UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Rafael Parola

Uma solução para mapeamento de bancos de dados NoSQL baseados em documentos para bancos de dados relacionais NewSQL

Florianópolis
2023

Rafael Parola

Uma solução para mapeamento de bancos de dados NoSQL baseados em documentos para bancos de dados relacionais NewSQL

Proposta de Trabalho de Conclusão de Curso do Curso de Graduação em Sistemas de Informação do Campus Florianópolis da Universidade Federal de Santa Catarina para a obtenção do título de bacharel.
Orientador: Prof. Ronaldo dos Santos Mello, Dr.

Florianópolis
2023

Ficha de identificação da obra

A ficha de identificação é elaborada pelo próprio autor.

Orientações em:

http://portalbu.ufsc.br/ficha

Rafael Parola

Uma solução para mapeamento de bancos de dados NoSQL baseados em documentos para bancos de dados relacionais NewSQL

Este Proposta de Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel e aprovado em sua forma final pelo Curso de Graduação em Sistemas de Informação.

Prof. Álvaro Junio Pereira Franco, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Ronaldo dos Santos Mello, Dr.
Orientador

Prof. Geomar André Schreiner, Dr.
Avaliador(a)
Universidade Federal da Fronteira Sul

Prof. Roberto Willrich, Dr.
Avaliador(a)
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Agradeço aos meus pais por todo o amor que sempre me deram e que são a minha maior inspiração. Agradeço também a Marina, minha esposa que está sempre comigo nos momentos bons e ruins me dando o apoio necessário para seguir em frente e que me inspira diariamente a ser uma pessoa melhor. Agradeço também a todos os professores que fizeram parte do meu aprendizado e evolução e que me forneceram a oportunidade para que eu chegasse neste ponto tão importante da minha vida.

RESUMO

Em meados do século passado se iniciou uma busca pela otimização do armazenamento de dados. Os dados que antes eram armazenados em arquivos físicos passaram a ser armazenados em bancos de dados digitais e a partir de então, diversos modelos de bancos de dados foram propostos, como o modelo relacional onde os dados seguem um padrão pré definido além de garantir a integridade dos dados. Com o avanço da tecnologia e o surgimento do big data os modelos apresentados até então se tornaram ineficientes em relação a esta grande demanda. Desta forma, novos desafios de gerenciamento de dados foram traçados e outros modelos de dados apareceram, dentre eles os modelos e bancos de dados NoSQL, que prometiam uma estrutura mais flexível e mais desempenho, porém relaxavam as propriedades que garantiam a integridade dos dados. Então, em busca de uma solução que combinasse o desempenho dos bancos de dados NoSQL e a integridade dos bancos de dados relacionais surgiu o movimento NewSQL, uma nova categoria de bancos de dados que entrega a robustez do modelo relacional, aliado a linguagem de alto nível SQL, com a escalabilidade e desempenho dos bancos de dados NoSQL. Neste contexto, este trabalho visa o desenvolvimento de uma solução para o mapeamento da estrutura de bancos de dados orientados a documentos para o modelo relacional. Como banco de dados origem deste trabalho foi escolhido o banco de dados NoSQL baseado em documentos, pois se trata de um modelo muito utilizado atualmente, sendo que algumas distribuições, como por exemplo o MongoDB, se destacam no mercado. Como banco de dados destino foi definido o banco de dados NewSQL, pois se trata de uma nova abordagem de gerenciamento de dados que busca unir as melhores qualidades dos modelos relacionais e NoSQL, fornecendo desempenho e integridade para os dados. Ao final, a solução desenvolvida terá seu desempenho analisado através de experimentos que medirão o seu tempo de execução, bem como a capacidade necessária para o seu armazenamento no banco de dados destino.

Palavras-chave: Banco de dados, Modelo relacional, NoSQL, NewSQL

ABSTRACT

In the middle of the last century, a search for storage optimization began of data. Data that was previously stored in physical files became stored in digital databases and from then on, various database models of data have been proposed, such as the relational model where data follows a pattern pre-defined in addition to guaranteeing data integrity. With the advancement of technology and the emergence of big data, the models presented so far have become inefficient in in relation to this great demand. In this way, new data management challenges were outlined and other data models appeared, including models and databases of NoSQL data, which promised a more flexible structure and more performance, but they relaxed the properties that guaranteed data integrity. So, in search of a solution that combines the performance of NoSQL databases and the integrity relational databases emerged the NewSQL movement, a new category of databases that deliver the robustness of the relational model, combined with the high level SQL, with the scalability and performance of NoSQL databases. In this context, this work aims to develop a solution for mapping the structure of document-oriented databases for the relational model. As source database for this work, the NoSQL database was chosen based in documents, as it is a model widely used today, and Some distributions, such as MongoDB, stand out in the market. As target database, the NewSQL database was defined, as it is a new data management approach that seeks to combine the best qualities of models relational and NoSQL, providing performance and data integrity. In the end, the developed solution will have its performance analyzed through experiments that will measure its execution time, as well as the capacity required for its storage in the target database.

Keywords: Databases, Relational Model, NoSQL, NewSQL

LISTA DE ALGORITMOS

*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Os bancos de dados (BDs) estão presentes em praticamente todos os contextos da sociedade. Com o avanço da tecnologia e o surgimento do movimento *big data*, grande parte de tudo o que fazemos geram dados, seja em uma compra no mercado ou na busca de artigos para o desenvolvimento deste trabalho. Sendo assim, a busca por uma melhor forma de armazenar e manipular estes dados surgiu (M. ANDREAS, 2019).

Dessa forma, surgiram diversos modelos de BDs, dentre eles o modelo relacional, mais maduro, fornecendo uma estrutura simples para os dados e integridade em suas transações. Entretanto, ele não se mostrou eficiente quando utilizado em aplicações que geram e precisam armazenar um grande volume de dados muitas vezes heterogêneos e complexos. A partir disso surgiram os BDs NoSQL, que em muitos casos relaxam sua integridade e fornecem uma maior flexibilidade de estruturação dos dados em troca de um melhor desempenho com grandes volumes de dados (F. MARTIN, 2019). Mesmo assim, a comunidade acadêmica e da indústria na área de BD continuou na busca de alternativas eficientes para o *big data* e que fornecessem a estrutura e a integridade das bases de dados relacionais. Então, surgiram os BDs NewSQL, que aliam a escalabilidade horizontal e alta disponibilidade dos BDs NoSQL com o suporte à integridade e robustez do modelo relacional (MICHAEL, 2012).

Os BDs NoSQL orientados a documentos geralmente são *schemaless*, ou seja, não possuem uma estrutura predefinida de seus dados. Formados por documentos que atribuem um valor a uma chave, não há uma regra que define quais atributos deverão estar em cada documento, ou seja, documentos em um mesmo BD podem conter objetos com diferentes atributos. (D. KONSTANTINOS L. PETER, 2021).

A partir deste contexto, com a chegada dos BDs NewSQL, ideais para sistemas de segmento transacional que necessitem de alto desempenho (REBECCA, 2020), verifica-se que a migração de BDs NoSQL para o modelo relacional, adotado por BDs NewSQL, é interessante para agregar desempenho e consistência forte para dados de diversos domínios de aplicação. Entretanto, levando em consideração a grande diferença na estrutura de armazenamento e flexibilidade do modelo de dados orientado a documentos, se observa uma grande dificuldade do mapeamento de uma estrutura de um BD orientado a documentos para o modelo relacional.

A partir disso, é abordado o estudo de técnicas, que serão apresentadas durante o desenvolvimento, que foram obtidas através de trabalhos relacionados, para o mapeamento de BDs orientados a documentos baseados no formato JSON para o modelo relacional. Além disso, este trabalho tem como diferencial identificar as características dos dados através de regras de mapeamento que suportam os três tipos de relações disponíveis no modelo relacional. Também é realizada a conversão para o modelo relacional através das regras de mapeamento, que é o foco deste trabalho.

1.1 OBJETIVO GERAL

O objetivo geral deste trabalho é o desenvolvimento de uma solução que realize o mapeamento do esquema de documentos presentes em BDs NoSQL orientados a documentos para o modelo relacional, com foco em BDs NewSQL.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos para alcançar o objetivo geral são os seguintes:

- Definir regras de mapeamento de um BD orientado a documentos para o modelo relacional;
- Desenvolver solução capaz de realizar a conversão automática da estrutura de BD orientado a documento para o modelo relacional, para execução em um BD NewSQL;
- Avaliar o mapeamento da estrutura gerada pela solução, bem como o seu desempenho a partir de consultas predefinidas em ambas as bases de dados.

1.3 JUSTIFICATIVA

A justificativa para este trabalho envolve a análise e a necessidade de avanço nas técnicas de armazenamento e manipulação de dados no contexto de *big data*. Levando em consideração as limitações dos BDs relacionais tradicionais, em termos de escalabilidade e desempenho diante do crescente volume de dados, a relevância dos BDs NoSQL tornou-se evidente, especialmente devido a sua capacidade de lidar com grandes quantidade de dados de maneira eficiente. No entanto, os modelos de dados NoSQL apresentam desafios quanto à gestão da integridade e da estrutura de seus dados.

Neste cenário, os BDs NewSQL surgiram como uma solução promissora, buscando unir a escalabilidade e desempenho dos NoSQL com a robustez e integridade dos modelos relacionais. Assim, o foco deste trabalho é desenvolver uma solução para o mapeamento da estrutura de um BD NoSQL orientado a documentos para BDs relacionais NewSQL.

1.4 METODOLOGIA

Em uma primeira etapa, foi realizando o levantamento do estado da arte no que diz respeito ao "mapeamento de estruturas de BDs orientados à documentos para o modelo relacional"em fontes de trabalhos científicos, sendo elas, DBLP (Computer Science Bibliography)[1], ACM (Association for Computer Machinery)[2], Research Gate[3],

[1] https://dblp.org/
[2] https://www.acm.org/
[3] https://www.researchgate.net/

Scopus[4] e Google Scholar[5]. Desta forma, foram estudados livros, artigos e demais trabalhos publicados referentes a este tema. As obras selecionadas tiveram sua relevância para o final deste trabalho.

Na segunda etapa, os trabalhos relacionados foram analisados em conjunto e comparados, de forma a tornar possível a extração das melhores técnicas utilizadas até o momento para o mapeamento da estrutura do modelo baseado em documentos para o modelo relacional.

A terceira etapa deste trabalho, consistiu na elaboração de regras de mapeamento do modelo orientado a documentos para o modelo relacional, bem como o levantamento e comparação de ferramentas e processos de extração do esquema de uma BD orientado a documento. Por fim, uma ferramenta que satisfez as necessidades deste trabalho foi escolhida.

Na quarta etapa deste trabalho, foi realizado o projeto da solução proposta levando em consideração sua arquitetura e requisitos funcionais e não funcionais necessários a sua implementação. Deste modo, a solução foi implementada utilizando as tecnologias e processos pré definidos. Por fim, foi realizada uma avaliação do resultado obtido, bem como uma análise comparativa da ferramenta proposta com os trabalhos relacionados.

1.5 CONTEÚDO DA MONOGRAFIA

O presente trabalho está organizado em 6 capítulos. O primeiro capítulo traz a introdução, os objetivos, metodologia e justificativa para o seu desenvolvimento. O segundo capítulo descreve a fundamentação teórica, que fornece a base para a o entendimento da proposta deste trabalho. No terceiro capítulo são reunidos e analisados trabalhos correlatos de forma a obter conhecimento a respeito do estado da arte, além de também destacar qual será o diferencial do presente trabalho em relação aos demais. O quarto trata do desenvolvimento da solução proposta, incluindo um entendimento a respeito das ferramentas e tecnologias utilizadas, bem como o projeto da solução, seus requisitos funcionais e não funcionais, além das regras de mapeamento utilizadas para a conversão da estrutura da base de dados orientadas a documentos para o modelo relacional. O quinto capítulo trata da demonstração do uso da aplicação em diferentes cenários das regras de mapeamento definidas neste trabalho. O sexto capítulo trata da avaliação dos resultados obtidos pela aplicação desenvolvida, realizando uma comparação de operações executas na base de dados de origem orientada a documentos e na estrutura criada no banco de dados relacional. Por fim, o sétimo capítulo trará as considerações finais, reforçando a contribuição deste trabalho e apresentando possibilidades para uma continuação do mesmo.

[4] https://www.scopus.com/home.uri
[5] https://scholar.google.com

2 FUNDAMENTAÇÃO TEÓRICA

2.1 EVOLUÇÃO DOS SISTEMAS DE GERÊNCIA DE BD

O surgimento dos BDs como conhecemos hoje foi um feito histórico da sociedade, tornando possível usufruirmos das tecnologias disponíveis que vão desde o mais simples cadastro de dados em um sistema computacional até complexos algoritmos de Machine Learning que utilizam estes dados para realizar as mais distintas tarefas que ajudam a facilitar a nossa vida. Eles surgiram durante o início dos anos 60 a partir da dificuldade de manipular grandes volumes de arquivos físicos, ou seja, documentos e arquivos armazenados e indexados de forma manual. Este problema, enfrentado pelas empresas da época resultou em esforços voltados para a pesquisa e desenvolvimento de soluções automatizadas, que facilitassem a manipulação destes dados.

O primeiro SGBD surgiu no início dos anos 60, na empresa General Eletric. Seu projeto foi idealizado por Charles Bachman, rendendo o prêmio Turing da ACM em 1973. O SGBD foi chamando de *Integrated Data Store* e se tornou a base do modelo de rede, um dos primeiros modelos de armazenamento de dados computacionais, padronizado pela *Conference on Data System Languages (CODASYL)* (R. RAGHU, 2008).

No final dos anos 60, diversos outros BDs começaram a surgir, influenciados pelo *Integrated Data Store*. Foi aí que a empresa de computadores IBM desenvolveu o *IMS (Information Management System)*, ainda usado atualmente. Este BD se diferenciou por adotar um modelo de dados diferente do anterior. Esse novo modelo que ficou conhecido como modelo hierárquico, tornou possível diversas pessoas acessarem o mesmo dado, por meio de uma rede de computadores (R. RAGHU, 2008).

Segundo (CODD, 1970), os modelos hierárquicos e de rede ainda careciam de independência entre os dados, ofereciam também dificuldade de mudanças na representação dos dados e muitos não suportavam o crescimento de tipos de dados. Foi então que ele propôs o modelo relacional, organizado em tabelas e oferecendo independência entre os dados, características como atomicidade, consistência, isolamento e durabilidade das transações, além de uma linguagem de alto nível para a manipulação destes dados. Esse modelo foi o que mais se tornou relevante, até os dias atuais.

De acordo com o (DB-ENGINES. . . , s.d.), iniciativa que coleta e apresenta informações sobre gerenciadores de dados, dos 10 SGBDs mais utilizados desde 2014, 7 utilizam o modelo relacional como base, sendo que, grande parte destes BDs, ainda apresentam suporte para mais de um modelo de dados, como por exemplo o Oracle e o Microsoft SQL Server, conforme mostra a Figura 1. Já como mostra a Figura 2 é possível percerber que os grandes BDs relacionais, como o Oracle e o SQL Server se mantiveram entre os mais utilizados desde 2014.

Por fim, é possível concluir que, como os dados são cada vez mais importantes

Rank Dec 2023	Rank Nov 2023	Rank Dec 2022	DBMS	Database Model	Score Dec 2023	Score Nov 2023	Score Dec 2022
1.	1.	1.	Oracle	Relational, Multi-model	1257.41	-19.62	+7.10
2.	2.	2.	MySQL	Relational, Multi-model	1126.64	+11.40	-72.76
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	903.83	-7.59	-20.52
4.	4.	4.	PostgreSQL	Relational, Multi-model	650.90	+14.05	+32.93
5.	5.	5.	MongoDB	Document, Multi-model	419.15	-9.40	-50.18
6.	6.	6.	Redis	Key-value, Multi-model	158.35	-1.66	-24.22
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	137.75	-1.87	-7.18
8.	8.	↓ 7.	IBM Db2	Relational, Multi-model	134.60	-1.40	-12.02
9.	↑ 10.	9.	Microsoft Access	Relational	121.75	-2.74	-12.08
10.	↑ 11.	↑ 11.	Snowflake	Relational	119.88	-1.12	+5.11

Figura 1 – Ranking dos SGBDs mais utilizados.

Fonte: Captura de tela obtida em https://db-engines.com/en/ranking

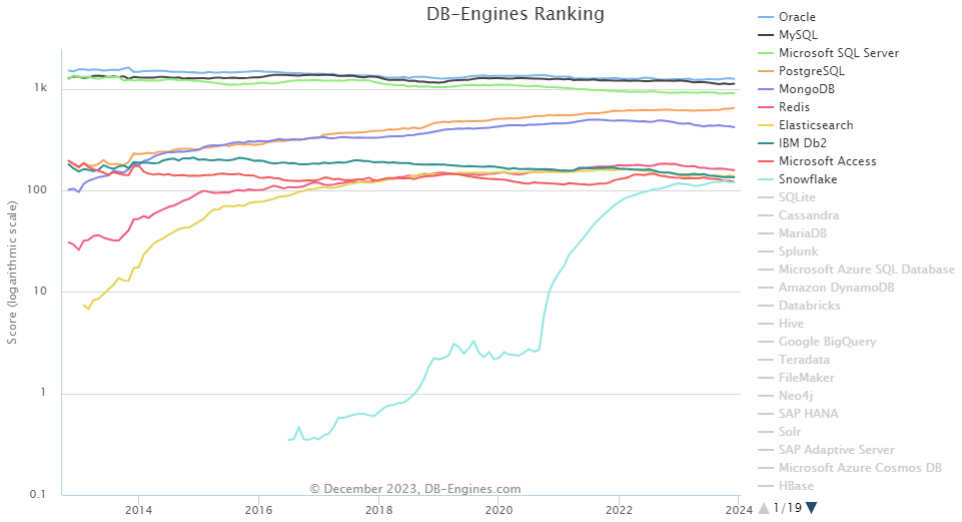


Figura 2 – SGBDs mais utilizados dos últimos 9 anos.

Fonte: Captura de tela obtida do site DB-engines.com

para a sociedade, a necessidade por inovação no seu armazenamento e manipulação nos trouxe diferentes perspectivas de como estes dados podem ser armazenados para diferentes propósitos. Apesar dos BDs relacionais ocuparem o lugar de mais utilizados até hoje, nada impede que amanhã, novas tecnologias apareçam e tomem o seu lugar, pois cada vez mais descobrimos formas de tratar os dados.

2.2 MODELO RELACIONAL E SQL

Compreender o modelo relacional é entender a sua importância em relação a facilidade de manipulação de seus dados em estado natural bem como a integridade destes dados. O modelo relacional foi superior aos seus antecessores em diversos aspectos, como a utilização de uma linguagem de alto nível para sua manipulação, bem como fornecer a base para o tratamento de consistência, redundância e derivação em suas relações (CODD, 1970).

No modelo relacional, os dados estão distribuídos em tabelas, contendo linhas ou registros, e colunas, também chamadas de atributos. Estes atributos podem estar relacionados a outras tabelas por meio de chaves primárias e chaves estrangeiras. As chaves primárias formam os identificadores únicos de um registro em uma tabela, enquanto as chaves estrangeiras referenciam este identificador em outra tabela, com o objetivo de formar uma relação entre elas (D. KONSTANTINOS L. PETER, 2021). Na Figura 3 é possível visualizar um exemplo deste modelo, composto por tabelas, atributos e suas relações.

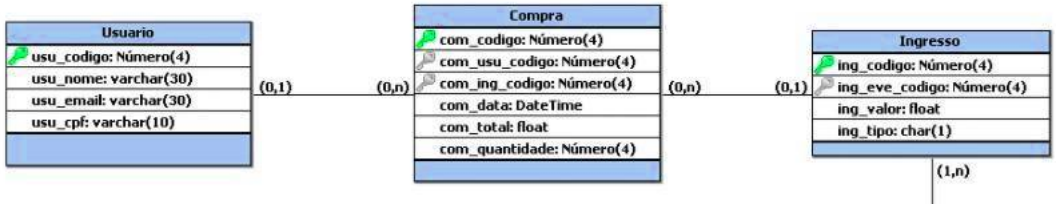


Figura 3 – Exemplo de tabelas, atributos e suas relações.

Fonte: Figura obtida do site https://pc-solucion.es/terminos/modelo-relacional/

Conforme mencionado anteriormente, os BDs relacionais garantem a integridade de seus dados, por meio de características denominadas ACID, ou seja, atomicidade, consistência, isolamento e durabilidade. Atomicidade garante que todas as ações em uma transação serão executadas, ou nenhuma será executada, por exemplo, quando há alguma falha na execução. Consistência de uma transação garante que o estado original dos dados é são preservados caso ocorram erros durante a execução, se a transação é bem sucedida, um novo estado dos dados são criados. Isolamento garante que cada transação é executada independentemente de outras transações, mesmo quando o BD suporta concorrência para fins de desempenho. Durabilidade é a garantia de que, uma vez informada ao usuário o fim com sucesso de uma transação, o dado manterá este valor, independentemente de falhas que possam ocorrer no SGBD (R. RAGHU, 2008).

Como referenciado, o modelo relacional deve suportar uma linguagem de alto nível para manipulação de seus dados. Surgiu então a linguagem SQL (*Structured Query Language*). SQL é a liguagem responsável por interagir com os BDs relacionais. Existem diversas distribuições desta linguagem, como por exemplo o ANSI SQL, Oracle SQL e o Postgres SQL. Inspirada na álgebra relacional, ela permite a criação de BDs e os seus objetos, permite a inserção de dados nestes BDs, a alteração e análise destes dados e também a seleção destes dados para a sua utilização. Esta linguagem pode ser utilizada separadamente ou em conjunto com outras linguagens de programação, como por exemplo o Java ou o Python, que possuem suporte para acesso a diversas distribuições de BDs relacionais e sua manipulação via SQL (STEVE, 2017).

2.2.1 Relacionamentos

Segundo (HERNANDEZ, 2021), existem 3 tipos de relacionamentos que podem ser aplicados às entidades, sendo eles, um-para-um, um-para-muitos e muitos-para-muitos.

2.2.1.1 Um-para-um

Relacionamentos um-para-um ocorrem quando um registro de uma primeira entidade se relaciona com 0 ou 1 registro de uma segunda entidade, e na segunda entidade, 1 e apenas 1 registro se relaciona com a primeira entidade. De forma a implementar relações um-para-um a chave compartilhada na segunda entidade deve ser primária ou única (HERNANDEZ, 2021). A Figura 4 mostra um exemplo de relacionamento um-para-um entre 2 entidades, *Employees* e *Compensation*. Um empregado está associado a apenas uma compensação e uma compensação está associada à apenas um empregado. É possível notar também que os 2 dividem o mesmo valor de chave primária.

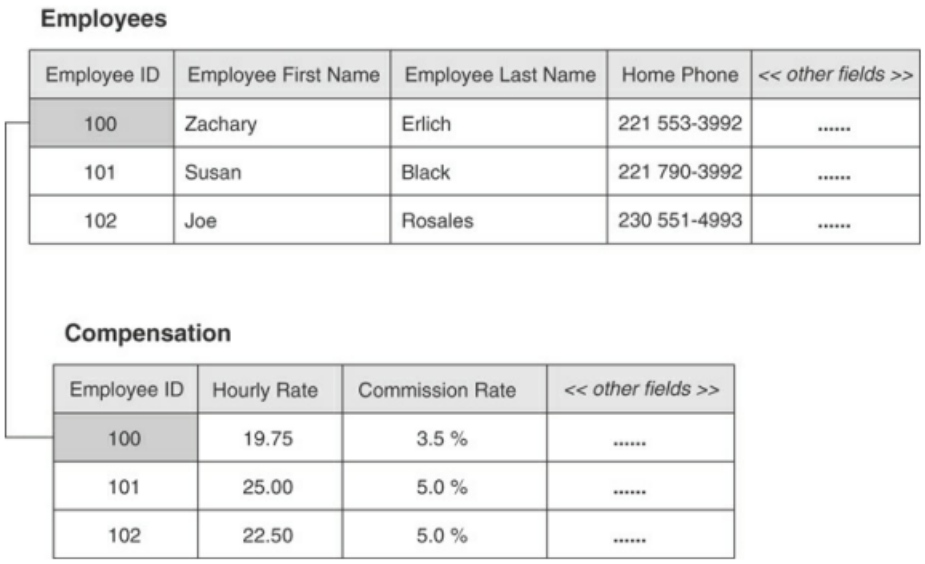


Figura 4 – Exemplo de relacionamento um-para-um.

Fonte: Database Design for Mere Mortals (HERNANDEZ, 2021)

2.2.1.2 Um-para-muitos

Relacionamentos um-para-muitos ocorrem quando um registro na primeira tabela pode estar relacionado à 0, 1 ou muitos registros em uma segunda tabela, mas um registro na segunda tabela deve estar relacionado à apenas 1 registro na primeira tabela (HERNANDEZ, 2021). A Figura 5 representa um relacionamento um para muitos entre as entidades *Agents* e *Entertainers*. Um agente está relacionado a mais de um registro em entretenimento, ou seja, cada agente pode gerenciar múltiplos grupos de entretenimento, porém não é possível existir mais de um agente para o mesmo grupo. É possível observar

que a tabela *Enterteiners* possui uma chave estrangeira *Agent ID*, apontando para a chave primária de *Agent ID* na entidade *Agents*.

Figura 5 – Exemplo de relacionamento um-para-muitos.

Fonte: Database Design for Mere Mortals (HERNANDEZ, 2021)

2.2.1.3 Muitos-para-muitos

Por último, relacionamentos muitos-para-muitos ocorrem quando um registro na primeira entidade pode se relacionar com 0, 1 ou muitos registros em uma segunda entidade, e um registro na segunda entidade pode se relacionar com 0, 1 ou muitos registros na primeira entidade também. Este tipo de relacionamento pode ser atingido por meio de uma tabela associativa, ou seja, uma terceira tabela deve ser criada de forma a permitir este tipo de relacionamento (HERNANDEZ, 2021).

A Figura 6 mostra a definição de uma tabela associativa. Uma tabela *Student Schedule* é criada com 2 chaves estrangeiras, uma apontando para a tabela *Studants* e outra para a tabela *Classes*. Dessa forma, é possível que um estudante esteja presente em 0, uma ou 1 disciplinas e uma disciplina possua 0, 1 ou mais estudantes.

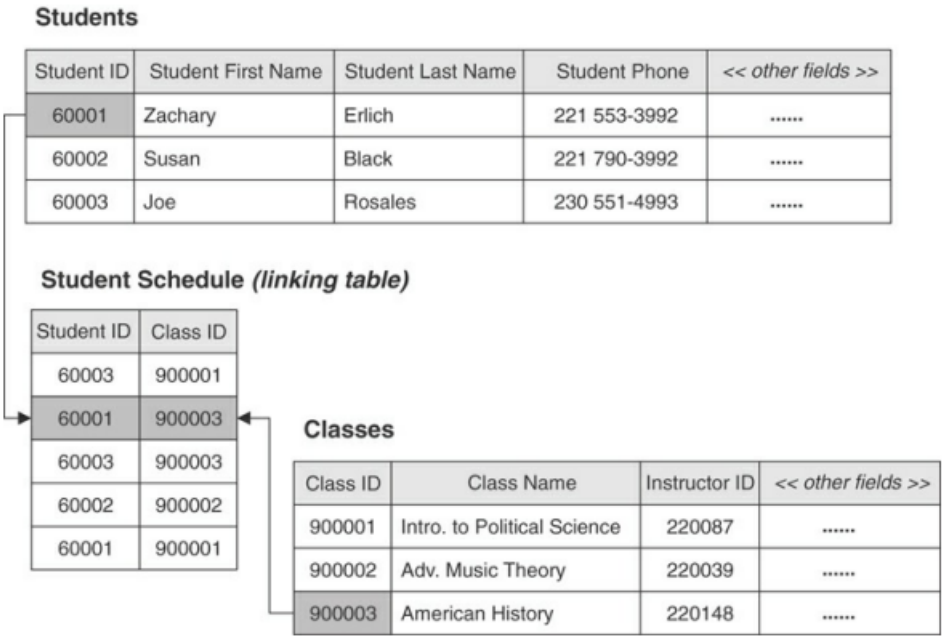


Figura 6 – Exemplo de relacionamento muitos-para-muitos.

Fonte: Database Design for Mere Mortals (HERNANDEZ, 2021)

2.3 NOSQL

O movimento *big data* trouxe novos desafios para o armazenamento e manipulação dos dados. Segundo (F. MARTIN, 2019), o modelo mais relevante até então, sendo esse o modelo relacional, possuía desafios quando se trata de um grande volume de dados, pois os BDs relacionais tradicionais não foram feitos para serem executados em clusters e sim ter sua capacidade aumentada por meio de máquinas maiores e com mais capacidade computacional, exigindo assim um espaço físico maior, além de um maior investimento financeiro. Segundo (F. MARTIN, 2019), existem dois principais motivos para adotar modelos NoSQL, que são produtividade no desenvolvimento de aplicativos e uma grande quantidade de dados.

NoSQL, ou *Not Only SQL*, não somente SQL em português, é um movimento que surgiu a partir da necessidade de encontrar diferentes meios de armazenar grandes quantidades de dados. Como diz (DAVID, 2016), com o surgimento de aplicações, como por exemplo, o Facebook, que precisa realizar a inserção de um grande volume de dados, entre usuários, posts e comentários, ou o Google que necessita realizar a indexação de sites e armazenar buscas de usuários, se tornou inviável a utilização do modelo relacional. O NoSQL surgiu para suprir esta necessidade, oferecendo escalonamento horizontal para entregar uma maior velocidade.

Segundo (DAVID, 2016), os BDs NoSQL são apresentados em diversos tipos de abordagens, como por exemplo, os BDs orientados a documentos, grafos, colunas e chave-

valor. Cada um com suas vantagens e desvantagens.

Além das vantagens mencionadas anteriormente, os diferentes modelos apresentados no NoSQL fornecem uma maior flexibilização da estrutura dos dados, fazendo com que uma aplicação consiga adequar melhor o seu modelo de negócio ao modelo utilizado no BD. Por exemplo, para o Twitter, uma famosa rede social, os modelos orientados a documento e chave-valor fariam mais sentido do que o modelo relacional (D. KONSTANTINOS L. PETER, 2021).

Uma grande diferença entre os modelos relacional e os NoSQL é que, de forma a possibilitar a adquirir velocidade em suas transações, no geral os BDs NoSQL carecem das características ACID, fundamentais para garantir a integridade dos dados que estão presentes nos BDs relacionais. Porém, segundo (D. KONSTANTINOS L. PETER, 2021), os BDs NoSQL realizam uma outra abordagem por meio das características CAD, consistência, disponibilidade e partição de tolerância. Consistência garante que todos os dados no BD sigam as regras impostas a ele, disponibilidade referência que os dados estarão sempre disponíveis para os usuários, por meio de diversos nodos e partição de tolerância referencia a habilidade do BD de encontrar uma rota entre seus diversos nodos para entregar os dados ao usuário.

2.3.1 Bancos de Dados Orientados a Documentos

Os BDs orientados a documentos são uma abordagem não normalizada. É também uma abordagem *schemaless*, ou seja, seus dados não precisam seguir uma estrutura pré-definida como por exemplo nos BDs relacionais, os dois principais motivos para isso são um melhor desempenho e também a flexibilidade de seus dados (D. KONSTANTINOS L. PETER, 2021).

Para exemplificar a característica flexível de um BD orientado a documentos, podemos dizer que enquanto no modelo relacional, todos os dados de uma mesma tabela precisam possuir os mesmos atributos, em BDs orientados a documentos, dados em documentos podem ter diferentes atributos (N. AMEYA P. ANIL, 2013). Além destas características, muitos BDs orientados a documentos são baseados em nuvem e possuem como característica serem escalonáveis horizontalmente, o que permite a criação de clusters para a execução do BD bem como o reaplicamento de seus dados. Além disso essa caraterística permite que este tipo de BD seja resistente a falhas e consiga entrega alta disponibilidade (M. ANDREAS, 2019).

É importante mencionar também que esta característica flexível passa a responsabilidade da estrutura do BD orientado a documentos para o desenvolvedor. Além disso, essa característica traz também algumas desvantagens, como não garantir a integridade dos dados e a sua normalização (M. ANDREAS, 2019).

Os BDs orientados a documentos realizam o armazenamento de seus dados em

arquivos JSON, acrônimo para *Javascript Object Notation* e derivados. Sua estrutura se baseia em pares de atributos e valores onde os valores são atribuídos a uma chave utilizada para a sua identificação, sendo que documento é independente, ou seja, não está conectado a outro documento (M. ANDREAS, 2019). Na Figura 7 é possível vizualizar um exemplo de um documento, conténdo os valores relacionados a atributos, bem como um array de valores.

```
{
    "nome": "João da Silva",
    "idade": 20,
    "matricula": "2018123490",
    "curso": "Sistemas de Informação",
    "cadeiras": [
        "Estrutura de Dados",
        "Organização de Computadores",
        "Matemática Discreta"
    ]
}
```

Figura 7 – Exemplo de arquivos JSON.

Fonte: Figura obtida do site
https://www.ufsm.br/pet/sistemas-de-informacao/2021/08/09/postgressql-json-types/

2.3.2 MongoDB

Para atingir os objetivos deste trabalho, foi escolhido o MongoDB como o SGBD orientado a documentos. A decisão para a utilização deste SGBD se deu pela sua popularidade. Atualmente é o SGBD orientado a documentos mais utilizado e o quinto SGBD mais utilizado entre todos os modelos de dados disponíveis (DB-ENGINES. . . , s.d.).

Ele têm como característica a entrega de alto desempenho e eficiência se comparado ao modelo relacional. Além disso, outras características desse SGBD são o fornecimento de buscas complexas utilizando agregação e indexação (N. AMEYA P. ANIL, 2013).

Apesar de escrito em C++, a interação com o MongoDB é feita através de Javascript. Esse mecanismo de busca fornece ao desenvovedor uma fácil adaptação a utilização deste SGBD, assim como uma fácil integração com essa linguagem de programação (DAVID, 2016).

Os dados no MongoDB ficam armazenados em arquivos BSON, uma variação do JSON, contendo listas de elementos compostos por chaves, valores e o seu tipo de dado. O arquivo BSON é mais eficiente em relação a armazenamento e velocidade se comparado ao JSON. Além disso, possui funcionalidades específicas para o armazenamento de arquivos volumosos, o que o torna ideal para aplicações com grande armazenamento de counteúdo, como por exemplo, redes sociais e também análise de dados em tempo real (N. AMEYA P. ANIL, 2013).

2.4 NEWSQL

Com o surgimento do movimento *Big Data*, aumentou cada vez mais a procura por soluções alternativas aos BDs relacionais, que se possuíam desafios quanto a sua escalabilidade para tratar grandes volumes de dados. Segundo (**Grolinger**), NoSQL e NewSQL apresentam-se como alternativas capazes de lidar com este grande volume de dados, pois os BDs NoSQL e NewSQL permitem escalonamento horizontal, melhorando assim o desempenho de acesso.

O movimento NewSQL surgiu a partir do posicionamento de que não era preciso abandonar o modelo relacional e as características ACID, fundamentais para a sua integridade, em troca de desempenho. Segundo (MICHAEL, 2010), quando tratamos de OLTP, todos os BDs relacionais modernos deveriam oferecer um alto desempenho possibilitando a criação de diferentes nodos para a execução de suas transações.

Então o ponto de destaque para NewSQL é o diferencial de manter as características ACID presentes nos BDs relacionais tradicionais, mantendo também um modelo estruturado e fazendo uso da linguagem de alto nível SQL enquanto realiza o escalonamento horizontal de forma a melhorar o seu desempenho (**Grolinger**).

2.4.1 CockroachDB

O foco deste trabalho está relacionado a utilização de um BD relacional, sendo assim, foi definido que o SGBD relacional utilizado seria o CockroachDB. Este SGBD NewSQL, foi escolhido por se tratar do BD deste segmento que se tornou mais relevante, sendo atualmente o mas utilizado SGBD NewSQL (DB-ENGINES. . . , s.d.).

O CockroachDB se difere dos BDs relacionais tradicionais, pois se trata de um BD baseado em nuvem. Ele abstrai toda a configuração de um BD, como por exemplo, instalação em servidores, administração do BD, escalonamento vertical, configuração de replicas, dentre outras funções de manutenção em um BD (ROB, 2022).

A arquitetura do CockroachDB é formada por nodos que podem estar em um mesmo *datacenter* ou espalhados pelo globo. Este nodos podem possuir tanto dados quanto capacidade computacional. Esta arquitetura permite ao CockroachDB ser resistente a falhas, ou seja, se um nodo ficar indisponível, outros nodos com dados replicados serão capazes de fornecer estes dados aos usuários de maneira automática. Isso também é chamado de alta disponibilidade (REBECCA, 2020). O CockroachDB funciona por camadas, sendo que a sua interface de interação é feita a partir da linguagem SQL. Para o usuário manipulando dados via SQL, de maneira geral não se torna necessário o conhecimento específico a respeito dos nodos ou replicação dos dados no BD para a busca ou manipulação de dados, pois camadas abaixo dessa abstraem todo esse gerenciamento.

As requisições da camada SQL chegam a camada transacional, responsável pelas características de atomicidade e isolamento das transações efetuadas no BD. Depois os

dados são passados para a camada de distribuição, onde os dados que chegam ao BD são associados a chaves para o mapeamento e depois são direcionados a diferentes partes do *cluster* formado pelos nodos. Depois os dados chegam a camada de replicação, onde eles são replicados em diferentes nodos, de forma a preservar estes dados. Só depois, eles chegam a camada de armazenamento, onde são escritos de forma a priorizar o desempenho (REBECCA, 2020).

É possível concluir que o CockroachDB se trata de uma solução capaz de unir características dos BDs relacionais com as não relacionais. Ele torna possível unir o alto desempenho com o escalonamento horizontal, alta disponibilidade ao modelo relacional, armazenamento em tabelas e relações e também a integridade dos dados no BD.

3 TRABALHOS CORRELATOS

O mapeamento de dados entre BDs baseados em documentos e sistemas relacionais se tornou necessário devido à grande diversidade de SGBDs que surgiram devido ao crescimento do *Big Data*. Para investigar a literatura existente nesse contexto foi realizada uma revisão sistemática. A motivação para essa revisão foi identificar como soluções tem lidado com a dificuldade de armazenar dados de documentos em BDs relacionais, bem como o desafio de migrar a estrutura destes documentos baseados em JSON, que permitem uma maior flexibilidade, para estruturas relacionais, onde os dados inseridos nelas, devem se enquadrar a uma estrutura definida pela tabela.

A dificuldade de migração dos dados e das estruturas flexíveis de BDs não relacionais baseadas em documentos JSON para BDs estruturados relacionais baseadas em tabelas pode acarretar no aparecimento de problemas devido a má conversão destes dados e estruturas, fazendo com que no futuro seja necessário um retrabalho em cima deste BD.

3.1 METODOLOGIA PARA BUSCA DE TRABALHOS

Uma metodologia clássica de revisão sistemática foi aplicada (KITCHENHAM, 2004). Para tanto, foi realizada uma busca em fontes de trabalhos científicos, onde buscas por palavras-chaves foram executadas de forma a selecionar os estudos de interesse deste trabalho. Após uma revisão inicial dos títulos alguns estudos foram reunidos, e posteriormente foram aplicados critérios de inclusão específicos, para selecionar dentre estes, apenas os que de fato possuem informação relevante para o desenvolvimento deste trabalho.

Para a busca de trabalhos correlatos, foram utilizadas fontes eletrônicas de trabalhos acadêmicos. As fontes escolhidas são organizações internacionais, sendo que algumas delas com acervos voltados à área de Ciência da Computação a fim de obter o o máximo de trabalhos possíveis e fontes seguras e atuais de informação para a realização desta revisão sistemática. As fontes escolhidas foram as seguintes: *DBLP* (Computer Science Bibliography), *ACM* (Association for Computer Machinery), *Research Gate, Scopus* e *Google Scholar* (https://scholar.google.com). Para a busca foram realizadas combinações e palavras-chave em inglês e o período de abrangência dos trabalhos escolhidos foram os desenvolvidos nos últimos 15 anos.

Para a localização dos trabalhos de interesse a partir das fontes selecionadas anteriormente, foram definidas as seguintes combinações de palavras-chave em inglês:

- NoSQL **AND** Relational;
- JSON **AND** Relational;
- Document **AND** Relational;

- NoSQL **AND** SQL;
- JSON **AND** SQL;
- Document **AND** SQL.

Estas combinações possibilitaram a obtenção máxima possível de referências. Os títulos e resumos dos estudos obtidos foram revisados a fim de realizar a exclusão de temas que não fossem de interesse para a realização deste trabalho. Os estudos escolhidos como fontes de informação relevantes, respeitando assim a pergunta proposta para essa revisão sistemática, foram adquiridos para a criação de uma lista de artigos para análise por este trabalho.

Os critérios para a inclusão dos estudos acadêmicos nesta revisão sistemática foram: (1) devem ser artigos de pesquisa já publicados; (2) tratarem da migração de dados; (3) tratarem da migração da estrutura de BDs; (4) a fonte da migração destes dados e estruturas devem ser BDs orientadas a documentos ou devem ser documentos JSON; (5) o destino da migração destes dados e estruturas devem ser BDs relacionais e; (6) os trabalhos devem apresentar as técnicas utilizadas para a migração destes dados e/ou estruturas dos BDs de uma base para a outra.

Tabela 1 – Critérios de inclusão e exclusão de trabalhos previamente selecionados

Trabalhos correlatos	**Critérios de Inclusão**	**Critérios de Exclusão**	**Resultado**
(WERNER; BACH, 2018)	1, 2, 3, 4, 5, 6		Selecionado
(PETKOVIĆ, 2020)	1, 2, 3, 4, 5, 6		Selecionado
(AFTAB, 2020)	1, 2, 3, 4, 5, 6		Selecionado
(MAITY, 2018)	1, 2, 3, 4, 5, 6		Selecionado
(**Ying-Ti**)	1, 2, 3, 6	4, 5	Não selecionado
(K. O. KENECHUKWU, 2016)	1, 2	3, 4, 5, 6	Não selecionado
(RAMON, 2014)	1, 2, 6	3, 4, 5	Não selecionado
(**Lei**)	1, 6	2, 3, 4, 5	Não selecionado
(R. JULIAN S. L. PHILIPP, 2014)	1, 6	2, 3, 4, 5	Não selecionado
(CHRISTOPH, 2017)	1, 6	2, 3, 4, 5	Não selecionado

A Tabela 1 apresenta o resultado da aplicação dos critérios de inclusão e exclusão. Dos 10 estudos previamente selecionados, apenas 4 cumpriram com todos os requisitos necessários mencionados anteriormente para o desenvolvimento desta revisão sistemática, sendo eles (WERNER; BACH, 2018), (PETKOVIĆ, 2020), (AFTAB, 2020) e (MAITY, 2018). Destes estudos selecionados, 3 realizam uma conversão de BDs orientadas a documentos, como o MongoDB e o CouchDB para BDs relacionais e detalham no texto como é feita essa conversão. Estes trabalhos mencionam as técnicas utilizadas, bem como uma avaliação das mesmas. Estes estudos selecionados são detalhados a seguir.

3.2 VISÃO GERAL DOS ESTUDOS

Segundo (WERNER; BACH, 2018) a motivação que a levou ao desenvolvimento do seu trabalho foi a falta de tecnologias para a aprendizagem de SGBDs do tipo NoSQL interativas. Então foi proposto um sistema onde alunos pudessem realizar buscas em BDs MongoDB e CouchDB. Além disso, ela também realizou a conversão dos documentos contidos nestes BDs para o modelo relacional a fim de identificar aos alunos o equivalente dos documentos em BDs relacionais. Para isso ela utilizou uma técnica que converte documentos em tabelas, sendo que, se existe arrays ou documentos dentro destes documentos, então são criadas tabelas separadas, porém relacionadas através de chaves estrangeiras.

Já (PETKOVIĆ, 2020) converte documentos JSON para o modelo relacional utilizando 2 técnicas diferentes: *STDM* e *AdjacencyList*. A técnica *STDM* mantém a estrutura da árvore JSON original, inserindo registros para cada nodo em uma mesma tabela com relações pai-filho e preservando a ordenação original. Já a AdjacencyList também resulta em apenas uma tabela, onde cada linha representa um elemento do JSON, com colunas para o identificador do elemento juntamente com o identificador do seu elemento pai. A principal diferença entre elas está em como lidam com arrays e documentos dentro de outros documentos. No final, as 2 técnicas geram apenas uma tabela a partir de um documento. A partir da avaliação feita pelo autor, foi possível identificar uma grande vantagem do algoritmo *STDM* contra o *AdjacencyList* tanto em espaço de disco para armazenagem dos dados na tabela final quanto pelo tempo de execução.

Em contrapartida, (AFTAB, 2020) elaborou um método mais complexo, porém mais completo. Seu trabalho consiste na utilização de uma ferramenta OpenSource para a análise do SGBD MongoDB e a extração de seus *schemas* flexíveis. A partir disso, o sistema desenvolvido pelo autor realiza a conversão deste schema para o modelo relacional, transformando documentos em tabelas no BD relacional alvo. Quando existem arrays ou documentos dentro de outros documentos, então tabelas pais e filhas são criadas, e posteriormente chaves primárias e estrangeiras são criadas para realizar a sua relação. O sistema do autor, construído em Javascript, posteriormente, realiza a inserção dos dados contidos no BD MongoDB para o BD relacional alvo. O autor utiliza técnicas de concorrência para a execução de um processo de ETL, além de realizar experimentos em diferentes SGBDs relacionais, como o Postgres e o MySQL.

Já (MAITY, 2018) propôs uma solução semiautomática para a conversão de BDs NoSQL orientados a documentos e BDs NoSQL orientados a grafos para o modelo relacional. Para os BDs orientados a documentos, o BD é analisado e se verifica a presença de atributos com mesmo nome em diferentes documentos. Se não existe, a solução varre a coluna e verifica se todos os valores contidos são únicos. Se sim, é perguntado ao usuário se ele pretende manter essa coluna como chave primária. Se a coluna existe em mais de um lugar no documento, é realizada a junção das 2 colunas em uma só, e então se todos os

valores são únicos, é perguntado ao usuário se ele pretende manter a coluna como chave primária. A solução do autor pode criar mais de 1 tabela caso existam 2 atributos iguais em 2 documentos diferentes, e nestes 2 documentos existam outros atributos diferentes. Porém, este estudo não informa como atributos complexos como arrays são trabalhados, ou seja, estes tipos de atributos não são suportados.

Uma comparação entre os diferentes trabalhos correlatos selecionados pode ser vista na tabela 2.

Tabela 2 – Comparação entre os trabalhos selecionados.

Trabalhos correlatos	**(WERNER; BACH, 2018)**	**(PETKOVIĆ, 2020)**	**(MAITY, 2018)**	**(AFTAB, 2020)**
Suporte a Múltiplas Tabelas	Sim	Não	Sim	Sim
Mapeamento autmático	Sim	Sim	Não	Sim
Identifica relações um-para-um	Não	Não	Sim	Não
Identifica relações um-para-muitos	Sim	Não	Não	Sim
Identifica relações muitos-para-muitos	Sim	Não	Não	Não

3.3 DIFERENCIAL

O estudo de (WERNER; BACH, 2018) apresenta técnicas relevantes para o mapeamento da estrutura de dados do modelo orientado a documentos para o relacional, incluindo a identificação das cardinalidade um-para-muitos e muitos-para-muitos entre as tabelas geradas. Já o presente trabalho identifica os 3 tipos de cardinalidades, um-para-muitos, muitos-para-muitos e um-para-um, possibilitando um modelo mais fiel a estrutura dos dados contidos na origem.

Já o trabalho de (PETKOVIĆ, 2020), apesar de apresentar 2 técnicas diferentes para a conversão, ambas realizam apenas a criação de uma tabela por documento, sendo que se a tabela possui array e documentos contidos dentro, são criadas colunas extras (atributos *ArrayID* e *ObjectID*), além de todas os demais atributos que possam estar contidos dentro do array e documento interno. Esta abordagem não normalizada acaba por gerar duplicação de dados pela falta de relações entre diferentes tabelas. O presente trabalho se destaca por possuir regras para a normalização dos dados, de forma a oferecer uma estrutura capaz de realizar a divisão dos dados em diferentes tabelas relacionadas entre si com a utilização de chaves primárias e estrangeiras, evitando redundâncias.

(MAITY, 2018) propôs uma solução semiautomática, que não leva em consideração tipos de dados complexos contidos em JSON, como por exemplo, arrays. Já (AFTAB,

2020) propôs uma solução muito completa. Ele apresentou uma forma de obter o esquema a partir de um documento e uma solução ETL para a carga de dados em BDs relacionais, onde objetos aninhados e arrays são convertidos em tabelas filhas e relações, onde são criadas chaves estrangeiras apontando para o objeto pai. Apesar disso, as relações entre as tabelas geradas não apresentam uma análise profunda dos dados, resultando em uma solução menos fiel ao modelo relacional, e portanto, uma falta de diferenciação das possíveis cardinalidades que podem estar presentes nos relacionamentos aninhados existentes em documentos JSON.

Como diferencial e contribuição adicional a todos os trabalhos correlatos apresentados, o presente trabalho distribui os atributos complexos presentes nos bancos de dados orientados a documentos em diferentes tabelas de forma a normalizar a estrutura e diminuir a redundância de dados. Além disso ele realiza o mapeamento de forma automática e apresenta regras para a definição de cardinalidade entre as relações das diferentes tabelas geradas a partir do mapeamento de dados complexos oriundos da BD de origem. Dessa forma, este trabalho procura abranger as distintas possibilidades oferecidas pelo modelo relacional e se manter fiel à proposta deste modelo.

4 DOCUMENT2SQL

A partir do estudo do estado da arte e de técnicas de extração e de mapeamento de uma estrutura de um BD orientado a documentos para o modelo relacional, que é o foco deste trabalho, foi realizado o planejamento do projeto proposto. Esta seção traz informações a respeito do planejamento realizado e a implementação da solução denominada *document2sql*, bem como as ferramentas utilizadas e algoritmos produzidos.

4.1 TECNOLOGIAS E FERRAMENTAS

Esta seção descreve as tecnologias e ferramentas utilizadas durante o processo de desenvolvimento da *document2sql*.

4.1.1 Java

Segundo (D. PAUL, 2017), a linguagem de programação Java, surgiu em 1991, como um projeto da empresa *Sun Microsystem* e foi desenvolvida como uma linguagem orientada a objetos cujo objetivo chave, foi o de ser capaz de escrever programas para serem executados em uma grande variedade de dispositivos computacionais.

Com o aumento do acesso à internet na decada de 1990, a *Sun Microsystem* identificou um potencial na linguagem Java para trabalhar dinamicamente com páginas web e animações. Dessa forma, novas funcionalidades foram atribuídas à linguagem para atingir a estes objetivos. Foi também nesta época que o Java chamou a atenção da área de negócios, se tornando também referência de linguagem de programação para o desenvolvimento de aplicativos de grande porte, além de funcionalidades para servidores web, aplicativos para dispositivos móveis, dentre outros (D. PAUL, 2017).

Devido à grande variedade de funcionalidades encontradas no Java, e por se tratar de uma linguagem de programação orientada a objetos, com um bom suporte e ferramentas boas e consolidadas para acesso e manipulação de distintas BDs, ele foi escolhido para este trabalho.

4.1.2 Spring Boot

O *Spring Boot* é um *framework* para desenvolvimento em Java. Ele possui muitas funcionalidades que facilitam todos os aspectos do desenvolvimento, de modo que o desenvolvedor tenha que focar apenas nas regras de negócio do aplicativo a ser desenvolvido.

O *Spring Boot* foi escolhido para este trabalho pois já fornece a abstração necessária para acesso a diferentes BDs como o H2 e o *MongoDB*, entregando funcionalidades já testadas e consolidadas e facilitando o desenvolvimento. Além disso, o Spring Boot também possui uma abstração para o desenvolvimento de serviços *REST*, facilitando assim o desenvolvimento. Por último o *Spring Boot* também possui um servidor de aplicações

embutido, dispensando a configuração de um servidor de aplicações para o desenvolvimento deste trabalho (SPRING..., s.d.).

4.1.3 Banco de Dados H2

O H2 é um BD relacional lançado pela primeira vez em 2005 e feito para ser utilizado em ambientes Java. Pode ser utilizado tanto em memória, quanto para persistir dados em arquivos.

Este BD foi escolhido para este trabalho pela sua integração simples à aplicações desenvolvidas em Java (H2..., s.d.), bem como sua capacidade de persistir dados em arquivos no servidor onde a aplicação está inserida, o que é útil para este trabalho, pois o mesmo tem como objetivo fornecer uma capacidade de armazenamento de dados já embutida na aplicação, de forma a persistir o esquema utilizado como entrada, bem como o mapeamento realizado.

O diagrama de tabelas definido para o BD H2 de forma a armazenar os mapeamentos realizados pode ser vista na figura 8, sendo possível identificar as seguintes entidades:

- JSON_SCHEMA: Persiste o esquema extraído do BD orientado a documentos.
- SQL_SCHEMA: Tabela responsável por relacionar tabelas, colunas e relações à um esquema JSON persistido.
- SQL_TABLE: Persiste tabelas geradas no mapeamento, seu nome, profundidade do nodo dentro esquema do BD orientado a documentos lido e a data e hora de quando o objeto foi criado. Além disso armazena também uma lista de colunas que pertencem a esta tabela.
- SQL_COLUMN: Persiste as colunas geradas pelo mapeamento, incluindo seu nome e propriedades, dentre elas se é chave primária ou chave estrangeira. Além disso está ligada a uma tabela.
- SQL_RELATION: Persiste relações geradas pelo mapeamento, o tipo da relação e as duas tabelas que estão sendo relacionadas, sendo elas a *ORIGIN_TABLE* e a *REFERENCED_TABLE*.
- SQL_TABLE_COLUMNS: Relaciona tabelas a colunas.

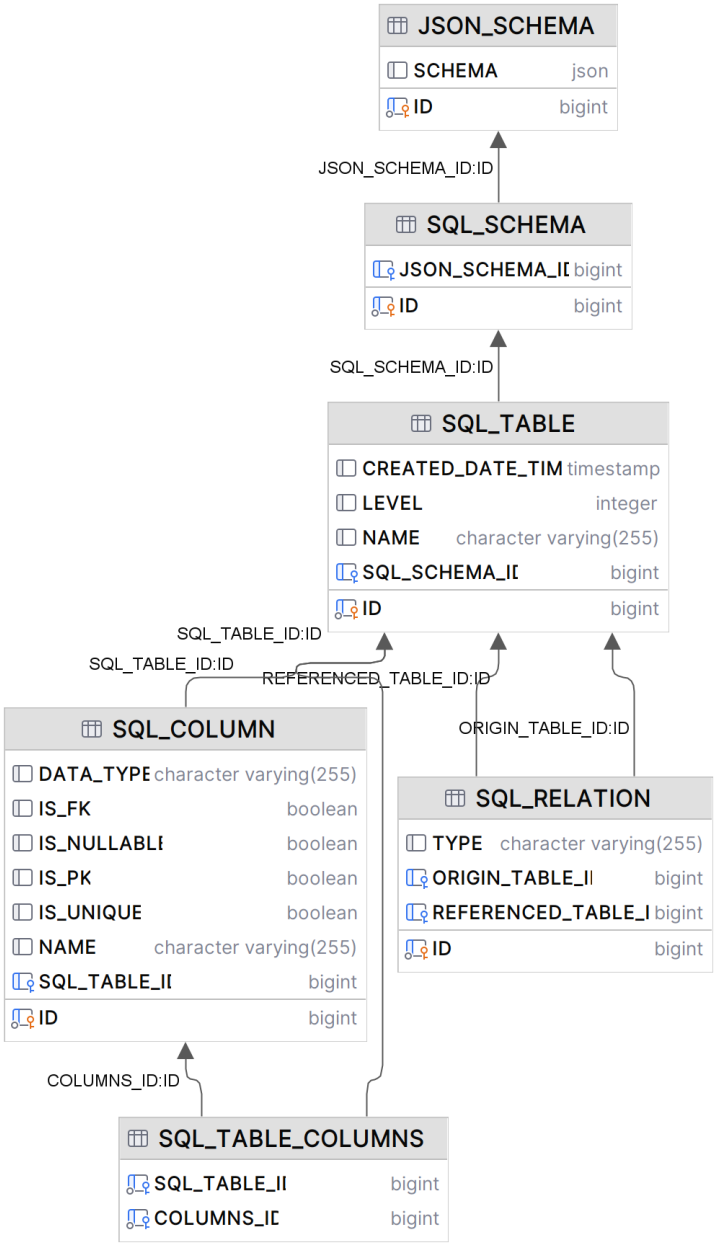


Figura 8 – Diagrama de tabela da aplicação *document2sql* no BD H2.

4.1.4 REST

Representational State Transfer, ou *REST*, foi definido pela primeira vez em 2000 e é um estilo arquitetural para serviços web, ou seja, são utilizados para expor dados e funcionalidades de forma a facilitar interações entre diferentes componentes de software e permitir a troca de dados entre eles (MASSÉ, 2012).

Este estilo arquitetural foi escolhido para este trabalho por permitir que o *document2sql* se integre à demais componentes de software, facilitando assim, seu acoplamento à diferentes cenários.

4.2 PROJETO

O *document2sql* é um serviço Web que pode ser acessado por ferramentas que forneçam suporte ao seu acesso de APIs, como por exemplo, o *Postman* ou o *Insomnia*. A aplicação utiliza o protocolo HTTP e é baseada no modelo REST.

A Figura 9 mostra uma visão geral de como é realizado o processo executado pela solução desenvolvida. Primeiramente é realizada a extração de um esquema único dos documentos armazenados no BD MongoDB de origem, utilizando a ferramenta externa *extract-mongo-schema*[1]. Diferente do estudo de (AFTAB, 2020), a ideia é unificar esquemas de documentos similares para gerar um esquema mínimo representativo de todos os dados dos documentos armazenados.

Os serviços disponíveis na aplicação são os seguintes:

- Leitura do Esquema;
- Mapeamento:
- Geração de código DDL para execução no BD destino.
- Lista esquemas JSON inseridos na aplicação.

É possível identificar na figura 9 que o serviço de mapeamento executa internamente três diferentes processos, sendo eles:

- Mapeamento de tabelas e colunas;
- Leitura dos dados;
- Definição de relações e cardinalidades;

Com o esquema gerado, procede-se a inserção na aplicação do esquema gerado, do esquema de documentos para o esquema relacional a partir de um conjunto de regras de mapeamento. Algumas dessas regras levam em consideração tipos complexos dentro dos documentos analisados e a melhor forma de adequá-los a dados contidos ao modelo relacional tendo como objetivo a normalização de sua estrutura. Esta é a principal contribuição deste trabalho.

Com o esquema relacional definido, são gerados os comandos DDLs necessários para a sua execução no SGBD relacional NewSQL CockroachDB com o objetivo de realizar a criação da sua estrutura.

De forma a seguir com as etapas do projeto foram levantados requisitos funcionais (RF) que descrevem o que o sistema deve fazer, suas ações, comportamentos e serviços. Também foram levantados requisitos não funcionais (RNF), que não estão diretamente

[1] https://github.com/perak/extract-mongo-schema

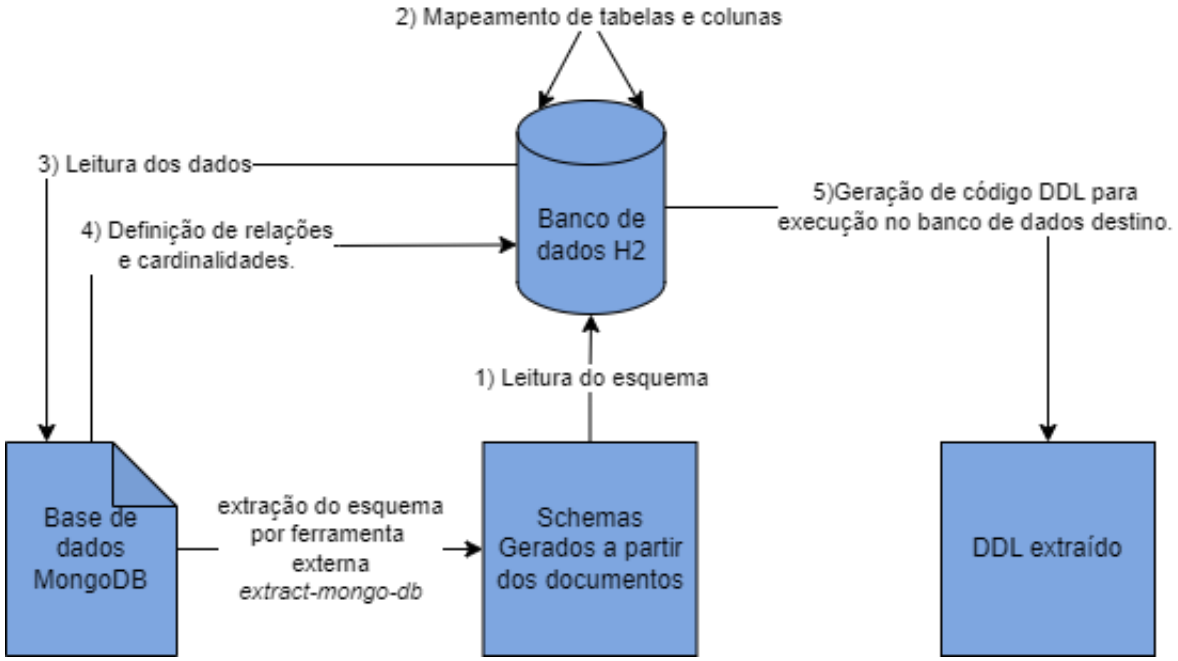


Figura 9 – Processo executado pelo *document2sql*.

ligadas às ações e serviços do sistema, mas dizem respeito à qualidade do sistema em si. Os RF são:

- RF01 - Inserir novo esquema JSON;
- RF02 - Executar mapeamento;
- RF03 - Exportar DDL gerado;
- RF05 - Listar esquemas JSON.

Já os RNF são:

- RNF01 - Arquitetura REST;
- RNF02 - BD H2;
- RNF03 - Linguagem de programação Java - versão 20;
- RNF04 - Framework web Spring Boot.

4.3 ESQUEMA UTILIZADO NO MAPEAMENTO

A escolha do processo extração do esquema JSON do MongoDB levou em consideração alguns fatores:

- Deve unificar em um JSON todas as coleções da BD;

- Deve agrupar todas as similaridades entre objetos, de forma a minimizar o esquema obtido;
- Deve trabalhar com estruturas de dados complexas como vetores e objetos aninhados.

A partir destes fatores levantados, foi dado início a um processo de pesquisa por ferramentas capazes de suprir estas necessidades. Foi então descoberta a *extract-mongo-schema*[2], biblioteca escrita na linguagem de programação *Javascript*, que inserida globalmente em uma máquina possuindo um servidor *NodeJs*, permite a extração do esquema. O resultado gerado se trata de um documento JSON contendo todas as coleções do BD selecionado e suas estruturas.

A Figura 10 mostra um exemplo que detalha o esquema gerado da coleção "zips". Este documento possui várias chaves, dentre elas, "_id", "city", "zip", "loc", "pop"e "state".

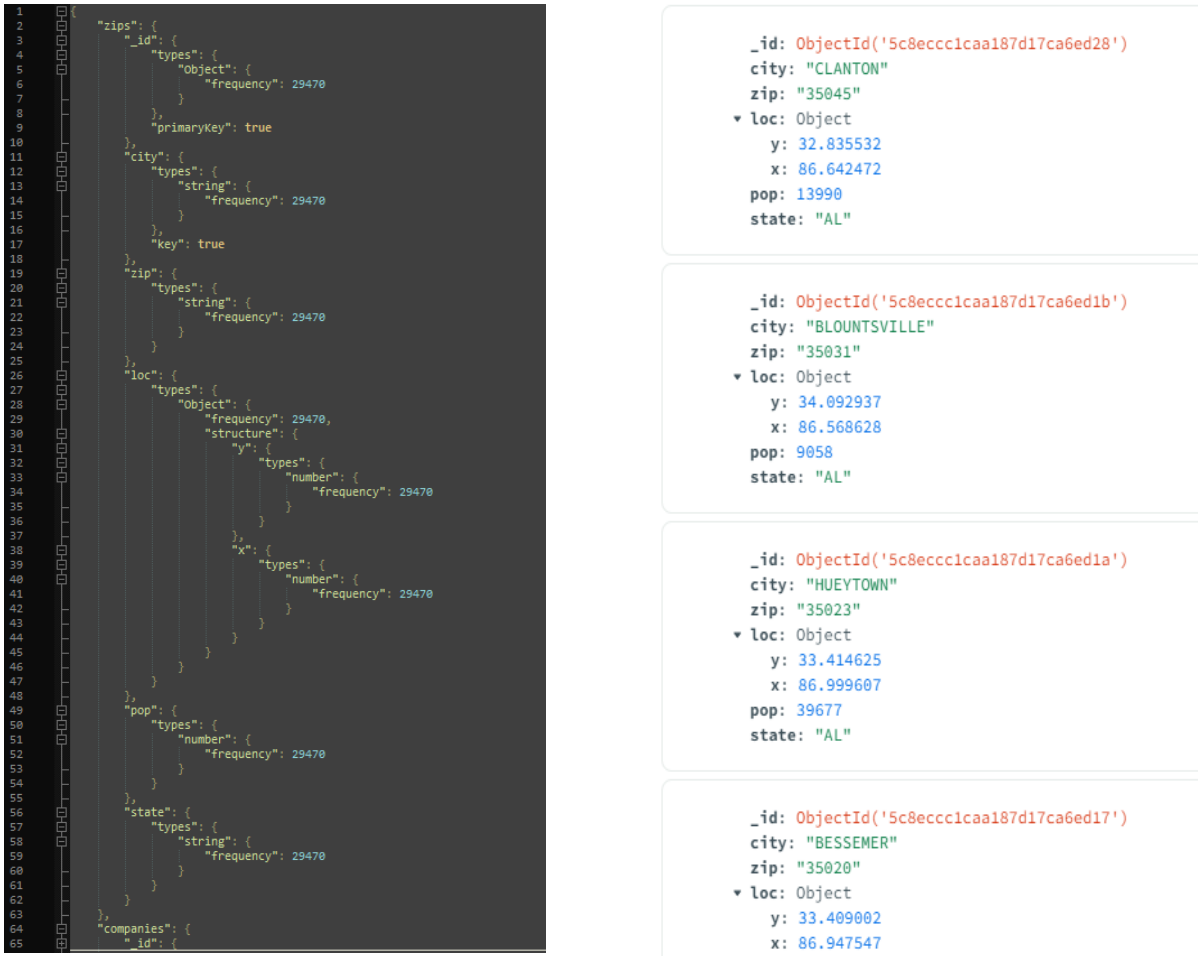


Figura 10 – À esquerda, esquema extraído pela *extract-mongo-schema* e à direita a documentos pertencentes à coleção de onde foram extraídos.

É possível notar, no esquema gerado, que cada uma dessas chaves abre em um novo documento aninhado, onde há o seu tipo de dado e a frequência com que aparece

[2] https://github.com/perak/extract-mongo-schema

dentro dos documentos "zips"no BD, além de outros possíveis atributos, como por exemplo, "primaryKey"e "key". Um detalhe a se levar em consideração são os tipos "Object", que podem representar campos "_id"que são as chaves primárias de um documento padrão no mongoDB, como também podem representar documentos aninhados, como é possível observar na chave "loc"do documento raiz, onde seu valor é outro documento, que possui, por sua vez, outras 2 chaves do tipo number, "x"e "y", com uma frequência de 29470 cada. Também na Figura 10 podemos observar logo abaixo da estrutura da coleção "zips"o início de outra coleção do mesmo BD, sendo ela, a "companies."

4.4 MAPEAMENTO DO MODELO BASEADO EM DOCUMENTOS PARA O RELACIONAL

Nesta etapa, as informações do esquema do modelo de documentos selecionado são extraídas e regras são aplicadas de forma a identificar possíveis tabelas, colunas, limitações, relações e suas cardinalidades no esquema a ser gerado para a criação de uma estrutura relacional correspondente.

É possível dividir este processo em subprocessos. No primeiro subprocesso analisa-se o esquema do modelo de documentos e identifica-se tabelas, colunas e limitações a serem criadas no BD destino. Já, no segundo subprocesso acessa-se o BD no MongoDB para identificar a cardinalidade entre as relações das tabelas geradas no primeiro subprocesso.

4.4.1 Identificação de Tabelas, Colunas e Limitações

A primeira parte deste processo é realizar a leitura do esquema do BD orientado a documentos. Para cada coleção contida no esquema é gerada uma tabela no esquema relacional. Feito isso, para cada uma dessas coleções, são verificadas suas chaves, que caso sejam de tipos simples, como string ou numérico, são definidas colunas para a tabela a qual pertencem. Além disso, é verificado também se a chave possui atributos de limitação no esquema de documentos. Caso possua a chave "key"verdadeira, como é possível observar na chave "city"dentro da coleção "zips", exemplificada na Figura 10, essa coluna é definida como única na tabela de dados destino, ou seja, nesta tabela não é permitida a persistência de 2 registros com dados de mesmo valor para esta coluna.

Já quando os documentos possuem chaves do tipo "Object"ou "Array", que são tipos de dados complexos, uma outra abordagem é adotada. Aqui, em ambos os casos, identifica-se uma relação entre as tabelas. Sendo assim, tendo como exemplo para "Object"a chave "loc"contida dentro da coleção "zips"na Figura 10, é criada uma nova tabela chamada "loc", onde seus atributos são, por sua vez, analisados, e como é possível observar, são criadas outras 2 colunas "y"e "x"de tipos numéricos.

Três algoritmos descrevem esse subprocesso. O Algoritmo 1 a seguir lê o esquema de documentos escolhido pelo usuário na linha 2 e, para cada uma das coleções presentes

nele, uma nova tabela é criada e passada como parâmetro para o Algoritmo 2 na linha 9 juntamente com o nodo JSON referente à coleção que é extraído na linha 8.

Algorithm 1 Inicia mapeamento para o esquema relacional

```
 1: function mapToSql(jsonSchemaId)
 2:     jsonSchema ← getJsonSchemaById(jsonSchemaId);
 3:     node ← getSchemaJsonNodeInJsonSchema(jsonSchema);
 4:     for each tableName in node.fieldNames do
 5:         tableName ← replaceSpacesWithUnderscores(tableName);
 6:         sqlTable ← newSqlTable(tableName);
 7:         sqlTable.setCreatedDateAndTime(newDate());
 8:         tableNode ← node.get(tableName);
 9:         createSqlObjects(tableNode, sqlTable);
10:         dynamicMongoService.analyzeCollectionCardinality(tableName)
11:     end for
12: end function
```

O Algoritmo 2 realiza a criação de uma chave primária do tipo UUID para a tabela. Feito isso, ele verifica todas as chaves presentes no documento juntamente com o seu tipo, e, caso não seja um "Object"ou "Array", uma nova coluna é criada e associada à tabela passada como parâmetro e identificada como *parentTable*.

Algorithm 2 Cria objetos no esquema relacional a partir do documento

```
1: function createSqlObjects(parentNode, parentTable)
2:     if not parentTable.getColumns().stream().anyMatch(SqlColumn :: isIsPk) then
3:         primaryKey ← newSqlColumn;
4:         primaryKey.setName(parentTable.getName() + "_gen_uuid");
5:         primaryKey.setIsPk(true);
6:         primaryKey.setDataType("UUID");
7:         primaryKey.setSqlTable(parentTable);
8:         parentTable.setColumn(primaryKey);
9:     end if
10:    for each fieldName in schema.fieldNames do
11:        fieldNameNode ← schema.get(fieldName);
12:        if fieldName ≠ "_id" then
13:            nodeType ← getNodeObjectType(fieldNameNode);
14:            type ← getStringObjectType(nodeType);
15:            if type ≠ OBJECT & type ≠ ARRAY & type ≠ "Null" then
16:                column ← newSqlColumn;
17:                setColumnAttributes(column, fieldNameNode, fieldName, type);
18:                parentTable.setColumn(column);
19:            else if type == OBJECT then
20:                childTable ← newSqlTable(parentTable.getName() + "___" +
   fieldName.replace(, "_"));
21:                childNodeStructure ← getObjectStructure(fieldNameNode)
22:                createSqlObjects(childNodeStructure, childTable);
23:            else if type == ARRAY then
24:                childArrayTable ← newSqlTable(parentTable.getName() + "___" +
   fieldName.replace(, "_"));
25:                arrayStructure ← getArrayStructure(fieldNameNode);
26:                createSqlObjectsArray(arrayStructure, childArrayTable)
27:            end if
28:        end if
29:    end for
30:    sqlTableRepository.save(parentTable)
31: end function
```

Caso a chave tenha um tipo "Object", o algoritmo realiza a criação de uma *childTable* com nome da tabela pai, concatenado com "___"e concatenado com o nome da chave. Após, ele obtém o valor desta chave, que é um nodo JSON e chama novamente o Algoritmo 2, passando como parâmetro o nodo da *childTable* como o novo esquema a ser analisado, e a *childTable* criada como sendo a nova *parentTable*.

Caso a chave possua um tipo "Array", é criada uma nova *childTable* onde o esquema da estrutura deste array é obtida do nodo passado anteriormente como parâmetro, e passada como parâmetro para o Algoritmo 3, como um novo nodo JSON. Além disso, a *childTable* também é passada como o parâmetro *parentTable*.

Algorithm 3 Cria objetos no esquema relacional a partir de array

```
1: function createSqlObjectsArray(parentNode, parentTable)
2:     primaryKey ← newSqlColumn;
3:     primaryKey.setName(parentTable.getName() + "_gen_uuid")
4:     primaryKey.setIsPk(true);
5:     primaryKey.setDataType("UUID");
6:     primaryKey.setSqlTable(parentTable);
7:     parentTable.setColumn(primaryKey);
8:     for each type in schema.fieldNames do
9:         if type ≠ OBJECT & type ≠ ARRAY & type ≠ "Null" then
10:            column ← newSqlColumn;
11:            column.setName(parentTable.getName().replace(, "_") + type);
12:            if type == "number" then
13:                column.setDataType("float")
14:            else
15:                column.setDataType(type)
16:            end if
17:            column.setSqlTable(parentTable);
18:            parentTable.setColumn(column)
19:        end if
20:        if type == OBJECT then
21:            structure ← getArrayObjectStructure(schema);
22:            createSqlObjects(structure, parentTable)
23:        end if
24:    end for
25:    sqlTableRepository.save(parentTable)
26: end function
```

O Algoritmo 3 repete o processo do algorimo anterior, realizando a criação de uma chave primária para a nova tabela e percorrendo seus atributos de forma a identificar seus tipos. Caso o tipo de uma chave não seja um "Object", "Array"ou nulo, uma nova coluna é criada. Caso a chave seja do tipo "Object", a estrutura desta chave é obtida e passada como parâmetro para o Algoritmo 2 como um nodo JSON e a própria *parentTable* desta vez é passada como parâmetro *parentTable*, diferentemente do Algoritmo 2.

Os algoritmos trabalham recursivamente de forma a não limitar o número de objetos aninhados que possam estar presentes dentro de uma mesma coleção. Porém, uma limitação é que ele não executa vetores dentro de vetores. Desta forma, transformações devem ser realizadas na origem dos dados para evitar este tipo de cenário.

4.4.2 Identificação das Relações e Cardinalidade entre as Tabelas

O segundo subprocesso realiza uma análise dos dados na origem (BD MongoDB) de forma a identificar relacionamentos e suas cardinalidades.

Para o desenvolvimento deste trabalho foram definidas regras a serem aplicadas no BD de origem (MongoDB) de forma a mapear os relacionamentos e e suas cardinalidades entre seus diferentes objetos aninhados e arrays. As regras definidas para cada tipo de relacionamento são as seguintes:

1. Um-para-um:

a) ocorre quando um subobjeto não se repete em diferentes objetos pais, que por sua vez, não possuem mais de um subobjeto;

2. Um-para-muitos:

a) ocorre quando um subobjeto está presente em diferentes objetos pais, que por sua vez, não possuem mais de um subobjeto;

b) ocorre quando um objeto pai possui diferentes subobjetos, porém os subobjetos não possuem mais de um objeto pai;

c) ocorre quando um objeto em um array de objetos não se repete em diversos arrays;

3. Muitos-para-muitos:

a) ocorre quando um objeto em um array de objetos se repete em diversos arrays;

b) ocorre quando um subobjeto se repete para diferentes objetos pais e os objetos pais possuem diferentes subobjetos;

c) ocorre quando um valor em um array que não é de objetos se repete em diversos arrays.

A aplicação dessas regras é detalhada nos algoritmos a seguir. O subprocesso inicia pelo Algoritmo 4. Ele recebe como parâmetro um nome de coleção. Após, o BD MongoDB é acessado, de onde toda a coleção e seus documentos são retornados e analisados.

Algorithm 4 Inicia análise da cardinalidade

```
1: function analyzeCardinality
2:     database ← mongoTemplate.getDb()
3:     collection ← database.getCollection(collectionName)
4:     /*Inicializa HashMaps para estruturas de subdocumentos, arrays e ocorrências de documentos*/
5:     subDocumentStructureOccurrences ← newHashMap;
6:     arrayStructureOccurrences ← newHashMap;
7:     parentDocumentOccurrences ← newHashMap;
8:     parentDocumentOccurrencesPerFullKey ← newHashMap;
9:     documents ← collection.find()
10:    /*Chama o método analyzeDocument para cada documento*/
11:    for each document in documents do
12:        analyzeDocument(document
13:        , subDocumentStructureOccurrences
14:        , arrayStructureOccurrences
15:        , parentDocumentOccurrences
16:        , parentDocumentOccurrencesPerFullKey
17:        , null)
18:    end for
19:    identifyObjectCardinality(subDocumentStructureOccurrences,
   parentDocumentOccurrences, parentDocumentOccurrencesPerFullKey);
20:    identifyArrayCardinality(arrayStructureOccurrences)
21: end function
```

Para cada objeto presente na coleção, o Algoritmo 5 é chamado e o objeto é passado como parâmetro, juntamento com as estruturas de dados necessárias para a identificação de repetições de objetos. Neste algoritmo é realizada uma iteração por todas as chaves e seus respectivos valores. Para cada valor, alguns conjuntos de regras são aplicados. Se for do tipo objeto, então caso possua objetos aninhados e arrays, estes são removidos utilizando o Algoritmo 12 e, com o restante das suas chaves e valores, é gerado um texto e adicionado à um *HashMap*, somando o número de ocorrências, deste mesmo texto, para todas as ocorrências deste objeto. O mesmo é realizado com o objeto pai.

Após a análise de ocorrências de objetos e valores dentro de arrays, a execução retorna para o Algoritmo 4, que por sua vez chama os Algoritmos 6 e 7, que realizam a análise das estruturas de dados alimentadas e aplicam as regras as cardinalidades para os objetos e arrays, respectivamente.

Algorithm 6 Identifica a Cardinalidade dos Objetos e Subobjetos

```
1: function identifyObjectCardinality(subDocumentStructureOccurrences,
   parentDocumentOccurrences, parentDocumentOccurrencesPerFullKey)
2:     for each (key, value) in document do
3:         for each (field, structureMap) in subDocumentStructureOccurrences do
4:             for each structureHash in structureMap.keySet() do
5:                 parentDocs ← parentDocumentOccurrences.get(structureHash)
6:                 if parentDocs ≠ null & parentDocs > 1 then
7:                     if structureMap.values.anyMatch(count > 1)
8:                         & parentDocumentOccurrencesPerFullKey.get(field).values.anyMatch(count >
   1) then
9:                         createManyToMany(field, collectionName)
10:                    else
11:                        createManyToOne(field, collectionName)
12:                    end if
13:                else if parentDocs ≠ null & parentDocs == 1 then
14:                    if structureMap.values.anyMatch(count > 1)
15:                        & parentDocumentOccurrencesPerFullKey.get(field).values.allMatch(count ==
   1) then
16:                        createOneToMany(field, collectionName)
17:                    else
18:                        createOneToOne(field, collectionName)
19:                    end if
20:                end if
21:            end for
22:        end for
23:
```

Algorithm 7 Identifica a Cardinalidade dos Arrays

```
1: function identifyArrayCardinality(arrayStructureOccurrences)
2:     for each (field, structureMap) in arrayStructureOccurrences do
3:         if structureMap.values().anyMatch(count > 1) then
4:             createManyToMany(field, collectionName)
5:         else
6:             createOneToMany(field, collectionName)
7:         end if
8:     end for
9: end function=0
```

Algorithm 5 Conta ocorrências de objetos nos documentos

```
1: function ANALYZEDOCUMENT(Object, subDocumentStructureOccurrences,
   arrayStructureOccurrences, parentDocumentOccurrences, parentDocumentOccurrencesPerFullKey,
   fullKey)
2:     for each (key, value) in document do
3:         fullKey ← empty
4:         if parentKey == null then
5:             fullKey ← key
6:         else
7:             fullKey ← panretKey.key
8:         end if
9:         documentHash ← generateTopLevelHash(document)
10:        if value.instanceOfObject then
11:            structureHash ← generateTopLevelHash(value)
12:            /*Soma um para cada ocorrência do subobjeto*/
13:            subDocumentStructureOccurrences
14:            .computeIfAbsent(fullKey, k− > newHashMap)
15:            .merge(structureHash, 1, sum)
16:            /*Soma um para cada vez que o objeto pai*/
17:            parentDocumentOccurrencesPerFullKey
18:            .computeIfAbsent(fullKey, k− > newHashMap)
19:            .merge(documentHash, 1, sum)
20:            /*Adiciona ocorrências de objetos pais para este subobjeto*/
21:            parentDocumentOccurrences
22:            .computeIfAbsent(structureHash, k− > newHashSet)
23:            .add(documentHash)
24:            /*Chama novamente o analyzeDocument passando o subobjeto, as estruturas de dados
   e e a chave*/
25:            analyzeDocument(value, subDocumentStructureOccurrences
26:            , arrayStructureOccurrences, parentDocumentOccurrences
27:            , parentDocumentOccurrencesPerFullKey, fullKey)
28:        else if valueinstanceOfArray then
29:            for iteminList do
30:                if item.instanceOf ≠ Array & item.instanceOf ≠ Object then
31:                    structureHash ← generateArrayHash(item)
32:                    /*Soma um para cada vez que o valor se repete*/
33:                    arrayStructureOccurrences
34:                    .computeIfAbsent(fullKey, k− > newHashMap)
35:                    .merge(structureHash, 1, sum)
36:                else if item.instanceofDocument then
37:                    structureHash ← generateTopLevelHash(item);
38:                    /*Soma um para cada vez que o subobjeto se repete*/
39:                    arrayStructureOccurrences
40:                    .computeIfAbsent(fullKey, k− > newHashMap)
41:                    .merge(structureHash, 1, sum)
42:                    /*Chama novamente o analyzeDocument passando o subobjeto, as estruturas de
   dados e e a chave*/
43:                    analyzeDocument(item
44:                    , subDocumentStructureOccurrences...parentDocumentOccurrencesPerFullKey
45:                    , fullKey)
46:                end if
47:            end for
48:        end if
49:    end for
50: end function
```

Após a identificação das cardinalidades, os Algoritmos 8, 9, 10 e 11 são chamados para realizar a criação dos objetos no BD H2, que representam as relações entre as estruturas geradas pelo mapeamento, bem como suas chaves, e regras definidas, como por exemplo, a criação da tabela de associação para casos *muitos-para-muitos* ou a definição de chave estrangeira na chave primária de tabelas filhas para casos *um-para-muitos*.

Algorithm 8 Cria Relações Muitos-Para-Muitos

```
1: function createManyToMany(field, collectionName)
2:     parentTableName ← empty
3:     childTableName ← empty
4:     /*Verifica se é a raiz da coleção e identifica o nome das tabelas*/
5:     if field.contains(".") then
6:         parentTableName ← collectionName+"___"+field.substring(0, field.lastIndexOf(".")).replace(".", "___")
7:         childTableName ← collectionName + "___" + field.replace(".", "___")
8:     else
9:         parentTableName ← collectionName;
10:        childTableName ← collectionName + "___" + field.replace(".", "___");
11:    end if
12:    associationTable ← newSqlTable(parentTableName + "___R___" + childTableName)
13:    primaryKey ← newSqlColumn;
14:    primaryKey.setName(parentTable.getName() + "_gen_uuid");
15:    primaryKey.setIsPk(true);
16:    primaryKey.setDataType("UUID");
17:    primaryKey.setSqlTable(associationTable);
18:    associationTable.setColumn(primaryKey);
19:    parentTable ← sqlTableRepository.findByName(parentTableName);
20:    childTable ← sqlTableRepository.findByName(childTableName);
21:    parentRelation ← newSqlRelation();
22:    parentRelation.setOriginTable(associationTable);
23:    parentRelation.setReferencedTable(parentTable);
24:    parentRelation.setType("M − N");
25:    childRelation ← newSqlRelation();
26:    childRelation.setOriginTable(associationTable);
27:    childRelation.setReferencedTable(childTable);
28:    childRelation.setType("M − N");
29:    fkParentTable ← newSqlColumn();
30:    fkParentTable.setName(parentTable.getName() + "_id");
31:    fkParentTable.setFk(true);
32:    fkParentTable.setDataType("UUID");
33:    fkParentTable.setSqlTable(associationTable);
34:    associationTable.setColumn(fkParentTable);
35:    fkChildTable ← newSqlColumn();
36:    fkChildTable.setName(childTable.getName() + "_id");
37:    fkChildTable.setFk(true);
38:    fkChildTable.setDataType("UUID");
39:    fkChildTable.setSqlTable(associationTable);
40:    associationTable.setColumn(fkChildTable);
41:    sqlRelationRepository.save(parentRelation);
42:    sqlRelationRepository.save(childRelation)
43: end function
```

Algorithm 9 Cria Relações Muitos-Para-Um

```
1: function CreatesManyToOne(field, collectionName)
2:     parentTableName ← empty
3:     childTableName ← empty
4:     /*Verifica se é a raiz da coleção e identifica o nome das tabelas*/
5:     if field.contains(".") then
6:         childTableName ← collectionName+"___"+field.substring(0, field.lastIndexOf(".")).replace(".", "___")
7:         parentTableName ← collectionName + "___" + field.replace(".", "___")
8:     else
9:         childTableName ← collectionName;
10:        parentTableName ← collectionName + "___" + field.replace(".", "___");
11:    end if
12:    parentTable ← sqlTableRepository.findByName(parentTableName);
13:    childTable ← sqlTableRepository.findByName(childTableName);
14:    sqlRelation ← newSqlRelation();
15:    sqlRelation.setOriginTable(childTable);
16:    sqlRelation.setReferencedTable(parentTable);
17:    sqlRelation.setType("N − 1");
18:    foreignKey ← newSqlColumn();
19:    foreignKey.setName(parentTable.getName() + "_id");
20:    foreignKey.setFk(true);
21:    foreignKey.setDataType("UUID");
22:    foreignKey.setSqlTable(childTable);
23:    childTable.setColumn(foreignKey);
24:    sqlRelationRepository.save(sqlRelation);
25:    sqlTableRepository.save(childTable)
26: end function
```

Algorithm 10 Cria Relações Um-Para-Muitos

```
1: function CreatesOneToMany(field, collectionName)
2:     parentTableName ← empty
3:     childTableName ← empty
4:     /*Verifica se é a raiz da coleção e identifica o nome das tabelas*/
5:     if field.contains(".") then
6:         parentTableName ← collectionName+"___"+field.substring(0, field.lastIndexOf(".")).replace(".", "___")
7:         childTableName ← collectionName + "___" + field.replace(".", "___")
8:     else
9:         parentTableName ← collectionName;
10:        childTableName ← collectionName + "___" + field.replace(".", "___");
11:    end if
12:    parentTable ← sqlTableRepository.findByName(parentTableName);
13:    childTable ← sqlTableRepository.findByName(childTableName);
14:    sqlRelation ← newSqlRelation();
15:    sqlRelation.setOriginTable(childTable);
16:    sqlRelation.setReferencedTable(parentTable);
17:    sqlRelation.setType("N − 1");
18:    foreignKey ← newSqlColumn();
19:    foreignKey.setName(parentTable.getName() + "id");
20:    foreignKey.setFk(true);
21:    foreignKey.setDataType("UUID");
22:    foreignKey.setSqlTable(childTable);
23:    childTable.setColumn(foreignKey);
24:    sqlRelationRepository.save(sqlRelation);
25:    sqlTableRepository.save(childTable)
26: end function
```

Algorithm 11 Cria Relações Um-Para-Um

```
1: function CreatesOneToOne(field, collectionName)
2:     parentTableName ← empty
3:     childTableName ← empty
4:     /*Verifica se é a raiz da coleção e identifica o nome das tabelas*/
5:     if field.contains(".") then
6:         parentTableName ← collectionName+"___"+field.substring(0, field.lastIndexOf(".")).replace(".", "___")
7:         childTableName ← collectionName + "___" + field.replace(".", "___")
8:     else
9:         parentTableName ← collectionName;
10:        childTableName ← collectionName + "___" + field.replace(".", "___");
11:        parentTable ← sqlTableRepository.findByName(parentTableName);
12:        childTable ← sqlTableRepository.findByName(childTableName);
13:        sqlRelation ← newSqlRelation();
14:        sqlRelation.setOriginTable(childTable);
15:        sqlRelation.setReferencedTable(parentTable);
16:        sqlRelation.setType("1 − 1");
17:        for each columninchildTable.getColumns do
18:            /*Seta a chave primária da tabela filha como a chave estrangeira*/
19:            if column.isPk() then
20:                column.setIsFk(true);
21:                sqlColumnRepository.save(column)
22:            end if
23:        end for
24:        sqlRelationRepository.save(sqlRelation);
25:        sqlTableRepository.save(childTable)
26:
```

O Algoritmo 5 trabalha de forma similar com os vetores, porém, utilizando o Algoritmo 12 para realizar a limpeza dos objetos e vetores aninhados. Com a execução do Algoritmo 5, retorna-se ao Algoritmo 4, onde as estruturas de dados geradas são analisadas de acordo com as regras definidas para o mapeamento de cardinalidades.

Algorithm 12 Exclui vetores e documentos aninhados de documentos e gera hash apenas com atributos simples.

```
1: function generateTopLevelHash(document)
2:     ret ← empty;
3:     for docindocument do
4:         if doc.instaceOf ≠ Object & doc.instaceOf ≠ Array then
5:             ret ← ret + doc
6:         end if
7:     end for
8:     return retornullifempty
9: end function
10: function generateArrayHash(array)
11:     ret ← empty;
12:     for docindocument do
13:         if doc.instaceOf ≠ Object & doc.instaceOf ≠ Array then
14:             ret ← ret + doc
15:         end if
16:     end for
17:     return retornullifempty
18: end function=0
```

De acordo com cada regra, objetos da relação são criados relacionando a tabela pai e a tabela filha, armazenando também o tipo de sua relação. A exceção é a relação muitos para muitos, onde uma terceira tabela é criada (tabela associativa), que possui como coluna sua chave primária e mais 2 chaves estrangeiras que fazem referência às tabelas definidas anteriormente.

4.5 EXTRAÇÃO DAS INSTRUÇÕES DDL

A última etapa realizada pela ferramenta é a geração do código SQL DDL a ser executado no BD relacional de destino, respeitando a sintaxe do BD escolhido, que, neste caso, é o BD relacional NewSQL CockroachDB. Esse código DDL é responsável pela criação do BD relacional.

Os algoritmos definidos exploram as tabelas, colunas e relacionamos identificados nas etapas anteriores para realizar a criação do BD relacional no CockroachDB. O processo inicia com o Algoritmo 13, onde todas as tabelas são retornadas do BD H2 e percorridas para a geração do código *DDL* com a execução do Algoritmo 14, responsável pelo geração de tabelas, suas colunas e seus relacionamentos.

Algorithm 13 Retorna o esquema DDL

```
1: function getSqlSchemaDdl
2:     tables ← findAll(sqlTableRepository);
3:     tableRelations ← mapTableRelations;
4:     ddl ← new StringBuilder();
5:     for all table in tables do
6:         tableDdl ← createTableDdl(table, tableRelations.get(table.getId()));
7:         ddl.append(tableDdl);
8:         ddl.append(\n)
9:     end for
10:    return ddl.toString
11: end function
```

Na sequência, a instrução *DDL* gerada para uma tabela é retornada ao Algoritmo 13, sendo adicionado a uma *String* de instruções *DDL*. Ao final, o conjunto de instruções DDL para todas as tabelas é retornado ao usuário.

Algorithm 14 Gera os comandos DDL

```
 1: function createTableDdl(table, relations)
 2:     ddl ← new StringBuilder();
 3:     ddlIndex ← new StringBuilder();
 4:     Initialize variables for foreign key and unique columns;
 5:     ddl.Append("CREATETABLE");
 6:     for all column in table.getColumns do
 7:         ddl.Append("column");
 8:         if column.isPk == true then
 9:             ddl.Append("PRIMARYKEY")
10:         end if
11:         if column.isFk == true then
12:             ddl.Append("FOREIGNKEY")
13:             ddlIndex.Append("Index")
14:         end if
15:     end for
16:     ddl ← ddl + ddlIndex
17:     return ddl.toString
18: end function
```

5 DEMONSTRAÇÃO DE USO DA DOCUMENT2SQL

Este capítulo apresenta a demonstração dos esquemas do modelo relacional gerados pela ferramenta *Document2SQL*. Essa demonstração tem como objetivo apresentar a geração das cardinalidades e estruturas no CockroachDB a partir das regras de mapeamento definidas por este trabalho. Sendo assim, foram criadas no MongoDB as seguintes coleções de um BD *exemplo* (ver Figura 11), de forma a validar as diferentes regras de mapeamento: *pessoas*, *professores*, *biblioteca*, *disciplinas*, *artigos*, *situacoes* e *trabalhos*. Foram também criados documentos para cada uma destas coleções para fins de estudo de caso e validação dos cenários de mapeamento.

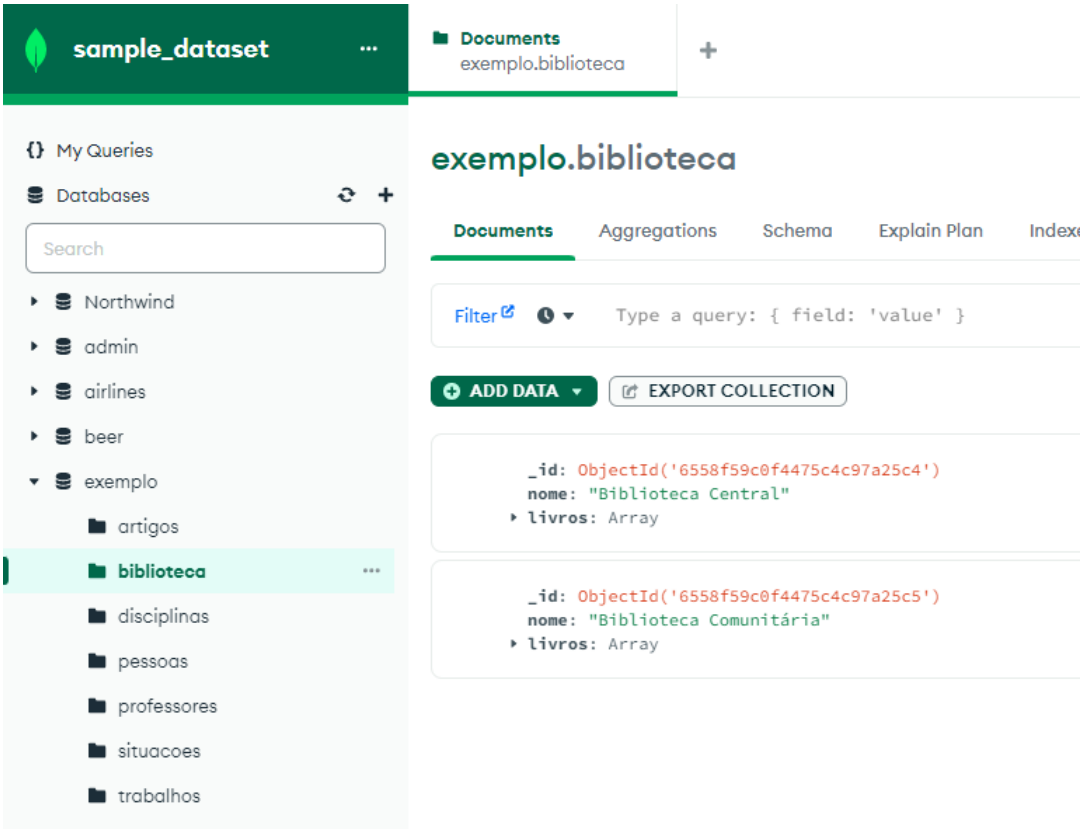


Figura 11 – BD e coleções presentes em um servidor MongoDB.

5.1 PESSOAS

A coleção pessoas, que pode ser visualizada na Figura 12, é um exemplo de relação *um-para-um*, pois o objeto dentro de passaporte não se repete para as demais pessoas.

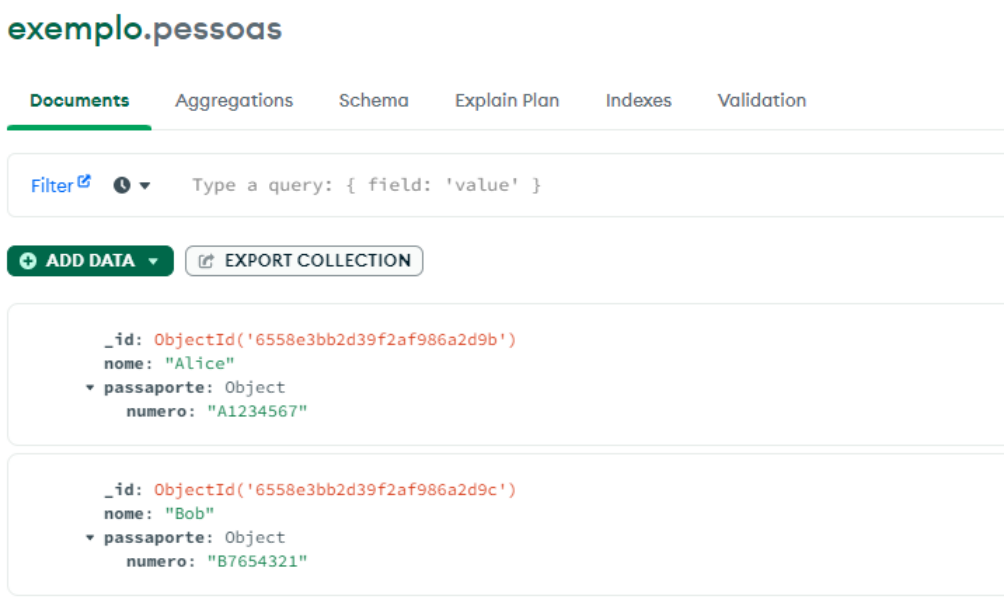


Figura 12 – Coleção *pessoas* no BD MongoDB.

De forma a implementar esta relação no BD destino, foi realizada a criação de uma chave estrangeira na tabela filha *pessoas___passaporte* a partir da chave primária da mesma tabela *passaporte*, que referencia a tabela pai *pessoas*. O diagrama com essa relação pode ser visto na Figura 13 e a estrutura gerada pode ser vista na Figura 14.

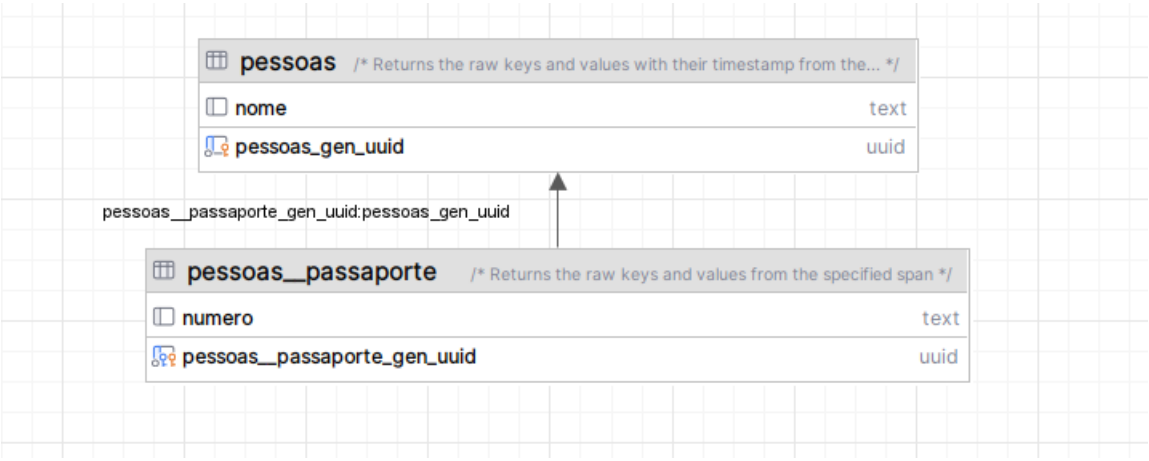


Figura 13 – Diagrama de tabela da tabela pai *pessoas* e tabela filha *pessoas___passaporte*.

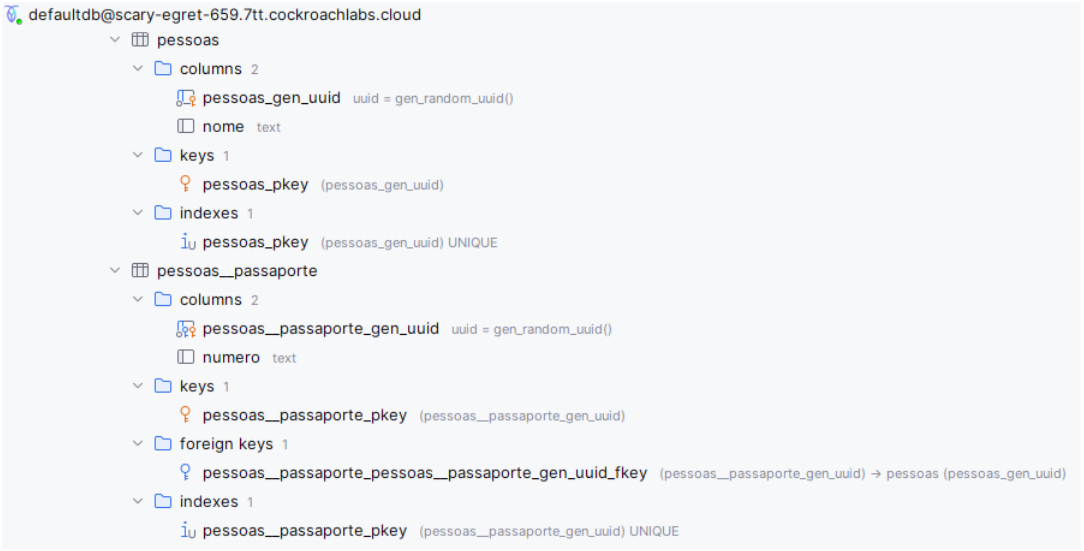


Figura 14 – Estrutura das tabelas *pessoas* e *pessoas___passaporte* geradas no CockroachDB.

5.2 PROFESSORES

A coleção *professores*, visualizada na Figura 16, é um exemplo de relação *um-para-muitos*, pois nenhum objeto pertencente ao vetor de *disciplinas* se repete em outro documento de *professores*.

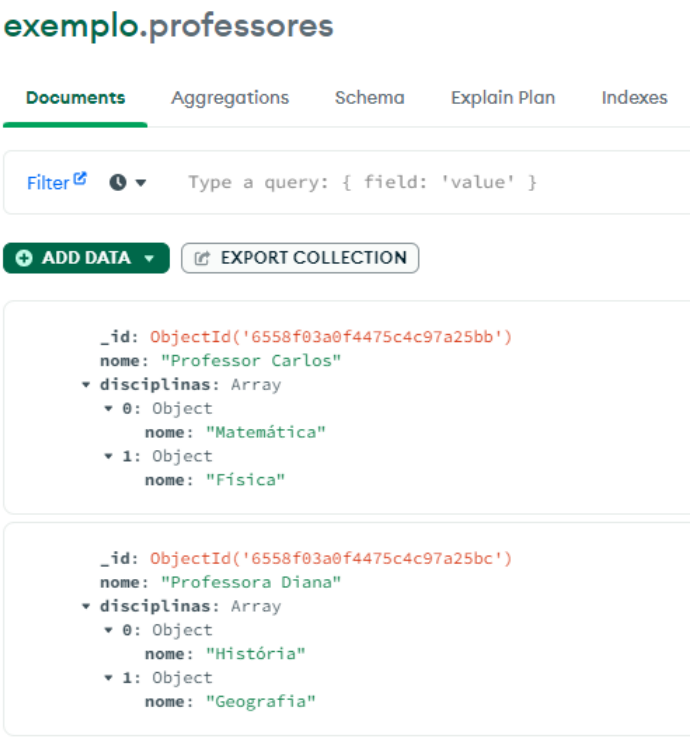


Figura 15 – Coleção *professores* no BD MongoDB.

De forma a implementar esta relação, uma chave estrangeira *professores_id* foi criada na tabela filha professores_disciplinas, que referencia a tabela pai *professores*. Esta relação pode ser visualizada no diagrama da Figura 16 e a estrutura gerada pode ser vista na Figura 17.

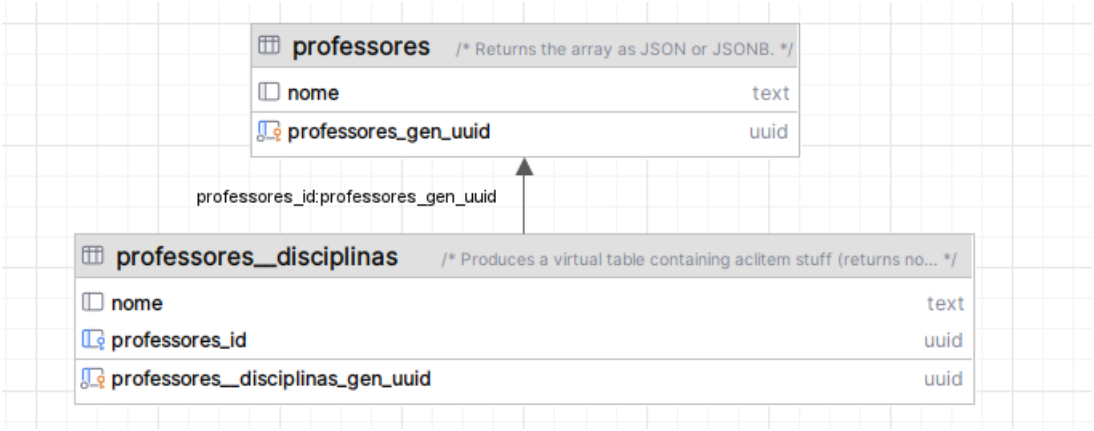


Figura 16 – Diagrama da tabela pai *professores* e tabela filha *professores___disciplinas*.

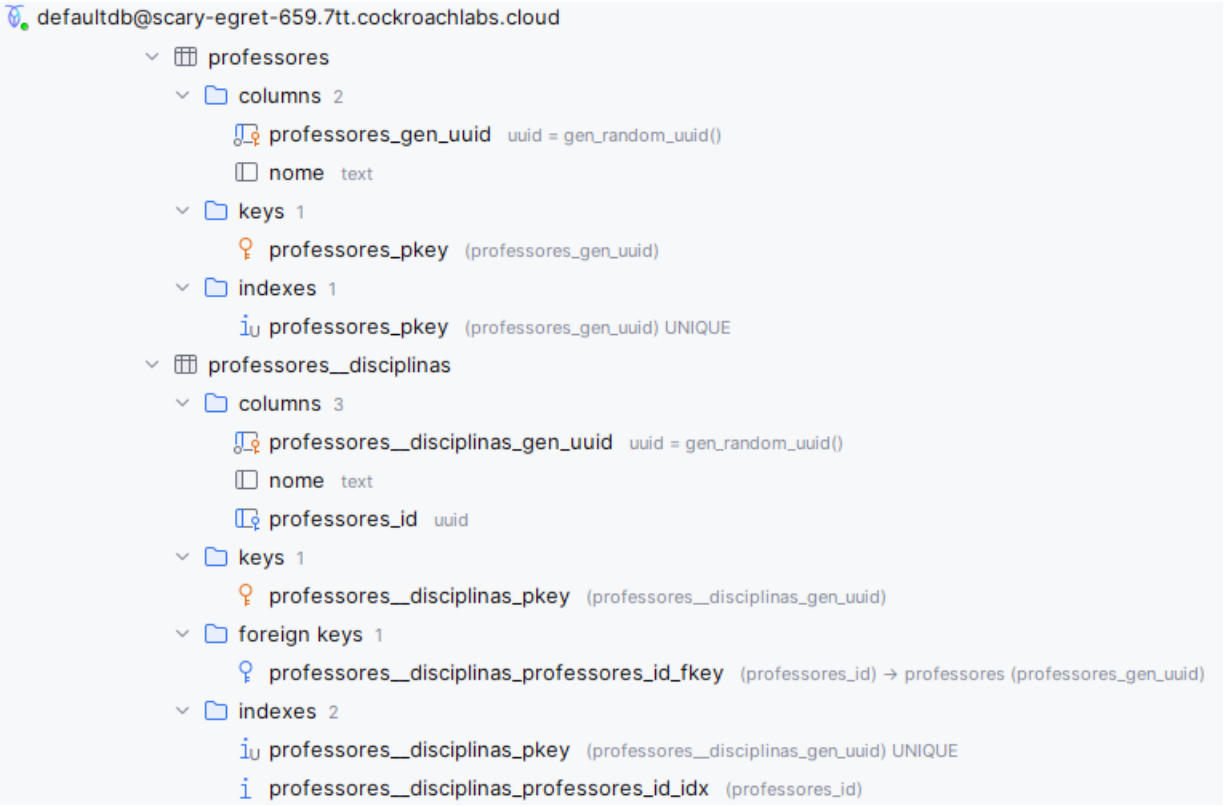


Figura 17 – Estrutura das tabelas *professores* e *professores___disciplinas* geradas no CockroachDB.

5.3 DISCIPLINAS

A coleção *disciplinas*, visualizada na Figura 18, apresenta também um exemplo de relacionamento *muitos-para-muitos*, pois objetos dentro do vetor *estudantes* se repetem dentro de vetores *estudantes* de outras *disciplinas*.

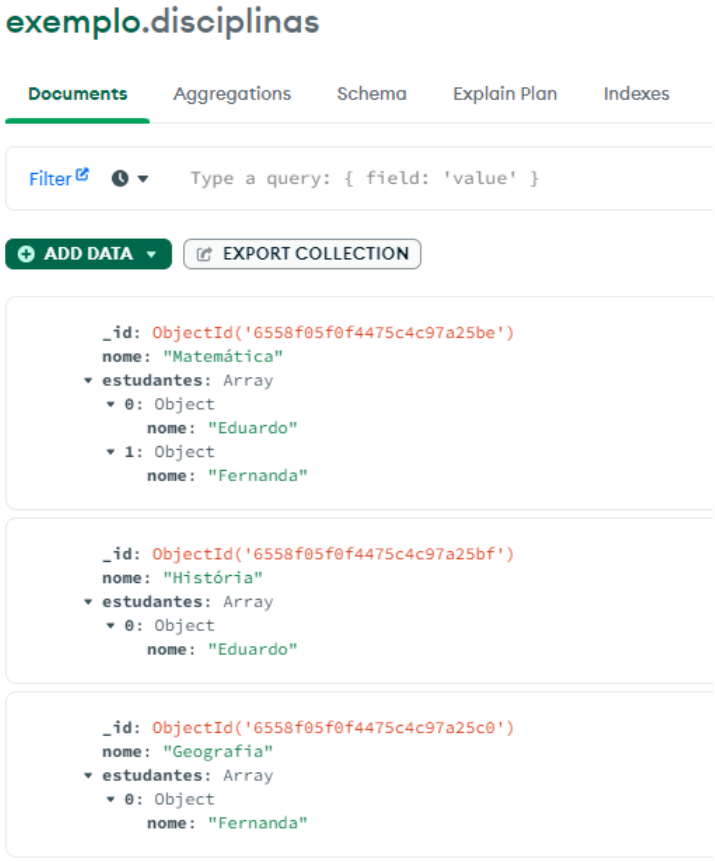


Figura 18 – Coleção *disciplinas* no BD MongoDB.

De forma a implementar esta relação no modelo relacional, foi realizada a criação de uma terceira tabela (tabela associativa), onde foram criadas também 2 chaves estrangeiras: uma fazendo referência à tabela *disciplinas* e outra à tabela *disciplinas___estudantes* de forma a permitir que muitos estudantes participassem de muitas disciplinas e muitas disciplinas possuíssem muitos estudantes.

O diagrama representando esta relação pode ser visto na Figura 19. As estruturas geradas no BD CockroachDB podem ser vistas nas Figuras 20, 21 e 22.

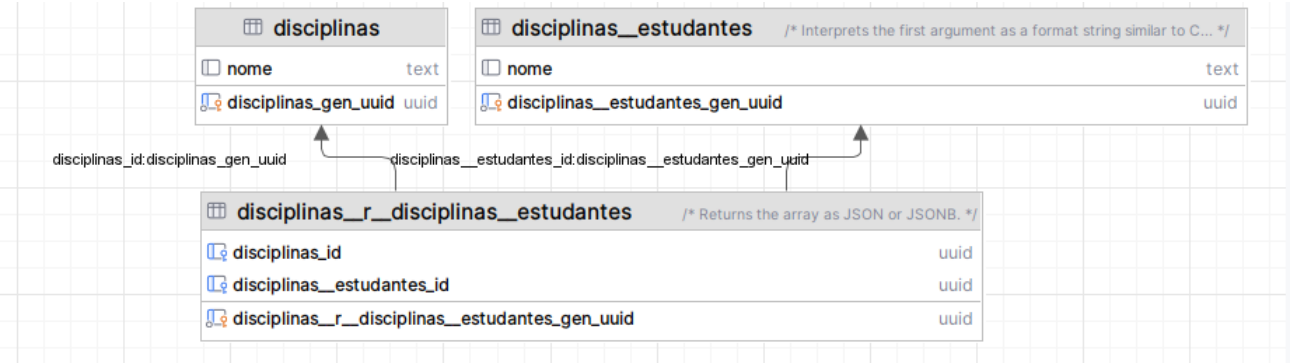


Figura 19 – Diagrama de tabela das tabelas *disciplinas*, *estudantes* e da tabela associativa *disciplinas___r___disciplinas__estudantes*.

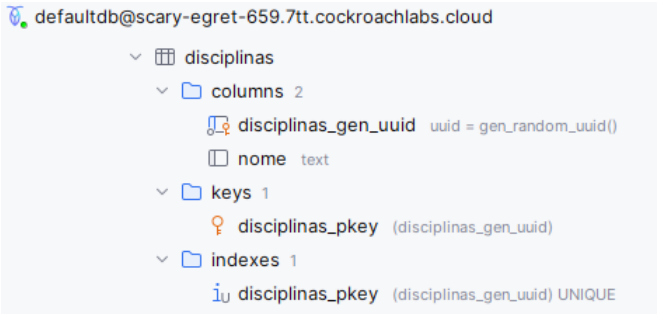


Figura 20 – Estrutura da tabela *disciplinas* gerada no CockroachDB.

Figura 21 – Estrutura da tabela *disciplina___estudantes* gerada no CockroachDB.

Figura 22 – Estrutura da tabela *disciplina___r___estudantes* gerada no CockroachDB.

5.4 BIBLIOTECA

A coleção *biblioteca*, visualizada na Figura 23, é um exemplo de relação *muitos-para-muitos*, pois pertence à um vetor onde valores se repetem em diferentes vetores pais. Sendo assim, de forma a normalizar a estrutura dentro do modelo relacional, uma tabela associativa é criada para prevenir a redundância de dados.

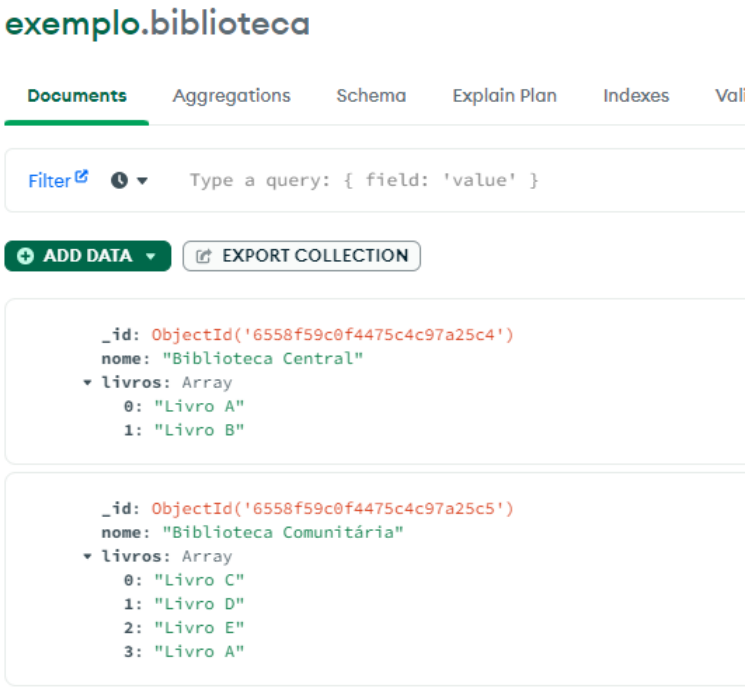


Figura 23 – Coleção *biblioteca* no BD MongoDB.

A implementação deste exemplo segue o padrão de de relações *muitos-para-muitos*, onde a tabela associativa *biblioteca___r___biblioteca_livros* é criada possuindo uma chave

primária e duas chaves estrangeiras apontando para as tabelas *biblioteca___livros* e *biblioteca*. Um diagrama demonstrando essa relação pode ser visto na Figura 24 e a estrutura gerada pela criada no BD CockroachDB pode ser vista nas Figuras 25, 26 e 27.

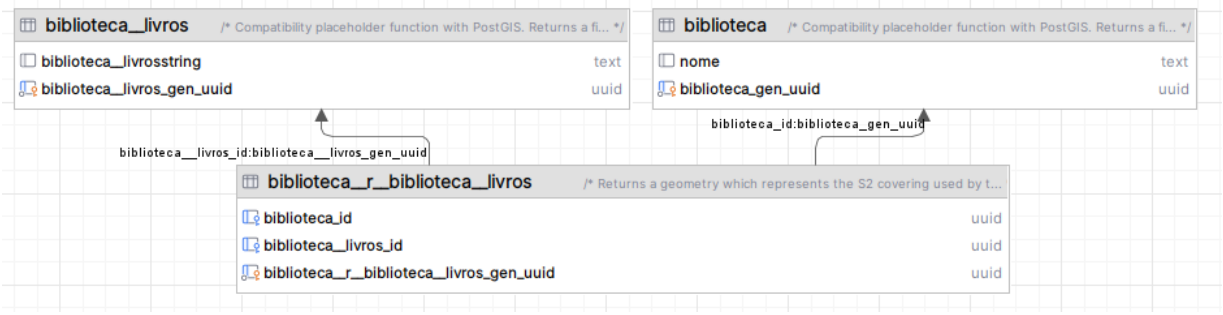


Figura 24 – Diagrama de tabela da tabela de junção *biblioteca___r___biblioteca__livros* e das tabelas *biblioteca___livros* e *biblioteca*.

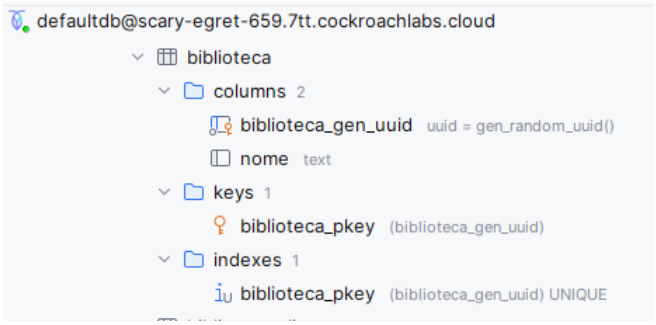


Figura 25 – Estrutura da tabela *biblioteca* gerada, no CockroachDB.

Figura 26 – Estrutura da tabela *biblioteca___livros* gerada, no CockroachDB.

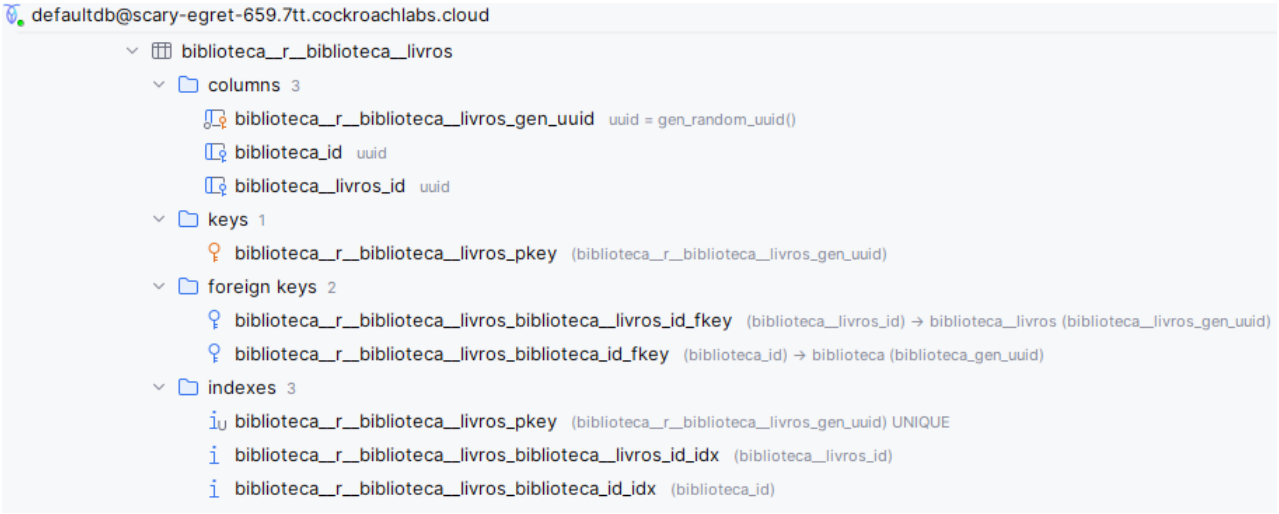


Figura 27 – Estrutura da tabela *biblioteca___r___biblioteca_livros* gerada, no CockroachDB.

5.5 ARTIGOS

A coleção artigos, visualizada na Figura 28, se encaixa em um exemplo de relacionamento *um-para-muitos*, porém de forma inversa. É possível observar que o mesmo objeto *autor* se repete dentro diversos *artigos*, porém não existem diferentes autores para o mesmo artigo.

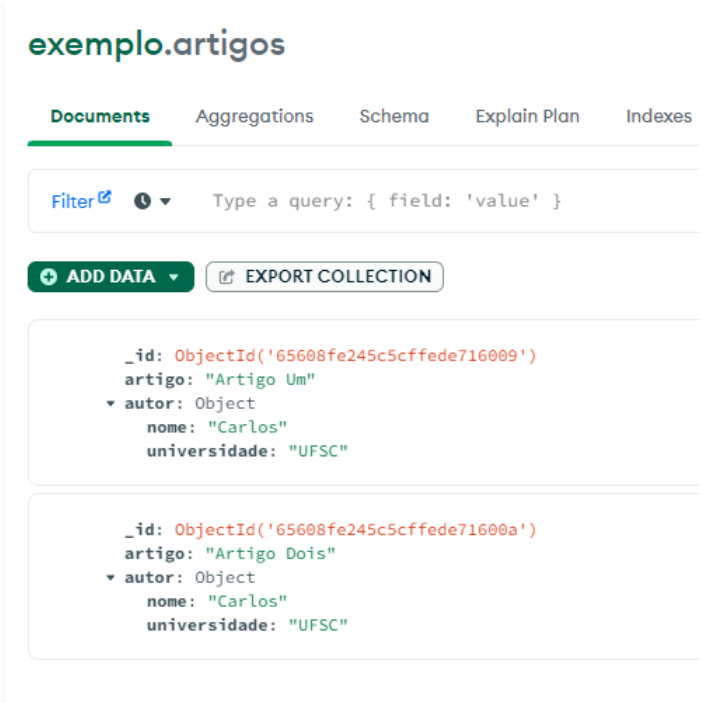


Figura 28 – Coleção *artigos* no BD MongoDB.

Sendo assim, foi criada a tabela *artigos*, contendo sua estrutura original, bem como uma chave estrangeira fazendo referência à tabela pai *artigos___autor*. O diagrama representando esta relação pode ser visualizado na Figura 29 e as estruturas das tabelas geradas podem ser identificadas na Figura 30.

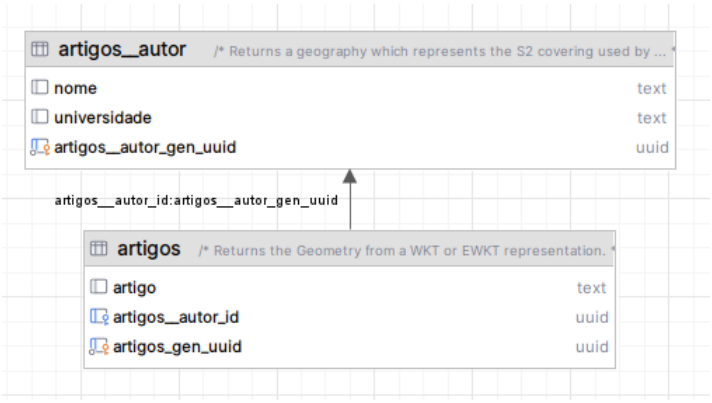


Figura 29 – Diagrama de tabela da tabela pai *artigos___autor* e tabela filha *artigos*.

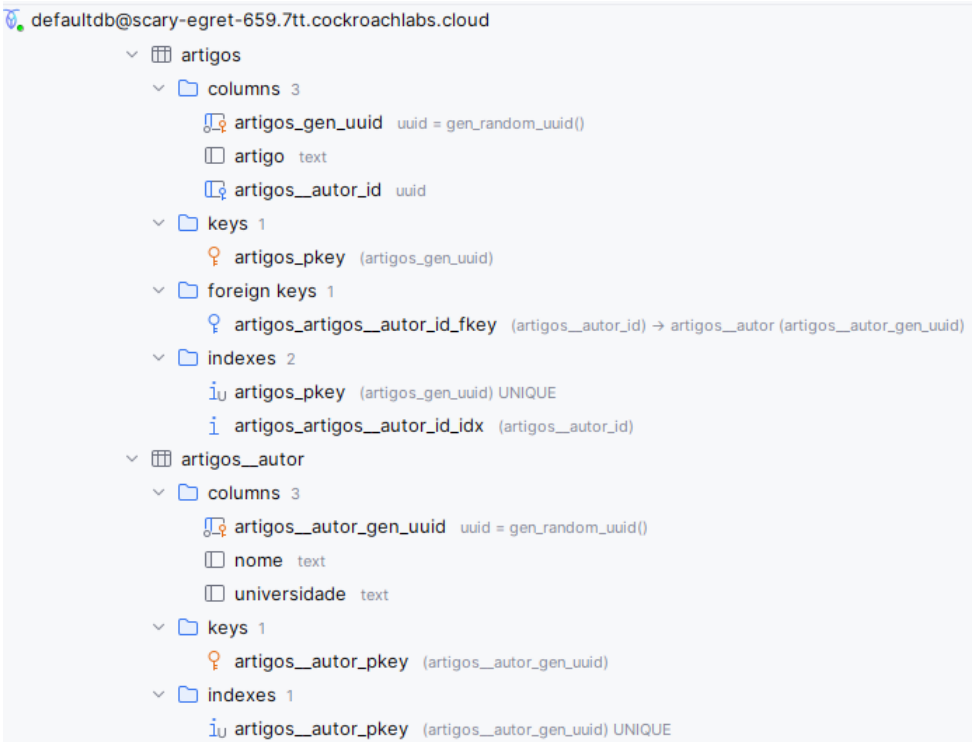


Figura 30 – Estrutura das tabelas *artigos___autor* e *artigos* geradas no CockroachDB.

5.6 SITUAÇÕES

A coleção *situacoes*, visualizada na Figura 31, é um exemplo de relação *muitos-para-muitos*, pois os mesmos objetos possuem diferentes subobjetos e os mesmos subobjetos podem pertencer a diferentes objetos pais.

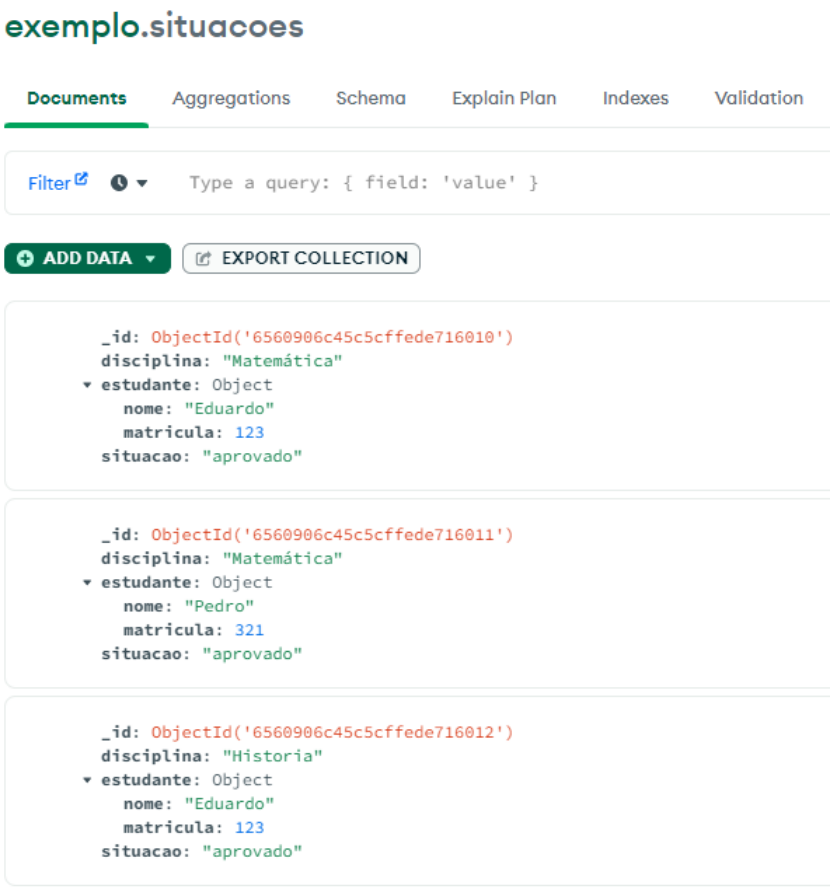


Figura 31 – Coleção *situacoes* no BD MongoDB.

A implementação deste exemplo segue o padrão de relações *muitos-para-muitos*, onde, a tabela associativa *situacoes___r___situacoes_estudante* é criada possuindo uma chave primária e 2 chaves estrangeiras apontando para as tabelas *situacoes___estudante* e *situacoes*. O diagrama representando as relações pode ser visto na Figura 32. Além disso a estrutura das tabelas criadas no BD Cockroach pode ser vista nas Figuras 33, 34 e 35.

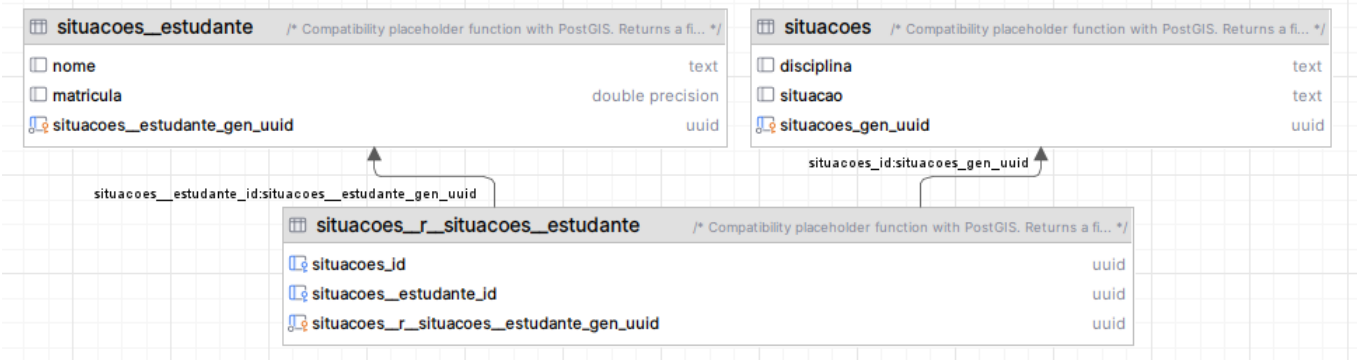


Figura 32 – Diagrama de tabela da tabela de associação *situacoes___r___situacoes___estudante* e das tabelas filhas *situacoes___estudante* e *situacoes*

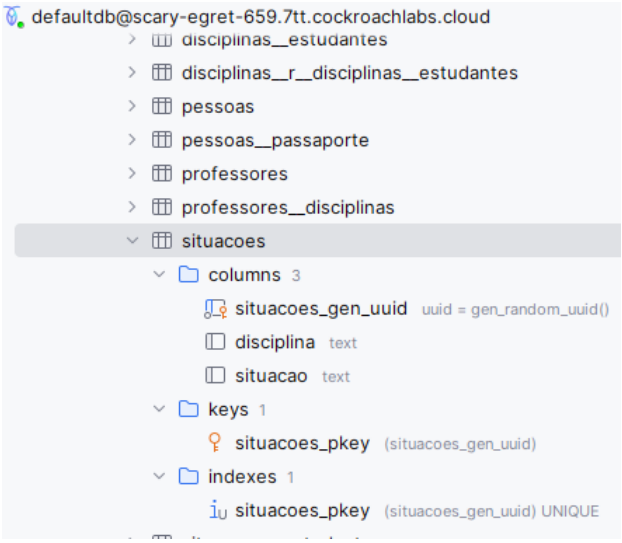


Figura 33 – Estrutura da tabela *situacoes* gerada no CockroachDB.

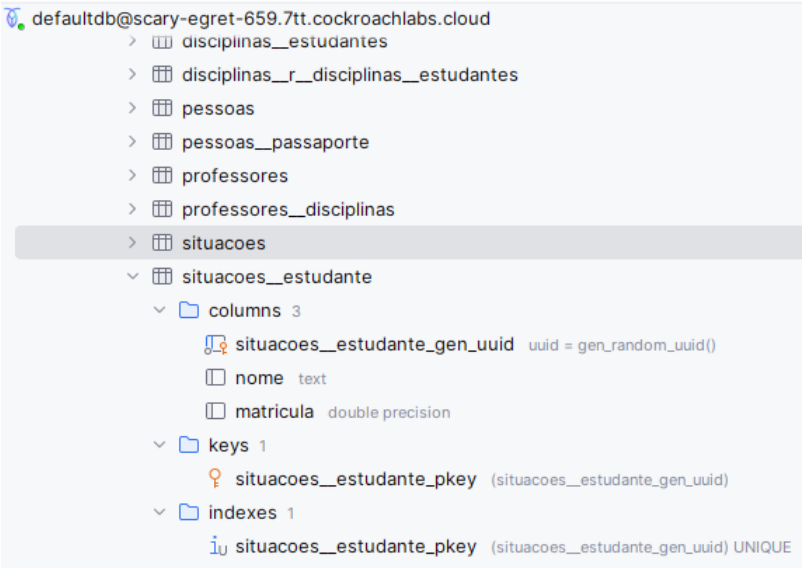


Figura 34 – Estrutura da tabela *situacoes___estudante* gerada no CockroachDB.

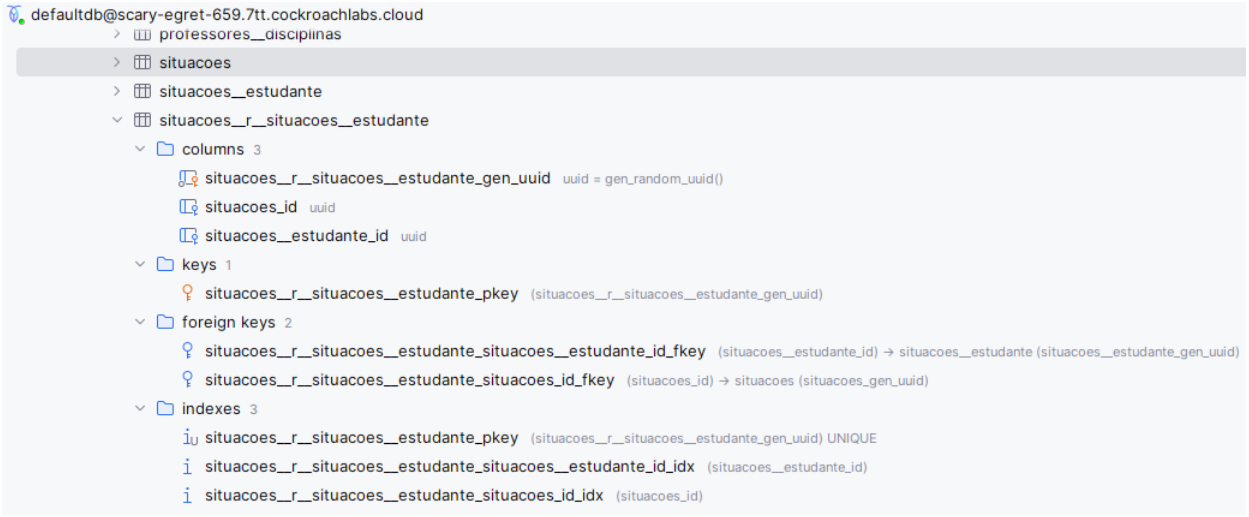


Figura 35 – Estrutura da tabela associativa *situacoes___r___situacoes_estudante* gerada no CockroachDB.

5.7 TRABALHOS

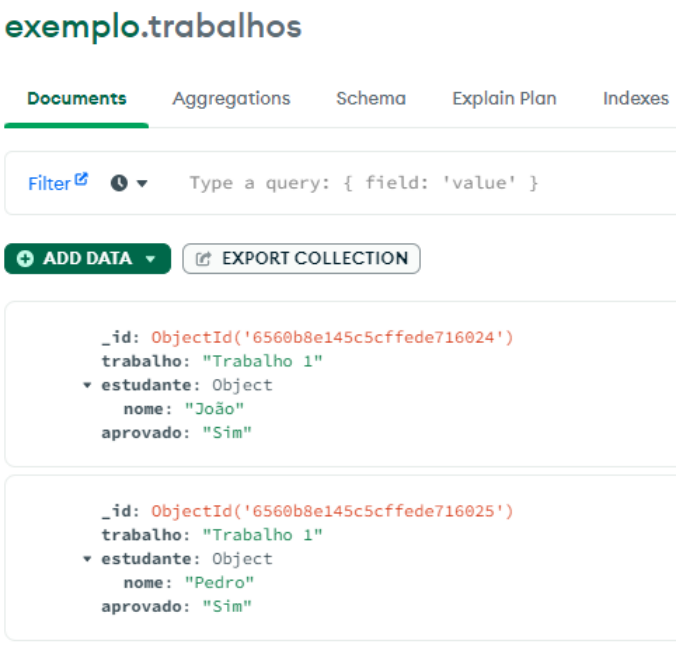


Figura 36 – Coleção *trabalhos* no BD MongoDB.

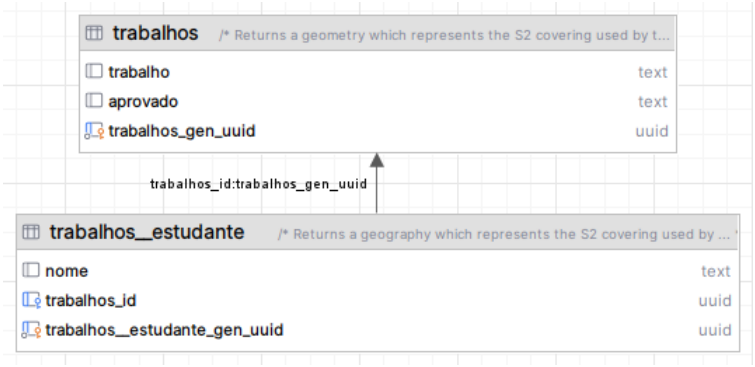


Figura 37 – Diagrama de tabela da tabela pai *trabalhos* e da tabela filha *trabalhos___estudante*.

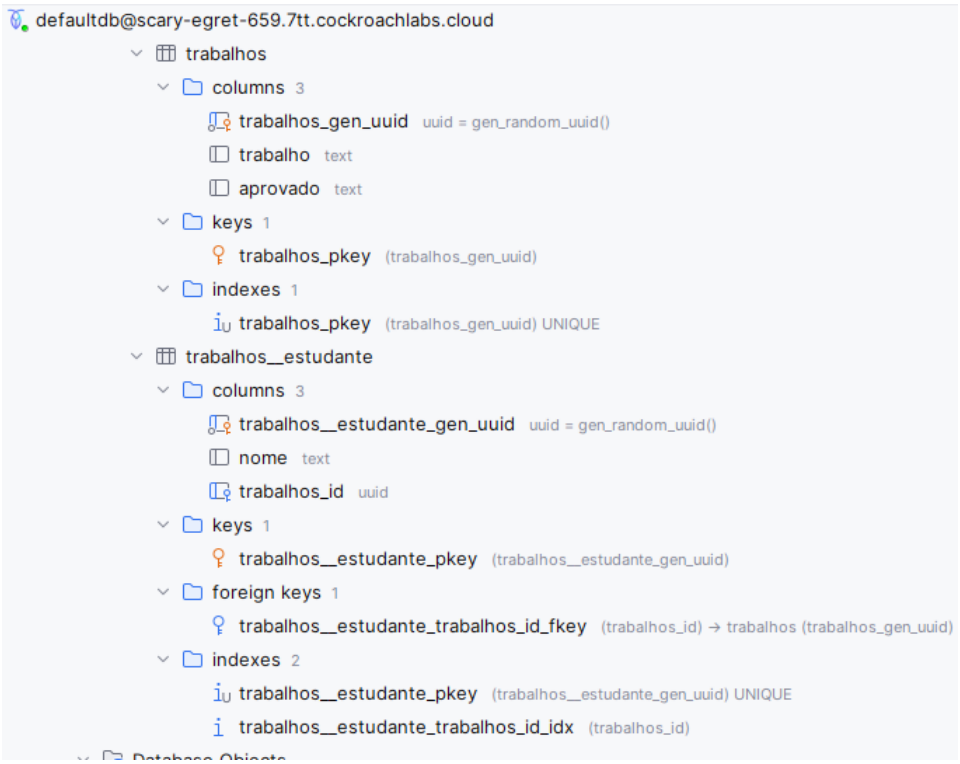


Figura 38 – Estrutura das tabelas *trabalhos* e *trabalhos___estudante* geradas, no CockroachDB.

6 AVALIAÇÃO

Este capítulo apresenta duas avaliações da ferramenta desenvolvida.

A primeira avaliação demontra o comportamento dos dados no BD orientado a documentos e no BD NewSQL de acordo com consultas equivalentes executadas. A intenção aqui é verificar se consultas equivalentes em ambos os BDs geram os mesmos resultados.

A segunda avaliação tem como foco uma análise comparativa de algumas classes de consultas realizadas no BD orientado a documentos MongoDB e no BD NewSQL CockroachDB, com foco no desempenho das consultas executadas. A intenção aqui é mostrar cenários de busca de dados nos quais uma tecnologia NewSQL se mostra superior a uma tecnologia NoSQL, justificando a realização do mapeamento proposto neste trabalho.

6.1 MATERIAIS E MÉTODOS

A metodologia da avaliação consistiu em executar um conjunto de consultas predefinidas em ambos os BDs. As seguintes classes de consultas foram consideradas: consultas simples, consultas com junções, consultas com agregações e ordenação dos dados. De forma a atender estes critérios levantados, um BD JSON no domínio de *Filmes* foi carregado no MongoDB. Além disso, ambos os BDs estavam operando localmente em um mesmo dispositivo com as seguintes configurações:

- Processador 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz;
- RAM instalada 8,00 GB (utilizável: 7,74 GB).

O BD escolhido[1] reúne dados de filmes, como ano, atores e gêneros em um arquivo JSON com 36273 documentos. Exemplos de documentos podem ser vistos na Figura 39.

[1] https://raw.githubusercontent.com/prust/wikipedia-movie-data/master/movies.json

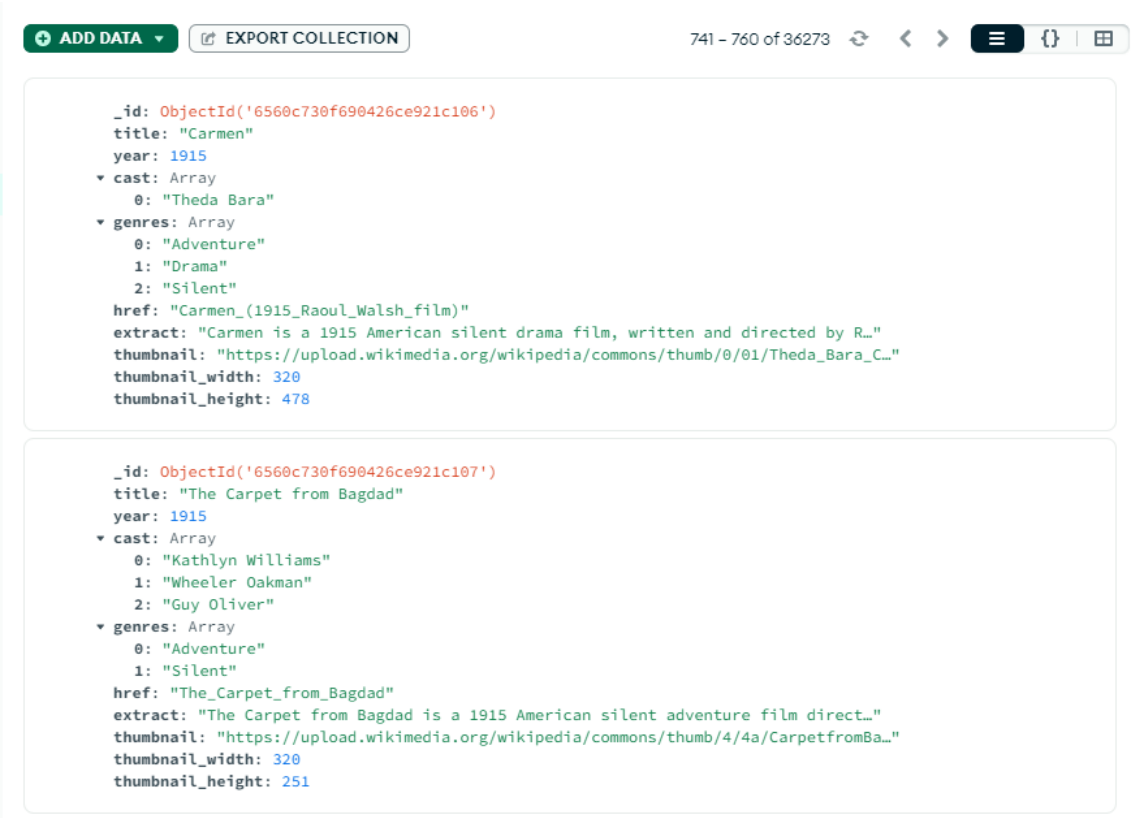


Figura 39 – Exemplos de documentos sobre filmes.

Então seu esquema foi extraído utilizando a ferramenta *extract-mongo-schema*, onde foram identificadas as estruturas do BD de origem, bem como seus tipos de dados. O esquema extraído é mostrado nas Figuras 40 e 41.

```
{
    "movies": {
        "_id": {
            "types": {
                "Object": {
                    "frequency": 36273
                }
            },
            "primaryKey": true
        },
        "title": {
            "types": {
                "string": {
                    "frequency": 36273
                }
            },
            "key": true
        },
        "year": {
            "types": {
                "number": {
                    "frequency": 36273
                }
            }
        },
        "cast": {
            "types": {
                "Array": {
                    "frequency": 36273,
                    "structure": {
                        "types": {
                            "string": {
                                "frequency": 133326
                            }
                        }
                    }
                }
            }
        },
        "genres": {
            "types": {
                "Array": {
                    "frequency": 36273,
                    "structure": {
                        "types": {
                            "string": {
                                "frequency": 64228
                            }
```


Figura 40 – Extração do esquema do BD de filmes(1/2).

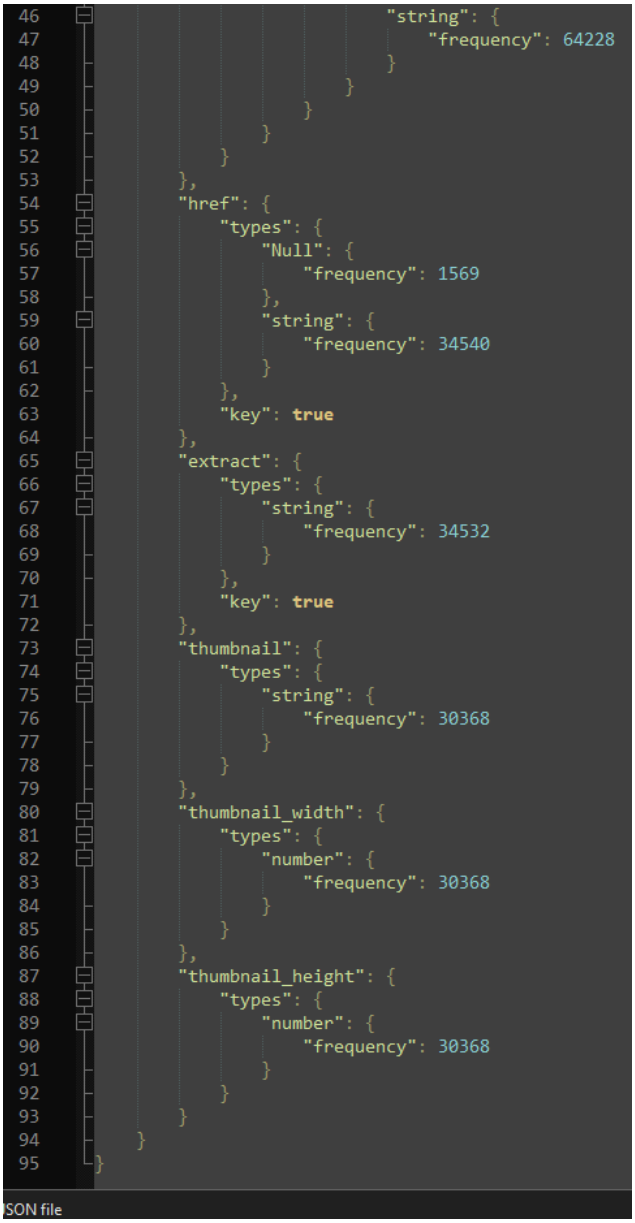


Figura 41 – Extração do esquema do BD de filmes(2/2).

Os documentos extraídos foram então inseridos na ferramenta *document2sql* e seu mapeamento para o modelo relacional foi realizado, gerando o seguinte código DDL:

```
1 CREATE TABLE movies (movies_gen_uuid UUID PRIMARY KEY DEFAULT
    gen_random_uuid(), title string, year float, href string, extract
    string, thumbnail string, thumbnail_width float, thumbnail_height
    float);
2 CREATE TABLE movies__cast (movies__cast_gen_uuid UUID PRIMARY KEY
    DEFAULT gen_random_uuid(), movies__caststring string);
3 CREATE TABLE movies__genres (movies__genres_gen_uuid UUID PRIMARY KEY
    DEFAULT gen_random_uuid(), movies__genresstring string);
4 CREATE TABLE movies__R__movies__cast (movies__R__movies__cast_gen_uuid
    UUID PRIMARY KEY DEFAULT gen_random_uuid(), movies_id UUID,
```

```
    movies__cast_id UUID, FOREIGN KEY (movies_id) REFERENCES movies,
    FOREIGN KEY (movies__cast_id) REFERENCES movies__cast);
CREATE INDEX ON movies__R__movies__cast(movies_id);
CREATE INDEX ON movies__R__movies__cast(movies__cast_id);
CREATE TABLE movies__R__movies__genres (
    movies__R__movies__genres_gen_uuid UUID PRIMARY KEY DEFAULT
    gen_random_uuid(), movies_id UUID, movies__genres_id UUID, FOREIGN
    KEY (movies_id) REFERENCES movies, FOREIGN KEY (movies__genres_id)
    REFERENCES movies__genres);
CREATE INDEX ON movies__R__movies__genres(movies_id);
CREATE INDEX ON movies__R__movies__genres(movies__genres_id);
```

A partir de então, o código DDL foi executado no BD CockroachDB e os dados do BD de filmes no MongoDB foram migrados para uma estrutura tabular intermediária no BD H2 utilizando *scripts* de inserção gerados pelo ambiente de desenvolvimento *DataGrip*[2], para cada documento existente no BD. É possível identificar pela Figura 42 que os arrays estavam contidos dentro de campos de texto. Na sequência, rotinas SQL foram aplicadas sobre esta tabela intermediária de forma a separar e distribuir os dados dentro do esquema relacional gerado pela ferramenta *document2sql* no CockroachDB.

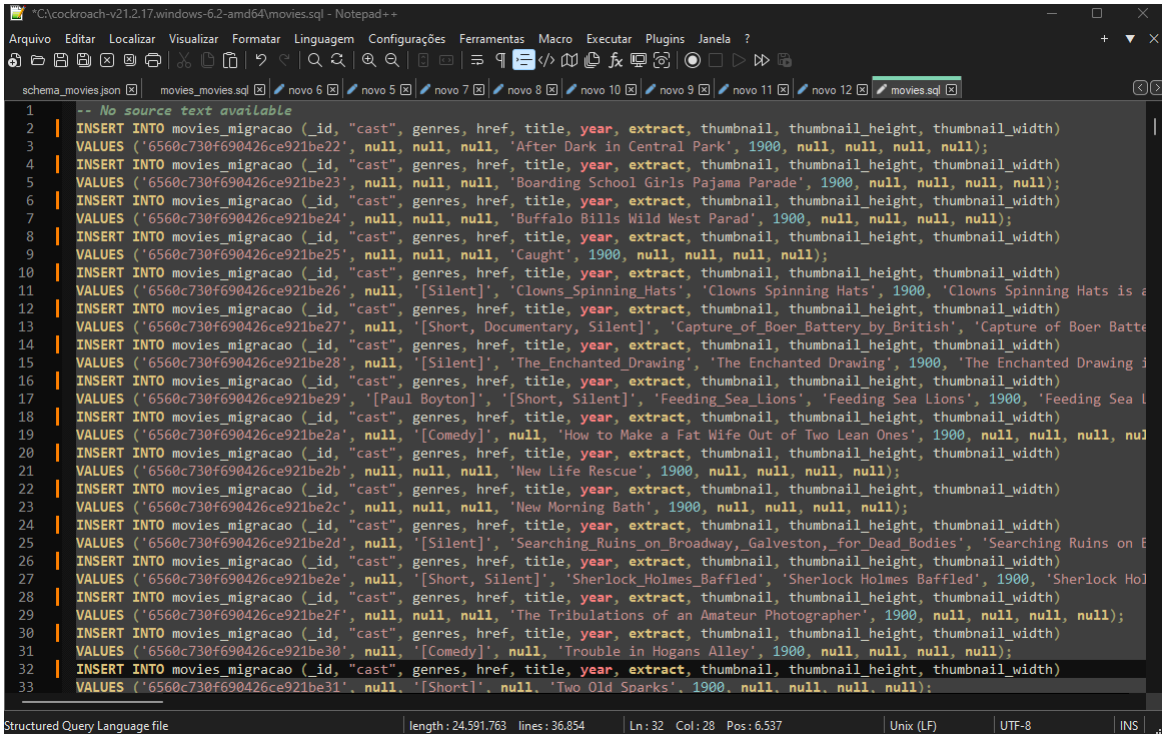


Figura 42 – *Script* de inserção de dados do BD Filmes gerado pelo *DataGrip*.

Com a migração dos dados realizada, foram definidas consultas equivalentes para o MongoDB e o CockroachDB de forma a avaliar a compatibilidade dos esquemas de dados bem como o desempenho dos mesmos nas categorias de consultas comentadas

[2] https://www.jetbrains.com/pt-br/datagrip/

anteriormente. Para a avaliação de desempenho cada categoria de consulta, executada com o auxílio de índices ou não, foi executada 10 vezes e sua média foi calculada. Os resultados das avaliações são detalhados a seguir.

6.2 AVALIAÇÃO DA COMPATIBILIDADE DO ESQUEMA DE DADOS GERADO

Esta avaliação tem por objetivo verificar se os resultados no MongoDB e no CockroachDB são compatíveis, quando utilizando junções, funções de agregação e operações condicionais. Para a execução desta consulta no CockroachDB foram utilizadas todas as tabelas geradas pelo mapeamento, de forma a validar se a estrutura gerada é compatível com a estrutura do MongoDB.

A seguinte consulta foi executada no MongoDB e o seu resultado pode ser visto na figura 43.

```
db.movies.aggregate([
    { $match: { year: 2023 } },
    { $unwind: "$genres" },
    { $unwind: "$cast" },
    { $group: {
        _id: "$genres",
        uniqueActors: { $addToSet: "$cast" }
    }},
    { $project: {
        _id: 0,
        genre: "$_id",
        numberOfActors: { $size: "$uniqueActors" }
    }},
    { $sort: { numberOfActors: -1 } }
]);
```

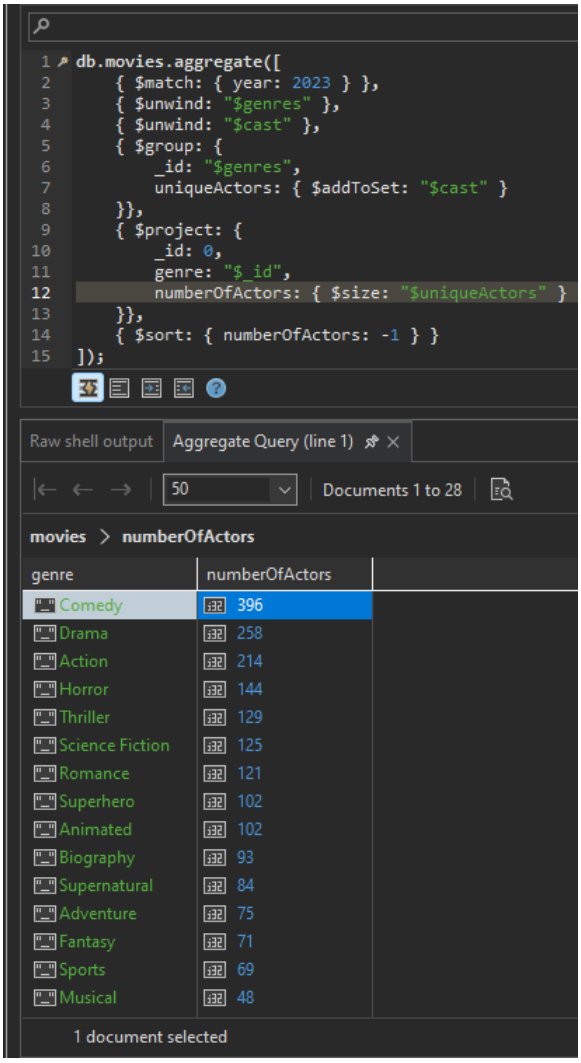


Figura 43 – Resultado da consulta no MongoDB, para a avaliação de compatibilidade dos esquemas de dados.

A consulta equivalente a seguir foi executada no CockroachDB, sua execução e resultados podem ser vistos na figura 44.

```
SELECT
    mg.movies__genresstring AS genre,
    COUNT(DISTINCT mc.movies__caststring) AS numberOfActors
FROM movies m
JOIN movies__R__movies__cast rmc ON m.movies_gen_uuid = rmc.movies_id
JOIN movies__cast mc ON rmc.movies__cast_id = mc.movies__cast_gen_uuid
JOIN movies__R__movies__genres rmg ON m.movies_gen_uuid = rmg.movies_id
JOIN movies__genres mg ON rmg.movies__genres_id = mg.
   movies__genres_gen_uuid
WHERE m.year = 2023
GROUP BY mg.movies__genresstring
ORDER BY numberOfActors DESC;
```

```
SELECT
    mg.movies__genresstring AS genre,
    COUNT(DISTINCT mc.movies__caststring) AS numberOfActors
FROM movies m
JOIN movies__R__movies__cast rmc ON m.movies_gen_uuid = rmc.movies_id
JOIN movies__cast mc ON rmc.movies__cast_id = mc.movies__cast_gen_uuid
JOIN movies__R__movies__genres rmg ON m.movies_gen_uuid = rmg.movies_id
JOIN movies__genres mg ON rmg.movies__genres_id = mg.movies__genres_gen_uuid
WHERE m.year = 2023
GROUP BY mg.movies__genresstring
ORDER BY numberOfActors DESC;
```

Output | Result 10

28 rows

	genre	numberofactors
1	Comedy	396
2	Drama	258
3	Action	214
4	Horror	144
5	Thriller	129
6	Science Fiction	125
7	Romance	121
8	Animated	102
9	Superhero	102
10	Biography	93
11	Supernatural	84
12	Adventure	75
13	Fantasy	71
14	Sports	69
15	Musical	48

Figura 44 – Resultado da consulta no CockroachDB, para a avaliação de compatibilidade dos esquemas de dados.

É possível visulizar através das figuras 43 e 44 que as consultas equivalentes em ambos os BDs geraram os mesmos resultados, justificando que o mapeamento realizado entre o modelo orientado a objetos e o modelo relacional gerou resultados compatíveis.

6.3 AVALIAÇÃO DE DESEMPENHO

6.3.1 Consulta Simples

Essa categoria de consulta tem por finalidade avaliar o desempenho de uma busca sem a utilização de junções e funções de agregação. O código a seguir foi executado no MongoDB e teve um tempo médio de execução de 71,8 milissegundos.

```
db.movies.find({ "year": { $gt: 2000 } }, { "title": 1, "year": 1 });
```

Adicionando índices nas colunas *title* e *year* o MongoDB atingiu um tempo médio de execução de 115,5 milissegundos, ou seja, houve uma queda em seu tempo de execução. Já no CockroachDB, a consulta equivalente (código a seguir) teve uma média de tempo de execução de 230,1 milissegundos.

```
SELECT title, year FROM movies WHERE year > 2000;
```

O CockroachDB se mostrou menos performático em consultas simples sem a inclusão de índices. Com a inclusão de índices nas colunas *year* e *title*, obteve-se uma média de execução de 116,1 milissegundos, equivalendo em desempenho ao MongoDB, conforme mostra a Tabela 3.

Tabela 3 – Tempos de execução para consultas simples no CockroachDB e no MongoDB.

Tempo médio de execução	**Sem índices**	**Com índices**
CockroachDB	230,1 ms	116,1 ms
MongoDB	71,8 ms	115,5 ms

6.3.2 Consultas com Junções

Este experimento tem por objetivo avaliar o desempenho de consultas envolvendo junções. A seguinte consulta foi executada no BD MongoDB. Ela apresentou um tempo médio de execução de 144,5 milissegundos.

```
db.movies.find({ "year": { $gt: 2000 }, "cast": "Keanu Reeves" }, { "
    title": 1, "year": 1, "cast": 1 });
```

Na sequência, foram adicionados índices na chave *year* e no array *cast*, o que resultou em um tempo médio de processamento no MongoDB de 6,9 milissegundos, evidenciando a relevância de índices em arrays para consultas.

A consulta equivalente foi executada no BD CockroachDB, que obteve um tempo médio de execução de 167,5 milissegundos sem o uso de índices. Também foram inseridos índices nas colunas *year* e mc.*movies___caststring*. Estas alterações tornaram possível atingir um tempo médio de execução de 144,3 milissegundos. Um resumo dos resultados é mostrado na Tabela 4.

```
SELECT m.title, m.year
FROM movies m
JOIN movies__R__movies__cast rmc ON m.movies_gen_uuid = rmc.movies_id
JOIN movies__cast mc ON rmc.movies__cast_id = mc.movies__cast_gen_uuid
WHERE mc.movies__caststring = 'Keanu Reeves' AND m.year > 2000;
```

Tabela 4 – Tempos de execução parabom consultas com junções no CockroachDB e no MongoDB.

Tempo médio de execução	**Sem índices**	**Com índices**
CockroachDB	167,5ms	144,3 ms
MongoDB	144,5ms	6,9 ms

Neste experimento percebe-se um melhor desempenho do MongoDB uma vez que os dados dos arrays estão agregados no documento, evitando junções. Já no BD relacional junções foram necessárias.

6.3.3 Consultas complexas com junções e agregações

Este experimento avalia o desempenho de uma consulta mais complexa envolvendo junções de múltiplas tabelas e agregações. A seguinte instrução foi executada no MongoDB, obtendo um tempo médio de 1956,7 milissegundos. Com a adição de índices nas chaves *year*, *cast* e *genres* no MongoDB obteve-se um tempo médio de 1296,7 milissegundos.

```
db.movies.aggregate([
    { $match: { "year": { $gte: 1937, $lte: 2015 } } },
    { $unwind: "$cast" },
    { $unwind: "$genres" },
    { $group: {
        _id: { cast: "$cast", genre: "$genres", movieId: "$_id" }
    }},
    { $group: {
        _id: { cast: "$_id.cast", genre: "$_id.genre" },
        total_movies: { $sum: 1 }
    }},
    { $match: { total_movies: { $gt: 1 } }},
    { $project: {
        cast: "$_id.cast",
        genre: "$_id.genre",
        total_movies: 1,
        _id: 0
    }}
]);
```

Já no CockroachDB a consulta equivalente obteve um tempo de médio de 1389,1 milissegundos. Ao adicionar índices nas colunas *year*, *movies_caststring movies_genrestring* obteve-se um tempo médio de 1266,4 milissegundos. A Tabela 5 resume os resultdos obtidos.

```
SELECT
    mc.movies__caststring,
    mg.movies__genresstring,
```

```
4       COUNT(*) as total_movies
5   FROM
6       (SELECT
7           rmc.movies__cast_id,
8           rmg.movies__genres_id,
9           m.movies_gen_uuid
10       FROM movies m
11       JOIN movies__R__movies__cast rmc ON m.movies_gen_uuid = rmc.
        movies_id
12       JOIN movies__R__movies__genres rmg ON m.movies_gen_uuid = rmg.
        movies_id
13       WHERE
14          m.year BETWEEN 1937 AND 2015) AS subquery
15  JOIN movies__cast mc ON subquery.movies__cast_id = mc.
        movies__cast_gen_uuid
16  JOIN movies__genres mg ON subquery.movies__genres_id = mg.
        movies__genres_gen_uuid
17  GROUP BY
18      mc.movies__caststring,
19      mg.movies__genresstring
20  HAVING
21      COUNT(*) > 1;
```

Tabela 5 – Tempos de execução alcançados pela consultas complexas baseadas em múltiplas junções e agregações no CockroachDB e no MongoDB.

Tempo médio de execução	**Sem índices**	**Com índices**
CockroachDB	1389,1 ms	1266,4 ms
MongoDB	1956,7 ms	1296,7 ms

Neste experimento percebe-se um melhor desempenho do CockroachDB para lidar com uma consulta complexa, evidenciando a melhor utilização de um BD relacional e uma vantagem ao migrar dados de um BD NoSQL orientado a documentos para uma tecnologia relacional.

6.4 CONCLUSÃO

Neste capítulo apresenta uma avaliação comparativa entre o MongoDB, um BD orientado a documentos, e o CockroachDB, um sistema de gerenciamento de BD relacional NewSQL. As avaliações foram realizadas utilizando um conjunto de consultas predefinidas, executadas em ambos os sistemas, para avaliar as diferenças no desempenho. O foco principal foi o impacto do uso de índices, junções e funções de agregação, além da avaliação da estrutura proposta pela ferramenta *document2sql*.

Os resultados mostraram que, para consultas simples houve um desempenho equilibrado entre as 2 tecnologias. Para consultas envolvendo junções, quando essa junções envolvem filtros sobre dados aninhados em um mesmo documento o melhor desempenho do MongoDB já era esperado, uma vez que os dados estão naturalmente agrupados em documento JSON. Em contrapartida, para consultas complexas, com múltiplas junções e operações de agregação, o CockroachDB obteve um melhor desempenho.

Esta avaliação experimental demonstra que tanto o MongoDB quanto o CockroachDB têm seus pontos fortes em diferentes tipos de operações de BD. O MongoDB se destaca em cenários que envolvem operações simples em documentos e dados aninhados, enquanto o CockroachDB demonstra vantagens em consultas relacionais mais complexas, especialmente quando otimizado com índices. Esses resultados são cruciais para determinar a escolha do sistema de BD, dependendo das necessidades específicas de desempenho para determinados tipos de operação. É possível concluir também que a estrutura gerada pela ferramenta *document2sql* atingiu seu objetivo em definir um mapeamento que mantenha as características dos dados mesmo migrados para o modelo de dados relacional, aproveitando seus benefícios no sentido de manter a integridade dos dados bem como um melhor desempenho em certos cenários que justifique o mapeamento NoSQL para SQL.

7 CONSIDERAÇÕES FINAIS

O presente trabalho teve como principal objetivo o desenvolvimento de uma ferramenta capaz de gerar o mapeamento estrutural de um BD orientado a documentos para um BD relacional a partir de regras de mapeamento predefinidas. A ferramenta desenvolvida *document2sql* se diferencia dos trabalhos relacionados pela sua capacidade de trabalhar com diferentes cardinalidades nas relações identificadas em conjuntos de dados.

A principal motivação deste trabalho é entregar uma ferramenta capaz de auxiliar na integração e migração de dados e observa-se que todos os objetivos definidos foram cumpridos. A aplicação é capaz de ler e analisar dados em um BD MongoDB de origem, identificar o esquema JSON deste BD de origem e definir a estrutura equivalente de um BD no modelo relacional levando em consideração as relações entre suas diferentes entidades. Diferentes cenários envolvendo diferentes cardinalidades foram testados com diferentes conjuntos de dados, demonstrando as regras de mapeamento.

Este trabalho também foi de grande relevância para o desenvolvimento teórico e técnico do autor, em grande parte à pesquisa realizada a respeito de tecnologias de BDs, principalmente os modelos orientado a documento e relacional.

A primeira versão do *document2sql* se encontra funcional, porém, diversos aprimoramentos podem ser realizados como trabalhos futuros:

- Uma interface gráfica que facilite a usabilidade do usuário e que seja capaz de gerar diagramas de tabela do mapeamento realizado;
- Adição de novos BDs de origem e de destino para fins de mapeamento;
- Possibilidade de realizar a migração dos dados do BD de origem para o BD destino de forma automática.

Todo o código gerado para o desenvolvimento deste trabalho se encontra em um repositório no *GitHub* que pode ser acessado através deste link: https://github.com/rafaelparola/document2sql.

REFERÊNCIAS

AFTAB, Zain. Automatic NoSQL to Relational Database Transformation with Dynamic Schema Mapping. Hindawi, 2020.

CHRISTOPH, B. SQL for Nosql Databases: Deja Vu (part 2). CIDR2017, 2017.

CODD, E. F. A Relational Model of Data for Large Shared Data Banks. **Communications of the ACM**, v. 13, p. 377–387, jun. 1970.

D. KONSTANTINOS L. PETER, C. Paul. **Concise Guide to Databases**. [*S.l.*]: Springer, 2021.

D. PAUL, D. Harvey. **Java: Como Programar, 10ª Edição**. [*S.l.*]: Pearson Education do Brasil Ltda., 2017.

DAVID, P. **NoSQL: Como armazenar os dados de uma aplicação moderna**. [*S.l.*]: Casa do Código, 2016.

DB-ENGINES Ranking. [*S.l.*: *s.n.*]. https://db-engines.com/en/ranking. "Acessado em: 11/12/2023".

F. MARTIN, J. S. Pramos. **NoSQL Essencial: um Guia Conciso Para o Mundo Emergente da Persistência Poliglota**. [*S.l.*]: Novatec Editora, 2019.

H2 Database. [*S.l.*: *s.n.*]. https://www.h2database.com/html/history.html. "Acessado em: 20/11/2023".

HERNANDEZ, Michael J. **Database Design for Mere Mortals**. [*S.l.*]: Pearson Education, Inc., 2021.

K. O. KENECHUKWU, E. E. Virginia. Implementation of Cross-Platform Language between SQL and NoSQL Database Systems, 2016.

KITCHENHAM, Barbara. Procedures for performing systematic reviews. **Keele, UK, Keele University**, v. 33, n. 2004, p. 1–26, 2004.

M. ANDREAS, K. Michael. **SQL NoSQL Databases**. [*S.l.*]: Springer, 2019.

MAITY, Biswajit. A Framework to Convert NoSQL to Relational Model. ACM, 2018.

MASSÉ, Mark. **Rest API. Design Rulebook**. [*S.l.*]: O'Reilly Media, Inc., 2012.

MICHAEL, S. New Opportunities for New SQL. **Communications of the ACM**, v. 55, p. 10–11, nov. 2012.

_____. SQL Databases v. NoSQL Databases. **Communications of the ACM**, v. 53, p. 10–11, abr. 2010.

N. AMEYA P. ANIL, P. Dikshay. Type of NOSQL Databases and its Comparison with Relational Databases. International Journal of Applied Information Systems, 2013.

PETKOVIĆ, Dušan. Non-native Techniques for Storing JSON Documents into Relational Tables. iiWAS2020, 2020.

R. JULIAN S. L. PHILIPP, M. Klaus. Speaking in Tongues: SQL Access to NoSQL Systems. ACM, 2014.

R. RAGHU, G. Johannes. **Sistemas de Gerenciamento Sistemas de Gerenciamento de Banco de Dados**. [*S.l.*]: McGraw-Hill, 2008.

RAMON, L. Integration and Vizualization of Relational SQL and NoSQL Systems including MySQL and MongoDB. Conference: 2014 International Conference on Computational Science e Computational Intelligence (CSCI), 2014.

REBECCA, T. CockroachDB: The Resilient Geo-Distributed SQL Database. **SIGMOD '20: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data**, pages 1493–1509, abr. 2020.

ROB, R. **Practical CockroachDB: Building Fault-Tolerant Distributed SQL Databases**. [*S.l.*]: Appress, 2022.

SPRING Boot. [*S.l.*: *s.n.*]. https://spring.io/projects/spring-boot. "Acessado em: 20/11/2023".

STEVE, O. **Oca Oracle Database SQL Exam Guide (Exam 1z0-071)**. [*S.l.*]: McGraw Hill Education, 2017.

WERNER, Aleksandra; BACH, Ma lgorzata. NoSQL E-learning Laboratory—Interactive Querying of MongoDB and CouchDB and Their Conversion to a Relational Database. International Conference on Man–Machine Interactions, 2018.

A APÊNDICE - ARTIGO NO FORMATO SBC

Uma solução para mapeamento de bancos de dados NoSQL baseados em documentos para bancos de dados relacionais NewSQL

Rafael Parola[1], **Ronaldo dos Santos Mello**[2]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil

Abstract. *This article aims to present structure mapping rules document-oriented databases for the relational model. The intention is to generate a data schema compatible with the database-oriented documents and demonstrating better performance in specific scenarios.*

Resumo. *Este artigo visa apresentar regras de mapeamento de estruturas de bases de dados orientadas a documentos para o modelo relacional. A intenção é gerar um esquema de dados compatível com a base de dados orientada a documentos e demonstrando melhor performance em cenários específicos.*

1. Introdução

Os bancos de dados (BDs) estão presentes em praticamente todos os contextos da sociedade. Com o avanço da tecnologia e o surgimento do movimento *big data*, grande parte de tudo o que fazemos geram dados, seja em uma compra no mercado ou na busca de artigos para o desenvolvimento deste trabalho. Sendo assim, a busca por uma melhor forma de armazenar e manipular estes dados surgiu [M. Andreas 2019].

Dessa forma, surgiram diversos modelos de BDs, dentre eles o modelo relacional, mais maduro, fornecendo uma estrutura simples para os dados e integridade em suas transações. Entretanto, ele não se mostrou eficiente quando utilizado em aplicações que geram e precisam armazenar um grande volume de dados muitas vezes heterogêneos e complexos. A partir disso surgiram os BDs NoSQL, que em muitos casos relaxam sua integridade e fornecem uma maior flexibilidade de estruturação dos dados em troca de um melhor desempenho com grandes volumes de dados [F. Martin 2019]. Mesmo assim, a comunidade acadêmica e da indústria na área de BD continuou na busca de alternativas eficientes para o *big data* e que fornecessem a estrutura e a integridade das bases de dados relacionais. Então, surgiram os BDs NewSQL, que aliam a escalabilidade horizontal e alta disponibilidade dos BDs NoSQL com o suporte à integridade e robustez do modelo relacional [Michael 2012].

Os BDs NoSQL orientados a documentos geralmente são *schemaless*, ou seja, não possuem uma estrutura predefinida de seus dados. Formados por documentos que atribuem um valor a uma chave, não há uma regra que define quais atributos deverão estar em cada documento, ou seja, documentos em um mesmo BD podem conter objetos com diferentes atributos. [D. Konstantinos 2021].

A partir deste contexto, com a chegada dos BDs NewSQL, ideais para sistemas do segmento OLTP que necessitem de alto desempenho [Rebecca 2020], verifica-se que

a migração de BDs NoSQL para o modelo relacional, adotado por BDs NewSQL, é interessante para agregar desempenho e consistência forte para dados de diversos domínios de aplicação. Entretanto, levando em consideração a grande diferença na estrutura de armazenamento e flexibilidade do modelo de dados orientado a documentos, se observa uma grande dificuldade do mapeamento de uma estrutura de um BD orientado a documentos para o modelo relacional.

A partir disso, este trabalho aborda o estudo de técnicas, obtidas através de trabalhos relacionados, apresentados durante o desenvolvimento deste trabalho, para o mapeamento de BDs orientados a documentos baseados no formato JSON para o modelo relacional. Além disso, este trabalho tem como diferencial identificar as características dos dados através de regras de mapeamento que suportam os três tipos de relações disponíveis no modelo relacional. Também é realizada a conversão para o modelo relacional através das regras de mapeamento, que é o foco deste trabalho.

2. Trabalhos Correlatos

A tabela 1 contém uma comparação entre os trabalhos correlatos reunidos no que se refere à diferenças em suas características de mapeamento.

Table 1. Comparação entre os trabalhos selecionados.

Trabalhos correlatos	**[Werner and lgorzata Bach 2018]**	**[Petković 2020]**	**[Maity 2018]**	**[Aftab 2020]**
Suporte a Múltiplas Tabelas	Sim	Não	Sim	Sim
Mapeamento autmático	Sim	Sim	Não	Sim
Identifica relações um-para-um	Não	Não	Sim	Não
Identifica relações um-para-muitos	Sim	Não	Não	Sim
Identifica relações muitos-para-muitos	Sim	Não	Não	Não

O estudo de [Werner and lgorzata Bach 2018] apresenta técnicas relevantes para o mapeamento da estrutura de dados do modelo orientado a documentos para o relacional, incluindo a identificação das cardinalidade um-para-muitos e muitos-para-muitos entre as tabelas geradas. Já o presente trabalho identifica os 3 tipos de cardinalidades, um-para-muitos, muitos-para-muitos e um-para-um, possibilitando um modelo mais fiel a estrutura dos dados contidos na origem.

Já o trabalho de [Petković 2020], apesar de apresentar 2 técnicas diferentes para a conversão, ambas realizam apenas a criação de uma tabela por documento, sendo que se a tabela possui array e documentos contidos dentro, são criadas colunas extras (atributos *ArrayID* e *ObjectID*), além de todas os demais atributos que possam estar contidos dentro do array e documento interno. Esta abordagem não normalizada acaba por gerar duplicação de dados pela falta de relações entre diferentes tabelas. O presente trabalho se destaca por possuir regras para a normalização dos dados, de forma a oferecer uma estrutura capaz de realizar a divisão dos dados em diferentes tabelas relacionadas entre si com a utilização de chaves primárias e estrangeiras, evitando redundâncias.

[Maity 2018] propôs uma solução semiautomática, que não leva em consideração tipos de dados complexos contidos em JSON, como por exemplo, arrays. Já [Aftab 2020] propôs uma solução muito completa. Ele apresentou uma forma de obter o esquema a partir de um documento e uma solução ETL para a carga de dados em BDs relacionais, onde objetos aninhados e arrays são convertidos em tabelas filhas e relações, onde são criadas chaves estrangeiras apontando para o objeto pai. Apesar disso, as relações entre as tabelas geradas não apresentam uma análise profunda dos dados, resultando em uma solução menos fiel ao modelo relacional, e portanto, uma falta de diferenciação das possíveis cardinalidades que podem estar presentes nos relacionamentos aninhados existentes em documentos JSON.

Como diferencial e contribuição adicional a todos os trabalhos correlatos apresentados, o presente trabalho distribui os atributos complexos presentes nos bancos de dados orientados a documentos em diferentes tabelas de forma a normalizar a estrutura e diminuir a redundância de dados. Além disso ele realiza o mapeamento de forma automática e apresenta regras para a definição de cardinalidade entre as relações das diferentes tabelas geradas a partir do mapeamento de dados complexos oriundos da BD de origem. Dessa forma, este trabalho procura abranger as distintas possibilidades oferecidas pelo modelo relacional e se manter fiel à proposta deste modelo.

3. Document2SQL

A solução desenvolvida se trata de um web service, que recebe como input um esquema gerado pela ferramenta externa *extract-mongo-schema*[1] de um BD no MongoDB. Com o esquema dentro da ferramenta, o mapeamento é executado em três processos distintos, primeiramente visando a criação de tabela e colunas a partir deste mapeamento e em um segundo processo, será realizado o mapeamento das relações entre estas tabelas. Por fim, em um terceiro processo, será realizada a criação do DDL para execução na base de dados CockroachDB.

O web service possui 3 serviços relacionados ao mapeamento. Possuindo como referência a figura 1, os processos enumerados são executados pelos seguintes serviços:

- **Serviço de Carregamento do Esquema**: 1.
- **Mapeamento da Estrutura**: 2, 3, 4.
- **Geração do código DDL**: 5.

3.1. Estrutura de Armazenamento do Mapeamento

De forma a dar suporte à estrutura mapeada, a aplicação utiliza um BD H2 de forma a armazenar a estrutura mapeada. Isso é feito a partir de entidades relacionadas as tabelas, colunas e relações geradas pelo mapeamento. Esta estrutura pode ser vista na figura 2.

3.2. Mapeamento de tabelas, colunas e limitações

A primeira parte deste processo é realizar a leitura do esquema do BD orientado a documentos. Para cada coleção contida no esquema é gerada uma tabela no esquema relacional. Feito isso, para cada uma dessas coleções, são verificadas suas chaves, que caso sejam de tipos simples, como string ou numérico, são definidas colunas para a tabela a

[1] https://github.com/perak/extract-mongo-schema

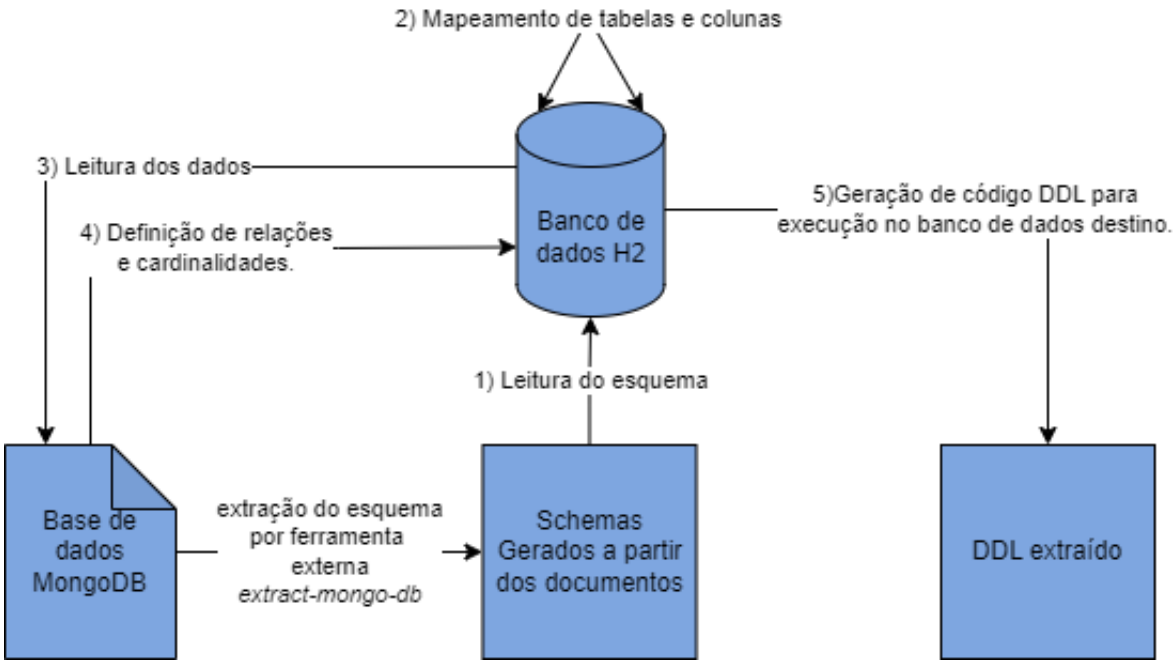


Figure 1. Processo executado pelo *document2sql*.

qual pertencem. Além disso, é verificado também se a chave possui atributos de limitação no esquema de documentos. Caso possua a chave ”key” verdadeira, como é possível observar na chave ”city” dentro da coleção ”zips”, exemplificada na Figura **??**, essa coluna é definida como única na tabela de dados destino, ou seja, nesta tabela não é permitida a persistência de 2 registros com dados de mesmo valor para esta coluna.

Já quando os documentos possuem chaves do tipo ”Object” ou ”Array”, que são tipos de dados complexos, uma outra abordagem é adotada. Aqui, em ambos os casos, identifica-se uma relação entre as tabelas. Sendo assim, tendo como exemplo para ”Object” a chave ”loc” contida dentro da coleção ”zips” na Figura **??**, é criada uma nova tabela chamada ”loc”, onde seus atributos são, por sua vez, analisados, e como é possível observar, são criadas outras 2 colunas ”y” e ”x” de tipos numéricos.

Três algoritmos descrevem esse subprocesso. O Algoritmo 1 a seguir lê o esquema de documentos e, para cada uma das coleções presentes nele, uma nova tabela é criada e passada como parâmetro para o Algoritmo 2 juntamente com o nodo JSON referente à coleção.

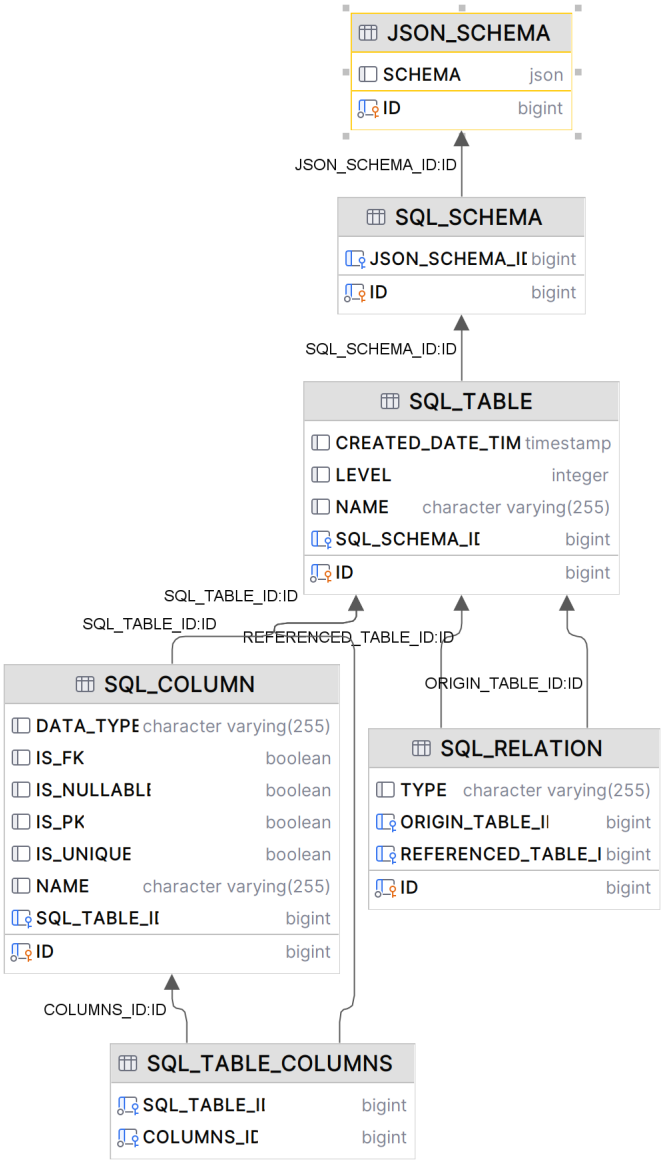


Figure 2. Diagrama de tabela da aplicação *document2sql* no BD H2.

Algorithm 1 Inicia mapeamento para o esquema relacional

function MAPTOSQL(jsonSchemaId)
    $jsonSchema \leftarrow getJsonSchemaById(jsonSchemaId)$;
    $node \leftarrow getSchemaJsonNodeInJsonSchema(jsonSchema)$;
    for each $tableName$ in $node.fieldNames$ **do**
        $tableName \leftarrow replaceSpacesWithUnderscores(tableName)$;
        $sqlTable \leftarrow newSqlTable(tableName)$;
        $sqlTable.setCreatedDateAndTime(newDate())$;
        $tableNode \leftarrow node.get(tableName)$;
        $createSqlObjects(tableNode, sqlTable)$;
        $dynamicMongoService.analyzeCollectionCardinality(tableName)$
    end for
end function

O Algoritmo 2 realiza a criação de uma chave primária do tipo UUID para a tabela. Feito isso, ele verifica todas as chaves presentes no documento juntamente com o seu tipo, e, caso não seja um ”Object” ou ”Array”, uma nova coluna é criada e associada à tabela passada como parâmetro e identificada como *parentTable*.

Algorithm 2 Cria objetos no esquema relacional a partir do documento

```
1: function CREATESQLOBJECTS(parentNode, parentTable)
2:     if not parentTable.getColumns().stream().anyMatch(SqlColumn :: isIsPk) then
3:         primaryKey ← newSqlColumn;
4:         primaryKey.setName(parentTable.getName() + "_gen_uuid");
5:         primaryKey.setIsPk(true);
6:         primaryKey.setDataType("UUID");
7:         primaryKey.setSqlTable(parentTable);
8:         parentTable.setColumn(primaryKey);
9:     end if
10:    for each fieldName in schema.fieldNames do
11:        fieldNameNode ← schema.get(fieldName);
12:        if fieldName ≠ "_id" then
13:            nodeType ← getNodeObjectType(fieldNameNode);
14:            type ← getStringObjectType(nodeType);
15:            if type ≠ OBJECTtype ≠ ARRAYtype ≠ "Null" then
16:                column ← newSqlColumn;
17:                setColumnAttributes(column, fieldNameNode, fieldName, type);
18:                parentTable.setColumn(column);
19:            else if type == OBJECT then
20:                childTable ← newSqlTable(parentTable.getName() + "__" +
   fieldName.replace("","_"));
21:                childNodeStructure ← getObjectStructure(fieldNameNode)
22:                createSqlObjects(childNodeStructure, childTable);
23:            else if type == ARRAY then
24:                childArrayTable ← newSqlTable(parentTable.getName() + "__" +
   fieldName.replace("","_"));
25:                arrayStructure ← getArrayStructure(fieldNameNode);
26:                createSqlObjectsArray(arrayStructure, childArrayTable)
27:            end if
28:        end if
29:    end for
30:    sqlTableRepository.save(parentTable)
31: end function
```

Caso a chave tenha um tipo ”Object”, o algoritmo realiza a criação de uma *childTable* com nome da tabela pai, concatenado com ”__” e concatenado com o nome da chave. Após, ele obtém o valor desta chave, que é um nodo JSON e chama novamente o Algoritmo 2, passando como parâmetro o nodo da *childTable* como o novo esquema a ser analisado, e a *childTable* criada como sendo a nova *parentTable*.

Caso a chave possua um tipo ”Array”, é criada uma nova *childTable* onde a estrutura deste array é obtida e passada como parâmetro para o Algoritmo 3, como um nodo JSON. Além disso, a *childTable* também é passada como o parâmetro *parentTable*.

Algorithm 3 Cria objetos no esquema relacional a partir de array

```
1: function CREATESQLOBJECTSARRAY(parentNode, parentTable)
2:     primaryKey ← newSqlColumn;
3:     primaryKey.setName(parentTable.getName() + "_gen_uuid")
4:     primaryKey.setIsPk(true);
5:     primaryKey.setDataType("UUID");
6:     primaryKey.setSqlTable(parentTable);
7:     parentTable.setColumn(primaryKey);
8:     for each type in schema.fieldNames do
9:         if type ≠ OBJECTtype ≠ ARRAYtype ≠ "Null" then
10:            column ← newSqlColumn;
11:            column.setName(parentTable.getName().replace("","_") + type);
12:            if type == "number" then
13:                column.setDataType("float")
14:            else
15:                column.setDataType(type)
16:            end if
17:            column.setSqlTable(parentTable);
18:            parentTable.setColumn(column)
19:        end if
20:        if type == OBJECT then
21:            structure ← getArrayObjectStructure(schema);
22:            createSqlObjects(structure, parentTable)
23:        end if
24:    end for
25:    sqlTableRepository.save(parentTable)
26: end function
```

O Algoritmo 3 repete o processo do algorimo anterior, realizando a criação de uma chave primária para a nova tabela e percorrendo seus atributos de forma a identificar seus tipos. Caso o tipo de uma chave não seja um ”Object”, ”Array” ou nulo, uma nova coluna é criada. Caso a chave seja do tipo ”Object”, a estrutura desta chave é obtida e passada como parâmetro para o Algoritmo 2 como um nodo JSON e a própria *parentTable* desta vez é passada como parâmetro *parentTable*, diferentemente do Algoritmo 2.

Os algoritmos trabalham recursivamente de forma a não limitar o número de objetos aninhados que possam estar presentes dentro de uma mesma coleção. Porém, uma limitação é que ele não executa vetores dentro de vetores. Desta forma, transformações devem ser realizadas na origem dos dados para evitar este tipo de cenário.

3.3. Identificação das Relações e Cardinalidade entre as Tabelas

O segundo subprocesso realiza uma análise dos dados na origem (BD MongoDB) de forma a identificar relacionamentos e suas cardinalidades.

Para o desenvolvimento deste trabalho foram definidas regras a serem aplicadas no BD de origem (MongoDB) de forma a mapear os relacionamentos e e suas cardinalidades entre seus diferentes objetos aninhados e arrays. As regras definidas para cada tipo de relacionamento são as seguintes:

1. Um-para-um:
   (a) ocorre quando um subobjeto não se repete em diferentes objetos pais, que por sua vez, não possuem mais de um subobjeto;

2. Um-para-muitos:
    (a) ocorre quando um subobjeto está presente em diferentes objetos pais, que por sua vez, não possuem mais de um subobjeto;
    (b) ocorre quando um objeto pai possui diferentes subobjetos, porém os subobjetos não possuem mais de um objeto pai;
    (c) ocorre quando um objeto em um array de objetos não se repete em diversos arrays;
3. Muitos-para-muitos:
    (a) ocorre quando um objeto em um array de objetos se repete em diversos arrays;
    (b) ocorre quando um subobjeto se repete para diferentes objetos pais e os objetos pais possuem diferentes subobjetos;
    (c) ocorre quando um valor em um array que não é de objetos se repete em diversos arrays.

A aplicação dessas regras é detalhada nos algoritmos a seguir. O subprocesso inicia pelo Algoritmo 4. Ele recebe como parâmetro um nome de coleção. Após, o BD MongoDB é acessado, de onde toda a coleção e seus documentos são retornados e analisados.

Algorithm 4 Inicia análise da cardinalidade

```
1: function ANALYZECARDINALITY
2:     database ← mongoTemplate.getDb()
3:     collection ← database.getCollection(collectionName)
4:     /*Inicializa HashMaps para estruturas de subdocumentos, arrays e ocorrências de documentos*/
5:     subDocumentStructureOccurrences ← newHashMap;
6:     arrayStructureOccurrences ← newHashMap;
7:     parentDocumentOccurrences ← newHashMap;
8:     parentDocumentOccurrencesPerFullKey ← newHashMap;
9:     documents ← collection.find()
10:    /*Chama o método analyzeDocument para cada documento*/
11:    for each document in documents do
12:        analyzeDocument(document
13:        , subDocumentStructureOccurrences
14:        , arrayStructureOccurrences
15:        , parentDocumentOccurrences
16:        , parentDocumentOccurrencesPerFullKey
17:        , null)
18:    end for
19:    identifyObjectCardinality(subDocumentStructureOccurrences,
       parentDocumentOccurrences, parentDocumentOccurrencesPerFullKey);
20:    identifyArrayCardinality(arrayStructureOccurrences)
21: end function
```

Para cada objeto presente na coleção, o Algoritmo 5 é chamado e o objeto é passado como parâmetro, juntamento com as estruturas de dados necessárias para a identificação de repetições de objetos. Neste algoritmo é realizada uma iteração por todas as chaves e seus respectivos valores. Para cada valor, alguns conjuntos de regras são aplicados. Se for do tipo objeto, então caso possua objetos aninhados e arrays, estes são removidos utilizando o Algoritmo 12 e, com o restante das suas chaves e valores, é gerado

Algorithm 5 Conta ocorrências de objetos nos documentos

```
1: function ANALYZEDOCUMENT(Object, subDocumentStructureOccurrences,
   arrayStructureOccurrences, parentDocumentOccurrences,
   parentDocumentOccurrencesPerFullKey, fullKey)
2:     for each (key, value) in document do
3:         fullKey ← empty
4:         if parentKey == null then
5:             fullKey ← key
6:         else
7:             fullKey ← panretKey.key
8:         end if
9:         documentHash ← generateTopLevelHash(document)
10:        if value.instanceOfObject then
11:            structureHash ← generateTopLevelHash(value)
12:            /*Soma um para cada ocorrência do subobjeto*/
13:            subDocumentStructureOccurrences
14:            .computeIfAbsent(fullKey, k− > newHashMap)
15:            .merge(structureHash, 1, sum)
16:            /*Soma um para cada vez que o objeto pai*/
17:            parentDocumentOccurrencesPerFullKey
18:            .computeIfAbsent(fullKey, k− > newHashMap)
19:            .merge(documentHash, 1, sum)
20:            /*Adiciona ocorrências de objetos pais para este subobjeto*/
21:            parentDocumentOccurrences
22:            .computeIfAbsent(structureHash, k− > newHashSet)
23:            .add(documentHash)
24:            /*Chama novamente o analyzeDocument passando o subobjeto, as estruturas de dados e e a
   chave*/
25:            analyzeDocument(value, subDocumentStructureOccurrences
26:            , arrayStructureOccurrences, parentDocumentOccurrences
27:            , parentDocumentOccurrencesPerFullKey, fullKey)
28:        else if valueinstanceOfArray then
29:            for iteminList do
30:                if item.instanceOf ≠ Arrayitem.instanceOf ≠ Object then
31:                    structureHash ← generateArrayHash(item)
32:                    /*Soma um para cada vez que o valor se repete*/
33:                    arrayStructureOccurrences
34:                    .computeIfAbsent(fullKey, k− > newHashMap)
35:                    .merge(structureHash, 1, sum)
36:                else if item.instanceofDocument then
37:                    structureHash ← generateTopLevelHash(item);
38:                    /*Soma um para cada vez que o subobjeto se repete*/
39:                    arrayStructureOccurrences
40:                    .computeIfAbsent(fullKey, k− > newHashMap)
41:                    .merge(structureHash, 1, sum)
42:                    /*Chama novamente o analyzeDocument passando o subobjeto, as estruturas de da-
   dos e e a chave*/
43:                    analyzeDocument(item
44:                    , subDocumentStructureOccurrences...parentDocumentOccurrencesPerFullKey
45:                    , fullKey)
46:                end if
47:            end for
48:        end if
49:    end for
50: end function
```

um texto e adicionado à um *HashMap*, somando o número de ocorrências, deste mesmo texto, para todas as ocorrências deste objeto. O mesmo é realizado com o objeto pai.

Após a análise de ocorrências de objetos e valores dentro de arrays, a execução retorna para o Algoritmo 4, que por sua vez chama os Algoritmos 6 e 7, que realizam a análise das estruturas de dados alimentadas e aplicam as regras as cardinalidades para os objetos e arrays, respectivamente.

Algorithm 6 Identifica a Cardinalidade dos Objetos e Subobjetos

```
1: function IDENTIFYOBJECTCARDINALITY(subDocumentStructureOccurrences,
   parentDocumentOccurrences, parentDocumentOccurrencesPerFullKey)
2:     for each (key, value) in document do
3:         for each (field, structureMap) in subDocumentStructureOccurrences do
4:             for each structureHash in structureMap.keySet() do
5:                 parentDocs ← parentDocumentOccurrences.get(structureHash)
6:                 if parentDocs ≠ null parentDocs > 1 then
7:                     if structureMap.values.anyMatch(count > 1)
8:                         parentDocumentOccurrencesPerFullKey.get(field).values.anyMatch(count >
   1) then
9:                         createManyToMany(field, collectionName)
10:                    else
11:                        createManyToOne(field, collectionName)
12:                    end if
13:                else if parentDocs ≠ null parentDocs == 1 then
14:                    if structureMap.values.anyMatch(count > 1)
15:                        parentDocumentOccurrencesPerFullKey.get(field).values.allMatch(count ==
   1) then
16:                        createOneToMany(field, collectionName)
17:                    else
18:                        createOneToOne(field, collectionName)
19:                    end if
20:                end if
21:            end for
22:        end for
23:
```

Algorithm 7 Identifica a Cardinalidade dos Arrays

```
1: function IDENTIFYARRAYCARDINALITY(arrayStructureOccurrences)
2:     for each (field, structureMap) in arrayStructureOccurrences do
3:         if structureMap.values().anyMatch(count > 1) then
4:             createManyToMany(field, collectionName)
5:         else
6:             createOneToMany(field, collectionName)
7:         end if
8:     end for
9: end function=0
```

Após a identificação das cardinalidades, os Algoritmos 8, 9, 10 e 11 são chamados para realizar a criação dos objetos no BD H2, que representam as relações entre as estruturas geradas pelo mapeamento, bem como suas chaves, e regras definidas, como por exemplo, a criação da tabela de associação para casos *muitos-para-muitos* ou a definição de chave estrangeira na chave primária de tabelas filhas para casos *um-para-muitos*.

Algorithm 8 Cria Relações Muitos-Para-Muitos

```
1: function CREATEMANYTOMANY(field, collectionName)
2:     parentTableName ← empty
3:     childTableName ← empty
4:     /*Verifica se é a raiz da coleção e identifica o nome das tabelas*/
5:     if field.contains(".") then
6:         parentTableName ← collectionName + "__" + field.substring(0, field.lastIndexOf(".")).replace(".", "__")
7:         childTableName ← collectionName + "__" + field.replace(".", "__")
8:     else
9:         parentTableName ← collectionName;
10:        childTableName ← collectionName + "__" + field.replace(".", "__");
11:    end if
12:    associationTable ← newSqlTable(parentTableName + "__R__" + childTableName)
13:    primaryKey ← newSqlColumn;
14:    primaryKey.setName(parentTable.getName() + "_gen_uuid");
15:    primaryKey.setIsPk(true);
16:    primaryKey.setDataType("UUID");
17:    primaryKey.setSqlTable(associationTable);
18:    associationTable.setColumn(primaryKey);
19:    parentTable ← sqlTableRepository.findByName(parentTableName);
20:    childTable ← sqlTableRepository.findByName(childTableName);
21:    parentRelation ← newSqlRelation();
22:    parentRelation.setOriginTable(associationTable);
23:    parentRelation.setReferencedTable(parentTable);
24:    parentRelation.setType("M − N");
25:    childRelation ← newSqlRelation();
26:    childRelation.setOriginTable(associationTable);
27:    childRelation.setReferencedTable(childTable);
28:    childRelation.setType("M − N");
29:    fkParentTable ← newSqlColumn();
30:    fkParentTable.setName(parentTable.getName() + "_id");
31:    fkParentTable.setFk(true);
32:    fkParentTable.setDataType("UUID");
33:    fkParentTable.setSqlTable(associationTable);
34:    associationTable.setColumn(fkParentTable);
35:    fkChildTable ← newSqlColumn();
36:    fkChildTable.setName(childTable.getName() + "_id");
37:    fkChildTable.setFk(true);
38:    fkChildTable.setDataType("UUID");
39:    fkChildTable.setSqlTable(associationTable);
40:    associationTable.setColumn(fkChildTable);
41:    sqlRelationRepository.save(parentRelation);
42:    sqlRelationRepository.save(childRelation)
43: end function
```

Algorithm 9 Cria Relações Muitos-Para-Um

```
1: function CREATESMANYTOONE(field, collectionName)
2:     parentTableName ← empty
3:     childTableName ← empty
4:     /*Verifica se é a raiz da coleção e identifica o nome das tabelas*/
5:     if field.contains(".") then
6:         childTableName ← collectionName + "__" + field.substring(0, field.lastIndexOf(".")).replace(".", "__")
7:         parentTableName ← collectionName + "__" + field.replace(".", "__")
8:     else
9:         childTableName ← collectionName;
10:        parentTableName ← collectionName + "__" + field.replace(".", "__");
11:    end if
12:    parentTable ← sqlTableRepository.findByName(parentTableName);
13:    childTable ← sqlTableRepository.findByName(childTableName);
14:    sqlRelation ← newSqlRelation();
15:    sqlRelation.setOriginTable(childTable);
16:    sqlRelation.setReferencedTable(parentTable);
17:    sqlRelation.setType("N − 1");
18:    foreignKey ← newSqlColumn();
19:    foreignKey.setName(parentTable.getName() + "_id");
20:    foreignKey.setFk(true);
21:    foreignKey.setDataType("UUID");
22:    foreignKey.setSqlTable(childTable);
23:    childTable.setColumn(foreignKey);
24:    sqlRelationRepository.save(sqlRelation);
25:    sqlTableRepository.save(childTable)
26: end function
```

Algorithm 10 Cria Relações Um-Para-Muitos

```
1: function CREATESONETOMANY(field, collectionName)
2:     parentTableName ← empty
3:     childTableName ← empty
4:     /*Verifica se é a raiz da coleção e identifica o nome das tabelas*/
5:     if field.contains(".") then
6:         parentTableName ← collectionName + "__" + field.substring(0, field.lastIndexOf(".")).replace(".", "__")
7:         childTableName ← collectionName + "__" + field.replace(".", "__")
8:     else
9:         parentTableName ← collectionName;
10:        childTableName ← collectionName + "__" + field.replace(".", "__");
11:    end if
12:    parentTable ← sqlTableRepository.findByName(parentTableName);
13:    childTable ← sqlTableRepository.findByName(childTableName);
14:    sqlRelation ← newSqlRelation();
15:    sqlRelation.setOriginTable(childTable);
16:    sqlRelation.setReferencedTable(parentTable);
17:    sqlRelation.setType("N − 1");
18:    foreignKey ← newSqlColumn();
19:    foreignKey.setName(parentTable.getName() + "id");
20:    foreignKey.setFk(true);
21:    foreignKey.setDataType("UUID");
22:    foreignKey.setSqlTable(childTable);
23:    childTable.setColumn(foreignKey);
24:    sqlRelationRepository.save(sqlRelation);
25:    sqlTableRepository.save(childTable)
26: end function
```

Algorithm 11 Cria Relações Um-Para-Um

```
1: function CREATESONETOONE(field, collectionName)
2:     parentTableName ← empty
3:     childTableName ← empty
4:     /*Verifica se é a raiz da coleção e identifica o nome das tabelas*/
5:     if field.contains(".") then
6:         parentTableName ← collectionName + "__" + field.substring(0, field.lastIndexOf(".")).replace(".", "__")
7:         childTableName ← collectionName + "__" + field.replace(".", "__")
8:     else
9:         parentTableName ← collectionName;
10:        childTableName ← collectionName + "__" + field.replace(".", "__");
11:        parentTable ← sqlTableRepository.findByName(parentTableName);
12:        childTable ← sqlTableRepository.findByName(childTableName);
13:        sqlRelation ← newSqlRelation();
14:        sqlRelation.setOriginTable(childTable);
15:        sqlRelation.setReferencedTable(parentTable);
16:        sqlRelation.setType("1 − 1");
17:        for each columninchildTable.getColumns do
18:            /*Seta a chave primária da tabela filha como a chave estrangeira*/
19:            if column.isPk() then
20:                column.setIsFk(true);
21:                sqlColumnRepository.save(column)
22:            end if
23:        end for
24:        sqlRelationRepository.save(sqlRelation);
25:        sqlTableRepository.save(childTable)
26:
```

O Algoritmo 5 trabalha de forma similar com os vetores, porém, utilizando o Algoritmo 12 para realizar a limpeza dos objetos e vetores aninhados. Com a execução do Algoritmo 5, retorna-se ao Algoritmo 4, onde as estruturas de dados geradas são analisadas de acordo com as regras definidas para o mapeamento de cardinalidades.

Algorithm 12 Exclui vetores e documentos aninhados de documentos e gera hash apenas com atributos simples.

```
1: function GENERATETOPLEVELHASH(document)
2:     ret ← empty;
3:     for docindocument do
4:         if doc.instaceOf ≠ Objectdoc.instaceOf ≠ Array then
5:             ret ← ret + doc
6:         end if
7:     end for
8:     return retornullifempty
9: end function
10: function GENERATEARRAYHASH(array)
11:     ret ← empty;
12:     for docindocument do
13:         if doc.instaceOf ≠ Objectdoc.instaceOf ≠ Array then
14:             ret ← ret + doc
15:         end if
16:     end for
17:     return retornullifempty
18: end function=0
```

De acordo com cada regra, objetos da relação são criados relacionando a tabela pai e a tabela filha, armazenando também o tipo de sua relação. A exceção é a relação muitos para muitos, onde uma terceira tabela é criada (tabela associativa), que possui como coluna sua chave primária e mais 2 chaves estrangeiras que fazem referência às tabelas definidas anteriormente.

3.4. Extração das Instruções DDL

A última etapa realizada pela ferramenta é a geração do código SQL DDL a ser executado no BD relacional de destino, respeitando a sintaxe do BD escolhido, que, neste caso, é o BD relacional NewSQL CockroachDB. Esse código DDL é responsável pela criação do BD relacional.

Os algoritmos definidos exploram as tabelas, colunas e relacionamos identificados nas etapas anteriores para realizar a criação do BD relacional no CockroachDB. O processo inicia com o Algoritmo 13, onde todas as tabelas são retornadas do BD H2 e percorridas para a geração do código *DDL* com a execução do Algoritmo 14, responsável pelo geração de tabelas, suas colunas e seus relacionamentos.

Algorithm 13 Retorna o esquema DDL

```
1: function GETSQLSCHEMADDL
2:     tables ← FINDALL(sqlTableRepository);
3:     tableRelations ← MAPTABLERELATIONS;
4:     ddl ← newStringBuilder();
5:     for all tableintables do
6:         tableDdl ← CREATETABLEDDL(table, tableRelations.get(table.getId()));
7:         ddl.APPEND(tableDdl);
8:         ddl.APPEND("\n")
9:     end for
10:    return ddl.TOSTRING
11: end function
```

Na sequência, a instrução *DDL* gerada para uma tabela é retornada ao Algoritmo 13, sendo adicionado a uma *String* de instruções *DDL*. Ao final, o conjunto de instruções DDL para todas as tabelas é retornado ao usuário.

Algorithm 14 Gera os comandos DDL

```
1: function CREATETABLEDDL(table, relations)
2:     ddl ← newStringBuilder();
3:     ddlIndex ← newStringBuilder();
4:     Initializevariablesforforeignkeyanduniquecolumns;
5:     ddl.Append("CREATE TABLE");
6:     for all columnintable.GETCOLUMNS do
7:         ddl.Append("column");
8:         if column.isPk == true then
9:             ddl.Append("PRIMARY KEY")
10:        end if
11:        if column.isFk == true then
12:            ddl.Append("FOREIGN KEY")
13:            ddlIndex.Append("Index")
14:        end if
15:    end for
16:    ddl ← ddl + ddlIndex
17:    return ddl.TOSTRING
18: end function
```

4. Demonstração

Esta sessão apresenta a demonstração dos esquemas do modelo relacional gerados pela ferramenta *Document2SQL*.

Essa demonstração tem como objetivo apresentar a geração das cardinalidades e estruturas no CockroachDB a partir das regras de mapeamento definidas por este trabalho. Sendo assim, foram criadas no MongoDB as seguintes coleções de um BD *exemplo* (ver Figura 3), de forma a validar as diferentes regras de mapeamento: *pessoas*, *professores*, *biblioteca*, *disciplinas*, *artigos*, *situacoes* e *trabalhos*. Foram também criados documentos para cada uma destas coleções para fins de estudo de caso e validação dos cenários de mapeamento.

4.1. Pessoas

A coleção pessoas, que pode ser visualizada na Figura 4, é um exemplo de relação *um-para-um*, pois o objeto dentro de passaporte não se repete para as demais pessoas.

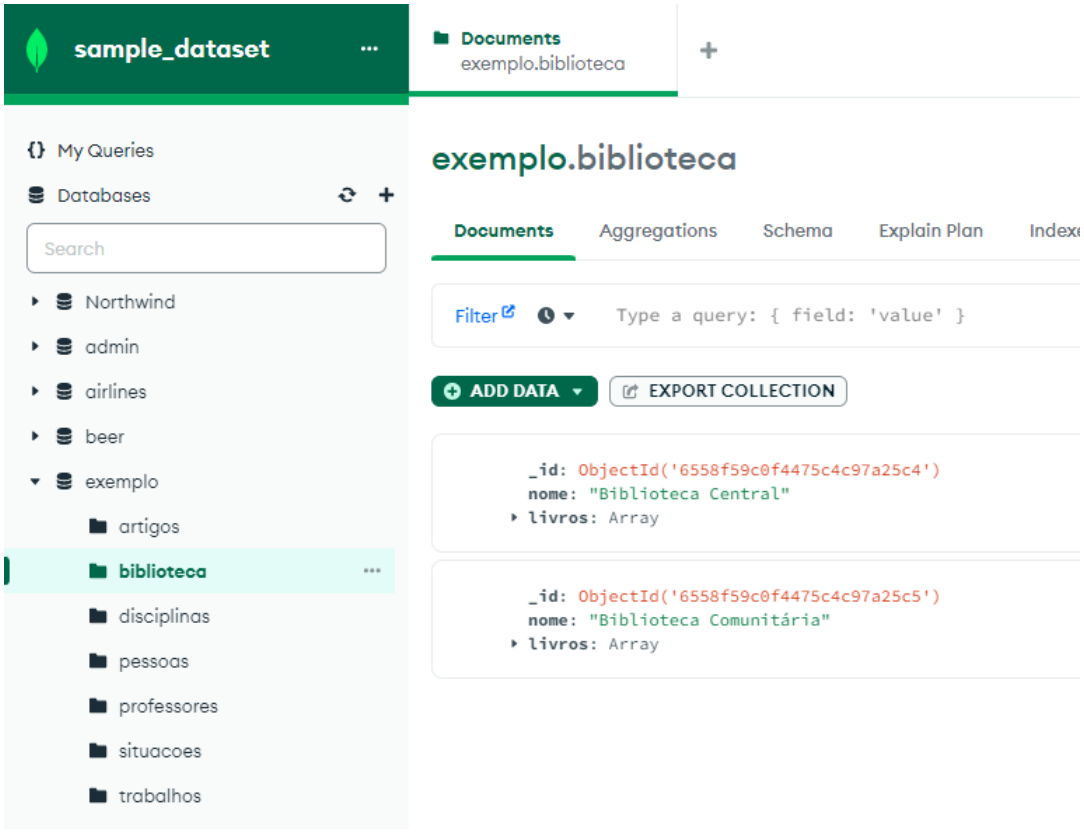


Figure 3. BD e coleções presentes em um servidor MongoDB.

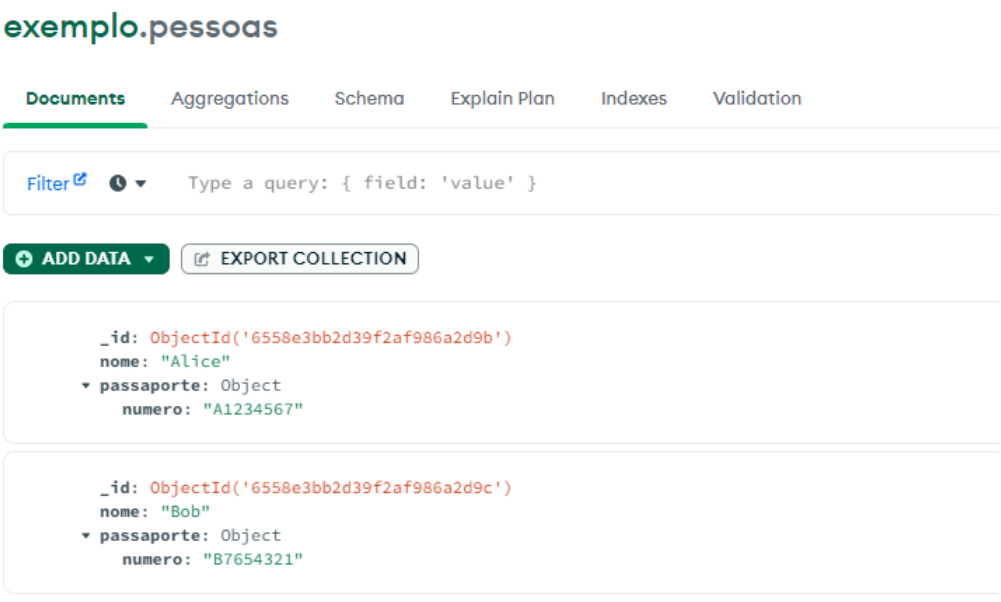


Figure 4. Coleção *pessoas* no BD MongoDB.

De forma a implementar esta relação no BD destino, foi realizada a criação de uma chave estrangeira na tabela filha *pessoas__passaporte* a partir da chave primária da mesma

tabela *passaporte*, que referencia a tabela pai *pessoas*. O diagrama com essa relação pode ser visto na Figura 5 e a estrutura gerada pode ser vista na Figura 6.

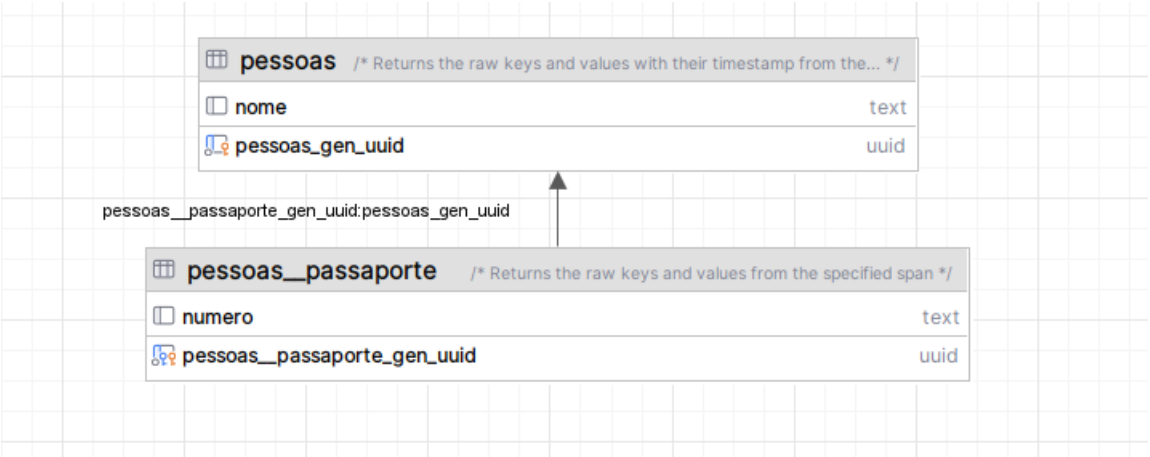


Figure 5. Diagrama de tabela da tabela pai *pessoas* e tabela filha *pessoas__passaporte*.

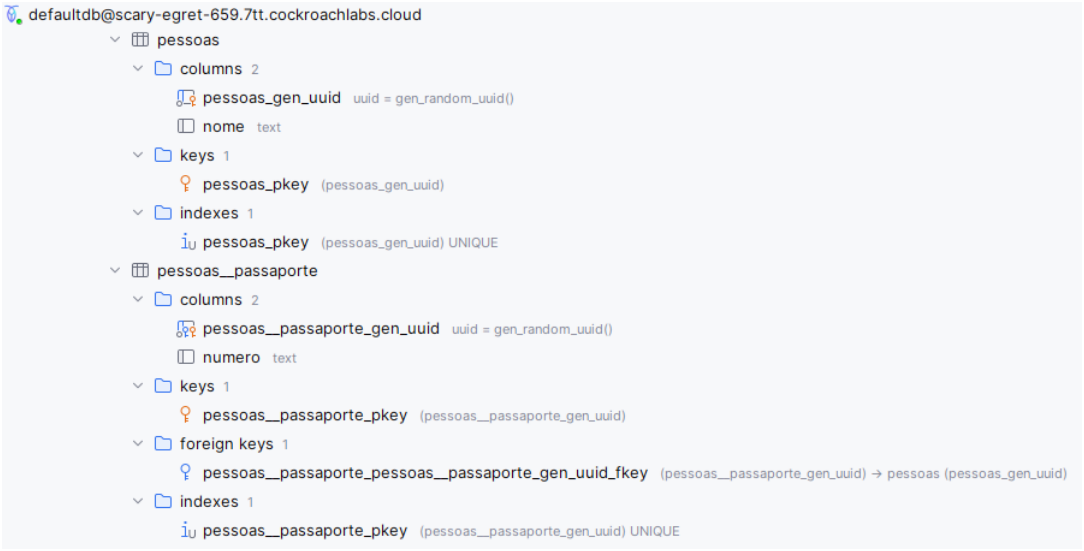


Figure 6. Estrutura das tabelas *pessoas* e *pessoas__passaporte* geradas no CockroachDB.

4.2. Professores

A coleção *professores*, visualizada na Figura 8, é um exemplo de relação *um-para-muitos*, pois nenhum objeto pertencente ao vetor de *disciplinas* se repete em outro documento de *professores*.

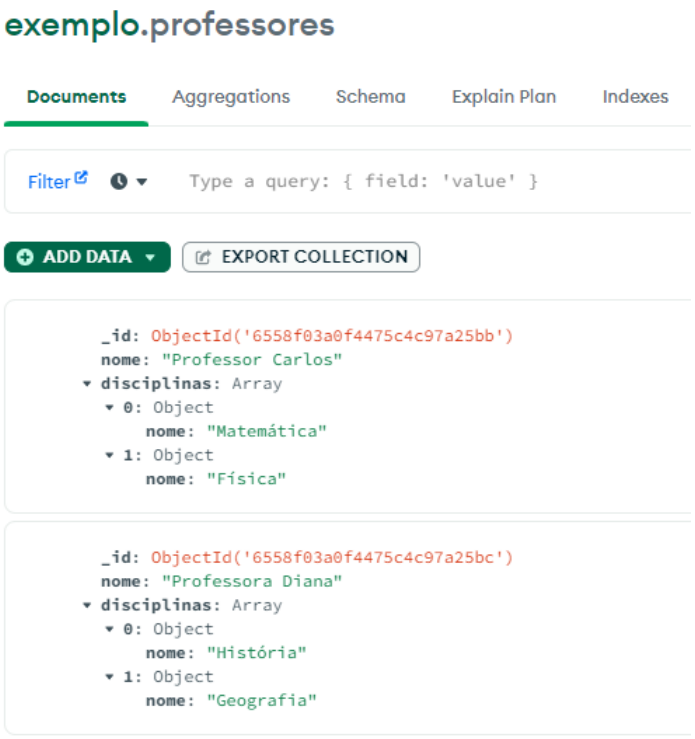


Figure 7. Coleção *professores* no BD MongoDB.

De forma a implementar esta relação, uma chave estrangeira *professores_id* foi criada na tabela filha professores_disciplinas, que referencia a tabela pai *professores*. Esta relação pode ser visualizada no diagrama da Figura 8 e a estrutura gerada pode ser vista na Figura 9.

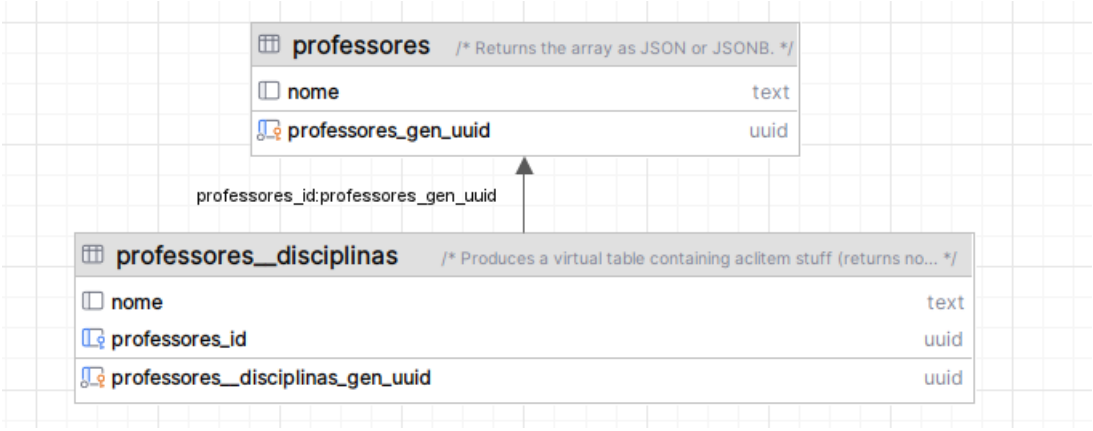


Figure 8. Diagrama da tabela pai *professores* e tabela filha *professores__disciplinas*.

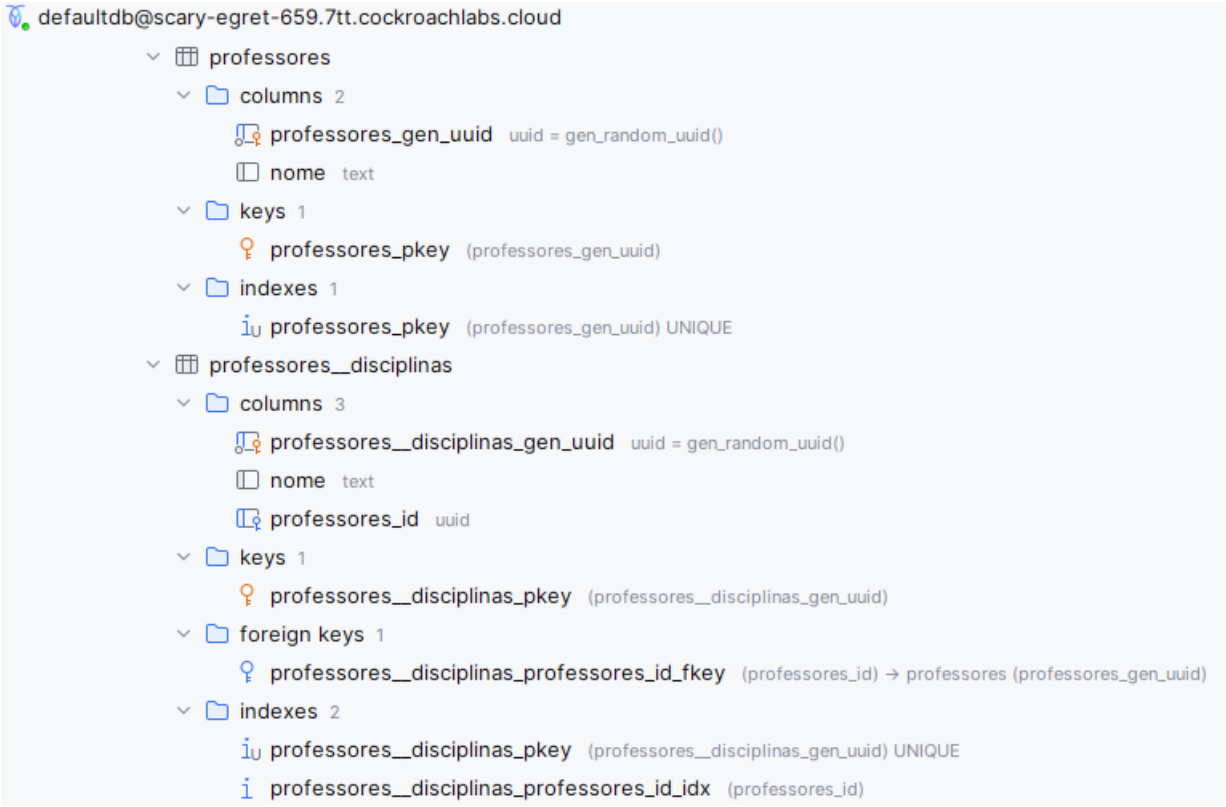


Figure 9. Estrutura das tabelas *professores* e *professores__disciplinas* geradas no CockroachDB.

4.3. Disciplinas

A coleção *disciplinas*, visualizada na Figura 10, apresenta também um exemplo de relacionamento *muitos-para-muitos*, pois objetos dentro do vetor *estudantes* se repetem dentro de vetores *estudantes* de outras *disciplinas*.

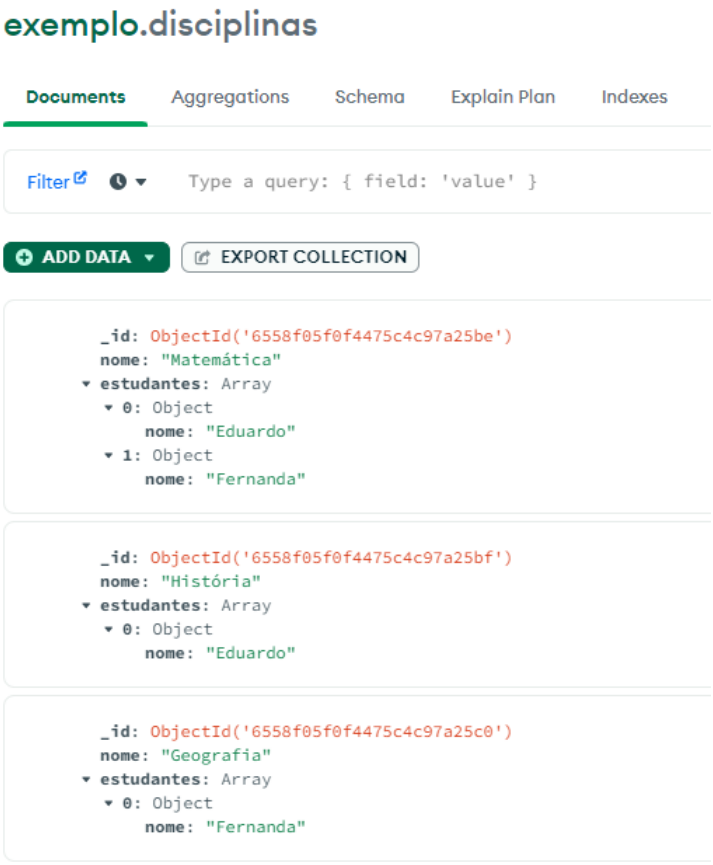


Figure 10. Coleção *disciplinas* no BD MongoDB.

De forma a implementar esta relação no modelo relacional, foi realizada a criação de uma terceira tabela (tabela associativa), onde foram criadas também 2 chaves estrangeiras: uma fazendo referência à tabela *disciplinas* e outra à tabela *disciplinas__estudantes* de forma a permitir que muitos estudantes participassem de muitas disciplinas e muitas disciplinas possuíssem muitos estudantes.

O diagrama representando esta relação pode ser visto na Figura 11. As estruturas geradas no BD CockroachDB podem ser vistas nas Figuras 12, 13 e 14.

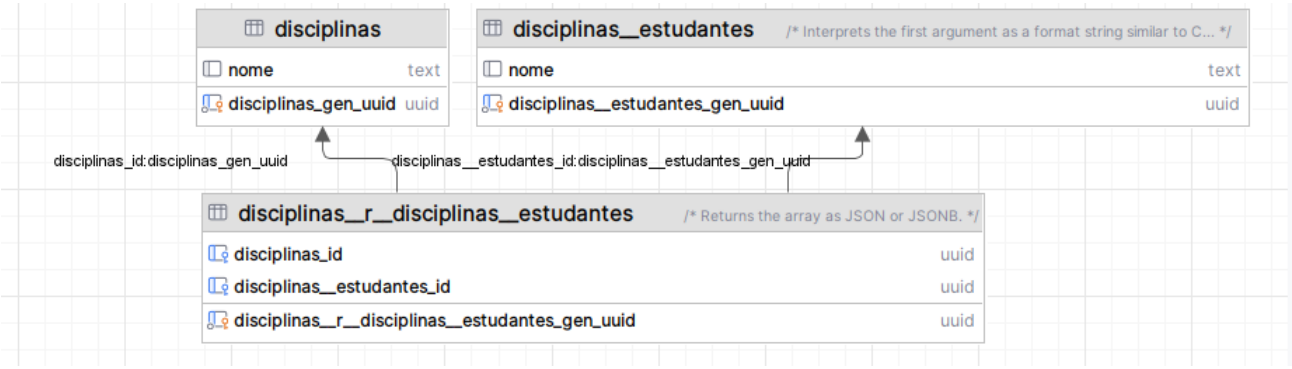


Figure 11. Diagrama de tabela das tabelas *disciplinas*, *estudantes* e da tabela associativa *disciplinas__r__disciplinas_estudantes*.

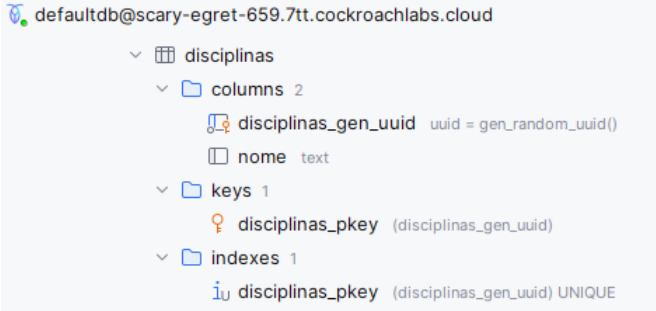


Figure 12. Estrutura da tabela *disciplinas* gerada no CockroachDB.

Figure 13. Estrutura da tabela *disciplina__estudantes* gerada no CockroachDB.

Figure 14. Estrutura da tabela *disciplina__r__estudantes* gerada no CockroachDB.

5. Biblioteca

A coleção *biblioteca*, visualizada na Figura 15, é um exemplo de relação *muitos-para-muitos*, pois pertence à um vetor onde valores se repetem em diferentes vetores pais. Sendo assim, de forma a normalizar a estrutura dentro do modelo relacional, uma tabela associativa é criada para prevenir a redundância de dados.

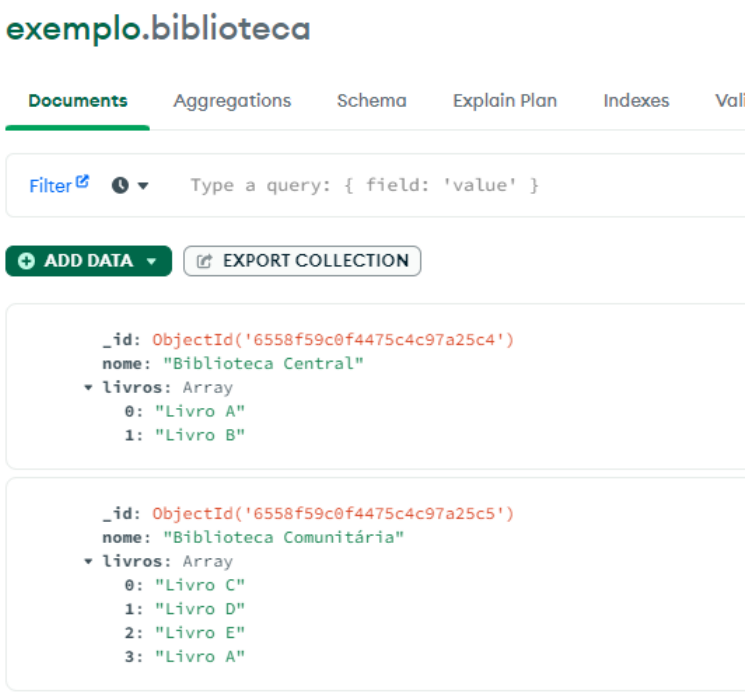


Figure 15. Coleção *biblioteca* no BD MongoDB.

A implementação deste exemplo segue o padrão de de relações *muitos-para-muitos*, onde a tabela associativa *biblioteca__r__biblioteca_livros* é criada possuindo uma chave primária e duas chaves estrangeiras apontando para as tabelas *biblioteca__livros* e *biblioteca*. Um diagrama demonstrando essa relação pode ser visto na Figura 16 e a estrutura gerada pela criada no BD CockroachDB pode ser vista nas Figuras 17, 18 e 19.

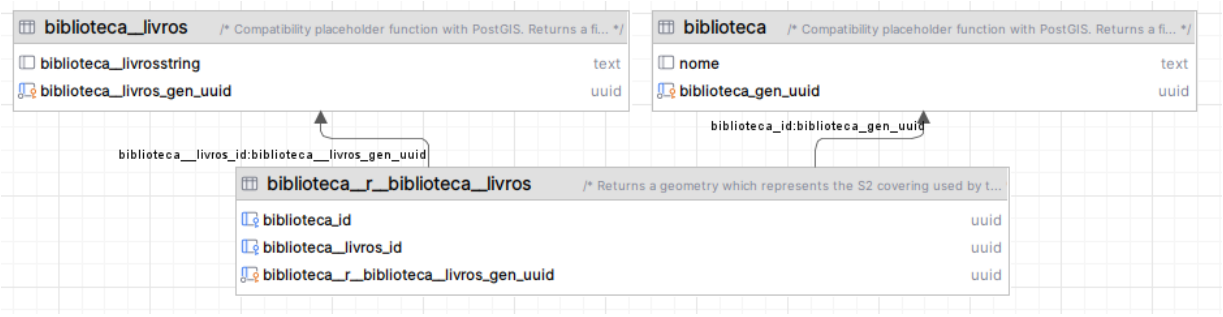


Figure 16. Diagrama de tabela da tabela de junção *biblioteca__r__biblioteca_livros* e das tabelas *biblioteca__livros* e *biblioteca*.

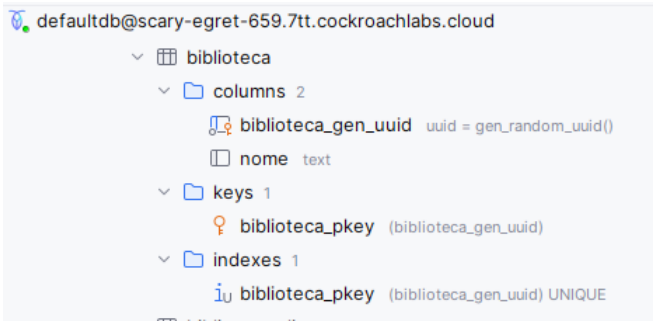


Figure 17. Estrutura da tabela *biblioteca* gerada, no CockroachDB.

Figure 18. Estrutura da tabela *biblioteca__livros* gerada, no CockroachDB.

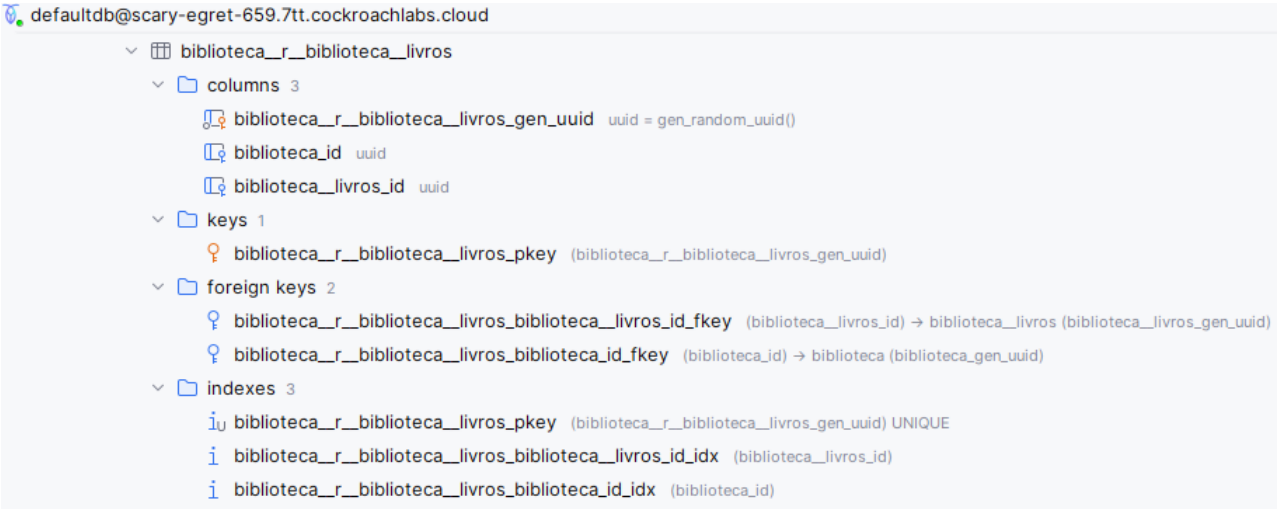


Figure 19. Estrutura da tabela *biblioteca__r__biblioteca_livros* gerada, no CockroachDB.

5.1. Artigos

A coleção artigos, visualizada na Figura 20, se encaixa em um exemplo de relacionamento *um-para-muitos*, porém de forma inversa. É possível observar que o mesmo objeto *autor* se repete dentro diversos *artigos*, porém não existem diferentes autores para o mesmo artigo.

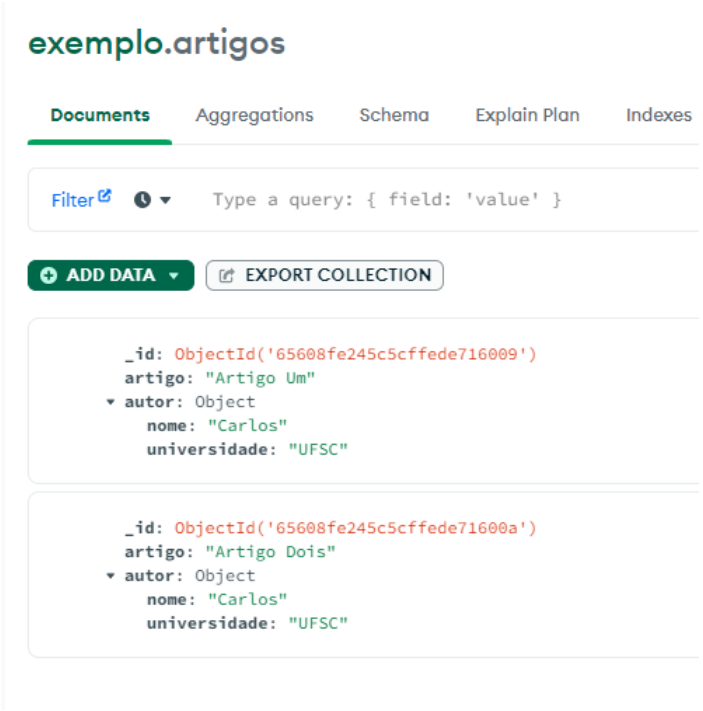


Figure 20. Coleção *artigos* no BD MongoDB.

Sendo assim, foi criada a tabela *artigos*, contendo sua estrutura original, bem como uma chave estrangeira fazendo referência à tabela pai *artigos__autor*. O diagrama representando esta relação pode ser visualizado na Figura 21 e as estruturas das tabelas geradas podem ser identificadas na Figura 22.

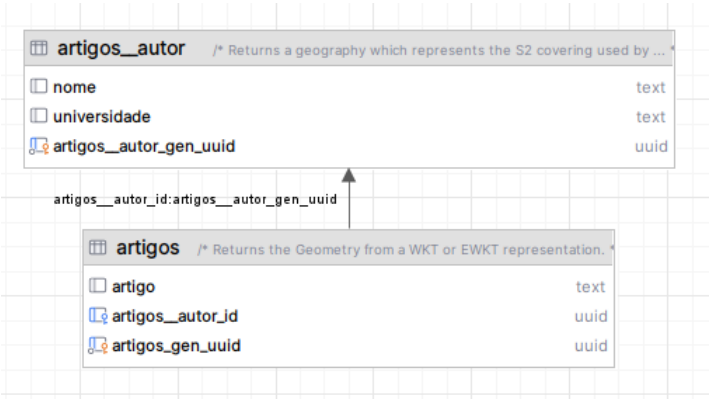


Figure 21. Diagrama de tabela da tabela pai *artigos__autor* e tabela filha *artigos*.

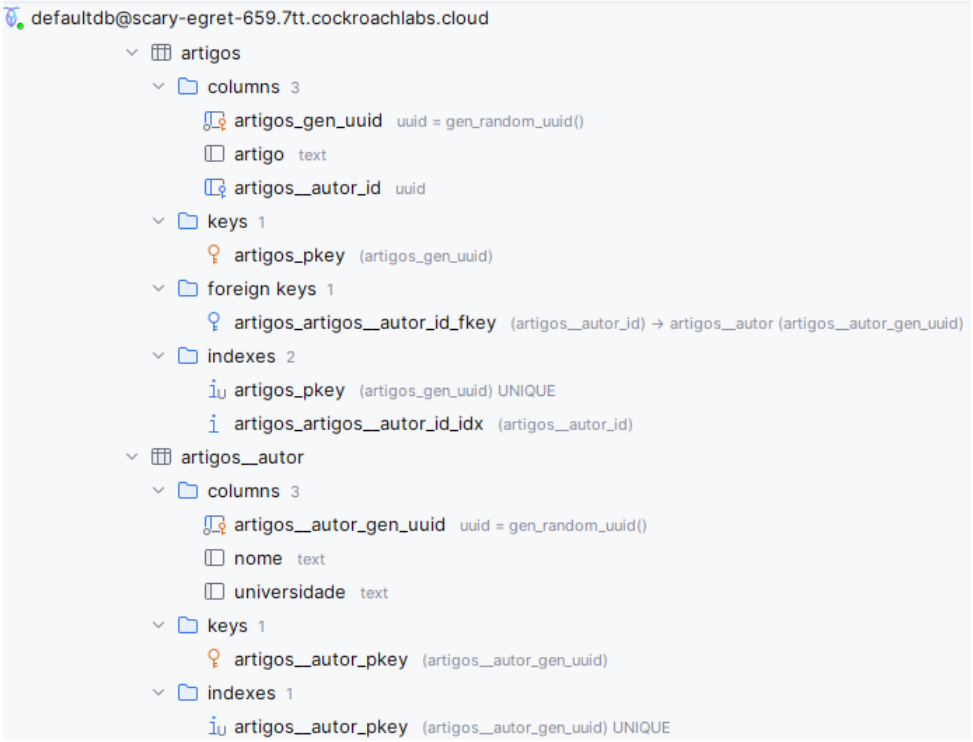


Figure 22. Estrutura das tabelas *artigos__autor* e *artigos* geradas no CockroachDB.

5.2. Situações

A coleção *situacoes*, visualizada na Figura 23, é um exemplo de relação *muitos-para-muitos*, pois os mesmos objetos possuem diferentes subobjetos e os mesmos subobjetos podem pertencer a diferentes objetos pais.

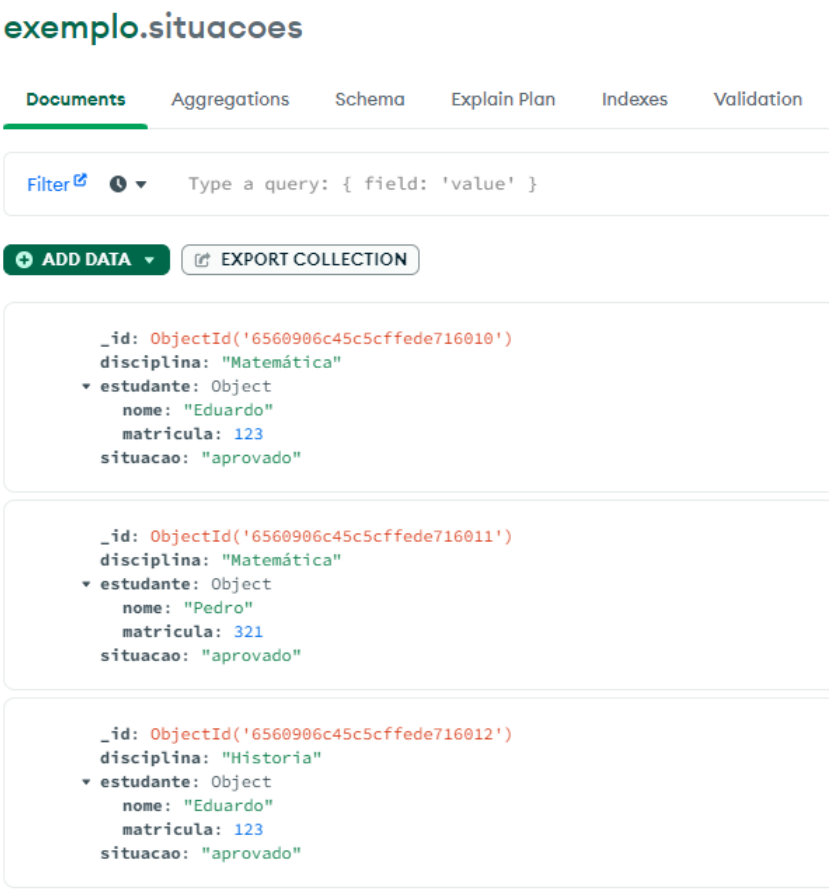


Figure 23. Coleção *situacoes* no BD MongoDB.

A implementação deste exemplo segue o padrão de relações *muitos-para-muitos*, onde, a tabela associativa *situacoes__r__situacoes_estudante* é criada possuindo uma chave primária e 2 chaves estrangeiras apontando para as tabelas *situacoes__estudante* e *situacoes*. O diagrama representando as relações pode ser visto na Figura 24. Além disso a estrutura das tabelas criadas no BD Cockroach pode ser vista nas Figuras 25, 26 e 27.

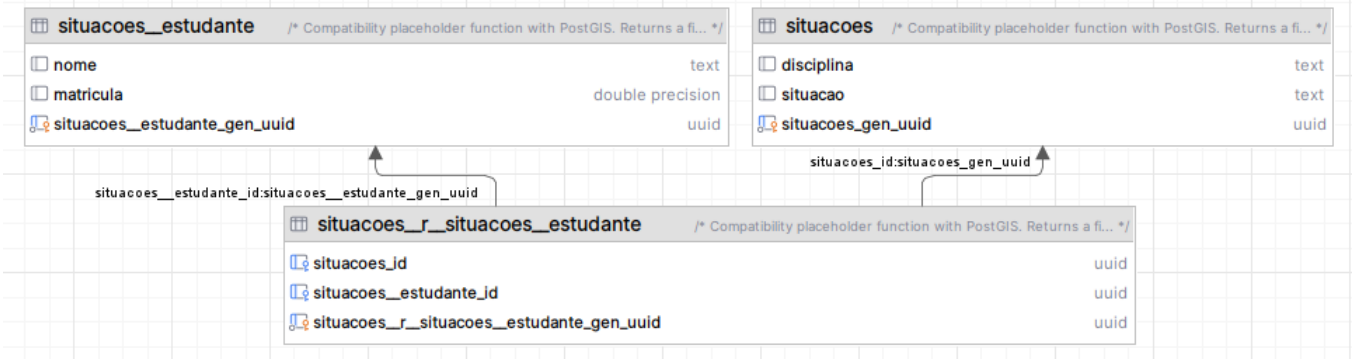


Figure 24. Diagrama de tabela da tabela de associação *situacoes__r__situacoes__estudante* e das tabelas filhas *situacoes__estudante* e *situacoes*

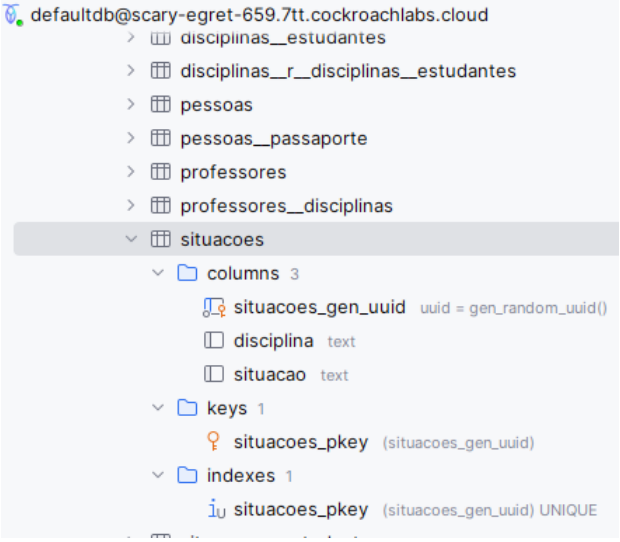


Figure 25. Estrutura da tabela *situacoes* gerada no CockroachDB.

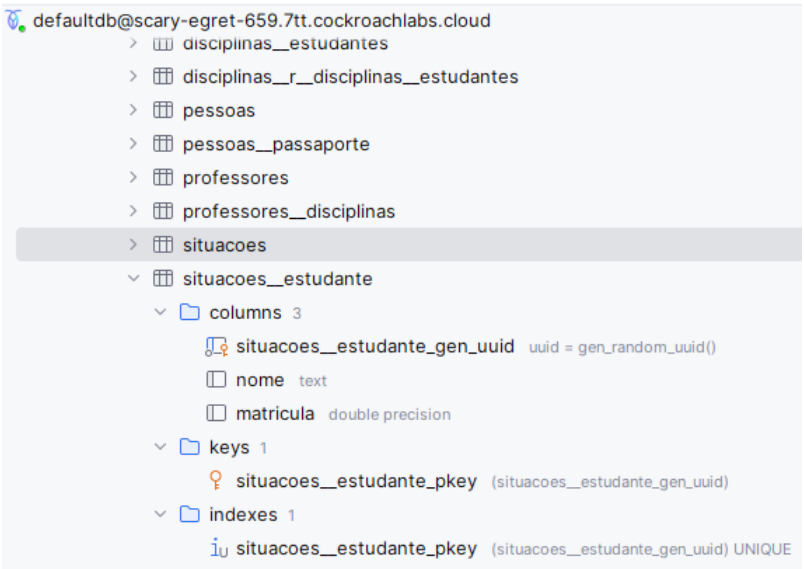


Figure 26. Estrutura da tabela *situacoes__estudante* gerada no CockroachDB.

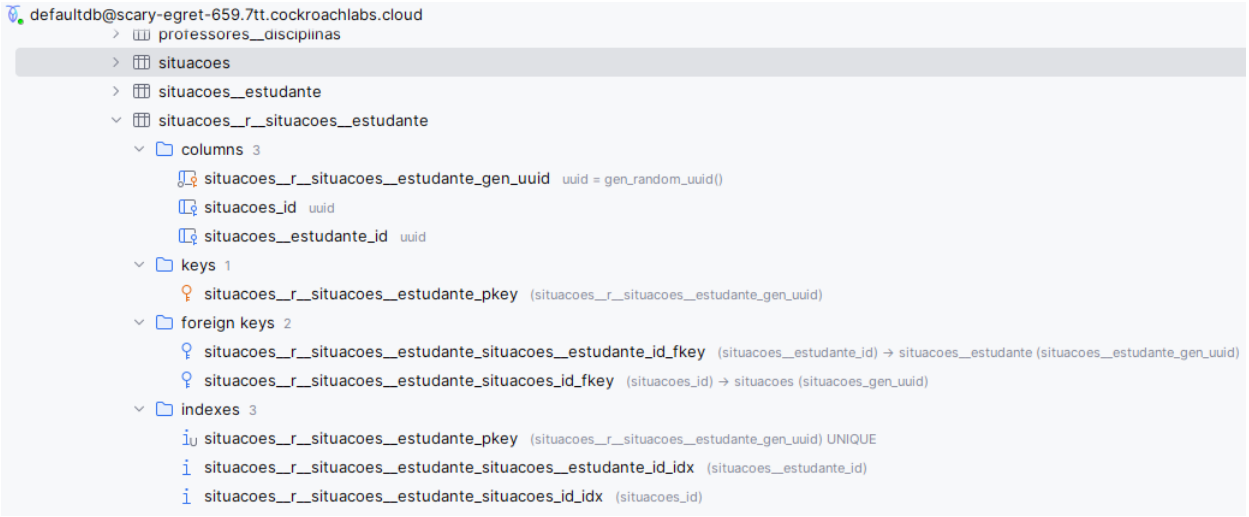


Figure 27. Estrutura da tabela associativa *situacoes__r__situacoes_estudante* gerada no CockroachDB.

5.3. Trabalhos

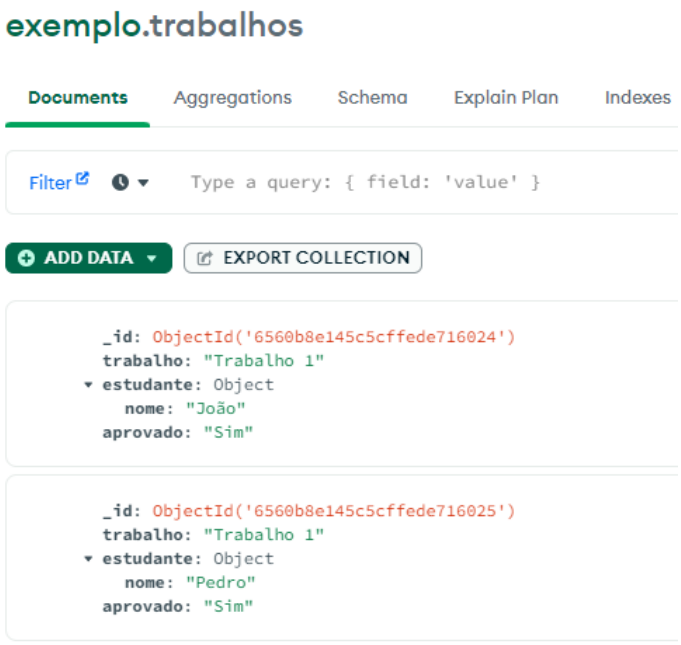


Figure 28. Coleção *trabalhos* no BD MongoDB.

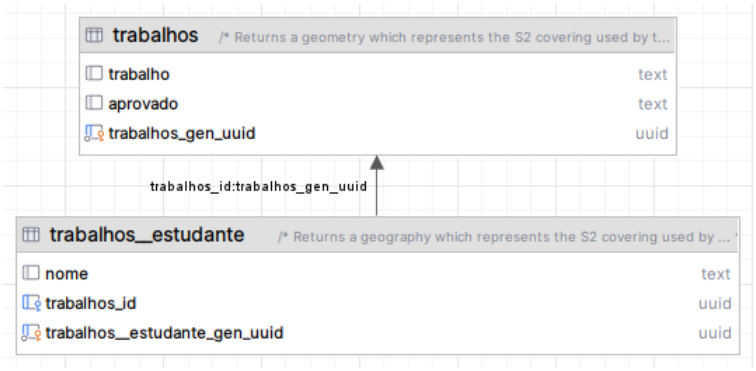


Figure 29. Diagrama de tabela da tabela pai *trabalhos* e da tabela filha *trabalhos__estudante*.

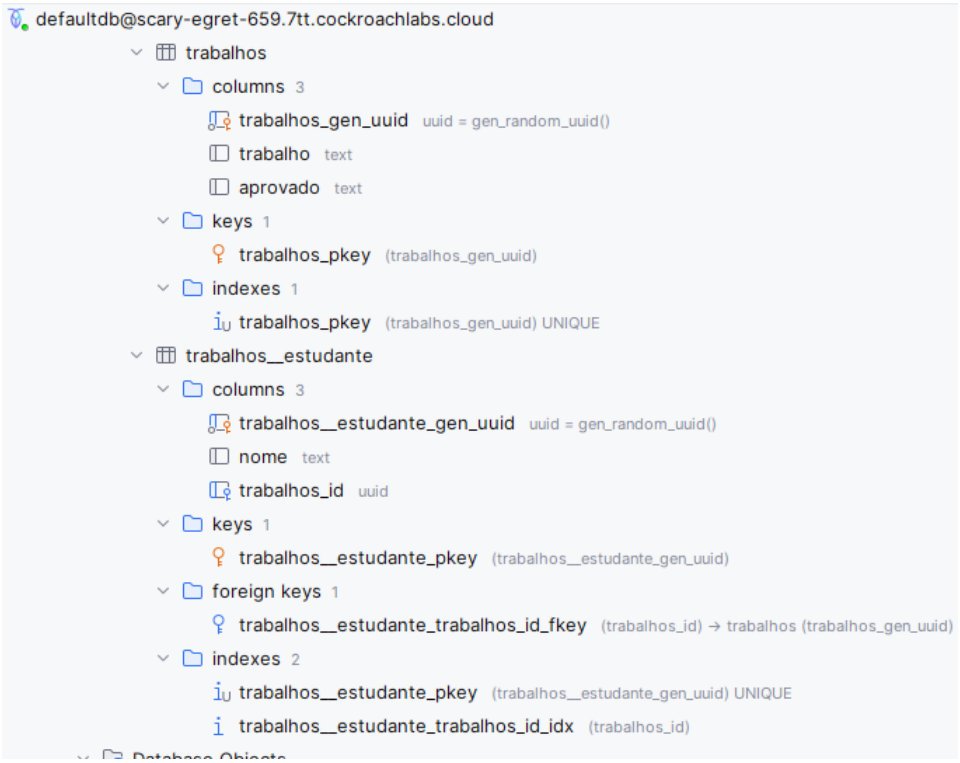


Figure 30. Estrutura das tabelas *trabalhos* e *trabalhos__estudante* geradas, no CockroachDB.

6. Avaliação

A metodologia da avaliação consistiu em executar um conjunto de consultas predefinidas em ambos os BDs. As seguintes classes de consultas foram consideradas: consultas simples, consultas com junções, consultas com agregações e ordenação dos dados. De forma a atender estes critérios levantados, um BD JSON no domínio de *Filmes* foi carregado no MongoDB. Além disso, ambos os BDs estavam operando localmente em um mesmo dispositivo com as seguintes configurações:

- Processador 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz;

- RAM instalada 8,00 GB (utilizável: 7,74 GB).

O BD escolhido[2] reúne dados de filmes, como ano, atores e gêneros em um arquivo JSON com 36273 documentos. Exemplos de documentos podem ser vistos na Figura 31.

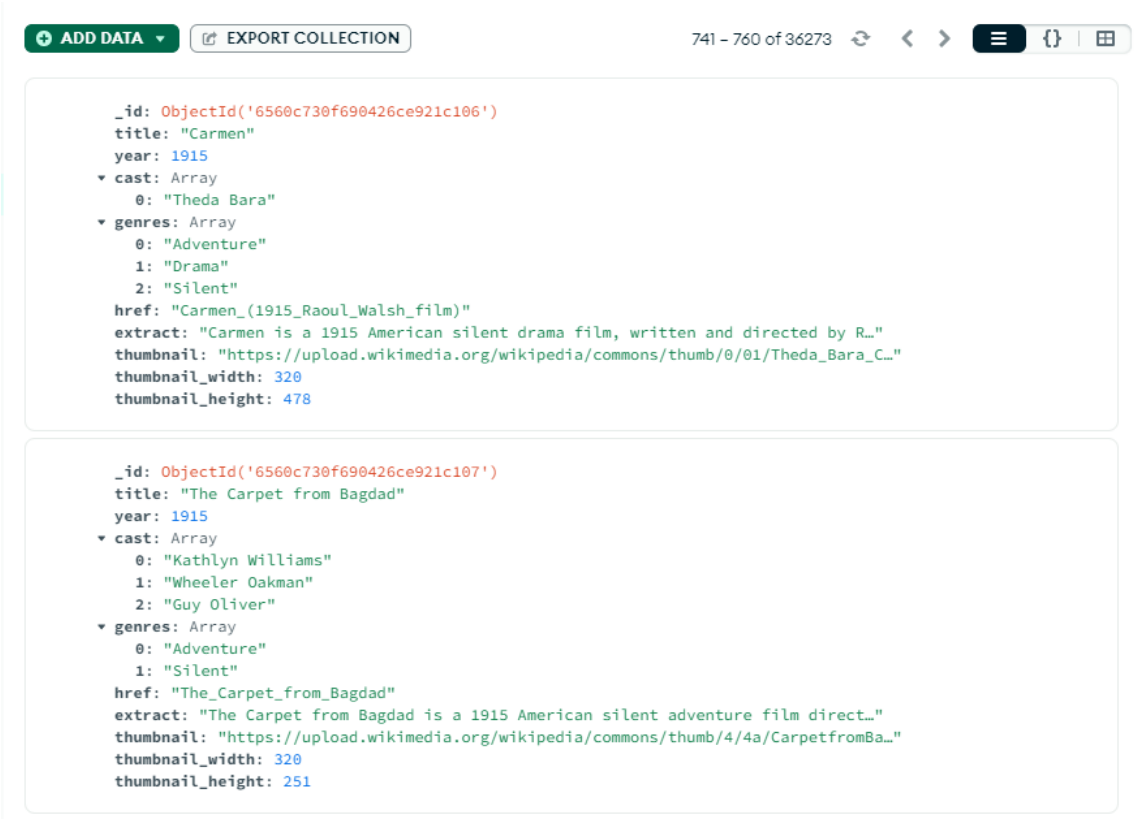


Figure 31. Exemplos de documentos sobre filmes.

Então seu esquema foi extraído utilizando a ferramenta *extract-mongo-schema*, onde foram identificadas as estruturas do BD de origem, bem como seus tipos de dados. O esquema extraído é mostrado nas Figuras 32 e 33.

[2]https://raw.githubusercontent.com/prust/wikipedia-movie-data/master/movies.json

```
{
    "movies": {
        "_id": {
            "types": {
                "Object": {
                    "frequency": 36273
                }
            },
            "primaryKey": true
        },
        "title": {
            "types": {
                "string": {
                    "frequency": 36273
                }
            },
            "key": true
        },
        "year": {
            "types": {
                "number": {
                    "frequency": 36273
                }
            }
        },
        "cast": {
            "types": {
                "Array": {
                    "frequency": 36273,
                    "structure": {
                        "types": {
                            "string": {
                                "frequency": 133326
                            }
                        }
                    }
                }
            }
        },
        "genres": {
            "types": {
                "Array": {
                    "frequency": 36273,
                    "structure": {
                        "types": {
                            "string": {
                                "frequency": 64228
                            }
```


Figure 32. Extração do esquema do BD de filmes(1/2).

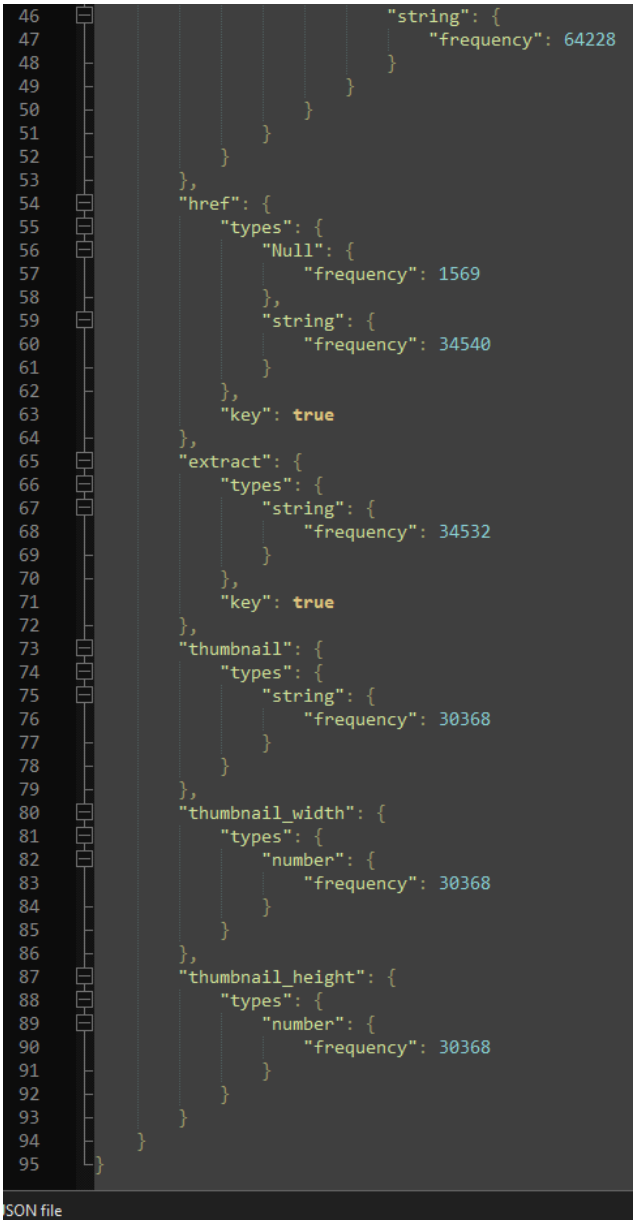


Figure 33. Extração do esquema do BD de filmes(2/2).

Os documentos extraídos foram então inseridos na ferramenta *document2sql* e seu mapeamento para o modelo relacional foi realizado, gerando o seguinte código DDL:

```
CREATE TABLE movies (
  movies_gen_uuid UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  title string, year float, href string, extract string,
  thumbnail string, thumbnail_width float, thumbnail_height float
);

CREATE TABLE movies__cast (
  movies__cast_gen_uuid UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  movies__caststring string
);
```

```
CREATE TABLE movies__genres (
 movies__genres_gen_uuid UUID
   PRIMARY KEY DEFAULT gen_random_uuid(),
 movies__genresstring string
);

CREATE TABLE movies__R__movies__cast (
 movies__R__movies__cast_gen_uuid UUID
   PRIMARY KEY DEFAULT gen_random_uuid(),
 movies_id UUID, movies__cast_id UUID,
 FOREIGN KEY (movies_id) REFERENCES movies,
 FOREIGN KEY (movies__cast_id) REFERENCES movies__cast
);

CREATE INDEX ON movies__R__movies__cast(movies_id);

CREATE INDEX ON movies__R__movies__cast(movies__cast_id);

CREATE TABLE movies__R__movies__genres (
 movies__R__movies__genres_gen_uuid UUID
  PRIMARY KEY DEFAULT gen_random_uuid(),
  movies_id UUID, movies__genres_id UUID,
  FOREIGN KEY (movies_id) REFERENCES movies,
  FOREIGN KEY (movies__genres_id) REFERENCES movies__genres);

CREATE INDEX ON movies__R__movies__genres(movies_id);
CREATE INDEX ON movies__R__movies__genres(movies__genres_id);
```

A partir de então, o código DDL foi executado no BD CockroachDB e os dados do BD de filmes no MongoDB foram migrados para uma estrutura tabular intermediária no BD H2 utilizando *scripts* de inserção gerados pelo ambiente de desenvolvimento *DataGrip*[3], para cada documento existente no BD. É possível identificar pela Figura 34 que os arrays estavam contidos dentro de campos de texto. Na sequência, rotinas SQL foram aplicadas sobre esta tabela intermediária de forma a separar e distribuir os dados dentro do esquema relacional gerado pela ferramenta *document2sql* no CockroachDB.

[3]https://www.jetbrains.com/pt-br/datagrip/

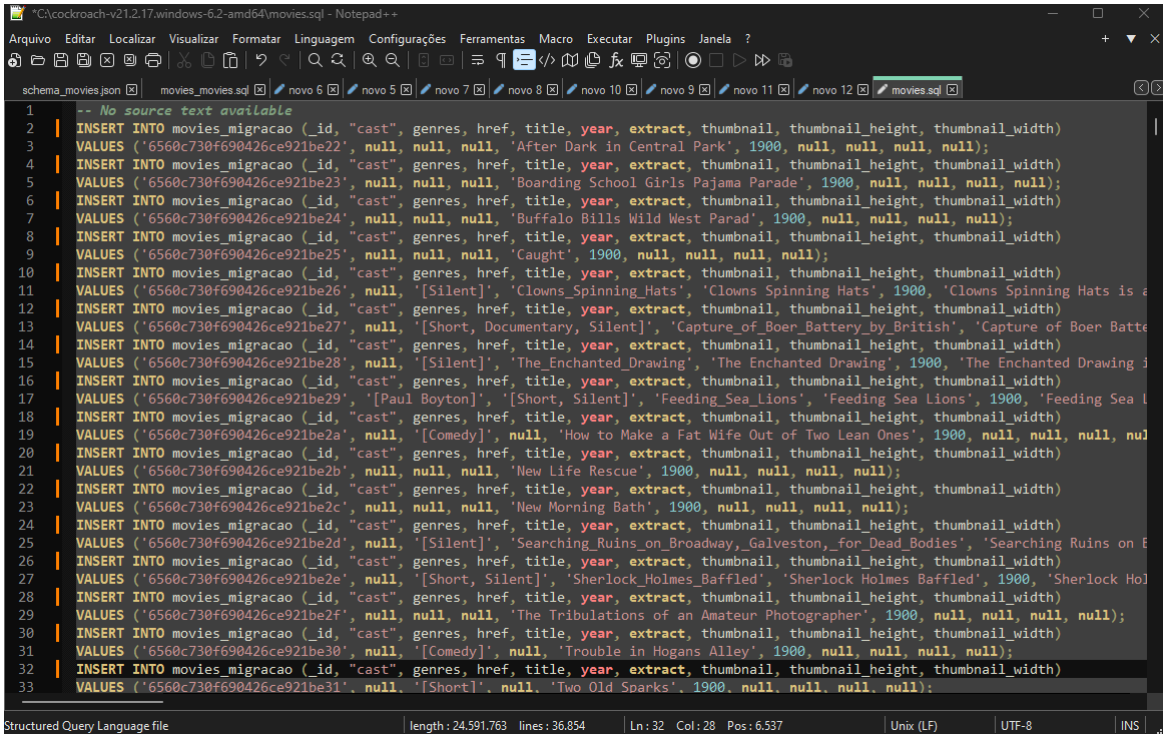


Figure 34. *Script* de inserção de dados do BD Filmes gerado pelo *DataGrip*.

Com a migração dos dados realizada, foram definidas consultas equivalentes para o MongoDB e o CockroachDB de forma a avaliar a compatibilidade dos esquemas de dados bem como o desempenho dos mesmos nas categorias de consultas comentadas anteriormente. Para a avaliação de desempenho cada categoria de consulta, executada com o auxílio de índices ou não, foi executada 10 vezes e sua média foi calculada. Os resultados das avaliações são detalhados a seguir.

6.1. Avaliação da Compatibilidade do Esquema de Dados Gerado

Esta avaliação tem por objetivo verificar se os resultados no MongoDB e no CockroachDB são compatíveis, quando utilizando junções, funções de agregação e operações condicionais. Para a execução desta consulta no CockroachDB foram utilizadas todas as tabelas geradas pelo mapeamento, de forma a validar se a estrutura gerada é compatível com a estrutura do MongoDB.

A seguinte consulta foi executada no MongoDB e o seu resultado pode ser visto na figura 35.

```
db.movies.aggregate([
    { $match: { year: 2023 } },
    { $unwind: "$genres" },
    { $unwind: "$cast" },
    { $group: {
        _id: "$genres",
        uniqueActors: { $addToSet: "$cast" }
    }},
    { $project: {
```

```
        _id: 0,
        genre: "$_id",
        numberOfActors: { $size: "$uniqueActors" }
    }},
    { $sort: { numberOfActors: -1 } }
]);
```

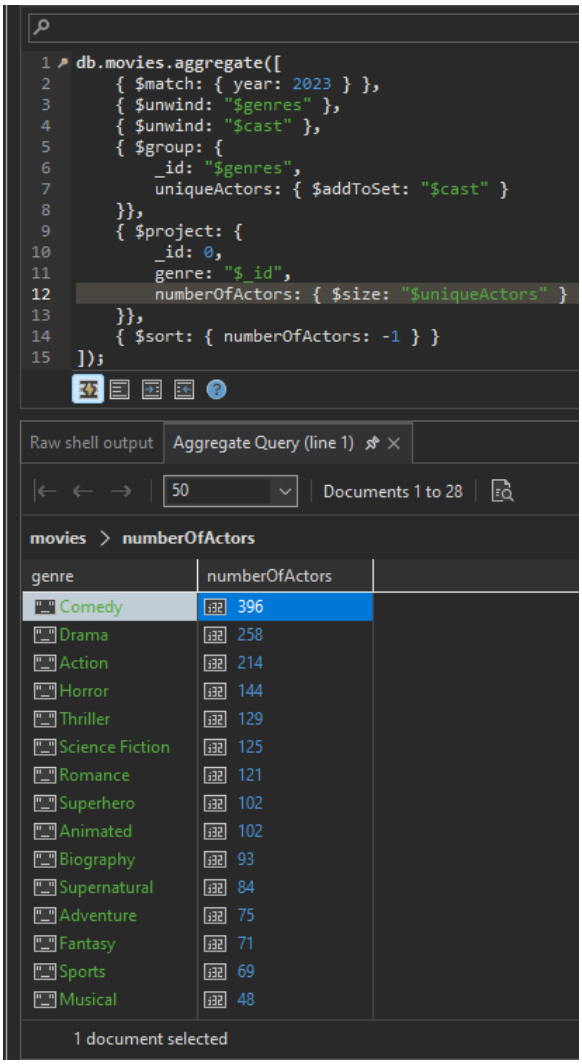
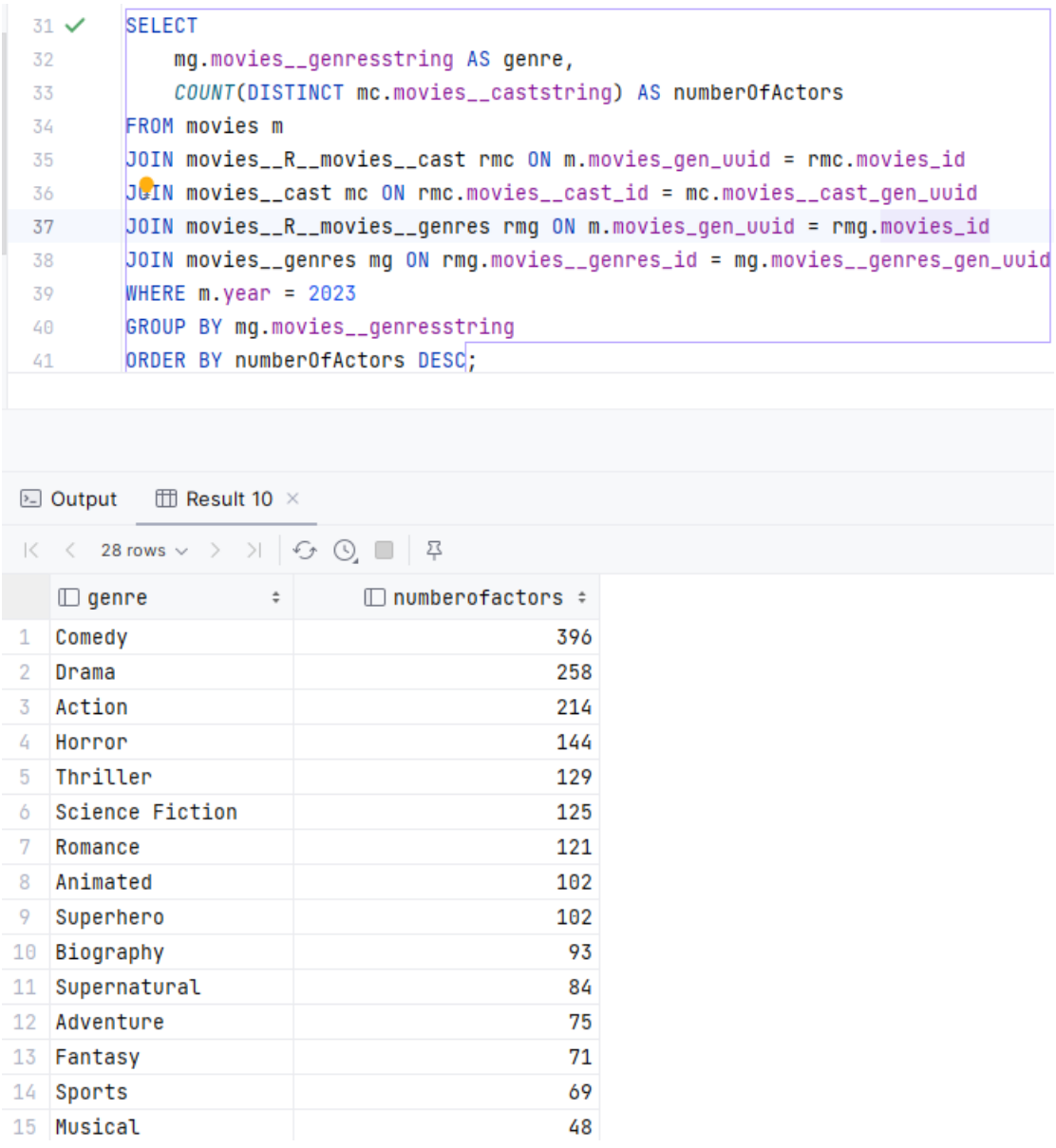


Figure 35. Resultado da consulta no MongoDB, para a avaliação de compatibilidade dos esquemas de dados.

A consulta equivalente a seguir foi executada no CockroachDB, sua execução e resultados podem ser vistos na figura 36.

```
SELECT
    mg.movies__genresstring AS genre,
    COUNT(DISTINCT mc.movies__caststring) AS numberOfActors
FROM movies m
JOIN movies__R__movies__cast rmc
```

```
  ON m.movies_gen_uuid = rmc.movies_id
JOIN movies__cast mc
  ON rmc.movies__cast_id = mc.movies__cast_gen_uuid
JOIN movies__R__movies__genres rmg
  ON m.movies_gen_uuid = rmg.movies_id
JOIN movies__genres mg
  ON rmg.movies__genres_id = mg.movies__genres_gen_uuid
WHERE m.year = 2023
GROUP BY mg.movies__genresstring
ORDER BY numberOfActors DESC;
```



```
SELECT
    mg.movies__genresstring AS genre,
    COUNT(DISTINCT mc.movies__caststring) AS numberOfActors
FROM movies m
JOIN movies__R__movies__cast rmc ON m.movies_gen_uuid = rmc.movies_id
JOIN movies__cast mc ON rmc.movies__cast_id = mc.movies__cast_gen_uuid
JOIN movies__R__movies__genres rmg ON m.movies_gen_uuid = rmg.movies_id
JOIN movies__genres mg ON rmg.movies__genres_id = mg.movies__genres_gen_uuid
WHERE m.year = 2023
GROUP BY mg.movies__genresstring
ORDER BY numberOfActors DESC;
```

Output | Result 10

28 rows

	genre	numberofactors
1	Comedy	396
2	Drama	258
3	Action	214
4	Horror	144
5	Thriller	129
6	Science Fiction	125
7	Romance	121
8	Animated	102
9	Superhero	102
10	Biography	93
11	Supernatural	84
12	Adventure	75
13	Fantasy	71
14	Sports	69
15	Musical	48

Figure 36. Resultado da consulta no CockroachDB, para a avaliação de compatibilidade dos esquemas de dados.

É possível visulizar através das figuras 35 e 36 que as consultas equivalentes em ambos os BDs geraram os mesmos resultados, justificando que o mapeamento realizado entre o modelo orientado a objetos e o modelo relacional gerou resultados compatíveis.

6.2. Avaliação de Desempenho

6.2.1. Consulta Simples

Essa categoria de consulta tem por finalidade avaliar o desempenho de uma busca sem a utilização de junções e funções de agregação. O código a seguir foi executado no MongoDB e teve um tempo médio de execução de 71,8 milissegundos.

```
db.movies.find({ "year": { $gt: 2000 } },
               { "title": 1, "year": 1 });
```

Adicionando índices nas colunas *title* e *year* o MongoDB atingiu um tempo médio de execução de 115,5 milissegundos, ou seja, houve uma queda em seu tempo de execução. Já no CockroachDB, a consulta equivalente (código a seguir) teve uma média de tempo de execução de 230,1 milissegundos.

```
SELECT title, year FROM movies WHERE year > 2000;
```

O CockroachDB se mostrou menos performático em consultas simples sem a inclusão de índices. Com a inclusão de índices nas colunas *year* e *title*, obteve-se uma média de execução de 116,1 milissegundos, equivalendo em desempenho ao MongoDB, conforme mostra a Tabela 2.

Table 2. Tempos de execução para consultas simples no CockroachDB e no MongoDB.

Tempo médio de execução	Sem índices	Com índices
CockroachDB	230,1 ms	116,1 ms
MongoDB	71,8 ms	115,5 ms

6.2.2. Consultas com Junções

Este experimento tem por objetivo avaliar o desempenho de consultas envolvendo junções. A seguinte consulta foi executada no BD MongoDB. Ela apresentou um tempo médio de execução de 144,5 milissegundos.

```
db.movies.find({ "year": { $gt: 2000 }, "cast": "Keanu Reeves" }
               , { "title": 1, "year": 1, "cast": 1 });
```

Na sequência, foram adicionados índices na chave *year* e no array *cast*, o que resultou em um tempo médio de processamento no MongoDB de 6,9 milissegundos, evidenciando a relevância de índices em arrays para consultas.

A consulta equivalente foi executada no BD CockroachDB, que obteve um tempo médio de execução de 167,5 milissegundos sem o uso de índices. Também foram inseridos índices nas colunas *year* e mc.*movies__caststring*. Estas alterações tornaram possível

atingir um tempo médio de execução de 144,3 milissegundos. Um resumo dos resultados é mostrado na Tabela 3.

```
SELECT m.title , m.year
FROM movies m
JOIN movies__R__movies__cast rmc
  ON m.movies_gen_uuid = rmc.movies_id
JOIN movies__cast mc
  ON rmc.movies__cast_id = mc.movies__cast_gen_uuid
WHERE mc.movies__caststring = 'Keanu■Reeves' AND m.year > 2000;
```

Table 3. Tempos de execução parabom consultas com junções no CockroachDB e no MongoDB.

Tempo médio de execução	Sem índices	Com índices
CockroachDB	167,5ms	144,3 ms
MongoDB	144,5ms	6,9 ms

Neste experimento percebe-se um melhor desempenho do MongoDB uma vez que os dados dos arrays estão agregados no documento, evitando junções. Já no BD relacional junções foram necessárias.

6.2.3. Consultas complexas com junções e agregações

Este experimento avalia o desempenho de uma consulta mais complexa envolvendo junções de múltiplas tabelas e agregações. A seguinte instrução foi executada no MongoDB, obtendo um tempo médio de 1956,7 milissegundos. Com a adição de índices nas chaves *year*, *cast* e *genres* no MongoDB obteve-se um tempo médio de 1296,7 milissegundos.

```
db.movies.aggregate([
    { $match: { "year": { $gte: 1937, $lte: 2015 } } },
    { $unwind: "$cast" },
    { $unwind: "$genres" },
    { $group: {
        _id: { cast: "$cast", genre: "$genres", movieId: "$_id" }
    }},
    { $group: {
        _id: { cast: "$_id.cast", genre: "$_id.genre" },
        total_movies: { $sum: 1 }
    }},
    { $match: { total_movies: { $gt: 1 } }},
    { $project: {
        cast: "$_id.cast",
        genre: "$_id.genre",
        total_movies: 1,
        _id: 0
```

```
    }}
]);
```

Já no CockroachDB a consulta equivalente obteve um tempo de médio de 1389,1 milissegundos. Ao adicionar índices nas colunas *year*, *movies_caststring* *movies_genrestring* obteve-se um tempo médio de 1266,4 milissegundos. A Tabela 4 resume os resultdos obtidos.

```
SELECT
    mc.movies__caststring,
    mg.movies__genresstring,
    COUNT(*) as total_movies
FROM
    (SELECT
        rmc.movies__cast_id,
        rmg.movies__genres_id,
        m.movies_gen_uuid
     FROM movies m
     JOIN movies__R__movies__cast rmc
       ON m.movies_gen_uuid = rmc.movies_id
     JOIN movies__R__movies__genres rmg
       ON m.movies_gen_uuid = rmg.movies_id
     WHERE
        m.year BETWEEN 1937 AND 2015) AS subquery
JOIN movies__cast mc
  ON subquery.movies__cast_id = mc.movies__cast_gen_uuid
JOIN movies__genres mg
  ON subquery.movies__genres_id = mg.movies__genres_gen_uuid
GROUP BY
    mc.movies__caststring,
    mg.movies__genresstring
HAVING
    COUNT(*) > 1;
```

Table 4. Tempos de execução alcançados pela consultas complexas baseadas em múltiplas junções e agregações no CockroachDB e no MongoDB.

Tempo médio de execução	Sem índices	Com índices
CockroachDB	1389,1 ms	1266,4 ms
MongoDB	1956,7 ms	1296,7 ms

Neste experimento percebe-se um melhor desempenho do CockroachDB para lidar com uma consulta complexa, evidenciando a melhor utilização de um BD relacional e uma vantagem ao migrar dados de um BD NoSQL orientado a documentos para uma tecnologia relacional.

6.3. Conclusão

Neste capítulo apresenta uma avaliação comparativa entre o MongoDB, um BD orientado a documentos, e o CockroachDB, um sistema de gerenciamento de BD relacional NewSQL. As avaliações foram realizadas utilizando um conjunto de consultas predefinidas, executadas em ambos os sistemas, para avaliar as diferenças no desempenho. O foco principal foi o impacto do uso de índices, junções e funções de agregação, além da avaliação da estrutura proposta pela ferramenta *document2sql*.

Os resultados mostraram que, para consultas simples houve um desempenho equilibrado entre as 2 tecnologias. Para consultas envolvendo junções, quando essa junções envolvem filtros sobre dados aninhados em um mesmo documento o melhor desempenho do MongoDB já era esperado, uma vez que os dados estão naturalmente agrupados em documento JSON. Em contrapartida, para consultas complexas, com múltiplas junções e operações de agregação, o CockroachDB obteve um melhor desempenho.

Esta avaliação experimental demonstra que tanto o MongoDB quanto o CockroachDB têm seus pontos fortes em diferentes tipos de operações de BD. O MongoDB se destaca em cenários que envolvem operações simples em documentos e dados aninhados, enquanto o CockroachDB demonstra vantagens em consultas relacionais mais complexas, especialmente quando otimizado com índices. Esses resultados são cruciais para determinar a escolha do sistema de BD, dependendo das necessidades específicas de desempenho para determinados tipos de operação. É possível concluir também que a estrutura gerada pela ferramenta *document2sql* atingiu seu objetivo em definir um mapeamento que mantenha as características dos dados mesmo migrados para o modelo de dados relacional, aproveitando seus benefícios no sentido de manter a integridade dos dados bem como um melhor desempenho em certos cenários que justifique o mapeamento NoSQL para SQL.

7. Considerações Finais

O presente trabalho teve como principal objetivo o desenvolvimento de uma ferramenta capaz de gerar o mapeamento estrutural de um BD orientado a documentos para um BD relacional a partir de regras de mapeamento predefinidas. A ferramenta desenvolvida *document2sql* se diferencia dos trabalhos relacionados pela sua capacidade de trabalhar com diferentes cardinalidades nas relações identificadas em conjuntos de dados.

A principal motivação deste trabalho é entregar uma ferramenta capaz de auxiliar na integração e migração de dados e observa-se que todos os objetivos definidos foram cumpridos. A aplicação é capaz de ler e analisar dados em um BD MongoDB de origem, identificar o esquema JSON deste BD de origem e definir a estrutura equivalente de um BD no modelo relacional levando em consideração as relações entre suas diferentes entidades. Diferentes cenários envolvendo diferentes cardinalidades foram testados com diferentes conjuntos de dados, demonstrando as regras de mapeamento.

Este trabalho também foi de grande relevância para o desenvolvimento teórico e técnico do autor, em grande parte à pesquisa realizada a respeito de tecnologias de BDs, principalmente os modelos orientado a documento e relacional.

A primeira versão do *document2sql* se encontra funcional, porém, diversos aprimoramentos podem ser realizados como trabalhos futuros:

- Uma interface gráfica que facilite a usabilidade do usuário e que seja capaz de gerar diagramas de tabela do mapeamento realizado;
- Adição de novos BDs de origem e de destino para fins de mapeamento;
- Possibilidade de realizar a migração dos dados do BD de origem para o BD destino de forma automática.

Todo o código gerado para o desenvolvimento deste trabalho se encontra em um repositório no *GitHub* que pode ser acessado através deste link: https://github.com/rafaelparola/document2sql.

References

Aftab, Z. (2020). Automatic nosql to relational database transformation with dynamic schema mapping.

D. Konstantinos, L. Peter, C. P. (2021). *Concise Guide to Databases*. Springer.

F. Martin, J. S. P. (2019). *NoSQL Essencial: um Guia Conciso Para o Mundo Emergente da Persistência Poliglota*. Novatec Editora.

M. Andreas, K. M. (2019). *SQL NoSQL Databases*. Springer.

Maity, B. (2018). A framework to convert nosql to relational model.

Michael, S. (2012). New opportunities for new sql. *Communications of the ACM*, 55:10–11.

Petković, D. (2020). Non-native techniques for storing json documents into relational tables.

Rebecca, T. (2020). Cockroachdb: The resilient geo-distributed sql database. *SIGMOD '20: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, page Pages 1493–1509.

Werner, A. and lgorzata Bach, M. (2018). Nosql e-learning laboratory—interactive querying of mongodb and couchdb and their conversion to a relational database.

B APÊNDICE - CÓDIGO FONTE

```
spring:
  datasource:
    url: jdbc:h2:file:./data/database
    username: sa
    password: password
    driverClassName: org.h2.Driver
  jpa:
    hibernate:
      ddl-auto: update
    database-platform: org.hibernate.dialect.H2Dialect
    show-sql: true
    properties:
      hibernate:
        dialect: org.hibernate.dialect.H2Dialect
  h2:
    console.enabled: true
  data:
    mongodb:
      uri:
          mongodb+srv://<<username>>:<<password>>@cluster0.aopulmp.mongodb.net/movies?retryWrites=true
```

```
package com.parola.document2sql.mapper.controller;

import com.fasterxml.jackson.databind.JsonNode;
import com.parola.document2sql.mapper.repository.SqlSchemaRepository;
import com.parola.document2sql.mapper.service.DynamicMongoService;
import com.parola.document2sql.mapper.service.JsonSchemaService;
import com.parola.document2sql.mapper.service.SqlSchemaService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Map;
import java.util.NoSuchElementException;

import com.parola.document2sql.mapper.repository.JsonSchemaRepository;
import com.parola.document2sql.mapper.entity.JsonSchema;

@RestController
@RequestMapping(path = "/JsonSchema", produces = "application/json",
    consumes = "application/json")
public class JsonSchemaController {

    @Autowired
    JsonSchemaService jsonSchemaService;
```

```
@Autowired
DynamicMongoService dynamicMongoService;

@PostMapping("/load")
public ResponseEntity<String> loadJsonSchema(@RequestBody JsonNode
    payload){
    // Persist the schema in the database

    System.out.println(payload.toString());
    System.out.println(payload.get("theaters"));
    jsonSchemaService.saveJsonSchema(payload);

    return ResponseEntity.status(HttpStatus.OK).body("The schema was
        successfully loaded");
}



@GetMapping("/all")
public ResponseEntity<List<JsonSchema>> getAllJsonSchemas() {
    return
        ResponseEntity.status(HttpStatus.OK).body(jsonSchemaService.getAllJsonSchemas());
}

@GetMapping("/map-to-sql/{schemaId}")
public ResponseEntity<String> mapToSql(@PathVariable long schemaId){

    try {
        jsonSchemaService.mapToSql(schemaId);
    } catch (NoSuchElementException e) {
        return
            ResponseEntity.status(HttpStatus.NOT_FOUND).body("There
            is no schema for the given id: " + schemaId + e);
    }
      catch (Exception e) {
        throw new RuntimeException(e);
    }

    return ResponseEntity.status(HttpStatus.OK).body("The mapping
        was succesfull");
}

@GetMapping("/create-relation")
public ResponseEntity<String> mapToSql(){

    try {
        //dynamicMongoService.analyzeCollectionCardinality("listingsAndReviews");
        //dynamicMongoService.analyzeCollectionCardinality("restaurants");
        dynamicMongoService.analyzeCollectionCardinality("companies");
    }
```

```
        catch (Exception e) {
            throw new RuntimeException(e);
        }

        return ResponseEntity.status(HttpStatus.OK).body("The mongodb
            connection was succesfull");
    }
}
```

```
package com.parola.document2sql.mapper.controller;

import com.parola.document2sql.mapper.service.SqlSchemaService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(path = "/SqlSchema", produces = "application/json",
    consumes = "application/json")
public class SqlSchemaController {

    @Autowired
    SqlSchemaService sqlSchemaService;

    @GetMapping("/sql-schema-ddl")
    public ResponseEntity<String> getSqlSchemaDdl(){

        String ddl;
        ddl = sqlSchemaService.getSqlSchemaDdl();

        return ResponseEntity.status(HttpStatus.OK).body(ddl);
    }
}
```

```
package com.parola.document2sql.mapper.entity;

import com.fasterxml.jackson.databind.JsonNode;
import jakarta.persistence.*;
import org.hibernate.annotations.Type;

import java.util.Map;

@Entity(name="JSON_SCHEMA")
public class JsonSchema {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    //@ElementCollection
    @Column(columnDefinition = "json")
    private String schema;

    public JsonSchema() {
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getSchema() {
        return schema;
    }

    public void setSchema(String schema) {
        this.schema = schema;
    }
}
```

```
package com.parola.document2sql.mapper.entity;

import jakarta.persistence.*;

@Entity(name = "SQL_COLUMN")
public class SqlColumn {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Column
    private String name;
    @Column
    private String dataType;
    @Column(columnDefinition = "boolean default true")
    private boolean isNullable = true;
    @Column
    private boolean isPk;
    @Column
    private boolean isFk;
    @Column
```

```
private boolean isUnique;

@ManyToOne
private SqlTable sqlTable;


public SqlColumn(String name, String dataType, boolean isPk, boolean
     isFk, boolean isUnique, boolean isNullable, SqlTable sqlTable) {
    this.name = name;
    this.dataType = dataType;
    this.isNullable = isNullable;
    this.sqlTable = sqlTable;
    this.isPk = isPk;
    this.isFk = isFk;
    this.isUnique = isUnique;
}

public SqlColumn() {
}

// Getters for name, dataType, and nullable
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getDataType() {
    return dataType;
}

public void setDataType(String dataType) {
    this.dataType = dataType;
}

public boolean isIsNullable() {
    return isNullable;
}

public void setIsNullable(boolean nullable) {
    this.isNullable = nullable;
}

public long getId() {
    return id;
}

public void setId(long id) {
```

```
        this.id = id;
    }

    public SqlTable getSqlTable() {
        return sqlTable;
    }

    public void setSqlTable(SqlTable sqlTable) {
        this.sqlTable = sqlTable;
    }

    public boolean isIsPk() {
        return isPk;
    }

    public void setIsPk(boolean pk) {
        this.isPk = pk;
    }

    public boolean isIsFk() {
        return isFk;
    }

    public void setIsFk(boolean fk) {
        this.isFk = fk;
    }

    public boolean isIsUnique() {
        return isUnique;
    }

    public void setIsUnique(boolean unique) {
        this.isUnique = unique;
    }
}
```

```
package com.parola.document2sql.mapper.entity;

import jakarta.persistence.*;
import org.springframework.data.annotation.Reference;

@Entity(name = "SQL_RELATION")
public class SqlRelation {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @ManyToOne(cascade=CascadeType.PERSIST)
```

```
    private SqlTable originTable;

    @ManyToOne(cascade=CascadeType.PERSIST)
    private SqlTable referencedTable;
    private String type; // e.g., "object" or "array"

    public SqlRelation(SqlTable referencedTable) {
        this.referencedTable = referencedTable;
    }

    public SqlRelation() {
    }

    // Getters for referencedTable and type

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public SqlTable getReferencedTable() {
        return referencedTable;
    }

    public void setReferencedTable(SqlTable referencedTable) {
        this.referencedTable = referencedTable;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public SqlTable getOriginTable() {
        return originTable;
    }

    public void setOriginTable(SqlTable originTable) {
        this.originTable = originTable;
    }
}
```

```
package com.parola.document2sql.mapper.entity;
```

```java
import jakarta.persistence.*;

import java.util.List;

@Entity(name = "SQL_SCHEMA")
public class SqlSchema {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @OneToOne
    @JoinColumn(name = "json_schema_id")
    private JsonSchema jsonSchema;


    public SqlSchema(){
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public JsonSchema getJsonSchema() {
        return jsonSchema;
    }

    public void setJsonSchema(JsonSchema jsonSchema) {
        this.jsonSchema = jsonSchema;
    }
}
```

```java
package com.parola.document2sql.mapper.entity;


import jakarta.persistence.*;
import org.hibernate.annotations.CreationTimestamp;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

@Entity(name="SQL_TABLE")
public class SqlTable {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Column
    private String name;

    @Column
    private int level;

    @CreationTimestamp
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_date_time")
    private Date createdDateAndTime;

    @OneToMany(cascade=CascadeType.ALL)
    private List<SqlColumn> columns;

    @ManyToOne
    private SqlSchema sqlSchema;

    public SqlTable(String name) {
        this.name = name;
        this.columns = new ArrayList<>();
/*
        this.relations = new ArrayList<>();
*/
    }

    public SqlTable(){

    }

    // Getters and setters for name, columns, and relations

    public void setColumn(SqlColumn column) {
        columns.add(column);
    }

/*
    public void setRelation(SqlRelation relation) {
        relations.add(relation);
    }
*/

    public List<SqlColumn> getColumns() {
        return columns;
    }

/*
```

```
    public List<SqlRelation> getRelations() {
        return relations;
    }
*/

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getCreatedDateAndTime() {
        return createdDateAndTime;
    }

    public void setCreatedDateAndTime(Date createdDateAndTime) {
        this.createdDateAndTime = createdDateAndTime;
    }

    public int getLevel() {
        return level;
    }

    public void setLevel(int level) {
        this.level = level;
    }
}
```

```
package com.parola.document2sql.mapper.repository;

import com.parola.document2sql.mapper.entity.JsonSchema;
import org.springframework.data.repository.ListCrudRepository;

public interface JsonSchemaRepository extends
    ListCrudRepository<JsonSchema,Long> {
}
```

```
package com.parola.document2sql.mapper.repository;
```

```
import com.parola.document2sql.mapper.entity.SqlColumn;
import org.springframework.data.repository.ListCrudRepository;

public interface SqlColumnRepository extends
    ListCrudRepository<SqlColumn,Long> {
}
```

```
package com.parola.document2sql.mapper.repository;

import com.parola.document2sql.mapper.entity.SqlRelation;
import org.springframework.data.repository.ListCrudRepository;

public interface SqlRelationRepository extends
    ListCrudRepository<SqlRelation,Long> {
}
```

```
package com.parola.document2sql.mapper.repository;

import com.parola.document2sql.mapper.entity.SqlSchema;
import org.springframework.data.repository.ListCrudRepository;

public interface SqlSchemaRepository extends
    ListCrudRepository<SqlSchema,Long> {
}
```

```
package com.parola.document2sql.mapper.repository;

import com.parola.document2sql.mapper.entity.SqlTable;
import org.springframework.data.repository.ListCrudRepository;

import java.util.List;

public interface SqlTableRepository extends
    ListCrudRepository<SqlTable,Long> {
   List<SqlTable> findByLevel(int level);

   SqlTable findByName(String name);
}
```

```
package com.parola.document2sql.mapper.service;

import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoDatabase;
import com.parola.document2sql.mapper.entity.SqlColumn;
import com.parola.document2sql.mapper.entity.SqlRelation;
```

```
import com.parola.document2sql.mapper.entity.SqlTable;
import com.parola.document2sql.mapper.repository.SqlColumnRepository;
import com.parola.document2sql.mapper.repository.SqlRelationRepository;
import com.parola.document2sql.mapper.repository.SqlTableRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Service;
import com.mongodb.client.MongoCollection;
import org.bson.Document;

import java.util.*;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.stream.Collectors;

@Service
public class DynamicMongoService {

    @Autowired
    private MongoTemplate mongoTemplate;

    @Autowired
    private SqlRelationRepository sqlRelationRepository;

    @Autowired
    private SqlTableRepository sqlTableRepository;

    @Autowired
    private SqlColumnRepository sqlColumnRepository;



    //SqlRelation sqlRelation;

    public void analyzeCollectionCardinality(String collectionName) {
        MongoDatabase database = mongoTemplate.getDb();
        MongoCollection<Document> collection =
            database.getCollection(collectionName);

        // For structures without identifiable unique keys, using string
            representation as a "structure hash"
        Map<String, Map<String, Integer>>
            subDocumentStructureOccurrences = new HashMap<>();
        // For arrays, tracking their "structure hash"
        Map<String, Map<String, Integer>> arrayStructureOccurrences =
            new HashMap<>();
        // For parents, tracking their "structure hash" and it's childs
        Map<String, Set<String>> parentDocumentOccurrences = new
            HashMap<>();
        // For parent per full key
        Map<String, Map<String, Integer>>
```

```
    parentDocumentOccurrencesPerFullKey = new HashMap<>();


FindIterable<Document> documents = collection.find();
for (Document document : documents) {
    analyzeDocument(document, subDocumentStructureOccurrences,
        arrayStructureOccurrences, parentDocumentOccurrences,
        parentDocumentOccurrencesPerFullKey, null);
}


AtomicBoolean isManyToMany = new AtomicBoolean(false);
AtomicBoolean isManyToOne = new AtomicBoolean(false);
AtomicBoolean isOneToMany = new AtomicBoolean(false);
AtomicBoolean isOneToOne = new AtomicBoolean(false);

subDocumentStructureOccurrences.forEach((field, structureMap) ->
    {

    for (String structureHash : structureMap.keySet()) {
        Set<String> parentDocs =
            parentDocumentOccurrences.get(structureHash);
        System.out.println(structureHash);
        System.out.println(parentDocs);
        System.out.println(parentDocumentOccurrencesPerFullKey.get(field));
        if (parentDocs != null && parentDocs.size() > 1
                && structureMap.values().stream().anyMatch(count
                    -> count > 1)
                &&
                    parentDocumentOccurrencesPerFullKey.get(field).values().stream().anyMatch(co
                    -> count > 1)
        ) {
            isManyToMany.set(true);
            break;
        }
        else if (parentDocs != null && parentDocs.size() > 1
                    &&
                        structureMap.values().stream().anyMatch(count
                        -> count > 1)
                    &&
                        parentDocumentOccurrencesPerFullKey.get(field).values().stream().allMatc
                        -> count == 1)
                ) {
            isManyToOne.set(true);
            break;
        }
        else if ((parentDocs == null || parentDocs.size() == 1)
                && structureMap.values().stream().allMatch(count
                    -> count == 1)
                &&
```

```
                parentDocumentOccurrencesPerFullKey.get(field).values().stream().anyMatch(co
                -> count > 1)) {
            isOneToMany.set(true);
            break;
        }
        else if ((parentDocs == null || parentDocs.size() == 1)
            && structureMap.values().stream().noneMatch(count ->
            count > 1)) {
            isOneToOne.set(true);
        }
    }

    if (isManyToMany.get()) {
        System.out.println("Many-to-Many");
    }
    if (isManyToOne.get()) {
        System.out.println("Many-to-One");
    }
    if (isOneToMany.get()) {
        System.out.println("One-to-Many");
    }
    if (isOneToOne.get()) {
        System.out.println("One-to-One");
    }



    /*if(isManyToMany.get()) {

    }*/
    if (isManyToOne.get()) {
        String parentTableName = "";
        String childTableName = "";

        System.out.println("Sub-document field '" + field + "'
            has a N-1 relationship.");

        if(field.contains(".")) {
            childTableName = collectionName+"__" +
                field.substring(0,
                field.lastIndexOf(".")).replace(".","__");
            parentTableName =
                collectionName+"__"+field.replace(".","__");
        } else {
            childTableName = collectionName;
            parentTableName =
                collectionName+"__"+field.replace(".","__");
        }

        SqlTable parentTable =
```

```
            sqlTableRepository.findByName(parentTableName);
        SqlTable childTable =
            sqlTableRepository.findByName(childTableName);

        SqlRelation sqlRelation = new SqlRelation();
        sqlRelation.setOriginTable(childTable);
        sqlRelation.setReferencedTable(parentTable);
        sqlRelation.setType("N-1");

        SqlColumn foreignKey = new SqlColumn();
        foreignKey.setName(parentTable.getName()+"_id");
        foreignKey.setIsFk(true);
        foreignKey.setDataType("UUID");
        foreignKey.setSqlTable(childTable);
        childTable.setColumn(foreignKey);

        sqlRelationRepository.save(sqlRelation);
        sqlTableRepository.save(childTable);
    }
    else if (isManyToMany.get()) {
        String parentTableName = "";
        String childTableName = "";

        System.out.println("Array field '" + field + "' appears
            to have an M-N relationship across documents. ou seja
            M-N");
        if(field.contains(".")) {
            parentTableName = collectionName+"__" +
                field.substring(0,
                field.lastIndexOf(".")).replace(".","__");
            childTableName =
                collectionName+"__"+field.replace(".","__");
        } else {
            parentTableName = collectionName;
            childTableName =
                collectionName+"__"+field.replace(".","__");
        }
        SqlTable sqlTable = new
            SqlTable(parentTableName+"__R__"+childTableName);

        // Creates the table primary key
        SqlColumn primaryKey = new SqlColumn();
        primaryKey.setName(sqlTable.getName()+"_gen_uuid");
        primaryKey.setIsPk(true);
        primaryKey.setDataType("UUID");
        primaryKey.setSqlTable(sqlTable);
        sqlTable.setColumn(primaryKey);

        SqlTable parentTable =
            sqlTableRepository.findByName(parentTableName);
```

```
    SqlTable childTable =
        sqlTableRepository.findByName(childTableName);

    SqlRelation sqlRelationParent = new SqlRelation();
    sqlRelationParent.setOriginTable(sqlTable);
    sqlRelationParent.setReferencedTable(parentTable);
    sqlRelationParent.setType("M-N");

    SqlRelation sqlRelationChild = new SqlRelation();
    sqlRelationChild.setOriginTable(sqlTable);
    sqlRelationChild.setReferencedTable(childTable);
    sqlRelationChild.setType("M-N");

    SqlColumn foreignKeyOriginTable = new SqlColumn();
    foreignKeyOriginTable.setName(parentTable.getName()+"_id");
    foreignKeyOriginTable.setIsFk(true);
    foreignKeyOriginTable.setDataType("UUID");
    foreignKeyOriginTable.setSqlTable(sqlTable);
    sqlTable.setColumn(foreignKeyOriginTable);

    SqlColumn foreignKeyReferencedTable = new SqlColumn();
    foreignKeyReferencedTable.setName(childTable.getName()+"_id");
    foreignKeyReferencedTable.setIsFk(true);
    foreignKeyReferencedTable.setDataType("UUID");
    foreignKeyReferencedTable.setSqlTable(sqlTable);
    sqlTable.setColumn(foreignKeyReferencedTable);

    sqlRelationRepository.save(sqlRelationParent);
    sqlRelationRepository.save(sqlRelationChild);
}

else if (isOneToMany.get()) {
    String parentTableName = "";
    String childTableName = "";

    System.out.println("Sub-document field '" + field + "'
        has a 1-N relationship.");

    if(field.contains(".")) {
        parentTableName = collectionName+"__" +
            field.substring(0,
            field.lastIndexOf(".")).replace(".","__");
        childTableName =
            collectionName+"__"+field.replace(".","__");
    } else {
        parentTableName = collectionName;
        childTableName =
            collectionName+"__"+field.replace(".","__");
    }
```

```
        SqlTable parentTable =
            sqlTableRepository.findByName(parentTableName);
        SqlTable childTable =
            sqlTableRepository.findByName(childTableName);

        SqlRelation sqlRelation = new SqlRelation();
        sqlRelation.setOriginTable(childTable);
        sqlRelation.setReferencedTable(parentTable);
        sqlRelation.setType("1-N");

        SqlColumn foreignKey = new SqlColumn();
        foreignKey.setName(parentTable.getName()+"_id");
        foreignKey.setIsFk(true);
        foreignKey.setDataType("UUID");
        foreignKey.setSqlTable(childTable);
        childTable.setColumn(foreignKey);

        sqlRelationRepository.save(sqlRelation);
        sqlTableRepository.save(childTable);

    } else if (isOneToOne.get()) {
        String parentTableName = "";
        String childTableName = "";
        System.out.println("Sub-document field '" + field + "'
            has a 1-1 relationship.");

        if(field.contains(".")) {
            parentTableName = collectionName+"__" +
                field.substring(0,
                field.lastIndexOf(".")).replace(".","__").replace("
                ", "_");
            childTableName =
                collectionName+"__"+field.replace(".","__").replace("
                ", "_");
        } else {
            parentTableName = collectionName.replace(" ", "_");
            childTableName =
                collectionName+"__"+field.replace(".","__").replace("
                ", "_");
            System.out.println(parentTableName);
            System.out.println(childTableName);
        }
        SqlTable parentTable =
            sqlTableRepository.findByName(parentTableName);
        SqlTable childTable =
            sqlTableRepository.findByName(childTableName);

        SqlRelation sqlRelation = new SqlRelation();
        sqlRelation.setOriginTable(childTable);
        sqlRelation.setReferencedTable(parentTable);
```

```
        sqlRelation.setType("1-1");

        //SqlColumn foreignKey = new SqlColumn();
        List<SqlColumn> columns = childTable.getColumns();
        for (SqlColumn column : columns) {
            if (column.isIsPk()) {
                column.setIsFk(true);
                sqlColumnRepository.save(column);
            }
        }
        sqlRelationRepository.save(sqlRelation);
        sqlTableRepository.save(childTable);

    }
});

arrayStructureOccurrences.forEach((field, structureMap) -> {
    String parentTableName = "";
    String childTableName = "";
    if (structureMap.values().stream().anyMatch(count -> count >
         1)) {
        /*for(int i = 0; i < structureMap.size(); i++) {
            System.out.println(structureMap);
        }*/
        //System.out.println(structureMap.values());
        System.out.println("Array field '" + field + "' appears
             to have an M-N relationship across documents. ou seja
             M-N");
        if(field.contains(".")) {
            parentTableName = collectionName+"__" +
                 field.substring(0,
                 field.lastIndexOf(".")).replace(".","__");
            childTableName =
                 collectionName+"__"+field.replace(".","__");
        } else {
            parentTableName = collectionName;
            childTableName =
                 collectionName+"__"+field.replace(".","__");
        }
        SqlTable sqlTable = new
             SqlTable(parentTableName+"__R__"+childTableName);

        // Creates the table primary key
        SqlColumn primaryKey = new SqlColumn();
        primaryKey.setName(sqlTable.getName()+"_gen_uuid");
        primaryKey.setIsPk(true);
        primaryKey.setDataType("UUID");
        primaryKey.setSqlTable(sqlTable);
        sqlTable.setColumn(primaryKey);
```

```
    SqlTable parentTable =
        sqlTableRepository.findByName(parentTableName);
    SqlTable childTable =
        sqlTableRepository.findByName(childTableName);

    SqlRelation sqlRelationParent = new SqlRelation();
    sqlRelationParent.setOriginTable(sqlTable);
    sqlRelationParent.setReferencedTable(parentTable);
    sqlRelationParent.setType("M-N");

    SqlRelation sqlRelationChild = new SqlRelation();
    sqlRelationChild.setOriginTable(sqlTable);
    sqlRelationChild.setReferencedTable(childTable);
    sqlRelationChild.setType("M-N");

    SqlColumn foreignKeyOriginTable = new SqlColumn();
    foreignKeyOriginTable.setName(parentTable.getName()+"_id");
    foreignKeyOriginTable.setIsFk(true);
    foreignKeyOriginTable.setDataType("UUID");
    foreignKeyOriginTable.setSqlTable(sqlTable);
    sqlTable.setColumn(foreignKeyOriginTable);

    SqlColumn foreignKeyReferencedTable = new SqlColumn();
    foreignKeyReferencedTable.setName(childTable.getName()+"_id");
    foreignKeyReferencedTable.setIsFk(true);
    foreignKeyReferencedTable.setDataType("UUID");
    foreignKeyReferencedTable.setSqlTable(sqlTable);
    sqlTable.setColumn(foreignKeyReferencedTable);

    sqlRelationRepository.save(sqlRelationParent);
    sqlRelationRepository.save(sqlRelationChild);


} else {
    System.out.println("Array field '" + field + "' does not
        appear to have an M-N relationship across documents.
        Ou seja 1-N");
    if(field.contains(".")) {
        parentTableName = collectionName+"__" +
            field.substring(0,
            field.lastIndexOf(".")).replace(".","__");
        childTableName =
            collectionName+"__"+field.replace(".","__");
    } else {
        parentTableName = collectionName;
        childTableName =
            collectionName+"__"+field.replace(".","__");
    }

    SqlTable parentTable =
```

```
                sqlTableRepository.findByName(parentTableName);
            SqlTable childTable =
                sqlTableRepository.findByName(childTableName);

            SqlRelation sqlRelation = new SqlRelation();
            sqlRelation.setOriginTable(childTable);
            sqlRelation.setReferencedTable(parentTable);
            sqlRelation.setType("1-N");

            SqlColumn foreignKey = new SqlColumn();
            foreignKey.setName(parentTable.getName()+"_id");
            foreignKey.setIsFk(true);
            foreignKey.setDataType("UUID");
            foreignKey.setSqlTable(childTable);
            childTable.setColumn(foreignKey);

            sqlRelationRepository.save(sqlRelation);
            sqlTableRepository.save(childTable);
        }
    });
}

private void analyzeDocument(Document doc,
                             Map<String, Map<String, Integer>>
                                 subDocumentStructureOccurrences,
                             Map<String, Map<String, Integer>>
                                 arrayStructureOccurrences,
                             Map<String, Set<String>>
                                 parentDocumentOccurrences,
                             Map<String, Map<String, Integer>>
                                 parentDocumentOccurrencesPerFullKey,
                             String parentKey) {
    doc.forEach((key, value) -> {
        // Construct a composite key to represent the field's full
            path
        String fullKey = (parentKey == null) ? key : parentKey + "."
            + key;

        // Cdigo novo
        String documentHash = generateTopLevelHash(doc);

        if (value instanceof Document) {
            // Generate a hash for the top-level document fields only
            String structureHash = generateTopLevelHash((Document)
                value);
            if (structureHash != null) {
                subDocumentStructureOccurrences.computeIfAbsent(fullKey,
                    k -> new HashMap<>())
                        .merge(structureHash, 1, Integer::sum);
```

```
            parentDocumentOccurrences.computeIfAbsent(structureHash,
                k -> new HashSet<>())
                    .add(documentHash);

            parentDocumentOccurrencesPerFullKey.computeIfAbsent(fullKey,
                k -> new HashMap<>())
                    .merge(documentHash, 1, Integer::sum);
        }


        analyzeDocument((Document) value,
            subDocumentStructureOccurrences,
            arrayStructureOccurrences, parentDocumentOccurrences,
            parentDocumentOccurrencesPerFullKey, fullKey);
    } else if (value instanceof List) {
        // Handle the list of items, considering only top-level
            items
        ((List<?>) value).stream().distinct().forEach(item -> {
            // Generate a simple hash based on the item's
                toString, for immediate items only
            if (!(item instanceof Document) && !(item instanceof
                List)) {
                //String structureHash = "Not M-N";
                String structureHash = generateArrayHash((List<?>)
                    value);
                arrayStructureOccurrences.computeIfAbsent(fullKey,
                    k -> new HashMap<>())
                        //.merge(structureHash, 0, Integer::sum);
                        .merge(structureHash, 1, Integer::sum);
                return;
            }

            if (item instanceof List) {
                String structureHash = generateArrayHash((List<?>)
                    value);

                if (structureHash != null) {
                    arrayStructureOccurrences.computeIfAbsent(fullKey,
                        k -> new HashMap<>())
                            .merge(structureHash, 1, Integer::sum);
                }
            }

            if (item instanceof Document) {
                String structureHashObject =
                    generateTopLevelHash((Document) item);


                if(structureHashObject != null) {
                    arrayStructureOccurrences.computeIfAbsent(fullKey,
```

```
                        k -> new HashMap<>())
                            .merge(structureHashObject, 1,
                                Integer::sum);
                }


                analyzeDocument((Document) item,
                    subDocumentStructureOccurrences,
                    arrayStructureOccurrences,
                    parentDocumentOccurrences,
                    parentDocumentOccurrencesPerFullKey, fullKey);
            }
        });
    }
});
}

private String generateTopLevelHash(Document document) {
    // This will create a concatenated string of the top-level
        fields and their values,
    // excluding values that are Documents, Lists (arrays), or null
    String ret = document.entrySet().stream()
            .filter(entry -> !(entry.getValue() instanceof Document)
                    && !(entry.getValue() instanceof List)
                    && entry.getValue() != null
                    && !"_id".equals(entry.getKey()))
            .map(entry -> entry.getKey() + "=" + entry.getValue())
            .collect(Collectors.joining(","));
    if(ret.trim().isEmpty()) {
        ret = null;
    }
    return ret;
}

private String generateArrayHash(List<?> array) {
    // This will create a concatenated string of all non-null,
        non-Document elements in the array
    String ret = array.stream()
            .filter(item -> !(item instanceof Document)
                    && !(item instanceof List)
                    && item != null)
            .map(Object::toString)
            .collect(Collectors.joining(","));
    if(ret.trim().isEmpty()) {
        ret = null;
    }
    if(ret != null) {
        //System.out.println(ret);
    }
```

```
        return ret;
    }
}
```

```
package com.parola.document2sql.mapper.service;

import java.io.IOException;
import java.util.*;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.JsonNode;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.parola.document2sql.mapper.entity.*;
import com.parola.document2sql.mapper.repository.JsonSchemaRepository;
import com.parola.document2sql.mapper.repository.SqlColumnRepository;
import com.parola.document2sql.mapper.repository.SqlRelationRepository;
import com.parola.document2sql.mapper.repository.SqlTableRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class JsonSchemaService {

    @Autowired
    private JsonSchemaRepository jsonSchemaRepository;

    @Autowired
    private SqlTableRepository sqlTableRepository;

    @Autowired
    private SqlColumnRepository sqlColumnRepository;

    @Autowired
    private SqlRelationRepository sqlRelationRepository;

    @Autowired
    SqlSchemaService sqlSchemaService;

    @Autowired
    private ObjectMapper objectMapper;

    @Autowired
    DynamicMongoService dynamicMongoService;

    public static final String TYPES = "types";
    public static final String PRIMARY_KEY = "primaryKey";
    public static final String KEY = "key";
    public static final String FOREIGN_KEY = "foreignKey";
```

```
public static final String REFERENCES = "references";
public static final String OBJECT = "Object";
public static final String STRUCTURE = "structure";
public static final String ARRAY = "Array";

public void saveJsonSchema(JsonNode payload) {
    // Generates a JsonSchema Object and populates the schema
        attribute with the mongo db schema input
    JsonSchema jsonSchema = new JsonSchema();
    jsonSchema.setSchema(payload.toString());

    // Persist the schema in the database
    jsonSchemaRepository.save(jsonSchema);
}

public List<JsonSchema> getAllJsonSchemas() {
    return jsonSchemaRepository.findAll();
}

public JsonSchema getJsonSchemaById(long jsonSchemaId) {
    // Get the JsonSchema by ID
    Optional<JsonSchema> jsonSchemaOptional =
        jsonSchemaRepository.findById(jsonSchemaId);

    if (jsonSchemaOptional.isPresent()) {
        return jsonSchemaOptional.get();
    } else {
        throw new NoSuchElementException("JSON schema not found with
            ID: " + jsonSchemaId);
    }
}

public JsonNode getSchemaJsonNodeInJsonSchema(JsonSchema jsonSchema)
    {
    String jsonString = jsonSchema.getSchema();
    try {
        // Parse the JSON string into a JsonNode using Spring Boot's
            ObjectMapper

        JsonNode jsonNode =
            objectMapper.readTree(jsonString.substring( 1,
            jsonString.length() - 1 ).replace("\\", "") );
        //JsonNode jsonNode = objectMapper.readTree(jsonString);

        return jsonNode;
    } catch (IOException e) {
        // Handle any exceptions that may occur during JSON parsing
        e.printStackTrace();
    }
    return null;
```

```
}

public void mapToSql(long jsonSchemaId) throws
    JsonProcessingException {
    JsonSchema jsonSchema = this.getJsonSchemaById(jsonSchemaId);
    JsonNode node = this.getSchemaJsonNodeInJsonSchema(jsonSchema);

    for (Iterator<String> it = node.fieldNames(); it.hasNext(); ) {
        String tableName = it.next().replace(" ", "_");
        SqlTable sqlTable = new SqlTable(tableName);
        sqlTable.setCreatedDateAndTime(new Date());
        JsonNode tableNode = node.get(tableName);
        this.createSqlObjects(tableNode, sqlTable);

        dynamicMongoService.analyzeCollectionCardinality(tableName);
    }
}
public void createSqlObjectsArray(JsonNode schema, SqlTable
    parentTable) {
    // Creates the table primary key
    SqlColumn primaryKey = new SqlColumn();
    primaryKey.setName(parentTable.getName()+"_gen_uuid");
    primaryKey.setIsPk(true);
    primaryKey.setDataType("UUID");
    primaryKey.setSqlTable(parentTable);
    parentTable.setColumn(primaryKey);

    for (Iterator<String> it = schema.fieldNames(); it.hasNext(); ) {
            String type = it.next();

        if(type != OBJECT && type != ARRAY && type != "Null") {
            SqlColumn column = new SqlColumn();

            column.setName(parentTable.getName().replace(" ", "_") +
                type);
            if(type == "number"){
                column.setDataType("float");
            } else {
                column.setDataType(type);
            }
            column.setSqlTable(parentTable);

            // Set the column object in the parent table
            parentTable.setColumn(column);
        }
        if(type == OBJECT) {
            JsonNode structure = this.getArrayObjectStructure(schema);

            this.createSqlObjects(structure, parentTable);
        }
```

```
    }
    sqlTableRepository.save(parentTable);
}

public void createSqlObjects(JsonNode schema, SqlTable parentTable) {
    // Creates the table primary key

    if
        (!parentTable.getColumns().stream().anyMatch(SqlColumn::isIsPk))
        {
        SqlColumn primaryKey = new SqlColumn();
        primaryKey.setName(parentTable.getName()+"_gen_uuid");
        primaryKey.setIsPk(true);
        primaryKey.setDataType("UUID");
        primaryKey.setSqlTable(parentTable);
        parentTable.setColumn(primaryKey);
    }

    for (Iterator<String> it = schema.fieldNames(); it.hasNext(); ) {

        // Get the actual Object
        String fieldName = it.next();
        JsonNode fieldNameNode = schema.get(fieldName);
        if (fieldName != "_id") {
            // Get the JSON node which contains the types of the
                given fieldName Object
            JsonNode nodeType = getNodeObjectType(fieldNameNode);
            // Get the first type in string (Change later)
            String type = getStringObjectType(nodeType);

            if (type != OBJECT && type != ARRAY && type != "Null") {

                // Creates the column object
                SqlColumn column = new SqlColumn();

                for (Iterator<String> attribute =
                    fieldNameNode.fieldNames(); attribute.hasNext();)
                    {
                    String attributeName = attribute.next().replace("
                        ", "_");
                    if (attributeName == KEY) {
                        column.setIsUnique(true);
                        column.setIsNullable(false);
                    }
                }

                column.setName(fieldName.replace(" ", "_"));
                if(type == "number"){
                    column.setDataType("float");
```

```
                } else {
                    column.setDataType(type);
                }
                column.setSqlTable(parentTable);
                // Populates the table object with the column object
                parentTable.setColumn(column);

            } else if (type == OBJECT) {
                // Creates child table
                SqlTable childTable = new
                    SqlTable(parentTable.getName() +"__"+
                    fieldName.replace(" ", "_"));
                childTable.setCreatedDateAndTime(new Date());

                // Get the childNodeStructure
                JsonNode childNodeStructure =
                    this.getObjectStructure(fieldNameNode);
                this.createSqlObjects(childNodeStructure, childTable);
            } else if (type == ARRAY) {
                SqlTable childArrayTable = new
                    SqlTable(parentTable.getName()
                    +"__"+fieldName.replace(" ", "_"));

                // Gets Array Structure
                JsonNode arrayStructure =
                    getArrayStructure(fieldNameNode);

                // Creates sqlTable from Array
                this.createSqlObjectsArray(arrayStructure,
                    childArrayTable);

            }
            sqlTableRepository.save(parentTable);

        }
    }
}

//  No  esquecer que existem fields com mais de um tipo ex numero e
    string
public JsonNode getNodeObjectType(JsonNode node) {
    return node.get(TYPES);
}

public String getStringObjectType(JsonNode node) {
    for (Iterator<String> it = node.fieldNames(); it.hasNext();) {
        String fieldName = it.next(); // Store the next field name
        if (!"Null".equals(fieldName)) { // Correct string comparison
            return fieldName; // Return the stored field name
        }
```

```
        }
        // If there are no field names or all field names are "Null",
            return null or some default value
        return null;
    }

    public void createRelations() {

    }

    public JsonNode getObjectStructure(JsonNode node) {
        return node.get(TYPES).get(OBJECT).get(STRUCTURE);
    }

    public JsonNode getArrayStructure(JsonNode node) {
        return node.get(TYPES).get(ARRAY).get(STRUCTURE).get(TYPES);
    }

    public JsonNode getArrayObjectStructure(JsonNode node) {
        return node.get(OBJECT).get(STRUCTURE);
    }

}
```

```
package com.parola.document2sql.mapper.service;

import com.parola.document2sql.mapper.entity.*;
import com.parola.document2sql.mapper.repository.SqlColumnRepository;
import com.parola.document2sql.mapper.repository.SqlRelationRepository;
import com.parola.document2sql.mapper.repository.SqlSchemaRepository;
import com.parola.document2sql.mapper.repository.SqlTableRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.*;

@Service
public class SqlSchemaService {

    @Autowired
    private SqlSchemaRepository sqlSchemaRepository;

    @Autowired
    private SqlTableRepository sqlTableRepository;

    @Autowired
    private SqlColumnRepository sqlColumnRepository;

    @Autowired
```

```
private SqlRelationRepository sqlRelationRepository;

public void saveSqlSchema(JsonSchema jsonSchema) {
    SqlSchema sqlSchema = new SqlSchema();
    sqlSchema.setJsonSchema(jsonSchema);

    sqlSchemaRepository.save(sqlSchema);
}
public String getSqlSchemaDdl() {
    List<SqlTable> tables = sqlTableRepository.findAll();
    Map<Long, List<SqlRelation>> tableRelations =
        mapTableRelations();

    StringBuilder ddl = new StringBuilder();

    for (SqlTable table : tables) {
        ddl.append(createTableDdl(table,
            tableRelations.get(table.getId()))).append("\n");
    }

    return ddl.toString();
}

private Map<Long, List<SqlRelation>> mapTableRelations() {
    Map<Long, List<SqlRelation>> relationsMap = new HashMap<>();
    List<SqlRelation> relations = sqlRelationRepository.findAll();

    for (SqlRelation relation : relations) {
        relationsMap.computeIfAbsent(relation.getOriginTable().getId(),
            k -> new ArrayList<>()).add(relation);
    }

    return relationsMap;
}

private String createTableDdl(SqlTable table, List<SqlRelation>
    relations) {
    StringBuilder ddl = new StringBuilder();
    StringBuilder ddlIndex = new StringBuilder();

    String foreignKey = "";
    List<String> uniqueColumns = new ArrayList<>();

    ddl.append("CREATE TABLE ").append(table.getName()).append(" (");

    for (SqlColumn column : table.getColumns()) {
        ddl.append(column.getName()).append(" ")
                .append(column.getDataType());

        if (column.isIsPk()) {
```

```
            ddl.append(" PRIMARY KEY DEFAULT gen_random_uuid()");
        }

        /*if (!column.isIsNullable()) {
            ddl.append(" NOT NULL");
        }*/

        if(column.isIsFk()) {
            foreignKey = column.getName();
        }

        if (column.isIsUnique()) {
            uniqueColumns.add(column.getName());
        }

        ddl.append(", ");
    }

    // Append Unique Constraints
    if (uniqueColumns.size() > 0){
        for (String column : uniqueColumns) {
            ddl.append("UNIQUE (").append(column).append(") ,");
        }
    }

    // Append foreign key constraints
    if (relations != null) {
        for (SqlRelation relation : relations) {
            if (!Objects.equals(relation.getType(), "1-1")){
                SqlTable referencedTable =
                    relation.getReferencedTable();
                SqlTable originTable = relation.getOriginTable();
                ddl.append("FOREIGN KEY
                    (").append(referencedTable.getName()).append("_id)
                    ")
                       .append("REFERENCES
                           ").append(referencedTable.getName()).append(",
                           ");
                // Creates index for foreign key
                ddlIndex.append("\n");
                ddlIndex.append("CREATE INDEX ON
                    ").append(originTable.getName()).append("(").append(referencedTable.getName()).a

            } else {
                SqlTable referencedTable =
                    relation.getReferencedTable();
                ddl.append("FOREIGN KEY
                    (").append(foreignKey).append(") ")
                       .append("REFERENCES
                           ").append(referencedTable.getName()).append(",
```

```
                            ");
                }
            }
        }

        // Remove the trailing comma and space
        int lastIndex = ddl.lastIndexOf(", ");
        if (lastIndex >= 0) {
            ddl.delete(lastIndex, ddl.length());
        }

        ddl.append(");");
        ddl.append(ddlIndex);
        return ddl.toString();
    }
}
```