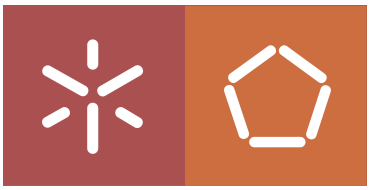


Universidade do Minho
Escola de Engenharia

# Bases de Dados NoSql

## Trabalho prático

### Library Database Project

---

MEI - 1º Ano - 2º Semestre

**Trabalho realizado por:**

PG57542 - Francisco Lameirão
PG57588 - Matilde Fernandes
PG57891 - Maya Gomes
PG57902 - Rui Cerqueira

Braga, 2 de junho de 2025

# Índice

# 1. Introdução

No contexto atual da gestão de dados, assistimos a uma evolução constante dos paradigmas de bases de dados, onde os sistemas relacionais tradicionais coexistem com soluções NoSQL emergentes, cada uma oferecendo vantagens específicas consoante os requisitos da aplicação.

Este trabalho prático, desenvolvido no âmbito da disciplina de Bases de Dados NoSQL, tem como propósito fundamental equipar os estudantes com as competências necessárias para compreender, analisar e implementar diferentes paradigmas de bases de dados, explorando as suas aplicações práticas no design e implementação de sistemas de informação modernos.

# 2. Contexto

O projecto centra-se na migração de um sistema de gestão de livraria, originalmente implementado numa base de dados relacional Oracle, para dois modelos de bases de dados não-relacionais distintos: uma base de dados orientada a documentos utilizando MongoDB e uma base de dados de grafos implementada em Neo4j. Esta escolha reflecte a diversidade de abordagens NoSQL disponíveis, permitindo uma análise comparativa abrangente das diferentes filosofias de armazenamento e consulta de dados.

O sistema de gestão de livraria *Gravity Bookstore* serve como caso de estudo, apresentando uma estrutura complexa que inclui múltiplas entidades interrelacionadas como livros, autores, clientes, encomendas e histórico de transações. Esta complexidade é propositadamente escolhida para demonstrar como diferentes paradigmas de bases de dados lidam com relacionamentos entre entidades, desde as tradicionais relações tabulares do modelo relacional até às estruturas embebidas do modelo de documentos e às conexões explícitas do modelo de grafos.

Os objectivos deste trabalho estendem-se além da simples conversão de esquemas, abrangendo a migração completa de objectos de base de dados incluindo triggers, procedimentos, índices, garantindo que as regras de negócio, integridade de dados e automatização do sistema sejam adequadamente replicadas nos novos paradigmas NoSQL.

# 3. Exploração da Base de Dados fornecida

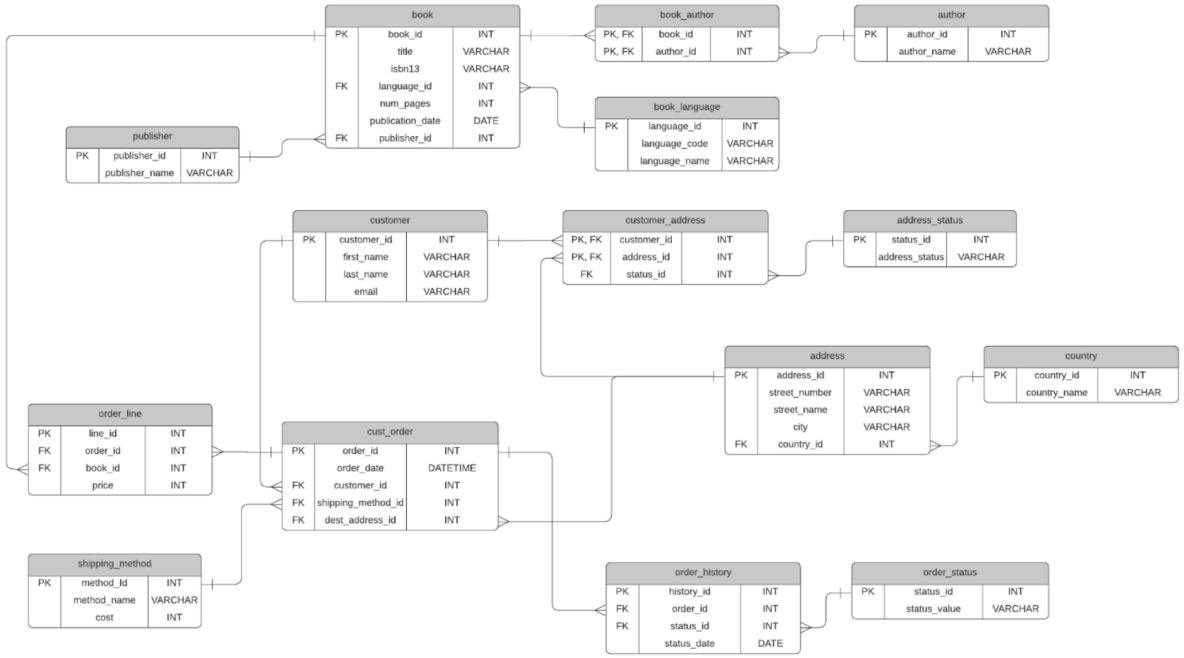



Figura 1: Modelo lógico da base de dados.

A base de dados relacional do sistema *Gravity Bookstore* apresenta uma arquitetura bem estruturada que modela as operações típicas de uma livraria através de quinze tabelas interconectadas. O núcleo do sistema centra-se na gestão de livros, onde a tabela *book* armazena as informações fundamentais como título, ISBN, número de páginas e data de publicação, estabelecendo relacionamentos com as tabelas *publisher* para editoras e *book_language* para idiomas. A relação muitos-para-muitos entre livros e autores é adequadamente tratada através da tabela de associação *book_author*, que liga as entidades *book* e *author*. A gestão de clientes é implementada através da tabela *customer*, que se relaciona com um sistema complexo de endereços. Esta estrutura permite que cada cliente tenha múltiplos endereços através da tabela de junção *customer_address*, onde cada endereço pode ter diferentes estados (atual ou antigo) controlados pela tabela *address_status*. A tabela *address* contém os detalhes geográficos específicos e refere-se à tabela *country* para informações do país, criando uma hierarquia geográfica consistente. O processo de encomendas é modelado através da tabela central *cust_order*, que regista informações como data da encomenda, cliente, método de envio e endereço de destino. Cada encomenda pode conter múltiplos livros através da tabela *order_line*, que especifica a quantidade e preço de cada item. O acompanhamento do estado das encomendas é assegurado pelo sistema de histórico implementado nas tabelas *order_history* e *order_status*, permitindo rastrear toda a evolução de uma encomenda desde a sua criação até à entrega final. Esta estrutura relacional normalizada garante a integridade referencial através de chaves estrangeiras bem definidas e minimiza a redundância de dados, características fundamentais dos sistemas relacionais tradicionais.

## 3.1. Indexes

```
CREATE INDEX idx_book_title ON book(title);
CREATE INDEX idx_customer_email ON customer(email);
CREATE INDEX idx_order_date ON cust_order(order_date);
CREATE INDEX idx_address_country ON address(country_id);
COMMIT;
```

Os índices implementados no sistema *Gravity Bookstore* foram estrategicamente selecionados para otimizar as operações de consulta mais frequentes do sistema:

- *idx_book_title*: Este índice sobre o campo título da tabela *book* é fundamental para acelerar as pesquisas de livros por nome, o que representa uma das operações mais comuns numa livraria. O índice permite que os clientes encontrem rapidamente os livros desejados através de consultas alfabéticas ou de pesquisa parcial por título, evitando percorrer a tabela de livros por completo.
- *idx_customer_email*: O índice no campo email da tabela *customer* otimiza as operações de autenticação e identificação de clientes. Dado que o email é tipicamente utilizado como identificador único para login e comunicações, este índice garante que as verificações de unicidade e as consultas de autenticação sejam executadas de forma eficiente.
- *idx_order_date*: Este índice na data da encomenda (*order_date*) da tabela *cust_order* é essencial para relatórios temporais e consultas de histórico. O índice facilita a geração de relatórios de vendas por período, consultas de encomendas recentes e análises de tendências temporais, operações que podem ser críticas para a gestão do negócio.
- *idx_address_country*: O índice sobre *country_id* na tabela *address* acelera as consultas geográficas e de agrupamento por país. É particularmente útil para relatórios de distribuição geográfica de clientes ou ainda análises de mercado por localização.

## 3.2. Procedure

```
CREATE OR REPLACE PROCEDURE update_order_status (
    p_order_id IN INT,
    p_status_id IN INT
) AS
    v_previous_status_id INT;
BEGIN
    SELECT status_id
    INTO v_previous_status_id
    FROM order_history
    WHERE order_id = p_order_id
    ORDER BY status_date DESC
    FETCH FIRST ROW ONLY;

    IF v_previous_status_id != p_status_id THEN

        INSERT INTO order_history (history_id, order_id, status_id, status_date)
        VALUES (seq_orderhist.NEXTVAL, p_order_id, p_status_id, SYSDATE);

        DBMS_OUTPUT.PUT_LINE('Status atualizado com sucesso para o pedido ' ||
p_order_id);
    ELSE
        DBMS_OUTPUT.PUT_LINE('O status do pedido ' || p_order_id || ' já está como
' || p_status_id);
```

```
    END IF;
END;
```

A *procedure update_order_status* gere de forma consistente e controlada as atualizações do estado das encomendas. A *procedure* recebe como parâmetros o identificador da encomenda (*p_order_id*) e o novo estado (*p_status_id*). A sua lógica interna primeiro verifica o estado atual da encomenda consultando o registo mais recente na tabela *order_history*. Apenas procede à atualização se o novo estado for diferente do atual, evitando registos duplicados desnecessários. Quando efetua uma mudança de estado, insere automaticamente um novo registo no histórico com a data atual, mantendo assim um trilho completo de todas as transições de estado. O procedimento inclui *feedback* através de mensagens *DBMS_OUTPUT*, informando sobre o sucesso da operação ou indicando quando não há alterações a efetuar.

## 3.3. Views

```
-- View list books and their authors
CREATE OR REPLACE VIEW book_with_authors AS
SELECT b.book_id, b.title, a.author_name
FROM book b
JOIN book_author ba ON b.book_id = ba.book_id
JOIN author a ON ba.author_id = a.author_id;

-- View to list orders and their current status
CREATE OR REPLACE VIEW orders_with_status AS
SELECT o.order_id, o.order_date, os.status_value
FROM cust_order o
JOIN order_history oh ON o.order_id = oh.order_id
JOIN order_status os ON oh.status_id = os.status_id
WHERE oh.status_date = (SELECT MAX(status_date)
                        FROM order_history
                        WHERE order_id = o.order_id);

COMMIT;
```

As *views* implementadas fornecem perspetivas simplificadas e agregadas dos dados para consultas frequentes:

- *book_with_authors*: Esta *view* apresenta uma perspetiva desnormalizada da relação muitos-para-muitos entre livros e autores. Combina informações das tabelas *book, book_author e author* para fornecer uma lista direta de livros com os respetivos autores. É particularmente útil para catálogos de produtos e pesquisas de livros por autor, eliminando a necessidade de múltiplas junções nas consultas de aplicação.
- *orders_with_status*: Esta *view* fornece uma perspetiva atual do estado de todas as encomendas. Através de uma subconsulta correlacionada, identifica o estado mais recente de cada encomenda na tabela *order_history* e apresenta essa informação juntamente com os dados básicos da encomenda. Pode ser fundamental para *dashboards* de gestão ou ainda relatórios de estado de encomendas oferecendo uma interface simplificada para aceder ao estado atual sem a necessidade da lógica complexa de agregação.

### 3.4. Triggers

```
CREATE OR REPLACE TRIGGER validate_email
BEFORE INSERT OR UPDATE ON customer
FOR EACH ROW
BEGIN
    IF NOT REGEXP_LIKE(:NEW.email, '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]
{2,}$') THEN
        RAISE_APPLICATION_ERROR(-20003, 'Invalid email format.');
    END IF;
END;
/

CREATE OR REPLACE TRIGGER insert_order_history
AFTER INSERT ON cust_order
FOR EACH ROW
BEGIN
    INSERT INTO order_history (order_id, status_id, status_date)
    VALUES (:NEW.order_id, 1 , SYSDATE);
END;
/


CREATE OR REPLACE TRIGGER prevent_book_deletion
BEFORE DELETE ON book
FOR EACH ROW
DECLARE
    v_count NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_count FROM order_line WHERE book_id = :OLD.book_id;
    IF v_count > 0 THEN
        RAISE_APPLICATION_ERROR(-20002, 'Cannot delete book as it exists in
orders.');
    END IF;
END;
/
```

Os triggers implementados garantem a integridade de dados e automatizam processos críticos:

- *validate_email*: Este trigger de validação executa-se antes de inserções ou atualizações na tabela *customer*, garantindo que todos os endereços de email sigam um formato válido. O trigger utiliza uma expressão regular para verificar a estrutura do email, incluindo a presença de caracteres válidos, do símbolo @ e do domínio apropriado. Levanta uma exceção personalizada (erro −20003) quando deteta formatos inválidos, prevenindo a inserção de dados incorretos que poderiam comprometer comunicações futuras com clientes.
- *insert_order_history*: Este trigger executa-se após cada inserção na tabela *cust_order*, garantindo que todas as novas encomendas tenham imediatamente um registo inicial no histórico de estados. Insere automaticamente um registo na tabela *order_history* com o estado inicial (*status_id* = 1, presumivelmente "Criada" ou "Pendente") e a data atual. Esta automatização garante consistência no sistema de rastreamento e elimina a possibilidade de encomendas sem histórico de estado.
- *prevent_book_deletion*: Este *trigger* de proteção executa-se antes de tentativas de eliminação na tabela *book*, impedindo a remoção de livros que já foram encomendados. Verifica se existem registos na tabela *order_line* que referenciem o livro a ser eliminado e, em caso afirmativo, levanta uma exceção

(erro –20002) que impede a operação. Esta proteção é crucial para manter a integridade referencial e preservar o histórico comercial, evitando inconsistências nos dados de encomendas históricas.

## 3.5. Queries

Nesta secção iremos apresentar um conjunto de *queries* desenvolvidas para extrair informações da base de dados e que visam a responder a certas questões que podem ser úteis para o gerenciamento da *bookstore*. Neste contexto, a análise dos dados desempenha um papel na melhoria da qualidade e gestão de recursos. As queries podem ser visualizadas na pasta *bookstore* do trabalho.

- **QUERY 1:** Listar todos os livros com seus respectivos autores e editoras.

**Objetivo:** Demonstra JOINs múltiplos e relacionamentos *many-to-many*. Esta *query* estabelece a base para consultas complexas ao demonstrar como navegar através de relacionamentos *many-to-many*. Ela utiliza múltiplos JOINs para conectar as tabelas *book, book_author, author, publisher e book_language*. A função *LISTAGG* permite agregar múltiplos autores de um mesmo livro numa única linha, evitando duplicação de registos. Esta consulta é essencial para relatórios de catálogo e serve como exemplo de como gerir relacionamentos complexos numa base de dados normalizada.

- **QUERY 2:** Top 10 dos clientes com mais encomendas.

**Objetivo:** Análise do comportamento de clientes. Esta análise identifica os clientes mais ativos da livraria, proporcionando *insights* valiosos para estratégias de fidelização e marketing. Através da agregação de dados de encomendas e linhas de encomenda, a *query* calcula tanto o número total de encomendas quanto o valor monetário gasto por cada cliente. A limitação aos top 10 clientes permite focar nos clientes mais importantes para o negócio, facilitando futuramente, por exemplo, a implementação de programas VIP ou campanhas personalizadas.

- **QUERY 3:** Livros mais vendidos por quantidade

**Objetivo:** Análise dos produtos mais populares. Esta *query* é fundamental para a gestão de inventário e estratégias comerciais, ela identifica os títulos com melhor performance de vendas. Além da quantidade vendida, calcula métricas como preço médio e número de clientes únicos que compraram cada livro. Esta informação é crucial para decisões de reposição de *stock*, negociações com editoras, e identificação de tendências de mercado que podem influenciar futuras aquisições.

- **QUERY 4:** Análise de encomendas por país

**Objetivo:** Distribuição geográfica das vendas. Esta análise geográfica é útil para compreender a distribuição internacional dos clientes e otimizar estratégias de marketing regional. Calcula métricas por país incluindo número de clientes únicos, total de encomendas, receita total e valor médio por encomenda. Os resultados podem orientar decisões sobre expansão de mercados, políticas de envio, e adaptação de catálogos para diferentes regiões geográficas.

- **QUERY 5:** Estado atual das encomendas.

**Objetivo:** Monitorização operacional. Esta *query* de monitorização operacional fornece uma visão em tempo real do estado das encomendas, essencial para a gestão da cadeia de fornecimento. Identifica o estado mais recente de cada encomenda através do histórico de estados e calcula distribuições percentuais. Esta informação é crítica para identificar *bottlenecks* no processo de *fulfillment*, medir eficiência operacional, e garantir que os clientes recebem atualizações adequadas sobre suas encomendas.

- **QUERY 6:** Autores mais produtivos.

**Objetivo:** Análise de catálogo por autor. Esta análise de catálogo examina a produtividade dos autores representados na livraria, considerando não apenas o número de livros publicados, mas também a diversidade de editoras e o sucesso comercial. Inclui métricas temporais (primeira e última publicação)

e de vendas, permitindo identificar autores consistentemente produtivos versus aqueles com picos de atividade. Esta informação é valiosa para negociações comerciais e decisões editoriais.

- **QUERY 7:** Análise temporal de vendas (por mês).

**Objetivo:** Tendências de vendas ao longo do tempo. A análise temporal é essencial para identificar padrões sazonais, tendências de crescimento, e planear estratégias futuras. Esta *query* agrega vendas por mês, calculando número de encomendas, quantidade de livros vendidos, e receita total. Os dados temporais permitem identificar períodos de maior atividade, avaliar o impacto de campanhas de marketing, e fazer previsões para planeamento financeiro.

- **QUERY 8:** Clientes com endereços múltiplos.

**Objetivo:** Análise da gestão de endereços. Esta *query* identifica clientes com múltiplos endereços registados, uma informação importante para a gestão logística e experiência do cliente. Clientes com múltiplos endereços podem indicar diferentes necessidades, maior fidelidade, ou complexidade na gestão de entregas. A concatenação de todos os endereços com seus estados permite uma visão completa da situação de cada cliente para otimização do processo de envio.

- **QUERY 9:** Livros por idioma e editora.

**Objetivo:** Análise do catálogo por características. Esta análise bidimensional do catálogo examina a distribuição de livros por idioma e editora, fornecendo *insights* sobre a diversidade e foco editorial da livraria. Calcula métricas como número médio de páginas, amplitude temporal das publicações, e concentração de títulos. Esta informação é valiosa para avaliar lacunas no catálogo, negociar com editoras, e compreender a estratégia de diversificação linguística e editorial.

- **QUERY 10:** Métodos de envio mais utilizados.

**Objetivo:** Análise logística A análise dos métodos de envio é crucial para otimizar a logística e custos operacionais. Esta *query* calcula a utilização percentual de cada método, valor médio das encomendas por método, e receita total gerada. Os resultados podem orientar negociações, ajustes de preços de envio, e decisões sobre quais métodos manter, expandir, ou descontinuar com base na preferência dos clientes e rentabilidade.

- **QUERY 11:** Livros nunca vendidos.

**Objetivo:** Identificação de stock parado. Esta análise identifica livros que nunca foram vendidos, uma informação crítica para gestão de inventário e decisões comerciais. Calcula há quantos dias cada livro está disponível sem vendas, permitindo identificar títulos que podem necessitar de promoções, reavaliação de preço, ou remoção do catálogo.

- **QUERY 12:** Análise de clientes por fidelidade.

**Objetivo:** Segmentação de clientes. Esta análise avançada segmenta clientes em categorias (VIP, Fiel, Regular, Novo) com base em critérios de frequência de compra e valor gasto. Utiliza *Common Table Expressions (CTEs)* (consultas temporárias) para calcular métricas de *lifetime value* e comportamento temporal. A segmentação permite implementar estratégias de marketing diferenciadas, programas de fidelidade personalizados, e alocação eficiente de recursos de relacionamento com cliente.

- **QUERY 13:** Relatório de performance por editora.

**Objetivo:** Análise de parceiros comerciais. Esta *query* avalia a performance de cada editora, considerando diversidade de catálogo, sucesso comercial, e alcance linguístico. Calcula métricas como a média de vendas por livro, receita total gerada, e número de autores únicos representados. Esta análise é fundamental para negociações comerciais, decisões sobre parcerias estratégicas, e avaliação do retorno sobre investimento em diferentes editoras.

# 4. MongoDB

## 4.1. Coleções

Neste tópico iremos apresentar as coleções criadas no MongoDB, detalhando o trabalho realizado e as estratégias de desenvolvimento com base na nossa análise. Foram desenvolvidas 3 coleções através de código python, com dados provenientes da base de dados Oracle fornecida pela equipa docente. Optou-se por uma estrutura simplificada em apenas três coleções principais: *books*, *customers* e *orders*. Esta decisão baseia-se nos princípios de modelagem orientada a documentos do MongoDB, que favorecem a desnormalização controlada e o agrupamento lógico de dados relacionados, de modo a otimizar a performance de leitura e a facilitar a consulta de dados. Para fazer a ligação entre ambas as bases de dados usamos as bibliotecas oracledb e pymongo, tal como estudado nas aulas práticas. Esta ligação permite uma integração eficaz e otimizada entre as duas bases de dados.

### 4.1.1. Coleção *books*

Esta coleção armazena informações sobre os livros da nossa *bookstore* desde o autor do livro e os seus dados, a linguagem do livro e o seu editor. A coleção possui um total de 11127 livros.

A coleção *books* centraliza todas as informações relevantes sobre os livros, incluindo autores, linguagem e editor. Como estes dados são frequentemente consultados em conjunto, especialmente em contextos de pesquisa ou visualização de detalhes de um livro, a sua agregação num único documento reduz a necessidade de múltiplas operações de junção (*joins*), que são dispendiosas em bases de dados não relacionais.

### 4.1.2. Coleção *customers*

Esta coleção armazena informações sobre os *customers* da *bookstore* desde o nome, a morada e o país. A coleção possui um total de 2000 *customers*.

A coleção *customers* reúne os dados pessoais dos clientes, incluindo moradas e país. Uma vez que um cliente pode ter várias moradas e uma morada pode estar associada a vários clientes, optou-se por incorporar diretamente essa informação no documento do cliente, simplificando o acesso e a manutenção destes dados, que tendem a ser utilizados em conjunto (ex. envio de encomendas, faturação).

### 4.1.3. Coleção *orders*

Esta coleção armazena informações sobre os pedidos da *bookstore* desde a lista dos pedidos do utilizadores, a lista de livros que fazem parte do pedido, o *shipping method*, o histórico de um pedido, e o estado do pedido A coleção possui um total de 7550 *customers*.

A coleção *orders* agrega todas as informações relativas aos pedidos realizados, como a lista de livros por pedido, método de envio, histórico e estado atual. Esta abordagem facilita a consulta completa de um pedido num único documento, o que é altamente vantajoso em sistemas de *e-commerce*, onde a rastreabilidade e o histórico são essenciais.

Em resumo, a modelagem adotada permite uma estrutura mais eficiente, legível e adaptada às operações mais comuns do sistema, reduzindo a complexidade e melhorando o desempenho geral da base de dados.

## 4.2. Indexes

```
db.book.createIndex({ title: 1 });
db.customer.createIndex({ email: 1 });
db.cust_order.createIndex({ order_date: 1 });
db.address.createIndex({ country_id: 1 });
```

A conversão dos índices de SQL para MongoDB é relativamente direta, pois ambos os sistemas partilham o conceito fundamental de indexação para otimizar consultas. No entanto, existem nuances importantes na sintaxe e no comportamento com as quais devemos ter alguma atenção. No MongoDB, a sintaxe é mais concisa do que no SQL, mas igualmente poderosa. O método *createIndex()* é chamado directamente na collection, eliminando a necessidade de especificar um nome explícito para o índice. O valor 1 indica a ordem crescente, enquanto −1 indica a ordem decrescente. Esta notação numérica é mais compacta que a sintaxe SQL mas é equivalente a ASC e DESC. Para além disso, no MongoDB não necessitamos de operações de *commit* pois a criação de índices é uma operação atómica que é imediatamente persistida.

A funcionalidade dos índices mantém-se essencialmente a mesma: acelerar consultas que filtram ou ordenam por esses campos específicos. O índice em *book.title* vai optimizar consultas como *db.book.find({title: "Nome do Livro"})*, tal como o índice SQL optimizaria *SELECT * FROM book WHERE title = 'Nome do Livro'*. Da mesma forma, o índice em *customer.email* vai beneficiar pesquisas por email. O índice em *order_date* é particularmente útil para consultas temporais, como encontrar pedidos dentro de um intervalo de datas específico, e o índice em *country_id* na collection de endereços vai acelerar agregações geográficas ou filtragens por país. Estes casos de uso mantêm-se idênticos entre as duas tecnologias.

## 4.3. Procedure

Para a realizar um equivalente à procedure SQL em MongoDB foi necessária a criação do ficheiro Python procedures.py(ver na pasta /mongoDb do trabalho). A passagem de uma stored procedure SQL para um ficheiro Python reflecte uma diferença fundamental na filosofia e na arquitetura dos sistemas de base de dados relacionais e NoSQL. De facto, no MongoDB, a filosofia é diferente. Em MongoDB a lógica de negócio é tradicionalmente implementada na camada de aplicação. Isto acontece porque o MongoDB foi desenhado para ser mais simples e focado no armazenamento de documentos, delegando a lógica complexa para as aplicações cliente.

A lógica desenvolvida em Python mantém essencialmente a mesma funcionalidade da stored procedure original, mas adapta-se à estrutura desnormalizada do MongoDB. A função *update_order_status* começa por procurar o documento da encomenda usando o *order_id* fornecido. Esta operação é equivalente ao acesso directo à tabela *cust_order* da versão SQL, mas com a vantagem de que todos os dados relacionados já estão no mesmo documento. A recuperação do status anterior é feita através da manipulação do array *order_history* que está embebido no documento da encomenda. Em vez de fazer uma query com ORDER BY *status_date* DESC FETCH FIRST ROW ONLY como na versão Oracle, o código Python utiliza a função *sorted()* para ordenar o array por data em ordem decrescente e selecciona o primeiro elemento. Esta abordagem é possível porque o histórico já está armazenado localmente no documento, eliminando a necessidade de operações de JOIN que seriam necessárias num ambiente relacional. A verificação de mudança de *status* mantém a mesma lógica: compara o *status_id* actual com o novo status proposto. Se forem diferentes, procede com a actualização. Caso contrário, informa que o *status* já está no estado desejado. Esta validação é crucial para evitar entradas duplicadas desnecessárias no histórico. A inserção de um novo registo de status no MongoDB utiliza o operador *push* para adicionar um novo elemento ao array *order_history*. Este novo elemento contém todos os campos necessários: *history_id, status_id, status_value, e status_date*. O *history_id* é calculado

automaticamente como o tamanho atual do array mais um, simulando o comportamento da *sequence seq_orderhist.NEXTVAL* da versão SQL. A data é obtida através de *datetime.datetime.utcnow()*, que é equivalente ao *SYSDATE* do SQL. Para além disso, temos uma função auxiliar *get_status_value()*, que faz uma consulta à collection *order_status* para obter a descrição textual do status. Esta função simula parcialmente um JOIN que seria automático na versão SQL, mas que no MongoDB requer uma operação adicional devido à desnormalização dos dados. Esta é uma das poucas situações onde a versão MongoDB requer mais operações do que a versão SQL, apesar do impacto ser mínimo. Relativaemnte ao tratamento de erros, em vez de usar *DBMS_OUTPUT.PUT_LINE* para comunicar resultados, a função utiliza declarações *print()* padrão do Python. Além disso, inclui verificações adicionais como a existência do pedido antes de tentar actualizá-lo.

## 4.4. Views

```
db.createView(
  "book_with_authors",
  "books",
  [
    {
      $project: {
        _id: 0,
        book_id: "$_id",
        title: 1,
        authors: 1
      }
    },
    { $unwind: "$authors" },
    {
      $project: {
        book_id: 1,
        title: 1,
        author_name: "$authors.author_name"
      }
    }
  ]
)
```

Na primeira *View* do modelo relacional proposta pela equipa docente temos três tabelas separadas (book, book_author, author) que estão ligadas por chaves estrangeiras. Assim, são necessários os dois JOINs da *View* para obter os livros com os seus respetivos autores. No MongoDB, os autores já estão embebidos na coleção do livro como array authors. Assim, no *unwind* decompõe-se o array de autores, criando-se um documento separado para cada autor, o que é equivalente ao JOIN da versão sql. De seguida, no *project* selecciona-se apenas os campos necessários, mapeando o _id para book_id. Em MongoDB, é comum embutir dados relacionados (neste caso, os autores dentro de livros), evitando JOINs pesados.

```
db.createView(
  "orders_with_status",
  "orders",
  [
    {
      $set: {
```

```
        order_history_sorted: {
          $sortArray: {
            input: "$order_history",
            sortBy: { status_date: -1 }
          }
        }
      }
    },
    {
      $set: {
        latest_status: { $arrayElemAt: ["$order_history_sorted", 0] }
      }
    },
    {
      $project: {
        _id: 0,
        order_id: "$_id",
        order_date: 1,
        status_value: "$latest_status.status_value"
      }
    }
  ]
)
```

A segunda *view*, no modelo relacional, necessita de um JOIN entre *orders* e *order_history*, de um JOIN com *order_status* para obter o nome do status e ainda de uma subquery para encontrar o *status* mais recente de cada pedido. No MongoDB temos que a coleção *orders* tem um array *order_history* embutido. Assim, usamos o *sortArray* para ordenar o array *order_history* por data (equivalente ao MAX() + subquery). Utilizamos também o *arrayElemAt* de forma a extrair o primeiro elemento (o mais recente) do array ordenado. O histórico de *status* já está no documento do pedido com *status_value* desnormalizado. Mais uma vez, evitamos os JOINs e as subqueries complexas tornando a operação mais eficiente.

## 4.5. Triggers

### 4.5.1. validate_email

Para este decidimos criar um trigger que ao ser efetuado um insert de um documento na coleção *customers* verificaria através da expressão de regex se este email teria um formato válido.

```
exports = async function(changeEvent) {
  const customers =
context.services.get("BDNSQL").db("bookstore").collection("customers");

  const customerId = changeEvent.documentKey._id;

  // exp validação
  const emailRegex = /^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$/;

  if(changeEvent.fullDocument.email){
    if(!emailRegex.test(changeEvent.fullDocument.email)){
      console.log("Invalid email format");
      throw new Error('Invalid email format.');
```

```
    }
  }
}
```

No entanto apercebemo-nos que este trigger não iria ter o objetivo pretendido, pois em MongoDB não é possível utilizar funcionalidades como *Before Insert* e como tal este trigger não conseguiria prevenir a inserção do documento. Descobrimos então a funcionalidade de ***Schema Validation*** que o MongoDB oferece, com esta funcionalidade podemos verificar o *schema* e validar o formato do email, foi então desenvolvido o seguinte comando:

```
db.runCommand({
  collMod: "customers",
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "email"],
      properties: {
        email: {
          bsonType: "string",
          pattern: "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$",
          description: "Must be a valid email address"
        }
      }
    }
  },
  validationAction: "error", // previne a inserção se email não for válido
  validationLevel: "strict"
});
```

#### 4.5.2. insert_order_history

Para este trigger, criamos um *trigger* que escuta inserções (*insert*) na coleção orders. Quando uma nova ordem é inserida, o *trigger* cria o objeto de histórico inicial e realiza um *update* no pedido correspondente, adicionando esta entrada ao campo *order_history*.

```
exports = async function(changeEvent) {
  const orders =
context.services.get("BDNSQL").db("bookstore").collection("orders");

  const newOrder = changeEvent.fullDocument;
  const orderId = newOrder._id;

  try {
    const initOrderStatus = {
      status_id: 1,
      status_value: "Order Received",
      status_date: new Date()
    };

    await orders.updateOne(
      {"_id": orderId},
      {
        $push: {
          order_history: initOrderStatus
```

```
        }
      }
    )
    console.log(`inserted : ${orderId}`)
  }
  catch(error){
    console.log("Error adding order hist", error)
  }

};
```

#### 4.5.3. prevent_book_deletion

Este trigger tem o mesmo problema do trigger *validate_email*, sendo que apenas podes executar um trigger após o delete ser realizado, para resolver este problema, decicimos utilizar a funcionalidade ***Document Pre-Image*** presente no MongoDB atlas, este permite obtermos o documento que foi apagado antes do evento, e com isto caso o livro esteja presente em encomendas, podemos apenas voltar a inserir na base de dados.

```
exports = async function(changeEvent) {
  const orders =
context.services.get("BDNSQL").db("bookstore").collection("orders");
  const books =
context.services.get("BDNSQL").db("bookstore").collection("books");

  const deletedBookId = changeEvent.documentKey._id;

  try {
    const existsOrderWithBook = await orders.findOne({
      "order_lines.book_id": deletedBookId
    })

    if (existsOrderWithBook){
      console.log(` Book ${deletedBookId} was found in ongoing orders. Order ID:
${existsOrderWithBook._id}`);

      try{
        const deletedBook = changeEvent.fullDocumentBeforeChange;
        await books.insertOne(deletedBook);
        console.log("reinsert sucessfull")

      }
      catch(error){
        console.error("Error re-inserting book:", error);
      }

    }
  }
  catch(error){
    console.log("Error", error)
  }
};
```

## 4.6. Queries

### 4.6.1. Query 1

```
def query_1(db):
    pipeline = [
        {
            "$project": {
                "_id": 0,
                "book_id": "$_id",
                "Título do Livro": "$title",
                "Autores": "$authors.author_name",
                "Editora": "$publisher.publisher_name",
                "Idioma": "$language.language_name",
                "Páginas": "$num_pages",
                "Data de Publicação": "$publication_date"
            }
        },
        {
            "$sort": SON([("Título do Livro", 1)])
        }
    ]
    return list(db.books.aggregate(pipeline))
```

A transformação da *query* 1 em MongoDB consistiu na conversão de uma consulta SQL relacional com múltiplos JOINs e agregação para uma *aggregation pipeline* no MongoDB, considerando uma estrutura de documentos mais integrada. No SQL original, os dados de livros, autores, editoras e idiomas estão distribuídos em várias tabelas relacionadas, exigindo junções (JOIN) para combinar as informações. Além disso, é utilizada a função *LISTAGG* para concatenar os nomes dos autores de um mesmo livro. Na versão MongoDB, os dados estão estruturados de forma aninhada num único documento na coleção *books*, com campos incorporados como *authors, publisher, e language*. Assim, não são necessárias junções externas. A operação *project* seleciona os campos desejados e renomeia-os conforme o formato da consulta original. A ordenação pelo título é feita com a etapa *$sort*.

### 4.6.2. Query 2

```
def query_2(db):
    pipeline = [
        {"$unwind": "$order_lines"},
        {"$group": {
            "_id": "$customer_id",
            "Número de Encomendas": {"$sum": 1},
            "Valor Total Gasto": {"$sum": "$order_lines.price_at_order"}
        }},
        {"$lookup": {
            "from": "customers",
            "localField": "_id",
            "foreignField": "_id",
            "as": "cliente"
        }},
        {"$unwind": "$cliente"},
        {"$project": {
            "_id": 0,
            "Nome do Cliente": {
                "$concat": ["$cliente.first_name", " ", "$cliente.last_name"]
```

```
            },
            "Email": "$cliente.email",
            "Número de Encomendas": 1,
            "Valor Total Gasto": 1
        }},
        {"$sort": SON([("Número de Encomendas", -1), ("Valor Total Gasto", -1)])},
        {"$limit": 10}
    ]
    return list(db.orders.aggregate(pipeline))
```

A transformação da *query* 2 de SQL para MongoDB adaptou uma consulta relacional que identifica os 10 clientes com mais encomendas, para uma *aggregation pipeline* no MongoDB, refletindo as particularidades do modelo orientado a documentos. Na versão SQL, os dados estão distribuídos em três tabelas: *customer, cust_order e order_line*. São utilizados JOINs para relacionar as encomendas aos clientes e às suas linhas de pedido, além de agregações com COUNT e SUM, seguidas de ordenação e limitação dos resultados com FETCH FIRST 10 ROWS ONLY. Na versão MongoDB, os dados estão organizados na coleção orders, com as linhas de encomenda armazenadas em arrays (*order_lines*) dentro de cada documento de pedido. A etapa $*unwind* é usada para desestruturar esse array, permitindo o cálculo do número de encomendas e do valor total gasto por cliente usando $*group*. Como os dados do cliente estão numa coleção separada (*customers*), a operação $*lookup* simula um JOIN com base no campo _id, seguida por $*unwind* para acessar o documento do cliente individual. Por fim, são usados $*project* para formatar os campos, $*sort* para ordenar os resultados conforme os critérios da consulta original e $*limit* para restringir a saída aos 10 principais clientes.

### 4.6.3. Query 3

```
def query_3(db):
    pipeline = [
        {"$unwind": "$order_lines"},
        {
            "$lookup": {
                "from": "books",
                "localField": "order_lines.book_id",
                "foreignField": "_id",
                "as": "book"
            }
        },
        {"$unwind": "$book"},
        {
            "$match": {
                "$and": [
                    {"book.authors": {"$exists": True}},
                    {"book.authors": {"$ne": []}},
                    {"book.authors": {"$not": {"$size": 0}}}
                ]
            }
        },
        {
            "$group": {
                "_id": "$book._id",
                "titulo": {"$first": "$book.title"},
                "authors_raw": {"$first": "$book.authors"},
```

```
                "quantidade_vendida": {"$sum": 1},
                "preco_medio": {"$avg": "$order_lines.price_at_order"},
                "clientes_unicos": {"$addToSet": "$customer_id"}
            }
        },
        {
            "$addFields": {
                "autores": {
                    "$reduce": {
                        "input": {
                            "$sortArray": {
                                "input": "$authors_raw",
                                "sortBy": {"author_name": 1}
                            }
                        },
                        "initialValue": "",
                        "in": {
                            "$cond": {
                                "if": {"$eq": ["$$value", ""]},
                                "then": {"$ifNull": ["$$this.author_name", ""]},
                                "else": {"$concat": ["$$value", ", ", {"$ifNull":
["$$this.author_name", ""]}]}
                            }
                        }
                    }
                },
                "clientes_unicos_count": {"$size": "$clientes_unicos"}
            }
        },
        {
            "$project": {
                "_id": 0,
                "Título do Livro": "$titulo",
                "Autores": "$autores",
                "Quantidade Total Vendida": "$quantidade_vendida",
                "Preço Médio": {"$round": ["$preco_medio", 2]},
                "Clientes Únicos": "$clientes_unicos_count"
            }
        },
        {"$sort": {"Quantidade Total Vendida": -1}},
        {"$limit": 15}
    ]

    return list(db.orders.aggregate(pipeline))
```

A transformação da consulta SQL para MongoDB envolveu reestruturar a lógica relacional da consulta original em uma pipeline de agregação, que reflete a forma como os dados são armazenados em documentos. No SQL, usamos JOINs entre as tabelas *book, order_line, cust_order e author* para combinar informações dos livros vendidos, autores, preços e clientes. No MongoDB, como os dados estão em documentos aninhados, usamos etapas de $*lookup* para simular esses JOINs, como o caso da relação entre *orders* e *books*. A operação de $unwind foi usada para expandir os arrays de linhas de pedido (*order_lines*), permitindo o cálculo da quantidade vendida e média de preços por item. O agrupamento por livro foi feito com $group, acumulando a quantidade total de vendas ($sum), a média de preços ($avg) e os clientes únicos ($addToSet). Para listar os autores em ordem alfabética, usamos $reduce

junto com $sortArray, simulando o LISTAGG do SQL Oracle. Por fim, usamos $project para formatar a saída final e $sort e $limit para ordenar e limitar os resultados, assim como no ORDER BY ... FETCH FIRST da versão SQL.

#### 4.6.4. Query 4

```
def query_4(db):
    pipeline = [
        {"$match": {"destination_address.country.country_id": {"$exists": True}}},

        {"$unwind": "$order_lines"},

        {"$group": {
            "_id": {
                "country_id": "$destination_address.country.country_id",
                "country_name": "$destination_address.country.country_name"
            },
            "clientes_unicos": {"$addToSet": "$customer_id"},
            "total_encomendas": {"$sum": 1},  # Cada linha do pedido conta como
uma encomenda?
            "receita_total": {"$sum": "$order_lines.price_at_order"}
        }},

        {"$project": {
            "País": "$_id.country_name",
            "Clientes Únicos": {"$size": "$clientes_unicos"},
            "Total de Encomendas": "$total_encomendas",
            "Receita Total": "$receita_total",
            "Valor Médio por Encomenda": {
                "$cond": [
                    {"$gt": ["$total_encomendas", 0]},
                    {"$divide": ["$receita_total", "$total_encomendas"]},
                    0
                ]
            }
        }},

        {"$sort": {"Receita Total": -1}}
    ]

    return list(db.orders.aggregate(pipeline))
```

A transformação da *query* 4 de SQL para MongoDB converte uma análise relacional de encomendas por país em uma *aggregation pipeline*, adaptada à estrutura orientada a documentos do MongoDB. Na consulta SQL, as informações estão distribuídas por várias tabelas normalizadas (*country, address, customer_address, customer, cust_order, order_line*). São realizadas diversas junções para ligar clientes aos seus países por meio de endereços, agregando os dados por país para calcular:

- número de clientes únicos (COUNT(DISTINCT ...)),
- total de encomendas,
- receita total (SUM),
- valor médio por encomenda (AVG)

No MongoDB, os dados dos pedidos estão na coleção *orders*, e que cada pedido contém o endereço de destino com informações embutidas do país (*destination_address.country*). A pipeline executa os seguintes passos:

- $match: Garante que cada pedido analisado possua um país no endereço de destino.
- $unwind: Desestrutura o array *order_lines*, permitindo o cálculo por linha de pedido.
- $group: Agrupa os dados por país, acumulando: clientes únicos com $addToSet (simulando DISTINCT), total de encomendas (contagem de linhas, assumindo que cada linha representa uma encomenda) e a receita total com $sum.
- $project: Formata os dados finais, incluindo o cálculo do valor médio por encomenda com $divide e $cond, que simula a função AVG.
- $sort: Ordena os resultados pela receita total em ordem decrescente.

Essa transformação reflete a complexidade de mapear uma estrutura altamente relacional para um modelo de documentos, demonstrando como MongoDB permite análises sofisticadas mesmo sem o uso explícito de JOINs, graças à agregação e modelagem embutida de dados.

#### 4.6.5. Query 5

```
def query_5(db):
    pipeline = [
        {"$unwind": "$order_history"},

        {"$sort": {"_id": 1, "order_history.status_date": -1}},

        {"$group": {
            "_id": "$_id",  # pedido_id
            "latest_status_id": {"$first": "$order_history.status_id"},
            "latest_status_value": {"$first": "$order_history.status_value"}
        }},

        {"$group": {
            "_id": "$latest_status_value",
            "num_encomendas": {"$sum": 1}
        }},

        {"$group": {
            "_id": None,
            "total_encomendas": {"$sum": "$num_encomendas"},
            "estados": {
                "$push": {
                    "Estado da Encomenda": "$_id",
                    "Número de Encomendas": "$num_encomendas"
                }
            }
        }},

        {"$unwind": "$estados"},

        {"$project": {
            "Estado da Encomenda": "$estados.Estado da Encomenda",
            "Número de Encomendas": "$estados.Número de Encomendas",
            "Percentagem": {
                "$round": [
                    {
                        "$multiply": [
                            {"$divide": ["$estados.Número de Encomendas",
"$total_encomendas"]},
```

```
                            100
                        ]
                    },
                    2
                ]
            }
        }},

        {"$sort": {"Número de Encomendas": -1}}
    ]

    return list(db.orders.aggregate(pipeline))
```

A transformação da *query* 5 de SQL para MongoDB traduz uma análise de distribuição de encomendas por estado (*status*), com cálculo de percentuais, utilizando a *aggregation pipeline* para simular construções SQL como subqueries, `JOINs`, e `WINDOW FUNCTIONS`.

Em MongoDB, essa lógica é reestruturada da seguinte forma:

- $unwind: Desestrutura o *array* `order_history` de cada pedido para tratar os status individualmente.
- $sort: Ordena os históricos por `order_id` e `status_date` em ordem decrescente, garantindo que o status mais recente venha primeiro.
- $group: Agrupa por pedido (`_id`) e seleciona o primeiro status (mais recente), simulando a subquery SQL.
- Segundo $group: Agrupa por `status_value` para contar o número de encomendas por estado.
- Terceiro $group: Soma o total de encomendas e agrupa os dados em um array `estados` para facilitar o cálculo de percentagens.
- $unwind + $project: Reestrutura os dados do array e calcula a percentagem de cada estado sobre o total de encomendas, com arredondamento.
- $sort: Ordena por número de encomendas em ordem decrescente.

#### 4.6.6. Query 6

```
def query_6(db):
    pipeline = [
        {"$unwind": "$authors"},
        {
            "$group": {
                "_id": "$authors.author_id",
                "author_name": {"$first": "$authors.author_name"},
                "number_of_books": {"$sum": 1},
                "publishers": {"$addToSet": "$publisher.publisher_id"},
                "first_publication": {"$min": "$publication_date"},
                "last_publication": {"$max": "$publication_date"},
                "books_ids": {"$addToSet": "$_id"}
            }
        },
        {
            "$lookup": {
                "from": "orders",
                "let": {"books_ids": "$books_ids"},
                "pipeline": [
                    {"$unwind": "$order_lines"},
                    {
```

```
                        "$match": {
                            "$expr": {"$in": ["$order_lines.book_id", "$
$books_ids"]}
                        }
                    },
                    {
                        "$group": {
                            "_id": None,
                            "total_sold": {"$sum": 1}
                        }
                    }
                ],
                "as": "order_matches"
            }
        },
        {
            "$addFields": {
                "total_books_sold": {
                    "$ifNull": [{"$arrayElemAt": ["$order_matches.total_sold",
0]}, 0]
                }
            }
        },
        {
            "$addFields": {
                "different_publishers": {"$size": "$publishers"}
            }
        },
        {
            "$project": {
                "_id": 0,
                "Nome do Autor": "$author_name",
                "Número de Livros": "$number_of_books",
                "Editoras Diferentes": "$different_publishers",
                "Primeira Publicação": "$first_publication",
                "Última Publicação": "$last_publication",
                "Total de Livros Vendidos": "$total_books_sold"
            }
        },
        {
            "$sort": {"Número de Livros": -1}
        }
    ]
    return list(db["books"].aggregate(pipeline))
```

A transformação da *query* 6 para MongoDB consistiu em adaptar a lógica relacional para um modelo orientado a documentos, utilizando a pipeline de agregação do MongoDB.

No SQL, a consulta utiliza múltiplos `JOINs` entre as tabelas `author`, `book_author`, `book`, `publisher` e `order_line` para identificar autores, contar livros distintos, editoras associadas, datas de publicação e total de vendas.

No MongoDB, partimos da coleção `books`, onde os autores estão armazenados como um array embutido. Usamos `$unwind` para expandir esse *array*, processando cada autor individualmente. Com `$group`,

agrupamos os dados por autor, contando os livros (`$sum`), acumulando editoras únicas (`$addToSet`) e obtendo as datas da primeira e última publicação (`$min` e `$max`).

Para contar as vendas, utilizamos `$lookup` com `let` e uma subpipeline, simulando um `LEFT JOIN` com a coleção `orders`, filtrando apenas os pedidos que contêm livros do autor. O total de vendas é calculado com `$sum` sobre as linhas de pedido correspondentes. Em casos sem vendas, usamos `$ifNull` para garantir o valor 0.

Após isso, calculamos o número de editoras distintas com `$size` e projetamos os campos finais com `$project`, formatando a saída da mesma forma que a consulta SQL. Por fim, a ordenação por número de livros é feita com `$sort` em ordem decrescente.

Essa abordagem mostra como o MongoDB pode lidar com operações analíticas complexas, mesmo sem o modelo relacional tradicional, aproveitando seu modelo de documentos flexível e expressivo.

#### 4.6.7. Query 7

```
def query_7(db):
    pipeline = [
        {"$unwind": "$order_lines"},

        {
            "$group": {
                "_id": {
                    "year": {"$year": "$order_date"},
                    "month": {"$month": "$order_date"},
                    "monthName": {"$dateToString": {"format": "%B", "date":
"$order_date"}}
                },
                "encomendas": {"$addToSet": "$_id"},         # pedidos únicos no
mês
                "livros_vendidos": {"$sum": 1},              # cada linha = 1
livro vendido
                "receita_total": {"$sum": "$order_lines.price_at_order"}  # soma
dos preços
            }
        },

        {
            "$project": {
                "_id": 0,
                "Ano": "$_id.year",
                "Mês": "$_id.month",
                "Nome do Mês": "$_id.monthName",
                "Encomendas": {"$size": "$encomendas"},
                "Livros Vendidos": "$livros_vendidos",
                "Receita Total": "$receita_total"
            }
        },

        {
            "$sort": {"Ano": -1, "Mês": -1}
        }
    ]
```

```
    return list(db.orders.aggregate(pipeline))
```

A transformação da *query* 7 SQL para MongoDB adapta uma análise de vendas mensais, por ano, mês e nome do mês, para o modelo de agregação do MongoDB, reproduzindo funções como EXTRACT, TO_CHAR, COUNT(DISTINCT), SUM e GROUP BY.

- $unwind: desestrutura order_lines para processar cada livro vendido separadamente.
- $addFields: extrai o ano ($year), o mês numérico ($month) e o nome do mês ($dateToString com locale: "pt" para português).
- $group: agrupa por ano, mês e nome do mês. Para simular COUNT(DISTINCT order_id), utiliza $addToSet para reunir IDs únicos de pedidos.
- $project: calcula o número de encomendas com $size, além de expor os demais campos agregados.
- $sort: ordena os resultados do período mais recente para o mais antigo.

#### 4.6.8. Query 8

```
def query_8(db):
    pipeline = [
        {"$unwind": "$addresses"},
        {
            "$addFields": {
                "formatted_address": {
                    "$concat": [
                        {"$toString": "$addresses.street_number"}, " ",
                        "$addresses.street_name", ", ",
                        "$addresses.city", ", ",
                        "$addresses.country.country_name", " (",
                        "$addresses.status.status_value", ")"
                    ]
                }
            }
        },
        {
            "$group": {
                "_id": {
                    "customer_id": "$_id",
                    "first_name": "$first_name",
                    "last_name": "$last_name",
                    "email": "$email"
                },
                "addresses": {
                    "$push": {
                        "address_id": "$addresses.address_id",
                        "formatted_address": "$formatted_address"
                    }
                },
                "address_count": {"$sum": 1}
            }
        },

        {"$match": {"address_count": {"$gt": 1}}},

        {
```

```
            "$project": {
                "_id": 0,
                "Nome do Cliente": {
                    "$concat": ["$_id.first_name", " ", "$_id.last_name"]
                },
                "Email": "$_id.email",
                "Número de Endereços": "$address_count",
                "Todos os Endereços": {
                    "$reduce": {
                        "input": {
                            "$map": {
                                "input": {
                                    "$sortArray": {
                                        "input": "$addresses",
                                        "sortBy": {"address_id": 1}
                                    }
                                },
                                "as": "addr",
                                "in": "$$addr.formatted_address"
                            }
                        },
                        "initialValue": "",
                        "in": {
                            "$cond": [
                                {"$eq": ["$$value", ""]},
                                "$$this",
                                {"$concat": ["$$value", "; ", "$$this"]}
                            ]
                        }
                    }
                }
            }
        },

        {"$sort": {"Número de Endereços": -1}}
    ]

    return list(db.customers.aggregate(pipeline))
```

A *query* 8 junta várias tabelas (*customer, customer_address, address, country, address_status*) para listar clientes que têm mais de um endereço, concatenando os endereços formatados em uma única string, ordenada pelo número de endereços.

No MongoDB, os dados dos clientes e seus endereços estão provavelmente embutidos como arrays dentro do documento do cliente (ou via referência populada). A pipeline executa:

- $unwind para "desdobrar" o array de endereços, permitindo manipular cada endereço individualmente.
- $addFields cria uma string formatada para cada endereço, concatenando número, rua, cidade, país e status.
- $group reagrupa os dados por cliente, juntando novamente os endereços em um array e contando quantos endereços ele tem.
- $match filtra para clientes com mais de um endereço.
- $project monta a saída final, concatenando os endereços formatados em uma única string separada por "; ", usando $reduce para replicar o LISTAGG do SQL.

- $sort ordena os clientes pelo número de endereços em ordem decrescente.

Assim, a estrutura relacional do SQL é adaptada para manipulação de arrays e operações agregadas nativas do MongoDB.

#### 4.6.9. Query 9

```
def query_9(db):
    pipeline = [
        {
            "$group": {
                "_id": {
                    "language_id": "$language.language_id",
                    "language_name": "$language.language_name",
                    "publisher_id": "$publisher.publisher_id",
                    "publisher_name": "$publisher.publisher_name"
                },
                "numero_de_livros": {"$sum": 1},
                "media_paginas": {"$avg": "$num_pages"},
                "livro_mais_antigo": {"$min": "$publication_date"},
                "livro_mais_recente": {"$max": "$publication_date"}
            }
        },
        {
            "$project": {
                "_id": 0,
                "Idioma": "$_id.language_name",
                "Editora": "$_id.publisher_name",
                "Número de Livros": "$numero_de_livros",
                "Média de Páginas": {"$round": ["$media_paginas", 2]},  #
arredonda para 2 casas decimais
                "Livro Mais Antigo": "$livro_mais_antigo",
                "Livro Mais Recente": "$livro_mais_recente"
            }
        },

        {
            "$sort": {
                "Idioma": 1,
                "Número de Livros": -1
            }
        }
    ]
    return list(db.books.aggregate(pipeline))
```

A *query* 9 agrega livros por idioma e editora, retornando o número de livros, a média de páginas, a data do livro mais antigo e mais recente, ordenando os resultados.

Em MongoDB, os dados de idioma e editora estão embutidos ou referenciados no documento do livro. A pipeline executa:

- $group agrupa os livros por idioma e editora (usando os campos embutidos language.language_id e publisher.publisher_id).
- Para cada grupo, calcula o número de livros ($sum), média de páginas ($avg), data mínima ($min) e máxima ($max).
- $project organiza a saída, arredondando a média de páginas para 2 casas decimais, similar ao ROUND do SQL.

- $sort ordena pelo nome do idioma (ascendente) e número de livros (descendente), espelhando o ORDER BY da consulta SQL.

#### 4.6.10. Query 10

```
def query_10(db):
    pipeline = [
        {"$unwind": "$order_lines"},

        {
            "$group": {
                "_id": {
                    "method_id": "$shipping_method.method_id",
                    "method_name": "$shipping_method.method_name"
                },
                "numero_de_utilizacoes": {"$addToSet": "$_id"},  # conjunto de
order_ids únicos
                "total_linhas": {"$sum": 1},                     # total de linhas
de encomenda (livros vendidos)
                "receita_total": {"$sum": "$order_lines.price_at_order"}
            }
        },

        {
            "$addFields": {
                "numero_de_encomendas": {"$size": "$numero_de_utilizacoes"},
                "valor_medio_encomenda": {
                    "$divide": ["$receita_total", {"$size":
"$numero_de_utilizacoes"}]
                }
            }
        },

        {
            "$group": {
                "_id": None,
                "metodos": {
                    "$push": {
                        "method_name": "$_id.method_name",
                        "numero_de_utilizacoes": {"$size":
"$numero_de_utilizacoes"},
                        "valor_medio_encomenda": "$valor_medio_encomenda",
                        "receita_total": "$receita_total"
                    }
                },
                "total_utilizacoes": {"$sum": {"$size": "$numero_de_utilizacoes"}}
            }
        },

        {
            "$unwind": "$metodos"
        },
        {
            "$project": {
                "_id": 0,
                "Método de Envio": "$metodos.method_name",
```

```
                "Número de Utilizações": "$metodos.numero_de_utilizacoes",
                "Percentagem de Uso": {
                    "$round": [
                        {
                            "$multiply": [
                                {"$divide": ["$metodos.numero_de_utilizacoes",
"$total_utilizacoes"]},
                                100
                            ]
                        },
                        2
                    ]
                },
                "Valor Médio das Encomendas": {
                    "$round": ["$metodos.valor_medio_encomenda", 2]
                },
                "Receita Total": {
                    "$round": ["$metodos.receita_total", 2]
                }
            }
        },

        {
            "$sort": {"Número de Utilizações": -1}
        }
    ]

    return list(db.orders.aggregate(pipeline))
```

A transformação da *query* 10 de SQL para MongoDB reimplementa uma análise de uso de métodos de envio, calculando quantidade de utilizações, percentagem relativa, receita total e valor médio das encomendas por método. Isso é feito utilizando a *aggregation pipeline* para simular funcionalidades como JOINs, COUNT, AVG, SUM e WINDOW FUNCTIONS.

- $unwind: desestrutura o *array order_lines* de cada encomenda, permitindo calcular a receita linha a linha.
- Primeiro $group: agrupa por método de envio, usando $addToSet para coletar encomendas únicas (_id do pedido), acumulando a receita total e o número total de livros vendidos.
- $addFields: calcula o número de encomendas (tamanho do conjunto) e o valor médio por encomenda (divisão da receita pelo número de encomendas).
- Segundo $group: agrega todos os métodos num *array* e calcula o total geral de utilizações, necessário para computar a percentagem de uso.
- $unwind + $project: itera sobre os métodos e calcula a percentagem de uso relativa ao total, além de aplicar arredondamentos nos valores financeiros.
- $sort: ordena os métodos de envio pelo número de utilizações em ordem decrescente.

#### 4.6.11. Query 11

```
def query_11(db):
    pipeline = [
        {
            "$lookup": {
                "from": "orders",
```

```
                "let": {"book_id": "$_id"},
                "pipeline": [
                    {"$unwind": "$order_lines"},
                    {"$match": {"$expr": {"$eq": ["$order_lines.book_id", "$
$book_id"]}}}
                ],
                "as": "orders_with_book"
            }
        },
        {
            "$match": {
                "orders_with_book": {"$size": 0}
            }
        },
        {
            "$addFields": {
                "dias_desde_publicacao": {
                    "$divide": [
                        {"$subtract": [{"$toDate": "$$NOW"},
"$publication_date"]},
                        86400000  # milliseconds in a day
                    ]
                }
            }
        },
        {
            "$addFields": {
                "dias_desde_publicacao": {"$toInt": "$dias_desde_publicacao"}
            }
        },
        {
            "$addFields": {
                "autores": {
                    "$reduce": {
                        "input": "$authors",
                        "initialValue": "",
                        "in": {
                            "$cond": {
                                "if": {"$eq": ["$$value", ""]},
                                "then": "$$this.author_name",
                                "else": {"$concat": ["$$value", ", ", "$
$this.author_name"]}
                            }
                        }
                    }
                }
            }
        },
        {
            "$project": {
                "_id": 0,
                "titulo_do_livro": "$title",
                "autores": 1,
                "editora": "$publisher.publisher_name",
                "idioma": "$language.language_name",
```

```
                "data_de_publicacao": "$publication_date",
                "dias_desde_publicacao": 1
            }
        },
        {
            "$sort": {"data_de_publicacao": -1}
        }
    ]

    return list(db.books.aggregate(pipeline))
```

A consulta SQL original busca listar livros que nunca foram vendidos, ou seja, que não aparecem em nenhuma linha de pedido (*order_line*). Para isso, ela faz várias junções (JOIN) entre as tabelas de livros, autores, editoras, idiomas e pedidos, e filtra os livros sem ocorrência em pedidos (LEFT JOIN com condição IS NULL). Também agrega os nomes dos autores usando LISTAGG e calcula o número de dias desde a publicação. Na versão MongoDB, a lógica foi adaptada para o modelo orientado a documentos:

- Utilizamos um $lookup com pipeline para verificar se o livro aparece em algum pedido, buscando nas *order_lines* das *orders* onde o *book_id* coincida.
- Filtramos com $match para selecionar apenas os livros que não aparecem em nenhum pedido (*orders_with_book* vazio).
- Calculamos o número de dias desde a publicação subtraindo a data atual ($$NOW) pela data de publicação do livro, convertendo de milissegundos para dias e truncando para inteiro.
- Para agregar os autores, usamos $reduce para concatenar os nomes, simulando o LISTAGG do SQL.

Finalmente, projetamos os campos desejados e ordenamos os livros pela data de publicação em ordem decrescente.

#### 4.6.12. Query 12

```
def query_12(db):
    pipeline = [
        {
            "$lookup": {
                "from": "orders",
                "localField": "_id",
                "foreignField": "customer_id",
                "as": "orders"
            }
        },
        {
            "$unwind": {
                "path": "$orders",
                "preserveNullAndEmptyArrays": False
            }
        },
        {
            "$unwind": {
                "path": "$orders.order_lines",
                "preserveNullAndEmptyArrays": False
            }
        },
        {
            "$group": {
                "_id": "$_id",
```

```
                "customer_name": {
                    "$first": {
                        "$concat": ["$first_name", " ", "$last_name"]
                    }
                },
                "email": {"$first": "$email"},
                "total_orders": {"$addToSet": "$orders._id"},  # para contar
ordens únicas
                "total_spent": {"$sum": "$orders.order_lines.price_at_order"},
                "first_order": {"$min": "$orders.order_date"},
                "last_order": {"$max": "$orders.order_date"}
            }
        },
        {
            "$addFields": {
                "total_orders": {"$size": "$total_orders"},
                "customer_lifespan_days": {
                    "$trunc": {
                        "$divide": [
                            {"$subtract": ["$last_order", "$first_order"]},
                            1000 * 60 * 60 * 24  # converter ms para dias
                        ]
                    }
                }
            }
        },
        {
            "$addFields": {
                "categoria": {
                    "$switch": {
                        "branches": [
                            {
                                "case": {
                                    "$and": [
                                        {"$gte": ["$total_orders", 10]},
                                        {"$gte": ["$total_spent", 500]}
                                    ]
                                },
                                "then": "VIP"
                            },
                            {
                                "case": {
                                    "$or": [
                                        {"$gte": ["$total_orders", 5]},
                                        {"$gte": ["$total_spent", 200]}
                                    ]
                                },
                                "then": "Fiel"
                            },
                            {
                                "case": {"$gte": ["$total_orders", 2]},
                                "then": "Regular"
                            }
                        ],
                        "default": "Novo"
```

```
                    }
                }
            }
        },
        {
            "$project": {
                "_id": 0,
                "Nome do Cliente": "$customer_name",
                "Email": "$email",
                "Total de Encomendas": "$total_orders",
                "Total Gasto": "$total_spent",
                "Categoria de Cliente": "$categoria",
                "Primeira Encomenda": "$first_order",
                "Última Encomenda": "$last_order",
                "Dias Como Cliente": "$customer_lifespan_days"
            }
        },
        {
            "$sort": {
                "Total Gasto": -1
            }
        }
    ]
    return list(db.customers.aggregate(pipeline))
```

A transformação da *query* 12 envolveu converter a lógica de análise de clientes numa *pipeline* de agregação partindo da coleção *customers*. No MongoDB, simulamos a CTE com múltiplas etapas:

- $lookup com *orders*: Une os dados de clientes com suas encomendas através da correspondência entre customer._id e orders.customer_id.
- $unwind: Desaninha os arrays de encomendas (*orders*) e suas linhas (*order_lines*), permitindo análise por item comprado.
- $group: Agrupa os dados novamente por cliente, calculando:
  - Nome completo e email;
  - Conjunto de encomendas únicas ($addToSet);
  - Soma dos valores de compra;
  - Data da primeira e última encomenda.
- $addFields: Conta o número total de encomendas. Calcula o tempo como cliente (*customer_lifespan_days*) convertendo a diferença entre datas de milissegundos para dias. Define a categoria do cliente com um $switch, replicando a lógica do CASE SQL (VIP, Fiel, Regular, Novo).
- $project: Formata a saída para refletir os nomes de colunas esperados na consulta original.
- $sort: Ordena os clientes pelo total gasto, do maior para o menor.

#### 4.6.13. Query 13

```
def query_13(db):
    pipeline = [
        {
            "$unwind": {
                "path": "$authors",
                "preserveNullAndEmptyArrays": True
            }
        },
        {
```

```
            "$group": {
                "_id": "$publisher.publisher_id",
                "publisher_name": { "$first": "$publisher.publisher_name" },
                "books_in_catalog": { "$addToSet": "$_id" },
                "unique_authors": { "$addToSet": "$authors.author_id" },
                "published_languages": { "$addToSet": "$language.language_id" }
            }
        },
        {
            "$lookup": {
                "from": "orders",
                "let": { "publisherBookIds": "$books_in_catalog" },
                "pipeline": [
                    { "$unwind": "$order_lines" }, #
                    {
                        "$match": {
                            "$expr": {
                                "$in": ["$order_lines.book_id", "$
$publisherBookIds"]
                            }
                        }
                    },
                    {
                        "$group": {
                            "_id": None,
                            "total_sold_lines": { "$sum": 1 },
                            "total_revenue": { "$sum":
"$order_lines.price_at_order" }
                        }
                    }
                ],
                "as": "sales_data"
            }
        },
        {
            "$unwind": {
                "path": "$sales_data",
                "preserveNullAndEmptyArrays": True
            }
        },
        {
            "$project": {
                "_id": 0, #
                "Editora": "$publisher_name",
                "Livros no Catálogo": { "$size": "$books_in_catalog" },
                "Autores Únicos": { "$size": { "$ifNull": ["$unique_authors",
[]] } }, #
                "Total de Linhas Vendidas": { "$ifNull":
["$sales_data.total_sold_lines", 0] },
                "Receita Total": { "$ifNull": ["$sales_data.total_revenue", 0] },
                "Média de Vendas por Livro": {
                    "$cond": [
                        { "$gt": [{ "$size": "$books_in_catalog" }, 0] },
                        { "$round": [{ "$divide": [{ "$ifNull":
["$sales_data.total_sold_lines", 0] }, { "$size": "$books_in_catalog" }] }, 2] },
```

```
                    ]
                },
                "Idiomas Publicados": { "$size": "$published_languages" }
            }
        },
        {
            "$sort": {
                "Receita Total": -1
            }
        }
    ]
    return list(db.books.aggregate(pipeline))
```

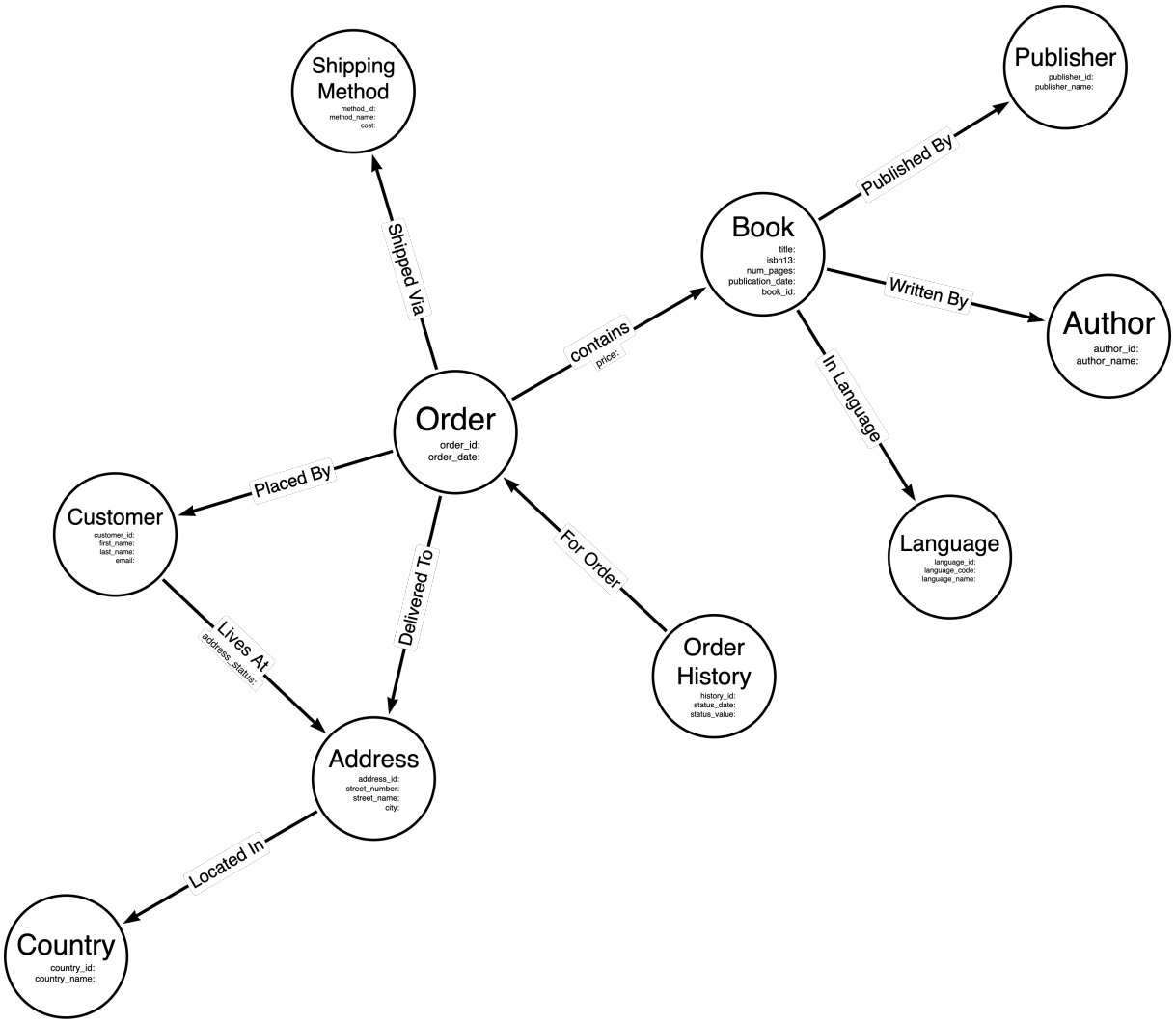
A transformação da *query 13* converte uma análise relacional com múltiplas subconsultas numa *pipeline* de agregação sobre documentos da coleção *books*. No SQL, a consulta envolve a tabela *publisher* com várias subconsultas correlacionadas para contar livros, autores, idiomas e vendas relacionadas a cada editora. O objetivo é gerar estatísticas completas por editora, com agregações cruzando várias entidades. No MongoDB, a pipeline realiza as seguintes etapas:

- $unwind em authors: Expande o array de autores de cada livro para permitir a contagem correta de autores únicos por editora. A opção preserveNullAndEmptyArrays: *True* garante que livros sem autores não sejam descartados.
- $group por editora:
  - Agrupa os livros por publisher.publisher_id.
  - Usa $addToSet para coletar:
    - IDs únicos dos livros publicados (books_in_catalog);
    - IDs únicos dos autores (unique_authors);
    - IDs dos idiomas (published_languages).
- $lookup com a coleção orders: Faz um *join* por subpipeline (*let + pipeline*) entre os livros da editora e as linhas de pedidos (*order_lines*) que os contêm. Agrupa estas linhas para calcular:
  - Total de linhas vendidas;
  - Receita total associada aos livros da editora.
- $unwind do resultado do lookup: Desenrola o array *sales_data* com *preserveNullAndEmptyArrays: True* para garantir que editoras sem vendas também apareçam no relatório.
- $project final: Calcula diretamente os campos exigidos:
  - Número de livros ($size de *books_in_catalog*);
  - Autores únicos e idiomas publicados;
  - Total de linhas vendidas e receita (com $ifNull para lidar com editoras sem vendas);
  - Média de vendas por livro, usando $divide com verificação para evitar divisão por zero.
- $sort: Ordena o relatório por Receita Total em ordem decrescente, como na cláusula ORDER BY COALESCE(SUM(...)) DESC do SQL.

# 5. Neo4j

## 5.1. Estrutura dos dados

Antes de proceder a conversão de Oracle para Neo4J, precisamos de identificar quais tabelas serão realmente necessárias de representar como labels de nodos e quais poderão ser representadas por relações. Depois de ser feita uma análise, decidimos usar a seguinte estrutura:




## 5.2. Nodos

### 5.2.1. Nodo “Book”

O nodo “Book” é o equivalente à tabela “book” na base de dados Oracle. Mantém basicamente a mesma estrutura, perdendo só as chaves estrangeiras que possuía (language/publisher_id), pois estas irão ser transformadas em relações.

### 5.2.2. Nodo “Publisher”

O nodo “Publisher” é o equivalente á tabela “publisher” na base de dados Oracle, mantendo exatamente os mesmos atributos.

### 5.2.3. Nodo “Author”

O nodo “Author” é o equivalente à tabela “author”, contendo os mesmos dados que esta.

#### 5.2.4. Nodo "Language"

O nodo "Language" é equivalente à tabela "book_language", contendo outra vez exatamente a mesma informação dessa tabela.

#### 5.2.5. Nodo "Order"

O nodo "Order" é equivalente à tabela "cust_order", contendo o mesmo atributo e removendo as chaves estrangeiras presentes, pela mesma razão do nodo "Book" (irão ser transformadas em relações).

#### 5.2.6. Nodo "Shipping Method"

O nodo "Shipping Method" é equivalente à tabela "shipping_method", contendo a mesma informação das entradas nessa tabela.

#### 5.2.7. Nodo "OrderHistory"

O nodo "OrderHistory" é equivalente às tabelas "order_history" e "order_status". Decidimos concatenar a informação da tabela "order_status" (ou seja, o status equivalente ao status_id) para reduzir a quantidade de nodos diferente na nossa base de dados, apesar de isto resultarem em dados redundantes. De resto, mantém a mesma informação da tabela original (a data da atualização de status).

#### 5.2.8. Nodo "Order"

O nodo "Order" é o equivalente à tabela "cust_order", tendo só o atributo "order_date" pois o resto são chaves estrangeiras.

#### 5.2.9. Nodo "Customer"

O nodo "Customer" é o equivalente à tabela "customer", contendo exatamente a mesma informação da tabela original em Oracle.

#### 5.2.10. Nodo "Address"

O nodo "Address" é o equivalente à tabela "address", contendo a mesma informação da tabela com a excepção da chave estrangeira "country_id", que irá ser representada por uma relação.

#### 5.2.11. Nodo "Country"

O nodo "Country" é o equivalente à tabela "country", contendo toda a informação presente na tabela original.

### 5.3. Relações

#### 5.3.1. Relações "Published By" e "In Language"

As relações "Published By" e "In Language" substituem as chaves estrangeiras presentes na tabela original a partir da qual foi criado o nodo "Book". Esta modelação será a parte mais importante da migração para Neo4j, pois é necessário representar as relações entre as tabelas Oracle (ou seja, as chaves estrangeiras) em relações entre nodos para usufruir das vantagens que uma base de dados de grafos possui.

#### 5.3.2. Relação "Written By"

Ao contrário das relações anteriores, a relação "Written By" substitui uma tabela da base de dados original ("book_author"). Esta tabela existe na base de dados original para permitir a possibilidade de um único livro ter vários autores. Se esta tabela não existisse, a relação teria de ser representada por uma chave estrangeira e portanto só poderia referenciar um único autor. Como em Neo4J utilizamos relações que não têm restrições em termos de quantidade, podemos transformar a tabela original numa relação simples sem problemas.

#### 5.3.3. Relação "Contains"

A relação "Contains" é similar à relação descrita anteriormente no sentido em que ambas traduzem uma tabela ("order_line", neste caso) para uma relação. Neste caso, a tabela original também tinha a coluna "price", que é transformada num atributo da relação quando convertida para Neo4J.

#### 5.3.4. Relações "Shipped Via", "Placed By" e "Delivered To"

Estas três relações são representadas por chaves estrangeiras da tabela "cust_order" na base de dados original, portanto o processo de migração é simples, associando o nodo "Order" ao nodo "Shipping-Method", "Customer" e"Address" correspondentes, respetivamente.

#### 5.3.5. Relação "For Order"

Esta relação é igual às anteriores, ou seja, representada por uma chave estrangeira, com a diferença sendo que esta relação é de um nodo "OrderHistory" para um nodo "Order".

#### 5.3.6. Relação "Lives At"

A relação "Lives At" substitui duas tabelas na base de dados original, "customer_address" e "address_status". A primeira tabela é igual as tabelas originais das relações "Contains" e "Written By" e portanto o processo de migração é igual. A segunda tabela é uma tabela lookup para os dois estados possíveis de um *Address*, "active" ou "inactive". Para simplificar a base de dados, decidimos adicionar diretamente o "address_status" aos atributos da relação em vez criar um tipo de nodo para o *lookup*.

#### 5.3.7. Relação "Located In"

A última relação a modelar é simples, sendo, como algumas das anteriores, expressa por uma chave estrangeira na base de dados original, que identificava a que país pertencia um certo endereço. É tranformada numa relação simples entre nodos do tipo "Address" e "Country".

### 5.4. Indexes

A migração dos indíces presente na base de dados original é relativamente fácil, pois a criação de indíces em Neo4J é bastante parecida à de Oracle, como podemos ver no seguinte excerto:

```
CREATE INDEX IF NOT EXISTS idx_book_title ON Book(title)
CREATE INDEX IF NOT EXISTS idx_customer_email ON Customer(email)
CREATE INDEX IF NOT EXISTS idx_order_date ON Order(order_date)
```

Também é importante referir que, ao criarmos os *Constraints de Uniqueness* para os IDs de cada nodo, como no seguinte excerto:

```
CREATE CONSTRAINT IF NOT EXISTS FOR (a:Author) REQUIRE a.author_id IS UNIQUE
```

É criado também um índice no atributo alvo do *Constraint*.

### 5.5. Procedure

O código abaixo de Neo4j define e instala um procedimento personalizado APOC chamado *updateOrderStatus*. O objetivo deste procedimento é atualizar o estado de uma encomenda (Order) e registar essa alteração no seu histórico (*OrderHistory*), mas apenas se o novo estado for diferente do estado atual mais recente.

```
CALL apoc.custom.installProcedure(
  'updateOrderStatus (p_order_id::INT, p_status_value::STRING) ::
(message::STRING)',
  'MATCH (order:Order {order_id: $p_order_id})
```

```
   OPTIONAL MATCH (order)-[:HAS_HISTORY]->(history:OrderHistory)
   WITH order, history
   ORDER BY history.status_date DESC
   WITH order, collect(history)[0] AS latestHistory

   WITH order,
        CASE WHEN latestHistory IS NULL THEN "" ELSE latestHistory.status_value
END AS previousStatus,
        $p_status_value AS newStatus

   WHERE previousStatus <> newStatus

   OPTIONAL MATCH (existing:OrderHistory)
   WITH order, previousStatus, newStatus, max(existing.history_id) AS maxId

   CREATE (newHistory:OrderHistory {
       history_id: CASE WHEN maxId IS NULL THEN 1 ELSE maxId + 1 END,
       status_value: newStatus,
       status_date: datetime()
   })
   CREATE (order)<-[:FOR_ORDER]-(newHistory)

   RETURN "Status atualizado com sucesso para o pedido " +
toString(order.order_id) AS message',
  'bookstore',
  'write'
);
```

Este procedimento, registado através de *CALL apoc.custom.installProcedure*, é projetado para atualizar o estado de uma encomenda e, simultaneamente, criar um registo histórico dessa alteração. Como parâmetros de entrada, o procedimento aceita *p_order_id* (o identificador inteiro da encomenda) e *p_status_value* (o novo valor de estado, em formato string). Ao concluir a sua execução, retorna uma *message* (string) a confirmar o resultado da operação. O procedimento começa por localizar o nó da encomenda com o ID fornecido. De seguida, são consultados todos os registos de histórico (*OrderHistory*) associados a essa encomenda. Esses registos são ordenados cronologicamente pela status_date em ordem decrescente, e a expressão *collect(history)[0]* é utilizada para selecionar o registo mais recente, que é então referido como *latestHistory*. Caso a encomenda não possua qualquer histórico prévio, *latestHistory* permanecerá nulo. O estado atual da encomenda, designado *previousStatus*, é extraído do *latestHistory*. Se não existir histórico, *previousStatus* é considerado uma string vazia. Este estado é então comparado com o *newStatus* (o estado recebido como parâmetro). Uma cláusula condicional WHERE previousStatus <> newStatus é fundamental, pois garante que o procedimento apenas prossiga se houver uma alteração efetiva no estado. Se a condição de alteração de estado for satisfeita, o procedimento avança para gerar um novo *history_id*. Este identificador é obtido calculando o valor máximo de *history_id* entre todos os nós *OrderHistory* existentes na base de dados e incrementando-o po uma unidade (maxId + 1). Na ausência de quaisquer registos de histórico prévios na base de dados (o que resultaria em maxId nulo), o *history_id* é inicializado com o valor 1. Com o *history_id* definido, um novo nó *OrderHistory* é criado, contendo o *history_id* gerado, o *status_value* correspondente ao novo estado da encomenda, e a data e hora atuais, capturadas pela função *datetime()*. Para conectar este novo nodo de histórico à encomenda respetiva, é criada a relação CREATE (order)<-[:FOR_ORDER]--(newHistory). Após a execução bem-sucedida destes passos, o procedimento retorna uma mensagem de sucesso.

## 5.6. Views

O Neo4j não possui um conceito nativo de "views". Assim, para replicar a funcionalidade que seria obtida através de views em SQL, foram desenvolvidas consultas diretas em Cypher que produzem o mesmo resultado ou conjunto de dados que a *view* SQL correspondente.

### 5.6.1. View 1

```
MATCH (b:Book)-[:WRITTEN_BY]->(a:Author)
RETURN b.book_id AS book_id, b.title AS title, a.author_name AS author_name
ORDER BY b.book_id, a.author_name
```

Esta consulta identifica todos os nós do tipo Book que possuem uma relação WRITTEN_BY com nós do tipo *Author*. Para cada combinação encontrada, retorna o *book_id* e *title* do livro, juntamente com o *author_name* do autor. Os resultados são ordenados primariamente pelo *book_id* e, depois, pelo *author_name*, garantindo uma listagem consistente.

### 5.6.2. View 2

```
MATCH (o:Order)<-[:FOR_ORDER]-(oh:OrderHistory)
WITH o, oh
ORDER BY oh.status_date DESC
WITH o, collect(oh)[0] AS latestHistory
RETURN
    o.order_id AS order_id,
    o.order_date AS order_date,
    latestHistory.status_value AS status_value
ORDER BY o.order_id
```

Esta consulta começa por identificar todas as encomendas (*Order*) que possuem registos de histórico (*OrderHistory*) associados através da relação FOR_ORDER. Para cada encomenda, os seus registos de histórico são ordenados pela *status_date* por ordem decrescente, garantindo que o mais recente apareça primeiro. De seguida, para cada encomenda, apenas o registo de histórico mais recente (*latestHistory*) é selecionado. Finalmente, a consulta retorna o *order_id* e a *order_date* da encomenda, acompanhados pelo *status_value* do seu histórico mais recente. Os resultados são apresentados ordenados pelo *order_id*.

## 5.7. Triggers

Para garantir a integridade dos dados, tal como faríamos com triggers em SQL, utilizámos os mecanismos de triggers disponibilizados pela biblioteca APOC no Neo4j.

### 5.7.1. validate_email

A validação do formato de email, que em SQL seria implementada com um único trigger para operações de INSERT e UPDATE, exigiu uma abordagem ligeiramente diferente no Neo4j. Foi necessário criar dois triggers distintos: um para validar o email durante a criação de novos nós de cliente e outro para validar o email durante a atualização dessa propriedade em nós existentes.

```
CALL apoc.trigger.install(
  'bookstore',
  'validate_email_update',
  '
   UNWIND keys($assignedNodeProperties) AS nodeId
   UNWIND $assignedNodeProperties[nodeId] AS propChange
```

```
   WITH propChange.node AS node, propChange.key AS propKey, propChange.new AS
newValue
   WHERE propKey = "email" AND NOT newValue =~ "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+
\.[A-Za-z]{2,}$"
   CALL apoc.util.validate(true, "Invalid email format.", [0])
  ',
  {phase: "before"}
);
```

```
CALL apoc.trigger.install(
  'bookstore',
  'validate_email_create',
  '
   UNWIND $createdNodes AS n
   WITH n
   WHERE n:Customer AND NOT n.email =~ "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-
z]{2,}$"
   CALL apoc.util.validate(true, "Invalid email format.", [0])
   RETURN n
  ',
  {phase: "before"}
);
```

Este trigger é acionado antes da conclusão de uma transação que cria novos nós. Regista um novo trigger na base de dados 'bookstore' com o nome *validate_email_create*, itera sobre todos os nós (n) que estão a ser criados na transação atual, filtra estes nós para atuar apenas sobre aqueles que possuem a *label Customer*, uma propriedade email e o valor da propriedade email não corresponde á expressão regular fornecida. Se as condições do WHERE forem satisfeitas (ou seja, um email inválido é encontrado num novo cliente), a função `CALL apoc.util.validate(true, "Invalid email format for new customer.", [0])` é chamada. Como o primeiro argumento é *true*, ela lança uma exceção com a mensagem especificada, o que causa a anulação da transação.

### 5.7.2. insert_order_history

Este trigger é configurado para ser acionado após a conclusão bem-sucedida (*commit*) e de forma assíncrona a uma transação que cria novos nós.

```
CALL apoc.trigger.install(
    'bookstore',
    'insert_order_history',
    'UNWIND $createdNodes AS node
     WITH node
     WHERE "Order" IN labels(node)
     MATCH (existing:OrderHistory)
     WITH node, max(existing.history_id) AS maxId
     CREATE (history:OrderHistory {
         history_id: CASE WHEN maxId IS NULL THEN 1 ELSE maxId + 1 END,
         status_date: datetime(),
         status_value: "Order Received"
     })
     CREATE (node)<-[:FOR_ORDER]-(history)',
    {phase: 'afterAsync'}
);
```

Começa por iterar sobre todos os nós que foram criados na transação que o ativou, utilizando `UNWIND \$createdNodes AS node`. De seguida, o trigger filtra estes nós com `"Order" IN labels(node)`, assegurando que a sua ação subsequente se aplique apenas aos nós que acabaram de ser criados e que possuem o rótulo *Order*. Para cada novo nó *Order* identificado, o trigger executa `MATCH (existing:OrderHistory) WITH node, max(existing.history_id) AS maxId`. Esta etapa consulta todos os nós *OrderHistory* já existentes na base de dados para determinar o valor máximo atual da propriedade *history_id*, designando-o como maxId. Este valor é crucial para gerar um identificador único para o novo registo de histórico. Com o maxId (ou null se nenhum histórico existir) e o nó da nova encomenda (node), o trigger procede à criação de um novo nó, e de seguida estabelece uma ligação entre a nova encomenda e o seu registo de histórico que acabou de ser criado.

### 5.7.3. prevent_book_deletion

Este trigger que impede a eliminação de um livro se este existir em encomendas, tem como objetivo garantir a integridade referencial. Antes de permitir que um registo de *book* seja eliminado, o trigger verifica se esse livro está referenciado na tabela *order_line*. Se estiver, a eliminação é impedida para evitar que as linhas de encomenda fiquem a referenciar um livro que já não existe. A tradução direta do trigger *prevent_book_deletion* não foi realizada porque o Neo4j, por defeito, já oferece uma proteção robusta contra a eliminação de nós que possuem relações, cumprindo o objetivo principal do trigger relacional sem necessidade de código adicional.

## 5.8. Queries

- **QUERY 1:** Listar todos os livros com seus respectivos autores e editoras.

```
MATCH (b:Book)-[:WRITTEN_BY]->(a:Author),
          (b)-[:PUBLISHED_BY]->(p:Publisher),
          (b)-[:IN_LANGUAGE]->(l:Language)
    RETURN b.title AS `Título do Livro`,
           collect(DISTINCT a.author_name) AS Autores,
           p.publisher_name AS Editora,
           l.language_name AS Idioma,
           b.num_pages AS Páginas,
           b.publication_date AS `Data de Publicação`
    ORDER BY b.title
```

Esta consulta começa por identificar todos os livros que possuem um autor, uma editora e um idioma associados através das respetivas relações. De seguida, para cada livro encontrado, retorna uma visão consolidada que inclui o seu título, a lista dos nomes dos autores, o nome da editora, o nome do idioma, o número de páginas e a data de publicação. Finalmente, os resultados são apresentados ordenados alfabeticamente pelos títulos dos livros.

- **QUERY 2:** Top 10 dos clientes com mais encomendas.

```
    MATCH (c:Customer)<-[:PLACED_BY]-(o:Order)-[co:CONTAINS]->(b:Book)
    WITH c, count(o) AS num_orders, sum(co.price) AS total_spent
    RETURN c.first_name + ' ' + c.last_name AS `Nome do Cliente`,
           c.email AS Email,
           num_orders AS `Número de Encomendas`,
           total_spent AS `Valor Total Gasto`
    ORDER BY num_orders DESC, total_spent DESC
    LIMIT 10
```

Esta consulta identifica clientes, as suas encomendas e os livros dentro dessas encomendas (incluindo o preço de cada livro). De seguida, para cada cliente, calcula o número total de encomendas realizadas e o valor total gasto em todos os livros dessas encomendas. Por fim, retorna o nome completo e o email do cliente, o seu número de encomendas e o valor total gasto, apresentando os 10 clientes com mais encomendas e maior valor gasto, por essa ordem.

- **QUERY 3:** Livros mais vendidos por quantidade

```
    MATCH (b:Book)<-[co:CONTAINS]-(o:Order)-[:PLACED_BY]->(c:Customer)
    MATCH (b)-[:WRITTEN_BY]->(a:Author)
    WITH b,
         collect(DISTINCT a.author_name) as authors,
         collect(DISTINCT c.customer_id) as unique_customers,
         count(DISTINCT co) as total_quantity,
         avg(co.price) as avg_price
    RETURN b.title AS `Título do Livro`,
           apoc.text.join(authors, ', ') AS Autores,
           total_quantity AS `Quantidade Total Vendida`,
           avg_price AS `Preço Médio`,
           size(unique_customers) AS `Clientes Únicos`
    ORDER BY avg_price DESC
    LIMIT 15;
```

Inicialmente, a consulta identifica todos os livros que foram incluídos em encomendas feitas por clientes, e também localiza os autores de cada um desses livros. Posteriormente, para cada livro, ela calcula diversas métricas agregadas: a lista dos seus autores, o número total de vezes que o livro foi vendido (quantidade total), o preço médio pelo qual foi vendido em todas as encomendas, e o número de clientes distintos que o adquiriram. Finalmente, a consulta retorna o título do livro, uma string com os nomes dos autores, a quantidade total vendida, o preço médio de venda e o número de clientes únicos que o compraram. Os resultados são ordenados de forma decrescente pelo preço médio, e apenas os 15 livros no topo desta lista são exibidos.

- **QUERY 4:** Análise de encomendas por país

```
    MATCH (co:Country)<-[:LOCATED_IN]-(a:Address)<-[:LIVES_AT]-(c:Customer)<-
[:PLACED_BY]-(o:Order)-[con:CONTAINS]->(b:Book)
    WITH co,
         count(DISTINCT c) AS `Clientes Únicos`,
         count(o) AS `Total de Encomendas`,
         sum(con.price) AS `Receita Total`,
         avg(con.price) AS `Valor Médio por Encomenda`
    RETURN co.country_name AS País,
           `Clientes Únicos`,
           `Total de Encomendas`,
           `Receita Total`,
           `Valor Médio por Encomenda`
    ORDER BY `Receita Total` DESC
```

Começa por identificar o percurso desde cada país até aos clientes aí localizados, seguindo para as encomendas feitas por esses clientes e, finalmente, para cada item de livro individual (con) dentro dessas encomendas, incluindo o preço de cada item. De seguida, para cada país, a consulta agrega várias métricas: o número de clientes distintos que efetuaram compras, uma contagem dos itens encomendados (denominada Total de Encomendas), a soma total dos preços de todos os itens vendidos

(resultando na Receita Total) e o preço médio por item de encomenda (denominado Valor Médio por Encomenda). Por fim, a consulta exibe o nome do país juntamente com estas estatísticas de vendas consolidadas, priorizando na listagem os países com maior receita total.

- **QUERY 5:** Estado atual das encomendas.

```
    MATCH (o:Order)<-[:FOR_ORDER]-(oh:OrderHistory)
    WITH o, oh ORDER BY oh.status_date DESC
    WITH o, head(collect(oh)) AS latest_oh
    WITH latest_oh.status_value AS current_status_value, count(o) AS
number_of_orders
    ORDER BY number_of_orders DESC


    WITH collect({status: current_status_value, count: number_of_orders}) AS
status_counts_list,
         sum(number_of_orders) AS total_orders_overall

    UNWIND status_counts_list AS status_data
    WITH status_data.status AS `Estado da Encomenda`,
         status_data.count AS `Número de Encomendas`,
         round((status_data.count * 100.0 / total_orders_overall), 2) AS
`Percentagem`

    RETURN `Estado da Encomenda`,
           `Número de Encomendas`,
           `Percentagem`
```

Inicialmente, para cada encomenda, a consulta identifica o seu estado mais recente, baseando-se no registo de histórico com a data mais atual. De seguida, agrupa todas as encomendas por este estado corrente e conta quantas encomendas se encontram em cada um destes estados. Esta contagem por estado é então ordenada de forma decrescente. Posteriormente, a consulta utiliza estes totais por estado e o número global de encomendas para calcular a percentagem que cada estado representa no conjunto total. Finalmente, a consulta retorna o nome de cada estado da encomenda, o número absoluto de encomendas nesse estado e a sua respetiva percentagem em relação ao total, mantendo a ordenação decrescente previamente estabelecida pelo número de encomendas.

- **QUERY 6:** Autores mais produtivos.

```
    MATCH (a:Author)<-[:WRITTEN_BY]-(b:Book)
    OPTIONAL MATCH (b)-[:PUBLISHED_BY]->(p:Publisher)
    OPTIONAL MATCH (b)<-[co:CONTAINS]-(:Order)
    WITH a, count(DISTINCT b) AS `Número de Livros`, count(DISTINCT p) AS
`Editoras Diferentes`,
         min(b.publication_date) AS `Primeira Publicação`, max(b.publication_date)
AS `Última Publicação`,
         count(DISTINCT co) AS `Total de Livros Vendidos`
    RETURN a.author_name AS `Nome do Autor`, `Número de Livros`, `Editoras
Diferentes`,
           `Primeira Publicação`, `Última Publicação`, `Total de Livros Vendidos`
    ORDER BY `Número de Livros` DESC
```

A consulta começa por identificar todos os autores e os respetivos livros que escreveram. Em seguida, de forma opcional, para cada livro, procura a editora associada e regista cada vez que o livro foi

incluído numa encomenda (representando uma venda). Com base nestas informações, para cada autor, a consulta calcula: o número total de livros distintos da sua autoria, a quantidade de editoras diferentes que publicaram as suas obras, as datas da primeira e da última publicação dos seus livros, e o número total de vezes que os seus livros foram vendidos. Finalmente, a consulta apresenta o nome de cada autor juntamente com estas métricas agregadas, listando primeiro os autores com maior número de livros publicados.

- **QUERY 7:** Análise temporal de vendas (por mês).

```
    MATCH (o:Order)-[r:CONTAINS]->(:Book)
    WHERE o.order_date IS NOT NULL
    WITH o.order_date.year AS ano,
         o.order_date.month AS mes_numero,
         CASE o.order_date.month
           WHEN 1 THEN 'Janeiro'
           WHEN 2 THEN 'Fevereiro'
           WHEN 3 THEN 'Março'
           WHEN 4 THEN 'Abril'
           WHEN 5 THEN 'Maio'
           WHEN 6 THEN 'Junho'
           WHEN 7 THEN 'Julho'
           WHEN 8 THEN 'Agosto'
           WHEN 9 THEN 'Setembro'
           WHEN 10 THEN 'Outubro'
           WHEN 11 THEN 'Novembro'
           WHEN 12 THEN 'Dezembro'
           ELSE toString(o.order_date.month)
         END AS nome_do_mes,
         count(DISTINCT o) AS numero_encomendas,
         count(r) AS livros_vendidos,
         sum(r.price) AS receita_total

    ORDER BY ano DESC, mes_numero DESC

    RETURN ano AS `Ano`,
           mes_numero AS `Mês`,
           nome_do_mes AS `Nome do Mês`,
           numero_encomendas AS `Encomendas`,
           livros_vendidos AS `Livros Vendidos`,
           receita_total AS `Receita Total`
```

A consulta inicia por identificar todas as encomendas que contêm livros e que possuem uma data de encomenda definida. Com base nestas encomendas, ela agrupa os dados por ano e por mês. Para cada um destes períodos mensais, a consulta calcula o número de encomendas únicas realizadas, a quantidade total de livros vendidos (contando cada item de livro numa encomenda) e a receita total obtida com essas vendas. Adicionalmente, converte o número do mês para o seu nome correspondente. Finalmente, a consulta retorna, para cada período mensal, o ano, o número e o nome do mês, o total de encomendas, o total de livros vendidos e a receita total, com os resultados organizados de forma cronológica inversa, mostrando os meses mais recentes primeiro.

- **QUERY 8:** Clientes com endereços múltiplos.

```
    MATCH (cust:Customer)-[r_lives_at:LIVES_AT]->(addr:Address)-[:LOCATED_IN]-
>(country:Country)
```

```
    WITH cust, addr, country, r_lives_at
    ORDER BY cust.customer_id, addr.address_id
    WITH cust,
         collect(
             toString(addr.street_number) + ' ' + addr.street_name + ', ' +
country.country_name + ' (' + r_lives_at.status + ')'
         ) AS address_strings_list,
         count(addr) AS numero_de_enderecos

    WHERE numero_de_enderecos > 1
    RETURN cust.first_name + ' ' + cust.last_name AS `Nome do Cliente`,
           cust.email AS `Email`,
           numero_de_enderecos AS `Número de Endereços`,
           apoc.text.join(address_strings_list, '; ') AS `Todos os Endereços`
    ORDER BY numero_de_enderecos DESC
```

Inicialmente, a consulta localiza clientes, os seus endereços e os respetivos países, incluindo também o estado de cada relação de morada (por exemplo, 'principal' ou 'anterior'). De seguida, para cada cliente, agrupa todos os seus endereços numa lista formatada (que inclui rua, número, país e o estado da morada) e conta o número total de endereços associados a esse cliente.

A consulta então filtra estes resultados para focar apenas nos clientes que têm mais do que um endereço registado. Por fim, para cada um destes clientes, retorna o seu nome completo, email, o número de endereços que possui, e uma string única que consolida todos os seus endereços formatados. Os resultados são apresentados ordenados de forma decrescente pelo número de endereços, destacando primeiro os clientes com mais moradas.

- **QUERY 9:** Livros por idioma e editora.

```
    MATCH (b:Book)-[:WRITTEN_IN]->(l:Language), (b)-[:PUBLISHED_BY]->(p:Publisher)
    WITH l, p, count(b) AS `Número de Livros`, avg(b.num_pages) AS `Média de
Páginas`, min(b.publication_date) AS `Livro Mais Antigo`, max(b.publication_date)
AS `Livro Mais Recente`
    RETURN l.language_name AS Idioma, p.publisher_name AS Editora, `Número de
Livros`, `Média de Páginas`, `Livro Mais Antigo`, `Livro Mais Recente`
    ORDER BY Idioma, `Número de Livros` DESC
```

Inicialmente, a consulta identifica todos os livros para os quais existe informação tanto sobre o idioma em que foram escritos como sobre a editora que os publicou. De seguida, para cada combinação distinta de idioma e editora, a consulta calcula o número total de livros correspondentes, a média do número de páginas desses livros, e as datas de publicação do livro mais antigo e do mais recente dentro desse grupo específico. Por fim, a consulta apresenta o nome do idioma, o nome da editora, e as métricas agregadas calculadas (número de livros, média de páginas, data do livro mais antigo e data do livro mais recente). Os resultados são ordenados primariamente pelo nome do idioma e, secundariamente, dentro de cada idioma, pelo número de livros em ordem decrescente, destacando as editoras mais prolíficas em cada língua.

- **QUERY 10:** Métodos de envio mais utilizados.

```
    MATCH (sm:ShippingMethod)<-[:SHIPPED_VIA]-(:Order)-[r_ol:CONTAINS]->(:Book)
    WITH sm,
         count(r_ol) AS numeroDeLinhasPorMetodo,
         avg(r_ol.price) AS valorMedioLinhaPorMetodo,
```

```
        sum(r_ol.price) AS receitaTotalPorMetodo

    WITH collect({
            method_name: sm.method_name,
            linhas: numeroDeLinhasPorMetodo,
            avg_linha_price: valorMedioLinhaPorMetodo,
            total_revenue: receitaTotalPorMetodo
        }) AS method_stats_list,
        sum(numeroDeLinhasPorMetodo) AS grandTotalLinhas
    UNWIND method_stats_list AS stats

    WITH stats.method_name AS metodoEnvio,
        stats.linhas AS numeroDeUtilizacoes,
        round(toFloat(stats.linhas) * 100.0 / grandTotalLinhas, 2) AS
percentagemDeUso,
        stats.avg_linha_price AS valorMedioEncomendas,
        stats.total_revenue AS receitaTotal

    RETURN metodoEnvio AS `Método de Envio`,
           numeroDeUtilizacoes AS `Número de Utilizações`,
           percentagemDeUso AS `Percentagem de Uso`,
           valorMedioEncomendas AS `Valor Médio das Encomendas`,
           receitaTotal AS `Receita Total`
    ORDER BY numeroDeUtilizacoes DESC
```

Inicialmente, a consulta identifica cada método de envio e todas as linhas de encomenda (ou seja, os itens individuais de livros vendidos) que foram processadas através desse método. Para cada método de envio, ela calcula o número total dessas linhas de encomenda, o valor médio de cada linha (preço médio por item) e a receita total gerada.

Numa etapa seguinte, a consulta agrupa estas estatísticas por método e calcula também o número total de linhas de encomenda em todos os métodos combinados. Isto permite, subsequentemente, determinar a percentagem de utilização de cada método de envio em relação ao total.

Finalmente, para cada método, a consulta apresenta o seu nome, o número total de vezes que foi utilizado (representado pelas linhas de encomenda), a sua percentagem de uso, o valor médio dos itens enviados através dele e a receita total que gerou. Os resultados são ordenados pelo número de utilizações, mostrando primeiro os métodos de envio mais populares.

- **QUERY 11:** Livros nunca vendidos.

```
    MATCH (b:Book)-[:PUBLISHED_BY]->(p:Publisher)
    MATCH (b)-[:IN_LANGUAGE]->(l:Language)
    WHERE NOT EXISTS ((:Order)-[:CONTAINS]->(b))
    WITH b, p, l
    MATCH (b)-[:WRITTEN_BY]->(a:Author)
    WITH b, p, l, a ORDER BY a.author_name
    WITH b, p, l, COLLECT(a.author_name) AS authorNames
    WITH b, p, l, authorNames,
        duration.inDays(date(b.publication_date), date()).days AS
daysSincePublication
    RETURN
        b.title AS `Título do Livro`,
        authorNames AS `Autores`, // This will be a list of author names, e.g.,
```

```
["Author A", "Author B"]
        p.publisher_name AS `Editora`,
        l.language_name AS `Idioma`,
        b.publication_date AS `Data de Publicação`, // Or use
toString(b.publication_date) if needed for specific formatting
        daysSincePublication AS `Dias Desde Publicação`

    ORDER BY b.publication_date DESC
```

A consulta começa por encontrar livros que têm uma editora e um idioma associados, mas que não estão ligados a nenhuma encomenda, indicando que nunca foram vendidos. Para cada um destes livros não vendidos, a consulta reúne os nomes dos seus autores e calcula o número de dias desde a sua data de publicação até à data atual. Por fim, para cada livro não vendido, são apresentados o seu título, a lista dos nomes dos autores, o nome da editora, o idioma, a data de publicação e o número de dias decorridos desde essa publicação. Os resultados são ordenados pela data de publicação em ordem decrescente, mostrando primeiro os livros não vendidos que foram publicados mais recentemente.

- **QUERY 12:** Análise de clientes por fidelidade.

```
    MATCH (c:Customer)<-[:PLACED_BY]-(o:Order)-[co:CONTAINS]->(b:Book)
    WITH c, count(o) AS total_orders, sum(co.price) AS total_spent,
min(o.order_date) AS first_order, max(o.order_date) AS last_order,
duration.inDays(min(o.order_date), max(o.order_date)).days AS
customer_lifespan_days
    RETURN c.first_name + ' ' + c.last_name AS `Nome do Cliente`, c.email AS
Email, total_orders AS `Total de Encomendas`, total_spent AS `Total Gasto`,
           CASE WHEN total_orders >= 10 AND total_spent >= 500 THEN 'VIP'
                WHEN total_orders >= 5 OR total_spent >= 200 THEN 'Fiel'
                WHEN total_orders >= 2 THEN 'Regular'
                ELSE 'Novo' END AS `Categoria de Cliente`,
           first_order AS `Primeira Encomenda`, last_order AS `Última Encomenda`,
customer_lifespan_days AS `Dias Como Cliente`
    ORDER BY total_spent DESC
```

Inicialmente, para cada cliente, a consulta agrega informações cruciais da sua interação com as encomendas: calcula o número total de itens de livros encomendados, o montante total que gastou, e identifica as datas da sua primeira e última encomenda. Com base nestas datas, determina também a duração, em dias, do período de atividade de compra do cliente. De seguida, utilizando o total de itens encomendados e o montante gasto, a consulta classifica cada cliente numa categoria específica ('VIP', 'Fiel', 'Regular' ou 'Novo'). Finalmente, são retornados o nome completo e o email do cliente, o total de itens encomendados, o valor total gasto, a categoria atribuída, as datas da primeira e última encomenda e o período de atividade como cliente em dias. Os resultados são ordenados de forma decrescente pelo total gasto, destacando no topo os clientes mais valiosos.

- **QUERY 13:** Relatório de performance por editora.

```
    MATCH (p:Publisher)
    WHERE EXISTS((p)<-[:PUBLISHED_BY]-(:Book))
    CALL {
        WITH p
        MATCH (p)<-[:PUBLISHED_BY]-(b_sub:Book)
        RETURN count(b_sub) AS livrosNoCatalogoCalc
```

```
    }
    CALL {
        WITH p
        MATCH (p)<-[:PUBLISHED_BY]-(b_sub:Book)-[:WRITTEN_BY]->(a_sub:Author)
        RETURN count(DISTINCT a_sub) AS autoresUnicosCalc
    }
    CALL {
        WITH p
        MATCH (p)<-[:PUBLISHED_BY]-(b_sub:Book)-[:IN_LANGUAGE]->(l_sub:Language)
        RETURN count(DISTINCT l_sub) AS idiomasPublicadosCalc
    }
    WITH p, livrosNoCatalogoCalc, autoresUnicosCalc, idiomasPublicadosCalc
    MATCH (p)<-[:PUBLISHED_BY]-(b_for_sales:Book)
    OPTIONAL MATCH (b_for_sales)<-[ol:CONTAINS]-(:Order)

    WITH p,
         livrosNoCatalogoCalc,
         autoresUnicosCalc,
         idiomasPublicadosCalc,
         count(ol) AS totalLinhasVendidasCalc,
         sum(ol.price) AS receitaTotalCalc

    WITH p.publisher_name AS editoraName,
         livrosNoCatalogoCalc,
         autoresUnicosCalc,
         idiomasPublicadosCalc,
         totalLinhasVendidasCalc,
         coalesce(receitaTotalCalc, 0) AS receitaTotalFinal,
         round(toFloat(totalLinhasVendidasCalc) / livrosNoCatalogoCalc, 2) AS
mediaVendasPorLivroCalc

    RETURN
        editoraName AS `Editora`,
        livrosNoCatalogoCalc AS `Livros no Catálogo`,
        autoresUnicosCalc AS `Autores Únicos`,
        totalLinhasVendidasCalc AS `Total de Linhas Vendidas`,
        receitaTotalFinal AS `Receita Total`,
        mediaVendasPorLivroCalc AS `Média de Vendas por Livro`,
        idiomasPublicadosCalc AS `Idiomas Publicados`
    ORDER BY receitaTotalFinal DESC
```

A consulta começa por identificar as editoras ativas e, para cada uma, utiliza subconsultas para determinar o número total de livros no seu catálogo, a quantidade de autores distintos associados a esses livros e o número de idiomas diferentes em que publicam. Após estas contagens iniciais, a consulta foca-se nos aspetos de vendas, calculando para cada editora o número total de linhas de encomenda (representando vendas individuais de livros) e a receita total gerada por essas vendas.

Com todos estes dados, a consulta calcula ainda a receita total final (convertendo para zero caso não existam vendas) e a média de vendas por livro para cada editora. Por fim, são apresentados o nome da editora juntamente com um conjunto completo de métricas: o número de livros no catálogo, o total de autores únicos, o total de linhas vendidas, a receita total, a média de vendas por livro e o número de idiomas publicados, com as editoras que geraram maior receita a ficarem no topo da lista.

# 6. Análise Crítica

## 6.1. Comparação dos resultados

| Número | Query | Oracle(ms) | MongoDB(ms) | Neo4j(ms) |
|---|---|---|---|---|
| 1 | Listar todos os livros com seus autores e editoras | 260 ms | 222.68 ms | 181.5 ms |
| 2 | Top 10 clientes com mais encomendas | 90 ms | 190.65 ms | 130.4 ms |
| 3 | Livros mais vendidos por quantidade | 300 ms | 5180.10 ms | 149.2 ms |
| 4 | Análise de encomendas por país | 450 ms | 52.32 ms | 83.2 ms |
| 5 | Estado atual das encomendas | 430 ms | 69.37 ms | 99.1 ms |
| 6 | Autores mais produtivos | 110 ms | 278080.25 ms | 70.5 ms |
| 7 | Análise temporal de vendas (por mês) | 60 ms | 51.51 ms | 129.8 ms |
| 8 | Clientes com endereços múltiplos | 170 ms | 55.01 ms | 109.1 ms |
| 9 | Livros por idioma e editora | 80 ms | 74.59 ms | 46.4 ms |
| 10 | Métodos de envio mais utilizados | 30 ms | 43.50 ms | 55.1 ms |
| 11 | Livros nunca vendidos | 110 ms | 190988.64 ms | 150.2 ms |
| 12 | Análise de clientes por fidelidade | 30 ms | 13553.78 ms | 64.5 ms |
| 13 | Relatório de performance por editora | 290 ms | 72230.73 ms | 151.3 ms |

## 6.2. Base de dados relacional

### 6.2.1. Padrões de Otimização Identificados:

A eficiência dos relacionamentos diretos no Oracle manifesta-se através da performance consistente das queries que utilizam JOINs entre tabelas principais com relacionamentos 1:N, onde os *foreign key indexes* permitem navegação rápida entre as entidades relacionadas sem a necessidade de *scans* completos das tabelas. As *window functions* demonstram superioridade técnica notável, com operações como LAG, LEAD e RANK sendo processadas de forma extremamente eficiente através de algoritmos internos otimizados que explicam a excelente performance observada nas queries onde estas funções são utilizadas para análises temporais e de ranking (*query* 12, por exemplo). As agregações nativas do Oracle beneficiam significativamente de algoritmos *hash-based* e *sort-merge* altamente otimizados, permitindo que operações como COUNT, SUM, AVG e GROUP BY sejam executadas com performance superior através de estruturas de dados especializadas e processamento paralelo interno. A otimização específica para operações TOP-N e FETCH FIRST representa uma vantagem crucial, pois permite ao Oracle interromper o processamento assim que encontra o número necessário de resultados, evitando o processamento desnecessário de dados adicionais. A gestão inteligente de memória do Oracle mostra-se fundamental para a performance superior, especialmente nas consultas mais simples, onde os dados frequentemente acedidos permanecem no *buffer* cache, resultando numa performance superior através da redução significativa das operações de I/O em disco.

#### 6.2.2. Pontos Fortes do Oracle SQL:

- JOINs Otimizados: O Oracle tem décadas de otimização em algoritmos de JOIN (*nested loop, hash join, sort-merge join*), com planeadores de *query* sofisticados que escolhem automaticamente as melhores estratégias de execução baseadas nas estatísticas das tabelas e distribuição dos dados.
- Agregações Complexas: Operações como GROUP BY, *window functions* (RANK, ROW_NUMBER, LAG, LEAD), e cálculos estatísticos são implementadas ao nível do motor de base de dados com algoritmos altamente otimizados que aproveitam paralelização e estruturas de dados eficientes.
- Índices Sofisticados: Suporte nativo a múltiplos tipos de índices especializados (B-tree indexes, Bitmap indexes, etc).
- Otimizações de Memória: Gestão inteligente de cache que mantém dados frequentemente acedidos em memória, reduzindo drasticamente o I/O para queries repetitivas.
- Execução Paralela: Capacidade de paralelizar automaticamente queries complexas através de múltiplos processos, especialmente benéfico para operações de *scan* e agregação em grandes volumes de dados.
- Estatísticas Automáticas: Recolha e manutenção automática de estatísticas sobre distribuição de dados, permitindo ao otimizador tomar decisões mais precisas sobre planos de execução.

## 6.3. MongoDB

### 6.3.1. Análise Individual das Queries

#### 6.3.1.1. Query 1: Listar todos os livros com os seus autores e editoras

Tempos: Oracle: 260ms | MongoDB: 222.68ms

O MongoDB tem vantagem ligeira devido à sua capacidade de armazenar dados relacionados de forma desnormalizada. Se os documentos de livros incluem informações dos autores e editoras embebidas, elimina-se a necessidade de múltiplos JOINs. O Oracle precisa navegar através de 4 tabelas relacionais com JOINs complexos, enquanto o MongoDB pode resolver isto com menos operações de lookup.

#### 6.3.1.2. Query 2: Top 10 clientes com mais encomendas

Tempos: Oracle: 90ms | MongoDB: 190.65ms

O Oracle é mais eficiente nesta query de agregação e ordenação. Os bancos relacionais são otimizados para operações GROUP BY e ORDER BY, especialmente quando há índices apropriados. O MongoDB, embora tenha o *framework* de agregação, requer mais processamento para calcular estas métricas, particularmente quando precisa de fazer lookup entre coleções de clientes e encomendas.

#### 6.3.1.3. Query 3: Livros mais vendidos por quantidade

Tempos: Oracle: 300ms | MongoDB: 5180.10ms

Nesta *query* nota se uma diferenças significativa. O Oracle beneficia enormemente de índices otimizados e do processamento eficiente de JOINs complexos. A query envolve múltiplas agregações (COUNT, AVG, COUNT DISTINCT) que são altamente otimizadas em SQL. O MongoDB da resultados muitos piores porque precisa de múltiplos estágios de agregação e possivelmente múltiplos $*lookup* operations, tornando-se computacionalmente mais pesado.

#### 6.3.1.4. Query 4: Análise de encomendas por país

Tempos: Oracle: 450ms | MongoDB: 52.32ms

O MongoDB obtem melhores resultados nesta *query* geográfica. Se os documentos de encomendas incluem informação do país embebida ou indexada geograficamente, o MongoDB pode resolver esta consulta muito mais rapidamente. O Oracle precisa de JOIN através de múltiplas tabelas (país →

endereço → cliente → encomenda), enquanto o MongoDB pode ter esta informação mais acessível na estrutura do documento.

#### 6.3.1.5. Query 5: Estado atual das encomendas

Tempos: Oracle: 430ms | MongoDB: 69.37ms

O MongoDB é significativamente mais rápido aqui. Esta *query* envolve encontrar o estado mais recente de cada encomenda, o que pode ser otimizado no MongoDB mantendo o estado atual diretamente no documento da encomenda ou usando índices temporais eficientes. O Oracle precisa de uma *subquery* complexa para encontrar a data máxima do histórico.

#### 6.3.1.6. Query 6: Autores mais produtivos

Tempos: Oracle: 110ms | MongoDB: 278080.25ms

Esta é a pior performance do MongoDB da nossa análise. A query requer agregações complexas através de múltiplas entidades (autores, livros, editoras, vendas). O Oracle beneficia dos seus algoritmos de JOIN otimizados e do planeador de queries sofisticado. O MongoDB precisa de múltiplos estágios de agregação com $lookup operations caros, resultando numa performance dramaticamente pior.

#### 6.3.1.7. Query 7: Análise temporal de vendas (por mês)

Tempos: Oracle: 60ms | MongoDB: 51.51ms

Ambos têm uma performance similar, com ligeira vantagem para o MongoDB. As funções de data são bem otimizadas em ambos os sistemas. O MongoDB pode ter vantagem se as datas estão bem indexadas e a agregação temporal é *straightforward*, ou seja, quando a query envolve operações temporais simples e diretas.

#### 6.3.1.8. Query 8: Clientes com endereços múltiplos

Tempos: Oracle: 170ms | MongoDB: 55.01ms

O MongoDB é mais eficiente aqui. Se os endereços estão embebidos nos documentos dos clientes como *arrays*, o MongoDB pode resolver esta *query* mais rapidamente usando operadores de *array*. O Oracle precisa de JOINs através da tabela de relacionamento *customer_address*.

#### 6.3.1.9. Query 9: Livros por idioma e editora

Tempos: Oracle: 80ms | MongoDB: 74.59ms

Performance muito similar. Esta é uma agregação relativamente simples que ambos os sistemas conseguem resolver eficientemente. O MongoDB tem ligeira vantagem, possivelmente devido à estrutura dos documentos que pode incluir idioma e editora como campos diretos.

#### 6.3.1.10. Query 10: Métodos de envio mais utilizados

Tempos: Oracle: 30ms | MongoDB: 43.50ms

O Oracle é ligeiramente mais eficiente nesta *query* de agregação simples. As operações de GROUP BY e cálculos percentuais são muito otimizadas em SQL, enquanto o MongoDB requer um pipeline de agregação mais complexo.

#### 6.3.1.11. Query 11: Livros nunca vendidos

Tempos: Oracle: 110ms | MongoDB: 190988.64ms

Outra performance dramaticamente pior no MongoDB. Esta *query* requer um LEFT JOIN complexo para encontrar livros sem vendas. O MongoDB precisa de fazer múltiplos $lookup operations e depois filtrar os resultados onde não há *matches*, o que é computacionalmente muito caro comparado com o LEFT JOIN otimizado do Oracle.

#### 6.3.1.12. Query 12: Análise de clientes por fidelidade

Tempos: Oracle: 30ms | MongoDB: 13553.78ms

O Oracle é muito superior nesta *query* que usa CTEs e análises complexas de *window functions*. O MongoDB precisa de múltiplos estágios de agregação para calcular as estatísticas de clientes e depois aplicar a lógica de categorização, resultando numa performance muito pior.

#### 6.3.1.13. Query 13: Relatório de performance por editora

Tempos: Oracle: 290ms | MongoDB: 72230.73ms

Esta query envolve múltiplas subqueries e cálculos complexos que são muito otimizados no Oracle. O MongoDB precisa de múltiplos $lookup operations e agregações aninhadas, resultando numa performance significativamente pior.

### 6.3.2. Pontos Fortes do MongoDB:

- Estrutura de Documentos: Quando os dados estão desnormalizados apropriadamente, elimina-se a necessidade de JOINs.
- *Queries* Geográficas e Temporais: Otimizações específicas para dados temporais e geoespaciais.
- Simplicidade Estrutural: *Queries* que não requerem relacionamentos complexos podem ser mais rápidas.
- Flexibilidade de *Schema*: Permite otimizações específicas baseadas na estrutura dos documentos.

### 6.3.3. Conclusões

Com base nos resultados e na análise temos que:

- *Queries* Relacionais Complexas: O Oracle é consistentemente superior (nas queries 3, 6, 13, 14) quando há múltiplos relacionamentos e agregações complexas.
- *Queries* Simples ou Bem Estruturadas: O MongoDB pode ter vantagem (nas queries 4, 5, 7, 8) quando a estrutura dos documentos está alinhada com o padrão de acesso.
- Diferenças significativas: As maiores discrepâncias favorecem o Oracle, sugerindo que queries analíticas complexas são o ponto forte dos bancos relacionais.
- Consistência: O Oracle mostra performance mais consistente e previsível, enquanto o MongoDB tem variação maior dependendo da complexidade da query.

Os resultados confirmam que bancos relacionais como o Oracle são superiores para análises complexas e queries que requerem relacionamentos sofisticados entre entidades. O MongoDB pode ser competitivo em cenários específicos onde a estrutura dos documentos está otimizada para os padrões de acesso, mas sofre significativamente em operações analíticas complexas que são comuns em sistemas OLAP ou ainda em relatórios empresariais.

## 6.4. Neo4j

### 6.4.1. Análise Individual das Queries

#### 6.4.1.1. Query 1: Listar todos os livros com os seus autores e editoras

Tempos: Oracle: 260ms | Neo4j: 181.5ms

O Neo4j apresenta uma performance excelente nesta query, significativamente mais rápido que o Oracle e mesmo que o MongoDB. Isto deve-se à natureza do Neo4j de percorrer relações diretas ([:WRITTEN_BY], [:PUBLISHED_BY], [:IN_LANGUAGE]). Para um nó *Book*, encontrar os seus autores, editora e idioma é uma travessia eficiente de ponteiros diretos, evitando a complexidade dos JOINs de múltiplas tabelas relacionais que o Oracle necessita.

#### 6.4.1.2. Query 2: Top 10 clientes com mais encomendas

Tempos: Oracle: 90ms | Neo4j: 130.4ms

A travessia de (Customer)<-[:PLACED_BY]-(Order)-[:CONTAINS]->(Book) é natural para o Neo4j. As agregações *count(o)* e *sum(co.price)* agrupadas por cliente (c) são eficientes. Embora o Oracle seja altamente otimizado para GROUP BY e ORDER BY com índices, o Neo4j lida bem com este tipo de agregação em dados conectados, especialmente se o número de relações por nó não for excessivamente grande.

#### 6.4.1.3. Query 3: Livros mais vendidos por quantidade

Tempos: Oracle: 300ms | Neo4j: 149.2ms

O Neo4j é notavelmente mais rápido que o Oracle. A capacidade de agregar informações como collect(DISTINCT a.author_name), collect(DISTINCT c.customer_id), count(DISTINCT co) e avg(co.price) por livro (b) após travessias de relações é uma força do Neo4j. A consulta beneficia da facilidade com que o Neo4j navega e agrega dados de nós e relações adjacentes. A função *apoc.text.join* é também eficiente para formatação.

#### 6.4.1.4. Query 4: Análise de encomendas por país

Tempos: Oracle: 450ms | Neo4j: 83.2ms

O Neo4j tem uma performance competitiva, consideravelmente mais rápido que o Oracle. A travessia longa (Country)<-[:LOCATED_IN]-(Address)<-[:LIVES_AT]-(Customer)<-[:PLACED_BY]--(Order)-[:CONTAINS]->(Book) é o forte do Neo4j. Cada seta (<- ou ->) é uma navegação eficiente. Agrupar por país (co) e calcular count(DISTINCT c), count(o), sum(con.price) e avg(con.price) é direto.

#### 6.4.1.5. Query 5: Estado atual das encomendas

Tempos: Oracle: 430ms | Neo4j: 99.1ms

O Neo4j tem uma performance muito melhor que o Oracle. Encontrar o *OrderHistory* mais recente (ORDER BY oh.status_date DESC e depois head(collect(oh))) por encomenda é uma operação que o Neo4j realiza bem. As agregações subsequentes para contar encomendas por estado e calcular percentagens são também eficazes. A capacidade de passar coleções e totais entre cláusulas WITH para cálculos progressivos é uma vantagem.

#### 6.4.1.6. Query 6: Autores mais produtivos

Tempos: Oracle: 110ms | Neo4j: 70.5ms

O Neo4j apresenta a melhor performance, sendo mais rápido que o Oracle. A consulta envolve encontrar autores, os seus livros, e opcionalmente editoras e informações de vendas (OPTIONAL MATCH). As agregações count(DISTINCT b), count(DISTINCT p), min/max(b.publication_date) e count(DISTINCT co) por autor são operações que o Neo4j lida muito bem, especialmente porque o OPTIONAL MATCH é tratado eficientemente sem o custo de *outer joins* complexos.

#### 6.4.1.7. Query 7: Análise temporal de vendas (por mês)

Tempos: Oracle: 60ms | Neo4j: 129.8ms

Nesta query, o Neo4j é um pouco mais lento que o Oracle. Embora o Neo4j possua boas capacidades de manipulação de datas e agregações, a forma como os dados temporais são extraídos (o.order_date.year, o.order_date.month) e agrupados pode não ser tão otimizada internamente como as funções de data e indexação temporal em Oracle.

#### 6.4.1.8. Query 8: Clientes com endereços múltiplos

Tempos: Oracle: 170ms | Neo4j: 109.1ms

O Neo4j tem uma performance sólida, mais rápido que o Oracle. A travessia (Customer)--[:LIVES_AT]->(Address)-[:LOCATED_IN]->(Country) é eficiente. A coleta de strings de endereço formatadas e a contagem de endereços por cliente (count(addr)) são bem geridas. O WHERE numero_de_enderecos > 1 filtra eficientemente.

#### 6.4.1.9. Query 9: Livros por idioma e editora

Tempos: Oracle: 80ms | Neo4j: 46.4ms

O Neo4j é o mais rápido nesta *query*. A consulta envolve encontrar livros com um idioma e editora e depois agregar por essa combinação. O MATCH (b:Book)-[:WRITTEN_IN]->(l:Language), (b)-[:PUBLISHED_BY]->(p:Publisher) identifica rapidamente os padrões, e a agregação WITH l, p, count(b), avg(b.num_pages), min/max(b.publication_date) é muito eficiente para o Neo4j, pois opera sobre dados já localizados e conectados.

#### 6.4.1.10. Query 10: Métodos de envio mais utilizados

Tempos: Oracle: 30ms | Neo4j: 55.1ms

O Neo4j é ligeiramente mais lento que o Oracle. A *query* agrega dados por método de envio e depois calcula percentagens. A agregação inicial WITH sm, count(r_ol), avg(r_ol.price), sum(r_ol.price) é eficiente. O passo subsequente de coletar tudo numa lista, calcular um total geral e depois fazer UNWIND para calcular percentagens, embora funcional, pode introduzir alguma sobrecarga comparado com otimizações SQL diretas para este tipo de cálculo.

#### 6.4.1.11. Query 11: Livros nunca vendidos

Tempos: Oracle: 110ms | Neo4j: 150.2ms

O Neo4j é um pouco mais lento que o Oracle. A cláusula WHERE NOT EXISTS ((:Order)-[:CONTAINS]->(b)) é uma forma eficiente no Cypher de verificar a não existência de uma relação/padrão, o que é análogo a um LEFT JOIN ... WHERE key IS NULL ou NOT IN em SQL. O Neo4j é otimizado para este tipo de verificação de padrões existenciais.

#### 6.4.1.12. Query 12: Análise de clientes por fidelidade

Tempos: Oracle: 30ms | Neo4j: 64.5ms

O Neo4j é um pouco mais lento que o Oracle. A consulta agrega dados por cliente (count(o), sum(co.price), min/max(o.order_date), duration) e depois aplica uma lógica CASE para categorização. O Neo4j processa estas agregações e a lógica condicional de forma eficaz sobre os dados conectados.

#### 6.4.1.13. Query 13: Relatório de performance por editora

Tempos: Oracle: 290ms | Neo4j: 151.3ms

O Neo4j é significativamente mais rápido que o Oracle. A utilização de subconsultas CALL {} para calcular *livrosNoCatalogoCalc*, *autoresUnicosCalc* e *idiomasPublicadosCalc* por editora é uma forma modular e eficiente de realizar agregações parciais. A subsequente agregação de dados de vendas (count(ol), sum(ol.price)) e o cálculo de *mediaVendasPorLivroCalc* são bem geridos. O Neo4j beneficia da capacidade de passar resultados de subconsultas e continuar o processamento de forma fluida

### 6.4.2. Pontos Fortes do Neo4j:

- Travessia de Relações Diretas: A performance é excecional em queries que navegam relações entre entidades (como livros para autores, encomendas para clientes).
- Agregações em Dados Conectados: Demonstra grande eficiência ao agregar dados que estão diretamente ou indiretamente conectados através de relações.

- Queries com Múltiplos Graus de Separação (Travessias Profundas): O Neo4j é otimizado para queries que envolvem seguir cadeias longas de relações, onde as bases de dados relacionais necessitariam de múltiplos JOINs custosos.
- Verificações de Existência de Padrões (WHERE NOT EXISTS): É otimizado para verificar a não existência de relações ou padrões.
- Subconsultas Modulares (CALL {}): A capacidade de usar subconsultas permite decompor problemas complexos em partes mais simples e agregar resultados progressivamente.

#### 6.4.3. Conclusões

Com base nos resultados e na análise temos que:

- **Queries Dependentes de Relações e Travessias Profundas:** O Neo4j é consistentemente superior (verificado, por exemplo, nas queries 1, 3, 4, 6, 9, 11, 13) quando a consulta se baseia na navegação e análise de múltiplas relações interconectadas.
- **Agregações Complexas sobre Dados Conectados:** O Neo4j demonstra uma forte capacidade para realizar agregações que dependem da recolha de informações através de diversos nós e relações (como nas queries 2, 3, 5, 12 e, de forma proeminente, na 13 com subconsultas). Isto simplifica consultas que seriam mais dificeis noutros sistemas.
- **Diferenças Significativas em Padrões de Grafo:** Para queries que exploram intensivamente a estrutura do grafo, como a identificação de nós que não possuem certas relações (query 11 - livros não vendidos) ou a compilação de relatórios complexos baseados em múltiplas facetas dos dados conectados (query 13 - performance de editoras), o Neo4j oferece vantagens de performance claras, especialmente quando comparado com a simulação dessas operações em MongoDB.
- **Consistência e Performance Geral:** O Neo4j exibiu uma performance geral robusta e tempos de execução consistentemente baixos (total de 1.4821 segundos para as 13 queries). Embora possa haver variações (queries 7 e 10 foram competitivas mas não as mais rápidas entre os três sistemas), a sua arquitetura orientada a grafos provou ser altamente eficiente para a maioria das queries analíticas apresentadas.

Os resultados confirmam que bases de dados orientadas a grafos como o Neo4j são particularmente adequadas para análises que se focam nas interconexões e relações entre os dados.

# 7. Recomendações e considerações futuras para melhorias no sistema.

Com base na experiência adquirida durante o desenvolvimento deste projeto, identificamos diversas oportunidades de melhoria e expansão que podem ser consideradas em trabalhos futuros.

Primeiramente, achamos que seria interessante realizar testes de desempenho mais aprofundados, com volumes de dados maiores e cenários de uso mais próximos da realidade, a fim de avaliar a escalabilidade de cada sistema em ambientes de produção.

No caso do Neo4j, seria relevante aprofundar o uso de algoritmos de grafos para funcionalidades avançadas, como sistemas de recomendação, análise de comunidades ou detecção de padrões de comportamento dos leitores.

Finalmente, poderíamos combinar os diferentes tipos de bases de dados conforme os requisitos de cada componente do sistema. Por exemplo, o OracleDB poderia ser utilizado para operações transacionais e consistência de dados, o MongoDB para armazenamento flexível de catálogos e metadados, e o Neo4j para análise de relações entre os autores, preferências de leitores, etc. Para viabilizar esta integração de forma eficaz, poderíamos procurar desenvolver processos de ETL (*Extract, Transform, Load*) que permitam extrair dados de uma fonte, transformá-los conforme o modelo de destino e carregá-los nos diferentes sistemas. A utilização de ferramentas ou *scripts* de ETL poderá facilitar a sincronização entre as bases de dados e a automação da migração e atualização dos dados, possibilitando um ecossistema coeso e mais eficiente.

# 8. Conclusão

Em suma, consideramos que os objetivos propostos para este projeto foram plenamente alcançados.

Através da experimentação com diferentes paradigmas de bases de dados, isto é, relacional (*OracleDB*), orientado a documentos (*MongoDB*) e orientado a grafos (*Neo4j*), foi possível analisar de forma crítica as suas principais diferenças, vantagens e limitações no contexto fornecido de uma livraria.

O OracleDB demonstrou ser altamente eficiente para transações completas, integridade referencial e consultas estruturadas, revelando-se adequado para sistemas com regras de negócio bem definidas. No entanto, a sua rigidez dificulta a gestão de dados não estruturados ou altamente inter-relacionados.

O MongoDB destacou-se pela sua flexibilidade e escalabilidade, sendo especialmente indicado para dados semiestruturados e aplicações que exigem ciclos rápidos de desenvolvimento. A sua principal limitação reside na menor adequação a cenários com transações complexas ou fortes exigências de consistência.

Já o Neo4j mostrou ser a solução ideal para modelar e explorar relações complexas entre entidades, graças à sua estrutura de grafos. Contudo, apresenta desafios na modelação e performance quando aplicado a grandes volumes de dados distribuídos.

Em conclusão, este projeto permitiu-nos desenvolver uma compreensão abrangente sobre como diferentes modelos de bases de dados podem ser aplicados de forma estratégica conforme os requisitos específicos de um sistema. A experiência prática com migração e adaptação de dados reforçou a importância de escolher o paradigma certo com base na natureza dos dados, das consultas e das necessidades da aplicação.