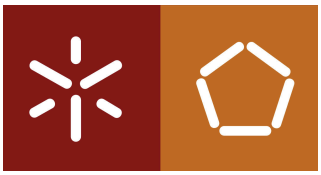

Junho 02, 2025

BDNoSQL - Trabalho Prático
Grupo 5

António Silva (pg57867)
João Pastore (pg55963)
Jorge Rodrigues (pg55966)
Tiago Rodrigues (pg56013)

Índice

1) Introdução

Atualmente, o armazenamento de dados assume um papel fulcral no desenvolvimento de sistemas informáticos, sendo um dos pilares fundamentais para a gestão eficiente de informação. Com o crescimento exponencial da quantidade de dados gerados e processados, torna-se imperativo adotar soluções de armazenamento que se adequem às necessidades específicas de cada aplicação.

Existem atualmente diversos paradigmas de bases de dados, entre os quais se destacam o modelo relacional, o modelo orientado a documentos e o modelo baseado em grafos. Cada um destes paradigmas apresenta características distintas ao nível da estruturação, consulta e escalabilidade dos dados, influenciando diretamente como a informação é persistida e manipulada.

Apesar de estes modelos poderem ser utilizados como base de dados, a escolha do paradigma mais adequado depende fortemente da natureza do problema a resolver. Fatores como a complexidade das relações entre entidades, a flexibilidade do esquema de dados e os requisitos de desempenho influenciam essa decisão.

O objetivo deste trabalho é explorar as diferenças entre o paradigma relacional, com opções também conhecidas como *NoSQL*, mais concretamente, documental e em grafo, aplicando-os a um mesmo problema. Pretende-se analisar as vantagens e desvantagens de cada abordagem, para compreender melhor as suas potencialidades e limitações em contextos práticos.

2) Sistemas de Gestão de Base de Dados

O modelo de dados utilizado neste trabalho representa o contexto de uma livraria, abrangendo não apenas as informações relacionadas com os livros e os respetivos processos de conceção (como autores e editoras), mas também os dados associados às encomendas realizadas e aos clientes que as efetuam. Este cenário permite explorar uma variedade de relações entre entidades, oferecendo um caso de estudo concreto para analisar a aplicação de diferentes paradigmas.

2.1) Modelo Relacional

O sistema relacional foi desenvolvido com recurso ao OracleDB, um sistema robusto e amplamente utilizado em ambientes empresariais. A implementação fornecida inclui a criação das tabelas, o respetivo povoamento, bem como a definição de objetos adicionais como *procedures*, *triggers* e *views*, que permitem uma maior funcionalidade e controlo sobre os dados.

Através da análise do modelo relacional, é possível inferir vários tipos de relacionamentos entre as entidades. Por exemplo, verifica-se uma relação de muitos-para-muitos (N:N) entre livros e encomendas, uma vez que um livro pode estar presente em várias encomendas e uma encomenda pode incluir vários livros. Adicionalmente, existe uma relação de um-para-muitos (1:N) entre livros e editoras, dado que uma editora pode ter vários livros. Estes relacionamentos são expressos, dependendo da natureza do relacionamento, via tabelas intermédias e chaves estrangeiras, respeitando os princípios da normalização.

2.2) Modelos Não Relacionais

Contrariamente ao caso anterior, este trabalho explora mais as abordagens não relacionais, utilizando dois sistemas distintos: MongoDB e Neo4j, representando, respetivamente, os modelos documentais e em grafo.

MongoDB é um sistema de gestão de bases de dados orientado a documentos, amplamente utilizado no contexto de aplicações *web* e distribuídas. Neste paradigma, os dados são armazenados em documentos no formato BSON (uma extensão binária do JSON), permitindo uma estrutura flexível e parcialmente estruturada. Esta flexibilidade torna o MongoDB eficaz em cenários onde os esquemas de dados podem variar ou evoluir ao longo do tempo, facilitando a representação de entidades complexas e aninhadas.

Neo4j, por sua vez, é um sistema de gestão de bases de dados orientado a grafos, projetado especificamente para lidar com dados altamente interligados. Neste modelo, a informação é armazenada sob a forma de nós (entidades) e relações (arestas) entre nós, ambas podendo conter propriedades. A estrutura em grafo permite uma navegação rápida entre relações, sendo especialmente vantajosa para consultas que envolvem múltiplos níveis de ligação ou relacionamentos complexos, como a associação entre clientes, encomendas e livros.

A utilização destes dois paradigmas não relacionais permite uma análise comparativa das suas capacidades face ao modelo relacional, evidenciando as vantagens e desvantagens de cada abordagem na representação e consulta de dados num mesmo domínio de aplicação.

Figura 1: Motores de Base de Dados Utilizados

3) Migração de Dados e Implementação do Sistema

O objetivo desta etapa consiste em partir da implementação atual baseada no modelo relacional, desenvolvida em OracleDB, e proceder à sua conversão para os paradigmas não relacionais garantindo a preservação integral da informação existente e assegurando, sempre que possível, capacidades funcionais equivalentes ao nível dos objetos do sistema.

Neste processo, foram consideradas funcionalidades como *triggers*, índices,*views* e *procedures*, que, apesar de apresentarem diferenças significativas na sua implementação consoante o sistema de gestão alvo, foram adaptadas para manter a coerência funcional entre os modelos.

Importa destacar que a migração foi realizada com o intuito de otimizar a estrutura de dados e o seu acesso conforme as características específicas de cada paradigma. Assim, a modelação adotada em MongoDB e em Neo4j teve em consideração as melhores práticas associadas a cada tecnologia, podendo, por isso, não refletir uma correspondência direta, entidade a entidade, com o modelo relacional original.

A representação dos dados em cada paradigma não é, portanto, única nem definitiva. Existem várias formas válidas de estruturar a informação dependendo dos objetivos e requisitos do sistema em causa. A abordagem aqui adotada visa alcançar um equilíbrio entre fidelidade à estrutura original e eficiência na utilização dos recursos e das capacidades oferecidas por cada SGBD.

Esta etapa não só permite demonstrar a viabilidade da migração entre paradigmas distintos, como também oferece uma base concreta para análise comparativa posterior, em termos de desempenho, complexidade de implementação e expressividade nas operações de consulta.

3.1) MongoDB

Na transposição do modelo relacional para o paradigma documental, a estrutura de dados foi organizada em três coleções principais: *Books*, *Customers* e *Orders*. Esta divisão visa refletir as principais entidades do modelo original, ajustando, no entanto, a modelação conforme as boas práticas e características do MongoDB.

No caso da coleção *Books*, os atributos existentes na tabela relacional mantiveram-se, sendo incorporadas diretamente no documento informações que, anteriormente, estavam normalizadas em relações do tipo 1:N ou N:N. Assim, a *linguagem* e a *editora* passaram a estar embebidas dentro do próprio documento do livro, enquanto os autores, anteriormente representados recorrendo a uma tabela associativa devido à relação muitos-para-muitos, foram integrados como uma lista de objetos dentro do campo *authors*. Esta decisão simplifica a estrutura e as respetivas consultas, ainda que introduza alguma redundância, uma vez que a mesma linguagem, editora ou autor poderá surgir em múltiplos documentos distintos.

A coleção *Customers* segue uma lógica semelhante. Os campos simples, como o primeiro e o último nome, foram mantidos como atributos diretos. Contudo, os *endereços* passaram a ser representados como uma lista de objetos embutidos no documento do cliente, refletindo a relação N:N existente no modelo relacional entre clientes e endereços. Cada endereço inclui, ainda, o respetivo *status*, integrando diretamente a informação necessária para interpretação do estado do cliente relativo ao local específico. Para facilitar a agregação e rastreio de atividade, foi também incluída uma lista com os identificadores das encomendas associadas a cada cliente, contendo apenas os códigos identificadores, e não os documentos completos.

Por fim, na coleção *Orders*, manteve-se a representação direta dos atributos simples, como a data da encomenda. O método de entrega foi incorporado como um objeto embebido, refletindo

a sua natureza descritiva e uso específico por encomenda. Foi ainda adicionada uma lista de *histórico de estados*, que regista todas as mudanças de estado da encomenda, e uma lista de *linhas de compra*, onde cada elemento indica o livro encomendado e o respetivo preço. De forma a evitar a duplicação desnecessária de dados, os livros são aqui referenciados apenas através do seu identificador.

Importa salientar que os identificadores primários mantiveram-se consistentes com os usados no sistema relacional. Embora o MongoDB gere automaticamente identificadores únicos (*_id*) em formato ObjectId, optou-se por preservar os IDs originais para garantir coerência entre coleções e facilitar a rastreabilidade entre modelos.

Abaixo apresentam-se exemplos representativos de documentos reais retirados de cada uma das coleções:


```
{
  "_id": 1,
  "title": "The World's First Love: Mary  Mother of God",
  "isbn13": {
    "$numberLong": "8987059752"
  },
  "language": {
    "_id": 2,
    "code": "en-US",
    "name": "United States English"
  },
  "publisher": {
    "_id": 1010,
    "name": "Ignatius Press"
  },
  "authors": [
    {
      "_id": 2778,
      "name": "Fulton J. Sheen"
    }
  ]
}
```


Listagem 1: Exemplo de Documento Retirado da Coleção *Books*


```
    {
  "_id": 1,
  "first_name": "Ursola",
  "last_name": "Purdy",
  "email": "upurdy0@cdbaby.com",
  "addresses": [
    {
      "_id": 707,
      "status": {
        "_id": 1,
        "status": "Active"
      },
      "street_number": 56061,
      "street_name": "Fairfield Court",
      "city": "Horokhiv",
```

```
      "country": {
        "_id": 213,
        "name": "Ukraine"
      }
    },
    {
      "_id": 721,
      "status": {
        "_id": 1,
        "status": "Active"
      },
      "street_number": 3,
      "street_name": "Laurel Circle",
      "city": "Mimbaan Timur",
      "country": {
        "_id": 92,
        "name": "Indonesia"
      }
    },
    {
      "_id": 973,
      "status": {
        "_id": 1,
        "status": "Active"
      },
      "street_number": 19539,
      "street_name": "Bonner Avenue",
      "city": "Fort Lauderdale",
      "country": {
        "_id": 217,
        "name": "United States of America"
      }
    }
  ],
  "orders": [
    86,
    1793,
    3969,
  ]
}
```


Listagem 2: Exemplo de Documento Retirado da Coleção *Customers*

```
    {
  "_id": 265,
  "order_date": "2025-03-11",
  "method": {
    "_id": 2,
    "name": "Priority",
    "cost": 8.9
  },
  "address": {
    "_id": 717,
    "street_number": 0,
    "street_name": "Laurel Pass",
    "city": "Drawsko Pomorskie",
    "country": {
      "_id": 163,
      "name": "Poland"
    }
  },
  "history": [
    {
      "_id": 1,
      "status": {
        "_id": 1,
        "date": "2025-03-11",
        "status": "Order Received"
      }
    },
    {
      "_id": 11429,
      "status": {
        "_id": 2,
        "date": "2025-03-12",
        "status": "Pending Delivery"
      }
    },
    {
      "_id": 16599,
      "status": {
        "_id": 3,
        "date": "2025-03-12",
        "status": "Delivery In Progress"
      }
    }
  ],
  "lines": [
    {
      "_id": 7466,
      "book_id": 2586,
      "price": 1.12
    },
  ]
}
```


Listagem 3: Exemplo de Documento Retirado da Coleção *Orders*

3.1.1) Índices

A primeira estrutura convertida para MongoDB foram os índices, que estavam previamente definidos no sistema relacional, para preservar o propósito e funcionalidade destas estruturas dentro do novo paradigma documental. Esta decisão foi tomada tendo em conta que ainda não existem *queries* de acesso totalmente definidas que justifiquem a otimização dos índices, pelo que o foco foi garantir que os campos que, no modelo relacional, justificavam a criação de índices por questões de pesquisa, unicidade ou filtragem, continuassem a ser otimizados igualmente no novo sistema.

A transposição seguiu uma lógica direta, replicando os índices definidos em SQL nas coleções correspondentes do MongoDB. Assim, o índice originalmente criado sobre o campo *title* da tabela de livros foi agora criado sobre o campo *title* na coleção books. Da mesma forma, o campo *email*, que já era alvo de indexação na tabela *customer*, foi indexado na coleção *customers*. Também o campo *order_date*, que no modelo relacional se encontrava indexado na tabela de encomendas, foi alvo de indexação na coleção *orders*.

No caso do campo *country_id*, foi necessário adaptar a indexação à estrutura aninhada dos documentos em MongoDB. Considerando que os endereços se encontram embutidos dentro dos documentos dos clientes e das encomendas, foi necessário aplicar índices sobre os caminhos completos *addresses.country._id* na coleção *customers* e *address.country._id* na coleção *orders*.

A criação dos índices foi realizada com recurso ao módulo *pymongo*, conforme exemplificado no seguinte trecho de código:

```
db["books"].create_index([("title", pymongo.ASCENDING)])
db["customers"].create_index([("email", pymongo.ASCENDING)])
db["orders"].create_index([("order_date", pymongo.ASCENDING)])
db["customers"].create_index([("addresses.country._id", pymongo.ASCENDING)])
db["orders"].create_index([("address.country._id", pymongo.ASCENDING)])
```

Listagem 4: Criação dos Índices

3.1.2) Procedures

Segundo a documentação oficial do MongoDB [1], a funcionalidade que mais se aproxima a um *stored procedure* no paradigma documental corresponde às *Atlas Functions*. Estas permitem encapsular lógica de negócio reutilizável, escrita em JavaScript, diretamente no ambiente da base de dados, à semelhança dos *procedures* no modelo relacional.

```
exports = async function (orderId, statusId) {
  function getStatusName(statusId) {
    const statusNames = {
      1: "Order Received",
      2: "Pending Delivery",
      3: "Delivery In Progress",
      4: "Delivered",
      5: "Cancelled",
      6: "Returned",
    };
    return statusNames[statusId] || "Unknown Status";
  }
```

```
  try {
    // Access the default database and collection
    const serviceName = "NoSQL";
    const database = "bookstore";
    const collection =
context.services.get(serviceName).db(database).collection("orders");

    // Find the order document
    const order = await collection.findOne({ _id: orderId });

    if (!order) {
      console.log(`Order ${orderId} not found`);
      return { success: false, message: `Order ${orderId} not found` };
    }

    // Get the current status
    let currentStatus = null;
    if (order.history && order.history.length > 0) {
      currentStatus = order.history[order.history.length - 1].status._id;
    }

    // If status is different and valid, update it
    if (currentStatus !== statusId && statusId >= 1 && statusId <= 6) {
      // Get the next history ID
      let nextHistoryId = 1;
      if (order.history && order.history.length > 0) {
        nextHistoryId = Math.max(...order.history.map((h) => h._id)) + 1;
      }

      // Create new status entry
      const newStatus = {
        _id: nextHistoryId,
        status: {
          _id: statusId,
          date: new Date().toISOString().split("T")[0], // YYYY-MM-DD format
          status: getStatusName(statusId),
        },
      };

      // Update the document
      const result = await collection.updateOne(
        { _id: orderId },
        { $push: { history: newStatus } },
      );

      if (result.modifiedCount > 0) {
        console.log(`Status updated successfully for order ${orderId}`);
        return {
          success: true,
          message: `Status updated successfully for order ${orderId}`,
          newStatus: getStatusName(statusId),
        };
      } else {
        console.log(`Failed to update status for order ${orderId}`);
```

```
        return {
          success: false,
          message: `Failed to update status for order ${orderId}`,
        };
      }
    } else {
      const message =
        currentStatus === statusId
          ? `Order ${orderId} already has status ${statusId}`
          : `Invalid status ID ${statusId}`;
      console.log(message);
      return { success: false, message: message };
    }
  } catch (error) {
    console.error(`Error updating order status: ${error.message}`);
    return {
      success: false,
      message: `Error updating order status: ${error.message}`,
    };
  }
};
```

Listagem 5: Implementação do procedure update_order_status

Este *procedure* tem como objetivo permitir a atualização controlada do campo *history* presente nos documentos da coleção *orders*, adicionando um novo estado sempre que tal se justificar. Esta função poderá ser chamada no serviço *Atlas* quando um serviço ou administrador precisar de atualizar um estatuto de uma encomenda.

3.1.3) Views

Tal como acontece no paradigma relacional, também o MongoDB disponibiliza a funcionalidade de criação de *views*, permitindo encapsular lógica de agregação ou transformação de dados de forma reutilizável e acessível, tal como uma coleção normal. No contexto deste projeto, procurou-se replicar duas das *views* definidas originalmente em SQL, demonstrando que, mesmo em paradigmas distintos, é possível atingir resultados equivalentes com abordagens adaptadas às características do sistema.

A solução adotada em MongoDB recorre a *pipelines* de agregação, que oferecem uma forma declarativa e modular de transformar documentos. Estas possibilitam operações como *joins* implícitos, filtragens, transformações condicionais, entre outras, tudo num formato fácil de programar e interpretar.

A primeira *view* replicada corresponde à *book_with_authors*, que no sistema relacional realiza um *join* entre as tabelas *book*, *book_author* e *author*, para apresentar, para cada livro, o seu título e os respetivos autores. No MongoDB, como os autores foram incorporados diretamente nos documentos *books* como uma lista de objetos, bastou utilizar a operação *$unwind* para expandir os autores num documento por linha, seguido de um ajuste de campos com *$addFields* e remoção dos atributos irrelevantes com *$unset*. A *view* criada fornece assim uma listagem compatível com a do modelo relacional.

A segunda *view*, *orders_with_status*, pretende mostrar, para cada encomenda, o seu estado mais recente. Esta *view* envolve uma lógica um pouco mais complexa, pois no MongoDB o histórico de estados encontra-se embebido como uma lista de objetos dentro de cada *order*. Para

obter o estado mais recente, foi necessário recorrer à operação $reduce dentro de *$addFields*, comparando datas e identificadores para encontrar o estado cronologicamente mais atual. Após esta operação, *$project* foi usado para formatar a saída com os campos desejados.


```
{
  "BOOK_ID": 1,
  "TITLE": "The World's First Love: Mary  Mother of God",
  "AUTHOR_NAME": "Fulton J. Sheen"
}
```


Listagem 6: Exemplo de documento da *view book_with_authors*


```
{
  "_id": 265,
  "ORDER_ID": 265,
  "ORDER_DATE": "2025-03-11",
  "STATUS_VALUE": "Delivery In Progress"
}
```


Listagem 7: Exemplo de documento da *view orders_with_status*

3.1.4) Triggers

A funcionalidade de triggers em MongoDB está disponível através do MongoDB Atlas, sendo suportada inclusive no plano gratuito. Os triggers permitem definir ações que executam automaticamente em resposta a operações específicas numa coleção. São escritos em JavaScript e podem ser associados a eventos de tipo insert, update, replace ou delete. No âmbito deste projeto, foram implementados três triggers distintos que visam manter a coerência funcional previamente existente no sistema relacional. Cada um destes triggers replica uma lógica de negócio importante, ajustada ao paradigma documental.

O trigger "validate_email" visa validar o campo de e-mail sempre que ocorre um insert, update ou replace na coleção customers. A lógica foi adaptada da implementação SQL e recorre a uma expressão regular para verificar se o valor do campo email cumpre um formato padrão válido. A implementação atual adota uma abordagem reativa: permite que a operação inicial seja executada e posteriormente valida o resultado. Caso o campo email não respeite a estrutura esperada, o trigger toma medidas corretivas adequadas ao tipo de operação. Para operações de insert, o documento inválido é eliminado da coleção. Para operações de update ou replace, o documento é revertido para o seu estado anterior utilizando o campo fullDocumentBeforeChange. Se não existir estado anterior disponível, o documento é eliminado como medida de segurança.

```
exports = async function(changeEvent) {
  const docId = changeEvent.documentKey._id;
  const serviceName = "NoSQL";
  const database = "bookstore";
  const collection =
context.services.get(serviceName).db(database).collection(changeEvent.ns.coll);

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

    if (changeEvent.operationType === "insert") {
      const newCustomer = changeEvent.fullDocument;
      if (newCustomer.email && !emailRegex.test(newCustomer.email)) {
        // Delete invalid insert
        await collection.deleteOne({_id: docId});
        console.log(`Deleted customer ${docId} due to invalid email:
${newCustomer.email}`);
      }
    }
    else if (changeEvent.operationType === "update" || changeEvent.operationType
=== "replace") {
      const currentDocument = changeEvent.fullDocument;
      if (currentDocument?.email && !emailRegex.test(currentDocument.email)) {
        // Email is invalid, revert to previous state
        if (changeEvent.fullDocumentBeforeChange) {
          await collection.replaceOne(
            {_id: docId},
            changeEvent.fullDocumentBeforeChange
          );
          console.log(`Reverted customer ${docId} due to invalid email:
${currentDocument.email}`);
        } else {
          // Delete the document if we cant revert
          await collection.deleteOne({_id: docId});
          console.log(`Deleted customer ${docId} due to invalid email and no
previous state available`);
        }
      }
    }
  } catch(err) {
    console.log("Error in trigger:", err.message);
  }
};
```

Listagem 8: Trigger validate_email

O trigger "insert_order_history" é ativado exclusivamente em operações de insert na coleção orders e serve para inicializar automaticamente o histórico de estados de uma encomenda. Sempre que uma nova encomenda é criada, o trigger adiciona automaticamente o primeiro estado "Order Received" ao array history.

A implementação utiliza duas operações sequenciais: primeiro verifica se o campo history existe no documento e, caso não exista, cria um array vazio. De seguida, adiciona o novo elemento de histórico com a operação `$push`. Cada elemento do histórico contém um identificador único, uma estrutura de estado com ID, data atual e o status inicial.

```
exports = async function(changeEvent) {
  const docId = changeEvent.documentKey._id;
  const serviceName = "NoSQL";
  const database = "bookstore";
  const collection =
context.services.get(serviceName).db(database).collection(changeEvent.ns.coll);

  try {
    if (changeEvent.operationType === "insert") {
      const currentDate = new Date().toISOString().split('T')[0];
      const historyElement = {
        _id: new BSON.ObjectId(),
        status:{
        _id:1,
        date: currentDate,
        status: "Order Received"
        }
      };

      // First, ensure the history field exists
      await collection.updateOne(
        { _id: docId, history: { $exists: false } },
        { $set: { history: [] } }
      );

      // Then push the new element
      await collection.updateOne(
        { _id: docId },
        { $push: { history: historyElement } }
      );
    }
  } catch (err) {
    console.log("Error performing MongoDB write:", err.message);
  }
};
```

Listagem 9: Trigger insert_order_history

O terceiro trigger, "prevent_book_deletion", serve para evitar a eliminação de livros que tenham sido encomendados, garantindo assim a integridade referencial da informação, mesmo num sistema sem foreign keys. Antes de permitir a eliminação definitiva de um livro da coleção books, é executada uma pipeline de agregação para verificar se o identificador do livro em questão está presente em alguma das lines de documentos da coleção orders.

A pipeline utiliza as operações `$unwind` para expandir o array de linhas e `$match` para filtrar documentos que contenham referências ao livro. Se for encontrada uma correspondência, o trigger previne a eliminação restaurando o documento através da inserção do fullDocumentBeforeChange, mantendo assim a integridade dos dados.

```
exports = async function(changeEvent) {
  const docId = changeEvent.documentKey._id;
  const serviceName = "NoSQL";
  const database = "bookstore";
  const collectionName = changeEvent.ns.coll;
```

```
  const collection =
context.services.get(serviceName).db(database).collection(collectionName);

  // This is the collection containing lines with references to books
  const ordersCollection =
context.services.get(serviceName).db(database).collection("orders");

  try {
    if (changeEvent.operationType === "delete") {
      const pipeline = [
        { $unwind: "$lines" },
        { $match: { "lines.book_id": docId } }
      ];

      const result = await ordersCollection.aggregate(pipeline).toArray();

      if (result.length > 0) {
        await collection.insertOne(changeEvent.fullDocumentBeforeChange)
        console.log(`Deletion prevented: Book ${docId} is referenced in
'orders'.`);
        return;
      }
    }
  } catch (err) {
    console.log("Error performing MongoDB operation:", err.message);
  }
};
```

Listagem 10: Trigger prevent_book_deletion

3.2) Neo4J

Para representar o modelo de dados no paradigma de grafos, utilizando o sistema Neo4j, definimos que, através de nós, representaríamos as entidades mais relevantes, e que as arestas representam as relações entre essas entidades. Com base na estrutura relacional original, foi desenhado um grafo onde cada tipo de entidade relevante no domínio foi transformado num nó. Assim, os *customers*, os *adderesses*, as *orders*, os *books*, os *authors* e os *status* das encomendas surgem como nós distintos, ligados entre si via relações que mantêm o significado semântico original. Os clientes estão ligados aos endereços através da relação *LIVES*, às encomendas que efetuam através da relação *PLACED*, e as encomendas por sua vez estão ligadas ao endereço de entrega através da relação *TO* e ao diferente estados pela relação *HAS_STATUS*. As encomendas também estão ligadas aos livros que incluem por meio da relação *CONTAINS*, enquanto os livros ligam se aos seus autores através da relação *WRITTENBY*. Este modelo permite não só preservar a informação original como também as mesmas capacidades de consulta, como, por exemplo, encontrar todos os livros encomendados por clientes de uma determinada localização, seguir o histórico de estados de uma encomenda, ou mapear a relação entre autores e o número de vezes que os seus livros foram vendidos. Ao contrário do modelo relacional, onde a profundidade de consulta e o número de junções pode comprometer a performance, o paradigma em grafo permite atravessar rapidamente os caminhos entre entidades, sendo por isso mais expressivo e mais eficiente em casos como este. A figura seguinte ilustra graficamente o modelo concebido para a plataforma Neo4j, representando de forma simplificada a estrutura do grafo resultante da migração dos dados da livraria para este novo paradigma.

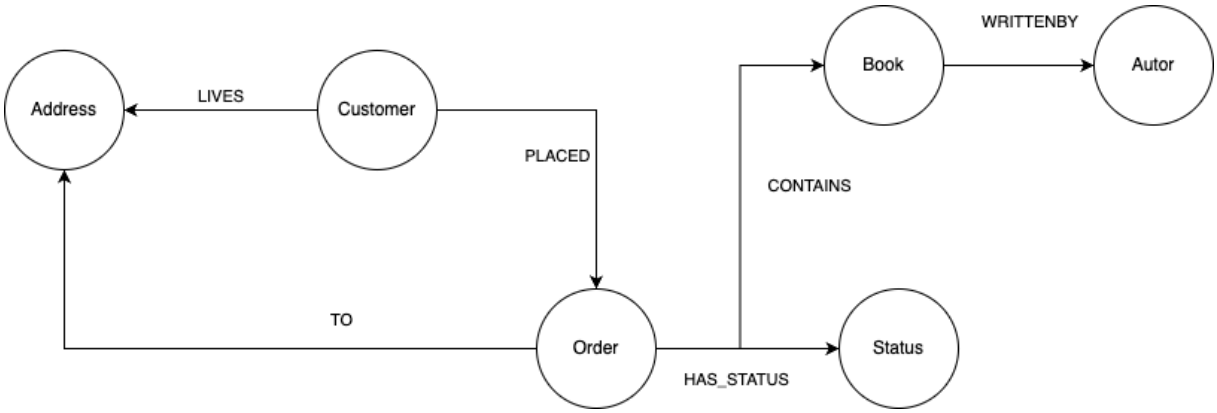

Figura 2: Esquema do modelo de dados utilizado para o Neo4j

3.2.1) Índices

À semelhança de sistemas relacionais e documentais, o Neo4j também permite a criação de índices, o que contribui para a otimização do desempenho em operações de leitura, nomeadamente nas pesquisas por propriedades específicas de nós.

Na linguagem Python, com recurso ao *driver* oficial do Neo4j, os mesmos índices foram implementados da seguinte forma:

```
  tx.run("CREATE INDEX idx_book_title IF NOT EXISTS FOR (n:Book) ON (n.title)")
  tx.run("CREATE INDEX idx_customer_email IF NOT EXISTS FOR (n:Customer) ON
(n.email)")
  tx.run("CREATE INDEX idx_order_date IF NOT EXISTS FOR (n:Order) ON
(n.order_date)")
  tx.run("CREATE INDEX idx_address_country IF NOT EXISTS FOR (n:Address) ON
(n.country)")
```

Listagem 11: Criação de índices Neo4j

O Neo4j permite que diversos tipos de índices sejam criados. Para esta transposição, não é claro qual o tipo mais ajustado, dado que existe pouca informação sobre o tipo de operações na base de dados que devem ser otimizados. Desta forma, optamos por utilizar a configuração *default*, visto que são especialmente relevantes para consultas com operadores de comparação.

3.2.2) Procedures

O motor Neo4j permite a criação de *procedures* graças à biblioteca APOC (Awesome Procedures on Cypher) que permitem extender algumas das funcionalidades da base de dados. Desta forma, escrevemos o seguinte *procedure* segundo as funções da biblioteca referida.

```
CALL apoc.custom.declareProcedure(
  'update_order_status(p_order_id :: INTEGER, p_status_id :: INTEGER) :: (out ::
STRING)',
  '
    MATCH (o:Order {id: $p_order_id})
    MATCH (s:Status {id: $p_status_id})
    OPTIONAL MATCH (o)-[h:HAS_STATUS]->(s)
    WITH o, s, h, $p_order_id AS orderId, $p_status_id AS statusId

    CALL apoc.do.when(
      h IS NULL,
```

```
      "
        MERGE (o)-[rel:HAS_STATUS]->(s)
        SET rel.status_date = date()
        RETURN \'Status atualizado com sucesso para o pedido \' +
toString(orderId) AS out
      ",
      "
        RETURN \'O status do pedido \' + toString(orderId) + \' já está como \' +
toString(statusId) AS out
      ",
      {o: o, s: s, orderId: orderId, statusId: statusId}
    ) YIELD value

    RETURN value.out AS out
  ',
  'write'
)
```

Listagem 12: Implementação do procedure update_order_status

Este *procedure* em Listagem 12 recorre, primeiramente, à função "declareProcedure", que habilita a introdução de funções, seguida da assinatura do mesmo, descrevendo os argumentos de entrada assim como os de saída. A seguir, ainda recorremos à função "when" que permite determinar dois caminhos possíveis na nossa lógica: Um para quando o estado é atualizado; Um para quando o estado já existe.

3.2.3) Views

Apesar de o Neo4j não suportar diretamente o conceito de *views* como nas bases de dados relacionais, é possível obter comportamentos equivalentes através da definição de *queries* na linguagem Cypher, permitindo assim resultados semelhantes aos de *views* tradicionais. No contexto desta migração, foram replicadas as duas *views* originalmente presentes no modelo relacional, agora expressas em Cypher.

A view book_with_authors procura apresentar, para cada livro, o seu título e os nomes dos autores correspondentes. No modelo em grafos, a relação entre livros e autores é representada via uma aresta que liga nós do tipo *Author* aos nós *Book*.Esta querie realiza uma correspondência dos autores que escreveram livros, retornando o identificador e o título do livro com o nome de cada autor.

```
{
    "keys": [
      "BOOK_ID",
      "TITLE",
      "AUTHOR_NAME"
    ],
    "length": 3,
    "_fields": [
      {
        "low": 10539,
        "high": 0
      },
      "Baseball: a Literary Anthology",
      "A. Bartlett Giamatti"
    ],
    "_fieldLookup": {
      "BOOK_ID": 0,
      "TITLE": 1,
      "AUTHOR_NAME": 2
    }
  }
```


Listagem 13: Exemplo do resultado da querie book_with_authors Neo4j

A *view* orders_with_status pretende listar, para cada encomenda, o seu estado mais recente. No modelo de grafos, o histórico de estados é representado por nós *Status* ligados aos nós *Order* através da relação *HAS_STATUS*, acompanhada do atributo de *date* que indica o momento em que o estado foi atribuído.


```
{
    "keys": [
      "ORDER_ID",
      "ORDER_DATE",
      "STATUS_VALUE"
    ],
    "length": 3,
    "_fields": [
      {
        "low": 671,
        "high": 0
      },
      "2025-03-31",
      "Cancelled"
    ],
    "_fieldLookup": {
      "ORDER_ID": 0,
      "ORDER_DATE": 1,
      "STATUS_VALUE": 2
    }
  }
```


Listagem 14: Exemplo do resultado da querie orders_with_status Neo4j

3.2.4) Triggers

A biblioteca APOC permite ainda escrever *triggers*, novamente sobre a linguagem Cypher, permitindo a qualquer instância da base de dados suportar este tipo de comportamento.

O trigger "validate_email" foi implementado para garantir a integridade dos dados de email na base de dados, validando automaticamente o formato dos endereços de email sempre que são inseridos ou atualizados.

```
CALL apoc.trigger.install(
  'neo4j',
  'validate_email',
  '
   UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties, "email") AS prop
   CALL apoc.util.validate(
     NOT prop.new =~ "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\\\.[A-Za-z]{2,}$",
     "Invalid email format. Got %s",
     [prop.new]
   ) RETURN null
   ',
  {phase: 'before'}
)
```

Listagem 15: Trigger validate_email

Este trigger utiliza uma expressão regular para validar o formato dos endereços de email sempre que a propriedade "email" é atribuída ou modificada. Funciona na fase "before", impedindo que emails com formato inválido persistam na base de dados através de uma exceção com uma mensagem descritiva.

O *trigger* "insert_order_history" serve se novamente da biblioteca APOC e cria a relação com o primeiro estado assim que uma encomenda é adicionada.

```
CALL apoc.trigger.install(
   'neo4j',
   'insert_order_history',
   '
   UNWIND $createdNodes AS node
   WITH node
   WHERE "Order" IN labels(node)
   Merge (node)-[rel:HAS_STATUS {status_date:date()}]->(:Status {id:1})
  ',
    {phase:"before"}
)
```

Listagem 16: Trigger insert_order_history

É importante referir que este comportamento funciona desta forma, pois sabemos antecipadamente que para todas as encomdendas adicionadas, será sempre atribuido como primeiro estado o estado com *id* igual a 1.

O *trigger* "prevent_book_deletion" foi também implementado sobre a linguagem Cypher. Contudo, devido à natureza da nossa implementação, o *trigger* nunca será "ativo".

```
   CALL apoc.trigger.install(
   'neo4j',
   'prevent_book_deletion',
   '
   UNWIND $deletedNodes AS node
   WITH node
   WHERE "Book" IN labels(node)
   OPTIONAL MATCH (:Order)-[c:CONTAINS]->(node)
   CASE c
    WHEN IS NULL "ok"
    WHEN IS NOT NULL "Cannot delete book as it exists in orders."
   END AS result
  ',
    {phase:"before"}

)
```

Listagem 17: Trigger prevent_book_deletion

O Neo4j possui um comportamento *default* de não permitir que nós que contenham arestas apontadas na sua direção, naturalmente sobre a forma de relacionamento, não podem ser apagados [2]. Desta forma, devido à nossa abstração utilizar a ligação no sentido *Order* para *Book*, este comportamento já garantido pelo próprio motor. É ainda relevante mencionar que todos os *triggers* são apenas ativados após a verificação do próprio Neo4j, sendo que não há forma de contornar esta situação sem alterar o esquema da base de dados.

4) Resultados

4.1) Queries

4.1.1) `getTopSellingBooksByRegion`

Esta *query* foi projetada para identificar os livros mais vendidos em Portugal. A análise agrega métricas como **total de vendas por livro** e associa os resultados ao país de destino dos pedidos, fornecendo *insights* valiosos para otimização de *stock*, estratégias de *marketing* e planeamento logístico.

4.1.1.1) OracleDB – SQL

A abordagem implementada em OracleDB utiliza junções e agregação para calcular o número de vendas por livro em Portugal. A *query* combina várias tabelas para associar livros, pedidos e endereços, filtrando pelo país de destino.

```sql
SELECT
    BOOK.BOOK_ID,
    BOOK.TITLE,
    COUNTRY.COUNTRY_NAME,
    COUNT(CUST_ORDER.ORDER_ID) AS VENDAS
FROM
    BOOK
    INNER JOIN ORDER_LINE ON BOOK.BOOK_ID = ORDER_LINE.BOOK_ID
    INNER JOIN CUST_ORDER ON ORDER_LINE.ORDER_ID = CUST_ORDER.ORDER_ID
    INNER JOIN ADDRESS ON CUST_ORDER.DEST_ADDRESS_ID = ADDRESS.ADDRESS_ID
    INNER JOIN COUNTRY ON ADDRESS.COUNTRY_ID = COUNTRY.COUNTRY_ID
WHERE
    COUNTRY.COUNTRY_NAME = 'Portugal'
GROUP BY
    BOOK.BOOK_ID, BOOK.TITLE, COUNTRY.COUNTRY_NAME
ORDER BY
    VENDAS DESC;
```

A eficácia desta consulta reside nos seguintes aspectos:

1. **Filtragem por país:** A cláusula `WHERE COUNTRY.COUNTRY_NAME = 'Portugal'` restringe os resultados a pedidos enviados para Portugal, garantindo foco no mercado-alvo.
2. **Junções estruturadas:** As operações `INNER JOIN` conectam as tabelas `BOOK`, `ORDER_LINE`, `CUST_ORDER`, `ADDRESS` e `COUNTRY`, refletindo a relação entre entidades no modelo relacional e permitindo a associação precisa de livros a endereços de entrega.
3. **Agregação de vendas:** A função `COUNT(CUST_ORDER.ORDER_ID)` calcula o número de pedidos por livro, fornecendo uma métrica clara de popularidade.
4. **Resultados organizados:** A cláusula `GROUP BY` agrupa os resultados por `BOOK_ID`, `TITLE` e `COUNTRY_NAME`, enquanto `ORDER BY VENDAS DESC` ordena os livros por número de vendas, facilitando a identificação dos mais vendidos.
5. **Estrutura relacional otimizada:** A consulta aproveita a natureza tabular do OracleDB, produzindo resultados estruturados ideais para relatórios ou integração com ferramentas analíticas (*e.g.*, PowerBI).

Esta *query* reflete a lógica pretendida, utilizando junções e agregações para fornecer uma visão clara dos livros mais vendidos em Portugal.

4.1.1.2) MongoDB – Pipeline de agregação

A implementação em MongoDB utiliza um pipeline de agregação com 7 estágios para processar os dados de forma eficiente, aproveitando a estrutura de documentos natural do Mongo.

```
[
  // 1. Filtrar pedidos com endereço em Portugal
  { $match: { "address.country.name": "Portugal" } },

  // 2. Decompor as linhas de pedido
  { $unwind: { path: "$lines" } },

  // 3. Associar detalhes dos livros
  { $lookup: {
      from: "books",
      localField: "lines.book_id",
      foreignField: "_id",
      as: "TITLE"
  } },

  // 4. Extrair título do livro
  { $addFields: {
      title: { $arrayElemAt: ["$TITLE.title", 0] }
  } },

  // 5. Agrupar por livro e país
  { $group: {
      _id: {
          country: "$address.country",
          book_id: "$lines.book_id",
          title: "$title"
      },
      Vendas: { $count: {} }
  } },

  // 6. Formatar os resultados
  { $project: {
      _id: 0,
      BOOK_ID: "$_id.book_id",
      COUNTRY_NAME: "$_id.country.name",
      TITLE: "$_id.title",
      Vendas: "$Vendas"
  } },

  // 7. Ordenar por número de vendas
  { $sort: { Vendas: -1 } }
]
```

A **eficácia** desta abordagem deve-se a:

1. **Filtragem inicial eficiente:** O estágio `$match` seleciona apenas pedidos com endereço em Portugal, reduzindo o conjunto de dados processado nos estágios subsequentes.

2. **Decomposição de linhas:** O `$unwind` expande o array `lines`, permitindo processar cada linha de pedido individualmente.
3. **Integração de dados:** O `$lookup` associa os detalhes dos livros da coleção `books`, enquanto `$addFields` extrai o título do livro, simplificando a estrutura dos dados.
4. **Agregação precisa:** O `$group` calcula o número de vendas por livro e país, utilizando `$count` para contabilizar as ocorrências.
5. **Formatação e ordenação:** Os estágios `$project` e `$sort` eliminam campos desnecessários, formatam o resultado de forma clara e ordenam os livros por vendas decrescentes, facilitando análises.

A *pipeline* demonstra a flexibilidade do MongoDB para processar dados complexos, oferecendo resultados otimizados para relatórios ou tomadas de decisão.

4.1.1.3) Neo4j – *Cypher Query*

A implementação em Neo4j utiliza a linguagem Cypher para explorar as relações entre entidades no modelo de grafo, onde pedidos, livros e endereços são nós conectados por relacionamentos.

```
MATCH (o:Order)-[c:CONTAINS]->(b:Book)
WITH o, c, b
MATCH (o:Order)-[:TO]->(a:Address {country: "Portugal"})
RETURN b.id, b.title, a.country, count(c) AS Vendas
ORDER BY Vendas DESC
```

A eficácia desta abordagem destaca-se pelos seguintes aspectos:

1. **Navegação natural por grafos:** A query utiliza `MATCH` para explorar os relacionamentos `CONTAINS` (de pedidos para livros) e `TO` (de pedidos para endereços), refletindo a semântica do domínio.
2. **Filtragem por país:** A condição `a:Address {country: "Portugal"}` filtra pedidos enviados para Portugal, aproveitando a estrutura do grafo para acesso direto aos nós relevantes.
3. **Cálculo de vendas:** A função `count(c)` agrega o número de relacionamentos `CONTAINS`, representando o total de vendas por livro.
4. **Resultados ordenados:** O `ORDER BY Vendas DESC` organiza os resultados por número de vendas, facilitando a identificação dos livros mais populares.
5. **Simplicidade e clareza:** A query é concisa, aproveitando a expressividade do Cypher para combinar filtragem, agregação e ordenação em poucos passos.

Esta abordagem em Neo4j é ideal para análises baseadas em relações, oferecendo desempenho eficiente em cenários com dados altamente conectados.

4.1.2) `getTopEarningAuthors`

Esta *query* foi projetada para identificar os autores que geraram maior faturação na livraria, considerando o valor total das vendas dos seus livros dividido pelo número de autores por livro. A presente análise permite identificar a contribuição financeira de cada autor, apoiando decisões como negociações de royalties, promoções de marketing ou parcerias editoriais.

4.1.2.1) OracleDB – SQL

A abordagem em OracleDB utiliza uma consulta SQL com subconsultas e junções para calcular os ganhos por autor, dividindo a faturação de cada livro pelo número de autores associados.

```
SELECT
    AUTHOR.AUTHOR_ID,
    AUTHOR.AUTHOR_NAME,
    ROUND(SUM(BOOKSALES.TOTAL/BOOKSALES.NUM_AUTHORS), 2) AS EARNINGS
FROM (
    SELECT
        BOOK.BOOK_ID,
        SUM(ORDER_LINE.PRICE) as TOTAL,
        (SELECT COUNT(*) FROM BOOK_AUTHOR WHERE BOOK_AUTHOR.BOOK_ID =
BOOK.BOOK_ID) AS NUM_AUTHORS
    FROM BOOK
    INNER JOIN ORDER_LINE ON BOOK.BOOK_ID = ORDER_LINE.BOOK_ID
    GROUP BY BOOK.BOOK_ID
) BOOKSALES
INNER JOIN BOOK_AUTHOR ON BOOKSALES.BOOK_ID = BOOK_AUTHOR.BOOK_ID
INNER JOIN AUTHOR ON BOOK_AUTHOR.AUTHOR_ID = AUTHOR.AUTHOR_ID
GROUP BY AUTHOR.AUTHOR_ID, AUTHOR.AUTHOR_NAME
ORDER BY EARNINGS DESC;
```

A eficácia desta consulta reside nos seguintes aspectos:

1. **Cálculo de faturação por livro:** A subconsulta `BOOKSALES` utiliza `SUM(ORDER_LINE.PRICE)` para somar o valor das vendas por livro, agregando dados da tabela `ORDER_LINE`.
2. **Divisão por autores:** A subconsulta `SELECT COUNT(*) FROM BOOK_AUTHOR` determina o número de autores por livro, permitindo dividir a faturação igualmente entre eles.
3. **Junções estruturadas:** As operações `INNER JOIN` conectam `BOOKSALES`, `BOOK_AUTHOR` e `AUTHOR`, associando a faturação de cada livro aos respectivos autores.
4. **Agregação por autor:** A cláusula `GROUP BY AUTHOR.AUTHOR_ID, AUTHOR.AUTHOR_NAME` soma os ganhos proporcionais de cada autor, enquanto `ROUND(..., 2)` garante precisão financeira.
5. **Ordenação clara:** O `ORDER BY EARNINGS DESC` organiza os resultados por faturação decrescente, destacando os autores mais lucrativos.

Esta consulta SQL reflete a lógica pretendida, aproveitando a estrutura relacional do OracleDB para cálculos precisos e resultados estruturados, ideais para relatórios financeiros.

4.1.2.2) MongoDB – Pipeline de agregação

A implementação em MongoDB utiliza um pipeline de agregação com 9 estágios para processar os dados, aproveitando a flexibilidade do modelo de documentos.

```
[
  // 1. Decompor as linhas de pedido
  { $unwind: { path: "$lines" } },

  // 2. Associar detalhes dos livros
  { $lookup: {
      from: "books",
      localField: "lines.book_id",
      foreignField: "_id",
      as: "book"
  } },

  // 3. Extrair objeto do livro
  { $addFields: {
      oneBook: { $arrayElemAt: ["$book", 0] }
  } },

  // 4. Calcular número de autores
  { $addFields: {
      num_authors: { $size: "$oneBook.authors" }
  } },

  // 5. Agrupar por livro
  { $group: {
      _id: "$lines.book_id",
      totalSales: { $sum: "$lines.price" },
      num_authors: { $first: "$num_authors" },
      authors: { $first: "$oneBook.authors" }
  } },

  // 6. Decompor lista de autores
  { $unwind: { path: "$authors" } },

  // 7. Calcular ganhos por autor
  { $group: {
      _id: "$authors",
      EARNINGS: { $sum: { $divide: ["$totalSales", "$num_authors"] } }
  } },

  // 8. Formatar resultados
  { $project: {
      AUTHOR_ID: "$_id._id",
      AUTHOR_NAME: "$_id.name",
      _id: 0,
      EARNINGS: { $round: ["$EARNINGS", 2] }
  } },

  // 9. Ordenar por ganhos
  { $sort: { EARNINGS: -1 } }
]
```

A **eficácia** desta abordagem deve-se a:

1. **Decomposição inicial:** O `$unwind` expande o array `lines`, permitindo processar cada linha de pedido individualmente.

2. **Integração de dados:** O `$lookup` associa detalhes dos livros, enquanto `$addFields` extrai o objeto do livro e calcula o número de autores com `$size`.
3. **Agregação por livro:** O primeiro `$group` soma a faturação por livro (`$sum: "$lines.price"`) e armazena o número de autores e a lista de autores.
4. **Divisão por autor:** O segundo `$unwind` e `$group` dividem a faturação por autor, utilizando `$divide` para alocar ganhos proporcionais.
5. **Formatação e ordenação:** Os estágios `$project` e `$sort` formatam o resultado com campos relevantes e ordenam por ganhos decrescentes, facilitando análises financeiras.

O pipeline demonstra a capacidade do MongoDB de lidar com cálculos complexos em dados documentais, produzindo resultados otimizados para relatórios.

4.1.2.3) Neo4j – *Cypher Query*

A implementação em Neo4j utiliza a linguagem Cypher para explorar as relações entre livros, autores e pedidos no modelo de grafo.

```
MATCH (b:Book)-[:WRITTEN_BY]->(a:Author)
WITH b, count(a) AS num_authors
MATCH (b)<-[c:CONTAINS]-(o:Order)
WITH b, num_authors, sum(c.price) AS book_total
MATCH (b)-[:WRITTEN_BY]->(author:Author)
WITH author, book_total/num_authors AS author_earnings
RETURN
    author.id AS AUTHOR_ID,
    author.name AS AUTHOR_NAME,
    round(sum(author_earnings), 2) AS EARNINGS
ORDER BY EARNINGS DESC
```

A eficácia desta abordagem destaca-se pelos seguintes aspectos:

1. **Exploração de relacionamentos:** O `MATCH` navega pelos relacionamentos `WRITTEN_BY` (livros para autores) e `CONTAINS` (pedidos para livros), refletindo a semântica do domínio.
2. **Cálculo de autores por livro:** O primeiro `WITH` utiliza `count(a)` para determinar o número de autores por livro, essencial para divisão de ganhos.
3. **Faturação por livro:** O segundo `MATCH` e `sum(c.price)` calculam a faturação total de cada livro com base nos pedidos.
4. **Divisão por autor:** O cálculo `book_total/num_authors` aloca ganhos proporcionais a cada autor, enquanto `sum(author_earnings)` agrega os totais por autor.
5. **Resultados formatados:** O `RETURN` com `round(..., 2)` e `ORDER BY EARNINGS DESC` produz uma lista ordenada de autores por faturação, ideal para análises.

Esta query Cypher aproveita a estrutura de grafo do Neo4j para cálculos baseados em relações, oferecendo clareza e eficiência.

4.1.3) `getTopCountryForHarperSales`

Esta *query* foi projetada para identificar o país onde os livros publicados pelo grupo "Harper" registaram o maior número/volume de vendas, um dado estratégico para a livraria avaliar o desempenho de editoras específicas em diferentes mercados. A análise agrega o total de vendas por país, fornecendo insights para estratégias de distribuição, campanhas de marketing e parcerias com editoras.

4.1.3.1) OracleDB – SQL

A abordagem em OracleDB utiliza uma consulta SQL com múltiplas junções e agregação para calcular o número de vendas de livros do grupo "Harper" por país, com base nos endereços de entrega dos pedidos.

```sql
SELECT
    COUNTRY.COUNTRY_NAME,
    COUNT(ORDER_LINE.BOOK_ID) AS VENDAS
FROM BOOK
    INNER JOIN PUBLISHER ON BOOK.PUBLISHER_ID = PUBLISHER.PUBLISHER_ID
    INNER JOIN ORDER_LINE ON BOOK.BOOK_ID = ORDER_LINE.BOOK_ID
    INNER JOIN CUST_ORDER ON ORDER_LINE.ORDER_ID = CUST_ORDER.ORDER_ID
    INNER JOIN ADDRESS ON CUST_ORDER.DEST_ADDRESS_ID = ADDRESS.ADDRESS_ID
    INNER JOIN COUNTRY ON ADDRESS.COUNTRY_ID = COUNTRY.COUNTRY_ID
WHERE
    PUBLISHER.PUBLISHER_NAME LIKE '%Harper%'
GROUP BY COUNTRY.COUNTRY_NAME
ORDER BY VENDAS DESC;
```

A eficácia desta consulta reside nos seguintes aspectos:

1. **Filtragem por editora:** A cláusula `WHERE PUBLISHER.PUBLISHER_NAME LIKE '%Harper%'` restringe os resultados a livros publicados pelo grupo "Harper", garantindo foco na editora-alvo.
2. **Junções estruturadas:** As operações `INNER JOIN` conectam as tabelas `BOOK`, `PUBLISHER`, `ORDER_LINE`, `CUST_ORDER`, `ADDRESS` e `COUNTRY`, associando livros vendidos aos países de entrega dos pedidos.
3. **Agregação de vendas:** A função `COUNT(ORDER_LINE.BOOK_ID)` calcula o número de livros vendidos por país, fornecendo uma métrica clara de desempenho.
4. **Agrupamento por país:** A cláusula `GROUP BY COUNTRY.COUNTRY_NAME` organiza os resultados por país, enquanto `ORDER BY VENDAS DESC` ordena os países por número de vendas, destacando o mercado mais relevante.

Esta consulta SQL reflete a lógica pretendida, utilizando junções e agregações para fornecer uma visão clara dos mercados mais fortes para o grupo "Harper".

4.1.3.2) MongoDB – Pipeline de agregação

A implementação em MongoDB utiliza um pipeline de agregação com 6 estágios para processar os dados, aproveitando a estrutura de documentos do Mongo.

```
[
  // 1. Associar detalhes dos livros
  { $lookup: {
```

```
      from: "books",
      localField: "lines.book_id",
      foreignField: "_id",
      as: "book"
  } },

  // 2. Decompor o array de livros
  { $unwind: { path: "$book" } },

  // 3. Filtrar por editora Harper
  { $match: { "book.publisher.name": { $regex: "Harper" } } },

  // 4. Agrupar por país
  { $group: {
      _id: "$address.country.name",
      Vendas: { $count: {} }
  } },

  // 5. Ordenar por número de vendas
  { $sort: { Vendas: -1 } },

  // 6. Formatar resultados
  { $project: {
      COUNTRY_NAME: "$_id",
      Vendas: "$Vendas",
      _id: 0
  } }
]
```

A **eficácia** desta abordagem deve-se a:

1. **Integração de dados:** O `$lookup` associa detalhes dos livros da coleção `books`, permitindo acesso às informações da editora.
2. **Decomposição eficiente:** O `$unwind` expande o array `book`, preparando os dados para filtragem e agregação.
3. **Filtragem precisa:** O `$match` com `$regex: "Harper"` seleciona apenas livros do grupo "Harper", garantindo foco na editora-alvo.
4. **Agregação por país:** O `$group` calcula o número de vendas por país usando `$count`, produzindo uma métrica clara.
5. **Formatação e ordenação:** Os estágios `$project` e `$sort` formatam o resultado com campos relevantes e ordenam os países por vendas decrescentes, facilitando análises.

O pipeline demonstra a flexibilidade do MongoDB para processar dados documentais, oferecendo resultados otimizados para tomadas de decisão.

4.1.3.3) Neo4j – *Cypher Query*

A implementação em Neo4j utiliza a linguagem Cypher para explorar as relações entre livros, pedidos e endereços no modelo de grafo.

```
MATCH (b:Book)<-[:CONTAINS]-(o:Order)
WITH b, o
```

```
MATCH (o)-[:TO]->(a:Address)
WHERE b.publisher CONTAINS 'Harper'
RETURN a.country, count(*) as VENDAS
ORDER BY VENDAS DESC
```

A eficácia desta abordagem destaca-se pelos seguintes aspectos:

1. **Navegação por relacionamentos:** O `MATCH` explora os relacionamentos `CONTAINS` (de pedidos para livros) e `TO` (de pedidos para endereços), refletindo a semântica do domínio.
2. **Filtragem por editora:** A condição `WHERE b.publisher CONTAINS 'Harper'` restringe os resultados a livros do grupo "Harper", garantindo foco na análise.
3. **Cálculo de vendas:** A função `count(*)` agrega o número de livros vendidos por país, com base nos pedidos associados.
4. **Resultados ordenados:** O `ORDER BY VENDAS DESC` organiza os resultados por número de vendas, destacando o país com maior desempenho.
5. **Simplicidade e clareza:** A query é concisa, aproveitando a expressividade do Cypher para combinar filtragem, agregação e ordenação em poucos passos.

Esta abordagem em Neo4j é ideal para análises baseadas em relações, oferecendo desempenho eficiente em dados altamente conectados.

4.1.4) `getOrderMetricsByCustomer`

Esta *query* foi projetada para analisar o comportamento de compra dos clientes com morada na China, um segmento estratégico para a livraria devido ao seu elevado número de clientes na zona e potencial de crescimento. A análise calcula métricas essenciais como **valor total dos pedidos**, **quantidade de livros adquiridos** e **tempo desde a última atualização de status**, fornecendo insights valiosos para decisões comerciais e logísticas.

4.1.4.1) OracleDB – SQL

A abordagem implementada em OracleDB utiliza uma consulta SQL com *subqueries* correlacionadas e operadores condicionais para obter os resultados estratégicos definidos na definição da query. Cada linha representa um pedido de cliente com os principais indicadores calculados dinamicamente.

```sql
SELECT DISTINCT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    o.order_id,

    -- Total order value
    (SELECT SUM(ol_inner.price)
     FROM order_line ol_inner
     WHERE ol_inner.order_id = o.order_id) AS total_value,

    -- Number of items
    (SELECT COUNT(*)
     FROM order_line ol_inner
     WHERE ol_inner.order_id = o.order_id) AS total_items,

    -- Days since last status update
    CASE
        WHEN EXISTS (
            SELECT 1 FROM order_history oh_inner
            WHERE oh_inner.order_id = o.order_id
        ) THEN
            ROUND(CURRENT_DATE - (
                SELECT MAX(oh_inner.status_date)
                FROM order_history oh_inner
                WHERE oh_inner.order_id = o.order_id
            ), 0)
        ELSE NULL
    END AS days_since_last_status

FROM
    customer c
JOIN
    cust_order o ON o.customer_id = c.customer_id

-- Filter by country
WHERE c.customer_id IN (
    SELECT ca.customer_id
    FROM customer_address ca
```

```
    JOIN address a ON a.address_id = ca.address_id
    JOIN country co ON a.country_id = co.country_id
    WHERE co.country_id = 42
)

ORDER BY
    c.customer_id, o.order_id
```

A eficácia desta consulta reside nos seguintes aspetos:

1. **Filtragem por país:** Os dados são filtrados para incluir apenas clientes com morada na China, utilizando subqueries com `EXISTS`.
2. **Ligação entre entidades:** A junção (`JOIN`) entre as tabelas `customer` e `cust_order` permite explorar os pedidos por cliente, refletindo a relação lógica entre as entidades no modelo relacional.
3. **Cálculo de métricas via subqueries:**

- Cada pedido inclui três métricas-chave:
  - `total_value`: Soma de preços dos livros do pedido (`SUM(...)`).
  - `total_items`: Contagem de livros no pedido (`COUNT(...)`).
  - `days_since_last_status`: Diferença em dias desde a última atualização de estado, com cálculo condicional (`CASE`) e uso de `MAX(...)` sobre `status_date`.

4. **Estrutura tabelar otimizada:** Contrariamente a modelos não relacionais, os resultados em Oracle são altamente estruturados, sendo apropriado para uso direto em relatórios ou integração com ferramentas analíticas (*e.g.*, PowerBI).
5. **Ordenação sequencial:** O resultado é ordenado por (`ORDER BY`) `customer_id` e `order_id`, mantendo uma estrutura intuitiva para visualização e navegação de dados.

Esta consulta SQL em OracleDB reproduz com precisão a lógica da pretendida, respeitando as particularidades do modelo relacional e explorando os mecanismos avançados de subconsulta e manipulação temporal da linguagem SQL padrão Oracle.

4.1.4.2) MongoDB - Pipeline de agregação

O pipeline implementado consiste em 8 estágios estratégicos que transformam e analisam os dados:

```
[
  // 1. Filter customers by country
  { $match: { "addresses.country._id": 42 } }, // China

  // 2. Unwind the orders list
  { $unwind: { path: "$orders", preserveNullAndEmptyArrays: false } },

  // 3. Join order details
  { $lookup: { from: "orders", localField: "orders", foreignField: "_id", as:
"order" } },

  // 4. Transform order array into object
  { $unwind: "$order" },
```

```
  // 5. Calculate metrics
  { $addFields: {
      "order.lines": { $ifNull: ["$order.lines", []] },
      "order.history": { $ifNull: ["$order.history", []] },
      "order.total_value": { $sum: "$order.lines.price" },
      "order.total_items": { $size: "$order.lines" },
      "order.days_since_last_status": {
        $cond: {
          if: { $gt: [{ $size: "$order.history" }, 0] },
          then: { $dateDiff: {
            startDate: { $toDate: { $max: "$order.history.status.date" } },
            endDate: "$$NOW", unit: "day" }
          },
          else: null
        }
      }
    }
  },

  // 6. Select important fields
  { $project: {
      _id: 1, first_name: 1, last_name: 1, email: 1,
      order: { _id: 1, total_value: 1, total_items: 1, days_since_last_status:
1 }
    }
  },

  // 7. Regroup by customer
  { $group: {
      _id: "$_id", first_name: { $first: "$first_name" },
      last_name: { $first: "$last_name" }, email: { $first: "$email" },
      orders: { $push: {
        $cond: { if: { $ne: ["$order", {}] }, then: "$order", else: "$$REMOVE" }
      }}
    }
  },

  // 8. Sort results
  { $sort: { _id: 1 } }
]
```

A **eficácia** desta abordagem deve-se a:

1. **Filtragem inicial**: Seleciona (`$match`) apenas clientes com morada na China, segmento de maior relevância para a análise.
2. **Decomposição e junção**: Os estágios [2, 4] decompõem (`$unwind`) a lista de pedidos e incorporam (`$lookup`) os seus detalhes, criando uma estrutura adequada para processamento.
3. **Cálculo de métricas**: O estágio 5 adiciona (`$addFields`) e calcula (`$sum` e `$size`) três indicadores essenciais para análise de vendas:
   - Valor total gasto por cliente (faturação)
   - Quantidade de livros adquiridos (volume)
   - Tempo desde a última atualização de status (eficiência logística)

4. **Otimização de dados**: Os estágios [6,7] selecionam apenas campos relevantes (`$project`) e reagrupam (`$group`) os resultados por cliente, eliminando redundâncias e preparando os dados para poderem ser utilizados, por exemplo, em relatórios executivos.
5. **Ordenação lógica**: O estágio final organiza (`$sort`) os resultados por ID do cliente, facilitando consultas e análises comparativas.

O presente pipeline demonstra a capacidade de processamento eficiente do MongoDB relativamente a dados complexos e relacionados, criando insights valiosos para decisões estratégicas no contexto de uma livraria com clientela internacional.

4.1.4.3) Neo4j – *Cypher Query*

A implementação em Neo4j utiliza a linguagem Cypher, ideal para trabalhar com grafos e explorar relações entre entidades. A query reflete a estrutura do modelo de dados em grafo, onde clientes, endereços, pedidos, livros e status são representados como nós e relacionamentos. Abaixo, detalhamos a implementação no contexto do capítulo, refletindo a sua eficácia e características.

```
// 1. Find customers who live in China
MATCH (c:Customer)-[:LIVES]->(a:Address)
WHERE a.country = "China"

// 2 & 3. Find orders placed by these customers
MATCH (c)-[:PLACED]->(o:Order)

// 4 & 5. Get order line details and calculate metrics
WITH c, o
OPTIONAL MATCH (o)-[cont:CONTAINS]->(b:Book)
WITH c, o, COLLECT({book: b, price: cont.price}) AS lines, SUM(cont.price) AS
total_value

// 6. Get status history and calculate days since last status
WITH c, o, total_value, SIZE(lines) AS total_items
OPTIONAL MATCH (o)-[h:HAS_STATUS]->(s:Status)
WITH c, o, total_value, total_items, COLLECT(h.status_date) AS status_dates

// Find the latest status date and calculate days since
WITH c, o, total_value, total_items, status_dates,
     REDUCE(latest = null, date IN status_dates |
         CASE WHEN latest IS NULL OR date > latest THEN date ELSE latest END
     ) AS last_status_date

WITH c, o, total_value, total_items, last_status_date,
     CASE
       WHEN last_status_date IS NOT NULL
       THEN duration.inDays(datetime(last_status_date), datetime()).days
       ELSE NULL
     END AS days_since_last_status

// 7. Group by customer and format as expected result
WITH c,
     COLLECT({
       _id: o.id,
       total_value: total_value,
```

```
      total_items: total_items,
      days_since_last_status: days_since_last_status
    }) AS orders
RETURN {
  _id: c.id,
  first_name: c.first_name,
  last_name: c.last_name,
  email: c.email,
  orders: orders
} AS result
ORDER BY c.id
```

A eficácia desta abordagem destaca-se pelos seguintes aspectos:

1. **Exploração natural de relacionamentos**: O modelo de grafo permite representar as entidades (clientes, endereços, pedidos, livros, status) como nós e as suas conexões como relacionamentos (`LIVES`, `PLACED`, `CONTAINS`, `HAS_STATUS`). A *query* utiliza `MATCH` para navegar nas relações de forma intuitiva, refletindo a semântica do domínio.
2. **Filtragem eficiente por país**: O primeiro passo (`WHERE a.country = "China"`) filtra clientes com endereço na China, aproveitando a estrutura do grafo para aceder diretamente ao nó `Address` via relacionamento `LIVES`. Esta abordagem é tendencialmente mais eficiente do que junções em bases de dados relacionais para grandes volumes de dados.
3. **Cálculo de métricas em tempo real**:
   - **Valor total** (`total_value`): Calculado com `SUM(cont.price)` sobre os relacionamentos `CONTAINS`, agregando os preços dos livros em cada pedido.
   - **Total de itens** (`total_items`): Obtido com `SIZE(lines)`, contando as linhas de pedido.
   - **Dias desde último status** (`days_since_last_status`): Utiliza `REDUCE` para encontrar a data mais recente no histórico de status e `duration.inDays` para calcular a diferença em dias até à data atual.
4. **Flexibilidade com dados opcionais**: O uso de `OPTIONAL MATCH` para linhas de pedido e histórico de status garante que pedidos sem livros ou status sejam incluídos nos resultados, com valores `NULL` apropriados, mantendo a integridade da análise.
5. **Agrupamento e formatação**: A *query* agrupa os resultados por cliente utilizando `COLLECT` para criar uma lista de pedidos com as suas métricas. O resultado final é formatado como um objeto semelhante ao *json*, facilitando integração com sistemas externos ou visualizações.
6. **Ordenação lógica**: O `ORDER BY c.id` organiza os resultados por ID do cliente, garantindo consistência e facilidade de navegação.

4.2) Benchmarks

A execução das queries `getTopSellingBooksInPortugal`, `getTopEarningAuthors`, `getTopCountryForHarperSales` e `getOrderMetricsByCustomer` nos sistemas de gestão de bases de dados MongoDB, OracleDB e Neo4j revela diferenças notáveis de desempenho. Os testes foram realizados em dois dispositivos: um Apple M1 com 16 GB de RAM e um Ryzen 5 3600 com a mesma memória, o que permitiu observar a relação entre as características de hardware e o comportamento de cada SGBD.

4.2.1) Tempos de execução

A seguir, os tempos médios de execução das queries em cada SGBD para o dispositivo Apple M1:

Query	T_{exec}MongoDB	T_{exec}OracleDB	T_{exec}Neo4j
`getTopSellingBooksInPortugal`	2.85 ms	5.48 ms	82.51 ms
`getTopEarningAuthors`	1.67 ms	60.19 ms	177.2 ms
`getTopCountryForHarperSales`	46.76 ms	3.05 ms	28.27 ms
`getOrderMetricsByCustomer`	55.77 ms	1503.59 ms	85.56 ms

Tabela 1: Tempos médios de execução de cada query relativamente ao SGBD - Apple M1, 16GB RAM.

E, abaixo, os mesmos testes executados no dispositivo Ryzen 5 3600:

Query	T_{exec}MongoDB	T_{exec}OracleDB	T_{exec}Neo4j
`getTopSellingBooksInPortugal`	3.8 ms	12.59 ms	356.34 ms
`getTopEarningAuthors`	1.94 ms	98.18 ms	416.7 ms
`getTopCountryForHarperSales`	97.46 ms	10.45 ms	32.07 ms
`getOrderMetricsByCustomer`	108.19 ms	2310.92 ms	182.05 ms

Tabela 2: Tempos médios de execução de cada query relativamente ao SGBD - Ryzen 5 3600, 16GB RAM.

Nos dois cenários, o MongoDB destaca-se como o mais rápido em três das quatro queries, demonstrando grande eficiência em consultas baseadas em agregações documentais. O OracleDB mostra desempenho sólido em queries estruturadas, embora seja penalizado em operações com lógica temporal e subconsultas. O Neo4j, por sua vez, apresenta os maiores tempos médios, principalmente em hardware AM4 (Ryzen), sendo mais sensível à complexidade das agregações do que às relações de grafo.

4.2.2) Utilização de CPU e Memória

Para interpretar os custos computacionais, foi avaliado a utilização média de CPU (%) e de memória (MB) durante a execução das queries no dispositivo Apple M1. Os resultados estão organizados por SGBD:

Query	$\textbf{OracleDB}_{\textbf{CPU}}$(%)	$\textbf{OracleDB}_{\textbf{Mem}}$ (MB)
Query 1: `getTopSellingBooksInPortugal`	7.9	42.15
Query 2: `getTopEarningAuthors`	21.8	45.69
Query 3: `getTopCountryForHarperSales`	3.4	46.23
Query 4: `getOrderMetricsByCustomer`	0.4	46.25

Tabela 3: Utilização média de CPU e consumo médio de memória para OracleDB - Apple M1, 16GB RAM.

Query	$\textbf{MongoDB}_{\textbf{CPU}}$(%)	$\textbf{MongoDB}_{\textbf{Mem}}$ (MB)
Query 1: `getTopSellingBooksInPortugal`	10.89	0.02
Query 2: `getTopEarningAuthors`	8.75	0
Query 3: `getTopCountryForHarperSales`	1.03	0.01
Query 4: `getOrderMetricsByCustomer`	8.27	0.36

Tabela 4: Utilização média de CPU e consumo médio de memória para MongoDB - Apple M1, 16GB RAM.

Query	$\textbf{Neo4J}_{\textbf{CPU}}$(%)	$\textbf{Neo4J}_{\textbf{Mem}}$ (MB)
Query 1: `getTopSellingBooksInPortugal`	36.66	0.07
Query 2: `getTopEarningAuthors`	60.5	0.54
Query 3: `getTopCountryForHarperSales`	18.35	0
Query 4: `getOrderMetricsByCustomer`	42.23	0

Tabela 5: Utilização média de CPU e consumo médio de memória para Neo4J - Apple M1, 16GB RAM.

O MongoDB apresenta um consumo praticamente nulo de memória e baixa utilização de CPU, tornando-se excelente para ambientes com poucos recursos. O OracleDB tem consumo de memória consistente (em torno de 45 MB), mas apresenta variação na utilização de CPU conforme a complexidade da query. Já o Neo4j consome pouca memória, mas exige bastante da CPU — superando os 60% em algumas operações — o que pode impactar a escalabilidade em ambientes concorrentes.

4.2.3) Considerações finais

Os testes evidenciam que:

- MongoDB é ideal para dados aninhados e pipelines simples a moderados, com excelente desempenho geral e consumo mínimo de recursos.
- OracleDB é robusto para consultas estruturadas e operações relacionais clássicas, mas é penalizado em queries com lógica condicional ou temporal.
- Neo4j é mais adequado para análises centradas em relações e grafos complexos, embora o seu desempenho caia em agregações e operações financeiras.

Conclusão: A escolha do SGBD deve estar alinhada com o tipo de consulta predominante e com a arquitetura dos dados. MongoDB oferece velocidade e leveza, OracleDB entrega consistência relacional e precisão, e Neo4j é vantajoso quando as relações são mais importantes que os próprios dados.

5) Conclusão

A realização deste trabalho permitiu explorar de forma aprofundada três paradigmas distintos de bases de dados — relacional, documental e em grafo — aplicando-os a um caso concreto e suficientemente complexo como o de uma livraria. Através da modelação, migração, implementação de funcionalidades e análise de desempenho, foi possível obter uma perspetiva clara sobre as vantagens, limitações e especificidades de cada abordagem.

A transposição para MongoDB (secção 3.1) exigiu uma reestruturação adaptada às características do modelo documental. A utilização de documentos embebidos e arrays trouxe benefícios evidentes na simplificação de queries e flexibilidade do esquema, como observado nas coleções Books, Customers e Orders. Contudo, tal flexibilidade implicou a implementação de mecanismos adicionais de controlo, como triggers e validações personalizadas (secção 3.1.4), para assegurar a integridade dos dados em cenários onde, nativamente, não existem restrições como chaves estrangeiras.

O modelo em grafo, através do sistema Neo4j (secção 3.2), destacou-se pela capacidade de representar e consultar dados altamente interligados com grande expressividade. A linguagem Cypher, combinada com a estrutura de nós e arestas, revelou-se particularmente eficiente em consultas que envolvem múltiplos níveis de ligação, como se verificou nas queries para obter métricas de autores ou análises por país (secções 4.1.2 e 4.1.3). As funcionalidades oferecidas pela biblioteca APOC permitiram replicar com sucesso logic business complexas através de procedures e triggers equivalentes aos de sistemas relacionais.

A análise comparativa de desempenho e uso de recursos (secção 4.2) mostrou que cada paradigma possui contextos onde é mais vantajoso. Enquanto o OracleDB apresentou tempos de resposta estáveis e previsíveis, MongoDB demonstrou melhor desempenho em operações de leitura sobre documentos extensos e com estrutura variável. Já o Neo4j revelou a sua superioridade na execução de queries complexas com múltiplas relações, oferecendo tempos de execução bastante competitivos em cenários de elevada conectividade entre dados.

Em síntese, este trabalho reforça a importância de compreender profundamente os diferentes modelos de dados disponíveis, de forma a fazer escolhas tecnológicas informadas e ajustadas aos objetivos de cada sistema. A adoção consciente de paradigmas alternativos ao relacional tradicional pode traduzir-se em ganhos substanciais de eficiência, escalabilidade e expressividade, especialmente em domínios com relações complexas e dinâmicas como o aqui analisado.

Bibliografia

[1] [Online]. Disponível em: https://www.mongodb.com/resources/products/capabilities/stored-procedures

[2] [Online]. Disponível em: https://neo4j.com/docs/cypher-manual/current/clauses/delete/#:~:text=It%20is%20not%20possible%20to,using%20the%20DETACH%20DELETE%20clause.