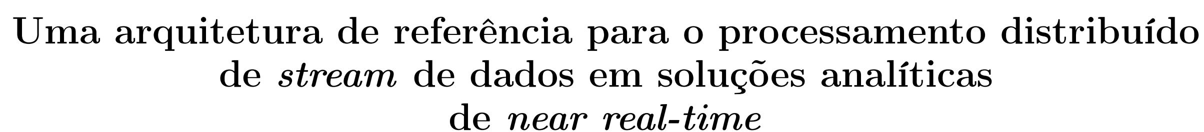

Uma arquitetura de referência para o processamento distribuído de *stream* de dados em soluções analíticas de *near real-time*

DANIEL DA CUNHA RODRIGUES DE SOUZA

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASILIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Uma arquitetura de referência para o processamento distribuído de *stream* de dados em soluções analíticas de *near real-time*

DANIEL DA CUNHA RODRIGUES DE SOUZA

ORIENTADOR: RAFAEL TIMÓTEO DE SOUSA JÚNIOR
COORIENTADOR: EDISON PIGNATON DE FREITAS

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Brasília, Março de 2015

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

UMA ARQUITETURA DE REFERÊNCIA PARA O PROCESSAMENTO DISTRIBUÍDO DE STREAM DE DADOS EM SOLUÇÕES ANALÍTICAS DE NEAR REAL-TIME

DANIEL DA CUNHA RODRIGUES

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADA POR:

RAFAEL TIMÓTEO DE SOUSA JÚNIOR, Dr., ENE/UNB
(ORIENTADOR)

JOÃO PAULO CARVALHO LUSTOSA DA COSTA, Dr., ENE/UNB
(EXAMINADOR INTERNO)

ROBSON DE OLIVEIRA ALBUQUERQUE Dr., Abin
(EXAMINADOR EXTERNO)

Brasília, 20 de março de 2015.

FICHA CATALOGRÁFICA

SOUZA, DANIEL DA CUNHA RODRIGUES
Uma arquitetura de referência para o processamento distribuído de *stream* de dados em soluções analíticas de *near real-time*. [Distrito Federal] 2015.

x, 94p., 297 mm (FT/UnB, Mestre, 2015). Dissertação de Mestrado – Universidade de Brasília.Faculdade de Tecnologia.

1. *Business Intelligence*
2. *Big Data*
3. Arquitetura de software

I. ENE/FT/UnB

REFERÊNCIA BIBLIOGRÁFICA

RODRIGUES, D. C., (2015). Uma arquitetura de referência para o processamento distribuído de *stream* de dados em soluções analíticas de *near real-time*. Dissertação de Mestrado em Engenharia Elétrica, Publicação PPGEE.DM-603/2015, Faculdade de Tecnologia, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 94wp.

CESSÃO DE DIREITOS

AUTOR: Daniel da Cunha Rodrigues de Souza

TÍTULO: Uma arquitetura de referência para o processamento distribuído de *stream* de dados em soluções analíticas de *near real-time*.

GRAU: Mestre ANO: 2015

É concedida à Universidade de Brasília permissão para reproduzir cópias deste Trabalho de Pós-Graduação e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desse Trabalho de Pós-Graduação pode ser reproduzida sem autorização por escrito do autor.

Daniel da Cunha Rodrigues de Souza
Quadra 15, conjunto B, Casa 17, Sobradinho
Brasília - DF - Brasil - 73045-152

Dedicatória

À minha amada esposa Diana e ao meu querido filho Daniel.

Daniel da Cunha Rodrigues de Souza

Agradecimentos

Em primeiro lugar à Deus porque Ele sempre foi minha força e inspiração.
À toda minha maravilhosa família por toda paz e força necessárias para a conclusão dessa etapa.
Ao meu professor orientador, Dr. Rafael Timóteo Sousa Júnior, pelo apoio e pelo incentivo durante toda a pesquisa e que pacientemente esclareceu diversas dúvidas pertinentes a esta dissertação.
Aos professores Edison Pignaton de Freitas e João Paulo Carvalho Lustosa da Costa por toda a paciência, orientação e amizade, essenciais para a evolução do trabalho.
Enfim, agradeço a todos que direta ou indiretamente contribuíram para a realização desse trabalho. Apesar de não mencioná-los por nome, saibam que foram fundamentais para o desenvolvimento e conclusão dessa etapa.

Daniel da Cunha Rodrigues de Souza

RESUMO

UMA ARQUITETURA DE REFERÊNCIA PARA O PROCESSAMENTO DISTRIBUÍDO DE *STREAM* DE DADOS EM SOLUÇÕES ANALÍTICAS DE *NEAR REAL-TIME*

Autor: Daniel da Cunha Rodrigues de Souza

Orientador: Rafael Timóteo de Sousa Júnior

Coorientador: Edison Pignaton de Freitas

Programa de Pós-graduação em Engenharia Elétrica

Brasília, Março de 2015

Os novos requisitos para o processamento em baixa latência de *streams* de dados distribuídos desafiam as arquiteturas tradicionais de processamento de dados. Uma nova classe de sistemas denominados *Distributed Stream Processing Systems* (DSPS) emergiram para facilitar a analise desses dados em baixa latência. Entretanto, a diversidade de arquiteturas, modelos de processamento e *Application Programming Interfaces* (APIs) nesses DSPSs aumentaram a complexidade no processo de desenvolvimento de sistemas para o processamento de dados. Nesse contexto, este trabalho propõe uma arquitetura de referência para o processamento de *streams* para soluções analíticas de *near real-time*. Essa arquitetura tem como base conceitos arquiteturais que estabelecem uma separação em camadas com responsabilidades bem definidas, resultando em um modelo de referência que promove o reuso de decissões de projeto e suporta a gestão da complexidade no desenvolvimento de sistemas de processamento de *stream* de dados. Para validar a solução proposta, essa arquitetura de referência é instanciada em um experimento que aborda o uso de dois algoritmos algorítimos probabílisticos: HyperLogLog e Count-Min Sketch.

Palavras Chave: *Business Intelligence*, *Internet of Things*, *Big Data*, Arquitetura de Software, Processamento distribuído, Padrões de projeto

ABSTRACT

A REFERENCE ARCHITECTURE FOR DISTRIBUTED PROCESSING STREAMS OF DATA FOR NEAR REAL-TIME ANALYTICS

Author: Daniel da Cunha Rodrigues de Souza

Supervisor: Rafael Timóteo de Sousa Júnior

Co-Supervisor: Edison Pignaton de Freitas

Programa de Pós-graduação em Engenharia Elétrica

Brasília, March of 2015

The current requirement of low latency processing for high volume of data streams is pushing the limits of the traditional data processing architectures. A new class of applications called Distributed Stream Processing Systems (DSPS) has emerged to facilitate such large scale real time data analytics. Nevertheless the diversity of architectures, data models and APIs introduced by the use of these systems resulted in a greater complexity to the development of data processing systems. In this context, a reference architecture to data stream processing for near real-time analytics is proposed in this work. This proposal is based on a layered architecture pattern, with clearly defined responsibilities providing a strong reference model, to improve the maintainability and reuse for data stream processing systems. In order to evaluate the proposed architecture and its framework, a case study is used in which two probabilistic algorithms are applied: the HyperLogLog and the Count-Min Sketch.

Keywords: Business Intelligence, Internet of Things, Big Data, Software architecture, Distributed processing, Design patterns

SUMÁRIO

1 Introdução **1**
1.1 Objetivos 4
1.2 Contribuições 4
1.3 Organização do Trabalho 4

2 Trabalhos Relacionados **6**

3 Arquitetura de Software **9**
3.1 Introdução 9
3.2 Descrição arquitetural 10
3.3 Estruturas de sistemas de software 11
3.4 Padrões Arquiteturais 13
3.5 Arquitetura de referência 15

4 Big Data **17**
4.1 Tecnologias de *Big Data* 20
4.2 Bancos de dados NoSQL 21
4.3 *Stream* de dados 22
4.4 Algoritmos probabilísticos 26
4.4.1 HyperLogLog 27
4.4.2 Count-min Sketch 28

5 *Business Intelligence* (BI) e *Business Analytics* (BA) **31**
5.1 Introdução 31
5.2 *Data Warehouse* 33
5.3 Modelagem dimensional 34
5.4 *Extract, Transform and Load* (ETL) 35

6 Solução proposta **38**
6.1 Análise dos DSPS 39
6.1.1 Borealis 39
6.1.2 Apache Storm 40
6.1.3 Apache Samza 41
6.2 Arquitetura de referência 42

6.2.1 Modelo de processamento ... 43
6.2.2 Camada de sistemas de processamento ... 44
6.2.3 Camada de *Middleware* de processamento ... 45
6.2.4 Camada de Infraestrutura ... 45

7 Resultados experimentais ... 46
7.1 Introdução ... 46
7.2 Arquitetura do experimento ... 47
7.2.1 Visão geral ... 47
7.2.2 Visão de fluxo de dados ... 50
7.2.3 Visão de distribuição ... 51
7.3 Análise dos DSPS ... 52
7.4 Análise de algorítimos para o processamento de *stream* de dados ... 53
7.5 Padrões de projeto identificados ... 56

8 Conclusões ... 58

REFERÊNCIAS BIBLIOGRÁFICAS ... 61

TRABALHOS PUBLICADOS PELO AUTOR ... 68

Anexos ... 69

I Códigos-fonte produzidos ... 70
I.1 Produtor de dados ... 70
I.2 Algoritmo convencional para contagem de elementos distintos ... 72
I.3 Algoritmo HyperLogLog para a estimativa de elementos distintos . 72
I.4 Algoritmo convencional para contagem da frequência de elementos 73
I.5 Algoritmo Count-Min Sketch para a estimativa da frequência de elementos ... 74
I.6 Monitor de utilização de memória ... 75
I.7 Quantidade de endereços IP distintos agrupados por país ... 76

LISTA DE FIGURAS

1.1 Latência no processamento de dados para diferentes abordagens 2

3.1 Projeto de software na engenharia de software 10
3.2 Visão conceitual de uma descrição arquitetural [54] 12
3.3 Diagrama da UML que representa a composição de um padrão de projeto 14
3.4 Propósito de uma arquitetura de referência 16
3.5 Relação entre contexto, objetivos e descrição de uma arquitetura de referência 16

4.1 Relação entre o crescimento no volume de dados e inovações no processamento de dados 17
4.2 Os 3Vs que definem *Big Data* 19
4.3 Teorema de CAP 22
4.4 Áreas do processamento de *stream* de dados 25
4.5 Representação lógica do processamento de *streams* de dados 26
4.6 Matriz de contagem do algoritmo Count-min sketch 29

5.1 Visão geral de BI 31
5.2 Fontes de dados operacionais e a orientação ao assunto 32
5.3 Fluxo de atividades para a concepção de uma solução de BI 34
5.4 Representação conceitual do modelo de dados dimensional do tipo *star schema* 34
5.5 Representação conceitual do modelo de dados dimensional do tipo *Snowflake schema* 35
5.6 Representação conceitual do modelo de dados dimensional do tipo *Fact constellation schema* 35
5.7 Processo de extração, transformação e carga de dados (ETL) 36

6.1 Proposta de áreas do processamento de *streams* sugerida pelo trabalho 39
6.2 Composição de um servidor Borealis [31] 40
6.3 Composição de um *cluster* Apache Storm [33] 40
6.4 Distribuição do Apache Samza em *cluster* Apache Hadoop YARN 41
6.5 Visão geral da arquitetura de referência 42
6.6 Diagrama da UML representando o metamodelo para modelagem de fluxos de processamento 43
6.7 Elementos básicos do modelo de processamento proposto 44

7.1 Visão geral da arquitetura do experimento 48

7.2 Elementos do modelo de processamento proposto 48
7.3 Fluxo de dados para o processamento de *streams* distribuídos 50
7.4 Diagrama da UML representando a distribuição dos elementos de *software streams* distribuídos 51
7.5 Análise do consumo de memória de cada estratégia para o processamento dos dados . 55
7.6 Padrão de projeto proposto para agrupamento e contagem de elementos em *stream* de dados 56

LISTA DE TABELAS

2.1 Comparativo entre os principais trabalhos relacionados e esta dissertação 8

4.1 Comparação entre o processamento *batch* e o processamento de *stream* de dados...... 23

7.1 Estrutura e amostra dos dados utilizados no experimento 46
7.2 Cenários e estratégias de processamento.. 47
7.3 Comparativo entre DSPS 52
7.4 Comparativo entre algoritmos para o processamento do cenário de frequência de um endereço IP 53
7.5 Comparativo entre algoritmos para o processamento do cenário de quantidade de endereços IPs distintos analisados 54
7.6 Descrição do padrão de projeto *Group and Count* 57

LISTA DE SÍMBOLOS

Siglas

BA	*Business Analytics*
BI	*Business Intelligence*
ETL	*Extract, Transform and Load*
ADL	*Architectural Description Language*
TI	Tecnologia da Informação
HDFS	*Hadoop Distributed File System*
GFS	*Google File System*
RDBMS	*Relational Database Management Systems*
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
CAP	*Consistency, Availability and Partition Tolerance*
DSPS	*Distributed Stream Processing Systems*
OLTP	*Online Transaction Processing application*
OLAP	*Online Analytical Processing*
RTOLAP	*Real-Time Online Analytical Processing*
DW	*Data Warehouse*
DM	*Data Mart*
WWW	*World Wide Web*
GFS	*Google File System*
BASE	*Basically Available, Soft-state and eventually consistent*
SQL	*Structured Query Language*
RM	*Resource Manager*
NM	*Node Manager*
PE	*Processing Element*
UML	*Unified Modeling Language*
API	*Application Programming Interface*
DM	*Data Marts*
IoT	*Internet of Things*
RPC	*Remote Procedure Call*

Capítulo 1

Introdução

A proliferação de dispositivos móveis ligados à Internet é um fenômeno evidente nos últimos anos. Essa adoção, em um nível global, impacta em diversos domínios de aplicação, como por exemplo: monitoramento de sinais vitais, sensores espalhados geograficamente, redes sociais, registros de operações financeiras, análises de segurança e auditoria de sistemas de informação. Entretanto, esse crescimento no número de dispositivos ligados à Internet, conhecidos como Internet das Coisas (*Internet of Things* - IoT), resulta na produção de grandes volumes de dados em diferentes formatos e em curto espaço de tempo [1].

Por exemplo, o processamento dos dados produzidos por sistemas de venda online, como a Amazon.com, pode produzir análises de comportamento e preferências de compra para recomendações de ofertas direcionadas para cada cliente. [2] [3].

A detecção de possíveis desastres naturais tem impacto direto na vida de milhares de pessoas. O tempo de resposta envolvido no processamento de dados de satélites e sensores distribuídos geograficamente diminuem o impacto de tais ameaças na vida dessas pessoas [4].

Na segurança pública, o uso de sensores tem aplicações fundamentais para identificar situações de risco. A análise em *near real-time* de dados produzidos por câmeras de monitoramento, sensores de movimento e áudio tem potencial utilidade na detecção de atividades suspeitas.

Na área de marketing digital, o processamento de quantidades de cliques recebidos por um anúncio permite analisar a eficiência de um anúncio em relação ao seu horário de exibição ou público alvo. Em grandes sites, o ajuste na estratégia de publicação desses anúncios, em um curto espaço de tempo, é fator crítico para o atingimento dos resultados esperados.

Esses são alguns exemplos de como o processamento de dados em baixa latência pode resultar em subsídios críticos para o processo de tomada de decisão.

Um recente relatório do *International Data Corporation* (IDC) [1] indica que, do ano de 2005 a 2020, o crescimento no volume global de dados produzidos será de 130 Exabytes para 40.000 Exabytes. Essa tendência supera as previsões de crescimento para a capacidade de processamento como, por exemplo, da conhecida lei de Moore [5]. Dessa maneira, o estudo de áreas da computação, como o processamento distribuído, é uma solução para superar as limitações encontradas no

processamento centralizado desses dados.

A análise em tempo hábil desses dados é de vital importância para o sucesso das empresas. Portanto, o processamento desse grande volume de dados é um dos principais desafios para a análise de dados.

Nesse contexto, sistemas de *Business Intelligence* (BI) visam disponibilizar os meios necessários para a transformação de dados em informação e divulgar essas informações por meio de ferramentas analíticas, a fim de subsidiar o processo decisório [6]. Tais soluções são compostas tipicamente de rotinas de Extração, Transformação e Carga, do inglês *Extract, Transform and Load (ETL)*, de uma ou mais fontes de dados, necessárias para a composição da análise de informações [7].

Uma abordagem convencional das rotinas de ETL é o processamento *batch*, caracterizado pelo processamento de dados com latência de horas ou até mesmo dias, demandando um tempo considerável para processamento. Este processamento do tipo centralizado é uma prática reforçada, principalmente para diminuir a latência entre as consultas de dados na origem e o processamento dos dados. Entretanto, o tempo de resposta nesse tipo de abordagem tem impacto na análise dos dados e no processo de tomada de decisão nas mais diversas corporações.

Nesse sentido, soluções com base nas tecnologias MapReduce [8] e Hadoop [9], apesar de permitirem o processamento de grandes volumes de dados, utilizando computação distribuída e altamente escalável, não permitem o processamento de dados em tempo real. Essas tecnologias são orientadas ao paradigma de processamento *bacth* e apresentam limitações ao serem aplicadas em soluções que demandam o processamento dos dados de baixa latência.

A Figura 1 representa uma comparação entre o tempo de resposta esperado para o processamento de dados em diferentes abordagens.

Figura 1.1: Latência no processamento de dados para diferentes abordagens

A abordagem de processamento de dados com chamadas de procedimento remoto, do inglês *Remote Procedure Call (RPC)*, possuem latência de milissegundos. O protocolo RPC estabelece uma estrutura de comunicação, que permite a execução de rotinas em programas distribuídos em uma rede de computadores [10].

O processamento distribuído de *stream* de dados é uma abordagem com latência de segundos ou minutos. Essa abordagem contempla características de alta disponibilidade, particionamento

dos dados e escalabilidade vertical.

As abordagens de processamento baseadas em Hadoop e no paradigma MapReduce tem latência de minutos ou horas. Essa abordagem permite o processamento de grandes volumes de dados e dispõe de recursos para alta disponibilidade e escalabilidade vertical.

O processamento centralizado de ETL é uma abordagem com latência de dias. Essa abordagem permite o processamento *batch* dos dados obtidos a partir de diferentes sistemas de origem. O processamento centralizado de ETL é uma abordagem bastante difundida na área de processamento de dados para sistemas de BI.

De acordo com [11], quando o tempo de resposta no processamento de dados é fator crítico para soluções de BI, é necessário substituir o processamento *batch* por uma estratégia orientada a *streaming* de dados. As soluções de ETL orientadas a rotinas *batch* são incapazes de realizar o processamento desses dados em tempo real. O uso do processamento de *stream* de dados não deve ser considerado uma evolução nas rotinas de processamento de dados, mas a um paradigma diferente que permite a entrega das informações em um menor tempo de resposta.

A alta complexidade envolvida na engenharia de sistemas de processamento de *stream* de dados distribuídos é uma característica comum em projetos de desenvolvimento de soluções analíticas de *near real-time*. O estudo de técnicas de engenharia de *software* e arquitetura de sistemas de *software* combinadas às técnicas de processamento de dados para soluções de BI é uma alternativa para o gerenciamento da complexidade envolvida ao processamento de *stream* de dados.

O custo de se desenvolver uma nova arquitetura para cada nova rotina de processamento de dados é elevado. Desse modo, para se reduzir o impacto da fase arquitetural em novos projetos, é importante definir uma arquitetura de referência base sobre a qual outras arquiteturas de referência podem ser construídas. Essas, por sua vez, são pilares das arquiteturas de software. Uma arquitetura de referência base propicia o aproveitamento dos investimentos já realizados, além de permitir gerir a complexidade envolvida no desenvolvimento de sistemas de *software* complexos.

Em sistemas de análise de dados em *near real-time*, a diversidade de tecnologias no mercado agregada à elevada complexidade no desenvolvimento de sistemas para o processamento de *stream* de dados impactam diretamente na escalabilidade, reuso e manutenibilidade desses sistemas.

Em contraposição, esse trabalho propõe um modelo reutilizável, independente de tecnologia, para ser implementado em sistemas de BI que requerem a análise de *stream* de dados em *near real-time*.

O termo "modelo"indica um conjunto de especificações que podem ser instanciadas por novos sistemas ou arquiteturas de *software*. O termo "reutilizável"refere-se ao uso do modelo em diferentes sistemas de *software*, estabelecendo características e comportamentos comuns para componentes do mesmo tipo.

1.1 Objetivos

1.1.0.1 Objetivo Geral

Esta dissertação objetiva definir uma arquitetura de referência para o desenvolvimento rotinas de processamento de *stream* de dados dispondo de decisões de *design* reutilizáveis e independentes de tecnologia.

1.1.0.2 Objetivo Específicos

Avaliar a arquitetura de referência proposta com a execução de prova de conceito e desenvolvimento de um experimento para análise das técnicas empregadas. Esse experimento instancia a arquitetura proposta e utiliza algorítimos inovadores para o processamento de *stream* de dados: HyperLogLog para a estimativa da frequência de elementos e Count-min Sketch para a estimativa de elementos distintos.

Estabelecer decisões de *design* que apoiem a construção componentes de *software* reutilizáveis, escaláveis horizontalmente, desacoplados e com alta coesão no desenvolvimento de sistemas de processamento de dados.

Além disso, serão apresentados, previamente, conceitos utilizados neste trabalho para a engenharia de sistemas analíticas em *near real-time*: *Business Intelligence* (BI), Tecnologias para desafios de *Big Data* e Arquitetura de software.

1.2 Contribuições

Esta dissertação define uma arquitetura de referência para o processamento de *stream* de dados distribuídos para sistemas de análise de dados em *near real-time*, por meio de técnicas de engenharia de *software*, descrição de visões arquiteturais e de modelos para o processamento de *stream* de dados distribuídos. A arquitetura de referência não é limitada à tecnologia ou contexto específico, podendo ser evoluída e aplicada em diversos cenários.

1.3 Organização do Trabalho

Tendo estabelecido a finalidade desta dissertação, os seguintes capítulos focam nos trabalhos relacionados, fundamentação teórica, descrição da solução proposta e os resultados experimentais obtidos, e culminam em um resumo que inclui as conclusões e recomendações para um estudo mais aprofundado.

Este trabalho é dividido em Capítulos conforme relacionado a seguir. O Capítulo 2, trabalhos relacionados, inclui um resumo de pesquisas recentes sobre o processamento e armazenamento de dados. No Capítulo 3, inicia-se a fundamentação teórica, apresentando os conceitos e técnicas para definição e descrição de arquiteturas de *software*. O Capítulo 4, apresenta as tecnologias envolvidas

em desafios de *Big Data* e análise de dados em *near real-time*. No Capítulo 5, são apresentados os conceitos e metodologias relacionados ao desenvolvimento de sistemas de BI. No Capítulo 6, solução proposta, descreve-se a arquitetura de referência proposta por esta dissertação. No Capítulo 7 são apresentadas as análises e os resultados obtidos com a instanciação da arquitetura de referência proposta no desenvolvimento de um experimento como prova de conceito arquitetural. O Capítulo 8 apresenta a conclusão dessa dissertação e expõe os trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Os métodos para o processamento de dados em ambientes computacionais centralizados são técnicas amplamente utilizadas na engenharia de sistemas de análises de dados [12]. [7] e [13] são responsáveis pelas principais e mais difundidas contribuições relacionadas à metodologias para o desenvolvimento de sistemas de *Business Intelligence* e procedimentos de extração, transformação e carga (ETL) dos dados. Além disso, os trabalhos de [7] e [13] especificam a maior parte das práticas conhecidas para a arquitetura e desenvolvimento de sistemas de análises de dados.

A descoberta de padrões em sistemas de análise de dados é vital para o enriquecimento dessas análises [14]. O desenvolvimento de novas técnicas, e a evolução de técnicas existentes, são alvo de pesquisas nas áreas de algoritmos computacionais [15], estatística e *machine learning* [16]. Como resultado dessas pesquisas, sistemas de *software* como o WEKA [17] disponibilizam uma abstração de dados, permitindo definir uma estrutura de dados específica contendo somente os dados relevantes à mineração de dados, e *Application Programming Interfaces* (APIs) que viabilizam a descoberta de dados em soluções de processamento de dados centralizadas.

Ainda na abordagem centralizada, o processamento de *stream* de dados é fundamental para sistemas de análise em *near real-time* [18] [19]. Nos últimos anos diversas pesquisas propuseram algoritmos [20], modelos [21] e arquiteturas [22] para o processamento e análise de *streams* de dados.

O uso de técnicas arquiteturais para a descrição de decisões de *design* importantes em sistemas de processamento de dados, promovendo o reuso e manutenabilidade nesses sistemas, pode ser observado em [22]. Além disso, modelos únicos que padronizam a estrutura de dados alvo do processamento são encontrados em [17].

Entretanto, essas pesquisas desenvolveram tecnologias para o processamento centralizado dos dados. A abordagem centralizada de processamento é incapaz de atender as demandas impostas pelo crescente volume e variedade dos dados produzidos por diversos tipos de aplicativos, uma vez que sua escalabilidade é limitada a capacidade de processamento de um único nó computacional [1]. Em vista dessa necessidade, o desenvolvimento de novas tecnologias tornou-se alvo de diversas pesquisas que buscam viabilizar o processamento desse grande volume de dados, destacando-se [23], [8] e [9].

Entre as pesquisas desenvolvidas para viabilizar o processamento desse grande volume de dados, [23] propõe o *Google File System* (GFS). O GFS é um sistema de arquivos escalável e distribuído em grandes *clusters* de computadores. [8] propõe o MapReduce, um novo paradigma capaz de processar grandes volumes de dados MapReduce. Em seguida, o Apache Hadoop [9] integrou um *framework* para análise de grandes volumes de dados baseado no MapReduce e um novo sistema de arquivos distribuído, o *Hadoop Distributed File System* (HDFS).

Além disso, emergiu uma nova categoria de Sistemas Gerenciadores de Bancos de Dados (SGBDs), conhecidos como *Not Only SQL* (NoSQL). Essa categoria de SGBD visa o gerenciamento e o processamento do grande volume e variedade de dados produzidos [24]. A pesquisa de [25] desenvolveu o Google BigTable, utilizado de forma massiva no Google para o armazenamento de petabytes (PB) de dados em *clusters* computacionais compostos por milhares de computadores. Em seguida, [26] propõe o HBase, um SGBD desenvolvido pela Apache Foundation como uma alternativa aberta ao Google BigTable. Os SGBD NoSQL têm sido tecnologia fundamental para o processamento e armazenamento de grandes volumes de dados [27] [28].

A análise de *Big Data* em *near real-time* é ponto focal de pesquisas que propõe tecnologias capazes de processar, em baixa latência, esse grande volume e variedade de dados [29]. O trabalho de [30] define as principais características que devem ser contempladas por esses sistemas. Ele estabelece requisitos fundamentais como tolerância a falhas, mobilidade dos dados e consistência de resultados.

As pesquisas no campo de processamento de *Big Data* em *near real-time* produziram diversos *Distributed Stream Processing Systems* (DSPS). O Borealis [31], desenvolvido a partir do sistema de processamento centralizado Aurora [32], foi um dos primeiros sistemas estabelecidos para o processamento de *streams* distribuídos. Em seguida, para a análise de seus dados, o Twitter desenvolveu um DSPS chamado Storm [33], enquanto o LinkedIn, desenvolveu um sistema de mensageria distribuída chamado Apache Kafka [34] e um DSPS chamado Apache Samza [35].

A descoberta de padrões em *streams* distribuídos é alvo de pesquisas que buscam reduzir o consumo de recursos computacionais e maior precisão dos resultados [36]. Em [37] e [38] propõe-se algoritmos de baixo consumo de memória, capazes de estimar frequências de valores e cardinalidades com uma pequena taxa de erro. Nesse sentido, diversas pesquisas propõe o uso de algoritmos estatísticos na análise de grandes volumes de dados [39] [15].

Existem ainda pesquisas focadas em fornecer uma plataforma de *software* para mineração de dados, capaz de realizar a descoberta de padrões e estimativa de valores em *streams* de dados distribuídos [40]. A pesquisa de [40] propõe o sistema *Scalable Advanced Massive Online Analysis* (SAMOA), uma plataforma escalável para mineração de *streams* de *Big Data*. A pesquisa de [40] propõe ainda um modelo comum para a análise desses dados, o que permite que sejam utilizados diferentes DSPS para a execução dos processos de mineração de dados.

Técnicas para a descrição de arquiteturas de *software* são propostas por [41] e [42] para a redução da complexidade e melhora na manutenabilidade de sistemas de *software*. O trabalho de [43] propõe uma arquitetura genérica, para a separação de tecnologias de *Big Data* de acordo com suas responsabilidades e principais características.

A definição de arquitetura de referência e sua utilização como estratégia de reduzir o retrabalho e promovendo a reutilização de decisões de projeto no desenvolvimento de famílias de *software* em série pode ser visto em [44] [45] [46]. No entanto, é em [47] que encontramos subsídios para a aplicação de arquiteturas de referência em diversos níveis, abrangendo diferentes contextos e visões de um sistema.

Esta dissertação relaciona-se com os trabalhos apresentados pela proposição de técnicas para o processamento de *Big Data*, como todos os citados, entretanto a proposta apresentada diferencia-se de [22], por empregar, em uma arquitetura de referência, tecnologias para o processamento distribuído do *stream* de dados. Difere de [48], por propôr uma arquitetura de referência independente de tecnologia, descrita em várias visões arquiteturais. Distingue de [40], pelo fato de fornecer uma arquitetura e modelo de processamento reutilizáveis em diferentes contextos de negócio. É dissímil de [17], por não precisar de dados históricos armazenados para o processamento de dados e suportar o processamento distribuído de *streams* de dados; E diferencia-se de [49], por propôr uma arquitetura para análise de dados sem necessidade de armazenamento prévio e em *near real-time*;

A Tabela 2.1 faz um comparativo entre os trabalhos relacionados e esta dissertação, mostrando que o presente trabalho integra diferentes técnicas em uma arquitetura de referência reutilizável, apresentada ao longo desta obra.

Trabalhos	**Técnicas empregadas**			
	Distribuído	***Streaming***	**Abordagem arquitetural**	**Modelo unificado**
[17]	-	-	-	x
[40]	x	x	-	x
[22]	-	x	x	-
[48]	x	x	-	-
[49]	x	-	x	-
Esta dissertação	x	x	x	x

Tabela 2.1: Comparativo entre os principais trabalhos relacionados e esta dissertação

Além disso, essa arquitetura complementa os trabalhos relacionados conforme apresentado na Tabela 2.1, integrando aspectos relevantes ao processamento distribuído de *stream* de dados em *near real-time*, que não são plenamente abordados pelos referidos trabalhos relacionados.

Capítulo 3

Arquitetura de Software

3.1 Introdução

Na engenharia de *software*, a complexidade envolvida no desenvolvimento de sistemas tem crescido em níveis sem precedentes. Nesse sentido, conceitos, princípios e procedimentos arquiteturais tem sido cada vez mais aplicados durante o projeto de sistemas de *software* para gerenciar essa complexidade.

O projeto de um sistema de *software* é definido pela International Organization for Standardization (ISO) como o processo de definir a arquitetura, componentes, interfaces e outras características de um sistema ou componente de *software* [50].

A Figura 3.1 representa, do ponto de vista do processo de engenharia de *software*, o momento em que o projeto de um sistema é elaborado. Nesse etapa da engenharia de software, as regras necessárias para a resolução de um problema são elicitados e descritos como requisitos de um sistema (a). Em seguida, durante a análise e projeto (b) os requisitos são descritos em elementos de projeto que retratam as características desse sistema. O código-fonte (c) é construido a partir da especificação de requisitos e dos elementos de projeto, resultando em um sistema ou componente de *software* [42].

De acordo com um processo padrão para o ciclo de vida de software, o ISO/IEC/IEEE Std. 12207 Software Life Cycle Processes [51], o projeto de um software compreende duas atividades:

1. **Projeto arquitetural de software** define a organização do software, estruturas de alto nível e como o software identifica seus principais componentes.

2. **Projeto detalhado de software** especifica cada componente no nível necessário para permitir a sua construção de forma adequada.

É na atividade de projeto arquitetural de software que a arquitetura de um sistema recebe maior atenção. Existem diversas definições para arquitetura software, várias delas catalogadas pelo Software Engineering Institute (SEI) da Carnegie Mellon University [52]. Neste trabalho, adotamos a definição de [53]:

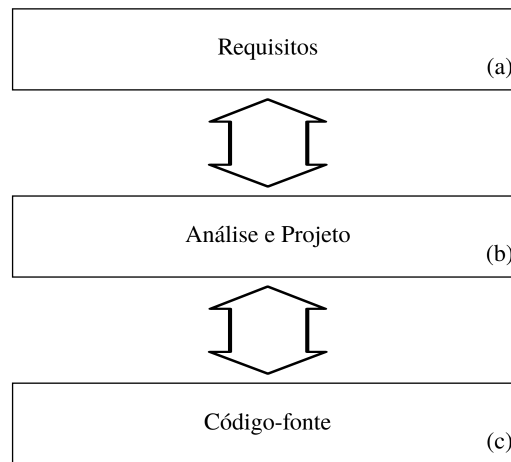

Figura 3.1: Projeto de software na engenharia de software

> *A arquitetura de um programa ou sistema de computação é a estrutura ou conjunto de estruturas de um sistema, composta de seus elementos de software, as propriedades externas desses elementos e a relação entre eles.*

Na arquitetura de software os elementos de um sistema de software podem ser compreendidos, do ponto de vista da arquitetura de software, como elementos que possuem atributos, responsabilidades e comportamentos bem definidos.

As propriedades externas são aquelas visíveis a partir de outros elementos de software como, por exemplo, características de performance, utilização recursos comuns, serviços e tolerância a falhas.

A partir da definição de [53], entendemos que a arquitetura de um sistema software é uma abstração do sistema, em que são considerados os elementos de software e como eles se relacionam. Do ponto de vista arquitetural, as particularidades internas do funcionamento dos elementos de *software* são irrelevantes, por não afetarem como estes elementos utilizam, são utilizados ou interagem com outros elementos de *software*.

3.2 Descrição arquitetural

De acordo com [54], existe uma distinção entre arquitetura de software e descrição arquitetural de software. Essa diferença pode ser explicada pela constatação de que todo sistema de software pode ser observado como seus elementos e as relações entre eles, enquanto a descrição arquitetural depende da documentação da arquitetura.

Uma descrição arquitetural é composta de produtos e artefatos [47], que são produzidos no projeto de um sistema para a representação de sua arquitetura. Dessa forma, separamos arquitetura de software, que é algo conceitual, de sua descrição, que é obtida a partir de artefatos e produtos concretos.

As descrições arquiteturais são organizadas em uma ou várias visões. Essas visões são res-

ponsáveis por transmitir o entendimento dos elementos de um sistema de uma maneira clara e consistente para todos os envolvidos em um projeto de software.

Cada visão pode contemplar um ponto de vista específico, que define a perspectiva em que uma visão deve ser construída e observada. Além disso, os pontos de vista definem as informações que uma visão deve transmitir, as notações e técnicas de modelagem que devem ser aplicadas e as justificativas para essas escolhas [47].

A decisão sobre qual notação deve ser utilizada em uma descrição arquitetural envolve balancear o esforço necessário para construção e a precisão na representação das visões arquiteturais. Normalmente, notações formais demandam maior tempo para documentação, mas resultam em representações completas e sem ambiguidade. Por outro lado, as notações informais podem ser documentadas em menor tempo, mas resultam em visões subjetivas e ambíguas.

[55] define três categorias principais de notações para a documentação de visões arquiteturais:

1. Notações informais – representam as visões arquiteturais por meio de diagramação genérica e convenções visuais adotadas de acordo com a ferramenta utilizada. A semântica da descrição é caracterizada pela linguagem natural e não pode ser formalmente analisada.
2. Notações semi-informais – as visões são expressadas de forma padronizada que aplicam elementos gráficos e regras de construção. Entretanto, as notações semi-informais não fazem um tratamento semântico desses elementos. Análise rudimentar pode ser usada para determinar se a descrição atende as propriedades sintáticas. Nesse sentido, podemos entender UML como uma notação semi-informal.
3. Notações formais – descrevem as visões em uma notação que tem semântica precisa. É possível realizar uma análise formal da sintaxe e semântica utilizada. Há uma variedade de linguagens formais para a representação de arquiteturas de software, geralmente referenciadas como linguagens de descrição arquitetural, do inglês *Architecture Description Languages (ADL)*.

O diagrama de classes da *Unified Modelling Language* (UML), na Figura 3.2, representa a composição conceitual proposta pelo padrão IEEE-1471:2000 de uma descrição de software [54]. Essa representação estabelece uma terminologia para a declaração dos requisitos de um sistema em sua descrição arquitetural.

A arquitetura de um sistema de software é descrita com a ajuda de ADL e das representações de visões arquiteturais. Essas visões buscam transmitir o entendimento adequado para todos os envolvidos em um projeto de software.

3.3 Estruturas de sistemas de software

Na ótica da descrição de uma arquitetura de software, um sistema de software contém dois tipos de estruturas: estruturas estáticas e estruturas dinâmicas [56].

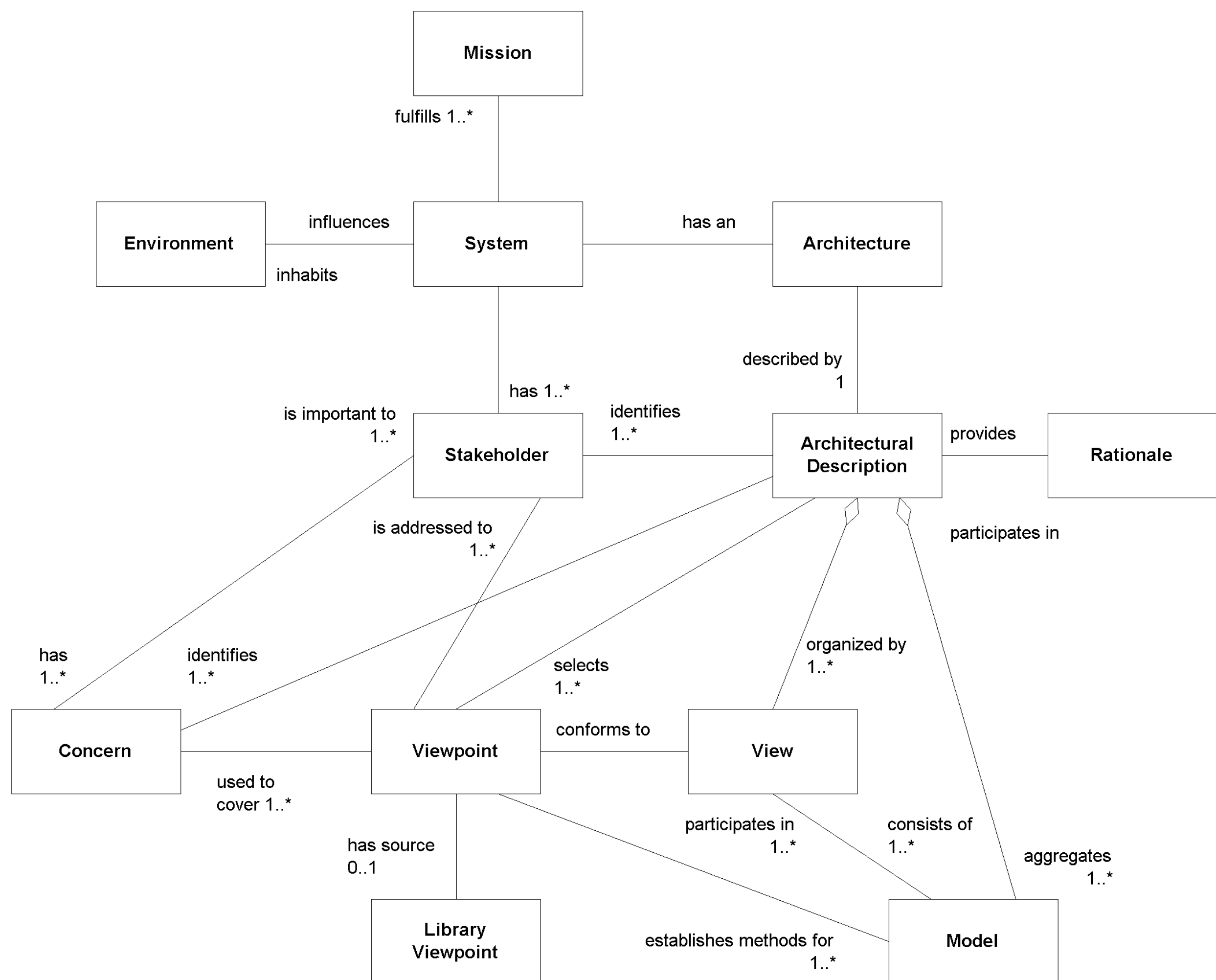

Figura 3.2: Visão conceitual de uma descrição arquitetural [54]

1. **Estruturas estáticas**

   As estruturas estáticas são os elementos descritos durante o projeto de software. Podem ser compreendidas como estruturas estáticas módulos, classes orientadas a objeto, pacotes, serviços ou qualquer outra unidade de código de um sistema de software. A organização estática desses elementos define seus relacionamentos, associações ou conectividade entre eles. A representação e descrição dessas estruturas, seus principais componentes e relacionamentos são, em sua maioria, notações gráficas:

   - Linguagens de Descrição Arquitetural (ADLs)
   - Diagramas de classes e diagramas de objetos
   - Diagramas de componentes
   - Diagramas de implantação

2. **Estruturas dinâmicas**

As estruturas dinâmicas são os elementos de um software em tempo de execução e como eles interagem entre si. As estruturas dinâmicas permitem entender como o sistema de software trabalha em tempo de execução, ou seja, como ele responde aos estímulos internos ou externos. Essas interações podem ser fluxos de informações entre elementos ou execuções paralelas ou sequenciais de tarefas. As notações e linguagens abaixo são usadas para representar de forma textual ou gráfica os comportamentos dinâmicos de um sistema de software:

- Diagramas de atividade
- Diagramas de comunicação
- Diagramas de fluxo de dados
- Diagramas de sequência
- Diagramas de estado
- Fluxogramas
- Pseudocódigo

A partir da utilização dessas estruturas e de notações, encontramos as ferramentas necessárias para a descrição arquitetural de um sistema de software. Essa descrição é a especificação essencial para a construção de sistemas de software, promovendo o reuso e ganhos de maturidade no processo de desenvolvimento de software.

3.4 Padrões Arquiteturais

A lógica de analisar problemas e soluções compatíveis é uma abordagem comum e natural, que nos permite associar comportamentos e soluções para qualquer tipo de interação social ou problema [57].

Não é comum que especialistas inventem uma solução totalmente nova para a resolução de diferentes problemas. Especialistas, como por exemplo, os engenheiros de software, identificam em sua base de conhecimento, problemas similares na tentativa de aplicar a essência da solução utilizada à resolução de um novo problema.

A Figura 3.3 representa a composição de um padrão de projeto que, segundo [58, pg. 247], é definido como uma regra de três partes, que expressa a relação entre um contexto, um problema e uma solução.

O **contexto** é o conjunto de circunstâncias onde o problema, que é solucionado pelo padrão, foi identificado. Ele pode ser extremamente generalizado, o que resulta em maior aplicabilidade para o padrão [44, pg. 104].

O **problema** é a formalização dos incidentes percebidos repetidamente no contexto. A especificação de um problema delimita de maneira genérica o incidente, podendo ser acompanhada de restrições e características desejadas para a solução do problema [44, pg. 104-105].

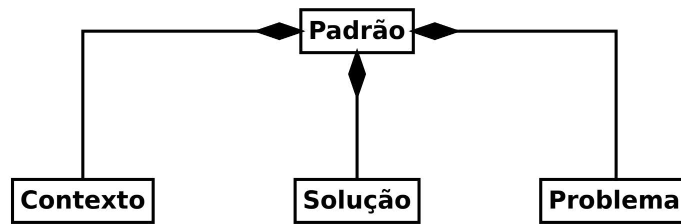

Figura 3.3: Diagrama da UML que representa a composição de um padrão de projeto

A **solução** propõe uma resolução para o problema recorrente. A estrutura estática da solução genérica pode ser expressada em diferentes ADLs [44, pg. 105].

Como uma regra geral, quanto maior o nível de abstração de um padrão, maior deve ser sua aplicabilidade. Além disso, um mesmo padrão pode ser aplicado em diversos cenários, de modo a sofrer especializações de acordo com as necessidades envolvidas [44, pg. 6-11].

O trabalho de [59] observa que na arquitetura de software os padrões são aplicados na construção de sistemas de *software* para:

- Gerenciar a complexidade de software
- Promover o reuso de decisões de projeto bem sucedidas
- Promover um vocabulário comum e o entendimento para os princípios de projeto de *software*.
- Auxiliar a construção de arquitetura de *software* complexas e heterogêneas.
- Identificar e especificar abstrações que vão além da visão de unidades de código como classes, funções ou componentes.

Na ótica da engenharia de software, os padrões arquiteturais representam o maior nível de abstração entre os padrões. Eles permitem a especificação da estrutura fundamental de uma aplicação, seus subsistemas, responsabilidades e regras que definem as relações entre eles.

Um padrão arquitetural auxilia o atingimento de critérios globais em um sistema de software, como a extensibilidade de uma interface de software. Segundo [60], oito padrões arquiteturais podem ser definidos:

- *Layers*, que facilitam a estruturação de sistemas de *software* que podem ser decompostos em grupos menores, com responsabilidades especificas e níveis de abstração particulares.
- *Pipes and Filters*, um padrão que disponibiliza uma estrutura composta de *Steps*, *Events* e *Pipes*, que subsidiam o desenho de sistemas para o processamento de *stream* de dados.
- *Blackboard*, originado em sistemas de Inteligência Artificial é um padrão que apoia a especificação de diversos subsistemas que implementam sua lógica e conhecimento para constituir um resultado parcial, ou solução aproximada. Esse padrão é geralmente aplicado para a arquitetura de sistemas que tratam problemas sem solução determinística.

- *Broker*, utilizado em arquiteturas de sistemas *software* de estruturas distribuídas e componentes desacoplados que interagem a partir de invocações sucestiveis remotas.
- *Model-View-Controller (MVC)*, que separa um sistema de software interativo em três componentes: O modelo (*model*), que deve conter as principais funcionalidades e os dados, a visão (*views*), que exibe as informações ao usuário e o controle (*controller*), que trata as ações do usuário.

  As visões e controles compoem a interface com o usuário, enquanto mecanismos de propagação garantem a consistência entre o modelo e a interface com o usuário.
- *Presentation-Abstraction-Control* – define uma estrutura interativa para sistemas de *software* na forma de componentes de processamento, hierárquicos e cooperativos, responsáveis por aspectos específicos das funcionalidades do sistema. Esse padrão consiste na separação em três componentes: apresentação, abstração e controle.

  Essa divisão permite separar a interação homem-máquina dos componentes de processamento e a sua comunicação com outros componentes.
- *Microkernel* – padrão arquitetural que é aplicado em sistemas que são suscetíveis à mudanças frequentes em seus requisitos. Esse padrão permite separar um *core* mínimo funcional, de funcionalidades estendidas e componentes de clientes específicos.
- *Reflection* – é um padrão que disponibiliza mecanismos necessários para sistemas de *software* que sofram mudanças de estrutura e comportamento dinamicamente. Nesse padrão um sistema é separado em dois níveis: Meta, que define as informações a respeito das propriedades do sistema e permite o auto-conhecimento do *software*, e Base, responsável pela lógica da aplicação e alterações nos dados manipulados pela aplicação.

O processo que permite identificar novos padrões é um processo particular que envolve avaliar o sucesso obtido com a aplicação de uma mesma solução em problemas com características recorrentes [61].

3.5 Arquitetura de referência

A crescente demanda por rápidas mudanças e frequentes adaptações em sistemas de *software*, com prazos cada vez menores, além do aumento na complexidade desses sistemas, têm adquirido a atenção dos engenheiros de *software* para a arquitetura de software.

Uma arquitetura de referência captura a essência de arquiteturas existentes associadas a visão de necessidades futuras, provendo uma referência para o desenvolvimento de novas arquiteturas de *software* [45]. As arquiteturas de referência estabelecem um conjunto de recomendações e restrições que tendem a reduzir o esforço e tempo necessários em novos projetos de sistemas de *software* [46]. A Figura 3.4 representa a relação entre uma arquitetura de referência e uma arquitetura de software instanciada.

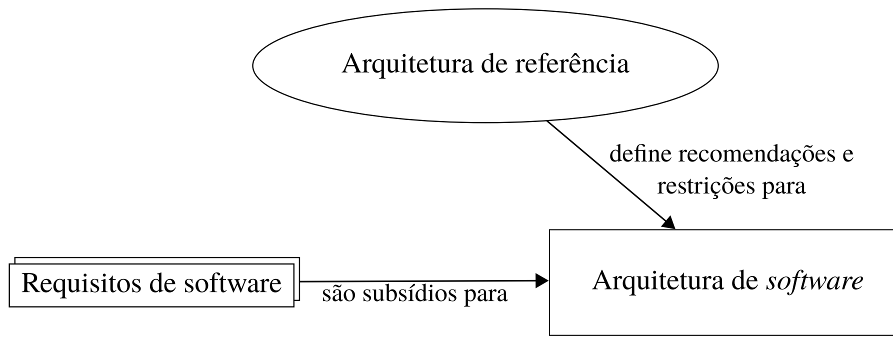

Figura 3.4: Propósito de uma arquitetura de referência

Uma arquitetura de referência é, em sua essência, um padrão arquitetural ou um conjunto de padrões definidos previamente, parcialmente ou completamente instanciados, projetados e comprovados para uso em necessidades específicas de negócio e em contextos técnicos, acompanhados de artefatos de documentação que possibilitem seu uso [62].

De acordo com [45], as arquiteturas de referência disponibilizam uma taxonomia e visões arquiteturais, além de modularização e contexto complementares que facilitam o ciclo de vida da criação de sistemas de *software*. As arquiteturas de referência permitem gerenciar a sinergia entre diferentes produtos, provendo uma orientação contendo melhores práticas, princípios e padrões arquiteturais que podem ser compartilhados entre diferentes sistemas de *software*.

O esforço envolvido nas decisões arquiteturais durante o desenvolvimento de novos sistemas é um fator crítico para o sucesso de um projeto de *software*. Do ponto de vista industrial, uma arquiteturas de referência se propõe a atuar principalmente no gerenciamento da complexidade, mitigação de riscos, reuso e entendimento comum e institucional no desenvolvimento de sistemas de *software* [45].

Os benefícios de uma arquitetura de referência estão diretamente ligados ao nível de sucesso em sua utilização. Essas arquiteturas devem ser descritas em um nível de abstração adequado para a instanciação em novas arquiteturas de software. Além disso, a divulgação dessas arquiteturas deve ocorrer de forma institucional em bases de conhecimento genéricas ou bibliotecas de referência [47].

De acordo com [46], arquiteturas de referência podem ser classificadas com base em seus contextos, objetivos e descrições arquiteturais. A Figura 3.5 representa essa relação, onde observamos que essas características são estabelecidas durante o fluxo de elaboração de uma arquitetura de referência.

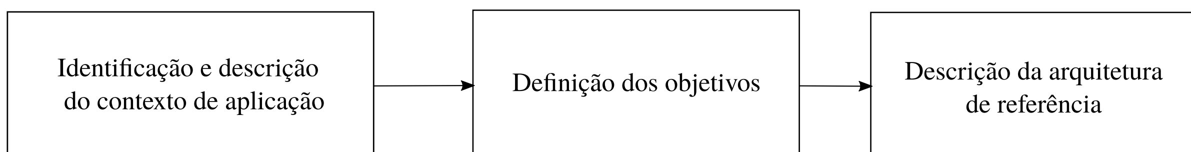

Figura 3.5: Relação entre contexto, objetivos e descrição de uma arquitetura de referência

Capítulo 4

Big Data

Na atualidade, os dados produzidos e gerenciados por sistemas de software são bastante diversificados e complexos. Essas características são consequências de evoluções tecnológicas em hardware e software, que resultaram em ampla utilização de aplicativos ligados à Internet. O acesso à Internet, a partir de pequenos dispositivos, permite, por exemplo, que cientistas produzam dados detalhados de diversos tipos de monitoramento ao redor do globo [29].

Na Figura 4.1, representamos os quatro principais marcos tecnológicos no processamento de dados. O desenvolvimento de novas tecnologias está diretamente relacionado à evolução no volume de dados produzidos. De acordo com o volume e inovações tecnológicas, podemos agrupar os desafios de *Big Data* em quatro principais marcos: *Megabyte* para *Gigabyte*, *Gigabyte* para *Terabyte*, *Terabyte* para *Petabyte* e *Petabyte* para *Exabyte* [63].

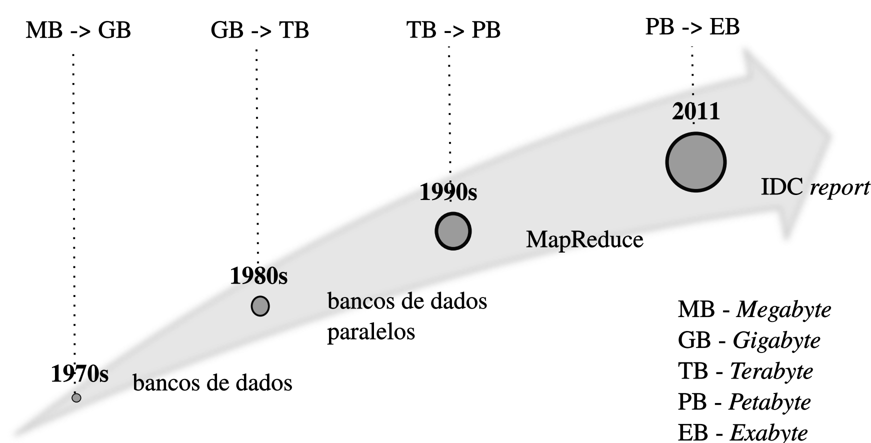

Figura 4.1: Relação entre o crescimento no volume de dados e inovações no processamento de dados

- **Megabyte para Gigabyte**

  Entre os anos de 1970 e 1980, a demanda crescente pelo armazenamento e execução de consultas relacionais em dados históricos foi o ponto focal da indústria nos primeiros desafios de *Big Data*. Como resultado, os pesquisadores introduziram uma tecnologia integrada de

software e *hardware*, conhecida como *database machine* [64] [65]. Essa tecnologia tinha como proposta oferecer a melhor performance com custos reduzidos.

- **Gigabyte para Terabyte**

  A disseminação da tecnologia digital, no final dos anos 1980, resultou no crescimento do volume de dados produzidos. Após algum período de sucesso, a indústria constatou que a combinação entre *hardware* e *software* não acompanhava a evolução dos computadores genéricos. Dessa forma, para atender a demanda da indústria por esta maior capacidade de processamento de dados, uma nova geração de sistemas de banco de dados surgiu, propondo o paralelismo como estratégia para atingir maior performance por meio da distribuição dos dados e tarefas de processamento [66].

- **Terabyte para Petabyte**

  Ao final dos anos 1990, a comunidade de bancos de dados ainda admirava os resultados dos bancos de dados paralelos, quando a *World Wide Web* (WWW) iniciou uma tendência de produção massiva de terabytes para petabytes de dados semi estruturados em páginas da *web*. Como resultado, companhias começaram a indexar o conteúdo da web para permitir a consulta à estes dados não estruturados.

  Entretanto, as capacidades de armazenamento dos sistemas existentes eram limitadas. O desafio de gerenciar e analisar os dados da web resultaram na criação do *Google File System* (GFS) [23] e do modelo de programação MapReduce [8]. Os sistemas que utilizem GFS e MapReduce permitem a escalabilidade horizontal dos recursos computacionais. A utilização de hardware comodite, interligados em *clusters* de computadores, resultam em grande capacidade de armazenamento.

  Uma nova geração de bancos de dados emerge. Os bancos de dados NoSQL oferecem estratégias para o gerenciamento de dados não estruturados e semi estruturados de maneira rápida, escalável e segura.

- **Petabyte para Exabyte**

  As tecnologias disponíveis atualmente não são capazes de processar esse novo volume de dados. Um relatório publicado pela EMC [67] explicita a necessidade do desenvolvimento de novas tecnologias para o processamento de *Petabytes* a *Exabytes* de dados. Após a publicação desse relatório, grandes empresas como Oracle, Microsoft, Google, Amazon e Facebook investiram fortemente no desenvolvimento de novas tecnologias para o processamento de *Big Data* [68] [69] [70].

O conceito de *Big Data* é usado para definir os dados que excedem as capacidades de um banco de dados convencional. Os dados são produzidos em grande volume e velocidades altíssimas ou não podem ser armazenados em arquiteturas de bancos de dados [71]. A literatura geralmente define *Big Data* pelo volume de dados. Entretanto, de acordo com [72], desafios de *Big Data* devem considerar mais duas dimensões, representadas na Figura 4.2, onde *Big Data* é definido levando em consideração as relações entre o Volume, a Variedade e Velocidade (3Vs) dos dados.

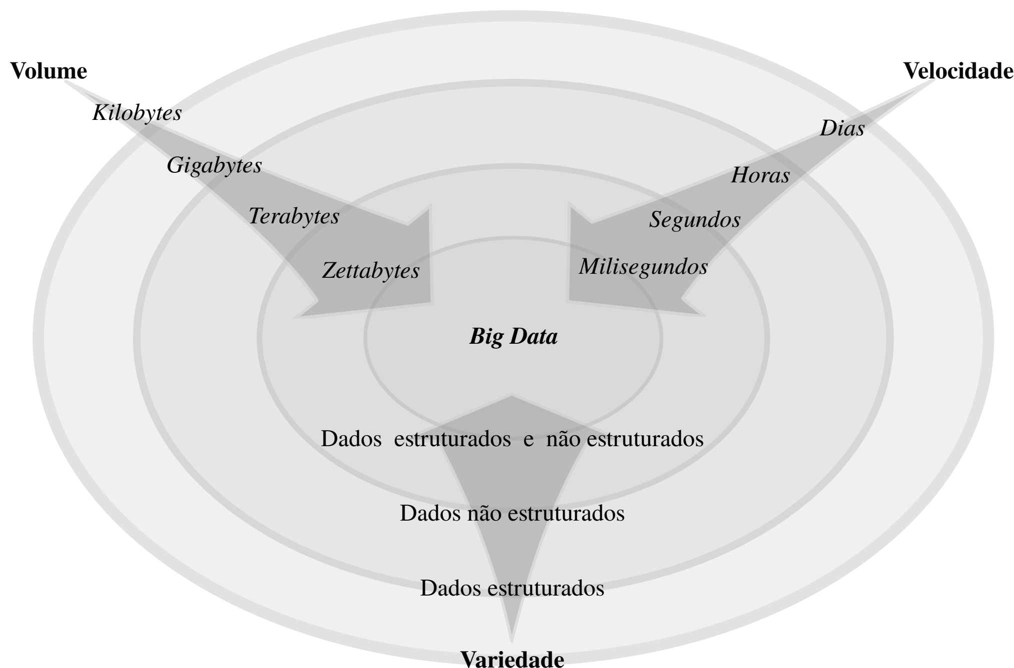

Figura 4.2: Os 3Vs que definem *Big Data*

- Volume

  O volume de dados é o primeiro atributo da definição de Big Data. O volume representa o primeiro desafio para as estruturas convencionais de Tecnologia da Informação (TI). Com base nisso, podemos quantificar o volume de Big Data em *Terabytes* (TB), *Petabytes* (PB) ou até mesmo *Zettabytes* (ZB) [72]. Algumas instituições já atingiram grandes volumes de dados armazenados, por exemplo: Google, Yahoo e Facebook. Ao assumir que esse volume de dados são maiores do que as capacidades de processamento de infraestruturas de bancos de dados relacionais convencionais, as opções para o processamento e armazenamento de dados se limitam a escolha de soluções de processamento paralelo. Nesse contexto, soluções baseadas em Hadoop, que utilizam um sistema de arquivos distribuído o *Hadoop Distributed File System* (HDFS), viabilizam a disponibilidade desses dados em múltiplos nós computacionais.

- Variety

  Os dados podem ser armazenados em diversos formatos. A origem desses dados pode ser textos de redes sociais, dados de imagem ou, ainda, dados puros recebidos diretamente de um sensor. Nenhum desses dados está prontos para serem utilizados em integrações com outros sistemas de *software*. O processamento desses grandes volumes de dados não estruturados é um uso comum de tecnologias de *Big Data*, que permite extrair informações significantes que podem ser consumidas por outras aplicações ou, até mesmo, para observaçao humana. Nesse contexto, existem classes de bancos de dados mais adequados para tipos específicos de dado [71]. Por exemplo, documentos codificados como XML podem ser melhor explorados quando armazenados em uma estrutura de armazenamento XML. Relacionamentos de redes sociais são basicamente grafos e, nesse sentido, seu armazenamento é mais eficiente quando feito em bancos de dados baseados em grafos.

- Velocity

  A velocidade com que os dados são gerados também podem descrever *Big Data*. A frequência com que os dados são gerados ou entregues também são relevantes à definição de *Big Data*. Nesse contexto, o crescimento na produção de dados e a explosão na utilização de mídias sociais transformaram as maneiras de se analisarem os dados. Em mídias sociais, usuários tendem a acessar dados relacionados a mensagens recentes, por exemplo um tweet e atualizações de *status*. As mensagens antigas são descartadas com frequência e tendem a acessar as informações recentes. O fluxo dos dados é praticamente em tempo real e as janelas de atualização são reduzidas a transações de frações de segundos. Dessa forma, mais uma vez dispositivos ligados à Internet, como os celulares e *tablets* propiciam o crescimento na taxa de produção de dados, o que resulta na caracterização dos *stream* de dados [71].

Existem várias justificativas para considerar o processamento de *stream* de dados. Por exemplo, quando o recebimento dos dados acontece em tempo real e não pode ser armazenado completamente. O armazenamento desses dados depende de análises que após, realizadas, permitem o armazenamento do subconjunto relevante do *stream* de dados. Outro exemplo, quando aplicações enviam uma resposta imediata a estes dados. Essa situação tem crescido com a disseminação da Internet ligada a dispositivos móveis e jogos online.

4.1 Tecnologias de *Big Data*

O crescimento acelerado na produção de dados afetou profundamente os sistemas de bancos de dados tradicionais como, por exemplo, os bancos relacionais. Sistemas tradicionais e suas técnicas para o gerenciamento de dados enfrentam dificuldades no gerenciamento de *Big Data*. Nesse contexto, um novo conjunto de técnicas e tecnologias emergiram para o tratamento adequado dos desafios de *Big Data*.

Hadoop [9] e MapReduce [8] foram as principais tecnologias envolvidas em desafios de *Big Data*. Elas surgiram como tecnologias novas e velozes para a extração de valor de negócio a partir de grandes volumes de dados. O projeto Hadoop se tornou uma das mais populares implementações do MapReduce, suportado pela Fundação Apache. HadoopMapReduce é uma implementação Java do framework criado pelo Google. O Hadoop foi originalmente desenvolvido na Yahoo para gerenciar e analisar conjuntos de dados na Web e foi rapidamente adotado por outras instituições.

O projeto Hadoop é um projeto para o desenvolvimento de *software* de código aberto, distribuído, escalável e seguro. Para atingir essas metas, o *framework* Hadoop executa aplicações em grandes *clusters* em *hardware* comodite. O Hadoop implementa MapReduce, que permite aos aplicativos serem divididos em diversas tarefas menores, de maneira que cada tarefa pode ser executada em qualquer nó do cluster. Os dados são armazenados em um Nó específico, utilizando o *Hadoop Distributed File System* (HDFS). O HDFS é uma implementação de código aberto do *Google File System* (GFS), que fornece um canal de comunicação agregado através do cluster. Ambos, MapReduce e HDFS, são projetados para permitir o gerenciamento de falhas automaticamente pelo framework [9].

Existem outras tecnologias que podem ser agrupadas pelo termo *Not-Only SQL* (NoSQL). Esses sistemas têm como principal característica a alta escalabilidade para o processamento de um grande conjunto de dados. Estas tecnologias utilizam efetivamente novas técnicas para o gerenciamento de *Big Data* [29]. Consequentemente, o armazenamento, manipulação e análise desses grandes volumes de dados são facilitados. Além de oferecem tempos de resposta nunca antes vistos, o uso de tecnologias de *Big Data* viabilizaram a extração de valor de grandes volumes em uma vasta variedade de dados, permitindo a captura, descobrimento e análises em curto espaço de tempo.

4.2 Bancos de dados NoSQL

Os sistemas de bancos de dados *Not-Only SQL*(NoSQL) surgiram no inicio do século XXI como uma alternativa aos tradicionais bancos de dados relacionais (*Relational Database Management Systems* (RDBMS)). Eles são bancos de dados distribuídos, projetados para atender a demanda de alta escalabilidade e tolerância a falhas no gerenciamento e análises de grandes volumes de dados, os conhecidos desafios de *Big Data*. No decorrer dos últimos 10 anos, surgiram várias tecnologias NoSQL, cada uma delas com suas particularidades, vantagens e desvantagens.

NoSQL é uma classe de sistemas de bancos de dados caracterizados pela não aderência ao modelo relacional. Os bancos de dados NoSQL são utilizados quando armazenar e consultar grandes volumes de dados é mais importante que as relações entre os elementos desses dados. Os bancos de dados NoSQL funcionam em sistemas distribuídos que oferecem escalabilidade a partir da utilização de múltiplos nós computacionais. Ao contrário dos modelos relacionais, os bancos de dados NoSql não disponibilizam uma consistência rigorosa aos dados, como a definida pelos princípios de Atomicidade, Consistência, Isolamento e Durabilidade (ACID).

De fato, essa definição de consistência é bastante rigorosa e não é necessária em alguns casos, especialmente se estamos trabalhando em um ambiente distribuído. Contrário ao restrito modelo ACID, os bancos de dados NoSql são baseados na teoria de Consistência, Disponibilidade e Tolerância a particionamento (Consistency, Availability and Partition Tolerance (CAP)), que é aplicado para todos os sistemas de armazenamento de bancos de dados [73].

A Figura 4.3 ilustra o teorema de CAP. O termo consistência indica que todos os clientes devem possuir a mesma visão do dado. A disponibilidade refere-se à capacidade de permitir que um cliente sempre consiga ler e escrever dados. A tolerância ao particionamento indica que o sistema deve trabalhar como o esperado, independentemente da quantidade de particionamentos na rede física. De acordo com [73], um sistema de armazenamento de bancos de dados só consegue atingir duas dessas características.

Os modernos sistemas de bancos de dados relacionais (Relational Database Management System (RDBMS)) escolhem por consistência e disponibilidade, mas não conseguem aderir a tolerância de particionamento. Por outro lado, os bancos de dados NoSQL optam por tolerância ao particionamento e consistência ou disponibilidade. Além disso, diversos bancos de dados NoSQL reduzem os critérios de consistência para atingir melhor disponibilidade e particionamento. O resultado são sistemas conhecidos como sistemas *Basically Available, Soft-state and eventually consistent*

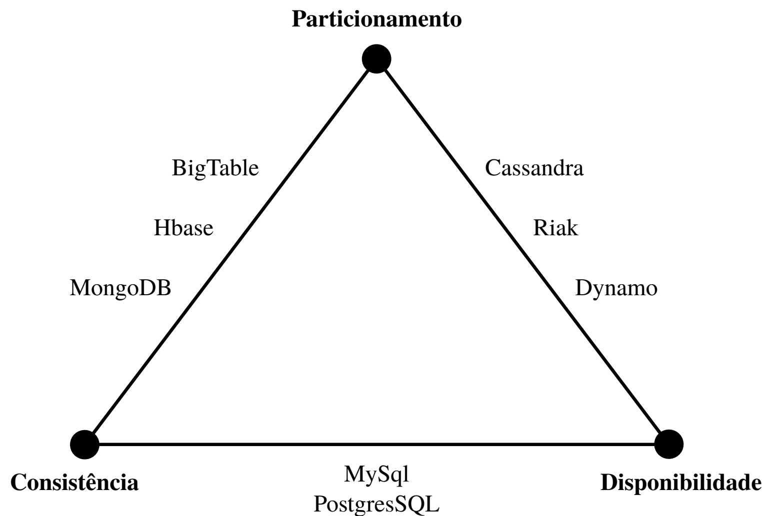

Figura 4.3: Teorema de CAP

(BASE) [74].

No contexto dos bancos de dados NoSQL, existem três categorias principais: *key-value*, *column-family* e *document-based* [75]. Conforme a Figura 4.3, o teorema de CAP é aplicável à esses bancos de dados NoSQL. Eles podem funcionar como conjuntos de dados em memória ou com armazenamento em disco [75].

4.3 *Stream* de dados

Na atualidade, diversos tipos de aplicativos produzem grandes volumes de dados em tempo real. Esses dados são enviados para que os servidores executem diversos tipos de processamento. Entre os aplicativos, podemos relacionar: monitoramento de dados de sensores, dados de negociação de ações em bolsas de valores, tráfego de operações na web e monitoramento de redes de comunicação. Esses aplicativos enviam os dados para processamento no formato de sequência contínua de tuplas de eventos, conhecidos como *stream* de dados [76].

Nos últimos anos, o processamento *batch* em grandes *clusters* de computadores foi o ponto focal para o processamento distribuído[27]. Entretanto, os requisitos para o processamento de *stream* de dados são significativamente diferentes dos requisitos para o processamento *batch* [43].

- Processamento de stream de dados - Processa e analisa os dados antes do armazenamento. É um modelo encontrado em aplicações de baixa latência, geralmente com tempos de reposta de milisegundos.
- Processamento *batch* de dados - Processa e analisa os dados após o seu armazenamento. É praticado em processamentos agendados, resultando em tempos de reposta de horas ou até mesmo dias.

As soluções baseadas em Hadoop dependem do armazenamento prévio dos dados para o seu

processamento. Dessa forma, apesar de processar grandes volumes de dados em grandes *clusters* de computadores, as características desse processamento, assim como o seu tempo de resposta, é caracterizado como um processamento *batch*. A tabela 4.1 apresenta um comparativo entre as principais características desses dois paradigmas de processamento.

	Stream	***Batch***
Entrada	*Stream* de novos dados ou novos eventos	Conjuntos de dados
Tamanho	Infinito ou previamente desconhecido	Finito e conhecido
Armazenamento	Não armazena ou armazena parcela irrelevante em memória	Armazenado
Processamento	Única ou poucas passagens pelo dado	Múltiplas passagens pelo dado
Tempo	Poucos segundos ou até mesmo milissegundos	Horas ou dias

Tabela 4.1: Comparação entre o processamento *batch* e o processamento de *stream* de dados

Segundo [30], os sistemas para processamento de *stream* de dados possuem características distintas dos consolidados sistemas para processamento *batch*. De acordo com essas características, [30] estabelece os principais requisitos de um sistema de processamento de *stream* de dados.

- Mobilidade de dados Permitir que os dados continuem em movimento é fundamental para atingir baixas latências. Por isso, um processamento ativo com operações livres de bloqueio aumentam a mobilidade dos dados e evitam *overhead* adicional ao processamento.

- Consulta de dados

  Recuperar eventos de interesse em um *stream* de dados é uma funcionalidade altamente desejável. Recursos de consulta aos dados reduzem o custo de descoberta e extração dos dados a partir do *stream* de dados.

- Armazenamento de dados

  O armazenamento de dados específicos para referência posterior permite a otimização de rotinas, execução de Algoritmos preditivos, verificação e validação de resultados.

- Garantias de processamento e alta disponibilidade

  Garantias de processamento estão intimamente relacionadas com alta disponibilidade. A habilidade de recuperação após falhas é crítico e deve funcionar de uma maneira eficiente e com baixas latências.

- Particionamento de dados

Fundamental para a boa escalabilidade da solução. O particionamento de dados permite o uso da computação distribuída para o processamento do *stream* de dados. A estratégia adotada para o particionamento impacta diretamente em como o sistema deve tratar o grande volume de dados.

- Tratamento de inconsistências no *stream* de dados

  A capacidade de tratar os dados que são entregues desordenados e fora de tempo, sem operações que bloqueiem o processamento é fundamental para atingir baixar latências. Em sistemas de *near real-time* os dados são processados antes de serem armazenados, dessa forma, o processamento do stream de dados deve ser capaz de tratar inconsistentes e desordenação dos eventos existentes nesses streams.

- Garantias de resultados consistentes e previsíveis

  Diferentes execuções do processamento de um mesmo dado devem produzir resultados previsíveis, garantindo que o processo é determinístico e repetível. Além disso, resultados previsíveis em diferentes execuções de um mesmo processamento de dados, é fundamental para a tolerância a falhas.

- Baixo tempo de resposta

  Um fluxo de execução de baixa latência, altamente otimizado e com *overhead* mínimo, que permita o processamento de grandes volumes de dados em tempo real.

Os requisitos estabelecidos por [30] são extremamente relacionados às características de alta disponibilidade e tempo de resposta do processamento. Essas são características essenciais para a análise de dados em *near real-time*.

De acordo com [77], o processamento de *stream* de dados pode ser dividido em três diferentes áreas: sistemas *query-based*, algoritmos de pesquisa online e as plataformas de *streaming* de uso geral (Figura 4.4).

- Sistemas *Query-based*

  Os Sistemas *Query-based* utilizam linguagens de consulta em alto nível para a execução de processamento de *stream* de dados. Esses sistemas têm origem na pesquisa de RDBMS. Eles utilizam linguagens de consulta semelhantes a dialetos baseados *Structured Query Language* (SQL), geralmente acrescidas de instruções para especificar janelas de processamento utilizadas no cálculo de agregações ou para especificar o intervalo de resultados.

- *Algoritmos de pesquisa online*

  Algoritmos de pesquisa *online* é uma área que explora diferentes aspectos em algoritmos capazes de computar resultados em *streams* de dados. Nessa área, encontramos pesquisas relacionadas a Algoritmos *sketch-based* que permitem a estimativa de valores sem reprocessar o *stream* de dados [37] [78]. As características desse conjunto infinito e desordenado de dados transformam problemas que são facilmente solucionados em uma abordagem estática,

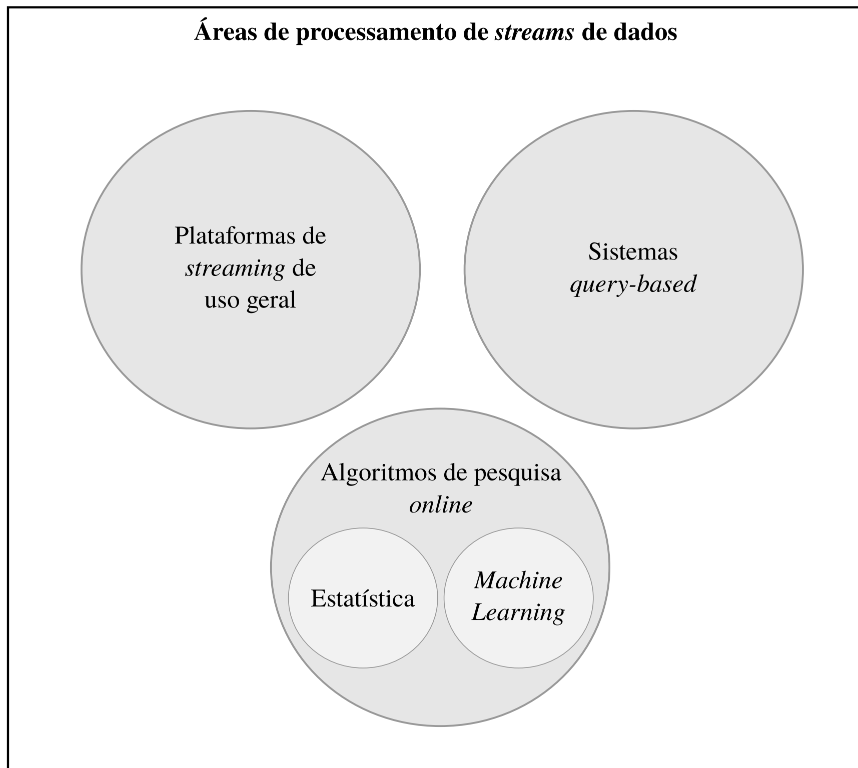

Figura 4.4: Áreas do processamento de *stream* de dados

em processamentos complexos, principalmente pelas limitações de recursos computacionais como armazenamento e processamento.

- Plataformas de *streaming* de uso geral

  As plataformas de *streaming* de uso geral surgiram a partir da necessidade de processar dados continuamente e permitir o desenvolvimento de aplicações de negócio específicas que utilizem esses dados. As áreas de algoritmos e sistemas *query-based* atuam em problemas específicos, enquanto as plataformas de uso geral disponibilizam plataformas para a execução de sistemas para o processamento de *stream* de dados com escalabilidade e tolerância a falhas.

A demanda pelo processamento contínuo de dados fomentou o surgimento de inúmeros sistemas para o processamento de *stream* de dados. Esses sistemas são conhecidos como *Distributed Processing Systems* (DSPS). Eles surgiram para facilitar o processamento de dados em larga escala para a análise em *near real-time* [79] [33] [35].

Os DSPS apresentam um modelo de processamento dos *stream* de dados que deriva do modelo de programação *data driven*. Esse modelo pode ser representado em forma de grafo, composto por dois tipos de elementos: fonte de dados (*stream*) e unidade de processamento [80].

De acordo com [80], é possível fazer uma abstração conceitual do modelo implementado em diferentes DSPS. Dessa forma, as unidades de processamento podem ser definidas de forma genérica como *Processing Element* (PE) [81].

A Figura 4.5 é uma representação desse modelo conceitual. Nessa representação, os PEs são

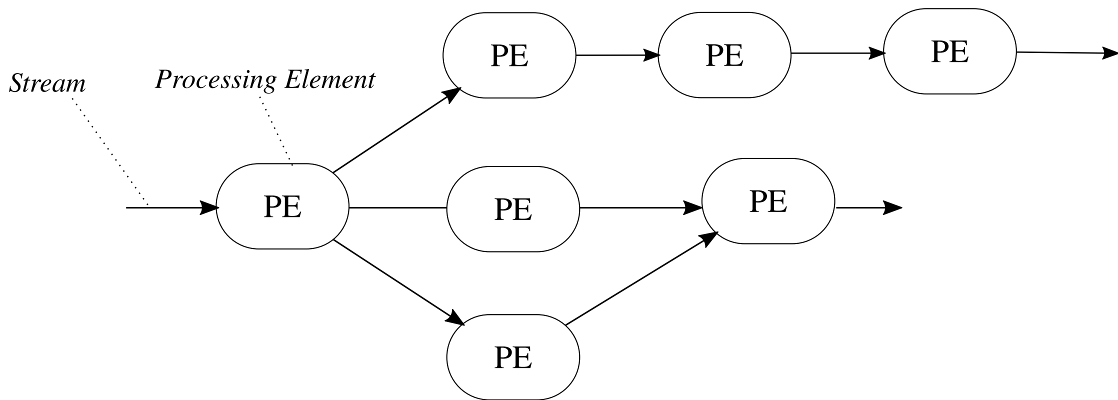

Figura 4.5: Representação lógica do processamento de *streams* de dados

nós de um grafo. Os *streams* são as fontes de dados representados por setas que interligam os PEs.

Essa abstração conceitual pode ser observada na implementação dos modelos utilizados em diferentes DSPS [33] [35] [31] [81].

A execução do processamento de *stream* de dados distribuído começa com a definição desse fluxo de processamento. O DSPS, por sua vez, é responsável pela execução do fluxo. Esse processamento pode acontecer em um único nó computacional ou de acordo com a necessidade, executado em paralelo em múltiplos nós computacionais.

A distribuição de um DSPS em *clusters* computacionais possibilita que o próprio DSP gerencie o número de contextos de execução e o balanceamento destes contextos entre os nós computacionais disponíveis. Além disso, os nós computacionais tratam as falhas de execução e - dependendo da implementação de tolerância a falhas do DSPS - podem reiniciar os PEs e executar o *stream* de dados a partir do último ponto de checagem.

4.4 Algoritmos probabilísticos

As características do processamento de grandes volumes de *stream* de dados, caracterizado principalmente por seu conjunto infinito e desordenado de dados, transformam problemas facilmente solucionados em uma abordagem centralizada, em processamentos complexos, principalmente pelas limitações de recursos computacionais como armazenamento e processamento.

Em estudos recentes, os algorítimos probabilísticos têm sido aplicados para estimar a frequência e cardinalidade de elementos em um conjunto de dados, com uma pequena fração da memória utilizada com algorítimos convencionais [36] [20]. Essas estimativas incluem uma pequena imprecisão no cálculo desses indicadores, o que dificulta sua aplicação para indicadores que envolvem, por exemplo, o risco de vida.

Os algorítimos probabilísticos também têm sido alvo de pesquisas que relatam a sua eficiência em análises de dados na internet, como contagem de endereços IP distintos que realizaram uma operação específica, detecção de anomalias, contagem de cliques e análises da eficiência em

propagandas em sites de internet [39].

Para o processamento dos indicadores dessa pesquisa, testamos e comparamos os algorítimos probabilísticos "estado da arte", HyperLogLog [38] para contagem de elementos distintos, e Count-min sketch [37] para o cálculo de itens frequentes.

Ambos os algoritmos são beneficiados da execução em ambiente distribuído. Eles suportam a mesclagem de resultados obtidos a partir da estimativa de subconjuntos do *stream* de dados. Além disso, permitem configurar a quantidade de recursos computacionais utilizados, de maneira que o maior uso desses recursos resulta em melhora na precisão das estimativas.

Para o desenvolvimento deste trabalho, foi utilizada uma implementação em linguagem Java desses algoritmos. Essa implementação é disponibilizada através de licença *open source* pela biblioteca Stream-lib [82].

4.4.1 HyperLogLog

A estimativa de cardinalidade permite identificar, em um conjunto de dados, quantos elementos estão duplicados, contar quantas vezes um único usuário clicou em um anúncio ou, ainda, monitorar redes de computadores.

No processamento de grandes volumes de dados, a cardinalidade de um conjunto é um desafio para o processamento de dados em larga escala. O cálculo preciso dos elementos distintos, em um conjunto de dados, requer uma quantidade expressiva de recursos computacionais.

O algoritmo para estimativa de cardinalidades HyperLogLog é o resultado de uma sequência de trabalho iniciado por Flajolet em [83], sendo aperfeiçoado em [84] e [38]. O resultado é um algoritmo que permite calcular cardinalidades acima de 10^9 com um erro padrão de 2%, consumindo de somente 1.5 kilobytes.

A lógica do HyperLogLog é representada pelo pseudocódigo 1, que descreve as operações de inclusão de elementos e calculo da cardinalidade do conjunto de dados oferecidos ao algorítimo. Para a inclusão de novos elementos, o algoritmo aplica uma função *hash* em cada elemento recebido (linha 5). Em seguida, o algoritmo separa a representação binária desse *hash*, de acordo com a precisão configurada para o algoritmo, em *header*(linha 6) e *body* (linha 7). Em uma estrutura interna, ele registra a posição do 1 mais a esquerda na variável *body* indexado pelo valor da variável *header* (linha 8).

A função *cardinality* (linha 10) utiliza os valores inseridos na estrutura de dados do algoritmo para estimar o número de elementos distintos. Primeiramente, as variaveis de controle são inicializadas (linhas 11-13). Em seguida, a estrutura de dados interna é iterada (linhas 14-21). Nessa iteração, o algoritmo sumariza a quantidade de elementos, consultando a estrutura de dados interna (linhas 15-16). Finalmente, após aplicar a lógica de correção de erro (linhas 23-27), o valor estimado da cardinalidade é retornado (linha 28).

Algorithm 1 Pseudocódigo do HyperLogLog

```
1: procedure HyperLogLog(accuracy)
2:     registerSet ← createRegisterSet(accuracy)
3:     alphaMM ← calculateAlphaMM()
4:     function offer(item)
5:         hValue ←murmu rHashing(item)
6:         header ←first accuracy bits of hValue
7:         body ←hValue stripped of initial accuracy bits
8:         registerSet ← max(registerSet.get(header), leftmost 1 position of body)
9:     end function
10:    function cardinality
11:        rSum ← 0
12:        count ← registerSet.count
13:        zeroes ← 0.0
14:        while counter < registerSet.size do
15:            val ← registerSet.get(counter)
16:            rSum ← 1.0/(1shiftedleftval
17:            if val = 0 then
18:                zeroes ← +1.0;
19:            end if
20:            counter ← +1
21:        end while
22:        estimate ← alphaMM ∗ (1/rSum)
23:        if estimate ≤ (5.0/2.0) ∗ count then
24:            lCount ← linearCount(count,zeroes)
25:            returnVal ← round(lCount)
26:        else
27:            returnVal ← round(estimate)
28:        end if
           return returnVal
29:    end function
30: end procedure
```

4.4.2 Count-min Sketch

A identificação de elementos frequentes é um problema para o processamento de *stream* de dados quando o tamanho conjunto de dados é indefinido e de crescimento contínuo. Além disso, identificar quais são os elementos mais populares em um conjunto de dados tem um custo computacional elevado.

O Count-min sketch é um algorítimo probabílistico para estimativa de valores, introduzido por [37], que pode ser aplicado para a sumarização de *stream* de dados, tendo como principais funções calcular frequências simples, identificar elementos frequentes e computar quantidades.

Ele permite economizar uma quantidade considerável de memória na sumarização de dados, além de suportar a mesclagem de resultados, o que possibilita sua execução de forma paralela e distribuída. O Count-min sketch realiza suas operações utilizando uma matriz de contadores de proporções configuráveis, representada na Figura 4.6.

$hash_1$	Mtr_{11}	Mtr_{12}	Mtr_{13}	Mtr_{14}	Mtr_{15}	...	Mtr_{1n}
$hash_2$	Mtr_{21}	Mtr_{22}	Mtr_{23}	Mtr_{24}	Mtr_{25}	...	Mtr_{2n}
$hash_3$	Mtr_{31}	Mtr_{32}	Mtr_{33}	Mtr_{34}	Mtr_{35}	...	Mtr_{3n}

$item_t$

Figura 4.6: Matriz de contagem do algoritmo Count-min sketch

O funcionamento do algorítimo, descrito no pseudocódigo 2, consiste em operações de inserção e consulta dos dados. Para a consulta de estimativas a ocorrência do elemento com o menor contador registrado é retornado.

Algorithm 2 Pseudocódigo do Count-min sketch

1: **procedure** COUNTMINSKETCH(depth, width)
2:     **function** ADD(item)
3:         **while** $counter < depth$ **do**
4:             $buckets \leftarrow$GETHASHBUCKETS(item,depth,width)
5:             $Mtr[counter][buckets[counter]] \leftarrow +1$
6:             $counter \leftarrow +1$
7:         **end while**
8:     **end function**
9:     **function** ESTIMATECOUNT(item)
10:         $r \leftarrow MAXVALUE$
11:         **while** $counter < depth$ **do**
12:             $val \leftarrow Mtr[counter][buckets[counter]]$
13:             $r \leftarrow$MIN(r, val)
14:             $counter \leftarrow +1$
15:         **end while**
        return r
16:     **end function**
17: **end procedure**

A atualização dos valores acontece com o suporte de uma *hash*, aleatória de uma família independente de pares, que é aplicada no elemento para, em seguida, atualizar seu contador em uma estrutura de dados previamente configurada.

Para adicionar elementos, o algoritmo Count-min sketch itera na profundidade da matriz (linhas 4-7) previamente configurada para, em seguida, aplicar uma função *hash* arbitrária (linha 4). De acordo com o *hash* obtido, o contador é incrementado e atualizado na matriz (linha 5).

Para retornar a estimativa da frequência, o algoritmo itera nos elementos da matriz(linhas 11-15) para buscar a menor ocorrência registrada para o *hash*(linhas 12-13) informado como parâmetro da função *estimateCount(item)*(linha 9).

Capítulo 5

Business Intelligence (BI) e *Business Analytics* (BA)

5.1 Introdução

O processo de tomada de decisões é baseado em dados históricos que foram produzidos há algum tempo. Desde a década de 1950, sistemas de *Business Intelligence* (BI), representado pela 5.1, são utilizado para processar informação e produzir indicadores que expressem informações relevantes para auxiliar o processo de apoio à decisão [85]. Os sistemas de apoio à decisão permitem aos gestores e executivos a tomada de decisões, subsidiados por informações previamente processadas de diferentes fontes de dados, retratando os mais diversos indicadores de uma corporação [86].

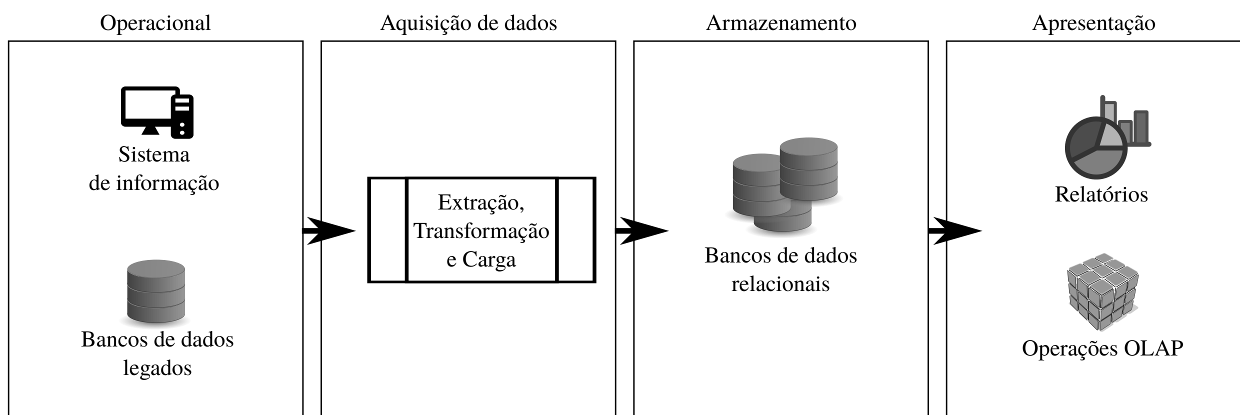

Figura 5.1: Visão geral de BI

O gerenciamento dos dados e o *Data Wharehousing* são fundamentos de soluções de BI convencionais. A modelagem de *Data Marts* e aplicativos de ETL são essenciais para converter e integrar as diferentes fontes de dados. As consultas a bases de dados, *Online Analytical Processing* (OLAP e relatórios intuitivos são amplamente utilizados para explorar os dados. As técnicas de Business performance management (BPM) suportam a análise e avaliação das métricas de performance. Além disso, técnicas consolidadas de análise estatística e mineração de dados são adotadas para

análises de correlação, segmentação, clusterização, classificação, análise de regressão, detecção de anomalias e preditividade de dados em diversas áreas de negócio. A maior parte dessas características podem ser encontradas em soluções de mercado, distribuídas pelas indústrias líderes de mercado como Microsoft, IBM, Oracle e SAP [87].

No final dos anos 2000, *Business Analytics* se tornou amplamente utilizado para representar o componente chave em uma solução analítica de BI [88]. Na atualidade, o termo *Big Data Analytics* tem sido utilizado para se referir às técnicas utilizadas em aplicações analíticas de grande volume e complexidade de dados. Essas técnicas avançadas e inovadoras são utilizadas em tecnologias para o armazenamento gerenciamento, análise e visualização dos dados [89].

De acordo com [7], um sistema de BI deve ser guiado pelos seguintes princípios:

- **Orientado ao negócio:**

  Os dados devem ser agrupados pelo que significam no assunto analisado, independentemente do sistema de origem ou localização física. Os dados não devem ser agrupados pelo departamento de origem, autor ou localização física.

  O nível de detalhe mantido por sistemas de informação em um contexto operacional é detalhado e modelado para atender um conjunto de requisitos necessários para atividades operacionais de setores menores de uma corporação. Esse tipo de informação não é adequado para o processo de tomada de decisão. Por essa razão, a orientação ao assunto visa à organização dos dados de acordo com o tema de negócio, o que permite agrupar diversas origens de dados em um único assunto, conforme ilustrado pela Figura 5.2. Nesse sentido, percebe-se que a integração entre diferentes bases de dados é um aspecto importante para a organização de um DW.

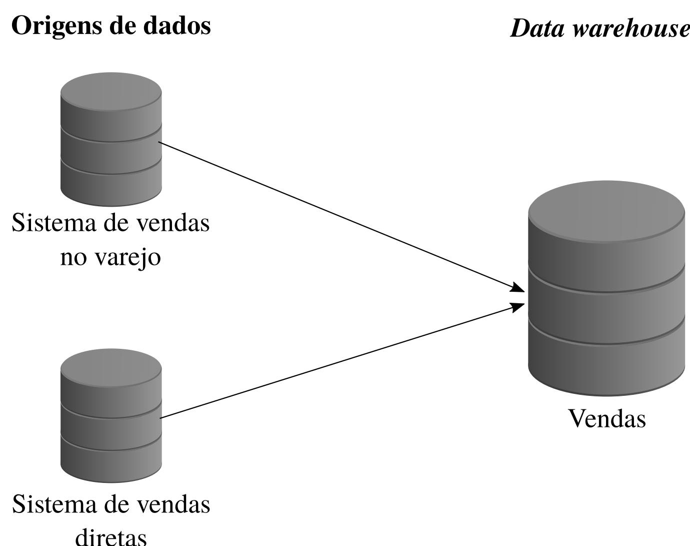

Figura 5.2: Fontes de dados operacionais e a orientação ao assunto

- **Integração de dados** Os dados devem ter o mesmo aspecto, independentemente de sua origem. Eles devem apresentar a mesma forma, função e grão.

- **Forma** – Um mesmo dado pode ter diferentes representações dependendo de sua origem. Por exemplo, um número de telefone pode ser armazenado em um Sistema A como (12) 1234-3211 e em um segundo Sistema B como 12 12343211. Um mesmo formato deve ser imposto na integração desses dados quando carregados em uma estrutura de BI.
- **Função** – Dois elementos diferentes, que significam a mesma informação, mas possuem códigos diferentes. Por exemplo, um parafuso pode ser registrado pelo código P10 em uma origem de dados, enquanto em outra, pelo código PAR. Esses códigos devem ser substituídos por um código único em uma solução de BI.
- **Grão** – O mesmo dado pode ser registrado com diferentes níveis de detalhe em várias origens de dados. A informação precisa ser traduzida para um mesmo nível de detalhe em uma solução de BI.

- **Não volátil** Em aplicativos operacionais, os dados podem ser descartados ao final de um processo de negócio. Em uma solução de BI, os dados armazenados em um *data warehouse* devem permanecer registrados, garantindo a informação histórica e a integridade do fato analisado.

- **Variável no tempo** Todos os dados são armazenados relacionados a um momento no tempo. Um *data warehouse* mantém essa relação. Por exemplo, informações registradas em 1990 estarão sempre relacionadas ao ano de 1990.

- **Investimento de longo prazo** Um *data warehouse* deve ser flexível o bastante para absorver as mudanças da corporação e do mundo, além de se manter escalável o suficiente para permitir o crescimento da corporação.

As soluções de BI e BA baseadas em uma abordagem centralizada de dados têm sua origem na área de gerenciamento de banco de dados. Essa técnica, representada pela Figura 5.3, consiste em analisar diversas fontes de dados, extrair e analisar os dados [90]. As tecnologias e aplicações de BI atualmente adotadas pela indústria consistem, em sua maioria, de soluções com dados estruturados, coletados a partir de diversos sistemas de *Online Transaction Processing application* (OLTP) e em sistemas de gerenciamento de bancos de dados relacionais comerciais (*Relational Database Management Systems* (RDBMS)). As técnicas aplicadas nessas soluções de BI foram popularizadas, em grande parte, nos anos 1990 e têm seus fundamentos em métodos estatísticos desenvolvidos nos anos 1970. Além disso, as técnicas de *data mining* utilizadas foram desenvolvidas nos anos 1980s [89].

5.2 *Data Warehouse*

O processo de *data warehousing* permite adquirir dados de diferentes origens, provendo uma base de dados completa e confiável para a consulta e análise desses dados. Em sistemas de BI, um *Data Warehouse* (DW) refere-se a um conjunto de dados consistente, integrado, orientado ao assunto, variável no tempo e não volátil necessários para o processo de tomada de decisão [13].

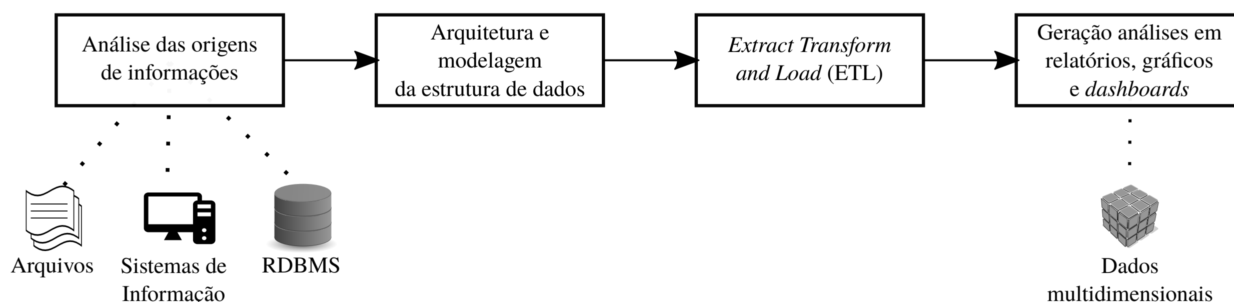

Figura 5.3: Fluxo de atividades para a concepção de uma solução de BI

A análise e consulta de dados em um DW raramente é ocorre diretamente em sua estrutura de dados. Os *Data Marts* (DM) são estruturas que envolvem os conceitos de DW, de modo a possibilitar a construção de subconjuntos menores de dados, com o objetivo de abordar um contexto de negócio específico, para permitir a consulta e análise de dados diretamente pelo usuário final [91].

5.3 Modelagem dimensional

A modelagem dimensional é uma técnica adotada para tornar possível a entrega e apresentação dos dados em uma estrutura padrão, intuitiva ao usuário final. O modelo dimensional permite analisar os fatos e métricas, com a construção dinâmica e de alta performance aos dados [7]. Ele é composto de tabelas fato, cercadas por tabelas de dimensões e também é conhecido em sua estrutura mais básica como *star schema* ou *star join*.

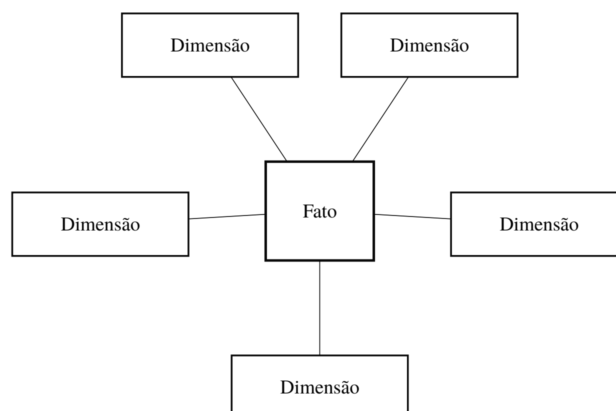

Figura 5.4: Representação conceitual do modelo de dados dimensional do tipo *star schema*

- **Tabela fato** é o principal elemento em um modelo dimensional. As tabelas fato são responsáveis por registrar eventos, transações ou um acontecimentos específicos no tempo e suas métricas.

- **Tabelas de dimensão** acompanham as tabelas fato, contendo atributos textuais básicos, que permitem agrupar e filtrar as análises de um fato. As dimensões registram uma permutação de mesma hierarquia de diferentes fontes de dados em uma corporação.

De acordo com a necessidade, o modelo dimensional *star schema* pode ser derivado em outras variantes como, por exemplo, os modelos *Snowflake schema* e *Fact constellation schema.*

- *Snowflake schema*

  O modelo dimensional *Snowflake schema*, representado pela Figura 5.5, deriva do modelo *Star schema* quando as hierarquias de uma dimensão são expandidas ou normalizadas em tabelas separadas. Essa abordagem permite economizar espaço em disco e atribuir características diferenciadas a alguns atributos da dimensão. Entretanto, ela resulta em maior lentidão e complexidade no processamento das consultas aos dados.

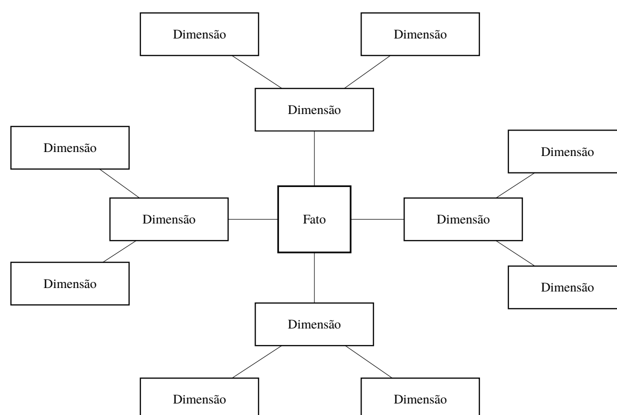

Figura 5.5: Representação conceitual do modelo de dados dimensional do tipo *Snowflake schema*

- *Fact constellation schema*

  O modelo dimensional *Fact constellation shcema*, representado pela Figura 5.6, que também é conhecido como *Galaxy schema*, é um abordagem que consiste na coleção de modelos *star schema* com tabelas fato ligadas hierarquicamente [92]. Essa abordagem possibilita a realização de operações de *drill down* entre os níveis de detalhe oferecidos pelas tabelas fato.

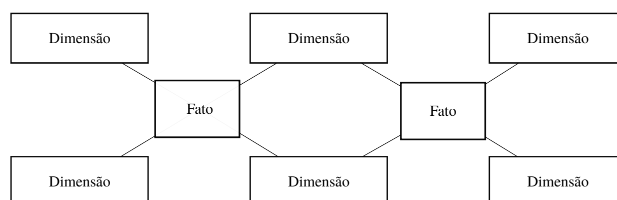

Figura 5.6: Representação conceitual do modelo de dados dimensional do tipo *Fact constellation schema*

5.4 *Extract, Transform and Load* (ETL)

O estágio preliminar da construção de uma solução de BI é o processo de ETL [7]. O processo de ETL, ilustrado pela 5.7, é responsável pela extração das informações de uma ou várias fontes de dados, sua transformação e consolidação e finalmente, por carregar estes dados em uma estrutura

Figura 5.7: Processo de extração, transformação e carga de dados (ETL)

de dados multidimensional para ser consultado e apresentado por um motor de *Online Analytical Processing* (OLAP).

De acordo com [7], o desenvolvimento de sistemas de ETL é um desafio para os projetos de sistemas de BI. São diversas limitações impostas para a modelagem dos sistemas de ETL: os requisitos de negócio, a realidade das origens de dados, orçamento disponível, tempo de resposta para análise dos dados e a qualificação das equipes envolvidas no desenvolvimento do sistema de ETL.

- *Extract*

  As etapas iniciais de um processo de ETL tratam do entendimento, extração e transferência dos dados para uma área controlada no ambiente de DW. Essa área controlada, conhecida como *staging area*, permite que o sistema de ETL manipule os dados de forma independente aos sistemas de origem. Nesse momento, os dados são mantidos em um formato idêntico ao encontrado nos sistemas de origem [7].

- *Transform*

  Em seguida, o sistema de ETL executa as transformações necessárias nos dados obtidos nos sistemas de origem. Essas modificações geralmente envolvem a mesclagem entre os dados de diferentes sistemas de origem, limpeza e conformação dos dados. Nessa etapa os dados ainda estão na *staging area* do sistema de ETL [13].

- *Load*

  Por fim, os dados são enviados aos servidores utilizados pelo usuário final. Nessa fase do processo, o resultado do processos de extração e transformação dos dados é registrado em servidores de banco de dados responsáveis pelo DW [13] [7].

Essa é uma abordagem bastante difundida e bem estabelecida para sistemas de ETL, que tem fornecido excelentes resultados para as soluções convencionais de BI. Entretanto, os novos cenários nois quais acontece a demanda pelo processamento de grandes volumes de dados, em tempos de reposta reduzidos, têm submetido o processo a diversas adaptações

Além disso, no cenário de *Big Data*, os sistemas de BI demandam sistemas de processamento de dados poderosos que executem algoritmos capazes de identificar padrões e relações desconhecidas em dados não estruturados. Esse tipo de abordagem requer plataformas de execução de alto

custo para as corporações que, de acordo com o tempo de resposta requerido pela análise, podem ser classificadas em dois paradigmas distintos: processamento de *Streaming* e processamento *Batch* [43].

Capítulo 6

Solução proposta

A difusão de sistemas que processam dados caracterizados como desafios de *Big Data* tem impactado profundamente a indústria. Um recente relatório do International Data Corporation (IDC) indica que, do ano de 2005 para 2020, o crescimento no volume global de dados produzidos acontecerá em um fator de 300, de 130 Exabytes para 40.000 Exabytes, representando um crescimento dobrado a cada dois anos [1].

De acordo com a explanação do Capítulo 5, a análise em tempo hábil desses dados é de vital importância para o sucesso das empresas. Nesse contexto, soluções de *Business Intelligence* (BI) visam a oferecer os meios necessários para a transformação de dados em informação e para a sua divulgação através de ferramentas analíticas, a fim de suportar o processo decisório [6]. Nesse cenário, sistemas para o processamento dos dados utilizam rotinas de Extração,Transformação e Carga (*Extract, Transform and Load*)(ETL) de uma ou mais fontes de dados necessárias para a composição da análise de informações [7].

Na atualidade, a diversidade de tecnologias e modelos conceituais de dados utilizados nesses sistemas elevam a complexidade envolvida no desenvolvimento de sistemas para a análise de dados em *near real-time*, tal complexidade tem impacto em atributos de qualidade do sistema de software, como por exemplo, escalabilidade e manutenibilidade.

Conforme representado na Figura 6.1, este trabalho propõe o uso de técnicas de arquitetura de sistemas de *software*, descritas no Capítulo 3, nas áreas de processamento de *stream* de dados sugeridas por [77] e explanadas no Capítulo 4, Seção 4.3: Plataformas de *streaming* de uso geral, Sistemas *query-based* e Algoritímos de pesquinsa *online*. Dessa forma, propõe-se o estudo e utilização de técnicas arquiteturais e de modelagem conceitual para promover o reuso de fluxos de processamento e subsidiar a gestão da complexidade envolvida no desenvolvimento de análises de dados em *near real-time*.

Nas seções a seguir, são apresentados os *Distributed Stream Processing Systems* (DSPS) avaliados durante a pesquisa. Em seguida, os algoritmos probabílisticos são explanados e a arquitetura do experimento descrita com a utilização das técnicas arquiteturais explanadas no Capítulo 3.

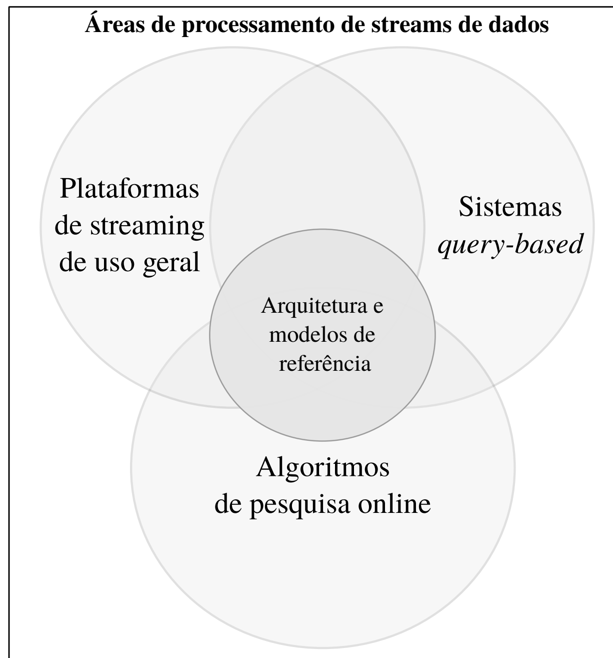

Figura 6.1: Proposta de áreas do processamento de *streams* sugerida pelo trabalho

6.1 Análise dos DSPS

De acordo com a fundamentação do Capítulo 4, Seção 4.3, diferentes DSPS emergiram com o objetivo de suprir a necessidades dos novos sistemas de análise de dados. Nesse trabalho são estudados os seguintes DSPS: Apache STORM [33], Apache SAMZA [35] e Borealis [31]. Esses DSPS utilizam diferentes estratégias, modelos e tecnologias para a implementação de rotinas de processamento de *stream* de dados.

A seguir, a estratégia de processamento e o funcionamento desses DSPS é alvo de estudo. Dessa forma, serão analisados os sistemas com o objetivo de comparar suas características e identificar as semelhanças entre seus modelos de processamento de *stream* de dados.

6.1.1 Borealis

O Borealis [31] é um DSPS desenvolvido a partir do sistema Aurora, um sistema de processamento de *stream* desenvolvido pela Universidade de Brown e o MIT para o processamento centralizado de *streams* [32].

Cada servidor Borealis possui um processador de consultas, representado pela Figura 6.2 é composto por: um motor de processamento interno para a execução individual dos PEs, um gerenciador de armazenamento para guardar os *streams*, um balanceador de carga para distribuir o processamento em situações de sobrecarga e um agendador de operações que determina a ordem de execução os PEs de acordo com a prioridade das tuplas. Além disso, cada nó é equipado de um otimizador que utiliza informações locais e remotas para distribuir a carga através dos servidores.

O seu funcionamento consiste em diversos servidores Borealis interligados para permitir a exe-

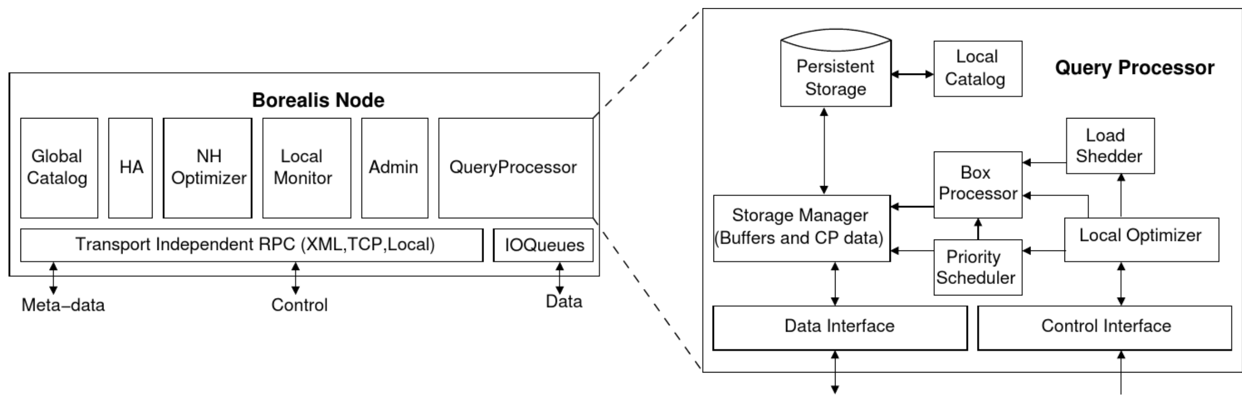

Figura 6.2: Composição de um servidor Borealis [31]

cução coordenada dos processamentos de dados. Cada servidor possui um processador de consultas para a execução do fluxo de processamento fornecido. A administração dos servidores não é centralizada em um nó específico. Todos os servidores têm um módulo de administração, onde é possível informar se o fluxo de processamento precisa ser executado de forma local ou remota.

6.1.2 Apache Storm

O Apache Storm [33] é um DSPS desenvolvido pelo Twitter e posteriormente disponibilizado à comunidade como um projeto da Apache Software Foundation. O principal objetivo da modelagem do Storm é garantir o processamento de todos os dados recebidos. Para isso, ele anexa um identificador de 64bits, gerado aleatoriamente, para cada tupla de dados recebido pelo sistema. Esse identificador é verificado ao final do processamento por um mecanismo reverso que valida se todos os identificadores recebidos foram processados.

O seu funcionamento, representado pela Figura 6.3, consiste em um *cluster* de servidores composto por três tipos de nós: Nimbus, Zookeeper e Supervisores. Nessa topologia de servidores, os nós Nimbus [93] são responsáveis por distribuir e coordenar a execução do fluxo de processamento, os nós Zookeeper [94] gerenciam o estado do *cluster*; os nós supervisores disponibilizam um ambiente para a execução do fluxo de processamento.

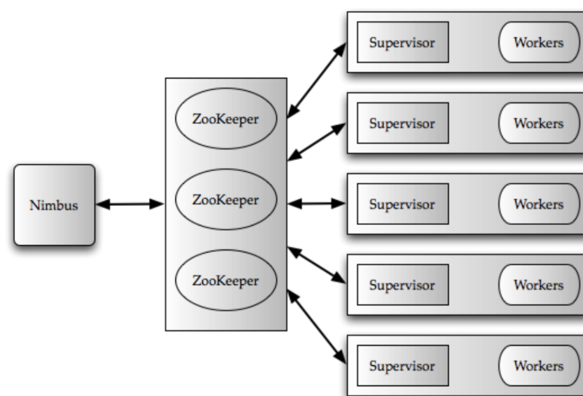

Figura 6.3: Composição de um *cluster* Apache Storm [33]

O processamento de dados com o Storm consiste em fluxos de processamento compostos de *spouts* e *bolts*. Os *spouts* recebem dados a partir de *stream* de dados. Os *bolts* consomem e processam os dados recebidos dos *spouts* e os enviam para o próximo conjunto de *bolts*.

6.1.3 Apache Samza

O Apache Samza é um DSPS desenvolvido pelo LinkedIn e posteriormente incubado como projeto *open-source* através da Apache Software Foundation. Ele é um sistema desenvolvido sob a plataforma Apache Hadoop [9] e altamente integrado com o Apache Kafka [34].

O seu funcionamento, representado pela Figura 6.4, consiste na distribuição de *Task Runners*, que são o contextos de execução para os *Samza Jobs* em uma plataforma gerenciada pelo Apache Hadoop YARN [9]. Essa plataforma provê uma API desacoplada de alto nível, que permite alocar *containers* de execução em um *cluster* de servidores. Além disso, o YARN é responsável pela tolerância a falhas na execução de processamentos, uma vez que o YARN pode reiniciar os processos na ocorrência de falhas. O Apache Kafka é utilizado como estrutura de dados distribuída. Ele provê um sistema de filas de alta disponibilidade que garante a entrega de todas as mensagens gerenciadas pelo sistema.

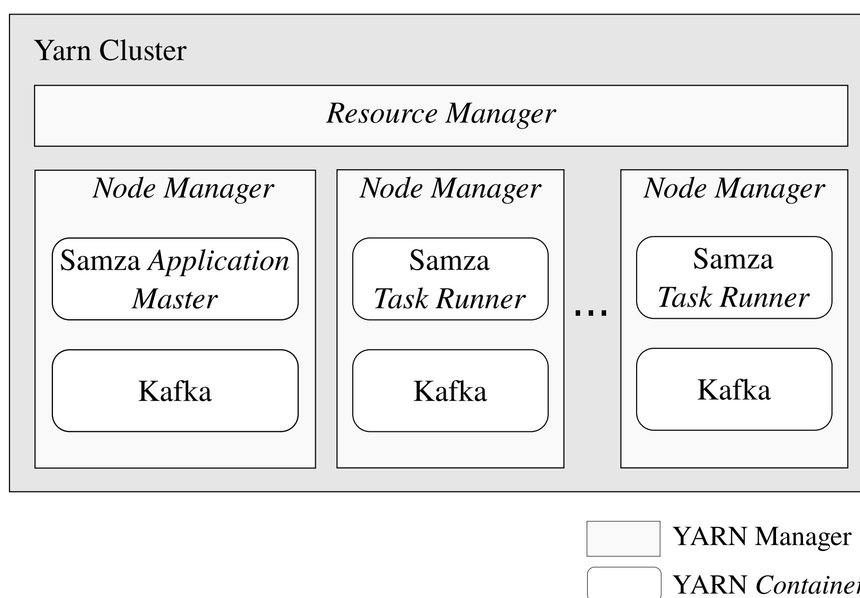

Figura 6.4: Distribuição do Apache Samza em *cluster* Apache Hadoop YARN

O processamento de dados com o Samza consiste na execução de Samza *Jobs*, conectados por *streams* de dados de entrada e saída. Cada Samza *Job* é composto por uma entrada e saída de dados e uma *Stream Task*, executados para cada tupla de dados recebida pelo *Samza Job*. A execução de um Samza *Job* é coordenada pelos *Task Runners*, que gerenciam a quantidade de *Stream tasks* que devem ser instanciadas para a execução do fluxo de processamento de dados.

6.2 Arquitetura de referência

A diversidade de tecnologias, modelos e arquiteturas no processamento de *stream* de dados eleva a complexidade na análise desses dados. Além disso, o desenvolvimento e manutenção desses sistemas de processamento de dados se tornam atividades de alto custo para as organizações.

Nessa seção, descrevemos uma arquitetura referência para o processamento distribuído de *stream* de dados. Essa arquitetura refere-se às estruturas de mais alto nível de abstração para o processamento distribuído de *streams*. Para isso, ela reúne uma série de elementos arquiteturais organizados de modo a satisfazer aos princípios arquiteturais definidos.

Essa arquitetura de referência é resultado das lições aprendidas durante esta pesquisa. Uma arquitetura de referência captura a essência de arquiteturas de *software*, e a visão de necessidades futuras, provendo um guia para o desenvolvimento de novas arquiteturas [45]. Elas estabelecem um conjunto de recomendações e restrições, que tendem a reduzir o esforço e tempo necessários em novos projetos de sistemas de *software* [46].

Ela propicia o aproveitamento de decisões arquiteturais e pretende subsidiar a gestão da complexidade no processamento distribuído de *streams*. Além disso, uma arquitetura de referência pode ser base para a definição de novas arquiteturas de referência. Estas, por sua vez, seriam pilares para as arquiteturas de sistemas de processamento distribuído de dados.

A Figura 6.5 é uma representação conceitual dessa arquitetura. Essa visão modular permite relacionar as camadas de *software* com o uso de conceitos e metodologias auxiliares.

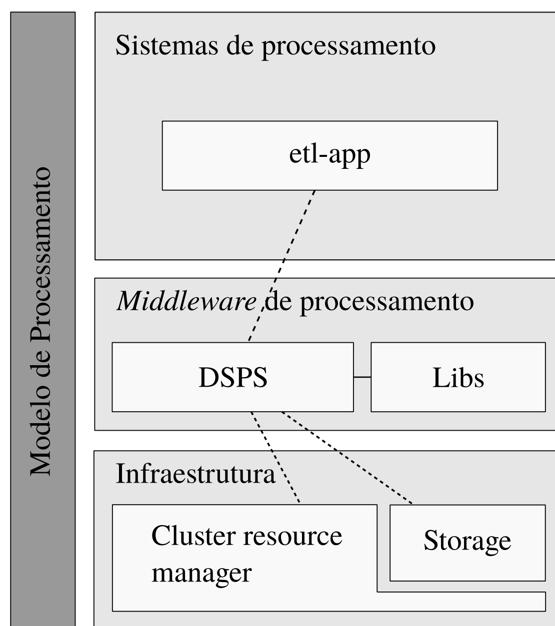

Figura 6.5: Visão geral da arquitetura de referência

A arquitetura de referência proposta define uma camada conceitual para a modelagem de fluxos de processamento. Dessa forma, uma notação para a representação é estabelecida, além de um modelo comum, independente de tecnologia paralelo as camadas da arquitetura.

As camadas propostas são organizadas de acordo com suas responsabilidades com o objetivo de desacoplar a lógica de processamento da lógica de implementação dos DSPS e das bibliotecas auxiliares, e estes por sua vez, da complexa lógica necessária para o uso de infraestrutura distribuída e de alta disponibilidade.

Nas seções a seguir, o modelo de processamento e as camadas que compõe essa arquitetura de referência são descritos.

6.2.1 Modelo de processamento

O Modelo de Processamento estabelece um padrão para a modelagem e execução de fluxos de processamento. Esse modelo provê o suporte conceitual para todas as camadas da arquitetura. Dessa forma, essa camada estabelece um modelo de referência para o processamento distribuído de *streams* de dados. Ela propõe uma abstração, independente de tecnologia, para a modelagem de fluxos de processamento de dados.

A Figura 6.6 utiliza um diagrama da UML para representar os elementos que compõe esse modelo de processamento.

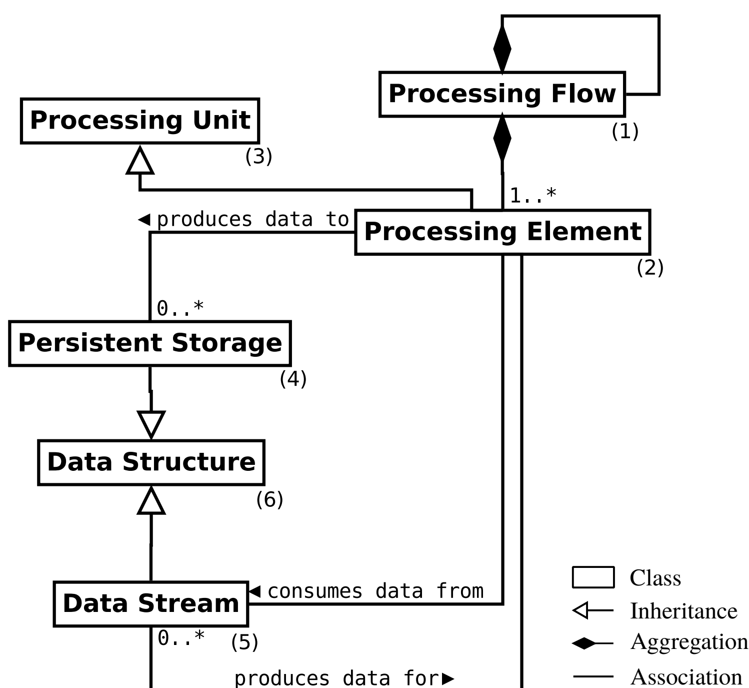

Figura 6.6: Diagrama da UML representando o metamodelo para modelagem de fluxos de processamento

Esse metamodelo descreve as relações entre os elementos que compõem o modelo de referência proposto para a elaboração de fluxos de processamento. Essas relações e suas cardinalidades são representados com o uso de notação formal, através de diagrama da *Unified Modeling Language* (UML).

De acordo com o metamodelo na Figura 6.6, o fluxo de processamento (1) pode ser composto de

outros fluxos de processamento e *Processing Elements* (PEs) (2). Uma PE é uma especialização de uma unidade de processamento (3) que contém a lógica de negócio específica de um processamento de dados. Cada PE pode produzir dados para um ou vários armazenamentos persistentes (4) e um ou vários *streams* de dados (5). Os armazenamentos persistentes e *streams* são especializações de uma estrutura de dados (6).

A Figura 6.7 apresenta uma simbologia proposta por esse trabalho para a representação dos dois principais elementos do modelo proposto: *Processing Element* (PE) (a) e *Streams* (b). Essa simbologia busca a representação desses elementos com uma notação amigável e de fácil entendimento.

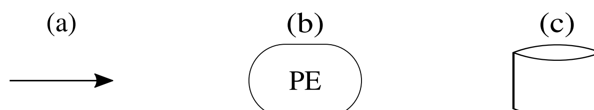

Figura 6.7: Elementos básicos do modelo de processamento proposto

- *Processing Element* (PE)

  O *Processing Element* (PE) corresponde a uma unidade de processamento. Uma PE permite a organização do código em unidades menores, responsáveis por uma lógica específica. As PEs organizam o fluxo de processamento em unidades menores, o que promove o reuso dessas unidades em diferentes fluxos, além de subsidiar o paralelismo do processamento de dados.

- *Stream*

  O *Stream* de dados corresponde à estrutura de dados para o transporte das tuplas recebidas para processamento. Os *streams* são utilizados como entrada e saída de dados nas PEs que compõem um fluxo de processamento.

- Armazenamento Persistente

  O armazenamento persistente refere-se à um SGBD externo, utilizado para registrar resultados de processamento, ou ainda, para realizar o gerenciamento de estado de processamentos complexos. Esse elemento de dados deve ser utilizado como saída de dados de um PE.

As PEs permitem a organização da lógica de processamento em unidades menores. Essa organização permite o uso de um mesmo PE em diferentes fluxos de processamento. Dessa forma, o reuso da lógica implementada, promovido por essa organização, pode ser considerado um grande benefício do modelo proposto.

6.2.2 Camada de sistemas de processamento

A camada de sistemas de processamento permite abstrair toda lógica necessária para o processamento distribuído de *stream* de dados. Ela contém apenas os componentes de *software* que implementam a lógica de processamento necessária para atingir os requisitos de negócio do sistema de análise de dados.

Essa camada permite organizar a lógica de processamento em componentes de *software* catalogáveis em repositórios institucionais. Além disso, permite a abstração de particularidades relacionadas as tecnologias para o processamento distribuído de *stream* de dados.

Dessa forma, a separação da lógica de processamento aumenta o reuso da lógica de processamento. A organização da lógica de processamento em pequenos componentes de *software*, além de promover o reuso, reforça a manutenabilidade desses componentes.

6.2.3 Camada de *Middleware* de processamento

A camada de *Middleware* protege a camada de Sistemas de processamento das complexidades relacionadas à heterogeneidade de tecnologias e detalhes de comunicação dos DSPS e bibliotecas auxiliares com a camada de infraestrutura. Dessa forma, a camada de processamento precisa conter somente a lógica do processamento dos dados.

Essa camada da arquitetura permite a organização e integração das tecnologias necessárias para o processamento de *stream* de dados. São encontrados nessa camada componentes de *software* dos DSPS, APIs para comunicação com bancos No-SQL, bibliotecas auxiliares para qualidade de dados, algoalgoritmos*machine learning* e estatística.

A abstração fornecida para a camada de sistemas de processamento garante independência para a lógica de processamento. Essa abstração é essencial para o reuso da lógica e flexibilidade para a substituição de componentes específicos da camada de *middleware* de processamento.

6.2.4 Camada de Infraestrutura

A camada de infraestrutura disponibiliza para a camada de *middleware* de processamento, uma interface de acesso aos recursos computacionais necessários para distribuição e execução dos sistemas de processamento de dados. Essa camada utiliza tecnologias de processamento distribuído para abstrair a lógica para acesso a recursos computacionais instalados em redes privadas ou na nuvem.

As principais atribuições da camada de infraestrutura são: disponibilizar áreas de armazenamento no disco, acesso a memória volátil e capacidade de processamento requerido para a execução dos fluxos de processamento de dados.

Dessa forma, as camadas de *Middleware* de processamento e Sistemas de processamento ficam isentas das particularidades relacionadas ao gerenciamento dos recursos computacionais necessários para a execução e distribuição dos componentes de *software* do processamento de dados.

Capítulo 7

Resultados experimentais

7.1 Introdução

Neste capítulo apresentamos os resultados obtidos com o processamento do *stream* de dados para a composição da análise de três cenários distintos utilizados para avaliar a arquitetura de referência proposta.

O primeiro permite uma análise da frequência de um endereço IP específico. O segundo sumariza o total de endereços IP distintos processados. Finalmente, o terceiro cenário consiste na quantidade de endereços IP distintos agrupados pelo país de origem.

Para a avaliação da solução proposta nesse trabalho, foram produzidos dados que correspondem a cliques de usuário em um anúncio, disponível em uma página da web. De acordo com a Tabela 7.1, o volume total de dados produzidos equivalem a 1.20GB de dados, contendo endereço de IP de visitante, país de origem do visitante e o registro de data e hora do clique.

Endereço IP	**País de origem**	***Timestamp***
192.10.56.10	BR	1410351571
192.3.160.45	AR	1410351563
192.160.56.23	US	1410351550
...	...	...

Tabela 7.1: Estrutura e amostra dos dados utilizados no experimento

Os valores de cada coluna foram produzidos aleatoriamente e armazenados para permitir a repetição dos testes. O atributo dos endereço IP é gerado seguindo padrão: 192.[0-254].[0-254].[0-254]. O país de origem é aleatoriamente escolhido em um domínio de siglas: {"BR", "US", "ES", "MX", "JP", "AR", "IT", "SE"}. O campo *timestamp* corresponde à representação numérica da data e hora do evento.

Um programa que simula o envio contínuo dessa massa de dados foi desenvolvido para simular

a ocorrência dos cliques em uma página web. Dessa forma, foi possível avaliar diferentes estratégias de processamento, eliminando variações dos dados que possam impactar nas comparações.

A Tabela 7.2 relaciona os cenários e as estratégias definidas para o processamento dos cenários. Os processamento dos cenários 1 e 2 ocorre com o uso de algorítimos probabilísticos. Essa abordagem permite estimar o valores a partir de grandes volumes de dados, com a inclusão de uma pequena imprecisão em troca da economia de recursos computacionais na sumarização dos valores [95].

Código	**Cenário**	**Estratégia**
1	Frequência de endereços IP	Algoritmo probabilístico
2	Quantidade de endereços IP distintos	Algoritmo probabilístico
3	Quantidade de endereços IP distintos agrupados por país	Hibrida

Tabela 7.2: Cenários e estratégias de processamento

A frequência de endereços IP permite simular uma análise de quantas vezes um mesmo usuário acessou um item específico de uma loja virtual. A quantidade de endereços IP distintos possibilita a análise de quantos usuários únicos acionaram um determinado anúncio de uma página da web. O cenário de quantidade de endereços IP distintos por país permite a análise de quantos usuários únicos, em cada país, acionaram um determinado anúncio de uma pagina da web.

Nas seções a seguir, a arquitetura de software do experimento é descrita. Em seguida, são apresentados os resultados das análises dos DSPS utilizados nessa pesquisa. Após isso, é possível apresentar uma comparação entre abordagens de processamento com algoritmos convencionais e os algoritmos probabilísticos. Finalmente, é detalhado o experimento com uma abordagem híbrida para o processamento distribuído dos dados.

7.2 Arquitetura do experimento

A seguir, a arquitetura é descrita em visões arquiteturais conforme taxonomia e processo proposto por [54]. Essas visões explicitam, em diferentes perspectivas, os conceitos e decisões arquiteturais do experimento.

7.2.1 Visão geral

Essa visão arquitetural oferece uma perspectiva generalizada da arquitetura proposta. As camadas de responsabilidade, componentes e conceitos são organizados, abstraindo as características intrínsecas desses elementos. Essa organização, representada pela Figura 7.1, propõe o entendimento geral das partes identificadas e suas fronteiras.

A perspectiva generalizada dessa visão permite dividir os elementos desse sistema de processamento em: Modelo de processamento, Padrões de Projetos, Sistemas de processamento, *Middleware*

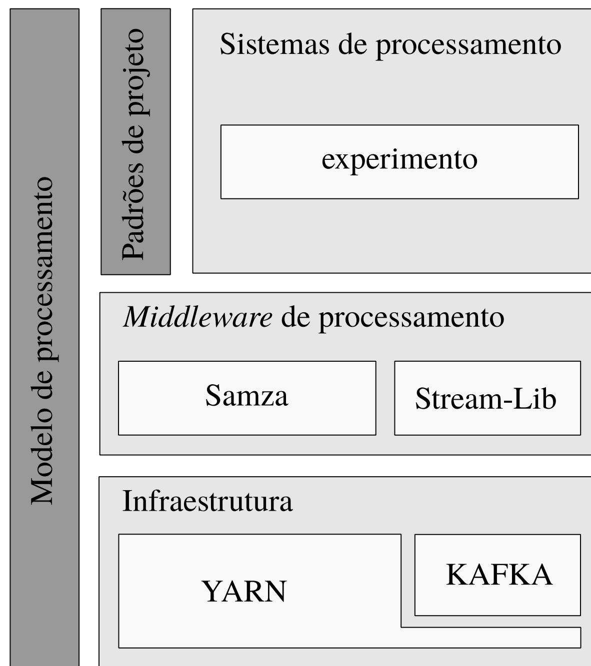

Figura 7.1: Visão geral da arquitetura do experimento

de processamento e Infraestrutura.

- Camada de Modelo de processamento

  A camada de Modelo de Processamento estabelece uma referência para a modelagem de fluxos de processamento distribuído de *streams* de dados. Essa camada propõe uma abstração independente de tecnologia, conforme apresentado em Capítulo 4, Figura 4.5, que disponibiliza um conjunto de elementos para a modelagem de fluxos de processamento coesos e reútilizáveis.

  Dessa forma, conforme a representado na Figura 7.2, a modelagem de fluxos de processamento é composta de (a)*Processing Elements* (PEs) e (b) *Streams* e armazenamentos persistentes (c). Esses elementos podem ser identificados no modelo de processamento dos DSPS estudados.

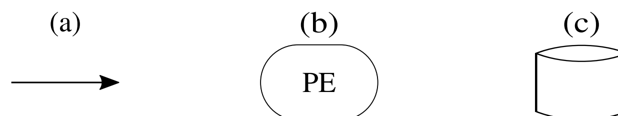

Figura 7.2: Elementos do modelo de processamento proposto

  - *Processing Element* (PE)

    O PE corresponde a uma unidade de processamento. Uma PE permite a organização do código em unidades menores, responsáveis por uma lógica específica. As PEs organizam o fluxo de processamento em unidades menores, o que apoia o reuso dessas unidades em diferentes fluxos, além de subsidiar o paralelismo do processamento de dados.

- *Stream*

  O *Stream* corresponde à estrutura de dados qpara o transporte das tuplas recebidas para processamento. Eles são utilizados como entrada e saída de dados no fluxo de processamento. O seu formato e estratégia de funcionamento são altamente acoplados com o DSPS utilizado.

- Armazenamento Persistente

  O armazenamento persistente refere-se à um SGBD externo, utilizado para registrar resultados de processamento, ou ainda, para realizar o gerenciamento de estado de processamentos complexos. Esse elemento de dados deve ser utilizado como saída de dados de um PE.

A utilização desse modelo viabiliza a definição de fluxos de processamento a partir da junção de diferentes PEs, interligados por *Streams* de dados. Dessa forma, um mesmo PE pode ser utilizado em diferentes fluxos, o que promove seu reuso em diferentes processamentos de dados.

Além disso, a modelagem de grandes fluxos de processamento é facilitada, uma vez que estes fluxos podem ser elaborados com a composição de diversos fluxos menores.

- Camada de Padrões de projeto

  Esta camada da arquitetura agrupa os padrões de projetos utilizados na engenharia dos sistemas de processamento. De acordo com a explanação do Capítulo 3, Seção 3.4, os padrões de projeto são soluções para problemas comuns, em contextos de características semelhantes.

  Na engenharia de sistemas, os padrões de projeto promovem o reuso de decisões bem sucedidas, redução da complexidade de *software* e auxilia na comunicação entre a equipe de projeto, uma vez que estabelece um vocabulário comum entre os integrantes de uma equipe de trabalho (Capítulo 3, Seção 3.4).

- Sistemas de processamento

  A camada de sistemas de processamento permite abstrair toda lógica necessária para o processamento distribuído de *stream* de dados. Ela contém apenas os componentes de *software* que implementam a lógica de processamento necessária para atingir os requisitos de negócio do sistema de análise de dados.

  Nesta pesquisa, o componente de software Experimento é parte dessa camada da arquitetura. Ele contém lógica de processamento de dados das análises realizadas neste trabalho.

- Camada de *Middleware* de processamento

  A camada de *Middleware* de processamento fornece uma abstração comum para o desenvolvimento de sistemas de processamento distribuído de dados. Essa camada protege a camada de Sistemas de processamento de complexidades relacionadas à heterogeneidade de tecnologias e detalhes de comunicação com a camada de infraestrutura. Dessa forma, a camada de Sistemas de processamento precisa conter somente a lógica do processamento dos dados.

No contexto desta pesquisa, encontramos nessa camada da arquitetura, os componentes de *software* da biblioteca de algoritmos probabílisticos stream-lib [82] e do DSPS Apache SAMZA [35].

- Camada de Infraestrutura

  A camada de infraestrutura disponibiliza à camada de *Middleware* o acesso a todos os recursos computacionais necessários para distribuição e execução dos sistemas de processamento de dados. Essa camada utiliza tecnologias de processamento distribuído com a finalidade de abstrair a lógica de acesso aos recursos computacionais instalados em redes privadas ou na nuvem.

  As principais atribuições da camada de infraestrutura são disponibilizar áreas de armazenamento no disco, acesso à memória volátil e capacidade de processamento necessário para o processamento dos dados.

7.2.2 Visão de fluxo de dados

Essa visão permite observar o fluxo de processamento dos dados das análises produzidas nesta pesquisa. Nessa visão, propõe-se um fluxo de dados que abstrai as particularidades dos DSPS estudados, representado pela Figura 7.3. Dessa forma, é possível modelar um único fluxo de processamento dos dados, compatível com todos os DSPS estudados.

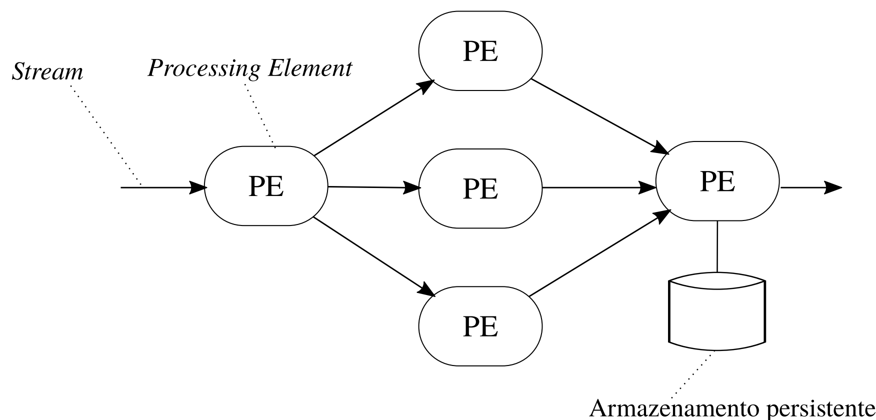

Figura 7.3: Fluxo de dados para o processamento de *streams* distribuídos

Esse fluxo é aderente ao modelo definido para a arquitetura na camada de Modelo de processamento. Ele apresenta uma composição de PEs que permite o processamento das três cenários utilizados na análise de dados desta pesquisa.

A execução das PEs ocorrem de forma isolada. O fluxo de execução do processamento é conduzido pelos *streams* de entrada de cada PE. Dessa forma, a execução do processamento é livre de operações que possam causar bloqueio de todo o fluxo de processamento de dados, de acordo com o Capítulo 5, seção 4.3, uma característica importante para o processamento distribuído de *stream* de dados.

Além disso, a execução isolada dos PEs permite a sua atualização em tempo de execução, sem impactar nos demais processamentos em andamento. Consequentemente, o processo de atualização, manutenção ou substituição desses PEs é facilitado significativamente.

7.2.3 Visão de distribuição

Nessa visão, é descrita como estão dispostos os elementos de *software*, necessários para a execução do sistema e como esses elementos são publicados e executados na infraestrutura computacional. Além disso, são explicitadas as dependências requeridas para a execução do sistema de processamento.

A Figura 7.4 utiliza um diagrama da UML para representar a disposição dos elementos de *software* do processamento distribuído do *stream*. Essa disposição ocorre em nós computacionais de uma infraestrutura distribuída que viabiliza o processamento de grandes volumes de dados em *near real-time*.

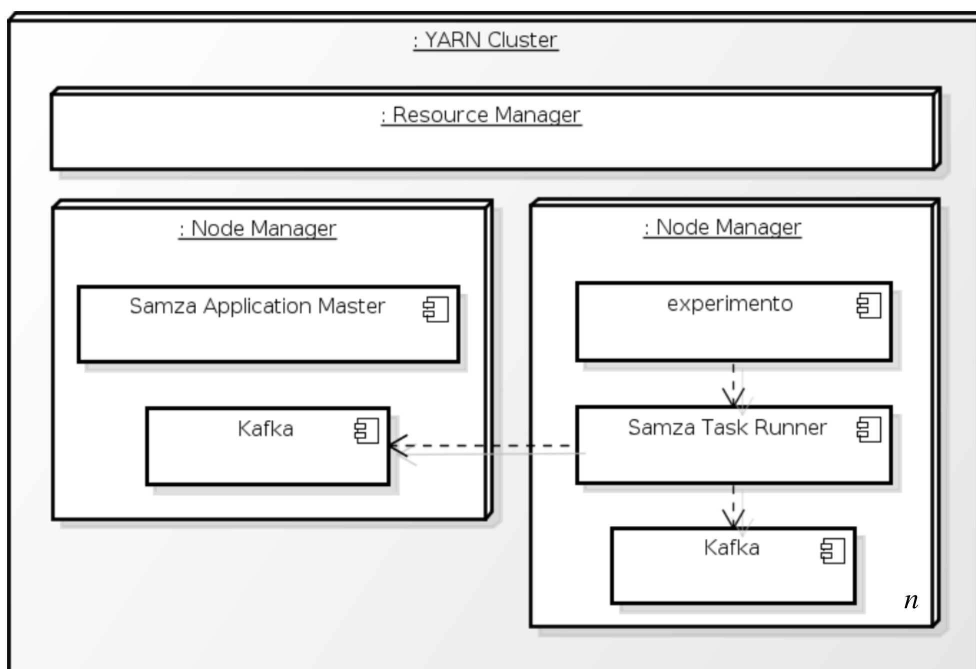

Figura 7.4: Diagrama da UML representando a distribuição dos elementos de *software streams* distribuídos

Nessa infraestrutura, um *cluster* de servidores com Apache YARN é utilizado para a publicação e execução da lógica de processamento de dados e dos componentes de *software* que compõem a camada de *Middleware* de processamento: Apache Samza e Stream-Lib.

A publicação dos componentes no *cluster* é coordenada pelo *Resource Manager* (RM). O RM é responsável por definir quais *Node manager* receberam os componentes do DSPS. Além disso, o *cluster* computacional pode ser composto, dependendo da quantidade de nós configurados, por n *Node manager*. Os componentes do DSPS coordenam, a partir dessa infraestrutura, a execução da lógica de processamento.

Além disso, o Apache KAFKA é publicado em todos os *Node Managers* do *cluster* YARN onde o DSPS está publicadoq. Dessa forma, o Apache KAFKA, que gerencia o *stream* de dados, oferece ao DSPS um acesso rápido à esses *streams*.

7.3 Análise dos DSPS

Nesta pesquisa foram estudados diferentes DSPS e sua aderência às principais características de um sistema para processamento de *stream* de dados. Essas características foram fundamentadas no Capítulo 4, Seção 4.3.

A Tabela 7.3 apresenta uma matriz com os DSPS estudados e as principais características esperadas nesses sistemas. Esses dados foram subsidios fundamentais para uma comparação entre os DSPS estudados.

Característica	**Storm**	**Borealis**	**Samza**
Mobilidade de dados	Sim	Sim	Sim
Consulta de dados	Sim	Sim	Sim
Armazenamento de dados	Não	Sim	Sim
Garantias de processamento e alta disponibilidade	Sim	Sim	Sim
Particionamento de dados	Sim	Sim	Sim
Tratamento de inconsistências no *stream* de dados	Implementável	Nativo	Implementável
Garantias de resultados consistentes e previsíveis	Configurável	Sim	Implementável
Baixo tempo de resposta	Sim	Sim	Sim

Tabela 7.3: Comparativo entre DSPS

Em relação à movimentação de dados, as implementações dos sistemas SAMZA e STORM possuem características semelhantes. Esses sistemas utilizam um modelo baseado em mensageria, geralmente utilizando sistemas como o ZeroMQ ou KAFKA para o gerenciamento dessas filas. Dessa forma, o apache SAMZA e STORM garantem uma movimentação dos dados livres de operações que possam bloquear o fluxo de processamento.

O sistema Borealis é suscetível a operações que podem bloquear o processamento. Ele implementa um modelo em que os dados são recebidos para processamento e redistribuídos entre os nós de seu *cluster* de servidores Borealis.

No armazenamento de dados, o Storm utiliza bancos de dados externos, através de comunicação remota, para o armazenado de informações. Essa estratégia inclui o custo da comunicação remota com o banco de dados no tempo de processamento de dados. O SAMZA e o Borealis apresentam uma abordagem otimizada, livres de chamadas remotas para o armazenamento de dados. O SAMZA utiliza a própria infraestrutura de *containers* do YARN para disponibilizar ao seu sistema um banco de dados NoSQL para armazenamento dos dados. O Borealis permite o armazenamento de dados na estrutura de dados de cada nó de processamento.

Apesar de apresentar detalhes de implementação, arquiteturas e métodos de distribuição distintos, os sistemas estudados apresentam diversas semelhanças em suas funcionalidades. Um exemplo claro é o modelo de processamento que diverge em nomenclatura, mas é conceitualmente compatível.

O Apache SAMZA foi o DSPS utilizado nos experimentos desta pesquisa. Ele é um DSPS *open source* aderente aos requisitos recomendados para o processamento de *stream* de dados explanados no Capítulo 4. O modelo de processamento do SAMZA é altamente integrado com o Apache KAFKA, que fornece uma infraestrutura de mensageria, com garantias de entrega e alta disponibilidade [34].

Além disso, a arquitetura e modelo de processamento do SAMZA se integram com o Apache YARN, que possibilita a execução de código arbitrário em diferentes *containers* em um *cluster* de computadores, com tolerância a falhas, registro de operações, isolamento de recursos e segurança [35].

7.4 Análise de algorítimos para o processamento de *stream* de dados

Nesta seção, são apresentados os resultados do experimento utilizado no processamento dos dados para a análise dos cenários propostos. Esses resultados explicitam o consumo de memória para cada estratégia sugerida para a pesquisa.

Primeiramente, para calcular o cenário de frequência de um endereço IP específico, foi utilizado a implementação Java da coleção HashMap [96].Essa implementação permite registrar itens em uma coleção, no formato de <Chave, Valor>. Dessa forma, sempre que um novo endereço de IP é processado, é incluído como chave no mapa. O valor desse IP é um contador que é incrementado para cada nova ocorrência do mesmo endereço IP.

A Tabela 7.4 compara o algoritmo Count-min sketch e a implementação HashMap da linguagem Java, quanto ao consumos de memória e taxa de erro, considerando a quantidade de ocorrências do elemento mais frequente no conjunto de dados analisado.

Algoritmo	Consumo de memória (kB)	Frequência
Count-Min Sketch	413.890	14
HashMap	1.726.191	13

Tabela 7.4: Comparativo entre algoritmos para o processamento do cenário de frequência de um endereço IP

O algoritmo Count-Min Sketch consumiu 413.890kB de memória para processar toda massa de dados do experimento. Esse valor representa 23% da memória requerida pela implementação Java da coleção HashMap, que exigiu 1.726.191kB para processar o mesmo conjunto de dados. Além

disso, o algoritmo HashMap sumarizou 13 ocorrências do elemento mais frequente, enquanto Count-Min Sketch estimou 14 ocorrências no conjunto de dados analisados. Dessa forma, o algoritmo probabílistico Count-Min Sketch apresentou uma taxa de erro de 7,69% em relação a contagem do algoritimo HashMap. Esse resultado confirma a expectativa de redução no consumo de recursos computacionais em detrimento da precisão do processamento explanados no Capítulo 6, Seção 4.4.

No segundo cenário, para calcular a quantidade de endereços IP distintos encontrados durante o processamento, utilizamos o HashSet [97]. O HashSet é uma coleção implementada em linguagem Java que garante unicidade dos elementos registrados em sua estrutura. A estratégia do HashSet é comparada com o algoritmo probabilístico HyperLogLog, que permite a estimar a quantidade de elementos distintos.

Na Tabela 7.5, comparamos o consumo de memória e os valores obtidos com as estratégias de processamento propostas para o cenário de quantidade de endereços IPs distintos.

Algoritmo	Consumo de memória (kB)	IPs distintos
HyperLogLog	57.237	14.632.542
HashSet	1.681.680	14.945.210

Tabela 7.5: Comparativo entre algoritmos para o processamento do cenário de quantidade de endereços IPs distintos analisados

Nesse experimento, o algoritmo HyperLogLog consumiu 57.237 kB de memória para processar toda a massa de dados do experimento. Dessa forma, ele consumiu 0,03% da memória necessária para a implementação Java da coleção HashSet, que exigiu 1.681.680 kB para processar o mesmo conjunto de dados. Além disso, o algoritmo HashSet contou 14.945.210 elementos distintos e o algoritmo HyperLogLog estimou 14.632.542 elementos distintos no conjunto de dados analisados. Dessa forma, o algoritmo probabílistico HyperLogLog apresentou uma taxa de erro de erro de 2,09% em relação a contagem do algoritimo HashSet. Esse resultado confirma a expectativa de redução no consumo de recursos computacionais em detrimento a precisão do processamento explanados no Capítulo 6, Seção 4.4).

A Figura 7.5 compara os resultados evidenciados durante o processamento dos cenários. O valor analisado de 1.2 GB (1.258.291 kB) corresponde ao processamento de toda a massa de dados produzida para o experimento.

O processamento dos cenários Frequência de endereços IP (Indicador 1) e Quantidade de endereços IP distintos (Indicador 2) apresentaram uma redução de até 97% no consumo de memória. O uso de algoritmos probabilísticos representou uma economia no consumo de memória. Essa estratégia pode ser aplicada em processamento de dados distribuídos, propiciando maior escalabilidade para processamentos de grandes volumes de dados.

Além disso, a capacidade de estimar o valor de conjuntos infinitos de dados, utilizando somente uma parte desses conjuntos é um recurso capaz de viabilizar análises complexas de informações em *near real-time*.

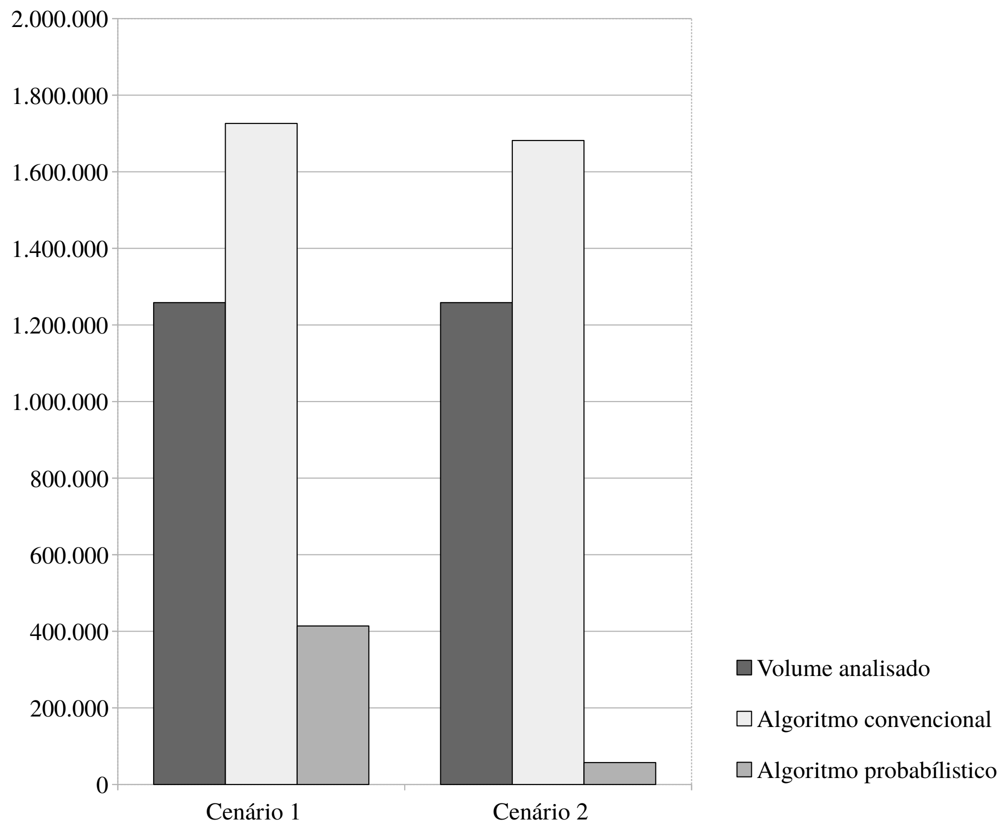

Figura 7.5: Análise do consumo de memória de cada estratégia para o processamento dos dados

A seguir, são detalhados os resultados para o processamento do terceiro cenário com uma estratégia híbrida com algoritmos convencionais e probabilísticos.

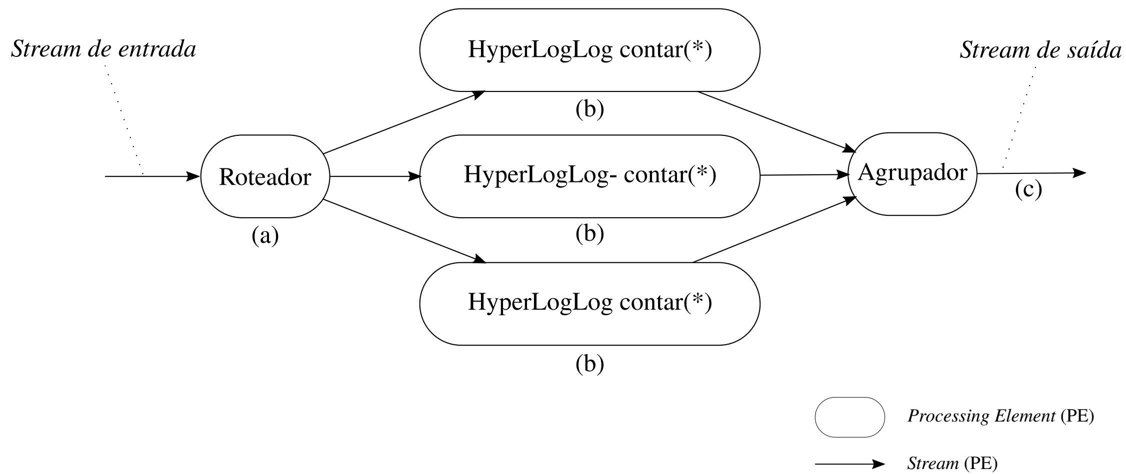

Figura 7.6: Padrão de projeto proposto para agrupamento e contagem de elementos em *stream* de dados

7.5 Padrões de projeto identificados

Durante a avaliação de possíveis estratégias para o processamento do terceiro cenário, quantidade de endereços IP distintos agrupados por país, foi identificada a oportunidade de utilizar uma estratégia híbrida. Nessa estratégia, representada pela Figura 7.6, modelamos um PE (a) responsável por filtrar os elementos de acordo com o país de origem. Essa PE tem a função de rotear os elementos recebidos para contagem em um próximo estágio. Nesse momento, um grupo de PEs (b) estima a quantidade de elementos distintos existentes em cada grupo utilizando um algoritmo probabilístico, nesse caso, o algoritmo HyperLogLog. Finalmente, os resultados do processamento são consolidados em um *Stream* de saída (c), ficando disponível para em uma base centralizada para análise do usuário final.

A característica de agrupamento de dados por um atributo específico, necessário para esse terceiro cenário, foi fundamental para o estabelecimento dessa estratégia de processamento. Dessa forma, foi possível delimitar diversos subconjuntos de dados com tamanho indefinido, onde o algoritmo probabilístico HyperLogLog é capaz de realizar a estimativa de quantidade de elementos distintos.

Esse experimento permitiu catalogar um padrão de projeto descrito na Tabela 7.6. De acordo com a explanação do Capítulo 3, Seção 3.4, um padrão de projeto é capaz de solucionar problemas de características semelhantes em outros sistemas de processamento distribuído de dados, promovendo o reuso de decisões de projeto bem sucedidas.

Nome	*Group and count*
Problema	Contar a quantidade de elementos distintos agrupados por um atributo específico
Fluxo	*Stream de entrada* Roteador contar(*) contar(*) contar(*) Agrupador *Stream de saída* *Processing Element* (PE) *Stream* (PE)
Solução	1. Consumir o registro a partir do *stream* de entrada. 2. Testar o atributo agrupador para identificar o *stream* de destino do registro. 3. Submeter o resultado para o *stream* de dados adequado. 4. Aplicar no subconjunto de processamento o algoritmo escolhido para a contagem de elementos distintos. 5. Consolidar os resultados em um *stream* de saída.

Tabela 7.6: Descrição do padrão de projeto *Group and Count*

Capítulo 8

Conclusões

O processamento de grandes volumes de dados para o cálculo de indicadores demandam poderosas soluções de armazenamento e processamento de dados como o Apache Hadoop e MapReduce [9]. Essas soluções produzem análises em uma abordagem de alta latência, onde são necessárias horas ou até mesmo dias para o cálculo dos indicadores desejados.

O crescimento no volume de dados produzidos por aplicativos na web utilizados por milhares de usuários, por uma infinidade de sensores que produzem dados em diversos formatos, por diversos dispositivos ligados à Internet, além de registros de segurança e auditoria em sistemas de informação em diversas outras origens de dados fazem dos recursos computacionais um tema crítico para o processamento desses dados.

O desenvolvimento de soluções analíticas em tempo *near real-time* envolvendo o processamento de *Big Data*, mantidos em uma infraestrutura distribuída, requer a construção de componentes de software em uma solução complexa. Estabelecer uma arquitetura de referência, com definições de camadas e suas responsabilidades, promovendo o reuso de decisões de design bem sucedidas, resultam em uma solução econômica e manutenível.

Este trabalho propõe uma arquitetura de referência inovadora para o processamento distribuído de *stream* de dados em larga escala que suporta todas as oito principais características descritas para o processamento de dados em *stream* recomendados em trabalhos recentes [30] [98], além de permitir a construção de componentes coesos e reutilizáveis através dos *Processing Elements*.

A arquitetura de referência proposta pelo trabalho também inclui recursos que permitem a escalabilidade vertical da solução, isolamento de processos, baixo acoplamento dos componentes, reuso de código e melhor uso de recursos computacionais. Em complemento, identificamos padrões de projetos para a construção de fluxos de transformação de dados, que podem compor um catálogo que agrupa soluções padrões para problemas com características semelhantes.

Para conduzir a solução do problema abordado neste trabalho, fez-se necessário desenvolver os conceitos e as aplicações de uma arquitetura de software, técnicas de processamento e armazenamento de dados distribuído, arquitetura em camadas, técnicas comunicação através de mensageria, estimativa de valores com algoritmos probabilísticos, além de técnicas BI em *near real-time*. Es-

sas tecnologias são elementos que permitem incrementar recursos à arquitetura de referência, de maneira que solucionamos o problema abordado por este trabalho, com a proposição de uma abordagem para o desenvolvimento de soluções analíticas em *near real-time*.

Antes de apresentar os resultados obtidos, detalhamos a arquitetura proposta, com as técnicas e melhores práticas arquiteturais, descrevendo em diferentes níveis e visões, os elementos que compõem essa arquitetura de referência, de forma que o leitor pôde se familiarizar com as técnicas aplicadas no estudo de caso, nos fluxos de processamento de dados distribuído, na modelagem dos dados e na construção de indicadores em tempo real.

Todo o processo, que se inicia na origem do evento produtor dos dados até a apresentação dos resultados dos eventos produzidos em janelas curtas de tempo em uma solução analítica, é explanado em uma notação adequada, que permite o entendimento adequado do ciclo de vida da informação.

Em seguida, conseguimos aplicar a arquitetura de referência em uma arquitetura instanciada que forneceu as decisões de projeto necessárias para o estudo de caso, tal estudo permitiu verificar os resultados experimentais, com o processamento de diversos indicadores de BI em *near real-time*. Dessa forma, consideramos o estudo de caso, também uma prova de conceito arquitetural, que fornece, do ponto de vista da metodologia, todos os recursos necessários para a validação da arquitetura de referência proposta por este trabalho.

Com o intuito de agregar ao trabalho recomendações e práticas para a economia de recursos computacionais, aplicamos algorítimos probabilísticos para a estimativa de valores em subconjuntos de dados no *stream* de dados, utilizando o processamento em janelas de tempo. Foi possível verificar no estudo de caso que os algoritmos probabilísticos incluem uma pequena imprecisão no resultado final do indicador. Isso inviabiliza aplicar essa técnica em indicadores com informações críticas e que requerem precisão. Entretanto, conseguimos verificar que os algorítimos podem ser otimizados, o que reduz drasticamente a imprecisão, ampliando as possibilidades de aplicação de algoritmos probabilísticos na estimativa de valores de indicadores no processamento distribuídos dos *stream* de dados.

Este trabalho proporciona a base para diversos outros que podem envolver aplicações práticas da arquitetura de referência, como também, aperfeiçoamentos em várias visões arquiteturais e técnicas aplicadas no estudo de caso.

O metamodelo para a modelagem de fluxos de processamento proposto pode ser utilizado como referência para o desenvolvimento de um padrão para modelagem de fluxos de processamento. Além disso, esse metamodelo define a estrutura básica para a o desenvolvimento de uma interface gráfica para a modelagem de fluxos de processamento.

No campo de algoritmos probabilísticos e mineração de dados, o uso de registro histórico dos processamentos pode ser útil para otimizar a configuração entre o uso de recursos computacionais e a precisão dos resultados estimados por algoritmos para a estimativa de valores.

Em relação a arquitetura e padrões de projeto, o refinamento dessa arquitetura de referência a partir de experiências práticas de sua instanciação em arquiteturas de sistemas para o pro-

cessamento distribuído de *stream* de dados pode subsidiar ajustes e melhorias na arquitetura de referência. Além disso, o monitoramento das decisões arquiteturais baseadas nessa arquitetura de referência, além da identificação de novos padrões, podem resultar em um catálogo de padrões de projetos e proposição de modelos de referência para soluções de processamento de *Big Data* em *near real-time*.

A aplicação dessa arquitetura pode reduzir a complexidade de desenvolvimento, promover o reuso e aumentar a manutenibilidade de processamentos de dados aplicados em soluções de aquisição e análise pervasiva em *near real-time* de dados difusos, advindos de diferentes dispositivos ligados à Internet (IoT) em ambientes de dados heterogêneos, como sistemas aplicados à Cidades Inteligentes (SmartCities).

REFERÊNCIAS BIBLIOGRÁFICAS

[1] GANTZ, J.; REINSEL, D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2012.

[2] JACOBI, J. A.; BENSON, E. A.; LINDEN, G. D. *Computer processes for identifying related items and generating personalized item recommendations.* [S.l.]: Google Patents, maio 17 2011. US Patent 7,945,475.

[3] MCAFEE, A. et al. Big data. *The management revolution. Harvard Bus Rev*, v. 90, n. 10, p. 61–67, 2012.

[4] HONG, Y. et al. Flood and landslide applications of near real-time satellite rainfall products. *Natural hazards*, Springer, v. 43, n. 2, p. 285–294, 2007.

[5] GANTZ, J.; REINSEL, D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2012.

[6] SIMMERS, C. A. A stakeholder model of business intelligence. In: *Business Intelligence Techniques.* [S.l.]: Springer, 2004. p. 227–242.

[7] KIMBALL, R.; ROSS, M. *The data warehouse toolkit: the complete guide to dimensional modeling.* [S.l.]: John Wiley & Sons, 2011.

[8] DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, ACM, v. 51, n. 1, p. 107–113, 2008.

[9] APACHE. *Apache Hadoop.* 2014. Http://hadoop.apache.org/docs/current/. 24/06/2014.

[10] BIRRELL, A. D.; NELSON, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 2, n. 1, p. 39–59, 1984.

[11] Kimball, Ralph; Caserta, Joe. *The data warehouse ETL toolkit : practical techniques for extracting, cleaning, conforming, and delivering data.* [S.l.]: John Wiley & Sons, 2004.

[12] RÄDKE, S. *Analysis of Data Warehouse Tools IBM Websphere DataStage and Informatica PowerCenter with Recommendation for Selected Data Acquisition Scenarios.* 2007.

[13] INMON, W. H. Building the data warehouse. John wiley & sons, 2005.

[14] WITTEN, I. H.; FRANK, E. *Data Mining: Practical machine learning tools and techniques*. [S.l.]: Morgan Kaufmann, 2005.

[15] KNUTH, D. E. *Art of Computer Programming, Volume 2: Seminumerical Algorithms, The*. [S.l.]: Addison-Wesley Professional, 2014.

[16] GABER, M. M. *Scientific data mining and knowledge discovery: Principles and foundations*. [S.l.]: Springer, 2009.

[17] HALL, M. et al. The weka data mining software: An update; sigkdd explorations, volume 11, issue 1 (2009) 11. *Affendey, LS, Paris, IHM, Mustapha, N., Sulaiman, MN, Muda, Z.: Ranking of Influencing Factors in Predicting Student Academic Performance. Information Technology Journal*, v. 9, n. 4, p. 832–837, 2010.

[18] GOLAB, L.; ÖZSU, M. T. Issues in data stream management. *ACM Sigmod Record*, ACM, v. 32, n. 2, p. 5–14, 2003.

[19] ELLIS, B. *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data*. [S.l.]: John Wiley & Sons, 2014.

[20] JIANG, N.; GRUENWALD, L. Research issues in data stream association rule mining. *ACM Sigmod Record*, v. 35, n. 1, p. 14–19, 2006. Disponível em: <http://dl.acm.org/citation.cfm?id=1121998>.

[21] BABCOCK, B. et al. Models and issues in data stream systems. In: ACM. *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. [S.l.], 2002. p. 1–16.

[22] REAL, L. C. V. *Uma arquitetura para análise de fluxo de dados estruturados aplicada ao sistema brasileiro de TV digital*. Tese (Doutorado) — Universidade de São Paulo, 2009.

[23] GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The google file system. In: ACM. *ACM SIGOPS Operating Systems Review*. [S.l.], 2003. v. 37, n. 5, p. 29–43.

[24] SADALAGE, P. J.; FOWLER, M. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. [S.l.]: Pearson Education, 2012.

[25] CHANG, F. et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 26, n. 2, p. 4, 2008.

[26] APACHE. Hbase - a distributed database for large datasets. *The Apache Software Foundation, Los Angeles, CA. URL http://hbase. apache. org*, 2014.

[27] R. C. Huacarpuma et al. Artificial intelligence technologies and the evolution of web 3.0. In: _____. [S.l.]: IGI Global, 2014. cap. EVALUATING NoSQL DATABASES FOR Big Data PROCESSING WITHIN THE BRAZILIAN MINISTRY OF PLANNING, BUDGET AND MANAGEMENT.

[28] CATTELL, R. Scalable sql and nosql data stores. *ACM SIGMOD Record*, ACM, v. 39, n. 4, p. 12–27, 2011.

[29] MARZ, N.; WARREN, J. *Big Data: Principles and best practices of scalable realtime data systems*. [S.l.]: O'Reilly Media, 2013.

[30] STONEBRAKER, M.; ÇETINTEMEL, U.; ZDONIK, S. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, ACM, v. 34, n. 4, p. 42–47, 2005.

[31] ABADI, D. J. et al. The design of the borealis stream processing engine. In: *CIDR*. [S.l.: s.n.], 2005. v. 5, p. 277–289.

[32] ABADI, D. J. et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal—The International Journal on Very Large Data Bases*, Springer-Verlag New York, Inc., v. 12, n. 2, p. 120–139, 2003.

[33] TOSHNIWAL, A. et al. Storm@ twitter. In: ACM. *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. [S.l.], 2014. p. 147–156.

[34] APACHE. *Apache Kafka*. 2014. Https://kafka.apache.org/. 24/06/2014.

[35] APACHE. *Apache Samza*. 2014. Http://samza.incubator.apache.org/. 24/06/2014.

[36] KAMBURUGAMUVE, S. et al. *Survey of Streaming Data Algorithms*. [S.l.], 2013.

[37] CORMODE, G.; MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, Elsevier, v. 55, n. 1, p. 58–75, 2005.

[38] Philippe Flajolet et al. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Conference on Analysis of Algorithms*, p. 127–146, 2007.

[39] CLIFFORD, I. A. C. P. A statistical analysis of probabilistic counting algorithms. 2010.

[40] MORALES, G. D. F. Samoa: A platform for mining big data streams. In: INTERNATIONAL WORLD WIDE WEB CONFERENCES STEERING COMMITTEE. *Proceedings of the 22nd international conference on World Wide Web companion*. [S.l.], 2013. p. 777–778.

[41] TYREE, J.; AKERMAN, A. Architecture decisions: Demystifying architecture. *IEEE software*, v. 22, n. 2, p. 19–27, 2005.

[42] BOURQUE, P.; FAIRLEY, R. E.; IEEE Computer Society. *Swebok: guide to the software engineering body of knowledge*. [Los Alamitos, CA]: IEEE Computer Society, 2014. ISBN 9780769551661 0769551661.

[43] HU, H. et al. Toward scalable systems for big data analytics: A technology tutorial. *Access, IEEE*, v. 2, p. 652–687, 2014. ISSN 2169-3536.

[44] GOVERNOR, J.; HINCHCLIFFE, D.; NICKULL, D. *Web 2.0 Architectures: What entrepreneurs and information architects need to know*. [S.l.]: "O'Reilly Media, Inc.", 2009.

[45] CLOUTIER, R. et al. The concept of reference architectures. *Systems Engineering*, Wiley Online Library, v. 13, n. 1, p. 14–27, 2010.

[46] MULLER, G. A reference architecture primer. *Eindhoven Univ. of Techn., Eindhoven, White paper*, 2008.

[47] The Open Group. *TOGAF 9 - The Open Group Architecture Framework Version 9*. USA: The Open Group, 2009. 744 p.

[48] HP. *HP Reference Architecture for MapR M5*. [S.l.], 2013.

[49] BOJA, C. Distributed parallel architecture for"big data. *Informatica Economica*, INFOREC Association, v. 16, n. 2, p. 116–127, 2012.

[50] ISO. International standard iso/iec 24765:2010 systems and software engineering — vocabulary. *ISO/IEC/IEEE*, 2010.

[51] ISO. International standard iso/iec 12207 software life cycle processes. *Software Process Improvement and Practice*, 1996.

[52] SEI. *SEI Glossary*. 2014. Http://www.sei.cmu.edu/architecture/start/glossary/. 24/10/2014.

[53] BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software architecture in practice*. Boston: Addison-Wesley, 2003. ISBN 0321154959 9780321154958.

[54] HILLIARD, R. Ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems. *IEEE, http://standards. ieee. org*, v. 12, p. 16–20, 2000.

[55] GARLAN, D. et al. *Documenting Software Architectures: Views and Beyond*. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2010. ISBN 0321552687, 9780321552686.

[56] ROZANSKI, N.; WOODS, E. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Upper Saddle River, NJ: Addison-Wesley, 2005. ISBN 0321112296 9780321112293.

[57] NEWELL, A.; SIMON, H. A. et al. *Human problem solving*. [S.l.]: Prentice-Hall Englewood Cliffs, NJ, 1972.

[58] ALEXANDER, C. *The timeless way of building*. [S.l.]: Oxford University Press, 1979.

[59] GAMMA, E. et al. *Design patterns: Abstraction and reuse of object-oriented design*. [S.l.]: Springer, 1993.

[60] BUSCHMANN, F. et al. *Pattern-oriented software architecture, volume 1: A system of patterns*. [S.l.]: John Wiley and Sons, 1996.

[61] LANDAY, J. A.; BORRIELLO, G. Design patterns for ubiquitous computing. *Computer*, IEEE, v. 36, n. 8, p. 93–95, 2003.

[62] REED, P. Reference architecture: The best of best practices. *The Rational Edge*, 2002.

[63] BORKAR, V. R.; CAREY, M. J.; LI, C. Big data platforms: what's next? *XRDS: Crossroads, The ACM Magazine for Students*, ACM, v. 19, n. 1, p. 44–49, 2012.

[64] HSIAO, D. K. *Advanced database machine architecture.* [S.l.]: Prentice Hall Professional Technical Reference, 1983.

[65] DEWITT, D. J. et al. *A High Performance Dataflow Database Machine.* [S.l.]: Computer Science Department, University of Wisconsin, 1986.

[66] DEWITT, D.; GRAY, J. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, ACM, v. 35, n. 6, p. 85–98, 1992.

[67] GANTZ, J.; REINSEL, D. Extracting value from chaos. *IDC iview*, p. 1–12, 2011.

[68] DIJCKS, J. P. Oracle: Big data for the enterprise. *Oracle White Paper*, 2012.

[69] CHAUDHURI, S. What next?: a half-dozen data management research goals for big data and the cloud. In: ACM. *Proceedings of the 31st symposium on Principles of Database Systems.* [S.l.], 2012. p. 1–4.

[70] VARIA, J.; MATHEW, S. Overview of amazon web services. *http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-intro/intro. html*, 2013.

[71] DUMBILL, E. *Planning for Big Data.* [S.l.]: "O'Reilly Media, Inc.", 2012.

[72] RUSSOM, P. et al. Big data analytics. *TDWI Best Practices Report, Fourth Quarter*, 2011.

[73] RAMAKRISHNAN, R. Cap and cloud data management. *Computer*, Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, 17 th Fl New York NY 10016-5997 United States, v. 45, n. 2, p. 43–49, 2012.

[74] PRITCHETT, D. Base: An acid alternative. *Queue*, ACM, v. 6, n. 3, p. 48–55, 2008.

[75] PADHY, R. P.; PATRA, M. R.; SATAPATHY, S. C. Rdbms to nosql: reviewing some next-generation non-relational databases. *International Journal of Advanced Engineering Science and Technologies*, v. 11, n. 1, p. 15–30, 2011.

[76] CHARDONNENS, T. et al. Big data analytics on high velocity streams: A case study. In: *Big Data, 2013 IEEE International Conference on.* [S.l.: s.n.], 2013. p. 784–787.

[77] BOCKERMANN, C. A survey of the stream processing landscape. 2014.

[78] HEULE, S.; NUNKESSER, M.; HALL, A. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In: *Proceedings of the 16th International Conference on Extending Database Technology.* ACM, 2013. p. 683–692. Disponível em: <http://dl.acm.org/citation.cfm?id=2452456>.

[79] NEUMEYER, L. et al. S4: Distributed stream computing platform. In: IEEE. *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on.* [S.l.], 2010.

[80] ZHANG, Z. et al. A hybrid approach to high availability in stream processing systems. In: IEEE. *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*. [S.l.], 2010. p. 138–148.

[81] AMINI, L. et al. Spc: A distributed, scalable platform for data mining. In: ACM. *Proceedings of the 4th international workshop on Data mining standards, services and platforms*. [S.l.], 2006. p. 27–37.

[82] STREAM-LIB. *Stream-lib*. 2014. Https://github.com/addthis/stream-lib. 26/09/2014.

[83] Philippe Flajolet; Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 1985.

[84] Marianne Durand; Philippe Flajolet. Loglog counting of large cardinalities. *European Symposium on Algorithms*, 2003.

[85] LUHN, H. P. A business intelligence system. *IBM Journal of Research and Development*, IBM, v. 2, n. 4, p. 314–319, 1958.

[86] SILVERS, F. *Building and maintaining a data warehouse*. [S.l.]: CRC Press, 2008.

[87] SALLAM, R. L. et al. Magic quadrant for business intelligence platforms. *Gartner Group, Stamford, CT*, 2011.

[88] DAVENPORT, T. H. Competing on analytics. *harvard business review*, v. 84, n. 1, p. 98, 2006.

[89] CHEN, H.; CHIANG, R. H.; STOREY, V. C. Business intelligence and analytics: From big data to big impact. *MIS quarterly*, v. 36, n. 4, p. 1165–1188, 2012.

[90] CHAUDHURI, S.; DAYAL, U.; NARASAYYA, V. An overview of business intelligence technology. *Commun. ACM*, ACM, New York, NY, USA, v. 54, n. 8, p. 88–98, ago. 2011. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1978542.1978562>.

[91] BALLARD, C. et al. *Dimensional Modeling: In a Business Intelligence Environment*. [S.l.]: IBM Redbooks, 2012.

[92] CHAUDHURI, S.; DAYAL, U. An overview of data warehousing and olap technology. *ACM Sigmod record*, ACM, v. 26, n. 1, p. 65–74, 1997.

[93] NIMBUS. *Nimbus Project*. 2014. Http://www.nimbusproject.org/. 20/10/2014.

[94] HUNT, P. et al. Zookeeper: Wait-free coordination for internet-scale systems. In: *USENIX Annual Technical Conference*. [S.l.: s.n.], 2010. v. 8, p. 9.

[95] Cormode, G.; Muthukrishnan, S. Summarizing and mining skewed data streams. In: . [S.l.]: SIAM, 2005.

[96] ORACLE. *The Java HashMap Collection*. 2014. Http://docs.oracle.com/javase/7/docs/api/java/util/HashM
17/09/2014.

[97] ORACLE. *The Java HashSet Collection*. 2014. Http://docs.oracle.com/javase/7/docs/api/java/util/HashSet
24/06/2014.

[98] KAMBURUGAMUVE, S. et al. *Survey of Distributed Stream Processing for Large Stream Sources*. [S.l.], 2013.

TRABALHOS PUBLICADOS PELO AUTOR

D. C. RODRIGUES, R. CHAVES, R. C. HUACARPUMA, L. LIMA, A. M. R. SERRANO, M. T. DE HOLANDA, J. P. C. L. DA COSTA, E. P. DE FREITAS, R. T. DE SOUSA JR., “A reference architecture to distributed processing streams of data for near real-time analytics,”, IEEE/WIC/ACM International, 2015

R. C. HUACARPUMA, D. C. RODRIGUES, A. M. R. SERRANO, J. P. C. L. DA COSTA, R. T. DE SOUSA JR., LEITE, L., RIBEIRO, E., M. T. DE HOLANDA, E ARAUJO, A. P. F, “Evaluating NoSQL databases for Big Data processing within the Brazilian Ministry Of Planning, Budget And Management,” Artificial Intelligence Technologies and the Evolution of Web 3.0, IGI Global, 2015

R. C. HUACARPUMA, D. C. RODRIGUES, A. M. R. SERRANO, J. P. C. L. DA COSTA, R. T. DE SOUSA JR., E M. T. DE HOLANDA, "Big Data: A Study case on data from Brazilian Ministry Of Planning, Budgeting And Management,"IADIS Applied Computing 2013 (AC 2013) Conference, Oct. 2013, Fort Worth, Texas, USA

S. R. CAMPOS, A. A. FERNANDES, R. T. DE SOUSA JR., E. P. DE FREITAS, J. P. C. L. DA COSTA, A. M. R. SERRANO, D. C. RODRIGUES, E C. T. RODRIGUES, "ONTOLOGIC AUDIT TRAILS MAPPING FOR DETECTION OF IRREGULARITIES IN PAYROLLS,"International Conference on Next Generation Web Services Practices (NWeSP 2012), São Carlos, Brazil,Nov. 2012

ANEXOS

I. CÓDIGOS-FONTE PRODUZIDOS

I.1 Produtor de dados

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayDeque;
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class GeradorMassa extends Thread {

   private final static ArrayDeque<GeradorMassaEvent> stream = new
       ArrayDeque<GeradorMassaEvent>();

   private final Set<GeradorMassaListener> listeners;
   private final static String countries[] = { "BR", "US", "ES", "MX", "JP", "AR",
       "IT", "SE" };

   private final static ExecutorService poolThread = Executors.newFixedThreadPool(2);;

   public static long start = 0;

   public GeradorMassa() {
      this.listeners = new HashSet<GeradorMassaListener>();
   }

   @Override
   public void run() {
      while (true) {
            String ip = 192 + "."
                   + (Math.round(Math.random() * 254)) + "."
                   + (Math.round(Math.random() * 254)) + "."
                   + (Math.round(Math.random() * 254));

         GeradorMassaEvent evt = new GeradorMassaEvent(
               System.currentTimeMillis(), ip,
               countries[Integer.parseInt(String.valueOf(Math.round((Math
                     .random() * (countries.length - 1)))))]);

         for (GeradorMassaListener listener : listeners) {
            listener.onEvent(evt);
```

```
        }
    }
}

public void listen(GeradorMassaListener listener) {
    this.listeners.add(listener);
}

public void unlisten(GeradorMassaListener listener) {
    listeners.remove(listener);
}

public static interface GeradorMassaListener {
    void onEvent(GeradorMassaEvent event);
}

public static void main(String[] args) throws InterruptedException,
        IOException {

    GeradorMassa feed = new GeradorMassa();
    feed.listen(new GeradorMassaListener() {
        FileWriter writer = new FileWriter("/home/daniel/template.csv");

        public void onEvent(GeradorMassaEvent event) {
            try {
                if ((((System.currentTimeMillis() - start) / 1000) < 110)) {
                    writer.append(String.valueOf(event.getTime()) + ",");
                    writer.append(String.valueOf(event.getCountry()) + ",");
                    writer.append(String.valueOf(event.getUser()) + "\n");

                    writer.flush();
                } else {
                    System.out.println((((System.currentTimeMillis() - start) / 1000) < 5));
                    writer.close();
                    System.out.println(stream.size() + " itens in "
                            + ((System.currentTimeMillis() - start) / 1000)
                            + "s");
                    System.exit(0);
                }
            } catch (IOException e) {
                e.printStackTrace();
                System.exit(0);
            }

        }
    });
```

```
      start = System.currentTimeMillis();
      poolThread.execute(feed);
   }
}
```

I.2 Algoritmo convencional para contagem de elementos distintos

```
public void contarDistintosSet() throws IOException {
   logger.info("==============Simple Distinct===========");
   logMemory();
   long start = System.currentTimeMillis();
   LineIterator it = FileUtils.lineIterator(new File(FILE_PATH), "UTF-8");
   Set<String> counter = new HashSet<String>();

   long itSize = 0;
   try {
      while (it.hasNext()) {
         String line = it.nextLine();
         counter.add(line.split("[,]")[2]);
         itSize++;
      }

   logger.info(search + " ("+counter.keySet().size()+")= " + counter.get(search));

   } finally {
      LineIterator.closeQuietly(it);
   }
   callGc();
   logger.info("("+itSize+")= " + counter.size());
   long end = System.currentTimeMillis();
   logger.info("Processing time: " + (end - start));
   logMemory();
   logger.info("==============Simple Distinct===========");

}
```

I.3 Algoritmo HyperLogLog para a estimativa de elementos distintos

```
public void testHll() throws IOException {

   logger.info("==============HLL===========");
```

```
    logMemory();
   long start = System.currentTimeMillis();
   LineIterator it = FileUtils.lineIterator(new File(FILE_PATH), "UTF-8");

   logger.info(""+RegisterSet.REGISTER_SIZE);
   HyperLogLog hyperLogLog = new HyperLogLog(24);

   long itSize = 0;

   try {
      while (it.hasNext()) {
         hyperLogLog.offer(String.valueOf(it.nextLine().split("[,]")[2]));
         itSize++;
      }
   } finally {
      LineIterator.closeQuietly(it);
   }

   callGc();
   logger.info("("+itSize+"|"+hyperLogLog.sizeof()+")= " + hyperLogLog.cardinality());
   logger.info(""+RegisterSet.REGISTER_SIZE);

   logMemory();
   long end = System.currentTimeMillis();
   logger.info("Processing time: " + (end - start));
   logger.info("==============HLL============");
}
```

I.4 Algoritmo convencional para contagem da frequência de elementos

```
      logger.info("==============Simple Count============");
      logMemory();
      long start = System.currentTimeMillis();
      LineIterator it = FileUtils.lineIterator(new File(FILE_PATH), "UTF-8");
      Map<String, Integer> counter = new TreeMap<String, Integer>();
        List<String> linhas = new ArrayList<String>();

      long itSize = 0;
      try {
         while (it.hasNext()) {
            String line = it.nextLine();
                String ip = line.split("[,]")[2];

                counter.put(ip, (counter.get(ip) == null) ? 1 : counter.get(ip) + 1);
```

```
            itSize++;
        }
    } finally {
        LineIterator.closeQuietly(it);
    }

      List<Map.Entry<String, Integer>> orderedCounter =
          Lists.newArrayList(counter.entrySet());

      Collections.sort(orderedCounter, mapOrdering);
      String outPut = "";
      for(int i = 1; i<=5;i++) {
          outPut = outPut + "["+i+"=" + orderedCounter.get(i) +" ]";
      }
      logger.info(outPut);

    logger.info(search + " ("+itSize+")= " + counter.get(search));
    long end = System.currentTimeMillis();
    logger.info("Processing time: " + (end - start));
    logMemory();
    logger.info("==============Simple Count============");
}

Ordering<Map.Entry<String, Integer>> mapOrdering = new Ordering<Map.Entry<String,
    Integer>>() {
      @Override
      public int compare(Map.Entry<String, Integer> left, Map.Entry<String, Integer>
          right) {
          return right.getValue().compareTo(left.getValue());
      }
 };
```

I.5 Algoritmo Count-Min Sketch para a estimativa da frequência de elementos

```
public void contarFrequenciaCms(String search) throws Exception {
     logger.info("==============CMS============");
     logMemory();
     long start = System.currentTimeMillis();
     LineIterator it = FileUtils.lineIterator(new File(FILE_PATH), "UTF-8");
     CountMinSketch cms1 = new CountMinSketch(10, 4642000, 3181);

       double c=0;
     try {
        while (it.hasNext()) {
```

```
            String line = it.nextLine();
            cms1.add(line.split("[,]")[2], 1);
                c++;
        }
    } catch(Exception e) {
            e.printStackTrace();
      }finally {
        LineIterator.closeQuietly(it);
    }

      logger.info(search + " ("+cms1.size()+")= ["+cms1.getRelativeError()+"] " +
            cms1.estimateCount(5));
    logger.info(search + " ("+cms1.size()+")= ["+cms1.getRelativeError()+"] " +
        cms1.estimateCount(search));
    long end = System.currentTimeMillis();
    logger.info("Processing time: " + (end - start));
    logMemory();
    logger.info("==============CMS============");
}
```

I.6 Monitor de utilização de memória

```
private final void logMemory() {
    for(int i=0;i<3;i++) {
        callGc();
    }

    logger.info("Max Memory: {} Kb",
            Runtime.getRuntime().maxMemory() / 1024);
    logger.info("Total Memory: {} Kb",
            Runtime.getRuntime().totalMemory() / 1024);
    logger.info("Free Memory: {} Kb",
            Runtime.getRuntime().freeMemory() / 1024);
    logger.info("Used Memory: {} Kb",
            (Runtime.getRuntime().totalMemory() - Runtime.getRuntime()
                    .freeMemory()) / 1024);
}

private void callGc() {
    Runtime.getRuntime().gc();
    boolean pause=true;
    long start=System.currentTimeMillis();
    while(pause) {
        if(System.currentTimeMillis()-start>2000) {
            pause=false;
```

```
        }
    }
}
```

I.7 Quantidade de endereços IP distintos agrupados por país

```
//TransporteFeedStreamTask.java
public class TransporteFeedStreamTask implements StreamTask {
   private static final SystemStream OUTPUT_STREAM = new SystemStream("kafka",
       "transporte-raw");

   @Override
   public void process(IncomingMessageEnvelope envelope, MessageCollector collector,
       TaskCoordinator coordinator) {
      Map<String, Object> outgoingMap = TransporteFeedEvent.toMap((TransporteFeedEvent)
          envelope.getMessage());
      collector.send(new OutgoingMessageEnvelope(OUTPUT_STREAM, outgoingMap));
   }
}

//TransporteParserStreamTask.java
public class TransporteParserStreamTask implements StreamTask {
   @SuppressWarnings("unchecked")
   @Override
   public void process(IncomingMessageEnvelope envelope, MessageCollector collector,
       TaskCoordinator coordinator) {
      Map<String, Object> jsonObject = (Map<String, Object>) envelope.getMessage();
      TransporteFeedEvent event = new TransporteFeedEvent(jsonObject);

      try {
         Map<String, Object> parsedJsonObject = new HashMap<String, Object>();

         parsedJsonObject.put("user", event.getUser());
         parsedJsonObject.put("country", event.getCountry());
         parsedJsonObject.put("time", event.getTime());

         collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka",
             "transporte-evts"), parsedJsonObject));
      } catch (Exception e) {
         System.err.println("Unable to parse line: " + event);
      }
   }
}

//TransporteTypeStreamTask.java
```

```
public class TransporteTypeStreamTask implements StreamTask, WindowableTask {

    private final static List<String> countries = new
        ArrayList<String>(Arrays.asList("BR", "US", "ES", "MX", "JP", "AR", "IT", "SE"));
    private Map<String, Integer> counts = new HashMap<String, Integer>();

    @SuppressWarnings("unchecked")
    @Override
    public void process(IncomingMessageEnvelope envelope, MessageCollector collector,
        TaskCoordinator coordinator) {

        Map<String, Object> evt = (Map<String, Object>) envelope.getMessage();
        String strCntry = String.valueOf(evt.get("country"));

        Integer counter=0;
        if(counts.containsKey(strCntry)) {
            counter = counts.get(strCntry);
        }
        counts.put(strCntry, counter+1);

        Integer counterTotal=0;
        if(counts.containsKey("total")) {
            counterTotal = counts.get("total");
        }
        counts.put("total", counterTotal+1);
    }

    @Override
    public void window(MessageCollector collector, TaskCoordinator coordinator) {

        collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka",
            "transporte-counts"), counts));
//        collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka",
    "transporte-types-groups"), groups));

        // Reset groups after windowing.
    evts = 0;
   groups = new HashMap<String, List<Object>>();
   counts = new HashMap<String, Integer>();
    }
}


# transporte-feed.properties
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
```

```
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied. See the License for the
# specific language governing permissions and limitations
# under the License.

# Job
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=transporte-feed

# YARN
yarn.package.path=file://${basedir}/target/${project.artifactId}-${pom.version}-dist.tar.gz

# Task
task.class=tum.examples.transporte.task.TransporteFeedStreamTask
task.inputs=transporte.tums
task.opts=-agentlib:jdwp=transport=dt_socket,address=localhost:9009,server=y,suspend=n

# Serializers
serializers.registry.json.class=org.apache.samza.serializers.JsonSerdeFactory

# Transporte System
systems.transporte.samza.factory=tum.examples.transporte.system.TransporteSystemFactory

# Kafka System
systems.kafka.samza.factory=org.apache.samza.system.kafka.KafkaSystemFactory
systems.kafka.samza.msg.serde=json
systems.kafka.consumer.zookeeper.connect=localhost:2181/
systems.kafka.producer.metadata.broker.list=localhost:9092
systems.kafka.producer.producer.type=sync
# Normally, we'd set this much higher, but we want things to look snappy in the demo.
systems.kafka.producer.batch.num.messages=100

# transporte-parser.properties
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
```

```
# "License"); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied. See the License for the
# specific language governing permissions and limitations
# under the License.

# Job
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=transporte-parser

# YARN
yarn.package.path=file://${basedir}/target/${project.artifactId}-${pom.version}-dist.tar.gz

# Task
task.class=tum.examples.transporte.task.TransporteParserStreamTask
task.inputs=kafka.transporte-raw
task.checkpoint.factory=org.apache.samza.checkpoint.kafka.KafkaCheckpointManagerFactory
task.checkpoint.system=kafka
# Normally, this would be 3, but we have only one broker.
task.checkpoint.replication.factor=1

# Metrics
metrics.reporters=snapshot,jmx
metrics.reporter.snapshot.class=org.apache.samza.metrics.reporter.MetricsSnapshotReporterFactory
metrics.reporter.snapshot.stream=kafka.metrics
metrics.reporter.jmx.class=org.apache.samza.metrics.reporter.JmxReporterFactory

# Serializers
serializers.registry.json.class=org.apache.samza.serializers.JsonSerdeFactory
serializers.registry.metrics.class=org.apache.samza.serializers.MetricsSnapshotSerdeFactory

# Systems
systems.kafka.samza.factory=org.apache.samza.system.kafka.KafkaSystemFactory
systems.kafka.samza.msg.serde=json
systems.kafka.consumer.zookeeper.connect=localhost:2181/
systems.kafka.consumer.auto.offset.reset=largest
systems.kafka.producer.metadata.broker.list=localhost:9092
systems.kafka.producer.producer.type=sync
# Normally, we'd set this much higher, but we want things to look snappy in the demo.
systems.kafka.producer.batch.num.messages=100
systems.kafka.streams.metrics.samza.msg.serde=metrics
```

```
#transporte-type.properties
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied. See the License for the
# specific language governing permissions and limitations
# under the License.

# Job
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=transporte-type

# YARN
yarn.package.path=file://${basedir}/target/${project.artifactId}-${pom.version}-dist.tar.gz

# Task
task.class=tum.examples.transporte.task.TransporteTypeStreamTask
task.inputs=kafka.transporte-evts
task.window.ms=500

# Serializers
serializers.registry.json.class=org.apache.samza.serializers.JsonSerdeFactory

# Systems
systems.kafka.samza.factory=org.apache.samza.system.kafka.KafkaSystemFactory
systems.kafka.samza.msg.serde=json
systems.kafka.consumer.zookeeper.connect=localhost:2181/
systems.kafka.consumer.auto.offset.reset=largest
systems.kafka.producer.metadata.broker.list=localhost:9092
systems.kafka.producer.producer.type=sync
# Normally, we'd set this much higher, but we want things to look snappy in the demo.
systems.kafka.producer.batch.num.messages=10
```