Universidade do Minho
Escola de Engenharia

Cristiano José Ribeiro Miranda

Processamento em *Streaming:* Avaliação de *Frameworks* em contexto *Big Data*

Dissertação de Mestrado

Mestrado Integrado em Engenharia e Gestão de Sistemas de Informação

Trabalho efetuado sob a orientação de

Professora Doutora Maribel Yasmina Santos

Dezembro,2018

Aos meus pais
Francisco e Olívia

Resumo

Nos dias de hoje, o vasto volume de dados produzido é um dos focos de atenção da comunidade científica de Sistemas de Informação. As ferramentas de gestão de dados tradicionais existentes não conseguem processar estes dados em tempo útil, sendo por isso necessário utilizar tecnologias mais adequadas de forma a possibilitar o processamento de um volume de dados mais elevado. Neste contexto, surge o termo *Big Data,* que descreve conjuntos de dados de grandes dimensões, de diferentes tipos e com diferentes graus de complexidade. *Big Data* tem um papel de extrema importância seja qual for a área de negócio, auxiliando a tomada de decisão e perceção das tendências futuras, alavancando a vantagem competitiva das organizações. Apesar das reconhecidas vantagens de *Big Data* e das tecnologias associadas, as aplicações que requerem processamento em tempo real de grandes fluxos de dados têm levado ao limite estas tecnologias. Para colmatar estas limitações surgiram novas ferramentas de processamento de dados em *streaming*. Estas ferramentas permitem a obtenção de resultados com tempos de espera reduzidos e resolvem o problema da elevada latência que os sistemas de processamento anteriores apresentavam. O objetivo desta dissertação é realizar um *benchmark* das principais *frameworks* de processamento em *streaming* no contexto *Big Data*. Para o sucesso da mesma realizou-se um enquadramento conceptual e tecnológico, onde foram levantados os principais conceitos associados ao termo *Big Data*, assim como das principais técnicas e ferramentas com especial destaque no *streaming*. Para a elaboração do benchmark, foi definida uma infraestrutura tecnológica no Google Cloud Platform e ainda os indicadores e métricas para posterior análise.

Concluídos todos os testes definidos, foi possível perceber o comportamento de cada framework, as suas vantagens e desvantagens face ás diferentes necessidades no contexto de streaming.

Palavras-Chave

Big Data, Streaming, Real-time, Benchmark, *Spark Streaming, Flink*

Abstract

Nowadays, the vast volume of data produced is one of the focus of attention of the scientific community of Information Systems. Existing traditional data management tools are unable to process these data in a timely manner, so it is necessary to use more appropriate technologies in order to allow the processing a higher volume of data. In this context, the term *Big Data* appears, which describes large dimensions *datasets*, of different types and with different degrees of complexity. *Big Data* plays an extremely important role in all business areas, helping to make decisions and perceive future trends, leveraging the competitive advantage of organizations. Despite the recognized advantages of *Big Data* and associated technologies, applications that require real-time processing of large data streams have pushed these technologies to the limit. To address these limitations, new tools for *streaming* data processing have emerged. These tools allow the obtaining of results with reduced waiting times and solve the problem of high latency that previous processing systems had. The objective of this dissertation is to perform a benchmark of the main processing *streaming frameworks* in the *Big Data* context. For his success was realized a conceptual and technological framework, where were raised the main concepts associated with the term *Big Data*, as well as of the main techniques and tools with special emphasis in *streaming* tools. For the elaboration of the benchmark, a technological infrastructure was defined in the Google Cloud Platform and all the indicators and metrics needed for later analysis.

After all the tests were executed, it was possible to perceive the behavior of each framework, its advantages and disadvantages in relation to different needs in the context of streaming.

Keywords

Big Data, Stream, Real-time, Benchmark, Spark Streaming, Flink

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE GRÁFICOS

ÍNDICE DE TABELAS

Lista de Abreviaturas, Siglas e Acrónimos

Este documento utiliza um conjunto de abreviaturas, siglas e acrónimos listados de seguida:

ACID - Atomicity, Consistency, Isolation, Durability

API - Application Programming Interface

BASE - Basically Available, Soft-State and Eventual Consistency

BSP - Bulk Synchronous Parallel

CAP - Consistency, Availability, Partition Tolerance

COTS - Commercial Off The Shelf

CPU – Central Processing Unit

DAG - Directed Acyclic Graph

DSRM - Design Science Research Methodology

DVD - Digital Video Disc

ETL - Extract Transform Load

HDD – Hard Disk Drive

HDFS - Hadoop Distributed File System

IBM - International Business Machines

JSON - JavaScript Object Notation

NBDRA - NIST *Big Data* Reference Architecture

NIST - National Institute of Standards and Technology

NoSQL - Not Only Structured Query Language

RAM - Random Access Memory

RDBM - Relational Database Management System

RDD - Distributed Resilient Dataset

SPB - Sistemas de Processamento em Batch

SPS - Sistemas de Processamento em Streaming

SQL - Structured Query Language

UI - User Interface

YARN - Yet Another Resource Negotiator

1. INTRODUÇÃO

Neste capítulo são apresentados o enquadramento e a motivação do presente projeto de dissertação. É apresentada a abordagem metodológica que será utilizada, a descrição das tarefas a realizar e, ainda, a sua calendarização. Por fim, é apresentado o processo de revisão bibliográfica utilizado no trabalho de investigação realizado.

Posteriormente, são apresentados os contributos que resultarão deste projeto de investigação e, por fim, é apresentada a organização do presente documento.

1.1 Enquadramento e motivação

Vivemos numa era de *Big Data*, onde todos os dias são criados mais de 2500 petabytes de dados, sendo de destacar que 90% dos dados existentes em 2014 foram criados apenas nos dois anos anteriores (Wu, Zhu, Wu, & Ding, 2014). Com o crescimento acentuado dos dados é fácil compreender que as ferramentas tradicionais que gerem os mesmos não são capazes de o fazer em tempo útil (Wu et al., 2014).

O termo *Big Data* pode ser definido como vastos volumes de dados, de diferentes tipos, com diferentes graus de complexidade, gerados a uma velocidade variável e que não conseguem ser processados utilizando tecnologias ou processos tradicionais (Krishnan, 2013).

A incapacidade de gestão de grandes volumes de dados por parte das organizações pode levar a que estas percam ou desperdicem oportunidades de mercado que poderiam ser benéficas para o seu negócio. Para que tal não aconteça, e utilizando tecnologias/ferramentas mais adequadas, é possível efetuar uma gestão que permita uma análise de um volume mais elevado de dados, levando as organizações a compreenderem melhor o seu próprio negócio, os seus clientes e, desta forma, proporcionar uma resposta mais adequada aos mesmos (Zikopoulos, Eaton, DeRoos, Deutsch, & Lapis, 2011).

Para dar resposta a este crescente aumento de dados surgiu o Hadoop, um ecossistema que integra um conjunto de aplicações que permitem o processamento de grandes quantidades de dados utilizando a computação distribuída. Embebida nesta solução encontra-se o MapReduce, um *framework* de processamento de dados que revolucionou o mundo *Big Data* (Hadoop, 2017).

É de extrema importância que as organizações saibam o que acontece no momento, em tempo real, de forma a reagir e antecipar novas oportunidades de negócio (Bifet, 2013). De forma a responder

a estas necessidades torna-se indispensável e pertinente o estudo dos novos *frameworks* capazes de trabalhar com *streams* de *Big Data*.

1.2 Objetivos e resultados esperados

A presente dissertação tem como principais objetivos a identificação das várias características diferenciadoras de um conjunto de *frameworks* de processamento de dados em *streaming*, assim como a avaliação do seu desempenho em contextos de utilização específicos. Para a análise de desempenho, será definido um protocolo de testes que permita avaliar as diferentes *frameworks* em contextos de recolha e processamento de *streams* de dados.

Em termos de resultados espera-se:

- Contextualização das tecnologias e principais cenários de utilização;
- Definição de protocolo de avaliação das *frameworks* de *streaming* selecionadas;
- Avaliação do seu desempenho utilizando um benchmark apropriado;
- Caracterização das vantagens e desvantagens das diferentes tecnologias no processamento de vastas quantidades de dados.

1.3 Abordagem metodológica

A metodologia de investigação que será aplicada para o desenvolvimento da dissertação é a "Design Science Research Methodology (DSRM) for Information Systems," (Peffers, Tuunanen, Rothenberger, & Chatterjee, 2007). Esta metodologia enquadra-se nesta investigação uma vez que dela irá surgir um artefacto, bem como a sua avaliação de forma a resolver um problema existente. Este artefacto é considerado relevante para a solução uma vez que vai permitir às organizações escolher a melhor *framework* de processamento em *streaming* de acordo com os parâmetros definidos. Deste modo, pode-se dizer que o artefacto que irá resultar será uma instanciação, pois permite que constructos, modelos ou métodos possam ser implementados num sistema operacional, demonstrando ainda a sua viabilidade e a avaliação concreta da sua adequação ao seu propósito (Hevner, March, Park, & Ram, 2004; Peffers et al., 2007).

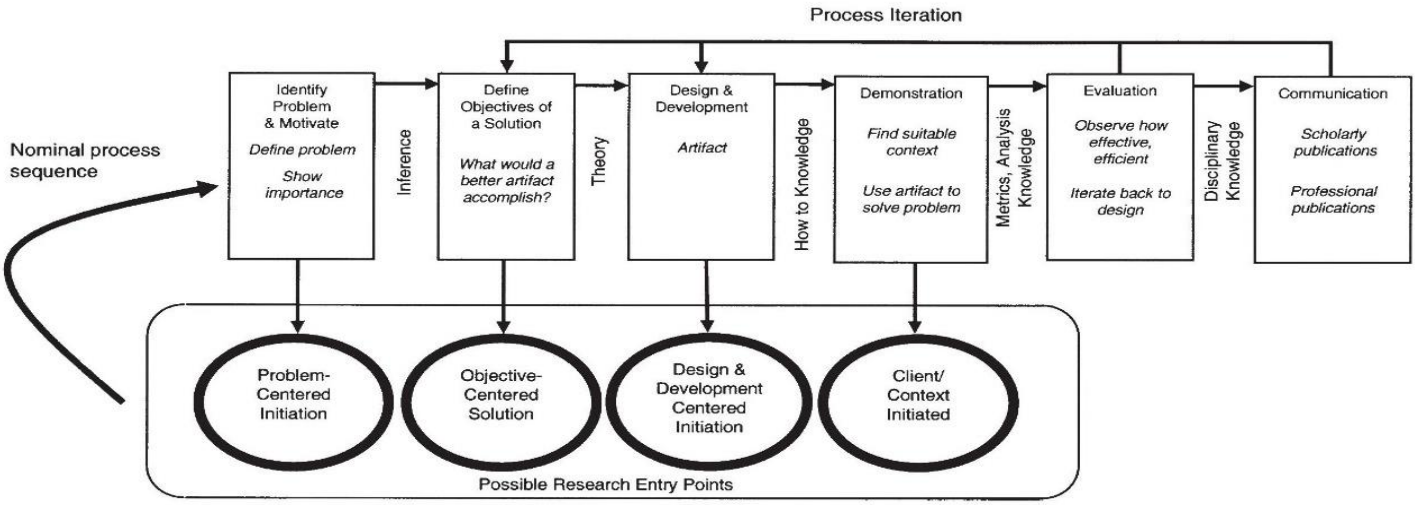

Figura 1 - DSRM for Information Systems. Retirado de (Peffers et al., 2007).

A metodologia "*Design Science Research Methodology (DSRM) for Information Systems*" encontra-se dividida em seis fases ilustradas na Figura 1.

A implementação da abordagem metodológica a seguir neste trabalho será executada através das seguintes tarefas:

- **Plano de Trabalho** – Elaboração do presente documento onde consta a enquadramento e motivação, objetivos e resultados esperados, abordagem metodológica e calendarização das tarefas.
- **Identificação do Problema e Motivação** – Identificação do problema que a dissertação pretende resolver. A motivação advém da tentativa de encontrar a solução para o problema.
- **Definição dos Objetivos da Solução** – Identificação e definição dos objetivos que permitirão resolver o problema anteriormente identificado.
- **Revisão de Literatura** – Seleção e análise de literatura científica adequada para a realização da dissertação.
- **Enquadramento Tecnológico** – Estudo das tecnologias de *Big Data* que serão utilizadas ao longo da dissertação.
- **Conceção e Desenvolvimento** – Elaboração de vários cenários que incluem modelos e testes com os diferentes *frameworks* de processamento de *streaming* de dados em contextos de *Big Data*.

- **Demonstração** – Verificação dos contextos em que cada *framework* se destaca das restantes, salientando os contextos organizacionais em que se recomenda a utilização de uma determinada *framework*.
- **Avaliação** – Avaliação dos vários cenários apresentados, comparando os resultados obtidos entre eles assim como a comparação com outros trabalhos desenvolvidos sobre a mesma temática. Nesta fase poderão surgir aos testes ou casos de demonstração utilizados.
- **Comunicação** – Difundir na comunidade científica os resultados obtidos, através da escrita da dissertação, podendo ainda surgir a publicação de artigos científicos.

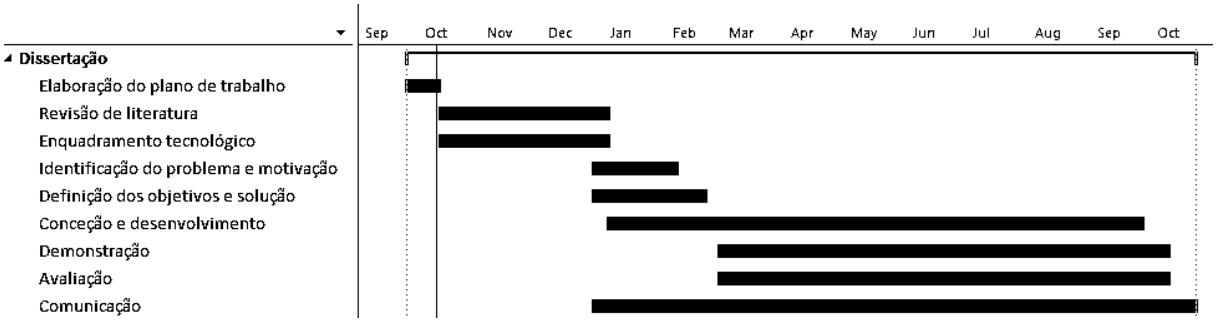

Figura 2 - Diagrama de Gant com as tarefas associadas à dissertação

1.4 Processo de revisão bibliográfica

Nos dias de hoje, a pesquisa e seleção de literatura adequada a uma dissertação pode ser difícil, devido à disponibilidade de um elevado número de documentos na área em questão. Assim, no processo de recolha de literatura foram definidas palavras-chave, bases de dados de referência e um método de avaliação dos artigos.

Tendo em conta que o trabalho pertence a uma área relativamente recente, não foram impostas quaisquer limitações temporais, dando-se maior importância à relevância do documento para o presente trabalho.

O processo de pesquisa bibliográfica foi realizado nas bases de dados, "Scopus", "IEEE Xplorer", "repositoriUM" utilizando as palavras-chave "*Big Data*", "*streaming*", "*processing frameworks*", "*benchmark*" isoladamente ou em associação. Uma vez obtidos os resultados da pesquisa, os artigos foram selecionados com base no título e no *abstract* para posteriormente serem devidamente analisados para serem organizados por ordem de relevância. Os artigos considerados relevantes foram lidos na integra, e analisados com detalhe, e foi feito um levantamento de conceitos de forma a criar uma tabela de cruzamento para facilitar a concretização do presente documento.

1.5 Organização do documento

O documento encontra-se estruturado em seis capítulos explanados de seguida. No capítulo 1 é introduzido o tema da dissertação, definidos os objetivos e resultados esperados e a abordagem metodológica. No capítulo 2 é feita a revisão de literatura sobre o estado de arte relativo a Big Data e Frameworks de Streaming. As técnicas e tecnologias relevantes para o desenvolvimento da dissertação são apresentadas no capítulo 3. No capítulo 4 é exposta a metodologia utilizada para a elaboração do benchmark proposto. A apresentação dos resultados e a sua discussão é realizada no capítulo 5 e 6, respetivamente.

2. *BIG DATA*

Vivemos atualmente numa era de *Big Data*, tendo sido criados cerca de 2500 petabytes de dados até 2014 e 90% dos quais foram gerados apenas nos últimos dois anos (Wu et al., 2014).

Como se pode verificar na Figura 3, o tráfego de dados na internet aumentou de forma drástica nas últimas duas décadas, passando de 100 gigabyte por dia para cerca de 26600 gigabytes por segundo. Segundo o *Witepaper* da Cisco, o trafego de internet deverá aumentar para 105800 gigabytes por segundo em 2021 (CISCO, 2017).

Year	Global Internet Traffic
1992	100 GB per day
1997	100 GB per hour
2002	100 GB per second
2007	2,000 GB per second
2016	26,600 GB per second
2021	105,800 GB per second

Figura 3 - Global Internet Traffic. Adaptado de (CISCO, 2017).

Com o aumento acentuado do volume de dados, surge o termo *Big Data* que normalmente é utilizado para descrever conjuntos de dados de grandes dimensões. O volume de dados associado ao fenómeno *Big Data* traz novos desafios, para além do volume, pois estes dados são mais variados (estruturados, semiestruturados ou não-estruturados) e complexos, pois advêm da massificação do uso de sensores, dispositivos inteligentes e outras tecnologias (Zikopoulos et al., 2011).

2.1 Conceito

Big Data é um conceito abstrato, que segundo (Krishnan, 2013) pode ser definido como um variado volume de dados, com diferentes graus de complexidade, gerados a uma velocidade variável e que não conseguem ser processados utilizando tecnologias ou processos tradicionais.

O termo *Big Data* surgiu em 2005 introduzido por Roger Magoulas numa conferência da O'Reilly Media, com o objetivo de definir uma grande quantidade de dados que as técnicas tradicionais não conseguiam processar devido ao seu tamanho e complexidade elevada (Popeanga & Lungu, 2012). Posteriormente, em 2010, Apache Hadoop definiu *Big Data* como *datasets* que não conseguem ser capturados, geridos e processados por computadores convencionais (Hadoop, 2017).

Big Data passa então a ser utilizado para processar um conjunto de dados que são difíceis de serem adquiridos, armazenados, visualizados e analisados, com um elevado volume e complexidade e em rápido crescimento, como já referido (Liu, Yang, & Zhang, 2013). Para além disto, *Big Data* passa ainda a representar mais variedade, mais quantidade, mais utilizadores e maior velocidade, sendo a utilização de um grande volume de dados vista como um desafio, mas ao mesmo tempo uma oportunidade para as organizações adquirirem e aumentarem à sua eficácia (Krishnan, 2013; Mathur, Sihag, Bagaria, & Rajawat, 2014).

Nos últimos anos surgiu a preocupação com o grande volume de dados produzidos e, por consequência, a necessidade de infraestruturas com maior capacidade de armazenamento de forma a albergar todos os dados gerados. Este crescimento deve-se às alterações nos modelos de negócio das organizações que passam a adotar serviços via internet, bem como ao surgimento de organizações orientadas a extração de valor de grandes quantidades de dados como o caso da Google, Facebook e Twiter (Chandarana & Vijayalakshmi, 2014).

Observando as definições de *Big Data* dos vários autores já citados, pode-se destacar três características principais sendo elas o Volume, a Variedade e a Velocidade. Tais características estão na base do modelo dos 3Vs (Gandomi & Haider, 2015):

- **Volume** - caracteriza a quantidade de dados que é gerada continuamente e que pode provir de múltiplas fontes. A sua natureza massiva é normalmente representada por petabytes e distingue *Big Data* das demais tecnologias e ferramentas ditas tradicionais. O volume passa assim a representar o "Big" em *Big Data* (Demchenko, Grosso, De Laat, & Membrey, 2013; Katal, A., Wazid, M., & Goudar, 2013).
- **Variedade** - refere-se à heterogeneidade estrutural de um *dataset*, podendo este ser composto por dados completamente estruturados, não estruturados ou ainda semiestruturados. A sua natureza pode, também, resultar de várias fontes distintas.
- **Velocidade -** representa a celeridade à qual os dados são gerados, sendo que complementa a caracterização de volume. A velocidade representa, também, a rapidez com que os dados gerados devem ser analisados, bem como a utilização da informação resultante.

Tendo por base o modelo dos 3Vs de Krishan (2013), verifica-se a necessidade de cruzar entre si cada uma das características deste modelo, tal como retratado na Figura 4.

- **Ambiguidade** – esta manifesta-se sobretudo na combinação das características volume-variedade, sendo que a existência dos metadados bem definidos passam a ser um elemento crucial para que a ambiguidade seja reduzida.
- **Viscosidade** – prende-se com a resistência que surge nos fluxos de dados. Esta pode se manifestar, para além do seu aparecimento nos fluxos de dados, nos modelos de negócio ou ainda numa limitação tecnológica.
- **Viralidade** – remete à capacidade de medir e descrever a rapidez que os dados se propagam numa rede *people-to-people*. De notar que a medida utilizada para verificar a taxa de propagação numa rede é o "tempo".

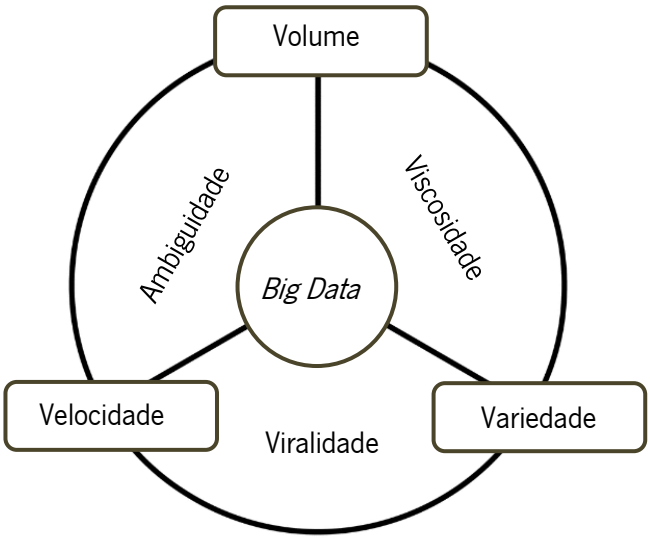

Figura 4 - Modelo 3Vs. Adaptado de (Krishnan, 2013).

Durante um período de pelo menos 10 anos, o modelo dos 3Vs foi uma referência em termos de conceito de *Big Data*, sendo este utilizado por organizações como a IBM e a Microsoft (Chen, Mao, & Liu, 2014). Posteriormente, surgem mais duas características associadas ao modelo apresentado dando origem ao modelo denominado de 5Vs, como representado na Figura 5. O Valor e a Veracidade aparecem devido ao crescente interesse por parte da comunidade no tema *Big Data* (Costa & Santos, 2017):

- **Valor -** pode ser obtido através da integração de diferentes tipos de dados de forma a melhorar a eficiência das organizações e assim ganharem vantagens competitivas. Este representa os resultados esperados do processamento e da análise de *Big Data*. Uma forma de evidenciar a sua importância pode ser demonstrada pela indústria médica dos Estados Unidos da América,

ou seja, o valor potencial de poupança pode ultrapassar os 300 mil milhões de dólares, reduzindo a despesa com o *healthcare* em 8% (Chen et al., 2014).

- **Veracidade -** tem em consideração a possibilidade de existência de dados imprecisos nos *datasets*, e potencia que estes sejam usados para a obtenção de valor através da utilização de técnicas e tecnologias adequadas.

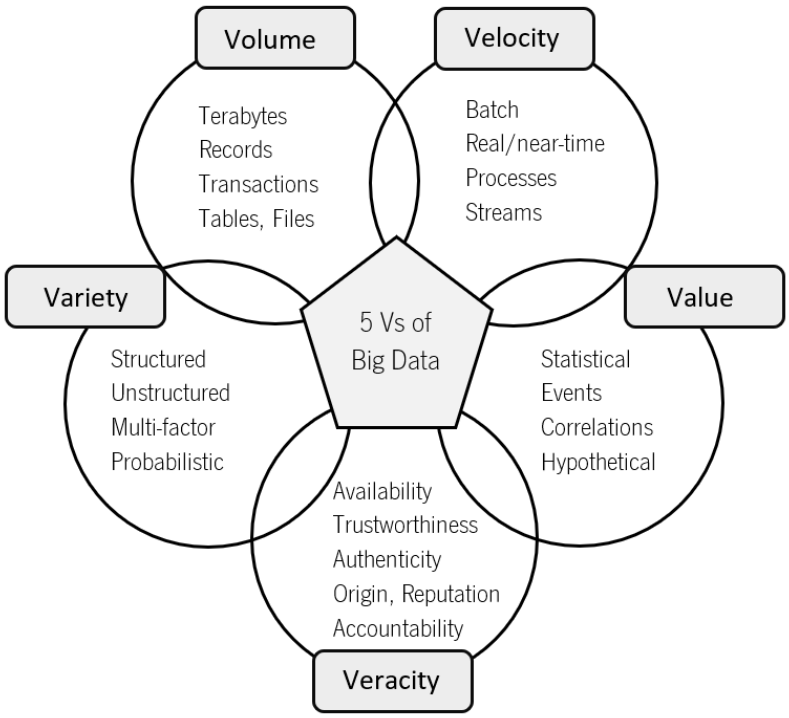

Figura 5 - Modelo 5Vs. Retirado de (Chandarana & Vijayalakshmi, 2014).

Não obstante, a SAS (considerada a empresa pioneira em *Business Inteligente* e bases de dados) introduz mais duas dimensões adicionais de *Big Data*, nomeadamente a variabilidade e a complexidade (Gandomi & Haider, 2015):

- **Variabilidade** - refere-se à variação do fluxo dos dados, uma vez que a velocidade de chegada dos dados tem picos e declínios ao longo do tempo.
- **Complexidade** - refere-se ao facto de os dados serem gerados por uma quantidade enorme de fontes, sendo que estas impõem um desafio que pode ser considerado crítico no que toca à necessidade de conectar, integrar, limpar e transformar os dados provenientes dessa variedade.

2.2 Oportunidades e desafios

Uma das características de *Big Data* reside no alcance de novas oportunidades, potenciando o conhecimento mais detalhado e possibilitando a descoberta de padrões e tendências que muitas vezes se encontram ocultas nos dados. Estes benefícios, bem como a transparência na análise dos dados, detém um grande e crescente poder no que toca à criação de valor para o negócio das organizações bem como para os consumidores (Chen et al., 2014; Kaisler, Armour, Espinosa, & Money, 2013). Acelera, também, o progresso cientifico e proporciona inovação permitindo levantar novas questões, fomentando o desenvolvimento de algoritmos e de novas ferramentas de análise de dados que se repercutirão em crescimento económico e ainda na melhoria da qualidade de vida (Chen, H., Chiang, R. H., & Storey, 2012).

As ferramentas de *Big Data* apresentam várias vantagens, sendo uma delas a possibilidade de análise de dados independente do seu formato, podendo os dados serem estruturados, semiestruturados ou não estruturados, levando ao conhecimento dos dados relevantes, ou irrelevantes, para a organização (Zikopoulos et al., 2011). Para além dos benefícios já apresentados, percebe-se que *Big Data* tem um papel de extrema importância seja qual for a área de negócio, permitindo tomar as melhores decisões e conhecer tendências futuras, alavancando a vantagem competitiva das organizações (Chandarana & Vijayalakshmi, 2014).

Uma vez apresentadas algumas das oportunidades que *Big Data* oferece à comunidade, passa a ser importante entender quais os seus desafios. Segundo Kaisler et al. (2013), existem três desafios técnicos fundamentais quando se trabalha em contexto de *Big Data*, sendo os mesmos relacionados com armazenamento, gestão e processamento:

- **Armazenamento e Transferência** – a tecnologia de armazenamento atual está limitada a cerca de 16 terabytes por disco, sendo necessários 65536 discos rígidos para o armazenamento de 1 exabyte de dados excluindo os discos necessários para a redundância dos mesmos. Para se ter ideia do que representa 1 exabyte, este corresponde a aproximadamente duzentos e quarenta milhões de DVD's comuns. A tecnologia atual de comunicação por fibra ótica permite 1 gigabits por segundo. Com esta taxa de transferência serão necessários 278 anos para a transferência de 1 exabyte.
- **Gestão** – questões como a resolução de problemas de acesso, metadados, direitos de utilização e atualização são consideradas as principais barreiras ao nível da gestão de dados. A recolha de dados de forma digital é mais flexível que a recolha de dados através de métodos manuais onde

são seguidos protocolos rígidos com o objetivo de garantir precisão e validade. Dado o grande volume de dados envolvidos na captação de dados digitais, passa a ser impraticável a validação de todos os itens desses mesmos dados. Este facto leva à necessidade de novas abordagens para a qualificação e validação dos dados.

- **Processamento** – assumindo que 1 exabyte necessita de ser totalmente processado, que os dados estão divididos em blocos de 8 palavras e que o processador de 5 GHz utiliza 100 instruções para processar cada bloco de dados, seriam necessários 625 anos para processar a totalidade dos dados. Desta forma, é necessária capacidade de processamento paralelo e novos algoritmos analíticos para obter informação atual e em tempo adequado.

Para além dos três desafios técnicos apresentados anteriormente, Chen et al. (2014) apresentam vários obstáculos no desenvolvimento e implementação de aplicações *Big Data*. Estes, segundo os autores, podem ser assumidos como desafios, sendo eles:

- **Representação dos dados** - Dada a heterogeneidade dos dados usados em *Big Data*, é importante que a sua representação reflita a sua estrutura de modo a manter o valor dos mesmos, o que permitirá um processamento eficiente dos *datasets*.
- **Redução de redundância e compressão de dados** – Existe uma grande redundância de dados nos *datasets* utilizados em *Big Data*, como tal a diminuição da mesma e a aplicação de compressão dos dados permitirá reduzir custos, não afetando o valor dos dados.
- **Gestão do ciclo de vida dos dados** – A geração de dados cresceu a um ritmo maior que as tecnologias de armazenamento dos mesmos, não sendo possível armazenar tudo o que é gerado. Passa a ser importante, portanto, definir a importância dos dados de forma a decidir o que armazenar e o que descartar.
- **Mecanismos analíticos** – Os mecanismos analíticos de *Big Data* devem processar grandes volumes de dados heterogéneos num espaço de tempo limitado, tendo em conta as limitações de escalabilidade das bases de dados relacionais opta-se pela adoção de bases de dados NoSQL. No entanto, dados os problemas de performance das bases de dados NoSQL, é recomendável optar-se por soluções híbridas.
- **Confidencialidade dos dados** – A grande maioria dos proprietários de grandes volumes de dados não tem a capacidade para os processar, sendo que a forma de contornar este problema passa a ser a subcontratação do serviço a organizações externas. Uma vez que os dados podem conter informações sensíveis para a organização, como por exemplo números de cartões de

crédito dos seus clientes, tomar medidas preventivas para a proteção desses dados passa a ser extremamente importante.

- **Consumo energético** – O aumento do volume dos dados e das operações associadas ao processamento, armazenamento, análise e transmissão dos mesmos, reflete-se no inevitável aumento de consumo energético. Desta forma, e de modo a contornar este problema, torna-se pertinente implementar um sistema de controlo e mecanismos de gestão de consumo de forma a otimizar o mesmo.
- **Escalabilidade** – Um sistema analítico de *Big Data* deve suportar a capacidade de processamento dos *datasets* atuais e os futuros, devendo estar preparado para suportar o aumento de complexidade e tamanho dos dados.
- **Cooperação** – A análise de *Big Data* é uma área multidisciplinar, que requer que especialistas nas mais diversas áreas criem sinergias para obter o total potencial de *Big Data*. Assim, deve ser estabelecida uma arquitetura de cooperação em *Big Data* de forma a ajudar todos os membros envolvido.

2.3 Arquiteturas de suporte

Em contexto de *Big Data*, e segundo a literatura, existem duas arquiteturas de referencia que são frequentemente aplicadas, nomeadamente a arquitetura Lambda e a arquitetura de referência da *National Institute of Standards and Technology* (NIST) (Costa & Santos, 2017).

2.3.1 Arquitetura Lambda

Em 2011, Nathan Marz introduziu a Arquitetura Lambda representada abaixo Figura 6. Esta é considerada um modelo híbrido que utiliza várias camadas, sendo:

- **Batch –** Permite a gestão de um *dataset* mestre e a análise em grande escala de dados históricos.
- **Speed -** Tem baixa latência e permite o processamento dos novos dados.

- **Serving -** Permite a visualização dos dados provenientes das duas primeiras camadas.

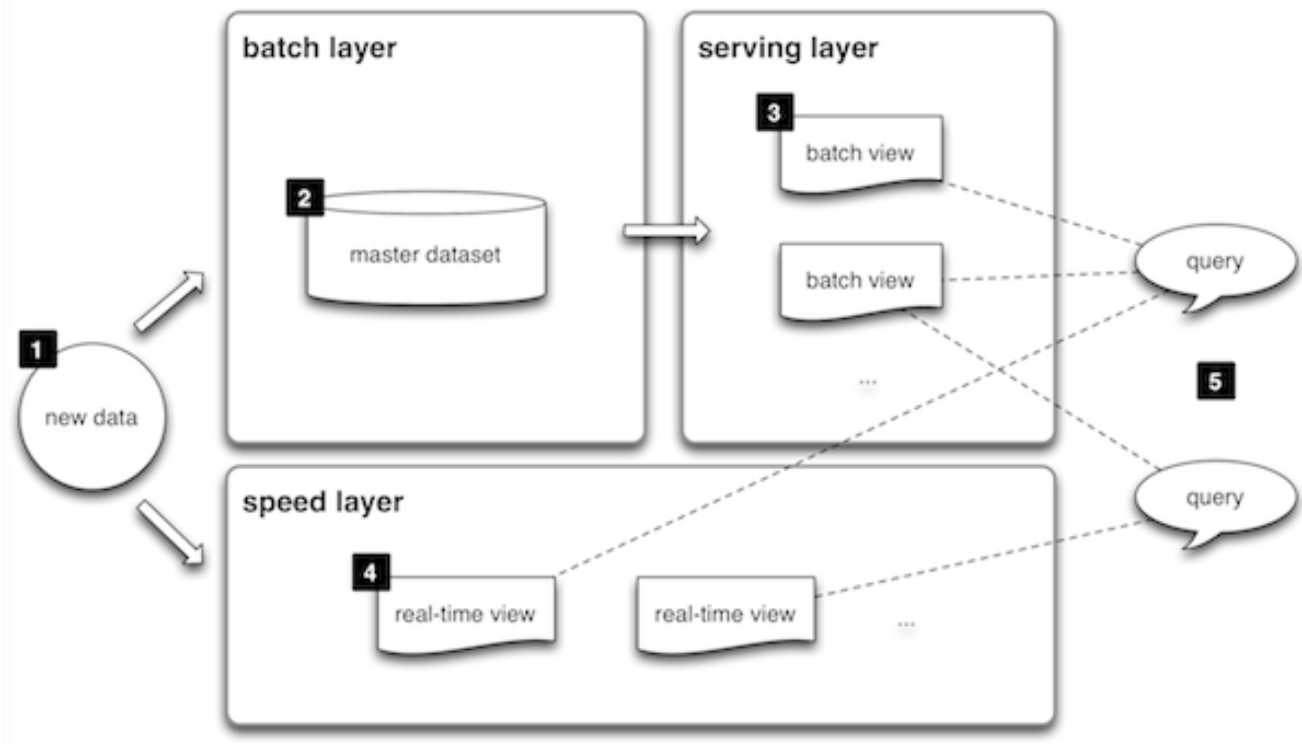

Figura 6 - Arquitetura Lambda. Retirado de (Hausenblas & Bijnens, 2017).

A Arquitetura Lambda pretende satisfazer as necessidades de um sistema robusto e ainda permitir tolerância a falhas de caracter humano e de *hardware*. Esta é capaz de satisfazer um grande leque de *use cases* nos quais a necessidade de leitura e atualização dos dados com latências baixas sejam uma prioridade (Hausenblas & Bijnens, 2017).

De acordo coma arquitetura representada na Figura 6, todos os novos dados são enviados para a camada *batch* e *speed* dependendo do seu tipo. Na camada *batch* os novos dados são adicionados ao *dataset master,* sendo este composto por um conjunto de ficheiros que contêm apenas dados não tratados. Esta camada pré-computa, continuamente, algumas funções de tratamento e agregação de dados num ciclo *"while (true)"* resultando em novas *batch views*.

A camada *serving* funciona como uma base de dados escalável que troca as *views* criadas pela camada *batch* à medida que elas são disponibilizadas. Tendo em conta a alta latência da camada *batch,* os resultados disponíveis para a camada *serving* passam a herdar esta característica.

De forma a mitigar a alta latência já referida, a camada *speed* produz *views* em tempo real e atualizadas sendo armazenadas em bases de dados para consulta e escrita. Estas *views* são descartadas

à medida que os mesmos dados são computados pela camada *batch* e disponibilizados na camada *serving* (Marz & Warren, 2015).

2.3.2 Arquitetura NIST

A *NIST Big Data Reference Architecture* (NBDRA) foi criada de forma a representar um modelo conceptual independente e neutro em aspetos infraestruturais, tecnológicos e comerciais. O modelo conceptual da NBDRA, representado na Figura 7, é subdividido em cinco componentes lógicos conectados por interfaces. Todos os componentes lógicos inserem-se em dois envelopes, que representam nomeadamente a camada de gestão e a camada de segurança e privacidade (Chang, 2015).

Os cinco componentes lógicos e a sua função na arquitetura apresentada são:

- ***System Orchestrator*** - Define e integra as aplicações de dados necessárias.
- ***Data Provider*** – Introduz novos dados ou *feeds* de informação no sistema *Big Data*.
- ***Big Data Application Provider*** – Executa o ciclo de vida dos dados, desde a recolha ao acesso, de forma a ir de encontro aos requisitos de segurança e privacidade assim como os requisitos impostos pelo *System Orchestrator.*
- ***Big Data Framework Provider*** – Estabelece um *framework* de computação no qual serão executadas as devidas transformações garantindo a privacidade e integridade dos dados.
- ***Data Consumer*** – Inclui os utilizadores finais ou outros sistemas que usam os resultados do *Application Provider.*

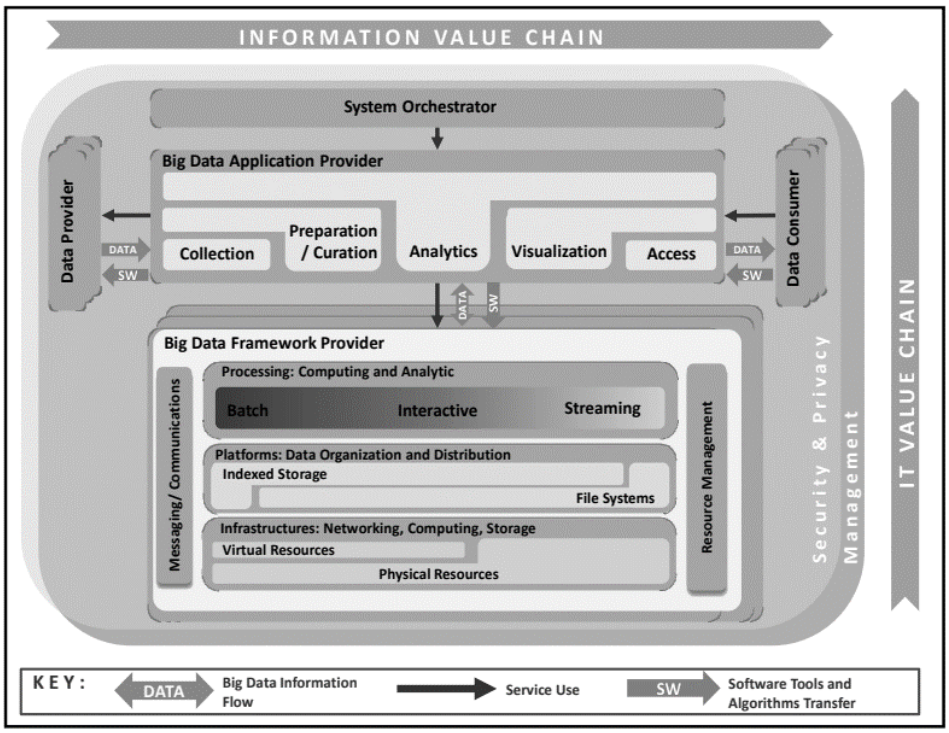

Figura 7 - Arquitetura NITS. Retirada de (Chang, 2015).

O objetivo da NBDRA é permitir o desenvolvimento de soluções que requerem diversas abordagens, mantendo um alto nível de abstração tecnológica por parte dos profissionais de *Big Data*. Esta característica assume-se importante dada a interoperabilidade que um ecossistema *Big Data* permite. Proporciona um framework que suporta uma grande variedade de ambientes analíticos e ainda um conhecimento mais profundo sobre como o ecossistema *Big Data* se complementa e se diferencia de outros sistemas existentes (Chang, 2015).

2.4 **Processamento de dados**

Explicadas as arquiteturas de referência passa a ser pertinente abordar quais as formas do processamento de dados em contexto *Big Data*. Segundo Marz et al. (2015), os sistemas de processamento de *Big Data* devem ter a capacidade de processar vários tipos de dados. Estes sistemas devem permitir a adição de novas funcionalidades e suportar novos tipos de dados que eventualmente possam surgir, não acarretando um aumento elevado no custo de implementação.

De forma a realizar o processamento de dados, verifica-se que existem duas abordagens possíveis, sendo uma a abordagem tradicional em que os dados são processados centralmente e a outra em *Big Data*, onde o processamento é distribuído num conjunto de máquinas.

Existem três tipos de *frameworks* que são capazes de lidar com o processamento distribuído, apresentadas de seguida (Chandio et al., 2015):

- ***Batch*** – é utilizado, no contexto de *Big Data*, para definir um bloco ou conjunto de dados. O processamento é utilizado quando se pretende analisar dados históricos que já se encontram armazenados, sendo entes processados bloco a bloco.
- ***Streaming*** - o processamento por *streaming* permite a computação de fluxos contínuos de dados. Este tipo de processamento é mais complexo, sendo de extrema importância em alguns contextos das organizações em que a necessidade de se ter informação em *real time* é imprescindível.

Tendo em conta que o foco desta dissertação é a avaliação de *frameworks* para processamento de *Streams* em contexto *Big Data*, a próxima secção incide na explanação das características dos dados em *streaming*, assim como nas técnicas de processamento.

2.5 Dados em *streaming*

O processamento de dados em *streaming* permite a obtenção de resultados com tempos de espera reduzidos. Este tipo de processamento veio resolver o problema da elevada latência que os sistemas de processamento de dados em *batch* apresentam. Inclui, também, a maioria das características que os sistemas de processamento em *batch* oferecem, como a tolerância a falhas e a otimização de utilização de recursos.

Ao contrário dos sistemas de processamento em *batch* (SPB), os sistemas de processamento em *streaming* (SPS) realizam o processamento de dados em pequenos períodos de tempo, tendo estes que estar sincronizados com o fluxo de dados.

De forma a melhor compreender o funcionamento dos dados em *streaming* deu-se início ao estudo as suas características.

2.5.1 Características

Aplicações que requerem processamento em tempo real de grandes fluxos de dados estão a levar ao limite as infraestruturas tradicionais. Estas aplicações que incluem, por exemplo, a alimentação continua de dados em mercados financeiros, deteção de fraudes e operações militares são sensíveis ao ponto de um segundo de latência ser inaceitável. Segundo Stonebraker et al., (2005) é possível definir

oito características que um sistema deve exibir de forma a ser eficiente no processamento de *streams* em tempo-real, apresentadas de seguida:

- **Manter os dados em movimento** – de forma a obter baixa latência, um SPS dever ser capaz de processar as mensagens sem ter a necessidade de executar operações de armazenamento durante esse processo. Uma operação de armazenamento adiciona latência desnecessária ao processo e para muitas aplicações de processamento em *streaming*, não é aceitável nem necessário que estas existam.
- **Utilização de SQL (StreamSQL)** – em aplicações de *streaming* é necessário existir algum tipo de mecanismo de *query* para encontrar eventos de interesse ou computar estatísticas em tempo real. Desde o seu aparecimento, estas aplicações são frequentemente desenvolvidas e utilizam linguagens como C+ ou Java. Infelizmente, o uso destas linguagens implica tempos de desenvolvimento e custos de manutenção altos. Desta forma, é desejável processar dados em tempo real utilizando linguagens de mais alto nível como a SQL. Dado que existem atualmente milhões de servidores de bases de dados relacionais em que se utiliza a SQL, faz sentido usar esta linguagem devido à sua familiaridade e apenas estender as suas capacidades para funcionarem com processamento contínuo de fluxos de dados. Surge assim uma variante do SQL, o StreamSQL, que estende a sua semântica e utiliza uma técnica de janela deslizante. Como um *stream* de dados não termina, a técnica da janela deslizante permite que o processador de dados saiba quando deve terminar de processar e passar a apresentar os resultados desse processamento.
- **Resiliência das *streams*** – atrasos nos dados, dados em falta e dados deslocados são consideradas imperfeições que podem interromper sistemas tradicionais. É necessário que existam mecanismos, como timestamps, que assegurem a resiliência dos sistemas de *streaming* quando surgem imperfeições.
- **Gerar resultados previsíveis** – um SPS deve computar conjuntos de mensagens de forma previsível, e assim assegurar que os resultados do processamento sejam determinísticos e replicáveis. A capacidade de produzir resultados previsíveis é importante sob a perspetiva da tolerância a falhas e recuperação, na medida que o reprocessamento do mesmo fluxo de dados deverá originar o mesmo resultado independentemente do tempo de execução.
- **Integrar dados armazenados e em *streaming*** – um SPS deve ser capaz de comparar dados atuais em *streaming* com dados históricos. Esta característica é fundamental na prevenção de fraudes, pois é necessário comparar um acontecimento atual com os hábitos de

compra de um determinado utilizador. É importante também que um SPS consiga alternar entre a computação de dados históricos para dados em *streaming*.

- **Garantir a segurança e disponibilidade dos dados** – de forma a preservar a integridade dos dados e evitar interrupções em aplicações de processamento em tempo real, um SPS deve utilizar uma solução que lhe permita ter grande disponibilidade. Em aplicações em que se utiliza o processamento de dados em tempo real, não é viável reiniciar um sistema e recuperar os dados pois resultará na interrupção do serviço. Desta forma é necessário que exista um sistema de *backup* também em real time, normalmente sincronizado com o sistema primário, que passa a funcionar caso o sistema primário seja comprometido.
- **Particionar e escalar aplicações automaticamente** – atualmente a oferta de *clusters* com boa relação qualidade preço é muito grande, dessa forma deverá ser possível que um SPS consiga escalar utilizando estes *clusters*, sem exigir programação de baixo nível. Os SPS devem também suportar operações de *multi-tread* para tirar partido dos processadores *multi-core*. Para além da escalabilidade em diferentes máquinas, um SPS deve utilizar mecanismos de *load-balance* para evitar a sobrecarga das diferentes máquinas e, consequentemente, evitar *bottlenecks*.
- **Processar e responder instantaneamente** – todas as características apresentadas anteriormente não fazem sentido se uma aplicação de *streaming* de dados não conseguir processar grandes volumes de dados com latências na ordem dos milissegundos e, ainda, serem capazes de o fazer em *hardware* COST (*Commercial Off The Shelf*). Para atingir este alto desempenho os SPS devem reduzir ao máximo o rácio entre tarefas de suporte e tarefas que realizam trabalho útil. Todos os sistemas SPS devem ser desenhados tendo em consideração o seu alto desempenho, e passa a ser imperativo que estes sistemas sejam testados utilizando fluxos de dados que repliquem as condições para o qual este é desenhado.

2.5.2 Técnicas de processamento

O processamento de dados em contexto de SPS traz novos desafios. Tendo em conta que os fluxos de dados em *streaming* são potencialmente ilimitados no seu tamanho, o tamanho necessário de memória para computar também o seria. Embora existam algoritmos para lidar com conjuntos de dados maiores que a memória disponível, estes não são eficazes em aplicações de *streaming* de dados, uma vez que não suportam *queries* continuas e aumentam a latência do sistema (Demchenko et al., 2013).

Nem sempre é possível produzir resultados precisos com as *queries* de *streaming*, quando existe uma limitação na memória. No entanto é possível produzir resultados aproximados e de boa qualidade aceites como resposta adequada. Existem várias abordagens no que diz respeito à aproximação dos resultados, sendo as mais relevantes apresentadas de seguida:

- **Sliding windows -** uma das técnicas utilizada para dar uma resposta aproximada a uma *query* sobre dados em *streaming* é utilizar uma janela deslizante. Esta técnica utiliza apenas a parte mais recente dos dados, sendo possível definir o seu tamanho através de uma medida temporal ou por um número de mensagens. Por exemplo, apenas considerar os dados dos últimos quatro dias ou as últimas cem mil mensagens. A utilização desta técnica como método de aproximação tem como vantagem o facto de ser determinística, assim não existe o perigo dos dados selecionados ao acaso não serem uma boa representação da realidade. Como as aplicações de *streaming* do mundo real dão mais importância aos dados atuais, esta técnica não é vista como uma aproximação, mas como uma resposta exata (Demchenko et al., 2013; Kranen, 2011).
- **Batch Processing and Sampling –** outra classe de técnicas para produzir resultados aproximados passa por abdicar do processamento de todos os dados que entram no sistema. Para o conseguir é preciso utilizar técnicas como a amostragem ou o agrupamento de dados para aumentar a velocidade de execução das queries. Supondo que uma *stream* de dados é continuamente processada utilizando uma estrutura de dados que pode ser mantida de forma incremental, na generalidade, pode concluir-se que essa estrutura de dados suporta duas operações, update() e computeAnswer(). A operação update é invocada para atualizar a estrutura dos dados a cada entrada de uma mensagem e o método computeAnswer produz ou atualiza os resultados da *query*. No processamento de *queries* de forma continua, o melhor cenário é que ambas as operações sejam mais rápidas a ser executadas que o fluxo de entrada dos dados, não sendo necessário técnicas adicionais. Contudo, se alguma das operações descritas anteriormente for lenta, não é possível obter resultados precisos em tempo real. Tendo em conta os dois *bottlenecks* são apresentadas as seguintes soluções:
    - ***Batch processing*** – solução utilizada quando *a operação update* é rápida e a operação *computeAnswer* lenta, a solução passa por processar os dados em *batches.* À medida que as mensagens vão chegando, vão sendo armazenadas em *buffer* e a resposta à *query* é computada periodicamente. Os resultados podem ser considerados aproximados uma vez que não retratam dados atuais, no entanto são considerados resultados precisos uma vez que não são excluídas mensagens.

- ***Sampling*** – quando a operação *update* é lenta e a operação *computeAnswer* rápida, o tempo de execução combinado das duas operações é maior que o tempo médio entre a chegada das várias mensagens. É impossível tentar processar todos os dados, uma vez que estes chegam mais rápido do que podem ser processados. Assim, alguns tuplos devem ser ignorados, para que a *query* consiga executar. É obtida uma resposta aproximada. Em alguns casos, podem ser definidos intervalos de confiança sobre o erro introduzido.

De forma a manipular os dados de um *stream* é necessário utilizar operadores. Estes recolhem dados, processam-nos e de seguida movem os tuplos resultantes para o próximo operador. De certa forma, os operadores são o *core* de um SPS. De seguida são apresentados os operadores mais comuns para construir uma aplicação em *Streaming*.

- ***Filter*** – o objetivo deste operador é remover os tuplos de um fluxo de dados de acordo com as condições definidas pelo utilizador. Depois de definir uma condição para o *filter*, é necessário definir o *output* dos dados. É possível definir dois destinos para os dados, uma para os tuplos que cumprem as condições e outra para tuplos que não cumprem.
- ***Functor*** – este operador lê os dados de um *stream* de entrada, transforma-o de alguma forma e envia os tuplos para um *stream* de saída. A transformação pode manipular todos os elementos de um *stream*.
- ***Sort –*** o operador *sort* tem como *output* os mesmos tuplos que recebe, mas ordenados de uma determinada forma. Este operador recorre à técnica de *sliding window,* apresentada anteriormente, para poder executar. Para além da especificação do tamanho da janela, é necessário especificar a expressão que define como os dados devem ser ordenados.
- ***Join –*** o operador *Join* recebe duas *streams* de dados, combina os tuplos de acordo com as condições especificadas e envia-os para uma *stream* de saída. Este operador recorre também à técnica de *sliding window.*
- ***Aggregate –*** este operador também recorre à técnica de *sliding window* de forma a agrupar os dados para posteriormente os agregar. Este operador permite ainda adicionar parâmetros como o *groupBy* e *partitionBy* de forma a dividir os tuplos e posteriormente fazer operações de agregação nestes subgrupos.
- ***Throttle e delay -*** este par de operadores permite manipular o tempo e o fluxo de um determinado *stream* de dados. O operador *throtle* permite definir o ritmo ao qual os dados fluem

num *stream* de dados, recebendo tuplos esporadicamente e enviando-os a um ritmo definido. O operador *delay* permite enviar os tuplos com um determinado atraso.

- ***Split and Union –*** o operado *split* recebe como entrada um stream de dados e divide em múltiplos *streams*. Este operador utiliza uma lista de valores para um determinado atributo de um tuplo para efetuar uma comparação com os atributos dos tuplos que chegam, divide-os para diferentes fluxos de saída consoante a lista. O operador *union* funciona com o mesmo processo invertido.
- ***Beacon*** - o operador *beacon* permite criar tuplos no momento. Esta característica é importante por exemplo para testes e *debugging* de aplicações de *stream*.
- ***Punctor*** - o operador *punctor* adiciona pontuação a um *stream* de dados, que pode ser usada posteriormente para o separar em diferentes janelas e para auxiliar as funções de agregação.

3. TÉCNICAS E TECNOLOGIAS *BIG DATA*

Neste capítulo serão apresentadas as principais tecnologias *Big Data*, arquiteturas e tecnologias de *streaming*, e ainda as ferramentas de *streaming*.

3.1 Hadoop

Nos últimos anos o uso de SQL, em contexto *Big Data*, voltou a ser uma realidade uma vez que as pessoas estão familiarizadas com os seus benefícios. Esses benefícios incluem a concisão, familiaridade generalizada e a *performance* das técnicas de otimização de *query*.

Contudo, a SQL não é o ideal para realizar tarefas como a limpeza dos dados durante processos de Extract, Transform and Load (ETL), assim como o processamento de eventos complexos, surgindo assim a necessidade de novos modelos.

Surgiu então o Hadoop, criado por Doug Cuttin, que também criou o Apache Lucene. Em Janeiro de 2008, o Hadoop já era considerado um projeto de extrema importância para o Apache, sendo usado pelo Facebook e New York Times. Em Abril do mesmo ano bateu o record de organizar 1 *terabyte* de dados em apenas 209 segundos, sendo que no ano seguinte esse número foi reduzido para 68 segundos (White, 2012).

O Apache Hadoop pode ser descrito como um *framework* que permite o processamento distribuído de grandes conjuntos de dados em *clusters* de computadores, utilizando modelos de programação simples. Foi desenhado de forma a ser escalável, existindo a possibilidade de funcionar numa máquina ou em milhares, oferecendo também processamento e armazenamento local (Hadoop, 2017).

O Framework Hadoop assenta em quatro módulos distintos representados na Figura 8, nomeadamente:

- Haddop Common – Permite o suporte para outros módulos Hadoop.
- Hadoop Distributed File System (HDFS) - Sistema de ficheiros distribuídos que proporciona elevado desempenho no acesso aos dados.
- Hadoop YARN – Framework para agendamento de tarefas e gestão de recursos dos *clusters*.
- Hadoop MapReduce – Sistema que permite o processamento paralelo e distribuído de grandes conjuntos de dados.

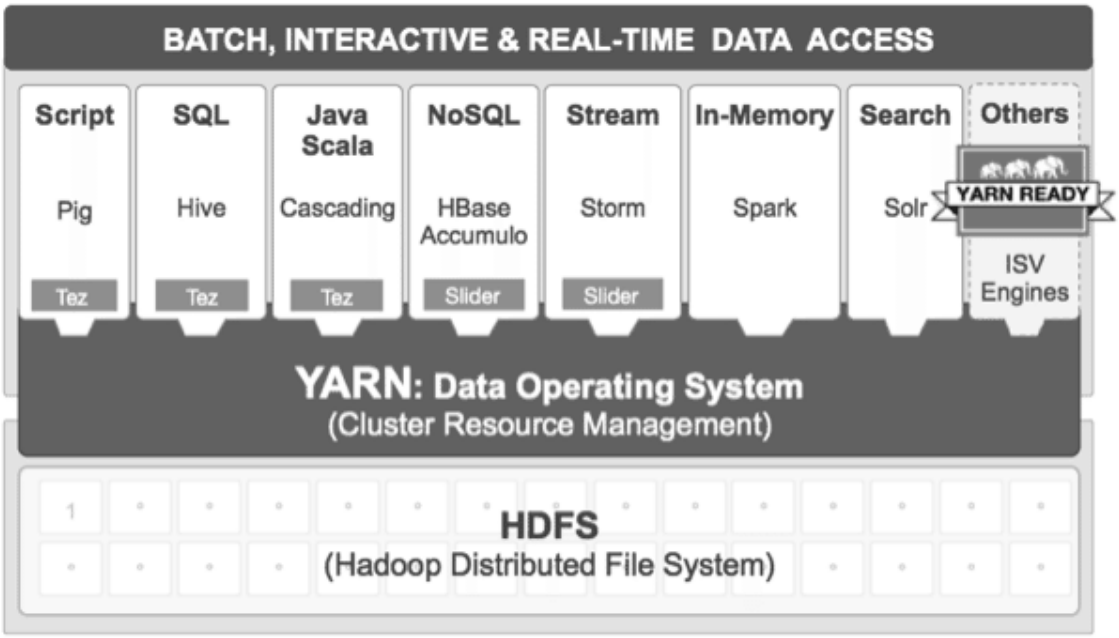

Figura 8 - Ecossistema Hadoop. Retirado de (Hadoop, 2017).

Para além dos quatro módulos referidos o ambiente Hadoop agrega vários projetos que lhe adicionam valor, sendo os mais pertinentes para esta dissertação os seguintes:

- Ambari – Ferramenta web que permite a monitorização e gestão dos vários serviços Hadoop instalados num *cluster*. Permite ainda a visualização do estado e do desempenho do *cluster* de uma forma *user-friendly.*
- Cassandra – Base de dados escalável sem ponto de falha único.
- HBase – Base de dados escalável e distribuída que suporta dados estruturados em tabelas orientadas a colunas.
- Hive – Infraestrutura para a concretização de *datawarehouses* em *Big Data.*
- Pig – Linguagem de alto nível para computação paralela.
- Spark – Motor de computação de dados para Hadoop, disponibiliza uma linguagem de programação simples que suporta um vasto leque de aplicações como ETL, *machine learning*, processamento em *streaming* e computação por grafos.

3.1.1 HDFS

O HDFS ou Hadoop Distributed File System é um sistema de ficheiros distribuído e escalável, desenhado para potenciar o armazenamento de grandes volumes de dados de forma fiável e ainda permitir a sua transferência a grande velocidade.

Um cluster HDFS é composto por dois tipos distintos de nós, um *namenode* e vários *datanodes*. O *namenode* é responsável por manter a árvore de ficheiros e os *metadados*, e ainda por guardar informação sobre os dados que cada *datanode* contém. Os *datanodes* armazenam e providenciam blocos de dados, quando instruídos pelos *namenode*.

Tal como os sistemas de ficheiros tradicionais, o HDFS suporta operações de leitura e escrita assim como a criação e remoção de diretorias, sendo essa interação realizada através de um HDFS *client.* Quando uma aplicação pretende ler um ficheiro, o HDFS *client* questiona o *namenode* quais os datanodes que armazenam réplicas dos blocos dos dados pretendidos sendo posteriormente disponibilizados consoante a topologia da rede e distancia à aplicação (White, 2012).

3.1.2 MapReduce

O MapReduce é um paradigma de programação que permite a escalabilidade de inúmeros servidores (*nós*) num *cluster* Hadoop. O objetivo do MapReduce é dividir o *dataset* a analisar em partes mais pequenas e independentes, para serem processadas de forma paralela e assim reduzir o tempo de processamento. O termo MapReduce refere as duas tarefas distintas que lhe são inerentes, o *map* e o *reduce* (Zikopoulos et al., 2011).

O Hadoop é responsável pela criação de uma tarefa *map* por cada subconjunto de dados, onde é executada a função *map* previamente implementada pelo utilizador. Quando a função *map* termina a sua execução grava os dados no disco local. Os dados que previamente foram computados pela tarefa *map* são os dados de entrada para a tarefa *reduce*, que agrega os dados e no final grava os mesmos no HDFS (White, 2012).

A Figura 9 ilustra o fluxo de dados de um sistema que apenas usa uma tarefa *reduce*. As caixas delimitadas por tracejado representam diferentes os nós de um cluster Hadoop.

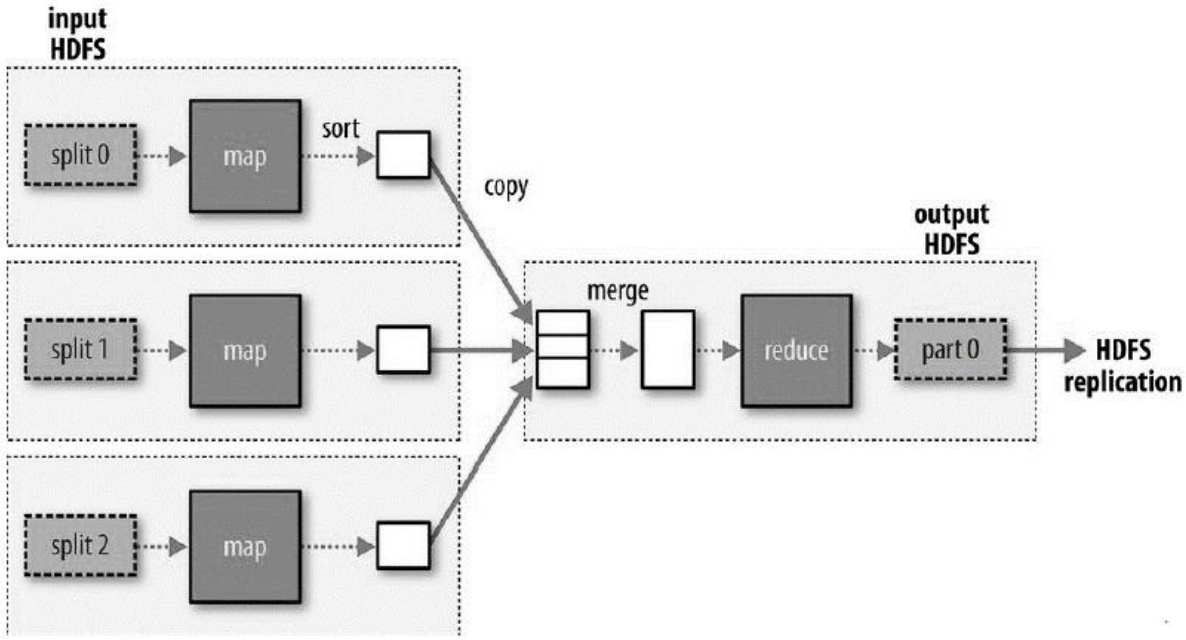

Figura 9 - Fluxo de dados com MapReduce. Retirado de (White, 2012).

3.1.3 YARN

O YARN (Yet Another Resource Negotiator) é um gestor de recursos do Hadoop, sendo introduzido na Hadoop 2 para melhorar o funcionamento do MapReduce, no entanto as suas características genéricas suportam outros paradigmas de programação.

O propósito do YARN é proporcionar APIs para o facilitar o acesso a recursos do *cluster*, no entanto estas APIs não estão concebidas para serem usadas diretamente pelo utilizador. Os utilizadores utilizam *frameworks* de computação distribuída, como por exemplo o MapReduce, que por sua vez utilizam as APIs fornecidas pelo YARN para utilizar os recursos do *cluster* (White, 2012).

3.2 **Bases de Dados NoSQL**

Um dos desafios inerentes a *Big Data* é a escalabilidade, este desafio é transversal às bases de dados relacionais, já que estas têm pouca capacidade para escalar horizontalmente. De forma a ultrapassar esse problema foram criadas as bases de dados NoSQL (Not Only SQL).

As bases de dados NoSQL apresentam as seguintes características:

- Capacidade de escalar horizontalmente através de vários servidores
- Capacidade de replicar e distribuir dados por vários servidores
- Interface de chamadas ao sistema mais simples
- Modelo de concorrência mais débil em relação aos modelos das bases de dados relacionais
- Uso mais eficiente dos índices e da RAM
- Capacidade para adicionar novos atributos a registos já inseridos

Os sistemas NoSQL não disponibilizam transações com propriedades ACID (Atomicity, Consistency, Isolation, Durability), propriedade das bases de dados relacionais. As atualizações são propagadas, mas as garantias na consistência das leituras não esta assegurada. Assim surge o acrónimo BASE (Basically Available, Soft-State and Eventual Consistency) que contrasta com o ACID das bases de dados relacionais (Cattell, 2011).

Com a utilização de sistemas NoSQL surgiu a teorema CAP (Consistency, Availability, Partition Tolerance), que afirma que é impossível um sistema distribuído proporcionar consistência, disponibilidade e partição dos dados ao mesmo tempo, tendo que se optar por apenas duas destas características (He, 2015).

Existem atualmente mais de 225 bases de dados NoSQL,estas são habitualmente divididas em quatro categorias diferentes a apresentar em seguida:

- **Par chave/valor** – estas bases de dados utilizam um conjunto de chaves e os seus respetivos valores para armazenar os dados. No *relational database management system* (RDBM) cada linha contém um número fixo de colunas, neste modelo cada linha é um conjunto de pares chave/valor não necessitando de manter o mesmo número de pares em cada linha (Cattell, 2011; Krishnan, 2013).
- **Documentos** – nestas bases de dados todas as entradas de dados correspondem a documentos. A maioria das soluções orientadas a documentos utiliza como padrão documentos JSON (JavaScript Object Notation) para o armazenamento de dados, mas é possível a utilização de diferentes tipos de documentos dentro da mesma base de dados (Cattell, 2011; Krishnan, 2013).
- **Colunas** – estas bases de dados são uma evolução do modelo chave/valor, acrescentando colunas e famílias de colunas. Providenciam ferramentas de indexação mais eficazes o que melhora o desempenho da consulta de dados (Cattell, 2011; Krishnan, 2013).

- **Grafos** – baseadas na teoria dos grafos, usam o conceito de vértices e ramos para representar a informação. Substituem, em algumas circunstâncias, as RDBMs com os seus dados estruturados e interconectados a pares chave/valor para um ambiente distribuído conseguindo um melhor desempenho. São comumente utilizadas quando a relação entre os dados é mais importante que os dados em si (Khan & Shahzad, 2017; Moniruzzaman & Hossain, 2013).

3.3 Apache Kafka

O objetivo do Apache Kafka é a unificação do processamento online e offline, fornecendo um mecanismo para a ingestão de dados em paralelo nos sistemas Hadoop e ainda a capacidade de particionamento do consumo de dados, em *real-time,* por um *cluster* de máquinas. O Kafka pode ser comparado a nível funcional com o Scribe ou Flume uma vez que é útil para o processamento de *streams* de dados de atividades. Pela perspetiva da sua arquitetura pode ser comparado de forma mais próxima com sistemas de mensagens mais tradicionais como o ActiveMQ ou o RabitMQ.

Segundo Gart, (2015), o Apache Kafka é um sistema mensagens do tipo *publish-subscribe,* distribuído, particionado e replicado tendo como principais características as seguintes:

- **Mensagens persistentes** – a persistência das mensagens deve-se ao armazenamento das mesmas em disco e à sua replicação no cluster, de forma a evitar perda de informação.
- **Alto throughput** – o Kafka foi desenhado para trabalhar em COTS *hardware* e suporta centenas de MB de leitura e escrita por segundo, executado por um grande número de clientes.
- **Distribuído** – o Kafka utiliza um *design* orientado a clusters, suportando particionamento de mensagens e ingestão distribuída por um cluster de *consumers*, mantendo a ordem das mensagens por partição.
- **Suporte multicliente** – o Kafka suporta e proporciona a integração de forma fácil de clientes de variadas plataformas, como Java, .Net, PHP, Ruby e Python.
- **Tempo Real** – as mensagens produzidas pelos *producers* deverão ser visíveis imediatamente pelos *consumers*.

Existe um grande número de casos de uso comuns na utilização do Kafka, aplicados em distintas organizações de renome. Os casos de uso normalmente aplicados são a agregação de *logs*, o *commit* de *logs*, a monitorização de *stream* de cliques, a partilha de mensagens e o processamento de *streams.*

A rede social LinkedIn tem como principal *use case* do Kafka o *streaming*, tanto de dados de atividade como de métricas operacionais. O Twitter utiliza o Kafka em conjunto com o Storm na sua arquitetura de processamento em *streaming*.

3.4 Sistemas de processamento em streaming

Existem atualmente diversos sistemas de processamento em streaming, como por exemplo Apache Apex, Aurora, S4, Storm, Samza, Flink, Samsa, Spark Streaming, IBM InfoSphere Streams. Todos eles diferem no seu *design*, arquitetura, desempenho e características. Novos SPS estão a ainda a ser desenvolvidos, e não sendo possível estudar todos eles no decorrer desta dissertação serão selecionados os mais pertinentes para a sua análise mais detalhada.

Dos SPS existentes apenas serão considerados aqueles que se apresentam como sistemas *open-source*, o que permitirá a sua implementação e posterior execução de testes de desempenho. Foram selecionados Apache Spark Streaming e Apache Flink tendo em conta a sua popularidade na comunidade científica e as suas abordagens distintas no processamento em streaming.

3.4.1 Apache Spark Streaming

O desenvolvimento do Apache Spark iniciou-se na Universidade da Califórnia, escrito nas linguagens de programação Java e Scala. Contém diversas bibliotecas, que incluem o Spark Streaming, para o processamento distribuído em contexto de *streaming.*

As aplicações Spark funcionam como um conjunto independente de processos num cluster, cluster este ilustrado na Figura 10, e são coordenadas pelo *SparkContext* também designado de *driver program.* O *SparkContext* pode conectar-se a diversos *Cluster Managers* (o proprietário Spark, o Mesos ou YARN) sendo o YARN o selecionado no decorrer da dissertação, ficando responsável pela distribuição de recursos para a execução das aplicações Spark.

Os *executors* presentes nos nós do cluster, são os processos que executam processamentos de dados e armazenam os mesmos das aplicações. O *SparkContext* é responsável pelo envio do código da aplicação e das tarefas para os *executors.*

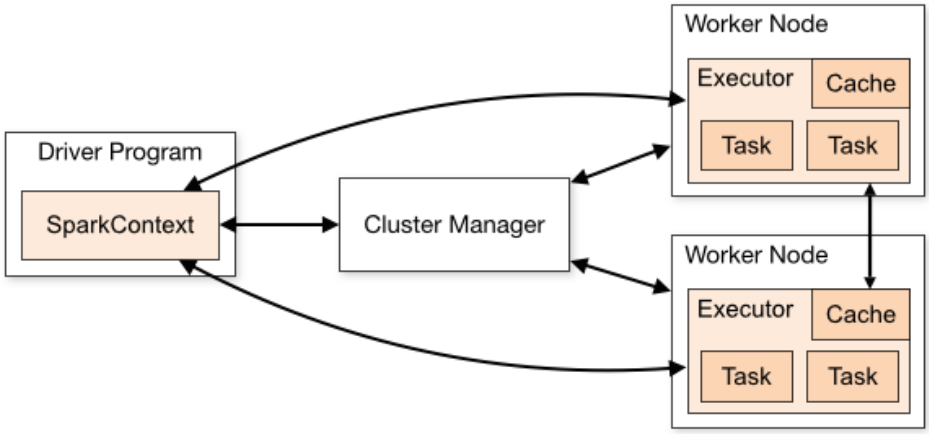

Figura 10 - Ilustração de um cluster Spark. Retirado de (Spark, 2018).

O Spark Streaming utiliza os *Discrete Stream (D-Streams)* como uma abstração para um *stream* de dados, que são definidos como um conjunto de pequenas tarefas determinísticas e independentes entre si.

Quando há entrada de um *stream* de dados no Spark Streaming, é feita a divisão dos dados em *micro-batches,* denominados *Distributed Resilient Dataset* (RDD), e que funcionam como dados de entrada para o *Spark Engine.* Os RDD são armazenados em memoria RAM e posteriormente são processados numa serie de pequenos *batches* computacionais. O processo é ilustrado na Figura 11

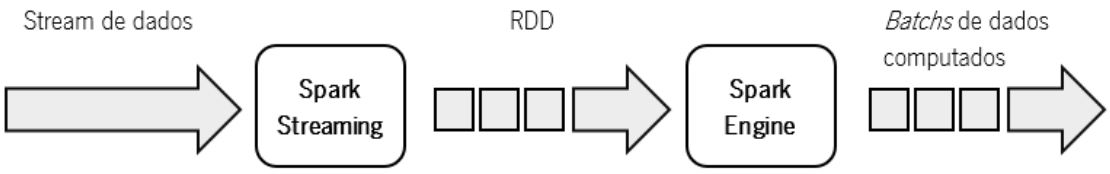

Figura 11 - Processamento do Spark Streaming. Retirado de (Lopez, Lobato, & Duarte, 2016).

Um RDD é capaz de armazenar toda a informação sobre como foi gerado, tornando-se assim tolerante a falhas, uma vez que pode ser reconstruído caso seja necessário. Os RDD suportam dois tipos de operações, transformações e ações. Quando um operador de transformação é invocado é criado um novo RDD com os resultados e o RDD inicial mantem-se imutável. Uma ação retorna um valor para o *driver* no final da execução de uma função sobre o RDD.

3.4.2 Apache Flink

Apache Flink conhecido anteriormente por Stratosphere começou como um projeto *open-source* criado em 2010. Flink é nativamente um SPS onde o processamento em *batch* é representado como um caso especial de *streaming.*

A arquitetura do Flink é do tipo *master-slave*, onde existe um *JobManager* e um ou vários *TaskManagers.* O *JobManager* é o coordenador no sistema Flink e os vários *TaskManagers* executam parte do processamento paralelo. Existe também um *JobClient* responsável por receber uma tarefa, submete-la ao *JobManager*. Quando uma tarefa é terminada, o seu resultado é comunicado ao *JobClient.* Este processo é visível na Figura 12.

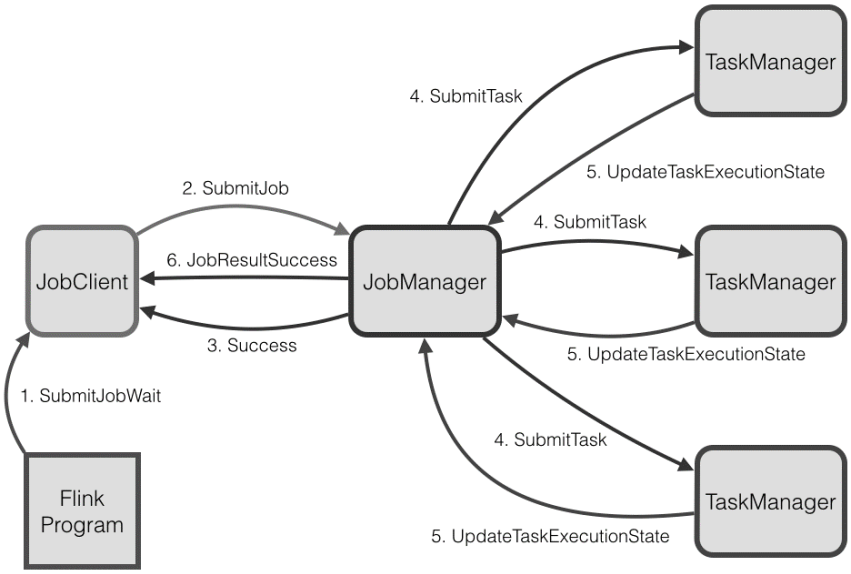

Figura 12 - Esquematização dos processos do Flink. Retirado de (Flink, 2018a).

O Flink utiliza como blocos construtores os *streams* e as *transformations*, combinando-os para criar um *streaming dataflow.* Cada *dataflow* inicia com uma ou mais fonte de dados e termina em um ou mais *sinks.* Os *dataflows* assemelham-se a *directed acyclic graphs* (DAGs). Uma *transformation* pode ser definida com uma ou mais operações efetuadas com os operadores.

O Flink oferece um mecanismo de tolerância a falhas, este mecanismo assegura que na existência de uma falha os dados não são perdidos e o estado do programa permanece o mesmo. É utilizado um algoritmo que armazena continuamente *snapshots* sobre o estado da aplicação, em armazenamento persistente. Quando acontece uma falha, o fluxo de dados distribuído é interrompido,

de seguida os operadores são reiniciados e é feito um restauro do último ponto de controlo (*checkpoint*) bem-sucedido. Os *streams* de entrada são posteriormente redefinidos para o momento em que foi executado o *snapshot* e todos os dados entretando processados têm a garantia não ter sido processados anteriormente.

Os elementos fundamentais no sistema de *snapshots,* representados na Figura 13, são as barreiras inseridas nos streams. Estas barreiras são injetadas no stream de dados e fluem linearmente com o mesmo, são usadas para separar os registos e definem um *snapshot.* Cada barreira é composta pelo ID do *snapshot,* estas barreiras não interrompem o fluxo do stream dado o seu pequeno tamanho.

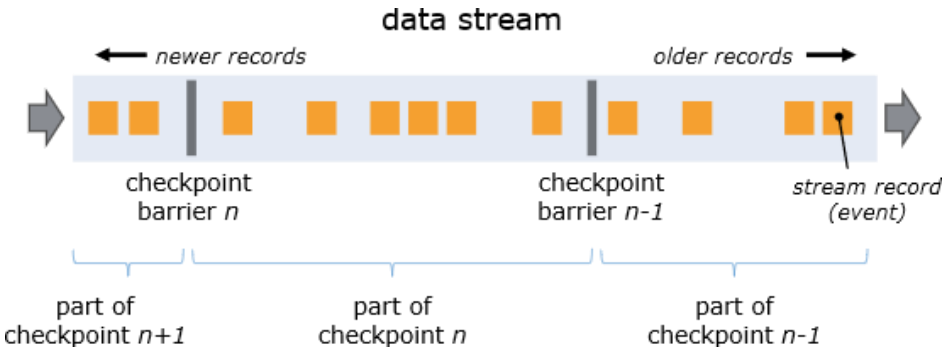

Figura 13 - Ilustração do sistema de Checkpointing presente no Flink. Retirado de (Flink, 2018).

4. Caracterização do Benchmark e do Ambiente de Testes

Esta secção pretende caracterizar o benchmark aplicado às frameworks de streaming selecionadas, de forma a ser um processo sistemático e replicável. É apresentada a infraestrutura tecnológica, o processo o protocolo de testes, indicadores e métricas utilizados.

4.1 Infraestrutura

Para a realização desta dissertação foi configurado um cluster com 5 máquinas na plataforma Google Cloud Platform. Todas as máquinas foram criadas com 32GB de RAM, um vCPU (virtual CPU) de 4 núcleos (intel xenon de 2.2 Ghz), um HDD (Hard Disk Drive) de 256 GB e uma interface Ethernet 10 Gigabit. Nas 4 máquinas destinadas a *slaves* foi adicionado um *Solid-State Drive* (SSD) de 100 GB para um melhor desempenho no acesso e leitura de dados.

Foi instalado o CentOS 7.5.1804 Minimal como sistema operativo em todas as máquinas e configurada uma rede interna para a comunicação entre as mesmas. A largura de banda obtida entre as máquinas foi de 7.8 Gbits/s ou 975 MB/s.

Foi utilizada a distribuição Cloudera Express 5.14.3 de forma a facilitar o acesso ao ecossistema Hadoop. Os diferentes *frameworks* usados foram atualizados para a última versão disponível à data de 1 de Maio de 2018. No caso das *frameworks* não disponibilizadas pela distribuição Cloudera, foi feita a instalação e configuração das mesmas nos diferentes nós segundo a sua documentação. Os *frameworks* utilizados e a sua versão podem ser consultados na Tabela 1

Os diferentes clusters tecnológicos resultantes da instalação das diferentes tecnologias têm as versões descritas na Tabela 1.

Tabela 1 - Frameworks e as suas versões

Apache Spark	2.3.0
Apache Flink	1.5.0
Apache Kafka	1.1.0
Apache ZooKeeper	3.4.5
Hadoop	2.6.0

Como é possível observar na Figura 14, existem dois *clusters* tecnológicos, o *cluster* resultante da instalação do Apache Spark e o *cluster* resultante da instalação do Apache Flink, assentes num *cluster* de 5 máquinas. O *cluster* Spark é constituído por 4 *worker* e uma instância que funciona como

History Server e Gateway. O History Server permite consultar os resultados em tempo real das aplicações Spark e o Gateway permite submeter as aplicações

O cluster Flink é composto por 4 *workers* e por 1 master, sendo que um dos *workers* disponibiliza também uma interface de utilização (UI), onde é possível submeter aplicações e ainda acompanhar em tempo estado das mesmas e obter resultados.

O kafka e o Zookeaper estão instalados no *Master* uma vez que é a máquina do cluster com mais recursos computacionais.

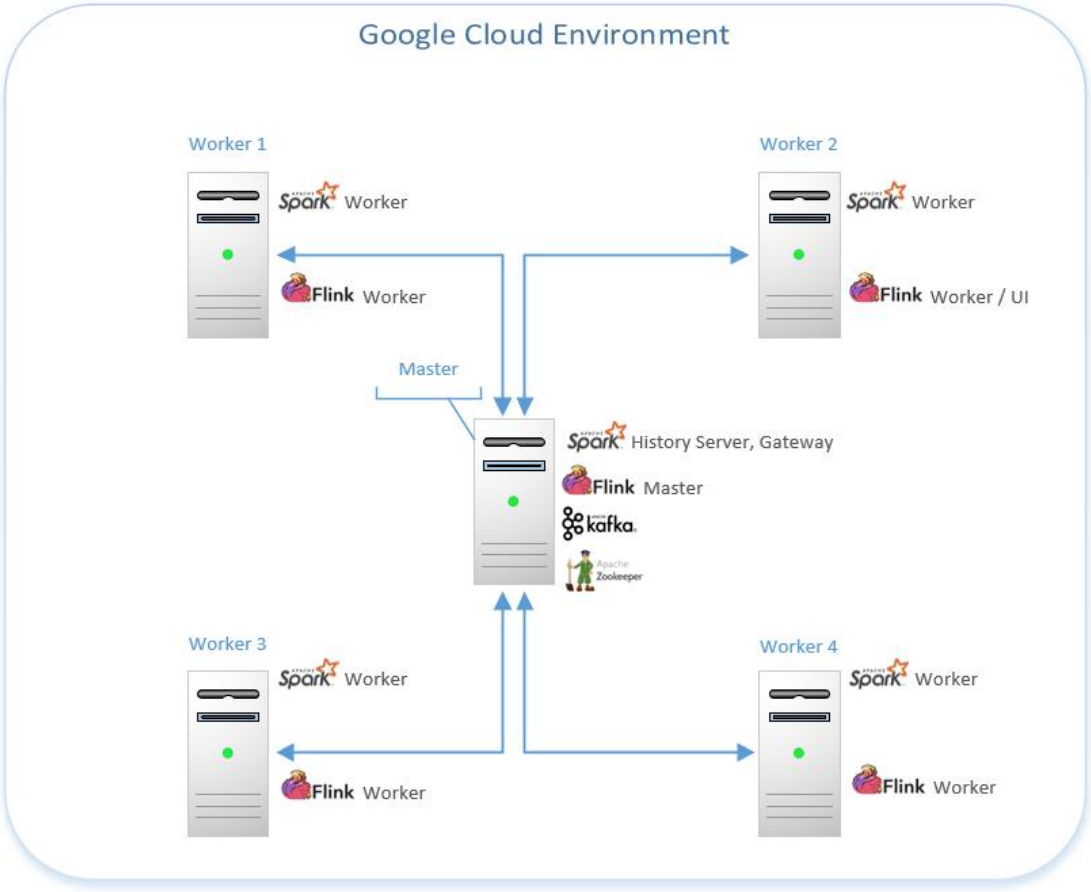

Figura 14 - Infraestrutura tecnológica

4.2 Definição do protocolo de testes

Para o *benchmark* de ferramentas de *streaming* é importante salvaguardar a consistência dos testes, assim como ter controlo total sobre a especificidade dos mesmos. Assim, foi definido que os *streams* de dados que alimentariam o *benchmark* seriam completamente replicáveis e passíveis de sofrer modificações.

Foram criados dois *producers* em Java que permitem simular dois *streams* de dados. Desta forma é possível controlar em detalhe o fluxo de dados, a cadência das mensagens, o seu tamanho, o número de atributos que contêm e o seu tipo. Assim, é possível garantir a consistência dos testes e a sua variedade. O *stream* 1 resultante do *producer 1*, e cujo o *output* pode ser consultado na Figura 15, é composto por uma mensagem padrão que contém quatro atributos, id, id de comentário, geocoordenada e *timestamp*.

```
111070;1;-89.56861103579614,0.23937893202302363;1541542753016
111071;2;-89.63985275305672,0.5036381099507059;1541542753016
111072;3;-89.0861833399809,0.2522886720793662;1541542753016
111073;4;-89.20947105956074,0.1837188512825525;1541542753016
111074;5;-89.30195672513508,0.4140318543187115;1541542753016
```

Figura 15 - Exemplo de mensagens produzidas pelo producer 1.

O *producer 2* é responsável pela criação do *stream* 2, e cujo o *output* pode ser consultado na Figura 16, é composto por dois atributos, id e comentário.

```
1;spoil squeeze fun also appear present tuna snow sketch fury venture quiz evidence engine appear decide alert evoke link unable effort maze please camp
2;feed ball milk crater gold meat sadness talk bottom rack pitch casino found hawk rude side note side exhaust vintage business tired bitter twin aim an
3;canoe brass lens amount close adjust beach scale artefact nut ready enemy vanish rotate ladder crater squirrel learn accuse creek balance kick silent
4;toddler argue once gown mammal idle burger insect animal defense duty tobacco purse flock merge aunt salmon topple sphere stable pool ridge thought ab
5;shed rather cost ivory summer clutch ready bicycle switch staff never lawn lawn scheme rain alien wire isolate exchange hurry oil renew lobster keen a
```

Figura 16 - Exemplo de mensagens produzidas pelo producer 2.

Foi utilizado um broker Kafka devido à sua utilização massificada pela indústria e pelas características apresentadas no capítulo 3. Este *broker* recebe as mensagens num tópico com 20 partições e alimenta as *frameworks* de processamento em *streaming*.

Foram desenvolvidos 2 *consumers*, um para cada *framework* de *streaming*, utilizando a linguagem de programação JAVA e o conector Kafka adequado para cada versão do framework. A Figura 17 esquematiza esta arquitetura.

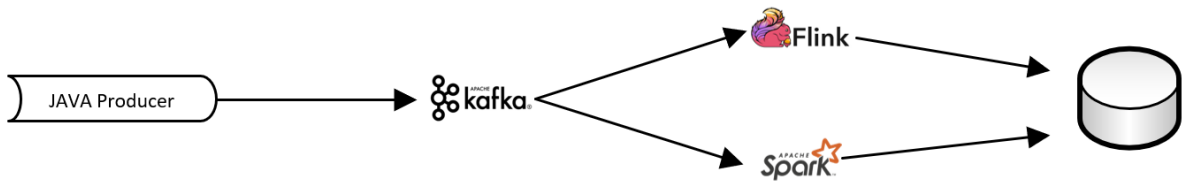

Figura 17 - Ilustração da arquitetura de streaming.

Foi definido um problema prático cuja resolução eficaz necessita da utilização de *frameworks* de *streaming*. Dados dois fluxos constantes de mensagens, definidos anteriormente, quantas mensagens ocorriam, dentro de um espaço geográfico e em tempo real. Este problema foi definido desta forma para que fosse possível utilizar um grande número de operadores de *streaming*.

São gerados dois *streams* de dados que sofrem um *join* por id - id de comentário. Posteriormente o *stream* resultante é filtrado utilizando o operador *filter*. O operador filter verifica se determinada mensagem, que contém o atributo geocoordenada, pertence a uma área geográfica. Todas as mensagens que cumprirem esse requisito formam um novo *stream*.

De seguida é executado o operador *map* às mensagens e realizada uma operação de *reduce,* de forma a obter a latência média e o número de mensagens processados num intervalo de tempo.

Este processo ilustrada-se na Figura 17:

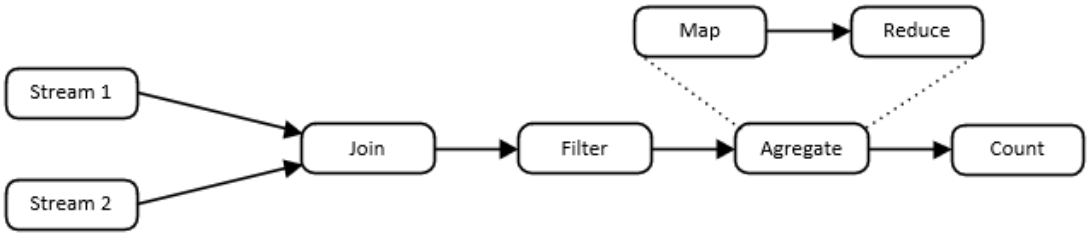

Figura 18 - Processamento e fluxo de mensagens.

De seguida, é explicado em detalhe o funcionamento dos operadores utilizados nos *consumers* desenvolvidos para ambas ferramentas de *streaming*.

4.2.1 Window Join

O modelo comumente usado na realizaçã0o de *joins* de dados é impraticável em ambiente de streaming. Um stream de dados pode ser definido como tendo um tamanho infinito, o que leva a uma utilização de memória infinita na utilização de *joins*, sendo necessário recorrer a *Window Joins*

Um *Window Join* pode seguir um modelo baseado no tempo ou em tuplos. O modelo baseado em tuplos dita que a operação *join* deve acontecer num intervalo de n tuplos definido pelo utilizador. Já o modelo orientado ao tempo, utilizado neste *benchmark*, define uma janela temporal dentro da qual acontecem as operações de *join*. Nos dois modelos é feita a exclusão dos dados assim que estes deixam de fazer parte do intervalo definido pelo utilizador, de forma a libertar espaço de memória e evitar *joins* com os mesmos.

O Window Join utilizado neste benchmark utiliza dois *streams* criados com pares chave/valor e uma janela deslizante com o mesmo valor temporal para os dois *streams*, esquematizado na Figura 19. O Stream A é gerado *on-running* pelo *producer 1* e é composto pelos atributos id, id de comentário, geocoordenada e *timestamp*. O *Stream* B é gerado da mesma forma e é composto por id e comentário. Cada tuplo do *Stream* A é um par de chave/valor (id, v1) sendo o id a chave primária, que é referenciada pela chave estrangeira id dos tuplos do Stream B (id, v2). O resultado do operador *join* é um *stream* de tuplos (id, v1, v2) ou seja, é composto pelos atributos *id*, comentário, geocoordenada e *timestamp*.

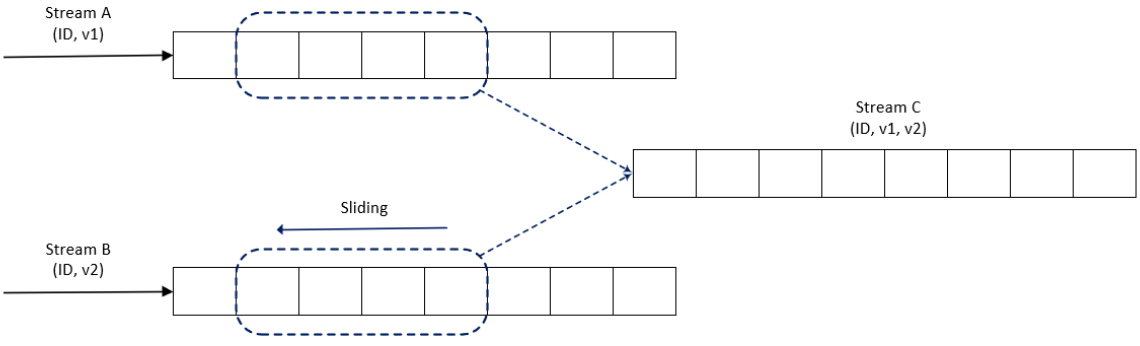

Figura 19 - Esquematização do Window Join.

O *join* executado no Spark streaming utiliza um mecanismo semelhante à janela deslizante, sendo que a janela deslizante é o próprio *micro-batch* para executar o *join*. Para melhorar a avaliação do Flink foram usados dois métodos para executar operações de *join*, o Sliding Window Join e o Tumbling Window Join. Ambos simulam o funcionamento do *join* utilizado pelo Spark se forem utilizados os valores adequados para o tamanho da janela e para o tempo de deslizamento.

4.2.2 Filter

Este operador permite filtrar as mensagens de um *stream*, dividindo o mesmo em um ou vários *streams* de acordo com as condições definidas. Neste caso, o operador *filter* foi utilizado para verificar a existência de mensagens criadas com uma latitude inferior a 40º Norte.

4.2.3 Agregate

Os dados sofrem uma agregação por forma a ser possível obter os resultados do desempenho das ferramentas de streaming no decorrer da mesma. É feita uma operação *map* a todas as mensagens em que adiciona uma constante de valor 1 e a latência até ao momento, de seguida é feita a sua agregação de forma a obter o número de mensagens e a latência num determinado intervalo de tempo.

4.3 **Indicadores**

Para a elaboração do *benchmark* serão alteradas uma série de variáveis de forma a compreender o comportamento de cada ferramenta de *streaming*. Assim, será definido um cenário base de testes e posteriormente alterada cada variável individualmente para um dos dois valores possíveis.

4.3.1 Tamanho da mensagem

A estratégia de variação do tamanho das mensagens consiste em aumentar o tamanho do comentário do *Stream* 2. De forma a manter a consistência dos testes, são pré computados 50 comentários com três tamanhos diferentes, 150, 512 e 1024 bytes. Os comentários são gerados de forma aleatória e utilizam palavras do dicionário bip39. O valor base selecionado foi 150 bytes.

4.3.2 Tamanho da janela

O tamanho da janela deslizante utilizada no Window Join entre os dois *streams* é alterado nos *consumers* para cada ferramenta de *streaming*. É importante referir que o tamanho mínimo da janela é limitado pelo tamanho do *micro-batch* do Spark Streaming. Assim sendo foram definidos os tamanhos 10s-10s, 20s-20s e 30s-30s segundos. Sendo o tamanho 10s-10s o selecionado como valor base.

4.3.3 Tamanho do micro-batch

O tamanho do *micro-batch* é apenas alterado no Spark streaming. O valor base selecionado para o *micro-batch* é 10 segundos, e os outros testes utilizam o valor de 5 e 2 segundos. Estes valores foram selecionados tendo em conta o contexto de *streaming* desta dissertação que inclui a necessidade de resultados em *real-time*.

4.3.4 Paralelismo

Tendo em conta a forma como as diferentes ferramentas de *streaming* fazem a gestão dos recursos, foram definidos diferentes valores para cada uma delas. O Spark streaming utiliza recursos dinâmicos, que lhe são atribuídos pelo YARN, consoante o *workload* de uma aplicação. Depois da obtenção do máximo de *executors* que o YARN fornece, a caraterística *dynamic allocation* foi desabilitada para que existisse uma melhor consistência dos testes. Assim, foi medido o número máximo de *executors* que o Yarn disponibiliza ao Spark Streaming, sendo feita a atribuição de 100% a este valor, valor este que é equivalente à utilização de 5 nós por parte do Flink. No seguimento desta estratégia, 60% do número máximo de *executors* corresponde a 3 nós do *cluster* e 20% corresponde a 1 nó. O valor padrão selecionado foi de 1 nó que corresponde a 20%.

4.3.5 Número de mensagens produzidas

A variação do número de mensagens produzidas é realizada no *producer* 1 que dá origem ao *stream* 1. A configuração padrão permite produzir 100 milhões de mensagens. O produtor que dá origem ao *stream* 2 tem uma cadência de cinco mensagens por segundo, volume insignificante quando comparado com o volume de mensagens geradas pelo *producer 1* e, portanto, não se consideram quando se analisa o número de mensagens produzidas.

4.3.6 Número de mensagens por segundo

O número de mensagens por segundo é definido pela combinação dos dois *producers*. O stream 2 é limitado a 5 mensagens por segundo. O *stream* 1 é produzido de forma a que o número de mensagens seja superior ao que as ferramentas de *streaming* conseguem processar num determinado cenário. Desta forma garante-se que o *bottleneck* são as ferramentas de *streaming* e não o *producer*. Sempre que os testes executados apresentavam valores próximos do máximo do *throughput* do *stream* 2, eram repetidos com a adição de outro *producer 2* de forma aumentar o *throughput do stream 1*.

4.4 Métricas

Nesta secção são referidas as várias métricas que serão usadas para avaliar a performance das ferramentas de *streaming*.

4.4.1 Throughput

O *throughput* é obtido através do rácio do total de mensagens processadas e do tempo total de processamento. O número total de mensagens processadas é obtido através da operação de agregação das mensagens que é executado depois do *join*. Este valor pode ser diferente do rácio entre o número de mensagens ingeridas e o tempo de execução devido à operação de *join* entre *streams*. Os valores para esta métrica serão apresentados em mensagens por segundo (mps).

4.4.2 Tempo de processamento

É medido o tempo de processamento de um conjunto de mensagens definido no *producer* 1. É ainda medido o tempo de produção das mensagens para posterior análise. Os valores para esta métrica serão apresentados em segundos.

4.4.3 Latência

O Flink permite medir de forma nativa a latência das mensagens que percorrem o sistema. É definido nas aplicações do Flink um intervalo no qual são gerados periodicamente registos especiais, denominados *Latency Markers*. Estes marcadores contêm o *timestamp* de quando foram emitidos nas diferentes fontes. Os *Latency Markers* acompanham o fluxo de mensagens e não as ultrapassam, assim, na existência de filas de espera a latência também é adicionada ao marcador. No entanto, estes registos não contabilizam o seu tempo do processamento, uma vez que não sofrem processamento.

De forma a uniformizar o processo de recolha da latência foi implementado um procedimento semelhante na própria arquitetura dos *consumers,* para ambas as ferramentas de *streaming*. Aquando da ingestão das mensagens é adicionado um *timestamp* a cada uma delas e no final das operações de transformação é feita uma subtração do *timestamp* daquele instante pelo *timestamp* criado inicialmente. Com este valor é possível medir a latência introduzida pelas diversas operações.

Os valores para esta métrica serão apresentados em milissegundos.

4.5 Cenários

Para efetuar a comparação de desempenho das diferentes ferramentas de *streaming* definiu-se um conjunto de cinco cenários. Estes permitirão identificar não só a diferença de desempenho entre as várias ferramentas, mas também o seu comportamento com diferentes especificidades.

Todos os cenários criados são a combinação dos valores dos primeiros 5 indicadores apresentados anteriormente. Foi criado um cenário base que serve como meio de comparação com os outros cenários, criado através dos valores base de cada indicador.

Foi criado um cenário por indicador, de forma a se perceber o impacto que os distintos valores de um determinado indicador impõem nas ferramentas de *streaming.* O indicador "Número de mensagens por segundo" não gerou um cenário, uma vez que é utilizado de forma dinâmica e o seu valor é mantido acima da capacidade de processamento das ferramentas de *streaming*. Cada um destes cenários contém 3 valores diferentes para o indicador em estudo e todos os outros indicadores são mantidos com o valor base. No Cenário 1 pretende-se avaliar o impacto que o tamanho da mensagem tem no desempenho das ferramentas de streaming. Assim, são definidos dois valores distintos do valor base para o tamanho das mensagens, sendo eles 256 e 1024 bytes. Os valores dos restantes indicadores não são alterados.

Os vários cenários criados podem ser consultados na Tabela 2.

Tabela 2 - Definição dos cenários

	Tamanho da mensagem	**Tamanho da janela**	**Tamanho do *micro-batch***	**Paralelismo**	**Número mensagens produzidas**
Base	150 bytes	10s-10s	10 segundos	20%	100 Milhões
Cenário 1	150 bytes 256 bytes 1024 bytes	10s-10s	10 segundos	20%	100 Milhões
Cenário 2	150 bytes	10s-10s 20s-20s 30s-30s	10 segundos	20%	100 Milhões
Cenário 3	150 bytes	10s-10s	10 segundos 5 segundos 2 segundos	20%	100 Milhões
Cenário 4	150 bytes	10s-10s	10 segundos	20% / 1nó 60% / 3 nós 100% / 5 nós	100 Milhões

5. Apresentação de Resultados

Um teste é caracterizado pela execução de uma aplicação de streaming, com os parâmetros definidos num dos cenários. Os testes são repetidos três vezes para cada combinação de parâmetros e os resultados apresentados são a média dos mesmos.

Para a aumentar a coerência dos resultados, os testes são executados com apenas uma das ferramentas de *streaming* iniciada, maximizando também os recursos do cluster e consequentemente aumentando o desempenho das aplicações de streaming. Entre cada teste são recriados os dois tópicos utilizados pelo Kafka, para evitar problemas de acumulação de *logs*. Quando existe a necessidade de utilizar outra ferramenta de *Streaming*, é reiniciado o *cluster* e apenas iniciada a ferramenta pretendida e as suas dependências.

5.1 Cenário Base

Este cenário serve como *baseline* de comparação entre todos os outros cenários definidos e foi criado através da combinação dos valores predefinidos dos diferentes indicadores apresentados anteriormente.

Foram executados os testes nas diferentes ferramentas de streaming com as configurações definidas pelo cenário e os resultados podem ser consultados na Tabela 3.

Tabela 3 - Resultados do cenário base

Métricas	Spark Streaming	Flink
Tempo de processamento	628 s	710 s
Latência	4.078 ms	6.246 ms
Throughput	162.983 mps	140.798 mps

5.2 Cenário 1

O cenário 1 permite perceber o impacto do tamanho das mensagens nas ferramentas de *streaming*. Assim sendo, são utilizados três tamanhos de mensagens diferentes, sendo avaliado o desempenho das ferramentas recorrendo às métricas apresentadas previamente.

No Gráfico 1 é possível visualizar o impacto que a variação do tamanho das mensagens impõe no processamento do fluxo de dados. Pode-se observar que quanto maior o tamanho da mensagem maior o tempo de processamento, no entanto o impacto do aumento do tamanho das mensagens é em média

inferior a 4% em ambas as ferramentas. O Fink apresentou um tempo de processamento médio superior ao Spark Streaming de 15%.

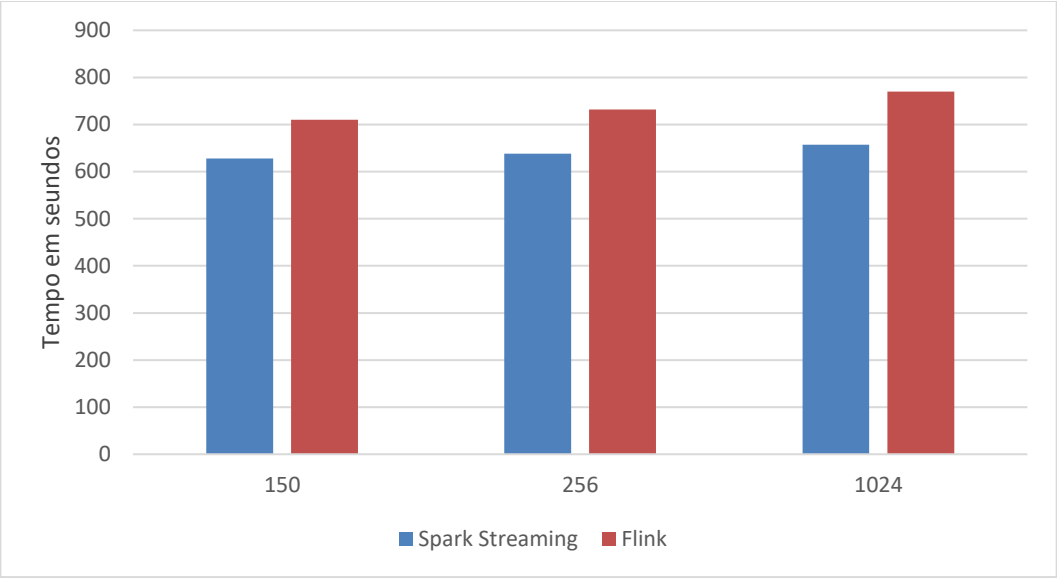

Gráfico 1 - Tempo de processamento médio do cenário 1

Pode-se observar, a partir do Gráfico 2, que existe um aumento inferior a 5% do valor da latência conforme aumenta o tamanho da mensagem. É possível também constatar que existe uma superioridade do Spark Streaming, apresentando um valor de latência médio 33% mais baixo que o Flink.

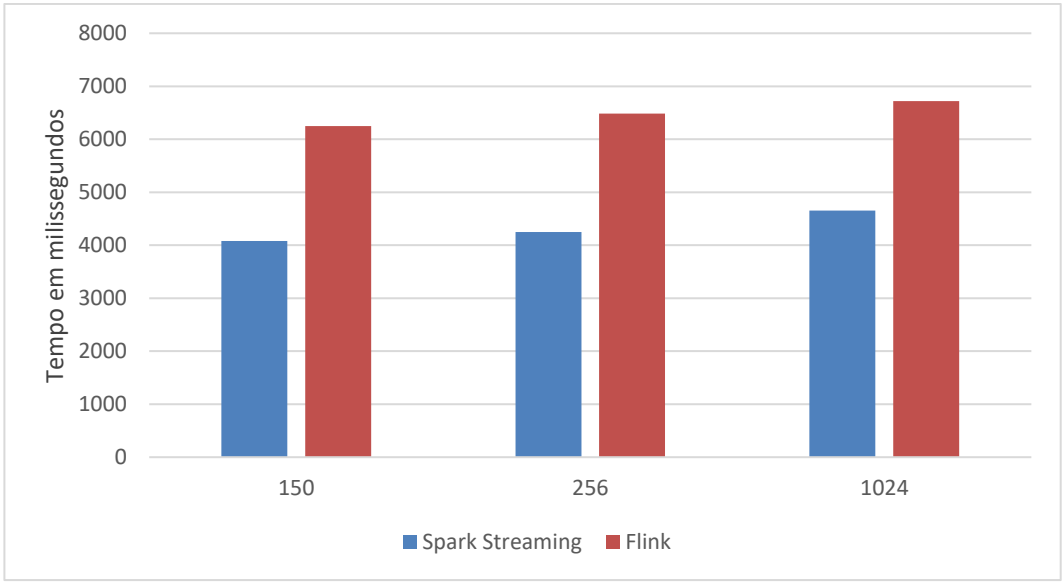

Gráfico 2 - Latência média do cenário 1

O *throughput* médio pode ser consultado no Gráfico 3, onde se constata que existe um decréscimo médio de 4% com o aumento do tamanho das mensagens. O Flink apresenta um em média um valor 16% mais baixo para todos os tamanhos de mensagens.

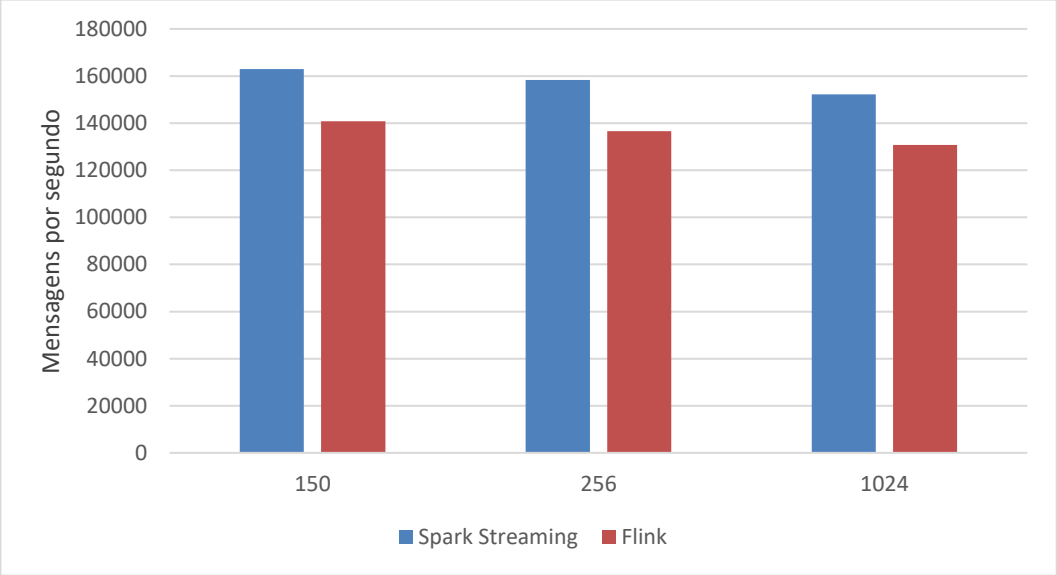

Gráfico 3 -Troughput médio do cenário 1

5.3 Cenário 2

O intuito deste cenário é permitir a perceção do impacto do tamanho da janela temporal (deslizante) da operação *join* usada neste benchmark.

Para a elaboração dos testes deste cenário são mantidos todos os valores dos indicadores, à exceção do tamanho da janela da operação de *join*. Para o Flink foram ainda testadas duas operações de join, o Sliding Window Join e o Tumbling Window Join. Os resultados presentes na Tabela 4 demonstram que existe uma diferença inferior a 1% no desempenho entre eles. Ainda assim foi utilizado o Sliding Window Join devido ao seu desempenho ligeiramente superior.

Tabela 4 - Diferentes operações de join no Flink

Join	Sliding Window Join	Tumbling Window Join
Tempo de processamento	710 s	711 s
Latência	6.246 ms	6.311 ms
Throughput	140.798 mps	140.646 mps

No Gráfico 4 verifica-se que o tempo de processamento aumenta conforme o aumento do tamanho da janela e que é muito mais significativo no contexto 30sx30s.

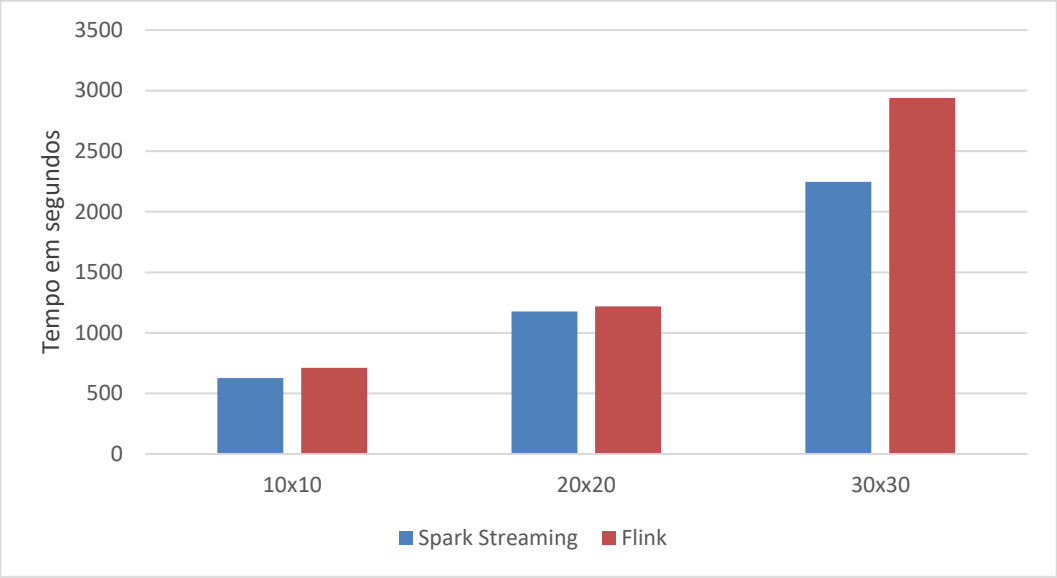

Gráfico 4 - Tempo de processamento médio no cenário 2

É possível observar no Gráfico 5 que a latência aumenta com o tamanho da janela. No caso do Spark Streaming esta latência aproxima-se de metade do valor do intervalo temporal da janela definido. Como exemplo, quando o valor do tamanho da janela é 30x30 a latência do Spark Streaming aproxima-se de 15 segundos. Já no caso do Flink, a latência sofre um aumento não linear conforme o aumento do tamanho da janela.

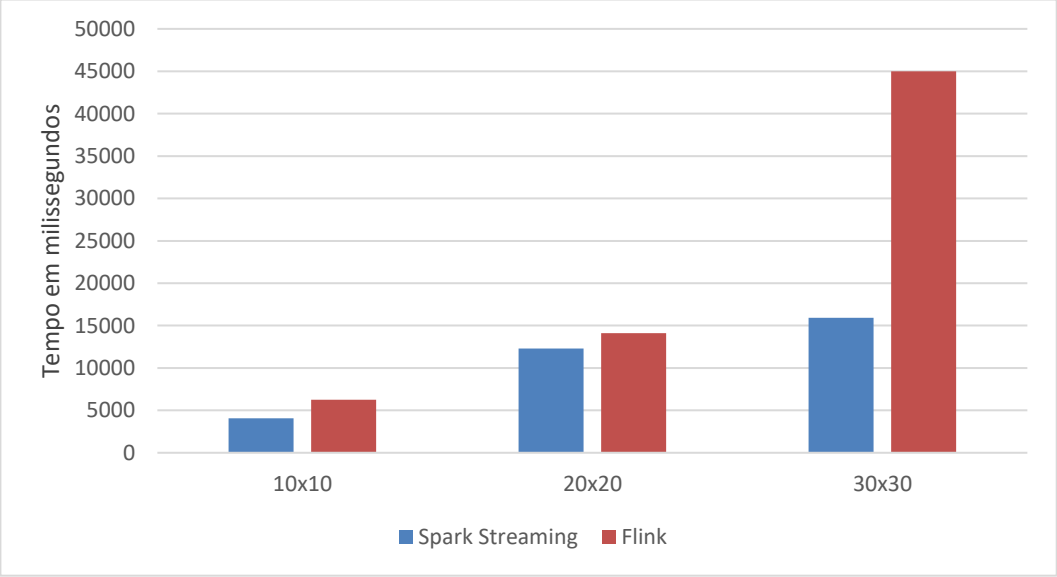

Gráfico 5 - Latência média no cenário 2

Neste cenário, constata-se que o *throughput* não é inversamente proporcional ao tempo de processamento. Estes resultados divergentes devem-se ao facto de se criarem novas mensagens na realização da operação *join* entre os dois *streams*. Quando o intervalo da janela deslizante do *join* existente é do tipo "10s-10s", cada mensagem ingerida produz apenas uma mensagem, resultando assim num rácio de 1:1. Num *join* do tipo "20s-20s" existe um rácio de 1:2, ou seja, por cada mensagem ingerida o *join* dá origem a duas mensagens. De modo idêntico um *join* do tipo "30s-30s" tem um rácio de 1:3 e gera o triplo das mensagens ingeridas.

Como se verifica no Gráfico 6, existe um decréscimo de desempenho quando o tamanho da janela é de 30 segundos. Este decréscimo é justificado pelo grande número de mensagens criado, sendo que as ferramentas de *streaming* não as conseguem processar em tempo útil.

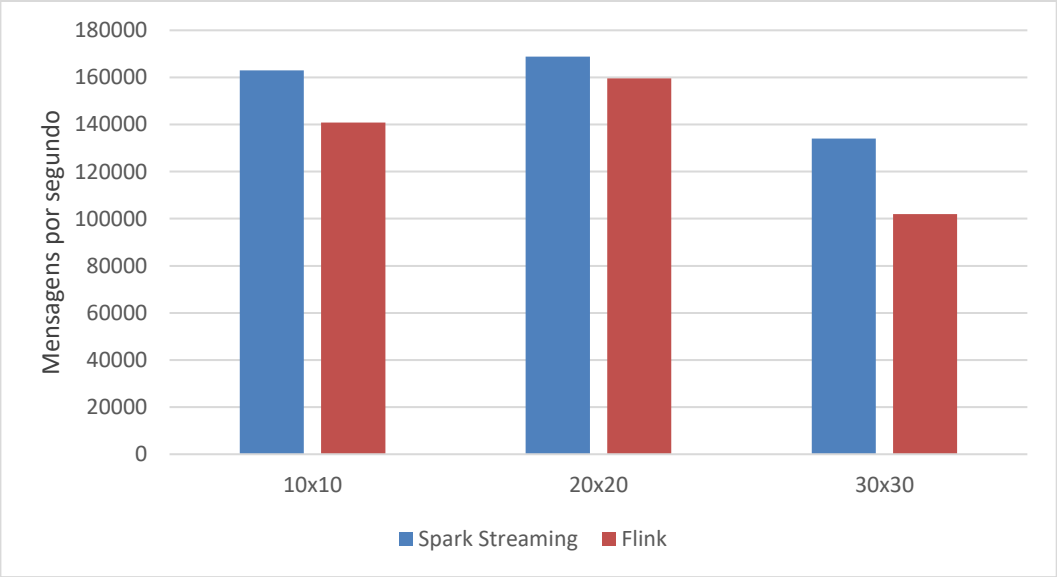

Gráfico 6 - Throughput médio no cenário 2

5.4 Cenário 3

O intuito deste cenário é perceber de que forma o tamanho do *micro-batch* influencia o desempenho da aplicação de processamento de *streams* desenvolvida. Como apenas o Spark Streaming utiliza esta abstração para simular um fluxo continuo de dados (já que o Flink processa mensagem a mensagem), os valores obtidos serão comparados com os valores do cenário base do Flink.

Como se pode observar no Gráfico 7, o tempo de processamento do Spark Streaming aumenta conforme diminui o tamanho do *micro-batch* de forma não proporcional. Quando se reduz o tamanho do microbatch de 10 segundo para 5 segundos há um amento de apenas de 21% no tempo de processamento, no entanto quando se passa de um microbatch de 5 segundos para um de 2 segundos

há um aumento de 75% no tempo de processamento. Verifica-se também que a superioridade do Spark Streaming sobre o Flink diminui quando o *micro-batch* se aproxima de um valor baixo, valor necessário para a obtenção de um sistema em *real-time.*

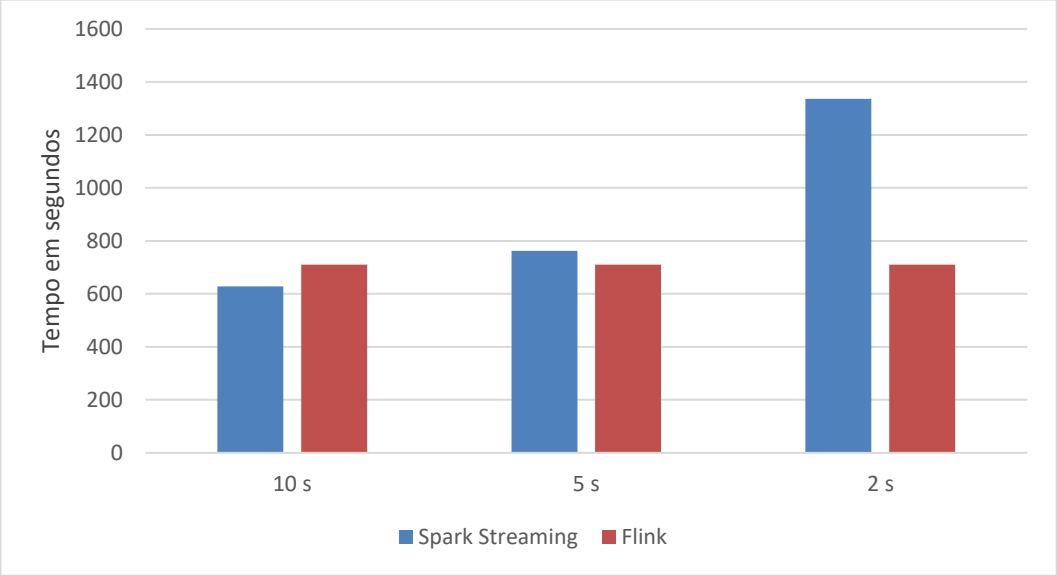

Gráfico 7 - Tempo de processamento médio no cenário 3

Em relação à latência é possível perceber, através do Gráfico 8, que esta está diretamente relacionada com o tamanho do *micro-batch,* no Spark Streaming. O seu valor corresponde a 41% do tamanho do micro batch quando o *micro-batch* é de 10 segundos, 47% quando o micro-batch é de 5 segundos e 55% quando o *micro-batch* é de 2 segundos.

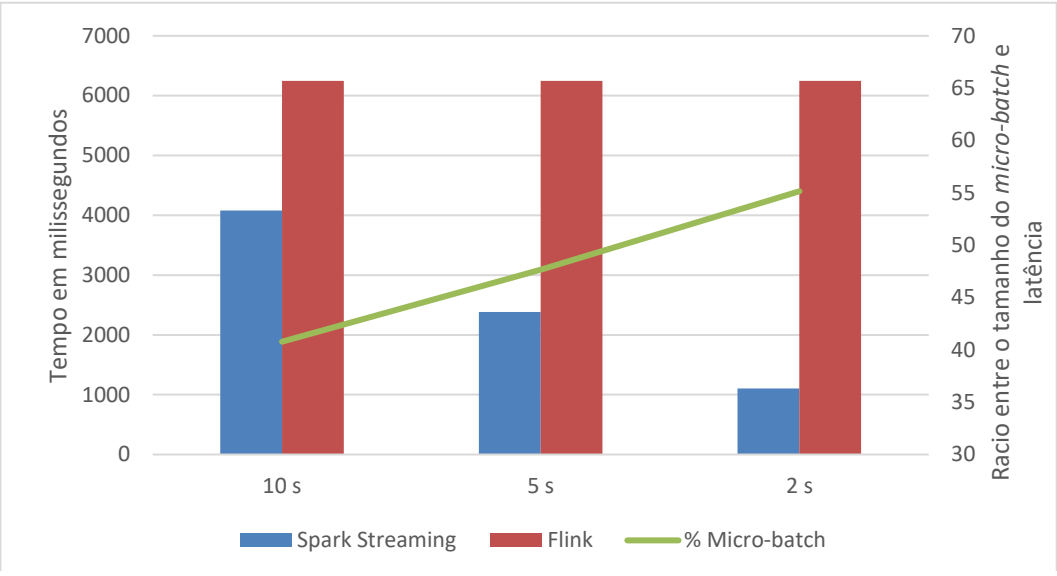

Gráfico 8 - Latência média no cenário 3

O valor do *throughput* obtido no Spark Streaming, representado a azul no Gráfico 9, decresce drasticamente à medida que diminui o tamanho do *microbatch.* Observa-se uma redução do throughput de 17% quando o tamanho do *micro-batch* é reduzido de 10 para 5 segundos (redução do *micro-batch* de 50%). No entanto, a redução do tamanho do *micro-batch* de 5 para 2 segundos (redução do *micro-batch* de 60%) provoca uma diminuição de 44% do valor do *throughput*. Os exemplos anteriores demonstram que a diminuição do micro-batch não é proporcional à diminuição do desempenho, e que, há uma perda de desempenho mais significativa quanto existe uma redução para valores menores de *micro-batch.*

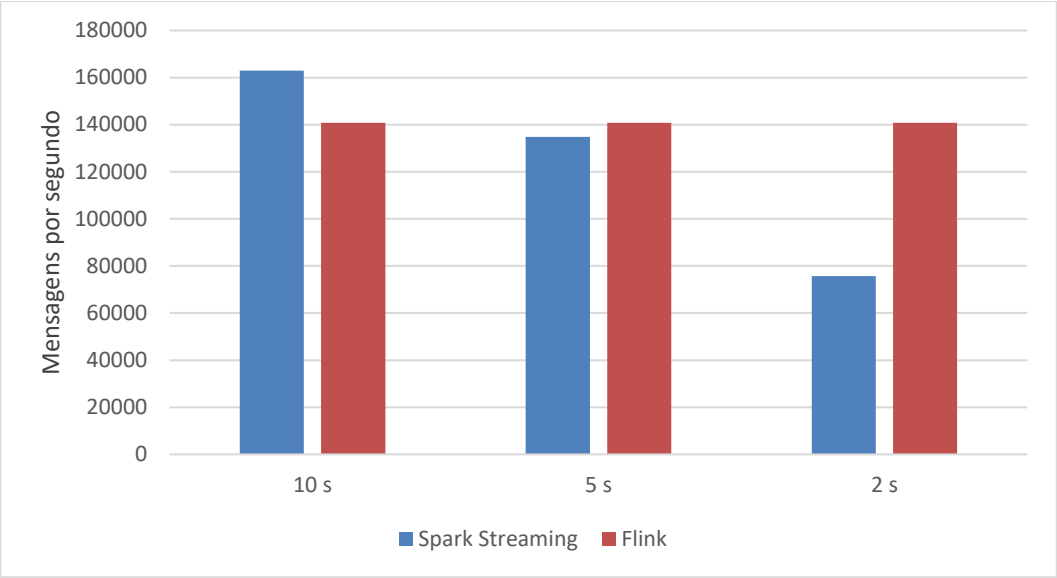

Gráfico 9 - Throughput médio no cenário 3

5.5 Cenário 4

No estudo da influência do paralelismo nas ferramentas de *streaming* é importante compreender como estas utilizam os recursos do *cluster*. O Spark Streaming utiliza recursos alocados pelo YARN quando executado/*deployed* em modo "*cluster*", enquanto que o Flink funciona de forma *standalone*. De modo a uniformizar os testes foi utilizada a metodologia explicitada no subcapítulo 4.3.4 onde foram definidos os valores de 100%, 60% e 20% do máximo *executores* para o Spark Streaming e que correspondem respetivamente à utilização de cinco, três e um nó por parte do Flink.

O Gráfico 10 demonstra que existe um decréscimo no tempo de processamento consoante se aumenta o paralelismo e uma clara superioridade de desempenho por parte do Flink. Esta ferramenta apresenta um tempo de processamento 13% superior ao obtido pelo Spark Streaming no valor mínimo de paralelismo, no entanto, para o valor médio e máximo de paralelismo apresenta uma diminuição do tempo de processamento de 14% e 23%, respetivamente.

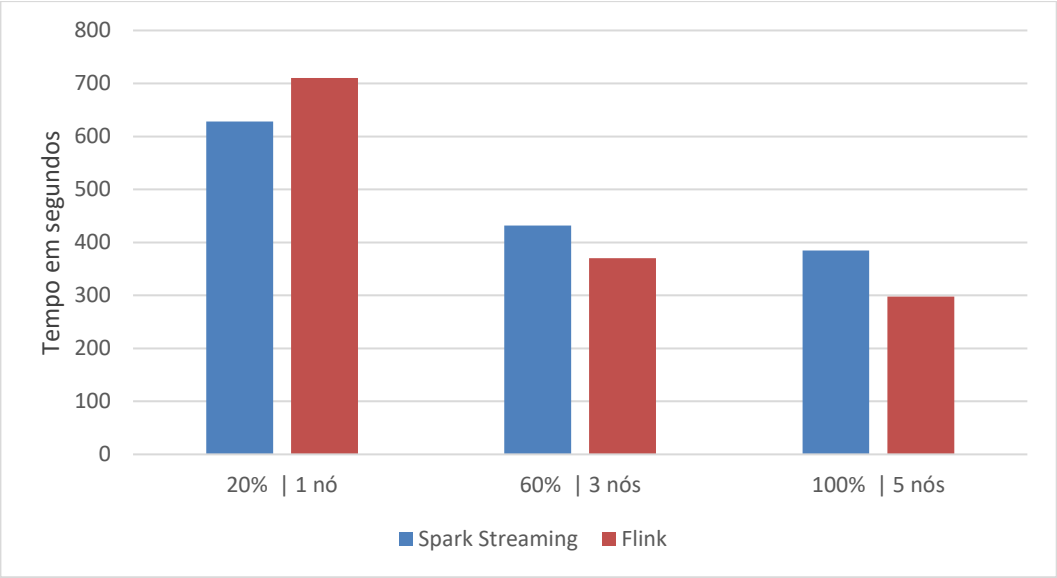

Gráfico 10 - Tempo de processamento do cenário 4

Em relação à análise da latência é possivel verificar, a partir do Gráfico 11, que o seu valor mínimo é obtido para o valor de paralelismo de "60% | 3 nós". O aumento da latência verificado quando se aumenta o paralelismo deve-se ao facto de existir um tempo extra necessário na coordenação das diferentes máquinas e na agregação e desagregação dos dados. A latência média obtida pelo Flink é em média 70% superior que a obtida no Spark Streaming.

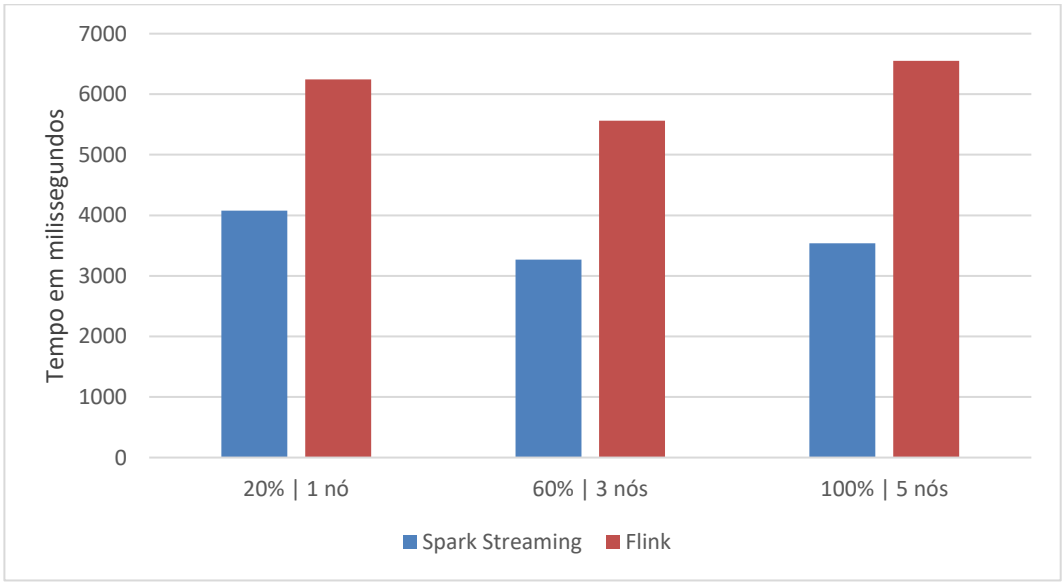

Gráfico 11 - Latência média do cenário 4.

Como expectável, existe um aumento do *throughput* conforme se aumenta o paralelismo das aplicações de *streaming*. Esse aumento, conforme se observa no Gráfico 12, é mais evidente no caso do Flink, em que se regista um aumento de 138% do valor do throughput quando se passa da utilização de apenas 1 nó para a utilização de 5 nós. O Spark Streaming sofre apenas um aumento de 61% nas mesmas circunstâncias.

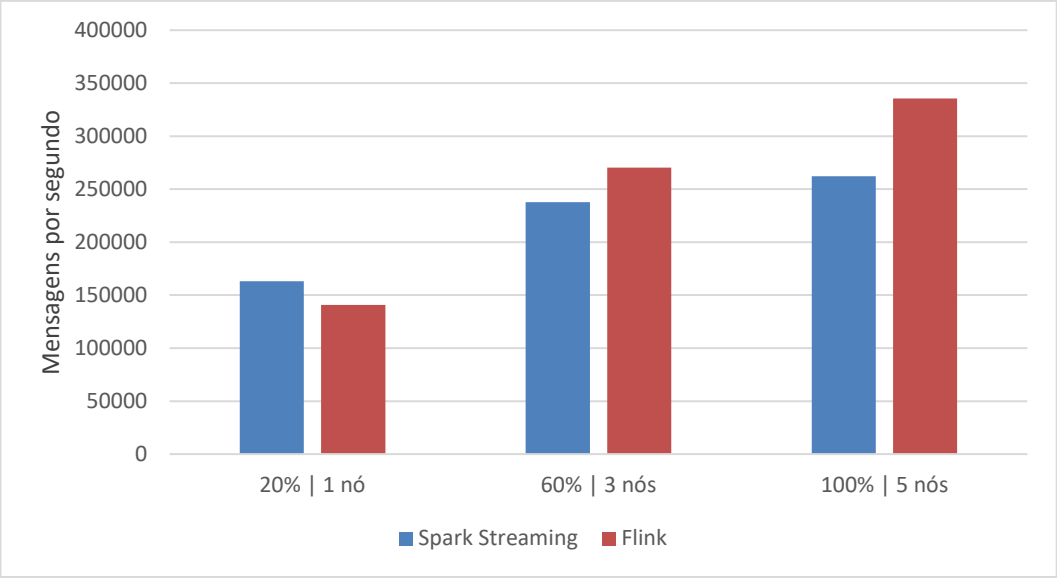

Gráfico 12 - Throughput médio do cenário 4

5.6 Cenário 5

O cenário 5 visa estudar o impacto da variação do número de mensagens ingeridas nas ferramentas de *streaming*. Como evidenciado no Gráfico 13, o tempo de processamento está diretamente relacionado com o número de mensagens ingeridas pelas ferramentas de *streaming*. De facto, o aumento do número de mensagens de 100 milhões para 1000 milhões corresponde a um aumento de 900% do tempo de processamento, valor este que é semelhante ao valor de 890% resultante da diferença entre o valor de processamento de 100 e 1000 milhões, por parte do Spark streaming.

É possível verificar também que a diferença de desempenho entre as ferramentas de *streaming* é acentuada conforme se aumenta o número de mensagens produzidas, dilatando a vantagem por parte do Spark Streaming de 13% para 22%.

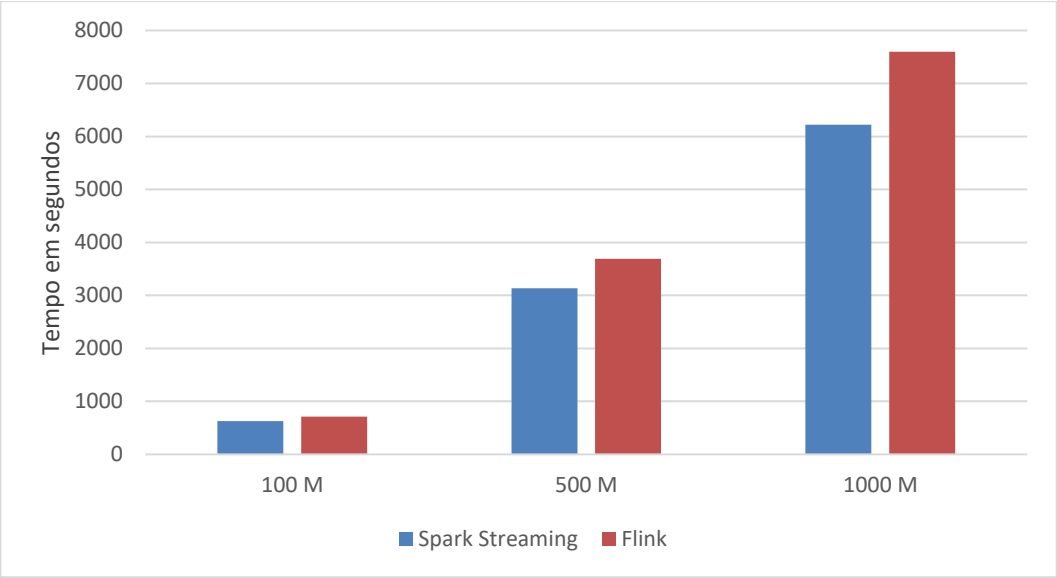

Gráfico 13 - Tempo de processamento no cenário 5

No Gráfico 14 é apresentada a latência média do cenário 5, verificando-se que existe um aumento de apenas 1% nas diferentes medições realizadas no Flink e um aumento de 11% no caso das medições efetuadas no Spark Streaming. O Flink apresenta em média um valor de latência 33% superior ao do Spark Streaming.

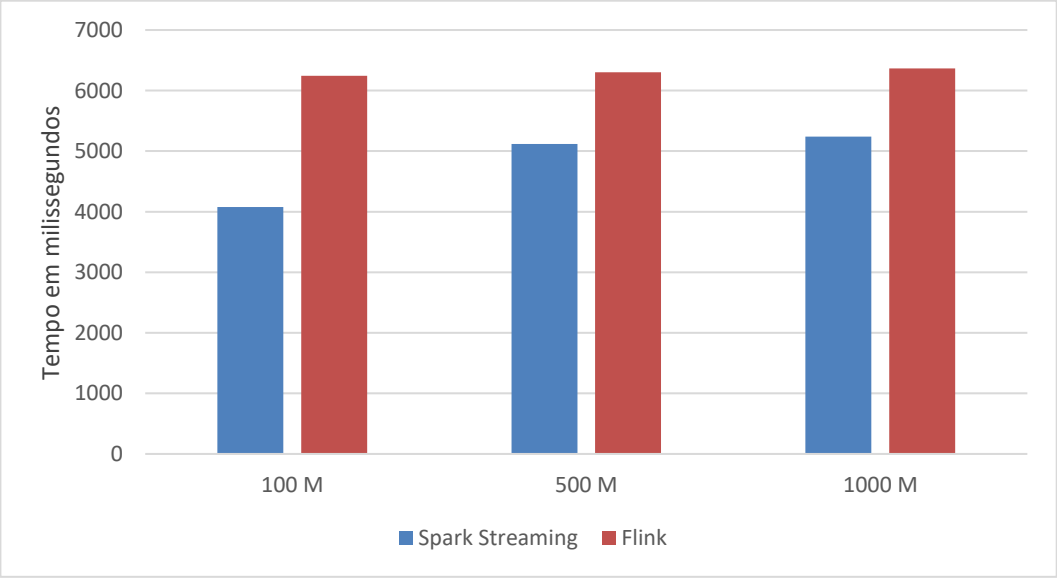

Gráfico 14 - Latência média no cenário 5

No Gráfico 15 constata-se que o *throughput* é constante nas diferentes ferramentas de *streaming* independentemente do número de mensagens ingeridas. Verifica-se uma diminuição média de 1% do

valor do *throughput* quando se aumenta o valor do número de mensagens ingeridas no Spark Streaming. No caso da ferramenta de streaming, esta diminuição é de 2% no Flink. O *throughput* medido no Flink é em média 18% superior ao medido no Spark Streaming.

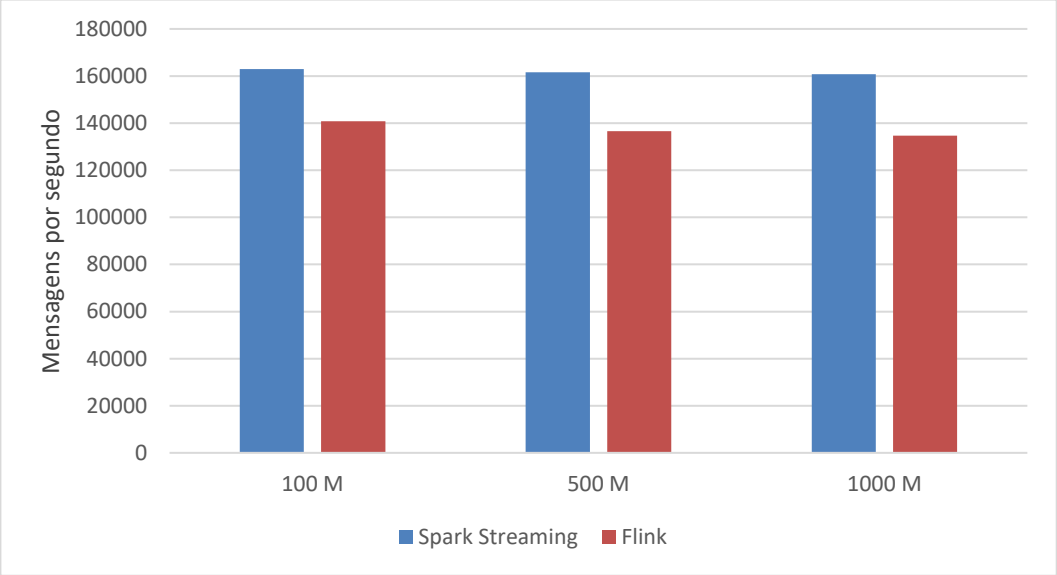

Gráfico 15 - Throughput médio no cenário 5

6. Discussão

Neste capítulo são discutidos os resultados orientados às métricas de modo a permitir uma perceção mais ampla do impacto de cada indicador no desempenho das ferramentas streaming. São ainda discutidas a dificuldades de implementação de aplicações de *streaming.*

6.1 Tempo de processamento

O tempo de processamento é influenciado por todos os indicadores anteriormente apresentados e está diretamente relacionado com o número de mensagens ingeridas.

O Cenário 5, onde se estudou o impacto da variação do número de mensagens, permitiu estabelecer uma correlação direta entre o tempo de processamento e o número de mensagens ingeridas, verificando-se que aumentando apenas o número de mensagens ingeridas e mantendo todos os outros indicadores constantes, obtém-se um aumento proporcional no tempo de processamento.

O indicador "número de mensagens", estudado no Cenário 5, foi excluído do Gráfico 16 de modo a permitir uma melhor perceção do impacto dos restantes indicadores no tempo de processamento (o número de mensagens ingeridas era 10 vezes superior ao dos restantes indicadores, dificultando a análise visual dos resultados sobre a forma de gráfico).

O Gráfico 16 foi construído para melhorar a perceção e comparar o impacto dos diversos indicadores no tempo de processamento das duas ferramentas de *streaming.* Deste modo, foram representados os diferentes cenários no eixo das abcissas (cada cenário corresponde à avaliação de um indicador) e no eixo das ordenadas o valor mais divergente em relação ao valor base.

Como se pode verificar no Gráfico 16, o cenário 2 é o que regista tempo de processamento mais elevado. Este facto é justificável pela exigência computacional que uma operação de *join* entre dois *streams* acarreta. Assim, quanto maior o valor da janela temporal associada à operação de *join*, maior o tempo de processamento em ambas as ferramentas de *streaming*.

Constata-se também que existe uma diferença de 88% no tempo de processamento entre as ferramentas de *streaming* no cenário 3. Como o Flink utiliza uma abordagem de processamento mensagem a mensagem e não *micro-batch,* o valor de tempo de processamento medido é igual ao valor do cenário base. No caso do Spark Streaming, conforme se diminui o tamanho do *micro-batch* para valores próximos de 2 segundos, o tempo de processamento aumenta de forma não linear ao tamanho do micro-batch.

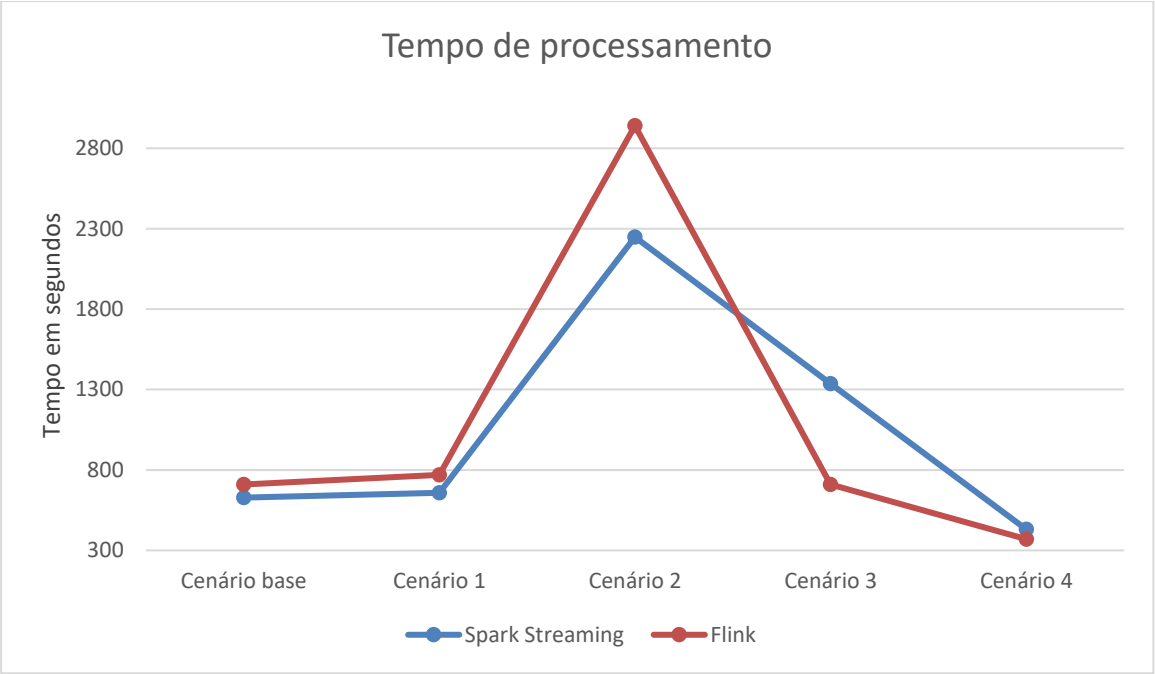

Gráfico 16 - Valor do tempo de processamento mais divergente, por cenário

É importante concluir que os indicadores Tamanho da janela e Tamanho do micro-batch, representados no Gráfico 16 como cenário 2 e cenário 3 respetivamente, são os que mais impactam negativamente no tempo de processamento de um conjunto de mensagens. Percebe-se que aumentando o paralelismo se consegue diminuir o tempo de processamento de uma aplicação de *streaming*, não obstante, é de extrema importância a seleção do tamanho adequado do *micro-batch* e da janela do *join* quando se desenha uma aplicação de *streaming*.

6.2 **Latência**

Como se pode verificar, através do Gráfico 17, o valor máximo de latência é registado para o cenário 2. Neste cenário, varia-se o valor do tamanho da janela do *join* entre streams, e como verificado anteriormente a latência aumenta linearmente com o aumento do valor do tamanho da janela. É utilizado um tamanho máximo de 30 segundos para o valor da janela temporal dos dois streams em que se pretende fazer join. No caso do Spark Streaming corresponde a armazenar em memoria 3 *micro-batchs* de 10 segundos por stream e no caso do Flink corresponde a armazenar o conteúdo ingerido em 30 segundos nos dois streams. Dadas as características apresentadas anteriormente de um *Window Join*, é percetível que arquiteturalmente esta operação impacta diretamente no valor da latência, mesmo não contabilizando o processamento extra necessário para a sua execução. Assim, justifica-se este aumento

do valor da latência comparativamente ao valor base de 620% por parte do Flink e de 290% por parte do Spark Streaming.

A diferença entre os valores máximos de latência obtido pelas duas ferramentas pode ser explicado pela diferença arquitetural das mesmas. O operador de *join* do Flink necessita de aguardar pela entrada dos dados até satisfazer o tamanho da janela e apenas quando termina pode executar a operação de *join*, enquanto que no Spark Streaming a operação de *join* é realizada entre os diferentes *micro-batchs*, não sendo preciso aguardar que estes fiquem completos para iniciar a operação.

Excluindo o indicador "tamanho da janela", o Flink apresenta uma variação máxima de 8% entre o valor base e os valores mais divergentes de cada indicador, portanto é possível concluir que as operações de *join* são as que mais afetam negativamente esta ferramenta de streaming.

O Spark streaming permite obter uma maior variação do valor da latência através da alteração dos valores dos vários indicadores apresentados, em relação ao cenário base. É obtida uma diminuição do valor da latência de 73% quando selecionado um microbatch de 2 segundos e 20% quando aumentado o paralelismo. O valor da latência aumenta 29% quando o número de mensagens ingeridas é 10 vezes superior ao cenário base.

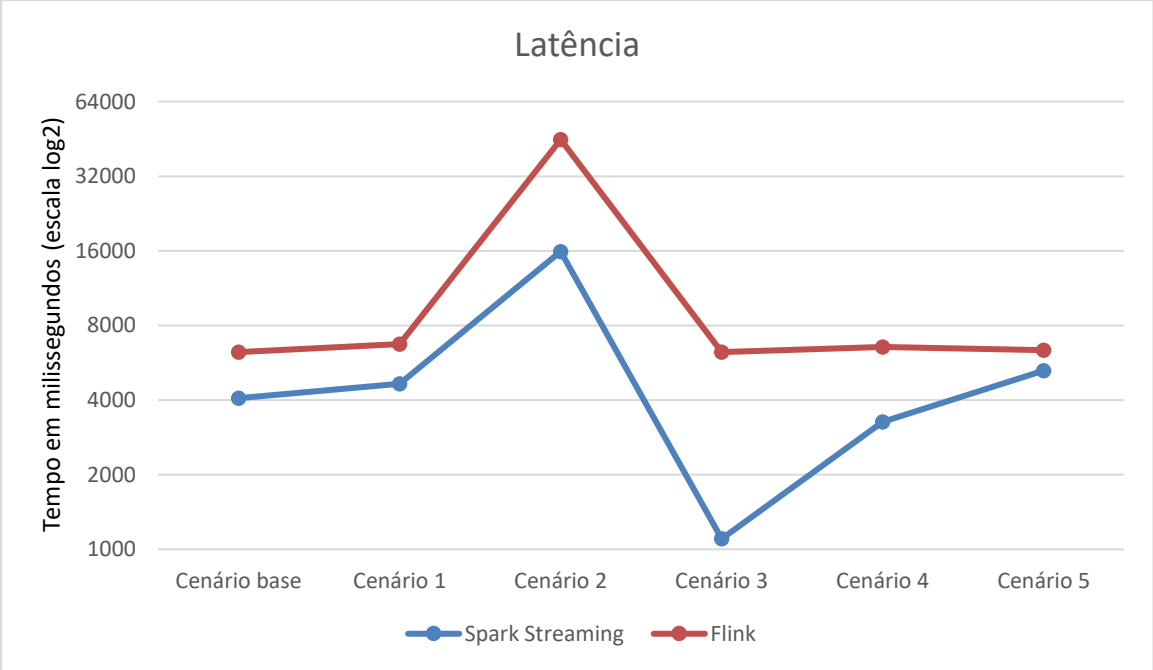

Gráfico 17 - Valor da latência mais divergente, por cenário

6.3 **Throughput**

O throughput é definido como o número de mensagens processados por segundo. Existe uma relação inversamente proporcional entre o throughput e o tempo de processamento em condições

normais, ou seja, quando o número de mensagens processadas é igual ao número de mensagens ingeridas.

Como se pode verificar no Gráfico 18, o valor máximo do *throughput* é registado no cenário 4 quando o paralelismo das ferramentas de *streaming* se encontra maximizado. Este valor é espectável uma vez que são utilizados mais recursos para processar o mesmo número de mensagens.

O cenário 2 apresenta um decréscimo de 22% do valor de *throughput* relativamente ao cenário base. No entanto, este decréscimo não é tao acentuado como o que se fazia prever tendo em conta o valor do tempo de processamento registado no Gráfico 16. Este fenómeno deve-se ao facto de serem processadas mais mensagens do que as que são ingeridas. No *join* de tipo "30s-30s", representado no cenário 2, são geradas e posteriormente processadas 3 mensagens por cada mensagem ingerida. Este comportamento deve-se ao facto de no intervalo de 30 segundos, definido para o tamanho da janela do *stream* 2, existirem chaves estrangeiras em triplicado.

O valor do *throughput* decresce, em ambas as ferramentas, apenas 7% quando existe um amento do tamanho da mensagem de 583% e aumenta, em média, apenas 3% quando o número de mensagens ingeridas é 10 vezes superior (cenário 1 e 5 respetivamente).

Os indicadores que mais influenciam negativamente o valor do throughput são o tamanho do *micro-batch* e o tamanho da janela da operação *join*, este último não é percetível apenas pela visualização do Gráfico 18. A forma mais direta de se aumentar o throughput das aplicações de streaming, em ambas as ferramentas, consiste em aumentar o paralelismo das mesmas.

É de extrema importância realçar o decréscimo de 54% do valor do *throughput* em relação ao cenário base e de 46% em relação ao Flink, por parte do Spark Streaming, no cenário 3. Este decréscimo deve-se ao facto de ser utilizado um tamanho de *micro-batch* de 2 segundos, detalhado na subsecção seguinte.

A aparente superioridade do Spark Streaming desvanece no cenário 3, dadas as suas características de abstração de *streams*. Havendo a necessidade de processamento de dados em real-time o tamanho do *micro-batch* teria de ser menor, uma vez que só se obtém os dados processados em intervalos maiores que o do *micro-batch.* Como o throughput decresce de forma não linear com a diminuição o do tamanho do *micro-batch*, este seria extremamente baixo para valores de *micro-batch* pequenos o suficiente para se considerar uma aplicação de streaming em *real-time*. Assim, é preciso existir um equilíbrio entre o tamanho do *micro-batch* definido e o throughput que se pretende obter.

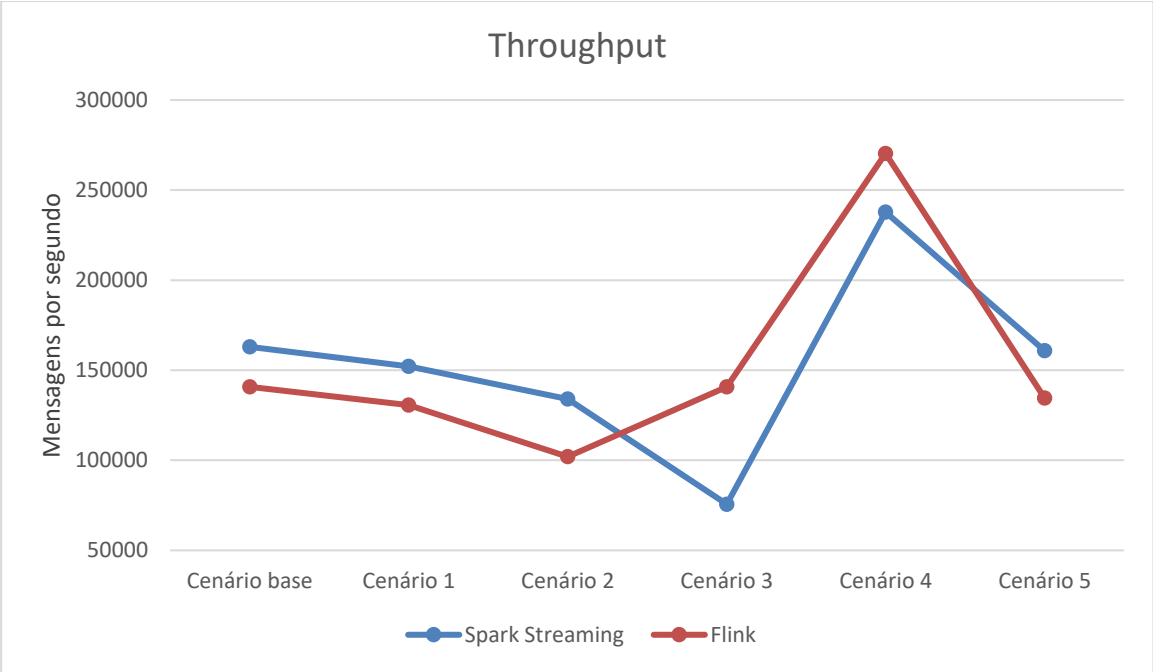

Gráfico 18 - Valor do throughput mais divergente, por cenário

6.4 **Spark Streaming**

Para que uma aplicação de Spark Streaming seja executada num cluster de forma estável, é necessário que o sistema seja capaz de processar os dados tão rápido quanto a sua chegada, ou seja, o tempo de processamento de um *micro-batch* deve ser menor que o seu tamanho. Para verificar estas condições é possível consultar o *dashboard web* providenciado pela ferramenta.

A documentação do Spark Streaming recomenda que se teste a aplicação de *streaming* com um tamanho de *micro-batch* conservador (entre 5 a 10 segundos) e ainda limitar o fluxo de dados que será ingerido, neste caso limitar o número de mensagens geradas pelos *producer 1.*

Estas recomendações foram seguidas e verificado, no *dashboard web*, se o tempo de processamento do *micro-batch* é menor o seu tamanho e se o *Total Delay* é constante.

Na Figura 20 é possível visualizar os resultados de uma experiência inicial de uma aplicação do Spark Streaming. Verifica-se que o *Processing Time* médio é superior ao tamanho do *micro-batch* o que consequentemente leva a um aumento do *Total Delay* e à consequente instabilidade do Spark Streaming.

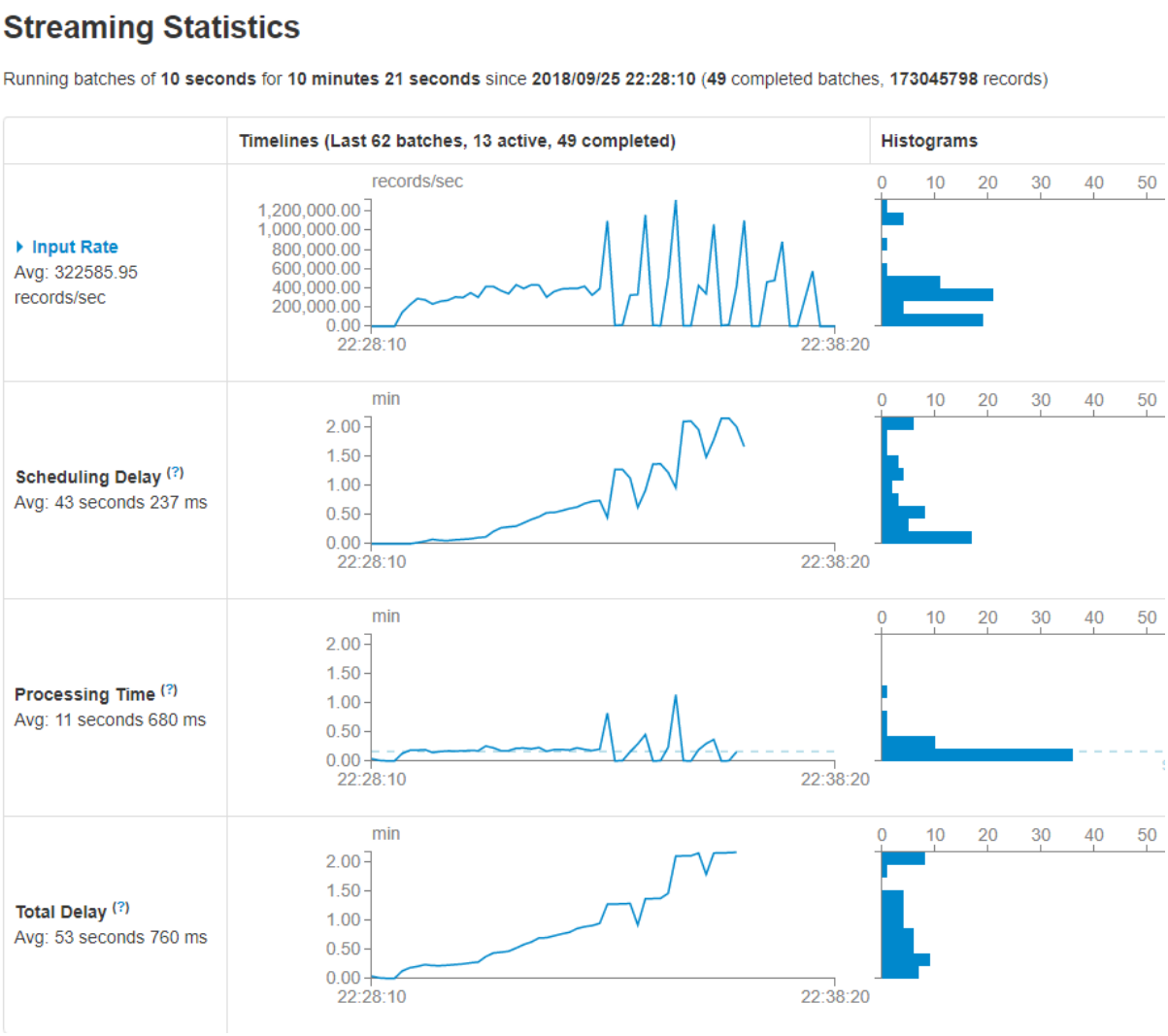

Figura 20 - Dashboard web do Spark Streaming.

A abordagem utilizada para estabilizar as aplicações de Spark Streaming foi reduzir o fluxo de dados, limitando o *producer* 1 a produzir apenas aproximadamente 100 mil mensagens por segundo. Na Figura 21, é visível que a redução do fluxo de ingestão de dados fez reduzir o *Processing Time* para um valor médio inferior ao tamanho do *micro-batch,* e que o *Total Delay* não aumenta ao longo do tempo, como anteriormente. Analisando atentamente o *dashboard* percebe-se a existência de uma grande instabilidade da aplicação, ocorrendo vários picos onde o *Processing Time* de um *micro-batch* excede o seu tamanho. Ou seja, a instabilidade mantém-se mesmo seguindo as recomendações da documentação.

Depois de realizados alguns testes e analisados os dados provenientes do *dashboard web* e dos *logs* relativos à aplicação de *stream*, conclui-se que o problema é resultante do aumento do *Processing Time* esporádico e o seu impacto nas operações de *join* entre *streams*.

Como consequência do aumento do *Processing Time,* o Spark Streaming gera *micro-batchs* com um número de mensagens muito superior ao que era previsto, este aumento de mensagens condiciona a operação de *join,* não funcionando como foi projetada.

O *stream* 2 tem uma *throughput* de 5 mensagens por segundo e contêm 50 mensagens diferentes, desta forma, um *micro-batch* de 10 segundos deveria conter no máximo 50 mensagens diferentes. Quando existe um *join* entre o *stream* 1 e o *stream* 2 por id, com uma janela de 10 segundos, existiria a garantia de haver uma relação de 1 para 1 no número de mensagens do *stream* resultante.

Como o Spark Streaming cria *micro-batchs* com um número de mensagens superior ao suposto, implica que no mesmo *micro-batch* existam várias mensagens iguais (chaves estrangeiras repetidas). Esta divergência leva a um funcionamento errado da operação de *join* entre *streams,* levando à criação de múltiplas mensagens e à redução da performance da aplicação.

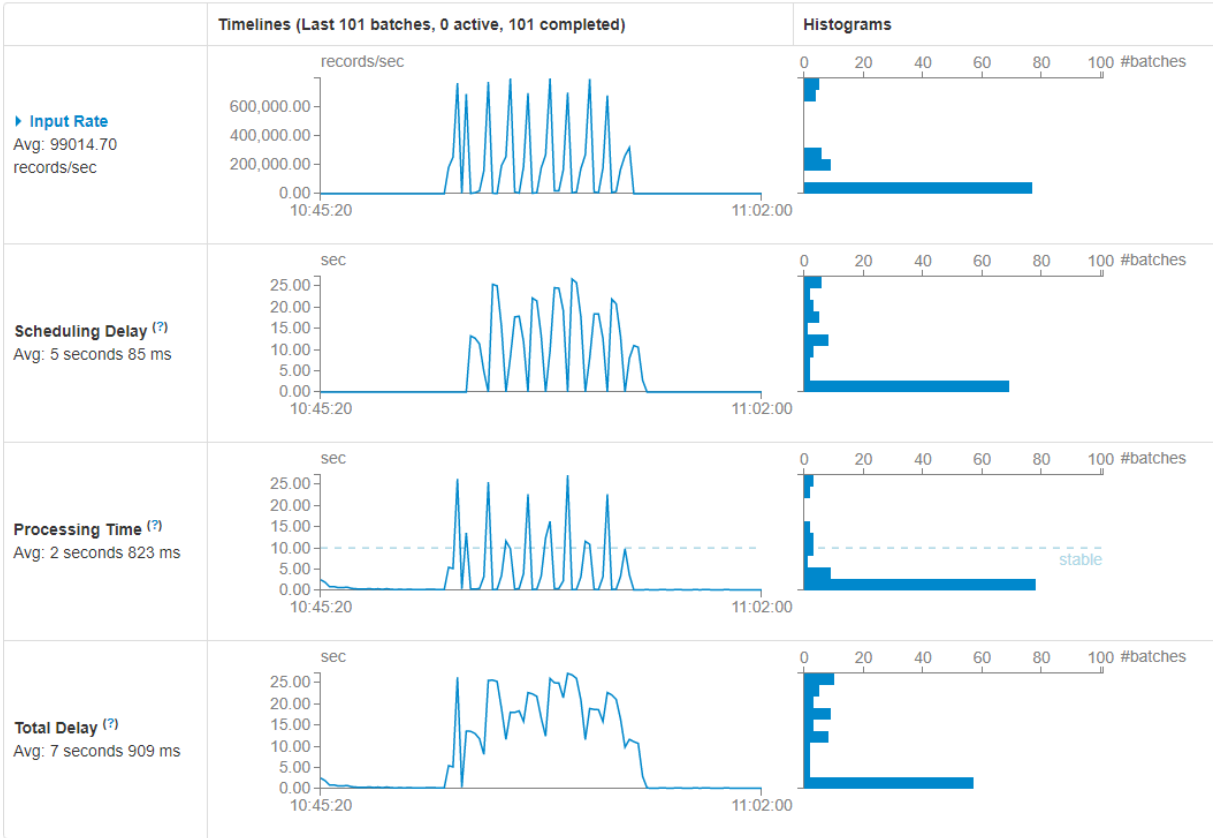

Figura 21 - Dashboard web do Spark Streaming com baixo throughput

De modo a mitigar este problema limitou-se o número de mensagens por partição do Kafka. Esta solução foi apresentada em diversos fóruns, como forma de resolver a instabilidade apresentada pela operação de *joins* entre streams, uma vez que esta funcionalidade foi apenas introduzida na versão do Spark Streaming utilizada na realização desta dissertação.

Esta alteração foi a que mais aumentou a estabilidade das aplicações do Spark Streaming permitindo aumentar a sua *performance*. Como se pode verificar na Figura 22, o *Imput Rate* é mantido de forma constante mesmo sendo superior ao medido na Figura 21. O *Processing Time* dos diferentes *micro-batches* tem um valor médio dentro do limite do recomendado para uma execução estável e os seus picos são menores.

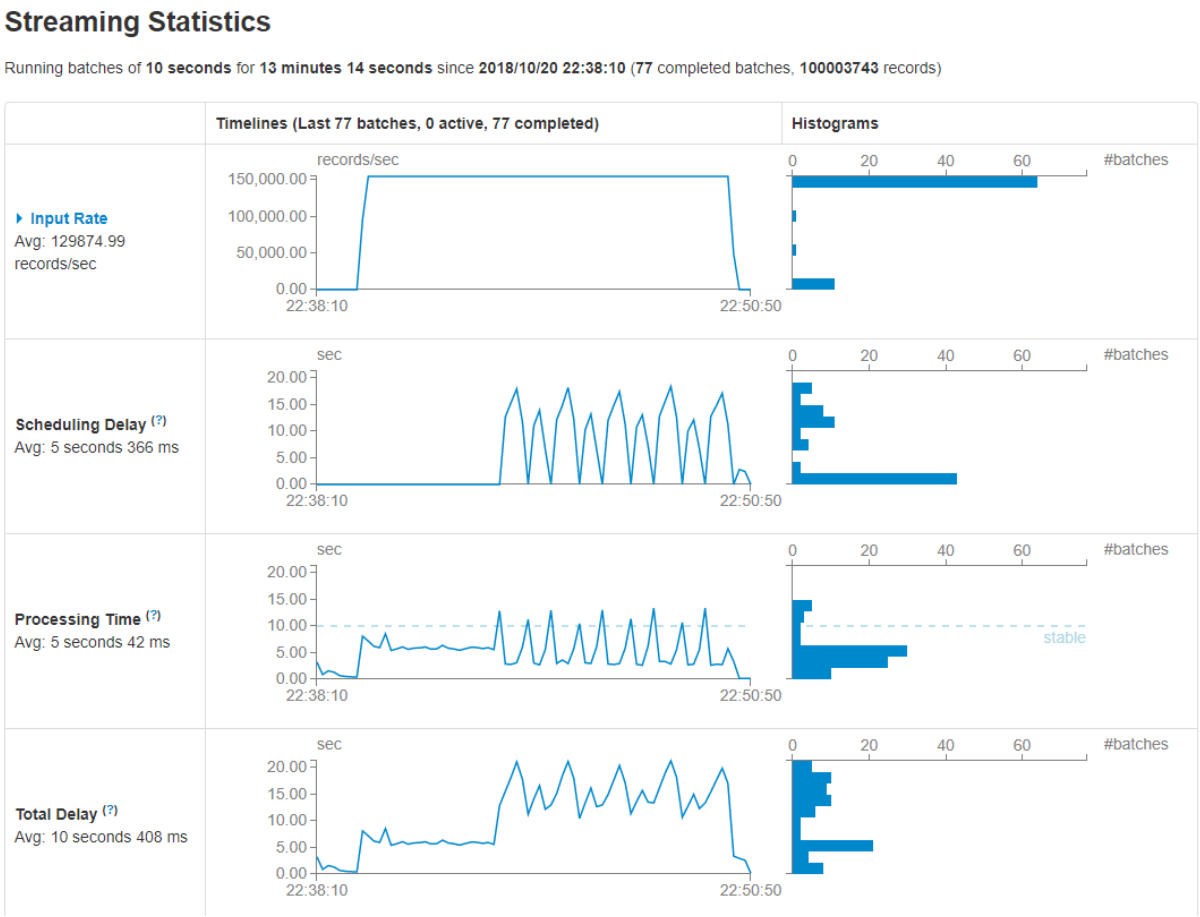

Figura 22 - Dashboard web do Spark Streaming limitado pelo Kafka

Para executar os testes para todos os cenários criados no máximo do desempenho do Spark Streaming, foi necessário utilizar uma abordagem de ajustes iterativos de throughput. Eram realizados testes preliminares e consoante a análise dos resultados obtidos pela interface web, ajustado o valor do Imput Rate através da limitação do número de mensagens por tópico do Kafka. O objetivo era maximizar o desempenho do Spark Streaming, aproximando a aplicação do limite da sua estabilidade.

6.5 Flink

O Flink mostrou-se mais estável que o Spark Streaming quando o throughput de mensagens enviadas pelo Kafka excedia aquele que esta conseguia suportar. O Flink manteve um desempenho constante independentemente do fluxo de mensagens ingeridas, apresentando valores consistentes nos vários testes realizados.

A *interface web* apresenta várias métricas por operador, sendo possível identificar as causas de *bottleneck*. Outra vantagem associada ao Flink é a capacidade de se poder ajustar o paralelismo por operador e assim fazer-se uma melhor gestão de recursos.

6.6 Considerações finais

Como se pode verificar pela análise dos resultados dos vários cenários, existem vantagens e desvantagens na utilização das duas ferramentas de *streaming.*

Através da análise dos resultados e do estudo realizado das duas ferramentas, foi possível perceber que o Spark Streaming obtém um bom desempenho em situações especificas. Para se obter uma melhor performance é necessário utilizar um tamanho de *micro-batch* tanto maior quanto possível. No entanto, sendo uma aplicação de streaming presume-se a importância do processamento e obtenção dos dados em tempo real e consequente necessidade de utilização de um *micro-batch* pequeno. Este equilíbrio, entre o tamanho do *micro-batch* e a necessidade de obter dados em tempo real é muito delicado, sendo muito afetado pela variação do fluxo de ingestão de mensagens. A utilização do Spark streaming é aconselhada em cenários em que a variação do fluxo de mensagens seja reduzido e que a obtenção dos resultados em tempo real não seja uma prioridade máxima.

O Flink apresenta uma performance consistente, maximizando a sua superioridade em relação ao Spark Streaming quando há a necessidade da obtenção de dados em tempo real. O Flint comporta-se de forma estável mesmo quando há uma grande variação do fluxo de mensagens. Assim, é recomendável utilizar o Flink em cenários em que existe variação do fluxo de mensagem e exista a necessidade da obtenção dos dados em tempo real.

7. CONCLUSÃO

A presente dissertação insere-se no contexto atual da existência de grandes volumes de dados e do desafio da gestão dos mesmos em tempo real. Neste âmbito foram expostas neste documento as principais características e desafios do *Big Data* bem como as abordagens e particularidades do *streaming* de dados em contexto *Big Data*, tendo por base a bibliografia disponível.

No decorrer da elaboração deste trabalho de investigação, foi percetível que os trabalhos dos autores de referência abordam detalhadamente as várias ferramentas de processamento de dados em *Big Data,* bem como as características do streaming. No entanto, identificou-se uma lacuna no que remete ao *benchmark* de ferramentas de streaming.

Assim, e verificada a necessidade de uma comparação de ferramentas de *streaming* através do processo de revisão de literatura, definiu-se como objetivo dar resposta a esta lacuna. Para tal, foi feita uma contextualização das tecnologias e dos principais cenários de utilização, com o objetivo de definir um protocolo de avaliação das várias *frameworks* de *streaming.*

Para dar origem ao protocolo de avaliação, foi definida de uma infraestrutura tecnológica assente num *cluster* constituído por 5 máquinas. Posteriormente passou-se à definição do protocolo de avaliação das *frameworks* de *streaming,* com o propósito de possibilitar uma avaliação equivalente entre as várias ferramentas. Assim, foi definido num problema prático que necessitava da utilização de *frameworks* de *streaming* para uma resolução eficaz. Foram criados indicadores por forma a aumentar a variabilidade dos testes, dando origem a um cenário de teste por indicador. Para a recolha de dados, foram estabelecidas três métricas que possibilitaram a análise do desempenho das diferentes *frameworks* de *streaming.*

O desempenho das *frameworks* de *streaming* apresentadas foi avaliado recorrendo à criação de cinco cenários, sendo que cada um dos cenários apresenta três valores diferentes para o indicador de estudo naquele cenário. Os resultados foram apresentados por cenário, tendo sido realizada uma análise posterior orientada às métricas definidas.

Por fim são discutidas as dificuldades de implementação das aplicações de streaming e ainda os cenários vantajosos para cada uma das ferramentas.

7.1 Trabalho realizado

A nível prático, o objetivo desta dissertação consistia em fazer um benchmark a ferramentas de Streaming em contexto de Big Data. Para tal foi realizada uma contextualização das tecnologias de processamento de *streaming* e posteriormente realizada a definição de protocolo de avaliação das mesmas. Foi realizada a avaliação do seu desempenho das ferramentas de streaming utilizando um benchmark desenvolvido. Todos os objetivos iniciais foram cumpridos através das seguintes tarefas:

- Revisão de literatura pertinente;
- Desenvolvimento de um protocolo de testes;
- Exploração das ferramentas necessárias para o processamento de *streams;*
- Desenvolvimento da infraestrutura tecnológica;
- Instalação e configuração das diferentes tecnologias;
- Desenvolvimento de produtores e consumidores de dados;
- Realização de testes às ferramentas de *streaming;*
- Otimização dos testes para maximizar o desempenho das ferramentas de *streaming*;
- Apresentação e discussão dos resultados.

7.2 Dificuldades e Limitações

Depois da realizada as revisões de literatura instalaram-se três ferramentas que se consideraram relevantes para a dissertação. Durante utilização exploratória das mesmas sentiu-se bastante dificuldade na criação de consumidores de dados para uma delas (Apache Storm). Dada a falta de documentação e a complexidade da ferramenta, o processo de desenvolvimento do consumidor de dados para esta verificou-se demasiado moroso. Embora tenha sido criado um consumidor de dados funcional, surgiram muitas dificuldades na implementação e configuração de alguns dos operadores.

Foi tomada a decisão de abandonar esta ferramenta e aumentar o foco nas outras duas, uma vez que a documentação existente era mais basta e de melhor qualidade. Assim, embora se tenha realizado o *benchmark* de apenas duas ferramentas, foi possível otimizá-las para a obtenção de um melhor desempenho.

7.3 Trabalho futuro

Existindo uma constante evolução das ferramentas de processamento de *streams* de dados, considera-se pertinente uma avaliação das novas versões, para verificar se estas corrigem os problemas de estabilidade verificados no Spark Streaming. Para se obter uma melhor perspetiva do desempenho destas ferramentas considera-se importante, como trabalho futuro, a adição de mais ferramentas a este *benchmark,* como por exemplo o Apache Samza e o Apache Storm.

Seria importante também avaliar o comportamento de tolerância a falhas das várias ferramentas.

8. REFERÊNCIAS BIBLIOGRÁFICAS

Bifet, A. (2013). Mining big data in real time. *Informatica (Slovenia)*, *37*(1), 15–20. https://doi.org/10.1.1.368.1416

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, *39*(4), 12. https://doi.org/10.1145/1978915.1978919

Chandarana, P., & Vijayalakshmi, M. (2014). Big Data analytics frameworks. *2014 International Conference on Circuits, Systems, Communication and Information Technology Applications (CSCITA)*, 430–434. https://doi.org/10.1109/CSCITA.2014.6839299

Chandio, A. A., Academy, C., Tziritas, N., Academy, C., Xu, C.-Z., Academy, C., … Academy, C. (2015). Big-Data Processing Techniques and Their Challenges in Transport Domain Big-Data Processing Techniques and Their Challenges in Transport Domain. *ZTE Communications*, *1*(November), 10. https://doi.org/10.3969/j.issn.1673-5188.2015.01.007

Chang, W. L. (2015). NIST Special Publication 1500-6 : NIST Big Data Interoperability Framework - Reference Architecture, *6*, 1–62. https://doi.org/10.6028/NIST.SP.1500-6

Chen, H., Chiang, R. H., & Storey, V. C. (2012). *Understanding Big Data analytics*. Retrieved from http://searchstorage.techtarget.com/feature/Understanding-Big-Data-analytics

Chen, M., Mao, S., & Liu, Y. (2014). Big data: A survey. *Mobile Networks and Applications*, *19*(2), 171–209. https://doi.org/10.1007/s11036-013-0489-0

CISCO. (2017). The Zettabyte Era: Trends and Analysis. *Cisco*, (June 2017), 1–29. https://doi.org/1465272001812119

Costa, C., & Santos, M. Y. (2017). Big Data: State-of-the-art concepts, techniques, technologies, modeling approaches and research challenges. *IAENG International Journal of Computer Science*, *44*(3), 285–301.

Demchenko, Y., Grosso, P., De Laat, C., & Membrey, P. (2013). Addressing big data issues in Scientific Data Infrastructure. *Proceedings of the 2013 International Conference on Collaboration Technologies and Systems, CTS 2013*, 48–55. https://doi.org/10.1109/CTS.2013.6567203

Gandomi, A., & Haider, M. (2015). Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management*, *35*(2), 137–144. https://doi.org/10.1016/j.ijinfomgt.2014.10.007

Gart, N. (2015). *Learning Apache Kafka* (Second). Packt Publishing Ltd.

Hadoop. (2017). Welcome to Apache™ Hadoop®! Retrieved from http://hadoop.apache.org/

Hausenblas, M., & Bijnens, N. (2017). Lambda Architecture » λ lambda-architecture.net. Retrieved February 13, 2018, from http://lambda-architecture.net/

He, C. (2015). Survey on NoSQL Database Technology, *2*(2), 50–54. https://doi.org/10.1109/ICPCA.2011.6106531

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *Design Science in IS Research MIS Quarterly*, *28*(1), 75–105. https://doi.org/10.2307/25148625

Kaisler, S., Armour, F., Espinosa, J. A., & Money, W. (2013). Big Data : Issues and Challenges Moving Forward, 995–1004. https://doi.org/10.1109/HICSS.2013.645

Katal, A., Wazid, M., & Goudar, R. H. (2013). Big data: issues, challenges, tools and good practices. In Contemporary Computing (IC3), 2013 Sixth International Conference on (pp. 404-409). IEEE.

Khan, W., & Shahzad, W. (2017). Predictive Performance Comparison Analysis of Relational & NoSQL Graph Databases, *8*(5).

Kranen, P. (2011). Anytime Algorithms for Stream Data Mining, 1–323.

Krishnan, K. (2013). *Datawarehousing in the age of big data*. *Newnes*. https://doi.org/10.1007/s13398-014-0173-7.2

Liu, Z., Yang, P., & Zhang, L. (2013). A sketch of big data technologies. *Proceedings - 2013 7th International Conference on Internet Computing for Engineering and Science, ICICSE 2013*, 26–29. https://doi.org/10.1109/ICICSE.2013.13

Marz, N., & Warren, J. (2015). *Big Data, Principles and best practices of scalable real-time data systems*. *Big Data - Principles and best practices of scalable real-time data systems* (Vol. 37). https://doi.org/10.1073/pnas.0703993104

Mathur, A., Sihag, A., Bagaria, E. G., & Rajawat, S. (2014). A new perspective to data processing: Big data. *2014 International Conference on Computing for Sustainable Global Development, INDIACom 2014*, 110–114. https://doi.org/10.1109/IndiaCom.2014.6828111

Moniruzzaman, A. B. M., & Hossain, S. A. (2013). Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *ArXiv Preprint ArXiv:1307.0191*, *6*(4), 1–14. https://doi.org/10.1109/ICPCA.2011.6106531

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45–77. article. https://doi.org/10.2753/MIS0742-1222240302

Popeanga, J., & Lungu, I. (2012). Perspectives on Big Data and Big Data Analytics. *Database Systems Journal*, *III*(4), 3–14. https://doi.org/10.5406/jaesteduc.46.4.iii

Spark. (2018). Cluster Mode Overview. Retrieved from http://spark.apache.org/docs/2.3.0/cluster-overview.html

Stonebraker, M., Çetintemel, U., & Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, *34*(4), 42–47. https://doi.org/10.1145/1107499.1107504

White, T. (2012). *Hadoop: The Definitive Guide*. (I. O'Reilly Media, Ed.). O'Reilly Media, Inc.

Wu, X., Zhu, X., Wu, G. Q., & Ding, W. (2014). Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, *26*(1), 97–107. https://doi.org/10.1109/TKDE.2013.109

Zikopoulos, P., Eaton, C., DeRoos, D., Deutsch, T., & Lapis, G. (2011). *Understanding Big Data*. The McGraw-Hill Companies.