INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Estudo de tecnologias para sistemas de Big Data

SAULO ABEL RAMOS SOBREIRO
Julho de 2018

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Estudo de tecnologias para sistemas de Big Data

Saulo Abel Ramos Sobreiro

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Computacionais

Orientador: Fernando Jorge Ferreira Duarte

Júri:

Presidente:

Vogais:

Porto, julho de 2018

Resumo

Big Data é um conceito da moda, consequência da evolução tecnológica dos últimos anos, que tem potenciado o aumento do volume de dados gerados diariamente. As empresas têm cada vez mais noção da oportunidade que é gerar valor dos dados que lhe pertencem, mas isso implica conseguir dar resposta às características dos dados, que se têm tornado intratáveis pelas tecnologias tradicionais. Uma solução para responder a esses desafios é o ecossistema *Hadoop,* que disponibiliza várias tecnologias dedicadas a resolver problemas específicos no âmbito dos desafios de Big Data.

Com o intuito de produzir um estudo de tecnologias de Big Data, é aqui feita uma análise teórica do ecossistema, de quais as *Stacks* de tecnologias mais comuns e de como estas são integradas num sistema *Hadoop*. Posteriormente, e partindo de um caso de uso baseado num sistema para processamento de dados de sensores foi feito um estudo teórico de quais as tecnologias mais adequadas a usar e qual a melhor arquitetura a seguir. Esta análise revelou-se, em termos teóricos, inconclusiva.

Desta forma, surgiu a necessidade de fazer testes práticos de quatro combinações diferentes entre tecnologias de processamento - onde se considerou *Spark Streaming* e *Storm* - e arquitetura seguida - onde se considerou a arquitetura *Lambda* e a arquitetura *Kappa*. O objetivo dos testes foi identificar qual a combinação com melhor desempenho e menor consumo de recursos para o caso de uso em questão. Os testes realizados revelaram, entre outros, que o *Spark* seguindo uma arquitetura *Kappa* é a abordagem com melhor relação desempenho – recursos consumidos.

Palavras-chave: *Big Data*, *Apache Hadoop*, Arquitetura *Lambda*, Arquitetura *Kappa*

Abstract

The technological evolution we've been witnessing these last years has increased dramatically the volume of data generated every single day and, therefore, has turned Big Data into a trendy concept. Companies are also increasingly aware of the potential value of the data they have in their hands. However, treating the volume of data in question means going further than the traditional technologies, which no longer can be used in this context. A solution for this problem is the *Hadoop* ecosystem, which is made by several dedicated technologies developed specifically to tackle Big Data problems.

To study the Big Data technologies available today, this paper starts by presenting a theoretical analysis of the *Hadoop* ecosystem, of which are its most common Stacks and how they are integrated with the system itself. For that, we used a case study based on a data sensor processing system and previous published works to determine the technologies most suitable for this case and the best possible architecture. This theoretical analysis has, however, produced inconclusive results.

Therefore, the opportunity arose to test four different combinations testing two processing technologies - *Spark Streaming* and *Storm* – and two architectures – *Lambda* and *Kappa*. The aim of these tests was to identify which processing technology/architecture combination has the best performance and uses the least amount of resources for the case in question. The results show us, among other conclusions, that the *Spark-Kappa* approach is the one with the best performance/resources ratio.

Keywords: Big Data, *Apache Hadoop, Lambda* Architecture*, Kappa* Architecture

Índice

Lista de Figuras

Acrónimos e Símbolos

Lista de Acrónimos

AHP	Analytic Hierarchy Process
API	*Application Programming Interface*
ASF	*Apache Software Foundation*
AWS	*Amazon Web Services*
BI	Business Intelligence
CDH	Cloudera Distribution for Hadoop
CDP	Converged Data Platform
CQL	Cassandra Query Language
CSV	Comma-separated Values file
CDH	*Cloudera's Distribtion for Hadoop*
DICOM	*Digital Imaging and Communications in Medicine*
EDW	*Enteprise Data Warehouse*
ELB	Amazon Elastic Load Balancer
EMR	(*Amazon) Elastic MapReduce*
ETL	*Extract Transform and Load*
FQDN	Full Qualified Domain Name
GPS	Global Positioning System
HA	High-Availability
HDF5	Hierarchical Data Format (v5)
HDFS	*Hadoop Distributed File System*
HDP	*Hortonworks Data Platform*
HL7	*Health Level Seven*
IoT	Internet of Things

IC	Índice de Consistência
JAR	Java Archive
JDBC	*Java Database Connectivity*
JSON	*JavaScript Object Notation*
netCDF	*Network Common Data Form*
NRT	*Near Real-Time*
ODBC	*Open DataBase Connectivity*
QFS	Quantcast File System
RC	Razão de Consistência
RDB	Relational DataBase
RDBMS	*Relational Database Management System*
RDD	*Resilient Distributed Dataset*
SDK	Software Development Kit
SELinux	Security Enhanced Linux
SQL	*Structured Query Language*
SSH	Secure Shell
TTL	*Time-to-Live*
UI	User Interface
vCPU	virtual Central Process Unit
WORM	*Write-Once Read-Many (times)*
XML	*eXtensible Markup Language*
YARN	*Yet Another Resource Negotiator*

1 Introdução

Neste capítulo é feita uma abordagem ao contexto em que se insere este documento e à motivação do mesmo. É também descrita a abordagem metodológica de forma a identificar a estrutura dos restantes capítulos do documento.

1.1 Contexto e Motivação

A evolução dos meios de comunicação, da disponibilidade da internet e das tecnologias de uso pessoal como telemóveis e computadores, entre outros fatores, tem levado à geração de quantidades de informação nunca antes imaginadas nos últimos anos. Esta explosão de informação a que se tem assistido promete não ficar por aqui, e estima-se que a partir de 2020 o volume total de dados gerados e consumidos anualmente esteja na ordem dos 40 zettabytes[1] (Maier, 2013). O volume de dados deve-se especialmente à capacidade de as empresas adquirirem com relativa facilidade terabytes de informação sobre, entre outros, os seus clientes, os seus fornecedores e informações operacionais e dados provenientes de múltiplos sensores já ligados em rede e embebidos em objetos particulares, como *smartphones* ou veículos, e distribuídos por edifícios ou locais públicos, como *smartmeters* e outros dispositivos industriais ou não com capacidade de medir e comunicar. As pessoas, por seu lado, independentemente do meio usado, vão continuar a potenciar este crescimento de dados, em especial através das redes sociais, das mais variadas formas com atos simples e quotidianos da "Era Digital" como "comunicar, procurar, comprar e partilhar" (Brown, Manyika, et al., 2011). Há ainda conceitos que, apesar de muito recentes, estão a ser importados para os mais variados setores como é o caso da *Internet of Things,* que potenciará o crescimento dos dados a que se tem assistido, em particular por ser um conceito cada vez mais na moda e que começa a fazer parte de todos os setores e funções da economia global.

O Big Data também é um conceito da moda e tem atraído as atenções pelos potenciais benefícios para vários domínios. No entanto, apesar da aplicabilidade poder ser vista como

[1] 1024 Terabytes = 1 Petabyte, 1024 Petabytes = 1 Exabyte, 1024 Exabytes = 1 Zettabyte

transversal, nem todos os domínios conseguirão tirar partido de igual forma (Brown, Manyika, et al., 2011). Brad Brown, estima o benefício que o Big Data pode representar para as várias áreas através de uma relação entre a facilidade de coletar valor e o potencial para o criar, considerando que os domínios que melhor conseguem gerar valor com o Big Data são as entidades de cuidados de saúde, produção, retalho e governo. Estas são também, de acordo com o estudo, as áreas onde se gera mais dados, considerando aqui as redes sociais apenas como um veículo para os dados gerados nestas áreas.

Atendendo, portanto, a que a sua aplicabilidade se relaciona diretamente com a capacidade de gerar dados, é necessário discutir um tópico que se destaca das restantes áreas, a *Internet of Things*. Apesar de ainda estarmos a homogeneizar a definição deste conceito, a filosofia da *Internet of Things* suporta que "*o que não se pode medir, não se pode controlar*" - Peter Drucker – e que, portanto, tudo deve ser monitorizado. Certamente que afetará a sociedade com aplicações e comodidades em diversas áreas ao trazer inteligência e interoperabilidade entre objetos que até hoje eram desprovidos, sequer, da capacidade de comunicar. Por exemplo, "*ligar a máquina de café 15 minutos depois de o despertador tocar*" ou "*abrir o portão e ligar o aquecimento quando o carro do morador se prepara para entrar*" são amostras desta interoperabilidade que pode trazer conforto ao individuo da era digital. O tema é promissor tanto em aplicabilidade como em potencial para gerar dados.

Articulando o princípio de Peter Drucker e o *Internet of Things* numa possível aplicação real, imagine-se medir os consumos energéticos de uma residência e armazenar esses dados para posterior análise estatística. Esta medição pode ocupar aproximadamente 3,76GB[2]. O valor não parece significativo para a capacidade do *hardware* atual. Contudo, pensando que esses dados seriam armazenados, tratados e processados, por exemplo, pela EDP e que a "EDP Comercial" tem 3.660.826 clientes de acordo com o relatório de "Quotas em Regime de Mercado" de 2015, os 3,76GB por ano transformam-se em aproximadamente 13,4 TB de dados anuais apenas de valores em bruto (dados binarizados, sem referência temporal nem metadados), para este tipo de aplicação. Da mesma forma, haverá muitas outras aplicações, especialmente vindas de empresas de tecnologia especializadas em sistemas de monitorização e controlo, que serão muito provavelmente mais complexas e com aquisições baseadas em muito mais parâmetros que o exemplo anterior da EDP. Este exemplo mostra que os sistemas de monitorização e controlo orientados para as aplicações de monitorização, por si só, constituem um domínio de aplicabilidade do Big Data.

De forma genérica, e independentemente dos domínios de aplicabilidade, o Big Data é já usado por organizações de forma a gerar valor. As suas características tornam análises que levariam muito tempo segundo métodos tradicionais, em ações responsivas. Esta otimização marca a diferença em práticas já seguidas levando-as a novos níveis. Por exemplo, uma aplicabilidade desta otimização é relativa ao desempenho de gestão. A gestão estratégica requer resposta a questões relacionadas com o segmento de mercado, vendas, picos de procura, entre outros aspetos, e requer que potencialmente vários sejam correlacionados de forma a obter as

[2] 3,76GB = 32 parâmetros numéricos não inteiros, adquiridos por segundo, durante 1 ano

respostas concretas ao que se procura. Estas práticas têm, por vezes, processos demorados e são baseadas em relatórios periódicos que não variam durante muito tempo. Os tempos de resposta obtidos pelo Big Data transformam esta perspetiva pois permitem que estas pesquisas sejam feitas de forma mais simples, com melhores tempos de resposta e de forma mais customizada que os métodos tradicionais. Neste ponto também encontramos otimização na capacidade de testar hipóteses, isto é, assumindo o histórico relativo a determinado facto, o que aconteceria se determinada ação fosse tomada. A capacidade de analisar alterações de parâmetros do negócio e fazer previsões é também uma arma de melhoria contínua de negócio e de qualidade de gestão (Salvatore Parise, 2012).

Outra estratégia, também possibilitada pela capacidade de análise rápida, tem como base a publicidade. A publicidade é uma ferramenta poderosa e é alvo de investimentos bastante significativos. Esta tem vindo a evoluir e a tendência é para que se mostre ao cliente cada vez mais apenas aquilo que é mais provável que ele compre. Este crescimento pode ser obtido de várias formas, como é o caso de registos, pesquisas ou a própria localização geográfica de cada individuo. A acrescida capacidade de fazer análises cruzadas de diferentes tipos de informação, orientadas ao cliente e às suas práticas, é também uma mais-valia otimizada da mesma forma. A *Google*, por exemplo, utiliza conceitos semelhantes na sua funcionalidade do *AdSense* de forma procurar o conteúdo adequado a representar para o utilizador (cliente) enquanto navega. Fatores que suportam este tipo de metodologias são os estudos que indicam que o intervalo de tempo decorrido entre a pesquisa por um produto/serviço e a sua compra efetiva é cada vez menor, logo a capacidade de compreender qual a publicidade correta a mostrar em tempo quase real fará cada vez mais a diferença (Salvatore Parise, 2012).

Uma outra possibilidade para explorar a rápida análise dos grandes volumes de dados é a análise dos dados sensoriais, o que se coaduna com a *Internet of Things*. Dados de sensores podem ir desde dados de GPS adquiridos dos telemóveis pessoais até dados de *smartmeters*, tal como no exemplo da EDP, e tem aplicabilidade em diversas áreas desde aplicações domóticas, passando por gestão inteligente de edifícios até operações de produção orientadas a sensores. O volume dos dados, a velocidade associada e a sua variedade são critérios importantes e cada vez se afirma mais a necessidade de os ter em conta no desenvolvimento das soluções.

Como pode ser percebido pelos exemplos, as características dos dados gerados nestes contextos são cada vez mais desafiantes e tem-se assistido a uma dificuldade crescente de processar estes dados por parte das tecnologias tradicionais. Pela necessidade e oportunidade identificadas, começou um grande investimento no desenvolvimento de um conjunto de tecnologias capazes de dar resposta a estes novos desafios. Uma das maiores iniciativas é da *Apache Software Foundation* (ASF) que começou por suportar o desenvolvimento da *Hadoop framework*. A esta seguiram-se muitas outras tecnologias que acabaram por formar um ecossistema que, conjugado, apresenta soluções tecnológicas para tratar grandes volumes de dados, a altas taxas de transferências e com grande variedade de formatos. Neste ecossistema, cada tecnologia disponível foi desenhada para resolver algum problema específico do domínio

de Big Data, como a recolha eficiente de grandes volumes de dados em tempo real, o armazenamento distribuído e não rígido quanto às estruturas ou formatos de dados, a disponibilização em tempo real de dados processados, a criação de novos métodos de visualização dos dados, entre outros. A análise dos dados com estas características obrigou, portanto, a um afastamento do *mindset* tradicional e ao desenvolvimento de novas abordagens para recolher, processar e representar os dados. Neste sentido, e com o intuito orientar o desenvolvimento das novas aplicações de processamento de dados no contexto de Big Data, surgiram arquiteturas de referência que, sendo agnósticas às tecnologias usadas, ajudam a abordar os problemas, ao definir um conjunto de boas práticas a seguir na implementação de fluxos de dados.

De forma a analisar com detalhe as metodologias de Big Data é realizado este estudo, que tem como objetivo analisar quais as tecnologias e arquiteturas consideradas mais relevantes, tendo como referência as tecnologias do ecossistema *Apache*. Estas são *Open Source*, têm uma comunidade considerável que as suporta e, entre outras características, são pensadas para executar em *comodity hardware*[3], que corresponde ao investimento que pequenas e médias empresas estarão dispostas a fazer, mesmo com consciência das limitações que isso pode implicar. As abordagens seguidas pela comunidade estão moderadamente padronizadas, e existem algumas arquiteturas de referência, como as Arquitetura *Lambda* ou *Kappa*. Estas abordagens serão analisadas de forma a comparar as várias soluções disponíveis tanto a nível de eficiência como de dependência de recursos, variando as arquiteturas de referência e a forma como é feita a preparação do seu ecossistema, tendo em vista aplicações de análise sensorial. Sempre que possível, a ideia será testar as arquiteturas de referência, tendo em conta limitações existentes, em particular relativas ao *hardware* disponível, o tempo de integração que as tecnologias naturalmente requerem e a falta de experiência nas tecnologias a abordar.

1.2 Abordagem Metodológica e Organização do Documento

A abordagem a seguir consiste em fazer um estudo preliminar do que se considera Big Data em termos gerais, das situações onde este é aplicado de forma a compreender melhor a extensão do seu uso e quando deverá, ou não, ser a solução a adotar. Em conjunto com o estudo do conceito será feita uma análise do estado-da-arte do Big Data. Aqui serão seguidos alguns dos mais importantes casos de uso de referência, as redes sociais, e casos de em empresas de desenvolvimento e consultoria como é o caso da *Cloudera* ou da *Hortonworks*. Seguidamente serão analisadas as arquiteturas de referência juntamente com o levantamento das tecnologias aplicáveis. Por fim, tanto as arquiteturas de referência como as tecnologias serão analisadas com testes práticos, de forma a propor uma arquitetura conceptual adequada a sistemas de análise sensoriais.

[3] *Commodity Hardware* – expressão para indicar computadores ou outros dispositivos economicamente acessíveis, que é facilmente encontrado à venda e substituível por equipamentos semelhantes.

2 Big Data: Evolução dos Sistemas de Dados

Este capítulo apresenta o que se entende por Big Data, quais as suas definições, com breve referência ao consenso entre elas, e quais as suas características abordadas na bibliografia. A secção das características abrange também a mudança de paradigma trazida pelo Big Data, e uma breve análise de valor. São aqui abordadas as práticas de entidades de conhecidas no mercado que usam Big Data.

2.1 Definição do Conceito de Big Data

É devido à crescente disponibilidade e capacidade de guardar dados que surge o conceito Big Data. Não existe uma definição unívoca do conceito, no entanto, têm surgido várias abordagens ao longo dos últimos anos. As definições mais referenciadas em geral comungam na ideia de que a definição de Big Data assenta nas características dos dados e na limitação das tecnologias tradicionais para transformar esses dados em valor, de acordo com os requisitos do sistema. A definição do conceito varia ainda de acordo com a área que o está a definir (Danesh, 2014).

A *Gartner Inc*., uma empresa de investigação em tecnologias de informação, numa análise levada a cabo por Mark A. Beyer e Douglas Laney, define Big Data como:

> "Big data is high-volume, high-velocity and high-variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making" (Danesh, 2014).

Cukier e Mayer-Schönberger, no seu livro "Big Data: A Revolution That Will Transform How We Live, Work, and Think", assumem que:

> "Big data refers to things one can do at a large scale that cannot be done at a smaller one, to extract new insights or create new forms of value, in ways that change markets, organizations, the relationship between citizens and governments, and more." (Danesh, 2014).

Jason Bloomberg do portal *DevX*, um dos maiores portais técnicos da área do desenvolvimento de aplicações, tem uma visão mais técnica do conceito e define-o da seguinte forma:

> "Big Data: a massive volume of both structured and unstructured data that is so large that it's difficult to process using traditional database and software techniques." (Bloomberg, 2013).

Edd Dumbill, diretor do programa *O'Reilly Strata* e do *Open Source Convention Conferences*, vai de encontro à definição de Bloomberg que destaca a incapacidade dos métodos tradicionais ao definir Big Data como:

> "Data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or doesn't fit the structures of your database architectures. To gain value from this data, you must choose an alternative way to process it." (Edd Dumbill, 2012).

Um estudo sobre a definição de Big Data, dirigido por Andrea De Mauro e Marco Greco, concluiu que a definição mais consensual seria:

> "Big Data represents the Information assets characterized by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value." (De Mauro et al., 2015).

Há ainda abordagens mais concentradas na criação de valor como por exemplo a da IDC (*International Data Corporation*) que num relatório de 2011 constatou que:

> "Big data technologies describe a new generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data, by enabling high-velocity capture, discovery, and/or analysis." (John Gantz, 2011).

Atendendo às definições, quando numa organização se coloca a questão "*Como transformar dados em valor em tempo útil?*", a análise necessária para a resposta deverá passar pelas características dos dados, pelo que se entende por valor e o que consideram ser "tempo útil". A resposta a essa questão irá revelar a necessidade de uso de tecnologias de Big Data ou não. Assim, e ao contrário do que a nomenclatura "Big Data" pode levar a entender, não há "números mágicos" que definam a partir de que volume, velocidade ou variedade de dados se está a falar de Big Data. Não há, nem poderá consensualmente haver pois, atendendo a que o conceito depende do contexto e objetivos organizacionais, das limitações das tecnologias

tradicionais (hardware e software "tradicional") e da frequência com que estas são atualizadas e melhoradas, não é praticável utilizar valores estáticos na definição.

2.2 Características do Big Data

A definição de Big Data é geralmente acompanhada de uma referência aos 3*V*'s: Volume, Velocidade, Variedade, e diz respeito à capacidade de analisar os dados em tempo útil para a aplicação em causa. Algumas outras abordagens, mais recentes, vão além da referência aos 3*V*'s e introduzem um quarto *V* que, dependendo dos autores, pode variar entre a Variabilidade, a Veracidade ou o Valor. Curiosamente, alguns autores destacam ainda mais *V*'s como a Visualização, e outros.

Volume

No contexto do Big Data, o *Volume* refere-se à dimensão ou à quantidade de dados. O benefício proveniente da capacidade de processar grandes quantidades de dados é uma das principais atrações do Big Data e promete mudar o conceito de vários modelos de análise dos dados. Representa também um desafio para infraestruturas de IT convencionais, uma vez que as tecnologias de Big Data estão geralmente pensadas para operar em *cluster* [4] , com armazenamento escalável e distribuído. As tecnologias de Big Data não são a única solução considerada pelas empresas, mas têm sido adotadas em detrimento ou de forma complementar a tecnologias de *Business Intelligence* (BI). Um dos fatores a pesar mais nessa decisão está relacionado com a flexibilidade natural das tecnologias de Big Data para se conseguirem adaptar com relativa facilidade a diversas fontes de informação, com uma estrutura não rígida, desde os tradicionais ficheiros Excel ou JSON, até outras estruturas, como estruturas de ficheiro proprietárias ou mesmo áudio e vídeo (Khan, Uddin, & Gupta, 2014). Por outro lado, em BI é usada a abordagem dos armazéns de dados de fonte com esquemas de dados rígidos predeterminados como bases de dados relacionais, ou ficheiros de Excel ou CSV (Edd Dumbill, 2012). Como já referido, para se considerar Big Data a partir do *Volume* dos dados não há um "número mágico". Não é praticável impor limites rígidos tendo em conta que o conceito está diretamente relacionado com os limites das tecnologias tradicionais e do *hardware*. Estes têm atualizações com melhorias regularmente, aumentando a capacidade de armazenamento e de tratar dados, pelo que se houvesse limiares no volume para se considerar Big Data, estes seriam atualizados com frequência (Oracle, 2013).

Velocidade

A disponibilidade de dados processados em quase tempo real é cada vez mais importante para os modelos de negócio. Por exemplo, uma aplicação que publicite promoções a utilizadores quando estes se aproximam de uma loja, precisa que os dados de localização do utilizador sejam processados em tempo útil, e quanto mais rápida for a resposta do sistema, maior a

[4] *Cluster* – consiste num conjunto de computadores ligados entre si, partilhando recursos, de tal forma que podem ser vistos como um único sistema.

probabilidade de converter a publicidade em vendas. Este exemplo está relacionado com o conceito de *Velocidade* do Big Data e refere-se à taxa com que se geram dados a absorver por um sistema e os requisitos de tempo para apresentar os resultados do processamento desses dados (Khan et al., 2014). Considera-se que é na velocidade com que se obtém *feedback* de uma entrada no sistema, no tempo entre a chegada de um dado e a tomada de uma decisão consequente, que está a verdadeira importância da *Velocidade* no contexto do Big Data. Este processo de transformação rápida de dados em informação ou decisões é chamado na indústria do Big Data de *streaming* ou de "*processamento complexo de eventos*". Tipicamente o processo concentra-se em dar ao sistema as respostas que ele precisa para cumprir os objetivos, podendo fazer processamento sobre os dados e gerar a sua resposta mesmo antes de os armazenar, visto que quanto mais rápido se gera a resposta, maior a vantagem no mercado (Edd Dumbill, 2012).

Variedade

A *Variedade* diz respeito à estrutura/formato dos dados. Os dados podem surgir em variados formatos desde os mais tradicionais como Excel, XML, ficheiros de texto e bases de dados relacionais, até aos formatos que só recentemente começaram a ser vistos como fonte de informação para os sistemas, como áudio, vídeo, imagens, entre outros (Edd Dumbill, 2012). Esta característica que denota a flexibilidade do Big Data é muito atrativa comercialmente e abre portas para o uso eficaz do Big Data em outras áreas de negócio como análise meteorológica, onde muitos sistemas utilizam HDF5[5] ou NetCDF[6], e na área médica, onde pela primeira vez os ficheiros imagiológicos em DICOM gerados podem ser de facto integrados no sistema de informação, podendo ser processados e analisados por rotinas automáticas. Contudo, a *Variedade* introduz também um problema que se tem constatado que é, com o aumento da *Variedade* num sistema, decresce a taxa de integridade dos dados (Khan et al., 2014).

O *Volume*, a *Velocidade* e a *Variedade* são os principais 3V's do Big Data e também os mais consensuais surgindo em todas as referências. Há, no entanto, mais definições que assumem um quarto *V*. Este quarto *V* varia na normalmente entre *Veracidade*, *Variabilidade* e *Valor*.

Veracidade

A *Veracidade* é introduzida pela *IBM* e pelo *LinkedIn* como o quarto *V* e diz respeito à qualidade dos dados, a sua correção e precisão (Danesh, 2014; Marr, 2014). Com frequência, os sistemas onde se usa Big Data aceitam *inputs* humanos sem qualquer tipo de controlo ou obrigatoriedade de formato. A filosofia seguida no conceito dos armazéns de dados exige que seja garantida (o mais possível) a qualidade dos dados inseridos e, para isso, socorre-se de práticas como o ETL. O Big Data não é compatível com essa filosofia, tal como se pode perceber ao analisar 2 dos 3*V*'s fundamentais: a *Velocidade* e a *Variedade*. A *Variedade* é um fator que,

[5] *HDF5* (*Hierarchical Data File 5*) - é um modelo de dados de estrutura hierárquica capaz de representar vários objetos complexos num único ficheiro, com qualquer tipo de metadados associados.
[6] NetCDF – é um conjunto de bibliotecas e formatos de dados que suportam a criação, acesso e partilha de dados científicos orientados a vetores.

por si só, influencia a qualidade dos dados. Tal como já foi referido, a *Variedade* dos dados num sistema coloca em causa a sua integridade pois são precisas adaptações para compatibilizar com os diferentes formatos e quanto mais personalizações, maior o risco de erro. A *Velocidade* faz com que não se coloque em hipótese apresentar processos de ETL pois a natureza do sistema quando recebe nova informação é processá-la para gerar uma resposta, logo o tempo disponível para esta ação não é normalmente compatível com processos ETL. (Danesh, 2014; Marr, 2014)

Variabilidade

A *Variabilidade* refere-se diretamente à inconsistência dos dados, e é vista de duas perspetivas: estrutura e interpretação. Quando vista pela perspetiva da estrutura, o conceito refere-se à variação na estrutura dos dados, por exemplo a variação na estrutura de registos, formatos de imagem, estruturas de mensagens de redes sociais, entre outros, ou seja, reflete-se no quanto e como a estrutura dos dados pode variar. Por outro lado, pode também ser vista da perspetiva da interpretação, em que uma mesma estrutura pode ter vários significados diferentes dependendo do contexto. Isto é mais percetível quando se fala de semântica, por exemplo, onde, dependendo do contexto, uma mesma palavra pode assumir diversos significados válidos. Por exemplo, no caso das redes sociais, uma palavra em comentários semelhantes pode assumir significados completamente diferentes (Demchenko, De Laat, & Membrey, 2014).

Esta característica pode ser confundida ou mesmo considerada um problema associado à *Variedade*. Analisando os conceitos, é possível pode constatar que são características diferentes e podem ser vistas como consequência da evolução dos sistemas e meios de informação ou, generalizando, da sociedade. As necessidades dos produtores da informação que alimentam o Big Data variam, logo é provável e inevitável que a estrutura dos dados varie com o tempo. O mesmo é válido para o significado dessa informação. Um exemplo simples será pensar nas redes sociais e no número diferente de formas como as pessoas podem exprimir a sua opinião acerca de uma marca. Uma análise estritamente computacional torna-se falível nesta análise, não só pela quantidade de possibilidades, mas também porque o número de possibilidades irá variar com o decorrer do tempo, isto para não falar de expressões humanas como o sarcasmo, que nem sempre são, à primeira vista, claras.

Valor

Muitos autores como, por exemplo, a *Enterprise Architects*, a *TIBCO* ou a *CISCO,* introduzem o *Valor* como o quarto *V*. Há ainda outras empresas, como a IBM e *LinkedIn*, que o introduzem como quinto *V*. O *Valor* no contexto do Big Data não é diretamente uma característica técnica do Big Data, mas sim o objetivo máximo do seu uso (Khan et al., 2014). É algo que pode ser associado a qualquer conjunto de dados, e não necessariamente apenas a Big Data. No entanto, caracteristicamente, o *Valor* inerente ao Big Data tende a ser incremental, em particular devido ao detalhe da informação armazenada que, com o passar do tempo, permite ter uma visão não só global mas, por exemplo, por cliente. Em contrapartida, o Big Data tende também a ter uma baixa densidade de valor (Chen, Chiang, & Storey, 2012), ou seja, para obter o mesmo nível de informação são armazenados muito mais dados (Vorhies, 2014).

As características descritas acima são as mais comuns e as que habitualmente surgem entre as primeiras listadas pelos diferentes autores. Há, no entanto, autores que definem até 7, ou mesmo 12, outras características defendidas que incluem *Visualização* ou *Volatilidade* (Maheshwari, 2015; McNulty, 2014). A *Volatilidade* é relativa ao período durante o qual os valores devem ou podem ser retidos como, por exemplo, dados com um tempo útil de apenas de alguns segundos. Outro exemplo em casos de integração é o relativo aos acordos bilaterais, que podem autorizar a "persistência" dos dados apenas durante algum tempo, o necessário para gerar, por exemplo, respostas de exercícios de *nowcasting* (Lorentz, 2013). Esta característica é abordada pelas tecnologias existentes e é referida como TTL, *Time-To-Live* ou tempo de vida, em português (Madappa, 2013).

2.3 Análise de Valor

Valor pode ser visto como todo um conjunto de fatores inerentes a um produto ou serviço capazes de levar à preferência do consumidor por este. Estes fatores podem ser identificados como vantagens para o consumidor sob várias formas como poupar tempo ou dinheiro, mas também pode ser visto sob a formas de outras vantagens como o *status* associado a quem usa aquele produto, ser ecológico, ser novidade, as próprias funcionalidades associadas, entre outros. A perceção que um consumidor tem sobre estas vantagens é o **valor percebido**. Diferentes consumidores têm uma visão diferente do valor de um produto por vários motivos. Os gostos pessoais e experiências são diferentes e, como tal, a fidelidade à marca pode estar ligada a esta preferência. Por outro lado, também as necessidades de cada indivíduo são diferentes e, portanto, a funcionalidade disponível num produto é também um critério de preferência, que eleva ou não o valor percebido por cada consumidor de um determinado produto.

Posto isto, todos os produtos devem ser únicos e diferentes entre si e, assim sendo, há a necessidade de levar as vantagens de cada um ao seu cliente alvo. Surge aqui a **proposição de valor** (MaRS, 2012) que é a promessa de valor de um produto ou serviço dada ao consumidor. Esta deve consistir, pelo menos, na definição clara e concisa do produto, no seu público-alvo e em constatar como é que o produto é único e inovador. Esta mensagem de promoção do produto é utlizada nos meios de promoção e na imagem do produto e tem de ser objetiva e clara. Por exemplo o *Skype,* que pretende passar como valor a sua simplicidade de uso, mostra na imagem de promoção crianças a usar o *software* para mostrar o quão simples e funcional é. Tal como no caso do *Skype*, as empresas utilizam a proposta de valor para criar estratégias de marketing e definir como abordar os potenciais consumidores da forma mais vantajosa e persuasiva. A proposta de valor é assim o veículo de promoção de um produto, a partir do qual um consumidor pode ganhar preferência por este em detrimento da concorrência.

2.3.1 Proposta de Valor

O valor do Big Data varia bastante consoante a área de negócio, essencialmente porque as áreas em si têm um aproveitamento diferente dos dados. Há a necessidade de analisar alguns fatores preponderantes, nomeadamente a capacidade de gerar informação, a capacidade de a recolher, a necessidade de elevado desempenho na análise dos dados e os custos destas tecnologias para as diferentes áreas em comparação com as tecnologias já existentes. Geralmente, todos estes fatores são influenciados pela maturidade tecnológica de cada área no que diz respeito à gestão de informação (Brown, Manyika, et al., 2011). Isto é, as áreas que usam mais intensamente e desde há mais tempo as tecnologias de informação para gestão de negócio, como a área da saúde e governamental, são tecnologicamente mais maduras e, portanto, menos relutantes em investir em novas tecnologias. Por exemplo, um estudo da *McKinsey* (Brown, Chui, & Manyika, 2011) constata que uma das áreas com maior potencial de aproveitar o Big Data é a área da saúde e prestação de cuidados. Por outro lado, uma das áreas com menor potencial são os serviços educacionais. A decisão sobre qual o mercado alvo, deve ser orientada segundo um estudo que indique o potencial de aproveitamento do Big Data em cada área. A decisão deve ser fundamentada com uma prospeção das necessidades reais de cada área, do investimento que o consumidor está disposto a fazer e do potencial de extração de valor resultante das atividades de cada sector de atividade. Os setores potencialmente mais promissores concentram-se nos segmentos das tecnologias de informação e redes sociais, na área médica e de cuidados de saúde, na área governamental e Estado e na área da produção (Brown, Manyika, et al., 2011).

Atendendo a que o objetivo do estudo é analisar as tecnologias e arquiteturas padrão de Big Data, o maior valor encontrado consiste no exercício de análise realizado às várias tecnologias e distribuições e no exercício de implementação e demonstração de uma arquitetura. Este estudo está orientado numa perspetiva "*hands-on*", ao analisar e selecionar as tecnologias para um sistema de Big Data. Partindo de um sistema para dados de sensores como caso de uso, é feita uma análise das arquiteturas padrão usadas neste tipo de sistemas, as quais são demonstradas e comparadas. O exercício de análise aqui apresentado, oferece um ponto de partida para quem queira começar a trabalhar com este tipo de tecnologias. De forma genérica, em todas as áreas de negócio, a decisão sobre adotar ou não tecnologias de Big Data passa por reconhecer as necessidades atuais e conhecer que tecnologias existem e com que finalidades são usadas. É também recorrente numa primeira fase a implementação de um sistema piloto baseado numa distribuição com suporte da comunidade, por parte de quem pretende investir na área. Como é também natural, procura-se sempre que este piloto seja conseguido com o mínimo de custos. É em agilizar este processo que se encontra a proposta de valor deste estudo, na disponibilização de uma análise preliminar e de um piloto de um sistema resultante do exercício prático do caso de uso, bem como, na disponibilização de uma análise das principais tecnologias e distribuições de Big Data, que facilitará a sua aplicação e diminuirá a curva de aprendizagem para o desenvolvimento de casos de uso.

2.3.2 Modelo de Canvas

No contexto do presente estudo é possível identificar os diversos componentes constituintes do modelo de Canvas:

Parceiros Chave

Começando pelos "Parceiros Chave", as entidades mais importantes identificadas são os provedores dos componentes do sistema e das distribuições – *Apache Software Foundation*, *Hortonworks*, *Cloudera*, *MapR* -, provedores de dados livres que potencialmente poderão ser usados na simulação e testes do sistema e, eventualmente, entidades de investigação mutuamente interessadas no desenvolvimento e manutenção de sistemas de Big Data que suportem as análises complexas de dados.

Atividades Chave

As "Atividades Chave" podem ser consideradas: o desenvolvimento dos sistemas de demonstração, a configuração e manutenção da infraestrutura de equipamentos a usar (virtuais ou não), as análises comparativas entre as arquiteturas e as análises comparativas (teóricas) das distribuições da *Hortonworks*, *Cloudera* e *MapR*.

Recursos Chave

Os "Recursos Chave" a apresentar correspondem ao conhecimento do Ecossistema *Hadoop*. Atendendo à complexidade e quantidade das tecnologias disponíveis, é necessário ter uma compreensão global do ecossistema e compreender que funcionalidades ou grupo de funcionalidades, as principais tecnologias disponibilizam e como estas se integram ou como geram conflitos com outras. A experiência em análise de dados é também importante para a execução do estudo. É necessário, mais que adquirir e comparar os dados, validar a qualidade dos dados e quão válidas são as conclusões a que estes permitem chegar.

Proposta de Valor

A "Proposta de Valor" deste estudo foca-se na disponibilização de um sistema personalizável para aplicações de monitorização e controlo de acordo com o caso de uso (sistema de análise de dados sensoriais), justificando este caso de uso as arquiteturas usadas nos sistemas de teste. Outro componente da proposta de valor consiste na análise das distribuições e tecnologias de Big Data. Este estudo é comum quando se está a planear um sistema e, portanto, ajudará a reduzir as curvas de aprendizagem.

Relacionamento

Para o tópico "Relacionamento" do modelo de Canvas compreenda-se o meio de comunicação e a discussão dos temas a abordar no estudo. Estes consistem essencialmente em plataformas de discussão *online,* como fóruns de discussão.

Canais

Considerando os "Canais" para distribuição dos conteúdos do estudo, as apostas seriam em distribuição do conhecimento *online* através de *Blogs* e/ou fóruns. Pode-se considerar também *feeds* de notícias da especialidade ou bibliotecas de artigos *online*.

Segmento de Clientes

O segmento de clientes esperado é composto por "*Data Scientists*[7]" e "*Data Engineers* [8]", visto serem as profissões com mais proximidade à área e com maior interesse neste tipo de estudos.

Estrutura de Custos

Aquisição e manutenção do equipamento necessário.

Fontes de Receita

A fonte de receita, atendendo a tratar-se de um estudo, consiste no valor intangível gerado durante o desenvolvimento do estudo, ou seja, toda a transmissão de conhecimento para a comunidade e *know-how* gerado.

[7] *Data Scientists* – profissionais da área do Big Data dedicados exclusivamente à exploração de dado
[8] Data Engineers – profissionais da área do Big Data dedicados ao desenvolvimento e gestão das infraestruturas de Big Data

3 Tecnologias e Arquiteturas de Big Data

Neste capítulo é abordado o estado da arte das tecnologias e arquiteturas de referência mais comuns usadas em contexto de Big Data. É feita por uma breve abordagem ao ecossistema *Hadoop* e às tecnologias mais comuns por cada uma das principais responsabilidades dos sistemas de Big Data aqui abordadas: integração de dados, armazenamento e processamento distribuído. Depois é analisado o conceito de *Stack*, as principais distribuições de ecossistemas *Hadoop*, as arquiteturas padrão para desenvolvimento de casos de uso. Por fim são analisados alguns casos de uso publicados por algumas das redes sociais mais conhecidas para mostrar em exemplo a aplicabilidade de tudo o foi analisado anteriormente no capítulo.

3.1 Análise de Tecnologias de Big Data

Neste capítulo é feito um levantamento e análise de algumas das tecnologias do ecossistema *Hadoop* mais utilizadas e por abordar em que consiste a *Hadoop Framework*. A análise passa de seguida pelas tecnologias utilizadas para *integração de dados*, onde se considera quais tecnologias são aplicáveis para aceder e integrar sistemas heterogéneos. Serão também analisadas tecnologias usadas para armazenamento e, por fim, serão abordadas as principais tecnologias usadas para processamento de dados.

3.1.1 Ecossistema *Hadoop* e Categorização de Tecnologias

O ecossistema *Hadoop* apresenta uma segmentação das tecnologias que mantêm por categoria [9]. Esta é uma prática que facilita não só a gestão por parte da comunidade desenvolvedora, mas também facilita a pesquisa de tecnologias aplicáveis a aplicações específicas por parte dos "consumidores" destas tecnologias. O ecossistema, apesar de estar em constante evolução, está essencialmente segmentado nas seguintes categorias:

- *Sistema de ficheiros distribuído*: esta categoria contém tecnologias como o *HDFS* ou o QFS (*Quantcast File System*), entre outros sistemas de ficheiros distribuídos. Estas tecnologias geralmente são mapeadas para os componentes base das *Stacks* usadas nos sistemas de Big Data;
- Programação Distribuída: a categoria de programação distribuída apresenta o maior número de tecnologias com maior suporte da comunidade como o *MapReduce*, o *Pig*, o *Spark*, o *Storm*, o *Flink*, o *Tez*, entre outras. Todas estas são, ferramentas usadas para programação distribuída, estando mais ou menos otimizadas para a pesquisa de dados (como o *MapReduce*) ou para o processamento em *streaming* (como o *Storm*). Esta

[9] Conferir em: https://hadoopecosystemtable.github.io/

categoria apresenta tecnologias mapeadas para os componentes de escalonamento e fluxo de dados;

- Bases de dados: Existem 3 categorias de bases de dados:
    - *NoSQL*: esta categoria subdivide-se ainda em modelos de dados orientados a colunas, documentos, *key-value*, *stream* e grafos. Cada um destes modelos apresenta uma série de tecnologias também com suporte ativo da comunidade e muito referenciadas em soluções comerciais como o *HBase*, o *Cassandra* ou o *Accumulo*;
    - *NewSQL*: esta categoria apresenta tecnologias menos conhecidas, mas que prometem aos poucos começar a ser usadas em soluções comerciais atendendo a que parte delas são componentes complementares a outros sistemas de bases de dados relacionais, como o *TokuDB* ou o *HandlerSocket* que são complementares ao *MySQL*;
    - *SQL-On-Hadoop*: esta categoria apresenta também alguns sistemas de armazenamento complementares, direcionados para a infraestrutura *Hadoop*, sendo maioritariamente componentes integrados com o *HDFS* para melhorar a performance ou modificar a filosofia de armazenamento ou acesso. Aqui apresentam-se tecnologias bastante conhecidas como o *Hive*, o *HCatalog*, o *Drill*, o *Phoenix*, entre outros também populares;
- Integração de Dados: A integração de dados corresponde a um dos componentes chave das *Stacks*. Este determinará muitas vezes a capacidade de absorção de dados e consequentemente influencia a performance de todo o sistema. Aqui apresentam-se algumas tecnologias bastante conhecidas como o *Flume*, o *Sqoop*, o *Kafka*, o *Samza*, entre outros;
- Programação de Serviços: nesta categoria encontram-se tecnologias orientadas para facilitar a gestão de componentes do sistema (como cópias de segurança das filas de mensagens do *Kafka*) ou facilitar a comunicação entre componentes em funcionamento no sistema. Os componentes mais conhecidos desta categoria são, provavelmente, o *Zookeeper*, o *Thrift* e o *Avro*;
- Escalonamento e fluxo de dados: nesta categoria apresentam-se algumas tecnologias especificamente desenhadas para escalonamento de tarefas do sistema e facilitadores da gestão do ciclo de vida dos dados. Apresentam-se aqui alguns nomes também bastante conhecidos como o *Oozie* e o *Azkaban*;
- Desenvolvimento do sistema: entre as ferramentas desta categoria encontram-se as tecnologias que automatizam parte do desenvolvimento e facilitam a posterior gestão do sistema usado pelos grandes distribuidores como é o caso do *Ambari* (da *Hortonworks*) e do *HUE* (da *Cloudera*), entre muitos outros;
- As restantes categorias enquadram-se como ferramentas ou bibliotecas complementares a algumas das ferramentas das categorias anteriores:
    - *Machine Learning*: apresentam-se aqui bibliotecas como o *Mahout* ou o *Oryx*;
    - *Benchmarking* e *QA*: esta categoria introduz ferramentas de suporte às práticas de qualidade e de benchmarking como o *Hadoop Benchmarking*;

- Segurança: esta categoria apresenta os principais componentes de segurança e gestão de acesso aos sistemas de Big Data como o *Knox Gateway* e o *Ranger*, que implementam controlo de acesso.

3.1.2 Apache *Hadoop*

O *Apache Hadoop* é um dos projetos mais relevantes da *Apache Software Foundation* e consiste numa *framework* desenvolvida para executar processamento distribuído de conjuntos de dados (*datasets*) de grandes dimensões, em *clusters* de computadores. A filosofia desta *framework* procura fazer com que a sua performance não dependa de estar instalada num supercomputador, pois foi construída a pensar em alta disponibilidade, mas sem descuidar a gestão autónoma de falhas e a escalabilidade. Podem por isso ser usados *clusters* de *commodity hardware* para suportar estes sistemas (Apache Software Foundation, 2014).

O *Hadoop framework* consiste em quatro módulos fundamentais, que constituem o núcleo para o ecossistema de aplicações e sistemas (Bhandarkar, 2012). Esses módulos são:

- *Hadoop Common* – é composto por um conjunto de funcionalidades que suportam todos os outros módulos. Providencia abstração da interação entre o sistema operativo e o sistema de ficheiros, e inclui os executáveis e *scripts* necessários para a gestão básica do *Hadoop*;
- *Hadoop Distributed File System (HDFS)* – consiste num sistema de ficheiros distribuído, otimizado para permitir altas taxas de transferência. Este é, portanto, usado essencialmente para armazenar grandes quantidades de dados de forma confiável, devido à tolerância e recuperação automática a falhas e permitir acessos de leitura com grande performance. Ao distribuir o armazenamento e a computação por vários servidores, e com a capacidade de adicionar servidores ao *cluster*, é possível escalar o sistema e aumentar os recursos disponíveis de acordo com as necessidades da aplicação (Kulkarni & Khandewal, 2014). O *HDFS* será melhor explorado no próximo subcapítulo;
- *Hadoop MapReduce* – é definido como um modelo de programação simplificado e é usado no *Hadoop* para processamento paralelo de grandes quantidades de dados. Este consiste num nó-mestre, *JobTracker*, e vários *TaskTrackers*. As ações submetidas são alocadas ao *JobTracker* que, por sua vez transforma cada tarefa em várias sub-tarefas de *Map* e *Reduce*. Os *TaskTrackers* monitorizam a execução das tarefas, *Map* e *Reduce*, e notificam de que elas terminaram. O *Map* consiste no passo de "entrada e transformação" dos dados, onde os dados de entrada são processados em paralelo. O *Reduce* consiste no passo de "agregação" onde os resultados do *Map* são processados em conjunto (Venner, 2009);
- *Hadoop YARN (Yet Another Resource Negotiator)* – Apesar de ser uma solução *standard* neste momento para resolver questões de Big Data, a tecnologia *Hadoop* ainda possui algumas limitações. Um exemplo consiste na limitação do *MapReduce* só conseguir operar sobre 4000 nós (Murdopo & Dowling, 2013). O *YARN* surge para resolver estas

limitações de escalabilidade através da separação de funcionalidades na medida em que define um gestor de recursos global que se divide em dois componentes essenciais: o escalonador, que aloca recursos às aplicações, e o gestor de aplicações, que aceita pedidos de execução de tarefas e garante a execução da tarefa. Com o *YARN,* várias aplicações podem partilhar uma mesma gestão de recursos (Kulkarni & Khandewal, 2014).

3.1.3 Integração de Dados

O acesso a dados heterogéneos e a consequente integração dos mesmos tem sido um tema cada vez mais relevante.

A geração de informação evoluiu consideravelmente ao longo dos últimos anos (Maier, 2013). Nos anos 90, as transações bancárias, registos de comerciais e arquivos governamentais eram os principais produtores de dados. A década seguinte testemunhou o grande impulso dos motores de pesquisa, como a *Google* e o *Bing*, de aplicações de comércio online e também das redes sociais. Posteriormente, surgiu um novo fator que marcou o início de uma nova era de produção de dados, o uso dos *smartphones* como meio de acesso a plataformas e aplicações em rede.

No contexto do Big Data, os dados a considerar para um sistema estarão sempre ligados ao domínio em questão e são essencialmente divididos em duas categorias fundamentais: estruturados e não estruturados. Os dados estruturados referem informação com um grau de organização próximo do possível numa base de dados relacional. Considere-se, por exemplo, estruturas de informação como as bases de dados relacionais e ficheiros de texto com formatos padrão como CSV, XML/JSON ou até folhas de cálculo. Estima-se que de toda a informação gerada, cerca de 20% sejam dados estruturados (Nemschoff, 2014). As tecnologias de Big Data potenciaram a exploração de tipo de dados estruturados adicionais como os dados sensoriais, como o GPS, dados de linhas de produção ou HL7[10]. Por sua vez, os dados não estruturados são todos os outros tipos de formato dos quais não é linear a extração do significado, como é o caso de imagens, vídeos/áudio, apresentações, documentos de texto (que não usem formatos padrão), PDF, ou mensagens. Por exemplo, um email tem campos que podem levar a crer que se trata um objeto estruturado como os remetentes, o assunto ou o corpo. No entanto, uma grande parte da informação de um email reside nos conteúdos do corpo do mesmo e se for visto como um campo de texto comum, estaremos a desperdiçar informação. O mesmo acontece com as mensagens das redes sociais. Por exemplo o *Twitter* faz análise e deteção de emoções com base no texto dos *tweets* (Kumar, 2013).

Há, portanto, uma necessidade crescente de integrar informação proveniente de fontes diferentes, com diferentes formatos, em diferentes contextos. Um padrão que se encontra na génese de várias soluções adotadas consiste na utilização de mediadores (ou *wrappers*) associados a camadas de abstração que permitem ao sistema ver de forma transparente a

[10] HL7, ou *Health Level Seven*, é um padrão para estruturar dados médicos

informação recebida. Este tipo de soluções reduz o esforço de adaptação de um sistema a novos formatos de informação, melhorando a flexibilidade e escalabilidade do sistema. Este problema é abordado pelas características da *Variedade* e da *Variabilidade* do Big Data e é um dos fatores a considerar quando se planeia que tecnologias usar num sistema.

3.1.3.1 Apache Kafka

O *Kafka* é um projeto da *Apache Software Foundation*, desenvolvido com o objetivo de garantir baixa latência na rede e alta taxa de transmissão. É um sistema de transmissão de mensagens distribuído que segue o padrão *publisher-subscribe*. Neste padrão, os emissores (*publishers*) limitam-se a enviar a mensagens para o sistema, categorizadas por determinada classe. Os recetores (*subscribers*) adquirem as mensagens disponíveis no sistema de acordo com as classes que lhes interessam. Este método abstrai a necessidade dos emissores e dos recetores terem conhecimento uns dos outros e de conseguirem comunicar diretamente (Garg, 2013).

As aplicações de *Kafka* conseguem tratar milhares de mensagens em intervalos reduzidos de tempo. Para evitar erros e perda de mensagens, o *Kafka* possui um sistema de cópias de segurança (*backup*) de mensagens que previne a perda de dados, mesmo em caso de falha do sistema. Este sistema consegue manter um desempenho constante, mesmo com grandes volumes de dados. Também é possível a partição de mensagens através de diferentes servidores e a distribuição de computação através de um *cluster* (Kreps, Narkhede, & Rao, 2011).

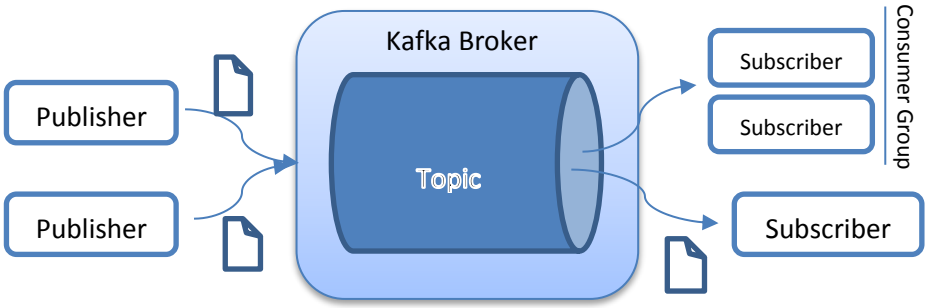


Figura 1 - Esquema da de funcionamento do *Kafka*, adaptado (Thein, 2014)

O desenvolvimento do *Kafka* teve sempre em vista o elevado desempenho do sistema e, portanto, a sua arquitetura permite que as mensagens recebidas dos *publishers* fiquem de imediato visíveis pelos *subscribers*. Pelos argumentos anteriores, o *Kafka* é uma das tecnologias utilizada em sistemas críticos baseados em eventos. Consegue tratar centenas de *megabytes* de escritas e leituras por segundo em vários clientes.

Os *subscribers* são normalmente agrupados. Cada mensagem recebida no sistema pode ser "consumida" por um processo de cada grupo de *subscribers*, pelo que, se houver necessidade de mais que um processo consumir a mesma mensagem, estas têm de estar em grupos de *subscribers* diferentes. No entanto, as mensagens podem não ser apagadas imediatamente

quando são “consumidas”, permitindo a um serviço “re-consumir” a mensagem se for necessário (Garg, 2013). Não remover imediatamente a mensagem consumida permite tanto distribuir a mesma mensagem por vários *subscribers* como gerir a recuperação em caso de falha de um consumidor, sem depender de escritas em disco. A desvantagem neste caso é que esta abordagem obriga a que os consumidores tenham consciência do índice da mensagem que o *Kafka* associa a cada mensagem que distribui para evitar consumir mais que uma vez a mesma mensagem em funcionamento normal.

3.1.3.2 Apache Flume

O *Flume* foi uma tecnologia desenvolvida pela *Cloudera* em 2011, tendo sido cedida à ASF e incluída no ecossistema *Hadoop* no mesmo ano. É uma tecnologia que implementa um serviço distribuído, fiável e de alta disponibilidade com o objetivo de coletar, agregar e mover grandes quantidades de dados para o *HDFS*.

A arquitetura do *Flume* é descrita pelos autores como “simples e flexível”. Na sua arquitetura, em cada agente há, pelo menos, um componente de entrada (*source*), um de saída (*sink*), e o intermediário entre os anteriores (*channel*). Os *sources* geram eventos para os *channels* e, por sua vez, os *channels* reportam eventos para os *sinks* (Hoffman, 2013).

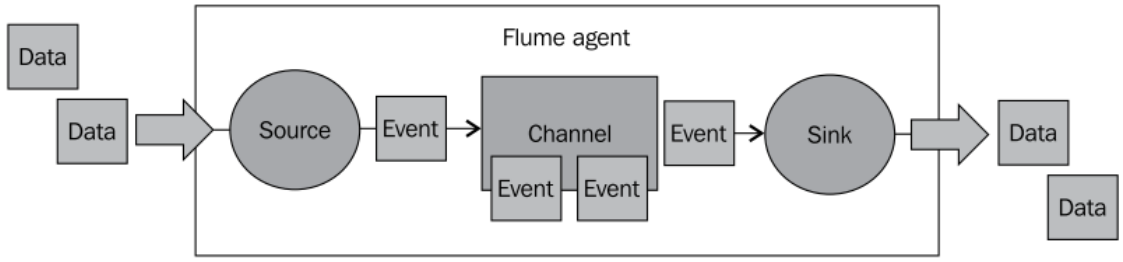


Figura 2 - Esquema geral da arquitetura do *Flume* (Hoffman, 2013)

Na perspetiva desta arquitetura, o cliente é o responsável por transmitir os eventos para o agente do *Flume*. A receção dos eventos nos agentes é feita através dos *sources* que enviam os eventos recebidos para a próxima etapa do fluxo, podendo enviar para um ou mais canais. Os canais são acedidos por um ou mais *sinks* que encaminharão os valores para a próxima etapa do sistema, podendo essa ser outro agente do *Flume* ou outro elemento do sistema como por exemplo uma instância do *Storm* ou para o *HDFS*. Os agentes implementam o padrão *publisher-subscriber* para direcionar o fluxo de informação, pelo que permitem fazer separação de partes dos eventos recebidos e distribuir cada evento por diferentes *sinks* do mesmo agente. De forma semelhante, é possível configurar um *source* para entregar o evento a um ou mais canais, como lustrado na Figura 3.

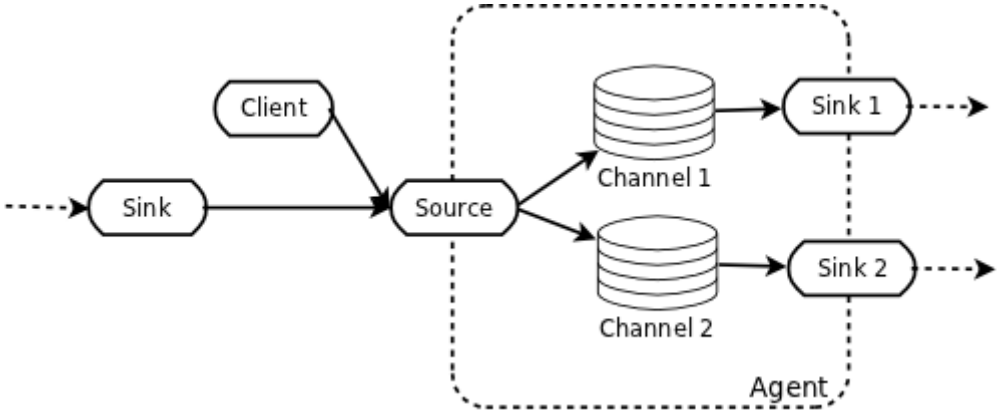


Figura 3 - Esquema da lógica de fluxo possível nos agentes do *Flume* (Prabhakar, 2011)

Para garantir que o sistema é tolerante a falhas o *Flume* utiliza o padrão *transaction-commit*. Este serve para garantir que cada evento é entregue ao próximo componente do fluxo de dados. Esta troca é baseada numa resposta relativamente ao sucesso na próxima fase do fluxo, ou seja, se o agente 1 envia um evento para o agente 2, espera por uma mensagem de *feedback* positiva antes de fechar a ação relativamente ao evento em causa.

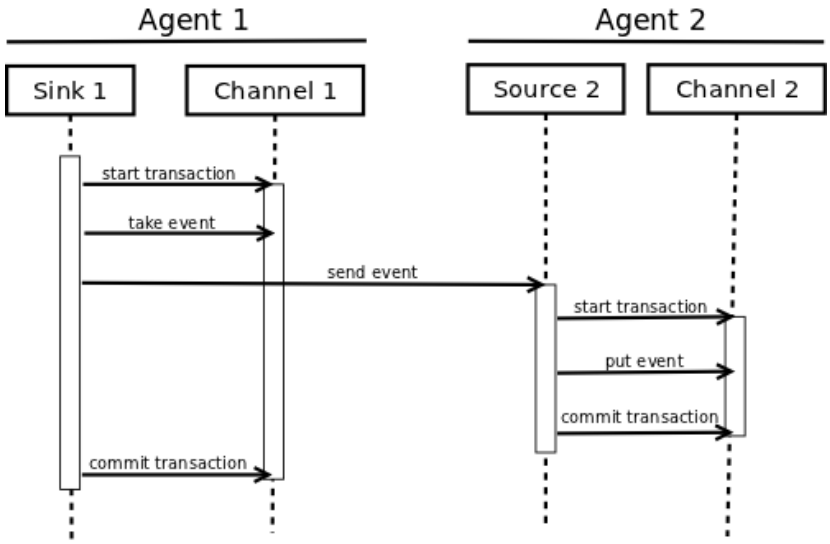


Figura 4 - Diagrama de sequência: envio de eventos entre agentes (Prabhakar, 2011)

Este mecanismo existe em cadeia ao longo dos agentes que partilham o fluxo de um evento. Em caso de falha num agente intermédio, o evento vai ficar em espera no último agente onde não ocorreu problema. Se, dentro de um determinado período de tempo, a falha não for resolvida é provável que esta se propague pelos agentes anteriores desse fluxo em sentido contrário. Quando ocorrer no primeiro agente dessa cadeia, o erro vai ser reportado ao cliente, que terá a liberdade de tomar as ações que achar convenientes. Se o sistema recuperar da falha, o fluxo é retomado, os eventos em espera são enviados e, ao fim de algum tempo, o sistema acabará por restabelecer a taxa de transferência e o funcionamento normal (Prabhakar, 2011).

O *Flume* apresenta algumas limitações. Por exemplo, dependendo do *hardware* disponível, o tamanho de cada evento pode constituir uma limitação, já que não pode ser maior do que a disponibilidade de memória na máquina que aloja o agente (Flume, 2012). O *Flume* vê, tipicamente, os conteúdos dos eventos como um conjunto de bytes apenas, pelo que também não é usado para fazer operações sobre estes (Namiot, 2015). Apesar de ser possível reimplementar tanto *sources* como *sinks* para que sejam conscientes dos tipos de dados em trânsito e para implementar operações sobre estes, tal deve ser evitado para garantir que a transmissão dos eventos ocorre dentro do período aceitável.

3.1.3.3 Apache Sqoop

O nome *Sqoop* resulta da conjugação entre os termos SQL e *Hadoop*. Assim, como se pode perceber pelo nome, é uma tecnologia otimizada para transitar informação entre bases de dados relacionais (RDB) e o *HDFS* ou outras alternativas do mesmo ambiente como o *Hive*. O *Sqoop* utiliza o *MapReduce* para importar ou exportar informação, o que lhe permite executar operações paralelas e melhorar a tolerância a falhas nas operações de leitura e escrita (Ting & Cecho, 2013).

A importação de dados com o *Sqoop* de uma RDB para o *HDFS* aceita como parâmetro uma tabela, da qual lê linha a linha para o *HDFS*. O resultado desta importação no *HDFS* é um conjunto de ficheiros que contém uma cópia da tabela em questão. O resultado pode surgir em vários ficheiros porque o processo é paralelo e dessa forma evitam-se problemas de concorrência de acesso ao disco. O formato escrito também pode variar entre formatos usuais com o CSV ou ficheiros binários serializados como por exemplo o *SequenceFiles*[11]. O formato pode ser personalizado de acordo com o pretendido, seguindo esta parte da tecnologia uma arquitetura baseada em *wrappers*. No sentido contrário, é possível com o *Sqoop* transferir dados processados a presentes no *HDFS* de volta para a base de dados relacional (Ting & Cecho, 2013).

Os conectores existentes de SQL para o *Sqoop* são o *MySQL*, o *PostgreSQL*, o *Netezza*, o *Teradata* e o Oracle e são distribuídos gratuitamente, podendo ser usados sem limitação.

3.1.4 Armazenamento

Com o crescimento dos meios de gerar e armazenar informação, tem-se assistido também ao aparecimento de várias soluções para armazenar e potenciar o processamento de dados em grande escala. Para contornar limitações e melhorar a flexibilidade dos sistemas, têm sido adotados sistemas de ficheiros distribuídos. Entenda-se por "sistemas de ficheiros distribuídos", sistemas de ficheiros capazes de gerir, através da rede e de forma transparente, o armazenamento existente em várias máquinas, abstraindo a complexidade física do sistema.

[11] *SequenceFiles* é um formato binário de ficheiro que contém estruturas de acordo com o padrão *key-value pair*.

Várias soluções com este propósito foram adicionadas ao ecossistema *Hadoop* tal como o *HDFS*, o *Hive* ou o *Cassandra*.

3.1.4.1 Apache HDFS

O *HDFS*, tal como já foi referido, faz parte dos módulos base do *Hadoop Framework* e foi desenhado para armazenar ficheiros com tamanho considerável, com padrões de acesso aos dados em *streaming*, em *clusters* de máquinas, caracteristicamente de uso genérico e, portanto, mais acessíveis economicamente (*commodity hardware*). Na bibliografia referencia-se o uso ficheiros na ordem das centenas de *gigabytes* ou mesmo *terabytes* (Bakshi, 2012).

Uma característica importante do *HDFS* é a aplicação de padrões para facilitar o acesso em *streaming*. Esta funcionalidade foi construída em torno da ideia de "*write-once, read-many-times*" (White, 2009). A filosofia de uso destes sistemas consiste na criação de um conjunto de dados, gerado ou copiado de uma fonte, que depois é sujeito a várias análises ao longo do tempo. Tipicamente, nestas análises todo o conjunto de dados é usado pelo que o tempo de leitura do ficheiro completo é mais importante que a latência de encontrar o primeiro valor a ler ou mesmo que o tempo de escrita.

O *HDFS* em si tem também algumas limitações. Não é adequado, por exemplo, para aplicações que requeiram muito baixa latência no acesso aos dados. Não é também a melhor opção quando é necessário fazer escritas com muita frequência ou atualização de ficheiros, sendo usado normalmente como uma base de dados incremental. Para cada ficheiro, chegam a ser guardados, em memória, 150 *bytes* de *metadata* (informações de sistema relativas ao ficheiro), ocupando memória do sistema de forma persistente, que deveria ser alocada, por exemplo ao processamento dos dados (White, 2009). Por isto, também não é recomendável ter aplicações que usem muitos ficheiros pequenos. O *HDFS* foi desenhado a pensar em ficheiros de grandes dimensões. Ele próprio gere o armazenamento e a segmentação dos ficheiros em blocos de 128MB (por omissão), apesar de ser um parâmetro configurável. Esta segmentação é também usada para otimizar a gestão do espaço ocupado pelas réplicas do *HDFS* através de um *cluster* e para aumentar a disponibilidade e rápido acesso aos dados, pois permite paralelizar a leitura.

O *HDFS* funciona com uma arquitetura *master-slave* e os seus componentes principais são os *NameNodes* e os *DataNodes*. Um *NameNode* é o componente responsável por gerir o *Namespace* do sistema, assim como os acessos aos ficheiros feitos pelos clientes. Os *DataNodes*, tipicamente um por *cluster*, gerem o armazenamento respetivo ao nó do sistema em questão, gerindo operações como abrir, fechar ou renomear ficheiros ou diretórios. São também responsáveis por responder aos pedidos de leitura e escrita dos clientes do sistema. Os ficheiros são separados em 1 ou mais blocos e cada bloco é distribuído por um conjunto de *DataNodes* (Liu, Bing, & Meina, 2010).

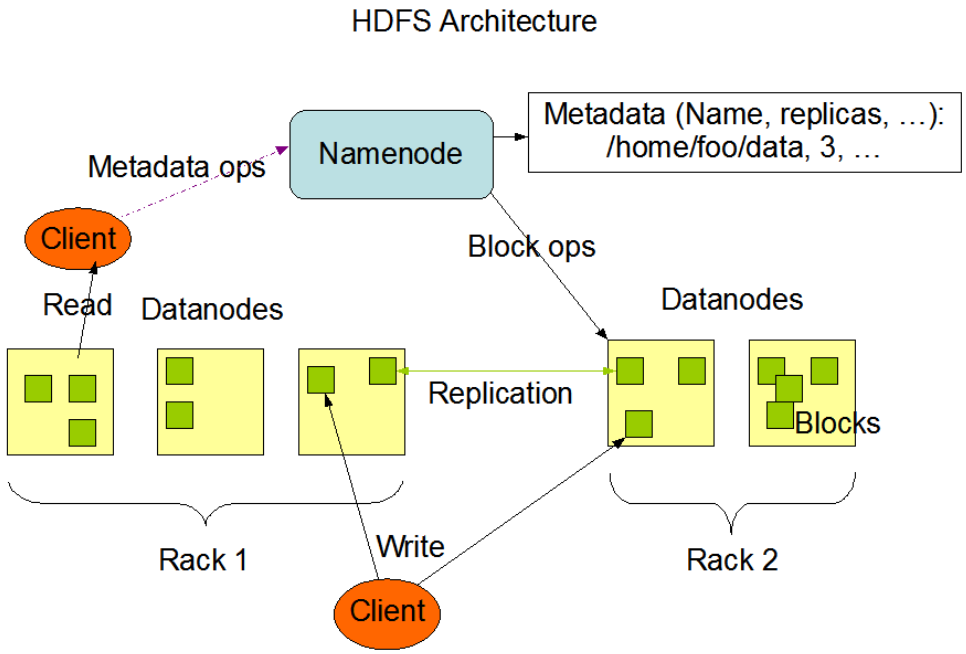


Figura 5 - Esquema da arquitetura do *HDFS* (Borthakur, 2013)

Uma das características mais importantes do *HDFS* é a sua capacidade de replicar informação e gerir essas réplicas automaticamente. Os blocos de cada ficheiro são replicados para assegurar tolerância a falhas. As decisões sobre a replicação de informação são feitas pelo *NameNode* que, periodicamente, recebe relatórios de cada *DataNode* sobre o estado de funcionamento dos blocos que este está a gerir (Kala Karun & Chitharanjan, 2013).

3.1.4.2 Apache Hive

O *Hive* é mais uma das tecnologias mais utilizadas do ecossistema *Hadoop*. Esta, à semelhança de sistemas de base de dados tradicionais, organiza a informação de forma compreensível, com estruturas como tabelas, colunas e partições, e suporta a maioria dos tipos primitivos de dados assim como tipos complexos, como mapas ou listas. As estruturas são extensíveis no sistema, ou seja, um utilizador pode adicionar tipos complexos de dados personalizados. A linguagem utilizada, *HiveQL*, é semelhante em sintaxe ao SQL, o que constitui um dos maiores motivos para a adoção desta tecnologia, pois a familiaridade com os formatos e linguagem otimiza o tempo de desenvolvimento e facilita a manutenção (Thusoo, Sarma, et al., 2010).

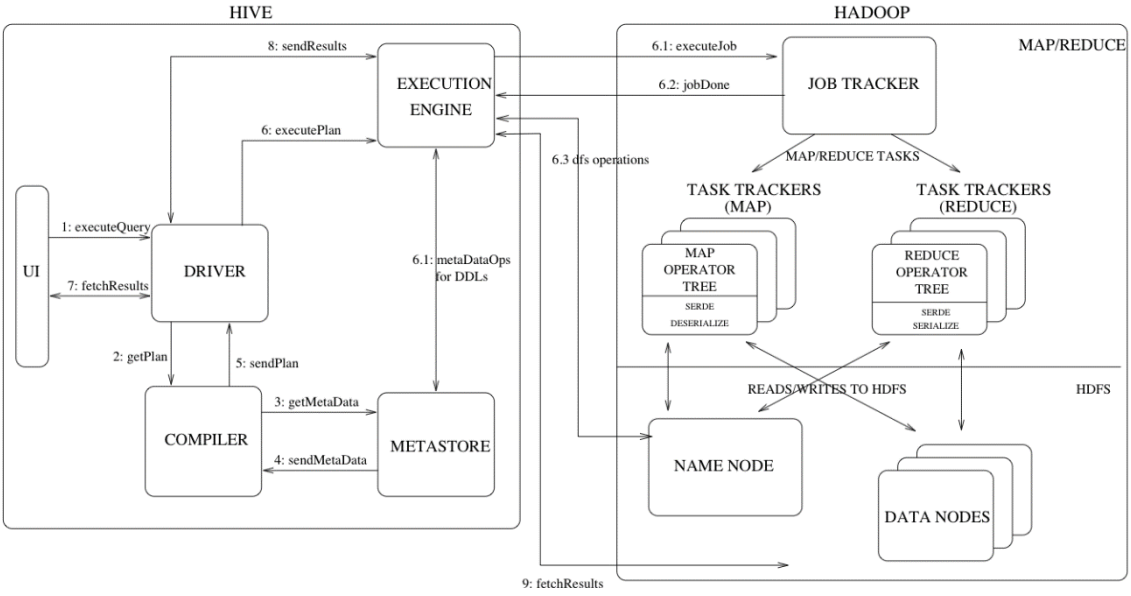


Figura 6 - Arquitetura do *Hive* e integração com o *Hadoop* (Leverenz, 2015)

Os componentes principais do *Hive*, tal como se pode confirmar na Figura 6, são:

- Interface do Utilizador – é através desta interface que os utilizadores submetem os pedidos e outras operações. A interface tem dois formatos: interface por linha de comandos e interface *web*;
- *Driver* – é o componente que recebe os pedidos vindos da interface do utilizador. Este usa APIs com base em JDBC/ODBC;
- Compilador – este componente interpreta o pedido. Faz a análise semântica e gera o plano de execução com base nos *metadata* disponíveis;
- *Metastore* – é o componente que armazena toda a informação sobre as várias tabelas e partições, como informação sobre as colunas, o seu tipo, entre outros;
- Motor de execução – é responsável por correr o plano de execução criado pelo compilador.

Pelas características e pela proximidade ao SQL, o *Hive* pode ser visto como uma das tecnologias ideais do ecossistema quando se sabe que os sistemas externos com que se vai interagir são maioritariamente *Relational Database Management Systems* (RDBMS) (Leverenz, 2015).

Apesar da facilidade permitida aos utilizadores inerente à possibilidade de utilizar SQL, o *Hive* peca no desempenho de acesso devido à necessidade de comunicar com o *Hadoop*. Os comandos enviados ao *Hive*, como se pode ver na Figura 6, são transformados em operações de *MapReduce* e escalonados para execução pelo *YARN*, o qual vai analisar os recursos disponíveis e a melhor forma de distribuir o processo antes de criar as condições para a execução do comando. A "entropia" gerada por este processo é suficiente para que o *Hive* não seja adequado para aplicações em que é necessário ter uma resposta aos pedidos em menos de 1 segundo. Por outro lado, a capacidade de expandir as estruturas usadas, e a mesma capacidade de distribuição de processamento através do *YARN*, fazem com que seja uma tecnologia ideal quando a flexibilidade e escalabilidade de uma base de dados, a longo prazo, é

tão ou mais importante do que o desempenho e a capacidade de resposta rápida a pedidos. O desconhecimento destas características é muitas vezes responsável pela escolha do *Hive* para finalidades ao qual este não se adequa.

3.1.4.3 Apache Cassandra

O projeto *Apache Cassandra*, inicialmente desenvolvido pelo *Facebook*, consiste numa base de dados híbrida, tirando partido da fusão entre dois padrões para armazenamento diferentes, o *key-value pair* e o *column-oriented*. Esta base de dados é desenhada para ser completamente descentralizada, pelo que não há um ponto de falha único (ou *single point of failure*), ao contrário do *NameNode* do *HDFS*. É capaz de gerir grandes quantidades de ações em vários nós em simultâneo. O modelo de dados do *Cassandra* permite que sejam adicionadas colunas apenas a chaves específicas, pelo que diferentes chaves podem conter um número diferente de colunas em qualquer família de colunas, à semelhança do *BigTable*[12] da *Google* (Pokluda & Sun, 2013). "Família de colunas" refere-se à forma como as colunas são armazenadas no disco. Todas as famílias de colunas coexistem no mesmo ficheiro, ou conjunto de ficheiros relacionados (Eini, 2010).

De forma a garantir a persistência dos dados, mesmo em caso de falha, o *Cassandra* disponibiliza vários métodos para replicação de informação como *Rack Unaware*, o *Rack Aware* e o *Datacenter Aware*. O utilizador configura o fator de replicação por cada instância, sendo que o número do fator equivale ao número de vezes que os dados serão replicados no sistema. A tolerância a falhas é garantida através do método *Accrual Failure Detector*. Este consiste em atribuir a cada nó do sistema um valor numérico, um indicador de nível de falha. Os *metadata* de cada nó são armazenados em memória no *Zookeeper*, uma outra tecnologia do ecossistema de que o *Cassandra* precisa para o seu bom funcionamento. Quando ocorre uma falha num nó o espaço gerido pelo nó em falha é delegado no nó mais próximo até que ele seja recuperado (Pokluda & Sun, 2013).

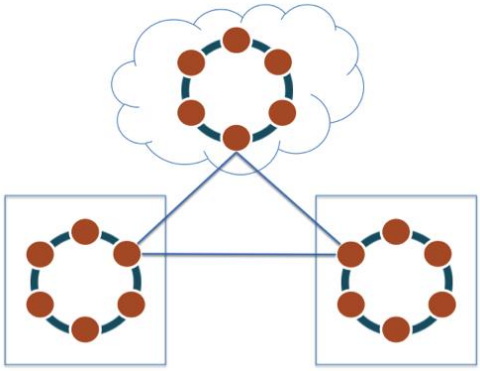


Figura 7 - Representação da arquitetura geral do *Apache Cassandra* (Schumacher, 2015)

A arquitetura geral do *Cassandra* é tipicamente representada nos sistemas como um conjunto de anéis interligados entre si, como pode ser visto na Figura 7, chamada também de

[12] *BigTable* – Base de dados criada pela *Google*, usada pelo *Google File System* para gerir armazenamento na ordem dos petabytes

"arquitetura em anel" pela *DataStax* (Schumacher, 2015). A arquitetura em anel não tem nenhum nó mestre e é constituída por um conjunto de nós que dialogam através de um protocolo escalável, o *gossip*. Todos os nós são idênticos em responsabilidade e funcionalidade. Esta arquitetura garante a capacidade e a resposta a milhares de utilizadores por segundo e a grandes quantidades de dados. A capacidade do sistema de responder a mais utilizadores simultâneos e a mais dados depende apenas da configuração do mesmo, ou seja, do número de nós disponível.

A informação escrita para o *Cassandra* é primeiro armazenada num *commit log* e depois escrita para uma estrutura em memória, a *memtable*. Quando a quantidade de dados na *memtable* passa determinado limite, configurado pelo utilizador, a informação dessa tabela é transferida para um espaço em disco, a *SSTable*. Desta forma, podem surgir várias *SSTables* referentes à mesma tabela lógica (Schumacher, 2015). O *Cassandra* otimiza este processo através de um procedimento periódico que compacta as diferentes *SSTables* numa só. Esta estratégia, ilustrada na Figura 8, reduz a quantidade de escritas para o disco e garante tanto a durabilidade dos dados como a performance do sistema (Pokluda & Sun, 2013).

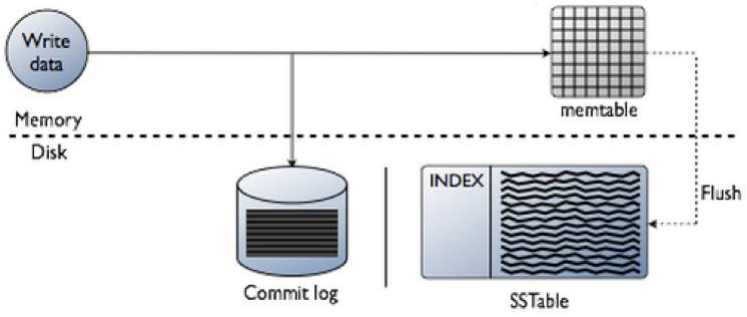


Figura 8 - Esquema da estratégia de escrita do *Apache Cassandra* (Schumacher, 2015)

Por sua vez, a leitura usa uma estrutura chamada *Bloom filter*. Esta estrutura ajuda a compreender se um *SSTable* tem a informação procurada com base em cálculos probabilísticos. Se na análise se entender provável que os dados procurados estejam numa tabela, o *Cassandra* procura na informação em memória os dados e só de seguida o armazenado em disco. Se não houver, passa para a *SSTable* seguinte. Desta forma, não requer obrigatoriamente que os dados em disco sejam lidos para identificar se estão naquela *SSTable*, conseguindo uma muito boa performance de leitura (Pokluda & Sun, 2013).

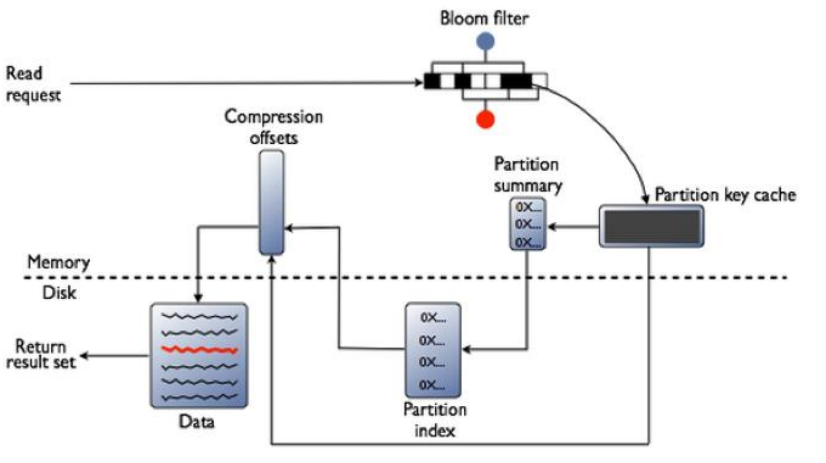


Figura 9 - Esquema da estratégia de leitura do *Apache Cassandra* (DataStax, 2016)

3.1.4.4 Apache HBase

O projeto *Apache HBase* é descrito pela ASF como uma base de dados orientada a colunas, distribuída, tolerante a falhas e altamente escalável. É organizada em tabelas, e cada tabela é armazenada como um mapa multidimensional, organizado em linhas e colunas. Cada linha contém uma chave de ordenação. As células apresentam a família de colunas, nome da coluna e versões, em que cada versão consiste num registo temporal do momento de escrita. É ainda possível associar mais informação, *metadata*, arbitrária a cada célula. Esta propriedade é utilizada para aplicações, por exemplo, de segurança, para registar acessos feitos individualmente a cada célula ou mesmo gerir o acesso de cada utilizador (George, 2010). Cada coluna pode conter também várias versões da mesma linha. As tabelas do *HBase* são organizadas pelo *HDFS*, de forma distribuída entre vários ficheiros e blocos. A replicação é, a este ponto, gerida pelo próprio *HDFS* (Vora, 2011).

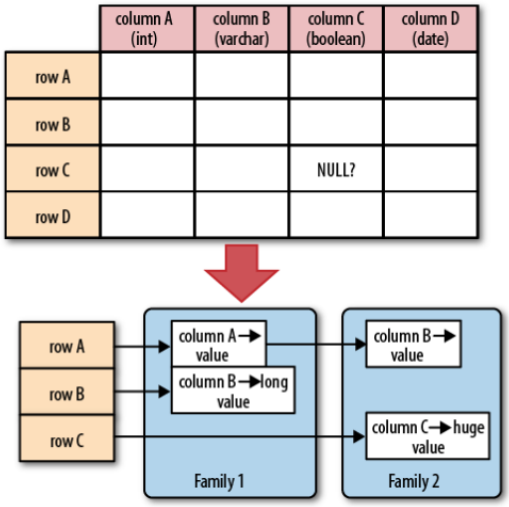


Figura 10 - Estrutura de linhas e colunas no *HBase* (George, 2010)

O *HBase* tem mais um elemento chamado *Region*, a região. As regiões são conjuntos contíguos de linhas, armazenadas juntas. Estas são separadas ou juntas dinamicamente de acordo com as

necessidades do sistema. Se uma região estiver demasiado grande, esta pode ser dividida em duas regiões distintas ou, por outro lado, se o sistema entender que deve juntar duas regiões então estas voltam a ser juntas. Esta característica é chamada de *autosharding*. Uma outra característica implementada pelo *HBase* é o TTL dos dados. Por exemplo, durante os procedimentos de compactação ou descompactação de regiões, podem ocorrer processos de limpeza em que dados que estejam fora do prazo definido no TTL são descartados (George, 2010).

Há três componentes de mais alto nível no *HBase*: a biblioteca, o servidor mestre e o(s) servidor(es) regional(ais). Estes servidores podem ser adicionados ou removidos durante o funcionamento do sistema, de forma transparente, e são responsáveis pelos pedidos de leitura e escrita das regiões que lhes estão atribuídas. É uma estratégia do *HBase* para gerir picos de uso e equilibrar a carga do sistema. O servidor mestre é responsável pela gestão dos restantes servidores regionais e usa o *Zookeeper* para coordenação e escalonamento das tarefas. A biblioteca é o meio pelo qual os utilizadores interagem com os servidores regionais para tratar operações de leitura e escrita de dados (George, 2010).

3.1.5 Processamento Distribuído de Dados: *Batch* e *Stream*

A exploração e extração de informação dos dados é o veículo para conseguir extrair valor da informação de um sistema. É aí que reside o verdadeiro valor acrescentado do Big Data, e é parte imprescindível deste tipo de sistemas distribuídos. Existem algumas tecnologias do ecossistema *Hadoop* capazes de fazer processamento distribuído como, por exemplo, o *MapReduce* ou as tecnologias *Apache Storm* e *Apache Spark*.

3.1.5.1 Apache Storm

O *Storm* é um sistema de computação distribuída em tempo real. A arquitetura, suportada tipicamente apenas em memória, faz com que seja extremamente eficaz no processamento de *streams* de dados, conseguindo processar milhões de eventos por segundo por nó (Jones, 2012).

O *Storm* é descrito pelos autores como simples, e é constituído por dois tipos de componentes principais: os *Spouts* e os *Bolts*. Os *Spouts* funcionam como recetores de dados e geram cadeias de *Bolts* que farão o processamento. Os *Bolts* podem ser encadeados em série ou executados em paralelo, podendo ser adequados a qualquer topologia necessária para processamento de dados. O *Storm* requer que o sistema onde está configurado também tenha o *Zookeeper*, uma vez que é essa tecnologia do ecossistema que garante a boa gestão dos seus processos (Ryza, Laserson, Owen, & Wills, 2015).

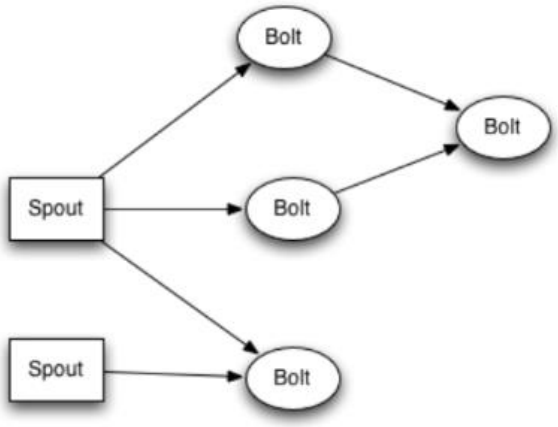


Figura 11 - Topologia do *Apache Storm* (Prakash, 2016)

Como sugere a Figura 11, a topologia deste sistema consiste numa rede de *Spouts* e *Bolts*. Cada nó desta rede faz algum processamento aos dados em trânsito. O *Spout* é tipicamente usado para se ligar a uma fonte de dados do sistema - como o *Kafka*, o *Flume* ou outro - para adquirir dados continuamente, transformando os dados num *stream* de tuplos que envia aos *Bolts*. Os *Bolts* ao conterem a lógica de processamento são usados para encaminhar os resultados de processamento para outros *Bolts*, ou então para exportar a informação ou enviá-la para armazenamento, entre outras ações (Ryza et al., 2015).

O *Storm* processa os eventos um de cada vez, ao contrário de outras tecnologias, que agrupam eventos antes do processamento. Cada evento é também gerido ao longo da cadeia de processamento individualmente (Leibiusky, Eisbruch, & Simonassi, 2012). Uma das vantagens do *Storm* é a capacidade de paralelizar este processamento e a facilidade com que se obtém essa paralelização pois basta, na definição da topologia, configurar o número de executores que se pretende para cada *Spout* e cada *Bolt*. Por outro lado, apesar do desempenho e da estabilidade de recursos que a paralelização permite atingir, existem também limitações associadas. Por exemplo, o *Storm* não possui uma estrutura de dados distribuída dentro da topologia, o que complica a aplicação de cálculos em janela, como cálculo de médias e semelhantes. Este facto obriga ao uso de estratégias, a desenhar caso a caso, para colmatar estas barreiras na fase de processamento.

3.1.5.2 Apache Spark

O *Apache Spark* é uma *framework* otimizada para computação distribuída nos *clusters*. Segundo a *Cloudera*, foi uma das tecnologias do ecossistema com maior adesão. A arquitetura do *Spark* permite carregar dados para memória e fazer pedidos sobre esses dados carregados entre 10 a 100 vezes mais rápido que o próprio *MapReduce*, e tem sido adotado para, por exemplo, aplicações de *Machine Learning* (Ryza et al., 2015).

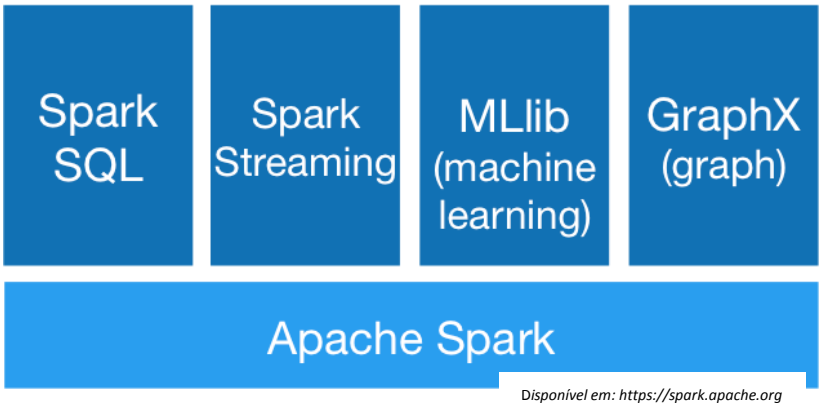


Figura 12 - Ecossistema *Spark* (Spark, 2017)

O *Spark* tem o seu próprio ecossistema, que pode também ser expandido adicionando diferentes módulos de processamento. Os módulos base são:

- *Spark Core* API (*Apache Spark*, na Figura 12): é a base do projeto. É um motor de processamento genérico, que possibilita as funcionalidades de computação. Apresenta também um modelo de execução que suporta várias linguagens de programação de forma a facilitar o desenvolvimento (DataStrax, 2015);
- *Spark SQL*: introduz as estruturas *DataFrames*, que pode representar tabelas das bases de dados relacionais tal como um *DataFrame* nas linguagens de programação *R* e *Python*;
- *Spark Streaming*: é o módulo responsável por executar as análises, possibilitando o uso da tecnologia em aplicações analíticas tanto através de *streaming* como de dados históricos. É facilmente integrável com fontes de dados como o *HDFS* ou outras como o *Kafka* e o *Flume* (Namiot, 2015);
- *Spark MLLib*: é o módulo de *machine learning* do *Spark*;
- *GraphX*: é o módulo que contém uma *framework* de construção/processamento distribuído de grafos.

O *Spark*, para funcionar corretamente, requer que no *cluster* também esteja presente o *Hadoop YARN* ou, alternativamente, o *Apache Mesos* e um sistema distribuído de armazenamento, como o *HDFS* ou o *Cassandra* (DataStrax, 2015).

O modelo de processamento do *Spark* baseia-se em processamento em *batch*. O processamento de eventos em tempo real do *Spark* é obtido desta forma, com pequenos *batchs (microbatching*[13]*)*. Para diminuir latências, os eventos são agregados durante curtas janelas de tempo, para depois se lançar esses mesmos eventos para processamento. Assim, com a gestão desta agregação, o *Spark* garante que os eventos são processados apenas uma vez, o que confere consistência ao sistema apesar do método introduzir latência (Ryza et al., 2015).

[13] *Microbatching* – Nome dado à metodologia usada pelo *Spark* para processamento em *streaming*.

3.2 Arquiteturas de Referência usadas em Big Data

De forma genérica, espera-se que uma arquitetura de referência revele que funcionalidades são necessárias, de forma mais ou menos específica, para resolver determinados problemas conhecidos. Deve referir como as funcionalidades são segmentadas, explicitando quais os componentes considerados e quais os fluxos de informação entre eles. Com a crescente utilização do conceito Big Data e as consequentes tentativas de implementação, tornou-se necessário definir componentes padrão e modelos operacionais para formar o ecossistema de Big Data, tal como por exemplo sucedeu com o ecossistema *Hadoop*, que é composto por um conjunto de tecnologias (componentes) com diferentes características para cada um dos blocos funcionais previstos na chamada *Stack* (Roman, 2015). Entenda-se *Stack* pelo conjunto de tecnologias integradas e disponibilizadas enquanto produto chave-na-mão numa distribuição.

Podem ser encontradas algumas descrições e notas sobre arquiteturas usadas para resolver problemas específicos em algumas das maiores empresas que usam Big Data como *Facebook* ou *LinkedIn*. A maioria das descrições das arquiteturas encontradas são orientadas às tecnologias usadas, o que geralmente omite em parte a visão das funcionalidades e distribuição destas pelos componentes (Maier, 2013).

3.2.1 Hadoop Stack

O *Hadoop* apresenta uma das maiores bibliotecas de tecnologias livres para Big Data e permite aos utilizadores tirar partido dessas tecnologias. A biblioteca disponibilizada é composta por tecnologias que estão, na sua maioria, categorizadas e podem ser usadas para complementar o núcleo da arquitetura *Hadoop*, o *Hadoop Framework,* com funcionalidades adicionais. Estas tecnologias, mantidas pela ASF, na sua maioria, surgiram quer como consequência da fama que o *Hadoop Framework* ganhou no meio, quer das necessidades que as empresas foram apresentando nos últimos anos. Juntamente com esta capacidade de complementar a *framework* com novas tecnologias, surgiu na documentação da *O'Reil*ly (White, 2009), a *technology stack*, ou pilha de tecnologias em português.

A *Stack* consiste numa forma de representar os grandes componentes funcionais das arquiteturas padrão e em distinguir as diferentes funcionalidades genéricas que são implementadas por diferentes tecnologias da biblioteca, de forma que possa ser feito um mapeamento das mesmas e se tenha uma visão mais simples e integrada dos componentes (Apache Software Foundation, 2014). Diferentes entidades que sigam este modelo de arquitetura farão um mapeamento de tecnologias total ou parcial de acordo com os requisitos das suas atividades e o seu contexto. Por exemplo, a *Cloudera* faz este mapeamento de acordo com a Figura 13. Esta representação é um suporte para planear e avaliar quais os componentes necessários para um sistema, como estes poderão ser utilizados e para que finalidade/funcionalidade.

3.2.2 Distribuições

As maiores entidades envolvidas no suporte empresarial em soluções de Big Data são a *Cloudera*, a *Hortonworks* e a *MapR*. Todas apresentam distribuições com a sua própria *Stack*, orientadas ao negócio como soluções chave-na-mão e apresentando serviços complementares, sempre baseada nas tecnologias do ecossistema *Hadoop*.

3.2.2.1 *Cloudera Distribution for Hadoop* (CDH)

A *Cloudera*, fundada em 2008, foca-se no desenvolvimento de tecnologias para o ecossistema *Hadoop*, além dos seus próprios serviços de suporte. Um dos principais produtos da *Cloudera* é a *Cloudera Distribution for Hadoop* (CDH), que apresenta uma *Stack* baseada nas tecnologias do ecossistema *Hadoop*. Estes componentes são os seguintes (Bhandarkar, 2012):

- *Hadoop Framework*. Este corresponde ao núcleo das tecnologias *Hadoop*, consistindo em quatro tecnologias fundamentais: *Hadoop Common*, que é o conjunto de bibliotecas e funcionalidades básicas; *HDFS*, que é o sistema de ficheiros distribuído da *Hadoop*; *Hadoop YARN*, que contém as funcionalidades de gestão de recursos; e o *Hadoop MapReduce*, um modelo de programação para processamento de dados em grande escala (Bakshi, 2012);
- *Coordenação*. Diz respeito aos componentes adicionais dedicados à coordenação e manutenção dos processos do sistema;
- *Integração de Dados*. A integração de dados refere-se às funcionalidades de acesso e disponibilização de dados, que permitem integrar um sistema com outras fontes externas e heterogéneas de dados;
- *Acesso rápido de leitura e escrita*. Há componentes na biblioteca específicos para otimizar a performance de acesso aos dados. Esses componentes são mapeados para este conceito;
- *Linguagens e Compiladores*. O *Hadoop* e as suas tecnologias podem usar diferentes linguagens e compiladores, podendo cada uma ter as suas especificidades, pelo que são também consideradas para a *Stack*;
- *Fluxo e Escalonamento*: O sistema *Hadoop* considera também o fluxo de informação e escalonamento de tarefas que pode ser assegurado por componentes do ecossistema;
- *SDK* e *UI*. Por fim, na arquitetura é considera a interoperabilidade com o próprio sistema através dos componentes de *Software Development Kit* (SDK) e *User Interface* (UI).

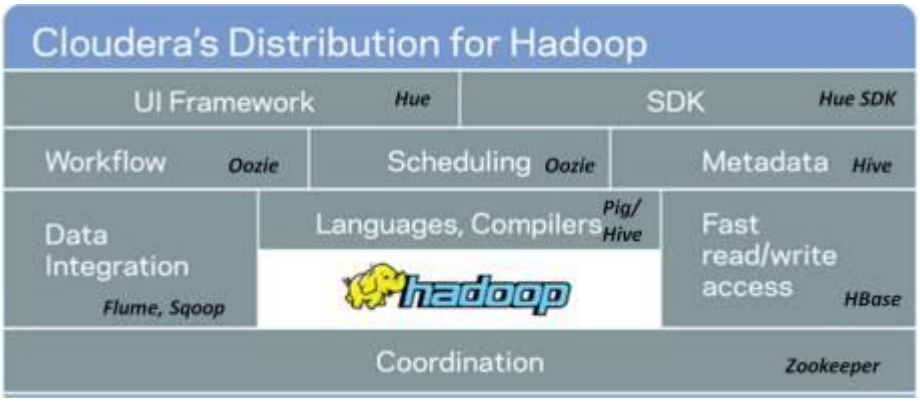


Figura 13 - CDH - *Cloudera Distribtion for Hadoop* (Bhandarkar, 2012)

O CDH é, no entanto, apenas um dos produtos da *Cloudera*. A solução de facto empresarial promovida pela *Cloudera* é a *Cloudera Enterprise* (Figura 14) que consiste numa *Stack* mais completa e flexível em termos de aplicabilidade, ao acomodar mais tecnologias opcionais e com mais atenção dada a funcionalidades valorizadas pelas empresas, como por exemplo a segurança. Contudo apenas parte se encontra disponível como *Open Source*.

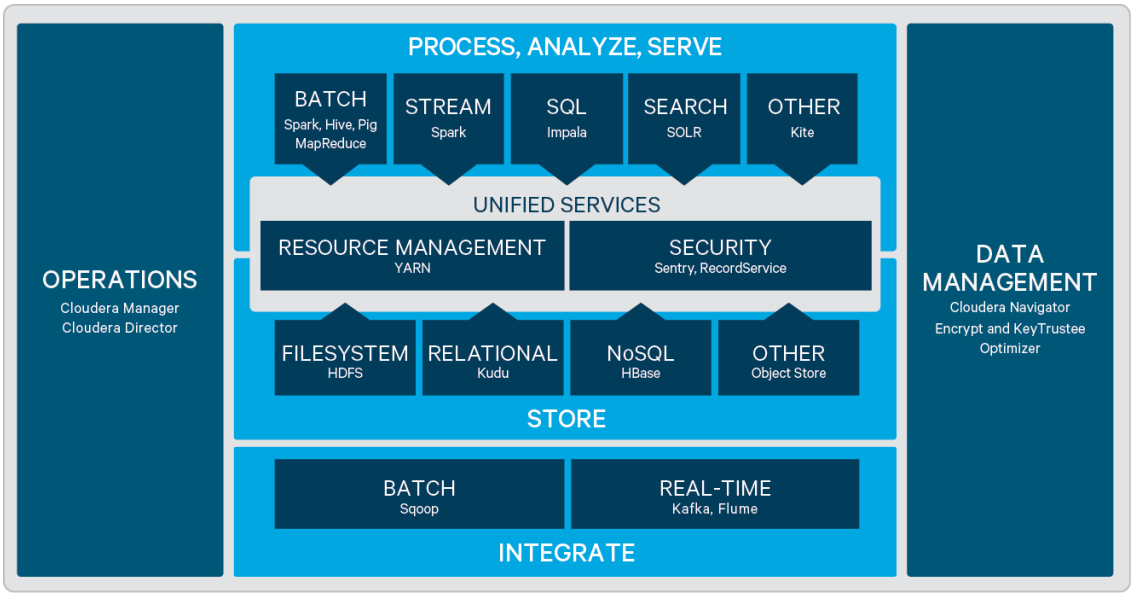


Figura 14 – *Cloudera Enterprise Stack.* (Cloudera, 2017)

3.2.2.2 *Hortonworks Data Platform* (HDP)

A *Hortonworks*, empresa fundada em 2011, dedica-se ao desenvolvimento e suporte do *Apache Hadoop*. O seu produto, o *Hortonworks Data Platform* (HDP), consiste num conjunto de formas de distribuir um sistema que resulta da seleção de várias tecnologias da *Hadoop Stack*. A distribuição é feita através de, por exemplo, procedimentos complexos desde instalação manual até máquinas virtuais (as chamadas *Sandboxes*) com sistemas prontos a serem usados.

A *Hortonworks* promove o *Hadoop* como algo vital para centros de dados (Hortonworks, 2014). A *Stack* apresentada pela *Hortonworks* difere da *Stack* apresentada pela *Cloudera*, e define os componentes entre:

- *Gestão e Integração*. Este bloco inclui funcionalidades de inclusão e acesso a dados externos, assim como a gestão desse acesso orientado a políticas previamente definidas;
- *Acesso a Dados*. Refere-se às funcionalidades de acesso aos dados presentes no sistema com os mais variados fins como processamento em *batch* ou *stream*;
- *Gestão de Dados*. Corresponde por natureza ao sistema de armazenamento e gestão do mesmo;
- *Segurança*. A segurança é um conceito não contemplado como um bloco funcional na definição original do *Hadoop*. Este componente deve garantir o cumprimento de requisitos de segurança como autenticação, autorização, contabilização (uso de recursos por utilizador) e proteção de dados;
- *Operações*. Monitorização, manutenção e escalonamento das tarefas e aplicações do sistema;

- *Apresentação e aplicações, Gestão e Segurança Corporativas*. Estes dois conceitos são vistos como blocos adicionais a esta arquitetura de Big Data. Estes são os componentes que incluem as interfaces quer de utilizador quer APIs, e os módulos de segurança de acesso aos restantes componentes da arquitetura.

O esquema disponibilizado pela *Hortonworks* num *white paper* sobre o conceito de "data lake[14]", apresenta uma *Stack* que inclui os componentes referidos acima e pode ser consultado na Figura 15.

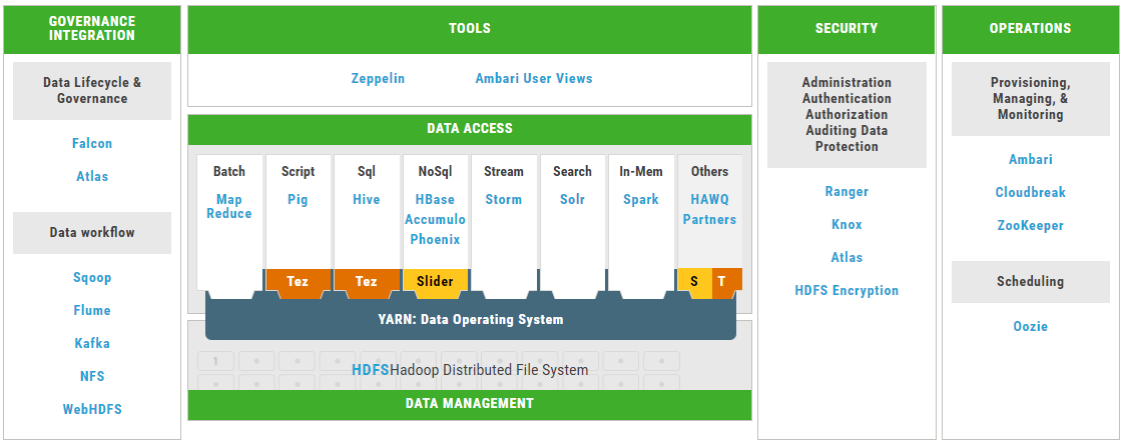


Figura 15 - *Hortonworks Data Platform Stack* (Hortonworks, 2014)

[14] Data Lake – Conceito que refere a um repositório de dados de grandes dimensões, destinado a alojar dados no seu formato nativo, tipicamente ficheiros não estruturados (Hortonworks, 2014).

3.2.2.3 *Converged Data Platform* (CDP)

O *Converged Data Platform* (CDP) é a distribuição "*community edition*" das distribuições da *MapR*, empresa que se dedica ao desenvolvimento dos próprios serviços e suporte associados às tecnologias de Big Data orientando- dessa forma, a sua estratégia comercial (MapR, 2016). A *MapR* concentra-se também nas ferramentas disponibilizadas no ecossistema *Hadoop* para constituir as suas distribuições, e investe no desenvolvimento de tecnologias privadas que acrescenta à solução com a intenção de otimizar a solução disponibilizada ao cliente. Um bom exemplo deste investimento é a tecnologia *MapRFS,* a qual usam como alternativa ao tradicional *HDFS*, presente e basilar em todas as outras distribuições. A sua solução não disponibiliza componentes de acesso a dados armazenados de alta disponibilidade, como o *NameNode-HA* ou *HFS-HA*, na sua versão de comunidade.

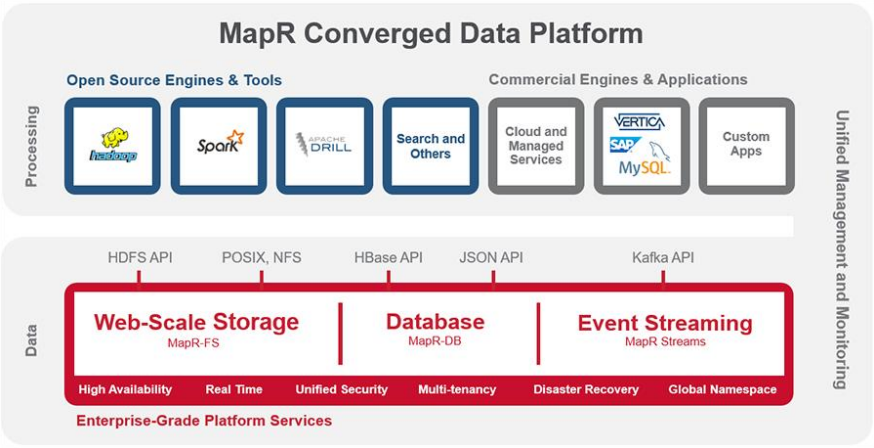


Figura 16 - *Converged Data Platform Stack* (MapR, 2016)

A *Stack* disponibilizada pela *MapR* (ver Figura 16) é apresentada de forma relativamente diferente à dos concorrentes *Cloudera* e *Hortonworks*, mas pode facilmente ser mapeada para o mesmo conjunto de componentes funcionais. Se se fizer o exercício de apresentar a *Stack* da mesma forma que as anteriores, tomando como referência de comparação a *Hortonworks*, nota-se que a *MapR* apresenta uma alternativa completa para componentes de integração de dados, em que apresenta o *MapR Streams* (compatível com a *Kafka* API) em detrimento do *Kafka* e do *Flume*. O sistema de ficheiros distribuído padrão (*HDFS*) também é substituído pelo *MapR*. Esta distribuição apresenta também os seus próprios componentes de segurança, e evidencia a potencialidade de integração com tecnologias terceiras em detrimento da apresentação de tecnologias para outros componentes, como gestão ou escalonamento de tarefas do sistema.

3.2.2.4 Simulação de Seleção entre Distribuições – Método AHP

Para entidades que estejam a iniciar-se na área do Big Data e a começar a sua base de conhecimento, uma decisão comum a tomar-se após uma análise preliminar do mercado é a

escolha de como abordar as tecnologias de Big Data para começar a usufruir das mesmas o mais rápido possível. Tipicamente, o interesse de empresas nas tecnologias está relacionado com o potencial de melhoria do negócio, pelo que se evita constituir um sistema de raiz pois exige um conhecimento muito especializado dos colaboradores para que o processo seja ágil. Na ausência deste conhecimento, a curva de aprendizagem e o esforço necessário podem ser fatores que as empresas não estejam dispostas a suportar. O uso de uma distribuição elimina uma grande parte do esforço inicial e torna-se lógico adotar uma distribuição entre as apresentadas anteriormente. Surge, portanto, mais uma questão: "Qual distribuição usar?". A análise deverá essencialmente passar pelo objetivo do uso deste tipo de tecnologias, quais os casos de uso que se pretendem implementar, quais as tecnologias que aparentam ser as mais adequadas para implementar os casos de uso, qual o esforço de gestão e manutenção a colocar posteriormente no sistema desenvolvido, entre outros. Essas questões devem ser contrapostas com as características das distribuições disponíveis para tomar uma decisão ponderada e que evite custos imprevistos.

Um método para fazer esta análise e ajudar na decisão sobre qual destas distribuições adotar, é o método *Analytic Hierarchy Process* (AHP). Este método compreende 7 fases que vão desde a definição do que se pretende até à decisão final (Saaty, 2008). Essas fases são junto com uma curta análise exemplo que se segue.

Entenda-se como objetivo deste exemplo, a seleção de uma distribuição de um ecossistema de Big Data, para implementação de um demonstrador. No que diz respeito a distribuições, como já foi referido, existem três grandes entidades que disponibilizam distribuições: a *Cloudera*, com o CDH;- a *Hortonworks* - com o HDP; e a *MapR* - com a CDP. Numa fase inicial, para a decisão de uma entre as distribuições possíveis, podem-se utilizar três critérios:

- *Open Source*. Numa fase inicial, supondo que o objetivo final será fazer um demonstrador, os custos deverão ser reduzidos ao mínimo pelo que este é um critério relevante;
- Suporte para Processamento em *Batch* e *Streaming*. Os casos de uso a implementar serão criados e ajustados durante o processo. A plataforma deverá suportar os dois tipos de processamento convenientemente para que o mau suporte de um deles não se torne uma barreira a um caso de uso a implementar;
- Facilidade de Gestão e Manutenção. Tendo em conta que cada distribuição é composta por várias tecnologias e que estas são todas integradas criando várias dependências entre elas, uma ferramenta de gestão e manutenção da plataforma é também essencial.

Uma vez identificados os objetivos, os critérios de seleção e as alternativas - que irão variar de caso para caso - pode concluir-se a fase 1 com uma árvore hierárquica, que consiste em expor de forma hierárquica as definições anteriores (Figura 17).

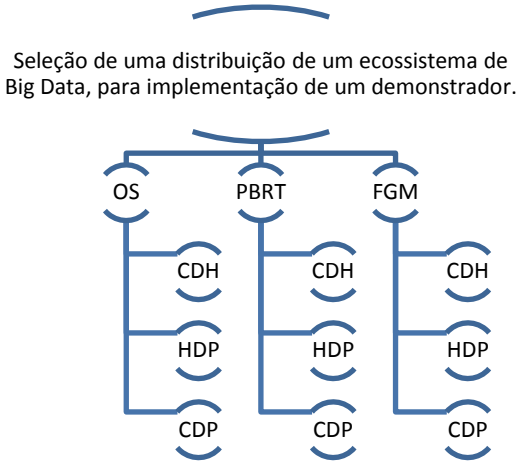


Figura 17 – Seleção entre Distribuições - Árvore Hierárquica

De seguida, é necessário evidenciar a importância relativa entre os critérios (fase 2), apresentados na Tabela 1.

Tabela 1 – Prioridades relativas dos critérios

	OS	PBRT	FGM
OS	1	1/3	1/2
PBRT	3	1	2
FGM	2	1/2	1

Depois de decidida a importância relativa dos critérios, calcula-se o vetor de prioridades relativas, normalizando a tabela de prioridades.

Tabela 2 – Cálculo das prioridades relativas

	OS	PBRT	FGM	Prio. Relativa
OS	1/6	2/11	1/7	0,1638
PBRT	3/6	6/11	4/7	0,539
FGM	2/6	3/11	2/7	0,2972

Desta forma identifica-se a vetor do peso dos critérios (fase 3):

$$\begin{bmatrix} 0{,}1638 \\ 0{,}539 \\ 0{,}2972 \end{bmatrix}$$

Posteriormente, procede-se à avaliação da consistência das prioridades relativas (fase 4), onde o objetivo é calcular a razão de consistência (RC). Para isso, o primeiro passo é calcular o Índice de Consistência (IC), para medir a consistência dos julgamentos. Para isso, considera-se:

$$A\,x = \lambda_{\max}\,x$$

onde,

A – Matriz de comparação (matriz das prioridades relativas dos critérios)

x – Vetor próprio

$\lambda_{\max}$ – valor próprio

$$\begin{bmatrix} 1 & 0{,}33 & 0{,}5 \\ 3 & 1 & 2 \\ 2 & 0{,}5 & 1 \end{bmatrix} \begin{bmatrix} 0{,}1638 \\ 0{,}539 \\ 0{,}2972 \end{bmatrix} = \lambda_{\max} \begin{bmatrix} 0{,}1638 \\ 0{,}539 \\ 0{,}2972 \end{bmatrix}$$

$$\begin{bmatrix} 0{,}4921 \\ 1{,}6248 \\ 0{,}8943 \end{bmatrix} = \lambda_{\max} \begin{bmatrix} 0{,}1638 \\ 0{,}539 \\ 0{,}2972 \end{bmatrix}$$

$$\lambda_{\max} = \frac{\left(\frac{0{,}4921}{0{,}1638} + \frac{1{,}6248}{0{,}539} + \frac{0{,}8943}{0{,}2972}\right)}{3} = 3{,}009$$

Uma vez obtido o valor próprio ($\lambda_{\max}$), o IC é obtido através de:

$$IC = \frac{(\lambda_{\max} - n)}{(n-1)}$$

Sendo que n = 3, $IC = 0{,}0046043395$ e dado que na escala AHP a ordem n de valor 3 corresponde a 0.58, pode-se calcular a RC:

$$RC = \frac{IC}{0.58} = 0{,}008$$

Atendendo a que uma RC menor do que 0,1 é considerada no método como um valor que identifica que há consistência nos valores das entradas, e o valor obtido 0,008 < 0,1, pode concluir-se que os valores das prioridades relativas são consistentes.

Uma vez validado este processo, passa-se à construção das matrizes de comparação paritária para cada um dos critérios para cada alternativa (fase 5), seguindo o mesmo raciocínio aplicado para calcular o vetor do peso dos critérios.

1. Critério - *Open Source*

Ponderação: Todas as distribuições apresentam tecnologias *Open Source*. No entanto, tanto a CDH como a CDP apresentam *software* proprietário na sua distribuição, ao contrário do HDP. O CDH utiliza ferramentas complementares essencialmente de gestão, ao passo que a CDP substitui mesmo a tecnologia de armazenamento padrão, HDFS, por uma tecnologia própria, *MapR-FS*.

Tabela 3 - Matriz de comparação e vetor de prioridades para critério: *Open Source*

	CDH	HDP	CDP	Vec Prioridade
CDH	1	1/6	1/3	$\begin{bmatrix} 0{,}093 \\ 0{,}685 \\ 0{,}221 \end{bmatrix}$
HDP	6	1	4	
CDP	2	1/4	1	

2. Critério - suporte para processamento em *Batch* e *Streaming*

Ponderação: As tecnologias mais maduras e suportadas pela comunidade para processamento dentro do ecossistema *Hadoop* são o *Spark* e o *Storm*. Quanto melhor for a integração das tecnologias mais suportadas pela comunidade na distribuição, melhor será a experiência e a facilidade em implementar novos casos de uso no sistema implementado. O HDP suporta tanto o *Spark* como o *Storm* nas distribuições mais recentes, ao passo que o CDH suporta perfeitamente o *Spark* mas não suporta o *Storm*. Alternativamente, o CDH, dá suporte a tecnologias como o *Impala* e o *ClouderaSearch* (ambas tecnologias desenvolvidas pela *Cloudera*) para procura e acesso rápido a dados. A *MapR*, por sua vez, suporta o *Spark* e posteriormente adiciona o *MapR-Streams* à sua distribuição, que é também uma ferramenta de processamento de eventos.

Tabela 4 - Matriz de comparação e vetor de prioridades para critério: suporte para processamento em *Batch* e *Streaming*

	CDH	HDP	CDP	Vec Prioridade
CDH	1	1/3	2	$\begin{bmatrix} 0{,}240 \\ 0{,}623 \\ 0{,}137 \end{bmatrix}$
HDP	3	1	4	
CDP	1/2	1/4	1	

3. Critério - Facilidade de Gestão e Manutenção

Ponderação: O suporte da distribuição a ferramentas integradoras de gestão e manutenção num *cluster* são de grande importância. Um sistema tão complexo como os das distribuições, com tantas tecnologias e com dependências entre elas, é muito trabalhoso de manter sem uma ferramenta adequada que abstraia o utilizador da complexidade do sistema. O HDP utiliza, por omissão, o *Ambari*, que é uma ferramenta de gestão que pertence ao ecossistema *Hadoop* (*Open Source*), que tem um suporte assíduo da comunidade e que suporta as ações de gestão, configuração e instalação das tecnologias da distribuição. Além desta ferramenta, o HDP permite integrar outras como o *Zeppelin,* que é uma interface *Web* facilitadora do acesso e análise de informação. O CHD por sua vez utiliza o *Hue* e as suas ferramentas de gestão privadas, como o *ClouderaManager,* para fazer a gestão da configuração e a integração das tecnologias da distribuição. O CDP apresenta também integração com o *Hue* e apresenta como gestor do ecossistema o *MapR Control System* (MCS), que fornece a interface de gestão e configuração da distribuição.

Tabela 5 - Matriz de comparação e vetor de prioridades para critério: Facilidade de Gestão e Manutenção

	CDH	HDP	CDP	Vec Prioridade
CDH	1	2	4	$\begin{bmatrix} 0{,}557 \\ 0{,}320 \\ 0{,}123 \end{bmatrix}$
HDP	1/2	1	3	
CDP	1/4	1/3	1	

Depois de obter todos os vetores de prioridade necessários, calcula-se a prioridade composta para as alternativas em causa (fase 6) pelo produto do peso dos critérios pela matriz formada pelos vetores de prioridade anteriores:

$$\begin{bmatrix} 0{,}16 \\ 0{,}54 \\ 0{,}30 \end{bmatrix} \begin{bmatrix} 0{,}093 & 0{,}240 & 0{,}557 \\ 0{,}685 & 0{,}623 & 0{,}320 \\ 0{,}221 & 0{,}137 & 0{,}123 \end{bmatrix} = \begin{bmatrix} 0{,}31 \\ 0{,}54 \\ 0{,}15 \end{bmatrix}$$

Encontra-se, assim, neste exemplo, as prioridades compostas para as três alternativas, que são de 31% para o CDH, 54% para o HDP e 15% para o CDP. Atendendo aos resultados obtidos, escolhe-se (fase 7) a distribuição que melhor satisfaz de forma geral os critérios definidos, neste caso, a distribuição da *Hortonworks*, HDP.

3.2.3 Arquiteturas Padrão

Com a mudança de paradigma que os sistemas de Big Data trouxeram, notou-se também uma evolução no conceito das arquiteturas padrão de Big Data. Estas arquiteturas apresentam soluções para os problemas mais comuns dos sistemas de Big Data, e servem normalmente como base de trabalho a adequar de acordo com os casos de uso a implementar. As arquiteturas mais comuns são apresentadas nos próximos subcapítulos.

3.2.3.1 Arquitetura "Tradicional" de Big Data

As tecnologias de Big Data começaram por ser uma forma das empresas superarem as dificuldades que tinham quanto ao armazenamento e ao processamento massivo de dados em variados formatos. É também na sua génese que se encontra a arquitetura de referência, que consistia em alguns passos: a aquisição/importação de dados (provenientes de sistemas já em produção) periodicamente; o processamento massivo de dados importados desta forma, recorrendo a tecnologias de acesso e ao processamento distribuído de dados, tipicamente recorrendo a tarefas *MapReduce*; a reinserção dos dados processados de novo no sistema original devidamente formatados para entrarem noutros processos do sistema, como por exemplo, em processos de ETL[15]. Note-se, a título de exemplo, na Figura 18, o processo "*Data Destillation*" de uma arquitetura "tradicional" proposta por Mike Barlow. O proposto consiste exatamente neste tipo de processo descrito acima: coletar informação de diversas fontes,

[15] *Extract, Transform and Load*

processar os grandes volumes de dados e formatá-los de novo para uma estrutura do sistema tradicional que eventualmente estará em produção, ou, citando: "*The data distillation phase includes extracting features for unstructured text, combining disparate data sources, filtering for populations of interest, selecting relevant features and outcomes for modeling, and exporting sets of distilled data to a local data mart.*", (Barlow, 2013).

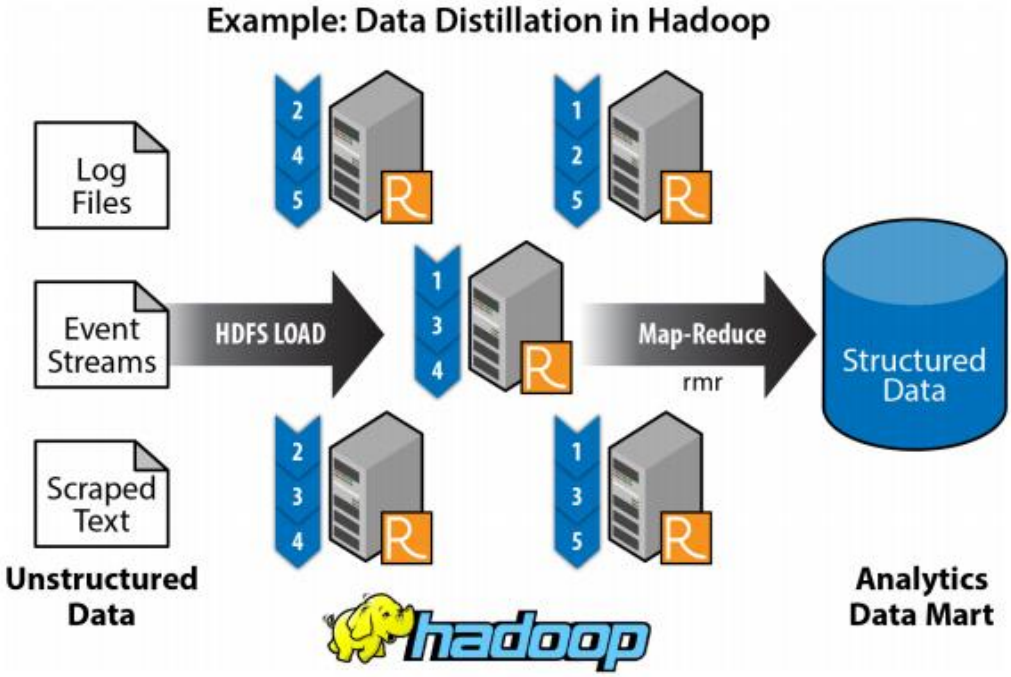


Figura 18 - Arquitetura "*Real-Time*" usada por Mike Barlow, publicada pela O'Reilly em 2013 (Barlow, 2013)

Nesta mesma publicação, o autor critica a performance do método afirmando que: "*Note also that at this phase, the limitations of Hadoop become apparent. Hadoop today is not particularly well-suited for real-time scoring, although it can be used for "near real-time" applications such as populating large tables or pre-computing scores."*, (Barlow, 2013). Atendendo às potencialidades do Big Data, foram surgindo novas tecnologias e arquiteturas capazes e otimizadas para dar resposta adequada aos mais variados casos de uso. Contudo, na definição do conceito de *Batch Layer* da arquitetura *Lambda* a filosofia continua semelhante à apresentada.

3.2.3.2 Arquitetura *Lambda*

A arquitetura *Lambda* é apresentada com um conceito diferente da arquitetura anterior, sendo representada de forma mais orientada aos objetivos da arquitetura. Esta consiste nos componentes essenciais para assegurar um sistema robusto e tolerante a falhas, quer de *hardware,* quer de origem humana, mantendo o desempenho de processamento do sistema com recurso a ambas as filosofias de processamento, "*Stream*" e "*Batch processing*" (Bertran, 2014). O *Batch processing* consiste no processamento em bloco de grandes quantidades de

dados. Neste contexto, é também a parte responsável pela gestão do armazenamento permanente da informação no sistema. Por sua vez, o *Stream processing* consiste no processamento assíncrono de cada evento recebido. O processamento é, geralmente, rápido, por forma a otimizar o tempo de resposta para capacitar o sistema de tempos de resposta para o utilizador próximos do tempo real (Namiot, 2015).

Para cumprir com os requisitos e com a filosofia de funcionamento desta arquitetura, ela é dividida em três componentes principais, denominados "camadas" (Nicolas, 2014):

- *Batch Layer*. Esta camada contém uma imagem do conjunto de informação a pesquisar (na Figura 19 identificado pelo ponto *2 – master dataset*). Este conjunto de dados é uma imagem incremental dos dados que entram no sistema. Aqui são também criadas as *Batch Views*, que consistem em respostas pré-calculadas dos pedidos que seriam feitos sobre os dados;
- *Speed Layer*. É a camada de processamento em tempo real da arquitetura, mantendo apenas o último registo em análise ou uma janela curta de dados. Esta cria *Real-Time Views* à medida que os dados chegam ao sistema;
- *Serving Layer*. Esta camada é responsável por combinar e agregar as vistas (*Views*) criadas por forma facilitar o acesso das camadas aplicacionais posteriores a esta arquitetura.

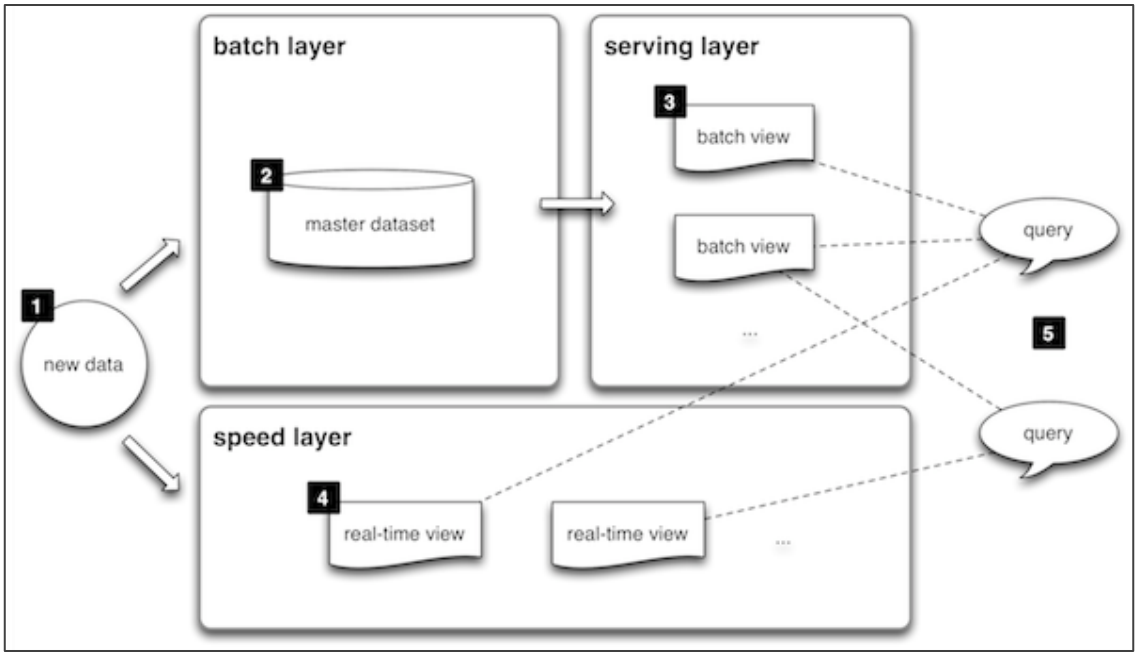


Figura 19 - Visão de alto nível da arquitetura *Lambda* (Hausenblas & Bijnens, 2015)

Esta arquitetura tem por objetivo ser um sistema híbrido, combinando o processamento lento e pesado com o processamento rápido e curto, de forma a procurar o melhor resultado de cada um, ou seja, resultados rápidos e profundos a partir de volumes de dados muito significativos (Hausenblas & Bijnens, 2015).

O fluxo padrão consiste em receber um conjunto de dados ao qual é aplicado o mínimo processamento possível até ser distribuído pelas duas camadas: *Batch* e *Speed*. Na camada *Batch,* os dados são inseridos no conjunto de dados e analisados periodicamente por tarefas do *Hadoop*. Os resultados desses processamentos são passados para armazenamentos posteriores (as *views*). Na camada *speed*, os dados são recebidos e processados sem passar por processos de armazenamento, ou seja, os dados só existem em memória. Após o processamento, estes são inseridos nas *views* de tempo real, usando o conceito de janela deslizante[16] (Prokopp, 2014). Conceptualmente, para evitar ausências instantâneas de informação, o tamanho da janela deslizante deverá permitir armazenar tanto tempo de informação, quanto tempo é espectável que a camada *Batch* demore a processar uma entrada.

Esta abordagem tem claramente vantagens. Permite, na camada *Batch*, processar grandes volumes de informação sem perder a responsividade do sistema, como alertas ou estatísticas, pois os valores instantâneos são disponibilizados pela camada *Speed*. Permite também uma gestão mais eficiente do armazenamento, tendo em conta que tipicamente os dados são disponibilizados com bases de dados que seguem o padrão *key-value pair*, estruturados especificamente para o acesso previsto das aplicações, e que estas estruturas de dados podem a qualquer momento ser limpas da camada *serving* com pouco impacto para o sistema. Por fim, em caso de falha ou erro, como trabalha diretamente com imagens dos conjuntos de dados armazenados originalmente, é simples descartar qualquer *view* e recriá-la, custando apenas o tempo de processamento. Pode aplicar o conceito de *Time-To-Live* aos dados em qualquer estado do processo (Grover et al., 2014).

Em última análise, a arquitetura *Lambda* pode ser mapeada na *Hadoop Stack*, no entanto é referida como um padrão que permite utilizar eficientemente as tecnologias do ecossistema *Hadoop*.

3.2.3.3 Arquitetura *Kappa*

A arquitetura *Kappa* surge como uma versão simplificada da arquitetura *Lambda*. Esta foca-se no processamento de dados em tempo real apenas, dispensando o *Batch Layer* (Fernandez et al., 2015). Em casos de uso em que o objetivo seja apenas processar eventos, como dados de sensores de *IoT*, *logs* ou mensagens de redes sociais, sem precisar de executar análises pesadas sobre grandes blocos de dados, a arquitetura *Kappa* apresenta-se como uma alternativa à *Lambda*. Assim, uma característica desta arquitetura é a relevância da informação mais recente, em detrimento do histórico dos dados. Em casos em que a análise requeira dados históricos, o sistema deverá ser complementado com um mecanismo que suporte a filosofia de consumo de eventos "*at least once"*, ou seja, uma tecnologia como o *Kafka,* que permita ler os eventos ordenados mais que uma vez. Alternativamente, a informação terá de estar armazenada em disco, o que atrasará o processamento dos eventos. A ordenação dos eventos entregues à *Speed Layer* é também relevante neste caso. O processamento, ao ser executado por evento,

[16] Janela deslizante – método que consiste em usar uma quantidade estática de elementos. Para cada entrada de um registo novo, um registo antigo é descartado.

requer, para manter a integridade dos resultados calculados, que os eventos cheguem ordeiramente, mesmo em caso de falha momentânea de algum componente. Se tal não acontecer, será necessário implementar lógica no sistema para prevenir a falha ou o envio desordenado de eventos. Isto acontece porque o tipo de análise mais coerente quando aplicada esta arquitetura é baseada em cálculos incrementais, onde o evento seguinte apenas acrescenta algo ao resultado atual.

Em última análise, a maior desvantagem desta arquitetura em relação à arquitetura *Lambda* talvez seja que a arquitetura *Lambda* garante a capacidade de reconstruir as *Real-Time Views,* se necessário, enquanto que a arquitetura *Kappa* está dependente da capacidade da sua fonte reter os dados e da implementação desta lógica em caso de falha.

Por outro lado, segundo os autores, um dos motivos que levaram ao desenvolvimento da arquitetura *Kappa* foi precisamente o reduzido esforço de implementação e manutenção que esta exige, quando comparada com a arquitetura *Lambda*. Uma vez que a arquitetura *Lambda* deve implementar a mesma lógica nas duas camadas, mas garantindo mais precisão no resultado na *Batch Layer*, esta acaba por ser muito trabalhosa para manter, pois todas as modificações de uma camada devem ser replicadas e mantidas em duplicado. Por isto, recomenda-se que a escolha entre estas duas arquiteturas seja feita com base no que se pretende processar. Se, em última análise, os algoritmos forem idênticos entre as camadas, então a recomendação será dispensar a *Batch Layer* e, se necessário, garantir que as *Real-Time Views* podem ser reconstruídas com um histórico aceitável, mesmo que a custo de reprocessamento. Por outro lado, se o esforço não for redundante e, de facto, o processamento da *Batch Layer* variar de acordo com o histórico disponível (por exemplo: algoritmos de *Machine Learning*) ao contrário da *Speed Layer*, então a recomendação é optar pela arquitetura *Lambda* (Forgeat, 2015).

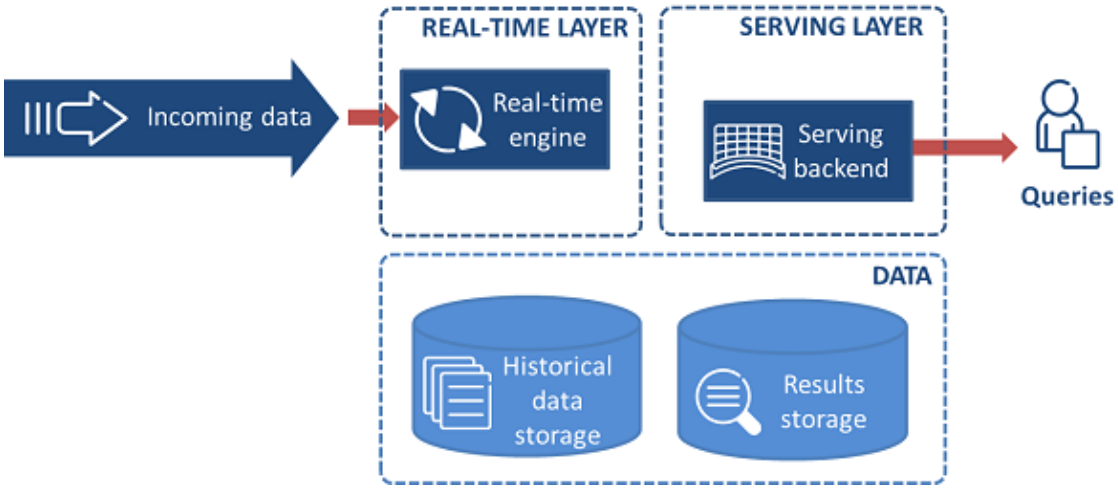


Figura 20 - Visão de alto nível da *Arquitetura Kappa* (Forgeat, 2015)

3.3 Casos de Uso de Sistemas de Big Data

Atualmente todas as grandes organizações, em especial as redes sociais, enfrentam o problema do Big Data, seja por os requisitos da sua área de negócio exigirem análise rápida de grandes volumes de informação, seja pela simples oportunidade de extrair conhecimento e valor dos dados já anteriormente armazenados para introduzir novos serviços ou produtos. Consequentemente, face à disponibilidade de tecnologias livres e à confiança que já ganharam no mercado, as grandes empresas tendem a utilizar o *Hadoop* e as tecnologias do seu ecossistema para abordar estes novos desafios. Em alguns casos, além de apenas utilizar as ferramentas do ecossistema chegam mesmo a integrar tecnologias desenvolvidas internamente no ecossistema, tal como acontece no caso do *Facebook* e do *LinkedIn*. As tecnologias são escolhidas tendo em conta as suas características e as necessidades das organizações.

3.3.1 Caso de Uso do *Facebook*

O *Facebook é*, atualmente, a rede social que mais informação produz por ano. Em 2014, registou a ingestão de quase 600 TB de dados diários, distribuídos entre cerca de 4,5 mil milhões de *likes* por dia, 350 milhões de fotos carregadas por dia, 4,75 mil milhões de *Shares* por dia, 10 mil milhões de mensagens de conversação enviadas por dia, entre outras funcionalidades da rede social que levam ao incremento de dados (Vagata & Wilfong, 2014).

Arquitetura Big Data do Facebook

Muitas das funcionalidades e aplicações do *Facebook*, desde simples relatórios ou operações de BI, a aplicações mais complexas de *machine learning*, requerem processamento de grandes quantidades de dados. Em 2011, por dia, o sistema do Facebook chegou a executar 10 mil tarefas associadas a estas funcionalidades, requeridas pelos utilizadores (Thusoo, Shao, & Anthony, 2010). A rápida taxa de crescimento dos dados levou a que se impusessem requisitos exigentes de escalabilidade na infraestrutura de processamento de dados. Idealmente, o sistema deveria ser escalável com o uso de *commodity hardware*, tendo em conta que é a única solução razoável considerando os custos inerentes.

O *Facebook* assume essencialmente dois tipos de fonte de dados: as bases de dados *MySQL*, que contêm toda a informação relativa ao *site* - por exemplo, contém informação sobre a publicidade como a categoria de um anúncio, o nome, informações de quem publica a informação, entre outros - e registos ou *logs* do sistema que contêm informações - como por exemplo, quando uma publicidade foi vista, quando foi clicada, entre outras ações (Menon, 2012).

De forma a abordar os problemas de armazenamento em que estava a incorrer, o *Facebook* adotou as soluções *Hadoop*. Os componentes de armazenamento e gestão de acesso à informação foram ocupados com a *Hadoop Framework* e com o *Hive*, uma tecnologia desenvolvida pelo próprio Facebook e mais tarde doada à ASF, que a incluiu no ecossistema

Hadoop. A estratégia de complementar a *framework* com o *Hive* consistiu em adicionar funcionalidades para acesso a dados com o SQL, metadados, partição, entre outros. O uso do *Hive* trouxe ganhos significativos na produtividade dos utilizadores que executavam as análises dos dados (Menon, 2012).

O *Scribe*, que ,tal como o *Hive*, foi desenvolvido pelo *Facebook* e doado à ASF, foi também adicionado ao sistema com o intuito de coletar e agregar de forma eficiente os *logs* de dezenas de servidores web distribuídos na infraestrutura (Thusoo, Shao, et al., 2010).

Na arquitetura do *Facebook*, o *Hive* publica tabelas de dados com uma cadência horária ou diária (Thusoo, Shao, et al., 2010). A interface do *Hive* para execução de pedidos é essencialmente feita através de uma ferramenta web, o *HiPal*, e da interface por linha de comandos, *Hive CLI* (Menon, 2012). O *HiPal* é uma ferramenta que permite gerar os pedidos de SQL para este sistema graficamente, o que é particularmente útil quando há utilizadores não familiarizados com SQL e especializados noutras áreas como marketing ou gestão. Além da criação dos pedidos, também permite acompanhar o estado do sistema com o progresso das tarefas em execução, inspeção de resultados de tarefas terminadas e interação com dados diretamente, podendo transferir ou carregar dados. Este tipo de funcionalidades é particularmente útil para fins de teste. Por exemplo, testar o sistema com pedidos mal construídos e analisar a resposta pode ajudar a descobrir vulnerabilidades ou erros (Thusoo, Sarma, et al., 2010).

A gestão e escalonamento das tarefas deste ecossistema são feitos pelo *Databee*. Este é uma *framework* desenvolvida em *Python* para especificar tarefas. É bastante flexível a nível de escalonamento e permite resolver dependências entre tarefas e agendar tarefas com base na disponibilidade de determinados conjuntos de dados. Por exemplo, pode ser estipulado que, quando determinado procedimento termina e cria um conjunto de dados como resultado, outro procedimento vai ser executado usando os dados resultantes do procedimento anterior. Isto permite criar sequências de execução, partindo uma tarefa complexa em várias fases de processamento, se necessário (White, 2009).

Mais recentemente, o *HBase* foi também aplicado no sistema do *Facebook*, onde foi integrado com o *Hive*. O *HBase* é uma base de dados que segue o padrão chave-valor (*key-value store*) e disponibiliza informação de forma semelhante ao conceito de *BigTable*, do *Google*. Este foi introduzido com o intuito de otimizar tempos de acesso à informação (Harter et al., 2014).

A arquitetura de Big Data do *Facebook* pode ser mapeada no *Hadoop Stack,* tal como pode ser visto no esquema Figura 21.

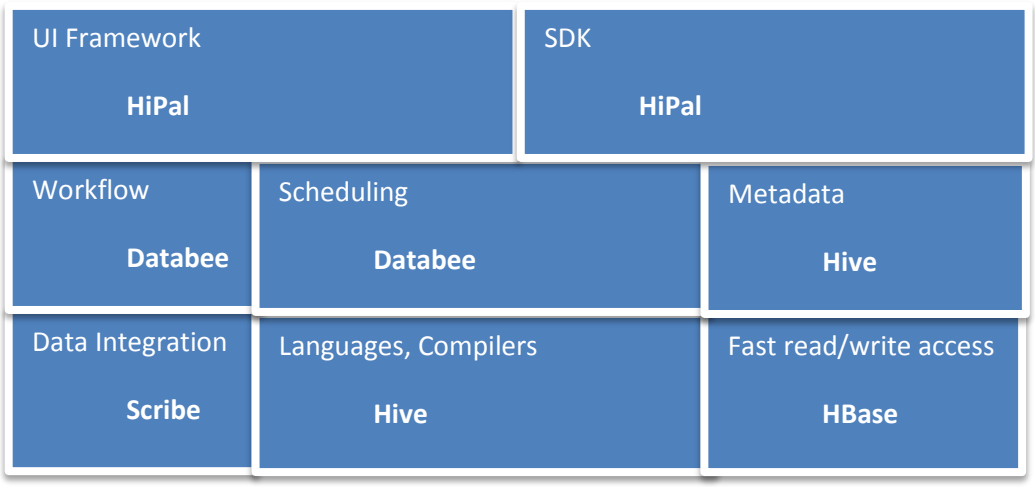


Figura 21 - Mapeamento de tecnologias Big Data usadas pelo *Facebook* no *Hadoop Stack*

O armazém de dados do *Facebook*, até 2012, estava armazenado em alguns *clusters* que corriam tecnologias *Hadoop*, que estão projetadas para grande escala de dados e performance. Com o crescimento acentuado do armazém de dados, este atingiu a escala do *petabyte* e foi necessário repensar a arquitetura utilizada até ao momento e otimizá-la. Nessa altura, o *Facebook* desenvolveu o *Presto*, em alternativa ao *Hive*, o qual é caracterizado por Martin Traverso, do *Facebook,* como um motor processamento de SQL otimizado para análises *Near Real-Time* (NRT), capaz de executar os pedidos de forma distribuída. Contrariamente à arquitetura *Hadoop* usada anteriormente, o *Presto* não usa *MapReduce*. Usa um sistema desenvolvido à medida das necessidades do *Facebook*, que suporta SQL nativamente e executa todo o processamento apenas com dados em memória para evitar operações de escrita e de leitura para o disco desnecessárias, otimizar o armazenamento e reduzir a latência com que o sistema consegue gerar respostas (Traverso, 2013).

Outra particularidade tida em conta no desenvolvimento do *Presto* foi desenvolvê-lo para ser facilmente extensível. Esta extensibilidade é necessária para se lidar com fontes de dados heterogéneas, pelo que o *Presto* foi desenhado com uma camada de abstração do armazenamento que permite o uso de pedidos de SQL sobre fontes de dados não relacionadas. Esta flexibilidade é implementada através de um componente chamado de *Connector*. O desenvolvimento dos *Connectors* é feito especificamente para cada nova fonte de dados e tem de implementar interfaces para acesso a metadados, obtenção da localização dos dados e o próprio acesso aos dados, seguindo para isso, um padrão de arquitetura baseada em mediadores, também chamados *wrappers* (Shi, 2002).

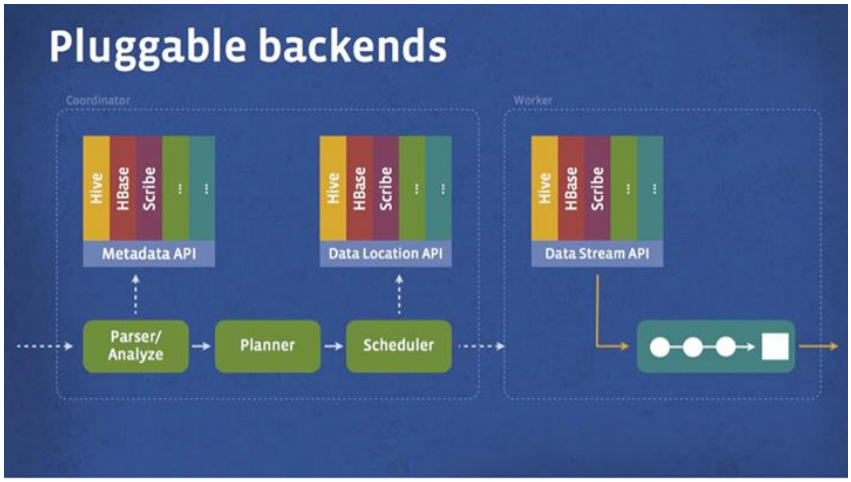


Figura 22 - Exemplo de plugins aplicáveis no "Conector de API" (Traverso, 2013)

3.3.2 Caso de Uso do Twitter

De 2012 para 2013, os cerca de 200 milhões de utilizadores ativos do *Twitter* passaram de 340 milhões de *tweets*[17] enviados por dia para 400 milhões, dos quais alguns disponibilizam[18] parte para fins de investigação através de interfaces públicas gratuitas (Tsukayama, 2013). Em 2014, apenas o texto escrito nos *tweets* durante uma semana representou aproximadamente 1TB de armazenamento (Penchalaiah et. al., 2014). Em 2015 geriam cerca de 5 mil milhões de sessões por dia (Hoff, 2013; Penchalaiah et al., 2014; Solovey, 2015).

Arquitetura Big Data do Twitter

No que concerne a receção de mensagens, os objetivos do *Twitter* foram orientados às plataformas móveis e, portanto, este está otimizado para economizar a energia dos dispositivos. Para isso, a estratégia nos dispositivos consiste em enviar eventos analíticos (não os *tweets*) em pacotes comprimidos. Estes dados são sempre comprimidos e temporariamente armazenados de acordo com as características do dispositivo, como o espaço disponível. O envio destes pacotes é feito através de eventos temporizados (*triggers*) que ocorrem periodicamente para tentar enviar esta informação armazenada e fazer também o controlo de falhas no envio, de forma a garantir que toda a informação chega ao servidor. Este protocolo de comunicação faz com que os dispositivos em conjunto acabem por enviar um número considerável de pacotes de mensagens a cada segundo, sendo que cada pacote pode conter mais do que um evento para ser processado (Mishne, Dalton, Li, Sharma, & Lin, 2012; Solovey, 2015).

Devido às características do sistema, o *Twitter* utiliza, assim como o *Facebook*, as tecnologias *Hadoop* como tecnologias base para a sua arquitetura. Define a sua arquitetura como uma

[17] *Tweet – Post* ou mensagem típica do *Twitter*, tipicamente bastante curtas, até 140 caracteres.
[18] A API está disponível através de *https://dev.twitter.com/rest/public/search.*

arquitetura *Lambda* e implementa além da *Hadoop Framework*, outras tecnologias do ecossistema *Hadoop*, como o *Kafka*, o *Storm* e o *Cassandra*, e soluções da *Amazon* para o armazenamento e para equilibrar a utilização do sistema entre as suas instâncias.

Para fazer a receção dos eventos, o *Twitter* utiliza o serviço *Endpoint*, que desempenha a única tarefa de empilhar os pacotes para a instância de *Kafka*. O envio para o *Endpoint* é mediado pelo serviço *Elastic Load Balancer* (ELB) da *Amazon*. Este serviço da *Amazon* garante maior tolerância a falhas e é usado para redirecionar automaticamente tráfego para diferentes instâncias, sendo que verifica a viabilidade da instância (integridade/erros e disponibilidade). As mensagens recebidas pelo *Kafka* são mantidas durante horas, funcionando como armazenamento temporário de rápido acesso (Solovey, 2015).

Os dados armazenados temporariamente pelo *Kafka* são transferidos para um servidor *Amazon S3* (*Simple Storage Service*) onde são armazenados permanentemente. O *Amazon S3* é um serviço de armazenamento online com interface através de serviços web (*REST*, *SOAP* e *BitTorrent*). A transferência é feita através do *Apache Storm* (Jones, 2012).

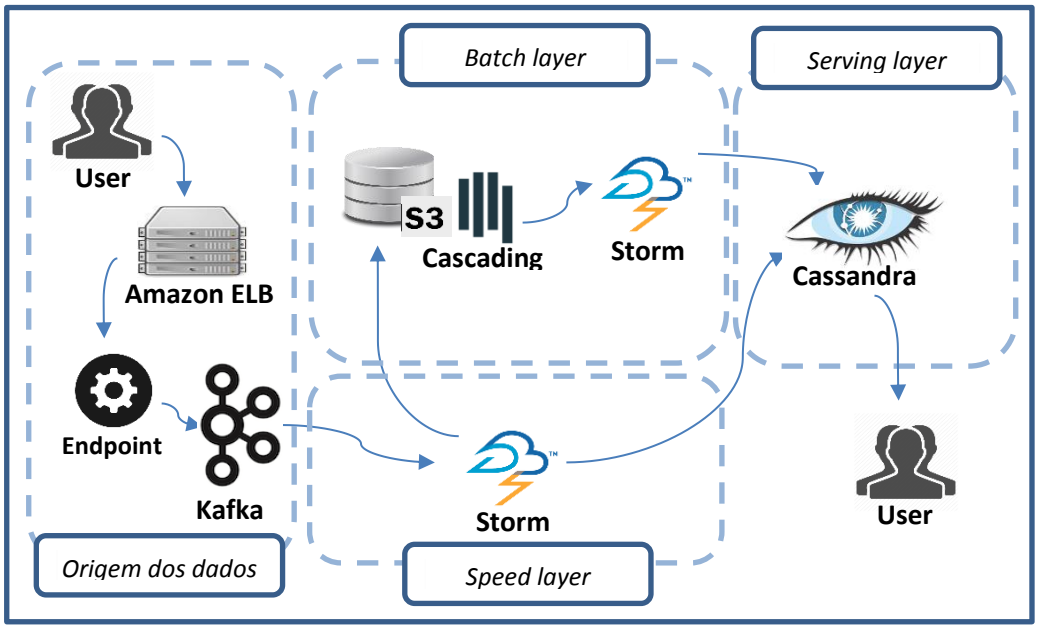


Figura 23 - Arquitetura de Big Data e fluxo de informação no *Twitter. Imagem adaptada de* (Solovey, 2015)

Na camada *Batch* da arquitetura *Lambda* do *Twitter*, é seguida uma cadeia de processamento para gerar as informações necessárias às aplicações que a empresa disponibiliza. Nesta camada são também realizadas experiências por parte dos investigadores da organização. O processo inicia-se com tarefas de *MapReduce* do *Hadoop*. Para facilitar a aplicação destas operações, o *Twitter* usa o *Cascading*, que cria uma camada de abstração sobre o *MapReduce* (Cascading, 2015). Essas operações, tipicamente com a duração de algumas horas, são executadas a partir

do *Amazon Elastic MapReduce*[19] (*Amazon* EMR) e, quando terminam, escrevem os resultados de novo para o *Amazon S3*. O *Storm* deteta o fim da execução destas tarefas e corre processos para transferir os novos resultados para o *Cassandra*, de forma a que os dados fiquem disponíveis para novos acessos (Farris, Guerra, & Sen, 2014).

Na camada *Speed*, o *Twitter* usa uma nova instância do *Storm* para aceder diretamente ao *Kafka*, de forma a processar a mesma informação e disponibilizá-la com o mínimo atraso possível. Os resultados das operações aqui executadas são quase os mesmos que no processamento em *Batch*, mas são utilizados algoritmos probabilísticos (como *Bloom Filters* ou *HyperLog*) para que o processamento seja mais rápido. No entanto, os resultados são menos precisos, logo, de menor qualidade. Estes resultados são também distribuídos para uma instância do *Cassandra* para que fiquem disponíveis às aplicações dos utilizadores (Solovey, 2015).

Na *Serving Layer*, a estratégia é disponibilizar sempre os dados de maior qualidade entre os disponíveis. Na prática, por exemplo, supondo que um utilizador executa uma ação que requer acesso tanto a dados recentes como a dados mais antigos ou históricos, a resposta processada vai usar os resultados provenientes da *Speed Layer* apenas na ausência dos mesmos resultados equivalentes provenientes da *Batch Layer* (Solovey, 2015).

Assim, podemos mapear a arquitetura do *Twitter* na *Hadoop Stack* da seguinte forma:

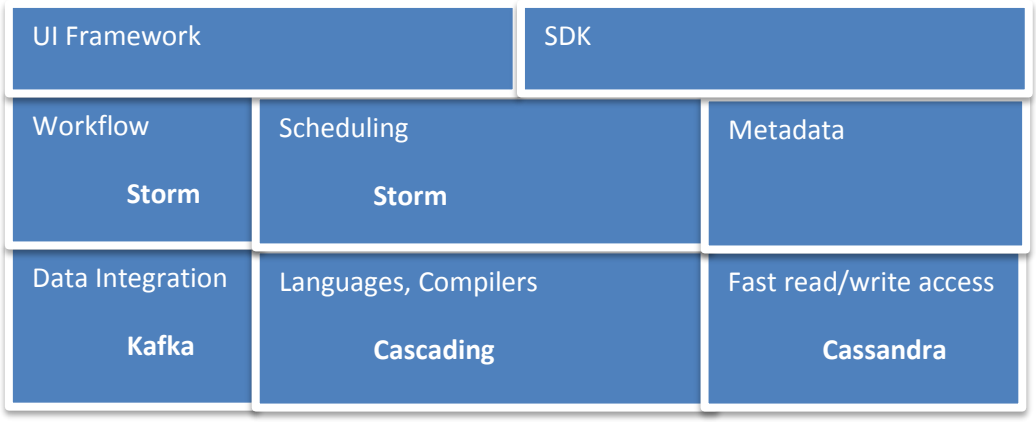


Figura 24 - Mapeamento de tecnologias Big Data usadas pelo *Twitter* na *Hadoop Stack*

3.3.3 Caso de Uso do LinkedIn

Já em 2010, o *LinkedIn* contava com cerca de 350 milhões de membros, 4,8 mil milhões de recomendações, 3,5 milhões de perfis empresariais ativos e 25 mil milhões de visualizações de páginas. Essa informação, além do uso regular da plataforma *online* é também usada pelos colaboradores, entre analistas, engenheiros, gestores e investigadores (*data science*) com o intuito de analisar padrões e compreender as interações e relações presentes na rede social

[19] *Amazon Elastic MapReduce* – É um serviço da *Amazon* (AWS) para fazer processamento de dados e análises.

(Naga, Kuan, & Wu, 2014). Um exemplo deste tipo de atividade pode ser encontrado na utilização da ferramenta *Socilab* do *LinkedIn*. Nesta ferramenta *online*, qualquer utilizador pode visualizar a sua própria rede de contactos com os métodos de análise desenvolvidos pelo *LinkedIn*. Outras funcionalidades que correm algoritmos sobre quantidades consideráveis de dados são aplicações como a "Pessoas que talvez você conheça" que é usada para sugerir pessoas que possam ser conhecidas pelo utilizador com base nos seus contactos diretos e nos contactos destes (Sumbaly, Kreps, & Shah, 2013).

Estas aplicações são possíveis devido à arquitetura *Hadoop*. O *LinkedIn* usufrui do facto do *Hadoop* permitir escalabilidade horizontal, tolerância a falhas (e recuperação autónoma) e *multitenancy*. Entenda-se *multitenancy* como a capacidade de processar *petabytes* de dados distribuídos por várias (milhares) de servidores baseados em *commodity hardware* (Sumbaly et al., 2013). As tecnologias usadas pelo *LinkedIn*, além da *Hadoop Framework*, são o *Kafka*, o *Azkaban* e o *Hive*.

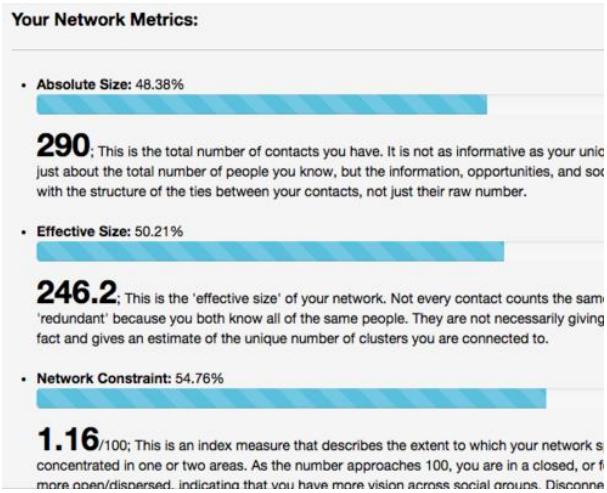


Figura 25 - Exemplo de resultado da ferramenta *Socilab* (*http://socilab.com*)

O *Kafka* permitiu ao *LinkedIn* ter acesso quase em tempo real a qualquer fonte de dados, melhorando a performance do sistema (Goodhope, Koshy, & Kreps, 2012). Esta característica permitiu assim melhorar a performance da monitorização feita ao portal e da capacidade de gerar alertas. Em 2015, o *Kafka* tratou cerca de 500 mil milhões de eventos por dia (Clemm, 2015).

A monitorização e gestão do fluxo e o escalonamento das tarefas no *LinkedIn* são responsabilidades do *Azkaban*. O *Azkaban* resolve também ordenação de execução de tarefas através das dependências das mesmas e disponibiliza uma interface web para facilitar o acompanhamento do estado das tarefas em execução (ASF, 2015).

Para otimizar o acesso aos dados do sistema, o *LinkedIn* implementa essencialmente duas metodologias diferentes. A primeira, e mais usada pelas aplicações, é o acesso rápido seguindo o padrão *key-value pair*. A tecnologia de armazenamento que implementa este padrão para o

acesso aos dados é o *Voldemort*, também criado pelo *LinkedIn*, que consiste num armazém de dados *NoSQL* distribuído, desenhado para permitir escalabilidade no armazenamento (Kreps, 2009). O *Hive* é utilizado para armazenar *metadata* e no fluxo para facilitar o acesso a dados através de pedidos de SQL.

O mapeamento das tecnologias *Hadoop* usadas pode ser visto na Figura 26.

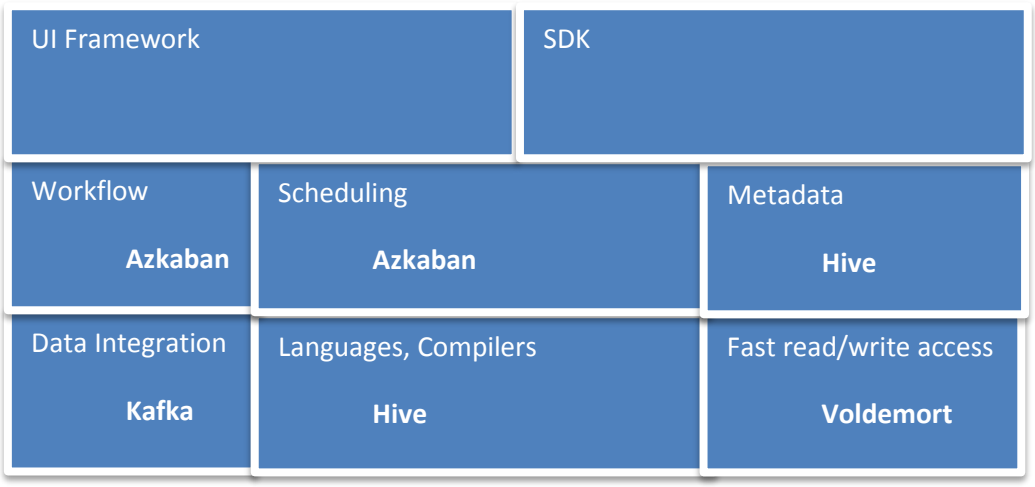


Figura 26 - Mapeamento de tecnologias Big Data usadas pelo *LinkedIn* na *Stack*

O *LinkedIn* disponibiliza mais alguma informação sobre a arquitetura do seu sistema e do uso que faz das tecnologias *Hadoop*, tal como pode ser visto na Figura 27.

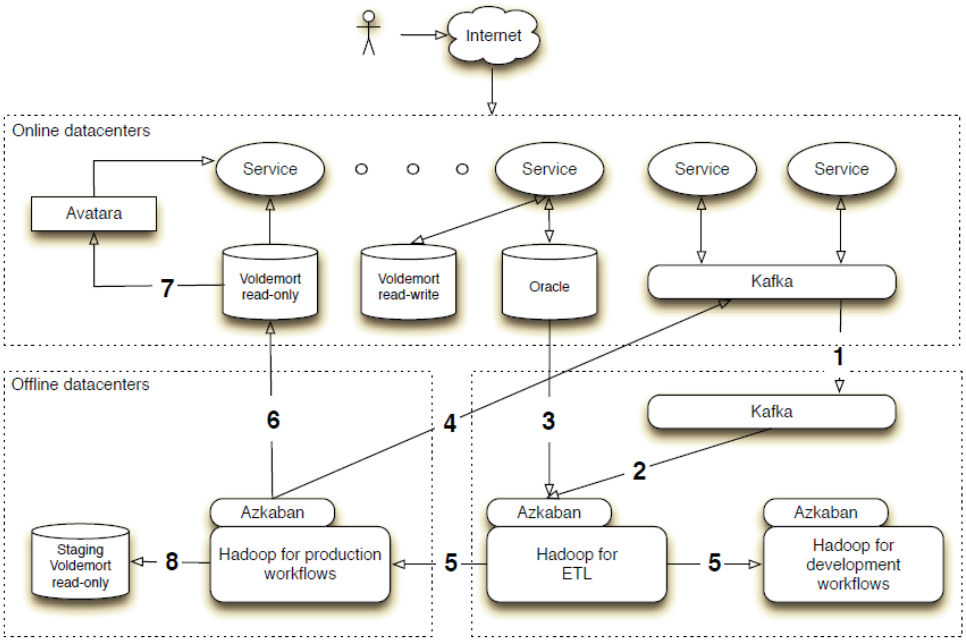


Figura 27 - Arquitetura e fluxo de dados no *LinkedIn* (Solovey, 2015)

O armazenamento geral do *LinkedIn* é feito para o *HDFS*. Os dados que entram para o *HDFS* são classificados em 2 tipos: dados de atividade e *snapshots* da base de dados. Os registos de alterações da base de dados são periodicamente compactados para gerar os *snapshots*. Os dados de atividades correspondem aos eventos chegados através dos pedidos aos serviços do *LinkedIn* (Solovey, 2015). Por exemplo, é gerado um evento quanto um utilizador vê outro perfil. Os eventos são agrupados por tópico e disponibilizados para o sistema através do *Kafka*. Uma vez que os dados estão disponíveis no sistema, eles são distribuídos entre duas instâncias distintas do *Hadoop* (Sumbaly et al., 2013). Uma das instâncias é destinada ao desenvolvimento e é usada pelos investigadores e analistas. A outra instância é a de produção, onde correm os fluxos de dados que vão gerar informação de valor para os utilizadores. Os resultados destes processos são novamente enviados ao *Kafka* e para uma instância *online* do *Voldemort* onde são disponibilizados aos serviços acessíveis aos utilizadores. Os mesmos resultados são armazenados numa instância *offline* do *Voldemort* com informação adicional da execução para facilitar a análise e perceção do que ocorreu durante a execução (Solovey, 2015).

4 Arquitetura e Seleção de Tecnologias de um Sistema de Big Data

O desenvolvimento de uma arquitetura para um sistema com recursos as tecnologias de Big Data depende, entre outros, da origem e cadência dos dados em questão, da disponibilidade de recursos e das análises que se pretendam fazer. Desta forma, e para efeitos de estudo, considere-se o exemplo de sistema de análise de dados sensoriais anteriormente referido como caso de uso para a seleção de tecnologias e arquitetura geral.

Neste capítulo é feito um exercício de avaliação das tecnologias identificadas e abordadas no capítulo anterior. Essa avaliação tem o intuito de gerar uma seleção de tecnologias, através de uma análise teórica das funcionalidades e características de cada tecnologia. Esta seleção identifica uma tecnologia para cada um dos objetivos: "integração de dados", "armazenamento em Big Data" e "processamento distribuído de dados".

4.1 Seleção de Tecnologias

A decisão sobre quais as tecnologias que devem compor um sistema de Big Data nunca é uma decisão fácil no contexto de uma organização. É necessário refletir sobre todos os aspetos como funcionalidades, quais as características dos dados, qual o fluxo de informação, que garantias deve o sistema dar, qual o modelo de negócio, entre outros. A oferta de tecnologias, nomeadamente de tecnologias livres da ASF, é vasta, sendo que todos os componentes funcionalmente concorrentes têm as suas características e serão mais ou menos adequados a determinada aplicação. Assim, é necessário uma análise comparativa das características de cada tecnologia para perceber como podem ou devem ser usadas em detrimento das similares.

A arquitetura a seguir é, por si só, também uma decisão importante para estes sistemas, visto que pode limitar as escolhas das tecnologias. Nathan Marz fez uma experiência onde resolveu o mesmo problema com uma arquitetura *Lambda* e com uma arquitetura do Big Data tradicional com armazenamento incremental. A experiência consistiu em calcular o número de

visitantes a um endereço web ao longo de um período de tempo. Era objetivo que os pedidos tivessem resposta em menos de 100 milissegundos. As duas soluções foram avaliadas em termos de precisão, latência e taxa de transferência (*throughput*). A arquitetura *Lambda* teve melhores resultados em todos os aspetos. Constatou-se que os procedimentos de atualização de versões do armazenamento era um processo muito custoso, que introduzia bastante latência, ao ponto de afetar os outros dois parâmetros medidos (Marz & Warren, 2015).

4.1.1 Tecnologias de Armazenamento

A escolha da tecnologia usada para armazenamento é, provavelmente, uma das mais importantes, tendo em conta que, geralmente, o esforço de migrar grandes quantidades de informação é maior do que simplesmente criar novas formas de acesso ou novos fluxos de dados.

Por exemplo, na maioria das aplicações em que o Big Data é utilizado, como aplicações móveis ou aplicações de *e-commerce*, o acesso à informação acaba por seguir a filosofia "*write once read many*" (WORM, ou "escrito uma vez, lido muitas"). O WORM define sistemas em que a informação uma vez escrita nunca mais é modificada, existindo leituras dos dados já escritos com muito mais frequência do que atualizações.

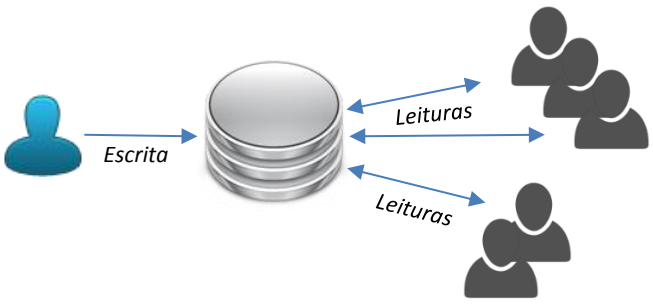


Figura 28 - Ilustração do conceito WORM

Esta filosofia levou, por exemplo, a que vários motores de acesso a dados e aplicações desenvolvessem estratégias para armazenar em *cache* [20] as informações lidas com mais frequência, para otimizar tempos de resposta. Será que a filosofia é adequada a aplicações do exemplo anterior? De acordo com uma análise de tecnologias feita pela *MarkedUp* (Aaron, 2013), a filosofia não é adequada quando o objetivo é produzir análises de dados. A empresa defende que é tendencialmente ao contrário, isto é, há poucas leituras para uma grande quantidade de escritas.

A tolerância a falhas é uma outra característica importante. De forma geral, os sistemas que tratam Big Data são distribuídos, portanto a facilidade de configuração, de gestão e a tolerância a falhas de cada tecnologia são fatores que tem de ser tidos em conta.

[20] Cache – no presente contexto, pode ser referida como uma memória auxiliar destinada a otimizar os tempos de acesso a informação pedida com mais frequência no sistema.

Um outro fator que pode ter influência é a capacidade de compactação (Marz & Warren, 2015). Esta deve ser tida em especial atenção se se planear que o armazenamento seja puramente incremental, ou seja, que todos os registos novos sejam incrementados e nunca resultem na atualização de dados já escritos. Os processos de compactação são tipicamente muito exigentes para o sistema, pois usam tanto tempo de processamento como operações sobre dados no disco. Estas duas operações a ocorrer num mesmo sistema podem fazer com que faltem recursos aos restantes processos. Por exemplo, o *HBase* (George, 2010) e o *Cassandra* (DataStax, 2016) requerem especial cuidado neste aspeto. A definição de períodos específicos de manutenção para estas operações é necessária para evitar problemas nos servidores, como a indisponibilidade de serviços ou mesmo corrupções por falta de recursos.

A "alta disponibilidade" é mais um dos potenciais requisitos de um sistema e, portanto, mais uma fonte de cuidados a ter na configuração e/ou escolha de tecnologias. Em caso de falha parcial da rede, o resto do sistema deverá tentar manter os níveis de disponibilidade sem comprometer a consistência dos dados. Considere-se consistência, a capacidade de um sistema de fornecer resultados coerentes com todas as operações anteriores. Num exemplo simplista, se um valor for incrementado duas vezes, é esperado que na próxima leitura o valor que inicialmente era 1 seja retornado como 3.

É importante ter em conta tanto as fragilidades das tecnologias como os requisitos do sistema. Assumindo que o sistema terá de ser capaz de disponibilizar respostas em NRT, de ser tolerante a falhas, de garantir a disponibilidade e de escalar consoante as necessidades do negócio, é possível identificar qual das duas tecnologias será mais aplicável, *HBase* ou o *Cassandra*. O *Hive*, apesar de ser um facilitador de acesso ao introduzir uma linguagem semelhante ao SQL, não aparenta ser o mais adequado para o sistema. O maior objetivo do *Hive* é introduzir uma sintaxe mais amiga do utilizador e melhorar a interoperabilidade com bases de dados relacionais. No entanto, no contexto esperado de aplicações sensoriais, potencialmente as fontes de dados são *streams* provenientes de sensores, ficheiros estruturados (XML/JSON) e/ou não estruturados.

No que concerne à tolerância a falhas, tanto o *HBase* como o *Cassandra* apresentam partição e replicação de informação, como já referido anteriormente. No entanto, o *HBase* suporta-se no funcionamento do *HDFS* e, portanto, possui o ponto único de falha, isto é, possui um nó principal do sistema responsável por gerir os restantes agentes delegados do sistema, neste caso, os servidores regionais. Se este nó falhar, a disponibilidade e o bom funcionamento de todo o sistema podem ser comprometidos. Por sua vez, o *Cassandra* tem um sistema de gestão completamente distribuído, não possuindo nenhum "nó mestre", o que representa uma clara vantagem a este nível.

Pela arquitetura do *Cassandra*, a sua configurabilidade será também mais simples que a do *HBase*. No entanto, esta simplicidade envolve um maior nível de abstração e, consequentemente, será mais complexo fazer a análise de erros do sistema. As arquiteturas de ambos não parecem influenciar a escalabilidade. No entanto, a arquitetura com os sistemas de anéis permite uma maior autonomia de cada nó do sistema, o que facilita a expansão do sistema.

O *HBase* não suporta mais de três famílias de colunas (HBase, 2016). Por sua vez o limite do *Cassandra* é de cerca de 2 mil milhões de famílias de colunas (Jain, 2013). Quanto à consistência, o *HBase* é feito de raiz com a consistência como objetivo, ao contrário do *Cassandra* em que a consistência pode ser ajustada através de parâmetros e suporta melhor a variedade e variabilidade dos dados.

Segue-se a Tabela 6, com um resumo de características relevantes das tecnologias de armazenamento referidas, algumas das quais discutidas acima.

Tabela 6 – Comparação de características entre *Hive*, *HBase* e *Cassandra*

Base de Dados / Característica	Hive	HBase	Cassandra
Modelo	Inspirado na base de dados relacional	Estrutura colunar	Estrutura colunar
Acesso	JDBC/ODBC Thrift	RESTful HTTP API Thrift	CQL Thrift
Suporta SQL	Sim	Com recurso ao *Apache Phoenix*	Com recurso a *driver* JDBC por terceiros[21]
Tolerância a Falhas (***Single Point of Failure***)	Sim Problema contornado por balanceadores de carga externos e redundância de serviço	Sim Contornado pela implementação do modo de HA (*High-Availability*)	Não
Alta Disponibilidade	Não	Sim	Sim
Usa cache ***(otimizar tempo de resposta)***	Sim	Sim	Sim
Compressão	Sim Codecs: gzip, lzo, bzip2, lz4, 4mc e snappy	Sim Codecs: lzo, lz4, gz e snappy	Sim Codecs: lz4 e snappy
Configurabilidade e Gestão	Moderado Integrado nativamente com ferramentas de gestão de *cluster*. Dependência de *HDFS*, *TEZ/MapReduce* e *YARN* carece de conhecimento do conjunto de tecnologias e da integração entre elas	Simples Integrado nativamente com ferramentas de gestão de *cluster*. Não apresenta dependências de configuração além do *Zookeeper*. Integração entre *RegionNodes* e *DataNodes* é transparente	Moderado Não está nativamente integrado com ferramentas de gestão de *cluster*. Não apresenta dependências das restantes tecnologias do sistema

[21] Driver JDBC para *Cassandra* disponibilizado, por exemplo, pela *DataStax* (datastax.com) ou pela *DBSchema* (dbschema.com)

Tendo em conta esta curta comparação de características das tecnologias abordadas, o *Apache Cassandra* aparenta ser a tecnologia mais adequada para uma solução baseada em dados sensoriais.

4.1.2 Integração de Dados

A análise da tecnologia para integração de dados consiste neste caso numa comparação entre as características do *Kafka* e do *Flume*. O *Kafka* e o *Flume* são as tecnologias mais comuns para este tipo de funcionalidade, no entanto foram desenvolvidas com propósitos diferentes. O principal objetivo e aplicabilidade do *Flume* é a ingestão de dados para o *Hadoop*. No entanto, a sua arquitetura permite que seja usado para transferir dados entre sistemas. Já o *Kafka* é desenhado para ser um sistema de mensagens. O seu foco é permitir aos *subscribers* ler as mensagens, sejam os *subscribers* o *Hadoop*, uma tecnologia de processamento como o *Storm*, um sistema externo ou diretamente um cliente que possua essa funcionalidade. É aqui que se encontra a maior diferença entre estas duas tecnologias. Apesar de poderem ser usadas ambas para receber dados do exterior e dar a sua entrada no sistema, o *Flume* está muito mais vinculado com o armazenamento, podendo interagir diretamente com ele, ao passo que o *Kafka* tipicamente apenas disponibiliza a entrada de dados no sistema. Esta característica do *Kafka* é visível no caso do uso do *Twitter*, no qual são as tarefas do *Storm* que capturam a informação do *Kafka* e a inserem na camada de armazenamento. Este facto faz com que, quando o objetivo é o armazenamento direto da informação, o *Flume* seja mais eficiente que o *Kafka*.

O *Kafka*, por ser um sistema de mensagens, traz outras vantagens. É mais versátil, pode funcionar tanto para a entrada de dados no sistema como para a saída, uma vez que os dados podem ser subscritos por entidades externas, ou mesmo servir diferentes serviços internos do sistema. Por isto, o *Kafka* é mais adequado quando o sistema requer alta disponibilidade, pois pode servir logo os componentes responsáveis pelo processamento de tempo real. Em alguma bibliografia, o *Kafka* é, inclusive, referido como um sistema de armazenamento temporário de dados (Casado & Younas, 2015), o que reflete a versatilidade deste.

Quanto ao processamento de dados no sistema, o *Flume* é capaz de executar pequenos algoritmos nos seus agentes, não tendo o *Kafka* este tipo de funcionalidade (Jiang, 2015).

Quanto à escalabilidade, o *Kafka* é teoricamente a ferramenta mais confiável. O número de *subscribers* da fila de mensagens do *Kafka* pode aumentar durante o funcionamento do sistema, sem que essa ação surta efeito na disponibilidade de serviço. Para facilitar a gestão nestes termos, o *Kafka* não se preocupa com conteúdos de mensagens, numeração por "tópico de mensagem" ou se determinado conjunto de dados já foi consumido. Entenda-se "tópico de mensagem" como, por exemplo, todas as mensagens tecnicamente do mesmo tipo, como os *tweets* no caso do *Twitter*. É responsabilidade dos consumidores de dados controlar que mensagens já foram lidas entre as disponíveis. Os dados armazenados temporariamente nas filas de mensagens do *Kafka* assumem um tempo de vida configurado, à semelhança do que

acontece nos sistemas de armazenamento. Já o *Flume* não escala de forma tão linear como o *Kafka*. Adicionar consumidores ao *Flume* pode implicar adaptar a topologia para que este envie os dados para um novo *sink*. Esta adaptação levará a breves momentos de indisponibilidade do sistema para que ocorra a reconfiguração (Jiang, 2015; Shreedharan, 2014).

Relativamente à capacidade de assegurar o bom funcionamento em situações de pico, o *Kafka* também trás uma melhor solução que o *Flume*. O *Flume* tem vantagens enquanto *interface* de entrada no sistema, ao permitir pequenas análises e ao tolerar que eventos fiquem "em espera". No entanto, esse mecanismo de processamento, junto com o mecanismo que mantém as mensagens em espera, pode fazer com que cadeias de agentes do *Flume* falhem, em particular porque os agentes usam um mecanismo de *push*, isto é, em que todas as mensagens recebidas são encaminhadas para o próximo passo da cadeia, obrigatoriamente. Por esta razão, tendem a ter uma capacidade menor de tratar picos de mensagens de entrada. O *Kafka*, por sua vez, não suporta nativamente o processamento nas mensagens de entrada, mas é capaz de absorver o impacto causado no sistema por uma taxa anormalmente alta de mensagens. As mensagens entram no sistema e ficam armazenadas à espera de serem lidas pelos componentes de processamento. Esta forma de funcionamento permite relaxar o resto do sistema, que não vai sentir tanto o pico e, portanto, vai manter a capacidade de resposta normal. Contudo, esta questão também implica um compromisso. O tempo de vida dos elementos do *Kafka* pode fazer, nesta fase, com que se percam mensagens, pelo que as situações de pico devem ser tidas em conta nas operações de configuração. Por seu lado, o *Flume* dá mais garantias de que não ocorrem perdas de dados se os agentes em si não falharem (Jiang, 2015).

Como se pode constatar, não há uma tecnologia obviamente melhor. É necessário considerar outros fatores como a natureza dos dados esperados. Estes na sua maioria devem ser provenientes de sensores ou de agregações de subsistemas de sensores, pelo que se prevê que sejam normalmente periódicos. No entanto, em sistemas de medição não periódicos que dependam de certos acontecimentos para iniciar e parar a aquisição, o mais provável é que ocorram *streams* de dados com duração potencialmente imprevisível, à taxa com que os sensores fizerem a aquisição. Não será estranho encontrar subsistemas para medição de vibrações com taxas de aquisição de dados até 10kHz, ou seja, 10000 valores por segundo. Geralmente, estes valores são agregados antes de serem enviados, ou são todos armazenados e processados localmente no subsistema. No entanto, é uma situação limite possível. Uma situação de pico com dados com a frequência do exemplo, que pode ocorrer num ou mais pontos do sistema, tem de ser considerada e, nesse sentido, o *Kafka* aparenta ser a tecnologia mais capaz de tratar uma situação destas (Kreps, 2014). O *Kafka* é também mais flexível no que toca à interligação entre sistemas ou ao armazenamento temporário entre estágios de processamento na camada de *stream*, pelo que é a tecnologia proposta para implementação no sistema resultante.

A Tabela 7 apresenta uma comparação sucinta entre algumas das mais importantes características a comparar entre o *Kafka* e o *Flume,* para suportar uma decisão entre os mesmos.

Tabela 7 – Comparação de características entre *Kafka* e *Flume*

Tec. Ingestão / Característica	Kafka	Flume
Atividade	Passivo Simplesmente um serviço de mensagens que é disponibilizado para *publishers* e *subscribers* interagirem	Ativo Dependendo do *Source* usado para o agente do *Flume*, ele pode também ser passivo, mas tem possibilidade de fazer *polling* da informação e de enviar (através do *Sink*) o evento para o próximo serviço
Segurança	Controlo por ACL (*Access Control List*); Autenticação; encriptação do canal de comunicação (TLS) e; *Kerberos*;	Encriptação do canal de comunicação (TLS); *Kerberos*;
Tolerância a falha	Sim Informação é escrita para armazenamento em disco com frequência e, em caso de falha, não se perde informação	Não Se os agentes falharem, os dados contidos em memória no momento serão perdidos
Replicação de dados	Sim Caso haja mais que 1 *broker* no *cluster*, pode ser feita a replicação de dados, que garante que o consumo continua, mesmo no caso de falha de 1 *broker*.	Não
Facilidade de Gestão	Simples É fácil adicionar tópicos e consumidores sem interrupção de serviço	Moderada Pode ser necessário parar uma sequência de agentes caso seja necessário fazer atualizações ou adicionar um novo tipo de informação
Facilidade de Uso	Simples, contudo é necessário desenvolver o próprio produtor e consumidor	Simples Existe bastante suporte da comunidade e agentes-tipo desenvolvidos que podem ser usados sem ser necessário implementar

4.1.3 Processamento Distribuído de Dados

As duas tecnologias de processamento mais utilizadas no mercado do Big Data são as exploradas no subcapítulo anterior, o *Apache Storm* e o *Apache Spark*. Apesar de serem soluções concorrentes para o processamento de tempo real, estas têm funcionamentos diferentes e foram desenhadas com potencialidades e objetivos diferentes. Uma das maiores vantagens do *Spark* sobre o *Storm* é o suporte que tem para outras funcionalidades ao integrar módulos, tais como o *MLib* e o *GraphX*. No que diz respeito à escalabilidade, tanto o *Spark* como o *Storm* foram desenhados para escalarem e tratarem grandes dimensões de dados facilmente, mantendo o desempenho e a facilidade de gestão do sistema (Oliver, 2014).

Enquanto o *Storm* é puro processamento em *stream* e *real-time*, o *Spark* funciona com um mecanismo de *microbatchs* pela agregação que faz dos eventos de entrada antes do processamento dos mesmos. O *Spark,* apesar de introduzir latência no sistema com esta metodologia faz, teoricamente, com que o processo seja mais eficiente. Processar mais dados em conjunto diminui naturalmente o *stress* dos componentes do sistema, ocorrendo menos ações de processamento e, a certo ponto, pode conferir melhor taxa de transferência do que o processamento em tempo real do *Storm*. Contudo, se o sistema for dedicado, com as entradas bem definidas, o *Storm* consegue tratar e disponibilizar resultados de processamento em tempos inferiores a um segundo, o que num sistema de Big Data, com os volumes a considerar, é do mais próximo de "tempo real" que se pode obter (Huynh, 2014).

Ambas as tecnologias, *Storm* e *Spark*, implementam mecanismos de tolerância a falhas. Ambas também dependem, para garantir que não se perde informação, de que a fonte de dados permita repetir os pedidos de informação já acedida anteriormente. Este facto reforça a seleção do *Kafka* enquanto tecnologia de integração de dados (Ballou, 2014).

Mais uma vez não há uma resposta ideal. Neste caso específico, foi necessário testar a implementação de um sistema com cada uma das duas tecnologias. A tabela de comparação entre o *Spark* e o *Storm* é acrescentada no final dos testes feitos durante, os próximos capítulos, com cada uma das tecnologias.

4.2 Arquitetura Geral e Tecnologias a usar para Processamento de Dados em Streaming

Assumindo o caso de uso de um sistema de análise de dados de sensores, e tendo em conta a análise dos três grandes componentes das *Stacks* para este tipo de aplicações – o sistema de ficheiros distribuído, a integração de dados e o processamento distribuído de dados – propõe-se uma arquitetura teoricamente adequada de forma genérica a este tipo de sistemas, sugerindo também as devidas tecnologias a usar na implementação da mesma.

Uma aplicação de dados sensoriais, se vista de forma genérica, pode receber os mais variados tipos de dados, desde dados estruturados a não estruturados. Uma característica que a arquitetura deve permitir é adaptar-se, pois os dados sensoriais, em particular com exemplos de aplicação como a *Internet of Things* (Namiot, 2015), estão em constante alteração. Assim, a tecnologia de integração de dados deverá permitir uma fácil adaptabilidade a diferentes fontes e garantir sempre a performance de entrada de dados no sistema. Quanto ao processamento de dados de sensores, é comum pretender-se informação atualizada em tempo real. Assuma-se, mais uma vez, o exemplo dos IoT, num caso de uso simples de domótica. Se queremos que o aquecimento se ative quando o GPS do telemóvel indicar que um individuo está a uma certa distância de casa, em direção a esta, é importante a resposta em tempo útil e, por isto, a melhor forma de o garantir é através do suporte, pela arquitetura, de processamento em tempo real. Por outro lado, a nível da informação extraível do histórico, por exemplo, para algoritmos de aprendizagem ou previsões, pode ser importante que a arquitetura suporte devidamente

também processamento bruto em *Batch* de dados. Atendendo às tecnologias e arquiteturas analisadas anteriormente, a arquitetura *Lambda* aparenta ser adequada a este tipo de sistemas e, portanto, é aqui proposta para o desenvolvimento e teste de protótipo em comparação com uma arquitetura *Kappa*, que é a alternativa teoricamente mais interessante.

No balanço das tecnologias para os componentes funcionais sugeridos para o sistema, concluiu-se que, para o acesso a dados de sistemas externos, o *Apache Kafka* é uma excelente opção pelas suas características enquanto sistema de mensagens, por conseguir garantir recuperabilidade em caso de falhas nas camadas de processamento, pela aplicabilidade enquanto tecnologia de entrada no sistema e pelo uso do padrão *publishe-subscribe* com capacidade de fazer *queueing*. Das tecnologias analisadas para armazenamento, o *Cassandra,* dada a sua grande capacidade de tolerância a falhas, escalabilidade e disponibilidade, aparenta ser a solução mais adequada. O *HDFS* é também um requisito do sistema, e pode ser usado para armazenamento na *batch layer*. O processamento tanto em *batch* como em *stream* é, teoricamente, mais adequado se for implementado em *Spark*. No, entanto, e apesar da aparente vantagem do *Spark*, em linha com o descrito no capítulo 4.1.3, nota-se necessária uma análise de performance entre o *Spark* e o *Storm*, que compara a velocidade de processamento e o impacto da tecnologia para o sistema, através de um caso de uso de *streaming*.

As tecnologias propostas acima são aplicáveis a um sistema que siga o padrão da arquitetura *Lambda*. A opção pela arquitetura *Lambda* é também corroborada pelos bons resultados analisados no caso de uso do *Twitter* (ver capítulo 3.3.2) e pela aparente simplicidade da arquitetura - quando comparada, por exemplo, com outras abordagens como a do *LinkedIn*.

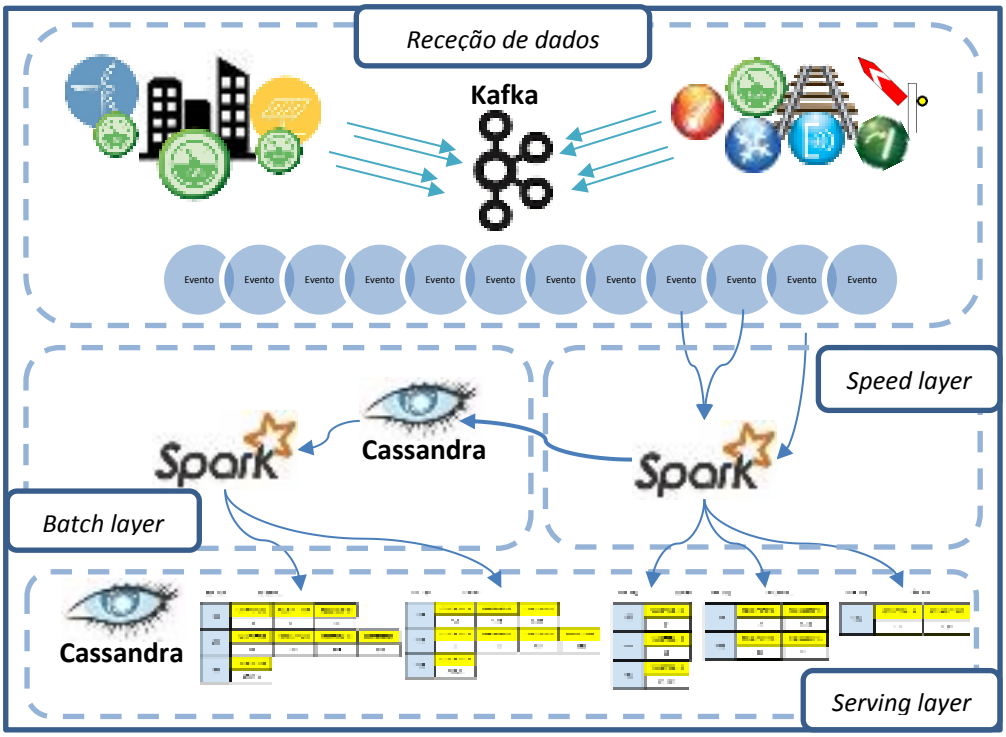


Figura 29 - Esquema orientado às tecnologias do fluxo de dados do sistema proposto

Desta forma, para a implementação de uma aplicação de *streaming* genérica para dados de sensores, o fluxo começaria na receção dos *streams* de dados (sistemas de aquisição). Esses dados são recebidos pelos *brokers* do *Kafka*. As tarefas do *Spark* devem consumir os dados recebidos pelo *Kafka*, encaminhá-los para o armazenamento baseado no *Cassandra* e iniciar as tarefas de processamento de dados em tempo real. O *Kafka* mantém os dados durante algum tempo após a sua leitura, isto é, o mesmo evento pode ser lido mais que uma vez, caso necessário. Este facto permite que as tarefas do *Spark* que tratam da leitura dos dados e do processamento em tempo real não tenham dependências entre si, pelo que o sistema poderá fazer os dois processos em simultâneo. Outras tarefas do *Spark* deverão alimentar-se dos dados armazenados no *Cassandra* para correr o processamento em *Batch*. Não consta nos objetivos da implementação, mas é nesta fase, no processamento em *Batch*, que seriam usados os módulos do *MLlib* e *GraphX* do *Spark* para se usar algoritmos de *machine learning* e gerar métricas de interesse e valor acrescentado para serem apresentadas em relatórios e *dashboard*. Desta forma, é possível determinar que o *Spark* possuiria três tipos distintos de tarefas a implementar no sistema. Os resultados de processamento, para efeitos de demonstração, poderão ser cálculos matemáticos simples e facilmente verificáveis, conhecendo os dados de origem. O *Cassandra* deverá por fim disponibilizar as tabelas nas suas estruturas orientadas a colunas.

O fluxo de dados descrito acima foi usado no desenvolvimento dos sistemas necessários à comparação tanto entre as arquiteturas aparentemente adequadas, *Lambda* e *Kappa*, assim como foi usado para comparar, em cada uma das arquiteturas, a tecnologia de processamento, Spark e Storm.

5 Implementação e Avaliação de Sistemas de Big Data

Neste capítulo é abordada a forma como foi instalado e configurado o sistema e quais as personalizações, passos e problemas surgidos durante todo o processo. Após os sistemas estarem devidamente instalados e funcionais, foram realizados os testes de acordo com os objetivos propostos. Finalmente são apresentados e discutidos os resultados, apresentadas as limitações e as dificuldades encontradas.

5.1 Desenvolvimento de Sistemas-Protótipo para Teste de Tecnologias

O procedimento de testes da arquitetura proposta teve duas fases distintas. Uma primeira, que consistiu no desenvolvimento de sistemas e em testes funcionais aos mesmos, em que se deve garantir que o sistema cumpre funcionalmente com o proposto. Posteriormente, cada sistema é comparado com outros sistemas com arquitetura e/ou tecnologias diferentes. Para suportar a escolha da tecnologia de *streaming* a usar, sendo as alternativas o *Apache Storm* e o *Apache Spark*, são comparadas entre si em ambientes o mais semelhantes possível. É também comparada uma arquitetura *Lambda* e uma arquitetura *Kappa*, implementadas com os mesmos componentes tecnológicos para testes comparativos a nível de desempenho, carga para o sistema e vantagens/desvantagens funcionais.

5.1.1 Recursos e Ambiente de Teste

Para os ambientes de teste, foram instaladas quatro máquinas virtuais (*VirtualBox*) com Ubuntu 16.04 LTS. A cada máquina virtual foram alocados 12GB de RAM (11GB de *swap*), 2 processadores a 2,5 GHz e um SSD com 40GB de espaço total disponível. Em cada máquina, foi instalada a distribuição HDP. Foi utilizada esta distribuição para estudar o funcionamento, na

prática, de um ecossistema de Big Data, implementar as aplicações de acordo com as arquiteturas a testar com as tecnologias disponibilizados em cada máquina e comparar as métricas obtidas durante o funcionamento da arquitetura *Lambda* proposta com uma arquitetura *Kappa* funcionalmente equivalente para obter uma comparação de desempenho, carga para o sistema, entre outros.

Os recursos disponíveis, especialmente a nível de número de cores e de quantidade de máquinas, estão longe de serem suficientes para testar ao limite e de forma distribuída as tecnologias propostas. Por exemplo, o tamanho recomendado da memória *Heap* reservada para os *DataNodes* é de 1GB, o que não será possível alocar. O mesmo acontece com componentes do *Spark* por exemplo. A disponibilidade de *hardware* adequado a executar este tipo de sistemas constitui uma limitação aos testes a realizar. Por isso, a comparação feita é especialmente focada no desempenho e na variação dos consumos entre períodos de teste e antes dos mesmos.

Das quatro máquinas, duas foram usadas para implementar arquiteturas *Kappa*, variando entre as duas apenas a tecnologia que era utilizada para processamento. As restantes duas foram usadas para implementar as arquiteturas *Lambda*. As implementações das arquiteturas *Kappa* e das arquiteturas *Lambda* são funcionalmente equivalentes, para garantir a maior confiança nos testes feitos. Os resultados de performance e consumo de recursos são no fim comparados entre todos os sistemas, com maior foco nas comparações entre os sistemas com a mesma arquitetura, para destacar a influência de cada tecnologia em cada um dos casos.

Para suportar os testes, foi também desenvolvida uma aplicação em *Python* para simulação de dados de sensores que, em simultâneo, trata de garantir a monitorização dos recursos consumidos no sistema.

5.1.2 Instalação e Configuração dos Sistemas

Para desenvolver os sistemas de teste, como referido anteriormente, começou-se pela instalação e configuração do próprio sistema operativo em máquinas virtuais.

Há várias boas práticas e ações recomendadas, por exemplo, pela *Hortonworks* quanto à preparação de um servidor para ser um nó de um *cluster* de Big Data. Estas práticas são seguidas no procedimento ilustrado acima (Figura 30).

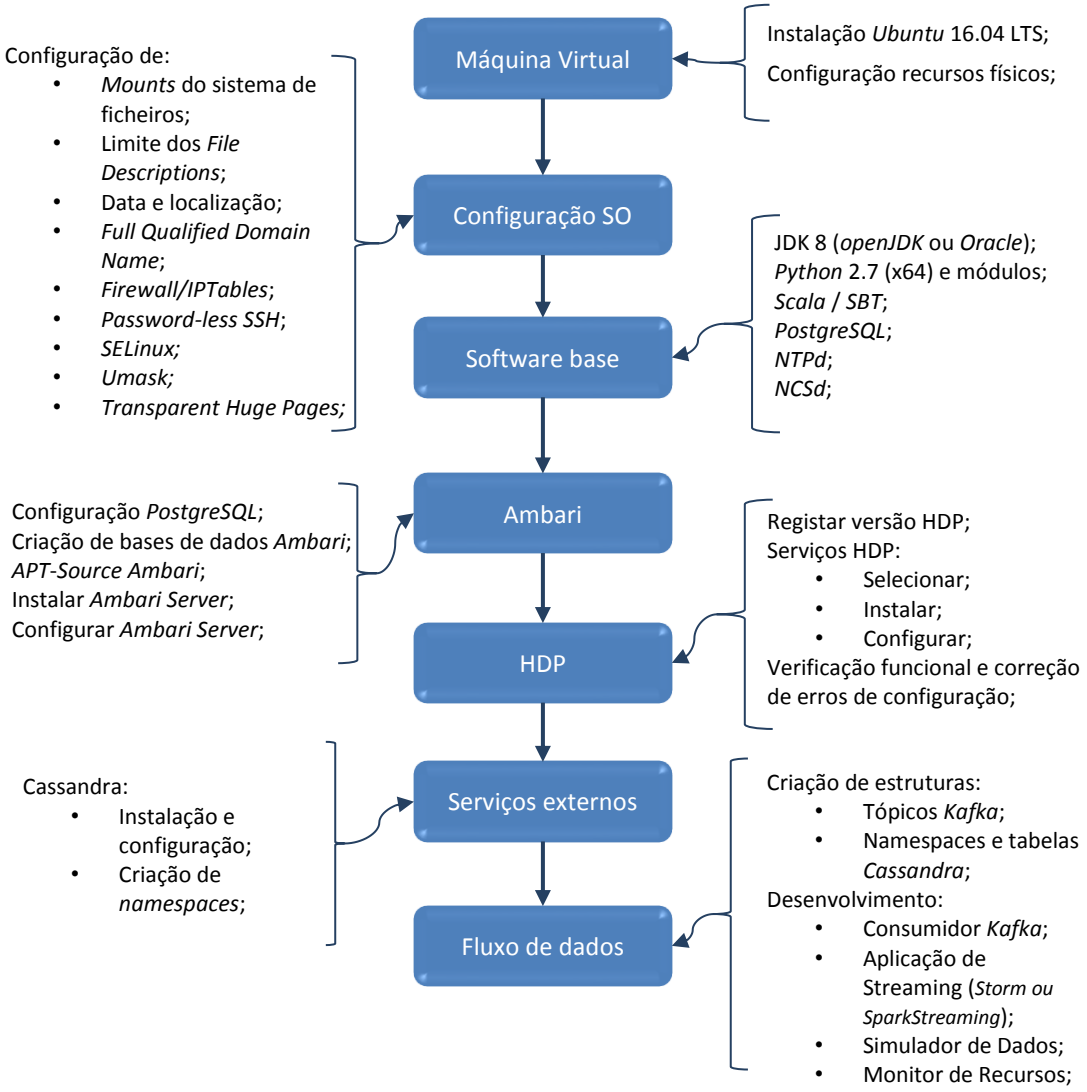


Figura 30 – Fluxo de implementação de um sistema de teste de Big Data

Os conjuntos principais de ações de instalação e configuração podem ser agrupados da seguinte forma:

- Configurações base do sistema operativo, ou pré-configurações – Estas consistem no conjunto de configurações necessárias para garantir que o sistema vai funcionar de acordo com o esperado. Aqui surgem:
    - configurações de como garantir que o sistema de ficheiros apresenta os volumes necessários;
    - configurações recomendadas para o limite de *File Descriptors* abertos, o que se não for tido em conta leva facilmente a erros complicados de diagnosticar;
    - configurações de data e localização, que são particularmente importantes na configuração de um *cluster*. Nas tecnologias que funcionam de forma distribuída todas as máquinas devem ter a mesma data e hora para o bom funcionamento dos serviços;

    - configuração do FQDN (*Full Qualified Domain Name*), que consiste numa boa prática para tratar a máquina pelo nome no domínio em vez do IP. Isto é particularmente importante na configuração de um *cluster*;
    - configurações de Firewall/IPTables, que devem ser configuradas com especial atenção. Os serviços instalados em cada nó de um *cluster* devem conseguir tanto poder ser acedidos de fora como aceder eles mesmos a serviços externos. Uma má configuração de rede quanto à Firewall/IPTables pode criar barreiras de comunicação;
    - configuração de *password-less SSH,* que é mais um dos pontos recomendados de que não se pode abdicar. Para executar operações, o *Ambari*, que é a ferramenta de gestão da distribuição da *Hortonworks*, abre ligações por *SSH* para o servidor destino, mesmo que o destino seja o próprio. Esta funcionalidade espera conhecer qual o certificado usado para configurar a ligação sem *password* entre os servidores e, portanto, é uma configuração obrigatória;
    - configuração do *SELinux* (*Security Enhanced Linux*), que deve estar especificamente ativo ou inativo durante alguns dos passos da instalação;
    - configuração de *umask, que* é também relevante para garantir que os ficheiros de configuração e temporários criados pelos serviços têm as permissões esperadas.
- Instalação de software base necessário para o funcionamento do sistema – É necessário ter disponíveis no sistema o *Java* e o *Python,* com as versões recomendadas. Adicionalmente, e dependendo do trabalho que se pretender desenvolver, pode ser recomendado usar *Scala* e *SBT/Maven* (usados, por exemplo, para desenvolver topologias do *Storm*). O *PostgreSQL* é opcional, uma vez que o próprio *Ambari* Server pode instalar uma base de dados. No entanto, para facilitar a gestão e manutenção é recomendado fazer a própria instalação. Por fim, é recomendado instalar serviços de *NTP* para garantir que todos os nós de um *cluster* apresentam o mesmo tempo e o *NCSd* para *caching* de *DNS*;
- Instalação e configuração do serviço do *Ambari Server* – Para a instalação do *Ambari Server* é recomendado primeiro transferir para o repositório gestor de pacotes do servidor (*APT* no *Ubuntu*) o repositório do *Ambari*. Posteriormente, este é instalado pelo gestor de pacotes para facilitar a gestão e atualização. Uma vez instalado, é necessário configurar o *Ambari Server* antes de correr o serviço pela primeira vez. Neste momento, é necessário ter o *PostgreSQL* devidamente configurado e com as estruturas esperadas (base de dados, tabelas, relações entre tabelas, etc);
- Instalação e configuração dos serviços disponibilizados pela distribuição HDP – Quando o *Ambari Server* está em execução, vai disponibilizar uma página web que é usada para gerir o *cluster* (de 1 ou mais servidores). Para começar o processo, foi escolhido o único nó disponível para fazer parte do *cluster* onde foram instalados os serviços necessários para cada teste a fazer posteriormente. Após a instalação dos serviços é sempre

recomendado fazer as devidas verificações funcionais (*service checks*, *smoke tests*[22], entre outros);

- Instalação de serviços não incluídos na distribuição HDP – A distribuição da *Hortonworks* é completa e procura selecionar pelo menos uma tecnologia por cada género. A escolha para base de dados de estrutura colunar da *Hortonworks* é o *HBase,* pelo que é necessário fazer uma instalação separadamente do *Cassandra* no servidor. O *plugin* existente tem vários erros, por isso a configuração do *Cassandra* terá que ser feita de forma isolada, ao passo que a dos restantes componentes pode ser feita através do *Ambari*;
- Implementação do fluxo de dados – Por fim, uma vez que todos os serviços necessários estão devidamente instalados e configurados, é possível começar a preparar as estruturas necessárias para o fluxo de dados, como criar e configurar os tópicos de *Kafka* e criar os *namespaces* e tabelas do *Cassandra* e as aplicações para os fluxos de dados.

Os pontos anteriores indicam o procedimento base usado para a instalação de todos os sistemas. Podem, no entanto, ser necessárias algumas configurações e *software* base diferentes se os objetivos do sistema forem diferentes. Por exemplo, num servidor *Hadoop* que se destine a operações de exploração de dados e visualização, além de um reforço considerável nos módulos de *Python* a instalar, seria também boa prática instalar e configurar tanto o *R* como a biblioteca *SparkR*.

5.1.3 Medições e Análises de Resultados

Atendendo a que o objetivo é medir os diferentes consumos das configurações em teste, as medidas dos recursos enquadram-se em dois tipos:

1. Medidas de Consumo. Estas vão corresponder a tabelas de consumos instantâneos como:
    a. *CPU*, medido em percentagem. Isto é relevante tendo em conta que o *hardware* virtual base será o mesmo para todos os sistemas em teste;
    b. *Memória*, medida em percentagem. Corresponde ao consumo de memória geral do sistema;
    c. *DiskIO*, medida em MB/s. Corresponde à taxa de transferência de escrita e leitura do disco.
2. Medidas de Desempenho. As medidas de desempenho deverão identificar se o sistema é eficiente independentemente dos recursos que usar. Um sistema pode, por exemplo, ser mais eficiente, mas consumir mais recursos de forma geral. A medidas a usar serão:
    a. *Tempo de Consumo*, medido em segundos. Corresponde ao tempo decorrido entre a geração do evento no simulador e o consumo desse evento do Kafka, ou seja, até ao momento em que o evento entra no sistema analítico;

[22] No contexto, os *smoke tests* são aplicações ou funcionalidades disponibilizadas pelas próprias tecnologias para serem executas após instalação para despistar problemas de configuração.

b. *Tempo de Resposta*, medido em segundos. Corresponde ao tempo decorrido entre a geração do evento no simulador até aos dados não tratados desse evento serem visíveis para o utilizador em pesquisas no *Cassandra*.

Poderia ainda ser utilizada para esta análise uma métrica para identificar o tempo decorrido entre a entrada do evento no sistema e a disponibilização de cada valor processado. No entanto, o processamento dos dados consiste em agregações implementadas com uma janela temporal, que só no fim do período vai poder disponibilizar as métricas. Assim, este processo vai sempre ocorrer no mesmo período e a métrica não seria representativa de desempenho do sistema.

As medidas usadas permitem ter uma noção da eficiência dos sistemas comparados e de quão exigentes são para o *hardware*. Ao mesmo tempo, permitem ganhar sensibilidade do tempo de resposta esperados das tecnologias testadas, o que facilitará a análise de requisitos para aplicações futuras.

A metodologia de avaliação consiste em monitorizar os sistemas testados em funcionamento regular. As monitorizações são todas correspondentes ao mesmo período de tempo, cerca de 7 minutos, com taxa de entrada de dados constante durante os primeiros minutos 2 a 3 minutos.

Desta monitorização deverão resultar amostras significativas de:

- CPU [%];
- MEM [MB];
- DiskIO (*Throughput*) [MB/s];
- Tempo de Consumo [s];
- Tempo de Resposta [s].

Para a medição do tempo da resposta, a cada evento gerado são associados alguns atributos de tempo ao longo do processamento. A cada evento é adicionado um atributo de tempo no momento da criação do evento, um atributo de tempo na leitura do evento no módulo de processamento e um novo atributo de tempo no fim do processamento de cada evento, quando fica pronto para escrita na base de dados. Há ainda o registo disponível no *Cassandra*, do momento da escrita de cada evento. Estas medições permitem tanto ter uma noção do tempo que cada evento demorou a estar disponível para visualização, como permite perceber qual foi ponto do processamento onde o evento demorou mais tempo, entre outras conclusões possíveis.

Para cada sistema de teste implementado, há várias execuções do simulador para gerar estas métricas. No caso específico das tecnologias usadas, existem alguns cuidados a ter para obter valores representativos só dos consumos do processamento porque vários componentes são *lazily loaded*, ou seja, são carregados apenas quando são precisos em tempo de execução. Por isso, é importante ter todos os processos de funcionamento contínuo a executar e em *IDLE*, correr o processo de simulação uma primeira vez com um conjunto de dados pequeno, de alguns segundos, só para garantir que todos os componentes são devidamente carregados e só

então fazer de facto uma execução do simulador com o propósito de guardar os dados como resultados a usar para a análise.

Em relação à responsividade do sistema, é importante ter noção destes resultados para fazer a melhor escolha, tanto das tecnologias, como dos métodos de análise dos dados ao fazer a primeira seleção da distribuição, tecnologias, arquitetura e casos de uso a implementar num contexto real. Este método de análise permite ganhar sensibilidade para estas questões constituindo um ponto base de comparação para eventuais análises futuras que recaiam sobre as mesmas arquiteturas.

Neste caso, as hipóteses que se procuram testar, em linha com a análise teórica, são as seguintes:

> *H1*. Os sistemas desenvolvidos com *Spark* são mais eficazes que os restantes sistemas testados.
>
> *H2*. Uma arquitetura baseada em *streaming* (*Kappa*) resulta num sistema mais responsivo do que se o sistema fizer processamento em *Batch* (*Lambda*).
>
> *H3*. Sistemas com tecnologias menos dependentes de operações em disco são mais responsivos do que sistemas com mais operações em disco.

Para as primeiras 2 hipóteses (H1 e H2), as amostras são comparadas entre si, isto é, os consumos de CPU de um sistema são comparados com os consumos de CPU dos restantes sistemas testados. O mesmo se aplica aos restantes parâmetros a serem medidos. As amostras apresentam uma distribuição normal, e são amostras independentes visto não estarem relacionadas entre si. Dada a probabilidade de falha no processamento de eventos, é possível que as amostras (série temporal) a comparar não tenham exatamente o mesmo tamanho. Neste sentido, são feitas várias execuções com cada sistema, com as mesmas condições em quantidade estatisticamente significativa, em que para cada série temporal será extraída a média de cada variável lida.

Para a última hipótese, as condições da amostra mantêm-se, mas neste caso é necessário correlacionar dois parâmetros: a taxa de escrita/leitura do disco e a responsividade do sistema. Apesar de não ser esperado que a relação entre eles seja linear, é esperado mostrar que estes parâmetros se influenciam.

5.2 *Stack* Usadas e Aplicações de *Streaming*

Para a comparação das tecnologias de *streaming*, como referido em 5.1.1, foram instaladas e configuradas duas máquinas virtuais com a distribuição HDP. Em cada uma, foi desenvolvido o mesmo caso de uso para gerar métricas que permitam comparar a performance do *Spark Streaming* com o *Storm*.

5.2.1 Simulador de Dados

O simulador de dados foi implementado com a intenção de conseguir simular, de forma controlada, um volume conhecido de dados para promover a repetibilidade entre simulações diferentes. A utilização do mesmo simulador para todos os testes de todos os sistemas, dá também mais confiança aos resultados pois, como é idêntica em todos os sistemas, não vai fazer variar de forma imprevisível as medições dos recursos.

Outra funcionalidade implementada no simulador de dados, consistiu na monitorização de recursos consumidos no sistema. Desta forma, o simulador assume as duas funcionalidades, simulação de dados e monitorização de recursos, o que permite controlar melhor os consumos e ajuda a distinguir os recursos consumidos nos diferentes momentos: antes, durante e após a simulação.

O simulador de dados, o *SensorDataSimulator.py*, foi implementado em *Python*, com uso dos módulos *psutil*, para suportar as operações de monitorização de recursos, e o *confluent_kafka* o qual foi utilizado para produzir os dados para o *Kafka*.

O simulador aceita como parâmetros o endereço do *broker* de *Kafka* ao qual se vai ligar, o nome do tópico do *Kafka*, o número de sensores a simular e o número de eventos a gerar por sensor (Figura 31).

SensorDataSimulator.py | SensorDataStreamHandler.py | Metrics.py

```
261     # validate inputs
262     if len(sys.argv) != 5:
263         #print("Usage: kafka_wordcount.py <zk> <topic>", file=sys.stderr)
264         print "Usage: SensorDataSimulator.py <brokers> <topic> <num_sensors> <num_writes>"
265         exit(-1)
266
267     # Get inputs from terminal
268     brokers, topic, num_sensors, num_writes = sys.argv[1:]
269
270     # Run data simulator task:
271     # RunProducerTask ( brokers, topic, int(num_sensors), int(num_writes) )
272     runResourcesMonitorTasks (test_duration, offset_duration, brokers, topic, num_sensors, num_writes)
```

Figura 31 – Validação do número de argumentos no *SensorDataSimulator.py*

Após a validação dos parâmetros de entrada, o simulador vai monitorizar o consumo de recursos do sistema durante 10 segundos antes de iniciar a produção de dados simulados. A monitorização (Figura 32) durante este período é necessária para estabelecer um valor de referência dos consumos do sistema sem carga. Esta noção permite compreender o real impacto do fluxo de dados em funcionamento. Nesta fase, é esperado que todos os serviços e aplicações estejam a correr, ainda que sem dados no fluxo para processar.

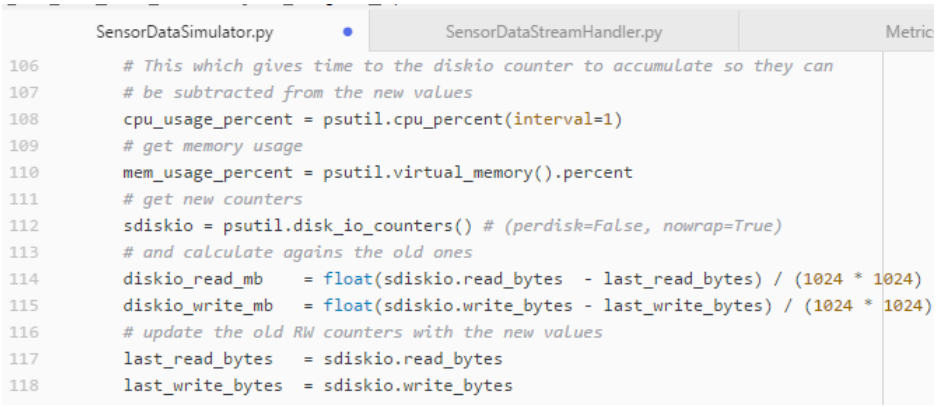


Figura 32 – Tarefa de monitorização de recursos no *SensorDataSimulator.py*

A monitorização dos recursos continua até ao fim no fim dos 10 segundos iniciais, mas a partir desse momento é iniciada uma tarefa paralela que vai iniciar a simulação dos sensores e o envio dos eventos destes para o *Kafka* (Figura 33). A monitorização continua a ser executada em paralelo com o processo de simulação e vai durar tanto tempo quanto configurado. O simulador não consegue prever quanto tempo o resto do sistema vai demorar a processar e armazenar os dados gerados e, portanto, optou-se por criar uma configuração para controlar o período total de monitorização.

Para que os valores simulados sejam facilmente analisados no final do processo, o simulador gera a mesma informação para todos os sensores, em sequência. A noção da ordem e dos valores enviados permite ter uma ideia da capacidade de manter a ordenação no consumo desta mesma informação, o que foi importante ao fazer o cálculo de métricas em janela.

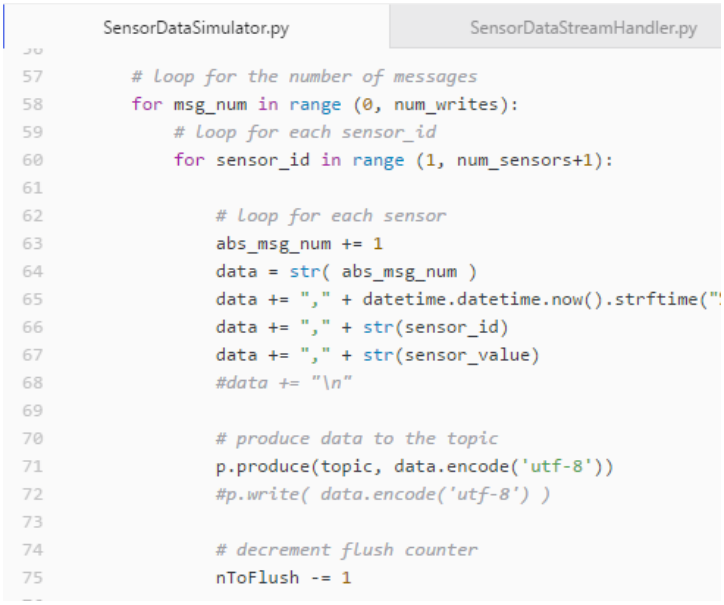


Figura 33 – Produção de eventos para Kafka

A publicação dos dados gerados no simulador, implementada com a API *confluent_kafka* (Figura 33), consiste em enviar as mensagens geradas no simulador para o tópico de *Kafka*. Esta publicação gerou erros quando feita com grandes volumes de mensagens, pelo que foi necessário analisar quais os parâmetros que se podiam configurar no *Producer* e qual a melhor forma de controlar esta parte do fluxo. Para diagnosticar o problema, primeiro foi analisado o comportamento da publicação dos dados para o tópico sem controlo da operação de envio dos dados, o *Producer.flush*. Sem este controlo, a API usada teve problemas de memória ao fim de poucos segundos, porque os dados estavam a ser acumulados na memória reservada pela instância do *Producer*. A segunda abordagem foi forçar a chamada do *Producer.flush* a seguir a cada registo submetido (*Producer.produce*). Esta abordagem resolveu o problema de memória, mas diminuiu, significativamente, a velocidade com que era feita a publicação dos dados. Este défice na velocidade fez com que não fosse possível obter uma alta taxa de mensagens publicadas para o sistema e, portanto, não se adequava ao tipo de simulação que se pretendia fazer. A terceira abordagem consistiu em forçar o *flush* ao fim de um número pré-definido de registos submetidos. Esta terceira abordagem foi experimentada com valores entre 10 e 5000 registos. Os valores que se traduziam em carga mais estável, sem grandes picos para o sistema, estavam entre os 100 e os 500 registos antes de cada nova chamada do *flush*. 100 registos por *flush* foi o valor usado nas simulações a partir de desse ponto.

5.2.2 Sistemas para Comparação de Tecnologias de Streaming

Para fazer a comparação entre as tecnologias foram, como referido anteriormente, instaladas duas máquinas com arquitetura *Kappa*, uma com *Spark* e outra com *Storm*. Foram também instaladas duas máquinas com a arquitetura *Lambda*, uma com *Spark* e outra com *Storm*. Desta forma, entre os quatro sistemas implementados varia apenas a arquitetura seguida na implementação, o motor de processamento e as respetivas dependências. As aplicações desenvolvidas para o caso de uso são também elas o mais equivalente possível entre si.

Sistemas com Spark Streaming

Como referido anteriormente, o objetivo para estes sistemas foi implementar uma aplicação de *streaming*, com *Spark Streaming*. Nesse sentido, a *Stack* usada na instalação e configuração do servidor foi concluída com os seguintes sete serviços (Figura 34):

- *Kafka*. Este é um serviço comum entre os sistemas implementados. É o responsável por receber os dados dos sensores simulados e deixá-los disponíveis para consumo interno;
- *Zookeeper*. O *Kafka* depende do *Zookeeper* para manter o estado e armazenar parte da informação (*metadata*) sobre os tópicos e consumidores por exemplo;
- *Spark2*. O *Spark* é o responsável pelo processamento de dados em *microbatches* através das funcionalidades implementadas na biblioteca *Spark Streaming*. Foi usada a versão 2.1 do *Spark*;

- *Hadoop Framework*. O *Spark* depende dos serviços disponibilizados pelo *Hadoop* para a implementação deste demonstrador. Desta forma, foram também instalados no servidor o *HDFS*, o *YARN* e o *MapReduce*. É também necessário ter os serviços *Hive Metastore* e *HiveServer2* disponíveis no sistema para o *Spark* poder utilizar devidamente a interface *SparkSQL*. Por esse facto, foi necessário instalar também estes serviços;
- *Cassandra*. Da mesma forma que o *Kafka*, o *Cassandra* é um serviço comum entre os sistemas implementados. É usado como base de dados para garantir a alta disponibilidade tanto dos dados originais dos eventos (*raw*), como das métricas processadas.

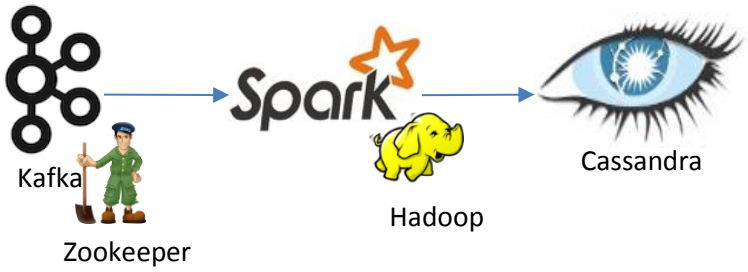


Figura 34 – Tecnologias usadas em sistema de teste com *Spark Streaming*

Sistemas com Storm

Para a implementação da aplicação de *streaming* com o *Storm*, a instalação e configuração dos sistemas foi concluída com os seguintes quatro serviços (Figura 35):

- *Kafka*. Tal como no sistema anterior, o *Kafka* é aqui usado para alojar os eventos que dão entrada no sistema;
- *Storm*. O *Storm* é neste caso o motor de processamento, no qual serão implementadas as topologias necessárias à aplicação final;
- *Zookeeper*. O *Storm* e, como descrito acima, o *Kafka* utilizam as capacidades do *Zookeeper* para manter o bom funcionamento do sistema e, portanto, este foi instalado como dependência;
- *Cassandra*. Tal como no sistema anterior, o *Cassandra* é o serviço usado para a *Serving Layer*, comum entre todos os sistemas.

Figura 35 - Tecnologias usadas em sistema de teste com *Storm*

Como se pode compreender pela composição dos sistemas, o *Spark* implica mais dependências e, portanto, é esperado que tenha um maior impacto nos recursos do sistema. Isso não implica que tenha défice de desempenho e que neste caso de uso não tenha um melhor desempenho que o *Storm*.

A semelhança entre os sistemas, onde varia apenas a tecnologia usada para processamento de eventos e as respetivas dependências, permite fazer a análise comparativa tanto do desempenho da tecnologia como do impacto para o sistema, mesmo apesar das limitações de recursos.

5.2.3 Aplicações de *Streaming*

O objetivo destas aplicações é, sucintamente, consumir dados em repouso no *Kafka* em *streaming*, calcular métricas para cada sensor simulado e armazenar a informação processada na *Serving Layer*, onde as métricas e os dados dos eventos ficam disponíveis para acesso imediato por terceiros.

As aplicações de *streaming* usadas para a avaliação de cada sistema, foram desenvolvidas de forma a serem o mais semelhantes possível tanto a nível de operação como a nível de estruturas usadas, apesar de inicialmente terem sido implementadas em linguagens de programação diferentes.

Para a implementação do processamento em *streaming*, pretendia-se gerar valores agregados dos dados dos sensores simulados em janela. Por exemplo, definindo que o intervalo da janela seria um minuto, a cada minuto deve ser gerada uma métrica de valores agregados para cada sensor. Para simplificação da implementação, as métricas definidas inicialmente foram apenas um somatório e a média para cada conjunto de valores.

Nas experiências feitas de simulação e agregação de dados, verificou-se que os recursos usados pelo processamento eram crescentes a cada período da janela. Isto devia-se a toda a informação de uma janela ser guardada em memória durante o tempo definido por cada sensor. Ao fim de algumas simulações, com o intuito de analisar o comportamento do sistema nesta situação, confirmou-se que os recursos consumidos cresciam de forma aproximadamente linear com o aumento do número de sensores simulados. Este comportamento numa aplicação real de IoT poderia causar problemas de dimensionamento de recursos no *cluster* que executasse

estas agregações. Também devido à análise estar a ser feita em janela, a taxa de emissão de eventos mostrou ter um grande impacto no consumo de recursos. Para contornar este problema, foram implementados cálculos incrementais, descartando assim a necessidade de guardar em memória toda a informação da janela. Para o cálculo da média incremental, foi usada a seguinte expressão (conferir Figura 36):

$$A_i = A_{i-1} + \frac{v_i + A_{i-1}}{i}$$

em que,

- A_i – valor da média incremental calculado na iteração atual (i);
- v_i – valor a acrescentar à média;
- i – índice da iteração atual (iniciado em 1).

Metrics.py SensorDataStreamHandler.py

```
19        # Add Measure Method
20        def AddMeasure(self, measure):
21            # increment the index as a new value is coming
22            self.idx += 1
23            # SUM: add the value to the previous sum result
24            self.sum += measure
25            # i-AVG: calculare incremental average
26            self.avg = self.avg + ( measure - self.avg ) / float(self.idx)
```

Figura 36 – Implementação do cálculo de métricas de forma incremental

Com esta nova implementação, o tempo de cálculo do valor da média foi diluído entre as aquisições. Sempre que um sensor recebe um novo valor dentro da janela, esse valor é utilizado para atualizar as métricas e pode de seguida ser descartado. Esta "diluição" do cálculo reduziu o pico de processamento no fim do período da janela de análise, o qual atrasava o processo de escrita das métricas para a *Serving Layer*. O volume de dados armazenado passou a ser estático para cada sensor, independentemente da taxa de receção de eventos. As métricas calculadas permitem verificar se houve corrupção dos valores ou falha de envio na transferência dos eventos de um sensor, por comparação com as métricas dos restantes sensores.

5.2.3.1 *Spark Streaming* – Arquitetura *Kappa*

A primeira aplicação a ser implementada, o *SensorDataStreamHandler.py*, foi implementada em *Python*, com uso dos módulos de *PySpark*, o *Cassandra Driver* da *DataStax* e o módulo *KafkaUtils* do *Spark Streaming*.

SensorDataStreamHandler.py

```
19  # Run Spark imports
20  from pyspark import SparkContext
21  from pyspark.streaming import StreamingContext
22  from pyspark.streaming.kafka import KafkaUtils
23
24  # Run Cassandra imports
25  #from dse.cluster import Cluster
26  from cassandra.cluster import Cluster
27  from cassandra.query import BatchStatement
28  from cassandra import ConsistencyLevel
29  from cassandra.query import SimpleStatement
```

Figura 37 – Módulos de *Python* necessários para aplicação de *Spark Streaming*

Os resultados da implementação deste caso de uso com *Python* apresentavam desempenhos muito abaixo do esperado. Por exemplo, para tratar um *dataset* de entrada de 500000 eventos, que correspondia a cerca de 3 minutos de dados em fluxo contínuo, eram necessários perto de 8 minutos. Tal não era aceitável para um possível ambiente de produção e foram feitas várias tentativas de reconfigurar o método de consumo de dados, de processamento e de escrita para a base de dados. Nenhuma das abordagens resolveu de facto o problema apesar da otimização do acesso ao *Cassandra* ter conseguido algumas melhorias.

No desenvolvimento do *Spark*, são primeiro desenvolvidas as funcionalidades em *Scala* e só depois é criado o suporte para as restantes linguagens, com a preocupação de reaproveitar ao máximo o que já foi implementado para *Scala*. De acordo com o *feedback* da comunidade do *Spark*, parte das operações usadas estavam, em cada *microbatch*, a fazer transformações não eficientes entre o *dataframe* do *Pyspark* e o original do *Scala*. Por esse facto, optou-se por reimplementar o caso de uso em *Scala*. Com esta alteração, os resultados foram consideravelmente melhores, como se poderá constatar na análise de resultados feita nos próximos capítulos.

Para qualquer aplicação de *Spark*, é necessário instanciar um objeto do *SparkContext* - tipicamente atribuído à variável *sc*. Este é o ponto de entrada para as funcionalidades específicas do *Spark* enquanto ferramenta de processamento distribuído. É, por exemplo, através do *SparkContext,* que é possível comunicar com o *cluster Spark*, criar RDD (*Resilient Distributed Dataset*), *accumulators* ou *broadcastVariables* no *cluster*, correr operações paralelizadas e distribuídas por todos os nós de um *cluster*, entre outros. Cada instância da JVM (*Java Virtual Machine*) pode ter apenas um *SparkContext*. Na aplicação *SensorDataStreamHandler.py*, foi usado o *SparkContext* para criar o *StreamingContext* e assim ter acesso às funcionalidades de computação de fluxos do *Spark* (Figura 38).

SensorDataStreamHandler.py

```
303
304  # Configure the spark streaming context
305  ssc = StreamingContext(sc, batchIntervalSeconds)
```

sspeedmetrics.scala

```
56
57  val ssc = new StreamingContext(sparkConf, Seconds(1))
58  val sc = ssc.sparkContext
```

Figura 38 – Criação do *StreamingContext* do *Spark*

Com o *StreamingContext*, é possível instanciar um *directStream* (Figura 39). Este vai ser o objeto que permite aceder a dados do *Kafka* como um fluxo para posterior processamento. Dos métodos disponíveis para fazer a interface ao *Kafka* a partir do *Spark*, de acordo com a documentação, este é o método mais indicado. A interface feita ao Kafka com o *createStream*, tenta garantir que não há perda de dados através da semântica "*at least once*", o que implica que os dados podem ser repetidos no consumo, como contrapartida da garantia de que não se perdem mensagens. A interface feita ao *Kafka* com o *createDirectStream*, por sua vez, permite ter visibilidade, do lado do consumidor, dos offsets por cada *microbatch* (Figura 40) e deixa assim à responsabilidade do consumidor a gestão dos offsets e permite fazer um consumo seguindo a semântica "*exactly once*", que é a mais adequada para o sistema que se pretende (Kestelyn, 2015).

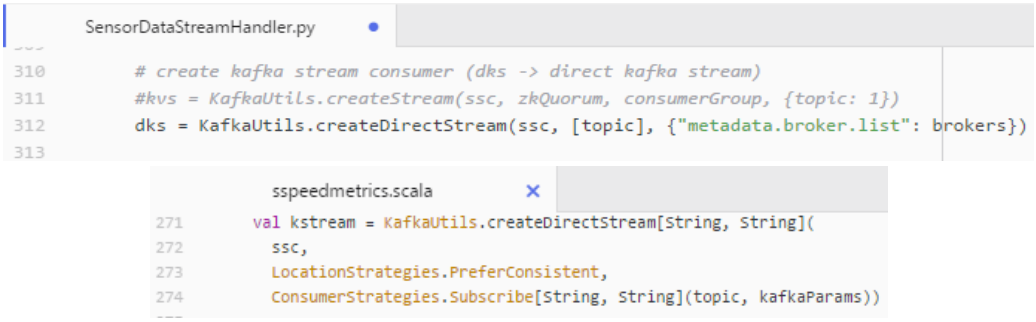


Figura 39 – Criação do consumidor de fluxos de Kafka com o *StreamingContext*

No objeto retornado pelo método *createDirectStream* é possível definir uma cadeia de eventos a executar sobre cada conjunto de mensagens lido do *Kafka*, como ilustrado na Figura 40. Depois de passar por todo este processo, é possível iniciar-se o tratamento das mensagens.

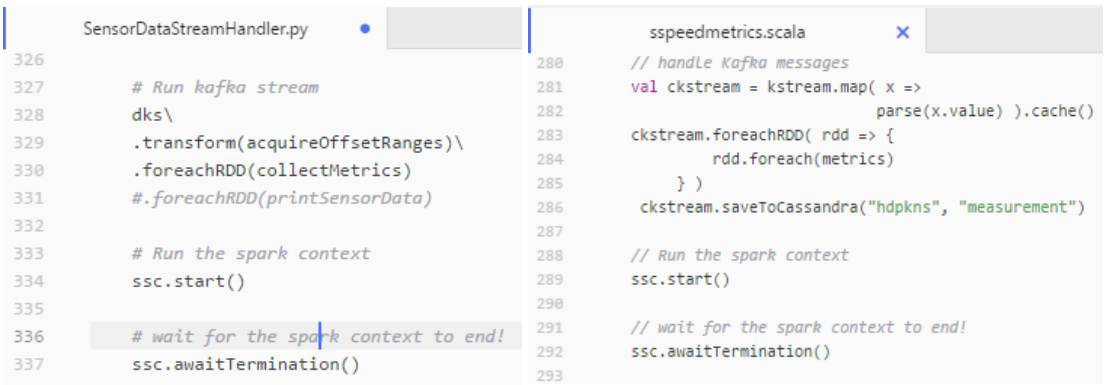


Figura 40 – Atribuição da cadeia de eventos a executar sobre o *stream*

O processo de implementação da aplicação passou por várias fases e experiências, nomeadamente:

- Foi implementado o processamento do *stream* de várias formas, como por exemplo, com todo o processamento sobre o RDD na primeira operação (*transform*, Figura 40), com tarefas assíncronas paralelas a comunicar por filas de mensagens os eventos

recebidos e com o processamento feito exclusivamente com operações *map()* para processar os dados e *foreachRDD* para armazenar;

- Foram variados vários parâmetros do fluxo como o intervalo entre execuções do *timer* do *Spark Streaming*, os recursos e o número de executores do sistema alocados à tarefa de *Spark*, o modo de execução entre o *local[*]* e o *YARN*;
- Foram experimentadas várias combinações entre os parâmetros do *Cassandra,* como por exemplo as referentes a escritas concorrentes, a consistência de escrita, o tamanho total de escrita por *batch*, entre outros;
- Para avaliar o desempenho exclusivamente da escrita para o *Cassandra*, em algumas das experiências feitas, foi ainda removido o código do processamento das métricas para garantir o mínimo de atrasos possível.

Por exemplo, foram feitas várias simulações com exatamente as mesmas condições à exceção do parâmetro *batchIntervalSeconds* (Figura 38). Este parâmetro controla o intervalo de tempo decorrido entre cada aquisição de dados do *Kafka* e tem um impacto significativo na aplicação. Se forem usados valores muito baixos, como 0,1 a 0,2 segundos, o processo de consumo acaba por ser lançado com uma frequência maior do que a velocidade com que a primeira operação consegue libertar os objetos em uso. Notou-se, nesta situação, que os recursos usados aumentaram consideravelmente e a aplicação começou a falhar eventos. Como consequência, o próprio consumo tendeu também a ser atrasado. Valores entre 0,5 segundos e 1 segundo mostraram-se mais eficazes tanto a nível de estabilidade dos recursos usados como a nível de desempenho. Quanto maior este parâmetro, mais dados tendem a ser consumidos do *Kafka* de cada vez.

Na implementação em *Python*, o processamento estava a ser feito de forma bloqueante na chamada do evento *collectMetrics*, o que se revelou um problema. Ocorreram diversos erros pelo facto do objeto RDD - objeto transferido entre as chamadas do fluxo – estar a ser usado para processamento. Foram tentadas algumas abordagens para resolver o problema. Uma delas consistiu em extrair a informação do RDD com o método *collect*, que retorna os conteúdos do RDD como uma lista de tuplos, e usar essa lista para processamento em vez do RDD. A solução resolveu o problema do bloqueio do objeto, mas como o processamento do volume de dados obtido demorava mais do que o intervalo entre os consumos, foi necessário seguir abordagens diferentes. Uma hipótese seria paralelizar o processamento e o consumo. Neste sentido, foram implementados dois processos adicionais: um para processar a informação e outro para escrever os eventos não tratados para a base de dados.

Depois das várias experiências e otimizações, chegou-se à solução final desta aplicação, sendo ela a que garantia o melhor desempenho no processamento em *streaming*. Para a versão final foram implementados dois processos (*tasks*) que são executados paralelamente ao evento *collectMetrics*. O evento apenas faz *collect* do RDD para prevenir colisão nas operações e insere cada tuplo recebido numa pilha partilhada entre este evento e a tarefa de processamento (Figura 41).

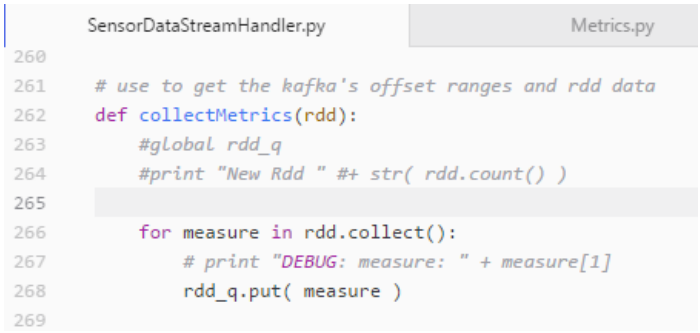


Figura 41 – Função que empilha os eventos do Kafka para processamento

A primeira tarefa paralela implementada (*worker*, Figura 42) foi a tarefa que trata de recolher dados da pilha partilhada e de os processar. Após o processamento das métricas, é construído o pedido a enviar ao serviço *Cassandra* para escrever o evento tratado na base de dados.

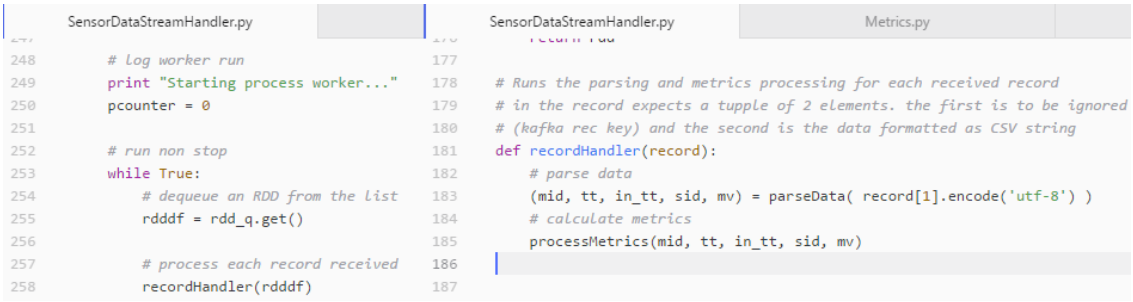


Figura 42 – Tarefa *worker*: recolhe dados da pilha interna e executa o processamento

Os eventos chegados ao *Cassandra Driver* inicialmente estavam a ser escritos de imediato. No entanto, a escrita desta forma estava a causar, em situações de carga, atrasos significativos na inserção de dados. Foram seguidas as boas práticas e as sugestões da comunidade do *Cassandra* para tratar este tipo de problemas. De acordo com a pesquisa é recomendado, em vez de processar de imediato a escrita, usar uma estrutura do *Cassandra Driver*, o *BatchStatement*, que consiste num pedido padrão e numa pilha de parâmetros a executar em *batch* com o padrão definido. Esta aparentou ser teoricamente uma solução adequada ao processamento em *microbatching*, e teve resultados razoáveis, reduzindo o tempo total de escrita de dados de cerca de 30 minutos para aproximadamente 10 minutos.

Para controlar a escrita para o serviço *Cassandra*, foi implementada a segunda tarefa paralela, o *cassandraWorker* (Figura 43), que procura escrever sempre que a pilha tem cerca de 3000 eventos no *BatchStatement* ou sempre que passam 5 segundos. O parâmetro da quantidade de eventos no *BatchStatement* foi calculado com base no espaço que os eventos ocupam e com base no máximo de *bytes* que o serviço *Cassandra* suporta por escrita. O parâmetro do tempo passado serve apenas para prevenir que dados fiquem por escrever no fim do processo de simulação, ou para forçar a escrita para a base de dados, no caso de simulações com baixa taxa e envio de dados.

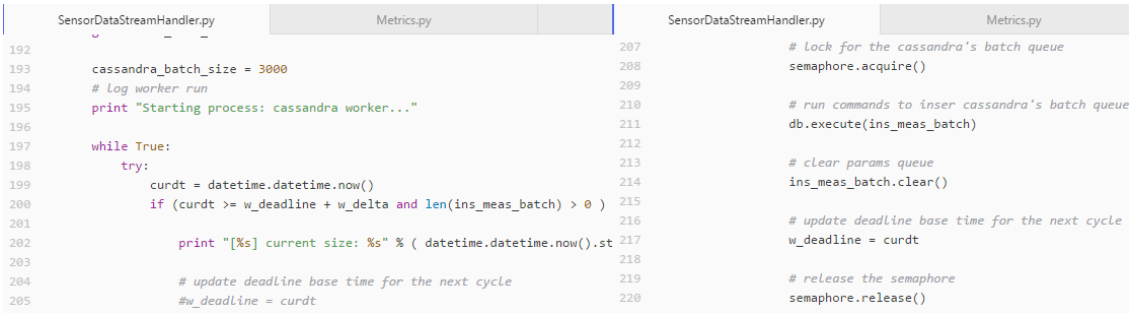


Figura 43 – Tarefa *cassandraWorker*: escreve eventos não tratados para *Cassandra*

Apesar das tentativas, a implementação de *Python* não apresentava resultados satisfatórios e que potencialmente pudessem ser considerados para um ambiente de produção. Na implementação do mesmo processo em *Scala*, foi seguido um procedimento equivalente à implementação em *Python*. O processamento foi dividido também em três partes: a chamada que recebe o RDD e faz parse da informação - equivalente ao *parseData* do *Python* -, a chamada que analisa os dados – equivalente ao *processMetrics* do *Python* – e por fim a chamada que escreve o RDD para o *Cassandra*. A implementação em *Scala* deste fluxo permitiu ainda mais algumas otimizações. A primeira otimização consistiu na forma como era interpretada a resposta do *parseData* (Figura 42). A primeira operação sobre o RDD foi, em vez de um *collect()*, um *map()* para converter os eventos chegados do Kafka de um RDD de *Strings* para um RDD de tuplos. Este RDD de tuplos é retornado de forma a ser usado no fluxo. Para prevenir outros problemas ocorridos na implementação em *Python* foi também forçada a manutenção deste RDD em cache até deixar de ser necessário (Figura 44).

```
// handle Kafka messages
val ckstream = kstream.map( x =>
                            parse(x.value) ).cache()
```

Figura 44 – *Map()* como primeira operação sobre os dados lidos em fluxo do *Kafka*

O próximo passo do fluxo é o tratamento de cada tuplo retornado no RDD do *map()*, o *ckstream* (Figura 44), onde se calculam e guardam as métricas para o *Cassandra*. Nesta operação, para cada evento do RDD é verificado se o seu *timestamp* de geração está dentro da janela de 1 minuto na qual se está a fazer a agregação das métricas. Se estiver fora da janela e for anterior, o evento não vai entrar para as métricas da janela. Se for um valor novo, assume-se que começa uma nova janela e o valor é incluído. Quando se identifica que é para começar uma nova janela, as métricas são escritas para base de dados e os contadores são reiniciados a 0. Por fim o RDD *ckstream* é escrito para o *Cassandra* (Figura 40).

```
def metrics( record: (Long, java.util.Date, Int, java.util.Date, java.util.Date, Double)

    val measure_window_size:Int = 60 // seconds
    var tocass : Seq[(java.sql.Timestamp,Int,Double,Double)] = Seq()
```

```
val winfit = isInMeasureWindow( (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS"))
                    .format(record._2), dt_window_offset, measure_window_size)
//println(winfit)
```

```
val (sidx, ssum, savg) = GetSensorMetrics(sid)
//tocass = tocass:+ (dt_window_offset, sidx, ssum, savg)
tocass = tocass:+ (new java.sql.Timestamp(dt_window_offset.getTime), sid, ssum, savg)
```

```
// reset metrics
AllSensorsReset()
```

```
if (tocass.length > 0) {
    rwlock.synchronized {
        println("tocass.length: " + tocass.length)
        sc.parallelize(tocass).saveToCassandra("hdpkns", "batch_metrics",
                            SomeColumns("tt", "sensor_id", "sum", "avg"))
    }
```

Figura 45 – Segmentos de código da chamada da função *metric*

5.2.3.2 *Storm* – Arquitetura *Kappa*

A implementação da aplicação de *Storm* para o processamento dos eventos gerados foi feita em *Scala*, que é uma das linguagens melhor suportadas para desenvolver nesta tecnologia. Ao contrário do *Spark*, no qual é criada a ligação ao motor de processamento pela instanciação do *SparkContext* ao executar a aplicação, no *Storm* é necessário compilar a aplicação num JAR e posteriormente submeter a topologia para o *Storm* (Figura 46).

```
To compile this topology and get the jar ready to upload:
$ sbt clean compile assembly

To upload this topology:
$ storm jar $STORMK_PROJ/target/scala-2.12/StormK-assembly-1.0.jar \
        com.tmdei.scala.storm.SensorDataTopology \
        tmdei-sensor-data-simulated
```

Figura 46 – Exemplo de comandos para compilar e submeter uma topologia para o *Storm*

Como descrito anteriormente, no capítulo 3.1.5.1, os componentes principais do *Storm* são os *Spouts* e os *Bolts* que constituem as topologias criadas. Os *Spouts* são tipicamente responsáveis por transferir/receber dados da fonte e encaminhá-los para processamento na cadeia de *Bolts* que se seguir. Para reduzir a entropia, mas manter a implementação o mais semelhante possível à implementação do *Spark*, foram implementados para esta topologia três componentes: um *Spout* para consumo de tópicos de *Kafka*, um *Bolt* para processamento de métricas e um último *Bolt* para gerir o processo de escrita dos dados para a base de dados. Obtém-se assim algum paralelismo entre a implementação dos três componentes do *Storm* e dos três processos usados no *Spark*.

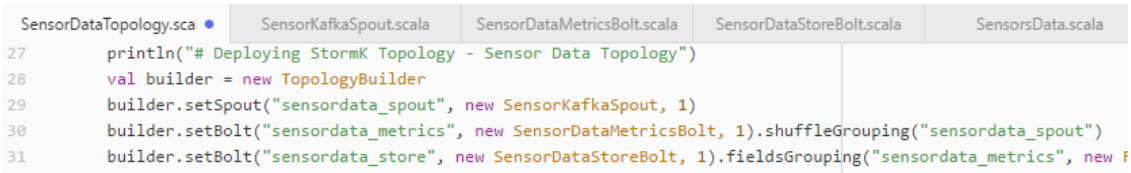


Figura 47 – Definição da topologia do *Storm*

Para o desenvolvimento de uma topologia, implementa-se o comportamento da mesma e a sequência de componentes que os eventos devem seguir. O processo para implementar uma topologia, consiste essencialmente em:

- Implementar os *Spouts* e os *Bolts* a usar de acordo com as interfaces disponibilizadas pelo *Storm* e o fluxo de dados pretendido. Para os *Spouts,* fazer *extends* à classe *BaseRichSpout* e para os *Bolts*, fazer *extends* à classe *BaseBasicBolt*;
- Implementar a topologia. Para isso, usar o *TopologyBuilder* para adicionar os *Spouts* e os *Bolts* necessários (Figura 47). É necessário configurar, para cada um o nome - que deve ser único entre todos os componentes adicionados, a interface do *Spout* ou do *Bolt* a usar - por exemplo, dos tipos *IBaseBolt* ou *IRichSpout* - e a recomendação para a paralelização, que o *Storm* vai usar para decidir a quantidade de executores que vai criar para o componente em questão. O último parâmetro é opcional, mas se for omitido o *Storm* vai considerar que apenas deve ser lançado um executor;
- Configurar a topologia. Há várias configurações possíveis na topologia. A única usada foi a *config.setNumberOfWorkers,* que configura o número de executores a criar alocados a esta topologia no *cluster* (Figura 48). Foi usada para limitar o número de executores criados devido aos recursos. As restantes configurações disponíveis não pareceram relevantes para o objetivo da aplicação a desenvolver;
- Por fim, usam-se as configurações definidas e o *TopologyBuilder.createTopology* para submeter a topologia (*StormSubmitter.submitTopology*).

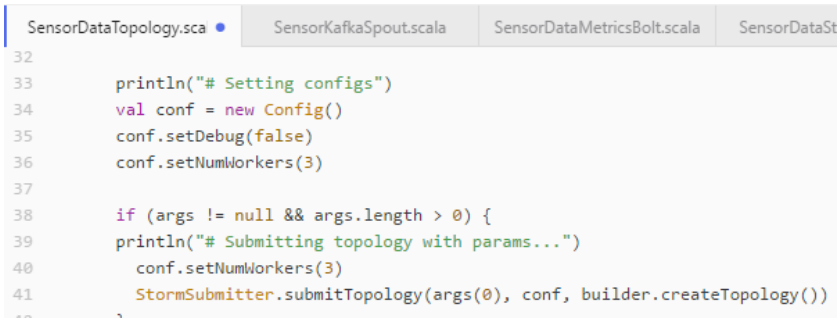


Figura 48 – Configuração e submissão da topologia

Na aplicação, o *Spout SensorKafkaSpout* tem a responsabilidade de consumir mensagens do tópico do *Kafka*, de fazer alguma conversão inicial necessária ao tratamento dos eventos nos *Bolts*, de acrescentar à mensagem o momento em que ela foi consumida (data e hora) e de encaminhar para o primeiro *Bolt* a mensagem. Para a implementação, utilizou-se a *interface BaseRichSpout*, da qual se fez *override* dos métodos *open*, *nextTuple* e *declareOutputFields*.

O método *open* é usado essencialmente para receber o *SpoutOutputCollector*, que vai ser usado para transmitir os eventos para o *Bolt* seguinte e para subscrever o tópico com o *KafkaConsumer* (Figura 49). O *KafkaConsumer* usado foi o da API de Java disponibilizada nas bibliotecas do *Kafka*. Alternativamente, poderia ter sido usado o *KafkaSpout*, mas a abordagem seguida permite um controlo mais detalhado da *interface* com o *Kafka* e, consequentemente, facilita a avaliação dos diferentes comportamentos ao variar os parâmetros.

SensorDataTopology.scala | SensorKafkaSpout.scala | SensorDataMetricsBolt.scala | SensorDataStoreBolt.scala | SensorsData.scala

```
33
34      override def open(conf: java.util.Map[_, _], context: TopologyContext, collector: SpoutOutputCollector): Unit = {
35        _collector = collector
36        _rand = Random
37
38        // kafka consumer properties
39        val  props = new Properties()
40        props.put("bootstrap.servers", "hdp.dei.isep.ipp.pt:6667")
41        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer")
42        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer")
43        props.put("group.id", "kconsumer")
44
45        // instantiate the Kconsumer
46        consumer = new KafkaConsumer[String, String](props)
47
48        // Subscribe the configured topic
49        consumer.subscribe(util.Collections.singletonList(TOPIC))
50      }
```

Figura 49 – Implementação do método open do *SensorKafkaSpout*

O método *nextTuple* é o método que está consecutivamente a ser chamado pelo *Spout* para promover o consumo de dados das fontes (Figura 50). Foi no *nextTuple* que foi implementado o consumo do tópico. Este método permite controlar a quantidade de mensagens a consumir por cada *poll*. À semelhança do que foi experimentado no *Spark*, ao variar o valor em *batchIntervalSeconds*, verifica-se que ao aumentar a quantidade de dados no consumo do *Kafka*, obtém-se um melhor desempenho de consumo. Contudo, o *Storm* está desenhado para processar evento a evento, e para emitir os eventos para os *Bolts* é necessário extraí-los da lista de mensagens lida.

SensorDataTopology.scala | SensorKafkaSpout.scala | SensorDataMetricsBolt.scala

```
52      override def nextTuple(): Unit = {
53
54          // retrieve data from the kafka topic
55          val records_as = consumer.poll( _kafka_batch_size ).asScala
56          //fw.write( "# SensorKafkaSpout @ data retrieved (" + records_
57
58          // evaluate if there is in fact any message coming or if it ju
59          if( records_as.size > 0 )
60          {
61              // iterate over all read registers
62              for (sensordata <- records_as.toList) {
63                  if ( sensordata.value().toString.length() > 0 ) {
64                      // send data forth ( only if there is any data at
65                      _collector.emit( new Values( sensordata.value().t
```

Figura 50 – Implementação do método *nextTuple* do *SensorKafkaSpout*

O método *declareOutputFields* (Figura 51), é usado para declarar os campos que vão ser emitidos, seja por um *Bolt*, seja por um *Spout*. Neste caso, é passado o parâmetro identificado pelo nome de "*sensordata*". O valor emitido é o valor originalmente lido do tópico do *Kafka* concatenado com o valor da data e hora do momento desta leitura, para identificar o momento temporal do consumo.

Figura 51 - Implementação do método *declareOutputFIelds* do *SensorKafkaSpout*

Os valores emitidos pelo *SensorKafkaSpout* são recebidos pelo *SensorDataMetricsBolt* (Figura 52), que é uma extensão do *BaseBasicBolt*. Neste, é feito *override* dos métodos *execute* e *declareOutputFields*. O *declareOutputFields* do *SensorDataMetricsBolt* é usado da mesma forma que o anterior.

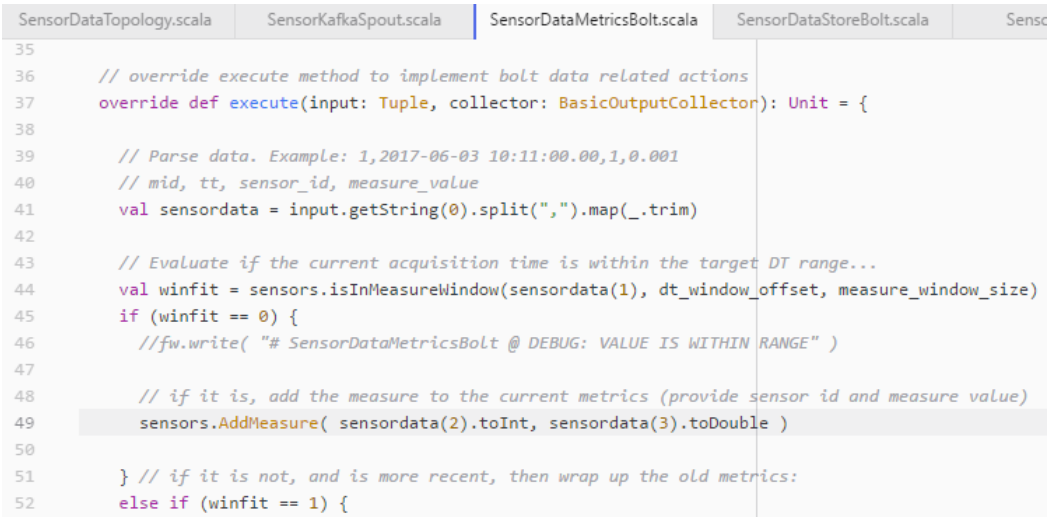


Figura 52 – Implementação do método execute do *SensorDataMetricsBolt*

As estruturas criadas para gerir o cálculo das métricas e a lógica implementada são idênticas às implementadas em *Python* no sistema implementado com *Spark*. Este facto é relevante para a implementação do método *execute*, que é chamado pela *interface* a cada evento recebido. Aqui são calculadas as métricas, mais uma vez em janelas de um minuto, e no fim de cada período os valores das métricas são transmitidos ao próximo *Bolt* para escrita na *Serving Layer* (Figura 53). As métricas calculadas são as mesmas agregações do processo implementado em *Spark*, uma soma dos valores recebidos e uma média calculada incrementalmente.

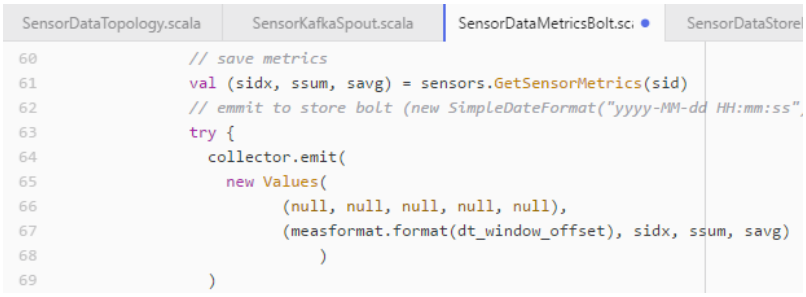


Figura 53 – Transferência de métricas calculadas no *SensorDataMetricsBolt*

Os valores emitidos pelo *SensorDataMetricsBolt* são recebidos pelo *SensorDataStoreBolt*. Este implementa uma gestão de escrita para o *Cassandra*. A ligação ao *Cassandra* é implementada com o *Cassandra Driver* da *DataStax*. A cada ciclo do método *execute,* é inicialmente verificado se a sessão para a base de dados está estabelecida e, se não estiver, cria-se uma nova. De seguida, é verificado se os valores recebidos são válidos e, se forem, é utilizada a interface ao *BatchStatement* para fazer acumular eventos a escrever para a base de dados, da mesma forma que foi implementado o controlo no caso do *Spark*. O *execute* é chamado sempre que há um evento novo e, se o processamento do evento não for rápido o suficiente, os eventos podem acabar por concorrer pelo acesso a recursos globais no contexto do *Bolt*. Este problema ocorre, por exemplo, no acesso ao *BatchStatement* (*batch_meas*, Figura 54) e, portanto, foi necessário usar um mecanismo para gerir o acesso concorrente de momentos críticos da execução deste método.

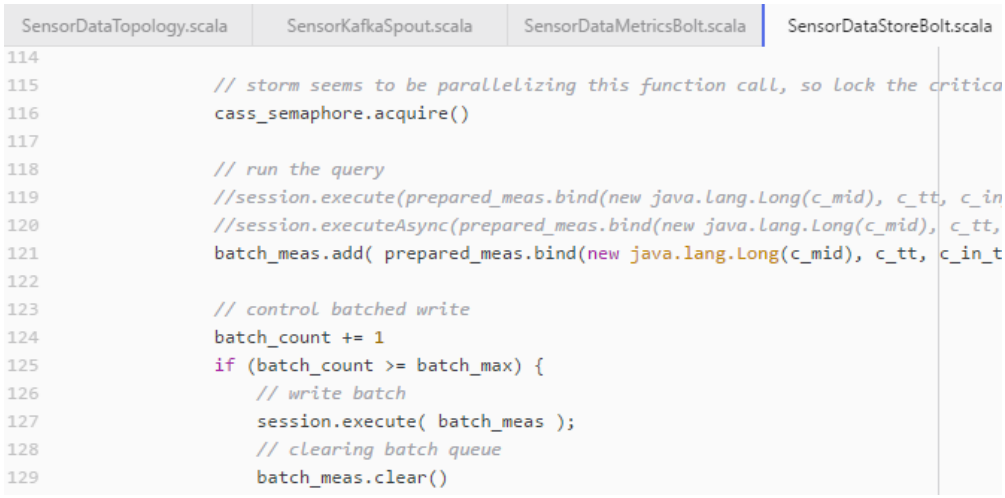


Figura 54 – Adição de eventos a escrever para o *Cassandra* e execução da escrita em bloco

Tal como no caso do sistema anterior, durante a implementação deste método foram testados os diferentes métodos para escrita para o *Cassandra*: o *session.execute(statement)*, o *session.executeAsync(statement)* e o *session.execute(batchstatement)*. O método que obteve melhor desempenho foi o *BatchStatement*.

5.2.3.3 *Spark Streaming* – Arquitetura *Lambda*

A implementação do caso de uso do *Spark Streaming* com a arquitetura *Lambda* foi baseado na aplicação de *Spark Streaming* desenvolvida em *Scala* seguindo a arquitetura *Kappa*. A arquitetura *Lambda* assume alguns pressupostos, como referido anteriormente (ver 3.2.3.2), e estes fazem com que seja necessário implementar esta aplicação com mais do que um processo. Para o caso, foram implementadas três aplicações separadas, uma para aquisição de dados e processamento na *Speed Layer* (*SensorDataSpeedMetrics*), uma para o consumo e armazenamento na *Batch Layer* (*SensorDataConsumer*) e outro para o processamento dos dados armazenados na *Batch Layer* (*SensorDataBatchMetrics*). Para simplificação, foi criado um script, o *spark-lambda-launcher.sh*, para parametrizar e executar as chamadas destas aplicações. Este script faz a chamada das aplicações *SensorDataSpeedMetrics* e *SensorDataConsumer* com *detach* das aplicações do contexto do script para não ser bloqueante, uma vez que estas aplicações serão de execução continua. Depois, o script chama periodicamente, com um intervalo de um minuto, a aplicação *SensorDataBatchMetrics*.

```
272       // handle Kafka messages
273       kstream
274            .map( x => parse(x.value) )
275            .foreachRDD( rdd => {
276                rdd.foreach(metrics)
277            } )
```

Figura 55 – Fluxo da aplicação *SensorDataSpeedMetrics*

A aplicação *SensorDataSpeedMetrics* é executada de forma contínua, e tem uma implementação muito semelhante à aplicação de *Spark Streaming* da arquitetura *Kappa* (Figura 55). Esta usa também a *interface* do Spark Streaming para aceder às mensagens do Kafka, *KafkaUtils.createDirectStream*, calcula as métricas com base nos eventos recebidos do *Kafka* e calcula as métricas geradas com base numa janela de 1 minuto. A maior diferença entre esta e a implementação da arquitetura *Kappa* é que os dados dos eventos não são armazenados após serem usados para o cálculo das métricas. As métricas calculadas neste script são escritas para a tabela *speed_metrics* do *Cassandra* (Figura 56).

```
sspeedmetrics.scala ×   ssmetrics.scala
234          if (tocass.length > 0) {
235                rwlock.synchronized {
236                  sc.parallelize(tocass)
237                            .saveToCassandra("hdpkns", "speed_metrics")
238                }
```

Figura 56 – Método usado para escrita de métricas calculadas para *Cassandra*

A aplicação *SensorDataConsumer* executa também de forma contínua e usa o mesmo método que as anteriores para consumir dados do *Kafka*. Neste caso, como vamos ter dois consumidores diferentes do mesmo tópico de *Kafka*, para estes não partilharem os índices de consumo é necessário utilizarem grupos de consumo diferentes (ver 3.1.3.1). A missão desta aplicação neste contexto é interpretar os eventos lidos do *Kafka* e convertê-los para um RDD

de tuplos que possa ser usado para armazenar, logo de seguida, no *Cassandra*, da forma mais eficiente (Figura 57). A *Batch Layer* poderia usar o *HDFS* ou o *Hive* para armazenar a informação em alternativa ao *Cassandra*, mas o *Cassandra* mostrou ter melhor desempenho e associa um registo temporal à escrita de cada registo, com resolução de microssegundos, pelo que se optou por usá-lo nesta fase também.

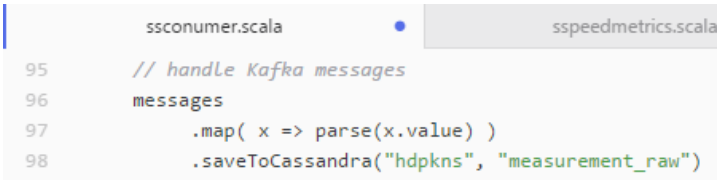


Figura 57 - Fluxo da aplicação *SensorDataConsumer*

A última aplicação deste conjunto, a *SensorDataBatchMetrics*, é uma aplicação executada de forma periódica pelo script. Esta aplicação vai ler os últimos 10 minutos do Cassandra que são armazenados pela aplicação *SensorDataConsumer*. Depois de carregar a informação, agrega os valores de cada minuto recalculando as métricas de uma forma mais robusta do que é feita na aplicação *SensorDataSpeedMetrics* e com mais garantias de que não ficarem dados por incluir nas métricas devido, por exemplo, à falta de ordenação dos eventos do Kafka.

Como descrito, para o cálculo das métricas primeiro é feita uma pesquisa nos dados escritos para o *Cassandra* nos últimos 10 minutos. O objeto retornado, o RDD, vai permitir interagir com uma sintaxe similar ao SQL (Figura 58). Por exemplo, para a agregação, dos valores lidos por sensor e por minuto, é usado o método *groupBy* disponibilizado no RDD. Uma funcionalidade que simplifica a execução de análises em janela sobre dados no *Spark* é possibilidade de definir o tamanho da janela a usar na agregação na própria pesquisa sobre o RDD. Esta é usada em conjunto com funções de agregação para fazer o cálculo dos somatórios e médias necessárias e retorna um RDD que pode ser usado para escrita diretamente para a tabela de métricas calculadas na *Batch Layer*.

Este procedimento previne que se percam dados que cheguem com até 10 minutos de atraso do *Kafka* ao *Spark*. Pela análise do procedimento da Figura 58, como o método de escrita para o *Cassandra* é *SaveMode.Append,* pode dar a entender que vão ser repetidos registos de métricas calculadas. Tal não acontece porque as colunas usadas para a indexação da tabela, data e hora da métrica e identificador do sensor, vão ser sempre os mesmos. Nesta condição, o comportamento do *Cassandra* é atualizar o registo que já conhece com a nova informação.

ssmetrics.scala | spark-lambda-laucher.sh

```
73      val measraw = ss.sqlContext
74        .read.format("org.apache.spark.sql.cassandra")
75        .options(Map("table" -> "measurement_raw", "keyspace" -> "hdpkns"))
76        .load()
77        .select("mid", "tt", "sensor_id", "in_tt", "out_tt", "measure")
78        .where("tt >= '" + measformat.format(cal.getTime()) + "'" )
79
80      // calculate metrics (group per sensor per minute)
81      val metrics = measraw
82        .groupBy($"sensor_id", window($"tt", "1 minutes"))
83        .agg(sum("measure").as("sum"), avg("measure").as("avg"))
84        .select($"window.start".as("tt"), $"sensor_id", $"sum", $"avg")
85
86
87      // update new cassandra table with calculated data
88      metrics.write
89        .format("org.apache.spark.sql.cassandra")
90        .options( Map("table" -> "metrics", "keyspace" -> "hdpkns") )
91        .mode(SaveMode.Append)
92        .save()
```

Figura 58 – Cálculo das métricas na aplicação *SensorDataBatchMetrics*

Para ser possível utilizar os métodos do *Spark* de escrita para a base de dados, é necessário ter ligação ao serviço *Hive Metastore*. Caso contrário, não é possível instanciar o *sqlContext,* que é usado por estes métodos de escrita, como o *saveToCassandra*. Desta forma, ao resolver a cadeia de dependências entre os serviços conclui-se que, para ter o *Spark* funcional, quando integrado com a *Stack* da *Hortonworks*, é necessário também ter disponível no sistema os serviços *Hive*, *HDFS*, *MapReduce2* e *YARN,* sejam estes de facto usados ou não pelo *Spark*. Este facto faz com que o *Spark* tenha associada uma carga por omissão maior para o sistema do que as implementações com *Storm*.

5.2.3.4 *Storm* – Arquitetura *Lambda*

O desenvolvimento da aplicação para o *Storm* seguindo a arquitetura *Lambda* incluiu todos os componentes necessários na mesma topologia (Figura 59), tanto o fluxo da *Speed Layer* como o fluxo da *Batch Layer*, uma vez que não há outro serviço neste sistema com capacidade de processamento. Uma dificuldade na implementação desta aplicação, foi que o *Storm* não foi desenhado para processamento em *Batch* pelo que foi necessário usar uma estratégia para executar periodicamente o cálculo das métricas nos *Bolts*.

SensorDataLambdaTopology.scal ✕ | SensorKafkaSpeedSpout.scala | SensorDataMetricsSpeedBolt.scala | Sensor

```
29        val builder = new TopologyBuilder
30        builder.setSpout("sensordata_batchspout", new SensorKafkaBatchSpout, 1)
31        builder.setBolt ("sensordata_batchstore", new SensorDataStoreBatchBolt, 1)
32                                                  .shuffleGrouping( "sensordata_batchspout" )
33        builder.setBolt ("sensordata_batchmetrics", new SensorDataMetricsBatchBolt, 1)
34                                                  .shuffleGrouping( "sensordata_batchstore" )
35
36        builder.setSpout("sensordata_speedspout", new SensorKafkaSpeedSpout, 1)
37        builder.setBolt ("sensordata_speedmetrics", new SensorDataMetricsSpeedBolt, 1)
38                                                  .shuffleGrouping("sensordata_speedspout")
39        builder.setBolt ("sensordata_speedstore", new SensorDataStoreSpeedBolt, 1)
40                                                  .shuffleGrouping( "sensordata_speedmetrics")
```

Figura 59 – Topologia do *Storm* para arquitetura *Lambda*

Como a arquitetura pressupõe um consumo paralelo para ambas as *Layers* foram desenvolvidos dois *Spouts*, um para a *Batch Layer* (*SensorKafkaBatchSpout*) e outro para a *Speed Layer* (*SensorKafkaSpeedSpout*), sendo que estes usam grupos de consumo diferentes para evitar partilha dos índices do tópico do *Kafka*. Ambos os *Spouts* consomem pequenos blocos de registos do *Kafka* de cada vez, à semelhança do que foi implementado para a arquitetura *Kappa* com *Storm* e do que acontece no *Spark*. Após cada consumo de eventos, é feito o tratamento dos mesmos, que são convertidos para tuplos e emitidos para os *Bolts* com a implementação da arquitetura correspondente, o *SensorDataMetricsSpeedBolt* e o *SensorDataStoreBatchBolt*.

Na *Speed Layer*, os tuplos recebidos do *Spout* no *SensorDataMetricsSpeedBolt* são usados para cálculo de métricas e, sempre que chega um valor fora da janela de análise (1 minuto), as métricas do minuto anterior são emitidas para o próximo *Bolt* deste fluxo, o *SensorDataStoreSpeedBolt*, e os objetos usados para calcular as métricas são reiniciados a 0. O *SensorDataStoreSpeedBolt*, sempre que recebe um evento, formata os valores recebidos para que sejam compatíveis com os tipos das colunas da tabela do *Cassandra* para onde os valores vão ser escritos e, por fim, usa o *Driver do Cassandra* da *DataStax* para escrever as métricas.

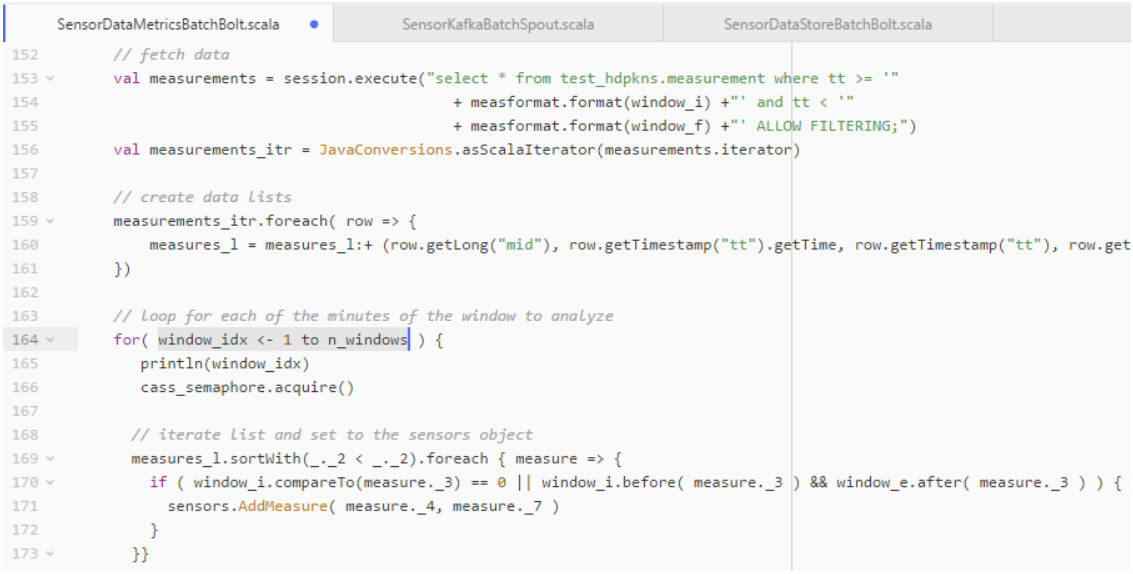


Figura 60 - Cálculo das métricas na *Batch Layer* do *Storm*, no *SensorDataMetricsBatchBolt*

Na *Batch Layer*, o fluxo ocorre ao contrário do que acontece na *Speed Layer*, ou seja, os tuplos recebidos do *Spout* no *SensorDataStoreBatchBolt* são devidamente formatados e escritos para o *Cassandra*. O *SensorDataStoreBatchBolt* implementa também a estratégia para executar o cálculo das métricas periodicamente, visto que não é possível agendar tarefas do *Storm* em intervalos de tempo fixos. Para escrever os dados dos eventos para o *Cassandra*, estes são acumulados em conjuntos de 1000 registos antes de serem escritos. Isto é feito, como já referido, por questões de desempenho. O *Cassandra Driver* não é eficiente se tiver de resolver um *INSERT* por cada evento. Este intervalo de tempo para acumular 1000 eventos é usado como intervalo entre os eventos emitidos para o *SensorDataMetricsBatchBolt* para iniciar o processo de cálculo de métricas. Este processo não é otimizado e introduz mais carga no sistema do que seria desejável. No entanto, este é o ponto existente no fluxo mais adequado para emitir uma mensagem entre os *Bolts* que inicie o processo de cálculo de métricas em *Batch*. Este tuplo é composto pela data e hora do evento mais recente escrito para o *Cassandra*. O *SensorDataMetricsBatchBolt*, sempre que recebe este tuplo, cria uma janela de 1 minuto para cada um dos 10 minutos anteriores. Estas 10 janelas são usadas como referência para filtrar a estrutura de dados retornada no pedido ao *Cassandra* e para agregar os valores com as mesmas estruturas de sensores usadas na *Speed Layer*. Posteriormente, os valores agregados por cada minuto são escritos para a tabela *batch_metrics* do *Cassandra*.

Todas as aplicações descritas anteriormente, depois de devidamente testadas e estáveis, foram usadas para:

- adquirir valores de desempenho, com base no tempo que demorou a consumir os dados do *Kafka* e no tempo que demorou a disponibilizar cada evento para consulta;
- adquirir valores de consumo de recursos de cada um dos sistemas com base na monitorização feita durante cada simulação de dados.

As métricas obtidas através destas aplicações foram usadas para comparar quão eficiente é cada uma das tecnologias de processamento, usadas segundo os pressupostos de cada uma das arquiteturas abordadas para o caso de uso em questão.

Durante o processo surgiram diversos imprevistos e desafios, como por exemplo, a situação do *PySpark* não permitir obter um desempenho comparável com o *Storm*. Essa situação solucionou-se, reescrevendo a aplicação em *Scala*. Outra situação, que poderia ser interessante de avaliar do ponto de vista tanto do desempenho, como do consumo dos recursos, era o modo de execução do *Spark*. Este pode variar entre local, *yarn-client* e *yarn-cluster*. Nos modos *yarn-client* e *yarn-cluster*, o *Spark* usa o *YARN* para distribuir o processamento pelo *cluster* e, dentro da mesma máquina, pelos *containers* que criar, cada um com uma quantidade mínima pré-definida de recursos. Contrariamente, o modo local apenas usa os recursos locais para executar as aplicações. No ambiente disponível só há dois vCPU, o que é muito limitado para todos os serviços em funcionamento e em particular para interagir com o *YARN*. As aplicações lançadas em qualquer um dos modos *YARN* ficavam permanentemente suspensas à espera de ter mais recursos disponíveis. Por esse facto, não foi possível testar o *Spark* integrado com o *YARN*. Uma outra experiência que seria interessante realizar para verificar se o problema de desempenho do *PySpark* era apenas devido a recursos, seria distribuir o processamento. Esta avaliação seria também uma adição interessante aos resultados, no entanto não houve possibilidade de testar de forma distribuída as aplicações desenvolvidas.

5.3 Aquisição de Métricas e Interpretação de Resultados

Para calcular os resultados finais de desempenho e de consumo de recursos de cada sistema foram usados os produtos das simulações de dados. Para cada sistema - *Spark-Kappa*, *Spark-Lambda*, *Storm-Kappa* e *Storm-Lambda* - foram feitas simulações em que, para cada uma, foram simulados 500000 eventos para induzir carga durante cerca de 3 minutos em fluxo contínuo. De cada uma destas simulações resultou um ficheiro com uma tabela de consumo de recursos e uma tabela com os dados dos eventos registados no *Cassandra*. É também possível extrair uma tabela com as métricas calculadas durante cada simulação por cada fluxo do processo. Por ter dois fluxos, no caso das arquiteturas *Lambda* há duas tabelas de métricas por arquitetura, uma calculada pelo fluxo da *Speed Layer* e outra calculada pelo fluxo da *Batch Laye*r.

As tabelas extraídas de cada um dos sistemas disponibiliza a informação necessária para conseguir avaliar quais os consumos médios e o tempo médio de resposta, de forma a conseguir compará-los posteriormente.

De forma a melhor avaliar os sistemas e a facilitar a análise, foi desenvolvido um conjunto de scripts em *Python*. O objetivo destes scripts foi ajudar a transformar os resultados exportados de cada sistema num conjunto de indicadores que os resumisse, nomeadamente, os valores médios, seja de consumo de recursos seja de desempenho, mas também os desvios padrão, valores mínimos e valores máximos.

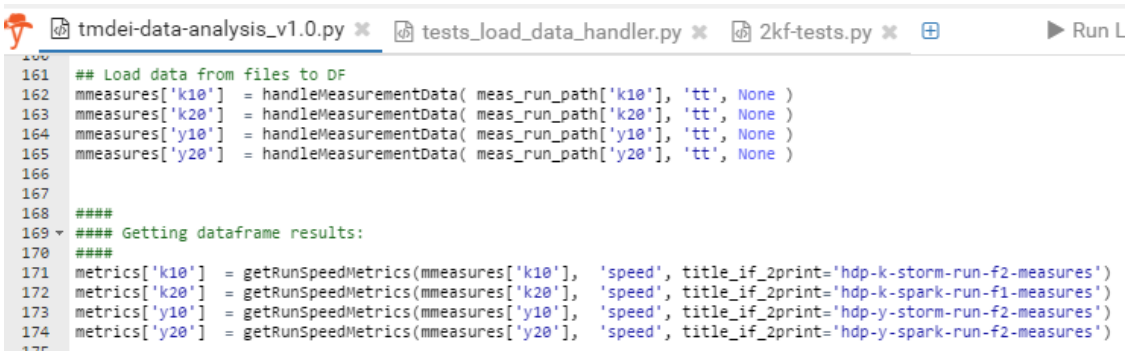


Figura 61 – Excerto do script de apoio tmdei-data-analysis_1.0.py

Para o caso dos resultados de desempenho, o script está a carregar os dados exportados das tabelas do *Cassandra*. Como se pode ver na Figura 61, a função *handleMeasurementData* recebe como argumento o caminho para o ficheiro com os dados exportados do *Cassandra*, formatado como CSV, a coluna pela qual deve ordenar o *dataframe* gerado, caso não esteja já ordenado, e, por fim, um parâmetro para indicar se a primeira linha do ficheiro tem os nomes das colunas ou não. Esta função retorna um *dataframe* com os dados do CSV, com as datas formatadas para o tipo de data do *Python* e duas colunas relevantes de valores derivados – o *consumption_delta*, o *speed_delta* (Figura 62).

Desta forma, a estrutura final do *dataframe* calculado tem as seguintes colunas:

- mid - identificador incremental do evento gerado no simulador de dados;
- tt – timestamp (date e hora) da criação do evento;
- sensor_id – identificador do sensor simulado;
- in_tt – timestamp do momento em que o evento foi consumido no *Kafka* e, portanto, deu entrada no sistema;
- measure – valor da medição do sensor. Para facilitar a verificação dos resultados dos testes, foi usado sempre um valor constante;
- out_tt – *timestamp* do momento em que o evento oriundo do simulador ficou disponível no *Cassandra*;
- consumption_delta – intervalo de tempo decorrido entre o tt e in_tt;
- speed_delta – intervalo de tempo decorrido entre tt e out_tt.

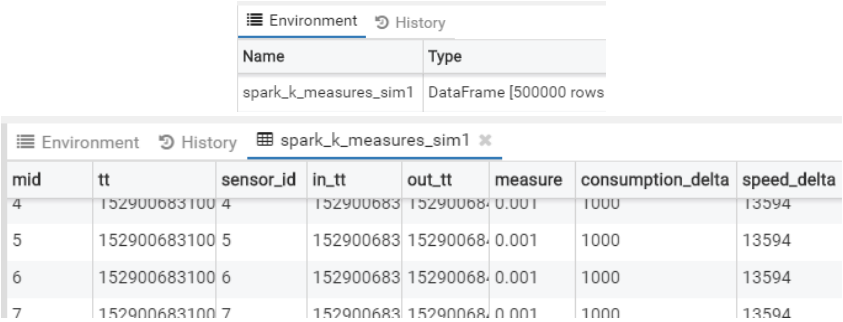


Figura 62 – Exemplo de um *dataframe* gerado dos dados do *Cassandra* de uma simulação

Este *dataframe*, disponível por cada simulação realizada, permite calcular os vários parâmetros globais a partir dos dados que estão a ser analisados. A função *getRunSpeedMetrics* (Figura 63) explora este *dataframe* e retorna vários parâmetros. Entre os mais relevantes temos a média, o desvio padrão, o mínimo e o máximo, tanto do tempo de consumo, como do tempo de resposta do sistema. Obtendo estes parâmetros para os diferentes sistemas, é possível comparar o seu desempenho face ao mesmo conjunto de dados de entrada.

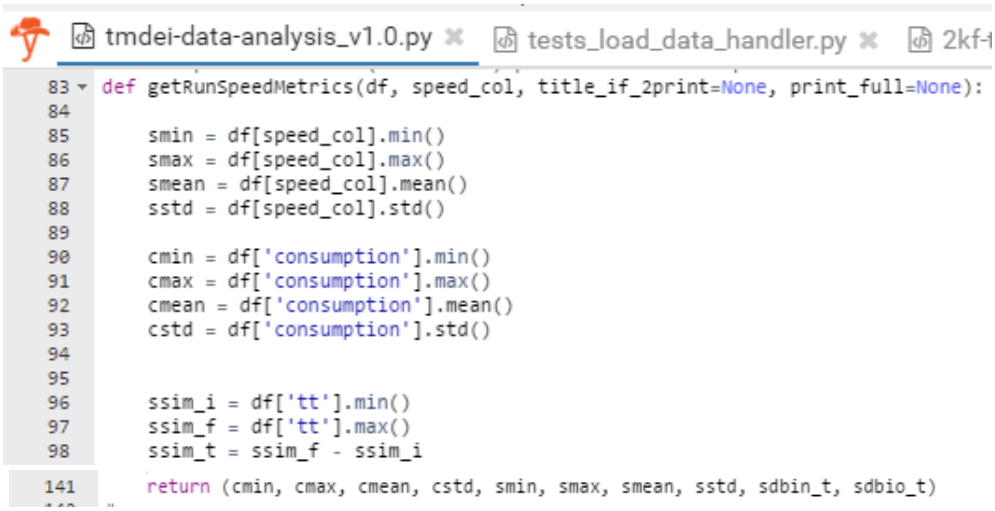


Figura 63 – Excerto da função *getRunSpeedMetrics*

O outro script usado para suportar esta análise, dedicado ao tratamento dos dados de consumo de recursos dos sistemas (Figura 64), é o *tmdei-consumptions-data-analysis_v1.0.py*. À semelhança do script anterior, este carrega os ficheiros da monitorização dos recursos e usa-os para calcular as métricas mais relevantes para a comparação entre os sistemas desenvolvidos.

A função *handleLoadData*, análoga à *handleMeasurementData*, recebe os caminhos dos ficheiros e o nome do parâmetro que deve considerar para a ordenação e carrega a estrutura para o *dataframe* (Figura 65). Neste caso não há necessidade de colunas derivadas, pelo que a composição do *dataframe* é a seguinte:

- time – registo temporal associado à monitorização dos recursos;
- cpu – valor do consumo de CPU, em percentagem;
- mem – valor do consumo de memória, em percentagem;
- diskio_r – valor do consumo de disco com operações de leitura, em MB/s;
- diskio_w – valor do consumo de disco com operações de escrita, em MB/s.

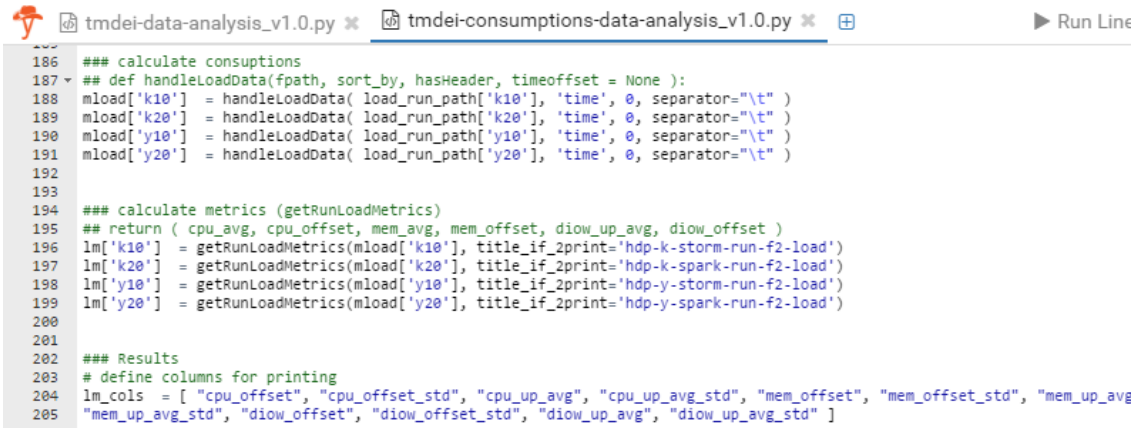
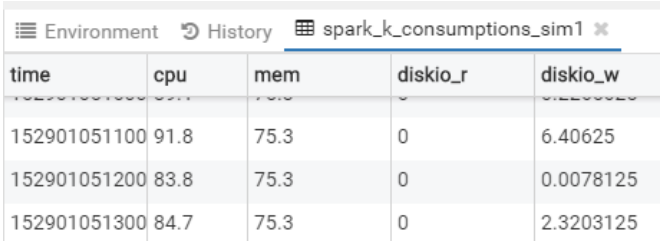


Figura 64 - Excerto do script de apoio *tmdei-consumptions-data-analysis_v1.0.py*

Este *dataframe* permite calcular os valores de consumo de recursos necessários à comparação entre os sistemas. Essas métricas são calculadas na função *getRunLoadMetrics*, que usa o *dataframe* para calcular a média, o desvio padrão, o mínimo e o máximo de cada um dos consumos.

Uma exceção é o caso da leitura do disco que nos sistemas que seguiram a arquitetura *Kappa* é sempre nula e nos sistemas de arquitetura *Lambda* tem periodicamente um valor maior que 0. Nesta situação, os valores são tão próximos de zero que podem ser desconsiderados desta análise. Seria necessário um caso de uso exigente do ponto de vista de leitura para conseguir fazer uma avaliação devida com este parâmetro.



time	cpu	mem	diskio_r	diskio_w
152901051100	91.8	75.3	0	6.40625
152901051200	83.8	75.3	0	0.0078125
152901051300	84.7	75.3	0	2.3203125

Figura 65 - Exemplo de um dataframe obtido pela monitorização de recursos durante uma simulação

A simulação de dados começa 10 segundos após começar a monitorização dos recursos e continua durante mais algum tempo depois da simulação de dados terminar, no máximo 10 minutos. A análise dos recursos tem de ser considerada em dois momentos diferentes: enquanto a monitorização ocorre com carga no sistema derivada da simulação (RUN) e enquanto a monitorização ocorre sem carga no sistema derivada da simulação (IDLE). Em IDLE, todos os serviços e aplicações estão em funcionamento, a consumir os recursos mínimos para o seu funcionamento.

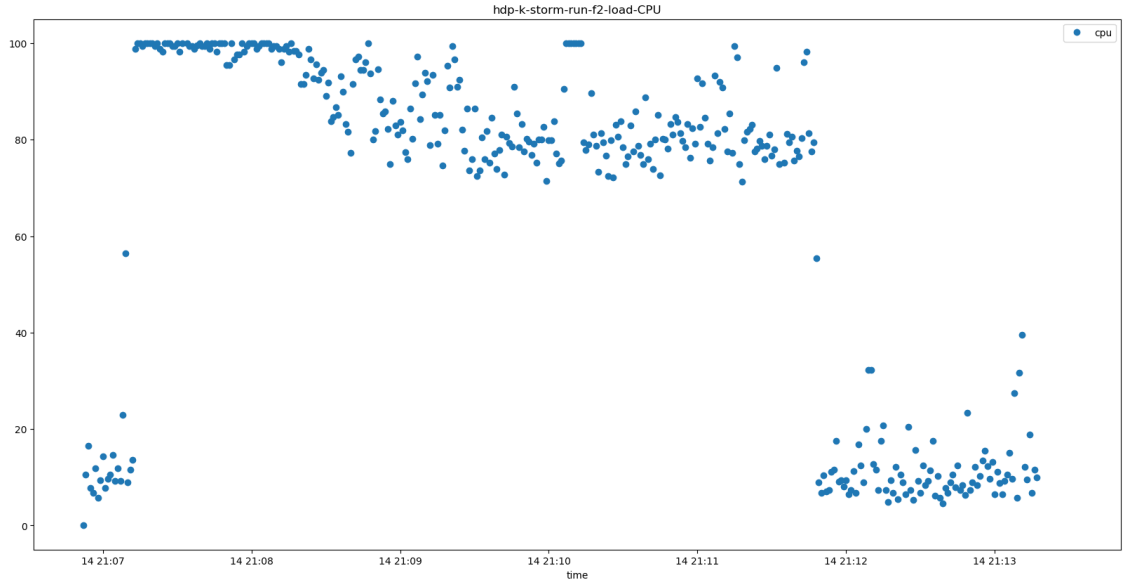


Figura 66 – Exemplo de série temporal relativa a consumo de CPU durante uma simulação com *Storm-Kappa*

Para a distinção entre o que são os dados em RUN e IDLE, foram analisadas algumas series temporais, como é o exemplo da representada na Figura 66. Nesta, é bastante percetível que, nos primeiros segundos da monitorização, a simulação ainda não está a ocorrer (CPU IDLE), portanto as aplicações dos casos de uso ainda não estão a consumir recursos. Após o início, o consumo de CPU sobe bastante (CPU RUN) e no fim do processo volta a decrescer (CPU IDLE).

tmdei-data-analysis_v1.0.py | tmdei-consumptions-data-analysis

```
95      lm.add( df['cpu'].max() )
96      lm.add( df['cpu'][df.cpu < df['cpu'].mean()].mean() )
97      lm.add( df['cpu'][df.cpu < df['cpu'].mean()].std() )
98      lm.add( df['cpu'][df.cpu > df['cpu'].mean()].mean() )
99      lm.add( df['cpu'][df.cpu > df['cpu'].mean()].std() )
100
101     # Memory metrics
102     base_mem = df['mem'][1:22].mean()
103     lm.add( df['mem'].mean() )
104     lm.add( df['mem'].min() )
105     lm.add( df['mem'].max() )
106     lm.add( df['mem'][df.mem < base_mem].mean() )
107     lm.add( df['mem'][df.mem < base_mem].std() )
108     lm.add( df['mem'][df.mem > base_mem].mean() )
109     lm.add( df['mem'][df.mem > base_mem].std() )
110
111     # DiskIO(Write) metrics
112     base_diskio_w = 0.1
113     lm.add( df['diskio_w'].mean() )
114     lm.add( df['diskio_w'].min() )
115     lm.add( df['diskio_w'].max() )
116     lm.add( df['diskio_w'][df.diskio_w < base_diskio_w].mean() )
117     lm.add( df['diskio_w'][df.diskio_w < base_diskio_w].std() )
118     lm.add( df['diskio_w'][df.diskio_w > base_diskio_w].mean() )
119     lm.add( df['diskio_w'][df.diskio_w > base_diskio_w].std() )
```

Figura 67 - Excerto da função *getRunLoadMetrics*

Para o consumo de memória, foi feita também uma análise que mostrou que, após o consumo muito estável nos primeiros segundos, enquanto a carga não começa (MEM IDLE), a memória consumida sobe e varia de acordo com um padrão diretamente relacionado com a aplicação

(MEM RUN). Por exemplo, aumenta com a acumulação de valores nas aplicações com *Storm* até ao momento em que o conjunto é escrito para a base de dados. No entanto, a memória nunca baixou para o estado original, pelo menos no período de tempo em que se monitorizou estes processos. No caso das aplicações com *Spark*, a memória consumida variou mais com a acumulação das métricas, visto que, para estas, tiveram de ser usados os acumuladores do *Spark,* e também nunca baixou para os valores originais. Por esse facto, no caso da análise da memória optou-se por manter a média dos primeiros valores, anteriores ao início da carga induzida pela simulação, como referência para distinguir entre RUN e IDLE.

O consumo em disco tendeu sempre para 0 e acabou por ser ignorado na análise, pois não havia amostras com taxa de leitura significativa. A maioria dos valores da taxa de escrita mantinham-se sempre próximos de 0 (DISK_W IDLE) e os momentos de escrita identificavam-se pelos "picos" mais ou menos periódicos (DISK_W RUN, Figura 68). Este comportamento pode explicar-se pelo método de funcionamento do *Cassandra,* que mantém os dados em memória e acumula-os antes de os persistir. Neste caso, como o próprio sistema mostrou valores residuais de taxa de uso do disco aleatoriamente, para os excluir, considerou-se que os registos com taxa de escrita abaixo de 0,1 MB/s corresponderiam a momentos em IDLE no sistema.

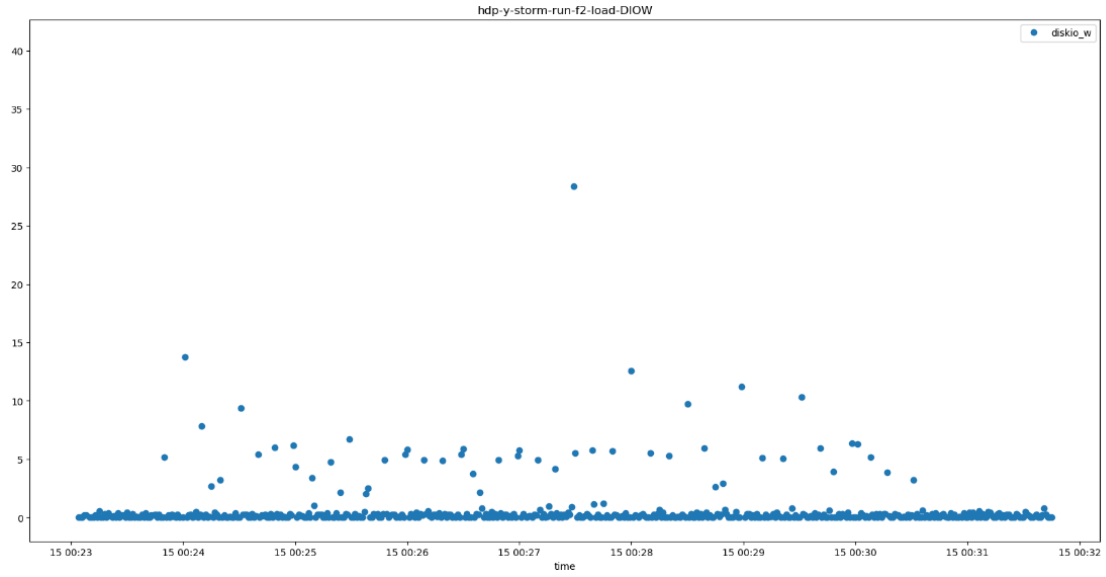


Figura 68 - Exemplo de série temporal relativa à taxa de escrita para disco durante uma simulação com *Storm-Lambda*

Estas análises foram feitas para os diferentes parâmetros, para identificar pelo padrão do consumo do recurso qual a melhor forma de distinguir o momento em que a simulação está, de facto, a induzir carga.

5.4 Discussão de Resultados e Comparação dos Sistemas e Tecnologias

Para a comparação, primeiro foram feitas simulações em todos os sistemas para a aquisição das métricas. As métricas foram analisadas como descrito em 5.3, e dos resultados desta análise foi construída a Tabela 8.

Tabela 8 – Métricas de desempenho e consumo de recursos do sistema

Métrica	hdp-κ-storm	hdp-κ-spark	hdp-λ-storm	hdp-λ-spark
Tempo médio de consumo (s)	**0,03895**	**0,74302**	**0,29345**	**1,11961**
σ Tempo médio de consumo (s)	0,22474	0,87261	1,02289	1,60032
Tempo médio de resposta (s)	**1,46791**	**1,09053**	**1,73652**	**1,13909**
σ Tempo médio de resposta (s)	2,16276	0,94350	3,59794	1,60961
CPU IDLE (%)	14,58917	44,10082	59,61579	47,02500
σ CPU IDLE (%)	13,01313	27,49129	16,69860	33,91052
CPU RUN (%)	**87,24928**	**92,02370**	**97,64859**	**98,83945**
σ CPU RUN (%)	9,46158	8,50198	2,58140	2,05453
MEM IDLE (%)	70,01500	74,20000	73,32500	87,74565
σ MEM IDLE (%)	16,47982	32,72079	0,12583	13,22794
MEM RUN (%)	**75,68704**	**89,43282**	**77,94952**	**95,1356**
σ MEM RUN (%)	0,47462	0,44728	1,21862	2,85233
DISK_W IDLE (MB/s)	0,00828	0,02269	0,01016	0,01404
σ DISK_W IDLE (MB/s)	0,01528	0,02738	0,01806	0,02075
DISK_W RUN (MB/s)	**2,00245**	**4,49250**	**1,54680**	**8,68395**
σ DISK_W RUN (MB/s)	3,58566	21,80947	3,70449	21,49463

Na Tabela 8 são apresentadas as médias e os respetivos desvios padrão (σ) para cada uma das métricas calculadas para cada um dos sistemas implementados. Para as métricas relacionadas com os consumos de recursos são apresentados os consumos com o sistema em carga (RUN) e sem carga no sistema (IDLE). As métricas apresentadas são as seguintes:

- CPU RUN – Consumo médio de CPU do sistema, em percentagem, com serviços em funcionamento e com simulação de dados em funcionamento;
- CPU IDLE – Consumo médio de CPU do sistema, em percentagem, com serviços em funcionamento, mas com simulação de dados parada;
- MEM RUN – Consumo médio de memória do sistema, em percentagem, com serviços em funcionamento e com simulação de dados em funcionamento;
- MEM IDLE – Consumo médio de memória do sistema, em percentagem, com serviços em funcionamento, mas com simulação de dados parada;
- DISK_W RUN – Média da taxa de escrita para o disco do sistema, em MB/s, com serviços em funcionamento e com simulação de dados em funcionamento;

- DISK_W IDLE – Média da taxa de escrita para o disco do sistema, em MB/s, com serviços em funcionamento, mas com simulação de dados parada.

Para todas as métricas listadas, é também considerado o respetivo desvio padrão (σ) que vai correlacionar-se com a estabilidade do sistema.

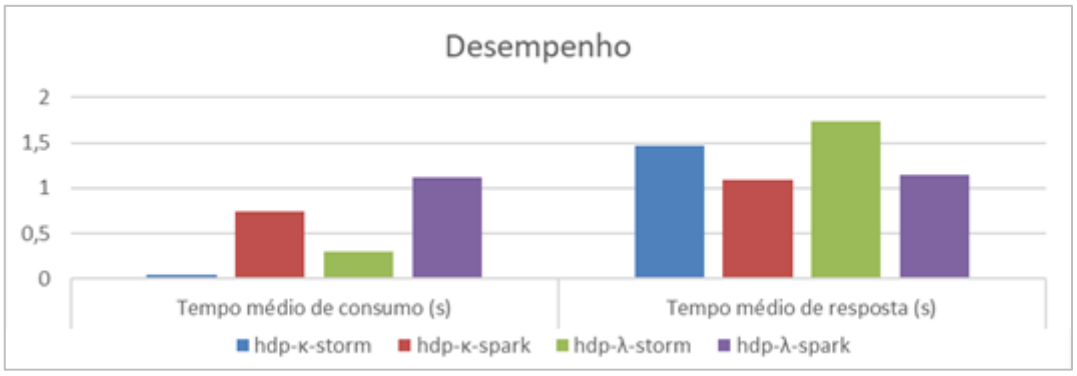


Figura 69 - Tempos de consumo e de resposta dos sistemas

Para a avaliação de desempenho dos sistemas, comparou-se as métricas de desempenho entre cada um. A métrica que de facto indica qual o sistema com melhor desempenho geral é o tempo de resposta. Na tabela de métricas manteve-se o tempo de consumo pois a comparação entre os dois tempos permite retirar algumas conclusões em relação ao desempenho de escrita para a base de dados do *Storm* e do *Spark*.

Como pode ser confirmado no gráfico da Figura 69, o sistema que apresenta um tempo de consumo de dados do *Kafka* claramente melhor do que os restantes é a implementação da arquitetura *Kappa* com *Storm*, sendo seguido em velocidade pelo sistema da arquitetura *Lambda* com *Storm*. Por isto, pode afirmar-se que o *Storm* consegue extrair dados do *Kafka* mais rápido que o *Spark*. Ao contrário do que seria de esperar depois de analisado o tempo de consumo, verifica-se que os sistemas com Spark, por sua vez, apresentam um tempo de resposta melhor do que os sistemas com *Storm*. Quanto ao tempo de resposta, o sistema que apresentou melhor desempenho foi a arquitetura *Kappa* implementada com *Spark*. Um dos fatores mais relevantes para a diferença no desempenho do *Storm* e do *Spark* aparenta ser a *interface* para escrita para o *Cassandra*. No caso, a do *Spark* é mais eficaz.

As métricas de desvio padrão relativas aos tempos são, neste caso, um indicador da estabilidade do sistema. Um sistema que apresente um desvio padrão menor nas métricas de desempenho revelou-se mais regular durante a simulação. Por exemplo, o consumo do CPU de um sistema mais estável vai variar menos durante o processamento. Comparando os desvios padrão das métricas de tempo de resposta, verifica-se que os sistemas com *Spark* são mais estáveis do que os sistemas com *Storm*.

Do ponto de vista dos recursos consumidos pelos sistemas, como se pode confirmar nas figuras Figura 70, Figura 71 e Figura 72, a implementação da arquitetura *Kappa* com *Storm* é a combinação menos exigente para o sistema em quase todas as situações analisadas, consumindo claramente menos recursos do que os restantes. É também claro que os sistemas

com *Spark* consomem sempre mais recursos do que os sistemas com *Storm*, sejam IDLE ou não. Uma outra situação clara é que os sistemas com *Spark* apresentam maiores taxas de escrita para disco. Este fator está relacionado com a maior taxa de escrita para disco conseguida pelas arquiteturas de *Spark*, mas também pela agregação feita dos eventos no *Storm* antes destes serem escritos para o *Cassandra*, para que a escrita seja feita como *microbatch*, à semelhança do que ocorre com o *Spark*.

A Tabela 8 e os gráficos derivados permitem ter uma ideia de qual é o sistema que garante melhor velocidade e qual é o sistema que garante menor consumo de recursos. No entanto, não é claro perceber qual é o sistema que apresenta a melhor relação entre o consumo de recursos e o desempenho.

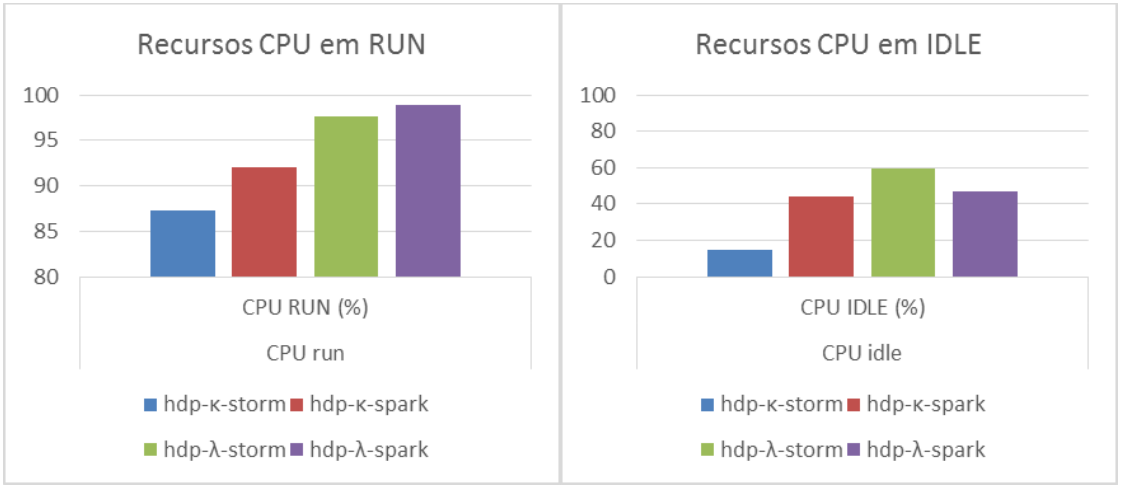


Figura 70 - Métricas de CPU com (RUN) e sem (IDLE) carga no sistema

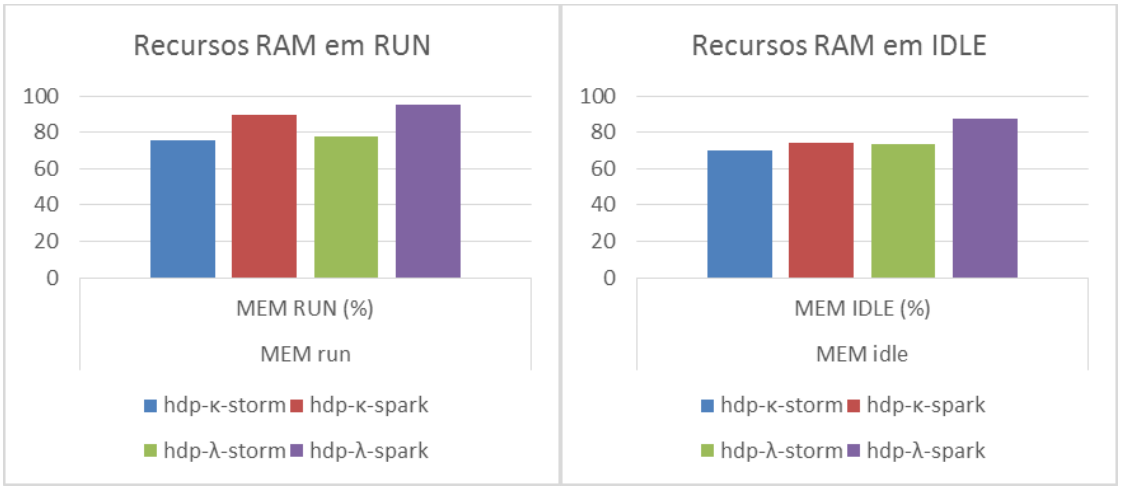


Figura 71 - Métricas de memória com (RUN) e sem (IDLE) carga no sistema

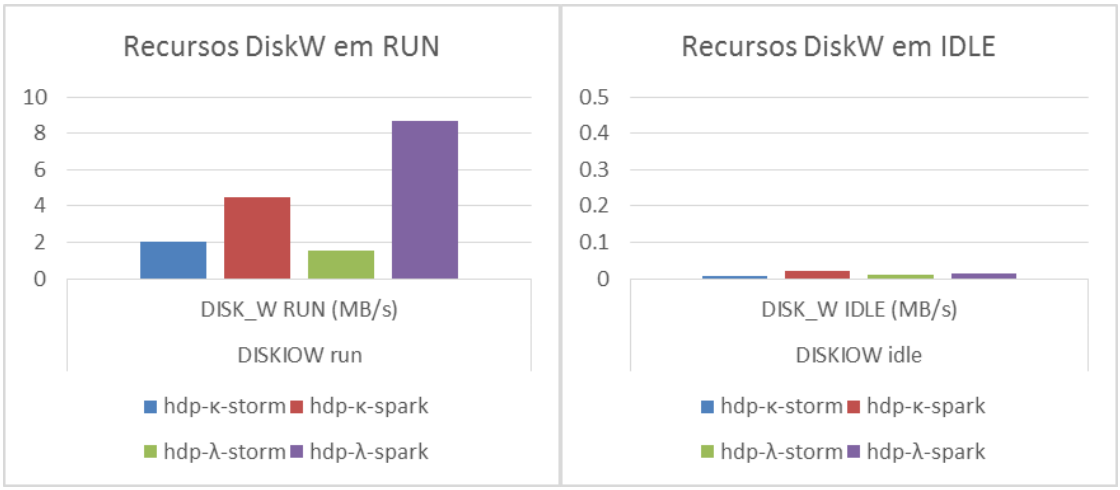


Figura 72 - Métricas de consumo disco em operações de escrita

Para ajudar a determinar qual o sistema com o melhor desempenho em função do consumo de recursos, foi feita uma análise adicional que consistiu em, para cada métrica de cada sistema, atribuir uma pontuação (Tabela 9). A pontuação atribuída é de 1 a 4, sendo que o 4 é atribuído ao resultado mais desejável de cada métrica e 1 ao menos desejável. A título de exemplo:

- O sistema com tempo de resposta mais baixo recebe um 4, ao passo que o sistema com pior (maior) tempo de resposta recebe um 1;
- O sistema com maior consumo de CPU em RUN recebe um 1 e o sistema com menor consumo de CPU em RUN recebe um 4.

Tabela 9 - Métricas de desempenho e consumo de recursos do sistema

Métrica	hdp-κ-storm	hdp-κ-spark	hdp-λ-storm	hdp-λ-spark
Tempo médio de consumo (s)	4	2	3	1
σ Tempo médio de consumo (s)	4	3	2	1
Tempo médio de resposta (s)	**2**	**4**	**1**	**3**
σ Tempo médio de resposta (s)	2	4	1	3
CPU IDLE (%)	4	3	1	2
σ CPU IDLE (%)	4	2	3	1
CPU RUN (%)	**4**	**3**	**2**	**1**
σ CPU RUN (%)	1	2	3	4
MEM IDLE (%)	4	2	3	1
σ MEM IDLE (%)	2	1	4	3
MEM RUN (%)	**4**	**2**	**3**	**1**
σ MEM RUN (%)	3	4	2	1
DISK_W IDLE (MB/s)	4	1	3	2
σ DISK_W IDLE (MB/s)	4	1	3	2
DISK_W RUN (MB/s)	**3**	**2**	**4**	**1**
σ DISK_W RUN (MB/s)	4	1	3	2

Depois de atribuída a pontuação, os valores da Tabela 9 foram agregados de uma forma diferente, somando os pontos para cada característica a avaliar, como pode ser visto na Tabela 10. A título de exemplo, para o consumo de CPU em RUN todos os sistemas têm 5 pontos, porque corresponde à soma dos pontos da média com a soma dos pontos do desvio padrão da mesma média. Ao considerar também os desvios padrão está a considerar-se, não só o consumo ou desempenho médio, mas também a estabilidade do sistema. Um sistema que seja mais estável vai ser mais previsível e mais facilmente escalado, seja para outros casos de uso ou outros volumes de entrada, logo é uma característica que é importante considerar na avaliação.

Tabela 10 – Avaliação da relação desempenho/recursos (RUN)

Pontuação	hdp-κ-storm	hdp-κ-spark	hdp-λ-storm	hdp-λ-spark
Desempenho (Tempo de resposta)	4	8	2	6
Total Recursos (RUN)	19	14	17	10
CPU (RUN)	5	5	5	5
MEM (RUN)	7	6	5	2
DISKIOW (RUN)	7	3	7	3
Total Recursos (IDLE)	22	10	17	11
CPU (IDLE)	8	5	4	3
MEM (IDLE)	6	3	7	4
DISKIOW (IDLE)	8	2	6	4
Relação Desempenho/Recursos (RUN)	64,583	79,167	47,917	58,3
Estabilidade (menor variação de recursos)	8	7	8	7

Para o cálculo da relação desempenho/recursos apresentada na Tabela 10, foram usados os valores de desempenho e de recursos (RUN) normalizados. Por exemplo, para calcular o valor de relação desempenho/recursos do sistema hdp-κ-spark:

$$\text{Relação}_{\text{hdp-κ-spark}} = \frac{Desempenho}{Max_{Desempenho}} * 100 + \frac{Total\ Recursos\ (RUN)}{Max_{Total\ Recursos\ (RUN)}} * 100 = \frac{8}{8} * 100 + \frac{14}{24} * 100 = 79{,}167$$

considerando que:

- $Max_{Desempenho}$ – corresponde ao número máximo de pontos possível entre pontuação do tempo médio de resposta e o respetivo desvio padrão;
- $Max_{TotaRecursos(Run)}$ – corresponde ao número máximo de pontos possível entre pontuação no campo Total Recursos (RUN).

Para calcular o valor do $Max_{Desempenho}$ é somada a pontuação máxima do tempo médio de resposta, 4 pontos, e do respetivo tempo de resposta, 4 pontos. Para calcular o valor do $Max_{TotaRecursos(Run)}$ é somada a pontuação máxima, 4 pontos, dos seis parâmetros relacionados com os recursos durante a simulação (CPU RUN, σ CPU RUN, MEM RUN, σ MEM RUN, DISKIOW RUN e σ DISKIOW RUN). Para calcular a estabilidade (menor variação de recursos) fez-se o somatório dos desvios padrão dos recursos durante a simulação (RUN).

Desta forma, a Tabela 10 reflete quais as características em que cada sistema é melhor e o que esperar, de forma relativa, de cada um deles. Fica assim claro que o sistema com melhor relação desempenho/recursos em RUN é o que seguiu a arquitetura *Kappa* implementado com *Spark*, o qual é também o que apresenta o melhor desempenho. Quanto ao consumo de recursos, os

sistemas implementados com *Storm* são os que apresentam os melhores resultados, pois consomem menos recursos. Em particular, o sistema que seguiu a arquitetura *Kappa* implementado com *Storm* é o que tem os melhores resultados gerais neste aspeto.

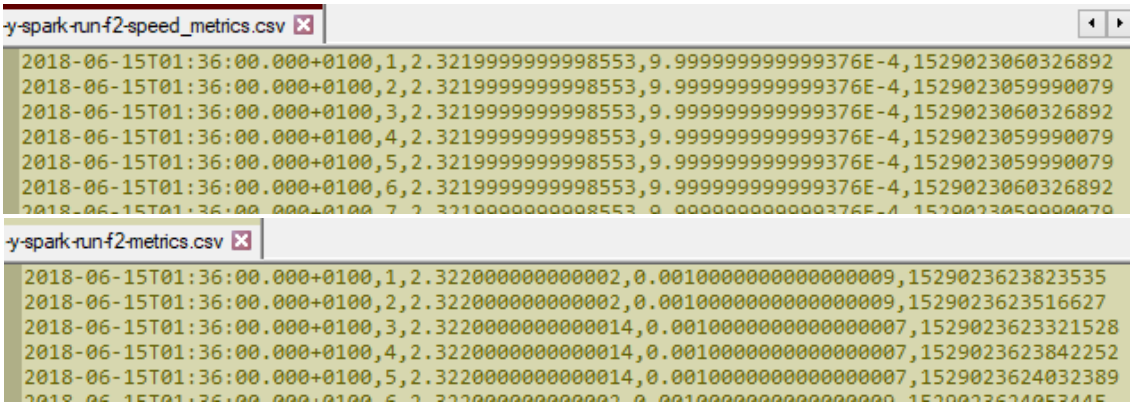


Figura 73 – Exemplo de métricas calculadas entre *Speed Layer* (em cima) e *Batch Layer* (em baixo)

Os resultados apontam para a utilização de uma arquitetura *Kappa* em detrimento da *Lambda*. No entanto, apesar desses resultados e como já foi referido anteriormente, a arquitetura *Lambda* considera características funcionais que a *Kappa* não considera. Uma delas, e talvez a mais relevante neste caso, é a utilização da *Batch Layer* para fazer, de facto, os cálculos de métricas de forma mais precisa. Para validar se a precisão é um problema, foram analisadas as métricas calculadas durante as simulações.

A Figura 73 apresenta um pequeno segmento das métricas calculadas na *Batch Layer* e um excerto das métricas calculadas na *Speed Layer* para usar como referência nesta avaliação. As métricas foram exportadas do *Cassandra* no formato CSV, onde o terceiro campo é o somatório de todos os valores dos eventos para o minuto em questão e o quarto campo é a média dos valores do mesmo período. Como se pode confirmar pelo excerto, a diferença entre as métricas calculadas não faria diferença para qualquer aplicação que não exigisse agregações sem erro até à 14ª casa decimal, momento a partir do qual os arredondamentos fariam diferença. Como esta exigência na resolução não é comum, pode considerar-se que não seria uma vantagem do sistema da arquitetura *Lambda* em relação à *Kappa*. Assim, esta avaliação indica ainda que o pressuposto da arquitetura *Lambda* de manter os dois processos e não apenas um, é uma desvantagem, neste caso, porque não é um esforço que resulte em melhorias significativas nos resultados obtidos.

Tendo estes resultados em conta, o sistema proposto como solução teórica para o caso de uso de processamento de dados de sensores não seria uma arquitetura *Lambda*, mas sim uma arquitetura *Kappa* com *Spark*.

Quanto às hipóteses colocadas em 5.1.3, é feita uma análise para aceitação ou rejeição das hipóteses com base nos resultados obtidos e discutidos acima.

> *H1*. Os sistemas desenvolvidos com *Spark* deverão ser, para uma alta taxa de dados, mais eficazes que os restantes sistemas testados.

Para a primeira hipótese, *H1*, considerando que um sistema é mais eficiente se disponibilizar dados mais rapidamente com o mínimo de recursos possível, então a hipótese é aceite. Confirmando na Tabela 10, os sistemas desenvolvidos com *Spark* apresentam um desempenho superior ao desempenho dos sistemas desenvolvidos com *Storm* para a mesma arquitetura e, no caso, é a arquitetura *Kappa* desenvolvida com *Spark* que apresenta melhor relação de desempenho face ao consumo de recursos.

> *H2*. Uma arquitetura baseada em *streaming* (*Kappa*) resulta num sistema mais responsivo do que se o sistema fizer processamento em *batch* (*Lambda*).

Para a segunda hipótese, *H2*, considera-se que um sistema mais responsivo é um sistema capaz de interagir com um utilizador mais rápido e, portanto, é tão mais responsivo quanto mais rápido a disponibilizar dados para pedido dos utilizadores. Se se comparar sistemas baseados na arquitetura *Kappa* com sistemas com a arquitetura *Lambda*, mas com a mesma *Stack* tecnológica, então pode considerar-se que a hipótese é aceite. De acordo com os resultados obtidos, tal já não é verdade de forma transversal visto que a arquitetura *Kappa* implementada com *Storm* demora mais tempo a disponibilizar dados do que a arquitetura *Lambda* implementada com *Spark*.

> *H3*. Sistemas com tecnologias menos dependentes de operações em disco são mais responsivos do que sistemas com mais operações em disco.

A última hipótese, *H3*, não pode ser aceite de acordo com os resultados obtidos. Os sistemas mais responsivos são os implementados com *Spark*. No entanto, estes são os que têm mais dependências de operações com disco. Isto porque o *Spark* depende de vários outros serviços (*Hive*, *HDFS*, *MR2*, *YARN*), enquanto que o *Storm* depende apenas do *Zookeeper*. Por outro lado, os ambientes desenvolvidos com *Spark* são também os mais dependentes de operações em disco, de acordo com os resultados obtidos, apesar de terem melhor desempenho.

Com os resultados obtidos é finalmente possível fazer uma tabela comparativa entre as tecnologias *Spark* e *Storm*. A Tabela 11 considera, além das propriedades analisadas na experiência realizada, um conjunto de características para reforçar a distinção entre as duas tecnologias de processamento distribuído usadas.

Tabela 11 – Comparação de características entre *Spark Streaming* e *Storm*

Tec. Stream. / Característica	Spark Streaming	Storm
Latência	Latência corresponde, neste caso, ao tempo médio de consumo. O *Spark*, apresenta mais latência no processo de aquisição de dados em *streaming* e de processamento	Tende a ter menos latência no processamento do que o *Spark*.
Desempenho Geral	Tem melhor desempenho geral no processamento de dados e disponibilização das métricas em base de dados.	Apresenta pior desempenho no processamento de métricas do que o *Spark*, em particular no desempenho de escrita para a base de dados.
Recursos	Mais exigente a nível de recursos no processamento. Em IDLE tende também a ser maior o consumo de recursos do que *Storm* devido a ter mais dependências de outros serviços.	Menos exigente a nível de recursos do que o *Spark*.
Tolerância a Falha	Sim. As instâncias de processamento do *Spark Streaming* são criadas pelo *YARN* e, portanto, o próprio trata de gerir a execução e o estado de cada instância. Em caso de falha crítica não há mecanismos próprio para reiniciar o processo.	Sim. Se um processo falhar o *Supervisor* (componente que monitoria os processos do *Storm*) vai reiniciar o processo. O estado do processo é recuperado pelo *Zookeeper*.
Principais Linguagens	*Java, Scala, Python, R*. *R* e *Python* não recomendados para fins de produção ou sempre que houver critérios de desempenho apertados	*Java*, *Scala*
Facilidade de Gestão	Moderado. É necessário gerir através da monitorização dos processos em execução no *YARN* ou de ferramentas como *Ganglia*. Interface de gestão permite apenas visualizar, mas não gerir, os DAG das aplicações de *Spark* executados e os logs.	Simples. O serviço *Nimbus* disponibiliza um portal que pode ser usado para gerir topologias em execução. Apenas não permite submeter topologias novas.
Facilidade de Desenvolvimento	Simples. Apenas é necessário instanciar os devidos objetos dos contextos e usar as funcionalidades disponibilizadas. Nomes dos métodos intuitivos e de funcionamento semelhante a outras funcionalidades conhecidas.	Moderado. É necessário cumprir com várias interfaces e compreender o ciclo de vida de cada componente a implementar.
Aplicabilidade	Ideal para processamento em *batch*. Pode ser usado para processamento em *streaming* com muito bom desempenho.	Ideal para processamento de evento a evento, em particular se do processamento em tempo real resultarem poucas métricas para persistir.
Persistência de Dados	Sim, por RDD (*Resilient Distributed Dataset*)	Sim, com recurso ao *MapState* disponibilizado pelo *Trident*[23]

[23] Trident – Extensão do Apache Storm desenvolvida pelo Twitter

6 Conclusões e Trabalho Futuro

O Big Data é um conceito da moda e uma consequência da crescente disponibilidade de tecnologia e do uso que se pretende fazer com os dados gerados em todas as atividades. Pode ser encarado como um fenómeno "técnico-social", definido por características como a *Variedade* de formatos, o *Volume* de informação que os sistemas têm de tratar e a que *Velocidade* esses dados devem ficar disponíveis desde que são gerados até que estão transformados em ações para potenciar o valor. De facto, quanto a características, além dos *3 V's* há muitas outras idealizadas por diversas entidades que trabalham diariamente com estas tecnologias, numa tentativa de ilustrar quais as dificuldades sentidas com os seus sistemas e como elas são refletidas no conceito de Big Data.

A nível de soluções, o ASF apresenta uma quantidade considerável de tecnologias desenvolvidas por várias organizações e depois cedidas ao grupo. Apresenta um vasto leque de tecnologias, bem definidas e com características conhecidas, que chama a atenção de qualquer organização que esteja a dar os seus primeiros passos na área do Big Data, especialmente pelo suporte da comunidade e pelas ferramentas em si serem distribuídas como *software open source*. Isto é também um catalisador para que as empresas tendam a usar estas ferramentas, talvez até não porque têm quantidades de dados gerados que o exijam, mas porque pretendem explorar as soluções na área e tirar partido de uma filosofia diferente para resolver problemas do quotidiano, diversificando o tipo de serviços que disponibilizam e a qualidade dos serviços existentes até então. A dimensão do ecossistema *Hadoop* e a oportunidade de negócio gerada pela dificuldade das organizações se adaptarem aos ambientes de Big Data, levou ao aparecimento de empresas especializadas. Estas criaram *Stacks* de tecnologias para suprir as necessidades de ambiente analíticos de propósito genérico e distribuem-nas enquanto soluções "chave na mão", garantindo a integração dos serviços, a capacidade de gestão, a automação de processos e, ainda, algo fundamental para as empresas que investem em ambientes de Big Data, o suporte oficial tanto na resolução de problemas como de validação e certificação dos ambientes *Hadoop* em uso nas empresas. As *Stacks* disponibilizadas desta forma são, por princípio, de uso genérico, e podem ser adaptadas para a resolução de uma grande variedade de problemas do dia a dia onde as tecnologias tradicionais apresentariam diversas limitações.

A análise teórica das tecnologias neste documento, procurou focar-se em tecnologias que poderiam ser usadas para as responsabilidades mais comuns de sistemas de Big Data, passando pelas arquiteturas consideradas como base, pelo conceito de *Stack*, e por quais as tecnologias de uso mais comum para cada um dos pontos principais em sistema de Big Data: a integração de dados, o armazenamento distribuído e o processamento distribuído de dados. Para cada um destes pontos foram apontadas algumas tecnologias, mas seria oportuno e interessante ter abordado mais algumas, visto que estão sempre a surgir novas tecnologias, mais específicas, otimizadas para resolver novos tipos de problemas. Por exemplo, o *OpenTSDB* poderia ter sido abordado em contexto do caso de uso seguido, visto ser uma tecnologia dedicada a armazenar e permitir operar sobre séries temporais. Talvez o desempenho do *Storm* fosse superior ao do *Spark* nos testes se fosse usado o *OpenTSDB* para a *Serving Layer*. Contudo esta tecnologia não foi abordada porque não é uma das mais comuns nem deve usada com propósitos genéricos.

À luz do caso de uso, foi analisada uma *Stack* e uma arquitetura proposta para obter um sistema eficiente no processamento de dados de sensores em tempo real. Para estudar possibilidades teoricamente adequadas para resolver o problema e compará-las foram implementados os quatro sistemas: processamento em *Storm* seguindo uma arquitetura *Kappa*, processamento em *Storm* seguindo uma arquitetura *Lambda*, processamento em *Spark* seguindo uma arquitetura *Kappa* e processamento em *Spark* seguindo uma arquitetura *Lambda*. Em cada um destes sistemas foi desenvolvida uma aplicação usando os serviços disponíveis, de forma a conseguir perceber qual o ambiente com as melhores características. Durante o processo, surgiram diversas limitações, sendo exemplos o facto do desenvolvimento de topologias para *Storm* em linguagens que não o *Scala* ou *Java* não parecer robusto e não ter boa documentação. O que existe para o efeito, por exemplo, em *Python* são adaptações feitas por utilizadores da comunidade, mas nada suportado oficialmente. O *Spark,* por sua vez, suporta bem *Python* e *R* além do *Scala* e *Java* mas descobriu-se, fruto dos testes de desempenho, que desenvolver aplicações para altas taxas de dados ou com requisitos apertados de velocidade não é viável com as *interfaces* disponibilizadas de *Python* (*PySpark*). Para despistar se o problema com o *PySpark* era exclusivamente um problema de recursos, teria sido interessante testar a aplicação de forma distribuída. Tal não foi possível, precisamente por falta de disponibilidade de recursos físicos. O *hardware* disponível permanentemente foi apenas um PC com recursos para, no máximo, virtualizar máquinas com 12GB de memória e 2 vCPU. Estes recursos são teoricamente muito limitativos para um ecossistema *Hadoop*, o que se comprovou em alguns detalhes como, por exemplo, a necessidade de executar as aplicações de *Spark* sem recurso ao *YARN*. As aplicações de *Spark*, quando executadas a partir do *YARN*, não executavam, ficando pendentes à espera de ter recursos disponíveis. Este e outros problemas foram identificados durante o desenvolvimento do estudo devido aos recursos serem limitados. No caso do problema do *PySpark* a alternativa foi desenvolver a mesma aplicação em *Scala*, o que resolveu o problema de desempenho do *Spark*. Desta forma, conclui-se que para desenvolvimento em *Storm* e *Spark*, em particular se for para fins de produção onde há requisitos de velocidade apertados, é necessário fazer o desenvolvimento em *Scala* ou *Java*.

Os resultados obtidos nos testes feitos permitiram avaliar as tecnologias de processamento *Spark* e *Storm*, em cada uma das arquiteturas de referência mais adequadas para

processamento de dados em *streaming*. Durante o desenvolvimento das aplicações de *streaming*, verificou-se também que o *Spark* é muito mais simples para desenvolver aplicações, tem um suporte para processamento de dados consideravelmente melhor do que o *Storm*, tem melhor documentação e tem potencial para ter melhor desempenho com abordagens mais robustas para persistência de dados do que o *Storm*. Por outro lado, o *Storm* tem muito menos dependência de outros serviços, para processamento evento a evento, é mais eficiente a coletar e trabalhar os dados e tem melhor interface web, onde é possível gerir as topologias em execução. Quanto à resolução para o caso de uso, verificou-se que o *Spark*, apesar de ser mais exigente para o sistema, apresenta um melhor desempenho geral no sistema ao conseguir disponibilizar mais rapidamente dados na *Serving Layer*. No entanto, em ambientes com recursos reduzidos, o *Storm* traz uma vantagem, uma vez que se revelou muito leve quando implementado com a arquitetura *Kappa*. A diferença de consumo de recursos do *Storm* com arquitetura *Kappa* e *Lambda* é significativa, e apoia o facto de que o *Storm* foi desenhado para fazer processamento exclusivamente em *streaming* e não em *batch*. Curiosamente, apesar do *Spark* ser mais exigente para os recursos do sistema, também acabou por revelar maior estabilidade.

Uma possível continuação do trabalho desenvolvido passaria pela repetição dos mesmos testes em ambientes com mais recursos disponíveis e com o processamento distribuído entre nós de um *cluster* com pelo menos três nós para processamento. Estas condições permitiriam fazer de facto, uma boa avaliação da capacidade de processamento das tecnologias usadas de forma distribuída. Seria também interessante abordar novas tecnologias para a *Serving Layer* como o *OpenTSDB*, referido acima, o *Druid* e novas metodologias de processamento com uso, por exemplo, do *NiFi* para integração e processamento distribuído dos dados. Uma avaliação do ponto de vista de segurança de informação aplicáveis ao mesmo caso de uso, seria também um novo ponto de vista, certamente interessante, a abordar das tecnologias usadas. Com novas tecnologias em uso, seria interessante avaliar também a capacidade de aplicar métodos avançados de análise, mais elaborados, para distinguir o desempenho das diferentes tecnologias em função da complexidade do processamento a fazer.

As tecnologias do ecossistema de *Hadoop* têm evoluído e vão continuar a evoluir enquanto se traduzirem numa vantagem para as entidades que adotam estes novos métodos e filosofias para abordar os seus problemas técnicos. Certamente este mesmo estudo repetido futuramente traria diferentes tecnologias para análise e resultados substancialmente diferentes.

Referências

Aaron. (2013). Cassandra, Hive, and Hadoop: How We Picked Our Analytics Stack. Retrieved from http://blog.markedup.com/2013/02/cassandra-hive-and-hadoop-how-we-picked-our-analytics-stack/

Apache Software Foundation. (2014). Welcome to Apache ™ Hadoop ® ! Retrieved from https://hadoop.apache.org/

ASF, A. (2015). Azkaban. Retrieved from https://azkaban.github.io/

Bakshi, K. (2012). Considerations for big data: Architecture and approach. *IEEE Aerospace Conference Proceedings*, 1–7. https://doi.org/10.1109/AERO.2012.6187357

Ballou, K. (2014). Apache Storm vs. Apache Spark. Retrieved from http://zdatainc.com/2014/09/apache-storm-apache-spark/

Barlow, M. (2013). *Real-Time Big Data Analytics: Emerging Architecture*. *Zhurnal Eksperimental'noi i Teoreticheskoi Fiziki*. O'Reilly Media, Inc. https://doi.org/10.1007/s13398-014-0173-7.2

Bertran, P. F. (2014). Lambda Architecture: A state-of-the-art. Retrieved from http://www.datasalt.com/2014/01/lambda-architecture-a-state-of-the-art/

Bhandarkar, M. (2012). Introduction to Apache Hadoop. *Cloudera*.

Bloomberg, J. (2013). The Big Data Long Tail. *DevXtra Editors' Blog*. Retrieved from http://www.devx.com/blog/the-big-data-long-tail.html

Borthakur, D. (2013). HDFS Architecture Guide. Retrieved from https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction

Brown, B., Chui, M., & Manyika, J. (2011). Are you ready for the era of " big data "? *McKinsey Global Institute*, *October*(October), 1–12. https://doi.org/00475394

Brown, B., Manyika, J., Chui, M., Bughin, J., Dobbs, R., Roxburgh, C., & Byers, A. H. (2011). Big data: The next frontier for innovation, competition, and productivity. *McKinsey Global Institute*, (June), 156. https://doi.org/10.1080/01443610903114527

Casado, R., & Younas, M. (2015). Emerging trends and technologies in big data processing. *Concurrency and Computation-Practice & Experience*, *27*(SEPTEMBER 2014), 2078–2091. https://doi.org/10.1002/cpe.3398

Cascading, A. (2015). Cascading. Retrieved from http://www.cascading.org/documentation/

Chen, H., Chiang, R. H. L., & Storey, V. C. (2012). Business Intelligence and Analytics: From Big Data To Big Impact. *Mis Quarterly*, *36*(4), 1165–1188. https://doi.org/10.1145/2463676.2463712

Clemm, J. (2015). A Brief History of Scaling LinkedIn. Retrieved from https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin

Cloudera. (2017). Cloudera. Retrieved from https://www.cloudera.com/products/open-source/apache-hadoop/apache-spark.html

Danesh, S. (2014). *Big data - From Hype to Reality*. Örebro University.

DataStax. (2016). *Apache Cassandra™ 2.0*. DataStax, Inc. Retrieved from https://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureIntro_c.html

DataStrax. (2015). Getting Started with Apache Spark and Cassandra. Retrieved from http://www.planetcassandra.org/getting-started-with-apache-spark-and-cassandra/

De Mauro, A., Greco, M., Grimaldi, M., De Mauro, A., Greco, M., & Grimaldi, M. (2015). What is big data? A consensual definition and a review of key research topics. *Proceedings of the 4th International Conference on Integrated Information*, *1644*, 97–104. https://doi.org/10.1063/1.4907823

Demchenko, Y., De Laat, C., & Membrey, P. (2014). Defining architecture components of the Big Data Ecosystem. *2014 International Conference on Collaboration Technologies and Systems, CTS 2014*, 104–112. https://doi.org/10.1109/CTS.2014.6867550

Edd Dumbill. (2012). What is big data? In *O ' Reilly Media* (pp. 1–9). Retrieved from https://beta.oreilly.com/ideas/what-is-big-data

Eini, O. (2010). Column (Family) Databases. Retrieved from https://ayende.com/blog/4500/that-no-sql-thing-column-family-databases

Farris, A., Guerra, P., & Sen, R. (2014). Benchmarking the Apache Accumulo Distributed Key – Value Store.

Fernandez, R. C., Pietzuch, P., Kreps, J., Narkhede, N., Rao, J., Koshy, J., … Wang, G. (2015). Liquid: Unifying Nearline and Offline Big Data Integration. *Cidr*.

Flume, A. (2012). https://flume.apache.org/FlumeUserGuide.html. Retrieved from https://flume.apache.org/FlumeUserGuide.html

Forgeat, J. (2015). Data processing architectures – Lambda and Kappa. Retrieved from https://www.ericsson.com/research-blog/data-knowledge/data-processing-architectures-lambda-and-kappa/

Garg, N. (2013). *Apache Kafka*. Packt Publishing Ltd.

George, L. (2010). *HBase: The Definitive Guide* (2nd Editio). O'Reilly Media, Inc. Retrieved from papers2://publication/uuid/786A0D0D-0622-487D-B4CB-031924A92B71

Goodhope, K., Koshy, J., & Kreps, J. (2012). Building LinkedIn's Real-time Activity Data Pipeline. *IEEE Data Eng*, 1–13.

Grover, M., Malaska, T., Seidman, J., & Shapira, G. (2014). *Hadoop Application Architectures*.

Harter, T., Borthakur, D., Dong, S., Aiyer, A., Tang, L., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2014). Analysis of HDFS Under HBase: A Facebook Messages Case Study. *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, 199–212. Retrieved from http://blogs.usenix.org/conference/fast14/technical-sessions/presentation/harter

Hausenblas, M., & Bijnens, N. (2015). Lambda Architecture. Retrieved from http://lambda-architecture.net/

HBase, A. (2016). Apache HBase ™ Reference Guide. Retrieved from https://hbase.apache.org/book.html#number.of.cfs

Hoff, T. (2013). The Architecture Twitter Uses to Deal with 150M Active Users, 300K QPS, a 22 MB/S Firehose, and Send Tweets in Under 5 Seconds. Retrieved from http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html?utm_source=feedly

Hoffman, S. (2013). Apache Flume: Distributed Log Collection for Hadoop. In Intergovernmental Panel on Climate Change (Ed.), *Climate Change 2013 - The Physical Science Basis*. Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9781107415324.004

Hortonworks. (2014). A modern data architecture with Apache Hadoop: the journey to a data lake. *The Journey to a Data Lake*, (March), 18.

Huynh, X. (2014). Storm vs. Spark Streaming: Side-by-side comparison. Retrieved from http://xinhstechblog.blogspot.pt/2014/06/storm-vs-spark-streaming-side-by-side.html

Jain, A. (2013). Using Cassandra for Real-time Analytics. Retrieved from http://blog.markedup.com/2013/04/cassandra-real-time-analytics-part-2/

Jiang, L. (2015). Flume or Kafka for Real-Time Event Processing. Retrieved from https://www.linkedin.com/pulse/flume-kafka-real-time-event-processing-lan-jiang

Jones, M. T. (2012). Process real-time big data with Twitter Storm- An introduction to streaming big data. *Ibm*, 1–9.

Kala Karun, A., & Chitharanjan, K. (2013). A review on hadoop - HDFS infrastructure extensions. *2013 IEEE Conference on Information and Communication Technologies, ICT 2013*, (Ict), 132–137. https://doi.org/10.1109/CICT.2013.6558077

Kestelyn, J. (2015). Exactly-once Spark Streaming from Apache Kafka. Retrieved from http://blog.cloudera.com/blog/2015/03/exactly-once-spark-streaming-from-apache-kafka/

Khan, M. A. U. D., Uddin, M. F., & Gupta, N. (2014). Seven V's of Big Data understanding Big Data to extract value. *Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education - "Engineering Education: Industry Involvement and Interdisciplinary Trends", ASEE Zone 1 2014*.

https://doi.org/10.1109/ASEEZone1.2014.6820689

Kreps, J. (2009). Project Voldemort: Scaling Simple Storage at LinkedIn. Retrieved from http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin/

Kreps, J. (2014). Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). Retrieved from https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: a Distributed Messaging System for Log Processing. *NetDB*.

Kulkarni, A., & Khandewal, M. (2014). Survey on Hadoop and Introduction to YARN. *Ijetae.Com*, *4*(5), 82–87. Retrieved from http://www.ijetae.com/files/Volume4Issue5/IJETAE_0514_15.pdf

Kumar, S. (2013). Twitter Data Analytics, 89. https://doi.org/10.1007/978-1-4614-9372-3

Leibiusky, J., Eisbruch, G., & Simonassi, D. (2012). *Getting started with Storm*.

Leverenz, L. (2015). Hive Design. Retrieved from https://cwiki.apache.org/confluence/display/Hive/Design

Liu, J., Bing, L., & Meina, S. (2010). The optimization of HDFS based on small files. *Proceedings - 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology, IC-BNMT2010*, 912–915. https://doi.org/10.1109/ICBNMT.2010.5705223

Lorentz, A. (2013). Big Data, Fast Data, Smart Data. Retrieved from http://www.wired.com/insights/2013/04/big-data-fast-data-smart-data/

Madappa, S. (2013). Announcing EVCache: Distributed in-memory datastore for Cloud. Retrieved from http://techblog.netflix.com/2013/02/announcing-evcache-distributed-in.html

Maheshwari, R. (2015). 3 V's or 7 V's - What's the Value of Big Data- - Rajiv Maheshwari - LinkedIn. Retrieved from https://www.linkedin.com/pulse/3-vs-7-whats-value-big-data-rajiv-maheshwari

Maier, M. (2013). Towards a Big Data Reference Architecture, (October), 1–144.

MapR. (2016). MapR Technologies. Retrieved from https://www.mapr.com/products/mapr-converged-data-platform

Marr, B. (2014). Big Data: The 5 Vs Everyone Must Know. Retrieved from https://www.linkedin.com/pulse/20140306073407-64875646-big-data-the-5-vs-everyone-must-know

MaRS. (2012). *MaRS: Fundamentals of Entrepreneurial Management*. MaRS Discovery District.

Marz, N., & Warren, J. (2015). *Big Data*. Retrieved from http://nathanmarz.com/about/

McNulty, E. (2014). Understanding Big Data: The Seven V's. Retrieved from http://dataconomy.com/seven-vs-big-data/

Menon, A. (2012). Big Data @ Facebook. *MBDS '12: Proceedings of the 2012 Workshop on Management of Big Data Systems*, 31. https://doi.org/10.1145/2378356.2378364

Mishne, G., Dalton, J., Li, Z., Sharma, A., & Lin, J. (2012). Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture. Retrieved from http://arxiv.org/abs/1210.7350

Murdopo, A., & Dowling, J. (2013). Next Generation Hadoop : High Availability for YARN.

Naga, P. N., Kuan, C.-Y., & Wu, J. (2014). How LinkedIn Democratizes Big Data Visualization. Retrieved from http://conferences.oreilly.com/strata/stratany2014/public/schedule/detail/36459

Namiot, D. (2015). On Big Data Stream Processing. *International Journal of Open Information Technologies*, *3*(8), 48–51.

Nemschoff, M. (2014). A Quick Guide to Structured and Unstructured Data. Retrieved from http://www.smartdatacollective.com/michelenemschoff/206391/quick-guide-structured-and-unstructured-data

Nicolas, B. (2014). *Investigating the Lambda Architecture*.

Oliver, A. C. (2014). Storm or Spark: Choose your real-time weapon. Retrieved from http://www.infoworld.com/article/2854894/application-development/spark-and-storm-for-real-time-computation.html

Oracle. (2013). Information Management and Big Data: A Reference Architecture. Redwood Shores: Oracle Corporation. Retrieved from http://www.oracle.com/technetwork/topics/entarch/articles/info-mgmt-big-data-ref-arch-1902853.pdf

Penchalaiah, C., Murali, G., & Suresh Babu, A. (2014). Effective Sentiment Analysis on Twitter Data using : Apache Flume and Hive, *1*(8), 101–105.

Pokluda, A., & Sun, W. (2013). Benchmarking Failover Characteristics of Large-Scale Data Storage Applications: Cassandra and Voldemort. *Alexanderpokluda.Ca*. Retrieved from http://alexanderpokluda.ca/coursework/cs848/CS848 Project Report - Alexander Pokluda and Wei Sun.pdf

Prabhakar, A. (2011). Apache Flume – Architecture of Flume NG. Retrieved from http://blog.cloudera.com/blog/2011/12/apache-flume-architecture-of-flume-ng-2/

Prakash, C. (2016). Apache Storm : Architecture Overview. Retrieved from https://www.linkedin.com/pulse/apache-storm-architecture-overview-chandan-prakash

Prokopp, C. (2014). Lambda Architecture: Achieving Velocity and Volume with Big Data. Retrieved from http://www.semantikoz.com/blog/lambda-architecture-velocity-volume-big-data-hadoop-storm/

Roman, J. (2015). The Hadoop Ecosystem Table. Retrieved from https://hadoopecosystemtable.github.io/

Ryza, S., Laserson, U., Owen, S., & Wills, J. (2015). *Advanced Analytics with Spark*.

Saaty, T. L. (2008). The Analytic Hierarchy and Analytic Network Measurement Processes: Applications to Decisions under Risk. *European Journal of Pure and Applied Mathematics*, *1*(1), 122–196. https://doi.org/10.1007/0-387-23081-5_9

Schumacher, R. (2015). A Brief Introduction to Apache Cassandra. Retrieved from https://academy.datastax.com/demos/brief-introduction-apache-cassandra

Shi, G. (2002). Data Integration using Agent based Mediator-Wrapper Architecture. *Tutotial Report for Agent Based Software Engineering (SENG 609.22)*.

Shreedharan, H. (2014). *Using Flume - Stream Data into HDFS and HBase*. *O'Reilly Media*.

Solovey, E. (2015). Handling five billion sessions a day – in real time - Twitter Blogs. Retrieved from https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time

Spark, A. (2017). Apache Spark - Lightning-fast unified analytics engine. Retrieved from https://spark.apache.org/

Sumbaly, R., Kreps, J., & Shah, S. (2013). The "Big Data" Ecosystem at LinkedIn. *… Conference on Management of Data*, 1–10. https://doi.org/10.1145/2463676.2463707

Thein, K. M. M. (2014). Apache Kafka : Next Generation Distributed Messaging System, *3*(47), 9478–9483.

Thusoo, A., Sarma, J. Sen, Jain, N., Shao, Z., Chakka, P., Zhang, N., … Murthy, R. (2010). Hive - A petabyte scale data warehouse using hadoop. *Proceedings - International Conference on Data Engineering*, 996–1005. https://doi.org/10.1109/ICDE.2010.5447738

Thusoo, A., Shao, Z., & Anthony, S. (2010). Data warehousing and analytics infrastructure at facebook. *… on Management of Data*, 1013. https://doi.org/10.1145/1807167.1807278

Ting, K., & Cecho, J. J. (2013). *Apache Sqoop Cookbook*. Retrieved from http://books.google.de/books?id=3qKAW063BhoC

Traverso, M. (2013). Presto : Interacting with petabytes of data at Facebook. Retrieved from https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920

Tsukayama, H. (2013). Twitter turns 7: Users send over 400 million tweets per day. Retrieved from https://www.washingtonpost.com/business/technology/twitter-turns-7-users-send-over-400-million-tweets-per-day/2013/03/21/2925ef60-9222-11e2-bdea-e32ad90da239_story.html

Vagata, P., & Wilfong, K. (2014). Scaling the Facebook data warehouse to 300 PB. Retrieved from https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/

Venner, J. (2009). *Pro Hadoop: Build scalable, distributed applications in the cloud* (Vol. 41). https://doi.org/10.1145/1539024.1508904

Vora, M. N. (2011). Hadoop-HBase for large-scale data. *Proceedings of 2011 International Conference on Computer Science and Network Technology, ICCSNT 2011*, *1*, 601–605. https://doi.org/10.1109/ICCSNT.2011.6182030

Vorhies, W. (2014). How Many "V's" in Big Data? The Characteristics that Define Big Data. Retrieved from http://www.datasciencecentral.com/profiles/blogs/how-many-v-s-in-big-data-the-characteristics-that-define-big-data#_edn5

White, T. (2009). *Hadoop : The Definitive Guide* (First Edit). O'Reilly Media, Inc.