Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística

João Victor Azevedo Esteves

Deduplicação de dados não-estruturados de processos *streaming* em tempo real

Rio de Janeiro
2021

João Victor Azevedo Esteves

Deduplicação de dados não-estruturados de processos streaming em tempo real

Dissertação apresentada como requisito parcial para obtenção do título de Mestre ao Programa de Pós-Graduação em Ciências Computacionais da Universidade do Estado do Rio de Janeiro.

Orientadora: Profª. Dra. Rosa Maria Esteves Moreira da Costa

Coorientadora: Profª. Dra. Ana Carolina Brito de Almeida

Rio de Janeiro

2021

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC-A

E79 Esteves, João Victor Azevedo.
Deduplicação de dados não-estruturados de processos streaming em tempo real / João Victor Azevedo Esteves. – 2021.
86f.: il.

Orientadora: Rosa Maria Esteves Moreira da Costa.
Coorientadora: Ana Carolina Brito de Almeida.
Dissertação (Mestrado em Ciências Computacionais) – Universidade do Estado do Rio de Janeiro, Instituto de Matemática e Estatística.

1. Processamento de dados - Teses. 2. Processamento de arquivo (Computação) - Teses. 3. Mineração de dados (Computação) – Teses. I. Costa, Rosa Maria Esteves Moreira. II. Almeida, Ana Carolina Brito. III. Universidade do Estado do Rio de Janeiro. Instituto de Matemática e Estatística. III. Título.

CDU 004.62

Márcia França Ribeiro – CRB7- 3664 - Bibliotecária responsável pela elaboração da ficha catalográfica

_____ _____

Assinatura Data

João Victor Azevedo Esteves

Deduplicação de dados não-estruturados de processos *streaming* em tempo real

Dissertação apresentada como requisito parcial para obtenção do título de Mestre ao Programa de Pós-Graduação em Ciências Computacionais da Universidade do Estado do Rio de Janeiro.

Aprovada em 11 de junho de 2021.

Banca Examinadora:

_____
Prof.ª Dra. Rosa Maria Esteves Moreira da Costa
Instituto de Matemática e Estatística - UERJ

_____
Prof.ª Dra. Ana Carolina Brito de Almeida
Instituto de Matemática e Estatística - UERJ

_____
Prof. Dr. Sergio Lifschitz
Pontifícia Universidade Católica do Rio de Janeiro

_____
Prof.ª Dra. Fernanda Araujo Baião
Pontifícia Universidade Católica do Rio de Janeiro

_____
Prof. Dr. Alexandre Sztajnberg
Instituto de Matemática e Estatística – UERJ

Rio de Janeiro

2021

AGRADECIMENTOS

Primeiramente a Deus que me presenteou com a tenacidade e capacidade para vencer cada dia desta jornada.

Aos meus pais Antônio Esteves e Leonice Ferreira por toda educação e suporte, por sempre me incentivarem lutar e acreditar que é possível. À minha noiva Ester Monteiro por estar ao meu lado apoiando-me em todos os momentos.

Às minhas orientadoras Ana Almeida e Rosa Costa que me auxiliaram em cada etapa. Por cada sugestão dada ao decorrer deste projeto, por cada motivação, sempre me jogando para cima e me fazendo acreditar que é possível. Por sempre estarem pacientes com seu aluno, me passando o conhecimento de forma clara e específica. Afinal, posso dizer que não ganhei só orientadoras, mas duas mentoras em minha vida acadêmica.

Aos professores Sergio Lifschitz, Fernanda Baião e Alexandre Sztajnberg, por terem aceitado participar da avaliação deste trabalho.

À Universidade do Estado do Rio de Janeiro, por toda a estrutura fornecida para a pesquisa, e especialmente a todo o corpo docente, que me auxiliou durante as aulas das disciplinas, que com certeza contribuíram muito para meu desenvolvimento acadêmico.

E a todos que de alguma forma contribuíram para a chegada deste grande dia.

Alguns homens veem as coisas como são, e se perguntam “Por quê?”. Eu sonho com as coisas que nunca foram e me pergunto “Por que não?”.

Back to Methuselah– George Bernard Shaw

RESUMO

ESTEVES, João Victor Azevedo. *Deduplicação de dados não-estruturados de processos streaming em tempo real*. 2021. 86 f. Dissertação (Mestrado em Ciências Computacionais) Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2021.

A duplicação de dados é um problema comum em aplicações de processamento contínuo de dados, que pode ocorrer devido aos erros de software ou à adoção de medidas de prevenção de perda de dados, esse problema é usualmente tratado após a ingestão dos dados de um repositório, seja por um processo auxiliar ou pelas próprias análises sobre os dados desconsiderarem duplicatas. Entretanto, com a necessidade de análises feitas o mais próximo do momento da criação de um dado e no menor tempo possível, ambas abordagens se tornam insuficientes para atender a ambos os requisitos, sendo necessário que a deduplicação ocorra em tempo de ingestão. Este trabalho explora métodos podem ser utilizados com a biblioteca Apache Spark para tratar a deduplicação de dados em tempo real, analisando o uso de recursos e o tempo de entrega de cada método e identificando seus casos de usos. E investiga operadores de deduplicação nativos do Apache Spark (*distinct* e *dropDuplicates*) e ferramentas auxiliares (RocksDB, Apache Ignite e Apache Hudi), que fornecem mecanismos de deduplicação de dados e tolerância a falhas a aplicação. Os resultados experimentais mostram que há um aumento no tempo de entrega dos dados ao utilizar mecanismos externos, mas que estes mecanismos se tornam primordiais para que um processo de ingestão garanta que não haja a perda dos dados utilizados durante a deduplicação, garantindo que nenhuma duplicata seja persistida. Além disso, outros fatores influenciam a escolha do melhor método de deduplicação, como o uso de recursos computacionais e o tamanho dos dados persistidos.

Palavras-chave: Deduplicação de dados. Processamento de dados em *streaming*. Tolerância a falhas.

ABSTRACT

ESTEVES, João Victor Azevedo. Deduplication of unstructured data from real-time streaming processes 2021. 86 f. Dissertação (Mestrado em Ciências Computacionais) - Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2021.

Data duplication is a widespread problem in data streams processing applications that can occur due to software errors or the adoption of data loss prevention measures, this problem is usually treated after ingesting the data into a repository, either by an auxiliary process or by the analyzes themselves about the data disregarding duplicates. However, with the need for analyzes made as close to the moment of data creation and in the shortest possible time, both approaches become insufficient to meet both requirements, making it necessary for deduplication to occur during ingestion. This work explores methods that can be used with the Apache Spark library to deal with data deduplication in real time, analyzing the use of resources and the delivery time of each method and identifying their use cases. This work investigates Apache Spark native deduplication operators (distinct and drop Duplicates) and auxiliary tools (RocksDB, Apache Ignite and Apache Hudi) that provide data deduplication and fault tolerance mechanisms to the application. The experimental results show that there is an increase in the data delivery time when using external mechanisms, but that these mechanisms become essential for an ingestion process to ensure that there is no loss of data used during deduplication, ensuring that no duplicates are persisted. In addition, other factors influence the choice of the best deduplication method, such as the use of computational resources and the size of the persisted data.

Keywords: Data deduplication. Streaming data processing. Fault tolerance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
ANSI	*American National Standards Institute* (Instituto nacional americano de padrões)
API	*Application Programming Interface* (Interface de programação de aplicação)
AWS	*Amazon Web Services*
CAP	*Consistency, Availability e Partition tolerance* (Consistência, disponibilidade e tolerância a partições)
CPU-bound	*Bound to CPU usage* (Limitado por operações que usam o processador)
EMR	*Elastic MapReduce*
ETL	*Extract, Transform and Load* (Extrair, transformar e carregar)
GDPR	General Data Protection Regulation (Regulamento Geral de Proteção de Dados)
HDFS	*Hadoop Distributed File System* (Sistema de arquivos distribuídos do Hadoop)
IDE	*Integrated Development Environment* (Ambiente de desenvolvimento integrado)
IO-bound	*Bound to input and output usage* (Limitado por operações de entrada e saída de dados)
IoT	*Internet of Things* (Internet das coisas)
LGPD	Lei Geral de Proteção de Dados Pessoais
NoSQL	*Not only SQL* (Não apenas SQL)
NRT	*Near Realtime* (Próximo do tempo real)
PubSub	*Publish and Subscribe* (Publicar e inscrever)
RDD	*Resilient Distributed Dataset* (Conjunto de dados distribuído resiliente)
S3	*Simple Storage Service* (Serviço de armazenamento simples)
SLA	*Service Level Agreemen* (Acordo de nível de serviço)
SQL	Structured Query Language (Linguagem de consulta estruturada)
SSD	*Solid State Drive* (Disco de estado sólido)
UDF	*User-Defined Function* (Funções definidas pelo usuário)

vCPU	*Virtual CPU* (Núcleos de processamento virtuais)
WAL	*Write-ahead Log* (Registros de gravação antecipada)

SUMÁRIO

INTRODUÇÃO

Desde o início do uso comercial da Internet no fim dos anos 80, é possível observar que a quantidade de dados gerados cresce exponencialmente. Segundo Jacobson (2013), 90% do total de dados havia sido criado nos últimos dois anos e 2,5 exabytes (2,5 quintilhões de bytes) eram gerados diariamente. Tal explosão é ocasionada tanto por *hardware*, pelo surgimento dos *smartphones* e do movimento IoT (*Internet of Things*) [Atzori *et al.* 2010], quanto por *software*, pela popularização das redes sociais e dos serviços de *streaming* (serviços de consumo de conteúdo digital).

À medida que a sociedade se torna mais interconectada, as organizações estão produzindo grandes quantidades de dados como resultado de processos de rastreamento de *websites* [Guo *et al*. 2019], movimentações financeiras [Hendricks 2017], experimentos científicos de larga escala [Dong *et al.* 2013], entre outras razões.

Devido a este aumento no volume de dados, a dificuldade de analisar e processar tais dados torna-se evidente. Carregar um volume grande de dados em memória, ultrapassando o limite de várias máquinas com discos rígidos de alta capacidade, e processá-los, em tempo viável, torna-se impraticável.

Além do grande volume de dados e da velocidade em que estes dados devem ser processados, surgem outros desafios característicos, como a variedade e a veracidade dos dados, que atualmente são estudadas sobre o termo *Big Data*, permitindo maiores estudos e resoluções de tais desafios tanto para a academia quanto para a indústria.

Inicialmente, estes dados podem ser processados em lotes (*batches*) [Ingram 2009], isto é, através de uma aplicação que é agendada de tempos em tempos para pontualmente extrair os dados de uma origem, transformá-los de acordo com a estrutura desejada e persistir os resultados em uma fonte de dados acessível após o término do processamento. Este processamento inicial dos dados é conhecido como *extract, transform and load* (ETL) [Theodorou *et al*. 2017].

Porém, o processo ETL está se tornando cada vez mais custoso devido ao acúmulo de dados durante os anos. Dessa forma, o processamento de dados em *batch* torna-se ineficaz para uma análise dos dados em tempo real. A definição de “tempo real”, neste caso, é considerada como o tempo de disponibilização dos dados o mais próximo possível

do momento de sua criação, como exemplo, neste trabalho era um requisito inegociável do domínio que os dados fossem disponibilizados em até 10 minutos após a sua criação.

Atendendo a esta necessidade, surgem abordagens para criação de aplicações de processamento de dados em fluxo (*streaming*), isto é, os dados são processados à medida que são criados.

Entretanto, processar dados próximos ao momento de sua criação inviabiliza que um processo de validação remova dados que foram gerados com inconsistências de domínio (como a unicidade de um dado), ou malformados, prejudicando os resultados obtidos por este processo, caso a aplicação não possua nenhum tratamento para essas eventualidades.

Para tratar estas inconsistências, o *streaming* não pode validar os dados presentes no destino da ETL, visto que o tempo de consulta a estes dados tende a aumentar drasticamente, aumentando proporcionalmente o volume de dados consultados no destino, conforme o *streaming* prossegue sua execução.

Com o objetivo de evitar esta consulta, é persistido na memória da aplicação *streaming* todos os dados estritamente necessários para o tratamento desta inconsistência. Esta memória é conhecida como o estado do *streaming*, e não é utilizada somente neste caso de uso, como também para processamento de agregações [Quoc *et al.* 2017], entre outros exemplos.

A adição deste tratamento ao *streaming*, contudo, pode prejudicar seu desempenho, causando atrasos na entrega de um resultado, ou na má utilização de recursos computacionais. Por exemplo, para garantir a unicidade de dados é necessário que todo o dado ou um identificador único esteja disponível no estado do *streaming* por um período, porém, caso novos dados sejam processados antes dos dados anteriores serem eliminados da memória, poderá ocorrer um erro de falta de memória nas máquinas hospedeiras da aplicação.

Além disso, devido à natureza de aplicações *streaming*, de estar em execução constantemente, sem tempo específico para conclusão, é possível que ocorra algum erro imprevisto ou não tratável no ambiente em que a aplicação é executada, como por exemplo, uma falha de *hardware* ou uma exceção não tratada pela própria aplicação. Nestes cenários, é mandatório que a aplicação seja capaz de persistir o estado conhecido até então fora de seu escopo de memória, preferencialmente em um ambiente externo e autônomo, para que a reinicialização da aplicação seja capaz de produzir resultados precisos.

Com base nestes aspectos, este trabalho busca estudar algumas estratégias de deduplicação de dados (isto é, garantir que não haja dois registros idênticos em uma massa

de dados) em tempo real, com o objetivo de reduzir o tempo total desde a ingestão de um dado novo até a sua disponibilização, avaliando a capacidade de uma estratégia ser tolerante a falhas, isto é, não perder o estado consistente de dados e da aplicação caso haja uma reinicialização.

O processamento dos dados, em algumas organizações, deve considerar a existência de duplicidade de informações para tentar diminuir o volume grande destes dados, assim como prevenir inconsistências em uma análise posteriori. Um exemplo real deste tipo de problema é o que ocorre em uma empresa global de comércio eletrônico através da disponibilização de anúncios digitais em seus classificados, onde todas as etapas do ciclo de vida de todo anúncio em sua plataforma são persistidas.

Em princípio, os dados dos anúncios eram registrados em um banco de dados relacional. Entretanto, devido ao seu crescimento e a velocidade de produção dos registros, a infraestrutura deste banco não tem suportado a quantidade de operações de escritas geradas por cada evento do ciclo de vida de um anúncio diariamente.

Além disso, a alta quantidade de processos e aplicações legadas, que dependem da unicidade destes dados na fonte, torna impraticável a reescrita das consultas para que ocorra a eliminação da duplicidade dos registros durante a execução da consulta. Deve-se considerar também, que o uso da cláusula DISTINCT nas consultas submetidas ao banco de dados relacional degrada muito o seu desempenho.

O ideal é que a duplicidade dos dados seja eliminada no nível de armazenamento dos dados, para que não seja necessário reescrever as consultas legadas que, muitas vezes, estão inseridas em aplicações caixa preta (isto é, não é conhecido, ou se perdeu o conhecimento, de como a aplicação funciona internamente).

Além da demanda pela unicidade de dados, é mandatório que o novo repositório de dados atenda aos seguintes requisitos:

- Ter o tempo médio de execução de consultas sobre os dados menor ou igual ao tempo obtido no banco de dados atual, visto que muito do valor obtido por análises sobre estes dados é devido a este tempo ser considerado satisfatório;
- Tenha uma infraestrutura que utilize recursos computacionais de forma similar ou mais econômica que justifique a adoção de uma nova solução de repositório de dados ao invés de escalar a infraestrutura atual.

Com base na necessidade da empresa de manter a unicidade dos dados no processo de ingestão (isto é, de aquisição de dados de várias fontes), de diminuir a quantidade de recursos utilizados e de agilizar o tempo de processamento e resposta às consultas, este

trabalho tem como objetivo principal pesquisar as vantagens, desvantagens e limitações de algumas soluções, utilizando o *framework* Apache Spark[1], amplamente utilizado nos processos de transformação de dados da empresa.

A presente dissertação tem como objetivos específicos analisar:

- As capacidades e os limites das funções nativas de deduplicação do Apache Spark (*distinct* e *dropDuplicates*);
- As ferramentas complementares que possam auxiliar na garantia da unicidade dos registros de toda a população dos dados (ex.: um banco de dados, um formato de arquivo);
- O cumprimento do tempo mínimo de disponibilização dos dados ingeridos e do tempo de resposta das consultas;
- A tolerância a falhas do *streaming* ao utilizar cada função ou ferramenta, tanto em nível de aplicação quanto em nível de *hardware*.

Devido ao custo de desenvolvimento de soluções que fogem do uso destas ferramentas e pela dificuldade em passar o conhecimento destas possíveis soluções para o time de Engenharia de Dados desta empresa, este trabalho se limita apenas a estudar algumas ferramentas de *Big Data open-source*.

Os resultados esperados dessa pesquisa são:

- Um comparativo entre as soluções estudadas, em termos de tolerância a falhas, uso de memória e de armazenamento;
- Identificação de cenários em que cada solução poderia ser benéfica ou limitar o processamento de dados.

Esta dissertação está organizada da seguinte maneira:

O **Capítulo 1** apresenta uma visão geral dos conceitos utilizados neste trabalho;

O **Capítulo 2** discute a natureza dos dados utilizados nesta dissertação, os requisitos do domínio dos dados e a metodologia empregada pelos testes;

O **Capítulo 3** avalia os resultados obtidos por cada teste sobre a perspectiva de cada requisito;

O **Capítulo 4** sumariza os resultados obtidos, apresentando as conclusões finais, limitações, contribuições e possíveis trabalhos futuros.

[1] https://spark.apache.org/, acessado em março de 2020.

1. CONCEITOS BÁSICOS

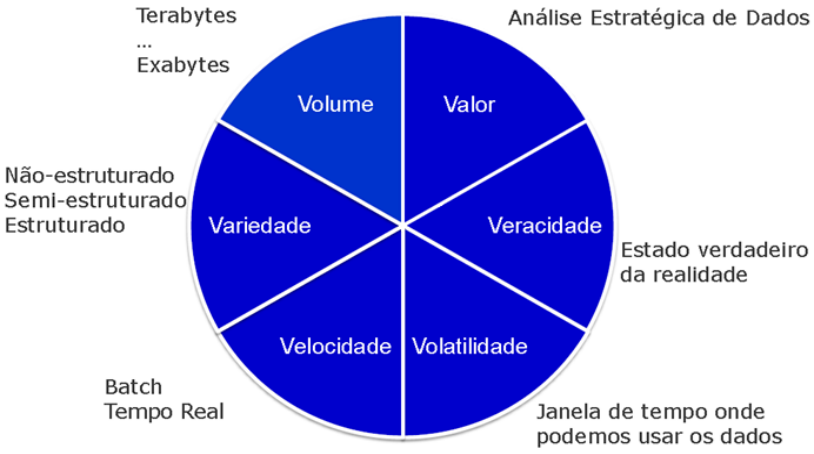
Este capítulo apresenta os principais conceitos e ferramentas envolvidas na área de *Big Data*, assim como a computação em nuvem, que tem auxiliado o processamento de um grande volume de dados.

1.1. Big Data

O termo *Big Data*, cunhado para se referir a soluções que lidam com quantidades massivas de dados, está em uso desde a década de 90, popularizado por John Mashey, quando fez referência à necessidade de se processar dados que são gerados em volume crescente a cada segundo [Mashey 1998]. Este termo também engloba aplicações que devem atuar sobre conjuntos de dados com tamanhos além da capacidade de ferramentas de *software* comumente usadas na indústria para capturar, organizar, gerenciar e processar dados dentro de um tempo decorrido tolerável [Snijders *el al.* 2012]. Esta área requer um conjunto de técnicas e tecnologias com novas formas de integração para conseguir lidar com conjuntos de dados que são diversificados, complexos e de grande escala [Hashem *et al.* 2015].

Normandeau (2013), Inderpal Bhandar, diretor de dados da Express Scripts, sinalizou em sua apresentação no evento "Big Data Innovation Summit", em Boston, que existem seis características da área de *Big Data* que merecem a atenção dos profissionais que trabalham com dados, e essas características podem ser definidas por seis “V’s”: Volume, Variedade, Velocidade, Volatilidade, Veracidade e Valor, conforme mostra a Figura 1. A seguir, estes aspectos são descritos mais detalhadamente.

Figura 1 – Os 6 “V’s” que definem Big Data.

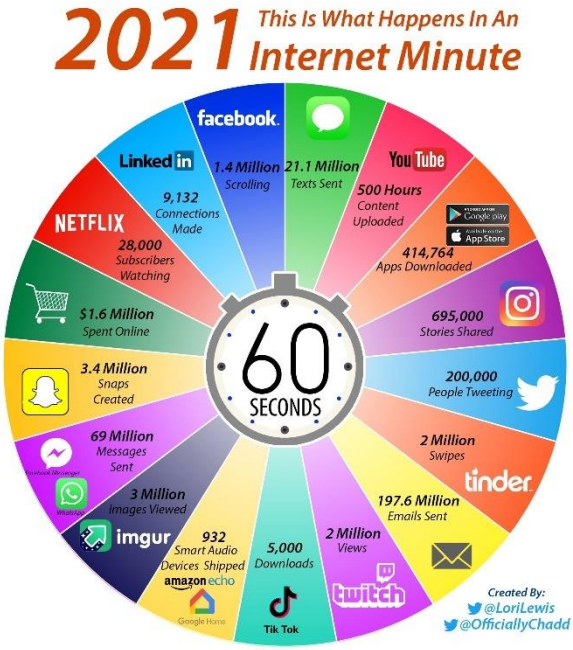


Fonte: Sousa, 2015

1.1.1. Volume

Volume diz respeito a larga quantidade de dados [Berman 2018]. Essa quantidade vem de dados provenientes de diversos fluxos de dados, contendo vários formatos e que estão sendo gerados em uma velocidade muito alta a partir de fontes de dados físicas, digitais e humanas [Gubbi *et al.* 2013], conforme apresentado na Figura 2.

Figura 2 – Quantificação de eventos que ocorreram em um minuto na Internet em 2021.



Fonte: Lewis, 2021

A escala de dados agora é de terabytes, petabytes e exabytes, e para pôr em perspectiva, até o final de 2020 estima-se que serão gerados 50 vezes a quantidade de dados que se registrou em 2011 [Gantz & Reinsel 2012], sendo que em 2013 foi estimado que 90% de todos os dados já gerados foram criados nos dois anos anteriores [Jacobson 2013] [Manyika 2017].

O grande desafio deste aspecto está sendo em como armazenar todo este volume para o fácil e rápido acesso de leitura e de escrita, sendo algumas soluções de sistemas de arquivos distribuídos como o HDFS[2] (*Hadoop Distributed File System*) e o Amazon S3[3] (*Simple Storage Service*) grandes destaques para solucionar este problema. Tais tecnologias são detalhadas posteriormente.

1.1.2. Velocidade

A partir da Figura 2, também é possível perceber o quão rápido os dados são gerados a cada minuto. Esta velocidade de geração e atualização dos dados é resultado da constante mudança do conteúdo dos dados, consequente da absorção de coleções de dados complementares, da introdução de dados legados e de fluxo de dados contínuos de várias fontes [Berman 2018].

No início dos sistemas de informação, em que os dados eram recebidos e enviados em *batches*, era usual a arquitetura de tais sistemas considerarem receber uma atualização da base de dados uma vez no final de cada dia ou até mesmo uma vez ao final de cada semana. Computadores e servidores exigiam um tempo considerável para processar os dados e atualizar a base de dados. Na era do *Big Data*, os dados são atualizados e consumidos em um curto período. Assim, requer-se que o processamento e o consumo destes dados sejam feitos o mais próximo possível do momento de sua captura, praticamente em tempo real.

Por exemplo, o aplicativo de navegação e mapa de trânsito Waze[4] necessita receber dados em tempo real de acidentes, engarrafamentos, interdições, entre outros aspectos do trânsito para que seja possível o cálculo do trajeto que levará seus usuários aos seus

[2] https://hadoop.apache.org/, acessado em março de 2020.
[3] https://aws.amazon.com/s3/, acessado em março de 2020.
[4] https://www.waze.com/, acessado em março de 2020.

respectivos destinos no menor tempo possível. A abordagem em *batch*, neste caso, só permitiria que o usuário pudesse consultar este trajeto com antecedência, recebendo o resultado somente após a execução do *batch* agendando, possivelmente demorando algumas horas.

1.1.3. Variedade

A variedade de dados refere-se aos vários tipos e formatos de dados (ex.: imagens, documentos entre outros) que precisam ser suportados pelas soluções de tecnologia da área de *Big Data* [Berman 2018]. A variedade de dados traz desafios para as aplicações em relação à integração, transformação, processamento e armazenamento de dados.

O foco do *Big Data* é em geral associado aos dados não estruturados, isto é, dados gerados sem um formato padrão e homogêneo a todos os dados. Por exemplo, aplicações que lidam com vídeos, imagens e textos em diferentes formatos exigem diferentes atributos a serem gerados em seus dados e podem ser levados ao tratamento de uma única aplicação. No entanto, tais aplicações, frequentemente, também englobam dados semiestruturados (por exemplo, arquivos CSV, JSON, XML) e estruturados (por exemplo, dados de uma base de dados relacional) [Dedic & Stanier 2016].

1.1.4. Veracidade e Valor

Estes 3 “V”s, descritos anteriormente, são responsáveis apenas por definir como popular repositórios de grandes volumes de dados, entretanto, realizar a ingestão de dados sem a curadoria e a catalogação da informação armazenada nestes repositórios pode dificultar a extração de veracidade e valor destes dados por meio de relatórios e modelos preditivos, contribuindo para “poluição” desses repositórios (fenômeno conhecido como *data swamp* [Brackenbury *et al*. 2018]).

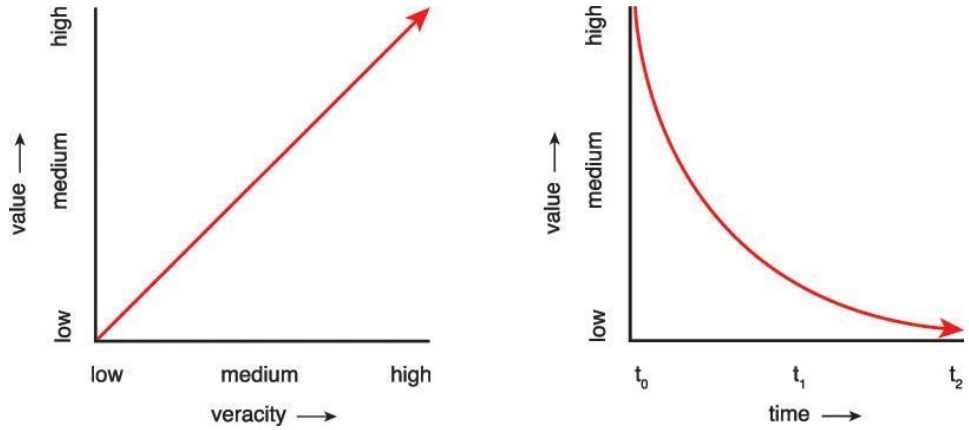
Veracidade refere-se à qualidade, ou fidelidade dos dados. Os dados que entram nos repositórios de dados precisam ser avaliados quanto à sua qualidade, o que pode levar a atividades de processamento de dados para resolver dados inválidos e remover ruídos, isto

é, dados que não podem ser convertidos em informações e, portanto, não têm valor [Erl *et al.* 2016].

Valor é definido como a utilidade dos dados para um negócio [Erl *et al.* 2016], sendo este "V" intuitivamente relacionado à veracidade, pois quanto maior a fidelidade dos dados, mais valor eles possuem.

Além de ser proporcional à veracidade dos dados, o valor também é dependente do tempo de processamento dos dados, pois os resultados da análise têm prazo de validade; por exemplo, uma cotação de ações com atraso de 20 minutos tem pouco ou nenhum valor para negociar em comparação com uma cotação com 20 milissegundos. Estas relações são ilustradas pela Figura 3.

Figura 3 – Relações entre valor e veracidade e valor e tempo.



Fonte: Erl *et al*., 2016

O monitoramento dessas duas características não é tão simples de ser realizada quanto monitorar volume, velocidade e variedade, sendo necessário analisar, na maioria das vezes, manualmente os dados. Entretanto, o nível de dificuldade da análise aumenta proporcionalmente ao aumento das próprias características originais. Este desafio serve de base para vários estudos sobre ferramentas de validação automática de dados [Rubin & Lukoianova 2013] [Debattista *et al.* 2015] [Brackenbury *et al*. 2018].

1.1.5. Volatilidade

A volatilidade refere-se a quanto tempo os dados são válidos e quanto tempo devem ser armazenados [Normandeau 2013]. No entanto, os repositórios de dados que foram projetados no início do desenvolvimento da área de *Big Data* foram idealizados para

abdicarem de operações de atualização e exclusão devido ao alto custo de se localizar os dados que devem sofrer a operação, comparado ao custo de acrescentar dados ao repositório e a atenção de tratar duplicidade dos dados na consulta ao repositório.

Atualmente, esta abdicação deve ser revista para que os usuários de um serviço possam exercer seus direitos concebidos por leis de proteção aos dados como a *General Data Protection Regulation* (GDPR[5]) e a Lei Geral de Proteção de Dados Pessoais (LGPD[6]), como o direito de reivindicar os dados que ele possui sobre si assim como a remoção desses dados.

Para o atendimento destas leis, é necessário que a gerência dos dados persistidos no repositório de dados seja feita em tempo real, não permitindo que dados sensíveis sejam persistidos, ou que sejam indexados para fácil acesso, em caso de uma operação de atualização, ou exclusão.

1.1.6. Características de Big Data neste trabalho

Analisando este trabalho através da ótica destas características de Big Data, é possível encontrar a ocorrência dos seguintes "V"s:

- **Velocidade e Volume:** devido a quantidade de dados que devem ser processados rapidamente, no máximo 10 minutos;
- **Veracidade e Valor:** devido a necessidade de haver a deduplicação de dados, já que uma consulta pode gerar resultados incorretos, com uma contagem de ocorrências maior do que a realidade;

Contudo, não há ocorrência dos "V"s de Volatidade, por não haver a necessidade de atualização ou remoção de dados históricos, ou de Variedade, por todas as mensagens serem consumidas em formato JSON.

[5] https://gdpr.eu/, acessado em março de 2020.
[6] http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/L13709.htm, acessado em março de 2020.

1.2. **Armazenamento de dados**

Por muito tempo, o uso de bancos de dados relacionais supriu a necessidade de persistir dados para consultas futuras, e muitas dessas bases ainda são utilizadas como primeiro repositório de dados no início de projetos, ainda com pouco volume de dados, assim como para projetos com *terabytes* de dados [Szalay *et al.* 2000] [Dobos *et al.* 2013].

Entretanto, o custo e a dificuldade de escalar (isto é, adicionar mais capacidade de processamento e/ou armazenamento) uma base de dados relacional para atender a múltiplas requisições de leitura e escrita de dados, além de validar possíveis restrições impostas sobre os dados, podem inviabilizar o uso de apenas uma base de dados relacional [Cattell 2011] [Li & Manoharan 2013].

Bases de dados não-relacionais, também conhecidas como bases NoSQL (*Not only SQL*), tem ganhado mais visibilidade recentemente, mesmo existindo desde a década de 1960 [Leavitt 2010], propondo formas de armazenamento alternativas ao formato tabular das bases de dados relacionais, sendo os formatos mais difundidos, segundo Elmasri & Navathe (2016):

- **Baseado em documentos:** estes sistemas armazenam dados na forma de documentos usando formatos conhecidos, como JSON. Os documentos são acessíveis através da identificação do documento, mas também podem ser acessados rapidamente usando outros índices. Exemplo de banco de dados orientado a documentos: MongoDB[7];
- **Baseado em chave-valor**: esses sistemas possuem um modelo de dados simples baseado no acesso rápido da chave ao valor associado à chave; o valor pode ser um registro simples de tipo primitivo ou um objeto mais complexo. Exemplo de banco de dados chave-valor: Redis[8];
- **Baseado em colunas**: esses sistemas particionam uma tabela por coluna em famílias de colunas, onde cada família de colunas é armazenada em seus próprios arquivos. Exemplo de banco de dados orientado a colunas: Apache Cassandra[9];

[7] https://www.mongodb.com/, acessado em março de 2020.
[8] https://redis.io/, acessado em março de 2020.
[9] https://cassandra.apache.org/, acessado em março de 2020.

- **Baseado em grafos**: os dados são representados como grafos, onde os nós contêm os registros de dados e as arestas entre os nós representam o relacionamento entre os nós. Exemplo de banco de dados orientado a grafos: Neo4j[10].

Estas bases, comumente, não dão garantias de propriedades transacionais ACID (Atomicidade, Consistência, Isolamento e Durabilidade), como as bases relacionais, além de priorizarem até duas das seguintes características, de acordo com o teorema CAP (*Consistency, Availability e Partition tolerance*) [Brewer2000] [Elmasri & Navathe 2016]:

- **Consistência (*Consistency*):** significa que os nós possuem as mesmas cópias de um item de dados replicado e visível para várias transações (cópias replicadas de forma consistente);
- **Disponibilidade (*Availability*):** toda operação de leitura ou escrita de um registro é processada com êxito ou caso contrário, o nó recebe uma mensagem de que a operação não pode ser concluída (dados disponíveis).
- **Tolerância à partição (*Partition tolerance*):** o banco de dados consegue operar de forma distribuída, mesmo com a falha de um ou mais nós de processamento.

Para a consulta de dados frequentemente acessados, também conhecidos como "dados quentes" [Levandoski *et al*. 2013], bases relacionais e não-relacionais são excelentes e recomendadas. Contudo, para consulta de dados históricos ("dados frios"), soluções de modelagem e armazenamento em *data warehouse* e *data lake* são mais promissores. Segundo Stein & Morrison (2014), esses dois tipos de soluções possuem as seguintes diferenças:

- **Estrutura dos dados:** dados persistidos em um *data warehouse* tendem a ser pré-processados, ganhando estrutura (*schema*), podendo perder a informação do dado original; *data lakes* não impõe estrutura aos dados, e armazena o dado original, assim como o dado processado;
- **Exploração dos dados:** *data warehouses* possuem dados bem catalogados, isto é, estão organizados em uma hierarquia bem definida, com políticas de acesso restritas; *data lakes* comumente são projetados sem organização restrita e sem restrição de acessos;
- **Custo monetário:** a maioria das soluções de *data warehouse* são proprietárias e de alto custo; *data lakes* podem ser configurados em *hardware* barato utilizando

[10] https://neo4j.com/, acessado em março de 2020.

soluções *open-source* como o HDFS ou em armazenamento em nuvem, como o Amazon S3; e

- **Dificuldade:** *data lakes* são mais fáceis de configurar que *data warehouses*, tendo inclusive soluções autogerenciadas por serviços de nuvem, como o Azure Data Lake Store[11].

Pela facilidade de implementação, baixo custo monetário e pela alta resiliência oferecida por serviços de armazenamento em nuvem, é comum que empresas prefiram implementar *data lakes* [Stein & Morrison 2014].

Como *data lakes* tendem a ser permissivos quanto a entrada de dados, é comum que o volume de *bytes* cresça exponencialmente, aumentando rapidamente o custo de armazenamento. Uma solução para este problema é, durante o processo de ingestão, os dados sejam agrupados em grandes arquivos, e estes serem comprimidos por bibliotecas como Snappy[12] e o LZ4[13].

Para auxiliar a compressão dos dados, e assim diminuir ainda mais o espaço ocupado, os arquivos são escritos em um formato de armazenamento binário, orientados a colunas (ex.: Parquet[14] ou ORC[15]) ou linhas (ex.: Avro[16]), sendo a escolha de qual formato utilizar dependente de qual operação, dentre leitura (com preferência por formatos colunares) e escrita (com preferência por formatos baseados em linha), será mais recorrente sobre o dado [Ivanov & Pergolesi 2019].

1.3. **Processo de ingestão de dados**

Ingestão de dados, também conhecido como ETL, é o processo de obter dados de uma ou mais fonte(s) e persisti-los em um repositório de dados; este processo pode apenas copiar os dados da fonte, mantendo o formato original do dado, ou transformá-lo (ex.: atribuindo uma estrutura ou convertendo tipos dos dados originais), como mostra a Figura 4.

---

[11] https://azure.microsoft.com/services/storage/data-lake-storage/, acessado em março de 2020.
[12] https://google.github.io/snappy/, acessado em março de 2020.
[13] https://lz4.github.io/lz4/, acessado em março de 2020.
[14] https://parquet.apache.org/, acessado em março de 2020.
[15] https://orc.apache.org/, acessado em março de 2020.
[16] https://avro.apache.org/, acessado em março de 2020.

Figura 4 – Diagrama de uma ETL.

Fonte: O autor, 2021

Os primeiros processos de ingestão eram realizados por aplicações simples, através da consulta dos dados de outras aplicações, seja por meio de consultas na base de dados ou por meio de API's (*Application Programming Interface*), e persistidos em uma base de dados local ou em arquivo.

Entretanto, como foi apresentado na seção 2.2, tanto o aumento do volume quanto a velocidade da geração de dados se tornam fatores limitantes para este processo de persistência, sendo necessário processos que se beneficiem de arquiteturas distribuídas e que otimizem o processo de ETL, de forma a processar todo conjunto de dados disponível em tempo hábil.

ETLs que agem sobre grandes lotes (*batches*) de dados foram as primeiras alternativas aos processos originais, estes são processos descritos por *frameworks* de programação distribuída, como o Open MPI[17] e o Hadoop MapReduce, que carregam um *batch* de dados para processamento e persistem os resultados processados.

Os exemplos de *frameworks* anteriores tiveram como objetivo facilitar a implementação desses processos distribuídos em uma linguagem de alto nível, entretanto, pouco auxiliam no tratamento otimizado dos dados, sendo responsabilidade do desenvolvedor de implementar rotinas otimizadas para estas ETLs, aumentando a complexidade de implementação de uma ETL.

Alguns *frameworks* surgiram com primitivas que também abstraíram a implementação dessas rotinas, contando com o benefício da maioria destas serem *open-source*, o que permite que um público grande de desenvolvedores utilize as mesmas rotinas e proponham melhorias a estas e que outros desenvolvedores possam usufruir; o principal exemplo deste tipo de biblioteca é o Apache Spark, com a utilização da abstração de dados RDD (*Resilient Distributed Dataset*) que evoluiu para uma abstração mais otimizada e com

[17] https://www.open-mpi.org/, acessado em março de 2020.

primitivas semelhantes às utilizadas em consultas de bases de dados, o *DataFrame* [Zaharia *et al.* 2016].

A execução de processos em *batch* acontece em períodos agendados pelo desenvolvedor, tentando equilibrar o tempo estipulado pelo cliente para a disponibilização dos dados no repositório através de um *service level agreement* (SLA), o tempo de produção dos dados na fonte, o tempo de execução da ETL para o volume de dados produzidos e o tempo da execução do próximo *batch*.

Esta característica dos processos em *batch* pode ser um fator limitante quando a SLA necessita que os dados estejam disponíveis próximos do tempo em que foram criados na fonte (NRT, *near realtime*), já que muitos destes processos, ao serem inicializados, precisam de um tempo para configurar o ambiente de execução, ou até mesmo alocar os recursos onde as ETLs irão ocorrer.

Eis então que surgem *frameworks* para processamento de fluxo contínuo de dados (*streaming*), como Flink[18] e o Storm[19], além de outros como o próprio Apache Spark, que possuem primitivas de *streaming*.

Este tipo de processamento, diferente do processamento em *batch*, inicializa uma aplicação, mantendo-a em execução por tempo indeterminado, que recebe novos dados durante sua execução, seja por meio de notificações de novos arquivos em um sistema de arquivos, ou por meio de mensagerias, como o Apache Kafka[20].

Embora os processos de *streaming* possam atender aos mesmos requisitos de processos em *batches*, podendo dar a impressão de não haver mais necessidade deste tipo de processo, o mesmo ainda é preferido para ETLs históricas, que possuem SLAs maiores (ex.: um dia, um mês), e para ETLs com pouco volume de dados de entrada, devido ao desenvolvimento mais fácil destas ETLs, assim como a economia de recursos, visto que um processo em *batch* não demanda o uso contínuo de uma máquina.

1.3.1. Arquiteturas de ingestão de dados

O que se observa na prática é o uso de processos em *streaming* e *batches* em diferentes estágios da ingestão de dados, visto que a análise sobre dados em tempo real nem

[18]https://flink.apache.org/, acessado em março de 2020.
[19] https://storm.apache.org/, acessado em março de 2020.
[20] https://kafka.apache.org/, acessado em março de 2020.

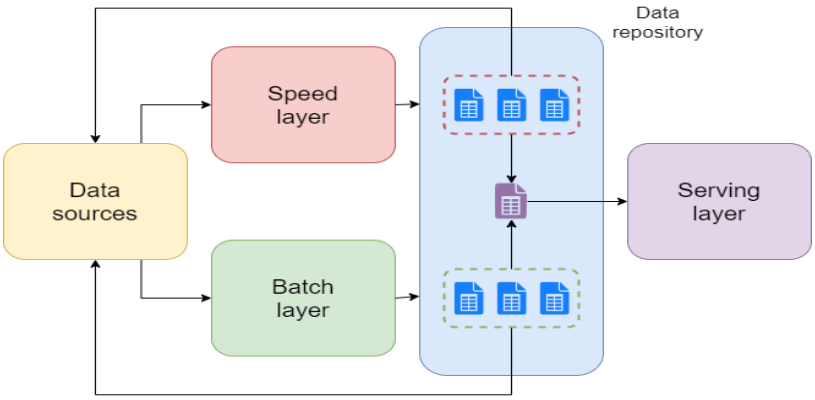
sempre necessita que todos os campos de um dado sejam avaliados ou que um campo esteja consolidado, isto é, um campo pode registrar apenas uma aproximação do resultado final durante a análise corrente.

As SLAs de tempo real tendem a priorizar ter o máximo de dados no momento da análise, podendo relaxar algumas restrições quanto ao valor do dado. Por exemplo, um sistema que monitora a temperatura de uma sala, pode aceitar a média aproximada das temperaturas obtidas por vários sensores desta sala desde que a média obtida tenha uma taxa de erro abaixo de um limite razoável.

Os resultados dessa primeira ingestão são então disponibilizados em um repositório de dados, ou também em uma mensageria, para que uma ETL posterior possa transformar este resultado intermediário no dado final, que será analisado a longo prazo por análises históricas.

Um padrão de arquitetura difundido para este tipo de ingestão de dados é a arquitetura Lambda [Marz & Warren 2015] (Figura 5), que define uma camada de processamento em *streaming* (*Speed Layer*) e a camada de processamento em *batch* (*Batch Layer*) atuando sobre uma ou mais fontes de dados (*Data sources*). Os resultados de ambas as camadas são unificados para serem disponibilizados ao usuário final em uma camada de consulta (*Serving Layer*) que abstraia a complexidade das duas camadas anteriores.

Figura 5 – Arquitetura Lambda.



Fonte: O autor, 2021

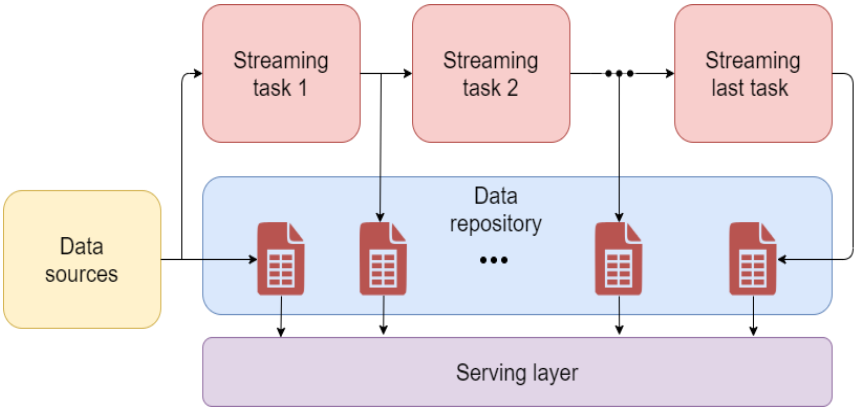
Apesar desta arquitetura facilitar o atendimento de uma SLA de tempo real, a lógica de unificação dos resultados pode dificultar sua implementação, visto que cada conjunto de dados terá de ter sua própria lógica implementada por trás da camada de consulta, e qualquer possível mudança na fonte de dados (ex.: adição ou remoção de um novo campo) pode demorar a se refletir para o usuário final. Além do problema de unificação, a

implementação de duas aplicações por processo de ingestão pode dificultar a manutenção conforme a quantidade de processos aumenta.

A arquitetura Kappa [Kreps 2014] foi proposta para resolver ambos os problemas, substituindo os processos em *batches* por outros processos de *streaming*, o conceito geral é ter um *pipeline* de *streaming* (isto é, vários processos de *streaming* em sequência, onde o resultado de um é a entrada do seguinte), onde o primeiro processo consome os dados da fonte inicial, produzindo um resultado que será armazenado no repositório de dados, assim como será consumido pelo próximo processo do *pipeline*, até que o resultado final seja obtido.

Esta arquitetura (Figura 6) tem o benefício de disponibilizar com mais rapidez os dados históricos, já que por existirem mais etapas entre o consumo da origem e a consolidação do resultado, é esperado que os processos interajam com lotes menores de dados ao invés de aplicarem todas as transformações diretamente. Sendo assim, a análise de um usuário, que só demande as primeiras transformações de um conjunto de dados, pode ser adiantada.

Figura 6 – Arquitetura Kappa.



Fonte: O autor, 2021

1.3.2. Ingestão de dados com persistência de estado

Existem ocasiões em que uma ETL necessita consultar dados de execuções posteriores à execução corrente (ex.: quando é necessário evitar que dados que já existam no repositório sejam persistidos novamente), sendo esta uma situação comum e de fácil resolução para processos em *batch*, visto que necessita apenas que o processo receba os

dados produzidos anteriormente, assim como os dados da fonte. Entretanto, para processos em *streaming*, o consumo contínuo dos dados e necessidade das transformações ocorrerem em tempo NRT, impossibilita a leitura de dados históricos, conforme o volume destes dados cresce.

Uma solução ingênua para este problema é persistir os dados históricos na memória principal dos processos de *streaming*, para assim, serem acessados durante a transformação de novos dados sem ser necessária a leitura dos dados no repositório. Essa solução é conhecida como persistência de estado em memória, sendo seu principal problema, o alto consumo de memória principal para volumes grandes. Para mitigar isto, é usado o conceito de *windowing*, onde uma janela de tempo (*window*) específica quanto tempo um dado pode ser persistido em memória, até que este possa ser eliminado sem prejudicar as transformações da ETL.

Geralmente, o tamanho de janela está em acordo com regras de negócio (por exemplo, até quando um dado é necessário para a análise). Para casos em que o dado é necessário por um tempo impraticável (exemplo: um ano), com relação à razão entre volume total de dados e tamanho da memória principal disponível, são criadas SLAs, que garantem que algum percentual dos dados irá ser analisado de acordo com as regras de negócio.

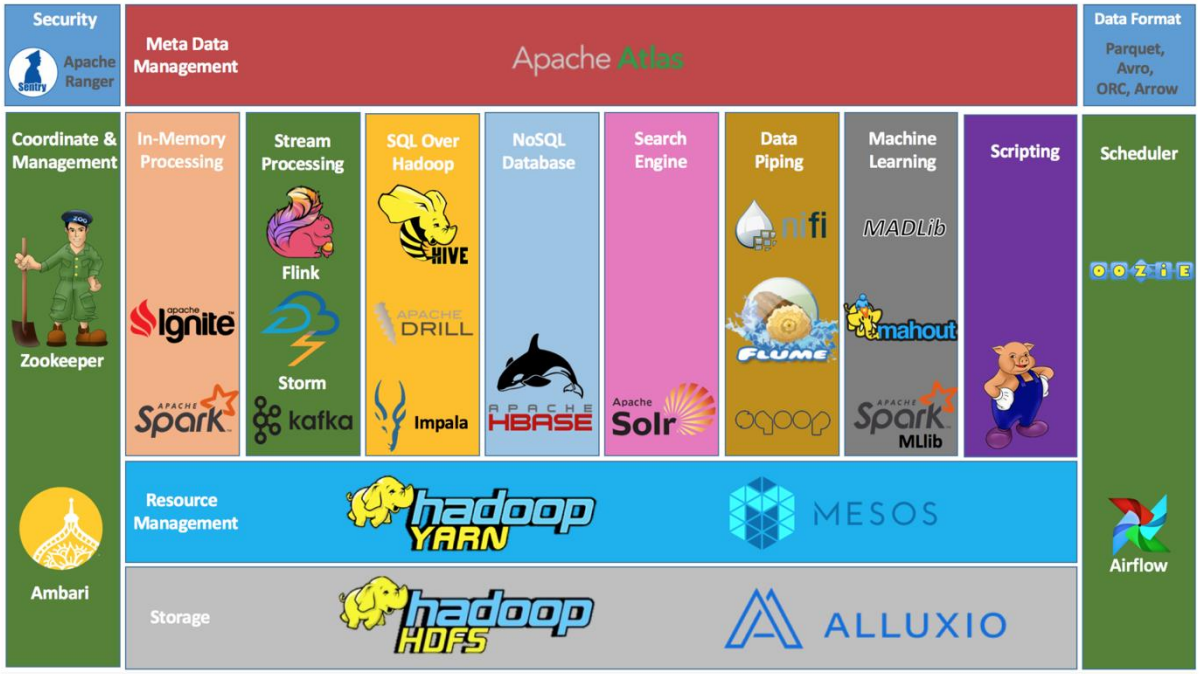
1.4. **Tecnologias *open-source* para a área de *Big Data***

Todos os conceitos apresentados nas subseções anteriores são estudados e aperfeiçoados até hoje, entretanto muitos destes conceitos já foram consolidados em algum *framework* utilizado tanto na academia, quanto na indústria.

Um dos primeiros *frameworks* na área de *Big Data*, e que serviu de base para a maioria dos *frameworks* que o sucederam, foi o Apache Hadoop, cuja base consiste em três ferramentas: um sistema de arquivos distribuídos (HDFS), uma biblioteca para implementação de processos distribuídos (MapReduce) e um gerenciador de recursos (YARN).

A modularidade deste *framework* garantiu que muitas ferramentas fossem desenvolvidas para substituir uma ou mais ferramentas de sua base (ex.: o Apache Pig[21] e o Apache Hive[22] criaram formas mais simples de descrever aplicações distribuídas, que interagem com o HDFS), e isto permitiu que novas funcionalidades fossem adicionadas a este *framework*, tornando ele um ecossistema completo para o desenvolvimento de aplicações para *Big Data*, como mostra a Figura 7.

Figura 7 – Ecossistema Hadoop.



Fonte: Du, 2018

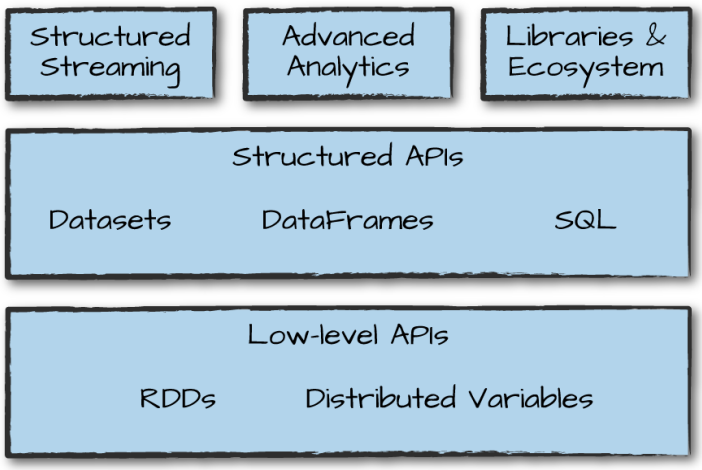
O restante desta subseção apresenta as ferramentas deste ecossistema, que são utilizadas nos testes realizados neste trabalho.

1.4.1. Apache Spark

O Apache Spark é um *framework* composto por bibliotecas de processamento distribuído de dados com diversas primitivas para análise de dados, como mostra a Figura 8.

[21] https://pig.apache.org/, acessado em março de 2020.
[22] https://hive.apache.org/, acessado em março de 2020.

Figura 8 – Primitivas de análise de dados do Apache Spark.



Fonte: Chambers & Zaharia, 2018

Sua forma de escalonar as etapas de transformação de dados de forma não-linear e utilizando a memória principal em conjunto com a memória secundária, proporciona um grande aumento de desempenho com relação ao Hadoop MapReduce [Zaharia *et al.* 2010].

Além disso, sua API estruturada (*Structured* API) também permite que ETLs sejam implementadas com primitivas semelhantes às encontradas no padrão ANSI SQL, sem impossibilitar o uso de funções definidas pelo usuário (UDF, *user-defined function*), o que facilita a adoção por usuários já acostumados com a sintaxe e com a funcionalidade dessas primitivas.

O Spark surgiu originalmente como um *framework* de processamento em *batches* de dados, entretanto, a inclusão da API de *streaming* do Spark sobre a *Structured* API, possibilita que os processos originais em *batches sejam* reutilizados com pequenas alterações, garantindo a rápida adoção a este tipo de ingestão.

Devido a sua base originária dos processos em *batches*, a API de *streaming* do Spark surge com o modelo de processamento em *micro-batches*, isto é, o mecanismo de *streaming* do Spark verifica periodicamente a fonte dos dados e executa uma consulta em *batches* sobre novos dados, que chegaram desde o último *batch*. Atualmente, há um esforço para diminuir a latência causada por este modelo de computação, com um modo experimental de processamento contínuo[23].

[23] https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html, acessado em março de 2020.

1.4.2. Apache Kafka

Um *streaming* pode ingerir dados de todo tipo de fonte, entretanto, muitos destes processos utilizam mensagerias como fonte, devido ao alto desempenho de escrita nestas mensagerias e a melhor distribuição de dados (mensagens) entre os consumidores interessados.

O Apache Kafka é uma mensageria que segue o modelo PubSub, onde um produtor de uma mensagem a persiste em algum compartimento de mensagens na mensageria, para então, um ou mais consumidores receberem esta mensagem.

Este compartimento no Kafka é conhecido como tópico e se difere das filas (que são outro tipo de compartimento utilizado por mensagerias), no sentido em que uma mensagem em uma fila é entregue apenas para um consumidor e, caso este não sinalize o recebimento da mensagem, haverá alguma forma de retentativa até que seja confirmada a entrega da mensagem.

Evitar a política de retentativa demanda que o usuário desta mensageria implemente uma política de tratamento das mensagens não recebidas, porém, permite um aumento considerável em sua taxa de transferência (*throughput*) de mensagens, o que a torna um melhor candidato para um grande volume de dados recebidos por segundo.

1.4.3. RocksDB

O RocksDB é um banco de dados não-relacional de chave-valor, que surgiu a partir do projeto do banco de dados LevelDB, comumente utilizado para armazenar metadados [Fisk 2019].

Uma característica chave do RocksDB é a forma em que os dados são gravados nos níveis do banco de dados, onde novos dados são gravados em uma tabela baseada em memória com um log de transações opcional no armazenamento persistente, o WAL (*Write-Ahead Log*).

Conforme essa tabela baseada em memória é preenchida, os dados são movidos para o próximo nível do banco de dados por um processo chamado compactação. Quando esse

nível é preenchido, os dados são migrados para o nível inferior novamente e assim por diante.

1.4.4. Apache Ignite

O Apache Ignite é um banco de dados não-relacional do tipo chave-valor, onde a memória do computador é utilizada como o meio principal para o armazenamento dos dados.

Para garantir a tolerância a falhas, os dados deste banco de dados podem ser replicados entre um subconjunto das instâncias do Ignite, aumentando a disponibilidade dos dados e diminuindo o tráfego de dados em tempo de consulta. É importante ressaltar que esse mecanismo duplica dados, contudo, diferente do problema estudado neste trabalho, isto não influencia o resultado das consultas, pois o Ignite processa um determinado dado replicado apenas em uma das instâncias.

Uma outra estratégia permitida para tornar o Ignite resiliente, é a movimentação dos dados menos acessados na memória para o disco, permitindo que o consumo de memória seja reduzido.

Em aplicações *Big Data*, o Apache Ignite pode ser utilizado como uma camada de *cache* para outros *frameworks* [Rovnyagin *et al.* 2020], como o Apache Spark e o Apache Flink. Contudo, este banco possui uma API compatível com operadores SQL, o que facilita o desenvolvimento de ETL que processam dados diretamente neste banco de dados.

Este banco também possui métodos nativos para interoperação com o Spark e o Flink, permitindo por exemplo, que *dataframes* Spark interoperem com dados no Ignite. Neste caso, as operações sobre os dados do Ignite são mapeadas para operações diretamente sobre os dados no banco e, caso seja necessária uma junção entre os dados de ambas as aplicações, somente os dados filtrados do Ignite serão projetados para o destino final.

1.4.5. Apache Hudi

Em geral, os dados de um repositório não são alterados ou apagados devido à alta complexidade de se realizar estas operações conforme o volume dos dados cresce e quando não há um identificador claro, que auxilie o acesso dos registros que se deseja alterar.

Neste caso, a solução inicial é criar uma ETL, que consuma todos os dados da coleção que se deseja atualizar, aplicando as alterações que se deseja e persistindo os resultados no mesmo destino, sendo que para grandes volumes isto se torna inviável quando se demanda o acesso para análise dos dados já atualizados em tempo NRT.

Para estes casos, foi desenvolvido o Apache Hudi[24] que permite o controle dos dados em um repositório a nível de registro. Isto é possível graças ao uso de filtros de Bloom [Bloom 1970] nos metadados de um arquivo, o que permite de forma probabilística verificar se um dado pertence a um determinado arquivo (com garantia exata caso o filtro retorne que o dado não pertence ao arquivo).

Essa rápida validação da existência de um dado em um conjunto de arquivos permite que o Hudi disponibilize primitivas de atualização e remoção de dados em arquivos com base em versões, isto é, uma nova ocorrência de um dado já conhecido (ou sua remoção) irá resultar em uma nova linha no arquivo em que se encontra, com um número de revisão incrementado.

Para otimizar o processo de escrita, o método de inserção de dados pode ser configurado pela aplicação, sendo os seguintes métodos disponibilizados pelo Hudi:

- ***UPSERT*:** método padrão do Hudi, que valida a existência dos dados antes de inseri-los para possibilitar a criação de novas versões de um dado;
- ***INSERT***: método que não valida a existência dos dados para novas versões, mas que utiliza esta validação para eliminar possíveis duplicatas, se configurado pela aplicação;
- ***BULKINSERT***: método semelhante ao de *insert*, mas que reorganiza os dados entre os arquivos, com o propósito de aproveitar a compactação dos dados do formato Parquet, para gerar o menor tamanho de arquivos possível, ao custo de ser o método mais lento entre os três.

Outra otimização do Hudi é a sua capacidade de inserir novos dados em arquivos já persistidos. Isso ocorre com o objetivo de evitar que muitos arquivos pequenos sejam

[24] https://hudi.apache.org/, acessado em março de 2020.

criados, o que dificultará futuras análises sobre esses dados em decorrência de uma quantidade maior de metadados persistidos por arquivo.

1.5. **Soluções de *Big Data* em nuvem**

Um dos principais problemas no uso das ferramentas de *Big Data* é a configuração do ambiente de execução destas ferramentas, algo que demanda um conhecimento aprofundado dos componentes que compõem cada ferramenta. Por exemplo, a escolha da quantidade de recursos de *hardware* necessária por máquina, a configuração do ambiente para o melhor uso do *hardware*, e a configuração de tolerância a falhas.

Esta razão é o principal motivador para o surgimento de empresas como Cloudera[25] e Hortonworks[26], que vendem soluções pré-configuradas de ferramentas Hadoop para atender aos requisitos de dados de uma empresa.

Outro facilitador do uso dessas ferramentas é o advento da computação em nuvem que permite ao desenvolvedor de aplicações utilizar e escalar recursos de *hardware* sob demanda e altamente configuráveis (ex.: um conjunto de máquinas com quaisquer quantidades de processadores, memória principal e secundária). Após o uso destes recursos, o desenvolvedor os descarta, sendo comumente cobrado apenas pelo tempo de uso e pelos recursos escolhidos.

Além de máquinas, esses serviços de nuvem também possuem ferramentas que permitem o provisionamento facilitado de bancos de dados, mensagerias e sistemas de arquivos distribuídos, assim como *clusters* com as ferramentas Hadoop já configuradas, como o AWS EMR[27] (Elastic MapReduce) e o Google Cloud Dataproc[28].

Um grande benefício de se utilizar os serviços de nuvem é a SLA garantida pela maioria das empresas que prestam esses serviços, algo que seria a maior dificuldade de um usuário caso este decida por uma solução própria. Por exemplo, soluções de armazenamento baseada em objetos, como o Google Cloud Storagee o AWS S3, garantem que um dado armazenado possua disponibilidade anual de 99,999999999%[29].

---

[25] https://www.cloudera.com/, acessado em março de 2020.
[26] https://www.hortonworks.com/, acessado em março de 2020.
[27] https://aws.amazon.com/emr/, acessado em março de 2020.
[28] https://cloud.google.com/dataproc/, acessado em março de 2020.
[29] http://cloud.google.com/storage/docs/faq#replication, acessado em março de 2020.

Estes serviços de armazenamento são bons exemplos, pois estes são comumente utilizados para substituir o HDFS como sistema de arquivo por remover a dificuldade de planejar *clusters* tolerantes à falha, com alta disponibilidade dos dados, pela adaptação de ferramentas como Apache Hive, para suportarem nativamente estes sistemas de arquivos.

O caráter dispensável das máquinas dos serviços de nuvem também permite que o primeiro passo de processos em *batch* seja o de provisionar um *cluster* para a execução de uma ETL e o último passo seja o descarte desse *cluster*, isso motiva inclusive o uso de vários *clusters* com menos recursos de *hardware*, mas que sirvam apenas para a execução de pequenos conjuntos de ETLs.

2. METODOLOGIA

Este capítulo discute a natureza dos dados utilizados nesta dissertação, os requisitos do domínio dos dados e a metodologia empregada pelos testes e a definição de cada teste realizado.

2.1. Descrição do domínio

Como exemplo de um processo de *streaming*, este trabalho utiliza dados reais, extraídos de uma empresa de classificados brasileira que migrou um processo em *batch* de extração de dados de um banco de dados relacional para um novo processo *streaming*, extraindo dados de uma mensageria.

Essa migração foi motivada devido à fonte de dados começar a perder a constância da quantidade e o tamanho total de eventos gerados por dia para um subsistema específico, o que ocorreu devido ao surgimento de novos tipos de testes gerados com base na experiência do usuário no site e no ciclo de vida dos anúncios da plataforma.

É possível observar, na Tabela 1, que a média de eventos diários registrados em 2015 era de, aproximadamente 3 milhões, ocupando em média 1,5 GB de espaço diariamente. Nos demais anos, esses valores aumentaram consideravelmente, chegando a registrar quase 8GB de espaço (2018).

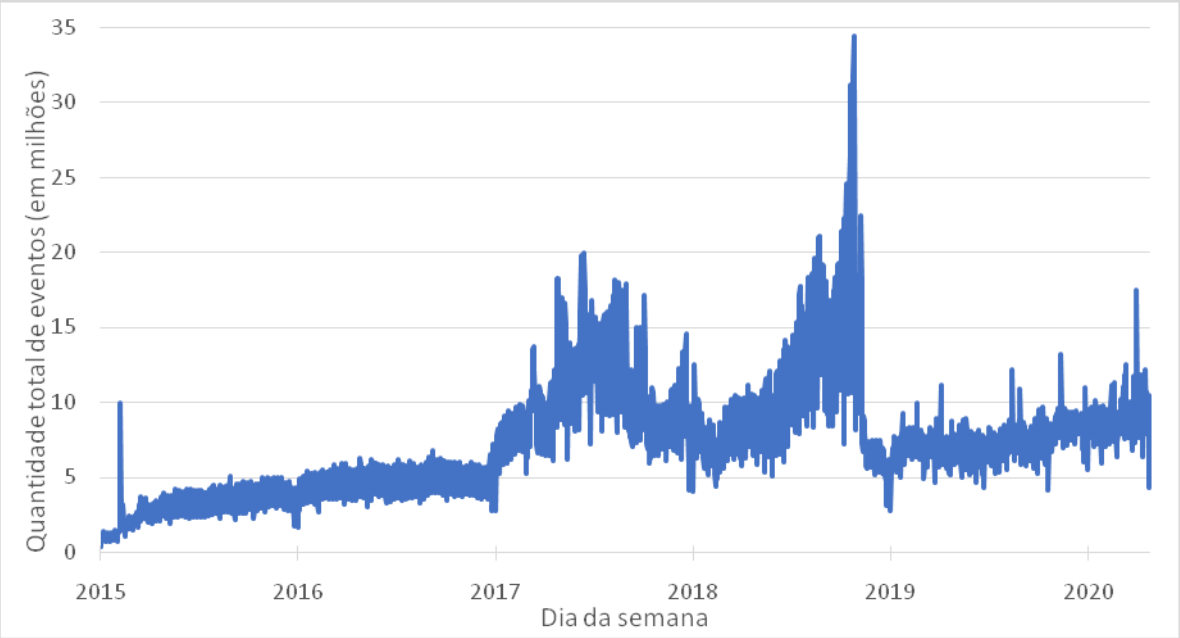
Tabela 1 – Tamanho e quantidade média de eventos diários por ano

Ano	Tamanho (em GBs)	Quantidade (em milhões)
2015	1,545604	3,183938
2016	2,92892	4,93282
2017	6,532226	10,80746
2018	7,922492	11,09315
2019	4,756737	7,428618
2020	6,32158	9,056052

Houve um crescimento na quantidade de eventos devido a popularização da plataforma, tendo picos de eventos entre 2017 e 2018, como mostram as Figura 9 e Figura 10. Após uma revisão nos produtores de eventos no final de 2018, houve uma estabilização na quantidade e no tamanho dos eventos gerados por dia, com picos mais ocasionais, como

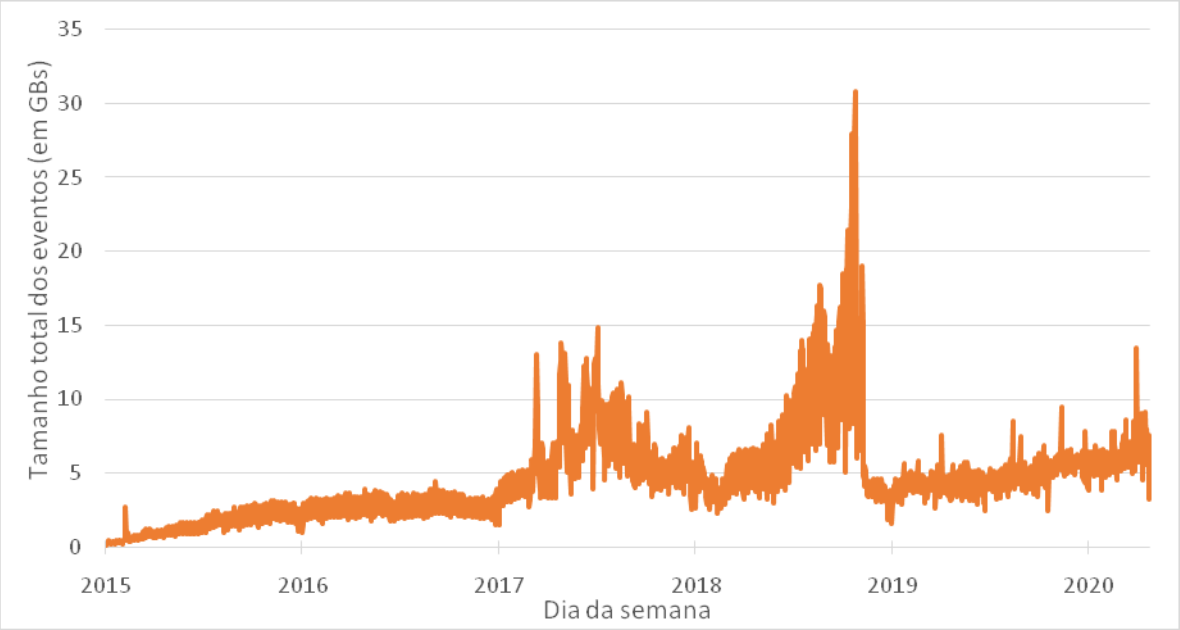
no final de Março de 2020, quando os estados brasileiros começaram a decretar quarentena devido a pandemia de Coronavirus.

Figura 9 – Quantidade de eventos por dia, em milhões.



Fonte: O autor, 2021

Figura 10 – Tamanho ocupado por eventos, por dia, em GB.



Fonte: O autor, 2021

Além disso, devido a algumas revisões na lógica de geração de eventos e na implementação de políticas de reenvio de mensagem da mensageria, o processo de extração de dados começou a identificar várias duplicatas de dados na fonte. É importante ressaltar

que o processo legado, embora tenha desempenho abaixo do esperado, garante não haver duplicatas em tempo de consulta.

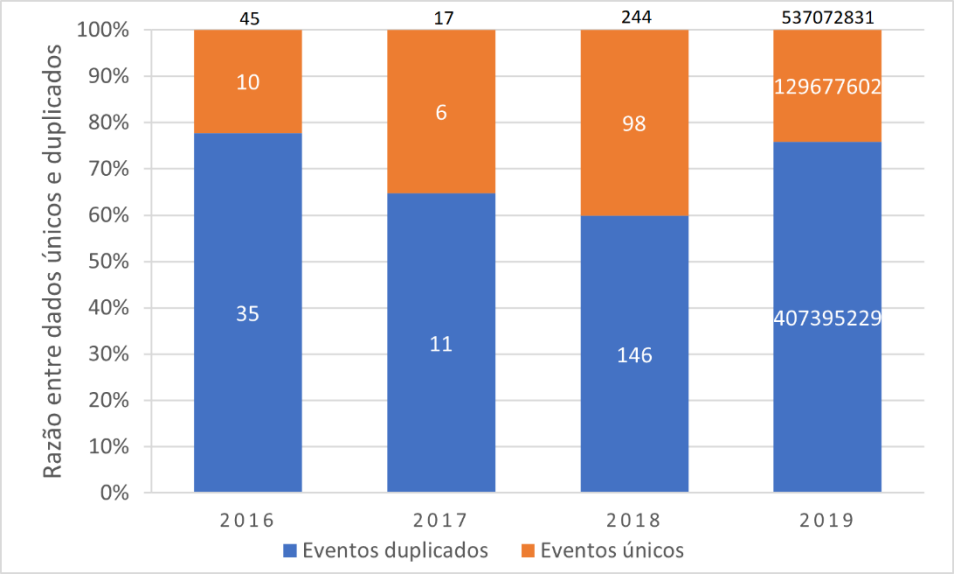
Outro fator que pode ocorrer é a geração de um evento com uma taxa de atraso muito elevado, ocorrendo casos de produção de um evento duplicado com até um ano de diferença entre sua data de criação e a data em que de fato ocorreu o evento.

Esses fatores dificultam a implementação de uma solução que cumpra o tempo de SLA de dez minutos apenas utilizando janelas de tempo, impedindo a duplicação de registros, o que motiva o estudo e a implementação de uma arquitetura de alto desempenho e tolerante a falhas.

Para análise das soluções avaliadas neste trabalho, é utilizada uma massa de dados de aproximadamente 1,6 *terabytes*, correspondendo a um período de 14 (quatorze) dias de anúncios do mês de novembro de 2019.

Entretanto, como é possível observar na Figura 11, mesmo tendo sido publicadas na mensageria durante este período, alguns dados correspondiam a eventos com anos de atraso, que não poderiam ser descartados durante a ingestão pela possibilidade de estes não existirem no repositório de dados final.

Figura 11 – Razão entre a quantidade de evento únicos e os eventos duplicados.



Fonte: O autor, 2021

2.2. **Desafios**

O grande problema de se implementar um mecanismo de deduplicação em tempo real é a necessidade de consultar registros históricos quando novos registros são processados. Tal fato acarreta o aumento do tempo de processamento do *streaming* para cada registro processado conforme o volume histórico cresce, eventualmente causando o descumprimento do tempo de SLA.

O mecanismo de persistência de estado baseado em janelas de tempo de processos *streaming* reduz a quantidade de registros consultados com base no conhecimento de até quando é possível que uma duplicata seja ingerida pelo processo. Entretanto, alguns problemas desafiadores podem surgir com essa abordagem:

- **Alto consumo de recursos pela janela:** o processo necessitará armazenar os registros da janela enquanto ela for necessária, podendo eventualmente esgotar os recursos do *cluster* hospedeiro, terminando o *streaming* abruptamente, e podendo inutilizar o *cluster*;
- **Processo não tolerante a falhas:** para retomar um *streaming* que falhou, é necessário que este recupere o estado da janela antes de sua falha, assim como reinicie a ingestão a partir do último dado consumido. Caso não haja algum mecanismo de salvamento de progresso, também conhecido como *checkpoint*, o estado da janela poderá ser perdido na nova execução;
    - Também é necessário considerar que os recursos utilizados em um provedor de *cloud* são considerados efêmeros[30,31], ou seja, os *checkpoints* do *streaming* devem ser persistidos fora do armazenamento local do *cluster* para que a aplicação retome do ponto de parada em caso de perda ou descarte do *cluster*.
- **Inflexibilidade do processo a inconsistências na ingestão:** caso a regra de negócio que garante até quando uma duplicata apareça mude, ou caso ocorra um atraso na fonte dos dados, o período da janela será comprometido, podendo ocasionar duplicatas.

[30] https://aws.amazon.com/blogs/big-data/best-practices-for-running-apache-spark-applications-using-amazon-ec2-spot-instances-with-amazon-emr/, acessado em março de 2020.
[31] A maioria das instâncias de computação de provedores como a AWS e o Google Cloud são projetadas para que arquitetos de infraestrutura aumentem ou diminuam o número de máquinas utilizadas a qualquer momento para escalar de acordo com o volume de trabalho, podendo o número de instâncias de *cluster* chegar a zero, caso não haja mais trabalho em um determinado momento.

2.3. **Requisitos**

Baseado na descrição do domínio e nos desafios desse tipo de aplicação, é possível listar os possíveis requisitos do processamento *streaming*:

- **R1:** Reduzir o uso de memória RAM para armazenar estados entre *micro-batches*.

Manter o uso de recursos reduzidos irá possibilitar o uso de instâncias mais econômicas da AWS ou o aproveitamento de um *cluster* para a execução de mais aplicações *streaming* paralelamente.

Este fator também é importante devido a possibilidade de um dado duplicado aparecer até um ano após sua primeira aparição, visto que caso a solução de janelas seja a mais indicada para qualquer caso, ela poderá armazenar bilhões de chaves em memória.

É importante, também, enfatizar que, durante os testes realizados para medir o atendimento a este requisito, cada repetição de cada teste foi executada em um *cluster* isolado para evitar que uma possível quantidade de memória não desalocada pudesse interferir na medida do consumo de recursos.

- **R2:** Reduzir o tempo de execução dos *micro-batches*.

O tempo de SLA para as análises sobre os dados finais é de 10 minutos, ou seja, a partir do momento de sua ingestão, o dado processado deve estar disponível em até 10 minutos.

Além disso, reduzir o tempo de execução dos *micro-batches* é importante para precaver que possíveis picos (*bursts*) descumpram esse tempo de SLA.

- **R3:** Garantir que seja possível interromper e reiniciar o *streaming* sem que haja a ocorrência de duplicatas ou perda de dados

Devido a possibilidade de haver algum erro, como falta de memória, durante a execução de uma aplicação *streaming*, é necessário que seja possível reiniciar esta aplicação, com a garantia de que as regras do domínio ainda sejam cumpridas.

Para evitar a duplicidade de dados, é necessário que a aplicação tenha alguma forma de recuperar a informação de quais chaves de mensagens ela já consumiu, seja pela recuperação completa do estado da execução anterior, ou por meio de um repositório de dados externo e de acesso rápido.

Garantir essa tolerância a falhas também permite a execução desse *streaming* ser mais econômica, visto que isso possibilita o uso de instâncias efêmeras, conhecidas na

AWS como instâncias *spot*[32]. Essas instâncias possuem tempo de vida curto, sendo excluídas de forma imprevisível.

- **R4:** Garantir que o tempo de consulta aos dados processados seja igual ou menor ao obtido pelo banco de dados atual

A principal motivação de realizar a deduplicação dos dados durante o processamento é cumprir a regra de domínio que garante que os dados analisados serão únicos, não tornando necessária a reescrita de consultas legadas para incluir essa deduplicação.

Evitar a adição desta lógica às consultas existentes manterá o tempo de execução que havia com o processo de ingestão original. Entretanto, caso o *streaming* produza arquivos pequenos, poderá haver um aumento considerável no tempo de execução das consultas, mesmo sem a lógica de deduplicação, visto que haverá uma quantidade maior de metadados a serem consumidos por arquivo.

Da mesma forma, caso a quantidade de arquivos gerados seja menor, e com tamanho maior, poderá haver um ganho de desempenho, comparado ao processo *batch* legado.

2.4. **Arquitetura**

Esta seção discute a arquitetura do *pipeline* utilizado pelos testes realizados, detalhando a infraestrutura e as ferramentas utilizadas em cada camada.

Os testes realizados neste trabalho foram executados nas máquinas disponíveis pela AWS, sendo as especificações das máquinas utilizadas nestes testes detalhadas na Tabela 2.

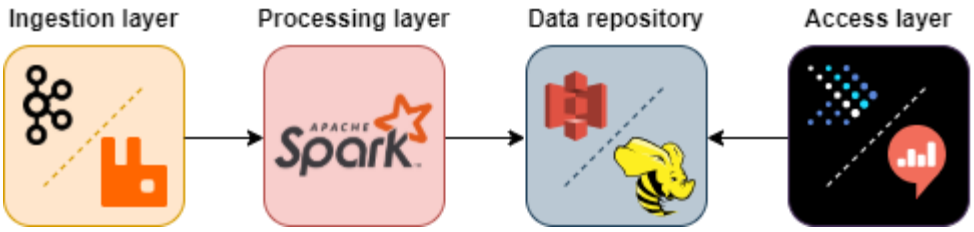
Tabela 2 – Especificação das máquinas da AWS utilizadas nos testes

Nome	CPU	Número de *cores*	Quantidade de memória RAM
`m5.large`	Intel Xeon 8175	2	8 GB
`m5.xlarge`	Intel Xeon 8175	4	16 GB
`r5.xlarge`	Intel Xeon 8175	4	32 GB
`r5.2xlarge`	Intel Xeon 8175	8	64 GB

[32] https://aws.amazon.com/ec2/spot/, acessado em março de 2020.

Conforme pode ser visto na Figura 12, as camadas usadas são: ingestão (*Ingestion layer*), processamento (*Processing layer*), repositório de dados (*Data repository*) e acesso aos dados (*Access layer*). Cada uma dessas camadas é detalhada em seguida.

Figura 12 – Visão geral do *pipeline* de dados

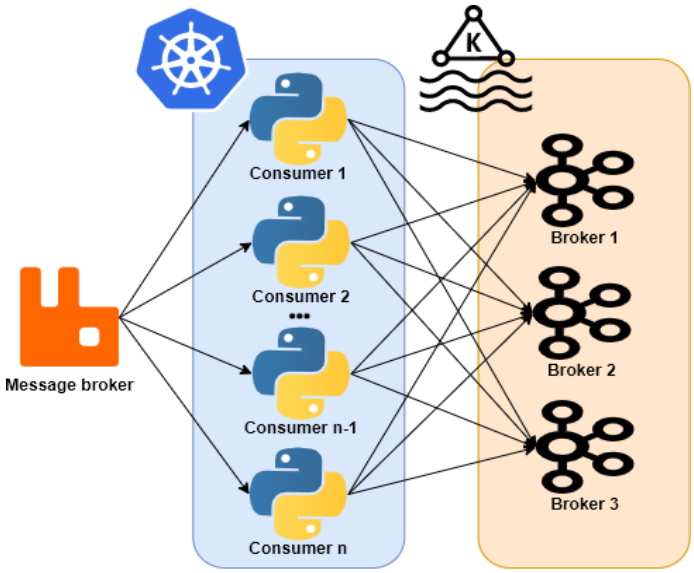


Fonte: O autor, 2021

2.4.1. Camada de ingestão

Para consumir os registros da mensageria, implementada com o RabbitMQ[33], foram instanciados em um *cluster* Kubernetes[34],consumidores descritos em Python[35]com objetivo de repassar os registros em *batches* para um *cluster* Apache Kafka provisionado para este *pipeline*, como mostra a Figura 13.

Figura 13 – Arquitetura da camada de ingestão



Fonte: O autor, 2021

[33] https://www.rabbitmq.com/, acessado em março de 2020.
[34] https://www.kubernetes.io/, acessado em março de 2020.
[35] https://www.python.org/, acessado em março de 2020.

A motivação de persistir os registros em um *cluster* Kafka antes do processamento, ao invés de persisti-los diretamente no repositório de dados ou de processá-los diretamente é justificada pelo fato do Apache Kafka mitigar os seguintes pontos:

- A tolerância a falhas nativa do Apache Kafka em que as mensagens não são excluídas após terem sido consumidas pela primeira vez[36], possibilitando que haja uma releitura de mensagens perdidas em caso de falha na camada de processamento (Requisito R3);
- Através do particionamento de um tópico, é possível controlar o paralelismo da aplicação que consumirá as mensagens, já que a aplicação poderá instanciar até um máximo de n consumidores, sendo n o número de partições de um tópico;
- Através do uso de chaves de mensagens, é possível enviar mensagens com a mesma chave utilizada para deduplicação para um mesmo consumidor, o que evitará a duplicidade de dados entre consumidores;
- Grande parte das ferramentas para implementação de processos em *streaming* possuem primitivas nativas de leitura de dados em tempo real originárias do Apache Kafka.

Além disso, a AWS provê um serviço de *cluster* Apache Kafka autogerenciado, o MSK[37] (*Managed Streaming for Apache Kafka*), o que simplifica o uso da ferramenta sem a necessidade de se gerenciar um *cluster* para o Apache Kafka e outro para o Apache Zookeeper[38], que possuem componentes críticos que devem ser monitorados com atenção.

Foram escolhidas três instâncias AWS do tipo[39] `m5.large` para o *cluster* Apache Kafka, pois suas operações são *IO-bound*, isto é, possuem mais carga de entrada e saída de dados do que de processamento (*CPU-bound*).

Para cada instância, foi alocado um SSD (*Solid State Drive*, isto é, disco de estado sólido) com 2 TB de armazenamento, sendo que cada um desses discos teve 81,2% (isto é, 1,624 TB) de sua capacidade utilizada pela massa de dados de testes deste trabalho.

Este uso elevado de disco é justificado por ter sido configurado uma política de exclusão de mensagens de quatorze dias, a mesma quantidade de dias analisados nos testes, e por cada partição do tópico ter um fator de replicação igual a três. Essa replicação permite

[36] Ao invés da remoção imediata, é implementado uma política de remoção de mensagens baseada no tempo de vida da mensagem ou no espaço de armazenamento consumido por um tópico.
[37] https://aws.amazon.com/msk/, acessado em março de 2020.
[38] https://zookeeper.apache.org/, acessado em março de 2020.
[39] https://aws.amazon.com/ec2/instance-types/, acessado em março de 2020.

que todos os nós do *cluster* tenham todas as mensagens no Apache Kafka, evitando-se a perda de grandes blocos de mensagens em caso de perda de algum nó.

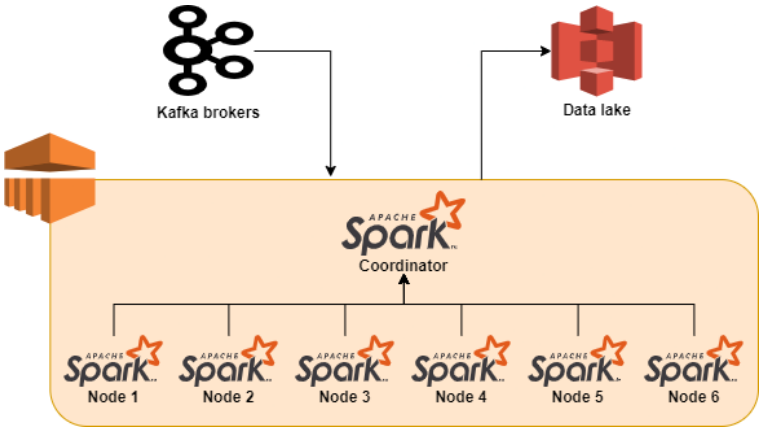
Todas as mensagens consumidas referem-se a apenas um tópico, particionado em trinta pedaços. Esse número de partições foi escolhido para maior paralelismo e para equilibrar o número de consumidores instanciados pelo *streaming* para cada nó no *cluster* da camada de processamento.

É também importante enfatizar que a leitura das partições do tópico é centralizada na partição principal, e não em uma réplica, o que permite o Apache Kafka também distribuir a quantidade de requisições entre os nós do *cluster*. Por exemplo, neste *cluster*, cada máquina teve até dez partições sendo consumidas simultaneamente.

2.4.2. Camada de processamento

Esta camada é responsável pela execução do *streaming*, sendo esta camada, durante os testes, utilizada apenas para a execução de um teste por vez, ou seja, todos os recursos do *cluster* são utilizados exclusivamente para apenas uma aplicação *streaming*, como mostra a Figura 14.

Figura 14 – Arquitetura da camada de processamento



Fonte: O autor, 2021

A ETL executada nesta camada pelos testes realizados neste trabalho pode ser resumida pela Figura 15, seguindo os seguintes passos:

1. Consumir as mensagens, em formato JSON, do Apache Kafka (*Ingestion Layer*);

2. Deduplicar as mensagens com base em suas chaves consumidas e nas chaves persistidas no repositório de dados;
3. Extrair o valor da mensagem (em formato JSON) aplicando um *schema*;
4. Persistir os dados no repositório de dados em um formato compactado e otimizado para leitura por coluna (neste caso, o formato utilizado foi o Apache Parquet).

Figura 15 – ETL de deduplicação executada na camada de processamento

Fonte: O autor, 2021

Para cada teste realizado, altera-se a lógica de deduplicação definida no passo 2, variando-se as funções de deduplicação, as ferramentas e os componentes de infraestruturas auxiliares (essas diferenças são detalhadas na seção de requisitos).

Para criar as instâncias do *cluster* de processamento foi utilizado o AWS EMR, que provisiona todas as ferramentas de ETL que foram utilizadas nos testes (com exceção de alguns bancos de dados auxiliares), retirando a responsabilidade de configurar todos os detalhes de cada ferramenta e agilizando o início de cada teste.

Um *cluster* EMR, assim como a maioria das ferramentas de *Big Data* espera uma distribuição de máquinas em *cluster* no modelo *master/slave* (mestre e escravo), isto é, é necessário ao menos uma máquina que deve atuar como coordenador de um conjunto de máquinas que de fato executam a ETL.

Para este *cluster* é utilizado uma máquina do tipo `r5.xlarge` para *master*, com quatro vCPU's e 32 GB de memória RAM, para as máquinas *slaves* são utilizadas seis máquinas do tipo `r5.2xlarge`, que dobram a quantidade de vCPU's e de memória RAM do tipo `r5.xlarge`, totalizando 64 vCPU's e 384 GB de memória RAM.

Essas máquinas foram escolhidas devido ao alto consumo de memória RAM utilizado durante a execução da ETL, especialmente quando é necessário persistir as chaves utilizadas para deduplicação entre momentos do *streaming*.

Devido ao uso exclusivo do *cluster* por cada teste e as trinta partições do tópico do Apache Kafka, as aplicações de *streaming* puderam instanciar cinco consumidores em cada máquina *slave* do *cluster* com um núcleo de processamento e até 15 GB dedicados a execução da ETL sobre os dados consumidos.

Como os resultados foram persistidos em um repositório externo, não foi necessário alocar um SSD de alta capacidade como na camada de ingestão, entretanto, a persistência

dos *checkpoints*, produzidos ao fim do processamento de cada *micro-batch*, foi feita localmente para evitar qualquer latência no acesso a estes dados. Contudo, quando um novo *checkpoint* era gerado, também se iniciava um processo em paralelo para *backup* deste dado em um repositório externo.

2.4.3. Repositório de dados e a camada de acesso

Depois que uma mensagem é processada, ela é persistida em um repositório de dados para consulta e análise futura. O sistema de armazenamento deste repositório escolhido é o AWS S3, devido a integração madura do sistema às ferramentas utilizadas para a ETL e para a consulta posterior dos dados e a transparência da infraestrutura, não sendo necessário gerenciá-la diretamente.

Com relação a tolerância a falhas, é importante ressaltar que o próprio serviço da AWS garante que os dados não serão perdidos, de forma transparente, e sem a necessidade do cliente replicar os dados em mais de um nó, e de forma a não degradar o desempenho do acesso aos dados. Com isto, o repositório de dados atende o requisito R3 naturalmente, e evita que uma duplicação de dados manual possa interferir na garantia da unicidade dos dados no momento do acesso a estes.

Os dados armazenados no S3 são persistidos pela ETL de forma particionada pela data da criação do evento, isto permite que consultas aos dados deste repositório não necessitem acessar todos os objetos persistidos, caso a consulta apenas necessite de um subconjunto de datas, o que otimiza o consumo dos dados e diminui o tempo de execução da consulta (Requisito R4).

Entretanto, por mais que os dados persistidos neste repositório possuam a definição de seu *schema* no metadado dos objetos, o motor de consultas utilizado, conhecido como Apache Presto[40], não consegue acessar os dados na busca se ele não for direcionado ao caminho completo do objeto desejado. Este fator dificulta a escrita de consultas que lidem com mais de um objeto, além de tornar necessário que o usuário tenha conhecimento de como o repositório de dados está organizado.

[40] https://www.prestodb.io/, acessado em março de 2020.

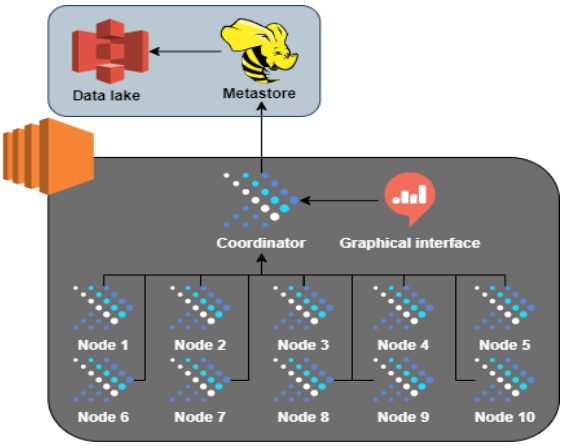
Para solucionar essa dificuldade, e criar a abstração para o usuário de que este está interagindo com um banco de dados relacional tradicional, isto é, com abstrações de *schemas*, tabelas e colunas, é utilizado o Apache Hive.

O Apache Hive foi criado justamente com a finalidade de criar essa abstração para os arquivos armazenados no HDFS (e posteriormente evoluindo para dar suporte a outros sistemas de dados, como o AWS S3), e permitir a execução de consultas em formato SQL sobre esta abstração.

Por mais que o Apache Hive por si só possa ser utilizado como um motor de consultas da mesma forma que o Apache Presto, seu desempenho é inferior comparado ao Apache Presto, além de não possuir a mesma quantidade de integrações com outros repositórios de dados que o Apache Presto possui, como com bancos de dados relacionais e NoSQL, o Apache Kafka etc.

Nesta arquitetura, ilustrada na Figura 16, o Apache Hive é utilizado apenas como um repositório de metadados, mais conhecido como *metastore*, permitindo a consulta do repositório de dados pelo Apache Presto.

Figura 16 – Arquitetura da camada de acesso ao *data lake*



Fonte: O autor, 2021

As consultas realizadas pelo Apache Presto, assim como pelo Apache Hive, são realizadas apenas por linha de comando ou por chamada de API, o que pode dificultar a consulta de um usuário que não saiba utilizar nenhum destes dois métodos de acesso.

Por este motivo, foi utilizado o Redash[41] como uma aplicação de interface gráfica para que o usuário tenha um acesso mais familiar a uma IDE (*Integrated Development Environment*), além de poder criar painéis de métricas conhecidos como *dashboards* simples.

[41] https://redash.io/, acessado em março de 2020.

Quanto a infraestrutura utilizada por estes dois componentes: o Apache Presto utiliza um *cluster* de dez máquinas `r5.2xlarge` (8 vCPU's/64 GB) coordenadas por uma `r5.xlarge` (4 vCPU's/32 GB); já o Redash utiliza apenas uma máquina `m5.xlarge` (4 vCPU's/16 GB).

2.5. **Cenários de testes realizados**

A seguir, são apresentados os cenários de testes realizados neste trabalho, detalhando as decisões específicas de cada teste, assim como as configurações utilizadas.

- **T1:** Deduplicação com o operador *distinct* do Apache Spark

O método mais simples para deduplicação de dados no Spark é o operador *distinct*, que remove as linhas duplicadas de um *dataframe* a partir da comparação coluna a coluna.

A comparação entre duas linhas irá ser interrompida no primeiro par de colunas diferentes, entretanto, para *dataframes* com múltiplas colunas, como nas mensagens processadas nos testes deste trabalho, as comparações poderão ser demoradas caso a diferença for encontrada somente nas últimas colunas comparadas.

Também é importante ressaltar que, por ser necessário ter conhecimento de todas as colunas de cada linha do *dataframe* para remover duplicatas, toda a linha deve ser armazenada no estado do *streaming* para que possíveis duplicatas de futuros *micro-batches* não sejam persistidos no repositório de dados, o que resultará em um uso elevado de memória pela aplicação, impactando no requisito R1.

- **T2:** Deduplicação com o operador *dropDuplicates* do Apache Spark

O operador *dropDuplicates* permite remover duplicatas de um *dataframe* apenas pela comparação de um subconjunto de colunas do *dataframe*, sendo este subconjunto denominado como as chaves do *dataframe*.

Devido ao uso de menos colunas comparado ao operador *distinct*, o tempo de execução do *dropDuplicates* terá um tempo de execução consideravelmente inferior ao *distinct* para *dataframes* com muitas colunas e muitas linhas. Isso também implicará no menor uso de memória, pois nesta solução, apenas as chaves do *dataframe* precisão ser persistidas entre *micro-batches*.

Entretanto, este operador só pode ser utilizado caso o domínio da aplicação declare que estas chaves existam para todo dado processado pela aplicação. Logo, o uso deste operador nem sempre será uma alternativa ao operador *distinct*.

- **T3:** Deduplicação com o operador *dropDuplicates* do Apache Spark com persistência de estado no RocksDB

Uma deficiência do Apache Spark é a persistência apenas em memória do estado do *streaming*, o que implica na perda destes dados no caso de uma falha parar a execução da aplicação.

O RocksDB[42] é um banco de dados chave-valor, que pode ser utilizado como alternativa ao comportamento padrão do Spark. Neste caso, ao final de cada *micro-batch*, o Spark persistirá os dados do estado neste banco de dados ao invés do espaço de memória da aplicação. Isto permite que a aplicação recupere o estado de uma execução anterior, caso a aplicação seja reiniciada.

Para evitar que os dados trafeguem pela rede, o RocksDB foi instanciado em cada máquina *slave*, permitindo que a aplicação obtivesse um desempenho superior por evitar o aumento da latência com uma conexão externa ao *cluster*.

- **T4:** Deduplicação com o Apache Ignite.

O uso do banco de dados chave-valor Apache Ignite é uma solução interessante ao problema de deduplicação em *streaming*. Isto porque este banco de dados possui integração nativa com o Spark, tendo seus comandos SQL interoperáveis com as primitivas SQL dos *dataframes* Spark.

Isto permite que operações sobre os dados de um *dataframe*, armazenado no Ignite, aconteça no espaço de memória do banco de dados. A vantagem desta característica é evidenciada quando há a necessidade de realizar a junção entre um *dataframe* Spark e um *dataframe* Ignite. Neste caso, apenas as linhas do Ignite que atendem aos critérios da junção serão projetadas para o Spark, diminuindo assim o tráfego de dados entre os espaços de memória.

Devido a replicação dos dados entre nós de um *cluster* Ignite, este banco de dados também previne que um conjunto de linhas do estado de um *streaming* seja perdido, caso um dos nós do *cluster* esteja indisponível, tornando a aplicação tolerante a falha de *software* e *hardware*.

[42] https://www.rocksdb.org/, acessado em março de 2020.

Para este teste, foi configurado o uso do armazenamento em disco para conter todas as linhas consumidas por todos os *micro-batches*, e para manter os dados mais acessados recentemente em memória.

Para permitir o acesso rápido aos dados e evitar o tráfego de rede entre nós, as instâncias do Ignite foram inicializadas nos nós *slave* do Spark. Além disso, a replicação também foi configurada para manter uma cópia de todas as linhas do estado em todas as instâncias do Ignite.

Como o Ignite possui as chaves de *micro-batches* já processados, não foi necessário utilizar nenhum operador nativo do Spark para realizar a deduplicação, sendo esta operação realizada por uma consulta SQL utilizando a sintaxe de LEFT ANTI JOIN do Spark, interoperável com o Ignite.

Isto permitiu a junção do *dataframe* Spark do *micro-batch* com as chaves persistidas no Ignite, para recuperar as linhas novas do *micro-batch* corrente, sem a necessidade do uso de memória do Spark para armazenar o estado da aplicação.

- **T5:** Deduplicação com o Apache Hudi

O formato de arquivos do Apache Hudi permite à aplicação que produz arquivos neste formato determinar o número de versões de um dado que será persistido. Neste teste, a aplicação foi configurada para persistir apenas uma versão, eliminando a possibilidade de dados duplicados.

Como não há atualizações ou exclusões neste teste, os dados foram persistidos utilizando o método *insert*, com validação de duplicatas, em arquivos com tamanho máximo de 128MB (configuração padrão do Hudi), reduzindo a quantidade de arquivos consultados.

3. RESULTADOS

Este capítulo apresenta os resultados obtidos pelos testes descritos na seção 2.5, com base nos desafios e requisitos detalhados nas seções 2.2 e 2.3 respectivamente. Entretanto, é importante ressaltar que:

- Os dados apresentados neste capítulo são a média de cinco execuções de cada teste;
- Os gráficos 17, 18 e 19 não incluem o desvio padrão dos testes, para destacar a diferença entre cada *micro-batch*. Os gráficos de desvio padrão são apresentados no Apêndice deste trabalho;
- Para evitar algum acúmulo de memória entre testes, um *cluster* da camada de processamento foi provisionado para cada execução de cada teste, sendo este excluído ao fim do *streaming*;
- Também foram executados testes (chamados de *baseline*) com apenas a lógica de ler mensagens do Kafka e persistir no S3, com o objetivo de entender o quanto a lógica de deduplicação impacta nas métricas dos outros testes.

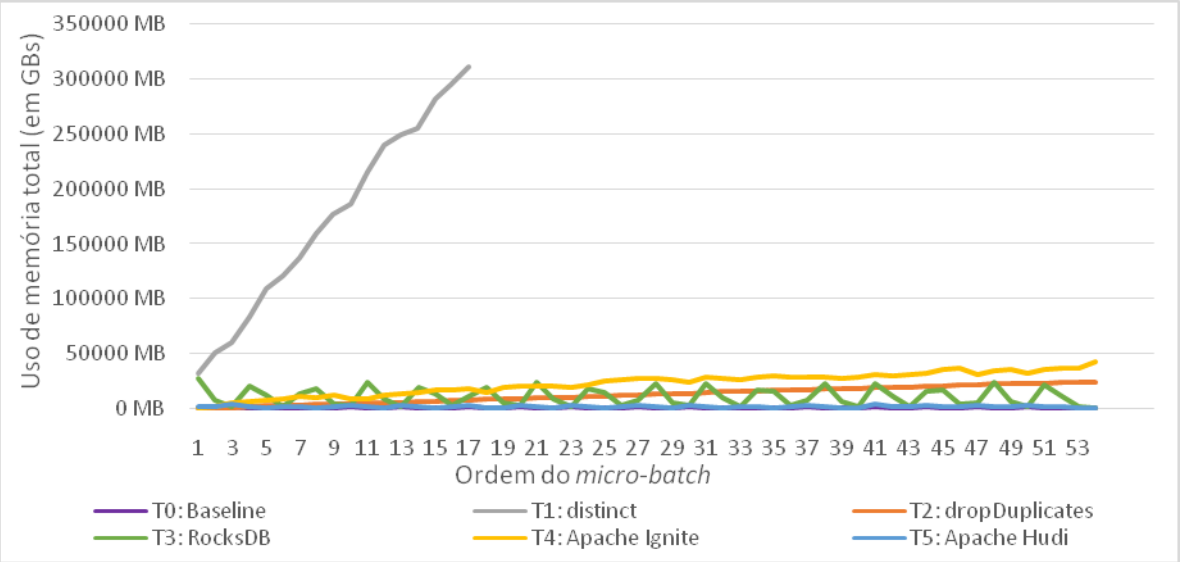
3.1. Uso de memória

Com base no requisito R1, foi analisada a utilização de memória RAM entre os testes realizados, com base nas métricas disponibilizadas nos registros (*logs*) de cada *streaming* ao final de cada *micro-batch*.

Esta análise considerou o uso de memória total pelos testes realizados, isto é, o uso de memória pelos operadores do Spark, o uso de memória pelo estado das aplicações e o uso de memória dos bancos utilizados nos testes T3 e T4.

O gráfico da Figura 17 apresenta a média de uso de memória das cinco execuções de cada teste realizado, onde o eixo x representa a ordem de execução dos *micro-batches* e o eixo y representa a quantidade total de memória utilizada por cada *micro-batches* nos testes realizados.

Figura 17 – Uso de memória entre *micro-batches* dos testes realizados



Fonte: O autor, 2021

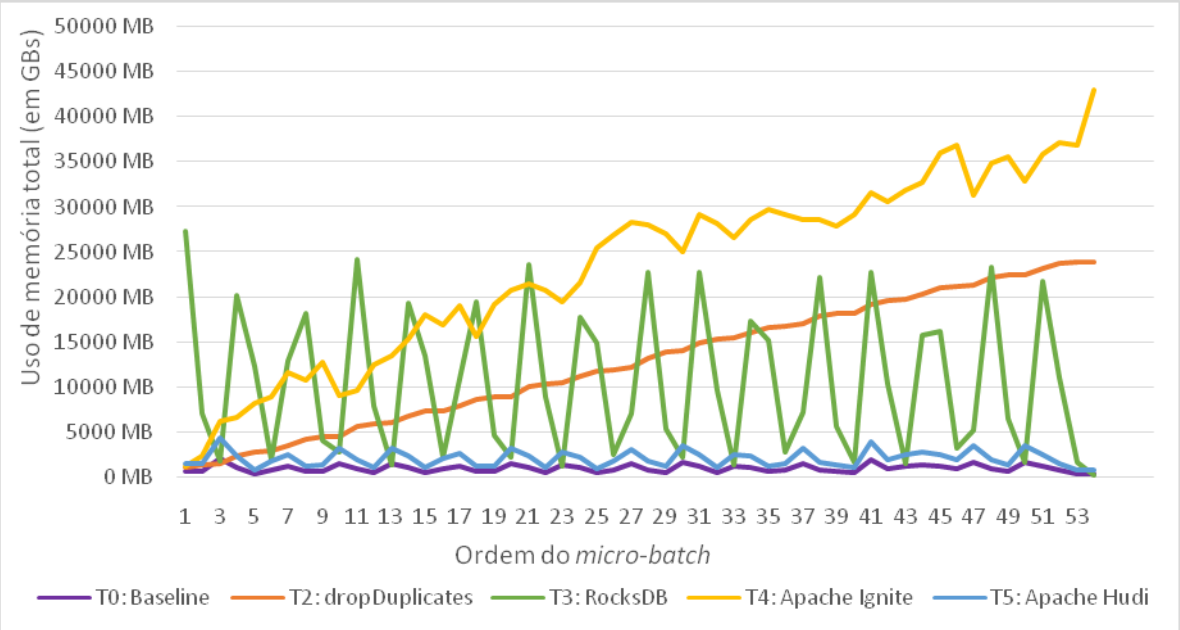
Neste gráfico, é possível perceber o crescimento exponencial no uso de memória ao utilizar o operador *distinct* no teste T1. Isso ocorre devido ao número elevado de colunas dos dados processados, o que implica em uma quantidade maior de comparações entre os dados e de colunas para armazenar no estado da aplicação entre *micro-batches*.

Nos dados utilizados para este estudo, existe uma percentagem maior de dados duplicados comparados aos dados únicos, como é apresentado na Figura 11, na seção 3.1. Isso implica que as comparações serão realizadas sobre todas as colunas múltiplas vezes na maioria dos casos, sem haver nenhuma interrupção por valores distintos entre duas colunas.

Pelo excesso de memória utilizada nos testes realizados com este operador, ocorreu um erro de falta de memória durante o processamento do 18º *micro-batch*, esse comportamento era esperado visto que seu uso de memória depois do primeiro *micro-batch* foi superior ao uso de todos os *micro-batches* entre os outros testes, com exceção do Apache Ignite no teste T4.

Removendo o teste T1 e ajustando a escala do eixo y, obtém-se o gráfico da Figura 18.

Figura 18 – Uso de memória entre *micro-batches* dos testes realizados, com exceção do teste T1



Fonte: O autor, 2021

Somente por trocar o uso do operador *distinct*, pelo operador *dropDuplicates* no teste T2, houve uma redução de 92% do uso de memória RAM comparado ao fim de ambos os testes. Além disso, também foi possível terminar a execução de todos os *micro-batches* e manter o crescimento no uso de memória em uma escala linear.

Utilizar o RocksDB como repositório de estado no teste T3 permitiu o *dropDuplicates* reduzir o uso de memória utilizada entre *micro-batches* para armazenar o estado, entretanto, o uso de memória pela aplicação se tornou instável, visto que a consulta aos dados do RocksDB por dado processado manteve mais operações de comparação em memória.

No teste T4, onde foi utilizado o Apache Ignite, ocorreu a maior utilização de memória entre os testes, excluindo o teste T1. Contudo, o uso de memória por esse teste foi utilizado em sua maior parte pelo Ignite, visto que as responsabilidades de validar a existência prévia das chaves e o armazenamento de chaves processadas anteriormente passaram a ser do banco de dados.

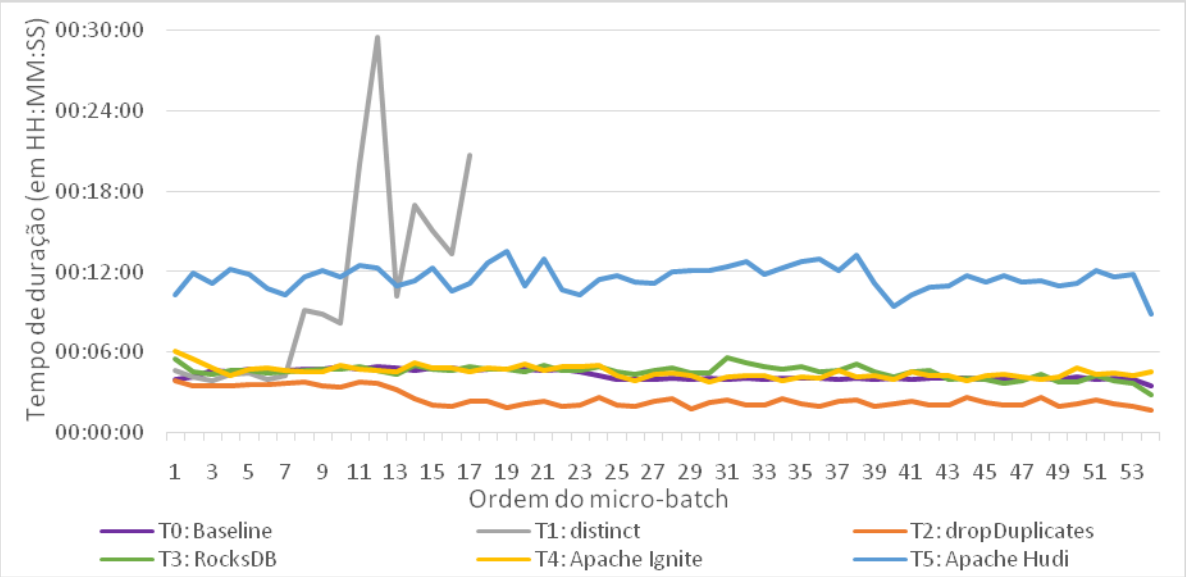
Por fim, o teste T5 obteve o menor uso de memória entre todos os testes, se mantendo abaixo dos 5GB em todos os *micro-batches*, mantendo um uso de memória semelhante ao *baseline*, isso ocorre devido ao Hudi não precisar persistir nenhum dado histórico no estado do *streaming*, assim como o *baseline*.

3.2. **Tempo de execução dos *micro-batches***

Utilizando os mesmos *logs* dos testes, foi possível analisar o tempo de execução dos testes, atendendo ao requisito R2. Esta análise considerou o tempo total de cada *micro-batch*, desde o momento que começou a ser processado, até o fim da persistência dos dados no repositório de dados.

O gráfico da Figura 19 apresenta a quantidade de *micro-batches* gerados (eixo x) e o tempo de execução média de cada *micro-batch*(eixo y – Formato: hh:mm:ss) entre os cinco testes realizados.

Figura 19 – Tempo de execução entre *micro-batches* dos testes realizados, em minutos



Fonte: O autor, 2021

Devido ao número maior de comparações, o operador *distinct* obteve o pior desempenho em questão de tempo de execução, extrapolando o limite de 10 minutos definido pelo domínio da aplicação.

No 12º *micro-batch* do teste T1, ocorreu o maior tempo de execução para o operador *distinct*. Isso ocorreu devido aos dados processados neste *micro-batch* terem tido a maior concentração de dados novos, ocasionando mais comparação entre elementos do *micro-batch* e do estado.

Em contrapartida, o operador *dropDuplicates* no teste T2, obteve o melhor desempenho, tendo um tempo médio de 2 minutos e 31 segundos, alcançando um tempo médio menor que o *baseline* (4 minutos e 22 segundos). Isso ocorre pelo fato da deduplicação reduzir a quantidade de dados que será persistida no repositório. A quantidade

menor de comparações que o *dropDuplicates* realiza comparado ao operador *distinct*, e o fato de por si só, o operador não salva o estado em um armazenamento externo ao Spark, também são características que justificam o tempo médio baixo.

Um aspecto importante de ser evidenciado, é que mesmo o processo sem deduplicação do *baseline* persistindo um volume maior de dados comparado a todos os outros testes, o tempo de execução do *micro-batch* não é impactado de forma significativa. Isso se deve não só pela ausência da lógica de deduplicação, como também ao fato de que qualquer que seja a lógica empregada, deverá haver o movimento (conhecido como *shuffle*) de dados entre nós do *cluster* para que seja garantido que um dado seja único, o que aumenta consideravelmente o tempo de execução do *micro-batch*.

Por utilizarem bancos de dados externos, os testes T3 e T4 que utilizaram o RocksDB e o Apache Ignite respectivamente, obtiveram resultados semelhantes ao *baseline*, com tempo médio de 4 minutos e 30 segundos. Esse tempo se manteve baixo, pois mesmo havendo consultas à disco, os bancos foram instanciados no mesmo *cluster* que o Spark, diminuindo o tráfego de rede externo.

Pelo fato de o Apache Hudi ter que realizar várias consultas ao repositório de dados localizado fora do *cluster*, no S3, o teste T5 obteve o segundo pior desempenho, com um tempo médio de 11 minutos e 32 segundos, consideravelmente acima do limite de tempo da SLA.

3.3. **Tolerância a falhas**

Nesta discussão, a tolerância a falhas é separada em duas perspectivas:

- **Falhas de aplicação,** que acontecem devido a ocorrência de um erro ou exceção não tratadas ou esperadas no escopo da aplicação. Como, por exemplo, a necessidade de uma maior quantidade de memória ou a divisão por zero;
- **Falhas de infraestrutura,** que acontecem fora do escopo da aplicação devido a falha ou remoção de um componente de *hardware*. Como, por exemplo, a falha em um disco rígido ou no módulo de memória, a remoção de uma instância *spot* do *cluster* entre outras falhas.

Devido a imprevisibilidade de erros de infraestrutura em um ambiente de nuvem, não foram criados cenários artificiais de falhas nos recursos dos *clusters* instanciados para

cada trabalho. Entretanto, devido ao comportamento característico de cada solução estudada, é possível discorrer sobre como cada uma é capaz de tratar uma falha inesperada, caso esta ocorra.

Por serem soluções nativas do Spark que persistem dados somente no espaço de memória da aplicação (volátil), os operadores *distinct* e *dropDuplicates* não são considerados tolerantes a falhas, visto que caso a aplicação seja encerrada por algum erro não tratado, todos os dados da memória serão perdidos.

A solução do RocksDB é tolerante a falhas de aplicação, ou seja, a qualquer encerramento da aplicação devido a persistência dos dados em disco. Entretanto, como não há replicação de dados, em caso de um nó do *cluster* ser encerrado, o banco não provê nenhum mecanismo de recuperação dos dados deste disco.

Uma forma de resolver essa deficiência do RocksDB, em ambientes de nuvem, é a utilização de discos anexados às máquinas via rede, que podem ser montados a outras instâncias, caso a instância original falhe. Entretanto, o processo de montar um volume a uma nova instância do RocksDB é demorado e não é trivial, causando lentidão no tempo de recuperação.

A replicação de dados do Apache Ignite permite que qualquer instância que possua o dado possa retorná-lo em uma consulta, isto permite que caso um nó do *cluster* seja encerrado, outros nós do *cluster* disponibilizem este dado enquanto um novo nó seja provisionado.

Como o Ignite pode operar em um *cluster* a parte do Spark, o término de nós do *cluster* do Ignite ou do Spark não gerará inconsistência nos dados utilizados pela ETL, tornando o processo mais tolerante a falha, garantindo o cumprimento do requisito R3.

O Apache Hudi mantém as chaves de todos os registros persistidos nos próprios arquivos gerados, o que garante que: se todo o *cluster* Spark for terminado, a reinicialização do *streaming* em um novo *cluster* irá considerar todas as chaves já conhecidas, garantindo a unicidade dos dados a serem persistidos. Desta forma, o Hudi pode ser classificado como a solução mais tolerante a falhas (tanto de aplicação quanto de infraestrutura) das testadas neste trabalho.

3.4. **Tamanho e quantidade de arquivos gerados**

Para permitir que o tempo de execução das consultas na camada de acesso não seja prejudicado, é necessário priorizar a geração de arquivos maiores em menor quantidade comparado a geração de uma quantidade grande de arquivos pequenos, conforme apresentado no requisito R4.

O Apache Spark gerará uma quantidade de arquivos igual a quantidade de partições de *dataframe*, que também estão relacionadas ao paralelismo configurado pelo *framework*. Uma forma de controlar o número de partições de um *dataframe* é através dos seguintes comandos:

- *repartition*, que aglomera partições entre nós de um *cluster*, balanceando o tamanho em *bytes* de cada partição final, ao custo de ter que realizar operações de *shuffle*;
- *coalesce*, que aglomera as partições dentro de cada nó do *cluster*, evitando o *shuffle* mas não gerando partições balanceadas com o mesmo tamanho.

Apesar destes operadores, não há nenhum mecanismo nativo do Spark que permita que os dados finais destas partições sejam adicionados a arquivos já existentes, fazendo com que processos ETL que processem poucos dados em um período curto gerem arquivos menores.

Além deste problema, adicionar uma etapa de aglomeração de partições a ETL irá significativamente aumentar o tempo de execução da ETL, mesmo evitando o *shuffle*. O comparativo da quantidade total dos arquivos gerados pela média das cinco execuções de cada teste, assim como a média dos tamanhos mínimos, máximos, médios e totais, são apresentados na Tabela 3 e Tabela 4 – Tamanho dos arquivos gerados por cada teste, respectivamente.

Tabela 3 – Quantidade total de arquivos gerados por cada teste

	Baseline	***distinct*** **(teste T1)**	***dropDuplicates*** **(teste T2)**	**RocksDB (teste T3)**	**Ignite (teste T4)**	**Hudi (teste T5)**
Quantidade de arquivos gerados	114.276	106.785*	291.633	293.453	226.581	1.451
***teste não finalizado**						

Tabela 4 – Tamanho dos arquivos gerados por cada teste

Tamanho dos arquivos gerados	***Baseline***	***distinct* (teste T1)**	***dropDuplicates* (teste T2)**	**RocksDB (teste T3)**	**Ignite (teste T4)**	**Hudi (teste T5)**
Tamanho mínimo	55,0 KB	494,4 KB	71,1 KB	56,1 KB	114,4 KB	2,5 MB
Tamanho médio	2,8 MB	1,4 MB	668,8 KB	671,3 KB	740,1 KB	114,3 MB
Tamanho máximo	21,7 MB	3,1 MB	2,3 MB	2,3 MB	2,4 MB	122,5 MB
Tamanho total	323,3 GB	73,5 GB*	110,5 GB	111,5 GB	93,4 GB	86,2 GB
***teste não finalizado**						

Nestas tabelas, é importante ressaltar que o operador *distinct* não foi capaz de processar todos os dados devido à falta de memória para armazenar o estado entre *micro-batches*, por isso que o teste T1 obteve ótimos valores de métricas nesta tabela. Por essa mesma razão, houve menos arquivos com tamanhos variados, obtendo mínimos, máximos e médias melhores.

Quanto ao *baseline*, é possível perceber a quantidade menor de arquivos gerados no total (pouco mais que o dobro, comparado aos testes T2, T3 e T4). Isso ocorre devido a forma como são consumidos os dados do Kafka, sendo um processo de consumo instanciado pelo Spark para cada partição do tópico (neste caso, trinta partições). Por não haver a necessidade de operações posteriores, além da persistência no destino, o Spark não realiza operações de *shuffle* e tudo que é consumido por um consumidor, é persistido em um mesmo arquivo.

Utilizar o RocksDB no teste T3 para persistência do estado com o operador *dropDuplicates* também não melhorou a forma como o Spark persistiu os arquivos, comparado ao teste T2, obtendo valores nas métricas muito próximos, sendo considerados equivalentes.

Com o uso do Ignite no teste T4, foi possível reduzir a quantidade de arquivos comparado ao teste T2 em 65.052 unidades (22,3%). Essa redução também resultou na diminuição do tamanho total dos dados processados em 18,1 GB (16,2%).

Isso ocorreu devido ao LEFT ANTI JOIN utilizado no Ignite ter auxiliado na comparação de partições com mesma chave entre nós do Spark. Desta forma, as linhas do *dataframe* mais semelhantes puderam ser armazenadas na mesma partição, resultando em uma melhor taxa de compressão de arquivo ao ser persistido no repositório de dados.

O teste T5 obteve o melhor resultado neste requisito, obtendo a menor quantidade de arquivos gerados, com estes arquivos tendo em média 114,3 MB, próximo ao limite de 128 MB padrão do Apache Hudi.

Isso ocorreu porque o Hudi nativamente aglomera dados em arquivos já existentes, criando arquivos apenas quando um limiar configurado seja atingido, permitindo assim que menos metadados sejam criados.

Também é importante notar que os mínimos, os máximos e as médias entre o tamanho dos arquivos dos testes T2, T3 e T4 estão muito próximos, afetando o tempo de execução das consultas sobre esses arquivos de maneira semelhante.

Para analisar o quanto a quantidade de arquivos influencia na quantidade de metadados gerados, o resultado do teste T2 foi comprimido por um processo *batch* utilizando o operador *repartition*, com o objetivo de aproximar o tamanho dos arquivos a 128 MB. É importante ressaltar que o Spark não possui nenhum método de aglomeração de partições que se baseie no tamanho final dos arquivos comprimidos, logo este processo foi feito com a melhor aproximação possível.

Em seguida, o mesmo processo foi executado novamente, para aproximar o tamanho dos arquivos a 64 MB. Os resultados destes dois processos, comparados aos testes T2 e T5 são apresentados nas Tabela 5 e Tabela 6.

Tabela 5 – Quantidade de arquivos consolidados comparados aos testes T2 e T5

	dropDuplicates	***dropDuplicates* + *repartition* (128 MB)**	***dropDuplicates* + *repartition* (64 MB)**	**Hudi**
Quantidade de arquivos gerados	291.633	1.237	4.280	1.451

Tabela 6 – Tamanho dos arquivos consolidados comparados aos testes T2 e T5

Tamanho dos arquivos gerados	***dropDuplicates***	***dropDuplicates* + *repartition* (128 MB)**	***dropDuplicates* + *repartition* (64 MB)**	**Hudi**
Tamanho mínimo	71,1 KB	130,8 MB	61,0 MB	2,5 MB
Tamanho médio	668,8 KB	133,6 MB	67,3 MB	114,3 MB
Tamanho máximo	2,3 MB	135,9 MB	64,1 MB	122,5 MB
Tamanho total	110,5 GB	88,0 GB	83,5 GB	86,2 GB

Com esses dois processos, é possível notar que a quantidade de metadados gerados reduz consideravelmente o tamanho total ocupado pelos arquivos persistidos no repositório de dados.

Entretanto, é possível perceber que há um limite do quanto é possível reduzir o tamanho total dos arquivos em razão do tamanho de cada arquivo, com os arquivos próximos a 64 MB ocupando 4,5 GB (5,1%) a menos comparados aos arquivos próximos a 128 MB.

Estes dois processos mostram que é benéfico consolidar os arquivos do *streaming* do teste T2, entretanto, como este tipo de aplicação é concebida para nunca terminar, não é possível determinar o momento ideal para realizar esta consolidação.

Da mesma forma, consolidar os dados ao fim de cada *micro-batch* também não seria benéfico, visto que à medida que a quantidade de arquivos cresce, esta etapa aumentaria o tempo de execução do *micro-batch* consideravelmente por ter que analisar todos os dados persistidos novamente.

Além disso, esta estratégia também poderá causar erros nas consultas que ocorrem simultaneamente a esta consolidação, visto que como não há nenhum mecanismo para bloquear os dados para leitura enquanto eles são consolidados, a consulta poderá perder a referência ao arquivo analisado quando este for excluído e recriado pela consolidação.

O Apache Hudi possui um desempenho consideravelmente melhor neste aspecto, visto que o formato irá apenas adicionar dados ao fim dos respectivos arquivos, sem removê-los. Além disso, o acesso ocorrerá somente aos arquivos necessários e não a todos os dados persistidos.

Mesmo assim, é possível notar com este processo de consolidação que o Hudi possui uma variância nos tamanhos dos arquivos gerados muito alta. Isto ocorre pelo fato de o Hudi executar uma política de melhor esforço para reduzir o tamanho dos arquivos.

Comparado com a consolidação de arquivos para 128 MB, obteve uma quantidade próxima de arquivos gerados no total, assim como um tamanho total semelhante. Com isso, é possível observar que a quantidade de metadados inseridos pelo Hudi sobre o formato de arquivos Parquet, que é utilizado internamente pelo Hudi, é mínima.

3.5. **Tempo de consulta**

Utilizando o Apache Presto para acessar os dados no S3, foi possível analisar o tempo de execução das consultas sobre os dados persistidos por cada teste realizado, com

exceção do teste T1, além dos dados persistidos pelo processo de ingestão em *batch* original.

Para isso, foram utilizadas cinco consultas comumente realizadas pelos analistas da empresa de classificados, obtendo o tempo de execução final (hh:mm:ss) como a média de cinco execuções, como mostra a Tabela 7. As consultas são descritas no Apêndice.

Tabela 7 – Tempo de execução das consultas sobre os dados finais dos testes

Consulta	***Baseline***	**Processo em *batch***	***dropDuplicates* (teste T2)**	**RocksDB (teste T3)**	**Ignite (teste T4)**	**Hudi (teste T5)**
1	00:28:48	00:03:20	00:56:29	00:56:01	00:49:58	00:01:02
2	> 05:00:00	00:01:25	> 05:00:00	> 05:00:00	> 05:00:00	00:01:11
3	> 05:00:00	00:06:01	> 05:00:00	> 05:00:00	> 05:00:00	00:03:06
4	> 05:00:00	00:06:13	> 05:00:00	> 05:00:00	> 05:00:00	00:03:12
5	01:15:50	00:02:38	00:59:39	00:58:42	00:50:47	00:01:29

Os tempos obtidos nas consultas realizadas nos testes T2, T3 e T4, impossibilitaram a utilização dos respectivos métodos como alternativa ao processo original, sendo que nas consultas 2, 3 e 4, o tempo de execução superou o tempo máximo que foi configurado no *cluster* de Presto da empresa. Isso ocorre devido a quantidade de arquivos a serem acessados pelo Presto, ressaltando a necessidade de consolidar a quantidade de arquivos gerados pelo *streaming*.

Em contrapartida, devido a quantidade de arquivos reduzida produzidos no teste T5, os tempos obtidos foram inferiores ao processo original, tornando o ideal para substituir o processo em *batch* e atender ao requisito de reduzir o tempo de consulta (requisito R4).

Com relação ao *baseline*, é importante que as consultas foram modificadas para realizar a deduplicação em cima do resultado obtido, entretanto, na consulta 1 esta massa de dados obteve um tempo menor que os testes T2, T3 e T4. Isso é devido a quantidade menor de arquivos gerados no *baseline*, assim como o tipo da própria consulta, isto é, devido ao fato da consulta retornar uma contagem feita sobre um intervalo de partições que continham arquivos mais compactados. Já comparada a consulta 5, o *baseline* obteve o pior desempenho devido a esta consulta demandar muitas comparações com o operador OU-condicional, sobre uma quantidade maior de dados no total.

3.6. **Análise dos resultados**

Com base nos resultados obtidos, é notável que a escolha de qual método utilizar deve ser baseada em qual requisito deve ser priorizado, com exceção do método *distinct* que não se mostra adequado para aplicações *streaming*.

Caso não haja a necessidade de remover duplicatas entre *micro-batches* de forma tolerante a falhas (por exemplo, quando um termo de SLA permitir uma percentagem baixa de duplicatas), é possível utilizar o operador *dropDuplicates* sem o auxílio de nenhuma solução que persista o estado da aplicação fora do escopo do Spark, conseguindo o máximo de desempenho.

O uso de um banco de dados externo, como o Apache Ignite e o RocksDB, para persistência das chaves utilizadas pelo processo de deduplicação não prejudicou significativamente o desempenho do *streaming*, o que garante uma tolerância maior a falhas.

Caso o banco de dados utilizado possa ser executado em um *cluster* e possua um mecanismo de replicação de dados entre nós que possa ser executado paralelamente, há um aumento ainda maior na tolerância a falhas da aplicação, por este motivo, o Apache Ignite possui uma vantagem comparado ao RocksDB.

Por fim, se o tempo de SLA for negociável ou a quantidade de dados processados por *micro-batch* puder ser reduzida, o Apache Hudi é o mais recomendado por garantir a unicidade dos dados internamente nos metadados dos arquivos persistidos, e por reduzir a quantidade e o tamanho total dos arquivos.

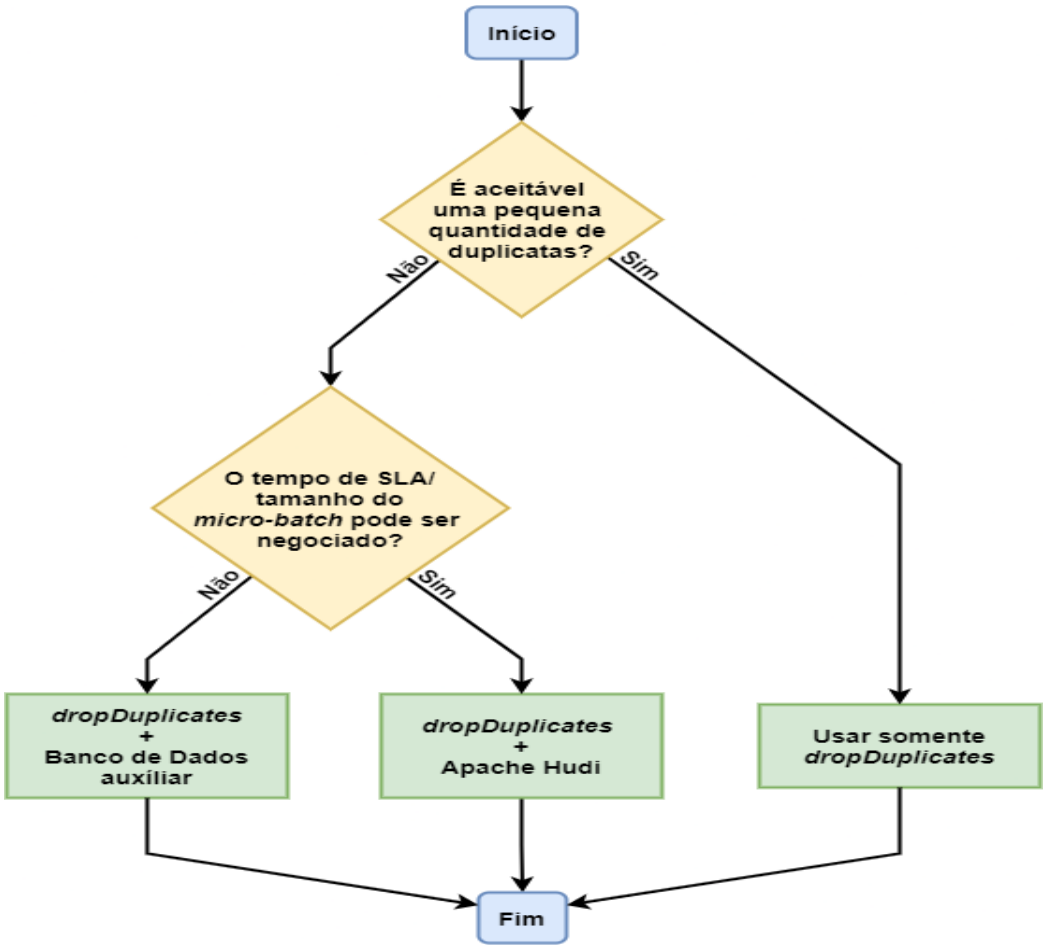
Estes critérios estão resumidos na Tabela 8 no fluxograma da Figura 20.

Tabela 8 – Comparativo entre as soluções estudadas

Solução	**Uso de memória**	**Tempo de entrega**	**Tempo de consulta**	**Tolerância a falhas**	**Observações**
distinct	Alto	Alto	Alto	Nenhuma	Não utilizar em aplicações *streaming*
dropDuplicates	Baixo	Baixo	Alto	Nenhuma	Utilizar caso seja aceitável um percentual de duplicatas

RocksDB	Baixo	Baixo	Alto	Tolerância a falhas de aplicação	É preferível utilizar uma solução mais tolerante a falhas
Apache Ignite	Baixo	Baixo	Alto	Tolerância a falhas de aplicação e infraestrutura	Utilizar caso seja necessário garantir totalmente a unicidade dos dados
Apache Hudi	Baixo	Alto	Baixo	Tolerância a falhas de aplicação e infraestrutura	Utilizar caso seja necessário garantir totalmente a unicidade dos dados e um baixo tempo de consulta

Figura 20 – Fluxograma para decidir o método de deduplicação



Fonte: O autor, 2021

4. TRABALHOS RELACIONADOS

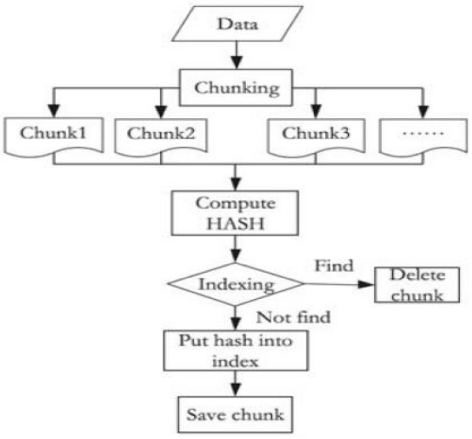
Este capítulo apresenta os trabalhos relacionados, segundo duas perspectivas: deduplicação de dados e gerenciamento de estado de aplicações *streaming*. Ao final de cada perspectiva é realizada uma comparação com a abordagem apresentada pelo presente trabalho.

4.1. Deduplicação de dados

Os trabalhos revisados sobre deduplicação de dados, em sua maioria, investem na deduplicação de dados quando eles já se encontram persistidos em disco, com o objetivo de reduzir o volume ocupado pelos arquivos armazenados, assim como reduzir a quantidade de metadados associados a estes arquivos.

Uma característica em comum entre estes trabalhos, são as etapas do processo de deduplicação para essa redução de volume nos dados, como mostra a Figura 21, que são: os arquivos analisados são agrupados, divididos em arquivos menores (*chunks*), comprimidos utilizando uma função *hash*, indexados, comparados e, caso uma duplicata seja encontrada, descartada.

Figura 21 – Processo de deduplicação de dados em arquivos



Fonte: Zhang et al. 2017

O trabalho de Debnath *et al.* (2010) apresenta uma análise sobre como o uso de discos baseados em memórias *flash* pode melhorar o acesso ao disco durante o processo de deduplicação *inline* (isto é, que ocorre durante a operação de escrita em disco), ao reduzir o

tempo de consulta à dados em disco. Para este fim, os autores desenvolveram um indexador de *chunks*, o ChunkStash. Este indexador mantém o mapeamento de *hashes* em memória, apontando para regiões contíguas em disco. Para acessar um *chunk* em disco, o ChunkStash utiliza algoritmos que aproveitam a organização dos dados em memória *flash* para maximizar a quantidade de dados recuperados em uma operação de leitura. Para avaliar o desempenho, foi utilizado um indexador proposto por Zhu *et al.* (2008), baseado no BerkeleyDB[43],e no uso de filtros de Bloom, em ambientes utilizando tanto discos rígidos tradicionais quanto os baseados em *flash*. No comparativo, o ChunkStash obteve um desempenho 36 vezes superior ao proposto por Zhu *et al.* (2008) executando em um disco rígido(HD) e 3,5 vezes superior quando executado em memória *flash*.

O trabalho de Srinivasan *et al.* (2012) também analisou o processo de deduplicação *inline*, com o objetivo de propor uma solução que permitisse que a deduplicação ocorresse com o menor impacto possível nas operações de acesso a disco realizadas pela aplicação de origem dos dados. A solução implementada, o iDedup, aproveitou duas propriedades encontradas em dados duplicados persistidos em disco: esses dados costumam estar persistidos na mesma seção do disco rígido (localidade espacial) e suas duplicatas costumam ser acessadas logo em seguida (localidade temporal). Dessa forma, as leituras ao disco puderam ser otimizadas e o dado duplicado mantido em memória por tempo suficiente para a consulta do processo de deduplicação. Utilizando essas propriedades, também foi possível reduzir a quantidade de espaço em disco utilizada ao salvar novos arquivos deduplicados, assim como reduzir a fragmentação dos arquivos pelo disco, otimizando as futuras operações de leitura destes arquivos e diminuindo a quantidade de metadados gerados.

O trabalho de El-Shim *et al.* (2012) estudou processos de deduplicação de arquivos em sistemas Windows Server de algumas empresas para servir como base para arquitetar um novo mecanismo de deduplicação no Windows Server 2012. Nos sistemas de deduplicação analisados, foram encontradas duas técnicas de comparação entre arquivos: a comparação total entre arquivos e a comparação entre *chunks* de arquivos. Comparando estas técnicas, foi constatado que a comparação por *chunks* resultou na melhor economia de espaço em disco, obtendo resultados que variaram entre 2,3x a 15,8x de espaço ocupado reduzido. Essa vantagem obtida pela comparação baseada em *chunks* ocorre devido a existência de dados duplicados dentro dos próprios arquivos comparados. Desta forma, para

[43] https://www.oracle.com/database/technologies/related/berkeleydb.html, acessado em março de 2020.

obter o melhor desempenho possível na etapa de comparação é necessário reduzir ao máximo o tamanho dos *chunks* para que uma duplicata não seja persistida no resultado. O ganho em desempenho na redução do tamanho dos *chunks* entretanto começa a ser anulado pela perda do fator de compressão de dados, visto que em *chunks* menores há uma baixa taxa de compressão obtida. Por esse motivo, segundo os autores, o resultado obtido utilizando *chunks* de 4KB se mostra idêntico a *chunks* de 64KB.

O trabalho de Zhang *et al.* (2017) implementou um sistema de deduplicação de arquivos armazenados no HDFS, utilizando o Hadoop MapReduce para implementar as etapas de agregação, divisão e compressão em *hashes* de arquivos, e o Apache HBase[44]como banco de dados para indexação dos *hashes*. Neste trabalho, foi endereçado o problema de deduplicação de arquivos pequenos no HDFS, agregando vários arquivos em um arquivo maior antes do processo de deduplicação ser executado. Com arquivos maiores, a aplicação MapReduce obteve um ganho significativo de desempenho. Entretanto, uma limitação dessa prática foi que devido a agregação de arquivos menores sem considerar o contexto destes, o acesso aos dados nos arquivos resultantes necessita consultar o índice criado no HBase, para que se extraia somente os trechos de dados necessários pelo usuário.

Por fim, comparando os trabalhos relacionados a este trabalho, é possível notar algumas divergências com relação a abordagem e as motivações:

- O processo de deduplicação dos dados neste trabalho ocorre em tempo de ingestão, enquanto os trabalhos relacionados realizam a deduplicação após os dados serem persistidos;
- Embora a deduplicação dos dados gere uma economia em espaço de armazenamento, o foco deste trabalho está na exploração deste processo como forma de atender a um requisito do domínio dos dados, algo que não é explorado pelos trabalhos estudados;
- Os trabalhos estudados têm como objetivo explorar processos internos do sistema operacional ou de uma ferramenta em especial, enquanto este trabalho realiza uma análise exploratória de várias soluções que possam auxiliar no processo de deduplicação, comparando os resultados obtidos e identificando os casos de uso de cada solução.

Um comparativo detalhado entre os trabalhos relacionados e este trabalho é apresentado na Tabela 9, onde são comparados o tamanho e a natureza das amostras de

[44] https://hbase.apache.org/, acessado em março de 2020.

dados utilizadas, as soluções de deduplicação estudadas, as contribuições de cada trabalho e as limitações apontadas pelos autores de cada trabalho.

Tabela 9 – Comparativo entre este trabalho e os trabalhos relacionados a deduplicação de dados

Trabalho	Amostra de dados	Soluções utilizadas	Contribuições	Limitações
Debnath *et al.* (2010)	Amostras de 8GB, 32GB e 126GB de dados	ChunkStash	• Implementação de um processo de deduplicação de arquivos otimizado para discos baseados em memórias flash • Redução do tempo de busca de dados em disco durante o processo de deduplicação	Não especificado no trabalho
Srinivasan *et al.* (2012)	Operações de leitura e escrita em dois sistemas corporativos, totalizando 396GB lidos e 172 GB escritos	iDedup	• Implementação de um processo de deduplicação de arquivos que aproveita as localidades espacial e temporal dos dados duplicados • Redução da fragmentação dos arquivos resultantes e da quantidade de metadados gerados	Não especificado no trabalho
El-Shimi *et al.* (2012)	Arquivos provenientes de 20 empresas, totalizando 6,8TB	Processo interno do Windows Server 2012	• Comparativo entre o processo de deduplicação baseado na comparação de arquivos contra o baseado em *chunks* • Análise do tamanho ideal de um *chunk* para a melhor taxa de compressão de dados e de eliminação de duplicatas	Os resultados encontrados estão fortemente correlacionados ao funcionamento do próprio sistema operacional
Zhang *et al.* (2017)	300 figuras com tamanhos entre 100KB e 5MB, totalizando 600MB	• Hadoop MapReduce • Hadoop HDFS • Apache HBase	• Implementação de um processo de deduplicação de arquivos • Implementação de um processo de agrupamento de arquivos pequenos • Comparação entre dois algoritmos de *hashing*: SHA-2 e Keccak	• O acesso aos arquivos deduplicados devem consultar um índice no HBase • A consulta a dados de arquivos pequenos deve ler totalmente um arquivo grande
Esteves (2021)	1,6 TB de mensagens recebidas por *streaming*	• Apache Kafka • Apache Spark • Apache Ignite • Apache Hudi	• Comparação entre cinco soluções de deduplicação de dados em tempo real baseada em tempo de entrega dos dados e utilização de recursos computacionais • Identificação dos cenários onde é recomendado cada solução	Os resultados encontrados não são generalizáveis a outros *frameworks* de processamento de dados

4.2. **Gerenciamento do estado de aplicações *streaming***

Este trabalho analisou os mecanismos nativos do Apache Spark para gerência do estado de uma aplicação *streaming*, assim como a utilização de outras ferramentas para auxiliar na persistência desse estado em caso de um término inesperado do processo de deduplicação.

Estas ferramentas não necessariamente são desenvolvidas especificamente com esse objetivo, entretanto estas obtiveram êxito em garantir que as chaves necessárias para o processo de deduplicação não fossem perdidas, sem que o processo principal sofresse um grande impacto no tempo de entrega de *micro-batches* ou no uso de recursos do *cluster*.

Na literatura, existem trabalhos que discorrem sobre os mecanismos de gerenciamento de estado nativos de outros *frameworks* de processamento de dados, como o SAND [Liu *et al.* 2016] e o Apache Flink [Carbone *et al.* 2017], assim como trabalhos que propõe gerenciadores com alta disponibilidade e tolerância a falhas que são agnósticos ao *framework* ou linguagem de programação utilizada, como o Megaphone [Hoffmann *et al.* 2019] e o Rhino [Del Monte *et al.* 2020].

O SAND foi desenvolvido com o objetivo de realizar o processamento de dados que trafegam pela rede que permita lidar com um alto volume de dados e que seja tolerante a falhas. Os autores decidiram criar uma plataforma para processamento de dados em tempo real, que não dependesse de mecanismos que aumentassem consideravelmente o uso de recursos computacionais, como uma máquina virtual, diminuindo a taxa de dados processados por segundo em razão dos recursos disponíveis [Liu *et al.* 2016].A limitação de recursos é dada pela necessidade de executar as aplicações próximas a origem dos pacotes de rede, o que motivou os autores a utilizar a linguagem de programação C++ para desenvolver o processador de dados. Quanto a tolerância a falhas, o SAND remove a persistência do estado do escopo da aplicação e a transfere a um *cluster* Zookeeper, que retoma a aplicação ao último estado caso ela seja reiniciada.

Em uma análise comparativa com o Apache Storm e o Blockmon [Huici *et al.* 2012], o SAND conseguiu uma vantagem considerável com relação a quantidade de pacotes de rede processados por segundo, assim como a quantidade de *bits* processados por segundo tanto do corpo quanto do cabeçalho dos pacotes, como mostra a Tabela 10.

Tabela 10 – Comparação entre os *frameworks* Storm, Blockmon e SAND

Framework	**Pacotes por segundo**	**Taxa de consumos de corpo dos pacotes**	**Taxa de consumo de cabeçalhos dos pacotes**
Storm	260 mil	840 Mb/s	81,15 Mb/s
Blockmon	2,7 milhões	8,4 Gb/s	844,9 Mb/s
SAND	9,6 milhões	31,4 Gb/s	3031,7 Mb/s

Fonte: Huici *et al.*, 2012

O Apache Flink gerencia o estado de um *streaming* através de rotinas internas de criação de *snapshot* de todo estado da aplicação que são executados ao fim de cada janela de processamento, possibilitando que esses *snapshots* sejam persistidos tanto em um sistema de arquivos (do próprio *cluster* de processamento ou externo), quanto em bancos de dados externos.

Uma vantagem de utilizar um banco de dados como armazenamento para o estado, além de possibilitar que a aplicação seja tolerante a falha do *cluster*, é a redução do tráfego de dados do *snapshot*, visto que o *framework* irá aproveitar os mecanismos de inserção e atualização de dados do banco para operar somente sobre dados novos, ao invés de ter de sobrescrever totalmente um *snapshot* antigo.

O Megaphone e o Rhino são soluções de gerenciamento de estado agnósticas a *frameworks* de processamento de dados, possibilitando a utilização de qualquer uma destas soluções em uma aplicação que seja implementada sem o auxílio de quaisquer *frameworks*.

Ambas as soluções se especializam em redistribuir as chaves persistidas no estado da aplicação quando os nós são adicionados ou removidos de um *cluster*. Essa abordagem é empregada enquanto a aplicação é executada, ao invés de ser empregada na reinicialização dela, o que melhora a tolerância a falhas, visto que na hipótese de um nó ficar indisponível, um novo nó pode substituí-lo.

Para que este processo de redistribuição ocorra, três etapas devem ser seguidas: a redistribuição deve ser agendada pela aplicação, os dados a serem distribuídos entre os nós devem ser recuperados e os dados devem ser carregados nos nós de destino.

Del monte *et al.* (2020)realizou um comparativo sobre o tempo destas três etapas entre o Rhino, o Megaphone e a solução nativa utilizada pelo Flink com amostras de 250GB, 500GB, 750GB e 1TB. Neste comparativo, o Rhino obteve um desempenho médio50x maior que o Flink e 15x maior que o Megaphone, com exceção das duas maiores cargas, onde o Megaphone falhou por falta de memória.

A Tabela 11 apresenta um comparativo entre este trabalho e os trabalhos relacionados discutidos acima, comparando as soluções empregadas, a tolerância a falhas alcançada por

cada solução, as contribuições de cada trabalho e as limitações apontadas pelos autores de cada trabalho.

Tabela 11 – Comparativo entre este trabalho e os trabalhos relacionados a gerenciamento de estado de aplicações *streaming*

Trabalho	Soluções analisadas	Tolerância a falhas	Contribuição	Limitações
Liu *et al.* (2016)	Apache Flink	Apenas de aplicação (salva o estado localmente)	Descrição do processo de gerência de estado baseado em *snapshots* utilizado pelo Apache Flink	Não especificado
Carbone *et al.* (2017)	• SAND • Blockmon • Apache Storm	Apenas de aplicação (as três soluções salvam o estado localmente)	Criação de um processador de dados em tempo real com alta tolerância a falhas para análise de pacotes de rede	A lógica empregada por esta solução é específica ao problema de análise de pacotes de rede
Hoffmann *et al.* (2019)	Megaphone	De aplicação e infraestrutura (um processo externo replica e gerencia o estado da aplicação)	Criação de um gerenciador de estados agnóstico a *frameworks*	Este gerenciador não suporta a migração de um grande volume de dados em um estado
Del Monte *et al.* (2020)	• Rhino • Megaphone • Apache Flink	• Apenas de aplicação (Flink) • De aplicação e infraestrutura (Rhino e Megaphone)	Criação de um gerenciador de estados agnóstico a *frameworks*	Não especificado
Esteves (2021)	• Apache Spark • Apache Kafka • Apache Ignite • Apache Hudi	• Apenas de aplicação (Spark) • De aplicação e infraestrutura (Hudi e Ignite)	Análise do uso de recursos e de tempo de processamento empregado por diferentes soluções de *streaming*	Os resultados encontrados não são generalizáveis a outros *frameworks* de processamento de dados

CONCLUSÃO

Este trabalho apresentou o problema de deduplicação de dados em tempo real no contexto de aplicações de *Big Data*, os desafios deste processo em aplicações *streaming* comparado a aplicações em *batch*, métodos de deduplicação em *streaming* utilizando o *framework* Apache Spark, os cenários em que cada método é preferível e, por fim, um método de decisão, ilustrado por um fluxograma, para qual método de deduplicação escolher.

Limitações

Com exceção dos operadores *distinct* e *dropDuplicates*, os métodos estudados neste trabalho não são específicos ao Apache Spark, sendo possível que a utilização destes como solução para o problema de deduplicação, ou qualquer outro problema que envolva persistir o estado do *streaming*, gere resultados diferentes.

Da mesma forma, é possível que uma amostra maior de dados de teste ou um fluxo maior de mensagens por segundo, enviadas pelo Apache Kafka, torne necessário a revisão destes métodos. Neste trabalho, foi utilizado um grande volume de dados, com uma taxa de consumo de mensagens razoavelmente alta, mas é possível que uma amostra de centenas de TBs ou que ultrapasse 1PB inviabilize-as. Entretanto, também não foi encontrado nenhum trabalho relacionado que utilize uma amostra tão grande assim.

Também é possível que outros problemas de *streaming*, como agregações em tempo real, tenham outros requisitos que os métodos estudados aqui não atendam. Por exemplo, o Apache Hudi possui um mecanismo específico para deduplicar dados com base nas chaves, entretanto, para agregações baseadas em valores que não sejam chaves esse método não será suficiente.

Quanto as observações quanto a tolerância a falhas dos métodos estudados, é importante ressaltar que os seguintes cenários de falha foram analisados:

- **Falha da aplicação**, onde o *streaming* é aleatoriamente interrompido e reiniciado;
- **Falha de um nó**, onde um dos nós de processamento é aletoriamente desligado e outro nó assume;
- **Falha de todo o *cluster***, onde todos os nós de processamento são aleatoriamente desligados e um novo *cluster* é provisionado para dar continuidade ao *streaming*.

Entretanto, existem outros cenários de falha, como a falha na comunicação entre nós via rede ou a indisponibilidade de um componente de *hardware* (ex.: HD sem espaço de armazenamento, *bit* corrompido em memória)que não foram estudados neste trabalho e que demandam uma análise complementar.

Por fim, a análise das consultas sobre os dados resultantes considerou apenas os dados salvos no S3, que acrescenta um atraso significativo de tempo por ser acessado pela rede e não localmente como um sistema de arquivos locais baseado no HDFS, por exemplo. Por esse mesmo motivo, otimizações de acesso a dados, como as empregadas em sistemas de arquivos em discos rígidos, não podem ser empregadas para diminuir o tempo de acesso, o que poderia melhorar o tempo de execução de cada consulta analisada.

Contribuições

Por ser tratar de um *framework* amplamente utilizado na academia e na indústria, os resultados encontrados neste trabalho, utilizando o Apache Spark, auxiliarão a decidir qual o melhor método de deduplicação a ser utilizado em uma aplicação *streaming* sobre um grande volume de dados, implementada utilizando este *framework*, direcionando o foco de futuros trabalhos ao problema a ser resolvido ao invés da configuração do ambiente e de quais ferramentas auxiliares utilizar.

Esta decisão é resumida pelo fluxograma da Figura 20, que abrange métricas como o uso de recursos computacionais, a necessidade de um ambiente altamente disponível e o tempo máximo para a entrega dos dados transformados pela aplicação, consolidando uma base ainda maior para a escolha do melhor método.

Trabalhos futuros

Como trabalhos futuros, alguns parâmetros da metodologia apresentada sobre a análise de métricas e requisitos de aplicações *streaming* que persistem seu estado podem ser alterados e gerar novos cenários de testes e formas de análises, tais como:

- O tipo de problema a ser solucionado pela aplicação (ex.: agregações, aprendizado de máquina em tempo real);
- O *framework* de processamento de dados (ex.: Apache Storm, Apache Flink);
- A infraestrutura dos nós do *cluster* (ex.: ambiente em nuvem, servidores locais);
- As ferramentas auxiliares (ex.: outros bancos de dados, relacionais e não-relacionais);

- O fluxo de mensagens enviadas por segundo;
- O tamanho da amostra utilizada;
- O *schema* dos dados de origem.

Com os resultados destas variações, é possível compilar um método mais genérico para a escolha do método mais recomendado para determinado requisito.

Outra possibilidade de trabalho futuro, é a implementação de uma rotina que gere aleatoriamente falhas na infraestrutura e na aplicação em si, simulando perda de nós, exceções sem tratamento no código, para que uma análise mais ampla no requisito de tolerância a falhas possa ser feita.

REFERÊNCIAS

[Atzori *et al.* 2010] Atzori, L., Iera, A., & Morabito, G. (2010). “The internet of things: A survey”. *Computer networks*, v. 54, n. 15, pp. 2787-2805, October.

[Berman 2018] Berman, J. J. (2018). “Principles and Practice of Big Data: Preparing, Sharing, and Analyzing Complex Information”, 2nd edition, Academic Press, July.

[Brackenbury *et al*. 2018] Brackenbury, W., Liu, R., Mondal, M., Elmore, A. J., Ur, B., Chard, K., & Franklin, M. J. (2018). “Draining the data swamp: A similarity-based approach”. *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pp. 13, June.

[Bloom 1970] Bloom, B. H. (1970). “Space/time trade-offs in hash coding with allowable errors”. *Communications of the ACM*, v. 13, n. 7, pp. 422-426, July.

[Brewer 2000] Brewer, E. A. (2000). “Towards robust distributed systems”. *ACM Symposium on Principles of Distributed Computing*, v. 7, July.

[Carbone *et al.* 2017] Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., & Tzoumas, K. (2017). “State management in Apache Flink: consistent stateful distributed stream processing”. *Proceedings of the VLDB Endowment*, v. 10, n. 12, pp. 1718-1729, August.

[Cattell 2011] Cattell, R. (2011). “Scalable SQL and NoSQL data stores”. *ACM Sigmod Record*, v. 39, n. 4, pp. 12-27, May.

[Chambers & Zaharia 2018] Chambers, B., & Zaharia, M. (2018). “Spark: the definitive guide: big data processing made simple”. *O'Reilly Media, Inc*, Febuary.

[Debattista *et al.* 2015] Debattista, J., Lange, C., Scerri, S., & Auer, S. (2015). “Linked'Big'Data: towards a manifold increase in big data value and veracity”. *2nd International Symposium on Big Data Computing (BDC)*, pp. 92-98, December.

[Debnath *et al.* 2010] Debnath, B. K., Sengupta, S., & Li, J. (2010). “ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory”. *USENIX annual technical conference*, pp. 1-16, June.

[Dedic & Stanier 2016] Dedic, N., Stanier, C. (2016). “Towards differentiating business intelligence, big data, data analytics and knowledge discovery.” *Springer International Conference on Enterprise Resource Planning Systems*. pp. 114–122, November.

[Del Monte *et al.* 2020] Del Monte, B., Zeuch, S., Rabl, T., & Markl, V. (2020). “Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines”. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2471-2486, June.

[Dobos *et al.* 2013] Dobos, L., Szüle, J., Bodnár, T., Hanyecz, T., Sebők, T., Kondor, & Vattay, G. (2013). “A multi-terabyte relational database for geo-tagged social network data”.

4th International Conference on Cognitive Infocommunications (CogInfoCom), pp. 289-294, December.

[Dong & Naumann2009] Dong, X. L., & Naumann, F. (2009). "Data fusion: resolving data conflicts for integration". *Proceedings of the VLDB Endowment*, v. 2, n. 2, pp. 1654-1655.

[Dong *et al.* 2013] Dong, J., Wang, F., & Yuan, B. (2013). "Accelerating BIRCH for clustering large scale streaming data using CUDA dynamic parallelism". *Intelligent Data Engineering and Automated Learning (IDEAL)*, vol 8206, pp. 409-416, October.

[Du 2018] Du, D. (2018). "Apache Hive Essentials". 2nd edition, *Packt Publishing Ltd*, June.

[Elmasri & Navathe 2016] Elmasri, R., & Navathe, S. (2016). "Fundamentals of database systems". 7th edition, Pearson, July.

[El-Shimi *et al*. 2012] El-Shimi, A., Kalach, R., Kumar, A., Ottean, A., Li, J., & Sengupta, S. (2012). "Primary data deduplication—large scale study and system design". *2012 USENIX Annual Technical Conference*, pp. 285-296, June.

[Erl *et al*. 2016] Erl, T., Khattak, W., & Buhler, P. (2016). "Big data fundamentals: concepts, drivers & techniques". Prentice Hall Press, January.

[Fisk 2019] Fisk, N. (2019). "Mastering Ceph: Infrastructure storage solutions with the latest Ceph release". 2nd edition, Packt Publishing Ltd, March.

[Gantz & Reinsel 2012] Gantz, J.; Reinsel, D. (2012) "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east." *IDC iView: IDC Analyze the future*, v. 2007, pp. 1–16, December.

[Gubbi *et al.* 2013] Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M. (2013). "Internet of Things (IoT): A vision, architectural elements, and future directions." *Future generation computer systems*, v. 29, n. 7, pp. 1645-1660, September.

[Guo *et al*. 2019] Guo, L., Hua, L., Jia, R., Zhao, B., Wang, X., & Cui, B. (2019). "Buying or Browsing?: Predicting Real-time Purchasing Intent using Attention-based Deep Network with Multiple Behavior". *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1984-1992, July.

[Hashem *et al.* 2015] Hashem, I. A. T.; Yaqoob, I.; Anuar, N. B.; Mokhtar, S.; Gani, A.; Khan, S. U. (2015). "The rise of big data on cloud computing: Review and open research issues". *Information systems*, v. 47, pp. 98–115, January.

[Hoffmann *et al.* 2019] Hoffmann, M., Lattuada, A., & McSherry, F. (2019). "Megaphone: Latency-conscious state migration for distributed streaming dataflows". *Proceedings of the VLDB Endowment*, v. 12, n. 9, pp. 1002-1015, May.

[Hendricks 2017] Hendricks, D. (2017). "Using real-time cluster configurations of streaming asynchronous features as online state descriptors in financial markets". *Pattern Recognition Letters*, v. 97, pp. 21-28, October.

[Huici *et al.* 2012] Huici, F., Di Pietro, A., Trammell, B., Gomez Hidalgo, J. M., Martinez Ruiz, D., & d'Heureuse, N. (2012). "Blockmon: a high-performance composable network

traffic measurement system". *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 79-80, August.

[Ilyas & Chu2019] Ilyas, I. F., & Chu, X. (2019). "Data cleaning". *Morgan & Claypool.*

[Ingram 2009] Ingram, D. (2009). "Design-Build-Run: Applied Practices and Principles for Production Ready Software Development". 1st edition, Wrox, February.

[Ivanov & Pergolesi 2019] Ivanov, T., & Pergolesi, M. (2019). "The impact of columnar file formats on SQL-on-Hadoop engine performance: A study on ORC and Parquet". Disponível em: https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.5523, acessado em março de 2020.

[Jacobson 2013] Jacobson, R. (2013). "2.5 quintillion bytes of data created every day. how does CPG & Retail manage it?". *IBM*.

[Kreps 2014] Kreps, J. (2014). "Questioning the lambda architecture". *O'Reilly Media*. Disponível em: https://www.oreilly.com/radar/questioning-the-lambda-architecture/, July.

[Leavitt 2010] Leavitt, N. (2010). "Will NoSQL databases live up to their promise?". *Computer*, v. 43, n. 2, pp. 12-14, Febuary.

[Levandoski *et al*. 2013] Levandoski, J. J., Larson, P. Å., & Stoica, R. (2013). "Identifying hot and cold data in main-memory databases". *IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 26-37, April.

[Lewis 2021] Lewis, L. (2021) "Infographic: What Happens in An Internet Minute 2021." *Merge*. Disponível em: https://www.allaccess.com/merge/archive/32972/infographic-what-happens-in-an-internet-minute/, acessado em abril de 2021.

[Li & Manoharan 2013] Li, Y., & Manoharan, S. (2013). "A performance comparison of SQL and NoSQL databases". *2013IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pp. 15-19, August.

[Liu *et al.* 2016] Liu, Q., Lui, J. C., He, C., Pan, L., Fan, W., & Shi, Y. (2016). "SAND: A fault-tolerant streaming architecture for network traffic analytics". *Journal of Systems and Software, Elsevier*, v. 122, pp. 553-563, December.

[Manyika 2017] Manyika, J. (2017) "A future that works: AI, automation, employment, and productivity." *McKinsey Global Institute Research*. Disponível em: https://www.mckinsey.com/˜/media/mckinsey/featured%20insights/Digital%20Disruption/Harnessing%20automation%20for%20a%20future%20that%20works/MGI-A-future-thatworks-Executive-summary.ashx, acessado em março de 2020.

[Marz & Warren 2015] Marz, N., & Warren, J. (2015). "Big Data: Principles and best practices of scalable real-time data systems". *Manning Publications*, 2 edition, April.

[Mashey 1998] Mashey, J. R (1998). "Big data and the next wave of InfraStress: Problems, solutions, opportunities". *Computer Science Division Seminar*, April.

[Meyer & Bolosky 2012] Meyer, D. T., & Bolosky, W. J. (2012). "A study of practical deduplication". *ACM Transactions on Storage*, v. 7, n. 4, pp. 1-20, Febuary.

[Normandeau 2013] Normandeau, K. (2013). “Beyond volume, variety and velocity is the issue of big data veracity”. Inside big data.

[Quoc *et al.* 2017] Quoc, D. L., Chen, R., Bhatotia, P., Fetzer, C., Hilt, V., & Strufe, T. (2017). “Streamapprox: Approximate computing for stream analytics”. *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pp. 185-197, December.

[Rovnyagin *et al.* 2020] Rovnyagin, M. M., Kozlov, V. K., Mitenkov, R. A., Gukov, A. D., & Yakovlev, A. A. (2020). “Caching and Storage Optimizations for Big Data Streaming Systems”. *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pp. 468-471. IEEE, January

[Rubin & Lukoianova 2013] Rubin, V., & Lukoianova, T. (2013). “Veracity roadmap: Is big data objective, truthful and credible?”. *Advances in Classification Research Online*, v. 24, n. 1, November.

[Srinivasan *et al*. 2012] Srinivasan, K., Bisson, T., Goodson, G. R., & Voruganti, K. (2012). “iDedup: latency-aware, inline data deduplication for primary storage”. *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, v. 12, n. 1, pp. 1-14, Febuary.

[Sousa 2014] Sousa, F. R. C. (2014) “Big Data: Conceitos e Desafios”. X Escola Regional de Banco de Dados, Abril.

[Snijders *et al.* 2012] Snijders, C., Matzat, U., Reips, U. (2012). “Big Data: big gaps of knowledge in the field of internet Science”. *International Journal of Internet Science*, v. 7, n. 1, pp. 1–5, January.

[Sousa 2015] Sousa, F. (2015) “Big Data: Conceitos e Desafios”, *Universidade Federal do Ceará, Departamento de Engenharia de Teleinformática*, Fortaleza, 2015.

[Srinivasan *et al.* 2012] Srinivasan, K., Bisson, T., Goodson, G. R., & Voruganti, K. (2012). “iDedup: latency-aware, inline data deduplication for primary storage”. *Fast*, v. 12, pp. 1-14, February.

[Stein & Morrison 2014] Stein, B., & Morrison, A. (2014). “The enterprise data lake: Better integration and deeper analytics”. *PwC Technology Forecast: Rethinking integration*. Disponível em: https://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/assets/pdf/pwc-technology-forecast-data-lakes.pdf, acessado em março de 2020.

[Szalay *et al.* 2000] Szalay, A. S., Kunszt, P. Z., Thakar, A., Gray, J., Slutz, D., & Brunner, R. J. (2000). “Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey”. *ACM SIGMOD Record*, v. 29, n. 2, pp. 451-462, June.

[Theodorou et al. 2017] Theodorou, V., Abelló, A., Thiele, M., & Lehner, W. (2017). “Frequent patterns in ETL workflows: An empirical approach”. Data & Knowledge Engineering, v. 112, pp. 1-16, November.

[Zaharia *et al.* 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). “Spark: Cluster computing with working sets”. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, v. 10, pp. 10, July.

[Zaharia *et al.* 2016] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., & Ghodsi, A. (2016). “Apache Spark: a unified engine for big data processing”. *Communications of the ACM*, v. 59, n. 11, pp. 56-65, October.

[Zhang *et al.* 2017] Zhang, D., Liao, C., Yan, W., Tao, R., & Zheng, W. (2017). “Data deduplication based on Hadoop”. *Fifth International Conference on Advanced Cloud and Big Data*, pp. 147-152, August.

[Zhu *et al.* 2008] Zhu, B., Li, K., & Patterson, R. H. (2008). “Avoiding the Disk Bottleneck in the Data Domain Deduplication File System”. *Fast*, v. 8, pp. 1-14, February.

APÊNDICE A - Consultas

Consulta 1: Recupera a quantidade de anúncios distintos removidos pela exclusão da conta de um usuário em um mês, utilizando a coluna de partição **dt**.

```
SELECT count(distinct ad_id)
FROM <tabela>
WHERE dt BETWEEN date '2019-11-01' AND date '2019-11-30'
  AND delete_reason ='user_account_deleted'
```

Consulta 2: Recupera todos os estágios do ciclo de vida de todos os anúncios, assim como a data e a hora do momento em que o evento de ciclo de vida ocorreu, em um dia específico utilizando a coluna de partição **dt**.

```
SELECT ad_id, lifecycle_id, event_timestamp
FROM <tabela>
WHERE dt = date('2019-11-23')
```

Consulta 3: Recupera todos os títulos e os textos de todos os anúncios em um dia, que possuam menção a frase “GB memória” precedida de um número, no mínimo uma vez, no máximo três vezes, ignorando letras minúsculas e maiúsculas, utilizando a coluna de partição **dt** e expressões regulares.

```
SELECT ad_id, lifecycle_id, body, subject
FROM <tabela>
WHERE dt = date('2019-11-23')
AND (
  regexp_like(body, '(?i)((((\s|\S){5,}|\A))(\d.?GB memória)){1,3}')
  OR regexp_like(subject, '(?i)((((\s|\S){5,}|\A))(\d.?GB memória)){1,3}}')
)
```

Consulta 4: Recupera a contagem por dias de registros, por dia, que possuam um *lifecycle_id* igual a 20 ou 52, em um intervalo de dois dias, utilizando a coluna de partição **dt**.

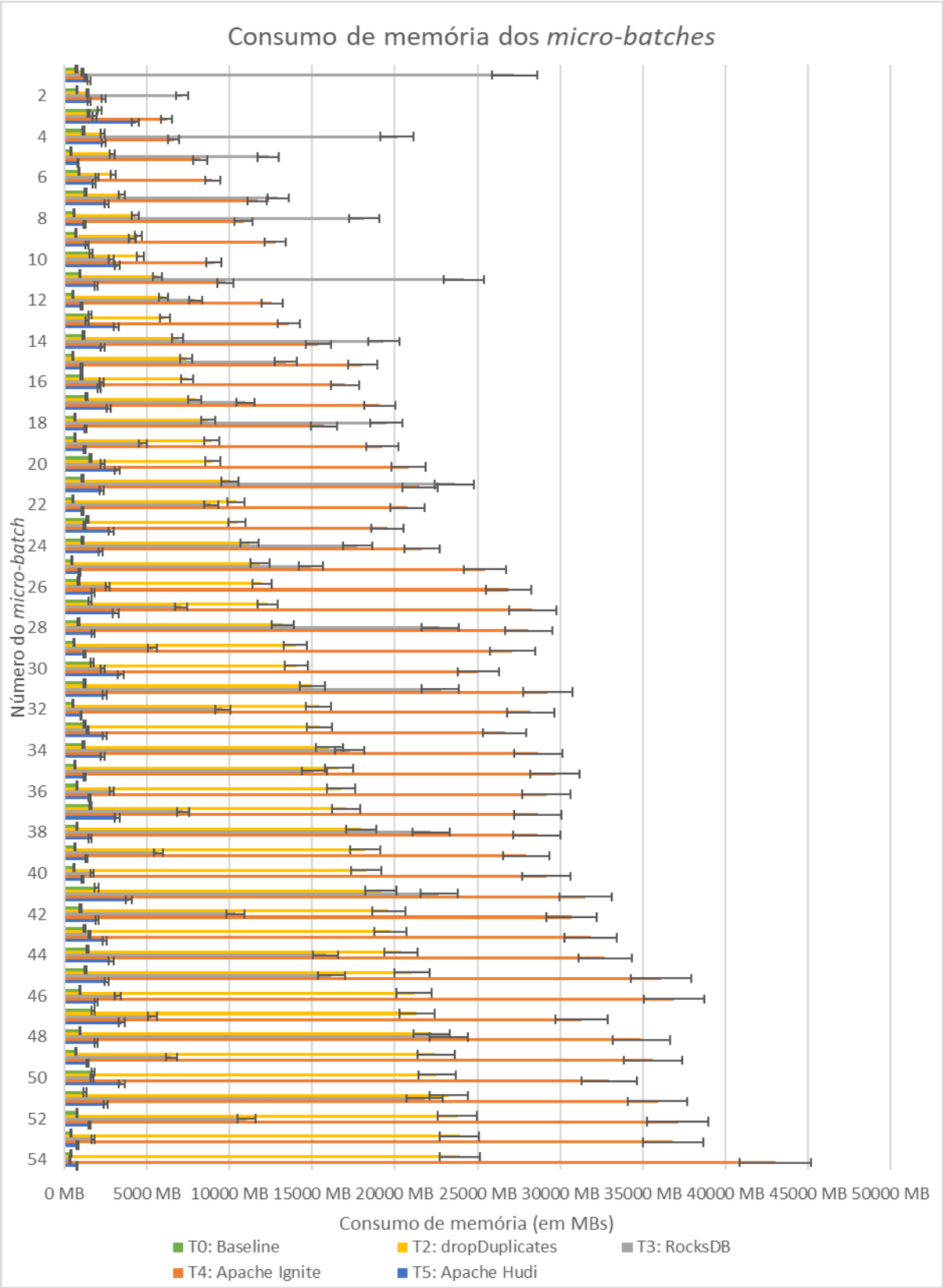
```
SELECT date(timestamp) date, count(1)
FROM <tabela>
WHERE dt BETWEEN date('2019-11-22') AND date('2019-11-24')
AND lifecycle_id IN (20, 52)
GROUP BY 1
```

Consulta 5: Recupera a contagem total de registros persistidos que possuam o título do anúncio com menção a uma das quatro frases: “luva cirúrgica”, “máscara”, “álcool em gel” ou “álcool gel”, sem utilizar a partição das tabelas (com exceção da consulta aos dados originais, onde as partições lidas foram somente as que possuíam a coluna **dt** igual as obtidas pela massa de dados utilizada nos testes deste trabalho).

```
SELECT count(1)
FROM <tabela>
WHERE lower(subject) LIKE '%luva cirurgica%'
  OR lower(subject) LIKE '%mascara%'
  OR lower(subject) LIKE '%alcool em gel%'
  OR lower(subject) LIKE '%alcool gel%'
```

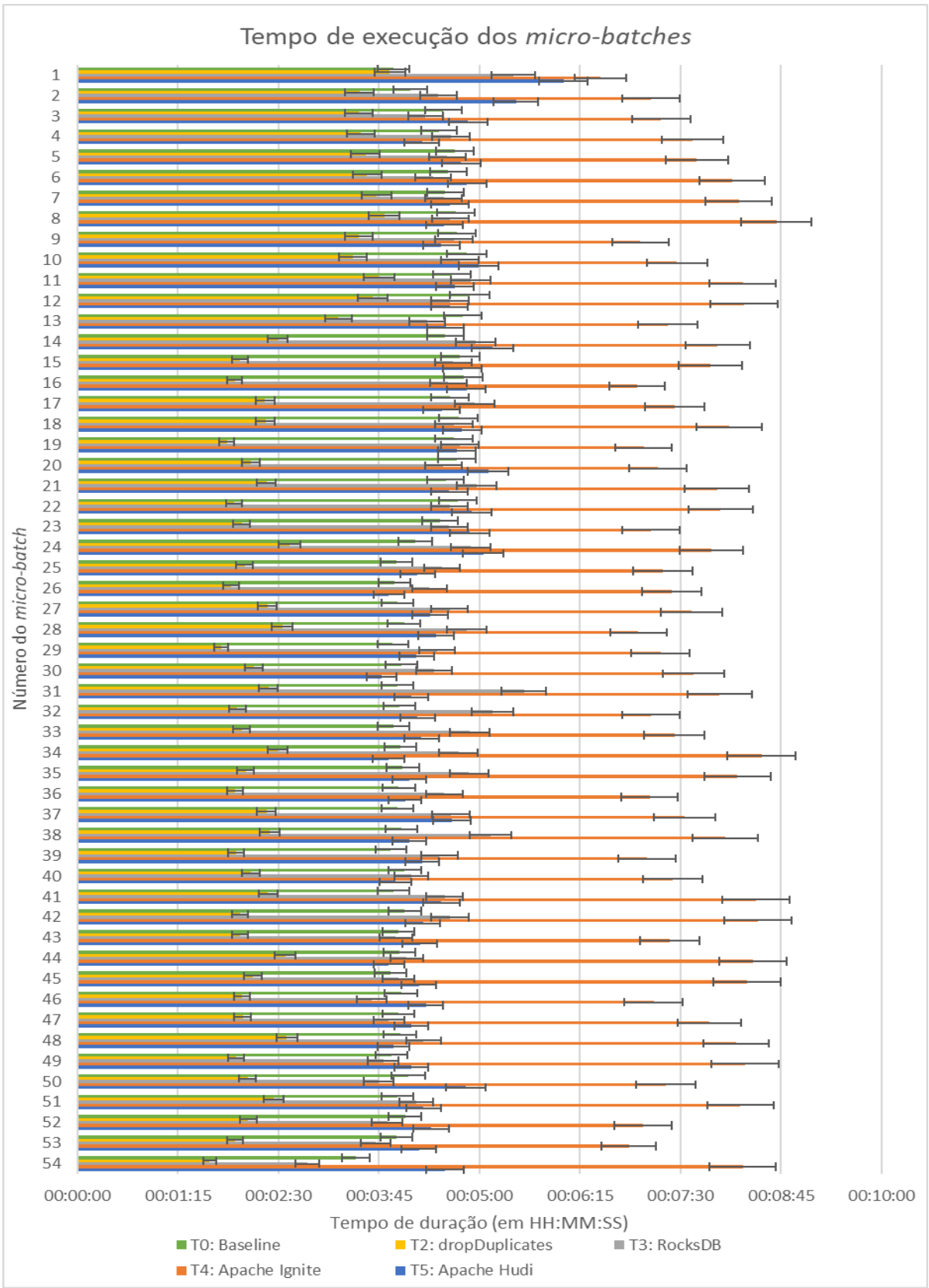
APÊNDICE B – Gráficos de desvio padrão

Figura 22 – Consumo médio de memória entre *micro-batches* por teste (contendo desvio padrão)



Fonte: O autor, 2021

Figura 23 – Duração média de duração entre *micro-batches* por teste (contendo desvio padrão)



Fonte: O autor, 2021