Universidade do Minho
Escola de Engenharia

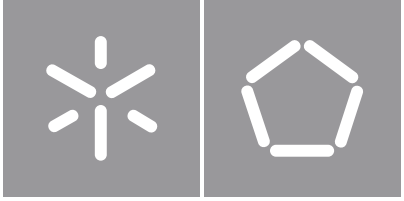
Nelson José Dias Teixeira

HyLake:
Atualização de lagos de dados com granularidade fina

setembro de 2021

HyLake: Atualização de lagos de dados com granularidade fina
Nelson José Dias Teixeira

UMinho | 2021



Universidade do Minho
Escola de Engenharia

Nelson José Dias Teixeira

HyLake: Atualização de lagos de dados com granularidade fina

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação do:
Professor Doutor José Orlando Roque Nascimento Pereira
Doutor Fábio André Castanheira Luís Coelho

setembro de 2021

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Braga, 27 de setembro de 2021

(Local) (Data)

Nelson José Dias Teixeira

(Nelson José Dias Teixeira)

“If the plan doesn’t work, change the plan, but never the goal.”

Agradecimentos

Com a concretização do vigente documento, dou por terminado o respetivo tema de dissertação, concluindo assim mais uma etapa importante da minha vida académica. Esta fase encerra um ciclo de estudos árduo e desafiante, sobretudo em época de pandemia, pelo que não seria completado com sucesso sem a presença de muitas pessoas neste longo percurso de estudos. Como tal, quero agradecer a todas as entidades que fizeram parte desta caminhada durante os últimos cinco anos, tanto a nível institucional como a nível pessoal.

Em primeiro lugar, quero expressar a minha gratidão para com o meu orientador, Professor Doutor José Orlando Roque Nascimento Pereira, que aceitou a minha candidatura para este tema de forma transparente e que se demonstrou sempre disponível no esclarecimento de dúvidas em todos os estágios deste projeto. Para além disso, tenho que destacar o tempo despendido nas várias sessões estipuladas ao longo do ano letivo e, como não poderia deixar de ser, as diversas discussões construtivas e esclarecedoras que tivemos acerca do tópico subjacente a este trabalho. Por fim, queria reconhecer a experiência e o conhecimento manifestados pelo professor, que acabaram por se tornar fulcrais na finalização do mesmo, sobretudo no que toca às decisões mais complexas a nível conceptual e de implementação. Quero também agradecer ao coorientador deste tema, Doutor Fábio André Castanheira Luís Coelho, pelas opiniões emitidas em relação à redação deste documento, tanto na perspetiva de ex-aluno da academia como de um profissional dotado neste ramo da informática.

À instituição de investigação INESCTEC (Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência), em particular ao grupo *HASLab* (*High-Assurance Software Laboratory*), quero destacar a entreajuda e as ótimas condições exibidas durante o desenvolvimento deste trabalho. Também quero saudar o apoio exibido por parte desta instituição e da FCT (Fundação para a Ciência e a Tecnologia) através de um financiamento plurimensal sob a forma de uma bolsa de iniciação à investigação (9034/BII-E_-B4/2021).

Quero também agradecer à comunidade académica da Universidade do Minho pela minha formação e pela forma como fui recebido por todas as partes envolvidas. É mais do que evidente o ambiente saudável vivido nesta casa, desde o momento em que entrei na mesma até à data atual. A bagagem enriquecida que levo desta instituição permite-me enfrentar comodamente o mercado de trabalho, pelo que me resta agradecer novamente à Universidade do Minho por transformar a possibilidade de assegurar um futuro risonho na minha carreira profissional numa realidade.

Não posso também deixar de salientar todos os docentes que integraram a Licenciatura em Ciências

da Computação (LCC) entre 2016 e 2019. Todos eles foram importantes no culminar deste percurso, uma vez que foi nessa fase que me foram dadas as bases de conhecimento expectáveis nas áreas da matemática e da computação. Aos restantes professores pertencentes ao Departamento de Informática da Universidade do Minho e que fazem parte do Mestrado em Engenharia Informática (MEI), obrigado por terem consolidado e expandido o meu conhecimento nos diferentes tópicos intrínsecos a este curso, nomeadamente na área de sistemas distribuídos e de ciência de dados.

Aos meus colegas de curso obrigado pela boa disposição, pela camaradagem, pela amizade e pelas histórias e momentos passados que fizeram deste percurso uma aventura incrível e memorável.

Por fim, quero agradecer a todos os elementos da minha família por me terem apoiado nesta jornada, em particular, aos meus pais que me incentivaram a integrar este curso e que me ofereceram uma oportunidade que nunca lhes tinha sido dada anteriormente.

Resumo

HyLake: Atualização de lagos de dados com granularidade fina

Os lagos de dados, também conhecidos por *data lakes*, suportam a recolha de grandes quantidades de informação em ficheiros imutáveis para processamento analítico. No entanto, tem surgido a necessidade de modificar e atualizar esta informação de forma fiável, seja porque os dados são recebidos de forma incremental (por exemplo, de sensores e outras fontes de eventos) ou para eliminar os mesmos (por exemplo, devido ao RGPD (Regulamento Geral sobre a Proteção de Dados)). As soluções atuais para o fazer não são no entanto ideais: o armazenamento em SGBD (Sistema de Gestão de Bases de Dados) *NoSQL (Not only SQL)* tem um grande impacto no desempenho analítico, enquanto que sistemas baseados em ficheiros, como o *Delta Lake*, permitem apenas atualizações de granularidade grossa.

Neste trabalho aborda-se este problema propondo uma solução híbrida que combina o armazenamento de longo prazo em ficheiros com um armazenamento transitório num SGBD *NoSQL* de forma a obter as vantagens de ambos os sistemas. Para o efeito, é implementado uma prova de conceito usando *Spark*, com ficheiros *Parquet*, e *MongoDB*. Assim, com a introdução deste sistema pretende-se possibilitar a execução de transações frequentes e de granularidade fina para suportar uma carga de trabalho *OLTP (Online Transaction Processing)*. Os resultados experimentais obtidos confirmam que esta proposta obtém desempenho analítico e transacional comparável a cada um dos sistemas isolados.

Palavras-chave: Lagos de dados, Transações, Processamento híbrido transacional-analítico, Bases de dados, Sistemas distribuídos.

Abstract

HyLake: Fine granularity updates to data lakes

Data lakes support the collection of large amounts of information in immutable files for analytical processing. However, there has been a need to reliably modify and update this information, either because data is received incrementally (for example, from sensors and other event sources) or to eliminate them (for example, due to GDPR (*General Data Protection Regulation*)). Current solutions for doing this aren't ideal: storage in NoSQL (*Not only SQL*) DBMS (*Database Management System*) has a big impact on analytical performance, while file-based systems, such as Delta Lake, only allow coarse-grained updates.

This work addresses this problem by proposing a hybrid solution that combines long-term file storage with transient storage in a NoSQL DBMS in order to obtain the advantages of both systems. For this purpose, a proof of concept is implemented using Spark, with Parquet files, and MongoDB. Thus, with the introduction of this system, it's intended to enable the execution of frequent and fine-grained transactions to support an OLTP (*Online Transaction Processing*) workload. The experimental results obtained confirm that this proposal obtains analytical and transactional performance comparable to each of the isolated systems.

Keywords: Data lakes, Transactions, HTAP (*Hybrid Transactional Analytical Processing*), Databases, Distributed systems.

Índice

Apêndices

Índice de Figuras

Índice de Tabelas

Índice de Listagens

Glossário

big data Conceito que descreve o grande volume de dados estruturados e não estruturados que são gerados a cada segundo. 1, 3, 5, 6, 7, 8, 11

cloud computing Modelo de desenvolvimento, baseado na nuvem, que disponibiliza uma quantidade flexível de capacidade de armazenamento e processamento para satisfazer as necessidades inerentes à implementação de um projeto computacional. 1

cluster Designação atribuída a um conjunto de computadores independentes que se encontram ligados a uma rede de modo a desempenhar operações complexas. Ao atuarem sobre as mesmas tarefas computacionais, estes equipamentos, também conhecidos por nós, formam um sistema unificado para dar resposta às necessidades dos seus utilizadores, acelerando substancialmente a capacidade de processamento de informação. 19, 20, 58, 67

online Termo utilizado no âmbito da informática para tanto designar uma ligação ou conexão à rede com sucesso como para indicar que um determinado sistema se encontra operacional ou pronto a usar. 3, 6

query Ação exercida sobre uma base de dados de forma a solicitar informações acerca de uma ou mais tabelas presentes na mesma. 9, 10, 14, 15, 21, 37, 38, 39, 49, 73

Acrónimos e Siglas

ACID Atomicidade, Consistência, Isolamento e Durabilidade 2, 3, 8, 9, 20, 24

API *Application Programming Interface* 19, 36, 40, 47

CPU *Central Processing Unit* 67

CSV *Comma-Separated Values* 15

DBMS *Database Management System* x

ETL *Extract, Transform, Load* 12

GB *Gigabyte* 35, 36, 38, 39, 67

GDPR *General Data Protection Regulation* x

HDFS *Hadoop Distributed File System* 61

HTAP *Hybrid Transactional Analytical Processing* x, xiv, 8, 10, 11, 18, 72, 73

JSON *JavaScript Object Notation* xvii, 3, 15, 22, 23, 29, 31, 33, 40, 41, 45, 48, 50, 53, 72

JVM *Java Virtual Machine* 36

NoSQL *Not only SQL* ix, x, 15, 36, 39, 44, 52, 72

NVMe *Non-Volatile Memory Express* 35

OLAP *Online Analytical Processing* 8, 10, 11, 12, 17, 18, 38, 39, 70, 72, 73

OLTP *Online Transaction Processing* ix, x, 2, 3, 8, 9, 10, 11, 16, 17, 18, 30, 44, 50, 61, 62, 72, 73

RAM *Random Access Memory* 35, 67

RDBMS *Relational Database Management System* 8

RDD *Resilient Distributed Dataset* 19, 20, 36

RGPD Regulamento Geral sobre a Proteção de Dados ix

SGBD Sistema de Gestão de Bases de Dados ix

SQL *Structured Query Language* 8, 19, 20, 43, 54, 55

SSD *Solid State Drive* 35

UDF *User-Defined Function* 38

XML *Extensible Markup Language* 15

Capítulo 1

Introdução

A utilização e vulgarização de aplicações digitais e o aumento do número de utilizadores que as frequentam diariamente acentua gravemente um problema já existente no contexto computacional, isto é, a enorme quantidade de dados por armazenar e processar. De facto, o volume de dados gerados atualmente encontra-se num nível sem precedentes, pelo que o seu tratamento ganha cada vez mais importância. Para demonstrar a influência destas plataformas nos dias de hoje e a quantidade de dados e utilizadores que lhes estão associadas, apresenta-se de seguida o exemplo da rede social *Instagram* [1], atualmente pertencente à empresa *Facebook* [2].

Segundo os estudos estatísticos mais recentes [12], após uma semana a rede social *Instagram* ter sido fundada, cerca de cem mil utilizadores usufruíam esta plataforma para partilhar fotografias e vídeos. Em meados de fevereiro de 2011, aproximadamente dois meses após a sua criação, registou-se nesta aplicação um milhão de utilizadores ativos por mês, evidenciando-se um crescimento significativo. Apesar desta empresa ter sido comprada pela *Facebook* no início de 2012, hoje em dia entre quinhentos a seiscentos milhões de utilizadores usam diariamente a aplicação *Instagram*, sendo que, por mês, este valor já disparou para um marco histórico de um bilião de utilizadores. Tal como seria de esperar, este número de utilizadores traduz-se num volume considerável de informação consumida pelos mesmos que, no caso desta rede social, corresponde à partilha diária de noventa e cinco milhões de fotografias, perfazendo um total de quarenta biliões de fotografias partilhadas desde a sua criação [26].

Em resposta, os últimos anos testemunharam a consolidação do uso do paradigma de computação na nuvem, também conhecido por *cloud computing*. Este foi um dos maiores impulsionadores do conceito *big data*, permitindo não só a ingestão de grandes quantidades de informação como o desbloqueio dos casos de uso de analítica de dados. O armazenamento e processamento de informação em sistemas

[1] https://about.instagram.com
[2] https://about.facebook.com

cloud tem vindo a ser uma solução bastante popular entre as diversas companhias do meio tecnológico, sendo que *Google* [3], *Microsoft* [4] e *Amazon* [5] constituem-se como três das maiores empresas a disponibilizarem este tipo de serviços. *Google Cloud* [11], *Microsoft Azure* [24] e *Amazon Web Services* [25] são alguns exemplos destes serviços e permitem aos seus clientes usufruírem de infraestruturas desejáveis para o desenvolvimento das suas soluções. Esta preferência deve-se sobretudo ao facto de os utilizadores poderem escalar significativamente os recursos de computação e de armazenamento separadamente. Para além disso, aspetos como a disponibilidade, a gestão e monitorização de infraestruturas, a simplicidade, a segurança e o custo reduzido na adesão a estes serviços acabam por tornar ainda mais atrativa a utilização destes sistemas num contexto empresarial. Assim, a virtualização e utilização do modelo de desenvolvimento baseado na nuvem são cada vez mais consideradas como a primeira escolha para o *deployment* das aplicações atuais.

1.1 Problema

Todavia, a enorme quantidade de dados alojados neste tipo de repositórios e a sua desnormalização fazem com que se recorra frequentemente ao armazenamento por objetos ou chave-valor. Por conseguinte, a execução de transações torna-se cada vez mais delicada e complexa, quer a nível de desempenho quer a nível de consistência. Tanto os casos de uso analíticos atuais como os sistemas que os suportam possibilitam a ingestão de dados de várias fontes, criando *pipelines* com dados heterogéneos. Nestes ambientes, a oferta de propriedades transacionais é limitada ou até inexistente, sendo que neste último caso é ainda mais difícil considerarem-se várias fontes de dados em simultâneo. Como tal, é imprescindível a adoção de um sistema que proporciona a oferta de propriedades transacionais ACID (Atomicidade, Consistência, Isolamento e Durabilidade) sobre várias fontes de dados que, por sua vez, recorrem ao armazenamento baseado em objetos de dados. Desta forma, este sistema deve ser capaz de executar várias transações de granularidade fina num ambiente analítico.

1.2 Motivação

As transações descritas anteriormente dizem respeito a um ambiente computacional que utiliza cargas de trabalho *OLTP (Online Transaction Processing)*. Os sistemas associados a este ambiente encarregam-se de registar todas as transações contidas numa determinada operação organizacional. A carga de trabalho envolvida é identificada por uma base de dados que recebe solicitações e alterações de dados de vários utilizadores. Desta forma, evidencia-se um tempo de resposta curto na execução de transações de pequenas dimensões. Caraterísticas como a consulta rápida de dados e a preservação da integridade

[3] https://about.google
[4] https://www.microsoft.com/about
[5] https://www.aboutamazon.com

dos mesmos fazem com que este tipo de carga de trabalho seja o mais indicado em bases de dados tradicionalmente relacionais [13, 14, 16–18]. Máquinas multibanco, sistemas de reservas em restaurantes e aplicações *online* de manutenção de registos na área da saúde são alguns exemplos de sistemas *OLTP*.

Os ambientes analíticos modernos permitem conjugar várias fontes de dados em ambientes com alta concorrência de escritas e leituras. Tentar garantir coerência nos dados a partir de múltiplas origens é uma tarefa difícil, sobretudo num ambiente analítico. De forma a ultrapassar estes desafios computacionais e, consequentemente, garantir a consistência durante o processamento de transações, o repositório de dados considerado necessita de aderir obrigatoriamente às propriedades ACID.

O sistema *Delta Lake* [8] oferece estas mesmas propriedades transacionais, permitindo operar nestas arquiteturas, potencialmente com muitas fontes de dados. Este sistema inclui uma camada de armazenamento que é aplicada sobre lagos de dados de modo a executar transações com as propriedades ACID, lidando com variações de esquemas de dados e a manipulação de metadados de uma forma escalável. Uma tabela *delta* corresponde a uma diretoria num sistema de armazenamento orientado a objetos ou num sistema de ficheiros. Nela existem objetos de dados, sob o formato *Parquet* [6], com o conteúdo da tabela, e um conjunto de registos, codificados na extensão *JSON (JavaScript Object Notation)*, com a respetiva metainformação. Com o uso da ferramenta *Spark* [35] é também possível executar interrogações analíticas, em particular aquelas que envolvem atualizações cirúrgicas aos dados. Para além disso, este utensílio é particularmente útil no processamento distribuído de *big data*. Apesar deste sistema permitir a execução de poucas transações de granularidade grossa para efetuar atualizações em lagos de dados, este apresenta-se como o candidato apropriado à elaboração de uma proposta que envolva muitas transações de pequenas proporções.

1.3 Objetivos e contribuições

Atendendo às caraterísticas do sistema *Delta Lake*, este trabalho tem como objetivo o estudo e a expansão do limite da granularidade dos dados presentes na execução de transações relativas ao último sistema. Como tal, o foco deste tema de dissertação é possibilitar a execução de transações *OLTP* num ambiente analítico com as propriedades da arquitetura do sistema *Delta Lake*. Para isso, elaborou-se um novo sistema que permite a execução de várias transações de granularidade fina em lagos de dados. O surgimento desta solução, isto é, o sistema *HyLake*, tem por base a realização de diversas experiências que estudam a viabilidade do sistema *Delta Lake* na execução de cargas de trabalho *OLTP*. A implementação do sistema *HyLake* corresponde assim a um sistema capaz de suportar transações de atualização de dados utilizados em interrogações analíticas por sistemas distribuídos, como é o caso da ferramenta *Spark*. A última *framework*, ao proporcionar o processamento distribuído de grandes quantidades de informação, ajudará imenso na avaliação da usabilidade e escalabilidade do sistema produzido. Com a realização de leituras e escritas numa base de dados não relacional antecipam-se obter melhores resultados no sistema *HyLake* em relação ao que é observado no sistema *Delta Lake*, onde as operações em

causa envolvem a manipulação de ficheiros. Assim, com a introdução do sistema *HyLake* pretende-se provar que é possível estender a granularidade dos dados na execução de transações analíticas no sistema *Delta Lake*.

1.4 Estrutura da dissertação

O resto do documento está estruturado da seguinte forma:

O Capítulo 2 apresenta o estado de arte relativo ao sistema *Delta Lake*. Nele evidenciam-se alguns conceitos relevantes que complementam o sistema em causa.

Posteriormente são expostos no Capítulo 3 todos os testes intermédios realizados sobre o sistema *Delta Lake*. Nesse processo são indicadas as ferramentas computacionais utilizadas, fazendo-se referência à instalação e configuração das mesmas.

Tendo em conta as experiências realizadas no capítulo anterior, procede-se, no Capítulo 4, à construção da arquitetura do sistema idealizado, isto é, o sistema *HyLake*. Ao longo da descrição desta nova proposta apresenta-se também a sua implementação, incluindo as operações de leitura e de escrita. Para interpretar corretamente a especificação do sistema *HyLake* são referenciados alguns operadores secundários que ajudam a elaborar a solução em questão.

No Capítulo 5 avalia-se o sistema *HyLake*, comparando o seu desempenho com os dos sistemas *Delta Lake* e *MongoDB* [28].

Por fim, no Capítulo 6 conclui-se o vigente trabalho ao apontar o sistema que apresenta o melhor comportamento na execução de transações frequentes e de granularidade fina. Após essa tomada de decisão, é abordado o trabalho futuro deste tema de dissertação.

Capítulo 2

Estado de arte

Esta secção do documento apresenta diversas definições e detalhes computacionais de forma a introduzir o sistema *Delta Lake*. Este sistema é o ponto essencial deste trabalho pelo que será explorado a nível de estruturação e de funcionalidades.

2.1 *Big data*

Big data é a área do conhecimento que estuda como tratar, analisar e obter informações a partir de grandes conjuntos de dados. O termo em questão surgiu em 1997 e foi utilizado para referenciar a manipulação de grandes quantidades de dados não estruturados.

Os principais aspetos de *big data* podem ser definidos à custa da regra dos três *V's*:

- **Volume**: designação relacionada com a grande quantidade de dados gerada por unidade de tempo;
- **Velocidade**: nome que traduz o ritmo com que os dados são produzidos e manipulados;
- **Variedade**: palavra que expressa quer a diversidade das fontes de informação quer a multiplicidade dos formatos de dados. Estas duas caraterísticas aumentam, claro está, a complexidade das análises efetuadas.

A quantidade de dados gerada atualmente tem crescido de forma exponencial. Na Figura 1 salienta-se o crescimento contínuo e evolutivo de dados ao longo dos últimos anos. De notar que os dados apontados entre 2018 e 2025 são valores baseados em predições. Da análise, a quantidade total de dados criados, capturados, copiados e consumidos globalmente aumentou rapidamente desde 2010. Por conseguinte, prevê-se que a capacidade dos sistemas de armazenamento em questão acompanhe este

ritmo de manipulação de dados. Isto traduz-se, de certa forma, na necessidade de controlar a produção desmedida de informação nos sistemas computacionais atuais.

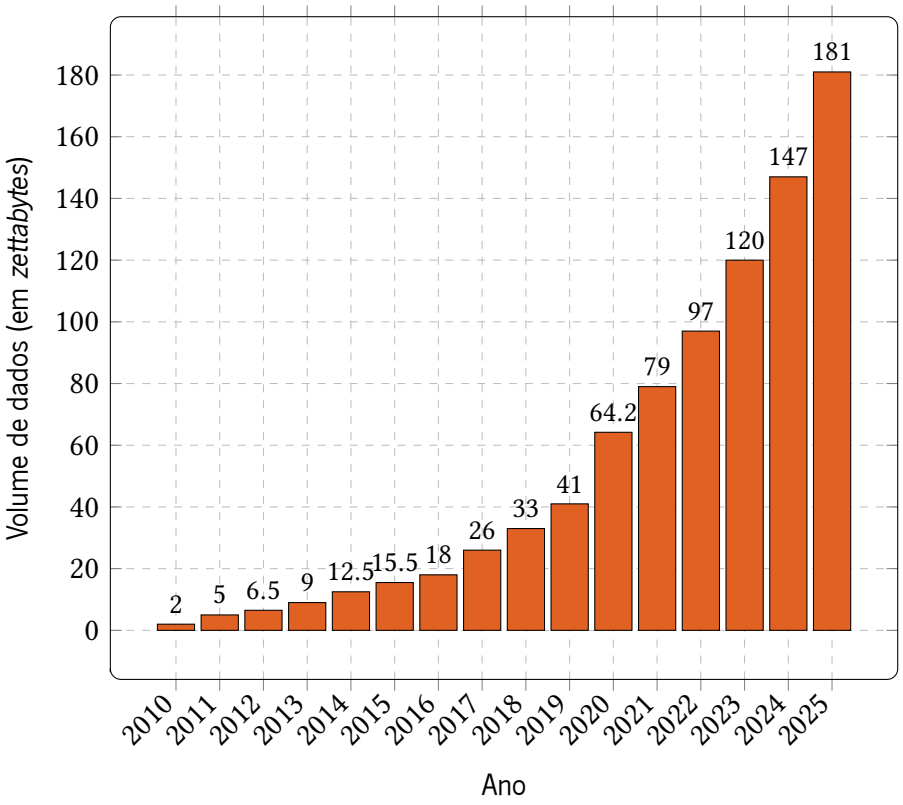


Figura 1: Volume de dados manipulado a nível mundial entre 2010 e 2025. Fonte: *Statista* [31]

Outro dado curioso é o facto da maioria das empresas tecnológicas apenas analisarem uma pequena porção da sua informação, cerca de doze porcento [30]. Ou seja, oitenta e oito porcento da mesma não é devidamente tratada, pelo que a área de *big data* ajudará a executar esta tarefa.

A Figura 2 evidencia a enorme quantidade de informação consumida e produzida hoje em dia nas plataformas digitais mais populares. Tal como se pode observar nesta imagem, existem muitas aplicações que geram informação num curto período de tempo, podendo até haver gastos monetários acentuados durante a sua utilização. As redes sociais e as plataformas de *streaming* são claramente as duas principais categorias das aplicações exibidas, mostrando implicitamente a preferência dos clientes no uso das soluções *online* atualmente disponíveis. Os valores apontados nesta figura transparecem ainda a urgência em adotar mecanismos de processamento e tratamento sobre um conjunto arbitrariamente grande de dados a um nível elevado de escalabilidade.

Atualmente o conceito de *big data* é essencial nas relações socioeconómicas e representa uma evolução nos sistemas de negócio e da ciência. Para além disso, as ferramentas relacionadas com esta área são de grande importância na definição de estratégias de *marketing*. Com elas é possível, por exemplo, aumentar a produtividade, reduzir custos e tomar decisões de negócios mais inteligentes. Desta forma, as companhias que seguem esta estratégia ganham uma vantagem competitiva face à sua concorrência.

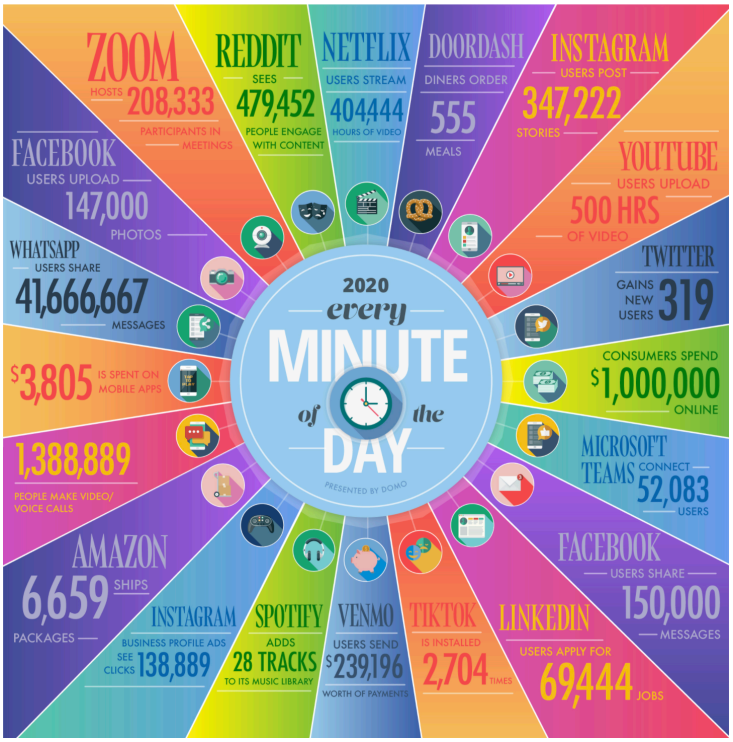


Figura 2: Crescimento incontrolável de dados em 2020. Fonte: *Domo* [19]

Segundo os dados estatísticos mais recentes [1], as empresas que adotam este mecanismo tecnológico obtêm um aumento lucrativo entre oito a dez porcento, reduzindo dez porcento dos seus custos. Para além de se constituir como uma solução inovadora, outras das vantagens associadas a esta área são a melhoria no atendimento aos clientes, segundo as suas necessidades, a deteção de erros e fraudes e, ainda, a melhor agilidade na definição de uma estratégia de negócio. Por estes motivos, pode-se afirmar que o propósito deste conceito passa por gerar valor para os negócios subjacentes.

Apesar do tratamento, da análise e obtenção de grandes conjuntos de dados proporcionar uma vasta coleção de vantagens para as empresas, existem alguns problemas associados com a sua utilização. O maior deles é nada mais nada menos que a privacidade, ou seja, a ameaça representada pelo aumento de armazenamento e integração de informações pessoalmente identificáveis. Aliada a esta adversidade está a segurança da informação. Isto deve-se ao facto de as empresas alojarem este tipo de dados internamente e, consequentemente, se constituírem como potenciais alvos de ataque. Nos dias de hoje já existem políticas bastantes restritivas neste sentido, contudo este tópico requer sempre um tratamento cuidadoso por parte das diversas companhias. Para além destes pontos negativos, existem outros que estão mais relacionados com a natureza de *big data*. Um deles é não haver garantias na qualidade de informação. Isto é diretamente provocado pela existência de enormes quantidades de dados não estruturados. Dada a volumetria da informação gerada, o tempo despendido na sua análise não é útil a curto prazo. Por conseguinte, a realização desta tarefa necessita de ser concretizada por longos períodos de tempo. Por fim, a execução de atualizações rápidas nos dados podem levar a resultados incoerentes,

não assegurando a sua manutenibilidade.

2.2 Modelos de processamento de dados

Hoje em dia, a heterogeneidade encontrada na maioria das aplicações de *big data* deve-se à adequação das últimas a um subconjunto de problemas do mesmo ramo computacional, como o aumento do volume de dados em sistemas que tratam da sua gestão. É possível observar o surgimento de novas tecnologias apropriadas ao atendimento das necessidades de *big data*, como por exemplo a introdução de bases de dados distribuídas. Contudo, atualmente ainda se verifica a persistência de dados em sistemas de gestão de base de dados relacionais, também conhecidos por *RDBMS (Relational Database Management System)*. *Oracle* [17], *MySQL* [16], *Microsoft SQL (Structured Query Language) Server* [14], *IBM Db2* [13] e *PostgreSQL* [18] são alguns exemplos deste tipo de sistemas.

Habitualmente, os *RDBMS* estão associados à consistência e durabilidade dos dados. No entanto, este tipo de sistema pode não aderir a algumas das principais caraterísticas do paradigma da computação em nuvem, isto é, a alta disponibilidade e escalabilidade. Ainda assim, um *RDBMS* lida com a manutenção de alguns aspetos fulcrais, nomeadamente as propriedades ACID, permitindo aos ambientes computacionais que assentam em cargas de trabalho *OLTP* a preservação de operações concorrentes e consistentes. Porém, a realização de análises de *big data* como parte de um sistema de suporte à decisão coloca uma carga adicional sobre os sistemas *OLTP*. Isto deve-se sobretudo ao facto dos últimos se constituírem como a principal fonte de dados neste género de sistemas e, também, por haver divergências a nível dos requisitos inerentes às cargas de trabalho analíticas.

Desta forma, as cargas de trabalho *OLAP (Online Analytical Processing)* são tipicamente selecionadas para superar as anomalias apresentadas pelos sistemas *OLTP* em relação à análise de dados. Estes sistemas são alimentados com dados de diversas fontes, nomeadamente de sistemas *OLTP*, e devolvem os resultados obtidos nas análises dos dados capturados.

Neste secção faz-se a distinção entre os diversos modelos de processamento, em particular o *OLTP*, *OLAP* e *HTAP (Hybrid Transactional Analytical Processing)*. Para cada um dos mesmos é feita uma caraterização detalhada, sendo apresentadas as circunstâncias em que os próprios devem ser utilizados.

2.2.1 *OLTP*

Uma carga de trabalho *OLTP* é baseada na perspetiva transacional de gestão de dados. Esta é caraterizada por ter uma grande quantidade de transações de pequenas dimensões que, por sua vez, realizam operações de inserção, atualização ou remoção de dados. O foco de tais sistemas passa por obter um mecanismo de processamento de consulta rápido, preservando a integridade dos dados ao concluir as operações numa transação. Desta forma, este modelo de processamento é o mais indicado na execução de muitas operações de pequenas proporções sobre todas as linhas de um conjunto de dados. Tratando-se de um sistema responsável pela gestão de bases de dados relacionais, as tabelas

pertencentes às últimas encontram-se totalmente normalizadas, pelo que os dados estão consistentes, bem organizados e não possuem redundância. Com esta estruturação, é possível obter execuções de *queries* eficientes.

Tendo em consideração estes aspetos, pode-se apontar esta classe de sistemas para a realização de tarefas que necessitam de obter respostas rápidas sobre dados que são regularmente utilizados. Em suma, uma carga de trabalho *OLTP* é a mais apropriada para proceder à administração de execução de transações frequentes numa determinada organização.

2.2.1.1 Transação

Uma transação é uma sequência de operações de leitura e escrita que é executada como uma única unidade de trabalho. Depois de iniciada e executadas as operações que a compõem, a transação encontra-se parcialmente confirmada e é feito o pedido de confirmação ao gestor de transações, responsável por determinar o seu resultado. Caso ocorram algumas falhas antes, durante ou depois da execução das operações correspondentes, a transação diz-se falhada. Se a última executar corretamente e produzir o efeito antecipado diz-se confirmada, caso contrário diz-se abortada. Uma transação que tenha iniciado a sua execução, sem a ocorrência de falhas, mas ainda não tenha sido confirmada ou abortada diz-se ativa.

A Figura 3 divulga os diferentes estágios de uma transação, desde a sua génese até ao momento que a mesma termina.

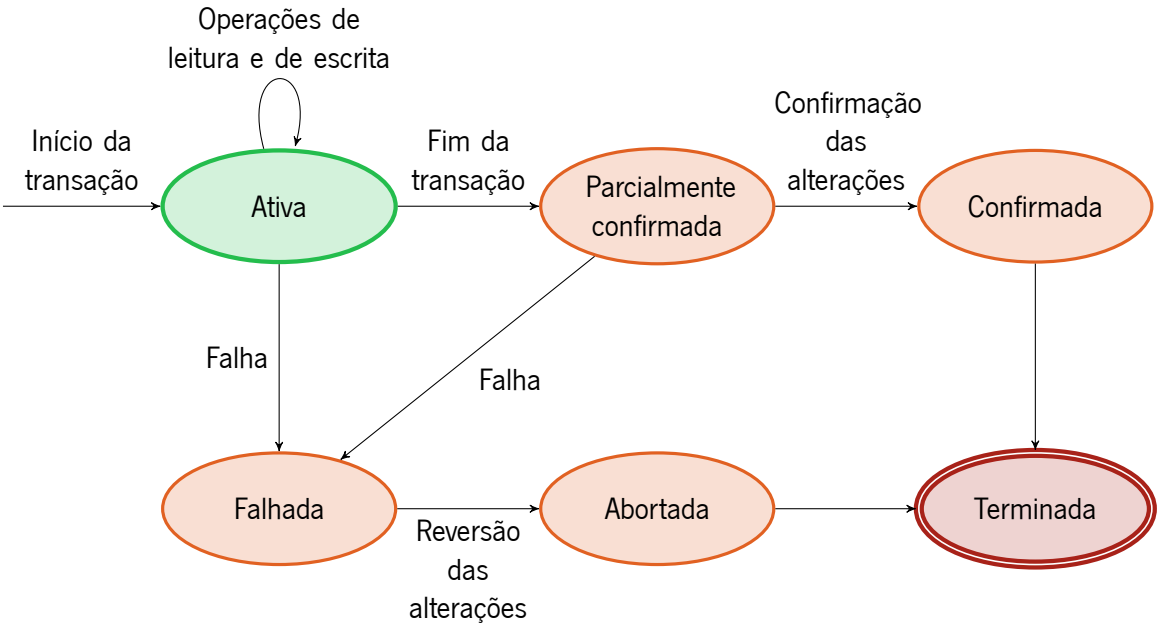


Figura 3: Estado de uma transação

Por definição, uma transação deve possuir quatro propriedades fundamentais, conhecidas como ACID. Exibem-se de seguida as mesmas:

- **Atomicidade**: a transação deve ser uma unidade atómica de trabalho, ou seja, ou todas as operações da transação são executadas ou nenhuma é;

- **Consistência**: quando concluída, a transação deve deixar o sistema num estado coerente, respeitando as restrições da integridade dos dados incluídos;

- **Isolamento**: a execução de uma transação deve ser isolada da execução de outras transações concorrentes, isto é, nenhuma transação deve interferir com outras e as alterações de uma transação incompleta não devem ser visíveis pelas restantes.

- **Durabilidade**: os efeitos de uma transação executada corretamente devem permanecer no sistema, mesmo na presença de falhas.

2.2.2 *OLAP*

Uma carga de trabalho *OLAP* define uma abordagem diferente daquela que foi observada nos sistemas *OLTP*. Esta, por oposição, permite responder a interrogações analíticas. O grupo de sistemas em causa é geralmente considerado para a análise simultânea de dados de diversas dimensões que, por sua vez, são provenientes de múltiplas bases de dados. O modelo de processamento de dados em questão é primordialmente caraterizado por um grande volume de dados onde são realizadas operações de leitura. As tabelas relativas a uma carga de trabalho *OLAP* encontram-se desnormalizadas, pelo que o tempo de resposta observado na execução de *queries* é superior relativamente ao que se verifica num ambiente *OLTP*. De referir que todas as interrogações mencionadas anteriormente apresentam uma natureza seletiva, isto é, contêm sobretudo operações de leitura. Para além disso, estas conservam uma complexidade elevada devido à integração de operações que envolvem agregações de dados de diversas tabelas. Como tal, questões como a integridade de dados não são prioritárias nestes sistemas uma vez que os dados não sofrem grandes alterações durante o seu ciclo de vida.

2.2.3 *HTAP*

A designação *HTAP* define um sistema capaz de lidar eficientemente com cargas de trabalho transacionais e analíticas em simultâneo. Esta nova classe de sistemas de bases de dados evidencia níveis elevados de operações transacionais (*OLTP*), fornecendo ao mesmo tempo uma análise escalável diretamente sobre os dados operacionais (*OLAP*).

Com estas caraterísticas, um sistema *HTAP* disponibiliza simplicidade e velocidade num só local de armazenamento. Com este ambiente, ferramentas como *Spark* [6] e *Kafka* [4] podem ser combinadas para manipular uma base de dados concreta. Por um lado, este modelo requer um nível substancial de atomicidade e consistência na execução de transações. Por outro lado, com a análise de dados, impõe-se a necessidade de examinar rapidamente uma ou mais tabelas presentes em diversas bases de dados.

HTAP, ao possuir uma única base de dados, estabelece uma arquitetura mais simplificada comparativamente ao que se observa nos modelos *OLTP* e *OLAP*, não havendo a necessidade de efetuar cópias de dados previamente armazenados. Para além disso, em vez dos dados serem armazenados em *OLTP*, para transações, e depois transferidos para *OLAP*, para a análise dos mesmos, o sistema *HTAP* possui uma única fonte de informação. Desta forma, tal como a Figura 4 demonstra, assim que o processamento dos últimos seja concluído, estes ficam automaticamente disponíveis para futuras análises.

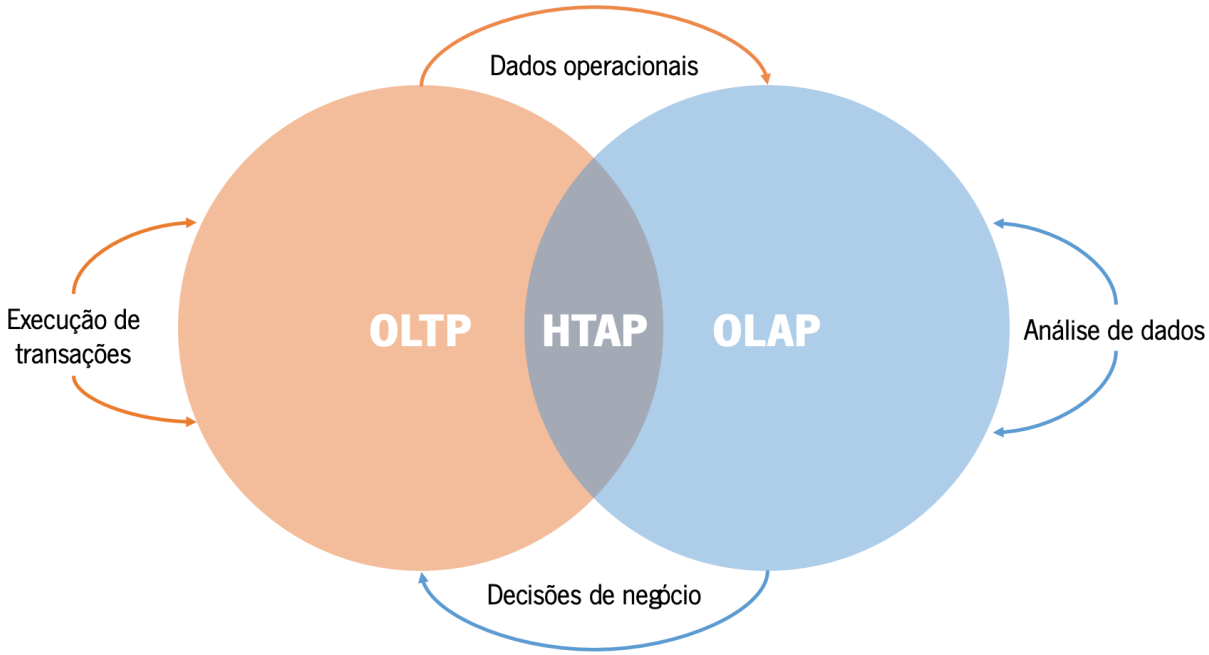


Figura 4: Modelo de processamento *HTAP*

2.3 Modelos de armazenamento de dados

Esta secção expõe dois dos principais modelos de armazenamento de dados no contexto de *big data*, explorando as diferentes estruturas e formatos de dados presentes nestes modelos.

2.3.1 *Data warehouse*

Um *data warehouse* corresponde a um repositório de dados que possibilita a execução de consultas otimizadas e análises avançadas num ambiente relacional e transacional de base de dados. Neste modelo de armazenamento dá-se apenas a retenção de dados estruturados que originalmente se encontram guardados em sistemas transacionais, como é o caso de um sistema *OLTP*. Por definição, estes dados estão unicamente disponíveis para leitura e permanecem inalterados durante o seu ciclo de vida, salvo quando é necessário fazer alguns ajustes aquando do seu carregamento. Por conseguinte, é assegurada uma boa qualidade dos dados, algo que não acontece por exemplo num lago de dados. Desta forma, as

organizações que adotam este modelo conseguem armazenar consistentemente a informação proveniente das suas atividades, tomando melhores decisões sobre os seus negócios. Para dar suporte a estas decisões é comum realizarem-se análises sobre grandes conjuntos de dados com recurso a cargas de trabalho *OLAP*.

A Tabela 1 e a Figura 5 sumariam as caraterísticas e a arquitetura deste modelo, respetivamente.

Caraterística	***Data warehouse***
Tipo de dados	Relacionais, provenientes de sistemas transacionais.
Armazenamento	Captura de informação previamente formatada.
Processamento	Requer maior manutenção e utiliza o processo *ETL (Extract, Transform, Load)*.
Esquema	Concebido antes da implementação e execução de tarefas.
Desempenho e custo	Resultados rápidos de consulta usando armazenamento de alto custo.
Qualidade de dados	Elevada.
Utilizadores	Analistas de dados e de negócio.
Finalidade	Decisões analíticas.

Tabela 1: Caraterísticas de um *data warehouse*

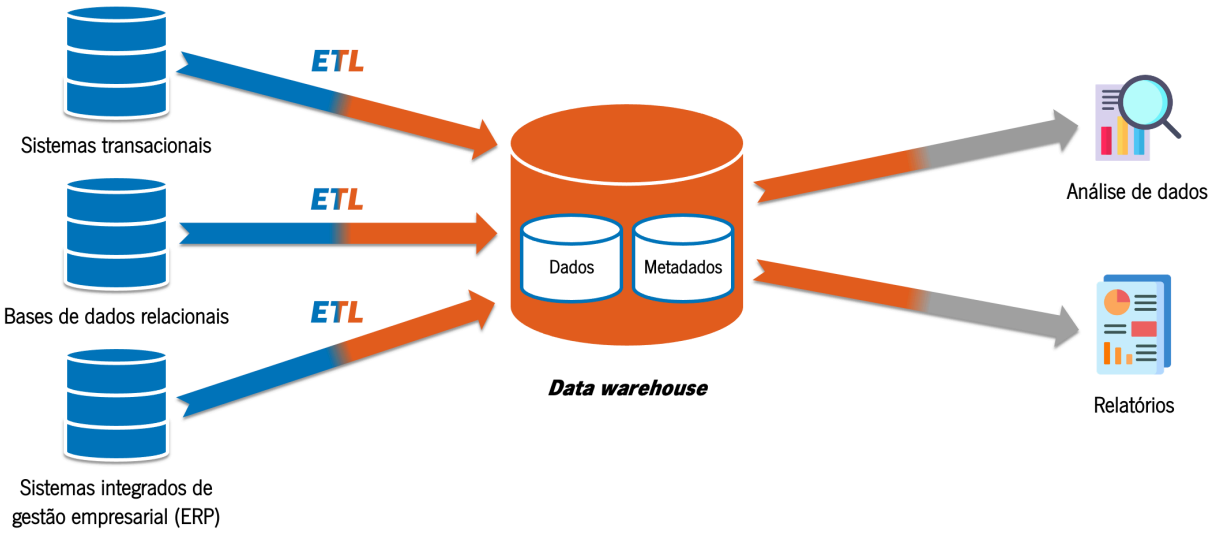


Figura 5: Arquitetura de um *data warehouse*

2.3.2 *Data lake*

Um lago de dados, também conhecido por *data lake*, define uma coleção centralizada com dados diversificados. Estes dados possuem diferentes formatos a nível de estruturação, pelo que os mesmos se podem constituir como estruturados, semi-estruturados ou não estruturados. De forma mais informal, um *data lake* pode ser visto como um contentor de informação de grandes proporções onde se armazenam dados nativamente, independentemente das suas dimensões ou origens. O esquema de dados deste repositório não é definido quando os mesmos são capturados. Estes são guardados até que sejam estritamente necessários para a seleção, organização e execução de determinadas tarefas computacionais. Tal como o termo em inglês transparece, este modelo de persistência permite uma acumulação significativa de informação, sem haver qualquer tipo de tratamento prévio sobre os mesmos. Como tal, é possível produzir cargas de trabalho mistas e, em último caso, aumentar o desempenho analítico sobre este volume de informação.

A Tabela 2 sintetiza as caraterísticas deste modelo.

Caraterística	***Data lake***
Tipo de dados	Estruturados, semi-estruturados e não estruturados.
Armazenamento	Informação persistida no seu formato original, independentemente da sua origem.
Processamento	Execução moderada consoante o tipo de dados.
Esquema	Definido após o armazenamento de dados e identificado no momento de análise.
Desempenho e custo	Resultados moderados de consulta usando armazenamento de baixo custo.
Qualidade de dados	Diversificada, dada a natureza dos dados.
Utilizadores	Engenheiros e cientistas de dados.
Finalidade	Mineração e análise preditiva de dados.

Tabela 2: Caraterísticas de um *data lake*

Quanto à arquitetura de um *data lake*, pode-se inferir a sua simplicidade dado os diferentes tipos de estruturação que este modelo suporta. Por conseguinte, é possível atingir índices consideráveis de escalabilidade, algo que não acontece, por exemplo, em sistemas de armazenamento tradicionais.

2.3.3 Estruturação e formatação de dados

Tal como foi possível constatar anteriormente, existem múltiplas estruturas de dados para diferentes modelos de armazenamento. Esta secção explora não só os detalhes inerentes a estes formatos como

também as vantagens e desvantagens que lhes estão associadas. Fatores como a organização, a flexibilidade e escalabilidade são realçados no momento da sua caraterização. Ao longo destas descrições são também dados alguns exemplos de formatos de dados para sustentar a definição da sua estruturação.

A Figura 6 apresenta alguns exemplos relativos aos três tipos de estruturação de dados presentes nos modelos de armazenamento atuais.

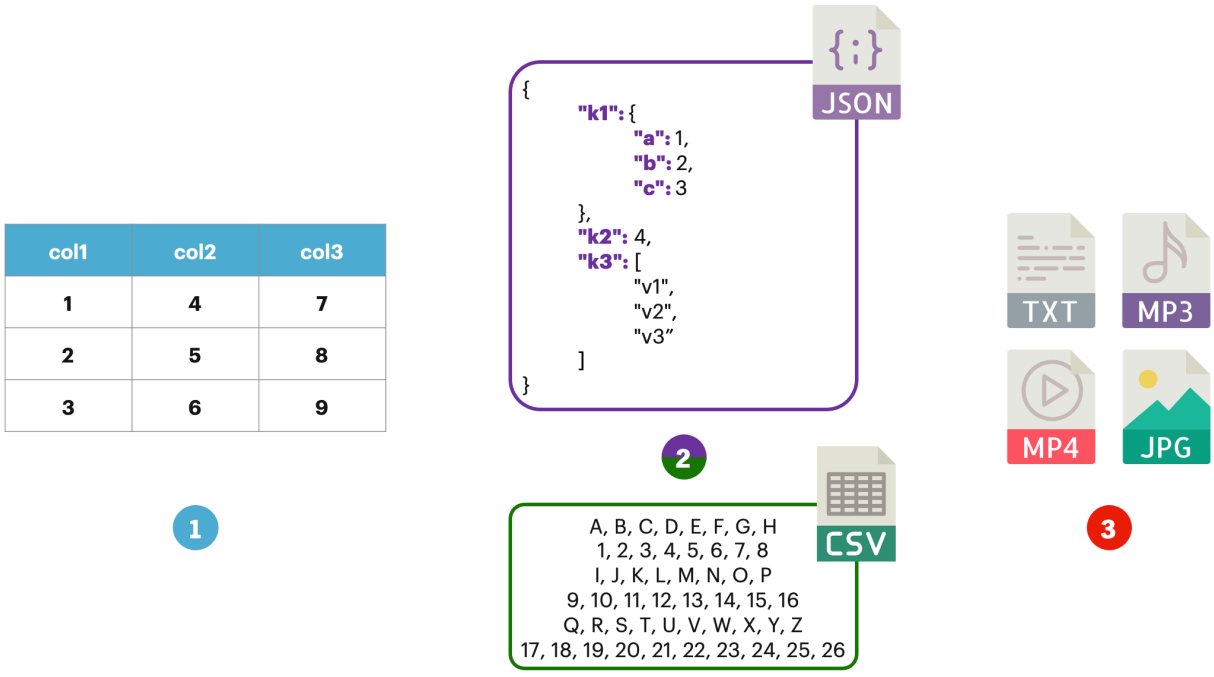


Figura 6: Dados estruturados (1), semi-estruturados (2) e não estruturados (3)

2.3.3.1 Dados estruturados

Os dados estruturados disponibilizam tanto um esquema de dados fixo e bem definido como a existência de registos orientados à linha e à coluna. Esta estruturação apresenta velocidades superiores na execução de *queries* e garante uma persistência de dados eficiente. O último facto transparece, de certa forma, o requerimento de uma menor capacidade de armazenamento. Assim sendo, estes podem ser vistos como registos que podem ser guardados em bases de dados relacionais. Hoje, os dados estruturados são a maneira mais comum e simples de gerir informações. No entanto, estes representam apenas cinco a dez porcento de todos os dados gerados atualmente. São casos desta estruturação de dados os formatos *Parquet* [6], *Avro* [33] e *ORC* [5].

Tal como a Figura 6 clarifica, os dados estruturados são pouco flexíveis, possuem o maior nível organizacional de todas estruturações de dados referidas e são persistidos num formato bem definido. Nesta estruturação é nativamente disponibilizado pelo gestor de uma base de dados relacional os processos relativos ao controlo de concorrência e à gestão de transações.

2.3.3.2 Dados semi-estruturados

Ficheiros com extensão *JSON*, *CSV (Comma-Separated Values)* ou *XML (Extensible Markup Language)* adotam a semi-estruturação de dados. Este tipo de estrutura representa entre a cinco a dez porcento dos dados gerados atualmente. Apesar destes não possuírem um esquema de dados por omissão, é possível deduzir o seu conteúdo pela existência de registos delimitados por algum tipo de separador (vírgula, dois pontos, entre outros). Assim, quando comparado com o caso anterior (Secção 2.3.3.3), esta estruturação, ainda que não seja rígida, garante uma melhoria substancial na rapidez do respetivo processamento de dados e, consequentemente, obtêm-se execuções de *queries* mais velozes. Por conseguinte, adquire-se um armazenamento eficiente e com melhor desempenho. Bases de dados não relacionais, também conhecidas pelo termo *NoSQL (Not only SQL)*, fazem parte deste tipo de estruturação. De salientar que este género de dados pode ser facilmente convertido para uma estrutura fixa, como é o caso dos dados estruturados (Secção 2.3.3.1).

Dito isto, pode-se concluir que os dados semi-estruturados têm um grau organizacional intermédio, dispõem uma flexibilidade superior aos dados não estruturados e são mais simples de escalar. Todavia, não integram o controlo de concorrência e incorporam uma gestão de transações adaptada.

2.3.3.3 Dados não estruturados

Os dados não estruturados correspondem a cerca de oitenta porcento dos dados produzidos atualmente. Estes dados possuem uma formatação heterogénea e contêm alguma redundância e ambiguidade, pelo que o seu processamento é difícil. Nesta estrutura de dados, não existe a noção de registo, isto é, não há um esquema de dados predefinido. Consequentemente, não é possível exibir este tipo de dados em linhas, colunas ou em bases de dados relacionais. Apesar dos próprios possuírem uma estrutura interna, o conteúdo dos mesmos não é incorporável numa base de dados sem a ocorrência de processamento. Desta maneira, este tipo de dados é extraído e retido em *data lakes* para desempenhar análises preditivas. Ficheiros com extensão *TXT*, isto é, ficheiros com conteúdo textual, são exemplos deste formato. Tendo em consideração a volumetria deste tipo de dados, é possível inferir que a ausência de uma estrutura fixa prejudica o processamento dos mesmos, ainda que exista flexibilidade.

Em síntese, os dados não estruturados fornecem principalmente informações qualitativas, pelo que não podem ser mapeados num modelo de dados predefinido. Estes constituem-se como a estrutura de dados com maior flexibilidade e escalabilidade, uma vez que não adotam um esquema de dados concreto. Por fim, nos dados não estruturados não se verifica qualquer tipo de controlo de concorrência, pelo que também não há nenhuma gestão a nível de execução de transações.

2.3.4 Modelos lógico e físico de armazenamento de dados

Nesta secção exibem-se os modelos lógico e físico de armazenamento de dados. Neles é possível ter um visão externa de como se procede à persistência de informação num sistema de armazenamento

local ou até remoto. No que toca ao plano lógico, é possível observar o conteúdo original de um conjunto de dados e inferir a sua dimensão e esquematização. Quanto ao plano físico, transparece-se a orientação dos dados persistidos num determinado local de armazenamento.

Na Figura 7 é possível identificar três tipos de orientações no armazenamento de dados relativamente ao plano físico: orientação à linha, à coluna e, ainda, uma estratégia híbrida. Na primeira dá-se a partição horizontal da tabela, ou seja, cada registo presente numa determinada linha é guardado segundo essa orientação, preservando a sua ordem. Na segunda ocorre a partição vertical da tabela, isto é, cada registo presente numa determinada coluna é armazenado de acordo com essa orientação, mantendo a respetiva ordem. Por fim, na terceira verifica-se uma combinação das estratégias anteriores, percorrendo-se e extraindo-se porções de registos presentes quer numa determinada linha quer numa coluna em específico.

Plano lógico

	Coluna A	Coluna B	Coluna C
Linha 0	A0	B0	C0
Linha 1	A1	B1	C1
Linha 2	A2	B2	C2
Linha 3	A3	B3	C3
Linha 4	A4	B4	C4
Linha 5	A5	B5	C5

Plano físico

Linha

A0	B0	C0	A1	B1	C1	A2	B2	C2
A3	B3	C3	A4	B4	C4	A5	B5	C5

Coluna

A0	A1	A2	A3	A4	A5	B0	B1	B2
B3	B4	B5	C0	C1	C2	C3	C4	C5

Híbrida

A0	A1	A2	B0	B1	B2	C0	C1	C2
A3	A4	A5	B3	B4	B5	C3	C4	C5

Figura 7: Esquematização de modelos físicos na persistência de dados

2.3.4.1 Orientação à linha

Este tipo de orientação no armazenamento de dados traduz-se na partição horizontal de um conjunto de dados. A Figura 8 ilustra a orientação à linha na persistência de dados. Tendo em consideração os aspetos mencionados na Secção 2.2.1, é simples concluir que esta estratégia de armazenamento adequa-se perfeitamente a uma carga de trabalho *OLTP*. Alterações sobre uma determinada tabela, isto é, operações como a inserção, atualização e remoção de registos afetam as linhas que contêm a informação. Por outro lado, pode-se também inferir que esta orientação não é apropriada para um ambiente onde hajam poucas

operações de grandes dimensões sobre um subconjunto de todas as colunas de um repositório de dados, como é o caso de uma carga de trabalho *OLAP*.

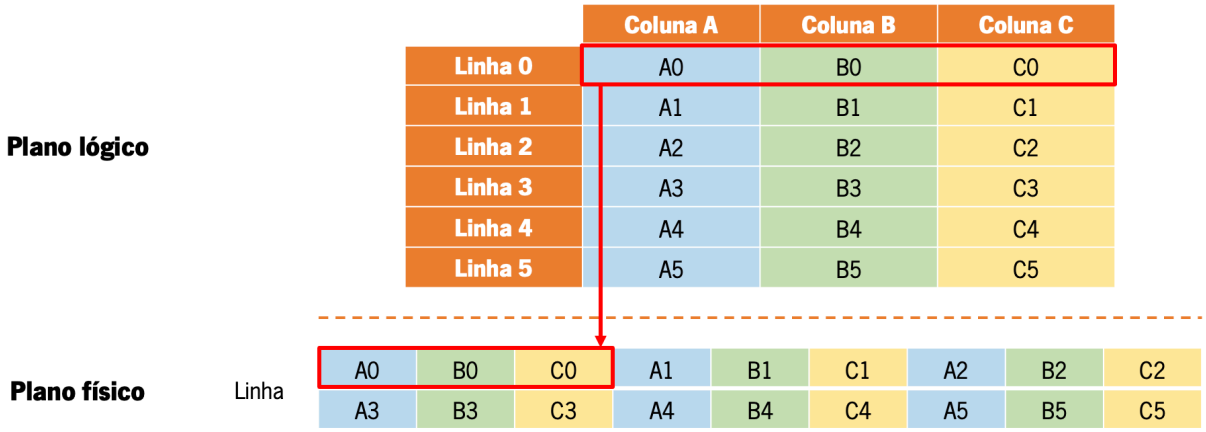


Figura 8: Modelo físico de armazenamento de dados orientado à linha

2.3.4.2 Orientação à coluna

Tal como a Figura 9 demonstra, este tipo de orientação exprime-se, de forma simplificada, na partição vertical de um conjunto de dados. Recuando ao que foi exposto na Secção 2.2.2, pode-se afirmar que esta estratégia alinha-se com as caraterísticas de uma carga de trabalho *OLAP*. Uma vez que o interesse da última passa pela obtenção de uma porção de todas as colunas de um conjunto de dados, a execução de projeções torna-se bastante eficiente devido à contiguidade dos últimos. Dado que os registos de uma determinada coluna são sequencialmente persistidos, existe a possibilidade de efetuar uma compressão dos mesmos. Em contrapartida, esta orientação não se adequa a uma carga de trabalho *OLTP* porque, para a realização de operações que envolvam a inserção de registos, é necessário a adição dos mesmos em diferentes locais do armazenamento.

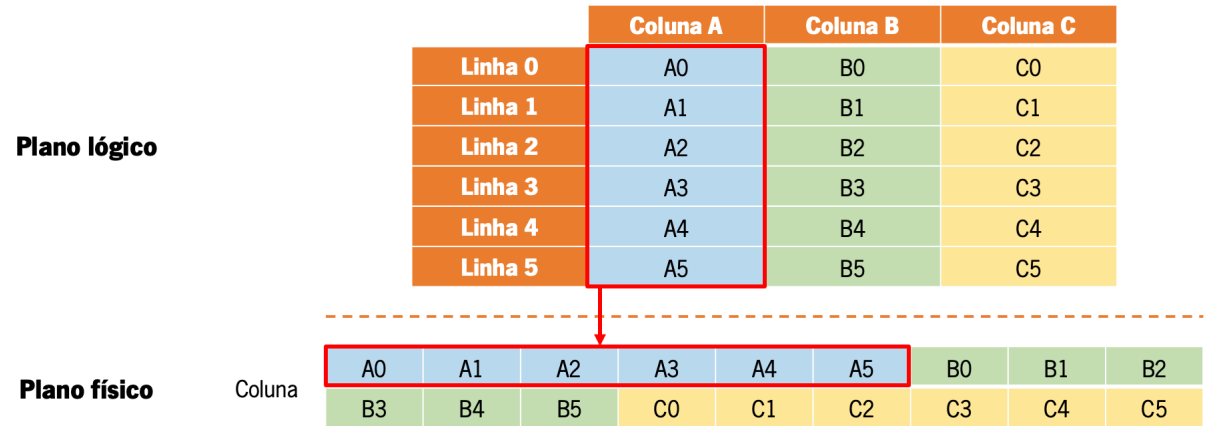


Figura 9: Modelo físico de armazenamento de dados orientado à coluna

2.3.4.3 Orientação híbrida

Retomando o seu conceito, este tipo de orientação no armazenamento de dados representa uma junção das caraterísticas inerentes à orientação à linha (Secção 2.3.4.1) e à coluna (Secção 2.3.4.2). Desta forma, dá-se tanto a partição horizontal como a partição vertical da tabela. No caso da Figura 10, a partição horizontal é invocada a cada três linhas, sendo-lhe posteriormente aplicada a partição vertical. Este procedimento de armazenamento de dados é usado em alguns formatos, como *Parquet* e *ORC*, e oferece tanto as propriedades de localização de registos relativas a uma carga de trabalho *OLTP* como a contiguidade dos últimos num ambiente *OLAP*. Assim, a orientação híbrida é apropriada a uma carga de trabalho *HTAP*.

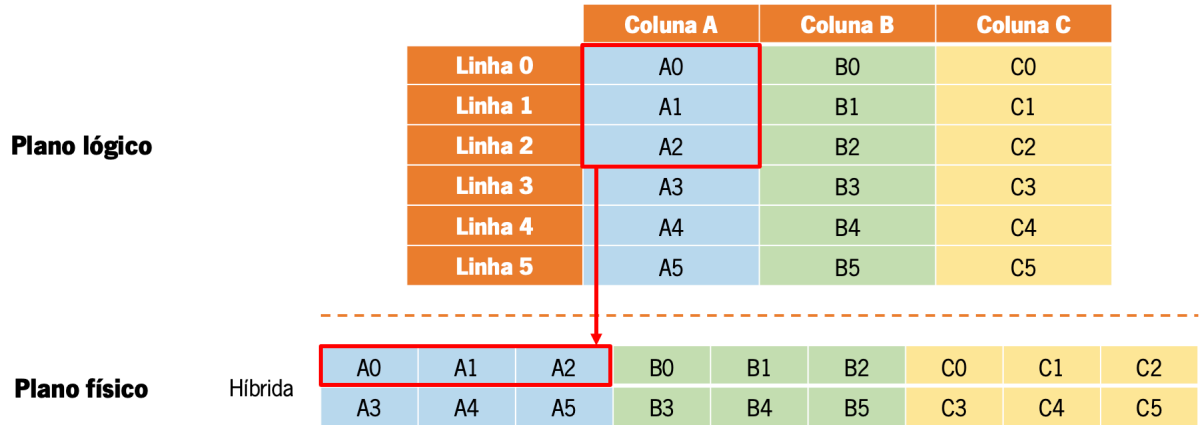


Figura 10: Modelo físico de armazenamento de dados com orientação híbrida

2.4 *Spark*

O paradigma *MapReduce* [10], introduzido pela *Google* em 2004, é um modelo que visa a computação paralela de grandes volumes de dados segundo a estratégia *divide and conquer*. O surgimento deste paradigma permitiu resolver alguns problemas computacionais, como por exemplo o balanceamento de cargas de trabalho e a movimentação de dados observada durante a fase de processamento. A estratégia adotada no processamento de um conjunto de dados arbitrário pode ser decomposta em quatro etapas: divisão, mapeamento, agrupamento e ordenação e, por fim, redução. Inicialmente é realizada a divisão do conjunto de dados considerado em diversos segmentos com aproximadamente a mesma dimensão. De seguida, cada bloco de dados é mapeado num ou mais pares chave-valor, sendo os mesmos agrupados e ordenados segundo a primeira componente. Por fim, para cada chave de cada par é aplicada a redução dos respetivos valores. A Figura 11 ilustra cada uma das etapas descritas.

Apesar deste modelo ser caraterizado pela sua escalabilidade, simplicidade, flexibilidade e, ainda, pela sua tolerância a faltas, é possível apontar alguns defeitos. Tal como a Figura 11 transparece, existem demasiadas escritas em disco ou em memória entre os vários estágios de processamento. Para além

disso, a composição destas etapas não é feita de uma forma livre. Com a ausência de abstrações de memória distribuída, observa-se ineficiência na execução de tarefas que necessitam de reutilizar cálculos intermédios em diferentes etapas de computação.

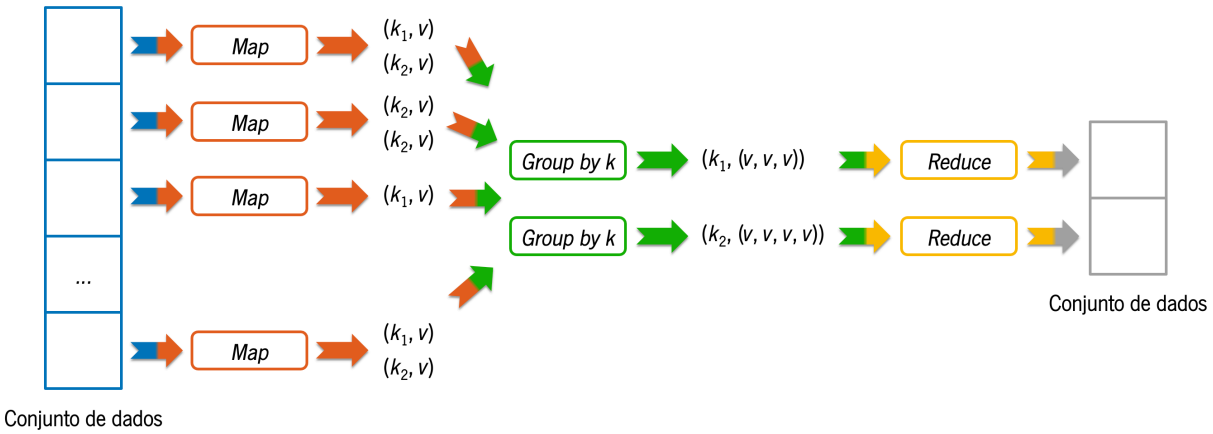


Figura 11: Paradigma *MapReduce*

De forma a ultrapassar estas adversidades, foi proposta uma nova solução, denominada por *Spark* [35], que introduz o conceito de *RDD (Resilient Distributed Dataset)*. Esta abstração de memória distribuída corresponde a uma coleção de objetos particionados que podem ser reconstruídos caso uma das partições seja perdida. Esta coleção é computada e armazenada pelos nós que constituem um determinado *cluster*. Com estas caraterísticas, os utilizadores desta ferramenta podem preservar *RDDs* em memória pelos diversos nós de um *cluster* e, desta forma, reutilizá-los em fases posteriores de processamento. *Spark* é por isso um motor de processamento distribuído que possibilita a análise de dados em grande escala. Esta ferramenta fornece *APIs (Application Programming Interfaces)* em diversas linguagens de programação, nomeadamente em *Python* [34] e *Scala* [29], para garantir o paralelismo de dados e a tolerância a faltas em *clusters* de variadas dimensões. Ainda assim, *Spark* não oferece, por exemplo, um sistema de gestão de ficheiros. A quantidade de memória usufruída na execução de tarefas e a ausência de suporte no processamento de dados em tempo real são duas das suas principais limitações. Uma vez discutida a sua natureza, expõe-se de seguida os diferentes módulos disponibilizados pela ferramenta *Spark*:

- ***Spark SQL***: biblioteca usada no processamento de dados estruturados;
- ***MLlib***: módulo utilizado em aprendizagem automática;
- ***GraphX***: biblioteca usada no processamento de grafos;
- ***Structured streaming***: módulo utilizado no processamento de fluxo (*stream*) e na computação incremental.

Dos quatro módulos apontados nesta enumeração, apenas a biblioteca *Spark SQL* [7] é utilizada na construção da solução deste trabalho, dada a sua utilidade no armazenamento e na manipulação de dados estruturados. No caso do sistema elaborado, estes dados são usados sob a forma de um *dataframe*. Esta estrutura de dados pode ser vista como uma tabela presente numa base de dados relacional, onde cada coluna possui um único nome e tipo de dados. Na perspetiva da ferramenta *Spark*, um *dataframe* representa uma coleção de dados distribuída que inclui diversas otimizações para conseguir atingir um melhor desempenho comparativamente a um *RDD*. Durante a distribuição de dados pelos nós de um *cluster*, a ausência de técnicas otimizadas na serialização de dados traduz um dos contratempos verificados em aplicações *Spark* que recorrem ao uso de *RDDs*. Uma vez que a informação contida num *dataframe* é guardada sob o formato binário, não existe a necessidade de executar a serialização dos dados envolvidos, pelo que é recomendada a sua utilização.

2.5 *Delta Lake*

O modelo de armazenamento de dados subjacente a um lago de dados é atualmente utilizado para a persistência de informação com diferentes estruturas. Como tal, é comum verificar-se a coleção da mesma, numa primeira fase, e posteriormente a sua persistência. Com estes dados são subsequentemente exercidas análises em contextos de ciência de dados ou até de aprendizagem automática. Todavia, a qualidade inicial dos dados coletados é bastante reduzida, sobretudo pela sua heterogeneidade. Desta forma, não é possível assegurar a sua qualidade, pelo que este processo acaba por se tornar inútil para o efeito pretendido, ou seja, extrair valor dos dados previamente armazenados. A esta contrariedade juntam-se também alguns aspetos negativos, como por exemplo o facto de num lago de dados não existir atomicidade e, ainda, consistência ou isolamento dos dados. Na eventualidade de ocorrerem falhas durante a execução de uma tarefa computacional, os dados poderão terminar numa vista incoerente.

Com a introdução do sistema *Delta Lake* [8] pretende-se combater todos os aspetos negativos que foram expostos anteriormente em relação a um lago de dados. A principal caraterística que distingue este sistema de um lago de dados reside na inclusão de uma camada de armazenamento onde é possível executar transações que integram as propriedades ACID (Secção 2.2.1.1). Assim, conseguem-se obter ambientes transacionais mais coerentes num âmbito de execução analítica distribuída. Para além disso, este utiliza a ferramenta *Spark* como o motor de computação distribuída. Com a integração da mesma, dá-se a unificação de tarefas que envolvem o processamento de dados em tempo real (*streaming*) e em lotes (*batch*). Ainda acerca da incorporação da ferramenta referida, o sistema *Delta Lake* utiliza-a para proceder a uma manipulação escalável dos metadados subjacentes. Como tal, realizam-se escritas pontuais derivadas de operações analíticas de uma forma coerente, algo que não acontece naturalmente num lago de dados. Outra funcionalidade importante é a capacidade que o mesmo tem em lidar com variações de esquema de dados. Este último tópico é tratado automaticamente pelo sistema *Delta Lake* para evitar a inserção de registos inválidos durante o processamento de uma tarefa. Por fim, o sistema

em questão disponibiliza aos seus utilizadores o acesso a todas as versões dos dados persistidos, sendo possível observar o histórico de todas as operações efetuadas e, ainda, executar *queries* sobre uma versão em específico.

2.5.1 Arquitetura

A Figura 12 ilustra a categorização de dados segundo a sua qualidade. Esta divisão é efetuada por parte do sistema *Delta Lake* para incrementalmente melhorar a qualidade dos dados até que estes estejam disponíveis para serem consumidos. Nesta segmentação evidenciam-se três classes de dados distintas. Na primeira, a categoria bronze, apresentam-se todos os dados que ainda possuem a sua estrutura original, ou seja, que permaneceram inalterados. Com o armazenamento destes, ainda que não tenham a estrutura desejada, consegue-se preservar toda a informação desde o momento da sua génese. Para além disso, ao proceder-se à persistência inicial destes dados, evita-se prematuramente o pré-processamento dos mesmos. Quanto aos dados pertencentes à categoria prata, estes, à semelhança da classe anterior, também não se encontram prontos para serem utilizados. Contudo, já contêm algum tipo de tratamento, como por exemplo filtragens, incorporação de esquemas de dados ou até junções com outros conjuntos do mesmo tipo. Desta forma, este grupo já possui uma ligeira melhoria a nível de estruturação face ao que se observou na categoria bronze. Esta categoria também pode ser visualizada como um classe intermédia onde se podem realizar processos de verificação sobre os dados de forma a detetar algumas anomalias. Por fim, na classe ouro, existe uma coletânea de dados totalmente processados para posteriormente serem consumidos em operações de análise.

Relativamente à execução de operações do tipo *streaming*, estas movimentam os dados através do sistema *Delta Lake*, percorrendo as diversas classes mencionadas anteriormente. Com este comportamento é possível eliminar a gestão do escalonamento das respetivas tarefas computacionais.

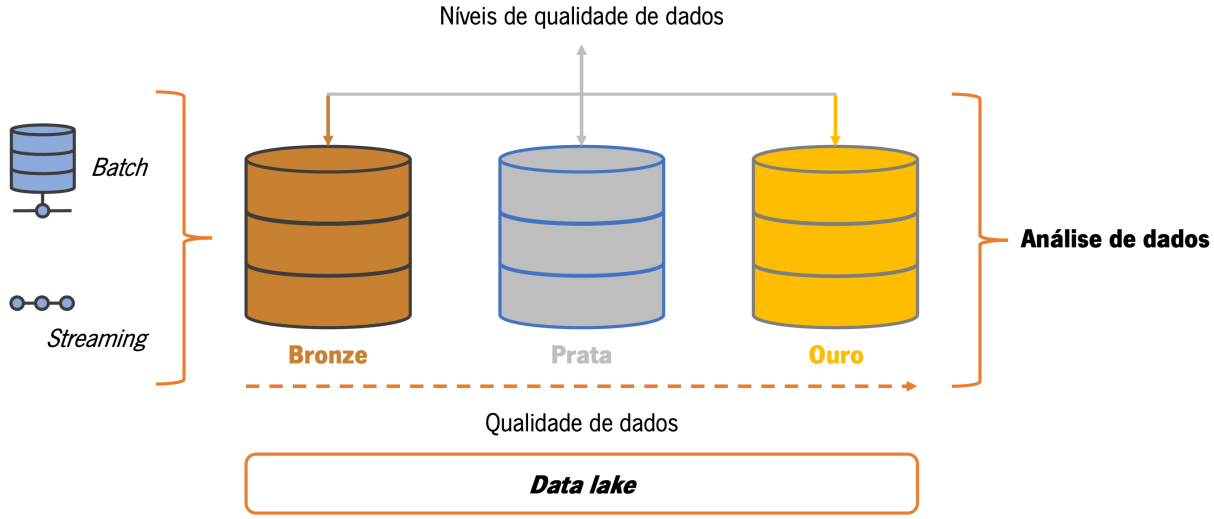


Figura 12: Processo de categorização de dados no sistema *Delta Lake*

Em síntese, a arquitetura deste sistema passa por proporcionar aos *data lakes* aspetos como a confiabilidade e serialização na execução de transações. Para tal, independentemente do tipo de operações, os dados são categorizados segundo três níveis de qualidade e, assim que estejam prontos para serem utilizados, são enviados para a concretização de análises.

2.5.2 Formato de armazenamento

No sistema *Delta Lake* uma tabela é definida por uma diretoria presente num sistema de armazenamento orientado a objetos ou num sistema de ficheiros tradicional. Nela existem objetos de dados, sob o formato *Parquet*, com o conteúdo da tabela e um conjunto de registos, codificados na extensão *JSON*, com todas as transações efetuadas até ao momento (com pontos de verificação ocasionais). Os utilizadores atualizam esta estrutura de dados usando protocolos de controlo de concorrência otimistas, adaptados às caraterísticas dos sistemas armazenamento de objetos em nuvem.

A Figura 13 mostra o formato de uma tabela, intitulada por *deltaTable*, caraterística de um sistema *Delta Lake*.

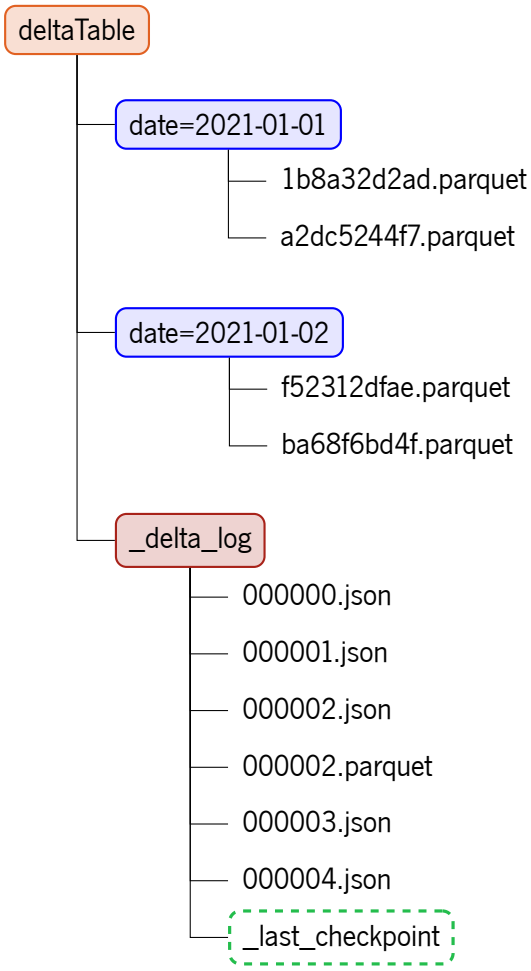


Figura 13: Armazenamento de objetos numa tabela *delta*

2.5.2.1 Objetos de dados

O conteúdo da tabela em causa é armazenado em objetos de dados com o formato *Parquet*. Estes possuem nomes exclusivos e podem ser organizados em pastas através da convenção de partição do utensílio computacional *Hive* [3]. Na Figura 13 a tabela é particionada segundo uma determinada coluna, isto é, a coluna *date*. Desta forma, para cada data, existem objetos de dados em diretorias distintas. Por norma, o atributo escolhido para realizar a partição da tabela é aquele que melhor identifica a maioria dos registos persistidos.

A seleção do formato *Parquet* para a persistência de dados deve-se ao facto de o mesmo ser orientado à coluna, oferecendo diversas atualizações de compactação. Para além disso, este formato já tem implementações de desempenho em muitos mecanismos e suporta tipos de dados aninhados para dados semi-estruturados. A inclusão deste formato *open-source* possibilita ainda ao sistema *Delta Lake* tirar proveito das atualizações mais recentes do mesmo. Outros formatos como o *ORC* poderiam ter sido escolhidos para o efeito, contudo o formato *Parquet* possui o suporte mais maduro na ferramenta *Spark*.

2.5.2.2 Registos transacionais

De maneira a guardar todas as modificações exercidas sobre uma tabela *delta*, é inicialmente criada uma sub diretoria, denominada por *_delta_log*. Dentro dela existe uma sequência de objetos *JSON* com identificadores numéricos e crescentes, a começar pelo ficheiro *000000.json*, para armazenar os registos transacionais. Cada ficheiro *JSON* exprime uma versão distinta da tabela e contém o nome de todos os objetos de dados que lhe dizem respeito. A estes juntam-se os ficheiros com extensão *Parquet* que traduzem os pontos de verificação ocasionais da tabela. Assim, é possível reconstruí-la com as versões anteriores até esse ponto. A Figura 14 exibe a ocorrência de um ponto de verificação que resume o estado transacional de uma tabela *delta* até à sua segunda versão.

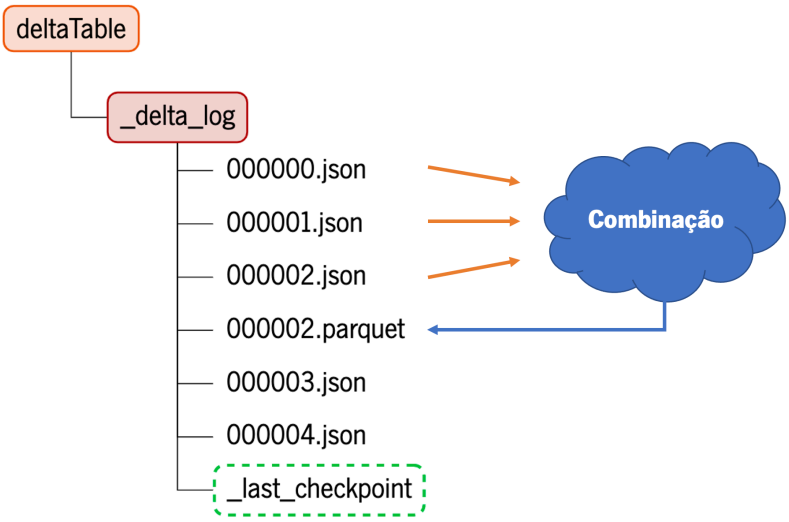


Figura 14: Pontos de verificação numa tabela *delta*

Cada registo transacional posterior ao inicialmente criado, como é o caso do ficheiro *000002.json*, contém uma lista de ações a serem aplicadas à versão anterior da tabela para gerar a próxima. Assim, os registos transacionais são a chave para compreender corretamente o sistema *Delta Lake*. São neles onde se refletem muitas das suas principais caraterísticas, nomeadamente as transações ACID e a manipulação escalável de metadados. Esta coleção de registos é usada para saber, de forma fidedigna, o estado atual de uma tabela *delta* por múltiplos utilizadores e para garantir atomicidade na execução de transações.

Desta forma, uma tabela *delta* pode ser vista como o resultado da aplicação de um conjunto de ações. As ações em causa podem ser as seguintes:

⇒ ***Change Metadata*:** a ação *metaData* altera na íntegra os metadados atuais da tabela sempre que haja uma nova versão da mesma. Na primeira versão de uma tabela *delta* é garantida a existência de uma ação deste tipo no respetivo registo transacional. Desta forma, existe no máximo uma ação deste género numa determinada versão de uma tabela *delta*. Os metadados correspondem a uma estrutura de dados que contém o esquema e a designação da tabela, os nomes das colunas de partição (caso existam), o formato de armazenamento dos objetos de dados e outras opções de configuração. Por convenção, os objetos de dados associados a uma tabela *delta* encontram-se codificados no formato *Parquet*. Outro parâmetro relevante é o identificador da tabela *delta* sobre o qual todos os restantes campos dizem respeito. De forma mais concreta, a Tabela 3 apresenta a informação de todos os campos associados a esta ação.

Nome do campo	Descrição
`id`	Identificador único de uma tabela *delta*.
`name`	Identificador de uma tabela *delta* disponibilizado pelo utilizador.
`description`	Descrição de uma tabela *delta* indicada pelo utilizador.
`format`	Especificação da codificação dos ficheiros armazenados numa tabela *delta*.
`schemaString`	Esquema de dados de uma tabela *delta*.
`partitionColumns`	Lista que contém os nomes das colunas pelas quais os dados devem ser particionados.
`createdTime`	Instante em que a ação em causa foi criada, em milissegundos.
`configuration`	Dicionário que contém as opções de configuração para a ação em questão.

Tabela 3: Esquema com o conteúdo relativo à ação *metaData*

Para tornar ainda mais percetível a aplicação desta ação, exibe-se na Listagem 1 um exemplo representativo desta ação num registo transacional. Neste exemplo os campos *name* e *description* não são

indicados uma vez que a sua especificação é de teor facultativo.

```
{
    "metaData": {
        "id": "af23c9d7-fff1-4a5a-a2c8-55c59bd782aa",
        "format": {
            "provider": "parquet",
            "options": {}
        },
        "schemaString": "...",
        "partitionColumns": [],
        "configuration": {
            "appendOnly": "true"
        },
        "createdTime": 1612555693855
    }
}
```

Listagem 1: Ação correspondente à modificação dos metadados presentes numa tabela *delta*

⇒ ***Add or Remove Files*:** as ações relativas à adição e remoção de ficheiros são usadas para modificar os dados presentes numa tabela *delta*. Com esta ação é possível determinar o conjunto de objetos de dados que a compõem.

A Tabela 4 e a Listagem 2 evidenciam, respetivamente, o esquema de dados relativo a esta ação e um exemplo elucidativo desta mesma estrutura num registo transacional.

Nome do campo	Descrição
`path`	Caminho relativo à localização da tabela *delta* no sistema de ficheiros onde se encontra o ficheiro adicionado à última.
`partitionValues`	Dicionário que estabelece uma relação entre o nome de partição da coluna presente numa tabela *delta* e o respetivo valor.
`size`	Tamanho do ficheiro adicionado, em *bytes*.
`modificationTime`	Momento em que o ficheiro foi criado, em milissegundos.
`dataChange`	Campo que exprime a existência de uma modificação do estado atual da tabela *delta*.
`stats`	Estatísticas associadas aos dados presentes no ficheiro adicionado à tabela *delta*.
`tags`	Dicionário que contém metadados sobre o ficheiro adicionado à tabela *delta*.

Tabela 4: Esquema com o conteúdo relativo à ação *add*

```
{
    "add": {
        "path": "date=2021-01-01/part-000...c000.gz.parquet",
        "partitionValues": {
            "date": "2021-01-01"
        },
        "size": 841454,
        "modificationTime": 1512909768000,
        "dataChange": true
        "stats": "{\"numRecords\":1,\"minValues\":{\"val..."
    }
}
```

Listagem 2: Ação correspondente à adição de um ficheiro numa tabela *delta*

De salientar que o campo *path* presente neste tipo de ações atua como uma chave primária sobre o conjunto de ficheiros da tabela. Relativamente à adição de um objeto de dados, esta ação pode incluir dados estatísticos, como a contagem total de registos, valores mínimo e máximo por coluna e, ainda, as contagens nulas. Para além disso, é também incorporado o tamanho do objeto adicionado, em *bytes*, e o último instante em que o mesmo foi modificado. Na eventualidade de existir um caminho no sistema de ficheiros, já existente na tabela, numa ação deste género, as respetivas estatísticas e restantes informações da última versão devem ser substituídas face às versões anteriores. Consequentemente, estes dados podem ser acrescentados num caminho do sistema de ficheiros já existente numa tabela *delta*.

Quanto à remoção de um objeto de dados, esta ação inclui essencialmente o instante em que a mesma ocorreu. A exclusão física do ficheiro pode ocorrer lentamente após algum limite de tempo de expiração especificado pelo utilizador. Este atraso permite que vários leitores concorrentes continuem a executar operações numa versão desatualizada da tabela. Uma ação de remoção deve permanecer no estado da tabela como uma marca para exclusão até que tenha expirado. Uma marca de exclusão expira quando o instante em que se deu a criação do ficheiro *delta* excede o limite do tempo de expiração adicionado no momento em que ocorreu a ação de remoção.

A Tabela 5 sintetiza o modelo adotado no registo do conteúdo desta ação.

Nome do campo	**Descrição**
`path`	Caminho exato ou relativo do ficheiro que deve ser removido de uma tabela *delta*.
`deletionTimestamp`	Instante em que ocorreu a remoção do ficheiro de uma tabela *delta*.
`dataChange`	Campo que exprime a existência de uma modificação do estado atual da tabela *delta*.

Tabela 5: Esquema com o conteúdo relativo à ação *remove*

À semelhança da ação *add*, evidencia-se na Listagem 3 um exemplo da ação *remove* nos registos transacionais de uma tabela *delta*.

```
{
    "remove": {
        "path": "part-00001-9….snappy.parquet",
        "deletionTimestamp": 1515488792485,
        "dataChange": true
    }
}
```

Listagem 3: Ação correspondente à remoção de um ficheiro numa tabela *delta*

Dado que não há garantias que as ações que integram um determinado registo transacional sejam aplicadas de forma ordenada, não é possível existir, numa versão da tabela *delta*, o mesmo caminho para um ficheiro *Parquet* em múltiplas operações sobre o mesmo. Por fim, a propriedade *dataChange*, presente quer numa ação do tipo *add* quer numa do tipo *remove*, pode ser definida como falsa para indicar que uma ação quando combinada com outras na mesma versão atómica da tabela apenas reorganiza os dados existentes ou adiciona novas estatísticas. Este campo é útil, por exemplo, nas consultas de *streaming* para ignorar ações que não afetariam os resultados finais.

⇒ ***Set Transaction*:** a ação *txn* é utilizada em tarefas que envolvem *structured streaming* para registar ficheiros numa tabela *delta*. Os sistemas atuais de processamento incremental (por exemplo, sistemas de *streaming*) que monitorizam a execução de transações nas suas aplicações precisam de anotar o progresso feito na realização de tais operações de modo a evitar a duplicação de dados na ocorrência de falhas ou de novas tentativas de execução.

A Tabela 6 mostra o modelo de armazenamento inerente à ação *txn*.

Nome do campo	Descrição
`appId`	Identificador único da aplicação que executou uma transação sobre a tabela *delta*.
`version`	Identificador numérico específico da aplicação para uma transação.
`lastUpdated`	Instante em que esta ação da transação em causa foi criada, em milissegundos.

Tabela 6: Esquema com o conteúdo relativo à ação *txn*

Os identificadores de transações permitem que essas informações sejam registadas atomicamente no estado transacional de uma tabela *delta* junto com outras ações que modificam o conteúdo da mesma. Com isto é possível saber o ponto atual de processamento para posteriormente resumir a execução da

tarefa. Os identificadores em causa são armazenados em pares, onde o primeiro componente, *appId*, é o identificador único do processo que modifica a tabela e o segundo componente, *version*, transparece o progresso obtido por uma determinada aplicação. O registo atómico destas informações junto com as modificações na tabela permite que os sistemas externos em questão façam as suas alterações numa tabela *delta* idempotente. Para assimilar melhor esta estruturação de dados, apresenta-se na Listagem 4 um exemplo desta ação.

```
{
    "txn": {
        "appId": "3ba13872-2d47-4e17-86a0-21afd2a22395",
        "version": 364475
    }
}
```

Listagem 4: Ação relativa ao progresso da execução de uma transação numa tabela *delta*

⇒ ***Change Protocol*:** esta ação é usada para atualizar a versão do protocolo transacional do sistema *Delta Lake* na execução de leituras e escritas. Por conseguinte, é possível interpretar os metadados da tabela e os respetivos registos transacionais em diferentes versões do protocolo *delta*. Assim, os utilizadores que possuem versões antigas ou mais recentes deste sistema podem executar normalmente as operações de leitura ou de escrita numa determinada tabela. De notar que a versão do protocolo será aumentada sempre que alterações não compatíveis com versões futuras forem feitas nesta especificação. Uma vez que mudanças significativas sobre uma tabela *delta* devem ser acompanhadas por um aumento na versão do protocolo, os utilizadores podem assumir que as propriedades ou ações não reconhecidas nunca são necessárias para interpretar corretamente os registos transacionais da mesma.

A Tabela 7 apresenta o esquema de dados relativo a esta ação.

Nome do campo	Descrição
`minReaderVersion`	Versão mínima do protocolo de leitura de uma tabela *delta* que um utilizador deve adotar de modo a ler corretamente a tabela em causa.
`minWriterVersion`	Versão mínima do protocolo de escrita de uma tabela *delta* que um utilizador deve seguir de maneira a escrever corretamente na tabela em questão.

Tabela 7: Esquema com o conteúdo relativo à ação *protocol*

Para interpretar na totalidade o conteúdo da Tabela 7, evidencia-se na Listagem 5 um exemplo da ação *protocol* num registo transacional.

```json
{
    "protocol": {
        "minReaderVersion": 1,
        "minWriterVersion": 2
    }
}
```

Listagem 5: Ação relativa à atualização do protocolo de uma tabela *delta*

⇒ ***Commit Info*:** esta ação indica alguns detalhes que dizem respeito à alteração efetuada. Aspetos como a metainformação, a versão da tabela e as listas de ficheiros e de transações são devidamente apresentados. Neste tipo de ação, qualquer formato *JSON* válido pode ser persistido, pelo que não existe um esquema de dados predefinido. Tal como se verificou para as ações anteriores, expõe-se na Listagem 6 um exemplo para a ação *commitInfo*.

```json
{
    "commitInfo": {
        "timestamp": 1612555761805,
        "userId": "8",
        "userName": "a@b.com",
        "operation": "WRITE",
        "operationParameters": {
            "mode": "ErrorIfExists",
            "partitionBy": "[]"
        },
        "isBlindAppend": true,
        "operationMetrics": {
            "numFiles": "46",
            "numOutputBytes": "428499090",
            "numOutputRows": "11997996"
        },
        "notebook": {
            "notebookId": "4443029",
            "notebookPath": "Users/a@b.com/actions",
            "clusterId": "1027-202406-0000991"
        }
    }
}
```

Listagem 6: Ação associada à metainformação de uma operação executada numa tabela *delta*

Ao considerar que um utilizador pretende criar uma transação que envolva a adição de uma nova coluna, com novos dados, a uma tabela *delta*, o sistema *Delta Lake* vai separar a transação em causa em dois segmentos distintos e, assim que a mesma termine, acrescenta as seguintes ações aos registos transacionais:

1. *Change metadata*: alteração do esquema da tabela para incluir a nova coluna;

2. *Add file*: ação invocada por cada ficheiro adicionado.

Tal como se pode observar no exemplo acima, as ações são ordenadas atomicamente segundo a transação requisitada por parte do utilizador.

2.5.2.3 Pontos de verificação

O sistema *Delta Lake*, para obter um bom desempenho na execução da operação de leitura, procede à compactação periódica dos registos transacionais em pontos de verificação. Os pontos de verificação armazenam todas as ações não redundantes no estado da tabela até um determinado identificador de registo, no formato *Parquet*. Alguns conjuntos de ações são redundantes e podem ser removidos. A Figura 15 apresenta esses casos.

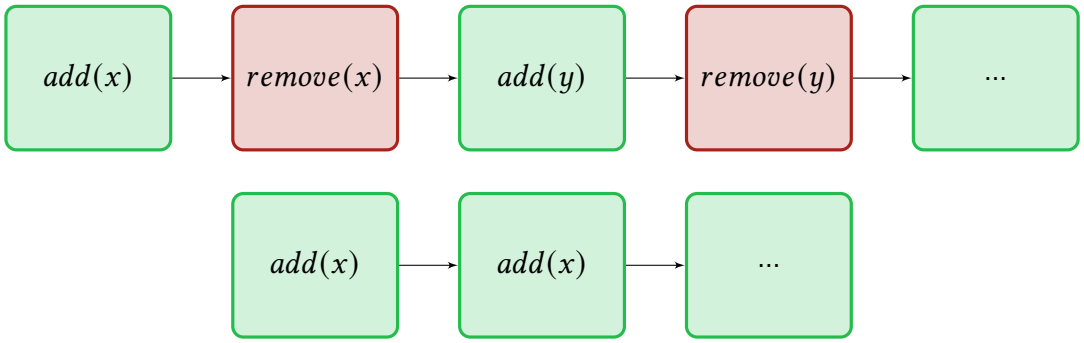


Figura 15: Ações redundantes no sistema *Delta Lake*

O resultado final do processo de verificação corresponde a um ficheiro *Parquet* que contém um registo de adição para cada objeto ainda presente na tabela. Mais, o mesmo também possui os registos de objetos que foram excluídos, cujo período de retenção ainda não expirou, e um pequeno número de registos, como *txn*, *protocol* e *changeMetadata*.

Tal como a Figura 14 transparece, o ficheiro *000002.parquet* representa um ponto de verificação com todos os registos anteriores, ou seja, deste o registo *000000.json* até ao *000002.json*, inclusive. Por omissão, o sistema *Delta Lake* realiza pontos de verificação a cada dez transações. Os ficheiros referidos com extensão *Parquet* encontram-se num formato ideal para consultar metadados sobre a tabela e para descobrir quais os objetos que podem conter dados relevantes para uma consulta seletiva com base nas suas estatísticas de dados. Este facto potencia o propósito desta dissertação, isto é, a adaptação do sistema *Delta Lake* na execução de transações frequentes e de granularidade fina para suportar cargas de trabalho *OLTP*. De modo a aceder eficientemente ao último ponto de verificação da tabela armazenada no sistema *Delta Lake*, os utilizadores podem encontrar essa informação no ficheiro *_last_checkpoint*, tal como a Figura 16 indica.

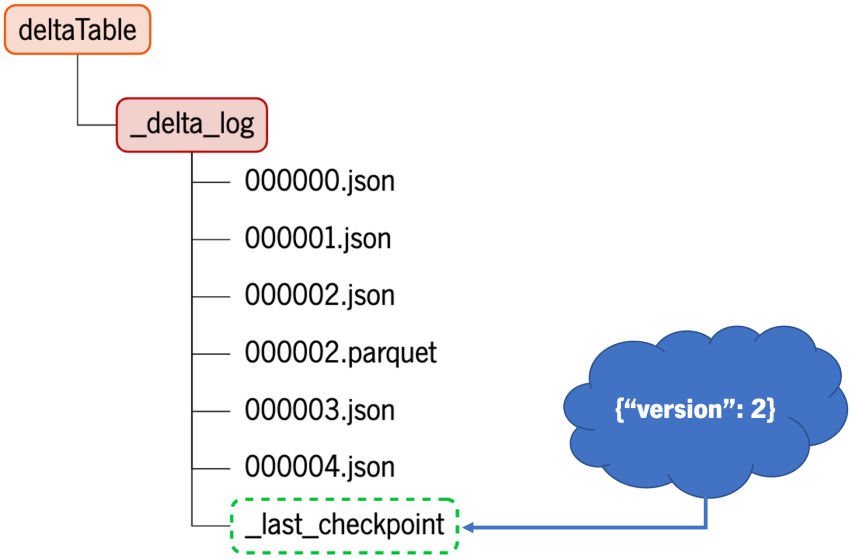


Figura 16: Último ponto de verificação de uma tabela *delta*

2.5.3 Protocolos de acesso

Os protocolos de acesso do sistema *Delta Lake* foram concebidos de maneira a obter execuções de transações serializáveis sobre uma determinada tabela. Esta secção descreve as operações de leitura e de escrita presentes na realização de transações.

2.5.3.1 Leitura de uma tabela *delta*

De modo a executar transações que apenas contêm operações de leitura de uma tabela pertencente ao sistema *Delta Lake* é necessário seguir diversos passos:

1. Ler o ficheiro *_last_checkpoint* para descobrir o último ponto de verificação concretizado (caso exista);

2. Listar todos os registos transacionais desde o último ponto de verificação (caso exista, se não existir considera-se o registo transacional com a versão inicial da tabela, isto é, a 0) até ao ficheiro *JSON* ou *Parquet* mais recente (isto é, o correspondente à última versão da tabela *delta*). Esta operação é usada através do método *listFrom*;

3. Usar os ficheiros do passo anterior para reconstruir o estado da tabela, nomeadamente aqueles que possuem a ação *add* sem haver uma ação *remove* associada;

4. Utilizar as estatísticas das ações presentes nos registos transacionais para identificar quais os objetos de dados a serem usados;

5. Executar a operação de leitura sobre os objetos de dados do passo anterior.

Caso um utilizador leia uma versão antiga do ponto de verificação presente no ficheiro *_last_checkpoint*, este consegue encontrar na mesma os registos transacionais na futura operação de listagem e, consequentemente, reconstruir o estado da tabela *delta*. Desta forma, um utilizador consegue tolerar inconsistência na listagem dos registos mais recentes ou na leitura dos objetos de dados referenciados nos registos transacionais. Tal como foi referido na Secção 2.5.2.3, o ficheiro *_last_checkpoint* ajuda apenas a reduzir o tempo de execução da operação *listFrom* ao disponibilizar o identificador do ponto de verificação mais recente. Assim, na existência de um número assinalável de registos, é possível aceder eficientemente ao último ponto de verificação de uma tabela *delta*.

De modo a simplificar a interpretação do processo relativo à leitura de uma tabela *delta*, procedeu-se à esquematização do mesmo. A Figura 17 indica os passos necessários a uma realização válida deste tipo de operação na execução de uma determinada transação.

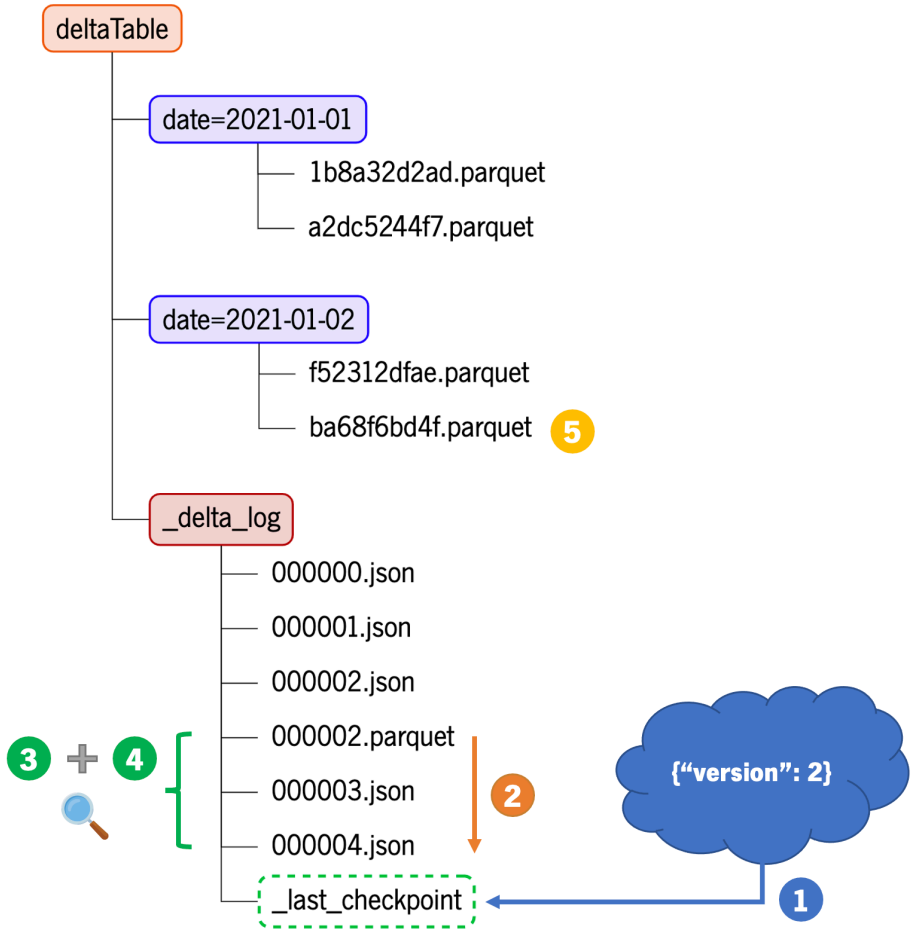


Figura 17: Leitura de uma tabela *delta*

2.5.3.2 Escrita de um novo estado transacional numa tabela *delta*

De forma a executar transações que incluem operações de escrita numa tabela *delta*, devem-se tomar os seguintes passos:

1. Identificar o registo transacional mais recente, versão i (última versão da tabela *delta*), usando os passos 1 e 2 referidos no processo de leitura. Após a transação efetuar a leitura dos registos transacionais procede-se à escrita do registo $i + 1$;
2. Leitura dos dados da tabela (idêntico ao processo de leitura);
3. Escrita dos objetos de dados (*Parquet*) da transação em causa para a diretoria em questão. Uma vez terminada esta ação, os últimos estão prontos para serem referenciados no registo transacional que se pretende adicionar;
4. Escrita atómica do ficheiro *JSON* com a versão $i + 1$ no estado transacional da tabela.

À semelhança da leitura de uma tabela *delta*, elaborou-se um esquema que sumaria o processo relativo à escrita de objetos de dados, em formato *Parquet*, e os respetivos registos transacionais, com extensão *JSON*. A Figura 18 expõe esse mesmo esquema.

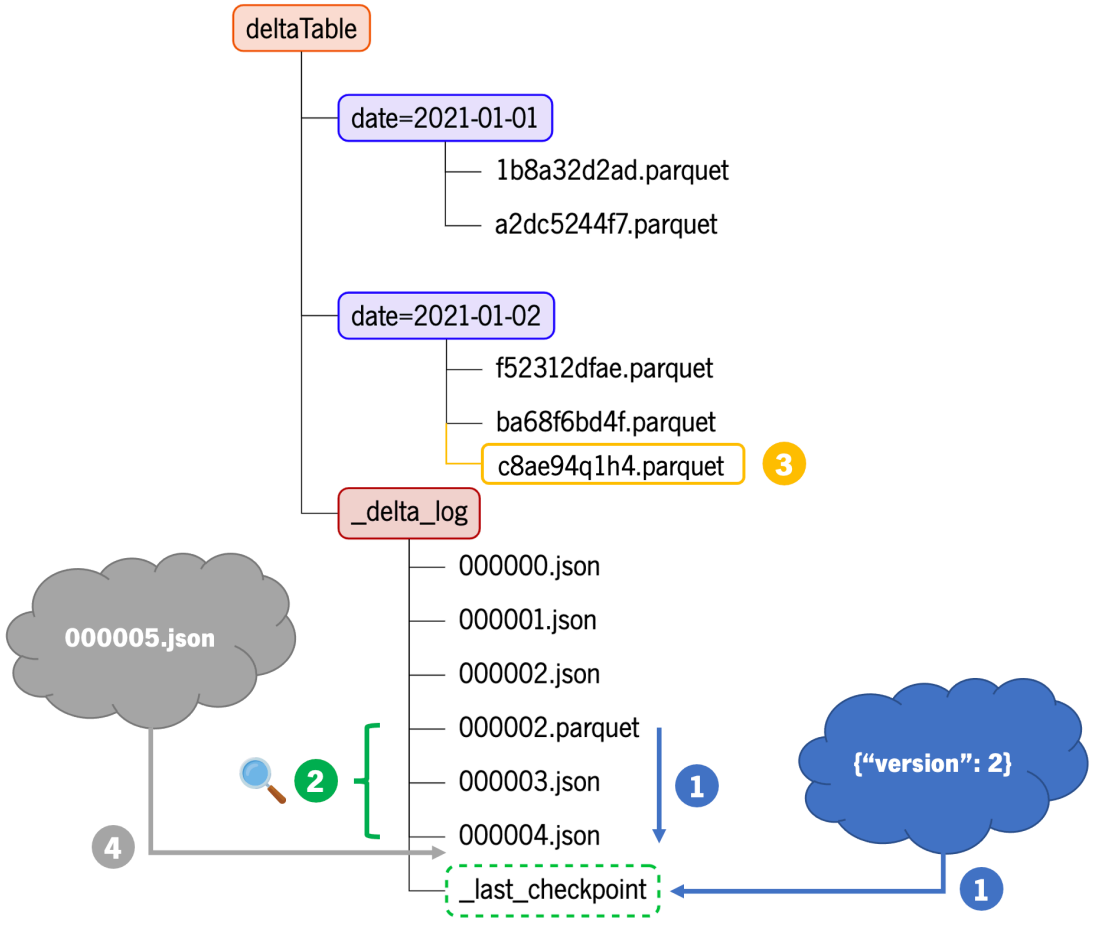


Figura 18: Escrita de um novo estado transacional numa tabela *delta*

Para além dos quatro passos mencionados em cima, é também possível executar outro opcionalmente. Após a escrita do ficheiro *JSON* com o conteúdo da operação de escrita efetuada, existe a possibilidade de também adicionar nos registos transacionais o ponto de verificação, com formato *Parquet*, para

essa versão da tabela *delta*. Tal como foi referido na Secção 2.5.2.3, o sistema *Delta Lake*, por omissão, realiza pontos de verificação a cada dez transações. Contudo, este último valor pode ser configurado por parte do utilizador. Assim que a escrita do ficheiro em questão esteja concluída, é devidamente atualizado o ficheiro *_last_checkpoint* com o ponto de verificação relativo à versão $i + 1$. De notar que este último passo apenas afeta o desempenho do sistema em causa e, na eventualidade da ocorrência de uma falha, os dados correspondentes não são corrompidos. Tal como seria de esperar, este passo opcional só é completado se o anterior for efetuado com sucesso.

Capítulo 3

Testes intermédios

De forma a compreender na totalidade as particularidades do sistema *Delta Lake* [8], foram realizados alguns testes intermédios para ganhar uma maior intuição acerca do seu funcionamento. As experiências que se seguem foram implementadas tanto em *Python* [34] como na linguagem de programação *Scala* [29].

A primeira linguagem referida foi selecionada em detrimento doutras para a implementação do teste intermédio da Secção 3.3 devido à sua simplicidade, manutenção e legibilidade. Neste teste efetuou-se a análise do tempo de execução das interrogações do *benchmark TPC-H* [32] sobre tabelas *delta* de diversas dimensões. A concretização deste teste intermédio permitiu conhecer o nível de desempenho do sistema *Delta Lake* na execução de interrogações com diferentes caraterísticas. Para tornar a codificação deste teste ainda mais percetível, foi utilizada a ferramenta *JupyterLab* [20] para, por exemplo, repetir a execução de blocos de código relevantes.

Dado que o sistema *Delta Lake* é nativamente codificado em *Scala*, esta foi usada na implementação do testes intermédio que estuda a possibilidade da integração do estado transacional de uma tabela *delta* numa base de dados não relacional e no teste de desempenho que estabelece comparações entre o sistema *Delta Lake* e uma versão adaptada do último em *MongoDB* [28]. Deste modo, os testes intermédios presentes nas Secções 3.4 e 3.5 foram especificados nesta linguagem de programação. Esta decisão foi tomada desta forma uma vez que para aceder ao estado transacional de uma tabela *delta* é necessário invocar um conjunto de instruções muito específicas que só estão disponíveis em *Scala*. Apesar desta realidade, também é possível reproduzir os mesmos resultados em *Python* utilizando instruções semelhantes às anteriores.

Quanto ao ambiente computacional, os testes em questão são conduzidos localmente num computador com seis núcleos de processamento e 16 *GB (Gigabyte)* de memória *RAM (Random Access Memory)*. Em relação ao espaço de armazenamento, existe um disco *SSD (Solid State Drive) NVMe (Non-Volatile*

Memory Express) com a capacidade de 512 *GB*.

Dito isto, com a realização de múltiplas experiências sobre o sistema *Delta Lake* e com a aglomeração dos resultados obtidos, pretende-se estudar e validar a solução que será posteriormente proposta no Capítulo 4, onde é exposta a sua arquitetura e implementação.

3.1 Instalação

Recorreu-se às ferramentas *Spark* (versão 3.1.2) [6], *framework* útil para o processamento distribuído, *JupyterLab* (versão 3.1.0), onde é implementado o primeiro teste intermédio, e aos sistemas *Delta Lake* (versão 1.0.0) e *MongoDB* (versão 5.0.1), que é importante para a persistência do estado transacional de uma tabela *delta* numa base de dados não relacional (*NoSQL*). Considerou-se ainda as linguagens *Python* e *Scala*, com as versões 3.9.6 e 2.12.14, respetivamente. Relativamente à primeira linguagem de programação, foram também instalados três pacotes distintos: *PySpark* [22] (versão 3.1.2), *FindSpark* [15] (versão 1.4.2) e *PyMongo* [21] (versão 3.12.0).

Dado que *Spark* é nativamente implementado em *Scala*, existe a necessidade de usar um utensílio que estabeleça a ponte entre a própria e a linguagem de programação *Python*. Como tal, *PySpark* surgiu para tratar esta interação e representa, de forma sucinta, uma *API*, em *Python*, para *Spark*. Além disso, o último ajuda a fazer a interface com conjuntos de dados distribuídos e resilientes, também conhecidos por *RDD*, através da biblioteca *Py4j*. Esta biblioteca é bastante popular e também permite estabelecer uma interface dinâmica com objetos *JVM (Java Virtual Machine)*. A resiliência dos dados apontados deve-se ao facto dos mesmos serem regenerados caso sejam perdidos. Quanto à sua distribuição, estes são computados e guardados em diversos nós de armazenamento. Em suma, este pacote fornece uma *API* simples e abrangente com várias opções de visualização de dados, o que não se verifica nas linguagens *Java* ou *Scala*.

Em relação ao pacote *FindSpark* este foi usado para localizar convenientemente a pasta de instalação da ferramenta *Spark*. Isto é bastante útil sobretudo no que diz respeito à configuração inicial de um *notebook* em *JupyterLab* onde se têm de especificar quais as ferramentas a serem utilizadas.

Quanto ao pacote *PyMongo*, este proporciona a conexão entre um cliente *Python* e uma base de dados em *MongoDB*. No caso do teste relativo à Secção 3.3, este pacote foi também utilizado para lidar com a criação de uma base de dados concreta que, por sua vez, possui diversas coleções.

No que toca à linguagem *Scala*, foi incluído um utensílio, equivalente ao que foi mencionado previamente, que estabelece uma conexão entre a ferramenta *Spark* e uma base de dados *MongoDB* [27]. Para além disso, foi ainda incorporado o pacote *ScalaTest* [23] que possibilita a especificação de testes unitários de modo a assegurar a funcionalidade e a qualidade do código desenvolvido.

De modo a inspecionar e a visualizar o conteúdo dos objetos de dados, sob o formato *Parquet* [6], que se encontram disponíveis no estado transacional de uma tabela *delta*, foi também usado o utensílio *parquet tools*, na versão 1.12.0, para o efeito.

Por fim, foi usada uma outra ferramenta dedicada à realização de *benchmarks*, isto é, o *TPC-H*. O *TPC-H* é um *benchmark* desenhado para avaliar sistemas analíticos, isto é, sistemas que realizam operações de leitura sobre dados de variadas proporções que, por sua vez, pertencem a múltiplas bases de dados. Para o contexto deste tema de dissertação, foram utilizados tanto os conjuntos de dados disponibilizados pelo mesmo como as respetivas *queries*. Estes dados estão estruturados em diversas tabelas, com múltiplas relações e diversos graus de complexidade. O esquema do conjunto de dados em causa pode ser visualizado com mais detalhe no Apêndice A do vigente documento.

3.2 Configuração

A utilização do *benchmark TPC-H* pressupõe a escolha de um factor de escalonamento, isto é, a quantidade de dados gerada e que por sua vez será carregada no sistema de dados. Este fator, denominado por *SF*, é aplicado à maioria das tabelas em questão para ir ao encontro com o valor representativo da dimensão do conjunto de dados especificado pelo utilizador. Durante a execução do *benchmark*, as interrogações incidem sobre os dados carregados, sendo que os resultados estão relacionados com a dimensão das relações.

A Tabela 8 expõe a cardinalidade de cada tabela presente no utensílio *TPC-H* em função de *SF*, que traduz o fator mencionado.

Nome da tabela	**Cardinalidade**
LINEITEM	$SF \times 6000000$
ORDERS	$SF \times 1500000$
PARTSUPP	$SF \times 800000$
PART	$SF \times 200000$
CUSTOMER	$SF \times 150000$
SUPPLIER	$SF \times 10000$
NATION	25
REGION	5

Tabela 8: Cardinalidade de cada tabela da ferramenta *TPC-H* em função do fator *SF*

3.3 Análise do tempo de execução das interrogações da ferramenta *TPC-H*

No primeiro teste intermédio planeou-se o estudo e medição do tempo de execução de *queries* para diferentes dimensões de um conjunto de dados persistido em diversas tabelas *delta*. Para averiguar

esta situação, integraram-se os ficheiros com extensão *TBL* relativos à ferramenta *TPC-H* em múltiplas tabelas *delta*. De notar que cada um destes ficheiros possui o conteúdo de uma determinada tabela do *benchmark TPC-H*, onde os dados presentes em cada linha são delimitados por um símbolo específico (|). Para iniciar este processo, desempenhou-se o pré-processamento das oito tabelas em causa. Nesta etapa definiu-se explicitamente o esquema de dados das mesmas, produzindo-se o respetivo *dataframe* com os dados processados. Estas duas últimas tarefas são particularmente importantes uma vez que para executar as *queries* mencionadas é necessário garantir que o tipo de dados associado a cada coluna das tabelas esteja correto. Com recurso à ferramenta *Spark*, reproduziu-se o conteúdo das tabelas sob o formato de *dataframes*, mapeando-se cada linha de acordo o seu esquema de dados e com o delimitador referido. No fim, investigou-se a natureza global de cada tabela, isto é, a dimensão e a existência de valores nulos, de modo a averiguar a presença de alguma incongruência nos dados correspondentes.

Tendo em consideração a existência de muitas tabelas, os dados das últimas foram guardados num dicionário, onde a chave refere-se ao nome da tabela e o valor corresponde ao *dataframe* que lhe está associado. Com a utilização desta estrutura de dados é assegurado o acesso à informação em causa em tempo constante $(O(1))$. É importante salientar que grande parte das *queries* do utensílio *TPC-H* acedem à maioria das tabelas instanciadas e fazem uso de operações como `JOIN`, `GROUP BY`, `ORDER BY` e `SELECT`, pelo que se constituem como um bom caso de estudo. Isto deve-se ao facto do sistema *Delta Lake* ser orientado a cargas de trabalho *OLAP*. Neste ambiente invocam-se operações de leitura de dados com gamas de valores alargadas, onde nas quais existe a possibilidade de executar escritas de granularidade fina. Como as vinte e duas interrogações do *benchmark TPC-H* estão nativamente implementadas em *Scala*, foi ainda necessário traduzi-las na linguagem de programação *Python*. Algumas destas também requerem a definição explícita de funções, também conhecidas por *UDF (User-Defined Function)*, pelo que foi preciso fazer essa adaptação.

A Figura 19 mostra os resultados obtidos nesta experiência, onde foram considerados os fatores 1 *GB*, 2.5 *GB* e 5 *GB* para observar o comportamento e o desempenho do sistema *Delta Lake*. Olhando atentamente para o gráfico em causa, pode-se apurar que, para algumas interrogações, existem variações mais notórias a nível de tempo de execução consoante o tamanho do conjunto de dados considerado. São exemplos deste facto as *queries* Q2, Q11 e Q21. Isto deve-se sobretudo à execução de um elevado número de operações com um grau de complexidade elevado. Nestas ocorrem maioritariamente operações do tipo `JOIN` que, tal como o nome indica, junta pares de tabelas para realizar uma determinada ação. Dado que este conjunto de dados possui um número significativo de registos, é expectável que este tipo de operação escale de acordo com a dimensão do próprio. Para além disso, a existência de múltiplas operações de agregação, como `GROUP BY`, potenciam o aumento do tempo de execução de uma determinada *query* com um acréscimo evolutivo de registos. Relativamente às restantes *queries*, pode-se verificar que o sistema *Delta Lake* apresentou um desempenho bastante positivo, uma vez que para diferentes tamanhos do conjunto de dados o tempo de execução manteve-se quase inalterado.

Em síntese, foi possível averiguar o desempenho do sistema *Delta Lake* durante a execução de diferentes tipos de interrogações sobre conjuntos de dados de diversas proporções. A isto acrescenta-se o

entendimento quer do processo de especificação explícito de esquemas de dados quer da codificação de *queries* mais complexas.

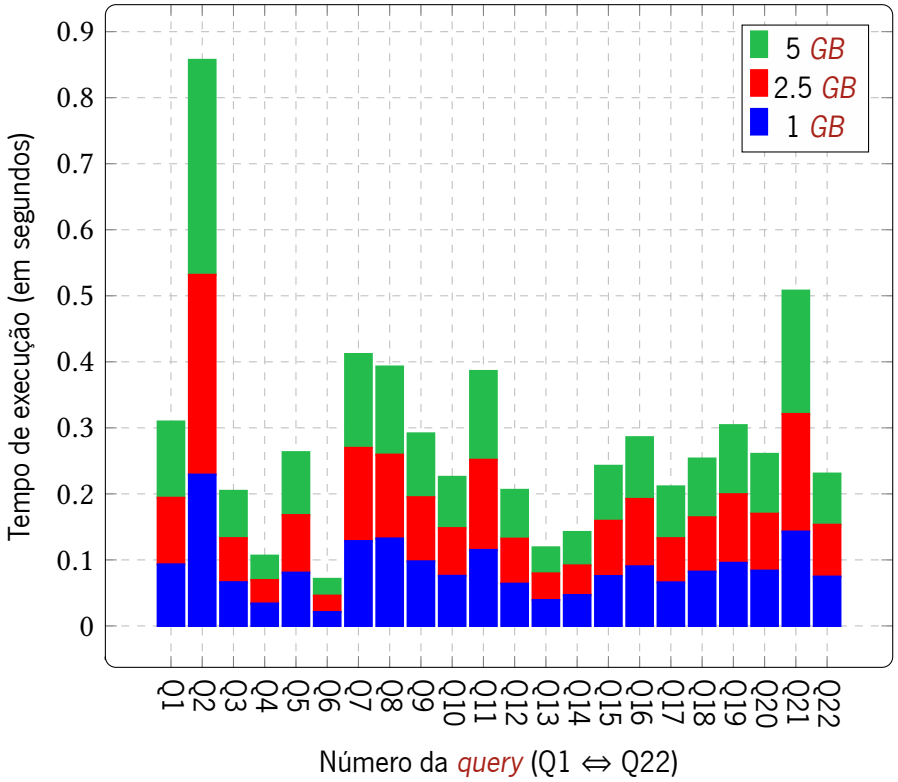


Figura 19: Comparação entre a dimensão do conjunto de dados e o respetivo tempo de execução

3.4 Incorporação do estado transacional de uma tabela *delta* em *MongoDB*

Neste teste procura-se integrar os registos transacionais de uma tabela *delta* numa base de dados não relacional (*NoSQL*). Para tal, foi usada uma base de dados *MongoDB* e o respetivo conector em *Scala* [27]. O propósito deste teste passa por adaptar o sistema *Delta Lake* à execução de transações com pequenas dimensões, ponto fulcral deste tema de dissertação. O tempo despendido a escrever cada ficheiro localmente e, ainda, o *overhead* obtido ao ler vários ficheiros faz com que o sistema *Delta Lake* só seja usável com poucos ficheiros *delta*. Consequentemente, verificam-se poucas atualizações com grandes proporções, aspetos estes caraterísticos de uma carga de trabalho *OLAP*. Com a leitura e escrita dos vários objetos *delta* numa base de dados não relacional antecipa-se a obtenção de um desempenho superior, devendo permitir mais transações de pequenas dimensões. Como tal, pretende-se investigar esta adaptação sobre o sistema *Delta Lake* e verificar se esta solução é ou não viável.

À semelhança do que se sucedeu no primeiro teste intermédio (Secção 3.3), foram usadas as tabelas do conjunto de dados disponibilizados pela ferramenta *TPC-H*, sendo que para cada uma das últimas

criou-se a respetiva tabela *delta*. A abordagem relativa ao processamento do conteúdo de cada tabela foi exatamente igual ao que se verificou no teste anterior, isto é, com o auxílio da ferramenta *Spark*.

Inicialmente conseguiu-se realizar a transferência do estado de cada tabela *delta* para uma coleção de uma base de dados em *MongoDB* com recurso ao método *history*, função essa disponibilizada pela *API* do sistema *Delta Lake*. Todavia, detetou-se um resultado que não se ajustava ao objetivo deste teste. Com a utilização exclusiva desta função ocorre apenas a transferência do cabeçalho dos registos transacionais de cada tabela *delta*, ou seja, os que dizem respeito à ação *commitInfo*. Para realizar a transferência do estado completo de cada uma das mesmas é necessário adotar uma estratégia diferente. Assim, é imposto um processamento apropriado e seletivo de modo a extrair toda a informação relevante dos ficheiros *JSON* presentes. Para isso, existem duas alternativas:

1. Utilizar a classe *DeltaLog* do sistema *Delta Lake* para aceder ao estado transacional de uma determinada versão de uma tabela *delta*:

```
1 val version = 0
2 val path = "..."
3 val deltaLog = DeltaLog.forTable(spark, path).getSnapshotAt(version)
```

Listagem 7: Acesso ao estado transacional de uma tabela *delta*

2. Usar o método convencional da *API* do sistema *Delta Lake* para realizar a leitura de uma determinada versão da tabela *delta*:

```
1 val version = 0
2 val path = "..."
3 val deltaDF = spark.read.format("delta").option("versionAsOf", version).load(path)
```

Listagem 8: Leitura de uma tabela *delta*

Em relação à primeira opção, o uso da classe *DeltaLog* permite consultar toda a informação pertinente do estado transacional de uma tabela *delta*, isto é, dados e metadados. Já a segunda alternativa apenas devolve o conteúdo da tabela *delta* sob a forma de um *dataframe*. Por conseguinte, nesta alternativa é preciso aglomerar os últimos dados com os metadados devolvidos pelo método *history*. Ambas as soluções são válidas, contudo é importante referir que em *Python* a primeira alternativa não é disponibilizada ao utilizador, uma vez que a classe *DeltaLog* é uma classe interna da implementação do sistema *Delta Lake*. Como o último sistema é especificado em *Scala*, não existe este impedimento na linguagem de programação em questão.

Para além deste ajuste, é ainda necessário especificar um padrão explícito no momento da persistência dos dados e metadados de uma tabela *delta* em *MongoDB*. A Listagem 9 exibe uma proposta para o armazenamento da informação mencionada.

```
{
    "version": "...",
    "timestamp": "...",
    "operation": "...",
    "operationParameters": { "..." },
    "operationMetrics": { "..." },
    "add": [ { "..." } ],
    "remove": [ { "..." } ]
}
```

Listagem 9: Modelo de armazenamento idealizado para um ficheiro *JSON* em *MongoDB*

Nesta solução, os cinco primeiros campos dizem respeito a metadados enquanto que as propriedades *add* e *remove* estão associadas às linhas adicionadas e removidas de uma tabela delta, respetivamente. Com a utilização deste modelo é possível alojar corretamente toda a informação relevante das tabelas *delta*, versão após versão.

Assim, com o estudo desta possibilidade, ou seja, incluir parte do estado transacional do sistema *Delta Lake* numa base de dados não relacional, pode-se concluir que de facto esta solução é computacionalmente viável, pelo que será considerada na especificação do sistema *HyLake*.

3.5 Estudo do desempenho dos sistemas *Delta Lake* e *MongoDB*

No último teste intermédio pretende-se conhecer o nível de desempenho do sistema *Delta Lake* na execução das operações de inserção e remoção de dados numa tabela *delta* e compará-lo com *MongoDB*. Para isso, são medidos vinte vezes os tempos de execução para cada operação de forma a obter valores mais coerentes. No fim dessas medições, é efetuada a média desses valores de forma a extrair o tempo de execução final. Para além disso, a dimensão da tabela *delta* varia tanto a nível do número de linhas como de colunas. O número de linhas oscila entre cem a dez mil e o número de colunas varia entre um a cem. Para esta experiência é também realizado um teste dual, isto é, uma adaptação do sistema *Delta Lake* que integra uma base de dados não relacional em *MongoDB*. No fim deste teste, é devidamente comparado o desempenho de ambas as alternativas deste sistema de modo a determinar qual delas é a melhor em termos computacionais. Dado que os resultados obtidos para uma e dez porcento das linhas da tabela *delta*, em ambas as operações, são muito idênticos, optou-se por apresentar apenas os resultados da primeira experiência de maneira a tornar mais clara a interpretação dos mesmos.

3.5.1 Operação de inserção

Esta operação compara uma inserção de n linhas numa tabela *delta* com a junção dos ficheiros *Parquet* com a informação das n linhas provenientes de uma base de dados não relacional em *MongoDB*. No caso representado na Figura 20, onde se insere apenas uma e uma só linha, é clara a diferença entre ambos os sistemas a nível do tempo de execução para as mesmas dimensões da tabela *delta*.

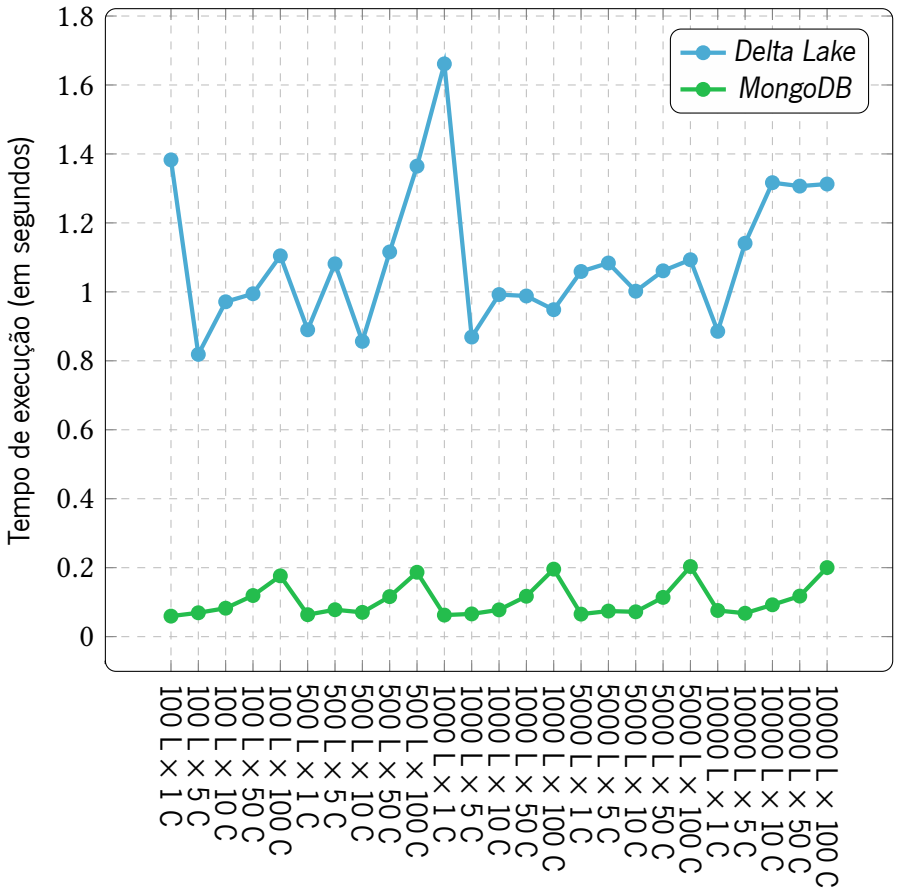


Figura 20: Operação de inserção - *Delta Lake vs MongoDB*

É também evidente que, para a mesma quantidade de linhas presentes na tabela em causa, o número de colunas provoca uma variação no respetivo tempo de execução. Outro fator importante é a escalabilidade de ambas as soluções. Enquanto que no sistema *Delta Lake* o acréscimo de colunas desencadeia um oscilamento irregular no tempo de execução, a versão adaptada em *MongoDB* possui um excelente desempenho independentemente da dimensão da tabela considerada. Assim, pode-se concluir que neste tipo de operação, o sistema que inclui uma base de dados *MongoDB* é claramente a melhor opção quando comparado com o sistema *Delta Lake*.

3.5.2 Operação de remoção

No caso da operação de remoção tenciona-se estabelecer a comparação entre uma remoção de n linhas numa tabela *delta* com a filtragem dos ficheiros *Parquet* com a informação das n linhas removidas provenientes de uma base de dados não relacional em *MongoDB*. Para fazer essa filtragem, é utilizado o operador *exceptAll* [1], proveniente do módulo *Spark SQL* [7]. Este operador desempenha a diferença entre os registos presentes em duas tabelas, incluindo no resultado as linhas repetidas. A Figura 21 mostra os resultados obtidos na remoção de uma linha de uma tabela *delta*.

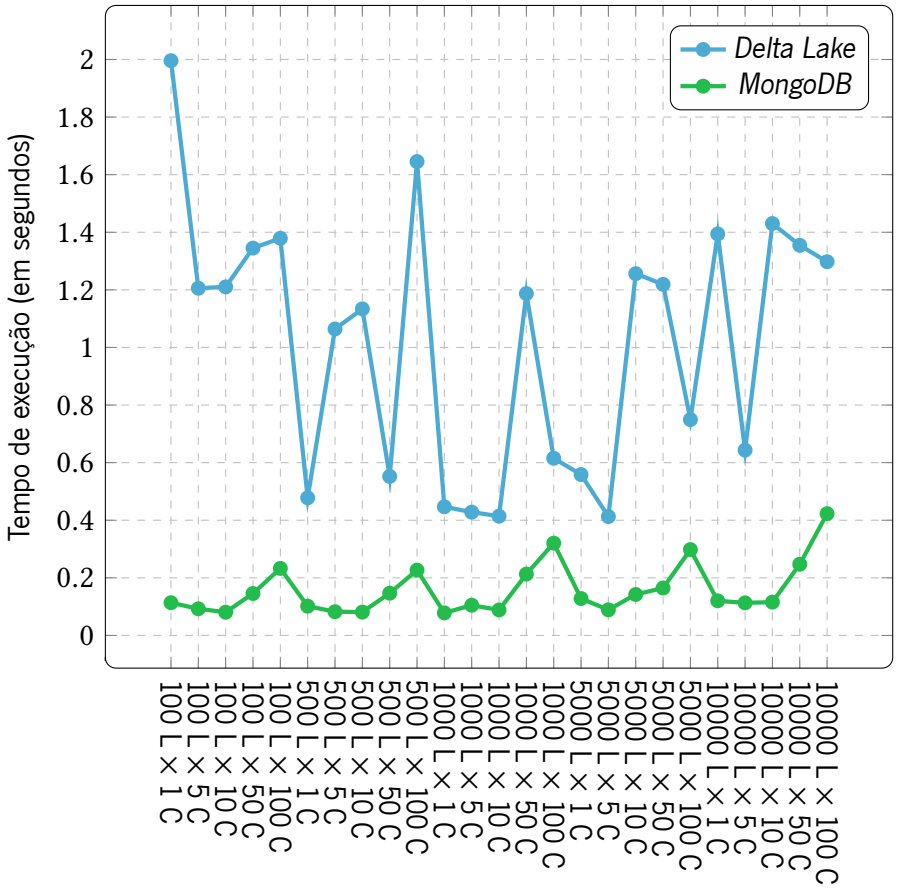


Figura 21: Operação de remoção - *Delta Lake vs MongoDB*

Ao contrário do que se sucedeu na operação de inserção, para certas dimensões da tabela *delta*, uma maior aproximação do comportamento de ambos os sistemas. Contudo, a diferença do desempenho destas versões continua a ser evidente. A escalabilidade da versão adaptada em *MongoDB* mantém-se superior à do sistema *Delta Lake* e o número de colunas permanece como o grande fator para as variações observadas. Dito isto, o sistema que integra uma base de dados *MongoDB* é mais uma vez a melhor escolha para a realização de remoções de dados numa tabela deste género.

[1] https://spark.apache.org/docs/latest/sql-ref-syntax-qry-select-setops.html

3.6 Observações

Atendendo aos resultados obtidos nos testes intermédios deste capítulo, é inequívoco que o sistema *Delta Lake* apresenta alguns compromissos na execução de operações que transformam o seu conteúdo. Com a realização do último teste (Secção 3.5), confirma-se que a adaptação em *MongoDB* é possível ser alcançada, ainda que a mesma tenha sido desenvolvida num ambiente experimental. Olhando exclusivamente para as Figuras 20 e 21 pode-se afirmar que, em média, a versão adaptada em *MongoDB* é 90% mais rápida na inserção de uma linha e 84% mais rápida na remoção de uma linha em relação ao sistema *Delta Lake*, independentemente da dimensão da tabela considerada. Como tal, a integração do estado transacional de uma tabela *delta* numa base de dados não relacional (*NoSQL*), como é o caso do *MongoDB*, constitui-se como uma boa solução para a realização de transações que afetam as linhas de uma tabela deste tipo. Assim, este sistema seria adequado num ambiente computacional onde se realizam várias atualizações de granularidade fina, ou seja, onde ocorrem cargas de trabalho *OLTP*.

Capítulo 4

HyLake

De modo a adaptar o sistema *Delta Lake* [8] à execução de transações frequentes e de granularidade fina em lagos de dados, desenhou-se um novo sistema, intitulado por *HyLake*, que assenta quer nas componentes do sistema *Delta Lake* quer nas caraterísticas de uma base de dados *MongoDB* [28]. Este novo sistema pode ser visto como uma solução híbrida onde os dados correspondentes à versão inicial da tabela considerada são armazenados em *Delta Lake* e as transformações são guardadas em *MongoDB*. Os dados com maiores proporções são armazenados na infraestrutura clássica do sistema *Delta Lake* enquanto que os dados mais recentes e de menores dimensões são guardados em *MongoDB*. A integração da base de dados *MongoDB* neste sistema deve-se não só aos resultados obtidos no teste intermédio da Secção 3.5 como também a outros fatores relevantes, nomeadamente a obtenção de uma escalabilidade horizontal, a inclusão de esquemas de dados flexíveis, o acesso a dados através de qualquer linguagem e a elevada capacidade de interrogar e analisar dados a diferentes níveis de complexidade. Com a leitura e escrita dos vários objetos *delta* numa base de dados não relacional procura-se a obtenção de um desempenho superior, permitindo mais transações de pequenas dimensões. Atendendo à primeira propriedade e ao facto do *MongoDB* ter sido concebido inicialmente como uma base de dados distribuída, esta adequa-se perfeitamente ao objetivo deste trabalho.

A Figura 22 evidencia a arquitetura do sistema *HyLake*. Os objetos de dados representados com as cores azul e branca dizem respeito aos ficheiros codificados no formato *Parquet* [6]. Estes exprimem a primeira versão da tabela *delta* e, por sua vez, da tabela *HyLake*, pelo que permanecem armazenados no sistema *Delta Lake*. À semelhança do que acontece para os ficheiros *Parquet*, o ficheiro *JSON* que possui todos os registos transacionais relativos à criação da tabela também é persistido no sistema *Delta Lake*. Os ficheiros com extensão *JSON* são assinalados na Figura 22 com as cores verde e branca. Relativamente às transformações, estas são alojadas numa coleção de uma base de dados *MongoDB* com o mesmo nome da tabela *delta*. Ao contrário do que acontece no sistema *Delta Lake*, as alterações efetuadas

sobre uma tabela *HyLake* dão origem a novos documentos com versões sucessivamente crescentes. Aos dois componentes presentes na arquitetura do sistema *HyLake* junta-se ainda a ferramenta *Spark* [6] que possibilita, por exemplo, a leitura de uma tabela *delta*. Esta permite o estabelecimento de uma conexão a uma base de dados *MongoDB*, promovendo a leitura e a escrita de dados. Como tal, considerou-se para o efeito o conector oficial do *MongoDB* [27]. No que toca à escrita de dados, este utensílio também é útil na criação e no armazenamento de *dataframes* numa base de dados orientada a documentos.

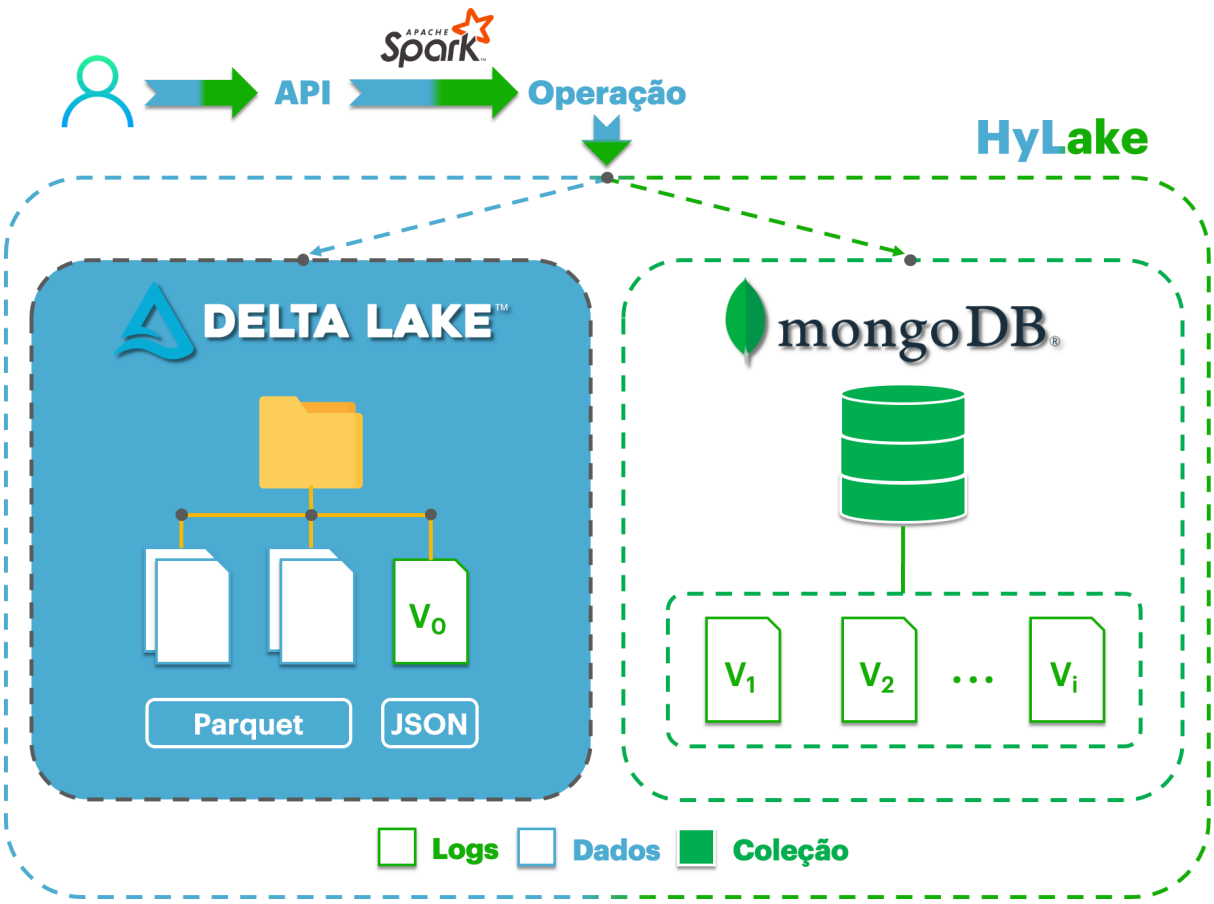


Figura 22: Arquitetura do sistema *HyLake*

Quanto aos pontos de verificação presentes no estado transacional de uma tabela *delta*, estes não são incorporados na base de dados em causa, uma vez que o objetivo principal deste sistema é obter um desempenho positivo no momento em que ocorrem múltiplas escritas de dados. O sistema *Delta Lake* adota este mecanismo para simplesmente reduzir o tempo da leitura do estado atual de uma tabela *delta*, pelo que esta estratégia não será considerada no sistema final. Contudo, na Secção 5.2, onde se aborda a avaliação da solução construída, será comparado o sistema *HyLake* original com uma versão do último onde existem pontos de verificação para provar que a primeira proposta é realmente aquela que se pretende utilizar.

Na receção de um pedido de um utilizador, o sistema *HyLake* apresenta um comportamento distinto consoante a operação invocada. Como tal, é indispensável discutir nesta fase todos os detalhes

relacionados com a criação, a atualização, a inserção, a remoção e, por último, a leitura de uma tabela *HyLake*.

4.1 Criação da tabela

A criação de uma tabela no sistema *HyLake* reduz-se globalmente à instanciação de uma tabela *delta*. Em relação à definição da coleção em *MongoDB* que diz respeito à tabela *HyLake*, esta tarefa é tratada implicitamente pelo respetivo conector [27] e é apenas efetuada no momento em que ocorre a primeira transformação sobre o seu estado inicial. No que toca à sua implementação, a criação de uma tabela *HyLake* é trivial, bastando invocar o método disponibilizado pela *API* do sistema *Delta Lake*.

A Figura 23 apresenta o fio de execução da criação de uma tabela *HyLake*.

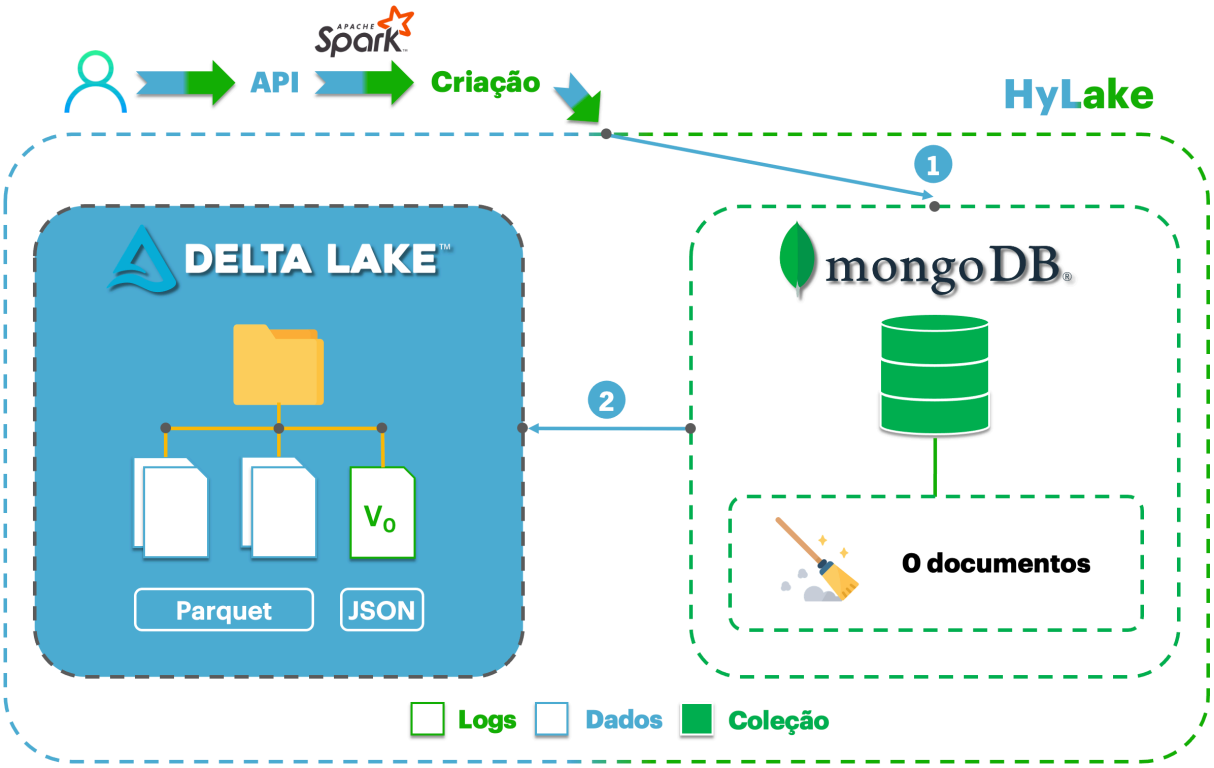


Figura 23: Fio de execução relativo à criação de uma tabela *HyLake*

No início da génese de uma tabela *HyLake* são removidos todos os documentos presentes na respetiva coleção da base de dados *MongoDB* (1). A eliminação destes documentos assegura que não ocorra nenhuma sobreposição de dados entre o antigo e o novo estado transacional de uma tabela *HyLake*. Desta forma, quando a tabela em questão sofrer alterações, os documentos em causa são armazenados sequencialmente, ou seja, de acordo com a ordem com que as transformações são exercidas. Uma vez concluído este passo, procede-se à criação propriamente dita da tabela (2). Nesta etapa é gerado aleatoriamente o seu conteúdo de acordo com a dimensão especificada pelo utilizador. No fim deste

processo, os dados produzidos são inseridos na tabela *delta* de modo a expressar a versão inicial da tabela *HyLake*.

4.2 Atualização e inserção de dados

No que toca à atualização e inserção de dados, optou-se por aglomerar o comportamento das duas operações numa só, isto é, a operação *upsert*. De forma sucinta, no instante em que é invocada esta operação, é implicitamente verificado se o conteúdo que é passado como argumento existe ou não no estado atual da tabela *HyLake*. Se existir, procede-se naturalmente à atualização do seu conteúdo, caso contrário é executada a inserção desses mesmos dados.

Como tal, surge uma nova abordagem na execução desta operação no sistema *HyLake* face ao que o sistema *Delta Lake* oferece. Esta modificação reside no facto do estado transacional de uma tabela *delta* ser agora persistido numa base de dados não relacional em *MongoDB*. Aquando da sua invocação, a operação *upsert* regista todas as alterações efetuadas no repositório de dados em questão, quer se trate de uma atualização ou de uma inserção de dados. De notar que apenas a informação dos ficheiros que possuem o formato *JSON* é guardada em *MongoDB*. Dado que se deseja transferir o conteúdo destes ficheiros para documentos, é indispensável estabelecer um modelo de armazenamento para cada documento alojado em *MongoDB*. A Listagem 10 evidencia o esquema de dados adotado para a operação *upsert*. Para simplificar a sua interpretação, exibe-se nesta listagem um exemplo de uma inserção de dados.

```
{
    "version": 1,
    "timestamp": "Fri Jan 1 12:00:00 WEST 2021",
    "operation": "UPSERT",
    "operationParameters": {
        "predicate": "id IN (1,2)"
    },
    "operationMetrics": {
        "numUpdatedOrInsertedRows": 2
    },
    "add": [
        { "id": 1, "col1": 1 },
        { "id": 2, "col1": 2 }
    ],
    "remove": []
}
```

Listagem 10: Documento relativo à operação *upsert* do sistema *HyLake*

Dos campos presentes nesta estrutura, destacam-se os seguintes três: *version*, *add* e *remove*. Tal como o próprio nome indica, o campo *version* indica o número da versão relativo a uma determinada

tabela *HyLake*. Quanto aos campos *add* e *remove*, estes evidenciam as transformações exercidas sobre a mesma. Caso se trate de uma inserção de dados, o campo *add* é devidamente preenchido para refletir a nova modificação do estado da tabela. Na eventualidade de ocorrer uma atualização, são adicionados tanto ao campo *add* como ao campo *remove* as informações relativas a essa mesma operação. Com esta estratégia é possível realizar uma leitura válida do estado transacional de uma tabela *HyLake* em qualquer estágio, desde a sua génese. Relativamente aos restantes campos, estes possuem apenas metadados que complementam a informação da operação *upsert*. A Figura 24 resume o novo comportamento desta operação.

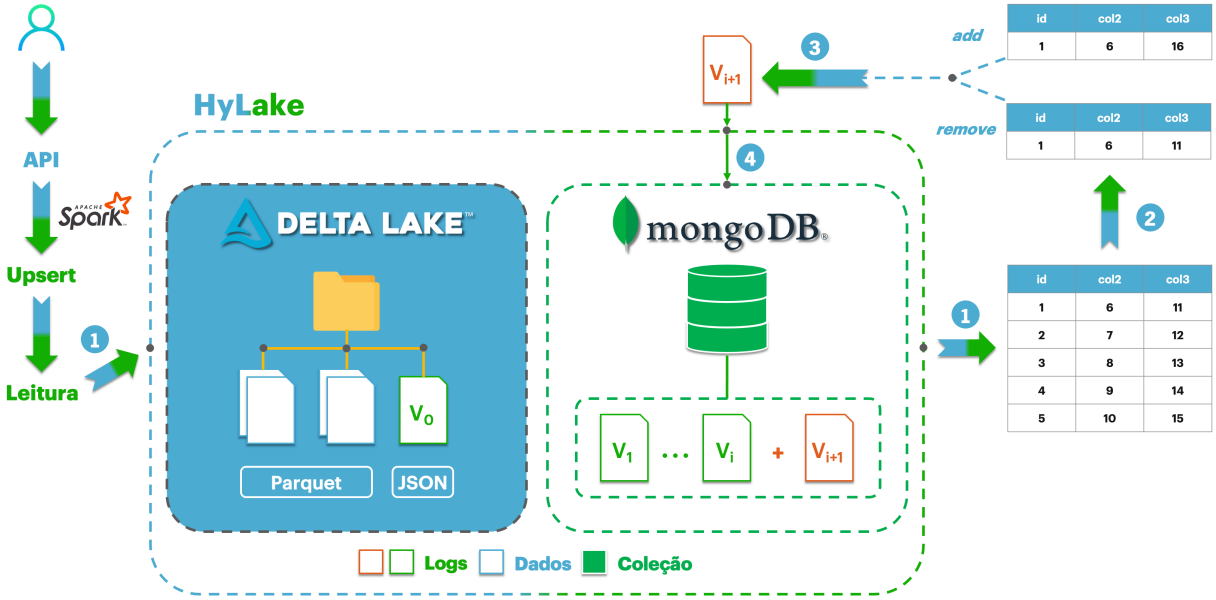


Figura 24: Fio de execução relativo à operação *upsert* do sistema *HyLake*

Na execução de uma *query* que envolva a inserção ou a atualização de dados, é invocada a operação *upsert* sobre o sistema *HyLake*. Inicialmente é determinado o estado atual da tabela *HyLake* para posteriormente colocar nos campos *add* e *remove* os dados referentes às linhas afetadas. Como tal, procede-se à leitura da tabela (1), sendo filtrados todos os registos cujas chaves encontram-se presentes no argumento passado ao método desta operação. Uma vez computadas as linhas da tabela que possuem estes identificadores, a estrutura de dados resultante é assinalada para remoção (2). Imediatamente a seguir a este cálculo é gerado deterministicamente um *dataframe* com as chaves referidas para serem devidamente anotadas para adição (2). A Listagem 11 demonstra este processo.

```
1 val removeDF = hyLakeDF.filter(s"id IN ${ids.mkString("(", ",", ")")}")
2 val addDF = createRandomDataframe(ids, hyLakeDF.columns.length, isUpsert = true)
```

Listagem 11: Computação do conteúdo a ser atualizado ou inserido numa tabela *HyLake*

Após a criação dos *dataframes* referentes às linhas adicionadas e removidas, é construído um novo documento (3) segundo o esquema de dados apontado na Listagem 10 e os dados produzidos nos passos computacionais da Listagem 11. De modo a encerrar a execução desta operação, é persistido o documento em causa na respetiva coleção da base de dados *MongoDB* (4), incrementando não só a variável *version*, que diz respeito à versão atual da tabela, como a variável *upserts*, que espelha o número de execuções da operação *upsert*.

Com estas alterações, antecipa-se uma melhoria substancial, em termos de desempenho, durante a execução de transações com as caraterísticas de um ambiente *OLTP* quando comparado com o sistema *Delta Lake*. O tratamento de ficheiros presentes no estado transacional de uma tabela *delta* é substituído pelo tratamento de documentos alojados numa coleção de uma base de dados em *MongoDB*. Com o aumento do número de transações que envolvem inserções ou atualizações, são muitos os ficheiros *delta*, *Parquet* e *JSON*, que iriam ser instanciados no sistema *Delta Lake*. Por conseguinte, observaria-se neste caso um pobre desempenho desta operação.

4.3 Remoção de dados

Tratando-se de uma operação que modifica o estado atual de uma tabela *HyLake*, é novamente necessário especificar uma estrutura para o armazenamento de documentos relativos à operação de remoção de dados. A Listagem 12 aponta um exemplo concreto do modelo subjacente a esta operação.

```
{
    "version": 1,
    "timestamp": "Fri Jan 1 12:00:00 WEST 2021",
    "operation": "DELETE",
    "operationParameters": {
        "predicate": "id IN (1,2)"
    },
    "operationMetrics": {
        "numDeletedRows": 2
    },
    "add": [],
    "remove": [
        { "id": 1, "col1": 1 },
        { "id": 2, "col1": 2 }
    ]
}
```

Listagem 12: Documento relativo à remoção de dados presentes numa tabela *HyLake*

Tal como se verificou na Listagem 10, o esquema de dados desta operação é muito semelhante ao da operação *upsert*. Todavia, existem algumas diferenças. A mais notória é sem dúvida ao campo *add* estar agora associada uma lista vazia para exprimir a essência da operação em causa. Quanto às

restantes modificações, estas residem na indicação da metainformação relativa à operação de remoção, incluindo, por exemplo, o número total de registos eliminados. Em relação ao campo *remove* é colocada a informação que diz respeito às linhas da tabela que são removidas para posteriormente realizar uma reconstituição correta do estado atual da mesma. Aquando da invocação desta operação, é adicionado um novo documento com a respetiva metainformação e com os dados a serem eliminados da tabela. A Figura 25 exibe o fio de execução da operação discutida.

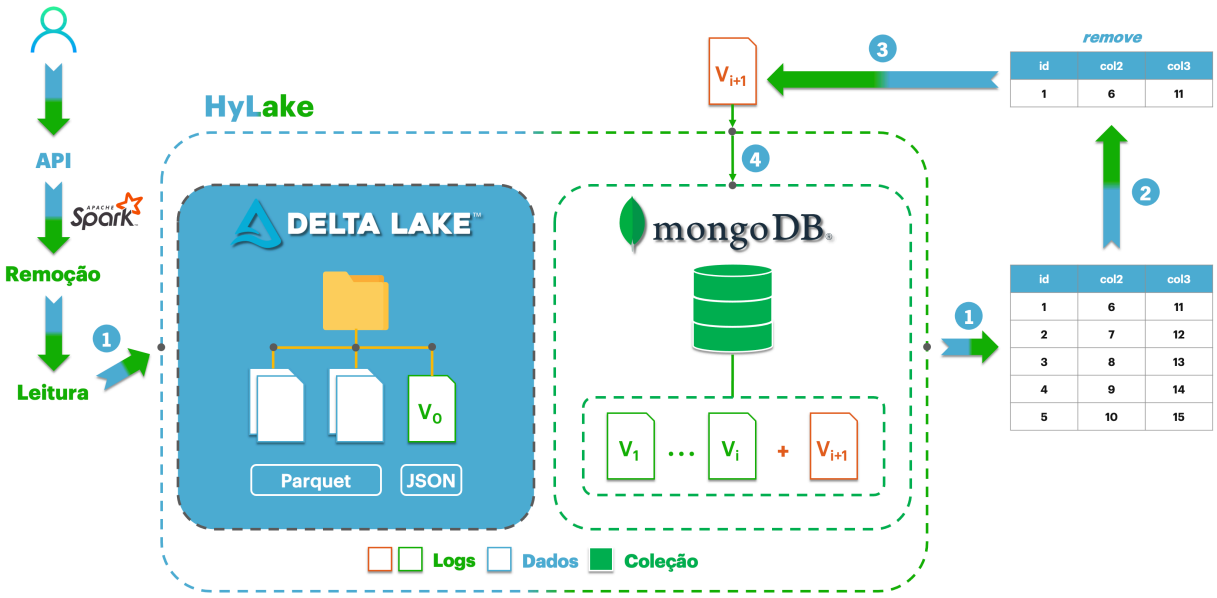


Figura 25: Fio de execução relativo à operação de remoção do sistema *HyLake*

O caminho observado desde que a operação é iniciada até ao instante em que a última é finalizada não modifica os ficheiros *Parquet* presentes no sistema *Delta Lake*. Por conseguinte, para não comprometer o propósito desta operação, dá-se a preservação do estado inicial de uma tabela *HyLake*. Considerando o comportamento da operação de remoção do sistema *HyLake*, é crucial calcular o estado atual da tabela para indicar corretamente os dados referentes às linhas afetadas. Como tal, inicia-se a leitura da tabela (1), recolhendo-se no fim todas as linhas cujos identificadores encontram-se presentes no argumento passado à função em questão. A Listagem 13 exibe a filtragem presente no cálculo anterior.

```
1 val removeDF = hyLakeDF.filter(s"id IN ${ids.mkString("(", ",", ")")}")
```

Listagem 13: Computação do conteúdo a ser removido numa tabela *HyLake*

Assim que estes registos sejam totalmente determinados, a estrutura de dados resultante é assinalada para remoção (2). Após este procedimento, é elaborado o documento alusivo a esta operação (3) de acordo com o modelo de dados referido na Listagem 12 e os dados produzidos no passo computacional da Listagem 13. De maneira a concluir a execução desta operação, é armazenado o documento em causa

na respetiva coleção da base de dados *MongoDB* (4), atualizando a versão atual da tabela em questão com mais uma unidade.

4.4 Leitura da tabela

Atendendo à nova arquitetura proposta para as operações que transformam uma tabela *HyLake*, a leitura do estado transacional da mesma passa a ser totalmente distinta daquela que é definida por omissão no sistema *Delta Lake*. Com a escrita de registos transacionais numa coleção de uma base de dados não relacional (*NoSQL*), a operação de leitura requer uma adaptação de modo a acompanhar as modificações efetuadas. Este ajuste resume-se não só na leitura dos ficheiros *Parquet* presentes no estado transacional de uma tabela *delta* como também na leitura das alterações guardadas em *MongoDB*. Sendo uma operação de consulta, é requerido saber de antemão o formato com que os registos transacionais são persistidos na base de dados não relacional. Neste esquema de armazenamento, que é indicado explicitamente nas Listagens 10 e 12, os campos *version*, *add* e *remove* são os únicos utilizados para realizar a leitura da tabela. Para cada versão desta estrutura de dados, é extraída a informação em causa, aplicando-a aos ficheiros *Parquet* que se encontram persistidos no sistema *Delta Lake*. Tendo em consideração não só o último esquema de dados como também o facto dos nomes das coleções presentes na base de dados *MongoDB* corresponderem aos respetivos nomes das tabelas *delta*, a operação de leitura é intuitiva a nível arquitetural. A Figura 26 sintetiza a forma como a operação de leitura é executada.

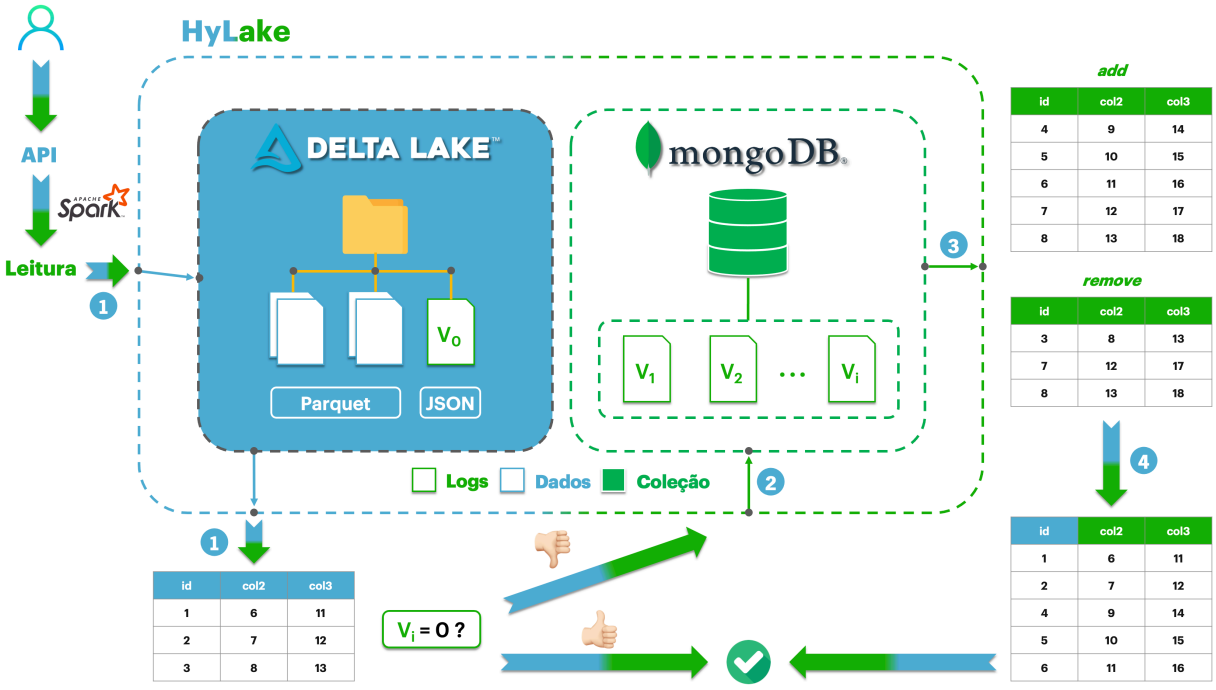


Figura 26: Fio de execução relativo à leitura de uma tabela *HyLake*

Com esta operação é possível aceder ao estado atual de uma tabela *HyLake*, independentemente

da sequência de operações exercida sobre a última. Tal como se pode verificar nas Figuras 24 e 25, a leitura de uma tabela *HyLake* está presente tanto na operação *upsert* como como no instante em que se dá a remoção de dados. Atendendo à importância desta operação no funcionamento do sistema *HyLake*, foram elaborados alguns testes unitários adicionais em *Scala* [29] para reforçar a qualidade da sua implementação.

Aquando da invocação da operação de leitura, esta é automaticamente redirecionada tanto para o sistema *Delta Lake* como para a respetiva coleção de documentos em *MongoDB*. Numa primeira fase é lida a tabela *delta* em questão a partir dos seus ficheiros *Parquet* e do único documento *JSON* (1). A leitura desta tabela é obrigatória uma vez que a mesma traduz a versão inicial da tabela *HyLake* considerada. Caso não exista nenhuma transformação efetuada, o estado atual da tabela *HyLake* é exatamente igual ao da tabela *delta*, pelo que é devolvido o conteúdo dessa estrutura de dados sob a forma de um *dataframe*. Na eventualidade de haver alguma modificação persistida em *MongoDB*, são então acedidas e extraídas as mesmas com recurso a quatro operadores fundamentais: *filter* [1], *explode* [2], *unionAll* [3] e, por fim, *exceptAll* [3].

Tal como o próprio nome indica, o primeiro operador possibilita a filtragem dos documentos armazenados em *MongoDB* até uma certa versão da tabela *HyLake*. Assim, com o uso deste operador, é possível aceder os documentos pretendidos para posteriormente proceder-se à extração dos dados presentes nos campos *add* e *remove* (2). A Listagem 14 apresenta a filtragem mencionada.

```
1 rdd.toDF().filter($"version" <= version.get()).cache()
```

Listagem 14: Filtragem de documentos de acordo com a versão atual da tabela *HyLake*

No que toca à operação *explode*, esta permite expandir uma lista nos seus diversos elementos. No caso do sistema *HyLake*, esta operação é utilizada para expandir as linhas do *dataframe* persistido nos campos *add* e *remove* num determinado documento em *MongoDB*. A Listagem 15 exibe a aplicação desta operação.

```
1 mongoDF.select(explode($"add").as("add"))
```

Listagem 15: Desconstrução das linhas persistidas num documento em *MongoDB*

Para tornar a interpretação da operação *explode* ainda mais clara, a Figura 27 ilustra um exemplo elucidativo. Tal como se pode observar nesta figura, o campo *add* é decomposto em dois segmentos

[1] https://spark.apache.org/docs/latest/api/sql/#filter
[2] https://spark.apache.org/docs/latest/api/sql/#explode
[3] https://spark.apache.org/docs/latest/sql-ref-syntax-qry-select-setops.html

distintos que traduzem as diferentes linhas do *dataframe* em causa. No contexto do sistema *HyLake*, esta operação é essencial na recolha de todas as linhas adicionadas e removidas desde o momento em que a tabela *HyLake* foi instanciada (3). Consequentemente, a operação *explode*, que é disponibilizada pelo módulo *Spark SQL* [7], tem um papel preponderante na realização da leitura de uma tabela do sistema *HyLake*.

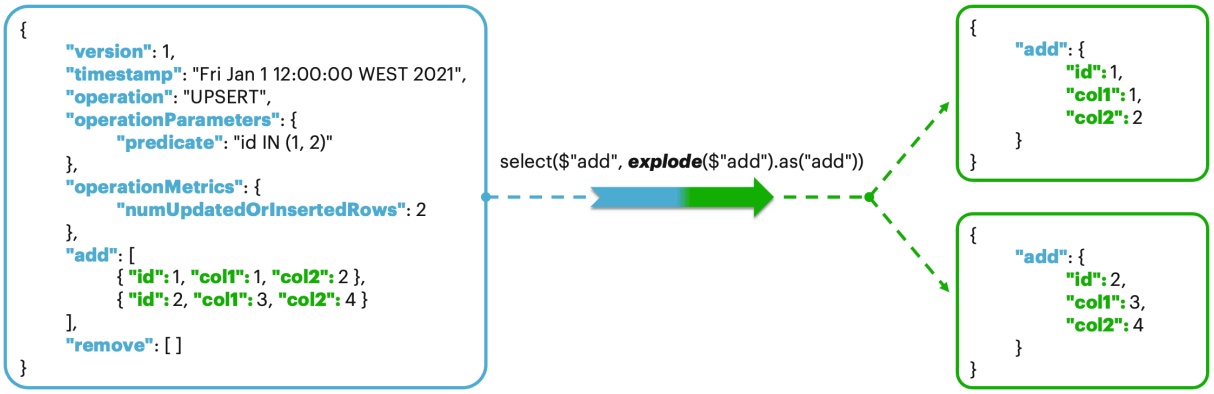


Figura 27: Operação *explode*

Quanto aos dois últimos operadores, isto é, *unionAll* e *exceptAll*, estes realizam a adição e remoção de linhas entre dois *dataframes*, respetivamente. Ambos os operadores não menosprezam os registos repetidos, pelo que também são incluídos no resultado final da operação. Na perspetiva do sistema *HyLake*, a Listagem 16 revela o último estágio de computação da operação de leitura.

```
1 deltaDF.unionAll(addDF).exceptAll(removeDF)
```

Listagem 16: Adição e remoção de linhas entre dois *dataframes* distintos

Neste passo computacional dá-se a aplicação de todas as transformações exercidas sobre o conteúdo dos ficheiros *Parquet* do sistema *Delta Lake*, que, por sua vez, exprimem a versão inicial da tabela *HyLake* (4). Nesta aplicação, as linhas associadas à adição são acrescentadas ao *dataframe* do sistema *Delta Lake* com recurso ao operador *unionAll*, enquanto que a eliminação das linhas que dizem respeito ao campo *remove* é efetuada através do operador *exceptAll*. Para representar melhor o comportamento dos operadores em questão, apresenta-se na Figura 28 um exemplo funcional de ambos.

Com a estratégia adotada na leitura de uma tabela *HyLake* não se antecipa uma evolução significativa no desempenho de tal operação face ao que observa no sistema *Delta Lake*. O tempo despendido a ler uma tabela *HyLake* corresponde essencialmente ao tempo da leitura da tabela *delta*, que contém os ficheiros *Parquet* que exprimem a versão inicial da tabela *HyLake*, mais o tempo da leitura dos documentos alojados em *MongoDB*, que traduzem as transformações desempenhadas. Mesmo que o processo relativo à operação de leitura do sistema *HyLake* seja linear, espera-se atingir índices de desempenho ligeiramente inferiores aos que são evidenciados pelo sistema *Delta Lake*.

Uma vez divulgado o funcionamento da operação de leitura do sistema *HyLake*, apresentam-se nas próximas secções diferentes abordagens na execução do quarto e último passo da Figura 26. A implementação destas alternativas vai determinar aquela que será utilizada nos testes de desempenho finais, onde se comparam os sistemas *HyLake*, *Delta Lake* e uma adaptação do último em *MongoDB*.

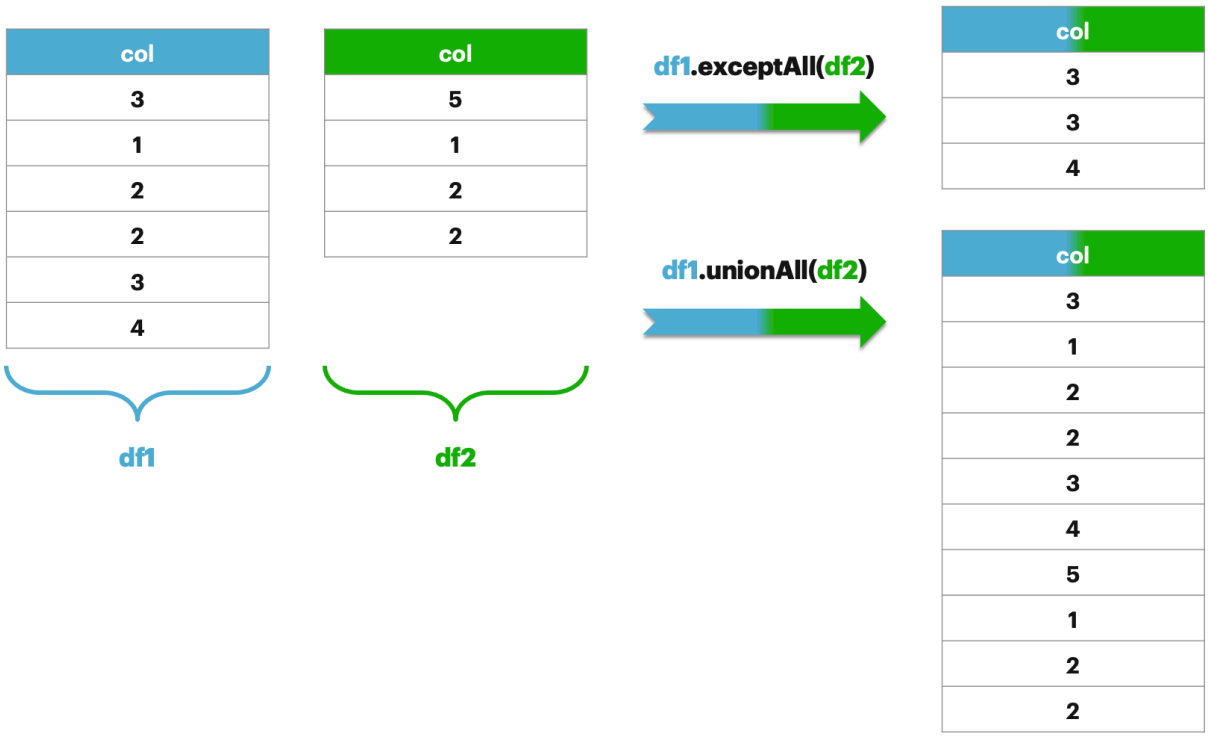


Figura 28: Operação *unionAll* e *exceptAll*

4.4.1 Alternativa *Scala* + *Spark* com aplicação simultânea de transformações

Para desempenhar a reconstrução do estado transacional de uma tabela *HyLake*, esta alternativa da operação de leitura aplica todos os pares de transformações, *add* e *remove*, de uma só vez, utilizando a linguagem de programação *Scala* e a ferramenta *Spark*, em particular o módulo *Spark SQL*. Para isso foi usado o operador *explode* para possibilitar, numa fase posterior da computação, a extração do conteúdo relativo aos campos referidos anteriormente. Inicialmente ocorre a filtragem dos documentos em *MongoDB* até à versão atual da tabela *HyLake*. Imediatamente a seguir a este cálculo dá-se a construção dos *dataframes* que possuem todas as linhas adicionadas e removidas da tabela *HyLake*, desde o momento da sua criação. No fim deste processo são devidamente aplicadas as modificações em causa. A forma como esta última tarefa é realizada é exatamente igual à que é evidenciada na Listagem 16.

4.4.2 Alternativa *Scala* + *Spark* com aplicação sucessiva de transformações

No que toca a esta alternativa da operação de leitura, esta é idêntica à anterior. Contudo, em vez de aplicar as transformações realizadas sobre uma tabela *HyLake* de uma só vez, a aplicação em causa é feita modificação após modificação. Isto é, são extraídas as adições e as remoções de uma determinada versão da tabela mencionada de cada vez. Para desempenhar esta tarefa, os documentos alojados em *MongoDB* são acedidos de forma isolada de maneira a extrair o conteúdo relevante de uma certa versão.

Para além deste comportamento distinto em relação à primeira alternativa desta operação, esta alternativa procede à adição e à remoção das linhas persistidas nos documentos pela ordem inversa daquela que é indicada na Listagem 16. Ou seja, é invocado o operador *exceptAll* seguido da operação *unionAll*. A principal razão pela qual esta abordagem é tomada desta forma deve-se única e exclusivamente a questões de eficiência em termos computacionais. Assim, a remoção das linhas em questão é concretizada sobre um conjunto menor de registos.

4.4.3 Alternativa *MongoDB aggregation pipeline* com aplicação simultânea de transformações

Ao contrário do que se constatou para as alternativas da operação de leitura do sistema *HyLake* presentes nas Secções 4.4.1 e 4.4.2, esta utiliza unicamente a notação disponibilizada por uma base de dados *MongoDB*, isto é, a notação *MongoDB aggregation pipeline*. Globalmente esta versão pode ser vista como a alternativa apresentada na Secção 4.4.1 mas com a introdução da notação referida previamente. Uma vez que pode ocorrer a remoção de linhas previamente adicionadas numa tabela *HyLake*, a aplicação das modificações é rigorosamente igual à da alternativa descrita na Secção 4.4.1.

4.4.4 Alternativa *MongoDB aggregation pipeline* com aplicação sucessiva de transformações

Tendo em consideração todas as caraterísticas da alternativa da Secção 4.4.2, esta alternativa da operação de leitura usufrui a notação *MongoDB aggregation pipeline* para atingir o mesmo propósito. Ou seja, dá-se a aplicação de uma transformação de cada vez, invocando-se o operador *exceptAll* seguido da operação *unionAll* para aplicar corretamente as remoções e as adições das linhas referentes a uma certa versão do estado transacional. De maneira a preservar todo o conteúdo registado em cada transformação efetuada numa tabela *HyLake*, é inicialmente criado um *dataframe* para o efeito. Desta forma, assim que a aplicação das modificações termine, é devolvido corretamente o estado atual da última sob o mesmo formato.

4.5 Implementação

Passando para a implementação do sistema *HyLake*, este foi inicialmente especificado em *Scala* dado que o sistema original também é codificado na mesma linguagem de programação. Contudo, para obter uma maior abrangência a nível de execução, este também foi implementado em *Python* [34].

De modo a encapsular e a abstrair tanto o comportamento como as funcionalidades do sistema em questão, foram instanciadas uma interface, uma classe e quatro subclasses para o efeito. No que toca à interface são anotados cinco métodos, sendo que quatro deles dizem respeito às operações fundamentais do sistema *HyLake*.

A Tabela 9 apresenta as funções da interface do sistema *HyLake*.

Nome da função	Descrição
`clear`	Função que elimina quer os ficheiros *Parquet* relativos à versão inicial da tabela *HyLake* quer os documentos guardados em *MongoDB*. Este método foi implementado apenas para facilitar a execução de testes de desempenho, testes esses que serão exibidos no Capítulo 5. De notar que a remoção da primeira componente é facultativa enquanto que a segunda é de teor obrigatório.
`create`	Método que trata da criação de uma tabela *HyLake* com l linhas e c colunas.
`delete`	Função que remove uma coleção de linhas a partir dos seus identificadores, isto é, as chaves dos registos.
`read`	Método que procede à leitura da tabela *HyLake*.
`upsert`	Função que permite efetuar ou uma atualização ou uma inserção de dados, consoante o conjunto de identificadores disponibilizado.

Tabela 9: Métodos da interface do sistema *HyLake*

A interface em questão é partilhada tanto pela classe mencionada como pelas subclasses de forma a garantir uma especificação válida do sistema *HyLake* e, claro está, a ausência de redundância.

Relativamente à classe do sistema *HyLake*, são definidas diversas variáveis de instância para satisfazer as necessidades impostas pelas suas operações. No instante em que a última é declarada, é no mínimo requerida a indicação da localização da diretoria onde a tabela *delta* irá ser persistida. Para além desta informação, o utilizador pode ainda especificar explicitamente a configuração da conexão à base de dados *MongoDB* onde serão alojados os documentos relativos às transformações efetuadas sobre a tabela *HyLake*. Por omissão, a configuração adotada corresponde a uma ligação local à base de dados referenciada.

A Tabela 10 apresenta as variáveis de instância que são mais relevantes para compreender os excertos de código que foram exibidos neste capitulo. Das variáveis referidas, a variável *version* é aquela possui maior importância. Sendo que existem operações que alteram o estado atual de uma tabela *HyLake*, existe a necessidade de acompanhar o número atual da versão da mesma, transformação após transformação. Dado que este tipo de operações podem ocorrer em simultâneo, existe a possibilidade de verificarem-se múltiplos acessos a esta informação. Consequentemente, poderão suceder-se modificações incorretas sobre esta variável. Para lidar com esta situação, que é caraterística, por exemplo, de sistemas distribuídos, à incógnita *version* é associado o tipo `AtomicInteger`. Com este tipo, é garantido que a escrita de um novo valor é feita atomicamente, isto é, para duas ou mais atualizações desse conteúdo só é possível executar uma nova escrita de cada vez. Assim, o problema denominado por *race conditions* é extinto por completo.

Nome da variável	**Descrição**
`deltaTableName`	Nome da tabela *delta*.
`deltaTablePath`	Localização da tabela *delta* no sistema de ficheiros.
`spark`	Variável que contém a sessão da ferramenta *Spark*.
`sparkContext`	Variável que representa a conexão a um *cluster* da ferramenta *Spark*.
`version`	Versão atual da tabela *HyLake*.
`upserts`	Número de execuções da operação *upsert*.

Tabela 10: Variáveis de instância da classe do sistema *HyLake*

Às variáveis de instância referidas na Tabela 10 juntam-se também os métodos que exprimem as operações fundamentais do sistema *HyLake*. Estas funções podem ser consultadas na Tabela 9. Aos métodos das operações do sistema *HyLake* é ainda adicionado um método privado que é importante salientar, isto é, o método *createRandomDataframe*. Esta função é usada quer na criação de uma tabela deste formato quer no instante em que a operação *upsert* é invocada para auxiliar a geração do conteúdo a ser armazenado. Na eventualidade de se tratar do último caso, esta função possui um comportamento ligeiramente diferente do que é definido por omissão. Em vez de produzir informação de forma aleatória para todas as colunas da estrutura de dados em causa, dá-se, para uma das colunas, a geração determinística de dados, tendo em consideração o valor atual da variável *upserts*. Desta forma, com este comportamento, o conteúdo da tabela *HyLake* varia a cada invocação da operação *upsert* de maneira a garantir a ocorrência de uma modificação pura do seu estado atual.

Para além destes métodos, são disponibilizadas mais três funções que auxiliam a execução de testes unitários sobre o sistema *HyLake*. Estas funções devolvem apenas a informação contida numa determinada variável de instância da classe do sistema *HyLake*. A Tabela 11 exibe os métodos referidos.

Nome da função	Descrição
`getDeltaTableDirectory`	Método que devolve a localização da tabela *delta* no sistema de ficheiros.
`getSpark`	Função que retorna uma nova sessão da ferramenta *Spark* de uma tabela *HyLake*.
`getVersion`	Método que divulga a versão atual de uma tabela *HyLake*.

Tabela 11: Métodos secundários da classe do sistema *HyLake*

Quanto às subclasses, estas foram criadas isoladamente para possibilitar a codificação de múltiplas alternativas da operação de leitura sem haver a necessidade de reescrever o comportamento base do sistema *HyLake*. Como tal, os métodos das operações de criação, atualização, inserção e remoção de dados do último sistema são devidamente herdados, assim como grande parte das variáveis de instância referidas previamente na Tabela 10. Assim, para cada subclasse é apenas implementada uma alternativa distinta desta operação para, no momento da avaliação do sistema em questão, determinar qual aquela que possui um melhor desempenho.

Capítulo 5

Avaliação

De modo a avaliar o sistema *HyLake* procede-se à execução de diversos testes de desempenho com o intuito de o comparar não só com o sistema *Delta Lake* [8] como com a adaptação do último em *MongoDB* [28]. Em relação ao último sistema, este preserva exclusivamente o conteúdo das transações numa base de dados não relacional. Para atingir este propósito é utilizado o mesmo conector [27] que foi usado no sistema *HyLake*. O modelo de armazenamento subjacente a este conector corresponde precisamente ao conteúdo inerente às linhas que atualmente pertencem à tabela em causa. Esta informação equipara-se, por exemplo, aos dados alojados nos campos *add* e *remove* da Figura 10. Contudo, como se pode ver na Figura 29, cada linha armazenada na tabela dá origem a um documento diferente.

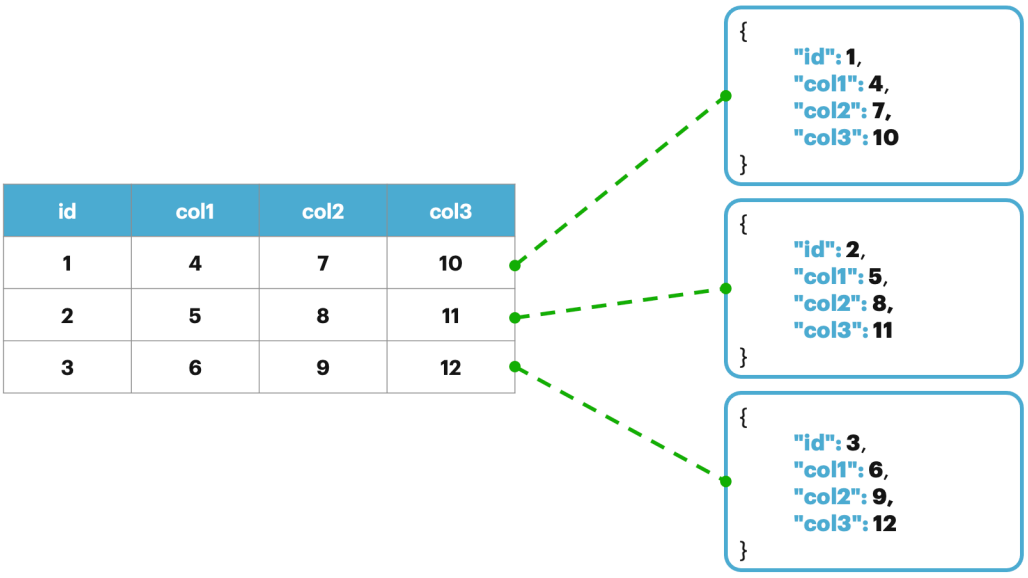


Figura 29: Modelo de armazenamento do conector *MongoDB-Spark*

Desta maneira, aquando da invocação de uma atualização, inserção ou remoção de linhas, o sistema *MongoDB* manipula apenas as linhas afetadas pela operação desencadeada. Ao contrário dos restantes sistemas, esta solução não guarda o estado transacional da tabela de uma forma iterativa, isto é, versão após versão. Em vez disso, o sistema *MongoDB* preserva apenas o estado atual da tabela em causa. Com a introdução desta nova solução é possível determinar qual destes três sistemas é o mais indicado à execução de transações frequentes e de granularidade fina para suportar uma carga de trabalho *OLTP*.

5.1 *Benchmark*

Para obter uma comparação válida entre estes sistemas, foi construído um *benchmark* minimalista, em *Scala* [29], para os diferentes testes de desempenho. O surgimento deste *benchmark* deve-se ao facto da ferramenta *TPC-H* [32] apenas avaliar sistemas analíticos, ou seja, sistemas que realizam operações de leitura sobre dados de múltiplas dimensões e origens. Como para estudar o sistema *HyLake* é necessário medir o desempenho das suas escritas, o uso do *benchmark TPC-H* torna-se prescindível. Assim, optou-se pela elaboração de um novo *benchmark* que permita a leitura e a escrita de dados numa tabela *HyLake*. Para além disso, os resultados dos testes de desempenho deste utensílio podem ser guardados tanto num sistema de ficheiros tradicional como num sistema de ficheiros distribuído, como é o caso do *HDFS (Hadoop Distributed File System)* [2].

Por forma a encapsular e a abstrair o seu comportamento, foram instanciadas uma interface e uma classe para cada um dos três sistemas considerados. Em cada classe são especificados os testes de desempenho das operações de escrita e de leitura. Em ambas as operações dá-se a produção de dados de forma aleatória para injetar nas tabelas criadas. Esta geração de dados é feita através da ferramenta *Spark* [35] e obedece às restrições impostas pela configuração adotada no sistema *HyLake*, em particular a dimensão da tabela utilizada. Na medição do tempo de execução das operações de leitura e de escrita do sistema *HyLake* são ainda produzidos conjuntos aleatórios de identificadores numéricos. No caso da primeira operação, estes valores são passados como argumento tanto à operação *upsert* como à operação de remoção de dados de maneira a assegurar uma quantidade fixa de transformações numa determinada tabela *HyLake*. Quanto à segunda operação, o conjunto de identificadores gerado é passado como argumento à operação *upsert* para possibilitar a avaliação do desempenho das escritas do sistema *HyLake*. De modo a obter resultados fidedignos, no fim de cada teste realiza-se a média dos tempos de execução da operação pretendida.

Quanto ao ambiente computacional, este *benchmark* executa os testes de desempenho em contextos distintos. Os testes executados localmente inserem-se no mesmo ambiente que foi descrito no início do Capítulo 3. No que toca aos testes remotos, estes enquadram-se no ambiente que é caraterizado na Secção 5.3.

A fim de avaliar corretamente o comportamento de tais soluções, as configurações dos testes de desempenho são iguais entre os três sistemas. Nela é especificada, por exemplo, o número de tentativas

de execução por teste. Quanto aos restantes parâmetros, estes variam de acordo com o ambiente computacional usado, isto é, local ou remoto. A Tabela 12 apresenta a única configuração comum entre os testes de desempenho concebidos.

Variável	Valor
Número de tentativas de execução por teste	20

Tabela 12: Configuração global dos testes de desempenho

5.2 *HyLake*

O primeiro teste de desempenho procura confirmar se de facto a inclusão de pontos de verificação no sistema *HyLake* é ou não benéfica na execução de transações em ambientes que utilizam cargas de trabalho *OLTP*.

Tal como foi mencionado na Secção 4.4, foram elaboradas quatro alternativas distintas para a operação de leitura de uma tabela relativa ao sistema *HyLake*. Consequentemente, produzem-se quatro formas diferentes de executar a reconstrução do estado transacional da mesma. Para facilitar a identificação destas propostas na legenda dos respetivos gráficos, estas são numeradas entre um a quatro da seguinte forma:

- Alternativa 1: alternativa mencionada na Secção 4.4.1 onde ocorre a aplicação simultânea de transformações usando a linguagem de programação *Scala* e a ferramenta *Spark*;
- Alternativa 2: proposta referida na Secção 4.4.2 onde se dá a aplicação sucessiva de transformações utilizando a linguagem de programação *Scala* e e a ferramenta *Spark*;
- Alternativa 3: solução exposta na Secção 4.4.3 onde se realiza a aplicação simultânea de transformações usando a notação *MongoDB aggregation pipeline*;
- Alternativa 4: alternativa apontada na Secção 4.4.4 onde acontece a aplicação sucessiva de transformações utilizando a notação *MongoDB aggregation pipeline*.

Para realizar esta experiência, desenhou-se uma nova vertente do sistema *HyLake* que inclui pontos de verificação no seu estado transacional. Nela, à semelhança do que se sucede no sistema *Delta Lake*, é assinalada, de dez em dez transações, a existência de pontos de verificação que, por sua vez, resumem o estado transacional até uma determinada versão. De modo a identificar a ocorrência de um ponto de verificação no sistema *HyLake*, basta observar se a versão correspondente a um determinado documento guardado em *MongoDB* é ou não múltiplo de dez. Para tornar ainda mais evidente o seu reconhecimento,

é acrescentado um novo campo, com o nome *checkpoint*, para informar a presença de um ponto de verificação.

A Listagem 17 exibe um exemplo do novo modelo de armazenamento adotado na execução da operação *upsert* do sistema *HyLake* que inclui pontos de verificação. Nesta listagem destaca-se a existência do campo *checkpoint* e o facto do número da versão do documento alusivo a esta transformação ser múltiplo de dez.

```
{
    "checkpoint": true,
    "version": 10,
    "timestamp": "Fri Jan 1 12:00:00 WEST 2021",
    "operation": "UPSERT",
    "operationParameters": {
        "predicate": "id IN (1,2)"
    },
    "operationMetrics": {
        "numUpdatedOrInsertedRows": 2
    },
    "add": [
        { "id": 1, "col1": 1 },
        { "id": 2, "col1": 2 }
    ],
    "remove": []
}
```

Listagem 17: Modelo de armazenamento do sistema *HyLake* com pontos de verificação

Em relação às configurações inicialmente apontadas na Tabela 12 foram adicionados dois novos parâmetros. O primeiro define o número total de transformações efetuadas sobre a tabela *HyLake*. A esta configuração junta-se também a especificação de uma dimensão fixa para a tabela gerada de modo a permitir uma análise coerente do comportamento do sistema *HyLake*. A Tabela 13 indica as configurações referidas.

Variável	**Valor**
Número de transformações	1, 5, 10, 25, 50
Dimensão da tabela	100 linhas × 10 colunas

Tabela 13: Configuração dos testes de desempenho executados num ambiente computacional local

5.2.1 Operação de leitura

A Figura 30 evidencia o comportamento das diferentes alternativas do sistema *HyLake* sem pontos de verificação ao executar a operação de leitura. Olhando atentamente para esta figura, verifica-se um

comportamento divergente entre os dois pares de alternativas deste sistema. Relativamente à primeira e terceira alternativas, é clara a sua superioridade em termos de desempenho quando comparado com as restantes. Tal como foi referido nas Secções 4.4.1 e 4.4.3, estas duas alternativas aplicam de uma só vez as transformações efetuadas sobre uma tabela *HyLake* aos ficheiros *Parquet* [6] que exprimem a sua primeira versão, de forma a reconstruir corretamente o seu estado transacional. A eficiência da última tarefa nestas alternativas é nitidamente evidenciada na Figura 30. O comportamento de ambas é praticamente idêntico, isto é, para um número de transformações sucessivamente maior, o tempo de execução mantém-se reduzido e constante. Já para a segunda e quarta alternativas isto já não acontece, ou seja, o tempo de execução cresce à medida que a quantidade de modificações aumenta. Este desempenho deve-se ao facto da aplicação referida anteriormente ser feita alteração após alteração.

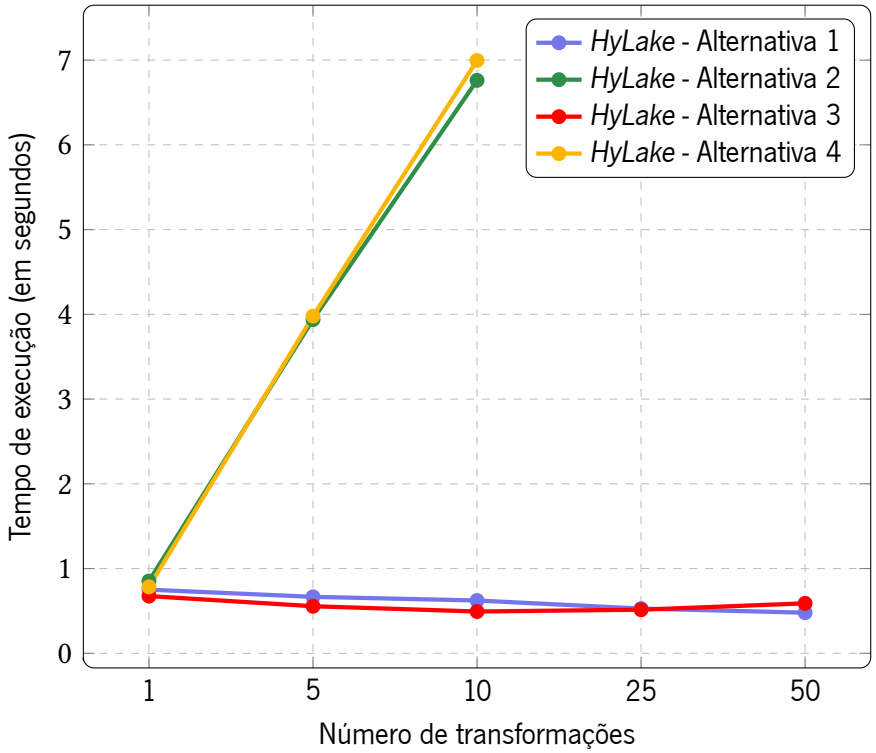


Figura 30: Teste de desempenho relativo à leitura de uma tabela *HyLake* sem pontos de verificação

Com esta experiência, é irrefutável a escolha da primeira e terceira alternativas em detrimento da segunda e quarta alternativas. Desta forma, a partir da Secção 5.3 apenas são consideradas estas alternativas para se realizar a comparação com os diferentes sistemas.

Passando para o sistema *HyLake* com pontos de verificação, é possível averiguar na Figura 31 praticamente o mesmo comportamento para as alternativas consideradas previamente. No entanto, para a segunda e quarta alternativas observa-se uma oscilação em termos de desempenho consoante o número de transformações. Isto ocorre tanto pelos mesmos motivos do sistema anterior como pela ocorrência de pontos de verificação de dez em dez modificações. Por conseguinte, a leitura do estado transacional de uma tabela *HyLake* nesse instante é um pouco mais rápida. Para os restantes casos, o tempo de

execução é superior uma vez que é necessário executar não só a leitura desse mesmo estado, que é potencialmente cada vez maior, como as modificações que foram exercidas sobre a tabela.

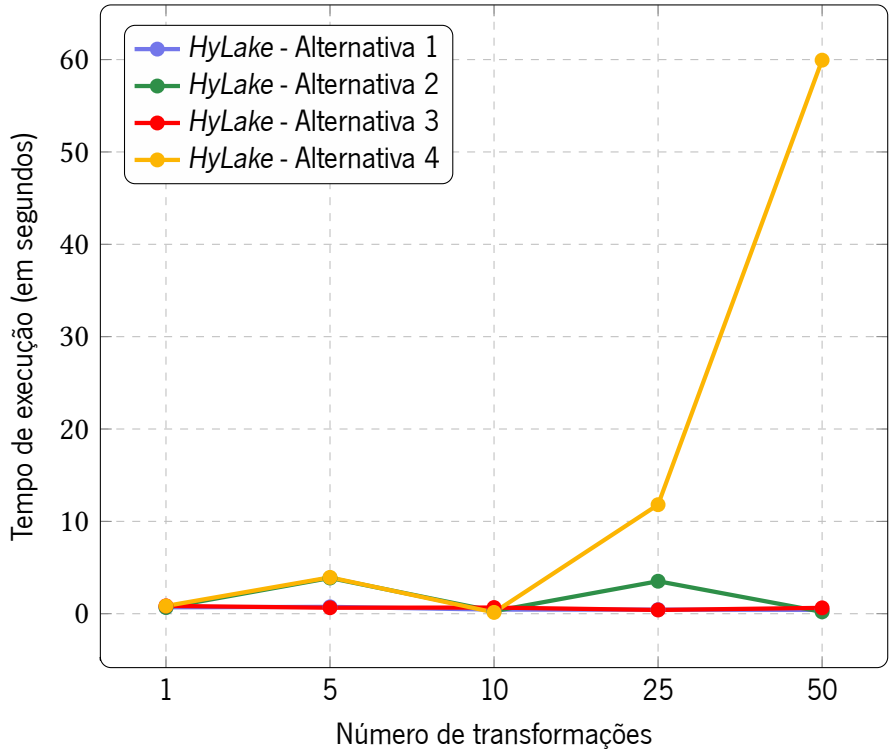


Figura 31: Teste de desempenho relativo à leitura de uma tabela *HyLake* com pontos de verificação

Analisando o desempenho das quatro alternativas da leitura de uma tabela *HyLake* com e sem pontos de verificação, confirma-se que a aplicação sucessiva de transformações tem um impacto negativo na reconstrução do seu estado transacional. Por este motivo, a segunda e a quarta alternativas desta operação não são ponderadas no sistema *HyLake* final. Quanto à primeira e terceira alternativas, estas apresentam um comportamento similar nas duas propostas do último sistema. Dado que a atualização, a inserção e a remoção de dados dependem diretamente da operação de leitura e que, no sistema *HyLake* com pontos de verificação, é escrito de dez em dez transações o estado completo da tabela, a adoção da última proposta prejudicaria a execução de escritas. Assim, na avaliação final deste projeto escolheu-se apenas a primeira e a terceira alternativas do sistema *HyLake* que não inclui pontos de verificação.

5.2.2 Operação de escrita

A Figura 32 apresenta o comportamento das diferentes alternativas do sistema *HyLake* sem pontos de verificação ao executar a operação de escrita. Nela ainda são exibidas a segunda e quarta alternativas para comprovar o péssimo desempenho que estas têm quando comparadas com as restantes. Basta por exemplo visualizar que o tempo dispendido na execução de dez transformações é idêntico aos resultados

da primeira e terceira alternativas para cinquenta modificações. Às quatro alternativas mencionadas junta-se também a primeira versão do sistema *HyLake* com pontos de verificação, versão essa que se encontra representada com uma linha a tracejado. Tal como se pode observar na Figura 32, esta possui um pior comportamento do que a mesma versão no sistema *HyLake* sem pontos de verificação, validando o que foi dito no fim da Secção 5.2.1.

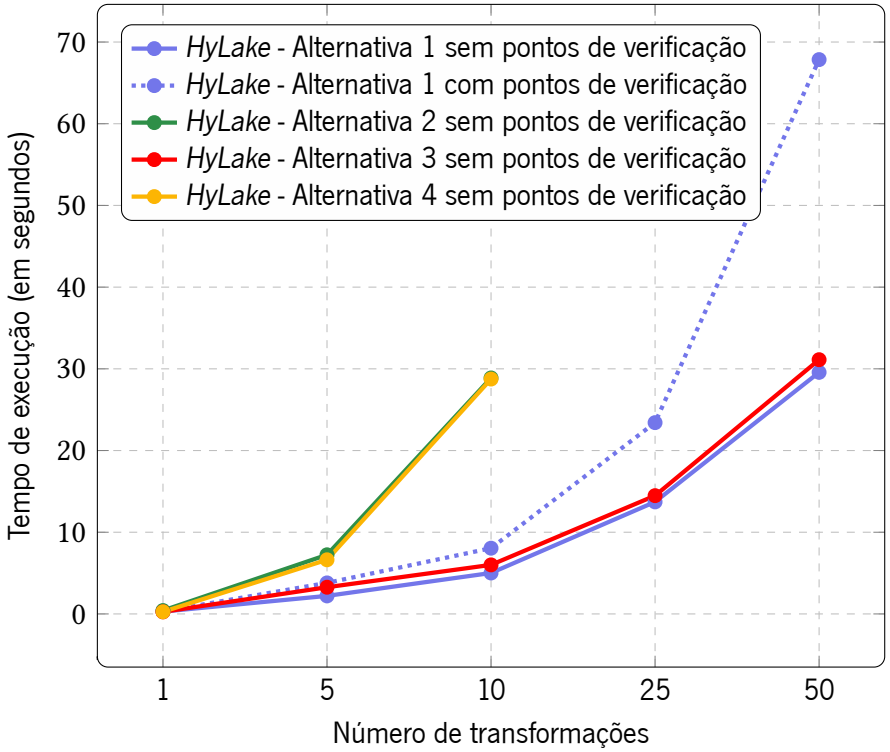


Figura 32: Teste de desempenho relativo à operação de escrita do sistema *HyLake*

Olhando exclusivamente para a primeira e terceira alternativas, cujas linhas estão coloridas a roxo e a vermelho respetivamente, pode-se afirmar que a primeira é aquela que exibe a maior eficiência na execução de escritas. A única diferença que separa estas implementações é a linguagem utilizada, ou seja, a primeira usa a linguagem *Scala*, com o auxílio da ferramenta *Spark*, enquanto que a terceira usufrui a notação *MongoDB aggregation pipeline*. Dada a proximidade entre estas últimas duas versões em termos de desempenho, não é totalmente claro qual delas é que deve ser utilizada na comparação efetuada na Secção 5.3. Com a existência desta incerteza, são ainda consideradas nesse segmento ambas as versões.

5.3 *Delta Lake vs HyLake vs MongoDB*

Com a escolha definitiva da versão final do sistema *HyLake*, avança-se para a realização dos testes de desempenho das operações de leitura e escrita para os três sistemas referidos neste capítulo. À semelhança do que se sucedeu na Secção 5.2, os sistemas *Delta Lake*, *HyLake* e *MongoDB* são sujeitos

aos mesmos testes. Estes são executados tanto localmente como remotamente, pelo que são tomadas diferentes configurações.

Quanto aos testes locais, estes adotam as mesmas configurações que foram apresentadas nas Tabelas 12 e 13. Dado que na Secção 5.2 já foram determinadas as versões mais eficientes do sistema *HyLake*, estas são unicamente utilizadas neste segmento para efeitos de comparação, restando apenas medir o desempenho dos sistemas *Delta Lake* e *MongoDB*. Os resultados destes testes são expostos nas Figuras 33 e 35.

No que toca à execução dos testes de desempenho remotos, estes são conduzidos num *cluster* pertencente à *Google Cloud Platform* [11]. Ao contrário do que se verificou nos testes locais, neste ambiente computacional são avaliados todos os sistemas em causa. O *cluster* mencionado é especificado através do serviço *DataProc* [9] e é constituído por um nó *master* e dois nós *workers* para permitir uma taxa significativa de execuções transacionais. Estes três nós correspondem a máquinas da série *N1* (n1-standard-2) com dois núcleos de processamento e 7,5 *GB* de memória *RAM*. Em relação ao espaço de armazenamento, cada uma das instâncias possui cerca de 50 *GB*. Para além destas três instâncias, é adicionada uma outra máquina que contém uma base de dados não relacional, isto é, *MongoDB*. Esta máquina partilha a mesma série das instâncias mencionadas previamente, sendo que usufrui apenas um *CPU (Central Processing Unit)* e 3,75 *GB* de memória *RAM* (n1-standard-1). Ao contrário do que se constatou nos três nós do *cluster*, esta instância requer uma quantidade reduzida de armazenamento persistente, isto é, cerca de 10 *GB*. Após a realização de múltiplas experiências, sintetizam-se nas Figuras 34 e 36 os resultados obtidos no presente contexto. Por fim, expõem-se ainda na Tabela 14 as configurações exclusivas a este ambiente.

Variável	Valor
Número de linhas afetadas	10
Número de linhas da tabela	1000, 10000, 100000, 1000000
Número de colunas da tabela	10

Tabela 14: Configuração dos testes de desempenho executados num ambiente computacional remoto

Em relação ao parâmetro relativo ao número de linhas afetadas, este é fixado num único valor para facilitar a interpretação dos resultados obtidos. Em suma, aquando da invocação de uma transformação, é executada uma atualização, inserção ou remoção de dados que afeta a quantidade de linhas especificada. Isto é conseguido através da identificação das chaves que determinam univocamente cada linha presente numa determinada tabela.

5.3.1 Operação de leitura

Para iniciar a execução dos testes que dizem respeito à operação de leitura, são criadas tabelas com uma dimensão fixa ou variável para os diversos sistemas, consoante o ambiente computacional usado.

As tabelas são preenchidas de acordo com os parâmetros de configuração desse mesmo ambiente. Dado que nos testes remotos a dimensão das tabelas é superior em relação ao que se verifica nos testes locais, é imprescindível evitar a criação de uma tabela completa ao longo da execução para refletir uma certa quantidade de transformações. Para solucionar este problema eficientemente, são calculadas as diferenças das quantidades das modificações especificadas. Desta forma, assim que a tabela seja instanciada, não existe a repetição da execução de transformações. Aquando da finalização deste processo, dá-se início à leitura propriamente dita, iterando-se no fim o conteúdo da tabela usada.

Olhando em primeiro lugar para a Figura 33, onde são exibidos os resultados da execução local da operação de leitura, é possível desde logo destacar a superioridade, em termos de desempenho, dos sistemas *Delta Lake* e *MongoDB*. A leitura de uma tabela referente ao sistema *MongoDB* resume-se exclusivamente à leitura de documentos que exprimem o conteúdo atual da última. Tal como foi dito no início do Capítulo 5, cada linha pertencente à tabela dá origem a um documento distinto, pela que a sua leitura é efetuada rapidamente. Quanto ao sistema *Delta Lake*, este apresenta um comportamento bastante satisfatório na execução da operação de leitura, sobretudo à medida que o número de transformações aumenta. Ainda que esta solução utilize ficheiros para guardar o estado transacional de uma tabela, esta consegue obter um melhor desempenho do que o do sistema *HyLake*. O comportamento do último deve-se ao facto da operação de leitura se traduzir na leitura quer dos ficheiros *Parquet* guardados numa tabela *delta* quer dos documentos armazenados em *MongoDB*. Consequentemente, observa-se um pobre desempenho.

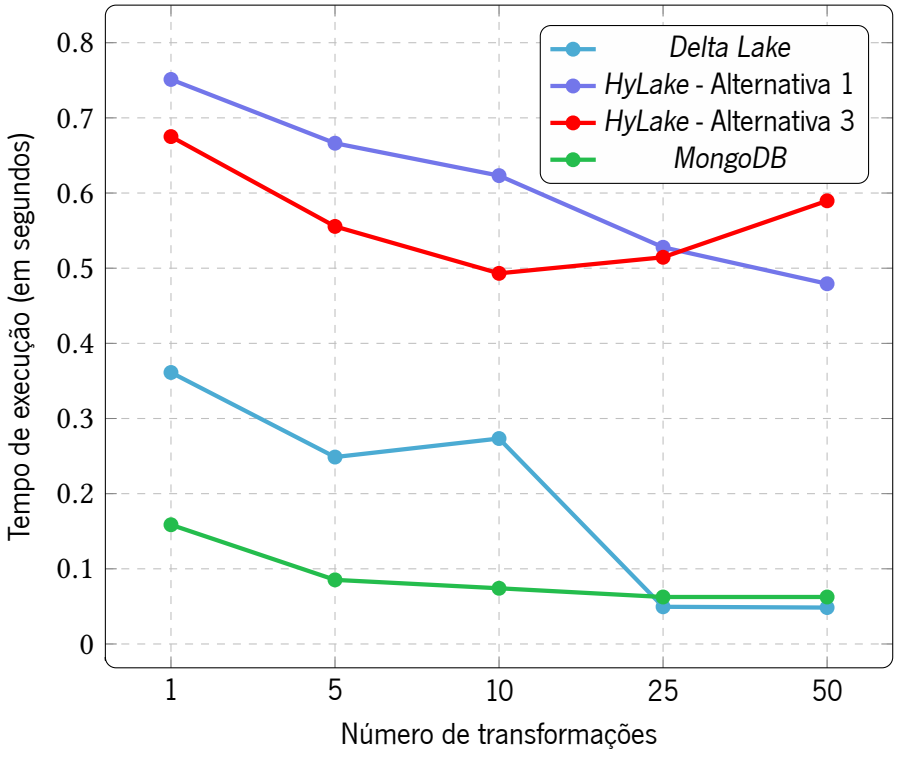


Figura 33: Teste de desempenho relativo à execução local da operação de leitura

Passando para a análise da Figura 34, onde são evidenciados os resultados da execução da operação de leitura num ambiente remoto, pode-se observar um comportamento totalmente distinto para o sistema *MongoDB*. Este contraste em termos de desempenho acontece pelo aumento significativo do número de linhas da tabela considerada. Dado que no sistema *MongoDB* cada linha presente na tabela dá origem a um documento distinto, produzem-se muitos documentos na respetiva base de dados. No caso deste teste chega-se a atingir a geração de um milhão de documentos. Por conseguinte, a leitura de uma tabela pertencente ao sistema *MongoDB* torna-se cada vez mais lenta. Tal como se pode ver na Figura 34 e regressando ao exemplo anterior, a leitura de uma tabela com um milhão de linhas demora quase 18 segundos. Em relação ao sistema *Delta Lake*, este exprime o melhor comportamento na execução da operação de leitura, demonstrando o seu potencial em ambientes analíticos. Ainda assim, para uma tabela com um milhão de linhas, esta solução é ultrapassada, em termos de desempenho, pelo sistema *HyLake*. Tratando-se de um sistema que adota um comportamento híbrido, este consegue manter um desempenho positivo, independentemente da dimensão da tabela considerada. Como tal, o sistema *HyLake* apresenta um bom comportamento em ambientes transacionais que envolvem cargas de trabalho mistas.

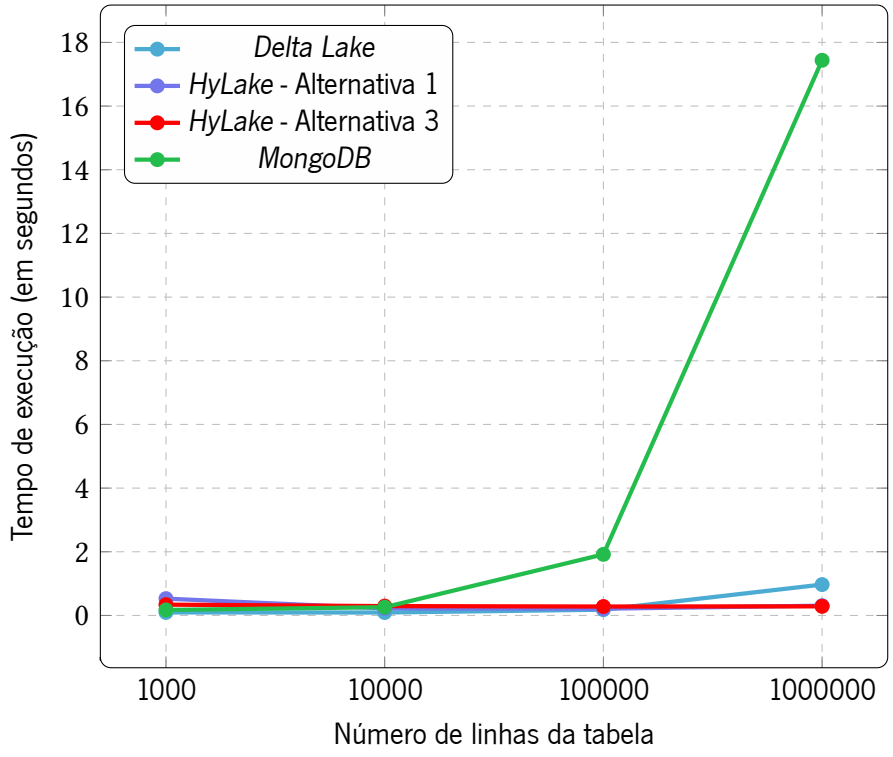


Figura 34: Teste de desempenho relativo à execução da operação de leitura num ambiente remoto

Assim, para cargas de trabalho onde ocorrem leituras de tabelas com uma dimensão reduzida, o sistema *Delta Lake* é o mais apropriado e, para um número significativo de linhas, deverá optar-se pela utilização do sistema *HyLake*.

5.3.2 Operação de escrita

No que toca à operação de escrita, esta desempenha ações que alteram parcialmente o estado transacional de uma tabela referente a um dos três sistemas apontados anteriormente. Para desempenhar tal tarefa, é invocado neste teste de desempenho ou uma inserção ou uma atualização de um conjunto arbitrário de registos. Com a execução da operação *upsert* foi possível medir corretamente o tempo de execução das escritas nos sistemas *Delta Lake*, *HyLake* e *MongoDB*, independentemente do ambiente computacional usado. As Figuras 35 e 36 apresentam os resultados obtidos na execução local e remota de escritas nestes sistemas, respetivamente.

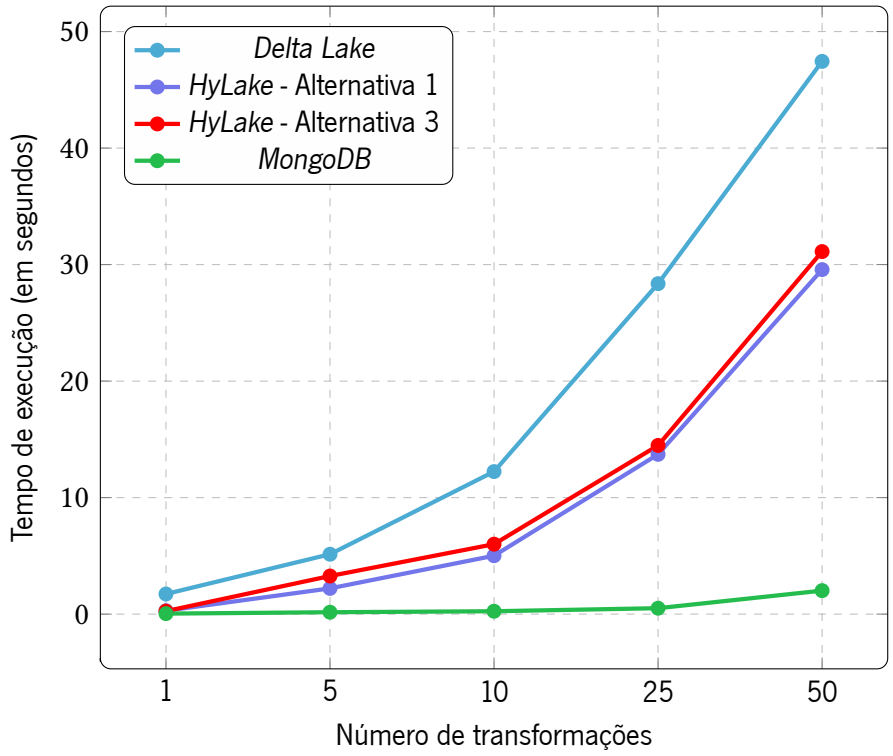


Figura 35: Teste de desempenho relativo à execução local da operação de escrita

Tal como se pode averiguar em ambas as figuras, o sistema *Delta Lake* apresenta o pior comportamento quando comparado com os restantes. Aspetos como o *overhead* obtido na leitura de vários ficheiros contribuem para a existência de poucas atualizações de grandes proporções. Isto prova, juntamente com os resultados das Figuras 33 e 34, que este sistema é mais vocacionado à execução de cargas de trabalho *OLAP*. Quanto aos sistemas *HyLake* e *MongoDB*, estes evidenciam um desempenho superior face ao sistema *Delta Lake*. Globalmente, o sistema *MongoDB* é aquele que possui o melhor desempenho, contudo, para uma tabela com um milhão de linhas (Figura 36), o sistema *HyLake* exibe melhores resultados, demonstrando de certo modo a sua escalabilidade enquanto solução. Neste caso, a atualização ou inserção de dez linhas numa tabela referente ao sistema *MongoDB* é ultrapassada em termos de desempenho pelo registo das mesmas transformações numa tabela *HyLake*. Para as restantes

configurações, a diferença observada entre o sistema *HyLake* e *MongoDB* está relacionada com a filtragem dos documentos relevantes no momento em que ocorre a operação *upsert*. Este pormenor juntamente com a construção customizada de documentos faz com que nestas situações o sistema *HyLake* tenha um desempenho ligeiramente pior em relação ao *MongoDB*.

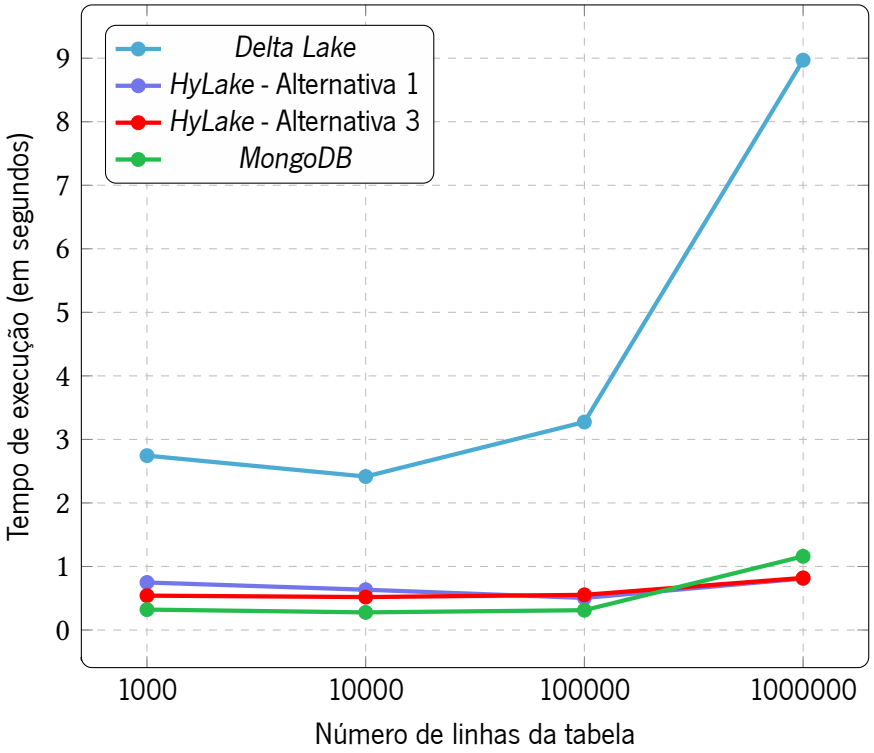


Figura 36: Teste de desempenho relativo à execução da operação de escrita num ambiente remoto

Em síntese, para ambientes transacionais onde são executadas muitas escritas sobre uma tabela com poucas centenas de milhares de linhas, o sistema *MongoDB* é o mais indicado. Caso se trate de uma tabela com muitas centenas de milhares ou até milhões de entradas, o sistema *HyLake* é a melhor escolha.

Capítulo 6

Conclusão

Este trabalho compara dois sistemas existentes, *Delta Lake* [8] e *MongoDB* [28], com uma proposta que visa combinar as vantagens de ambos.

Das experiências realizadas ao longo deste tema de dissertação, confirma-se que o sistema *Delta Lake* é unicamente viável em ambientes *OLAP*. A partir das Figuras 33, 34, 35 e 36, que estão disponíveis na Secção 5.3, pode-se concluir que o sistema anterior apresenta um desempenho positivo na execução de leituras, algo que não acontece na ocorrência de escritas. O tempo despendido a escrever cada ficheiro localmente e o *overhead* obtido ao ler vários ficheiros faz com que o sistema *Delta Lake* só seja utilizável com poucos ficheiros *delta*, isto é, ficheiros com extensão *Parquet* [6] e *JSON*. Por conseguinte, verificam-se poucas atualizações de granularidade grossa que, por sua vez, são caraterísticas de uma carga de trabalho *OLAP*.

Por outro lado, a utilização de uma base de dados *NoSQL*, como é o caso do *MongoDB*, tem um impacto negativo no desempenho de operações analíticas com um número elevado de linhas numa tabela. Desta forma, o último sistema manifesta um ótimo comportamento para a execução de transações inseridas em cargas de trabalho *OLTP*, ou seja, onde se realizam várias atualizações de granularidade fina.

Atendendo a estas observações, o sistema *HyLake* proposto é pois o mais indicado a cargas de trabalho *HTAP*, isto é, a ambientes transacionais híbridos, ao aproximar os melhores casos obtidos com os sistemas anteriores. Para tabelas com muitas centenas de milhares ou até milhões de linhas, esta proposta exibe um desempenho superior em relação aos restantes sistemas elaborados. Isto evidencia a eficiência do sistema *HyLake* na execução quer de leituras quer de escritas. O comportamento testemunhado nesta solução deve-se à sua adaptabilidade na execução das operações de leitura e de escrita. Nas leituras observa-se que o sistema *HyLake* mantém um desempenho semelhante ao do *Delta Lake*, enquanto que nas escritas o comportamento do último é idêntico ao do *MongoDB*. Daqui pode-se inferir que

o propósito desta solução foi atingido, agregando num só sistema os ganhos de desempenho equivalentes ao melhor dos sistemas originais. Como resultado, a introdução do sistema *HyLake* permitiu confirmar a hipótese levantada no início deste trabalho, ou seja, que é possível estender ainda mais a granularidade do sistema *Delta Lake* ao possibilitar a execução de escritas frequentes e de baixas proporções em lagos de dados.

No que toca às duas alternativas salientadas nas Secções 4.4.1 e 4.4.3, é também importante determinar qual aquela que deve ser escolhida para um eventual sistema em produção. Estas, apesar de utilizarem linguagens distintas, seguem a mesma estratégia na reconstrução do estado transacional de uma tabela *HyLake*. Dada a proximidade de ambas as alternativas a nível de desempenho, optou-se por calcular a média dos valores obtidos nos testes para proceder à seleção da melhor alternativa. Com recurso a esta métrica, conclui-se que a terceira alternativa do sistema *HyLake* sem pontos de verificação é a que deve ser adotada, ou seja, aquela que aplica as transformações simultaneamente utilizando a notação *MongoDB aggregation pipeline*.

6.1 Trabalho futuro

Dado que os sistemas *Delta Lake*, *HyLake* e *MongoDB* estão associados a diferentes tipos de cargas de trabalho (*OLAP*, *HTAP* e *OLTP*, respetivamente), seria interessante explorar outras propostas alternativas que se assemelham a estes. O foco deste tema de dissertação passou pela conceção de um sistema que estendesse a granularidade dos dados presentes nas transações executadas pelo sistema *Delta Lake*. Uma vez que apenas se investigou o último sistema e o estado de arte circundante, seria relevante pesquisar e estudar outras soluções que se inserissem no mesmo contexto computacional.

Tal como foi discutido na Secção 5.1, foi construído um novo *benchmark* para possibilitar a avaliação simultânea das operações de leitura e de escrita do sistema *HyLake*. Com a conceção deste *benchmark*, conseguiu-se implementar testes de desempenho mais adaptados ao sistema anterior, extraindo os respetivos resultados mais rapidamente. Todavia, poderiam-se usar outras ferramentas para realizar a mesma tarefa de modo a comprovar a robustez dos resultados obtidos no Capítulo 5. Da mesma forma que se utilizou o *benchmark TPC-H* [32] para estudar o desempenho do sistema *Delta Lake* numa perspetiva analítica, também se conseguiria avaliar, por exemplo, os sistemas *HyLake* e *MongoDB* de um ponto de vista transacional com recurso aos *benchmarks TPC-C* ou *TPC-E*. Com a adoção desta estratégia poderia ser possível compreender ainda melhor a variação da granularidade dos dados em cada um dos sistemas referidos.

Por fim, uma última sugestão seria melhorar a implementação do sistema *HyLake* para suportar a execução de *queries* arbitrárias sobre dados. Tendo em consideração que o sistema *HyLake* foi construído a partir do sistema *Delta Lake*, este disponibiliza através de uma interface chave-valor apenas as suas operações fundamentais, isto é, a atualização, a inserção e remoção de dados e, ainda, a criação de uma tabela deste formato. Como o desenvolvimento dos métodos destas operações teve também por base a

forma como se iriam realizar os testes de desempenho, seria importante proporcionar ao utilizador deste sistema a execução de qualquer tipo de interrogações, independentemente da sua complexidade.

Bibliografia

[1] *44 Noteworthy Big Data Statistics*. 9 de set. de 2021. url: https://www.g2.com/articles/big-data-statistics (ver p. 7).

[2] *Apache Hadoop*. Inglês. 25 de ago. de 2021. url: https://hadoop.apache.org (ver p. 61).

[3] *Apache Hive TM*. 3 de mai. de 2016. url: https://hive.apache.org (ver p. 23).

[4] *Apache Kafka*. 9 de set. de 2021. url: https://kafka.apache.org (ver p. 10).

[5] *Apache ORC • High-Performance Columnar Storage for Hadoop*. Inglês. 8 de jan. de 2021. url: https://orc.apache.org (ver p. 14).

[6] *Apache Parquet*. 2 de mar. de 2021. url: https://parquet.apache.org (ver pp. 3, 10, 14, 36, 45, 46, 64, 72).

[7] M. Armbrust et al. "Spark SQL: Relational Data Processing in Spark". Em: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 27 de mai. de 2015, pp. 1383–1394. isbn: 9781450327589. doi: 10.1145/2723372.2742797 (ver pp. 20, 43, 54).

[8] M. Armbrust et al. "Delta lake: high-performance ACID table storage over cloud object stores". Em: *Proc. VLDB Endow.* 13.12 (1 de ago. de 2020), pp. 3411–3424. issn: 2150-8097. doi: 10.14778/3415478.3415560. url: https://doi.org/10.14778/3415478.3415560 (ver pp. 3, 20, 35, 45, 60, 72).

[9] *Dataproc | Google Cloud*. Português. 1 de set. de 2021. url: https://cloud.google.com/dataproc (ver p. 67).

[10] J. Dean e S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". Em: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. Vol. 51. San Francisco, CA: USENIX Association, jan. de 2004, pp. 137–150. doi: 10.1145/1327452.1327492 (ver p. 18).

[11] *Google Cloud*. Português. 9 de set. de 2021. url: https://cloud.google.com (ver pp. 2, 67).

[12] *How Many People Use Instagram? 95+ User Statistics (2021)*. Inglês. 9 de set. de 2021. url: https://backlinko.com/instagram-users (ver p. 1).

[13] *IBM Docs*. Inglês. 9 de set. de 2021. url: https://www.ibm.com/docs/en/db2/11.5 (ver pp. 3, 8).

[14] Mashamsft. *SQL Server technical documentation - SQL Server*. Inglês. 9 de set. de 2021. url: https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15 (ver pp. 3, 8).

[15] minrk. *findspark*. Inglês. 9 de set. de 2021. url: https://github.com/minrk/findspark (ver p. 36).

[16] *MySQL :: MySQL 8.0 Reference Manual*. Inglês. 9 de set. de 2021. url: https://dev.mysql.com/doc/refman/8.0/en (ver pp. 3, 8).

[17] *Oracle Database 21c - Get Started*. Inglês. 8 de set. de 2021. url: https://docs.oracle.com/en/database/oracle/oracle-database/21/index.html (ver pp. 3, 8).

[18] *PostgreSQL 13.4 Documentation*. Inglês. 12 de ago. de 2021. url: https://www.postgresql.org/docs/current (ver pp. 3, 8).

[19] *Press Release - Domo Releases Eighth Annual "Data Never Sleeps" Infographic | Domo*. Inglês. 9 de set. de 2021. url: https://www.domo.com/news/press/domo-releases-eighth-annual-data-never-sleeps-infographic (ver p. 7).

[20] *Project Jupyter*. 5 de ago. de 2021. url: https://jupyter.org (ver p. 35).

[21] *PyMongo 3.12.0 Documentation — PyMongo 3.12.0 documentation*. Inglês. 15 de jul. de 2021. url: https://pymongo.readthedocs.io/en/stable (ver p. 36).

[22] *PySpark Documentation — PySpark 3.1.2 documentation*. 27 de mai. de 2021. url: https://spark.apache.org/docs/latest/api/python/index.html (ver p. 36).

[23] *ScalaTest*. 9 de set. de 2021. url: https://www.scalatest.org (ver p. 36).

[24] *Serviços de Computação na Cloud | Microsoft Azure*. Português. 9 de set. de 2021. url: https://azure.microsoft.com/pt-pt (ver p. 2).

[25] *Serviços de nuvem – Amazon Web Services (AWS)*. Português. 7 de set. de 2021. url: https://aws.amazon.com/pt (ver p. 2).

[26] *Social Media Statistics: Top Social Networks by Popularity*. Inglês. 28 de mai. de 2021. url: https://dustinstout.com/social-media-statistics (ver p. 1).

[27] *Spark Connector Scala Guide — MongoDB Spark Connector*. Inglês. 14 de jun. de 2021. url: https://docs.mongodb.com/spark-connector/current/scala-api (ver pp. 36, 39, 46, 47, 60).

[28] *The most popular database for modern apps*. Inglês. 9 de set. de 2021. url: https://www.mongodb.com (ver pp. 4, 35, 45, 60, 72).

[29] *The Scala Programming Language*. 9 de set. de 2021. url: https://www.scala-lang.org (ver pp. 19, 35, 53, 61).

[30] *Top 20 Big Data Statistics for 2020 | Sigma Computing*. Inglês. 6 de jul. de 2021. url: https://www.sigmacomputing.com/blog/top-20-big-data-statistics-for-2020 (ver p. 6).

[31] *Total data volume worldwide 2010-2025 | Statista*. Inglês. 9 de set. de 2021. url: https://www.statista.com/statistics/871513/worldwide-data-created (ver p. 6).

[32] *TPC-H Homepage*. Inglês. 9 de set. de 2021. url: http://www.tpc.org/tpch (ver pp. 35, 61, 73, 78).

[33] *Welcome to Apache Avro!* 17 de mar. de 2021. url: https://avro.apache.org (ver p. 14).

[34] *Welcome to Python.org*. Inglês. 9 de set. de 2021. url: https://www.python.org (ver pp. 19, 35, 57).

[35] M. Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". Em: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, abr. de 2012, pp. 15–28. url: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia (ver pp. 3, 19, 61).

Apêndice A

Apêndice

A Figura 37 evidencia o esquema do conjunto de dados da ferramenta *TPC-H* [32]. Nesta ilustração são exibidas as chaves primárias e estrangeiras de cada tabela, sendo que sob cada uma das mesmas é apresentada a respetiva cardinalidade. Esta última informação pode ser consultada na Tabela 8.

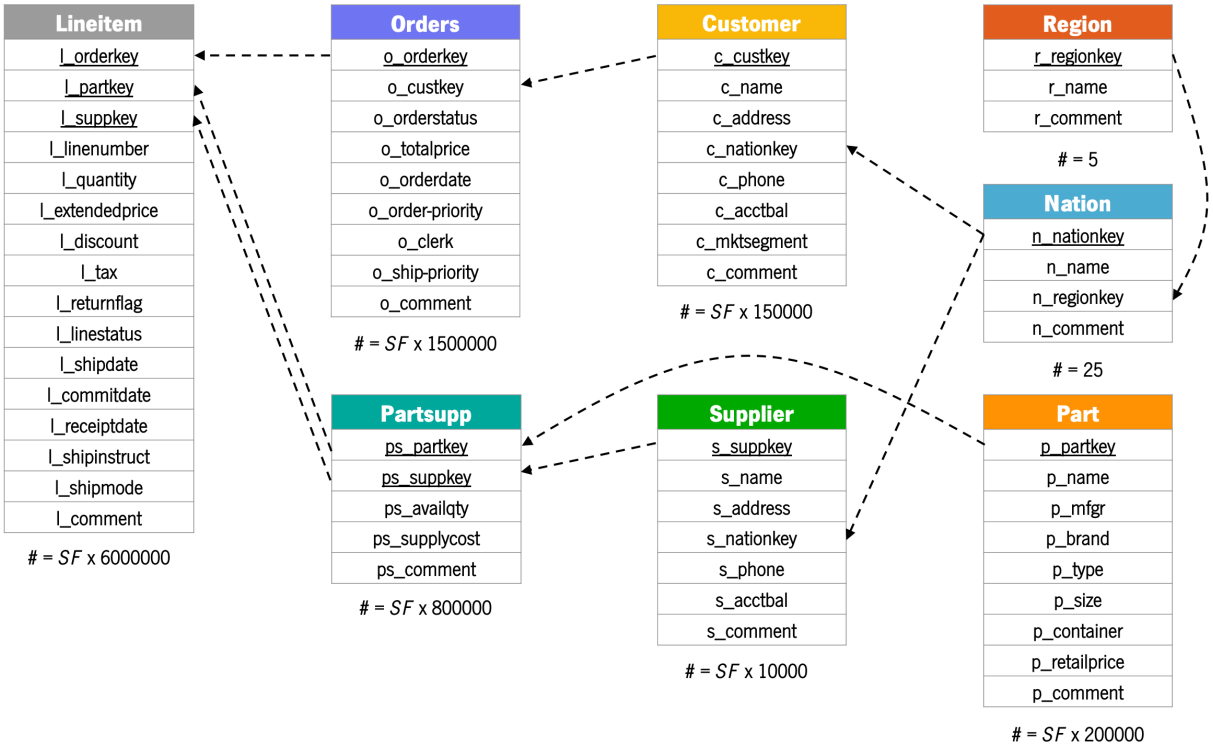


Figura 37: Esquema do conjunto de dados da ferramenta *TPC-H*