }
```

O que causa a repetição de código nos dois *branches* da expressão "switch".

3.2 Funcionalidade de extensão

Por defeito o *StreamInsight* só fornece 5 operadores de agregação: *Avg*, *Count*, *Max*, *Min* e *Sum*. Isso torna-o insuficiente para muitas aplicações. No entanto, o *StreamInsight* fornece várias funcionalidades de extensão, permitindo aos seus utilizadores definir novos operadores que podem depois ser utilizados na escrita das *queries* do LINQ.

Estas novas funcionalidades são:

- UDF - *User Defined Function* [MSDh]
- UDO - *User Defined Operator* [MSDg]
- UDA - *User Defined Aggregate* [MSDg]
- UDSO - *User Defined Stream Operator* [MSDi]

Uma UDF é uma função normal escrita em C# que pode ser utilizada em expressões de LINQ para fazer filtros e projeções. Por exemplo, uma UDF que deteta os números pares seria:

```
public static bool IsPairNumber(int number)
{
    return number % 2 == 0;
}
```

Que depois seria utilizado da seguinte forma numa *query*:

```
query = from reading in stream
        where IsPairNumber(reading)
        select reading;
```

De notar que uma UDF tem de ser utilizada diretamente nos eventos da *stream*, não podendo ser utilizada em eventos que já estejam inseridos numa janela temporal. Por outras palavras, o código que se segue não é válido:

```
query = from reading in stream.TumblingWindow(TimeSpan.FromSeconds(10))
        where isPairNumber(reading)
        select reading;
```

Para se poder aplicar uma função aos elementos que já estão numa janela temporal é necessário utilizar um UDO ou um UDA. A única diferença entre os dois é que o UDO pode devolver um ou mais eventos, enquanto que o UDA só pode devolver um evento.

A definição de um UDO ou UDA é dividida em duas partes. Primeiro é necessário implementar o UDO ou UDA numa classe que implementa, respetivamente, as interfaces "CepOperator" e "CepAggregate".

Uma implementação de um UDO que devolve os números pares de uma janela temporal seria:

```
// Operador que recebe e devolve "ints"
public class GetPairs : CepOperator<int, int>
{
    public override IEnumerable<int> GenerateOutput(IEnumerable<int> payloads)
    {
        return payloads.Where(payload => payload % 2 == 0);
    }
}
```

Listing 3.1: Definição de um UDO que devolve os números pares de uma janela

Uma vez definido UDO ou UDA é necessário indicar o motor do LINQ que este pode utilizar-lo. Isso é feito através da criação de um método estático com o atributo "CepUserDefinedOperator", caso se esteja a definir um UDO, ou com o atributo "CepUserDefinedAggregate" caso se

esteja a definir um UDA. O corpo deste método estático nunca é executado por isso não é importante o seu conteúdo. Para evitar executar acidentalmente o corpo do método o *StreamInsight* fornece a *Exception* "CepUtility.DoNotCall()" que deve ser usada no corpo destes métodos:

```
[CepUserDefinedOperator(typeof(GetPairs))]
public static int GetPairs(this CepWindow<int> window)
{
    throw CepUtility.DoNotCall();
}
```

Existe também a possibilidade de definir UDOs e UDAs, que recebem informação acerca da janela temporal onde os eventos se inserem, como o *timestamp* de início e de fim da mesma bem como os *timestamps* dos eventos. Para tal basta modificar o UDO de forma a implementar a interface "CepTimeSensitiveOperator" em vez da interface "CepOperator". Deve-se proceder de forma análoga para os UDAs, implementando a interface "CepTimeSensitiveAggregate" em vez da interface "CepAggregate". Por exemplo, para modificar o UDO "GetPairs" da listagem 3.1 de maneira a só devolver os números pares cujos eventos tenham ocorrido à segunda-feira seria necessário utilizar o seguinte código:

```
public class GetPairs : CepTimeSensitiveOperator<int, int>
{

    public override IEnumerable<IntervalEvent<int>>
            GenerateOutput(IEnumerable<IntervalEvent<int>> events,
                           WindowDescriptor windowDescriptor)
    {
        return events.Where(e => e.StartTime.DayOfWeek == 1)
                     .Where(e => e.Payload % 2 == 0);
    }
}
```

Um fator importante em relação aos "CepTimeSensitiveOperators" e "CepTimeSensitiveAggretate" é que eles recebem sempre os eventos no formato "*Interval Event*", mesmo que eles tenham sido inseridos na *stream* como "*Point Event*" ou "*Edge Event*".

Tendo finalmente definidos os UDOs ou UDAs, estes podem ser utilizados como métodos das janelas temporais:

```
var query = from window in stream.TumblingWindow(TimeSpan.FromSeconds(10))
            select window.GetPairs();
```

Um problema que os UDFs, UDOs e UDAs sofrem é que as funções que implementam, não podem guardar nenhum estado internamente. Por isso caso se queira guardar utilizar um operador

que permita guardar estado interno deve-se utilizar um UDSO.

Ao contrário dos UDFs, que são aplicados aos eventos, dos UDOs e UDAs, que são aplicadas às janelas temporais, os UDSOs são aplicados diretamente à *stream* de dados.

Tal como um UDOs, os UDSO pode devolver um ou mais eventos para a *stream*.

Para implementar um UDSO é necessário implementar a interface "CepPointStreamOperator" ou "CepEdgeStreamOperator" dependendo de como os eventos de entrada devem ser modelados. No caso dos eventos de entrada serem do tipo "*Interval Events*" então só são utilizados os seus "*Start Time*".

Por exemplo, para implementar um UDSO que devolve os números da *stream* que surjam imediatamente a seguir a um número par seria utilizado o seguinte código:

```
1  // O UDSO tem de ser serializable e recebe e devolve
2  // payloads do tipo "int"
3  [Serializable]
4  public class GetPairs : CepPointStreamOperator<int, int>
5  {
6      private int lastPayload;
7
8      public GetPairs() {}
9
10     public override IEnumerable<int> ProcessEvent(PointEvent<int> inputEvent)
11     {
12         // Ao contrario dos UDFs, UDOs e UDAs, os UDSO
13         // recebem tambem os eventos CTI
14         if (inputEvent.EventKind == EventKind.Insert)
15         {
16             // Retirar o payload do PointEvent
17             int currentPayload = inputEvent.Payload;
18
19             if (lastPayload == null)
20             {
21                 lastPayload = currentPayload;
22             }
23             else if (lastPayload % 2 == 0)
24             {
25                 lastPayload = null;
26
27                 // Devolve o payload atual
28                 yield return currentPayload;
29             }
30         }
31     }
32
33     public override bool IsEmpty
34     {
35         get { return false; }
```

```
36      }
37  }
```

Para implementar o *CmStream* foi necessário recorrer ao uso de UDOs, UDAs e UDSOs.

3.3 União de streams no StreamInsight

Consideremos as seguintes duas tabelas numa base de dados:

Id	Payload1
1	10
2	25

Id	Payload2
1	15
2	30

Numa base de dados é possível unir estas duas tabelas através do campo "Id" utilizando a seguinte *query*:

```
1  SELECT tabela1.id, payload1, payload2 FROM tabela1 JOIN tabela2
2      ON tabela1.id = tabela2.id;
```

O resultado dessa *query* é a seguinte tabela:

Id	Payload1	Payload2
1	10	15
1	10	30

Como já foi dito anteriormente a utilização do *StreamInsight* adiciona a dimensão do tempo às *queries*. Por isso a junção de duas *streams* é feita não só ao nível dos campos do *payload*, como também ao nível do tempo. Isto faz com que os eventos de duas *streams* só sejam juntados se sobreporem uns aos outros.

Portanto, dadas as *streams* da figura 3.5, a seguinte *query* do *StreamInsight*:

```
1  from payload1 in stream1
2  from payload2 in stream2
3  select new Event {
4      payload1 = payload1
5      payload2 = payload2
6  };
```

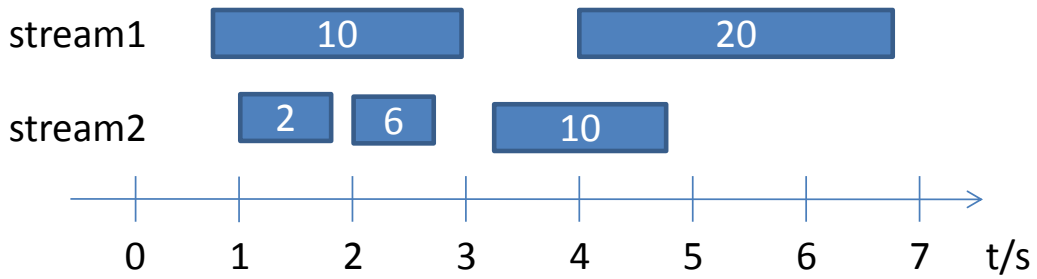

Figura 3.5: Exemplo de duas *streams* que se intersectam no tempo.

Gera a *stream* que pode ser vista na figura 3.6.

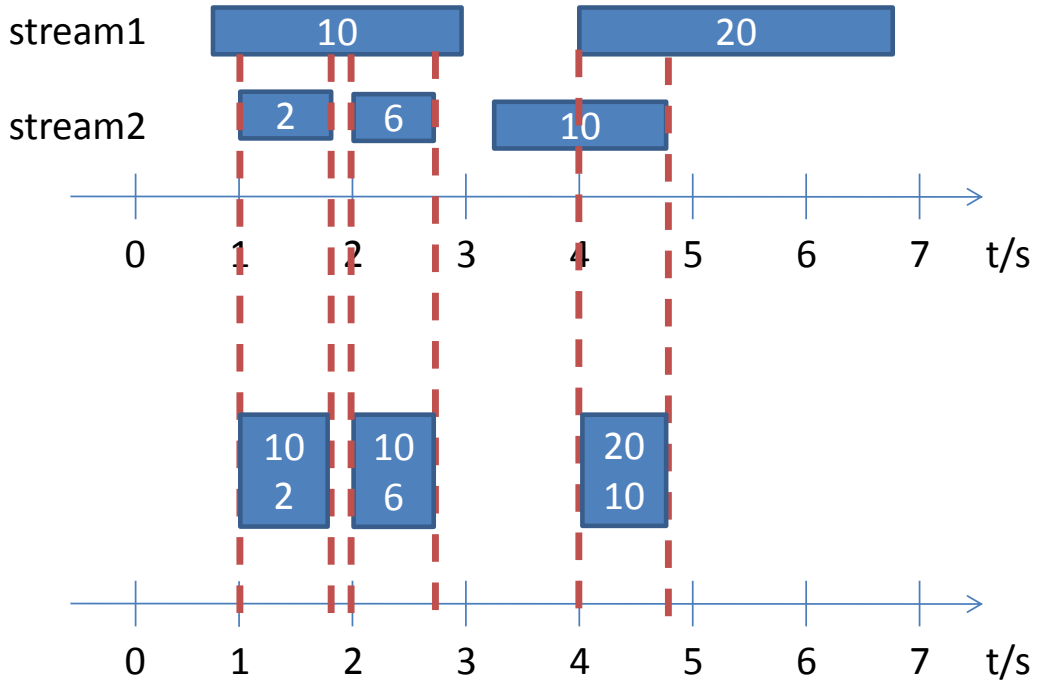

Figura 3.6: Resultado da união no tempo das duas *streams* da figura 3.5.

Tal como um "join" da base de dados permite adicionar restrições para a junção das tabelas através da *keyword* "ON", também o *StreamInsight* permite adicionar restrições à união das *streams*, utilizando a *keyword* "Where":

```
1 var query = from payload1 in stream1
2             from payload2 in stream2
3             where payload1.Id == payload2.Id
4             select new {
5                 id = payload1.Id,
6                 payload1 = payload1.Payload,
7                 payload2 = payload2.Payload,
8             };
```

3.4 Operações de agregação sobre streams

Consideremos a seguinte tabela existente numa base de dados:

Id	Payload
1	10
2	25

Existindo esta tabela na base de dados é fácil determinar quantos elementos ela contém através do uso da operação de agregação "Count()" como na *query* seguinte:

```
1   SELECT COUNT(*) FROM TABELA;
```

No entanto, já foi dito que as *streams* de eventos podem ser infinitas. Assim, calcular a resposta para esta *query*[1]:

```
1   SELECT COUNT(*) FROM STREAM;
```

não é simples pois não se conhece à priori o número de eventos na *stream*.

A maneira que os sistemas de *Complex Event Processing* utilizam para fazer estas agregações é através do uso de janelas temporais, isto é, subdivide-se a *stream* infinita em partes mais pequenas (e finitas) e aplica-se o operador de agregação aos eventos que estão em cada uma dessas janelas [GHM$^+$].

No caso do *StreamInsight* este fornece 4 tipos de janelas temporais diferentes:

1. *Hopping Window*
2. *Tumbling Window*
3. *Count Window*
4. *Snapshot Window*

A *Hopping Window*, designada normalmente na literatura por "*Sliding Window*", permite gerar janelas temporais de tamanho constante e que se deslocam no tempo a um intervalo constante. Um exemplo do seu comportamento pode ser visto na figura 3.7.

A *Tumbling Window* é um caso especial da *hopping window* onde o intervalo de deslocação é igual ao tamanho da janela. Um exemplo do seu comportamento pode ser visto na figura 3.8.

A *Count Window* permite gerar janelas que contêm um dado número de elementos. Caso exista vários eventos que ocorram ao mesmo tempo, o *StreamInsight* considera isso como sendo só um elemento, como se pode ver na figura 3.9. Desta forma o número de elementos existentes numa *Count Window* pode ser superior (mas nunca inferior) ao número de elementos pedidos.

[1] Esta *query* está escrita numa sintaxe não existente, serve só para exemplo

```
FROM window IN stream.HoppingWindow(TimeSpan.FromSeconds(3), TimeSpan.FromSeconds(2))
SELECT window.Avg(e => e.payload);
```

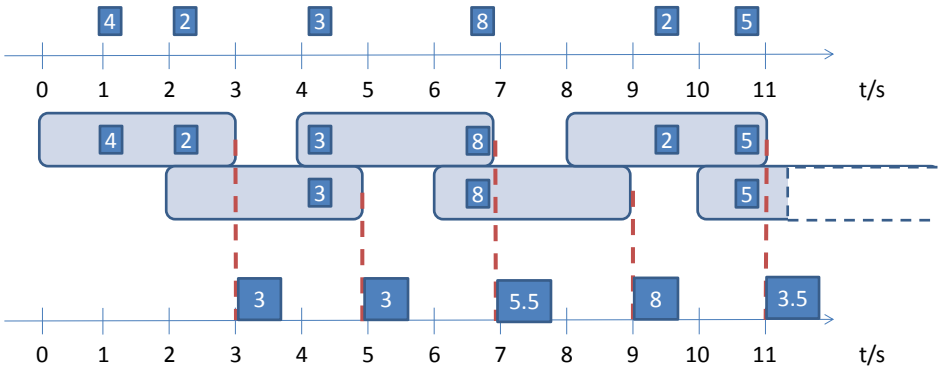

Figura 3.7: Exemplo de uma agregação usando uma *Hopping Window*.

A *Snapshot Window* cria janelas sempre que um evento entra ou sai da *stream* de dados. Um exemplo do seu comportamento pode ser visto na figura 3.10.

O *CmStream* implementou também um tipo de novo de janelas chamadas *Session Windows*. Estas encontram-se descritas em 4.5.

Após criar estas janelas é possível aplicar um operador, como por exemplo, o operador *Avg()* que recebe uma lista com os eventos de uma dada janela.

Como se pode ver nas figuras 3.7, 3.8, 3.9 e 3.10, o resultado final da agregação é uma nova *stream* onde para cada janela, existe no final da mesma, um *PointEvent* que contêm a resposta da aplicação do operador aos eventos dessa janela.

```
FROM window IN stream.TumblingWindow(TimeSpan.FromSeconds(5))
SELECT window.Avg(e => e.payload);
```

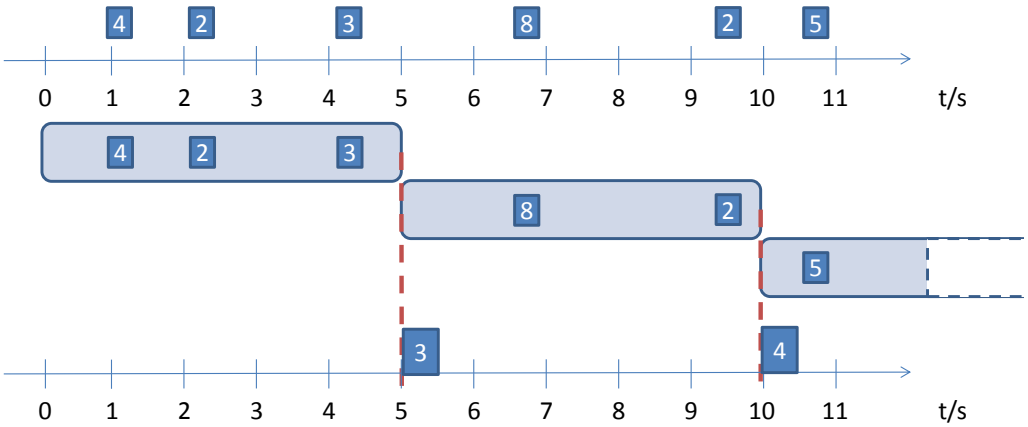

Figura 3.8: Exemplo de uma agregação usando uma *Tumbling Window*.

```
FROM window IN stream.CountWindow(3)
SELECT window.Avg(e => e.payload);
```

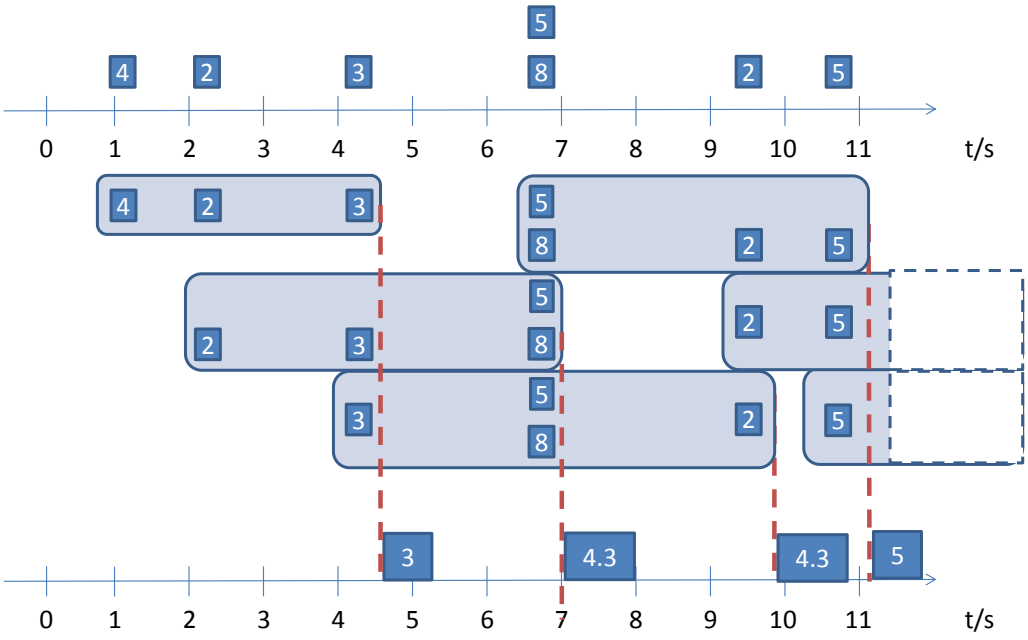

Figura 3.9: Exemplo de uma agregação usando uma *Count Window*.

```
FROM window IN stream.SnapshotWindow()
SELECT window.Avg(e => e.payload);
```

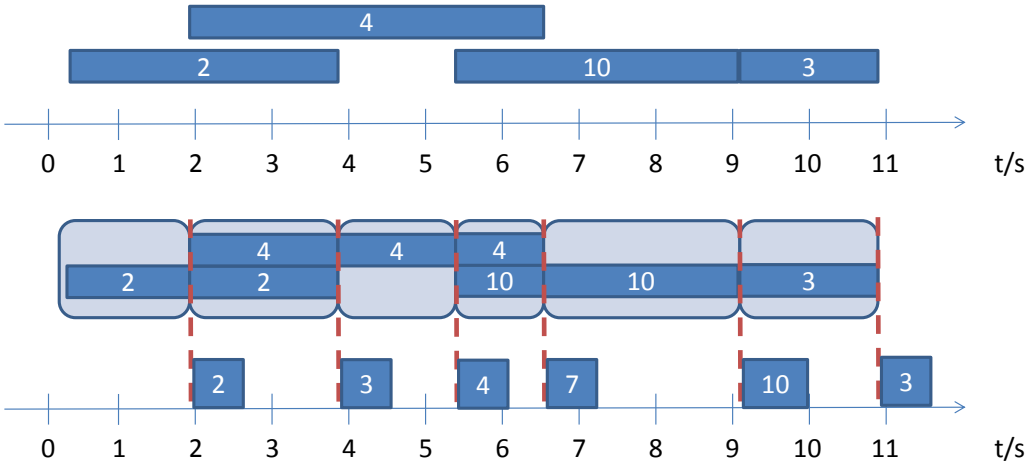

Figura 3.10: Exemplo de uma agregação usando uma *Snapshot Window*.

Capítulo 4

CmStream

4.1 Forma de gerar CTI

As *queries* do *StreamInsight* funcionam exclusivamente pelo "*Application Time*" e não pelo tempo do relógio do computador onde o *StreamInsight* se encontra a correr.

Isso faz com que quando o *StreamInsight* recebe eventos ele guarda sempre o último elemento em memória, isto é, não faz *flush* do evento. Isto acontece porque como o *StreamInsight* utiliza o conceito de "*Application Time*", ele não sabe que o tempo avançou no "Mundo Real".

A forma que o *Streaminsight* tem de avançar o tempo das *stream* é através da emissão de eventos CTI. Um evento CTI é uma garantia de que não irão aparecer eventos novos com um *timestamp* mais antigo que o *timestamp* deste.

Como os eventos CTI são produzidos quando são introduzidos eventos do tipo *Insert* na *stream* e são baseados nos *timestamps* destes últimos, não é possível criar por defeito eventos CTIs na *stream*.

De notar que, como se pode ver na figura 4.1 cada *stream* tem o seu próprio CTI, ou seja, o "tempo" interno das *streams* são independente umas das outras.

Na arquitetura antiga do *StreamInsight*, baseada em *Input* e *Output Adapters* era possível importar os CTIs de uma outra *stream* que estive a correr. Isto permitia que as *stream* pudessem ser sincronizadas. Caso contrário quando se faz um *join* de duas ou mais *stream* estas sincronizam o *output* com a *stream* mais lenta, isto é, a *query* só avança quando todas as *stream* tiverem emitido pelo menos um CTI. Isto causa problemas quando temos uma *stream* com baixa latência a fazer *join* a outra *stream* com alta latência (por exemplo estamos a comparar leituras vindas diretamente de sensores com dados de referência vindos de uma base de dados) porque o *StreamInsight* guardará em memória os eventos todos da *stream* com baixa latência até receber um evento CTI na *stream* com alta latência. Pode-se ver um exemplo disso na figura 4.2. A importação de CTIs permitia resolver isso através, tal como o nome indica, da transferência dos CTIs da *stream* com baixa latência para a *stream* com alta latência.

No entanto, a nova arquitetura do *StreamInsight* não permite fazer a importação de CTIs de uma *stream* para a outra.

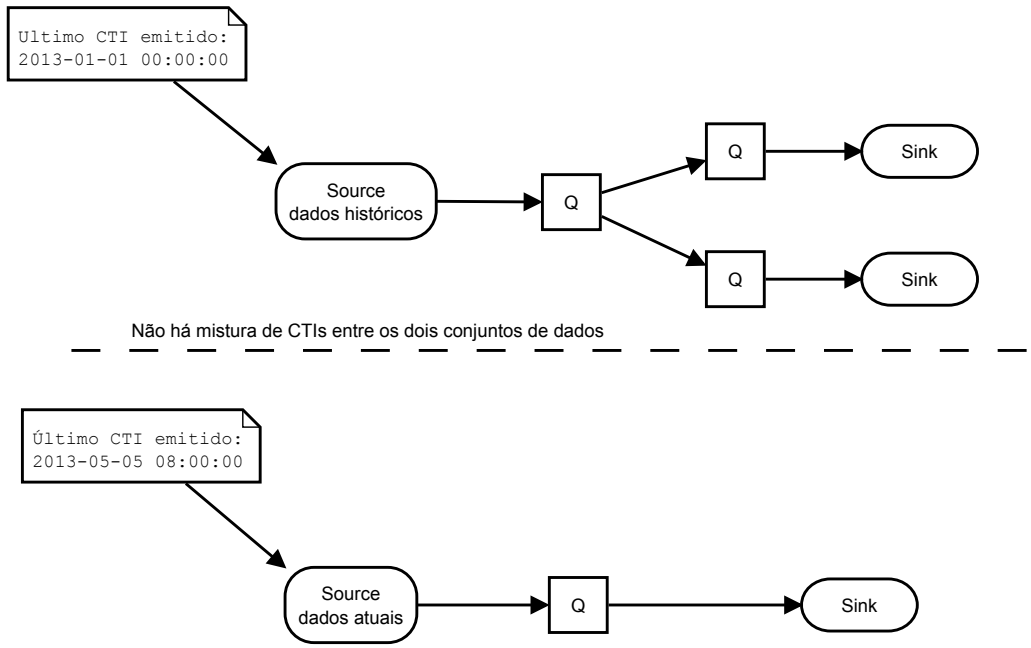

Figura 4.1: Exemplo de uma arquitetura de *StreamInsight* onde as *streams* não se intersectam. Os eventos CTIs das mesmas são independentes. Os blocos "Q" correspondem a *queries* escritas utilizando o LINQ.

Foi por isso implementada no *CmStream* uma forma de injetar CTIs através de um "*Timer*" numa *stream*, forçando-a a avançar no tempo e a fazer *flush* de qualquer evento que nela estivesse. No entanto a utilização deste sistema, designado de "*RealTime Query*", faz com que a *stream* fique com uma habilidade reduzida de processar eventos em fora de ordem. Essa funcionalidade encontra-se descrita na secção 4.4.

Os eventos CTIs servem de "*Puntuation Marks*" nas *streams* do *StreamInsight*, indicando a este que não irá surgir mais nenhum evento com um *timestamp* de início menor do que o *timestamp* desse evento CTI.

No entanto, no mundo real os eventos podem por vezes chegar eventos "atrasados" ao sistema, isto é, podem surgir eventos que têm um *timestamp* de início menor que o do atual evento CTI da *stream*.

Nestas situações o *StreamInsight* permite escolher dois comportamentos, que são escolhidos durante a chamada às funções "To*Streamable":

- *Drop* – Faz com que o *StreamInsight* ignore completamente o evento que chegou atrasado
- *Adjust* – Faz com que o *StreamInsight* modifique o *timestamp* de início do evento de forma a que este fique igual ao *timestamp* do actual CTI da *stream*.

A figura 4.3 mostra o que acontece quando um evento com um *timestamp* de início menor que o *timestamp* do evento CTI chega a uma *stream* para os diferentes comportamentos fornecidos pelo *StreamInsight*.

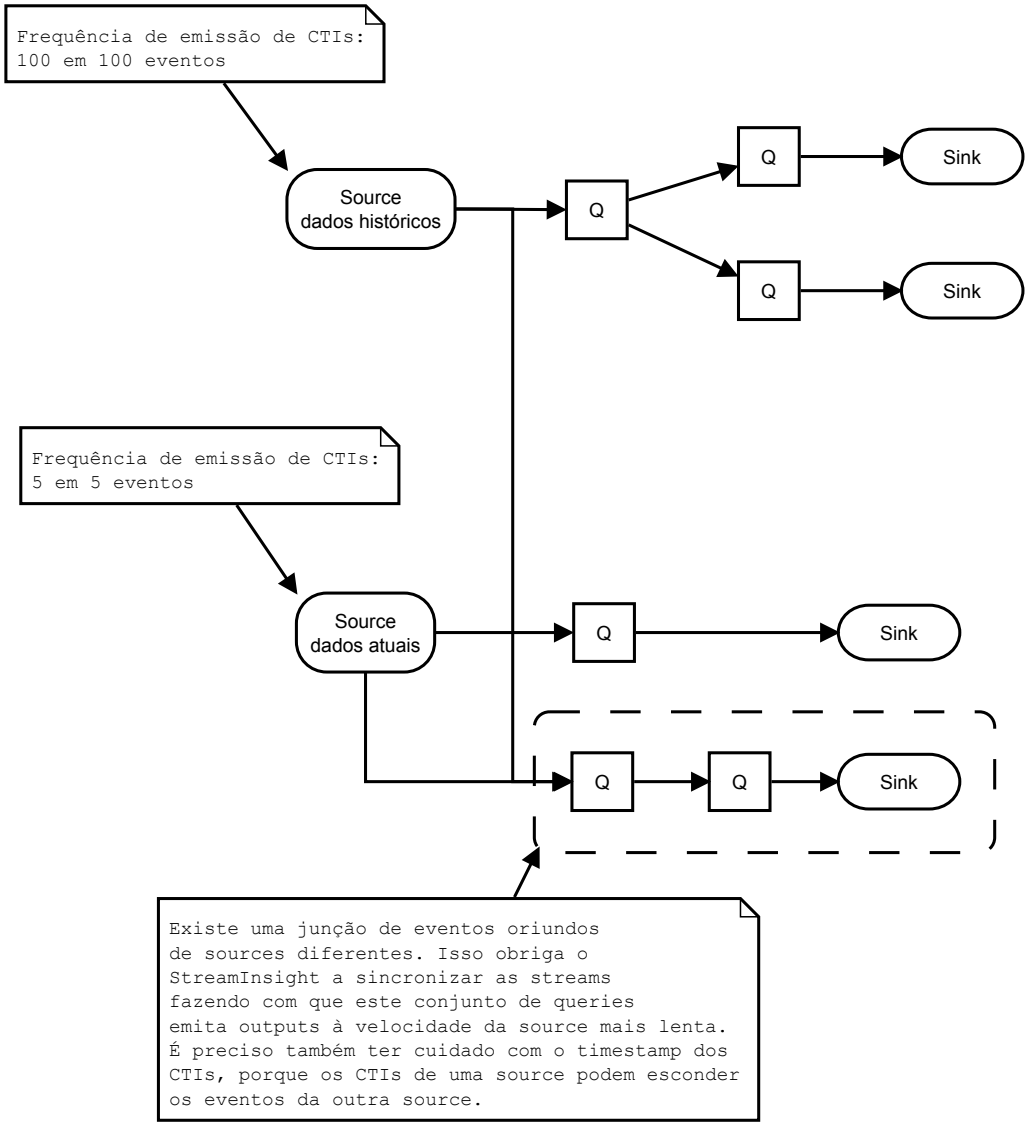

Figura 4.2: Exemplo de uma união entre duas *streams* que recebem eventos com frequências diferentes. Os blocos "Q" correspondem a *queries* escritas utilizando o LINQ.

Como se pode ver pela figura, o comportamento "*Adjust*" apenas modifica o *timestamp* de início do evento e nunca o *timestamp* de fim. Caso o evento tenha começado e terminado antes do evento CTI, o *StreamInsight* ignora sempre o evento, como se pode ver pela figura 4.4.

Outro fator importante é que o comportamento "*Adjust*" não pode afetar os eventos do tipo *Point Event* porque estes não têm uma duração, ou seja, o *timestamp* de início é igual ao *timestamp* de fim (lembrar que internamente o *StreamInsight* representa tudo como sendo *Interval Events* [RJA12]).

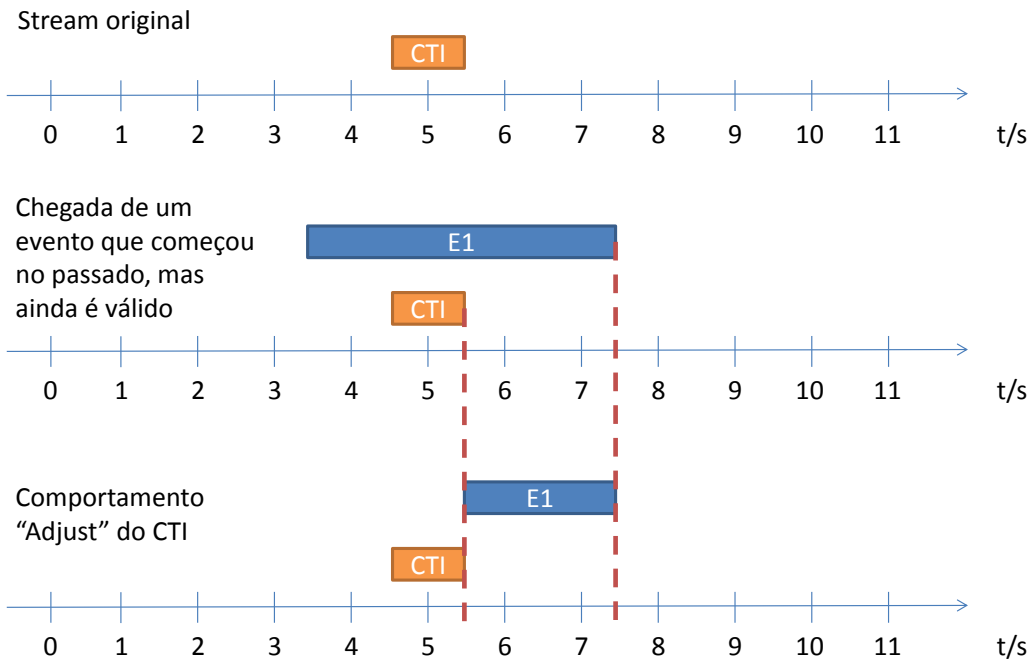

Figura 4.3: Comparação entre o comportamento "*Adjust*" e "*Drop*" dos eventos CTIs.

4.2 Elementos genéricos no payload

Para utilizar o *StreamInsight* é necessário especificar o *payload* dos eventos. Esta especificação tem de ser feita antes de o programa ser compilado, obrigando assim a que todos os tipos de eventos tenham de ser declarados antes de o programa ser compilado. Isto faz com que o *StreamInsight* não consiga processar novos tipos de eventos que possam surgir durante a sua execução.

Esta especificação *à priori* de eventos é feita através da escrita de uma classe onde as suas *properties* correspondem aos campos do evento. Por exemplo, um evento de um sensor de uma máquina podia ser representado pela classe:

```
1 public class MachineReading
2 {
3     public string MachineID { get; set; }
4     public string Lot { get; set; }
5     public double Temperature { get; set; }
6 }
```

De notar que a classe não incluí os campos *StartTime* e *EndTime*. Estes são adicionados automaticamente pelo *StreamInsight* sendo especificados quando o evento entra no mesmo.

A especificação *à priori* dos *payloads* dos eventos fornece algumas vantagens ao programador pois permite que os ambientes de desenvolvimento forneçam informações úteis como *IntelliSense* durante a escrita das *queries*, algo que não existe nos outros sistemas de CEP.

No entanto esta especificação torna o *StreamInsight* demasiado rígido, impedindo-o de processar eventos que não tenham sido especificados *à priori*.

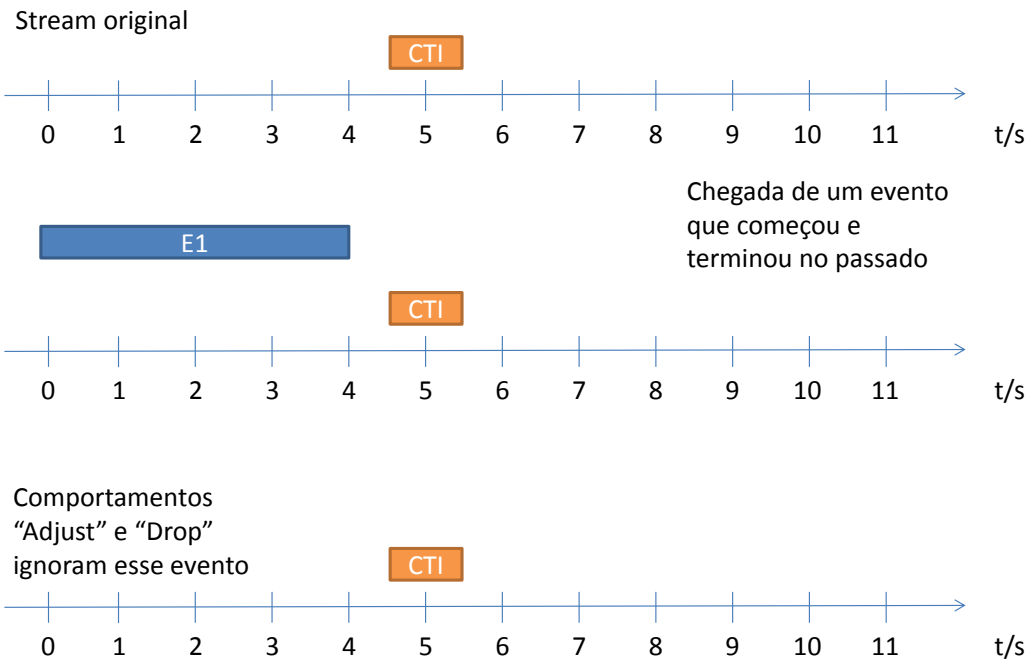

Figura 4.4: Comparação entre o comportamento "*Adjust*" e "*Drop*" dos eventos CTIs quando o evento que chega à *stream* tem um *timestamp* de fim inferior ao *timestamp* do evento CTI.

Um outro problema em processar eventos com *payloads* que não estejam especificados *à priori* prende-se com o fato de o *StreamInsight* não suportar todo o tipo de variáveis nos seus eventos. Os tipos de eventos que o *StreamInsight* suporta podem ser vistos na tabela 4.1. O tipo "*Nested Class*" corresponde a uma classe que também seja aceite pelo *StreamInsight*. Isto permite a criação de eventos que englobam outro tipo de eventos. No entanto uma lacuna importante é que o *StreamInsight* não suporta *containers*, isto é, não se pode utilizar "List<T>", "Dictionary<K,V>," etc. Isto complica a implementação de um sistema de processamento de eventos genérico.

Tabela 4.1: Tipos de eventos suportados pelo *StreamInsight*. Adaptado de: [MSDb].

Numérico	Temporal	Tamanho Variável	Outros
(u)int	DateTime	byte[]	guid
(u)short	TimeSpan	string	char
(u)long			Nested Class
float/double			
(s)byte			
Decimal			

Por exemplo assumindo que existe o evento "MachineEvent", definido na listagem 7.1 seria possível criar no *StreamInsight* o seguinte evento:

```
1 public class WrappedMachineEvent
```

```
{
    public string Commentary { get; set; }
    public MachineEvent machineEvent {get; set;}
}
```

O *CmStream* adiciona a habilidade de processar eventos de qualquer tipo. Para tal este assume que o *input* de entrada é uma *string* (que corresponde a um *payload* aceite pelo *StreamInsight*) no formato XML, que segue esta gramática:

```
          <xml> ::= "<xml>" <field>* "</xml>"
        <field> ::= "<Field Name=\"" <field-name> "\">"
                        <field-content>
                    "</Field>"
   <field-name> ::= string
<field-content> ::= string
```

Por exemplo a *string*:

```
"<xml>
  <Field Name=\"MachineID\">lps320</Field>
  <Field Name=\"Lot\">12A9L</Field>
  <Field Name=\"Temperature\">102</Field>
</xml>"
```

corresponde a um evento com um *payload* semelhante à classe apresentada anteriormente na listagem 7.1.

De notar que a gramática implica que não é possível utilizar XML com campos que contenham também eles sub-campos, isto é, a *string*:

```
"<xml>
  <Field Name=\"MachineID\">lps320</Field>
  <Field Name=\"Piece\">
    <Field Name=\"Lot\">12A9L</Field>
    <Field Name=\"Temperature\">102</Field>
  </Field>
</xml>"
```

não é um evento válido.

Infelizmente como o *StreamInsight* não suporta a utilização de eventos com *containers*, não é possível transformar essa *string* num evento com um *payload* mais fácil de trabalhar como um "Dictionary".

Para processar esta *string* foi adicionado um *Extension Method* à classe *String*, chamado "GetFieldValueIfXml" que permite obter o valor de um campo XML caso a *string* corresponda a um

XML com o formato especificado pela gramática mostrada anteriormente e que devolve *NULL* em caso de erro.

Isto permite que as *queries* consigam aceder ao valor dos campos XML do evento.

Esse *extension method* tem o seguinte algoritmo:

1. Converter a *string* em XML
2. Construir uma *string XPath* que acede ao campo pretendido. O formato da *string* é:

   ```
   "//Field[@Name = '$FIELD']"
   ```

   onde "$FIELD" é substituído pelo nome do campo que se quer obter
3. Aplicar o *XPath* ao XML e devolver o valor do primeiro nó encontrado

No entanto como esse método de extensão só devolve *strings* não é possível aplicar os operadores que o *StreamInsight* têm por defeito como o *Sum()* ou o *Avg()*. Como tal esses operadores foram implementados utilizando *User Defined Aggregate*s de forma a tirar partido do método "GetFieldValueIfXml". Estes operadores novos começam com o prefixo "*String*" e recebem um conjunto de *strings* representando XML e devolvem uma *string* XML com o resultado mais o StartTime e EndTime da janela onde o resultado foi calculado. Esta última informação permite que os resultados que são emitidos pelo *CmStream* possam ser enviado de volta para o mesmo, sem ser necessário fazer externamente uma fase de conversão.

Para além do método "GetFieldValueIfXml" foram criados mais dois *Extension Methods* sobre a classe *String* que trabalham com o XML da gramática descrita anteriormente:

1. AppendFieldToXml – Adiciona o campo com um determinado valor a ao XML de uma *string*.
2. HasXmlField – Devolve "*true*" se o XML da *string* contém o campo pretendido.

Os seguintes operadores foram implementados de forma a poderem receber um conjunto de eventos em formato XML e a devolverem uma *string* XML com o resultado pretendido:

- StringAverage – Implementação do operador *Avg* do *StreamInsight* mas que atua sobre *strings* XML
- StringCount – Implementação do operador *Count* do *StreamInsight* mas que atua sobre *strings* XML
- StringSum – Implementação do operador *Sum* do *StreamInsight* mas que atua sobre *strings* XML
- StringStandardDeviation – Calcula o desvio-padrão dos elementos

Exceto no caso das *Snapshots Windows*, todos estes operadores indicam no XML que devolvem o tempo de início e de fim da janela temporal aonde foram aplicados. Como as *Snapshot Windows* não permitem chamar operadores que utilizam esta informação, estes operadores foram implementados também numa versão em que o tempo de início e de fim das janelas temporais não é devolvido. Para estes operadores foi adicionado o sufixo "Insensitive", ou seja, foram implementados os operadores:

- StringAverageInsensitive
- StringCountInsensitive
- StringSumInsensitive
- StringStandardDeviationInsensitive

4.3 Arquitetura inicial

O *CmStream* disponibiliza os seguintes serviços aos seus clientes:

- Criar uma *query* do tipo *Hopping Window*, *Tumbling Window*, *Count Window*, *Snapshot Window* e *Session Window*
- Listar as *queries* que estão a ser executadas pelo *CmStream* a uma dada altura
- Terminar uma *query* que esteja a correr no *CmStream*

Quando o *CmStream* arranca este não tem conhecimento acerca das *queries* que vai executar. É necessário haver então uma forma de definir *queries* durante o *runtime* através de um pedido de um cliente do *CmStream*.

Para tal quando o *CmStream* arranca é criado um *source* que lê eventos, representados por *strings* de XML, a partir de uma de TCP/IP. Esta porta tem de ser *à priori* conhecida pelos clientes que queiram usar o *CmStream*.

A esta *source* são ligados sete *queries*:

1. Uma *query* que filtra os eventos para criar *Hopping Windows*;
2. Uma *query* que filtra os eventos para criar *Tumbling Windows*;
3. Uma *query* que filtra os eventos para criar *Count Windows*;
4. Uma *query* que filtra os eventos para criar *Snapshot Windows*;
5. Uma *query* que filtra os eventos para criar *Session Windows*;
6. Uma *query* que filtra os eventos para listar as *queries* que estão; a correr no servidor;

7. Uma *query* que filtra os eventos usados para terminar uma *query* que esteja a correr no *CmStream*.

Cada uma dessas *query* é então ligada ao um *sink* apropriado e que realiza as ações pedidas. Esta arquitetura pode ser vista na figura 4.5.

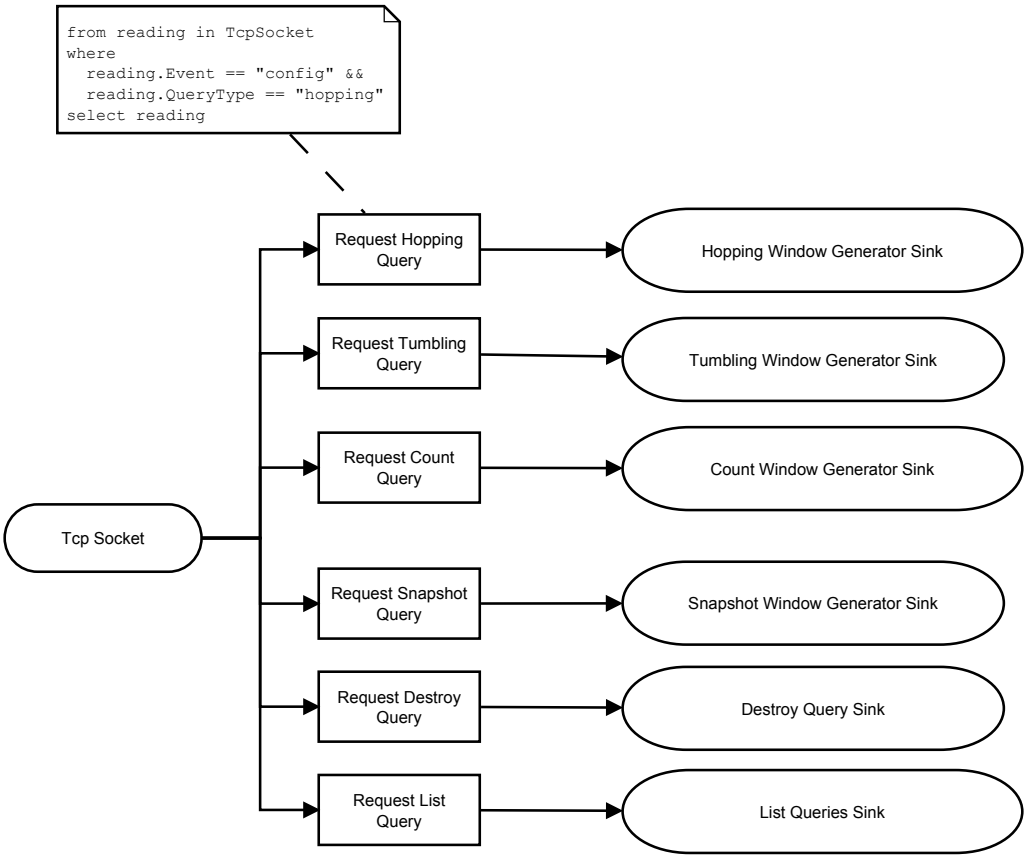

Figura 4.5: Arquitetura inicial do *CmStream*.

Cada uma destas *query* tem uma gramática específica que deve ser conhecida pelos clientes para que estes possam utilizar o *CmStream*. No entanto, devido à complexidade das gramáticas para a criação de *Hopping Windows*, *Tumbling Windows*, *Count Windows*, *Snapshot Windows* e *Session Windows* estas não estão descritas nesta secção, mas sim nos anexos deste documento.

Desta forma, a gramática para a criação de *Hopping Windows* pode ser consultada no anexo A.1, a gramática para a criação de *Tumbling Windows* no anexo A.2, a gramática para a criação de *Count Windows* no anexo A.3, a gramática para a criação de *Snapshot Windows* no anexo A.4 e a gramática para a criação de *Session Windows* no anexo A.5.

A *query* que filtra os eventos que são usados para a listagem das *queries* que estão a ser executadas pelo *CmStream* tem a seguinte gramática:

```
1 <xml> ::= "<xml><Field Name='event'>config</Field>"
2           "<Field Name='queryType'>list</Field>" <pattern>
```

```
3           <output> "</xml>"
4  <pattern> ::= "<Field Name='pattern'>" string "</Field>"
5  <output> ::= <output-source> <output-arguments>
6  <output-source> ::= "<Field Name='outputType'>"
7                      <output-type> "</Field>"
8  <output-argumpppppents> ::= "<Field Name='outputArguments'>"
9                              string "</Field>"
10 <output-type> ::= "socket" | "file" | "console"
```

A *query* que filtra os eventos que são usados para terminar um *query* que esteja a ser executada pelo *CmStream* tem a seguinte gramática:

```
1  <xml> ::= "<xml><Field Name='event'>config</Field>"
2            "<Field Name='queryType'>destroy</Field>" <query-id>
3            "</xml>"
4  <query-id> ::= "<Field Name='queryId'>" string "</Field>"
```

Estando definidas as gramáticas e ativadas as *queries* iniciais, o conjunto de passos que o *CmStream* realiza para criar uma *query* em *runtime* são:

1. Conectar-se ao sistema do *StreamInsight*
2. Transformar a *string* em XML num "Dictionary" em que as "*keys*" correspondem aos campos que o "*sink*" aceita e os "*values*" ao conteúdo desses campos
3. Realizar a operação pedida pelo cliente
4. Emitir uma mensagem em *broadcast* a informar os clientes da ação que o *server* acabou de realizar.

Ao criar uma *query* é possível aplicar um filtro à *stream*, o que permite restringir a aplicação dos operadores a um subconjunto dos eventos da *stream*.

Este filtro é passado como uma *substring* de código C# no campo "<filter-expression>" das gramáticas e deve avaliar em "*True*" ou "*False*". Este filtro é implementado utilizando a biblioteca "NCalc" [Cod].

Ao processar o filtro, qualquer conjunto de caracteres que não seja um número, operador booleano ou uma *string* é considerando o nome do campo de um evento.

Por exemplo o filtro:

```
Area > 1000
```

indica ao *CmStream* para só aplicar o operador aos eventos da *stream* que tenham o campo "Area" com o valor superior a 1000.

Outro exemplo de um filtro é:

```
Package == '12M10' && (SizeX < 2000 && SizeX > 1000)
```

que indica ao *CmStream* para só aplicar o operador aos eventos que tenham o campo "Package" com o valor "12M10" e o campo "SizeX" com um valor entre 1000 e 2000.

Caso não exista nenhum campo "<filter-expression>", a *query* é aplicada a todos os eventos que surjam na *stream*.

Para processar eventos fora de ordem é possível indicar ao *CmStream* que deva produzir os eventos CTIs com um determinado atraso. Esse atraso é especificado nas gramáticas através da regra "<grace-period>" que permite especificar um "TimeSpan" que indica quanto tempo é que o *CmStream* deve esperar pelos eventos fora de ordem.

Se o campo "<grace-period>" não existir, então os CTIs são sempre emitidos com o *timestamp* do último evento na *stream*, fazendo com que a *query* não espere por elementos fora de ordem.

4.4 Queries em real time

Tal como já foi referido, o *StreamInsight* funciona utilizando o conceito de "*Application Time*" e não o "*System Time*" (isto é o relógio do sistema onde o *StreamInsight* está a correr). Isto faz com que as *streams* só processem os eventos que têm quando recebem um evento CTI com um *timestamp* superior ao *timestamp* dos desses eventos.

Como estes eventos CTI só são emitidos, por defeito, quando a *stream* recebe um evento do tipo *Insert*. Isto faz com que se possa demorar muito tempo a receber um evento que tenha ficado retido na *stream*.

Para isso o *CmStream* disponibiliza o conceito de *queries* "*realtime*".

As *queries* "*realtime*" são *queries* onde o *CmStream* periodicamente injeta um evento CTI com a data e hora do sistema (isto é, o "*System Time*") o que força o *flush* das *streams*. A criação das *queries* "*realtime*" é feita através da regra "<real-time>" existente nas gramáticas descritas no anexo A. Esta regra das gramáticas permite passar ao sistema um "TimeSpan" que indica a frequência com que a *query* deva fazer "*flush*" dos seus eventos.

Isto faz com que as *queries* deixem de ser determinísticas, que é uma característica que o *StreamInsight* fornece [ACSK10, RAK+10], pois a resposta da mesma *query*, e com os mesmos dados pode ser diferente dependendo do *timestamp* a que o CTIs são emitidos. Este não-determinismo acontece mesmo que os CTIs sejam emitidos com a mesma frequência, pois a emissão dos CTIs não é alinhada.

Por exemplo, consideremos duas instâncias da mesma *query realtime* onde o *CmStream* emite um CTI de 10 em 10 minutos. Caso a primeira *query* seja criada às 12:00, os CTI são emitidos com os seguintes *timestamps*: 12:10, 12:20, etc. Por outro lado a mesma *query* arrancada às 12:05 vai receber CTIs com *timestamps* de 12:15, 12:25, etc. Isso pode mudar as respostas calculadas pelas *queries* porque os períodos de tempo que o *CmStream* espera por eventos fora de ordem vão ser diferentes.

Para injetar os CTIs numa *stream* é criado uma *stream* auxiliar que utiliza um "Observable.Interval" para gerar periodicamente um evento com o *timestamp* de "DateTime.UtcNow". Este

observable é depois convertido numa *stream* de "*Point Events*" utilizando a função "ToPointStreamable" que emite um CTI com o *timestamp* do evento que criou.

É depois criado um "Subject" que faz a junção entre a *stream* original e esta *stream* auxiliar. A razão pela qual não se pode utilizar o operador "Merge" ou "Union" fornecidos pelo *StreamInsight* é que este sincroniza as *streams* com a *stream* mais lenta, isto é, se tivermos uma *stream* que emita eventos de 1 em 1 segundos e outra que emita eventos de 1 em 1 minutos, o *StreamInsight* armazena um minuto de eventos na primeira *stream*, só libertando-os quando surge um evento na segunda *stream*. Isso pode ser visto na figura 4.2. Isto não é útil neste caso porque queremos que as duas *streams* (a principal e a dos CTIs) avance mesmo que a *stream* principal não tenha eventos. Utilizando um *Subject* é possível contornar essa limitação.

Esse *Subject* é depois filtrado, removendo os eventos do tipo *Insert* que estavam originalmente na *stream* CTI. Isso faz com que no final só fiquem os eventos CTI que a *stream* auxiliar tinha produzido para emitir os eventos *Insert* mais os eventos todos da *stream* original.

O conteúdo dos eventos *Insert* que a *stream* CTI produz contêm simples a *string* "ping". Esta *string* pode ser usada de forma segura porque não é uma *string* de XML válida e é imediatamente filtrada à saída do *Subject*.

Este operador encontra-se aplicado como um método de extensão sobre a classe "IQStreamable".

No final a arquitetura fica como pode ser vista no gráfico 4.6.

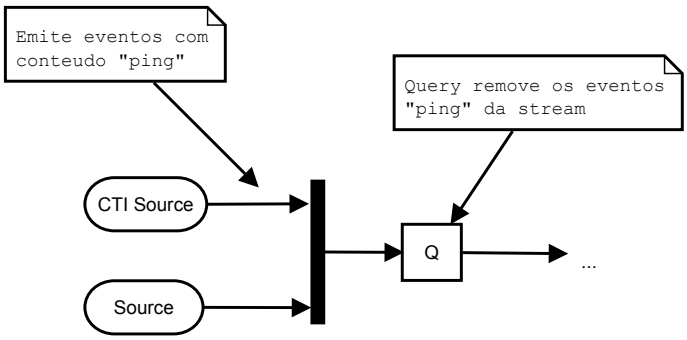

Figura 4.6: Arquitetura utilizada para implementar as "*realtime*" *queries*.

4.5 Session Windows

Um tipo de janelas novas fornecidas pelo *CmStream*, que não existem no *StreamInsight*, são as *Session Windows*. Este tipo de janelas permite a criação de agregações entre os eventos existentes entre dois eventos da *stream*, designados de "*Start Event*" e de "*End Event*", permitindo assim gerar janelas de tamanhos arbitrários, designada por "*session*".

Para construir uma *query* que crie "*Session Windows*" é necessário definir, para além do operador a aplicar (coisa que é comum aos outros tipos de janelas vistos em 3.4), o filtro que determina se um evento é um "*Start Event*", o filtro que determina se um evento é um "*End Event*" e o *timeout* que indica qual o tempo máximo que as "*session*" podem ter.

Assumindo que a *stream* original é a que pode ser vista na figura 4.7 e que o *timeout* é de 5 segundos, o algoritmo para a geração das *Session Windows* executa os seguintes passos:

1. Extração dos eventos do tipo "*Start Event*" da *stream* vista na figura 4.7 e adição do valor de "*timeout*" às suas durações. Este passo pode ser visto na figura 4.8.

2. Extração dos eventos do tipo "*End Event*" da *stream* vista na figura 4.7. Este passo pode ser visto na figura 4.9.

3. Modificar os eventos da *stream* da figura 4.8 de forma a que não ultrapassem os eventos de "*End Event*" da figura 4.9. Esta operação é designada por "ClipEventDuration" o resultado final pode ser vista na figura 4.10. Os eventos dessa *stream* recebem um número de identificação que vai funcionar como "*tag*" para os eventos que lhe vão pertencer.

4. Fazer a união no tempo entre a *stream* da figura 4.10 e os eventos (ignorando os "*Start Events*" e "*End Events*") que estavam na *stream* original. Aos eventos que intersectam uma janela temporal (isto é, um evento da figura 4.10) é-lhes adicionado a "*tag*" dessa janela. Esse passo pode ser visto na figura 4.11.

5. Finalmente é aplicado o operador "*group by*" à *stream* da figura 4.11 que agrupa os eventos por "*tag*" e aplica a cada grupo o operador pretendido.

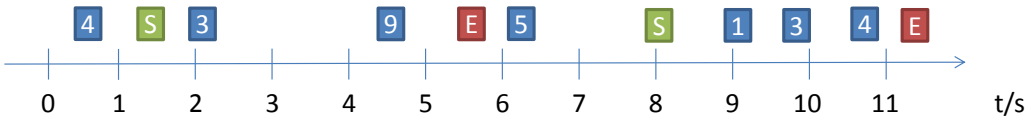

Figura 4.7: *Stream* original contendo uma mistura de eventos normais e eventos de início (os eventos verdes) e fim de sessão (os eventos vermelhos).

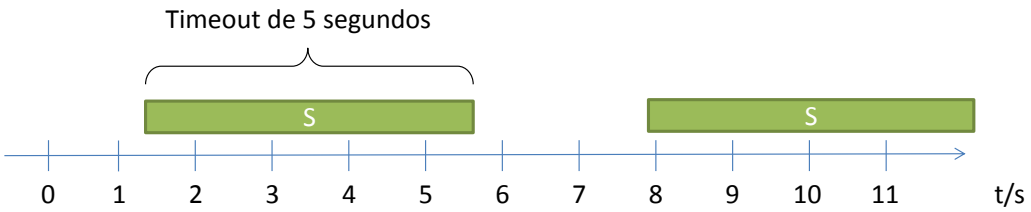

Figura 4.8: 1º Passo do algoritmo para a criação de *Session Windows*: Resultado da extração dos eventos de início de sessão e da adição do *timeout* a eles.

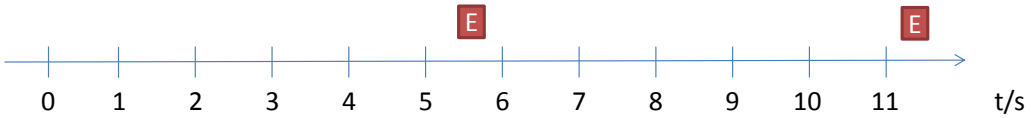

Figura 4.9: 2º Passo do algoritmo para a criação de *Session Windows*: Resultado da extração dos eventos de fim de sessão

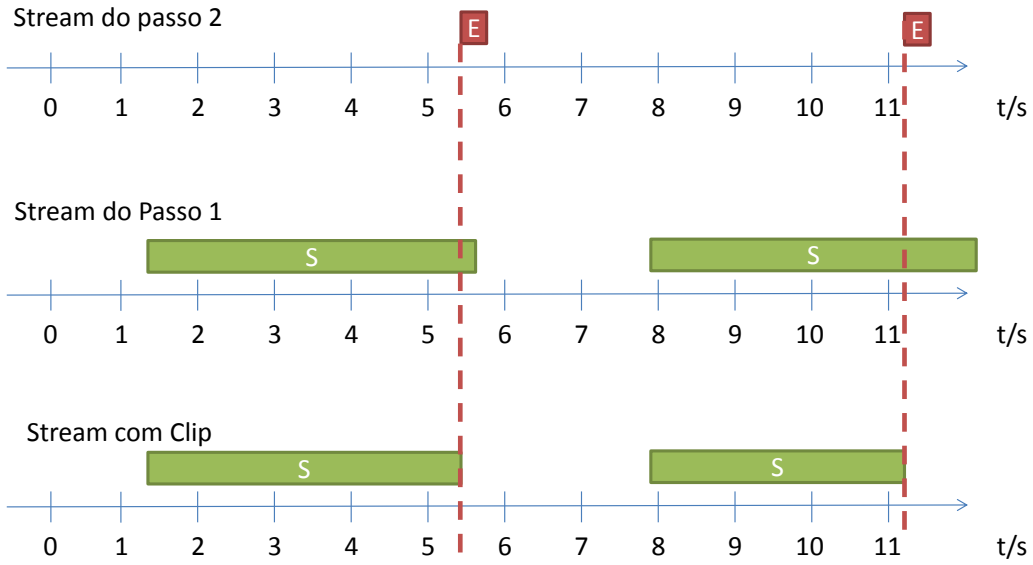

Figura 4.10: 3º Passo do algoritmo para a criação de *Session Windows*: Aplicar o operador "ClipEventDuration" para encurtar os eventos de início de sessão utilizando os eventos de fim de sessão encontrados. O resultado é uma *stream* em que cada evento corresponde a uma sessão.

4.6 Tipos de inputs e output

Tal como está especificado nas gramáticas de configuração no anexo A, o campo "inputType" pode ter dois valores: "socket" e "file".

Se o campo "inputType" tiver o valor "socket" então o campo "inputArguments" especifica uma porta e um endereço IP que o *CmStream* deve abrir para receber os eventos. Nesses casos o campo "inputArguments" deve ser uma *string* que segue o formato "hostname:porta", por exemplo "localhost:20001".

Se o campo "inputType" tiver o valor "file" então o campo "inputArguments" especifica o caminho para um ficheiro que será processado pelo *CmStream*. Esse ficheiro deverá ter um evento XML por linha. O XML desses eventos segue a mesma gramática que os eventos enviados por "socket".

No caso do campo "outputType" da gramática, este suporta os mesmos valores que o campo "inputType", ou seja, "socket" e "file". No entanto suporta também o valor "console" que indica que o *CmStream* deve fazer output dos eventos gerados para o "*Standard Output*". Quando o campo "outputType" é "console", o campo "outputArguments" não tem significado.

4.7 Interface gráfica

Para facilitar o teste do *CmStream* foi criado uma pequena interface gráfica que permite correr no *CmStream* vários tipos de *queries* e alimentar-las com eventos pré-definidos. Essa pode ser vista na figura 4.12. Como esta interface gráfica foi feita apenas para testes, ela não permite aceder a todas as funcionalidades do *CmStream*.

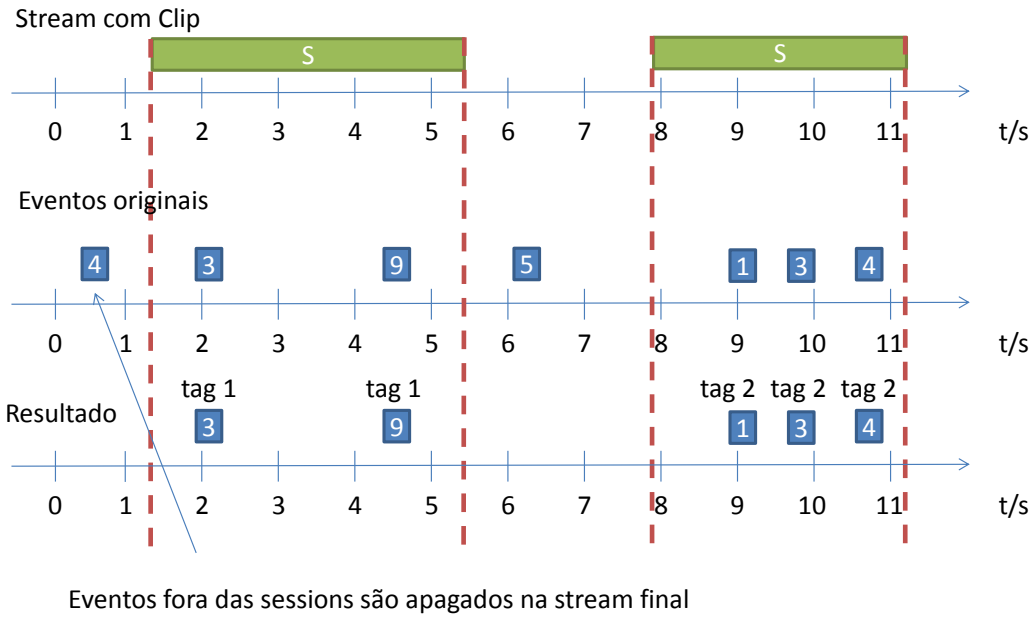

Figura 4.11: 4º Passo do algoritmo para a criação de *Session Windows*: Etiquetar os eventos da *stream* original (vista na figura 4.7) de forma a que cada evento tenha da mesma session tenha a mesma etiqueta.

4.8 Operadores fornecidos pelo CmStream

Pode-se ver nas gramáticas descritas na secção A dos anexos, os operadores que o *CmStream* suporta, são especificados pela regra "<operations>".

Desta forma os operadores que o *CmStream* suporta são:

- "average" – Extrai os valores de um campo dos eventos na janela e calcula a sua média. Internamente invoca o UDA "StringAverage".
- "sum" – Extrai os valores de um campo dos eventos na janela e calcula a sua soma. Internamente invoca o UDA "StringSum".
- "stddev" – Extrai os valores de um campo dos eventos na janela e calcula o seu desvio-padrão. Internamente invoca o UDA "StringStandardDeviation".
- "count" – Conta o número de vezes que um campo aparece nos eventos de uma janela. Internamente invoca o UDA "StringCount".
- "filter" – Extrai os valores de um campo dos eventos na janela e devolve-os. Internamente invoca o UDO "FetchXmlFieldFromString".
- "timeDifference" – Calcula a diferença temporal entre 2 eventos. Internamente invoca o UDSO "TimeDifferenceStateMachine".

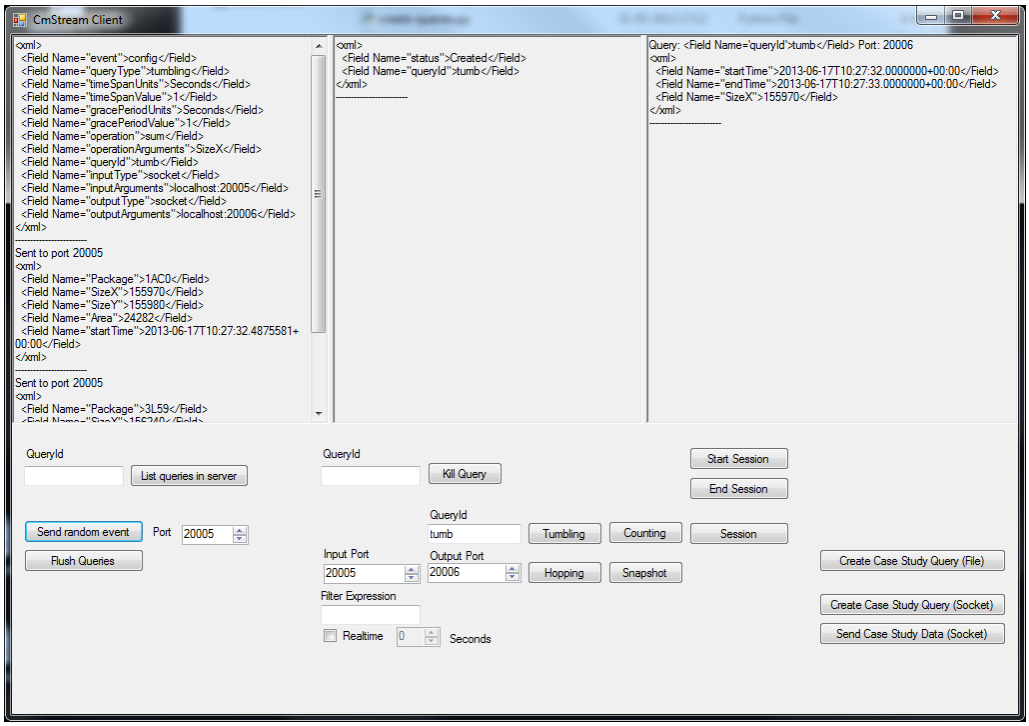

Figura 4.12: Screenshot da interface gráfica criada para testar o *CmStream*

Para os operadores “average”, “sum”, “stddev”, “count” e “filter”, o nome do campo ao qual se deve passar o operador é passado através da regra “<operation-arguments>” definidas nas gramáticas.

O operador “timeDifference” sendo implementado como um UDSO requer um conjunto de campos diferentes do que os outros operadores. Esses campos estão definidos na regra “<time-difference-extra-information>” e consistem em:

- Um filtro, semelhante ao usado pela regra “<filter-expression>”, que extrai da *stream* os eventos iniciais
- Um filtro, semelhante ao usado pela regra “<filter-expression>”, que extrai da *stream* os eventos finais
- Um *timeout* que indica quanto tempo se deve esperar pelo aparecimento de um evento final após encontrar um evento inicial.

Uma vez definidos estes campos, o operador “timeDifference” devolve a diferença temporal entre os eventos finais e os eventos iniciais.

O operador “timeDifference” funciona utilizando o seguinte algoritmo:

```
1 // Cleanup the stored "start" event if we have passed over the timeout
2 var currentApplicationTime = inputEvent.StartTime;
3
```

```
if (lastStartEventFound != null)
{
    if (inputEvent.StartTime > (lastStartEventFound.StartTime + _timeout))
    {
        lastStartEventFound = null;
    }
}

string payload = null;

if (inputEvent.EventKind == EventKind.Insert)
{
    payload = inputEvent.Payload;
}

var isStartEvent = QueryFilter.FilterCsharp(payload, _startFilter);
var isEndEvent = QueryFilter.FilterCsharp(payload, _endFilter);

// Ignore the "end" events that don't have a matching "start"
if (isEndEvent && lastStartEventFound == null)
{

}
else if (isStartEvent)
{
    lastStartEventFound = inputEvent;
}
else if (isEndEvent)
{
    var returnValue = new TimeDifference()
        {
            StartTime = lastStartEventFound.StartTime.DateTime,
            EndTime = inputEvent.StartTime.DateTime,
            Length = inputEvent.StartTime - lastStartEventFound.StartTime
        };

    lastStartEventFound = null;

    yield return returnValue;
}
```

A classe "TimeDifference" contêm a seguinte definição:

```
public class TimeDifference
{
    public DateTime StartTime { get; set; }
    public DateTime EndTime { get; set; }
    public TimeSpan Length { get; set; }
```

```
6 }
```

Este algoritmo assume que inicialmente o campo “lastStartEventFound” tem o valor “*null*”.

Capítulo 5

Resultados Experimentais

Para validar a arquitetura do *CmStream* descrita nesta dissertação foi realizado um caso de estudo que se focou na utilização de CEP para a análise de ficheiros de *log*. Estes *logs* contêm informação sobre os testes efetuados em máquinas de uma fábrica. Estes testes têm uma data de início e uma data de fim. Pretendeu-se com este caso de estudo determinar a quantidade de testes que estavam a ocorrer num determinado momento, para que se pudesse criar um gráfico semelhante ao da figura 5.1 (a construção do gráfico não é analisada neste trabalho).

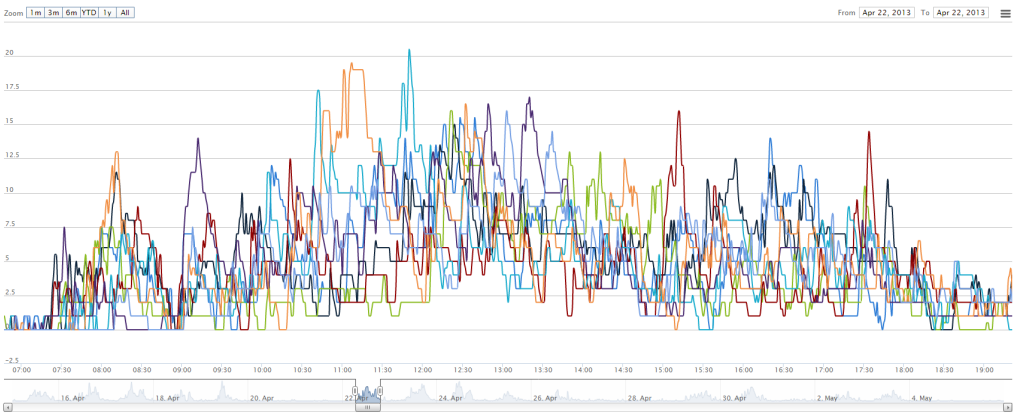

Figura 5.1: Exemplo da análise que é possível realizar utilizando os dados recolhidos para este caso de estudo. Cada linha do gráfico corresponde a uma máquina, sendo que o eixo das abcissas corresponde a linha temporal e a linha das ordenadas corresponde ao número de testes que a máquina encontrava a realizar em determinado momento.

Os ficheiros de *log* eram um ficheiro CSV com 4 campos:

1. MachineID - uma *string*
2. TestName - uma *string*
3. StartTime - uma data
4. EndTime - uma data

Todas as datas tinham precisão aos segundos, podendo ocorrer testes em que o campo "*StartTime*" fosse igual ao campo "*EndTime*". Isso representava os testes que demoraram menos de 1 segundo.

Por exemplo, isto seria uma linha válida do *log*:

```
maquina_foo;teste_bar;18 Apr 2013 08:35:54;18 Apr 2013 08:54:48
```

Para responder à pergunta pedida, basta utilizar uma *Tumbling Window* e aplicar o operador de *Count()* sobre os seus elementos. Em relação ao tamanho da janela considerou-se uma janela temporal de 5 minutos.

Este caso de estudo serviu também para fazer uma análise em termos de desempenho entre o *CmStream* e uma solução implementada sobre o *StreamInsight* normal.

O desempenho destas duas soluções foram também comparados com uma solução implementada numa base de dados relacional, implementada em *PostgreSQL* 9.2.4, servindo esta de *baseline*. A utilização de uma base de dados relacional como *baseline* serve para contrastar a solução de processamento de *logs* utilizada habitualmente pelas empresas com uma solução baseada num sistema de CEP.

Finalmente o desempenho destas duas soluções foi também comparada com a com duas soluções implementadas utilizando dois sistemas NoSQL. Estes sistemas foram implementados utilizando a tecnologia *Redis* 2.4.6, pois tal como o *StreamInsight* e o *CmStream* esta armazena e processa os dados em memória, e a tecnologia *MongoDB* 2.4.4 que permite processar grandes quantidades de dados através da utilização do paradigma *Map-Reduce*.

Foram utilizados 12 *datasets* podendo as características de cada *dataset* vistos na tabela 5.1. Cada *dataset* contêm dados relativos ao período entre 15 de abril de 2013 e 5 de maio de 2013.

Tabela 5.1: *Datasets* utilizados no caso de estudo. Nos *datasets* que contêm eventos fora de ordem os eventos podem chegar com 21 dias de atraso.

Dataset	Nº de linhas	Nº de máquinas	Eventos fora de ordem	Média de linhas por *log*
d25	149727	25	sim	5989,08
d50	304118	50	sim	6082,36
d75	447296	75	sim	5963,95
d100	600080	100	sim	6000,80
d125	766017	125	sim	6128,14
d150	928120	150	sim	6187,47
d25-ordered	149727	25	não	5989,08
d50-ordered	304118	50	não	6082,36
d75-ordered	447296	75	não	5963,95
d100-ordered	600080	100	não	6000,80
d125-ordered	766017	125	não	6128,14
d150-ordered	928120	150	não	6187,47

No caso dos *datasets* com elementos fora de ordem os elementos podem chegar com 21 dias de atraso. Isto força a que a nesses casos o sistema de CEP *stream* tenha de emitir os CTIs com 21 dias de atraso.

No caso da base de dados relacional e dos sistemas NoSQL os elementos fora de ordem não foram considerados.

Os testes foram corridos num portátil com as seguintes especificações:

- Windows 7 Enterprise 64 bits Service pack 1
- Processador Intel Core2 Duo P8400 2.26 GHZ
- 4 Gb de Ram

5.1 Solução PostgreSQL

A solução implementada em *PostgreSQL* consistiu na criação da seguinte tabela para cada um dos *datasets* com elementos fora de ordem:

```
1 CREATE TABLE dataset
2 (
3     machineid VARCHAR(30),
4     starttime TIMESTAMP,
5     endtime TIMESTAMP
6 );
```

De notar que o campo "TestName" que existia nos ficheiros CSV não foi colocado na tabela da base de dados.

Após a criação das tabelas, foram criados, para cada uma das tabelas, os seguintes índices, tendo sido depois executado o comando "VACUUM FULL ANALYSE" para os ativar:

```
1 CREATE INDEX ds ON dataset(starttime);
2 CREATE INDEX de ON dataset(endtime);
3 CREATE INDEX dse ON dataset(starttime, endtime);
```

Para calcular a resposta pedida pelo caso de estudo foi executada a seguinte *query*:

```
1 SELECT machineid, starttime, endtime, COUNT(*) FROM
2 (SELECT machineid, win.starttime, win.endtime
3         FROM dataset AS dataset,
4         generate_time_windows(
5             (SELECT MIN(starttime) FROM dataset),
6             (SELECT MAX(endtime) FROM dataset),
7             INTERVAL '5 minutes') AS win
8 WHERE (dataset.starttime, dataset.endtime)
```

```
        OVERLAPS (win.starttime, win.endtime))
    AS subquery
GROUP BY machineid, starttime, endtime
ORDER BY starttime;
```

Onde "generate_time_windows" corresponde à seguinte função de PL/SQL:

```
CREATE OR REPLACE FUNCTION generate_time_windows(
     IN f TIMESTAMP,
     IN t TIMESTAMP,
     IN i INTERVAL)
  RETURNS TABLE(starttime TIMESTAMP, endtime TIMESTAMP) AS
$BODY$
DECLARE
    -- date_trunc e uma funcao disponibilizadas pelo PostgreSQL
    s TIMESTAMP := date_trunc('day', f);
    e TIMESTAMP := date_trunc('day', t) + INTERVAL '1 days';
BEGIN
RETURN QUERY SELECT
    -- generate_series e uma funcao disponibilizada pelo PostgreSQL
    generate_series(s, e - i, i),
    generate_series(s + i, e, i);
RETURN;
END;
$BODY$
  LANGUAGE plpgsql;
```

A função "generate_series" utilizada por "generate_time_windows" é uma função fornecida pelo *PostgreSQL*.

A figura 5.2 mostra o plano de execução que o *PostgreSQL* utiliza para a *query*.

5.2 Solução StreamInsight

A solução implementada em *StreamInsight* consistiu apenas na realização da seguinte *query*, para cada ficheiro CSV do *dataset*:

```
// Se o dataset nao tiver eventos fora de ordem entao usar o
// AdvanceTimeSettings.IncreasingStartTime para gerar os CTIs, senao
// aguarda-se 21 dias por eventos fora de ordem
if (dataset.Contains("ordered"))
{
    source = myApp.DefineObservable(() => readCsvFile(ficheiro))
        .ToIntervalStreamable(
            x => IntervalEvent.CreateInsert(x.StartTime, x.EndTime, x.MachineId),
            AdvanceTimeSettings.IncreasingStartTime);
```

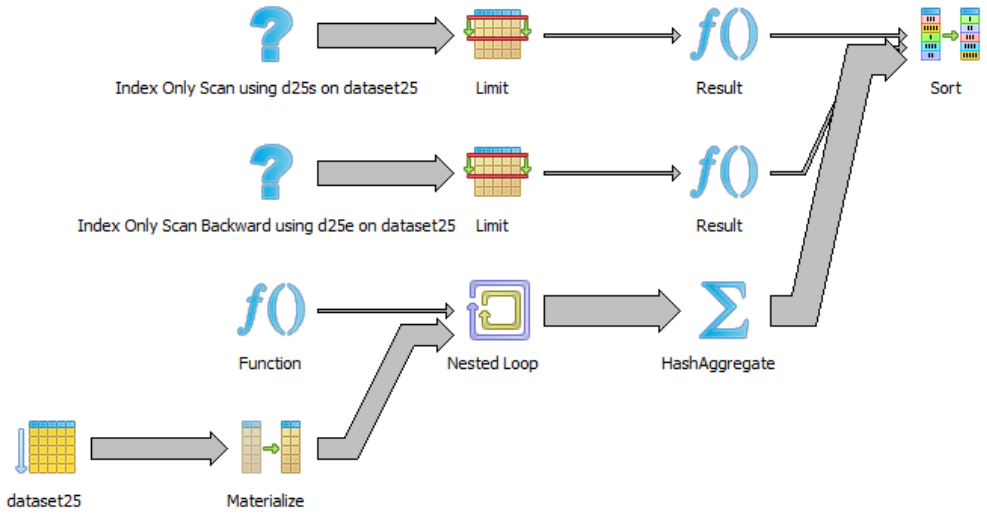

Figura 5.2: Plano de execução utilizado pelo *PostgreSQL* para responder à *query* do caso de estudo. Neste exemplo é utilizado o *dataset* "d25" da tabela 5.1. O plano da *query* encontra-se descrito em maior detalhe no anexo B.

```
10 }
11 else
12 {
13     source = myApp.DefineObservable(() => readCsvFile(ficheiros))
14         .ToIntervalStreamable(
15             x => IntervalEvent.CreateInsert(x.StartTime, x.EndTime, x.MachineId),
16             new AdvanceTimeSettings(
17                 new AdvanceTimeGenerationSettings(
18                     TimeSpan.FromDays(21),
19                     TimeSpan.FromDays(21),
20                     true),
21                 null, AdvanceTimePolicy.Adjust));
22 }
23
24 // Tamanho da janela
25 var frequency = TimeSpan.FromMinutes(5);
26
27 var query = from reading in source.TumblingWindow(frequency)
28             select new Window
29             {
30                 StartTime = reading.WindowStartTime(),
31                 EndTime = reading.WindowEndTime(),
32                 AggregateResult = reading.Count().ToString()
33             };
34
35 var sinkGenerator = myApp.DefineObserver(
36     (string filename) => Observer.Create<Window>(
```

```
37          x => File.AppendAllText(
38            String.Format(outputFolder + filename),
39            String.Format("{3};{0};{1};{2}\n", x.StartTime, x.EndTime,
40                          x.AggregateResult, filename))));
41
42  query.Bind(sinkGenerator(outputFile));
```

Onde os operadores "WindowsStartTime" e "WindowsEndTime" são dois "Time Sensitive" UDAs que devolvem, respetivamente o início e fim da janela temporal:

```
1  public class WindowStartTime : CepTimeSensitiveAggregate<string, DateTime>
2  {
3      public override DateTime GenerateOutput(
4          IEnumerable<IntervalEvent<string>> events,
5          WindowDescriptor windowDescriptor)
6      {
7          return windowDescriptor.StartTime.DateTime;
8      }
9  }
10
11 public class WindowEndTime : CepTimeSensitiveAggregate<string, DateTime>
12 {
13     public override DateTime GenerateOutput(
14         IEnumerable<IntervalEvent<string>> events,
15         WindowDescriptor windowDescriptor)
16     {
17         return windowDescriptor.EndTime.DateTime;
18     }
19 }
```

5.3 Solução CmStream

Para responder ao caso de estudo utilizando o *CmStream*, foi necessário primeiro converter os ficheiros CSV no formato de ficheiro que o *CmStream* aceita.

Assim, cada entrada num ficheiro CSV foi convertida de forma a ficar com esta sintaxe:

```
1  <xml>
2    <Field Name='Machine'>...</Field>
3    <Field Name='startTime'>...</Field>
4    <Field Name='endTime'>...</Field>
5  </xml>
```

Após essa conversão foi enviada para o *CmStream* o seguinte XML de configuração para cada um dos ficheiros do *dataset*:

```
<xml>
  <Field Name="event">"config"</Field>
  <Field Name="queryType">"tumbling"</Field>
  <Field Name="timeSpanUnits">"Minutes"</Field>
  <Field Name="timeSpanValue">"5"</Field>
  <Field Name="operation">"count"</Field>
  <Field Name="operationArguments">"machine"</Field>
  <Field Name="gracePeriodUnits">"Days"</Field>
  <Field Name="gracePeriodValue">"25"</Field>
  <Field Name="queryId">"caseStudyQuery-" + $input_file</Field>
  <Field Name="inputType">"file"</Field>
  <Field Name="inputArguments">$input_file</Field>
  <Field Name="outputType">"file"</Field>
  <Field Name="outputArguments">$output_file</Field>
</xml>
```

Onde "$input_file" corresponde ao caminho do ficheiro com os eventos em formato XML e "$output_file" corresponde ao caminho do ficheiro para onde o *CmStream* deve escrever as suas respostas.

5.4 Solução MongoDB

A solução implementada em *MongoDB* consistiu na criação de uma "*collection*"[1] para cada um dos *datasets* com elementos fora de ordem. Os elementos de cada *dataset* foram transformados em *objetos* JSON com o seguinte formato:

```
{
    Machine = <maquina>
    StartTime = new Date(<start_time>),
    EndTime = new Date(<end_time>)
}
```

Uma vez inseridos os objetos nas *collections* pode-se proceder à implementação da *query* para dar a resposta pretendida. Ao contrário de uma base de dados relacional, o *MongoDB* não suporta SQL. Sendo assim as suas *queries* são normalmente implementadas utilizando o paradigma *Map-Reduce*. Este divide a *query* em duas fases:

1. *Map* – Parte do algoritmo da *query* que itera sobre os elementos da "*collection*" e realiza projeções e outro tipo de processamento, enviando esses dados para a fase de "*Reduce*".

2. *Reduce* – Parte do algoritmo da *query* que recebe os dados gerados durante a fase de "*Map*" e que agrega esses dados numa só resposta.

[1] Uma "*collection*" pode ser pensada como sendo uma tabela de uma base de dados onde não existe um *schema*, permitindo assim guardar documentos com tipos diferentes de campos

Ambas as fases recebem uma função, escrita em Javascript com estas caraterísticas:

- Fase "*Map*" – Neste caso a função não recebe nenhum argumento, sendo a variável "this" a representar um objeto da "*collection*". Dentro desta função é possível utilizar a função "*emit*" que envia para a fase "*Reduce*" um objeto composto por um objeto "*key*" e por um objeto "*value*".
- Fase "*Reduce*" – Neste caso a função recebe dois argumentos: um objeto que tenha sido emitido na fase "*Map*" como sendo uma "*key*" e um *array* contendo objetos emitidos como "*value*" com essa chave.

Neste caso de estudo a função utilizada para a fase de "*Map*" foi utilizada para decompor os intervalos de tempos entre os campos "StartTime" e "EndTime" em blocos de 5 minutos. Para cada um desses blocos a função emite um objeto em que a "*key*" corresponde ao identificador da máquina e a um bloco de tempo em que a máquina esteve a trabalhar e o "*value*" é igual a 1.

O código Javascript utilizado para implementar este algoritmo foi:

```
1  function() {
2      var startDate = this.StartTime;
3      var endDate = this.EndTime;
4
5      // Criar janelas com este tamanho (medido em minutos)
6      var delta = 5;
7
8      // Converter as datas no numero de minutos que passaram desde
9      // de 1 de janeiro de 1970 (designado "Unix Epoch")
10     var startDelta = Math.floor(startDate.getTime() / 1000 / 60);
11     var endDelta = Math.ceil(endDate.getTime() / 1000 / 60);
12
13     var startInterval = startDelta;
14
15     var endInterval = 0;
16
17     if ((endDelta % delta) != 0) {
18         endInterval = endDelta + (delta - (endDelta % delta));
19     }
20     else {
21         endInterval = endDelta;
22     }
23
24     // Numero da janela temporal atual
25     var current = startInterval - (startInterval % delta);
26
27     if (startDelta == endDelta || (endDelta - current) < delta) {
28         // emit e uma funcao fornecida pelo MongoDB que gera um objecto
29         // que e enviado para a funcao reduce
30         emit({ machine: this.Machine, time: current }, 1);
```

```
31      }
32      else {
33          while (current < endInterval) {
34              emit({ machine: this.Machine, time: current }, 1);
35
36              current += delta;
37          }
38      }
39  }
```

A função utilizada para a fase "*Reduce*" simplesmente calcula a soma dos números que estão no *array* de "*values*" emitidos durante a fase de "*Map*". O seu código é:

```
1  function(key, values) {
2      var returnValue = 0;
3
4      for (var i = 0; i != values.length; i++) {
5          returnValue += values[i];
6      }
7
8      return returnValue;
9  }
```

No final da fase de "*Reduce*" o *MongoDB* devolve um objeto composto pela "*key*" e o "*value*" devolvido pela função utilizada para a fase de "*Reduce*". O *MongoDB* permite também definir uma função que é chamada após a fase de "*Reduce*" e que permite realizar alguma operação sobre os valores devolvidos por esta fase.

Esta função, designada como "*Finalize*", recebe dois argumentos, o valor de uma "*key*" e o respetivo "*value*" criado durante a fase de "*Reduce*". Neste caso de estudo a função "*Finalize*" devolve o objeto criado durante a fase de "*Reduce*" mas com o campo "time" convertido num objeto do tipo "Date".

O código para esta função foi:

```
1  function(key, value) {
2      return {
3          machine: key.machine,
4          time: new Date(key.time * 60 * 1000),
5          count: value
6      };
7  }
```

5.5 Solução Redis

A solução implementada em *Redis* consistiu na implementação de um programa C# que subdividia os tempos de execução dos elementos do *dataset* em blocos de tempo de 5 minutos utilizando o mesmo algoritmo que a função utilizada pela solução *MongoDB* na fase de "*Map*" descrita na secção 5.4.

Nesta implementação o *Redis* foi utilizado como um conjunto de *Hash-Tables* (cada máquina do *dataset* tinha uma *Hash-Table* própria) de contadores onde cada "*key*" era o número da janela temporal onde a máquina tinha realizado pelo menos um teste e o "*value*" era o número de testes que a máquina esteve a realizar durante essa janela temporal.

Para fazer a ligação entre o C# e o *Redis* foi utilizado a versão 3.9.54.0 da biblioteca "ServiceStack.Redis"[2].

Desta forma o código para implementar este algoritmo foi (ignorando o código para a escrita das respostas para um ficheiro):

```
using (var client = new RedisClient())
{
    // Limpa a cache do Redis
    client.FlushDb();

    // "dataset" corresponde ao nome do dataset que esta
    // a ser processado
    var totalFiles = Directory.GetFiles(
          "C:\\Users\\re-pereira\\Desktop\\escola\\datasets\\" + dataset,
          "*.csv");

    foreach (var file in totalFiles)
    {
        ParseFile(file, client);
    }
}
```

Onde a função "ParseFile" implementa o algoritmo descrito na secção 5.4 para a fase de "*Map*" do *MongoDB*

```
private static RedisClient ParseFile(string filename,
                                     RedisClient client)
{
    var fileContents = File.ReadAllLines(filename);

    var machineName = filename.Split('\\').Last();

    var fields = fileContents.Select(line => line.Split(';'))
```

[2]Repositório de código disponível em: `https://github.com/ServiceStack/ServiceStack.Redis`

```
9         .Select(csvFields => new
10        {
11            MachineId = csvFields[0],
12            StartTime = (int)Math.Floor(
13                            TimeSinceUnix(DateTimeOffset.Parse(
14                                          csvFields[2],
15                                          null,
16                                          DateTimeStyles.AssumeUniversal))
17                           .TotalMinutes),
18            EndTime = (int)Math.Ceiling(
19                          TimeSinceUnix(DateTimeOffset.Parse(
20                                        csvFields[3],
21                                        null,
22                                        DateTimeStyles.AssumeUniversal))
23                          .TotalMinutes),
24            StartDateTime = DateTimeOffset.Parse(
25                                  csvFields[2],
26                                  null,
27                                  DateTimeStyles.AssumeUniversal),
28            EndDateTime = DateTimeOffset.Parse(
29                                  csvFields[3],
30                                  null,
31                                  DateTimeStyles.AssumeUniversal)
32        });
33
34    var delta = 5;
35
36    foreach (var testEvent in fields)
37    {
38        var startInterval = testEvent.StartTime;
39        int endInterval = 0;
40
41        if (testEvent.EndTime % delta != 0)
42            endInterval = testEvent.EndTime + (delta - (testEvent.EndTime % delta));
43        else
44            endInterval = testEvent.EndTime;
45
46        var current = startInterval - (startInterval % delta);
47
48        if (testEvent.StartTime == testEvent.EndTime ||
49                (testEvent.EndTime - current) < delta)
50            client.HIncrby(machineName,
51                           Encoding.UTF8.GetBytes(current.ToString()),
52                           1);
53        else
54        {
55            while (current < endInterval)
56            {
57                client.HIncrby(machineName,
```

```
58                                  Encoding.UTF8.GetBytes(current.ToString()),
59                                  1);
60
61                  current += delta;
62              }
63          }
64      }
65
66      return client;
67  }
```

A função "TimeSinceUnix" é uma função auxiliar que converte um objeto do tipo "DateTimeOffset" para um objeto "TimeSpan" que indica o tempo que passou desde de 1 de janeiro de 1970 (designado de "Unix Epoch"). Esta função tem o seguinte código:

```
1  public static TimeSpan TimeSinceUnix(DateTimeOffset date)
2  {
3      return date - new DateTimeOffset(1970, 1, 1, 0, 0, 0, TimeSpan.Zero);
4  }
```

5.6 Medições

Para cada *dataset* foram realizadas 10 experiências, tendo sido armazenados as médias obtidas. Para cada implementação foram medidos o tempo de execução bem como o consumo de CPU e de memória, exceto na análise realizada com a solução implementada em *PostgreSQL* onde só foi medido o tempo de execução.

Os gráficos gerados nesta secção foram feitos com os dados que foram recolhidos durante a execução das várias implementações. Devido à grande quantidade de dados que foram recolhidos e que foram utilizados para criar os gráficos desta secção estes dados não estão enumerados neste documento. No entanto o autor disponibiliza na página `http://paginas.fe.up.pt/~ei08150/doku/doku.php?id=bd_logs` uma base de dados *SQLite* com todos os dados necessários para gerar estes gráficos, bem como os passos utilizados os gerar.

5.6.1 Tempo de execução

O gráfico da figura 5.3 tempo de execução que cada implementação precisou para dar a resposta à *query* do caso estudo, dependendo da existência de eventos fora de ordem. O gráfico da figura 5.4 permite observar melhor as diferenças entre as implementações de *StreamInsight*, *CmStream*, *MongoDB* e *Redis*. Consegue-se ver nesses gráficos que apesar do *CmStream* ter pior desempenho que a implementação utilizando o *StreamInsight* e o *MongoDB*, este tem uma melhor desempenho do que a implementação utilizando o sistema de base de dados e o *Redis*.

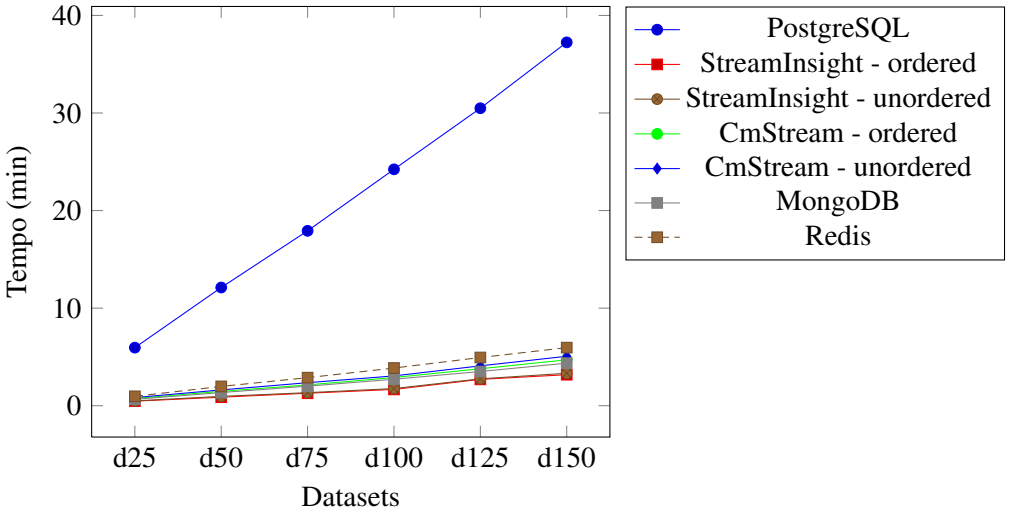

Figura 5.3: Tempo de execução das diversas implementações. Os tempos mostrados são a média de 10 execuções. Os valores com "ordered" na legenda correspondem à utilização de *datasets* que não contém eventos fora de ordem.

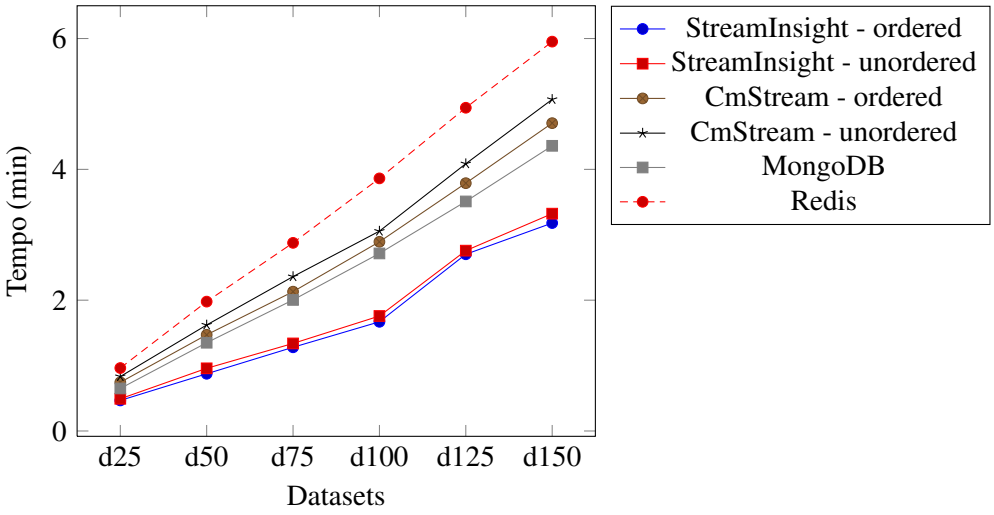

Figura 5.4: Tempo de execução das implementações de Complex Event Processing e NoSQL. Os tempos mostrados são a média de 10 execuções. Os valores com "ordered" na legenda correspondem à utilização de *datasets* que não contém eventos fora de ordem.

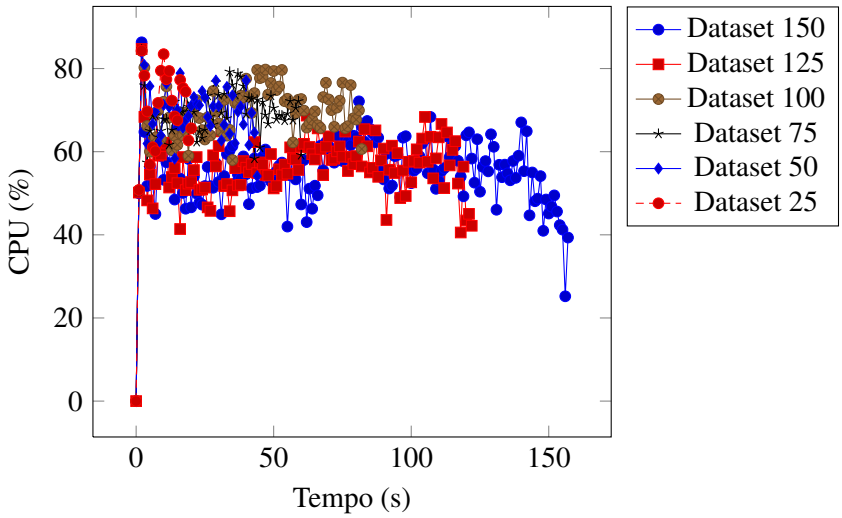

Figura 5.5: Consumos CPU ao longo do tempo para a implementação usando *StreamInsight*. A percentagem de CPU corresponde à media dos dois CPUs existentes na máquina. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*. Os *datasets* mostrados contêm eventos fora de ordem.

5.6.2 Consumos StreamInsight

Os gráficos das figuras 5.5 e 5.6 mostram, respetivamente os consumos de CPU e de memória da implementação utilizando o *StreamInsight* para os *datasets* que contêm eventos fora de ordem. Os gráficos das figuras 5.7 e 5.8 mostram a mesma informação, mas para os *datasets* que não contêm eventos fora de ordem.

5.6.3 Consumos CmStream

Os gráficos das figuras 5.9 e 5.10 mostram, respetivamente os consumos de CPU e de memória da implementação utilizando o *CmStream* para os *dataset* que contêm eventos fora de ordem. Os gráficos das figuras 5.11 e 5.12 mostram a mesma informação mas para os *datasets* que não contêm eventos fora de ordem.

5.6.4 Consumos MongoDB

Os gráficos das figuras 5.13 e 5.14 mostram, respetivamente os consumos de CPU e de memória da implementação utilizando o *MongoDB*.

5.6.5 Consumos Redis

Os gráficos das figuras 5.15 e 5.16 mostram, respetivamente os consumos de CPU e de memória da implementação utilizando o *Redis*. De notar que o gráfico da figura 5.15 não ultrapassa os 50% porque o *Redis* é uma aplicação "*single-threaded*".

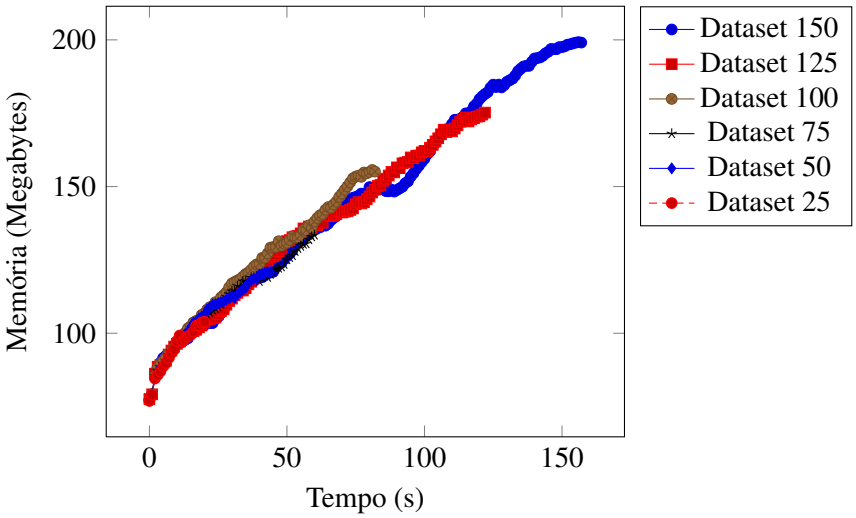

Figura 5.6: Consumos de memória ao longo do tempo para a implementação usando *StreamInsight*. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*. Os *datasets* mostrados contêm eventos fora de ordem.

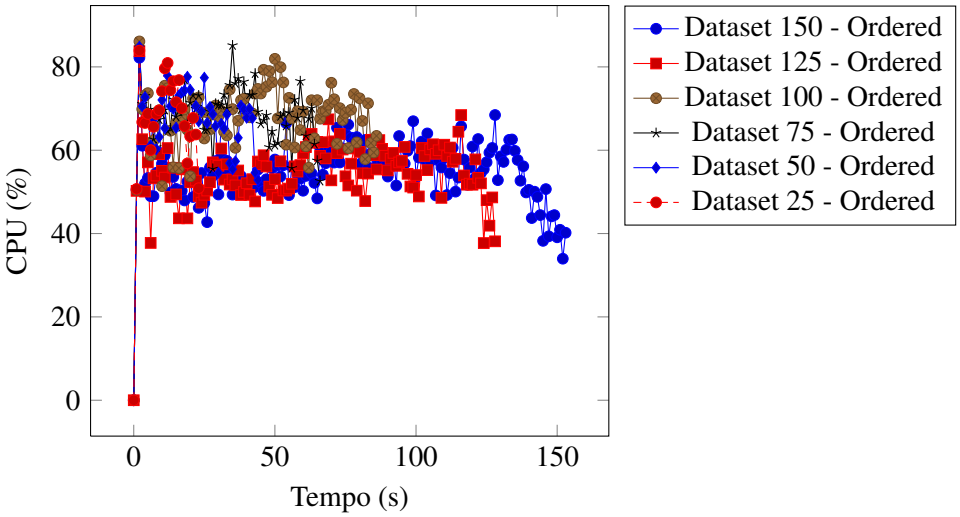

Figura 5.7: Consumos CPU ao longo do tempo para a implementação usando *StreamInsight*. A percentagem de CPU corresponde à media dos dois CPUs existentes na máquina. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*. Os *datasets* mostrados não contêm eventos fora de ordem.

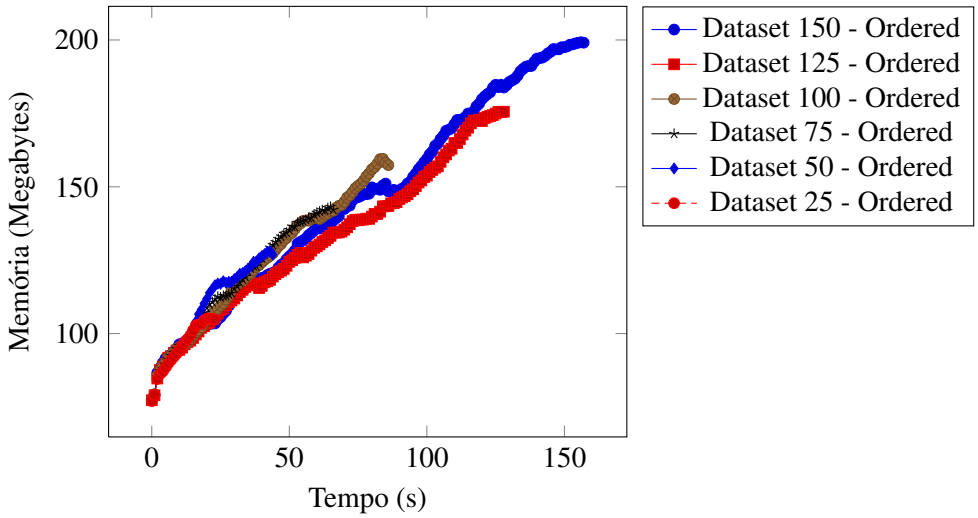

Figura 5.8: Consumos de memória ao longo do tempo para a implementação usando *StreamInsight*. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*. Os *datasets* mostrados não contêm eventos fora de ordem.

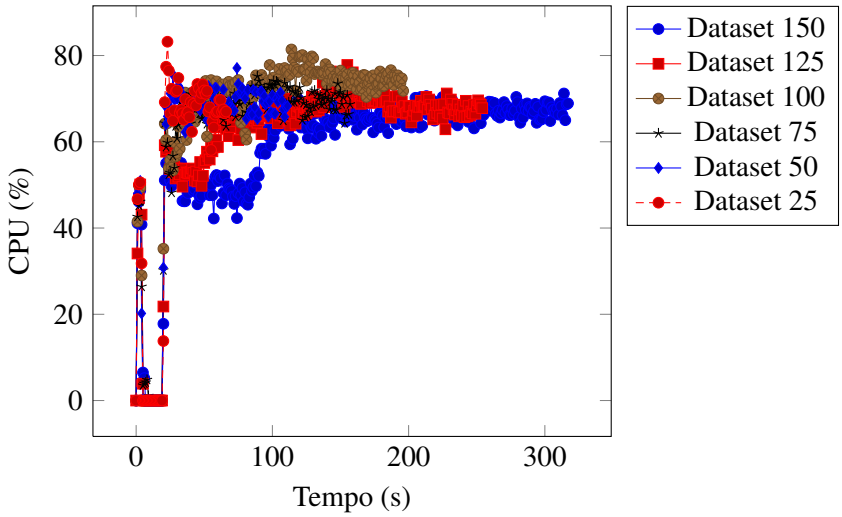

Figura 5.9: Consumos de CPU ao longo do tempo para a implementação usando *CmStream*. A percentagem de CPU corresponde à media dos dois CPUs existentes na máquina. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*. Os *datasets* mostrados contêm eventos fora de ordem.

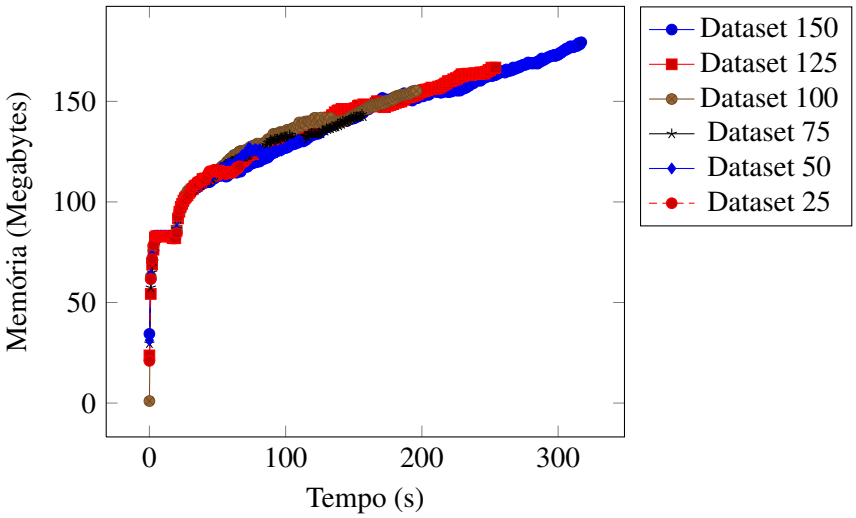

Figura 5.10: Consumos de memória ao longo do tempo para a implementação usando *CmStream*. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*. Os *datasets* mostrados contêm eventos fora de ordem.

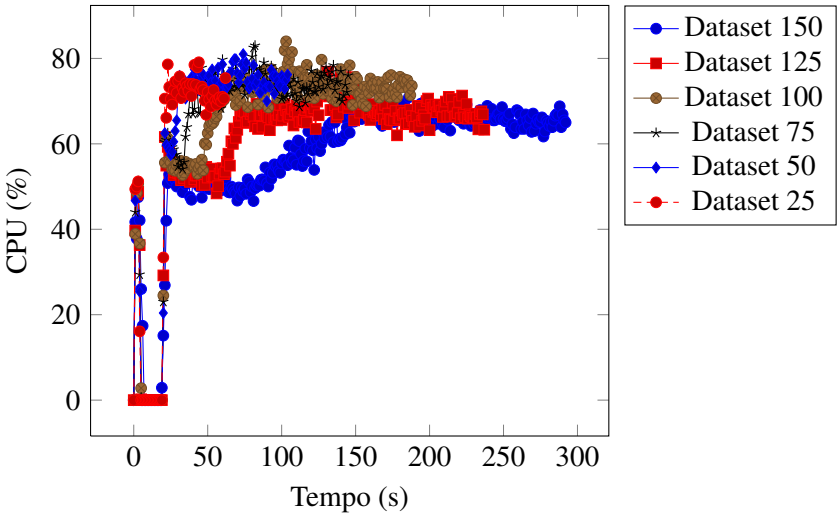

Figura 5.11: Consumos de CPU ao longo do tempo para a implementação usando *CmStream*. A percentagem de CPU corresponde à media dos dois CPUs existentes na máquina. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*. Os *datasets* mostrados não contêm eventos fora de ordem.

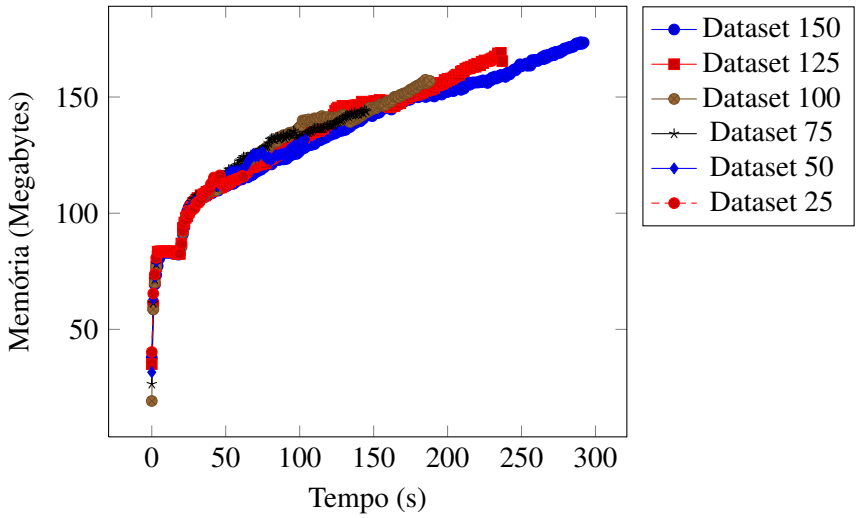

Figura 5.12: Consumos de memória ao longo do tempo para a implementação usando *CmStream*. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*. Os *datasets* mostrados não contêm eventos fora de ordem.

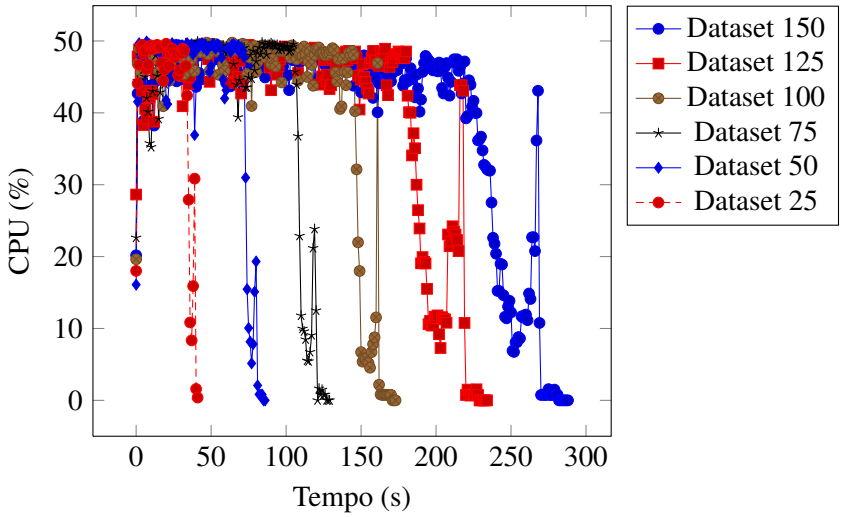

Figura 5.13: Consumos CPU ao longo do tempo para a implementação usando *MongoDB*. A percentagem de CPU corresponde à media dos dois CPUs existentes na máquina. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*.

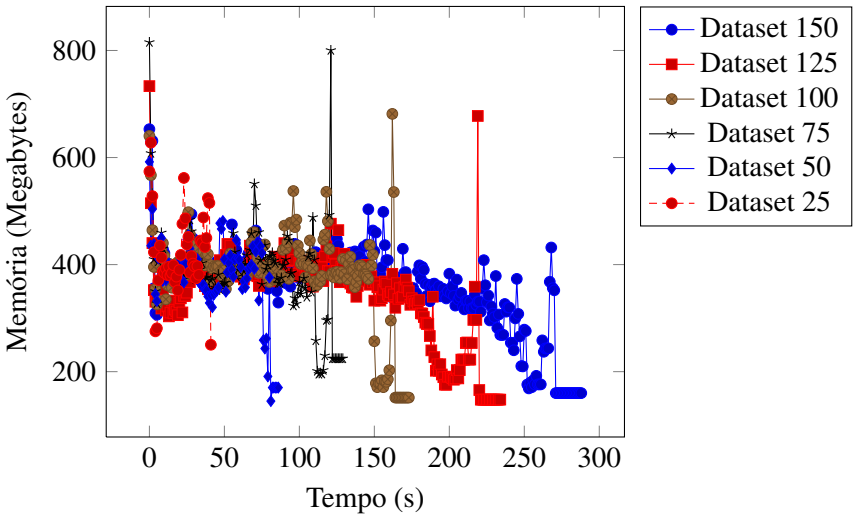

Figura 5.14: Consumos de memória ao longo do tempo para a implementação usando *MongoDB*. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*.

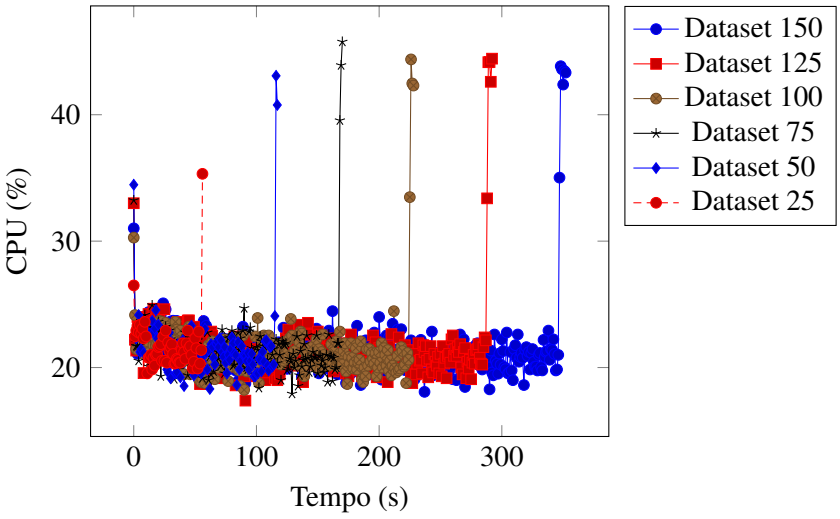

Figura 5.15: Consumos CPU ao longo do tempo para a implementação usando *Redis*. A percentagem de CPU corresponde à media dos dois CPUs existentes na máquina. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*.

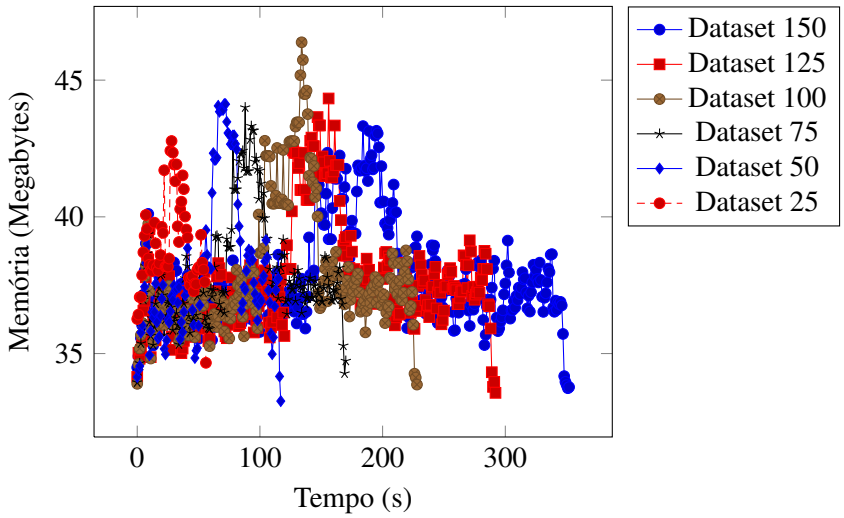

Figura 5.16: Consumos de memória ao longo do tempo para a implementação usando *Redis*. Os consumos mostrados são a média das 10 experiências que foram realizadas para cada *dataset*.

5.6.6 Comparação entre os consumos de várias soluções

Finalmente, os gráficos das figuras 5.4, 5.18 e 5.19 mostram a comparação entre, respetivamente, os tempos de execução, consumos de CPU e consumos de memória das soluções implementadas em *StreamInsight*, *CmStream*, *MongoDB* e *Redis*.

Consegue-se ver com este caso de estudo que o *CmStream* consegue funcionar sem ter conhecimento à priori dos eventos que vai receber e que apesar de ter pior desempenho, em termos de tempo gasto para calcular a resposta à *queries*, que as implementações feitas em *MongoDB* e em *StreamInsight*, consegue ser melhor que as implementações feitas sobre *PostgreSQL* e *Redis*.

A tabela 5.2 mostra algumas vantagens e desvantagens em utilizar cada um destes sistemas.

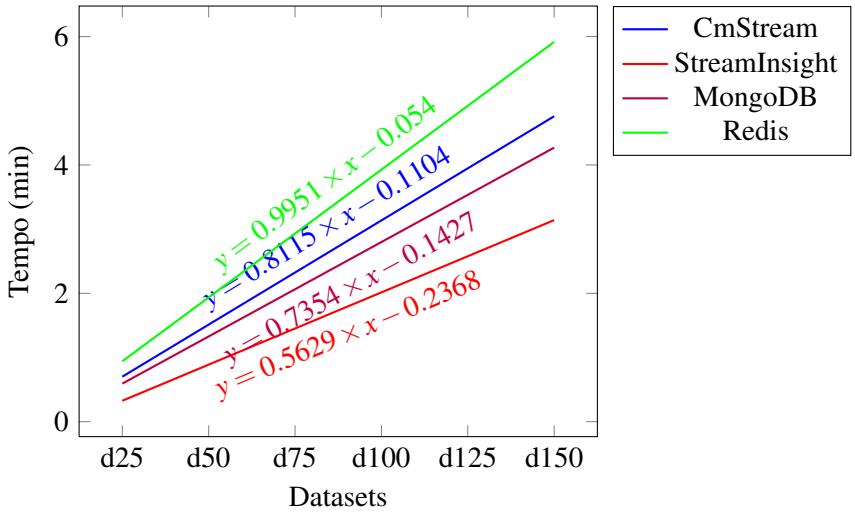

Figura 5.17: Relação entre o número de máquinas no *dataset* e o tempo de processamento das implementações *StreamInsight*, *CmStream*, *MongoDB* e *Redis*. Estes valores são calculados através da regressão linear dos dados do gráfico 5.4.

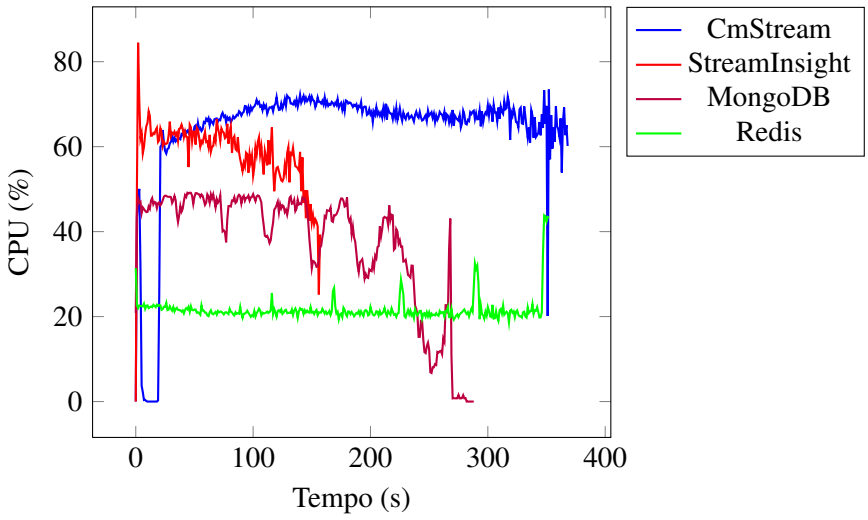

Figura 5.18: Relação entre o tempo de processamento e o consumo do CPU das implementações em *StreamInsight*, *CmStream*, *MongoDB* e *Redis*. A percentagem de CPU corresponde à media dos dois CPUs existentes na máquina. Os valores mostrados para o *StreamInsight* são a média dos valores do gráfico da figura 5.5, os valores do *CmStream* a média dos valores do gráfico da figura 5.9, os valores mostrados para o *MongoDB* são a média dos valores do gráfico da figura 5.13 e os valores mostrados para o *Redis* são a média dos valores do gráfico da figura 5.15.

Tabela 5.2: Vantagens e desvantagens dos vários sistemas estudados neste caso de estudo.

Classe	Sistema	Vantagens	Desvantagens	Velocidade (eventos/s)
RDBMS	*PostgreSQL*	- Sistema familiar às empresas - Escrita de *queries* em SQL - *Open Source* - *Cross-plataform*	- Não consegue realizar o processamento de eventos em tempo real	416,75
CEP	*StreamInsight*	- Processamento em tempo real (mais rápido que uma base de dados) - Capacidade de efetuar agregações complexas sobre eventos que ocorrem no tempo - Distribuído como um conjunto de ficheiros ".dll" permitindo a sua fácil utilização em programas escritos em C#	- Não consegue processar eventos para os quais não conheça à priori o *payload* - A versão gratuita não consegue recuperar as *queries* caso o sistema vá abaixo inesperadamente - *Closed Source* - Só funciona em sistemas Windows	5424,38
	CmStream	- Processamento em tempo real - Processa eventos sem conhecer os seus *payloads* à priori - Capacidade de efetuar agregações complexas sobre eventos que ocorrem no tempo	- Mais lento que o *StreamInsight* - Conjunto reduzido de operadores disponíveis durante a execução - Só funciona em sistemas Windows	3122,28
NOSQL	*MongoDB*	- Mais rápido que o *CmStream* a processar eventos em tempo real - *Open Source* - *Cross-plataform*	- Dificuldade em fazer *debugging* às funções utilizadas no *Map/Reduce* - Não tem suporte nativo para fazer agregações temporais - Mais lento que o *StreamInsight*	3697,33
	Redis	- Fácil de usar - Processamento em memória - *Open Source* - *Cross-plataform*	- *Single-threaded* - Não tem suporte nativo para fazer agregações temporais	2586,47

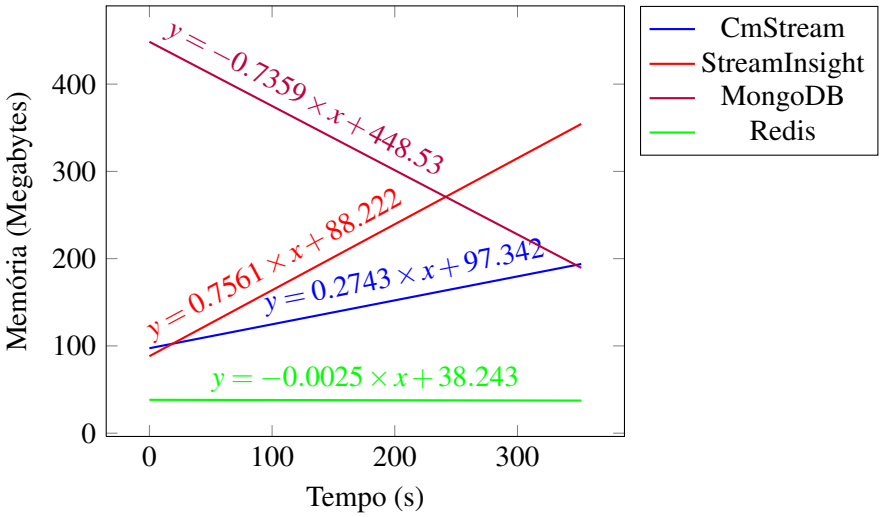

Figura 5.19: Relação entre o o tempo de execução e o consumo de memória das implementações *StreamInsight*, *CmStream*, *MongoDB* e *Redis*. Os valores mostrados para o *StreamInsight* foram obtidos através do cálculo da regressão linear da média dos valores mostrados nos gráficos 5.6 e 5.8. Os valores mostrados para o *CmStream* foram obtidos através do cálculo da regressão linear da média dos valores mostrados nos gráficos 5.10 e 5.12. Os valores mostrados para o *MongoDB* e *Redis* foram obtidos através do cálculo da regressão linear dos valores mostrados, respetivamente, nos gráficos 5.14 e 5.16.

Capítulo 6

Conclusões e trabalho futuro

Concluiu-se a partir dos testes experimentais que a utilização de sistemas de *Complex Event Processing* permite realizar com melhor desempenho certas análises que requerem agregações temporais quando comparada com o desempenho de um sistema de base de dados, tendo assim um desempenho semelhante a alguns sistemas NoSQL. Concluiu-se também que apesar o desempenho do *CmStream* para esse tipo de tarefas ser inferior ao desempenho em *StreamInsight*, esta continua a ser superior ao desempenho de um sistema de base de dados.

Um dos sistemas comerciais de *Complex Event Processing* é o *StreamInsight*. Foi visto que este um dos problemas que este tem é a necessidade de ter os eventos todos definidos à priori antes de ser executado.

Procurou-se então com este trabalho desenvolver uma arquitetura que eliminasse esse problema do *StreamInsight*. O resultado desse trabalho foi o *CmStream*. A arquitetura foi validada recorrendo a um caso de estudo que consistia na análise de *logs* de máquinas, onde foi visto que apesar de o *CmStream* ser mais lento que uma solução implementada puramente sobre o *StreamInsight*, este era mais rápido que uma solução implementada sobre uma base de dados.

No entanto existem ainda alguns problemas com o *CmStream* que poderão vir a ser removidos no futuro.

Um dos problemas do *CmStream* é que tal como o *StreamInsight* não é possível definir operadores em "*run-time*". Isso obriga a que sempre que se queira adicionar um novo operador, se tenha que terminar a aplicação para adicionar esse operador e depois arrancar de novo a aplicação. De notar que o *StreamInsight* partilha também este problema com o *CmStream*.

Outro problema do *CmStream* prende-se com a leitura de eventos a partir de um *socket*. Neste momento o *CmStream* requer que o XML do evento seja enviado todo no mesmo *buffer* do *socket*. Caso o XML do evento seja maior que o tamanho desse *buffer*, ele é enviado em várias partes. Este comportamento acontece ao nível do sistema operativo, não sendo possível modificar o tamanho desse *buffer* através do *CmStream*. Atualmente o tamanho máximo que a *string* XML pode ter são 8192 bytes, sendo que *strings* maiores que isso são divididas pelo sistema operativo. Uma forma de processar esses eventos será através da criação de um UDSO que guarde as *strings* que surjam

entre a *substring* “<xml>” e “</xml>” fazendo com que o evento só seja emitido quando surgir a *substring* “</xml>”.

Outra adição possível ao *CmStream* seria permitir definir alguns *payloads* de eventos em “*compile-time*” como o *StreamInsight* faz. Isto permitiria ter uma arquitetura híbrida que aproveitasse algumas funcionalidades do *StreamInsight*, como o menor tempo de execução, com as funcionalidades do *CmStream*, como a criação de eventos em “*run-time*”.

Na análise dos resultados experimentais do caso de estudo não foi analisada nenhuma base de dados relacional que permitisse guardar as tabelas em memória. Como o *PostgreSQL* requer a escrita e leitura dos dados para o disco rígido, isto faz com que o acesso aos dados seja mais lento do que o *StreamInsight* e o *CmStream*, que guardam os dados em memória, como se pode ver pela figura 5.3. Seria portanto importante analisar o impacto que teria guardar essa tabela em memória no desempenho dos sistemas de base de dados.

Parte do trabalho desenvolvido nesta dissertação foi apresentado na conferência *CISTI’2013* [RA13].

Anexo A

Gramáticas para os eventos de configuração de XML

Nesta secção encontra-se a especificação da gramática aceites pelos eventos de XML descritos na secção 4.3 e 4.5. Estas gramáticas estão descritas na forma “Backus-Naur”.

A.1 Gramática para os eventos de configuração de Hopping Windows

```
<xml> ::= "<xml><Field Name='event'>config</Field>"
          "<Field Name='queryType'>hopping</Field>"
          <window-size> <window-jump> <operation> <query-id> <input>
          <output> [<filter-expression>] [<real-time>]
          [<grace-period>]
          [<time-difference-extra-information>] "</xml>"
<window-size> ::= <time-size-units> <time-size-value>
<window-jump> ::= <time-jump-units> <time-jump-value>
<time-size-units> ::= "<Field Name='timeSizeUnits'>"
                      <time-unit> "</Field>"
<time-size-value> ::= "<Field Name='timeSizeValue'>"
                       number "</Field>"
<time-jump-units> ::= "<Field Name='timeJumpUnits'>"
                       <time-unit> "</Field>"
<time-jump-value> ::= "<Field Name='timeJumpValue'>"
                       number "</Field>"
<time-unit> ::= "Days" | "Hours" | "Minutes" | "Seconds"
                  | "Milliseconds" | "Ticks"
<operation> ::= <operation-name> <operation-arguments>
<operation-name> ::= "<Field Name='operation'>"
                     <operations> "</Field>"
<operation-arguments> ::= "<Field Name='operationArguments'>"
                          string "</Field>"
```

```
<operations> ::= "average" | "sum" | "stddev" | "count"
                   | "filter" | "timeDifference"
<query-id> ::= "<Field Name='queryId'>" string "</Field>"
<input> ::= <input-source> <input-arguments>
<input-source> ::= "<Field Name='inputType'>" <input-type> "</Field>"
<input-arguments> ::= "<Field Name='inputArguments'>"
                      string "</Field>"
<input-type> ::= "socket" | "file"
<output> ::= <output-source> <output-arguments>
<output-source> ::= "<Field Name='outputType'>"
                    <output-type> "</Field>"
<output-arguments> ::= "<Field Name='outputArguments'>"
                       string "</Field>"
<output-type> ::= "socket" | "file" | "console"
<filter-expression> ::= "<Field Name='filterExpression'>"
                        string "</Field>"
<real-time> ::= <is-real-time> <refresh-frequency>
<is-real-time> ::= "<Field Name='isRealTime'>" bool "</Field>"
<refresh-frequency> ::= <refresh-frequency-units>
                        <refresh-frequency-value>
<refresh-frequency-units> ::=
              "<Field Name='refreshFrequencyUnits'>"
              <time-unit> "</Field>"
<refresh-frequency-value> ::= "<Field Name='refreshFrequencyValue'>"
                              number "</Field>"
<grace-period> ::= <grace-period-units> <grace-period-value>
<grace-period-units> ::= "<Field Name='gracePeriodUnits'>"
                         <time-unit> "</Field>"
<grace-period-value> ::= "<Field Name='gracePeriodValue'>"
                         number "</Field>"
<time-difference-extra-information> ::= <start-event>
                                        <end-event> <timeout>
<start-event> ::= "<Field Name='filterStartEvent'>"
                  string "</Field>"
<end-event> ::= "<Field Name='filterEndEvent'>"
                string "</Field>"
<timeout> ::= <timeout-units> <timeout-value>
<timeout-units> ::= "<Field Name='timeoutUnits'>"
                    <time-unit> "</Field>"
<timeout-value> ::= "<Field Name='timeoutValue'>"
                    number "</Field>"
```

A.2 Gramática para os eventos de configuração de Tumbling Windows

```
<xml> ::= "<xml><Field Name='event'>config</Field>"
          "<Field Name='queryType'>tumbling</Field>"
          <window-size> <operation> <query-id>
          <input> <output> [<filter-expression>] [<real-time>]
          [<grace-period>]
          [<time-difference-extra-information>] "</xml>"
<window-size> ::= <time-size-units> <time-size-value>
<time-size-units> ::= "<Field Name='timeSpanUnits'>"
                      <time-unit> "</Field>"
<time-size-value> ::= "<Field Name='timeSpanValue'>"
                      number "</Field>"
<time-unit> ::= "Days" | "Hours" | "Minutes" | "Seconds"
                | "Milliseconds" | "Ticks"
<operation> ::= <operation-name> <operation-arguments>
<operation-name> ::= "<Field Name='operation'>"
                     <operations> "</Field>"
<operation-arguments> ::= "<Field Name='operationArguments'>"
                          string "</Field>"
<operations> ::= "average" | "sum" | "stddev" | "count"
                 | "filter" | "timeDifference"
<query-id> ::= "<Field Name='queryId'>" string "</Field>"
<input> ::= <input-source> <input-arguments>
<input-source> ::= "<Field Name='inputType'>"
                   <input-type> "</Field>"
<input-arguments> ::= "<Field Name='inputArguments'>"
                      string "</Field>"
<input-type> ::= "socket" | "file"
<output> ::= <output-source> <output-arguments>
<output-source> ::= "<Field Name='outputType'>"
                    <output-type> "</Field>"
<output-arguments> ::= "<Field Name='outputArguments'>"
                       string "</Field>"
<output-type> ::= "socket" | "file" | "console"
<filter-expression> ::= "<Field Name='filterExpression'>"
                        string "</Field>"
<real-time> ::= <is-real-time> <refresh-frequency>
<is-real-time> ::= "<Field Name='isRealTime'>" bool "</Field>"
<refresh-frequency> ::= <refresh-frequency-units>
                        <refresh-frequency-value>
<refresh-frequency-units> ::=
                      "<Field Name='refreshFrequencyUnits'>"
                      <time-unit> "</Field>"
<refresh-frequency-value> ::=
                      "<Field Name='refreshFrequencyValue'>"
```

```
                        number "</Field>"
<grace-period> ::= <grace-period-units> <grace-period-value>
<grace-period-units> ::= "<Field Name='gracePeriodUnits'>"
                           <time-unit> "</Field>"
<grace-period-value> ::= "<Field Name='gracePeriodValue'>"
                           number "</Field>"
<time-difference-extra-information> ::= <start-event>
                                        <end-event>
                                        <timeout>
<start-event> ::= "<Field Name='filterStartEvent'>"
                    string "</Field>"
<end-event> ::= "<Field Name='filterEndEvent'>"
                  string "</Field>"
<timeout> ::= <timeout-units> <timeout-value>
<timeout-units> ::= "<Field Name='timeoutUnits'>"
                      <time-unit> "</Field>"
<timeout-value> ::= "<Field Name='timeoutValue'>"
                      number "</Field>"
```

A.3 Gramática para os eventos de configuração de Count Windows

```
<xml> ::= "<xml><Field Name='event'>config</Field>"
          "<Field Name='queryType'>count</Field>"
          <window-size> <operation> <query-id>
          <input> <output> [<filter-expression>] [<real-time>]
          [<grace-period>]
          [<time-difference-extra-information>] "</xml>"
<window-size> ::= "<Field Name='elementSize'>"
                    number "</Field>"
<time-size-units> ::= "<Field Name='timeSpanUnits'>"
                        <time-unit> "</Field>"
<time-size-value> ::= "<Field Name='timeSpanValue'>"
                        number "</Field>"
<time-unit> ::= "Days" | "Hours" | "Minutes" | "Seconds"
                | "Milliseconds" | "Ticks"
<operation> ::= <operation-name> <operation-arguments>
<operation-name> ::= "<Field Name='operation'>"
                       <operations> "</Field>"
<operation-arguments> ::= "<Field Name='operationArguments'>"
                            string "</Field>"
<operations> ::= "average" | "sum" | "stddev" | "count"
                 | "filter" | "timeDifference"
<query-id> ::= "<Field Name='queryId'>" string "</Field>"
<input> ::= <input-source> <input-arguments>
<input-source> ::= "<Field Name='inputType'>"
                     <input-type> "</Field>"
```

```
<input-arguments> ::= "<Field Name='inputArguments'>"
                      string "</Field>"
<input-type> ::= "socket" | "file"
<output> ::= <output-source> <output-arguments>
<output-source> ::= "<Field Name='outputType'>"
                    <output-type> "</Field>"
<output-arguments> ::= "<Field Name='outputArguments'>"
                       string "</Field>"
<output-type> ::= "socket" | "file" | "console"
<filter-expression> ::= "<Field Name='filterExpression'>"
                        string "</Field>"
<real-time> ::= <is-real-time> <refresh-frequency>
<is-real-time> ::= "<Field Name='isRealTime'>" bool "</Field>"
<refresh-frequency> ::= <refresh-frequency-units>
                        <refresh-frequency-value>
<refresh-frequency-units> ::=
                        "<Field Name='refreshFrequencyUnits'>"
                        <time-unit> "</Field>"
<refresh-frequency-value> ::=
                        "<Field Name='refreshFrequencyValue'>"
                        number "</Field>"
<grace-period> ::= <grace-period-units> <grace-period-value>
<grace-period-units> ::= "<Field Name='gracePeriodUnits'>"
                         <time-unit> "</Field>"
<grace-period-value> ::= "<Field Name='gracePeriodValue'>"
                         number "</Field>"
<time-difference-extra-information> ::= <start-event>
                                        <end-event> <timeout>
<start-event> ::= "<Field Name='filterStartEvent'>"
                  string "</Field>"
<end-event> ::= "<Field Name='filterEndEvent'>"
                string "</Field>"
<timeout> ::= <timeout-units> <timeout-value>
<timeout-units> ::= "<Field Name='timeoutUnits'>"
                    <time-unit> "</Field>"
<timeout-value> ::= "<Field Name='timeoutValue'>"
                    number "</Field>"
```

A.4 Gramática para os eventos de configuração de Snapshot Windows

```
<xml> ::= "<xml><Field Name='event'>config</Field>"
          "<Field Name='queryType'>snapshot</Field>"
          <operation> <query-id> <input> <output>
          [<filter-expression>] [<real-time>] [<grace-period>]
```

```
          [<time-difference-extra-information>] "</xml>"
<time-unit> ::= "Days" | "Hours" | "Minutes" | "Seconds"
                | "Milliseconds" | "Ticks"
<operation> ::= <operation-name> <operation-arguments>
<operation-name> ::= "<Field Name='operation'>"
                     <operations> "</Field>"
<operation-arguments> ::= "<Field Name='operationArguments'>"
                          string "</Field>"
<operations> ::= "average" | "sum" | "stddev" | "count"
                 | "filter" | "timeDifference"
<query-id> ::= "<Field Name='queryId'>" string "</Field>"
<input> ::= <input-source> <input-arguments>
<input-source> ::= "<Field Name='inputType'>"
                   <input-type> "</Field>"
<input-arguments> ::= "<Field Name='inputArguments'>"
                      string "</Field>"
<input-type> ::= "socket" | "file"
<output> ::= <output-source> <output-arguments>
<output-source> ::= "<Field Name='outputType'>"
                    <output-type> "</Field>"
<output-arguments> ::= "<Field Name='outputArguments'>"
                       string "</Field>"
<output-type> ::= "socket" | "file" | "console"
<filter-expression> ::= "<Field Name='filterExpression'>"
                        string "</Field>"
<real-time> ::= <is-real-time> <refresh-frequency>
<is-real-time> ::= "<Field Name='isRealTime'>" bool "</Field>"
<refresh-frequency> ::= <refresh-frequency-units>
                        <refresh-frequency-value>
<refresh-frequency-units> ::=
                        "<Field Name='refreshFrequencyUnits'>"
                        <time-unit> "</Field>"
<refresh-frequency-value> ::=
                        "<Field Name='refreshFrequencyValue'>"
                        number "</Field>"
<grace-period> ::= <grace-period-units> <grace-period-value>
<grace-period-units> ::= "<Field Name='gracePeriodUnits'>"
                         <time-unit> "</Field>"
<grace-period-value> ::= "<Field Name='gracePeriodValue'>"
                         number "</Field>"
<time-difference-extra-information> ::= <start-event>
                                        <end-event>
                                        <timeout>
<start-event> ::= "<Field Name='filterStartEvent'>"
                  string "</Field>"
<end-event> ::= "<Field Name='filterEndEvent'>"
                string "</Field>"
<timeout> ::= <timeout-units> <timeout-value>
<timeout-units> ::= "<Field Name='timeoutUnits'>"
```

```
                    <time-unit> "</Field>"
<timeout-value> ::= "<Field Name='timeoutValue'>"
                    number "</Field>"
```

A.5 Gramática para os eventos de configuração de Session Windows

```
<xml> ::= "<xml><Field Name='event'>config</Field>"
          "<Field Name='queryType'>session</Field>"
          <event-start> <event-end> <timeout> <operation>
          <query-id> <input> <output> [<filter-expression>]
          [<real-time>] [<grace-period>] "</xml>"
<event-start> ::= "<Field Name='eventStart'>"
                  string "</Field>"
<event-end> ::= "<Field Name='eventEnd'>" string "</Field>"
<timeout> ::= <timeout-units> <timeout-value>
<timeout-units> ::= "<Field Name='timeoutUnits'>"
                    <time-unit> "</Field>"
<timeout-value> ::= "<Field Name='timeoutValue'>"
                    number "</Field>"
<time-unit> ::= "Days" | "Hours" | "Minutes" | "Seconds"
                | "Milliseconds" | "Ticks"
<operation> ::= <operation-name> <operation-arguments>
<operation-name> ::= "<Field Name='operation'>"
                     <operations> "</Field>"
<operation-arguments> ::= "<Field Name='operationArguments'>"
                          string "</Field>"
<operations> ::= "average" | "sum" | "stddev" | "count"
                 | "filter" | "timeDifference"
<query-id> ::= "<Field Name='queryId'>" string "</Field>"
<input> ::= <input-source> <input-arguments>
<input-source> ::= "<Field Name='inputType'>"
                   <input-type> "</Field>"
<input-arguments> ::= "<Field Name='inputArguments'>"
                      string "</Field>"
<input-type> ::= "socket" | "file"
<output> ::= <output-source> <output-arguments>
<output-source> ::= "<Field Name='outputType'>"
                    <output-type> "</Field>"
<output-arguments> ::= "<Field Name='outputArguments'>"
                       string "</Field>"
<output-type> ::= "socket" | "file" | "console"
<filter-expression> ::= "<Field Name='filterExpression'>"
                        string "</Field>"
<real-time> ::= <is-real-time> <refresh-frequency>
<is-real-time> ::= "<Field Name='isRealTime'>" bool "</Field>"
<refresh-frequency> ::= <refresh-frequency-units>
```

```
                        <refresh-frequency-value>
<refresh-frequency-units> ::=
                        "<Field Name='refreshFrequencyUnits'>"
                        <time-unit> "</Field>"
<refresh-frequency-value> ::=
                        "<Field Name='refreshFrequencyValue'>"
                        number "</Field>"
<grace-period> ::= <grace-period-units> <grace-period-value>
<grace-period-units> ::= "<Field Name='gracePeriodUnits'>"
                          <time-unit> "</Field>"
<grace-period-value> ::= "<Field Name='gracePeriodValue'>"
                          number "</Field>"
```

Anexo B

Plano de execução da solução PostgreSQL

O plano de execução da figura 5.2 pode ser visto em maior detalhe utilizado uma *query* "EXPLAIN" na consola de "PSQL" fornecida pelo *PostgreSQL*. O seu *output* pode ser visto de seguida:

```
                                QUERY PLAN
-------------------------------------------------------------------------
GroupAggregate  (cost=113431490.94..117477462.71 rows=31800 width=27)
  InitPlan 2 (returns $1)
    ->  Result  (cost=0.06..0.07 rows=1 width=0)
          InitPlan 1 (returns $0)
            ->  Limit  (cost=0.00..0.06 rows=1 width=8)
                  ->  Index Only Scan using d25s on dataset25
                              (cost=0.00..8641.62 rows=149727 width=8)
                        Index Cond: (starttime IS NOT NULL)
  InitPlan 4 (returns $3)
    ->  Result  (cost=0.06..0.07 rows=1 width=0)
          InitPlan 3 (returns $2)
            ->  Limit  (cost=0.00..0.06 rows=1 width=8)
                  ->  Index Only Scan Backward using d25e on dataset25
                              (cost=0.00..8641.67 rows=149727 width=8)
                        Index Cond: (endtime IS NOT NULL)
  ->  Sort  (cost=113431490.81..114240621.56 rows=323652301 width=27)
        Sort Key: win.starttime, dataset25.machineid, win.endtime
        ->  Nested Loop  (cost=0.25..21221642.21 rows=323652301 width=27)
              Join Filter: "overlaps"
                            (dataset25.starttime, dataset25.endtime,
                             win.starttime, win.endtime)
              ->  Function Scan on generate_time_windows win
```

```
                (cost=0.25..10.25 rows=1000 width=16)
->  Materialize  (cost=0.00..28342.36 rows=970957 width=27)
      ->  Seq Scan on dataset dataset25
                (cost=0.00..16849.57 rows=970957 width=27)
```

Referências

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, e Jennifer Widom. Stream: The stanford data stream management system. Springer, 2004.

[ABW06] Arvind Arasu, Shivnath Babu, e Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, Junho 2006.

[ACSK10] Mohamed Ali, Badrish Chandramouli, Balan Sethu, e Raman Katibah. Spatio-temporal stream processing in microsoft StreamInsight. *Data Engineering*, Junho 2010.

[AGR+09] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, e I. Santos. Microsoft CEP server and online behavioral targeting. *Proceedings of the VLDB Endowment*, 2(2):1558–1561, Agosto 2009.

[CCW00] Stefano Ceri, Roberta Cochrane, e Jennifer Widom. Practical applications of triggers and constraints: Success and lingering issues (10-year award). In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pág. 254–262, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[CES04] D. Culler, D. Estrin, e M. Srivastava. Guest editors' introduction: Overview of sensor networks. *Computer*, 37(8):41–49, 2004.

[Cod] CodePlex. NCalc - mathematical expressions evaluator for .NET.

[dABB+08] A. dos Anjos, P. Bell, D. Berge, J. Haller, S. Head, Shumin Li, A. Hocker, T. Kono, T. McMahon, M. Nozicka, H. von der Schmitt, R. Spiwoks, J. Stelzer, T. Wengler, e W. Wiedenmann. The configuration system of the ATLAS trigger. *IEEE Transactions on Nuclear Science*, 55(1):392 –398, Fevereiro 2008.

[Dav11] David Rosenbaum. That new big data magic, Agosto 2011.

[DGG95] Klaus R. Dittrich, Stella Gatziu, e Andreas Geppert. The active database management system manifesto: A rulebase of ADBMS features. In Timos Sellis, editor, *Rules in Database Systems*, Lecture Notes in Computer Science, pág. 1–17. Springer Berlin Heidelberg, Janeiro 1995.

[EGA] E. Della Valle, G. Cugola, e A. Margara. Stream and complex event processing - introduction.

[Gar11] Gartner. Gartner says solving 'Big data' challenge involves more than just managing volumes of data, Junho 2011.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GHM+] Nikolaus Glombiewski, Bastian Hobach, Andreas Morgen, Franz Ritter, e Bernhard Seeger. Event processing on your own database.

[HBWN11] Yeye He, Siddharth Barman, Di Wang, e Jeffrey F. Naughton. On the complexity of privacy-preserving complex event processing. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pág. 165–174, New York, NY, USA, 2011. ACM.

[JAF+05] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, e Jennifer Widom. Virtual devices: An extensible architecture for bridging the physical-digital divide. Relatório técnico, Computer Science Division, University of California at Berkeley, Berkeley, California, Março 2005.

[JAF+06] S.R. Jeffery, G. Alonso, M.J. Franklin, Wei Hong Hong, e J. Widom. A pipelined framework for online cleaning of sensor data streams. In *Proceedings of the 22nd International Conference on Data Engineering, 2006. ICDE '06*, pág. 140–140, 2006.

[JMS+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, e Stan Zdonik. Towards a streaming SQL standard. *Proc. VLDB Endow.*, 1(2):1379–1390, Agosto 2008.

[LWZ11] Yan-Nei Law, Haixun Wang, e Carlo Zaniolo. Relational languages and data models for continuous queries on sequences and data streams. *ACM Trans. Database Syst.*, 36(2):8:1–8:32, Junho 2011.

[LZLL10] Yongxuan Lai, Wenhua Zeng, Ziyu Lin, e Guilin Li. LAMF: framework for complex event processing in wireless sensor networks. In *2010 2nd International Conference on Information Science and Engineering (ICISE)*, pág. 2155 –2158, Dezembro 2010.

[Mic10] Microsoft Corporation. C# language specification, 2010.

[MSDa] MSDN. Choosing a StreamInsight edition.

[MSDb] MSDN. Creating event types.

[MSDc] MSDN. IDisposable interface (system).

[MSDd] MSDN. Input and output adapters (legacy model).

[MSDe] MSDN. Reactive extensions.

[MSDf] MSDN. StreamInsight resiliency.

[MSDg] MSDN. User-defined aggregates and operators.

[MSDh] MSDN. User-defined functions (StreamInsight).

[MSDi] MSDN. User-defined stream operators.

[MSDj] MSDN. Using StreamInsight LINQ.

[MSDk] MSDN. What's new (StreamInsight).

[MSS+06] P. J. Marrón, R. Sauter, O. Saukh, M. Gauger, e K. Rothermel. Challenges of complex data processing in real world sensor network deployments. In *In: Proceedings of the ACM Workshop on Real-World Wireless Sensor Networks (REALWSN'06*, pág. 43–48, 2006.

[MTC+11] Baljeet Malhotra, Wee-Juan Tan, Jianneng Cao, Thomas Kister, Stéphane Bressan, e Kian-Lee Tan. ASSIST: access controlled ship identification streams. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pág. 485–488, New York, NY, USA, 2011. ACM.

[RA13] Rolando Pereira e Armando Sousa. Otimização de fluxo empresarial de dados - necessidade de ferramentas de complex event processing. In *Atas da 8ª Conferência Ibérica de Sistemas e Tecnologias de Informação*, CISTI'2013, Lisboa, Portugal, Junho 2013.

[RAK+10] Alex Raizman, Asvin Ananthanarayan, Anton Kirilov, Badrish Chandramouli, e Mohamed Ali. An extensible test framework for the microsoft StreamInsight query processor. In *Proceedings of the Third International Workshop on Testing Database Systems*, DBTest '10, pág. 2:1–2:6, New York, NY, USA, 2010. ACM.

[Riz05] Shariq Rizvi. Complex event processing beyond active databases: Streams and uncertainties. Relatório técnico, Technical Report UCB/EECS-2005-26, Electrical Engineering and Computer Sciences Department, University of California at Berkeley, 2005. 54, 84, 86, 2005.

[RJA12] Ramkumar Krishnan, Jonathan Goldstein, e Alex Raizman. A hitchhiker's guide to StreamInsight 2.1 queries - microsoft StreamInsight - site home - MSDN blogs, Junho 2012.

[RSS12] Dominik Riemer, Ljiljana Stojanovic, e Nenad Stojanovic. Using complex event processing for modeling semantic requests in real-time social media monitoring. In *Sixth International AAAI Conference on Weblogs and Social Media*, Maio 2012.

[RZS+12] Tilmann Rabl, Kaiwen Zhang, Mohammad Sadoghi, Navneet Kumar Pandey, Aakash Nigam, Chen Wang, e Hans-Arno Jacobsen. Solving manufacturing equipment monitoring through efficient complex event processing: DEBS grand challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pág. 335–340, New York, NY, USA, 2012. ACM.

[SD95] Eric Simon e Angelika Kotz Dittrich. Promises and realities of active database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pág. 642–653, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[SGA+12] Marc Schaaf, Stella Gatziu Grivas, Dennie Ackermann, Arne Diekmann, Arne Koschel, e Irina Astrova. Semantic complex event processing. In *Proceedings of the 5th WSEAS congress on Applied Computing conference, and Proceedings of the 1st international conference on Biologically Inspired Computation*, BICA'12, pág. 38–43, Stevens Point, Wisconsin, USA, 2012. World Scientific and Engineering Academy and Society (WSEAS).

[SUZ05] Michael Stonebraker, Uğur Çetintemel, e Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, Dezembro 2005.

[SW04] Utkarsh Srivastava e Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '04, pág. 263–274, New York, NY, USA, 2004. ACM.

[TGP05] Bob Thome, Dieter Gawlick, e Maria Pratt. Event processing with an oracle database. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pág. 863–867, New York, NY, USA, 2005. ACM.

[TM03] Peter A. Tucker e David Maier. Dealing with disorder. *MPDS, June*, 2003.

[WBC+09] E. Welbourne, L. Battle, G. Cole, K. Gould, K. Rector, S. Raymer, M. Balazinska, e G. Borriello. Building the internet of things using RFID: the RFID ecosystem experience. *IEEE Internet Computing*, 13(3):48–55, 2009.

[WHRN13] Di Wang, Yeye He, Elke Rundensteiner, e Jeffrey Naughton. Utility-maximizing event stream suppression. 2013.

[WRWE10] Di Wang, Elke A. Rundensteiner, Han Wang, e Richard T. Ellison,III. Active complex event processing: applications in real-time health care. *Proc. VLDB Endow.*, 3(1-2):1545–1548, Setembro 2010.

[ZU99] Detlef Zimmer e Rainer Unland. On the semantics of complex events in active database management systems. pág. 392–399. IEEE Computer Society Press, 1999.

[ZWCC07] Fred Zemke, Andrew Witkowski, Mitch Cherniak, e Latha Colby. Pattern matching in sequences of rows (11). *Mar*, 21:1–30, Março 2007.