Mestrado em Engenharia informática

Bases de Dados NoSQL

Library Database Project

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

Introdução e Contexto - Oracle sql

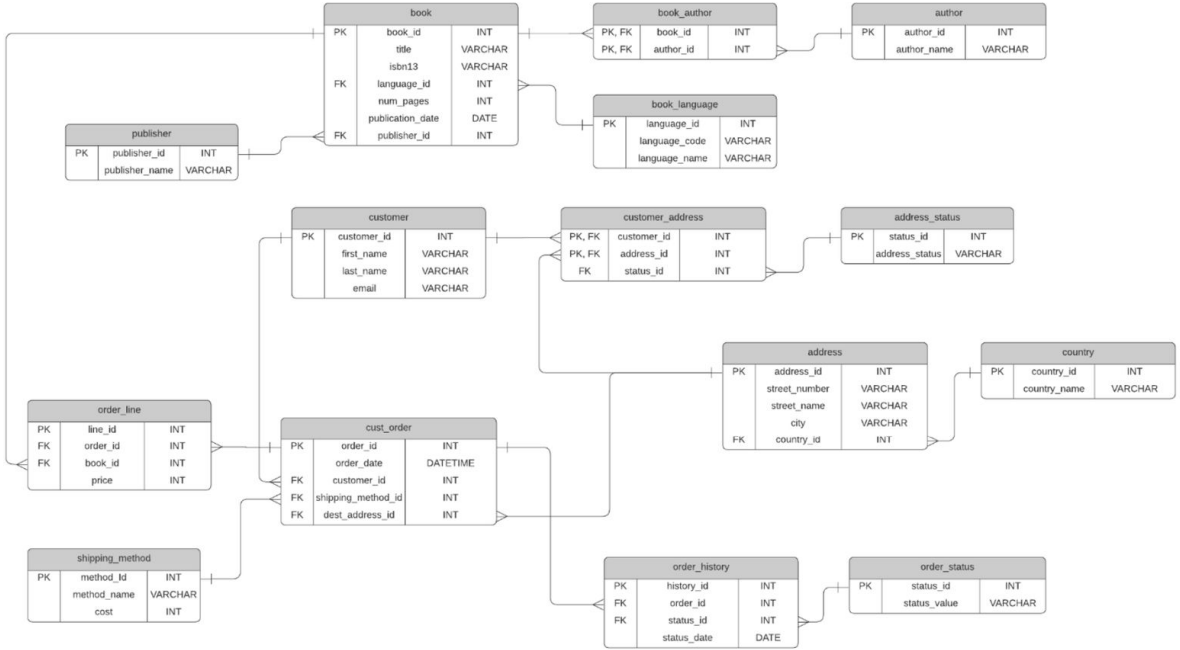

Fig 1: Modelo lógico da base de dados.

- Indexes
- Views
- Procedures
- Triggers

MongoDB

Coleções :

Books

Centraliza todas as informações relevantes sobre os livros, incluindo autores, linguagem e editor.

Customers

Reúne os dados pessoais dos clientes, incluindo moradas e país.

Orders

Agrega todas as informações relativas aos pedidos realizados, como a lista de livros por pedido, o método de envio, o histórico e o estado atual.

Indexes

Adaptação:

A conversão dos índices de Oracle SQL para MongoDB é relativamente direta, pois ambos os sistemas partilham o conceito fundamental de indexação para otimizar consultas.

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

Fig 3: Indexes no MongoDB

Procedure

Adaptação:

Para a realizar um equivalente à procedure SQL em MongoDB foi necessária a criação do ficheiro Python *procedures.py*. A passagem de uma stored procedure SQL para um ficheiro Python reflecte uma diferença fundamental na filosofia e na arquitectura dos sistemas de base de dados relacionais e NoSQL.

```python
# Função que simula a procedure
def update_order_status(mongo_db, order_id, new_status_id):
    orders_collection = mongo_db["orders"]

    order = orders_collection.find_one({"_id": order_id})

    if not order:
        print(f"Pedido com ID {order_id} não encontrado.")
        return

    order_history = order.get("order_history", [])

    last_status_id = None
    if order_history:
        last_status = sorted(order_history, key=lambda x: x["status_date"],
reverse=True)[0]
        last_status_id = last_status["status_id"]

    if last_status_id != new_status_id:
        new_status = {
            "history_id": len(order_history) + 1,
            "status_id": new_status_id,
            "status_value": get_status_value(mongo_db, new_status_id),
            "status_date": datetime.datetime.utcnow()
        }
```

Fig 4: Ficheiro equivalente a Procedure em MongoDB.

Views

Exemplo da primeira View:

(View list books and their authors)

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.

```
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"
      }
    }
  ]
)
```

Fig 4: View em MongoDB.

Triggers

Exemplo do Trigger validate_email:

```
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.');

    }
  }
}
```

Fig 5: Trigger em MongoDB.

```
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"
});
```

Fig 6: *Schema Validation* em MongoDB.

Criamos 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.

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***. Com esta funcionalidade podemos verificar o schema e validar o formato do email.

Queries desenvolvidas em Oracle SQL

Exemplos:

- QUERY 1: Listar todos os livros com seus respectivos autores e editoras.
- QUERY 2: Top 10 dos clientes com mais encomendas.
- QUERY 6: Autores mais produtivos.
- QUERY 11: Livros nunca vendidos.
- ...

```
SELECT
    b.title AS "Título do Livro",
    LISTAGG(a.author_name, ', ') WITHIN GROUP (ORDER BY a.author_name) AS
"Autores",
    p.publisher_name AS "Editora",
    bl.language_name AS "Idioma",
    b.num_pages AS "Páginas",
    b.publication_date AS "Data de Publicação"
FROM book b
JOIN book_author ba ON b.book_id = ba.book_id
JOIN author a ON ba.author_id = a.author_id
JOIN publisher p ON b.publisher_id = p.publisher_id
JOIN book_language bl ON b.language_id = bl.language_id
GROUP BY b.book_id, b.title, p.publisher_name, bl.language_name, b.num_pages,
b.publication_date
ORDER BY b.title;
```

Fig 2: Query 1 em SQL.

Queries em MongoDB

Exemplo da adaptação da primeira Query: Listar todos os livros com seus respectivos autores e editoras.

Conversão da query SQL com múltiplos JOINs e agregações para uma ***aggregation pipeline***.

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*.

```python
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))
```

Fig 4: Query 1 em MongoDB.

Neo4j

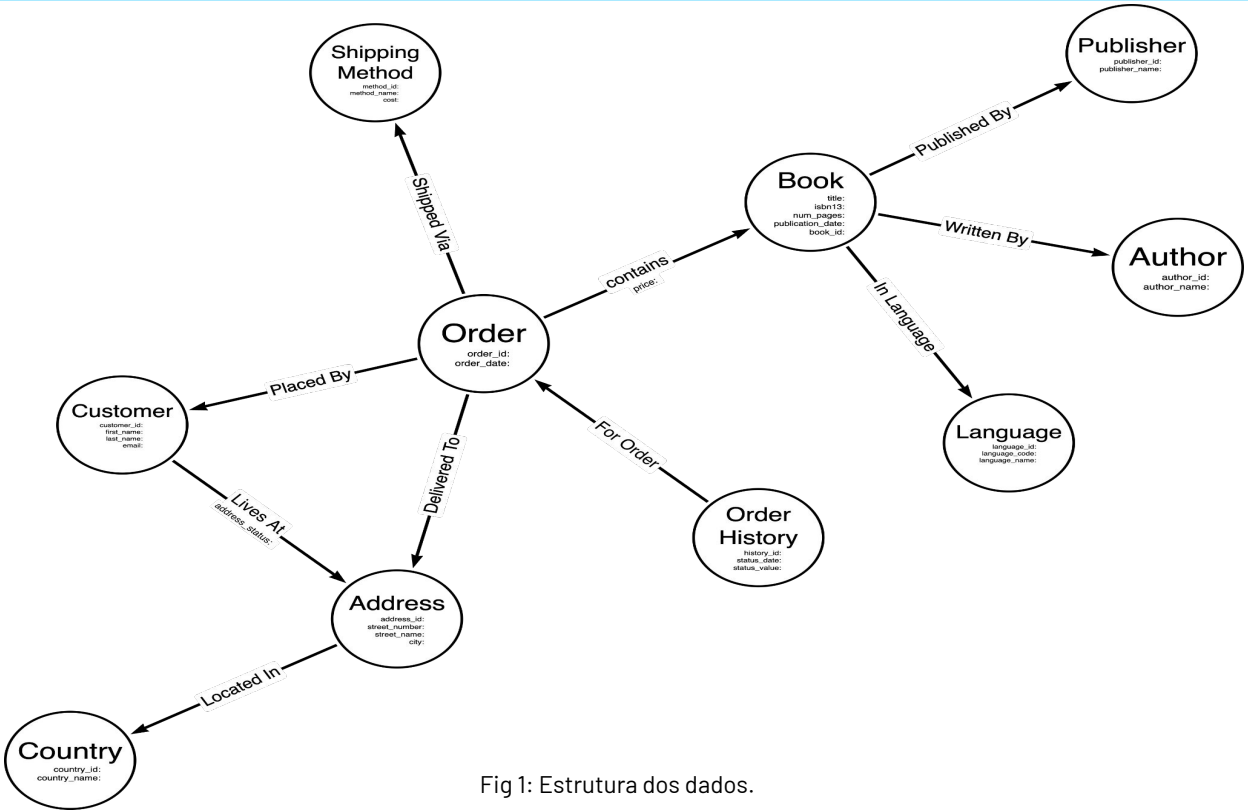

Fig 1: Estrutura dos dados.

Indexes

Adaptação:

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.

```
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)
```

Fig 2: Indexes dos dados

Procedures

Adaptação:

Para se obtermos um equivalente às procedures SQL em Neo4j, é definimos a procedure diretamente em Cypher, utilizando como de bibliotecas de apoio a extensão APOC para instalar as procedures (apoc.custom.installProcedure).

```
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'
);
```

Fig 3: Procedure em neo4j

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 a que a view SQL correspondente.

Exemplo de uma View:

```
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
```

Fig 4: View em Neo4j

Triggers

```
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"}
);
```

Fig 5: Trigger validate_email_update

```
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"}
);
```

Fig 6: Trigger validate_email_create

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

Foi necessário criar dois triggers distintos: um para validar o email durante a criação de novos nós e outro para validar o email durante a atualização dessa propriedade em nós existentes

Triggers

prevent_book_deletion

- A tradução direta do trigger prevent_book_deletion não foi realizada.
- O Neo4j, por defeito, já oferece uma proteção robusta contra a eliminação de nós que possuem relações, cumprindo o objetivo principal do trigger sem necessidade de código adicional.

Queries em Neo4j

Exemplo da adaptação da primeira Query: Listar todos os livros com seus respectivos autores e editoras.

A query, que em SQL exigiria múltiplos JOINs, é traduzida para Cypher descrevendo **padrões de relacionamento no grafo.**

Em Neo4j, a cláusula *MATCH* é utilizada para percorrer relações diretas entre nós (como *Book*, *Author*, *Publisher*, *Language*) através dos seus relacionamentos (como *WRITTEN_BY*, *PUBLISHED_BY*, *IN_LANGUAGE*), evitando a complexidade dos JOINs de múltiplas tabelas relacionais que o Oracle necessita.

```
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
```

Fig 7: Query em Neo4j

Análise crítica

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

Tabela 1: Comparação dos tempos de execução.

Pontos fortes

Oracle sql

- JOINs otimizados
- Agregações Complexas
- Índices Sofisticados

MongoDB

- Estrutura de Documentos
- Queries Geográficas e Temporais
- Simplicidade Estrutural
- Flexibilidade de *Schema*

Neo4j

- Travessia de Relações Diretas
- Agregações em Dados Conectados
- Travessias Profundas
- Subconsultas Modulares

Conclusões sobre a comparação

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 é menos eficiente em operações analíticas complexas que são comuns em sistemas OLAP.

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.

Conclusão do Trabalho

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.

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.

Mestrado em Engenharia informática

Bases de Dados NoSQL

Library Database Project

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